MARCIN SZELIGA

PRAKTYCZNE UCZENIE MASZYNOWE



PWN

MARCIN SZELIGA

PRAKTYCZNE UCZENIE MASZYNOWE

PWN

Projekt okładki i stron tytułowych **Joanna Andryjowicz**

Wydawca **Edyta Kawala**

Redaktor prowadzący **Jolanta Kowalczuk**

Redaktor **Irena Puchalska**

Koordynator produkcji **Anna Bączkowska**

Recenzent dr hab. Tadeusz Wieczorek, Politechnika Śląska

Skład wersji elektronicznej na zlecenie Wydawnictwa Naukowego PWN
Karol Ossowski/Woblink

Zastrzeżonych nazw firm i produktów użyto w książce wyłącznie w celu identyfikacji.

Książka, którą nabyłeś, jest dziełem twórcy i wydawcy. Prosimy, abyś przestrzegał praw, jakie im przysługują. Jej zawartość możesz udostępnić nieodpłatnie osobom bliskim lub osobiście znanym. Ale nie publikuj jej w internecie. Jeśli cytujesz jej fragmenty, nie zmieniaj ich treści i koniecznie zaznacz, czyje to dzieło. A kopiując jej część, rób to jedynie na użytek osobisty.

Szanujmy cudzą własność i prawo.
Więcej na www.legalnakultura.pl
Polska Izba Książki

Copyright © by Wydawnictwo Naukowe PWN SA
Warszawa 2019

ISBN 978-83-01-20784-7

eBook został przygotowany na podstawie wydania papierowego z 2019 r., (wyd. I)

Warszawa 2019

Wydawnictwo Naukowe PWN SA
02-460 Warszawa, ul. Gottlieba Daimlera 2
tel. 22 69 54 321, faks 22 69 54 288
infolinia 801 33 33 88
e-mail: pwn@pwn.com.pl; reklama@pwn.pl
www.pwn.pl

Patroni medialni:

















Dziękuję dr Tomaszowi Cieplakowi i Andrzejowi Kukule za cenne uwagi i poprawki. Szczególne podziękowania należą się profesorowi Tadeuszowi Wieczorkowi za recenzję książki.

Wszystkie błędy w książce są moje.

Marcin Szeliga

Celem autora było napisanie praktycznego przewodnika łączącego teorię z praktyką. Na rynku jest niewiele tytułów polskojęzycznych, które reprezentują takie podejście do problemu, ilustrując opisywane zagadnienia praktycznymi przykładami. Jest to na pewno najpełniejsze ujęcie problematyki uczenia maszynowego z praktycznego punktu widzenia, z jakim spotkał się recenzent. Opisano najważniejsze problemy uczenia maszynowego na kilkunastu przykładach. Dobór przykładów i ich układ jest dobrze uzasadniony, gdyż omawianie każdego przykładu jest okazją do wyjaśnienia określonych zagadnień związanych z uczeniem maszynowym.

Na podkreślenie również zasługuje, i jest to naprawdę rzadkie, że książka została poszerzona o wiele przykładów możliwych do analizy wraz z danymi wejściowymi. W tekście książki zostały opisane tylko wybrane fragmenty kodu użytego do ilustracji omawianych zagadnień. Kompletną wersję przykładów razem z użytymi danymi można pobrać ze strony Wydawnictwa, a ostatnią wersję przykładów z serwisu Github.

***

Książka napisana jest dobrze, poprawnym językiem i w sposób przejrzysty. Zaproponowany zestaw problemów obejmuje całość tematyki, którą można scharakteryzować jako przewodnik po stosowanych obecnie na świecie systemach uczenia maszynowego, z wykorzystaniem określonego środowiska obliczeniowego i prezentacyjnego (MS SQL Server i język R, język Phyton).

Dlatego w konkluzji stwierdzam, że proponowana książka autorstwa Marcina Szeligi, *Praktyczne uczenie maszynowe*, jest ważnym i wartościowym podręcznikiem informatyki stosowanej. Będzie on przydatny zarówno dla studentów kierunków informatycznych, zarządzania i marketingu, ekonomii i wielu kierunków ścisłych, jak i dla osób zawodowo pracujących w tych dziedzinach.

prof. zw. dr hab. Tadeusz Wieczorek

Kierownik Katedry Informatyki Przemysłowej Politechniki Śląskiej

Dane to nowa ropa.

Clive Humby, Data Scientist w Starcount, 2006

Sztuczna inteligencja to nowa energia elektryczna.

Andrew Ng, Data Scientist w Baidu, założyciel Coursera, profesor na uniwersytecie Stanforda, 2016

Od autora

W ciągu ostatnich 50 lat wydano setki prac poświęconych uczeniu maszynowemu. W większości z nich skoncentrowano się na zagadnieniach teoretycznych, takich jak statystyka, rachunek prawdopodobieństwa oraz budowa i zasada działania algorytmów uczenia maszynowego. Zawarte w tych książkach przykłady albo są opisowe, albo zawierają bardzo proste, w praktyce nieużywane fragmenty kodu. Pozostałe to pozycje czysto praktyczne, przedstawiające sposób budowania konkretnych modeli uczenia maszynowego, w których nie znajdziemy wyjaśnienia działania zastosowanych algorytmów i wynikających z tych zasad ogólniejszych wskazówek.

Książek łączących teorię z praktyką jest niewiele, szczególnie polskojęzycznych. Mam swój pewien wkład w ich liczbę – najpierw napisałem prace o eksploracji danych z użyciem usług analitycznych serwera SQL [1], [2], a następnie o uczeniu maszynowym w chmurze [3]. Zdobywane w krajowych i międzynarodowych projektach doświadczenie, a także rosnąca dostępność coraz łatwiejszych w użyciu i jednocześnie coraz bardziej funkcjonalnych narzędzi uczenia maszynowego pozwoliły mi napisać tę książkę.

Opisałem w niej rozwiązania kilkunastu typowych problemów, takich jak prognozowanie zysków, optymalizacja kampanii marketingowej, proaktywna

konserwacja sprzętu czy oceny ryzyka kredytowego. Ich układ jest celowy – każdy przykład jest okazją do wyjaśnienia określonych zagadnień, zaczynając od narzędzi, przez podstawy uczenia maszynowego, sposoby oceny jakości danych i ich przygotowania do dalszej analizy, zasady tworzenia modeli uczenia maszynowego i ich optymalizacji, po wskazówki dotyczące wdrożenia gotowych modeli do produkcji. Wszystkie przykłady są prowadzone według powszechnie stosowanej metodyki CRISP-DM (*CRoss Industry Standard Process for Data Mining*) [4].

Chociaż *termin sztuczna* inteligencja nie występuje w tytule, na wstępie wyjaśniam, czym jest sztuczna inteligencja, jak doszło do sytuacji, w której sztuczna inteligencja zmienia nasz świat, i jaki jest jej związek z uczeniem maszynowym.

Marcin Szeliga

Microsoft Most Valuable Professional
w kategorii Artificial Intelligence,
pracownik naukowy Wyższej Szkoły Bankowej
w Poznaniu, Wydział Zamiejscowy w Chorzowie

Spis treści

Od autora

Sztuczna inteligencja

O książce

Dla kogo jest ta książka?

Narzędzia

Przykłady

Bibliografia

Konwencje i oznaczenia

1. Narzędzia

1.1. Język Python

1.2. Język R

1.2.1. Microsoft R Open (MRO)

1.2.2. Microsoft R Client (MRC)

1.2.3. Microsoft Machine Learning Server (MLS)

1.3. SQL Server 2019

1.3.1. Instalacja

1.3.2. Microsoft SQL Server Machine Learning Services

1.3.3. Bezpieczeństwo

1.3.4. Wydajność

1.4. PyCharm Community Edition

1.5. RStudio Desktop

1.6. Instalacja dodatkowych pakietów

1.7. Power BI Desktop

2. Praca z SQL Server Machine Learning Services

2.1. Wykrywanie oszustw

2.2. Klasyfikacja przejazdów

2.3. Dodatkowe funkcjonalności serwera SQL Server i usługi SQL Machine Learning Services
2.3.1. Zapytania predykcyjne czasu rzeczywistego
2.3.2. Natywne zapytania predykcyjne
2.3.3. Tworzenie modeli dla partycji danych

3. Wstęp do uczenia maszynowego

3.1. Rodzaje uczenia maszynowego
3.2. Proces uczenia
3.3. Modele regresji
3.4. Modele partycjonujące
3.5. Metodyka CRISP-DM
3.6. Metodyka TDSP

4. Zrozumienie danych

4.1. Poznanie danych wymaga zrozumienia postawionego problemu
4.2. Statystyki opisowe
4.2.1. Zmienne numeryczne
4.2.2. Zmienne kategoryczne
4.3. Brakujące dane
4.4. Entropia
4.5. Ocena zmiennych za pomocą programu Power BI
4.6. Ocena zmiennych przy użyciu języka SQL
4.6.1. Automatyczny opis zmiennych
4.7. Wizualizacja zmiennych
4.8. Reprezentatywność danych
4.9. Korelacje między zmiennymi
4.9.1. Klątwa wymiarowości
4.9.2. Ocena przydatności zmiennych
4.9.3. Dwie zmienne numeryczne
4.9.4. Dwie zmienne kategoryczne

4.9.5. Dwie zmienne porządkowe
4.9.6. Zmienna kategoryczna i numeryczna
4.9.7. Korelacja oznacza współwystępowanie, a nie związek przyczynowo-skutkowy
4.10. Ocena korelacji za pomocą programu Power BI
4.11. Ocena korelacji przy użyciu języka SQL
5. Przygotowanie danych
5.1. Uporządkowanie danych
5.2. Wzbogacanie danych
5.2.1. Data i czas
5.3. Wyczyszczenie danych
5.3.1. Usuwanie brakujących wartości
5.3.2. Usuwanie duplikatów
5.3.3. Usuwanie błędnych danych
5.3.4. Usuwanie wartości odstających
5.4. Przekształcenie danych
5.4.1. Kodowanie
5.4.2. Generalizacja
5.4.3. Zaokrąglanie
5.4.4. Dyskretyzacja
5.4.5. Skalowanie
5.4.6. Wygładzanie
5.5. Redukcja danych
5.5.1. Selekcja zmiennych
5.5.2. Analiza składowych głównych
5.5.3. Wybór zmiennych przydatnych dla modelu
5.6. Podział danych
5.6.1. Podział warstwowy
5.6.2. Równoważenie danych

5.6.3. k-krotna walidacja krzyżowa
5.7. Danych walidacyjnych używa się do optymalizacji, a nie do oceny modeli
5.8. Kto miał szansę przeżyć katastrofę Titanica?

6. Analiza skupień

6.1. Grupowanie w celu zmniejszenia liczby przykładów
6.2. Algorytmy iteracyjno-optymalizacyjne
6.3. Segmentacja

7. Regresja

7.1. Zrozumienie problemu
7.2. Zrozumienie danych
7.3. Opisanie danych
7.4. Ocena przydatności danych
7.5. Wzbogacenie danych
7.6. Ocena zmiennych
7.7. Przekształcenie i wybór danych
7.8. Modelowanie
7.8.1. Uczenie na błędach
7.8.2. Regresja liniowa
7.8.3. Ogólny model liniowy GLM
7.8.4. Sztuczne sieci neuronowe
7.8.5. Drzewa regresyjne
7.8.6. Kombinacje modeli
7.9. Porównanie modeli
7.10. Wdrożenie modeli po stronie serwera SQL

8. Klasyfikacja

8.1. Klasyfikacja binarna
8.1.1. Przygotowanie danych
8.1.2. Regresja logistyczna
8.1.3. Sztuczne sieci neuronowe

8.1.4. Klasyfikacja przez indukcję drzew decyzyjnych
8.1.5. Kombinacje modeli
8.1.6. Porównanie modeli
8.1.7. Wdrożenie modeli po stronie serwera SQL
8.2. Klasyfikacja wieloklasowa
8.2.1. Przygotowanie danych
8.2.2. Regresja logistyczna
8.2.3. Sztuczne sieci neuronowe
8.2.4. Klasyfikacja przez indukcję drzew decyzyjnych
8.2.5. Porównanie modeli
8.2.6. Wdrożenie modeli po stronie serwera SQL
9. Ocena modeli
9.1. Nie ma darmowego lunchu
9.2. Błędy modeli predykcyjnych
9.2.1. Błąd systematyczny i błąd aproksymacji
9.3. Kryteria oceny modeli regresji
9.3.1. Średni błąd bezwzględny
9.3.2. Pierwiastek błędu średniokwadratowego
9.3.3. Znormalizowany błąd bezwzględny
9.3.4. Znormalizowany błąd kwadratowy
9.3.5. Współczynnik determinacji R2
9.3.6. Ocena modelu prognozującego pozostały czas bezawaryjnej pracy urządzeń
9.4. Kryteria oceny modeli klasyfikacji binarnej
9.4.1. Macierz błędów
9.4.2. Trafność
9.4.3. Precyzja
9.4.4. Czułość
9.4.5. F-miara

9.4.6. Współczynnik Kappa Cohena
9.4.7. Krzywa ROC i obszar pod krzywą
9.4.8. Ocena modelu klasyfikującego urządzenia jako wymagające lub niewymagające przeglądu
9.5. Kryteria oceny modeli klasyfikacji wieloklasowej
9.5.1. Macierz błędów
9.5.2. Metryki klasy większościowej
9.5.3. Metryki poszczególnych klas
9.5.4. Średnie makro
9.5.5. Średnie mikro
9.6. Ocena modelu klasyfikującego urządzenia do przeglądu
9.7. Interpretacja predykcji
10. Optymalizacja i wdrożenie modeli
10.1. Zrozumienie problemu
10.2. Zrozumienie i przygotowanie danych
10.2.1. Import danych
10.2.2. Ocena danych
10.3. Modelowanie
10.4. Optymalizacja
10.5. Wdrożenie
10.5.1. Analiza typu Co by było, gdyby?
Posłowie
Bibliografia
Przypisy

Sztuczna inteligencja

Ilość cyfrowych danych generowanych każdego dnia jest niewyobrażalna – szacuje się ją na 3 tryliony bajtów i liczba ta z dnia na dzień rośnie. Dla porównania, mózg człowieka składa się z około 90 miliardów neuronów. Oznacza to, że codziennie zapisujemy cyfrowo dane równe mózgom 330 mln ludzi reprezentowanych sumą ich neuronów.

Szybkość, z jaką przeszliśmy od mierzenia wielkości danych w kilobajtach (10^3) do megabajtów (10^6), gigabajtów (10^9), terabajtów (10^{12}) i jottabajtów(10^{24}), jest niewiarygodna. Każdego roku generujemy więcej danych niż we wszystkich poprzednich latach od początku historii naszego gatunku. Ten wzrost zawdzięczamy przede wszystkim urządzeniom mobilnym i wszechobecnym, podłączonym do sieci czujnikom.

Jednak dane jako takie są bezwartościowe. To zawarte w nich informacje są cenne. **Algorytmy uczenia maszynowego automatycznie wykrywają wzorce ukryte w danych i zapisują je w postaci modeli**. Tak nauczone modele są mózgami inteligentnych maszyn.

Dostępne w ogromnych ilościach dane uzupełnione o techniczne możliwości ich przetwarzania doprowadziły do bezprecedensowego rozwoju sztucznej inteligencji (ang. *artificial* intelligence, AI) – nie tylko do przełomowych badań nad algorytmami uczenia maszynowego, lecz także do coraz powszechniejszych zastosowań inteligentnych maszyn w najróżniejszych dziedzinach naszego życia. Od 2012 r. rynek AI – mierzony wartością globalnych inwestycji oraz wartością kontraktów – podwaja się co dwa lata: wartość ujawnionych inwestycji w 2012 r. wyniosła 590 mln USD przy sumarycznej wartości kontraktów

wynoszącej 160 mln USD; w 2017 r. inwestycje przekroczyły 9 miliardów USD, a wartość kontraktów – 900 mln USD.

Historia zna kilka przykładów tak głębokich zmian cywilizacyjnych i nazywa je rewolucjami przemysłowymi. Wszystkie dotychczasowe rewolucje przemysłowe były wywołane odkryciem nowej technologii[1]:

- pierwszą zawdzięczamy zastosowaniu pary wodnej w przemyśle, przede wszystkim w transporcie kolejowym (XVIII w.);
- drugą – użyciu na szeroką skalę elektryczności (początek XX w.);
- trzecią (nazwaną naukowo-techniczną) – integracji przemysłu z informatyką, co przyniosło cyfryzację procesów produkcyjnych (lata 70. XX w.).

Dzisiaj jesteśmy świadkami czwartej rewolucji przemysłowej, którą zawdzięczamy stosowaniu na coraz szerszą skalę sztucznej inteligencji i która, tak jak wszystkie wcześniejsze rewolucje przemysłowe, całkowicie zmienia świat, w którym żyjemy.

Czym jest sztuczna inteligencja? Inteligencję przypisuje się ludziom oraz zwierzętom, jednak nawet użyty wyłącznie w odniesieniu do ludzi termin ten jest niejednoznaczny. Psychologowie definiują inteligencję m.in. jako zdolność rozwiązywania problemów (Jean Piaget), umiejętność dostrzegania zależności (Charles Spearman) lub jako zdolność uczenia się (George Ferguson). Różne są też poglądy na temat charakteru inteligencji. Gdyby za jej cechę zasadniczą uznać zdolność do twórczego, a nie tylko mechanicznego przetwarzania informacji, za inteligentnych należałoby uznać tylko niektórych ludzi [39]. Jednocześnie, jeżeli przyjmiemy, że inteligencja polega na zdolności do przetwarzania informacji na poziomie abstrakcyjnych idei, za obdarzonych inteligencją można uznać nie tylko ludzi, lecz także zwierzęta i maszyny.

Przyjęcie takiej definicji inteligencji pozwala określić sztuczną inteligencję jako dziedzinę zajmującą się tworzeniem sztucznych, obdarzonych inteligencją maszyn. Prekursor AI, Alan M. Turing, w swojej pionierskiej, wydanej

w 1950 r. pracy *Computing Machinery and Intelligence* sformułował test, który miał odpowiedzieć na pytanie: Kiedy maszyna zachowuje się w sposób inteligentny? [5]. Według Turinga maszynę można uznać za inteligentną, gdy człowiek tester nie jest w stanie odróżnić odpowiedzi maszyny od odpowiedzi człowieka. Pomysł Turinga zmienił podejście do badań nad sztuczną inteligencją – **filozoficzne pytanie o istotę inteligencji zostało zastąpione pragmatyczną oceną skutków bycia inteligentnym**.

Aby zdać pełny test Turinga, maszyna powinna mieć zdolność do przetwarzania języka naturalnego, uczenia się i stosowania zdobytej wiedzy, wnioskowania oraz do rozpoznawania obrazów i poruszania się, w tym do przemieszczania obiektów – jak do tej pory żadnej maszynie się to nie udało. Sam test jest zresztą niesformalizowany i od dawna nie jest stosowany w praktyce.

Narodziny sztucznej inteligencji jako dyscypliny naukowej datuje się na 1956 r. Uczestnicy warsztatu, w którym uczestniczył też twórca terminu *sztuczna inteligencja* – John McCarthy, sformułowali wtedy hipotezę, według której każdy aspekt uczenia się, a także każda inna własność inteligencji, mogą być opisane tak precyzyjnie, że możliwe będzie zbudowanie maszyny zdolnej do ich symulacji [6]. Efektem przyjęcia tej hipotezy są nadal prowadzone badania nad opracowanymi wtedy pierwszymi wersjami algorytmów uczenia maszynowego, takich jak perceptron czy drzewa decyzyjne.

Rozczarowanie wynikające m.in. z niespełnienia nadmiernie optymistycznych zapowiedzi przyszło w 1974 r. Nie tylko systemy sztucznej inteligencji nie znalazły praktycznych zastosowań (czego powodem była niewystarczająca moc obliczeniowa ówczesnych komputerów), lecz pojawiły się także poważne teoretyczne wątpliwości co do możliwości rozwiązywania przez nie różnych zadań, bez konieczności dostosowywania danych treningowych do konkretnych problemów.

Drugą falę rozwoju badań nad sztuczną inteligencją, przypadającą na lata

1980–1987, zawdzięczamy nadziejom pokładanym w systemach eksperckich. System ekspercki jest systemem, który automatycznie uczy się rozwiązywać problemy na podstawie podanych mu reguł. Początkowo do działania wymagały one specjalistycznych i drogich komputerów, co przyczyniło się do kolejnego kryzysu. Udało się go przezwyciężyć w 1993 r., demonstrując praktyczne zastosowania systemów eksperckich działających już na standardowych komputerach, choć wielokrotnie szybszych od typowych serwerów z tego czasu. Największym sukcesem systemów eksperckich było pokonanie w 1997 r. przez system Deep Blue (superkomputer ze specjalnie zaprojektowanymi układami scalonymi VLSI do realizacji algorytmu przeszukiwania alfa-beta) ówczesnego szachowego mistrza świata Garriego Kasparova. W rewanżowym meczu szachowym komputer wygrał z nim 3½ do 2½ (pierwszy, rozegrany rok wcześniej mecz Kasparov wygrał z poprzednią wersją komputera Deep Blue 4 do 2).

Uczenie komputerów szczegółowych reguł postępowania okazało się jednak niesłychanie skomplikowane. Gra w szachy rządzi się niewielką liczbą dobrze ustalonych i niezmiennych reguł, więc Deep Blue, analizując 200 mln pozycji na sekundę, był w stanie sprawdzić wszystkie możliwe kombinacje kilkunastu kolejnych ruchów. Ale takie podejście nie pozwalało rozwiązywać problemów rządzących się bardziej skomplikowanymi, a tym bardziej nieznanymi regułami.

W tym samym roku, gdy Deep Blue wygrał z Kasparovem, Tom Mitchell wydał książkę zatytułowaną *Machine Learning*, w której podał definicję uczenia maszynowego: **Program komputerowy uczy się z doświadczenia E w odniesieniu do pewnej klasy zadań T i miary wydajności P, jeśli jego skuteczność w zadaniach w T, mierzona wydajnością P, poprawia się wraz z doświadczeniem E** [7]. Szybko okazało się, że podejście polegające na tym, że to komputer sam ma znaleźć sposób rozwiązywania problemów na podstawie danych historycznych, pozwala tworzyć systemy sztucznej inteligencji mające praktyczne zastosowanie w wielu dziedzinach życia. Rozpoczęła się trzecia fala

rozwoju AI – etap uczenia maszynowego. Wiele używanych dzisiaj systemów sztucznej inteligencji należy do tej właśnie, opisanej w książce, kategorii.

Algorytmy uczenia maszynowego trenuje się na danych historycznych, czyli na przykładach. Podczas uczenia algorytm wykrywa ukryte w tych przykładach wzorce i na ich podstawie tworzy model uczenia maszynowego. Taki model może być następnie używany do predykcji, czyli uzupełniania brakujących danych o nowych, nieznajdujących się w treningowym zbiorze danych obserwacji. Na przykład model nauczony na danych sprzedażowych może przewidzieć przyszłą wartość sprzedaży, model nauczony na historiach chorób pacjentów może oceniać postępy leczenia nowych pacjentów, a model nauczony na danych opisujących działanie sieci wodociągowej – przewidywać zużycie wody, wykrywać próby kradzieży albo znajdować bieżące i przyszłe awarie wodociągów.

Skoro uczenie maszynowe polega na wykrywaniu ukrytych w przykładach treningowych wzorców, to im większą ilością danych dysponujemy, tym lepsze osiągniemy rezultaty. Jednak wraz ze wzrostem ilości danych treningowych rosną również wymagania obliczeniowe. Podam prosty przykład – zdjęcie zrobione w rozdzielczości 800 × 600 zawiera prawie 1,5 mln liczb reprezentujących wartości RGB określających kolory poszczególnych pikseli, a jest to rozdzielczość mniejsza niż typowa dla współczesnych zdjęć. Problem wydajności obliczeń rozwiązały akceleratory graficzne ogólnego zastosowania (ang. *general-purpose computing on graphics processing units*, GPGPU) [40]. Przewagą GPU nad CPU jest ich równoległa architektura pozwalająca na wykonywanie w ciągu sekundy tysięcy razy większej liczby prostych operacji matematycznych, takich jak mnożenie macierzy. Trzecim, równie ważnym czynnikiem, który przyczynił się do kolejnego rozkwitu badań nad sztuczną inteligencją, było opracowanie nowych algorytmów pozwalających maszynom efektywnie uczyć się nie tylko na podstawie danych treningowych, lecz przede wszystkim na podstawie samodzielnie wyodrębnionych z tych danych

abstrakcyjnych właściwości. **Rozpoczęła się trwająca do dziś era głębokiego uczenia maszynowego** (ang. *deep learning*, DL)**, w której sztuczna inteligencja uczy się na podstawie automatycznie wykrywanych abstrakcyjnych cech**, z wykorzystaniem takich algorytmów uczenia maszynowego jak konwolucyjne sieci neuronowe i rekurencyjne sieci neuronowe [41].

Na tym etapie sztuczna inteligencja szybko dogoniła, a następnie przegoniła ludzi w wybranych dziedzinach. Na przykład boty potrafią rozmawiać na zadane tematy, takie jak rezerwacja samolotu czy hotelu, a systemy rozpoznawania obrazów klasyfikować przedmioty widoczne na zdjęciach czy nagraniach wideo. Cechą charakterystyczną systemów głębokiego uczenia maszynowego jest zdolność do rozwiązywania problemów, których pokonanie ludziom przychodzi stosunkowo łatwo, natomiast maszyny wcześniej nie potrafiły sobie z nimi poradzić.

Maszyny nauczyły się przede wszystkim słyszeć i rozumieć zdania wypowiadane w takich językach jak angielski czy chiński. Już w 2012 r. procent popełnianych przez nie błędów (błędnie zrozumianych wypowiedzi) spadł poniżej 4%. W tym czasie maszyny potrafiły oprócz tego mówić, popełniając mniej błędów niż statystyczny człowiek. Pięć lat później maszyny nauczyły się tłumaczyć (na język chiński, a następnie angielski) jak ludzie.

Postęp w dziedzinie rozpoznawania obrazów również był imponujący. W 2012 r. system uczenia maszynowego prawidłowo rozpoznał 74% pokazanych mu przedmiotów. W następnym roku było to już 85% (ten wynik zawdzięczamy rozwojowi konwolucyjnych sieci neuronowych), a trzy lata później głęboka sieć neuronowa osiągnęła wynik 97% – o trzy procent lepszy od wyniku typowego człowieka.

Szybkość, z jaką AI jest w stanie nauczyć się nowych umiejętności, najlepiej pokazuje historia systemu AlphaGo. Ludzie doskonalą umiejętność gry w Go od

stuleci. Tymczasem w 2015 r. AlphaGo wygrał 3:0 z trzykrotnym mistrzem Europy w Go Fan Hui. Rozegrany rok później w Seulu mecz (który obejrzało ponad 200 mln widzów) z Lee Sedolem, powszechnie uznawanym za najwybitniejszego gracza w Go ostatniego dziesięciolecia, AlphaGo wygrał 4 do 1. Co więcej, grał tak innowacyjnie, że w znaczący sposób wzbogacił gromadzoną przez stulecia wiedzę na temat tej gry. W lutym 2017 r. AlphaGo Master, udoskonalona wersja AlphaGo, wygrała 60 meczów pod rząd przeciwko światowej czołówce graczy Go, co pozwala uznać ją za najlepszego gracza Go w historii.

Systemy AlphaGo uczyły się gry w Go, analizując tysiące partii rozegranych przez zawodowych i amatorskich graczy. Zupełnie inaczej uczył się system AlphaGo Zero. W tym przypadku zastosowano metodę uczenia ze wzmacnianiem – komputer grał sam ze sobą, ucząc się na własnych błędach. Po trzech dniach takiej nauki przekroczył on poziom systemu AlphaGo z 2015 r., kiedy to AlphaGo wygrał z Fan Hui. 21 dni nauki wystarczyły do osiągnięcia poziomu AlphaGo Master z 2017 r., kiedy to system ten wygrał 60 meczów z najlepszymi graczami Go, w tym mistrzem Ke Jie. Po 40 dniach treningu AlphaGo Zero osiągnął poziom wyższy niż poziom wszystkich innych systemów AlphaGo, co pozwala go uznać za najlepszego gracza Go w tysiącletniej historii tej gry.

O książce

Model uczenia maszynowego z dużą dokładnością przewiduje postępy leczenia pacjentów – to jeden z wielu podobnych nagłówków prasy popularno-naukowej, znaleziony w czasopiśmie *Nature* podczas pisania tego wstępu [8]. Książka uczy, jak samodzielnie budować takie światowej klasy modele uczenia maszynowego i jak wdrażać gotowe modele do użycia.

Modele uczenia maszynowego mogą być używane do predykcji, opisywania wzorców ukrytych w danych (deskrypcji) oraz do kompresji i generowania danych (np. do generowania obrazów lub opisów sekwencji wideo). Książka koncentruje się na najpopularniejszych modelach predykcyjnych, czyli modelach stosujących wykryte w treningowych danych wzorce do uzupełniania brakujących danych o nowych przypadkach, niewidzianych przez model podczas treningu.

Dla kogo jest ta książka?

Rynek sztucznej inteligencji rośnie na tyle szybko, że specjalistów od przetwarzania danych ciągle brakuje. Inżynieria danych (ang. *data science*) to interdyscyplinarna wiedza, której opanowanie wymaga znajomości algebry, geometrii, statystyki, rachunku prawdopodobieństwa i algorytmiki, uzupełniona o praktyczną umiejętność programowania w przynajmniej dwóch z trzech najpopularniejszych językach danych: SQL, R lub Python. Co więcej, sztuczna inteligencja jest przedmiotem intensywnych badań naukowych i samo śledzenie postępów w tej dziedzinie wiąże się z regularnym (codziennym) dokształcaniem.

Nic dziwnego, że inżynierowie danych są jednymi z najbardziej pożądanych i najlepiej wynagradzanych pracowników.

Zbudowanie modelu uczenia maszynowego wymaga:

- specjalistycznej wiedzy z dziedziny, w ramach której projekt jest realizowany (np. medycyny czy logistyki transportu); prawie zawsze wymaga to wsparcia eksperta z danej dziedziny;
- praktycznej znajomości statystki i umiejętności wizualizacji danych niezbędnej do oceny jakości danych;
- praktycznej znajomości języka SQL, R lub Python niezbędnej do uporządkowania, wstępnego przygotowania i wzbogacenia danych;
- zrozumienia zasad działania poszczególnych algorytmów uczenia maszynowego koniecznych do ich wyboru i optymalizacji (do czego przyda się z kolei znajomość algebry i geometrii);
- użycia języka R lub Python do stworzenia, oceny, zoptymalizowania i wdrożenia do produkcji modeli eksploracji danych (do oceny jakości modeli ponownie przyda się znajomość statystyki uzupełniona o wiedzę z zakresu rachunku prawdopodobieństwa).

Książka jest adresowana do wszystkich, którzy chcieliby poznać lub udoskonalić swoją wiedzę z powyższych (z wyjątkiem pierwszego) obszarów. Tego typu książki mogą być przystępne, napisane językiem potocznym i ilustrowane praktycznymi przykładami lub dokładne – pełne precyzyjnych równań matematycznych. Z założenia książka ma być przystępna, co oznacza, że opisowe wyjaśnienia pozostawiają Czytelnikowi możliwość ich różnorodnego interpretowania. Problem ten starałem się rozwiązać, ilustrując opisywane zagadnienia praktycznymi przykładami, których samodzielne wykonanie powinno rozwiać ewentualne niejasności. Takie podejście ma tę dodatkową zaletę, że kładzie nacisk na cenniejsze od wiedzy teoretycznej umiejętności

praktyczne [9].

Liczę, że dzięki temu trafię do szerokiego grona Czytelników i zarówno studenci kierunków informatycznych, jak również analitycy, programiści, administratorzy baz danych oraz statystycy znajdą w książce informacje, które pozwolą im opanować praktyczne umiejętności potrzebne do samodzielnego tworzenia systemów uczenia maszynowego. Na koniec dodam, że umiejętność czytania ze zrozumieniem kodu SQL, R i Python na pewno ułatwi lekturę książki.

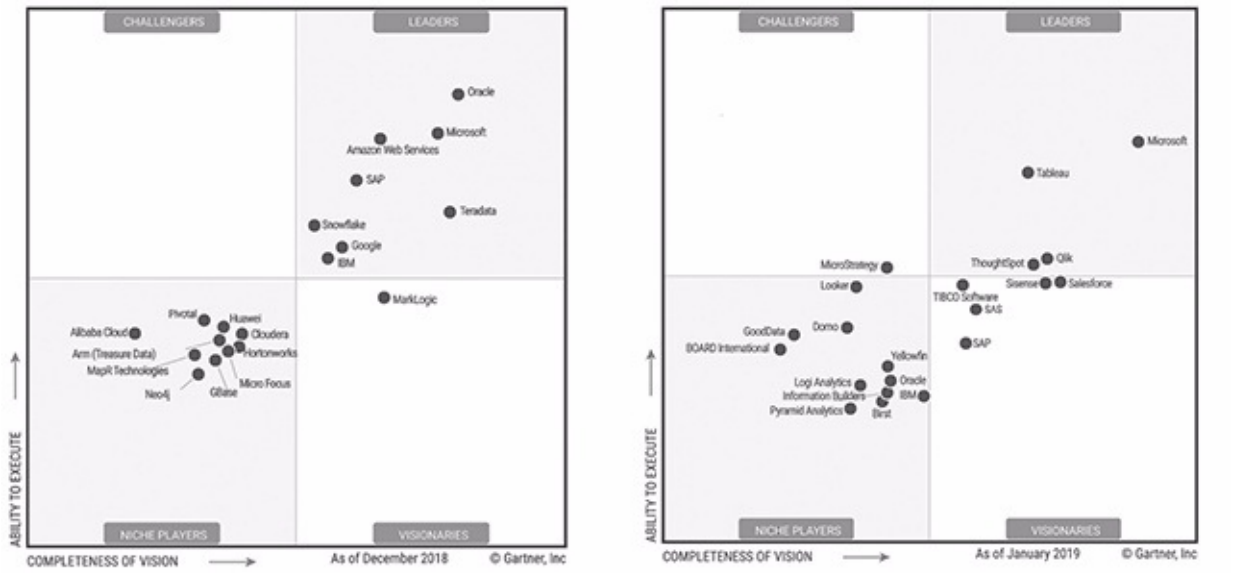
Narzędzia

Zilustrowanie opisanych w książce zagadnień praktycznymi przykładami wymagało wyboru jakichś narzędzi. **Wybór padł na serwer SQL Server 2019 i program Power BI Desktop**, ponieważ:

- oba te narzędzia są dostępne za darmo; edycja SQL Server Developer Edition ma pełne wsparcie dla opisanych w książce usług uczenia maszynowego i może być bezpłatnie używana do nauki oraz testów, jedynie produkcyjne wykorzystanie utworzonych modeli będzie wymagało zakupu odpowiedniej licencji; natomiast program Power BI Desktop jest całkowicie darmowy i może być wykorzystywany do dowolnych celów, również komercyjnych;
- SQL Server może być zainstalowany w środowiskach Windows, Linux lub na platformie Docker;
- SQL Server pozwala wydajnie przechowywać duże zbiory danych i przetwarzać je za pomocą języka SQL;
- usługi uczenia maszynowego serwera SQL Server pozwalają tworzyć modele uczenia maszynowego przy użyciu języków R, Python lub Java;
- integracja języków R, Python i Java z serwerem SQL Server wykracza poza

prostą możliwość uruchamiania skryptów tych języków po stronie serwera i pozwala nie tylko wydajnie analizować dane, lecz także wdrażać gotowe modele uczenia maszynowego do użycia;

- usługa Power BI i program Power BI Desktop są kompletnymi narzędziami do tworzenia samoobsługowych systemów BI; pozwalają one w prosty sposób pobrać dane z różnych źródeł, dowolnie je przekształcać, tworzyć rozbudowane modele biznesowe i interaktywne, rozbudowane wizualizacje;
- agencja Gartnera od wielu lat wysoko pozycjonuje oba te produkty w swoich corocznych raportach (rys. 1).



Rysunek 1. Po lewej stronie – raport dotyczący serwerów baz danych w zastosowaniach analitycznych (Gartner's 2018 Magic Quadrant for Data Management Solutions for Analytics), po prawej – raport przedstawiający platformy BI (Gartner's 2019 Magic Quadrant for Analytics and Business Intelligence Platforms)

Przykłady

W tekście książki zostały opisane tylko wybrane fragmenty kodu użytego do ilustracji omawianych zagadnień. Kompletną wersję przykładów razem z użytymi danymi można pobrać ze strony Wydawnictwa, a ich ostatnią wersję

z serwisu Github. Całość repozytorium możecie Państwo pozyskać wieloma sposobami, jednym z nich jest zastosowanie narzędzia Git, dostępnego w systemach Windows oraz Linux. Przykładowo, korzystając z polecenia:

```
$ git clone https://github.com/szelor/practical-machine-learning.
git
```

Pod adresem https://it.pwn.pl/Artykuly/Praktyczne-uczenie-maszynowe-materialy-dodatkowe znajdą Państwo archiwum ZIP z kopią bazy danych serwera SQL Server 2019 zawierającą oprócz tabel z danymi, widoki, procedury składowane i funkcje potrzebne do utworzenia opisywanych modeli uczenia maszynowego. Archiwum to zawiera również podzielone między foldery pliki z danymi, pliki Power BI Desktop i skrypty w językach R i Python, które pomogą Państwu wykonać opisywane projekty.

Pod adresem https://github.com/szelor/practical-machine-learning znajdą Państwo podzielone między foldery repozytorium zawierające pliki z danymi, pliki Power BI Desktop i skrypty w językach R i Python, które pomogą Wam wykonać opisywane projekty. W repozytorium nie znajdują jednak Państwo kopii przykładowej bazy danych, zbyt dużej, żeby można było ją tu udostępnić.

Najprościej jest skorzystać z gotowych plików i w ten sposób przekonać się, jak działają poszczególne modele, skorzystać z interaktywnych wizualizacji danych i spojrzeć na kolorowe wykresy w wysokiej rozdzielczości. Zachęcam jednak do wspólnego rozwoju tego projektu – każdy może modyfikować i rozbudowywać te skrypty, a następnie dzielić się wynikami swojej pracy z innymi.

Bibliografia

Do książki została dołączona bibliografia. Starałem się umieścić w niej jak najwięcej odnośników do ogólnodostępnych, elektronicznych wersji wymienionych w niej pozycji. Oczywiście, jeżeli jakaś pozycja nie jest dostępna

za darmo, odnośnika do niej nie ma. Namawiam do zapoznania się z tymi artykułami i książkami – zarówno tymi klasycznymi, pochodzącymi z lat 70. XX w., jak i opisującymi wyniki najnowszych badań w obszarze uczenia maszynowego.

Konwencje i oznaczenia

W książce zostały zastosowane następujące konwencje do oznaczania różnych typów tekstu:

- **czcionką pogrubioną są wyróżnione** nowe, istotne zagadnienia, na które czytelnik powinien zwrócić uwagę*;*
- *czcionką pochyłą* są pisane nazwy własne w miejscach ich pierwszego użycia oraz angielskie odpowiedniki wprowadzanych pojęć;
- `czcionką stałej szerokości` znaków są pisane przykładowe programy, pojawiające się w treści akapitu fragmenty programów (instrukcje, słowa kluczowe, modyfikatory itp.), polecenia wprowadzane z klawiatury, teksty wyświetlane na ekranie oraz adresy internetowe;
- uwagi, wskazówki, ciekawostki, dobre rady lub ostrzeżenia są pisane

   mniejszym stopniem pisma i wyróżnione znakiem widocznym obok.

Przeniesienie obliczeń jest tańsze niż przenoszenie danych.

Dokumentacja projektu Apache

Rozdział 1
Narzędzia

Tworzenie modeli uczenia maszynowego wymaga użycia odpowiednich narzędzi. Będziemy potrzebować wystarczająco wydajnego komputera z oprogramowaniem do przechowywania i przetwarzania danych, analizy statystycznej i wizualizacji danych, tworzenia i uczenia modeli oraz udostępnienia gotowych modeli użytkownikom.

Wszystkie opisane w książce projekty zostały utworzone na serwerze wyposażonym w osiem procesorów Intel Xeon E5-2673 2.4 GHz, 28 GB pamięci RAM i dysk SSD o wydajności 380 MB/s. Przykłady te jednak można odtworzyć na znacznie wolniejszym komputerze. Minimalne wymagania sprzętowe takiego komputera to **64-bitowy procesor taktowany zegarem o częstotliwości 2 GHz, 4 GB pamięci RAM i 15 GB wolnej przestrzeni na dysku**, z czego 10 GB wymaga serwer SQL Server 2019. Chociaż w kilku projektach zostały użyte sztuczne sieci neuronowe, to ich architektura jest na tyle prosta, że nauczenie ich na przykładowych danych nie wymaga użycia akceleratorów graficznych.

Użyte oprogramowanie zostało zainstalowane w środowisku anglojęzycznego systemu operacyjnego **Windows Server 2016 Datacenter**. Czytelnicy mogą użyć innego, 64-bitowego systemu operacyjnego. Jeżeli jednak serwer baz danych i wszystkie narzędzia programistyczne będą zainstalowane na tym samym komputerze, wymagany jest system Windows 8 lub nowszy, Windows

Server 2012 lub nowszy. Największą wydajność i stabilność uzyskamy w przypadku najnowszych wersji systemu Windows.

Do przechowywania i przetwarzania danych oraz do tworzenia modeli uczenia maszynowego i udostępniania ich użytkownikom został użyty serwer **SQL Server 2019 Developer Edition**. Wymaga on systemu Windows 8 lub nowszego, Windows Server 2012 lub nowszego, Red Hat Enterprise Linux 7.3+, SUSE Linux Enterprise Server v12 SP2 lub Ubuntu Linux 16.4. Serwer SQL Server 2019 można też zainstalować w środowisku Docker.

Przykładowa baza danych ML jest w wersji 2019, co oznacza, że jej odtworzenie we wcześniejszych wersjach serwera SQL nie jest możliwe. Część przykładowych danych jest dostępna w postaci plików tekstowych – w przypadku tych danych istnieje możliwość ich importu do wcześniejszych wersji serwera SQL. Pozostałe dane trzeba będzie samodzielnie wyeksportować z serwera SQL Server 2019.

Projekty utworzone w języku R można, po wprowadzeniu odpowiednich modyfikacji, uruchomić na wersjach 2016 i 2017 serwera SQL. Podobnie w przypadku projektów utworzonych w języku Python, po wprowadzeniu niezbędnych poprawek, jest możliwość ich uruchomienia w wersji 2017 serwera SQL Server. Systemy Windows lub Linux umożliwiają instalację użytych w książce narzędzi programistycznych w postaci **R Studio Desktop** i **PyCharm Community Edition**.

Do wizualizacji danych zostały użyte programy **Power BI Desktop** i **SQL Server Data Tools**. Do zainstalowania tych programów zalecane są systemy Windows 7 SP1 lub nowsze.

1.1. Język Python

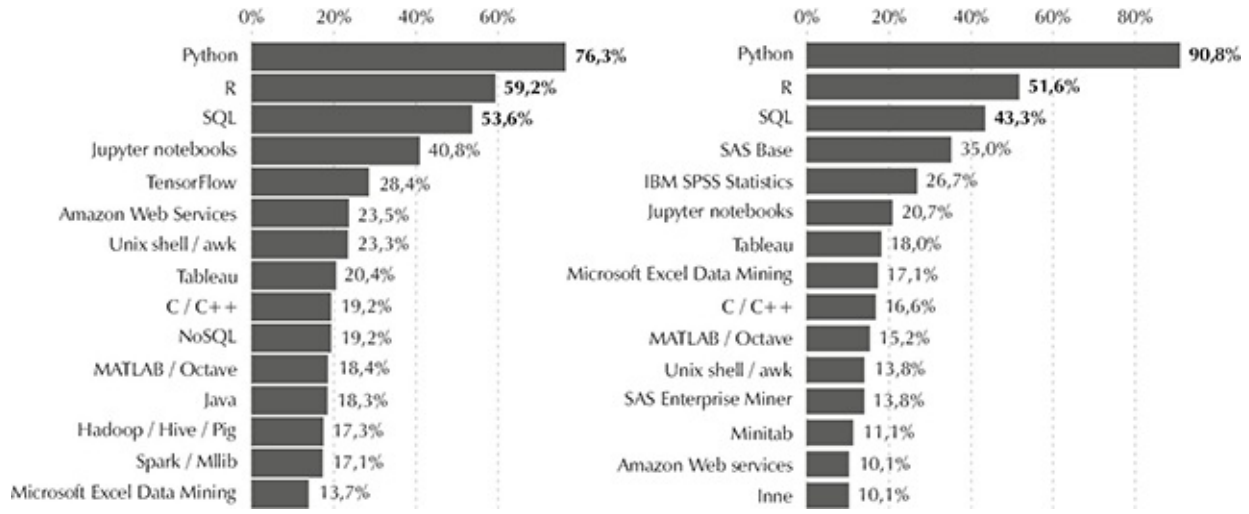
Guido van Rossum udostępnił kod źródłowy tworzonego przez siebie języka Python w 1991 r. Python od początku był rozwijany jako projekt otwarty i darmowy; jego pierwsza wersja została upubliczniona w styczniu 1994 r., druga – w październiku 2000 r., a trzecia – w grudniu 2008 r.

Python jest językiem programowania ogólnego przeznaczenia, którego mocną stroną jest duża liczba dodatkowych bibliotek, w tym bibliotek wspierających przetwarzanie i analizowanie danych. Rosnące zainteresowanie programistów sztuczną inteligencją spowodowało, że dzisiaj to Python, a nie R jest najczęściej używanym językiem uczenia maszynowego i de facto standardowym językiem głębokiego uczenia maszynowego.

Z usługą SQL Server Machine Learning Services jest instalowana dystrybucja Anaconda, zawierająca trzecią wersję języka Python, dostosowaną do zadań związanych z przetwarzaniem danych.

1.2. Język R

Stworzony przez Rossa Ihaka i Roberta Gentlemana, profesorów uniwersytetu Auckland w Nowej Zelandii, język R, jest otwartym i darmowym językiem analizy danych. Jego pierwsza wersja została udostępniona publicznie w 1995 r., pięć lat później pojawiła się pierwsza stabilna wersja języka. Język ten szybko zyskał popularność na uczelniach całego świata, a dzisiaj jest najczęściej używanym narzędziem do statystycznej analizy danych i drugim pod względem popularności narzędziem używanym przez inżynierów danych różnych specjalności (rys. 1.1).



Rysunek 1.1. Chociaż najpopularniejszym narzędziem inżynierów danych jest Python, to ponad 90% statystyków używa języka R. Trzy języki opisane w książce, R, Python i SQL, dominują wśród narzędzi zawansowanej analizy danych. Po lewej stronie – wykres pokazujący popularność narzędzi używanych przez wszystkich specjalistów od danych, po prawej – przez statystyków [10]

Podstawową siłą języka R jest ponad 5-milionowa rzesza jego użytkowników. Wielu z nich udostępnia wyniki swoich prac w postaci pakietów publikowanych na stronie projektów The Comprehensive R Archive Network (https://cran.r-project.org/), Bioconductor (https://bioconductor.org/) oraz GitHub (https://github.com/). Efektem jest zbiór ponad 15 000 ogólnodostępnych bibliotek zawierających rozwiązania wszystkich typowych problemów, na jakie można trafić, pracując z danymi. Barry Rowlingson opublikował pod adresem https://www.maths.lancs.ac.uk/~rowlings/R/TaskViews/ tematyczną listę popularnych pakietów projektu CRAN.

Język R ma też swoje słabe strony: niską wydajność, brak skalowalności i brak zintegrowanych narzędzi do udostępniania wyników użytkownikom. Niska wydajność wynika z tego, że większość funkcji jest napisanych w języku R, a nie w języku niższego poziomu, i jest wykonywanych jednowątkowo. Brak skalowalności jest efektem tego, że przetwarzane dane oraz tymczasowe wyniki muszą znajdować się w pamięci operacyjnej, czyli ilość pamięci RAM komputera ogranicza wielkość analizowanych zbiorów danych. Natomiast brak

integracji języka R z popularnymi programami do analizy danych powoduje problemy z udostępnieniem wyników analiz użytkownikom.

Problemy te zostały rozwiązane przez firmy interesujące się zastosowaniem języka R do analizy dużych zbiorów danych, w tym firmę Microsoft. Microsoft potraktował integrację swoich produktów z językiem R bardzo poważnie. Nie ograniczył się do możliwości uruchamiania instrukcji języka R po stronie serwera SQL Server i w programie Power BI Desktop, ale w 2015 r. rozpoczął zakrojony na szeroką skalę projekt integracji języka R ze swoimi produktami. W ramach tego projektu Microsoft dołączył do fundacji rozwijającej język R i przejął firmę Revolution Analytics – w 2015 r. wiodącego dostawcę oprogramowania i usług dla języka R. Produkty tej firmy, w tym biblioteka RevoScaleR i serwer Revolution R Enterprise, nadal są rozwijane w ramach projektów Microsoft R Open i Microsoft Machine Learning Server.

1.2.1. Microsoft R Open (MRO)

Microsoft R Open jest otwartą i darmową dystrybucją języka R. **MRO jest w 100% kompatybilny z oryginalnym językiem R**, czyli wszystkie skrypty działające w Open R można uruchomić, bez jakichkolwiek zmian w kodzie, w dystrybucji MRO.

Microsoft R Open zawiera kilka udoskonaleń:

- używa do obliczeń biblioteki Intel MKL Library; sprzętowa akceleracja operacji matematycznych pozwoliła wielokrotnie skrócić czas ich wykonywania – na przykład mnożenie macierzy wykonywane jest pięćdziesiąt razy szybciej, a analiza głównych składowych (PCA) – dwadzieścia razy szybciej; dodatkowe informacje na temat biblioteki Intel MKL Library są dostępne pod adresem https://www.sciencedirect.com/topics/computer-science/intel-mathkernel-

library;

- jako domyślnego repozytorium pakietów używa strony Microsoft R Application Network (https://mran.microsoft.com/); MRAN zawiera dzienne kopie strony projektu CRAN, czyli znajdziemy na niej wcześniejsze wersje wszystkich, dostępnych na stronie projektu CRAN, pakietów; dana wersja MRO (w czasie powstawania książki najnowszą wersją była wersja 3.5.1) domyślnie pobiera kopie pakietów z jednego, zawsze tego samego dnia;
- zawiera kilka dodatkowych bibliotek do współpracy z systemami Big Data i produktami firmy Microsoft.

1.2.2. Microsoft R Client (MRC)

Microsoft R Client zawiera wszystkie funkcjonalności Microsoft R Server, ale ich wydajność i skalowalność zostały ograniczone. Dane oraz tymczasowe wyniki ich przetwarzania muszą zmieścić się w pamięci RAM komputera, a stopień zrównoleglenia funkcji bibliotek RevoscaleR i MicrosoftML został ograniczony do dwóch wątków. MRC pozwala też zmienić kontekst wykonywania instrukcji języka R, co oznacza, że skrypty mogą zostać wykonane na serwerze SQL Server, Machine Learning Server lub Hadoop.

Ta wersja języka również jest darmowa. Jeżeli zainstalujemy edycję Standard serwera SQL, będziemy korzystać właśnie z wersji MRC. Microsoft R Client jest w pełni zgodny z dystrybucją Microsoft R Open.

1.2.3. Microsoft Machine Learning Server (MLS)

Microsoft Machine Learning Server to wysokowydajny i skalowalny serwer języków R i Python. Jest on dostępny w kilku wersjach. Funkcjonalność wszystkich wersji serwera jest taka sama, ale każda z nich działa w innym

środowisku: Windows, Linux albo Hadoop. Dostępna jest też wersja zintegrowana z edycją Enterprise serwerów SQL Server 2017 i 2019. Microsoft SQL Server Machine Learning Services, bo tak nazywa się ta wersja, została opisana w dalszej części rozdziału.

Microsoft Machine Learning Server (poprzednio nazywał się Microsoft R Server):

- jest oparty na Microsoft R Open, co oznacza, że **jest w 100% kompatybilny z oryginalną wersją języka R**;
- umożliwia wydajne przetwarzanie dużych zbiorów danych; dane są przetwarzane w blokach dowolnej wielkości, czyli wystarczy, żeby jednorazowo w pamięci RAM zmieścił się pojedynczy blok, a nie cały zbiór danych;
- udostępnia specjalistyczne biblioteki do przetwarzania i analizowania danych oraz tworzenia modeli uczenia maszynowego (biblioteki RevoscaleR, revoscalepy i MicrosoftML); funkcje tych bibliotek są wykonywane wielowątkowo;
- pozwala w prosty sposób wdrożyć modele do użycia – gotowe modele można opublikować jako zbiór bezstanowych interfejsów programistycznych (*RESTful APIs*) pozwalających w bezpieczny sposób zintegrować MLS z platformami Java, JavaScript i .NET.

1.3. SQL Server 2019

Historia serwera SQL Server zaczyna się w 1989 r. Wtedy Microsoft zaprezentował pierwszą, opracowaną wspólnie z Sybase i przeznaczoną dla systemu operacyjnego OS/2 wersję tego serwera baz danych. Kolejne wersje serwera SQL dla systemu OS/2 miały swoje premiery w 1992 r. (SQL Server

4.2A) i 1993 r. (SQL Server 4.2B). Równolegle Microsoft rozpoczął tworzenie wersji SQL Server działającej w środowisku systemu Windows. Ich efektem był wydany w 1993 r. SQL Server 4.21. Dwa lata później pojawiła się wersja serwera SQL działająca w środowisku systemu Windows 95 (SQL Server 6.0), a rok później świat ujrzała ostatnia wersja serwera, będąca wynikiem współpracy obu firm, czyli SQL Server 6.5.

Opublikowana w 1998 r. wersja SQL Server 7.0 była efektem przepisania w całości kodu wcześniejszych wersji (do kodu C++) i jednocześnie wprowadzenia zasadniczych zmian w architekturze. Ta wersja zawierała też dodatkową usługę analityczną o nazwie SQL OLAP Services. Wersja SQL Server 2000, która ujrzała światło dzienne dwa lata później, zawierała już usługi SQL Server Integration Services, SQL Server Reporting Services, Notification Services, a usługa analityczna – uzupełniona o możliwości tworzenia modeli uczenia maszynowego – zmieniła nazwę na SQL Server Analysis Services. Następna wersja SQL Server 2005 wprowadziła m.in. integrację z platformą .NET, możliwość przełączenia baz do optymistycznego (bazującego na wersjonowaniu, a nie blokowaniu) modelu współbieżności, oraz możliwość podwajania bazy danych.

Kolejna **wersja SQL Server 2008 rozpoczyna, wciąż trwający proces rozwijania tego serwera jako kompletnej platformy do przechowywania dowolnych** (nie tylko tabelarycznych) **danych i szeroko rozumianego ich przetwarzania**. Wśród wielu nowości zawiera ona usługę zarządzania danymi wzorcowymi (Master Data Services), funkcje szyfrowania baz danych, możliwość kompresji danych, obsługę danych przestrzennych i hierarchicznych oraz możliwość przechowywania dużych obiektów binarnych bezpośrednio na dyskach. Wersja SQL Server 2012 to przede wszystkim funkcje zapewnienia ciągłej dostępności Always On SQL Server Failover Cluster Instances oraz Always On Availability Groups. W tej wersji zadebiutowały też (opisane w dalszej części rozdziału) indeks kolumnowy i związany z nim blokowy model

przetwarzania danych oraz usługa poprawy jakości danych Data Quality Services. Udostępniona dwa lata później wersja SQL Server 2014 zawierała m.in. pamięciowe bazy danych i funkcje integracji z chmurą Azure. Długa lista nowości jest związana z wersją SQL Server 2016. Znajdują się na niej: integracja z językiem R, funkcja szyfrowania danych po stronie klienta AlwaysEncrypted, funkcja dynamicznego maskowania danych, możliwość nadawania użytkownikom uprawnień do wybranych wierszy, funkcja analizy czasu rzeczywistego będąca połączeniem indeksu kolumnowego i tabeli pamięciowej, funkcja PolyBase pozwalająca na integrację z systemami Big Data, tabele czasowe pozwalające na wersjonowanie danych, magazyn zapytań ułatwiający optymalizację wydajności czy usługa Stretch DB umożliwiająca niewidoczne dla aplikacji klienckich przeniesienie wybranych do SQL Azure. Zaledwie rok po premierze wersji 2016 Microsoft wydał wersję SQL Server 2017 – pierwszą wersję kompatybilną z systemem Linux i środowiskiem Docker. Zadebiutowały w niej funkcje adaptacyjnego wykonywania zapytań, obsługa danych grafowych i integracja z językiem Python.

Najnowsza wersja serwera SQL Server jest przedstawiana jako platforma sztucznej inteligencji. Oprócz nowego sposobu wykonywania funkcji skalarnych umożliwia ona tworzenie klastrów Big Data i uruchamianie na nich skryptów w językach R, Python, Java i Scala.

1.3.1. Instalacja

W tym punkcie została opisana instalacja serwera SQL Server i jego konsoli administracyjnej w systemie Windows. Szczegółowy opis instalacji serwera w środowiskach Linux i Docker oraz instalacji działającej w systemach Linux i MacOS konsoli Azure Data Studio znajdują się na stronach Microsoftu.

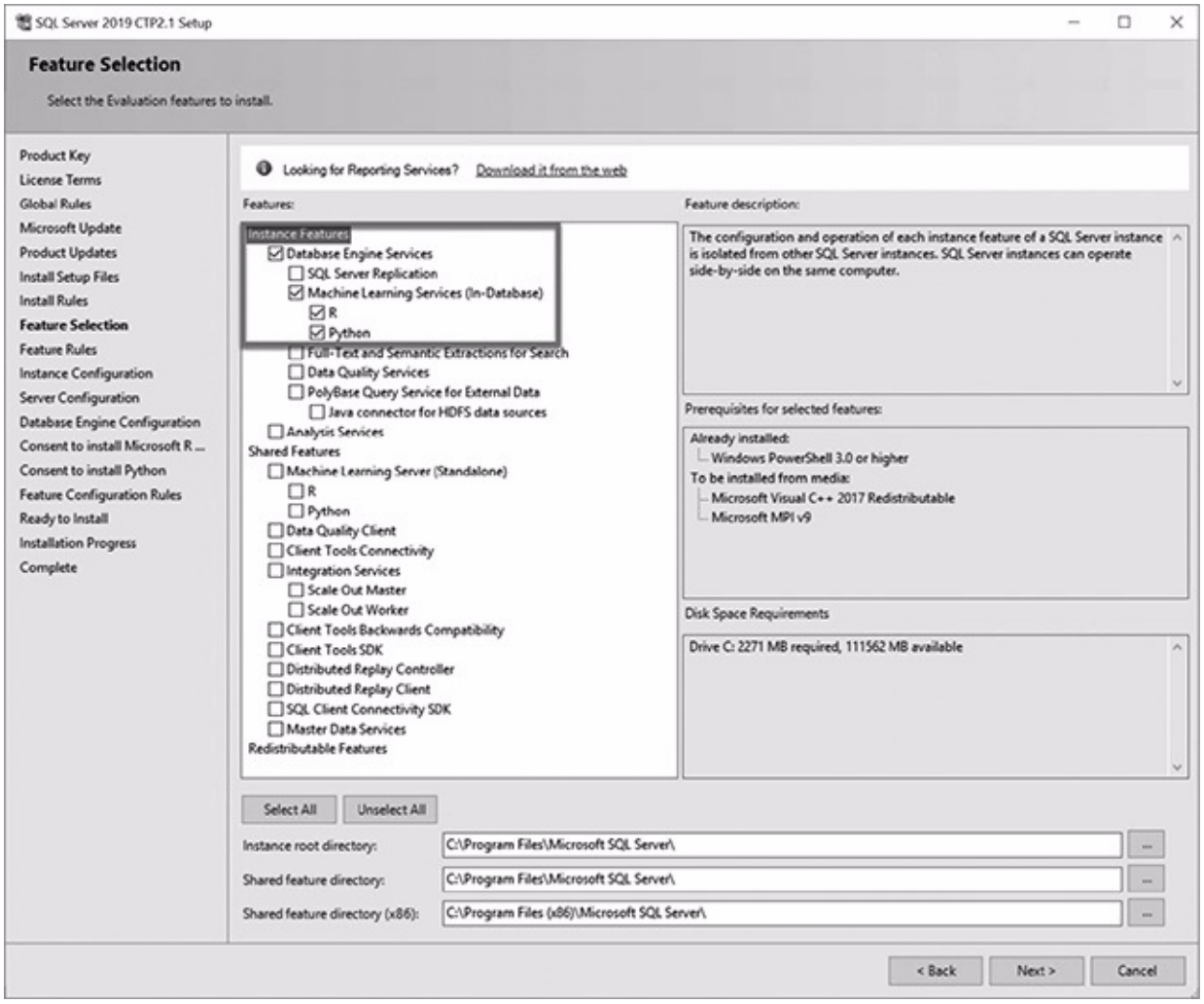
Serwer SQL Server 2019

Na stronie https://www.microsoft.com/en-us/sql-server/sql-server-downloads można znaleźć demonstracyjną edycję serwera SQL oraz, również darmowe, edycje Developer i Express. My będziemy potrzebować 180-dniowej edycji demonstracyjnej albo edycji Developer. Przykładowe projekty można też wykonać w komercyjnych edycjach Enterprise lub Standard serwera SQL Server.

Po pobraniu należy uruchomić graficzny instalator serwera. Z listy możliwych operacji należy wybrać *Installation*, a następnie wybrać zadanie *New SQL Server stand-alone installation or add features to an existing installation*. Kolejne pytanie będzie dotyczyło wyboru edycji instalowanego serwera. Po jej wybraniu zostaniemy poproszeni o zaakceptowanie licencji końcowego użytkownika. W kolejnym kroku instalator sprawdzi gotowość systemu do instalacji serwera SQL Server. Jeżeli na tym etapie zostanie wykryty jakiś problem, musi on zostać rozwiązany, aby możliwe było kontynuowanie instalacji.

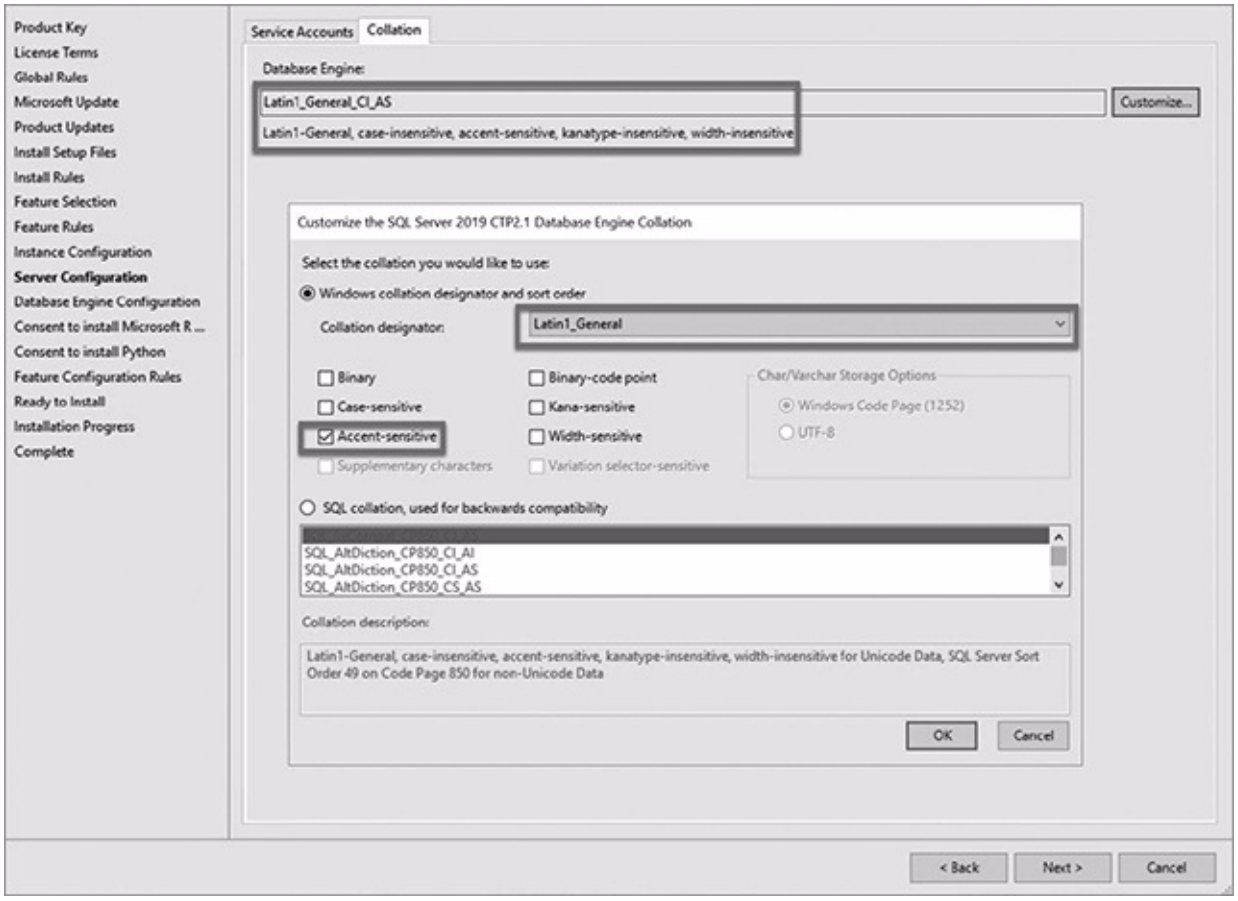
Jeżeli instalator nie zawierał wszystkich potrzebnych plików, zostaną one pobrane. Kolejny test sprawdzi zgodność systemu z wymaganiami instalatora – ostrzeżenie dotyczące zapory systemowej możemy zignorować, jeżeli planujemy pracować z serwerem SQL Server lokalnie.

Następnie zostaniemy poproszeni o wybór instalowanych komponentów. Należy zaznaczyć opcję Machine Learning Services (In-Database) i dodać oba dostępne języki (rys. 1.2).



Rysunek 1.2. Instalacja obu wersji serwera Machine Learning Server (zintegrowanej z bazą danych i samodzielnej) na tym samym komputerze, choć możliwa, jest niezalecana

Jeżeli na komputerze nie ma zainstalowanych innych serwerów SQL Server, to powinniśmy wybrać domyślną instancję (*Default instance*). W ten sposób będziemy odwoływać się do serwera SQL za pomocą samej nazwy komputera – taka konwencja została użyta w przykładowych skryptach. Kolejne pytanie będzie dotyczyło konfiguracji kont użytkowników użytych do uruchomienia poszczególnych usług serwera i wyboru wersji językowej. O ile domyślna konfiguracja kont użytkowników nie powinna być zmieniana, to jeżeli instalujemy serwer na potrzeby tej książki, należy się upewnić, że wybraną wersją językową jest *Latin1_General_CI_AS* (rys. 1.3).



Rysunek 1.3. Wersja językowa serwera SQL Server jest ustalana na podstawie ustawień regionalnych systemu, w którym jest on instalowany. Zmiana tego ustawienia jest niemożliwa bez ponownej instalacji serwera

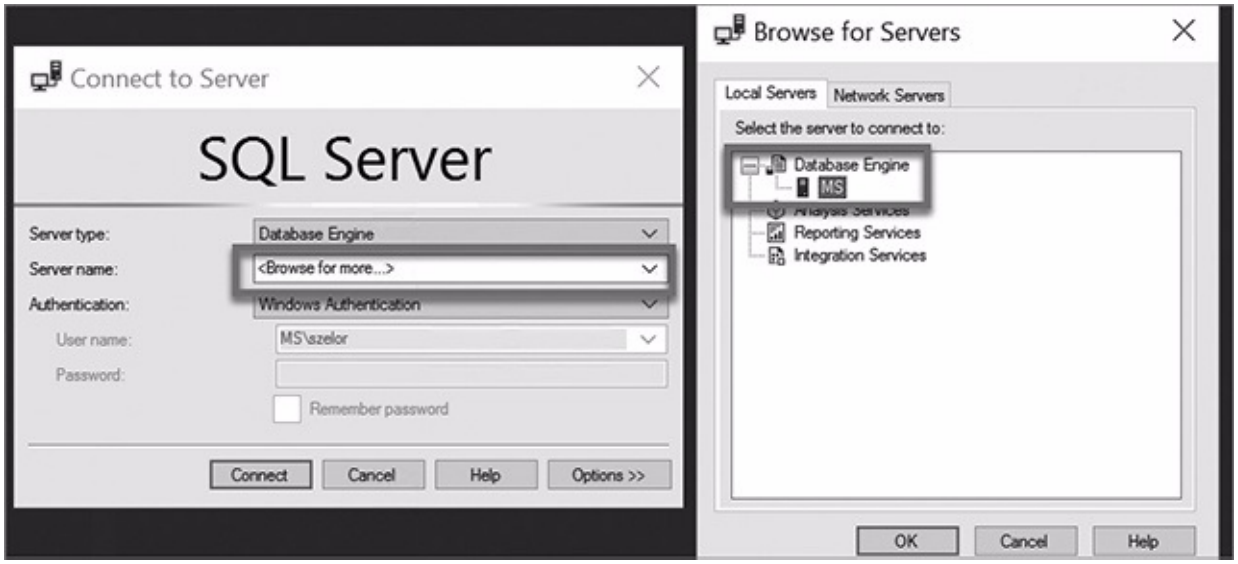
Odpowiadając na kolejne pytanie instalatora, należy dodać swoje konto do grupy administratorów serwera SQL Server. Jeżeli komputer ma dostęp do internetu, pozostałych opcji instalacyjnych nie będziemy musieli zmieniać, wystarczy, że zgodzimy się na instalację języków R i Python. Ich instalacja na komputerze bez dostępu do sieci wymaga wcześniejszego pobrania odpowiednich plików, co zostało opisane w dostępnym na stronie Microsoft artykule *Offline installation for Machine Learning Server for Windows*. Po zakończeniu instalacji należy się upewnić, czy została ona zakończona sukcesem.

Konsola SQL Server Management Studio (SSMS) 18

Konsola SQL Server Management Studio jest darmowa i dostępna na witrynie

firmy Microsoft. Należy pobrać i zainstalować wersję 18 lub nowszą tej konsoli.

Po zakończeniu instalacji będziemy mogli uruchomić konsolę SSMS i połączyć się z wcześniej zainstalowanym serwerem SQL Server (rys. 1.4).



Rysunek 1.4. Jeżeli wyświetlona automatycznie nazwa serwera SQL Server jest nieprawidłowa, listę nazw wszystkich lokalnych instancji wyświetlimy poleceniem Browse for more…

SQL Server Data Tools

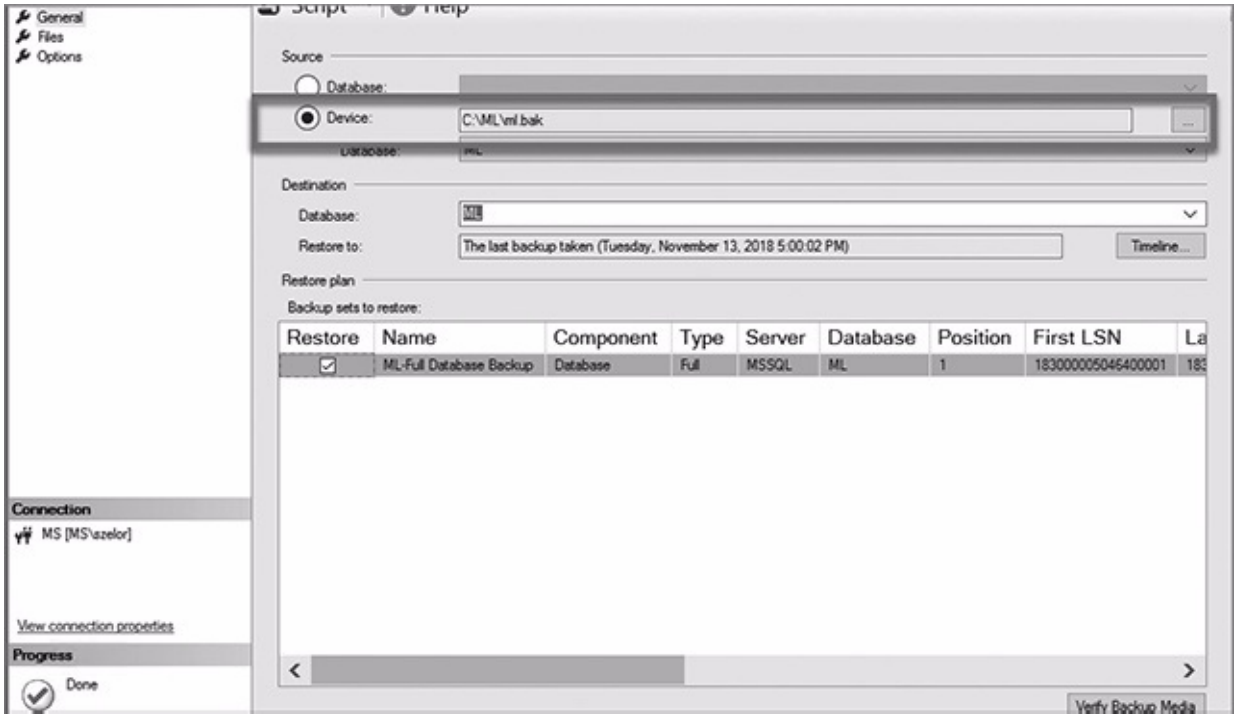
SQL Server Data Tools (SSDT) to zestaw narzędzi do tworzenia projektów bazodanowych, pakietów SSIS, modeli Analysis Services i raportów Reporting Services. W niniejszej książce SSDT zostało użyte do przedstawienia wyników analiz na raportach. Instalator wersji 15.8 SSDT jest dostępny pod adresem https://go.microsoft.com/fwlink/?linkid=2038031; po jego pobraniu i uruchomieniu należy zainstalować przynajmniej narzędzia SQL Server Reporting Services.

Przykładowa baza danych

Najprostszym sposobem odtworzenia przykładowej bazy z kopii zapasowej jest skorzystanie z menu kontekstowego eksploratora obiektów (*Object Explorer*). Po pobraniu dostępnej pod adresem https://it.pwn.pl/Artykuly/Praktyczne-

uczenie-maszynowe-materialy-dodatkowe kopii bazy należy kliknąć prawym przyciskiem folder *Databases*, z menu kontekstowego wybrać polecenie *Restore Database*, wskazać lokalizację pobranego pliku kopii zapasowej i kliknąć *OK* (rys. 1.5).

Po zakończeniu odtwarzania baza danych ML powinna być widoczna na liście baz danych serwera – jeżeli się tam nie pojawiła, należy odświeżyć tę listę, wybierając z menu kontekstowego folderu *Databases* polecenie *Refresh*.



Rysunek 1.5. Kreator odtwarzania baz danych automatycznie umieści pliki bazy danych w odpowiednim folderze

1.3.2. Microsoft SQL Server Machine Learning Services

Nie ma firmy, która nie chciałaby skorzystać z zaawansowanej analizy danych do zdobycia przewagi nad konkurencją. Jednak analiza posiadanych danych wymaga przezwyciężenia kilku poważnych problemów. Firmy muszą się zmierzyć z brakiem wiedzy (nie wszyscy pracownicy wiedzą, jak pracować

z danymi), utrudnionym dostępem do danych (dane są rozproszone między wiele specjalistycznych systemów) i niską wydajnością systemów analitycznych (pobranie setek megabajtów danych na lokalny komputer – tylko po to, żeby następnie w Excelu wybrać z milionów rekordów kilka tysięcy potrzebnych nam w danym momencie – jest tak wolne, że niepraktyczne). Wszystkie te problemy można rozwiązać za pomocą usługi SQL Server Machine Learning Services. Ta, zintegrowana z bazą danych, wersja serwera MLS pozwala wydajnie przetwarzać duże zbiory danych i pobierać na lokalne komputery jedynie wyniki ich analiz, a nie dane źródłowe.

SQL Server zarządza zasobami komputera, takimi jak czas procesora czy pamięć RAM, przydzielając je na potrzeby wykonania żądań użytkowników. Odpowiada za to moduł SQL Server Platform Abstraction Layer (SQLPAL). Ten sam moduł jest używany do zarządzania zasobami potrzebnymi do wykonywania przez serwer SQL Server instrukcji języków R i Python. Oznacza to, że analizując dane po stronie serwera, a nie na lokalnych komputerach, nie tylko unikniemy czasochłonnego przesyłania danych przez sieć, lecz także efektywnie wykorzystamy zasoby obliczeniowe naszego serwera.

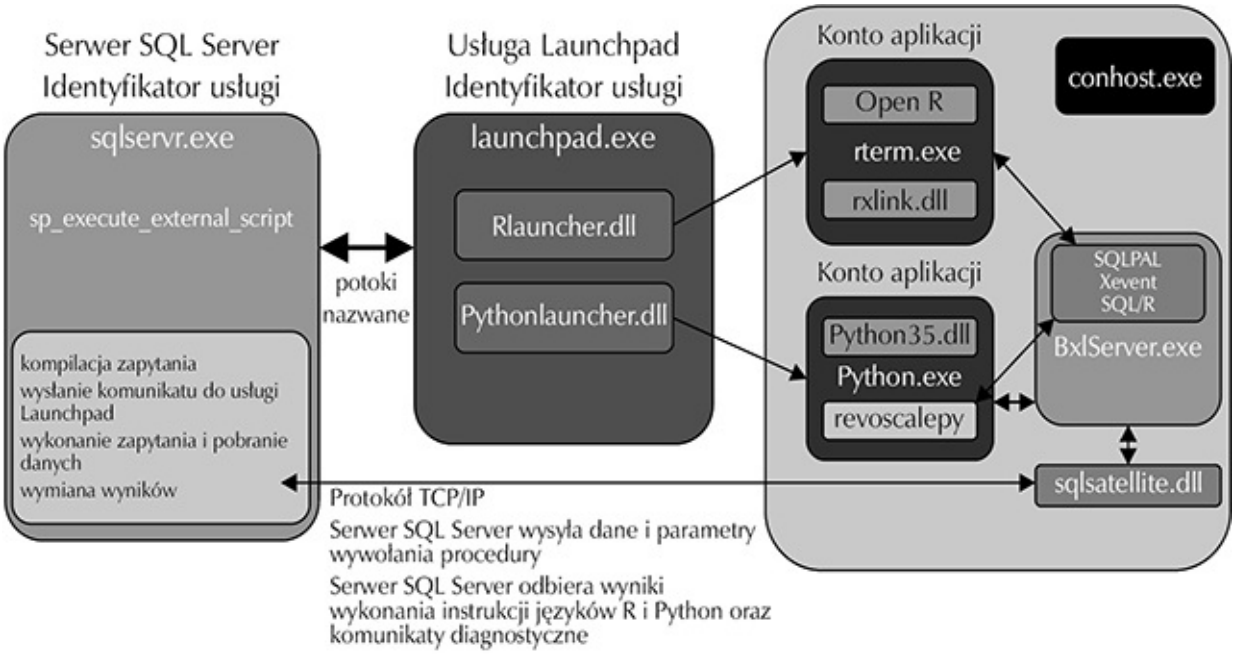
 Przeniesienie analiz na serwery, na których znajdują się dane, pozwala wielokrotnie skrócić czas analiz. Na przykład trening modeli uczenia maszynowego, który wcześniej trwał kilka godzin, skróci się do kilkudziesięciu sekund.

Znajomość budowy i działania usługi SQL Server Machine Learning Services pomoże nam efektywnie wykonywać instrukcje języków R i Python, w tym trenować modele uczenia maszynowego, po stronie serwera SQL Server.

Architektura

Usługa SQL Server Machine Learning Services została zaprojektowana z myślą o wydajnym i bezpiecznym wykonywaniu instrukcji języków R i Python. Wymaga to ścisłej integracji z serwerem baz danych – tylko w ten sposób potrzebne dane mogą być efektywnie pobierane z tabel. Jednocześnie,

wykonanie instrukcji tych języków nie może wpływać na wydajność serwera SQL Server i zagrozić bezpieczeństwu przechowywanych w nim danych. Pogodzenie tego wymagało zaprojektowania architektury pozwalającej na szybką wymianę danych między serwerem SQL Server a zewnętrznym procesem – interpretatorem instrukcji języków R i Python. Została ona pokazana na rysunku 1.6.



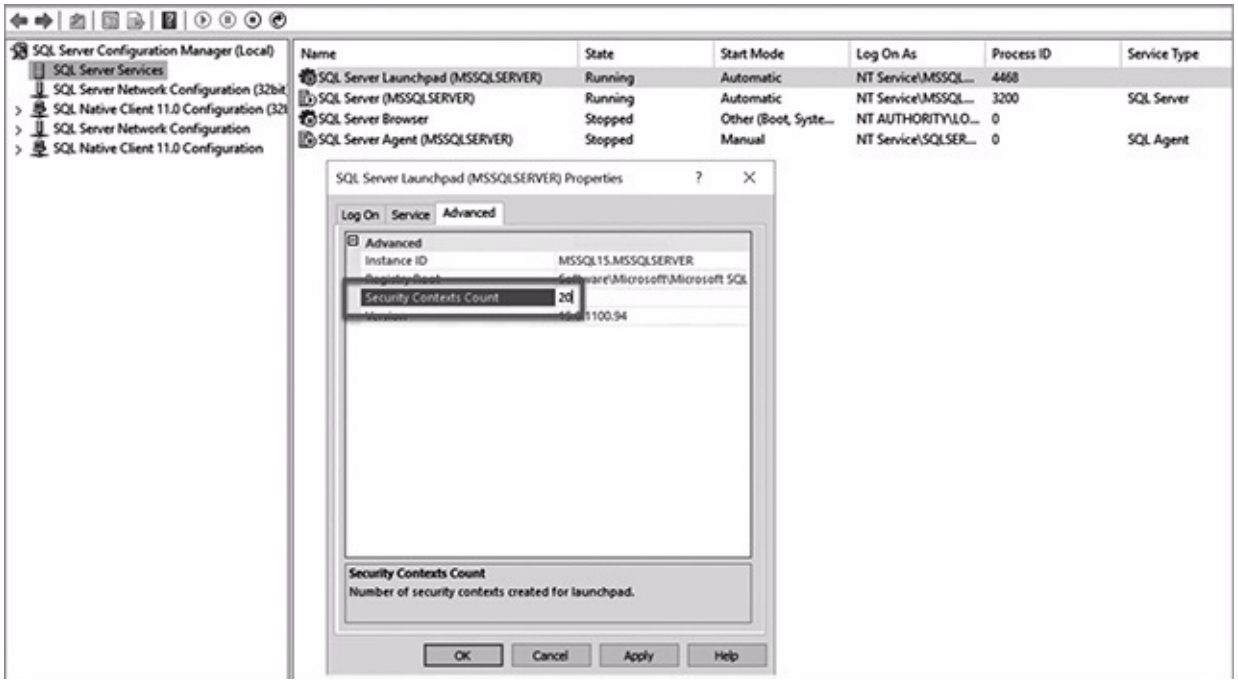
Rysunek 1.6. Architektura usługi SQL Server Machine Learning Services

Podczas instalacji usługi SQL Server Machine Learning Services na dysku, na którym jest instalowany serwer SQL Server, są tworzone trzy linki symboliczne: *SQL-MSSQLSERVERExtensibilityData* wskazujący na folder *C:\Program Files\Microsoft SQL Server\MSSQL15.MSSQLSERVER\MSSQL\ExtensibilityData*, *SQL-MSSQLSERVER-ExtensibilityData-PY* wskazujący na folder *C:\Program Files\Microsoft SQL Server\MSSQL15.MSSQLSERVER\MSSQL\ExtensibilityData* i *SQL-MSSQLSERVER-R_SERVICES* wskazujący na folder *C:\Program Files\Microsoft SQL Server\MSSQL15.MSSQLSERVER\R_SERVICES*.

Dwa pierwsze linki wskazują na folder zawierający 20 podfolderów dla 20 kontenerów aplikacji. Te podfoldery są używane do wymiany danych binarnych (np. grafiki) między serwerem SQL Server a środowiskami uruchomieniowymi języków R i Python. Po zakończeniu danej sesji, na przykład wykonaniu procedury składowanej, dane z tych podfolderów są automatycznie kasowane.

Za każdym razem, gdy SQL Server uruchamia zewnętrzny skrypt, wywołuje usługę Launchpad, przekazując jej ID któregoś z kontenerów aplikacyjnych. Usługa Launchpad musi działać, żeby możliwe było wykonanie instrukcji języków R lub Python. Usługa ta zależy od usługi MSSQLSERVER, czyli zatrzymanie serwera SQL Server spowoduje zatrzymanie usługi Launchpad.

Domyślnie zakładanych jest 20 kontenerów aplikacji, co oznacza, że jednocześnie zewnętrzne skrypty mogą być wykonywane przez maksymalnie 20 wątków. Zmienić tę liczbę możemy, uruchamiając konsolę SQL Server Configuration Manager i w zakładce zaawansowanych właściwości usługi Launchpad ustawiając nową wartość Security Context Count (rys. 1.7).



Rysunek 1.7. Jeżeli jednocześnie będzie używanych więcej niż 20 sesji R lub Python, zwiększenie wartości Security Context Count zapobiegnie oczekiwaniu na zakończenie się którejś z już działających sesji zanim

możliwe będzie uruchomienie nowej

Usługa Launchpad uruchamia (jeżeli były zatrzymane) procesy Rlauncher.exe lub Python.exe. Który, z tych procesów zostanie uruchomiony, zależy od tego, którego języka instrukcje mają zostać wykonane. Do działającego procesu Rlauncher lub Python zostaje przekazane zadanie wykonania instrukcji użytkownika.

Przekazane instrukcje są wykonywane przez procesy rterm.exe lub Python.exe. Biblioteka rxlink.dll przesyła do serwera BxlServer funkcje języka R, wysyła informacje diagnostyczne do modułu SQLPAL i rejestruje zdarzenia monitorowane przez mechanizm Extended Events. W przypadku języka Python, jeżeli używane są funkcje biblioteki revoscalepy, również wysyłają one informacje diagnostyczne do modułu SQLPAL i rejestrują zdarzenia monitorowane przez mechanizm Extended Events. Biblioteka sqlsatellite.dll przesyła dane między serwerem SQL a procesami rterm.exe lub Python.exe. W efekcie skrypty są uruchamiane jako zewnętrzne procesy, a serwer SQL Server nadzoruje ich wykonanie i dba o bezpieczeństwo danych przetwarzanych przez te procesy.

Uruchamianie instrukcji języków R i Python

Uruchomienie instrukcji języków R i Python wymaga wcześniejszego ustawienia opcji serwera SQL `external scripts enabled`. Wykonanie tej zmiany implikuje posiadanie uprawnień administratora serwera SQL Server:

```
EXEC SP_CONFIGURE 'external scripts enabled', 1
GO
RECONFIGURE
GO
```

```
Configuration option 'external scripts enabled' changed from 0 to 1.
Run the RECONFIGURE statement to install.
```

Zarówno w przypadku języka R, jak i Python, instrukcje są wykonywane przez wywołanie systemowej procedury `sp_execute_external_script`, której ogólną składnię można zapisać następująco:

```
sp_execute_external_script
     @language = N'language',
     @script = N'script'
     [ , @input_data_1 = N'input_data_1' ]
     [ , @input_data_1_name = N'input_data_1_name' ]
     [ , @input_data_1_order_by_columns =
       N'input_data_1_order_by_columns' ]
     [ , @input_data_1_partition_by_columns =
       N'input_data_1_partition_by_columns' ]
     [ , @output_data_1_name = N'output_data_1_name' ]
     [ , @parallel = 0 | 1 ]
     [ , @params = N'@parameter_name data_type [ OUT | OUTPUT ]
     [ ,...n ]' ]
     [ , @parameter1 = 'value1' [ OUT | OUTPUT ] [ ,...n ] ]
```

Pierwszym obowiązkowym parametrem jest `@language`. Może on przyjąć jedną z wartości: `R`, `Python` (od wersji 2017) lub `Java` (od wersji 2019). Drugi obowiązkowy parametr `@script` pozwala przekazać do procedury wykonywane instrukcje wybranego języka. **Instrukcje te są przekazywane jako dane tekstowe, a ich poprawność składniowa i semantyczna nie są sprawdzane przed wykonaniem procedury** `sp_execute_external_script`.

Pozostałe parametry procedury `sp_execute_external_script` są opcjonalne:

- `@input_data_1` pozwala podać zapytanie języka T-SQL, którego wynik zostanie przekazany do skryptu;
- `@input_data_1_name` pozwala zmienić domyślną nazwę zmiennej `InputDataSet` przechowującej wynik zapytania;
- `@input_data_1_order_by_columns` pozwala wskazać kolumnę użytą do posortowania wyniku zapytania; parametr ten został dodany w wersji 2019

serwera SQL Server;

- `@input_data_1_partition_by_columns` pozwala wskazać kolumnę użytą do podzielenia wyniku zapytania na partycje; parametr ten został dodany w wersji 2019 serwera SQL Server;
- `@parallel` pozwala wymusić wielowątkowe wykonanie instrukcji języka R lub Python;
- `@params` pozwala przekazać do skryptu języka R lub Python listę parametrów o określonych nazwach i typach;
- kolejne parametry procedury pozwalają przypisać wartości parametrom skryptu zdefiniowanym w poprzednim kroku.

Nazwy kolumn wyniku wywołania procedury, przekazywanego przez zmienną `OutputDataSet`, można ustalić klauzulą `WITH RESULT SETS` polecenia `EXECUTE`.

Wykorzystajmy wiedzę na temat procedury systemowej `sp_execute_external_script` do sprawdzenia działania usługi SQL Server Machine Learning Services. Proszę zwrócić uwagę, że parametry tekstowe są przekazywane jako ciągi UNICODE (tekst musi być poprzedzony dużą literą `N`):

```
EXEC sp_execute_external_script @language =N'R',
       @script=N'OutputDataSet <- InputDataSet',
       @input_data_1 =N'SELECT 1 as CheckToSeeIfRIsWorking'
WITH RESULT SETS ((CheckToSeeIfRIsWorking int not null));
GO

EXEC sp_execute_external_script  @language =N'Python',
       @script=N'OutputDataSet = InputDataSet',
       @input_data_1 = N'SELECT 1 AS CheckToSeeIfPythonIsWorking'
WITH RESULT SETS ((CheckToSeeIfPythonIsWorking int not null));
GO
```

W wyniku powinniśmy otrzymać dwa wiersze, każdy zawierający jedną

kolumną typu `int` z wartością `1`. Nazwy i typy tych kolumn zostały określone w klauzuli `WITH RESULT SETS` polecenia `EXEC`:

```
CheckToSeeIfRIsWorking
1
CheckToSeeIfPythonIsWorking
1
```

Konfiguracja

Wykonanie każdej instrukcji języka SQL wymaga pewnych zasobów: czasu procesora, pamięci RAM oraz operacji odczytu i zapisu. SQL Server kontroluje przyznawanie tych zasobów za pomocą mechanizmu Resource Governor. Tylko edycje Enterprise i Developer pozwalają tworzyć dowolne grupy zasobów i przypisywać im żądania użytkowników. Natomiast wszystkie edycje serwera SQL Server zawierają domyślną pulę zasobów, do której trafiają wszystkie żądania użytkowników. Nie zawiera ona żadnych ograniczeń zasobów, czyli zaklasyfikowane do niej żądania mogą korzystać ze wszystkich dostępnych zasobów komputera:

```
SELECT *
FROM sys.resource_governor_resource_pools
WHERE name = 'default';
```

```
pool_id name     min_cpu_percent   max_cpu_percent   min_memory_percent max_memory_percent
cap_cpu_percent  min_iops_per_volume        max_iops_per_volume
2       default  0        100      0        100      100      0        0
```

Wykonywanie zewnętrznych skryptów jest kontrolowane przez moduł SQL Server Extensibility Framework. Zasoby potrzebne do wykonania instrukcji języka R i Python, czyli dostępne procesory i ich czas oraz pamięć RAM, można ograniczać, konfigurując domyślną pulę zewnętrznych zasobów. Początkowa konfiguracja tej puli **ogranicza ilość pamięci RAM dostępnej dla usługi Machine Learning Services do 20% pamięci nieprzydzielonej serwerowi**

SQL Server. Innymi słowy, jeżeli serwer jest wyposażony w 28 GB pamięci RAM, z czego 24 GB używa serwer SQL Server, to z pozostałych 4 GB tylko 20% (800 MB) będzie dostępne dla skryptów języków R i Python.

Domyślną konfigurację możemy zmienić, ograniczając maksymalną ilość pamięci RAM dostępną dla serwera SQL Server i zwiększając procent pozostałej pamięci dostępnej dla usługi SQL Server Machine Learning Services:

```
EXEC SP_CONFIGURE 'show advanced options',1
RECONFIGURE
```

```
Configuration option 'show advanced options' changed from 0 to 1. Run the RECONFIGURE
statement to install.

EXEC SP_CONFIGURE 'max server memory (MB)' , 20000
RECONFIGURE
```

```
Configuration option 'max server memory (MB)' changed from 2147483647 to 20000. Run the
RECONFIGURE statement to install.

ALTER EXTERNAL RESOURCE POOL "default"
WITH (max_memory_percent = 40);
ALTER RESOURCE GOVERNOR  RECONFIGURE;
GO
SELECT *
FROM sys.resource_governor_external_resource_pools
WHERE name = 'default';
```

```
external_pool_id name     max_cpu_percent   max_memory_percent  max_processes    version
2               default  100               40                  0                2
```

Powyższa zmiana domyślnej konfiguracji jest zalecana dla każdej edycji serwera SQL Server. Jak wspomniałem, edycje Enterprise i Developer dodatkowo pozwalają na bardziej szczegółową kontrolę użycia zasobów komputera. Tworząc niestandardowe grupy zewnętrznych zasobów oraz przypisując do nich poszczególne aplikacje lub użytkowników, możemy na przykad zagwarantować ważnej aplikacji wymaganą do jej działania ilość pamięci RAM i czasu procesorów.

Microsoft opublikował kilka raportów pozwalających sprawdzić konfigurację i działanie usługi SQL Server Machine Learning Services. Raporty te znajdują się w repozytorium materiałów dodatkowych dla książki, w folderze *Chapter01*. Raport *ML Services – Configuration.rdl* zawiera podstawowe informacje o usłudze SQL Server Machine Learning Services. Aby go wyświetlić:

- uruchom konsolę SSMS i połącz się z serwerem SQL Server;
- prawym przyciskiem myszy kliknij widoczną w okienku eksploratora obiektów ikonę serwera i z menu kontekstowego wybierz opcję *Reports/Custom Report*;
- wskaż plik raportu *ML Services – Configuration.rdl* i kliknij *Open;*
- po zaakceptowaniu ostrzeżenia przed uruchamianiem nieznanych programów raport zostanie wyświetlony (rys. 1.8).

Warto zwrócić uwagę na wyłączoną funkcję niejawnego uwierzytelniania (*Implied Authentication*). Wrócimy do niej w następnym punkcie poświęconym bezpieczeństwu, w tym miejscu wspomnę tylko, że można ją włączyć, klikając przycisk *Configure instance*.

SQL Server Machine Learning Services - Configuration

This report provides configuration settings for SQL Server Machine Learning Services and properties of the R/Python installation.
Click the "Configure Instance" buton when visible to configure instance for external scripts execution & implied authentication.

Machine Learning Services Installed	External Scripts Enabled	Implied Authentication Enabled	TCP/IP Enabled
✓	✓	✗	✓

View SQL script

Configure Instance

Installed Runtimes
Not Supported
Python
R
Test

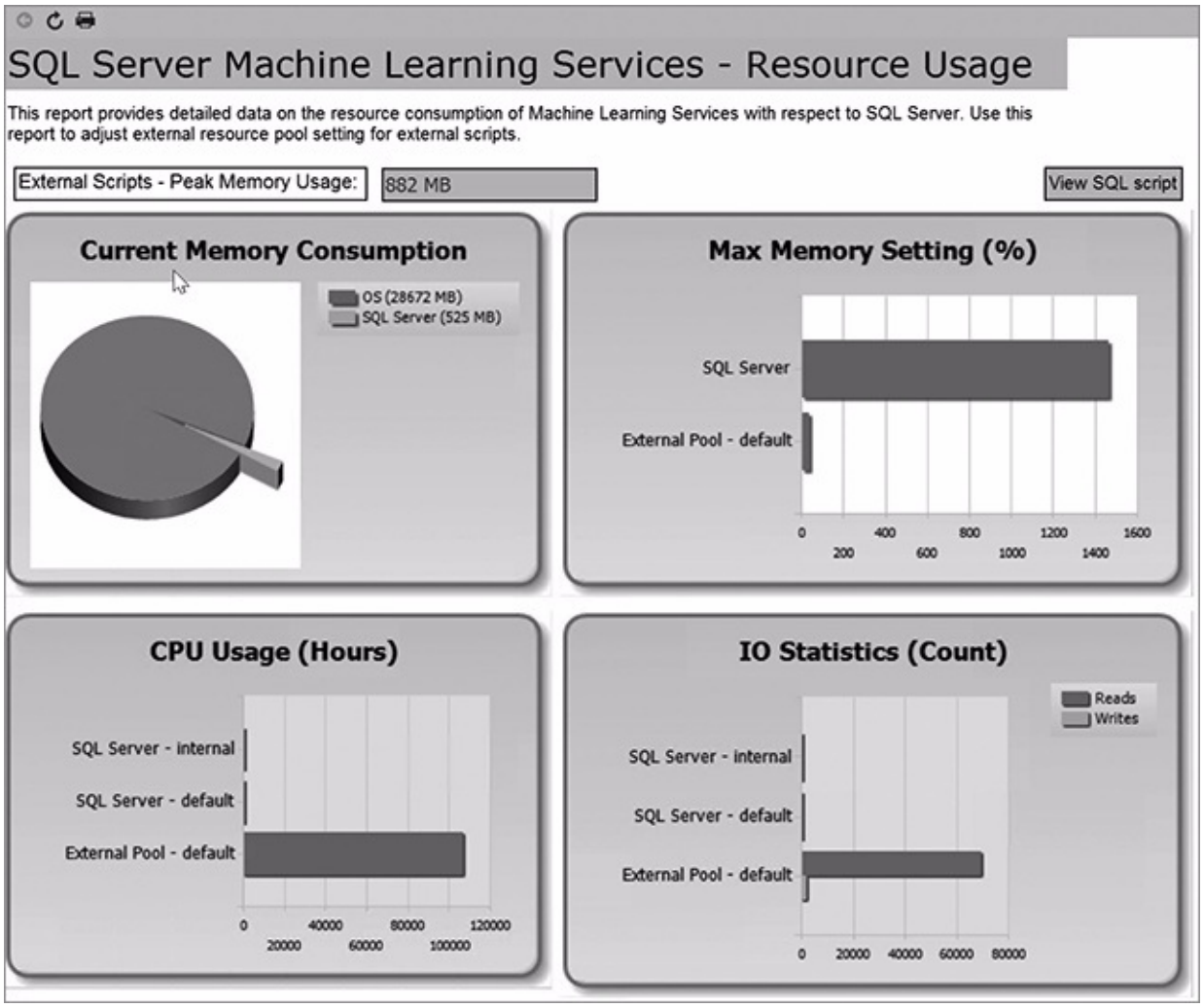
R Runtime Property	Property Value
R.version	3.4.4
Revo.version	9.4
MicrosoftML.version	9.4.6
R.home	C:/SQL-MSSQLSERVER-R_SERVICES
libPaths	C:/SQL-MSSQLSERVER-R_SERVICES/library
defaultPackages	datasets, utils, grDevices, graphics, stats, methods, rpart, lattice, RevoScaleR, mrsdeploy, MicrosoftML, RevoMods, RevoUtils, RevoUtilsMath

Python Runtime Property	Property Value
Python.home	C:\Program Files\Microsoft SQL Server\MSSQL15.MSSQLSERVER\PYTHON_SERVICES\
Python.version	3.5.2 |Anaconda 4.2.0 (64-bit)| (default, Jul 5 2016, 11:41:13) [MSC v.1900 64 bit (AMD64)]
Revo.version	9.4.6
MicrosoftML.version	9.4.6
libpaths	['', 'C:\\Program Files\\Microsoft SQL Server\\MSSQL15.MSSQLSERVER\\PYTHON_SERVICES\\python35.zip', 'C:\\Program Files\\Microsoft SQL Server\\MSSQL15.MSSQLSERVER\\PYTHON_SERVICES\\DLLs', 'C:\\Program Files\\Microsoft SQL Server\\MSSQL15.MSSQLSERVER\\PYTHON_SERVICES\\lib', 'C:\\Program Files\\Microsoft SQL Server\\MSSQL15.MSSQLSERVER\\PYTHON_SERVICES', 'C:\\Program Files\\Microsoft SQL Server\\MSSQL15.MSSQLSERVER\\PYTHON_SERVICES\\lib\\site-packages', 'C:\\Program Files\\Microsoft SQL Server\\MSSQL15.MSSQLSERVER\\PYTHON_SERVICES\\lib\\site-packages\\Sphinx-1.5.4-py3.5.egg', 'C:\\Program Files\\Microsoft SQL Server\\MSSQL15.MSSQLSERVER\\PYTHON_SERVICES\\lib\\site-packages\\win32', 'C:\\Program Files\\Microsoft SQL Server\\MSSQL15.MSSQLSERVER\\PYTHON_SERVICES\\lib\\site-packages\\win32\\lib', 'C:\\Program Files\\Microsoft SQL Server\\MSSQL15.MSSQLSERVER\\PYTHON_SERVICES\\lib\\site-packages\\Pythonwin', 'C:\\Program Files\\Microsoft SQL Server\\MSSQL15.MSSQLSERVER\\PYTHON_SERVICES\\lib\\site-packages\\setuptools-27.2.0-py3.5.egg', 'C:\\SQL-MSSQLSERVER-ExtensibilityData-PY\\Appcontainer1\\AC6EF421-B985-4A4F-A5FC-A49A0BE5C163\\rxLibs', 'C:\\SQL-MSSQLSERVER-ExtensibilityData-PY\\Appcontainer1\\6C8602B

Rysunek 1.8. Raport dodany do konsoli SSMS możemy wyświetlać, wybierając go z menu Reports

Pozostałe raporty znajdujące się w folderze to:

- *ML Services – Active Sessions.rdl* – zwraca informacje o aktywnych sesjach użytkowników wykonujących zewnętrzne skrypty;
- *ML Services – Execution Statistics.rdl* – zwraca statystyki wywołania procedury `sp_execute_external_script`;
- *ML Services – Extended Events.rdl* – zwraca listę rozszerzonych, zgłaszanych przez zewnętrzne skrypty, zdarzeń;
- *ML Services – Packages.rdl* – zwraca informacje o zainstalowanych bibliotekach języka R;
- *ML Services – Resource Usage.rdl* – przedstawia zużycie zasobów komputera przez serwer SQL i usługę SQL Server Machine Learning Services (rys. 1.9).



Rysunek 1.9. Właściwe zarządzanie zasobami komputera ma zasadniczy wpływ na wydajność działających na nim usług, w tym przypadku na czas analizowania danych i trenowania modeli uczenia maszynowego

1.3.3. Bezpieczeństwo

Bezpieczeństwo powinno być priorytetem każdego projektu uczenia maszynowego. Jeżeli dane są przesyłane do komputerów użytkowników i tam analizowane, ich skuteczne zabezpieczenie przed dostępem niepowołanych osób będzie trudne. Przeniesienie analiz na serwer baz danych rozwiązuje również kwestie ochrony danych i wyników ich analiz. W takim przypadku będziemy mogli skorzystać z rozbudowanego i sprawdzonego modelu bezpieczeństwa serwera SQL Server.

Model bezpieczeństwa serwera SQL Server pozwala na wdrożenie stosunkowo niewielkim nakładem pracy strategii dogłębnej obrony, czyli strategii polegającej na wykorzystaniu wielu niezależnych zabezpieczeń. W tym punkcie ograniczymy się do zabezpieczenia kont użytkowników i kontrolowania wykonywanych przez nich operacji.

Każdy użytkownik, zanim nawiąże sesję z serwerem, musi zostać uwierzytelniony. Tożsamość użytkowników może być potwierdzana przez:

1. System operacyjny (tryb *Windows Authentication*). Ten tryb jest zdecydowanie bezpieczniejszy. Jeśli użytkownicy serwera SQL mają konta w domenie Active Directory, tryb uwierzytelniania Windows pozwala w prosty sposób uzyskać funkcjonalny model zarządzania użytkownikami – wystarczy, że zostaną oni przypisani do grup systemowych, dla których zostały utworzone loginy serwera SQL Server. W tym trybie serwer baz danych w ogóle nie sprawdza tożsamości użytkowników, w pełni polegając na systemie operacyjnym.
2. Serwer SQL (tryb *SQL Server and Windows Authentication*) – w tym trybie możliwe jest uwierzytelnianie na podstawie konta systemu operacyjnego lub przez podanie loginu SQL i hasła. Podane przez użytkownika login i hasło są przed wysłaniem do serwera SQL szyfrowane jego certyfikatem, a następnie porównywane z zapisanymi w systemowej bazie master loginami i skrótami haseł. Loginy SQL mogą być (i domyślnie są) chronione zasadami kont obowiązującymi na lokalnym systemie Windows. Oznacza to, że jeżeli z jakiegoś powodu serwer SQL działa w trybie mieszanym, należy skonfigurować – za pomocą konsoli MMC – zasady zabezpieczeń lokalnych, zasady haseł oraz zasady blokady konta użytkowników.

Użytkownicy serwera SQL Server są reprezentowani przez loginy, konta użytkowników oraz role. Na poziomie instancji serwera mamy do dyspozycji

loginy, role serwera (stałe i definiowane przez administratora) oraz dodatkowe poświadczenia. Dodatkowe poświadczenia pozwalają uwierzytelniać użytkowników (również niemających kont w systemie operacyjnym), łączyć ich w role i nadawać uprawnienia do zewnętrznych zasobów, na przykład plików. Na poziomie bazy danych dysponujemy kontami użytkowników i rolami baz danych. Służą one do nadawania uprawnień w wybranej bazie danych, a jeden login serwera może być połączony z kontami użytkowników w wielu bazach. Użytkownik, który uwierzytelnił się na podstawie loginu niepowiązanego w danej bazie z żadnym kontem użytkownika, uzyska do niej dostęp, tylko jeżeli włączone jest w niej konto gościa. Od wersji 2012 możliwe jest również tworzenie kont użytkowników bazy danych niepowiązanych z loginem, za to chronionych hasłem.

Użytkownik uwierzytelniony podczas połączenia z serwerem SQL może wykonać tylko te operacje, do których posiada uprawnienia. Uprawnienia dzielą się na nadawane do konkretnych obiektów oraz do wykonywania wskazanych instrukcji (np. prawo do wykonywania zewnętrznych skryptów). Biorąc pod uwagę liczbę obiektów bazodanowych i kont użytkowników, zarządzanie uprawnieniami na poziomie poszczególnych kont użytkowników jest nieefektywne i niebezpieczne – administrator nie będzie mógł na bieżąco nadawać i odbierać uprawnień i w końcu wszystkim użytkownikom przyzna pełne uprawnienia. Rozwiązanie tego problemu polega na wykorzystaniu hierarchii obiektów oraz ról serwera i baz danych.

Obiekty serwera SQL Server tworzą hierarchię, a dostęp do każdego z nich jest kontrolowany przez listy ACL (ang. *access control list*). W większości przypadków obiekty podrzędne dziedziczą uprawnienia po obiektach nadrzędnych:

- na poziomie instancji serwera SQL znajdują się m.in. loginy, role serwera, urządzenia kopii zapasowych, wyzwalacze DDL i logowania, główny klucz

usługi SMK oraz powiązane serwery; ponieważ loginy są zarówno principiami, jak i obiektami, możemy nadawać uprawnienia do loginów, na przykład pozwolić wskazanym osobom na ich blokowanie czy zmienianie haseł;

- na poziomie bazy danych znajdują się schematy, typy danych, konta użytkowników, role bazy danych (do których domyślnie należą niektóre loginy i role serwera), certyfikaty, klucze kryptograficzne, wyzwalacze DDL, biblioteki kodu zarządzanego i schematy XML;
- na poziomie schematów znajdują się pozostałe obiekty bazodanowe, w tym tabele, widoki, procedury, funkcje użytkownika, wyzwalacze DML, synonimy i kolejki usługi Service Broker.

Uprawnienie może zostać użytkownikowi nadane, nadane z możliwością przekazania, odebrane lub nieustalone. Brak uprawnienia, tak samo jak jego jawne odebranie, oznacza, że użytkownik nie będzie mógł wykonać danej operacji. Różnica między odebraniem uprawnienia a jego brakiem polega na tym, że brakujące uprawnienie może być odziedziczone (np. jeżeli zostało nadane roli, do której należy użytkownik), natomiast uprawnienie odebrane użytkownikowi oznacza, że nie wykona on danej operacji, niezależnie od ról, do których należy.

Do tej pory łączyliśmy się z serwerem SQL Server jako jego administrator – nasze konto zostało dodane do roli administratorów serwera podczas jego instalacji. Teraz zobaczymy, jak wymagane uprawnienia nadać analitykowi.

Na potrzeby tego przykładu założyłem, że analityk nie ma konta w systemie operacyjnym i będzie się łączył z serwerem SQL za pomocą loginu i hasła. Proszę pamiętać, że rozwiązanie takie jest niezgodne z zaleceniami firmy Microsoft i obniża poziom bezpieczeństwa. Tym niemniej, żeby nie komplikować niepotrzebnie przykładu, posłużymy się właśnie taką metodą

logowania.

Najpierw włączymy mieszany tryb uwierzytelniania serwera SQL Server. Po kliknięciu prawym przyciskiem myszy widocznej w okienku eksploratora obiektów nazwy serwera należy wybrać opcję *Properties*, przejść do zakładki *Security* i wybrać opcję *SQL Server and Windows Authentication mode*. Ta zmiana wymaga ponownego uruchomienia serwera SQL Server (w menu kontekstowym serwera znajdziemy polecenie *Restart*).

Następnie włączymy funkcję niejawnego uwierzytelniania. W tym celu należy utworzyć login dla grupy *SQLRUserGroup*. Grupa ta została automatycznie założona podczas instalacji usługi SQL Server Machine Learning Services i domyślnie zawiera konto usługi Launchpad (w wersji 2019) albo lokalne konta użytkowników usługi SQL Server Machine Learning Services (w wersjach 2016 i 2017). W ten sposób umożliwimy użytkownikom, którzy nie mają konta w systemie operacyjnym i którzy łączą się z serwerem SQL Server na podstawie loginu i hasła, uruchamianie zewnętrznych skryptów. Najprościej jest wykonać tę operację, wyświetlając raport *ML Services – Configuration.rdl* i klikając przycisk *Configure Instance*.

Po skonfigurowaniu serwera i usługi SQL Server Machine Learning Services utworzymy dla naszego analityka login i powiązane z tym loginem konto użytkownika przykładowej bazy danych ML:

```
USE master
GO
CREATE LOGIN [Analyst] WITH PASSWORD=N'Str0ngP@$$w0rd'
        ,DEFAULT_DATABASE=[ML]
        ,CHECK_EXPIRATION=ON
        ,CHECK_POLICY=ON;
GO

USE ML
GO
CREATE USER [Analyst] FOR LOGIN [Analyst];
```

Sprawdźmy, czy ten użytkownik może wykonywać zewnętrzne skrypty:

```
EXECUTE AS USER = 'Analyst';
GO
EXEC sp_execute_external_script
@language = N'R'
,@script = N'OutputDataSet<- InputDataSet'
,@input_data_1 = N'SELECT 1;'
GO
REVERT;
```

```
Msg 297, Level 16, State 101, Procedure sp_execute_external_script,
Line 1 [Batch Start Line 65]
The user does not have permission to perform this action.
```

Tak jak się spodziewaliśmy, domyślnie żaden użytkownik nie może uruchamiać zewnętrznych skryptów. Nadajmy naszemu analitykowi uprawnienie do wykonywania zewnętrznych skryptów

```
GRANT EXECUTE ANY EXTERNAL SCRIPT
TO [Analyst];
```

Jeżeli jeszcze raz przedstawimy się jak użytkownik Analyst i ponowie sprawdzimy, czy może on wykonywać zewnętrzne skrypty, uzyskamy liczbę 1 zwróconą przez skrypt języka R.

Pozostało nam nadać analitykowi dostęp do danych potrzebnych mu do pracy.

Bez tego próba odczytania danych z dowolnej tabeli skończy się błędem:

```
EXECUTE AS USER = 'Analyst';
GO
SELECT *
FROM [BenfordFraud].[Invoices];
GO
REVERT;
```

```
Msg 229, Level 14, State 5, Line 80
The SELECT permission was denied on the object 'Invoices', database
'ML', schema 'BenfordFraud'.
```

W przykładowej bazie danych tabele, widoki, procedury i funkcje zostały podzielone między schematy reprezentujące obszary biznesowe. Nie tylko poprawia to czytelność bazy, lecz także upraszcza zarządzanie uprawnieniami. Skoro nasz analityk pracuje nad problemem wykrywania oszustw, nadajmy mu uprawnienia do wszystkich danych ze schematu:

```
GRANT SELECT, INSERT, UPDATE, DELETE
ON SCHEMA::[BenfordFraud]
TO [Analyst];
```

Powtórzenie powyższego testu pozwoli nam się przekonać, że od teraz użytkownik może odczytywać dane z tabeli `[BenfordFraud].[Invoices]`.

1.3.4. Wydajność

Pozostało nam zapewnić maksymalną wydajność odczytywania danych potrzebnych do analiz przechowywanych w bazie. Tu pomocne będą indeksy kolumnowe i tryb blokowego przetwarzania danych.

Indeksy kolumnowe

SQL Server, tak jak wiele serwerów baz danych, przechowuje dane w postaci

wierszy – jeden lub więcej wierszy tabeli jest zapisywanych w 8-kilobajtowych jednostkach nazywanych stronami. Takie rozwiązanie sprawdza się w bazach operacyjnych, których użytkownicy na bieżąco wstawiają, modyfikują, usuwają lub odczytują pojedyncze wiersze. Jednak w bazach analitycznych, w których użytkownicy odczytują całe tabele, a następnie grupują odczytane dane i łączą je z danymi z innych tabel, rozwiązanie to jest mało wydajne.

Sposobem na skrócenie czasu odczytania całej tabeli nie jest jej poindeksowanie, ale skompresowanie – jeżeli dane zajmują mniej miejsca (są zapisane na mniejszej liczbie stron), ich odczytanie będzie wymagać mniej operacji IO i w rezultacie będzie szybsze. Aby zmaksymalizować wydajność, współczynnik kompresji danych powinien być jak największy.

W algorytmach kompresji bezstratnej współczynnik kompresji zależy przede wszystkim od kompresowanych danych – im bardziej są różnorodne, tym jest on niższy. Innymi słowy, im więcej jest powtarzających się wartości, tym wyższy współczynnik kompresji danych. Ponieważ dane w kolumnach powtarzają się częściej niż w wierszach (w kolumnie *Państwo* we wszystkich wierszach może występować wartość *Polska*, ale szansa że w kilku kolumnach tego samego wiersza, na przykład w kolumnach *Nazwisko, Imię* i *Kraj* wystąpią te same wartości, jest bliska zeru), to przechowując na stronach kolumny, a nie wiersze, zwiększamy współczynnik kompresji.

Ponadto analitycy rzadko odczytują wszystkie kolumny tabeli – znacznie częściej interesują ich tylko wybrane zmienne. Z uwagi na to, że jednostką odczytu danych w serwerze SQL Server jest strona (odczytywane i zapisywane są strony, a nie wiersze, kolumny czy tabele) podzielenie kolumn między różne strony pozwala odczytać tylko te, na których znajdują się potrzebne dane.

Działanie indeksów kolumnowych najlepiej wyjaśnić na prostym przykładzie. Weźmy kilka wierszy przykładowej tabeli faktów (tabeli, w której są przechowywane miary, np. ilość i wartość sprzedaży) oraz klucze wiążące ją

z tabelami wymiarów (np. z kalendarzem, tabelą zawierającą dane o produktach, sklepach i regionach) (tab. 1.1).

W pierwszej kolejności dane są dzielone poziomo, na bloki liczące maksymalnie milion wierszy każdy; te bloki są nazywane grupami wierszy. Następnie każda grupa wierszy jest dzielona pionowo, na wartości poszczególnych kolumn. W rezultacie powstają segmenty zawierające do miliona wartości każdy (tab. 1.2).

Tabela 1.1. Przykładowa tabela z danymi sprzedażowymi

OrderDateKey	ProductKey	StoreKey	RegionKey	Quantity	SalesAmount
20101107	106	1	1	6	30
20101107	103	4	2	1	17
20101107	109	4	2	2	20
20101107	103	3	2	1	17
20101107	106	5	3	4	20
20101108	106	2	1	5	25
20101108	102	2	1	1	14
20101108	106	3	2	5	25
20101108	109	1	1	1	10
20101109	106	4	2	4	20
20101109	106	4	2	5	25
20101109	103	1	1	1	17

Tabela 1.2. Dwie grupy wierszy, każda podzielona na 6 segmentów

OrderDateKey	ProductKey	StoreKey	RegionKey	Quantity	SalesAmount
20101107	106	1	1	6	30
20101107	103	4	2	1	17
20101107	109	4	2	2	20
20101107	103	3	2	1	17
20101107	106	5	3	4	20
20101108	106	2	1	5	25
OrderDateKey	ProductKey	StoreKey	RegionKey	Quantity	SalesAmount

20101108		102		2		1		1	14
20101108		106		3		2		5	25
20101108		109		1		1		1	10
20101109		106		4		2		4	20
20101109		106		4		2		5	25
20101109		103		1		1		1	17

Każdy segment zostaje osobno skompresowany. Kompresja przebiega w kilku etapach. To, które operacje zostaną wykonane dla danego segmentu, zależy głównie od typu kompresowanych danych:

- wartości liczbowe zostają przeskalowane tak, aby możne je było zapisać na jak najmniejszej liczbie bitów; jeżeli na przykład najmniejszą wartością w segmencie było 10 000, od każdej wartości zostanie odjęte 10 000;
- wartości tekstowe mogą zostać zakodowane metodą jeden do wielu – powoduje ona zastąpienie jednej zmiennej tyloma zmiennymi binarnymi, ile różnych wartości występowało w oryginalnej kolumnie; na przykład przyjmującą cztery wartości kolumnę kolor należałoby zastąpić czteroma kolumnami: czarny, czerwony, srebrny i biały, z których każdy przyjmowałby wartość 1 lub 0;
- na etapie kodowania długości (ang. *Run-Length Encoding*) serii poszczególne wartości zostają zastąpione wartością serii uzupełnioną o liczbę jej powtórzeń (tab. 1.3);

Tabela 1.3. Wartości zakodowane algorytmem RLE

Oryginalna wartość	Wartość po zakodowaniu RLE
1	1:2
1	2
2	3:3
3	4
3	5:2

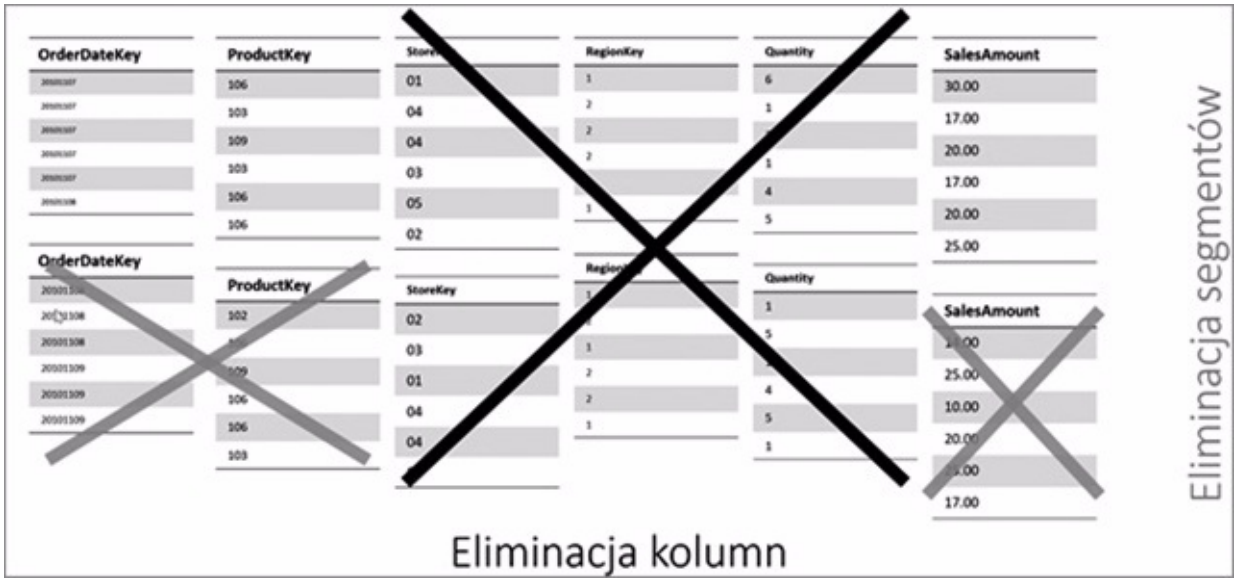
3	
4	
1	
5	
1	
5	

- kolejnym etapem jest kodowanie słownikowe, które polega na zastąpieniu oryginalnej wartości (np. imienia Marcin) identyfikatorem (np. liczbą 1); dane mogą też zostać skompresowane kodowaniem Huffmana [42];
- na końcu następuje kodowanie binarne za pomocą opatentowanego przez Microsoft, niejawnego algorytmu, w efekcie którego dane są konwertowane do postaci binarnej i zapisywane na 8-kilobajtowych stronach;
- opcjonalnie, użytkownik może dodatkowo skompresować wybrane dane za pomocą algorytmu XPress8 będącego nieco zmodyfikowaną wersją algorytmu LZ77, takiego jak używany do kompresji ZIP [42].

Taki sposób przechowywania danych pozwala ograniczyć liczbę stron odczytywanych podczas wykonywania zapytań – mechanizm eliminacji kolumn ogranicza odczytywane strony do tych, na których znajdują się wymagane kolumny, a mechanizm eliminacji segmentów dodatkowo ogranicza odczytywane strony do tych, na których znajdują się żądane wartości. Na przykład do wykonania zapytania

```
SELECT ProductKey, SUM (SalesAmount)
FROM SalesTable
WHERE OrderDateKey < 20101108;
```

wystarczy odczytać 3 z 12 stron przykładowej tabeli (rys. 1.10).



Rysunek 1.10. Oprócz wysokiego współczynnika kompresji indeksy kolumnowe zwiększają wydajność zapytań przez ograniczenie liczby odczytywanych stron danych

Skompresowane segmenty oraz opisujące je słowniki są niezmienne, co oznacza, że zapisane w nich dane nie są modyfikowane. Mimo to indeksy kolumnowe nie uniemożliwiają modyfikowania poindeksowanych danych – w tym celu oprócz segmentów zawierają one magazyny zmian.

Magazyn zmian przechowuje wiersze wstawione oraz zmienione po ostatnim przebudowaniu indeksu. Dane te są zapisane w postaci drzewa zrównoważonego. Osobne drzewo zrównoważone zawiera bitmapę usuniętych wierszy. Razem wzięte te struktury danych pozwalają na modyfikowanie poindeksowanych danych:

- podczas wykonywania instrukcji `INSERT` wstawiane wiersze trafiają do aktualnie używanego (mającego status otwartego) magazynu wersji;
- sposób usunięcia wierszy (wykonania instrukcji `DELETE`) zależy od tego, gdzie znajduje się usuwany wiersz:
  - jeżeli znajduje się w magazynie zmian, jest z niego usuwany,
  - jeżeli jest w grupie wierszy, do bitmapy usuniętych wierszy jest

dodawany jego identyfikator;

- zmiany (instrukcje `UPDATE`) są przeprowadzane jako kombinacje operacji `DELETE` i `INSERT`.

Odczytując dane z indeksu, SQL Server sprawdza, czy nie znajdują się one w bitmapie usuniętych wierszy – jeżeli tak, nie zostaną one zwrócone; oraz czy magazyn zmian nie zawiera zaktualizowanych rekordów – jeżeli tak, są one zwracane użytkownikowi.

Ponieważ magazyn zmian jest przechowywany w postaci wierszowej, taki sposób działania oznacza, że im jest on większy, tym mniejsza jest wydajność zapytań. Dlatego specjalny proces automatycznie przenosi dane z magazynu zmian do segmentów indeksu kolumnowego – co 5 minut sprawdza on, czy do magazynu zmian nie zostało wstawionych milion rekordów za pomocą instrukcji `INSERT` lub 100 tys. rekordów za pomocą instrukcji `BULK INSERT`. Jeżeli tak, dane są przenoszone do segmentów (kompresowane).

Proces przenoszenia danych z magazynu zmian do segmentów można też uruchomić samodzielnie instrukcjami `ALTER TABLE ... REBUILD`, `ALTER INDEX ... REBUILD` lub `REORGANIZE` z opcją `COMPRESS_ALL_ROW_GROUPS`.

Tryb blokowego przetwarzania danych

W trybie wierszowego przetwarzania danych każdy operator wykonywany przez SQL Server pobiera dane za pomocą funkcji *GetNext*. Ponieważ plan wykonania zapytania może zawierać wiele jednocześnie działających operatorów, pobranie jednego rekordu może wymagać od kilku do kilkudziesięciu wywołań funkcji *GetNext*. Jeżeli przetwarzanych danych jest dużo, taki tryb okazuje się mało wydajny.

W trybie blokowym są przetwarzane nie pojedyncze wartości, ale wektory

zawierające około 900 wartości. Pozwala to skrócić czas wykonywania zapytań od 10 do 100 razy.

Podsumowując, typowym wąskim gardłem wydajności serwerów baz danych są dyski. **Indeksy kolumnowe rozwiązują problem wydajnego odczytywania dużych zbiorów danych**. Po pierwsze, skompresowane dane zajmują wielokrotnie (ok. 10-krotnie) mniej miejsca na dysku, co pozwala ograniczyć liczbę operacji I/O niezbędnych do ich wczytania do pamięci. Po drugie, struktura indeksów kolumnowych umożliwia odczytywanie tylko potrzebnych do wykonania zapytania kolumn i segmentów, w dodatku za pomocą 16-krotnie większych operacji odczytu. Po trzecie, skompresowane dane zajmują mniej miejsca w pamięci operacyjnej, co pozwala serwerowi na buforowanie większej ilości danych i przekłada się na zmniejszenie liczby odczytów z dysku. Po ograniczeniu wpływu dysków na wydajność kolejnym wąskim gardłem okazuje się procesor. Tutaj rozwiązaniem okazał się tryb blokowego przetwarzania danych.

Praca z indeksami kolumnowymi

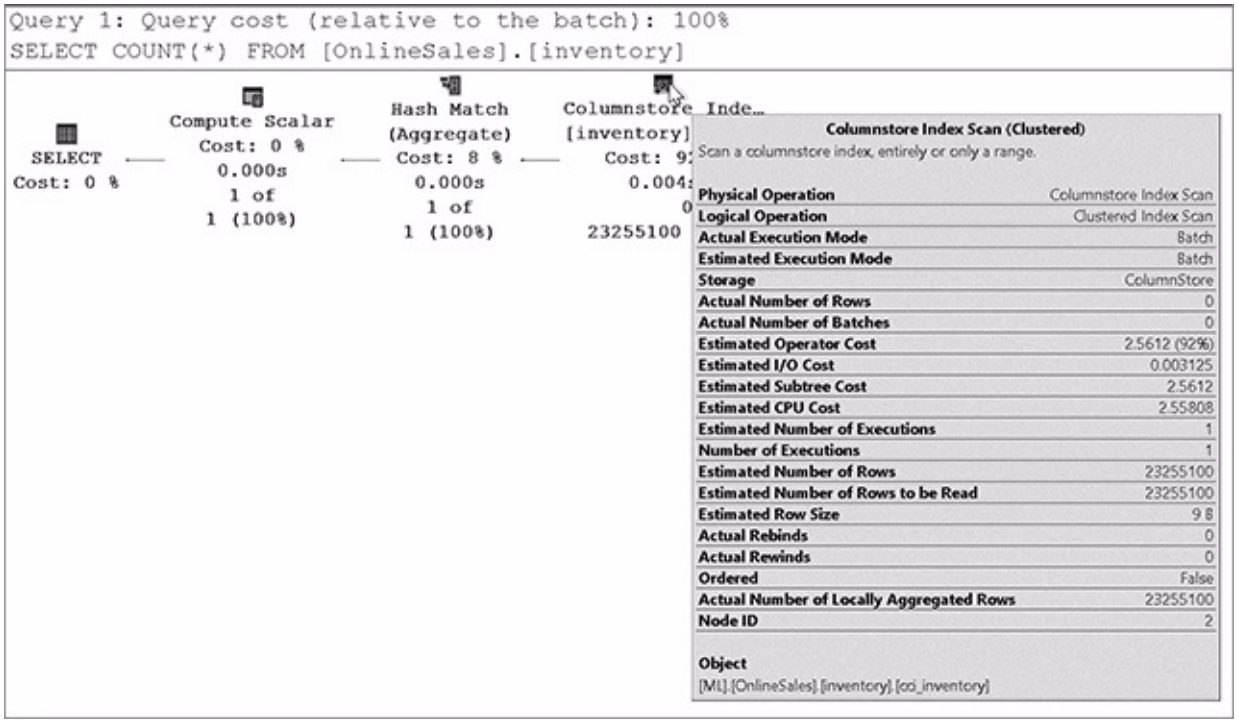
Przekonajmy się, jak działają opisane w tym punkcie funkcje. Zacznijmy od policzenia wierszy największej tabeli w przykładowej bazie danych, sprawdzając przy okazji czas potrzebny na odczytanie całej tabeli

```
SET STATISTICS TIME ON
GO
SELECT COUNT(*)
FROM [OnlineSales].[inventory];
GO
SET STATISTICS TIME OFF
```

```
23255100
SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 5 ms.
```

Okazuje się, że odczytanie ponad 23 mln wierszy zajęło 5 ms. Aby przekonać

się, jak SQL Server odczytał tak dużo danych w tak krótkim czasie, wykonajmy to zapytanie jeszcze raz, tym razem z włączoną opcją *Include Actual Execution Plan* (rys. 1.11).



Rysunek 1.11. Odczytanie danych było najkosztowniejszą operacją (92% kosztu wykonania zapytania). Dzięki skonwertowaniu tabeli do indeksu kolumnowego oraz blokowemu przetwarzaniu danych odczytanie 23 mln wierszy zajęło kilka milisekund

Sprawdźmy, ile czasu zajmują skompresowane dane i jaka byłaby wielkość tej samej tabeli z nieskompresowanymi danymi. Oszacować stopień kompresji tabeli pozwala procedura `sp_estimate_data_compression_savings`

```
EXEC sp_estimate_data_compression_savings
        @schema_name = 'OnlineSales',
        @object_name ='inventory',
        @index_id = NULL,
        @partition_number = NULL,
        @data_compression = 'NONE';
```

```
object_name      schema_name       index_id partition_number
size_with_current_compression_setting(KB)
size_with_requested_compression_setting(KB)
sample_size_with_current_compression_
setting(KB)      sample_size_with_requested_compression_setting(KB)
inventory        OnlineSales       1        1        78032    848000   40232    437216
```

Okazuje się, że 850 MB danych udało nam się skompresować do 78 MB.

W praktyce procedurę `sp_estimate_data_compression_savings` używa się do oszacowania zysku ze skompresowania danych. Sprawdźmy, czy włączenie dla tej tabeli dodatkowej kompresji algorytmem XPress8 byłoby opłacalne.

```
EXEC sp_estimate_data_compression_savings
        @schema_name = 'OnlineSales',
        @object_name ='inventory',
        @index_id = NULL,
        @partition_number = NULL,
        @data_compression = 'COLUMNSTORE_ARCHIVE';
```

```
object_name      schema_name       index_id partition_number
size_with_current_compression_setting(KB)
size_with_requested_compression_setting(KB)
sample_size_with_current_compression_setting(KB)
sample_size_with_requested_compression_setting(KB)
inventory        OnlineSales       1        1        78032    75112    40272    38768
```

Zmniejszenie wielkości tabeli o 3 MB kosztem dodatkowej, wymagającej kosztowych operacji arytmetycznych kompresji wydaje się nieopłacalne.

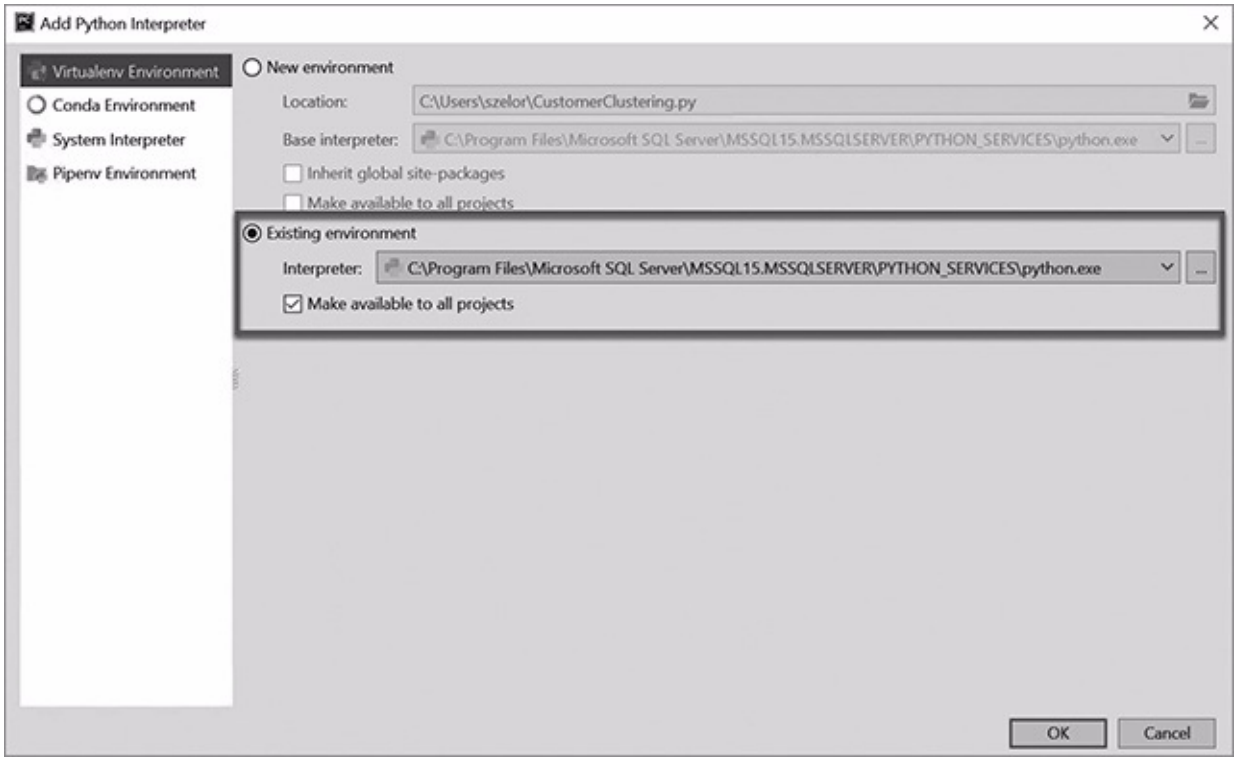
W przykładowej bazie danych duże tabele zostały już skonwertowane do indeksów kolumnowych. Samodzielnie taką operację można przeprowadzić, wykonując instrukcję `CREATE CLUSTERED COLUMNSTORE INDEX`. Ponieważ tabela

może mieć tylko jeden indeks zgrupowany (Clustered), w tym przypadku najpierw go usuniemy.

```
DROP INDEX [CCI_Invoices]
ON [BenfordFraud].[Invoices];
GO
CREATE CLUSTERED COLUMNSTORE INDEX [CCI_Invoices]
ON [BenfordFraud].[Invoices];
```

1.4. PyCharm Community Edition

Edytor PyCharm jest dostępny w dwóch edycjach: komercyjnej i darmowej. Wersje przeznaczone dla systemów Windows, Linux i MacOS można pobrać ze strony https://www.jetbrains.com/pycharm/download/. Po zainstalowaniu i uruchomieniu należy wskazać lokalizację interpretera języka Python usługi SQL Server Machine Learning Services (rys. 1.12).



Rysunek 1.12. Interpretator języka Python można też wybrać podczas uruchamiania poszczególnych skryptów

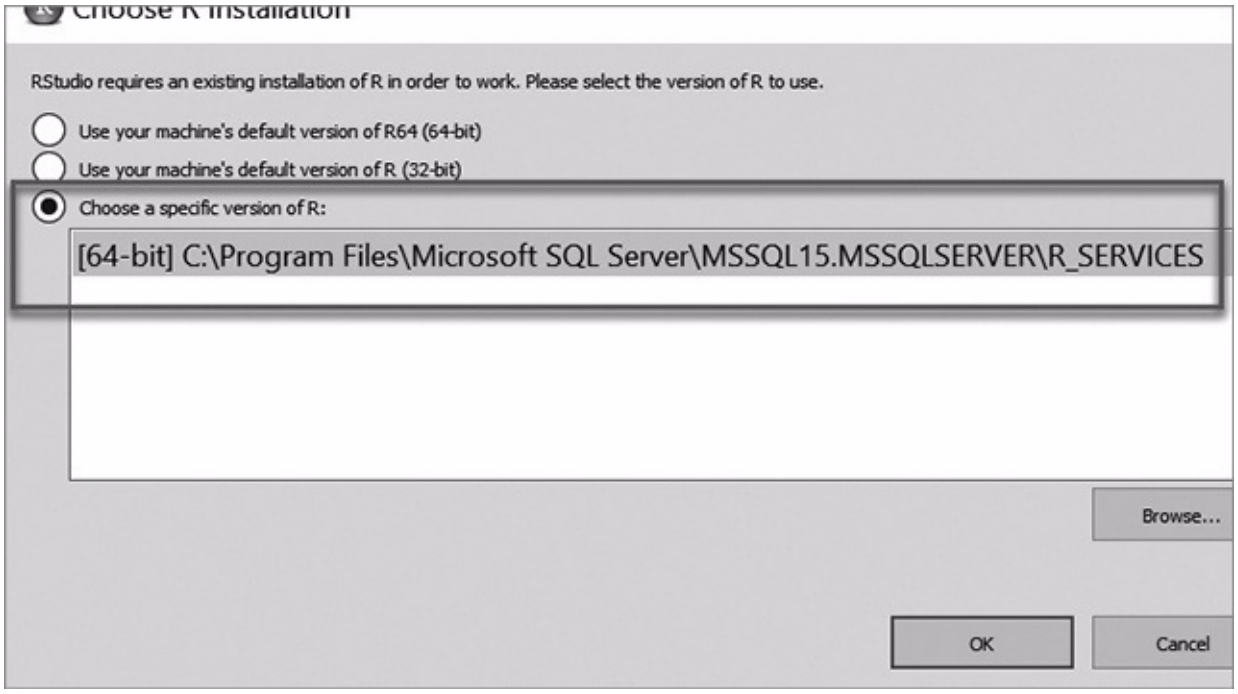
1.5. RStudio Desktop

RStudio jest popularnym, graficznym edytorem języka R. Jego darmową wersję Desktop można pobrać pod adresem https://www.rstudio.com/products/rstudio/download/. Pozwala on pisać, uruchamiać i debugować skrypty R. RStudio zawiera też narzędzia do zarządzania sesjami i pakietami, wyświetlania grafiki czy korzystania z pomocy. Graficzny przewodnik po tym edytorze jest dostępny pod adresem https://www.rstudio.com/resources/cheatsheets/.

Przy pierwszym uruchomieniu RStudio zostaniemy poproszeni o wskazanie folderu, w którym znajduje się interpretator języka R (rys. 1.13).

Po wskazaniu prawidłowego folderu w okienku konsoli zostaną wyświetlone najważniejsze informacje o używanej wersji języka R:

```
R version 3.4.4 (2018-03-15) -- "Someone to Lean On"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)
```



Rysunek 1.13. Chociaż na tym samym komputerze można zainstalować wiele wersji języka R, przełączanie się między nimi prowadzi do problemów z kompatybilnością pakietów

Przedpremierowa, użyta przeze mnie wersja SQL Server Machine Learning Services zainstalowała wersję 3.4.4 języka R. Poniżej zostanie wyświetlony komunikat informujący o korzystaniu z dystrybucji MRO:

```
Microsoft R Open 3.4.4
The enhanced R distribution from Microsoft
Microsoft packages Copyright © 2018 Microsoft
```

Następnie dowiemy się, czy załadowane zostaną pakiety w wersji MRC, czy MLS. Jeżeli zobaczymy poniższy komunikat, będzie to znaczyło, że pracujemy z edycją Standard serwera SQL Server:

```
Loading Microsoft R Client packages, version 3.4.4.0105.
Microsoft R Client limits some functions to available memory.
See: https://go.microsoft.com/fwlink/?linkid=799476 for information
about additional features.
```

Jeżeli komunikat będzie podobny do poniższego, będzie to znaczyło, że

pracujemy z edycją Enterprise lub Developer serwera SQL Server:

```
Loading Microsoft R Server packages, version 9.4.6.
Type 'readme()' for release notes, privacy() for privacy policy, or
'RevoLicense()' for licensing information.
```

Na końcu znajdują się dwie ważne informacje: pierwsza oznacza, że wszystkie opracowane przez Microsoft dystrybucje języka R używają biblioteki Intel MKL, a druga zawiera datę dziennej kopii strony projektu CRAN domyślnie używanej do pobierania pakietów:

```
Using the Intel MKL for parallel mathematical computing (using 8
cores).
Default CRAN mirror snapshot taken on 2018-04-01.
See: https://mran.microsoft.com/.
```

1.6. Instalacja dodatkowych pakietów

Listę zainstalowanych pakietów języka R możemy uzyskać, wywołując funkcję `installed.packages`. W R Studio instrukcje języka R można wpisywać bezpośrednio w konsoli albo w edytorze skryptów. Jeżeli wybierzemy drugą opcję, instrukcje zapisane w edytorze będziemy mogli wykonać, ustawiając w odpowiednim wierszu skryptu kursor myszki i naciskając kombinację klawiszy `Ctrl+Enter`.

Aby poznać najważniejsze informacje o zainstalowanych pakietach, należy wykonać poniższą instrukcję:

```
installed.packages()[,c("Package", "Version", "Depends", "License",
"Built", "LibPath")]
```

Teraz, kiedy upewniliśmy się, że nasza instrukcja działa prawidłowo, możemy ją

skopiować do procedury `sp_execute_external_script`. Jedyną modyfikacją wymaganą w tym przypadku jest skonwertowanie wyniku do postaci tabelarycznej. **Wszystkie skrypty języków R i Python uruchamiane po stronie serwera SQL Server muszą zwracać dane tabelaryczne albo nie zwracać żadnych wyników**. Dodatkowo sprawdzimy, czy wykonywanie zewnętrznych skryptów zostało włączone i określimy nazwy oraz typy wynikowych kolumn. W oknie edytora zapytań konsoli SSMS możemy wykonać zaznaczone instrukcje, naciskając klawisz `F5` (rys. 1.14).

Dodatkowe pakiety języka R możemy zainstalować na kilka sposobów. W wersji 2019 serwera SQL Server instalacja pakietów języka R nie wymaga uprawnień administratora systemu.

Pierwszy sposób polega na wywołaniu funkcji `install.packages` w edytorze R Studio.

```
install.packages("ggplot2")
```

```
also installing the dependencies 'glue', 'magrittr', 'stringi', 'colorspace',
'assertthat', 'utf8', 'Rcpp', 'stringr', 'RColorBrewer', 'dichromat', 'munsell',
'labeling', 'viridisLite', 'cli', 'crayon', 'pillar', 'rlang', 'digest', 'gtable', 'plyr',
'reshape2', 'scales', 'tibble', 'lazyeval'
trying URL 'https://mran.microsoft.com/snapshot/2018-04-01/bin/windows/contrib/3.4/
glue_1.2.0.zip'
Content type 'application/zip' length 75842 bytes (74 KB)
downloaded 74 KB
…
package 'ggplot2' successfully unpacked and MD5 sums checked
```

```
-- Get list of installed R packages:
if (select CAST(SERVERPROPERTY('IsAdvancedAnalyticsInstalled') as int) & CAST(value_in_use as int)
    from sys.configurations
   where name = 'external scripts enabled') = 1
begin
 exec sp_execute_external_script
   @language = N'R'
  ,@script = N'
 OutputDataSet <- data.frame(installed.packages()[,c("Package", "Version", "Depends", "License", "Built", "LibPath")]);'
 with result sets
   ((Package nvarchar(255), Version nvarchar(100), Depends nvarchar(4000), License nvarchar(1000), Built nvarchar(100), LibPath nvarchar(2000)));
end;
go
```

	Package	Version	Depends	License	Built	LibPath
19	iterators	1.0.9	R (>= 2.5.0), utils	Apache License (== 2.0)	3.3.1	C:/SQL-MSSQLSERVER-R_SERVICES/library
20	jsonlite	1.5	methods	MIT + file LICENSE	3.4.4	C:/SQL-MSSQLSERVER-R_SERVICES/library
21	KernSmooth	2.23-15	R (>= 2.5.0), stats	Unlimited	3.4.4	C:/SQL-MSSQLSERVER-R_SERVICES/library
22	lattice	0.20-35	R (>= 3.0.0)	GPL (>= 2)	3.4.4	C:/SQL-MSSQLSERVER-R_SERVICES/library
23	MASS	7.3-49	R (>= 3.1.0), grDevices, graphics, stats, utils	GPL-2 | GPL-3	3.4.4	C:/SQL-MSSQLSERVER-R_SERVICES/library
24	Matrix	1.2-12	R (>= 3.0.1)	GPL (>= 2) | file LICE...	3.4.4	C:/SQL-MSSQLSERVER-R_SERVICES/library
25	methods	3.4.4	NULL	Part of R 3.4.4	3.4.4	C:/SQL-MSSQLSERVER-R_SERVICES/library
26	mgcv	1.8-23	R (>= 2.14.0), nlme (>= 3.1-64)	GPL (>= 2)	3.4.4	C:/SQL-MSSQLSERVER-R_SERVICES/library
27	MicrosoftML	9.4.6	R (>= 3.3.2), methods, RevoScaleR (>= 9.2.1)	file LICENSE	3.4.4	C:/SQL-MSSQLSERVER-R_SERVICES/library
28	MicrosoftR	3.4.4....	NULL	file LICENSE	3.4.2	C:/SQL-MSSQLSERVER-R_SERVICES/library
29	mrsdeploy	1.1.3	R (>= 3.3.0)	file LICENSE	3.4.4	C:/SQL-MSSQLSERVER-R_SERVICES/library
30	mrupdate	1.0.1	R (>= 3.2.2)	file LICENSE	3.3.1	C:/SQL-MSSQLSERVER-R_SERVICES/library
31	nlme	3.1-1...	R (>= 3.3.0)	GPL (>= 2) | file LICE...	3.4.4	C:/SQL-MSSQLSERVER-R_SERVICES/library
32	nnet	7.3-12	R (>= 2.14.0), stats, utils	GPL-2 | GPL-3	3.4.4	C:/SQL-MSSQLSERVER-R_SERVICES/library
33	olapR	1.0.0	R (>= 3.2.2)	file LICENSE	3.4.4	C:/SQL-MSSQLSERVER-R_SERVICES/library
34	parallel	3.4.4	NULL	Part of R 3.4.4	3.4.4	C:/SQL-MSSQLSERVER-R_SERVICES/library
35	png	0.1-7	R (>= 2.9.0)	GPL-2 | GPL-3	3.4.4	C:/SQL-MSSQLSERVER-R_SERVICES/library
36	R6	2.2.2	R (>= 3.0)	MIT + file LICENSE	3.4.4	C:/SQL-MSSQLSERVER-R_SERVICES/library
37	RevoIOQ	8.0.10	R (>= 2.6.2), RUnit (>= 0.4.26), tools	GPL-2	3.4.3	C:/SQL-MSSQLSERVER-R_SERVICES/library
38	RevoMods	11.0.0	R (>= 2.6.2)	GPL-2	3.3.3	C:/SQL-MSSQLSERVER-R_SERVICES/library
39	RevoPemaR	10.0.0	R (>= 3.1.1), methods	Apache License 2.0	3.2.2	C:/SQL-MSSQLSERVER-R_SERVICES/library
40	RevoScaleR	9.4.6	R (>= 3.2.2)	file LICENSE	3.4.4	C:/SQL-MSSQLSERVER-R_SERVICES/library
41	RevoTreeView	10.0.0	NULL	file LICENSE	3.2.2	C:/SQL-MSSQLSERVER-R_SERVICES/library
42	RevoUtils	10.0.9	NULL	GPL-2	3.4.4	C:/SQL-MSSQLSERVER-R_SERVICES/library

Rysunek 1.14. W tym przypadku z usługą Microsoft SQL Server Machine Learning Services zostało zainstalowanych 56 pakietów języka R, wszystkie w wersji 3.4.4 lub niższej, bo używamy kopii strony projektu CRAN z 1 kwietnia 2018 r.

Możemy też skorzystać z wchodzącej w skład pakietu RevoScaleR funkcji `rxInstallPackages`:

```
packagesToInstall <- c("Hmisc","AUC")
RSqlServerCC <- RxInSqlServer(connectionString = "Driver=SQL Server;
Server=MS;Database=ML;Trusted_Connection=True;")
```

```
rxInstallPackages(pkgs = packagesToInstall, scope = "shared")
trying URL 'https://mran.microsoft.com/snapshot/2018-04-01/bin/windows/contrib/3.4/
Hmisc_4.1-1.zip'
Content type 'application/zip' length 1819242 bytes (1.7 MB)
downloaded 1.7 MB

trying URL 'https://mran.microsoft.com/snapshot/2018-04-01/bin/windows/contrib/3.4/
AUC_0.3.0.zip'
Content type 'application/zip' length 46844 bytes (45 KB)
downloaded 45 KB
package 'Hmisc' successfully unpacked and MD5 sums checked
package 'AUC' successfully unpacked and MD5 sums checked
```

Z usługą SQL Server Machine Learning Services jest instalowany interpretator R.exe. Pozwala on uruchamiać instrukcje języka R, a więc możemy go użyć do instalacji dodatkowych pakietów. Program R.exe można uruchomić bezpośrednio w oknie wiersza polecenia:

```
C:\Program Files\Microsoft SQL Server\MSSQL15.MSSQLSERVER\R_SERVICES\bin\x64> R -e install.
packages(,tidyverse')
```

```
also installing the dependencies 'praise', 'withr', 'backports', 'mnormt', 'bindr',
'rematch', 'debugme', 'testthat', 'evaluate', 'highr', 'markdown', 'yaml', 'htmltools',
'base64enc', 'rprojroot', 'psych', 'bindrcpp', 'pkgconfig', 'BH', 'plogr', 'DBI',
'tidyselect', 'mime', 'openssl', 'cellranger', 'callr', 'knitr', 'rmarkdown', 'whisker',
'selectr', 'broom', 'dplyr', 'dbplyr', 'forcats', 'haven', 'hms', 'httr', 'lubridate',
'modelr', 'purrr', 'readr', 'readxl', 'reprex', 'rstudioapi', 'rvest', 'tidyr', 'xml2'

  There is a binary version available but the source version is later:
           binary source needs_compilation
lubridate   1.7.2   1.7.3                TRUE

  Binaries will be installed
trying URL 'https://mran.microsoft.com/snapshot/2018-04-01/bin/windows/contrib/3.4/
praise_1.0.0.zip'
Content type 'application/zip' length 17952 bytes (17 KB)
==================================================
downloaded 17 KB
…
package 'tidyverse' successfully unpacked and MD5 sums checked
```

Od wersji 2017 możliwe jest też utworzenie po stronie serwera SQL Server zewnętrznej biblioteki zawierającej dodatkowe pakiety języka R:

```
CREATE EXTERNAL LIBRARY AUC
FROM (CONTENT = 'C:\ML\Chapter01\AUC_0.3.0.zip')
WITH (LANGUAGE = 'R');
```

Pakiety znajdujące się w zewnętrznej bibliotece są dostępne, tak samo jak ręcznie zainstalowane pakiety:

```
EXEC sp_execute_external_script
       @language =N'R',
       @script=N'library(AUC)'
```

```
STDERR message(s) from external script:
AUC 0.3.0
Type AUCNews() to see the change log and ?AUC to get an overview.
```

Listę zainstalowanych pakietów języka Python zwraca poniższy skrypt:

```
import pip
import pandas

OutputDataSet = pandas.DataFrame([(i.key, i.version, i.location)
                                  for i in pip.get_installed_distributions()])
print(OutputDataSet
```

W edytorze PyCharm do wykonania skryptu służy kombinacja klawiszy `Alt+Shift+F10`.

Uruchomienie tego skryptu po stronie serwera SQL Server wymaga dwóch zmian. Tym razem wynik skryptu jest już w postaci tabelarycznej, ale jest on wyświetlany w konsoli funkcją `print` – ostatni wiersz należy usunąć. Poza tym, w skrypcie importujemy pakiet `pandas`, który jest automatycznie importowany przez środowisko uruchomieniowe języka Python – tę instrukcję również należy usunąć. Uruchomienie tego skryptu przez procedurę `sp_execute_external_script` wygląda więc następująco:

```
exec sp_execute_external_script
  @language = N'Python'
  ,@script = N'
import pip
OutputDataSet = pandas.DataFrame([(i.key, i.version, i.location) for i in pip.
get_installed_distributions()])'
 with result sets ((Package nvarchar(128), Version nvarchar(128), Location
nvarchar(1000)));
```

```
blaze   0.10.1   c:\program files\microsoft sql server\mssql15.mssqlserver\python_services\
lib\site-packages
bokeh   0.12.5   c:\program files\microsoft sql server\mssql15.mssqlserver\python_services\
lib\site-packages
bottleneck      1.2.0    c:\program files\microsoft sql server\mssql15.mssqlserver\
python_services\lib\site-packages
cffi    1.9.1    c:\program files\microsoft sql server\mssql15.mssqlserver\python_services\
lib\site-packages
…
```

Okazuje się, że z usługą Microsoft SQL Server Machine Learning Services jest instalowanych ponad 160 pakietów języka Python. Jeżeli na tej liście zabraknie potrzebnego nam pakietu, możemy go zainstalować, uruchamiając działające z uprawnieniami administratora okno wiersza polecenia i wykonując poniższe instrukcje:

```
cd C:\Program Files\Microsoft SQL Server\MSSQL15.MSSQLSERVER\PYTHON_SERVICES\Scripts\
pip.exe install <nazwa pakietu>
```

Instalacja niektórych pakietów może wymagać ich kompilacji – na przykład wtedy, gdy próba zainstalowania pakietu Scrapy skończy się poniższym błędem:

```
error: Microsoft Visual C++ 14.0 is required. Get it with "Microsoft Visual C++ Build
Tools": http://landinghub.visualstudio.com/
visual-cpp-build-tools
```

W takim przypadku należy pobrać i zainstalować kompilator języka Microsoft Visual C++ 14.0, po czym ponowić instalację pakietu.

1.7.
Power BI Desktop

Usługa Power BI pozwala użytkownikom biznesowym samodzielnie analizować i wizualizować dane pochodzące z różnych źródeł. Jednym z jej elementów jest darmowy program Power BI Desktop. Pozwala on w prosty sposób tworzyć biznesowe modele danych i interaktywne raporty. Pod adresem https://powerbi.microsoft.com/pl-pl/desktop/ znajdziemy program Power BI Desktop i opis jego możliwości. W książce program Power BI Desktop został użyty do graficznej oceny danych oraz do zaprezentowania wyników modeli predykcyjnych.

Niektórzy najlepiej uczą się na przykładach. Tak samo jak niektóre maszyny.

Marcin Szeliga

Rozdział 2

Praca z SQL Server Machine Learning Services

W tym rozdziale podaję dwa przykłady zastosowania SQL Server Machine Learning Services do zaawansowanej analizy danych. Czytelnik nie znajdzie w nim ani słowa na temat teorii uczenia maszynowego. Tak jak w poprzednim, w tym **rozdziale omawiam narzędzia, a nie techniki uczenia maszynowego**. Różnica między rozdziałami polega na sposobie przedstawiania tych samych zagadnień – tym razem SQL Server Machine Learning Services zostało użyte do rozwiązania dwóch typowych problemów.

2.1. Wykrywanie oszustw

Wykrywanie oszustw było jednym z pierwszych praktycznych zastosowań zaawansowanej analityki. My też swoją przygodę z SQL Server Machine Learning Services zaczniemy od wykrycia prób oszustwa. Naszym zadaniem będzie wytypowanie firm wystawiających naszemu zleceniodawcy fałszywe faktury. Jedyne czym dysponujemy, to dane o fakturach wystawionych zleceniodawcy.

```
SELECT *
FROM [BenfordFraud].[Invoices];
```

```
VendorNumber    VoucherNumber    CheckNumber    InvoiceNumber    InvoiceDate
PaymentDate     DueDate  InvoiceAmount    PONumber
387707  62286   56229    8016    2005-07-19      2005-08-18       2005-08-18
9844.46 9872
387707  61574   56157    9384    2005-10-27      2005-12-02       2005-11-26
4614.70 6598
387707  97399   59740    8732    2005-09-12      2005-10-15       2005-10-12
2569.51 4045
387707  97680   59768    5769    2005-02-07      2005-02-28       2005-03-09
4185.07 6230
…
```

Każda faktura jest opisana w osobnym wierszu tabeli `[BenfordFraud].[Invoices]`; tabela liczy 245 830 wierszy. Chociaż danych mamy sporo, wykrycie w nich podejrzanych firm wymaga kreatywności i przeprowadzenia zaawansowanej analizy.

Spróbujmy porównać faktyczny rozkład wartości wybranej zmiennej z jej spodziewanym rozkładem. W określeniu spodziewanego rozkładu pomoże nam znajomość rozkładu Benforda. Opisuje on rozkład prawdopodobieństwa występowania pierwszych cyfr w wielu różnych danych statystycznych. Zgodne z rozkładem Benforda są m.in. częstości występowania pierwszych cyfr w rocznikach statystycznych i stałych fizycznych, długościach rzek, wielkościach miast, liczbie mieszkańców krajów itp.

Fenomen ten po raz pierwszy (w 1881 r.) opisał astronom Simon Newcomb [11]. Zauważył on, że pierwsze strony tablic logarytmicznych są bardziej wyświechtane, co sugerowałoby, że są częściej używane. Fakt występowania tego rozkładu w obserwowanych danych został potwierdzony w 1938 r. przez fizyka Franka Benforda. Po zbadaniu danych z 20 różnych obszarów sformułował on prawo, nazwane od jego nazwiska prawem Benforda [12]: **jeżeli zmienna może przyjmować różne rzędy wielkości, prawdopodobieństwo**

wystąpienia jej pierwszej cyfry *k* wynosi

$$\log_{10}\left(1 + \frac{1}{k}\right)$$

Rozkład Benforda został pokazany na rysunku 2.1.

Obecnie prawo Benforda jest powszechnie używane do wykrywania oszustw podatkowych. Spróbujmy go użyć do sprawdzenia wiarygodności kwot, na jakie poszczególne firmy wystawiły faktury.

Będziemy potrzebować danych o częstości występowania pierwszych cyfr w kwotach wystawionych faktur, czyli takich danych, jakie zawraca funkcja `[BenfordFraud].[VendorInvoiceDigits]`:

```
SELECT *
FROM [BenfordFraud].[VendorInvoiceDigits] (default)
ORDER BY VendorNumber;
VendorNumber     Digits   Freq
105313  1.00     173
105313  2.00     121
105313  3.00     83
105313  4.00     75
105313  5.00     42
105313  6.00     47
105313  7.00     42
105313  8.00     30
105313  9.00     27
105436  1.00     33
105436  2.00     5366
105436  3.00     16
105436  4.00     26
105436  5.00     27
105436  6.00     27
105436  7.00     28
105436  8.00     34
105436  9.00     27
…
```

d	P(d)	Względny rozkład częstości
1	30.1%	
2	17.6%	
3	12.5%	
4	9.7%	
5	7.9%	
6	6.7%	
7	5.8%	
8	5.1%	
9	4.6%	

Rysunek 2.1. Zgodnie z rozkładem Benforda cyfra 1 występuje jako pierwsza sześciokrotnie częściej niż cyfra 8

Dysponując danymi o częstości występowania cyfr, możemy sprawdzić, czy są one zgodne ze spodziewanymi, tj. czy częstość występowania poszczególnych cyfr odpowiada rozkładowi Benforda. W tym celu użyjemy procedury napisanej w języku R.

```
CREATE OR ALTER PROCEDURE [BenfordFraud].[getPotentialFraudulentVendors] (@threshold float
= 0.05)
AS
BEGIN
        -- Use Benford law to get the potential fraud vendors.
        EXEC sp_execute_external_script
                    @language = N'R',
                    @script = N'
                            library(reshape2);
                            dd = dcast(InputDataSet, VendorNumber ~ Digits, value.
var="Freq");
                            OutputDataSet = cbind(
                                    dd,
                                    apply(dd[,-1], 1, function(xx) chisq.test(xx,
p=(log(1+1/(1:9))/log(10)))$p.value)); ## Equation using Benford law
                            colnames(OutputDataSet) <- c("VendorNumber", "Digit1", "Digit2",
"Digit3", "Digit4", "Digit5", "Digit6", "Digit7", "Digit8", "Digit9", "Pvalue");
                           OutputDataSet <- subset(OutputDataSet, Pvalue < threshold);
                    ',
                    @input_data_1 = N'
        SELECT VendorNumber, Digits, Freq
          FROM [BenfordFraud].VendorInvoiceDigits(default)
        ORDER BY VendorNumber asc, Digits asc;
                    ',
                    @params = N'@threshold float',
                    @threshold = @threshold
        WITH RESULT SETS (( VendorNumber varchar(10), Digit1 int, Digit2 int, Digit3 int,
Digit4 int, Digit5 int, Digit6 int, Digit7 int, Digit8 int, Digit9 int, Pvalue float));
END;
```

Danymi wejściowymi skryptu R jest wynik pokazanego poprzednio zapytania. Dane te są zapisywane do zmiennej `InputDataSet`. Następnie długi (wierszowy) format danych zostaje zastąpiony szerokim (kolumnowym). Taką konwersję można przeprowadzić funkcją `spread` pakietu `tidyr` lub `dcast` pakietu `reshape2`. Do przekształconych danych jest dodawana kolumna zawierająca wynik testu chi-kwadrat, za pomocą którego została sprawdzona zgodność wartości zmierzonej (częstość występowania pierwszych cyfr w kwotach faktur wystawionych przez daną firmę) z oczekiwaną, obliczoną na podstawie prawa Benforda. Kolumny wyniku zostały odpowiednio nazwane, a następnie zostały

usunięte wiersze opisujące firmy, dla których przy przyjętym progu prawdopodobieństwa test chi-kwadrat wypadł negatywnie.

Zwrócone przez procedurę wyniki wstawiamy do tabeli *[BenfordFraud].[FraudulentVendors]*:

```
INSERT INTO [BenfordFraud].[FraudulentVendors]
EXEC [BenfordFraud].getPotentialFraudulentVendors 0.10;
```

Sprawdźmy zawartość tej tabeli:

```
SELECT *
FROM [BenfordFraud].[FraudulentVendors]
ORDER BY Pvalue;
```

VendorNumber	Digit1	Digit2	Digit3	Digit4	Digit5	Digit6	Digit7	Digit8	Digit9	Pvalue
105436	33	5366	16	26	27	27	28	34	27	0
E26417	158	115	84	63	67	59	21	30	32	0.00123
138478	206	110	88	48	38	47	48	35	19	0.04972
731409	210	113	76	78	60	35	37	32	15	0.05481
891424	207	99	75	64	44	45	23	34	41	0.07492
392938	158	111	100	59	43	49	33	34	32	0.08166
828246	212	119	93	48	69	60	37	33	30	0.08282
893509	218	118	62	59	49	48	25	32	23	0.08622

Im mniejsza wartość *P*, tym mniejsze jest prawdopodobieństwo, że pierwsze cyfry w kwotach wystawionych przez daną firmę faktur wystąpiły przypadkowo. Innymi słowy, wartość 1 – *P* wyznacza prawdopodobieństwo, z którym firma wystawiła fałszywe faktury. Udało nam się rozwiązać problem i zaleźć potencjalnych oszustów. Teraz musimy wyjaśnić naszemu zleceniodawcy,

w jaki sposób to zrobiliśmy.

W wyjaśnieniu, dlaczego właśnie te firmy zostały wskazane jako potencjalni oszuści pomoże nam zestawienie na wykresie częstości występowania poszczególnych cyfr na fakturach tych firm ze spodziewanymi, obliczonymi na podstawie prawa Benforda. Procedura generująca taki wykres dla podanego kontrahenta została pokazana poniżej:

```
CREATE OR ALTER PROCEDURE [BenfordFraud].[getVendorInvoiceDigits] (@VendorNumber
varchar(10))
AS
BEGIN
	-- Produces plot for a specific vendor showing the distribution of invoice amount
digits (Actual) vs. Benford distribution for the digit (Expected)
	EXEC sp_execute_external_script
		@language = N'R',
		@script = N'
			require(ggplot2)
			require(ggthemes)
			require(reshape2)

			qq = as.numeric(InputDataSet[,1])
			pp = data.frame(num=factor(1:9),pct=round(100*(log(1+1/(1:9))/
log(10))))
			pp = data.frame(num=pp$num, Actual=round(100*qq/sum(qq)),
Expected=pp$pct)
			pp = melt(pp)
			title = "Distribution of Leading Digits in Invoices"
			gg = ggplot(pp, aes(x=num, y=value, fill=variable)) +
geom_bar(stat="identity", position="dodge", alpha=0.85)
			gg = gg + labs(x="Leading Digit", y="Percent")
			windowsFonts(Verdana="TT Verdana")
			gg = gg + theme_igray(base_size=16, base_family="Verdana")
			gg = gg + theme(legend.title=element_blank())

			ff = tempfile()
			png(filename=ff, width=620, height=240)
			print(gg)
			dev.off()
			OutputDataSet = data.frame(data=readBin(file(ff, "rb"),
what=raw(), n=1e6))
		',
		@input_data_1 = N'select Freq from [BenfordFraud].VendorInvoiceDigits
(@vendor) order by Digits;',
		@params = N'@vendor varchar(10)',
		@vendor = @VendorNumber
	WITH RESULT SETS(([chart] varbinary(max)));
END;
```

Dane wejściowe są takie same jak dla procedury wyszukującej podejrzanych kontrahentów. Tym razem jednak odczytamy z nich częstości występowania poszczególnych cyfr i zapiszemy je w zmiennej `qq`. Następnie policzymy teoretyczne częstości występowania poszczególnych liczb, korzystając z prawa Benforda, i zapiszemy je w tabeli `pp`. Kolumnowy (szeroki) format tej tabeli zastąpimy wierszowym (długim) – w ten sposób uzyskamy łatwe do wizualizacji, uporządkowane dane. Konwersję szerokiego formatu do długiego można przeprowadzić funkcją `melt` pakietu `reshape` lub `gather` pakietu `tidyr`. Tak uzyskane dane przedstawimy na wykresie słupkowym utworzonym funkcją `ggplot` biblioteki `ggplot2`. Wykres ten zapiszemy do pliku tymczasowego `ff` tylko po to, żeby odczytać zawartość tego pliku w formie binarnej i zwrócić ją jako wyjściową tabelę `OutputDataSet`.

Jeżeli wywołanie procedury skończy się podobnym do poniższego błędem, należy zainstalować brakujące pakiety języka R (w tym przypadku pakiet `ggthemes`):

```
In library(package, lib.loc = lib.loc, character.only = TRUE, logical.return = TRUE,  :
  there is no package called 'ggthemes'
```

Udane wywołanie naszej procedury dla wybranego kontrahenta zostało pokazane poniżej:

```
EXEC [BenfordFraud].getVendorInvoiceDigits '105436';
```

```
chart
0x89504E470D0A1A0A0000000D49484452000000026C000000F008030000008A70C806000000D5504C544500000000
003300005C00335C003380005CA322C5C926C9CD33000033003333005C333300333333335C5C3380C54D4D4D
…
```

Procedurę tę należy wywołać dla wszystkich wykrytych podejrzanych kontrahentów, a jej wyniki zapisać w tabeli. Zadanie to wykonuje procedura `[BenfordFraud].[getPotentialFraudulentVendorsList]`. Wywołuje ona w pętli

procedurę generującą wykres dla wybranego kontrahenta i wstawia jego numer do tabeli z wykresami. Najważniejszy fragment tej procedury został pokazany poniżej:

```
        INSERT INTO [BenfordFraud].[FraudulentVendorsPlots] (Plot)
        EXEC [BenfordFraud].getVendorInvoiceDigits @vendor;
UPDATE [BenfordFraud].[FraudulentVendorsPlots]
SET VendorNumber = @vendor
WHERE VendorNumber IS NULL;
```

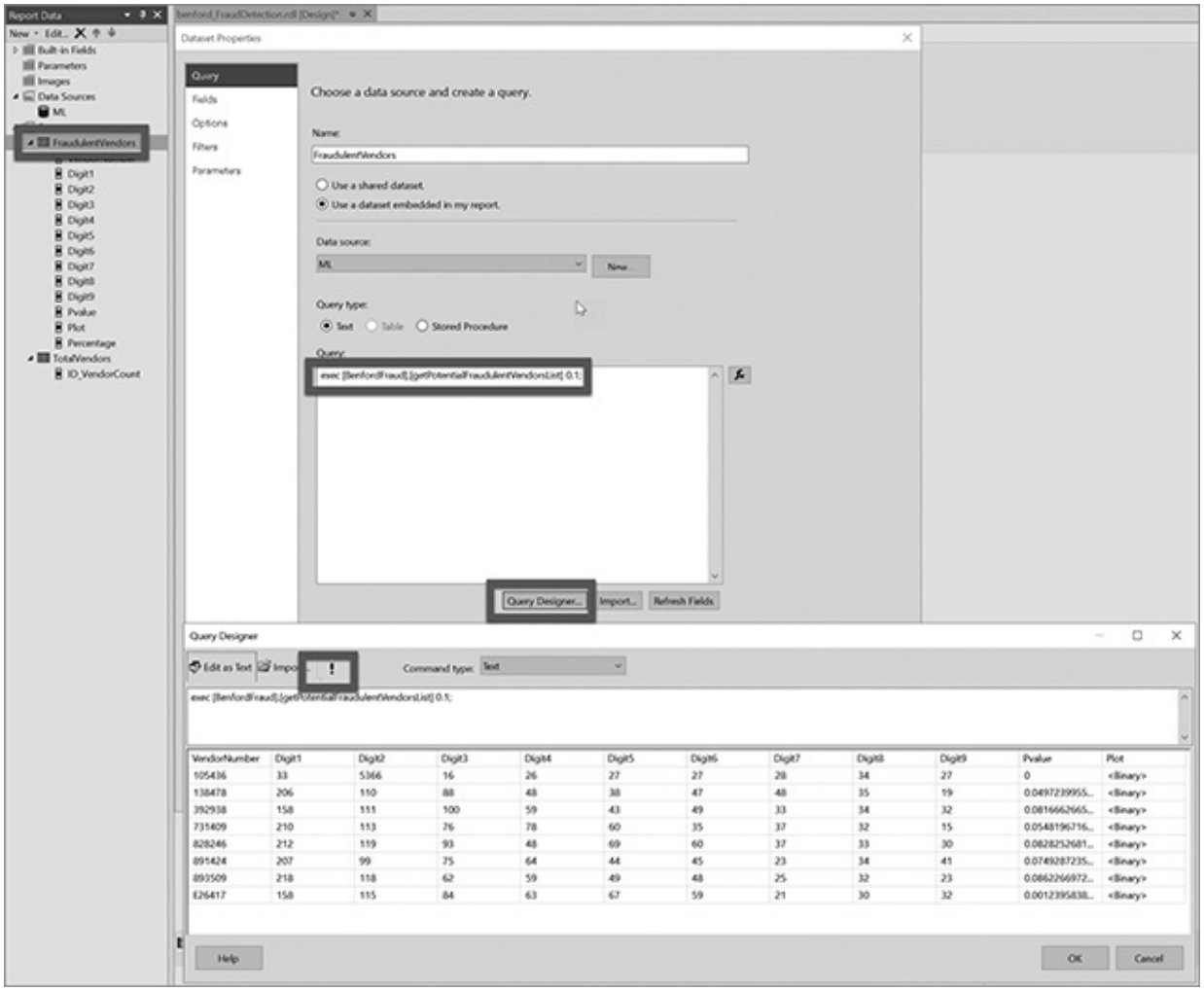
Po wywołaniu procedury, w tabeli `[BenfordFraud].[FraudulentVendorsPlots]` znajdziemy numery podejrzanych kontrahentów i wykresy porównujące częstość występujących w ich fakturach pierwszych cyfr kwot z rozkładem Benforda:

```
EXEC [BenfordFraud].getVendorInvoiceDigitsPlots 0.10;
GO
SELECT *
FROM [BenfordFraud].FraudulentVendorsPlots;
VendorNumber     Plot
105436  0x89504E470D0A1A0A0000000D4948445200000…
138478  0x89504E470D0A1A0A0000000D4948445200000…
…
```

Statyczne wykresy można udostępnić użytkownikom za pośrednictwem zintegrowanej z serwerem SQL Server usługi raportowania SSRS. My nie będziemy instalować serwera raportowania i ograniczymy się do uruchomienia przykładowego raportu w konsoli SSDT.

Najprostszym sposobem otworzenia w konsoli SSDT projektu przykładowego raportu jest dwukrotne kliknięcie myszą pliku *BenfordFraudReport\BenfordFraudReport.sln*. W lewej części ekranu zostanie wyświetlone okno *Report Data* zawierające źródło danych (bazę `ML`) oraz dwa użyte w raporcie zbiory danych. Zbiór *FraudulentVendors* zawiera wyniki wywołania procedury `[BenfordFraud].[getPotentialFraudulentVendorsList]`. Procedura ta odczytuje dane z tabel `[BenfordFraud].[FraudulentVendors]`

i `[BenfordFraud].[FraudulentVendorsPlots]`. Możemy się o tym przekonać, wyświetlając własności zbioru danych (z wyświetlanego prawym przyciskiem myszy menu kontekstowego należy wybrać opcję *Properties*) i uruchamiając tę procedurę w graficznym edytorze (rys. 2.2).



Rysunek 2.2. Dane źródłowe raportu zawierają m.in. numer kontrahenta i wygenerowany dla niego po stronie serwera SQL Server wykres

Drugi zbiór danych zawiera liczbę wszystkich kontrahentów odczytanych zapytaniem:

```
SELECT COUNT(DISTINCT VendorNumber) AS #VendorCount
FROM BenfordFraud.Invoices;
```

Najciekawszym polem raportu jest pole *Actual vs. Expected InvoiceAmounts*. Jeśli wyświetlimy jego własności, przekonamy się, że jest to pole typu *Image* i że pobiera ono dane z kolumny *Plot* zbioru *FraudulentVendors* (rys. 2.3).

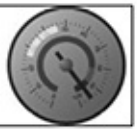


Rysunek 2.3. Zapisane w postaci binarnej wykresy są wyświetlane w kontrolce typu Image raportu SSRS

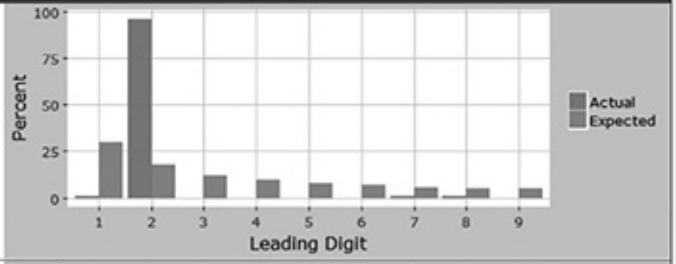
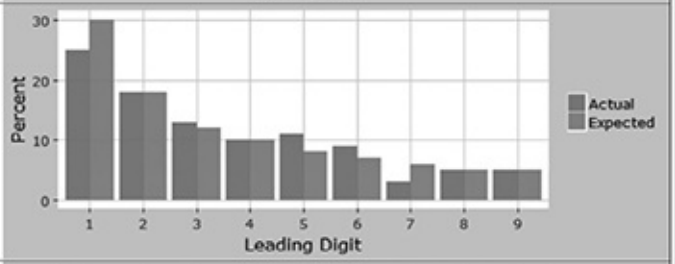
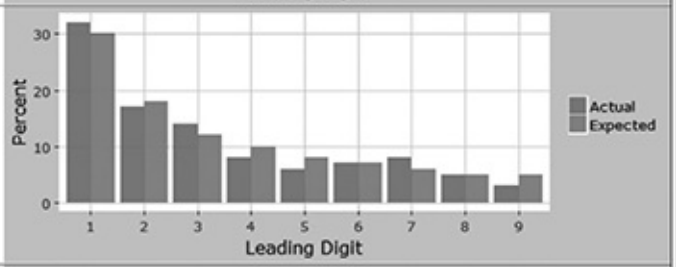
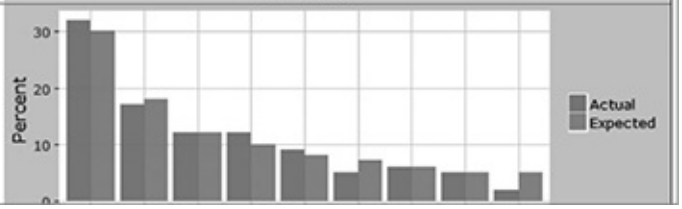
Okno edytora raportów może zostać przełączone w tryb ich podglądu. Po kliknięciu zakładki *Preview* raport zostanie wykonany i wyświetlony (rys. 2.4).

Vendors Flagged for Fraud Review

AVG CONFIDENCE
0.95



Number of Fraudulent Vendors
8 out of 195,416

Vendor #	Confidence %	Actual vs. Expected Invoice Amounts
105436	100%	
E26417	100%	
138478	95%	
731409	95%	

Rysunek 2.4. Raport porównujący częstość występowania pierwszych cyfr na fakturach podejrzanych kontrahentów z rozkładem Benforda

Podsumujmy. Wykonując to ćwiczenie, nauczyliśmy się:

- uruchamiać skrypty języka R po stronie serwera SQL Server,
- automatyzować uruchamianie zewnętrznych skryptów przy użyciu procedur składowanych,
- zapisywać wyniki wykonania zewnętrznych skryptów do tabel,
- konwertować wygenerowane przez zewnętrzne skrypty wykresy do postaci binarnej i zapisywać je w tabelach,
- osadzać przechowywane w tabelach wykresy na raportach SSRS.

Poznaliśmy też rozkład Benforda i nauczyliśmy się korzystać z prawa Benforda do wykrywania oszustw finansowych.

2.2. Klasyfikacja przejazdów

Kolega, taksówkarz z Nowego Jorku, poprosił nas o przysługę. Chciałby, żebyśmy mu z wyprzedzeniem powiedzieli, za który kurs dostanie napiwek. Jak przewidzieć, czy pasażer zapłaci napiwek? Ponieważ kolega dysponuje historycznymi danymi o kursach taksówek, możemy zbudować model klasyfikujący przejazdy jako z napiwkiem i bez.

Zadanie to potraktujemy jako okazję do lepszego poznania usługi SQL Server Machine Learning Services. Rozwiązanie przedstawione w tym podrozdziale jest zgodne z metodyką CRISP-DM i najlepszymi praktykami uczenia maszynowego, ale jego celem jest zdobycie praktycznej umiejętności pracy z usługą SQL Server Machine Learning Services i kwestie niezbędne do zbudowania wysokiej jakości modelu zostały tu pominięte.

Historyczne dane znajdują się w tabeli `[NYCTaxi].[Trips]`:

```
SELECT TOP 1 *
FROM [NYCTaxi].[Trips];
```

```
medallion        hack_license      vendor_idrate_codestore_and_fwd_flag pickup_datetime
dropoff_datetime passenger_count   trip_time_in_secs trip_distance     pickup_longitude
pickup_latitude  dropoff_longitude dropoff_latitude  payment_type      fare_amount
surcharge        mta_tax  tolls_amount      total_amount      tip_amount        tipped
tip_class
647C50AD2A7FEF04048010B84F6EB40E   0D5C7C263772B8280B083EE78FF08E1B     CMT      1        N
2013-07-03 06:59:57.000   2013-07-03 07:05:55.000    2        358      1        -73.978989
40.751766        -73.991806        40.750587CSH      6        0        0.5      0
6.5     0        0        0
```

Tabela ta liczy 1 703 957 wierszy. Naszą uwagę zwróciły trzy kolumny tej tabeli: `tipped`, `tip_amount` i `fare_amount`. Sprawdźmy rozkłady tych zmiennych oraz zależność między zmiennymi `tip_amount` a `fare_amount`. Tym razem do utworzenia odpowiednich wykresów użyjemy biblioteki `matplotlib` języka Python:

```
CREATE OR ALTER PROCEDURE [NYCTaxi].[SerializePlots]
AS
BEGIN
  SET NOCOUNT ON;
  DECLARE @query nvarchar(max) =
  N'SELECT cast(tipped as int) as tipped, tip_amount, fare_amount FROM [NYCTaxi].[Trips]'
  EXECUTE sp_execute_external_script
  @language = N'Python',
  @script = N'
import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt
import pickle

fig_handle = plt.figure()
plt.hist(InputDataSet.tipped)
plt.xlabel("Tipped")
plt.ylabel("Counts")
plt.title("Histogram, Tipped")
plot0 = pandas.DataFrame(data =[pickle.dumps(fig_handle)], columns=["plot"])
plt.clf()

plt.hist(InputDataSet.tip_amount)
plt.xlabel("Tip amount ($)")
plt.ylabel("Counts")
plt.title("Histogram, Tip amount")
plot1 = pandas.DataFrame(data =[pickle.dumps(fig_handle)], columns=["plot"])
plt.clf()

plt.hist(InputDataSet.fare_amount)
plt.xlabel("Fare amount ($)")
plt.ylabel("Counts")
plt.title("Histogram, Fare amount")
plot2 = pandas.DataFrame(data =[pickle.dumps(fig_handle)], columns=["plot"])
plt.clf()

plt.scatter( InputDataSet.fare_amount, InputDataSet.tip_amount)
plt.xlabel("Fare Amount ($)")
plt.ylabel("Tip Amount ($)")
plt.title("Tip amount by Fare amount")
```

```
plot3 = pandas.DataFrame(data =[pickle.dumps(fig_handle)], columns=["plot"])
plt.clf()

OutputDataSet = plot0.append(plot1, ignore_index=True).append(plot2, ignore_index=True).
append(plot3, ignore_index=True)
'
 ,@input_data_1 = @query
 WITH RESULT SETS ((plot varbinary(max)))
END
```

W pierwszej kolejności zdefiniowaliśmy zapytanie zwracające interesujące nas dane. Zmienna `@query` posłużyła nam do ustawienia wartości parametru `@input_data_1` procedury `sp_execute_external_script`. Następnie zaimportowaliśmy wymagane biblioteki, w tym użytą do serializacji wykresów bibliotekę `pickle`. Zatrzymajmy się na chwilę przy bibliotece `matplotlib`. Pracując ze skryptem na serwerze SQL, wszystkie wykresy muszą być wygenerowane do postaci binarnej. Zwróćmy uwagę, że po imporcie biblioteki `matplotlib` występuje deklaracja użycia tzw. backend przy użyciu metody `use('Agg')`. Taka deklaracja określa, że backend jest nieinteraktywny i wynikiem jest wykres jako obiekt zawierający plik PNG. Ważne jest również, aby deklaracja użycia odpowiedniego typu backend była wywołana przed importem biblioteki `matplotlib.pyplot`. Powtarzające się bloki instrukcji generują poszczególne wykresy i zapisują je w postaci tabeli z jedną, binarną kolumną. Na końcu binarne wersje wykresów zostają do siebie dodane, czyli nasz skrypt zwraca tabelę o jednej kolumnie i czterech wierszach:

```
EXEC [NYCTaxi].[SerializePlots];
plot
0x8003636D6174706C6F746C69622E6669677572650A4669677572650A710029…
0x8003636D6174706C6F746C69622E6669677572650A4669677572650A710029…
…
```

Tak utworzone wykresy odczytamy i zapiszemy do plików za pomocą skryptu języka Python:

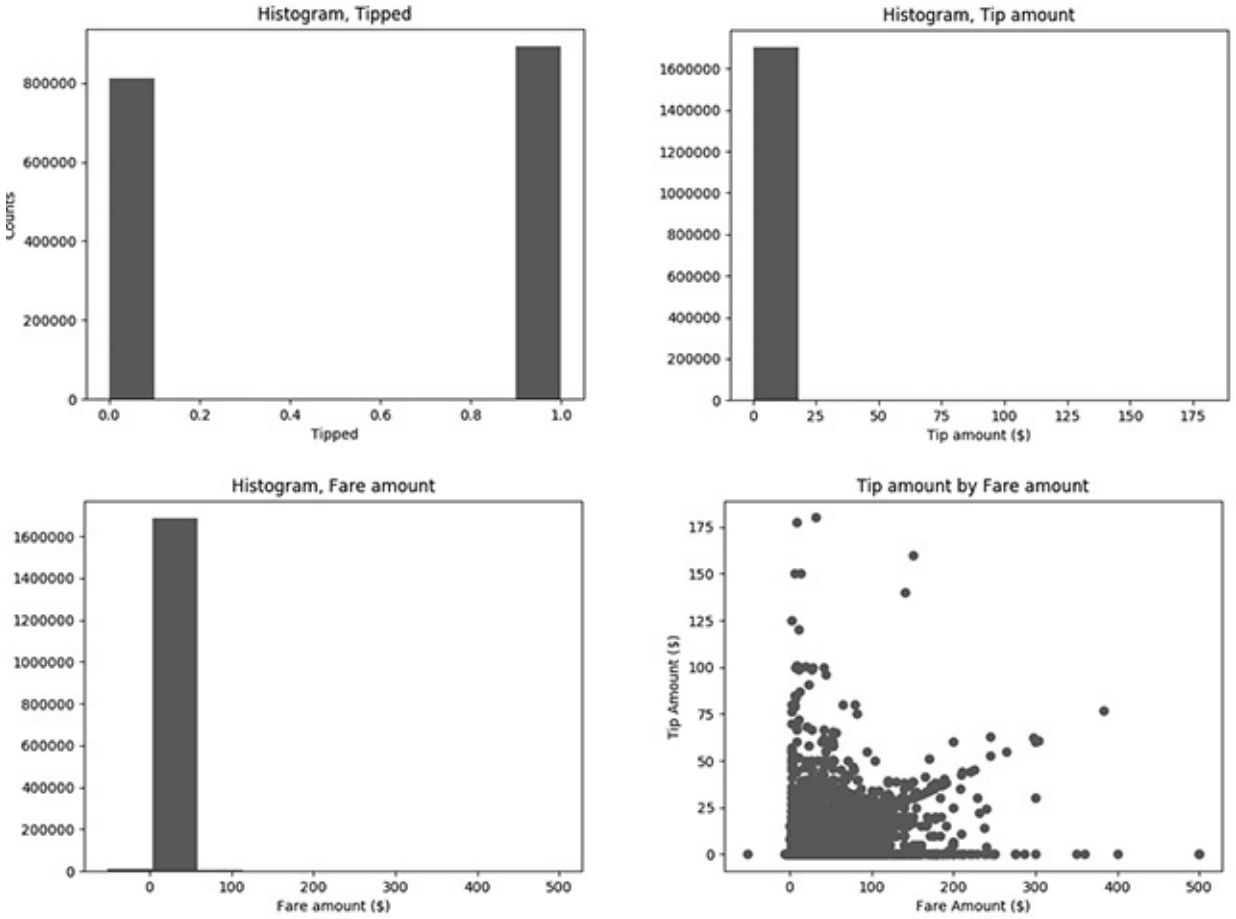
```
import pyodbc
import pickle
import os

cnxn = pyodbc.connect('DRIVER={SQL Server};SERVER={MS};DATABASE={ML};Trusted_Connection=
yes;')
cursor = cnxn.cursor()
cursor.execute("EXECUTE [NYCTaxi].[SerializePlots]")
tables = cursor.fetchall()
for i, table in enumerate(tables):
     fig = pickle.loads(table[0])
     fig.savefig(str(i)+'.png')
print("The plots are saved in directory:", os.getcwd())
"C:\Program Files\Microsoft SQL Server\MSSQL15.MSSQLSERVER\PYTHON_SERVICES\python.exe" "C:/
ML/Chapter02/Deserialize save plots.py"
The plots are saved in directory: C:\ML\Chapter02

Process finished with exit code 0
```

Do połączenia z serwerem SQL Server został użyty pakiet `pyodbc`. Wywołując jego funkcje `connect`, **należy podać m.in. nazwę docelowego serwera SQL – w kodzie została ona zaznaczona czcionką pogrubioną**. Następnie wywołaliśmy procedurę składowaną `[NYCTaxi].[SerializePlots]`, przeprowadziliśmy deserializację zwróconych przez nią wiersz i zapisaliśmy je w pliku *<numer wiersza>.png*. Na końcu wyświetliliśmy nazwę folderu, w którym zostały zapisane te pliki.

Utworzone w opisany sposób wykresy zostały pokazane na rysunku 2.5.



Rysunek 2.5. Okazuje się, że zmienna Tipped zawiera informację o tym, czy kierowca dostał napiwek (wartość 1), czy nie (wartość 0)

Z wykresów wynikałoby, że nieco ponad 50% kursów skończyła się napiwkiem (zaskakująco mało, jak na Stany Zjednoczone), wysokość największego napiwku wyniosła prawie 200 $ (dużo), najniższa opłata za kurs była ujemna (czyżby taksówkarz zapłacił pasażerowi?), a najwyższa opłata za kurs w obrębie Nowego Jorku przekroczyła 500 $ (drogo). Użycie takich mało wiarygodnych danych do treningu modelu spowoduje, że sam model również będzie mało wiarygodny. Ponadto brak silnej zależności liniowej między opłatą a napiwkiem sugeruje, że w modelu regresji liniowej zmienna `Fare Amount` będzie nieprzydatna.

Dane nie tylko należy wyczyścić, lecz także wzbogacić. W tym przypadku wzbogacanie danych sprowadziło się do wyliczenia odległości euklidesowej

między miejscami rozpoczęcia a zakończenia kursu. Długość tę zwraca funkcja `[NYCTaxi].[fnCalculateDistance]`.

Etap przygotowania danych obejmuje również usunięcie nieprzydatnych zmiennych. Na potrzeby przykładu arbitralnie ograniczyłem zmienne wejściowe do czterech: `passenger_count`, `trip_distance`, `trip_time_in_secs` i wyniku funkcji `[NYCTaxi].[fnCalculateDistance]`. Te dane zwraca funkcja tabelaryczna `[NYCTaxi].[fnEngineerFeatures]`.

Posiadane dane należy podzielić na treningowe i testowe – dane treningowe zostaną użyte do uczenia modeli, a dane testowe do ich oceny. Użyjemy do tego procedury `[NYCTaxi].[TrainTestSplit]`:

```
CREATE OR ALTER PROCEDURE [NYCTaxi].[TrainTestSplit] (@pct int)
AS
DROP TABLE IF EXISTS [NYCTaxi].[Training];
SELECT * INTO NYCTaxi.[Training]
FROM [NYCTaxi].[Trips]
WHERE (ABS(CAST(BINARY_CHECKSUM(medallion,hack_license) AS int)) % 100) <= @pct;
CREATE CLUSTERED COLUMNSTORE INDEX CCI_training on [NYCTaxi].[Training];

DROP TABLE IF EXISTS [NYCTaxi].[Testing];
SELECT * INTO [NYCTaxi].[Testing]
FROM [NYCTaxi].[Trips]
WHERE (ABS(CAST(BINARY_CHECKSUM(medallion,hack_license) AS int)) % 100) > @pct;
CREATE CLUSTERED COLUMNSTORE INDEX CCI_testing on [NYCTaxi].[Testing];
```

Procedura wykonuje następujące operacje:

- jeżeli tabela docelowa istnieje, to ją usuwa;
- następnie zakłada tabelę docelową i wstawia do niej określony procent losowo wybranych wierszy tabeli `[NYCTaxi].[Trips]`;
- konwertuje docelową tabelę do indeksu kolumnowego.

Operacje te powtórzymy dla drugiej tabeli, z tym że do drugiej tabeli zostaną

zapisane tylko wiersze, które nie trafiły do pierwszej. Poniższe wywołanie procedury zapisze 70% wierszy do tabeli z danymi treningowymi, a pozostałe 30% do tabeli z danymi testowymi:

```
EXEC [NYCTaxi].[TrainTestSplit] @pct=70
```

```
(1210277 rows affected)
(493680 rows affected)
```

Tak przygotowane dane posłużą nam do zbudowania modeli regresji logistycznej. Pierwszy model utworzymy, korzystając ze standardowej, popularnej biblioteki uczenia maszynowego `scikit-learn`:

```
CREATE OR ALTER PROCEDURE [NYCTaxi].[TrainTipPredictionModelSciKitPy] (@trained_model
varbinary(max) OUTPUT)
AS
BEGIN
  EXEC sp_execute_external_script
    @language = N'Python',
    @script = N'
import numpy
import pickle
from sklearn.linear_model import LogisticRegression

## Create model
X = InputDataSet[["passenger_count", "trip_distance", "trip_time_in_secs",
"direct_distance"]]
y = numpy.ravel(InputDataSet[["tipped"]])

SKLalgo = LogisticRegression()
logitObj = SKLalgo.fit(X, y)

## Serialize model
trained_model = pickle.dumps(logitObj)',
    @input_data_1 = N' select tipped, fare_amount, passenger_count, trip_time_in_secs,
trip_distance, [NYCTaxi].fnCalculateDistance(pickup_latitude, pickup_longitude,
dropoff_latitude, dropoff_longitude) as direct_distance
    from [NYCTaxi].[Training]',
      @input_data_1_name = N'InputDataSet',
      @params = N'@trained_model varbinary(max) OUTPUT',
      @trained_model = @trained_model OUTPUT;
END;
```

Działanie procedury `[NYCTaxi].[TrainTipPredictionModelSciKitPy]` sprowadza się do wywołania procedury systemowej `sp_execute_external_script`. Użyty w niej skrypt języka Python:

- importuje wymagane biblioteki,
- tworzy model regresji logistycznej i uczy go na danych treningowych,
- serializuje i zwraca nauczony model.

Warto zwrócić uwagę na sposób przekazywania parametrów między skryptem

zewnętrznym a serwerem SQL Server. Aby zwrócić nauczony model, należało użyć parametru `@params` do zdefiniowania parametrów zewnętrznego skryptu, a następnie zdefiniować parametr wyjściowy procedury `sp_execute_external_script`. Odpowiednie fragmenty kodu zostały wyróżnione pogrubieniem.

Skoro procedura `[NYCTaxi].[TrainTipPredictionModelSciKitPy]` zwraca nauczony model w postaci danych binarnych, to wynik jej wywołania należy zapisać do zmiennej typu `varbinary(max)`. Wartość tej zmiennej – nauczony model regresji logistycznej – zapiszemy w tabeli `[NYCTaxi].[Models]`:

```
DECLARE @model varbinary(max);
EXEC [NYCTaxi].[TrainTipPredictionModelSciKitPy] @model OUTPUT;
INSERT INTO [NYCTaxi].[Models] (name, model)
VALUES('Logistic Regression SciKi', @model);
```

```
(1 row affected)
```

Do zbudowania drugiego modelu regresji logistycznej użyjemy biblioteki `revoscalepy` instalowanej razem z usługą SQL Server Machine Learning Services. Działanie procedury `[NYCTaxi].[TrainTipPredictionModelRxPy]` jest takie samo jak poprzedniej. Sam skrypt języka Python wygląda jednak nieco inaczej:

```
from revoscalepy.functions.RxLogit import rx_logit
from revoscalepy import rx_serialize_model

logitObj = rx_logit("tipped ~ passenger_count + trip_distance +
trip_time_in_secs + direct_distance", data=InputDataSet);

## Serialize model
trained_model = rx_serialize_model(logitObj)
```

Zwróćmy uwagę że funkcje pakietów `revoscalepy i microsoftml` mają nazwy zaczynające się od prefiksu `rx`. Sposób definiowana modeli uczenia

maszynowego przy użyciu tych funkcji też jest nieco inny – zamiast wskazywać zmienne wejściowe i wyjściową, definiuje się formułę uczącą. W tej formule zmienną wyjściową umieszcza się po lewej stronie znaku ~, a po jego prawej stronie umieszcza się zmienne wejściowe. Tym razem do serializacji modelu została użyta funkcja `rx_serialize_model`.

Po uruchomieniu procedury zobaczymy dodatkowe komunikaty diagnostyczne informujące m.in. o liczbie wierszy przetwarzanych w poszczególnych turach i czasie trenowania modelu. W tym przypadku zostało również wyświetlone ostrzeżenie mówiące o tym, że podczas treningu kilka obserwacji zostało sklasyfikowanych ze 100% pewnością:

```
DECLARE @model varbinary(max);
EXEC [NYCTaxi].[TrainTipPredictionModelRxPy] @model OUTPUT;
INSERT INTO [NYCTaxi].[Models] (name, model) VALUES('Logistic Regression Revoscalepy',
@model);
```

```
STDOUT message(s) from external script:
Rows Read: 1210277, Total Rows Processed: 1210277, Total Chunk Time: 0.034 seconds

Starting values (iteration 1) time: 0.046 secs.
Rows Read: 1210277, Total Rows Processed: 1210277, Total Chunk Time: 0.113 seconds
…

Elapsed computation time: 1.128 secs.

 Warning: One or more fitted probabilities were numerically 0 or 1 during estimation.
(similar message repeated 2 times)
```

Modeli zapisanych w tabeli można łatwo używać do predykcji. Tym razem najpierw zobaczymy efekt wywołania procedury wysyłającej zapytanie predykcyjne do modelu, a następnie omówię jej działanie.

Zapytanie predykcyjne może dotyczyć jednej obserwacji albo wielu. W pierwszym przypadku wartości zmiennych są ustawiane bezpośrednio w procedurze. Aby obliczyć na przykład prawdopodobieństwo otrzymania

napiwku od pojedynczego pasażera na trasie długości 2,5 mili, jeżeli przejazd zajął 10 i pół minuty, a odległość między miejscem odjazdu a przyjazdu wynosiła 1,3 mili, należy wykonać poniższą instrukcję:

```
EXEC [NYCTaxi].[PredictTipSingleModeSciKitPy]
        @model ='Logistic Regression SciKi',
        @passenger_count = 1,
        @trip_distance = 2.5,
        @trip_time_in_secs = 630,
        @pickup_latitude = 40.763958,
        @pickup_longitude = -73.973373,
        @dropoff_latitude = 40.782139,
        @dropoff_longitude = -73.977303;
```

```
0.526013388715943
```

Procedura `[NYCTaxi].[PredictTipSingleModeSciKitPy]` jest warta przestudiowania z dwóch powodów. Po pierwsze, pokazuje ona, w jaki sposób można odczytać model zapisany jako ciąg bajtów, odtworzyć go i wykonać na modelu zapytanie predykcyjne. Po drugie, uczy przekazywania parametrów między procedurą użytkownika, procedurą systemową `sp_execute_external_script` a zewnętrznym skryptem. Definicja tej procedury wygląda następująco:

```
CREATE OR ALTER PROCEDURE [NYCTaxi].[PredictTipSingleModeSciKitPy] (@model varchar(50),
        @passenger_count int = 0,
        @trip_distance float = 0,
        @trip_time_in_secs int = 0,
        @pickup_latitude float = 0,
        @pickup_longitude float = 0,
        @dropoff_latitude float = 0,
        @dropoff_longitude float = 0)
AS
BEGIN
        DECLARE @inquery nvarchar(max) = N'
        SELECT * FROM [NYCTaxi].[fnEngineerFeatures](
        @passenger_count,
        @trip_distance,
        @trip_time_in_secs,
        @pickup_latitude,
        @pickup_longitude,
        @dropoff_latitude,
        @dropoff_longitude)'

        DECLARE @lmodel2 varbinary(max) = (SELECT model FROM [NYCTaxi].[Models] WHERE NAME
= @model);
        EXEC sp_execute_external_script
                 @language = N'Python',
                 @script = N'
import pickle
import numpy

# Load model and unserialize
mod = pickle.loads(model)

# Get features for scoring from input data
X = InputDataSet[["passenger_count", "trip_distance", "trip_time_in_secs",
"direct_distance"]]

# Score data to get tip prediction probability as a list (of float)
prob = [mod.predict_proba(X)[0][1]]

# Create output data frame
OutputDataSet = pandas.DataFrame(data=prob, columns=["predictions"])',
        @input_data_1 = @inquery,
```

```
        @params = N'@model varbinary(max),
                  @passenger_count int ,
                  @trip_distance float ,
                  @trip_time_in_secs int ,
                  @pickup_latitude float ,
                  @pickup_longitude float ,
                  @dropoff_latitude float ,
                  @dropoff_longitude float',
        @model = @lmodel2,
        @passenger_count =@passenger_count ,
        @trip_distance=@trip_distance,
        @trip_time_in_secs=@trip_time_in_secs,
        @pickup_latitude=@pickup_latitude,
        @pickup_longitude=@pickup_longitude,
        @dropoff_latitude=@dropoff_latitude,
        @dropoff_longitude=@dropoff_longitude
WITH RESULT SETS ((Score float));
      END;
```

Procedura jest wywoływana z ośmioma parametrami. Pierwszym jest nazwa modelu, która zostanie przekazana do zewnętrznego skryptu. Pozostałe parametry zostały użyte w zapytaniu, którego wynik będzie przekazany do zewnętrznego skryptu. Na przykład procedura wywołana z tymi samymi parametrami co poprzednio przekaże do zewnętrznego skryptu poniższe dane:

```
passenger_count  trip_distance     trip_time_in_secs direct_distance
1                        2.5              630      1.2729705407
```

W procedurze definiowana jest lokalna zmienna `@lmodel2`, a jej wartość jest ustalana na wynik zapytania `SELECT model FROM [NYCTaxi].[Models] WHERE name = @model`, czyli zawiera binarną wersję modelu o podanej nazwie.

Ta zmienna oraz parametry wywołania naszej procedury zostają przekazane do zewnętrznego skryptu przez odpowiednie parametry procedury systemowej `sp_execute_external_script`:

- zapytanie zwracające dane o kursie zostaje zapisane do parametru

`@input_data_1`,

- nazwy oraz typy przekazywanych parametrów zostały zdefiniowane w parametrze `@params`,
- ich wartości zostały ustawione w parametrach wymienionych po parametrze `@params`.

Działanie zewnętrznego skryptu sprowadza się do deserializacji przekazanego modelu, wywołania jego funkcji `predict_proba` i zapisania wyniku w postaci tabelarycznej.

Ten sam model możemy zapytać o klasyfikację wielu przejazdów, na przykład o wszystkie przejazdy zapisane w tabeli z danymi testowymi:

```
DECLARE @query_string nvarchar(max)
SET @query_string='
select tipped, fare_amount, passenger_count, trip_time_in_secs, trip_distance,
       NYCTaxi.fnCalculateDistance(pickup_latitude, pickup_longitude,  dropoff_latitude,
dropoff_longitude) as direct_distance
from [NYCTaxi].[Testing]'
EXEC [NYCTaxi].[PredictTipSciKitPy] 'Logistic Regression SciKi', @query_string;
ActualTipInfo    fare_amount       passenger_count   trip_time_in_secs trip_distance
direct_distance  ProbabilityOfTip
1       63.5     5        1320     16.97    10.4254386135     0.643383934140104
1       145      1        3360     33.34    33.9735505938     0.805074077551691
1       65       1        1620     17.56    10.9838154823     0.678189831991684
…
STDOUT message(s) from external script:
AUC on testing data is: 0.565219613285
(493680 rows affected)
```

Zachęcam Czytelników do samodzielnego zbudowania modelu klasyfikującego przejazdy jako nagrodzone napiwkiem lub nie. Po lekturze pozostałych rozdziałów książki powinni Państwo zbudować model, dla którego miara AUC przekroczy 85%.

Procedura `[NYCTaxi].[PredictTipSciKitPy]` różni się od wersji służącej do klasyfikacji pojedynczych przejazdów. Po pierwsze, zwraca ona dane wejściowe

uzupełnione o wynik zapytania predykcyjnego (prawdopodobieństwo napiwku), a po drugie, dodatkowo wylicza i zwraca jedną z popularnych metryk modeli klasyfikujących. Użyty w tej procedurze zewnętrzny skrypt został pokazany poniżej:

```
import pickle
import numpy
from sklearn import metrics

mod = pickle.loads(lmodel2)

X = InputDataSet[["passenger_count", "trip_distance", "trip_time_in_secs",
"direct_distance"]]
y = numpy.ravel(InputDataSet[["tipped"]])

prob_array = mod.predict_proba(X)
prob_list = [item[1] for item in prob_array]

prob_array = numpy.asarray(prob_list)
fpr, tpr, thresholds = metrics.roc_curve(y, prob_array)
auc_result = metrics.auc(fpr, tpr)
print("AUC on testing data is:", auc_result)

df1 = pandas.DataFrame(data=InputDataSet)
df2 = pandas.DataFrame(data=prob_array)
OutputDataSet = pandas.concat([df1,df2],axis=1,join="inner")
```

Podsumujmy. Wykonując to ćwiczenie, nauczyliśmy się:

- uruchamiać skrypty języka Python po stronie serwera SQL Server;
- łączyć się z serwerem SQL Server za pomocą pakietu `pyodbc`;
- wykonywać procedury składowane z poziomu skryptów Python;
- losowo dzielić dane na treningowe i testowe;
- budować, trenować i zapisywać modele uczenia maszynowego przy użyciu standardowych pakietów Python i pakietu `revoscalepy`;
- przekazywać parametry między procedurami użytkownika, procedurą

systemową `sp_execute_external_script` a zewnętrznymi skryptami;

- odczytywać modele i wykonywać na nich zapytania predykcyjne dla jednej i wielu obserwacji.

2.3. Dodatkowe funkcjonalności serwera SQL Server i usługi SQL Machine Learning Services

Poznane funkcjonalności serwera SQL Server i usługi SQL Server Machine Learning Services pozwolą Państwu definiować, trenować, optymalizować i wdrażać modele uczenia maszynowego. Zanim jednak przejdziemy do omówienia technik uczenia maszynowego, chcę pokazać Państwu trzy dodatkowe, przydatne funkcjonalności: zapytania predykcyjne czasu rzeczywistego, natywne zapytania predykcyjne i automatyczne tworzenie modeli dla partycji danych.

2.3.1. Zapytania predykcyjne czasu rzeczywistego

Wykonanie zapytania predykcyjnego nie musi wymagać wywołania procedury systemowej `sp_execute_external_script`. W pewnych sytuacjach możemy wykonać zapytanie predykcyjne za pośrednictwem procedury `sp_rxPredict`.

Procedura ta jest dostępna we wszystkich wersjach SQL Server Machine Learning Services i działa tak samo dla modeli stworzonych w języku R i dla modeli stworzonych w języku Python. Jej użycie wymaga:

- włączenia integracji serwera SQL Server z platformą CLR:

```
EXEC sp_configure 'clr enabled', 1
GO
RECONFIGURE
```

```
Configuration option 'clr enabled' changed from 0 to 1. Run the RECONFIGURE statement to
install.
```

- włączenia w bazie danych funkcji zapytań predykcyjnych czasu rzeczywistego; operację tę należy wykonać w wierszu polecenia:

```
C:\Program Files\Microsoft SQL Server\MSSQL15.MSSQLSERVER\R_SERVICES\library\RevoScaleR\
rxLibs\x64>RegisterRExt.exe /installRts /database:ML
```

jeżeli dla bazy danych była włączona wcześniejsza wersja tej funkcji, należy ją najpierw odinstalować:

```
RegisterRExt.exe /uninstallrts /database:ML
```

- użycia do treningu modelu jednego z algorytmów pakietów `revoscalepy`, `microsoftml` (jeżeli model został stworzony w języku Python) lub `RevoScaleR`, `MicrosoftML` (jeżeli model został stworzony w języku R);
- serializacji modelu za pomocą funkcji `rx_serialize_model` (jeżeli model został stworzony w języku Python) lub `rxSerialize` (jeżeli model został stworzony w języku R).

Procedura `sp_rxPredict` została napisana w kodzie CLR, a do deserializacji modeli i wysłania do nich zapytań predykcyjnych używa bibliotek C++. Użycie kompilowanych języków pozwoliło kilkukrotnie skrócić czas wykonywania zapytań predykcyjnych.

Procedura `sp_rxPredict` ma też swoje ograniczenia. Po pierwsze, zwraca ona jedynie otrzymany z modelu wynik zapytania – w naszym przypadku prawdopodobieństwo napiwku. **Brak możliwości połączenia wyniku procedury z danymi wejściowymi ogranicza jej użycie do zapytań**

o pojedyncze obserwacje. Po drugie, wymaga ona przekazania danych w takim samym formacie jak format danych treningowych modelu, z dokładnością do kolejności i typów zmiennych.

Poniżej został pokazany sposób użycia procedury `sp_rxPredict` do oceny prawdopodobieństwa otrzymania napiwków przez wybranego taksówkarza:

```
DECLARE @model varbinary(max) = (SELECT model FROM NYCTaxi.Models
  WHERE name = 'Logistic Regression Revoscalepy');
EXEC sp_rxPredict
       @model = @model,
       @inputData = N'select tipped, fare_amount, passenger_count, trip_time_in_secs,
trip_distance, NYCTaxi.fnCalculateDistance(pickup_latitude, pickup_longitude,
dropoff_latitude, dropoff_longitude) as direct_distance
       from [NYCTaxi].[Testing] where medallion =
''02536669BC1D149D7F2F867CC403E60C''';
```

```
tipped_Pred
0.509538176900241
0.511652184262108
0.51588910382184
0.508473071352673
…
```

2.3.2. Natywne zapytania predykcyjne

Funkcja `PREDICT` została dodana w wersji 2017 serwera SQL Server. Jest to funkcja tabelaryczna języka T-SQL, zaimplementowana w kodzie C++. Taka implementacja oznacza brak zależności między funkcją `PREDICT` a usługą SQL Server Machine Learning Services, czyli **możemy korzystać z tej funkcji na serwerach SQL Server 2017 i nowszych z niewłączoną integracją z językami R i Python**. Ponadto jest to funkcja tabelaryczna, więc możemy jej używać w zapytaniach i instrukcjach `INSERT`/`UPDATE`/`MERGE`, tak jak pozostałych funkcji tabelarycznych języka T-SQL; możemy na przykład połączyć jej wyniki z odczytanymi z tabeli danymi.

Niestety, funkcja ta wspiera tylko wybrane modele bibliotek `revoscalepy` (`rx_lin_mod`, `rx_logit`, `rx_btrees`, `rx_dtree`, `rx_dforest`) i `RevoScaleR` (`rxLinMod`, `rxLogit`, `rxBTrees`, `rxDtree`, `rxDForest`). Ponadto wymaga ona serializacji modelu przy użyciu funkcji `rx_serialize_model` (`rxSerialize` dla języka R) wywołanej z ustawionym parametrem `realtime_scoring_only` (`realtimeScoringOnly` dla języka R).

Procedura `[NYCTaxi].[TrainTipPredictionModelRxPyNative]` tworzy spełniający te wymagania model. Wykonywany w tej procedurze zewnętrzny skrypt wygląda następująco:

```
from revoscalepy.functions.RxLogit import rx_logit
from revoscalepy import rx_serialize_model
logitObj = rx_logit("tipped ~ passenger_count + trip_distance + trip_time_in_secs +
direct_distance", data=InputDataSet);
## Serialize model
trained_model = rx_serialize_model(logitObj, realtime_scoring_only = True)
```

Utworzonego za jej pomocą modelu możemy użyć do wykonania natywnego zapytania predykcyjnego:

```
DECLARE @model varbinary(max);
EXEC [NYCTaxi].[TrainTipPredictionModelRxPyNative] @model OUTPUT;
INSERT INTO [NYCTaxi].[Models] (name, model)
VALUES ('Native Logistic Regression Revoscalepy', @model);
GO
DECLARE @model varbinary(max)
SELECT @model = [model] from [NYCTaxi].[Models] WHERE name = 'Native Logistic Regression
Revoscalepy';
SELECT  d.*, p.*
FROM PREDICT(MODEL = @model, DATA = [NYCTaxi].[vTesting]  as d)
WITH (tipped_pred float) as p;
```

```
tipped  fare_amount       passenger_count   trip_time_in_secs
trip_distance    direct_distance   tipped_pred
1       5        1        213      0.9      0.6879574056      0.509539881577233
1       5        1        226      0.9      277.819377698     0.514604243045694
…
```

Na testowym serwerze odczytanie prawie 500 000 wierszy, policzenie odległości między punktami wyjazdu a przyjazdu oraz wykonanie dla tych danych zapytania predykcyjnego zajęło 10 sekund. Użycie funkcji `PREDICT` jest najszybszym sposobem wykonywania zapytań predykcyjnych przez SQL Server.

2.3.3. Tworzenie modeli dla partycji danych

W wersji 2019 serwera SQL Server procedura `sp_execute_external_script` zyskała dwa dodatkowe parametry: `@input_data_1_partition_by_columns` i `@input_data_1_order_by_columns`. Pierwszy z nich pozwala tworzyć modele uczenia maszynowego dla partycji danych. Przekazana przez ten parametr zmienna partycjonująca musi mieć format tabelaryczny. Drugi parametr pozwala sortować dane przekazywane do zewnętrznego skryptu.

Działanie parametru `@input_data_1_partition_by_columns` polega na wykonaniu zewnętrznego skryptu języka R/Python dla każdej wartości parametru. Dla poszczególnych wywołań skryptu zbiór danych wejściowych jest ograniczany do

wierszy z aktualnej partycji. Jeżeli do podziału danych na partycje użylibyśmy na przykład kolumny `Country`, skrypt zostałby wykonany tyle razy, ile nazw różnych krajów znajdowałoby się w tej kolumnie, przy czym podczas pierwszego wywołania do skryptu byłyby przekazane wiersze spełniające warunek `Country = ,Albania'`, przy drugim wiersze spełniające warunek `Country = ,Belgia'` itd.

Takie działanie oznacza, że parametr `@input_data_1_partition_by_columns` pozwala automatycznie tworzyć wiele kopii modelu uczenia maszynowego i trenować je na różnych partycjach danych. Przypuśćmy, że analizując dane o przejazdach taksówek, zauważyliśmy związek między sposobem płatności (zmienną `payment_type`) a napiwkiem. Postanowiliśmy więc stworzyć osobne modele dla płatności gotówką, kartą kredytową itd.

Tym razem do budowy modelu użyjemy języka R. W ten sposób sprawdzimy, czy funkcja `PREDICT` pozwala wysyłać zapytania predykcyjne do modeli stworzonych w obu językach. Ponieważ zewnętrzny skrypt będzie wykonywany wielokrotnie, a nasza procedura tylko raz, zamiast zwracać binarną wersję modeli, zapiszemy je do tabeli z poziomu skryptu języka R. Wymaga to kilku modyfikacji:

- procedura będzie wywoływana bez parametrów i nie będzie zwracała żadnych danych;
- ponieważ zewnętrzny skrypt będzie wykonywany wielokrotnie, za każdym razem dla innych danych, ich wykonanie można zrównoleglić; asynchroniczne wykonanie procedury można wymusić, ustawiając wartości parametru `@parallel` na `1`;
- asynchroniczne wywołanie procedury `sp_execute_external_script` wymaga jawnego określenia formatu zwracanych danych w klauzuli `WITH RESULT SETS`; jeżeli procedura nie zwraca danych, tak jak w tym przypadku,

w klauzuli należy wpisać `NONE`;

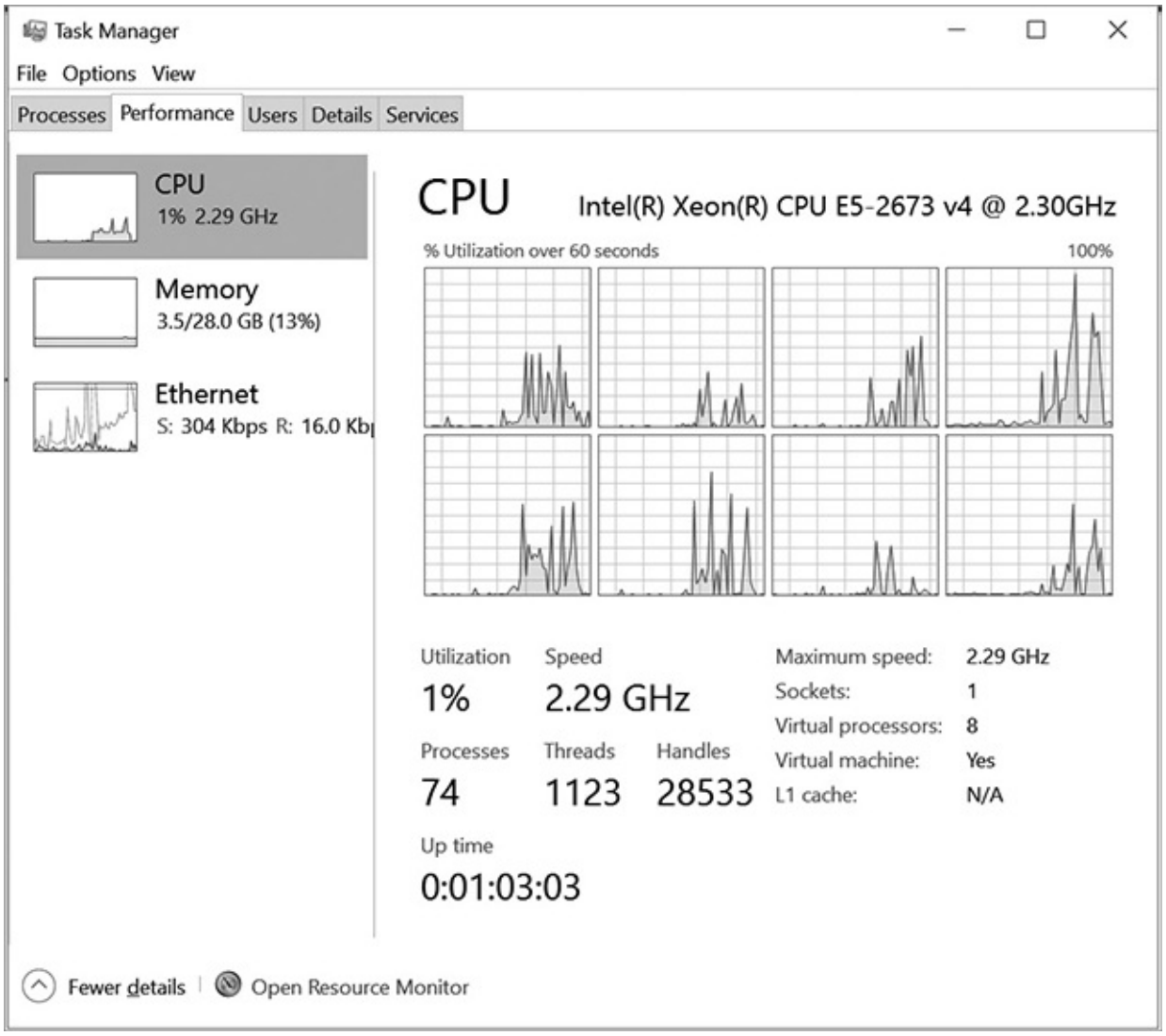
- trening i serializacja modelu zostaną przeprowadzone w taki sam sposób (warto zwrócić uwagę na podobieństwa między pakietami `revoscalepy` a `RevoScaleR`);
- zapis do tabeli wymaga nawiązania połączenia z serwerem SQL Server; do nawiązania połączenia użyjemy funkcji `RxOdbcData` pakietu `RevoScaleR`;
- nazwa modelu zostanie uzupełniona o typ zapłaty, dla którego został on utworzony;
- binarna wersja modelu i jego nazwa zostaną zapisane do tabeli przy użyciu funkcji `rxWriteObject` pakietu `RevoScaleR`.

Procedura realizująca te zadania wygląda tak:

```
CREATE OR ALTER PROCEDURE [NYCTaxi].[TrainTipPredictionModelsRxPyNativePerPartition]
AS
BEGIN
        DECLARE @model VARBINARY(max)
        DECLARE @instance_name NVARCHAR(100) = @@SERVERNAME
        EXEC sp_execute_external_script
     @language = N'R',
     @script = N'
        logitObj <- rxLogit(tipped ~ passenger_count + trip_distance + trip_time_in_secs +
direct_distance, data = InputDataSet)
        model <- rxSerializeModel(logitObj, realtimeScoringOnly = TRUE)

        connStr <- paste("Driver=SQL Server;Server=", instance_name, ";Database=ML;
Trusted_Connection=true;", sep="")
        ds <- RxOdbcData(table="[NYCTaxi].[Models]", connectionString=connStr)
        model_name <- paste0("Native Logistic Regression Revoscalepy.",
InputDataSet[1,]$payment_type)
        rxWriteObject(ds, model_name, model,keyName = "name", valueName = "model",
overwrite = TRUE, serialize = FALSE)',
     @input_data_1 = N'
                 select payment_type, CAST(tipped as BIT) AS tipped, fare_amount,
passenger_count, trip_time_in_secs, trip_distance,
                 [NYCTaxi].fnCalculateDistance(pickup_latitude, pickup_longitude,
dropoff_latitude, dropoff_longitude) as direct_distance
                 from [NYCTaxi].[Training]',
        @input_data_1_name = N'InputDataSet',
        @input_data_1_partition_by_columns = N'payment_type',
        @parallel = 1,
        @params = N'@instance_name nvarchar(100)',
        @instance_name = @instance_name
        WITH RESULT SETS NONE;
END;
```

Wywołując procedurę, przekonamy się, że wytrenowanych zostało pięć modeli, i że do ich treningu zostały użyte wszystkie procesory serwera (rys. 2.6).



Rysunek 2.6. W wersji 2019 ustawiając parametr @parallel na wartość 1, wymusimy zrównoleglenie wykonania wszystkich zewnętrznych skryptów, również jednowątkowych funkcji języka R

```
EXEC [NYCTaxi].[TrainTipPredictionModelsRxPyNativePerPartition];
SELECT [name], left(model,20)
FROM [NYCTaxi].[Models];
```

```
name                                          (No column name)
Logistic Regression SciKi
€──────────────────────────────────────────────csklearn.linear_mo
Logistic Regression Revoscalepy               blob!U¨p†IÍ…Š˝ÚĽÜI
Native Logistic Regression Revoscalepy.NOC    blobąĐšĽÖ&~ -
Native Logistic Regression Revoscalepy        blobUBłÂT¦*Ç
Native Logistic Regression Revoscalepy.CRD    blobßôŮ› ¸-gŽB ov…
Native Logistic Regression Revoscalepy.CSH    blobî·zFÇPú8şÜŔŞŤ
Native Logistic Regression Revoscalepy.DIS    blobňňdĎ‹ěFç˙Ä8sŽOĄ
Native Logistic Regression Revoscalepy.UNK    blobÎźU|ŃWąpNm±¨›
```

Skoro każdy model został wytrenowany na podzbiorze danych, powinien odpowiadać na pytanie o prawdopodobieństwo napiwku tylko w odniesieniu do przejazdów opłaconych w określony sposób. Parametr `DATA` funkcji `PREDICT` może wskazywać na tabelę, widok lub nazwane podzapytanie (CTE). Poniższy przykład pokazuje sposób wywołania funkcji `PREDICT` dla danych wybranych w nazwanym podzapytaniu:

```
DECLARE @model varbinary(max)
SELECT @model = [model]
FROM [NYCTaxi].[Models]
WHERE [name] = 'Native Logistic Regression Revoscalepy.CRD';

WITH ds AS (SELECT tipped, fare_amount, passenger_count, trip_time_in_secs, trip_distance,
                                NYCTaxi.fnCalculateDistance(pickup_latitude,
pickup_longitude,  dropoff_latitude, dropoff_longitude) AS direct_distance
                        FROM [NYCTaxi].[Testing] WHERE [payment_type] = 'CRD')

SELECT  ds.*, p.*
FROM PREDICT(MODEL = @model, DATA = ds)
```

```
WITH (tipped_pred float) as p;
tipped  fare_amount       passenger_count   trip_time_in_secs  trip_distance
direct_distance  tipped_pred
0       8.5       1       511       2       1.8902772886       0.970988195892648
0       14.5      1       1364      2.5     0.0559851794       0.971008926473009
0       30        1       1473      9.8     6.9945670951       0.970917651628559
…
```

Podsumujmy. W tym podrozdziale nauczyliśmy się:

- używać procedury `sp_rxPredict` do wykonywania zapytań predykcyjnych czasu rzeczywistego;
- używać funkcji `PREDICT` do wykonywania natywnych zapytań predykcyjnych;
- budować, trenować i zapisywać modele uczenia maszynowego przy użyciu biblioteki `RevoScaleR`;
- tworzyć wiele kopii modeli uczenia maszynowego wytrenowanych na różnych partycjach danych.

Istotą matematyki nie jest komplikowanie rzeczy prostych,
ale upraszczanie skomplikowanych.

Stan Gudder, profesor matematyki na Uniwersytecie Denver

Rozdział 3
Wstęp do uczenia maszynowego

Najprościej mówiąc, **uczenie maszynowe polega na uczeniu maszyn na przykładach**. Nauczonych na historycznych danych maszyn najczęściej używa się do uzupełniania brakujących danych o nowe, inne od treningowych, obserwacje. Możemy na przykład użyć danych historycznych z rynku nieruchomości do przewidywania cen mieszkań w Katowicach albo danych o działaniu urządzeń w celu przewidzenia zbliżających się awarii.

Takie podejście pozwala rozwiązywać problemy, których nie dałoby się rozwiązać w sposób algorytmiczny. Przykładem może być rozpoznawanie obrazów – wciąż nie potrafimy w ścisły sposób opisać, jak ten proces zachodzi w mózgu człowieka, a nie znając algorytmu, nie możemy go zaimplementować w jakimkolwiek języku programowania. Mimo to maszyny doskonale radzą sobie z rozpoznawaniem obrazów.

Uczenie na przykładach okazało się też skutecznym sposobem radzenia sobie z problemami, dla których znamy algorytmiczne rozwiązania, ale próby zaimplementowania takich algorytmów byłyby bardzo kosztowne albo nieskuteczne. Przykładem problemów tego typu jest wykrywanie możliwych oszustw popełnianych za pomocą kart kredytowych. Nawet po udanym zakodowaniu wszystkich reguł, na podstawie których transakcja powinna być sklasyfikowana jako próba oszustwa, otrzymalibyśmy niezwykle

skomplikowany program (oprócz ogólnych reguł typu *„ta sama karta została użyta w odstępnie krótszym niż godzina w dwóch odległych od siebie o ponad 100 km miejscach”* należałoby uwzględnić reguły specyficzne dla danej osoby, np. *„karta po raz pierwszy została użyta w nocnym klubie w Bangkoku”*). Co gorsza, reguły tego typu często się zmieniają (użytkownicy kart kredytowych nie zachowują się przez cały czas w ten sam sposób, a oszuści dostosowują swoje działania, by obejść wdrożone zabezpieczenia), dlatego chcąc, aby nasz program był skuteczny, musielibyśmy na bieżąco go aktualizować.

Krótko mówiąc, powinniśmy wybrać uczenie maszynowe do rozwiązania problemów, których:

- nie umiemy rozwiązać – dotyczy to sytuacji, gdy nie dysponujemy ekspertem z danej dziedziny, który mógłby znać rozwiązanie naszego problemu, oraz sytuacji, w których wiemy, że problem ma rozwiązanie, ale jest ono nieznane;
- rozwiązania szybko się zmieniają – jeżeli szukana funkcja ciągle się zmienia (tak jak np. funkcja prognozująca kursy spółek na giełdzie), jej zaimplantowanie w postaci statycznych reguł byłoby niepraktyczne;
- rozwiązania muszą być dostosowane do poszczególnych użytkowników – gdyby wszyscy klienci e-sklepu dostawali te same rekomendacje, ich skuteczność byłaby bliska zeru, a może nawet przyniosłaby sklepowi więcej strat niż zysków; jednocześnie indywidualne dobranie każdemu z 10 mln klientów rekomendacji kosztowałoby więcej niż zyski ze sprzedaży zarekomendowanych produktów.

Uczenie maszynowe ma nieskończenie wiele zastosowań, ale nie zawsze jest możliwe. Po pierwsze, **uczenie maszynowe wymaga przykładowych danych**. Jeżeli nie dysponujemy wystarczającą liczbą wartościowych przykładów, nie wytrenujemy modelu uczenia maszynowego. Po drugie, **niezbędnym**

składnikiem uczenia maszynowego jest model – jego architektura (reprezentacja wiedzy) i algorytm użyty do optymalizacji procesu uczenia. Wreszcie, postępy w uczeniu muszą być oceniane, bez tego jakość modelu nie będzie się poprawiała. Po trzecie, **elementem koniecznym do uczenia maszynowego jest zdefiniowanie funkcji kosztu użytej do oceny i poprawy jakości modelu**.

3.1. Rodzaje uczenia maszynowego

Uczenie maszynowe to zbiór różnych technik, dzielonych według wielu kryteriów. **Pierwszym kryterium podziału technik uczenia maszynowego jest format przypadków treningowych**.

Uczenie nienadzorowane (ang. *unsupervised learning*) polega na uczeniu na nieopisanych, niezawierających odpowiedzi przykładach, takich jak dane transakcji pozbawione informacji, czy była to próba oszustwa, albo zdjęć bez opisów tego, co się na nich znajduje. Uczenie nienadzorowane jest stosowane do:

- analizy skupień, więc do podziału przypadków na grupy o podobnych cechach, czyli na **klastry** lub skupienia; algorytm grupowania dzieli obserwacje bez żadnej wcześniejszej wiedzy na temat tego, jak docelowe grupy powinny wyglądać, wyłącznie na podstawie przyjętej w modelu funkcji niepodobieństwa przykładów;
- redukcji wymiarów polegającego na zmniejszeniu liczby zmiennych w taki sposób, żeby w jak najmniejszym stopniu wpłynęło to na utratę informacji;
- wykrywania anomalii, czyli przypadków znacząco różnych od pozostałych, przy założeniu, że nie znamy cech, które miałyby powodować tę różnicę;
- autokodowania polegającego na wykrywaniu przez głębokie sieci neuronowe

abstrakcyjnych cech przypadków, na przykład widocznych na kodowanych obrazach kształtów;

- generowania nowych obrazów będących swoistym połączeniem obrazów treningowych [16].

Uczenie nadzorowane (ang. *supervised learning*) polega na uczeniu zależności między zmiennymi wejściowymi a zmienną docelową, czyli wymaga opisania każdego przypadku. Nauczony na opisanych przypadkach model jest używany do automatycznego opisywania nowych obserwacji.

Opisując przypadki treningowe, nadzorujemy proces uczenia. Jeżeli na przykład opiszemy zdjęcia naszego zwierzaka etykietą `Pies`, nauczony model będzie rozpoznawał na nowych zdjęciach psa. Jeżeli te same zdjęcia opisalibyśmy etykietą `Bokser`, te same zdjęcia byłyby klasyfikowane przez model jako zdjęcia bokserów. Uczenie nadzorowane jest stosowane do:

- klasyfikacji, czyli przypisywania nowych obserwacji do jednej z określonych klas; z klasyfikacją mamy do czynienia wtedy, gdy zmienna docelowa jest dyskretna;
- regresji, która polega na oszacowaniu wartości zmiennej docelowej; o regresji mówimy, gdy zmienna docelowa jest ciągła.

Obecnie ponad 90% modeli uczenia maszynowego wdrożonych do użycia to modele uczenia nadzorowanego. W dalszej części książki termin uczenie maszynowe będzie używany w znaczeniu uczenia nadzorowanego.

Uczenie ze wzmocnieniem (ang. *reinforcement learning*) polega na uczeniu metodą prób i błędów, przy czym model jest nagradzany (wzmocniony) za postępy w uczeniu. Na przykład model kierujący samochodem wyścigowym może być nagradzany za wyprzedzenie przeciwnika, model grający w szachy – za wygranie partii, a chodzący robot – za wykonanie kroku do przodu. Uczenie

ze wzmocnieniem jest stosowane:

- do nauki różnych gier, czego najsłynniejszym przykładem jest przedstawiony we wstępie model AlphaGo Zero;
- w robotyce i automatyce, do nauki działania maszyn;
- do wyszukiwania optymalnej strategii postępowania, na przykład w negocjacjach handlowych.

Drugim kryterium podziału technik uczenia maszynowego jest rodzaj postawionej hipotezy:

- **w modelach regresji** hipoteza ma postać funkcji, a proces uczenia polega na znalezieniu optymalnych wartości parametrów tej funkcji; hipoteza jest tym lepsza, im jest bliższa prawdziwej funkcji zmiennej wyjściowej od zmiennych wejściowych; najpopularniejszymi, używanymi w modelach regresji algorytmami uczenia maszynowego są: regresja liniowa, regresja logistyczna, sztuczne sieci neuronowe i algorytm grupowania metodą k-średnich; nazwa modeli pochodzi od sformułowania *regresja do średniej* i może być myląca, bo tego typu modele mogą być używane zarówno do klasyfikacji (przypisywania obserwacji do jednej z klas), jak i regresji (szacowania wartości zmiennej docelowej);
- **w modelach partycjonujących** hipoteza ma postać reguł dzielących przykłady;. najpopularniejszymi, używanymi w modelach partycjonujących algorytmami uczenia maszynowego są: drzewa decyzyjne, las drzew decyzyjnych i wzmocnione drzewa decyzyjne; modele partycjonujące mogą być także używane do klasyfikacji i do regresji.

3.2. Proces uczenia

W tym podrozdziale ustalimy stosowaną (nie tylko w tej książce) terminologię. Posłuży nam ona do wykazania, że maszyny mogą się uczyć, do wskazania typowych problemów z uczeniem oraz ich rozwiązań.

Zmienne wejściowe, inaczej cechy, **będziemy oznaczali symbolem x_i. Zmienną wyjściową**, inaczej docelową, **będziemy oznaczać symbolem y_i.** Para (x_i, y_i) tworzy pojedynczy przykład treningowy. Indeksy dolne będą opuszczane wszędzie, gdzie nie będzie to miało wpływu na znaczenie formuły. Przykłady treningowe składają się na zbiór treningowy. Zbiór wartości zmiennej wejściowej będziemy oznaczać wielką literą *X*, a zbiór wartości zmiennej wyjściowej wielką literą *Y*.

 Celem uczenia maszynowego jest znalezienie funkcji *h* (z powodów historycznych nazywanej hipotezą), która na podstawie wartości zmiennych wejściowych zwraca wartość zmiennej docelowej: $h: X \rightarrow Y$. Możliwe hipotezy tworzą przestrzeń hipotez, a zadaniem algorytmu uczenia maszynowego jest znalezienie w tej przestrzeni hipotezy najbliższej prawdziwej funkcji $f: X \rightarrow Y$.

Zastanówmy się, na czym polega uczenie (nie tylko maszynowe). Gdyby uczenie sprowadzało się do zapamiętywania przykładów, nie bylibyśmy w stanie zastosować zdobytej wiedzy do nowych obserwacji. Ponieważ żyjemy w zmieniającej się rzeczywistości, a otaczające nas obiekty są nieskończenie złożone, szansa że drugi raz wsiądziemy do identycznego autobusu (tego z pękniętą szybą w drzwiach i niedziałającym kasownikiem), i którym będą podróżować ci sami pasażerowie (jak dziewczyna z czerwoną torebką na ramieniu), jest mikroskopijna. Mimo to raz nauczeni potrafimy się odpowiednio zachować w każdym nowym autobusie.

Zawdzięczamy to naszej zdolności do uogólniania, czyli generalizacji. Generalizacja jest najważniejszym etapem uczenia, również uczenia maszynowego. Pierwszym etapem przekształcania naszych doświadczeń i będących ich wynikiem obserwacji w uporządkowany system informacji – w wiedzę – jest **abstrakcja**. W jej trakcie jakościowe opisy zdarzeń zamieniane są na uogólnione teorie.

Wynikiem abstrakcji nie muszą być jednak liczby. Duża część naszej wiedzy jest reprezentowana za pomocą innych symboli, takich jak etykiety. Przykładem etykiet są nazwy kolorów (zielony, czerwony itd.) oraz opisy stanów cywilnych (żonaty, zamężna, wdowiec, wdowa itd.). Chociaż etykiety są mniej abstrakcyjne od liczb, ich wartości również muszą być (w ramach przyjętych modeli) mierzalne. Aby na przykład określić jakiś kolor jako zielony, musimy dysponować kryteriami pozwalającymi odróżnić ten kolor od innych.

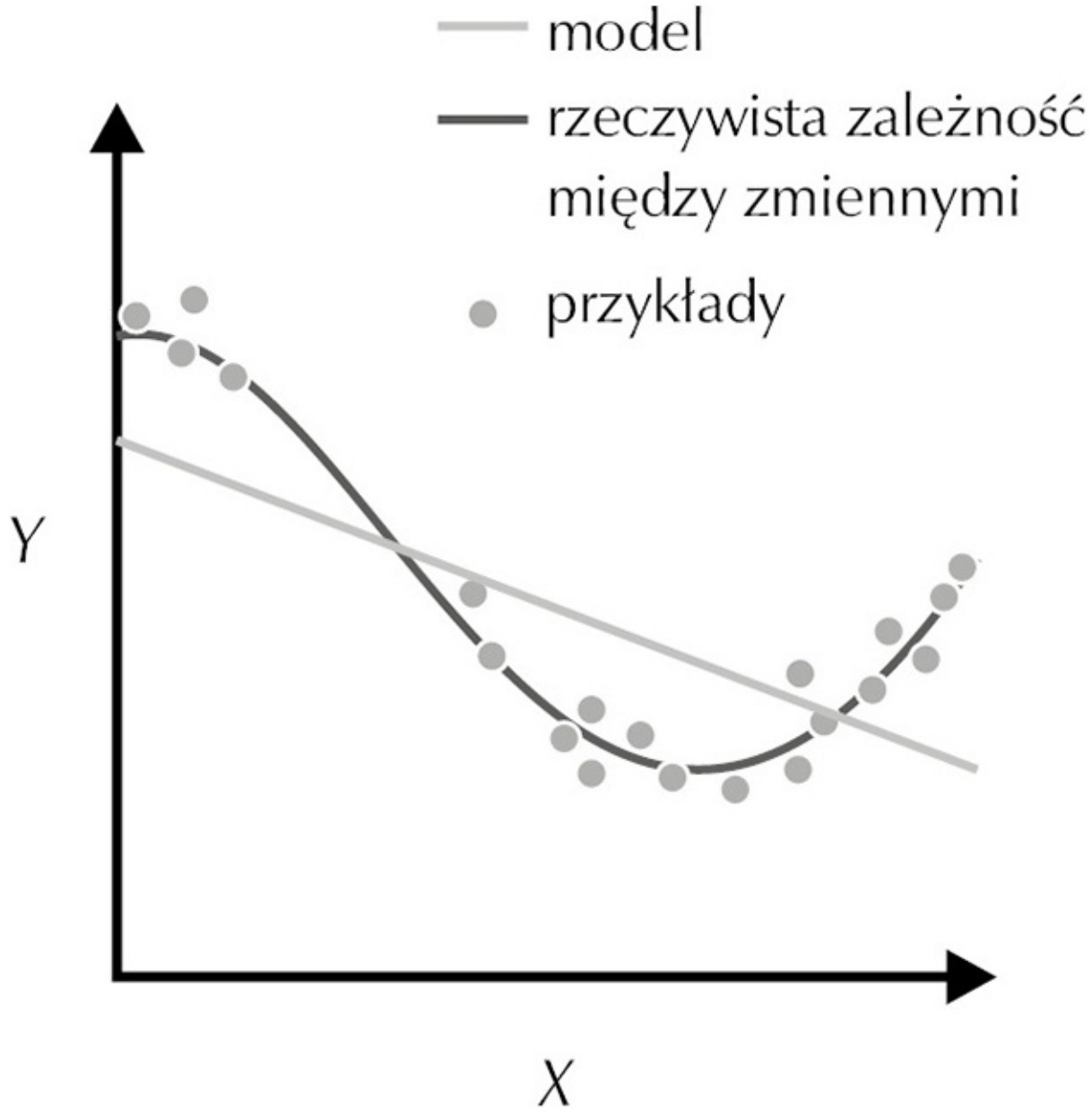
W przypadku ludzi ta dokonywana nieświadomie abstrakcja odbywa się według reguł obowiązujących w przyjętym modelu. W przypadku uczenia maszynowego procesem tym możemy sterować, odpowiednio przygotowując dane źródłowe.

Drugim etapem uczenia jest **generalizacja**. Zdolność do generalizacji pozwala na zastosowanie doświadczeń zdobytych w przeszłości do bieżących sytuacji. Dotyczy to również sztucznej inteligencji [13].

Jednak osoba, która nadmiernie generalizuje posiadane informacje, nie będzie w stanie dostrzec istotnych różnic między sytuacją, w której się w danej chwili znajduje, a uogólnioną sytuacją z przeszłości, i będzie podejmować albo błędne decyzje, albo nawet zawsze te same. W przypadku sztucznej inteligencji problem ten nazywamy **niedopasowaniem** (ang. *underfitting*). Niedopasowane modele uczenia maszynowego działają tak, jakby rozwiązanie problemu sprowadzało się do postępowania według kilku prostych reguł, jeżeli na przykład pasażer zapłacił za przejazd kartą, to da taksówkarzowi napiwek. Takie modele są nieprzydatne, bo są niedokładne – cechuje je duży błąd systematyczny (ang. *bias*).

Niedopasowany model nie nauczył się na przykładach, choć miał taką możliwość, a więc nie ma wiedzy, którą mógłby zastosować do nowych obserwacji. Ten problem jest łatwy do wykrycia, bo niedopasowane modele cechuje duży błąd treningowy (rys. 3.1).

stopień wielomianu = 1



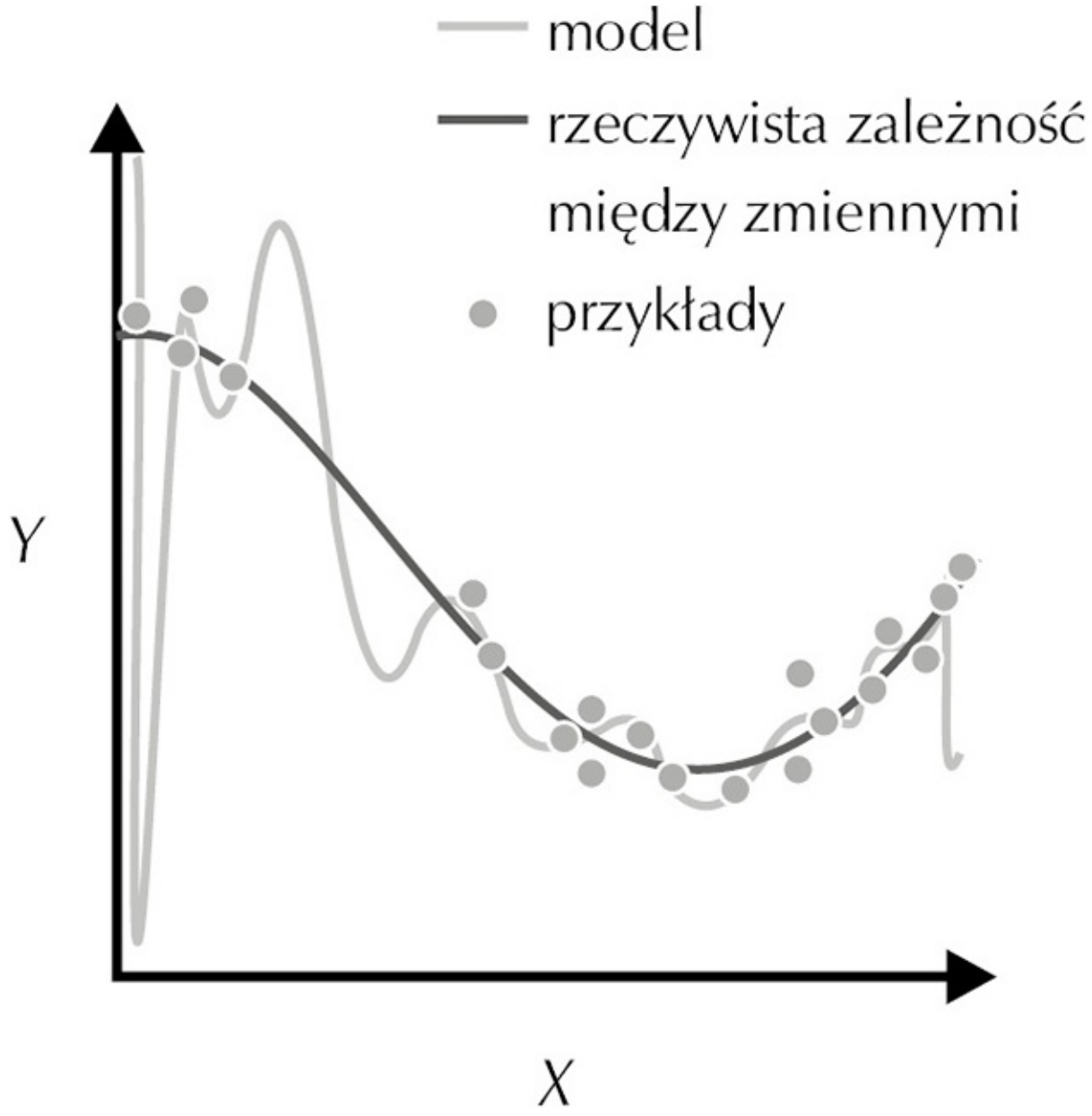
Rysunek 3.1. Najczęstszymi powodami niedopasowania są zbytnia prostota modelu (np. jego liniowość), przedwczesne przerwanie treningu lub niska jakość danych treningowych

Przechodząc z jednej skrajności w drugą, zastanówmy się, jak funkcjonowałaby osoba o fenomenalnej, wręcz fotograficznej pamięci połączonej z całkowitą niezdolnością do uogólniania zapamiętanych danych. Jeżeli ta osoba nie potrafiłaby uogólniać posiadanych informacji, nie byłaby w stanie działać w nowych sytuacjach, bo nie potrafiłaby dostrzec ich zasadniczych cech wspólnych z sytuacjami, których wcześniej doświadczyła. Jednocześnie, taka osoba byłaby bardzo efektywna w sytuacjach, które już zna. Zauważmy, że celem treningu sportowców i żołnierzy jest właśnie wyrobienie w nich automatycznych, powtarzalnych reakcji na te same sytuacje. Jednak w zdecydowanej większości przypadków celem uczenia nie jest wyrobienie automatycznych reakcji na sytuacje treningowe.

W przypadku sztucznej inteligencji problem ten nazywamy **nadmiernym dopasowaniem** (ang. *overtraining*/*overfitting*). Nadmiernie dopasowane modele uczenia maszynowego działają tak, jakby zapamiętały poszczególne przykłady treningowe, nie tylko ich istotne cechy, lecz także przypadkowe błędy w danych (szum informacyjny). Takie modele są mało wiarygodne, bo cechuje je duża zmienność predykcji.

Nadmiernie dopasowany model tak dokładnie zapamiętał przykłady treningowe, że nie potrafi zastosować posiadanej wiedzy do nowych obserwacji. Ten problem jest trudniejszy do wykrycia, bo nadmiernie dopasowane modele cechuje mały (w skrajnych przypadkach zerowy) błąd treningowy i duży błąd generalizacji (rys. 3.2).

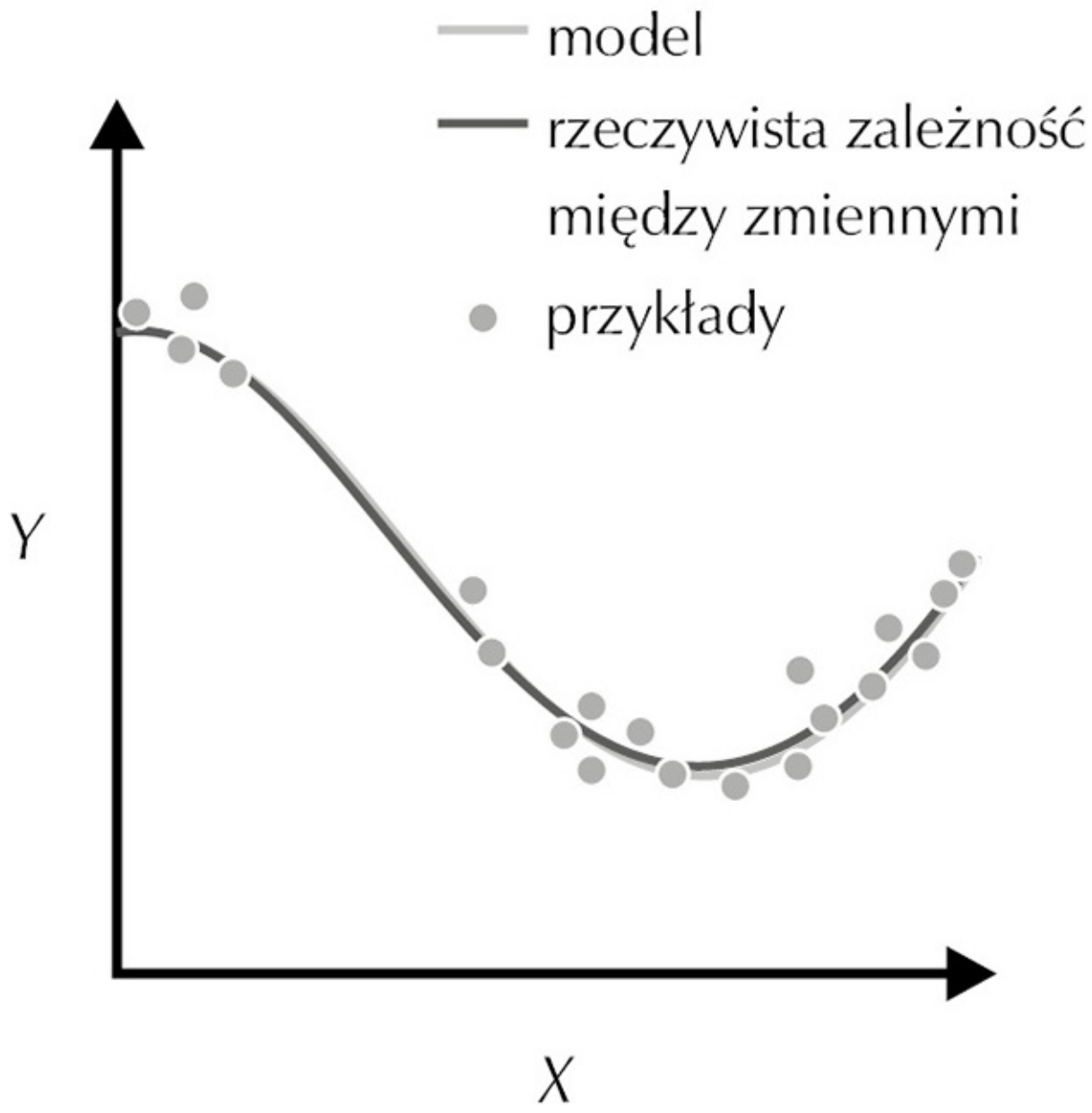
stopień wielomianu = 15



Rysunek 3.2. Nadmierne dopasowanie jest fundamentalnym, bardzo częstym problemem uczenia nadzorowanego. Wśród jego przyczyn jest zbyt mały zbiór treningowy, nadmiernie skomplikowany lub nieodpowiednio sparametryzowany model

Zbudowanie dobrego modelu uczenia maszynowego polega na uniknięciu obu skrajności (niedopasowania i nadmiernego dopasowania). Trudność tego zadania polega na tym, że do treningu są używane wyłącznie dane treningowe. Jak widzieliśmy, zminimalizowanie błędu treningowego samo w sobie łatwe, może oznaczać nadmierne dopasowanie modelu. Tymczasem naszym celem jest zbudowanie właściwie dopasowanego modelu (rys. 3.3).

stopień wielomianu = 4



Rysunek 3.3. Właściwe dopasowanie modelu wymaga znalezienia złotego środka między jego niedopasowaniem a nadmiernym dopasowaniem

Oba modele pokazane na rysunkach 3.2 i 3.3 cechują się małym błędem treningowym. Jak więc wykryć nadmierne dopasowanie? Dotarliśmy w naszych rozważaniach do pytania o wykonalność uczenia maszynowego.

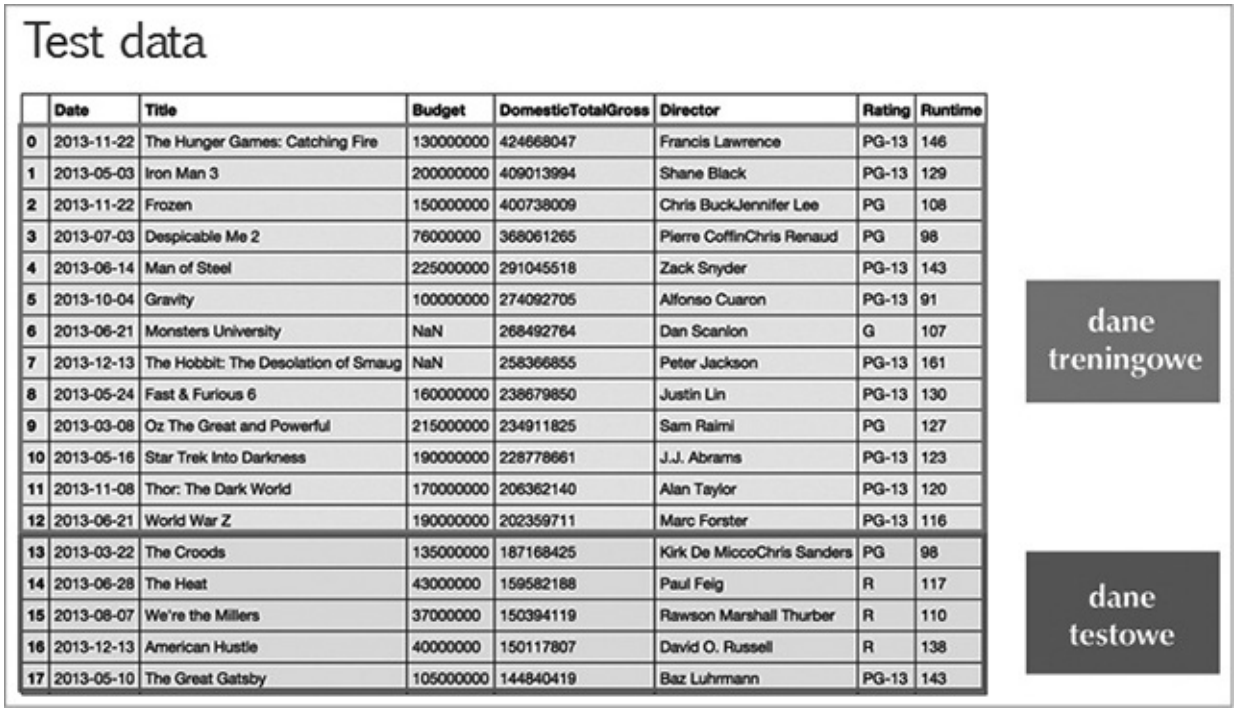
Przyjmijmy na chwilę, że trening modelu polega na sprawdzeniu wszystkich możliwych hipotez (funkcji zmiennej wyjściowej od zmiennych wejściowych). Najlepszym modelem będzie ten, który najdokładniej odzwierciedli rzeczywistą zależność między zmiennymi wejściowymi a wyjściową. Problem polega na tym, że nie znamy tej zależności, bo dysponujemy tylko próbą danych. Jeżeli jednak ta próba jest wystarczająco duża (N), to jej średnia v będzie zbliżona do wartości oczekiwanej μ. Nierówność Hoeffdinga pozwala określić prawdopodobieństwo, z jakim różnica między tymi wartościami będzie mniejsza od założonej wartości granicznej (ε):

$$P[\,|v - \mu| > \varepsilon] \leq 2e^{-2\varepsilon^2 N}$$

Ta nierówność, udowodniona w 1963 r. przez Wassila Hoeffdinga, pozwala z określonym zaufaniem stosować nauczone na próbie danych modele do nowych, niewystępujących w próbie treningowej obserwacji [14].

Po wykazaniu teoretycznej wykonalności uczenia maszynowego pora na praktyczne wskazówki, jak zbudować właściwie dopasowany model.

Po pierwsze, **do oceny modelu musimy użyć danych testowych**. Zbiory treningowy i testowy muszą być rozłączne, czyli danych treningowych nie możemy ani razu użyć do oceny modelu, a danych testowych nie możemy ani razu użyć do treningu. Bardziej rygorystyczni specjaliści uczenia maszynowego uważają, że danych testowych nie tylko nie wolno użyć do treningu, lecz także nie powinno się nawet na nie spojrzeć (rys. 3.4).



	Date	Title	Budget	DomesticTotalGross	Director	Rating	Runtime
0	2013-11-22	The Hunger Games: Catching Fire	130000000	424668047	Francis Lawrence	PG-13	146
1	2013-05-03	Iron Man 3	200000000	409013994	Shane Black	PG-13	129
2	2013-11-22	Frozen	150000000	400738009	Chris BuckJennifer Lee	PG	108
3	2013-07-03	Despicable Me 2	76000000	368061265	Pierre CoffinChris Renaud	PG	98
4	2013-06-14	Man of Steel	225000000	291045518	Zack Snyder	PG-13	143
5	2013-10-04	Gravity	100000000	274092705	Alfonso Cuaron	PG-13	91
6	2013-06-21	Monsters University	NaN	268492764	Dan Scanlon	G	107
7	2013-12-13	The Hobbit: The Desolation of Smaug	NaN	258366855	Peter Jackson	PG-13	161
8	2013-05-24	Fast & Furious 6	160000000	238679850	Justin Lin	PG-13	130
9	2013-03-08	Oz The Great and Powerful	215000000	234911825	Sam Raimi	PG	127
10	2013-05-16	Star Trek Into Darkness	190000000	228778661	J.J. Abrams	PG-13	123
11	2013-11-08	Thor: The Dark World	170000000	206362140	Alan Taylor	PG-13	120
12	2013-06-21	World War Z	190000000	202359711	Marc Forster	PG-13	116
13	2013-03-22	The Croods	135000000	187168425	Kirk De MiccoChris Sanders	PG	98
14	2013-06-28	The Heat	43000000	159582188	Paul Feig	R	117
15	2013-08-07	We're the Millers	37000000	150394119	Rawson Marshall Thurber	R	110
16	2013-12-13	American Hustle	40000000	150117807	David O. Russell	R	138
17	2013-05-10	The Great Gatsby	105000000	144840419	Baz Luhrmann	PG-13	143

Rysunek 3.4. Nieużyte do treningu modelu pary (x_i, y_i) tworzą zbiór testowy

Dane treningowe zostaną użyte do znalezienia ukrytych w nich zależności między zmiennymi wejściowymi a wyjściową. Algorytmy uczenia maszynowego wielokrotnie analizują zbiór treningowy, weryfikując różne hipotezy. Porównywanie stawianych przez te algorytmy hipotez wymaga zdefiniowania funkcji kosztu. Im jej wartość będzie mniejsza, tym lepsza weryfikowana hipoteza. **Wartość funkcji kosztu (błąd, koszt, ryzyko, zysk) określa rozbieżność między wynikami predykcji z rzeczywistymi wartościami zmiennej wyjściowej.** Wyznaczenie tej rozbieżności może być zrealizowane na wiele sposobów, opisanych w dalszej części książki.

Użycie funkcji kosztu na zbiorze treningowym pozwala wyliczyć błąd treningowy. **Błąd treningowy (empiryczny) jest średnim błędem predykcji dla danych treningowych**. Jak wspomniałem, błąd treningowy zawsze można ograniczyć do zera, wystarczy tylko, żeby model dokładnie zapamiętał wszystkie przykłady.

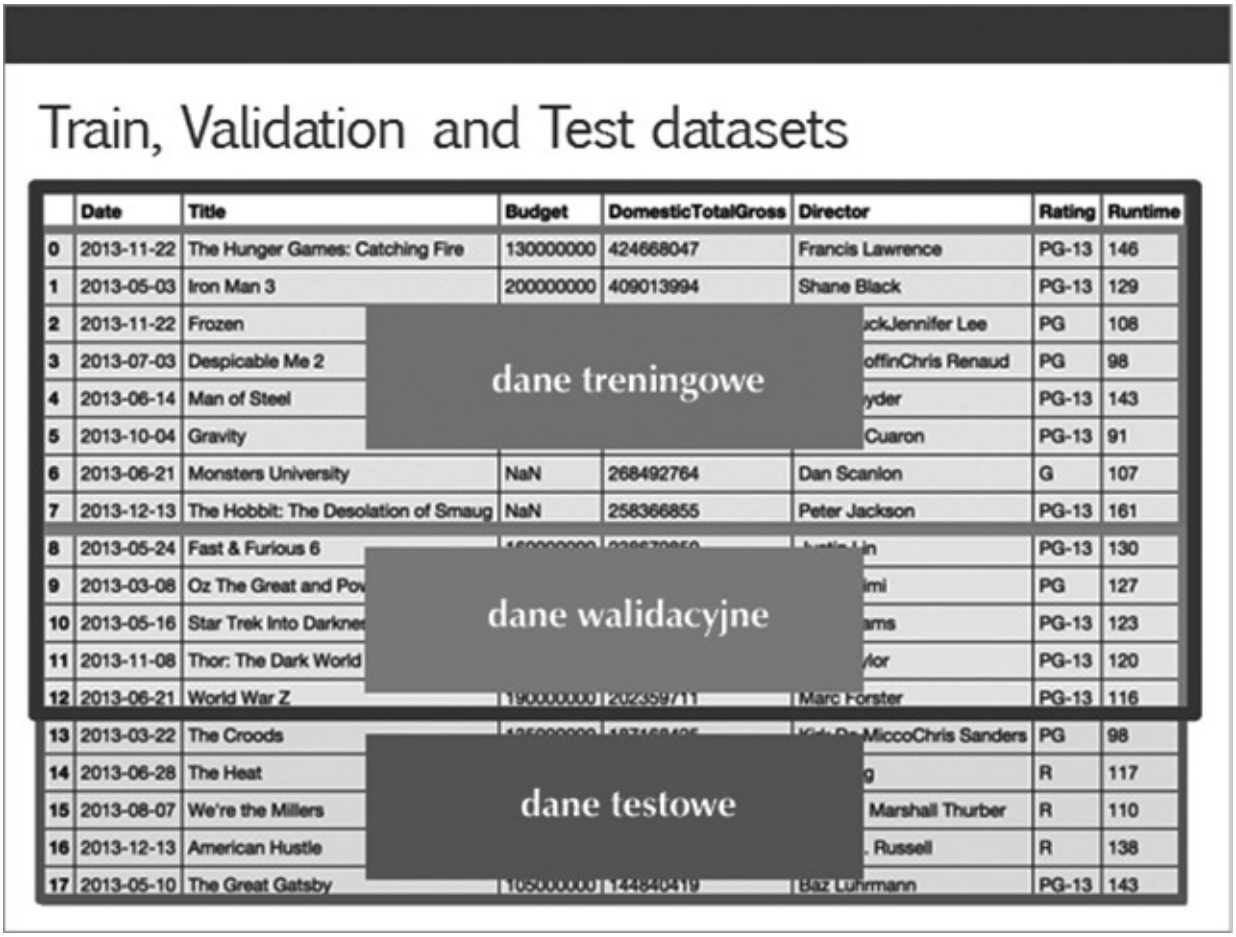
Naszym celem jest jednak **zminimalizowanie błędu testowego (realnego),**

nazywanego też błędem generalizacji, czyli uśrednionego błędu predykcji dla danych testowych. Dokładnie taki jest sens funkcyjnej definicji uczenia maszynowego, według której: Dla danej funkcji kosztu θ (theta) i próby danych D wybranej z populacji P o nieznanym rozkładzie należy znaleźć funkcję h o jak najmniejszym oczekiwanym błędzie ε dla populacji P względem funkcji kosztu θ [15].

Modele powinniśmy wybierać na podstawie ich błędu realnego. W praktyce błędu realnego nie wylicza się, stosując nierówność Hoeffdinga do błędu empirycznego, tylko mierzy się błąd otrzymany dla zbioru testowego. Takie podejście pozwala wykryć nadmierne dopasowanie modelu, ale jak podkreślałem, zbiór testowy może być użyty tylko raz. Tymczasem porównywanie modeli oznacza wielokrotne sprawdzanie ich błędu realnego. **Z każdym kolejnym użyciem danych testowych zwiększamy ryzyko nadmiernego dopasowania modelu do danych testowych**. Dopasowywanie modelu do danych testowych przebiega znacznie wolniej niż dopasowywanie do danych treningowych, ale ma równie katastrofalne konsekwencje.

 Nadmierne dopasowanie definiuje się jako wybór hipotezy *h*, której błąd treningowy jest mniejszy od alternatywnej hipotezy *h*', ale jej błąd testowy jest większy niż błąd testowy alternatywnej hipotezy *h*'.

Wykrycie nadmiernego dopasowania modelu do danych testowych wymaga sprawdzenia go na nowych, nieużytych wcześniej danych. Dlatego zbiór treningowy dzieli się na dwa: dane treningowe i walidacyjne (rys. 3.5).



	Date	Title	Budget	DomesticTotalGross	Director	Rating	Runtime
0	2013-11-22	The Hunger Games: Catching Fire	130000000	424668047	Francis Lawrence	PG-13	146
1	2013-05-03	Iron Man 3	200000000	409013994	Shane Black	PG-13	129
2	2013-11-22	Frozen			ckJennifer Lee	PG	108
3	2013-07-03	Despicable Me 2			offinChris Renaud	PG	98
4	2013-06-14	Man of Steel			yder	PG-13	143
5	2013-10-04	Gravity			Cuaron	PG-13	91
6	2013-06-21	Monsters University	NaN	268492764	Dan Scanlon	G	107
7	2013-12-13	The Hobbit: The Desolation of Smaug	NaN	258366855	Peter Jackson	PG-13	161
8	2013-05-24	Fast & Furious 6				PG-13	130
9	2013-03-08	Oz The Great and Pov			imi	PG	127
10	2013-05-16	Star Trek Into Darkne			ams	PG-13	123
11	2013-11-08	Thor: The Dark World			lor	PG-13	120
12	2013-06-21	World War Z	190000000	202359711	Marc Forster	PG-13	116
13	2013-03-22	The Croods			MiccoChris Sanders	PG	98
14	2013-06-28	The Heat			g	R	117
15	2013-08-07	We're the Millers			Marshall Thurber	R	110
16	2013-12-13	American Hustle			Russell	R	138
17	2013-05-10	The Great Gatsby	105000000	144840419	Baz Luhrmann	PG-13	143

Rysunek 3.5. Dane treningowe używane są do uczenia, walidacyjne do optymalizacji modelu, a testowe do końcowej oceny optymalnego modelu

Błąd testowy ma dwie składowe: błąd systematyczny, czyli błąd estymacji, oraz wynikający ze zmienności modelu błąd aproksymacji. Niech *f* będzie ocenianym modelem, wybranym z możliwych modeli danego typu *F*, wtedy błąd testowy modelu równa się

$$error(f) = [error(f) - \min error(f^* \in F] + [\min error(f)]$$

Pierwszy człon równania, błąd estymacji, mierzy, o ile gorszy jest model od modelu idealnego. Drugi człon, błąd aproksymacji, mierzy jakość modeli reprezentujących wiedzę w określony sposób.

Dane najczęściej dzieli się losowo; upraszcza to ocenę i optymalizację modeli.

Najwięcej przykładów trafia do danych treningowych. Danych walidacyjnych i testowych potrzebujemy tylko tyle, ile potrzeba do wiarygodnej oceny modeli. Jeżeli dysponujemy kilkudziesięcioma tysiącami przykładów, powinniśmy podzielić je w proporcjach 70% (dane treningowe), 20% (dane walidacyjne) i 10% (dane testowe). Dla kilkuset tysięcy przykładów proporcje te powinny wynosić 80%, 15%, 5%. Gdybyśmy mieli do dyspozycji kilkadziesiąt milionów przykładów, proporcje te mogłyby wynieść 95%, 4%, 1%, bo kilkaset tysięcy przykładów testowych wystarcza aż nadto do wiarygodnej oceny modeli. Jeśli jednak brakuje nam danych, nie powinniśmy ograniczać wielkości zbiorów walidacyjnego i testowego. Chociaż większy procent danych treningowych daje większe szanse na lepszy model, to niedostateczna liczba danych walidacyjnych zwiększa ryzyko jego nadmiernego dopasowania, a zbyt mało danych testowych uniemożliwia wiarygodną ocenę modelu. W takich sytuacjach powinniśmy zastosować, opisaną w dalszej części książki, *k*-krotną walidację krzyżową. W kolejnym podrozdziale przyjrzymy się budowie i działaniu modeli regresji i modeli partycjonujących.

3.3. Modele regresji

Najprostsza hipoteza modelu regresji polega na przyjęciu liniowej zależności między zmiennymi wejściowymi a zmienną docelową, czyli założeniu że przybliżone wartości zmiennej docelowej można obliczyć, używając równania liniowego. Taki model nazywamy modelem regresji liniowej.

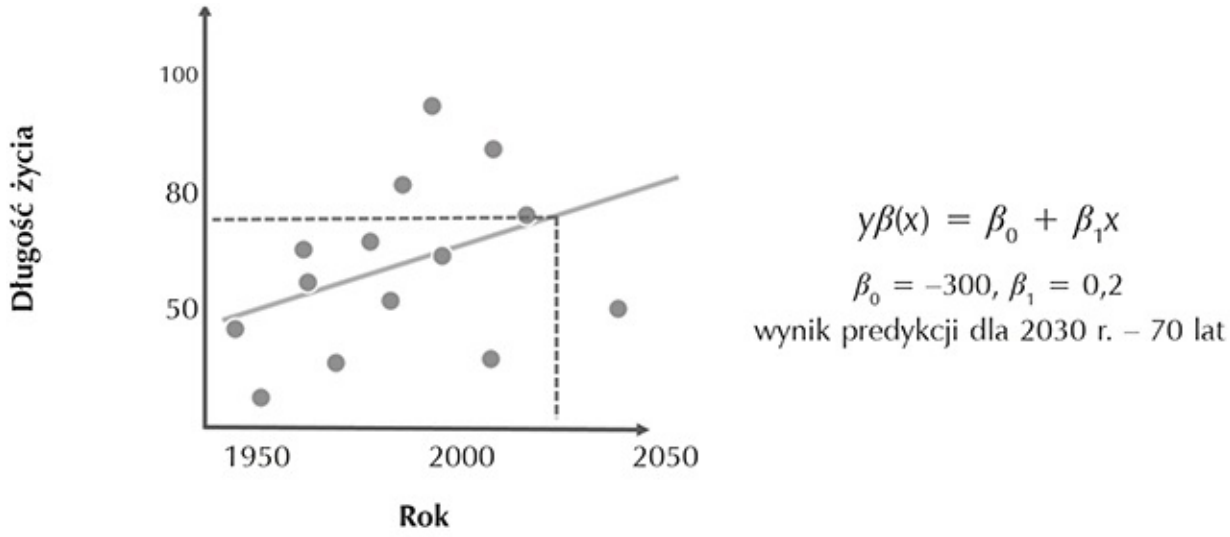
Najprostszym typem regresji liniowej jest model z jedną zmienną wejściową x i zmienną wyjściową y. W takim przypadku celem modelu jest znalezienie parametrów funkcji, która na podstawie wartości x_i będzie zwracała wartość y_i, czyli funkcji h:

$$y = f(x) = b_0 + b_1 x$$

gdzie wyraz wolny b_0 (ang. *intercept*) jest wartością zmiennej y_i dla wartości zmiennej x_i równej 0, a współczynnik regresji b_1 (ang. *slope*) określa, o ile zmienia się wartość zmiennej y_i wraz ze zmianą zmiennej x_i.

Uczenie modelu regresji polega na znalezieniu wystarczająco dobrego przybliżenia funkcji zmiennej wyjściowej od zmiennych wejściowych, czyli hipotezy bliskiej prawdziwej funkcji.

Na potrzeby tego przykładu przyjmijmy uproszczoną hipotezę zakładającą liniową zależność między rokiem a średnią długością życia Polaków. Gdyby ta hipoteza okazała się prawdziwa, nasz model obliczyłby średnią długość życia w Polsce w zadanym roku (rys. 3.6).



Rysunek 3.6. Graficzna reprezentacja modelu regresji liniowej zakładającego, że średnią długość życia Polaków można obliczyć na podstawie roku, używając równania liniowego $y = -300 + 0{,}2 * x$

Model ten zbudujemy w RStudio, używając danych statystycznych z pakietu `gapminder`. Do pracy z danymi i ich wizualizacji użyjemy funkcji pakietu `tidyverse`, a pracę z modelami ułatwią nam funkcje i operator `%>%` (*pipe*) pakietu `modelr`:

```
library(gapminder)
library(tidyverse)
library(modelr)
```

Zacznijmy od sprawdzenia wielkości zbioru danych:

```
dim(gapminder)
```

```
[1] 1704    6
```

Zbiór `gapminder` liczy 1704 wierszy i 6 kolumn. Przyjrzyjmy się strukturze tych danych:

```
str(gapminder)
Classes 'tbl_df', 'tbl' and 'data.frame':  1704 obs. of  6
variables:
 $ country  : Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ continent: Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3 3 3 3 3 ...
 $ year     : int  1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
 $ lifeExp  : num  28.8 30.3 32 34 36.1 ...
 $ pop      : int  8425333 9240934 10267083 11537966 13079460 14880372 12881816 13867957
16317921 22227415 ...
 $ gdpPercap: num  779 821 853 836 740 ...
```

Jest to tabela, a dokładnie obiekt klasy `tibble`. Pierwsza kolumna `country` zawiera nazwy 142 krajów, druga kolumna `continent` – nazwy 5 kontynentów. W tych dwóch kolumnach zapisane są czynniki (ang. *factor*), czyli zmienne kategoryczne. Pozostałe kolumny tabeli przechowują zmienne numeryczne. Spójrzmy na te dane:

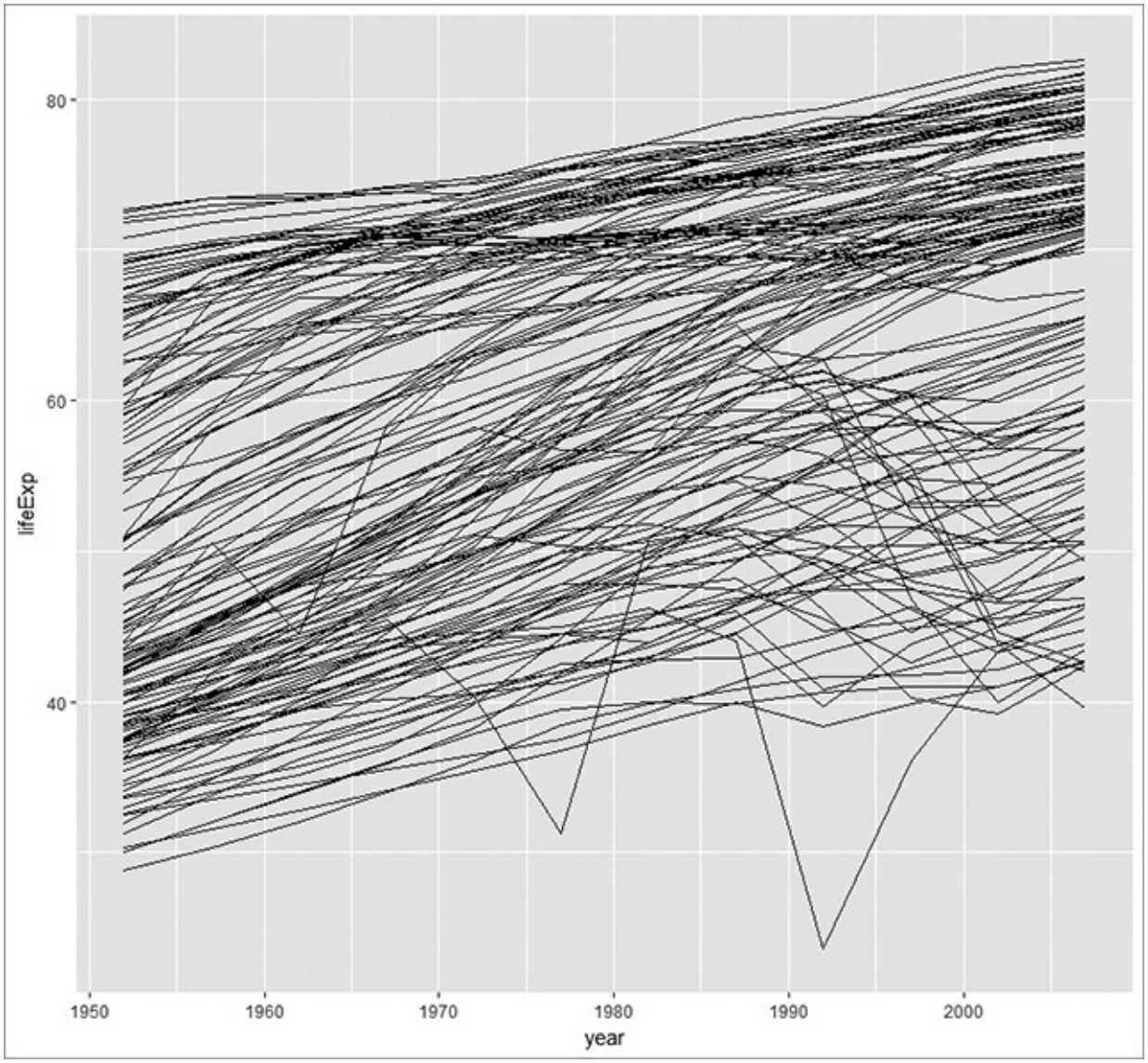
```
gapminder
```

```
# A tibble: 1,704 × 6
        country continent  year lifeExp      pop gdpPercap
         <fctr>    <fctr> <int>   <dbl>    <int>     <dbl>
1   Afghanistan      Asia  1952  28.801  8425333  779.4453
2   Afghanistan      Asia  1957  30.332  9240934  820.8530
3   Afghanistan      Asia  1962  31.997 10267083  853.1007
4   Afghanistan      Asia  1967  34.020 11537966  836.1971
5   Afghanistan      Asia  1972  36.088 13079460  739.9811
6   Afghanistan      Asia  1977  38.438 14880372  786.1134
7   Afghanistan      Asia  1982  39.854 12881816  978.0114
8   Afghanistan      Asia  1987  40.822 13867957  852.3959
9   Afghanistan      Asia  1992  41.674 16317921  649.3414
10  Afghanistan      Asia  1997  41.763 22227415  635.3414
# ... with 1,694 more rows
```

Każdy wiersz tabeli odpowiada jednemu przykładowi, a każda kolumna jednej zmiennej. Dane są więc uporządkowane (ang. *tidy data*) i gotowe do dalszej analizy.

Zobaczmy, jak wygląda zależność między zmiennymi `year` a `lifeExp` dla poszczególnych wartości zmiennej `country` (rys. 3.7):

```
ggplot(gapminder, aes(year, lifeExp, group = country)) +
  geom_line()
```



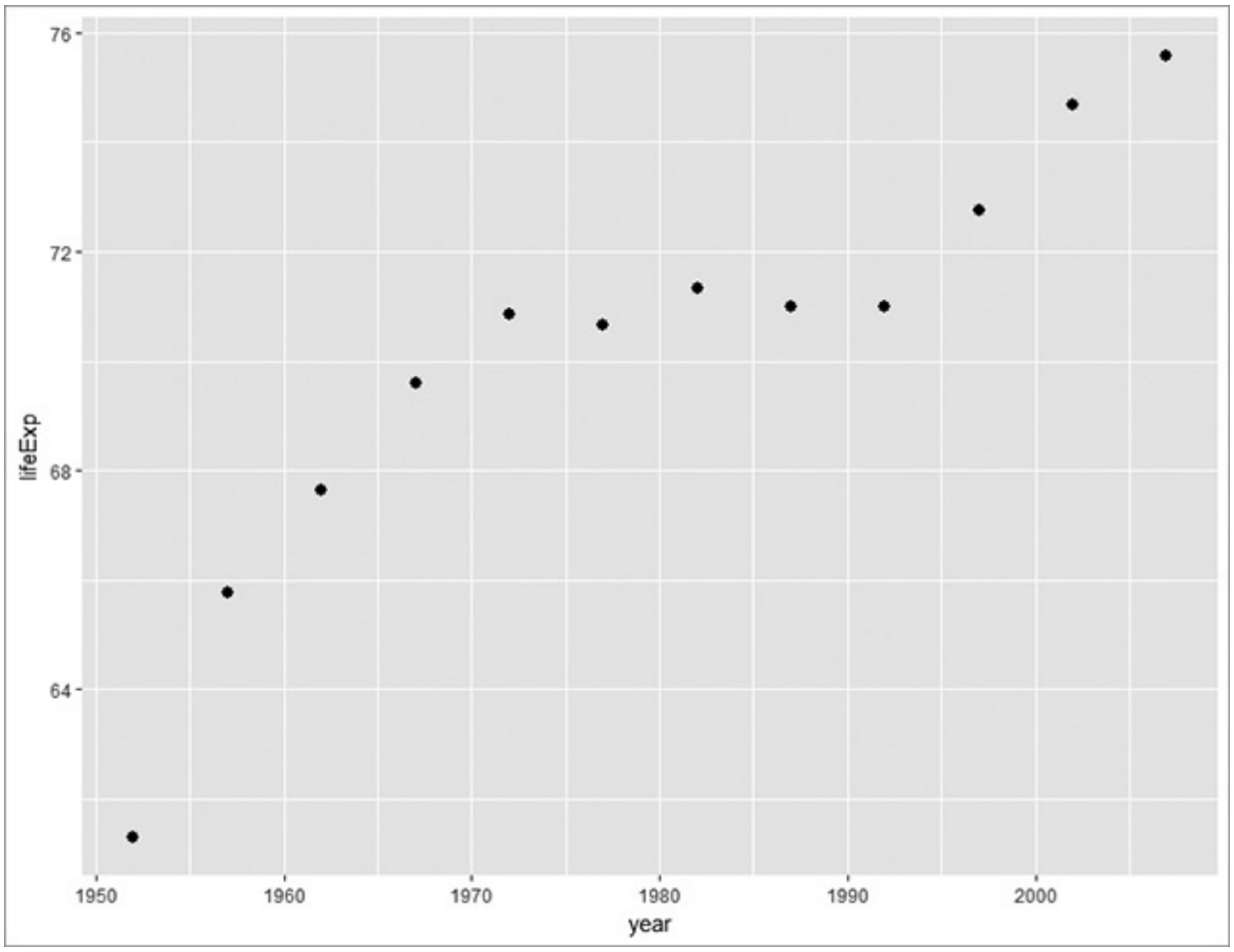
Rysunek 3.7. Dla niektórych krajów założenie linowej zależności między rokiem a średnią długością życia mieszkańców wydaje się być spełnione

Ograniczmy naszą analizę do Polski. W tym celu wybierzemy przykłady dotyczące naszego kraju, zapiszemy je w tabeli `pl` i sprawdzimy zależność między zmiennymi `year` a `lifeExp`, wyświetlając je na wykresie punktowym (rys. 3.8).

```
pl <- filter (gapminder, country =="Poland")
ggplot(pl, aes(year, lifeExp)) +
  geom_point(size = 2.5)
```

Naszym celem jest zbudowanie modelu regresji liniowej, który z możliwie największą dokładnością obliczy średnią długość życia Polaków w danym roku, czyli znalezienie optymalnych wartości parametrów b_0 i b_1 i równania liniowego

$$lifeExp = b_0 + b_1 * year$$



Rysunek 3.8. Przykładowe dane dotyczące średniej długości życia Polaków w latach 1952–2007

Zadanie to można rozwiązać na kilka sposobów. Poniżej znajduje się dokładny opis poglądowego rozwiązania wyjaśniającego zasadę działania modeli regresji. Bardziej niecierpliwi Czytelnicy mogą już zobaczyć rozwiązanie, wystarczy, że wywołają poniższą funkcję:

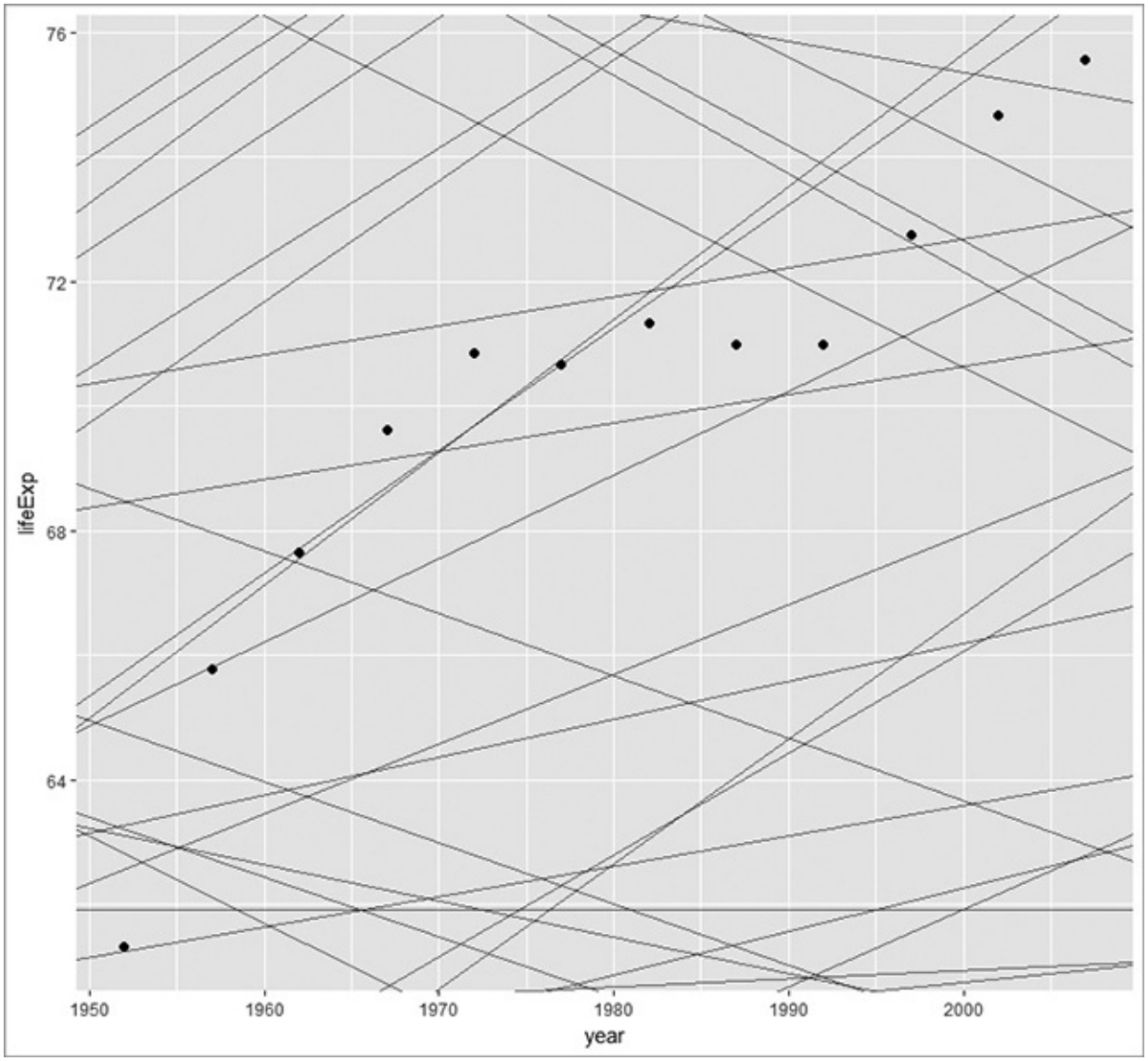
```
ggplot(pl, aes(year, lifeExp)) +
  geom_point(size = 2) +
  geom_smooth(method = "lm", size = 1.5, colour = "red")
```

Zacznijmy od utworzenia 5000 przypadkowych modeli. Dla każdego z nich wylosujemy wartości parametrów b_0 i b_1. 5000 kombinacji wartości parametrów, nawet jeżeli liczba tych parametrów jest ograniczona do dwóch, to znikomy promil wszystkich możliwości. Dlatego na potrzeby przykładu ograniczę zakres wartości obu parametrów do <–400,400> dla parametru b_0 i <–1,1> dla parametru b_1. Ograniczymy w ten sposób przestrzeń hipotez do 5000.

```
models <- tibble(
  a1 = runif(5000, -400, 400),
  a2 = runif(5000, -1, 1) )
```

Nałóżmy tak utworzone losowe modele na wykres z przykładowymi danymi (rys. 3.9):

```
ggplot(pl, aes(year, lifeExp)) +
  geom_abline( aes( intercept = a1, slope = a2),
                data = models, alpha = 1/2 ) +
  geom_point(size = 2)
```



Rysunek 3.9. Dla tych samych przykładów można utworzyć nieskończoną liczbę modeli. Na wykresie widać, że niektóre z nich lepiej, a niektóre gorzej odzwierciedlają ukrytą w przykładowych danych zależność między rokiem a średnią długością życia

Zastanówmy się, jak można oceniać i porównywać modele. Znalezienie odpowiedzi na to pytanie pozwoli nam określić metodę dobierania wartości parametrów b_0 i b_1, czyli określić algorytm uczenia modelu. Skoro zadaniem modelu regresji jest znalezienie funkcji, która jak najdokładniej obliczy wartości zmiennej wyjściowej na podstawie wartości zmiennych wejściowych, sensowna odpowiedź może polegać na minimalizacji różnic między wartościami obliczonymi dla przypadków treningowych a rzeczywistymi. Otrzymamy w ten

sposób równanie:

$$e_i = y_i - h$$

którego wartość należy zminimalizować.

Na tej podstawie możemy zdefiniować **funkcję kosztu *θ*** (theta), która pozwoli obliczyć różnicę między predykcjami modelu a danymi treningowymi, czyli między rzeczywistymi a obliczonymi przez model wartościami zmiennej docelowej.

Błędy mogą polegać na niedoszacowaniu lub przeszacowaniu wartości zmiennej docelowej. Ponieważ należy uwzględnić błędy obu typów, błąd modelu możemy obliczyć jako sumę wartości bezwzględnych błędów dla poszczególnych przykładów albo sumę ich kwadratów. W tym przypadku zastosujemy **funkcję najmniejszych kwadratów błędu *L2*** (ang. *Sum of Squared Errors, SSE*), bo uważamy, że koszt popełnienia błędu przez model rośnie nieliniowo (kwadratowo) wraz z wartością błędu. Wartość SSE obliczymy ze wzoru:

$$SSE = \sum_{i=1}^{n} e_i^2$$

gdzie e_i jest błędem pojedynczej predykcji. Model będzie tym dokładniejszy, im mniejsza będzie średnia wartość SSE. Należy więc tak dobrać jego parametry b_0 i b_1, żeby zminimalizować wartość funkcji kosztu

$$J(\theta) = \frac{1}{2n} \sum_{i=1}^{n} (h_\theta(x^i) - y^i)^2$$

Okazuje się, że **uczenie maszynowe polega na szukaniu takich wartości parametrów modelu, dla których przyjęta funkcja kosztu przyjmuje minimum**. Aby lepiej zrozumieć to zdanie, utwórzmy model regresji liniowej dla arbitralnie wybranych wartości parametrów $b_0 = -300$ i $b_1 = 0.2$

```
model1 <- function(a,data) {
  a[1] +data$year * a[2]
}
model1(c(-300,0.2), pl)
```

Funkcja `errors` pomoże nam obliczyć koszt modelu:

```
errors <- function (mod,data) {
  diff <- data$lifeExp - model1(mod,data)
  sqrt(mean(diff^2))
}
errors(c(-300,0.2), pl)
```

```
25.76571
```

Dla wartości parametrów $b_0 = -300$ i $b_1 = 0{,}2$ koszt modelu zmierzony sumą kwadratów błędów dla przykładów treningowych wyniósł 25,76571. Czyli nasz model myli się na danych treningowych średnio o prawie 26 lat.

Policzmy błędy empiryczne (treningowe) wszystkich losowych modeli. Pomoże nam w tym funkcja `pl_dist`, którą wywołamy dla wszystkich wierszy tabeli `models` (a więc dla każdego utworzonego modelu). Do wielokrotnego wywołania funkcji i zapisania jej wyniku w kolumnie tabeli model skorzystamy z funkcji `map2_dbl` pakietu `purrr`:

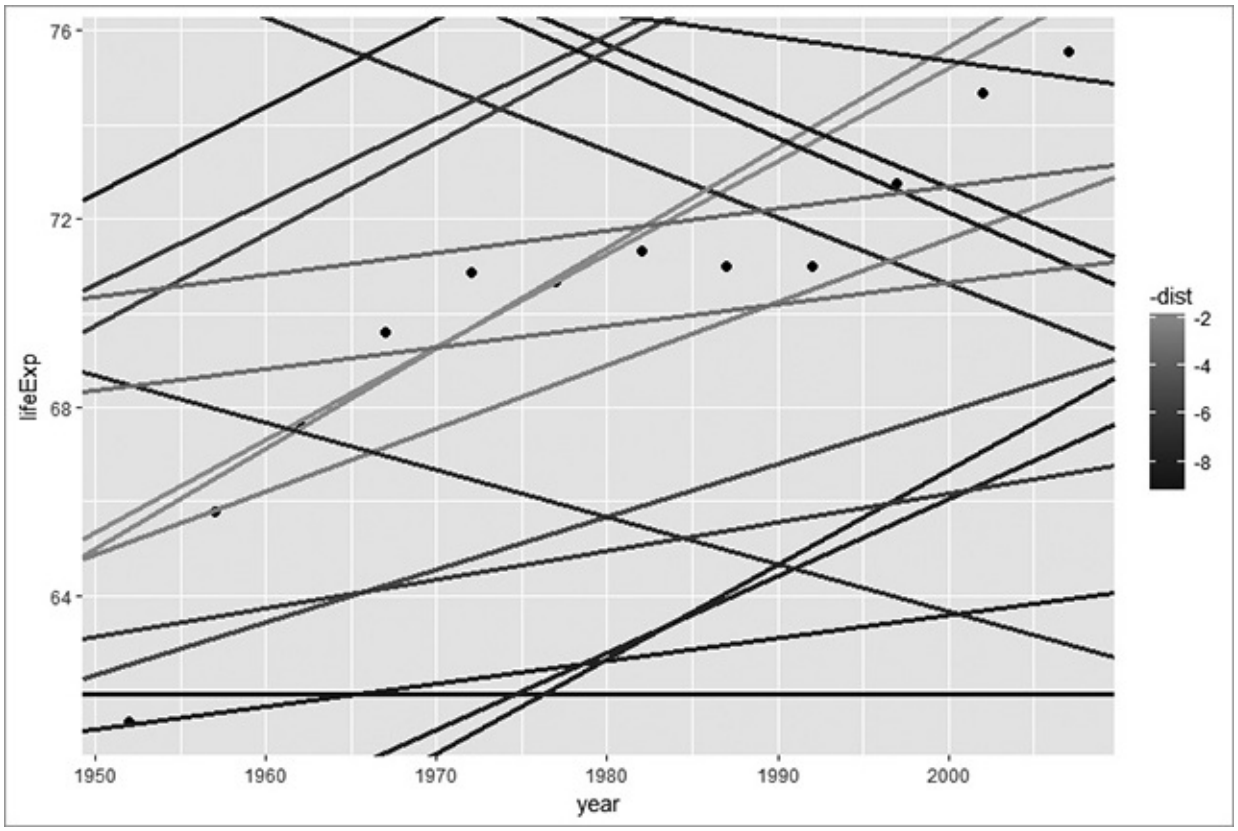
```
       pl_dist <- function(a1,a2) {
  errors(c(a1,a2),pl
}

models <- models %>%
  mutate(dist = map2_dbl(a1,a2,pl_dist))
models
```

```
# A tibble: 5,000 × 3
             a1            a2       dist
          <dbl>         <dbl>      <dbl>
1    -70.08865 -0.10088613   340.0116
2   -194.40309   0.42042689   567.6701
3    120.39034   0.54805330 1135.1021
4    -18.68663   0.81221465 1518.9533
5    307.73967   0.73610276 1694.7044
6   -284.47397 -0.42094883 1187.9678
7    165.98542   0.81507005 1709.2737
8   -316.54454   0.02817557   330.9639
9    129.03006 -0.24037997   417.0497
10 -345.26495 -0.97150733 2338.6280
# ... with 4,990 more rows
```

Dziewiętnaście najlepszych modeli (modeli o najmniejszym błędzie empirycznym) pokazano na rysunku 3.10.

```
ggplot(pl, aes(year, lifeExp)) +
  geom_point(size = 2) +
  geom_abline(aes(intercept = a1, slope = a2, color =-dist), size =
1.1,
                data = filter(models,  rank(dist) <20) )
```

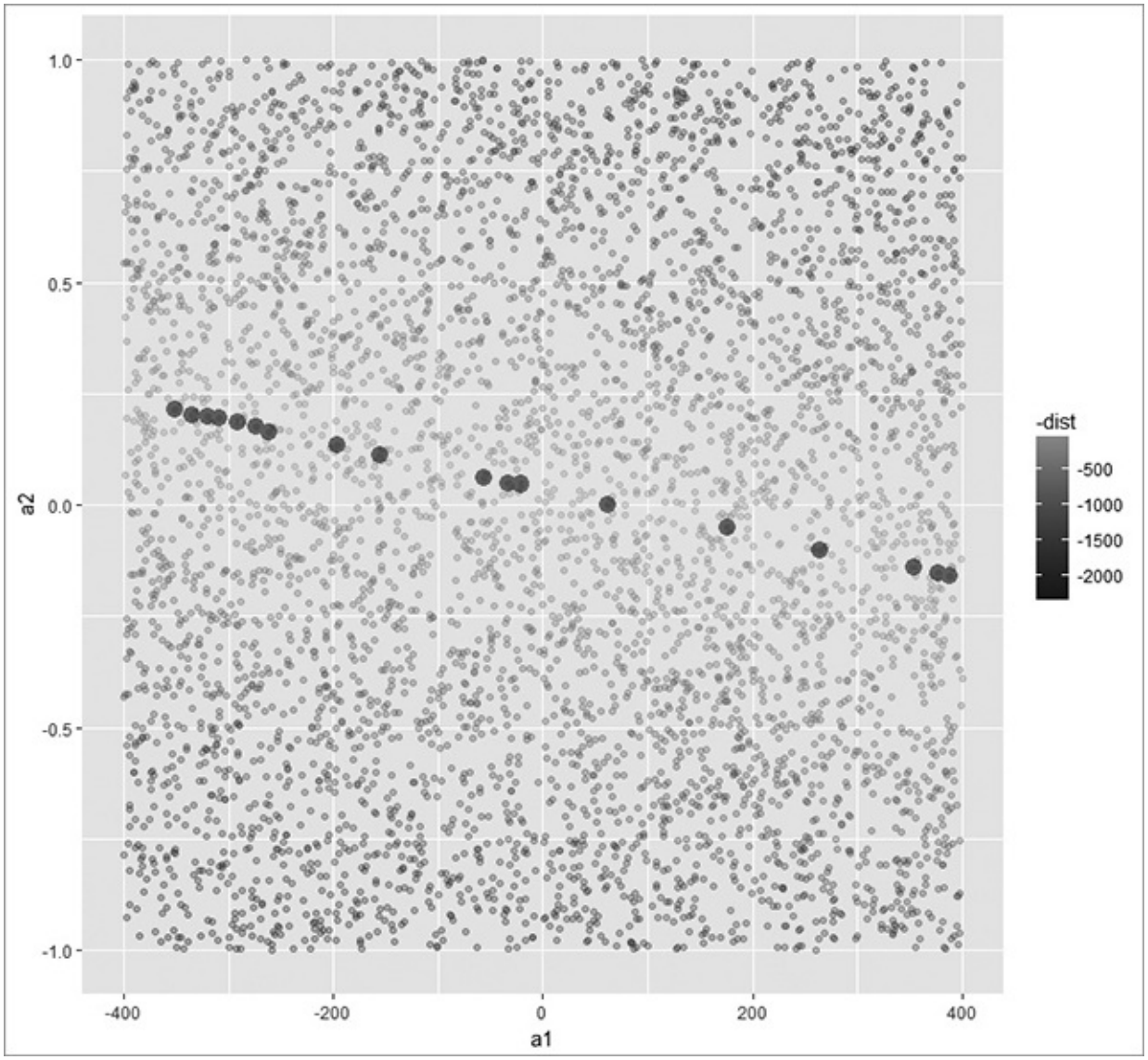


Rysunek 3.10. Im jaśniejsza jest linia reprezentująca model, tym mniejszy koszt modelu, czyli tym lepiej odzwierciedla on zależność w danych treningowych między zmienną wejściową a docelową

W praktyce wartości parametrów modelu nie ustala się losowo. Takie podejście nie tylko byłoby czasochłonne, lecz także nie gwarantowałoby znalezienia w skończonym czasie optymalnej kombinacji wartości parametrów. Nawet dla naszych 12 przykładów i dwóch zmiennych (roku i wyrazu wolnego) do wygenerowania 5000 modeli użyliśmy ograniczonych zakresów wartości obu parametrów.

Potrzebujemy lepszego algorytmu optymalizacji procesu uczenia od sprowadzającego się do losowania wartości parametrów i porównania błędów tak utworzonych modeli. W jego znalezieniu pomoże nam rysunek 3.11 przedstawiający kombinacje wartości obu parametrów naszych modeli, z zaznaczonymi dwudziestoma najlepszymi:

```
ggplot(models, aes(a1,a2)) +
 geom_point(data = filter(models,  rank(dist) <20),
              size=4, color = "red",  fill="red" ) +
 geom_point(aes(color =-dist), alpha = 1/4)
```



Rysunek 3.11. Kombinacje wartości parametrów najlepszych modeli nie są przypadkowe – zauważmy, że w tym przypadku im większy jest wyraz wolny, tym mniejsza jest optymalna wartość parametru b_1

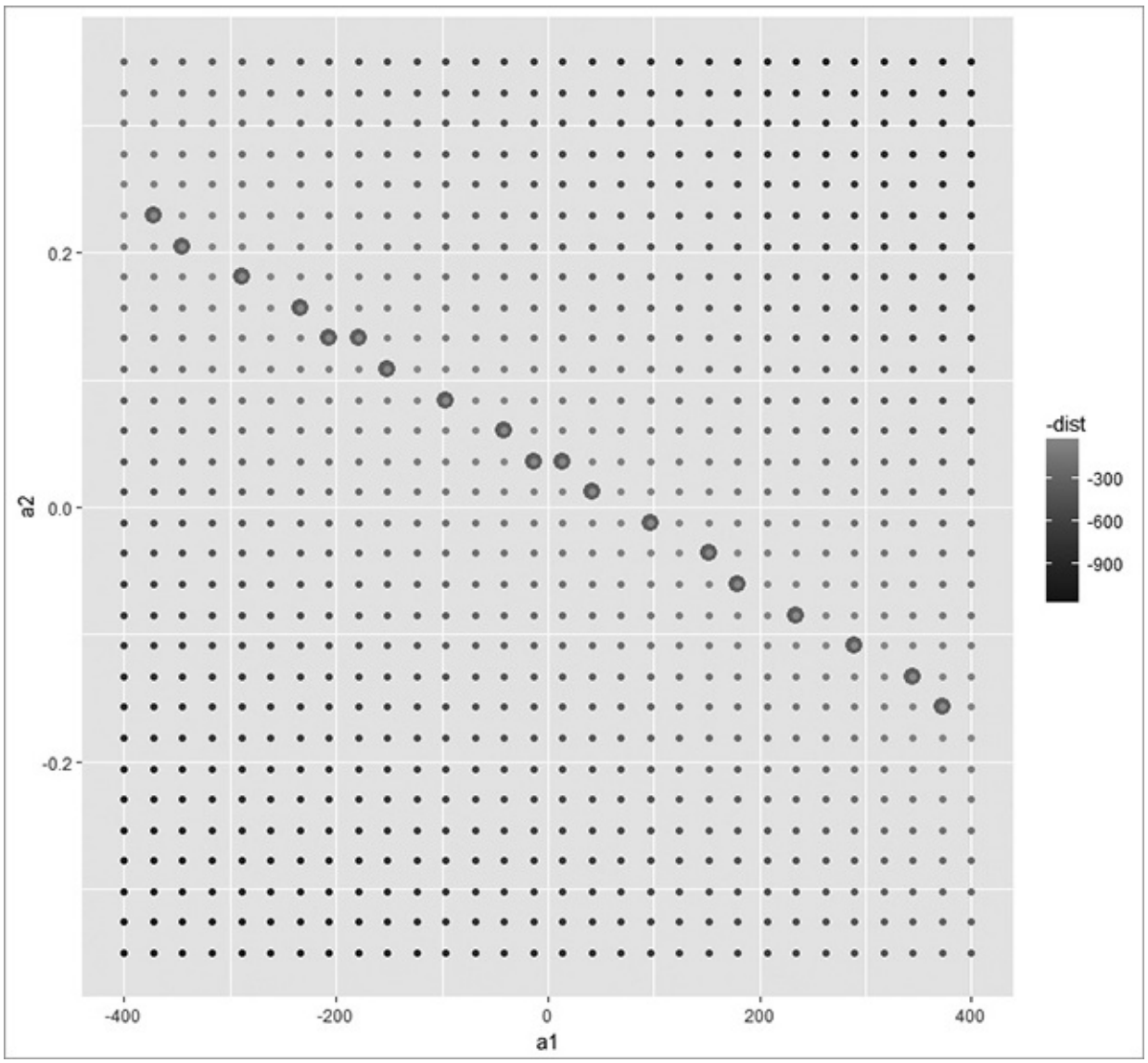
Zależności występujące między wartościami parametrów a kosztem modelu są używane do optymalizacji procesu uczenia. Jedną z najczęściej wykorzystywanych metod optymalizacji jest metoda najszybszego spadku

gradientu. Pozostawię tę kwestię otwartą, wspominając jedynie, że metoda ta polega na obliczaniu pochodnej funkcji kosztu względem przypadków treningowych i zwiększaniu lub zmniejszaniu wartości parametrów modelu w kierunku najszybszego spadku wartości tak obliczonej pochodnej. Teraz przedstawię dwie prostsze, ale w praktyce nieużywane, metody optymalizacji procesu uczenia.

Do znalezienia optymalnej wartości parametrów można użyć metody przeszukiwania siatki (ang. *grid search*). Metoda ta jest optymalizacją przeszukiwania losowych kombinacji wartości parametrów. Przeszukiwanie siatki polega na wygenerowaniu siatki równo rozłożonych wartości parametrów, a następnie obliczeniu kosztu poszczególnych modeli (modeli z różnymi wartościami parametrów) i wybraniu modelu o najmniejszym koszcie (rys. 3.12).

```
grid <- expand.grid(
   a1 = seq(-400, 400, length = 30),
   a2 = seq(-0.35, 0.35, length = 30)
) %>%
   mutate(dist = map2_dbl( a1, a2, pl_dist))

grid %>%
   ggplot( aes(a1, a2)) +
   geom_point(data = filter( grid, rank( dist) <20),
                  size = 4, colour = "red" ) +
   geom_point(aes( color = -dist))
```



Rysunek 3.12. Siatka wartości parametrów modeli z zaznaczonymi dziewiętnastoma najlepszymi kombinacjami

Najlepsze modele ze znalezionych metodą przeszukiwania siatki pokazano na rysunku 3.13.



Rysunek 3.13. Choć przeszukiwana siatka zawierała pięć razy mniej kombinacji wartości parametrów niż tabela z losowymi modelami, tą metodą udało się znaleźć lepsze modele

Optymalną kombinację wartości parametrów można też wyliczyć metodą stycznych (metodą Newtona–Raphsona). Polega ona na szukaniu zer funkcji liniowej stycznej do szukanej funkcji *h*, i została zaimplementowana w funkcji optim.

```
best <- optim(c(0,0), errors, data = pl)
best$par
```

```
-318.2268650    0.1962128
```

W praktyce do optymalizacji treningu modeli są używane specjalistyczne algorytmy uczenia maszynowego. Na przykład wchodząca w skład standardowego pakietu `stats` funkcja `lm` pozwala wytrenować model regresji liniowej na wskazanych danych. Parametr `data` pozwala wskazać zbiór treningowy, a parametr `formula` postawić hipotezę:

```
mod <- lm(formula = lifeExp ~ year, data = pl)
coef(summary(mod))
```

```
Estimate   Std. Error    t value      Pr(>|t|)
(Intercept) -318.2383585 53.67560428 -5.928920 1.452934e-04
year             0.1962189   0.02711471   7.236622 2.802333e-05
```

Użyjemy teraz naszego modelu do oszacowania średniej długości życia Polaków w latach 1952–2007. Po pierwsze, zapiszemy w tabeli te lata, które wystąpiły w zbiorze treningowym:

```
grid_pl <- pl %>%
  data_grid(year)
```

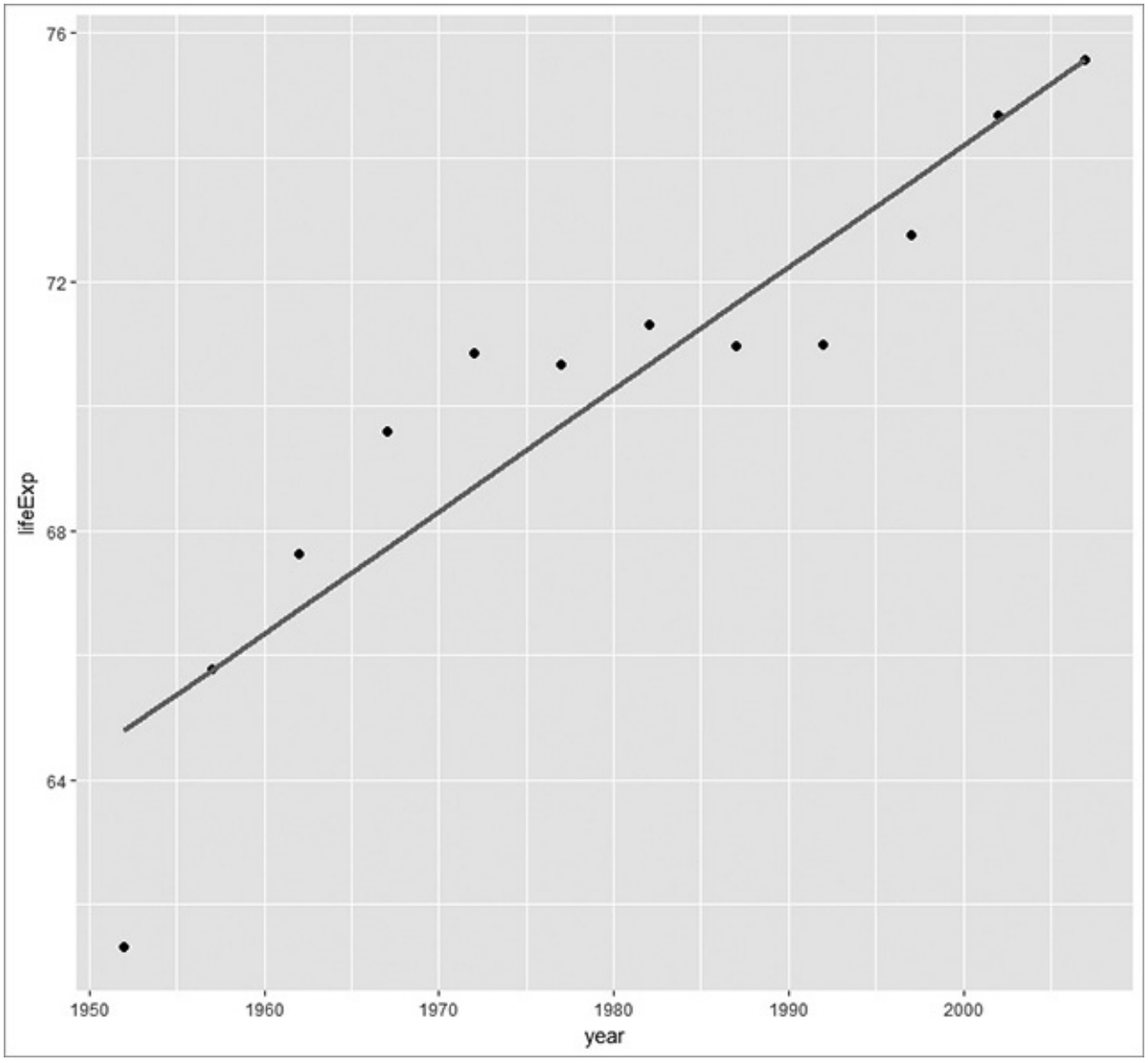
Następnie użyjemy funkcji `add_predictions` z pakietu `modelr` do wykonania zapytań predykcyjnych i zapisania ich wyników w nowej kolumnie tabeli pl:

```
grid_pl <- grid_pl %>%
  add_predictions(mod)
grid_pl
```

```
# A tibble: 12 × 2
     year      pred
    <int>     <dbl>
1    1952 64.78090
2    1957 65.76199
3    1962 66.74309
4    1967 67.72418
5    1972 68.70528
6    1977 69.68637
7    1982 70.66746
8    1987 71.64856
9    1992 72.62965
10   1997 73.61075
11   2002 74.59184
12   2007 75.57294
```

Przedstawmy na wykresie przypadki treningowe z nałożoną linią reprezentującą nasz model regresji (rys. 3.14):

```
ggplot(pl, aes(year, lifeExp)) +
  geom_point(size = 2) +
  geom_line(aes(y=pred), data = grid_pl, color = "red", size = 1.5)
```



Rysunek 3.14. Finalny model regresji liniowej zestawiony z przypadkami, na których został wytrenowany

Na koniec, wyliczymy reszty (różnice między rzeczywistymi a zwróconymi przez model wartościami zmiennej docelowej) i zapiszemy je w tabeli `p1`:

```
pl <- pl %>%
  add_residuals(mod)
pl
```

```
A tibble: 12 × 7
    country continent   year lifeExp       pop gdpPercap          resid
     <fctr>     <fctr> <int>   <dbl>     <int>      <dbl>          <dbl>
1    Poland     Europe  1952  61.310 25730551   4029.330   -3.470897436
2    Poland     Europe  1957  65.770 28235346   4734.253    0.008008159
3    Poland     Europe  1962  67.640 30329617   5338.752    0.896913753
4    Poland     Europe  1967  69.610 31785378   6557.153    1.885819347
5    Poland     Europe  1972  70.850 33039545   8006.507    2.144724942
6    Poland     Europe  1977  70.670 34621254   9508.141    0.983630536
7    Poland     Europe  1982  71.320 36227381   8451.531   0.652536131
8    Poland     Europe  1987  70.980 37740710   9082.351 -0.668558275
9    Poland     Europe  1992  70.990 38370697   7738.881 -1.639652681
10   Poland     Europe  1997  72.750 38654957 10159.584 -0.860747086
11   Poland     Europe  2002  74.670 38625976 12002.239  0.078158508
12   Poland     Europe  2007  75.563 38518241 15389.925 -0.009935897
```

Jak na tak prosty model, uzyskane wyniki są zaskakująco dobre – tylko cztery razy model pomylił się więcej niż o rok.

W praktyce wartości zmiennej wyjściowej są obliczane na podstawie wielu zmiennych wejściowych. Takim modelom regresji przyjrzymy się w kolejnych rozdziałach, w tym miejscu ograniczając się do uwagi, że uwzględnienie dodatkowych zmiennych wejściowych polega na zastąpieniu poznanej hipotezy następującą:

$$y = f(x) = b_0 + b_1x_1 + b_2x_2 + ... + b_mx_m$$

gdzie y jest zmienną wyjściową, x_m to kolejne zmienne wejściowe, a b_m to parametry modelu dobrane automatycznie, aby zminimalizować funkcję kosztu.

Aby na przykład wytrenować model zwracający długość życia na podstawie roku, populacji i dochodu na mieszkańca, należy wykonać poniższą instrukcję:

```
mod2 <- lm(lifeExp ~ year + pop + gdpPercap, data = pl)
coef(summary(mod2))
```

```
                    Estimate    Std. Error    t value    Pr(>|t|)
(Intercept)   3.224661e+02 2.177318e+02   1.481024 0.17687495
year          -1.457643e-01 1.162082e-01 -1.254338 0.24512656
pop            8.436052e-07 3.302453e-07   2.554481 0.03393719
gdpPercap     8.668567e-04 3.295598e-04   2.630347 0.03016192
```

Porównanie obu modeli, tego z jedną zmienną wejściową i tego z trzema zmiennymi wejściowymi, pozostawiam Czytelnikom.

3.4. Modele partycjonujące

Działanie modeli regresji polegało na znalezieniu funkcji zmiennej wyjściowej od zmiennych wejściowych. Modele partycjonujące, do których zalicza się drzewa decyzyjne i ich modyfikacje, działają inaczej:

- na początku wszystkie przypadki treningowe trafiają do jednego zbioru;
- oceniany jest wpływ poszczególnych zmiennych wejściowych na podział tego zbioru względem zmiennej wyjściowej;
- zbiór jest dzielony na podzbiory, aby każdy z tych podzbiorów zawierał jak najwięcej takich samych przypadków, czyli przypadków o takich samych wartościach zmiennej docelowej;
- dla każdego podzbioru następuje ponowna ocena zmiennych i ich dalszy podział.

Mimo swojej prostoty, modele partycjonujące mają kilka ważnych zalet w porównaniu do modeli regresji:

- są łatwe w interpretacji;
- nie wymagają przyjęcia jakichkolwiek założeń dotyczących rozkładu danych

i zależności między zmiennymi;

- są niewrażliwe na wartości odstające.

 Uczenie modelu partycjonującego polega na znalezieniu kryteriów podziału, które z wystarczająco dobrym przybliżeniem reprezentują prawdziwe zależności między zmiennymi wejściowymi a wyjściową. Parametry modelu partycjonującego są dyskretne, ale założenie jest takie samo jak w modelach regresji – należy znaleźć hipotezę będącą wystarczająco dobrym przybliżeniem prawdziwych zależności.

Modele partycjonujące mogą być używane do klasyfikacji i regresji. Przypomnijmy, dane treningowe tworzy zbiór przykładów, z których każdy jest opisany za pomocą listy atrybutów (zmiennych wejściowych, objaśniających, zmiennych *x*) i etykiety (zmiennej wyjściowej, objaśnianej, docelowej, zmiennej *y*). Jeżeli zmienna wyjściowa jest kategoryczna, mówimy o klasyfikacji, jeżeli ciągła – o regresji.

Zmienne te reprezentują złożone obiekty, tworząc model. **Model to uproszczona wersja rzeczywistości, zawierająca tylko te aspekty rzeczywistego świata, które są istotne dla postawionego zadania**. Powinniśmy pamiętać, że modele nie zawierają kompletu danych, dlatego ich zastosowanie jest ograniczone. Dane źródłowe są zbierane w ramach modeli, a ich analiza według nieprzyjętych dla tych modeli reguł, choć technicznie możliwa, prowadzi do niemających praktycznego zastosowania wyników.

Przypuśćmy, że naszym zadaniem jest ocena grzyba jako jadalnego lub trującego. Różnorodność gatunków grzybów oraz egzemplarzy tych samych gatunków jest ogromna (rys. 3.15). Jeżeli spróbujemy samodzielnie opisać grzyby, czyli wybrać zmienne, nie posiadając wiedzy na temat grzybów i tego, czy są jadalne, czy trujące, skończymy z tysiącami nieprzydatnych zmiennych. Z tego powodu **zbudowanie dobrego modelu uczenia maszynowego bez wsparcia eksperta z danej dziedziny jest trudne**. Specjalista z danej dziedziny potrzebny nam będzie na wielu etapach projektu, zaczynając od sprecyzowania problemu i oceny jakości danych.



Rysunek 3.15. Przykładowe grzyby jadalne. Nawet na takim uproszczonym rysunku ich rozmaitość jest trudna do opisania. Źródło: https://commons.wikimedia.org/wiki/File%3AEdible_Fungi.jpg

Zobaczmy, jak wygląda przygotowany przez takiego eksperta, dostępny pod adresem http://archive.ics.uci.edu/ml/machine-learning-databases/mushroom/agaricus-lepiota.dat, zbiór danych o grzybach:

```
str(mushrooms)
```

```
,data.frame':    8124 obs. of   23 variables:
 $ class                      : Factor w/ 2 levels "edible","poisonous": 2 1 1 2 1 1 1
1 2 1 ...
 $ cap_shape                  : Factor w/ 6 levels "bell","conical",..: 6 6 1 6 6 6 1 1
6 1 ...
 $ cap_surface                : Factor w/ 4 levels "fibrous","grooves",..: 3 3 3 4 3 4 3
4 4 3 ...
 $ cap_color                  : Factor w/ 10 levels "buff","cinnamon",..: 5 10 9 9 4 10
9 9 9 10 ...
 $ bruises                    : Factor w/ 2 levels "no","yes": 2 2 2 2 1 2 2 2 2 2 ...
 $ odor                       : Factor w/ 9 levels "almond","creosote",..: 7 1 4 7 6 1
1 4 7 1 ...
 $ gill_attachment            : Factor w/ 2 levels "attached","free": 2 2 2 2 2 2 2 2 2 2
...
 $ gill_spacing               : Factor w/ 2 levels "close","crowded": 1 1 1 1 2 1 1 1 1
1 ...
 $ gill_size                  : Factor w/ 2 levels "broad","narrow": 2 1 1 2 1 1 1 1 2
1 ...
 $ gill_color                 : Factor w/ 12 levels "buff","red","gray",..: 5 5 6 6 5 6
3 6 8 3 ...
 $ stalk_shape                : Factor w/ 2 levels "enlarging","tapering": 1 1 1 1 2 1 1
1 1 1 ...
 $ stalk_root                 : Factor w/ 5 levels "missing","bulbous",..: 4 3 3 4 4 3 3
3 4 3 ...
 $ stalk_surface_above_ring: Factor w/ 4 levels "fibrous","silky",..: 3 3 3 3 3 3 3 3 3
3 ...
 $ stalk_surface_below_ring: Factor w/ 4 levels "fibrous","silky",..: 3 3 3 3 3 3 3 3 3
3 ...
 $ stalk_color_above_ring   : Factor w/ 10 levels "buff","cinnamon",..: 8 8 8 8 8 8 8 8 8
8 ...
 $ stalk_color_below_ring   : Factor w/ 10 levels "buff","cinnamon",..: 8 8 8 8 8 8 8 8 8
8 ...
 $ veil_type                  : Factor w/ 1 level "partial": 1 1 1 1 1 1 1 1 1 1 ...
 $ veil_color                 : Factor w/ 4 levels "brown","orange",..: 3 3 3 3 3 3 3 3
3 3 ...
 $ ring_number                : Factor w/ 3 levels "none","one","two": 2 2 2 2 2 2 2 2 2
2 ...
 $ ring_type                  : Factor w/ 5 levels "evanescent","flaring",..: 5 5 5 5 1
5 5 5 5 5 ...
 $ spore_print_color         : Factor w/ 9 levels "buff","chocolate",..: 3 4 4 3 4 3 3 4
3 3 ...
 $ population                 : Factor w/ 6 levels "abundant","clustered",..: 4 3 3 4 1
3 3 4 5 4 ...
 $ habitat                    : Factor w/ 7 levels "wood","grasses",..: 6 2 4 6 2 2 4 4
2 4 ...
```

Dysponujemy 8124 przykładami, każdy gatunek grzyba opisują 23 zmienne. Zmienna `class` jest zmienną wyjściową, przyjmującą dwa stany: `edible` i `poisonous`. Pozostałe zmienne są zmiennymi wejściowymi. Wszystkie zmienne są kategoryczne. Czytelników zainteresowanych sposobem, w jaki zmienne te zostały nazwane oraz jak ich wartości zostały skonwertowane do czynników, odsyłam do pliku *Mushrooms classification.R*. Plik ten można znaleźć w repozytorium z przykładami do książki.

W tym przypadku zmienną docelową jest zmienna `class`. Jest to zmienna kategoryczna, a więc naszym zadaniem jest zbudowanie klasyfikatora. Klasyfikacja jest metodą uczenia nadzorowanego, czyli zbiór danych treningowych musi zawierać etykiety. Celem klasyfikacji jest znalezienie funkcji klasyfikującej *h*, która odwzoruje wartości zmiennych wejściowych *X* na wartość zmiennej docelowej *Y*. Najprostszym przykładem klasyfikacji jest **klasyfikacja binarna**. Odpowiada ona na pytania typu tak lub nie bądź prawda lub fałsz, czyli przypisuje przypadki do jednej z dwóch klas, dokładnie tak jak w tym przykładzie.

Zasada działania modeli partycjonujących polega na podziale przypadków na podzbiory na podstawie tego, w jaki sposób ten podział wpłynie na stany zmiennej wyjściowej. Dlatego w pierwszej kolejności należy sprawdzić rozkład zmiennej objaśnianej:

```
class <- plyr::count(mushrooms$class)
print(sprintf("Edible: %d | Poisonous: %d | Percent of poisonous classes: %.1f%%",
class$freq[1],class$freq[2], round(class$freq[1]/nrow(mushrooms)*100,1)))
```

```
"Edible: 4208 | Poisonous: 3916 | Percent of poisonous classes:
51.8%"
```

Oba stany zmiennej wyjściowej występują z podobną częstością. Nasz zbiór danych jest zrównoważony, co pozwala nam przejść do oceny zależności między zmiennymi wejściowymi a wyjściową.

```
mush_features <- colnames(mushrooms)[-1]
table_res <- lapply(mush_features, function(x) {table(mushrooms$class, mushrooms[,x])})
names(table_res) <- mush_features
```

Lista `table_res` zawiera tabele rozkładu częstości (tabele częstości) zmiennych wejściowych względem zmiennej docelowej. Analizując je, zauważymy, że na przykład rodzaj grzyba (to, czy jest on jadalny, czy trujący) jest silnie powiązany z typem obwódki kapelusza:

```
table_res$ring_type
```

	evanescent	flaring	large	none	pendant
edible	1008	48	0	0	3152
poisonous	1768	0	1296	36	816

Grzybów z kapeluszami o dużej obwódce (`large`) oraz z kapeluszami bez obwódki (`none`) nie powinniśmy jeść – w przykładowych danych nie znajdziemy ani jednego jadalnego grzyba o takim typie kapelusza.

Równie ważny okazuje się być zapach – grzyby pachnące anyżkowo lub migdałowo oraz te bez zapachu możemy jeść bezpiecznie:

```
table_res$odor
```

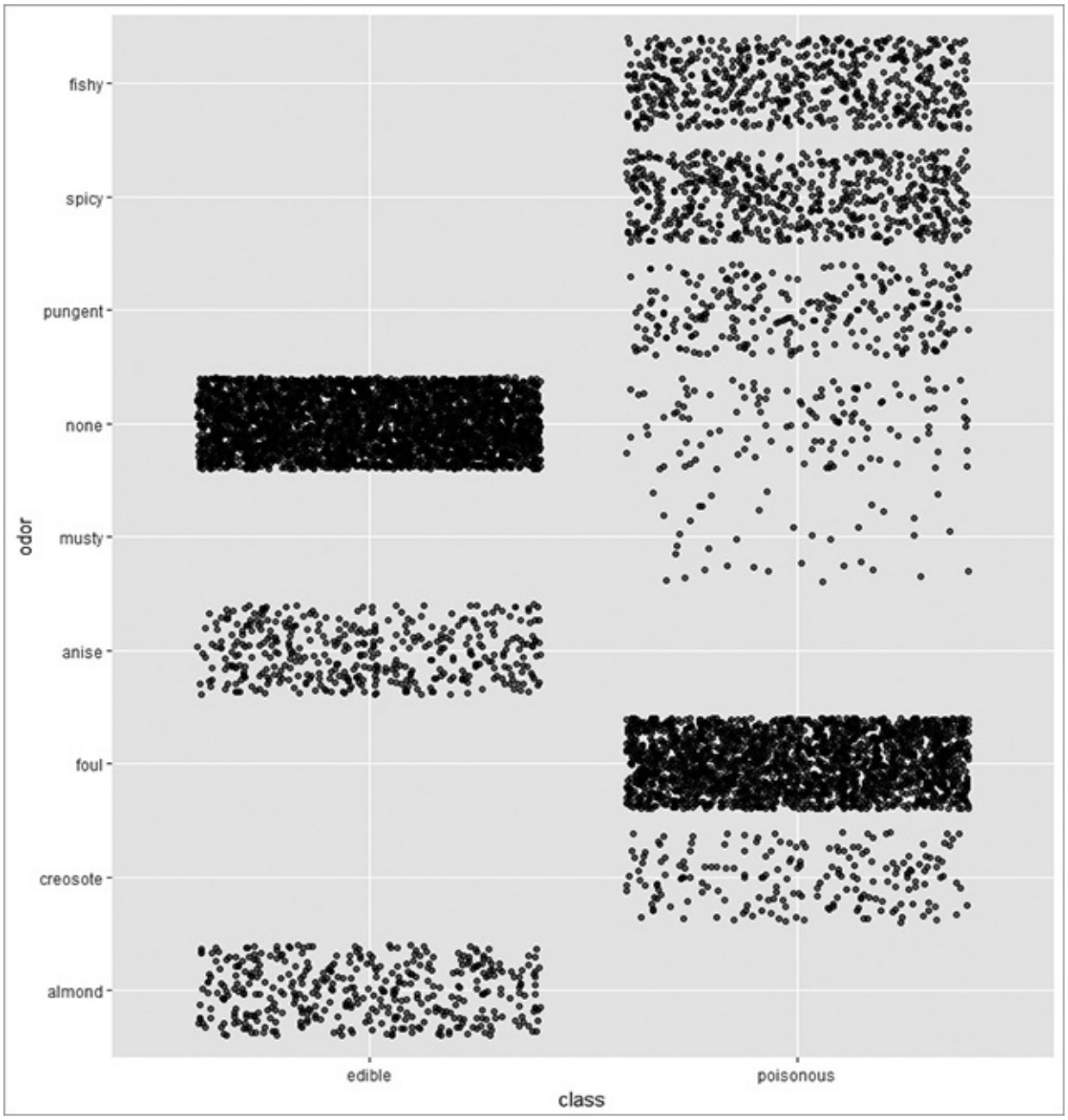
	almond	creosote	foul	anise	musty	none	pungent	spicy	fishy
edible	400	0	0	400	0	3408	0	0	0
poisonous	0	192	2160	0	36	120	256	576	576

Względny rozkład częstości zmiennych często przedstawia się na wykresach (rys. 3.16):

```
ggplot(mushrooms, aes(x = class, y = odor)) +
  geom_jitter(alpha = 0.5)
```

Analiza względnego rozkładu częstości zmiennych doprowadziła nas do odkrycia silnych zależności, takich jak: „Na 2160 zbadanych, śmierdzących

grzybów, 2160 okazało się trujących, a 0 jadalnych”. Takie zależności możemy przekształcić na reguły pozwalające oceniać jadalność grzybów. Jeżeli na przykład grzyb śmierdzi (zmienna `odor` ma wartość `foul`), to grzyb jest trujący (zmienna `class` ma wartość `poisonous`).

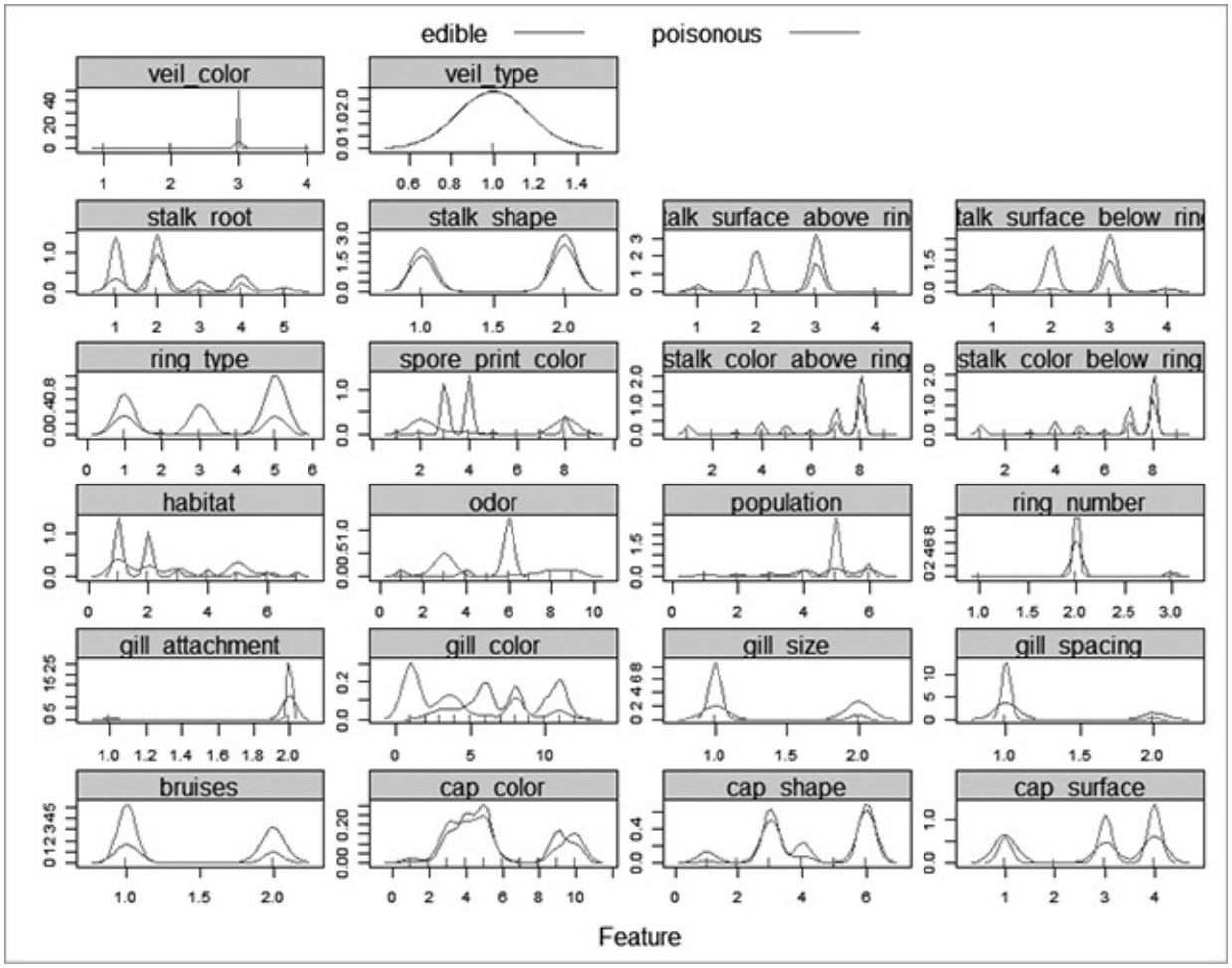


Rysunek 3.16. Gdyby oceniać jadalność grzybów tylko na podstawie ich zapachu, wystarczyłoby unikać grzybów o zapachu ryby, tych o ostrych zapachach, grzybów śmierdzących i tych o zapachu smoły

Nie wszystkie odkryte reguły tak czysto dzielą przypadki. Na przykład z 3528 grzybów bez zapachu, 3408 (ponad 96%) było jadalnych, a 120 (niecałe 4%) – trujących. W przedstawieniu tego typu zależności pomocne będzie zastąpienie poszczególnych stanów zmiennych kategorycznych liczbami. Pozwoli to wyświetlić rozkład wartości zmiennych nie na histogramach, ale za pomocą krzywej częstości. Zadanie to realizuje funkcja `featurePlot` z biblioteki `caret` (rys. 3.17):

```
label = mushrooms[,1]
features = mushrooms[,2:23]
features <- sapply(features, function (x) as.numeric(as.factor(x)))

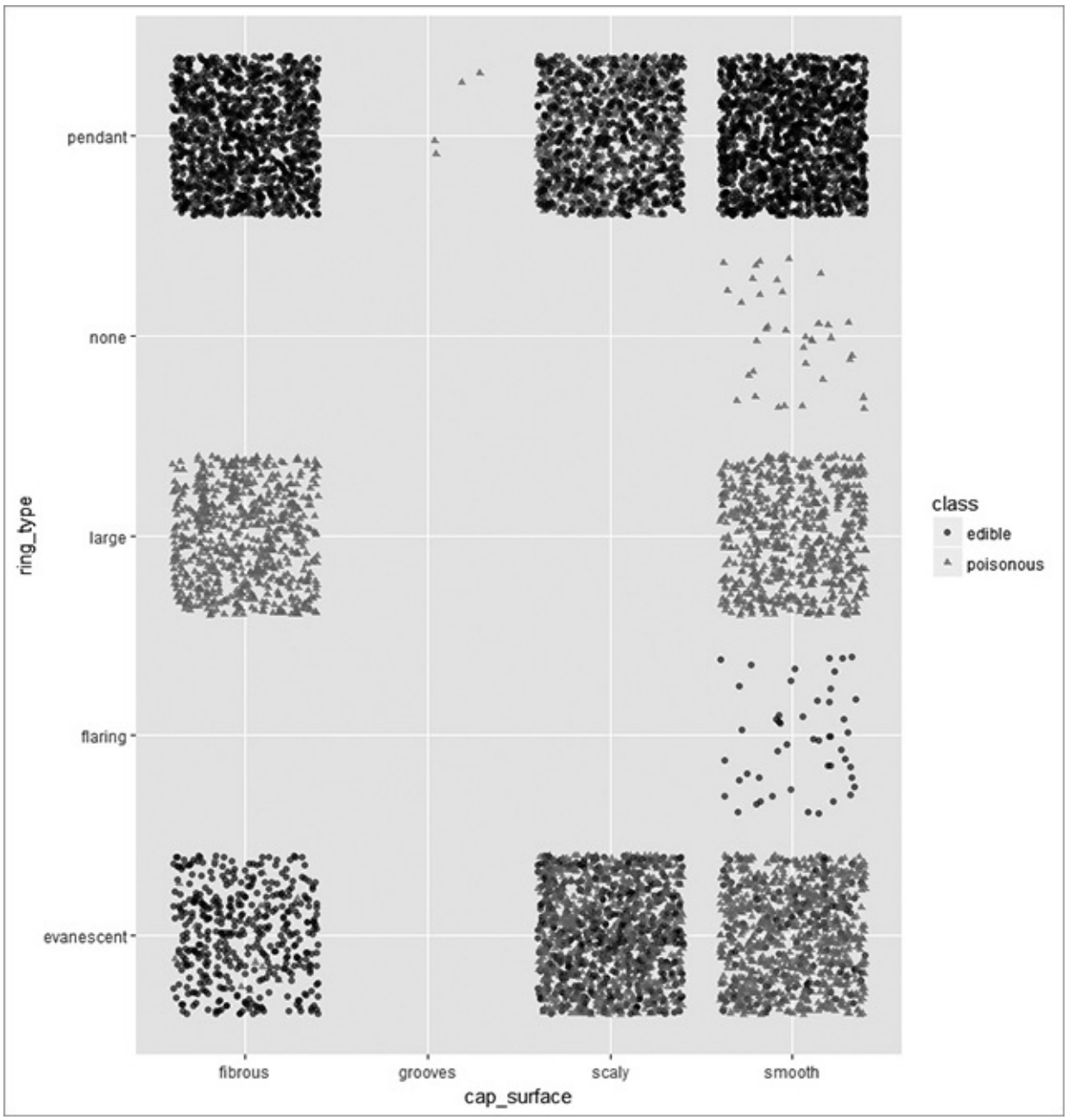
scales <- list(x=list(relation="free"),y=list(relation="free"),
cex=0.6)
featurePlot(x=features, y=label, plot="density",scales=scales,
                        layout = c(4,6), auto.key = list(columns = 2), pch = "|")
```



Rysunek 3.17. Im większe różnice między liniami reprezentującymi jadalne i trujące grzyby, tym czystszy podział przypadków według odpowiednich wartości zmiennej objaśniającej

Zaskakującą skuteczność metody określania wartości zmiennej wyjściowej przez podział przypadków według zmiennych wejściowych wynika z tego, że brane są pod uwagę wszystkie zmienne wejściowe. Na rysunku 3.18 pokazano efekt podziału względem dwóch zmiennych.

```
ggplot(mushrooms, aes(x = cap_surface, y = ring_type, col = class,
shape = class)) +
  geom_jitter(alpha = 0.5) +
  scale_color_manual(breaks = c("edible", "poisonous"),
                     values = c("black", "red"))
```


pendant
none
large
flaring
evanescent
ring_type
fibrous
grooves
scaly
smooth
cap_surface
class
edible
poisonous

Rysunek 3.18. Jeżeli grzyb ma włóknistą powierzchnię czubka, zanim go zjemy powinniśmy sprawdzić typ obwódki – grzyby o włóknistej powierzchni czubka i dużej obwódce są trujące, grzyby o obwisłej lub ledwo widocznej obwódce są w większości jadalne

Do sprawdzenia zależności między cechami grzybów a ich jadalnością możemy użyć testu zgodności chi-kwadrat. Polega on na porównaniu wartości zmierzonej (w tym przypadku częstości występowania jadalnych i trujących grzybów o danej cesze) z wartością oczekiwaną *e*. Następnie testem Pearsona sprawdzimy, czy zależność między komórkami tabeli częstości jest statystycznie istotna. Hipoteza zerowa (H_0) zakłada ich niezależność, hipoteza alternatywna (H_1) zakłada, że komórki tabeli częstości zależą od siebie.

Dla każdej komórki należy obliczyć jej spodziewaną wartość (wartość, która wystąpiłaby w tej komórce, gdyby hipoteza zerowa była prawdziwa). Użyjemy do tego wzoru

$$e = \frac{\textit{suma wierszy} * \textit{suma kolumn}}{\textit{suma całkowita}}$$

Znając wartość oczekiwaną, możemy obliczyć statystykę chi-kwadrat

$$\chi^2 = \Sigma \frac{(O - E)^2}{E}$$

Tak otrzymany wynik należy porównać z odczytanymi z tabel statystycznych wartościami krytycznymi dla stopnia swobody d*f* obliczonego jako

d*f* = (*liczba wierszy tabeli częstości* –1) * (*liczba kolumn tabeli częstości* – 1)

Otrzymamy w ten sposób wartość *p* testu. Często przyjmuje się poziom istotności statystycznej $p < 0{,}05$, czyli jeżeli wartość *p* jest mniejsza, to istnieje statystycznie istotna zależność między komórkami tabeli częstości. Chcąc oszczędzić trochę papieru, poniżej ograniczam pokazane statystyki do zależności między jedną zmienną wejściową, `odor`, a zmienną wyjściową `class`. Pokazuję jednak zaobserwowane i teoretyczne wartości rozkładu zmiennej `odor` względem

zmiennej wyjściowej.

```
chisq_test_res = list()
relevant_features = c()

for (i in 2:length(colnames(mushrooms))) {
  if (nlevels(mushrooms[,i]) > 1) {
    fname = colnames(mushrooms)[i]
    res = chisq.test(mushrooms[,i], mushrooms[,"class"], simulate.p.value = TRUE)
    res$data.name = paste(fname, "class", sep= " and ")
    chisq_test_res[[fname]] = res
    relevant_features = c(relevant_features, fname)
  }
}
chisq_test_res$odor
chisq_test_res$odor$observed
chisq_test_res$odor$expected
```

```
Pearson's Chi-squared test with simulated p-value (based on 2000 replicates)

data:  odor and class
X-squared = 7659.7, df = NA, p-value = 0.0004998
mushrooms[, "class"]
mushrooms[, i] edible poisonous
       almond       400         0
       creosote       0       192
       foul           0      2160
       anise        400         0
       musty          0        36
       none        3408       120
       pungent        0       256
       spicy          0       576
       fishy          0       576

mushrooms[, "class"]
mushrooms[, i]      edible   poisonous
       almond     207.18858  192.81142
       creosote    99.45052   92.54948
       foul      1118.81832 1041.18168
       anise      207.18858  192.81142
       musty       18.64697   17.35303
       none      1827.40325 1700.59675
       pungent    132.60069  123.39931
       spicy      298.35155  277.64845
       fishy      298.35155  277.64845
```

Oszacowaliśmy zależność między zmiennymi dla danego zbioru, co pozwoliłoby nam wybrać zmienną, według której należy ten zbiór podzielić. Taka jest idea opracowanego w latach 80. XX w. algorytmu klasyfikacji przez indukcję drzew decyzyjnych[2]. Działanie tego klasycznego algorytmu uczenia maszynowego polega na powtarzaniu poniższych kroków:

- jeżeli zmienna wejściowa jest numeryczna, zostaje ona poddana dyskretyzacji;
- ze zmiennych wejściowych wybierana jest ta, według której podział zminimalizuje ilość informacji potrzebnej do zaklasyfikowania przypadków względem zmiennej wyjściowej do otrzymanych w ten sposób podzbiorów;
- podzbiory są tworzone dla wszystkich stanów wybranej zmiennej lub metodą binarną, tak aby każdy podzbiór był maksymalnie jednorodny względem zmiennej wyjściowej i jednocześnie maksymalnie odrębny względem zmiennej wyjściowej od pozostałych podzbiorów;
- budowane są poddrzewa, których danymi treningowymi są otrzymane w poprzednim kroku podzbiory; jeżeli stosowana jest metoda wstępnego przycinania, poddrzewa, których podzbiory liczą zbyt mało przypadków lub dla których rozkład klas jest za mało wyraźny, są usuwane.

Powyższe kroki są powtarzane dla każdego poddrzewa, aż do wystąpienia jednej z sytuacji:

- wyczerpania przypadków w zbiorze danych treningowych,
- przypisania wszystkich przypadków do jednej z klas,
- przekroczenia zadanego czasu lub zadanej, maksymalnej wielkości drzew.

Budowanie drzewa decyzyjnego zaczyna się od znalezienia zmiennej, której podział wyznaczy korzeń drzewa. Do jej znalezienia można użyć różnych miar, na przykład entropii krzyżowej (ang. *cross-entropy*), prawdopodobieństwa

Bayesa, zysku informacyjnego lub współczynnika Giniego. Wszystkie one wywodzą się z pojęcia entropii.

Entropia jest miarą informacji. Przyjmuje ona maksimum (dokładniej o znaczeniu i jakie wartości przyjmuje entropia piszę w podrozdz. 4.4), gdy prawdopodobieństwa wszystkich możliwych stanów są takie same, a minimum (0), gdy prawdopodobieństwo jednego ze stanów wynosi 100%. Pokażmy to na przykładzie rzucania monetą. Pierwsza moneta będzie uczciwa, czyli przed rzutem prawdopodobieństwo, że wypadnie orzeł, jest takie samo, jak że wypadnie reszka. Przypuśćmy, że w serii 10 rzutów 6 razy wypadł orzeł, a 4 razy reszka. Entropia tej zmiennej jest duża, bliska maksymalnej (maksimum osiągnęłaby, gdyby orzeł i reszka wypadły tyle samo razy, czyli gdyby oba stany zmiennej były tak samo prawdopodobne):

```
library(entropy)
flips <- c("Head", "Head", "Tail", "Tail", "Head", "Tail", "Head",
"Head", "Head", "Tail")
freqs <- table(flips)/length(flips)
entropy(freqs, unit="log2")
 0.9709506
```

Za drugim razem użyjemy fałszywej monety. Ponieważ ta moneta zawsze upada orłem do góry, entropia zmiennej jest minimalna (wynosi 0). W takiej sytuacji nie musimy czekać, aż moneta spadnie, żeby wiedzieć, że wypadł orzeł:

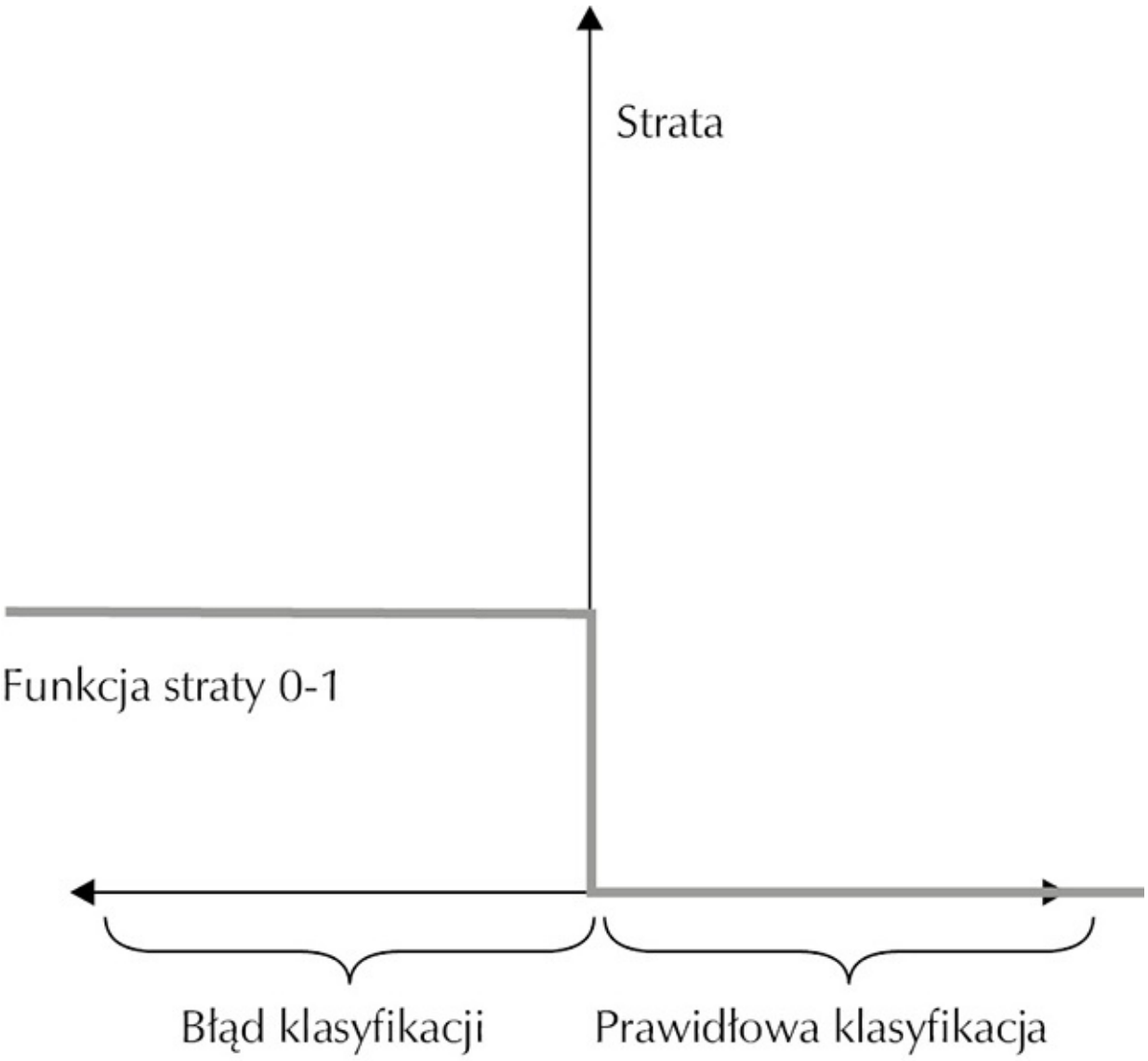
```
flips <- rep("Head", 10)
freqs <- table(flips)/length(flips)
entropy(freqs, unit="log2")
0
```

W ten sam sposób możemy policzyć entropię poszczególnych zmiennych zbioru treningowego. Funkcja `entropy` z biblioteki o tej samej nazwie pozwala policzyć entropię zmiennej:

```
entropy(table_res$odor, unit="log2")
entropy(table_res$ veil_type,unit="log2")
2.412407
0.9990679
```

Na tej podstawie możemy powiedzieć, że podział przypadków według zmiennej `odor` jest korzystniejszy niż według zmiennej `veil_type`.

Te krótkie rozważania o entropii pozwalają nam wrócić do omawiania funkcji kosztu klasyfikatorów. Ocena jakości modeli regresji (modeli szacujących wartość ciągłej zmiennej objaśnianej) była łatwa – wystarczyło porównać prawdziwe wartości zmiennej docelowej z wynikami predykcji. Ponieważ były to liczby, mogliśmy je odjąć, obliczyć wartość bezwzględną czy podnieść tak otrzymany wynik do kwadratu. Dlaczego by nie użyć równie prostej funkcji straty do oceny klasyfikatorów? Przecież można przyjąć, że każdy błąd klasyfikacji (w naszym przypadku ocena jadalnego grzyba jako trującego, a trującego jako jadalnego) będzie nasz model kosztowała 1, a każda prawidłowa klasyfikacja nie będzie modelu nic kosztowała (rys. 3.19).



Rysunek 3.19. Zero-jedynkowa funkcja kosztu użyta do oceny klasyfikacji pojedynczego przykładu

Wspomniałem, że najczęściej wybieranymi funkcjami kosztu klasyfikatorów są entropia krzyżowa, zysk informacyjny albo współczynnik Giniego. Przypomnijmy, entropia osiąga maksimum, gdy prawdopodobieństwo wystąpienia obu stanów zmiennej binarnej jest takie samo, a minimum – gdy prawdopodobieństwo jednego stanu wynosi 0 lub 1. Dokładnie takie same własności ma używany w statystyce indeks Giniego.

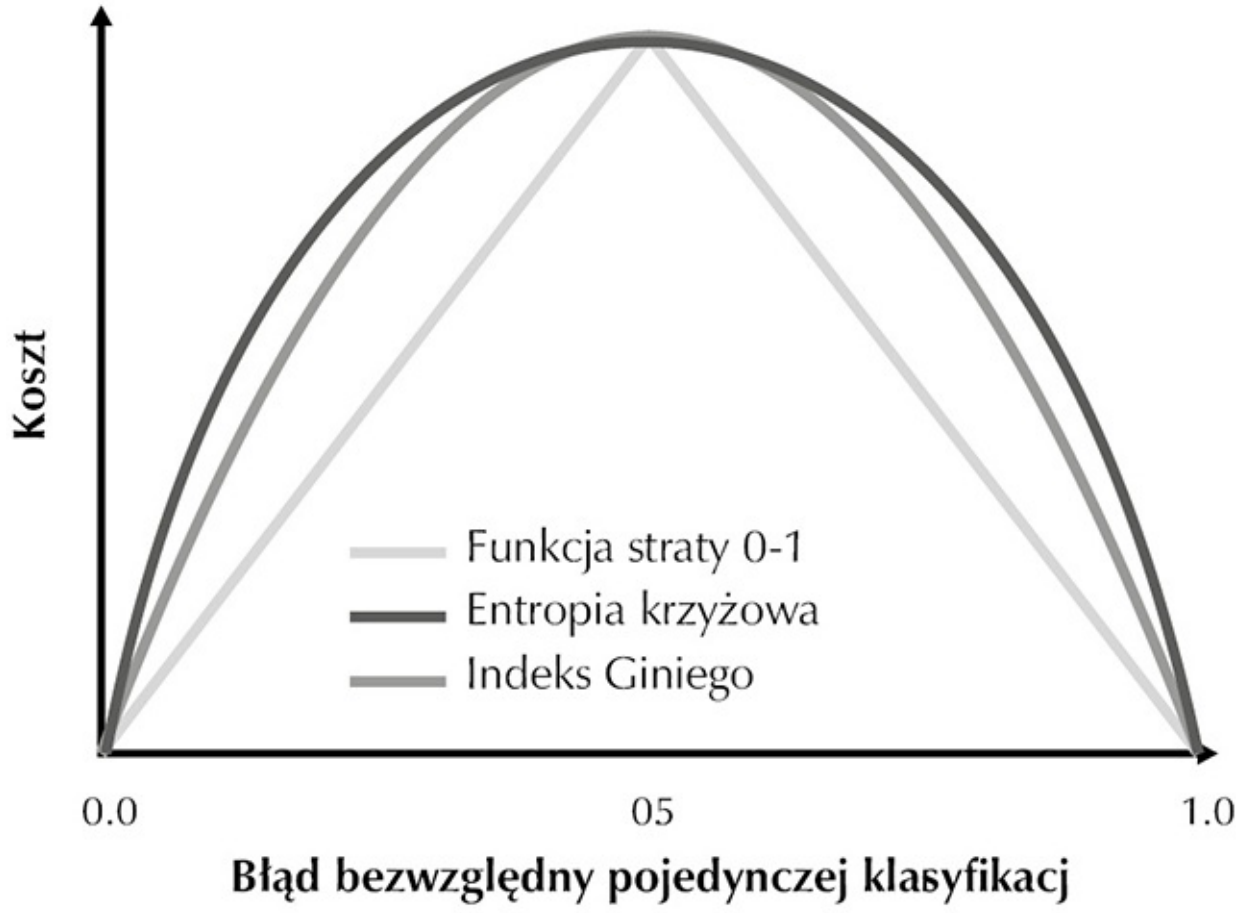
Weźmy teraz pod uwagę dwie zmienne. Ich zysk informacyjny to ilość informacji, jaką zdobędziemy o stanie zmiennej *B*, jeżeli poznamy stan zmiennej

A (zysk informacyjny jest symetryczny, więc zmiennych *A* i *B* można użyć zamiennie). Innymi słowy, zysk informacyjny jest ilością entropii, jaka została usunięta z jednej zmiennej dzięki poznaniu wartości drugiej:

$$I(A; B) = H(B) - \sum_b P(B = b) * H(A|B = b)$$

Entropia krzyżowa również pozwala zmierzyć redukcję entropii jednej zmiennej dzięki poznaniu wartości drugiej zmiennej, tyle że wyraża ją w liczbie bitów potrzebnych do zakodowania zmiennych o nieznanych rozkładach.

Zauważmy, że funkcja kosztu jest używana do oceny klasyfikacji wszystkich przykładów treningowych. Wyniki oceny poszczególnych przypadków się uśrednia. Uśredniając wyniki funkcji straty 0-1, otrzymamy funkcję, której wykres wygląda jak ramiona trójkąta – jej maksimum (największy błąd) odpowiada najmniejszemu uporządkowaniu i zmniejsza się linowo wraz ze zwiększaniem się prawdopodobieństw stanów 0 i 1. Natomiast pozostałe funkcje kosztu są funkcjami wklęsłymi, czyli łuk ich wykresów znajduje się powyżej cięciwy wyznaczonej wykresem funkcji straty 0-1 (rys. 3.20).



Rysunek 3.20. Funkcja straty 0-1, indeks Giniego i entropia krzyżowa klasyfikatora binarnego

Metody optymalizacji procesu uczenia zostały wyjaśnione w dalszych rozdziałach. Pozostawiam tę kwestę otwartą, na razie ograniczając się do stwierdzenia, że optymalizacja procesu uczenia daje najlepsze wyniki dla ciągłych, różniczkowalnych i wypukłych (funkcje wypukłe w górę nazywa się wklęsłymi) funkcji kosztu.

Wróćmy do metody konstruowania drzew decyzyjnych. Podział danych może być zupełny (ze względu na wszystkie wartości wybranej zmiennej) lub binarny, na dwa podzbiory. Powyższe metody podziału nie mają zastosowania dla zmiennych numerycznych. Dlatego ich wartości są dynamicznie dzielone na równe przedziały, oddzielnie dla każdego węzła drzewa decyzyjnego. Następnie sąsiednie przedziały są łączone, o ile takie połączenie daje możliwość lepszego

podzielenia przypadków. Ten proces jest powtarzany aż do osiągnięcia przedziałów najlepiej nadających się do podzielenia przypadków w danym węźle drzewa decyzyjnego.

Problematyczne dla drzew decyzyjnych są też zmienne kategoryczne, które przyjmują wiele stanów. Algorytmy, w których do podziału są wykorzystywane entropia lub zysk informacyjny, mają tendencję do faworyzowania takich zmiennych. W rezultacie uzyskuje się dużą liczbę małych podzbiorów (liczących niewiele przypadków), co prowadzi do przeuczenia modelu. W takich sytuacjach należy albo uogólnić zmienne o dużej liczbie stanów, albo do oceny podziałów użyć współczynnika Giniego.

Aby zmniejszyć ryzyko nadmiernego dopasowania drzew decyzyjnych do danych treningowych, buduje się je w dwóch etapach: podczas etapu konstrukcji (indukcji) drzewo decyzyjne jest tworzone na podstawie danych treningowych, a na etapie przycinania niektóre węzły drzewa są z niego usuwane.

Przycinanie drzewa decyzyjnego może odbywać się podczas jego konstrukcji – mówimy wtedy o **metodzie przycinania wstępnego** (ang. *prepruning*). Metoda ta jest lepiej skalowalna, szybsza i łatwiejsza do automatyzacji, ale może prowadzić do zbudowania nieoptymalnych drzew decyzyjnych. Przedwczesne przerwanie konstruowania drzew ma miejsce wtedy, gdy na danym etapie zysk z podziału jest mały (albo nawet zerowy), ale taki pozornie bezwartościowy podział umożliwiłby następne, wartościowe podziały.

Druga metoda przycinania drzew decyzyjnych – **metoda przycinania końcowego** (ang. *postpruning*) – polega na zbudowaniu pełnego drzewa decyzyjnego i następnie usunięciu z niego węzłów, które mają niewielkie znaczenie dla procesu klasyfikacji. Ta metoda daje lepsze rezultaty, ale jest bardziej kosztowna. Do oceny zbudowanych drzew są używane dane walidacyjne. Oceniany jest każdy podział, co może prowadzić do nadmiernego dopasowania przyciętych w ten sposób drzew do danych walidacyjnych.

Do zdefiniowania, treningu i oceny jakości modelu drzew decyzyjnych użyjemy biblioteki `caret`. Najpierw podzielimy przypadki na zbiór treningowy (do którego trafi 70% losowo wybranych przykładów) i testowy (zawierający pozostałe 30% przykładów):

```
set.seed(1023)
train_idx <- caret::createDataPartition(y = mushrooms$class, p=0.7, list=FALSE)

train_mushrooms <- mushrooms[train_idx, ]
test_mushrooms <- mushrooms[-train_idx, ]
```

Jeżeli sprawdzimy rozkład zmiennej wyjściowej `class` w obu zbiorach, przekonamy się, że jest on taki sam jak w danych źródłowych. Pozwala to przypuszczać, że dane testowe są podobne do treningowych:

```
round(prop.table(table(mushrooms$class)), 2)
round(prop.table(table(train_mushrooms$class)), 2)
round(prop.table(table(test_mushrooms$class)), 2)
```

```
edible          poisonous
0.52            0.48
edible          poisonous
0.52            0.48
edible          poisonous
0.52            0.48
```

Następnie postawimy hipotezę, że jadalność grzyba zależy od cech wykrytych testem chi-kwadrat:

```
formula <- paste(relevant_features, collapse = "+")
formula <- as.formula(paste("class ~ ", formula))
formula
```

```
class ~ cap_shape + cap_surface + cap_color + bruises + odor +
     gill_attachment + gill_spacing + gill_size + gill_color +
     stalk_shape + stalk_root + stalk_surface_above_ring + stalk_surface_below_ring +
     stalk_color_above_ring + stalk_color_below_ring + veil_color +
     ring_number + ring_type + spore_print_color + population +
     habitat + cap_shape + cap_surface + cap_color + bruises +
     odor + gill_attachment + gill_spacing + gill_size + gill_color +
     stalk_shape + stalk_root + stalk_surface_above_ring + stalk_surface_below_ring +
     stalk_color_above_ring + stalk_color_below_ring + veil_color +
     ring_number + ring_type + spore_print_color + population +
     habitat
```

Do treningu modelu użyjemy funkcji `rpart`. Parametr `cp` pozwala określić koszt utworzenia nowego węzła, czyli kontrolować wzrost drzew decyzyjnych metodą przycinania wstępnego:

```
model_tree <- rpart(formula = formula,
                        data = train_mushrooms,
                        method = "class",
                        cp = 0.00005)
model_tree
```

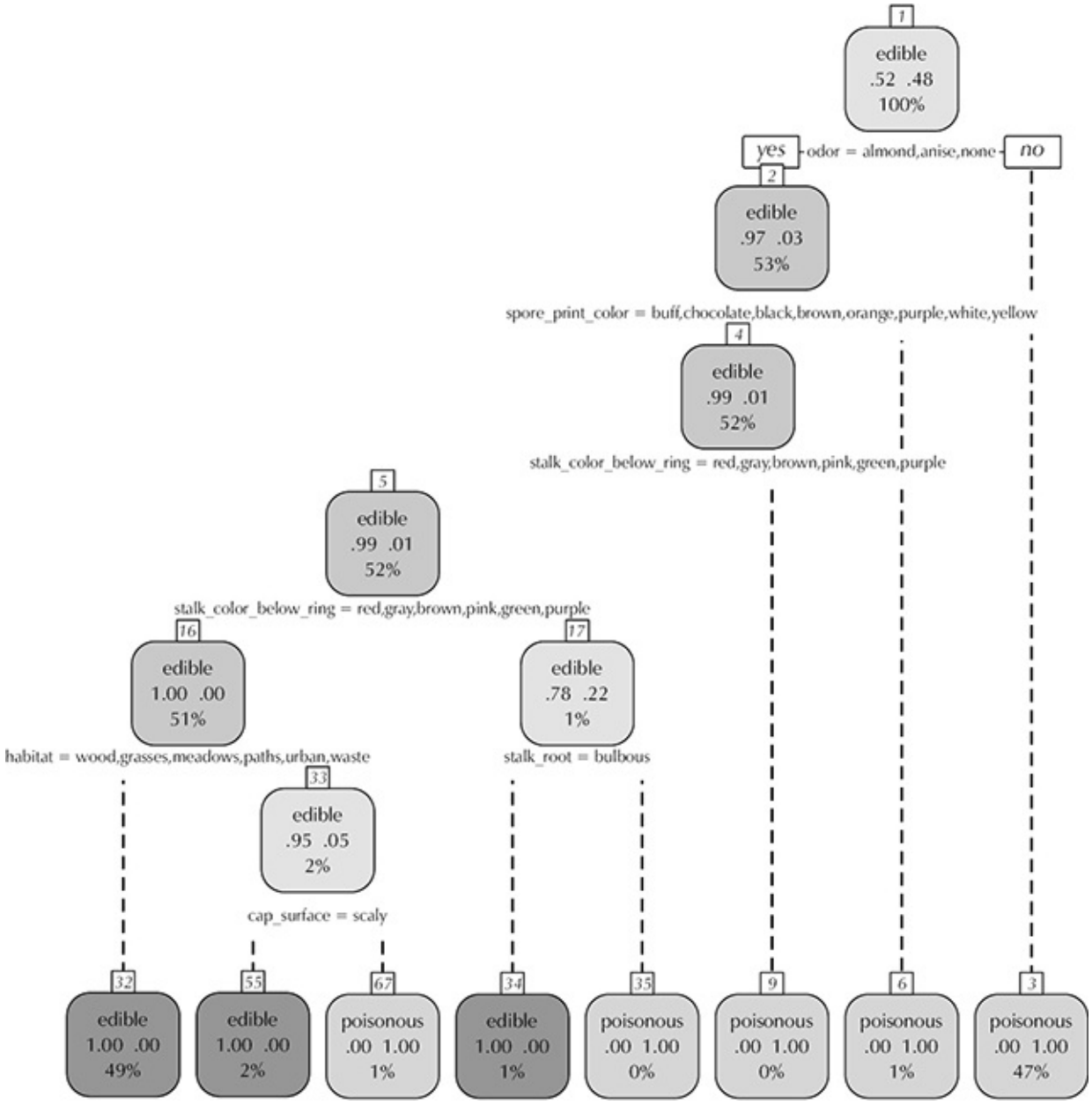
```
n= 5688

node), split, n, loss, yval, (yprob)
       * denotes terminal node

 1) root 5688 2742 edible (0.517932489 0.482067511)
   2) odor=almond,anise,none 3036    90 edible (0.970355731 0.029644269)
     4) spore_print_color=buff,chocolate,black,brown,orange,purple,white,yellow 2986    40
edible (0.986604153 0.013395847)
       8) stalk_color_below_ring=red,gray,brown,pink,green,purple 2965    19 edible
(0.993591906 0.006408094)
        16) stalk_color_below_ring=red,gray,pink,green,purple 2910     7 edible
(0.997594502 0.002405498)
          32) habitat=wood,grasses,meadows,paths,urban,waste 2778     0 edible
(1.000000000 0.000000000) *
          33) habitat=leaves 132     7 edible (0.946969697 0.053030303)
            66) cap_surface=scaly 125     0 edible (1.000000000 0.000000000) *
            67) cap_surface=grooves,smooth 7     0 poisonous (0.000000000 1.000000000) *
        17) stalk_color_below_ring=brown 55    12 edible (0.781818182 0.218181818)
          34) stalk_root=bulbous 43     0 edible (1.000000000 0.000000000) *
          35) stalk_root=missing 12     0 poisonous (0.000000000 1.000000000) *
       9) stalk_color_below_ring=white 21     0 poisonous (0.000000000 1.000000000) *
     5) spore_print_color=green 50     0 poisonous (0.000000000 1.000000000) *
   3) odor=creosote,foul,musty,pungent,spicy,fishy 2652     0 poisonous (0.000000000
1.000000000) *
```

Tak nauczony model drzew decyzyjnych zawiera 14 węzłów, w tym 8 węzłów najniższego poziomu (liści). Zgodnie z naszymi oczekiwaniami pierwszy podział nastąpił względem zmiennej `odor`. Pakiet `rpart.plot` pozwala przedstawić strukturę zbudowanego modelu drzew decyzyjnych na wykresie (rys. 3.21):

```
rpart.plot(model_tree, extra = 104, box.palette = "GnBn",
             branch.lty = 2, shadow.col = "gray", nn = TRUE)
```



Rysunek 3.21. Sześć znalezionych reguł wystarczyło do idealnego podzielenia wszystkich przykładów na grzyby jadalne i trujące

Przydatność poszczególnych zmiennych (ich siłę predykcyjną) w modelu możemy też sprawdzić funkcją `varImp`. Dla modeli drzew decyzyjnych utworzonych funkcją `rpart` przydatność zmiennych jest policzona na podstawie reguł, w których te zmienne zostały użyte i liczby przykładów treningowych sklasyfikowanych za pomocą tych reguł:

```
caret::varImp(model_tree)
```

	Overall
bruises	13.2575758
cap_color	29.4739052
cap_surface	13.2575758
gill_attachment	13.2575758
gill_color	1118.7530108
gill_size	15.8477586
gill_spacing	21.7354978
habitat	13.4723836
odor	2665.6777405
population	0.4714777
spore_print_color	1653.4785945
stalk_color_below_ring	86.0206134
stalk_root	18.7636364
stalk_surface_above_ring	1025.8094577
stalk_surface_below_ring	949.1826165
cap_shape	0.0000000
stalk_shape	0.0000000
stalk_color_above_ring	0.0000000
veil_color	0.0000000
ring_number	0.0000000
ring_type	0.0000000

Przykładowy model jest na tyle prosty, że mogliśmy go przedstawić graficznie i ocenić jego jakość, analizując wszystkie liście drzew decyzyjnych. Każdy z nich czysto dzieli przypadki, a więc wszystkie przykłady treningowe zostały bezbłędnie sklasyfikowane. W praktyce do oceny klasyfikatorów używa się standardowych miar jakości. W tym podrozdziale ograniczę się do policzenia błędnych i prawidłowych predykcji, omówienie miar jakości klasyfikatorów zostawiając na później.

Liczbę prawidłowych i błędnych predykcji można znaleźć w macierzy pomyłek. Macierz taką zwraca funkcja `confusionMatrix`. Jej pierwszym parametrem są wyniki predykcji, a drugim zbiór danych referencyjnych (do obliczenia błędu empirycznego użyjemy danych treningowych, a do obliczenia błędu rzeczywistego należy użyć danych testowych). Oba zbiory danych muszą mieć

wspólne wartości, dlatego do wykonania zapytań predykcyjnych została użyta funkcja `predict` z opcjonalnym parametrem `type`:

```
cm <- caret::confusionMatrix(data=predict(model_tree, type = "class"),
                                    reference = train_mushrooms$class)
cm$table
```

```
                Reference
Prediction       edible poisonous
  edible          2946       0
  poisonous          0    2742
```

Nasz model prawidłowo sklasyfikował 2946 grzybów jadalnych i 2742 trujących, nie popełniając przy tym żadnego błędu. Błąd empiryczny modelu mierzony funkcją kosztu 0-1 wynosi 0.

W ten sam sposób policzymy błąd rzeczywisty modelu. Zamiast danych treningowych dwukrotnie użyjemy danych testowych. Po pierwsze, za pomocą funkcji `predict` wykonamy zapytania predykcyjne, których wynikiem będzie ocena przez model grzyba jako jadalnego lub nie. Po drugie, w funkcji `confusionMatrix` wskażemy je jako dane referencyjne:

```
cm <- caret::confusionMatrix(data =
                            predict(model_tree, newdata = test_mushrooms, type = "class"),
                            reference = test_mushrooms$class)
cm$table
```

```
                       Reference
Prediction       edible poisonous
  edible          1262         0
  poisonous          0      1174
```

Przykłady testowe również zostały bezbłędnie sklasyfikowane przez nasz model.

3.5. Metodyka CRISP-DM

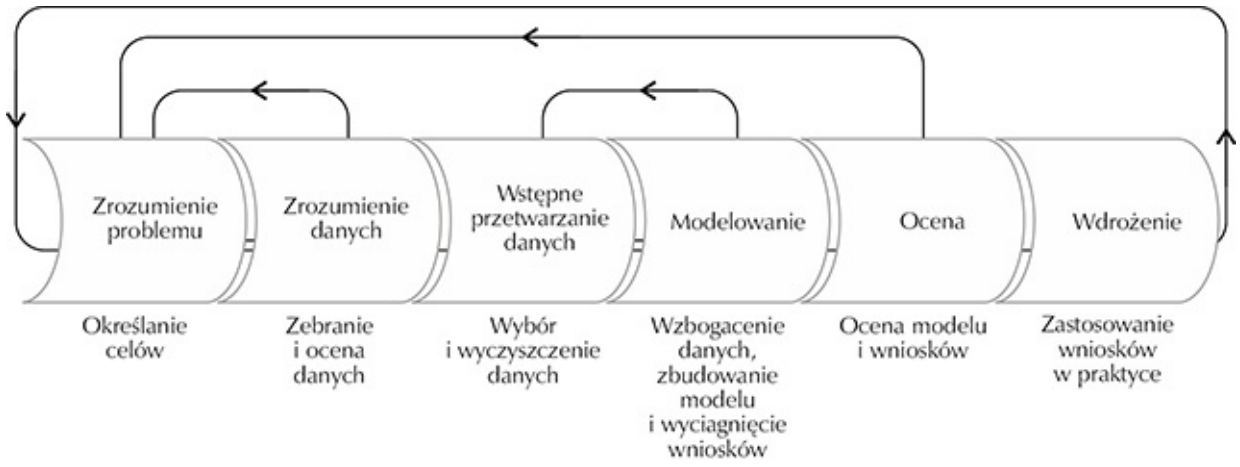
Przedstawiona we wstępie krótka historia sztucznej inteligencji zaczyna się w 1950 r. – wtedy została opublikowana praca Alana Turinga pt. *Computing Machinery and Intelligence*. Chociaż początki sztucznej inteligencji datuje się na czasy antyczne, to właśnie w latach 50. XX w. zostały wypracowane używane do dzisiaj metody uczenia maszynowego. W tym podrozdziale przedstawiam opracowaną w 1997 r. metodykę CRISP-DM (*CRoss Industry Standard Process for Data Mining)*. Tą metodą posługiwałem się we wszystkich opisanych w książce przykładach, akcentując w poszczególnych rozdziałach kolejne etapy określone w metodyce CRISP-DM.

Wybór i wdrożenie formalnej metodyki ma kapitalny wpływ na powodzenie projektów uczenia maszynowego. Przede wszystkim pozwala budować wiarygodne modele, czyli zapobiega próbom naginania rzeczywistości (danych) do postawionych hipotez i nieuprawomocnionego uogólniania hipotez. Ponadto, postępowanie zgodnie ze sprawdzonymi, dobrymi praktykami znacznie skraca czas i koszt projektów uczenia maszynowego.

Od przynajmniej dwudziestu lat naukowcy wiedzą o problemie dopasowywania przez badaczy danych do hipotez. Niestety wciąż nie wszyscy zdajamy sobie sprawę ze skali tego problemu – **większość opublikowanych prac naukowych zawiera fałszywe wnioski**. W teorii naukowcy formułują hipotezy na podstawie poczynionych obserwacji, a następie weryfikują je na podstawie przeprowadzonych doświadczeń, posługując się statystyką jako narzędziem do oceny danych. Jeżeli hipoteza okazuje się prawdziwa, publikowane są płynące z niej wnioski. W praktyce wiele badań zaczyna się od przyjęcia wniosku za prawdziwy i polega na takim dostosowaniu danych lub metod ich analizy, żeby z góry przyjęta teza została potwierdzona [17]. Przykładem takiego podejścia są analizy krzywej Phillipsa, czyli krzywej ilustrującej zależność między stopą bezrobocia a inflacją płac. Negatywna korelacja między tymi zmiennymi została zauważona w 1958 r. i choć od 50 lat jej prawdziwość jest podważana, nadal ma ona wpływ na politykę finansową niektórych krajów, a sześć spośród wielu

analiz krzywej Phillipsa zostało nagrodzonych nagrodą Nobla.

Postępowanie według sprawdzonych metod pozwala również uniknąć wielu błędów i skrócić czas projektu. Uczenie maszynowe jest trudnym zadaniem, a na końcowy wynik (model) ma wpływ wiele podjętych w trakcie projektu decyzji. Metodyka CRISP-DM pozwala firmom odnosić praktyczne korzyści z procesu wydobywania informacji ukrytych w danych (rys. 3.22).



Rysunek 3.22. Metodyka CRISP-DM podkreśla iteracyjny i zwinny charakter procesu odkrywania wiedzy z danych – wyniki każdego etapu są oceniane i na tej podstawie jest podejmowana decyzja o ewentualnym powtórzeniu któregoś z wcześniejszych kroków

Metodyka CRISP-DM określa kolejne etapy projektu wraz z opisami zadań do wykonania w każdym z nich [18]. Projekt powinien zacząć się od zrozumienia problemu, uwarunkowań biznesowych i określenia celów osiągalnych za pomocą uczenia maszynowego. Drugim etapem jest zrozumienie danych prowadzące do oceny ich reprezentatywności i przydatności w rozwiązaniu postawionego problemu. Trzeci etap polega na przygotowaniu danych do ich dalszej analizy. Jego efektem powinny być wyczyszczone i uporządkowane dane. Etap modelowania obejmuje dostosowanie danych do wymogów algorytmów uczenia maszynowego, postawienie hipotez i trening modeli. Następnie wytrenowane na tym etapie modele zostaną ocenione, a ich jakość poprawiona. Wdrożenie do produkcji wybranych, najlepszych modeli oraz opracowanie procedur ich aktualizacji kończy projekt.

Metodyka CRISP-DM jest opisowa, czyli określa zadania, ale nie precyzuje sposobu ich wykonania. Należy ją traktować jako zbiór wskazówek systematyzujących prace projektowe i pozwalających wracać do poprzednich etapów projektu. Przedstawione w kolejnych przykładach zadania są rozwiązane za pomocą różnych języków (SQL, R lub Python) i na różne sposoby (przy użyciu różnych bibliotek oraz własnych funkcji). Mam nadzieję, że urozmaici to lekturę oraz pozwoli wybrać najlepszy w danej sytuacji sposób wykonania postawionego zadania.

3.6. Metodyka TDSP

Metodyka TDSP (*Team Data Science Process*) to zwinna, opracowana przez firmę Microsoft z myślą o pracy zespołowej, metodyka prowadzenia projektów uczenia maszynowego. Jej założenia są zgodne z metodyką CRISP-DM, ale kładzie ona nacisk na wykonanie i udokumentowanie poszczególnych zadań.

Cykl życia projektu został podzielony na cztery etapy:

- zrozumienie problemu biznesowego,
- zdobycie i zrozumienie danych,
- modelowanie,
- wdrożenie.

Szczegółowy opis metodyki TDSP jest dostępny pod adresem https://docs.microsoft.com/en-us/azure/machine-learning/team-data-science-process/overview. Szczególnej uwadze polecam punkty poświęcone użyciu technologii Microsoft w projektach uczenia maszynowego oraz dostępne w serwisie GitHub narzędzia do analizy danych IDEAR.

Jeżeli będziesz analizował dane wystarczająco długo,
uzyskasz satysfakcjonującą odpowiedź.
Ronald H. Coase, laureat Nagrody Nobla w dziedzinie nauk ekonomicznych

Rozdział 4
Zrozumienie danych

W uczeniu maszynowym najważniejsze są dane. To dane powinny być źródłem stawianych hipotez i to od ich jakości i ilości zależą wyniki projektów. Klasyczne algorytmy uczenia maszynowego, takie jak przedstawione w książce, wydobywają ukryte w danych informacje na kilka sposobów. Jednak wszystkie używane przez te algorytmy metody mają swoje ograniczenia, a najskuteczniejszym sposobem ich przezwyciężenia jest użycie do treningu większej ilości lepszych jakościowo danych.


Zwiększając ilość danych, zbudujemy lepsze modele, niż stosując bardziej zaawansowane algorytmy uczenia maszynowego, ale najlepsze rezultaty uzyskamy, używając do treningu dużo danych lepszej jakości [19].

Używane do treningu modeli dane muszą mieć postać tabelaryczną. Ponadto dane te muszą być uporządkowane, czyli wiersze (rekordy) tabeli muszą reprezentować pojedyncze przykłady, a kolumny – zmienne [20]. Przykładowe dane zostały wcześniej uporządkowane, a proces ich uporządkowywania nie jest w książce przedstawiony. My, zgodnie z metodyką CRISP-DM, zaczynamy od zrozumienia problemu i danych.

Użyte w tym rozdziale dane pochodzą z serwisu Kaggle, https://www.kaggle.com/c/titanic/data.

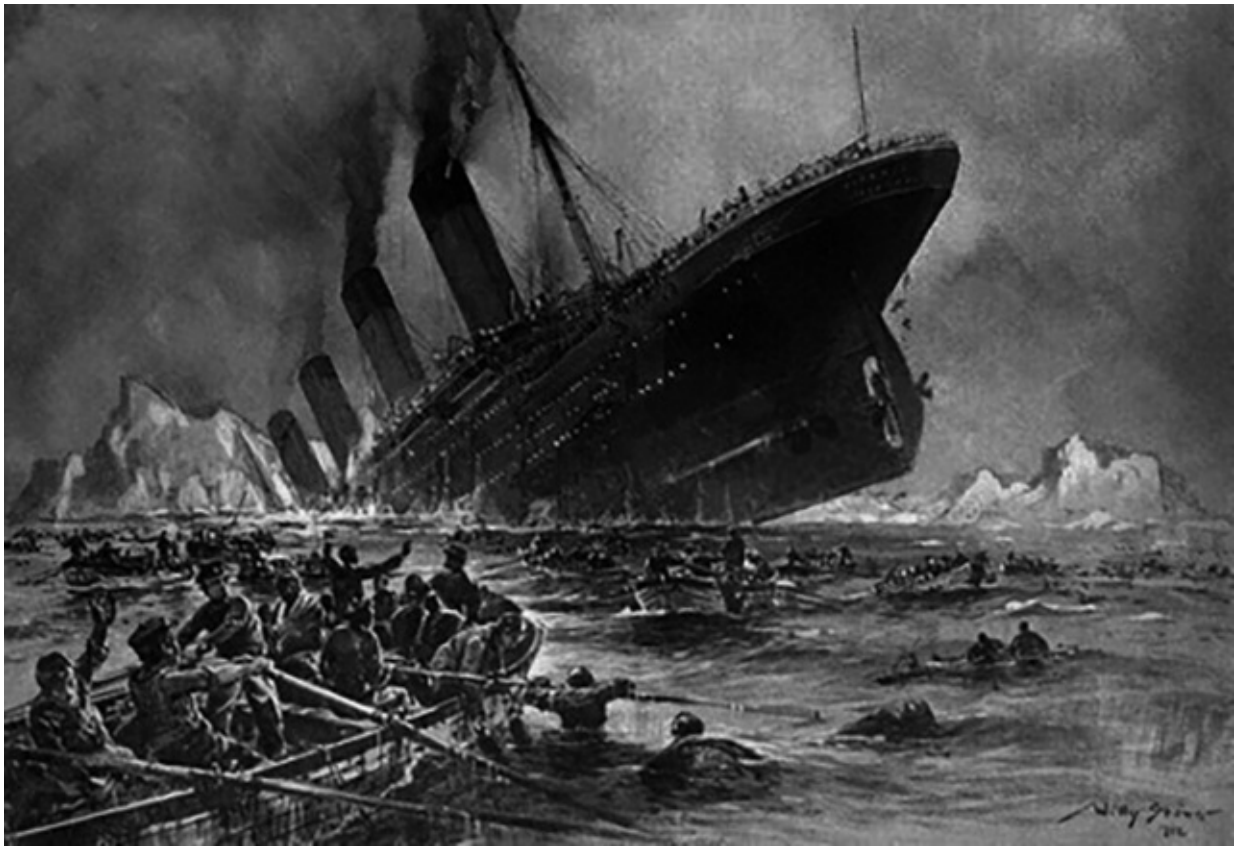
4.1.
Poznanie danych wymaga zrozumienia postawionego problemu

15 kwietnia 1912 r., w trakcie swojej pierwszej podróży, RMS Titanic zatonął po uderzeniu w górę lodową. W katastrofie zginęło 1502 z 2224 pasażerów i członków załogi. Jednym z powodów tak dużej liczby ofiar była niewystarczająca liczba łodzi ratunkowych. Oprócz szczęścia ocalonych pasażerów wyróżniały również inne cechy. W zrozumieniu, kto miał większą szansę przeżycia katastrofy, pomocne będzie poznanie jej przebiegu.

O godz. 23:40 14 kwietnia 1912 r. marynarze z bocianiego gniazda zauważyli górę lodową tuż przed dziobem statku. Oficer wachtowy wydał sternikowi rozkaz *prawo na burtę*, a maszynowni – *cała wstecz*. Od wykrycia góry lodowej do uderzenia w nią minęło 37 sekund. Początkowo nikt nie zdawał sobie sprawy z powagi sytuacji, a większość pasażerów nawet nie zauważyła uderzenia. Jednak w wyniku uderzenia dziobem i burtą o górę lodową woda zaczęła się wlewać do pięciu dziobowych przedziałów statku. RMS Titanic został zaprojektowany tak, aby utrzymać się na wodzie po zalaniu maksymalnie czterech przedziałów, a szybkość, z jaką wlewała się woda, uniemożliwiła jej wypompowywanie.

O godz. 24:00 kapitan Smith wysłał komunikat SOS, a dwadzieścia minut później zarządził ewakuację. O godz. 0:25 RMS Carpathia płynąca z Nowego Yorku potwierdziła odebranie sygnału SOS. Carpathia znajdowała się prawie 60 mil morskich od miejsca katastrofy, a maksymalna prędkość, do jakiej została zaprojektowana, wynosiła 14 mil morskich na godzinę. O godz. 0:45, godzinę po uderzeniu, do wody została opuszczona pierwsza szalupa (szalupa nr 7). Kolejne szalupy ratunkowe były opuszczane co kilka, kilkanaście minut, aż do zatonięcia statku o godz. 2:20. Carpathia, płynąc z prędkością, której nigdy przedtem, ani nigdy potem nie osiągnęła, przypłynęła na miejsce katastrofy o 4:10 i uratowała

wszystkich rozbitków znajdujących się w szalupach (rys. 4.1).



Rysunek 4.1. Zatonięcie RMS Titanic. Źródło: Willy Stöwer, Magazine Die Gartenlaube, Public Domain, https://commons.wikimedia.org/w/index.php?curid=97646

4.2.
Statystyki opisowe

Zrozumienie danych jest niezbędne do oceny ich jakości. W uczeniu maszynowym posługujemy się dużymi zbiorami danych, których liczba wierszy liczona jest w milionach, a kolumn – w dziesiątkach i setkach. Przeanalizowanie wszystkich przykładów, jeden po drugim, byłoby nie tylko niepraktyczne, lecz także niewykonalne. Tu z pomocą przychodzi statystyka. Statystyka uczy, jak zbierać i analizować dane oraz jak wnioskować na podstawie zebranych danych. Krótko mówiąc, statystyka pomaga zrozumieć dane.

Przyjrzyjmy się wyjątkowo małemu, jak na standardy uczenia maszynowego, zbiorowi danych treningowych:

```
SELECT *
FROM [ML].[Titanic].[Train];
```

```
PassengerId     Survived Pclass   Name     Sex      Age      SibSp    Parch
Ticket      Fare Cabin    Embarked
1       0        3        Braund, Mr. Owen Harris    male     22       1        0
A/5 21171       7.25     NULL     S
2       1        1        Cumings, Mrs. John Bradley (Florence Briggs
Thayer) female   38       1        0        PC 17599 71.2833  C85      C
3       1        3        Heikkinen, Miss. Laina     female   26       0        0
STON/O2. 3101282 7.925    NULL     S
…
```

Zbiór danych treningowych zawiera 891 przykładów (wierszy), każdy z nich jest opisany 12 atrybutami (zmiennymi). Poznawanie danych zaczyna się od analizy poszczególnych zmiennych. **Zmienne dzielą się na numeryczne (ilościowe) i kategoryczne (jakościowe)**.

Zmienne numeryczne to zmienne takich typów jak `bool`, `int`, `decimal`, `money` czy `date`, czyli zmienne, których wartościami mogą być wyłącznie liczby. **Ze względu na rozkład zmienne numeryczne dzieli się na dyskretne** (o skończonym zbiorze wartości) **i ciągłe** (o nieskończonym zbiorze wartości**)**. Zmienne numeryczne mogą być ze sobą porównywane, co więcej, możliwe jest określenie odległości między wartościami. Możliwe jest też wykonywanie na nich operacji arytmetycznych, takich jak mnożenie czy logarytmowanie.

Zmienne kategoryczne to zmienne pozostałych typów, przede wszystkim typów tekstowych, takich jak `char` czy `string`. Moc zbioru ich wartości zawsze jest ograniczona. Jeżeli wartości te mogą być w sensowny sposób ze sobą porównywane, to zmienną kategoryczną nazywa się **porządkową**, w przeciwnym razie mamy do czynienia z **regularną zmienną kategoryczną**. Zmienne kategoryczne nie umożliwiają określenia odległości między

wartościami, niemożliwe jest też wykonywanie na nich operacji arytmetycznych.

Aby ocenić rodzaj zmiennej, należy rozumieć jej znaczenie. W przykładowym zbiorze danych:

- zmienna `PassengerId` (*numer pasażera*) jest zmienną numeryczną ciągłą;
- zmienna `Survived` (*czy przeżył*) jest zmienną numeryczną dyskretną; ponieważ przyjmuje tylko dwa stany (pasażer mógł przeżyć lub nie), powinna być traktowana jako binarna zmienna kategoryczna;
- zmienna `Pclass` (*klasa*) jest zmienną numeryczną dyskretną; powinna być traktowana jako porządkowa zmienna kategoryczna, ponieważ jej stany opisują klasę wykupionej kajuty (stan 1 oznacza wyższą klasę, stan 2 – klasę średnią, a stan 3 – klasę niższą);
- zmienna `Name` (*nazwisko*) jest regularną zmienną kategoryczną;
- zmienna `Sex` (*płeć*) jest regularną zmienną kategoryczną o dwóch stanach, czyli zmienną binarną;
- zmienna `Age` (*wiek*) jest dyskretną zmienną numeryczną (moc zbioru wartości jest skończona);
- zmienna `SibSp` (*rodzeństwo*/*małżonkowie*) jest dyskretną zmienną numeryczną;
- zmienna `Parch` (*rodzice*/*dzieci*) jest dyskretną zmienną numeryczną;
- zmienna `Ticket` (*bilet*) jest regularną zmienną kategoryczną;
- zmienna `Fare` (*opłata*) jest przykładem ciągłej zmiennej numerycznej;
- zmienna `Cabin` (*kabina*) jest regularną zmienną kategoryczną;
- zmienna `Embarked` (*port zaokrętowania*) jest regularną zmienną kategoryczną.

Poniżej został przedstawiony sposób opisania rodzajów zmiennych w języku Python. Zmienna `Embarked` została uporządkowana – jej stany, czyli porty zaokrętowania, zostały ułożone w kolejności rejsu: Southampton–Cherbourg–Queenstown:

```
# Imports needed
import numpy as np
import pandas as pd
import pyodbc
import matplotlib.pyplot as plt
import seaborn as sns
import scipy as sc
import scipy.stats as st
import pylab
import statsmodels.api as sm

# Connecting and reading the data
con = pyodbc.connect(r'Driver={SQL
Server};Server=MS;Database=ML;Trusted_Connection=yes;')
query = """SELECT * FROM [Titanic].[Train];"""
data = pd.read_sql(query, con)
# Define variables as categorical
data['Survived'] = data['Survived'].astype('category')
data['Pclass'] = data['Pclass'].astype('category')
data['Name'] = data['Name'].astype('category')
data['Sex'] = data['Sex'].astype('category')
data['Ticket'] = data['Ticket'].astype('category')
data['Cabin'] = data['Cabin'].astype('category')
data['Embarked'] = data['Embarked'].astype('category')

# Reordering Embarked
data['Embarked'].cat.reorder_categories(
     ["S","C","Q"], inplace=True)
print(data['Embarked'].value_counts(sort = False))
```

```
S     644
C     168
Q      77
Name: Embarked, dtype: int64
```

Zrozumienie danych zaczyna się od poznania poszczególnych zmiennych. Najprostszym sposobem poznania zmiennej jest obliczenie jej statystyk opisowych. Opanowanie tego tematu jest niezbędne do zrozumienia dalszych rozdziałów książki.

Bezpośrednią korzyścią z wyliczenia i zrozumienia statystyk opisowych jest możliwość ich przedyskutowania z ekspertem z danej dziedziny. Taka dyskusja pozwala ocenić jakość i reprezentatywność danych. Niektóre statystyki opisowe, na przykład dominanta, są takie same dla zmiennych numerycznych i kategorycznych. Inne, do których należy rozstęp, mają takie samo znaczenie, ale dla różnych typów zmiennych są inaczej obliczane. Jeszcze inne, takie jak średnia, wyznacza się tylko dla zmiennych numerycznych.

4.2.1. Zmienne numeryczne

Rozkład wartości zmiennych numerycznych opisuje dziesięć najważniejszych statystyk opisowych:

- miejsce największej koncentracji wartości opisują miary tendencji centralnej: mediana, średnia arytmetyczna i dominanta (dominantę wyznacza się również dla zmiennych dyskretnych);
- zróżnicowanie wartości opisują miary rozproszenia (dyspersji): rozstęp, rozstęp (odchylenie) ćwiartkowy, wariancja, odchylenie standardowe i współczynnik zmienności;
- porównanie kształtu rozkładu zmiennej z rozkładem normalnym umożliwiają miary symetrii rozkładu: skośność i kurtoza.

Miary tendencji centralnej

Centralny punkt dystrybucji wartości często opisuje się za pomocą **średniej arytmetycznej**. Aby ją obliczyć, należy zsumować wszystkie wartości zmiennej i otrzymany wynik podzielić przez liczbę przykładów. Średnią arytmetyczną próby oznaczamy symbolem

$$\bar{x} = \frac{\sum x}{n}$$

```
print(data.Age.mean())
```

```
29.69911764705882
```

Średni wiek pasażerów Titanica wynosił 29,7 lat.

Drugą miarą tendencji centralnej jest **dominanta**. Wskazuje ona na najczęściej występującą wartość, lub wartości, zmiennej. Zmienne o jednej dominancie mają rozkład jednomodalny. Zmienne o dwóch dominantach mają rozkład dwumodalny, a zmienne o wielu dominantach – rozkład wielomodalny. Dominantę oznaczamy symbolem *D*.

```
print(data.Age.mode())
```

```
24.0
dtype: float64
```

Najwięcej pasażerów Titanica miało 24 lata.

Trzecią miarą tendencji centralnej jest **mediana**. Dzieli ona uszeregowany zbiór wartości na pół, czyli poniżej i powyżej mediany znajduje się 50% wszystkich wartości. Medianę oznacza się symbolem Me. Jeżeli liczba przykładów *n* jest nieparzysta,

$$Me = \frac{x_{n} + 1}{2}$$

czyli jest równa wartości przykładu będącego pośrodku uporządkowanej listy.

Jeżeli liczba przykładów n jest parzysta,

$$Me = \frac{1}{2}\left(x_{\frac{n}{2}} + x_{\frac{n+1}{2}}\right)$$

czyli jest równa średniej wartości dwóch środkowych przykładów.

```
print(data.Age.median())
```

```
28.0
```

Przeciętny wiek pasażerów Titanica wynosił 28 lat.

Mediana jest też nazywana drugim kwartylem (ang. *quartile*), co sugerowałoby istnienie innych kwartyli. Rzeczywiście tak jest. Oprócz mediany używane są pierwszy i trzeci kwartyl. Pierwszy (dolny) kwartyl znajduje się w 1/4 uszeregowanych wartości, a trzeci (górny) kwartyl dzieli uszeregowany zbiór wartości na dwie części: 75% jest położonych poniżej tego kwartyla, a 25% – powyżej. Warto zauważyć, że kwartyle są niektórymi z możliwych do wyznaczenia przy użyciu funkcji `quantile` kwantyli.

```
print(data.Age.quantile(0.25))
print(data.Age.quantile(0.75))
```

```
20.125
38.0
```

Pierwszy i trzeci kwartyl, poza drugim (medianą), wyznaczają stopień rozproszenia wartości. W następnym punkcie wyjaśniam, dlaczego powyższy wynik (`Q1 = 20, Q3 = 38`) pozwala przypuszczać, że co drugi pasażer Titanica miał między 20 a 38 lat.

Miary rozproszenia

Miary tendencji centralnej pozwalają wyznaczyć punkty największej koncentracji wartości. Miary rozproszenia pozwalają określić, jak bardzo wartości zmiennej są rozrzucone wokół tych punktów.

Najprostszą miarą rozproszenia jest rozstęp. **Rozstęp** to po prostu różnica między największą a najmniejszą wartością. Oznacza się go symbolem:

$$R = x_{\max} - x_{\min}$$

```
print(data.Age.min())
print(data.Age.max())
print('Range=%.3f' % (data.Age.max() - data.Age.min()))
```

```
0.42
80.0
Range=79.580
```

Różnica wieku między najstarszym a najmłodszym pasażerem Titanica wynosiła 79 lat i 6 miesięcy.

Prawidłowa interpretacja pozostałych miar rozproszenia wymaga znajomości rozkładu normalnego.

Spośród wszystkich występujących rozkładów jeden powtarza się tak często, że został nazwany rozkładem normalnym. **Rozkład normalny** (krzywa Gaussa) ma kształt dzwonu określonej wysokości i szerokości. Na osi odciętych *X* rozkładu normalnego znajdują się wartości zmiennej, a na osi rzędnych *Y* – gęstość (liczba wystąpień danej wartości).

Krzywa Gaussa jest symetryczna względem prostej równoległej do osi rzędnych przechodzącej w punkcie, w którym znajduje się wartość średnia. Oznacza to, że wartości skrajne, zarówno mniejsze, jak i większe od wartości średniej, występują rzadziej, a im bardziej są oddalone od punktu centralnego, tym mniejsze jest prawdopodobieństwo ich wystąpienia. W szczególności, 50% przykładów znajduje się w przedziale między pierwszym a trzecim kwartylem. Ta własność jest tak przydatna, że odległość od pierwszego do trzeciego kwartyla doczekała się własnej nazwy – nazywa się ją **rozstępem ćwiartkowym** (ang. *interquartile range*, IQR**)**.

```
print('IQR=%.3f' % (data.Age.quantile(0.75) - data.Age.quantile(0.25)))
```

```
IQR=17.875
```

Zróżnicowanie wieku pasażerów Titanica mierzone rozstępem ćwiartkowym wynosi 17,9.

Najczęściej używanymi miarami rozproszenia są wariancja i odchylenie standardowe. Obie są obliczane na podstawie odchyleń. Odchylenie to różnica między wartością danego przykładu a wartością średnią. **Wariancja (**ang. *variance*) to suma kwadratów odchyleń wyników od średniej podzielona przez liczbę wyników minus jeden. Wariancję oznaczamy symbolem σ^2

$$\sigma^2 = \frac{\Sigma\ (x_i - \bar{x})^2}{n - 1}$$

```
print(data.Age.var())
```

```
211.0191247463081
```

Ponieważ wartości wariancji są wyrażane w jednostkach kwadratowych, częściej używaną miarą rozproszenia jest **odchylenie standardowe** (ang. *standard deviation*, SD**)**, czyli pierwiastek kwadratowy z wariancji

$$\sigma = \sqrt{\sigma^2}$$

Odchylenie standardowe jest wyrażane w tych samych jednostkach co zmienna, dlatego łatwiej na jego podstawie oszacować rozproszenie wartości.

```
print(data.Age.std())
```

```
14.526497332334044
```

Typowa różnica między wiekiem pasażera a średnim wiekiem wszystkich pasażerów Titanica wyniosła 14,5 roku.

Dzieląc odchylenie standardowe przez średnią arytmetyczną i mnożąc

otrzymany wynik przez 100, otrzymamy **współczynnik zmienności (**ang. *coefficient of variation***)**. Współczynnik zmienności oznacza się symbolem

$$Vs = \frac{\sigma}{\bar{x}} * 100\%$$

Ponieważ jest on bezwymiarowy, pozwala porównywać rozproszenie zmiennych o różnych skalach i rozstępach.

```
print(data.Age.std() / data.Age.mean() * 100)
```

```
48.912218554656754
```

Różnorodność wieku pasażerów Titanica mierzona współczynnikiem zmienności wynosi 48,9%.

Miary symetrii rozkładu

Choć wiele zmiennych ma rozkład zbliżony do normalnego, rzadko ich rozkład jest idealnie normalny. Po pierwsze, wartości wyższe niż średnia mogą w zmiennej występować częściej lub rzadziej. Porównując średnią z dominantą, możemy z pewną dokładnością określić asymetrię rozkładu wartości. Miarą tego odchylenia jest **skośność (**ang. *skewness***)**. Skośność oznaczamy symbolem

$$W_s = \bar{x} - D$$

I tak, jeżeli średnia arytmetyczna jest większa od dominanty, to wartości powyżej średniej występują częściej niż wynikałoby to z rozkładu normalnego, a rzadziej występujące większe wartości tworzą na wykresie ogon ciągnący się w prawo. Taki rozkład nazywamy prawoskośnym.

Jeżeli wartość średnia jest mniejsza od dominanty, mamy do czynienia z rozkładem lewoskośnym, w którym wartości zmiennej częściej są większe od średniej, a wartości mniejsze tworzą charakterystyczny ogon ciągnący się w lewo na wykresie rozkładu.

Współczynnik skośności oprócz kierunku asymetrii wskazuje na jej siłę – wartości bliższe 0 oznaczają słabszą asymetrię. Im bliżej wartości ±1, tym asymetria jest silniejsza. Współczynnik asymetrii oznacza się symbolem A_S

$$A_S = \frac{W_S}{S}$$

```
print(data.Age.skew())
```

```
0.38910778230082704
```

Rozkład wieku pasażerów Titanica jest silnie prawoskośny, czyli wartości mniejsze od średniej występują w nim częściej niż wynikałoby to z rozkładu normalnego.

Drugą, oprócz skośności, miarą symetrii rozkładu jest **kurtoza** (ang. *kurtosis***)**. W rozkładzie normalnym wartość tej miary, mierzącej poziom zagęszczenia wokół wartości centralnej, wynosi 0. Kurtozę oznaczamy symbolem

$$\beta_2 = \frac{\mu_4}{s^4}$$

gdzie μ_4 jest czwartym momentem centralnym.

Jeżeli kurtoza jest mniejsza od zera, to mamy do czynienia z rozkładem spłaszczonym (garb na wykresie rozkładu zmiennej będzie niższy niż w rozkładzie normalnym), a więc wartości zmiennej będą bardziej zróżnicowane. Jeżeli kurtoza jest większa od zera, to mamy do czynienia z rozkładem wysmukłym (wysokość garbu na wykresie rozkładu zmiennej będzie większa niż dla rozkładu normalnego), czyli zmienna częściej będzie przyjmowała wartości zbliżone do średniej.

```
print(data.Age.kurt())
```

```
0.17827415364210353
```

Wiek poszczególnych pasażerów Titanica jest bliższy średniemu niż wynikałoby to z rozkładu normalnego.

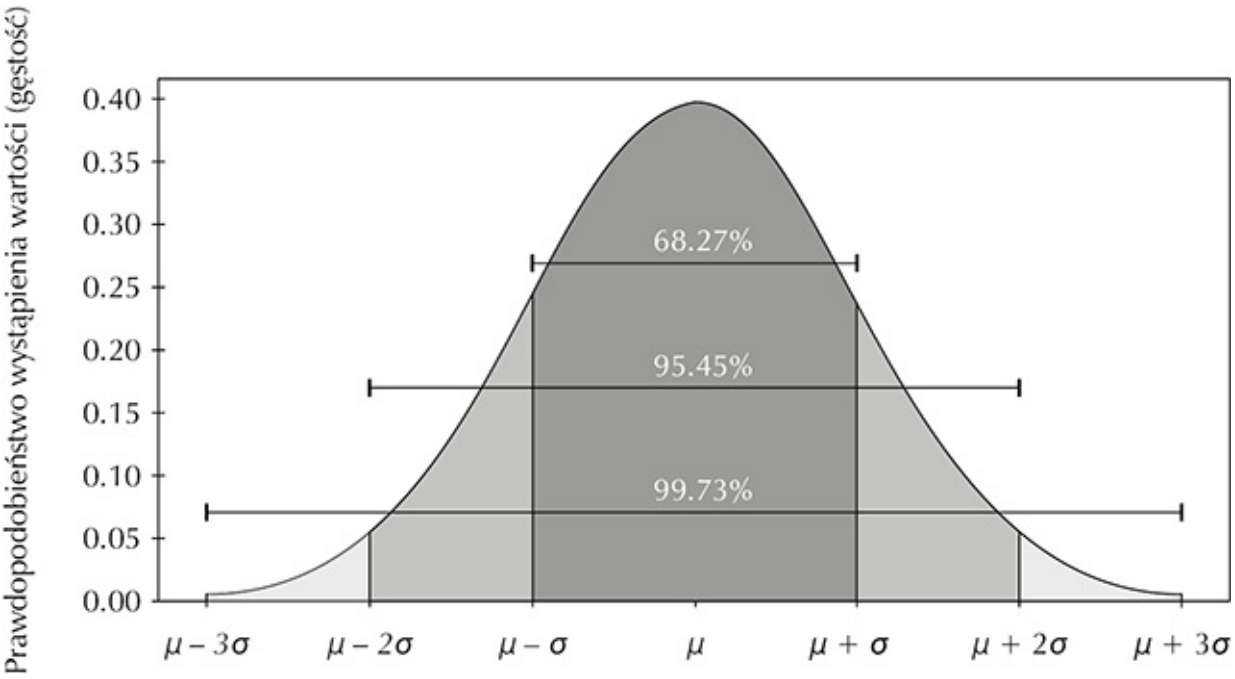
Wybrane statystyki opisowe zwraca funkcja `describe`:

```
print(data.Age.describe())
```

```
count    714.000000
mean      29.699118
std       14.526497
min        0.420000
25%       20.125000
50%       28.000000
75%       38.000000
max       80.000000
```

Rozkład normalny

Wielokrotnie wspominany w poprzednim punkcie rozkład normalny zawsze jest rozkładem jednomodalnym, czyli jedna wartość występuje w nim najczęściej. Odchylenie standardowe pozwala ocenić prawdopodobieństwo wystąpienia każdej wartości. Jeżeli rozkład badanej zmiennej jest zbliżony do normalnego, można do tego użyć **reguły 68–95–99,7**, na podstawie której 68% wartości znajduje się w odległości jednego odchylenia standardowego od średniej, a 95% i 99,7% wartości mieści się w przedziale, odpowiednio, ±dwóch i trzech odchyleń standardowych od średniej (rys. 4.2).



Rysunek 4.2. Rozkład normalny zmiennej i reguła 68–95–99,7, na podstawie której wiemy, że im wartość jest bardziej oddalona od średniej, tym mniejsze jest prawdopodobieństwo jej wystąpienia

Jak widać, niektóre wartości występują tak rzadko i są tak bardzo oddalone od średniej, że wydają się nienaturalne. Za **odstające wartości (**ang. *outliers***)** przyjmuje się wartości oddalone o więcej niż dwukrotność rozstępu ćwiartkowego od średniej arytmetycznej. Wartości odstające mają znaczący wpływ na niektóre statystyki opisowe, czyli występowanie wartości odstających oznacza, że niektóre statystyki nie są wiarygodne.

I tak, **miarą tendencji centralnej zależną od wartości odstających jest średnia, a miarami rozproszenia – rozstęp i odchylenie standardowe**. Oznacza to, że jeżeli występują wartości odstające, do określenia centralnego punktu dystrybucji należy użyć dominanty lub mediany, ale nie średniej. Natomiast do wyznaczenia rozproszenia należy użyć rozstępu ćwiartkowego. W tym przypadku średni wiek pasażerów wyniósł 29,7 lat, a IQR wynosi 17,9. Przedział typowych wartości zmiennej Age wynosi od minus sześciu (29,7 – 2*17,9 = –6.1) do sześćdziesięciu pięciu lat (29,7 + 2*17,9 = 65,5). Ponieważ największa wartość jest powyżej tak wyznaczonego przedziału (80),

bezpieczniej będzie założyć, że zmienna Age przyjmuje wartości odstające.

Powyższe reguły (reguła 68–95–99,7 i reguła dwukrotności IQR) mogą być stosowane do oceny wszystkich zmiennych o rozkładzie zbliżonym do normalnego. Należy je jednak traktować jako ogólne wskazówki, a nie ścisłe prawa. Na przykład wartość dwukrotności IQR jest arbitralnie wybrana i w przypadku niektórych zmiennych za wartości odstające należy uznać wartości odległe od średniej o trzykrotność przedziału IQR. Dlatego, jeżeli wymagana jest dokładniejsza analiza zmiennej, należy ją przeprowadzić na podstawie modelu rozkładu normalnego.

Ponieważ dominanta i odchylenie standardowe wystarczają do dokładnego opisania rozkładu normalnego, nazywa się je parametrami modelu rozkładu normalnego

$$N(\mu, \sigma)$$

Jeżeli najczęściej występującą wartością (dominantą) jest zero ($\mu = 0$), a odchylenie standardowe wynosi jeden ($\sigma = 1$), mamy do czynienia ze **standardowym rozkładem normalnym** (ang. *standard normal distribution*).

Z tego powodu normalizacja zmiennej, w wyniku której jej średnia otrzymuje wartość 0, a odchylenie standardowe 1, nazywa się standaryzacją. Najczęściej stosowaną metodą standaryzacji jest standaryzacja *Z*, polegająca na podzieleniu różnicy między zmienną nieustandaryzowaną a średnią przez odchylenie standardowe:

$$z = \frac{x - \mu}{\sigma}$$

Tak ustandaryzowaną zmienną można opisać statystyką *Z-score*. **Z-score definiuje się jako liczbę odchyleń standardowych, o którą wartość jest oddzielona od średniej**. *Z-score* równe 2 oznacza, że wartość jest o dwa odchylenia standardowe większa od średniej, a *Z-score* równe –1,5 oznacza, że

wartość jest o półtorakrotność odchylenia standardowego mniejsza od średniej. *Z-score* średniej arytmetycznej wynosi 0.

Statystykę *Z-score* można interpretować jako miarę odstawania – im większe *Z-score*, tym bardziej odstająca wartość, czyli wartości o większym *Z-score* są bardziej nietypowe od wartości o mniejszym *Z-score*, co pokazuje poniższy przykład:

```
print(st.zscore([0.45,23,25,28,33,60,80])), -1.4471941   -0.51970897 -0.43744864 -0.31405816
-0.10840735   1.002107   1.82471022]
```

Znając średnią arytmetyczną i odchylenie standardowe, możemy obliczyć Z-score wartości. Na przykład dla osiemdziesięcioletniego pasażera feralnego rejsu Titanica wynosi ono

$$Z\text{-}score = \frac{80 - 29{,}7}{14{,}5} = 3{,}46$$

Sprawdzając tę wartość w tabeli rozkładu zmiennej losowej albo używając odpowiedniej funkcji języka Python lub R, otrzymamy kwantyl rzędu jej wystąpienia (kwantylem rzędu *p* jest taka wartość zmiennej, że wartości mniejsze lub równe tej wartości występują z prawdopodobieństwem co najmniej *p*, a wartości większe – z prawdopodobieństwem 1 – *p*).

```
print(st.norm.cdf(3.46))
```

```
0.9997299123060365
```

Odejmując otrzymany wynik od 1, obliczymy prawdopodobieństwo wystąpienia wartości zmiennej o rozkładzie normalnym: 1 – 0,9997 = 0,0003. Czyli przy przyjęciu rozkładu normalnego zmiennej Age prawdopodobieństwo, że któryś z pasażerów Titanica miał 80 lat, wynosi 0,03%. Dzięki sprawdzeniu symetrii rozkładu zmiennej Age wiemy, że powinniśmy wziąć poprawkę na jej prawoskośność, czyli mniejsze prawdopodobieństwo wystąpienia wartości

powyżej średniej.

W ten sam sposób, czyli przez policzenie prawdopodobieństw wystąpienia wartości w pewnym kwantylu, została utworzona reguła 68–95–99,7. Sprawdźmy wartość *Z-score* dla 95 kwantyla

```
print(st.norm.ppf(.95))
```

```
1.6448536269514722
```

Okazuje się, że kwantyl o wartości *Z-score* równym 1,64 w zaokrągleniu wynosi 0,95:

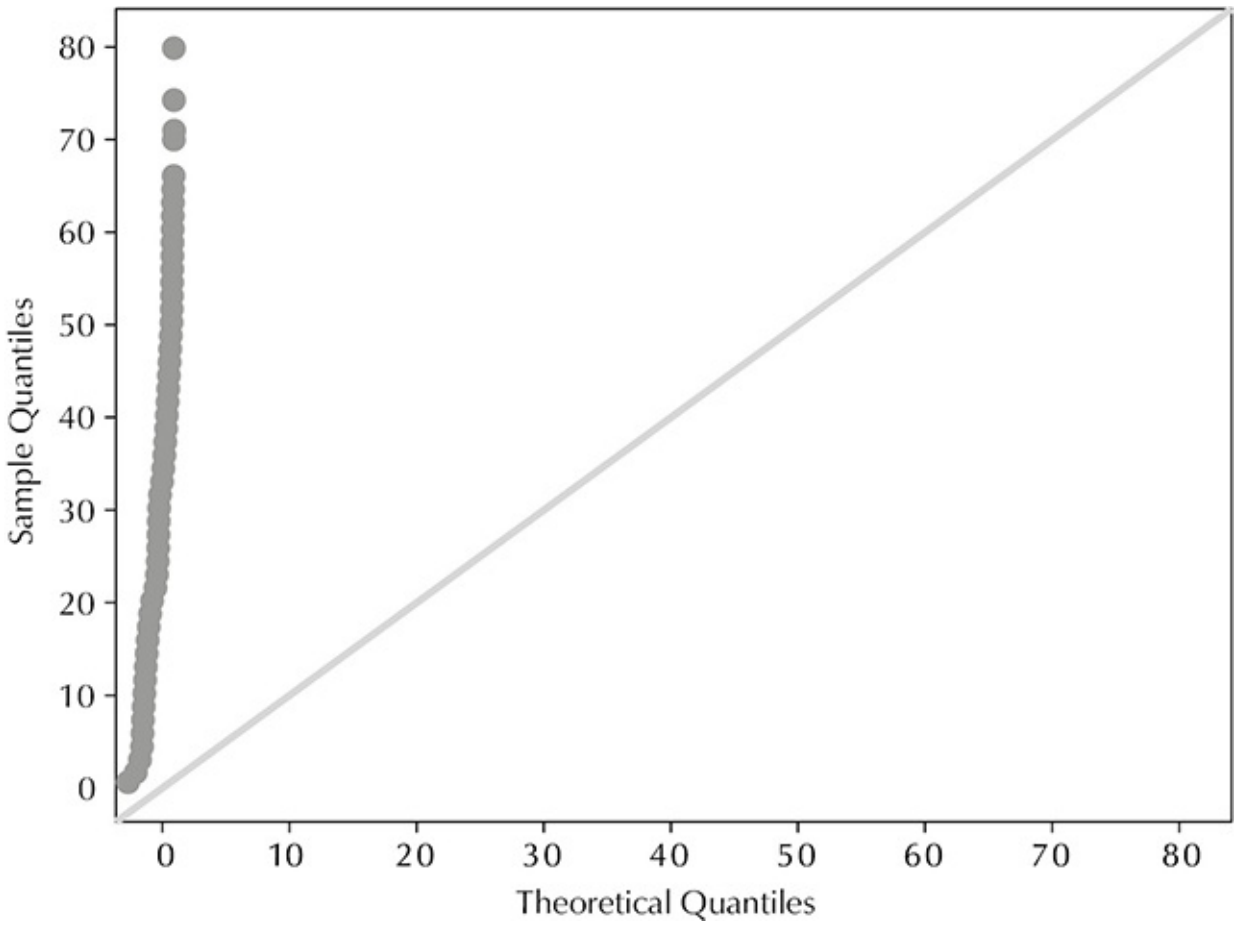
```
print(st.norm.cdf(1.64))
```

```
0.9494974165258963
```

Pora wyjaśnić, jak sprawdzić normalność rozkładu zmiennej. To ważne, bo duża część testów statystycznych zakłada określony rozkład zmiennych, najczęściej rozkład normalny. Takie testy nazywa się testami parametrycznymi. Z kolei testy, które nie zakładają określonego rozkładu zmiennej, nazywa się testami nieparametrycznymi. **Jeżeli zmienna ma rozkład zbliżony do normalnego, możemy użyć testów parametrycznych**. Zmienne o innych rozkładach można oceniać tylko testami nieparametrycznymi.

Najczęściej normalność rozkładu zmiennej sprawdza się na wykresach. Pierwszy sposób polega na nałożeniu krzywej rozkładu normalnego na histogram zmiennej. Histogramy przedstawione zostały w dalszej części rozdziału. Drugim, częściej stosowanym wykresem do oceny normalności rozkładu jest wykres kwantyl–kwantyl (wykres *QQ*). Na osi *X* wykresu *QQ* znajdują się teoretyczne (zgodne z rozkładem normalnym) kwantyle, a na osi *Y* zaobserwowane kwantyle. Im bliżej prostej znajdują się punkty, tym rozkład wartości zmiennej jest bliższy normalnemu (rys. 4.3).

```
sm.qqplot(data.Age, line='45')
pylab.show()
```



Rysunek 4.3. Rozkład wieku pasażerów Titanica jest prawoskośny, ale zbliżony do normalnego

Używanych jest też kilka testów statystycznych oceniających normalność rozkładu zmiennej. Najczęściej stosowanym testem jest test Shapiro–Wilka.

Hipoteza zerowa zakłada, że zmienna ma rozkład normalny (a dokładniej, że próba pochodzi z populacji o rozkładzie normalnym). Test Shapiro–Wilka polega na posortowaniu wartości i obliczeniu sumy kwadratów ich różnic od średniej:

$$S^2 = \sum(x - \overline{x})^2$$

Następnie oblicza się, przy użyciu tabeli statystycznych, wartości:

$$b = \sum_{(i-1)}^{m} (y_{n+1-i} - y_i)$$

Jeżeli liczba przykładów jest parzysta, *m* równa się jej połowie, jeżeli liczba przykładów jest nieparzysta, *m* równa się ich liczbie pomniejszonej o jeden i podzielonej przez dwa. Statystyka *W* jest ilorazem kwadratów wartości *b* i *S*:

$$W = \frac{b^2}{S^2}$$

Otrzymaną wartość statystyki *W* sprawdza się w tabelach statystycznych dla liczby obserwacji i przyjętego poziomu ufności alfa.

```
data_no_missing = data.dropna()
stat, p = st.shapiro(data_no_missing.Age)
print('Statistics=%.3f, p=%.3f' % (stat, p))
alpha = 0.05
if p > alpha:
     print('Sample looks Gaussian (fail to reject H0)')
else:
     print('Sample does not look Gaussian (reject H0)')
```

```
Statistics=0.991, p=0.300
Sample looks Gaussian (fail to reject H0)
```

Chociaż rozkład zmiennej `Age` nie jest idealnym rozkładem normalnym, brakuje nam dowodów na to, że różnica między rozkładem zmiennej `Age` a rozkładem normalnym jest dziełem przypadku. Powinniśmy więc przyjąć, że rozkład tej zmiennej jest zbliżony do normalnego.

4.2.2. Zmienne kategoryczne

Rozkład zmiennych kategorycznych ocenia się za pomocą tabel częstości oraz histogramów.

Tabela częstości zawiera zliczone częstości występowania poszczególnych stanów zmiennej, na przykład liczbę pasażerów, którzy zaokrętowali się

w poszczególnych portach.

```
print(pd.crosstab(index=data["Embarked"],  columns="Count"))
```

```
Embarked     Count
C              168
Q               77
S              644
```

Tabele częstości pozwalają też sprawdzić, ile stanów przyjmuje zmienna kategoryczna (w tym przypadku trzy), oraz czy wszystkie stany są poprawne. W praktyce zmienne kategoryczne są typów tekstowych i często się zdarza, że zawierają wiele błędów (literówek). Na szczęście w tym przypadku wszystkie trzy stany zmiennej `Embarked` są poprawne.

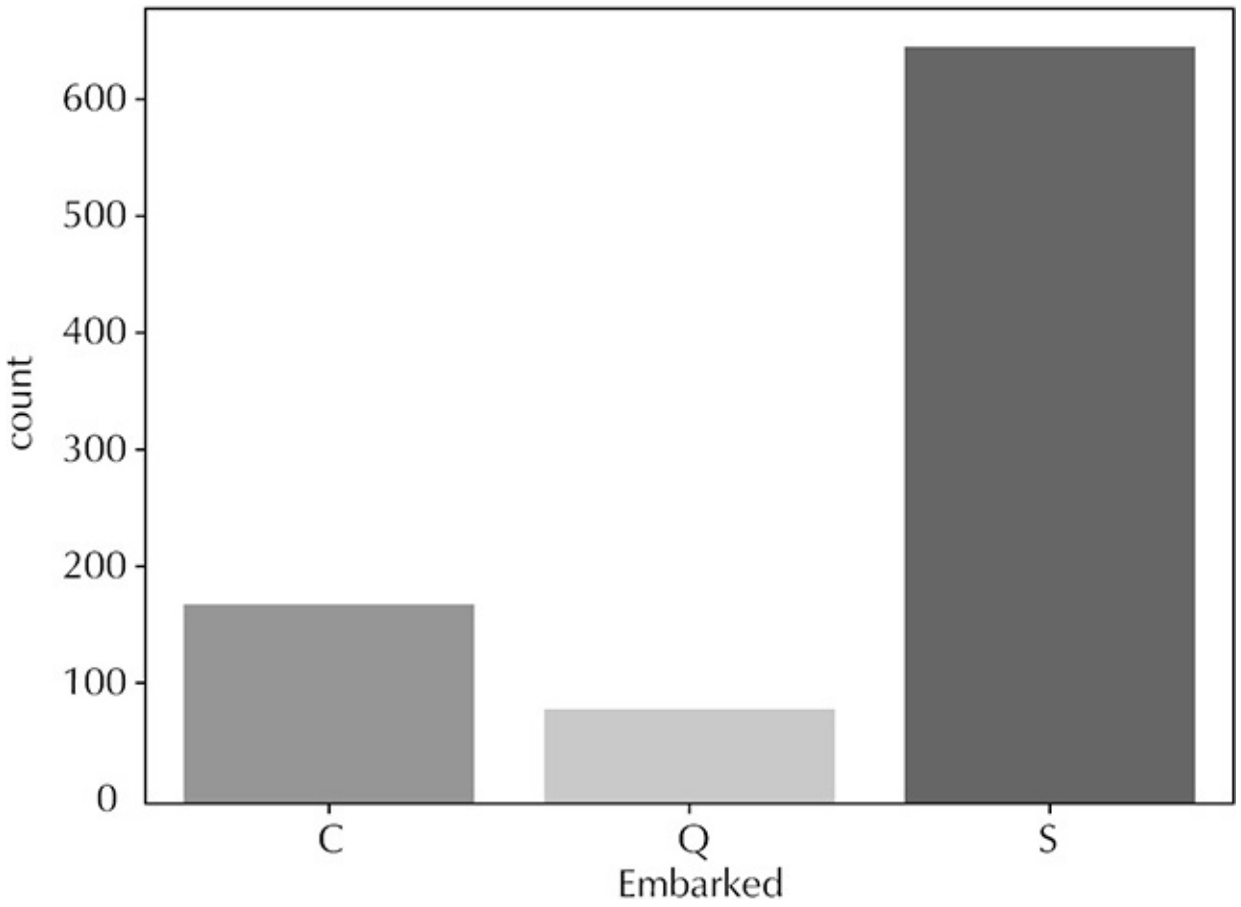
Pamiętajmy, że w zależności od funkcji użytej do ich obliczenia, tabele częstości mogą nie uwzględniać brakujących danych. Dlatego należy je dodatkowo policzyć. Więcej na ten temat jest w następnym punkcie.

```
print(data.Embarked.isnull().sum())
```

```
2
```

Graficzną reprezentacją tabel częstości są wykresy kolumnowe (rys. 4.4).

```
sns.countplot(x="Embarked", data=data);
plt.show()
```



Rysunek 4.4. Wysokość kolumn reprezentuje częstość występowania danego stanu zmiennej

Ponieważ możliwości oceny zmiennych kategorycznych są mniejsze niż zmiennych numerycznych, często konwertuje się je na zmienne numeryczne. Przetwarzaniu danych, w tym kodowaniu zmiennych kategorycznych za pomocą numerycznych, jest poświęcony rozdział 5.

4.3. Brakujące dane

Brakujące dane to nieznane wartości zmiennych. Braki w danych są koszmarem każdego projektu uczenia maszynowego. Ponieważ problem ten jest powszechny, musimy umieć sobie z nim radzić. Tę kwestię pozostawiam otwartą. Wrócę do niej w następnym rozdziale, na razie koncentruję się na

metodach oceny problemu brakujących danych.

Brak wartości może oznaczać, że jest ona istniejąca, ale nieznana. Z taką sytuacją mamy do czynienia, gdy nie wiemy, w jakiej kabinie podróżowała dana osoba. Ale brak wartości może też oznaczać jej nieistnienie. Z taką sytuacją będziemy mieli do czynienia, jeśli pasażer nie miał przydzielonej kabiny. Rozróżnienie tych dwóch, kompletnie różnych znaczeń może istotnie poprawić jakość danych. Wymaga to jednak opisania brakujących danych albo przez pobranie informacji o nich z innych źródeł, albo uzupełnienie z pomocą eksperta z danej dziedziny.

Najbardziej podstawowy i najprostszy test polega na policzeniu brakujących wartości. Ponieważ funkcja `isnull` zwraca 0 dla istniejących wartości, a 1 dla brakujących, wystarczy zsumować jej wyniki.

```
print(data.Age.isnull().sum())
```

```
177
```

Wiek 177 z 891 pasażerów znajdujących się w zbiorze treningowym jest nieznany, czyli brakuje danych o wieku prawie 20% pasażerów.

Brakujące dane mogą być też reprezentowane za pomocą zastępczych wartości, na przykład nieznany wiek może być zapisany jako 0. Takimi zastępczymi wartościami NULL najczęściej są wartości domyślne kolumn lub puste ciągi znaków. Jeżeli jakaś wartość występuje znacznie częściej niż pozostałe i niż wynika to z wiedzy dziedzinowej, prawdopodobnie reprezentuje ona wartości brakujące.

4.4. Entropia

W poprzednim rozdziale, przy okazji omawiania działania modeli

partycjonujących, wprowadziliśmy pojęcie entropii. Przypomnijmy – entropia jest miarą informacji zawartej w zmiennej. Metody jej pomiaru zostały określone w połowie XX w. przez Claude'a Elwooda Shannona [22]. Zmierzenie entropii zmiennej pozwala ocenić jej przydatność dla modeli uczenia predykcyjnego i choćby dlatego warto zrozumieć to pojęcie.

Najmniejszą jednostką informacji jest bit. Bit może przyjąć jeden z dwóch stanów (0 lub 1). Z tego powodu maksymalna ilość informacji, jaką można zapisać w zmiennej, jest równa logarytmowi o podstawie 2 z liczby wszystkich możliwych stanów tej zmiennej:

$$informacja\ w\ bitach = log_2(liczba\ możliwych\ stanów)$$

Im więcej informacji, czyli im bliższa jest liczba bitów informacji zmiennej tak obliczonemu maksimum, tym lepiej. Wyjątkiem od tej reguły są zmienne, których wartości się nie powtarzają, co zostanie wyjaśnione w punkcie poświęconym korelacji między zmiennymi. Policzmy i porównajmy maksymalną i faktyczną ilość informacji zawartej w trzech zmiennych: `Survived`, `Embarked` i `Name`.

```
def f_entropy(indata):
     indataprob = indata.value_counts() / len(indata)
     entropy=sc.stats.entropy(indataprob, base = 2)
     return entropy
print(np.log2(2), f_entropy(data.Survived), f_entropy(data.Survived) / np.log2(2))
print(np.log2(3), f_entropy(data.Embarked), f_entropy(data.Embarked) / np.log2(3))
print(np.log2(891), f_entropy(data.Name), f_entropy(data.Name) / np.log2(891))
```

```
1.0                    0.9607079018756469      0.9607079018756469
1.584962500721156      1.0968693499252116      0.6920475086483985
9.799281621521923      9.799281621521924       1.0000000000000002
```

Zmienna `Survived` może przyjąć jeden z dwóch stanów. Logarytm o podstawie 2 z 2 wynosi 1, czyli maksymalna ilość bitów informacji tej zmiennej wynosi 1. Faktyczna ilość informacji ukrytej w tej zmiennej wynosi 0,96. Maksymalna

ilość informacji ukrytej w zmiennej przyjmującej jeden z trzech stanów (takiej jak `Embarked`) wynosi 1,58 bita, a zmiennej przyjmującej jeden z 891 stanów – 9,79 bita. Zmienna `Name` niesie ze sobą maksymalną ilość informacji, bo nazwiska pasażerów się nie powtarzają, czyli każde z nich jest tak samo prawdopodobne.

4.5. Ocena zmiennych za pomocą programu Power BI

Języki Python i R mają nieograniczone możliwości analizowania danych. Mieliśmy okazję przekonać się o tym w poprzednich rozdziałach. W tym miejscu spróbujmy nieco innego sposobu szybkiej oceny danych. Jeżeli chcemy szybko ocenić dane, możemy użyć do tego programu Power BI Desktop – o ile ten program jest zainstalowany na komputerze. W takim przypadku należy uruchomić Power BI Desktop, połączyć się ze źródłem danych (w naszym przypadku serwerem SQL o nazwie `MS`) i wybrać tryb *DirectQuery*. W ten sposób unikniemy importowania ocenianych danych do pliku *pbix*. Po kliknięciu *OK* będziemy mogli wskazać analizowane dane, czyli tabelę `Titanic.Test` bazy `ML`. Po jej zaznaczeniu należy kliknąć przycisk *Edit* – zawartość tabeli zostanie wyświetlona w edytorze zapytań Power Query.

Tuż pod nagłówkami kolumn zobaczymy zielony lub szary pasek. Pasek koloru zielonego wskazuje procent istniejących wartości, a pasek szary – procent brakujących wartości[3]. Po ustawieniu kursora myszki nad tym paskiem zobaczymy liczby i procenty prawidłowych, brakujących i błędnych wartości zmiennej. Dane te, tak jak wszystkie pozostałe statystyki opisowe, są tworzone na podstawie próby tysiąca wierszy tabeli odczytanych jako pierwsze.

W zakładce *View* znajdują się dwie opcje pozwalające szybko ocenić brakujące dane: (opcja *Column quality*) oraz rozkład danych (opcja *Column distribution*).

Po ich zaznaczeniu poniżej nagłówka każdej kolumny zostanie wyświetlone okienko zawierające procent brakujących, błędnych i poprawnych danych, histogram oraz liczbę wszystkich i unikatowych (niepowtarzających się) wartości (rys. 4.5).



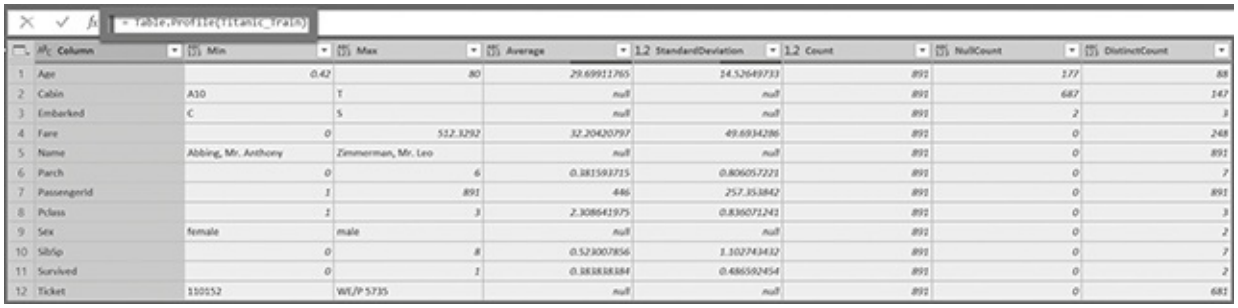
Rysunek 4.5. Przedstawiona wersja programu Power BI Desktop nie umożliwiała oceny danych na podstawie całego zbioru ani zapisania pokazanych wyników w postaci tabeli. Niemniej, prostota i szybkość, z jaką pozwala ona ocenić dane, stanowi potężną zaletę tego narzędzia

Statystyki opisowe zwraca funkcja `Table.Profile` używanego w programie Power BI Desktop języka Power Query. Można ją wywołać, przełączając się do widoku zaawansowanego edytora zapytań. W tym widoku zobaczymy tekst całego, składającego się z kroków, zapytania. Aby dodać kolejny krok, należy zakończyć poprzedni krok przecinkiem, a następnie podać nazwę kroku (np. `Profile`) i wywołać w nim funkcję `Table.Profile`, jako jej parametr podając nazwę poprzedniego kroku (w tym przypadku `Titanic_Train`). Ostatnia zmiana polega na uaktualnieniu nazwy kroku zwracanego przez zapytanie:

```
let
     Source = Sql.Databases("."),
     ML = Source{[Name="ML"]}[Data],
     Titanic_Train = ML{[Schema="Titanic",Item="Train"]}[Data],
     Profile = Table.Profile(Titanic_Train)
in
     Profile
```

Po zamknięciu okna zaawansowanego edytora zapytań zobaczymy efekt wykonania dodanej funkcji – tabelę zawierającą: wartości minimalne, maksymalne, średnie arytmetyczne, odchylenia standardowe, liczbę istniejących,

brakujących i niepowtarzających się wartości każdej zmiennej (rys. 4.6).



	Column	Min	Max	Average	StandardDeviation	Count	NullCount	DistinctCount
1	Age	0.42	80	29.69911765	14.52649733	891	177	88
2	Cabin	A10	T	null	null	891	687	147
3	Embarked	C	S	null	null	891	2	3
4	Fare	0	512.3292	32.20420797	49.6934286	891	0	248
5	Name	Abbing, Mr. Anthony	Zimmerman, Mr. Leo	null	null	891	0	891
6	Parch	0	6	0.381593715	0.806057221	891	0	7
7	PassengerId	1	891	446	257.353842	891	0	891
8	Pclass	1	3	2.308641975	0.836071241	891	0	3
9	Sex	female	male	null	null	891	0	2
10	SibSp	0	8	0.523007856	1.102743432	891	0	7
11	Survived	0	1	0.383838384	0.486592454	891	0	2
12	Ticket	110152	WE/P 5735	null	null	891	0	681

Rysunek 4.6. Statystyki opisowe zwracane przez funkcję Table.Profile

Przykładowy raport *Data profiles.pbix* znajduje się w folderze *Chapter04* repozytorium z przykładami do książki.

4.6.
Ocena zmiennych przy użyciu języka SQL

Jeżeli do analizy danych nie możemy użyć ani języków R lub Python, ani programu Power BI Desktop, kolejnym dostępnym narzędziem analizy danych jest język SQL. Policzenie niektórych statystyk, takich jak rozstęp, średnia czy odchylenie standardowe, sprowadza się do wywołania odpowiednich funkcji grupujących.

```
SELECT MIN(Age) AS minA,
 MAX(Age) AS maxA,
 MAX(Age) - MIN(Age) AS rngA,
 AVG(Age) AS avgA,
 COUNT(Age) AS cAge,
 STDEV(1.0*Age) as stdevAge,
 STDEV(1.0*Age) / AVG(1.0*Age) AS CVAge,
 COUNT(*) - COUNT(Age) AS missingAge,
 COUNT(DISTINCT(Age)) AS uniqueAge
FROM [Titanic].[Train];
```

minA	maxA	rngA	avgA	cAge	stdevAge	CVAge	missingAge	uniqueAge
0.42	80	79.58	29.69	714	14.52	0.48	177	88

Równie łatwo możemy policzyć dominantę:

```
SELECT TOP (1) WITH TIES Age, COUNT(*) AS Number
FROM [Titanic].[Train]
GROUP BY Age
ORDER BY COUNT(*) DESC;
```

```
Age     Number
NULL    177
```

Okazuje się, że – jak to czasami bywa – najczęstszą wartością jest wartość nieznana NULL. Jeżeli interesuje nas najczęściej występująca wartość, powinniśmy wyeliminować wartości nieznane:

```
SELECT TOP (1) WITH TIES Age, COUNT(*) AS Number
FROM [Titanic].[Train]
WHERE Age is NOT NULL
GROUP BY Age
ORDER BY COUNT(*) DESC;
```

```
Age     Number
24      30
```

Niestety, język SQL nie zawiera wielu funkcji statystycznych. Policzenie częstości – i wielu innych statystyk – wymaga ich samodzielnego zaimplementowania. Pomogą nam w tym podzapytania oraz klauzula `OVER`.

Język SQL pozwala na zagnieżdżenie zapytań, czyli umieszczenie instrukcji `SELECT` wewnątrz innej instrukcji języka. Tak **zagnieżdżone zapytania nazywa się podzapytaniami**. Jeżeli zapytanie wewnętrzne zwraca dane tabelaryczne, możemy odwoływać się do jego wyniku w klauzuli `FROM` zewnętrznej instrukcji; możemy też nadać mu nazwę. Nazwane wyrażenia tabelaryczne (ang. *Common Table Expressions*, CTE) definiuje się za pomocą słowa kluczowego `WITH`, a odwołuje się do niego przez nazwę.

Klauzula OVER pozwala wywołać funkcje rankingu, grupujące i analityczne dla określonych partycji. Grupowanie jest transformacją typu N

do 1, co oznacza, że każdy wiersz otrzymany w wyniku grupowania powstaje na podstawie jednego lub więcej wierszy tabeli źródłowej. Partycjonowanie jest transformacją typu N do N, co oznacza, że pozwala ono połączyć w wyniku zapytania dane szczegółowe z danymi na różnych poziomach ogólności. Do partycjonowania danych służy klauzula `OVER`. Zdefiniowane w tej klauzuli grupy są nazywane partycjami i mogą być użyte do wywołania funkcji grupujących oraz funkcji rankingu. Składnia klauzuli `OVER` wygląda następująco:

```
OVER  (
        [PARTITION BY wyrażenie partycjonujące] [,...n]
        [ORDER BY wyrażenie sortujące [ ASC | DESC ] [,...n] ]
        [okienko danych] )
```

gdzie:

- *okienko danych* umożliwia ograniczenie wierszy widocznych w danym momencie; w rezultacie wywołana dla tej klauzuli funkcja „zobaczy” w danym momencie tylko część wierszy, na przykład tylko wiersz bieżący; podczas kolejnego wywołania okienko przesunie się do następnego wiersza i ta sama funkcja zostanie wywołana dla kolejnego wiersza; okienko danych może być zdefiniowane przy użyciu znaczników:
  - `CURRENT ROW` – w ten sposób ograniczamy jego wielkość do bieżącego wiersza,
  - `PRECEDING` – do podanej liczby wierszy poprzedzających bieżący, przy czym konstrukcja `UNBOUNDED PRECEDING` oznacza wszystkie wiersze od pierwszego do bieżącego w danej partycji,
  - `FOLLOWING` – do podanej liczby wierszy następujących po bieżącym, przy czym konstrukcja `UNBOUNDED FOLLOWING` oznacza wszystkie wiersze od bieżącego do ostatniego w danej partycji.

Na przykład do policzenia częstości i kwantyli oraz narysowania histogramu

zmiennej Age możemy użyć poniższego zapytania[4]:

```
WITH freqCTE AS
(
SELECT v.[Age],
 COUNT(v.[Age]) AS AbsFreq,
 CAST(ROUND(100. * (COUNT(v.[Age])) /
        (SELECT COUNT(*) FROM [Titanic].[Train]), 0) AS INT) AS AbsPerc
FROM [Titanic].[Train] AS v
GROUP BY v.[Age]
)
SELECT [Age],
 AbsFreq,
 SUM(AbsFreq)
  OVER(ORDER BY [Age]
        ROWS BETWEEN UNBOUNDED PRECEDING
            AND CURRENT ROW) AS CumFreq,
 AbsPerc,
 SUM(AbsPerc)
  OVER(ORDER BY [Age]
        ROWS BETWEEN UNBOUNDED PRECEDING
            AND CURRENT ROW) AS CumPerc,
 CAST(REPLICATE('*',AbsPerc) AS VARCHAR(50)) AS Histogram
FROM freqCTE
ORDER BY [Age];
```

```
Age     AbsFreq  CumFreq  AbsPerc  CumPerc  Histogram
…
15      5        83       1        7        *
16      17       100      2        9        **
17      13       113      1        10       *
18      26       139      3        13       ***
19      25       164      3        16       ***
20      15       179      2        18       **
20.5    1        180      0        18
21      24       204      3        21       ***
22      27       231      3        24       ***
23      15       246      2        26       **
23.5    1        247      0        26
24      30       277      3        29       ***
…
```

Klauzula `OVER` pozwala też wywoływać funkcje analityczne. Zaimplementowana w serwerze SQL Server wersja języka SQL udostępnia cztery takie funkcje. Dwie z nich zwracają ranking wiersza w danej partycji, dwie kolejne kwartyl dla przekazanego procentu.

Ranking wiersza może być obliczony za pomocą funkcji `PERCENT_RANK` oraz `CUME_DIST`:

- funkcja `PERCENT_RANK` zwraca ranking wiersza w danej partycji obliczany według wzoru `(RN – 1) / (RC – 1)`, gdzie `RN` to kolejny numer wiersza, a `RC` to liczba wierszy w danej partycji;
- funkcja `CUME_DIST` zwraca ranking wiersza w danej partycji obliczany według wzoru `RS/RC`, gdzie `RS` to liczba wierszy o wartości mniejszej niż wiersza bieżącego lub mu równej, a `RC` to liczba wierszy w danej partycji.

Pozostałe funkcje analityczne są odwrotnościami poprzednich i zwracają kwartyl dla przekazanego procentu:

- funkcja `PERCENTILE_CONT` zwraca wynik, który nie musi pokrywać się z jakąkolwiek wartością w danej partycji;
- w przypadku funkcji `PERCENTILE_DISC` zwrócony wynik zawsze pokrywa się z najbliższą (względem pozycji) wartością występującą w danej partycji.

Wszystkie cztery funkcje analityczne do określenia kolumny, której wartości będą analizowane, używają znacznika `WITHIN GROUP`.

Znalezienie dominanty przy użyciu funkcji `PERCENTILE_CONT` sprowadza się do wykonania poniższego zapytania:

```
SELECT DISTINCT
 PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY 1.0*Age) OVER () AS Median
FROM [Titanic].[Train];
```

```
Median
28
```

Ta sama funkcja pozwoli nam obliczyć rozstęp ćwiartkowy:

```
SELECT DISTINCT
 PERCENTILE_CONT(0.75) WITHIN GROUP (ORDER BY 1.0*Age) OVER () -
 PERCENTILE_CONT(0.25) WITHIN GROUP (ORDER BY 1.0*Age) OVER () AS IQR
FROM [Titanic].[Train];
```

```
 IQR
 17.875
```

Język SQL nie udostępnia funkcji miar symetrii rozkładu, a więc ich policzenie wymaga samodzielnej implementacji.

Skośność policzymy ze wzoru:

$$Skew = \frac{n}{(n-1)*(n-2)} * \Sigma_{i=1}^{n}\left(\frac{X_i - \mu}{\sigma}\right)^3$$

```
WITH SkewCTE AS
(
SELECT SUM(1.0*Age) AS rx,
 SUM(POWER(1.0*Age,2)) AS rx2,
 SUM(POWER(1.0*Age,3)) AS rx3,
 COUNT(1.0*Age) AS rn,
 STDEV(1.0*Age) AS stdv,
 AVG(1.0*Age) AS av
FROM [Titanic].[Train]
)
SELECT
    (rx3 - 3*rx2*av + 3*rx*av*av - rn*av*av*av)
    / (stdv*stdv*stdv) * rn / (rn-1) / (rn-2) AS Skewness
FROM SkewCTE;
```

```
Skewness
0.389107782300826
```

Kurtozę można wyliczyć na podstawie wzoru:

$$Kurt = \frac{n*(n+1)}{(n-1)*(n-2)*(n-3)} * \sum_{i=1}^{n}\left(\frac{x_i - \mu}{\sigma}\right)^4 - \frac{3*(n+1)^2}{(n-2)*(n-3)}$$

```
WITH KurtCTE AS
(
SELECT SUM(1.0*Age) AS rx,
 SUM(POWER(1.0*Age,2)) AS rx2,
 SUM(POWER(1.0*Age,3)) AS rx3,
 SUM(POWER(1.0*Age,4)) AS rx4,
 COUNT(1.0*Age) AS rn,
 STDEV(1.0*Age) AS stdv,
 AVG(1.*Age) AS av
FROM [Titanic].[Train]
)
SELECT
    (rx4 - 4*rx3*av + 6*rx2*av*av - 4*rx*av*av*av + rn*av*av*av*av)
    / (stdv*stdv*stdv*stdv) * rn * (rn+1) / (rn-1) / (rn-2) / (rn-3)
    - 3.0 * (rn-1) * (rn-1) / (rn-2) / (rn-3) AS Kurtosis
FROM KurtCTE;
```

```
Kurtosis
0.178274153642101
```

Do policzenia entropii zmiennej potrzebne będą częstość i prawdopodobieństwo wystąpienia jej poszczególnych stanów. Dane te policzymy w kolejnych podzapytaniach nazwanych:

```
WITH ProbabilityCTE AS
(
SELECT Age,
 COUNT(Age) AS StateFreq
FROM [Titanic].[Train]
WHERE Age IS NOT NULL
GROUP BY Age
),
StateEntropyCTE AS
(
SELECT Age,
 1.0*StateFreq / SUM(StateFreq) OVER () AS StateProbability
FROM ProbabilityCTE
)
SELECT 'Age' AS Variable,
 LOG(COUNT(*),2) AS MaxPossibleEntropy,
 (-1)*SUM(StateProbability * LOG(StateProbability,2)) AS TotalEntropy,
 100 * ((-1)*SUM(StateProbability * LOG(StateProbability,2))) /
 (LOG(COUNT(*),2)) AS PctOfMaxPossibleEntropy
FROM StateEntropyCTE;
```

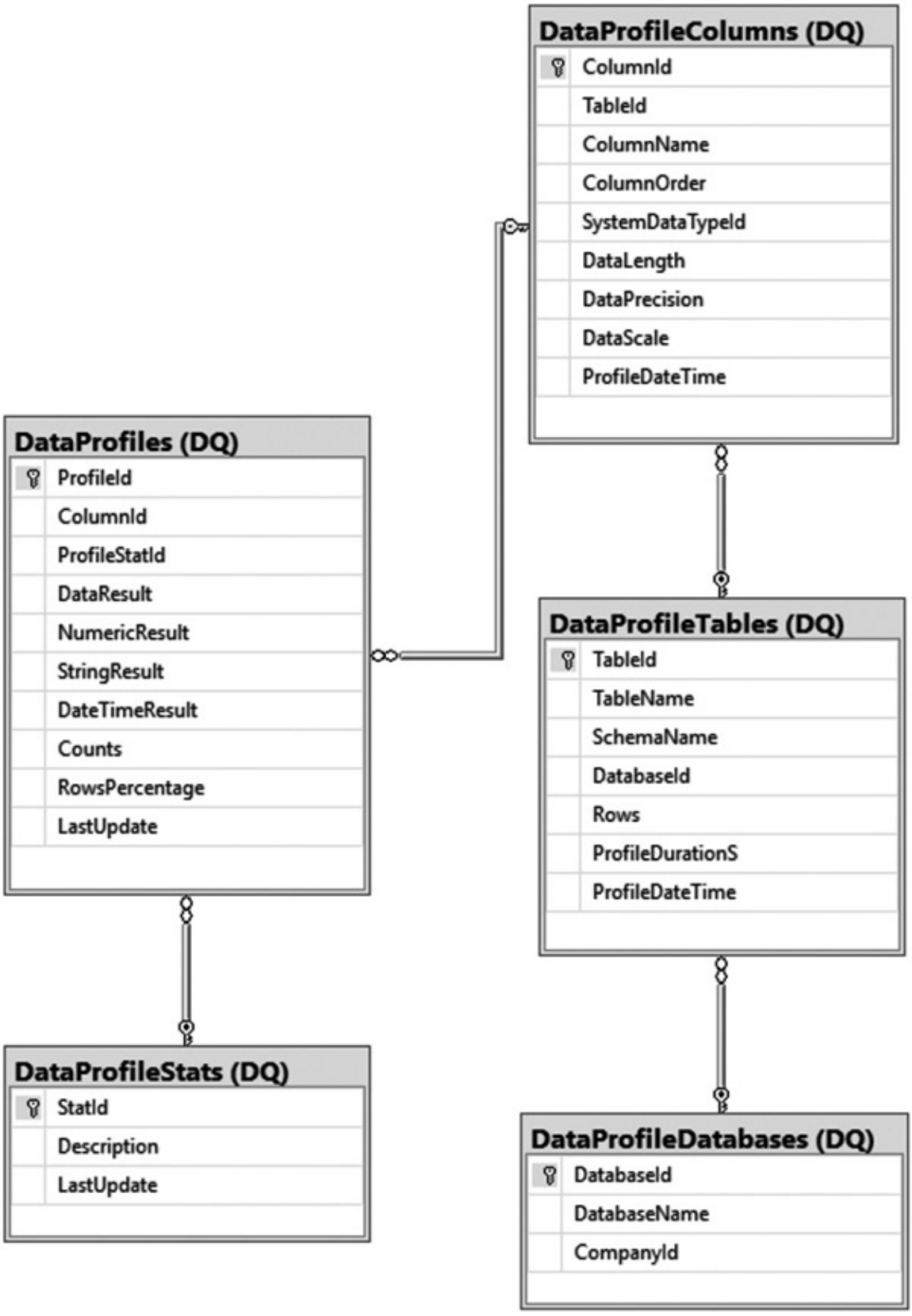
```
Variable         MaxPossibleEntropy TotalEntropy       PctOfMaxPossibleEntropy
Age     6.4594316186373    5.8365836339362    90.3575419406251
```

4.6.1. Automatyczny opis zmiennych

Każdy projekt uczenia maszynowego zaczyna się od zrozumienia problemu i danych. Ocena jakości danych jest też ważnym elementem innych projektów informatycznych. Dotyczy to projektów obejmujących przetwarzanie danych, takich jak projekty Business Intelligence, projekty migracji danych i wdrożeń systemów bazodanowych. Wiemy, że często wykonywane zadania należy automatyzować, więc ocenę jakości danych również należy zautomatyzować. Jeden ze sposobów polega na użyciu dynamicznego kodu SQL[5].

Aby narzędzie było uniwersalne, obliczone statystyki będą zapisywane w pięciu powiązanych ze sobą tabelach:

- tabela `[DQ].[DataProfileStats]` przechowuje nazwy wyliczanych statystyk, takie jak `Avg` czy `Values distribution`;
- tabela `[DQ].[DataProfileDatabases]` przechowuje nazwy baz danych, z których zostały odczytane analizowane dane;
- tabela`[DQ].[DataProfileTables]` przechowuje informacje o tabelach, z których zostały odczytane dane;
- tabela `[DQ].[DataProfileColumns]` przechowuje informacje o analizowanych kolumnach;
- obliczone statystyki są zapisywane w tabeli `[DQ].[DataProfiles]` (rys. 4.7).


DataProfileColumns (DQ)
ColumnId
TableId
ColumnName
ColumnOrder
SystemDataTypeId
DataLength
DataPrecision
DataScale
ProfileDateTime
DataProfiles (DQ)
ProfileId
ColumnId
ProfileStatId
DataResult
NumericResult
StringResult
DateTimeResult
Counts
RowsPercentage
LastUpdate
DataProfileTables (DQ)
TableId
TableName
SchemaName
DatabaseId
Rows
ProfileDurationS
ProfileDateTime
DataProfileStats (DQ)
StatId
Description
LastUpdate
DataProfileDatabases (DQ)
DatabaseId
DatabaseName
CompanyId

Rysunek 4.7. Taki model pozwala przechowywać wyniki testów statystycznych danych pochodzących z dowolnych baz

Wyliczenie statystyk dla danych z tabeli `Titanic.Train` sprowadza się do wywołania procedury `[DQ].[DataProfiling]`:

```
DECLARE @RC int
DECLARE @DatabaseName nvarchar(50)
DECLARE @SchemaName nvarchar(50)
DECLARE @TableView char(1)
DECLARE @Debug bit

EXECUTE @RC = [DQ].[DataProfiling]
    @DatabaseName = 'ML'
   ,@SchemaName = 'Titanic'
   ,@TableName ='Train'
   ,@TableView ='T'
   ,@Debug = 1
```

Jeżeli parametr `@Debug` zostanie ustawiony na wartość `1`, zostaną wyświetlone wszystkie wykonane zapytania. Zapytania te są tworzone dynamicznie. Najpierw są odczytywane metadane tabeli lub widoku, którego nazwa została przekazana do procedury `[DQ].[DataProfiling]`. Następnie jest tworzona lista kolumn tej tabeli lub widoku. Główne działanie procedury polega na dynamicznym konstruowaniu zapytań obliczających statystyki dla poszczególnych kolumn – to, jakie testy zostaną wykonane, zależy od typu kolumny; na przykład dla kolumn numerycznych jest wyliczana średnia arytmetyczna. Otrzymane wyniki są wstawiane do przedstawionych tabel, przy czym jeżeli zawierały już one wyniki wcześniejszych testów, zostaną zaktualizowane. Procedurę możemy wywoływać wielokrotnie dla tych samych tabel i widoków, aktualizując w ten sposób bazę statystyk.

Procedurę można też wykonać dla wszystkich tabel lub widoków wskazanego schematu bazy danych. Służy do tego procedura `[DQ].[DataProfilingAllObjects]`, której działanie polega na wywoływaniu w pętli

procedury `[DQ].[DataProfiling]`.

```
DECLARE @RC int
DECLARE @DatabaseName nvarchar(50)
DECLARE @SchemaName nvarchar(50)
DECLARE @TableView char(1)
DECLARE @Debug bit

EXECUTE @RC = [DQ].[DataProfilingAllObjects]
   @DatabaseName = 'ML'
   ,@SchemaName = 'Titanic'
   ,@TableView ='T'
   ,@Debug = 0
```

Zobaczmy, dla jakich tabel została wywołana procedura `[DQ].[DataProfiling]`.

```
SELECT d.DatabaseName, t.SchemaName, t.TableName, t.Rows, t.ProfileDurationS
FROM [DQ].[DataProfileDatabases] AS d
JOIN [DQ].[DataProfileTables] AS t
        ON d.DatabaseId = t.DatabaseId;
```

DatabaseName	SchemaName	TableName	Rows	ProfileDurationS
ML	Titanic	Train	891	0
ML	Titanic	Models	0	0
ML	Titanic	Test	418	0

Sprawdzone zostały dane ze wszystkich kolumn tych tabel.

```
SELECT t.TableName, c.ColumnName, c.SystemDataTypeId, c.DataLength, c.DataPrecision
FROM [DQ].[DataProfileTables] AS t
JOIN [DQ].[DataProfileColumns] AS c
       ON t.TableId = c.TableId;
```

TableName	ColumnName	SystemDataTypeId	DataLength	DataPrecision
Models	name	167	250	0
Test	Age	62	8	53
Test	Cabin	231	100	0
Test	Embarked	231	100	0
Test	Fare	62	8	53
Test	Name	231	200	0
Test	Parch	56	4	10
Test	PassengerId	56	4	10
Test	Pclass	56	4	10
Test	Sex	231	100	0
Test	SibSp	56	4	10
Test	Ticket	231	100	0
Train	Age	62	8	53
Train	Cabin	231	100	0
Train	Embarked	231	100	0
Train	Fare	62	8	53
Train	Name	231	200	0
Train	Parch	56	4	10
Train	PassengerId	56	4	10
Train	Pclass	56	4	10
Train	Sex	231	100	0
Train	SibSp	56	4	10
Train	Survived	56	4	10
Train	Ticket	231	100	0

Znacznie więcej wierszy (9335) zostało zapisanych do tabeli `[DQ].[DataProfiles]`. Większość z nich zawiera informacje o częstości występowania poszczególnych wartości zmiennych. Warto zwrócić uwagę, że wyniki testów dla kolumn różnych typów są zapisywane w osobnych kolumnach.

```
SELECT [TableName], [ColumnName], [Description], p.DataResult, p.NumericResult,
p.StringResult, p.Counts, p.RowsPercentage
FROM [DQ].[DataProfileTables] t
JOIN [DQ].[DataProfileColumns] c on t.TableId = c.TableId
JOIN [DQ].[DataProfiles] p on c.ColumnId = p.ColumnId
JOIN [DQ].[DataProfileStats] s on s.StatId = p.ProfileStatId
WHERE TableName = 'Train' and ColumnName = 'Age'
ORDER BY Description;
```

TableName	ColumnName	Description	DataResult	NumericResult	StringResult	Counts	RowsPercentage
Train	Age	Avg	NULL	29.69912	NULL	NULL	NULL
Train	Age	Emptyness	NULL	NULL	NULL	177	19.8653
Train	Age	Length distribution	NULL	1.00000	NULL	55	6.1728
Train	Age	Length distribution	NULL	4.00000	NULL	25	2.8058
Train	Age	Length distribution	NULL	NULL	NULL	177	19.8653
Train	Age	Length distribution	NULL	2.00000	NULL	634	71.1560
Train	Age	Max	NULL	80.00000	NULL	NULL	NULL
Train	Age	Min	NULL	0.42000	NULL	NULL	NULL
Train	Age	StdDev	NULL	14.52650	NULL	NULL	NULL
Train	Age	Uniqueness	NULL	NULL	No	802	9.9888
Train	Age	Values distribution	NULL	34.50000	NULL	1	0.1122
Train	Age	Values distribution	NULL	55.50000	NULL	1	0.1122
Train	Age	Values distribution	NULL	0.92000	NULL	1	0.1122

…

4.7. Wizualizacja zmiennych

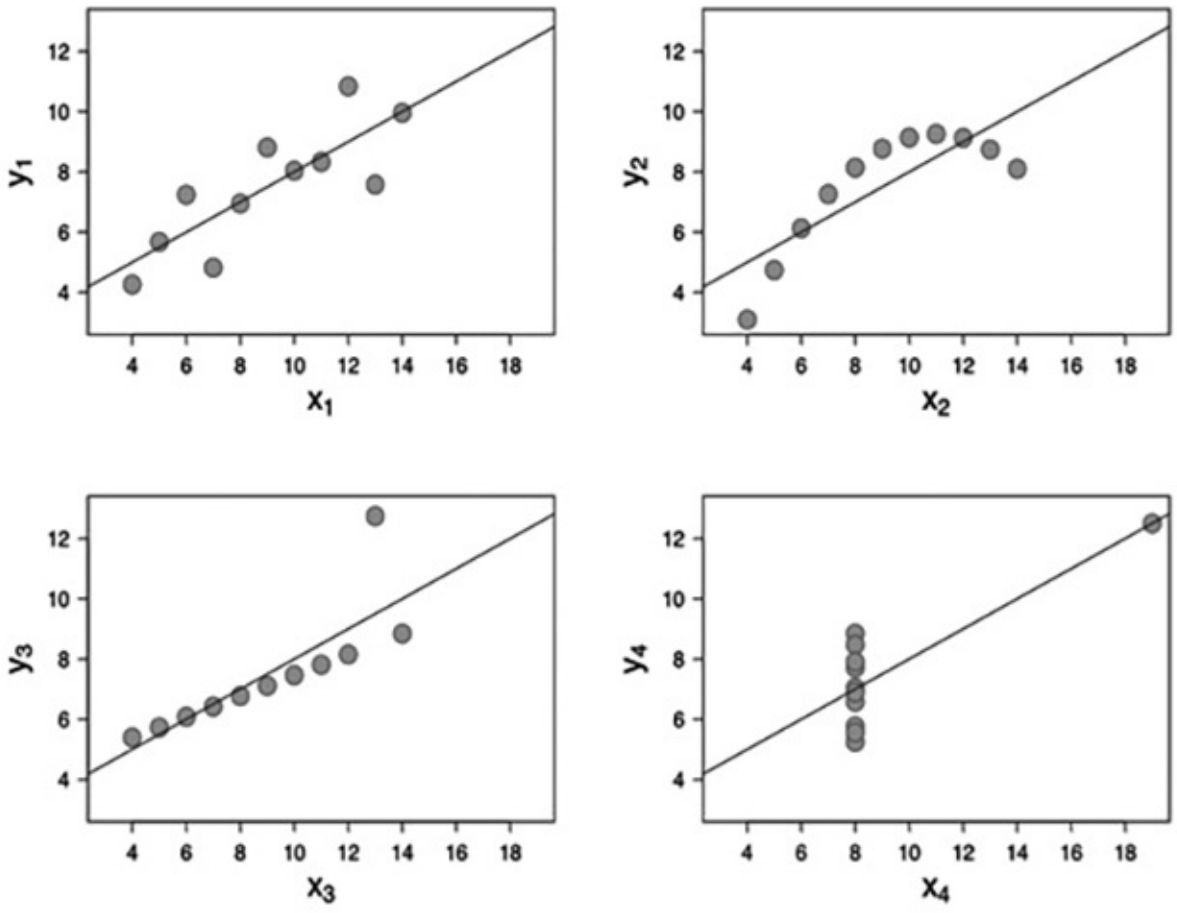
Poznawanie danych należy zacząć od obliczenia i przeanalizowania statystyk opisujących poszczególne zmienne. Jednak nawet najsumienniej policzone statystyki mogą wprowadzić nas w błąd. Nie dlatego, że ich interpretacja wymaga specjalistycznej wiedzy, ale ponieważ statystyki są znacznym uogólnieniem i nie zawierają kompletu informacji o danych. I tak jak wszystkie uogólnienia, statystyki sprawdzają się tylko w określonych przypadkach.

Problem ten zilustrował Francis Anscombe. Utworzył on cztery zbiory danych o prawie identycznych statystykach: średniej arytmetycznej zmiennej *X* równej 9

i zmiennej *Y* równej 7,5 oraz odchyleniu standardowym zmiennej *X* równemu 11 i zmiennej *Y* równej 4,125. Co więcej, model regresji liniowej dla wszystkich czterech zbiorów również był identyczny

$$y = 3 + 0{,}5x$$

a współczynnik determinacji R^2 dla każdego modelu regresji wyniósł 0,816 [21]. Wystarczy jednak przedstawić te cztery zbiory na wykresach punktowych, żeby zauważyć, jak bardzo się różnią (rys. 4.8).



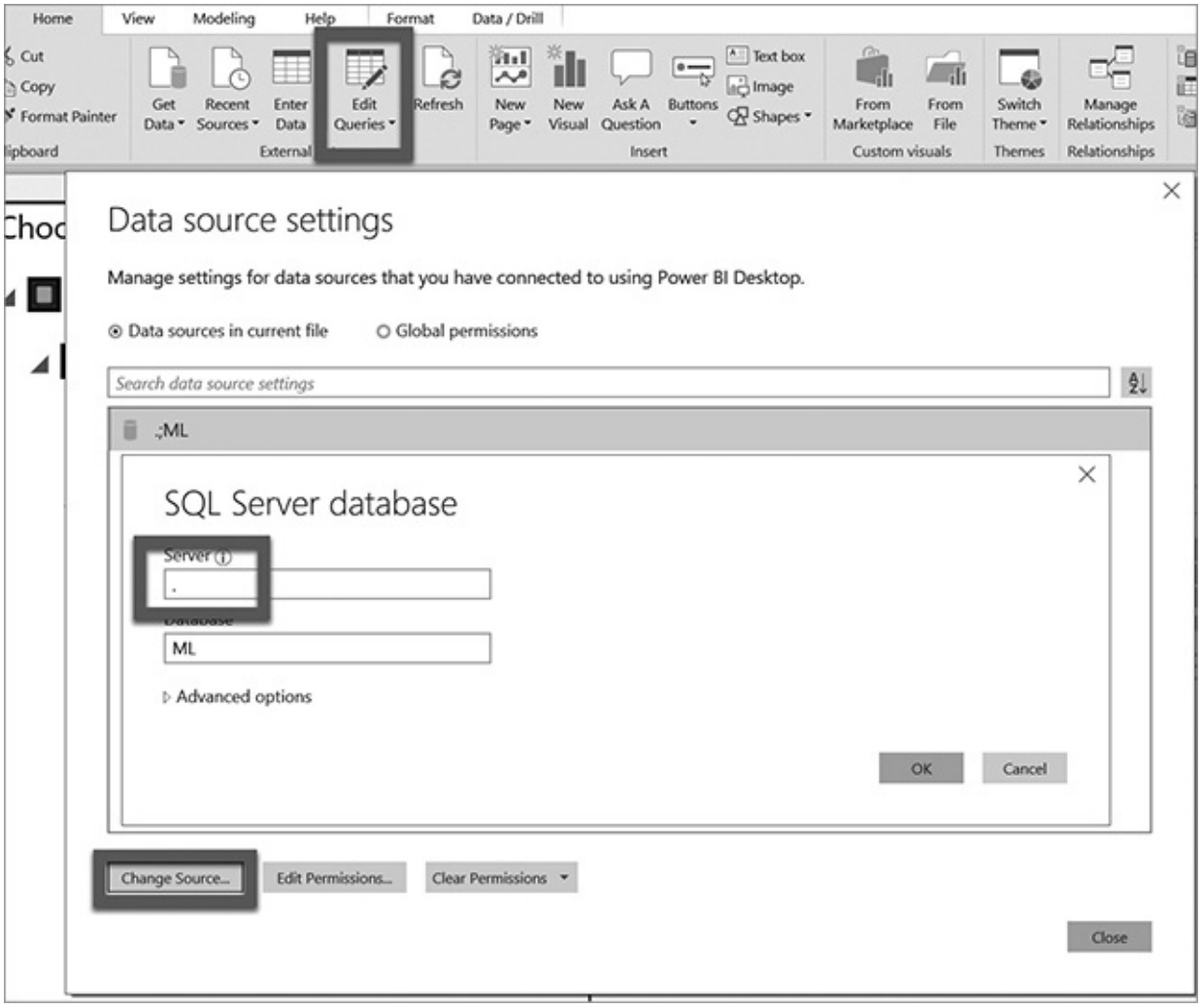
Rysunek 4.8. Kwartet Ascombe'a. Punkty przedstawiają przykładowe dane, linia prosta reprezentuje model regresji. Źródło: https://commons.wikimedia.org/w/index.php?curid=9838454

Osoby, które nadal ufają statystkom opisowym, powinny zapoznać się ze zbiorami danych The Datasaurus Dozen. Statystki opisowe danych ze wszystkich dwunastu zbiorów są identyczne, chociaż dane całkowicie się różnią [38].

Przedstawienie danych na wykresach jest tym ważniejsze, im rozkład zmiennych bardziej odbiega od rozkładu normalnego. W szczególności, rozkłady inne niż jednomodalne wymagają specjalnej uwagi przy ich opisywaniu za pomocą statystyk. Jeżeli na przykład rozkład jest dwumodalny i najczęściej występujące wartości są skrajne (np. *5* i *50*), wartość średnia i mediana nie będą miały żadnego znaczenia, bo nie będą reprezentowały typowego przypadku – w takim zbiorze mamy do czynienia z dwoma zupełnie różnymi „typowymi" przypadkami.

W niniejszej książce znajduje się kilkadziesiąt przykładowych wykresów, utworzonych w językach R i Python. Jednak w tym podrozdziale do graficznego przedstawienia danych użyłem programu Power BI Desktop. W ten sposób szybko i łatwo stworzyłem interaktywne raporty pozwalające dowolnie analizować zbiór danych o pasażerach Titanica.

Źródłem danych dla raportu *Variables.pbix* są uprzednio opisane tabele schematu `DQ`. Każda z nich została załadowana do raportu osobnym zapytaniem. Domyślnie, źródłem danych jest lokalny serwer SQL. Dane zapisane w pliku raportu możemy odświeżyć przyciskiem *Refresh*. Jeżeli po odświeżeniu raport nie zawiera danych, należy uruchomić opisaną uprzednio procedurę `[DQ].[DataProfilingAllObjects]`. Gdyby odświeżenie danych zakończyło się błędem, należy wskazać prawidłowe źródło danych (rys. 4.9).



Rysunek 4.9. Nowe źródło danych można wskazać, wybierając z menu Edit Queries polecenie Data source settings, a następnie polecenie Change Source. Kropka oznacza lokalną, domyślną instancję serwera SQL Server

Raport jest podzielony na dwie strony: na pierwszej znajdują się statystyki zmiennych numerycznych, na drugiej – kategorycznych. Odpowiednie wiersze tabel zostały wybrane za pomocą widocznych po prawej stronie filtrów stron. Na lewej stronie raportu została wstawiona niestandardowa, pobrana ze sklepu Microsoft, wizualizacja *HierarchySlicer*. Pozwala ona wybrać analizowaną zmienną.

W górnej części raportu została umieszczona tabela z informacjami na temat wybranej zmiennej. Poniżej znajdziemy dwie tabele z opisującymi tę zmienną

statystykami oraz wykres kolumnowy pokazujący rozkład długości wartości; dolną część raportu zajmuje histogram (rys. 4.10).



Rysunek 4.10. Statystyki opisowe uzupełnione histogramem pozwalają szybko ocenić poszczególne zmienne

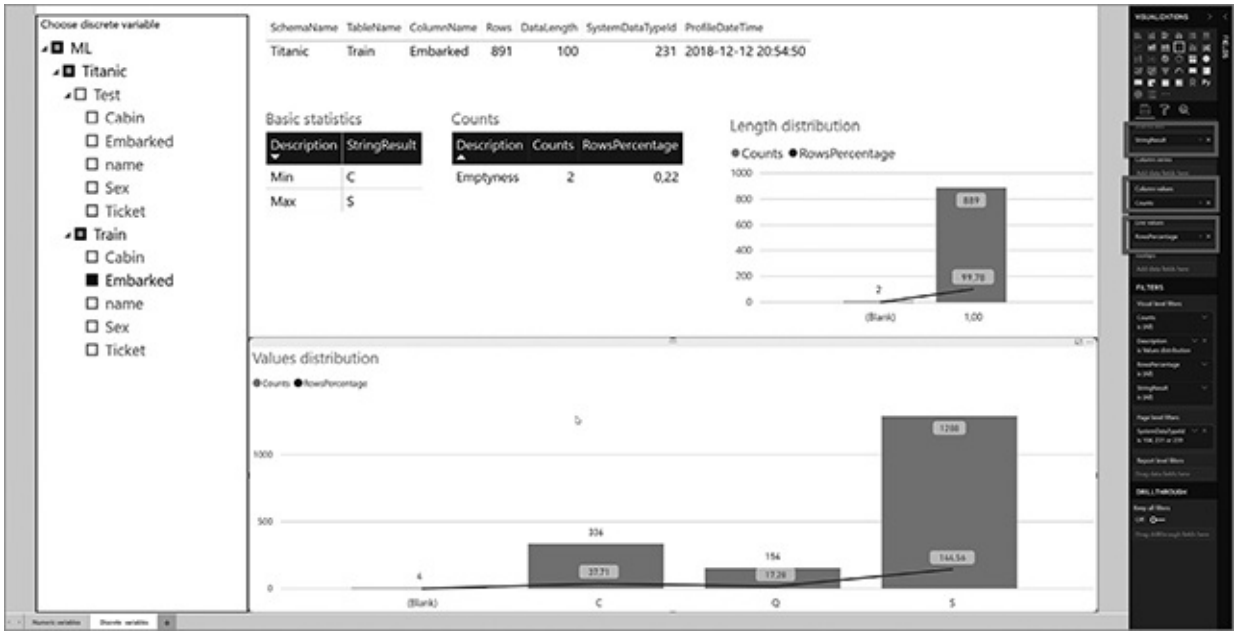
Analizując zmienne numeryczne, powinniśmy zauważyć, że:

- nie znamy wieku 177 pasażerów;
- niektóre bilety nic nie kosztowały, ale kilkanaście kosztowało ponad 100 dolarów;
- większość pasażerów podróżowała 3. klasą;
- większość pasażerów zginęła w katastrofie;
- zmienne `Age`, `SibSp` i `Parch` mają rozkład prawoskośny.

Takie same informacje znajdziemy na drugiej stronie raportu, poświęconej zmiennym kategorycznym. Jedyna różnica polega na zastąpieniu dolnego wykresu warstwowego kolumnowym. Każda wizualizacja zawiera odpowiednio wybrane dane (rys. 4.11).

Analizując zmienne kategoryczne, powinniśmy zauważyć, że:

- procent brakujących danych zmiennej `Cabin` przekracza 77%, co powoduje, że przydatność tej zmiennej jest wątpliwa;
- nie znamy portu zaokrętowania dwóch pasażerów, a najwięcej pasażerów zaokrętowało się w Southampton;
- numery niektórych biletów się powtarzają, czyli kilka osób (prawdopodobnie rodziny lub znajomi) podróżowało na tym samym bilecie;
- zmienna `name` zawiera, oprócz imienia i nazwiska, tytuł grzecznościowy; o ile imiona i nazwiska są nieprzydatne (ich wartości się nie powtarzają), to tytuły mogą wzbogacić nasz model.



Rysunek 4.11. Wygląd i działanie poszczególnych wizualizacji dostosowuje się do indywidualnych potrzeb w zakładce Format

4.8. Reprezentatywność danych

Głównym celem analizy zmiennych jest ocena ich przydatności. Statystyki

opisowe, informacje o brakujących danych, ilość informacji ukrytych w zmiennych oraz histogramy przedstawiające rozkłady wartości pozwalają na wczesnym etapie projektu wykryć i udokumentować problemy z danymi. Problemy z jakością danych występują zawsze, a niezauważone mogą doprowadzić do fiaska całego projektu albo co najmniej znacznie go skomplikować i wydłużyć. Im wcześniej zidentyfikujemy problemy z jakością danych, tym łatwiej i taniej będziemy mogli je rozwiązać.

Zdarza się, że wyniki analizy zmiennych wystarczają do podjęcia decyzji o ich wyeliminowaniu z danych treningowych. Ponieważ na tym etapie nasze zrozumienie danych jest jeszcze niepełne, powinniśmy usuwać tylko te zmienne, których brak przydatności nie budzi żadnych wątpliwości. Dotyczy to zmiennych przyjmujących niepowtarzające się wartości (w naszym przypadku imion i identyfikatorów pasażerów), zmiennych przyjmujących tylko jedną wartość (w zbiorze treningowym nie wystąpiła taka zmienna) i zmiennych z przeważająca ilością brakujących danych (takich jak zmienna `Cabin`).

Wyniki analizy pozostałych zmiennych posłużą nam do wyboru właściwych technik poprawy ich jakości.

Najważniejszą korzyścią z analizy zmiennych jest jednak możliwość oceny ich reprezentatywności. Znając tendencję centralną, rozproszenie i rozkład wartości, możemy ocenić, czy są one zgodne z oczekiwanymi. Ta ocena powinna być efektem dyskusji z ekspertem z danej dziedziny. Gdyby się okazało, że niektóre wartości są mało prawdopodobne, w następnym etapie je wyczyścimy.

Ocena reprezentatywności danych ma zasadniczy wpływ na powodzenie całego projektu uczenia maszynowego. W praktyce będziemy dysponować próbą danych. Nawet omawiany w tym rozdziale przykład nie jest wyjątkiem od tej reguły – zbiór danych o pasażach Titanica jest niekompletny i nie zawiera danych o członkach załogi. Zanim użyjemy posiadanej próby danych do

treningu modeli, musimy się upewnić, że właściwie reprezentuje ona całą populację. W przeciwnym razie model nauczy się fikcyjnych, obecnych w próbie, ale niewystępujących w rzeczywistości, zależności między zmiennymi.

Wbrew powszechnemu poglądowi większa próba (próba licząca więcej przykładów) nie zawsze jest bardziej reprezentatywna. W statystyce obowiązuje reguła mówiąca, że jeżeli przykłady były zebrane w specyficznych warunkach, wielkość próby nie powinna przekraczać 10% populacji. Wynika to z tego, że więcej przykładów będzie dokładniej odzwierciedlać lokalną charakterystykę miejsca lub czasu zbierania danych. Na przykład wyniki badań genetycznych prowadzonych na pacjentach jednego szpitala staną się specyficzne dla miasta, w którym znajduje się szpital, i ich uogólnienie na całą populację będzie nieuzasadnione.

Postulat reprezentatywności próby najłatwiej spełnić, losowo wybierając przykłady. Oznacza to, że każdy przypadek powinien mieć taką samą, niezerową szansę wybrania do próby. Do najczęstszych błędów popełnianych przy wyborze próby należą: preferowanie łatwiej dostępnych obserwacji (np. pacjentów z jednego szpitala) oraz nieuwzględnienie brakujących obserwacji (np. faktu, że pewne urządzenia nie przesłały danych albo część osób wytypowanych do udziału w ankiecie nie odpowiedziała na pytania). Dlatego oceniając reprezentatywność próby, należy uzyskać odpowiedzi na poniższe pytania:

- Jak zostały wybrane do niej przykłady, czyli jakimi kryteriami kierowano się przy ich wyborze?
- Czy wybrane przykłady są w jakiś sposób wyjątkowe?
- Czy osoba wybierająca przykłady miała jakiekolwiek, świadome lub nieświadome, preferencje przy wyborze przykładów?

Szczególnie trudne jest trzecie pytanie. Dlatego nawet największe i mające

wieloletnie doświadczenie w prowadzeniu projektów uczenia maszynowego firmy, takie jak Microsoft, nie ustrzegły się przed tym problemem. Mam na myśli używanie do treningu modeli próby niereprezentatywnej, obciążanej błędem systematycznym (*bias*). Dobrym przykładem jest historia bota Tay. W marcu 2016 r. Microsoft podłączył do Twittera bota zdolnego do prowadzenia prostych konwersacji i uczenia się od swoich rozmówców. W ciągu 24 godzin z super pozytywnie nastawionego do ludzi i świata bot zmienił się w nienawidzącego feministek faszystę. Pokazuje to, że drobne, często niezauważane przez nas uprzedzenia błyskawicznie potęgują się w modelach uczenia maszynowego.

Innym przykładem może być stworzony przez Amazon model do automatycznego oceniania życiorysów zawodowych. W teorii nauczony na życiorysach wysłanych w ciągu ostatnich 10 lat do firmy Amazon model powinien oceniać (w skali od 1 do 5) kandydatów na podstawie ich doświadczenia, wykształcenia itd. W praktyce okazało się, że oceniał je przede wszystkim na podstawie płci, dyskryminując kobiety. Powodem tego skrzywienia był nieznacznie większy procent życiorysów mężczyzn niż kobiet w danych treningowych oraz to, że mężczyźni nieco częściej dostawali pracę, o którą się ubiegali. Nauczony na takich danych model okazał się seksistą.

Jeżeli jednak próba nie jest obciążona, większa liczba przykładów często prowadzi do lepszych wyników. W szczególności zwiększenie liczby przykładów może zmniejszyć błąd rzeczywisty (błąd dla danych testowych). Zawdzięczamy to zjawisku zbieżności do rzeczywistego rozkładu. Polega ono na tym, że wraz ze zwiększaniem wielkości próby rozkład wartości zmiennych się zmienia aż do momentu, w którym będzie on taki sam jak rozkład całej populacji. Czyli większe próby, pod warunkiem że zostały prawidłowo zebrane, powinny być bardziej reprezentatywne. Podsumowując, im więcej przykładów, tym mniejsze ryzyko nadmiernego dopasowania modelu.

4.9.
Korelacje między zmiennymi

Z lektury poprzedniego rozdziału wiemy, że uczenie maszynowe polega na szukaniu zależności między zmiennymi wejściowymi a wyjściową. Analiza poszczególnych zmiennych jest tylko wstępnym, ale niezbędnym etapem poznawania danych. Drugim, jeszcze ważniejszym etapem, jest analiza korelacji między zmiennymi.

Pod pojęciem korelacji rozumiemy statystyczną zależność między zmiennymi. **Dwie zmienne są ze sobą skorelowane, jeżeli znajomość wartości jednej zmiennej pomaga, choćby w niewielkim stopniu, w określeniu wartości drugiej zmiennej**. Im silniejsza korelacja, tym lepiej potrafimy przewidzieć wartość jednej zmiennej na podstawie znajomości wartości drugiej.

Zmienną wyjściową, inaczej objaśnianą, nazywa się też zależną. Oznacza to, że zmienna wyjściowa powinna być skorelowana ze zmiennymi wejściowymi. Zmienne wejściowe w żaden sposób, bezpośrednio lub pośrednio, nieskorelowane z wyjściową należy usunąć z danych treningowych.

Usuwanie nieskorelowanych zmiennych jest skutecznym sposobem upraszczania modeli i prowadzi do poprawy jakości modeli oraz skrócenia czasu ich treningu. Z tego powodu z danych treningowych usuwa się zmienne przyjmujące tylko jedną wartość oraz zmienne o znacznej większości brakujących wartości.

Powód usuwania zmiennych o niepowtarzających się wartościach, takich jak `name`, jest zupełnie inny. Naszym celem jest zminimalizowanie błędu rzeczywistego (błędu predykcji dla danych testowych). Dane testowe są z założenia różne od treningowych (zbiory danych testowych i treningowych muszą być rozłączne). Skoro wartości zmiennej nie powtórzyły się ani razu w danych treningowych, z dużym prawdopodobieństwem możemy przyjąć, że te same wartości nie wystąpią ani razu w danych testowych. Innymi słowy, należy

przypuszczać, że w danych testowych nie znajdziemy ani Miss Elizabeth Bonnell, ani Mr. Hugh Woolner, ani żadnego innego imienia występującego w danych treningowych. W takim razie zależność funkcyjna istniejąca między zmienną wejściową a wyjściową[6] przeszkadza, a nie pomaga, w zmniejszeniu błędu testowego. Każda zmienna o niepowtarzających się wartościach pozwala w łatwy sposób zmniejszyć do zera błąd treningowy (znając imię pasażera, model uczenia maszynowego ze 100% dokładnością będzie potrafił powiedzieć, czy przeżyje on katastrofę, czy nie), ale jednocześnie może znacząco pogorszyć błąd testowy (zamiast szukać zależności m.in. zmiennymi wejściowymi a wyjściową, model jedynie nauczy się listy nazwisk pasażerów).

 Zmienne, których wartości nie można uogólnić, na przykład imiona i nazwiska, numery PESEL, numery faktur czy identyfikatory urządzeń, nie mają żadnej wartości dla modeli uczenia maszynowego. Co gorsza, jeżeli ich nie usuniemy, nadmiernie dopasujemy model do danych treningowych.

Czemu upraszczanie modelu przez eliminowanie nieprzydanych zmiennych wejściowych jest takie ważne? Otóż **wraz z każdą dodatkową zmienną zmiennych gwałtownie zwiększa się liczba możliwych kombinacji ich wartości**. I tak, 5 zmiennych można zestawić ze sobą na 26 sposobów, 6 – na 120 sposobów, a 10 – już na 1013 sposobów. Liczba możliwych kombinacji zbioru *n* zmiennych jest równa sumie liczby kombinacji 2-elementowych, 3-elementowych, …, *n*-elementowych (jednoelementowych kombinacji nie uwzględniamy, bo korelacja zmiennej z nią samą nas nie interesuje). Na przykład liczbę możliwych kombinacji 5 zmiennych policzymy ze wzoru

$$C = \binom{5}{2} + \binom{5}{3} + \binom{5}{4} + \binom{5}{5}$$

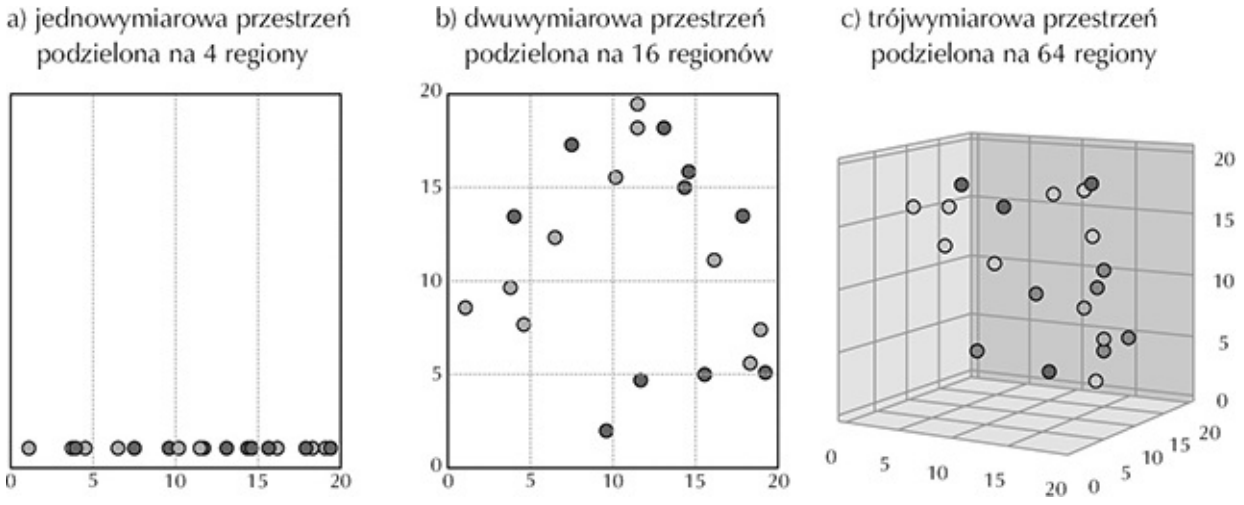
Związany z tym problem nazywa się **klątwą wymiarowości**.

4.9.1. Klątwa wymiarowości

Wyobraźmy sobie zmienne jako wymiary. Jedna zmienna oznacza, że mamy do

czynienia z przestrzenią jednowymiarową, dwie zmienne tworzą przestrzeń dwuwymiarową, trzy – trójwymiarową itd. Umieśćmy teraz w tych przestrzeniach 20 przykładów. Dla ułatwienia przyjmijmy też, że wszystkie zmienne przyjmują wartości całkowite z zakresu od 0 do 20. W przestrzeni jednowymiarowej punkty reprezentujące przykłady zostaną umieszczone na linii, a ich pozycja będzie wyznaczona przez wartość zmiennej *X*. Dodajmy wymiar zmiennej *Y*. Teraz nasze punkty – przykłady będą się znajdować na płaszczyźnie, w miejscu wyznaczonym przez parę wartości (*X*,*Y*). Zauważmy, że rozmiar naszej przestrzeni zwiększył się z 20 do 400 ($20^2 = 400$). Spowodowało to zwiększenie odległości między punktami. Po dodaniu trzeciej zmiennej pozycję punktów w trójwymiarowej przestrzeni będą wyznaczać trzy współrzędne (*X*,*Y*,*Z*). Rozmiar przestrzeni wynosi już 8000 ($20^3 = 8000$), a odległości między punktami są znacznie większe. Dziesięciowymiarowa przestrzeń będzie miała powierzchnię wielkości 10 240 000 000 000 (dziesięciu bilionów dwustu czterdziestu miliardów punktów).

Pasażerowie Titanica są opisani 12 zmiennymi. Spróbujmy wyobrazić sobie dwunastowymiarową przestrzeń, a w niej 891 punktów. Jaka będzie średnia odległość między nimi? Pomoże nam w tym podzielenie przestrzeni na obszary stałej wielkości, na przykład 5 jednostek (rys. 4.12).



Rysunek 4.12. Liczba regionów zwiększa się wykładniczo wraz z dodawaniem kolejnych wymiarów.

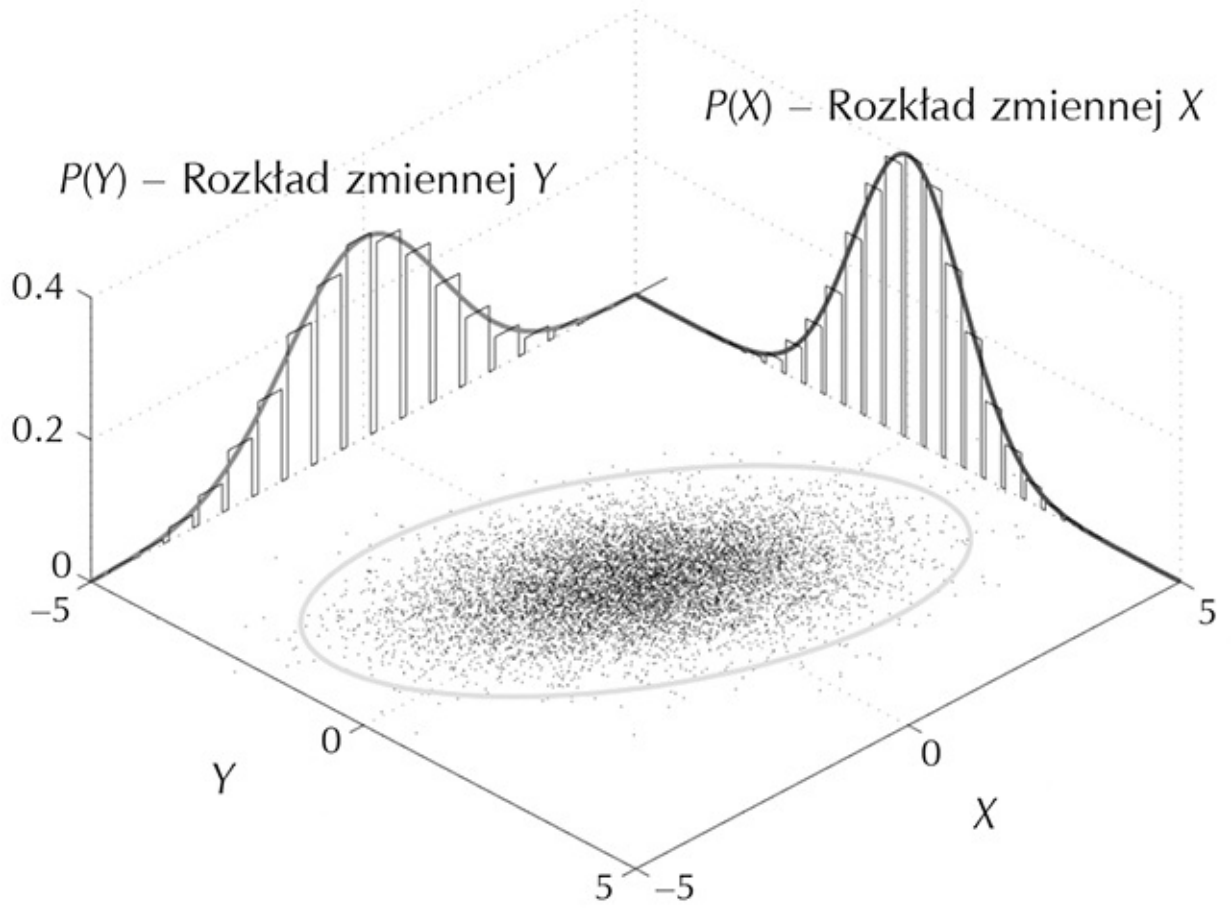
W dodatku, im więcej wymiarów, tym więcej pustych regionów – w przestrzeni jednowymiarowej każdy region zawierał jakieś przykłady, w przestrzeni dwuwymiarowej trzy regiony były puste, a w przestrzeni trójwymiarowej większość regionów nie zawiera już żadnych przykładów

Wraz ze wzrostem liczby zmiennych (wymiarów) ilość danych potrzebnych do stworzenia reprezentatywnej próby zwiększa się wykładniczo – to właśnie klątwa wymiarowości. Wymóg wykładniczego zwiększania liczby przykładów jest poważnym problemem. Po pierwsze, możemy nie dysponować tak dużą próbą. Po drugie, koszt obliczeniowy algorytmów eksploracji danych zależy co najmniej liniowo od liczby przykładów treningowych, a często jest znacznie wyższy.

Klątwa wymiarowości jest drugim, po nadmiernym dopasowaniu, najczęstszym powodem niepowodzeń projektów uczenia maszynowego. Wynika to z tego, że problemy łatwe do rozwiązania w przestrzeniach o małej liczbie wymiarów stają się trudne w przestrzeniach wielowymiarowych. Oto przykład. Weźmy dwadzieścia równooddalonych od siebie punktów. W przestrzeni jednowymiarowej (na linii) każdy z tych punktów będzie miał dwóch najbliższych, równo od niego oddalonych sąsiadów. W przestrzeni dwuwymiarowej (na płaszczyźnie) liczba najbliższych sąsiadów każdego punktu zwiększa się do czterech. W przestrzeni trójwymiarowej – do sześciu. Dzieje się tak dlatego, że wraz ze wzrostem liczby wymiarów zwiększa się liczba współrzędnych punktów, a liczba najbliższych sąsiadów, jeżeli te punkty są równo od siebie oddalone, wynosi dwukrotność liczby współrzędnych. Czyli w przestrzeni trzydziestowymiarowej każdy punkt będzie miał już sześćdziesięciu najbliższych sąsiadów, dziesięciokrotnie więcej niż miał w przestrzeni trójwymiarowej. Dla algorytmów uczenia maszynowego stanowi to poważny problem, szczególnie dla algorytmów bazujących na porównywaniu odległości między przykładami, takich jak algorytm *k* najbliższych sąsiadów czy algorytm klastrowania metodą *k* średnich.

Ponieważ od dziecka żyjemy w świecie trójwymiarowym, nasze doświadczenia

i zbudowana na nich intuicja dotyczą przestrzeni o małej liczbie wymiarów. Nie mamy kłopotu z wyobrażeniem sobie jednowymiarowych linii, dwuwymiarowych powierzchni i trójwymiarowych brył. Jednak mało kto potrafi sobie wyobrazić przestrzeń siedmio- albo czterdziestowymiarową. Dlatego za wielowymiarowe uważa się przestrzenie więcej niż trójwymiarowe. Próbujemy stosować reguły obowiązujące w przestrzeniach o liczbie wymiarów mniejszej niż cztery do przestrzeni wielowymiarowych; niestety, przestrzenie o dużej liczbie wymiarów rządzą się własnymi prawami. Weźmy najprostszy przykład – dobrze nam znany rozkład normalny. Na rysunku 4.2 przedstawiono rozkład normalny pojedynczej zmiennej. Spójrzmy na niego, zwracając uwagę, że większość przykładów znajduje się w okolicy centrum. Wiemy, że 68% punktów jest umieszczonych na linii w odległości nieprzekraczającej jednego odchylenia standardowego od średniej. Porównajmy ten rysunek z rysunkiem 4.13 przedstawiającym rozkład normalny dwóch zmiennych.



Rysunek 4.13. Rozkład normalny dwóch zmiennych. Zwróćmy uwagę, że większa cześć wyznaczonych przez osie linii zawiera punkty, chociaż większość wartości koncentruje się wokół punktu centralnego. Tymczasem ponad połowa płaszczyzny jest pusta, a obszar, w którym znajduje się większość punktów, stał się mniejszy

W przestrzeni pięciowymiarowej nie tylko punkty byłyby dalej od siebie (przestrzeń byłaby rzadko wypełniona wartościami), lecz także dodatkowo obszar ich skupienia, na przykład obszar, w którym prawdopodobieństwo wystąpienia wartości wynosiłoby 68%, byłby malutki, ledwie widoczny na pięciowymiarowym wykresie. Ten fenomen można udowodnić matematycznie. Maksymalna odległość między punktami reprezentującymi zmienne losowe o standardowym rozkładzie normalnym w *n*-wymiarowej przestrzeni wynosi

$$\mathrm{Max}D = \sqrt{n}$$

natomiast średnia odległość między tymi punktami jest równa

$$\text{Avg}D = \frac{\sqrt{n}}{3}$$

czyli w przestrzeni trójwymiarowej maksymalna odległość wynosi 3, a średnia to 1,73. Wraz ze wzrostem liczby wymiarów maksymalna odległość między punktami zwiększa się trzykrotnie szybciej niż średnia odległość. Natomiast wariancja zmiennej jest stała, równa

$$\text{Var} = \frac{1}{\sqrt{18}}$$

i nie zależy od liczby wymiarów.

4.9.2. Ocena przydatności zmiennych

Znając klątwę wymiarowości, wróćmy do rozważań o liczbie możliwych kombinacji zmiennych. Im jest ona większa, tym większe ryzyko wystąpienia przypadkowych, występujących tylko w zbiorze treningowym kombinacji. Na przykład 18-letni mężczyźni, którzy zaokrętowali się w Cherbourg, podróżowali pierwszą klasą bez rodziców i rodzeństwa, i zapłacili za bilet 80 funtów, przeżyli katastrofę. Czyli **duża liczba zmiennych wejściowych zwiększa ryzyko nadmiernego dopasowania modelu**.

Naszym zadaniem jest wybranie najbardziej przydatnych zmiennych. W jego rozwiązaniu pomoże nam sprawdzenie korelacji między zmiennymi. Przydatne będą zmienne silniej skorelowane ze zmienną wyjściową.

Zmienne wejściowe, inaczej objaśniające, nazywa się też niezależnymi. Oznacza to, że zmienne wejściowe nie powinny być ze sobą skorelowane. Jedną z silnie skorelowanych ze sobą zmiennych wejściowych należy usunąć z danych treningowych.

Korelację możemy sprawdzić, wykonując testy statystyczne odpowiednie dla

ocenianych zmiennych albo wizualizując te zmienne. Wynikiem testów statystycznych jest współczynnik korelacji, czyli liczba określająca stopień i kierunek zależności między zmiennymi. Zwykle jest on normalizowany do zakresu od –1 do 1, gdzie –1 oznacza maksymalną korelację ujemną (wraz ze wzrostem wartości jednej zmiennej wartości drugiej zmiennej maleją), 0 oznacza brak korelacji, a 1 maksymalną korelację dodatnią (wraz ze wzrostem wartości jednej zmiennej wartości drugiej zmiennej rosną).

Według popularnej reguły korelacje o współczynniku mniejszym niż 0,3 uznaje się za statystycznie nieistotne, korelacje o współczynniku między 0,3 a 0,6 – za średnio istotne, a o współczynniku większym niż 0,6 – za silne. W praktyce oceniając siłę korelacji, należy brać pod uwagę kontekst. Na przykład korelacja o współczynniku 0,4 między płcią a przeżyciem katastrofy powinna zostać uznana za istotną, a korelacja o współczynniku 0,8 między wskazaniami czujnika poziomu paliwa a ilością paliwa w baku za zaskakująco słabą.

4.9.3. Dwie zmienne numeryczne

Najpopularniejszym współczynnikiem korelacji zmiennych numerycznych jest **współczynnik korelacji liniowej r-Pearsona**. Jest to test parametryczny polegający na podzieleniu kowariancji zmiennych przez iloczyn ich średnich:

$$r_{XY} = \frac{cov(X,Y)}{\sigma X * \sigma Y}$$

Użyta we wzorze kowariancja jest miarą wspólnej zmienności dwóch zmiennych. Jeżeli większe wartości jednej zmiennej występują razem z większymi wartościami drugiej, mamy do czynienia z dodatnią kowariancją. Jeżeli większe wartości jednej zmiennej częściej występują z mniejszymi wartościami drugiej – z kowariancją ujemną. Kowariancje można wyznaczyć na podstawie różnicy między rzeczywistym a spodziewanym

prawdopodobieństwem wystąpienia wartości:

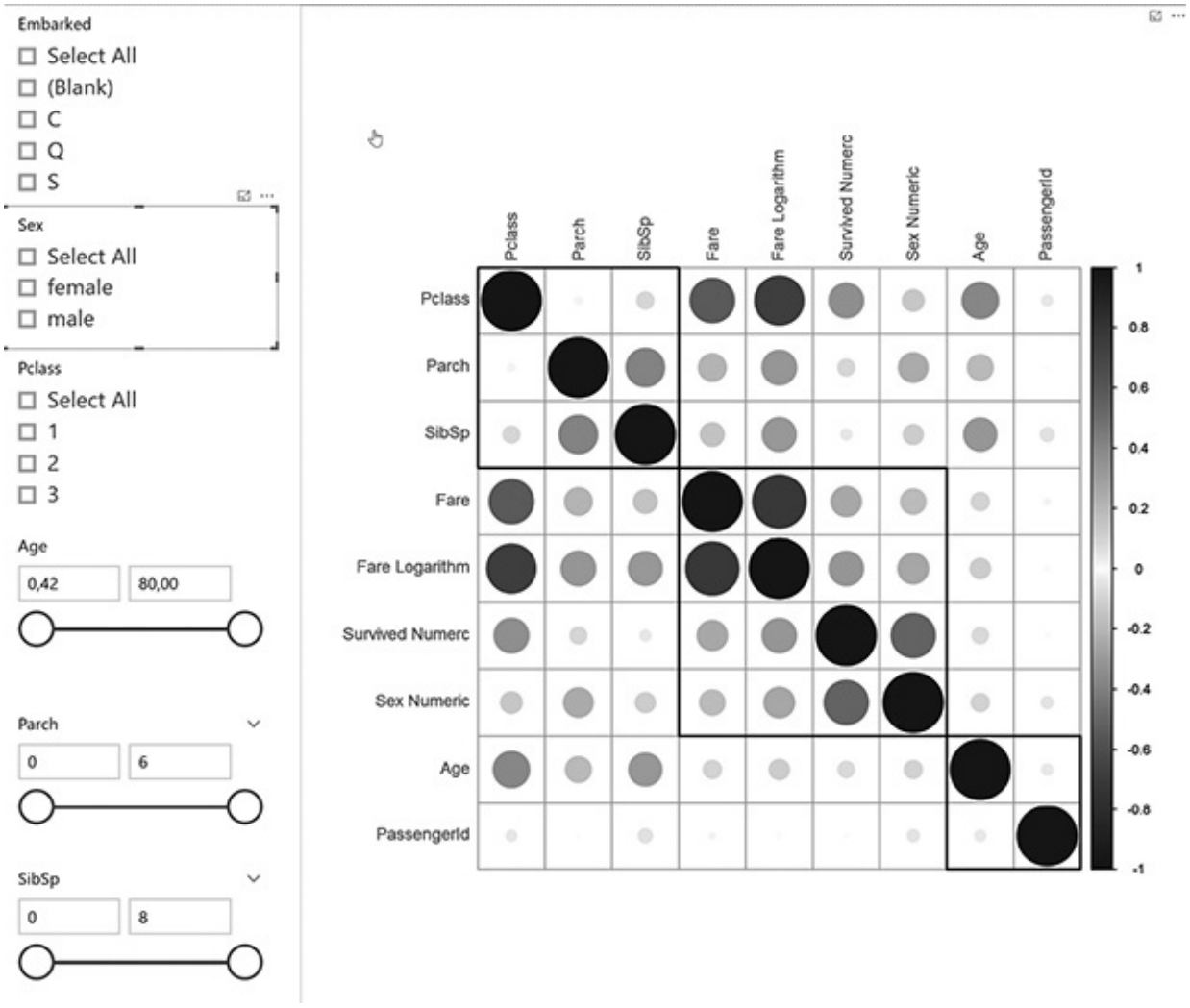
$$cov(X,Y) = SUM((X_i - \bar{x}\ (X))^*\ (Y_i - \bar{x}(Y))\ ^*\ P(X_i,Y_i))$$

Dzieląc kowariancję przez iloczyn średnich arytmetycznych, otrzymamy współczynnik korelacji liniowej r. Współczynnik korelacji linowej określa siłę korelacji zmiennych o rozkładzie zbliżonym do normalnego, bo do jego wyznaczenia wykorzystujemy parametry modelu normalnego: średnie i odchylenia standardowe. Należy też pamiętać, że *r* jest miarą związku liniowego – jeśli nie ma korelacji liniowej, to wartość współczynnika *r* będzie bliska zeru, ale nie wyklucza to istnienia silnej nieliniowej korelacji między zmiennymi. Ponadto powinniśmy zdawać sobie sprawę z podatności współczynnika *r* na wartości odstające. Prostym sposobem obliczenia współczynnika *r* jest skonwertowanie danych do tabeli Pandas i wywołanie metody `corr`.

```
df = pd.DataFrame(data=data)
print(df.corr())
```

	PassengerId	Age	SibSp	Parch	Fare
PassengerId	1.000000	0.036847	-0.057527	-0.001652	0.012658
Age	0.036847	1.000000	-0.308247	-0.189119	0.096067
SibSp	-0.057527	-0.308247	1.000000	0.414838	0.159651
Parch	-0.001652	-0.189119	0.414838	1.000000	0.216225
Fare	0.012658	0.096067	0.159651	0.216225	1.000000

Współczynnik *r* można też wyliczyć i graficznie przedstawić przy użyciu niestandardowej wizualizacji programu Power BI Desktop o nazwie *Correlation plot*. Wizualizację tę, tak jak wiele innych przydatnych wizualizacji, możemy zaimportować do raportu Power BI Desktop, klikając przycisk *Import a custom visual*. Wizualizacja *Correlation plot* została napisana w języku R, a więc wymaga wskazania programowi Power BI Desktop lokalizacji lokalnie zainstalowanego interpretera języka R (rys. 4.14).



Rysunek 4.14. Aby umożliwić obliczenie współczynnika *r*, niektóre zmienne, takie jak `Survived`, zostały przekształcone na zmienne numeryczne. Wizualizacja pochodzi z raportu *Correlations.pbix* znajdującego się w folderze *Chapter04* repozytorium z przykładami do książki

Wielkość i intensywność tła punktu zależy od bezwzględnej wartości współczynnika korelacji *r* – im jest ona większa, tym większy i ciemniejszy punkt. Natomiast kierunek korelacji jest oznaczony jasnością. Jasny punkt oznacza negatywną, a ciemny pozytywną korelację. Na wykresie możemy zobaczyć, że:

- zmienna `Survived` jest liniowo skorelowana ze zmiennymi `Sex`, `Pclass`, `Fare` i `Age`; korelacja ze zmienną `Pclass` jest negatywna;
- zmienna `Survived` nie jest liniowo skorelowana ze zmiennymi `Parch`, `SibSp`

i `PassangerID`;

- zmienne `Age`, `Pclass`, `Parch`, `SibSp` są ze sobą liniowo skorelowane.

Pamiętajmy że współczynnik korelacji liniowej r-Pearsona sprawdza wyłącznie siłę korelacji liniowej. Dlatego warto sprawdzić, jak wygląda on dla podzbiorów danych, na przykład tylko mężczyzn podróżujących drugą klasą, którzy zaokrętowali się w Queenstown. Jeżeli wyniki otrzymane dla różnych podgrup będą się różniły, test należy odrzucić jako niewiarygodny. Czyli że jego wynik nie musi reprezentować rzeczywistej korelacji między zmiennymi. Nie zapominajmy też o założeniu rozkładu normalnego danych – obliczony dla zmiennych o innym rozkładzie współczynnik korelacji r jest niewiarygodny.

4.9.4. Dwie zmienne kategoryczne

Korelacje między zmiennymi kategorycznymi można obliczyć na podstawie tabel częstości. Tym razem jednak interesuje nas warunkowy rozkład zmiennych, a więc policzymy, jak często występowały wartości *X* dla poszczególnych wartości *Y*.

```
survived_sex = pd.crosstab(index=data["Survived"],columns=data["Sex"])
survived_sex.index= ["died","survived"]
print(survived_sex)
```

```
        Survived
Sex         0    1
female     81   233
male      468   109
```

Z 342 ocalonych rozbitków 233 to kobiety, a 109 – mężczyźni. Proporcje płci są zupełnie inne wśród ofiar katastrofy – z 549 ofiar 468 to mężczyźni, a 81 – kobiety. Dane te możemy porównać ze spodziewanymi.

Hipoteza zerowa zakłada brak zależności między wierszami a kolumnami tabeli

częstości. Przyjmując tę hipotezę, obliczymy wartości teoretyczne (częstości które wystąpiłyby, gdyby zmienne były niezależne). Jeżeli zaobserwowane dane będą się różniły od spodziewanych, hipotezę zerową odrzucimy i uznamy zależność między wierszami a kolumnami tabeli częstości. Będzie to oznaczało istnienie korelacji między analizowanymi zmiennymi.

Test chi-kwadrat (**χ-kwadrat**) polega na zsumowaniu różnić między obserwowanymi i spodziewanymi wartościami. Otrzymany wynik należy sprawdzić w tablicach rozkładu chi-kwadrat. Zawierają one wartości punktów krytycznych dla określonych stopni swobody DF. Stopień swobody obliczymy ze wzoru

$$DF = (\textit{liczba kolumn} - 1) * (\textit{liczba wierszy} - 1).$$

Możemy też użyć odpowiedniej funkcji języków R lub Python.

```
chi2, p, dof, expected = chi2_contingency(pd.crosstab(data.Sex, data.Survived))
print('Statistics=%.3f, p=%.70f, dof=%.3f' % (chi2, p , dof))
alpha = 0.05
if p > alpha:
     print('Variables seems to be independent (fail to reject H0)')
else:
     print('Variables seems to be dependent (reject H0)')
print(expected)
```

```
Statistics=260.717,
p=0.0000000000000000000000000000000000000000000000000000000011197357062775,
dof=1.000
Variables look dependent (reject H0)
[[ 193.47474747  120.52525253]
 [ 355.52525253  221.47474747]]
```

Dla zmiennych `Sex` i `Survived` wartość chi-kwadrat wynosi 260,7. Przy 1. stopniu swobody prawdopodobieństwo, że zmienne te są niezależne wynosi $1{,}2^{-58}$, czyli jest znacznie mniejsze od typowego punktu krytycznego $p = 0{,}05$. Należy więc odrzucić hipotezę zerową i zmienne uznać za skorelowane. Gdyby

zmienne były niezależne, uratowałoby się 120 kobiet i 221 mężczyzn.

4.9.5.
Dwie zmienne porządkowe

Do oceny korelacji między kategorycznymi zmiennymi porządkowymi można użyć **testu rho-Spearmana**. Jest to test nieparametryczny, czyli niewymagający określonego rozkładu ocenianych zmiennych. Zakłada on jednak, że rangi są liczbami całkowitymi, czyli powinien być używany do oceny korelacji zmiennych porządkowych mierzonych na „długich” skalach.

```
coef, p = st.spearmanr(data.Survived, data.Pclass)
print('Statistics=%.3f, p=%.30f' % (coef, p))
alpha = 0.05
if p > alpha:
     print('Variables seems to be independent (fail to reject H0)')
else:
     print('Variables seems to be dependent (reject H0)')
```

```
Statistics=-0.340, p=0.000000000000000000000000168761
Variables seems to be dependent (reject H0)
```

Jeśli skale są „krótkie”, to występuje wiele rang wiązanych i wtedy bardziej odpowiednim współczynnikiem jest tau-Kendalla.

```
coef, p = st.kendalltau(data.Survived, data.Pclass)
print('Statistics=%.3f, p=%.50f' % (coef, p))
alpha = 0.05
if p > alpha:
     print('Variables seems to be independent (fail to reject H0)')
else:
     print('Variables seems to be dependent (reject H0)')
```

```
Statistics=-0.324, p=0.00000000000000000000000000000000000000000002244
Variables seems to be dependent (reject H0)
```

4.9.6.

Zmienna kategoryczna i numeryczna

Korelację między zmienną numeryczną a kategoryczną można ocenić, przeprowadzając **analizę wariancji ANOVA**. Polega ona na znalezieniu stosunku wariancji obliczonej między badanymi grupami a średnią wariancją zaobserwowaną wewnątrz grup. Hipoteza zerowa zakłada, że jedynym źródłem wariancji jest wariancja zaobserwowana wewnątrz grup, czyli że wariancja między grupami ma zerowy wpływ. Do weryfikacji hipotezy należy obliczyć średnie kwadraty

$$MS_A = \frac{\sum_{i=1}^{a} *(\mu_i - \mu)^2}{DF}, \text{ gdzie } DF = a - 1$$

Stosunek średnich kwadratów obliczanych dla poszczególnych stanów zmiennej kategorycznej w odniesieniu do średniego kwadratu błędu pozwala oszacować wpływ każdego z tych stanów oddzielnie na wartości zmiennej numerycznej.

Jednoczynnikowa analiza wariancji pozwala ocenić wpływ poszczególnych stanów zmiennej kategorycznej na zmienną numeryczną:

$$MS_E = \frac{\sum_{i=1}^{a} \sum_{j=1}^{ni} *(\mu_i - \mu)^2}{DF}, \text{ gdzie } \mathrm{DF} = \sum_{i=1}^{a}(n_i - 1)$$

Dzieląc policzoną wartość przez średnie kwadraty, otrzymamy statystykę *F*:

$$F = \frac{MS_A}{MS_E}$$

Określa ona, czy poszczególne stany zmiennej kategorycznej mają wpływ na zmienną numeryczną. Im większa wartość *F*, z tym większą pewnością możemy odrzucić hipotezę zerową. Otrzymany wynik należy sprawdzić w tablicach rozkładu statystyki *F*. Zawierają one wartości punktów krytycznych dla określonych stopni swobody *DF*.

```
mod = ols('Age ~ Sex', data=data).fit()
aov_table = sm.stats.anova_lm(mod, typ=2)
print (aov_table)
```

```
            sum_sq     df          F    PR(>F)
Sex       1308.405450    1.0   6.246032  0.012671
Residual  149148.230494  712.0      NaN       NaN
```

Dla zmiennych `Sex` i `Age` wartość F wynosi 6,24. Przy 1. stopniu swobody prawdopodobieństwo, że zmienne te są niezależne, wynosi 0,01, czyli jest mniejsze od typowego punktu krytycznego $p = 0{,}05$. Należy więc odrzucić hipotezę zerową i uznać, że różnica wieku między pasażerami a pasażerkami Titanica jest statystycznie istotna.

Istotny wynik testu *F* nie informuje nas, między którymi grupami są różnice. Oznacza jedynie, że zaobserwowano wpływ konkretnego stanu zmiennej kategorycznej na zmienną numeryczną. Aby stwierdzić, między którymi grupami są istotne statystycznie różnice (zmienna kategoryczna może przyjmować 4 stany, z których tylko dwa mają wpływ na zmienną numeryczną), należy wykonać porównania wielokrotne.

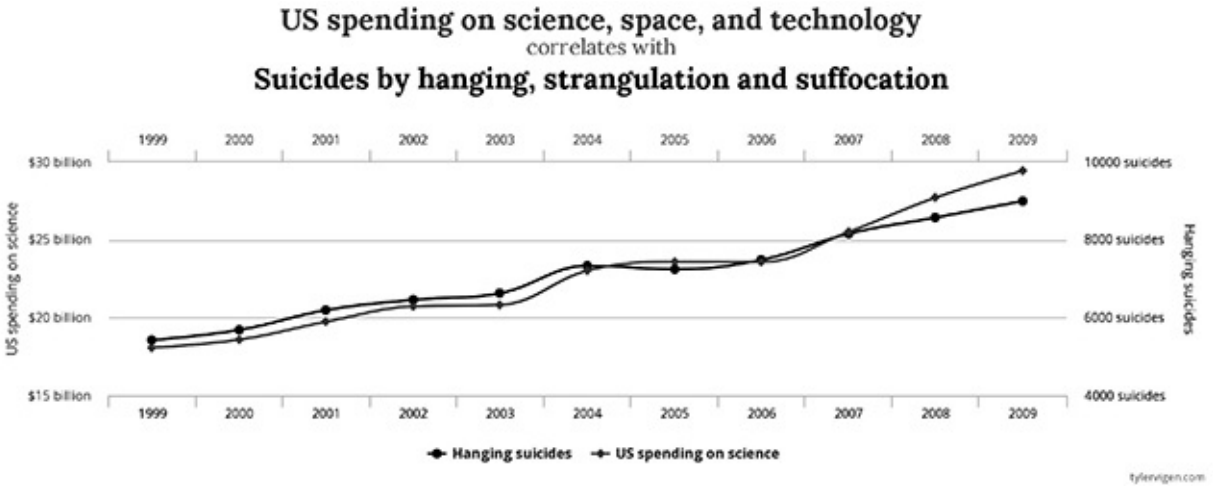
4.9.7. Korelacja oznacza współwystępowanie, a nie związek przyczynowo-skutkowy

Korelacja jest statystyczną miarą zależności między dwiema zmiennymi, czyli wysoki współczynnik korelacji wskazuje jedynie, że pewne wartości obu zmiennych współwystępują ze sobą częściej niż inne. **Nie oznacza jednak, że jedna zmienna w jakiś sposób wpływa na drugą**, w szczególności nie określa ona kierunku ewentualnego związku przyczynowo-skutkowego występującego między zmiennymi.

Badanie związków przyczynowych wymaga przeprowadzenia eksperymentów, czyli kontrolowania rozkładu zmiennych w zbiorach testowych i treningowych.

Bez tego na podstawie nawet najsilniejszej korelacji nie można wnioskować o istnieniu związku przyczynowo-skutkowego, gdyż w rzeczywistości na wartości zmiennych mogą mieć wpływ inne, niezmierzone czynniki. Nie możemy powiedzieć, że wiek pasażerów Titanica zależał od ich płci, tak samo jak nie możemy powiedzieć, że płeć pasażerów zależała od ich wieku.

Tak wiele osób myli korelację z przyczynowością, że stało się to źródłem niekończących się żartów. Na przykład na rysunku 4.15 przedstawiono silną korelację (współczynnik r wynosi 0,99) między wydatkami rządu USA na naukę i technologię a liczbą samobójstw przez powieszenie i uduszenie.



Rysunek 4.15. Tę i wiele innych pozornych zależności przyczynowych można znaleźć na stronie http://www.tylervigen.com/spurious-correlations

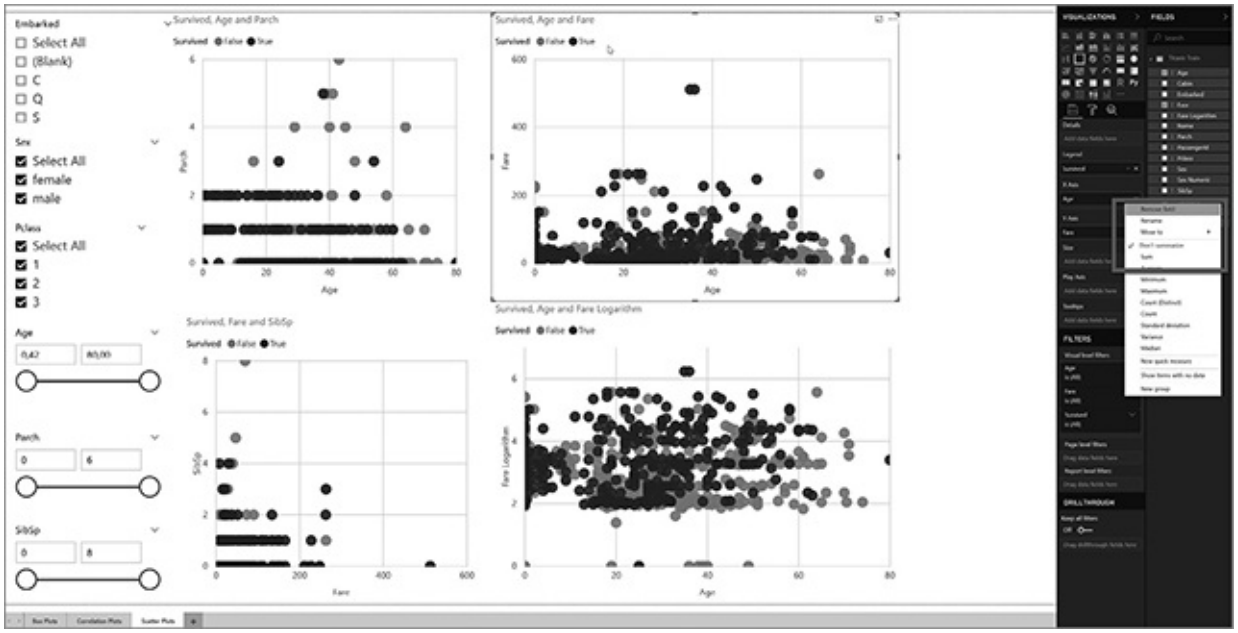
4.10.
Ocena korelacji za pomocą programu Power BI

Statystyki, takie jak współczynnik linowej zależności r, chi-kwadrat albo statystyka F, pozwalają szybko ocenić korelację między zmiennymi. W kolejnych rozdziałach przekonamy się, że używa się ich nawet do automatycznego wybierania zmiennych. Jednak, tak jak wszystkie inne statystyki, mogą nas one wprowadzić w błąd. Jeżeli wyniki testów

statystycznych budzą nasze wątpliwości albo jeżeli chcemy dokładniej przyjrzeć się danym, powinniśmy je zwizualizować.

Możemy wygenerować wiele statycznych wykresów i na każdym z nich pokazać dane z innej perspektywy. Na przykład korelację między wiekiem a przeżyciem katastrofy dla wszystkich pasażerów, osobno dla mężczyzn i kobiet, dla osób podróżujących samotnie albo tylko z jednym członkiem rodziny, itd. Albo możemy użyć programu Power BI do utworzenia dynamicznych raportów. W tym podrozdziale przyjrzymy się drugiemu rozwiązaniu.

Korelacje między zmiennymi numerycznymi przedstawia się na wykresach punktowych. Ponieważ Power BI Desktop domyślnie sumuje zmienne numeryczne, pamiętajmy o wyłączeniu automatycznego sumowania umieszczonych na osiach *X* i *Y* wykresu zmiennych (rys. 4.16).



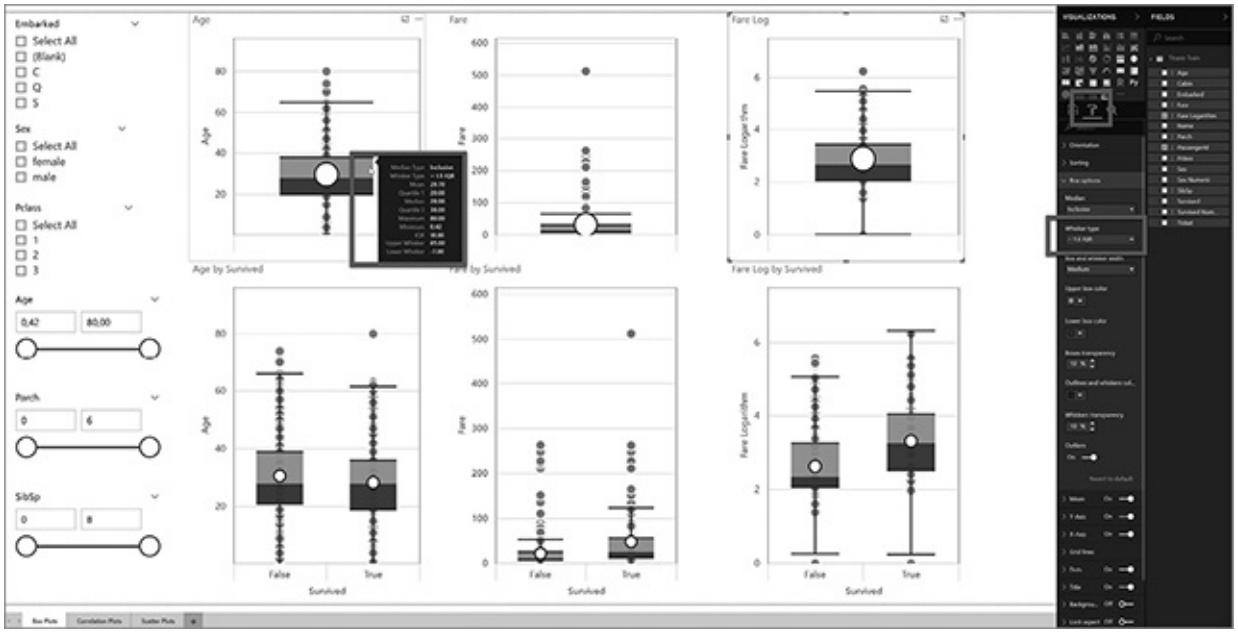
Rysunek 4.16. Ustawienie jasności punktów na podstawie zmiennej Survived pokazało dodatkowo związek między zmiennymi numerycznymi a kategoryczną (dwa wykresy z lewej strony). Znajdujące się po prawej stronie wykresy pokazują brak korelacji między wiekiem a ceną biletu

Jednym z najczęściej używanych do oceny danych wykresów jest wykres pudełkowy. Ten sposób przedstawiania rozkładu wartości zaproponował John

Tukey w 1977 r. [23]. Pozwala on na niewielkiej przestrzeni (mniejszej niż wymagana przez histogram) pokazać statystyki opisowe zmiennej. Niestandardowa wizualizacja Box and Whisker by MAQ Software przedstawia:

- miary tendencji centralnej – mediana jest zaznaczona poziomą linią dzielącą prostokąt (pudełko), średnia arytmetyczna jest zaznaczona okręgiem;
- pierwszy i trzeci kwartyl wyznaczają dolny i górny brzeg pudełka; rozstęp ćwiartkowy poznamy, umieszczając nad pudełkiem kursor myszki;
- powyżej i poniżej pudełka znajdują się wąsy; wąsy są rysowane na kilka sposobów, dlatego zawsze należy sprawdzić, co one oznaczają; w najprostszym przypadku wąsy mogą wyznaczać wartości minimalną i maksymalną (*Whisker type Min/Max*); popularnym typem wykresu pudełkowego jest wykres Tukeya, w którym wartości wąsów są wyznaczane przez półtorakrotność wartości rozstępu ćwiartkowego, o ile tylko te wartości mieszczą się między minimum a maksimum (*Whisker type =1.5 IQR*); na takich wykresach wartości wystające za wąsy są zaznaczane kropkami; spotyka się też wykresy pudełkowe, na których wąsy wyznaczają odchylenie standardowe albo wartości różnych kwantyli (np. 9 i 91 kwantyli lub 5 i 95 kwantyli).

Wykres pudełkowy doskonale nadaje się do przedstawiania korelacji między zmienną numeryczną a kategoryczną. Na rysunku 4.17 pokazałem rozkład zmiennych `Age`, `Fare` i zlogarytmizowanej zmiennej `Fare` (górny wiersz) oraz rozkład tych zmiennych dla obu stanów zmiennej `Survived` (dolny wiersz).



Rysunek 4.17. Młodsi pasażerowie mieli nieco większą szansę przeżycia katastrofy. Silniejsza korelacja występuje między zmiennymi `Fare` i `Survived` – więcej pasażerów, którzy kupili droższe bilety, przeżyło katastrofę. Zależność ta jest jeszcze lepiej widoczna po zlogarytmizowaniu ceny biletów

4.11. Ocena korelacji przy użyciu języka SQL

Mimo braku wbudowanych funkcji statystycznych, język SQL pozwala ocenić korelację zmiennych. Oprócz klauzuli `OVER` i podzapytań wymaga to znajomości matematyki i kreatywności[7].

Zaczniemy od oceny korelacji dwóch zmiennych numerycznych. Kowariancję oraz współczynnik korelacji liniowej r policzymy na podstawie średnich arytmetycznych:

```
WITH CoVarCTE AS
(
SELECT 1.0*Age as val1,
 AVG(1.0*Age) OVER () AS mean1,
 1.0*SibSp AS val2,
 AVG(1.0*SibSp) OVER() AS mean2
FROM [Titanic].[Train]
)
SELECT
 SUM((val1-mean1)*(val2-mean2)) / COUNT(*) AS Covar,
 (SUM((val1-mean1)*(val2-mean2)) / COUNT(*)) /
 (STDEVP(val1) * STDEVP(val2)) AS [r-Pearson]
FROM CoVarCTE;
```

Covar	r-Pearson
-3.33160163728791	-0.20824082128582

Test chi-kwadrat stosowany do oceny korelacji między zmiennymi kategorycznymi bazuje na tabeli częstości. W pierwszym kroku należy utworzyć tabelę z zaobserwowanymi i spodziewanymi rozkładami zmiennych kategorycznych:

```
WITH
ObservedCombination_CTE AS
(
SELECT [Survived] AS OnRows,
    [Sex] OnCols,
 COUNT(*) AS ObservedCombination
FROM [Titanic].[Train]
GROUP BY [Survived], [Sex]
),
ExpectedCombination_CTE AS
(
SELECT OnRows, OnCols, ObservedCombination
 ,SUM(ObservedCombination) OVER (PARTITION BY OnRows) AS ObservedOnRows
 ,SUM(ObservedCombination) OVER (PARTITION BY OnCols) AS ObservedOnCols
 ,SUM(ObservedCombination) OVER () AS ObservedTotal
 ,CAST(ROUND(SUM(1.0 * ObservedCombination) OVER (PARTITION BY OnRows)
  * SUM(1.0 * ObservedCombination) OVER (PARTITION BY OnCols)
  / SUM(1.0 * ObservedCombination) OVER (), 0) AS INT) AS ExpectedCombination
FROM ObservedCombination_CTE
)
SELECT * FROM ExpectedCombination_CTE;
```

OnRows	OnCols	ObservedCombination	ObservedOnRows	ObservedOnCols	ObservedTotal	ExpectedCombination
0	female	81	549	314	891	193
1	female	233	342	314	891	121
1	male	109	342	577	891	221
0	male	468	549	577	891	356

Takie dane pozwalają wyliczyć stopień swobody oraz wartość chi-kwadrat

$$\chi^2 = \Sigma\frac{(O-E)^2}{E}$$

```
…
SELECT (COUNT(DISTINCT OnRows) - 1) * (COUNT(DISTINCT OnCols) - 1) AS DegreesOfFreedom,
        SUM(SQUARE(ObservedCombination - ExpectedCombination)
        / ExpectedCombination) AS ChiSquared
FROM ExpectedCombination_CTE;
```

ChiSquared	DegreesOfFreedom
260.660376192108	1

Otrzymany wynik należy sprawdzić w tablicy rozkładu chi-kwadrat dla otrzymanego stopnia swobody.

Sposobem oceny korelacji między zmienną kategoryczną a numeryczną jest przeprowadzenie analizy wariancji *ANOVA* i wyznaczenie współczynnika *F*. Im jest on większy, tym większe prawdopodobieństwo korelacji między zmiennymi:

```
WITH Anova_CTE AS
(
SELECT Sex, Age,
 COUNT(*) OVER (PARTITION BY Sex) AS gr_CasesCount,
 DENSE_RANK() OVER (ORDER BY Sex) AS gr_DenseRank,
 SQUARE(AVG(Age) OVER (PARTITION BY Sex) -
         AVG(Age) OVER ()) AS between_gr_SS,
 SQUARE(Age -
         AVG(Age) OVER (PARTITION BY Sex))
               AS within_gr_SS
FROM [Titanic].[Train]
)
SELECT N'Between groups' AS [Source of Variation],
 SUM(between_gr_SS) AS SS,
 (MAX(gr_DenseRank) - 1) AS df,
 SUM(between_gr_SS) / (MAX(gr_DenseRank) - 1) AS MS,
 (SUM(between_gr_SS) / (MAX(gr_DenseRank) - 1)) /
 (SUM(within_gr_SS) / (COUNT(*) - MAX(gr_DenseRank))) AS F
FROM Anova_CTE
UNION
SELECT N'Within groups' AS [Source of Variation],
 SUM(within_gr_SS) AS SS,
 (COUNT(*) - MAX(gr_DenseRank)) AS df,
 SUM(within_gr_SS) / (COUNT(*) - MAX(gr_DenseRank)) AS MS,
 NULL AS F
FROM Anova_CTE;
```

Source of Variation	SS	df	MS	F
Between groups	1607.89508638752	1	1607.89508638752	9.58387992313
Within groups	149148.230493649	889	167.77078795686	NULL

Otrzymany wynik należy sprawdzić w tablicy rozkładu współczynnika *F*.

Większość praktyków ocenia, że przygotowanie danych stanowi od 60 do 80% całego projektu BI.

Wayne Eckerson i Colin White, 2003

Rozdział 5
Przygotowanie danych

Osiemdziesiąt procent czasu spędziliśmy na przygotowaniu danych. Trzy czwarte budżetu całego projektu BI pochłonęło przygotowanie danych. Jako Data Scientist dziewięćdziesiąt procent czasu spędzam, przygotowując dane. Każdy z nas przynajmniej raz słyszał taką uwagę. Może nawet sam tak mówił. Podobne wypowiedzi znajdziemy też w wielu wywiadach i opracowaniach statystycznych. Na przykład: *Eksperci zajmujący się danymi poświęcają od 50 do 80 procent czasu na przyziemne prace związane ze zbieraniem i przygotowywaniem danych* [New York Times, 2014]. *Analitycy spędzają do osiemdziesięciu procent swojego czasu, przygotowując dane* [Forrester, 2015]. *Od początku pojawienia się nauki o danych jej praktycy oceniają, że osiemdziesiąt procent ich pracy polega na zdobywaniu i przygotowywaniu danych* [Harvard Business Review, 2016]. Chociaż podawane procenty trochę się różnią (ostatnio zbliżają się do 90), zastanawia niesamowita zgodność wszystkich wypowiedzi. Jeszcze bardziej zaskakujący jest brak danych, na podstawie których powstały przytaczane opinie.

Wydaje się, że źródłem tej piętnastoletniej tradycji jest raport Wayne Eckersona i Colina White'a z którego pochodzi motto tego rozdziału [23]. Autorzy przedstawiają w nim wyniki ankiety wypełnionej przez 741 specjalistów biorących udział w projektach ETL (ang. *Extract, Transform and Load*).

Pozostaje pytanie: dlaczego wszyscy specjaliści od danych od piętnastu lat są tak zgodni w tej kwestii? Przecież projekty, w których biorą udział, różnią się od siebie rozwiązywanymi problemami, ilością danych i użytkowników, zastosowanymi technologiami. Tylko procent czasu potrzebnego na przygotowanie danych jest zawsze ten sam… Powodem jest to, że **dane źródłowe nigdy nie są gotowe do analiz, niezależnie od tego, jak są zbierane i przetwarzane**. Zawsze konieczne jest ich przygotowanie, choć zakres niezbędnych prac jest różny. Przygotowanie danych może obejmować:

- Pobranie danych. Import danych z różnych źródeł i z różnych formatów ułatwiają narzędzia ETL, takie jak usługa SSIS serwera SQL Server. Pobieranie danych nie zostało opisane w książce.
- Scalenie danych. Pobrane z różnych źródeł dane muszą zostać scalone. Często wymaga to usuwania powtarzających się rekordów, uwspólniania typów zmiennych i znalezienia łączących te dane kluczy. Przykładowe dane pochodzą z jednej bazy danych, a proces ich scalania nie jest w książce opisany.
- Uporządkowanie danych. Dane analityczne należy uporządkować, czyli przekształcić do postaci tabel, których wiersze reprezentują przykłady, a kolumny – zmienne. Przygotowywanie danych zaczniemy od przedstawienia technik i narzędzi ich uporządkowania.
- Wzbogacenie danych. Wzbogacanie danych polega na dodawaniu zmiennych wyliczeniowych, czyli zmiennych obliczonych na podstawie posiadanych danych. Stosuje się dwie metody wzbogacania danych. Albo wykorzystuje się wiedzę z dziedziny eksperymentu, na przykład lekarz będzie wiedział, że współczynnik masy ciała BMI jest silniej skorelowany z niektórymi chorobami, niż używane do obliczenia tego współczynnika wzrost i waga. Albo wykorzystuje się metody głębokiego uczenia maszynowego do automatycznego wyliczenia przydatnych zmiennych[8].

- Wyczyszczenie danych. Wykryte w danych błędy należy usunąć. Na tym etapie należy rozwiązać problemy brakujących wartości, duplikatów (powtarzających się przykładów), wartości odstających i błędnych.
- Przekształcenie danych. Metody przekształcania danych zależą od typów zmiennych. Zmienne numeryczne się normalizuje, uogólnia, wygładza oraz poddaje dyskretyzacji. Zmienne kategoryczne się uogólnia, porządkuje oraz koduje jako zmienne numeryczne.
- Redukcję danych. Redukcja danych ma na celu zmniejszenie liczby przykładów lub zmiennych wejściowych w taki sposób, żeby uprościć analizę i poprawić jakość modeli. Jeżeli danych jest na tyle dużo, że ich przetwarzanie i trening modeli staje się problematyczny, możemy do eksperymentów użyć próby danych, dane treningowe można też podzielić na klastry. Zawsze należy wziąć pod uwagę ograniczenie liczby zmiennych. Nieprzydatne zmienne (zmienne niepowiązane z wyjściową lub silnie powiązane ze sobą) należy usunąć. Powinniśmy również rozważyć redukcję liczby zmiennych, na przykład metodą analizy głównych składowych PCA.
- Podział danych. Podział danych obejmuje równoważenie danych (zmianę nierównomiernego rozkładu zmiennej wyjściowej) i wydzielenie danych walidacyjnych oraz testowych.


Celem projektu uczenia maszynowego jest zbudowanie jak najlepszego modelu predykcyjnego. Jakość modeli predykcyjnych mierzona jest ich błędem rzeczywistym (testowym). Użytkownicy nie oceniają modeli na podstawie zastosowanych algorytmów uczenia maszynowego czy znalezionych wartości hiperparametrów. Dane treningowe również nie mają wpływu na ocenę modeli przez użytkowników. Każdy sposób przygotowania danych prowadzący do zmniejszenia błędu testowego jest prawidłowy. Powiem więcej – nieprzygotowanie danych, na przykład nieusunięcie błędnych przykładów i niewyliczenie przydatnych zmiennych to częste błędy prowadzące do uzyskania nieoptymalnych modeli.

5.1. Uporządkowanie danych

Opracowany w latach 70. XX w. przez Edgara Franka Codda model relacyjnych baz danych narzuca uporządkowanie danych [25]. W modelu relacyjnym dane są przechowywane w wielu odrębnych, ale powiązanych ze sobą tabelach. Te powiązania łączące tabele nazywa się relacjami. Aby relacje były jednoznaczne, połączone nimi tabele muszą mieć **klucze podstawowe**, a zapisane w nich identyfikatory muszą być powtórzone w każdej z połączonych tabel. Kolumna, która zawiera identyfikator rekordu innej tabeli, nazywa się **kluczem obcym**.

Proces dostosowywania schematu bazy danych do wymogów modelu relacyjnego nazywa się normalizacją. Jego głównym celem jest wyeliminowanie wynikających z nadmiarowości danych anomalii, które mogłyby doprowadzić do utraty spójności danych podczas ich modyfikacji. Normalizacja polega na doprowadzeniu schematu bazy do założonej (przynajmniej trzeciej) postaci normalnej. Definicje kolejnych postaci normalnych są następujące:

- tabela jest zgodna z pierwszą postacią normalną, jeśli wszystkie jej kolumny przechowują atomowe, niepodzielne wartości; za atomowe wartości należy przyjąć takie, które mogą być użyte w przyszłości do wyszukiwania, grupowania lub sortowania; ponadto, aby spełnić wymogi pierwszej postaci normalnej, tabela musi mieć klucz podstawowy;
- tabela jest zgodna z drugą postacią normalną, jeżeli jest w pierwszej postaci normalnej i wartości jej wszystkich niekluczowych kolumn zależą od całego klucza podstawowego; doprowadzenie tabeli do drugiej postaci normalnej polega na usunięciu z niej kolumn, które zależą funkcyjnie od części klucza podstawowego;
- tabela jest zgodna z trzecią postacią normalną, jeżeli jest już w drugiej postaci normalnej i wartości jej kolumn nie zależą od niekluczowych atrybutów; doprowadzenie tabeli do trzeciej postaci normalnej polega na znalezieniu i usunięciu przechodnich zależności między zmiennymi;
- tabela jest zgodna z postacią Boyce’a–Codda (BCNF), jeżeli jest już

w trzeciej postaci normalnej i dla każdej nietrywialnej zależności między podzbiorami jej zmiennych zbiór będący wyznacznikiem jest jej zbiorem identyfikującym;

- tabela jest zgodna z czwartą postacią normalną, jeżeli jest już w postaci Boyce'a–Codda i nie występują w niej zależności wielowartościowe.

Relacyjny model baz danych i zasady normalizacji nie są powszechnie znane analitykom i specjalistom od danych. To był jeden z powodów, dla których w 2014 r. Hadley Wickham opublikował artykuł *Tidy Data* (Uporządkowane dane) [20]. Przedstawił w nim podobne do postaci normalnych zasady uporządkowywania danych, ale wyraził je za pomocą języka lepiej znanego statystykom. Ponadto uzupełnił on artykuł stworzoną przez siebie biblioteką `tidyr`. Dane o pasażerach Titanica są uporządkowane, dlatego w tym podrozdziale posłużymy się przykładowymi danymi z biblioteki `tidyr`.

Przykładem uporządkowanych danych jest tabela `table1`.

```
library(tidyverse)
```

```
table1
# A tibble: 6 × 4
       country  year  cases population
         <chr> <int>  <int>      <int>
1 Afghanistan  1999    745   19987071
2 Afghanistan  2000   2666   20595360
3       Brazil  1999  37737  172006362
4       Brazil  2000  80488  174504898
5        China  1999 212258 1272915272
6        China  2000 213766 1280428583
```

Dzięki temu, że każda kolumna reprezentuje jedną zmienną, a każdy wiersz jeden przykład, analiza i wizualizacja takich danych są łatwe. Na przykład policzenie dla poszczególnych lat liczby przypadków przypadających na 10 000 mieszkańców każdego kraju sprowadza się do wykonania jednej instrukcji.

```
table1$cases / table1$population * 10000
```

```
[1] 0.372741 1.294466 2.193930 4.612363 1.667495 1.669488
```

Hadley Wickham zauważył, że wśród wszystkich możliwych rodzajów nieuporządkowanych danych pięć błędów występuje najczęściej. Każdy z nich można naprawić, używając odpowiedniej funkcji biblioteki `tidyr`.

Pierwszy problem polega na przechowywaniu w różnych komórkach (różnych wierszach jednej kolumny) wartości wielu zmiennych. Z reguły nazwy tych zmiennych są przechowywane w innej kolumnie, a obie te kolumny razem tworzą parę klucz–wartość. Przykładem takiej nieuporządkowanej tabeli jest tabela `table2`.

```
table2
```

```
# A tibble: 12 × 4
       country   year       type       count
         <chr>  <int>      <chr>       <int>
1  Afghanistan   1999      cases         745
2  Afghanistan   1999 population    19987071
3    Afghanistan   2000      cases        2666
4    Afghanistan   2000 population    20595360
5         Brazil   1999      cases       37737
6         Brazil   1999 population   172006362
7         Brazil   2000      cases       80488
8         Brazil   2000 population   174504898
9          China   1999      cases      212258
10         China   1999 population  1272915272
11         China   2000      cases      213766
12         China   2000 population  1280428583
```

Analizowanie takich danych wymaga wybrania wierszy na podstawie kolumny klucza.

```
case_rows <- c(1, 3, 5, 7, 9, 11, 13, 15, 17)
pop_rows <- c(2, 4, 6, 8, 10, 12, 14, 16, 18)
table2$count[case_rows] / table2$count[pop_rows] * 10000
```

```
[1] 0.372741 1.294466 2.193930 4.612363 1.667495 1.669488       NA       NA       NA
```

Problem wielu zmiennych przechowywanych w jednej kolumnie rozwiązuje funkcja `spread`. Rozdziela ona pary klucz–wartość na osobne kolumny. W przykładowej tabeli wystąpiły dwie wartości klucza (kolumny `type`), więc uporządkowana tabela będzie zawierała dwie nowe kolumny, które zastąpią kolumny klucza i wartości.

```
spread(table2, type, count)
```

```
# A tibble: 6 × 4
       country  year    cases   population
*        <chr> <int>    <int>        <int>
1  Afghanistan  1999      745     19987071
2  Afghanistan  2000     2666     20595360
3       Brazil  1999    37737    172006362
4       Brazil  2000    80488    174504898
5        China  1999   212258   1272915272
6        China  2000   213766   1280428583
```

Drugim typowym problemem jest przechowywanie w jednej kolumnie połączonych wartości wielu zmiennych. Nie ma prostego i wydajnego sposobu analizowania takich danych.

```
table3
```

```
A tibble: 6 × 3
       country  year                 rate
*        <chr> <int>                <chr>
1  Afghanistan  1999          745/19987071
2  Afghanistan  2000         2666/20595360
3       Brazil  1999          37737/172006362
4       Brazil  2000          80488/174504898
5        China  1999        212258/1272915272
6        China  2000        213766/1280428583
```

Problem kolumn przechowujących złączone wartości wielu zmiennych musimy rozwiązać, rozdzielając te wartości. Do rozdzielenia zmiennych użyjemy funkcji `separate`.

```
separate(table3, rate, into = c("cases", "population"))
```

```
# A tibble: 6 × 4
       country  year    cases   population
*        <chr> <int>    <chr>        <chr>
1  Afghanistan  1999      745     19987071
2  Afghanistan  2000     2666     20595360
3       Brazil  1999    37737    172006362
4       Brazil  2000    80488    174504898
5        China  1999   212258   1272915272
6        China  2000   213766   1280428583
```

Kolejny, często występujący problem jest przeciwieństwem drugiego. **Trzecim typowym problemem jest podzielenie wartości tej samej zmiennej między różne kolumny**. Na przykład w tabeli `tabele5` wiek i rok są zapisane w osobnych kolumnach.

```
table5
```

```
# A tibble: 6 × 4
       country century   year               rate
*        <chr>   <chr>  <chr>              <chr>
1  Afghanistan      19     99       745/19987071
2  Afghanistan      20     00      2666/20595360
3       Brazil      19     99    37737/172006362
4       Brazil      20     00    80488/174504898
5        China      19     99  212258/1272915272
6        China      20     00  213766/1280428583
```

Analiza takich danych wymaga wcześniejszego złączenia zmiennej. Użyta do tego funkcja `unite` łączy dowolną liczbę kolumn tabeli w jedną kolumną o podanej nazwie.

```
unite(table5, "year", century, year, sep = "")
```

```
# A tibble: 6 × 3
       country  year                  rate
*        <chr> <chr>                 <chr>
1  Afghanistan  1999          745/19987071
2  Afghanistan  2000         2666/20595360
3       Brazil  1999    37737/172006362
4       Brazil  2000    80488/174504898
5        China  1999  212258/1272915272
6        China  2000  213766/1280428583
```

Czwarty typowy problem wynika z używania jako nagłówków kolumn wartości zamiast nazw zmiennych. Komórki otrzymanych w ten sposób tabel przestawnych przechowują wartości zmiennych wyznaczonych przez przecięcie kolumn i wierszy. Na przykład w kolumnach `1999` i `2000` tabeli `tabele4a` znajdują się liczby przypadków odnotowanych w tych latach w różnych krajach.

```
table4a
```

```
# A tibble: 3 × 3
       country  `1999`   `2000`
*        <chr>   <int>    <int>
1  Afghanistan     745     2666
2       Brazil   37737    80488
3        China  212258   213766
```

Takie tabele przestawne doskonale nadają się do prezentacji danych, ale ich analiza wymaga już zebrania podzielonych między wiersze i kolumny wartości zmiennej.

```
cases <- c(table4a$`1999`, table4a$`2000`)
population <- c(table4b$`1999`, table4b$`2000`)
cases / population * 10000
```

```
[1] 0.372741 2.193930 1.667495 1.294466 4.612363 1.669488
```

Czwarty problem rozwiążemy, zapisując w nowej zmiennej `year` nagłówki kolumn i zbierając wartości drugiej i trzeciej kolumny tabeli przestawnej do

zmiennej `cases`. Ułatwia to funkcja `gather`.

```
gather(table4a, "year", "cases", 2:3)
```

```
# A tibble: 6 × 3
       country  year    cases
         <chr> <chr>    <int>
1  Afghanistan  1999      745
2       Brazil  1999    37737
3        China  1999   212258
4  Afghanistan  2000     2666
5       Brazil  2000    80488
6        China  2000   213766
```

Piąty typowy problem polega na podzieleniu przykładów między wiele tabel. W poprzednim przekładzie mieliśmy do czynienia również z tym problemem – dane o liczbie mieszkańców krajów znajdowały się w osobnej tabeli `table4b`. W dodatku ta tabela miała format tabeli przestawnej. Uporządkowanie danych będzie wymagać zebrania wartości podzielonych między kolumny zmiennych (tym razem ostatnim parametrem funkcji `gather` jest `-1`, co oznacza, że zebrane zostaną wartości wszystkich kolumn z wyjątkiem pierwszej), a następnie złączenia obu tabel.

```
tidy4a <- gather(table4a, "year", "cases", -1)
tidy4b <- gather(table4b, "year", "population", -1)
left_join(tidy4a, tidy4b)
```

```
# A tibble: 6 × 3
       country  year    cases
         <chr> <chr>    <int>
1  Afghanistan  1999      745
2       Brazil  1999    37737
3        China  1999   212258
4  Afghanistan  2000     2666
5       Brazil  2000    80488
6        China  2000   213766
```

Odpowiednio użyte funkcje biblioteki `tidyr spread`, `gather`, `separate`, `unite`,

`inner_join`, `left_join`, `right_join` i `full_join` pozwalają uporządkować dowolne, tabelaryczne dane źródłowe.

5.2. Wzbogacanie danych

Niektóre zmienne wejściowe są nieprzydatne dla modeli uczenia maszynowego. Na przykład zmienna `Name` przyjmuje różne wartości dla wszystkich przykładów treningowych i żadna z tych wartości nie występuje w danych testowych. Może jednak zawiera ona informacje, które po wydobyciu okazałyby się przydatne? Takie jak tytuł (zwrot grzecznościowy) danego pasażera? Jeżeli panowie, panie, panny, hrabiny, kapitanowie czy wielebni występowali w danych treningowych i testowych, a dodatkowo istnieje korelacja między tytułem pasażera a przeżyciem przez niego katastrofy, dodanie zmiennej `Title` może znacząco poprawić jakość modelu.


Algorytmy uczenia maszynowego nie dzielą w żaden sposób zmiennych wejściowych, traktując ich wartości jako atomowe. Wyodrębnianie atomowych wartości ze zmiennych należy traktować jako element uporządkowywania danych, w wyniku którego wzbogacamy model o wartościowe zmienne.

Tytuły pasażerów zostały zapisane między nazwiskiem a imieniem. Na szczęście od nazwiska oddziela je przecinek, a od imienia kropka. Wyodrębnienie tytułów jest więc względnie łatwe, nawet w języku SQL. Dodając do początku szukanego ciągu znaków 2 i odejmując od jego końca 2 znaki pozbędziemy się oddzielających je znaków specjalnych (przecinka i kropki) oraz występujących po i przed nimi spacji.

```
SELECT [Name],
      SUBSTRING([Name], CHARINDEX(',', [Name])+2
      ,CHARINDEX('.',[Name],CHARINDEX(',', [Name])+2) - CHARINDEX(',', [Name])-2) AS Title
FROM [Titanic].[Train];
```

Name	Title
O'Dwyer, Miss. Ellen ""Nellie	Miss
Todoroff, Mr. Lalio	Mr
Uruchurtu, Don. Manuel E	Don
Spencer, Mrs. William Augustus (Marie Eugenie)	Mrs
Glynn, Miss. Mary Agatha	Miss
…	

Oryginalne zmienne wejściowe często są mało przydatne do rozwiązania postawionego problemu, ale po odpowiednim przekształceniu ich siła predykcyjna znacznie wzrasta. Widzieliśmy przykład takiej zmiennej w rozdziale 2. Same w sobie wartości poszczególnych faktur nie są skorelowane z próbami oszustwa. Przecież nie można założyć, że im większa kwota faktury, tym większe ryzyko oszustwa. Mimo to, wyodrębniając pierwszą cyfrę kwoty faktury i dodając do zmiennych wejściowych częstości występowania pierwszych cyfr kwot na fakturach otrzymaliśmy wartościowe zmienne i wykryliśmy próby oszustwa. Do przekształcenia kwot na częstości występowania w nich pierwszych cyfr została użyta poniższa funkcja.

```
SELECT VendorNumber, InvoiceAmount
                ,ROUND(CASE
                        WHEN InvoiceAmount >= 1000000000 THEN InvoiceAmount / 1000000000
                        WHEN InvoiceAmount >= 100000000 THEN InvoiceAmount / 100000000
                        WHEN InvoiceAmount >= 10000000 THEN InvoiceAmount / 10000000
                        WHEN InvoiceAmount >= 1000000 THEN InvoiceAmount / 1000000
                        WHEN InvoiceAmount >= 100000 THEN InvoiceAmount / 100000
                        WHEN InvoiceAmount >= 10000 THEN InvoiceAmount / 10000
                        WHEN InvoiceAmount >= 1000 THEN InvoiceAmount / 1000
                        WHEN InvoiceAmount >= 100 THEN InvoiceAmount / 100
                        WHEN InvoiceAmount >= 10 THEN InvoiceAmount / 10
                        WHEN InvoiceAmount < 10 THEN InvoiceAmount
                        END, 0, 1) as Digits
                , COUNT(*) OVER(PARTITION BY VendorNumber) as #Transactions
FROM [BenfordFraud].[Invoices];
```

VendorNumber	InvoiceAmount	Digits	#Transactions
U9324175	16.93	1.00	1
G20036	2232.68	2.00	613
Y5365360	188.45	1.00	1
480626	9445.25	9.00	628
232367	450.83	4.00	621
105436	2244.70	2.00	5584
Z8072622	482.94	4.00	1
C7757275	229.59	2.00	1
…			

Mała przydatność zmiennych może wynikać z ich niedopasowania do rozwiązywanego problemu i zastosowanego algorytmu uczenia maszynowego. Drzewa na przykład decyzyjne dzielą przypadki według znalezionych reguł. Reguły te zawsze mają postać warunków `zmienna =,>,< stała`. I tak, gdybyśmy do klasyfikacji gatunków irysów użyli drzew decyzyjnych, znalezione zostałyby poniższe reguły:

```
Petal.Length<2.45 50 setosa
Petal.Length>=2.45 50 versicolor 50 virginica
                Petal.Width<1.75 49 virginica 5 versicolor
                Petal.Width>=1.75 45 versicolor  1 virginica
```

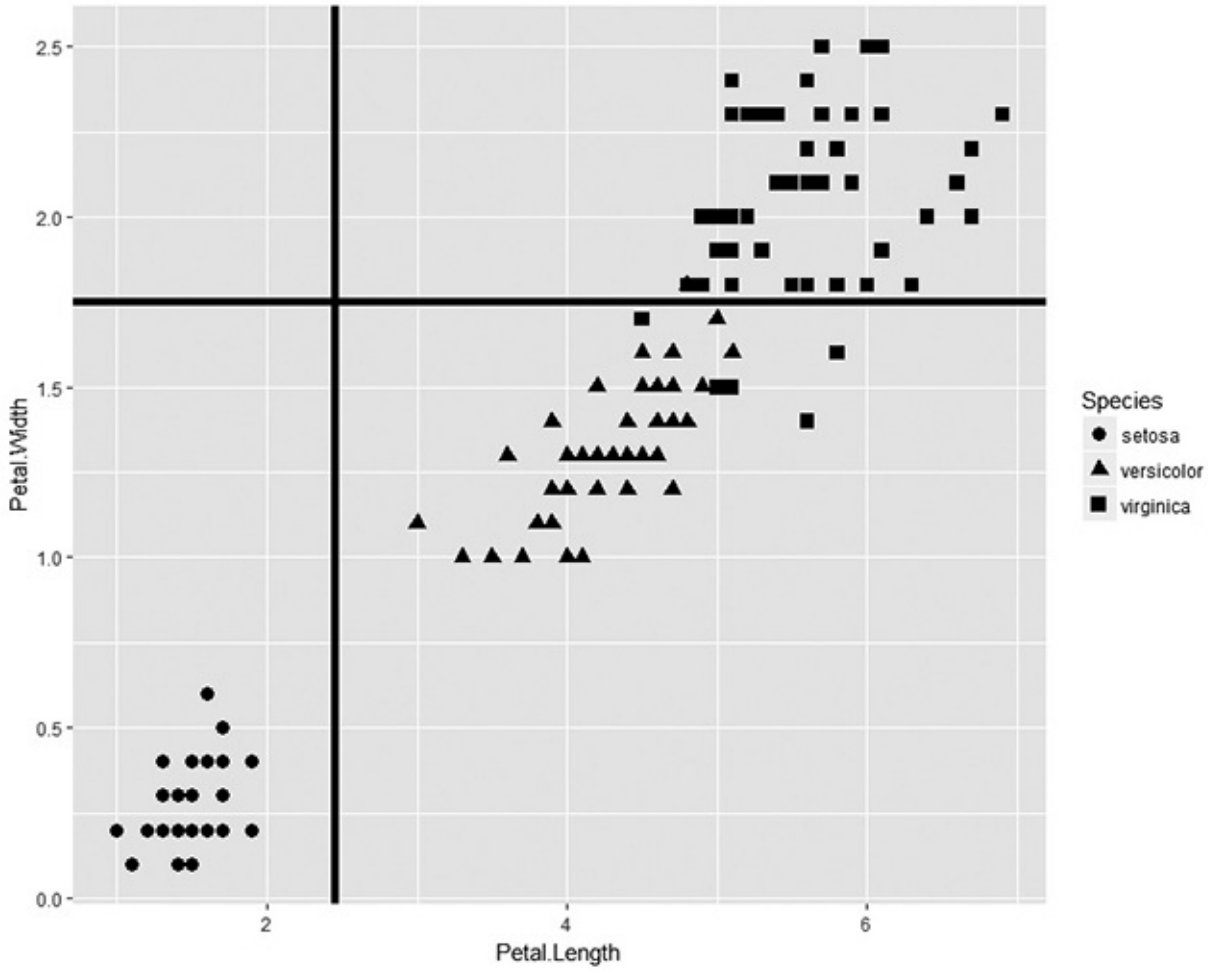
Zauważmy, że gdyby przedstawić te reguły na wykresie, otrzymalibyśmy proste

równoległe do osi zmiennych, przecinające się pod kątem prostym (rys. 5.1).

```
library(caret)
data <- iris
attach(data)

ggplot(data, aes(Petal.Length, Petal.Width)) +
  geom_point(aes(shape = Species), size = 3) +
  geom_vline(xintercept = 2.45, size = 2) +
  geom_hline(yintercept = 1.75, size = 2)
```

Ten sposób konstruowania reguł oznacza, że reprezentowane przez drzewa decyzyjne hipotezy nigdy nie podzielą dokładnie zmiennych ciągłych. W teorii możliwe jest zbudowanie dokładnego przybliżenia linii nierównoległej do osi za pomocą wielu reguł. Takie przybliżenie wyglądałoby jak nałożone na linię schodki. W praktyce drzewa decyzyjne zawierające takie reguły musiałyby być ogromne i prawie na pewno nadmiernie dopasowane do danych treningowych. Problem ten nie dotyczy zmiennych dyskretnych – zmienne dyskretne daje się dokładnie podzielić liniami równoległymi do osi, przecinającymi się pod kątem prostym. Powinniśmy też wiedzieć, że omawiany problem traci na znaczeniu w przestrzeniach wielowymiarowych, w których odległości między punktami–przykładami są duże. Więcej na temat specyfiki przestrzeni wielowymiarowych napisałem w punkcie 4.9.1. Zauważmy, że im więcej wymiarów, tym rzadsza przestrzeń i tym większe prawdopodobieństwo, że znajdujące się w niej punkty można podzielić liniami prostopadłymi.



Rysunek 5.1. Drzewa decyzyjne zawsze dzielą przykłady według linii prostopadłych

Co możemy zrobić, żeby ułatwić drzewom decyzyjnym dokładne znalezienie zależności między zmiennymi a ich kombinacjami? Zagadnienie to zilustrujemy na kanonicznych danych uczenia maszynowego, zbiorze *iris* [26]. Może iloczyny długości i szerokości płatków i liści są silniej skorelowane niż ich długości i szerokości? Aby to sprawdzić, **należy wzbogacić dane o wyniki operacji matematycznych na zmiennych**, na przykład ich iloczyny czy wyniki odejmowania.

Zmiennymi o dużej wartości predykcyjnej są też wyniki porównań, na przykład porównania długości liści z długością płatków, salda z debetem czy wydatków w tym miesiącu z wydatkami w poprzednich miesiącach. **Warto też sprawdzić siłę predykcyjną zmiennych otrzymanych przez zastąpienie**

wartości bezwzględnych procentowymi (np. sumę miesięcznych obciążeń na koncie procentem przyznanego kredytu).

Sprawdźmy siłę predykcyjną oryginalnych zmiennych w modelu drzew decyzyjnych[9].

```
model <- rpart(formula = Species ~., data = data)
caret::varImp(model)
```

```
                Overall
Petal.Length 87.35354
Petal.Width   88.96940
Sepal.Length 44.85092
Sepal.Width   22.59406
```

Dodajmy teraz kilka zmiennych wyliczeniowych, wytrenujmy ten sam model drzew decyzyjnych na wzbogaconych danych i porównajmy siłę predykcyjną poszczególnych zmiennych.

```
data$Petal <- Petal.Length*Petal.Width
data$Sepal <- Sepal.Length*Sepal.Width
data$Difference <- data$Sepal-data$Petal
data$IsLonger <- Petal.Length>Sepal.Width

model <- rpart(formula = Species ~., data = data)
caret::varImp(model)
```

```
Overall
IsLonger         50.00000
Petal            92.59259
Petal.Length     87.35354
Petal.Width      88.96940
Sepal            10.68687
Sepal.Length     44.85092
Sepal.Width      0.00000
Difference       0.00000
```

Najbardziej przydatną zmienną okazała się wyliczona zmienna `Petal`. Ten prosty przykład pokazuje, jak bardzo jakość modeli predykcyjnych zależy od

zmiennych wyliczeniowych. I jak ważnym etapem projektów uczenia maszynowego jest wzbogacanie danych. Jest też to etap wymagający ścisłej współpracy z ekspertem z dziedziny eksperymentu. O ile do zrozumienia danych, ich wyczyszczenia i przekształcenia niezbędna jest przede wszystkim wiedza z zakresu statystyki, o tyle wzbogacanie danych wymaga znajomości modelowanego zagadnienia. Bez takiej wiedzy musielibyśmy wygenerować ogromną liczbę zmiennych wyliczeniowych i ocenić ich przydatność dla modelu. Jak wspomniałem, ponieważ wiedza ekspercka jest trudno dostępna, od niedawna popularność zyskuje podejście polegające na użyciu głębokich sieci neuronowych do automatycznego generowania wartościowych zmiennych (metody głębokiego uczenia maszynowego wykraczają poza zakres książki).

Wróćmy do analizy katastrofy Titanica. Sprawdziliśmy, że zmienne `SibSp` i `Parch` są słabo skorelowane ze zmienną `Survived`. Może jednak łączna liczba podróżujących razem członków rodziny okaże się bardziej przydatna? A może najbardziej przydatna okaże się zmienna binarna zawierająca informację o tym, czy dany pasażer podróżował sam, czy z kimś z rodziny?

```
SELECT [SibSp] + [Parch] + 1 AS FamilySize,
       CASE WHEN [SibSp] + [Parch]        = 0 THEN 1 ELSE 0 END AS IsAlone
FROM Titanic.Train;
FamilySize      IsAlone
1                       1
5                       0
3                       0
2                       0
3                       0
1                       1
…
```

Znaczenie znajomości modelowanego zagadnienia ilustruje kolejny przykład. W rozdziale 2 zbudowaliśmy model klasyfikujący przejazdy taksówką w Nowym Jorku. Postawiliśmy wtedy hipotezę, że danie napiwku

taksówkarzowi lub brak napiwku zależy od tego, czy taksówkarz wiózł pasażera najkrótszą drogą do celu, czy krążył po mieście. Aby ją zweryfikować, do zmiennych dodaliśmy odległość euklidesową między współrzędnymi punktów rozpoczęcia i zakończenia przejazdu. Do jej wyliczenia użyliśmy funkcji `[NYCTaxi].[fnCalculateDistance]`.

```
ALTER FUNCTION [NYCTaxi].[fnCalculateDistance]
(@Lat1 float, @Long1 float, @Lat2 float, @Long2 float)
RETURNS float
AS
BEGIN
  DECLARE @distance decimal(28, 10)
  -- Convert to radians
  SET @Lat1 = @Lat1 / 57.2958
  SET @Long1 = @Long1 / 57.2958
  SET @Lat2 = @Lat2 / 57.2958
  SET @Long2 = @Long2 / 57.2958
  -- Calculate distance
  SET @distance = (SIN(@Lat1) * SIN(@Lat2)) + (COS(@Lat1) * COS(@Lat2) * COS(@Long2 -
@Long1))
  --Convert to miles
  IF @distance <> 0
  BEGIN
    SET @distance = 3958.75 * ATAN(SQRT(1 - POWER(@distance, 2)) / @distance);
  END
  RETURN @distance
END
```

Zachęcam do samodzielnego zbudowania modelu klasyfikatora przejazdów taksówkami z uwzględnieniem wiadomości z tego i poprzedniego rozdziału i porównaniu jego jakości z oryginalnym modelem.

5.2.1. Data i czas

Daty same w sobie dla większości modeli uczenia maszynowego są bezużyteczne. Popatrzmy na model oceniający, czy transakcja kartą kredytową

jest próbą oszustwa, czy nie. Nawet jeżeli wiele przykładowych transakcji pochodzi z tego samego dnia i nawet jeżeli transakcje z 25-12-2018 różniły się od transakcji z 03-04-2019, to i tak żadna z tych dat nie wystąpi w danych ocenianych przez model wdrożony do użycia 01-09-2019 r. Weźmy inny przykład – model oceniający ryzyko wystąpienia awarii. Tylko zagorzali wyznawcy astrologii użyją daty, w której wystąpiła awaria do przewidzenia, kiedy wystąpi kolejna usterka.

Zauważmy, że oba te modele (model wykrywający oszustwa i model prognozujący awarie) mogą wiele zyskać, jeżeli wyodrębnimy ukryte w datach informacje. Możliwe, że w dni wolne ryzyko oszustwa jest większe niż w robocze. Albo że urządzenia wyprodukowane w poniedziałki na pierwszej zmianie (czyli między 8:00 a 16:00) są bardziej awaryjne.

Punktem wyjścia do wyodrębniania ukrytych w datach informacji może być poniższy skrypt.

```
DECLARE @date DATETIME  = GETDATE();
SELECT
   [Date]        = @date,
   [Year]        = DATEPART(YEAR, @date),
   [DayOfYear]   = CONVERT(SMALLINT, DATEPART(DAYOFYEAR, @date)),
   [Quarter]     = CONVERT(TINYINT, DATEPART(QUARTER, @date)),
   [Month]       = CONVERT(TINYINT, DATEPART(MONTH, @date)),
   [Week]        = CONVERT(TINYINT, DATEPART(WEEK, @date)),
   [Day]         = CONVERT(TINYINT,  DATEPART(DAY, @date)),
   [Weekday]     = CONVERT(TINYINT, DATEPART(WEEKDAY, @date)),
   [IsWeekend]   = CONVERT(BIT, CASE WHEN  DATEPART(WEEKDAY, @date) IN (1,7) THEN 1 ELSE 0
END),
   [AM]          = CONVERT(TINYINT,CASE WHEN DATEPART(HOUR,@date) >= 12 THEN 0 ELSE 1 END),
   [Hour]        = CONVERT(TINYINT,CASE WHEN DATEPART(HOUR, @date) > 12 THEN
DATEPART(HOUR, @date) - 12 ELSE DATEPART(HOUR,@date) END),
   [Minute]      = CONVERT(TINYINT,DATEPART(MINUTE, @date)),
   [Second]      = CONVERT(TINYINT,DATEPART(SECOND,@date));
```

```
Date    Year     DayOfYear   QuarterMonth   Week     Day   WeekdayIsWeekend  AM
Hour    Minute   Second
2018-12-25 10:04:08.450 2018 359  4        12       52  25    3       0    1
10      4        8
```

W zależności od postawionego problemu należy go rozbudowywać. Z dat możemy wydobyć znacznie więcej wartościowych informacji, takich jak: czy był to dzień wolny, czy roboczy, czy jest to godzina szczytu, nazwę przypadającego tego dnia święta, liczbę dni od i do najbliższego święta, liczbę dni roboczych w bieżącym miesiącu itd.

5.3. Wyczyszczenie danych

Problemy wykryte podczas analizy zmiennych i oceny korelacji między nimi muszą zostać usunięte. Chociaż niektóre algorytmy uczenia maszynowego są bardziej wrażliwe na błędne dane niż inne, zawsze należy kierować się zasadą GIGO (ang. *garbage in, garbage out*). **Jeżeli do treningu modelu użyjemy śmieci, dostaniemy śmieciowy model** – nawet jeżeli zastosujemy najbardziej

wyrafinowany algorytm uczenia maszynowego, zaangażujemy najlepszych ekspertów i poświęcimy na trening tysiące godzin czasu procesorów. Wyniki przetwarzania błędnych danych i tak będą błędne, niezależnie od poprawności procedury ich przetwarzania.

Czyszczenie danych jest czteroetapowe. Zaczynamy od usunięcia brakujących wartości. Następnie należy usunąć duplikaty i błędne dane, a na końcu pozbyć się wartości odstających.



Wstępnym etapem czyszczenia danych jest usuwanie nieprzydatnych zmiennych. W poprzednim rozdziale wyjaśniłem, że nieprzydatne zmienne to: zmienne, które przyjmują jedną wartość; zmienne, których nie można uogólnić; zmienne nieskorelowane (bezpośrednio lub pośrednio) ze zmienną wyjściową oraz zmienne silnie skorelowane ze sobą (w takim przypadku należy usunąć jedną ze skorelowanych zmiennych wejściowych). Pozostawienie nieprzydatnych zmiennych zwiększa poziom szumu (ang. *noise*) w danych treningowych i prowadzi do nadmiernego dopasowania modeli.

5.3.1. Usuwanie brakujących wartości

Uczenie nadzorowane wymaga opisania przykładów. Dlatego każdy przykład, dla którego nie znamy wartości zmiennej wyjściowej, należy usunąć.

```
DELETE FROM [Titanic].[Train]
WHERE [Survived] IS NULL;
```

Problem brakujących wartości zmiennych wejściowych można rozwiązać na kilka sposobów. **Wybierając najlepszy dla danej zmiennej sposób, powinniśmy się kierować zasadą minimalizacji zmian w rozkładzie danych**. Stopień zmian określa się przy użyciu tabeli częstości dla zmiennych kategorycznych i statystykami opisowymi w przypadku zmiennych numerycznych. Chodzi o to, żeby w jak najmniejszy sposób zmienić charakterystykę uzupełnianej zmiennej. Dlatego dobrą praktyką jest ponowne ocenienie uzupełnionych danych i porównanie otrzymanych wyników z wynikami obserwacji danych źródłowych.

Minimalizacja zmian w rozkładzie wartości służy do ograniczenia zmian w korelacji między zmiennymi. Od tej zasady jest jeden wyjątek. Usuwając brakujące wartości, pozbywamy się potencjalnie przydatnej informacji – informacji o tym, że dana wartość była brakująca. Czasami jest to bardzo cenna informacja. Na przykład niektórzy z naszych klientów mogą częściej nie podawać swojej daty urodzenia niż pozostali. Brak daty urodzenia nie jest losowy i może się okazać wartościową informacją. Rozwiązanie tego problemu polega na dodaniu do danych treningowych zmiennych binarnych `<Zmienna X>_IsMissing` i ustawieniu ich wartości na `1` dla przykładów, z których brakująca wartość zmiennej została usunięta, i na `0` dla pozostałych przykładów.

Usunąć brakujące wartości możemy:

- **Zastępując je stałą**, na przykład średnią, medianą lub dominantą. Ta metoda powinna być używana, jeżeli brakujących danych jest mało. Mało oznacza, że procent przykładów o brakującej wartości zmiennej jest nie większy niż 10%. Wiedząc na przykład, że najwięcej pasażerów zaokrętowało się w Southampton, możemy oznaczyć i uzupełnić dwie brakujące wartości zmiennej `Embarked` wartością `S`.

```
SELECT *
INTO Titanic.#Temp
FROM [Titanic].[Train];

ALTER TABLE Titanic.#Temp
ADD Embarked_IsMissing bit DEFAULT 0;

UPDATE Titanic.#Temp
SET Embarked_IsMissing = 0;

UPDATE Titanic.#Temp
SET [Embarked] = 'S', Embarked_IsMissing = 1
WHERE [Embarked] IS NULL;

SELECT PassengerId, Name, Embarked
FROM Titanic.#Temp
WHERE Embarked_IsMissing = 1;
```

PassengerId	Name	Embarked
62	Icard, Miss. Amelie	S
830	Stone, Mrs. George Nelson (Martha Evelyn)	S

- **Usuwając niekompletne przykłady**. To rozwiązanie może być zastosowane, jeżeli dysponujemy na tyle dużą liczbą przykładów, że możemy sobie pozwolić na jej zmniejszenie. Usuwając przykłady, również możemy zmienić rozkłady wartości, bo nie usuwamy ich losowo. Dlatego ta metoda powinna być stosowana wtedy, gdy mamy wystarczająco dużo danych treningowych, a procent usuwanych przykładów jest nie większy niż 20.

```
DELETE FROM Titanic.#Temp
WHERE [Age] IS NULL OR [Embarked] IS NULL;
```

- **Usuwając niekompletne zmienne**. Ta metoda jest zalecana w przypadku zmiennych, w których brakuje więcej niż 25% danych.

```
ALTER TABLE Titanic.#Temp
DROP COLUMN [Cabin];
```

- **Uzupełniając je na podstawie wartości innych zmiennych**. W najprostszym przypadku możemy uzupełnić brakujący wiek typowym wiekiem pasażerów i pasażerek.

```
UPDATE Titanic.#Temp
SET [Age] = (SELECT DISTINCT PERCENTILE_DISC(0.5) WITHIN GROUP (ORDER BY Age) OVER ()
                        FROM Titanic.#Temp
                        WHERE [Sex] ='female')
WHERE [Age] IS NULL AND [Sex] ='female';

UPDATE Titanic.#Temp
SET [Age] = (SELECT DISTINCT      PERCENTILE_DISC(0.5) WITHIN GROUP (ORDER BY Age) OVER ()
                        FROM Titanic.#Temp
                        WHERE [Sex] ='male')
WHERE [Age] IS NULL AND [Sex] ='male';
```

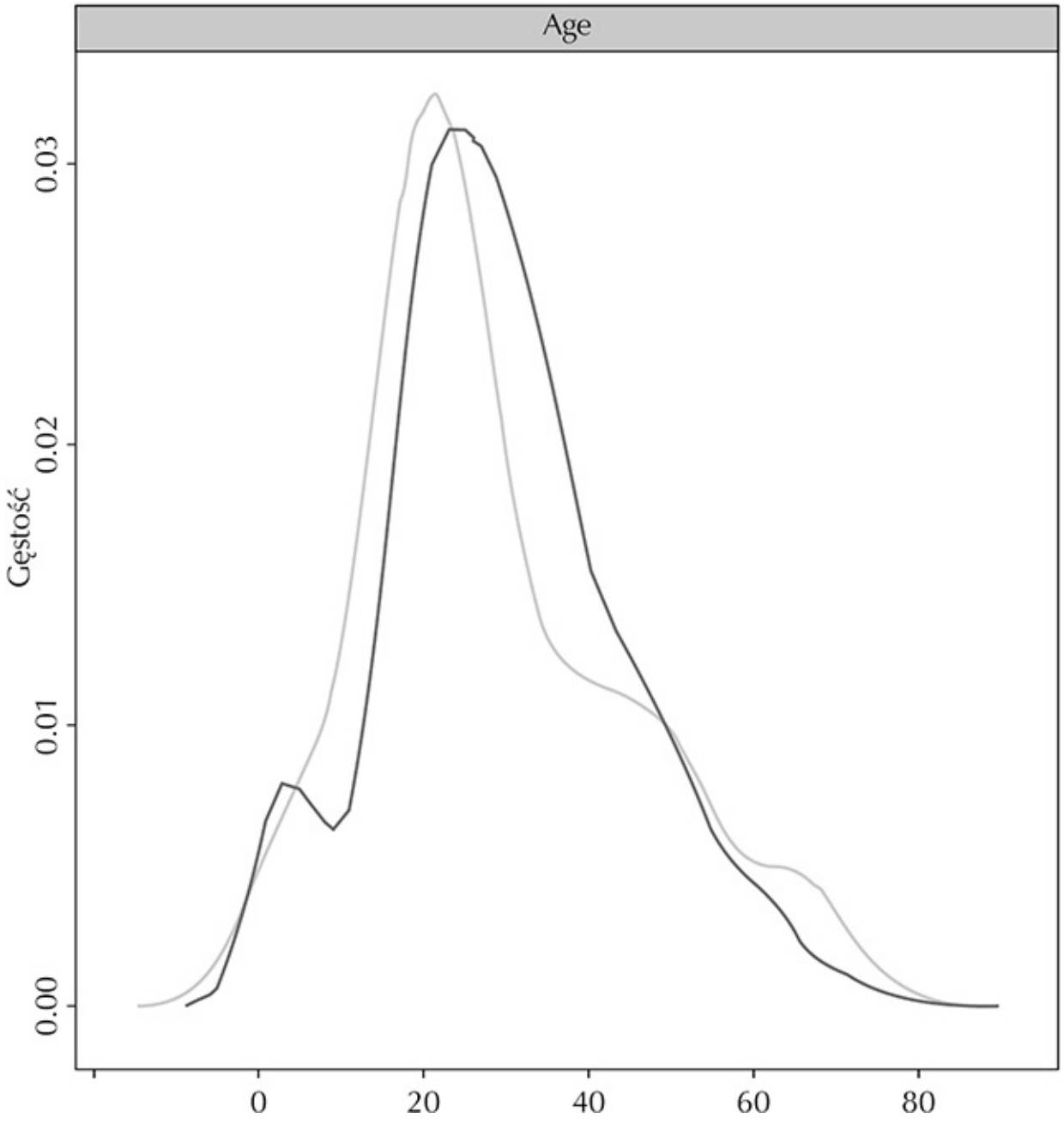
Możemy też użyć specjalistycznych funkcji uzupełniających brakujące wartości na podstawie zależności między zmiennymi, na przykład funkcji biblioteki `mice`. Funkcja `mice` generuje brakujące wartości – w tym przypadku utworzony został jeden model, w którym brakujące wartości zostały wyliczone metodą mapowania średnich. Do porównania rozkładów oryginalnej i uzupełnionej zmiennej została użyta funkcja `densityplot` (rys. 5.2). Funkcja `complete` uzupełniła brakujące wartości na podstawie wartości obliczonych w modelu. Na koniec porównałem statystyki opisowe oryginalnej i uzupełnionej zmiennej i uzupełnione dane zapisałem w tabeli.

```
connection_string <- "Driver=SQL Server; Server=MS; Database=ML; Trusted_Connection=yes"
train_table_name <- "[Titanic].[Temp]"
train_data_table <- RxSqlServerData(table = train_table_name,
                                    connectionString = connection_string)
train_table <- rxImport(train_data_table)

install.packages("mice")
library(mice)

temp_data <- mice(train_table,m=1,maxit=50,meth='pmm',seed=500)
densityplot(temp_data)
completed_data <- complete(temp_data,1)
summary(train_table$Age)
summary(completed_data$Age)
rxDataStep(inData = completed_data,
             outFile = train_data_table,
             overwrite = TRUE)
```

```
   Min. 1st Qu.   Median     Mean 3rd Qu.     Max.     NA's
    0.42    20.12    28.00    29.70    38.00    80.00      177
   Min. 1st Qu.   Median     Mean 3rd Qu.     Max.
    0.42    19.00    28.00    28.73    38.50    80.00
```


Age
Gęstość
0.00
0.01
0.02
0.03
0
20
40
60
80

Rysunek 5.2. Rozkład oryginalnej zmiennej pokazuje ciemniejsza linia, rozkład zmiennej z uzupełnionymi wartościami – jaśniejsza

5.3.2. Usuwanie duplikatów

Powtarzające się przykłady należy usunąć. Powtórzenia to wielokrotnie zapisane w danych te same przykłady. Powtarzające się kombinacje wartości zmiennych są czymś normalnym i nie należy ich usuwać. Z reguły powstają przez usunięcie zmiennych nieprzydatnych dla modelu, na przykład identyfikatorów. Jeżeli przyjmiemy, że nie było dwóch pasażerów o tym samym imieniu, płci, wieku i klasie, to ewentualne duplikaty usuniemy poniższą instrukcją.

```
WITH CTE ([Name],[Sex],[Age],[Pclass], DuplicateCount)
AS
(
  SELECT [Name],[Sex],[Age],[Pclass],
  ROW_NUMBER() OVER(PARTITION BY [Name],[Sex],[Age],[Pclass] ORDER BY [Name]) AS
DuplicateCount
  FROM Titanic.#Temp
)
DELETE FROM CTE
WHERE DuplicateCount>1;
```

5.3.3. Usuwanie błędnych danych

Błędne dane to dane niezgodne z naszą wiedzą. Wiedząc na przykład, ile kosztują taksówki w Nowym Jorku, za błędne dane uznamy opłaty przekraczające 200 dolarów. Znajomość nazw polskich miast pozwoliłaby nam z kolei za błąd uznać nazwę *Katowicze*.

Niespójne, niezgodne z przyjętymi regułami **dane również uważa się za błędne.** Przykładem może być ujemna kwota napiwku – przecież taksówkarze nie dają napiwków pasażerom. Innym przykładem niespójnych danych jest data

realizacji zamówienia wcześniejsza od daty jego złożenia.

Trzecim rodzajem błędnych danych są dane sprzeczne ze zdrowym rozsądkiem. Zerowa długość trasy nie ma sensu, chyba że taksówka nie ruszyła z miejsca, a pasażer zapłacił za trzaśnięcie drzwiami. Data urodzenia z przyszłości czy ujemny wiek klienta to inne przykłady błędnych, sprzecznych ze zdrowym rozsądkiem danych.

Opisana w poprzednim rozdziale analiza deskrypcyjna danych pozwala wykryć błędne dane. Wykryte błędy trzeba naprawić. **Błędne dane możemy skorygować albo usunąć.** Korekta jest lepszym rozwiązaniem, ale bardziej pracochłonnym i nie zawsze możliwym. Jeden jej sposób polega na przeanalizowaniu przykładów jeden po drugim z ekspertem z danej dziedziny. Drugi – na znalezieniu w innym źródle prawidłowych danych i zastąpieniu nimi błędnych.

Najczęściej żaden z powyższych sposobów poprawienia błędów nie jest możliwy. Wtedy powinniśmy usunąć przykłady zawierające błędne dane. Kierujemy się zasadą, że skoro wartość jednej zmiennej była błędna, to wartości wszelkich pozostałych zmiennych tego przykładu również powinny budzić nasze wątpliwości.

```
BEGIN TRAN
DELETE FROM [NYCTaxi].[Training]
WHERE [tip_amount] < 0
OR [total_amount] < 0
OR [total_amount] > 200
OR [trip_distance] < 0
OR [tip_amount] > [total_amount]
OR [pickup_datetime] < [dropoff_datetime]
OR [trip_time_in_secs] < 1
OR [trip_time_in_secs] > 20000
ROLLBACK TRAN
```

```
(1210025 rows affected)
```

5.3.4. Usuwanie wartości odstających

Wartości odstające to takie, które nie pasują do pozostałych, tj. są od nich o wiele za duże lub o wiele za małe. **Wartości odstające są mało przydatne dla modeli predykcyjnych.** Po pierwsze, w danych treningowych występują one na tyle rzadko, że model raczej niczego się na nich nie nauczy. Po drugie, szansa ich wystąpienia w danych testowych jest niewielka, a więc nawet gdyby model nauczył się czegoś na odstających wartościach, będzie to umiejętność mało przydatna. Jeżeli dodamy do tego wrażliwość niektórych algorytmów uczenia maszynowego (np. regresji liniowej) na wartości odstające, możemy dojść do wniosku, że wartości odstające należy usunąć.

Od tej reguły są dwa wyjątki. Po pierwsze, jeżeli zadaniem modelu jest przewidywanie nietypowych wartości, usunięcie wartości odstających zmiennej wyjściowej uniemożliwi mu nauczenie się najważniejszych reguł. Po drugie, jeśli przypuszczamy, że w przyszłości wartości, które w danych treningowych są odstające, będą się pojawiały częściej, usuwając je, pogorszymy jakość predykcji.

Problem wartości odstających można rozwiązać na kilka sposobów. Najpierw jednak musimy oznaczyć jako odstające wiersze tabeli. Użyjemy do tego procedury `[DQ].[MarkOutliersTable]`. Jej działanie polega na wywołaniu dla wszystkich kolumn numerycznych wskazanej tabeli procedury `[DQ].[MarkOutliersColumn]`. Ta procedura z kolei oblicza rozstęp ćwiartkowy dla kolumny i sprawdza, czy wartości mieszczą się w jego wielokrotności (wielokrotność podaje się jako drugi parametr wywołania procedury `[DQ].[MarkOutliersTable]`). Jeżeli w jakiejś komórce tabeli znajduje się wartość spoza typowego zakresu, informacja o tym zostaje dopisana do kolumny `OUTLIER`. Przykładowy wynik testu dla dwóch wierszy z kilkoma wartościami spoza trzykrotności rozstępu ćwiartkowego został pokazany poniżej.

```
ALTER TABLE [NYCTaxi].[Training]
ADD OUTLIER VARCHAR(4000);
GO
EXEC DQ.MarkOutliersTable 'NYCTaxi.Training',3;
GO
SELECT *
FROM [NYCTaxi].[Training]
WHERE [OUTLIER] IS NOT NULL;
```

```
7 column(s) checked in table NYCTaxi.Training.
[trip_distance] outside (-5, 9) , [fare_amount] outside (-17, 38) , [mta_tax] outside (1,
1) , [tolls_amount] outside (0, 0) , [total_amount] outside (-19, 44)
[trip_distance] outside (-5, 9) , [fare_amount] outside (-17, 38) , [mta_tax] outside (1,
1) , [tolls_amount] outside (0, 0) , [total_amount] outside (-19, 44) , [tip_amount]
outside (-6, 8)
```

Po przeanalizowaniu oznaczonych przykładów możemy zadecydować, jak usunąć odstające wartości. Możemy:

- **Usunąć nietypowe przykłady**. Jeżeli wartości niektórych zmiennych są mało prawdopodobne, takie przykłady nie tylko są mało przydatne, lecz także mogą być wynikiem błędu. Poniższe instrukcje usuwają, a następnie przywracają (żeby mieli Państwo do dyspozycji oryginalne dane) nietypowe wiersze tabeli `[NYCTaxi].[Training]`.

```
BEGIN TRAN
DELETE FROM [NYCTaxi].[Training]
WHERE [OUTLIER] LIKE '%[trip_distance]%outside%(-5, 9)%'
AND [OUTLIER] LIKE '%[total_amount]%outside%(-19, 44)%'
ROLLBACK TRAN
```

- **Zastąpić odstające wartości stałymi**, na przykład wartościami brzegowymi, użytymi do ich oznaczenia jako odstające. Również w tym przypadku zmiany zostaną automatycznie wycofane, żeby pozwolić Państwu pracować z oryginalnymi danymi

```
BEGIN TRAN
UPDATE [NYCTaxi].[Training]
SET trip_distance = 9
WHERE trip_distance > 9
AND [OUTLIER] LIKE '%[trip_distance]%outside%(-5, 9)%'
ROLLBACK TRAN
```

5.4.
Przekształcenie danych

Do zbudowania modelu uczenia maszynowego potrzebujemy czterech rzeczy: przykładów treningowych, algorytmu reprezentowania wiedzy, metody optymalizacji procesu uczenia i funkcji kosztu.

Opisane w poprzednim rozdziale metody analizy opisowej pozwalają nam zrozumieć dane. Z rozdziału 3 wiemy, że wiedza może być reprezentowana przez węzły drzew decyzyjnych albo parametry funkcji regresji. W dalszych rozdziałach poznamy inne, bardziej złożone algorytmy reprezentowania wiedzy. Poznamy też używane w praktyce metody optymalizacji procesu uczenia. Znamy też już dwie funkcje kosztu: często używaną do oceny regresorów funkcję najmniejszych kwadratów błędu L2 i stosowaną do oceny klasyfikatorów funkcję kosztu 0-1. W tym rozdziale skoncentrujemy się na przygotowaniu przykładów.

Skuteczność różnych metod optymalizacji procesu uczenia oraz jakość wyników poszczególnych funkcji kosztu silnie zależą od danych treningowych. Na przykład proces uczenia metodą najszybszego spadku gradientu jest najbardziej efektywny (uczenie trwa najkrócej i daje najlepsze wyniki) dla danych, których średnia wynosi 0. Inny przykład – metody podziału drzew decyzyjnych faworyzują zmienne kategoryczne o większej liczbie stanów (zmienne o dużej liczbie stanów mają nieproporcjonalnie duży wpływ na predykcje modelu). Z kolei funkcja minimalizująca kwadraty błędu L2 jest

bardzo wrażliwa na dane odstające (jedna skrajna wartość może całkowicie zmienić model). Przekształcenie danych ma na celu dostosowanie ich do algorytmu reprezentowania wiedzy, metody optymalizacji procesu uczenia i funkcji kosztu modelu uczenia maszynowego.

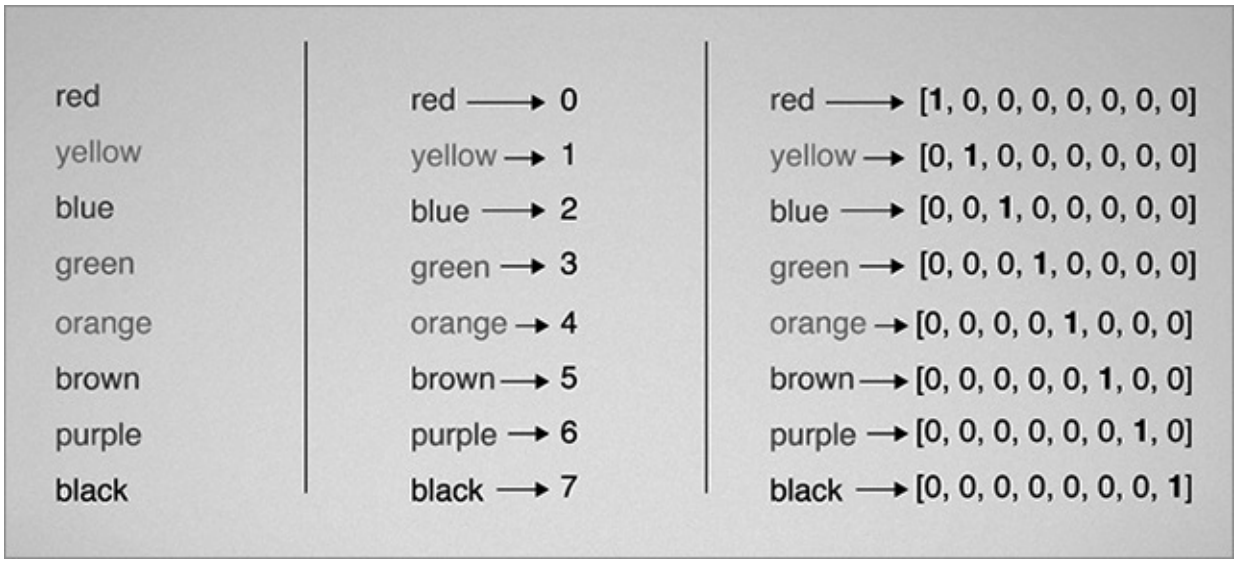
Dla modeli uczenia maszynowego absolutne wartości zmiennych nie są istotne, ważne są tylko względne wartości zmiennych i korelacje między nimi. Na przykład temperaturę można mierzyć według skal Celsjusza, Farenheita lub Kelwina. Normalna temperatura naszego ciała wynosi odpowiednio 36,6 stopni Celsjusza, 97,8 stopni Farenheita i 309,7 stopni Kelwina. Wszystkie te liczby równie dokładnie reprezentują normalną temperaturę ludzkiego ciała. To samo dotyczy zmiennych binarnych. Prawdę najczęściej reprezentuję się jako liczbę jeden, a fałsz jako liczbę zero. Moglibyśmy jednak zastąpić jedynkę liczbą 123,4 a fałsz liczbą –7. Utrudniłoby nam to wykonywanie pewnych operacji, na przykad zliczanie wartości, ale nie zmieniłoby to ich znaczenia. Stany zmiennych kategorycznych również są arbitralne. Zmienna kolor na przykład może przyjmować wartości Biały, Czarny i Czerwony albo White, Black i Red. Zmieniając wartości zmiennych, możemy ułatwić albo utrudnić modelom uczenie, nic ponadto.

5.4.1. Kodowanie

Zmienne numeryczne ocenia się i przetwarza łatwiej niż kategoryczne, bo ich wartości można ze sobą porównywać i obliczać, nawet nie znając kontekstu. Mamy więc do dyspozycji wiele metod matematycznych i statystycznych. Niekiedy korzyść z możliwości ich zastosowania jest na tyle duża, że zmienne kategoryczne koduje się jako numeryczne. Niektóre algorytmy uczenia maszynowego, w tym algorytmy stosowane w modelach regresji, wymagają zmiennych numerycznych. W ich przypadku, jeżeli któraś ze zmiennych jest kategoryczna, zostanie automatycznie przekształcona na numeryczną albo próba wytrenowania modelu skończy się błędem.

Najczęściej zmienne kategoryczne koduje się metodą jeden do wielu. W elektronice ten sposób kodowania nazywa się kodowaniem z gorącą jedynką (ang. *one-hot encoding*). Polega on na zastąpieniu zmiennej kategorycznej tyloma zmiennymi binarnymi, ile stanów przyjmowała kodowana zmienna. Dla danego przykładu tylko jedna z tak utworzonych zmiennych binarnych

przyjmuje wartość 1 – ta, która reprezentuje faktyczny stan kodowanej zmiennej. Wartości pozostałych zmiennych binarnych ustawia się na 0 (rys. 5.3).



Rysunek 5.3. Każda ze zmiennych binarnych reprezentuje prawdopodobieństwo wystąpienia jednego stanu zakodowanej zmiennej. Prawdopodobieństwa wszystkich stanów wynoszą 0, z wyjątkiem prawdopodobieństwa stanu, który rzeczywiście wystąpi, które wynosi 1

Kodowanie metodą *jeden do wielu* nie wymaga wiedzy z dziedziny eksperymentu i może być przeprowadzone automatycznie. Ponieważ w wyniku dostajemy zmienne, z których każda zawiera część informacji zakodowanej zmiennej, nazywa się je atrapami (ang. *dummies*).

```
SELECT [Embarked],
           IIF([Embarked] = 'S', 1, 0) AS [Southampton],
           IIF([Embarked] = 'C', 1, 0) AS [Cherbourg],
           IIF([Embarked] = 'Q', 1, 0) AS [Queenstown]
FROM [Titanic].[Train];
```

Embarked	Southampton	CherbourgQueenstown
C	0	1 0
S	1	0 0
S	1	0 0
Q	0	0 1
S	1	0 0
…		

Zaletą takiego kodowania jest zachowanie rozkładu stanów oryginalnej zmiennej. W powyższym przykładzie, skoro 72% pasażerów zaokrętowało się w Southampton, to zmienna `Southampton` przyjmie wartość 1 w 72% przykładów.

Kodowanie metodą jeden do wielu ma też wady, które powodują, że jest ono mało przydatne dla zmiennych przyjmujących wiele (więcej niż 20) stanów. Po pierwsze, znacznie zwiększa liczbę zmiennych wejściowych. Po drugie, rzadziej występujące stany będą reprezentowane przez zmienne przyjmujące wartość 1 dla tak małej liczby przykładów, że przydatność tych zmiennych atrap dla modelu będzie znikoma. Dlatego przed zakodowaniem metodą jeden do wielu zmiennych kategorycznych przyjmujących wiele stanów należy te zmienne uogólnić.

Skoro zmienną przyjmująca trzy stany można zakodować jako trzy zmienne atrapy, to ile atrap potrzeba do zakodowania zmiennej binarnej? Przeanalizujmy ten problem na przykładzie zmiennej `Survived`. Na pewno możemy ją zakodować dwoma zmiennymi `Died` i `Survived`. Pierwsza będzie miała wartość 1, jeżeli pasażer zginął w katastrofie, a druga przyjmie wartość 1, jeżeli pasażer został uratowany. Analizując te dwie zmienne, zauważymy, że można je zastąpić jedną zmienną binarną `Survived` – druga zmienna nie zawiera żadnych informacji, których nie byłoby w pierwszej. Kategoryczną zmienną binarną możemy przekształcić w numeryczną, kodując jej stany jako liczby – 1 będzie oznaczało, że pasażer przeżył, 0, że zginął. W przykładowym zbiorze danych zmienna `Survived` jest właśnie tak zakodowana. **Metoda zamieniania stanów zmiennej kategorycznej liczbami nazywa się kodowaniem stanów.**

Zmienne porządkowe również można przekształcić na numeryczne metodą kodowania stanów. W ten sposób nowa zmienna będzie przyjmowała tyle samo stanów, co oryginalna, ale będą one liczbami reprezentującymi kolejne stany. Jeżeli na przykład posortujemy porty zaokrętowania w kolejności rejsu,

otrzymamy poniższy wynik.

```
SELECT [Embarked],
       CASE WHEN [Embarked] = 'S' THEN 0
            WHEN [Embarked] = 'C' THEN 1
            ELSE 2 END
AS EmbarkedEncoded
FROM [Titanic].[Train];
```

Embarked	EmbarkedEncoded
Q	2
S	0
C	1
C	1
S	0
…	

W niektórych przypadkach regularne zmienne kategoryczne również zamienia się na numeryczne, kodując ich stany. Decydując się na takie rozwiązanie, powinniśmy pamiętać, że wprowadzamy do modelu nową, nieprawdziwą informację (o kolejności stanów), która może pogorszyć jego jakość.

5.4.2. Generalizacja

Zmienne kategoryczne przyjmujące dużo (kilkadziesiąt i więcej) stanów są problematyczne. Duża liczba stanów oznacza z reguły, że większość – a przynajmniej niektóre z nich – występują rzadko, za rzadko, żeby w danych treningowych znalazły się przykłady kombinacji tych stanów z typowymi wartościami innych zmiennych. Takie **rzadko występujące stany przypominają wartości odstające**. Skoro wartości odstające usunęliśmy z danych treningowych, to może tak samo powinniśmy zrobić z rzadko występującymi stanami zmiennych kategorycznych?

Najpierw jednak należy przyjrzeć się zmiennej i zastanowić, czy możemy sensownie (zgodnie z wiedzą z danej dziedziny) uogólnić rzadko występujące

stany, czyli zastąpić wiele stanów jednym. Często okazuje się, że przynajmniej część rzadziej występujących stanów jest wynikiem błędów, na przykład literówek. Albo są one efektem zapisania tej samej informacji na kilka różnych sposobów.

```
WITH Titles AS
       (SELECT SUBSTRING([Name], CHARINDEX(',', [Name])+2
                ,CHARINDEX('.',[Name],CHARINDEX(',', [Name])+2) - CHARINDEX(',',
[Name])-2)      AS Title
       FROM [Titanic].[Train])
SELECT Title, count(*) AS cnt
FROM Titles
GROUP BY Title
ORDER BY cnt;
```

Title	cnt
Capt	1
Don	1
Jonkheer	1
Lady	1
Mme	1
Ms	1
Sir	1
the Countess	1
Mlle	2
Major	2
Col	2
Rev	6
Dr	7
Master	40
Mrs	125
Miss	182
Mr	517

W 96% przykładów zmienna `Title` przyjmuje jeden z czterech najczęściej występujących stanów (*Pan*, *Pani*, *Panna* i *Panicz*). Pozostałe 13 stanów występuje na tyle rzadko, że powinniśmy je uogólnić.

Błędne stany `Mlle` i `Ms` zastąpimy stanem `Miss`, a błędny stan `Mme` stanem `Mr`.

Pozostałe z rzadko występujących stanów zastąpimy jednym stanem `Rare`.

```
WITH Titles AS
       (SELECT SUBSTRING([Name], CHARINDEX(',', [Name])+2
       ,CHARINDEX('.',[Name],CHARINDEX(',', [Name])+2) - CHARINDEX(',', [Name])-2) AS
Title
       FROM [Titanic].[Train]),
Cleaned AS
       (SELECT CASE Title
                WHEN 'Mme' THEN 'Mr'
                WHEN 'Mlle' THEN 'Miss'
                WHEN 'Ms' THEN 'Miss'
                WHEN 'Miss' THEN 'Miss'
                WHEN 'Mr' THEN 'Mr'
                WHEN 'Mrs' THEN 'Mrs'
                WHEN  'Master' THEN 'Master'
                ELSE 'Rare' END AS Title
       FROM Titles)
SELECT Title, count(*) AS cnt
FROM Cleaned
GROUP BY Title
ORDER BY cnt;
```

```
Title   cnt
Rare    23
Master  40
Mrs     125
Miss    185
Mr      518
```

5.4.3. Zaokrąglanie

Przekształceniem podobnym do generalizacji, ale dotyczącym zmiennych numerycznych, jest zaokrąglanie. Zauważmy, że prowadzi ono do zastąpienia zmiennej ciągłej (zmiennej o nieskończonej mocy zbioru wartości) zmienną dyskretną (zmienną numeryczną o skończonym zbiorze wartości). Taka zmiana upraszcza model i zmniejsza ryzyko jego nadmiernego dopasowania.

Zaokrąglać możemy do części dziesiętnych, jednostek, setek czy milionów, często zmieniając w ten sposób jednostkę zmiennej.

```
SELECT ROUND([Fare],0) as Fare
FROM [Titanic].[Train];
```

W ten sam sposób możemy zaokrąglić pomiary czasu wyrażone w sekundach i podać je w minutami.

```
SELECT [trip_time_in_secs], [trip_time_in_secs]/60 AS trip_time_in_mins
FROM [NYCTaxi].[Training];
```

trip_time_in_secs	trip_time_in_mins
296	4
334	5
308	5
…	

Zaokrąglić należy zmienne, które podejrzewamy o wyrażenie w na tyle małej jednostce, że ich wartości są obarczone dużym ryzykiem błędu pomiaru. Na przykład mierzenie czasu przejazdu taksówką w sekundach wydaje się dość ryzykowne. Możemy się obawiać, że spora część tak zmierzonych czasów pojazdów obarczona jest błędem wynoszącym od kilku do kilkudziesięciu sekund.

5.4.4. Dyskretyzacja

Jeżeli zaokrąglenie jest sposobem upraszczania modelu, to czemu nie pójść o krok dalej i nie zredukować jeszcze bardziej liczby wartości zmiennej numerycznej? Czasami takie przekształcenie potrafi zamienić mało przydatną zmienną wejściową w zmienną o dużej sile predykcyjnej.

Redukcja liczby wartości zmiennej numerycznej polega na jej przekształceniu w zmienną kategoryczną porządkową. W tym celu

przyporządkowuje się nowym stanom wskazane przedziały wartości oryginalnej zmiennej. Dyskretyzację można przeprowadzić na różne sposoby; trzy najpopularniejsze to: podział na przedziały stałej szerokości, podział na równoliczne przedziały i podział na przedziały o zdefiniowanych wartościach brzegowych.

Siła predykcyjna zmiennej zależy od jej korelacji z innymi zmiennymi, w szczególności od korelacji ze zmienną wyjściową. W poprzednim rozdziale wyjaśniłem, że siłę predykcyjną zmiennych i stopień ich korelacji można ocenić na podstawie rozkładu wartości zmiennej oraz mierząc informację zawartą w zmiennej. Dyskretyzacja jest przekształceniem, które może znacząco zmienić rozkład wartości zmiennej oraz zredukować ilość zawartej w niej informacji. Pamiętajmy, że teoretyczne maksimum informacji, jaka może być zawarta w zmiennej, zależy od liczby jej stanów – im więcej różnych wartości może przyjąć zmienna, tym większa jej pojemności informacyjna. Redukując liczbę stanów, zmniejszamy ilość informacji zawartej w zmiennej. **Dzieląc zmienną na przedziały stałej szerokości, staramy się zachować rozkład zmiennej. Podział na przedziały o takiej samej liczbie przykładów ma na celu zachowanie jej entropii, a podział na przedziały o zdefiniowanych wartościach służy do wprowadzenia do modelu wiedzy z dziedziny eksperymentu.**

Podział na przedziały stałej szerokości wymaga określenia liczby przedziałów. Od tej decyzji w dużym stopniu zależy liczność poszczególnych przedziałów (to, ile w każdym z nich znajdzie się przykładów). Niestety, nie można z góry przewidzieć, jaka powinna być liczba przedziałów. Sam podział nie jest skomplikowany – wystarczy podzielić rozstęp zmiennej przez ustaloną liczbę przedziałów. Skoro szerokość przedziału wyznacza się na podstawie wartości minimalnej i maksymalnej, czyli statystyk czułych na wartości odstające, to dyskretyzacja metodą podziałów stałej szerokości również będzie podatna na wartości odstające.

```
DECLARE @binwidth AS NUMERIC(5,2), @min AS INT;
SELECT @min = MIN(Fare),
 @binwidth = 1.0 * (MAX(Fare) - MIN(Fare)) / 6
FROM[Titanic].[Train];
WITH EWB AS
(SELECT
 CASE
  WHEN Fare >= @min + 0 * @binwidth AND Fare < @min + 1 * @binwidth
   THEN CAST((@min + 0 * @binwidth) AS VARCHAR(8)) + ' - ' +
        CAST((@min + 1 * @binwidth) AS VARCHAR(8))
  WHEN Fare >= @min + 1 * @binwidth AND Fare < @min + 2 * @binwidth
   THEN CAST((@min + 1 * @binwidth) AS VARCHAR(8)) + ' - ' +
        CAST((@min + 2 * @binwidth) AS VARCHAR(8))
  WHEN Fare >= @min + 2 * @binwidth AND Fare < @min + 3 * @binwidth
   THEN CAST((@min + 2 * @binwidth) AS VARCHAR(8)) + ' - ' +
        CAST((@min + 3 * @binwidth) AS VARCHAR(8))
  WHEN Fare >= @min + 3 * @binwidth AND Fare < @min + 4 * @binwidth
   THEN CAST((@min + 3 * @binwidth) AS VARCHAR(8)) + ' - ' +
        CAST((@min + 4 * @binwidth) AS VARCHAR(8))
  WHEN Fare >= @min + 4 * @binwidth AND Fare < @min + 5 * @binwidth
   THEN CAST((@min + 4 * @binwidth) AS VARCHAR(8)) + ' - ' +
        CAST((@min + 5 * @binwidth) AS VARCHAR(8))
  ELSE CAST((@min + 5 * @binwidth) AS VARCHAR(8)) + ' + '
 END AS FareEWB
FROM [Titanic].[Train])
SELECT FareEWB, count(*) AS cnt
FROM EWB
GROUP BY FareEWB;
```

```
FareEWB          cnt
0.00 - 85.39     825
85.39 - 170.78   46
170.78 - 256.17  11
256.17 - 341.56  6
426.95 +         3
```

Wynikiem podziału zmiennej `Fare` na 6 przedziałów stałej szerokości jest zmienna, która przyjmuje tylko 5 stanów (przedostatni przedział okazał się pusty) i której 92% przykładów trafiło do pierwszego przedziału. Przekonaliśmy się, że zmienne o takim rozkładzie jak `Fare` nie powinny być dzielone na przedziały stałej szerokości, w każdym razie nie przed usunięciem wartości odstających.

Podział na równoliczne przedziały również wymaga podania liczby przedziałów. Ta metoda jest częściej stosowana, bo jest niewrażliwa na rozkład wartości zmiennej. Jaki by on nie był, unikniemy utworzenia przedziału zawierającego większość przykładów i przedziałów pustych.

```
WITH EHB AS
(SELECT Fare, CAST(NTILE(6) OVER(ORDER BY Fare) AS CHAR(1)) AS FareEHB
        FROM [Titanic].[Train])
SELECT MIN(Fare) AS min, MAX(Fare) AS max, FareEHB, COUNT(*) AS cnt
FROM EHB
GROUP BY FareEHB
ORDER BY FareEHB;
```

min	max	FareEHB	cnt
0	7.775	1	149
7.775	8.6625	2	149
8.6625	14.4542	3	149
14.4583	26	4	148
26	52.5542	5	148
52.5542	512.3292	6	148

Tak podzielona zmienna przyjmuje 6 wartości, jej entropia jest bliska maksymalnej (trzy pierwsze stany występują minimalnie częściej niż pozostałe), ale jej rozkład zupełnie nie przypomina rozkładu oryginalnej zmiennej.

Trzecia metoda podziału wymaga zdefiniowania poszczególnych przedziałów. Najczęściej stosuje się ją wtedy, gdy podział wartości zmiennej na określone przedziały ma uzasadnienie biznesowe. Na przykład firma może określać klientów młodszych niż 8-letnich jako dzieci, osoby w wieku od 8 do 15 lat jako

nastolatków, osoby między 15 a 23 rokiem życia jako młodzież, klientów w wieku 23–35 lat jako osoby w średnim wieku, a osoby powyżej 35 roku jako osoby w wieku dojrzałym.

Gdybyśmy wiedzieli, że w 1912 r. bilet na statek uważany był za bardzo tani, jeśli kosztował mniej niż 8 funtów, bilety w cenie nie większej niż 14,5 funta były tanie, jeżeli na bilet trzeba było wydać nie więcej niż 35 funtów był on umiarkowanie drogi, a bilety powyżej tej kwoty odbierane były jako bardzo drogie, moglibyśmy zgodnie z tymi informacjami podzielić wartości zmiennej `Fare`.

```
WITH CB AS
 (SELECT CASE
       WHEN Fare <= 8 THEN 0
       WHEN Fare <= 14.5 THEN 1
       WHEN Fare <= 35 THEN 2
       ELSE 3 END AS FareCB
 FROM [Titanic].[Train])
SELECT FareCB, COUNT(*) as cnt
FROM CB
GROUP BY FareCB
ORDER BY FareCB;
```

```
FareCB  cnt
0               241
1               216
2               236
3               198
```

5.4.5. Skalowanie

Skalowanie zmiennych numerycznych może mieć na celu sprowadzenie tych zmiennych do wspólnej skali, często do zakresu wartości między zerem a jedynką. Zmienne skaluje się również po to, żeby zmienić ich rozkład, na przykład sprowadzić je do standardowego rozkładu normalnego.

Niektóre algorytmy uczenia maszynowego, takie jak przedstawiony w następnym rozdziale algorytm grupowania metodą k-średnich, są bardzo wrażliwe na różnice w skalach zmiennych. Takie algorytmy nadmiernie faworyzują zmienne o dużym rozstępie wartości kosztem zmiennych o małym rozstępie. Na przykład zmienna `YearlyIncome` przyjmująca wartości od –100 000 do 5 000 000 zdominowałaby zmienną `Children` o wartościach z przedziału od 0 do 10. Te zmienne należałoby przeskalować, chyba że użyty algorytm uczenia maszynowego zrobi to za nas.

 Sztuczne sieci neuronowe są tak wrażliwe na różnice w skalach zmiennych wejściowych, że automatycznie je skalują. Taka jest domyślna konfiguracja obu opisanych w dalszych rozdziałach implementacji sztucznych sieci neuronowych.

W pierwszej kolejności przyjrzymy się **skalowaniu metodą Min–Max, czyli normalizacji**. Polega ona na obliczeniu wartości minimalnej i maksymalnej, a następnie odjęciu od oryginalnej wartości minimum i podzieleniu wyniku przez rozstęp

$$X_{\text{norm}} = \frac{X - \min(X)}{[\max(X) - \min(X)]}$$

Otrzymamy w ten sposób wartości z zakresu od 0 do 1 (lub od –1 do 1, jeśli występują wartości ujemne), zachowując oryginalny rozkład danych.

```
SELECT Age, (Age - MIN(Age) OVER() )/(MAX(Age) OVER() - MIN(Age) OVER()) AS normAge
FROM [Titanic].[Train];
```

```
Age     normAge
22      0.271173661724051
38      0.472229203317416
26      0.321437547122393
35      0.43453128926866
…
```

Skalowanie metodą *Min–Max* jest wrażliwe na wartości odstające. Jeżeli one występują, powinniśmy użyć statystyki niewrażliwej na wartości odstające, czyli

rozstępu ćwiartkowego. **Skalowanie IQR polega na zastąpieniu wartości najmniejszej pierwszym kwartylem, a rozstępu – rozstępem ćwiartkowym**

$$X_{\text{robust}} = \frac{X - Q1(X)}{[Q3(X) - Q1(X)]}$$

```
SELECT Age, (Age -  PERCENTILE_CONT(0.25) WITHIN GROUP (ORDER BY 1.0*Age) OVER ())
       /(PERCENTILE_CONT(0.75) WITHIN GROUP (ORDER BY 1.0*Age) OVER () -
        PERCENTILE_CONT(0.25) WITHIN GROUP (ORDER BY 1.0*Age) OVER ()) AS robustAge
FROM [Titanic].[Train]
ORDER BY NEWID();
```

```
Age     robustAge
23      0.160839160839161
43      1.27972027972028
30      0.552447552447552
21      0.048951048951049
62      2.34265734265734
…
```

Skalowanie nie musi być przekształceniem liniowym. Często normalizuje się dane właśnie po to, żeby zmienić ich rozkład. Z reguły używa się do tego nieliniowych funkcji silnie spłaszczających wartości bardziej odległe od średnich, na przykład funkcji eksponencjalnej, czyli funkcji wykładniczej o podstawie logarytmu naturalnego. Otrzymamy w ten sposób **skalowanie SoftMax**.

```
SELECT Age, 1/(1+POWER(2.718281828, -Age)) AS smAge
FROM [Titanic].[Train];
```

```
Age     smAge
27      1.000000000000000000000000000
14      0.999999168000692223424070111
58      1.000000000000000000000000000
20      0.999999998000000039999999920
…
```

Ponieważ stała *e* (w przybliżeniu *2,72*) jest podnoszona do potęgi o ujemnej wartości (*–x*), wartość w mianowniku jest tym mniejsza, im większa była

wartość *x*, a dodanie do mianownika 1 gwarantuje nam, że znormalizowana wartość nie będzie większa niż 1. W rezultacie wartości bliskie średniej zostaną przeskalowane prawie liniowo, a wartości odstające – do wartości bliskich 0 lub 1.

Innym, popularnym przykładem nieliniowego skalowania jest **skalowanie logarytmiczne**. Stosuje się go do spłaszczenia rozkładu zmiennych przyjmujących odstające wartości. Skalowana zmienna nie może przyjmować wartości ujemnych. Najczęściej używa się logarytmów naturalnych (logarytmów o podstawie e) lub logarytmów dziesiętnych (logarytmów o podstawie 10).

```
SELECT Age,LOG(Age) AS logeAge, LOG(Age,10) AS log10Age
FROM [Titanic].[Train];
```

Age	logeAge	log10Age
54	3.98898404656427	1.73239375982297
2	0.693147180559945	0.301029995663981
27	3.29583686600433	1.43136376415899
14	2.63905732961526	1.14612803567824
4	1.38629436111989	0.602059991327962
…		

W poprzednim rozdziale poznaliśmy rozkład normalny i jego ustandaryzowaną wersję. **Przeskalowanie zmiennej tak, aby otrzymać zmienną o standardowym rozkładzie normalnym** (rozkładzie o średniej zero i odchyleniu standardowym jeden) **nazywa się standaryzacją**

$$Z = \frac{X - \bar{X}}{\sigma}$$

Po ustandaryzowaniu wartości mniejsze od średniej będą miały wartość ujemną, a wartości większe od średniej – dodatnią. Z reguły ustandaryzowane wartości mieszczą się w przedziale od –4 do 4, czyli nie przekraczają odległości czterech odchyleń standardowych od średniej.

```
SELECT Age, (Age - AVG(AGE) OVER())/STDEV(AGE) OVER() AS stAge
FROM [Titanic].[Train];
```

Age	stAge
54	1.67286592197629
2	-1.90679948602643
27	-0.185806501409734
14	-1.08072285341041
4	-1.76912004725709
58	1.94822479951497

5.4.6. Wygładzanie

W dotychczasowych przykładach nie interesowały nas zależności między wcześniejszymi a bieżącymi wartościami zmiennych. Nie badaliśmy, czy i w jakim stopniu wartości zmiennych zależą od ich poprzednich wartości, czy na przykład dzisiejsza temperatura jest skorelowana z wczorajszą i przedwczorajszą, albo czy na dzisiejszy kurs akcji firmy Microsoft miał wpływ zeszłotygodniowy kurs dolara amerykańskiego. Sytuacja się zmienia, jeżeli naszym zadaniem jest przewidzenie przyszłych wartości zmiennych, czyli postawienie prognozy. W takim przypadku należy wziąć pod uwagę dynamikę badanego procesu.

Prognozowanie wymaga przygotowania danych w postaci szeregów czasowych, czyli uporządkowania zmiennych numerycznych według wartości klucza reprezentującego czas. Wartości klucza muszą być monotonicznie rosnące, ale nie muszą to być dane typu daty i czasu. Do ilustracji omawianego zagadnienia użyjemy zmierzonych przez jeden miernik (miernik `s12`) parametrów pracy silnika samolotowego. Kluczem naszego szeregu czasowego jest numer cyklu pracy tego silnika. Kompletny przykład, w którym użyjemy większej ilości zmiennych do prognozowania awarii silników, znajdą Państwo w kolejnych rozdziałach. W tym miejscu ograniczymy się do omówienia metod oceny i przygotowania szeregów czasowych. Oto uproszczone, przykładowe

dane.

```
SELECT [cycle] ,[s12]
FROM [ML].[PredictiveMaintenance].[PM_Train]
WHERE [id] = 1
ORDER BY [cycle];
```

cycle	s12
1	521.66
2	522.28
3	522.42
4	522.86
5	522.19
6	521.68
7	522.32
8	522.47
9	521.79
10	521.79
…	

Szeregi czasowe muszą być kompletne. Kompletny szereg czasowy to taki, w którym:

- nie brakuje ani jednej wartości klucza, czyli, jeżeli jego pierwszą (najmniejszą) wartością jest 1, a ostatnią (największą) 500, to nie brakuje nam ani jednej wartości klucza z zakresu od 1 do 500;
- nie brakuje ani jednej wartości zmiennej, czyli dla każdej wartości klucza mamy wartość zmiennej zmierzonej w tym czasie;
- wszystkie pomiary były zebrane w takich samych odstępach czasu – oznacza to, że przedziały czasu między kolejnymi wartościami klucza muszą być takie same, jeżeli na przykład między wartością 10 a 11 upłynęła 1 godzina, to wszystkie inne wartości również muszą zostać zebrane w godzinnych odstępach.

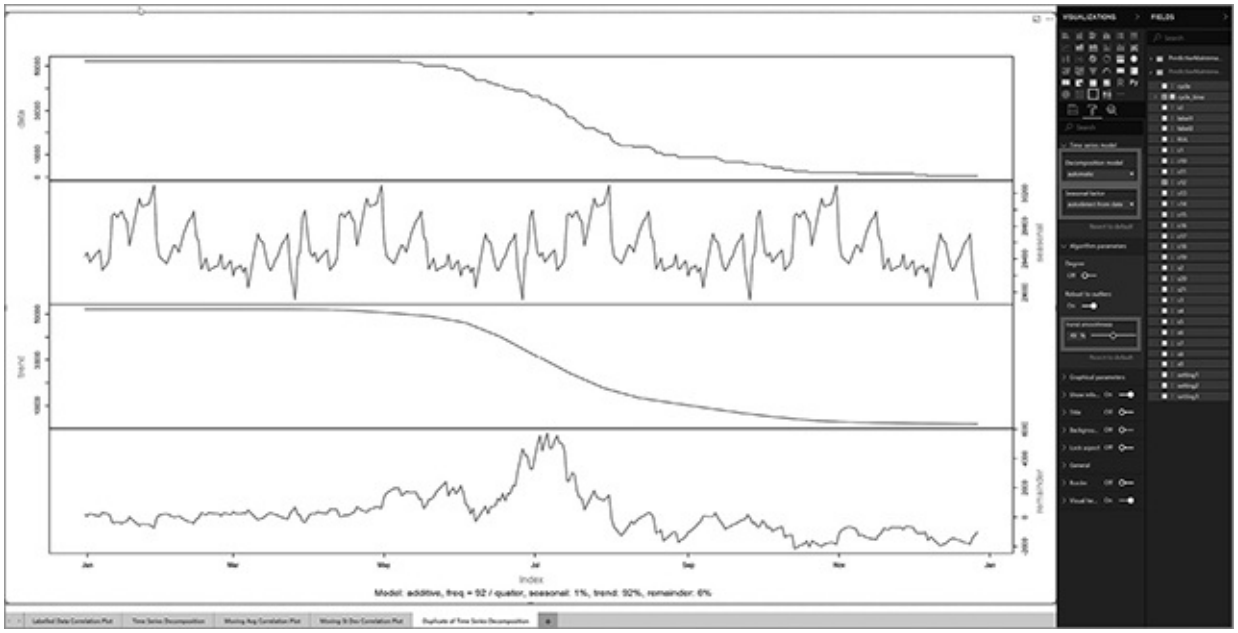
Próby uzupełniania niekompletnych szeregów czasowych przez pominięcie w nich wierszy dla brakujących danych i przenumerowaniu klucza tak, żeby nie

zawierał luk, prowadzą do zniekształcenia danych. Oto przykład. Co godzinę zbieramy dane o pracy urządzenia. Ostatni wykonany w niedzielę pomiar ma numer 240. Jeżeli z powodu awarii miernika brakuje nam danych z poniedziałku (czyli brakuje nam 24 pomiarów), nie powinniśmy wstawić pierwszej wartości zmierzonej po usunięciu awarii z kolejną wartością klucza (wartością 241). Jeżeli tak zrobimy, to przedział czasu między pomiarami o numerach 240 i 241 będzie wynosił 25 godzin, a nie 1 godzinę jak między pozostałymi pomiarami. W ten sposób zniekształcimy dwie najważniejsze dla modeli uczenia maszynowego składowe szeregu czasowego, czyli jego trend i cykl.

Szeregi czasowe zawierają trzy składowe: trend, cykl i szum. Trend reprezentuje monotoniczny składnik serii danych, czyli jest rosnący, malejący albo stały. Cykl reprezentuje okresowe (sezonowe) zmiany wartości. Szum jest tym, co pozostaje po usunięciu z szeregu czasowego trendu i cyklu. Prognozowanie opiera się na trendzie i cyklu.

Trend, cykl i szum możemy uzyskać, odejmując je od oryginalnych danych albo dzieląc oryginalne dane na trzy składowe. W pierwszym przypadku mamy do czynienia z modelem addytywnym, w którym wartości obserwacji są sumą ich składowych, a w drugim – z modelem multiplikatywnym, w którym wartości obserwacji są iloczynem ich składowych.

Prawidłowa ocena modelu wymaga określenia długości jego cyklów. Można to zrobić, przeprowadzając analizę Fouriera, my jednak skorzystamy z niestandardowej wizualizacji Power BI o nazwie *Time Series Decomposition* (rys. 5.4).



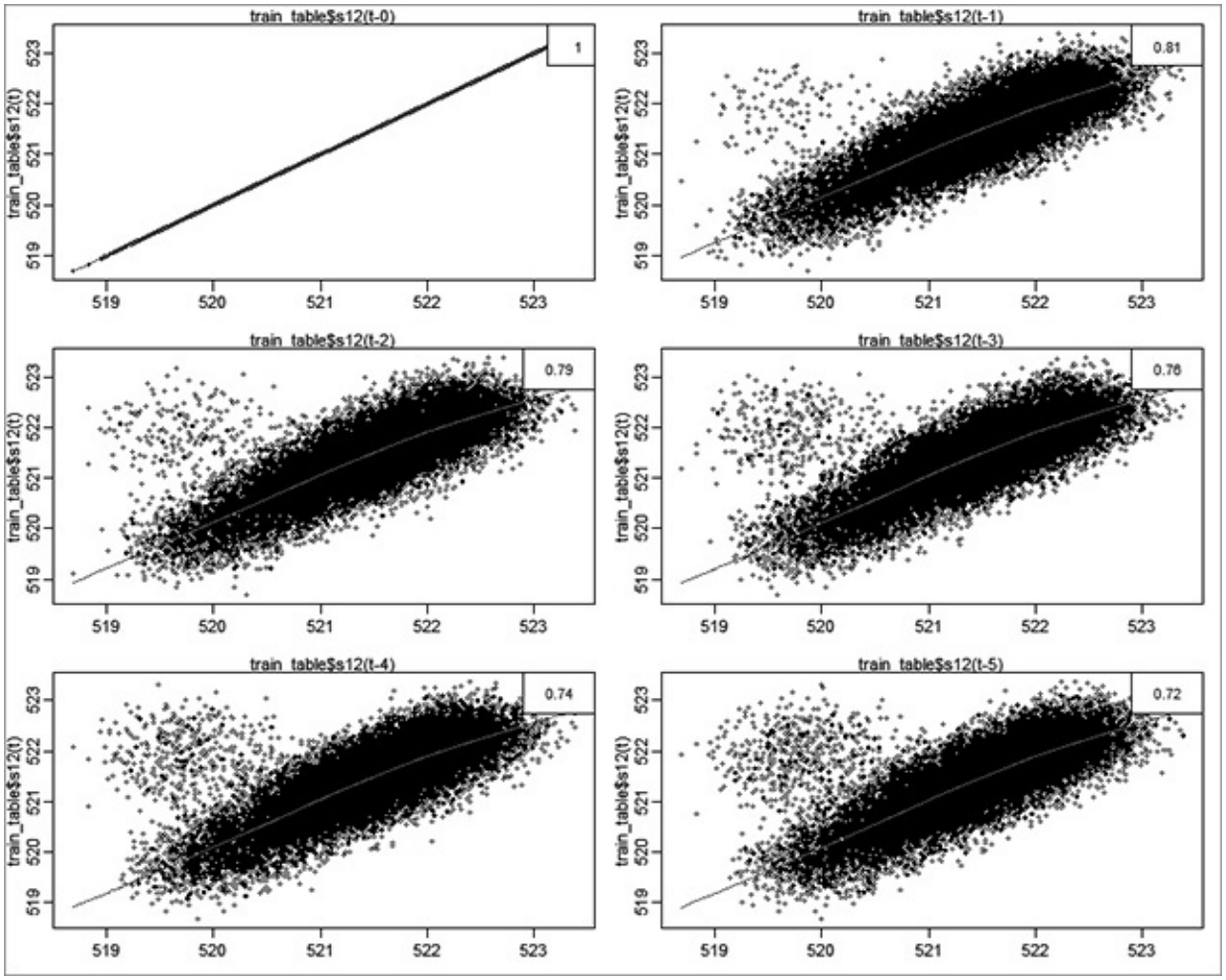
Rysunek 5.4. Oryginalny szereg czasowy znajduje się na górze, poniżej widzimy składową cykliczną, pod nią trend, a na dole szum. Do dekompozycji został użyty, automatycznie wybrany, model addytywny, długość cyklu również została automatycznie znaleziona. Ten model zredukował szum do poziomu 6%

Zwróćmy uwagę na trzecią z zaznaczonych opcji wykresu – opcję wygładzania trendu (ang. *trend smoothness*). Wyraźnie widoczny w danych trend został wygładzony, dzięki czemu został zmniejszony wpływ poszczególnych obserwacji. **Wygładzanie szeregów czasowych** (w tym przypadku trendu) **polega na lokalnym uśrednianiu danych**, tak żeby niesystematyczne składniki poszczególnych obserwacji nawzajem się znosiły.

Najbardziej powszechna technika wygładzania polega na użyciu średniej ruchomej, czyli na zastąpieniu poszczególnych wartości szeregu czasowego przez zwykłą lub ważoną średnią *n* sąsiadujących wartości, gdzie *n* jest szerokością okna wygładzania. Ma to na celu wzmocnienie siły predykcyjnej zmiennej przez zredukowanie występującego w niej szumu.

Jak wygładzenie może zwiększyć przydatność zmiennej dla modelu? W szeregach czasowych występują nie tylko korelacje między zmiennymi, lecz także autokorelacje, czyli korelację między kolejnymi (opóźnionymi) wartościami tej samej zmiennej. Autokorelację możemy zmierzyć i przedstawić

na wykresie za pomocą wchodzącej w skład biblioteki `astsa` funkcji `lag2.plot` (rys. 5.5).

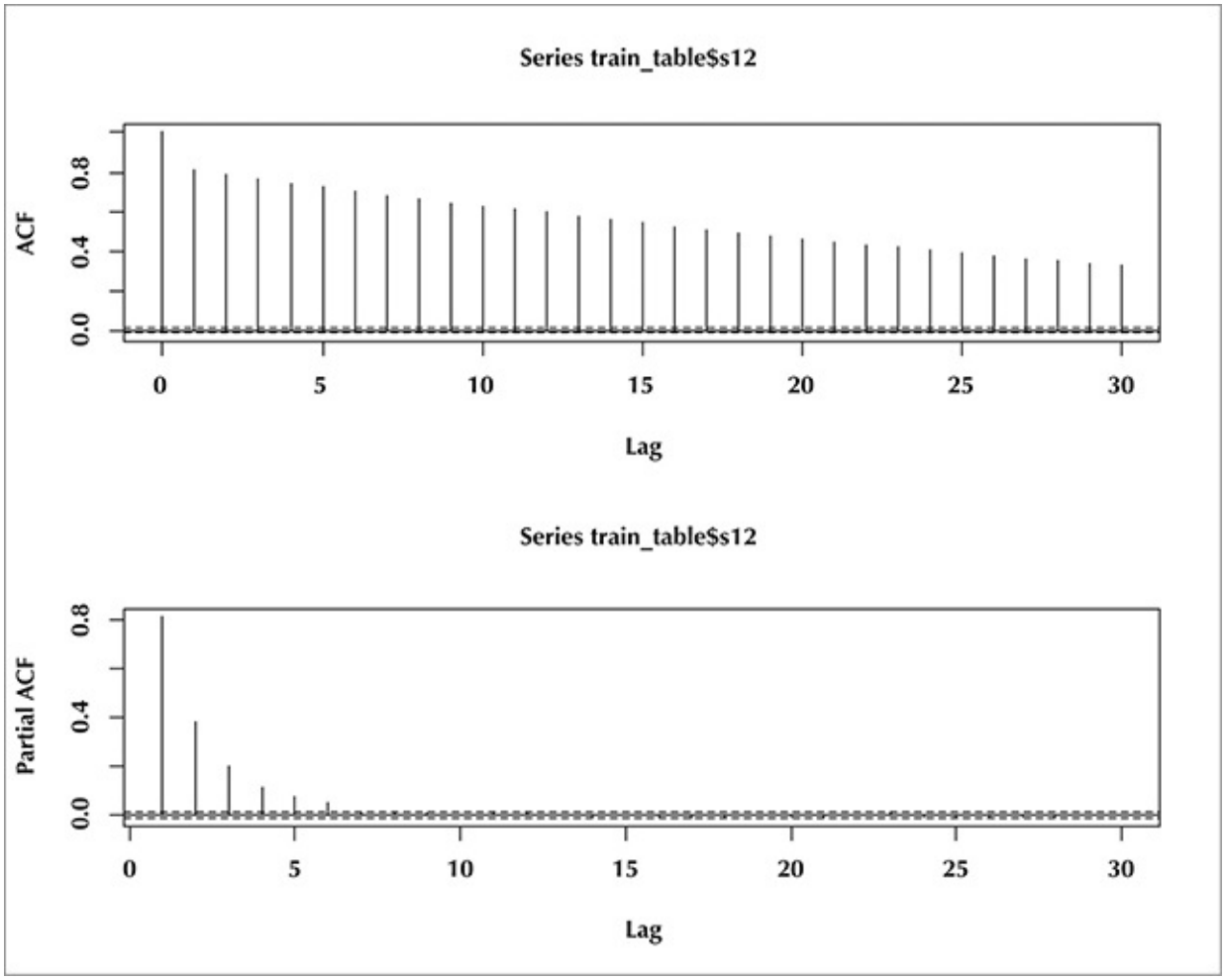


Rysunek 5.5. Autokorelacja pozwala ocenić, jak silnie bieżące wartości zmiennej (wartości dla klucza *t*) zależą od jej wartości dla klucza *t*-1, *t*-2 itd.

Siłę autokorelacji mierzy się za pomocą dwóch miar: miary autokorelacji ACF (ang. *Autocorrelation function*) i cząstkowej autokorelacji PACF (ang. *Partial autocorrelation function*). Pierwsza uwzględnia wszystkie obserwacje z badanego okresu, czyli mierzy siłę korelacji między wartością dla klucza *t* a wartością *t*-1, następnie wartościami między *t*-1 a *t*-2, następnie między wartościami *t*-1, *t*-2, *t*-3 itd. W cząstkowej autokorelacji są pomijane wartości dla pośrednich wartości klucza, czyli miara *PACF* pozwala ocenić siłę korelacji

między wartością dla klucza *t* a wartością *t*-1, następnie wartością *t*-2, następnie *t*-3 itd. Na wykresach na rysunku 5.6 przedstawiono wartości współczynników *ACF* i *PACF* dla zmiennej s12.

Skoro w szeregach czasowych występują korelacje między zmiennymi oraz autokorelacje, to można przypuszczać, że bieżące wartości zmiennej wyjściowej zależą nie tylko od bieżących, lecz także od poprzednich wartości zmiennych wejściowych. Na przykład awaria urządzenia jest skorelowana nie tylko z wynikami pomiarów jego pracy w momencie awarii, lecz także z pomiarami sprzed sekundy, minuty i godziny. Jak je jednak wykorzystać do uczenia modelu?



Rysunek 5.6. Zmienna s12 jest silnie skorelowana z sześcioma swoimi poprzednimi wartościami

Pierwsza myśl, jaka przychodzi nam do głowy, to dodanie zmiennych (kolumn) z poprzednimi wartościami zmiennych. Czyli zastąpienie wierszy kolumnami. Taka operacja przypomina rozdzielenie par klucz–wartość z użyciem poznanej w tym rozdziale funkcji `spread`.

```
str(data)
datw <- spread(data, key = cycle, value = s12)
str(datw)
```

```
'data.frame':    192 obs. of  2 variables:
 $ cycle: num  1 2 3 4 5 6 7 8 9 10 ...
 $ s12  : num  522 522 522 523 522 ...

'data.frame':    1 obs. of  192 variables:
 $ 1   : num 522
 $ 2   : num 522
 $ 3   : num 522
 $ 4   : num 523
…
```

Po zastanowieniu dochodzimy do wniosku, że wygenerowanie prawie 200 nowych zmiennych dla każdej oryginalnej zmiennej nie jest dobrym rozwiązaniem. Nawet gdybyśmy ograniczyli się do 10 ostatnich pomiarów, wciąż dziesięciokrotnie zwiększylibyśmy liczbę zmiennych i dosięgłaby nas klątwa wymiarowości. Musimy znaleźć inne rozwiązanie.

A gdyby policzyć średnią ruchomą i dodać ją do zmiennych? Mielibyśmy jedną dodatkową zmienną zawierającą informację o wygładzonych obserwacjach z *k* ostatnich okresów. Nie tylko rozwiązalibyśmy problem niekontrolowanego przyrostu liczby zmiennych, lecz także zwiększylibyśmy siłę predykcyjną nowej zmiennej przez ograniczenie jej szumu.

Prostą średnią ruchomą (ang. *simple moving average*, SMA) oblicza się jako średnią arytmetyczną z ostatnich *k* okresów

$$SMA = \frac{p_t + p_{t-1} + p_{t-2} + p_{t-k-1}}{k}$$

```
 SELECT [cycle] ,[s12] ,
 AVG([s12]) OVER (ORDER BY [cycle]
        ROWS BETWEEN 4 PRECEDING
             AND CURRENT ROW) AS SMA
FROM [PredictiveMaintenance].[PM_Train]
WHERE [id] = 1
ORDER BY [cycle];
```

cycle	s12	SMA
1	521.66	521.66
2	522.28	521.97
3	522.42	522.12
4	522.86	522.305
5	522.19	522.282
6	521.68	522.286
7	522.32	522.294
8	522.47	522.304

Zauważmy, że im dłuższy okres (większe *k*), tym lepsze wygładzenie. W tym przykładzie średnia ruchoma została obliczona dla $k = 5$.

Ważona średnia ruchoma (ang. *weighted moving average*, WMA) uwzględnia większy wpływ ostatnich wartości, przypisując różne wagi danym z poszczególnych okresów. Na przykład średnią ważoną z dwóch ostatnich wartości możemy policzyć następująco

$$WMA = \frac{p_t + p_{t-1} * W_{t-1}}{2}$$

```
SET @w = 0.7;
SELECT [cycle] ,[s12] AS Val,
 ISNULL((LAG([s12]) OVER (ORDER BY [cycle])), [s12]) AS PrevVal,
 @w * [s12] + (1 - @w) * ISNULL((LAG([s12]) OVER (ORDER BY [cycle])), [s12])  AS WMA
FROM [PredictiveMaintenance].[PM_Train]
WHERE [id] = 1
ORDER BY [cycle];
```

cycle	Val	PrevVal	WMA
1	521.66	521.66	521.66
2	522.28	521.66	522.094
3	522.42	522.28	522.378
4	522.86	522.42	522.728
5	522.19	522.86	522.391
6	521.68	522.19	521.833
7	522.32	521.68	522.128
8	522.47	522.32	522.425

Specjalnym przykładem średniej ruchomej jest wykładnicza średnia ruchoma (ang. *exponential moving average,* EMA). Nie wymaga ona określenia wag, bo przyjmuje, że znaczenie coraz bardziej odległych w czasie okresów maleje w sposób wykładniczy

$$EMA = \frac{p_t + (1-\alpha) * p_{t-1} + (1-\alpha)^2 * p_{t-2} + \ldots + (1-\alpha)^n * p_{t-n}}{1 + (1-\alpha) + (1-\alpha)^2 + \ldots + (1-\alpha)^n}$$

```
DECLARE @A AS FLOAT = 0.7, @B AS FLOAT;
SET @B = 1 - @A;
WITH cte_cnt AS
(
SELECT [cycle] ,[s12] AS Val,
  ROW_NUMBER() OVER (ORDER BY [cycle]) - 1 as exponent
FROM [PredictiveMaintenance].[PM_Train]
WHERE [id] = 1
)
SELECT [cycle], val,
 ROUND(
  SUM(CASE WHEN exponent=0 THEN 1
            ELSE @A
         END * val * POWER(@B, -exponent))
  OVER (ORDER BY [cycle]) * POWER(@B, exponent)
 , 2) AS EMA
FROM cte_cnt;
```

```
cycle   val      EMA
1       521.66   521.66
2       522.28   522.09
3       522.42   522.32
4       522.86   522.7
5       522.19   522.34
6       521.68   521.88
7       522.32   522.19
8       522.47   522.39
```

5.5. Redukcja danych

Zdarza się, że przykłady opisują dziesiątki, a nawet setki zmiennych, co oznacza, że po wzbogaceniu i przekształceniu dane treningowe mogą zawierać tysiące zmiennych. Musimy więc zmierzyć się z poznaną w poprzednim rozdziale klątwą wymiarowości. Naszym celem jest zmniejszenie wymiarowości danych tak, aby uprościć ich analizę, zmniejszyć wielkość reprezentatywnej próby (ograniczyć liczbę potrzebnych przykładów) i poprawić jakość modeli. Dlatego redukując liczbę wymiarów, powinniśmy starać się zachować jak

najwięcej wzorców ukrytych w danych.

5.5.1. Selekcja zmiennych

Najprostszą metodę redukcji danych już poznaliśmy. Polega ona na usuwaniu nieprzydatnych zmiennych wejściowych, czyli zmiennych nieskorelowanych w żaden sposób z wyjściową, oraz zmiennych silnie skorelowanych z innymi zmiennymi wejściowymi. **Ta metoda nazywa się selekcją zmiennych.**

Przypomnijmy – aby wybrać zmienne na podstawie ich przydatności w modelu, należy policzyć odpowiedni współczynnik korelacji między zmiennymi wejściowymi a wyjściową. Jeżeli zmienne mają rozkład zbliżony do normalnego, ich korelację zmierzymy współczynnikiem korelacji liniowej Pearsona. Korelację dwóch zmiennych kategorycznych ocenimy, przeprowadzając test zgodności *χ-kwadrat*. Do oceny korelacji między kategorycznymi zmiennymi porządkowymi użyjemy testu *rho-Spearmana*, a korelację między zmienną numeryczną a kategoryczną ocenimy, przeprowadzając analizę wariancji *ANOVA*.

Jako danych źródłowych użyjemy wstępnie przygotowanych danych o pasażerach Titanica.

```
SELECT *
FROM Titanic.vPreProcessedData;
```

Title	Embarked	Fare	Pclass	Sex	Age	FamilySize	IsAlone	Survived
Mr	1	37	2	male	31	3	0	0
Mrs	0	31	3	female	38	7	0	1
Mr	0	8	3	male	28	1	1	0
Mr	0	8	3	male	44	1	1	0
Mr	0	50	1	male	28	1	1	0

Proszę mieć na uwadze, że dane o pasażerach Titanica można przygotować inaczej, być może lepiej. Możne je też przygotować w innych językach niż SQL.

Na przykład w ostatnim punkcie rozdziału został przedstawiony sposób przygotowania tych samych danych w języku Python, a w poprzednim rozdziale ocena zmiennych i korelacji została przeprowadzona z użyciem funkcji języka R.

Przykłady są opisane dziewięcioma zmiennymi, zmienną wyjściową jest `Survived`. Policzmy odpowiednie współczynniki korelacji między zmiennymi wejściowymi w wyjściową. Dla poprawy czytelności przedstawiam tylko wyniki, kompletne testy znajdą Państwo w dołączonym do repozytorium książki pliku, o nazwie *Data preprocessing.sql*.

Zacznijmy od oceny korelacji zmiennych numerycznych z kategoryczną zmienną wyjściową. Uzyskane w wyniku analizy wariancji statystyki F są następujące:

- dla zmiennych `Embarked`, `Survived` F = 0,
- dla zmiennych `Fare`, `Survived` F = 63,
- dla zmiennych `Pclass`, `Survived` F = 299,
- dla zmiennych `Age`, `Survived` F = 4,
- dla zmiennych `FamilySize`, `Survived` F = 0.

Zależności między zmiennymi `Fare` i `Pclass` a zmienną wyjściową `Survived` są statystycznie istotne. Zależność między zmienną `Age` a `Survived` jest zaskakująco słaba, co może wynikać z uzupełnienia brakujących wartości zmiennej `Age` medianą. Należy znaleźć lepszy sposób uzupełnienia brakujących wartości tej zmiennej. A może lepiej byłoby podzielić wiek pasażerów na przedziały, na przykład dzieci, dorosłych i osoby starsze? Zmienne `FamilySize` i zakodowane stany zmiennej `Embarked` okazały się nieskorelowane ze zmienną wyjściową. Pierwszą, jako skorelowaną ze zmienną `IsAlone` należy usunąć. Przydatność drugiej zmiennej dla modelu należy sprawdzić w sposób opisany w dalszej

części rozdziału.

Do oceny korelacji między numerycznymi zmiennymi wejściowymi a zmienną wyjściową użyjemy testu Chi-kwadrat:

- dla zmiennych `Title`, `Survived` χ*-kwadrat* = 286,
- dla zmiennych `Sex`, `Survived` χ*-kwadrat* = 260,
- dla zmiennych `IsAlone`, `Survived` χ*-kwadrat* = 36.

Zależności między wszystkimi trzema wejściowymi zmiennymi kategorycznymi a zmienną wyjściową `Survived` są statystycznie istotne. Możemy przypuszczać, że obie zmienne wyliczeniowe (`Title` i `IsAlone`) przyczynią się do poprawy jakości modelu.

W tym przypadku przykłady są opisane tylko dziewięcioma zmiennymi, z czego dwie okazały się nieprzydatne. Zgodnie z naszą wiedzą na temat feralnego rejsu Titanica wszystkie sześć wybranych w ten sposób zmiennych wejściowych mogło mieć wpływ na szansę przeżycia katastrofy Titanica. Wyjątkowo pominiemy więc analizę korelacji między pozostałymi zmiennymi wejściowymi i nie usuniemy zmiennych, które mogłyby się okazać silnie ze sobą skorelowane.



Metodyki CRISP-DM i TDSP podkreślają znaczenie powtarzania w miarę potrzeb poszczególnych etapów projektu uczenia maszynowego. Na przykład po przygotowaniu danych należy ponownie ocenić zmienne. W poprzednim rozdziale wyjaśniłem, że statystyki są uogólnieniami i mogą wprowadzić nas w błąd. Jeżeli wyniki testów statystycznych budzą nasze wątpliwości, powinniśmy sprawdzić (np. za pomocą wykresów), czy spełnione są założenia przeprowadzonych testów, na przykład czy zmienne mają rozkład zbliżony do normalnego albo czy występują nieliniowe, a więc niezmierzone tymi testami korelacje między zmiennymi.

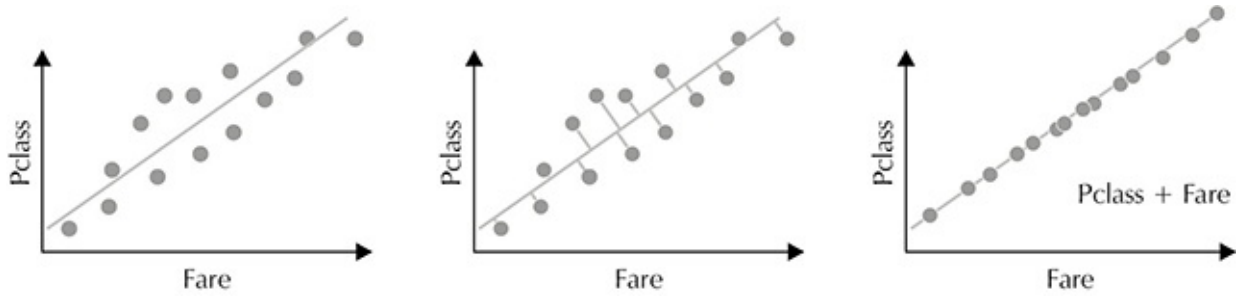
5.5.2. Analiza składowych głównych

Analiza składowych głównych PCA (ang. *principal component analysis*) jest najczęściej stosowaną metodą automatycznej (nienadzorowanej) redukcji liczby

wymiarów. Jej celem jest zastąpienie oryginalnych zmiennych mniejszą liczbą nowych zmiennych, tracąc jak najmniej informacji. Te nowe zmienne są nazywane składowymi głównymi, bo wylicza się je tak, żeby trafiło do nich jak najwięcej informacji znajdujących się w oryginalnych zmiennych.

Analiza składowych głównych wykorzystuje znane nam już pojęcia korelacji i wariancji, a tytułowe składowe główne to liniowe kombinacje oryginalnych zmiennych. PCA przypomina regresję liniową – oba modele są proste i oba analizują liniowe zależności między zmiennymi. W przeciwieństwie do regresji *PCA* jest metodą uczenia nienadzorowanego.

Na potrzeby tego przykładu przyjmijmy, że dwa wymiary chcemy zredukować do jednego. Weźmy dwie zmienne: `Pclass` i `Fare`. Jeżeli te dwie zmienne są ze sobą skorelowane, to zamiast na dwumiarowej płaszczyźnie można je przedstawić na jednowymiarowej linii – linii wyznaczonej przez model regresji liniowej tych dwóch zmiennych (rys. 5.7).



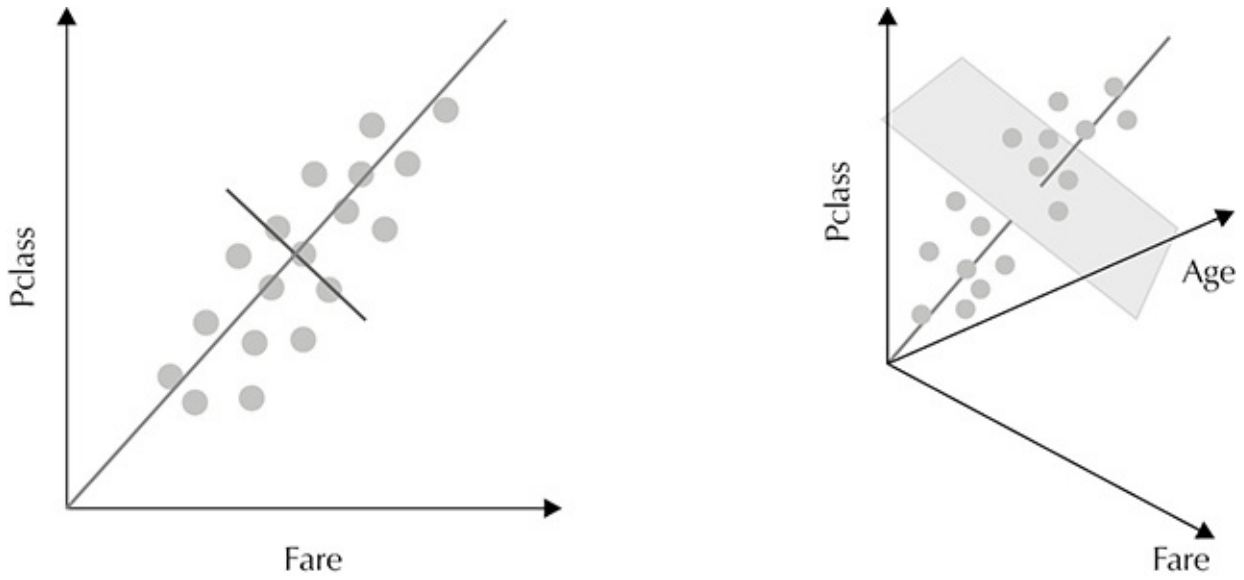
Rysunek 5.7. Zastępując dwie zmienne jedną – Pclass + Fare, straciliśmy tylko informacje o odległości oryginalnych punktów od linii regresji nowej zmiennej; tę linię nazywa się składową główną

Zauważmy, że gdyby ułożyć składową główną wzdłuż osi jednej z oryginalnych zmiennych, wyeliminowalibyśmy drugą zmienną, czyli przeprowadzili selekcję zmiennych. Ułożenie składowej głównej nie może więc być przypadkowe. W tym przypadku, gdyby składowa główna była prostopadła do pokazanej na rysunku 5.7, oryginalne punkty znalazłyby się na znacznie krótszym odcinku. Ich zmienność byłaby mniejsza, a utrata informacji większa.

Składowe główne wylicza się jako wektory własne macierzy kowariancji oryginalnych danych. Analiza *PCA* przebiega w czterech krokach:

- policzenie średniej i macierzy kowariancji zmiennych (macierzy zawierającej współczynnik wariancji wszystkich zmiennych);
- znalezienie wektorów własnych i ich własności własnych; wektor własny (ang. *eigenvector*) reprezentuje kierunek w macierzy o największej wariancji; przykładowy wektor własny pokazano na rysunku 5.7; własność własna (ang. *eigenvalue*) jest skalą wektora i pozwala mierzyć jego zmienność;
- wybór znalezionych wektorów własnych – wybiera się kolejne wektory o największych własnościach własnych;
- projekcja oryginalnych punktów do przestrzeni wyznaczonej przez wybrane wektory własne.

Składowe główne są ze sobą nieskorelowane, a każda kolejna składowa główna jest tak definiowana, żeby zawierała maksymalną ilość informacji, która nie trafiła do poprzedniej składowej głównej (rys. 5.8).



Rysunek 5.8. Dane z reguły są nierównomiernie rozłożone wzdłuż różnych, wyznaczonych zmiennymi,

współrzędnych. Analiza PCA polega na znalezieniu współrzędnej o największej zmienności, a następnie prostopadłej do niej współrzędnej o największej, pozostałej w zbiorze zmienności, itd.

Przeprowadźmy analizę składowych głównych dla danych o pasażerach Titanica. Ponieważ zmienne wejściowe muszą być numeryczne, zakodujemy stany zmiennych `Title` oraz `Sex`.

```
connection_string <- "Driver=SQL Server; Server=MS; Database=ML; Trusted_Connection=yes"
data_sql <- RxSqlServerData(sqlQuery = paste("SELECT CASE [Title]
       WHEN 'Mr' THEN 0
  WHEN 'Mrs' THEN 1
  WHEN 'Miss' THEN 2
  WHEN 'Master' THEN 3
  ELSE 4 END AS Title
 ,[Embarked]
 ,[Fare]
 ,[Pclass]
 ,CASE [Sex]
 WHEN 'male' THEN 0 ELSE 1 END AS Sex
 ,[Age]
 ,[IsAlone]
 FROM [ML].[Titanic].[vPreProcessedData];;"),
  connectionString = connection_string)
data <- rxImport(data_sql)
data.pca <- prcomp(data)
```

Sprawdźmy wyniki analizy.

```
summary(data.pca)
```

```
Importance of components:
                          PC1      PC2     PC3     PC4     PC5     PC6     PC7
Standard deviation     49.7068 12.95903 1.08492 0.67693 0.59943 0.45249 0.36494
Proportion of Variance  0.9355  0.06359 0.00045 0.00017 0.00014 0.00008 0.00005
Cumulative Proportion   0.9355  0.99912 0.99956 0.99974 0.99987 0.99995 1.00000
```

Pierwsza składowa główna zawiera 93% informacji (mierzonej wariancją) ukrytej w oryginalnych zmiennych. Odchylenie standardowej pierwszej składowej wynosi 49,7. Druga składowa główna zawiera 6% pozostałych (nieobecnych w pierwszej składowej) informacji, a jej odchylenie standardowe

to 12,9; pozostałe składowe główne zawierają tak mało informacji, że możemy je pominąć.

Sprawdźmy też, według jakich współrzędnych zostały wyliczone dwie pierwsze składowe główne.

```
data.pca$rotation[,c("PC1","PC2")]
```

```
              PC1           PC2
Title      0.002996162  -0.0209533472
Embarked   0.000882243  -0.0002686515
Fare       0.999573733  -0.0273866820
Pclass    -0.009291592  -0.0183456523
Sex        0.001743997  -0.0037489682
Age        0.027316142   0.9992008376
IsAlone   -0.002655346   0.0076107308
```

Pierwsza została wyliczona w 99% według współrzędnej `Fare`, a druga w 99% według współrzędnej `Age`. Jeżeli sprawdzimy wartości tych zmiennych, przekonamy się, że ich interpretacja jest co najmniej trudna. To akurat nie przeszkadza nam w użyciu tych dwóch zmiennych jako jedynych zmiennych wejściowych modelu uczenia maszynowego.

```
head(data.pca$x[,c("PC1","PC2")])
```

```
      PC1          PC2
[1,] -24.30534 -0.6629917
[2,] -19.30007 -0.7797074
[3,] -25.30103 -0.6604152
[4,] -25.30577 -0.6356289
[5,] -24.29760 -0.7087966
[6,] -24.30711 -0.6624970
```

Zanim ogłosimy sukces i zakończymy etap przygotowania danych, zastanówmy się, czemu akurat według tych współrzędnych zostały wyliczone dwie pierwsze składowe główne. Analiza PCA jest analizą nienadzorowaną, więc nie wskazaliśmy zmiennej wyjściowej; co więcej, zmiennej `Survived` nie ma nawet

w analizowanym zbiorze.

Skoro składowe główne są wyliczane jako wektory własne macierzy kowariancji, to może zmienne o dużych rozstępach zdominowały pozostałe? Przecież ceny biletów należą do przedziału od 9 do 512, najmłodszy pasażer miał 4 miesiące, a najstarszy 80 lat. Tymczasem pozostałe zmienne przyjmują tylko po kilka (a zmienne `Sex` i `IsAlone` tylko po dwie) wartości.

Powtórzmy analizę składowych głównych, tym razem skalując wszystkie zmienne.

```
data.pca <- prcomp(data, scale. = TRUE)
data.pca$scale
```

```
Title        Embarked     Fare        Pclass      Sex         Age         IsAlone
1.0768255   0.6401569   49.6869189   0.8360712   0.4779901 13.0196966   0.4896152
```

Rzeczywiście, skale zmiennych `Fare` i `Age` są wielokrotnie większe od skal pozostałych zmiennych. Sprawdźmy wyniki poprawionej analizy *PCA*.

```
summary(data.pca)
```

```
Importance of components:
                          PC1      PC2      PC3      PC4      PC5      PC6      PC7
Standard deviation       1.4486   1.2627   1.0210   0.9076   0.80656  0.66425  0.5910
Proportion of Variance   0.2998   0.2278   0.1489   0.1177   0.09293  0.06303  0.0499
Cumulative Proportion    0.2998   0.5275   0.6764   0.7941   0.88707  0.95010  1.0000
```

Otrzymane wyniki są zupełnie inne. Przyjmuje się, że procent wyjaśnionej przez wybrane składowe główne wariancji powinien przekraczać 75%. Do przekroczenia tego progu często wystarczy wybrać kilka (od 2 do 5) składowych głównych. Drugim kryterium wyboru, oprócz procentu wytłumaczonej zmienności oryginalnych danych, jest wartość własna poszczególnych składowych. Najczęściej pozostawia się składowe główne, które wyjaśniają więcej niż 5% wariancji. Po przeskalowaniu okazało się, że powinniśmy użyć pięciu pierwszych (a nie dwóch) składowych głównych jako zmiennych

wejściowych. Sprawdźmy jeszcze, według jakich współrzędnych zostały wyliczone te składowe główne.

```
data.pca$rotation[,c("PC1","PC2","PC3","PC4","PC5")]
```

	PC1	PC2	PC3	PC4	PC5
Title	0.48734799	-0.32571300	0.09571961	-0.33278525	0.24412347
Embarked	0.11797515	-0.09505124	0.86513881	0.43590862	-0.13254813
Fare	0.43195100	0.39773265	-0.07090929	0.41080780	0.30818510
Pclass	-0.35975626	-0.56183119	0.04415127	0.04766991	-0.23033124
Sex	0.49942400	-0.22913606	0.15014200	-0.45157596	-0.09845907
Age	-0.05758034	0.58407759	0.25740394	-0.44567138	-0.51505787
IsAlone	-0.42406101	0.13183623	0.38287821	-0.35448322	0.70698330

Teraz składowe główne faktycznie są kombinacjami zmiennych wejściowych, a wyniki analizy PCA nie wyglądają już jakbyśmy po prostu usuwali zmienne.

5.5.3. Wybór zmiennych przydatnych dla modelu

Trzecia metoda redukcji danych polega na ocenie faktycznej przydatności zmiennych w modelu. Ta metoda jest najbardziej kosztowna (obliczeniowo i czasowo), bo wymaga wielokrotnego trenowania modelu przy użyciu różnych kombinacji zmiennych wejściowych. Metoda ta występuje w dwóch wariantach:

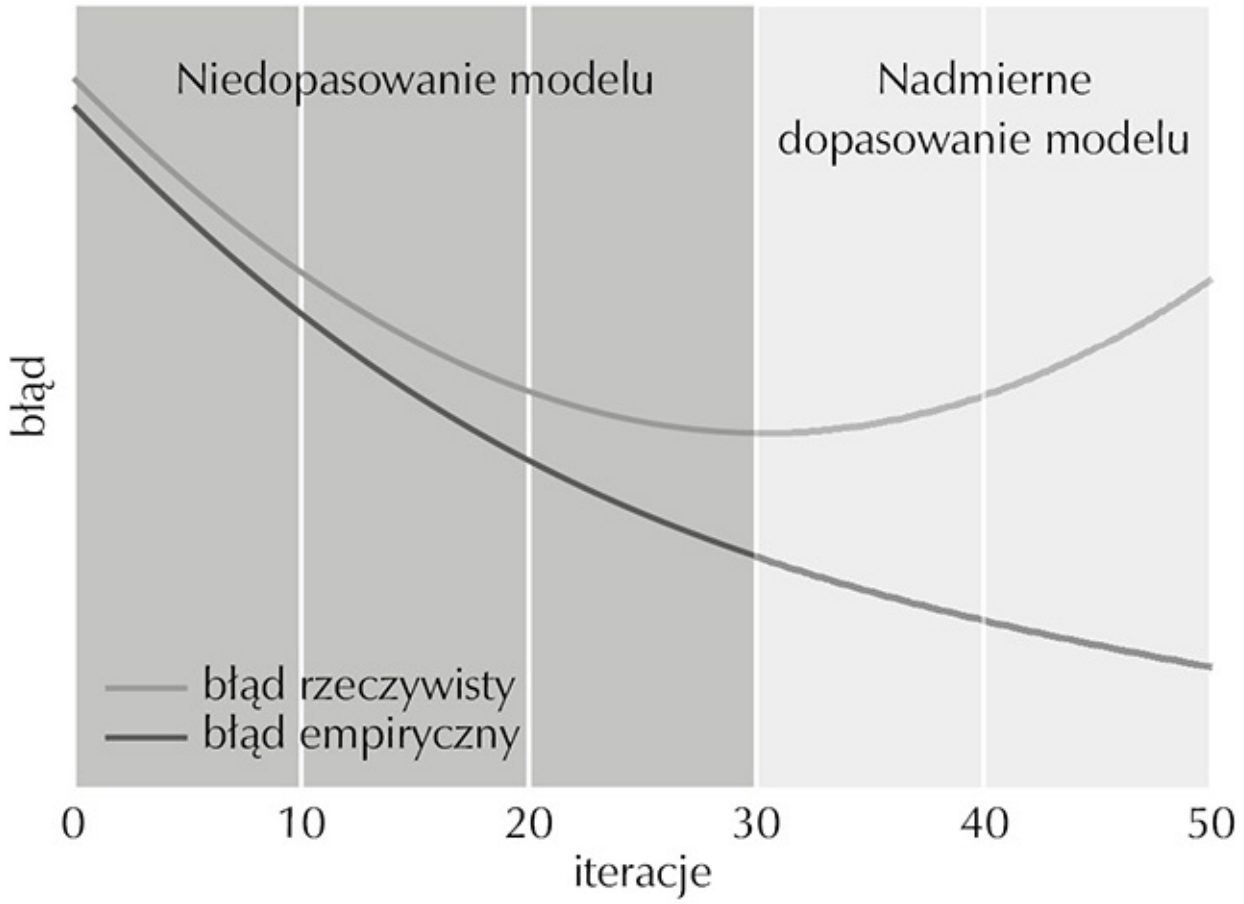
- Możemy zacząć od jednej, najsilniej skorelowanej z wyjściową, zmiennej wejściowej, i do kolejnych modeli kandydujących dodać po jednej z pozostałych zmiennych wejściowych. Jakość wszystkich modeli nauczonych na parach zmiennych wejściowych porównuje się i wybiera model najlepszy według wybranego kryterium oceny. Następnie dodaje się trzecią zmienną, znowu po kolei trenując modele kandydujące na kombinacjach pozostałych zmiennych. Cały proces powtarza się aż do sprawdzenia wszystkich zmiennych, co wymaga wytrenowania 2^n modeli, gdzie n jest liczbą zmiennych wejściowych. Na końcu należy wybrać najlepszy model. Wadą metody wstępującej jest to, że ocenia przydatność

zmiennej w modelu nauczonym na podstawie wcześniej wybranych zmiennych, a nie w modelach nauczonych na wszystkich możliwych kombinacjach pozostałych zmiennych. Tymczasem zmienna nieprzydatna dla jednego modelu może być wartościowa dla innego.

- Możemy też zacząć od wszystkich zmiennych, a następnie z nauczonego modelu usunąć jedną zmienną i ponownie go wytrenować. Zamiast usuwać zmienne, możemy też zastąpić je zmiennymi losowymi, co pozwala lepiej ocenić pośredni i bezpośredni wpływ usuniętej zmiennej na model. Po usunięciu wszystkich zmiennych z modeli kandydujących ocenia się ich jakość. Jeżeli jakość modelu ze zmienną i bez niej są zbliżone, zmienna była mało przydatna dla modelu. Najmniej przydatną zmienną się usuwa i cała procedura zostaje powtórzona. Wadą metody zstępującej jest to, że dla wielu (kilkudziesięciu lub więcej) staje się ona niepraktyczna, bo wymaga wytrenowania *n*! modeli, gdzie *n* jest liczbą sprawdzanych zmiennych.

5.6. Podział danych

Z rozdziału 3 wiemy, że dane treningowe są używane do wykrycia przez algorytm uczenia maszynowego ukrytych w nich zależności. W modelach partycjonujących wykryte wzorce są reprezentowane przez drzewa decyzyjne, w modelach regresji przez znalezione w czasie treningu wartości parametrów. Niestety, w danych oprócz prawidłowych wzorców zawsze znajduje się szum, czyli przypadkowe kombinacje wartości zmiennych. Dlatego do oceny i poprawy jakości kolejnych modeli kandydujących używa się danych testowych. Ocena modeli na podstawie danych testowych, a nie treningowych, pozwala wykryć nadmierne dopasowanie modelu (rys. 5.9).



Rysunek 5.9. Zwiększając pojemność modelu, zwiększając liczbę zmiennych wejściowych albo wydłużając czas treningu, zmniejszymy błąd treningowy (empiryczny). Jednak w którymś momencie model zacznie nadmiernie dopasowywać się do danych treningowych i chociaż błąd empiryczny nadal będzie malał, to błąd rzeczywisty (testowy) zacznie rosnąć

Jeżeli tylko dysponujemy wystarczającą liczbą przykładów, a dane treningowe są podobne do testowych, takie rozwiązanie pozwala rozwiązać najtrudniejszy problem uczenia maszynowego – polegający na tym, że model nauczony na zbiorze treningowym powinien prawidłowo działać na innych, nieużytych do treningu danych. Najprostszym sposobem zagwarantowania podobieństwa obu zbiorów jest losowy podział przykładów na treningowe i testowe.

5.6.1. Podział warstwowy

Jeżeli chcemy zachować proporcje jakieś zmiennej, dane należy podzielić

metodą podziału warstwowego. Przypuśćmy, że 30% naszych klientów ma wykształcenie podstawowe, 50% średnie, a 20% wyższe. Budujemy model rekomendujący. Wiedząc, że osoby o różnym wykształceniu interesują się innymi produktami, chcemy zachować proporcje zmiennej `Education` w zbiorze treningowym i testowym. W ten sposób ocena modelu prawidłowo uwzględni wszystkie trzy grupy klientów.

Losowanie metodą doboru warstwowego polega na podzieleniu danych na jednorodne grupy (*starty*) względem wybranej zmiennej, w tym przypadku zmiennej `Education`. Następnie z każdej grupy losuje się określoną liczbę przykładów, aby w zbiorze treningowym i testowym rozkład częstości wybranej zmiennej był taki sam. Sposób dzielenia danych metodą doboru warstwowego ilustruje poniższy przykład. Dzielimy w nim dane o pasażerach Titanica względem ich płci[10].

```
connection_string <- "Driver=SQL Server; Server=MS; Database=ML; Trusted_Connection=yes"
table_name <- "[Titanic].[Train]"
table <- RxSqlServerData(table = train_table_name, connectionString = connection_string)
data <- rxImport(train_data_table)
prop.table(table(data$Sex))

library(dplyr)
set.seed(13)
test <- data %>%
  group_by(Sex) %>%
  sample_frac(.10)
prop.table(table(train$Sex))
```

```
    female      male
0.352413     0.647587
    female      male
0.3525641    0.6474359
```

Chociaż zbiór `data` liczy tylko 891 wierszy, to po losowym wybraniu z niego 10% (89 wierszy) rozkład zmiennej `Sex` się nie zmienił. W praktyce do podziału danych metodą doboru warstwowego używa się specjalistycznych bibliotek,

takich jak biblioteka `splitstackshape`.

```
library(splitstackshape)
set.seed(123)
strata <- stratified(indt = data, group = "Sex", size =.70, bothSets=TRUE)
prop.table(table(strata$SAMP1$Sex))
prop.table(table(strata$SAMP2$Sex))
```

```
female       male
0.3525641 0.6474359
female       male
0.3520599 0.6479401
```

Podział warstwowy pozwala zachować proporcje kategorycznej zmiennej wyjściowej. Dane dzieli się względem zmiennej wyjściowej, jeżeli liczba przykładów jest mała, a więc istnieje ryzyko, że któryś ze stanów zmiennej wyjściowej będzie nieprawidłowo reprezentowany w zbiorze testowym. Ryzyko to jest tym większe, im więcej stanów przyjmuje zmienna. Czasami jednak chcemy zmienić rozkład binarnej zmiennej wyjściowej, czyli zrównoważyć dane.

5.6.2. Równoważenie danych

Największą szansę zbudowania dobrego modelu uczenia maszynowego mamy wtedy, gdy dane treningowe są reprezentatywną próbą populacji o nieznanym rozkładzie, a rozkład danych testowych jest zbliżony do rozkładu danych treningowych. Zdarza się jednak specjalnie zmieniać rozkład zmiennej wyjściowej w danych treningowych. Jeżeli jeden stan zmiennej wyjściowej występuje znacznie rzadziej niż drugi, powinniśmy równoważyć dane treningowe.

Typowym przykładem takiej sytuacji jest modelowanie wykrywania oszustw. Ponieważ zdecydowana większość transakcji nie jest próbą oszustwa, stan reprezentujący prawidłową transakcję będzie występował w ponad 99%

przypadków. Chociaż zmienna `fraud` o takim niezrównoważonym rozkładzie jest reprezentatywna, to dla algorytmów uczenia maszynowego może być problematyczna.

Jej problematyczność wynika z zastosowanej metody uczenia i oceniania modeli. Skoro uczenie maszynowe polega na minimalizowaniu funkcji kosztu, koszt prawidłowej klasyfikacji wynosi 0, a koszt błędnej klasyfikacji jest większy od zera, to model, który nauczyłby się tylko jednej, prostej reguły głoszącej, że wszystkie transakcje są legalne, będzie miał rację w ponad 99% przypadków. Taki model byłby jednocześnie bardzo dokładny i całkowicie bezużyteczny. Podobna sytuacja ma miejsce w modelowaniu rzadkich chorób. Tu również przykłady (historyczne dane pacjentów) będą zawierały bardzo dużo przypadków niewykrycia poszukiwanej choroby i tylko kilka czy kilkadziesiąt przypadków jej zdiagnozowania. Model, który wszystkich nowych pacjentów diagnozowałby jako zdrowych, byłby bardzo dokładny, ale śmiertelnie groźny. Problem ten można rozwiązać, równoważąc dane albo zmieniając kryteria oceny modelu. Trafność, jak widzimy, nie jest dobrym kryterium oceny modeli nauczonych na niezrównoważonych danych. W takim przypadku należy optymalizować model pod kątem innych, przedstawionych w dalszej części książki miar, takich jak współczynnik *Kappa* czy *F-miara*.

Kiedy należy zrównoważyć dane treningowe? To zależy od danych, zastosowanego algorytmu uczenia maszynowego i wybranej funkcji kosztu. Jeżeli rzadko występujący stan zmiennej wyjściowej jest łatwo (liniowo) separowany, to model nauczy się go rozpoznawać nawet na 10% przypadków. Jeśli jednak w danych treningowych występują nakładające się na siebie, ale należące do różnych klas, na przykład identyczne transakcje, z których niektóre były próbą oszustwa, a inne nie, nawet 30% przypadków rzadziej występującego stanu może się okazać za mało.

Dane możemy zrównoważyć na kilka sposobów. Jeżeli dysponujemy

wystarczającą liczbą przykładów reprezentujących klasy mniejszościowe, to możemy usunąć część przykładów większościowych. W przeciwnym razie powinniśmy albo zebrać więcej danych, albo zastosować metodę nadpróbkowania.

Usunięcie części przykładów większościowych (ang. *undersampling*) **nie tylko pozwala zbilansować dane,** lecz także **jest sposobem na ich redukcję**. Jeśli dysponujemy miliardem automatycznie zebranych przykładów prawidłowego działania urządzenia, to ich zredukowanie do 100 mln znacznie skróci czas uczenia modelu i, co ważniejsze, poprawi jego jakość. Można udowodnić, że jeżeli model nauczony na zbilansowanych w ten sposób danych działa prawidłowo, to również poprawnie będzie on działał na oryginalnych danych, zwłaszcza, że błąd klasyfikatora binarnego nauczonego na danych, z których usnęliśmy k procent przykładów większościowych, będzie *k*-razy większy dla oryginalnych danych.

Do zrównoważenia danych można użyć specjalistycznych bibliotek, takich jak biblioteka `ROSE`. Przypuśćmy, że chcemy zrównoważyć niezbilansowany stan zmiennej `Survived` i zdecydowaliśmy się to zrobić, usuwając losowo wybrane przykłady większościowe.

```
library(ROSE)
nrow(data)
prop.table(table(data$Survived))
data_balanced_under <- ovun.sample(Survived~.,data, method = "under", seed = 1)$data
prop.table(table(data_balanced_under$Survived))
nrow(data_balanced_under)
prop.table(table(data_balanced_under$Survived))
```

```
891
0         1
0.6161616 0.3838384
673
0         1
0.4918276 0.5081724
```

Usuwając 218 przykładów, zrównoważyliśmy dane treningowe. A co, jeśli danych nie mamy na tyle dużo, żeby pozwolić sobie na zmniejszenie liczby przykładów treningowych? Albo jeżeli jakość modelu nauczonego na mniejszej liczbie przykładów spadła poniżej akceptowalnej? Wtedy powinniśmy zrównoważyć je przez nadpróbkowanie.

Nadpróbkowanie (ang. *oversampling*) **polega na dodaniu do zbioru treningowego dodatkowych przykładów mniejszościowych**. Te dodatkowe, sztucznie wygenerowane przykłady nie powinny być identycznymi kopiami przykładów klasy mniejszościowej, ale powinny być do nich podobne. Do generowania nowych przykładów najczęściej używa się algorytmu SMOTHE (ang. *synthetic minority oversampling technique*). Generuje on nowe przypadki na podstawie cech przykładów klasy mniejszościowej, czyli otrzymany zbiór będzie zawierał dodatkowe wiersze, których zmienne wejściowe będą pewnym uogólnieniem zmiennych wejściowych oryginalnych przykładów tej klasy. Stopień tego uogólnienia zależy od liczby przykładów, na postawie których są generowane nowe. Wartość 1 oznacza, że zawartość wygenerowanych wierszy będzie kombinacją cech pojedynczych przypadków mniejszościowych, wartość 2 – że do wygenerowania jednego przypadku zostaną użyte kombinacje cech dwóch najbardziej podobnych do siebie przypadków mniejszościowych itd. Funkcja `ovun.sample` automatycznie ustala wartość tego parametru na podstawie macierzy kowariancji zmiennych.

```
data_balanced_over <- ovun.sample(Survived~.,data, method = "over", p=0.5, seed = 1)$data
nrow(data_balanced_over)
prop.table(table(data_balanced_over$Survived))
```

```
1078
0         1
0.5092764 0.4907236
```

Tym razem do danych treningowych zostało dodanych 187 wierszy. Jeżeli obawiamy się, że duży procent sztucznie wygenerowanych przykładów

pogorszy jakość modelu albo wydłuży czas trenowania, możemy połączyć obie metody.

```
data_balanced_both <- ovun.sample(Survived~.,data, method = "both", p=0.5, seed = 1)$data
nrow(data_balanced_both)
prop.table(table(data_balanced_both$Survived))
891
0         1
0.5140292 0.4859708
```

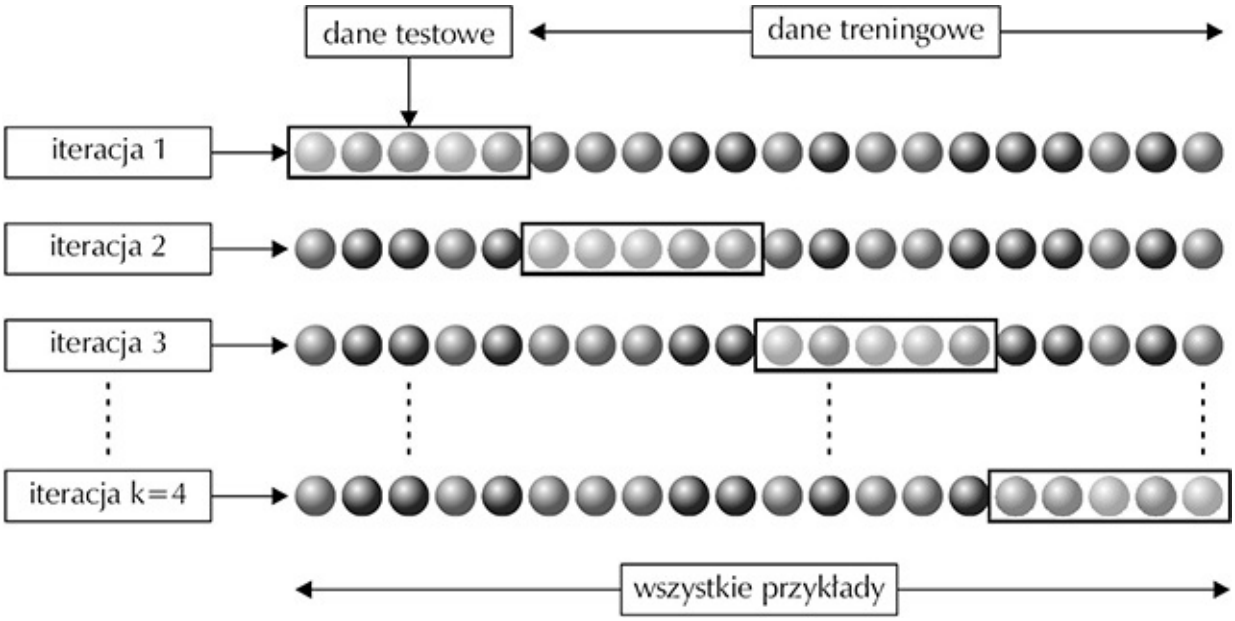
5.6.3. *k*-krotna walidacja krzyżowa

Gdybyśmy model oceniali na podstawie błędu empirycznego, a więc przy użyciu danych treningowych, najlepszym modelem zawsze okazywałby się model o dużej zmienności. Model o małym, czy nawet zerowym błędzie treningowym i dużej zmienności będzie jednak obarczony dużym (czasami bardzo dużym) błędem rzeczywistym. Wydzielenie danych testowych pozwala przeprowadzić wiarygodną ocenę modeli uczenia maszynowego. Co jednak, jeżeli dysponujemy małą liczbą przykładów? A w dodatku przeprowadzenie dodatkowych obserwacji jest trudne (np. kosztowne) albo niemożliwe (tak jak w przypadku danych o pasażerach Titanica)? Jeżeli mamy za mało przykładów, żeby wydzielić z nich wystarczająco duży (reprezentatywny) zbiór testowy, możemy użyć wszystkich danych do treningu, a mimo to prawidłowo ocenić model. Zawdzięczamy to metodzie k-krotnej walidacji krzyżowej (ang. *k-fold cross-validation*).

Walidacja krzyżowa przebiega następująco (rys. 5.10):

- zbiór danych źródłowych jest losowo dzielony na *k* możliwie równolicznych, rozłącznych partycji (*Fold*) S_1, S_2, …, S_k (często $k = 10$);
- tworzonych jest *k* kopii ocenianego modelu eksploracji danych;
- każda z tych kopii jest trenowana na podstawie $k - 1$ partycji, a do oceny jej

dokładności jest używana pozostała partycja – na przykład pierwsza kopia modelu zostanie nauczona na partycjach S_1, …, S_9 i oceniona na podstawie partycji S_{10}, druga kopia zostanie nauczona na partycjach S_2, …, S_{10} i oceniona na podstawie partycji S_1 itd; w ten sposób każda partycja zostanie użyta do treningu $k - 1$ kopii modelu i raz do oceny ich dokładności.



Rysunek 5.10. Czterokrotna walidacja krzyżowa polega na trenowaniu i ocenie czterech kopii modelu, przy czym każda kopia jest oceniana na innych, nieużytych do treningu tej kopii, danych

Taki sposób działania oznacza, że metodą *k*-krotnej walidacji krzyżowej można także ocenić zmienność modelu, czyli to, jak bardzo predykcje zwracane przez model zależą od różnych danych. Model jest tym stabilniejszy, im różnice w jego działaniu są mniejsze.

Zauważmy, że podział danych na treningowe i testowe jest specjalnym przypadkiem *k*-krotnej walidacji krzyżowej, przypadkiem, w którym $k = 1$. Różnica między obiema metodami polega na tym, że skoro model oceniamy tylko raz, zbiór danych testowych musi być odpowiednio liczny. Jeżeli ocenę powtórzymy dziesięć razy, każdy z dziesięciu zbiorów testowych może być kilkukrotnie mniejszy.

Przeciwstawnym ekstremum jest wybór wartości *k* jako równej liczbie przykładów *n*. Oba ekstrema mają własne nazwy: jeżeli *k* jest równe 1, mówimy o podziale danych na treningowe i testowe; jeśli *k* jest równe liczbie przykładów *n*, mamy do czynienia z *n*-krotną walidacją krzyżową (ang. *leave one out cross validation*). **Im większa wartość *k*, tym więcej kopii modelu zostanie wytrenowanych na większych zbiorach danych treningowych.** W naszym przykładzie, gdybyśmy zastosowali *n*-krotną walidacją krzyżową, 891 modeli zostałoby wytrenowanych na zbiorach treningowych liczących 890 przykładów i ocenionych na jednym, za każdym razem innym przykładzie testowym.

Wyniki ocen poszczególnych kopii modelu są uśredniane. Tak otrzymany wynik jest wiarygodnym przybliżeniem oceny modelu nauczonego na wszystkich danych – im większa wartość *k*, tym lepsze przybliżenie. Jedną z wielu bibliotek zawierających funkcje *k*-krotnej walidacji krzyżowej jest biblioteka `cvTools`. Poniższy przykład pokazuje jak w języku R:

- połączyć się z serwerem SQL Server;
- wynik zapytania zaimportować do obiektu typu `data.frame`;
- sformułować hipotezę (formułę uczącą);
- dane podzielić na 5 partycji;
- pięciokrotnie nauczyć model drzew decyzyjnych, za każdym razem trenując i testując go na danych z innych partycji;
- uśrednić wyniki oceny każdego modelu[11].

```
library(cvTools)
set.seed(12)
connection_string <- "Driver=SQL Server; Server=MS; Database=ML; Trusted_Connection=yes"
data_sql <- RxSqlServerData( sqlQuery = paste("SELECT CASE [Title]
          WHEN 'Mr' THEN 0
    WHEN 'Mrs' THEN 1
    WHEN 'Miss' THEN 2
    WHEN 'Master' THEN 3
    ELSE 4 END AS Title
  ,[Embarked]
  ,[Fare]
  ,[Pclass]
  ,CASE [Sex] WHEN 'male' THEN 0 ELSE 1 END AS Sex
  ,[Age]
  ,[IsAlone]
  ,[Survived]
 FROM [ML].[Titanic].[vPreProcessedData];"),connectionString = connection_string)
data <- rxImport(data_sql)
formula <- as.formula("Survived ~ Title+Embarked+Fare+Pclass+Sex+Age+IsAlone")
accuracy <- as.numeric()
kf<-cvFolds(nrow(data), K = 4, type = "random")
for (i in 1:5) {
  model <- rxFastTrees(formula, data = data[kf$which!=i,], type = "binary")
  scoreDS <- rxPredict(model, data = data[kf$which==i,], extraVarsToWrite = c("Survived"))
  model_metrics <- evaluate_classifier(actual = "Survived", predicted = "PredictedLabel",
                                       data = scoreDS)
  accuracy <- rbind(accuracy, model_metrics[1])
}
sprintf ('Model accuracy: %.2f and stdev %.3f' , 100*mean(accuracy), sd(accuracy))
```

```
"Model accuracy: 80.47 and stdev  0.023"
```

Po zwiększeniu liczby partycji do 20 trafność modelu wzrosła do 81,55%, a kolejne zwiększenie liczby partycji do 50 (przy 50 partycjach do kolejnych testów było używanych 18 przykładów) spowodowało poprawę średniej trafności do wartości 81,65%, przy odchyleniu standardowym równym 0,1. Z dużą pewnością możemy uznać, że trafność tego modelu, nauczonego na wszystkich danych, wyniesie 81,5%.

5.7.
Danych walidacyjnych używa się do optymalizacji, a nie do oceny modeli

Zanim zdecydujemy się na wdrożenie najlepszego modelu do użycia, powinniśmy poddać go końcowemu egzaminowi. Niestety, jeżeli do końcowej oceny użyjemy tych samych danych testowych, które wcześniej służyły nam do optymalizacji, otrzymana ocena będzie niewiarygodna. Niezależnie od tego, czy optymalizacja była przeprowadzana automatycznie (np. metodą wyboru przydatnych zmiennych), czy samodzielnie (np. metodą selekcji zmiennych), zawsze istnieje ryzyko nadmiernego dopasowania modelu do danych testowych. W przypadku metod automatycznych ryzyko to wprost wynika ze sposobu działania algorytmów optymalizacyjnych. Z definicji będą one wybierały te modele (czy to przez dobór zmiennych, czy hiperparametrów), których błąd jest mniejszy. Znowu znaleźliśmy się w sytuacji jak przedstawiona na rysunku 5.9. Im dłużej będziemy optymalizować model, tym dokładniej dopasuje się on do danych testowych.

Dokładnie to samo dzieje się podczas samodzielnej optymalizacji modelu. Tutaj ryzyko nadmiernego dopasowania do danych testowych jest nawet większe, bo nasze, ludzkie zdolności do rozpoznawania danych ukrytych we wzorcach przewyższają umiejętności maszyn. Czyli jeszcze szybciej, świadomie lub nieświadomie, dopasujemy model do danych testowych.

Oceniając model przy użyciu tych samych danych, których użyliśmy do jego optymalizacji, ryzykujemy wystawienie mu oceny wyższej, niż wystawią mu jego użytkownicy. Jedynym sposobem wiarygodnej oceny modelu jest użycie danych nieużytych wcześniej ani do treningu, ani do optymalizacji. Dlatego dane treningowe dzieli się na dwa podzbiory: treningowy i walidacyjny.

5.8.
Kto miał szansę przeżyć katastrofę Titanica?

Pora podsumować materiał z pierwszych pięciu rozdziałów książki i zbudować model predykcyjny z użyciem SQL Server Machine Learning Services. Dla odmiany ten model zbudujemy w Pythonie, korzystając z bibliotek `pandas`, `numpy` i `sklearn`. Kompletne rozwiązanie znajdą Państwo w dołączonych do książki plikach *Who had the chance to survive the disaster.py* i *Who had the chance to survive the disaster.sql*.

Zacznijmy od zbudowania prototypu modelu w języku Python. Uprości nam to pisanie i testowanie poszczególnych bloków kodu, bo pozwoli skorzystać z takich funkcjonalności edytora PyCharm, jak kontrola składnia czy pomoc kontekstowa. W tym przypadku liczba przykładów jest tak mała, że wszystkich użyjemy do zbudowania prototypu. Zaczniemy od wczytania danych treningowych i testowych do obiektów typu *pandas.DataFrame*.

```
train_df = pandas.read_csv('train.csv')
test_df = pandas.read_csv('test.csv')
```

Oba pliki mają identyczną strukturę, z jednym ważnym wyjątkiem – w danych testowych nie znajdziemy kolumny `Survived`. Jedyny sposób ich użycia polega na uzupełnieniu brakującej informacji o tym, czy dana osoba przeżyła katastrofę, a następnie wysłaniu pliku tekstowego z wynikami w postaci `PassengerId`, `Survived` do serwisu Kaggle.

Znając postawiony problem i dane, możemy zacząć ich przygotowywanie. W pierwszej kolejności usuniemy nieprzydatne zmienne.

```
train_df = train_df.drop([,Ticket', ,Cabin', ,PassengerId'], axis=1)
test_df = test_df.drop(['Ticket', 'Cabin'], axis=1)
```

Zanim zaczniemy przygotowywać pozostałe zmienne, zastanówmy się, czy

powinniśmy wzbogacać, uzupełniać, czyścić i przekształcać tylko dane treningowe, czy treningowe i testowe? Skoro nauczymy model na przekształconych danych treningowych, to dane testowe również muszą zostać tak samo przekształcone, żeby na ich podstawie można było sensownie ocenić model. Weźmy pierwszy z rzędu przykład – skoro w danych treningowych znajduje się zmienna `Title`, to ta zmienna (a nie zmienna `Name`, z której tytuł pasażera został wyodrębniony) powinna się znajdować w danych testowych. To samo dotyczy innych zmiennych. I tak, skoro podzieliliśmy zmienną `Fare` na przedziały i teraz jej wartość 1 odpowiada wartościom z przedziału 7,5 i 15 oryginalnej zmiennej, to używanie do testów wartości 14, a nie 1, nie ma sensu. Skoro będziemy przygotowywać oba zbiory, to warto je połączyć.

```
combine = [train_df, test_df]
```

W pierwszej kolejności wyliczymy zmienną `Title`, zakodujemy jej stany i usuniemy niepotrzebną już zmienną `Name`.

```
for dataset in combine:
    dataset['Title'] = dataset.Name.str.extract(' ([A-Za-z]+)\.', expand=False)

for dataset in combine:
    dataset['Title'] = dataset['Title'].replace(['Lady', 'Countess','Capt', 'Col','Don',
'Dr', 'Major', 'Rev', 'Sir', 'Jonkheer', 'Dona'], 'Other')

    dataset['Title'] = dataset['Title'].replace('Mlle', 'Miss')
    dataset['Title'] = dataset['Title'].replace('Ms', 'Miss')
    dataset['Title'] = dataset['Title'].replace('Mme', 'Mrs')

title_mapping = {"Mr": 1, "Miss": 2, "Mrs": 3, "Master": 4, "Rare": 5}
for dataset in combine:
    dataset['Title'] = dataset['Title'].map(title_mapping)
    dataset['Title'] = dataset['Title'].fillna(0).astype(int)

train_df = train_df.drop(['Name'], axis=1)
test_df = test_df.drop(['Name'], axis=1)
combine = [train_df, test_df]
```

Następnie do danych dodamy zmienną `IsAlone` i usuniemy zmienne, na podstawie których została ona wyliczona.

```
for dataset in combine:
    dataset['FamilySize'] = dataset['SibSp'] + dataset['Parch'] + 1

for dataset in combine:
    dataset['IsAlone'] = 0
    dataset.loc[dataset['FamilySize'] == 1, 'IsAlone'] = 1

train_df = train_df.drop(['Parch', 'SibSp', 'FamilySize'], axis=1)
test_df = test_df.drop(['Parch', 'SibSp', 'FamilySize'], axis=1)
combine = [train_df, test_df]
```

Zakodowanie stanów zmiennej `Sex` sprowadza się do wykonania dla obu zbiorów jednej instrukcji.

```
for dataset in combine:
    dataset['Sex'] = dataset['Sex'].map( {'female': 1, 'male': 0} ).astype(int)
```

Brakujące porty zaokrętowania uzupełnimy najczęściej występującą wartością, czyli nazwą portu, w którym na pokład weszło najwięcej pasażerów. Następnie zakodujemy stany zmiennej `Embarked`.

```
freq_port = train_df.Embarked.dropna().mode()[0]
for dataset in combine:
    dataset['Embarked'] = dataset['Embarked'].fillna(freq_port)

for dataset in combine:
    dataset['Embarked'] = dataset['Embarked'].map( {'S': 0, 'C': 1, 'Q': 2} ).astype(int)
```

Uzupełnienie brakujących wartości zmiennej `Age` jest bardziej skomplikowane. Brakujących wartości jest na tyle dużo (prawie 20%), że, jak się przekonaliśmy, zastąpienie ich stałą (np. medianą) zmniejsza przydatność zmiennej. Brakujący wiek możemy uzupełnić medianą wyliczoną osobno dla pasażerów i pasażerek podróżujących różnymi klasami.

```
guess_ages = np.zeros((2,3))
for dataset in combine:
    for i in range(0, 2):
        for j in range(0, 3):
            guess_df = dataset[(dataset['Sex'] == i) & \
                                (dataset['Pclass'] == j + 1)]['Age'].dropna()

            age_guess = guess_df.median()

            # Convert random age float to nearest .5 age
            guess_ages[i, j] = int(age_guess / 0.5 + 0.5) * 0.5

    for i in range(0, 2):
        for j in range(0, 3):
            dataset.loc[(dataset.Age.isnull()) & (dataset.Sex == i) & (dataset.Pclass
== j + 1), \
                        'Age'] = guess_ages[i, j]

    dataset['Age'] = dataset['Age'].astype(int)
```

Ostatnią zmienną wymagającą przygotowania jest `Fare`. Po uzupełnieniu wartości brakujących w zbiorze testowym medianę podzielimy na cztery przedziały.

```
test_df['Fare'].fillna(test_df['Fare'].dropna().median(), inplace=True)
for dataset in combine:
    dataset.loc[ dataset['Fare'] <= 7.91, 'Fare'] = 0
    dataset.loc[(dataset['Fare'] > 7.91) & (dataset['Fare'] <= 14.454), 'Fare'] = 1
    dataset.loc[(dataset['Fare'] > 14.454) & (dataset['Fare'] <= 31), 'Fare']   = 2
    dataset.loc[ dataset['Fare'] > 31, 'Fare'] = 3
    dataset['Fare'] = dataset['Fare'].astype(int)
```

Przygotowane dane zostały pokazane poniżej.

```
print(train_df.head(10))
```

Survived	Pclass	Sex	Age	Fare	Embarked	Title	IsAlone
0	3	0	22	0	0	1	0
1	1	1	38	3	1	3	0
1	3	1	26	1	0	2	1
1	1	1	35	3	0	3	0
0	3	0	35	1	0	1	1
0	3	0	25	1	2	1	1
0	1	0	54	3	0	1	1
0	3	0	2	2	0	4	0
1	3	1	27	1	0	3	0
1	2	1	14	2	1	3	0

Do treningu modeli użyjemy kilku funkcji biblioteki `sklearn`. Pozwoli nam to porównać trafność wybranych algorytmów, tak jak to się robi w projektach uczenia maszynowego. Zaczniemy od przygotowania danych treningowych. Zmienne wejściowe i wyjściową przekazuje się do funkcji uczenia maszynowego z biblioteki `sklearn` jako dwa osobne parametry, więc nasz zbiór treningowy podzielimy na dane wejściowe `X` i wyjściowe `Y`. Ponadto utworzymy kopię danych testowych, z której usuniemy zmienną `PassengerId`.

```
X = train_df.drop("Survived", axis=1)
Y = train_df["Survived"]
X_test = test_df.drop("PassengerId", axis=1).copy()
```

Ze zbiorów `X` i `Y` wyodrębnimy dane walidacyjne. Ponieważ danych jest mało, podzielimy je w proporcji 70–30.

```
X_train, X_validation, Y_train, Y_validation = train_test_split(X, Y, test_size=0.3,
random_state=123)
```

Utworzenie modelu sprowadza się do wyboru algorytmu uczenia maszynowego.

```
logreg = LogisticRegression()
```

Tak utworzony model należy nauczyć, używając do tego danych treningowych.

```
logreg.fit(X_train, Y_train)
```

Możemy już sprawdzić trafność naszego modelu. Najpierw policzymy błąd empiryczny (treningowy).

```
acc_log = round(logreg.score(X_train, Y_train) * 100, 2)
print("LogisticRegression train score: %2.2f" % acc_log)
```

```
LogisticRegression train score: 82.34
```

Policzenie błędu testowego wymaga wykonania predykcji dla zbioru walidacyjnego i porównania wyników zwróconych przez model z rzeczywistymi wartościami zmiennej `Survived`.

```
Y_pred = logreg.predict(X_validation)
acc_log = round(logreg.score(X_validation, Y_validation) * 100, 2)
print("LogisticRegression test score: %2.2f" % acc_log)
```

```
LogisticRegression test score: 81.34
```

Błąd realny jest tylko o 1% większy od empirycznego, czyli model regresji liniowej nie dopasował się nadmiernie do danych treningowych. Omówiona w rozdziale 3 regresja liniowa jest prostym modelem, więc ryzyko jej nadmiernego dopasowania jest małe.

Niestety, błąd testowy modelu regresji liniowej jest duży i wynosi 19%. **Modele o małej pojemności, takie jak modele regresji liniowej, cechuje duży błąd systematyczny** (*bias*). Błąd systematyczny definiuje się jako spodziewany błąd generalizacji modelu, nawet gdybyśmy dysponowali nieskończenie dużą liczbą przykładów treningowych. Możemy sprawdzić, że po zwiększeniu zbioru treningowego do 99% przykładów, błąd testowy modelu regresji liniowej niewiele się zmieni i będzie wynosił 18%. Za to zmniejszenie zbioru treningowego do 20% przykładów spowoduje zmniejszenie błędu testowego do 16%. Wygląda na to, że model regresji liniowej nie jest w stanie lepiej nauczyć się na naszych danych treningowych.

Sprawdźmy, czy inne algorytmy uczenia maszynowego lepiej poradzą sobie z tym problemem. W tym celu porównamy jakość modelu maszyny wektorów nośnych (zaimplementowaną w funkcji `SVC`), klasyfikatora *k*-najbliższych sąsiadów (funkcja `KNeighborsClassifier`), drzew decyzyjnych (funkcja `DecisionTreeClassifier`) i lasu drzew decyzyjnych (funkcja `RandomForestClassifier`). Praca ze wszystkimi funkcjami uczenia maszynowego biblioteki `sklearn` wygląda tak samo, a więc nie ma potrzeby powtarzania w książce takich samych instrukcji:

- model należy utworzyć, wybierając odpowiednią funkcję uczenia maszynowego;
- metoda `fit` modelu pozwala wytrenować model;
- metoda `score` ocenia model na podstawie domyślnego kryterium;
- metoda `predict` wykonuje zapytanie predykcyjne.

W tabeli 5.1 podano otrzymane wyniki.

Tabela 5.1. Patrząc na powyższe wyniki, można podejrzewać, że wszystkie modele, z wyjątkiem modelu regresji liniowej, nadmiernie dopasowały się do danych treningowych

Model	Trafność zmierzona dla danych trenigowych	Trafność zmierzona dla danych testowych
Regresja logistyczna	82,34	81,34
Maszyna wektorów nośnych	86,36	82,09
Klasyfikator *k*-najbliższych sąsiadów	87,64	80,60
Drzewa decyzyjne	94,06	82,84
Las drzew decyzyjnych	91,97	81,34

Błąd empiryczny modeli drzew decyzyjnych i lasu drzew decyzyjnych jest mały. Niestety, ich błąd rzeczywisty jest ponad dwukrotne większy od empirycznego.

Modele te cechuje mały błąd systematyczny, ale duża zmienność predykcji. Duża zmienność najczęściej oznacza nadmierne dopasowanie modelu. Spróbujmy poprawić stabilność wybranego modelu, nie zwiększając jednocześnie jego błędu systematycznego.

Najlepiej wypadły drzewa decyzyjne, a więc decydujemy się na użycie tego algorytmu. Musimy jednak rozwiązać problem nadmiernego dopasowania modelu. Sprawdźmy, czy jego powodem nie jest, jak to często bywa, klątwa wymiarowości. Wiemy, że zmienna `Embarked` nie jest liniowo skorelowana ze zmienną wyjściową `Survived`, więc usuńmy ją i powtórzmy trening oraz ocenę modelu.

```
train_df = train_df.drop(['Embarked'], axis=1)
test_df = test_df.drop(['Embarked'], axis=1)

decision_treeE = DecisionTreeClassifier(criterion="gini")
decision_treeE.fit(X_train, Y_train)

acc_decision_treeE = round(decision_treeE.score(X_train, Y_train) * 100, 2)
print("Decision Tree without Embarked train score: %2.2f" % acc_decision_treeE)

Y_pred = decision_treeE.predict(X_validation)
acc_decision_treeE = round(decision_treeE.score(X_validation, Y_validation) * 100, 2)
print("Decision Tree without Embarked test score: %2.2f" % acc_decision_tree)
```

```
Decision Tree without Embarked train score: 94.06
Decision Tree without Embarked test score: 83.58
```

Błąd empiryczny się nie zmienił i nadal wynosi 6%. Za to błąd rzeczywisty spadł o jeden procent. Usunięcie zmiennej poprawiło jakość modelu, ale nadal jest on nadmiernie dopasowany. Musimy więc go uprościć, ograniczając głębokość i szerokość drzewa. Więcej informacji na temat hiperparametrów modeli uczenia maszynowego znajduje się w dalszej części książki. Na razie zauważę tylko, że ograniczając głębokość drzew decyzyjnych zmniejszamy liczbę możliwych podziałów według wybranych zmiennych, zmuszając model

do wcześniejszego podjęcia decyzji. Otrzymane w ten sposób płytkie drzewa decyzyjne cechuje mniejsza zmienność, ale większy błąd systematyczny.

```
decision_tree = DecisionTreeClassifier(criterion="gini", max_depth=8, min_samples_split=4)
```

Błąd empiryczny tak zdefiniowanego modelu wyniósł 89,9, a błąd rzeczywisty – 84,70. Udało nam się trzykrotnie zmniejszyć różnicę między błędami empirycznym a testowym – z 12 do 4% i zmniejszyć błąd rzeczywisty o 2%.

Powinniśmy jednak pamiętać, że dysponujemy bardzo małą liczbą przykładów. Bardziej wiarygodną ocenę modelu uzyskamy metodą *k*-krotnej walidacji krzyżowej. Do jej przeprowadzenia użyjemy kolejnych dwóch funkcji tej samej biblioteki `sklearn`: `KFold` do podzielenia danych na partycje i `cross_val_score` do treningu i oceny.

```
kfold = model_selection.KFold(n_splits=10, random_state=12)
cv_decision_tree = DecisionTreeClassifier(criterion="gini", max_depth=8,
min_samples_split=4)

results = model_selection.cross_val_score(cv_decision_tree, X, Y, cv=kfold)
print("Decision Tree accuracy: Final mean:%.3f%%, Final standard deviation:(%.3f%%)" %
(results.mean()*100.0, results.std()*100.0))
print('Decision Tree accuracies from each of the 10 folds using kfold:',results)
```

```
Decision Tree accuracy: Final mean:79.574%, Final standard deviation:(3.282%)
Decision Tree accuracies from each of the 10 folds using kfold: [0.78888889 0.82022472
0.73033708 0.79775281 0.84269663 0.82022472, 0.7752809  0.75280899 0.82022472 0.80898876]
```

Nasz model nie jest zbyt stabilny. Jego trafność waha się od 73 do 84%, co pewnie wynika z bardzo małej liczby przykładów treningowych. Średnio możemy spodziewać się dokładności na poziomie 79,5%. Taki model spełnia nasze wymagania.

Przetrenujmy wybrany model na wszystkich danych treningowych i sprawdźmy, jak ocenia on dane testowe.

```
cv_decision_tree.fit(X,Y)
Y_test = cv_decision_tree.predict(X_test)
print(Y_test)
```

```
[0 0 0 0 1 …]
```

Utworzony prototyp należy zaimplementować po stronie serwera SQL Server. Procedura `[Titanic].[TrainSurvivedDTClassifier]` zawiera omówiony kod w języku Python. Niezbędne zmiany sprowadziły się do:

- Przekazania do serwera SQL Server wytrenowanego modelu w postaci binarnej oraz jego metryk jako danych typu `float` (`float`, w przeciwieństwie do typu `decimal`, jest jednym ze standardowych typów danych w języku Python, a między serwerem SQL a interpretatorem języka Python mogą być przekazywane tylko zmienne standardowych typów języka).

```
CREATE OR ALTER PROCEDURE [Titanic].[TrainSurvivedDTClassifier] (@trained_model
varbinary(max) OUTPUT , @score float OUTPUT, @deviation float OUTPUT))
AS
BEGIN
     EXECUTE sp_execute_external_script
      @language = N'Python'
     ,@script = N'
```

- Skrócenia listy importowanych bibliotek do pięciu (biblioteka `panads` jest importowana automatycznie).

```
import numpy as np
import pickle
from sklearn.tree import DecisionTreeClassifier
from sklearn import model_selection
from sklearn.model_selection import train_test_split
```

- Pobrania danych treningowych zwróconych przez zapytanie (dane testowe nie są przetwarzane przez tę procedurę).

```
train_df = train_data
…

,@input_data_1 = N'SELECT * FROM [Titanic].[Train]'
,@input_data_1_name = N'train_data'
```

- Usunięcia niepotrzebnych kolumn, w tym kolumny `Embarked`, i skasowania instrukcji uzupełniających i mapujących wartości zmiennej `Embarked`.

```
train_df = train_df.drop([''Ticket'', ''Cabin'', ''PassengerId'',''Embarked''], axis=1)
```

- Zwrócenia średniej dokładności i odchylenia standardowego dokładności poszczególnych modeli jako zmiennych typu `float`.

```
results = model_selection.cross_val_score(decision_tree, X_train, Y_train, cv=kfold)
score = float(results.mean())
deviation = float(results.std())
…
,@params = N'@trained_model varbinary(max) OUTPUT, @score float OUTPUT, @deviation float
OUTPUT'
,@trained_model = @trained_model OUTPUT
,@score = @score OUTPUT
,@deviation = @deviation OUTPUT;
```

- Zapisaniu wytrenowanego modelu w postaci ciągu bajtów.

```
trained_model = pickle.dumps(decision_tree)
```

Wytrenowanie modelu sprowadza się do wywołania procedury i zapisania jej wyników w tabeli.

```
DECLARE @model VARBINARY(MAX), @score float, @deviation float;
EXEC [Titanic].[TrainSurvivedDTClassifier] @model OUTPUT, @score OUTPUT, @deviation OUTPUT
INSERT INTO Titanic.Models
VALUES('DecisionTree max_depth=8, min_samples_split=4' ,@model, @score, @deviation);
```

Sprawdźmy, jak wygląda wytrenowany model.

```
SELECT name, score, deviation, left(model,15) as model
FROM Titanic.Models;
```

```
name                                                score    deviationmodel
DecisionTree max_depth=8, min_samples_split=4       0.802    0.038            €csklearn.
tree
```

Do wykonywania zapytań predykcyjnych potrzebna nam będzie osobna procedura. Procedura `[Titanic].[PredictSurvived]`:

- Odczytuje z tabeli `[Titanic].[Models]` model o podanej nazwie.

```
CREATE OR ALTER PROCEDURE [Titanic].[PredictSurvived] (@model varchar(250))
AS
BEGIN
DECLARE @lmodel varbinary(max) =
(SELECT model FROM [Titanic].[Models] WHERE [name] = @model);
…
```

- Przekazuje do skryptu w języku Python dane testowe i wybrany model, a zwraca dane testowe uzupełnione o obliczoną przez ten model zmienną `Survived`.

```
@input_data_1 = N'SELECT * FROM [Titanic].[Test]',
@input_data_1_name = N'InputDataSet',
@params = N'@lmodel varbinary(max)',
@lmodel = @lmodel
WITH RESULT SETS ((PassengerId int, Pclass int, Name nvarchar(100), Sex nvarchar(100),
Age float, SibSp int, Parch int, Ticket nvarchar(50), Fare float, Cabin nvarchar(50),
Embarked nvarchar(50), Survived int));
```

- Deserializuje odczytany model.

```
mod = pickle.loads(lmodel)
```

- Przekształca zmienne dokładnie w taki sam sposób, jak procedura `[Titanic].[TrainSurvivedDTClassifier]`.
- Wykonuje zapytanie predykcyjne i zapisuje jego wynik w zmiennej lokalnej.

```
Y_pred = mod.predict(X_test)
```

- Łączy przekazane do niej dane ze zmienną zawierającą wyniki predykcji i zwraca tak utworzoną tabelę serwerowi SQL Server.

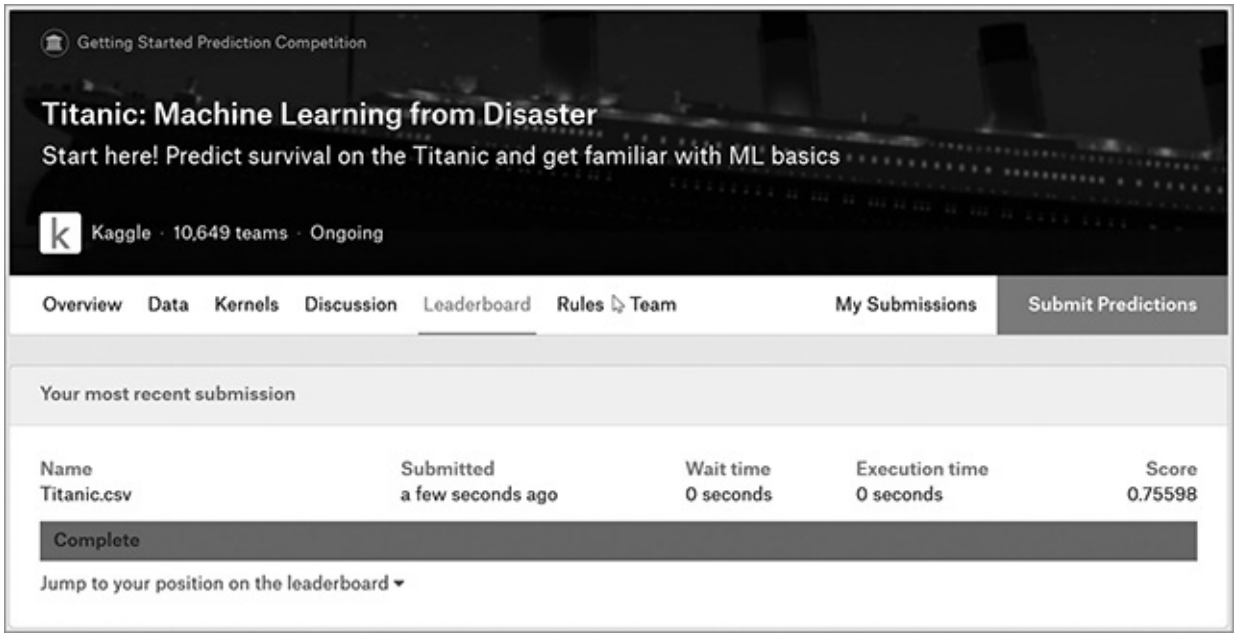
```
df1 = pandas.DataFrame(data=InputDataSet)
df2 = pandas.DataFrame(data=Y_pred)
OutputDataSet = pandas.concat([df1,df2],axis=1,join="inner")
```

Wywołując procedurę, otrzymamy listę pasażerów Titanica uzupełnioną o informację o tym, czy przeżyli oni katastrofę.

```
EXEC [Titanic].[PredictSurvived] 'DecisionTree max_depth=8, min_samples_split=4';
```

PassengerId	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare
Cabin	Embarked	Survived						
918	1	Ostby, Miss. Helene Ragnhild	female	22	0	1		
113509	61.9792	B36	C	1				
919	3	Daher, Mr. Shedid	male	22.5	0	0	2698	7.225
NULL	C	0						
920	1	Brady, Mr. John Bertram	male	41	0	0	113054	
30.5	A21	S	0					
921	3	Samaan, Mr. Elias	male	NULL	2	0	2662	21.6792
NULL	C	0						
…								

Wyników tych predykcji nie możemy zweryfikować inaczej, niż wysyłając je do serwisu Kaggle. Te dane pochodzą z pliku *test.csv*, w którym nie było zmiennej `Survived`. Zgłaszane wyniki powinny mieć format pliku *csv* z dwoma kolumnami: `PassengerId` i `Survived`. Po jego wgraniu (wysłanie wyników konkursu wymaga założenia konta w serwisie Kaggle) zostaną one automatycznie ocenione. Trafość naszego modelu zmierzona dla danych nie użytych ani do treningu ani do walidacji wyniosła 76% (rys. 5.11).



Rysunek 5.11. Chociaż wynik 76% nie wystarczył do zajęcia wysokiego miejsca (z sobie tylko znanych powodów wielu uczestników tego konkursu oszukuje, używając dostępnych w Internecie danych o przeżyciu wszystkich pasażerów Titanica), to odnieśliśmy sukces – zbudowaliśmy model, którego trafność oceniliśmy na 79%, plus/minus 3%, i po „wdrożeniu" modelu do użycia okazało się, że zmieściliśmy się w granicy błędu

Na koniec zastanówmy się, o ile dokładniejszy jest nasz model od zgadywania, czy ktoś przeżył, czy zginął w tej katastrofie. W danych treningowych procent przykładów, dla których zmienna `Survived` ma wartość 0, wynosi 62%. Sensowna strategia zgadywania, polegająca na odpowiadaniu, że osoba, o której nic nie wiemy, zginęła w katastrofie, będzie prawidłowa w 62%. Nasz model jest o 13% dokładniejszy.

Zachęcam do samodzielnego zbudowania modelu o większej dokładności. Proponuję skoncentrować się na ocenie i przygotowaniu danych – na przykład dodać zmienną binarną `IsMother` informującą o tym, czy pasażerka była na pokładzie z przynajmniej jednym dzieckiem. Może warto zbilansować dane? A może dyskretyzacja zmiennej `Age` wzmocni siłę predykcyjną tej zmiennej? Proszę pamiętać, poprawa dokładności nawet o 1% jest ważna i często może oznaczać różnice między udanym wdrożeniem a wdrożeniem zakończonym

niepowodzeniem.

Kiedy jako dzieci uczymy się patrzeć, nie dostajemy prawidłowych odpowiedzi – po prostu patrzymy. Od czasu do czasu wmama powie „To jest pies”, i to wszytko.

Geoffrey Hinton, profesor informatyki na uniwersytecie w Toronto, uczestnik projektu Google Brain, 1996

Rozdział 6
Analiza skupień

Ten rozdział różni się od pozostałych. Do tej pory poznawaliśmy narzędzia, algorytmy nadzorowanego uczenia maszynowego oraz metody opisywania i przygotowywania danych, a w tym rozdziale przyjrzymy się popularnemu algorytmowi uczenia nienadzorowanego.

Prace nad algorytmami uczenia nienadzorowanego są mniej intensywne i nie tak zaawansowane, jak prace nad przedstawionymi w rozdziale 3 algorytmami uczenia nadzorowanego. Algorytmy uczenia nienadzorowanego są też trudniejsze w użyciu i znacznie rzadziej stosowane w praktyce.

Głównym problemem jest brak oczywistych metryk pozwalających ocenić jakość modeli uczenia nienadzorowanego. Skoro w danych treningowych nie ma odpowiedzi, to nie ma z czym porównać wyników predykcji. Największa słabość modeli uczenia nienadzorowanego może jednak okazać się ich największą zaletą. Opisanie przykładów wymaga ogromnego nakładu pracy. Gdyby udało się opracować algorytmy nienadzorowanego uczenia maszynowego o jakości porównywalnej z algorytmami uczenia nadzorowanego, byłby to prawdziwy przełom. Wielu badaczy sztucznej inteligencji, w tym Geoffrey Hinton, uważa, że to właśnie algorytmy uczenia nienadzorowanego,

lub częściowo nadzorowanego, doprowadzą do zbudowania uniwersalnych (wielozadaniowych) modeli uczenia maszynowego.

Przedstawiony w tym rozdziale materiał ogranicza się do analizy skupień (grupowania). Oprócz omówienia algorytmu grupowania metodą *k*-średnich i przedstawienia przykładowego zastosowania analizy skupień do segmentacji klientów pokazuje on, jak ważne jest graficzne przedstawienie danych treningowych.

6.1. Grupowanie w celu zmniejszenia liczby przykładów

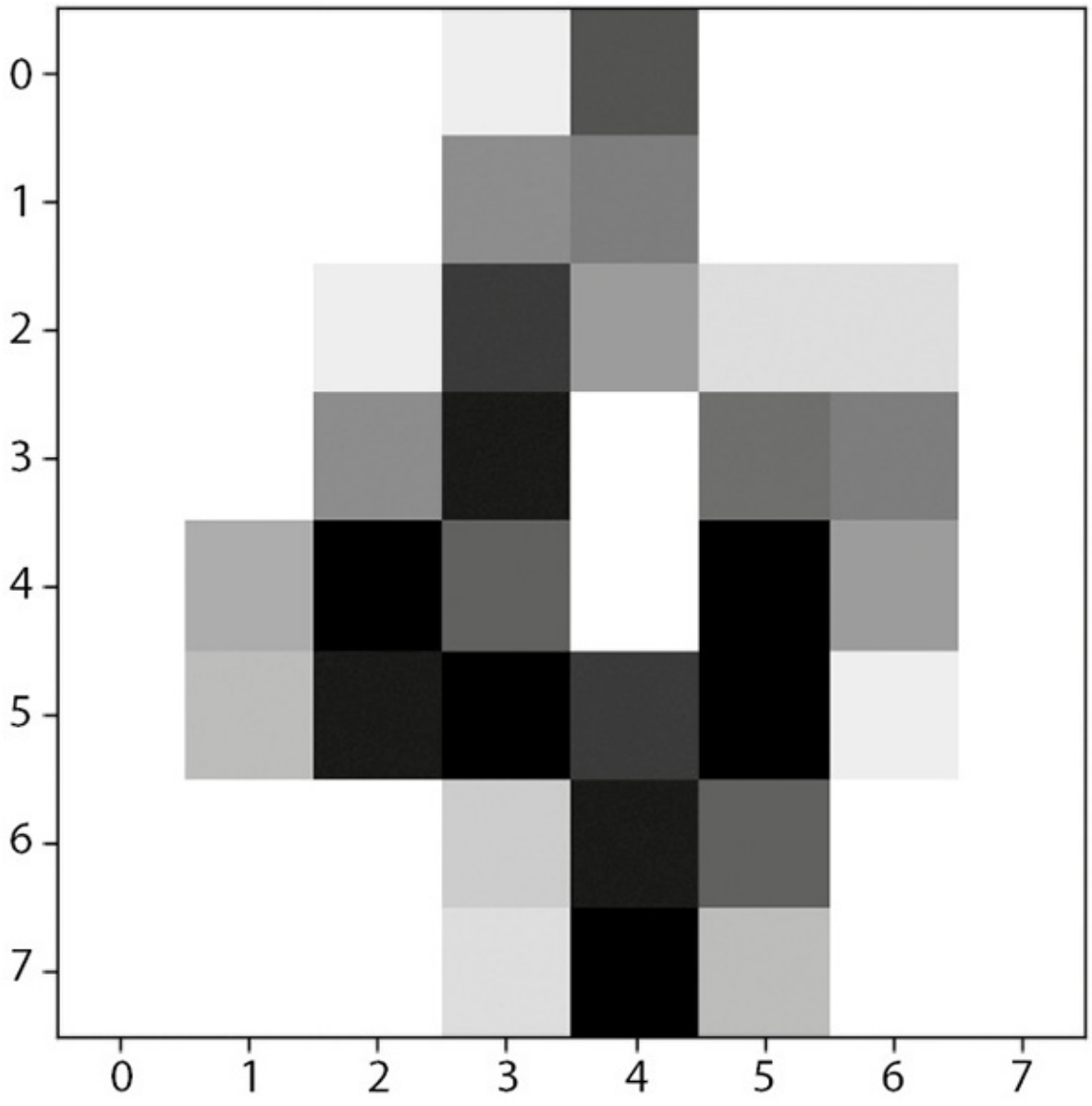
Pierwszy model uczenia nienadzorowanego poznaliśmy w punkcie 5.5.2. Jak pamiętamy, metoda *PCA* pozwala wielokrotnie zredukować liczbę zmiennych (np. z 5000 do 20), w taki sposób, że nowe zmienne (czyli składowe główne) zawierają większość (zwykle wybiera się tyle składowych głównych, żeby zachować ponad 90%) informacji ukrytych w oryginalnych zmiennych.

Grupowanie pozwala przeprowadzić redukcję danych o podobnej skali, ale wyniki modelowania, czyli klastry, zastępują oryginalne przypadki. Zmniejszeniu ulega więc liczba wierszy, a nie kolumn tabeli z danymi. Zanim poznamy budowę i działanie algorytmu grupowania metodą *k*-średnich, zobaczmy efekt jego zastosowania w celu zmniejszenia liczby przykładów.

W tym celu użyjemy zbioru 1797 odręcznie zapisanych, czarno–białych cyfr. Każda cyfra jest reprezentowana przez 64 piksele, czyli cyfry są malutkimi obrazkami o rozmiarach 8 × 8 pikseli. Przykładową cyfrę pokazano na rysunku 6.1.

```
digits = load_digits()
print(digits.data.shape)
plt.imshow(digits.images[4], cmap=plt.cm.gray_r)
plt.show()
```

```
(1797, 64)
```



Rysunek 6.1. Jedna z odręcznie zapisanych cyfr 4

W danych nie ma informacji o tym, co to za cyfra. Czyli dysponujemy nieopisanymi przykładami. Naszym zadaniem jest automatyczny podział

przykładów na dziesięć grup o podobnych cechach, czyli na **klastry** lub skupienia. Cechami, według których pogrupujemy przykłady, są poszczególne piksele, a ich wartościami – odcienie szarości.

Skoro przykłady są nieopisane, to jak model może się nauczyć dzielić je na grupy? Zastosowany algorytm *k*-średnich podzieli przykłady bez żadnej wcześniejszej wiedzy na temat tego, jak docelowe klastry powinny wyglądać, wyłącznie na podstawie przyjętej funkcji podobieństwa. Wystarczy, że będzie się kierował zasadą, według której **przykłady w ramach tych samych klastrów powinny być maksymalnie do siebie podobne i maksymalnie niepodobne do przykładów, które trafiły do pozostałych klastrów**.

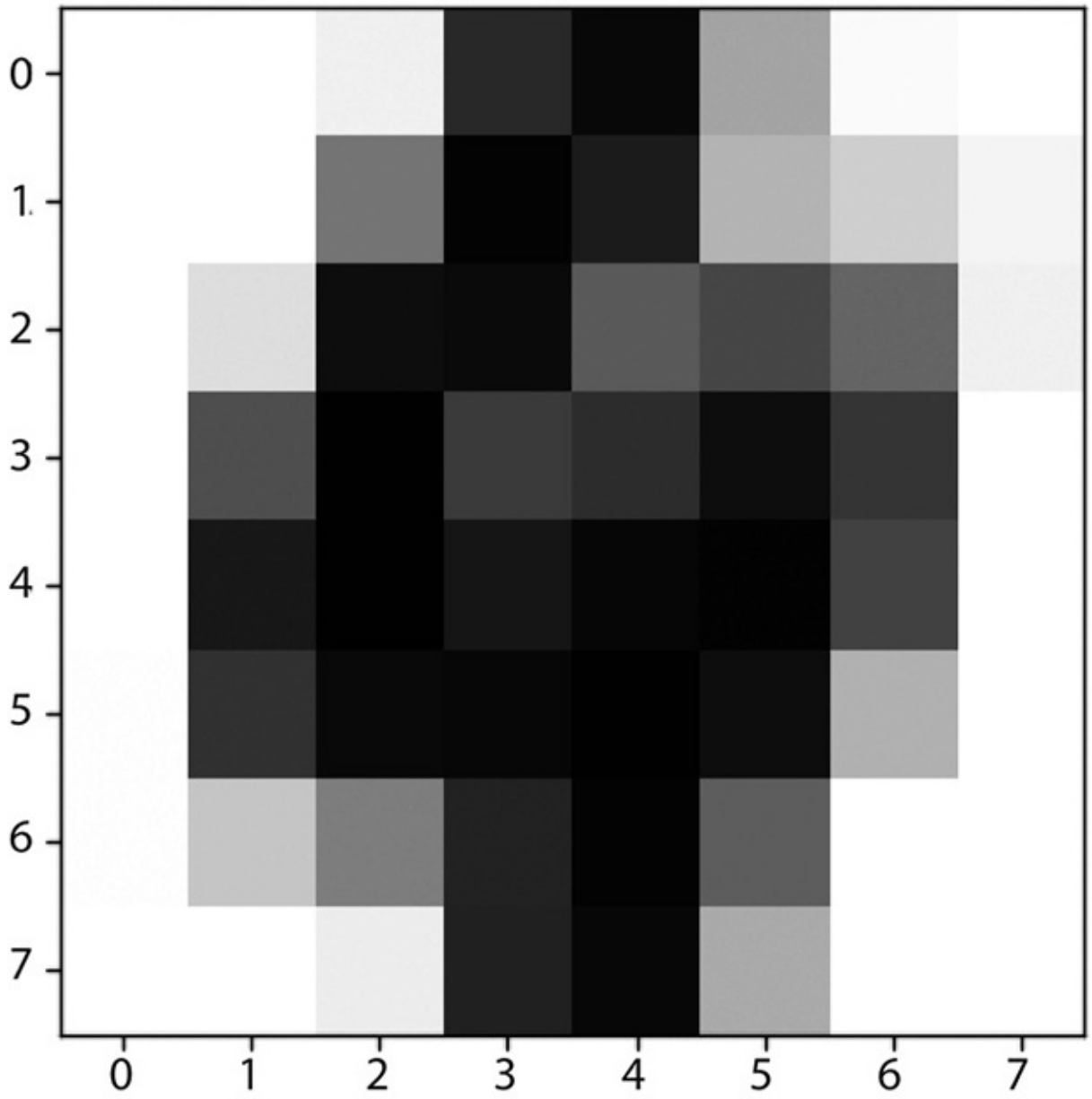
Zobaczmy, jak model grupowania metodą *k*-średnich poradzi sobie z zadaniem podziału cyfr. Zbudowanie modelu (zaimplementowanego w funkcji `kmeans` biblioteki `sklearn`) wymaga podania liczby klastrów. W tym przypadku wiemy, że przykłady dzielą się na 10 klastrów, bo są to odręcznie zapisane cyfry od 0 do 9. Następnie wytrenujemy model, używając przykładowego zbioru `digits`. W wyniku otrzymamy 10 klastrów, środkiem każdego z nich będzie znaleziony wzorzec cyfry o rozmiarze 8 × 8 pikseli.

```
kmeans = kmeans(n_clusters=10, random_state=0)
clusters = kmeans.fit_predict(digits.data)
print(kmeans.cluster_centers_.shape)
```

```
(10, 64)
```

Zredukowaliśmy liczbę przykładów z 1797 do 10. Sprawdźmy, jak wygląda znaleziony przez model wzorzec cyfry 4[12] (rys. 6.2).

```
centers = kmeans.cluster_centers_.reshape(10, 8, 8)
plt.imshow(centers[0] ,cmap=plt.cm.gray_r)
plt.show()
```



Rysunek 6.2. Rzeczywiście, środkiem znalezionego klastra jest wzorzec cyfry 4

Algorytm grupowania metodą *k*-średnich nauczył się rozpoznawać odręczne cyfry, chociaż do treningu użyliśmy nieopisanych danych! Żeby w pełni docenić to osiągnięcie, musimy wiedzieć, jak maszyny widzą obrazy i jak działają iteracyjno-optymalizacyjne algorytmy grupowania, do których zalicza się algorytm grupowania metodą *k*-średnich.

Dla maszyny czarno–biały obraz to wektor liczb z zakresu 0–16, z których każda

opisuje poziom czerni jednego piksela. Pokazaną na rysunku 6.1 cyfrę 4 maszyny widzą tak (dla poprawy czytelności liczby zostały ułożone w macierz 8 × 8, faktyczne dane zwraca instrukcja `print(digits.data[4])`):

```
print(digits.images[4])
```

```
[[ 0.  0.  0.  1. 11.  0.  0. 0.]
 [ 0.  0.  0.  7.  8.  0.  0. 0.]
 [ 0.  0.  1. 13.  6.  2.  2. 0.]
 [ 0.  0.  7. 15.  0.  9.  8. 0.]
 [ 0.  5. 16. 10.  0. 16.  6. 0.]
 [ 0.  4. 15. 16. 13. 16.  1. 0.]
 [ 0.  0.  0.  3. 15. 10.  0. 0.]
 [ 0.  0.  0.  2. 16.  4.  0. 0.]]
```

Każda liczba jest osobną zmienną. Przykładowe, miniaturowe obrazy opisane były sześćdziesięcioma czteroma zmiennymi wejściowymi. W tym miejscu warto zauważyć, że obrazy kolorowe to macierze liczb z zakresu 0–255, z których każda opisuje jedną z trzech składowych (R,G,B) danego piksela. Oznacza to, że zdjęcie wykonane w niskiej rozdzielczości 800 × 600 pikseli zostanie zapisane jako macierz 800 × 600 × 3, czyli macierz miliona czterystu czterdziestu tysięcy liczb. Wiemy już, że każdy piksel (a właściwie reprezentujące go liczby) to trzy osobne zmienne wejściowe. To dlatego przetwarzanie obrazów wymaga nie tylko sporej mocy obliczeniowej (w praktyce użycia procesorów graficznych), ale również zmierzenia się z ekstremalną postacią klątwy wymiarowości.

6.2. Algorytmy iteracyjno-optymalizacyjne

Algorytmy iteracyjno-optymalizacyjne najpierw dzielą przykłady na z góry określoną liczbę klastrów, a następnie iteracyjnie przenoszą obiekty między klastrami, tak aby uzyskać jak najbardziej zwarte i odległe od siebie klastry.

Najczęściej stosowanym algorytmem grupowania iteracyjno-optymalizacyjnego jest algorytm grupowania metodą *k*-średnich. Zaletami tego algorytmu są jego prostota i wydajność – jest to algorytm o złożoności rzędu *O(n)*, czyli jego koszt rośnie liniowo wraz ze zwiększeniem liczby przykładów. Dla porównania, algorytmy grupowania hierarchicznego, które sekwencyjnie grupują obiekty, tworząc drzewo klastrów, czyli dendrogram, są złożoności rzędu $O(n^2)$.

Grupowanie metodą *k*-średnich przebiega następująco:

1. *k* losowo wybranych przykładów zostaje uznanych za centra klastrów.
2. Pozostałe przykłady zostają przypisane do najbliższego im klastra, czyli do klastra, od centrum którego znajdują się najbliżej.
3. Obliczony zostaje nowy środek klastra jako średnia pozycji przykładów teraz do niego należących.

Drugi i trzeci krok są powtarzane aż do zbudowania końcowego modelu. Jeżeli określiliśmy minimalną liczebność klastrów, a w jakimś klastrze zostało mniej przykładów, zostanie on usunięty z modelu. Trening kończy się, jeżeli w ostatniej iteracji żaden z przykładów nie zmienił przynależności do klastra.

Taki sposób działania oznacza, że algorytm grupowania metodą *k*-średnich może zostać zastosowany, o ile:

- przykłady dzielą się naturalnie na rozłączne grupy, a mówiąc dokładniej – jeżeli cechy przykładów z jednej grupy są odseparowane od cech przykładów z innych grup; to samo założenie można sformułować następująco: **zbiory cech przykładów z różnych grup muszą być rozłączne***;* jeśli to założenie nie będzie spełnione, algorytm przydzieli niejednoznaczne przykłady do przypadkowych klastrów;
- kształt każdej naturalnej grupy przykładów jest regularny, zbliżony do sferycznego; jeśli to założenie nie będzie spełnione, przykłady najbardziej

oddalone od centrów swoich naturalnych grup trafią do klastrów reprezentujących inne grupy;

- w każdej naturalnej grupie znajduje się zbliżona liczba przykładów; w innym przypadku mniej liczne grupy mogą zostać wchłonięte do klastrów reprezentujących liczniejsze grupy;
- każda naturalna grupa jest zbliżonej wielkości; w innym przypadku mniejsze grupy mogą zostać wchłonięte do klastrów reprezentujących większe grupy.

Wszystkie te założenia rzadko są spełnione. W takich przypadkach lepiej jest zastosować inną metodę grupowania, na przykład algorytm *E-M* (*Expectation–Maximization*). Sprawdźmy, czy zbiór danych o odręcznie zapisanych cyfrach założenia spełniał założenia algorytmu grupowania metodą *k*-średnich:

- każda cyfra wygląda inaczej od pozostałych – założenie rozłączności naturalnych grup zostało spełnione;
- cyfry nie mają kształtów sferycznych – założenie regularności kształtów nie zostało spełnione; z tego powodu około 20% przykładów trafiło do niewłaściwych klastrów (reprezentujących inne cyfry o podobnym wyglądzie);
- w analizowanym zbiorze danych każda cyfra występowała w prawie takiej samej liczbie przykładów – założenie równoliczności naturalnych grup zostało spełnione;
- każda cyfra była podobnej wielkości – założenie podobnej wielkości naturalnych grup zostało spełnione.

W zrozumieniu założeń algorytm grupowania metodą *k*-średnich pomoże nam kolejne ćwiczenie, w którym liczbę zmiennych wejściowych (wymiarów) ograniczymy do dwóch, co pozwoli nam zwizualizować dane i wyniki ich grupowania. Przy tej okazji poznamy też miary (nie)podobieństwa przykładów

i metody inicjowania modeli grupujących.

Pogrupujmy irysy wyłącznie na podstawie długości i szerokości ich płatków. Po usunięciu dwóch pierwszych kolumn tabeli `iris.data` pozostałe dwie zmienne użyjemy do wytrenowania modelu. Przykłady podzielimy na trzy klastry. Wcześniej napisałem, że uczenie zaczyna się od wybrania *k* losowych przykładów i uznania ich za środki klastrów. Zastanówmy się, czy sposób zainicjowania modelu ma wpływ na wyniki uczenia.

Wystarczy przypomnieć sobie zasadę działania algorytmu grupowania metodą *k*-średnich, żeby zauważyć, że tak, ma wpływ i to zdecydowany. Niektóre początkowe lokalizacje klastrów są lepsze od innych. Skoro naszym celem jest uzyskanie jak najbardziej zwartych i odległych od siebie klastrów, to **początkowe centra powinny być od siebie maksymalnie odległe**.

Żeby wybrać takie odlegle od siebie przykłady, musimy mieć jakiś sposób mierzenia dzielących je odległości. Zdefiniowanie miary odległości jest też niezbędne do treningu modelu. Trening polega przecież na przypisywaniu przykładów do najbliższych klastrów. Potrzebujemy miary podobieństwa (lub niepodobieństwa) przykładów.

Jeżeli cechy przykładów są zmiennymi numerycznymi, ich (nie)podobieństwo można zdefiniować jako odległość euklidesową między obiektami x_i a x_j

$$d_E(i, j) = \sqrt{\sum_{n=1}^{k} (X_i - X_j)^2}$$

gdzie k oznacza wartość k-tego atrybutu x_i. Zamiast euklidesowej miary odległości możemy też posłużyć się odległością Manhattan wyrażoną wzorem

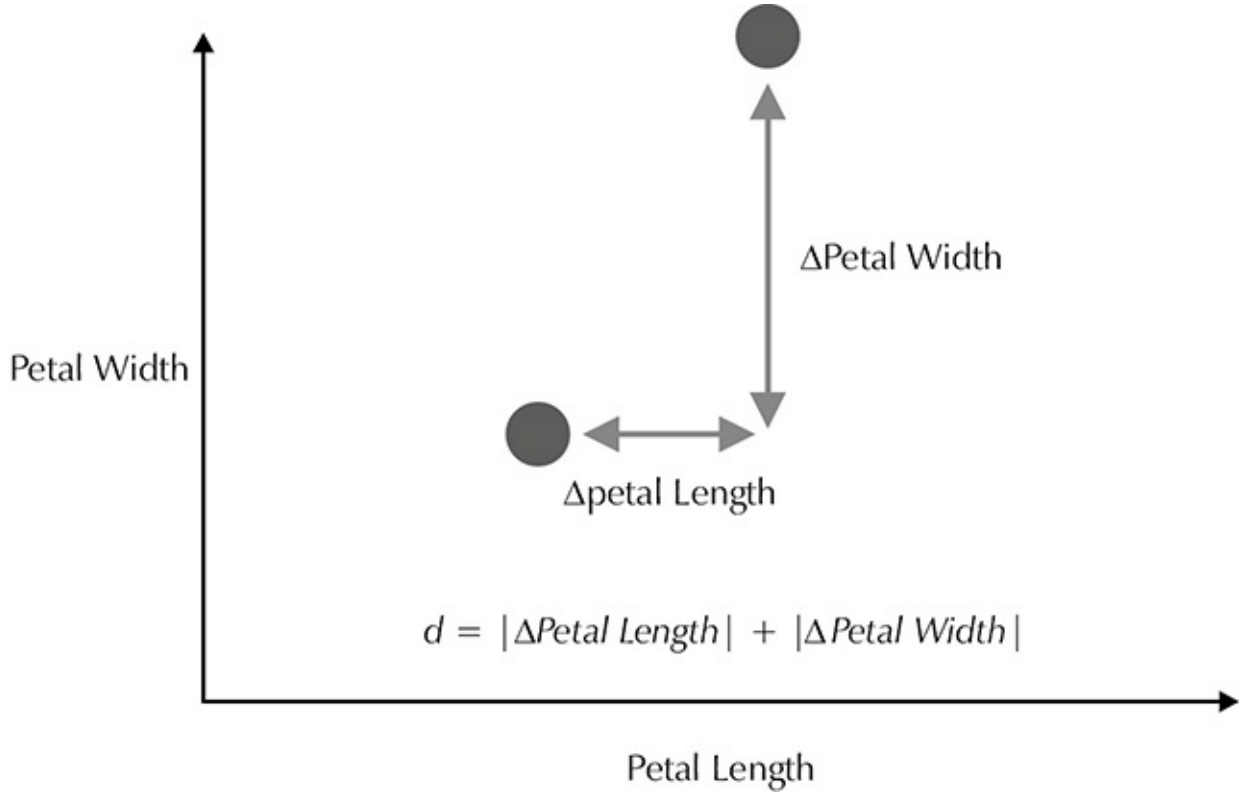
$$d_{Ma}(i, j) = \sum_{n=1}^{k} |x_i - x_j|$$

Obie te miary są specjalnymi przykładami odległości Minkowskiego

$$d_{Mi}(i, j) = (\Sigma_{n=1}^{k}(|x_i - x_j|)^q)^{1/q}$$

Odległość Manhattan otrzymamy dla $q = 1$, a odległość euklidesową dla $q = 2$ (rys. 6.3).

Inicjowanie modelu metodą *k*-means++ polega na wylosowaniu przykładu, który będzie centrum pierwszego klastra, a następnie znalezieniu maksymalnie odległego przykładu i wybraniu go jako centrum drugiego klastra. Na centrum trzeciego klastra zostanie wybrany przykład maksymalnie odległy od dwóch już wybranych, itd. Metoda ta sprawdza się w praktyce, ale nadal nie gwarantuje optymalnego układu początkowego klastrów. Dlatego zwykle tworzy się wiele modeli kandydujących (modeli zainicjowanych na różne sposoby) i wybiera najlepszy. Liczbę modeli kandydujących kontroluje parametr `n_init`.

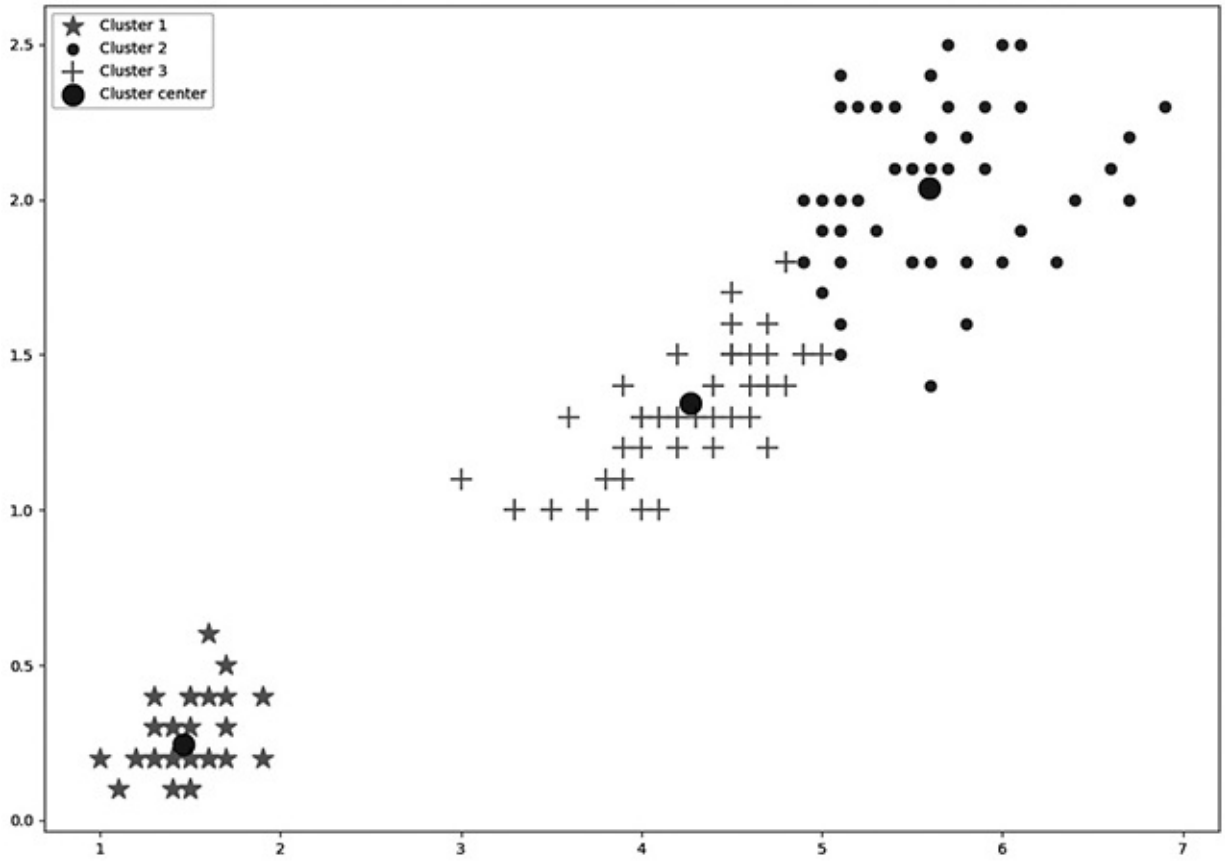


Rysunek 6.3. Odległość Manhatann między dwoma przykładami wyznaczona na podstawie różnic wielkości ich płatków

```
iris = datasets.load_iris()
x = iris.data
x = np.delete(x, np.s_[0:2], axis=1)
kmeans = kmeans(n_clusters = 3, init = 'k-means++', n_init = 10, random_state = 0)
```

Następnie nauczymy model i przypiszemy przykłady do klastrów, używając funkcji `fit_predict`. Na wykresie punktowym narysujemy przykłady, zaznaczając klaster, do którego zostały one przypisane, i uzupełnimy wykres o centra znalezionych klastrów (rys. 6.4).

```
y = kmeans.fit_predict(x)
plt.scatter(x[y == 0, 0], x[y == 0, 1], s = 200, c = 'red', label = 'Iris-versicolour',
marker= '*')
plt.scatter(x[y == 1, 0], x[y == 1, 1], s = 200, c = 'blue', label = 'Iris-setosa',
marker = '.')
plt.scatter(x[y == 2, 0], x[y == 2, 1], s = 200, c = 'green', label = 'Iris-virginica',
marker= '+')
centers = kmeans.cluster_centers_
plt.scatter(centers[:, 0], centers[:, 1], c='black', s=200, alpha=0.8, label = 'Centers',
marker= 'o');
plt.legend()
plt.show()
```



Rysunek 6.4. Po ograniczeniu liczby zmiennych do dwóch wyraźnie widać, jak algorytm *k*-średnich przypisał przykłady do najbliższych klastrów

Jeśli pamiętamy zbiór danych `iris`, to zauważymy, że do pierwszego klastra trafiły kwiaty z gatunku *setosa*. Kwiaty tego gatunku mają wyraźnie mniejsze płatki od pozostałych, spełniają więc warunek rozłączności naturalnych grup. Wszystkie irysy *setosa* mają płatki podobnej wielkości, spełniając warunek regularności kształtu naturalnej grupy. Wielkość grupy irysów *setosa* jest zbliżona do wielkości pozostałych grup, czyli założenie podobnej wielkości naturalnych grup zostało spełnione. W związku z tym, że w zbiorze jest po pięćdziesiąt przykładów kwiatów każdego gatunku, założenie równoliczności naturalnych grup zostało spełnione. Dzięki temu klaster 1 zawiera wszystkie przykłady irysów *setosa* i ani jednego przykładu irysa innego gatunku. Proszę samodzielnie sprawdzić, czy pozostałe dwie grupy też spełniają wszystkie cztery założenia algorytmu grupowania metodą *k*-średnich i co z tego wynikło.

Przekonaliśmy się, że grupowanie metodą *k*-średnich daje dobre wyniki, o ile spełnione są jego założenia. Algorytm ten zawsze znajdzie minimum funkcji kosztu, bo każda zmiana przypisania przykładu do nowego klastra zmniejsza sumę odległości przykładów od ich klastrów. Nie mamy jednak gwarancji, że znalezione minimum będzie minimum globalnym, a nie lokalnym. Rozwiązanie tego problemu polega na właściwym zainicjowaniu modelu metodą *k*-means++ i wygenerowaniu kilku (np. dziesięciu) modeli kandydujących.

Metodą *k*-średnich można grupować tylko według cech numerycznych. Nie znaczy to jednak, że niemożliwe jest ocenienie podobieństwa przykładów na podstawie ich cech kategorycznych. Zacznijmy od zmiennych binarnych (wiemy, że każdą zmienną kategoryczną można zakodować metodą jeden do wielu). Zmienne binarne mogą przyjmować jedną z dwóch wartości – przyjmijmy, że będą to wartości 0 lub 1. Dwa przykłady x_i i x_j są tym bardziej do siebie podobne, im więcej mają wspólnych cech, czyli im częściej zmienne przyjmują tę samą wartość (tab. 6.1).

Tabela 6.1. Macierz podobieństwa zmiennych binarnych

		Przykład x_i		
		1	0	suma
Przykład x_j	1	q	R	$q+r$
	0	s	T	$s+t$
	Suma	$Q + s$	$R + t$	$p = q + r + s + t$
Oznaczenia: q – liczba zmiennych przyjmujących wartość 1 dla obu obiektów, r – liczba zmiennych przyjmujących wartość 1 dla obiektu x_i i 0 dla x_j, s – liczba zmiennych przyjmujących wartość 0 dla obiektu x_i i 1 dla x_j, t – liczba zmiennych przyjmujących wartość 0 dla obu obiektów, p – podobieństwo obiektów $q + r + s + t$.				

Podobieństwo zmiennych kategorycznych definiuje się jako uogólnienie macierzy podobieństwa zmiennych binarnych, przyjmując, że podobne do siebie

są wyłącznie identyczne stany zmiennej, na przykład *niebieski* i *niebieski*, ale nie *niebieski* i *błękitny*. Przy takim założeniu niepodobieństwo między obiektami x_i i x_j można wyrazić wzorem

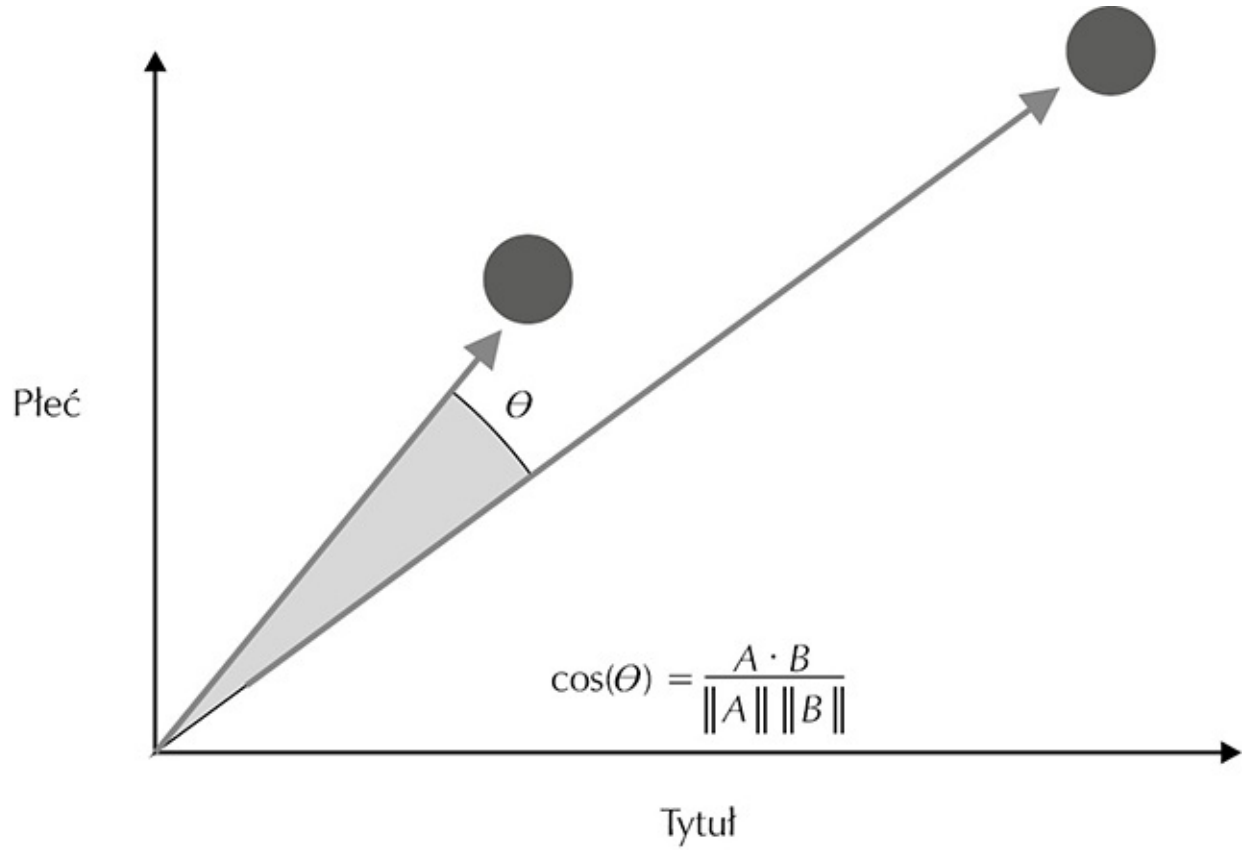
$$(i, j) = \frac{p - m}{p}$$

gdzie p oznacza łączną liczbę zmiennych, a m – liczbę zmiennych, których wartość jest identyczna dla obu przykładów.

Proszę zwrócić uwagę, że takie miary podobieństwa są niewrażliwe na odległości dzielące poszczególne stany zmiennych. To dobrze, bo odległość między tytułem *Kapitan* a *Pani* nie jest ani mniejsza, ani większa od odległości dzielącej tytuły *Panicz* i *Wielebny*. Podobny efekt uzyskamy, jeśli do porównania użyjemy odległości kosinusowej

$$d_{\text{cosine}}(i, j) = \cos(\theta) = \frac{A * B}{\|A\| \|B\|}$$

Miara ta jest powszechnie używana do oceny podobieństwa wyrazów. Podobne wyrazy są reprezentowane przez wektory leżące blisko siebie, czyli kąt między nimi jest mały, a jego kosinus duży, bliski jedynce. Wektory leżące daleko od siebie dzieli duży kąt, a jego kosinus będzie bliski zeru (rys. 6.5).



Rysunek 6.5. Miara kosinusowa jest niewrażliwa na odległość punktów wynikającą z dystansu między stanami zmiennych

Często chcemy pogrupować przykłady zarówno według cech numerycznych, jak i kategorycznych. W takich przypadkach możemy albo zakodować zmienne kategoryczne, albo – i takie rozwiązanie jest częściej stosowane – najpierw przeprowadzić analizę PCA i pogrupować otrzymane składowe główne.

Ostatnią rzeczą wymagającą wyjaśnienia są metody oceny modeli grupujących. Czyli trzeba odpowiedzieć na pytanie, jak został wybrany najlepszy z dziesięciu automatycznie utworzonych modeli kandydyjących. Uwzględnia się dwa kryteria:

- **Podobieństwo obiektów należących do tego samego klastra, czyli zwartość klastrów.** Miarą zwartości jest odchylenie wewnątrzklastrowe – suma odległości wszystkich obiektów klastra od jego środka

$$wc(C) = \Sigma_{j=1}^{k}\ wc(C_j)$$

Jego obliczenie wymaga wykonania *O(k)* operacji.

- **Niepodobieństwo obiektów należących do różnych klastrów, czyli rozłączność klastrów.** Miarą rozłączności jest odchylenie międzyklastrowe – suma odległości środków wszystkich par klastrów

$$bc(C) = \Sigma_{1 \le i \le j \le k} d(r_j, r_j)^2$$

Jego obliczenie wymaga wykonania $O(k^2)$ operacji.

Jakość modelu grupującego można ocenić na podstawie odchylenia wewnątrzklastrowego, odchylenia międzyklastrowego lub ich kombinacji, na przykład jako stosunek obu miar $bc(C)/wc(C)$.

6.3. Segmentacja

Przekonaliśmy się, że analiza skupień pozwala zredukować dane. Grupowanie jest też powszechnie używane w modelach deskryptywnych. Zadaniem modeli deskryptywnych jest wydobycie i opisanie wzorców ukrytych w danych. Natomiast w modelach predykcyjnych analiza skupień jest używana rzadziej, chociaż wciąż stosuje się ją na przykład do analizy i kategoryzowania tekstów (jednak ostatnio do analizy tekstów coraz częściej używa się głębokich sieci neuronowych).

W predykcyjnych zastosowaniach modeli grupowania wykorzystuje się fakt, że dzielenie przykładów na klastry odbywa się na podstawie ukrytej zmiennej reprezentującej (nie)podobieństwo przykładów. Ta zmienna, czyli identyfikator klastra, do którego trafił przykład, jest dodawana do danych i może być następnie używana do oceny nowych obserwacji. Nic dziwnego, że najczęstszym zastosowaniem modeli grupujących jest segmentacja.

Firmy, w szczególności działy marketingu, starają się jak najlepiej poznać

swoich klientów. Dopóki klientów jest mało, powiedzmy kilkuset, firma jest w stanie poznać każdego z nich. Sprzedawca pamięta, co zwykle kupuje pani Ania, a czego nie proponować panu Wojtkowi. Sytuacja zmienia się wraz ze zwiększeniem się liczby klientów. W którymś momencie utrzymanie komunikacji dopasowanej do indywidualnych potrzeb klienta wymaga przeprowadzenia analizy zachowania klientów i podzielenia ich na segmenty.

Jedna z najczęściej używanych metod wartościowania klientów polega na obliczeniu ich wskaźnika RFM (*Recency*, *Frequency*, *Monetary*). Doradcy klienta i marketerzy używają tego wskaźnika nie tylko do określenia wartości klienta, ale również do przewidywania jego reakcji na przyszłe działania marketingowe. Obliczenie wskaźnika RFM wymaga uwzględnienia trzech zmiennych:

- czasu, który upłynął od złożenia przez klienta ostatniego zamówienia; im jest on krótszy, tym większa wartość klienta;
- częstości, z jaką klient dokonuje zakupów; im częściej, tym jego wartość jest większa;
- sumy kwot zrealizowanych zamówień; oczywiście, im jest ona większa, tym większa jest wartość klienta.

Każdej z trzech zmiennych nadaje się odpowiednią wagę. Wagi ustala się na podstawie branży, w której działa dana firma, z uwzględnieniem specyfiki samej firmy. Dalej sytuacja się komplikuje. Chcemy podzielić klientów na podstawie ich trzech cech. Nawet przy tak małej liczbie zmiennych okazuje się, że wynik zależy od tego, w jakiej kolejności będziemy analizować zmienne – jeżeli zaczniemy od częstości zakupów, otrzymamy inny podział na segmenty, niż gdybyśmy zaczęli od pieniędzy wydanych przez klienta.

Najlepiej byłoby jednocześnie uwzględnić wszystkie zmienne. Niestety, ludzie nie są w stanie jednocześnie zestawiać ze sobą więcej niż kilku zmiennych,

a większość z nas najlepiej radzi sobie z jednoczesną analizą zaledwie dwóch zmiennych. Pamiętajmy, że korelacje dwóch zmiennych można łatwo przedstawić na wykresach. W rozwiązaniu problemu jednoczesnego uwzględnienia wielu zmiennych do przypisania klienta do odpowiedniego segmentu pomogą modele analizy skupień. Przecież algorytm grupowania metodą *k*-średnich robi dokładnie to, czego potrzebujemy – porównuje podobieństwo przykładów na podstawie wartości opisujących je cech. Z tą różnicą, że w przeciwieństwie do nas, jednoczesne uwzględnienie wszystkich zmiennych nie jest dla algorytmu grupowania problemem.

Sprawdźmy, jak poradzi sobie algorytm grupowania metodą *k*-średnich z segmentacją klientów. Dane o sprzedaży znajdują się w tabeli `[OnlineSales].[store_sales]`, a dane o zwrotach znajdziemy w tabeli `[OnlineSales].[store_returns]`. Na podstawie danych z obu tych tabel zostały policzone cztery zmienne:

- `orderRatio` – procent całkowicie lub częściowo zwróconych zamówień,
- `itemsRatio` – procent zwróconych produktów,
- `monetaryRatio` – procent zwróconych wpłat,
- `frequency` – liczba transakcji klienta.

```
SELECT
   ss_customer_sk AS customer,
   CAST( (ROUND(COALESCE(returns_count / NULLIF(1.0*orders_count, 0), 0), 7) ) AS FLOAT)
AS orderRatio,
   CAST( (ROUND(COALESCE(returns_items / NULLIF(1.0*orders_items, 0), 0), 7) ) AS FLOAT)
AS itemsRatio,
   CAST( (ROUND(COALESCE(returns_money / NULLIF(1.0*orders_money, 0), 0), 7) ) AS FLOAT)
AS monetaryRatio,
   CAST( (COALESCE(returns_count, 0)) AS FLOAT) AS frequency
FROM …
```

Identyfikatory klientów uzupełnione o cztery powyższe cechy zwraca

`OnlineSales.RFM`:

```
SELECT *
FROM [OnlineSales].[RFM];
```

customer	orderRatio	itemsRatio	monetaryRatio	frequency
29727	0	0	0	0
97643	0.0681818	0.0781759	0.037034	3
57247	0	0	0	0
32549	0.0869565	0.0686567	0.031281	4
…				

Prototyp zbudujemy w edytorze języka Python, a gotowy model zaimplementujemy w serwerze SQL Server. Do zaimportowania danych użyjemy funkcji biblioteki `revoscalepy`, a więc należy się upewnić, czy używamy właściwego, zintegrowanego z usługą SQL Server Machine Learning, interpretatora tego języka.

```
conn_str = 'Driver=SQL Server;Server=MS;Database=ML;Trusted_Connection=True;'
input_query = '''SELECT * FROM [OnlineSales].[RFM]'''

column_info = {
     "customer": {"type": "integer"},
     "orderRatio": {"type": "integer"},
     "itemsRatio": {"type": "integer"},
     "frequency": {"type": "integer"}
     }

data_source = revoscalepy.RxSqlServerData(sql_query=input_query, column_Info=column_info,
connection_string=conn_str)
```

```
revoscalepy.RxInSqlServer(connection_string=conn_str, num_tasks=1, auto_cleanup=False)
Express Edition will continue to be enforced.
```

Zwrócony komunikat informuje, że biblioteki klienckie licencjonowane są tak samo, jak edycja Express serwera SQL, czyli że dostępne są za darmo, ale ich wydajność i skalowalność zostały ograniczone.

Zaimportujmy przygotowane dane i przyjrzyjmy się im.

```
customer_data = pd.DataFrame(revoscalepy.rx_import(data_source))
print("Data frame:", customer_data.head(n=5))
print(customer_data.describe())
```

```
 Rows Read: 37336, Total Rows Processed: 37336, Total Chunk Time: 0.200 seconds
Data frame:     customer   orderRatio   itemsRatio   monetaryRatio   frequency
0    29727.0     0.000000     0.000000         0.000000         0.0
1    97643.0     0.068182     0.078176         0.037034         3.0
2    57247.0     0.000000     0.000000         0.000000         0.0
3    32549.0     0.086957     0.068657         0.031281         4.0
4     2040.0     0.000000     0.000000         0.000000         0.0
            customer      orderRatio        itemsRatio        monetaryRatio     frequency
count   37336.000000    37336.000000      37336.000000         37336.000000   37336.000000
mean    50876.778980        0.098900          0.062042             0.044831       0.223484
std     28720.140309        0.278889          0.232386             0.695286       0.697261
min         2.000000        0.000000          0.000000             0.000000       0.000000
25%     25805.750000        0.000000          0.000000             0.000000       0.000000
50%     52157.500000        0.000000          0.000000             0.000000       0.000000
75%     75922.250000        0.000000          0.000000             0.000000       0.000000
max     98999.000000        2.000000         11.000000           108.719154      10.000000
```

Wszystkie cztery zmienne (ocena numerów klientów nie ma sensu) są silnie prawoskośne. Ich wartości średnie są większe nie tylko od mediany, ale nawet od trzeciego kwartylu. Widzimy, że przynajmniej 75% wartości wszystkich czterech zmiennych to zera. Oznacza to, że przynajmniej ¾ klientów nie zwróciło żadnego towaru (to dobrze), ale również, że ponad ¾ klientów tylko raz odwiedziło nasz sklep (to źle). Warte uwagi są też wysokie wartości maksymalne zmiennych `itemsRatio` i `monetaryRatio`. Jakiś klient zwrócił 11 razy więcej produktów niż kupił, a przynajmniej jednemu klientowi zwrócono ponad 100 razy większą kwotę niż wyniosła suma jego wydatków.

Zanim zbudujemy model, musimy ustalić liczbę segmentów (klastrów), na które chcemy podzielić klientów. Po wyjaśnieniu kwestii wyboru miary podobieństwa przykładów i metody inicjalizacji modelu, ostatnim problemem który pozostał do rozwiązania, jest właśnie wybór optymalnej liczby klastrów.

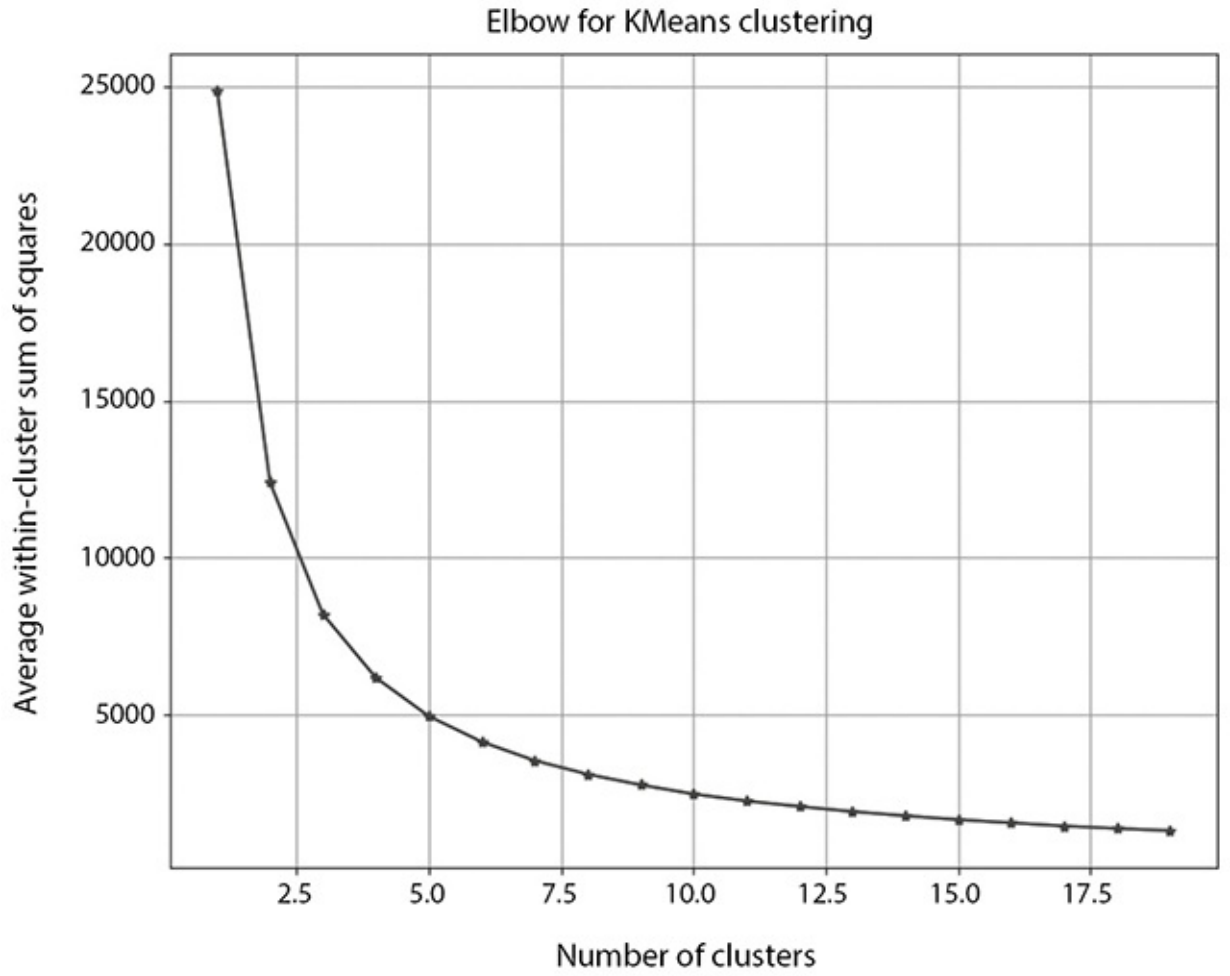
Często właściwa liczba klastrów wynika z przyjętego modelu biznesowego.

Jeżeli firma ma na przykład cztery strategie sprzedaży, adresowane do czterech segmentów klientów, będzie chciała, żeby model podzielił jej klientów na cztery segmenty. Ta firma nie będzie w żaden sposób zainteresowana modelem zwracającym 15 klastrów, niezależnie od jego jakości, bo nie chce, albo nie może, zmienić swojej, opartej na czterech segmentach klientów strategii sprzedaży.

Czasami będziemy musieli jednak samodzielnie znaleźć optymalną liczbę klastrów. Do jej znalezienia posłużymy się kryterium odchylenia wewnątrzklastrowego. Wytrenujemy *k* modeli kandydujących, każdy dla innej liczby klastrów, i wybierzemy najlepszy. Nie możemy jednak po prostu wybrać modelu o najmniejszym odchyleniu wewnętrzklastrowym, bo ta miara maleje wraz ze zwiększaniem się liczby klastrów i osiąga minimum dla liczby klastrów równej liczbie przykładów. Wtedy każdy klaster zawiera jeden przykład, a więc średnia odległość przykładów przypisanych do klastra od jego środka jest najmniejsza. Zamiast tego użyjemy metody łokcia. **Sprawdzimy na wykresie, jak zmniejsza się wartość wraz ze zwiększaniem się liczby klastrów, i wybierzemy liczbę klastrów, dla której spadek wartości zaczął maleć** (rys. 6.6).

```
cdata = customer_data
K = range(1, 20)
KM = (sk_cluster.KMeans(n_clusters=k).fit(cdata) for k in K)
centroids = (k.cluster_centers_ for k in km)

D_k = (sci_distance.cdist(cdata, cent, 'euclidean') for cent in centroids)
dist = (np.min(D, axis=1) for D in D_k)
avgWithinSS = [sum(d) / cdata.shape[0] for d in dist]
plt.plot(K, avgWithinSS, 'b*-')
plt.grid(True)
plt.xlabel('Number of clusters')
plt.ylabel('Average within-cluster sum of squares')
plt.title('Elbow for kmeans clustering')
plt.show()
```



Rysunek 6.6. Charakterystyczny łokieć wskazuje, że odchylenie wewnętrzklastrowe maleje szybko wraz z dodaniem drugiego, trzeciego i czwartego klastra. Dodawanie kolejnych klastrów ma coraz mniejszy wpływ na ich zwartość

Na podstawie wykresu 6.6 decydujemy się na cztery klastry. Segmentację przeprowadzimy na podstawie czterech zmiennych i uzupełnimy oryginalne dane o numer klastra, do którego trafił dany klient.

```
n_clusters = 4
means_cluster = sk_cluster.KMeans(n_clusters=n_clusters, random_state=111)
columns = ["orderRatio", "itemsRatio", "monetaryRatio", "frequency"]
est = means_cluster.fit(customer_data[columns])
clusters = est.labels_
customer_data['cluster'] = clusters
```

Popatrzmy na otrzymane wyniki.

```
# For each cluster, count the members.
for c in range(n_clusters):
     cluster_members=customer_data[customer_data['cluster'] == c][:]
     print('Cluster {}(n={})'.format(c, len(cluster_members)))

# Print mean values per cluster.
print(customer_data.groupby(['cluster']).mean())
```

```
Cluster 0(n=31675)
Cluster 1(n=4989)
Cluster 2(n=1)
Cluster 3(n=671)
                customer   orderRatio   itemsRatio   monetaryRatio   frequency
cluster
0          50854.809882     0.000000     0.000000        0.000000    0.000000
1          51332.535779     0.721604     0.453365        0.307721    1.097815
2          57044.000000     1.000000     2.000000      108.719154    1.000000
3          48516.023845     0.136277     0.078346        0.044497    4.271237
```

Prawie 85% klientów trafiło do pierwszego klastra (klastra o numerze 0). To typowi klienci, którzy raz coś kupili w sklepie i już do niego nie wrócili – ani po to, żeby zwrócić towar, ani żeby kupić coś innego. Drugi klaster to klienci, którzy trzykrotnie częściej niż pozostali zwracali kupione produkty – stanową oni 13% wszystkich klientów. Do czwartego klastra trafiły osoby, które są naszymi stałymi klientami i które rzadko zwracają kupione towary. Te dwa procenty (klastra o numerze 3) to najbardziej wartościowi klienci firmy.

Bardzo ciekawy jest trzeci klaster. Trafił do niego tylko jeden klient, wyraźnie różniący się od pozostałych. **Przykłady znacząco różne od pozostałych nazywa się anomaliami.** W tym przypadku różnica sprowadza się do odstającej wartości zmiennej `monetaryRatio`, ale jest to tylko szczególny przykład ogólnej metody wykrywania anomalii. Wykrywanie anomalii polega na sprawdzaniu, czy kombinacja wartości zmiennych pasuje do któregoś z klastrów, przy czym jednocześnie analizowane są (tak samo jak podczas tworzenia klastrów) odległości między wszystkimi zmiennymi. Anomalią jest obserwacja o nietypowej kombinacji wartości zmiennych, przy czym wartości te wcale nie

muszą być odstające. Na przykład, nie ma nic niezwykłego w byciu matką albo nastolatkiem, ale nastolatek będący matką jest rzadkim przypadkiem. Za anomalie uznaje się przykłady, które trafiły do najmniejszych klastrów. **Gdybyśmy chcieli wykryć więcej anomalii, należałoby zwiększyć liczbę klastrów.** Proszę samodzielnie sprawdzić wyniki dla 5 i więcej klastrów.

Zbudowany model spełnia nasze wymagania i możemy go udostępnić użytkownikom. Trening modeli umożliwia procedura `[OnlineSales].[CustomerSegmentation]`. Jej działanie sprowadza się do:

- odczytania przygotowanych danych z widoku `[OnlineSales].[RFM]`;
- wywołania skryptu w języku Python z użyciem procedury systemowej `sp_execute_external_script`;
- zwrócenia danych w postaci tabeli o sześciu kolumnach: `Customer`, `orderRatio`, `itemsRatio`, `monetaryRatio`, `frequency` i `cluster`.

Użyty w tej procedurze zewnętrzny skrypt:

- z biblioteki `sklearn` importuje funkcję `kmeans`;
- do zmiennej `customer_data` przypisuje dane przekazane przez parametr `@input_data_1` procedury `sp_execute_external_script`;
- wartość zmiennej `n_clusters` ustawia na 4;
- grupuje dane metodą *k*-średnich;
- przekazane dane uzupełnia o numery klastrów, do których zostali przypisani klienci, i tak uzupełnione dane zwraca serwerowi SQL Server.

```
CREATE OR ALTER PROCEDURE [OnlineSales].[CustomerSegmentation]
AS
BEGIN
DECLARE @input_query NVARCHAR(MAX) = N'SELECT * FROM [OnlineSales].[RFM]'

EXEC sp_execute_external_script
          @language = N'Python'
        , @script = N'
from sklearn.cluster import kmeans

customer_data = my_input_data

n_clusters = 4

est = kmeans(n_clusters=n_clusters, random_state=111).
fit(customer_data[["orderRatio","itemsRatio","monetaryRatio","frequency"]])
clusters = est.labels_
customer_data["cluster"] = clusters

OutputDataSet = customer_data
'
        , @input_data_1 = @input_query
        , @input_data_1_name = N'my_input_data'
          WITH RESULT SETS (("Customer" int, "orderRatio" float,"itemsRatio" float,
"monetaryRatio" float,"frequency" float,"cluster" float));
END;
```

Modyfikację procedury `[OnlineSales].[CustomerSegmentation]` pozwalającą podać jako parametr jej wywołania liczbę klastrów pozostawiam Państwu jako ćwiczenie do samodzielnego wykonania.

Skoro procedura zwraca tabelę, to można ją wywołać w ramach instrukcji `INSERT INTO`.

```
INSERT INTO OnlineSales.[py_customer_clusters]
EXEC [OnlineSales].[CustomerSegmentation];
```

```
(37336 rows affected)
```

W ten sposób do wskazanej tabeli zostały wstawione dane zwrócone przez

procedurę. Ponieważ jedną z kolumn jest numer klienta, tabelę zawierającą numery segmentów (klastrów) możemy złączyć z tabelą zawierającą dane klientów i w ten sposób przygotować dane adresowe do najbliższej kampanii marketingowej.

```
SELECT c.[c_email_address], c.c_first_name, c.c_salutation
FROM OnlineSales.customer AS c
JOIN OnlineSales.[py_customer_clusters] AS cl
ON cl.Customer = c.c_customer_sk
WHERE cl.cluster = 3;
```

```
c_email_address                                          c_first_name
c_salutation
Brian.Smith@mail2world.com                               Brian
Dr.
Jennifer.Rodriquez@shtrudel.net                          Jennifer
Ms.
Robert.Barrett@rediffmail.com                            Robert
Dr.
Jose.Brown@peacemail.com                                 Jose
Dr.
Louisa.Waggoner@born2frag.de                             Louisa
Miss
…
```

Możemy też odczytać informacje na temat wykrytych, nietypowych klientów.

```
SELECT c.c_first_name, c.c_last_name,  c.c_preferred_cust_flag
FROM OnlineSales.customer AS c
JOIN OnlineSales.[py_customer_clusters] AS cl
ON cl.Customer = c.c_customer_sk
WHERE cl.cluster = 2;
```

```
c_first_name            c_last_name                      c_preferred_cust_flag
Bonnie                  Rosenthal                        Y
```

W tym rozdziale poznaliśmy najpopularniejszy algorytm nienadzorowanego uczenia maszynowego i zastosowaliśmy go do wytypowania spośród wszystkich klientów firmy, tych którzy z największym prawdopodobieństwem odpowiedzą

na akcję marketingową. Ponadto przeprowadziliśmy analizę anomalii i wykryliśmy nietypowych klientów.

Nie wszystkie linie są proste.

Hans Rosling, dyrektor fundacji Gapminder, autor książki *Factfulness*

Rozdział 7
Regresja

W tym rozdziale zajmiemy się regresją rozumianą jako zbiór metod wyszukiwania korelacji ukrytych w danych, w celu jak najdokładniejszego oszacowania wartości zmiennej wyjściowej na podstawie zmiennych wejściowych. Tak definiowana regresja jest techniką uczenia nadzorowanego realizowaną za pomocą różnych algorytmów, takich jak regresja wieloraka, ogólny model liniowy (GLM), sztuczne sieci neuronowe i drzewa regresyjne.

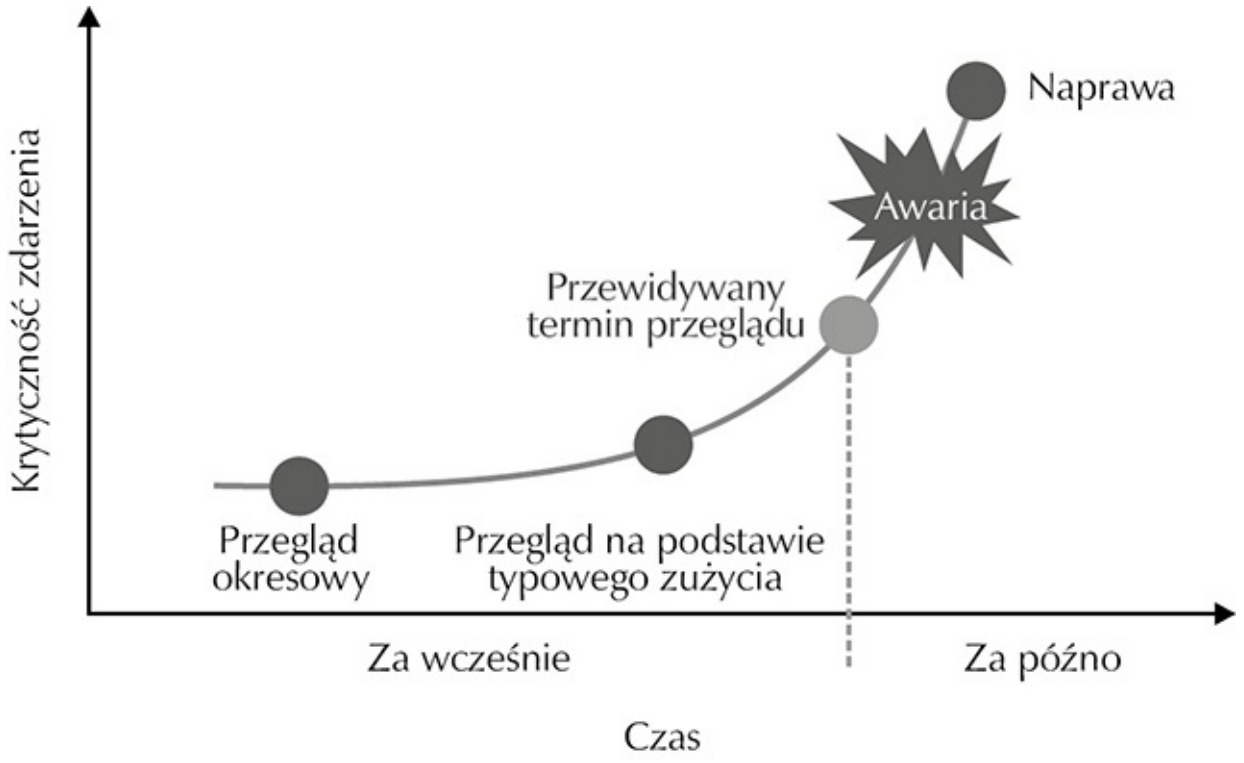
Budując model proaktywnej konserwacji urządzeń, nauczymy się przygotowywać dane z użyciem funkcji biblioteki `RevoScaleR`, przedstawimy trzy pierwsze algorytmy regresji i ich implementacje w bibliotece `MicrosoftML`, przyjrzymy się różnym metodom zapobiegania nadmiernemu dopasowaniu modeli oraz poznamy popularne funkcje kosztu i metody optymalizacji procesu uczenia. Czwarty z wymienionych algorytmów, drzewa regresji, został przedstawiony w następnym rozdziale. Tak jak poprzednio, modele zbudujemy zgodnie z metodyką CRISP-DM.

7.1.
Zrozumienie problemu

Do prawidłowego funkcjonowania firmy potrzebują sprawnych urządzeń. Działanie niektórych urządzeń ma krytyczne znaczenie. Na przykład sklep nie

sprzeda żadnego towaru, jeżeli awarii ulegnie kasa fiskalna, w kawiarni z uszkodzonym ekspresem nie kupimy kawy, a samolot nie wystartuje, jeżeli jego silnik nie będzie w 100% sprawny.

Większość firm naprawia urządzenia w momencie, gdy przestają one już działać. Strategia **konserwacji korygującej** zapewnia maksymalne wykorzystanie podzespołów urządzeń, bo są one wymieniane dopiero po całkowitym zużyciu. Jednocześnie taka strategia naraża firmy na kosztowne przestoje i wysokie koszty niespodziewanych napraw. Dlatego niektóre firmy stosują strategię **konserwacji zapobiegawczej**. Polega ona na przeprowadzaniu z góry zaplanowanych przeglądów urządzeń. Przeglądy mogą być okresowe albo realizowane na podstawie typowego zużycia urządzeń, na przykład przebiegu samochodu. Ta strategia pozwala uniknąć kosztownych przestojów i zmniejsza koszty napraw, bo ustalenie z wyprzedzeniem momentu naprawy daje czas potrzebny do sprowadzenia potrzebnych części i fachowców oraz urządzenia zapasowego, które zastąpi to naprawiane. Firma jednak ponosi wysokie koszty nie zawsze potrzebnych przeglądów i niepełnego wykorzystania podzespołów. **Proaktywna konserwacja** łączy zalety obu powyższych strategii, ale jest pozbawiona ich wad. Polega ona na przewidywaniu, kiedy urządzenie ulegnie awarii, i wymianie tylko tych podzespołów, których poprawne działanie nie jest już pewne (rys. 7.1).

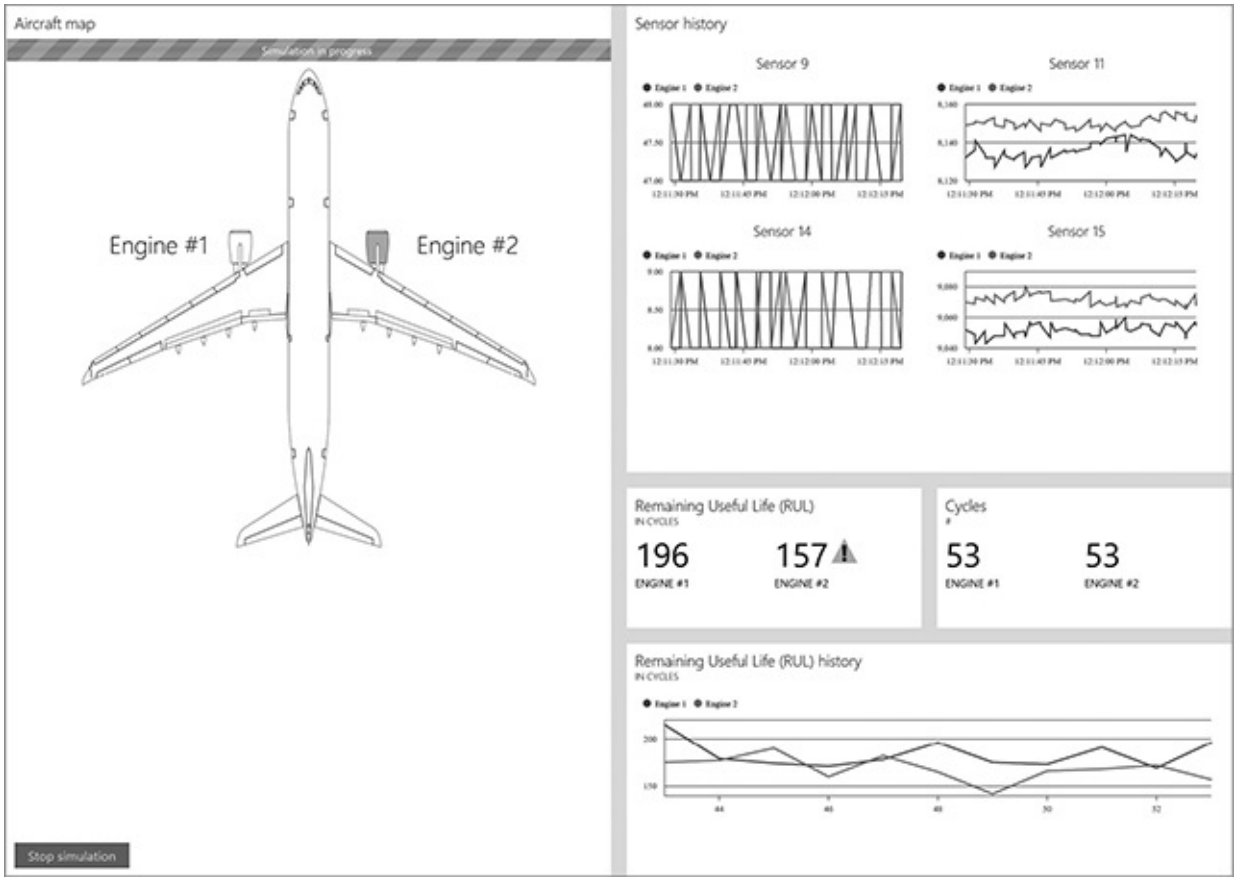


Rysunek 7.1. Strategia konserwacji korygującej oznacza spóźnione reakcje, strategia konserwacji zapobiegawczej – za wczesne reakcje na awarie. Strategia proaktywnej konserwacji minimalizuje koszty napraw i jednocześnie ogranicza do minimum ryzyko wystąpienia awarii

Przyjrzyjmy się linii lotniczej Oceanic Airlines. Jednym z powodów opóźniania lotów są problemy przewoźnika z serwisowaniem silników samolotów. Ponieważ w firmie obowiązuje zasada zapobiegania awariom silników za wszelką cenę, ich stan techniczny jest regularnie sprawdzany. Jednak zużycie elementów silnika samolotowego zależy od wielu czynników, nie tylko od czasu jego pracy i odległości przelotów. W efekcie niektóre czynności serwisowe są przeprowadzane niepotrzebnie, a zdarza się, że przeciągający się przegląd powoduje opóźnienie odlotu.

Wszystkie samoloty Oceanic Airlines są wyposażone w czujniki monitorujące pracę urządzeń pokładowych, w tym silników. Firma ma przykładowe dane zarówno bezbłędnie działających silników, jak i sinków, które uległy awarii. Naszym zadaniem jest zbudowanie modelu, który przewidzi czas pozostałej bezpiecznej eksploatacji *RUL* silników. Zbudowany model zostanie użyty do

wdrożenia strategii proaktywnej konserwacji (rys. 7.2).



Rysunek 7.2. Wiedząc, że silnik wymaga przeglądu, firma może z odpowiednim wyprzedzeniem zaplanować niezbędne czynności konserwacyjne i w ten sposób skrócić czas pobytu samolotu na lotnisku

7.2. Zrozumienie danych

Pierwszym etapem projektu uczenia maszynowego jest zrozumienie problemu i sprecyzowanie celów. Drugi etap obejmuje zebranie i ocenę danych.

Dane treningowe są zapisane w pliku *PM_train.csv*[13]. Zawiera on opis działania 100 silników tego samego typu. Praca każdego silnika jest przedstawiona w postaci 24 szeregów czasowych. Kluczem tych serii jest numer cyklu, a wartościami bieżące parametry pracy silnika i wyniki pomiarów jego działania.

Chociaż dla wszystkich 100 silników pierwszym numerem cyklu jest liczba 1, ich pomiary zaczęły się w różnym czasie. Czyli pierwszy pomiar został przeprowadzony dla silników o różnym, nieznanym nam stopniu zużycia. Wiemy natomiast, że na początku każdego cyklu wszystkie silniki działały prawidłowo oraz to, że z czasem ich działanie ulegało pogorszeniu. Po przekroczeniu określonej wartości granicznej parametrów

pracy silnik został uznany za wymagający przeglądu. Ostatni odnotowany pomiar należy uznać za moment, w którym silnik uległ awarii. Dla każdego silnika moment jego awarii może być inny, czyli długość opisujących różne silniki szeregów czasowych jest różna. Plik *PM_train.csv* zawiera 20 631 wierszy i 26 kolumn.

Dane treningowe załadujemy do tabeli `PredictiveMaintenance.PM_Train`, używając do tego funkcji biblioteki `RevoScaleR`. Ponieważ RStudio jest uruchomione na tym samym komputerze co docelowy serwer SQL, instrukcje języka R będą wykonywane lokalnie, a nie po stronie serwera SQL. Zmienimy natomiast sposób ich wykonywania z jednowątkowego na wielowątkowy.

```
connection_string <- "Driver=SQL Server; Server=MS; Database=ML; Trusted_Connection=yes"
local <- RxLocalParallel()
rxSetComputeContext(local)
```

Tekstowe źródło danych utworzymy za pomocą funkcji `RxTextData`. Jedynym obowiązkowym parametrem tej funkcji jest parametr `file` wskazujący na źródłowy plik z danymi. My dodatkowo utworzymy listę z nazwami i typami kolumn i przekażemy ją jako wartość parametru `colClasses`.

```
train_file <- "PM_Train.csv"
train_columns <- c(id = "numeric",
                   cycle = "numeric",
                   setting1 = "numeric",
                   setting2 = "numeric",
                   setting3 = "numeric",
                   s1 = "numeric",
                  …
                   s21 = "numeric")
train_data_text <- RxTextData(file = train_file, colClasses = train_columns)
```

Utworzona zmienna `train_data_text` jest obiektem klasy `RxTextData`. Chociaż możemy się do niej odwoływać, tak jakby była obiektem typu `data frame`, to zawartość pliku *PM_Train.csv* nie została jeszcze wczytana do pamięci. **Dane te będą wczytywane kawałkami na potrzeby wykonywania poszczególnych operacji**. Czyli tak zdefiniowane źródło danych może zawierać dowolnie wiele danych. W szczególności pozwala przetwarzać dane o rozmiarze przekraczającym ilość dostępnej pamięci RAM.

Docelową tabelę będzie reprezentowało źródło danych typu SQL. Tworząca takie źródło danych funkcja `RxSqlServerData` wymaga określenia nazwy tabeli (parametr `table`) lub zapytania SQL (parametr `sqlQuery`) i wskazania serwera SQL Server (parametr `connectionString`). Również tym razem jawnie określimy nazwy i typy kolumn, ustawiając parametr `colClasses` na nazwę zmiennej `train_columns`.

```
train_table_name <- "PredictiveMaintenance.PM_Train"
train_data_table <- RxSqlServerData(table = train_table_name,
                                    connectionString = connection_string,
                                    colClasses = train_columns)
```

Do załadowania danych użyjemy funkcji `rxDataStep`. Ta, jedna z najczęściej używanych funkcji biblioteki `RevoScaleR`, pozwala odczytywać, przekształcać i zapisywać dane. Załadowanie danych ze źródła tekstowego do tabeli i nadpisanie tej tabeli, jeżeli już istniała, sprowadza się do wywołania funkcji

z trzema parametrami: obowiązkowym parametrem `inData`, opcjonalnymi `outFile` oraz `overwrite`. Pierwszy z nich wskazuje docelową tabelę, ustawienie drugiego spowoduje zastąpienie istniejącej tabeli.

```
rxDataStep(inData = train_data_text,
           outFile = train_data_table,
           overwrite = TRUE)
```

Plik z danymi testowymi *PM_test.csv* ma taką samą strukturę. Zawiera on dane o prawidłowej pracy tych samych stu silników. Długość szeregów czasowych opisujących różne silniki też jest różna, ale ostatni pomiar nie oznacza awarii. Dane te zostały zaimportowane w taki sam sposób co dane treningowe, do tabeli `PredictiveMaintenance.PM_Test`. Plik *PM_ test.csv* zawiera 13 096 wierszy i 26 kolumn.

Liczba cykli, przez jakie każdy ze stu silników działał prawidłowo, po zakończeniu testów została zapisana w pliku *PM_truth.csv*. Te dane zostały zaimportowane do tabeli `PredictiveMaintenance.PM_Truth`. Plik *PM_ truth.csv* zawiera 100 wierszy i 1 kolumnę.

Przyjrzyjmy się danym treningowym. Ponieważ jest ich na tyle mało, że można je wszystkie naraz wczytać do pamięci RAM, zaimportujemy zawartość źródła danych SQL do obiektu typu *data frame*. W ten sposób kolejne operacje na danych nie będą wymagały ponownego odczytywania ich z serwera SQL.

```
train_table <- rxImport(train_data_table)
```

Podstawowe informacje o źródłach danych biblioteki `RevoScaleR` oraz o obiektach typu *data frame* zwraca funkcja `rxGetInfo`. Jej jedynym obowiązkowym parametrem jest nazwa źródła danych. Opcjonalny parametr `getVarInfo` pozwala odczytać nazwy, typy oraz najmniejsze i największe wartości zmiennych, a parametr `numRows` pozwala odczytać wskazaną liczbę

wierszy.

```
rxGetInfo(data = train_table, getVarInfo = TRUE, numRows = 1)
```

```
Data frame: train_table
Number of observations: 20631
Number of variables: 26
Variable information:
Var 1: id, Type: numeric, Low/High: (1.0000, 100.0000)
Var 2: cycle, Type: numeric, Low/High: (1.0000, 362.0000)
Var 3: setting1, Type: numeric, Low/High: (-0.0087, 0.0087)
Var 4: setting2, Type: numeric, Low/High: (-0.0006, 0.0006)
Var 5: setting3, Type: numeric, Low/High: (100.0000, 100.0000)
Var 6: s1, Type: numeric, Low/High: (518.6700, 518.6700)
…
Data (5 rows starting with row 1):
  id cycle setting1 setting2 setting3      s1      s2       s3       s4     s5     s6
s7       s8       s9 s10    s11     s12      s13      s14     s15  s16
1  1      1   -0.0007    -4e-04       100 518.67 641.82 1589.70 1400.60 14.62 21.61 554.36
2388.06 9046.19 1.3 47.47 521.66 2388.02 8138.62 8.4195 0.03
```

Statystki zmiennych numerycznych zwraca funkcja `rxSummary`. Jej pierwszym parametrem jest formuła określająca zmienne, dla których zostaną obliczone statystyki. Fomuła ~ . oznacza wszystkie zmienne numeryczne. Jeżeli po prawej stronie znaku tyldy umieścimy nazwę zmiennej, statystyki zostaną policzone tylko dla tej zmiennej. Drugi obowiązkowy parametr `data` wskazuje źródło danych. Zwracane wyniki można ograniczyć do statystyk wskazanych parametrem `summaryStats`.

```
rxSummary(formula =  ~ ., data = train_table, summaryStats = c("Mean", "StdDev", "Min",
"Max", "MissingObs"))
```

```
Summary Statistics Results for: ~.
Data: train_table
Number of valid observations: 20631

 Name         Mean           StdDev       Min        Max       MissingObs
 id             5.150657e+01 2.922763e+01     1.0000 100.0000 0
 cycle          1.088079e+02 6.888099e+01     1.0000 362.0000 0
 setting1      -8.870147e-06 2.187313e-03    -0.0087   0.0087 0
 setting2       2.350831e-06 2.930621e-04    -0.0006   0.0006 0
 setting3       1.000000e+02 0.000000e+00   100.0000 100.0000 0
 s1             5.186700e+02 0.000000e+00   518.6700 518.6700 0
…
```

```
rxSummary(formula = ~ cycle, data = train_table)
Summary Statistics Results for: ~cycle
Data: train_table
Number of valid observations: 20631

 Name   Mean      StdDev    Min Max ValidObs MissingObs
 cycle 108.8079   68.88099 1    362 20631    0
```

Jeżeli po lewej stronie tyldy umieścimy nazwę zmiennej numerycznej, a po prawej nazwę zmiennej kategorycznej, funkcja zwróci statystyki wyliczone dla zmiennej numerycznej względem poszczególnych stanów zmiennej kategorycznej. Specjalna funkcja `F()` pozwala poddać automatycznej dyskretyzacji zmienne numeryczne. Na przykład poniższe wywołanie zwróci najmniejszą i największą wartość zmiennej `cycle` dla wszystkich stanów zmiennej `id`.

```
rxSummary(formula = cycle ~ F(id), data = train_table, summaryStats = c("Min", "Max"))
```

```
Summary Statistics Results for: cycle ~ F(id)
Data: train_table
Number of valid observations: 20631

 Name         Min Max
 cycle:F_id   1   362

Statistics by category (100 categories):

 Category             F_id  Min  Max
 cycle for F(id)=1    1     1    192
 cycle for F(id)=2    2     1    287
 cycle for F(id)=3    3     1    179
…
```

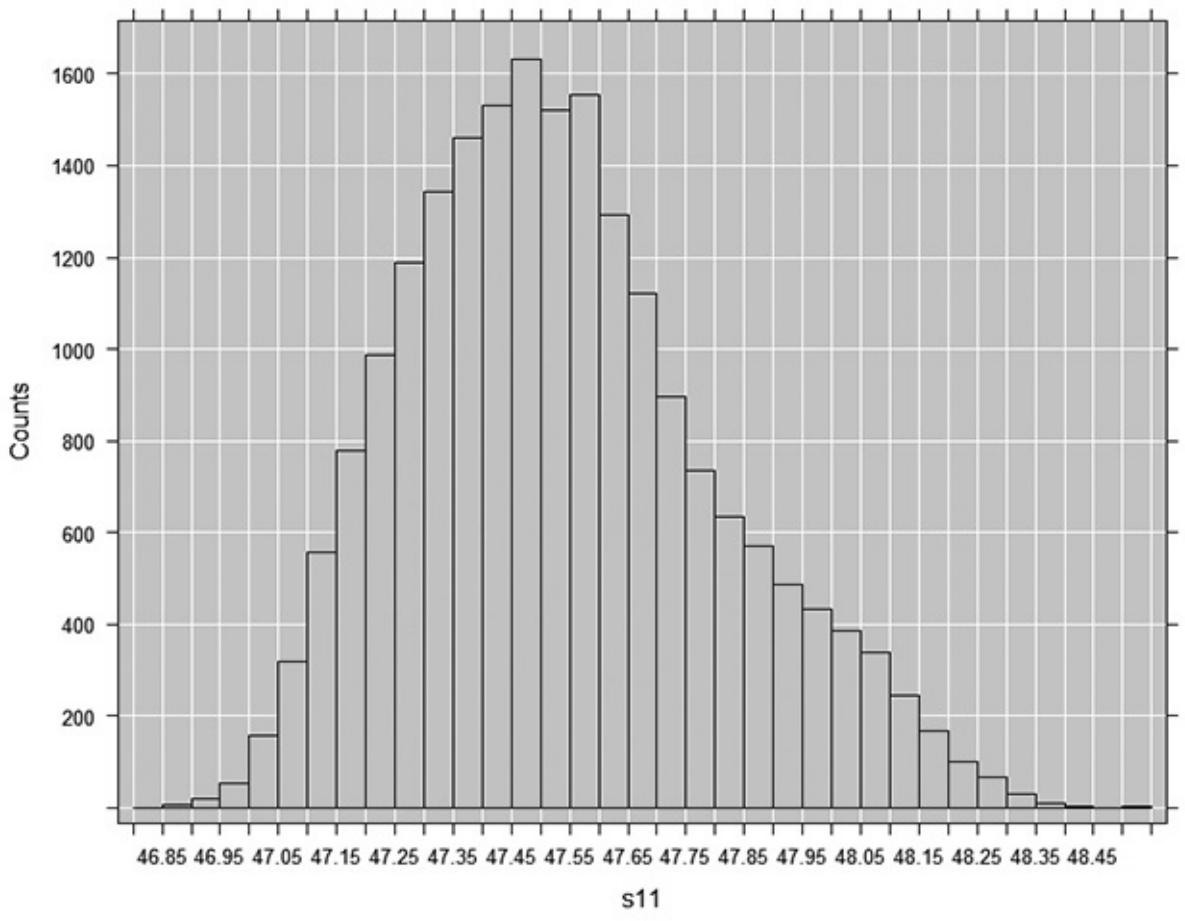
Biblioteka `RevoScaleR` zawiera też dwie funkcje wizualizacji danych: `rxHistogram` i `rxLinePlot`.

Rozkład wartości zmiennych możemy przeanalizować za pomocą funkcji `rxHistogram`. Jej obowiązkowymi parametrami są `formula` i `data`. Na rysunku 7.3 przedstawiono rozkład zmiennej `s11`.

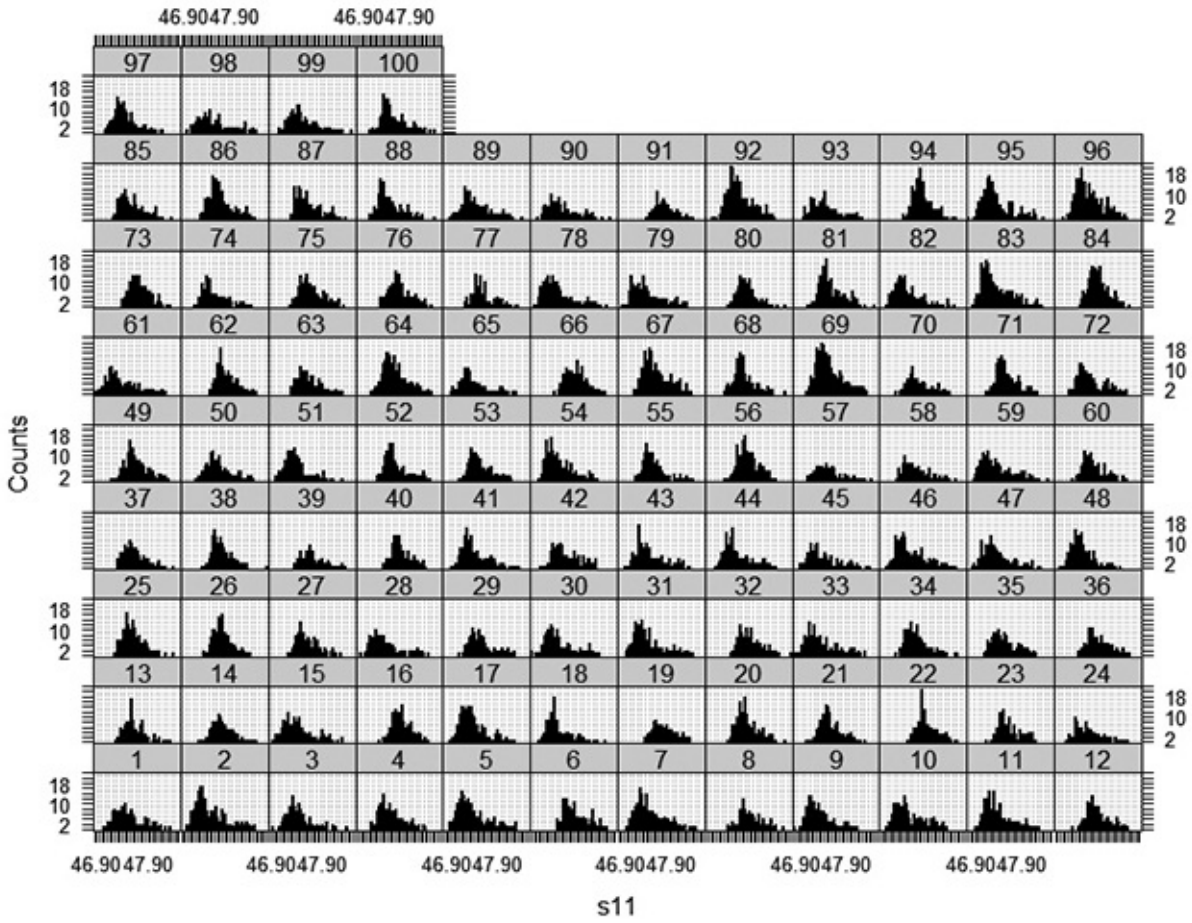
```
rxHistogram(formula = ~s11, data = train_table, numBreaks = 50)
```

Wiedząc, że zbiór treningowy zawiera dane opisujące pracę stu silników, możemy woleć przyjrzeć się rozkładom zmiennych dla poszczególnych wartości zmiennej `id`. Aby wygenerować tyle wykresów, ile stanów przyjmuje zmienna kategoryczna, należy tę zmienną umieścić po prawej stronie kreski pionowej `|` (rys. 7.4).

```
rxHistogram(formula = ~ s11 | F(id), data = train_tab)
```



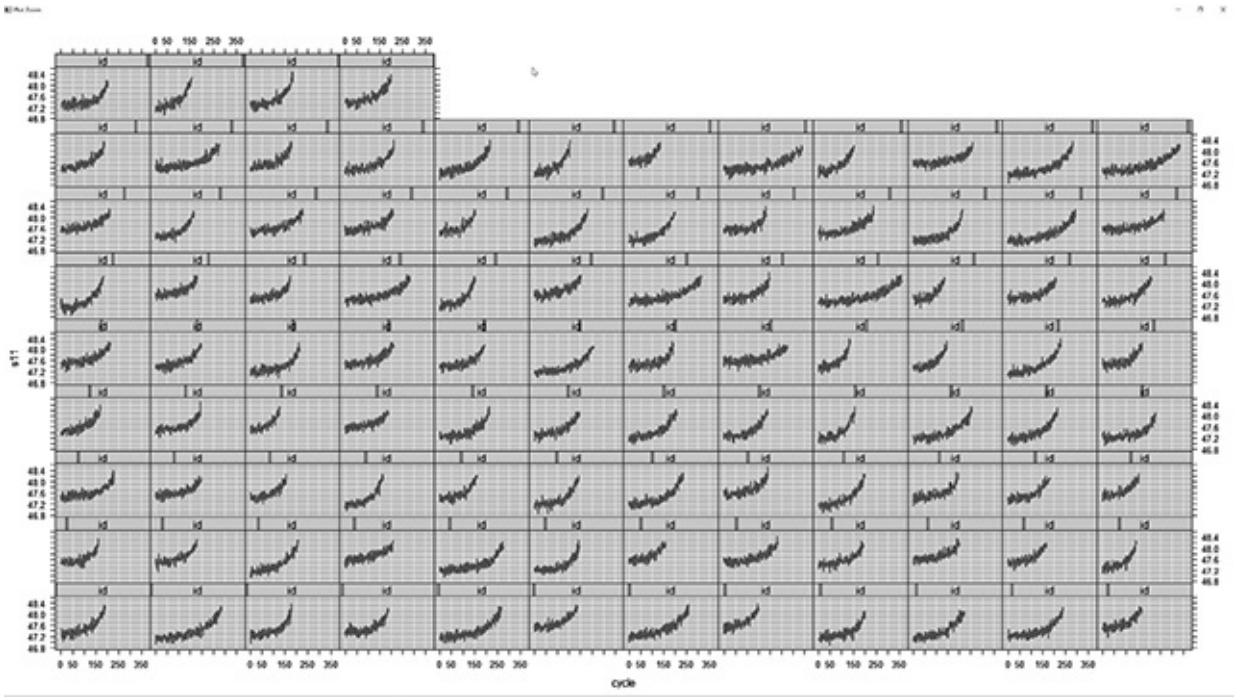
Rysunek 7.3. Rozkład wartości zmiennej s11 podzielonej na 50 przedziałów stałej szerokości



Rysunek 7.4. Rozkład wartości zmiennej s11 dla różnych stanów zmiennej id

Funkcja `rxLinePlot` zwraca wykres liniowy lub punktowy. Ponieważ analizujemy szeregi czasowe, możemy chcieć zobaczyć, jak zmieniały się wartości zmiennej `s11` w kolejnych cyklach. Wymaga to użycia formuły `s11~cycle`. Wiemy też, że dane zawierają szeregi czasowe opisujące pracę stu różnych silników. To, jak zmieniały się wartości zmiennej w czasie dla poszczególnych wartości zmiennej `id` możemy zobaczyć, wykonując poniższą instrukcję (rys. 7.5).

```
rxLinePlot(formula = s11~cycle|id, data = train_table)
```



Rysunek 7.5. Przebiegi zmiennej s11 w czasie wyznaczonym wartościami zmiennej cycle dla poszczególnych wartości zmiennej id

7.3. Opisanie danych

Naszym celem jest sporządzenie prognoz. Na podstawie danych opisujących pracę silnika mamy przewidzieć, kiedy ulegnie on awarii. Teraz, gdy poznaliśmy dane, możemy sprecyzować problem. Mamy zbudować model regresji, który oszacuje, przez ile jeszcze cykli może być eksploatowany silnik, zanim będzie wymagał przeglądu.

Regresja jest techniką uczenia nadzorowanego, a więc każdy przykład musi zostać opisany. Przygotowując dane, musimy dodać do nich zmienną wyjściową `RUL`. W zbiorze treningowym wyliczymy ją na podstawie ostatniego (największego) numeru cyklu. Natomiast w zbiorze testowym do obliczenia pozostałych cykli życia silnika użyjemy danych z tabeli `PredictiveMaintenance.PM_Truth`. Obie operacje przeprowadzimy za pomocą

funkcji `data_label`.

```
data_label <- function(data) {
  data <- as.data.frame(data)
  max_cycle <- plyr::ddply(data, "id", plyr::summarise, max = max(cycle))
  if (!is.null(truth)) {
    max_cycle <- plyr::join(max_cycle, truth, by = "id")
    max_cycle$max <- max_cycle$max + max_cycle$RUL
    max_cycle$RUL <- NULL
  }
  data <- plyr::join(data, max_cycle, by = "id")
  data$RUL <- data$max - data$cycle
  data$max <- NULL
  return(data)
}
```

Dane opiszemy w podobny sposób, w jaki je zaimportowaliśmy. Zdefiniujemy dwa źródła danych typu `RxSqlServerData`: jedno wskazujące na tabelę źródłową `PredictiveMaintenance.PM_Train`, drugie na tabelę docelową `PredictiveMaintenance.Train_Labels`. Jednak tym razem przekształcimy dane przenoszone między tymi tabelami. Funkcja `rxDataStep` pozwala wywołać dowolną funkcję przekazaną parametrem `transformFunc`. Żeby zapewnić powtarzalność wyników w ramach różnych sesji języka R, wszystkie obiekty, do których odwołuje się wywoływana funkcja, muszą zostać jawnie wymienione jako wartości parametru `transformFunc`.

```
  rxDataStep(inData = inDataSource,
             outFile = tagged_table_train,
             overwrite = TRUE,
             transformObjects = list(truth = truth_df),
             transformFunc = data_label)
```

Sprawdźmy, jak wyglądają opisane dane treningowe.

```
SELECT id, cycle, RUL
FROM [PredictiveMaintenance].[Train_Labels];
```

id	cycle	RUL
1	1	191
1	2	190
1	3	189
…		
1	190	2
1	191	1
1	192	0
…		

Skoro ostatni pomiar był przeprowadzony w momencie wystąpienia awarii, to liczba pozostałych cykli życia silnika równa się maksymalnej wartości klucza (zmiennej `cycle`) minus numer bieżącego cyklu zmiennej `id` pogrupowanej według wartości.

Do opisania danych testowych użyjemy tabeli `PredictiveMaintenance.PM_Truth`. Po ich zaimportowaniu do obiektu typu `data frame` ponumerujemy wiersze, co pozwoli połączyć je z danymi testowymi.

```
truth_df <- rxImport(truth_data_table)
truth_df$id <- 1:nrow(truth_df)
```

Od tego momentu opisanie danych testowych wymaga wykonania dokładnie tych samych operacji co opisanie danych treningowych. Po zdefiniowaniu tabeli źródłowej i docelowej należy przenieść między nimi dane, przekształcając je przy użyciu funkcji `data_label`.

Opisane dane testowe oraz użyte do wyliczenia zmiennej `RUL` liczby cyklów, w czasie których dany silnik działał prawidłowo, wyglądają następująco.

```
SELECT id, cycle, te.RUL, tr.RUL AS [Ground truth]
FROM [PredictiveMaintenance].[Test_Labels] AS Te
JOIN (SELECT RUL, ROW_NUMBER() OVER (ORDER BY (SELECT 1)) as rn
          FROM [PredictiveMaintenance].[PM_Truth]) as Tr
ON Te.id = Tr.rn;
```

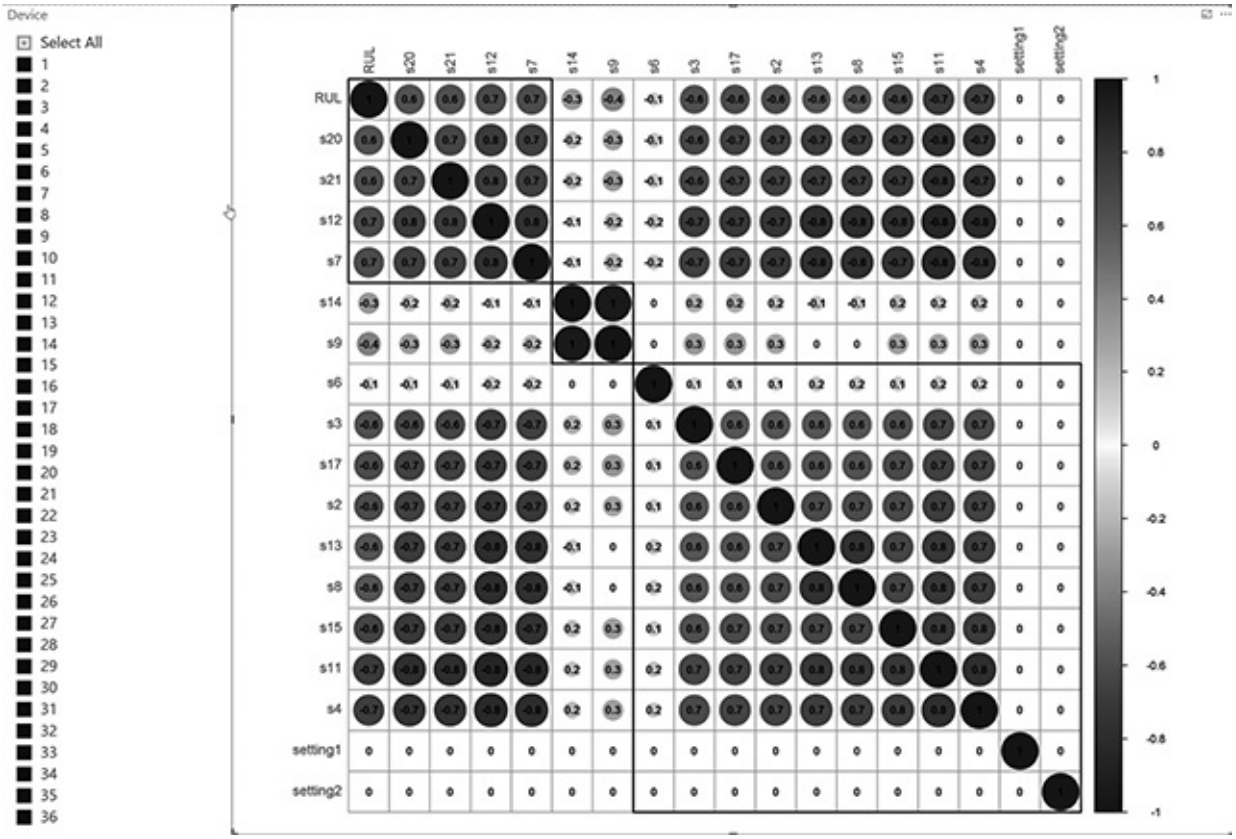
id	cycle	RUL	Ground truth
1	1	142	112
1	2	141	112
1	3	140	112
…			
1	29	114	112
1	30	113	112
1	31	112	112
…			

Zgodnie z opisem pozostały czas życia silnika został obliczony przez dodanie numeru ostatniego zmierzonego cyklu do wartości RUL z tabeli `PredictiveMaintenance.PM_Truth`.

7.4. Ocena przydatności danych

Analiza danych źródłowych pozwoliła ocenić jakość tych danych i zidentyfikować nieprzydatne, bo przyjmujące jedną wartość, zmienne. Przypomnijmy, zmienne, dla których odchylenie standardowe wynosi zero, to: `setting3`, `s1`, `s5`, `s10`, `s16`, `s18` i `s19`. Te zmienne należy usunąć.

Opisanie danych, czyli dodanie do ich zmiennej wyjściowej `RUL`, dało nam możliwość sprawdzenia korelacji między zmiennymi. Użyjemy do tego niestandardowej wizualizacji programu Power BI Desktop *Correlation plot*, tej samej, którą poznaliśmy w rozdziale 4. Dodając do raportu filtr, będziemy mogli sprawdzić korelacje między zmiennymi dla wszystkich oraz wybranych silników (rys. 7.6)[14].



Rysunek 7.6. Analiza korelacji między zmiennymi pokazała, że ustawienia silnika (zmienne `settings`) nie mają wpływu na pozostały czas ich bezpiecznej eksploatacji. Ponadto, wykryła dwie silnie skorelowane ze sobą zmienne wejściowe: `s14` i `s9`

Z dwóch skorelowanych ze sobą zmiennych wejściowych usuniemy tę, która jest słabiej skorelowana ze zmienną wyjściową, czyli zmienną `s14`. Wybrane w ten sposób zmienne zwracają widoki `PredictiveMaintenance.vTrain_Labels` i `PredictiveMaintenance.vTest_Labels`.

W punkcie 5.4.6 przeanalizowaliśmy wartości zmierzone przez jeden z czujników. Rozłożyliśmy szereg czasowy na jego składowe: trend, okres i szum oraz pokazaliśmy, jak wzmocnić trend i okres, wygładzając dane. Policzyliśmy też współczynnik autokorelacji i częściowej autokorelacji zmiennej i postawiliśmy hipotezę, że pozostały czas bezawaryjnej pracy silnika zależy nie tylko od wyników bieżących pomiarów, ale również wcześniejszych. Żeby ją zweryfikować, wzbogacimy dane o średnie ruchome. W tym przypadku będą to

średnie z pięciu ostatnich pomiarów. Jednak zastępując pięć wartości ich średnią, tracimy informację o różnicach między wartościami pojedynczych pomiarów. Ponieważ zmienność odczytów również może mieć wpływ na pozostały czas bezawaryjnej pracy silnika, dlatego do danych dodamy również ruchome odchylenie standardowe, policzone dla pięciu ostatnich wartości.

7.5. Wzbogacenie danych

Tym razem do wygładzenia szeregów czasowych użyjemy funkcji `rollapply` z biblioteki `zoo`. Ustawiając parametr `align` na wartość `right`, policzymy średnią i odchylenie k poprzednich obserwacji, a ustawiając wartość parametru `partial` na `TRUE`, wyliczymy statystyki dla wszystkich, również kilku pierwszych obserwacji. Obliczenie średniej ruchomej sprowadza się do pokazanego poniżej wywołania omawianej funkcji[15].

```
        rolling_mean <- zoo::rollapply(data = data,
                                       width = window,
                                       FUN = mean,
                                       align = "right",
                                       partial = 1)
```

Średnie i odchylenia ruchome policzymy dla wszystkich mierników. Ich nazwy zaczynają się literą `s`, po której występuje liczba, co pozwala automatycznie wybrać odpowiednie kolumny.

```
window_size <- 5
tagged_table_name <- "PredictiveMaintenance.vTrain_Labels"
tagged_table_train <- RxSqlServerData(table = tagged_table_name,
                                      colClasses = train_columns,
                                      connectionString = connection_string)
train_vars <- names(rxGetVarInfo(tagged_table_train))
sensor_vars <- train_vars[grep("s[[:digit:]]", train_vars)]
```

Wzbogacone dane zapiszemy w tabeli `PredictiveMaintenance.Train_Features`. Tak jak poprzednio, użyjemy do tego funkcji `rxDataStep`, ale tym razem przekształcimy zapisywane dane, dodając do nich wyniki wywołania funkcji `create_features`.

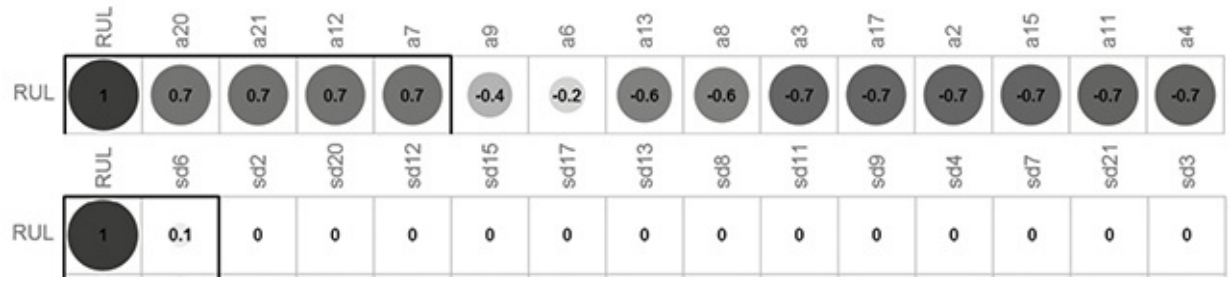
```
rxDataStep(inData = tagged_table_train,
             outFile = train_table_features,
             overwrite = TRUE,
             transformObjects = list(window = window_size,
                                     sensor = sensor_vars,
                                     data_type = "train"),
             transformFunc = create_features)
```

Dane testowe przygotujemy tak samo, tj. dla wszystkich mierników wyliczymy średnie i odchylenia ruchome. Z tak wzbogaconych danych usuniemy wszystkie cykle z wyjątkiem ostatniego dla każdego silnika. W ten sposób otrzymamy zbiór testowy zawierający niezależne od siebie przykłady. Ponadto zrównoważymy dane na potrzeby modeli klasyfikujących omówionych w następnym rozdziale.

```
if (!identical(data_type, "train"))
  {
    max_cycle <- plyr::ddply(data, "id", plyr::summarise, cycle = max(cycle))
    data <- plyr::join(max_cycle, data, by = c("id", "cycle"))
  }
```

7.6. Ocena zmiennych

Sprawdźmy korelacje między dodanymi cechami a zmienną wyjściową `RUL` (rys. 7.7).



Rysunek 7.7. Średnie ruchome (zmienne `a`...) są skorelowane z wyjściową, ale ruchome odchylenia standardowe (zmienne `sd`...) nie są

Okazało się, że przedstawienie zmienności pomiarów za pomocą odchylenia standardowego niewiele dało. Może policzenie różnic między największą a najmniejszą wartością zmierzoną w ostatnich pięciu cyklach okaże się lepszym pomysłem? Proszę samodzielnie sprawdzić tę hipotezę. Pomóc może przestudiowanie przedstawionej w następnym podrozdziale funkcji `normalize_data`.

Sprawdźmy jeszcze zakresy wartości zmiennych wejściowych.

```
train_summary <- rxSummary(formula = ~ .,
                           data = train_table_features,
                           summaryStats = c("min", "max"))
train_summary <- train_summary$sDataFrame
train_summary <- subset(train_summary, !Name %in% c("id", "RUL", "label1", "label2"))
train_summary
```

```
   Name         Min           Max
2   cycle      1.0000     3.620000e+02
3      s2    641.2100     6.445300e+02
4      s3   1571.0400     1.616910e+03
…
22     a4   1387.3800     1.432984e+03
23     a6     21.6000     2.161000e+01
…
34    sd2      0.0000     7.839133e-01
35    sd3      0.0000     1.244508e+01
36    sd4      0.0000     1.525936e+01
37    sd6      0.0000     7.071068e-03
…
```

Obok zmiennych mierzonych w tysiącach mamy zmienne mierzone w tysięcznych częściach całości. Może warto przeskalować wartości zmiennych?

7.7. Przekształcenie i wybór danych

Żeby zachować rozkład wartości, ale sprowadzić zmienne do wspólnej skali, należy je znormalizować. W danych nie ma wartości odstających, a więc możemy użyć metody *Min-Max*. Do tego będą nam potrzebne minimalne i maksymalne wartości każdej zmiennej wejściowej. Nie możemy ich jednak obliczyć, wywołując funkcje `min` i `max`. Proszę pamiętać, że **funkcja** `rxDataStep` **biblioteki** `RevoScaleR` **przetwarza dane w blokach, a więc jednocześnie w pamięci komputera znajduje się tylko jeden blok danych, a nie ich cały zbiór.** Zamiast używać wartości minimalne i maksymalne do poszczególnych grup, zapiszemy wyliczone w pokazany powyżej sposób statystyki w zmiennych.

```
train_vars <- train_summary$Name
train_vars_min <- train_summary$Min
train_vars_max <- train_summary$Max
```

Funkcja `normalize_data` normalizuje zmienne wejściowe, zachowując oryginalne wartości zmiennych `id` i `cycle`.

```
normalize_data <- function(data) {
  data <- as.data.frame(data)
  data_to_keep <- data[, c("id", "cycle")]
  names(data_to_keep) <- c("id", "cycle_orig")
  data$id <- NULL
  temp <- data[, vars]
  normalize <- function(x, min, max) {
    z <- (x - min) / (max - min)
    return(z)
  }
  temp <- mapply(normalize, temp, vars_min, vars_max)
  temp[is.nan(temp)] <- NA
  data <- data[, which(!names(data) %in% vars)]
  data <- cbind(data_to_keep, temp, data)
  data <- data[, apply(!is.na(data), 2, all)]
  data$cycle <- NULL
  return(data)
}
```

Znormalizowane dane zapiszemy w tabelach `PredictiveMaintenance.Train_Features_Normalized` i `PredictiveMaintenance.Test_Features_Normalized`. Użyty wyłącznie w celach poglądowych parametr `rowsPerRead` określa wielkość jednorazowo przetwarzanych bloków danych.

```
train_table_normalized <- RxSqlServerData(table = "PredictiveMaintenance.
Train_Features_Normalized",
                                    connectionString = connection_string)

rxDataStep(inData = train_table_features,
             outFile = train_table_normalized,
             transformObjects = list(vars = train_vars,
                                     vars_min = train_vars_min,
                                     vars_max = train_vars_max),
             transformFunc = normalize_data,
             overwrite = TRUE,
             rowsPerRead = 1000)
```

Na końcu ograniczymy liczbę zmiennych wejściowych na podstawie ich

współczynnika korelacji ze zmienną wyjściową. W tym celu odczytamy nazwy wszystkich zmiennych i usuniemy z nich zmienne `id` oraz `cycle`. Użycie numeru silnika do przewidywania czasu bezawaryjnej pracy innych silników nie ma sensu. Wiemy też, że początkowe numery cyklów, choć dla wszystkich silników zaczynają się od 1, w rzeczywistości nie odpowiadają pierwszemu cyklowi pracy silnika.

```
train_table <- RxSqlServerData(table = train_table_name,
                                    connectionString = connection_string)
train_df <- rxImport(train_table)
train_vars <- rxGetVarNames(train_df)
test_table <- RxSqlServerData(table = test_table_name,
                                   connectionString = connection_string)
test_df <- rxImport(inData = test_table)
train_vars <- train_vars[!train_vars  %in% c("id", "cycle_orig", "cycle")]
```

Do policzenia współczynnika korelacji między zmiennymi użyjemy funkcji `rxCor`. Jej pierwszym parametrem jest formuła w postaci `~ zmienna 1 + … + zmienna n`, drugim parametrem jest źródło danych. Zdefiniowaną formułę użyjemy do obliczenia macierzy korelacji między zmiennymi.

```
formula <- as.formula(paste("~", paste(train_vars, collapse = "+")))
correlation <- rxCor(formula = formula, data = train_df)
```

Następnie usuniemy z tej macierzy wszystkie kolumny z wyjątkiem kolumny zawierającej korelacje między zmiennymi wejściowymi a wyjściową zmienną `RUL`. Ponieważ interesuje nas siła korelacji, a nie jej kierunek, policzymy wartość bezwzględną współczynników korelacji, posortujemy je malejąco i usuniemy pierwszą pozycję (współczynnik korelacji między zmienną `RUL` a nią samą, który wynosi 1).

```
correlation <- correlation[, "RUL"]
correlation <- abs(correlation)
correlation <- correlation[order(correlation, decreasing = TRUE)]
correlation <- correlation[-1]
as.data.frame(correlation)
```

```
correlation
a4     0.73297608
a11    0.73227513
a21    0.72407226
a15    0.72314420
…
sd2    0.02385871
sd3    0.01515953
sd21   0.00569183
```

Na końcu pozostawimy *n* zmiennych najsilniej skorelowanych z wyjściową i ich nazwy przekształcimy w formułę uczącą o postaci `zmienna wyjściowa ~ zmienna wejsciowa 1 + … + zmienna wejściowa n`.

```
correlation <- correlation[1:top_variables]
formula <- as.formula(paste(paste("RUL~"), paste(names(correlation), collapse = "+")))
formula
```

```
RUL ~ a4 + a11 + a21 + a15 + a20 + a17 + a12 + a7 + a2 + a3 +
     s11 + s4 + s12 + s7 + s15 + s21 + s20 + s2 + s17 + a8 + a13 +
     s3 + s8 + s13 + a9 + s9 + sd6 + a6 + s6 + sd9 + sd13 + sd11 +
     sd20 + sd4 + sd12
```

7.8. Modelowanie

Do zbudowania modelu uczenia maszynowego są potrzebne:

- przykładowe dane; w tym przypadku dysponujemy trzema zestawami danych: opisanymi, wzbogaconymi i znormalizowanymi;
- algorytm reprezentujący wiedzę, na przykład w postaci równania liniowego

albo drzew decyzyjnych; w tym podrozdziale poznamy dwa nowe algorytmy: ogólny model liniowy i sztuczne sieci neuronowe; przypomnimy też regresję liniową i użyjemy wzmocnionych drzew decyzyjnych oraz lasu losowego; te dwa algorytmy zostały przedstawione w następnym rozdziale;

- funkcja kosztu; w rozdziale 3 poznaliśmy dwie popularne funkcje kosztu: błąd średniokwadratowy i funkcję 0-1;
- metoda optymalizacji procesu uczenia; w rozdziale 3 przedstawiłem trzy metody optymalizacji: przez losowy dobór wartości parametrów, przeszukiwania siatki wartości parametrów i metodę stycznych; w punkcie 7.8.1 poznamy powszechnie używaną do optymalizacji uczenia metodę najszybszego spadku gradientu wraz z jej najpopularniejszymi wariantami.

Zaczniemy od zbudowania pięciu modeli wytrenowanych na tych samych, wzbogaconych danych. Następnie powtórzymy trening, używając opisanych i znormalizowanych danych. Na końcu zautomatyzujemy procedury trenowania i korzystania z modeli, implementując je po stronie serwera SQL Server.

Przypomnijmy sobie najważniejsze informacje na temat regresji. Jeżeli zmienna wyjściowa jest numeryczna, zadaniem modelu jest znalezienie wystarczająco dobrego przybliżenia funkcji tej zmiennej do zmiennych wejściowych. W tym przypadku zmienną wyjściową jest zmienna `RUL`. Wyniki predykcji mogą się różnić od prawdziwych wartości zmiennej wyjściowej. Im mniejsze te różnice, tym dokładniejszy model. Do oceny modelu używa się funkcji kosztu θ. Uczenie modelu polega na znalezieniu takich wartości parametrów, dla których funkcja θ przyjmuje minimum. Zobaczmy, jak można znaleźć takie wartości parametrów.

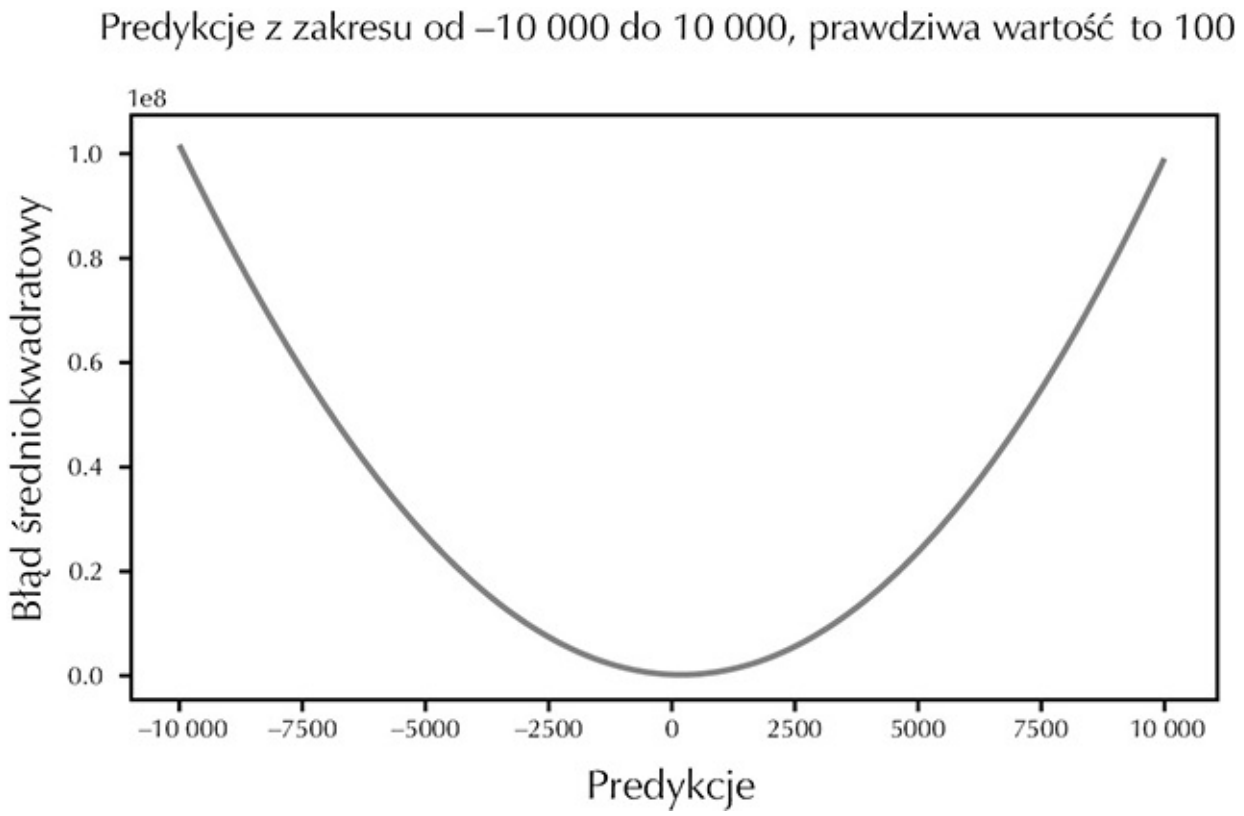
7.8.1. Uczenie na błędach

Jedną z najczęściej stosowanych do oceny modeli regresji funkcji kosztu jest

błąd średniokwadratowy *MSE*. Dla jednej zmiennej wejściowej *x* i *n* przykładów błąd średniokwadratowy wynosi

$$J(\beta_0, \beta_1) = \frac{1}{2n} \Sigma_{i=1}^{n} (\beta_0 + \beta_i(\mathrm{x}_i) - \mathrm{y}_i)^2$$

Zauważmy, że błąd rośnie z kwadratem różnicy między przewidzianą a prawdziwą wartością zmiennej wyjściowej. To dobrze, bo model będzie się szybciej uczył na dużych błędach. Jeszcze ważniejsze jest to, że **funkcja kosztu MSE ma tylko jedno, globalne minimum**. Brak lokalnych minimów znacznie upraszcza znalezienie globalnego minimum funkcji (rys. 7.8).



Rysunek 7.8. Jeden duży błąd kosztuje model znacznie więcej niż wiele drobnych pomyłek. Weźmy 1000 przykładów. Jeżeli błąd dla jednego z nich wyniósł 5000, a dla pozostałych 999 błąd wyniósł 0, to błąd średniokwadratowy MSE wyniesie 25 000. Jeżeli błąd dla wszystkich 1000 przykładów wyniósł 10, to błąd średniokwadratowy MSE modelu będzie równy 100

W tym miejscu powinniśmy zapytać, czy funkcja kosztu θ mierzy błąd treningowy, czy testowy? Skoro do uczenia używa się wyłącznie danych

treningowych, to **funkcja θ ocenia model na podstawie danych treningowych**. Wiemy już, że dostatecznie skomplikowany model pozwala zmniejszyć błąd treningowy do zera. Wiemy też, że mały błąd treningowy nie zawsze przekłada się na mały błąd testowy.

Żeby zapobiec nadmiernemu dopasowaniu modelu do danych treningowych, do reguły uczącej dodaje się człon z regularyzacją. Jego zadaniem jest zwiększenie kosztu bardziej złożonych modeli. Wróćmy do uproszczonego przykładu z jedną zmienną i dwoma parametrami β_0 i β_1. Uzupełniona o człon z regularyzacją funkcja kosztu wygląda następująco:

$$J(\beta_0, \beta_1) = \frac{1}{2n}\Sigma_{i=1}^{n}(\beta_0 + \beta_i(x_i) - y_i)^2 + \lambda\Sigma_{j=1}^{k}\beta_j^2$$

Hiperparametr λ (lambda) określa wpływ członu z regularyzacją na funkcję kosztu. Im jest on większy, tym większe ryzyko niedopasowania modelu. Im jest on mniejszy, tym większe ryzyko nadmiernego dopasowania modelu.

Jeżeli w członie z regularyzacją dodaje się kwadraty wartości parametrów, tak jak w tym przypadku, mówimy o regularyzacji *L2*, nieformalnie nazywanej *Ridge*. Zmniejsza ona wartości parametrów. Siła redukcji zależy od wartości λ. **Działanie regularyzacji *L2* polega na eliminowaniu mniej przydatnych wartości zmiennych wejściowych** (wartości, dla których błąd predykcji był największy).

Jeżeli w członie z regularyzacją dodaje się wartości bezwzględne parametrów, mamy do czynienia z regularyzacją *L1*, nieformalnie nazywaną *Lasso*

$$J(\beta_0, \beta_1) = \frac{1}{2n}\Sigma_{i=1}^{n}(\beta_0 + \beta_i(x_i) - y_i)^2 + \lambda\Sigma_{j=1}^{k}|\beta^2|$$

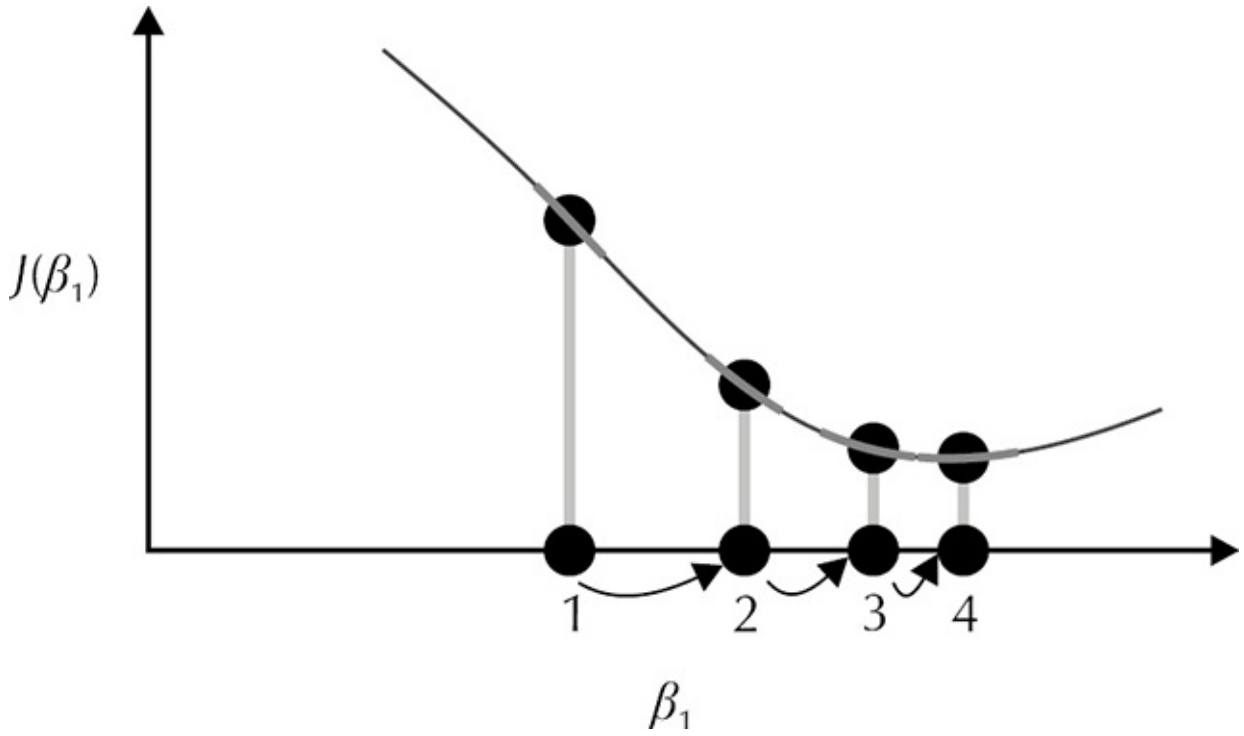
Ściąga ona wartości parametrów do zera. Siła ściągania zależy od wartości λ. **Działanie regularyzacji *L1* polega na eliminowaniu mniej przydatnych zmiennych wejściowych** (zmiennych o najmniejszej sile predykcyjnej).

Łącząc oba powyższe człony, otrzymamy elastyczną regularyzację sieci

$$J(\beta_0, \beta_1) = \frac{1}{2n}\Sigma_{i=1}^{n}(\beta_0 + \beta_i(x_i) - y_i)^2 + \lambda\Sigma_{j=1}^{k}|\beta^2| + \lambda\Sigma_{j=1}^{k}\beta_j^2 \quad [27]$$

Zatrzymajmy się ponownie na chwilę przy przykładzie modelu z jedną zmienną wejściową. Uprośćmy go jeszcze bardziej, usuwając parametr β_0. Teraz możemy przedstawić funkcję kosztu na wykresie liniowym jako krzywą, której wartości zależą od wartości parametru β_1. Uczenie na błędach polega na szukaniu minimum funkcji kosztu. Odpowiednio zmieniając wartości parametru, możemy zmniejszyć błędy modelu i w ten sposób zminimalizować funkcję kosztu. Kierunek zmian wartości parametru wyznacza pochodna funkcji kosztu.

Pochodna funkcji to miara szybkości zmian wartości funkcji względem najmniejszych zmian jej argumentów. Leibniz zapisywał pochodną jako $\frac{dy}{dx}$, co czyta się *pochodna y względem x*. Formalnie pochodną definiuje się jako $\frac{\Delta f(a)}{a}$. Pochodna krzywej, takiej jak funkcja kosztu *MSE*, reprezentuje nachylenie tej krzywej w danym punkcie. To znaczy, że mniejszą od bieżącej wartość funkcji kosztu możemy znaleźć, poruszając się w kierunku przeciwnym do znaku pochodnej (rys. 7.9).



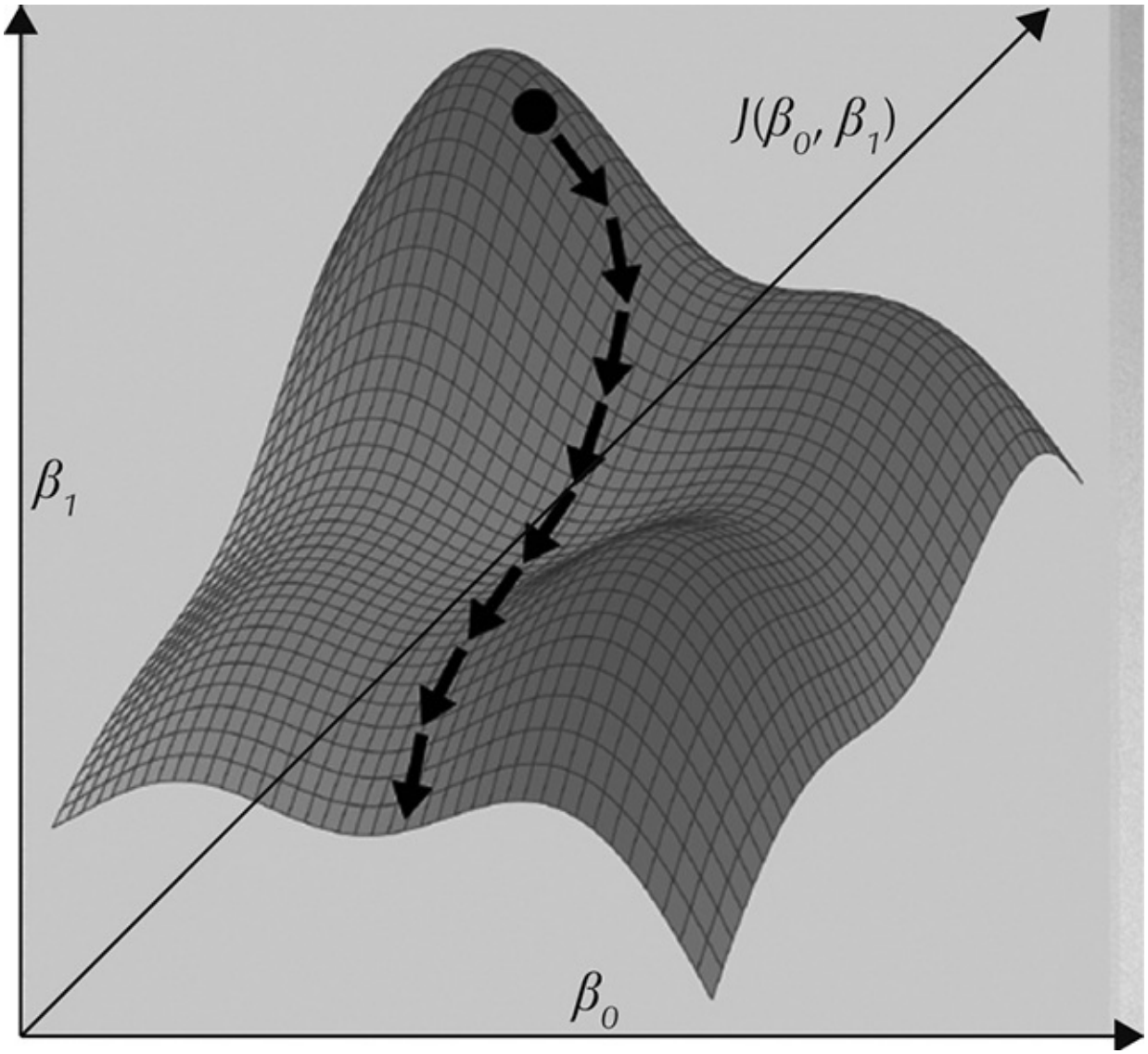
Rysunek 7.9. W pierwszym punkcie pochodna jest ujemna, czyli zmieniamy wartość parametru, przesuwając się w prawo, w kierunku przeciwnym do znaku pochodnej. Pochodna policzona w drugim punkcie też jest ujemna, czyli ponownie zwiększamy wartość parametru, itd.

Pochodną funkcji kosztu należy obliczyć względem wszystkich parametrów modelu. Tak obliczoną pochodną nazywa się gradientem i oznacza symbolem ∇ (*nabla*). **Gradient jest wektorem, którego składowe są pochodnymi cząstkowymi funkcji *f***

$$\nabla f(\vec{w}) = \frac{\partial f}{\partial w_0}, \ldots, \frac{\partial f}{\partial w_n}$$

Czyli uczenie polega na zmienianiu wartości parametrów w kierunku najszybszego spadku gradientu (rys. 7.10).

$$\Delta \vec{w} = -\eta \nabla f(\vec{w})$$



Rysunek 7.10. Wykres przedstawia funkcję kosztu zależną od dwóch parametrów. Uczenie modelu polega na zmianie wartości parametrów w kierunkach wyznaczonych przez gradient funkcji kosztu

Początkowo wartości parametrów są przypadkowe, ustalane metodą losowania. Dla tych wartości jest obliczany gradient funkcji kosztu. Następnie zmieniane są wartości parametrów w kierunku najszybszego spadku gradientu. Współczynnik uczenia (kolejny hiperparametr modelu) kontroluje, o ile jednorazowo zostaną zmienione wartości parametrów. Wystarczająco mały współczynnik uczenia gwarantuje znalezienie minimum funkcji kosztu, ale proces uczenia będzie wolny. Większy współczynnik uczenia skraca proces, ale zwiększa ryzyko nieznalezienia minimum funkcji kosztu.

Metody najszybszego spadku gradientu

Metoda optymalizacji procesu uczenia ma bezpośredni wpływ na szybkość uczenia i pośredni na jego wyniki [28]. Dane treningowe mogą liczyć miliony przykładów opisanych setkami zmiennych. Wielokrotne (uczenie maszynowe często wymaga powtórzenia kilkuset, a w niektórych przypadkach nawet kilkudziesięciu tysięcy cykli) liczenie gradientu dla wszystkich przykładów może wymagać tak dużej mocy obliczeniowej i tak wiele pamięci, że zakończenie treningu w skończonym czasie byłoby możliwe tylko dla dużych współczynników uczenia.

Podstawowa wersja metody najszybszego spadku gradientu, *Batch Gradient Descent*, polega na aktualizacji wartości parametrów na podstawie gradientu policzonego dla wszystkich przykładów. Czyli po policzeniu gradientu dla całego zbioru treningowego następuje jednorazowa aktualizacja wszystkich parametrów. Te dwie operacje są nazywane cyklem albo epoką

$$\omega_1 = \omega_0 - \lambda\nabla \frac{1}{2n} \sum_{i=1}^{n} (\beta_0 + \beta_i(x_i) - y_i)^2$$

Metoda ta daje gwarancję znalezienia minimum funkcji kosztu *MSE*, o ile tylko współczynnik uczenia jest wystarczająco mały.

Stochastyczna metoda najszybszego spadku gradientu, *Stochastic Gradient Descent*, estymuje gradient na podstawie jednego przykładu, losowo wybranego w danym cyklu. Obliczony w ten sposób gradient jest mało stabilny – jego wartości dla kolejnych cykli mogą być zupełnie różne, co daje efekt miotania

$$\omega_1 = \omega_0 - \lambda\nabla \frac{1}{2n} (\beta_0 + \beta_1(x_0) - y_0)^2$$

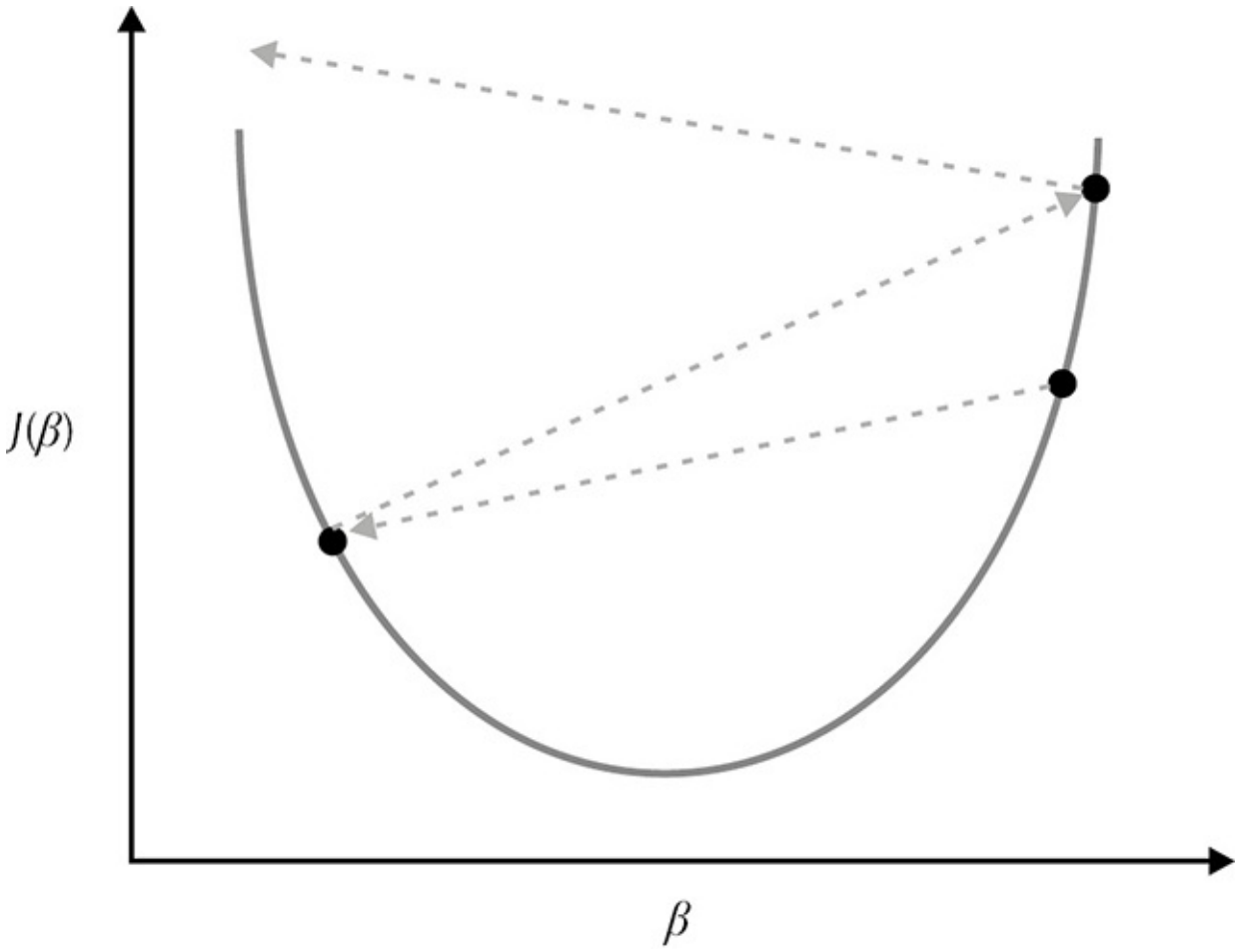
Okazuje się jednak, że wprowadzenie elementu losowości zapobiega przerwaniu uczenia w lokalnym minimum funkcji kosztu. Powszechnie uważa się, że

stochastyczna metoda najszybszego spadku gradientu daje najlepsze wyniki uczenia. Przede wszystkim dlatego, że aktualizacja wag po ocenieniu jednego przykładu wielokrotnie skraca proces uczenia. Często mamy do czynienia z sytuacją, w której już na podstawie gradientu wyznaczonego dla jednego przykładu można prawidłowo określić kierunek zmiany parametrów. W takich przypadkach ocena wszystkich pozostałych przykładów w ramach tego samego cyklu (parametry są zmieniane po zakończeniu każdego cyklu) jest stratą czasu. Użyta w dalszej części rozdziału funkcja `rxNeuralNet` pozwala ograniczyć liczbę przykładów analizowanych w poszczególnych cyklach parametrem `miniBatchSize`, ale jest on brany pod uwagę tylko przy uczeniu sztucznych sieci neuronowych za pomocą akceleratora GPU.

Najczęściej używaną metodą optymalizacji procesu uczenia jest metoda najszybszego spadku gradientu dla mini grupy, *Mini-Batch Gradient Descent*. Łączy ona zalety obu powyższych metod, estymując wartość gradientu na podstawie małego podzbioru przykładów. Wielkość tego podzbioru kontroluje hiperparametr *batch size*. Ze względu na architekturę procesorów graficznych ogólnego przeznaczenia wartości tego hiperparametru są potęgami liczby 2. Typowe wielkości mini grupy to 8, 16, 32, 64, 128 i 256. Im większa wartość hiperparametru *b*, tym wolniejszy jest proces uczenia

$$\omega_1 = \omega_0 - \lambda\nabla \frac{1}{2b} \sum_{i=1}^{b} (\beta_0 + \beta_i (x_i) - y_i)^2$$

Metoda najszybszego spadku gradientu dla minigrupy nie jest doskonała. Przede wszystkim utrudnia dobór optymalnego współczynnika uczenia. Zbyt mały współczynnik uczenia powoduje, że proces uczenia się nie zakończy, zbyt duży zwiększa ryzyko nietrafiania w minimum funkcji kosztu, a nawet oddalenia się od niego w kolejnych cyklach (rys. 7.11).



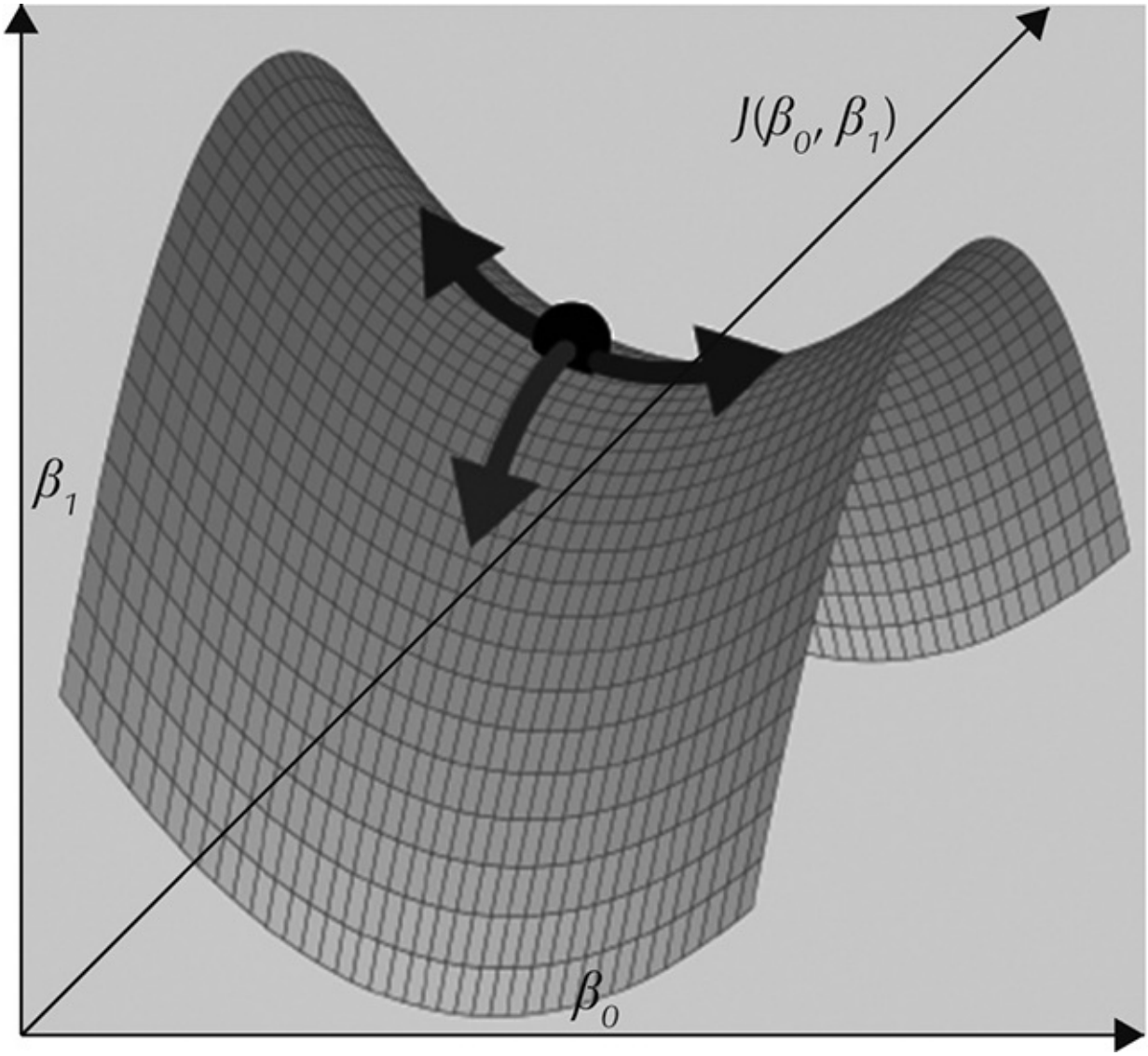
Rysunek 7.11. Jeżeli współczynnik uczenia jest zbyt duży, zmiana wartości parametrów w kierunku spadku gradientu może być zbyt duża i spowodować minięcie minimum funkcji kosztu. W kolejnym cyklu kierunek gradientu będzie przeciwny, a duża zmiana wartości parametrów może spowodować zwiększenie funkcji kosztu

Problem ten doczekał się wielu rozwiązań. Najprostsze polega na zmienianiu współczynnika uczenia w kolejnych cyklach. Na przykład początkową wartość współczynnika uczenia ustawia się na 0,01, w każdym kolejnym cyklu zmniejsza się ją o 0,005, żeby po 50 cyklach przywrócić wartość 0,01. Ta metoda nie tylko skraca czas uczenia, ale również zapobiega utknięciu w lokalnym minimum funkcji kosztu (używane w sztucznych sieciach neuronowych funkcje kosztu nie są wypukłe i mają lokalne minima). Jeszcze prostszym rozwiązaniem jest zmniejszanie współczynnika uczenia co określoną liczbę cykli. Jednak obie te optymalizacje wymagają podania wartości

dodatkowych hiperparametrów i nie uwzględniają charakterystyki danych treningowych. W kolejnym punkcie przyjrzymy się metodom automatycznego doboru współczynnika uczenia do danych i postępów w uczeniu modelu.

Optymalizacje metody najszybszego spadku gradientu

Metoda najszybszego spadku gradientu daje najlepsze wyniki, gdy gradient spada podobnie szybko we wszystkich kierunkach funkcji kosztu. Miejsca, w których spadek gradientu względem jednego parametru jest znacznie mniejszy niż względem innego, prowadzą do trawersowania zbocza funkcji kosztu albo nawet do przerwania uczenia w takim punkcie siodłowym (rys. 7.12).



Rysunek 7.12. Optymalizacja uczenia metodą najszybszego spadku gradientu zmienia wartości wszystkich parametrów na podstawie wspólnego współczynnika uczenia λ. W rezultacie proces uczenia nie przesuwa się w dół zbocza funkcji kosztu, tylko po nim zygzakuje albo nawet utyka na stałe w punkcie siodłowym

Dodanie pędu (momentu) pozwala metodzie najszybszego spadku gradientu szybciej przesuwać się w kierunku, w którym gradient rzeczywiście maleje. Pęd oblicza się na podstawie poprzedniej (przeprowadzonej w poprzednim cyklu) zmiany wartości parametrów pomnożonej przez hiperparametr γ

$$\Delta w_{i,j}(n) = \eta \delta_j x_{i,j} + \gamma \Delta w_{i,j}(n-1)$$

Ustawienie tego hipermarametru na 0,9 daje efekt przyspieszania ruchu

kontynuowanego w tym samym kierunku, osiągnięty przez szybsze zmienianie wartości tych parametrów, który zostały zmienione w poprzednim cyklu.

Kolejna optymalizacja polega na wyznaczeniu pędu nie na podstawie poprzedniej pozycji (pozycji z poprzedniej epoki), ale przyszłej, czyli pozycji, którą proces uczenia osiągnie w następnym cyklu. Dodanie pędu Nesterova *NAG* nie tylko zmniejsza efekt miotania, ale również poprawia kierunek zmian procesu uczenia. W tej metodzie najpierw wykonuje się dużą zmianę parametrów w kierunku wyznaczonym przez poprzednio obliczony gradient, następnie oblicza się gradient w tak osiągniętym punkcie i przeprowadza jego korektę. Otrzymany w ten sposób pęd Nesterova jest używany do zmiany wartości parametrów.

Optymalizacja przez dodanie pędu rozwiązuje problem niestabilnego gradientu i eliminuje efekt miotania. Nadal jednak musimy określić współczynnik uczenia. Tu z pomocą przychodzą metody automatycznego lub semiautomatycznego dobierania wartości tego hiperparametru.

Pierwszą metodą automatycznego dobierania współczynnika uczenia był algorytm Adagrad (*Adaptive Gradient*). Jego działanie polega na zmniejszaniu współczynnika uczenia parametrów o częściej współwystępujących wartościach i zwiększaniu tego współczynnika dla parametrów, których wartości współwystępują rzadziej. Dlatego metoda ta doskonale sprawdza się dla zbiorów rzadkich. Zastosowany algorytm powoduje jednak szybkie zmniejszanie się współczynnika uczenia i może prowadzić do znacznego wydłużenia czasu uczenia modelu [29]. Adadelta jest odmianą metody Adagrad i polega na ograniczeniu liczby zapamiętywanych gradientów do *w* ostatnich.

Połączeniem metody Adagrad z pędem jest algorytm Adam (*Adaptive Moment Estimation*). Metoda ta nie tylko przechowuje ważone średnie kwadratów poprzednich gradientów, tak jak Adagrad, ale również ważone średnie poprzednich gradientów, czyli pęd [30]. Jak można się domyślić,

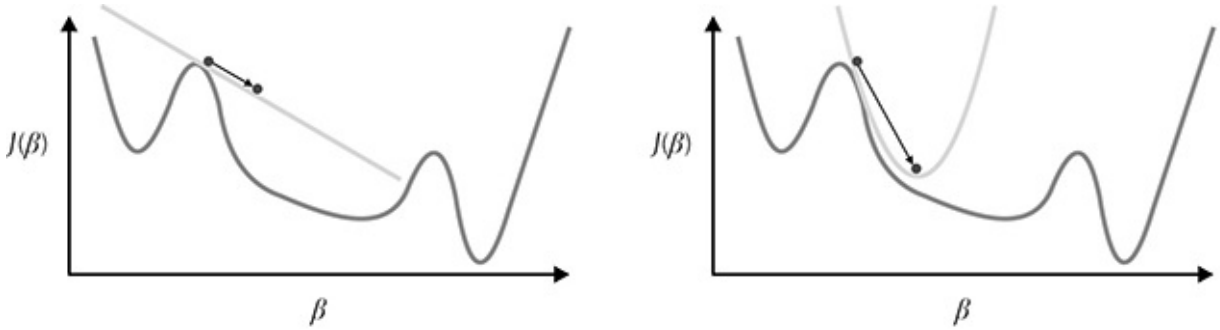
algorytm Nadam jest z kolei połączeniem metody Adagrad z pędem Nesterova. **Oba te algorytmy, Adam i Nadam, są obecnie uważane na najlepsze optymalizacje uczenia metodą najszybszego spadku gradientu.**

Metody optymalizacji wzdłuż współrzędnej

Zmiana wartości parametrów jest kosztowna, dlatego algorytmy najszybszego spadku gradientu aktualizują je raz po zakończeniu cyklu. Gdyby jednak jednorazowo zmieniać wartość wybranego parametru, można by taką zmianę przeprowadzać po obliczeniu gradientu dla pojedynczych przykładów. Ponadto minimalizowanie funkcji kosztu według jednej współrzędnej pozwala zastąpić skomplikowany problem obliczenia gradientu serią prostszych obliczeń, wykonywanych w pętli.

Podstawowa wersja metody optymalizacji wzdłuż współrzędnej *CCD*, *Cyclic Coordinate Descent*, polega na sprawdzaniu, jeden po drugim, parametrów i znalezieniu tego, którego modyfikacja najbardziej zmniejszy bieżącą wartość funkcji kosztu. Sprawdzanie i zmiana wartości znalezionego parametru są wykonywane dla każdego przykładu, czyli jest to algorytm liniowy, a nie blokowy.

Rozwinięciem metody optymalizacji wzdłuż współrzędnej jest algorytm *DCA* (*Dual Coordinate Ascent*). Oryginalny problem, czyli znalezienie minimum funkcji kosztu przez obliczenie jej pochodnych względem wszystkich parametrów, zastępuje problem dualny. Oszacowane zostają wartości drugich pochodnych funkcji kosztu względem wszystkich parametrów. Utworzona w ten sposób macierz Hessego jest używana do znalezienia parametru, którego zmiana wartości przybliży nas do maksimum dualnej funkcji kosztu i tym samym najbardziej zmniejszy bieżącą wartość funkcji kosztu (rys. 7.13).



Rysunek 7.13. Po lewej optymalizacja metodą najszybszego spadku gradientu, po prawej metodą SDCA

Metody optymalizacji problemu dualnego wymagają obliczenia, zapisania w pamięci i aktualizowania na bieżąco macierzy Hessego i wartości gradientu wyznaczonych bezpośrednio dla funkcji kosztu. Wpływa to na ich skalowalność i ogranicza stosowanie do niewielkich zbiorów danych treningowych. Nie wymagają one za to określenia współczynnika uczenia, bo jego optymalna wartość jest znajdowana dla każdego analizowanego przykładu – wyznacza ją minimum bieżącego przybliżenia oryginalnej funkcji kosztu. Rozwiązanie problemu ograniczonej skalowalności polega na zastosowaniu metody *SDCA* (*Stochastic Dual Coordinate Ascent*), w której macierz Hessego nie jest wyliczana, a jedynie estymowana [31].

7.8.2. Regresja liniowa

W rozdziale 3 poznaliśmy model regresji między jedną zmienną wejściową (objaśniającą) a zmienną wyjściową (objaśnianą). W praktyce modele regresji są stosowane do modelowania zależności między wieloma zmiennymi objaśniającymi a objaśnianą

$$y = f(x) = b_0 + b_1x_1 + b_2x_2 + ... + b_mx_m$$

W 1908 roku Karl Pearson nazwał taki model regresji liniowej regresją wieloraką.

Przypomnijmy, że do zbudowania modelu uczenia maszynowego są potrzebne dane, algorytm reprezentacji wiedzy, funkcja kosztu i algorytm optymalizacji procesu uczenia. Dane zostały już przygotowane. Pierwszym zastosowanym algorytmem reprezentacji zależności między zmiennymi wejściowymi a wyjściową będzie regresja wieloraka. Koszt modelu policzymy jako jego błąd średniokwadratowy, a proces uczenia zostanie zoptymalizowany metodą *SDCA*.

Modele regresji liniowej należą do jednych z najczęściej stosowanych modeli uczenia maszynowego. Dzięki swojej prostocie uczenie jest szybkie i skalowalne, a więc nie wymaga dużej mocy obliczeniowej ani dużo pamięci RAM. W dodatku prostota modelu zmniejsza ryzyko jego nadmiernego dopasowania. Jednak modele regresji liniowej prawidłowo reprezentują zależności między zmiennymi, o ile są spełnione cztery wymagania.

Założenie liniowości wymaga, żeby zmienne wejściowe były liniowo skorelowane z wyjściową. Wiemy już, że siłę korelacji liniowej między zmiennymi numerycznymi mierzy współczynnik *r*-Pearsona. Znak tego współczynnika korelacji określa kierunek zależności – dodatni oznacza, że wraz ze wzrostem wartości jednej zmiennej wartości drugiej zmiennej również rosną, ujemny – że wraz ze wzrostem wartości jednej zmiennej wartości drugiej zmiennej maleją. Wartości współczynnik *r*-Pearsona mieszczą się w zakresie od –1 do 1, gdzie –1 oznacza stuprocentową korelację ujemną, 0 brak korelacji, a 1 stuprocentową korelację dodatnią. Czyli wartość bezwzględna współczynnika *r*-Pearsona określa siłę korelacji – im jest ona większa, tym silniejsza korelacja.

Jeżeli zmienne są skorelowane nieliniowo, należy je przekształcić. W takich przypadkach najczęściej skaluje się je logarytmicznie. Skalowanie logarytmiczne spłaszcza odstające wartości, a że wartość współczynnika *r*-Pearsona silnie zależy od odstających wartości, ich spłaszczenie zwykle zwiększa jego wartość. W ten sposób spełnimy drugie wymaganie modelu regresji liniowej.

Założenie normalności rozkładu wymaga, żeby w danych nie występowały wartości odstające. Metody wykrywania wartości odstających zostały opisane w rozdziale 4, a w rozdziale 5 znajduje się punkt poświęcony ich usuwaniu.

Założenie niezależności zmiennych wejściowych wymaga, żeby korelacje między zmiennymi objaśniającymi nie były zbyt silne. Na etapie przygotowywania danych zmienne nadmiarowe należy usnąć.

Założenie braku autokorelacji wymaga, żeby bieżące wartości zmiennych wejściowych nie zależały zbyt silnie od ich poprzednich wartości. Żeby je spełnić, z szeregów czasowych usuwa się zmienne monotoniczne, takie jak znacznik czasu czy numer cyklu. Przed usunięciem ze znaczników czasu wydobywa się wartościowe zmienne, tak jak zostało to wyjaśnione w punkcie 5.2.1.

Poprawność modelu regresji liniowej ocenia się *ex post*, przez analizę jego wartości resztkowych (błędów modelu). Czyli najpierw buduje się model, uczy go, a następnie – na podstawie błędów zwróconych przez ten model, czyli reszt – ocenia się jego poprawność.



Wszystkie omówione w tym i następnym rozdziale modele uczenia maszynowego zostały zbudowane przy użyciu funkcji bibliotek `MicrosoftML i RevoScaleR`. Biblioteki te są automatycznie instalowane z usługą SQL Server Machine Learning Services i oprócz najnowszych implementacji algorytmów uczenia maszynowego udostępniają funkcje umożliwiające ocenę i przygotowanie danych. Wszystkie algorytmy uczenia maszynowego zaimplementowane w bibliotece `MicrosoftML` wymagają numerycznych zmiennych wejściowych.

Algorytm regresji liniowej został zaimplementowany w funkcji `rxFastLinear`. Funkcja ta jest wielowątkowa (uczenie odbywa się z użyciem wszystkich dostępnych procesorów) i nie ładuje jednocześnie całego zbioru treningowego do pamięci. Do optymalizacji procesu uczenia została zastosowana metoda *SDCA*, co oznacza, że wartości parametrów są modyfikowane na podstawie pojedynczych przykładów, a nie ich grup. Jest to więc algorytm uczenia online, którego wyniki mogą zależeć od kolejności przykładów. Dlatego, żeby zapewnić

powtarzalność wyników, należy wyłączyć tasowanie przykładów (`shuffle = FALSE`) i liczbę wątków ograniczyć do jednego (`trainThreads = 1`). Szczegółowe informacje na temat funkcji `rxFastLinear` uzyskamy, wyświetlając jej pomoc.

```
?MicrosoftML::rxFastLinear
```

Zbudujmy model dla 15 zmiennych wejściowych najsilniej skorelowanych ze zmienną wyjściową. Formuła ucząca wygląda następująco:

```
top_variables <- 15
…
formula
```

```
RUL ~ a4 + a11 + a21 + a15 + a20 + a17 + a12 + a7 + a2 + a3 +
     s11 + s4 + s12 + s7 + s15
```

Funkcja `rxFastLinear` może być używana do regresji lub klasyfikacji – typ uczonego modelu określa parametr `type`. Wymaganymi parametrami są formuła ucząca i zbiór danych treningowych. Pozostałe parametry mają wartości domyślne.

Wyjątkową cechą funkcji `rxFastLinear` jest zdolność automatycznego dobierania wartości niektórych parametrów do danych treningowych i jakości trenowanych modeli – te parametry można pominąć. Opcjonalne parametry, których wartości należy samodzielnie określić, zostały zaznaczone pogrubieniem.

```
linear_model <- MicrosoftML::rxFastLinear(formula = formula,
                                          data = train_df,
                                          type = "regression",
                                          lossFunction = squaredLoss(),
                                          convergenceTolerance = 0.01,
                                          shuffle = TRUE,
                                          trainThreads = NULL,
                                          normalize = "warn",
    maxIterations = NULL,
                                          l1Weight = NULL,
                                          l2Weight = NULL)
```

```
Warning: A normalizer is needed for this trainer. Either add a normalizing transform or
use the 'norm=Auto', 'norm=Yes' or 'norm=No' options.
Beginning processing data.
Rows Read: 20631, Read Time: 0, Transform Time: 0
Beginning processing data.
Beginning processing data.
Rows Read: 20631, Read Time: 0, Transform Time: 0
Beginning processing data.
Beginning processing data.
Rows Read: 20631, Read Time: 0, Transform Time: 0
Beginning processing data.
Using 2 threads to train.
Automatically choosing a check frequency of 2.
Auto-tuning parameters: maxIterations = 72.
Auto-tuning parameters: L2 = 0.0001.
Auto-tuning parameters: L1Threshold (L1/L2) = 0.
Using best model from iteration 52.
…
```

Domyślnie zmienne wejściowe są normalizowane, o ile tego wymagają. Komunikat informuje nas, że normalizację zmiennych można wyłączyć lub włączyć parametrem `norm`. Proszę pamiętać, że nie zawsze sprowadzenie wszystkich zmiennych do zakresu <-1:1> poprawia jakość modelu, niezależnie od wyświetlonego ostrzeżenia.

Model został wytrenowany przy użyciu dwóch procesorów, z automatycznie dobraną częstotliwością synchronizacji wątków równą 2.

Najciekawsze są cztery ostatnie z pokazanych wierszy wyniku. Widzimy, że maksymalna liczba iteracji została automatycznie ustalona na 72. To ważne, bo **odpowiednio wczesne przerwanie uczenia jest prostą i skuteczną metodą zapobiegania nadmiernemu dopasowaniu modeli**. Widzimy też, że finalnie został wybrany model, którego trening trwał 52 iteracje. Oznacza to, że jakość modelu jest sprawdzana po każdej iteracji i na końcu jest wybierany model najlepszy. Jeżeli kilka modeli kandydujących jest równie dobrych, wybierany jest najprostszy z nich.

Widzimy też, że automatycznie zostały dobrane wartości współczynników regularyzacji *L1* i *L2*. W tym przypadku wartość współczynnika regularyzacji *L1* została ustawiona na 0, a więc została zastosowana regularyzacja *Lasso*.

Wynikiem wywołania funkcji `rxFastLinear` jest obiekt `linear_model` klasy `mlModel`. Najważniejsze informacje o nauczonym modelu zwraca funkcja `summary`. Są to: sposób wywołania funkcji `rxFastLinear`, formuła ucząca, nazwa zbioru treningowego i znalezione wartości parametrów.

```
summary(linear_model)
```

```
Call:
MicrosoftML::rxFastLinear(formula = formula, data = train_df,
     type = "regression", lossFunction = squaredLoss(), l2Weight = NULL,
     l1Weight = NULL, trainThreads = NULL, convergenceTolerance = 0.01,
     maxIterations = NULL, shuffle = TRUE, normalize = "warn")

SDCAR (RegressorTrainer) for: RUL~a4+a11+a21+a15+a20+a17+a12+a7+a2+a3+s11+s4+s12+s7+s15
Data: train_df

First 16 of 16 Non-zero Coefficients:
(Bias): 0.1415848
a7: 2.943531
s7: 2.889531
a4: -2.716547
a12: 2.695788
s12: 2.637825
s4: -2.137015
a2: 1.744637
a20: 0.3153203
a21: 0.1904524
a3: -0.1455026
s11: -0.1183612
a17: 0.09869445
a11: -0.0847025
s15: -0.006014728
a15: -0.002864076
```

Zapytania predykcyjne do nauczonych modeli można wysłać funkcją `rxPredict`. Jej obowiązkowymi parametrami są `modelObject` oraz `data`. Pierwszy pozwala wskazać odpytywany model, a drugi użyte do predykcji dane. Domyślnie funkcja zwraca wyniki predykcji, ale ustawiając wartość parametru `writeModelVars` na `TRUE`, możemy je uzupełnić o wszystkie zmienne modelu. Możliwe jest też dodanie do wyników wskazanych zmiennych ze zbioru użytego do predykcji. Poniższy przykład pokazuje, jak zapisać wyniki predykcji uzupełnione o wartości zmiennej `RUL` odczytane ze zbioru testowego.

```
linear_model_prediction <- rxPredict(modelObject = linear_model,
                                     data = test_df,
                                     #writeModelVars = TRUE,
                                     extraVarsToWrite = "RUL")
head(linear_model_prediction)
```

```
   RUL       Score
1 112   159.51805
2  98   111.14557
3  69    65.35429
4  82   105.08727
5  91    73.19021
6  93   124.69797
```

Porównując prawdziwe wartości z wynikami predykcji, możemy ocenić jakość modelu regresji. Metryki używane do oceny regresorów zostały opisane w rozdziale 9, teraz wywołamy jedynie obliczającą te metryki funkcję `evaluateRegressor`.

```
linear_model_metrics <- evaluateRegressor (observed = linear_model_prediction$RUL,
                                           predicted =
linear_model_prediction$Score)
linear_model_metrics[c(1,2)]
```

```
Mean Absolute Error      Root Mean Squared Error
28.0728306                 34.6138481
```

Średnio nasz model myli się w przewidywaniu pozostałego czasu bezawaryjnej pracy silnika o 28 cykli (jeśli zmierzymy średni błąd absolutny) albo o 34 cykle (jeśli zmierzymy średni błąd kwadratowy). Proszę samodzielnie sprawdzić średnie błędy modeli nauczonych na większej i mniejszej liczbie zmiennych.

Przed wdrożeniem modelu uczenia maszynowego do użycia należy dodatkowo zweryfikować jego poprawność. Można to zrobić, analizując reszty (błędy) modelu. Sposób dodania błędów do tabeli z predykcjami został przedstawiony w rozdziale 3. Reszty modelu regresji liniowej powinny cechować: brak silnych korelacji i stała zmienność (homoskedastyczność). Ponadto, jeżeli do oceny statystycznej istotności parametrów planujemy użyć testów parametrycznych, na

przykład testu wariancji Fishera, rozkład błędów powinien być zbliżony do normalnego. Metody statystycznej analizy danych zostały przedstawione w rozdziale 4.

W tym miejscu należy zwrócić uwagę na częsty błąd polegający na używaniu modeli predykcyjnych do ekstrapolacji. **Predykcja polega na obliczeniu wartości zmiennej wyjściowej dla wartości mieszczących się w zakresie danych treningowych i testowych. Natomiast ekstrapolacja to prognozowanie wartości zmiennej wyjściowej dla wartości spoza zakresu posiadanych danych.** Dlaczego ta różnica jest taka ważna? Bo algorytmy uczenia maszynowego używają danych treningowych do obliczenia następnie minimalizowanego błędu, a standardowe metody oceny modeli używają danych testowych do oceny jego jakości. W rezultacie nie możemy w żaden sposób określić, jak dokładne są wyniki ekstrapolacji. Najlepiej widać to na prostym przykładzie modelu, który na podstawie jednej zmiennej – czasu – prognozuje liczbę mężów. Jeśli przykładowa pani Ola wczoraj miała 0 mężów, a dzisiaj ma jednego, to ekstrapolacja modelu regresji liniowej da pani Oli kilkudziesięciu mężów przed końcem miesiąca.

7.8.3. Ogólny model liniowy GLM

Podstawowym ograniczeniem regresji wielorakiej jest wymóg liniowej korelacji poszczególnych zmiennych wejściowych z wyjściową i wymóg braku silnej korelacji liniowej między zmiennymi wejściowymi. W praktyce spełnienie tych założeń często wymaga odpowiedniego przygotowania danych. Ogólny model liniowy GLM łagodzi oba te wymogi. Uogólnienie uzyskuje się, wiążąc zmienne wejściowe z wyjściową przez funkcję sklejaną (ang. *link function*). Funkcję sklejaną wybiera się na podstawie rozkładu wartości zmiennej wyjściowej.

- Znany nam rozkład Gaussa (rozkład normalny) wymaga użycia funkcji

`gaussian`. Odwrócony rozkład Gaussa wymaga użycia funkcji `inverse.gaussian`.

- Rozkład wykładniczy to rozkład zmiennej ciągłej, w którym im większa wartość, tym mniejsze prawdopodobieństwo jej wystąpienia. Rozkład wykładniczy jest specjalnym przypadkiem rozkładu gamma i wymaga użycia funkcji `Gamma`.
- Rozkład Poissona jest rozkładem zmiennej dyskretnej przyjmującej wiele stanów, z których każdy jest mało prawdopodobny. Do jego zamodelowania używa się funkcji `poisson`.
- Rozkład dwumianowy jest rozkładem zmiennej dyskretnej, której oba stany występują z takim samym prawdopodobieństwem. Wymaga on użycia funkcji `binomial`.
- Pozostałe rozkłady zmiennej wyjściowej należy samodzielnie zamodelować, używając funkcji `rxTweedie`.

Do utworzenia ogólnego modelu liniowego użyjemy funkcji `rxGlm` biblioteki `RevoScaleR`. Wymaga ona podania formuły, nazwy zbioru treningowego i funkcji sklejanej. Pozostałe parametry wywołania funkcji są opcjonalne.

```
GLM_model <- RevoScaleR::rxGlm(formula = formula,
                                    data = train_df,
                                    family = gaussian)
```

```
Rows Read: 20631, Total Rows Processed: 20631, Total Chunk Time: 0.010 seconds

Starting values (iteration 1) time: 0.018 secs.
Rows Read: 20631, Total Rows Processed: 20631, Total Chunk Time: 0.010 seconds

Iteration 2 time: 0.020 secs.

Elapsed computation time: 0.038 secs.
```

Odczytać informacje o nauczonym modelu, które zwraca funkcja `summary`:

```
summary(GLM_model)
```

```
Call:
RevoScaleR::rxGlm(formula = formula, data = train_df, family = gaussian)

Generalized Linear Model Results for: RUL ~ a4 + a11 + a21 + a15 + a20 + a17 + a12 + a7
+ a2 + a3 + s11 + s4 + s12 + s7 + s15
Data: train_df
Dependent variable(s): RUL
Total independent variables: 16
Number of valid observations: 20631
Number of missing observations: 0
Family-link: gaussian-identity

Residual deviance: 42296212.3343 (on 20615 degrees of freedom)

Coefficients:
               Estimate  Std. Error  t value   Pr(>|t|)
(Intercept)   2.642e+04   2.094e+03   12.617   2.22e-16 ***
a4           -1.248e+00   1.845e-01   -6.761   1.40e-11 ***
a11          -1.620e+01   7.200e+00   -2.250   0.024458 *
a21           1.037e+02   1.130e+01    9.181   2.22e-16 ***
a15          -1.769e+02   3.791e+01   -4.666   3.09e-06 ***
a20           3.154e+01   6.674e+00    4.726   2.30e-06 ***
a17          -7.717e+00   6.816e-01  -11.321   2.22e-16 ***
a12          -1.569e+01   2.275e+00   -6.899   5.38e-12 ***
a7           -7.327e+00   1.766e+00   -4.148   3.37e-05 ***
a2           -1.464e+01   2.248e+00   -6.515   7.44e-11 ***
a3           -1.948e+00   1.653e-01  -11.784   2.22e-16 ***
s11          -1.299e+01   3.483e+00   -3.729   0.000193 ***
s4           -2.230e-01   8.817e-02   -2.529   0.011438 *
s12           3.601e+00   1.175e+00    3.064   0.002186 **
s7            2.334e+00   8.711e-01    2.680   0.007369 **
s15          -4.611e+01   1.760e+01   -2.619   0.008826 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for gaussian family taken to be 2051.72)

Condition number of final variance-covariance matrix: 884.2477
Number of iterations: 2
```

Oprócz sposobu wywołania funkcji `rxGlm`, formuły i zbioru uczącego zwraca ona nazwę zmiennej wyjściowej, liczbę zmiennych wejściowych, liczbę przykładów i nazwę użytej funkcji sklejanej.

Zmierzoną zmienność reszt modelu dzieli się przez liczbę stopni swobody. Im większy jest wynik dzielenia, tym większa stabilność modelu. Wyniki powyżej 1 potwierdzają poprawność modelu.

```
GLM_model$deviance/GLM_model$df[2]
```

```
2051.72
```

Ocenę współczynników istotności statystycznej parametrów ułatwia przekształcenie ich do skali zmiennej wyjściowej.

```
as.data.frame(exp(coef(GLM_model)))
```

(Intercept)	Inf
a4	2.872090e-01
a11	9.206297e-08
a21	1.088025e+45
a15	1.487768e-77
a20	5.005649e+13
a17	4.452654e-04
a12	1.526748e-07
a7	6.578574e-04
a2	4.364463e-07
a3	1.425075e-01
s11	2.292460e-06
s4	8.001086e-01
s12	3.662408e+01
s7	1.032382e+01
s15	9.465523e-21

Praca ze wszystkimi modelami uczenia maszynowego bibliotek wygląda tak samo. Zapytania predykcyjne wysyła się do modeli, niezależnie od użytego w nich algorytmu uczenia maszynowego, za pomocą tej samej, wcześniej poznanej funkcji `rxPredict`.

```
GLM_prediction <- rxPredict (modelObject = GLM_model,
                                  data = test_df,
                                  predVarNames = "GLM_Prediction")
```

Na końcu zmierzymy trafność predykcji, obliczając średni błąd bezwzględny modelu i jego błąd średniokwadratowy. Wyniki przypiszemy do zmiennej globalnej `GLM_model_metrics`.

```
GLM_model_metrics <<- evaluateRegressor(observed = test_df$RUL,
                                                predicted = GLM_prediction$GLM_Prediction)
GLM_model_metrics[c(1,2)]
Mean Absolute Error       Root Mean Squared Error
26.2001528                 32.9749786
```

7.8.4. Sztuczne sieci neuronowe

Inspiracją dla sztucznych sieci neuronowych były badania mózgu przeprowadzone w latach 40. XX w. przez dwóch neurobiologów: Warrena McCullocha i Waltera Pitsa. Szczegółowa budowa i działanie sztucznych sieci neuronowych ma jednak niewiele wspólnego z budową i działaniem znacznie bardziej skomplikowanego mózgu. Żeby się o tym przekonać, zaczniemy od krótkiego porównania komórek mózgu z elementami sztucznych sieci neuronowych.

Podstawową komórką nerwową mózgu jest neuron. Jest to komórka zdolna do przetwarzania i przewodzenia informacji w postaci sygnału elektrycznego. Neurony mają około 0,1 mm średnicy i są zbudowane z ciała komórki przesyłającego sygnał aksonu i wielu dendrytów tworzących rozgałęzioną strukturę wyjściową – telodendron.

Neurony przekazują między sobą sygnały za pomocą synaps. Synapsa zawiera substancję chemiczną – neuroprzekaźnik. Gdy do synapsy przez akson dociera impuls, synapsa uwalnia neuroprzekaźnik, który pobudza połączony neuron,

i wywołuje w nim impuls. Jeśli połączenie między neuronami jest słabe, to siła synapsy jest mała. Uwalnia się wtedy za mało neuroprzekaźnika, żeby wywołać impuls w połączonym z nią neuronie.

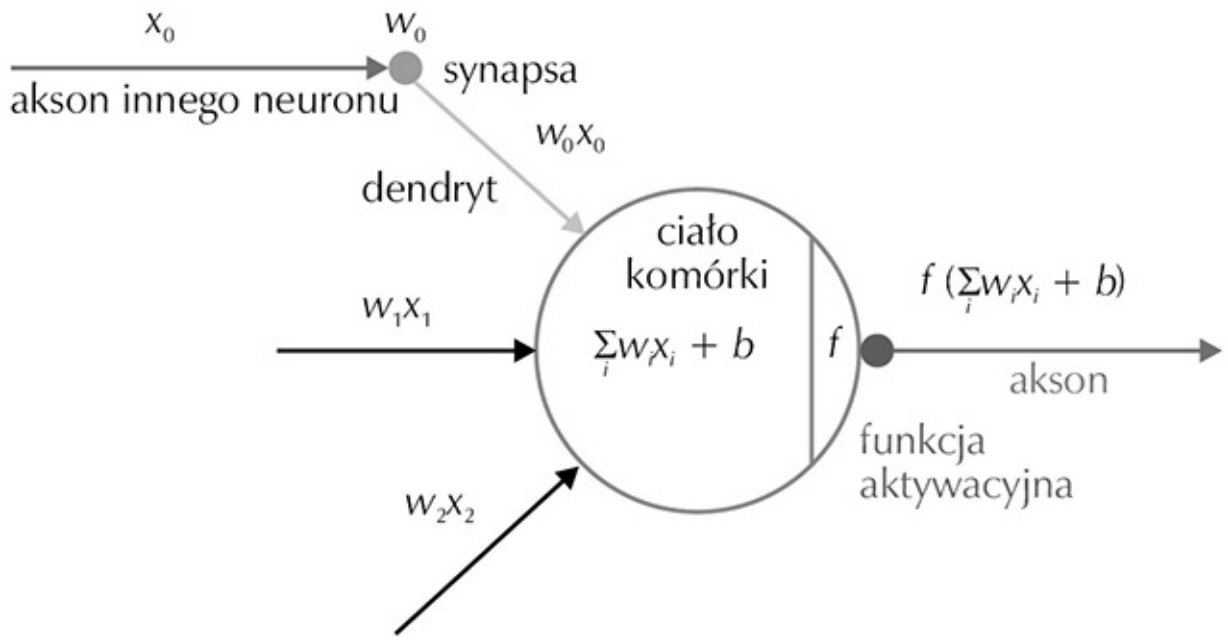
Neuron przechodzi w stan aktywny, gdy łączny sygnał, który dotarł do ciała komórki przez dendryty, przekroczy pewien poziom (próg aktywacji). Ponieważ przesyłanie sygnału wymaga uwolnienia chemicznego neuroprzekaźnika, jest stosunkowo wolne i trwa 1–2 ms. W sztucznych sieciach neuronowych synapsy są reprezentowane przez wagi (parametry), które mogą się zmieniać, co umożliwia uczenie modelu. Dla porównania, zbudowana z tranzystorów bramka logiczna przełącza się w czasie 1 ns, czyli milion razy szybciej od synapsy.

Możliwość błyskawicznego przetwarzania ogromnych ilości informacji docierających do mózgu zawdzięczamy ogromnej liczbie połączonych ze sobą neuronów. W korze mózgowej jest ich około 90 miliardów. Ponieważ rozpoznanie obrazu zajmuje ludziom około 100 ms, cały proces nie może składać się z więcej niż kilkudziesięciu kroków. Skoro wykonanie tak skomplikowanej operacji, jaką jest rozpoznanie obrazu, wymaga wykonania zaledwie kilkudziesięciu etapów, na każdym z nich sygnały muszą być przesyłane między tysiącami neuronów.

Odpowiednikiem neuronu w sztucznej sieci neuronowej jest węzeł. Węzeł, czyli neuron sztucznej sieci neuronowej, jest samodzielną jednostką obliczeniową. Mnoży on zmienne wejściowe przez ich wagi (parametry), a następnie je sumuje. Tak otrzymana wartość jest przekazywana do funkcji aktywacyjnej. Najczęściej stosowane funkcje aktywacyjne zostały opisane poniżej (rys. 7.14).

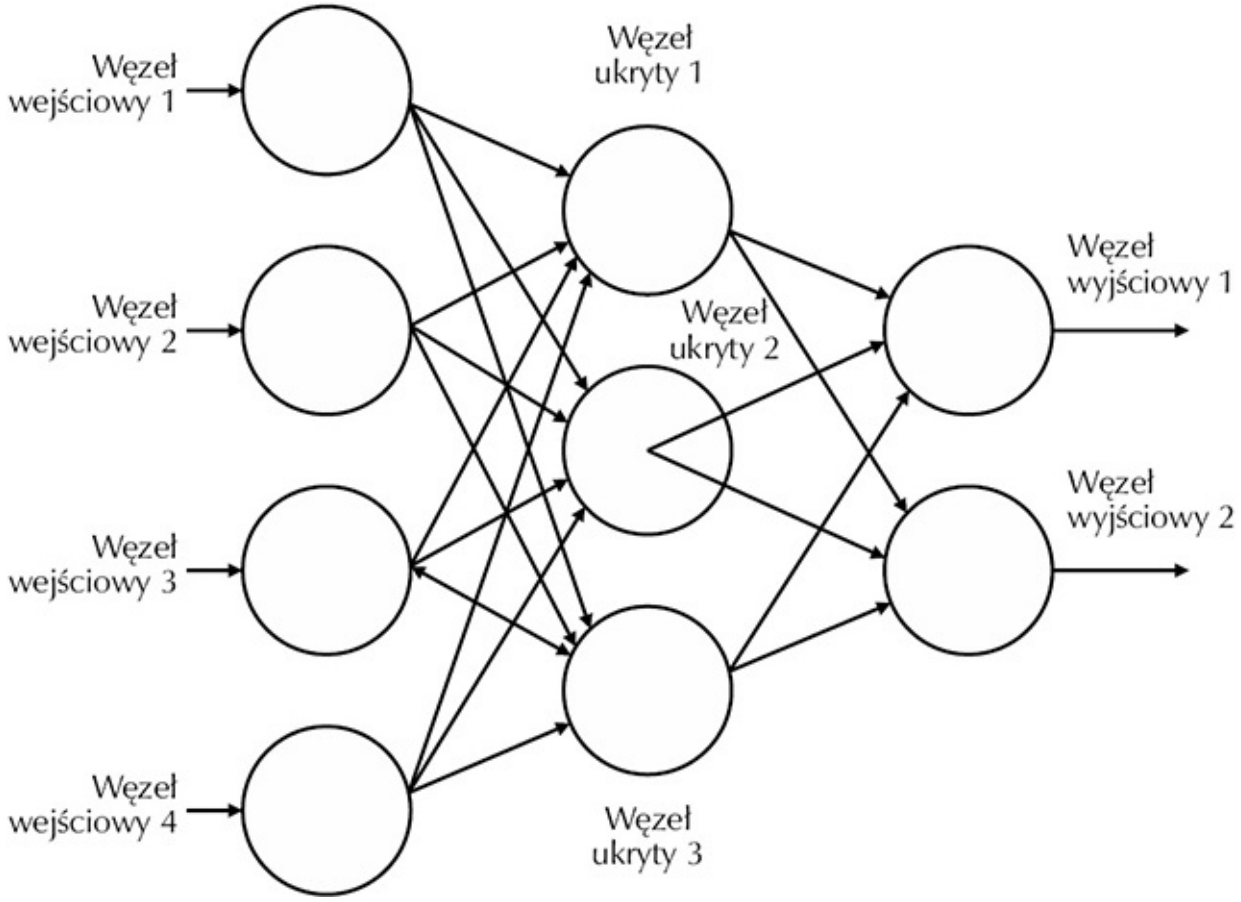
Największe sztuczne sieci neuronowe liczą setki miliardów węzłów, a więc więcej niż jest neuronów w mózgu człowieka. Jednak w mózgu każdy neuron jest połączony z kilkudziesięcioma tysiącami innych, podczas gdy pojedynczy tranzystor jest połączony zaledwie z kilkudziesięcioma innymi tranzystorami. Ponadto w mózgu występuje zjawisko neuroplastyczności – powstają nowe

połączenia między neuronami (synapsy), inne są eliminowane, a topologia sztucznych sieci neuronowych jest stała, definiowana przed rozpoczęciem uczenia.



Rysunek 7.14. Węzeł sztucznej sieci neuronowej jest uproszczonym neuronem mózgu. Jego działanie sprowadza się do policzenia sumy ważonej sygnałów wejściowych i zwróceniu na wyjściu wyniku zastosowanej funkcji aktywacyjnej

W sztucznych sieciach neuronowych węzły są dzielone na warstwy. Węzły w warstwie wejściowej (warstwie zerowej) jedynie reprezentują zmienne wejściowe. Sygnał wyjściowy neuronów z warstwy zerowej trafia na wejścia neuronów pierwszej warstwy. Te węzły obliczają sumę ważoną sygnałów wejściowych i po zastosowaniu funkcji aktywacyjnej zwracają ją na wyjściu. Ponieważ nie wiemy, co reprezentują poszczególne węzły tej warstwy, nazywamy ją warstwą ukrytą. Jeżeli sieć neuronowa zawiera wiele ukrytych warstw, nazywa się ją głęboką. Węzły w ostatniej warstwie reprezentują zmienną wyjściową, jeżeli przy tym zmienna wyjściowa była dyskretna, każdy jej stan będzie reprezentowany przez osobny węzeł. Węzły warstwy wyjściowej również obliczają sumę ważoną sygnałów wejściowych, ale z reguły stosuje się w nich inną funkcje aktywacyjną niż w warstwach ukrytych (rys. 7.15).



Rysunek 7.15. Jednokierunkowa sieć neuronowa z jedną warstwą ukrytą, w której każdy neuron jest połączony ze wszystkimi neuronami z sąsiednich warstw

Ciekawa jest też różnica między zużyciem energii przez mózg a przez superkomputer. Mózg człowieka zużywa dziesięciokrotnie więcej energii (i tlenu) na gram tkanki niż reszta naszego ciała. Do działania wymaga on mniej niż 20 W. Tymczasem Superkomputer IBM Watson do działania wymaga 20 000 W.

Opis sztucznych sieci neuronowych zaczniemy od przedstawienia perceptronu – najprostszego modelu sztucznej sieci neuronowej z jednym węzłem. Następnie zbudujemy sieć neuronową z ukrytą warstwą i metodą wstecznej propagacji błędów nauczymy ją prognozowania pozostałego czasu życia silnika. Na końcu przyjrzymy się głębokim sieciom neuronowym.

Perceptron

Perceptron jest modelem liniowego klasyfikatora binarnego. Oznacza to, że potrafi on znaleźć granicę między dwiema liniowo separowalnymi (dającymi się oddzielić linią prostą, płaszczyzną, a w przestrzeni o większej liczbie wymiarów – hiperpłaszczyzną) klasami przykładów. Znajomość tego modelu ułatwi zrozumienie budowy i działania bardziej skomplikowanych modeli sztucznych sieci neuronowych.

Najprostsza wersja perceptronu zawiera jeden węzeł, nie licząc węzłów wejściowych. W związku z tym, że węzły warstwy wejściowej nie wykonują żadnych obliczeń, z reguły pomija się je w opisie modelu. Jedynym węzłem perceptronu jest neuron McCullocha–Pittsa, dwóch naukowców z Massachusetts Institute of Technology, którzy w 1943 r. opracowali pierwszy matematyczny model neuronu sztucznej sieci neuronowej.

Węzeł ten ma wiele wejść i jedno wyjście. Wartość na wyjściu jest obliczana jako suma ważona wartości wejściowych:

$$s = f(w_0 + \Sigma_{i=1}^{n} x_i w_i)$$

Sygnały wejściowe przyjmują wartości 0 lub 1, a zastosowaną funkcją aktywacyjną *f* jest funkcja progowa. Jeżeli suma ważona sygnałów wejściowych przekracza wartość progową, na wyjściu neuronu pojawia się wartość 1, w przeciwnym przypadku neuron zwraca –1 lub rzadziej 0.

Dodatkowa waga (obciążenie) odgrywa rolę wartości progowej. W tym celu wartość progowa neuronu jest ustawiana na 0, a wartość sygnału (błędu systematycznego) na 1. W wyniku otrzymujemy następujący model matematyczny:

$$o(x_1, \ldots, x_n) = \begin{cases} 1 \text{ gdy } w_0 + x_1 w_1 + \ldots + x_n w_n > 0 \\ -1 \text{ gdy } w_0 + x_1 w_1 + \ldots + x_n w_n \leq 0 \end{cases}$$

Model McCullocha–Pittsa został zaimplementowany przez Franka Rosenblatta w 1957 r. Ponieważ perceptron składa się z jednego neuronu McCullocha–Pittsa, wymagało to jedynie opracowania metody uczenia modelu. Zastosowana metoda ucząca polega na zmianie wag, gdy odpowiedź modelu była nieprawidłowa. Jest to więc prosta metoda uczenia na błędach, nazywana metodą perceptronu:

$$w_i = w_i + \Delta w_i$$

gdzie

$$\Delta w_i = \eta(t - o)x_i$$

Symbol t oznacza prawdziwą wartość zmiennej, o wartość zwróconą przez perceptron, a η (eta) jest współczynnikiem uczenia, którego wartość ustawia się na 0,1.

Zauważmy, że dla prawidłowo sklasyfikowanych przykładów $t = o$, a więc wagi nie są zmieniane. Jeśli model błędnie zaklasyfikował przykład z klasy 1 do klasy –1, różnica $t - o$ daje 2, a błędna klasyfikacja przykładu z klasy –1 do klasy 1 daje różnicę –2.

Uczenie modelu tą metodą przebiega następująco:

- odczytywany i oceniany jest kolejny przykład;
- sprawdzany jest wynik predykcji:
  - jeżeli jest zgodny z prawdziwą wartością zmiennej wyjściowej, wagi połączeń nie są zmieniane;
  - jeżeli jest równy –1, a powinien wynosić 1, wagi zmiennych przykładu są zwiększane;

- jeżeli jest równy 1, a powinien być –1, wagi zmiennych przykładu są zmniejszane.

Procedura ta jest powtarzana dla wszystkich przykładów. Warto zauważyć, że zwiększanie wag powoduje, że prędzej czy później suma ważona wartości wejściowych dla analizowanego przykładu przekroczy próg, a więc przykład zostanie prawidłowo przypisany do klasy 1, natomiast zmniejszanie wag powoduje, że prędzej czy później suma ważona wartości wejściowych dla przykładu spadnie poniżej progu, a więc przykład zostanie prawidłowo przypisany do klasy –1. **Zastosowana metoda uczenia daje gwarancję znalezienia wag, które pozwolą prawidłowo sklasyfikować wszystkie przykłady, o ile jest wystarczająco mała i prawidłowe rozwiązanie istnieje.**

Kiedy istnieje prawidłowe rozwiązanie? Wtedy, gdy przykłady są liniowo separowalne. I tak, nauczenie perceptronu funkcji *AND* (koniunkcji) jest możliwe. Koniunkcja zwraca prawdę, jeśli oba jej czynniki są prawdą. Wystarczy ustawić wartość progową na 0, żeby suma ważona obu zmiennych była większa od zera tylko wtedy, gdy obie one są dodatnie. Podobnie nauczenie perceptronu funkcji *OR* (alternatywy) sprawdza się do ustawienia wartości progowej na 1. Wtedy warunek

$$w_0 + x_1 w_1 + ... + x_n w_n > 0$$

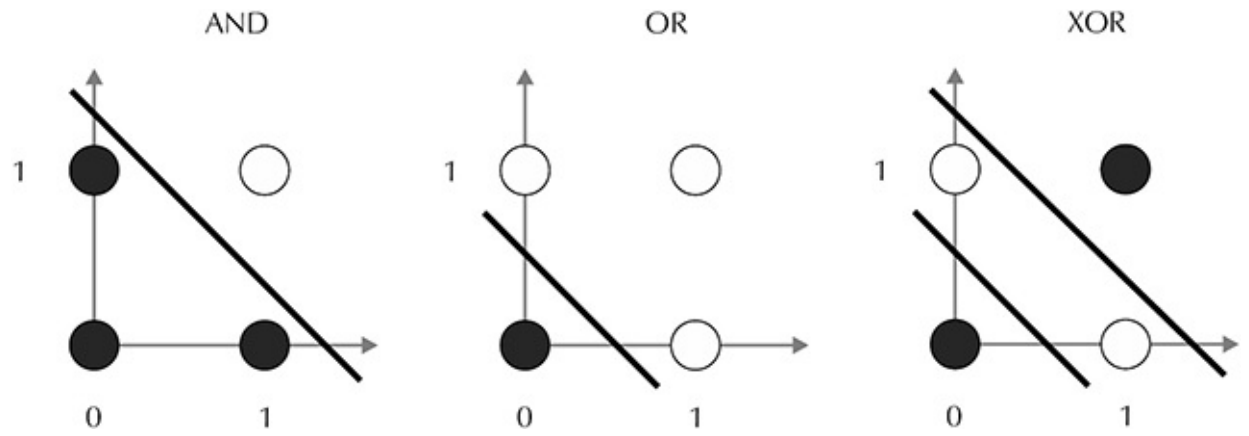
jest nieprawdziwy tylko wtedy, gdy obie zmienne mają wartości ujemne.

Zauważmy jednak, że wyrażenie

$$s = w_0 + x_1 w_1 + ... + x_n w_n$$

jest funkcją liniową. Jeśli nie istnieje pojedyncza linia prosta (a dla większej liczby zmiennych – hiperpłaszczyzna), która oddziela od siebie przykłady obu klas, uczenie perceptronu skończy się porażką. Weźmy funkcję *XOR*. Funkcja ta zwraca *prawdę*, jeżeli wartości czynników są różne, a *fałsz*, jeśli są takie same.

Jeżeli funkcję *XOR* przedstawimy graficznie, okaże się, że nie istnieje taka linia, która prawidłowo podzieliłaby wyniki funkcji, czyli oddzieliła od siebie przykłady należące do różnych klas. Żeby prawidłowo sklasyfikować przykłady, potrzebne są co najmniej dwie takie linie (rys. 7.16).



Rysunek 7.16. Na osiach są umieszczone wartości obu zmiennych wejściowych. Okręgi symbolizują wyniki funkcji AND, OR i XOR – pusty okręg przedstawia prawdę, a pełny fałsz. Łatwo zauważyć, że wyniki funkcji XOR nie są liniowo separowalne

Problem jest poważny, bo ogromna większość rzeczywistych problemów nie jest separowalna liniowo. Można go rozwiązać na dwa sposoby: odpowiednio przygotowując dane albo dodając do perceptronu dodatkowe neurony.

Pierwsze rozwiązanie stawia pod znakiem zapytania użyteczność takich modeli uczenia maszynowego. Skoro w praktyce potrafią one zaleźć tylko rozwiązanie, pod które przygotowane zostały dane treningowe, to ich zastosowanie jest bardzo ograniczone.

Drugie rozwiązanie wydaje się lepsze. Dodając warstwę ukrytą, można rozwiązać nieseparowalne liniowo problemy, bo każdy węzeł w ukrytej warstwie dzieli przestrzeń przykładów na dwie półprzestrzenie. Wyniki tych podziałów są następnie sumowane w węźle wyjściowym. W rezultacie uzyskujemy podprzestrzeń, która jest zbiorem wielościennym wypukłym. Jeśli więc jeden z dwóch podzbiorów przykładów, o wartościach odpowiednio równych 0 i 1, zawiera się w tym zbiorze wypukłym, to może on być przypisany

do odpowiedniej klasy. Pojawia się jednak problem aktualizacji wag łączy ukrytych neuronów.

Ponieważ nie odpowiadają one zmiennym wejściowym, lecz ich kombinacjom, należy je traktować jak zmienne wyliczeniowe. Procedura uczenia perceptronu polega na takim zmienianiu wag, żeby każda zmiana zbliżała je do aktualnie najlepszego zestawu wag. Tymczasem uśrednienie dwóch dobrych rozwiązań często nie prowadzi do znalezienia najlepszego rozwiązania. To ograniczenie perceptronu, opisane przez Marvina Minskego i Seymoura Paperta, bezpośrednio przyczyniło się do pierwszego kryzysu sztucznej inteligencji [32].

W latach 60. i 70. XX wieku wielu badaczy pracowało nad metodą wstecznej propagacji błędów, która pozwoliłaby efektywnie uczyć sztuczne sieci neuronowe z ukrytą warstwą, takie jak złożony perceptron. Metoda ta zaczęła być stosowana w praktyce na szeroką skalę dopiero w latach 80., po tym jak została spopularyzowana przez Geoffreya Hintona w artykule *Learning representations by back-propagating errors* [33]. Poznamy ją w następnym punkcie, a następnie przyjrzymy się różnym funkcjom aktywacyjnym oraz zbudujemy sieć neuronową z ukrytą warstwą i nauczymy ją prognozować pozostały czas bezawaryjnej pracy urządzeń.

Wsteczna propagacja błędów

Wiemy, że w sztucznych sieciach neuronowych występuje problem aktualizacji wag połączeń w ukrytych warstwach. Skoro nie wiemy, co reprezentują ukryte węzły, nie wiemy też, względem jakiej zmiennej należałoby policzyć gradient, który wyznaczyłby kierunek zmian ich wag. Losowa zmiana wag nie gwarantowałaby znalezienia minimum funkcji kosztu, ponadto mała efektywność tej metody jeszcze maleje wraz ze zmniejszeniem się korygowanego błędu. Rozwiązanie tego problemu polega na zmianie wag w ukrytych warstwach sieci przy użyciu metody wstecznej propagacji błędów

(ang. *backpropagation*).

Zakłada ona możliwość policzenia pochodnej funkcji kosztu względem wartości wyjściowych wszystkich węzłów sieci. Błąd węzła wyjściowego jest znany, bo jest to błąd średniokwadratowy modelu. Pochodne funkcji kosztu względem wartości zwróconych przez pozostałe węzły oblicza się, korzystając z reguły łańcuchowej oraz wzoru na pochodną sumy funkcji (pochodna sumy funkcji jest równa sumie ich pochodnych).

Reguła łańcuchowa pozwala obliczyć pochodną częściową złożonej funkcji na podstawie pochodnych jej funkcji bazowych. Weźmy prostą funkcję trzech zmiennych

$$(x+y)*z$$

i przyjmijmy, że $x = -2$, $y = 5$, $z = -4$. Częściowy wynik dodawania q równa się

$$q = x + y = 3$$

a końcowy wynik f wynosi

$$f = q*z = -12$$

W pierwszej kolejności należy policzyć pochodną częściową względem węzła

$$f \frac{\partial f}{\partial f} = 1$$

Ta wartość zostanie przesłana wstecz, do poprzednich węzłów q i z. Pochodna częściowa względem węzła q wynosi

$$\frac{\partial f}{\partial q} = -4$$

czyli jest równa wartości z, a pochodna częściowa względem węzła z wynosi

$$\frac{\partial f}{\partial z} = 3$$

czyli jest równa wartości węzła *q*. Policzenie pochodnych częściowych względem węzłów *x* i *y* wymaga zastosowania reguły łańcuchowej

$$\frac{\partial f}{\partial x} = \circ \frac{\partial f}{\partial q}\frac{\partial q}{\partial x} = -4, \ \frac{\partial f}{\partial y} = \circ \frac{\partial f}{\partial q}\frac{\partial q}{\partial y} = -4$$

Każdy ukryty węzeł sztucznej sieci neuronowej wykonuje dwie operacje: sumuje iloczyny poszczególnych sygnałów wejściowych i odpowiednich wag oraz oblicza dla tej sumy wynik funkcji aktywacyjnej. Reguła łańcuchowa ma zastosowanie do pojedynczych operacji, a więc musimy te operacje rozdzielić. Gradient dla węzła, w którym została zastosowana funkcja sigmoidalna, policzmy jako

$$\frac{\partial E}{\partial w_i} = \Sigma_d (t_d - o_d)\frac{\partial o_d}{\partial net_d}\frac{\partial net_d}{\partial w_i} = \Sigma_d (t_d - o_d) o_d (1 - o_d) x_{i,\,d}$$

czyli na podstawie sumy różnic między prawdziwymi a zwróconymi przez połączone z nim węzły wartości zmiennej wyjściowej oraz pochodnej funkcji sigmoidalnej.

W ten sposób, przekazując błędy z warstwy wyjściowej do ukrytej i z ukrytej do wejściowej, obliczyliśmy pochodne częściowe funkcji kosztu względem wszystkich parametrów modelu, czyli wyznaczyliśmy gradient. Znając gradient, możemy zmienić wartości parametrów w kierunku jego najszybszego spadku.

Podsumowując, uczenie jednokierunkowej, zawierającej jedną ukrytą warstwę sztucznej sieci neuronowej przebiega następująco:

1. Ustalana jest architektura sieci. W najprostszym przypadku w warstwie wejściowej jest tworzonych tyle węzłów, ile jest zmiennych wejściowych. Liczba węzłów w warstwie ukrytej musi zostać podana. Wyjście każdego węzła z warstwy wejściowej jest podłączane do wejść każdego węzła warstwy ukrytej. Warstwa wyjściowa zawiera węzeł reprezentujący zmienną

wyjściową. Do jego wejść są podłączane wyjścia wszystkich neuronów z warstwy ukrytej.

2. Wszystkie wagi (parametry modelu) są inicjowane jako losowe liczby z podanego zakresu.
3. Obliczany jest wynik dla wszystkich (jeżeli uczenie odbywa się metodą *Batch Gradient Descent*) albo wylosowanych (jeżeli została zastosowana metoda *Mini-Batch Gradient Descent* lub *Stochastic Gradient Descent*) przykładów.
4. Obliczany jest błąd bieżącego cyklu. Dla każdego wyjściowego węzła *k* błąd

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

dla każdego węzła ukrytego *h* błąd

$$\delta_h \leftarrow o_h(1 - o_h) \sum_k w_{h,k}\delta_k$$

5. Błąd cyklu jest propagowany do wcześniejszych warstw sieci. Na tej podstawie jest obliczany gradient.
6. Zmieniane są wagi w kierunku najszybszego spadku gradientu.
7. Kroki od 3. do 6. są powtarzane aż do wykonania zadanej liczby cykli, chyba że wcześniej wagi przestaną się zmieniać.

Złożone sieci neuronowe

Zastosowanie sieci neuronowych do regresji wymaga użycia innej procedury uczącej niż metoda perceptronu. Zamiast zmieniać wagi, tak żeby wynik predykcji odpowiadał wartości zmiennej wyjściowej, należy je zmieniać w taki sposób, żeby wynik był jak najbliższy rzeczywistemu, czyli żeby różnica między prawdziwą a przewidywaną wartością zmiennej wyjściowej była jak najmniejsza.

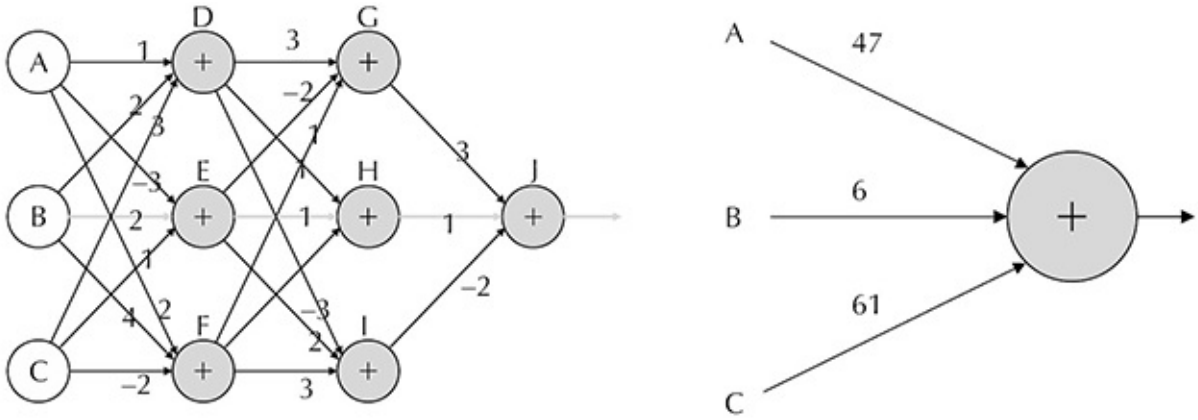
Wiemy, że jedną z funkcji kosztu najczęściej stosowanych do oceny modeli regresji jest błąd średniokwadratowy. Wiemy też, że minimum tej funkcji można znaleźć metodą najszybszego spadku gradientu i że metoda ta polega na obliczaniu gradientu, czyli pochodnej funkcji kosztu względem zmiennych wejściowych. Użyta do aktywacji neuronu McCullocha–Pittsa funkcja progowa nie ma pochodnej w punkcie, w którym zmienna przechodzi z wartości –1 do 1, a w pozostałych punktach jej pochodna wynosi 0. Uczenie sieci neuronowych metodą najszybszego spadku gradientu wymaga więc zastąpienia funkcji progowej inną.

Zobaczymy, czy funkcję progową da się zastąpić liniową. W ten sposób otrzymalibyśmy neuron, którego sygnał wyjściowy jest sumą ważoną sygnałów wejściowych

$$o(x_1, \ldots, x_n) = w_0 + x_1 w_1 + \ldots + x_n w_n$$

Oznacza to, że **użycie liniowej funkcji aktywacyjnej spowodowałoby zredukowanie dowolnie głębokiej sztucznej sieci neuronowej do pojedynczego neuronu**.

Zilustrujmy to przykładem w pełni połączonej sieci o dwóch ukrytych warstwach i trzech zmiennych wejściowych. W każdej warstwie sieci znajdują się trzy neurony, z wyjątkiem warstwy wyjściowej zawierającej jeden neuron. Przyjmijmy dowolne wagi, na przykład wejścia neuronu *D* będą miały wagi 1, 2 i 3, a wejścia neuronu *G*: 3, –2 i 1. Ponieważ każdy neuron zwraca sumę ważoną sygnałów wejściowych, całą sieć można sprowadzić do jednego równania liniowego (rys. 7.17).



Rysunek 7.17. Gdyby działanie neuronu sprowadzało się do dodawania i mnożenia, to operacje wykonywane przez wszystkie neurony sztucznej sieci neuronowej można by sprowadzić do policzenia jednej sumy ważonej

Zredukowanie dowolnie głębokiej sieci do pojedynczego neuronu nie tylko oznacza zmarnowanie mnóstwa czasu na znalezienie optymalnej wartości wielu niepotrzebnych wag, ale przede wszystkim spowoduje, że nasza sieć nie będzie się różniła od modelu regresji liniowej. Po dwóch dekadach badań uznano, że w ukrytych węzłach należy użyć nieliniowej, różniczkowalnej funkcji aktywacyjnej.

Minsky i Papert opisując w 1969 roku ograniczenia perceptronu zauważyli, że dodanie dodatkowych neuronów pozwoliłoby mu rozwiązywać nieliniowe problemy. Zauważyli też, że w sztucznych sieciach neuronowych o wielu warstwach konieczne jest użycie nieliniowej funkcji aktywacyjnej. Niestety, metoda perceptronu nie nadaje się do uczenia sieci złożonych, zbudowanych z więcej niż jednej warstwy. Dlaczego nie zaproponowali oni użycia nieliniowej, różniczkowalnej funkcji aktywacyjnej, skoro pozwoliłaby ona uczyć się metodą najszybszego spadku gradientu? Otóż zastosowanie takiej funkcji aktywacyjnej powoduje, że funkcja kosztu *MSE* zyskuje minima lokalne, a więc ta metoda uczenia nie gwarantowałaby już znalezienia optymalnego rozwiązania.

Trzeba było prawie 20 lat, żeby zaakceptować to ograniczenie. Między innymi dzięki pracom Geoffreya Hintona przekonaliśmy się, że minima lokalne nie są

tak poważnym problem, jak się początkowo wydawało. Po pierwsze, znalezienie jednego z lokalnych minimów często daje wystarczająco dobre rozwiązanie. Po drugie, optymalizacje metody najszybszego spadku gradientu pozwalają, przez zmienianie współczynnika uczenia, unikać zakończenia uczenia w lokalnym minimum.

W płytkich, liczących jedną lub kilka ukrytych warstw sztucznych sieciach neuronowych najpopularniejszymi funkcjami aktywacyjnymi są funkcja sigmoidalna

$$\text{sigmoid} = \frac{1}{1 + e^{-z}}$$

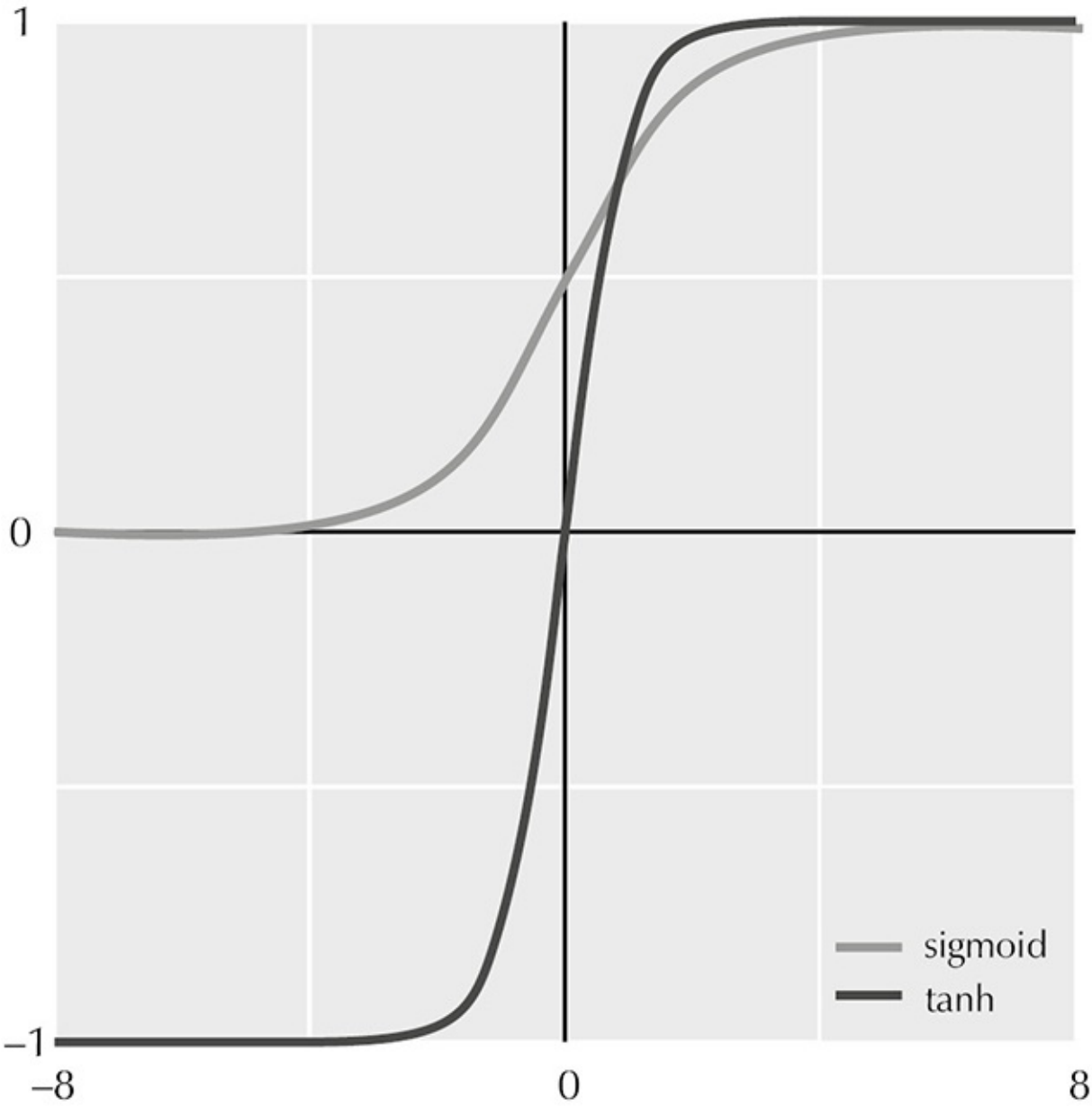
oraz tangens hiperboliczny. Funkcja sigmoidalna, gdzie

$$z = w_0 + x_1 w_1 + \ldots + x_n w_n$$

daje wyniki w zakresie od 0 do 1, tangens hiperboliczny

$$\frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

zwraca wartości w zakresie od –1 do 1. Obie te funkcje są prawie liniowe dla wartości bliskich zeru, a ich nieliniowość rośnie wraz z oddalaniem się wartości od zera (rys. 7.18).



Rysunek 7.18. Funkcja sigmoidalna jest używana w warstwie wyjściowej, w warstwach ukrytych używa się obu funkcji

W głębokich, liczących kilkadziesiąt czy kilkaset warstw sztucznych sieciach neuronowych najczęściej stosowaną funkcją aktywacyjną jest funkcja *ReLU* (*Rectified Linear Unit*). *ReLU* jest połączeniem funkcji progowej z liniową i na wyjściu zwraca 0 lub sygnał wejściowy, o ile przekroczył on wartość progową

$$o(x_1, \ldots, x_n) = \begin{cases} w_0 + x_1 w_1 + \ldots + x_n w_n & \text{gdy} \quad w_0 + x_1 w_1 + \ldots + x_n w_n > 0 \\ & \text{gdy} \quad w_0 + x_1 w_1 + \ldots + x_n w_n \leq 0 \end{cases}$$

Funkcja ta jest tak prosta, że jej wielokrotne obliczanie ma niewielki wpływ na czas uczenia sztucznej sieci neuronowej, a jej liniowość w zakresie dodatnich wartości ogranicza zjawisko zanikania gradientu.

Do zbudowania sztucznej sieci neuronowej użyjemy funkcji `rxNeuralNet` biblioteki `MicrosoftML`. Jej wymaganymi parametrami są formuła ucząca, zbiór danych treningowych i typ modelu. Funkcja ta pozwala dodatkowo określić wiele hiperparametrów modelu, w tym zdefiniować architekturę sztucznej sieci neuronowej i wybrać metodę optymalizacji procesu uczenia. Samo uczenie może się odbywać z wykorzystaniem procesorów graficznych ogólnego zastosowania GPU, o ile serwer dysponuje kartą graficzną zgodną z technologią NVidia CUDA Toolkit 6.5 i biblioteką NVidia cuDNN v2. Pierwszy model nauczymy metodą najszybszego spadku gradientu.

```
nn_model <- MicrosoftML::rxNeuralNet(formula,
                                     data = train_df,
                                     type = "regression",
                                     normalize = "warn",
                                     numHiddenNodes = 20,
 numIterations = 200,
 initWtsDiameter = 0.1,
 acceleration = "gpu",
                                     optimizer = sgd(learningRate = 0.01, momentum = 0.8, nag =
TRUE,
                                                     lRateRedRatio = 0.7, lRateRedFreq = 10)
                                    )
```

```
Warning: A normalizer is needed for this trainer. Either add a normalizing transform or
use the 'norm=Auto', 'norm=Yes' or 'norm=No' options.
Beginning processing data.
Rows Read: 20631, Read Time: 0, Transform Time: 0
…
***** Net definition *****
  input Data [15];
  hidden H [20] sigmoid { // Depth 1
    from Data all;
  }
  output Result [1] linear { // Depth 0
    from H all;
  }
***** End net definition *****
Input count: 15
Output count: 1
Output Function: Linear
Loss Function: SquaredLoss
PreTrainer: NoPreTrainer
_____________________________________________________________________
Starting training...
Learning rate: 0.010000
Momentum: 0.800000
InitWtsDiameter: 0.100000
_____________________________________________________________________
Initializing 1 Hidden Layers, 341 Weights...
Estimated Pre-training MeanError = 16371.326165
Iter:1/200, MeanErr=7008.647255(-57.19%%), 126.31M WeightUpdates/sec
Iter:2/200, MeanErr=6963.893286(-0.64%%), 125.25M WeightUpdates/sec
…
Iter:190/200, MeanErr=4748.750446(-0.01%%), 92.23M WeightUpdates/sec
Learning rate adapted to: 0.000011
Iter:191/200, MeanErr=4748.325130(-0.01%%), 74.22M WeightUpdates/sec
…
Iter:200/200, MeanErr=4748.378510(0.01%%), 95.89M WeightUpdates/sec
Done!
Estimated Post-training MeanError = 4746.926570
```

Ustawiając parametr `normalize = „warn”`, wyłączyliśmy automatyczną normalizację zmiennych. Sztuczna sieć neuronowa jest zbudowana z trzech warstw: warstwa wejściowa zawiera 15 węzłów, z których każdy reprezentuje jedną zmienną wejściową; w warstwie ukrytej znajduje się 20 węzłów, z których każdy jest połączony ze wszystkimi węzłami warstwy wejściowej; w tej warstwie funkcją aktywacyjną jest funkcja sigmoidalna; warstwa wyjściowa zawiera 1 węzeł reprezentujący zmienną wyjściową `RUL`; węzeł wyjściowy jest węzłem liniowym; funkcją kosztu jest funkcja błędu średniokwadratowego.

Po zbudowaniu sztucznej sieci neuronowej rozpoczęło się jej uczenie. Liczba cykli została ograniczona do 200. Przerwanie uczenia po określonej liczbie cykli jest popularnym sposobem zapobiegania nadmiernemu dopasowaniu modelu. Parametr `initWtsDiameter` ustawił początkowe wartości wag na losowe liczby z zakresu od –0,1 do 0,1. Wybraną metodą optymalizacji procesu uczenia była metoda najszybszego spadku gradientu. Początkowa wartość współczynnika uczenia została ustawiona na 0,01. Wartość ta była zmniejszana do 70% aktualnej co 10 cykli. Wartość pędu Nesterova została ustawiona na 0,8.

Najważniejsze informacje o wytrenowanym modelu, w tym parametry wywołania funkcji `rxNeuralNet`, formułę uczącą, nazwę zbioru treningowego, opis architektury sztucznej sieci neuronowej oraz znalezione wartości wag, zwraca funkcja `summary`.

```
summary(nn_model)
```

Po ocenieniu przez model danych testowych i policzeniu błędu rzeczywistego modelu możemy się przekonać, że sztuczna sieć neuronowa wypadła znacznie gorzej od obu modeli regresji.

```
nn_model_metrics[c(1,2)]
```

```
Mean Absolute Error      Root Mean Squared Error
41.56025                   53.62255
```

Żeby sprawdzić, czy tak duży błąd testowy nie jest efektem nadmiernego dopasowania modelu, należy sprawdzić jego błąd treningowy. W tym celu należy ocenić przy użyciu modelu zbiór treningowy i obliczyć błąd średniokwadratowy i średni błąd bezwzględny predykcji.

```
nn_model_prediction <- rxPredict(modelObject = nn_model,
                                 data = train_df,
                                 extraVarsToWrite = "RUL")

nn_model_metrics <- evaluateRegressor (observed = nn_model_prediction$RUL,
                                       predicted = nn_model_prediction$Score)
nn_model_metrics[c(1,2)]
```

```
Mean Absolute Error Root Mean Squared Error
57.12941                          68.99098
```

W tym przypadku błąd empiryczny jest większy od rzeczywistego. Sprawdźmy, czy zmiana metody optymalizacji procesu uczenia nie pomoże modelowi nauczyć się wzorców ukrytych w danych treningowych.

```
nn_model <- MicrosoftML::rxNeuralNet(formula,
                                     data = train_df,
                                     type = "regression",
                                     normalize = "warn",
                                     numHiddenNodes = 15,
                                     numIterations = 20,
                                     initWtsDiameter = 0.1,
                                     optimizer = adaDeltaSgd()
                                     )
```

```
…
Starting training...
Using Adadelta with decay 0.950000 and conditioner 0.000001
InitWtsDiameter: 0.100000

_____________________________________________________________________
Initializing 1 Hidden Layers, 256 Weights...
Estimated Pre-training MeanError = 16365.744267
Iter:1/20, MeanErr=5849.158586(-64.26%%), 130.09M WeightUpdates/sec
Iter:2/20, MeanErr=4747.245123(-18.84%%), 131.39M WeightUpdates/sec
…

Iter:20/20, MeanErr=4746.914485(-0.01%%), 126.51M WeightUpdates/sec
Done!
Estimated Post-training MeanError = 4745.648989
```

Tym razem ograniczyliśmy liczbę cykli do 20 – i tak zostawiając ją zbyt dużą, bo po kilku pierwszych cyklach błąd empiryczny już się nie zmniejszał. Współczynnik uczenia był dynamicznie obliczany na podstawie zmian błędu empirycznego przez algorytm Adadelta. Niestety, błąd rzeczywisty modelu jest jeszcze większy niż poprzedni i wynosi 68,88.

Jednokierunkowa, jednowarstwowa sztuczna sieć neuronowa (sieć, w której sygnały są przekazywane w jedną stronę – od neuronów wejściowych do ukrytych i od ukrytych do wyjściowych) jest uniwersalnym aproksymatorem dowolnej ograniczonej funkcji ciągłej. Innymi słowy, każda funkcja ciągła, która zwraca wartości z ograniczonego przedziału, może być z dowolnie dobrym przybliżeniem znaleziona przez jednowarstwową sztuczną sieć neuronową zawierającą skończoną liczbę ukrytych węzłów. Istnieje

matematyczny dowód tego twierdzenia, my jednak ograniczymy się do przedstawienia prostej intuicji. Proszę przypomnieć sobie kształt najpopularniejszych funkcji aktywacyjnych: funkcji sigmoidalnej i tangensa hiperbolicznego. A teraz zauważyć, że wykres dowolnej funkcji ciągłej można skonstruować, składając zwracane przez te funkcje *S*-kształtne krzywe.

Zwiększając liczbę węzłów, zwiększamy złożoność modelu, co prowadzi do lepszego przybliżenia szukanej funkcji, przynajmniej dla danych treningowych. Dlaczego więc błąd naszego modelu sztucznej sieci neuronowej jest dwukrotnie większy niż modelu regresji? Sprawdźmy założenia i architekturę modelu.

Wartości zmiennej `RUL` są ciągłe. Zresztą nawet gdyby nie były, to liniowa aproksymacja brakujących wartości prawie zawsze jest wystarczająco dobrym przybliżeniem. Proszę samemu sprawdzić, że w tym przypadku błąd treningowy modelu sieci z jednym ukrytym węzłem jest bardzo zbliżony do błędu sieci o 100 ukrytych węzłach.

Liczba ukrytych węzłów jest ważnym, ale nie jedynym hiperparametrem definiującym architekturę sztucznej sieci neuronowej. Duży wpływ na wynik uczenia mają też początkowe wartości parametrów (wagi węzłów).

Po pierwsze, muszą być one różne. Jeżeli początkowe wartości wag wszystkich węzłów będą takie same, to wszystkie będą tak samo modyfikowane, czyli po zakończeniu uczenia wagi wszystkich węzłów będą identyczne. W rezultacie dowolnie złożona sztuczna sieć neuronowa nauczy się dokładnie tyle samo, co sieć zbudowana z pojedynczego neuronu.

Po drugie, początkowe wagi muszą być małe. Przypomnijmy sobie kształty używanych funkcji aktywacji. Obie te funkcje, funkcja sigmoidalna i tangens hiperboliczny, są liniowe dla wartości bliskich zeru. Natomiast małe i duże wartości są spłaszczane do wartości bliskich –1 (0 w przypadku funkcji sigmoidalnej) i 1. Wiemy, że im mniejsza zmiana wartości funkcji, tym mniejsza

jej pochodna w tym punkcie – czyli tym mniejszy gradient względem danego parametru i tym mniejsze – w skrajnych przypadkach zerowe – postępy w uczeniu.

Po trzecie, duże końcowe wagi często świadczą o nadmiernym dopasowaniu modelu. Karanie dużych wag jest jedną z technik zapobiegania nadmiernemu dopasowaniu sztucznych sieci neuronowych. Efekt ten można osiągnąć, ustawiając początkowe wagi na małe, losowe wartości i ograniczając liczbę cykli uczenia.

Proszę samemu sprawdzić, jak zmniejszenie i zwiększenie wartości parametru `initWtsDiameter` wpływa na błąd treningowy modelu. Jednak w tym przypadku zmiana początkowych wag nie rozwiązała problemu. Może potrzebna jest zmiana architektury sieci?

Liczba ukrytych węzłów sieci powinna zależeć od liczby zmiennych wejściowych oraz przykładów treningowych. Z reguły do znalezienia wystarczająco dobrego przybliżenia funkcji wystarcza liczba ukrytych węzłów równa pierwiastkowi kwadratowemu liczby zmiennych wejściowych. Ustalając liczbę ukrytych węzłów, należy też wziąć pod uwagę liczbę przykładów. Im więcej przykładów, tym więcej ukrytych węzłów możemy dodać do sztucznej sieci neuronowej. To jedynie ogólne wskazówki, przydatne w ustaleniu optymalnej liczby ukrytych węzłów sieci.

Sztuczna sieć neuronowa może też ukrywać więcej niż jedną warstwę. W 99% zastosowań klasycznych metod uczenia maszynowego dodatkowe ukryte warstwy są niepotrzebne, a nawet szkodliwe, bo wydłużają czas uczenia i zwiększają ryzyko nadmiernego dopasowania modelu. Jednak znalezienie dokładnego przybliżenia niektórych hipotez (szukanych funkcji zmiennej wyjściowej od zmiennych wejściowych) wymaga dodania do sztucznej sieci neuronowej dodatkowych warstw.

Czemu podzielenie węzłów między dodatkowe warstwy pomaga, skoro sieć z jedną ukrytą warstwą jest uniwersalnym aproksymatorem ograniczonych, ciągłych funkcji? Otóż znalezienie wystarczająco dokładnego przybliżenia niektórych funkcji wymaga liczby ukrytych węzłów rosnącej wykładniczo wraz z liczbą zmiennych wejściowych. Jeżeli jednak węzły te zostaną podzielone między ukryte warstwy, to liczba węzłów potrzebnych do znalezienia tak samo dokładnego przybliżenia funkcji będzie liniowo zależna od liczby zmiennych wejściowych [44].

Biblioteka `MicrosoftML` pozwala konstruować sztuczne sieci neuronowe w języku Net#. Pozwala on m.in. definiować poszczególne warstwy sztucznej sieci neuronowej. Dla każdej warstwy należy określić: jej typ, nazwę, liczbę węzłów, funkcję aktywacyjną, warstwę poprzedzającą oraz sposób łączenia węzłów z tej warstwy z węzłami warstwy poprzedzającej. Na przykład poniższa instrukcja definiuje sieć o dwóch ukrytych warstwach: pierwszej z pięcioma węzłami, w których została użyta funkcja tangensa hiperbolicznego, i drugiej z dwoma węzłami, których funkcją aktywacyjną jest funkcja sigmoidalna. Sieć jest w pełni połączona, czyli wszystkie węzły kolejnej warstwy są połączone ze wszystkimi węzłami warstwy poprzedniej.

```
net_definition <- ("input Data auto;
                    hidden Hidden1 [5] tanh from Data all;
                    hidden Hidden2 [2] sigmoid from Hidden1 all;
                    output Result auto from Hidden2 all;")
```

Niestety, po nauczeniu tak zdefiniowanej sztucznej sieci neuronowej będziemy mogli się przekonać, że błąd empiryczny modelu niewiele się zmienił i nadal przekracza 50.

```
nn_model <- MicrosoftML::rxNeuralNet(formula,
                                     data = train_df,
                                     type = "regression",
                                     normalize = "warn",
                                     initWtsDiameter = 0.01,
                                     netDefinition = net_definition,
                                     numIterations = 10,
                                     optimizer = adaDeltaSgd() )
```

Proszę samodzielnie sprawdzić, czy zmiana parametrów metody najszybszego spadku gradientu znacząco wpływa na poprawę wyników.

Skoro problem nie leży po stronie budowy sieci, funkcji kosztu ani metody uczenia (trzech z czterech składników każdego modelu uczenia maszynowego), to w takim razie winne są dane (czwarty i ostatni składnik modelu uczenia maszynowego).

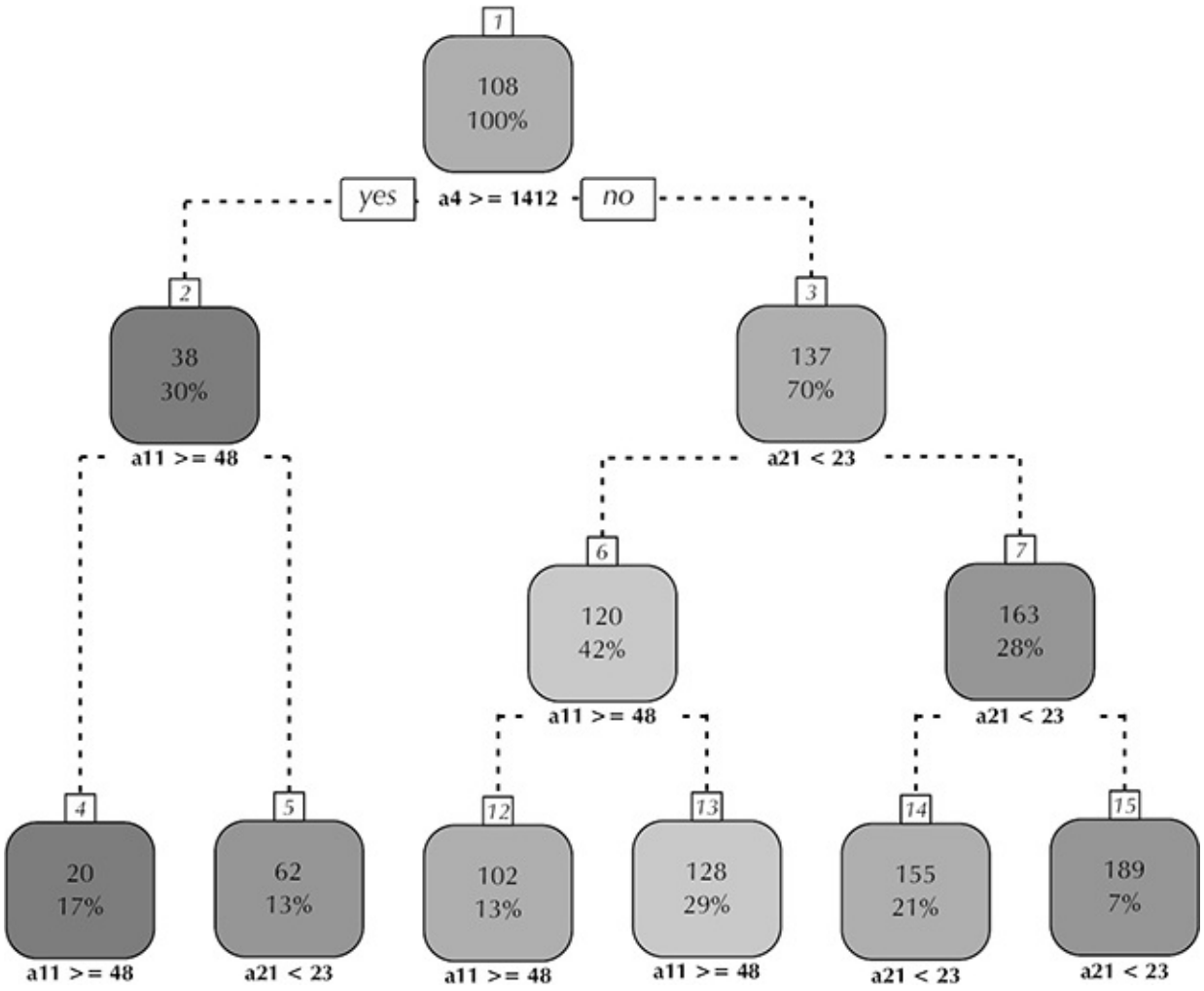
Po pierwsze, zmienne wejściowe wybraliśmy na podstawie współczynnika ich korelacji liniowej ze zmienną wyjściową. Sztuczne sieci neuronowe potrafią, dzięki ukrytym węzłom, uczyć się nieliniowych zależności między zmiennymi. Każdy ukryty węzeł reprezentuje inną kombinację zmiennych wejściowych. Może więc powinniśmy uwzględnić więcej zmiennych? Tym bardziej, że sztuczne sieci neuronowe są bardziej odporne na szum niż modele regresji. Pewna ilość szumu wręcz zapobiega przerwaniu uczenia w punkcie lokalnego minimum funkcji kosztu.

Po drugie, model uczy się, obliczając sumy ważone zmiennych, gdzie wagami są znalezione wartości ich parametrów. Czyli zmienne o większych wartościach mają większy wpływ na wynik ich dodawania. W skrajnym przypadku jedna zmienna o dużej skali może zdominować pozostałe. A ponieważ duże wartości są spłaszczane przez opisane funkcje aktywacyjne do wartości bliskich –1 lub 0 oraz 1, takie zmienne mogą zwolnić, a nawet przedwcześnie zatrzymać proces uczenia.

Problematyczne są też różnice w skali zmiennej wyjściowej i wejściowych. Modele regresji zwracają obliczoną wartość zmiennej wyjściowej, a nie nadany im stan (np. prawdę lub fałsz). Błąd modelu jest liczony jako średnia suma kwadratów błędów, czyli na podstawie rzeczywistych wartości zmiennej wyjściowej. Dalej zobaczymy, jak przygotowanie danych wpływa na jakość modeli, w tym sztucznych sieci neuronowych.

7.8.5. Drzewa regresyjne

Poznane w rozdziale 3 drzewa decyzyjne są stosowane nie tylko do klasyfikacji, ale również do regresji. W najprostszym, dziś już rzadko używanym przypadku, możemy zbudować drzewo decyzyjne, którego liście zawierają różne wartości ciągłej zmiennej wyjściowej, a prowadzące do nich węzły decyzyjne zawierają warunki typu `zmienna x jest większa lub mniejsza od znalezionej wartości brzegowej` (rys. 7.19).



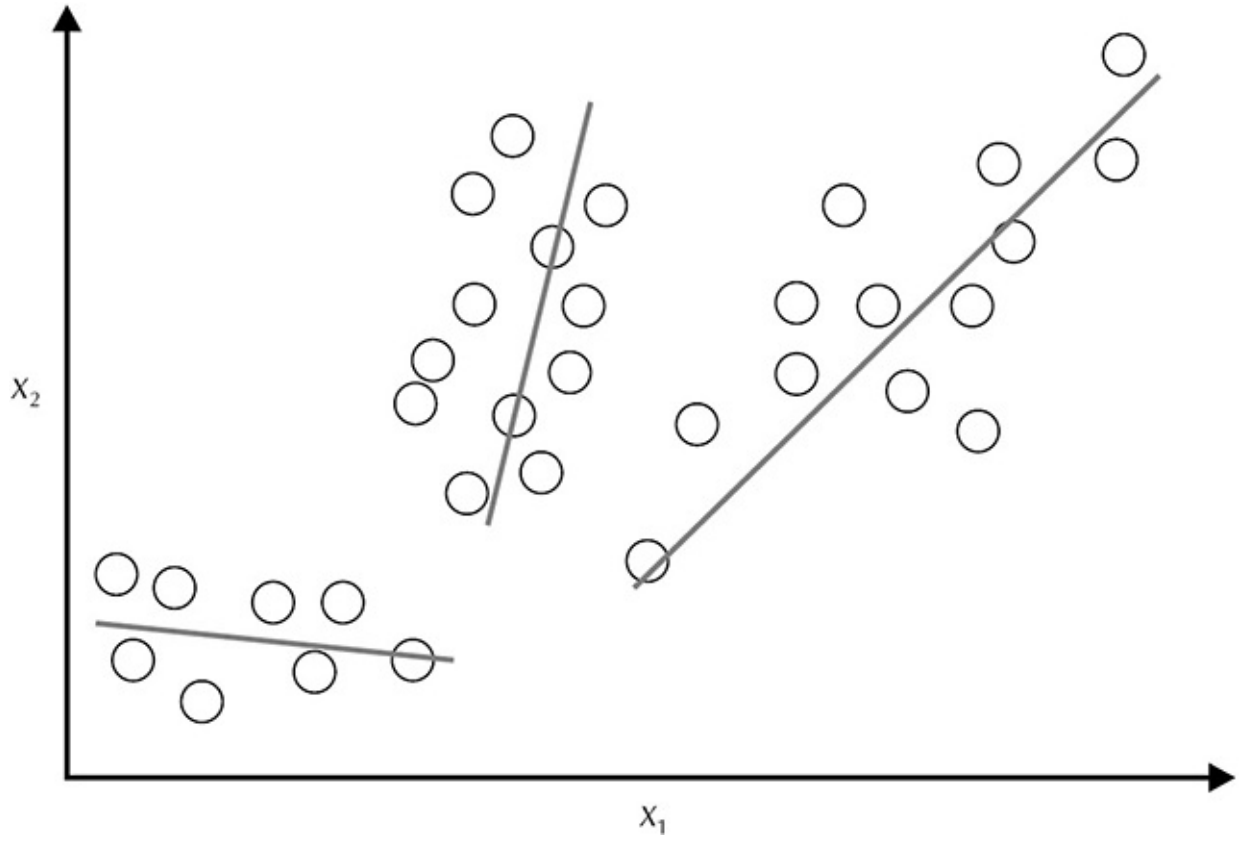
Rysunek 7.19. Drzewo regresyjne przewidujące pozostały czas bezawaryjnej pracy urządzenia na podstawie wartości zmiennych wejściowych

Liście drzew regresyjnych mogą też zawierać osobne, znalezione dla wybranego podzbioru danych treningowych, formuły regresji liniowej (rys. 7.20).

Pojedyncze drzewa decyzyjne są proste w interpretacji i można je łatwo przedstawić graficznie. Niestety, cechuje je duża niestabilność predykcji związana z ich podatnością na nadmierne dopasowanie do danych treningowych.

Wad tych są pozbawione kombinacje drzew decyzyjnych. Metody łączenia (kombinowania) modeli zostały przedstawione w następnym rozdziale. W tym punkcie użyjemy jedynie, dla kompletności wykładu, dwóch złożonych algorytmów drzew decyzyjnych biblioteki `MicrosoftML` do regresji. Ich działanie

polega na budowaniu wielu drzew regresyjnych.



Rysunek 7.20. Drzewa regresyjne zawierają wiele – znalezionych dla podzbiorów danych – formuł regresji liniowej. Pozwala to modelowi rozwiązywać problemy nieliniowe

Pierwszy algorytm, las losowy, jednocześnie trenuje wiele wersji drzew regresyjnych. Każde drzewo jest trenowane na losowo wybranych przykładach i oceniane na pozostałych przykładach, niewybranych do treningu. Wyniki zwrócone przez poszczególne drzewa regresyjne są scalane w taki sposób, żeby uzyskać predykcje o rozkładzie jak najbardziej zbliżonym do rozkładu normalnego.

Las losowy został zaimplementowany w funkcji `rxFastForest`. W tym punkcie przyjrzymy się tylko wybranym parametrom tej funkcji, odkładając opis pozostałych do momentu wyjaśnienia metod konstruowania złożonych modeli uczenia maszynowego.

```
random_forest_model <- MicrosoftML::rxFastForest(formula,
                            data = train_df,
                            type = "regression",
                            numTrees = 20,
                            numLeaves = 20,
                            minSplit = 5,
                            numBins = 20,
                            exampleFraction = 0.7,
                            featureFraction = 0.8,
                            splitFraction = 0.7)
```

Obowiązkowymi parametrami funkcji są formuła ucząca i nazwa zbioru treningowego. Domyślnie budowany jest model klasyfikatora binarnego; żeby zbudować model regresji, należy ustawić parametr `type = „regression”`. Zmienne wyjściowe są dyskretyzowane. Liczbę przedziałów, na jakie zostaną podzielone ich wartości, kontroluje parametr `numBins`. Liczbę drzew modelu ustawia się parametrem `numTrees`. Średniokwadratowy błąd rzeczywisty modelu wyniósł 32,69, czyli był zbliżony do błędów modeli regresji wielorakiej.

Drugi z przedstawionych algorytmów drzew regresyjnych – wzmocnione drzewa decyzyjne – implementuje algorytm dodawania gradientu *MART* (*Multiple Additive Regression Trees*), szerzej znanego jako maszyna wzmacniania gradientu *GBM*. Gradient pierwszego drzewa regresyjnego jest mnożony przez współczynnik uczenia i dodawany do następnego drzewa. W efekcie kolejne drzewa uczą się na błędach poprzednich. Algorytm ten został zaimplementowany w funkcji `rxFastTrees`.

```
gradient_boosting_forest_model <- MicrosoftML::rxFastTrees(formula,
                                          data = train_df,
                                          type = "regression",
                                          numTrees = 20,
                                          numLeaves = 20,
                                          minSplit = 5,
                                          numBins = 20,
                                          exampleFraction = 0.7,
                                          featureFraction = 0.8,
                                          splitFraction = 0.8,
                                          learningRate = 0.1)
```

```
Warning: The number of threads specified in trainer arguments is larger than the
concurrency factor setting of the environment. Using 2 training threads instead.
Not adding a normalizer.
Making per-feature arrays
Changing data from row-wise to column-wise
Beginning processing data.
Rows Read: 20631, Read Time: 0, Transform Time: 0
Beginning processing data.
Processed 20631 instances
Binning and forming Feature objects
Reserved memory for tree learner: 51480 bytes
Starting to train ...
```

Obowiązkowe parametry funkcji są takie same jak funkcji `rxFastForest`. Również domyślnie jest budowany model klasyfikatora binarnego, czyli w tym przypadku należy ustawić parametr `type = „regression”`. Współczynnik uczenia `learningRate` określa wielkość zmiany parametrów w kierunku najszybszego spadku gradientu.

Najważniejsze informacje o nauczonym modelu, w tym przydatność poszczególnych zmiennych wejściowych, zwraca funkcja `summary`.

```
summary(gradient_boosting_forest_model)
```

```
Call:
MicrosoftML::rxFastTrees(formula = formula, data = train_df,
     type = "regression", numTrees = 20, numLeaves = 20, learningRate = 0.1,
     minSplit = 5, exampleFraction = 0.7, featureFraction = 0.8,
     splitFraction = 0.8, numBins = 20)

Predictor for: RUL~a4+a11+a21+a15+a20+a17+a12+a7+a2+a3+s11+s4+s12+s7+s15
Data: train_df

Per-feature gain:
a4: 1
a11: 0.6881242
a17: 0.6232437
a21: 0.5522313
a15: 0.3249542
a20: 0.3012566
a3: 0.2645946
a2: 0.2340451
a7: 0.1489681
a12: 0.1105543
s4: 0.05010933
s15: 0.04834088
s11: 0.03149381
s7: 0.02250068
s12: 0.01777516
```

Sprawdźmy błąd rzeczywisty modelu.

```
gb_model_prediction <- rxPredict(modelObject = gradient_boosting_forest_model,
                                 data = test_df,
                                 extraVarsToWrite = "RUL",
                                 overwrite = TRUE)
gb_model_metrics <- evaluateRegressor (observed = GB_model_prediction$RUL,
                                       predicted = GB_model_prediction$Score)
gb_model_metrics[c(1,2)]
```

```
Mean Absolute Error Root Mean Squared Error
17.73932              24.04796
```

Oto dlaczego opracowany przez Jerome Friedmana algorytm dodawania

gradientu MART [34] oraz będący jego modyfikacją algorytm XGBoost (*Extreme Gradient Boosting Trees*) [35] są tak często stosowane.

7.8.6. Kombinacje modeli

Skoro połączenie drzew regresyjnych w jeden model poprawiło jakość modelu, to warto sprawdzić, czy kombinacja różnych modeli, na przykład wzmocnionych drzew regresji z lasem losowym i regresją liniową, nie da jeszcze lepszych rezultatów.

Istnieje wiele różnych metod łączenia modeli, najprostsza z nich polega na zbudowaniu wielu niezależnych modeli i uśrednieniu wyników ich predykcji. Zwykle poszczególne modele są trenowane z wylosowanych z danych treningowych przykładów, a oceniane przy użyciu pozostałych, niewylosowanych przykładów. Żeby zapewnić równoliczność wszystkich prób, przykłady są losowane ze zwracaniem, tj. każdy przykład może zostać wielokrotnie wylosowany do tej samej próby. Więcej informacji na temat metody agregacji bootstrapowej można znaleźć w punkcie Lasy losowe, w rozdziale 8.

Funkcja `rxEnsemble` pozwala zbudować i wytrenować kombinację modeli predykcyjnych. Trzy pierwsze parametry tej funkcji, `formula`, `data` i `type`, nie różnią się od analogicznych parametrów funkcji już poznanych. Listę modeli określa parametr `trainers`. Pozwala on przekazać nazwy łączonych modeli oraz ich hiperparametry. Proszę zwrócić uwagę, że łączone mogą być tylko modele zbudowane przy użyciu funkcji biblioteki `MicrosoftML` oraz że nazwy modeli różnią się od nazw wcześniej poznanych funkcji. Las losowy tworzy się za pomocą funkcji `fastForest`, a nie `rxFastForest`, wzmocnione drzewa regresji – funkcji `fastTrees`, a nie `rxFastTrees`, itd.

```
ensemble_model <- rxEnsemble(
  formula = formula,
  data = train_df,
  type = "regression",
  trainers = list(
    fastForest(numTrees = 20,
               numLeaves = 20,
               minSplit = 5,
               numBins = 20,
               exampleFraction = 0.7,
               featureFraction = 0.8,
               splitFraction = 0.7),
    fastTrees(numTrees = 20,
              numLeaves = 20,
              minSplit = 5,
              numBins = 20,
              exampleFraction = 0.7,
              featureFraction = 0.8,
              splitFraction = 0.8,
              learningRate = 0.1),
    fastLinear(normalize = "auto")),
  replace = TRUE,
  combineMethod = "average")
```

Funkcja `summary` zwraca najważniejsze informacje o wszystkich modelach bazowych, czyli partycjach modelu. Natomiast korzystanie z kombinacji modeli nie różni się od wykonywania zapytań predykcyjnych na wcześniej poznanych modelach. Należy jedynie liczyć się z dłuższym oczekiwaniem na wyniki niż w przypadku wysyłania zapytania predykcyjnego do prostego modelu.

```
ensemble_model_prediction <- rxPredict(modelObject = ensemble_model,
                                       data = test_df,
                                       extraVarsToWrite = "RUL",
                                       overwrite = TRUE)
ensemble_model_metrics <- evaluateRegressor (observed = ensemble_model_prediction$RUL,
                                             predicted =
ensemble_model_prediction$Score)

ensemble_model_metrics[c(1,2)]
```

```
Mean Absolute Error      Root Mean Squared Error
21.61955                   27.52559
```

Rzeczywisty błąd średniokwadratowy modelu złożonego z regresji liniowej, lasu losowego i wzmocnionych drzew regresji wyniósł 27,5. Był więc nieco większy od błędu najdokładniejszego ze zbudowanych modeli.

7.9. Porównanie modeli

W rozdziale 9 przyjrzymy się metodom oceny i poprawy jakości modeli predykcyjnych. Teraz skoncentrujemy się na prostszym zadaniu. Porównamy trafność modeli predykcyjnych zbudowanych przy użyciu różnych algorytmów i nauczonych na różnych danych. Pozwoli nam to poznać silne i słabe strony poszczególnych algorytmów uczenia maszynowego oraz zastanowić się, jak najlepiej przygotować dla każdego z nich dane treningowe. Proszę jednak pamiętać, że nie ma najlepszego algorytmu uczenia maszynowego oraz że trafność modeli również zależy od konfiguracji ich hiperparametrów, co zostanie wyjaśnione w rozdziałach 9 i 10.

Do testowania modeli użyjemy funkcji `model_factory`[16]. Jej działanie polega na trenowaniu wszystkich omówionych modeli przy użyciu danych przekazanych jako parametry jej wywołania. Na przykład, żeby wytrenować modele na 20 zmiennych wybranych z danych źródłowych, należy wykonać poniższe

instrukcje.

```
train_table_name <- "PredictiveMaintenance.train_Labels"
test_table_name <- "PredictiveMaintenance.test_Labels"
top_variables <- 20
model_factory (train_table_name, test_table_name, top_variables)
```

Metryki opisujące poszczególne modele, w tym ich trafność mierzoną błędem średniokwadratowym, opiszemy nazwą algorytmu, typem użytych danych oraz liczbą zmiennych i zapiszemy w tabeli `PredictiveMaintenance.Regression_metrics`.

```
metrics_df <- rbind(linear_model_metrics, GLM_model_metrics, nn_model_metrics,
                        rf_model_metrics, GB_model_metrics, ensemble_model_metrics)
metrics_df <- as.data.frame(metrics_df)
rownames(metrics_df) <- NULL
models <- c("rxFastLinear on raw data",
               "rxGlm on raw data",
               "rxNeuralNet on raw data",
               "rxFastForest on raw data",
               "rxFastTrees on raw data",
               "rxEnsemble on raw data")
models <- cbind(models, top_variables)
metrics_df <- cbind(models, metrics_df)
metrics_df$top_variables <- as.integer(trimws(metrics_df$top_variables))
colnames(metrics_df)[1] <- "Name"
colnames(metrics_df)[2] <- "Variables"

metrics_table <- RxSqlServerData(table = "PredictiveMaintenance.Regression_metrics",
                                        connectionString = connection_string)
rxDataStep(inData = metrics_df,
             outFile = metrics_table,
             overwrite = TRUE)
```

Proszę samemu sprawdzić wyniki modeli uczonych na różnej liczbie zmiennych opisanych, wzbogaconych i znormalizowanych danych. Wzbogacone dane są zapisane w tabelach `PredictiveMaintenance.train_Features` i `PredictiveMaintenance.test_Features`, a znormalizowane dane znajdziemy

w tabelach `PredictiveMaintenance.train_Features_Normalized` i `PredictiveMaintenance.test_Features_Normalized`. Ja ograniczyłem się do sprawdzenia modeli nauczonych na 10 i 20 zmiennych wybranych ze wszystkich trzech typów danych. Żeby dopisać wyniki do tabeli, należy zmienić sposób wywołania funkcji `rxDataStep` na poniższy.

```
rxDataStep (inData = metrics_df,
            outFile = metrics_table,
            overwrite = FALSE,
            append = "rows")
```

Dane pięciu najdokładniejszych i pięciu najmniej dokładnych modeli zwraca poniższe zapytanie.

```
WITH ranking AS
       (SELECT *, ROW_NUMBER() OVER (ORDER BY [Root Mean Squared Error]) as Rn
       FROM [PredictiveMaintenance].[Regression_metrics])
SELECT [Name], [Variables], [Root Mean Squared Error], Rn
FROM ranking
WHERE Rn <=5 OR Rn >31
ORDER BY [Root Mean Squared Error];
```

Name	Variables	Root Mean Squared Error	Rn
rxFastTrees on enchanced data	20	22.7645793657517	1
rxFastTrees on normalized data	20	22.7645793657517	2
rxFastTrees on normalized data	10	23.9738189461305	3
rxFastTrees on enchanced data	10	23.9738189461305	4
rxEnsemble on enchanced data	20	27.6608182940118	5
rxNeuralNet on enchanced data	10	54.5082996167996	32
rxFastTrees on raw data	10	54.9475712714661	33
rxFastLinear on raw data	20	55.3139545342142	34
rxNeuralNet on raw data	10	66.3718883874156	35
rxNeuralNet on raw data	20	75.083311931381	36

Rzeczywisty błąd średniokwadratowy najdokładniejszych modeli wynosi 22,7, a najmniej dokładnego – 75. Czyli najlepszy z tych modeli jest ponad trzykrotnie dokładniejszy od najgorszego.

 Proszę samodzielnie porównać nauczone na różnych danych modele. Informacje na temat budowy i działania poszczególnych algorytmów uczenia maszynowego powinny pozwolić Państwu zrozumieć, czemu uczone na różnych danych modele są mniej lub bardziej dokładne. Proszę też spróbować zbudować model, którego błąd średniokwadratowy nie przekracza 20.

Pora podsumować otrzymane wyniki i krótko scharakteryzować poszczególne algorytmy uczenia maszynowego.

Funkcje `rxFastLinear()` (R) i `rx_fast_linear()` (Python) pozwalają budować modele regresji wielorakiej. Powinny być używane, jeżeli zmienne wejściowe są liniowo skorelowane z wyjściową, w danych nie występują wartości odstające, a korelacje między zmiennymi wejściowymi nie są zbyt mocne. W praktyce oznacza to konieczność odpowiedniego przygotowania danych.

Większość hiperparametrów jest dobieranych automatycznie. Współczynnik uczenia jest ustalany metodą *SDCA*, wartości współczynników regularyzacji *L1* i *L2* są dobierane na podstawie danych treningowych, a uczenie modelu zostaje automatycznie przerwane w momencie, gdy błąd empiryczny modelu przestaje się zmniejszać. Brak konieczności samodzielnego dobrania wartości hiperparametrów powoduje, że funkcje te są łatwe w użyciu.

Obie funkcje są też bardzo szybkie i skalowalne, a więc powinniśmy ich użyć, jeżeli zależy nam na jak najkrótszym czasie uczenia modelu oraz gdy dysponujemy ogromnym zbiorem danych treningowych.

Funkcja `rxGlm` pochodzi z pakietu `RevoScaleR` i implementuje ogólny model liniowy. Umożliwia on powiązanie zmiennych wejściowych z wyjściową za pomocą jednej z predefiniowanych albo samodzielnie utworzonej funkcji sklejanej. Pozwala to budować modele dla zmiennych wyjściowych o różnych rozkładach wartości. Pozostałe założenia modelu regresji liniowej dotyczą również modeli *GLM*.

Funkcja jest prosta w użyciu, szybka i skalowalna.

Funkcje `rxNeuralNet` (R) i `rx_neural_net` (Python) pozwalają budować sztuczne sieci neuronowe. Sztuczne sieci neuronowe nie bez powodu są uważane za skomplikowane i trudne w uczeniu. Duży stopień skomplikowania wynika ze sporej liczby hiperparametrów, których wartości należy dobrać do konkretnego przypadku. Przekonaliśmy się, że błędna konfiguracja jednego hiperparametru powoduje zbudowanie niedokładnego i mało wiarygodnego modelu. Do hiperparametrów mających największy wpływ na jakość sztucznej sieci neuronowej należą: liczba ukrytych warstw, liczba węzłów w ukrytych warstwach, sposób zainicjowania wag i metoda optymalizacji procesu uczenia, w szczególności liczba analizowanych w każdym cyklu przykładów i wartości współczynnika uczenia.

Głównym problemem z uczeniem sztucznych sieci neuronowych jest zanikanie gradientu. Wraz ze zwiększaniem liczby ukrytych warstw zmniejsza się, aż w końcu zanika, przesyłany między nimi gradient, co powoduje przedwczesne przerwanie uczenia. W płytkich sztucznych sieciach neuronowych można temu zapobiegać, normalizując wartości zmiennych; w głębokich sieciach dodatkowo stosuje się w tym celu specjalne typy połączeń między węzłami w różnych warstwach. Drugą kwestią jest dobrze nam znany problem nadmiernego dopasowania modelu do danych treningowych. W sztucznych sieciach neuronowych rozwiązuje się go, ograniczając liczbę cykli i dodając człon z regularyzacją do funkcji kosztu. W głębokich sieciach neuronowych najczęściej dodaje się warstwy typu DropOut, w których pewien procent losowo wybranych węzłów jest wyłączany podczas kolejnych cykli treningu.

Funkcje te powinny być używane, jeżeli dysponujemy dużym zbiorem danych, możemy sobie pozwolić na długotrwały trening i zależy nam na jak największej dokładności modelu. Do trenowania sztucznych sieci neuronowych powinniśmy używać akceleratorów GPU. Pozwoli to nie tylko kilkukrotnie skrócić czas uczenia modelu, ale również określić wielkość używanej w każdym cyklu minigrupy. Jak wiemy, zmniejszenie minigrupy do jednego przykładu, czyli

uczenie stochastyczną metodą najszybszego spadku gradientu, pozwala osiągnąć najlepsze rezultaty.

Funkcje `rxFastForest` (R) i `rx_fast_forest` (Python) pozwalają budować modele lasów losowych. Model lasu losowego został przedstawiony w następnym rozdziale. Funkcje te są szybkie, ale wymagają dużej ilości pamięci operacyjnej.

Funkcje `rxFastTrees` (R) i `rx_fast_trees` (Python) pozwalają budować modele wzmocnionych drzew decyzyjnych. Model wzmocnionych drzew decyzyjnych został przedstawiony w następnym rozdziale. Uczenie wzmocnionych drzew decyzyjnych odbywa się sekwencyjnie, więc funkcje te są wolniejsze od funkcji lasu drzew decyzyjnych. Wymagają one dużej ilości pamięci operacyjnej, ale modele utworzone za ich pomocą cechuje duża dokładność.

7.10. Wdrożenie modeli po stronie serwera SQL

Najlepsze modele utworzymy i wytrenujemy po stronie serwera SQL. Na razie za najlepsze uznamy modele o największej dokładności, zostawiając kwestię oceny jakości modeli do rozdziału 9. Utworzona w tym celu procedura `PredictiveMaintenance.TrainRegressionModel` będzie wywoływana z trzema parametrami: nazwą modelu, nazwą tabeli zawierającej dane treningowe i liczbą użytych do treningu, najsilniej skorelowanych ze zmienną wyjściową `RUL`, zmiennych. Procedura ta będzie zwracała wytrenowany model w postaci binarnej.

```
ALTER PROCEDURE [PredictiveMaintenance].[TrainRegressionModel] @modelName varchar(50),
@tableName varchar(100), @topVariables int, @trained_model varbinary(max) OUTPUT
AS
…
```

Procedura wywołuje za pośrednictwem systemowej procedury `sp_execute_external_script` skrypt języka R. Do skryptu są przekazane:

- nazwa modelu – na jej podstawie zostanie zbudowany odpowiedni model;
- nazwa tabeli z danymi treningowymi – została ona użyta do skonstruowania zapytania odczytującego wszystkie dane z tej tabeli;
- liczba zmiennych, która pozwoli nam sformułować hipotezę w postaci formuły uczącej.

```
BEGIN
   DECLARE @inquery NVARCHAR(max) = N'SELECT * FROM ' + @tableName;
…
@input_data_1 = @inquery,
@params = N'@model_name varchar(30), @top_variables int, @trained_model varbinary(max)
OUTPUT',
@model_name = @modelName,
@top_variables = @topVariables,
@trained_model = @trained_model OUTPUT
END
```

Wykonywany skrypt języka R niewiele różni się od skryptów używanych do treningu poszczególnych modeli.

Dane treningowe zostały najpierw zapisane do obiektu typu `data frame` o nazwie `train_table`. Następnie zostały odczytane nazwy zmiennych i ze zbioru wszystkich nazw zostały wyeliminowane niepotrzebne, nieużywane do treningu zmienne. Następnie został policzony współczynnik korelacji między pozostałymi zmiennymi. Z macierzy korelacji została wybrana kolumna zawierająca korelacje między zmiennymi wejściowymi a zmienną wyjściową `RUL`. Policzenie wartości bezwzględnej tych współczynników pozwoliło je posortować od najsilniejszej do najsłabszej korelacji i zbudować formułę uczącą dla *n* najsilniej skorelowanych zmiennych. Znaleziona formuła została dodatkowo wyświetlona podczas wykonywania procedury w okienku

komunikatów – lista nazw zmiennych użytych w formule i ich kolejność będą potrzebne do skonstruowania zapytań predykcyjnych czasu rzeczywistego.

```
train_table <- InputDataSet
train_vars <- rxGetVarNames(train_table)
train_vars <- train_vars[!train_vars  %in% c("label1", "label2", "id", "cycle",
"cycle_orig")]

formula <- as.formula(paste("~", paste(train_vars, collapse = "+")))
correlation <- rxCor(formula = formula,
                         data = train_table)
correlation <- correlation[, "RUL"]
correlation <- abs(correlation)

correlation <- correlation[order(correlation, decreasing = TRUE)]
correlation <- correlation[-1]
correlation <- correlation[1:top_variables]
formula <- as.formula(paste(paste("RUL~"), paste(names(correlation), collapse = "+")))
print (formula)
```

Główna część skryptu sprawdza podaną nazwę modelu i na tej podstawie trenuje odpowiedni model uczenia maszynowego.

```
if (model_name == "rxFastLinear") {
   model <- MicrosoftML::rxFastLinear(formula = formula,
                                      data = train_table,
                           type = "regression",
                           lossFunction = squaredLoss(),
                                      convergenceTolerance = 0.01,
                                      shuffle = TRUE,
                                      trainThreads = NULL,
                                      normalize = "warn",
                                      maxIterations = NULL,
                                      l1Weight = NULL,
                                      l2Weight = NULL)
} else if (model_name == "rxGlm") {
   model <- RevoScaleR::rxGlm(formula = formula,
                                      data = train_table,
                                      family = gaussian())
}
…
```

Nauczony model zostaje zserializowany i zapisany w postaci umożliwiającej wykonywanie zapytań czasu rzeczywistego.

```
trained_model <- rxSerializeModel(model, realtimeScoringOnly = TRUE)
```

Zbudowanie za pomocą opisanej procedury i zapisanie w tabeli modelu wzmocnionych drzew decyzyjnych nauczonego na dwudziestu zmiennych odczytanych ze zbioru wzbogaconych danych treningowych wymaga wykonania poniższych instrukcji.

```
DECLARE @model VARBINARY(MAX);
DECLARE @name VARCHAR(255) = 'rxFastTrees regression on enchanced data';
EXEC [PredictiveMaintenance].[TrainRegressionModel] 'rxFastTrees','[PredictiveMaintenance].
[train_Features]', 20, @model OUTPUT;
IF EXISTS(SELECT * FROM [PredictiveMaintenance].[Models]  where model_name=@name)
  UPDATE [PredictiveMaintenance].[Models]
  SET [model] = @model
  WHERE [model_name] = @name
ELSE
       INSERT INTO [PredictiveMaintenance].[Models] (model_name, model)
       VALUES(@name, @model);
```

```
 STDOUT message(s) from external script:
Rows Read: 20631, Total Rows Processed: 20631, Total Chunk Time: 0.024 seconds
Computation time: 0.032 seconds.
RUL ~ s11 + s4 + s12 + s7 + s15 + s21 + s20 + s2 + s17 + s3 +
     s8 + s13 + s9 + s6 + a3 + a15 + a2 + a20 + a21 + a12
Not adding a normalizer.
Making per-feature arrays
Changing data from row-wise to column-wise
Beginning processing data.
Rows Read: 20631, Read Time: 0, Transform Time: 0
Beginning processing data.
Processed 20631 instances
Binning and forming Feature objects
Reserved memory for tree learner: 67548 bytes
Starting to train ...
Not training a calibrator because it is not needed.
Elapsed time: 00:00:01.4109823

(1 row affected)
```

Zbudowanie modelu lasu drzew decyzyjnych nauczonego na 10 znormalizowanych zmiennych zostało pokazane poniżej.

```
DECLARE @model VARBINARY(MAX);
DECLARE @name VARCHAR(255) = 'rxFastForest regression on normalized data';
EXEC [PredictiveMaintenance].[TrainRegressionModel] 'rxFastForest','[PredictiveMaintenance].
[Train_Features_Normalized]', 10, @model OUTPUT;
IF EXISTS(SELECT * FROM [PredictiveMaintenance].[Models]  where model_name=@name)
  UPDATE [PredictiveMaintenance].[Models]
  SET [model] = @model
  WHERE [model_name] = @name
ELSE
       INSERT INTO [PredictiveMaintenance].[Models] (model_name, model)
       VALUES(@name, @model);
```

Proszę zwrócić uwagę, że próba zbudowania modelu uogólnionej regresji liniowej skończy się niepowodzeniem.

```
DECLARE @model VARBINARY(MAX);
EXEC [PredictiveMaintenance].[TrainRegressionModel] 'rxGlm','[PredictiveMaintenance].
[Train_Features_Normalized]', 10,
        @model OUTPUT;
```

```
Msg 39004, Level 16, State 20, Line 56
A 'R' script error occurred during execution of 'sp_execute_external_script' with HRESULT
0x80004004.
Msg 39019, Level 16, State 2, Line 56
An external script error occurred:
Error in rxSerializeModel.default(model, realtimeScoringOnly = TRUE) :
  Real-time scoring is not supported for model type 'rxGlm
Calls: source ... eval -> rxSerializeModel -> rxSerializeModel.default
```

Powinniśmy pamiętać, że nie wszystkie modele bibliotek `RevoScaleR`, `revoscalepy` i `MicrosoftML` mogą być serializowane z opcją `realtimeScoringOnly = TRUE`, a tylko wybrane modele bibliotek `RevoScaleR` i `revoscalepy` pozwalają wykonywać natywne zapytania predykcyjne. **Brak możliwości wykonania zapytań czasu rzeczywistego albo natywnych zapytań predykcyjnych powinien być wzięty pod uwagę podczas wybierania algorytmu uczenia maszynowego.**

Zapisanego po stronie serwera SQL modelu użyjemy do wykonywania zapytań

predykcyjnych w czasie rzeczywistym. Niestety, utworzony za pomocą funkcji `rxFastTrees` model wzmocnionych drzew decyzyjnych nie może być odpytywany przy użyciu funkcji języka T-SQL `PREDICT`. Takie rozwiązanie byłoby najszybsze i najwygodniejsze. Po pierwsze, używany do predykcji serwer SQL Server nie wymagałby zainstalowania usługi uczenia maszynowego, po drugie, nie musielibyśmy wymieniać z nazwy wszystkich użytych do treningu zmiennych wejściowych, a po trzecie – wyniki predykcji moglibyśmy łatwo połączyć z danymi testowymi.

Funkcja `rxFastTrees` umożliwia za to wykonywanie zapytań predykcyjnych z użyciem procedury `sp_rxPredict`. Ta metoda jest szybsza od wywołania skryptu języka R lub Python, ale zwraca wyłącznie wyniki predykcji i wymaga nie tylko usługi SQL Server Machine Learning Services, ale również włączenia wykonywania kodu CLR i zainstalowania odpowiednich procedur systemowych na poziomie instancji i bazy danych, tak jak zostało to opisane w rozdziale 2.

```
DECLARE @model varbinary(max) = (SELECT model
        FROM [PredictiveMaintenance].[Models]
        WHERE model_name = 'rxFastTrees regression on enchanced data');
EXEC sp_rxPredict
        @model = @model,
        @inputData = N'SELECT RUL , s11 , s4 , s12 , s7 , s15 , s21 , s20 , s2 ,
        s17 , s3 , s8 , s13 , s9 , s6 , a3 , a15 , a2 , a20 , a21 , a12
        FROM [PredictiveMaintenance].[test_Features]';
```

```
Score
140.1025
123.6642
49.83231
102.2105
105.3903
130.891
…
```

Ten rozdział zaczął się od punktu poświęconego zrozumieniu problemu. Na rysunku 7.2 został pokazany cel przykładowego projektu. Mieliśmy stworzyć

aplikację, która na bieżąco będzie oceniała pozostały czas bezawaryjnego działania silników samolotowych. Dla uproszczenia przyjmijmy, że zbierane dane o pracy silników są przetwarzane w opisany w tym rozdziale sposób i zapisywane w tabeli serwera SQL. W pokazany powyżej sposób oceniliśmy pozostały czas bezawaryjnej pracy stu silników, a więc rozwiązaliśmy postawione zadanie.

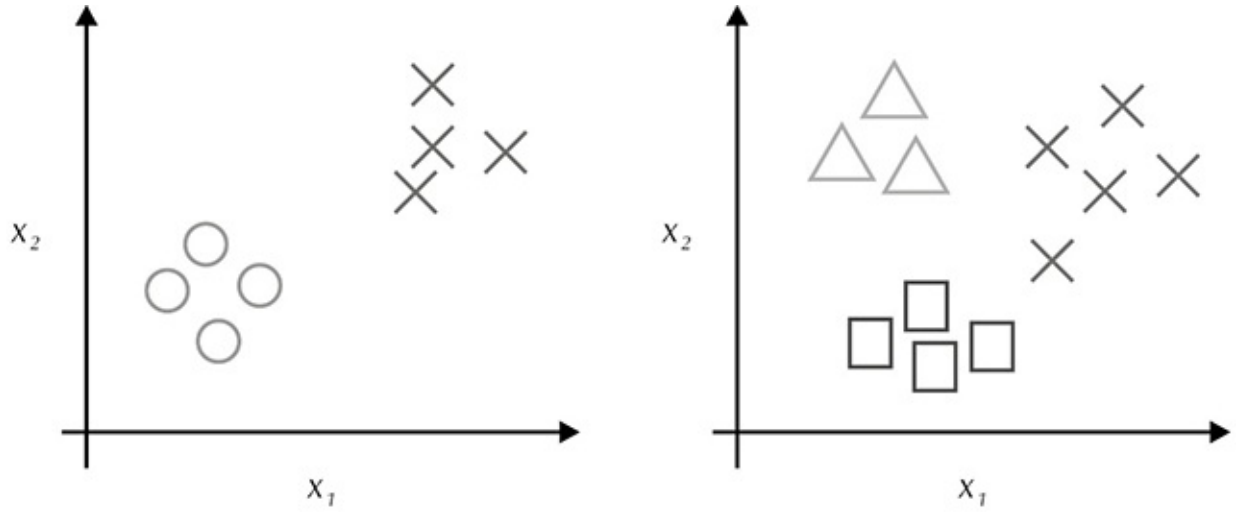
Prosta operacja dzielenia przykładów na klasy daje ogromne możliwości i budzi wiele pytań. Klasyfikując, wprowadzamy do świata porządek, upraszczając postawiony problem.

Daniel Parrochia, Internet Encyclopedia of Philosophy

Rozdział 8
Klasyfikacja

Klasyfikacja jest najstarszą i najczęściej stosowaną metodą uczenia maszynowego. Jej celem jest znalezienie klasyfikatora – modelu, który nauczony na przykładach będzie potrafił przypisać nowe obserwacje do jednej z możliwych klas.

Klasyfikacja, tak jak regresja, jest metodą uczenia nadzorowanego. Dane treningowe muszą więc mieć postać wektora zmiennych wejściowych opisanego zmienną wyjściową. Zmienne objaśniające mogą być numeryczne lub dyskretne, zmienna objaśniana jest dyskretna i przyjmuje przynajmniej dwa stany. Stany zmiennej objaśnianej nazywa się etykietami klas lub krótko – klasami. Klasy te muszą być rozłączne, czyli ten sam przykład nie może należeć do wielu klas. Celem klasyfikacji jest znalezienie dostatecznie dobrego przybliżenia funkcji klasyfikującej *f*, która odwzoruje przykłady $(x_1, \ldots, x_n, y_i)$ na klasę *y* (rys. 8.1).



Rysunek 8.1. Zmienna objaśniana może przyjmować dwa lub więcej stanów. W pierwszym przypadku mamy do czynienia z klasyfikacją binarną (po lewej), w drugim – z klasyfikacją wieloklasową (po prawej)

Tak rozumiana klasyfikacja może być realizowana za pomocą różnych algorytmów, takich jak regresja logistyczna, sztuczne sieci neuronowe, drzewa decyzyjne, maszyna wektorów nośnych, sieci Bayesa, klasyfikatory regułowe i asocjacyjne oraz klasyfikatory najbliższego sąsiedztwa.

W tym rozdziale będziemy kontynuować budowanie modeli proaktywnej konserwacji urządzeń, z tą różnicą, że najpierw spróbujemy zbudować model, który przewidzi, czy dany silnik może być bezpiecznie eksploatowany przez następne trzydzieści cyklów. Następnie zbudujemy model, który odpowie na pytanie, czy pozostały czas bezawaryjnej pracy silnika jest krótszy niż 15 cykli, wynosi 15–30 cykli, czy ponad 30 cykli. Użyjemy do tego pięciu najpopularniejszych klasyfikatorów: regresji logistycznej, sztucznych sieci neuronowych, lasu losowego, wzmocnionych drzew decyzyjnych oraz kombinacji powyższych modeli.

8.1. Klasyfikacja binarna

Najprostszym przykładem klasyfikacji jest klasyfikacja binarna. Odpowiada ona

na pytania typu tak lub nie albo prawda lub fałsz, czyli przypisuje obserwacje do jednej z dwóch możliwych klas. Jedną z tych klas nazywa się pozytywną, a drugą negatywną.

Jeśli jeden stan zmiennej wyjściowej występuje częściej niż drugi, to dane są niezrównoważone. I tak, w naszych danych treningowych jest 17 531 przykładów prawidłowo działających silników, a tylko 3100 przykładów silników, których pozostały czas bezawaryjnej pracy był krótszy niż 30 cykli. Czyli ten zbiór danych jest niezrównoważony.

```
SELECT [label1], COUNT(*) AS nr
FROM [PredictiveMaintenance].[Train_Labels]
GROUP BY [label1];
```

```
label1  nr
0               17531
1               3100
```

W rozdziale 3 zostały opisane techniki równoważenia danych. Jedną z nich było usunięcie części przypadków większościowych. Wzbogacając dane testowe, usunęliśmy z nich część przykładów klasy większościowej, pozostawiając 75 przykładów prawidłowo działających silników i 25 przykładów silników wymagających przeglądu. W efekcie rozkład zmiennej wyjściowej w danych treningowych jest różny od jej rozkładu w danych testowych.

```
SELECT [label1], COUNT(*) AS nr
FROM [PredictiveMaintenance].[Test_Features]
GROUP BY [label1];
```

```
label1  nr
0               75
1               25
```

Zdarza się, że stany zmiennej wyjściowej nie są symetryczne, czyli jeden z nich jest dla nas ważniejszy od drugiego. Tak też jest w omawianym przypadku. Awaria silnika samolotowego może mieć poważne konsekwencje,

a jego prawidłowe działanie traktujemy jako coś normalnego. Asymetrię zmiennej wyjściowej należy uwzględnić podczas oceny jakości klasyfikatora, czemu służyła zmiana rozkładu zmiennej wyjściowej w danych testowych. Metody oceny jakości modeli predykcyjnych zostały opisane w następnym rozdziale.

8.1.1. Przygotowanie danych

W poprzednim rozdziale została przedstawiona funkcja `data_label`, która zwraca wartości zmiennej `RUL` policzone dla zbioru treningowego i testowego. Opisanie danych na potrzeby klasyfikatorów binarnych wymaga dodania dyskretnej, przyjmującej dwa stany, zmiennej wyjściowej. W tym celu do funkcji `data_label` została dodana jedna linijka kodu.

```
data$label1 <- ifelse(data$RUL <= 30, 1, 0)
```

Opisanie danych za pomocą tej funkcji oraz zapisanie uzupełnionych danych w tabelach `PredictiveMaintenance.train_Labels` i `PredictiveMaintenance.train_Labels` nie różni się od metody użytej do opisania danych na potrzeby modeli regresji przedstawionej w rozdziale 7. Wycinek opisanych danych treningowych zwraca poniższe zapytanie.

```
SELECT [id]
       ,[cycle]
       ,[RUL]
       ,[label1]
  FROM [ML].[PredictiveMaintenance].[train_Labels]
  WHERE [RUL] BETWEEN 20 AND 35;
```

id	cycle	RUL	label1
1	158	34	0
1	159	33	0
1	160	32	0
1	161	31	0
1	162	30	1
1	163	29	1
1	164	28	1
1	165	27	1
1	166	26	1
1	167	25	1

Zmienna `label1` przyjmuje dwa stany, a więc możemy jej użyć jako zmiennej wyjściowej modeli klasyfikacji binarnej. Stany tej zmiennej zostały zakodowane za pomocą cyfr 0 i 1, gdzie 0 (stan negatywny) oznacza, że czas pozostałej, bezawaryjnej pracy urządzenia jest dłuższy niż 30 cykli, a 1 (stan pozytywny), że silnik wymaga przeglądu w ciągu 30 najbliższych cykli swojej pracy. Użycie cyfr jako etykiet klas pozwoli nam ocenić korelację między zmiennymi wejściowymi a wyjściową na podstawie współczynnika ich korelacji liniowej.

Przypomnijmy, że dysponujemy trzema wersjami danych: opisanymi, wzbogaconymi i znormalizowanymi. Pozostało nam jedynie wybrać *n* zmiennych wejściowych najsilniej skorelowanych z wyjściową. Wszystkie zmienne wejściowe są numeryczne, co jest wymogiem algorytmów uczenia maszynowego biblioteki `MicrosoftML`. Zmienna wyjściowa jest zmienną binarną, ale dla uproszczenia najsilniej skorelowane z nią zmienne wejściowe wybierzemy w ten sam sposób, co w rozdziale 7, z tym, że tym razem policzymy współczynnik korelacji liniowej między zmienną `label1` a zmiennymi wejściowymi.

W przykładowym rozwiązaniu użyjemy wzbogaconych danych i postawimy hipotezę, że bezawaryjna praca silnika zależy od 20 zmiennych wejściowych najsilniej skorelowanych ze zmienną `label1`. Proszę zwrócić uwagę, że stany zmiennej wyjściowej zostały jawnie podane jako `0` i `1` – dyskretyzacja zmiennej wyjściowej jest niezbędna do zbudowania modeli klasyfikujących. Jednak podczas wyznaczania współczynnika korelacji liniowej wartości te zostały przekształcone na liczby. Większość funkcji bibliotek `RevoScaleR`, `revoscalepy` i `MicrosoftML` może być wywołana z parametrem `transforms`, który określa sposób przekształcenia danych.

```
train_table_name <- "PredictiveMaintenance.train_Features"
test_table_name <- "PredictiveMaintenance.test_Features"
top_variables <-20

train_table <- RxSqlServerData(table = train_table_name,
                  connectionString = connection_string,
                  colInfo = list(label1 = list(type = "factor", levels = c("0", "1"))))
train_df <- rxImport(train_table)

test_table <- RxSqlServerData(table = test_table_name,
                 connectionString = connection_string,
                 colInfo = list(label1 = list(type = "factor", levels = c("0", "1"))))
test_df <- rxImport(test_table)

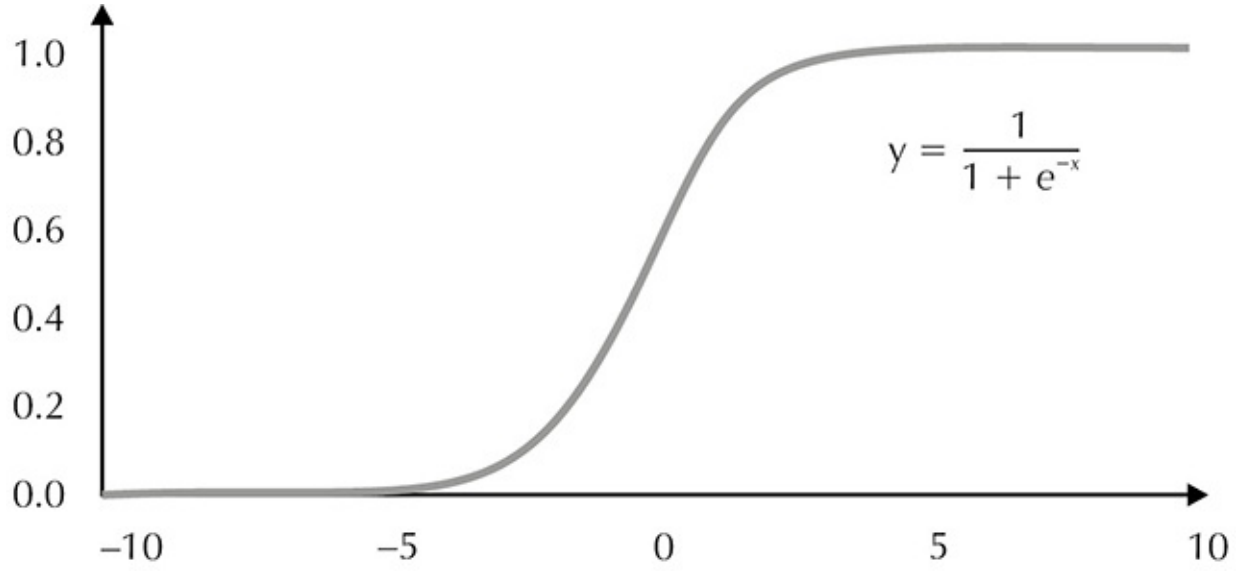
train_vars <- rxGetVarNames(train_df)
train_vars <- train_vars[!train_vars  %in% c("RUL", "label2", "id", "cycle_orig","cycle")]
formula <- as.formula(paste("~", paste(train_vars, collapse = "+")))
correlation <- rxCor(formula = formula,
                        data = train_df,
                        transforms = list(label1 = as.numeric(label1)))
correlation <- correlation[, "label1"]
correlation <- abs(correlation)
correlation <- correlation[order(correlation, decreasing = TRUE)]
correlation <- correlation[-1]
correlation <- correlation[1:top_variables]
formula <- as.formula(paste(paste("label1~"),
                                  paste(names(correlation), collapse = "+")))
formula
```

```
label1 ~ a11 + a4 + a15 + a21 + a17 + a3 + a20 + a2 + a12 + a7 +
     s11 + s4 + s12 + s7 + s15 + s21 + s20 + s17 + s2 + a8
```

8.1.2. Regresja logistyczna

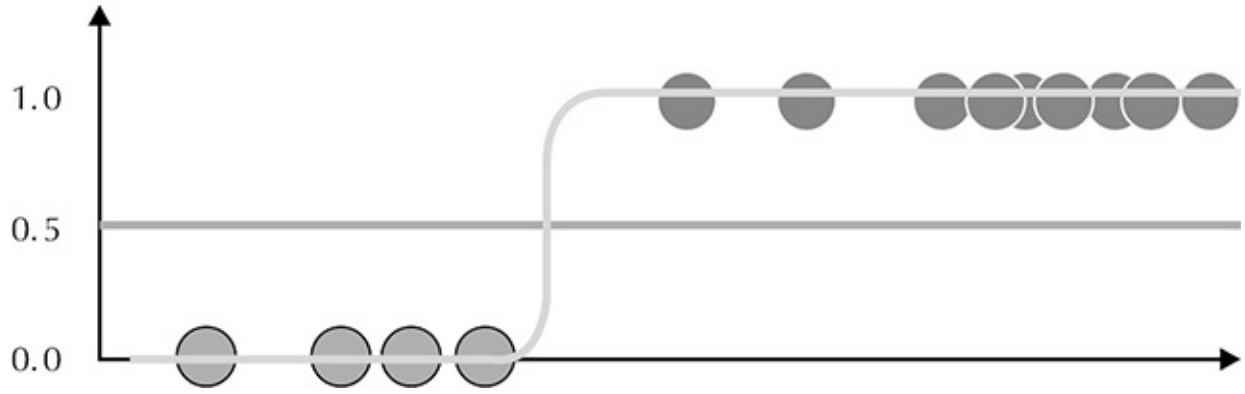
Poznany wcześniej model regresji liniowej przewiduje rzeczywistą wartość zmiennej wyjściowej na podstawie wartości zmiennych wejściowych. **Regresja logistyczna zwraca prawdopodobieństwo, z jakim przykłady należą do jednej z możliwych klas**.

Funkcja liniowa zwraca wartości z zakresu od –∞ do ∞. Trudno więc traktować je jako prawdopodobieństwo. Nie wiadomo, co miałyby oznaczać wartości mniejsze od zera i większe od 1. Potrzebujemy więc innej funkcji, najlepiej zwracającej wartości z zakresu 0–1, które łatwo można zinterpretować jako prawdopodobieństwa. Doskonale nadaje się to tego funkcja sigmoidalna (rys. 8.2).



Rysunek 8.2. Funkcja sigmoidalna przekształca duże wartości ujemne do bliskich zeru, a duże – dodanie do bliskich jednemu. Natomiast wartości bliskie zeru przekształca prawie liniowo

Wyniki funkcji sigmoidalnej można traktować jako prawdopodobieństwa, z jakimi przykład należy do klasy pozytywnej. Im większy wynik funkcji, tym większe prawdopodobieństwo, że mamy do czynienia z przykładem klasy pozytywnej. A jeżeli przekroczy ono określoną wartość progową, uznamy, że wynikiem predykcji jest klasa pozytywna (rys. 8.3).



Rysunek 8.3. Najczęściej używaną wartością progową jest 0,5, czyli jeżeli prawdopodobieństwo przekracza 50%, przykład jest klasyfikowany jako pozytywny

Do zbudowania modelu uczenia maszynowego potrzebujemy, oprócz danych treningowych i algorytmu reprezentującego wiedzę, funkcji kosztu i metody optymalizacji procesu uczenia. Zacznijmy od wyboru odpowiedniej dla klasyfikatorów funkcji kosztu.

Przypomnijmy, że funkcja kosztu jest używana podczas uczenia modelu do minimalizacji popełnianych przez niego błędów. **W przypadku klasyfikatorów binarnych błąd polega na zwróceniu innej klasy niż rzeczywista, czyli przypisaniu przykładów pozytywnych do klasy negatywnej lub przykładów negatywnych do klasy pozytywnej**.

W najprostszym przypadku uczenie modelu mogłoby polegać na ograniczeniu do minimum tak rozumianych błędów, czyli na użyciu zerojedynkowej funkcji kosztu do znalezienia optymalnych wartości parametrów *w*

$$\min_w \sum_n 1 \cdot [y_n (\beta_0 + \beta * x_n) > 0]$$

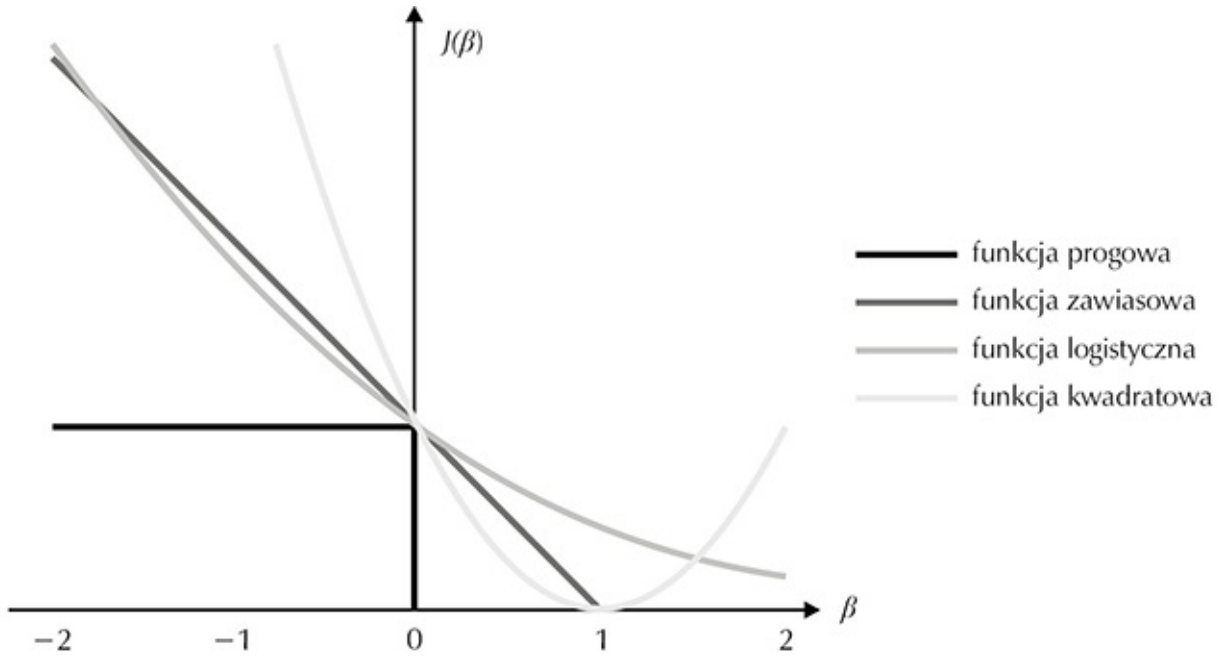
gdzie (·) jest funkcją wskaźnika zwracającą jeden, gdy (·) jest prawdą i zero w przeciwnym razie. Dokładnie w ten sposób uczy się poznany w poprzednim rozdziale perceptron.

Niestety, progowa (zerojedynkowa) funkcja kosztu nie nadaje się do rozwiązywania problemów nieseparowalnych liniowo. Nie możemy jej użyć

nawet do znalezienia przybliżonych wyników. Problem wynika z tego, że czasami bardzo mała zmiana wartości parametrów prowadzi do zmiany predykcji z zera na jeden, co ma znaczący wpływ na koszt modelu. Rozwiązanie tego problemu wymaga zastąpienia nieróżniczkowalnej funkcji progowej funkcją, która w pewnym zakresie wartości jest liniowa lub zbliżona do liniowej. Najlepiej funkcją wypukłą, bo jak wiemy z poprzedniego rozdziału funkcje mające tylko jedno globalne minimum najłatwiej optymalizuje się metodą spadku gradientu.

Trzema najczęściej używanymi w klasyfikatorach funkcjami kosztu są (rys. 8.4):

- funkcja zawiasowa (*Hinge*) hin $(y, \hat{y}) = \max[0,1 - y\hat{y}]$,
- funkcja logistyczna log $(y, \hat{y}) = \log(1+\exp[-y\hat{y}])$,
- funkcja kwadratowa sqr $(y, \hat{y}) = (y - \hat{y})^2$.



Rysunek 8.4. Koszty błędnych predykcji, a więc ocena postępów uczenia modelu, zależą od użytej funkcji kosztu

Pozostało nam wybrać algorytm optymalizacji procesu uczenia modelu. W omawianym modelu regresji logistycznej została zastosowana optymalizacja

drugiego rzędu algorytmem *BFGS* (*Broyden–Fletcher–Goldfarb–Shanno*). Zamiast wyznaczyć macierz Hessego, co jest procesem obliczeniowo skomplikowanym i wymagającym dużych ilości pamięci, zostaje ona oszacowana na podstawie bieżących gradientów metodą quasi-newtonowską. Parametr `memorySize` pozwala określić liczbę poprzednich gradientów przechowywanych w pamięci, na podstawie których są obliczane bieżące wartości parametrów modelu.

```
logit_model <- MicrosoftML::rxLogisticRegression(formula = formula,
                                    data = train_df,
                                    type = "binary",
                                    l1Weight = 0.9,
                                    l2Weight = 0.9,
                                    maxIterations = 100,
                                    optTol = 1e-07,
                                    normalize = "warn",
                                    memorySize = 100)
Warning: A normalizer is needed for this trainer. Either add a normalizing transform or
use the 'norm=Auto', 'norm=Yes' or 'norm=No' options.
Beginning processing data.
Rows Read: 20631, Read Time: 0, Transform Time: 0
Beginning processing data.
```

```
Warning: The number of threads specified in trainer arguments is larger than the
concurrency factor setting of the environment. Using 2 training threads instead.
LBFGS multi-threading will attempt to load dataset into memory. In case of out-of-memory
issues, turn off multi-threading by setting trainThreads to 1.
Beginning optimization
num vars: 21
improvement criterion: Mean Improvement
L1 regularization selected 21 of 21 weights.
```

Najważniejsze informacje o nauczonym modelu, w tym wywołanie funkcji `rxLogisticRegression`, formułę uczącą, nazwę zbioru danych treningowych oraz znalezione wartości parametrów zwraca funkcja `summary`.

```
summary(logit_model)
```

```
Call:
MicrosoftML::rxLogisticRegression(formula = formula, data = train_df,
     type = "binary", l2Weight = 0.9, l1Weight = 0.9, optTol = 1e-07,
     memorySize = 100, maxIterations = 100, normalize = "warn")

LogisticRegression (BinaryClassifierTrainer) for: label1~a11+a4+a15+a21+a17+a3+a20+a2+a12+
a7+s11+s4+s12+s7+s15+s21+s20+s17+s2+a8
Data: train_df

First 20 of 21 Non-zero Coefficients:
(Bias): -0.0002877593
a8: -0.3750789
a4: 0.3169563
a3: 0.3127396
s17: 0.1179028
a17: 0.1159189
a12: -0.103943
s12: -0.09981622
s7: -0.03417609
a7: -0.03386481
s11: 0.02263462
a11: 0.02177382
s21: -0.02038369
a21: -0.01948711
a20: -0.01067552
s20: -0.009868823
s2: -0.002136317
s15: 0.001861423
a15: 0.001683041
s4: -0.0003212563
```

Wywołana dla modelu regresji logistycznej funkcja `rxPredict` zwraca, oprócz klasy, do której została przypisana obserwacja, wynik predykcji (`Score`) oraz prawdopodobieństwo, na podstawie którego została ona sklasyfikowana jako pozytywna lub negatywna (`Probability`). Do wyników predykcji została dołączona odczytana z danych testowych klasa, do której należy obserwacja (`label1`).

```
logit_model_prediction <- rxPredict(modelObject = logit_model,
                                    data = test_df,
                                    extraVarsToWrite = "label1",
                                    overwrite = TRUE)
tail(logit_model_prediction)
```

	label1	PredictedLabel	Score.1	Probability.1
95	0	0	-8.5495434	1.935960e-04
96	0	0	-9.1746044	1.036275e-04
97	0	0	-7.8722968	3.810127e-04
98	0	0	-3.0762765	4.409650e-02
99	0	0	-10.6811962	2.297235e-05
100	1	1	0.4012015	5.989763e-01

Prawdopodobieństwa są policzone na podstawie wyników predykcji przy użyciu funkcji sigmoidalnej, co można łatwo sprawdzić samemu przy użyciu funkcji `sigmoid` z biblioteki `e1071`.

```
as.data.frame(tail(e1071::sigmoid(logit_model_prediction$Score.1)))
```

```
1.935960e-04
1.036275e-04
3.810127e-04
4.409650e-02
2.297235e-05
5.989763e-01
```

Prawdopodobieństwo przynależności do klasy pozytywnej pozwala samodzielnie określić wartość progową, po przekroczeniu której obserwacja ma być sklasyfikowana jako pozytywna. Czyli pozwala skalibrować czułość

klasyfikatora. Zrobimy to w następnym punkcie.

Znajomość tego prawdopodobieństwa umożliwia też posortowanie predykcji na podstawie stopnia ich pewności – na przykład odczytania informacji o pięciu urządzeniach, które z najwyższym prawdopodobieństwem ulegną awarii w ciągu następnych trzydziestu cykli pracy.

```
head(logit_model_prediction[order(-logit_model_prediction$Probability.1),c(1,2,4)])
```

```
       label1 PredictedLabel Probability.1
34          1              1     0.9930858
81          1              1     0.9913046
76          1              1     0.9852911
35          1              1     0.9829887
31          1              1     0.9814182
68          1              1     0.9696730
```

Do oceny modelu regresji logistycznej użyjemy rozbudowanej wersji funkcji `evaluate_classifier` przedstawionej w następnym rozdziale. W tym miejscu ograniczymy się do użycia trzech, najbardziej intuicyjnych metryk klasyfikatorów. Pierwszą z nich jest trafność, czyli procent prawidłowych predykcji.

```
logit_model_metrics <- evaluate_classifier(actual = "label1",
                           predicted = "PredictedLabel",
                           data = logit_model_prediction)
logit_model_metrics$Metrics[1]
```

```
0.92
```

Trafność naszego modelu wynosi 92%, czyli model średnio myli się 8 razy na 100 predykcji.

Trafność nie uwzględnia rozkładu wartości zmiennej wyjściowej. W tym przypadku 75% przykładów testowych należało do klasy negatywnej, a więc 75% trafność uzyskałby model, który wszystkie obserwacje klasyfikowałby jako negatywne. Rozkład ten uwzględnia współczynnik Kappa Cohena. Im jest on

bliższy 1, tym lepiej.

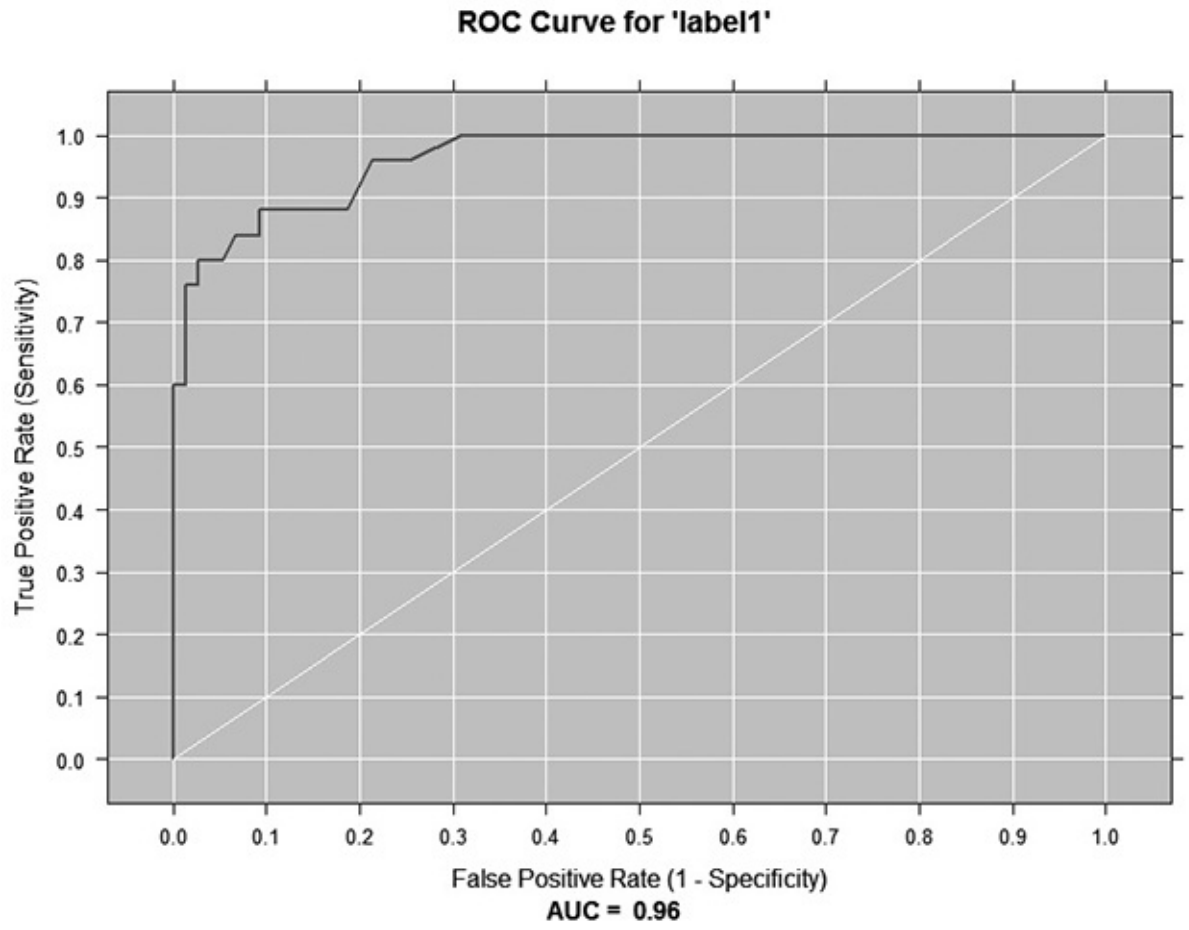
```
logit_model_metrics$Metrics[5]
```

```
0.7681159
```

Trzecią z użytych w tym rozdziale miar jest wskaźnik obszaru pod krzywą *AUC*. Wskaźnik ten można traktować jako połączenie obu powyższych metryk. Interpretuje się go jako prawdopodobieństwo z którym model wyżej oceni losowo wybrany pozytywny element niż losowo wybrany negatywny element (rys. 8.5).

```
roc_curve(data = logit_model_prediction,
            observed = "label1",
            predicted = "Probability.1")
```

Nauczony na dwudziestu wzbogaconych zmiennych model regresji logistycznej bardzo dobrze poradził sobie z klasyfikacją silników jako wymagających przeglądu lub zapewniających bezawaryjną pracę przez co najmniej 30 kolejnych cykli pracy.



Rysunek 8.5. Wysoki, wynoszący 96% współczynnik AUC nie musi oznaczać wiarygodnego modelu

8.1.3. Sztuczne sieci neuronowe

Przedstawione w poprzednim rozdziale sztuczne sieci neuronowe mogą być używane – i często są – do klasyfikacji. Do zbudowania klasyfikatora binarnego wystarczy zastosować w warstwie wyjściowej funkcję sigmoidalną, żeby sztuczna sieć neuronowa zwracała prawdopodobieństwo przynależności przykładu do klasy pozytywnej.

Do zbudowania sztucznej sieci neuronowej z pięcioma ukrytymi węzłami, losowymi wagami początkowymi z zakresu od –0,05 do 0,05 i nauczenia jej metodą *Adadelta* użyjemy funkcji `rxNeuralNet`. W tym przypadku liczba cykli

została ograniczona do 100, a dane treningowe zostały automatycznie znormalizowane metodą *Min-Max*.

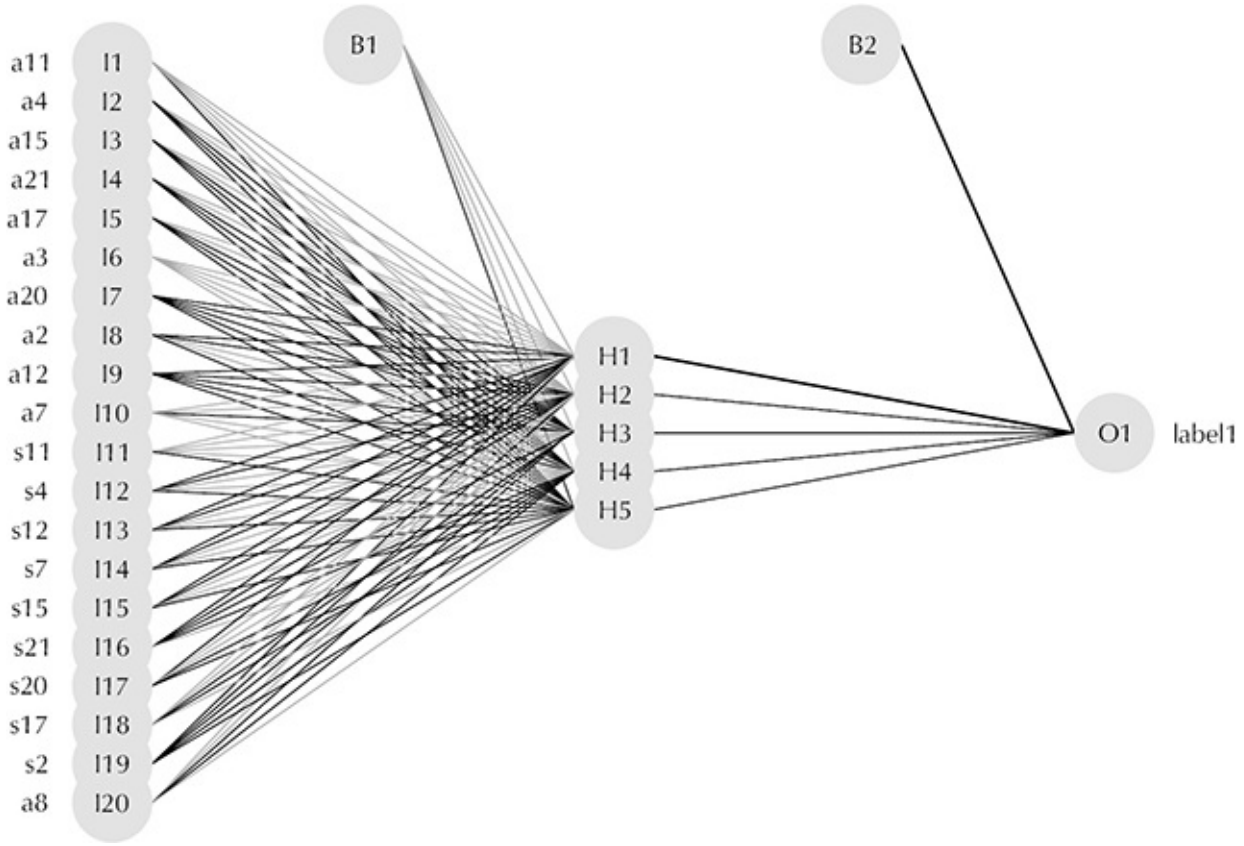
```
nn_model <- MicrosoftML::rxNeuralNet(formula,
                                     data = train_df,
                                     type = "binary",
                                     normalize = "auto",
                                     numIterations = 100,
                                     optimizer = adaDeltaSgd(),
                                     numHiddenNodes = 5,
                                     initWtsDiameter = 0.05)
```

```
Automatically adding a MinMax normalization transform, use 'norm=Warn' or 'norm=No' to
turn this behavior off.
Beginning processing data.
Rows Read: 20631, Read Time: 0, Transform Time: 0
Using: SSE Math

***** Net definition *****
  input Data [20];
  hidden H [5] sigmoid { // Depth 1
    from Data all;
  }
  output Result [1] sigmoid { // Depth 0
    from H all;
  }
***** End net definition *****
Input count: 20
Output count: 1
Output Function: Sigmoid
Loss Function: LogLoss
PreTrainer: NoPreTrainer
_____________________________________________________________________
Starting training...
Using Adadelta with decay 0.950000 and conditioner 0.000001
InitWtsDiameter: 0.050000
_____________________________________________________________________
Initializing 1 Hidden Layers, 111 Weights...
Estimated Pre-training MeanError = 0.704088
Iter:1/100, MeanErr=0.426624(-39.41%%), 66.96M WeightUpdates/sec
Iter:2/100, MeanErr=0.426069(-0.13%%), 61.87M WeightUpdates/sec
Iter:3/100, MeanErr=0.425957(-0.03%%), 63.33M WeightUpdates/sec
…
Done!
Estimated Post-training MeanError = 0.109530
```

Jak widać, funkcją kosztu modelu jest funkcja logistyczna, a funkcją aktywacyjną węzła wyjściowego – funkcja sigmoidalna. Liczba węzłów wejściowych jest równa liczbie zmiennych objaśniających, a węzeł wyjściowy reprezentuje zmienną objaśnianą (rys. 8.6).

Znalezione wartości parametrów zwraca, razem z definicją modelu, funkcja `summary`. Nie wiedząc jednak, co reprezentują ukryte węzły sieci, **nie możemy na podstawie wartości wag wnioskować o wpływie poszczególnych zmiennych wejściowych na wyniki predykcji**.



Rysunek 8.6. Każdy z pięciu ukrytych węzłów jest połączony z dwudziestoma węzłami wejściowymi i węzłem reprezentującym obciążenie. W sumie, w warstwie ukrytej występuje 105 wag (parametrów). Węzeł wyjściowy jest połączony z pięcioma węzłami ukrytymi i węzłem reprezentującym obciążenie. Całkowita liczba wag (parametrów modelu) wynosi więc 105+6, czyli 111

```
summary(nn_model)
```

```
Call:
MicrosoftML::rxNeuralNet(formula = formula, data = train_df,
     type = "binary", numHiddenNodes = 5, numIterations = 100,
     optimizer = adaDeltaSgd(), initWtsDiameter = 0.05, normalize = "auto")

BinaryNeuralNetwork (BinaryClassifierTrainer) for: label1~a11+a4+a15+a21+a17+a3+a20+a2+a12+
a7+s11+s4+s12+s7+s15+s21+s20+s17+s2+a8
Data: train_df

Input Layers: input Data [20];

Layers: hidden H [5] sigmoid { // Depth 1
  Biases = Biases_0;
  from Data all { Weights = Weights_0; }
}
output Result [1] sigmoid { // Depth 0
  Biases = Biases_1;
  from H all { Weights = Weights_1; }
}

Share Info: NA
Weights_0: -4.074187, -4.610146, -2.843689, 4.676655, -2.289007, -2.160217, 4.600793,
0.2759539, 2.06689, 2.182161, -4.30747, -4.723763, 2.122994, 2.240555, -2.973464,
4.715156, 4.624701, -2.426037, 0.2133011, 0.8633268, -4.082237, -4.608708, -2.83108,
4.678593, -2.319449, -2.131996, 4.573697, 0.2644021, 2.054986, 2.190969, -4.313144,
-4.742919, 2.1531, 2.259538, -2.978884, 4.75464, 4.628057, -2.430983, 0.2175103,
0.8340009, -4.100888, -4.609922, -2.834291, 4.688862, -2.281282, -2.160448, 4.601763,
0.2574176, 2.083648, 2.190844, -4.298857, -4.750398, 2.126997, 2.251541, -2.971022,
4.737001, 4.625963, -2.412868, 0.2029542, 0.8622264, -4.074539, -4.57041, -2.838381,
4.666986, -2.313461, -2.142382, 4.569792, 0.2393654, 2.05189, 2.187512, -4.280387,
-4.719258, 2.147023, 2.24834, -2.976488, 4.723338, 4.597361, -2.435449, 0.208795,
0.8734086, -4.119713, -4.585478, -2.844419, 4.692, -2.311992, -2.135069, 4.598332,
0.2541028, 2.051047, 2.173005, -4.286552, -4.74517, 2.148114, 2.23369, -2.96506...
<truncated>
Weights_1: -20.19048, -20.37185, -20.37922, -19.94861, -20.21306
Biases_0: 0.8792817, 0.8881673, 0.849664, 0.8831275, 0.8737063
Biases_1: 27.85569
Layer consts: NA
```

Proszę zwrócić uwagę, że większość znalezionych wag jest albo mniejsza od –2, albo większa od 2. Jeżeli przypominamy siebie kształt użytej w obu warstwach funkcji sigmoidalnej, zdamy sobie sprawę, że węzły naszej sieci neuronowej działają jak przełączniki binarne – zwracają zero lub jeden. Sprawdźmy, czy taki, przypominający działanie mózgu, tryb działania sztucznej sieci neuronowej okazał się skuteczny w klasyfikacji binarnej (jak pamiętamy z poprzedniego rozdziału, sztuczna sieć neuronowa nie poradziła sobie z obliczeniem liczby pozostałych cykli bezawaryjnej pracy urządzeń).

Wyniki predykcji na danych testowych są interesujące. Zbudowana sztuczna sieć neuronowa rzeczywiście działa jak przełącznik binarny.

```
nn_model_prediction <- rxPredict(modelObject = nn_model,
                                 data = test_df,
                                 extraVarsToWrite = "label1")
tail(nn_model_prediction[,-3],9)
```

	label1	PredictedLabel	Probability.1
92	1	1	0.833868980407715
93	0	0	0.215862259268761
94	0	0	0.256129652261734
95	0	0	0.000187654601177
96	0	0	0.000007976889719
97	0	0	0.002289201831445
98	0	0	0.014889473095536
99	0	0	0.000008902418813
100	1	0	0.478705972433090

Nasz model czasami popełnia błędy, na przykład silnik o numerze 100 został błędnie sklasyfikowany jako wymagający przeglądu. Ale co ważne, jeżeli klasyfikacja jest prawidłowa, to model jest bardzo pewny swoich decyzji, tj. prawdopodobieństwo, że przykład należy do klasy pozytywnej jest albo bardzo małe, albo bardzo duże.

Zobaczmy, jak przekłada się to na trafność, współczynnik Kappa i obszar pod krzywą modelu.

```
nn_model_metrics$Metrics[c(1,5)]
rxAuc(roc(data = nn_model_prediction,
            observed = "label1",
            predicted = "Probability.1"))
```

```
0.9     0.7014925373
0.9754666667
```

Sztuczna sieć neuronowa jest o 2% mniej dokładna od modelu regresji logistycznej. Ponieważ oba modele zostały ocenione na podstawie tych samych danych, współczynnik Kappa mniej dokładnego modelu też jest mniejszy. Mimo to współczynnik *AUC* jest większy i w zaokrągleniu wynosi 98%. Proszę samemu sprawdzić, jak wyłączenie automatycznej normalizacji zmiennych wpłynie na jakość modelu.

W tym przypadku koszt niewykrycia zbliżającego się uszkodzenia silnika jest większy niż koszt błędnego sklasyfikowania poprawnie działającego urządzenia jako mogącego ulec awarii, czyli zmienna wyjściowa jest niesymetryczna. **Zwiększmy więc czułość naszego klasyfikatora, obniżając próg prawdopodobieństwa do 40%.** Zamiast przypisywać do klasy pozytywnej przykłady, dla których prawdopodobieństwo było większe niż 50%, przypiszemy do tej klasy przykłady, dla których prawdopodobieństwo, że w ciągu najbliższych 30 cykli silnik ulegnie awarii, przekroczyło 40%.

```
nn_model_prediction$newLabel1 <- ifelse(nn_model_prediction$Probability.1<=0.4, 0,1)
```

Sprawdźmy trafność i współczynnik Kappa tak uzyskanych wyników (obszar pod krzywą jest obliczany na podstawie prawdopodobieństw, a więc będzie taki sam).

```
nn_model_metrics <- evaluate_classifier(actual = "label1", predicted = "newLabel1",
                                        data = nn_model_prediction)
nn_model_metrics$Metrics[c(1,5)]
```

```
0.93    0.8
```

Zrozumienie problemu biznesowego (niesymetryczności zmiennej wyjściowej) i danych (niezrównoważenia zbioru treningowego) pozwoliło nam odpowiednio przygotować dane (zrównoważyć dane testowe) oraz poprawić jakość predykcji (przez obniżenie czułości modelu).

8.1.4. Klasyfikacja przez indukcję drzew decyzyjnych

W trzecim rozdziale zostały przedstawione modele partycjonujące, czyli modele, które uczą się, dzieląc przykłady treningowe na odpowiednie podzbiory. Znalezienie kryteriów podziału, które z wystarczająco dobrym przybliżeniem reprezentują prawdziwe zależności między zmiennymi wejściowymi a wyjściową, kończy uczenie takich modeli. Najczęściej używanymi modelami partycjonującymi są drzewa decyzyjne, a dokładniej mówiąc – ich kombinacje. W tym punkcie przypomnimy sobie i sformalizujemy informacje na temat drzew decyzyjnych, a następnie omówimy trzy popularne metody ich łączenia: metodę agregacji bootstrapowej, metodę wzmacniania oraz najprostszą metodę kombinowania modeli przez uśrednianie wyników ich predykcji.

Najlepszym drzewem decyzyjnym jest model o najmniejszym błędzie rzeczywistym. Niezależnie od przyjętej funkcji kosztu może się wydawać, że zbudowanie najlepszego drzewa decyzyjnego sprowadza się do nauczenia wszystkich możliwych drzew decyzyjnych na danych treningowych, a następnie wyborze modelu o najmniejszym błędzie empirycznym. Uważni czytelnicy zauważyli w tym momencie, że taka naiwna metoda w praktyce preferowałaby nadmiernie dopasowane modele.

Metoda ta byłaby nie tylko nieskuteczna (bo prowadziłaby do wyboru nadmiernie dopasowanego modelu), ale również bardzo wolna i możliwa do zastosowania tylko do niewielkich zbiorów danych treningowych. Liczba możliwych drzew klasyfikatora binarnego rośnie wykładniczo wraz z liczbą

zmiennych wejściowych, czyli dla kilkudziesięciu zmiennych konieczne byłoby zbudowanie kilku tysięcy modeli drzew decyzyjnych. Dlatego w praktyce konstruowanie drzew decyzyjnych polega na szukaniu suboptymalnych rozwiązań zachłanną metodą dziel i rządź. Tak działają klasyczne, opisane w rozdziale 3 algorytmy *ID3* oraz *CART*.

Dodawanie do drzew decyzyjnych kolejnych węzłów wymaga określenia kryterium podziału przykładów. Jednym ze stosowanych kryteriów jest zysk informacyjny. Mierzy on różnicę między oczekiwaną ilością informacji w bitach niezbędną do sklasyfikowania przykładu przed i po podziale danych treningowych. Na danym etapie konstruowania drzewa jest wybierana zmienna, według której podział da największy zysk informacyjny, minimalizując w ten sposób ilość informacji potrzebnej do prawidłowego sklasyfikowania przykładu

$$I(S_1, \ldots S_n) = -\sum_{i=1}^{n} P_i \log_2(P_i)$$

gdzie S_1, …, S_n są uzyskanymi w wyniku podziału podzbiorami danych treningowych, a P_i oznacza prawdopodobieństwo sklasyfikowania przykładu.

Ilość informacji potrzebna do sklasyfikowania przykładu w danym węźle drzewa decyzyjnego nazywa się entropią tego węzła. Entropia jest najmniejsza, czyli zysk informacyjny jest największy, jeśli w wyniku podziału otrzymamy podzbiory zawierające wyłącznie przykłady należące do tej samej klasy. Odpowiednio, entropia węzła jest największa (zysk informacyjny jest najmniejszy), jeśli rozkład zmiennej wyjściowej otrzymany w wyniku podziału jest w obu podzbiorach równomierny. Procedura podziału jest powtarzana aż do skonstruowania całego drzewa decyzyjnego.

Konstruowanie drzew decyzyjnych na podstawie zysku informacyjnego faworyzuje zmienne przyjmujące wiele stanów. Jeżeli zmienna przyjmuje 500 stanów, to zysk z podziału według tej zmiennej będzie średnio większy niż podział względem zmiennej, która przyjmuje tylko 5 stanów. Prowadzi to do

tworzenia bardzo małych węzłów drzewa, w skrajnych przypadkach liczących pojedyncze obserwacje. To z kolei zwiększa ryzyko nadmiernego dopasowania modelu. Na przykład, nieusunięcie z danych treningowych identyfikatorów rekordów prowadzi do zbudowania drzewa decyzyjnego, które dzieli przykłady na podstawie ich unikatowych identyfikatorów. Zysk informacyjny z takich podziałów jest maksymalny, bo otrzymane podzbiory zawierają wyłącznie przykłady (często pojedyncze) należące do tej samej klasy.

Problem ten można rozwiązać na trzy sposoby. Po pierwsze, uogólniając zmienne kategoryczne przyjmujące wiele stanów. Druga metoda polega na odpowiednim dobraniu hiperparametrów modelu. Wreszcie możemy też użyć innego kryterium podziału. Przyjrzyjmy się tej trzeciej metodzie.

W klasycznym algorytmie C4.5 do oceny podziału zastosowano współczynnik zysku

$$Gain\ ratio = \frac{Zysk\ informacyjny}{Współczynnik\ podziału}$$

Uwzględnia on rozkład podzielonych względem wybranej zmiennej przykładów v oraz wielkości zbioru wejściowego D i uzyskanego w wyniku podziału podzbioru D_i

$$Współczynnik\ podziału = -\Sigma_{i=1}^{n} \frac{|S_n|}{|D|} \log_2\left(\frac{|S_n|}{|D|}\right)$$

Zauważmy, że wartość współczynnika podziału zwiększa się wraz ze zwiększaniem liczby podzbiorów, czyli współczynnik zysku jest tym mniejszy, im mniejsze są uzyskane w jego wyniku podzbiory.

Drugą popularną miarą nieuporządkowania zbiorów, zastosowaną m.in. w algorytmach CART i SPRINT, jest indeks podziału Giniego. W pierwszej kolejności wyznacza się wskaźnik nieczystości *Giniego*

$$gini = 1 - \Sigma_{i=1}^{n} P_i^2$$

W klasyfikatorach binarnych, w których zmienna wyjściowa przyjmuje dwa stany, wzór przyjmuje postać

$$gini = 1 - P_{pos}^2 - P_{neg}^2$$

gdzie P_{pos} jest prawdopodobieństwem przynależności obserwacji do klasy pozytywnej, a P_{neg} – prawdopodobieństwem przynależności obserwacji do klasy negatywnej. Indeks podziału *Giniego* jest ważoną sumą wskaźników nieczystości *Giniego*

$$gini_{split} = \Sigma_i \frac{|S_i|}{|D|} gini(S_i)$$

w przypadku klasyfikatorów binarnych obu uzyskanych w wyniku podziału podzbiorów

$$gini_{split}(D1, D2) = \frac{|D_1|}{|D|} gini(D_1) + \frac{|D_2|}{|D|} gini(D_2)$$

Najlepszy jest podział o najmniejszej wartości indeksu podziału *Giniego*. Ta metoda oceny podziałów może być stosowana zarówno do ciągłych, jak i dyskretnych zmiennych wejściowych.

Niezależnie od zastosowanego kryterium, konstruowanie drzew decyzyjnych metodą wybierania podziału najlepszego w danym momencie (tj. pozwalającego optymalnie podzielić przypadki przypisane do danego podzbioru) może prowadzić do otrzymania suboptymalnych modeli. Drzewa decyzyjne są też podatne na nadmierne dopasowanie. Oba te problemy rozwiązuje się, łącząc drzewa decyzyjne ze sobą. Otrzymane w ten sposób kombinacje modeli cechuje większa dokładność niż pojedynczych klasyfikatorów, mniejsze ryzyko nadmiernego dopasowania i większa odporność na błędne i zaszumione dane treningowe.

Kombinacje modeli uczenia maszynowego są względnie nowym pomysłem. Pierwsze metody kombinowania modeli zostały opracowane w latach 90. XX w. Podejście to zyskało na popularności

w pierwszej dekadzie XXI w. Dzisiaj pojedyncze drzewa decyzje nie są już w praktyce używane, bo zostały całkowicie zastąpione przez ich kombinację, takie jak lasy losowe i wzmocnione drzewa decyzyjne.

Wzrost dokładności kombinacji klasyfikatorów najprościej można wyjaśnić na przykładzie klasyfikacji binarnej, gdy każdy ze stanów zmiennej wyjściowej jest tak samo prawdopodobny. Skoro prawdopodobieństwo każdego stanu zmiennej wyjściowej jest takie samo, to jeżeli tylko trafność klasyfikatorów bazowych jest większa niż 50%, a poszczególne klasyfikatory są od siebie niezależne, czyli ich błędy nie są ze sobą skorelowane, model popełni błąd tylko wtedy, gdy ponad połowa klasyfikatorów bazowych błędnie sklasyfikuje przykład. Czyli kombinacja modeli będzie dokładniejsza od wchodzących w jej skład modeli bazowych.

Metody kombinowania modeli mogą polegać na modyfikowaniu:

- danych treningowych – dodatkowe, używane przez poszczególne klasyfikatory zbiory danych treningowych są tworzone przez wielokrotne losowanie ze zwracaniem, dzięki temu liczba przykładów użytych do uczenia poszczególnych modeli jest taka sama;
- listy zmiennych – dane treningowe poszczególnych klasyfikatorów zawierają jedynie wybrane (w najprostszym przypadku losowo) zmienne oryginalnego zbioru danych treningowych;
- algorytmu eksploracji danych – poszczególne klasyfikatory są uczone przy użyciu tego samego zbioru danych treningowych lub wylosowanych z niego prób, ale przy użyciu różnych albo różnie skonfigurowanych algorytmów.

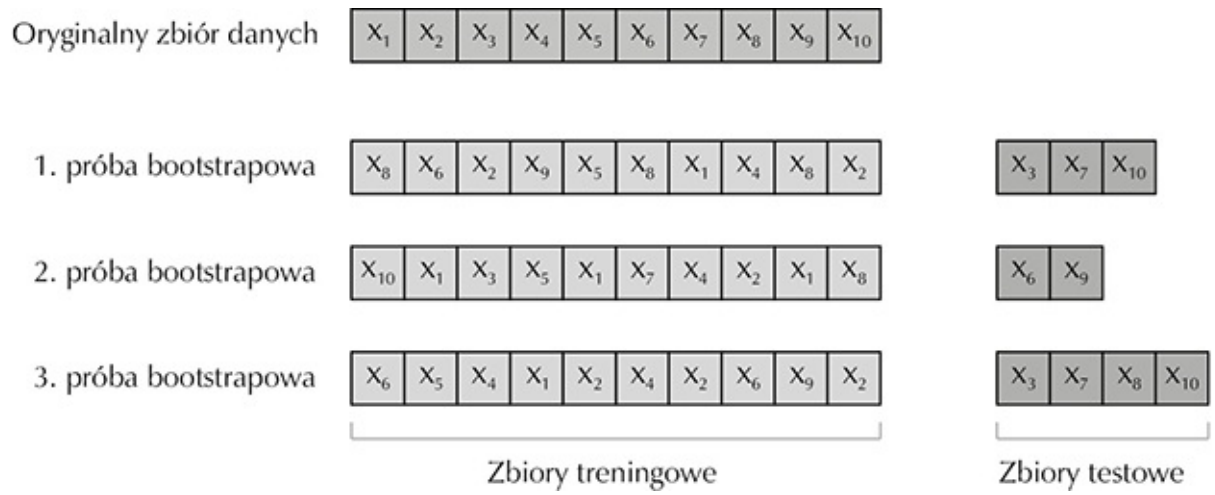
W kolejnych punktach przedstawimy poszczególne metody kombinowania modeli złożonych, zaczynając od metody agregacji bootstrapowej i wzmacniania. Następnie złączymy różne modele i uśrednimy ich wyniki.

Lasy losowe

Model lasów losowych jest konstruowany metodą agregacji bootstrapowej (*Bagging*). Polega ona na wielokrotnym losowaniu ze zwracaniem przykładów treningowych. Jeżeli takie losowanie powtórzymy *k* razy, otrzymamy *k* zbiorów treningowych, w których część przykładów będzie się powtarzać. Jeżeli wielkość oryginalnego zbioru danych treningowych *n* była wystarczająco duża, to podzbiory będą zawierały średnio 63% przypadków oryginalnego zbioru. Oszacowanie to wynika z tego, że prawdopodobieństwo wylosowania elementu podczas losowania ze zwracaniem do próbki o liczności *n* wynosi

$$1 - \left(1 - \frac{1}{n}\right)n$$

Dla wystarczająco dużych wartości *n* wynosi ono w przybliżeniu $1 - e^{-1}$, czyli 0,632. Wybrane przykłady zostaną użyte do uczenia poszczególnych drzew decyzyjnych, a pozostałe 37% przykładów do ich oceny (rys. 8.7).

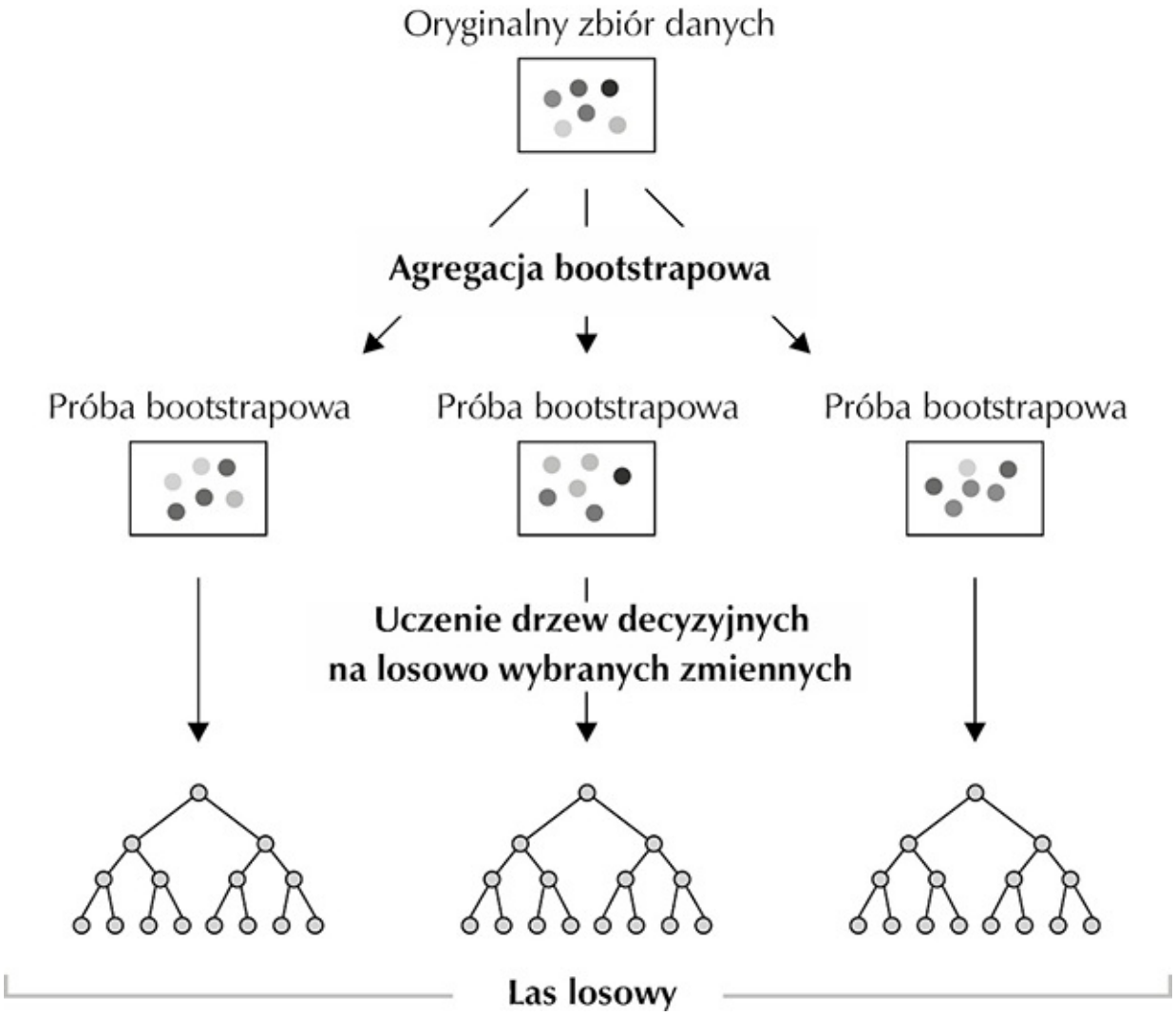


Rysunek 8.7. Każda próba bootstrapowa liczy tyle samo wierszy co oryginalny zbiór danych, ale zawiera powtarzające się przykłady, co znaczy, że niektóre przykłady nie zostały do niej wybrane

Działanie lasów losowych (*Random Forests*) polega na klasyfikacji przykładów przy użyciu wielu drzew decyzyjnych. Każde z tych drzew jest uczone na próbie bootstrapowej. Wiemy, że błąd kombinacji modeli będzie mniejszy od błędów poszczególnych modeli składowych, jeżeli będą spełnione dwa warunki: trafność poszczególnych klasyfikatorów musi być lepsza od trafności

klasyfikatora losowego oraz modele te muszą być niezależne.

Losowanie przykładów treningowych metodą agregacji bootstrapowej nie wystarcza do spełnienia warunku niezależności poszczególnych klasyfikatorów bazowych. Zauważmy, że próby bootstrapowe są do siebie statystycznie podobne, ale nie są identyczne. Lasy losowe mogą liczyć setki drzew decyzyjnych i nauczenie każdego z nich na 63% losowo wybranych przykładów spowodowałoby, że cześć klasyfikatorów bazowych byłaby bardzo do siebie podobna albo nawet byłyby one swoimi kopiami. Żeby zapewnić niezależność poszczególnych drzew decyzyjnych, każde z nich jest uczone i oceniane na losowo wybranych zmiennych, czyli poszczególne klasyfikatory są uczone na wylosowanych zmiennych wybranych do próby bootstrapowej przykładów (rys. 8.8).



Rysunek 8.8. Taki sposób konstruowania lasów losowych powoduje, że początkowo ich jakość się poprawia wraz z dodawaniem kolejnych drzew decyzyjnych, ale po przekroczeniu liczby 10–30 dodawanie kolejnych modeli składowych ma niewielki, a często zerowy wpływ na jakość całego modelu

Wynikowa predykcja zwracana przez model lasów losowych jest ustalana drogą głosowania większościowego poszczególnych modeli bazowych. W wersji zaimplementowanej w funkcji `rxFastForest` głosowanie polega na scaleniu prawdopodobieństw zwróconych przez modele bazowe w taki sposób, żeby scalone prawdopodobieństwa miały rozkład zbliżony do normalnego. W efekcie poprawia się stabilność predykcji, bo mniej prawdopodobne wyniki klasyfikacji zostają odrzucone w głosowaniu. Z kolei uczenie poszczególnych drzew decyzyjnych na próbach bootstrapowych zawierających tylko część zmiennych

zmniejsza ryzyko przeuczenia modelu i zwiększa szansę wykrycia nieliniowych korelacji między zmiennymi wejściowymi i ich kombinacjami a zmienną wyjściową.

Warto w tym miejscu podkreślić, że kombinowanie modeli metodą agregacji bootstrapowej nie jest metodą uniwersalną. Ponieważ wszystkie próby bootstrapowe są statystycznie do siebie podobne, algorytmy uczenia maszynowego bazując na statystykach danych treningowych, takie jak modele probabilistyczne, nic nie zyskają na wielokrotnym powtórzeniu uczenia. Metoda agregacji bootstrapowej doskonale sprawdza się w kombinowaniu niestabilnych modeli, takich jak drzewa decyzyjne, bo ten algorytm jest wrażliwy na pojedyncze przykłady.

Proces uczenia modelu lasów losowych przebiega następująco:

- z danych treningowych jest losowana próba bootstrapowa, czyli wybiera się z powtórzeniami *n* przykładów;
- próba bootstrapowa jest używana do uczenia pojedynczego drzewa decyzyjnego; sposób konstruowania drzewa nie różni się od standardowego, z jednym istotnym wyjątkiem – dla każdego tworzonego drzewa decyzyjnego jest wybieranych losowo *m* atrybutów z listy wszystkich dostępnych atrybutów; oznacza to, że poszczególne węzły utworzonego w ten sposób drzewa decyzyjnego będą analizowały zależności między losowym podzbiorem atrybutów wejściowych a atrybutem wyjściowym;
- nauczone drzewo decyzyjne jest oceniane za pomocą przypadków niewybranych do próby bootstrapowej; mierzone są m.in.: trafność predykcji i wpływ poszczególnych atrybutów wejściowych na trafność predykcji;
- powyższe operacje powtarzane są *k* razy;
- odpowiedzi na zapytania predykcyjne (wyniki zwracane przez model) są ustalane przez głosowanie przy użyciu wchodzących w jego skład drzew

decyzyjnych, na przykład, jeżeli 67% drzew decyzyjnych zaklasyfikuje obserwację do klasy pozytywnej, wynikiem będzie pozytywna klasyfikacja obserwacji.

Model lasów losowych utworzymy przy użyciu funkcji `rxFastForest`. Jej wymaganymi parametrami są `formula` i `data`. Pierwszy pozwala przekazać formułę ucząca, drugi – wskazać zbiór treningowy.

```
random_forest_model <- MicrosoftML::rxFastForest(,
                                data = train_df,
                                type = "binary",
                                numTrees = 10,
                                numLeaves = 15,
                                minSplit = 15,
                                exampleFraction = 0.6,
                                featureFraction = 0.7,
                                splitFraction = 0.6)
```

```
Not adding a normalizer.
Making per-feature arrays
Changing data from row-wise to column-wise
Beginning processing data.
Rows Read: 20631, Read Time: 0, Transform Time: 0
Beginning processing data.
Processed 20631 instances
Binning and forming Feature objects
Reserved memory for tree learner: 536520 bytes
Starting to train ...
Training calibrator.
Beginning processing data.
Rows Read: 20631, Read Time: 0.001, Transform Time: 0
Beginning processing data.
Elapsed time: 00:00:00.2063121
```

Parametr `type` został użyty do jawnego wskazania typu modelu – jest to klasyfikator binarny. Liczba drzew decyzyjnych została ograniczona do 10, każde z tych drzew zawiera do 15 węzłów najniższego poziomu (liści), z których każdy reprezentuje co najmniej 15 przykładów. Próby bootstrapowe zawierają

60% przykładów (`exampleFraction = 0.6`). Poszczególne drzewa decyzje uczone są na 70% zmiennych (`featureFraction = 0.7`), a zmienna podziałowa poszczególnych węzłów jest wybierana z 60% zmiennych wylosowanych dla danego drzewa (`splitFraction = 0.6`).

Najważniejsze informacje o nauczonym modelu zwraca funkcja `summary`.

```
summary(random_forest_model)
```

```
Call:
MicrosoftML::rxFastForest(formula = formula, data = train_df,
     type = "binary", numTrees = 10, numLeaves = 15, minSplit = 15,
     exampleFraction = 0.6, featureFraction = 0.7, splitFraction = 0.6)

Predictor for: label1~a11+a4+a15+a21+a17+a3+a20+a2+a12+a7+s11+s4+s12+s7+s15+s21+s20+s17+
s2+a8
Data: train_df

Per-feature gain:
a11: 1
a21: 0.3473681
a3: 0.3413941
a4: 0.1918349
a2: 0.1484739
a15: 0.1440899
a17: 0.1364433
a8: 0.1268276
a12: 0.1118301
a20: 0.07041322
s21: 0.0460737
s15: 0.0420381
s11: 0.03559432
s2: 0.02538886
s12: 0.02284951
s4: 0.02228593
s17: 0.0183893
```

Tak jak poprzednio, do oceny modelu zostały użyte dane testowe.

```
rf_model_prediction <- rxPredict(modelObject = random_forest_model,
                                 data = test_df,
                                 extraVarsToWrite = "label1",
                                 overwrite = TRUE)

rf_model_metrics <- evaluate_classifier(actual = "label1", predicted = "PredictedLabel",
                                        data = rf_model_prediction)

rf_model_metrics$Metrics[c(1,5)]
rxAuc(roc(data = rf_model_prediction,
          observed = "label1",
          predicted = "Probability.1"))
```

```
0.9     0.7014925
0.9354667
```

Trafność na poziomie 90%, współczynnik Kappa równy 0,7 i obszar pod krzywą wynoszący 0,93 oznaczają, że ten model lasów losowych jest gorszy od wcześniej zbudowanych modeli regresji logistycznej i sztucznych sieci neuronowych.

Wzmocnione drzewa decyzyjne

Kombinowanie modeli metodą agregacji bootstrapowej jest proste i skalowalne, ale mało efektywne. Wymaga ona uczenia wielu niezależnych od siebie modeli, z których każdy rozwiązuje ten sam problem. Natomiast wzmacnianie (ang. *boosting*) polega na uczeniu modeli na błędach ich poprzedników.

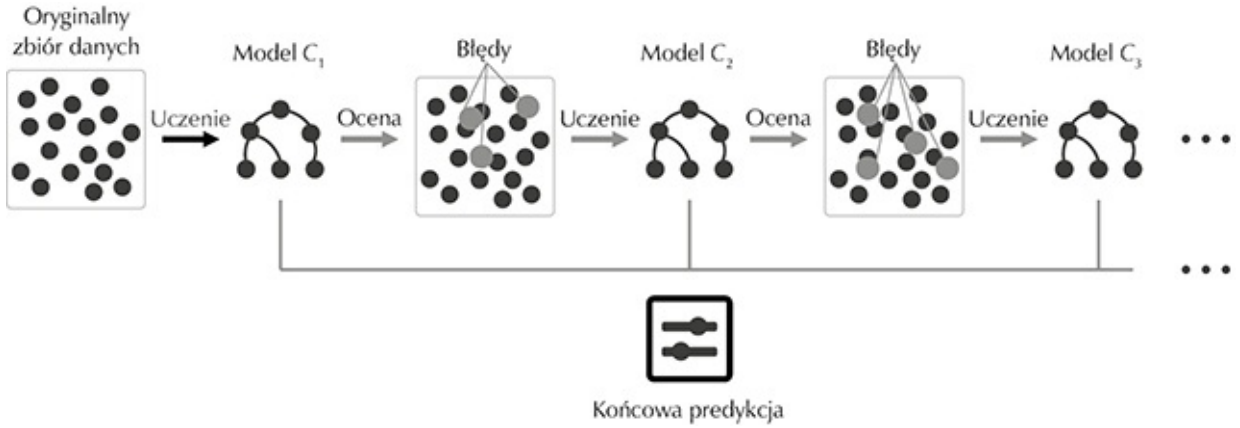
Ta wyjątkowo skuteczna metoda kombinowania modeli predykcyjnych działa według następującego schematu. Przykładom przypisuje się początkowe, losowe, ale takie same wagi. Uczy się pierwszy model. Następnie wagi błędnie sklasyfikowanych przykładów zostają zwiększone, co kończy rundę uczenia. Uczony w kolejnej rundzie model uwzględnia bieżące wagi przykładów, czyli uczy się na błędach swojego poprzednika. Po zakończeniu wszystkich rund końcowa predykcja jest ustalana metodą głosowania, przy czym waga głosu poszczególnych modeli zależy od ich jakości – im lepszy model, tym większy

ma wpływ na wynik głosowania.

Metoda wzmacniania może być używana do łączenia różnych modeli predykcyjnych. Jeżeli jakiś algorytm uczenia maszynowego nie uwzględnia wag przykładów, faworyzowanie błędnie sklasyfikowanych przykładów polega na uwzględnianiu wag podczas wybierania przykładów do kolejnego zbioru treningowego. Mamy więc do czynienia z modyfikacją metody wybierania próby bootstrapowej polegającą na zwiększaniu prawdopodobieństwa wylosowania błędnie sklasyfikowanych przykładów. W tej wersji metody wzmacniania:

- przypadkom są przypisywane losowe wagi; początkowe wagi wszystkich przypadków w zbiorze danych treningowych D są takie same;
- metodą losowania ze zwracaniem jest wybierana próba D_1*; p*rzy losowaniu przykładów uwzględnia się wagi, ale w związku z tym, że dla pierwszego zbioru treningowego są one takie same dla wszystkich przypadków, losowanie odbywa się z rozkładu jednostajnego;
- model C_1 zostaje nauczony na wybranych do próby D_1 przykładach i oceniony na pozostałych, niewybranych do tej próby przykładach;
- na podstawie predykcji modelu C_1 są zmieniane wagi przykładów; wagi błędnie sklasyfikowanych przypadków są mnożone przez ustalony współczynnik uczenia, a wagi przypadków poprawnie sklasyfikowanych są zmniejszane tak, aby suma wszystkich wag wyniosła 1;
- metodą losowania ze zwracaniem jest wybierana próba D_2; prawdopodobieństwa wylosowania do niej poszczególnych przypadków są już różne, ponieważ uwzględniane są bieżące wagi przypadków;
- model C_2 zostaje nauczony na wybranych do próby D_2 przykładach i oceniony na pozostałych, niewybranych do tej próby przykładach;
- na podstawie predykcji modelu C_2 są zmieniane wagi przykładów.

Powyższe kroki powtarza się, aż do utworzenia *k* modeli bazowych. Końcowa predykcja jest ustalana drogą przez głosowanie modeli składowych. Im mniejszy błąd modelu bazowego, tym większy jego wpływ na wynik głosowania (rys. 8.9).



Rysunek 8.9. Metoda wzmacniania modeli uczenia maszynowego przypomina stosowaną w szkołach metodę powtarzania zadań, których rozwiązanie sprawiło uczniom najwięcej problemów

Metody wzmacniania modeli predykcyjnych są przedmiotem intensywnych badań. Pierwszą praktyczną implementacją tej metody był opracowany w 1996 r. przez Yoava Freunda i Roberta Schapire'a algorytm *AdaBoost* (*Adaptive Boosting*). Ustawia ona ważność poszczególnych klasyfikatorów (modeli bazowych) na podstawie ich błędów klasyfikacji. Jeżeli jakość klasyfikatora jest gorsza niż modelu losowego, jest on usuwany z modelu. W przeciwnym razie jest obliczana jego ważność (im mniejszy procent błędów, tym większa ważność klasyfikatora). Tak obliczona ważność β jest używana do aktualizowania wag przypadków treningowych.

```
AdaBoost (C, Klasyfikator, k)
C: zbiór treningowy {(x1, y1), … (xn, yn)}
Klasyfikator: uczenie (S, wagi)
k: liczba rund
Dla wszytkich i∈C:w1(i) = 1/n
For t =1,2, … , k do
        For all i: pr(i) = wr(i)/ ∑i wr(i)
        hr = Klasyfikator(C, Pr)
        errr = ∑i pr(i) 1[hri ≠ y1]
        If errr>0,5 then
          K = r -1
   Exit
βr = errr /(1- errr)
Dla wszytkich i: wr+1(i) = wr(i)βr^1-1[hr xi ≠ y1]
h(x) = ∑(log(1/βr)1[hrx=y]
```

Proszę zwrócić uwagę, że algorytm *AdaBoost* zmienia wagi w taki sposób, że w poszczególnych rundach prawdopodobieństwa poprawnej i błędnej predykcji są takie same. Wynoszą one po 50%, niezależnie od rozkładu zmiennej wyjściowej w oryginalnych danych.

Na początku XXI wieku Jerome Friedman opublikował algorytm GBM (*Gradient Boosting Machine*), znany również pod nazwą MART (*Multiple Additive Regression Trees*). Jest on uniwersalną metodą wzmacniania wykorzystującą optymalizację gradientu w przestrzeni szukanej funkcji, a nie jej parametrów. Wymaga ona trzech elementów: funkcji kosztu, która ma zostać zminimalizowana, słabego modelu predykcyjnego oraz modelu, który zostanie dodany do słabego modelu predykcyjnego w celu jego wzmocnienia.

Skoro minimum funkcji kosztu jest znajdowane przy użyciu gradientu, funkcja ta musi być różniczkowalna. W modelach regresji najczęściej używa się funkcji błędu średniokwadratowego, a w klasyfikatorach funkcji logistycznej.

Rolę słabego modelu predykcyjnego pełnią drzewa regresji. Model F_1 jest uczony na przykładach losowo wybranych ze zbioru treningowego i używany do oceny wszystkich przykładów

$$F_1(x) = y$$

Następnie przy użyciu wybranej funkcji kosztu są obliczane błędy (różnice między prawdziwymi a zwróconymi przez model F_1 wartościami). Kolejny model jest uczony przy użyciu tych samych zmiennych wejściowych, ale zmienna wyjściowa zostaje zastąpiona błędami pierwszego modelu

$$F_2(x) = y - F_1(x)$$

Do wyników predykcji nowego modelu są dodawane predykcje zwrócone przez pierwszy model

$$F_2(x) = F_1(x) + F_2(x)$$

Obliczane są błędy drugiego modelu i cały proces się powtarza, czyli trzeci model bazowy jest uczony na pozostałych błędach.

Wzmocnione drzewa decyzyjne zbudujemy przy użyciu funkcji `rxFastTrees`.

```
gradient_boosting_model <- MicrosoftML::rxFastTrees(formula,
                                    data = train_df,
                                    type = "binary",
                                    numTrees = 20,
                                    exampleFraction = 0.6,
                                    featureFraction = 0.9,
                                    learningRate = 0.01,
                                    unbalancedSets = FALSE,
                                    numLeaves = 10,
                                    minSplit = 5,
                                    splitFraction = 0.8)
```

```
Not adding a normalizer.
Making per-feature arrays
Changing data from row-wise to column-wise
Beginning processing data.
Rows Read: 20631, Read Time: 0, Transform Time: 0
Beginning processing data.
Processed 20631 instances
Binning and forming Feature objects
Reserved memory for tree learner: 321912 bytes
Starting to train ...
```

Trzy wymagane parametry funkcji `rxFastTrees` już znamy. Pozwalają one przekazać formułę uczącą, nazwę zbioru treningowego i określić typ uczonego modelu. Ponieważ w każdej rundzie jest uczone jedno drzewo decyzyjne, liczbę rund określa parametr `numTrees`. Kolejne dwa parametry pozwalają określić sposób generowania prób bootstrapowych. Procent przykładów wybranych do każdej próby ustala parametr `exampleFraction`, a procent wybranych zmiennych parametr `featureFraction`. Następne dwa parametry służą do optymalizacji procesu uczenia. Wielkość kroku wykonanego w kierunku najszybszego spadku gradientu określa parametr `learningRate`. Ustawienie parametru `unbalancedSets` spowoduje, że gradient będzie obliczany w sposób optymalny dla silnie niezrównoważonych danych. Ostatnie trzy parametry sterują budową poszczególnych drzew decyzyjnych. Liczbę liści określa parametr `numLeaves`, minimalną liczbę przykładów potrzebnych do utworzenia nowego węzła –

parametr `minSplit`, a procent zmiennych wybranych do każdego podziału drzewa – parametr `splitFraction`.

Podsumowanie nauczanego modelu zwraca funkcja `summary`. Warto zwrócić uwagę, że niektóre zmienne wejściowe nie występują w modelu. Proszę sprawdzić, że wybranie do każdego drzewa decyzyjnego wszystkich zmiennych spowoduje zmniejszenie liczby zmiennych faktycznie użytych w modelu.

```
summary(gradient_boosting_model)
```

```
Call:
MicrosoftML::rxFastTrees(formula = formula, data = train_df,
     type = "binary", numTrees = 20, numLeaves = 10, learningRate = 0.01,
     minSplit = 5, exampleFraction = 0.6, featureFraction = 0.8,
     splitFraction = 0.8, unbalancedSets = FALSE)

Predictor for: label1~a11+a4+a15+a21+a17+a3+a20+a2+a12+a7+s11+s4+s12+s7+s15+s21+s20+s17+
s2+a8
Data: train_df

Per-feature gain:
a11: 1
a4: 0.3975194
a15: 0.2858499
a2: 0.2729159
a20: 0.2338486
a17: 0.17057
a8: 0.127583
a21: 0.1179563
a3: 0.07445795
a7: 0.07360715
a12: 0.03036864
s11: 0.02554868
```

Tak jak zwykle, błąd testowy modelu policzymy, wykonując funkcję `rxPredict` dla zbioru danych testowych i dodając do wyniku odczytaną z tego zbioru zmienną `label1`.

```
gb_model_prediction <- rxPredict(modelObject = gradient_boosting_model,
                                 data = test_df,
                                 extraVarsToWrite = "label1")
```

Trafność modelu oraz współczynnik Kappa policzymy przy użyciu niestandardowej, opisanej w następnym rozdziale, funkcji `evaluate_classifier`.

```
gb_model_metrics <- evaluate_classifier(actual = "label1", predicted = "PredictedLabel",
                                        data = GB_model_prediction)
gb_model_metrics$Metrics[c(1,5)]
```

```
0.9     0.7014925
```

Do policzenia obszaru pod krzywą użyjemy funkcji `rxAuc` biblioteki `RevoScaleR`. Krzywa, pod którą obliczamy obszar, została wyznaczona przy użyciu niestandardowej, opisanej w następnym rozdziale funkcji `roc`.

```
rxAuc(roc(data = GB_model_prediction,
          observed = "label1",
          predicted = "Probability.1"))
```

```
0.916
```

Algorytm, który najlepiej poradził sobie z oszacowaniem liczby pozostałych cykli bezawaryjnej pracy silników ma problemy z klasyfikacją binarną tych silników. Proszę sprawdzić, czy zwiększenie liczby zmiennych wejściowych poprawi jakość lasów losowych i wzmocnionych drzew decyzyjnych.

8.1.5. Kombinacje modeli

Dotychczasowe próby wykazały, że model sztucznej sieci neuronowej dzięki temu, że nauczył się działać jak zbiór binarnych przełączników, okazał się najdokładniejszy. Przypomnijmy – po zwiększeniu czułości modelu przez obniżenie progu prawdopodobieństwa jego trafność wyniosła 0,93. Model ten był też bardzo stabilny. Jego wiarygodność zmierzona współczynnikiem *AUC*

wyniosła 97%. Sprawdźmy, czy połączenie regresji logistycznej z lasem losowym i wzmocnionymi drzewami da równie dobre rezultaty. Do połączenia różnych modeli raz jeszcze użyjemy funkcji `rxEnsemble`.

```
ensemble_model <- rxEnsemble(
  formula = formula,
  data = train_df,
  type = "binary",
  trainers = list(
    logisticRegression(),
    logisticRegression(l1Weight = 0.9,
         l2Weight = 0.9,
         normalize = "warn"),
    fastTrees(numTrees = 20,
                exampleFraction = 0.6,
                featureFraction = 0.9,
                learningRate = 0.01,
                unbalancedSets = FALSE,
                numLeaves = 10,
                minSplit = 5,
                splitFraction = 0.8),
    fastForest(numTrees = 10,
                 numLeaves = 15,
                 exampleFraction = 0.6,
                 featureFraction = 0.7,
                 splitFraction = 0.6,
                 minSplit = 15)),
  modelCount = 8,
replace = TRUE,
  combineMethod = "average")
```

Tym razem na liście kombinowanych modeli znalazły się dwa modele regresji logistycznej: wzmocnione drzewa decyzyjne i las losowy. Ponieważ liczba wszystkich modeli bazowych została ustalona na 8 (`modelCount = 8`), każdy z czterech modeli bazowych zostanie zduplikowany. Poszczególne modele są uczone na próbie bootstrapowej (`replace = TRUE`), a wyniki ich predykcji są uśredniane (`combineMethod = „average”`).

Wywołując funkcje `summary`, przekonamy się, że model rzeczywiście zawiera 8 partycji oraz że w każdym modelu bazowym – również w modelach, które są kopiami zdefiniowanych architektur – przydatność zmiennych jest inna. Wynika to z tego, że modele bazowe były uczone na próbach bootstrapowych, a więc każdy z ich miał szansę znaleźć inne zależności między zmiennymi.

```
summary(ensemble_model)
```

```
Call:
rxEnsemble(formula = formula, data = train_df, trainers = list(logisticRegression(),
     logisticRegression(l1Weight = 0.9, l2Weight = 0.9, normalize = "warn"),
     fastTrees(numTrees = 20, exampleFraction = 0.6, featureFraction = 0.9,
          learningRate = 0.01, unbalancedSets = FALSE, numLeaves = 10,
          minSplit = 5, splitFraction = 0.8), fastForest(numTrees = 10,
          numLeaves = 15, exampleFraction = 0.6, featureFraction = 0.7,
          splitFraction = 0.6, minSplit = 15)), type = "binary",
     modelCount = 8, replace = TRUE, combineMethod = "average")

Ensemble for: label1 ~ a11 + a4 + a15 + a21 + a17 + a3 + a20 + a2 + a12 + a7 + s11 +
s4 + s12 + s7 + s15 + s21 + s20 + s17 + s2 + a8
Data: train_df

Partition model 0 summary:
Linear Binary Classification Predictor non-zero weights
(Bias): -60.8363
s4: 16.91317
a4: 16.46147
s11: 16.06403
a11: 15.61881
…
```

```
Partition model 1 summary:
Linear Binary Classification Predictor non-zero weights
(Bias): -0.0001212567
a4: 0.3557779
a8: -0.286315
a3: 0.2696079
s7: -0.1376834
a7: -0.1312622
s12: -0.1242865
…
Partition model 2 summary:
Per-feature gain summary for the boosted tree ensemble:
a11: 1
a4: 0.6522582
a2: 0.2981837
a8: 0.1541964
a21: 0.126448
a17: 0.1181326
…
Partition model 3 summary:
Per-feature gain summary for the boosted tree ensemble:
a11: 1
a4: 0.3741843
a15: 0.3576885
a3: 0.3501975
a21: 0.1443636
a8: 0.1284143
…
Partition model 4 summary:
Linear Binary Classification Predictor non-zero weights
(Bias): -58.1573
s4: 17.16209
a4: 17.00809
s11: 16.04127
a11: 15.17532
s21: -11.86774
s20: -11.60254
…
Partition model 5 summary:
Linear Binary Classification Predictor non-zero weights
(Bias): -0.0001210359
a4: 0.3511919
a8: -0.2863541
a3: 0.2489938
s7: -0.1326467
```

```
a7: -0.1293668
s12: -0.1225746
…
Partition model 6 summary:
Per-feature gain summary for the boosted tree ensemble:
a11: 1
a15: 0.2753478
a20: 0.2511657
a4: 0.2253585
a17: 0.1475749
a8: 0.125852
a3: 0.1130002
a21: 0.09400624
…
Partition model 7 summary:
Per-feature gain summary for the boosted tree ensemble:
a11: 1
a15: 0.374107
a3: 0.3385314
a4: 0.1781312
a21: 0.1630563
a20: 0.1499637
…
```

Sprawdźmy trafność predykcji dla danych testowych.

```
ensemble_model_prediction <- rxPredict(modelObject = ensemble_model,
                                       data = test_df,
                                       extraVarsToWrite = "label1")

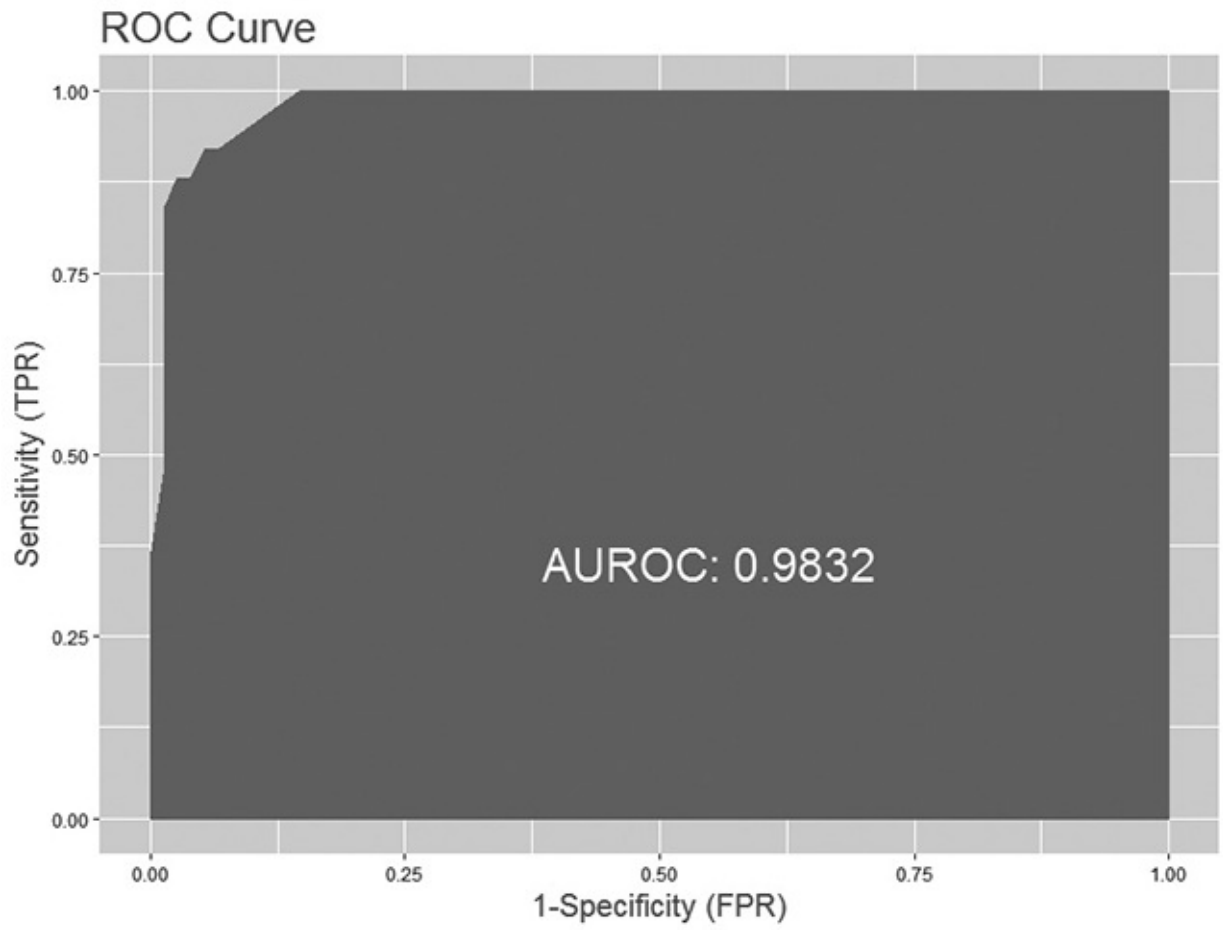
ensemble_model_metrics <- evaluate_classifier(actual = "label1",
                                       predicted = "PredictedLabel",
                                       data = ensemble_model_prediction)
ensemble_model_metrics$Metrics[c(1,5)]
```

```
0.9100000       0.7352941
```

Trafność modelu wynosi 91%, a współczynnik Kappa 0,73, co oznacza że kombinacja modeli nie jest dokładniejsza od najdokładniejszego modelu bazowego. Sprawdźmy jeszcze stabilność modelu, dla odmiany używając funkcji biblioteki `InformationValue` (rys. 8.10).

```
InformationValue::plotROC(ensemble_model_prediction$label1,
ensemble_model_prediction$Probability.1)
InformationValue::AUROC(ensemble_model_prediction$label1,
ensemble_model_prediction$Probability.1)
```

```
0.9832
```



Rysunek 8.10. Uśredniając wyniki predykcji ośmiu modeli, otrzymaliśmy model, który z bardzo dużą pewnością klasyfikuje przykłady do obu klas

Proszę sprawdzić, jak zmiana sposobu ustalania końcowej predykcji wpłynie na trafność i wiarygodność modelu. W pierwszej kolejności proszę użyć mediany (`combineMethod = „median”`), a następnie metody głosowania większościowego (`combineMethod = „vote”`).

8.1.6.

Porównanie modeli

Do porównania modeli użyjemy funkcji `model_factory`. Pozwala ona budować wszystkie opisane w tym punkcie klasyfikatory i trenować je na podanej liczbie zmiennych odczytanych ze wskazanej tabeli. Na przykład, żeby wytrenować modele na opisanych danych przy użyciu dziesięciu zmiennych wejściowych najsilniej skorelowanych ze zmienną wyjściową, należy wykonać poniższe instrukcje.

```
train_table_name <- "PredictiveMaintenance.train_Labels"
test_table_name <- "PredictiveMaintenance.test_Labels"
top_variables <-10
model_factory (train_table_name, test_table_name, top_variables)
```

Modele porównamy za pomocą uproszczonej wersji funkcji `evaluate_classifier`. Zwraca ona podstawowe metryki klasyfikatorów binarnych, w tym ich trafność. Jednak trafność, co zostało wyjaśnione w następnym rozdziale, nie nadaje się do oceniania klasyfikatorów, jeżeli dysponujemy niezrównoważonymi danymi. W takich przypadkach lepiej użyć *F*-miary, o czym przekonamy się, porównując modele nauczone na różnych danych.

Wyniki oceny modeli zapiszemy w tabeli `PredictiveMaintenance.Binary_metrics`. Wymaga to złączenia wyników poszczególnych modeli oraz opisania ich nazwą zastosowanego w modelu algorytmu uczenia maszynowego, typem użytych danych treningowych i liczbą zmiennych.

```
metrics_df <- rbind(logit_model_metrics, nn_model_metrics,
                        rf_model_metrics, GB_model_metrics,ensemble_model_metrics)
metrics_df <- as.data.frame(metrics_df)
rownames(metrics_df) <- NULL
models <- c("rxLogisticRegression on raw data",
                    "rxNeuralNet on raw data",
                    "rxFastForest on raw data",
                    "rxFastTrees on raw data",
                    "rxEnsemble on raw data")
metrics_df <- cbind(models, metrics_df)
models <- cbind(models, top_variables)
metrics_df <- cbind(models, metrics_df)
metrics_df$top_variables <- as.integer(trimws(metrics_df$top_variables))
```

Po wskazaniu docelowej tabeli i odpowiednim nazwaniu kolumn (musza one odpowiadać nazwom kolumn docelowej tabeli) możemy załadować do niej dane. Proszę pamiętać, żeby wyniki wszystkich kolejnych testów dopisać do tabeli, a nie nadpisać nimi poprzednich.

```
metrics_table <- RxSqlServerData(table = "PredictiveMaintenance.Binary_metrics",
                                        connectionString = connection_string)
colnames(metrics_df)[1] <- "Name"
colnames(metrics_df)[2] <- "Variables"
rxDataStep(inData = metrics_df,
             outFile = metrics_table,
             overwrite = TRUE)
```

Tak jak w przypadku modeli regresji, ograniczyłem się do porównania modeli nauczonych na 10 i 20 zmiennych odczytanych z tabel z opisanymi, wzbogaconymi i znormalizowanymi danymi. Przyjrzyjmy się jakości pięciu najlepszych i najgorszych modeli.

```
WITH ranking AS
       (SELECT *, ROW_NUMBER() OVER (ORDER BY [F-Score] DESC) as Rn
       FROM [PredictiveMaintenance].[Binary_metrics])
SELECT [Name], [Variables], [F-Score], Accuracy, Rn
FROM ranking
WHERE Rn <=5 OR Rn >25
ORDER BY [F-Score] DESC;
```

Name	Variables	F-Score	Accuracy	Rn
rxNeuralNet on enchanced data	20	0.844444444444444	0.93	1
rxNeuralNet on normalized data	10	0.818181818181818	0.92	2
rxLogisticRegression on enchanced data	20	0.818181818181818	0.92	3
rxNeuralNet on normalized data	20	0.818181818181818	0.92	4
rxLogisticRegression on normalized data	20	0.790697674418605	0.91	5
rxFastTrees on raw data	10	0.639455782312925	0.9838118509	26
rxEnsemble on raw data	10	0.637010676156584	0.9844227244	27
rxLogisticRegression on raw data	10	0.620338983050847	0.98289554062	28
rxNeuralNet on raw data	20	0.615384615384615	0.98281918142	29
rxLogisticRegression on raw data	20	0.594982078853047	0.98274282223	30

Zwraca uwagę fakt, że trafność modeli nauczonych na opisanych danych jest wyraźnie większa od dokładności pozostałych modeli, ale ich *F*-miara jest mniejsza. Wynika to z tego, że w 97,5% wierszy tabeli `PredictiveMaintenance.test_Labels` zmienna wyjściowa `label1` przyjmuje wartość 0, tymczasem w tabeli `PredictiveMaintenance.Test_Features` zmienna `label1` przyjmuje wartość 0 tylko w 75% wierszy.

```
SELECT [label1], 100. * count(*)/13096 AS pct
FROM [PredictiveMaintenance].[test_Labels]
GROUP BY [label1];

SELECT [label1], 1. * count(*) AS pct
FROM [PredictiveMaintenance].[Test_Features]
GROUP BY [label1];
```

label1	pct
0	97.464874
1	2.535125
0	75
1	25

Pełne wyjaśnienie odkładam do następnego rozdziału, podkreślając jedynie, że trafność jest miarą silnie zależną od rozkładu zmiennej wyjściowej i nie powinna być używana do oceny modeli, jeśli dane nie są zrównoważone.

Proszę też zauważyć, że niewrażliwa na stopień niezrównoważenia danych *F*-miara jest dużo niższa dla wszystkich modeli nauczonych na opisanych, ale niewzbogaconych w żaden sposób danych. Okazuje się, że wzbogacenie danych miało znacznie większy wpływ na jakość modeli od użytego w nich algorytmu uczenia maszynowego.

 Proszę samodzielnie porównać nauczone na różnych danych modele. Informacje na temat budowy i działania poszczególnych algorytmów uczenia maszynowego powinny pozwolić Państwu zrozumieć, czemu uczone na różnych danych modele są mniej lub bardziej dokładne. Proszę też spróbować zbudować model, którego *F*-miara przekracza 0,85.

Pora podsumować otrzymane wyniki i krótko scharakteryzować poszczególne algorytmy uczenia maszynowego.

Funkcje `rxLogisticRegression()` (R) i `rx_logistic_regression` (Python) pozwalają budować modele regresji logistycznej. Obie te funkcje są bardzo szybkie, a więc powinniśmy ich używać, jeżeli zależy nam na jak najkrótszym czasie uczenia modelu. Proces uczenia jest automatycznie optymalizowany przy użyciu algorytmu *BFGS*, co powoduje, że funkcje są łatwe w użyciu. Dane treningowe nie muszą zostać w całości załadowane do pamięci, jeśli parametr `trainThreads` ustawi się na wartość 1, a ilość pamięci potrzebnej do przechowywania poprzednio obliczonych gradientów można kontrolować parametrem `memorySize`. Dzięki temu obie funkcje mogą być używane do uczenia modelu na dużych zbiorach danych.

Funkcje `rxNeuralNet` (R) i `rx_neural_net` (Python) pozwalają używać sztucznych sieci neuronowych do klasyfikacji binarnej. Sztuczne sieci neuronowe są skomplikowane i podatne na nadmierne dopasowanie. Wymagają one dużych zbiorów treningowych, a czas ich uczenia jest dłuższy niż innych

modeli i gwałtownie rośnie wraz z dodawaniem kolejnych ukrytych węzłów.

Inaczej niż w modelach regresji, których błąd może zwiększać się do bardzo dużych wartości, maksymalny błąd klasyfikatorów wynosi 1. Dlatego konsekwencje znalezienia dużych wag mogą dla klasyfikatorów okazać się pozytywne. Chociaż duże wartości parametrów modelu świadczą najczęściej o jego nadmiernym dopasowaniu do przykładów treningowych, to końcowe predykcje są spłaszczane przy użyciu funkcji sigmoidalnej, co nie tylko niweluje skalę błędów, ale wręcz zwiększa pewność modelu co do jego wyników. Ponadto to, że działanie węzłów sztucznej sieci neuronowej o dużych wagach przypomina działania przełączników binarnych upraszcza samodzielne dostosowanie czułości modelu do konkretnego zadania. Dlatego funkcje te powinny być używane, jeżeli dysponujemy dużym zbiorem danych, możemy sobie pozwolić na długotrwały trening i zależy nam na jak największej dokładności modelu.

Funkcje `rxFastForest` (R) i `rx_fast_forest` (Python) pozwalają budować modele lasów losowych. Zastosowany algorytm regresji kwantylowej (ang. *quantile regression*) pozwala stosować je zarówno do regresji, jak i klasyfikacji binarnej. Ponieważ poszczególne drzewa decyzyjne są budowane równolegle, obie funkcje są szybkie, ale wymagają dużej ilości pamięci operacyjnej. Kombinacje drzew decyzyjnych cechuje mniejsze ryzyko nadmiernego dopasowania oraz większa trafność niż pojedynczych drzew. Nie należy jednak łączyć zbyt wielu drzew decyzyjnych – chociaż nie wiąże się to ze zwiększeniem ryzyka nadmiernego dopasowania lasu losowego, to wydłuża czas jego uczenia i zwiększa ilość potrzebnej pamięci operacyjnej.

Funkcje `rxFastTrees` (R) i `rx_fast_trees` (Python) pozwalają budować modele wzmocnionych drzew decyzyjnych. Zastosowany algorytm *GBM* powoduje, że kolejne drzewa są uczone na błędach ich poprzedników. Poszczególne drzewa decyzyjne są budowane sekwencyjnie. Ponieważ powinny być one względnie

małe (liczbę węzłów poszczególnych drzew ogranicza się w celu zmniejszenia ryzyka nadmiernego dopasowania modelu) i nie powinno ich być zbyt wiele (wraz ze zwiększaniem liczby drzew rośnie ryzyko nadmiernego dopasowania modelu), uczenie modeli wzmocnionych drzew decyzyjnych jest szybkie. Modele te cechuje bardzo duża trafność, co w połączeniu z prostą budowy i ograniczonymi wymaganiami w zakresie liczby przypadków treningowych przekłada się na ich popularność. Obie funkcje wczytują cały zbiór treningowy do pamięci, a więc ich użycie może wymagać przeniesienia uczenia na stronę wyposażonego w odpowiednią ilość pamięci serwera.

8.1.7. Wdrożenie modeli po stronie serwera SQL

Uczenie modeli na komputerze analityka ma liczne wady. Po pierwsze, jest wolne. Lokalny komputer, czy laptop, rzadko dysponują taką mocą obliczeniową co serwer. W dodatku takie uczenie wymaga przesłania z serwera danych treningowych, co dodatkowo wielokrotnie wydłuża jego czas. Po drugie, uczenie na komputerach analityków nie jest skalowalne. Serwery wyposażone w 512 GB pamięci RAM i więcej nie są niczym niezwykłym, podczas gdy ilość pamięci RAM w lokalnych komputerach i laptopach rzadko przekracza 32 GB. Po trzecie, jest niebezpieczne. Pobrane z serwera dane treningowe muszą być przesłane przez sieć i prawdopodobnie zostaną zapisane na lokalnym dysku komputera analityka. Komputery użytkowników, nie mówiąc już o laptopach, są znacznie bardziej narażone na ataki niż serwery. Dlatego uczenie modeli powinno się odbywać tam, gdzie znajdują się nasze dane, czyli po stronie serwera.

Stworzona w tym celu procedura składowana `PredictiveMaintenance.TrainBinaryClassificationModel` jest wywoływana z trzema parametrami: nazwą uczonego modelu, nazwą tabeli, w której znajdują się dane treningowe i liczbą zmiennych, które mają zostać użyte do uczenia.

Procedura zwraca nauczony, zserializowany model i dodatkowo wyświetla znalezioną formułę uczącą.

```
CREATE PROCEDURE [PredictiveMaintenance].[TrainBinaryClassificationModel]
@modelName varchar(50), @tableName varchar(100), @topVariables int,
@trained_model varbinary(max) OUTPUT
AS
BEGIN
…
```

W ramach procedury jest tworzona instrukcja odczytująca całą zawartość tabeli o podanej nazwie i wywoływana jest systemowa procedura `sp_execute_external_script`. To do niej będą przekazane wszystkie parametry wywołania, a odebrany zostanie nauczony model.

```
DECLARE @inquery NVARCHAR(max) = N'SELECT * FROM ' + @tableName;
   EXEC sp_execute_external_script @language = N'R',
                                        @script = N' …'
,@input_data_1 = @inquery
,@params = N'@model_name varchar(30), @top_variables int, @trained_model varbinary(max)
OUTPUT'
,@model_name = @modelName
,@top_variables = @topVariables
,@trained_model = @trained_model OUTPUT
END
```

Skrypt języka R składa się z trzech części. Pierwsza zawiera znane nam już instrukcje definiujące formułę ucząca na podstawie siły korelacji liniowej między zmiennymi. Jedyna różnica polega na jawnym przekształceniu zmiennej wyjściowej na czynnik, co zapobiega błędom przy konstruowaniu kombinacji modeli.

```
train_table$label1 <- as.factor(train_table$label1)
```

Druga, najdłuższa część to zagnieżdżona instrukcja warunkowa, która wybiera uczony model na podstawie przekazanej nazwy algorytmu.

```
if (model_name == "rxLogisticRegression") {
   model <- MicrosoftML::rxLogisticRegression(formula = formula,
                              data = train_table,
                              type = "binary",
                              l1Weight = 0.9,
                              l2Weight = 0.9,
                              maxIterations = 100,
                              optTol = 1e-07,
                              normalize = "warn",
                              memorySize = 100)
} else if (model_name == "rxNeuralNet") {
…
} else {
   model <- MicrosoftML::rxEnsemble(formula,
                              data = train_table,
                              type = "binary",
                              trainers = list(
                                           logisticRegression(),
                                                    logisticRegression(l1Weight = 0.9,
                                                             l2Weight = 0.9,
                                                             normalize = "warn"),
                                           fastTrees(numTrees = 20,
                                           exampleFraction = 0.6,
                                              featureFraction = 0.9,
                                              learningRate = 0.01,
                                              unbalancedSets = FALSE,
                                              numLeaves = 10,
                                              minSplit = 5,
                                              splitFraction = 0.8),
                                          fastForest(numTrees = 10,
                                               numLeaves = 15,
                                               exampleFraction = 0.6,
                                               featureFraction = 0.7,
                                               splitFraction = 0.6,
                                               minSplit = 15)),
                                    replace = TRUE,
                                    modelCount = 8,
                                    combineMethod = "average")
}
```

Na końcu nauczony model jest serializowany w trybie pozwalającym na wykonywanie natywnych zapytań predykcyjnych.

```
trained_model <- rxSerializeModel(model, realtimeScoringOnly = TRUE)'
```

Poniższe wywołanie procedury spowoduje zbudowanie modelu sztucznej sieci neuronowej. Model zostanie nauczony na 20 zmiennych odczytanych z tabeli ze znormalizowanymi danymi i zapisany w tabeli `PredictiveMaintenance.Models`.

```
DECLARE @model VARBINARY(MAX);
EXEC [PredictiveMaintenance].[TrainBinaryClassificationModel] 'rxNeuralNet',
'[PredictiveMaintenance].[train_Features_Normalized]', 20,
        @model OUTPUT;
INSERT INTO [PredictiveMaintenance].[Models] (model_name, model)
VALUES('rxNeuralNet binary classification', @model);
```

Żeby zbudować kombinację modeli i nauczyć ją na 20 wzbogaconych zmiennych, należy wykonać poniższe instrukcje.

```
DECLARE @model VARBINARY(MAX);
EXEC [PredictiveMaintenance].[TrainBinaryClassificationModel] 'rxEnsemble',
'[PredictiveMaintenance].[Train_Features]', 20,
        @model OUTPUT;
INSERT INTO [PredictiveMaintenance].[Models] (model_name, model)
VALUES('rxEnsemble binary classification', @model);
```

Zapytania predykcyjne do zapisanych w tabeli modeli wykonamy, używając procedury `sp_rxPredict`. Należy jej przekazać zserializowany model oraz zapytanie zwracające dane testowe. Model odczytamy z tabeli `PredictiveMaintenance.Models`. Ponieważ dane testowe muszą dokładnie odpowiadać danym treningowym, odczytamy z odpowiedniej tabeli 20 zmiennych wejściowych, a zmienną wyjściową skonwertujemy na typ tekstowy.

```
DECLARE @model varbinary(max) = ( SELECT model FROM [PredictiveMaintenance].[Models]
       WHERE model_name = 'rxEnsemble binary classification');
EXEC sp_rxPredict
       @model = @model,
       @inputData = N'SELECT CAST(label1 as varchar(100)) AS label1, s11 , s4 , s12 ,
s7 , s15,
       s21 , s20 , s17 , s2 , s3 , s8 , s13 , s9 , s6 , a9 , a6 , sd12 , sd6 , sd9 ,
sd7
       FROM [PredictiveMaintenance].[Train_Features]';
```

```
PredictedLabel    Score.1          Probability.1
1                  2.092926        0.8131049
0                 -7.059553        0.02556651
0                 -6.724531        0.0306649
1                  0.4787454       0.638516
1                  3.756077        0.9167542
0                 -1.46391         0.3738162
…
```

W związku z tym, że zdecydowaliśmy obniżyć czułość sztucznej sieci neuronowej, użycie tego modelu do predykcji wymaga samodzielnego przypisania obserwacji do jednej z dwóch klas na podstawie zwróconego przez model prawdopodobieństwa, z jakim należą one do klasy pozytywnej.

Bezpośrednie modyfikowanie wyników procedur składowanych nie jest możliwe. Serwer SQL pozwala jedynie zapisać wyniki wywołania procedur do tabel oraz zmiennych tabelarycznych. Zapisane w nich dane mogą już być odczytywane i dowolnie modyfikowane. Dlatego zanim wyślemy zapytanie predykcyjne do sztucznej sieci neuronowej, utworzymy zmienną tabelaryczną o strukturze odpowiadającej wynikowi zwróconemu przez procedurę `sp_rxPredict`. Procedurę tę wywołamy w ramach instrukcji `INSERT INTO`, a dane zapisane w zmiennej tabelarycznej odczytamy, wyliczając jednocześnie wartość zmiennej wyjściowej `PredictedLabel` przy użyciu niestandardowej wartości progowej.

```
 DECLARE @Predictions TABLE (
       [PredictedLabel] VARCHAR(256),
       [Score.1] DECIMAL (5,3),
       [Probability.1] DECIMAL (5,3));
DECLARE @model varbinary(max) = ( SELECT model FROM [PredictiveMaintenance].[Models] WHERE
model_name = 'rxNeuralNet binary classification');
INSERT INTO @Predictions
EXEC sp_rxPredict
       @model = @model,
       @inputData = N'SELECT CAST(label1 as varchar(100)) AS label1, s11 , s4 , s12 ,
s7 , s15,
       s21 , s20 , s17 , s2 , s3 , s8 , s13 , s9 , s6 , a9 , a6 , sd12 , sd6 , sd9 ,
sd7
       FROM [PredictiveMaintenance].[test_Features_Normalized]';
```

```
SELECT CASE
       WHEN [Probability.1]>0.4 THEN 1
       ELSE 0
       END AS [PredictedLabel],
       [Probability.1]
FROM @Predictions;
PredictedLabel   Probability.1
1                        0.997
0                        0.000
0                        0.001
1                        0.999
1                        0.999
1                        0.934
```

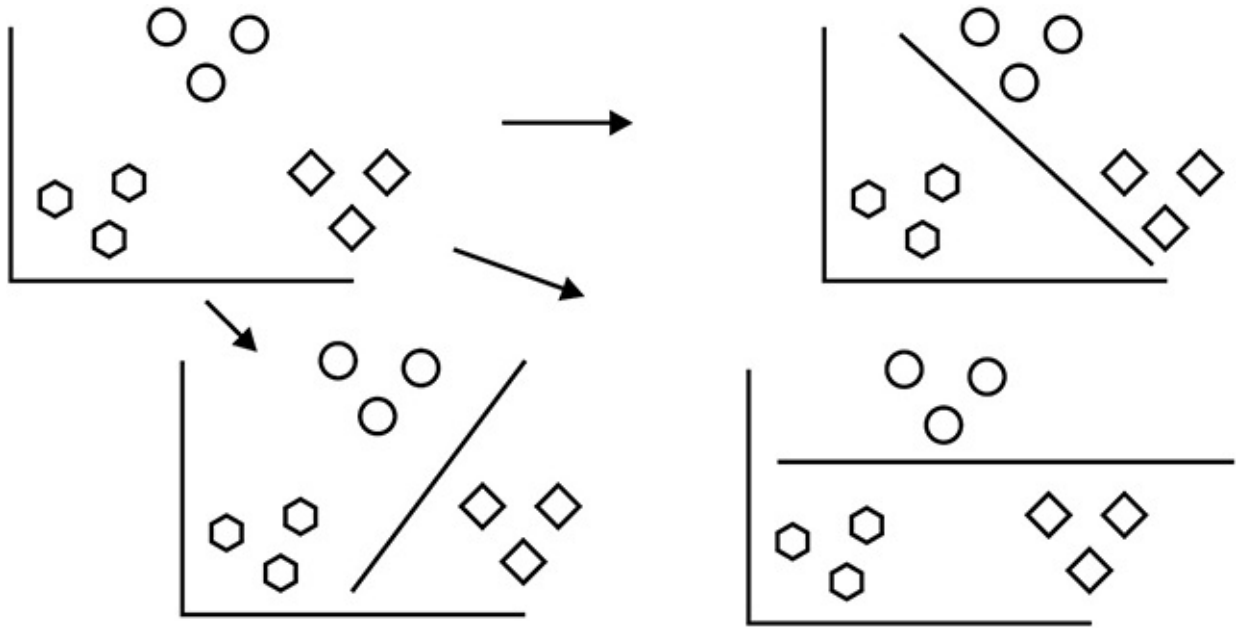
8.2.
Klasyfikacja wieloklasowa

Jeżeli zmienna wyjściowa przyjmuje więcej niż dwa stany, mamy do czynienia z klasyfikacją wieloklasową. Na przykład rozpoznanie widocznego na zdjęciu przedmiotu jest zadaniem dla klasyfikatora wieloklasowego. Tak samo jak określenie nastroju osoby na podstawie jej wyglądu, zidentyfikowanie autora tekstu czy zrozumienie sensu zdania. Przykłady treningowe mogą mieć postać tabelaryczną, graficzną czy dźwiękową, ale celem klasyfikacji wieloklasowej

zawsze jest znalezienie modelu, który nauczony na tych przykładach będzie potrafił przypisywać nowe obserwacje do jednej z możliwych klas.

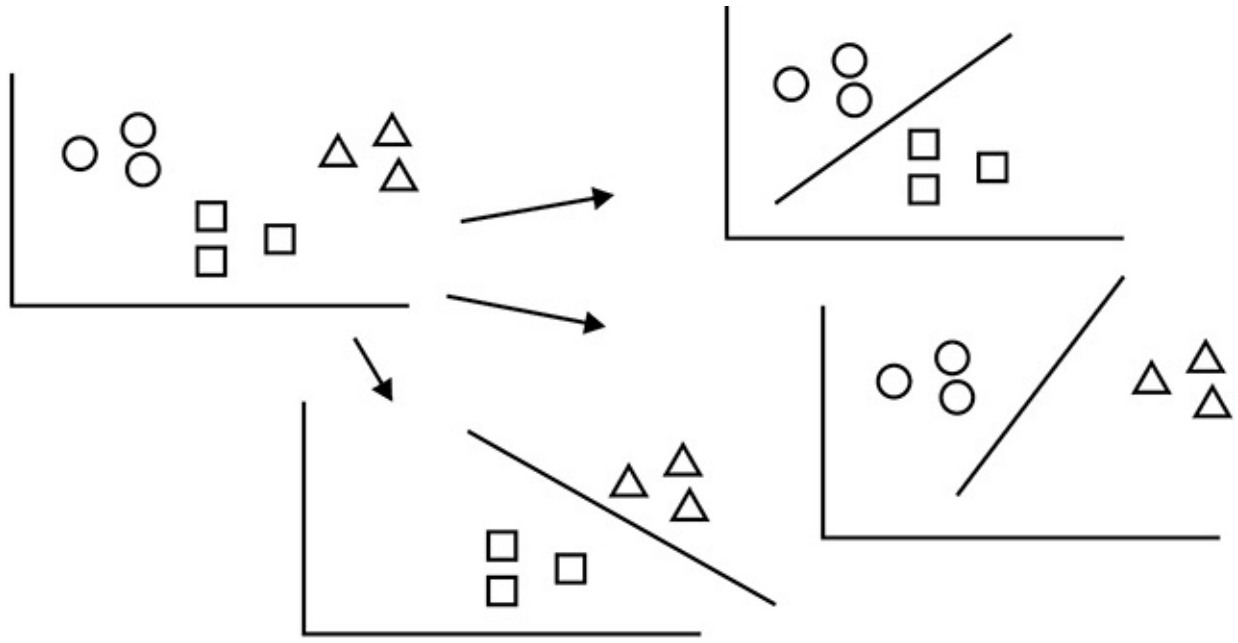
Klasyfikacja wieloklasowa jest uogólnieniem klasyfikacji binarnej. Możemy uznać przykłady należące do jednej z klas za przykłady pozytywne, a przykłady wszystkich pozostałych klas za negatywne, czyli porównać wybraną klasę z pozostałymi – tak działa metoda One-vs-All (*OVA*). Możemy też porównać ze sobą pary klas – tak działa metoda One-vs-One (*OVO*). Możemy też zbudować model, który określi prawdopodobieństwa przynależności przykładu do wszystkich możliwych klas. Przyjrzyjmy się tym trzem metodom.

Metoda One-vs-All pozwala użyć klasyfikatorów binarnych do klasyfikacji wieloklasowej. Polega ona na utworzeniu tylu klasyfikatorów binarnych, z iloma klasami mamy do czynienia. Dane służące do uczenia każdego z nich są dzielone na dwie klasy: jedną z nich jest klasa występująca w danych treningowych (np. *Czarne*), drugą – klasa powstała z połączenia pozostałych klas (np. *Nie czarne*). Klasyfikacja wieloklasowa polega na porównaniu wyników poszczególnych klasyfikatorów (w przypadku kolorów pierwszy z nich zwróci wynik *Czarne* lub *Nie czarne*, drugi *Czerwone* lub *Nie czerwone* itd.) i wybraniu tego, który daje najwyższy poziom ufności co do podjętej decyzji (rys. 8.11).



Rysunek 8.11. Dla *k* klas metoda OVA wymaga utworzenia *k* klasyfikatorów binarnych. Jednak jej całkowita złożoność wynosi km^2, gdzie *m* jest liczbą przykładów

Metoda One-vs-One również pozwala użyć klasyfikatorów binarnych do klasyfikacji wieloklasowej. Polega ona na utworzeniu wszystkich możliwych dwuelementowych kombinacji klas (liczbę klas oblicza się ze wzoru k_klas · (k_klas – 1)/2) i użyciu każdej z tych par do nauczenia odrębnego klasyfikatora. W przypadku kolorów będą to pary *Czarny–Czerwony*, *Czarny–Zielony* itp. Następnie każdy z klasyfikatorów głosuje na jedną z dwóch klas. Wygrywa klasa, która uzyska najwięcej głosów, czyli w wyniku klasyfikacji wieloklasowej uzyskuje się tę klasę, która najczęściej była wynikiem klasyfikacji przy użyciu poszczególnych klasyfikatorów binarnych (rys. 8.12).



Rysunek 8.12. Dla k klas potrzebujemy $(k^2 - k)/2$ klasyfikatorów, czyli liczba klasyfikatorów zwiększa się wraz z kwadratem liczby klas. Ale jeśli rozmiar zbioru jest m, liczba przypadków w poszczególnych klasach jest taka sama, a złożoność uczenia algorytmu jest kwadratowa, to uczenie jednego modelu ma złożoność $(2m/k)^2$, co daje całkowitą złożoność $(k^2 - k)4m^2/(2k^2) = 2(1–1/k)m^2$

Trzecia metoda pozwala przypisywać przykłady do różnych klas przy użyciu pojedynczego modelu klasyfikatora wieloklasowego. Wymaga ona zakodowania zmiennej wyjściowej metodą kodowania z gorącą jedynką (ang. *one-hot encoding*). Metoda ta została opisana w rozdziale 5 i przedstawiona na rysunku 5.2. Otrzymamy w ten sposób k binarnych zmiennych wyjściowych. Zmienna reprezentująca rzeczywistą klasę przykładu będzie miała wartość 1, pozostałe zmienne zostaną ustawione na 0.

Zadaniem klasyfikatora wieloklasowego jest określenie prawdopodobieństw, z jakimi poszczególne zmienne wyjściowe przyjmują wartość 1. Używa się do tego poznanej w poprzednim punkcie funkcji sigmoidalnej

$$Sigmoid(x) = \frac{e^x}{1 + e^x}$$

Dodając jednak do siebie wyniki funkcji sigmoidalnej, otrzymamy liczbę

z zakresu od 0 do nieskończoności. Takie wyniki są trudne do interpretacji, bo suma poszczególnych prawdopodobieństw będzie większa niż 1. Potrzebujemy funkcji, która przekształciłaby obliczone prawdopodobieństwa na liczby, które sumują się do 1, czyli zwróciła rozkład prawdopodobieństw. Taką funkcją jest funkcja softmax

$$Softmax(x) = \frac{e^x}{\sum e^x}$$

Porównanie obu funkcji jest zawarte w tabeli 8.1.

Skoro przykładowo suma wartości funkcji softmax wynosi 1, to im większa jest wartość jednej zmiennej wyjściowej, tym mniejsze będą wartości pozostałych zmiennych wyjściowych. Czyli im większe prawdopodobieństwo, że przykład należy do jednej klasy, tym mniejsze, że należy do innych klas. Co jest zgodne z naszą intuicją.

Tabela 8.1. W kolejnych kolumnach widzimy wartości zmiennych wyjściowych obliczone dla jednego przykładu oraz rezultaty ich przekształcenia za pomocą funkcji eksponencjalnej, sigmoidalnej i softmax

Klasa	Wynik	exp	sigmoid	softmax
czerwony	2.96	19.33	0.95080808	0.18
niebieski	3.47	32.09	0.969779138	0.30
żółty	2.57	13.08	0.928996811	0.12
zielony	2.94	18.83	0.949562464	0.18
pomarańczowy	–4.01	0.02	0.017811279	0.00
brązowy	0.25	1.28	0.561137332	0.01
purpurowy	2.94	18.90	0.949758301	0.18
czarny	0.62	1.86	0.650949337	0.02
suma	**11.74**	**105.39**	**5.98**	**1.00**

W dalszej części rozdziału użyjemy tej metody do zbudowania klasyfikatorów

wieloklasowych przypisujących urządzenia do jednej z trzech klas: urządzeń, których przewidywany czas bezawaryjnej pracy nie przekracza 15 cykli, urządzeń, które należy poddać przeglądowi w ciągu najbliższych 30 cykli i urządzeń, których prognozowany czas bezawaryjnej pracy jest dłuższy niż 30 cykli.

8.2.1. Przygotowanie danych

Przygotowanie danych do klasyfikacji wieloklasowej sprowadza się do dodania zmiennej wyjściowej `label2`, przyjmującej trzy wartości: 0 dla urządzeń, których czas bezawaryjnej pracy jest większy lub równy 30 cyklom, 1 dla urządzeń, których pozostały czas bezawaryjnej pracy mieścił się w zakresie 15–29 cykli i 2 dla urządzeń, którym pozostało mniej niż 15 cykli bezawaryjnej pracy. W tym celu do funkcji `data_label` została dodana poniższa instrukcja.

```
data$label2 <- ifelse(data$RUL <= 15, 2, data$label1)
```

Sprawdźmy rozkład nowododanej zmiennej w tabelach ze wzbogaconymi danymi treningowymi i testowymi.

```
SELECT [label2], COUNT(*) AS rn
FROM [PredictiveMaintenance].[Train_Features]
GROUP BY [label2];

SELECT [label2], COUNT(*) AS rn
FROM [PredictiveMaintenance].[Test_Features]
GROUP BY [label2];
```

```
label2  rn
0       17531
1       1500
2       1600

0       75
1       15
2       10
```

Okazuje się, że zmienna wyjściowa ma inny rozkład w danych treningowych niż w testowych, co znacząco utrudnia zbudowanie dobrego modelu.

Tak jak poprzednio, użyjemy wzbogaconych danych i postawimy hipotezę, że bezawaryjna praca silnika zależy od 20 zmiennych wejściowych najsilniej skorelowanych ze zmienną `label2`. Ponieważ danych jest na tyle mało, że wszystkie naraz zmieszczą się w pamięci RAM, załadujemy je do obiektów typu *data frame*.

```
train_table_name <- "PredictiveMaintenance.train_Features"
test_table_name <- "PredictiveMaintenance.test_Features"
top_variables <- 20

train_table <- RxSqlServerData(table = train_table_name,
                                    connectionString = connection_string,
                                    colInfo = list(label2 = list(type = "factor",
                                                                 levels = c("0",
"1", "2"))))
train_df <- rxImport(train_table)

test_table <- RxSqlServerData(table = test_table_name,
                                   connectionString = connection_string,
                                   colInfo = list(label2 = list(type = "factor",
                                                                levels = c("0",
"1", "2"))))
test_df <- rxImport(test_table)
```

Sposób znalezienia dwudziestu zmiennych wejściowych najsilniej skorelowanych z wyjściową jest taki sam jak w modelu regresji. Jedyna różnica polega na wyeliminowaniu zmiennych `RUL`, `label1`, `id` i `cycle_orig` oraz wskazaniu jako zmiennej wyjściowej zmiennej `label2`. Znaleziona formuła ucząca wygląda następująco.

```
train_vars <- rxGetVarNames(train_df)
train_vars <- train_vars[!train_vars  %in% c("RUL", "label1", "id", "cycle_orig")]
formula <- as.formula(paste("~", paste(train_vars, collapse = "+")))
correlation <- rxCor(formula = formula,
                         data = train_df,
                         transforms = list(label2 = as.numeric(label2)))
correlation <- correlation[, "label2"]
correlation <- abs(correlation)
correlation <- correlation[order(correlation, decreasing = TRUE)]
correlation <- correlation[-1]
correlation <- correlation[1:top_variables]
formula <- as.formula(paste(paste("label2~"),
                                  paste(names(correlation), collapse = "+")))
formula
```

```
label2 ~ a11 + a15 + a4 + a21 + a17 + a3 + a20 + a2 + a12 + a7 +
     s11 + s4 + s12 + s7 + s15 + s21 + s20 + s17 + s2 + a8
```

8.2.2. Regresja logistyczna

Jako pierwszy zbudujemy model regresji logistycznej. Użyjemy poznanej wcześniej funkcji `rxLogisticRegression`, zmieniając jedynie typ modelu.

```
logit_model <- MicrosoftML::rxLogisticRegression(formula = formula,
                                                 data = train_df,
                                                 type = "multiClass",
                                                 l1Weight = 0.95,
                                                 l2Weight = 0.95,
                                                 optTol = 1e-05,
                                                 normalize = "no")
```

```
Not adding a normalizer.
Beginning processing data.
Rows Read: 20631, Read Time: 0, Transform Time: 0
Beginning processing data.
Beginning processing data.
Rows Read: 20631, Read Time: 0.001, Transform Time: 0
Beginning processing data.
Warning: The number of threads specified in trainer arguments is larger than the
concurrency factor setting of the environment. Using 2 training threads instead.
LBFGS multi-threading will attempt to load dataset into memory. In case of out-of-memory
issues, turn off multi-threading by setting trainThreads to 1.
Beginning optimization
num vars: 63
improvement criterion: Mean Improvement
L1 regularization selected 63 of 63 weights.
```

Funkcja ta, tak samo jak pozostałe funkcje uczenia maszynowego, zwraca nauczony model. Do tej pory najważniejsze informacje o nauczonych modelach odczytywaliśmy funkcją `summary`.

```
summary(logit_model)
```

```
Call:
MicrosoftML::rxLogisticRegression(formula = formula, data = train_df,
     type = "multiClass", l2Weight = 0.95, l1Weight = 0.95, optTol = 1e-05,
     normalize = "no")

MultiClassLogisticRegression (MultiClassClassifierTrainer) for: label2~a11+a15+a4+a21+a17+
a3+a20+a2+a12+a7+s11+s4+s12+s7+s15+s21+s20+s17+s2+a8
Data: train_df

First 20 of 63 Non-zero Coefficients:
0+(Bias): 6.951371e-05
1+(Bias): -2.439695e-05
2+(Bias): -4.511702e-05
0+s4: -0.1799026
0+a4: -0.1696233
0+a8: 0.1642305
0+s7: 0.06516154
0+a7: 0.06409004
0+s12: 0.05900899
0+a12: 0.05809703
0+a3: -0.04788338
0+a2: 0.03121162
0+s2: 0.03064876
0+s17: -0.01274984
0+a17: -0.01234834
0+s20: 0.007920939
0+a20: 0.007707004
0+s21: 0.004778642
0+a21: 0.004651803
0+s11: -0.004478837
```

Jednak funkcja `summary` zwraca tylko pierwszych dwadzieścia znalezionych parametrów modelu. Tymczasem nasz model liczy 63 parametry. Dlaczego aż tyle, skoro model klasyfikatora binarnego zawierał 21 parametrów (parametry 20 zmiennych wejściowych plus parametr wyrazu wolnego)?

Podpowiedzią są nazwy parametrów. Proszę zwrócić uwagę, że pierwszym członem ich nazw jest stan zmiennej wyjściowej, a drugim nazwa zmiennej. Wynika to z tego, że zmienna wyjściowa `label2` zastała zakodowana metodą jeden-do-wielu i do modelu zostały dodane trzy zmienne binarne odpowiadające trzem stanom (0, 1 i 2) tej zmiennej. Czyli **model znalazł zależności między dwudziestoma zmiennymi wejściowymi a trzema zmiennymi wyjściowymi**.

Nauczone modele są zapisywane na listach. Na przykład informacje o modelu regresji logistycznej są zapisane na dziesięcioelementowej liście. Komplet informacji o strukturze obiektu, na przykład elementach listy, zwraca funkcja `str`.

```
str(logit_model)
```

```
List of 10
 $ formula       : chr "label2~a11+a15+a4+a21+a17+a3+a20+a2+a12+a7+s11+s4+s12+s7+s15+s21+
s20+s17+s2+a8"
 $ dataframeName: symbol data
 $ mamlCode      : chr "lab={label2} feat={a11} feat={a15} feat={a4} feat={a21} feat={a17}
feat={a3} feat={a20} feat={a2} feat={a12} feat={a7} feat={s1"| __truncated__
 $ outModelBin   : raw [1:3883] 50 4b 03 04 ...
 $ params        :List of 37
…
```

Proszę zwrócić uwagę, że niektóre elementy tej listy, na przykład element `params`, też jest listą. Znajomość struktury nauczonego modelu pozwoli nam odczytać wartości wszystkich znalezionych parametrów – są one zapisane w elemencie `summary` podlisty `cachedSummary`.

```
logit_model$cachedSummary
```

```
Call:
MicrosoftML::rxLogisticRegression(formula = formula, data = train_df,
     type = "multiClass", l2Weight = 0.95, l1Weight = 0.95, optTol = 1e-05,
     normalize = "no")

MultiClassLogisticRegression (MultiClassClassifierTrainer) for: label2~a11+a15+a4+a21+a17+
a3+a20+a2+a12+a7+s11+s4+s12+s7+s15+s21+s20+s17+s2+a8
Data: train_df

First 63 of 63 Non-zero Coefficients:
0+(Bias): 6.951371e-05
1+(Bias): -2.439695e-05
2+(Bias): -4.511702e-05
0+s4: -0.1799026
…
```

Nauczony model przetestujemy na danych testowych przy użyciu funkcji `rxPredict`.

```
 logit_model_prediction <- rxPredict(modelObject = logit_model,
                                     data = test_df,
                                     extraVarsToWrite = "label2")
options(scipen = 999)
tail(logit_model_prediction)
```

	label2	PredictedLabel	Score.0	Score.1	Score.2
95	0	0	0.9999304	0.00006309146	0.000006134364
96	0	0	0.9999752	0.00002331599	0.000001911649
97	0	0	0.9985181	0.00125504250	0.000227040786
98	0	0	0.9445327	0.03991737962	0.015549648553
99	0	0	0.9999819	0.00001705748	0.000001268704
100	1	1	0.1783964	0.43088042736	0.390722274780

Tabela predykcji zawiera klasę, do której została przypisana obserwacja oraz wynik funkcji softmax, czyli prawdopodobieństwa, z jakimi obserwacja należy do poszczególnych klas. Jak łatwo sprawdzić, ich suma zawsze wynosi 1.

```
round(rowSums(logit_model_prediction[,-c(1,2)]),3)
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 …
```

Sprawdźmy wartość *F*-miary dla poszczególnych klas i porównajmy je z prawdopodobieństwami losowego zgadywania.

```
logit_model_metrics <- evaluate_classifier(actual = logit_model_prediction$label2,
                                           logit_model_prediction$PredictedLabel)
logit_model_metrics$Metrics[c(4,20),]
```

```
F1              0.9375000 0.3809524 0.8421053
RandomGuessF1   0.4615385 0.2068966 0.1538462
```

Zbudowany model prawie dwukrotnie lepiej ocenia przypadki należące do klas 0 i 1 niż model losowy. W przypadku klasy 2 jakość modelu jest ponad pięciokrotnie lepsza od losowego zgadywania.

8.2.3. Sztuczne sieci neuronowe

Model sztucznej sieci neuronowej doskonale poradził sobie z klasyfikacją binarną. Zobaczmy, czy uda mu się uzyskać równie dobre wyniki klasyfikacji wieloklasowej. Do zbudowania modelu użyjemy funkcji `rxNeuralNet`, tym razem ograniczając maksymalne wielkości szukanych wag do przedziału od –2 do 2 (`maxNorm=2`).

```
nn_model <- MicrosoftML::rxNeuralNet(formula,
                                     data = train_df,
                                     type = "multiClass",
                                     normalize = "no",
                                     numIterations = 5,
                                     numHiddenNodes = 2,
                                     optimizer = adaDeltaSgd(),
                                     initWtsDiameter = 0.05,
                                     maxNorm=2)
Not adding a normalizer.
Beginning processing data.
Rows Read: 20631, Read Time: 0, Transform Time: 0
Beginning processing data.
Beginning processing data.
Rows Read: 20631, Read Time: 0, Transform Time: 0
Beginning processing data.
Using: SSE Math

***** Net definition *****
  input Data [20];
  hidden H [2] sigmoid { // Depth 1
    from Data all;
  }
  output Result [3] softmax { // Depth 0
    from H all;
  }
***** End net definition *****
Input count: 20
Output count: 3
Output Function: SoftMax
Loss Function: LogLoss
PreTrainer: NoPreTrainer
_______________________________________________________________________
Starting training...
Using Adadelta with decay 0.950000 and conditioner 0.000001
InitWtsDiameter: 0.050000
_______________________________________________________________________
Initializing 1 Hidden Layers, 51 Weights...
Estimated Pre-training MeanError = 1.895635
Iter:1/5, MeanErr=0.967274(-48.97%%), 13.20M WeightUpdates/sec
Iter:2/5, MeanErr=0.962886(-0.45%%), 13.74M WeightUpdates/sec
Iter:3/5, MeanErr=0.963261(0.04%%), 13.66M WeightUpdates/sec
Iter:4/5, MeanErr=0.962791(-0.05%%), 14.08M WeightUpdates/sec
Iter:5/5, MeanErr=0.963042(0.03%%), 12.24M WeightUpdates/sec
Done!
Estimated Post-training MeanError = 0.966021
```

Proszę zwrócić uwagę, że w warstwie wyjściowej sieci znajdują się trzy węzły, a użytą w tej warstwie funkcją aktywacyjną jest funkcja softmax.

Parametry modelu, czyli znalezione w ostatnim cyklu wartości wag węzłów można odczytać za pomocą funkcji `summary`.

```
summary(nn_model)
```

Po przetestowaniu modelu na danych testowych i odczytaniu jego *F*-miary okazało się, że dla klas 1 i 2 nie dało się ich policzyć.

```
nn_model_metrics$Metrics[c(4,20),]
```

	0	1	2
F1	0.8571429	NaN	NaN
RandomGuessF1	0.4615385	0.2068966	0.1538462

Oznacza to, że model nie zaklasyfikował ani jednej obserwacji do tych klas. Innymi słowy, wszystkie przykłady testowe zostały sklasyfikowane przez sztuczną sieć neuronową jako należące do klasy 0. Taki model jest z oczywistych względów nieprzydatny.

Proszę zwrócić uwagę, że prosta sieć neuronowa, tj. sieć z dwoma ukrytymi węzłami, której uczenie zostało przerwane po pięciu cyklach, zaklasyfikowała wszystkie przypadki testowe do klasy większościowej. Należy przypuszczać, że wszystkie przypadki treningowe również zostały przypisane do jednej, większościowej klasy. Żeby sprawdzić to przypuszczenie, policzymy *F*-miarę dla danych treningowych.

```
nn_model_prediction <- rxPredict(modelObject = nn_model,
                                 data = train_df,
                                 extraVarsToWrite = "label2")

nn_model_metrics <<- evaluate_classifier(actual = nn_model_prediction$label2,
                                         predicted =
nn_model_prediction$PredictedLabel)
nn_model_metrics$Metrics[c(4,20),]
```

	0	1	2
F1	0.9187674	NaN	NaN
RandomGuessF1	0.4788321	0.1193745	0.1258307

Nasze przypuszczenie okazało się słuszne. Model nie uczy się żadnych wzorców ukrytych w danych, zwracając zawsze takie same predykcje. Proszę samodzielnie sprawdzić, że zmiana architektury modelu, w tym zwiększenie liczby ukrytych węzłów, wydłużenie uczenia i zmiana metody optymalizacji procesu uczenia nie rozwiążą problemu. W tym przypadku jedynym rozwiązaniem jest zrównoważenie danych.

8.2.4. Klasyfikacja przez indukcję drzew decyzyjnych

Do zbudowania modeli partycjonujących użyjemy funkcji uczenia maszynowego z biblioteki `RevoScaleR`. Przede wszystkim dlatego, że poznane w poprzednim i w tym rozdziale funkcje biblioteki `MicrosoftML` nie pozwalają stosować lasów losowych i wzmocnionych drzew decyzyjnych do klasyfikacji wieloklasowej. Również po to, żeby zaprezentować Państwu te, bardzo rozbudowane, umożliwiające budowanie zarówno modeli regresji, jak i klasyfikacji, funkcje. Ostatnim, ale nie najmniej ważnym powodem jest chęć pokazania, że poznane w książce algorytmy uczenia maszynowego zostały zaimplementowane w różnych bibliotekach języków R i Python. Chociaż zasada ich działania jest taka sama, poszczególne implementacje algorytmów różnią się funkcjonalnością. Dlatego warto sprawdzić, które funkcje uczenia maszynowego

pozwalają ustawić potrzebne nam w danym projekcie hiperparametry.

Lasy losowe

Do zbudowania modelu lasów losowych użyjemy funkcji `rxDForest`. Szczegółowe informacje na jej temat uzyskamy, wykonując instrukcję

```
?RevoScaleR::rxDForest
```

Dwa obowiązkowe parametry funkcji, `formula` i `data`, pozwalają przekazać formułę uczącą i nazwę zbioru treningowego. Pozostałe parametry funkcji są opcjonalne. Przyjrzyjmy się możliwym do ustawienia hiperparametrom modelu lasu losowego.

```
forest_model <- RevoScaleR::rxDForest(formula = formula,
                                      data = train_df,
                                      method = "class",
                                      nTree = 20,
                                      mTry = 10,
                                      replace = TRUE,
                                      strata = "label2",
                                           seed = 5,
                                      maxDepth = 10,
                                      cp = 0.002,
                                      minSplit = 15,
                                      maxNumBins = 200,
                                      findSplitsInParallel = TRUE)
```

Zacznijmy od parametrów definiujących budowę całego modelu. Parametr `method` określa typ modelu: wartość `class` oznacza klasyfikację, `anova` – regresję. Liczbę drzew lasu losowego ustawia się parametrem `nTree`. Parametr `mTry` określa liczbę zmiennych wejściowych losowo wybranych dla każdego drzewa. Sposób losowania przykładów użytych do uczenia poszczególnych drzew określa parametr `replace`. Wartość `TRUE` oznacza losowanie ze zwracaniem (ang. *boosting*), wartość `FALSE` – losowanie bez zwracania. Procent losowanych

przykładów ustala parametr `sampRate`. Domyślnie jego wartość wynosi 1 dla losowania ze zwracaniem i 0,632 dla losowania bez zwracania. Ustawienie parametru `strata` oznacza losowanie przykładów metodą doboru warstwowego względem wskazanej zmiennej. W tym przypadku dane treningowe są silnie niezrównoważone, więc warto zadbać, aby każde drzewo decyzyjne było uczone na danych zawierających tyle samo przykładów poszczególnych klas. Parametr `seed` pozwala zainicjalizować podaną wartością generator liczb pseudolosowych. Jeżeli zostanie on ustawiony, powtórzenie uczenia da takie same wyniki.

Następnie skonfigurujemy hiperparametry poszczególnych drzew decyzyjnych. Maksymalną głębokość drzewa określa parametr `maxDepth`. Parametr `cp` pozwala ustawić minimalny zysk osiągnięty przez dodanie węzła do drzewa. Jeśli będzie on mniejszy, podział nie zostanie przeprowadzony. Minimalną liczbę przykładów potrzebną do dodania węzła ustawia parametr `minSplit`. Maksymalną liczbę przedziałów, na jakie mogą zostać podzielone zmienne numeryczne, definiuje parametr `maxNumBins`.

Ostatni parametr steruje procesem uczenia modelu. Ustawienie parametru `findSplitsInParallel` powoduje równoczesne szukanie optymalnych podziałów wszystkich węzłów znajdujących się na danym poziomie drzewa decyzyjnego, co z reguły skraca czas uczenia modelu.

Funkcja `summary` zwraca nazwy i typy elementów listy nauczonego modelu.

```
summary(forest_model)
```

```
Length   Class       Mode
ntree        1       -none-       numeric
mtry         1       -none-       numeric
type         1       -none-       character
forest      20       -none-       list
oob.err      2       data.frame   list
confusion   12       -none-       numeric
cutoff       3       -none-       numeric
params      65       -none-       list
formula      3       formula      call
call        14       -none-       call
```

Jak widać, model lasu losowego zawiera 10 elementów. Możemy je odczytać, używając operatora `$`. Na przykład liczbę drzew modelu odczytamy następująco:

```
forest_model$ntree
```

```
20
```

Błędy poszczególnych drzew decyzyjnych zmierzone jako procent błędnych predykcji dla danych walidacyjnych *OOB* (*out-of-bag*) oraz narastającą liczbę przykładów walidacyjnych zawiera element `oob.err`.

```
forest_model$oob.err
```

```
    oob.err      num.oob
1   0.07594095     7519
2   0.07404380    12236
3   0.07242503    15340
4   0.07050875    17317
5   0.07066185    18539
6   0.07085554    19321
7   0.06908393    19802
8   0.06791197    20129
9   0.06877863    20297
10 0.06710391    20431
11 0.06642282    20505
12 0.06731892    20544
13 0.06701332    20578
14 0.06781553    20600
15 0.06749151    20610
16 0.06714209    20613
17 0.06686061    20625
18 0.06705129    20626
19 0.06675069    20629
20 0.06689288    20630
```

Znajomość błędów walidacyjnych pozwala obliczyć macierz pomyłek modelu. Macierz pomyłek oraz wyprowadzone z niej miary jakości klasyfikatorów zostały opisane w następnym rozdziale.

```
forest_model$confusion
```

```
       Predicted
label 2    0             1        2        class.error
      0   17253          260      17       0.01580148
      1   627            631      242      0.57933333
      2   14             220      1366     0.14625000
```

Do wykonania zapytań predykcyjnych użyjemy, jak zwykle, funkcji `rxPredict`. Tym razem oprócz prawdziwych wartości zmiennej wyjściowej do wyników predykcji jawnie dodamy prawdopodobieństwo, z jakim dana obserwacja została przypisana do poszczególnych klas.

```
forest_model_prediction <- rxPredict(modelObject = forest_model,
                                     data = test_df,
                                     extraVarsToWrite = "label2",
                                     type = "prob")
tail(forest_model_prediction)
```

```
     X0_prob      X1_prob         X2_prob            label2_Pred     label2
95   0.9909241    0.008994882       0.00008105999      0               0
96   0.9909241    0.008994882       0.00008105999      0               0
97   0.9909241    0.008994882       0.00008105999      0               0
98   0.9803973    0.019418480       0.00018420800      0               0
99   0.9909241    0.008994882       0.00008105999      0               0
100 0.3575937    0.569425838       0.07298045085      1               1
```

Na końcu wyliczymy metryki modelu i porównamy jego *F*-miary z *F*-miarami modelu losowego.

```
forest_model_metrics <- evaluate_classifier(actual = forest_model_prediction$label2,
                                            predicted
=forest_model_prediction$label2_Pred)
forest_model_metrics$Metrics[c(4,20),]
```

```
                  0         1         2
F1              0.9367089 0.4347826 0.8421053
RandomGuessF1   0.4615385 0.2068966 0.1538462
```

Model lasów losowych równie dobrze co model regresji logistycznej klasyfikuje obserwacje do klas 0 i 2, ale lepiej radzi sobie z klasyfikowaniem obserwacji należących do klasy 1.

Wzmocnione drzewa decyzyjne

Model wzmocnionych drzew decyzyjnych zbudujemy, używając funkcji `rxBTrees`. Pochodzi ona z biblioteki `RevoScaleR` i udostępnia podobną listę parametrów co funkcja `rxDForest`. Opis wszystkich parametrów funkcji znajdziemy w pliku pomocy funkcji

```
?RevoScaleR::rxBTrees
```

Oprócz obowiązkowych parametrów `formula` i `data` pozwala ona określić budowę i działanie modelu wzmocnionych drzew decyzyjnych oraz, co nie zostało w książce pokazane, wybierać i przekształcać dane treningowe.

```
boosted_model <- RevoScaleR::rxBTrees(formula = formula,
                                      data = train_df,
                                      lossFunction = "multinomial",
                                      learningRate = 0.05,
                                      nTree = 25,
                                      mTry = 10,
                                      replace = FALSE,
                                      sampRate = 0.632,
                                      strata = "label2",
                                      seed = 215,
                                      cp = 0.01,
                                      minSplit = 15,
                                      minBucket = 8,
                                      maxNumBins = 300)
```

Zaczniemy od ustawienia hiperparametrów całego modelu. Parametr `lossFunction` pozwala wybrać funkcję kosztu. Determinuje ona typ modelu: wartość `multinomial` oznacza model klasyfikatora, `gaussian` – model regresji, a `bernoulli` – model regresji zwracający wartości z zakresu 0–1; współczynnik uczenia `learningRate` określa szybkość wzmacniania procesu uczenia. Liczbę drzew określa parametr `nTree`, a liczbę zmiennych wejściowych losowo wybranych do każdego drzewa parametr `mTry`. Tym razem przykłady treningowe dla poszczególnych drzew zostały wylosowane bez zwracania (`replace = FALSE`), każde drzewo było uczone na 63% przykładów wybranych metodą doboru warstwowego względem zmiennej wyjściowej `label2` (`sampRate = 0.632, strata = „label2”`).

Budową poszczególnych drzew decyzyjnych sterują następujące hiperparametry. Parametr `cp` określa minimalny zysk z podziału – im większa jego wartość, tym szybsze uczenie i mniejsze (zawierające mniej węzłów) drzewa decyzyjne.

Minimalną liczbę przykładów potrzebnych do utworzenia węzła określa parametr `minSplit`, a minimalną liczbę przykładów potrzebnych do utworzenia węzła na poziomie liści drzewa definiuje parametr `minBucket`. Parametr `maxNumBins` określa maksymalną liczbę przedziałów, na jakie mogą zostać podzielone zmienne numeryczne.

Strukturę listy nauczonego modelu zwraca funkcja `summary`. Kompletny model jest zapisany w elemencie `forest`.

```
boosted_model$forest
```

```
…
[[5]]
Number of valid observations:  13039
Number of missing observations:  7592

Tree representation:
n= 13039

node), split, n, deviance, yval
       * denotes terminal node

1) root 13039 827.50290 -0.2431746
  2) a12>=520.5 9968  96.28462 -1.4005710 *
  3) a12< 520.5 3071 599.40030 -0.2793722 *

[[6]]
Number of valid observations:  13039
Number of missing observations:  7592

Tree representation:
n= 13039

node), split, n, deviance, yval
       * denotes terminal node

1) root 13039 906.2836 -0.2332683
  2) a4< 1421.5 12013 189.3829 -1.3728210 *
  3) a4>=1421.5 1026 160.3905  2.1494410 *
…
```

Po przetestowaniu modelu na danych testowych możemy sprawdzić wyniki predykcji do obserwacji nienależących do klasy 0.

```
boosted_model_prediction <- rxPredict(modelObject = boosted_model,
                                      data = test_df,
                                      extraVarsToWrite = "label2",
                                      type = "prob")

subset(boosted_model_prediction, label2!=0, 1:5)
```

	X0_prob	X1_prob	X2_prob	label2_Pred	label2
18	0.74164348	0.1239359	0.13442067	0	1
20	0.22895201	0.4287889	0.34225913	1	1
24	0.76586674	0.1311496	0.10298369	0	1
31	0.15069376	0.2822243	0.56708198	2	2
34	0.12925921	0.2420809	0.62865987	2	2
35	0.12925921	0.2420809	0.62865987	2	2
36	0.12925921	0.2420809	0.62865987	2	1
37	0.77591518	0.1197499	0.10433488	0	1
40	0.36226359	0.3996114	0.23812497	1	1
41	0.73522732	0.1659090	0.09886371	0	1
42	0.25784522	0.4829010	0.25925375	1	2
49	0.22370882	0.3774696	0.39882155	2	1
52	0.77139904	0.1248734	0.10372760	0	1
53	0.77157678	0.1246717	0.10375151	0	1
56	0.20762553	0.2884856	0.50388884	2	2
61	0.44707109	0.3405005	0.21242837	0	1
64	0.46607628	0.3345615	0.19936226	0	1
66	0.26014643	0.4389518	0.30090180	1	2
68	0.12925921	0.2420809	0.62865987	2	2
76	0.12925921	0.2420809	0.62865987	2	2
81	0.06165168	0.1154633	0.82288501	2	2
82	0.24558436	0.4599385	0.29447714	1	2
90	0.31852509	0.4270184	0.25445653	1	1
92	0.27706678	0.4628030	0.26013021	1	1
100	0.33393097	0.4173649	0.24870411	1	1

Na końcu policzymy metryki modelu i porównamy jego *F*-miary z *F*-miarami modelu losowego.

```
boosted_model_metrics <- evaluate_classifier(actual = boosted_model_prediction$label2,
                                                      boosted_model_prediction$label2_Pred)
boosted_model_metrics$Metrics[c(4,20),]
```

```
                        0         1         2
F1              0.9493671 0.4347826 0.7368421
RandomGuessF1   0.4615385 0.2068966 0.1538462
```

Jakość trzech najlepszych modeli: regresji logistycznej, lasów losowych i wzmocnionych drzew decyzyjnych jest bardzo podobna. Oznacza to, że jej poprawa wymaga albo dodatkowego przygotowania danych, albo użycia kombinacji różnych modeli uczenia maszynowego. Niestety, w czasie pisania książki kombinacje ani biblioteki Microsoft Machine Learning Services, ani popularne biblioteki Open Source, takie jak `caret`, nie umożliwiały łączenia klasyfikatorów wieloklasowych.

8.2.5. Porównanie modeli

Instrukcje tworzące oraz oceniające cztery omówione modele klasyfikacji wieloklasowej zostały zapisane w funkcji `model_factory`. Do pomiaru jakości modeli została użyta uproszczona wersja funkcji `evaluate_classifier` – obie wersje tej funkcji zostały omówione w następnym rozdziale. W tym miejscu ograniczymy się do porównania modeli na podstawie ich uśrednionych *F*-miar. Ja porównam modele nauczone na 10 i 20 zmiennych odczytanych z tabel zawierających opisane, wzbogacone i znormalizowane dane. Dzięki utworzeniu odpowiednich funkcji nauczenie modeli na wskazanych danych sprowadza się do wykonania poniższych instrukcji:

```
train_table_name <- "PredictiveMaintenance.train_Labels"
test_table_name <- "PredictiveMaintenance.test_Labels"
top_variables <- 10
model_factory (train_table_name, test_table_name, top_variables)
```

Metryki, w tym uśrednione *F*-miary modeli, zostały zapisane w tabeli `PredictiveMaintenance.Multiclass_metrics`. Dołączony do książki plik *Multiclass classification.R* zawiera definicje użytych do uczenia i oceny modeli funkcji oraz instrukcje pozwalające zapisać wyniki w tabeli serwera SQL. Pięć modeli o najwyższej i najniższej średniej makro *F*-miar zostało pokazanych poniżej.

```
WITH ranking AS
       (SELECT *, ROW_NUMBER() OVER (ORDER BY [Macro-averaged F-1] DESC) as Rn
       FROM [PredictiveMaintenance].[Multiclass_metrics])
SELECT [Name], [Variables], [Macro-averaged F-1], [Overall accuracy], Rn
FROM ranking
WHERE Rn <=5 OR Rn >=20
ORDER BY [Macro-averaged F-1] DESC;
```

Name	Variables	Macro-averaged	F-1	Overall accuracy	Rn
rxLogisticRegression on enchanced data	10	0.898343041504255	1.87		1
rxLogisticRegression on normalized data	20	0.894704342710497	1.86		2
rxBTrees on normalized data	10	0.890627949528596	1.87		3
rxBTrees on normalized data	20	0.890627949528596	1.87		4
rxDForest on enchanced data	20	0.890627949528596	1.87		5
rxNeuralNet on raw data	10	NULL	1.9746		20
rxNeuralNet on normalized data	10	NULL	1.75		21
rxNeuralNet on enchanced data	20	NULL	1.75		22
rxNeuralNet on enchanced data	10	NULL	1.75		23
rxNeuralNet on normalized data	20	NULL	1.75		24

Przyjrzyjmy się dwóm użytym w tym ćwiczeniu po raz pierwszy funkcjom uczenia maszynowego.

Funkcja `rxDForest` pozwala uczyć model lasów losowych. Zawiera on określoną liczbę drzew decyzyjnych zaimplementowanych w funkcji `rxDTree`. Jak wszystkie funkcje pakietu `RevoScaleR`, te też zostały zoptymalizowane pod

kątem blokowego przetwarzania dużych zbiorów danych. Tego typu algorytmy nie ładują całego zbioru treningowego do pamięci, tylko wielowątkowo przetwarzają go w blokach określonej wielkości. Dlatego ich wydajność w dużym stopniu zależy od tego, ile razy dane zostaną odczytane. W przypadku tego algorytmu dane zostaną raz odczytane i posortowane na potrzeby policzenia kwartyli, podział danych wymaga tylu odczytów, ile poziomów liczy budowane drzewo plus jeden, a przypisanie przykładów do liści drzewa wiąże się z dodatkowym odczytaniem danych treningowych.

Funkcja `rxBTrees` pozwala budować model wzmocnionych drzew decyzyjnych. Wchodzące w jego skład drzewa zostały zaimplementowane w funkcji `rxDTree`, do ich wzmacniania został użyty algorytm *GBM*.

Proszę zwrócić uwagę na różnice jakości poszczególnych, nauczonych na tych samych danych modeli.

Najlepszym modelem regresji okazał się model wzmocnionych drzew decyzyjnych, drugie miejsce przypadło kombinacji modeli wzmocnionych drzew decyzyjnych, lasu losowego i regresji liniowej. W obu przypadkach modele uczone na większej liczbie zmiennych okazały się dokładniejsze od wersji uczonych na 10 zmiennych. Co istotne, wszystkie modele uczone na niewzbogaconych danych, niezależnie od zastosowanego algorytmu, są wyraźnie gorsze od modeli uczonych na wzbogaconych lub znormalizowanych danych. Najsłabsze wyniki uzyskał model sztucznych sieci neuronowych.

Najlepiej z klasyfikacją binarną poradziły sobie sztuczne sieci neuronowe, na drugim miejscu uplasowała się regresja logistyczna. Również w tym przypadku wszystkie modele uczone na niewzbogaconych danych okazały się gorsze od uczonych na danych wzbogaconych lub znormalizowanych. Znamienny jest fakt, że te same algorytmy, które w tym przypadku okazały się najlepsze, nauczone na niewzbogaconych zajęły trzy ostatnie miejsca w rankingu.

Najlepszym modelem klasyfikacji wieloklasowej okazał się model regresji logistycznej, minimalnie gorsze były wzmocnione drzewa decyzyjne. W tym zadaniu liczba użytych do uczenia zmiennych wejściowych nie miała istotnego wpływu na uzyskane wyniki. Z klasyfikacją wieloklasową zupełnie nie poradziły sobie sztuczne sieci neuronowe. Jeżeli je pominiemy, raz jeszcze się okaże, że najgorzej wypadły modele uczone na niewzbogaconych danych.



Przeprowadzony eksperyment potwierdza tezę, według której lepsze dane wygrywają z mądrzejszymi algorytmami. Największą poprawę jakości modeli uzyskaliśmy, przygotowując dane, różnice między poszczególnymi algorytmami były znacznie mniejsze. Przekonaliśmy się też, że po osiągnięciu pewnego poziomu poprawa jakości modelu staje się coraz trudniejsza.

8.2.6. Wdrożenie modeli po stronie serwera SQL

Ostatnim krokiem jest implementacja wybranych modeli uczenia maszynowego po stronie serwera baz danych. Tym razem zasymulujemy sytuację, w której modele uczone na jednym serwerze, z zainstalowaną usługą SQL Server Machine Learning, będą używane na innym serwerze SQL, na którym usługa uczenia maszynowego nie została zainstalowana. Do uproszczenia przykładu, w książce rolę obu serwerów będzie pełniła jedna instancja serwera SQL.

Najpierw należy połączyć serwer produkcyjny z serwerem, na którym modele są uczone. W tym celu wykonamy poniższe procedury systemowe:

```
USE [master]
GO
EXEC master.dbo.sp_addlinkedserver @server = N'MLServer',
  @srvproduct=N'',
  @provider=N'SQLOLEDB',
  @datasrc='(local)';
EXEC master.dbo.sp_addlinkedsrvlogin @rmtsrvname = N'MLServer',
  @locallogin = NULL ,
  @useself = N'True';
```

Pierwsza procedura tworzy serwer połączony i nadaje mu nazwę `MLServer`. Ponieważ parametr `datasrc` wskazuje na domyślną instancję lokalnego serwera, pod tą nazwą będzie dostępny nasz lokalny serwer SQL. W praktyce jako wartość tego parametru należy podać nazwę zdalnego, produkcyjnego serwera baz danych.

Wykonanie drugiej procedury spowoduje wykonywanie instrukcji na połączonym serwerze z poświadczeniami użytkownika, który zalogował się na serwerze lokalnym. W praktyce konfiguracja zabezpieczeń połączonego serwera może być inna, a przekazywanie poświadczeń użytkowników między serwerami może wymagać skonfigurowania protokołu Kerberos.

Do uczenia modeli użyjemy procedury `PredictiveMaintenance.TrainMulticlassClassificationModel`. Zawiera ona deklarację zapytania zwracającego dane treningowe oraz jedno wywołanie procedury systemowej `sp_execute_external_script`. Do zewnętrznego skryptu są przekazywane: nazwa algorytmu, dane treningowe zwrócone przez zapytanie oraz liczba zmiennych najsilniej skorelowanych ze zmienną wyjściową, które mają zostać użyte do uczenia modelu. To, jaki model zostanie nauczony, zależy od przekazanej nazwy algorytmu. Zewnętrzny skrypt zwraca serwerowi SQL nauczony, zserializowany w trybie wspierającym natywne zapytania predykcyjne, model.

```
ALTER PROCEDURE [PredictiveMaintenance].[TrainMulticlassClassificationModel]
       @modelName varchar(100), @tableName varchar(100),
       @topVariables int, @trained_model varbinary(max) OUTPUT
AS
BEGIN
  DECLARE @inquery NVARCHAR(max) = N'SELECT * FROM ' + @tableName;
  EXEC sp_execute_external_script @language = N'R',
                                        @script = N'
train_table <- InputDataSet
train_table$label2 <- factor(train_table$label2, levels = c("0", "1", "2"))
train_vars <- rxGetVarNames(train_table)
train_vars <- train_vars[!train_vars  %in% c("RUL", "label1", "id", "cycle_orig")]

formula <- as.formula(paste("~", paste(train_vars, collapse = "+")))
correlation <- rxCor(formula = formula,
                         data = train_table,
                         transforms = list(label2 = as.numeric(label2)))
correlation <- correlation[, "label2"]
```

```
correlation <- abs(correlation)
correlation <- correlation[order(correlation, decreasing = TRUE)]
correlation <- correlation[-1]
correlation <- correlation[1:top_variables]
formula <- as.formula(paste(paste("label2~"),
                                  paste(names(correlation), collapse = "+")))

if (model_name == "rxLogisticRegression") {
   model <- MicrosoftML::rxLogisticRegression(formula = formula,
                                                          data = train_table,
                                                      type = "multiClass",
                                                      l1Weight = 0.95,
                                                      l2Weight = 0.95,
                                                      optTol = 1e-05,
                                                      normalize = "no")
} else if (model_name == "rxNeuralNet") {
   model <- MicrosoftML::rxNeuralNet(formula,
                                            data = train_table,
                                            type = "multiClass",
                                            normalize = "no",
                                            numIterations = 5,
                                            numHiddenNodes = 2,
                                            optimizer = adaDeltaSgd(),
                                            initWtsDiameter = 0.05,
                                            maxNorm=2)
} else if (model_name == "rxDForest") {
   model <- RevoScaleR::rxDForest(formula = formula,
                                             data = train_table,
                                             method = "class",
                                       nTree = 20,
                                       mTry = 10,
                                       replace = TRUE,
                                       strata = "label2",
                                       seed = 5,
                                       maxDepth = 10,
                                       cp = 0.002,
                                       minSplit = 15,
                                       maxNumBins = 200,
                                       findSplitsInParallel = TRUE)
} else {
   model <- RevoScaleR::rxBTrees(formula = formula,
                                          data = train_table,
                                          lossFunction = "multinomial",
                                     learningRate = 0.05,
                                     nTree = 25,
```

```
                                    mTry = 10,
                                    replace = FALSE,
                                    sampRate = 0.632,
                                    strata = "label2",
                                    seed = 215,
                                    cp = 0.01,
                                    minSplit = 15,
                                    minBucket = 8,
                                    maxNumBins = 300)
}
trained_model <- rxSerializeModel(model, realtimeScoringOnly = TRUE)'

,@input_data_1 = @inquery
,@params = N'@model_name varchar(100), @top_variables int,
 @trained_model varbinary(max) OUTPUT'
,@model_name = @modelName
,@top_variables = @topVariables
,@trained_model = @trained_model OUTPUT
END
```

Zbudowanie modelu lasów losowych, nauczenie go na 20 zmiennych odczytanych z tabeli ze wzbogaconymi danymi i zapisanie w tabeli serwera produkcyjnego sprowadza się do wykonania poniższych instrukcji:

```
DECLARE @model VARBINARY(MAX);
EXEC [PredictiveMaintenance].[TrainMulticlassClassificationModel]
        'rxDForest','[PredictiveMaintenance].[Train_Features]', 20,
         @model OUTPUT;
INSERT INTO MLServer.ML.[PredictiveMaintenance].[Models] (model_name, model)
VALUES('rxDForest multiclass classification', @model);
```

Proszę zwrócić uwagę, że została użyta w pełni kwalifikowana nazwa tabeli docelowej – nazwa tabeli została poprzedzona nazwą schematu, bazy danych i połączonego serwera, a jej poszczególne człony są oddzielone kropkami. Proszę pamiętać, że na potrzeby demonstracji rolę produkcyjnego serwera pełni lokalny serwer SQL, czyli znajduje się na nim baza danych `ML` z tabelą `PredictiveMaintenance.Models` oraz tabele z danymi testowymi. W przypadku użycia zdalnego serwera należy utworzyć na nim bazę danych z wymaganymi

tabelami.

Zbudujemy drugi model, tym razem wzmocnionych drzew decyzyjnych, i nauczmy go na 20 zmiennych wybranych z tabeli `PredictiveMaintenance.train_Features_Normalized`.

```
DECLARE @model VARBINARY(MAX);
EXEC [PredictiveMaintenance].[TrainMulticlassClassificationModel]
        'rxBTrees','[PredictiveMaintenance].[train_Features_Normalized]', 20,
         @model OUTPUT;
INSERT INTO MLServer.ML.[PredictiveMaintenance].[Models] (model_name, model)
VALUES('rxBTrees multiclass classification', @model);
```

Następnie przełączymy się na serwer produkcyjny i wykonamy na nim zapytanie predykcyjne zwracające wyniki prognoz dla silników o numerach większych niż 95.

```
DECLARE @model_raw VARBINARY(MAX) =
        (SELECT model FROM [PredictiveMaintenance].[Models]
        WHERE model_name = 'rxDForest multiclass classification')
SELECT
  a.id, a.cycle, a.label2, p.*
 FROM PREDICT(MODEL = @model_raw, DATA = [PredictiveMaintenance].[Test_Features] as a)
 WITH("0_prob" float, "1_prob" float, "2_prob" float, "label2_Pred" nvarchar(max)) as p
 WHERE a.id >95;
```

Id	cycle	label2	0_prob	1_prob	2_prob	label2_Pred
96	97	0	0.988750932310838	0.0111298246368249	0.000119243052337289	0
97	134	0	0.988750932310838	0.0111298246368249	0.000119243052337289	0
98	121	0	0.986919103028182	0.0129616539194802	0.000119243052337289	0
99	97	0	0.988750932310838	0.0111298246368249	0.000119243052337289	0
100	198	1	0.413884259069292	0.540162003826339	0.0459537371043691	1

Ponieważ użyliśmy funkcji `PREDICT`, ich wykonanie nie wymagało instalacji usługi SQL Server Machine Learning Services na produkcyjnym serwerze baz danych.

W tym rozdziale kontynuowaliśmy pracę nad modelem proaktywnej konserwacji

urządzeń. Ponieważ modele regresji nie były wystarczająco dokładne, przeformułowaliśmy postawiony problem. Zamiast przewidywać, ile cykli niezawodnej pracy pozostało urządzeniu, podzieliliśmy je na urządzenia, które w ciągu najbliższych 30 cykli mogą ulec awarii. Dodatkowo zbudowaliśmy modele, które wskazywały urządzenia mogące ulec awarii w ciągu najbliższych 15 i 30 cykli.

Jak automatycznie wybrać model będący kompromisem między błędem systematycznym a zmiennością predykcji?

Andrew Ng, Data Scientist w Baidu, założyciel Coursera, profesor na Uniwersytecie Stanforda

Rozdział 9
Ocena modeli

Zawodników oceniamy na wiele sposobów. Bierzemy pod uwagę takie ich cechy, jak szybkość, siła, odporność, kultura osobista, zwinność czy skuteczność. Chociaż te i wiele innych czynników mają wpływ na wyniki, sędziowie oceniają zawodników na podstawie jednego, dobrze zdefiniowanego kryterium. Z modelami uczenia maszynowego jest podobnie. Ich skuteczna ocena i porównywanie wymagają zastosowania najlepszego w danym przypadku kryterium.

Kryteriów oceny modeli predykcyjnych jest wiele. Najczęściej oceniamy modele ze względu na **trafność i wiarygodność predykcji** – temu tematowi został poświęcony prawie cały niniejszy rozdział. Coraz większą uwagę przywiązuje się również do **interpretowalności predykcji** – modele działające jak czarne skrzynki lub magiczne kule są uważane za gorsze od modeli, których działanie daje się wyjaśnić.

Oprócz łatwości interpretacji, trafności i wiarygodności predykcji modele uczenia maszynowego są oceniane ze względu na ich **wydajność oraz skalowalność**. Jednak te kryteria powinny być stosowane inaczej. Szybkość działania czy zdolność do przetwarzania większej ilości danych są ważne, ale rzadko są najważniejszym kryterium wyboru modelu. Model, którego czas

uczenia był o 10 min krótszy, ale predykcje są błędne, raczej nie zostanie uznany za lepszy od modelu uczonego dłużej i zwracającego poprawne predykcje. Dlatego **wydajność i skalowalność powinny być stosowane jako kryteria progowe**. Należy określić minimalną akceptowalną wydajność, na przykład czas uczenia modelu nie może być dłuższy niż 2 h, a czas wykonania 1000 zapytań predykcyjnych nie może przekroczyć 20 ms, i odrzucić wszystkie modele niespełniające tego kryterium. Pozostałe modele należy ocenić ze względu na trafność, wiarygodność i interpretowalność predykcji. To samo dotyczy skalowalności – modele, które nie skalują się dostatecznie dobrze, należy odrzucić, a pozostałe ocenić na podstawie jakości i interpretowalności ich predykcji.

9.1. Nie ma darmowego lunchu

W poprzednich rozdziałach zbudowaliśmy kilkadziesiąt modeli uczenia maszynowego i za ich pomocą rozwiązaliśmy kilka problemów. Chociaż były to różne problemy i zajmowaliśmy się nimi, używając różnorakich algorytmów uczenia maszynowego, za każdym razem postępowaliśmy według tych samych reguł. Ocenialiśmy i przygotowywaliśmy dane, staraliśmy się zapobiec nadmiernemu dopasowaniu modeli do danych treningowych, uczyliśmy wiele modeli, a do ich finalnej oceny używaliśmy danych testowych.

Z tego i następnego podrozdziału dowiemy się, dlaczego te same reguły stosują się do wszystkich algorytmów uczenia maszynowego. Znajomość teorii, wartościowa sama w sobie, okazuje się niezastąpiona, gdy nasze modele nie działają tak, jak się spodziewaliśmy. W takich przypadkach popołudnie spędzone w bibliotece może zaoszczędzić nam miesięcy zmarnowanych przy komputerze.

Punktem wyjścia do oceny modeli jest uznanie, że nie istnieje uniwersalny

algorytm uczenia maszynowego. Wynika to z twierdzenia o braku darmowego lunchu *NFL*. Twierdzenie o braku darmowego lunchu sformułował pod koniec XX w. profesor David Wolpert. Posługuje się on w tym twierdzeniu dwoma terminami niezbędnymi do jego zrozumienia:

- uogólniona trafność modelu $A_{CCG}(L)$ oznacza trafność predykcji modelu na całej populacji, czyli jego trafność na danych testowych;
- zbiór wszystkich możliwych hipotez F to zbiór możliwych do znalezienia przybliżeń funkcji $y = f(x)$.

W najprostszej wersji twierdzenie *NFL* brzmi następująco: **średnia uogólniona trafność dowolnego modelu uczenia maszynowego wynosi 50%, czyli jest równa losowemu zgadywaniu**

$$\sum_F A_{CCG}(L) = ½$$

Można to udowodnić, zauważając, że dla każdej hipotezy f, której uogólniona trafność wynosi $0{,}5 + \delta$, istnieje alternatywna hipoteza f', której uogólniona trafność wynosi $0{,}5 - \delta$. Średnia tych hipotez będzie więc równa 50%.

Z twierdzenia *NFL* można wyprowadzić wniosek, że dla dowolnych dwóch modeli uczenia maszynowego L_1 i L_2, jeśli uogólniona trafność modelu L_1 w rozwiązaniu danego problemu jest większa od uogólnionej trafności modelu

$$L_2 A_{CCG}(L_1) > A_{CCG}(L_2)$$

to istnieje problem, w rozwiązaniu którego uogólniona trafność modelu L_2 będzie większa od uogólnionej trafności modelu

$$L_1 A_{CCG}(L_2) > A_{CCG}(L_1)$$

Ma to kilka ważnych konsekwencji.

Po pierwsze, nie powinniśmy zakładać, że jakiś algorytm uczenia maszynowego

lepiej sobie poradzi z problemem od innych algorytmów. Jeżeli użycie algorytmu *A* w danym przypadku dało model o dużej uogólnionej trafności, to muszą istnieć przypadki, w których użycie tego samego algorytmu da model o małej uogólnionej trafności. Innymi słowy, **nie ma najlepszego algorytmu uczenia maszynowego**.

Po drugie, model uczenia maszynowego, który dobrze sobie poradził z jednym zadaniem, czyli jego uogólniona trafność była duża, może nie rozwiązać innego problemu, czyli uogólniona trafność modelu o tej samej architekturze będzie mała. Oznacza to, że **nie istnieje idealna kombinacja hiperparametrów modelu**.

W tym miejscu powinniśmy zapytać, dlaczego bardziej złożony model, na przykład sztuczna sieć neuronowa, może być gorszy od prostszego modelu, na przykład modelu regresji logistycznej. Albo, jak uogólniona trafność sztucznej sieci neuronowej o pięciu ukrytych warstwach zawierających 10 000 węzłów może być mniejsza od uogólnionej trafności sztucznej sieci neuronowej o jednej ukrytej warstwie, w której znajduje się 100 węzłów? Przecież bardziej złożony model, taki jak sztuczna sieć neuronowa, jest w stanie nauczyć się tego wszystkiego, czego nauczył się prostszy model, na przykład model regresji logistycznej i wiele nieliniowych wzorców, których prostszy model nie jest w stanie opanować.

Znają już Państwo odpowiedź na to pytanie – dlatego, że bardziej złożone modele są bardziej narażone na nadmierne dopasowanie do danych treningowych. Często okazuje się, że prostszy model, właśnie dlatego, że nie był w stanie wykryć wszystkich zależności między danymi treningowymi, choć nie jest idealny, to lepiej sprawdza się na danych testowych.

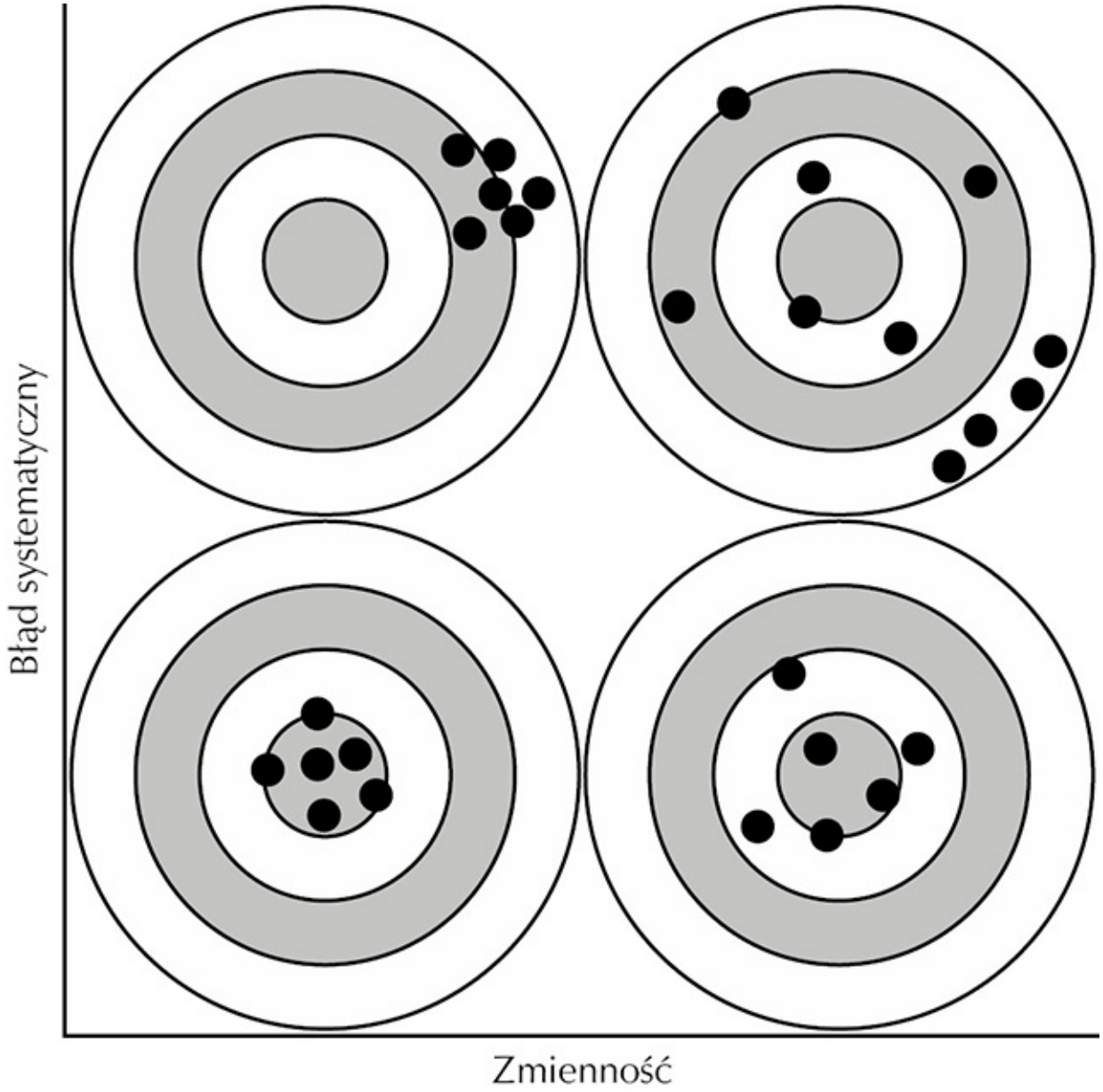
W praktyce oznacza to, że musimy budować i porównywać ze sobą wiele modeli. Zanim zdecydujemy, jakiego kryterium powinniśmy użyć w danym przypadku, musimy wyjaśnić, co składa się na błędy modeli predykcyjnych.

9.2.
Błędy modeli predykcyjnych

Predykcje zawsze są obarczone pewnym błędem. Niezależnie od zastosowanej funkcji kosztu można w tych błędach wyodrębnić trzy składowe: błąd systematyczny, błąd aproksymacji i szum, Loss = Bias + Variance + Noise. Ten podział pomaga wyjaśnić, czemu jakość zbudowanego modelu okazała się gorsza od spodziewanej i co zrobić, żeby ją poprawić.

Uświadomienie sobie, że na błąd predykcji zawsze składają się błąd systematyczny, błąd aproksymacji i szum pozwala odpowiedzieć na takie pytania, jak: Czemu prostszy model jest lepszy od złożonego?, Dlaczego lasy losowe i wzmocnione drzewa decyzyjne są lepsze od pojedynczych drzew decyzyjnych, oraz Dlaczego model nadmiernie dopasował się do danych treningowych i jak temu zapobiec?

Dwie pierwsze składowe, błąd systematyczny i błąd aproksymacji, można zredukować, odpowiednio dobierając hiperparametry modelu uczenia maszynowego. Szum jest nieredukowalny przez model, a jedyny sposób jego ograniczenia polega na odpowiednim przygotowaniu danych. Często błędy predykcji są spowodowane nieuwzględnieniem istotnych zmiennych lub nieprzygotowaniem uwzględnionych zmiennych. W tym miejscu ograniczymy się do przedstawienia błędu systematycznego i błędu aproksymacji (rys. 9.1); metody redukcji szumu zostały opisane w rozdziale 5.

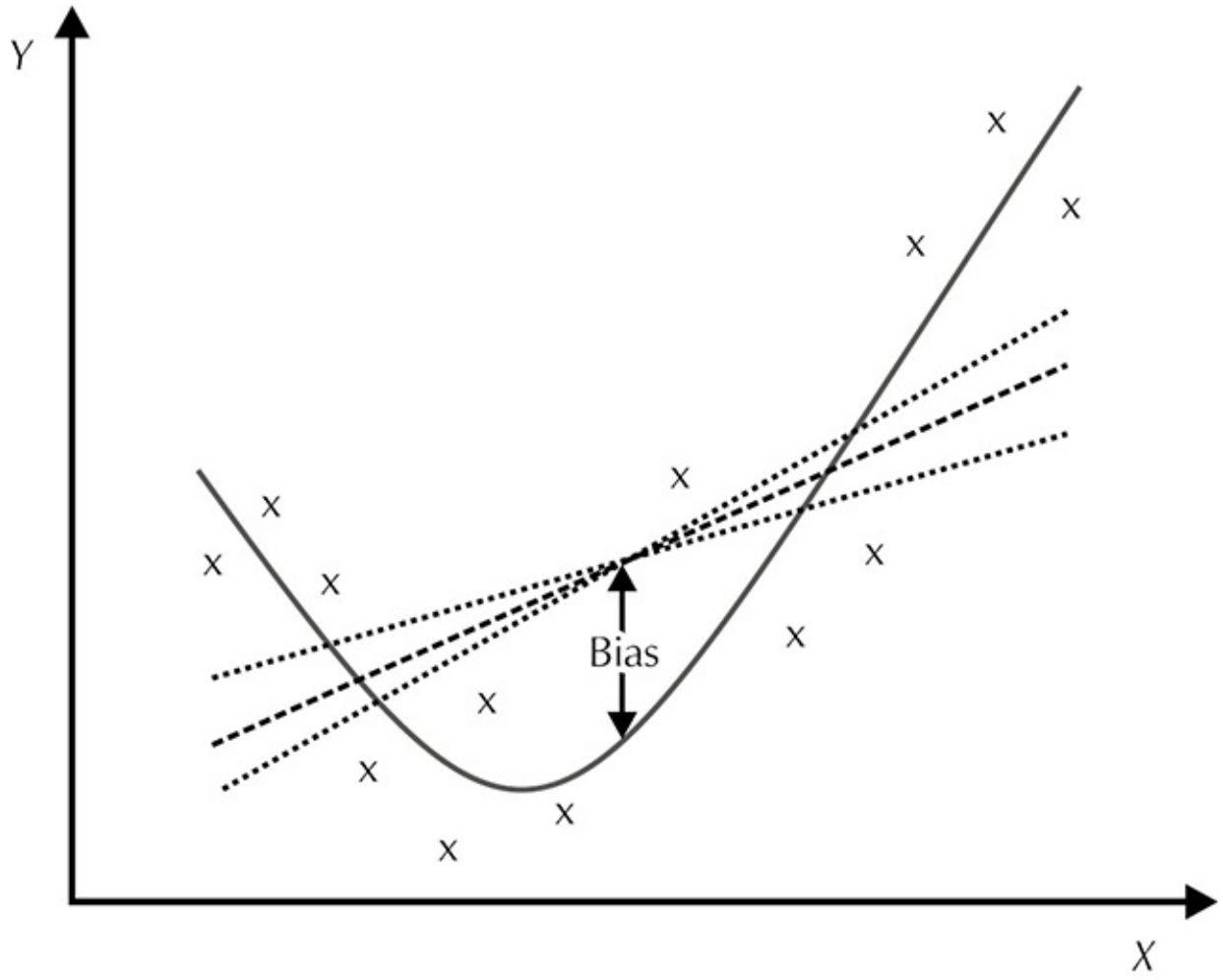


Rysunek 9.1. W lewym dolnym rogu znajduje się model idealny, czyli model o małym błędzie systematycznym i małej zmienności predykcji. Uogólniona trafność tego modelu jest duża. W lewym górnym rogu widzimy model o małym błędzie aproksymacji i dużym błędzie systematycznym. Predykcje tego modelu cechuje stałe „przesunięcie” względem prawdy. W prawym dolnym rogu znajduje się model o dużym błędzie aproksymacji i małym błędzie systematycznym. Jego predykcje po uśrednieniu byłyby poprawne, ale pojedyncze predykcje cechuje duża zmienność. W prawym górnym rogu widzimy najgorszy przypadek – model o dużym błędzie aproksymacji i dużym błędzie systematycznym

9.2.1.
Błąd systematyczny i błąd aproksymacji

Błąd systematyczny to różnica między prawdziwą wartością zmiennej wyjściowej a uśrednioną predykcją modelu. Duży błąd systematyczny jest objawem niedopasowania modelu do danych treningowych. Dwoma najczęstszymi przyczynami niedopasowania modelu są: użycie algorytmu uczenia maszynowego, którego wymagania nie były spełnione, oraz nadmierne uproszczenie modelu. Z pierwszą przyczyną mamy do czynienia wtedy, gdy używamy liniowego algorytmu do zamodelowania nieliniowego problemu. Z drugą – gdy nadmiernie upraszczamy model, na przykład ograniczając liczbę drzew decyzyjnych i ich głębokość.

Błąd systematyczny modelu obliczymy na podstawie błędów predykcji popełnionych na danych treningowych. Proszę przypomnieć sobie rysunek 3.1. Przedstawiał on niedopasowany, liniowy model. Teraz możemy wyjaśnić, że takie modele cechuje duży średni błąd systematyczny (rys. 9.2).

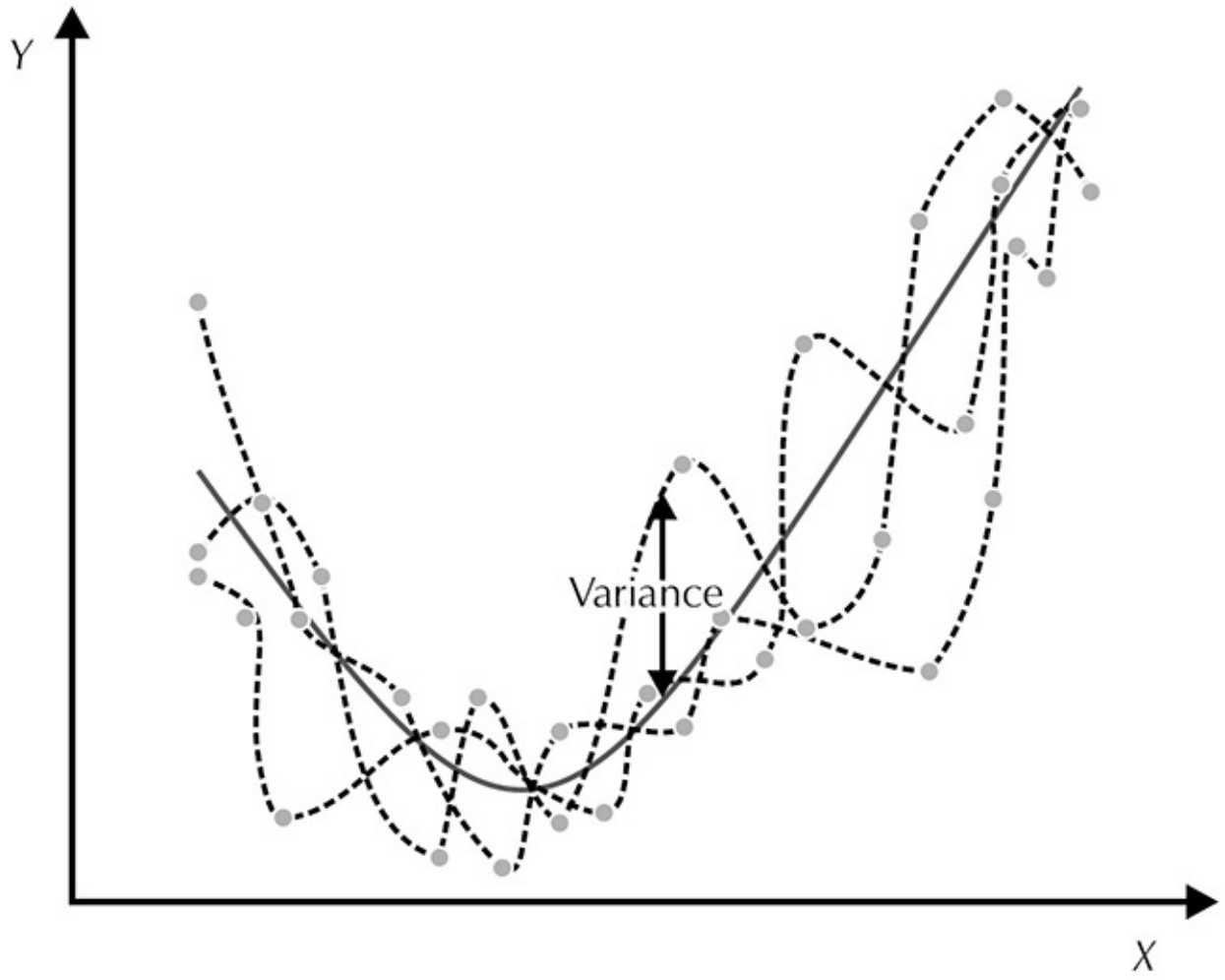


Rysunek 9.2. Dla dwóch wartości zmiennej *X* błąd systematyczny modelu liniowego jest zerowy – to punkty, w których linia prosta przecina wykres szukanej funkcji. Jednak dla pozostałych wartości zmiennej *X* błąd systematyczny modeli jest duży. Pisząc o błędzie systematycznym, będę miał na myśli średni błąd systematyczny modelu

Natomiast wynikający ze **zmienności predykcji błąd aproksymacji to różnica między średnią a pojedynczą predykcją.** Proszę zauważyć, że błąd aproksymacji nie zależy od prawdziwych wartości zmiennej wyjściowej. **Duża zmienność predykcji jest objawem nadmiernego dopasowania modelu do danych treningowych.** Do najczęstszych przyczyn nadmiernego dopasowania należą: nieprzygotowanie danych na potrzeby modelu, brak lub niewłaściwa regularyzacja oraz zbytnia złożoność modelu.

Błąd aproksymacji modelu obliczymy na podstawie różnic między błędami

predykcji popełnionymi na różnych zbiorach testowych. Na rysunku 3.2 był przedstawiony nadmiernie dopasowany model. Takie modele charakteryzują się dużą zmiennością predykcji. W tym przypadku uśrednienie predykcji trzech modeli przedstawionych na rysunku 9.3 zmniejszyłoby błąd, czyli należałoby połączyć wszystkie trzy modele.



Rysunek 9.3. Predykcje poszczególnych modeli są kompletnie różne. Pisząc o zmienności, również będą miał na myśli średnią zmienność predykcji modelu

Wspomniałem, że na błąd modelu, niezależnie od użytej w nim funkcji kosztu, składają się błąd systematyczny i błąd aproksymacji. Dwiema podstawowymi funkcjami kosztu są błąd średniokwadratowy i funkcja progowa 0–1. Pierwsza jest używana w modelach regresji, druga w klasyfikatorach. Zobaczmy, jak

rozkłada się mierzony tymi funkcjami błąd modelu na błąd systematyczny i aproksymacji.

Błąd średniokwadratowy jest równy kwadratowi różnicy między prawdziwą wartością zmiennej wyjściowej a wartością zwróconą przez model, co możemy zapisać jako

$$(t - y)^2 = (t - \bar{y})^2 + (\bar{y} - y)^2 + 2(t - \bar{y})(\bar{y} - y)$$

gdzie $\bar{y}$ jest typową wartością predykcji. **Oczekiwany błąd jest więc równy**

$$E(t - y)^2] = (t - \bar{y})^2 + E[(\bar{y} - y)^2]$$

gdzie $(t - \bar{y})^2$ reprezentuje błąd systematyczny, a $E[(\bar{y} - y)^2$ błąd aproksymacji.

W równaniu została użyta typowa wartość predykcji. Sposób jej obliczenia zależy od zastosowanej funkcji kosztu. Dla funkcji błędu średniokwadratowego $\bar{y}$ jest równe średniej predykcji. Natomiast dla funkcji progowej 0–1 jest to najczęściej występująca predykcja, czyli dominanta. To uogólnienie pozwala zdefiniować błąd systematyczny jako koszt typowej predykcji

$$Bias = L(t, \bar{y})$$

a błąd aproksymacji jako oczekiwany koszt predykcji względem typowej predykcji

$$Variance = E[L(\bar{y}, y)]$$

Zastosujmy tak uogólnione definicje do funkcji progowej. Jeśli predykcja jest prawidłowa, błąd systematyczny wynosi 0, a błąd systematyczny błędnej predykcji wynosi 1. Natomiast zmienność jest równa prawdopodobieństwu, z jakim predykcja jest różna od typowej predykcji $P(y \neq \bar{y})$.

Zwykle możemy zmniejszyć błąd systematyczny kosztem zwiększenia błędu

aproksymacji i vice versa – możemy ograniczyć zmienność predykcji modelu kosztem zwiększenia jego błędu systematycznego. Zastosujmy tę regułę do modelu klasyfikacji binarnej. Predykcje klasyfikatora mogą być prawdziwe, wtedy błąd wynosi 0, albo fałszywe, wtedy błąd wynosi 1. Czyli maksymalny błąd, jaki może popełnić klasyfikator binarny, jest równy 1. Powinniśmy rozważyć dwa przypadki: gdy błąd systematyczny jest równy 0 i gdy błąd systematyczny jest równy 1.

Jeżeli błąd systematyczny wynosi 0, to znaczy, że typowa predykcja jest prawdziwa $\overline{y} = t$, a więc **błąd**, czyli prawdopodobieństwo, z jakim predykcja jest nieprawdziwa $P(y \neq t)$, **jest równy sumie błędu systematycznego i błędu aproksymacji** $P(y \neq \overline{y})$. Ponieważ błąd systematyczny wynosi 0, błąd modelu odpowiada jego błędowi aproksymacji.

Jeżeli błąd systematyczny wynosi 1, to znaczy, że typowa predykcja jest nieprawdziwa $\overline{y} \neq t$, a więc model się nie myli, jeśli zwraca wynik różny od typowego $y \neq \overline{y}$. Prawdopodobieństwo, z jakim predykcja jest nieprawdziwa, jest równe $P(y \neq t)$, a błąd modelu jest równy $1 – P(y \neq \overline{y})$. Jedynka reprezentuje błąd systematyczny, prawdopodobieństwo, z jakim predykcja jest różna od typowej to błąd aproksymacji, a więc **błąd modelu jest równy różnicy między błędem systematycznym a błędem aproksymacji**. Wynika z tego, że zwiększając zmienność, możemy zmniejszyć błąd modelu. Ten wniosek dotyczy wyłącznie klasyfikatorów, a więc modeli, których maksymalny błąd predykcji jest ograniczony i wynosi 1. Zmienność modeli regresji zawsze jest problematyczna, a więc powinniśmy dążyć do zbudowania jak najbardziej stabilnych modeli regresji.

Ogólna formuła dekompozycji błędu na błąd systematyczny i błąd aproksymacji wygląda następująco:

$$Loss = Bias + c * Variance$$

gdzie *c* równa się 1 dla błędu systematycznego równego 0. Jeżeli błąd systematyczny jest różny od zera, a model jest klasyfikatorem binarnym, to *c* równa się –1, a jeżeli model jest klasyfikatorem wieloklasowym, to $c = -P(y = t \mid y \neq \bar{y})$. Powyższa analiza dotyczyła pojedynczej predykcji, w praktyce uśrednia się błędy systematyczne i zmienność predykcji modelu.

Ważnym wnioskiem jest to, że błąd systematyczny i błąd aproksymacji modeli nauczonych na tych samych danych, ale o różnej architekturze, będą inne. Dekompozycja błędu modelu pozwala poprawić jego jakość, bo błąd systematyczny ogranicza się zupełnie innymi metodami niż zmienność.

Jeżeli model cechuje duży błąd systematyczny, powinniśmy:

- zwiększyć zdolność uczenia modelu przez zwiększenie jego pojemności, na przykład zwiększyć liczbę drzew decyzyjnych lub ich wielkość, a w przypadku sztucznej sieci neuronowej – zwiększyć liczbę ukrytych węzłów;
- zwiększyć zdolność uczenia modelu przez zmianę algorytmu uczenia i metody optymalizacji procesu uczenia; możemy na przykład zastąpić regresję liniową sztuczną siecią neuronową;
- ograniczyć lub wyeliminować regularyzację, na przykład zwiększyć liczbę cykli uczenia, zmniejszyć wartości hiperparametrów *L1* i *L2* lub zmniejszyć minimalną liczbę przykładów wymaganą do podziału drzewa decyzyjnego;
- przygotować dane, w szczególności wzbogacić je o przydatne zmienne wyliczeniowe.

Jeżeli model cechuje się dużym błędem aproksymacji, należy:

- uprościć model przez zmniejszenie jego pojemności, na przykład zmniejszyć liczbę drzew decyzyjnych lub ich wielkość, a w przypadku sztucznej sieci neuronowej – zmniejszyć liczbę ukrytych węzłów;

- zwiększyć wielkość zbioru treningowego;
- dodać lub zwiększyć regularyzację, na przykład zmniejszyć liczbę cykli uczenia, zwiększyć wartości hiperparametrów *L1* i *L2* lub zwiększyć minimalną liczbę przykładów wymaganą do podziału drzewa decyzyjnego;
- przygotować dane, w szczególności zredukować liczbę zmiennych.

9.3. Kryteria oceny modeli regresji

Modele regresji szacują rzeczywistą wartość zmiennej wyjściowej, a więc ich ocena sprowadza się do porównania wyników predykcji z prawdziwymi wartościami. Najczęściej używanymi metrykami modeli regresji są: średni błąd bezwzględny, pierwiastek błędu średniokwadratowego, znormalizowane wersje tych metryk oraz współczynnik determinacji R^2.

Zbudowane w celu rozwiązania tego samego zadania projekty należy oceniać przy użyciu jednego, wybranego dla projektu kryterium. Tylko w ten sposób będziemy w stanie jednoznacznie porównywać i poprawiać modele. Często modele są oceniane według tego samego kryterium, które zostało zastosowane w funkcji kosztu. Takie rozwiązanie daje najlepsze wyniki, bo modele uczą się minimalizować ten sam błąd, który jest potem używany do ich oceny.

9.3.1. Średni błąd bezwzględny

Średni błąd bezwzględny *MAE* (*Mean Absolute Error*) mierzy wielkość różnic między prawdziwymi a przewidywanymi wartościami zmiennej wyjściowej bez uwzględniania ich znaku. *MAE* to średnia wartości bezwzględnych reszt modelu

$$MAE = \frac{\sum_{i=1}^{n} |p_i - a_i|}{n} = \frac{1}{n}, \sum_{i=1}^{n} |e_i|$$

gdzie n jest liczbą przykładów, i – kolejnym przykładem, a e_i – błędem estymacji dla przykładu i, czyli różnicą między faktyczną a oszacowaną wartością zmiennej wyjściowej. *MAE* przyjmuje wartości od zera do nieskończoności, mniejsze wartości oznaczają trafniejszy model.

Zaletą tego kryterium jest łatwość interpretacji wyników. *MAE* zwraca wynik w tej samej skali co zmienna wyjściowa i należy go interpretować jako średni błąd modelu.

MAE nigdy nie jest większe od drugiego kryterium oceny modeli regresji, pierwiastka błędu średniokwadratowego *RMSE*. Jeżeli model zawsze myli się o tę samą wartość, to *MAE* jest równe *RMSE*. Im większe różnice między poszczególnymi błędami modelu, tym *MAE* jest mniejsze od *RMSE*.

9.3.2. Pierwiastek błędu średniokwadratowego

Pierwiastek błędu średniokwadratowego *RMSE* (*Root Mean Squared Error*) mierzy kwadraty wielkości różnic między prawdziwymi a przewidywanymi wartościami zmiennej wyjściowej. Aby obliczyć *RMSE*, należy obliczyć pierwiastek ze średniej kwadratów błędów

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n} (p_i - a_i)^2}{n}} = \frac{1}{n}, \sum_{i=1}^{n} \sqrt{e_i^2}$$

gdzie n jest liczbą przykładów, i – kolejnym przykładem, a_i – rzeczywistą wartością zmiennej wyjściowej i, p_i – oszacowaną wartością zmiennej wyjściowej i, a e_i – błędem estymacji dla przykładu i. *RMSE* przyjmuje wartości od zera do nieskończoności, mniejsze wartości oznaczają trafniejszy model.

Zaletą tego kryterium jest silniejsze karanie dużych błędów. Ponieważ błędy

pojedynczych estymacji są podnoszone do kwadratu zanim zostaną uśrednione, jeden duży błąd ma większy wpływ na błąd modelu niż 10 małych. Policzenie pierwiastka z tak obliczonej średniej sprowadza wynik do skali zmiennej wyjściowej, co ułatwia jego interpretację.

RMSE nigdy nie jest większe od *MAE* pomnożonego przez pierwiastek kwadratowy liczby przykładów. Wynika z tego, że różnica między *RMSE* a *MAE* zwiększa się nie tylko wraz z większymi błędami, ale również wraz ze zwiększaniem się liczby przykładów.

9.3.3. Znormalizowany błąd bezwzględny

Znormalizowany błąd bezwzględny *RAE* (*Relative Absolute Error*) mierzy względną wielkość różnic między prawdziwymi a przewidywanymi wartościami zmiennej wyjściowej bez uwzględniania ich znaku. *RAE* jest wynikiem podzielenia średniego błędu bezwzględnego przez sumę wariancji wartości bezwzględnych zmiennej wyjściowej

$$RAE = \frac{\sum_{i=1}^{n} |p_i - a_i|}{\sum_{i=1}^{n} |\bar{a} - a_i|}$$

gdzie n jest liczbą przykładów, i – kolejnym przykładem, a_i – rzeczywistą wartością zmiennej wyjściowej i, p_i – jej oszacowaną wartością, a – średnią rzeczywistych wartości zmiennej wyjściowej. *RAE* przyjmuje wartości od zera do jeden, mniejsze wartości oznaczają trafniejszy model.

Zaletą tego kryterium jest reprezentowanie błędu modelu jako wartości względnej, często mnożonej przez 100, aby uzyskać wynik na skali 0–100%.

9.3.4. Znormalizowany błąd kwadratowy

Znormalizowany błąd kwadratowy *RSE* (*Relative Squared Error*) mierzy względne kwadraty wielkości różnic między prawdziwymi a przewidywanymi wartościami zmiennej wyjściowej. *RSE* jest wynikiem podzielenia pierwiastka błędu średniokwadratowego przez sumę kwadratów wariancji zmiennej wyjściowej

$$RSE = \frac{\sum_{i=1}^{n}(p_i - a_i)^2}{\sum_{i=1}^{n}(\bar{a} - a_i)^2}$$

gdzie n jest liczbą przykładów, i – kolejnym przykładem, a_i – rzeczywistą wartością zmiennej wyjściowej i, p_i – oszacowaną wartością zmiennej wyjściowej i, $\bar{a}$ – średnią rzeczywistych wartości zmiennej wyjściowej. *RSE* oblicza się, normalizując błąd średniokwadratowy przez jego podzielenie przez błąd kwadratowy prostego predykatora, czyli różnicę między faktycznymi wartościami a wartością średnią zmiennej wyjściowej. *RSE* przyjmuje wartości od zera do jeden, mniejsze wartości oznaczają trafniejszy model.

Zaletą tego kryterium jest reprezentowanie błędu modelu jako wartości względnej, często mnożonej przez 100, aby uzyskać wynik na skali 0–100% oraz silniejsze karanie dużych błędów.

9.3.5.
Współczynnik determinacji R^2

Współczynnik determinacji R^2 (*Coefficient of Determination*) reprezentuje zdolności predykcyjne modelu regresji. Oblicza się go na podstawie wariancji zmiennej wyjściowej w zależności od zmiennych wejściowych. Współczynnik determinacji R^2 jest miarą tego, jaki procent zmienności zmiennej wyjściowej został wyjaśniony przez zmienne wejściowe

$$R^2 = \frac{SSR}{SST} = 1 - \frac{SSE}{SST}$$

gdzie

$$SST = \sum(y - \bar{y})^2,\ SSR = \sum(y' - \bar{y}')^2,\ SSE = \sum(y - \bar{y}')^2$$

Jeżeli model jest idealny, suma kwadratów błędów *SS*E wynosi 0, a współczynnik R^2 przyjmuje maksymalną wartość 1. Jednocześnie, jeżeli model nie ma żadnej siły predykcyjnej, to suma kwadratów błędów *SSE* jest równa sumie kwadratów różnic między wartościami rzeczywistymi a średnią *SST*, a współczynnik R^2 przyjmuje minimalną wartość 0. Jeżeli współczynnik determinacji R^2 jest wyznaczany nie jako kwadrat *R*, ale według przedstawionej formuły, a model wyjaśnił mniej zmienności zmiennej wyjściowej niż model losowy, współczynnik R^2 przyjmie wartość ujemną.

9.3.6. Ocena modelu prognozującego pozostały czas bezawaryjnej pracy urządzeń

Chociaż języki R i Python zawierają specjalistyczne biblioteki ułatwiające ocenę modeli predykcyjnych, takie jak biblioteka `Metrics` czy moduł `metrics` biblioteki `scikit-learn`, w celach demonstracyjnych utworzymy własną funkcję oceniającą modele regresji.

```
evaluateRegressor <- function(observed, predicted) {
  mean_observed <- mean(observed)
  se <- (observed - predicted)^2
  ae <- abs(observed - predicted)
  sem <- (observed - mean_observed)^2
  aem <- abs(observed - mean_observed)
  mae <- mean(ae)
  rmse <- sqrt(mean(se))
  rae <- sum(ae) / sum(aem)
  rse <- sum(se) / sum(sem)
  rsq <- 1 - rse
  metrics <- c("Mean Absolute Error" = mae,
               "Root Mean Squared Error" = rmse,
               "Relative Absolute Error" = rae,
               "Relative Squared Error" = rse,
               "Coefficient of Determination" = rsq)
  return(metrics)
}
```

Następnie zbudujemy bardzo prosty model wzmocnionych drzew decyzyjnych i nauczymy go na danych odczytanych z tabeli `PredictiveMaintenance.Train_Features`.

```
formula <- "RUL ~ a4 + a11 + a21 + a15 + a20 + a17 + a12 + a7 + a2 + a3 + s11 + s4 +
s12 + s7 + s15"
model <- MicrosoftML::rxFastTrees(formula,
                                  data = train_df,
                                  type = "regression",
                                  numTrees = 2,
                                  numLeaves = 5,
                                  minSplit = 10,
                                  numBins = 100,
                                  exampleFraction = 0.9,
                                  featureFraction = 1,
                                  splitFraction = 1,
                                  learningRate = 0.05)
```

Błąd systematyczny modelu można ocenić na podstawie jego błędów empirycznych. Czyli, żeby go wyznaczyć, należy obliczyć miary jakości modelu dla danych treningowych.

```
model_predictions <- rxPredict(modelObject = model,
                               data = train_df,
                               extraVarsToWrite = "RUL")
model_empirical_metrics <- evaluateRegressor (observed = model_predictions$RUL,
                                              predicted =
model_predictions$Score)
as.data.frame(model_empirical_metrics)
```

	model_empirical_metrics
Mean Absolute Error	97.323170
Root Mean Squared Error	117.219287
Relative Absolute Error	1.711670
Relative Squared Error	2.896146
Coefficient of Determination	-1.896146

Wszystkie kryteria wyraźnie wskazują na bardzo duży błąd systematyczny modelu. Spróbujmy zwiększyć zdolność uczenia modelu przez zwiększenie jego pojemności.

```
model <- MicrosoftML::rxFastTrees(formula,
                                  data = train_df,
                                  type = "regression",
                                  numTrees = 10,
                                  numLeaves = 10,
                                  minSplit = 5,
                                  numBins = 100,
                                  exampleFraction = 0.9,
                                  featureFraction = 1,
                                  splitFraction = 1,
                                  learningRate = 0.05)
```

Zwiększenie liczby i wielkości drzew decyzyjnych powinno zmniejszyć błąd systematyczny modelu.

	model_empirical_metrics
Mean Absolute Error	64.9041758
Root Mean Squared Error	84.7169762
Relative Absolute Error	1.1415014
Relative Squared Error	1.5127361
Coefficient of Determination	-0.5127361

Zgodnie z oczekiwaniami wszystkie metryki się poprawiły, ale błąd systematyczny modelu nadal jest duży. Może to świadczyć o tym, że zdolności uczenia modelu nadal są zbyt małe. Dalsze zwiększenie liczby drzew decyzyjnych do 30 i liczby węzłów poszczególnych drzew do 20 dało poniższe wyniki.

	model_empirical_metrics
Mean Absolute Error	35.1404105
Root Mean Squared Error	50.6717610
Relative Absolute Error	0.6180316
Relative Squared Error	0.5411956
Coefficient of Determination	0.4588044

Ten model zdołał już czegokolwiek się nauczyć na danych treningowych. Błąd systematyczny jest wyraźnie mniejszy niż poprzednich dwóch modeli. Średni błąd bezwzględny wynosi 35, a pierwiastek błędu średniokwadratowego wynosi 50. Czyli ten model ma tendencje do popełniania dużych błędów przy szacowaniu wartości zmiennej `RUL`. Zdolności predykcyjne modelu są niewielkie – zdołał on wyjaśnić 46% zmienności zmiennej wyjściowej.

Żeby ocenić drugą składową błędu modelu, czyli zmienność, musimy porównać wyniki predykcji na różnych zbiorach danych testowych. Wykorzystamy do tego poznaną w rozdziale 5 metodę *k*-krotnej walidacji krzyżowej.

Podzielimy zbiór treningowy na dziesięć podzbiorów i zbudujemy 10 modeli, ucząc każdy z nich na 9 podzbiorach. Nieużyty do treningu podzbiór wykorzystamy do oceny modelu. Na końcu uśrednimy wyniki poszczególnych modeli i policzymy ich odchylenia standardowe. Do podzielenia danych na podzbiory użyjemy funkcji `cvFolds` z biblioteki `cvTools`.

```
library(cvTools)
MAE <- as.numeric()
RMSE <- as.numeric()
r2 <- as.numeric()
kf<-cvFolds(nrow(train_df), K = 10, type = "random")
for (i in 1:10) {
  model <- rxFastTrees(formula = formula,
                       data = train_df[kf$which!=i,],
                       type = "regression",
                       numTrees = 30,
                       numLeaves = 20,
                       minSplit = 5,
                       numBins = 100,
                       exampleFraction = 0.9,
                       featureFraction = 1,
                       splitFraction = 1,
                       learningRate = 0.05)
  scoreDS <- rxPredict(model, data = train_df[kf$which==i,], extraVarsToWrite = c("RUL"))
  model_generalization_metrics <- evaluateRegressor (observed = scoreDS$RUL,
                                                     predicted = scoreDS$Score)
  MAE <- rbind(MAE, model_generalization_metrics[1])
  RMSE <- rbind(RMSE, model_generalization_metrics[2])
  r2 <- rbind(r2, model_generalization_metrics[5])
}
sprintf ('Model mean MAE: %.2f, stdev MAE %.3f' , mean(MAE), sd(MAE))
sprintf ('Model mean RMSE: %.2f, stdev RMSE: %.3f' ,mean(RMSE), sd(RMSE))
sprintf ('Model mean R2: %.2f, stdev R2: %.3f' ,mean(r2), sd(r2))
```

```
"Model mean MAE: 35.46, stdev MAE 0.180"
"Model mean RMSE: 51.12, stdev RMSE: 0.259"
"Model mean R2: 0.45, stdev R2: 0.005"
```

Zmienność predykcji mierzona ich średnim błędem bezwzględnym, pierwiastkiem błędu średniokwadratowego i współczynnikiem determinacji R^2 jest bardzo mała. Przyjmijmy, że otrzymane wyniki nas satysfakcjonują. Przeprowadzamy więc końcową ocenę modelu na danych testowych.

```
model_prediction <- rxPredict(modelObject = model,
                                    data = test_df,
                                    extraVarsToWrite = "RUL")
model_test_metrics <- evaluateRegressor (observed = model_prediction$RUL,
                                                                predicted =
model_prediction$Score)
as.data.frame(model_test_metrics)
```

	model_test_metrics
Mean Absolute Error	17.0281555
Root Mean Squared Error	22.5126544
Relative Absolute Error	0.4631344
Relative Squared Error	0.2934904
Coefficient of Determination	0.7065096

Okazało się, że model znacznie lepiej sobie radzi z danymi testowymi niż treningowymi. Jeżeli dane testowe różnią się statystycznie od treningowych i walidacyjnych, błąd testowy może być zupełnie różny od błędu walidacyjnego. W tym przypadku model znalazł w danych treningowych zależności między zmiennymi, które jeszcze silnej występują w danych testowych.

Dla kompletności przykładu zbudujemy jeszcze bardzo złożony model wzmocnionych drzew decyzyjnych. Skoro zwiększanie liczby i wielkości drzew poprawiało jakość modelu, to utwórzmy model złożony z 300 drzew, z których każde będzie zawierało do 200 węzłów.

Zgodnie z przewidywaniami błąd systematyczny modelu jest bardzo mały – *RMSE* tego modelu wynosi 15, podczas gdy *RMSE* modelu złożonego z 30 drzew wyniósł 50.

	model_empirical_metrics
Mean Absolute Error	12.0797424
Root Mean Squared Error	15.6574373
Relative Absolute Error	0.2124523
Relative Squared Error	0.0516730
Coefficient of Determination	0.9483270

Natomiast błąd aproksymacji nowego modelu jest bardzo duży.

```
for (i in 1:10) {
  model <- MicrosoftML::rxFastTrees(formula,
                                    data = train_df,
                                    type = "regression",
                                    numTrees = 300,
                                    numLeaves = 200,
                                    minSplit = 1,
                                    numBins = 100,
                                    exampleFraction = 1,
                                    featureFraction = 1,
                                    splitFraction = 1,
                                    learningRate = 0.05)
  scoreDS <- rxPredict(model, data = train_df[kf$which==i,], extraVarsToWrite = c("RUL"))
  model_generalization_metrics <- evaluateRegressor (observed = scoreDS$RUL,
                                                     predicted = scoreDS$Score)
  MAE <- rbind(MAE, model_generalization_metrics[1])
  RMSE <- rbind(RMSE, model_generalization_metrics[2])
  r2 <- rbind(r2, model_generalization_metrics[5])
}
sprintf ('Model mean MAE: %.2f, stdev MAE %.3f' , mean(MAE), sd(MAE))
sprintf ('Model mean RMSE: %.2f, stdev RMSE: %.3f' ,mean(RMSE), sd(RMSE))
sprintf ('Model mean R2: %.2f, stdev R2: %.3f' ,mean(r2), sd(r2))
```

```
"Model mean MAE: 23.77, stdev MAE 11.996"
"Model mean RMSE: 33.39, stdev RMSE: 18.195"
"Model mean R2: 0.70, stdev R2: 0.256"
```

Mały błąd systematyczny i duża zmienność wskazują na nadmierne dopasowanie modelu. **Z nadmiernym dopasowaniem mamy do czynienia, jeżeli istnieje inny nauczony na tych samych przykładach model o większym błędzie treningowym, ale mniejszym błędzie testowym.** Sprawdźmy więc błąd testowy nowego modelu.

```
                                model_test_metrics
Mean Absolute Error                     23.5670005
Root Mean Squared Error                 32.0660842
Relative Absolute Error                  0.6409789
Relative Squared Error                   0.5954322
Coefficient of Determination             0.4045678
```

Według wszystkich kryteriów jest on większy od błędu testowego modelu zawierającego 30 drzew decyzyjnych. Nasze podejrzenie zostało potwierdzone – model cechujący się dużą zmiennością i małym błędem systematycznym nadmiernie dopasował się do danych treningowych.

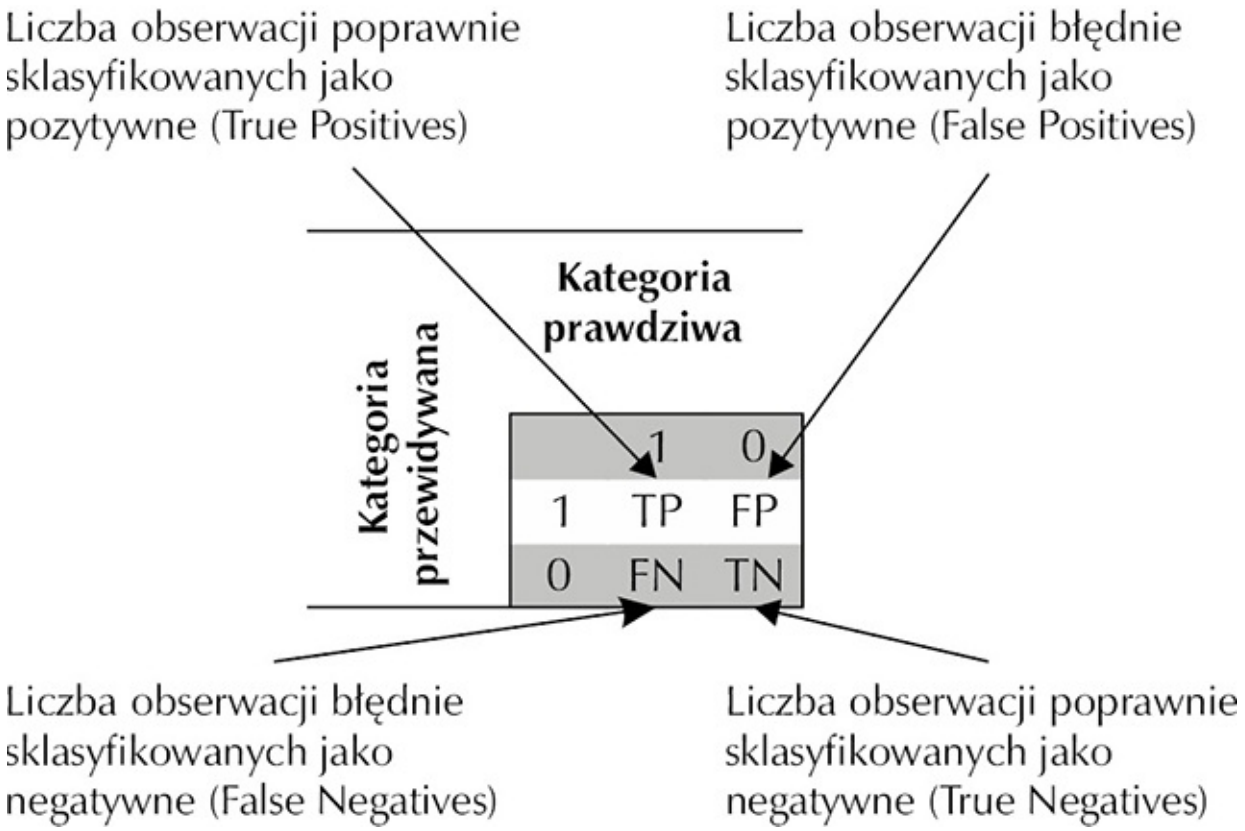
9.4. Kryteria oceny modeli klasyfikacji binarnej

Klasyfikatory binarne przypisują przykłady do jednej z dwóch klas: pozytywnej lub negatywnej. Oznacza to, że przykład może zostać prawidłowo sklasyfikowany, przykład klasy pozytywnej może zostać błędnie przypisany do klasy negatywnej albo przykład negatywny może zostać błędnie przypisany do klasy pozytywnej. Zaliczając otrzymane w ten sposób wyniki, otrzymamy macierz błędów modelu. Macierz błędów jest podstawą pozostałych kryteriów oceny klasyfikatorów.

Klasyfikatory, oprócz przypisywania przypadków do klas, zwracają również rozkład prawdopodobieństwa, z jakim przykład należy do poszczególnych klas. To prawdopodobieństwo pozwala modyfikować wartość graniczną, po przekroczeniu której przykład zostanie sklasyfikowany jako pozytywny. Proszę zauważyć, że zmieniając wartość graniczną, uzyskamy różne macierze pomyłek modelu. Tak otrzymane wyniki są używane do wyznaczenia krzywej *ROC* i wyliczanych na jej podstawie kryteriów oceny klasyfikatorów binarnych, takich jak obszar pod krzywą.

9.4.1. Macierz błędów

Macierz błędów klasyfikatora binarnego składa się z dwóch kolumn i dwóch wierszy, a jej komórki oznacza się symbolami *TP*, *TN*, *FP* i *FN* (rys. 9.4).



Rysunek 9.4. Macierz błędów jest macierzą kwadratową rozmiaru m × n , gdzie *m* jest liczbą stanów zmiennej wyjściowej

Oznaczenia *TP*, *TN*, *FP* i *FN* symbolizują cztery możliwe przypadki klasyfikacji binarnej. Symbole te oznaczają:

- *TP* (*True Positives*) – liczba pozytywnych przypadków poprawnie sklasyfikowanych jako należących do klasy pozytywnej;
- *FN* (*False Negatives*) – liczba pozytywnych przypadków niepoprawnie sklasyfikowanych jako należące do klasy negatywnej; **błędy typu *FN* nazywa się błędami drugiego rodzaju**; jeżeli konsekwencje błędnego przypisania obserwacji do klasy negatywnej mogą być katastrofalne, na przykład prowadzić do niewykrycia choroby nowotworowej, czego skutkiem może być przedwczesna śmierć pacjenta, priorytetem powinno być ograniczenie błędów drugiego rodzaju;
- *FP* (*False Positives*) – liczba negatywnych przypadków niepoprawnie

sklasyfikowanych jako należących do klasy pozytywnej; **są to błędy pierwszego rodzaju**; jeżeli konsekwencje błędnego przypisania obserwacji do klasy pozytywnej mogą być katastrofalne, na przykład awaryjne zatrzymanie pociągu z powodu błędnego uznania leżącego w pobliżu torów przedmiotu za człowieka może zagrozić zdrowiu i życiu pasażerów – priorytetem powinno być ograniczenie błędów pierwszego rodzaju;

- *TN* (*True Negatives*) – liczba negatywnych przypadków poprawnie sklasyfikowanych jako należących do klasy negatywnej.

Macierz błędów jest najbardziej intuicyjnym kryterium oceny jakości klasyfikatorów. Ma jednak tę wadę, że zawiera wiele wyników – w przypadku klasyfikatorów binarnych są to cztery liczby. W dodatku prawidłowa interpretacja każdej z nich wymaga uwzględnienia pozostałych. Na przykład, czy model, który nie popełnił ani jednego błędu typu pierwszego, jest lepszy od modelu, który błędnie sklasyfikował 10 przykładów jako pozytywne, ale za to nie popełnił ani jednego błędu typu drugiego?

Jeżeli błędne zaklasyfikowanie przypadków ma takie same konsekwencje dla każdej pary klas, czyli jeżeli koszty błędów typu pierwszego i typu drugiego są takie same, z macierzy błędów można bezpośrednio wyprowadzić łatwiejsze w użyciu metryki klasyfikatorów.

9.4.2. Trafność

Trafność (ang. *accuracy*) mierzy proporcję prawidłowych klasyfikacji do wszystkich klasyfikacji

$$trafność = \frac{TP + TN}{TP + FN + FP + TN}$$

Trafność czasami wyraża się liczbą błędów klasyfikacji (ang. *error ratio*)

$$\textit{błąd klasyfikacji} = \frac{FP + FN}{TP + FN + FP + TN}$$

Żadna z tych miar nie nadaje się to oceny modeli, jeżeli dysponujemy niezbilansowanymi danymi. W takich przypadkach modele o bardzo dużej trafności mogą być kompletnie nieprzydatne. Weźmy model diagnozujący rzadką chorobę. Jeżeli tylko 1% z 1000 zbadanych pacjentów cierpi na tę chorobę, trafność modelu, który wszystkich pacjentów sklasyfikuje jako zdrowych, wyniesie 99%, czyli *trafność* = $\frac{990}{1000}$. Model za każdym razem zwracający ten sam wynik, mimo doskonałej trafności, jest jednak co najmniej bezużyteczny, jeżeli nie szkodliwy.

Co więcej, jeżeli w nowej grupie 100 pacjentów nie będzie ani jednego cierpiącego na tę chorobę, to trafność modelu nauczonego na nowym, liczącym 1100 przykładów zbiorze danych, wzrośnie o 0,1%, czyli *trafność* = $\frac{1000}{1000}$, chociaż model ten nadal będzie wszystkich pacjentów klasyfikował jako zdrowych! **Fenomen ten jest znany jako paradoks dokładności.**

Ponieważ trafność, tak samo jak pozostałe kryteria wyprowadzone z macierzy pomyłek, jest miarą kontekstową, wymaga przyjęcia jakiejś wartości bazowej. Taką wartością może być:

- trafność, z jaką daje się klasyfikować przypadki na podstawie częstości występowania poszczególnych klas (trafność losowego zgadywania); użyte w dalszej części rozdziału funkcje oceniające jakość klasyfikatora zwracają, oprócz wyliczonych metryk, metryki losowego klasyfikatora;
- trafność, z jaką daje się klasyfikować przypadki na podstawie obserwacji (trafność a posteriori).

9.4.3. Precyzja

Precyzję modelu klasyfikacji binarnej możemy mierzyć na dwa sposoby.

Precyzja przewidywania pozytywnego (ang. *Positive Predictive Value*) mierzy proporcję prawidłowych pozytywnych klasyfikacji względem wszystkich pozytywnych klasyfikacji

$$PPV = \frac{TP}{TP + FP}$$

czyli odpowiada na pytanie: *Ile z pozytywnie sklasyfikowanych przypadków zostało dobrze sklasyfikowanych?* Precyzję przewidywania pozytywnego nazywa się precyzją modelu (ang. *precision*). **Precyzję powinniśmy wybrać jako miarę oceny modelu, jeżeli zależy nam przede wszystkim na ograniczeniu błędów typu pierwszego.**

Precyzja przewidywania negatywnego (ang. *Negative Predictive Value*) wskazuje, z jaką pewnością możemy ufać przewidywaniom negatywnym

$$NPV = \frac{TN}{TN + FN}$$

tzn. w jakim procencie przewidywania negatywne są zgodne ze stanem faktycznie negatywnym.

9.4.4. Czułość

Czułość (ang. *recall*), mierzy proporcję poprawnych klasyfikacji przykładów pozytywnych względem wszystkich przykładów należących do klasy pozytywnej

$$czułość = \frac{TP}{TP + FN}$$

czyli odpowiada na pytanie: *Ile z pozytywnych przypadków zostało dobrze sklasyfikowanych?* **Czułość powinniśmy wybrać jako miarę oceny modelu,**

jeżeli zależy nam przede wszystkim na ograniczeniu błędów typu drugiego.

Odpowiednikiem czułości dla klasy negatywnej jest specyficzność (ang. *specifity*), nazywana też współczynnikiem *TN* lub znamiennością. Mierzy ona proporcję prawidłowych negatywnych klasyfikacji względem wszystkich negatywnych przypadków, czyli odpowiada na pytanie: *Ile z negatywnych przypadków zostało dobrze sklasyfikowanych?*

9.4.5. *F*-miara

F-miara (ang. *F-Score*) jest średnią harmoniczną precyzji i czułości

$$F\text{-}miara = 2 * \frac{precyzja * czułość}{precyzja + czułość}$$

Do obliczenia *F*-miary używa się średniej harmonicznej, a nie arytmetycznej lub geometrycznej (średnią geometryczną zwraca rzadko używana *G*-miara), bo precyzja i czułość mierzą proporcje. O ile średnia arytmetyczna jest używana w przypadku wartości, które można sensownie dodawać, a średnia geometryczna w przypadku wartości, które dają się sensownie mnożyć, to do uśredniania proporcji używa się średniej harmonicznej. Zakres wartości *F*-miary wynosi od 0 do 1, gdzie 1 oznacza model bezbłędny.

Ponieważ precyzja i czułość mają dokładnie taki sam wpływ na wynik *F*-miary, należy wybrać ją jako miarę oceny modelu, jeżeli koszt błędów obu rodzajów jest zbliżony. Jeżeli ich konsekwencje są różne, nie należy używać *F*-miary do oceny jakości modelu.

9.4.6. Współczynnik *Kappa* Cohena

Współczynnik *Kappa* Cohena porównuje zmierzoną trafność modelu ze

spodziewaną, tj. trafnością losowego zgadywania

$$Kappa = \frac{accuracy - expAccuracy}{1 - expAccuracy}$$

Jest to pierwsze z przedstawionych kryteriów oceny klasyfikatorów, które bierze pod uwagę rzeczywisty rozkład zmiennej wyjściowej. Spodziewaną trafność oblicza się, mnożąc liczbę przykładów danej klasy przez liczbę przykładów przypisanych do tej klasy przez model i dzieląc tak uzyskany wynik przez liczbę wszystkich przykładów tej klasy. Wyniki uzyskane dla obu klas dodaje się do siebie i dzieli przez liczbę wszystkich przykładów.

Większe wartości współczynnika *Kappa* oznaczają lepszy, bardziej odległy od losowego zgadywania model. Kryterium to może być używane do porównywania różnych, uczonych na różnych przykładach klasyfikatorów binarnych.

9.4.7. Krzywa ROC i obszar pod krzywą

Jakość klasyfikatorów binarnych można przedstawić w postaci punktów w dwuwymiarowej przestrzeni *ROC* (*Receiver Operating Characteristics*), w której na osi odciętych została umieszczona 1 – specyficzność, a na osi rzędnych czułość modelu. Położenie w tej przestrzeni punktu reprezentującego klasyfikator przedstawia kompromis między zyskiem (współczynnikiem trafień) a kosztem (współczynnikiem fałszywych alarmów).

W przestrzeni *ROC*:

- punkt o współrzędnej (0, 0) reprezentuje model, który zawsze zwraca predykcje negatywne, dzięki czemu unika błędów pierwszego rodzaju, ale też nie uzyskuje prawidłowych wskazań (*TP=0*);
- punkt (1, 1) reprezentuje przeciwny model, który zawsze zwraca pozytywne

predykcje;

- punkt (0, 1) reprezentuje idealny model;
- punkty na przekątnej, czyli prostej wyznaczonej przez równanie 1 – specyficzność = czułość, reprezentują losowy model.

Przyjmuje się, że poprawne modele są reprezentowane przez punkty znajdujące się w górnych ćwiartkach wykresu. Im dalej na górze z lewej strony znajduje się model (gdzie wrażliwość i czułość są większe), tym jest on lepszy. Klasyfikatory znajdujące się po lewej stronie wykresu i przy osi odciętych są uznawane za konserwatywne, czyli takie, które zwracają pozytywną predykcję tylko wtedy, gdy są do tego silne przesłanki. Natomiast klasyfikatory znajdujące się w górnym prawym rogu określa się jako liberalne, czyli dające wynik pozytywny nawet przy słabych przesłankach.

Skoro model reprezentuje punkt, którego współrzędne wyznaczają czułość i specyficzność modelu, do wykreślenia krzywej *ROC* są potrzebne prawdopodobieństwa predykcji. Zmieniając wartość progową predykcji modelu, otrzymamy sekwencję punktów reprezentujących klasyfikator w przestrzeni *ROC*. Łącząc tak otrzymane punkty, otrzymamy krzywą *ROC*. Krzywa ta pokazuje zależność między liczbą poprawnie sklasyfikowanych przykładów pozytywnych (*TP*) a liczbą niepoprawnie sklasyfikowanych przykładów z klasy negatywnej (*FP*). **Zaletą krzywej ROC i obszaru pod tą krzywą AUC jest ich niezależność od przyjętej w modelu wartości progowej.**

Aby łatwiej porównywać klasyfikatory, oblicza się pole powierzchni pod krzywą *ROC*, czyli obszar *AUC* (*Area Under ROC Curve*). Idealny klasyfikator ma wartość *AUC* równą 1, natomiast dla klasyfikatora losowego wynosi ona 0,5. Ważną statystyczną własnością *AUC* jest to, że można ją interpretować jako prawdopodobieństwo, że model wyżej oceni losowo wybrany pozytywny element niż losowo wybrany negatywny element, co odpowiada testowi

Wilcoxona.

Oba te kryteria, ROC i AUC, częściej są używane w pracach naukowych niż w praktyce. O ile na podstawie krzywej *ROC* można, przez zmianę wartości brzegowej, poprawić jakość modelu, o tyle *AUC* ma, jak się wkrótce przekonamy, małe znaczenie praktyczne.

9.4.8. Ocena modelu klasyfikującego urządzenia jako wymagające lub niewymagające przeglądu

Do oceny klasyfikatorów binarnych użyjemy trzech funkcji. Funkcja `evaluate_classifier` zwraca macierz błędów oraz wszystkie wyprowadzone na jej podstawie kryteria. Dla każdego z kryteriów jest również obliczany jego odpowiednik dla modelu losowego – na podstawie częstości występowania poszczególnych klas. Funkcja zwraca listę, której pierwszym elementem jest macierz błędów, a drugim – obliczone metryki modelu.

```
evaluate_classifier <- function (actual, predicted, data, ...) {
  varInfo <- rxGetVarInfo(data)
  if (varInfo[[actual]]$varType != "factor") {
    actual <- paste0("F(",actual,")")
  }
  if (varInfo[[predicted]]$varType != "factor") {
    predicted <- paste0("F(",predicted,")")
  }
  myForm <- as.formula(paste("~",paste(actual,predicted, sep = ":")))
  cm <- rxCrossTabs(myForm,data = data, returnXtabs = TRUE, ...)
  names(dimnames(cm)) <- c("actual","predicted")

  n = sum(cm) # number of instances
  nc = nrow(cm) # number of classes
  diag = diag(cm) # number of correctly classified instances per class
  rowsums = apply(cm, 1, sum) # number of instances per class
  colsums = apply(cm, 2, sum) # number of predictions per class
  p = rowsums / n # distribution of instances over the classes
  q = colsums / n # distribution of instances over the predicted classes

  #accuracy
  accuracy = sum(diag) / n

  #per class prf
  recall = diag / rowsums
  precision = diag / colsums
  f1 = 2 * precision * recall / (precision + recall)
```

```
  #random/expected accuracy
  expAccuracy = sum(p*q)
  #kappa
  kappa = (accuracy - expAccuracy) / (1 - expAccuracy)

  #random guess
  rgAccuracy = 1 / nc
  rgPrecision = p
  rgRecall = 0*p + 1 / nc
  rgF1 = 2 * p / (nc * p + 1)


  classNames = names(diag)
  if(is.null(classNames)) classNames = paste("C",(1:nc),sep="")
  metrics = rbind(
    Accuracy = accuracy,
    Precision = precision,
    Recall = recall,
    F1 = f1,
    Kappa = kappa,
    RandomGuessAccuracy = rgAccuracy,
    RandomGuessPrecision = rgPrecision,
    RandomGuessRecall = rgRecall,
    RandomGuessF1 = rgF1)

  colnames(metrics) = classNames
  return(list(ConfusionMatrix = cm, Metrics = metrics))
}
```

Funkcja `roc_curve` wyznacza krzywą *ROC* i zwraca wykres, na którym znajduje się krzywa *ROC* oraz obszar pod nią.

```
roc_curve <- function(data, observed, predicted) {
  data <- data[, c(observed, predicted)]
  data[[observed]] <- as.numeric(as.character(data[[observed]]))
  rxRocCurve(actualVarName = observed,
             predVarNames = predicted,
             data = data)
}
```

Funkcja `auc` zwraca obszar pod krzywą `ROC`.

```
auc <- function(data, observed, predicted) {
  data <- data[, c(observed, predicted)]
  data[[observed]] <- as.numeric(as.character(data[[observed]]))
  rxAuc(rxRoc(actualVarName = observed,
              predVarNames = predicted,
              data = data))
}
```

W pierwszej kolejności zobaczmy, jak z zadaniem podziału urządzeń na te, które w ciągu najbliższych 30 cykli ulegną awarii, i te, których czas bezawaryjnej pracy będzie dłuższy, poradził sobie prosty model regresji logistycznej.

```
formula <- "label1 ~ a11 + a4 + a15 + a21 + a17 + a3 + a20 + a2 + a12 + a7 + s11 + s4
+ s12 + s7 + s15"

model <- MicrosoftML::rxLogisticRegression(formula = formula,
                                           data = train_df,
                                           type = "binary",
                                           l1Weight = 0.8,
                                           l2Weight = 0.8,
                                           maxIterations = 20,
                                           normalize = "yes")
```

Obliczone na podstawie danych treningowych metryki wyglądają następująco:

```
model_predictions <- rxPredict(modelObject = model,
                                    data = train_df,
                                    extraVarsToWrite = "label1")
model_empirical_metrics <- evaluate_classifier(actual = "label1",
                                                    predicted = "PredictedLabel",
                                                    data = model_predictions)
model_empirical_metrics
```

```
$ConfusionMatrix
       predicted
actual      0      1
      0 17531      0
      1  3097      3

$Metrics
                                0           1
Accuracy                0.849886094 0.8498860937
Precision               0.849864262 1.0000000000
Recall                  1.000000000 0.0009677419
F1                      0.918839592 0.0019336126
Kappa                   0.001643547 0.0016435468
RandomGuessAccuracy     0.500000000 0.5000000000
RandomGuessPrecision    0.849740681 0.1502593185
RandomGuessRecall       0.500000000 0.5000000000
RandomGuessF1           0.629558472 0.2310759942
```

Najpierw przyjrzymy się macierzy błędów. Proszę zwrócić uwagę, że w wierszach macierzy błędów znajdują się prawdziwe wartości, w kolumnach wyniki predykcji, a w pierwszej komórce – liczba prawdziwych predykcji negatywnych.

W danych znajduje się 17 531 przykładów należących do klasy negatywnej (17 531 + 0) oraz 3100 przykładów klasy pozytywnej (3097 + 3). Model przypisał 20 628 przykładów (17 531 + 3097) do klasy negatywnej, a 3 przykłady (0 + 3) zostały sklasyfikowane jako pozytywne.

Na podstawie macierzy błędów możemy powiedzieć, że:

- dane są niezrównoważone, prawie 85% przykładów (17 531 z 20 631) należy

do klasy negatywnej;

- model prawidłowo sklasyfikował 17 531 przykładów klasy negatywnej (*TN*) i 3 przykłady klasy pozytywnej (*TP*);
- model popełnił 3097 błędów drugiego rodzaju (*FN*) i 0 błędów pierwszego rodzaju (*FP*).

Popatrzmy teraz na kryteria oceny modelu wyprowadzone z macierzy błędów.

Trafność modelu wynosi 0,85 (17 534/20 631), co mogłoby świadczyć o dobrym modelu, gdyby nie to, że mamy do czynienia z niezrównoważonymi danymi.

Precyzja przewidywania pozytywnego jest równa 1 (3/3 + 0), a precyzja przewidywania negatywnego wyniosła 0,85 (17 531/(17 531 + 3097)). Proszę zwrócić uwagę, że 100% precyzja modelu niewiele znaczy, bo została obliczona na podstawie 3 przykładów.

Czułość modelu jest bliska zeru (3/3 + 3097), za to jego specyficzność jest idealna (17 531/(17 531 + 0)).

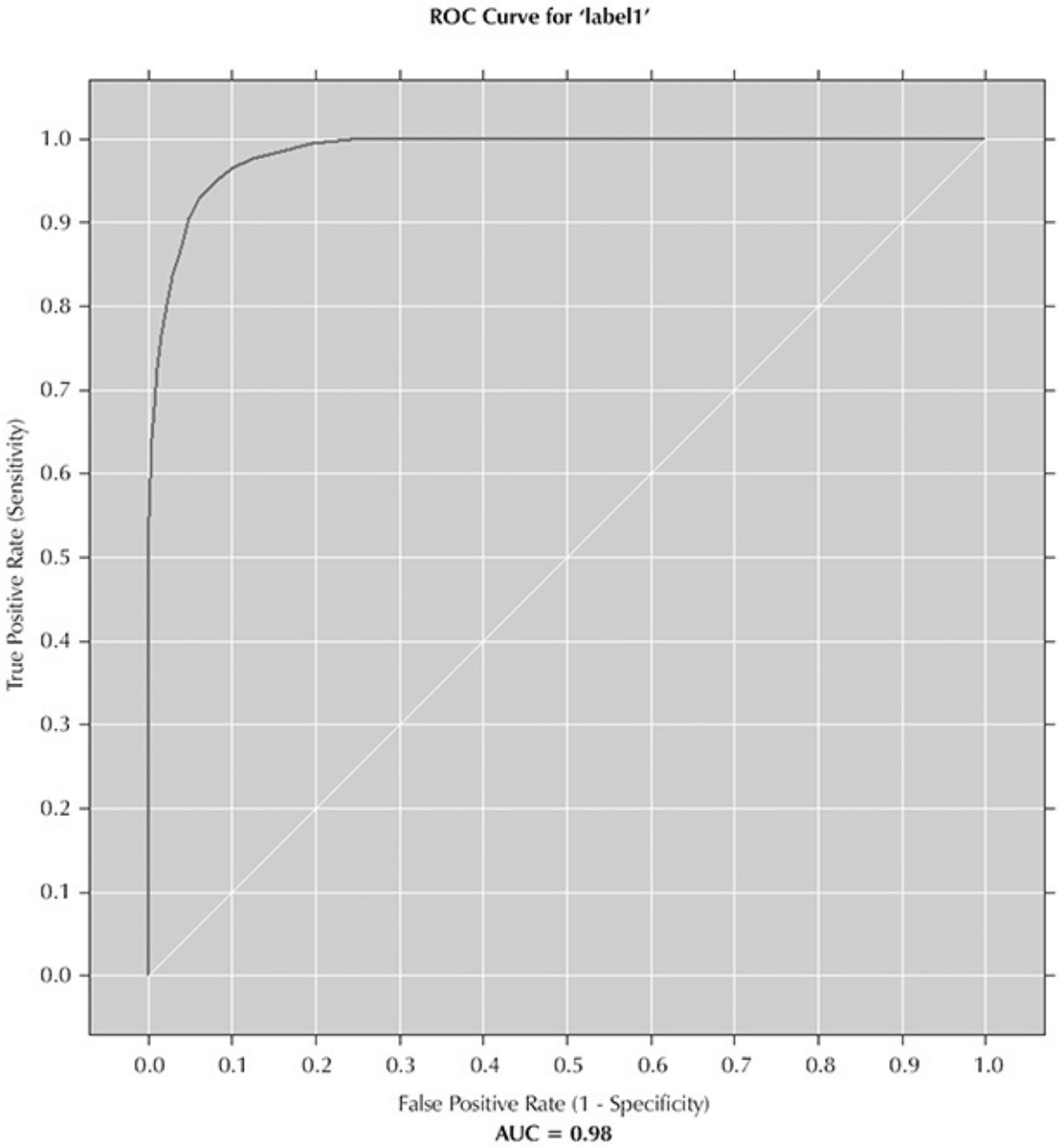
F-miara również jest bliska zeru $\left(2 * \frac{1 * 0{,}0009}{1 + 0{,}0009}\right)$. Teraz widzimy, dlaczego jest ona obliczana jako średnia harmoniczna precyzji i czułości. Dzięki temu żaden z członów równania nie dominuje, czyli żeby *F*-miara była duża, ani precyzja, ani czułość nie mogą być małe.

Współczynnik *Kappa* modelu wynosi 0,001 $\left(\frac{0{,}85 - 0{,}849}{1 - 0{,}849}\right)$, a więc jest bardzo niski.

Skoro obliczone dla danych treningowych *F*-miara i współczynnik *Kappa* są bliskie zeru, tzn. że błąd systematyczny modelu jest bardzo duży. Z tego wynika, że model jest niedopasowany do danych treningowych. W tej sytuacji bliska idealnej krzywa *ROC* (obszar pod krzywą wynosi 0,98) świadczy wyłącznie o tym, że dane są niezrównoważone, a model dobrze sobie radzi z ich

sortowaniem według prawdopodobieństwa przynależności do klasy pozytywnej (rys. 9.5).

```
roc_curve(data = model_predictions,
            observed = "label1",
            predicted = "Probability.1")
```



Rysunek 9.5. Przekonaliśmy się, że kompletnie bezużyteczny model może mieć prawie idealną krzywą ROC i AOC bliski 1

Skoro ten model regresji logistycznej nie dopasował się do danych treningowych, spróbujmy zbudować najprostszy model sztucznej sieci neuronowej.

```
model <- MicrosoftML::rxNeuralNet(formula,
                                  data = train_df,
                                  type = "binary",
                                  normalize = "yes",
                                  numIterations = 40,
                                  optimizer = adaDeltaSgd(),
                                  numHiddenNodes = 1,
                                  initWtsDiameter = 0.1)
```

Ponieważ precyzja modelu regresji logistycznej była idealna, ale czułość bardzo mała, zwiększmy czułość tego modelu, obniżając wartość progową predykcji z 50% do 40%.

```
model_predictions <- rxPredict(modelObject = model,
                               data = train_df,
                               extraVarsToWrite = "label1")
model_predictions$PredictedLabel <- ifelse(model_predictions$Probability.1<=0.4, 0,1)
```

Sprawdźmy błąd systematyczny modelu sztucznej sieci neuronowej.

```
model_empirical_metrics <- evaluate_classifier(actual = "label1",
                                               predicted = "PredictedLabel",
                                               data = model_predictions)
model_empirical_metrics
```

```
$ConfusionMatrix
       predicted
actual      0      1
      0 17464     67
      1  1149   1951

$Metrics
                                 0         1
Accuracy                 0.9410596 0.9410596
Precision                0.9382690 0.9667988
Recall                   0.9961782 0.6293548
F1                       0.9663568 0.7624072
Kappa                    0.7304698 0.7304698
RandomGuessAccuracy      0.5000000 0.5000000
RandomGuessPrecision     0.8497407 0.1502593
RandomGuessRecall        0.5000000 0.5000000
RandomGuessF1            0.6295585 0.2310760
```

Model popełnił 1216 błędów i jego trafność wyniosła 94%. Wiemy, że dane są niezrównoważone, a więc wysoka trafność nie musi oznaczać małego błędu systematycznego. Lepszym kryterium będzie współczynnik *Kappa*. Wynosi on 73%, czyli trafność modelu jest wyraźnie wyższa od trafności losowego zgadywania.

F-miara równa się 76%, co potwierdza zdolność modelu do uczenia się na niezrównoważonych danych. *F*-miara jest średnią precyzji i czułości, sprawdźmy więc, czy model częściej popełnia błędy typu pierwszego, czy drugiego. Precyzja modelu wynosi 96%, a więc model rzadko błędnie klasyfikował przykłady negatywne jako pozytywne. Za to czułość modelu jest niska, bo wynosi zaledwie 63%.

Proszę zwrócić uwagę, że obszar *AUC* modelu – który choć nie bezbłędnie, to jednak nauczył się klasyfikować urządzenia – jest taki sam jak pierwszego,

kompletnie bezużytecznego modelu.

```
auc(data = model_predictions,
     observed = "label1",
     predicted = "Probability.1")
```

```
0.9847748
```

Po sprawdzeniu za pomocą błędu systematycznego, że model jest w stanie nauczyć się na danych treningowych, możemy ocenić jego zdolności uogólniania wiedzy, czyli oszacować zmienność modelu. Metoda *k*-krotnej walidacji krzyżowej pozwoli nam wydzielić dane walidacyjne z treningowych i obliczyć interesujące nas metryki.

```
for (i in 1:10) {
  model <- rxNeuralNet(formula = formula,
                       data = train_df[kf$which!=i,],
                       type = "binary",
                       normalize = "yes",
                       numIterations = 40,
                       optimizer = adaDeltaSgd(),
                       numHiddenNodes = 1,
                       initWtsDiameter = 0.1)
  scoreDS <- rxPredict(model, data = train_df[kf$which==i,], extraVarsToWrite =
c("label1"))
  scoreDS$PredictedLabel <- ifelse(scoreDS$Probability.1<=0.4, 0,1)

  model_generalization_metrics <- evaluate_classifier(actual = "label1",
                                                      predicted =
"PredictedLabel",
                                                      data = scoreDS)
  Precision <- rbind(Precision, model_generalization_metrics$Metrics[2,2])
  Recall <- rbind(Recall, model_generalization_metrics$Metrics[3,2])
  F1 <- rbind(F1, model_generalization_metrics$Metrics[4,2])
  Kappa <- rbind(Kappa, model_generalization_metrics$Metrics[5,2])
}
sprintf ('Model mean F1: %.2f, stdev F1 %.3f' , mean(F1), sd(F1))
sprintf ('Model mean Kappa: %.2f, stdev Kappa: %.3f' ,mean(Kappa), sd(Kappa))
sprintf ('Model mean Precision: %.2f, stdev Precision: %.3f' ,mean(Precision),
sd(Precision))
sprintf ('Model mean Recall: %.2f, stdev Recall: %.3f' ,mean(Recall), sd(Recall))
```

```
"Model mean F1: 0.69, stdev F1 0.058"
"Model mean Kappa: 0.64, stdev Kappa: 0.054"
"Model mean Precision: 0.87, stdev Precision: 0.187"
"Model mean Recall: 0.65, stdev Recall: 0.230"
```

Obliczone na danych walidacyjnych wartości są mniejsze od uzyskanych dla danych testowych. Współczynnik *Kappa* spadł o 9%, *F*-miara o 8%, a precyzja o 9%. Za to czułość modelu wzrosła o 2%. Model jest bardzo stabilny, jego błąd aproksymacji mierzony *F*-miarą i współczynnikiem *Kappa* jest bliski zeru.

Poprawimy zdolności uczenia modelu, co powinno pozytywnie wpłynąć na jego czułość. W tym celu zwiększymy liczbę cykli uczenia do 100, zmniejszymy zakres początkowych wartości wag i dodamy kilka ukrytych węzłów. Przewidywany nadmierny wzrost czułości i związany z nim spadek precyzji spróbujemy zrównoważyć, podnosząc o 5% wartość brzegową predykcji.

```
for (i in 1:10) {
  model <- rxNeuralNet(formula = formula,
                       data = train_df[kf$which!=i,],
                       type = "binary",
                       normalize = "yes",
                       numIterations = 100,
                       optimizer = adaDeltaSgd(),
                       numHiddenNodes = 8,
                       initWtsDiameter = 0.05)
  scoreDS <- rxPredict(model, data = train_df[kf$which==i,], extraVarsToWrite =
c("label1"))
  scoreDS$PredictedLabel <- ifelse(scoreDS$Probability.1<=0.45, 0,1)

  model_generalization_metrics <- evaluate_classifier(actual = "label1",
                                                      predicted =
"PredictedLabel",
                                                      data = scoreDS)
  Precision <- rbind(Precision, model_generalization_metrics$Metrics[2,2])
  Recall <- rbind(Recall, model_generalization_metrics$Metrics[3,2])
  F1 <- rbind(F1, model_generalization_metrics$Metrics[4,2])
  Kappa <- rbind(Kappa, model_generalization_metrics$Metrics[5,2]              )
}
sprintf ('Model mean F1: %.2f, stdev F1 %.3f' , mean(F1), sd(F1))
sprintf ('Model mean Kappa: %.2f, stdev Kappa: %.3f' ,mean(Kappa), sd(Kappa))
sprintf ('Model mean Precision: %.2f, stdev Precision: %.3f' ,mean(Precision),
sd(Precision))
sprintf ('Model mean Recall: %.2f, stdev Recall: %.3f' ,mean(Recall), sd(Recall))
```

```
"Model mean F1: 0.73, stdev F1 0.074"
"Model mean Kappa: 0.69, stdev Kappa: 0.077"
"Model mean Precision: 0.82, stdev Precision: 0.175"
"Model mean Recall: 0.74, stdev Recall: 0.216"
```

Ten model spełnia nasze wymagania, a więc możemy przeprowadzić końcowy test jego jakości. W tym celu finalną wersję modelu nauczymy na wszystkich danych treningowych i ocenimy na danych testowych.

```
model <- rxNeuralNet(formula = formula,
                         data = train_df,
                         type = "binary",
                         normalize = "yes",
                         numIterations = 100,
                         optimizer = adaDeltaSgd(),
                         numHiddenNodes = 8,
                         initWtsDiameter = 0.05)

model_predictions <- rxPredict(modelObject = model,
                                    data = test_df,
                                    extraVarsToWrite = "label1")
model_predictions$PredictedLabel <- ifelse(model_predictions$Probability.1<=0.45, 0,1)

model_test_metrics <- evaluate_classifier(actual = "label1",
                                                    predicted = "PredictedLabel",
                                                    data = model_predictions)

model_test_metrics
```

```
$ConfusionMatrix
       predicted
actual  0  1
      0 74  1
      1  9 16

$Metrics
                                0         1
Accuracy                0.9000000 0.9000000
Precision               0.8915663 0.9411765
Recall                  0.9866667 0.6400000
F1                      0.9367089 0.7619048
Kappa                   0.7014925 0.7014925
RandomGuessAccuracy     0.5000000 0.5000000
RandomGuessPrecision    0.7500000 0.2500000
RandomGuessRecall       0.5000000 0.5000000
RandomGuessF1           0.6000000 0.3333333
```

Model popełnił 10 błędów, a że dane testowe liczyły 100 przykładów, to jego trafność wyniosła 90%. Niestety, czułość modelu okazała się niższa od spodziewanej i wyniosła zaledwie 64%. Ponieważ konsekwencje błędów drugiego rodzaju, czyli fałszywych negatywów, mogą być w tym przypadku bardzo poważne, z powrotem zwiększyliśmy czułość modelu, co przełożyło się na poprawę jego ogólnej jakości, w tym zwiększenie czułości z 64% do 82%.

```
model_predictions$PredictedLabel <- ifelse(model_predictions$Probability.1<=0.25, 0,1)
model_generalization_metrics$Metrics[2:5,2]
```

```
Precision    Recall         F1      Kappa
0.8718861 0.7852564 0.8263069 0.7972462
```

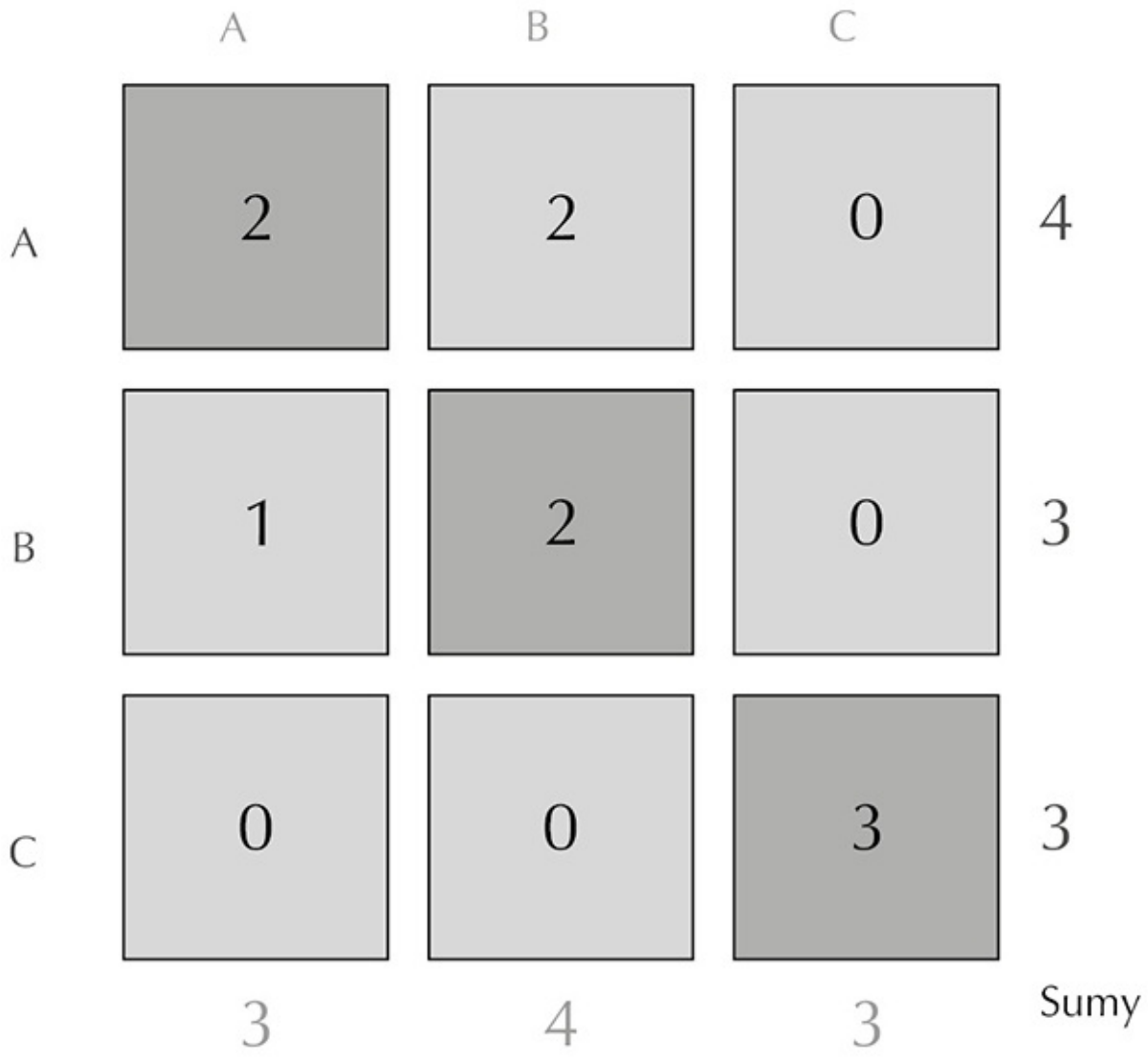
9.5.
Kryteria oceny modeli klasyfikacji wieloklasowej

Klasyfikatory wieloklasowe przypisują przykłady do jednej z wielu klas. Podstawą ich oceny również jest macierz błędów. Liczba wierszy i kolumn macierzy błędów odpowiada liczbie klas, a więc w tym przypadku będzie ona większa od dwóch.

W poprzednim punkcie przekonaliśmy się, że wyniki oceny klasyfikatorów zależą od rozkładu zmiennej wyjściowej. Im bardziej niezrównoważone dane, tym większe ryzyko, że słaby model dostanie wysokie oceny. Ponieważ rozkład zmiennych przyjmujących więcej wartości często jest bardziej niezrównoważony niż rozkład zmiennych binarnych, ryzyko to jest jeszcze większe w przypadku klasyfikatorów wieloklasowych. Proszę pamiętać, że zrównoważenie danych nie tylko poprawia jakość modeli, ale również ułatwia ich ocenę.

9.5.1.
Macierz błędów

W kolumnach macierzy błędów umieszcza się zliczone częstości klas, a w jej wierszach zliczone predykcje. W ten sposób na przecięciu wierszy i kolumn uzyskamy liczby przykładów należących do klasy, które zostały przypisane do tej klasy przez model. Na przekątnej macierzy znajdują się prawidłowe predykcje, pozostałe komórki zawierają błędy modelu (rys. 9.6).



Rysunek 9.6. Trzy z dziesięciu przykładów należały do klasy A, cztery do klasy B i trzy do klasy C. Model cztery przykłady przypisał do klasy A i po trzy do klas B i C

Macierz błędów jest łatwa w interpretacji, ale do porównywania modeli, i przede wszystkim do ich automatycznej optymalizacji, lepsze są pojedyncze kryteria. Zakładając, że każdy błąd jest równie kosztowny, można je obliczyć, używając funkcji progowej 0–1. Przyjmując, że koszt przypisania przykładu klasy A do klas B lub C kosztuje 1, tyle samo co koszt przypisania przykładu klasy B do klas A lub C, i tyle samo, co koszt przypisania przykładu klasy C do klas A lub B, z macierzy błędów wyprowadzimy pozostałe kryteria oceny klasyfikatorów wieloklasowych.

9.5.2. Metryki klasy większościowej

Jeżeli dane są silnie niezrównoważone, na przykład gdy 80% przykładów lub więcej należy do jednej klasy, wyniki predykcji dla tej klasy zdominują oceny modelu. W takim przypadku warto znać metryki dla klasy większościowej, takie jak trafność, precyzja, czułość i *F*-miara. Oblicza się je tak samo jak dla klasyfikatorów binarnych, dzieląc przykłady na dwa podzbiory: należące i nienależące do klasy większościowej.

9.5.3. Metryki poszczególnych klas

Metryki, takie jak trafność czy precyzja, można też obliczyć dla poszczególnych klas. Wyznacza się je metodą *One-vs-All*, osobno zliczając wartości w każdej kolumnie macierzy błędów.

Na przykład wyznaczona na podstawie pokazanej na rysunku 9.6 macierzy błędów precyzja

$$PPV = \frac{TP}{TP + FP}$$

dla klas A, B i C wynosi odpowiednio:

- precyzja dla kasy A równa się $\frac{2}{2+2+0} = 0{,}5$,

  precyzja dla kasy B równa się $\frac{2}{1+2+0} = 0{,}67$,

  precyzja dla kasy C równa się $\frac{3}{0+0+3} = 1$.

W ten sam sposób możemy wyznaczyć pozostałe metryki, na przykład czułość

$$\textit{czułość} = \frac{TP}{TP + FN}$$

- czułość dla kasy A równa się $\frac{2}{2+1+0} = 0{,}67$,
- czułość dla kasy B równa się $\frac{2}{2+2+0} = 0{,}5$,
- czułość dla kasy C równa się $\frac{3}{0+0+3} = 1$.

9.5.4. Średnie makro

Metryki policzone dla poszczególnych klas można uśrednić. W te sposób otrzymamy pojedyncze wartości, które mogą być użyte do porównywania i automatycznej optymalizacji modeli.

Makrotrafność wyznaczymy jako średnią arytmetyczną trafności dla poszczególnych klas

$$\textit{makrotrafność} = \frac{\sum_l \frac{TP + TN}{TP + FN + FP + TN}}{l}$$

uzyskując w ten sposób średnią trafność modelu.

Makroprecyzja jest średnią arytmetyczną precyzji poszczególnych klas

$$\textit{makro PPV} = \frac{\sum_l \frac{TP}{TP + FP}}{l}$$

Makroprecyzja przykładowego modelu wynosi $\frac{0{,}67 + 0{,}5 + 1}{3}$.

Makroczułość jest średnią arytmetyczną czułości poszczególnych klas

$$\textit{makroczułość} = \frac{\sum_l \frac{TP}{TP + FN}}{l}$$

Makroczułość przykładowego modelu wynosi $\frac{0{,}67 + 0{,}5 + 1}{3} = 0{,}72$.

Makro *F*-miara jest średnią harmoniczną makroprecyzji i makroczułości

$$\textit{makro F-miara} = \frac{(1 + \beta)^2 * \textit{makroprecyzja} * \textit{makroczułość}}{\beta^2 * \textit{makroprecyzja} + \textit{makroczułość}}$$

Średnie makro nie biorą pod uwagę liczności poszczególnych klas, a więc możliwe jest uzyskanie dużej wartości miary makro przez model, który zupełnie nie radzi sobie z przypisywaniem przykładów do niektórych klas. Z tego powodu średnich makro nie należy używać, jeżeli dysponujemy niezrównoważonymi danymi. Wady tej pozbawione są średnie mikro.

9.5.5. Średnie mikro

Średnie mikro oblicza się bezpośrednio na podstawie macierzy pomyłek. W tym celu należy policzyć *TP*, *TN*, *FP* i *FN* dla każdej klasy, a następnie na ich podstawie wyznaczyć mikroprecyzję, mikroczułość i mikro *F*-miarę.

Mikroprecyzja mierzy precyzje predykcji pozytywnych wszystkich klas

$$\textit{mikro PPV} = \frac{\sum_l TP}{\sum_l (TP + FP)}$$

Mikroprecyzja przykładowego modelu wynosi $\frac{2 + 2 + 3}{4 + 3 + 3} = 0{,}7$.

Mikroczułość mierzy czułość poprawnych klasyfikacji przykładów wszystkich klas

$$mikroczułość = \frac{\sum_l TP}{\sum_l (TP + FN)}$$

Mikroprecyzja przykładowego modelu wynosi $\frac{2 + 2 + 3}{3 + 4 + 3} = 0{,}7$.

Mikro F-miara jest średnią harmoniczną mikroprecyzji i mikroczułości

$$mikro\ F\text{-}miara = \frac{(1 + \beta)^2 * mikroprecyzja * mikroczułość}{\beta^2 * mikroprecyzja + mikroczułość}$$

Warto zauważyć, że mikroczułość klasyfikatorów wieloklasowych zawsze jest równa ich mikroprecyzji. Wynika to z tego, że błędne przypisanie przykładu klasy A do klasy B jest fałszywym pozytywem dla klasy B i jednocześnie fałszywym negatywem dla klasy A. **Ponieważ średnia dwóch takich samych wartości jest równa uśrednianym wartościom, jedyną używaną średnią mikro jest mikro *F*-miara**.

9.6. Ocena modelu klasyfikującego urządzenia do przeglądu

Przedstawione powyżej metryki, uzupełnione o wyniki modelu losowego, zwraca funkcja `evaluate_classifier`.

```
<- function(actual=NULL, predicted=NULL, cm=NULL){
  if(is.null(cm)) {
    naVals = union(which(is.na(actual)), which(is.na(predicted)))
    if(length(naVals) > 0) {
      actual = actual[-naVals]
      predicted = predicted[-naVals]
    }
    f = factor(union(unique(actual), unique(predicted)))
    actual = factor(actual, levels = levels(f))
    predicted = factor(predicted, levels = levels(f))
    cm = as.matrix(table(Actual=actual, Predicted=predicted))
  }

  n = sum(cm) # number of instances
  nc = nrow(cm) # number of classes
  diag = diag(cm) # number of correctly classified instances per class
  rowsums = apply(cm, 1, sum) # number of instances per class
  colsums = apply(cm, 2, sum) # number of predictions per class
  p = rowsums / n # distribution of instances over the classes
  q = colsums / n # distribution of instances over the predicted classes

  #accuracy
  accuracy = sum(diag) / n

  #per class prf
  recall = diag / rowsums
  precision = diag / colsums
  f1 = 2 * precision * recall / (precision + recall)

  #macro prf
  macroPrecision = mean(precision)
  macroRecall = mean(recall)
```

```
macroF1 = mean(f1)

#1-vs-all matrix
oneVsAll = lapply(1 : nc,
                      function(i){
                        v = c(cm[i,i],
                                rowsums[i] - cm[i,i],
                                colsums[i] - cm[i,i],
                                n-rowsums[i] - colsums[i] + cm[i,i]);
                        return(matrix(v, nrow = 2, byrow = T))})

s = matrix(0, nrow=2, ncol=2)
for(i in 1:nc){s=s+oneVsAll[[i]]}

#avg accuracy
avgAccuracy = sum(diag(s))/sum(s)

#micro prf
microPrf = (diag(s) / apply(s,1, sum))[1];

#majority class
mcIndex = which(rowsums==max(rowsums))[1] # majority-class index
mcAccuracy = as.numeric(p[mcIndex])
mcRecall = 0*p;  mcRecall[mcIndex] = 1
mcPrecision = 0*p; mcPrecision[mcIndex] = p[mcIndex]
mcF1 = 0*p; mcF1[mcIndex] = 2 * mcPrecision[mcIndex] / (mcPrecision[mcIndex] + 1)

#random/expected accuracy
expAccuracy = sum(p*q)
#kappa
kappa = (accuracy - expAccuracy) / (1 - expAccuracy)

#random guess
rgAccuracy = 1 / nc
rgPrecision = p
rgRecall = 0*p + 1 / nc
rgF1 = 2 * p / (nc * p + 1)

classNames = names(diag)
if(is.null(classNames)) classNames = paste("C",(1:nc),sep="")

metrics = rbind(
  Accuracy = accuracy,
  Precision = precision,
  Recall = recall,
```

```
    F1 = f1,
    MacroAvgPrecision = macroPrecision,
    MacroAvgRecall = macroRecall,
    MacroAvgF1 = macroF1,
    AvgAccuracy = avgAccuracy,
    MicroAvgPrecision = microPrf,
    MicroAvgRecall = microPrf,
    MicroAvgF1 = microPrf,
    MajorityClassAccuracy = mcAccuracy,
    MajorityClassPrecision = mcPrecision,
    MajorityClassRecall = mcRecall,
    MajorityClassF1 = mcF1,
    Kappa = kappa,
    RandomGuessAccuracy = rgAccuracy,
    RandomGuessPrecision = rgPrecision,
    RandomGuessRecall = rgRecall,
    RandomGuessF1 = rgF1)

  colnames(metrics) = classNames

  return(list(ConfusionMatrix = cm, Metrics = metrics))
}
```

Tak jak we wcześniejszych punktach rozdziału, oceniany model nauczymy na 15 zmiennych najsilniej skorelowanych ze zmienną wyjściową. Do uczenia i oceny modelu użyjemy wzbogaconych danych.

W pierwszej kolejności zbudujemy domyślny model regresji logistycznej, tj. podamy wartości tylko 3 wymaganych parametrów funkcji `rxLogisticRegression`.

```
formula <- "label2 ~ a11 + a15 + a4 + a21 + a17 + a3 + a20 + a2 + a12 + a7 + s11 + s4
+ s12 + s7 + s15"
model <- MicrosoftML::rxLogisticRegression(formula = formula,
                                           data = train_df,
                                           type = "multiClass")
```

Sprawdźmy błąd systematyczny tak zbudowanego modelu.

```
model_predictions <- rxPredict(modelObject = model,
                                    data = train_df,
                                    extraVarsToWrite = "label2")
model_empirical_metrics <- evaluate_classifier(actual = model_predictions$label2,
                                                   predicted =
model_predictions$PredictedLabel)
model_empirical_metrics
```

```
$ConfusionMatrix
        Predicted
Actual      0      1      2
      0 17531      0      0
      1  1500      0      0
      2  1600      0      0

$Metrics
                                0            1           2
Accuracy                0.8497407
Precision               0.8497407          NaN         NaN
Recall                  1.0000000   0.00000000   0.0000000
F1                      0.9187674          NaN         NaN
MacroAvgPrecision             NaN
MacroAvgRecall          0.3333333
MacroAvgF1                    NaN
AvgAccuracy             0.8998271
MicroAvgPrecision       0.8497407
MicroAvgRecall          0.8497407
MicroAvgF1              0.8497407
MajorityClassAccuracy   0.8497407
MajorityClassPrecision  0.8497407
MajorityClassRecall     1.0000000
MajorityClassF1         0.9187674
Kappa                   0.0000000
RandomGuessAccuracy     0.3333333
RandomGuessPrecision    0.8497407   0.07270612   0.0775532
RandomGuessRecall       0.3333333   0.33333333   0.3333333
RandomGuessF1           0.4788321   0.11937448   0.1258307
```

Macierz błędów pokazuje, że model wszystkie przykłady przypisał do klasy większościowej, czyli do klasy 0.

Trafność modelu wynosi 85% i odpowiada procentowi przykładów klasy większościowej.

Ponieważ model nie zwrócił żadnych predykcji klas 1 i 2, precyzja dla tych klas nie została policzona (w języku R wynikiem dzielenia 0 przez 0 jest wartość `NaN`, *Not a Number*). Precyzja predykcji klasy 0 wynosi 85%.

Czułość dla klasy 0 wynosi 1, a czułość dla pozostałych klas jest równa 0 (wynikiem dzielenia 0 przez dowolną liczbę jest 0).

Skoro trafność dla klas większościowych nie jest liczbą, to nie można policzyć jej średniej harmonicznej, a więc F-miara dla klas 1 i 2 również nie jest liczbą. *F*-miara dla klasy 0 wynosi 92%.

Ze średnich makro można było wyznaczyć jedynie średnią czułość i średnią trafność, pozostałe średnie makro nie są liczbami. Makroczułość modelu wynosi 33%. Proszę zwrócić uwagę, że średnie makro nie uwzględniają liczności poszczególnych klas. Na przykład idealna czułość dla klasy 0 ma taki sam wynik średniej czułości jak zerowa czułość znacznie mniej licznej klasy 1.

Średnia trafność modelu wynosi 89%, a więc jest większa niż trafność dla klasy 0. Mamy więc do czynienia z paradoksem dokładności.

Wszystkie średnie mikro wynoszą 85%, a współczynnik *Kappa* modelu jest równy 0. **Zerowy współczynnik** ***Kappa*** **oznacza, że model nie był w stanie nauczyć się czegokolwiek na danych treningowych**.

Sprawdźmy, czy powodem niepowodzenia było użycie domyślnych wartości hiperparametrów. Regresja logistyczna jest prostym modelem, ale nawet w przypadku prostych modeli uczenia maszynowego nie należy polegać na domyślnych wartościach ich hiperparametrów.

Wyłączenie automatycznej normalizacji zmiennych i wydłużenie uczenia przez zmniejszenie wartości `optTol` pozwoliło zbudować lepszy model.

```
model <- MicrosoftML::rxLogisticRegression(formula = formula,
                                           data = train_df,
                                           type = "multiClass",
                                           optTol = 1e-10,
                                           normalize = "no")
…
model_empirical_metrics
```

```
$ConfusionMatrix
        Predicted
Actual       0      1      2
       0 17212    292     27
       1   695    527    278
       2    41    216   1343

$Metrics
                              0           1           2
Accuracy              0.9249188
Precision             0.9589926   0.50917874   0.8149272
Recall                0.9818037   0.35133333   0.8393750
F1                    0.9702641   0.41577909   0.8269704
MacroAvgPrecision     0.7610329
MacroAvgRecall        0.7241707
MacroAvgF1            0.7376712
AvgAccuracy           0.9499459
MicroAvgPrecision     0.9249188
MicroAvgRecall        0.9249188
MicroAvgF1            0.9249188
MajorityClassAccuracy 0.8497407
MajorityClassPrecision0.8497407
MajorityClassRecall   1.0000000
MajorityClassF1       0.9187674
Kappa                 0.7007802
RandomGuessAccuracy   0.3333333
RandomGuessPrecision  0.8497407   0.07270612   0.0775532
RandomGuessRecall     0.3333333   0.33333333   0.3333333
RandomGuessF1         0.4788321   0.11937448   0.1258307
```

Model nieźle radzi sobie z przykładami klas 0 i 2, gorzej z przykładami należącymi do najmniej licznej klasy 1. Dlatego makro *F*-miara wynosi 74% i jest wyraźnie mniejsza niż mikro *F*-miara, która wynosi 92%. Współczynnik *Kappa* równy 70% oznacza, że model nauczył się przypisywać przykłady do odpowiednich kategorii.

Zanim wdrożymy model do użycia, powinniśmy ocenić jego zmienność, czyli przekonać się, jak sobie radzi z klasyfikacją przykładów niewidzianych podczas uczenia. Jak zwykle, zrobimy to metodą 10-krotnej walidacji krzyżowej.

```
MicroAvgF1 <- as.numeric()
Kappa <- as.numeric()
MacroAvgF1 <- as.numeric()
kf<-cvFolds(nrow(train_df), K = 10, type = "random")
for (i in 1:10) {
  model <- rxLogisticRegression(formula = formula,
                            data = train_df[kf$which!=i,],
                            type = "multiClass",
                            optTol = 1e-10,
                            normalize = "no")
  scoreDS <- rxPredict(model, data = train_df[kf$which==i,], extraVarsToWrite =
c("label2"))

  model_generalization_metrics <- evaluate_classifier(actual = scoreDS$label2,
                                                      predicted =
scoreDS$PredictedLabel)

  MicroAvgF1 <- rbind(MicroAvgF1, model_generalization_metrics$Metrics[11,2])
  Kappa <- rbind(Kappa, model_generalization_metrics$Metrics[16,2])
  MacroAvgF1 <- rbind(MacroAvgF1, model_generalization_metrics$Metrics[7,2])
}
sprintf ('Model mean MicroAvgF1: %.2f, stdev MicroAvgF1 %.3f' , mean(MicroAvgF1),
sd(MicroAvgF1))
sprintf ('Model mean MacroAvgF1: %.2f, stdev MacroAvgF1: %.3f' ,mean(MacroAvgF1),
sd(MacroAvgF1))
sprintf ('Model mean Kappa: %.2f, stdev Kappa: %.3f' ,mean(Kappa), sd(Kappa))
```

```
"Model mean MicroAvgF1: 0.92, stdev MicroAvgF1 0.005"
"Model mean MacroAvgF1: 0.73, stdev MacroAvgF1: 0.024"
"Model mean Kappa: 0.69, stdev Kappa: 0.023"
```

Średnie wyniki modeli są bardzo zbliżone do uzyskanych na danych testowych, a ich odchylenia standardowe są bliskie zeru. Zmienność modelu jest bardzo mała, co oznacza, że model nie dopasował się nadmiernie do danych treningowych. Pora przeprowadzić końcowy egzamin przy użyciu danych testowych.

```
model_predictions <- rxPredict(modelObject = model,
                                    data = test_df,
                                    extraVarsToWrite = "label2")

model_test_metrics <- evaluate_classifier(actual = model_predictions$label2,
                                                    predicted =
model_predictions$PredictedLabel)
model_test_metrics$ConfusionMatrix
model_test_metrics$Metrics[c(7,11,16),2]
```

```
       Predicted
Actual   0  1  2
     0 74  1  0
     1  9  4  2
     2  0  2  8
MacroAvgF1  MicroAvgF1       Kappa
 0.7001151   0.8600000   0.6078431
```

Rozkład zmiennej wyjściowej w danych testowych jest inny niż w danych treningowych i walidacyjnych, co przełożyło się na gorsze wyniki, w szczególności zmniejszenie współczynnika *Kappa*. Tak jak podejrzewaliśmy, model nie najlepiej sobie radzi z przypadkami klasy 1.

 Proszę samodzielnie zrównoważyć dane i zbudować model klasyfikatora wieloklasowego, którego *F*-miary dla wszystkich klas przekroczą 80%.

9.7. Interpretacja predykcji

Trafność i wiarygodność predykcji są ważnymi, ale nie powinny być jedynymi kryteriami oceny modeli uczenia maszynowego. Ostatnio coraz większą wagę przywiązuje się do interpretowalności ich predykcji. Po pierwsze dlatego, że traktowanie modeli uczenia maszynowego jak czarnych skrzynek lub magicznych wyroczni ogranicza zaufanie użytkowników do ich wyroków. Po drugie, bo wraz z coraz powszechniejszym użyciem modeli uczenia maszynowego zwiększa się liczba zastosowań, w których wyjaśnienie decyzji

jest równie ważne co sama decyzja. Na przykład, bank ma obowiązek podać przyczynę odmowy udzielenia kredytu. I tą przyczyną nie może być stwierdzenie, że tak zadecydował nasz komputer, a nikt nie rozumie powodów jego decyzji.

Najprostszy i historycznie pierwszy sposób wyjaśnienia predykcji polega na analizie samego modelu, w szczególności na ocenie siły predykcyjnej poszczególnych zmiennych wejściowych. Te informacje o modelu zwraca funkcja `summary`.

```
summary(model)
```

```
MicrosoftML::rxFastTrees(formula = formula, data = train_df,
     type = "regression", numTrees = 300, numLeaves = 200, learningRate = 0.05,
     minSplit = 1, exampleFraction = 1, featureFraction = 1, splitFraction = 1,
     numBins = 100)

Predictor for: RUL~a4+a11+a21+a15+a20+a17+a12+a7+a2+a3+s11+s4+s12+s7+s15
Data: train_df

Per-feature gain:
a4: 1
a21: 0.4804679
a11: 0.3741538
a15: 0.3716771
a3: 0.3005111
a2: 0.2862639
a20: 0.2740136
a7: 0.2666057
a17: 0.2626515
a12: 0.2626241
s15: 0.2242778
s7: 0.2165713
s11: 0.2138214
s4: 0.2104447
s12: 0.2045635
```

Jednak jest to opis modelu, a nie konkretnej predykcji. Trudno na jego podstawie powiedzieć, czemu przewidywany czas bezawaryjnej pracy jednego urządzenia

został oszacowany na 120 cykli, a innego na 25. Ponadto dla niektórych algorytmów uczenia maszynowego, na przykład dla sztucznych sieci neuronowych, nie jesteśmy w stanie uzyskać informacji o sile predykcyjnej poszczególnych zmiennych wejściowych.

Znalezienie uniwersalnej, dającej się zastosować do dowolnego modelu uczenia maszynowego metody wyjaśniania predykcji jest przedmiotem intensywnych, trwających ponad 20 lat badań. Ich efektem jest opracowanie dwóch głównych metod wyjaśniania predykcji.

Metoda *LIME* (*Local Interpretable Model-Agnostic Explanations*) polega na uczeniu kopii modelu na wybranych, najbardziej przypominających oceniany model przykładach [36]. Predykcje prostszego, nauczonego na kilku, góra kilkudziesięciu przykładach modelu można łatwiej wyjaśnić. W tym celu porównuje się wyniki predykcji zwrócone przez lokalny model dla zmodyfikowanego na wiele sposobów przykładu.

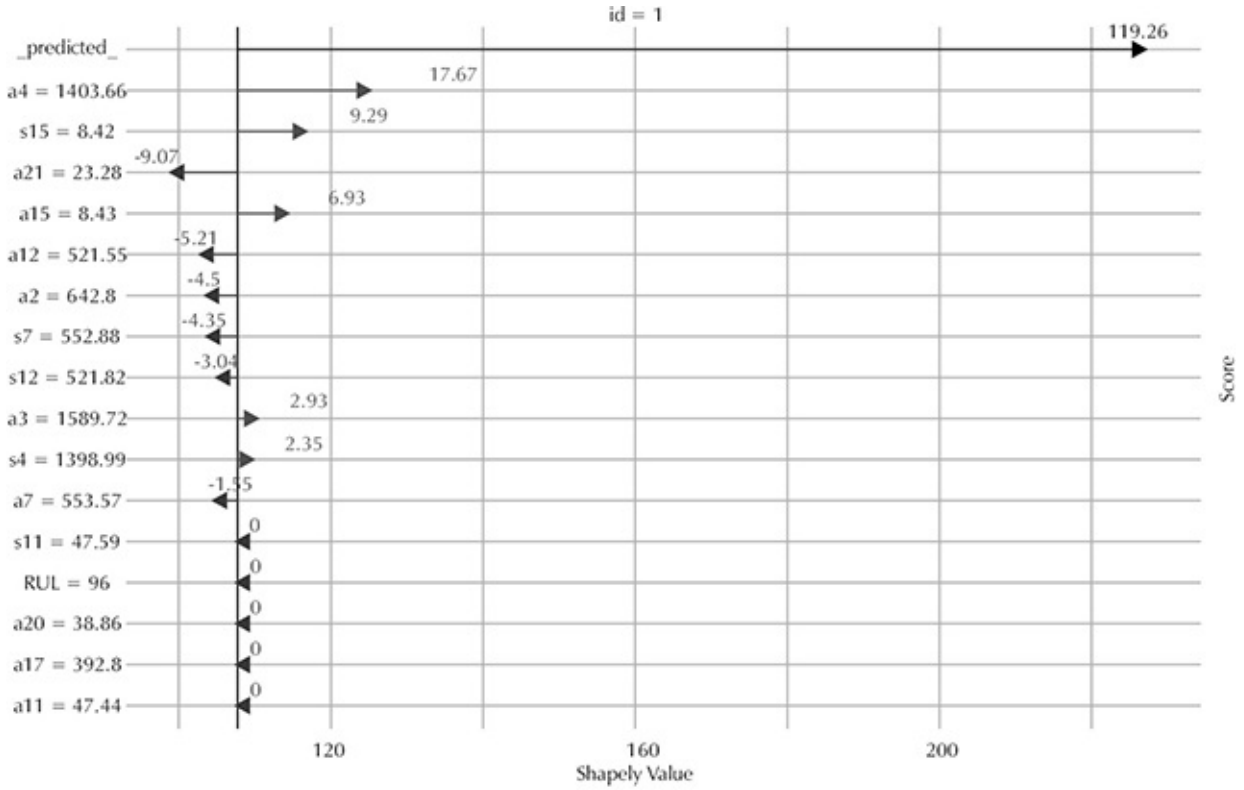
Metoda *SHAP* (*Shapley Additive Explanation*) polega na obliczeniu wpływu wartości poszczególnych zmiennych na wynik konkretnej predykcji [37]. Stosuje się kilka metod dekompozycji wpływu poszczególnych zmiennych – najprostsza polega na zastąpieniu wartości zmiennej wartością losową, wielokrotnym powtórzeniu predykcji i porównaniu uzyskanych wyników. Inne metody wykorzystują teorię gier do znalezienia optymalnych wartości poszczególnych zmiennych.

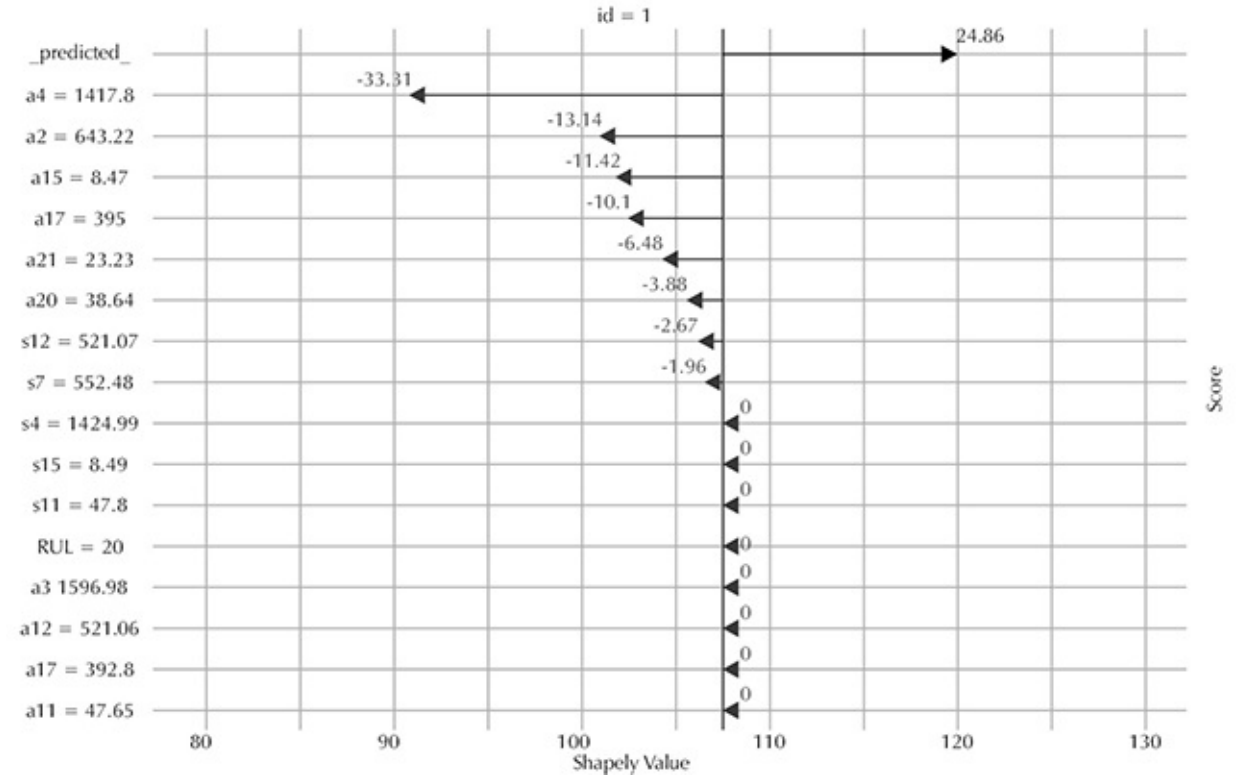
Zobaczmy, jak metoda SHAP wyjaśnia wyniki dwóch predykcji: w jednym przypadku przewidywany czas bezawaryjnej pracy urządzenia wyniósł 120 cykli, w drugim 25. Użyjemy do tego bibliotek `reticulate`, `DALEX` i `shapper`. Dwie pierwsze należy zainstalować ze strony projektu CRAN, najnowsza wersja trzeciej biblioteki jest dostępna w serwisie GitHub. Następnie korzystając z biblioteki `shapper`, należy zainstalować używane przez nią pakiety języka Python.

```
install.packages("reticulate ", repos = "https://cran.r-project.org/")
install.packages("DALEX2", repos = "https://cran.r-project.org/")
devtools::install_github("ModelOriented/shapper")
shapper::install_shap()
```

Implementacje algorytmów uczenia maszynowego są różne. Funkcja `explain` biblioteki `DALEX` tworzy ich uniwersalną reprezentację zawierającą nauczony model i dane treningowe. Ta reprezentacja może zostać użyta do wyjaśnienia predykcji funkcją `shap` (rys. 9.7).

```
 exp_model <- DALEX2::explain(model,
                          data = train_df[,c("RUL","a4","a11","a21","a15","a20","a17",
"a12","a7",
                                                    "a2","a3","s11","s4","s12","s7","s15")])
new_observation <- test_df[100, c("RUL","a4","a11","a21","a15","a20","a17","a12","a7",
                                        "a2","a3","s11","s4","s12","s7","s15")]
ive_model <- shapper::shap(exp_model,
                      predict_function = rxPredict,
                      nsamples = as.integer(100),
                      new_observation = new_observation)
plot(ive_model)
```


id = 1
predicted
a4 = 1403.66
s15 = 8.42
a21 = 23.28
a15 = 8.43
a12 = 521.55
a2 = 642.8
s7 = 552.88
s12 = 521.82
a3 = 1589.72
s4 = 1398.99
a7 = 553.57
s11 = 47.59
RUL = 96
a20 = 38.86
a17 = 392.8
a11 = 47.44
119.26
17.67
9.29
-9.07
6.93
-5.21
-4.5
-4.35
-3.04
2.93
2.35
-1.55
0
0
0
0
0
120
160
200
Shapely Value
Score


id = 1
predicted
a4 = 1417.8
a2 = 643.22
a15 = 8.47
a17 = 395
a21 = 23.23
a20 = 38.64
s12 = 521.07
s7 = 552.48
s4 = 1424.99
s15 = 8.49
s11 = 47.8
RUL = 20
a3 1596.98
a12 = 521.06
a17 = 392.8
a11 = 47.65
24.86
-33.31
-13.14
-11.42
-10.1
-6.48
-3.88
-2.67
-1.96
0
0
0
0
0
0
0
0
80
90
100
110
120
130
Shapely Value
Score

Rysunek 9.7. Dekompozycja wpływu zmiennych na wyniki dwóch predykcji. Proszę zwrócić uwagę, że w pierwszym przykładzie wartość 8,43 zmiennej a15 wpłynęła na zwiększenie prognozy, a w drugim wartość 8,47 tej zmiennej wpłynęła na skrócenie przewidywanego czasu bezawaryjnej pracy silnika

W tym rozdziale poznaliśmy standardowe kryteria oceny modeli predykcyjnych. Umiejętność porównywania modeli i wybierania najlepszego wykorzystamy w następnym, ostatnim rozdziale.

Optymalizacja hiperparametrów polega raczej na wynikach eksperymentów niż na teorii.

William Koehrsen, Data Scientist w Cortex Intel

Rozdział 10
Optymalizacja i wdrożenie modeli

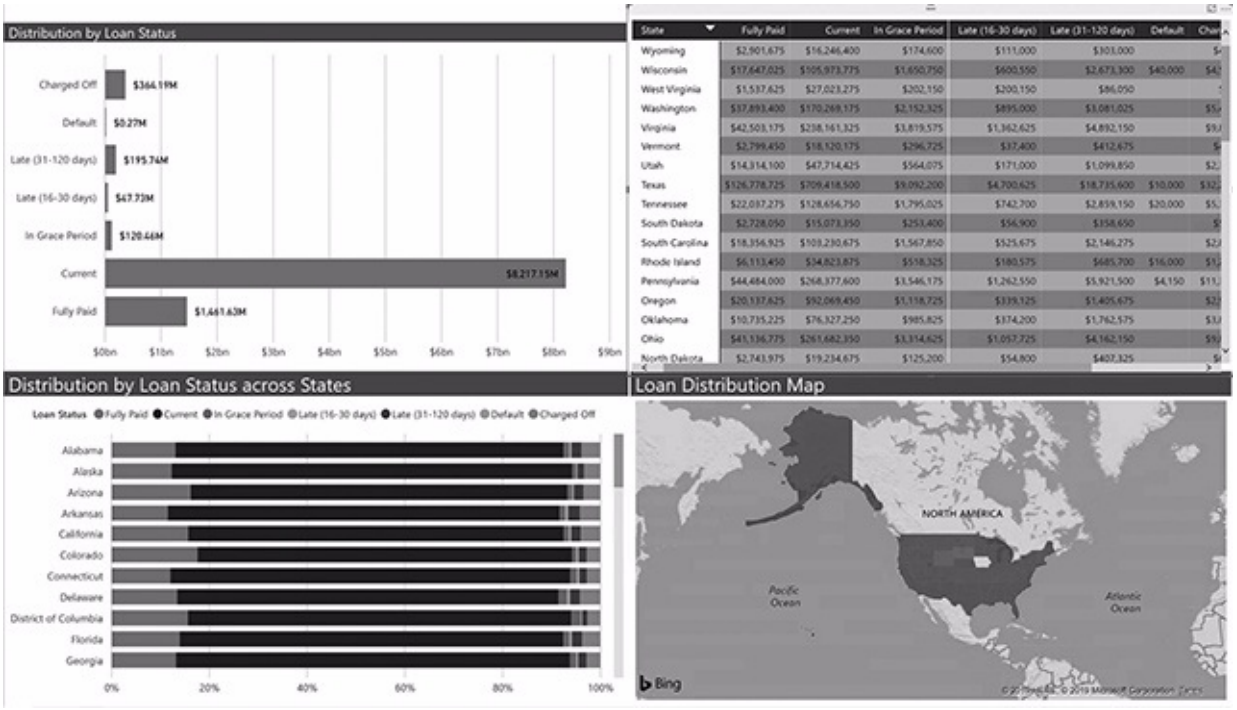
Nasz ostatni wspólny eksperyment będzie okazją do praktycznego zastosowania informacji z poprzednich rozdziałów. Budując model predykcyjny, tym razem skoncentrujemy się na jego optymalizacji, przede wszystkim na dobraniu optymalnych wartości hiperparametrów. Ponieważ będziemy pracować z dużym zbiorem danych, liczącym około miliona przykładów opisanych ponad setką zmiennych, wszystkie operacje wykonamy po stronie serwera SQL. Pozwoli to utrwalić znane i poznać nowe metody integracji serwera SQL z językami R i Python. Na końcu udostępnimy użytkownikom gotowy model i wyniki jego predykcji.

Jak zwykle, eksperyment przeprowadzimy zgodnie z wytycznymi metodyki CRISP-DM. Jednak w tym przypadku Państwa zadanie nie ograniczy się do zbudowania modelu według przedstawionych wskazówek. Chociaż byłby to działający model, to odpowiednio stosując poznane techniki uczenia maszynowego, będą Państwo mogli znacząco poprawić jego jakość. Zaczynajmy.

10.1.
Zrozumienie problemu

Gdybyśmy mieli kryształową kulę, pożyczalibyśmy pieniądze tylko tym, którzy nam je zwrócą. Firmy finansowe mogą skorzystać z analizy predykcyjnej, aby ograniczyć liczbę pożyczek oferowanych kredytobiorcom, którzy najprawdopodobniej ich nie spłacą. Ponieważ zwiększa to ich zyskowność, dzisiaj prawie wszystkie firmy finansowe przeprowadzają taką ocenę ryzyka kredytowego.

Naszym zadaniem jest zbudowanie modelu oceniającego ryzyko kredytowe dla fikcyjnej firmy pożyczkowej ze Stanów Zjednoczonych. Dysponuje ona historycznymi danymi o udzielonych kredytobiorcom pożyczkach obejmującymi informację o spłatach i statusie poszczególnych kredytów (rys. 10.1).



Rysunek 10.1. Najwięcej pożyczek jest na bieżąco spłacanych (status *Current*) lub zostało spłaconych w terminie (status *Fully Paid*). Pozostałe statusy świadczą o problemach ze zwrotem kredytów, co oznacza, że firma ma problem z odzyskaniem ponad 800 mln USD pożyczonych klientom

Naszym zadaniem jest zbudowanie modelu, który oceni kredyt jako dobry lub zły. Taki model będzie używany nie tylko do oceny udzielanych kredytów, ale również do analiz typu *Co by było, gdyby?*. Firma planuje wykorzystać nasz

model do sprawdzania, jak zwiększenie oprocentowania pożyczek wpłynęłoby w różnych stanach na ich spłacalność. Budując model, powinniśmy pamiętać, że na serwerze SQL, który będzie używany do predykcji, nie będzie zainstalowanej usługi SQL Server Machine Learning.

Mamy więc zbudować model klasyfikatora, który będzie dzielił pożyczki na dobre (spłacane w terminie) i złe (pozostałe). Finalny model wdrożymy na serwerze SQL 2019 bez zainstalowanej usługi uczenia maszynowego.

10.2. Zrozumienie i przygotowanie danych

Przykładowe dane pochodzą ze strony https://www.lendingclub.com/info/download-data.action. Są to dane o pożyczkach udzielonych w latach 2017–2018. Proszę zwrócić uwagę, że w pierwszym wierszu plików znajduje się nagłówek *Notes offered by Prospectus* (https://www.lendingclub.com/info/prospectus.action). Po pobraniu, z każdego z ośmiu plików tekstowych należy usnąć pierwszy wiersz albo go pominąć podczas ładowania danych. Na tej samej stronie dostępny jest arkusz Excel z opisami poszczególnych zmiennych.

10.2.1. Import danych

W pierwszej kolejności załadujemy dane z pobranych plików tekstowych do bazy danych. Import danych przeprowadzimy dwuetapowo. Najpierw wgramy zawartość pliku do tabeli przejściowej, a następnie przeniesiemy zawartość tej tabeli do tabeli docelowej. Takie rozwiązanie nie tylko jest wydajne i skalowalne, ale również ułatwia ponowienie operacji przerwanych z dowolnego powodu, na przykład wystąpienia błędu. Ponieważ pierwszy etap wykonamy przy użyciu instrukcji języka R, a drugi SQL, pozwoli nam to

wykorzystać zalety obu języków.

W skrypcie importującym dane do tabeli przejściowej zdefiniujemy połączenie z serwerem SQL i wskażemy tabelę, do której będziemy ładować dane.

```
connection_string <- "Driver=SQL Server; Server=MS; Database=ML; Trusted_Connection=yes"
local <- RxLocalParallel()
rxSetComputeContext(local)

stage_table <- RxSqlServerData(table = "[LendingClub].[LoanStatsStaging]",
                                    connectionString = connection_string)
```

Do załadowania danych użyjemy funkcji `read_csv` z biblioteki `readr`. Określa ona typy poszczególnych zmiennych na podstawie tysiąca pierwszych wierszy, co może prowadzić do błędów.

```
dat <- readr::read_csv("LoanStats_2017Q1.csv", na = "NULL", skip = 1)
```

```
Parsed with column specification:
cols(
  .default = col_integer(),
  id = col_character(),
  member_id = col_character(),
  term = col_character(),
  int_rate = col_character(),
  installment = col_double(),
  grade = col_character(),
  sub_grade = col_character(),
  emp_title = col_character(),
  emp_length = col_character(),
  home_ownership = col_character(),
  annual_inc = col_double(),
  verification_status = col_character(),
  issue_d = col_character(),
  loan_status = col_character(),
  pymnt_plan = col_character(),
  url = col_character(),
  desc = col_character(),
  purpose = col_character(),
  title = col_character(),
  zip_code = col_character()
  # ... with 45 more columns
)
See spec(...) for full column specifications.
|=========================================================================================
==========================================| 100%    75 MB
Warning: 241 parsing failures.
…
See problems(...) for more details.
```

Listę wszystkich błędów, które wystąpiły podczas odczytu danych, zwraca funkcja `problems`:

```
readr::problems(dat)
```

```
# A tibble: 241 × 5
      row             col               expected actual                    file
    <int>           <chr>                  <chr>  <chr>                   <chr>
1    2290 settlement_amount no trailing characters    .25 'LoanStats_2017Q1.csv'
2    2737 settlement_amount no trailing characters     .3 'LoanStats_2017Q1.csv'
3    3244 settlement_amount no trailing characters    .24 'LoanStats_2017Q1.csv'
4    3381 settlement_amount no trailing characters    .78 'LoanStats_2017Q1.csv'
5    3492 settlement_amount no trailing characters    .08 'LoanStats_2017Q1.csv'
6    3536 settlement_amount no trailing characters     .2 'LoanStats_2017Q1.csv'
7    3647 settlement_amount no trailing characters    .11 'LoanStats_2017Q1.csv'
8    4675 settlement_amount no trailing characters    .87 'LoanStats_2017Q1.csv'
9    5081 settlement_amount no trailing characters    .75 'LoanStats_2017Q1.csv'
10   5356 settlement_amount no trailing characters    .43 'LoanStats_2017Q1.csv'
# ... with 231 more rows
```

Tym razem możemy zignorować ostrzeżenia. Sposobem na uniknięcie tych błędów może być zwiększenie wartości parametru `guess_max`, co spowoduje dobranie typu kolumn na podstawie większej próby.

Na tym etapie importu danych należy sprawdzić brakujące wartości. Pomoże nam w tym funkcja `df_status` z biblioteki `funModeling`. Zwraca ona liczbę zer, wartości *NA* oraz *INF* każdej zmiennej.

```
meta_loans <- funModeling::df_status(dat, print_results = TRUE)
```

```
variable         q_zeros p_zeros   q_na   p_na       q_inf p_inf       type       unique
id               0       0.00      0      0.00       0     0           character       1
member_id        0       0.00      0      0.00       0     0           character       1
loan_amnt        0       0.00      0      0.00       0     0           integer      1465
funded_amnt      0       0.00      0      0.00       0     0           integer      1465
funded_amnt_inv  0       0.00      0      0.00       0     0           integer      1473
term             0       0.00      0      0.00       0     0           character       2
…
```

Wartość brakująca może oznaczać brak danych, dane nieznane lub domyślne. Na

podstawie nazw zmiennych możemy przyjąć, że niektóre wartości brakujące należy zastąpić zerami. Dotyczy to zmiennych `mths_since_last_delinq`, `mths_since_last_record` i `mths_since_last_major_derog`. W wykonaniu tej zmiany pomoże nam funkcja `mutate_at` z biblioteki `dplyr`.

```
library(tidyverse)
na_to_zero_vars <-
  c("mths_since_last_delinq", "mths_since_last_record",
    "mths_since_last_major_derog")
dat <-
  dat %>%
  mutate_at(.vars = na_to_zero_vars, .funs = funs(replace(., is.na(.), 0)))
```

Sprawdźmy jeszcze liczbę i procent unikatowych wartości zmiennych.

```
meta_loans <-
  meta_loans %>%
  mutate(uniq_rat = unique / nrow(dat)) %>%
  select(variable, unique, uniq_rat) %>%
  mutate(unique = unique, uniq_rat = scales::percent(uniq_rat))
meta_loans
```

variable	unique	uniq_rat
id	1	0.0%
member_id	1	0.0%
loan_amnt	1465	1.5%
funded_amnt	1465	1.5%
funded_amnt_inv	1473	1.5%
term	2	0.0%
int_rate	36	0.0%
installment	12943	13.4%
grade	7	0.0%
sub_grade	35	0.0%
emp_title	33058	34.2%

Jak widać, dwie pierwsze zmienne, `id` i `member_id`, są puste i nie będą nam potrzebne.

Dane załadujemy do tabeli za pomocą funkcji `rxDataStep` z biblioteki

`RevoScaleR`. Jej parametr `varsToDrop` pozwala wskazać pominięte zmienne. Ja ograniczę się do nadpisania tabeli docelowej wszystkimi zmiennymi.

```
rxDataStep(inData = dat,
           outFile = stage_table,
           overwrite = TRUE)
Rows Read: 96779, Total Rows Processed: 96779
```

 Proszę samodzielnie przeanalizować dane i pominąć nieprzydatne zmienne, tj. zawierające tylko jedną wartość. Ta zmiana będzie wymagała odpowiedniego dostosowania wykonywanych w dalszej części rozdziału instrukcji i definicji obiektów bazy danych.

Operacje wczytywania pliku tekstowego, zamieniania brakujących wartości wybranych zmiennych zerami i załadowania wybranych zmiennych do tabeli tymczasowej należy powtórzyć dla pozostałych siedmiu plików tekstowych. Każdorazowo, po wczytaniu do tabeli przejściowej danych z pliku tekstowego, należy uruchomić procedurę przenoszącą te dane do tabeli docelowej. Zobaczmy, jak wyglądają dane załadowane z pierwszego pliku *LoanStats_2017Q1.csv*.

Odczytując przykładowe dane z tabeli przejściowej, zauważymy, że wymagają one dalszego przygotowania. Pokażę to na kilku wybranych zmiennych.

```
SELECT TOP 5 term, issue_d, earliest_cr_line, loan_amnt, loan_status, int_rate
FROM [LendingClub].[LoanStatsStaging];
```

term	issue_d	earliest_cr_line	loan_amnt	loan_status	int_rate
60 months	Mar-2017	Dec-1998	15400	Current	11.39%
36 months	Mar-2017	Dec-2003	4600	Current	11.39%
36 months	Mar-2017	May-1996	14000	Current	15.99%
36 months	Mar-2017	Jan-2004	5000	Current	25.49%
60 months	Mar-2017	Feb-1974	14000	Current	12.74%

Takie zmienne tekstowe jak `issue_d` i `earliest_cr_line` zawierają niestandardowo sformatowane daty. Podczas przenoszenia danych to tabeli docelowej należy skonwertować je na daty. W procedurze

`LendingClub.PerformETL` zmiana ta jest wykonana przy użyciu funkcji tekstowych `SUBSTRING` i `CHARINDEX`.

```
CAST(SUBSTRING(issue_d, 1,CHARINDEX('-', issue_d)-1) +
SUBSTRING(issue_d, CHARINDEX('-', issue_d)+1, 100) AS DATE) AS issue_d
```

Takie zmienne jak `int_rate` czy `revol_util` zawierają wartości procentowe zapisane ze znakiem `%`. W efekcie takiego zapisu są one traktowane jako zmienne tekstowe. Podczas przenoszenia danych to tabeli docelowej należy skonwertować je na liczby. W procedurze `LendingClub.PerformETL` zmiana ta jest przeprowadzana przy użyciu funkcji `REPLACE`.

```
REPLACE([revol_util], '%', '') AS [revol_util]
```

Inne zmienne również mogą wymagać przekształcenia. Na przykład zmienna `term` przechowuje liczbę miesięcznych rat. Akurat ta zmienna przyjmuje tylko dwa stany: `36 months` lub `60 months`, a więc jest zmienną dyskretną i jej zmiana na zmienną numeryczną jest niepotrzebna, ale proszę samodzielnie sprawdzić, czy w danych znajdują się inne zmienne tekstowe, które powinny być przekształcone na daty i liczby. Jeśli tak, proszę odpowiednio zmodyfikować procedurę `LendingClub.PerformETL` i nadać odpowiednie typy kolumnom tabeli `LendingClub.LoanStats`.

Sprawdźmy jeszcze rozkład wartości zmiennej wyjściowej `loan_status`.

```
SELECT loan_status, count(*) AS cnt
FROM [LendingClub].[LoanStats]
GROUP BY loan_status;
```

loan_status	cnt
Charged Off	23606
Current	773761
Default	17
Fully Paid	115738
In Grace Period	8332
Late (16-30 days)	3386
Late (31-120 days)	13981

Przyjmuje ona siedem stanów, z których dwa występują znacznie częściej niż

pozostałe. Każdy z pięciu stanów wskazujących na problemy ze spłatą kredytu występuje na tyle rzadko, że przewidzenie poszczególnych stanów będzie co najmniej kłopotliwe, jeśli nie niemożliwe. Rozwiązaniem może być zastąpienie siedmiu stanów zmiennej wyjściowej dwoma, z których jeden będzie reprezentował dobre, a drugi złe kredyty. W procedurze `LendingClub.PerformETL` zmiana ta jest przeprowadzana przy użyciu funkcji `CASE`.

```
CASE
  WHEN loan_status IN ('In Grace Period', 'Late (16-30 days)', 'Late (31-120 days)',
  'Default', 'Charged Off') THEN 1
  ELSE 0
END AS is_bad
```

Dodatkowo, procedura `LendingClub.PerformETL` oznacza ładowane do tabeli `LendingClub.LoanStats` wiersze jako treningowe lub testowe. Jeżeli zostanie wywołana z ustawionym parametrem `@is_train`, wiersze zostaną oznaczone jako należące do zbioru treningowego. W ten sposób należy załadować dane pochodzące ze wszystkich, z wyjątkiem ostatniego, plików tekstowych.

```
EXEC [LendingClub].[PerformETL] @is_train =1;
```

Dane z pliku *LoanStats_2018Q4.csv* oznaczyłem jako testowe. Procedura ich importu do tabeli tymczasowej jest taka sama jak pozostałych danych. Natomiast ich przeniesienie do tabeli docelowej wymaga wywołania procedury `LendingClub.PerformETL` z nieustawionym parametrem `@is_train`.

```
EXEC [LendingClub].[PerformETL]  @is_train=0;
```

10.2.2. Ocena danych

Ocenę danych zaczniemy od sprawdzenia liczby przykładów treningowych i testowych.

```
SELECT is_train, COUNT(*) AS cnt, 100.*count(*)/938821 as pct
FROM [LendingClub].[LoanStats]
GROUP BY is_train;
```

is_train	cnt	pct
0	128412	13.6780067
1	810409	86.3219932

Mamy do dyspozycji ponad 810 000 przykładów treningowych i prawie 130 000 testowych. Chociaż zbiór testowy zawiera tylko niecałe 14% wszystkich przykładów, wydaje się że 130 000 wystarczy aż nadto do wiarygodnej oceny jakości modeli.

Następnie sprawdzimy rozkład zmiennej wyjściowej.

```
SELECT is_bad, count(*) as cnt, 100.*count(*)/938821 as pct
FROM [LendingClub].[LoanStats]
GROUP BY is_bad;
```

is_bad	cnt	pct
0	889499	94.7463893
1	49322	5.2536106

Dane są silnie niezrównoważone. Prawie 95% przykładów dotyczy dobrych kredytów, tj. spłacanych na czas. Zobaczmy, czy rozkład zmiennej wyjściowej jest taki sam w zbiorze treningowym i testowym.

```
SELECT is_bad, count(*) as cnt, 100. * count(*) /
       (SELECT count(*) FROM [LendingClub].[LoanStats] WHERE is_train = 1) as pct
FROM [LendingClub].[LoanStats]
WHERE is_train = 1
GROUP BY is_bad
UNION ALL
SELECT is_bad, count(*) as cnt, 100.*count(*)/
       (SELECT count(*) FROM [LendingClub].[LoanStats] WHERE is_train = 0) as pct
FROM [LendingClub].[LoanStats]
WHERE is_train = 0
GROUP BY is_bad;
```

is_bad	cnt	pct
0	762293	94.06275103065
1	48116	5.93724896934
0	127206	99.06083543594
1	1206	0.93916456405

Okazuje się, że w zbiorze treningowym prawie 6% kredytów było złych, a w zbiorze testowym odsetek złych kredytów nie przekracza jednego procenta. Oznacza to, że różnice między tymi zbiorami są statystycznie istotne. To źle, bo statystyki zbioru testowego powinny być zbliżone do statystyk zbioru treningowego. **Nie ma sensu uczyć modelu na danych różniących się statystycznie od danych, które będą używane do predykcji**. Okazuje się, że pomysł z użyciem najnowszych danych do testów był błędem. Błąd ten jest na tyle powszechny, że wymaga krótkiego wyjaśnienia.

Pomysł użycia najnowszych danych jako testowych wywodzi się z analizy szeregów czasowych. Analiza szeregów czasowych, w szczególności prognozowanie, polega na szacowaniu przyszłych wartości serii na podstawie historycznych wartości zmiennych, a dokładniej tego, w jaki sposób się one zmieniały. Wyjaśnijmy to na przykładzie.

Najprostszym modelem prognozującym jest autoregresja liniowa. Autoregresja różni się od regresji tym, że przyszłe wartości atrybutu są szacowane na podstawie określonej liczby jego wcześniejszych wartości. Ogólny wzór

opisujący autoregresję liniową wygląda więc następująco:

$$y_{t(0)} = a_1 x_{t(-1)} + a_2 x_{t(-2)} + a_3 x_{t(-3)} + ... + a_n x_{t(-n)} + \varepsilon$$

gdzie *t(–n)* jest znacznikiem czasu, a a_n reprezentuje znalezione przez algorytm autoregresji liniowej współczynniki wpływu wcześniejszych wartości zmiennej na jej przyszłą wartość.

Działanie wszystkich, również tych znacznie bardziej skomplikowanych modeli prognozujących polega na znalezieniu wystarczająco dobrego przybliżenia zależnej od czasu funkcji *f*, która na podstawie przeszłych wartości zmiennej (lub wielu zmiennych) zwróci jej przyszłe wartości, czyli prognozy:

$$y^*_t = f(t, y_{t-1}, ..., y_t - p, y^*_{t-1}, ..., y^*_{t-1}, \varepsilon_t)$$

gdzie *t* oznacza czas, y_{t-1} – wartości zmiennej, y^*_{t-1} – jej prognozy, a ε_t – składnik losowy. Oznacza to, że w analizie szeregów czasowych przykłady nie mogą być losowo dzielone na testowe i treningowe, bo w wyniku takiego podziału uzyskalibyśmy niekompletne szeregi, co zniekształciłoby dane treningowe. Dlatego jakość predykcji ocenia się, ograniczając długość danych treningowych, czyli usuwając z nich *x* ostatnich przykładów. Usunięte w ten sposób przykłady służą jako dane testowe.

W przeciwieństwie do prognozowania, w regresji i klasyfikacji nie uwzględnia się kolejności zdarzeń. W ich przypadku dane testowe powinny być jak najbardziej podobne do treningowych, co najłatwiej uzyskać, losowo dzieląc przykłady na treningowe i testowe.

Drugim powodem, dla którego wybór danych z czwartego kwartału 2018 r. jako testowych był błędem, była ich liczba. Im więcej danych treningowych, tym lepiej. Dlatego do minimum ogranicza się liczbę przykładów testowych. W celach demonstracyjnych zdecydowałem się dokończyć eksperyment, nie zmieniając błędnego podziału danych na treningowe i testowe. Podpowiem

jedynie, że prawidłowo można podzielić dane, wykonując poniższą instrukcję.

```
WITH cte AS (
       SELECT is_train, ABS(CHECKSUM(NEWID())) % 100 AS rnd
         FROM [LendingClub].[LoanStats] )
UPDATE cte
SET is_train = CASE WHEN rnd<6 THEN 0 ELSE 1 END;
```

W rezultacie 94% przykładów trafi do zbioru treningowego.

```
SELECT is_train, 100.*count(*)/
  (SELECT count(*) FROM [LendingClub].[LoanStats]) as pct
FROM [LendingClub].[LoanStats]
GROUP BY is_train;
```

```
is_train        pct
0               4.00076955212
1               95.99923044787
```

Co najważniejsze, taki podział da nam podobne do siebie zbiory treningowe i testowy. Na przykład, rozkład zmiennej wyjściowej w obu zbiorach jest teraz prawie identyczny.

```
SELECT is_bad, 100. * count(*) /
       (SELECT count(*) FROM [LendingClub].[LoanStats] WHERE is_train = 1) as pct
FROM [LendingClub].[LoanStats]
WHERE is_train = 1
GROUP BY is_bad
UNION ALL
SELECT is_bad, 100. * count(*) /
       (SELECT count(*) FROM [LendingClub].[LoanStats] WHERE is_train = 0) as pct
FROM [LendingClub].[LoanStats]
WHERE is_train = 0
GROUP BY is_bad;
```

```
is_bad   pct
0        94.45763687337
1        5.54236312662
0        94.33403681557
1        5.66596318442
```

Z poprzednich rozdziałów wiemy, że do zrozumienia danych pomocne są statystyki opisowe i wykresy. Do wyliczenia podstawowych statystyk użyjemy opisanej w rozdziale 4 procedury `DQ.DataProfiling`.

```
DECLARE @RC int
DECLARE @DatabaseName nvarchar(50)
DECLARE @SchemaName nvarchar(50)
DECLARE @TableView char(1)
DECLARE @Debug bit

EXECUTE @RC = [DQ].[DataProfiling]
   @DatabaseName = 'ML'
   ,@SchemaName = 'LendingClub'
   ,@TableName ='LoanStats'
   ,@TableView ='T'
   ,@Debug = 0;
```

Ja ograniczę się do sprawdzenia brakujących danych. W tym celu odczytam wyniki jednego z testów (testu *Emptyness*) wykonanych przez powyższą procedurę.

```
SELECT [ColumnName], 100. * p.Counts /
  (SELECT COUNT(*) FROM [LendingClub].[LoanStats]) AS pct
FROM [DQ].[DataProfileTables] t
JOIN [DQ].[DataProfileColumns] c on t.TableId = c.TableId
JOIN [DQ].[DataProfiles] p on c.ColumnId = p.ColumnId
JOIN [DQ].[DataProfileStats] s on s.StatId = p.ProfileStatId
WHERE TableName = 'LoanStats' and Description = 'Emptyness'
ORDER BY p.Counts DESC;
```

ColumnName	pct
verification_status_joint	88.66344063458
annual_inc_joint	88.16941674717
dti_joint	88.13298807759
mths_since_recent_bc_dlq	78.59698494175
mths_since_recent_revol_delinq	68.91526712759
il_util	15.19118127949
next_pymnt_d	14.84244600408
mths_since_recent_inq	12.06385455800
emp_title	9.26640967767
num_tl_120dpd_2m	3.69165155018
mths_since_rcnt_il	3.35910679458
mo_sin_old_il_acct	3.35910679458
bc_util	1.35819288234
…	

Jak widać, brakuje wartości wielu zmiennych. W przypadku pięciu zmiennych procent brakujących wartości jest tak duży, że należy je usunąć z danych.

Państwa zadaniem będzie uzupełnienie brakujących wartości pozostałych zmiennych. Proszę również wykryć i usunąć odstające wartości. W ich znalezieniu pomocny będzie dołączony do rozdziału 3 raport Power BI *Variables.pbix*.

Wybrane i odpowiednio przygotowane zmienne zostaną udostępnione przez widok `LendingClub.vLoanStats`. Na tym etapie należy też zrównoważyć dane. Ponieważ dysponujemy dużą ilością danych, zrównoważymy je, usuwając część przykładów większościowych.

Tworząc widok `LendingClub.vLoanStats`, sprowadziłem przygotowanie danych do wyeliminowania pięciu zmiennych z ponad 50% brakujących wartości oraz do wybrania 10% przykładów dobrych kredytów. W tym celu połączyłem dwa

prawie identyczne zapytania klauzulą `UNION ALL`. Pierwsze zapytanie zwraca wszystkie przykłady złych kredytów, a drugie – wybrane przykłady dobrych kredytów.

```
CREATE VIEW [LendingClub].[vLoanStats] AS
SELECT [loan_amnt]
       ,[funded_amnt]
              ...
FROM [LendingClub].[LoanStats]
WHERE is_bad = 1
UNION ALL
SELECT [loan_amnt]
,[funded_amnt]
              ...
FROM [LendingClub].[LoanStats]
WHERE is_bad = 0 AND (ABS(CAST((BINARY_CHECKSUM(NEWID())) as int)) % 100) < 10
```

Proszę zwrócić uwagę, że tak przygotowane dane nadal są niezrównoważone – prawie 65% przykładów to przykłady dobrych kredytów.

```
SELECT is_bad, COUNT(*) as cnt, 100.*COUNT(*)/
       (SELECT COUNT(*) FROM [LendingClub].[vLoanStats]) AS pct
FROM [LendingClub].[vLoanStats]
GROUP BY is_bad;
```

is_bad	cnt	pct
1	49322	35.70358251956
0	88794	64.27687251616



Modyfikując widok `LendingClub.vLoanStats`, proszę ograniczyć liczbę zmiennych, wyeliminować brakujące i odstające wartości pozostałych zmiennych oraz dokładniej zrównoważyć dane. Proszę również zastosować inne, poznane w rozdziale 5 techniki przekształcania danych, takie jak kodowanie, generalizacja i skalowanie.

Po ocenie i przygotowaniu zmiennych możemy sprawdzić korelacje występujące między zmiennymi. Pozwoli to ograniczyć liczbę zmiennych wejściowych modelu do najsilniej skorelowanych ze zmienną wyjściową. Korelację można ocenić, wyliczając odpowiednie współczynniki korelacji lub przedstawiając

zmienne na właściwych wykresach.

Ponieważ zmienna wyjściowa jest kategoryczna, korelację między nią a numerycznymi zmiennymi wejściowymi można sprawdzić, porównując rozkład wartości zmiennej numerycznej dla obu stanów zmiennej kategorycznej. Tego typu wykres zwraca funkcja `ggplot` wywołana z parametrami `geom_density` i `group`. Osobne wykresy dla wszystkich zmiennych numerycznych otrzymamy, dodając parametr `facet_wrap`. Procedura `LendingClub.PlotDistribution` tworzy omawiany wykres.

```
CREATE OR ALTER PROCEDURE [LendingClub].[PlotDistribution] (@input_query NVARCHAR(MAX))
AS BEGIN
SET NOCOUNT ON;
EXECUTE sp_execute_external_script
@language = N'R',
@script = N'
library("reshape2")
library("ggplot2")
print(getwd())
#filtering numeric columns
numeric_cols <- sapply(InputDataSet, is.numeric)

#turn the data into long format (key->value)
loans.lng <- melt(InputDataSet[,numeric_cols], id="is_bad")

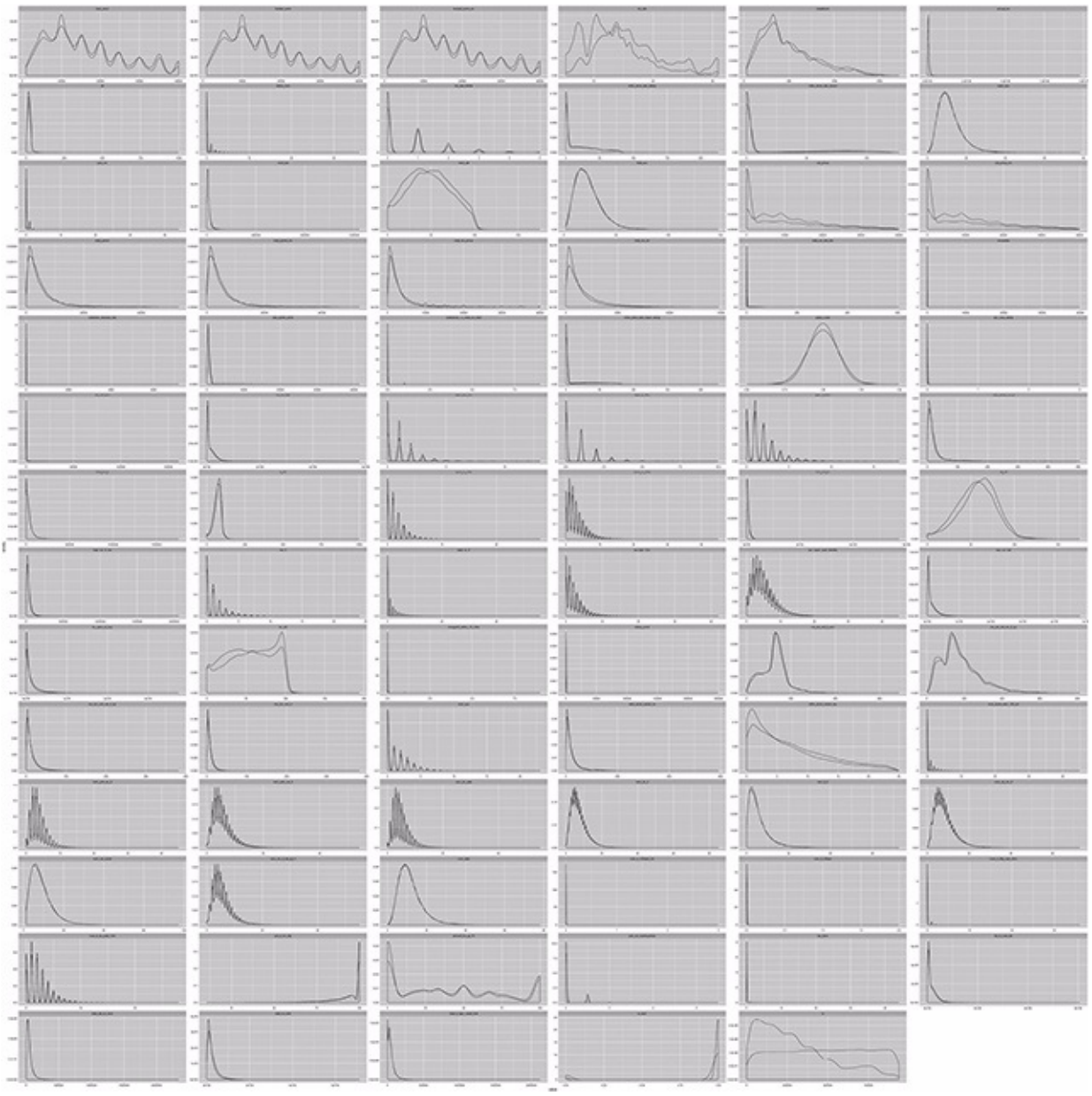
#plot the distribution for is_bad={0/1} for each numeric column
ggplot(aes(x=value, group=is_bad, colour=factor(is_bad)), data=loans.lng) + geom_density()
+ facet_wrap(~variable, ncol = 6, scales="free")
ggsave(''loans.jpg'',height=30, width = 50, dpi=300,limitsize = FALSE)
while (1 == 1) {}
',
@input_data_1 = @input_query
END
```

Proszę zwrócić uwagę, że po wygenerowaniu wykresu procedura wykonuje nieskończoną pętlę. W ten sposób obszedłem problem występujący w wersji CTP 2 serwera SQL Server 2019 polegający na niemożliwości nadania

uprawnień do folderów użytkownikowi, z którego uprawnieniami są wykonywane zewnętrzne skrypty. Dlatego wykres jest zapisany w automatycznie utworzonym folderze tymczasowym. W tym przypadku był to folder *C:\SQL-MSSQLSERVER-ExtensibilityData\Appcontainer1\7527EE33-46F2-4A58-947A-CD7C222D8E42*. Takie foldery są automatycznie usuwane po zakończeniu wykonywania zewnętrznych procedur. Pętla nieskończona opóźnia ten moment, pozwalając uzyskać dostęp do powyższego folderu na przykład z poziomu Eksploratora Windows, skopiować plik z wykresem, po czym z poziomu SSMS można przerwać wykonywanie procedury.

Wywołując procedurę `LendingClub.PlotDistribution`, należy wskazać oceniane dane – w naszym przypadku będą to dane odczytane z widoku `LendingClub.vLoanStats` (rys. 10.2).

```
EXEC [LendingClub].[PlotDistribution]
       @input_query = N'SELECT * FROM [LendingClub].[vLoanStats]'
```



Rysunek 10.2. Im większe różnice między widocznymi na wykresach liniami, tym silniejsza korelacja między zmiennymi

Do oceny korelacji między zmiennymi kategorycznymi użyłem testu *Chi-kwadrat*. Test przeprowadziłem, używając poznanego w rozdziale 4 zapytania. Im większa wartość *Chi-kwadrat*, tym silniejsza korelacja między zmiennymi. Na przykład zmienna `grade` jest silnie skorelowana ze zmienną wyjściową (*Chi-kwadrat* wynosi 14 843 przy sześciu stopniach swobody), a zmienna `application_type` nie jest z nią skorelowana (*Chi-kwadrat* wynosi 1,5 przy

jednym stopniu swobody).

```
WITH
ObservedCombination_CTE AS
(
SELECT is_bad AS OnRows,
    grade OnCols,
 COUNT(*) AS ObservedCombination
FROM [LendingClub].[vLoanStats]
GROUP BY is_bad, grade
),
ExpectedCombination_CTE AS
(
SELECT OnRows, OnCols, ObservedCombination
 ,SUM(ObservedCombination) OVER (PARTITION BY OnRows) AS ObservedOnRows
 ,SUM(ObservedCombination) OVER (PARTITION BY OnCols) AS ObservedOnCols
 ,SUM(ObservedCombination) OVER () AS ObservedTotal
 ,CAST(ROUND(SUM(1.0 * ObservedCombination) OVER (PARTITION BY OnRows)
  * SUM(1.0 * ObservedCombination) OVER (PARTITION BY OnCols)
  / SUM(1.0 * ObservedCombination) OVER (), 0) AS INT) AS ExpectedCombination
FROM ObservedCombination_CTE
)
SELECT (COUNT(DISTINCT OnRows) - 1) * (COUNT(DISTINCT OnCols) - 1) AS DegreesOfFreedom,
       SUM(SQUARE(ObservedCombination - ExpectedCombination)
 / ExpectedCombination) AS ChiSquared
FROM ExpectedCombination_CTE;
DegreesOfFreedom ChiSquared
6                        14843.6192583611
```

Państwa zadaniem jest wybór zmiennych wejściowych modelu. Proszę założyć, że użyjemy algorytmu, który odpowiednio skonfigurowany, jest względnie odporny na obecność mniej przydatnych zmiennych wejściowych.

10.3. Modelowanie

Jednym z założeń projektu był wymóg udostępniania modelu na serwerze SQL bez zainstalowanej usługi SQL Server Machine Learning. To ogranicza listę

możliwych do użycia algorytmów do wspierających natywne zapytania predykcyjne. W czasie powstawania książki lista ta zawierała regresję liniową i logistyczną, drzewa decyzyjne, las losowy i wzmocnione drzewa decyzyjne zaimplementowane w bibliotekach `RevoScale`. Biorąc pod uwagę dostępne dane i postawione zadanie, najlepszymi kandydatami wydają się dwa ostatnie algorytmy. Ja zdecydowałem się zbudować model lasu losowego.

Zwracająca model lasu losowego funkcja `rxDForest` biblioteki `RevoScaleR` wymaga wskazania zbioru treningowego i formuły uczącej. Zbiór treningowy został przygotowany i udostępniony przez widok `LendingClub.vLoanStats`. Pozostało nam wskazać wybrane zmienne wejściowe.



Jeżeli na liście wybranych przez Państwa zmiennych wejściowych znalazły się zmienne `delinq_2yrs` lub `mths_since_last_delinq`, proszę zapoznać się z ich opisem. Pierwsza zawiera liczbę 30-dniowych zaległości w spłacie kredytu, druga liczbę miesięcy, które upłynęły od ostatniej zaległości. Kiedy nasz model będzie oceniał nowe kredyty, tego typu informacje nie będą dostępne. Zmienne z przyszłości należy usunąć z danych treningowych.

Postawiona przez mnie hipoteza wygląda następująco:

```
formula <- 'is_bad ~ grade  + out_prncp_inv + int_rate + policy_code + installment  +
open_acc_6m + all_util + revol_util + total_rec_prncp + bc_util  + home_ownership'
```

Funkcja `rxDForest` pozwala też określić hiperparametry budowanego modelu. Optymalizacja modelu jest przedmiotem następnego podrozdziału, na razie użyję przypadkowych wartości. Las losowy będzie liczył 15 drzew, maksymalna głębokość drzewa to 8, minimalna wymagana do utworzenia węzła liczba przykładów została określona na 350, a liczbę uwzględnianych przy każdym podziale zmiennych ograniczyłem do 5.

Nauczony w ten sposób model zawiera m.in. użytą formułę uczącą oraz macierz błędów. Oznacza to, że błąd treningowy można odczytać bezpośrednio z modelu. Do oceny modeli zdecydowałem się użyć *F*-miary. Zadanie uczenia modeli oraz zapisywania modelu i informacji na jego temat realizuje procedura

LendingClub.BuildModel.

```
CREATE OR ALTER PROCEDURE [LendingClub].[BuildModel] (@input_query NVARCHAR(MAX), @name
NVARCHAR(100)) AS
BEGIN
DECLARE @model VARBINARY(MAX);
DECLARE @formula NVARCHAR(MAX);
DECLARE @fscore DECIMAL (5,4);

EXECUTE sp_execute_external_script
 @language = N'R'
 ,@script = N'
 formula <- ''is_bad ~ grade  + out_prncp_inv + int_rate + policy_code + installment  +
open_acc_6m + all_util + revol_util + total_rec_prncp + bc_util  + home_ownership''
 InputDataSet$is_bad <- as.factor(InputDataSet$is_bad)
 model <- RevoScaleR::rxDForest(formula =  formula,
                                          data = InputDataSet,
                                          minSplit = 350,
                                          nTree = 15,
                                  mTry = 5,
                                          maxDepth = 8)
 formula <- model$formula
 confusion <- model$confusion
 tn <- confusion[1, 1]
 fp <- confusion[1, 2]
 fn <- confusion[2, 1]
 tp <- confusion[2, 2]
 accuracy <- (tp + tn) / (tp + fn + fp + tn)
 precision <- tp / (tp + fp)
 recall <- tp / (tp + fn)
 fscore <- 2 * (precision * recall) / (precision + recall)

 model <- rxSerializeModel(model, realtimeScoringOnly = FALSE)'
 ,@input_data_1 = @input_query
 ,@params = N'@formula nvarchar(max) OUTPUT, @model varbinary(max) OUTPUT, @fscore
decimal(5,4) OUTPUT'
 ,@formula = @formula OUTPUT
 ,@model = @model OUTPUT
 ,@fscore = @fscore OUTPUT

 INSERT [LendingClub].[Models] ([Name], [Model], [Train F-score], [TrainSet], [Formula],
[nTree], [mTry], [maxDepth], [minSplit])
 VALUES(@name, @model, @fscore, @input_query, @formula, 15, 5, 8, 350) ;

END
```

Wywołując ją, należy podać nazwę uczonego modelu oraz wskazać dane treningowe. Na przykład, żeby nauczyć model na nieprzygotowanych, odczytanych z tabeli danych, należy wykonać poniższą instrukcję. Proszę zwrócić uwagę, że wybieram kolumny użyte przeze mnie jako zmienne wejściowe. W ten sposób unikam odczytywania niepotrzebnych danych do pamięci i skracam czas wykonania procedury.

```
EXEC [LendingClub].[BuildModel]
       @name = N'Decision Forest: All data',
       @input_query = N'SELECT id,[is_bad],[grade],[int_rate],[out_prncp_inv],
[policy_code],
  [installment],[open_acc_6m],[all_util],[revol_util],[total_rec_prncp],[bc_util],
  [percent_bc_gt_75],[home_ownership]
   FROM [LendingClub].[LoanStats]
  WHERE [is_train] = 1';
```

Nauczenie modelu na przygotowanych danych wymaga tylko zmiany parametru `@input_query`.

```
EXEC [LendingClub].[BuildModel]
       @name = N'Decision Forest: Balanced data',
       @input_query = N'SELECT * FROM [LendingClub].[vLoanStats] WHERE [is_train] = 1'
```

Modele oraz ich opisy zostały zapisane w tabeli `LendingClub.Models`.

```
SELECT [Name],nTree,mTry,maxDepth,minSplit,[Train F-score]
FROM [ML].[LendingClub].[Models]
WHERE [Name] LIKE 'Decision Forest:%';
```

Name	nTree	mTry	maxDepth	minSplit	Train F-score
Decision Forest: All data	15	5	8	350	0.6312
Decision Forest: Balanced data	15	5	8	350	0.6885

Model nauczony na przygotowanych danych wydaje się nieznacznie lepszy, bo jego błąd empiryczny jest mniejszy. Nas jednak interesuje błąd rzeczywisty. Żeby go sprawdzić, musimy ocenić model przy użyciu danych testowych.

W procedurze `LendingClub.BuildModel` modele zostały zserializowane z nieustawioną opcją `realtimeScoringOnly`. Ułatwia to ich ocenę za pomocą skryptów języka R. Zadanie to realizuje procedura `LendingClub.ScoreLoans`:

- odczytuje z tabeli `LendingClub.Models` model o podanej nazwie;
- kasuje zawartość tabeli `LendingClub.LoanStatsPredictions`;
- zapisuje czas rozpoczęcia oceny modelu w tabeli `LendingClub.RunTimeStats`;
- deserializuje pobrany model, wysyła do niego zapytania predykcyjne i zapisuje ich wyniki uzupełnione o rzeczywistą wartość zmiennej `is_bad` do tabeli `LendingClub.LoanStatsPredictions`;
- odnotowuje czas zakończenia oceny modelu w tabeli `LendingClub.RunTimeStats`;
- wylicza na podstawie danych z tabeli `LendingClub.LoanStatsPredictions` metryki modelu i zapisuje *F*-miarę w tabeli `LendingClub.Models`.

Poniżej zostały pokazane wywołania omawianej procedury w celu oceny obu utworzonych modeli:

```
EXEC [LendingClub].[ScoreLoans]
       @input_query = 'SELECT * FROM [LendingClub].[LoanStats] WHERE [is_train] = 0',
       @name = 'Decision Forest: All data';
EXEC [LendingClub].[ScoreLoans]
       @input_query = 'SELECT * FROM [LendingClub].[vLoanStats] WHERE [is_train] = 0',
       @name = 'Decision Forest: Balanced data';
```

Porównajmy empiryczne i rzeczywiste błędy obu modeli zmierzone *F*-miarą.

```
SELECT [Name],[Train F-score],[Test F-score]
FROM [LendingClub].[Models]
WHERE [Name] LIKE 'Decision Forest:%';
```

Name	Train F-score	Test F-score
Decision Forest: All data	0.6312	0.1053
Decision Forest: Balanced data	0.6885	0.2670

Różnica między błędami rzeczywistymi jest znaczna. **Efektem przygotowania danych jest właśnie zmniejszenie błędu rzeczywistego modeli**. Niestety, błąd rzeczywisty obu modeli jest zbyt duży, żeby któryś z nich mógł zostać wdrożony do użycia. Musimy znaleźć model o optymalnej architekturze, tj. model, który nauczony na naszych danych będzie miał najmniejszy błąd rzeczywisty.

10.4. Optymalizacja

Twierdzenie o braku darmowego lunchu mówi, że nie ma najlepszego algorytmu uczenia maszynowego i że nie istnieje idealna kombinacja hiperparametrów modelu. Znajomość zasad działania poszczególnych algorytmów uczenia maszynowego pozwala dobrać algorytm do rozwiązywanego problemu i posiadanych danych, ale jak znaleźć właściwą kombinację hiperparametrów?

To, że wybrałem algorytm lasów losowych, nie oznacza, że inny z dostępnych algorytmów nie okazałby się w tym przypadku lepszy. Proszę sprawdzić, jak z postawionym zadaniem poradzi sobie algorytm wzmocnionych drzew decyzyjnych.

Najprostsza metoda optymalizacji hiperparametrów polega na sprawdzeniu wszystkich możliwych kombinacji ich wartości. Po prostu buduje się *n* modeli kandydujących, z których każdy ma inną architekturę, uczy się je na tych samych danych, porównuje ich jakość i wybiera najlepszy model. Tak działa poznana w rozdziale 3 metoda pełnego przeszukiwania siatki. W praktyce jest ona jednak rzadko stosowana, bo wymaga nauczenia dziesiątków, jeśli nie setek tysięcy modeli kandydujących, z których wiele ma bardzo podobną jakość.

W tym przypadku postanowiłem znaleźć optymalną kombinację czterech parametrów funkcji `rxDForest`: `nTree`, `minSplit`, `maxDepth` i `mTry`. Założyłem, że znajdę ją w zakresie:

- liczby drzew (parametr `nTree`) między 30 a 68, dla wartości zwiększanych o 2;
- minimalnej liczby przykładów wymaganej do podzielania danych (parametr `minSplit`) między 100 a 480, dla wartości zwiększanych o 10;
- maksymalnej głębokości drzewa (parametr `maxDepth`) między 2 a 20, dla wartości zwiększanych o 2;
- liczby losowo wybranych zmiennych rozpatrywanych przy każdym podziale (parametr `mTry`) między 1 a 10, dla wartości zwiększanych o 1.

Do wygenerowania wartości poszczególnych hiperparametrów użyłem rekurencyjnych podzapytań CTE. Otrzymane w ich wyniku dane złączyłem krzyżowo ze sobą, otrzymując w ten sposób iloczyn kartezjański łączonych zbiorów, czyli wszystkie możliwe kombinacje wartości hiperparametrów.

```
DROP TABLE IF EXISTS #hiperparameters;
WITH Sequences20 (nTree,minSplit) AS
       (SELECT 30 AS nTree, 100 AS minSplit
         UNION ALL
     SELECT nTree + 2, minSplit + 20
          FROM Sequences20
          WHERE nTree < 67),
Sequences10 (maxDepth,mTry) AS
       (SELECT 2 AS maxDepth, 1 AS mTry
                UNION ALL
     SELECT maxDepth + 2,  mTry  + 1
          FROM Sequences10
          WHERE maxDepth < 20)
SELECT S1.nTree, S2.minSplit, S3.maxDepth, S4.mTry
INTO #hiperparameters
FROM Sequences20 AS S1
CROSS JOIN Sequences20 AS S2
CROSS JOIN Sequences10 AS S3
CROSS JOIN Sequences10 AS S4;
```

Mimo ograniczenia zakresu sprawdzanych wartości poszczególnych hiperparametrów oraz dzięki pominięciu wielu z nich (np. wartości hiperparametru `minSplit` zwiększałem o 10), otrzymałem 40 000 kombinacji hiperparametrów.

Nauczenie 40 000 modeli kandydujących zajęłoby ponad miesiąc. Dlatego zdecydowałem się na zastosowanie **metody losowego przeszukiwania** (ang. *random serch*). Metoda ta polega na sprawdzeniu losowo wybranych kombinacji wartości hiperparametrów. Z reguły sprawdzenie 5% możliwych kombinacji daje wyniki zbliżone do przeszukania pełnej siatki.

Żeby losowo wybrać 5% wierszy, posortowałem wyniki zapytania według wyników funkcji `NEWID`. Wyjątkowość tej funkcji polega na tym, że jest ona wywoływana dla każdego wiersza wyniku, a nie raz dla całego zapytania. Tak wybrane wiersze zapisałem do kursora.

```
DECLARE hiperparameters CURSOR FOR
SELECT TOP 5 PERCENT [nTree], [mTry], [maxDepth], [minSplit]
FROM #hiperparameters
ORDER BY NEWID();
```

Chociaż język SQL został stworzony w celu operowania na zbiorach, pozwala on wykonywać instrukcje dla poszczególnych wartości zmiennych. Właśnie do tego służą kursory. Poniższa pętla wywołuje procedurę `LendingClub.BuildModels` dwa tysiące razy, za każdym razem z innymi wartościami parametrów `@n_tree`, `@m_try`, `@max_depth` i `@minSplit`. Nazwy poszczególnych modeli kandydujących są uzupełniane o numer wywołania procedury. Żeby maksymalnie skrócić czas uczenia modeli, z widoku `LendingClub.vLoanStats` są odczytywane tylko wymagane zmienne. Nauczenie 2000 modeli zajęło mi dwie doby.

```
DECLARE @n_tree INT, @m_try INT, @max_depthINT, @minSplit INT;
DECLARE @name NVARCHAR (100);

OPEN hiperparameters

FETCH NEXT FROM hiperparameters
INTO @n_tree, @m_try, @max_depth, @minSplit;

WHILE @@FETCH_STATUS = 0
BEGIN
        SET @name = CONCAT('Decision Forest V',@i);
        PRINT '-------------------------------------'
        PRINT 'Training model ' + @name
        PRINT '-------------------------------------'
        EXEC [LendingClub].[BuildModels]
        @name = @name,
        @input_query = N'SELECT id,[is_bad],[grade],[int_rate],[out_prncp_inv],[policy_code]
           ,[installment],[open_acc_6m],[all_util],[revol_util],[total_rec_prncp],[bc_util]
           ,[percent_bc_gt_75],[home_ownership]
           FROM [LendingClub].[vLoanStats]
           WHERE [is_train] = 1',
        @n_tree = @n_tree,
        @m_try = @m_try,
        @max_depth = @max_depth,
        @minSplit = @minSplit;

        FETCH NEXT FROM hiperparameters
        INTO @n_tree, @m_try, @max_depth, @minSplit;
        SET @i+=1;

END
```

Przyjrzyjmy się dziesięciu najlepszym i dziesięciu najgorszym modelom, tj. modelom o najwyższej i najniższej *F*-mierze.

```
SELECT [Name],[nTree],[mTry],[maxDepth],[minSplit],[Train F-score]
FROM [LendingClub].[Models]
WHERE [Train F-score] in
(SELECT TOP 10 [Train F-score] FROM [LendingClub].[Models] ORDER BY [Train F-score] DESC)
OR [Train F-score] in
(SELECT TOP 10 [Train F-score] FROM [LendingClub].[Models] ORDER BY [Train F-score])
ORDER BY [Train F-score] DESC;
```

Name	nTree	mTry	maxDepth	minSplit	Train F-score
Decision Forest V1915	44	9	18	120	0.7199
Decision Forest V1918	40	7	20	100	0.7188
Decision Forest V1560	34	8	18	120	0.7184
Decision Forest V1951	48	10	18	160	0.7181
Decision Forest V1596	52	10	20	180	0.7167
Decision Forest V1722	56	9	18	200	0.7167
Decision Forest V1566	40	8	18	100	0.7166
Decision Forest V1793	30	9	20	160	0.7164
Decision Forest V1338	62	8	18	120	0.7157
Decision Forest V1292	42	9	18	180	0.7149
Decision Forest V297	46	1	2	400	0.2978
Decision Forest V60	68	1	2	140	0.2876
Decision Forest V326	60	1	2	260	0.2766
Decision Forest V341	56	1	2	100	0.2726
Decision Forest V244	44	1	2	200	0.2422
Decision Forest V911	52	1	2	160	0.2041
Decision Forest V2094	64	1	2	220	0.1354
Decision Forest V283	66	1	2	220	0.1292
Decision Forest V1796	54	1	2	320	0.0879
Decision Forest V1853	38	1	2	400	0.0787

Modele lasów losowych, w których głębokość drzewa została ograniczona do 2, a liczba analizowanych przy podziałach zmiennych zmniejszona do 1, mają zbyt małą pojemność, żeby nauczyć się klasyfikować dane. Ich błąd systematyczny jest ogromny, niezależnie od liczby drzew w lesie i minimalnej, wymaganej do utworzenia węzła liczby przykładów. Widać, że źle dobraliśmy dolny zakres wartości parametrów `@m_try` i `@max_depth` – powinny być większe.

Zgodnie z oczekiwaniami, wraz ze zwiększaniem pojemności modeli, tj. zwiększaniem głębokości drzew i liczby uwzględnianych przy każdym podziale

zmiennych, ich błąd systematyczny maleje. Liczba drzew modeli cechujących się najmniejszym błędem empirycznym oscyluje wokół 50, a minimalna liczba wymaganych do utworzenia węzła przykładów mieści się w dolnej połowie zakresu.

Warto zwrócić uwagę na niewielkie różnice między modelami. Pomijając zbyt proste modele, błąd empiryczny mierzony *F*-miarą ponad 400 modeli wynosi mniej niż 30%.

```
SELECT COUNT(*)
FROM [LendingClub].[Models]
WHERE [Train F-score] >=0.7
```

```
414
```

Do oceny wszystkich modeli kandydujących użyjemy znanej nam już procedury `LendingClub.ScoreLoans`. Tym razem wywołamy ją kursorem zwracającym nazwy tych modeli.

```
DECLARE models CURSOR FOR
SELECT [Name]
FROM [LendingClub].[Models]
WHERE [Name] like '%V%';
DECLARE @name NVARCHAR(100);
OPEN models

FETCH NEXT FROM models
INTO @name

WHILE @@FETCH_STATUS = 0
BEGIN
        EXEC [LendingClub].[ScoreLoans]
        @input_query = 'SELECT * FROM [LendingClub].[vLoanStats] WHERE [is_train] = 0',
        @name = @name

        FETCH NEXT FROM models
        INTO @name ;

END
CLOSE models;
```

Procedura `LendingClub.ScoreLoans` zapisuje wyliczoną *F*-miarę, ale wyświetla cztery metryki ocenianego modelu. Patrząc na trafność, precyzję i czułość poszczególnych modeli, można dojść do wniosku, że wybór *F*-miary jako kryterium oceny modeli był błędem. Jeśli zadaniem modelu ma być wykrywanie złych kredytów, należałoby go optymalizować pod kontem czułości.

```
Accuracy    Precision    Recall      FScore
0.8476964   0.1875428    0.2271973   0.2054743
…
Accuracy    Precision    Recall      FScore
0.91581197  0.77272727   0.02819237  0.05440000
…
Accuracy    Precision    Recall      FScore
0.8558032   0.2669018    0.3764511   0.3123495
```



Proszę zmodyfikować procedury uczące i oceniające model, tabelę z modelami oraz skrypt optymalizujący hiperparametry modelu tak, żeby *F*-miarę zastąpić czułością.

Pora sprawdzić błąd rzeczywisty modeli. Tak jak poprzednio, ograniczę się do pokazania dziesięciu najlepszych i najgorszych modeli kandydujących.

```
SELECT [Name],[nTree],[mTry],[maxDepth],[minSplit],[Train F-score],[Test F-score]
FROM [LendingClub].[Models]
WHERE [Test F-score] in (SELECT TOP 10 [Test F-score] FROM [LendingClub].[Models] ORDER
BY [Test F-score] DESC)
  OR [Test F-score] in (SELECT TOP 10 [Test F-score] FROM [LendingClub].[Models] ORDER BY
[Test F-score])
ORDER BY [Test F-score] DESC;
```

Name	nTree	mTry	maxDepth	minSplit	Train F-score	Test F-score
Decision Forest	V1727	56	3	14	100	0.6941	0.4375
Decision Forest	V1791	38	3	12	100	0.6911	0.4372
Decision Forest	V1815	56	4	10	120	0.6909	0.4371
Decision Forest	V1861	30	6	10	240	0.6922	0.4361
Decision Forest	V1426	48	3	12	100	0.6947	0.4308
Decision Forest	V1753	58	4	10	180	0.6913	0.4301
Decision Forest	V1239	66	5	10	120	0.6921	0.4292
Decision Forest	V1750	40	4	16	200	0.6978	0.4289
Decision Forest	V2066	32	5	12	120	0.6993	0.4288
Decision Forest	V1321	64	3	14	200	0.6923	0.4285
Decision Forest	V2094	64	1	2	220	0.1354	0.0132
Decision Forest	V341	56	1	2	100	0.2726	0.0098
Decision Forest	V640	32	1	2	180	0.3507	0.0098
Decision Forest	V1853	38	1	2	400	0.0787	0.0083
Decision Forest	V60	68	1	2	140	0.2876	0.0050
Decision Forest	V283	66	1	2	220	0.1292	0.0050
Decision Forest	V1796	54	1	2	320	0.0879	0.0033
Decision Forest	V911	52	1	2	160	0.2041	0.0033
Decision Forest	V326	60	1	2	260	0.2766	0.0033
Decision Forest	V297	46	1	2	400	0.2978	0.0017

Okazało się, że wszystkie dziesięć modeli o najmniejszym błędzie empirycznym jest nadmiernie dopasowanych. Jak pamiętamy, z nadmiernym dopasowaniem mamy do czynienia, jeśli istnieje inny, nauczony na tych samych danych model o mniejszym błędzie rzeczywistym i większym błędzie empirycznym. Sposobem zmniejszenia ryzyka nadmiernego dopasowania było ograniczenie głębokości drzew i zmniejszenie liczby zmiennych uwzględnianych przy podziałach.

10.5.
Wdrożenie

Najlepszy model, a więc model o architekturze, która cechuje się najmniejszym błędem rzeczywistym, wdrożymy do użycia na serwerze SQL bez zainstalowanej usługi SQL Server Machine Learning.

Finalny model często uczy się na całym zbiorze danych. Zwiększając liczbę przykładów treningowych przez dodanie do nich danych testowych, możemy poprawić jakość modelu. Tym razem tak nie zrobię, co pozwoli porównać modele zserializowane z wyłączoną i z włączoną opcją `realtimeScoringOnly`. Dzięki temu będę mógł także zademonstrować metodę oceniania jakości modeli przy użyciu języka SQL.

Do zbudowania finalnej wersji modelu została użyta procedura `LendingClub.BuildRealtimeScoringModel`. Jedyną różnicą między nią a procedurą `LendingClub.BuildModels` jest włączenie opcji `realtimeScoringOnly` podczas serializacji modelu.

```
model <- rxSerializeModel(model, realtimeScoringOnly = TRUE)
```

Wywołując ją, należy podać wartości hiperparametrów budowanego modelu. Wybierzemy je na podstawie błędu rzeczywistego modeli kandydujących. *F*-miara sześciu najlepszych modeli kandydujących przekroczyła 43%. Z tych sześciu modeli wybrałem najprostszy. Mniejsza liczba drzew lasu losowego oznacza mniejszy model, krótszy czas uczenia i przede wszystkim szybsze wykonywanie zapytań predykcyjnych. Mniejsze drzewa decyzyjne zmniejszają ryzyko nadmiernego dopasowania modelu, zmniejszają wielkość całego modelu i również poprawiają jego wydajność, a przede wszystkim skracają czas jego uczenia. Mniejsza liczba zmiennych losowo analizowanych przy poszczególnych podziałach zmniejsza ryzyko nadmiernego dopasowania modelu, tak samo jak zwiększenie minimalnej liczby przykładów wymaganej do

utworzenia węzła. Dlatego najlepszymi modelami wydają się modele `V1861` i `V1791`. Ja wybrałem ten drugi.

```
EXEC [LendingClub].[BuildRealtimeScoringModel]
        @name = N'RT Decision Forest',
        @input_query = N'SELECT * FROM [LendingClub].[vLoanStats] WHERE [is_train] = 1',
        @n_tree = 38,
        @m_try = 3,
        @max_depth = 12,
        @minSplit = 100;
```

Porównajmy obie wersje modelu o wybranej architekturze:

```
SELECT [Name], 1.*DATALENGTH (Model)/(1024*1024) AS [Model MB], [Train F-score]
FROM [ML].[LendingClub].[Models]
WHERE nTree = 38 AND mTry = 3 AND maxDepth=12 AND minSplit=100;
```

Name	Model MB	Train F-score
Decision Forest V1791	3.31576728820	0.6911
RT Decision Forest	2.34794235229	0.6900

Błędy testowe są bardzo podobne, ale model zserializowany z opcją `realtimeScoringOnly` jest o 30% mniejszy. Jednak najważniejsza różnica między modelami polega na tym, że zapytania predykcyjne do nowego modelu można wysyłać za pośrednictwem funkcji `PREDICT` języka SQL. Poniższe instrukcje:

- odczytują finalny model z tabeli `LendingClub.Models`;
- czyszczą tabelę `LendingClub.LoanStatsPredictions` z wynikami predykcji;
- odnotowują w tabeli `LendingClub.RunTimeStats` czas rozpoczęcia wykonywania zapytania predykcyjnego;
- zapisują do zmiennej `d` dane testowe odczytane z widoku `LendingClub.vLoanStats`;
- wysyłają za pośrednictwem funkcji tabelarycznej `PREDICT` dane zapisane w zmiennej `d` do modelu i zapisują zwrócone przez ten model wyniki do

tabeli `LendingClub.LoanStatsPredictions`;

- odnotowują w tabeli `LendingClub.RunTimeStats` czas zakończenia wykonywania zapytania predykcyjnego.

```
DECLARE @model varbinary(max)
SELECT @model = [model] FROM [LendingClub].[Models]
       WHERE [name] = 'RT Decision Forest';

TRUNCATE TABLE [LendingClub].[LoanStatsPredictions]
-- Log beginning of processing time
INSERT INTO [LendingClub].[RunTimeStats] VALUES (@@SPID, GETDATE(),'Start scoring model RT
Decision Forest')

WITH d AS
       (SELECT id, is_bad, grade, int_rate, out_prncp_inv, policy_code, installment,
       open_acc_6m, all_util, revol_util, total_rec_prncp, bc_util,home_ownership
       FROM [LendingClub].[vLoanStats] WHERE [is_train] = 0)
INSERT INTO [LendingClub].[LoanStatsPredictions] (id, actual, predicted)
SELECT id, is_bad, is_bad_Pred
FROM PREDICT(MODEL = @model, DATA = d)
WITH ([0_prob] float, [1_prob] float, is_bad_Pred nvarchar(max)) p ;
GO

-- Log end of processing time
INSERT INTO [LendingClub].[RunTimeStats] VALUES (@@SPID, GETDATE(),'End scoring model RT
Decision Forest');
```

```
(14045 rows affected)
```

Odczytując dane z tabeli `LendingClub.RunTimeStats`, możemy porównać czasy wykonywania zapytań predykcyjnych przez poszczególne modele. W moim przypadku wszystkie zapytania trwały około 3 s.

```
SELECT TOP 2 *
FROM [LendingClub].[RunTimeStats]
ORDER BY RunTime DESC;
```

```
SessionID       RunTime  Operation
66              2019-04-13 07:06:09.103    End scoring model RT Decision Forest
66              2019-04-13 07:06:06.337    Start scoring model RT Decision Forest
```

 Tabela z danymi o użyciu modeli nie musi przechowywać wyłącznie czasu rozpoczęcia i nazwy operacji. Proszę zastąpić numery sesji wykonujących dane operacje użytkowników ich nazwami zwracanymi przez funkcję systemową `SUSER_SNAME`.

Wyniki predykcji uzupełnione o identyfikatory pożyczek oraz prawdziwą wartość zmiennej wyjściowej zostały zapisane do tabeli `LendingClub.LoanStatsPredictions`.

```
SELECT top 5 *
FROM [LendingClub].[LoanStatsPredictions];
```

id	actual	predicted
907439	1	0
922015	1	1
932534	1	0
824796	1	1
825601	1	1

Dysponując takimi danymi, możemy obliczyć dowolne metryki jakości modelu. Tak jak poprzednio, wyznaczymy trafność, precyzję, czułość i *F*-miarę modelu, a w tabeli z modelami zapiszemy wartość *F*-miary. Jednak tym razem zrobimy to, używając instrukcji języka SQL. W poniższym skrypcie:

- najpierw zostały zadeklarowane cztery zmienne typu *INT;*
- następnie do odpowiednich zmiennych zostały przypisane liczby prawdziwych pozytywów, prawdziwych negatywów, fałszywych pozytywów i fałszywych negatywów;
- na tej podstawie zostały policzone cztery metryki modelu;
- wartość *F*-miary została zapisana w odpowiednim wierszu tabeli `LendingClub.Models`.

```
DECLARE @tp INT, @fn INT, @fp INT, @tn INT
SELECT @tp = COUNT(*)
        FROM [LendingClub].[LoanStatsPredictions]
        WHERE [actual] =1 and [predicted]=1 ;

SELECT @fn = COUNT(*)
        FROM [LendingClub].[LoanStatsPredictions]
        WHERE [actual] =1 and [predicted]=0;

SELECT @fp = COUNT(*)
        FROM [LendingClub].[LoanStatsPredictions]
        WHERE [actual] =0 and [predicted]=1;

SELECT @tn = COUNT(*)
        FROM [LendingClub].[LoanStatsPredictions]
        WHERE [actual] =0 and [predicted]=0;

SELECT @tn AS TN, @fp AS FP;
SELECT @fn AS FN, @tp AS TP;

DECLARE @accuracy DECIMAL(3,2) = (@tp + @tn)*1. / (@tp + @fn + @fp + @tn)
DECLARE @precision DECIMAL(3,2) = @tp*1. / (@tp + @fp)
DECLARE @recall DECIMAL(3,2) = @tp*1. / (@tp + @fn)
DECLARE @fscore DECIMAL(3,2)  = 2. * (@precision * @recall) / (@precision + @recall)

SELECT @accuracy AS Accuracy, @precision AS Precision, @recall AS Recall, @fscore AS
[F-score];

UPDATE [LendingClub].[Models]
SET [Test F-score] = @fscore
WHERE [Name] = 'RT Decision Forest';
```

TN	FP		
11551	1176		
FN	TP		
588	618		
Accuracy	Precision	Recall	F-score
0.87	0.34	0.51	0.41

Raz jeszcze porównajmy obie wersje modelu o najlepszej architekturze, tym razem zwracając uwagę na ich błąd rzeczywisty.

```
SELECT [Name],[Train F-score],[Test F-score]
FROM [LendingClub].[Models]
WHERE nTree = 38 AND mTry = 3 AND maxDepth=12 AND minSplit=100;
```

Name	Train F-score	Test F-score
Decision Forest V1791	0.6911	0.4372
RT Decision Forest	0.6900	0.4100

Błąd rzeczywisty nowego modelu jest trochę większy, co można wytłumaczyć tym, że dane treningowe do widoku `LendingClub.vLoanStats` są wybierane w sposób losowy. Proszę pamiętać, że równoważąc dane, usuwamy z tego widoku 90% losowo wybranych przykładów większościowych.

Pozostało nam udostępnić wyniki predykcji użytkownikom. Zrobimy to w następnym punkcie, w którym przeprowadzimy symulację, czyli analizę typu *Co by było, gdyby?*.

10.5.1. Analiza typu *Co by było, gdyby?*

Modele predykcyjne zwracają wartość zmiennej wyjściowej obliczoną na podstawie zmiennych wejściowych. W praktyce używa się ich do uzupełniania brakujących danych o nowych, nieużytych do uczenia obserwacjach. Na przykład, żeby sklasyfikować wszystkie kredyty udzielone w IV kwartale 2018 r., należy wykonać poniższe instrukcje:

```
DECLARE @model varbinary(max)
SELECT @model = [model] from [LendingClub].[Models]
        WHERE [name] = 'RT Decision Forest';

TRUNCATE TABLE [LendingClub].[LoanStatsPredictions];

WITH d AS
        (SELECT id, is_bad, grade, int_rate, out_prncp_inv, policy_code, installment,
        open_acc_6m, all_util, revol_util, total_rec_prncp, bc_util,home_ownership
        FROM [LendingClub].[LoanStats] WHERE [is_train] = 0)
INSERT INTO [LendingClub].[LoanStatsPredictions] (id, actual, predicted)
SELECT id, is_bad, is_bad_Pred
FROM PREDICT(MODEL = @model, DATA = d)
WITH ([0_prob] float, [1_prob] float, is_bad_Pred nvarchar(max)) p;
```

```
(128412 rows affected)
```


Proszę pamiętać, że ja użyłem danych z IV kwartału 2018 r. Państwo powinni do tej pory zmienić metodę podziału danych na treningowe i testowe. Chociaż państwa dane testowe będą inne, nie zmienia to jednak sensu ćwiczenia.

Wybranie 128 tysięcy obserwacji z tabeli zawierającej 938 821 wierszy, ich klasyfikacja i zapisanie wyników w tabeli `LendingClub.LoanStatsPredictions` zajęło 24 s. W wyniku tej operacji uzyskaliśmy cenne biznesowo dane, bo kredyty automatycznie ocenione jako ryzykowane mogą zostać dodatkowo zweryfikowane przez specjalistów, co pozwoli firmie uniknąć udzielenia złych pożyczek i zwiększyć zyskowność.

Modele predykcyjne mogą być też używane do symulacji. Jeżeli zmienimy wartości zmiennych wejściowych, model wykona predykcje dla zasymulowanych obserwacji. Możemy na przykład sprawdzić, jak wpłynęłoby na spłatę kredytu podniesienie początkowej raty o 9%. W tym celu należy dodać do faktycznej wysokości początkowej raty 9%. Wyniki uzyskane dla tak zmodyfikowanych danych zapiszemy w tabeli `LendingClub.LoanPredictionsWhatIf`.

```
DECLARE @model varbinary(max)
SELECT @model = [model] from [LendingClub].[Models]
        WHERE [name] = 'RT Decision Forest';

TRUNCATE TABLE [LendingClub].[LoanPredictionsWhatIf];

WITH d AS
        (SELECT id, is_bad, grade, int_rate+9 as int_rate, out_prncp_inv, policy_code,
installment,
        open_acc_6m, all_util, revol_util, total_rec_prncp, bc_util,home_ownership
          FROM [LendingClub].[LoanStats]
          WHERE [is_train] = 0)
INSERT INTO [LendingClub].[LoanPredictionsWhatIf] (id, actual, predicted)
SELECT id, is_bad, is_bad_Pred
FROM PREDICT(MODEL = @model, DATA = d)
WITH ([0_prob] float, [1_prob] float, is_bad_Pred nvarchar(max)) p;
```

Jak łatwo sprawdzić, zwiększenie początkowej raty zmieniło klasyfikację ponad 9000 pożyczek.

```
SELECT COUNT(*)
FROM [LendingClub].[LoanStatsPredictions] as P
JOIN [LendingClub].[LoanPredictionsWhatIf] as I
ON P.id=I.id
WHERE P.predicted<>I.predicted;
```

```
9250
```

Okazuje się, że raz zbudowany **model predykcyjny może być użyty nie tylko do oceniania ryzyka nowych pożyczek, ale również do przeprowadzania symulacji**, takich jak ocena wpływu zwiększenia początkowej raty kredytu na ryzyko jego niespłacenia w terminie.

Porównajmy wyniki predykcji dla IV kwartału 2018 r. z wynikami uzyskanymi przy założeniu podniesienia początkowej raty spłaty kredytu. Ponieważ model przypisuje złym kredytom wartość 1, a dobrym 0, możemy ocenić wpływ zmiany za pomocą statystyk opisowych, takich jak średnia arytmetyczna czy odchylenie standardowe.

```
SELECT AVG(1.*predicted) AS Average, STDEV(predicted) AS StDeviation
FROM [LendingClub].[LoanPredictionsWhatIf]
UNION ALL
SELECT AVG(1.*predicted), STDEV(predicted)
FROM [LendingClub].[LoanStatsPredictions];
```

Average	StDeviation
0.161386	0.367888767786388
0.091860	0.288830154958447

Przeprowadzona symulacja pozwala przypuszczać, że podniesienie początkowej raty o 9% zwiększy odsetek złych kredytów z 9% do 16%.

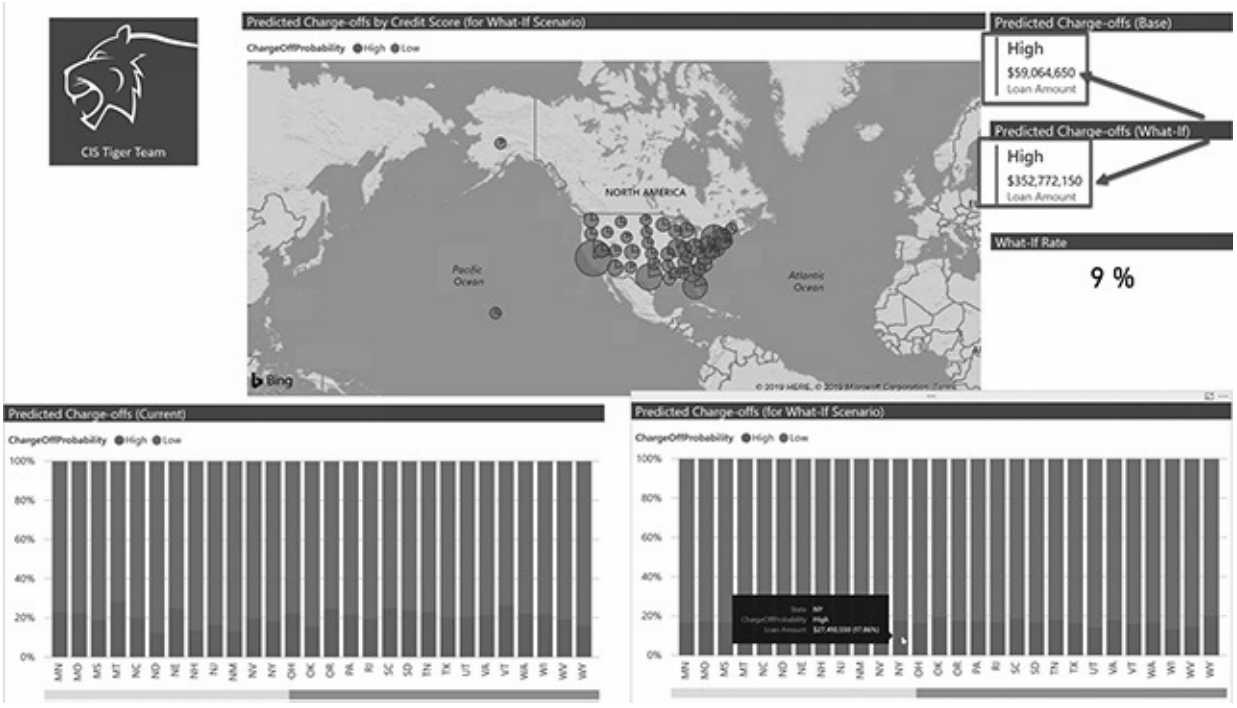
Użytkownicy biznesowi woleliby jednak zobaczyć wyniki naszej analizy w przyjaźniejszej, graficznej formie. Zależy im również na rozbiciu wyników na poszczególne stany i stosowaną przez nich ocenę kredytów. Ponadto, oprócz liczby kredytów interesuje ich wartość udzielonych pożyczek. Takie dane zwraca poniższe zapytanie. W ten sam sposób, zastępując jedynie tabelę `LendingClub.LoanStatsPredictions` tabelą `LendingClub.LoanPredictionsWhatIf`, otrzymamy wyniki symulacji.

```
SELECT loan_status,st.addr_state,CASE WHEN p.predicted = 1 THEN 'High'
       ELSE 'Low' END AS [ChargeOffProbability],
       COUNT(*) as [Number of Loans], SUM(funded_amnt) AS [Loan Amount]
FROM [LendingClub].[LoanStats] st
INNER JOIN [LendingClub].[LoanStatsPredictions] p
on st.id = p.id
GROUP BY loan_status,st.addr_state, CASE WHEN p.predicted = 1 THEN 'High'
       ELSE 'Low' END;
```

loan_status	addr_state	ChargeOffProbability	Number of Loans	Loan Amount
Late (31-120 days)	NE	High	1	22000
Fully Paid	DC	Low	13	140300
Current	AK	High	24	429150
Late (16-30 days)	IN	High	3	35000
Fully Paid	VT	Low	5	94725
…				

Do graficznego przedstawienia danych użyłem programu Power BI Desktop.

Pokazany na rysunku 10.3 raport ułatwia ocenę kosztów ewentualnego podniesienia początkowej raty i wpływ takiej zmiany na spłacalność pożyczek w poszczególnych stanach.



Rysunek 10.3. Podnosząc ratę o 9% firma ryzykuje wzrost wartości złych kredytów do 350 mln USD. Jednak taka zmiana różnie wpłynęłaby na procent złych kredytów w poszczególnych stanach

Posłowie

Tu zaczyna się Państwa historia. Uczenie maszynowe rozwija się niezwykle gwałtownie. Przedstawione w książce techniki pracy z danymi i algorytmy uczenia maszynowego są efektem ponad pięćdziesięciu lat badań, a ich skuteczność została sprawdzona w tysiącach, prowadzonych przez ostatnie trzy dekady, projektów. Dzisiaj stanowią one kanon uczenia maszynowego. Ich opanowanie nie tylko pozwoli rozwiązywać praktyczne problemy, takie jak opisane w książce, ale również pomoże wybrać dziedzinę uczenia maszynowego, w której chcą się Państwo specjalizować.

Bo uczenie maszynowe podzieliło się na kilka specjalności. Przedstawione metody nazywamy klasycznym uczeniem maszynowym, w przeciwieństwie do nowych metod głębokiego i wzmocnionego uczenia maszynowego.

Głębokie uczenie maszynowe polega na automatycznym tworzeniu coraz bardziej abstrakcyjnych cech przykładów treningowych i szukaniu funkcji, która zwróci wartość zmiennej wyjściowej na podstawie tak znalezionych cech, a nie bezpośrednio na podstawie zmiennych wejściowych. Głębokie uczenie maszynowe zdominowało analizę obrazu, mowy i języka naturalnego, ale bywa też używane do analizy danych tabelarycznych.

Z kolei uczenie maszynowe ze wzmacnianiem pozwala rozwiązać problemy, których nie potrafimy zamodelować. Polega ono na nagradzaniu lub karaniu agenta za podejmowane przez niego decyzje. Jedyne, czym dysponujemy, to informacje o stanie środowiska, w którym znajduje się agent. Wykonując akcje, agent poznaje środowisko i uczy się polityki prowadzącej do zdobywania jak największej liczby nagród. Uczenie maszynowe ze wzmacnianiem jest

powszechnie stosowane do optymalizacji procesów (np. sterowania sygnalizacją świetlną, tak żeby usprawnić ruch samochodów), w robotyce (m.in. przy uczeniu robotów chodzenia czy jeżdżenia), w systemach rekomendujących i grach.

Czego by Państwo nie wybrali, proszę pamiętać, że uczenie maszynowe nie sprowadza się do biegłego posługiwania się językiem R, Python czy SQL. Ani tym bardziej do opanowania pewnych bibliotek, na przykład `scikit-learn`, `MicrosoftML` czy `Keras`. Oczywiście, umiejętność zaimplementowania modeli jest ważna, ale uczenie maszynowe to interdyscyplinarna nauka, a od jej praktyków wymaga się więcej niż umiejętności programowania. Tu najważniejsza jest umiejętność rozwiązywania problemów przy użyciu danych.

Powodzenia!

Bibliografia

[1] Marcin Szeliga, Danuta Mendrala, *Serwer SQL 2008. Usługi biznesowe. Analiza i eksploracja danych*, Helion 2009.

[2] Marcin Szeliga, Danuta Mendrala, *Microsoft SQL Server. Modelowanie i eksploracja danych*, Helion 2012.

[3] Marcin Szeliga, *Data Science i uczenie maszynowe*, Wydawnictwo Naukowe PWN, 2017.

[4] Pete Chapman, Julian Clinton, Randy Kerber, Thomas Khabaza, Thomas Reinartz, Colin Shearer, Rüdiger Wirth, *Step-by-step data mining guide*, https://www.the-modeling-agency.com/crisp-dm.pdf, 2000.

[5] Alan Mathison Turing, *Computing Machinery and Intelligence*. Mind 49: 433-460, 1950, https://www.csee.umbc.edu/courses/471/papers/turing.pdf.

[6] John McCarthy, *The Logic and Philosophy of Artificial Intelligence*, Stanford University lecture, 1988, https://www.kyotoprize.org/wp/wp-content/uploads/2016/02/4kA_lct_EN.pdf.

[7] Tom M. Mitchell, *Machine Learning*, McGraw-Hill, 1977, https://www.cs.ubbcluj.ro/~gabis/ml/ml-books/McGrawHill%20-%20Machine%20Learning%20Tom%20Mitchell.pdf.

[8] Cai Huang, Evan A. Clayton, Lilya V. Matyunina, L. DeEtte McDonald, Benedict B. Benigno, Fredrik Vannberg & John F. McDonald, *Machine learning predicts individual cancer patient responses to therapeutic drugs with high accuracy*, Scientific Reports 8, Article number: 16444, 2018,

https://www.nature.com/articles/s41598-018-34753-5.

[9] Jerry Overton, *Going pro in data science*, O'Reilly, 2016.

[10] The State of Data Science & Machine Learning, Industry-wide survey by Kaggle, 2017, https://www.kaggle.com/surveys/2017.

[11] Simon Newcomb, *Note on the frequency of use of the different digits in natural numbers*, American Journal of Mathematics, Vol. 4, No. 1. 4 (1/4): 39–40, 1981.

[12] Frank Benford, *The law of anomalous numbers*, Proc. Am. Philos. Soc. 78 (4): 551–572, 1938.

[13] Max Welling, Donald Bren, *A first encounter with machine learning*, 2011, https://www.ics.uci.edu/~welling/teaching/ICS273Afall11/IntroMLBook.pdf, 26-11-2016.

[14] John Duchi, *CS229 Supplemental Lecture notes, Hoeffding's inequality*, 2017, http://cs229.stanford.edu/extra-notes/hoeffding.pdf.

[15] Hal Daumé III, *A Course in Machine Learning*, Self-published, 2017.

[16] Goodfellow Ian, Pouget-Abadie Jean, Mirza Mehdi, Xu Bing, Warde-Farley David, Ozair Sherjil, Courville Aaron, Bengio Yoshua, *Generative Adversarial Networks*, 2014, https://arxiv.org/pdf/1406.2661.pdf.

[17] John Ioannidis, *Why most published research findings are false?*, PLOS Medicine, 2(8): e124, 2005, doi: 10.1371/journal.pmed.0020124.

[18] Krzysztof Cios, Witold Pedrycz, Roman Swiniarski, Łukasz Kurgan, *Data Mining: A Knowledge Discovery Approach*, Springer 2011.

[19] Pedro Domingos, *A Few Useful Things to Know about Machine Learning*, 2012, https://homes.cs.washington.edu/~pedrod/papers/cacm12.pdf.

[20] Hadley Wickham, *Tidy Data*, Journal of Statistical Software, August 2014,

Volume 59, Issue 10, https://www.jstatsoft.org/article/view/v059i10/v59i10.pdf.

[21] Francis John Anscombe, *Graphs in Statistical Analysis*, American Statistician, Vol. 27, No. 1, s. 17–21.

[22] Claude Elwood Shannon, A Mathematical Theory of Communication, *The Bell System Technical Journal*, 1948, http://math.harvard.edu/~ctm/home/text/others/shannon/entropy/entropy.pdf.

[23] John W. Tukey, *Exploratory Data Analysis*, Pearson 1977.

[24] Wayne Eckerson, Colin White, *Evaluating ETL and Data Integration Platforms*, TDWI Report Series, 2003, http://download.101com.com/tdwi/research_report/2003ETLReport.pdf.

[25] Edgar Frank Codd, *A relational model of data for large shared data banks*, Magazine Communications of the ACM, Vol. 13 Issue 6, June 1970, https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf.

[26] Ronald Fisher, *Iris Data Set*, 1936, https://archive.ics.uci.edu/ml/datasets/iris.

[27] Mark Schmidt, *Least Squares Optimization with L1-Norm Regularization*, 2005, https://www.cs.ubc.ca/~schmidtm/Documents/2005_Notes_Lasso.pdf.

[28] Léon Bottou, Frank E. Curtis, Jorge Nocedal, *Optimization Methods for Large-Scale Machine Learning*, 2018, https://arxiv.org/pdf/1606.04838.pdf.

[29] John Duchi, Elad Hazan, Yoram Singer, *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization*, Journal of Machine Learning Research 12 (2011), http://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf.

[30] Diederik P. Kingma, Jimmy Lei Ba, *ADAM: a method for stochastic optimization, Published as a conference paper at ICLR 2015*,

https://arxiv.org/pdf/1412.6980.pdf.

[31] Shai Shalev-Shwartz, Tong Zhang, *Stochastic Dual Coordinate Ascent Methods for Regularized Loss Minimization, Journal of Machine Learning Research* 14 (2013), http://www.jmlr.org/papers/volume14/shalev-shwartz13a/shalev-shwartz13a.pdf.

[32] Marvin Minsky, Seymour Papert, *Perceptrons. An Introduction to Computational Geometry*. M.I.T. Press, 1969.

[33] David E. Rumelhart, Geoffrey E. Hinton, Ronald J. Williams, *Learning representations by back-propagating errors*, Nature, 323, 1986-10-09.

[34] Jerome H. Friedman, Greedy function approximation: *A gradient boosting machine*, The Annals of Statistics, Volume 29, Number 5 (2001), https://projecteuclid.org/download/pdf_1/euclid.aos/1013203451.

[35] Tianqi Chen, Carlos Guestrin, Xgboost: *A scalable tree boosting system*, Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining, 2016, https://www.kdd.org/kdd2016/papers/files/rfp0697-chenAemb.pdf.

[36] Marco Tulio Ribeiro, Sameer Singh, Carlos Guestrin, *Model-Agnostic Interpretability of Machine Learning*, 2016 ICML Workshop on Human Interpretability in MachineLearning (WHI 2016), New York, NY, USA, http://sameersingh.org/files/papers/lime-whi16.pdf.

[37] Scott M. Lundberg, Su-In Lee, *A Unified Approach to Interpreting ModelPredictions, 31st Conference on Neural Information Processing Systems* (NIPS 2017), Long Beach, CA, USA, https://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf.

[38] Justin Matejka, George Fitzmaurice, *Same Stats, Different Graphs:*

Generating Datasets with Varied Appearance and Identical Statistics through Simulated Annealing, https://www.autodeskresearch.com/sites/default/files/SameStats-DifferentGraphs.pdf.

[39] Ackoff, R. L., *From data to wisdom*, Journal of Applied Systems Analysis, 1989, Volume 15, pp. 3-9.

[40] Owens John D., Mike Houston, David Luebke, Simon Green, John E. Stone, James C. Phillips. *GPU computing*. (2008): 879-899. https://cloudfront.escholarship.org/dist/prd/content/qt0cv1p1nc/qt0cv1p1nc.p

[41] LeCun Y., Bengio Y., Hinton G. (2015). *Deep learning. Nature*, 521(7553), 436. https://creativecoding.soe.ucsc.edu/courses/cs523/slides/week3/DeepLearning

[42] Sharma Mamta. *Compression using Huffman coding*. IJCSNS International Journal of Computer Science and Network Security 10, no. 5 (2010): 133-141. https://pdfs.semanticscholar.org/89c2/03919ce446762d9b768ac0974f54021f3

[43] Larson Per-Ake, Cipri Clinciu, Campbell Fraser, Eric N. Hanson, Mostafa Mokhtar, Michal Nowakiewicz, Vassilis Papadimos et al. *Enhancements to SQL server column stores*. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, pp. 1159-1168. ACM, 2013.https://15721.courses.cs.cmu.edu/spring2016/papers/larson-sigmod2013.pdf.

[44] Ian Goodfellow, Yoshua Bengio, Aaron Courville, *Deep Learning*, MIT Press, 2016, http://www.deeplearningbook.org/.

Przypisy

1 Pod adresem https://www.thefourthrevolution.org/FourthRevolutionBookSchem.html znajdą Państwo kilka grafik z książki *The Fourth Revolution* ilustrujących to zjawisko.

2 Sposób uczenia się drzew decyzyjnych został opracowany niezależnie przez Johna Rossa Quinlana (twórcę algorytmu ID3) oraz Leo Breimana, Jerome'a Friedmana, Richarda Olshena i Charlesa Stone'a (twórców algorytmu CART).

3 Ocena jakości danych została dodana do programu Power BI Desktop w listopadzie 2018 r. i w czasie powstawania książki funkcjonalność ta była jeszcze w fazie testów. Testowe funkcjonalności trzeba samodzielnie włączyć w oknie ustawień programu.

4 Autorem zapytań jest Dejan Sarka. Zostały one przedstawione za zgodą Autora.

5 Autorem rozwiązania jest Sławomir Malinowski. Zostało ono przedstawione za zgodą Autora.

6 Zależność funkcyjna jest pojęciem z teorii relacyjnych baz danych. Zmienna *X* wyznacza funkcyjnie zmienną *Y*, jeżeli każda wartość zmiennej *X* jednoznacznie wskazuje na określoną wartość zmiennej *Y*. To samo możemy powiedzieć następująco – zmienna *Y* jest zależna funkcyjnie od zmiennej *X*, jeżeli każdą wartość zmiennej *Y* można jednoznacznie wskazać na podstawie dowolnej wartości zmiennej *X*.

7 Autorem zapytań statystycznych jest Dejan Sarka. Zostały one przedstawione za zgodą Autora.

8 Użycie głębokich sieci neuronowych do wzbogacania danych jest nową techniką. Jej popularność zaczęła się w 2018 r. Zainteresowani Czytelnicy znajdą informacje na jej temat, wyszukując w internecie frazę *deep learning for tabular data*.

9 Kompletny przykład znajdą Państwo w dołączonym do książki pliku *Feature engineering.R*.

10 Komplety przykład podziału danych znajdą Państwo w dołączonym do książki pliku *Sampling.R*.

11 Definicje funkcji `evaluate_classifier` niepokazanej na listingu znajdą Państwo w dołączonym do książki pliku *Cross validation.R*.

12 Wcześniej sprawdziłem, że cyfry 4 trafiły do pierwszego klastra, czyli klastra o numerze 0. Proszę samemu zobaczyć, jak wyglądają centra pozostałych klastrów.

13 Przykładowe dane oraz omówienie skryptu są dostępne w pliku z przykładami do książki.

14 Macierze korelacji oraz wykresy dekompozycji szeregów czasowych pochodzą z dołączonego do książki pliku *Correlations.pbix*.

15 Kompletne rozwiązanie znajduje się w dołączonym do książki pliku *Data preparation.R*.

16 Funkcja znajduje się w dołączonym do książki pliku *Regression.R*.