



Technologia i rozwiązania

# Python Uczanie maszynowe



**Sebastian Raschka**

[PACKT] open source\*  
PUBLISHING

Tytuł oryginału: Python Machine Learning

Tłumaczenie: Krzysztof Sawka

ISBN: 978-83-283-3614-8

Copyright © Packt Publishing 2016

First published in the English language under the title 'Python Machine Learning - (9781783555130)'.

Polish edition copyright © 2017 by Helion SA  
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicielami.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION  
ul. Kościuszki 1c, 44-100 GLIWICE  
tel. 32 231 22 19, 32 230 98 63  
e-mail: [helion@helion.pl](mailto:helion@helion.pl)  
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!  
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres  
[http://helion.pl/user/opinie/pythum\\_ebook](http://helion.pl/user/opinie/pythum_ebook)  
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

# Spis treści

<b>Przedmowa</b>	<b>11</b>
<b>Informacje o autorze</b>	<b>13</b>
<b>Informacje o recenzentach</b>	<b>15</b>
<b>Wstęp</b>	<b>17</b>
<hr/> <b>Rozdział 1. Umożliwianie komputerom uczenia się z danych</b>	<b>25</b>
<b>Tworzenie inteligentnych maszyn służących do przekształcania danych w wiedzę</b>	<b>26</b>
<b>Trzy różne rodzaje uczenia maszynowego</b>	<b>26</b>
Prognozowanie przyszłości za pomocą uczenia nadzorowanego	27
Rozwiązywanie problemów interaktywnych za pomocą uczenia przez wzmacnianie	29
Odkrywanie ukrytych struktur za pomocą uczenia nienadzorowanego	30
<b>Wprowadzenie do podstawowej terminologii i notacji</b>	<b>31</b>
<b>Strategia tworzenia systemów uczenia maszynowego</b>	<b>33</b>
Wstępne przetwarzanie — nadawanie danym formy	34
Trenowanie i dobór modelu predykcyjnego	35
Ewaluacja modeli i przewidywanie wystąpienia nieznanych danych	36
<b>Wykorzystywanie środowiska Python do uczenia maszynowego</b>	<b>36</b>
Instalacja pakietów w Pythonie	36
<b>Podsumowanie</b>	<b>38</b>
<hr/> <b>Rozdział 2. Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji</b>	<b>41</b>
<b>Sztuczne neurony — rys historyczny początków uczenia maszynowego</b>	<b>42</b>
<b>Implementacja algorytmu uczenia perceptronu w Pythonie</b>	<b>47</b>
Trenowanie modelu perceptronu na zestawie danych Iris	50
<b>Adaptacyjne neurony liniowe i zbieżność uczenia</b>	<b>54</b>
Minimalizacja funkcji kosztu za pomocą metody gradientu prostego	55
Implementacja adaptacyjnego neuronu liniowego w Pythonie	57
Wielkoskalowe uczenie maszynowe i metoda stochastycznego spadku wzduł gradientu	62
<b>Podsumowanie</b>	<b>67</b>

**Rozdział 3. Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn****69**

<b>Wybór algorytmu klasyfikującego</b>	<b>70</b>
<b>Pierwsze kroki z biblioteką scikit-learn</b>	<b>70</b>
Uczenie perceptronu za pomocą biblioteki scikit-learn	71
<b>Modelowanie prawdopodobieństwa przynależności do klasy za pomocą regresji logistycznej</b>	<b>76</b>
Teoretyczne podłożę regresji logistycznej i prawdopodobieństwa warunkowego	76
Wyznaczanie wag logistycznej funkcji kosztu	79
Uczenie modelu regresji logistycznej za pomocą biblioteki scikit-learn	81
Zapobieganie nadmiernemu dopasowaniu za pomocą regularizacji	84
<b>Wyznaczanie maksymalnego marginesu za pomocą maszyn wektorów nośnych</b>	<b>87</b>
Teoretyczne podłożę maksymalnego marginesu	87
Rozwiązywanie przypadków nieliniowo rozdzielnych za pomocą zmiennych uzupełniających	88
Alternatywne implementacje w interfejsie scikit-learn	90
<b>Rozwiązywanie nieliniiowych problemów za pomocą jądra SVM</b>	<b>91</b>
Stosowanie sztuczki z funkcją jądra do znajdowania przestrzeni rozdzielających w przestrzeni o większej liczbie wymiarów	93
<b>Uczenie drzew decyzyjnych</b>	<b>97</b>
Maksymalizowanie przyrostu informacji — osiąganie jak największych korzyści	98
Budowanie drzewa decyzyjnego	101
Łączenie słabych klasyfikatorów w silne klasyfikatory za pomocą modelu losowego lasu	104
<b>Algorytm k-najbliższych sąsiadów — model leniwego uczenia</b>	<b>106</b>
<b>Podsumowanie</b>	<b>109</b>
<b>Rozdział 4. Tworzenie dobrych zbiorów uczących — wstępne przetwarzanie danych</b>	<b>111</b>
<b>Kwestia brakujących danych</b>	<b>111</b>
Usuwanie próbek lub cech niezawierających wartości	113
Wstawianie brakujących danych	114
Estymatory interfejsu scikit-learn	114
<b>Przetwarzanie danych kategoryzujących</b>	<b>116</b>
Mapowanie cech porządkowych	116
Kodowanie etykiet klas	117
Kodowanie „gorącojedynkowe” cech nominalnych (z użyciem wektorów własnych)	118
<b>Rozdzielenie zestawu danych na podzbiory uczące i testowe</b>	<b>120</b>
<b>Skalowanie cech</b>	<b>121</b>
<b>Dobór odpowiednich cech</b>	<b>123</b>
Regularizacja L1	124
Algorytmy sekwencyjnego wyboru cech	129
<b>Ocenianie istotności cech za pomocą algorytmu losowego lasu</b>	<b>134</b>
<b>Podsumowanie</b>	<b>137</b>

<b>Rozdział 5. Kompresja danych poprzez redukcję wymiarowości</b>	<b>139</b>
<b>Nienadzorowana redukcja wymiarowości za pomocą analizy głównych składowych</b>	<b>140</b>
Wyjaśniona wariancja całkowita	141
Transformacja cech	145
Analiza głównych składowych w interfejsie scikit-learn	147
<b>Nadzorowana kompresja danych za pomocą liniowej analizy dyskryminacyjnej</b>	<b>150</b>
Obliczanie macierzy rozproszenia	151
Dobór dyskryminant liniowych dla nowej podprzestrzeni cech	154
Rzutowanie próbek na nową przestrzeń cech	156
Implementacja analizy LDA w bibliotece scikit-learn	156
<b>Jądrowa analiza głównych składowych jako metoda odwzorowywania nierozdzielnych liniowo klas</b>	<b>158</b>
Funkcje jądra oraz sztuczka z funkcją jądra	160
Implementacja jądrowej analizy głównych składowych w Pythonie	164
Rzutowanie nowych punktów danych	170
Algorytm jądrowej analizy głównych składowych w bibliotece scikit-learn	174
<b>Podsumowanie</b>	<b>175</b>
<b>Rozdział 6. Najlepsze metody oceny modelu i strojenie parametryczne</b>	<b>177</b>
<b>Usprawnianie cyklu pracy za pomocą kolejkowania</b>	<b>177</b>
Wczytanie zestawu danych Breast Cancer Wisconsin	178
Łączenie funkcji transformujących i estymatorów w kolejce czynności	179
<b>Stosowanie k-krotnego sprawdzianu krzyżowego w ocenie skuteczności modelu</b>	<b>180</b>
Metoda wydzielania	181
K-krotny sprawdzian krzyżowy	182
<b>Sprawdzanie algorytmów za pomocą krzywych uczenia i krzywych walidacji</b>	<b>186</b>
Diagnozowanie problemów z obciążeniem i wariancją za pomocą krzywych uczenia	186
Rozwiązywanie problemów nadmiernego i niewystarczającego dopasowania za pomocą krzywych walidacji	189
<b>Dostrajanie modeli uczenia maszynowego za pomocą metody przeszukiwania siatki</b>	<b>191</b>
Strojenie hiperparametrów przy użyciu metody przeszukiwania siatki	192
Dobór algorytmu poprzez zagnieżdżony sprawdzian krzyżowy	193
<b>Przegląd metryk oceny skuteczności</b>	<b>195</b>
Odczytywanie macierzy pomyłek	195
Optymalizacja precyzji i pełności modelu klasyfikującego	197
Wykres krzywej ROC	198
Metryki zliczające dla klasyfikacji wieloklasowej	201
<b>Podsumowanie</b>	<b>202</b>
<b>Rozdział 7. Łączenie różnych modeli w celu uczenia zespołowego</b>	<b>203</b>
<b>Uczenie zespołów</b>	<b>203</b>
<b>Implementacja prostego klasyfikatora wykorzystującego głosowanie większościowe</b>	<b>207</b>
Łączenie różnych algorytmów w celu klasyfikacji za pomocą głosowania większościowego	213

<b>Ewaluacja i strojenie klasyfikatora zespołowego</b>	<b>216</b>
<b>Agregacja — tworzenie zespołu klasyfikatorów za pomocą próbek początkowych</b>	<b>221</b>
<b>Usprawnianie słabych klasyfikatorów za pomocą wzmacnienia adaptacyjnego</b>	<b>226</b>
<b>Podsumowanie</b>	<b>232</b>
<b>Rozdział 8. Wykorzystywanie uczenia maszynowego w analizie sentymentów</b>	<b>235</b>
<b>Zestaw danych IMDb movie review</b>	<b>235</b>
<b>Wprowadzenie do modelu worka słów</b>	<b>237</b>
Przekształcanie słów w wektory cech	238
Ocena istotności wyrazów za pomocą ważenia częstości termów — odwrotnej częstości w tekście	239
Oczyszczanie danych tekstowych	241
Przetwarzanie tekstu na znaczniki	243
<b>Uczenie modelu regresji logistycznej w celu klasyfikowania tekstu</b>	<b>245</b>
<b>Praca z większą ilością danych — algorytmy sieciowe i uczenie pozardzeniowe</b>	<b>247</b>
<b>Podsumowanie</b>	<b>250</b>
<b>Rozdział 9. Wdrażanie modelu uczenia maszynowego do aplikacji sieciowej</b>	<b>251</b>
<b>Serializacja wyuczonych estymatorów biblioteki scikit-learn</b>	<b>252</b>
<b>Konfigurowanie bazy danych SQLite</b>	<b>254</b>
<b>Tworzenie aplikacji sieciowej za pomocą środowiska Flask</b>	<b>256</b>
<b>Nasza pierwsza aplikacja sieciowa</b>	<b>257</b>
Sprawdzanie i wyświetlanie formularza	258
Przekształcanie klasyfikatora recenzji w aplikację sieciową	262
<b>Umieszczanie aplikacji sieciowej na publicznym serwerze</b>	<b>269</b>
Aktualizowanie klasyfikatora recenzji filmowych	271
<b>Podsumowanie</b>	<b>272</b>
<b>Rozdział 10. Przewidywanie ciągłych zmiennych docelowych za pomocą analizy regresywnej</b>	<b>275</b>
<b>Wprowadzenie do prostego modelu regresji liniowej</b>	<b>276</b>
<b>Zestaw danych Housing</b>	<b>277</b>
Wizualizowanie ważnych elementów zestawu danych	278
<b>Implementacja modelu regresji liniowej wykorzystującego zwykłą metodę najmniejszych kwadratów</b>	<b>282</b>
Określanie parametrów regresywnych za pomocą metody gradientu prostego	283
Szacowanie współczynnika modelu regresji za pomocą biblioteki scikit-learn	286
<b>Uczenie odpornego modelu regresywnego za pomocą algorytmu RANSAC</b>	<b>288</b>
<b>Ocenianie skuteczności modeli regresji liniowej</b>	<b>291</b>
<b>Stosowanie regularyzowanych metod regresji</b>	<b>294</b>

<b>Przekształcanie modelu regresji liniowej w krzywą — regresja wielomianowa</b>	<b>295</b>
Modelowanie nieliniowych zależności w zestawie danych Housing	297
Analiza nieliniowych relacji za pomocą algorytmu losowego lasu	300
<b>Podsumowanie</b>	<b>305</b>
<b>Rozdział 11. Praca z nieoznaczonymi danymi — analiza skupień</b>	<b>307</b>
<b>Grupowanie obiektów na podstawie podobieństwa przy użyciu algorytmu centroidów</b>	<b>308</b>
Algorytm k-means++	311
Klasteryzacja twarda i miękka	312
Stosowanie metody łokcia do wyszukiwania optymalnej liczby skupień	315
Ujęcie ilościowe jakości klasteryzacji za pomocą wykresu profilu	316
<b>Organizowanie skupień do postaci drzewa klastrów</b>	<b>320</b>
Przeprowadzanie hierarchicznej analizy skupień na macierzy odległości	323
Dołączanie dendrogramów do mapy cieplnej	326
Aglomeracyjna analiza skupień w bibliotece scikit-learn	328
<b>Wyznaczanie rejonów o dużej gęstości za pomocą algorytmu DBSCAN</b>	<b>328</b>
<b>Podsumowanie</b>	<b>333</b>
<b>Rozdział 12. Trenowanie sztucznych sieci neuronowych w rozpoznawaniu obrazu</b>	<b>335</b>
<b>Modelowanie złożonych funkcji przy użyciu sztucznych sieci neuronowych</b>	<b>336</b>
Jednowarstwowa sieć neuronowa — powtóżenie	337
Wstęp do wielowarstwowej architektury sieci neuronowych	338
Aktywacja sieci neuronowej za pomocą propagacji w przód	340
<b>Klasyfikowanie pisma odręcznego</b>	<b>343</b>
Zestaw danych MNIST	344
Implementacja wielowarstwowego perceptronu	348
<b>Trenowanie sztucznej sieci neuronowej</b>	<b>356</b>
Obliczanie logistycznej funkcji kosztu	356
Uczenie sieci neuronowych za pomocą algorytmu wstecznej propagacji	359
<b>Ujęcie intuicyjne algorytmu wstecznej propagacji</b>	<b>361</b>
<b>Usuwanie błędów w sieciach neuronowych za pomocą sprawdzania gradientów</b>	<b>363</b>
<b>Zbieżność w sieciach neuronowych</b>	<b>368</b>
<b>Inne architektury sieci neuronowych</b>	<b>370</b>
Splotowe sieci neuronowe	370
Rekurencyjne sieci neuronowe	371
<b>Jeszcze słowo o implementacji sieci neuronowej</b>	<b>373</b>
<b>Podsumowanie</b>	<b>373</b>
<b>Rozdział 13. Równoległe przetwarzanie sieci neuronowych za pomocą biblioteki Theano</b>	<b>375</b>
<b>Tworzenie, komplikowanie i uruchamianie wyrażeń w interfejsie Theano</b>	<b>376</b>
Czym jest Theano?	377
Pierwsze kroki z Theano	378
Konfigurowanie środowiska Theano	379

Praca ze strukturami tablicowymi	381
Przejdzmy do konkretów — implementacja regresji liniowej w Theano	384
<b>Dobór funkcji aktywacji dla jednokierunkowych sieci neuronowych</b>	<b>387</b>
Funkcja logistyczna — powtóżenie	388
Szacowanie prawdopodobieństw w klasyfikacji wieloklasowej za pomocą znormalizowanej funkcji wykładniczej	390
Rozszerzanie zakresu wartości wyjściowych za pomocą funkcji tangensa hiperbolicznego	391
<b>Skuteczne uczenie sieci neuronowych za pomocą biblioteki Keras</b>	<b>393</b>
<b>Podsumowanie</b>	<b>398</b>
<b>Skorowidz</b>	<b>401</b>

# Zespół wydania oryginalnego

**Author**

Sebastian Raschka

**Reviewers**

Richard Dutton  
Dave Julian  
Vahid Mirjalili  
Hamidreza Sattari  
Dmytro Taranovsky

**Commissioning Editor**

Akram Hussain

**Acquisition Editors**

Rebecca Youe  
Meeta Rajani

**Content Development Editor**

Riddhi Tuljapurkar

**Technical Editors**

Madhunikita Sunil Chindarkar  
Taabish Khan

**Copy Editors**

Roshni Banerjee  
Stephan Copestake

**Project Coordinator**

Kinjal Bari

**Proofreader**

Safis Editing

**Indexer**

Hemangini Bari

**Graphics**

Sheetal Aute  
Abhinash Sahu

**Production Coordinator**

Shantanu N. Zagade

**Cover Work**

Shantanu N. Zagade



# Przedmowa

Żyjemy w erze obfitości danych. Zgodnie z najnowszymi badaniami codziennie generujemy 2,5 kwintyliona ( $10^{18}$ ) bajtów informacji. To jest taki ogrom danych, że ponad 90% współcześnie przechowywanych informacji zostało utworzone w ciągu poprzedniej dekady. Niestety, w zdecydowanej większości są one dostępne w postaci niezrozumiałej dla człowieka. Albo dane te wykraczają poza ramy standardowych metod analitycznych, albo po prostu są zbyt rozległe, aby ludzki umysł był w stanie je objąć.

Dzięki uczeniu maszynowemu wykorzystujemy komputery do przetwarzania tych nieprzeniknionych dla człowieka zbiorów danych, wyciągania z nich wniosków oraz prowadzenia na ich podstawie odpowiednich działań. Cały świat powoli uzależnia się od uczenia maszynowego — często nawet nieświadomie — począwszy od smartfonów schowanych w kieszeniach, skończywszy na potężnych superkomputerach zasilających silniki przeglądarek w siedzibie firmy Google.

Wypada więc nam, współczesnym pionierom nowego, wspaniałego świata wielkich danych, podskolić się co nieco w dziedzinie uczenia maszynowego. Czym jest ta dyscyplina i jakie są mechanizmy jej działania? W jaki sposób mogę wykorzystać uczenie maszynowe, żeby spojrzeć w nieznanie, wspomóc moją firmę albo po prostu dowiedzieć się, co użytkownicy internetu uważają o moim ulubionym filmie? Wszystkie te informacje (oraz znacznie więcej) zostały omówione w kolejnych rozdziałach niniejszej książki, napisanej przez mojego przyjaciela i kolegę z pracy — Sebastiana Raschkę.

Gdy Sebastian nie próbuje okiełznać mojego szalonego psa, niestrudzenie poświęca swój wolny czas otwartej społeczności uczenia maszynowego. W ciągu kilku ostatnich lat stworzył dziesiątki popularnych kursów dotyczących uczenia maszynowego i wizualizacji danych w Pythonie. Zaprojektował również kilka pakietów Pythona o jawnym kodzie źródłowym — niektóre z nich stały się podstawowymi bibliotekami systemu uczenia maszynowego w środowisku Python.

W związku z olbrzymią wiedzą autora w omawianej dziedzinie jestem przekonany, że jego przemyślenia na temat uczenia maszynowego w Pythonie okażą się bezcenne dla wszystkich użytkowników, bez względu na ich poziom zaawansowania. Polecam tę książkę z całego serca każdej osobie pragnącej w praktyczny sposób zrozumieć koncepcję uczenia maszynowego.

**Dr Randal S. Olson**

Badacz sztucznej inteligencji oraz uczenia maszynowego  
na Uniwersytecie Pensylwanii

# Informacje o autorze

**Sebastian Raschka** jest doktorantem na Michigan State University; opracowuje nowe metody obliczeniowe w dziedzinie biologii statystycznej. Został uznany przez zrzeszenie Analytics Vidhya za najbardziej wpływowego analityka danych w serwisie GitHub. Ma wieloletnie doświadczenie w programowaniu w Pythonie, ponadto poprowadził kilka wykładów na temat praktycznych zastosowań analizy danych i uczenia maszynowego. Rozmowy oraz pisanie publikacji na temat analizy danych, uczenia maszynowego i języka Python zmotywowały autora do napisania książki, która pomoże ludziom we wdrażaniu technik analitycznych bez konieczności dogłębnej znajomości uczenia maszynowego.

Bierze on również udział w różnych projektach open source oraz przyczynia się do wdrażania nowych metod wykorzystywanych w różnych turniejach związanych z uczeniem maszynowym, np. takich jak Kaggle. W wolnym czasie pracuje nad modelami predycyjnymi dyscyplin sportowych, a jeżeli nie siedzi przed monitorem, sam chętnie uprawia sport.

---

Chcę podziękować swoim profesorom, Arunowi Rossowi i Pang-Ning Tanowi, a także wielu innym za inspirację oraz rozpalenie we mnie zainteresowania klasifikacją wzorców, uczeniem maszynowym i pozywaniem danych.

Korzystam z okazji i składam wyrazy wdzięczności wspierającej społeczności Pythona oraz twórcom bezpłatnych pakietów, którzy pomogli mi stworzyć doskonałe środowisko badawcze.

Specjalne podziękowania kieruję do twórców pakietu scikit-learn. Jako jeden z udziałowców tego projektu miałem przyjemność współpracować z cudownymi ludźmi, którzy nie tylko mają olbrzymią wiedzę na temat uczenia maszynowego, lecz również są doskonałymi programistami.

Na koniec dziękuję również Wam, Czytelnikom zainteresowanym tą książką, i mam szczerą nadzieję, że zarażę Was swoim entuzjazmem, dzięki któremu dołączycie do społeczności Pythona i uczenia maszynowego.

---



# Informacje o recenzentach

**Richard Dutton** w wieku ośmiu lat zaczął pisać programy na komputer ZX Spectrum i ta obserwacja doprowadziła go do zagmatwanej mieszanki zajęć na skraju technologii i finansów.

Pracował w firmie Microsoft oraz w korporacji Barclays na stanowisku dyrektora. W polu jego obecnych zainteresowań znajduje się mieszanka Pythona, uczenia maszynowego oraz łańcuchów bloków.

Gdy nie siedzi przed monitorem, można go spotkać na siłowni albo w domu z lampką wina w dłoni, zapatrzonego w swojego iPhone'a. On nazywa to równowagą.

**Dave Julian** jest od przeszło 15 lat konsultantem informatycznym i nauczycielem. Pracował jako technik, kierownik projektu, programista oraz twórca aplikacji sieciowych. Obecnie zajmuje się tworzeniem narzędzia do analizy upraw rolniczych, stanowiącego część strategii zintegrowanego zarządzania szkodnikami w szklarniach. Bardzo interesuje go łączenie technologii z biologią, gdyż jest przekonany, że inteligentne maszyny są w stanie rozwiązać większość naglących problemów współczesnego świata.

**Vahid Mirjalili** uzyskał tytuł doktora inżynierii mechanicznej na Michigan State University, gdzie opracował nowatorską technikę określania struktur białkowych za pomocą dynamicznych symulacji cząsteczkowych. Korzystając z wiedzy na temat statystyki, pozyskiwania danych oraz fizyki, stworzył potężne metody, które pomogły jego zespołowi wygrać dwukrotnie światowej klasy zawody w prognozowaniu i ustalaniu struktur białkowych (CASP) — w latach 2012 i 2014.

W czasie robienia doktoratu dołączył do wydziału informatyki i inżynierii na Michigan State University, gdzie specjalizował się w uczeniu maszynowym. Obecnie zajmuje się projektowaniem nienadzorowanych algorytmów służących do pozyskiwania olbrzymich zbiorów danych.

Jest również zapalonym programistą w środowisku Python i udostępnia implementacje algorytmów skupień na swojej stronie: <http://vahidmirjalili.com/>.

**Hamidreza Sattari** jest zawodowym informatykiem, zaangażowanym w kilka obszarów inżynierii oprogramowania, od programowania po tworzenie architektury, a także zarządzanie projektami. Ma tytuł magistra inżynierii oprogramowania uzyskany na uczelni Heriot-Watt University (Wielka Brytania) oraz licencjata inżynierii elektrycznej (elektronice) zdobyty na Tehran Azad University (Iran). Obecnie interesuje się przetwarzaniem wielogabarytowych danych oraz uczeniem maszynowym. Jest współautorem książki *Spring Web Services 2 Cookbook* i prowadzi własny blog pod adresem <http://justdeveloped-blog.blogspot.com/>.

**Dmytro Taranovsky** jest inżynierem oprogramowania interesującym się środowiskiem Pythona, Linuksem i uczeniem maszynowym. Urodził się w Kijowie, do Stanów Zjednoczonych przemigosł się w 1996 roku. Od najmłodszych lat pasjonował się nauką oraz wiedzą, co udowadniał, wygrywając różne konkursy matematyczne i fizyczne. W 1999 roku dołączył do amerykańskiej drużyny fizyków. W 2005 roku zdobył tytuł magistra matematyki w MIT. Następnie zajął się tworzeniem systemu transformacji tekstu we wspomaganych komputerowo transkrypcjach medycznych (eScription). Pierwotnie korzystał ze środowiska Perl, jednak docenił potęgę i przejrzystość Pythona, dzięki któremu mógł dostosować system do olbrzymich zestawów danych. Następnie pracował jako inżynier oprogramowania i analityk w firmie handlującej algorytmami. Ma również istotny udział w rozwoju matematyki, gdyż stworzył i rozwinał rozszerzenie języka teorii mnogości oraz powiązał je z podstawowymi aksjomatami, tworząc w ten sposób notację konstruktywnej prawdy. Ponadto stworzył system notacji porządkowych, które zaimplementował w Pythonie. Uwielbia czytać, wychodzić na spacery oraz próbować zmieniać świat na lepsze.

# Wstęp

Chyba nie muszę Ci mówić, że uczenie maszynowe jest jedną z najbardziej ekscytujących technologii naszych czasów. Wiele wielkich firm, takich jak Google, Facebook, Apple, Amazon czy IBM, nie bez przyczyny inwestuje mnóstwo pieniędzy w rozwój i zastosowania uczenia maszynowego. Chociaż może się wydawać, że termin „uczenie maszynowe” stanowi obecnie jedynie popularny zwrot, zdecydowanie nie należy go bagateliizować. Ta fascynująca dziedzina wiedzy otwiera przed nami zupełnie nowe możliwości i powoli staje się nieodzowną częścią naszego życia. Mówienie do asystenta głosowego w smartfonie, zalecanie odpowiedniego produktu klientom, zapobieganie oszustwom przy użyciu kart kredytowych, filtrowanie niechcianych wiadomości w poczcie e-mail, wykrywanie i diagnozowanie dolegliwości medycznych — zastosowania można wymieniać w nieskończoność.

Jeżeli chcesz rozpocząć przygodę z uczeniem maszynowym, pragniesz lepiej rozwiązywać problemy, a może nawet zastanawiasz się nad karierą w branży badań nad uczeniem maszynowym, ta książka jest dla Ciebie! Jednak dla osoby poczynającej teoretyczne podstawy uczenia maszynowego mogą okazać się nieco przytłaczające. Na szczęście ostatnio wypuszczono na rynek wiele praktycznych pozycji, które pozwalają zrozumieć koncepcję uczenia maszynowego poprzez przedstawienie potężnych algorytmów. Moim zdaniem stosowanie przykładowych kodów pełni ważną funkcję. Pokazują one omawiane zagadnienia w bezpośrednim, praktycznym ujęciu. Pamiętaj przy tym, że z wielką władzą przychodzi wielka odpowiedzialność! Koncepcje kryjące się za uczeniem maszynowym są zbyt piękne i istotne, aby chować je w „czarnej skrzynce”. Dlatego powziąłem swój własny cel: dostarczyć Ci książkę inną niż wszystkie; opisującą niezbędne szczegóły dotyczące teoretycznych podwalin uczenia maszynowego; oferującą intuicyjne, ale wyczerpujące informacje na temat działania algorytmów uczenia maszynowego, sposobów ich wykorzystania oraz, co najistotniejsze, technik unikania najpowszechniejszych pułapek.

Jeżeli w usłudze Google Scholar wpiszesz „machine learning”, uzyskasz zawrotną liczbę 3 600 000 publikacji. Nie jestem oczywiście w stanie omówić wszystkich najdrobniejszych szczegółów każdego algorytmu stworzonego w ciągu ostatnich 60 lat. Za to wyruszymy w tej książce w fascynującą podróż po najważniejszych zagadnieniach i koncepcjach stanowiących podstawę

uczenia maszynowego. Jeżeli uznasz, że Twój głód wiedzy nie został zaspokojony, możesz zapoznać się z bogatą listą materiałów dokładnie omawiających każdy z opisanych w tej książce przełomów.

Być może masz już opanowane szczegóły uczenia maszynowego — w takim przypadku książka ta pokaże Ci, w jaki sposób przekuć wiedzę teoretyczną w praktykę. Jeżeli już wcześniej korzystałeś z technik uczenia maszynowego i chcesz dokładniej poznać ich teoretyczne podstawy, to trafiłeś w idealne miejsce! Nie martw się, jeśli dziedzina uczenia maszynowego jest dla Ciebie czymś zupełnie nowym; masz tym większy powód, aby czuć podekscytowanie! Obiecuję, że dzięki uczeniu maszynowemu zacznesz inaczej postrzegać kwestię rozwiązywania problemów oraz dowiesz się, jak stawać im czoło poprzez uwolnienie potęgi danych.

Zanim przejdziemy do sedna, pozwól, że odpowiem na najważniejsze chyba pytanie: dlaczego Python? Odpowiedź jest prosta: to bardzo potężne, a jednocześnie przystępne środowisko. Python stał się najpopularniejszym językiem programowania przeznaczonym do analizowania danych naukowych, ponieważ pozwala nam zapomnieć o żmudnej części pisania kodu oraz gwarantuje środowisko, w którym możemy szybko zapisywać swoje pomysły i z łatwością testować koncepcje.

Spoglądając na własne doświadczenia, mogę stwierdzić, że dzięki zgłębieniu tajników uczenia maszynowego stałem się lepszym naukowcem, myślicielem oraz analitykiem. Chcę się w tej książce podzielić z Tobą tą wiedzą. Zdobywamy wiedzę poprzez naukę, podstawę stanowi entuzjazm, a mistrzostwo w danej dziedzinie możemy osiągnąć, wyłącznie trenując. Droga ta czasami bywa wyboista, a niektóre zagadnienia stanowią większe wyzwanie od innych, mam jednak nadzieję, że skorzystasz z nadarzającej się okazji i skoncentrujesz się na ostatecznej nagrodzie. Pamiętaj, że wspólnie podążamy tą ścieżką, a podczas czytania niniejszej książki poszerzysz swój arsenał o wiele potężnych technik, dzięki którym będziesz w stanie rozwiązać najtrudniejsze nawet problemy z danymi.

## Zakres tematyki

W rozdziale 1., „Umożliwianie komputerom uczenia się z danych”, zaznajamiam Cię z głównymi działami uczenia maszynowego, które są stosowane do rozwiązywania różnorodnych problemów. Do tego wyjaśniam podstawowe etapy tworzenia typowego modelu uczenia maszynowego, które będziemy wykorzystywać w kolejnych rozdziałach.

W rozdziale 2., „Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji”, cofamy się do początków uczenia maszynowego i wprowadzam pojęcia binarnego klasyfikatora perceptronu oraz adaptacyjnego neuronu liniowego. Rozdział ten stanowi delikatne wprowadzenie do koncepcji klasyfikacji wzorców i koncentruje się na zależnościach pomiędzy optymalizacją algorytmów i uczeniem maszynowym.

W rozdziale 3., „Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn”, zostały omówione najważniejsze algorytmy klasyfikacji oraz zaprezentowane praktyczne przykłady, w których wykorzystałem jedną z najpopularniejszych i najbardziej zaawansowanych bibliotek uczenia maszynowego — scikit-learn.

W rozdziale 4., „Tworzenie dobrych zbiorów uczących — wstępne przetwarzanie danych”, zajmujemy się rozwiązywaniem najpopularniejszych problemów związanych z nieprzetworzonymi zbiorami danych, takich jak brakujące informacje. Przyglądamy się również kilku metodom identyfikowania najbardziej informatycznych parametrów zbiorów danych. Ponadto wyjaśniam, w jaki sposób przygotowywać zmienne różnych rodzajów jako właściwe dane wejściowe algorytmów uczenia maszynowego.

W rozdziale 5., „Kompresja danych poprzez redukcję wymiarowości”, opisuję podstawowe techniki redukowania cech zbioru danych do mniejszych zestawów przy jednoczesnym zachowaniu większości przydatnych i rozróżnialnych informacji. Wyjaśniam standardowe podejście do redukcji wymiarowości poprzez analizę głównych składowych, a także porównuję je do technik nadzorowanych oraz nieliniowych transformacji.

W rozdziale 6., „Najlepsze metody oceny modelu i strojenie parametryczne”, przytoczone zostają wszelkie zakazy i nakazy dotyczące szacowania wydajności modeli predykcyjnych. Do tego przyglądamy się różnym metrykom pomiarowym oraz metodom precyzyjnej regulacji algorytmów uczenia maszynowego.

W rozdziale 7., „Łączenie różnych modeli w celu uczenia zespołowego”, wprowadzam różne koncepcje wydajnego łączenia wielu algorytmów uczenia się. Dowiesz się, w jaki sposób skonstruować zespoły zaawansowanych algorytmów, dzięki którym można uniknąć słabości pojedynczych algorytmów uczących, co pozwala uzyskiwać dokładniejsze i pewniejsze prognozy.

W rozdziale 8., „Wykorzystywanie uczenia maszynowego w analizie sentymentów”, zostają omówione podstawowe etapy przekształcania danych tekstowych w postać zrozumiałą dla algorytmów uczenia maszynowego, dzięki czemu są one w stanie przewidywać opinie osób na podstawie sposobu pisania.

W rozdziale 9., „Wdrażanie modelu uczenia maszynowego do aplikacji sieciowej”, rozwijam omówiony w poprzednim rozdziale model predykcyjny poprzez wyjaśnienie krok po kroku sposobu tworzenia aplikacji sieciowych wykorzystujących wbudowane modele uczenia maszynowego.

W rozdziale 10., „Przewidywanie ciągłych zmiennych docelowych za pomocą analizy regresywnej”, przyglądamy się podstawowym technikom modelowania liniowych związków pomiędzy zmiennymi docelowymi a zmiennymi odpowiedzi w celu uzyskania ciągłych prognoz. Po omówieniu różnych modeli liniowych przechodzimy do regresji wielomianowej oraz do modeli drzewa.

W rozdziale 11., „Praca z nieoznaczonymi danymi — analiza skupień”, koncentrujemy się na innej dziedzinie uczenia maszynowego — uczeniu nienadzorowanym. Zapoznajemy się z przedstawicielami trzech różnych rodzin algorytmów skupień, które pozwalają wyszukiwać obiekty wykazujące pewien stopień podobieństwa.

W rozdziale 12., „Trenowanie sztucznych sieci neuronowych w rozpoznawaniu obrazu”, rozwijamy omówione w rozdziale 2. pojęcie optymalizacji gradientowej i wykorzystujemy je do stworzenia wielowarstwowej, potężnej sieci neuronowej na podstawie popularnego algorytmu wstępnej propagacji.

W rozdziale 13., „Równoległe przetwarzanie sieci neuronowych za pomocą biblioteki Theano”, wykorzystujemy wiedzę zdobytą w poprzednim rozdziale do wydajniejszego trenowania sieci neuronowych. Koncentrujemy się tutaj na otwartej bibliotece Theano, pozwalającej na korzystanie z wielordzeniowej technologii współczesnych procesorów graficznych.

## Czego będziesz potrzebować?

Aby uruchomić przykładowe kody zamieszczone w tej książce, musisz zainstalować środowisko Python w wersji co najmniej 3.4.3 dla systemów Windows, Linux lub Mac OS X. Będziemy często korzystać z podstawowych bibliotek Pythona do dokonywania obliczeń naukowych — najczęściej będziemy stosować biblioteki: SciPy, NumPy, scikit-learn, matplotlib oraz pandas.

W rozdziale 1., „Umożliwianie komputerom uczenia się z danych”, zamieścilem instrukcje i przydatne porady na temat konfigurowania środowiska Python oraz wymienionych bibliotek. W kilku rozdziałach wymagane będzie zainstalowanie dodatkowych bibliotek: biblioteki NLTK do przetwarzania języka naturalnego (rozdział 8., „Wykorzystywanie uczenia maszynowego w analizie sentymentów”), platformy sieciowej Flask (rozdział 9., „Wdrażanie modelu uczenia maszynowego do aplikacji sieciowej”), biblioteki seaborn do wizualizacji danych statystycznych (rozdział 10., „Przewidywanie ciągłych zmiennych docelowych za pomocą analizy regresywnej”) oraz biblioteki Theano do wydajnego trenowania sieci neuronowych za pomocą procesorów graficznych (rozdział 13., „Równoległe trenowanie sieci neuronowych za pomocą biblioteki Theano”).

## Dla kogo jest przeznaczona ta książka?

Jeżeli chcesz się dowiedzieć, w jaki sposób wykorzystać Pythona do uzyskiwania odpowiedzi na najbardziej palące pytania dotyczące danych, to jest książka dla Ciebie — bez względu na to, czy dopiero zaczynasz poznawać dziedzinę uczenia maszynowego lub pragniesz rozszerzyć swoją wiedzę, pozycja ta jest podstawowa i obowiązkowa.

# Konwencje

Zawarłem w tej książce wiele stylów tekstowych, dzięki którym na pierwszy rzut rozpoznasz różne rodzaje informacji. Poniżej przedstawiam te style wraz z wyjaśnieniem ich znaczenia.

Fragmenty kodu w tekście, nazwy tabel bazodanowych, nazwy folderów i plików, rozszerzenia plików, nazwy ścieżek, adresy URL, dane wprowadzane przez użytkownika oraz identyfikatory na Twitterze zostały oznaczone w następujący sposób: „Uprzednio zainstalowane pakiety mogą zostać zaktualizowane za pomocą flagi `--upgrade`”.

Fragment kodu wygląda tak:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np.

>>> y = df.iloc[0:100, 4].values
>>> y = np.where(y == 'Iris-setosa', -1, 1)
>>> X = df.iloc[0:100, [0, 2]].values
>>> plt.scatter(X[:50, 0], X[:50, 1],
...                 color='red', marker='x', label='Setosa')
>>> plt.scatter(X[50:100, 0], X[50:100, 1],
...                 color='blue', marker='o', label='Versicolor')
>>> plt.xlabel('Długość działki')
>>> plt.ylabel('Długość płatka')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Wszelkie dane wprowadzane do wiersza poleceń lub wyświetlane odpowiedzi prezentowane są następująco:

```
> dot -Tpng drzewo.dot -o drzewo.png
```

**Nowe pojęcia i istotne słowa** są pogrubione. Wyrazy, które pojawiają się na ekranie, np. w menu lub oknach dialogowych, są przedstawiane w tekście następująco: „Po kliknięciu ikony *Otwórz ustawienia* na górze okna uzyskujemy dostęp do panelu sterowania”.

# Uwagi czytelników

Zawsze z radością poznajemy opinie naszych czytelników. Podziel się z nami wrażeniami po przeczytaniu tej książki — co Ci się w niej podobało, a co byś zmienił. Twoje uwagi są dla nas bardzo ważne, ponieważ dzięki nim jesteśmy w stanie publikować maksymalnie przydatne pozycje.

Swoje uwagi wystarczy przesłać na adres mailowy: [helion@helion.pl](mailto:helion@helion.pl), a w temacie podać nazwę książki.

Jeżeli masz duże doświadczenie w danej dziedzinie i marzy Ci się napisanie własnej książki lub współautorstwo, zatrzymaj pod adres [https://helion.pl/autelion/zostan\\_autorem.cgi](https://helion.pl/autelion/zostan_autorem.cgi).

## Obsługa klienta

Dumnym posiadaczom niniejszej książki oferujemy kilka rozwiązań pozwalających na maksymalne wykorzystanie drzemiącego w niej potencjału.

## Materiały dodatkowe

Wszelkie pliki materiałów dodatkowych stworzone na potrzeby tej książki możesz pobrać ze strony <ftp://ftp.helion.pl/przyklady/pythum.zip>.

## Errata

Zawsze staramy się zapewnić jak najrzetelniejszą zawartość merytoryczną publikowanych książek, jednak każdemu zdarzają się pomyłki. Jeżeli natrafisz w którejś z naszych książek na błąd — bez znaczenia, czy będzie to literówka lub błędny zapis w kodzie — będziemy wdzięczni za jego zgłoszenie. W ten sposób zaoszczędzisz pozostałym Czytelnikom frustracji, a nam pomóżesz w wydaniu poprawionych przyszłych wersji danej pozycji. Prosimy o zgłoszanie wszelkich errat pod adresem <http://helion.pl/user/erraty/>, gdzie możesz wybrać tytuł książki oraz wpisać szczegółowe informacje na temat znalezionej błędu. Po zweryfikowaniu danych Twoje zgłoszenie zostanie zaakceptowane, a errata zostanie umieszczona na stronie wydawnictwa lub dodana do listy znanych błędów znalezionych w omawianej pozycji.

Aby przejrzeć listę wszystkich błędów zgłoszonych dla danej książki, wpisz jej nazwę w wyszukiwarce na stronie wydawnictwa Helion, a następnie kliknij odnośnik *Erraty*.

## Piractwo

Nielegalne rozpowszechnianie materiałów chronionych prawami autorskimi stanowi odwieczny problem w erze cyfryzacji. W wydawnictwie Helion podchodzimy bardzo poważnie do kwestii ochrony praw autorskich i licencyjnych. Jeżeli znajdziesz w internecie występujące w jakiejkolwiek postaci nielegalne kopie naszych książek, będziemy bardzo wdzięczni za natychmiastowe przesłanie odnośnika do nich lub chociaż nazwy witryny, z której można je pobrać, dzięki czemu będziemy mogli podjąć odpowiednie kroki prawne.

Zauważone naruszenie praw autorskich możesz zgłosić, używając formularza dostępnego pod adresem <http://helion.pl/piracy.phtml>.

Doceniamy każdą pomoc w ochronie naszych autorów oraz możliwości dostarczania Ci najwyższej jakości materiałów.

## Pytania

Jeżeli masz problemy z dowolnym aspektem niniejszej książki, skontaktuj się z nami, wysyłając pytanie na adres [helion@helion.pl](mailto:helion@helion.pl), a my postaramy się odpowiedzieć na nie w najlepszy możliwy sposób.



# Umożliwianie komputerom uczenia się z danych

Moim zdaniem **uczenie maszynowe** (ang. *machine learning*) — dział zajmujący się teorią i praktycznym zastosowaniem algorytmów analizujących dane — stanowi najciekawszą dziedzinę informatyki! Żyjemy w czasach przetwarzania olbrzymiej ilości informacji; za pomocą samouczących się algorytmów będących częścią uczenia maszynowego informacje te są przekształcane w rzeczywistą wiedzę. Dzięki licznym i potężnym bibliotekom o jawnym kodzie źródłowym, które powstały w ostatnich latach, prawdopodobnie teraz jest najlepszy czas, aby zainteresować się uczeniem maszynowym i nauczyć się wykorzystywać potężne algorytmy do wykrywania wzorców w przetwarzanych danych oraz prognozować przyszłe zdarzenia.

W tym rozdziale poznamy podstawowe pojęcia związane z uczeniem maszynowym oraz zdefiniujemy jego różne rodzaje. Dzięki wprowadzeniu podstawowej terminologii wspólnie położymy fundament pod skuteczne wykorzystywanie technik uczenia maszynowego w rozwiązywaniu praktycznych problemów.

Przyjrzymy się teraz następującym zagadnieniom:

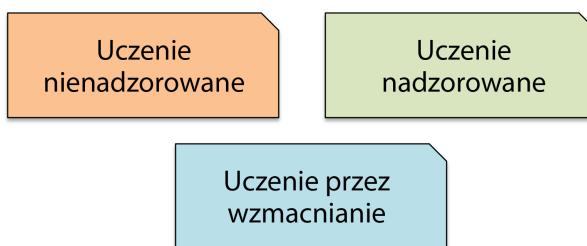
- ogólne pojęcia dotyczące uczenia maszynowego,
- trzy rodzaje uczenia się oraz podstawowa terminologia,
- główne elementy wykorzystywane w skutecznym projektowaniu systemów uczenia maszynowego,
- instalacja oraz konfiguracja środowiska Python pod kątem analizowania danych i uczenia maszynowego.

# Tworzenie inteligentnych maszyn służących do przekształcania danych w wiedzę

W erze współczesnej technologii istnieje pewien zasób, którego mamy pod dostatkiem: mianowicie olbrzymie ilości ustrukturyzowanych i nieustrukturyzowanych danych. W drugiej połowie XX wieku uczenie maszynowe wyewoluowało z badań nad **sztuczną inteligencją**, w których projektowano samouczące się algorytmy, zdolne do pozyskiwania wiedzy z informacji oraz tworzenia na ich podstawie prognoz. Dzięki uczeniu maszynowemu nie trzeba zatrudniać ludzi do ręcznego określania reguł oraz tworzenia modeli poprzez analizowanie olbrzymich pokładów danych; omawiana dziedzina wiedzy oferuje efektywniejsze rozwiążanie polegające na stopniowym poprawianiu skuteczności modeli predykcyjnych oraz podejmowaniu decyzji na podstawie analizowanych danych. Uczenie maszynowe nie tylko staje się coraz istotniejszą częścią nauk informatycznych, lecz również odgrywa coraz większą rolę w naszym codziennym życiu. Dzięki metodom opracowanym pod kątem uczenia maszynowego możemy cieszyć się zaawansowanymi filtrami antyspamowymi w poczcie e-mail, wygodnym oprogramowaniem do rozpoznawania mowy i tekstu, rzetelnymi silnikami wyszukiwarek internetowych, wymagającymi programami szachowymi, a także wkrótce (miejmy nadzieję) bezpiecznymi i wydajnymi pojazdami samojezdnymi.

## Trzy różne rodzaje uczenia maszynowego

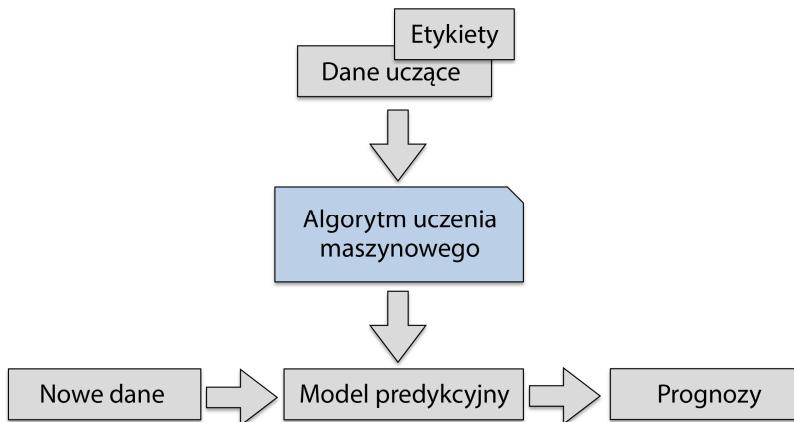
W tym podrozdziale zapoznamy się z trzema odmianami uczenia maszynowego (rysunek 1.1): **uczeniem nadzorowanym**, **uczeniem nienadzorowanym** oraz **uczeniem przez wzmacnianie**. Poznamy podstawowe różnice pomiędzy wspomnianymi typami nauki oraz, korzystając z odpowiednich przykładów, nauczymy się intuicyjnie rozpoznawać, które rodzaje uczenia najlepiej nadają się do rozwiązywania określonych kategorii problemów.



Rysunek 1.1. Podstawowe typy uczenia maszynowego

# Prognozowanie przyszłości za pomocą uczenia nadzorowanego

Głównym celem uczenia nadzorowanego (ang. *supervised learning*; rysunek 1.2) jest uczenie modelu za pomocą oznakowanych **danych uczących** (ang. *training data*), co pozwala przewidywać niewidoczne lub wygenerowane w przyszłości informacje. W tym przypadku człon **nadzorowane** odnosi się do zestawu próbek, w których pożądane sygnały wyjściowe (etykiety) są znane.



Rysunek 1.2. Ogólny schemat uczenia nadzorowanego

Za przykład weźmy filtr antyspamowy: możemy trenować dany model, stosując algorytm nadzorowanego uczenia maszynowego wobec treści oznakowanych wiadomości e-mail (poprawnie oznaczonych jak spam lub wartościowe wiadomości), dzięki czemu system jest w stanie przewidywać, czy przychodząca korespondencja zalicza się do jednej z tych dwóch kategorii. Czynność nadzorowanego uczenia przy użyciu dyskretnych **etykiet klas**, zaprezentowana na powyższym przykładzie, nazywana jest również czynnością **klasyfikacji**. Kolejną podkategorią uczenia nadzorowanego jest **regresja**, w której sygnał wyjściowy przyjmuje wartości ciągłe.

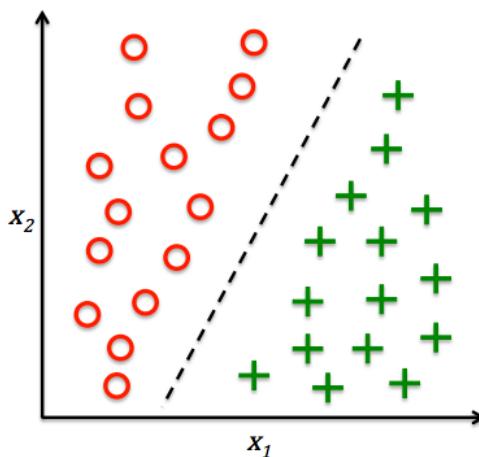
## Klasyfikacja — przewidywanie etykiet klas

Klasyfikacja stanowi podkategorię uczenia nadzorowanego służącą do przewidywania etykiet klas w nowych wystąpieniach na podstawie dotychczasowych obserwacji. Etykiety klas to dyskretne, nieuporządkowane wartości, które określają **przynależność** poszczególnych instancji do wyznaczonych grup. Wspomniany wcześniej przykład filtru antyspamowego jest typowym reprezentantem **klasyfikacji binarnej**, w której algorytm uczenia maszynowego uczy się zestawu reguł w celu rozróżniania dwóch możliwych klas: spamu oraz użytecznych wiadomości e-mail.

Jednak zbiór etykiet klas wcale nie musi mieć charakteru binarnego. Model predykcyjny wyuczony za pomocą algorytmu uczenia nadzorowanego może przydzielić dowolną, zaprezentowaną

w zestawie danych uczących etykietę klas do nowego, nieoznakowanego wystąpienia. Klasycznym przykładem takiej **klasyfikacji wieloklasowej** jest rozpoznawanie odręcznego pisma. W tym przypadku możemy stworzyć zestaw danych uczących składający się z wielu próbek odręcznego pisma każdej litery alfabetu. Jeśli użytkownik wprowadzi do urządzenia jakąś odręcznie napisaną literę, nasz model predyktacyjny będzie w stanie rozpoznać ten znak z określona dokładnością. Jeżeli jednak nie doliczymy symboli cyfr arabskich (0 – 9) do danych uczących, to nasz system nie będzie ich rozpoznawał.

Na rysunku 1.3 zaprezentowałem koncepcję klasyfikacji binarnej, w której wykorzystano 30 próbek uczących: 15 próbek zostało oznaczonych jako **klasa negatywna** (kółka), a pozostałym próbkom przydzieliłem etykiety **klasy pozytywnej** (krzyżyki). W takiej sytuacji zbiór danych jest dwuwymiarowy, co oznacza, że każda próbka zawiera dwie wartości:  $x_1$  oraz  $x_2$ . Teraz wykorzystujemy algorytm nadzorowanego uczenia maszynowego do nauki reguły — granicy decyzyjnej symbolizowanej przez czarną, przerywaną linię — która rozdziela te dwie klasy oraz klasyfikuje analizowane dane do jednej z tych dwóch kategorii w zależności od wartości parametrów  $x_1$  i  $x_2$ .



Rysunek 1.3. Przykład klasyfikacji binarnej

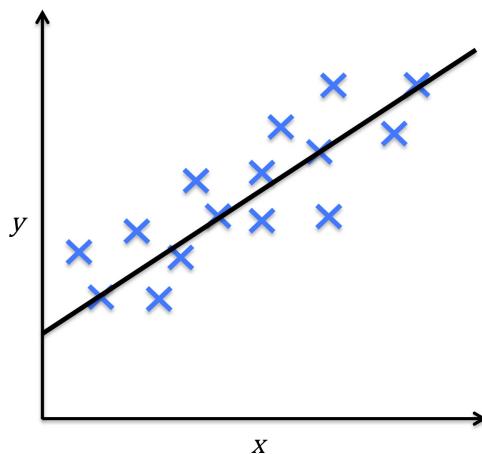
## Regresja dla przewidywania wyników ciągłych

Z poprzedniego podrozdziału dowiedzieliśmy się, że klasyfikacja służy do przydzielania kategorzowanych, nieuporządkowanych etykiet do wystąpień. Drugim rodzajem uczenia nadzorowanego jest prognozowanie wyników ciągłych, czyli tzw. **analiza regresji**. W tym modelu mamy dane zmienne **objaśniające** (prognozujące) oraz ciągłą zmienną **objaśnianą** (prognozowaną), natomiast zadaniem naszym zadaniem jest odkrycie relacji pomiędzy tymi zmiennymi, co pozwoli przewidywać przyszłe wyniki.

Załóżmy, że interesuje nas prognozowanie wyników egzaminów z matematyki naszych studentów. Jeżeli istnieje związek pomiędzy czasem przeznaczonym na naukę a ocenami, możemy wykorzystać te informacje jako zestaw danych uczących do trenowania modelu do przewidywania ocen przyszłych studentów planujących zdawać testy matematyczne.

Pojęcie regresji zostało ukute przez Francisza Galtona w artykule *Regression Towards Mediocrity in Hereditary Stature* (regresja w kierunku przeciętności w dziedziczeniu postury) z 1886 roku. Autor opisał zjawisko biologiczne, zgodnie z którym zróżnicowanie wzrostu w populacji nie ulega zwiększeniu wraz z upływem czasu. Zaobserwował, że wzrost rodziców nie jest przekazywany dzieciom, lecz wzrost potomków zmierza ku średnim wartościom populacji.

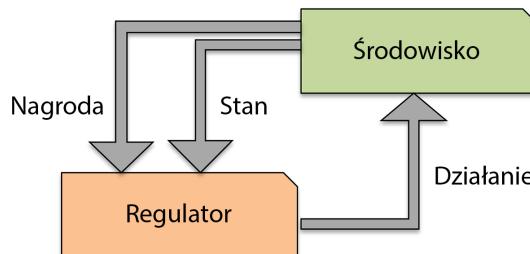
Na rysunku 1.4 zaprezentowana została koncepcja **regresji liniowej**. Mając zmienną objaśniającą  $x$  i zmienną objaśnianą  $y$ , wyznaczamy dla przykładowych danych prostą przebiegającą w jak najmniejszej odległości od zestawu próbek — uzyskujemy ją najczęściej metodą najmniejszych kwadratów. Możemy teraz wykorzystać punkt przecięcia tej prostej z osiami współrzędnych oraz jej nachylenie do przewidywania wyników pochodzących z nowych danych.



Rysunek 1.4. Przykład regresji liniowej

## Rozwiązywanie problemów interaktywnych za pomocą uczenia przez wzmacnianie

Kolejnym rodzajem uczenia maszynowego jest uczenie przez wzmacnianie (ang. *reinforcement learning*). W tym przypadku celem jest utworzenie systemu (**regulatora, agenta**), który poprawia własną skuteczność na podstawie interakcji ze **środowiskiem**. Informacje na temat bieżącego stanu środowiska zazwyczaj zawierają także tzw. sygnał **nagrody**, dlatego możemy uznać uczenie przez wzmacnianie jako model powiązany z uczeniem **nadzorowanym**. Jednak w przypadku uczenia przez wzmacnianie sprzężeniem tym nie są poprawne, wzorcowe etykiety lub wartości, lecz wartość skuteczności pomiaru działania przez **funkcję nagrody**. Poprzez oddziaływanie ze środowiskiem regulator może wykorzystywać uczenie przez wzmacnianie do treningu szeregu działań dążących do maksymalizowania nagrody metodą prób i błędów lub rozważnego planowania (rysunek 1.5).



Rysunek 1.5. Oddziaływanie w modelu uczenia przez wzmacnianie

Popularnym przykładem uczenia przez wzmacnianie jest silnik aplikacji szachowej. Regulator wybiera kolejne ruchy figur szachowych na podstawie stanu szachownicy (środowiska), a nagrodę można zdefiniować jako **zwycięstwo** lub **porażkę** na koniec rozgrywki.

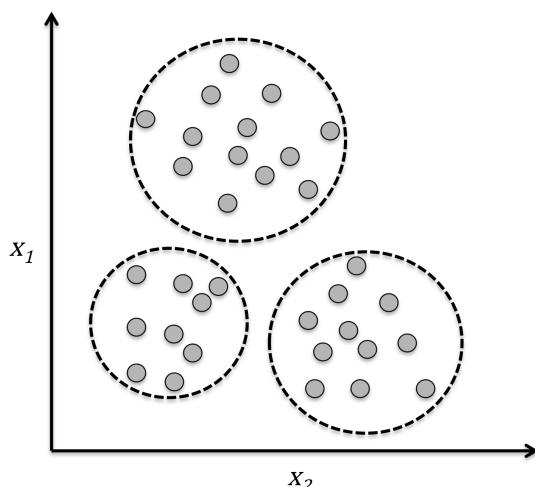
## Odkrywanie ukrytych struktur za pomocą uczenia nienadzorowanego

W przypadku uczenia nadzorowanego znamy właściwą odpowiedź jeszcze przed rozpoczęciem trenowania danego modelu, z kolei w uczeniu przez wzmacnianie wykorzystujemy regulator do definiowania wartości **nagród** dla poszczególnych działań. Natomiast korzystając z technik uczenia nienadzorowanego (ang. *unsupervised learning*), mamy do czynienia z nieoznaczonymi danymi lub danymi o **nieznanej strukturze**. Dzięki modelom uczenia nienadzorowanego jesteśmy w stanie poznawać strukturę przetwarzanych danych i uzyskiwać użyteczne informacje bez stosowania znanej zmiennej wyjściowej lub funkcji nagrody.

### Wyznaczanie podzbiorów za pomocą grupowania

**Grupowaniem (klasteryzacją, analizą skupień; ang. *clustering*)** nazywamy technikę badawczą analizy danych pozwalającą na organizowanie zestawów informacji w sensowne podzbiory (**klaster, grupy, skupienia**) bez uprzedniej wiedzy na temat przydziału grupowego poszczególnych danych. Każdy klaster powstający w wyniku analizy definiuje zbiór obiektów wykazujących między sobą pewne podobieństwa i odróżniających się od elementów umieszczonych w pozostałych grupach, dlatego grupowanie czasami jest nazywane „nienadzorowaną klasyfikacją”. Ta technika nadaje się znakomicie do strukturyzowania informacji oraz wyznaczania istotnych powiązań pomiędzy danymi; np. pozwala sprzedawcom odkrywać grupy klientów według ich zainteresowań, co jest wykorzystywane do tworzenia oddzielnych programów marketingowych.

Na rysunku 1.6 pokazuję, w jaki sposób można wykorzystać klasteryzację do zorganizowania danych w trzy oddzielne grupy na podstawie podobieństw cech  $x_1$  i  $x_2$ .



Rysunek 1.6. Wyznaczanie osobnych grup za pomocą klasteryzacji

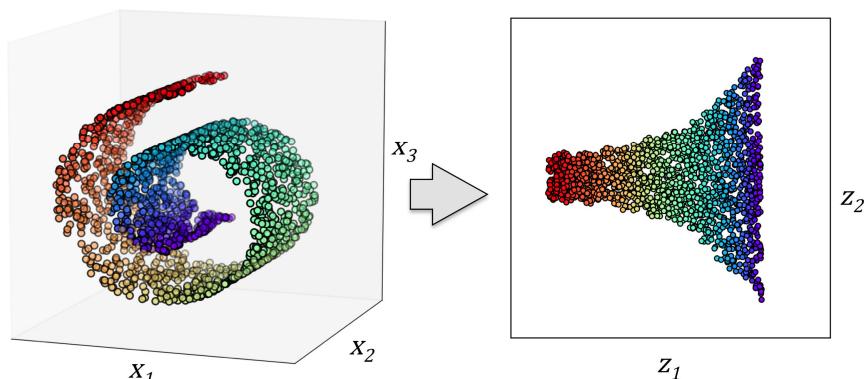
### Redukowanie wymiarowości w celu kompresji danych

Kolejną dziedziną uczenia nienadzorowanego jest **redukja wymiarowości** (ang. *dimensionality reduction*). Często pracujemy z danymi wielowymiarowymi — każda obserwacja daje nam dużą liczbę wartości pomiarowych — co stanowi wyzwanie w przypadku ograniczonej pojemności nośników danych oraz skuteczności obliczeniowej algorytmów uczenia maszynowego. Nienadzorowana redukcja wymiarowości jest stosowana powszechnie we wstępny przetwarzaniu cech w celu wykluczenia szumu z danych — który może zmniejszać skuteczność predykcji niektórych algorytmów — a do tego kompresuje dane do podprzestrzeni o mniejszej liczbie wymiarów przy zachowaniu większości istotnej informacji.

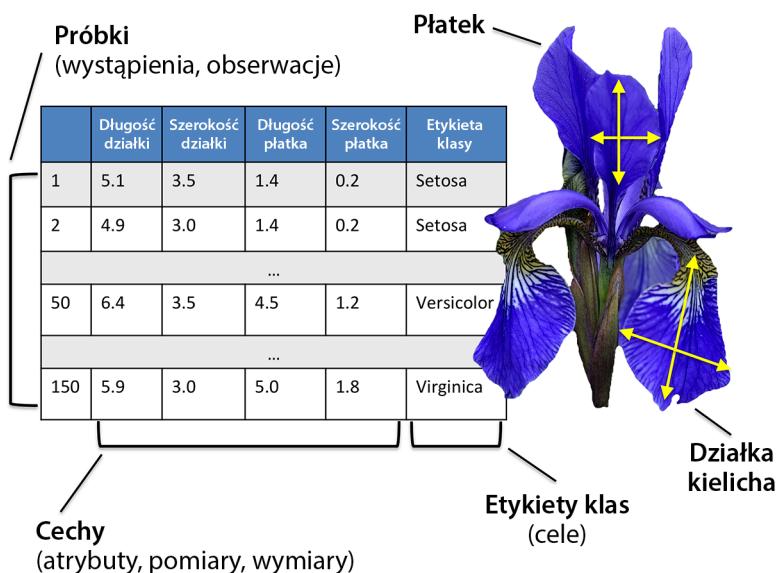
Czasami redukowanie wymiarowości przydaje się również do wizualizacji danych — np. zbiór cech wielowymiarowych można rzutować na jedno-, dwu- lub trójwymiarowe przestrzenie cech w celu wyświetlenia go w formie dwu- lub trójwymiarowego wykresu punktowego albo histogramu. Rysunek 1.7 zawiera przykład zastosowania nieliniowej redukcji wymiarowości do skompresowania wykresu trójwymiarowego do nowej dwuwymiarowej podprzestrzeni cech.

## Wprowadzenie do podstawowej terminologii i notacji

Po omówieniu trzech ogólnych kategorii uczenia maszynowego — nadzorowanego, nienadzorowanego oraz przez wzmacnianie — przeszędł czas na zapoznanie się z podstawową terminologią, która będzie wykorzystywana w kolejnych rozdziałach. Na rysunku 1.8 widzimy tabelę stanowiącą fragment zestawu danych **Iris**, będącego klasycznym przykładem w dziedzinie



Rysunek 1.7. Przykład redukowania wymiarowości



Rysunek 1.8. Fragment zestawu danych Iris

uczenia maszynowego. Na zbiór tych danych składają się wyniki pomiarów 150 kwiatów kosaćca z trzech różnych gatunków: *Setosa* (kosaciec szczecinkowy), *Versicolor* (kosaciec różnobarwny) oraz *Virginica* (kosaciec wirgiński). Każdy kwiat reprezentuje tu oddzielny wiersz w zbiorze danych, natomiast wyniki pomiarów (w centymetrach) są przechowywane w kolumnach, zwanych także zbiorami cech.

Aby zachować prostotę i czytelność notacji oraz implementacji, wprowadzimy pewne podstawy algebry liniowej. W następnych rozdziałach będę opisywać dane w notacji macierzowej oraz wektorowej. Wykorzystam popularną konwencję, zgodnie z którą każda próbka jest reprezentowana jako osobny wiersz w macierzy cech  $X$ , gdzie poszczególne cechy są przechowywane w oddzielnych kolumnach.

Zestaw danych Iris, składający się ze 150 próbek i z 4 cech, możemy zapisać jako macierz o rozmiarze  $150 \times 4$  lub  $X \in \mathbb{R}^{150 \times 4}$ :

$$\begin{bmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & x_4^{(1)} \\ x_1^{(2)} & x_2^{(2)} & x_3^{(2)} & x_4^{(2)} \\ \vdots & \vdots & \vdots & \vdots \\ x_1^{(150)} & x_2^{(150)} & x_3^{(150)} & x_4^{(150)} \end{bmatrix}$$

W dalszej części książki będziemy wykorzystywać indeks górnny ( $i$ ) do określenia  $i$ -tej próbki uczącej, a indeks dolny ( $j$ ) będzie odnosił się do  $j$ -tego wymiaru zbioru uczącego.

Pogrubionymi małymi literami będziemy oznaczać wektory: ( $x \in \mathbb{R}^{n \times 1}$ ), z kolei pogrubionymi dużymi literami będziemy określać macierze: ( $X \in \mathbb{R}^{n \times m}$ ). Pojedyncze elementy wektora lub macierzy definiujemy literami oznaczonymi kursywą (odpowiednio:  $x^{(n)}$  lub  $x_{(m)}^{(n)}$ ).

Na przykład element  $x_1^{150}$  oznacza pierwszy wymiar (czyli długość działki) próbki numer 150. W ten sposób każdy wiersz w tej macierzy cech reprezentuje jedno wystąpienie kwiatu, które można zapisać w postaci czteroelementowego wektora  $x^{(i)} \in \mathbb{R}^{1 \times 4}$ .  $x^{(i)} = \begin{bmatrix} x_1^{(i)} & x_2^{(i)} & x_3^{(i)} & x_4^{(i)} \end{bmatrix}$

Każdy wymiar cechy jest 150-elementowym wektorem kolumnowym  $x_j \in \mathbb{R}^{150 \times 1}$ , np.:

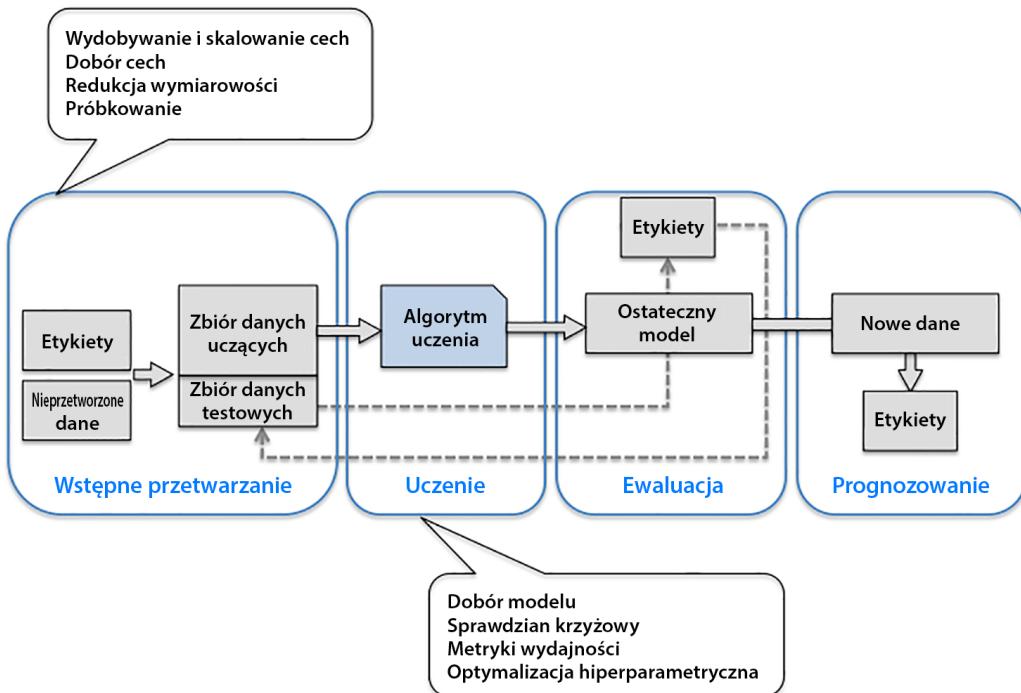
$$x_j = \begin{bmatrix} x_j^{(1)} \\ x_j^{(2)} \\ \vdots \\ x_j^{(150)} \end{bmatrix}$$

W podobny sposób przechowujemy zmienne docelowe (tutaj: etykiety klas), jako 150-elementowy wektor

$$\text{kolumnowy } y = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(150)} \end{bmatrix} (y \in \{\text{Setosa, Versicolor, Virginica}\}).$$

## Strategia tworzenia systemów uczenia maszynowego

W poprzednich podrozdziałach omówiliśmy podstawowe pojęcia uczenia maszynowego oraz poznaliśmy trzy jego rodzaje. Przyjrzymy się teraz innym istotnym elementom systemu uczenia maszynowego towarzyszącym algorytmom uczenia. Na rysunku 1.9 zaprezentowałem schemat typowego przebiegu procesów podczas stosowania uczenia maszynowego w **modelowaniu predykeyjnym**. Zagadnienie to rozwiniemy w dalszej części rozdziału.



Rysunek 1.9. Modelowanie predykcyjne

## Wstępne przetwarzanie — nadawanie danym formy

Rzadko kiedy nieprzetworzone dane mają postać umożliwiającą wykorzystanie optymalnej skuteczności algorytmu uczącego. Z tego powodu **wstępne przetwarzanie** (ang. *preprocessing*) danych stanowi jeden z najważniejszych etapów każdego rodzaju uczenia maszynowego. Weźmy za przykład omówiony we wcześniejszej części rozdziału zestaw danych Iris — danymi nieprzetworzonymi mogą być w tym przypadku zdjęcie kwiatów, z których chcemy wydobyć jak najwięcej sensownych cech. Do takich cech możemy zaliczyć barwę, odcień, intensywność koloru, wysokość rośliny, a także długość i szerokość elementów anatomicznych kwiatu. Wiele algorytmów uczenia maszynowego wymaga również, aby wybrane cechy były prezentowane w jednakowej skali, co często jest osiągane poprzez ich transformację do zakresu  $[0, 1]$  lub do standardowego rozkładu normalnego, ze średnią równą 0 i z wariancją równą 1, jak zostanie to przedstawione w dalszej części książki.

Niektóre wybrane cechy mogą być ze sobą ściśle skorelowane, a przez to w pewnym stopniu nadmiarowe. W takich sytuacjach przydają się techniki redukcji wymiarowości do skompresowania cech w przestrzeń o mniejszej liczbie wymiarów. Zmniejszenie przestrzeni cech pozwala na zaoszczędzenie przestrzeni dyskowej, a także na znaczne przyśpieszenie działania algorytmu uczącego.

Aby się dowiedzieć, czy nasz algorytm uczenia maszynowego działa dobrze nie tylko na zestawie danych uczących, ale i na innych danych, musimy również losowo rozdzielić zbiór danych na osobne zestawy danych uczących i testowych. Zbiór danych uczących służy do trenowania i optymalizowania modelu uczenia maszynowego, natomiast zestaw danych testowych przechowujemy do samego końca procesu i dzięki niemu oceniamy ostateczny model.

## Trenowanie i dobór modelu predykcyjnego

W kolejnych rozdziałach przekonamy się, że stworzono wiele różnorodnych algorytmów uczenia maszynowego przeznaczonych do rozwiązywania różnych kategorii problemów. Istotnym faktem, który stanowi sedno słynnego twierdzenia Davida H. Wolperta (*no free lunch theorem*<sup>1</sup>), jest zrozumienie, że nie możemy zmusić systemu do nauki „za darmo” (D.H. Wolpert, *The Lack of A Priori Distinctions Between Learning Algorithms*, 1996; D.H. Wolpert, W.G. Macready, *No Free Lunch Theorems for Optimization*, 1997). Zgodnie z intuicją możemy powiązać tę koncepcję z popularnym powiedzeniem: *Gdy twoim jedynym narzędziem jest młotek, wszystko zaczyna ci przypominać gwoździe* (A. Maslow, 1966). Przelóżmy to na przykład: każdy algorytm klasyfikacji zawiera integralne założenia i żaden z modeli klasyfikacji nie przeważa nad innymi, jeżeli nie opracujemy założeń dotyczących danego zadania. W praktyce niezbędne okazuje się porównanie przynajmniej kilku różnych algorytmów w celu wytrenowania i doboru najbardziej skutecznego modelu. Zanim jednak będziemy w stanie porównać różne modele, musimy najpierw ustalić metrykę służącą do pomiaru wydajności. Jedną z najpopularniejszych metryk jest dokładność klasyfikacji, którą definiujemy jako stosunek poprawnie sklasyfikowanych wystąpień do nieprawidłowo określonych instancji.

Możesz w tym momencie zadać rozsądne pytanie: *skąd mamy wiedzieć, że dany model dobrze się sprawuje wobec ostatecznego zestawu testowego oraz rzeczywistych danych, skoro nie wykorzystujemy danych testowych na etapie doboru systemu, lecz trzymamy je do momentu ewaluacji ostatecznego modelu?* W celu rozwiązania problemu zdefiniowanego w tym pytaniu można wykorzystać różnorodne techniki sprawdzianu krzyżowego (ang. *cross-validation*), w których zestaw uczący zostaje podzielony na podzbiory uczące i **testowe (walidacyjne)**, służące do oszacowania **wydajności generalizacji** modelu. Nie możemy również oczekwać, że domyślne parametry różnych algorytmów uczenia maszynowego znajdujących się w bibliotekach programowych będą od razu zoptymalizowane pod kątem rozwiązania Twojego określonego problemu. Z tego powodu w dalszych rozdziałach książki będziemy często używać **technik optymalizacji hiperparametrycznej**, które pomogą nam poprawić skuteczność modelu. Hiperparametry to parametry, których nie uzyskano z danych, lecz które stanowią elementy regulacyjne modelu, wykorzystywane do poprawienia jego przewidywań — pojęcie to stanie się znacznie bardziej zrozumiałe w dalszej części książki, gdy przejdziemy do praktycznych przykładów.

<sup>1</sup> W wolnym tłumaczeniu: twierdzenie o nieistnieniu darmowych obiadów — przyp. tłum.

## Ewaluacja modeli i przewidywanie wystąpienia nieznanych danych

Po wybraniu modelu dopasowanego do zestawu danych uczących możemy wykorzystać zbiór danych testowych do oszacowania skuteczności algorytmu wobec nieznanych danych, dzięki czemu będziemy w stanie określić błąd generalizacji. Jeżeli będziemy zadowoleni z jego skuteczności, możemy zacząć używać modelu do przewidywania nowych, przyszłych danych. Należy pamiętać o tym, że parametry wspomnianych wcześniej procedur, takich jak skalowanie cech oraz redukowanie wymiarowości, są określane wyłącznie na podstawie danych uczących, po czym wykorzystywane do przekształcania zbioru testowego oraz wszelkich nowych próbek — skuteczność mierzona jedynie na podstawie wyników z danych testowych może być nazbyt optymistyczna.

## Wykorzystywanie środowiska Python do uczenia maszynowego

Python jest jednym z najpopularniejszych języków programowania stosowanych w analizie danych; dzięki temu zawiera olbrzymią bazę dodatkowych bibliotek stworzonych przez współpracującą społeczność.

Mimo że wydajność języków interpretowanych (do których zalicza się Python) jest w przypadku zadań wymagających dużej mocy obliczeniowej niższa od wydajności języków niższego poziomu, istnieją biblioteki rozszerzeń, takie jak *NumPy* czy *SciPy*, które bazują na implementacjach języków Fortran i C, dzięki czemu uzyskujemy dostęp do szybkich i wektoryzowanych operacji na wielowymiarowych tablicach.

Będziemy najczęściej korzystać z biblioteki *scikit-learn*, która obecnie stanowi jedną z najpopularniejszych i najbardziej przystępnych darmowych bibliotek uczenia maszynowego.

## Instalacja pakietów w Pythonie

Omawiane środowisko programistyczne jest dostępne na wszystkie główne systemy operacyjne — Microsoft Windows, Mac OS X i Linuksa — a zarówno jego instalator, jak i dokumentację znajdziesz na oficjalnej stronie Pythona: <https://www.python.org/>.

Ta książka została napisana pod kątem Pythona w wersji co najmniej 3.4.3, natomiast zalecam korzystanie z najbardziej aktualnej implementacji tego środowiska (w wersji 3), chociaż większość przykładów kodu powinna być również kompatybilna z wersją  $\geq 2.7.10$ . Jeżeli postanowisz uruchamiać zawarte w książce przykłady kodu w wersji 2.7 Pythona, zapoznaj się najpierw z głównymi różnicami pomiędzy wersjami środowiska programowania.

Dobre podsumowanie różnic pomiędzy wersjami 2.7 i 3.4 Pythona (w języku angielskim) znajdziesz pod adresem <https://wiki.python.org/moin/Python2orPython3>.

Dodatkowe, wykorzystywane w dalszej części książki pakiety można zainstalować za pomocą aplikacji *pip*, stanowiącej część standardowej biblioteki Pythona od wersji 3.3. Więcej informacji (w języku angielskim) na temat instalatora *pip* znajdziesz pod adresem <https://docs.python.org/3/installing/index.html>.

Po zainstalowaniu środowiska Python dodajemy kolejne pakiety, wpisując następującą komendę w wierszu poleceń:

```
pip install JakiśPakiet
```

Zainstalowane pakiety możemy zaktualizować za pomocą flagi `--upgrade`:

```
pip install JakiśPakiet --upgrade
```

Bardzo polecaną, alternatywną dystrybucją Pythona przeznaczoną do obliczeń naukowych jest Anaconda stworzona przez firmę Continuum Analytics. Jest to bezpłatna dystrybucja — również w przypadku zastosowań komercyjnych — zawierająca wszystkie niezbędne pakiety wykorzystywane w analizie danych, obliczeniach matematycznych oraz inżynierii, dostępne w przyjaznej, międzyplatformowej postaci. Instalator Anaconda znajdziesz pod adresem <https://www.continuum.io/downloads#py34>, z kolei szybkie wprowadzenie do tego środowiska jest dostępne na stronie <https://conda.io/docs/using/cheatsheet.html>.

Po zainstalowaniu Anacondy możemy instalować nowe pakiety Pythona za pomocą następującego polecenia:

```
conda install JakiśPakiet
```

Zainstalowane pakiety aktualizujemy, korzystając z poniższej komendy:

```
conda update JakiśPakiet
```

Przez większość czasu będziemy korzystać z wielowymiarowych tablic biblioteki *NumPy* do przechowywania i przetwarzania danych. Sporadycznie zastosujemy również bibliotekę *pandas* — nakładkę biblioteki NumPy zapewniającą dodatkowe, zaawansowane narzędzia do manipulowania danymi, dzięki czemu praca z tabelarycznymi informacjami będzie jeszcze wygodniejsza. Aby usprawnić proces nauki i zwizualizować dane ilościowe (pozwala to w maksymalnie intuicyjny sposób zrozumieć wykonywane działania), wprowadzimy również do użytku wysoce konfigurowalną bibliotekę *matplotlib*.

Poniżej wymieniam numery wersji głównych pakietów Pythona, które były wykorzystywane w trakcie pisania niniejszej książki. Upewnij się, że masz na swoim komputerze zainstalowane przynajmniej te wersje (lub nowsze), dzięki czemu przykładowy kod będzie działał we właściwy sposób:

- NumPy 1.9.1
- SciPy 0.14.0

- scikit-learn 0.15.2
- matplotlib 1.4.0
- pandas 0.15.2

## Podsumowanie

W tym rozdziale zapoznaliśmy się bardzo ogólnie z uczeniem maszynowym i zaznajomiliśmy się z podstawowymi koncepcjami, którym poświęcimy znacznie większą uwagę w kolejnych rozdziałach.

Dowiedzieliśmy się, że na uczenie nadzorowane składają się dwie ważne dziedziny: klasyfikacja i regresja. Modele klasyfikacji pozwalają nam kategoryzować obiekty do znanych klas, natomiast dzięki analizie regresji jesteśmy w stanie prognozować wyniki ciągłe docelowych zmiennych. Uczenie nienadzorowane nie tylko zapewnia dostęp do przydatnych technik odkrywających struktury nieoznakowanych danych, lecz również pozwala na kompresowanie danych w czasie wstępnego przetwarzania cech.

Przyjrzaliśmy się побieżnie typowej strategii dopasowywania uczenia maszynowego do zadań problemowych, która stanowi dla nas podstawę do głębszych przemyśleń oraz ukazywania przykładów w dalszej części książki. Na koniec zaś przygotowaliśmy środowisko Pythona i zainstalowaliśmy oraz zaktualizowaliśmy wszystkie pakiety niezbędne do własnoręcznego testowania uczenia maszynowego.

W kolejnym rozdziale zaimplementujemy jeden z najwcześniejszych algorytmów uczenia maszynowego stosowanych w klasyfikacji, co przygotuje nas do rozdziału 3., „Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn”, w którym zapoznamy się z bardziej zaawansowanymi algorytmami dostępnymi w bibliotece o jawnym kodzie źródłowym — scikit-learn. Algorytmy uczenia maszynowego uczą się z danych, dlatego kluczową kwestią jest dostarczanie im użytecznych informacji, zatem w rozdziale 4., „Tworzenie dobrych zbiorów uczących — wstępne przetwarzanie danych”, przyjrzymy się istotnym technikom wstępnego przetwarzania danych. W rozdziale 5., „Kompresja danych poprzez redukcję wymiarowości”, poznamy metody redukowania wymiarowości, które pomogą nam skompresować zbiór danych do mniejszej przestrzeni cech, co może być korzystne dla szybkości obliczeń. Ważnym aspektem tworzenia modeli uczenia maszynowego jest ocena ich skuteczności i oszacowanie ich zdolności predykcji dla nowych, nieznanych danych. W rozdziale 6., „Najlepsze metody oceny modelu i strojenie parametryczne”, nauczymy się wykorzystywać najlepsze rozwiązania umożliwiające strojenie modelu i jego ewaluację. W pewnych przypadkach możemy być niezadowoleni ze skuteczności modelu predykcyjnego pomimo wielu godzin spędzonych na jego dopasowywaniu i testowaniu. Z rozdziału 7., „Łączenie różnych modeli w celu uczenia zespołowego”, dowiemy się, w jaki sposób łączyć różne modele uczenia maszynowego, aby uzyskiwać jeszcze potężniejsze systemy prognozujące.

Po omówieniu wszystkich najważniejszych elementów typowego systemu uczenia maszynowego zaimplementujemy model przewidywania emocji na podstawie rozdziału 8., „Wykorzystywanie uczenia maszynowego w analizie sentymentów”. Natomiast w rozdziale 9., „Wdrażanie modelu uczenia maszynowego do aplikacji sieciowej”, podłączymy ten model pod aplikację sieciową i podzielimy się nim z resztą świata. Następnie w rozdziale 10., „Przewidywanie ciągłych zmiennych docelowych za pomocą analizy regresywnej”, wykorzystamy algorytmy uczenia maszynowego do analizy regresywnej pozwalającej na prognozowanie ciągłych zmiennych wyjściowych, a w rozdziale 11., „Praca z nieoznaczonymi danymi — analiza skupień”, wprowadzimy algorytmy skupień wyszukujące ukryte struktury wśród danych. Ostatnie dwa rozdziały zostały poświęcone sztucznym sieciom neuronowym służącym do rozwiązywania złożonych problemów, takich jak rozpoznawanie mowy i tekstu — czyli jednym z najbardziej fascynujących zagadnień w świecie uczenia maszynowego.



# Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji

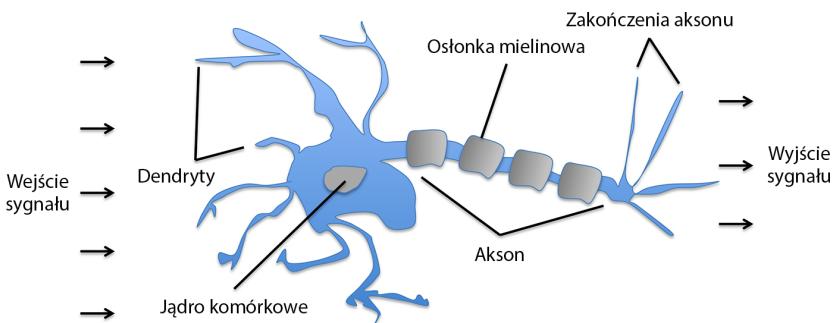
W tym rozdziale wykorzystamy jedne z najwcześniejszych algorytmów klasyfikacyjnych — modele **perceptronu** oraz **adaptacyjnego neuronu liniowego**. Zaczniemy od zaimplementowania krok po kroku perceptronu w środowisku Python oraz uczenia go klasyfikacji różnych odmian kosaćca na podstawie danych z zestawu Iris. W ten sposób lepiej zrozumiemy koncepcję algorytmów klasyfikacyjnych oraz dowiemy się, jak można je skutecznie wdrożyć za pomocą Pythona. Następnie przyjrzymy się podstawom optymalizacji przy użyciu adaptacyjnych neuronów liniowych, gdyż stanowi to wstęp do stosowania bardziej zaawansowanych klasyfikatorów przechowywanych w bibliotece scikit-learn, co zostało omówione w rozdziale 3., „*Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn*”.

Zajmiemy się w tym rozdziale omówieniem następujących zagadnień:

- opis teoretycznych podstaw tworzenia algorytmów uczenia maszynowego,
- wykorzystanie bibliotek pandas, NumPy i matplotlib do wczytywania, przetwarzania i wizualizowania danych,
- implementacja algorytmów liniowej klasyfikacji w środowisku Python.

# Sztuczne neurony — rys historyczny początków uczenia maszynowego

Zanim przejdziemy do dokładnego opisu modelu perceptronu oraz powiązanych z nim algorytmów, cofnijmy się na chwilę do początków dziedziny uczenia maszynowego. Warren McCulloch i Walter Pitts pragnęli zrozumieć mechanizm działania mózgu po to, aby zaprojektować sztuczną inteligencję, i w 1943 roku zaprezentowali pierwszą koncepcję uproszczonego modelu komórki nerwowej, tzw. **neuronu McCullocha-Pittsa** (ang. *McCulloch-Pitts neuron* — MCP; W.S. McCulloch i W. Pitts, *A Logical Calculus of the Ideas Immanent in Nervous Activity*, „The Bulletin of Mathematical Biophysics” 1943, nr 5 (4), s. 115 – 133). Neuronami nazywamy wzajemnie połączone komórki nerwowe w mózgu, które są odpowiedzialne za przetwarzanie oraz przesyłanie sygnałów chemicznych i elektrycznych, co zostało zaprezentowane na rysunku 2.1.



Rysunek 2.1. Model uproszczonego neuronu

McCulloch i Pitts opisali taką komórkę nerwową jako prostą bramkę logiczną zawierającą binarne wyjście; do dendrytów dociera wiele sygnałów, które są integrowane w ciele komórki i, jeżeli energia impulsu przekracza określona wartość graniczną, zostaje wygenerowany sygnał wyjściowy przepuszczany poprzez akson.

Już kilka lat później Frank Rosenblatt na podstawie modelu neuronu MCP opublikował pierwszą koncepcję reguły uczenia perceptronu (F. Rosenblatt, *The Perceptron, a Perceiving and Recognizing Automaton*, „Cornell Aeronautical Laboratory”, 1957). Korzystając z tej reguły, Rosenblatt zaproponował algorytm zdolny do automatycznego uczenia się za pomocą optymalnych współczynników wag, które są przemnażane przez wartości wejściowe, co pozwala określić, czy neuron przeleje dalej sygnał. W kontekście uczenia nadzorowanego i klasyfikacji taki algorytm może być wykorzystywany do przewidywania próbek przynależnych do różnych klas.

W ujęciu matematycznym możemy przedstawić ten problem jako klasyfikację binarną, w której dla uproszczenia odnosimy się do dwóch klas: 1 (klasy pozytywnej) oraz -1 (klasy negatywnej). Następnie definiujemy **funkcję aktywacji  $\phi(z)$** , na którą składa się liniowa kombinacja określonych wartości wejściowych  $x$  oraz powiązanego z nimi wektora wag  $w$ , gdzie  $z$  nosi nazwę całkowitego pobudzenia układu ( $z = w_1x_1 + \dots + w_mx_m$ ):

$$\mathbf{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}, \mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$$

Jeżeli całkowite pobudzenie  $z$  danej próbki  $\mathbf{x}^{(i)}$  jest wyższe od zdefiniowanej wartości progowej  $\theta$ , to przewidujemy, że dany obiekt przynależy do klasy pozytywnej 1, w przeciwnym wypadku — do klasy negatywnej -1. W algorytmie perceptronu funkcja aktywacji  $\varphi(\cdot)$  jest prostą **funkcją skoku jednostkowego**, zwaną czasem również **funkcją skokową Heaviside'a**:

$$\varphi(z) = \begin{cases} 1 & \text{jeśli } z \geq \theta \\ -1 & \text{jeśli } z < \theta \end{cases}$$

Możemy dla uproszczenia przenieść wartość progową  $\theta$  na lewą stronę równania i zdefiniować początkową wagę jako  $w_0 = -\theta$ , a  $x_0 = 1$ , dzięki czemu całkowite pobudzenie  $z$  przybierze prostszą postać  $z = w_0x_0 + w_1x_1 + \dots + w_mx_m = \mathbf{w}^T \mathbf{x}$  przy założeniu, że  $\varphi(z) = \begin{cases} 1 & \text{jeśli } z \geq 0 \\ -1 & \text{jeśli } z < 0 \end{cases}$ .

W kolejnych podrozdziałach będziemy często stosować podstawową notację z zakresu algebry liniowej, np. korzystać ze skróconego zapisu sumy iloczynów wartości  $\mathbf{x}$  i  $\mathbf{w}$  za pomocą **iloczynu skalarnego wektorów**, gdzie indeks górnny  $T$  oznacza **transpozycję** — operację przekształcania wiersza wektora w kolumnę i odwrotnie:

$$z = w_0x_0 + w_1x_1 + \dots + w_mx_m = \sum_{j=0}^m \mathbf{x}_j \mathbf{w}_j = \mathbf{w}^T \mathbf{x}$$

$$\text{Na przykład: } [1 \ 2 \ 3] \times \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = 1 \times 4 + 2 \times 5 + 3 \times 6 = 32.$$

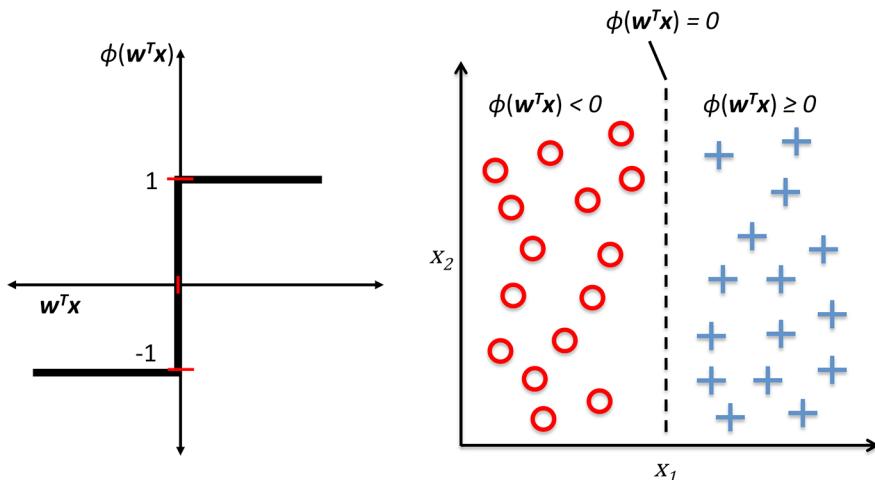
Ponadto operację transponowania można przeprowadzić również wobec macierzy, dzięki czemu następuje w niej zamiana wierszy z kolumnami:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

W tej książce będziemy wykorzystywać jedynie najprostsze pojęcia z algebry liniowej, jeżeli jednak chcesz odświeżyć sobie pamięć, polecam znakomity skrypt *Linear Algebra Review and Reference* autorstwa Zico Koltera, który można bezpłatnie przejrzeć na stronie [http://www.cs.cmu.edu/~zkolter/course/linalg/linalg\\_notes.pdf](http://www.cs.cmu.edu/~zkolter/course/linalg/linalg_notes.pdf)<sup>1</sup>.

<sup>1</sup> Dobrym odpowiednikiem w języku polskim jest skrypt *Matematyka dla studiów inżynierskich. Część 1. Algebra i geometria* autorstwa Stanisława Bialasa, Adama Ćmiela i Andrzeja Fitzkego, dostępny pod adresem <http://winnthbg.bg.agh.edu.pl/skrypty2/0077/bialas.pdf> — przyp. tłum.

Na rysunku 2.2 widzimy, w jaki sposób całkowite pobudzenie  $z = \mathbf{w}^T \mathbf{x}$  zostaje przetworzone na wartości binarne (-1 lub 1) przez funkcję aktywacji perceptronu (wykres po lewej), a także jak może zostać wykorzystane do rozdzielenia dwóch odrębnych liniowo klas (wykres po prawej).



Rysunek 2.2. Zastosowanie całkowitego pobudzenia w uczeniu maszynowym

Podstawowym założeniem w neuronie MCP i modelu perceptronu **progowego** jest wprowadzenie uproszczonego mechanizmu naśladującego działanie pojedynczej komórki nerwowej: albo zostaje ona **aktywniona**, albo nie. Z tego powodu pierwotna reguła uczenia perceptronu autorstwa Rosenblatta jest całkiem nieskomplikowana i można ją opisać następującymi etapami:

1. Wprowadź wagę o wartości 0 lub niewielkich, losowych wartościach.
2. Dla każdej próbki uczącej  $\mathbf{x}^{(i)}$  wykonaj poniższe czynności:
  - a) Oblicz wartość wyjściową  $\hat{y}$ .
  - b) Zaktualizuj wagę.

W tym przypadku wartością wyjściową jest etykieta klasy przewidziana przez wcześniej zdefiniowaną funkcję skoku jednostkowego, a równoczesną aktualizację każdej wagi  $w_j$  w wektorze wag  $\mathbf{w}$  można zapisać w bardziej formalny sposób:

$$w_j := w_j + \Delta w_j$$

Służąca do aktualizowania wagi  $w_j$  wartość  $w_j$  jest wyliczana za pomocą reguły uczenia perceptronu:

$$\Delta w_j = \eta(y^{(i)} - \hat{y}^{(i)})x_j^{(i)}$$

gdzie  $\eta$  jest współczynnikiem uczenia (stałą przyjmującą wartości w zakresie od 0 do 1),  $y^{(i)}$  stanowi rzeczywistą etykietę klas  $i$ -tej próbki uczącej, natomiast  $\hat{y}^{(i)}$  to przewidywana etykieta

klas. Bardzo istotna jest informacja, że wszystkie wagi w wektorze wag są jednocześnie aktualizowane, co oznacza, że nie możemy ponownie przeliczyć wartości  $\hat{y}^{(i)}$ , dopóki nie zaktualizujemy wszystkich wag  $\otimes w_j$ .

Zapis aktualizacji dla dwuwymiarowego zbioru danych możemy zdefiniować następująco:

$$\begin{aligned}\Delta w_0 &= \eta(y^{(i)} - \text{wyjście}^{(i)}) \\ \Delta w_1 &= \eta(y^{(i)} - \text{wyjście}^{(i)})x_1^{(i)} \\ \Delta w_2 &= \eta(y^{(i)} - \text{wyjście}^{(i)})x_2^{(i)}\end{aligned}$$

Zanim zaimplementujemy model perceptronu w Pythonie, przeprowadźmy mały eksperiment myślowy ukazujący piękno prostoty tej reguły uczenia. W dwóch scenariuszach, w których perceptron we właściwy sposób przewiduje etykietę klas, wagi pozostają niezmienione:

$$\begin{aligned}\Delta w_j &= \eta(-1 - -1)x_j^{(i)} = 0 \\ \Delta w_j &= \eta(1 - 1)x_j^{(i)} = 0\end{aligned}$$

Jednak w przypadku nieprawidłowego prognozowania wagi zostają przesunięte w kierunku pozytywnej lub negatywnej klasy docelowej, odpowiednio:

$$\begin{aligned}\Delta w_j &= \eta(1 - -1)x_j^{(i)} = \eta(2)x_j^{(i)} \\ \Delta w_j &= \eta(-1 - 1)x_j^{(i)} = \eta(-2)x_j^{(i)}\end{aligned}$$

Aby lepiej zrozumieć koncepcję mnożnika  $x_j^{(i)}$ , przyjrzyjmy się kolejnemu prostemu przykładowi, w którym:

$$y^{(i)} = +1, \hat{y}^{(i)} = -1, \eta = 1$$

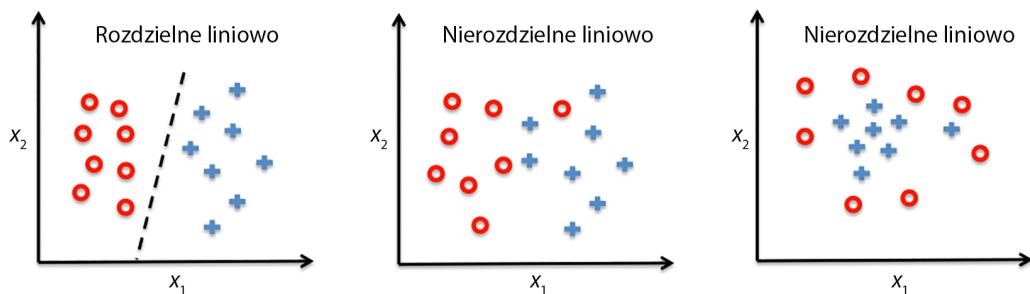
Załóżmy, że  $x_j^{(i)} = 0,5$ , a my tę próbkę nieprawidłowo sklasyfikowaliśmy jako  $-1$ . W takim przypadku zwiększamy wagę o 1, przez co całkowite pobudzenie  $x_j^{(i)} \times w_j$  będzie silniejsze w sytuacji ponownego natrafienia na tę próbkę, dzięki czemu z większym prawdopodobieństwem zostanie przekroczena wartość graniczna funkcji skokowej, a badany obiekt zostanie zaklasyfikowany do klasy  $+1$ :

$$\Delta w_j^{(i)} = (1 - -1)0,5 = (2)0,5 = 1$$

Aktualizacja wagi jest wprost proporcjonalna do wartości  $x_j^{(i)}$ . Założmy, że mamy kolejną próbke,  $x_j^{(i)} = 2$ , która została nieprawidłowo zaklasyfikowana jako  $-1$ . W tej sytuacji przesuwamy granicę decyzyjną w jeszcze większym stopniu po to, aby próbka została następnym razem właściwie zaklasyfikowana:

$$\Delta w_j = (1 - -1)2 = (2)2 = 4$$

Zwróć uwagę, że zbieżność perceptronu zostaje zapewniona jedynie wtedy, gdy dwie klasy są liniowo rozdzielne (rysunek 2.3), a współczynnik uczenia jest wystarczająco mały. Jeżeli nie można oddzielić dwóch klas za pomocą liniowej granicy decyzyjnej, możemy ustalić maksymalną liczbę przebiegów (**epok**) algorytmu z wykorzystaniem danych uczących i (lub) próg tolerancji nieprawidłowych klasyfikacji — w przeciwnym wypadku perceptron wiecznie aktualizowałby wagę.

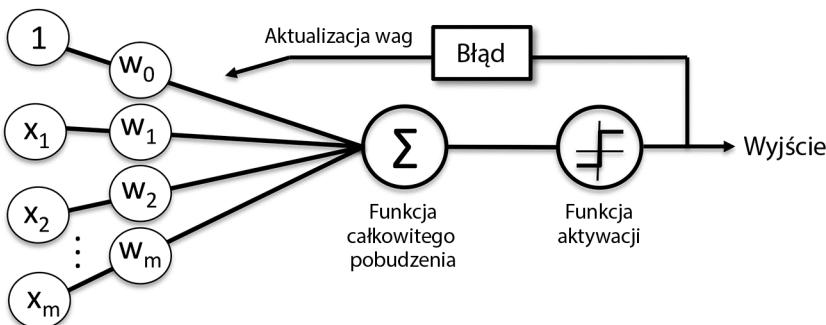


Rysunek 2.3. Rozdzielność liniowa klas

#### Kod źródłowy

Pliki kodu źródłowego są dostępne do pobrania pod adresem <ftp://ftp.helion.pl/przykłady/pythum.zip>.

Zanim przejdziemy do implementacji algorytmów w Pythonie, przyjrzymy się rysunkowi 2.4, stanowiącemu podsumowanie ogólnej koncepcji kryjącej się za modelem perceptronu.



Rysunek 2.4. Ogólny model perceptronu

Na rysunku 2.4 widzimy schemat ukazujący sposób, w jaki perceptron otrzymuje dane wejściowe próbek  $x$  i łączy z wagami  $w$  w celu obliczenia funkcji całkowitego pobudzenia. Wynik jest następnie przekazywany funkcji aktywacji (w tym przypadku funkcji skoku jednostkowego), która generuje wartość binarną  $-1$  lub  $+1$  — prognozowaną etykietę klas danej próbki.

W trakcie fazy uczenia dane wyjściowe są wykorzystywane do obliczenia błędu predykcji i aktualizowania wag.

## Implementacja algorytmu uczenia perceptronu w Pythonie

W poprzednim podrozdziale poznaliśmy mechanizm działania perceptronu Rosenblatta; pojedźmy dalej i zaimplementujmy go w Pythonie, a następnie przetestujmy na zestawie danych Iris, który wprowadziliśmy w rozdziale 1., „Umożliwianie komputerom uczenia się z danych”. Wykorzystamy strategię programowania obiektowego i zdefiniujemy interfejs perceptronu jako konstrukt `Class`, pozwalający na inicjowanie nowych obiektów perceptronu, które będą uczyć się przy użyciu metody `fit`. Z kolei do prognozowania wykorzystamy osobną metodę — `predict`. Zgodnie z konwencją będziemy dodawać podkreślnik do atrybutów, które nie są tworzone w momencie inicjowania obiektu, lecz w chwili wywoływania przez inne metody — np. `self.w_`.

Jeżeli nie znasz jeszcze bibliotek naukowych Pythona lub musisz odświeżyć pamięć, skorzystaj z następujących zasobów (w języku angielskim):

**NumPy.** [http://scipy.github.io/old-wiki/pages/Tentative\\_NumPy\\_Tutorial](http://scipy.github.io/old-wiki/pages/Tentative_NumPy_Tutorial)

**Pandas.** <http://pandas.pydata.org/pandas-docs/stable/tutorials.html>

**Matplotlib.** <http://matplotlib.org/users/beginner.html>

Dla zwiększenia przejrzystości listingów polecam również pobranie ze strony <ftp://ftp.helion.pl/przykłady/pythum.zip> notatników zawierających przykładowy kod. Informacje na temat notatników znajdziesz pod adresem <http://jupyter.readthedocs.io/en/latest/contents.html>.

```
import numpy as np
class Perceptron(object):
    """Klasifikator — perceptron.

    Parametry
    -----
    eta : zmiennoprzecinkowy
        Współczynnik uczenia (w przedziale pomiędzy 0.0 a 1.0).
    n_iter : liczba całkowita
        Liczba przebiegów po zestawach uczących.

    Atrybuty
    -----
    w_ : jednowymiarowa tablica
        Wagi po dopasowaniu.
    errors_ : lista
```

Liczba nieprawidłowych klasyfikacji w każdej epoce.

```
"""
def __init__(self, eta=0.01, n_iter=10):
    self.eta = eta
    self.n_iter = n_iter

def fit(self, X, y):
    """Dopasowanie danych uczących.
```

Parametry

```
-----
X : {tablicopodobny}, wymiary = [n_próbek, n_cech]
    Wektory uczące, gdzie n_próbek
    oznacza liczbę próbek, a
    n_cech — liczbę cech.
y : tablicopodobny, wymiary = [n_próbek]
    Wartości docelowe.
```

Zwrota

-----
self : obiekt

```
"""
self.w_ = np.zeros(1 + X.shape[1])
self.errors_ = []

for _ in range(self.n_iter):
    errors = 0
    for xi, target in zip(X, y):
        update = self.eta * (target - self.predict(xi))
        self.w_[1:] += update * xi
        self.w_[0] += update
        errors += int(update != 0.0)
    self.errors_.append(errors)
return self

def net_input(self, X):
    """Oblicza całkowite pobudzenie"""
    return np.dot(X, self.w_[1:]) + self.w_[0]

def predict(self, X):
    """Zwrota etykietę klas po obliczeniu funkcji skoku jednostkowego"""
    return np.where(self.net_input(X) >= 0.0, 1, -1)
```

Dzięki powyższej implementacji perceptronu możemy teraz inicjować nowe obiekty Perceptron mające wyznaczone współczynnik uczenia eta oraz liczbę epok (przebiegów po danych uczących) — n\_iter. Dzięki metodzie fit wprowadzamy wagę w obiekcie self.w\_ do wektora

zerowego  $\mathbb{R}^{m+1}$ , gdzie  $m$  oznacza liczbę wymiarów (cech) zestawu danych, do której dodajemy 1 w celu uzyskania podstawowej wagi (czyli wartości progowej)<sup>2</sup>.

W bibliotece NumPy indeksowanie jednowymiarowych tablic działa podobnie jak w przypadku list Pythona — za pomocą notacji wykorzystującej nawiasy kwadratowe ([]). W czasie używania tablic dwuwymiarowych pierwszy wskaźnik odnosi się do numeru wiersza, a drugi — numeru kolumny; np. za pomocą oznaczenia  $X[2, 3]$  wybieramy drugi wiersz i trzecią kolumnę w dwuwymiarowej tablicy  $X$ .

Po zainicjowaniu wag metoda `fit` analizuje każdą próbkę zestawu uczącego i aktualizuje wartości wag zgodnie z omówioną wcześniej regułą uczenia perceptronu. Etykiety klas są prognozowane poprzez metodę `predict`, która również zostaje wywołana w metodzie `fit` po to, aby przewidzieć etykietę klas dla aktualizacji wag, jednak może być także wykorzystywana do predykcji etykiet klas nowych danych po wytrenowaniu modelu. Do tego w liście `self.errors` zliczamy liczbę nieprawidłowych klasyfikacji w czasie każdej epoki, dzięki czemu możemy później przeanalizować wydajność perceptronu w procesie nauki. Wykorzystywana w metodzie `net_input` funkcja `np.dot` oblicza iloczyn skalarny wektorów  $w^T x$ .

Zamiast stosować bibliotekę NumPy do obliczenia ilocyznu skalarnego wektorów pomiędzy dwiema tablicami  $a$  i  $b$  za pomocą operacji `a.dot(b)` lub `np.dot(a, b)`, możemy tego dokonać również w „czystym” kodzie Pythona: `sum([i*j for i, j in zip(a, b)])`. Jednakże przewaga struktur pętli `for` występujących w bibliotece NumPy nad dostępnymi w klasycznym Pythonie polega na wektoryzacji operacji arytmetycznych. W procesie wektoryzacji podstawowe operacje arytmetyczne są automatycznie przeprowadzane na każdym elemencie tablicy. Wyznaczyszy operacje arytmetyczne jako sekwencję instrukcji przeprowadzanych wobec tablicy (zamiast tradycyjnego ujęcia, w którym zestaw operacji jest wykonywany oddzielnie na każdym elemencie zbioru), możemy w znacznie skuteczniejszy sposób wykorzystywać architekturę współczesnych procesorów obsługujących **architekturę SIMD** (ang. *Single Instruction, Multiple Data* — pojedyncza instrukcja, wielokrotność danych). Do tego w pakiecie NumPy stosowane są zoptymalizowane biblioteki algebry liniowej, takie jak **Basic Linear Algebra Subprograms (BLAS)** czy **Linear Algebra PACKage (LAPACK)**, napisane w językach C oraz Fortran. Na koniec warto dodać, że biblioteka NumPy gwarantuje zwięzlszy i bardziej intuicyjny zapis kodu podstaw algebry liniowej, np. ilocyznu skalarnego wektorów czy macierzy.

<sup>2</sup> **Errata:** Zwróć uwagę, że współczynnik uczenia  $\eta$  (eta) ma wpływ na wynik klasyfikacji jedynie wtedy, gdy początkowe wartości wag są niezerowe. Jeżeli wszystkie zainicjowane wagi mają wartość 0, zmienia się tylko skala wektora, nie jego kierunek. Aby współczynnik uczenia oddziaływał na wynik klasyfikacji, należy zainicjować początkowe wagi o niezerowej wartości. Poniżej przedstawiam odpowiednie wiersze kodu, które należy zmodyfikować, aby tego dokonać:

```
def __init__(self, eta=0.01, n_iter=50, random_seed=1): # Dodaj random_seed=1
    ...
    self.random_seed = random_seed # Dodaj ten wiersz
def fit(self, X, y):
    ...
    # self.w_ = np.zeros(1 + X.shape[1]) ## usuń ten wiersz
    rgen = np.random.RandomState(self.random_seed) # Dodaj ten wiersz
    self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1]) # Dodaj ten wiersz
```

## Trenowanie modelu perceptronu na zestawie danych Iris

W celu przetestowania naszej implementacji perceptronu, wczytamy ze zbioru danych Iris dwa gatunki kosaćca: *Setosa* i *Versicolor*. Reguła uczenia perceptronu nie ogranicza się wyłącznie do dwóch wymiarów, w celach usprawnienia wizualizacji będziemy rozważać jedynie dwie cechy: **długość działki** (ang. *sepal length*) i **długość płatka** (ang. *petal length*). Poza tym ze względów praktycznych wybraliśmy tylko dwa gatunki kosaćca. Pamiętajmy jednak, że algorytm perceptronu można rozszerzyć do wielowymiarowej klasyfikacji — np. poprzez technikę **OvA** (ang. *One versus All* — jeden przeciw wszystkim).

Technika **OvA**, zwana również czasami **OvR** (ang. *One versus Rest* — jeden przeciw reszcie), jest stosowana do rozszerzania klasyfikacji binarnej na problemy wieloklasowe. Za pomocą tej metody możemy uczyć jeden klasyfikator na klasę, przy czym ta klasa jest traktowana jako klasa pozytywna, a próbki z pozostałych klas są uznawane za obiekty klasy negatywnej. Do sklasyfikowania nowych danych wykorzystalibyśmy nasze  $n$  klasyfikatorów, gdzie  $n$  oznacza liczbę etykiet klas, i przydzieliłybyśmy etykietę klas o największej pewności do danej próbki. W przypadku perceptronu stosowałibyśmy mechanizm OvA do doboru etykiety klas powiązanej z największą wartością bezwzględną całkowitego pobudzenia.

Najpierw wykorzystamy bibliotekę *pandas* do wczytania zbioru danych Iris z bazy **UCI Machine Learning Repository** (z ang. repozytorium uczenia maszynowego na Uniwersytecie Kalifornijskim) do obiektu *DataFrame* oraz wyświetlimy pięć ostatnich linijek za pomocą metody *tail*, aby sprawdzić, czy informacje zostały prawidłowo odczytane (rysunek 2.5):

```
>>> import pandas as pd
>>> df = pd.read_csv('https://archive.ics.uci.edu/ml/'
...                   'machine-learning-databases/iris/iris.data', header=None)
>>> df.tail()
```

	<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	
<b>145</b>	6.7	3.0	5.2	2.3	Iris-virginica	
<b>146</b>	6.3	2.5	5.0	1.9	Iris-virginica	
<b>147</b>	6.5	3.0	5.2	2.0	Iris-virginica	
<b>148</b>	6.2	3.4	5.4	2.3	Iris-virginica	
<b>149</b>	5.9	3.0	5.1	1.8	Iris-virginica	

Rysunek 2.5. Pięć ostatnich wierszy zestawu danych Iris wyświetlonych za pomocą metody *tail*

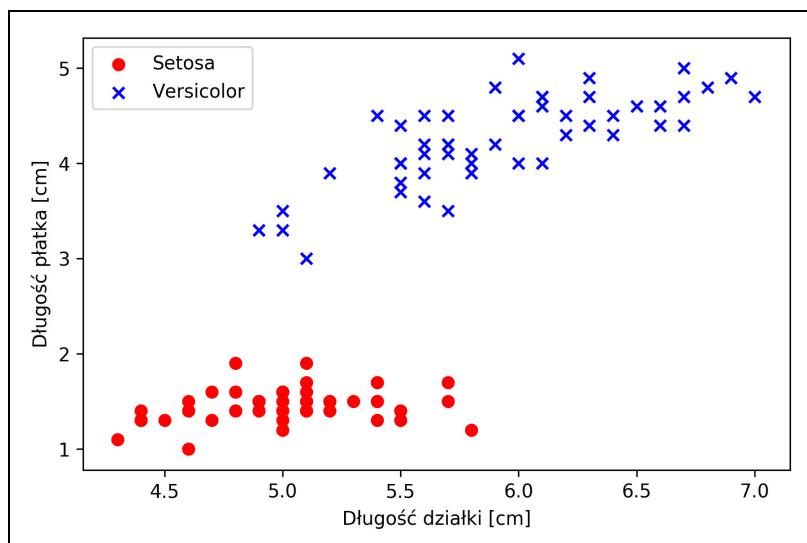
Wydzielamy następnie pierwsze 100 etykiet klas odpowiadających 50 kwiatom z gatunku *Setosa* (*Iris-setosa*) oraz 50 z gatunku *Versicolor* (*Iris-versicolor*) i przekształcamy je w dwie kategorie etykiet symbolizowane liczbami całkowitymi: 1 (*Versicolor*) i -1 (*Setosa*), które przydzielamy do wektora *y*, w którym wartości obiektu *DataFrame* (przynależnego do biblioteki *pandas*) będą

odpowiednikami danych otrzymywanych poprzez pakiet NumPy. W analogiczny sposób ze zbioru 100 próbek uczących wydzielamy pierwszą kolumnę cech (*Długość działki*) i trzecią kolumnę (*Długość płatka*) — uzyskane wartości przydzielamy do macierzy cech  $x$ , którą jesteśmy w stanie wyświetlić jako dwuwymiarowy wykres punktowy:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np

>>> y = df.iloc[0:100, 4].values
>>> y = np.where(y == 'Iris-setosa', -1, 1)
>>> X = df.iloc[0:100, [0, 2]].values
>>> plt.scatter(X[:50, 0], X[:50, 1],
... color='red', marker='o', label='Setosa')
>>> plt.scatter(X[50:100, 0], X[50:100, 1],
... color='blue', marker='x', label='Versicolor')
>>> plt.xlabel('Długość działki [cm]')
>>> plt.ylabel('Długość płatka [cm]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Po uruchomieniu powyższego kodu powinniśmy otrzymać wykres pokazany na rysunku 2.6.

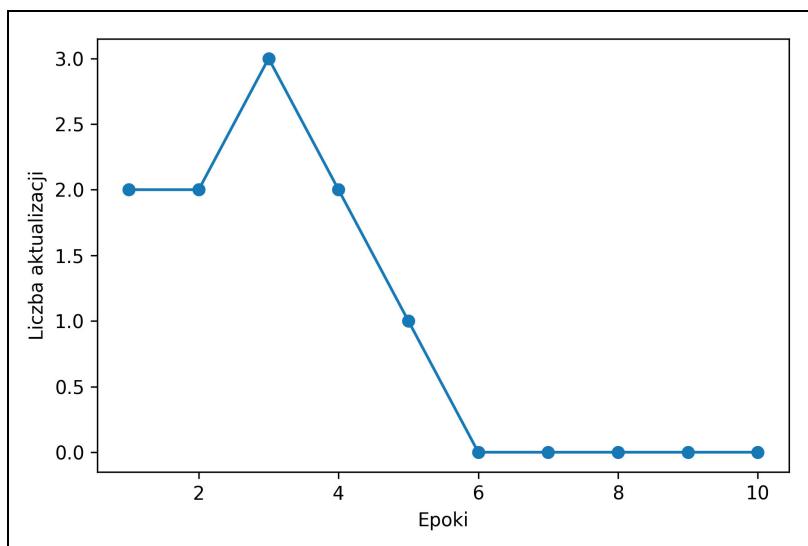


Rysunek 2.6. Graficzne przedstawienie zestawu danych uczących

Przejdźmy do trenowania algorytmu perceptronu na wydobytym podzbiorze danych Iris. Wyświetlimy do tego wykres **błędów nieprawidłowej klasyfikacji** dla każdej epoki, aby sprawdzić, czy algorytm jest zbieżny i zdolał odnaleźć granicę decyzyjną rozdzielającą obydwa gatunki kwiatów kosaćca:

```
>>> ppn = Perceptron(eta=0.1, n_iter=10)
>>> ppn.fit(X, y)
>>> plt.plot(range(1, len(ppn.errors_) + 1), ppn.errors_,
...           marker='o')
>>> plt.xlabel('Epoki')
>>> plt.ylabel('Liczba aktualizacji')
>>> plt.show()
```

Po uruchomieniu powyższego kodu naszym oczom powinien ukazać się wykres błędów klasyfikacji w funkcji epok, zaprezentowany na rysunku 2.7.



Rysunek 2.7. Wykres zbieżności algorytmu

Jak widać na rysunku 2.7, nasz perceptron osiągnął zbieżność już w szóstej epoce i teraz powinien znakomicie sobie radzić z klasyfikowaniem próbek uczących. Zaimplementujmy niewielką, wygodną funkcję służącą do wizualizowania granic decyzyjnych dla dwuwymiarowych zbiorów danych:

```
from matplotlib.colors import ListedColormap

def plot_decision_regions(X, y, classifier, resolution=0.02):

    # konfiguruje generator znaczników i mapę kolorów
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # rysuje wykres powierzchni decyzyjnej
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
```

```

x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                      np.arange(x2_min, x2_max, resolution))
Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
Z = Z.reshape(xx1.shape)
plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
plt.xlim(xx1.min(), xx1.max())
plt.ylim(xx2.min(), xx2.max())

# rysuje wykres próbek
for idx, cl in enumerate(np.unique(y)):
    plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                alpha=0.8, c=cmap(idx),3
                marker=markers[idx], label=cl)

```

Najpierw definiujemy liczbę barw (`colors`) i znaczników (`markers`), a następnie tworzymy mapę kolorów z listy barw za pomocą klasy `ListedColormap`. Określamy teraz wartości minimalne i maksymalne dwóch cech i używamy tak uzyskanych wektorów cech do utworzenia pary tablic `xx1` oraz `xx2` za pomocą funkcji `meshgrid`. Uczyliśmy nasz klasyfikator na dwóch wymiarach cech, dlatego musimy zmodyfikować tablice `xx1` i `xx2` oraz stworzyć macierz zawierającą taką samą liczbę kolumn jak zbiór uczący, dzięki czemu będziemy w stanie zastosować metodę `predict` do przewidywania etykiet klas z odpowiednich elementów tablic. Po przekształceniu przewidywanych etykiet klas do postaci tabelarycznej (o takich samych wymiarach jak tabele `xx1` i `xx2`) będziemy mogli narysować wykres konturowy, stosując funkcję `contourf`, która dopasowuje kolory do różnych regionów decyzyjnych dla każdej prognozowanej klasy w tablicy:

```

>>> plot_decision_regions(X, y, classifier=ppn)
>>> plt.xlabel('Długość działki [cm]')
>>> plt.ylabel('Długość płatka [cm]')
>>> plt.legend(loc='upper left')
>>> plt.show()

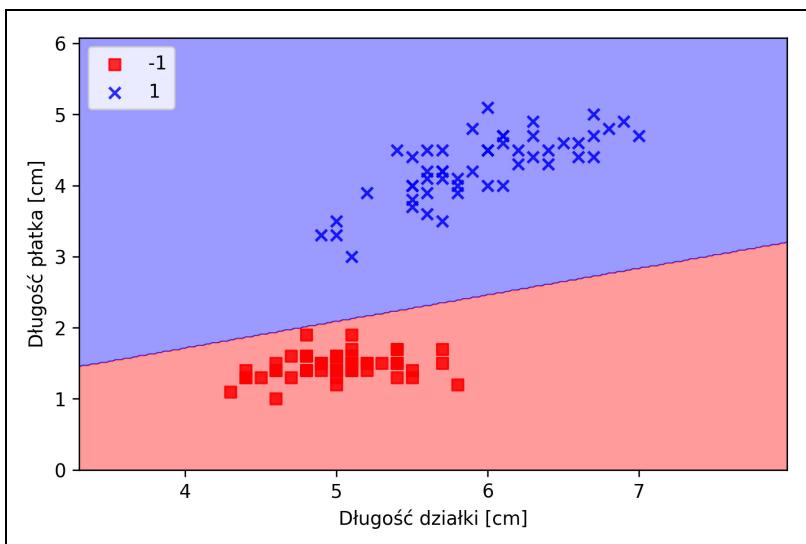
```

Po uruchomieniu powyższego kodu powinniśmy ujrzeć zaprezentowany na rysunku 2.8 wykres regionów decyzyjnych.

Jak widać na rysunku 2.8, perceptron wyznaczył granicę decyzyjną, dzięki której w idealny sposób sklasyfikował wszystkie próbki z podzbioru uczącego.

---

<sup>3</sup> **Errata:** Funkcja `plt.scatter` wykorzystywana do rysowania wykresu `plot_decision_regions` może powodować błędy w przypadku stosowania biblioteki `matplotlib` w wersji starszej od 1.5.0 — błędy pojawiają się, jeśli próbujemy za pomocą tej funkcji tworzyć wykresy ponad czterech klas. Rozwiązaniem tego problemu w starszych wersjach pakietu `matplotlib` jest zastąpienie wyrażenia `c=cmap(idx)` zwrotem `c=colors(idx)` — przyp. tłum.



Rysunek 2.8. Wykres regionów decyzyjnych

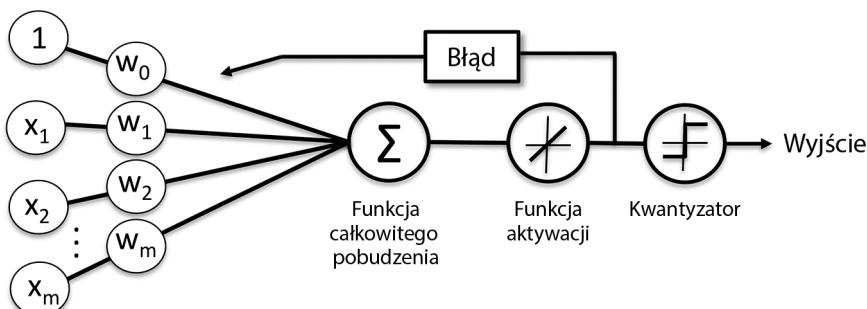
Choć perceptron perfekcyjnie sklasyfikował dwie odmiany kosańca, konwergencja stanowi jeden z największych problemów omawianego modelu. Rosenblatt dowód matematycznie, że reguła uczenia perceptronu wykazuje zbieżność, jeśli dwie klasy mogą zostać rozdzielone liniową hiperpłaszczyzną. Jeśli nie można idealnie odseparować tych klas wspomnianą granicą decyzyjną, wagi będą cały czas aktualizowane, chyba że ustalimy maksymalną liczbę epok.

## Adaptacyjne neurony liniowe i zbieżność uczenia

W tym podrozdziale przyjrzymy się kolejnej odmianie jednowarstwowej sieci neuronowej: **ADApacyjnemu LIniowemu NEuronowi (ADALINE)**. Model Adaline został zaprezentowany zaledwie kilka lat po algorytmie perceptronu Rosenblatta przez Bernarda Widrowa i jego doktoranta Tedda Hoffa; adaptacyjny neuron liniowy można uznać za twórcze rozwinięcie koncepcji perceptronu (B. Widrow i in., *An adaptive „Adaline” neuron using chemical „memistors”*, „Number Technical Report” 1553-2, Stanford Electron. Labs, Stanford, California, October 1960). Algorytm Adaline jest szczególnie interesujący, gdyż zaprezentowana w nim została kluczowa koncepcja definiowania i minimalizowania funkcji kosztu, co stanowi podstawę zrozumienia bardziej zaawansowanych algorytmów klasyfikujących, takich jak regresja logistyczna czy maszyny wektorów nośnych, a także modeli regresji omówionych w dalszych rozdziałach.

Podstawową różnicą pomiędzy regułą uczenia Adaline (zwaną również **regułą Widrowa-Hoffa**) a perceptronem Rosenblatta jest sposób traktowania wag: w przypadku adaptacyjnego neuronu liniowego wagi są aktualizowane na podstawie liniowej funkcji aktywacji, a nie funkcji skoku jednostkowego (jak to ma miejsce w perceptronie). W modelu Adaline taka liniowa funkcja aktywacji  $\phi(z)$  jest po prostu funkcją tożsamościową całkowitego pobudzenia w taki sposób, że  $\phi(w^T x) = w^T x$ .

Liniowa funkcja aktywacji służy do obliczania wag, natomiast podobny do funkcji skoku jednostkowego **kwantyzator** jest stosowany do przewidywania etykiet klas, co zostało zaprezentowane na rysunku 2.9.



Rysunek 2.9. Schemat modelu Adaline

Jeżeli porównamy rysunek 2.9 ze schematem algorytmu perceptronu (rysunek 2.4), zauważymy, że różnica polega na stosowaniu wartości ciągłych obliczanych za pomocą liniowej funkcji aktywacji do wyliczania błędu modelu i aktualizowania wag, a nie binarnych etykiet klas.

## Minimalizacja funkcji kosztu za pomocą metody gradientu prostego

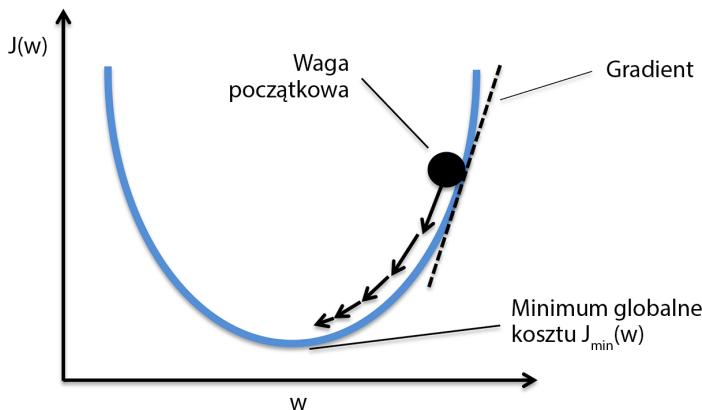
Jednym z najistotniejszych zadań w algorytmach nadzorowanego uczenia maszynowego jest zdefiniowanie **funkcji celu**, która będzie optymalizowana w procesie nauki. Funkcja celu często przyjmuje postać **funkcji kosztu**, którą pragniemy zminimalizować. W przypadku modelu Adaline możemy wyznaczyć funkcję kosztu  $J$ , wyznaczającą wagi za pomocą **sumy kwadratów błędów** (ang. *Sum of Squared Errors — SSE*) pomiędzy wyliczonymi wynikami a rzeczywistymi etykietami klas:

$$J(\mathbf{w}) = \frac{1}{2} \sum_i (y^{(i)} - \phi(z^{(i)}))^2$$

Wartość  $\frac{1}{2}$  została dodana jedynie dla naszej wygody; łatwiej nam będzie w ten sposób wyprawdzić gradient, o czym przekonamy się w dalszej części rozdziału. Główną zaletą tej liniowej funkcji aktywacji jest — w przeciwieństwie do funkcji skoku jednostkowego — umożliwienie

różniczkowania funkcji kosztu. Inną przydatną właściwością jest wypukłość funkcji kosztu; pozwala nam ją wykorzystywać prosty, ale potężny algorytm optymalizacyjny, zwany **gradientem prostym** (ang. *gradient descent*); umożliwia on znajdowanie wag minimalizujących funkcję kosztu klasyfikującą próbki zawarte w zbiorze danych Iris.

Na rysunku 2.10 widzimy, że działanie algorytmu gradientu prostego możemy przyrównać do schodzenia z górkę aż do osiągnięcia lokalnego lub globalnego minimum kosztu. W każdej iteracji schodzimy coraz niżej, a rozmiar kolejnego kroku jest określany przez wartości współczynnika uczenia, jak również przez nachylenie gradientu.



Rysunek 2.10. Schemat poglądowy działania algorytmu gradientu prostego

Z pomocą gradientu prostego możemy zaktualizować wagi poprzez oddalenie się od gradientu  $\nabla J(\mathbf{w})$  naszej funkcji kosztu  $J(\mathbf{w})$ :

$$\mathbf{w} := \mathbf{w} + \Delta \mathbf{w}$$

Zmiana wagi  $\Delta \mathbf{w}$  jest tu zdefiniowana jako ujemny gradient pomnożony przez współczynnik uczenia  $\eta$ :

$$\Delta \mathbf{w} = -\eta \nabla J(\mathbf{w})$$

Aby wyliczyć gradient funkcji kosztu, musimy obliczyć pochodną cząstkową tej funkcji przy uwzględnieniu każdej wagi  $w_j$ ,  $w_j, \frac{\partial J}{\partial w_j} = -\sum_i (y^{(i)} - \phi(z^{(i)}))x_j^{(i)}$ , dzięki czemu będziemy mogli zapisać aktualizację wagi  $w_j$  jako  $\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i (y^{(i)} - \phi(z^{(i)}))x_j^{(i)}$ . Aktualizujemy jednocześnie wszystkie wagi, dlatego reguła uczenia Adaline przyjmuje postać  $\mathbf{w} := \mathbf{w} + \Delta \mathbf{w}$ .

Osobom zaznajomionym z aparatem matematycznym przedstawiam sposób wyprowadzenia pochodnej cząstkowej funkcji kosztu za pomocą sumy kwadratów błędów w odniesieniu do j-tej wagi:

$$\begin{aligned}\frac{\partial J}{\partial w_j} &= \frac{\partial}{\partial w_j} \frac{1}{2} \sum_i (y^{(i)} - \phi(z^{(i)}))^2 = \\ &= \frac{1}{2} \frac{\partial}{\partial w_j} \sum_i (y^{(i)} - \phi(z^{(i)}))^2 = \\ &= \frac{1}{2} \sum_i 2(y^{(i)} - \phi(z^{(i)})) \frac{\partial}{\partial w_j} (y^{(i)} - \phi(z^{(i)})) = \\ &= \sum_i (y^{(i)} - \phi(z^{(i)})) \frac{\partial}{\partial w_j} \left( y^{(i)} - \sum_i (w_j^{(i)} x_j^{(i)}) \right) = \\ &= \sum_i (y^{(i)} - \phi(z^{(i)})) (-x_j^{(i)}) = \\ &= -\sum_i (y^{(i)} - \phi(z^{(i)})) x_j^{(i)}\end{aligned}$$

Mimo że reguła uczenia w modelu Adaline wygląda identycznie jak w przypadku perceptronu, wartość funkcji  $\phi(z^{(i)})$ , gdzie  $z^{(i)} = \mathbf{w}^T \mathbf{x}^{(i)}$ , stanowi liczbę rzeczywistą, a nie liczbę całkowitą etykiety klas. Ponadto aktualizacja wag jest obliczana na podstawie wszystkich próbek z zbiorze uczącym (czyli wagi nie są przyrostowo aktualizowane po sprawdzeniu każdej próbki), dlatego właśnie ta technika bywa również nazywana metodą wsadową gradientu prostego (ang. *batch gradient descent*).

## Implementacja adaptacyjnego neuronu liniowego w Pythonie

Reguły uczenia perceptronu i adaptacyjnego neuronu liniowego są do siebie bardzo podobne, dlatego wykorzystamy utworzoną wcześniej implementację perceptronu i zmodyfikujemy metodę `fit`, przez co wagi będą aktualizowane poprzez minimalizację funkcji kosztu za pomocą techniki gradientu prostego:

```
class AdalineGD(object):
    """Klasifikator — ADaptacyjny LIniowy NEuron.

    Parametry
    -----
    eta : zmienoprzecinkowy
        Współczynnik uczenia (w zakresie pomiędzy 0.0 i 1.0).
    n_iter : liczba całkowita
        Liczba przebiegów po zestawie uczącym.

    Metody
    -----
    fit(X, y)
        Miejsce do wpisania kodu uczenia.
    predict(X)
        Miejsce do wpisania kodu predycji.
    """

    def __init__(self, eta=0.01, n_iter=1000):
        self.eta = eta
        self.n_iter = n_iter
        self.w_initialized = False
        self.w_ = None
        self.errors_ = []

    def fit(self, X, y):
        self.w_ = np.zeros(1 + X.shape[1])
        self.w_initialized = True
        self.errors_ = []
        for _ in range(self.n_iter):
            errors = 0
            for xi, target in zip(X, y):
                update = self.eta * (target - self.predict(xi))
                self.w_[1:] += update * xi
                self.w_[0] += update
                errors += int(update != 0.0)
            self.errors_.append(errors)
        return self
```

### Atrybuty

```
-----  
w_ : jednowymiarowa tablica  
    Wagi po dopasowaniu.  
errors_ : lista  
    Liczba niewłaściwych klasyfikacji w każdej epoce.  
-----  
def __init__(self, eta=0.01, n_iter=50):  
    self.eta = eta  
    self.n_iter = n_iter  
  
def fit(self, X, y):  
    """ Trenowanie za pomocą danych uczących.
```

### Parametry

```
-----  
X : {tablicopodobny}, wymiary = [n_próbek, n_cech]  
    Wektory uczenia,  
    gdzie n_próbek oznacza liczbę próbek, a  
    n_cech — liczbę cech.  
y : tablicopodobny, wymiary = [n_próbek]  
    Wartości docelowe.
```

### Zwraca

```
-----  
self : obiekt
```

```
-----  
self.w = np.zeros(1 + X.shape[1])  
self.cost_ = []  
  
for i in range(self.n_iter):  
    output = self.net_input(X)  
    errors = (y - output)  
    self.w_[1:] += self.eta * X.T.dot(errors)  
    self.w_[0] += self.eta * errors.sum()  
    cost = (errors**2).sum() / 2.0  
    self.cost_.append(cost)  
return self  
  
def net_input(self, X):  
    """Oblicza całkowite pobudzenie""""  
    return np.dot(X, self.w_[1:]) + self.w_[0]  
  
def activation(self, X):  
    """Oblicza liniową funkcję aktywacji""""  
    return self.net_input(X)
```

```
def predict(self, X):
    """Zwraca etykietę klas po wykonaniu skoku jednostkowego"""
    return np.where(self.activation(X) >= 0.0, 1, -1)
```

W przeciwnieństwie do modelu perceptronu nie aktualizujemy tutaj wag po ocenieniu każdej pojedynczej próbki uczącej, lecz obliczamy gradient na podstawie całego zestawu danych uczących poprzez operację `self.eta * errors.sum()` dla wagi zerowej i `self.eta * X.T.dot(errors)` dla wag od 1 do  $m$ , gdzie `X.T.dot(errors)` jest **iloczynem macierzowo-wektorowym** macierzy cech z wektorem błędów. Podobnie jak w implementacji perceptronu, gromadzimy wartości kosztu w liście `self.cost_`, aby sprawdzić zbieżność algorytmu po zakończeniu uczenia.

Iloczyn macierzowo-wektorowy przypomina obliczanie iloczynu skalarnego wektorów, gdyż każdy wiersz macierzy jest traktowany jak pojedynczy wektor. Dzięki takiej wektoryzacji uzyskujemy zwięzlejszą notację oraz wydajniejszy proces obliczeniowy za pomocą biblioteki NumPy; np.:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix} = \begin{bmatrix} 1 \times 7 & 2 \times 8 & 3 \times 9 \\ 4 \times 7 & 5 \times 8 & 6 \times 9 \end{bmatrix} = \begin{bmatrix} 50 \\ 122 \end{bmatrix}$$

W praktyce znalezienie współczynnika uczenia  $\eta$  gwarantującego optymalną zbieżność wymaga odrobiny eksperymentowania. Dobieramy więc dwie wartości współczynnika uczenia:  $\eta = 0,1$  i  $\eta = 0,0001$  i narysujmy wykres funkcji kosztu dla liczby epok, dzięki czemu dowiemy się, jak skutecznie algorytm Adaline uczy się z danych uczących.

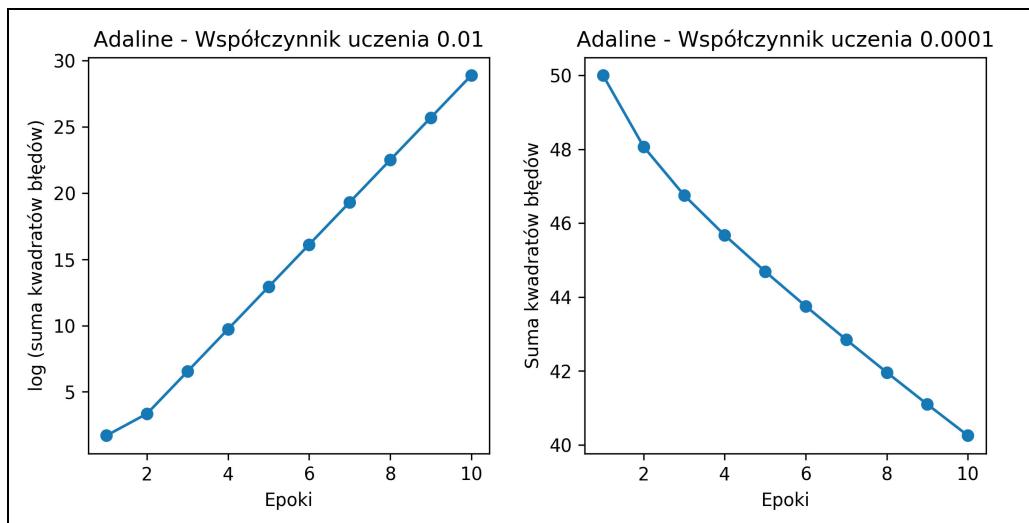
Zarówno współczynnik uczenia  $\eta$ , jak i liczba epok  $n\_iter$  to tak zwane **hiperparametry** implementacji perceptronu oraz adaptacyjnego neuronu liniowego. W rozdziale 4., „Tworzenie dobrych zbiorów uczących — wstępne przetwarzanie danych”, poznamy inne techniki umożliwiające automatyczne wyszukiwanie wartości różnych hiperparametrów zapewniających optymalną skuteczność modelu klasyfikacji.

Stwórzmy teraz wykres kosztów dla liczby epok przy założeniu dwóch różnych wartości współczynnika uczenia:

```
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(8, 4))
>>> ada1 = AdalineGD(n_iter=10, eta=0.01).fit(X, y)
>>> ax[0].plot(range(1, len(ada1.cost_) + 1),
...             np.log10(ada1.cost_), marker='o')
>>> ax[0].set_xlabel('Epoki')
>>> ax[0].set_ylabel('Log (suma kwadratów błędów)')
>>> ax[0].set_title('Adaline – Współczynnik uczenia 0.01')
>>> ada2 = AdalineGD(n_iter=10, eta=0.0001).fit(X, y)
```

```
>>> ax[1].plot(range(1, len(ada2.cost_) + 1),
...             ada2.cost_, marker='o')
>>> ax[1].set_xlabel('Epoki')
>>> ax[1].set_ylabel('Suma kwadratów błędów')
>>> ax[1].set_title('Adaline – Współczynnik uczenia 0.0001')
>>> plt.show()
```

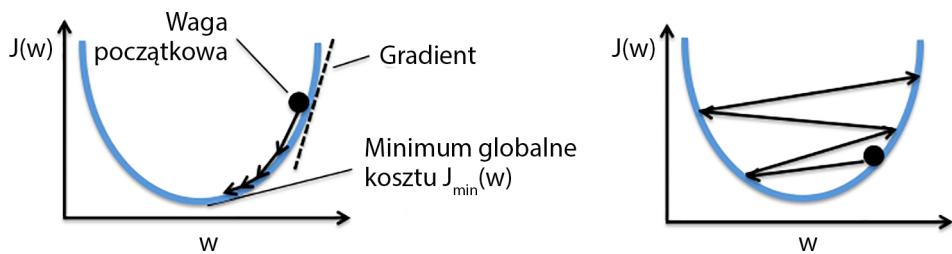
Jak widać na wykresach zaprezentowanych na rysunku 2.11, natrafiamy na dwa różne problemy. Wykres po lewej przedstawia sytuację, gdy dobieramy zbyt dużą wartość współczynnika uczenia — zamiast minimalizacji funkcji kosztu następuje powiększanie błędu wraz z każdą epoką, ponieważ **przeskakujemy nad minimum globalnym**.



Rysunek 2.11. Skutek doboru niewłaściwej wartości współczynnika uczenia

Na prawym wykresie z rysunku 2.11 widzimy, że koszt maleje, jednak wartość  $\eta = 0.001$  jest tak mała, że algorytm musiałby wykonać mnóstwo przebiegów, żeby uzyskać zbieżność. Na rysunku 2.12 prezentuję mechanizm zmiany wartości określonego parametru wag w celu minimalizacji funkcji kosztu  $J$  (lewy wykres). Z kolei prawy wykres pokazuje, co się dzieje, gdy dobieramy zbyt dużą wartość współczynnika uczenia i rozmijamy się z globalnym minimum.

Wiele omawianych w tej książce algorytmów uczenia maszynowego wymaga jakiejś formy skalowania cech w celu uzyskania optymalnej skuteczności, co zostanie dokładniej omówione w rozdziale 3., „Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn”. Metoda gradientu prostego stanowi jeden z wielu algorytmów, w których przydatne okazuje się skalowanie cech. W tym przypadku zastosujemy metodę skalowania zwaną **standardyzacją**, gdyż potraktowane nią dane uzyskują własności standardowego rozkładu normalnego. Średnia każdej cechy zostaje wyśrodkowana do wartości 0, a każda kolumna cech zawiera



Rysunek 2.12. Skutek dobioru wartości optymalnej (wykres po lewej) i za dużej (wykres po prawej) współczynnika uczenia

odchylenie standardowe równe 1. Przykładowo w celu standaryzacji  $j$ -tej cechy wystarczy odjąć średnią próbki  $\mu_j$  od każdej próbki uczącej i podzielić ją przez jej odchylenie standardowe  $\sigma_j$ :

$$\mathbf{x}'_j = \frac{\mathbf{x}_j - \mu_j}{\sigma_j}$$

Tutaj  $\mathbf{x}'_j$  jest wektorem składającym się z wartości  $j$ -tej cechy wszystkich próbek uczących  $n$ .

Możemy bardzo łatwo zaimplementować standaryzację za pomocą metod `mean` i `std` biblioteki NumPy:

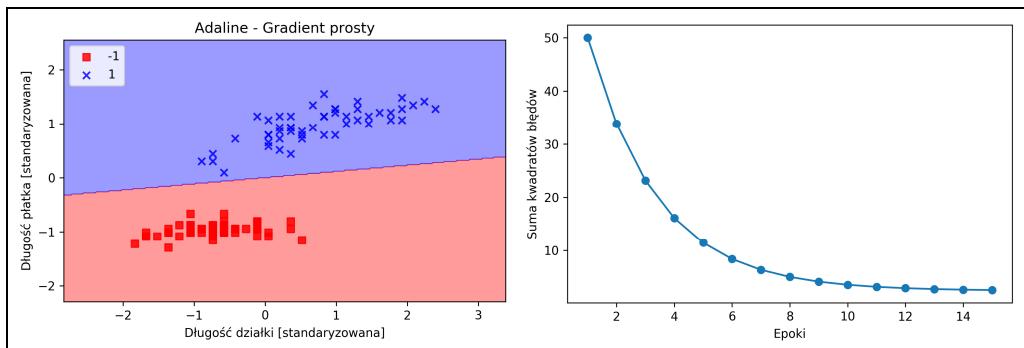
```
>>> X_std = np.copy(X)
>>> X_std[:,0] = (X[:,0] - X[:,0].mean()) / X[:,0].std()
>>> X_std[:,1] = (X[:,1] - X[:,1].mean()) / X[:,1].std()
```

Po wprowadzeniu standaryzacji ponownie wyuczymy model Adaline i sprawdzimy, czy algorytm będzie zbieżny przy współczynniku uczenia o wartości  $\eta = 0.01$ :

```
>>> ada = AdalineGD(n_iter=15, eta=0.01)
>>> ada.fit(X_std, y)
>>> plot_decision_regions(X_std, y, classifier=ada)
>>> plt.title('Adaline - Gradient prosty')
>>> plt.xlabel('Długość działalności [standaryzowana]')
>>> plt.ylabel('Długość płatka [standaryzowana]')
>>> plt.legend(loc='upper left')
>>> plt.show()
>>> plt.plot(range(1, len(ada.cost_) + 1), ada.cost_, marker='o')
>>> plt.xlabel('Epoki')
>>> plt.ylabel('Suma kwadratów błędów')
>>> plt.show()
```

Po uruchomieniu powyższego kodu powinniśmy ujrzeć wykresy regionów decyzyjnych oraz malejącego kosztu, pokazane na rysunku 2.13.

Jak widać, algorytm Adaline staje się teraz zbieżny po uczeniu się na standaryzowanych cechach przy stosowaniu współczynnika uczenia  $\eta = 0.01$ . Zauważ jednak, że suma kwadratów błędów pozostaje niezerowa pomimo właściwego sklasyfikowania wszystkich próbek.



Rysunek 2.13. Zbieżność algorytmu Adaline po standaryzacji cech

## Wielkoskalowe uczenie maszynowe i metoda stochastycznego spadku wzdłuż gradientu

W poprzednim podrozdziale nauczyliśmy się minimalizować funkcję kosztu poprzez wykonywanie kroku oddalającego od gradientu obliczonego z całego zbioru danych uczących; dlatego algorytm ten jest czasami nazywany **wsadową** metodą gradientu prostego. Wyobraź sobie teraz, że masz do dyspozycji olbrzymi zestaw danych zawierający miliony punktów danych, co jest dość często spotykana sytuacją w technikach uczenia maszynowego. W takim przypadku stosowanie metody wsadowej gradientu bywa dość kosztowne pod względem obliczeniowym, ponieważ musimy od nowa oceniać cały zbiór danych uczących za każdym razem, gdy wykonujemy kolejny krok w kierunku globalnego minimum.

Popularnym zamiennikiem algorytmu wsadowego gradientu prostego jest metoda **stochastycznego spadku wzdłuż gradientu** (ang. *stochastic gradient descent*), czasami nazywana także **iteracyjnym algorytmem spadku wzdłuż gradientu**. Nie aktualizujemy w tej sytuacji wag na podstawie sumy nagromadzonych błędów spośród wszystkich próbek  $\mathbf{x}^{(i)}$ :

$$\Delta \mathbf{w} = \eta \sum (y^{(i)} - \phi(z^{(i)})) \mathbf{x}^{(i)}$$

Aktualizujemy wagę przyrostowo dla każdej próbki uczącej:

$$\eta (y^{(i)} - \phi(z^{(i)})) \mathbf{x}^{(i)}$$

Chociaż algorytm stochastycznego spadku wzdłuż gradientu można uznawać za aproksymację gradientu prostego, zazwyczaj umożliwia on znacznie szybsze uzyskanie zbieżności, gdyż wagi są częściej aktualizowane. Każdy gradient jest wyliczany na podstawie pojedynczej próbki uczącej, dlatego powierzchnia błędów generuje większe szумy niż w gradiencie prostym, co również ma znaczenie, gdyż dzięki temu algorytm stochastyczny łatwiej może ignorować płytkie minima lokalne. Aby otrzymać dokładne wyniki za pomocą stochastycznego spadku wzdłuż

gradientu, bardzo ważne jest zaprezentowanie algorytmowi danych w przypadkowej kolejności, dlatego chcemy przed każdą epoką przetasować zbiór próbek, aby uniknąć cykliczności.

W implementacjach stochastycznego spadku wzdłuż gradientu niezmienny współczynnik uczenia  $\eta$  często jest zastępowany adaptacyjnym współczynnikiem uczenia, którego wartość maleje wraz z upływem czasu; może on przybrać np. następującą postać:  $\frac{c_1}{[liczba\ iteracji]+c_2}$ , gdzie  $c_1$  i  $c_2$  są stałymi. Zwrót uwagę, że algorytm ten nie osiąga globalnego minimum, lecz dociera do jego bardzo zbliżonych wartości. Dzięki adaptacyjnemu współczynnikiowi uczenia możemy jeszcze bardziej zbliżyć się do globalnego minimum.

Kolejną zaletą metody stochastycznego spadku wzdłuż gradientu jest możliwość wykorzystania jej do **uczenia przyrostowego**. Rozwiązywanie to polega na trenowaniu modelu za pomocą ciągle napływających nowych danych uczących. Jest to technika przydatna zwłaszcza w sytuacji gromadzenia dużych ilości informacji — np. danych o użytkownikach w typowej aplikacji sieciowej. Poprzez uczenie w locie system jest w stanie natychmiastowo dostosować się do zmian, a dane uczące mogą zostać usunięte po zaktualizowaniu modelu, jeżeli pojemność dyskowa stanowi problem.

Kompromisem pomiędzy metodą gradientu prostego a stochastycznego spadku wzdłuż gradientu jest tzw. **uczenie z pomocą minipaczek** (ang. *mini-batch learning*). Rozwiązywanie to można rozpatrywać jako stosowanie metody wsadowej gradientu do mniejszych podzbiorów danych uczących — np. 50 próbek. Zaletą uczenia z pomocą minipaczek jest znacznie szybsza konwergencja w porównaniu z gradientem prostym z powodu częstszych aktualizacji wag. Do tego metoda ta pozwala zastępować pętlę for używaną do próbek uczących w **stochastycznym spadku wzdłuż gradientu** operacjami wektorowymi, co jeszcze bardziej poprawia skuteczność obliczeniową danego algorytmu uczenia.

Wcześniej zaimplementowaliśmy regułę uczenia Adaline wykorzystującą metodę gradientu prostego, dlatego wystarczy wprowadzić do niej kilka modyfikacji, żeby algorytm zaczął aktualizować wagi poprzez stochastyczny spadek wzdłuż gradientu. Teraz wewnątrz metody fit będziemy aktualizować wagi po sprawdzeniu każdej próbki uczącej. Następnie zaimplementujemy dodatkową metodę partial\_fit, która nie inicjuje od nowa wag w przypadku uczenia w locie. Aby sprawdzić zbieżność algorytmu po treningu, będziemy obliczać koszt jako średni koszt próbek uczących w każdej epoce. Ponadto dodamy możliwość tasowania (shuffle) danych uczących przed rozpoczęciem każdej epoki, dzięki czemu unikniemy cykliczności podczas optymalizowania funkcji kosztu; parametr random\_state służy do generowania wartości losowej dla zachowania większej ciągłości:

```
from numpy.random import seed

class AdalineSGD(object):
    """Klasyfikator — ADaptacyjny LIniowy NEuron.
    Parametry
    -----
    ...
```

`eta` : zmiennoprzecinkowy

Współczynnik uczenia (w zakresie pomiędzy 0.0 i 1.0).

`n_iter` : liczba całkowita

Liczba przebiegów po zestawie uczącym.

Atrybuty

-----  
`w_` : jednowymiarowa tablica

Wagi po dopasowaniu.

`errors_` : lista

Liczba nieprawidłowych klasyfikacji w każdej epoce.

`shuffle` : wartość boolowska (domyślnie: True)

Jeżeli jest ustalona wartość True,

tasuje dane uczące przed każdą epoką w celu zapobiegnięcia cykliczności.

`random_state` : liczba całkowita (domyślnie: None)

Ustanawia przypadkowy stan dla operacji tasowania  
oraz inicjacji wag.

=====

```
def __init__(self, eta=0.01, n_iter=10,  
            shuffle=True, random_state=None):  
    self.eta = eta  
    self.n_iter = n_iter  
    self.w_initialized = False  
    self.shuffle = shuffle  
    if random_state:  
        seed(random_state)
```

def fit(self, X, y):

"""Dopasowanie danych uczących.

Parametry

-----  
`X` : {tablicopodobny}, wymiary = [n\_próbek, n\_cech]

Wektory uczące, gdzie n\_próbek

oznacza liczbę próbek, a

n\_cech określa liczbę cech.

`y` : tablicopodobny, wymiary = [n\_próbek]

Wartości docelowe.

Zwraca

-----  
`self` : obiekt

=====

```
self._initialize_weights(X.shape[1])  
self.cost_ = []  
for i in range(self.n_iter):  
    if self.shuffle:
```

```

        X, y = self._shuffle(X, y)
        cost = []
        for xi, target in zip(X, y):
            cost.append(self._update_weights(xi, target))
        avg_cost = sum(cost)/len(y)
        self.cost_.append(avg_cost)
    return self

def partial_fit(self, X, y):
    """Dopasowuje dane uczące bez ponownej inicjalizacji wag"""
    if not self.w_initialized:
        self._initialize_weights(X.shape[1])
    if y.ravel().shape[0] > 1:
        for xi, target in zip(X, y):
            self._update_weights(xi, target)
    else:
        self._update_weights(X, y)
    return self

def _shuffle(self, X, y):
    """Tasuje dane uczące"""
    r = np.random.permutation(len(y))
    return X[r], y[r]

def _initialize_weights(self, m):
    """Inicjuje wagę, przydzielając im wartości zerowe"""
    self.w_ = np.zeros(1 + m)
    self.w_initialized = True

def _update_weights(self, xi, target):
    """Wykorzystuje regułę uczenia Adaline do aktualizacji wag"""
    output = self.net_input(xi)
    error = (target - output)
    self.w_[1:] += self.eta * xi.dot(error)
    self.w_[0] += self.eta * error
    cost = 0.5 * error**2
    return cost

def net_input(self, X):
    """Oblicza całkowite pobudzenie"""
    return np.dot(X, self.w_[1:]) + self.w_[0]

def activation(self, X):
    """Oblicza liniową funkcję aktywacji"""
    return self.net_input(X)

def predict(self, X):
    """Zwraca etykietę klas po wykonaniu skoku jednostkowego"""
    return np.where(self.activation(X) >= 0.0, 1, -1)

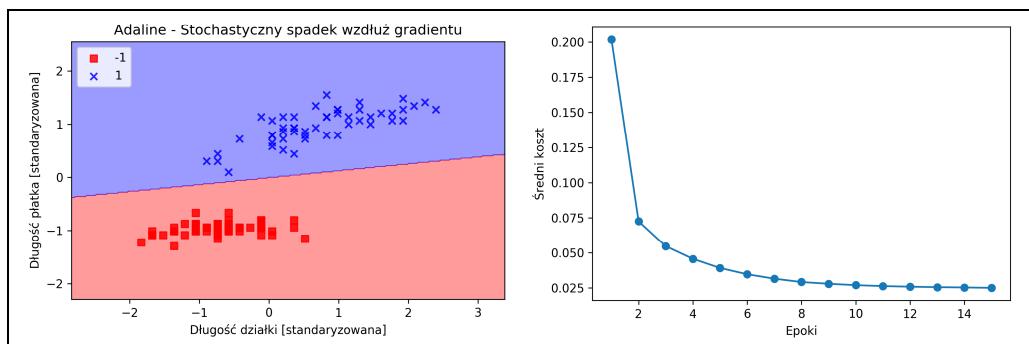
```

Stosowana w klasyfikatorze AdalineSGD metoda `_shuffle` działa w następujący sposób: dzięki funkcji `permutation` w `numpy.random` generujemy losową sekwencję unikatowych liczb w przedziale od 0 do 100. Liczby te są następnie wykorzystywane jako indeksy umożliwiające tasowanie macierzy cech i wektora etykiet klas.

Teraz możemy wykorzystać metodę `fit` do trenowania klasyfikatora `AdalineSGD` i wyświetlić wyniki nauki za pomocą funkcji `plot_decision_regions`:

```
>>> ada = AdalineSGD(n_iter=15, eta=0.01, random_state=1)
>>> ada.fit(X_std, y)
>>> plot_decision_regions(X_std, y, classifier=ada)
>>> plt.title('Adaline – Stochastyczny spadek wzduż gradientu')
>>> plt.xlabel('Długość działki [standaryzowana]')
>>> plt.ylabel('Długość płatka [standaryzowana]')
>>> plt.legend(loc='upper left')
>>> plt.show()
>>> plt.plot(range(1, len(ada.cost_) + 1), ada.cost_, marker='o')
>>> plt.xlabel('Epoki')
>>> plt.ylabel('Średni koszt')
>>> plt.show()
```

Wykresy uzyskane po uruchomieniu powyższego kodu zostały zaprezentowane na rysunku 2.14.



Rysunek 2.14. Uczenie przy zastosowaniu metody stochastycznego spadku wzduż gradientu

Jak widać, średni koszt maleje dość szybko, a ostateczna granica decyzyjna po 15 epokach przypomina uzyskaną za pomocą wsadowej metody gradientu prostego. Jeżeli chcemy zaktualizować nasz model — np. by zastosować go do uczenia przyrostowego za pomocą danych przesyłanych strumieniowo — wystarczy wywołać metodę `partial_fit` wobec pojedynczych próbek w następujący sposób: `ada.partial_fit(X_std[0, :], y[0])`.

## Podsumowanie

W tym rozdziale przyjrzaliśmy się uważnie podstawowym koncepcjom klasyfikatorów liniowych stosowanych w uczeniu nadzorowanym. Po zaimplementowaniu perceptronu dowiedzieliśmy się, jak można wydajnie uczyć adaptacyjne neurony liniowe poprzez wektoryzację gradientu prostego, a także jak można wykorzystać metodę stochastycznego spadku wzduż gradientu do uczenia w locie. Gdy już potrafimy implementować proste klasyfikatory w Pythonie, jesteśmy gotowi na następny rozdział, w którym wykorzystamy bibliotekę uczenia maszynowego scikit-learn do tworzenia bardziej zaawansowanych i potężniejszych klasyfikatorów, powszechnie używanych zarówno na uczelniach, jak i w przemyśle.



# **Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn**

W tym rozdziale poznamy niektóre z najpopularniejszych i najpotężniejszych algorytmów uczenia maszynowego, stosowane na co dzień zarówno na uczelniach, jak i w przemyśle. W trakcie analizowania różnic pomiędzy kilkoma klasyfikującymi algorytmami uczenia nadzorowanego przyjrzymy się również ich poszczególnym zaletom i wadom. Do tego nauczmy się korzystać z biblioteki scikit-learn, zawierającej przyjazny interfejs użytkownika umożliwiający wydajne oraz produktywne stosowanie omawianych algorytmów.

Zagadnienia, którymi zajmiemy się w niniejszym rozdziale, są następujące:

- omówienie popularnych algorytmów klasyfikujących,
- używanie biblioteki uczenia maszynowego scikit-learn,
- pytania, jakie należy sobie zadać podczas wyboru algorytmu uczenia maszynowego.

# Wybór algorytmu klasyfikującego

Wybór odpowiedniego algorytmu klasyfikującego do rozwiązywania konkretnego zadania problemowego wymaga odrobiny praktyki: każdy algorytm ma swoje cechy charakterystyczne i bazuje na określonych założeniach. Przypomnijmy twierdzenie „niedarmowego obiadu”: nie istnieje taki klasyfikator, który działałby optymalnie we wszystkich możliwych sytuacjach. W praktyce zawsze jest zalecane porównanie skuteczności przynajmniej kilku różnych algorytmów uczenia maszynowego, dzięki czemu można wybrać model sprawujący się najlepiej w rozwiązywaniu danego problemu; poszczególne zagadnienia problemowe różnią się pomiędzy sobą liczbą cech lub próbek, poziomami szumów w zestawie danych, a także liniową rozdzielnoscią (lub jej brakiem) klas.

Ostatecznie skuteczność klasyfikatora, mocy obliczeniowej oraz siły predykcyjnej zależą w olbrzymim stopniu od zbioru danych przeznaczonych do uczenia. Trenowanie algorytmu uczenia maszynowego możemy podsumować w pięciu głównych etapach:

1. Dobór cech.
2. Wybór metryki skuteczności.
3. Wybór klasyfikatora i algorytmu optymalizacji.
4. Ocena skuteczności modelu.
5. Strojenie algorytmu.

Zadaniem niniejszej książki jest przekazywanie wiedzy o uczeniu maszynowym krok po kroku, dlatego w tym rozdziale skoncentrujemy się na ogólnych założeniach poszczególnych algorytmów, a w kolejnych rozdziałach rozwiniemy poszczególne zagadnienia, takie jak wybór cech, wstępne przetwarzanie danych, metryki skuteczności czy strojenie hiperparametryczne.

## Pierwsze kroki z biblioteką scikit-learn

W rozdziale 2., „Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji”, poznaliśmy dwa powiązane ze sobą algorytmy stosowane w klasyfikacji: regułę perceptronu oraz model Adaline, które zaimplementowaliśmy w Pythonie. Teraz nauczymy się posługiwać biblioteką scikit-learn, cechującą się przyjaznym interfejsem użytkownika oraz zoptymalizowaną implementacją kilku algorytmów klasyfikujących. Jednak ta biblioteka to nie tylko spora baza zróżnicowanych algorytmów uczenia maszynowego, lecz także repozytorium wygodnych funkcji pozwalających na wstępne przetwarzanie danych, strojenie oraz ocenę modeli. Tematom tym zostały poświęcone rozdziały 4., „Tworzenie dobrych zbiorów uczących — wstępne przetwarzanie danych”, i 5., „Kompresja danych poprzez redukcję wymiarowości”.

## Uczenie perceptronu za pomocą biblioteki scikit-learn

Naukę korzystania z biblioteki scikit-learn rozpoczęliśmy od wytrenowania modelu perceptronu przypominającego implementację omówioną w rozdziale 2., „Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji”. Dla uproszczenia będziemy w kolejnych podrozdziałach nadal używać znanego nam zestawu danych **Iris**. Zestaw tych danych znajduje się w zasobach biblioteki scikit-learn, ponieważ jest on bardzo popularny i często wykorzystywany do testowania algorytmów. Do tego w celach wizualizacji będziemy wykorzystywać tylko dwie **cechy kwiatów** kosaćca zawarte w bazie danych Iris.

Przydzielimy **długość płatka** i **szerokość płatka** ze 150 próbek kwiatów do macierzy cech  $X$ , a etykiety klas odpowiednich gatunków kosaćca do wektora  $y$ :

```
>>> from sklearn import datasets
>>> import numpy as np
>>> iris = datasets.load_iris()
>>> X = iris.data[:, [2, 3]]
>>> y = iris.target
```

Jeżeli wykonamy operację `np.unique(y)`, aby zostały wyświetlane poszczególne etykiety klas występujące w `iris.target`, zobaczymy, że nazwy trzech gatunków kosaćca (*Iris-setosa*, *Iris-versicolor* oraz *Iris-virginica*) są już przechowywane w postaci liczb całkowitych (0, 1, 2), co jest zalecane, gdyż w ten sposób łatwiej uzyskać optymalną skuteczność wielu bibliotek uczenia maszynowego.

Aby ocenić skuteczność wyuczonego modelu wobec nieznanych informacji, podzielimy nasz zbiór danych na oddzielne zestawy danych uczących i testowych. W rozdziale 6., „Najlepsze metody oceny modelu i strojenie parametryczne”, zapoznamy się dokładniej z najlepszymi sposobami oceny modelu:

```
>>> from distutils.version import LooseVersion as Version
>>> from sklearn import __version__ as sklearn_version
>>> if Version(sklearn_version) < '0.18':
>>>     from sklearn.grid_search import train_test_split
>>> else:
>>>     from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=0.3, random_state=0)
```

Za pomocą funkcji `train_test_split` znajdującej się w module `model_selection` losowo rozdzielimy tablice  $X$  i  $y$  na dwie części: 30% danych tworzy teraz dane testowe (45 próbek), a pozostałe 70% — dane uczące (105 próbek).

Jak pamiętamy z umieszczonego w rozdziale 2., „Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji”, przykładu z modelem **gradientu prostego**, wiele algorytmów uczenia maszynowego i optymalizacji wymaga również skalowania cech w celu zoptymalizowania

ich przebiegów. W obecnym przypadku przeprowadzimy standaryzację cech za pomocą klasy StandardScaler znajdującej się w module preprocessing:

```
>>> from sklearn.preprocessing import StandardScaler  
>>> sc = StandardScaler()  
>>> sc.fit(X_train)  
>>> X_train_std = sc.transform(X_train)  
>>> X_test_std = sc.transform(X_test)
```

Dzięki powyższemu kodowi wczytaliśmy klasę StandardScaler z modulu przetwarzania wstępniego i zainicjowaliśmy nowy obiekt StandardScaler, który przydzieliśmy do zmiennej sc. Za pomocą metody fit klasa StandardScaler oszacowała parametry  $\mu$  (wartość średnia próbek) i  $\sigma$  (odchylenie standardowe) dla każdego wymiaru cech stanowiącego część danych uczących. Poprzez wywołanie metody transform dokonujemy standaryzacji danych uczących na podstawie wspomnianych parametrów  $\mu$  i  $\sigma$ . Zwrć uwagę, że wykorzystaliśmy te same parametry do standaryzacji zestawu testowego, dzięki czemu wartości danych ze zbioru testowego i uczącego są ze sobą porównywalne.

Po przeprowadzeniu standaryzacji danych testowych możemy przejść do uczenia modelu perceptronu. Większość algorytmów biblioteki scikit-learn zawiera domyślnie obsługę klasyfikacji wieloklasowej wykonywaną za pomocą metody **jeden przeciw reszcie** (ang. *One versus Rest* — OvR), dzięki której jesteśmy w stanie przekazać perceptronowi jednocześnie dane wszystkich trzech gatunków kosańca. Omawiany fragment kodu przedstawia się następująco:

```
>>> from sklearn.linear_model import Perceptron  
>>> ppn = Perceptron(n_iter=40, eta0=0.1, random_state=0)  
>>> ppn.fit(X_train_std, y_train)
```

Interfejs biblioteki scikit-learn przypomina nam implementację perceptronu, omówioną w rozdziale 2., „Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji”: po wczytaniu klasy Perceptron z modulu linear\_model zainicjowaliśmy nowy obiekt Perceptron i wyuczyliśmy model za pomocą metody fit. W tym przypadku parametr eta0 jest odpowiednikiem współczynnika uczenia eta, który stosowaliśmy w poprzedniej implementacji perceptronu, a poprzez parametr n\_iter definiujemy liczbę epok (przebiegów po zestawie danych uczących). Jak pamiętamy z rozdziału 2., „Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji”, dobór właściwego współczynnika uczenia wymaga eksperymentowania. Jeżeli przyjmiemy zbyt dużą jego wartość, algorytm rozminie się z globalnym minimum kosztu. Z kolei przy zbyt małej wartości współczynnika uczenia algorytm będzie potrzebował większej liczby epok do uzyskania zbieżności, co powoduje spowolnienie procesu uczenia — zwłaszcza w przypadku dużych zbiorów danych. Ponadto wprowadziliśmy parametr random\_state w celu odwzorowania tasowania danych uczących po zakończeniu każdej epoki.

Po wytrenowaniu perceptronu możemy wprowadzić przewidywanie wyników za pomocą metody predict, bardzo podobnie jak w implementacji omówionej w rozdziale 2., „Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji”. Odpowiedzialny za to kod został zaprezentowany poniżej:

```
>>> y_pred = ppn.predict(X_test_std)
>>> print('Nieprawidłowo sklasyfikowane próbki: %d' % (y_test != y_pred).sum())
Nieprawidłowo sklasyfikowane próbki: 4
```

Po uruchomieniu tego kodu widzimy, że perceptron nieprawidłowo klasyfikuje 4 z 45 próbek, zatem błąd nieprawidłowej klasyfikacji na zbiorze danych testowych wynosi 0,089 lub 8,9% ( $4/45 \approx 0,089$ ).

Wiele osób stosujących algorytmy uczenia maszynowego zamiast **błędu** nieprawidłowej klasyfikacji oblicza **dokładność** klasyfikacji modelu, której wzór wygląda następująco:

$1 - \text{błąd nieprawidłowej klasyfikacji} = 0,911 \text{ lub } 91,1\%$

Biblioteka scikit-learn zawiera również implementacje wielu różnych metryk skuteczności, które są dostępne w module `metrics`. Przykładowo dokładność klasyfikacji perceptronu wobec zestawu testowego wyliczamy w następujący sposób:

```
>>> from sklearn.metrics import accuracy_score
>>> print('Dokładność: %.2f' % accuracy_score(y_test, y_pred))
Dokładność: 0.91
```

Tutaj zmienna `y_test` oznacza rzeczywiste etykiety klas, a `y_pred` — przewidziane etykiety klas.

Zwróć uwagę, że oceniamy skuteczność naszych modeli na podstawie zestawu testowego opisywanego w niniejszym rozdziale. W rozdziale 5., „Komprezja danych poprzez redukcję wymiarowości”, poznasz przydatne techniki, w tym analizę graficzną (np. krzywe uczenia), pozwalające na wykrywanie i zapobieganie nadmiernemu dopasowaniu. **Nadmierne dopasowanie (przetrenowanie, ang. overfitting)** to sytuacja, w której model prawidłowo wykrywa wzorce w danych uczących, jednak nie potrafi ich uogólnić na nieznane dane.

Możemy w końcu wykorzystać poznaną w rozdziale 2., „Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji”, funkcję `plot_decision_regions` do narysowania wykresu **regionów decyzyjnych** naszego świeżo wyuczonego modelu perceptronu oraz ujrzeć na własne oczy, w jaki sposób algorytm rozdziela poszczególne próbki kwiatów. Dodajmy jednak niewielką modyfikację, w której zaznaczymy czarnymi kółkami próbki pochodzące z zestawu testowego:

```
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt

def versiontuple(v):
    return tuple(map(int, (v.split(".")))))

def plot_decision_regions(X, y, classifier,
                          test_idx=None, resolution=0.02):
```

```

# konfiguruje generator znaczników i mapę kolorów
markers = ('s', 'x', 'o', '^', 'v')
colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
cmap = ListedColormap(colors[:len(np.unique(y))])

# rysuje wykres powierzchni decyzyjnej
x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                       np.arange(x2_min, x2_max, resolution))
Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
Z = Z.reshape(xx1.shape)
plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
plt.xlim(xx1.min(), xx1.max())
plt.ylim(xx2.min(), xx2.max())

# rysuje wykres wszystkich próbek
for idx, cl in enumerate(np.unique(y)):
    plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                alpha=0.8, c=cmap(idx),
                marker=markers[idx], label=cl)

# zaznacza próbki testowe
if test_idx:
    if not versiontuple(np.__version__) >= versiontuple('1.9.0'):
        X_test, y_test = X[list(test_idx), :], y[list(test_idx)]
        warnings.warn('Please update to NumPy 1.9.0 or newer')
    else:
        X_test, y_test = X[test_idx, :], y[test_idx]

    plt.scatter(X_test[:, 0], X_test[:, 1], c='',
                alpha=1.0, linewidths=1, marker='o', edgecolors='k'
                s=80, label='Zestaw testowy')

```

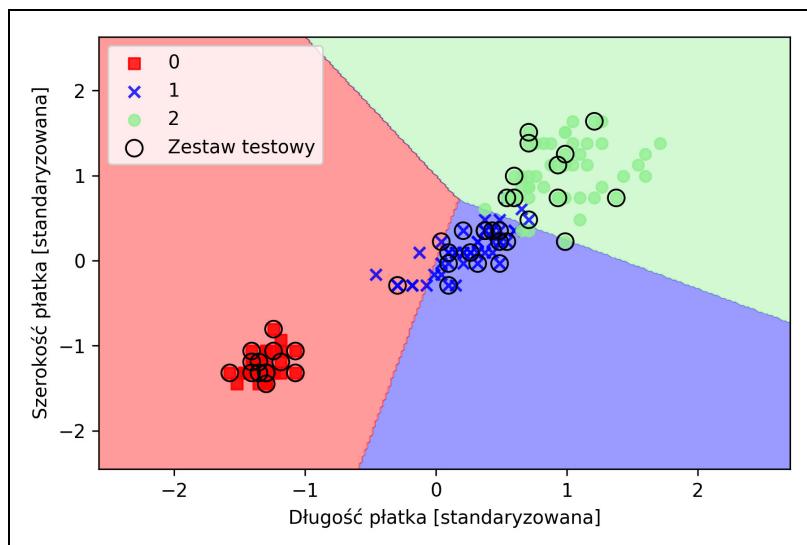
Po wprowadzeniu nieznacznej modyfikacji w funkcji `plot_decision_regions` (zaznaczonej w powyższym listingu) możemy teraz zdefiniować indeksy próbek, które chcemy zaznaczyć na wykresach. Odpowiedzialny jest za to poniższy fragment kodu:

```

>>> X_combined_std = np.vstack((X_train_std, X_test_std))
>>> y_combined = np.hstack((y_train, y_test))
>>> plot_decision_regions(X=X_combined_std,
...                         y=y_combined,
...                         classifier=ppn,
...                         test_idx=range(105,150))
>>> plt.xlabel('Długość płatka [standaryzowana]')
>>> plt.ylabel('Szerokość płatka [standaryzowana]')
>>> plt.legend(loc='upper left')
>>> plt.show()

```

Jak widać na rysunku 3.1, nie da się idealnie rozdzielić wszystkich trzech gatunków kwiatów liniowymi granicami decyzyjnymi.



Rysunek 3.1. Wykres wyników uczenia perceptronu stworzonego przy użyciu biblioteki scikit-learn

Pamiętamy z rozdziału 2., „Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji”, że algorytm perceptronu nigdy nie jest zbieżny ze zbiorami danych, które nie są idealnie rozdzielne liniowo, dlatego właśnie stosowanie tego modelu zazwyczaj nie jest zalecane. W następnych podrozdziałach przyjrzymy się późniejszym klasyfikatorom liniowym, które wykazują zbieżność z minimalną wartością kosztu, nawet jeśli klas nie cechuje doskonałej rozdzielalności liniowej.

Poszczególne funkcje i klasy biblioteki scikit-learn mają dodatkowe parametry, które pomijamy w celu zachowania przejrzystości. Więcej na temat tych parametrów dowiesz się po wpisaniu w Pythonie polecenia `help` (nazwa\_funkcji), np. `help` (`Perceptron`), lub ze znakomitej dokumentacji biblioteki scikit-learn, dostępnej pod adresem <http://scikit-learn.org/stable/>.

# Modelowanie prawdopodobieństwa przynależności do klasy za pomocą regresji logistycznej

Chociaż reguła uczenia perceptronu stanowi łatwe i przyjemne wprowadzenie do algorytmów klasyfikujących, jej największą wadą jest brak zbieżności w przypadku, gdy badane klasy nie są doskonale rozdzielne liniowo. Omówione w poprzednim podrozdziale zadanie klasyfikacji jest klasycznym przykładem wspomnianego problemu. Intuicyjnie rozumiemy, że wagi będą aktualizowane w nieskończoność, ponieważ w każdej epoce pojawia się przynajmniej jedna nie właściwie sklasyfikowana próbka. Możemy, oczywiście, zmienić współczynnik uczenia i zwiększyć liczbę epok, pamiętaj jednak, że perceptron nigdy nie stanie się całkowicie zbieżny z tym zbiorem danych. Lepiej wykorzystamy czas, analizując kolejny, również prosty, ale potężniejszy algorytm służący do rozwiązywania liniowych i binarnych problemów: model **regresji logistycznej** (ang. *logistic regression*). Zauważmy, że pomimo nazwy mamy tu do czynienia z modelem klasyfikacji, nie regresji.

## Teoretyczne podłoże regresji logistycznej i prawdopodobieństwa warunkowego

Regresja logistyczna to bardzo łatwy do zaimplementowania model klasyfikacji, który jednak doskonale sprawdza się w przypadku klas rozdzielnych liniowo. Jest to jeden z najczęściej stosowanych algorytmów klasyfikujących w przemyśle. Model ten bardzo przypomina algorytmy perceptronu i Adaline; można go również stosować do klasyfikacji binarnej, po czym rozwinać do klasyfikacji wieloklasowej za pomocą techniki OvR.

Aby wyjaśnić koncepcję regresji logistycznej jako modelu probabilistycznego, wyjaśnijmy najpierw pojęcie **ilorazu szans** (ang. *odds ratio*), czyli szansy wystąpienia danego zdarzenia. Wzór na iloraz szans można zapisać następująco:  $\frac{p}{(1-p)}$ , gdzie  $p$  oznacza prawdopodobieństwo pozytywnego zdarzenia. Termin **pozytywne zdarzenie** wcale nie musi oznaczać czegoś *dobrego*, ale mówi nam o zdarzeniu, którego wystąpienie chcemy przewidzieć, np. prawdopodobieństwo wykrycia konkretnej choroby u pacjenta; możemy zdefiniować pozytywne zdarzenie jako etykietę klas  $y = 1$ . Stąd możemy przejść do funkcji **logitowej**, będącej w istocie logarytmem ilorazu szans (zlogarytmowane szanse):

$$\text{logit}(p) = \log \frac{p}{(1-p)}$$

Funkcja logitowa przyjmuje wartości wejściowe w zakresie od 0 do 1 i przekształca je w wartości z pełnego przedziału liczb rzeczywistych, co możemy wykorzystać do ukazania liniowego związku pomiędzy wartościami cech a zlogarytmowanymi szansami:

$$\text{logit}\left(p(y=1 \mid \mathbf{x})\right) = w_0x_0 + w_1x_1 + \dots + w_mx_m = \sum_{i=0}^m w_i x_i = \mathbf{w}^T \mathbf{x}$$

Tutaj  $p(y=1 \mid \mathbf{x})$  oznacza prawdopodobieństwo warunkowe, zgodnie z którym dana próbka należy do klasy 1 przy znanych cechach  $x$ .

Interesuje nas prognozowanie prawdopodobieństwa przynależności próbki do określonej klasy, co jest odwrotnością funkcji logitowej. Mamy tu do czynienia z funkcją **logistyczną**, zwaną również funkcją **sigmoidalną** (s-kształtną) z racji charakterystycznego kształtu wykresu.

$$\phi(z) = \frac{1}{1 + e^{-z}}$$

Wartość  $z$  stanowi tutaj całkowite pobudzenie, czyli liniową kombinację wag i cech funkcji, którą można policzyć za pomocą wzoru  $z = \mathbf{w}^T \mathbf{x} = w_0 + w_1x_1 + \dots + w_mx_m$ .

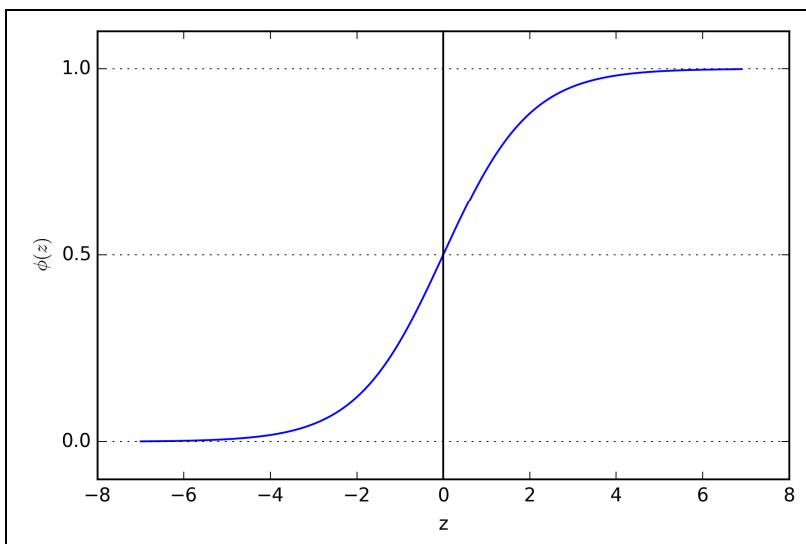
Narysujmy teraz wykres funkcji sigmoidalnej w zakresie wartości od  $-7$  do  $7$ , aby się przekonać, jak wygląda krzywa s-kształtna:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> def sigmoid(z):
...     return 1.0 / (1.0 + np.exp(-z))
>>> z = np.arange(-7, 7, 0.1)
>>> phi_z = sigmoid(z)
>>> plt.plot(z, phi_z)
>>> plt.axvline(0.0, color='k')
>>> plt.axhspan(0.0, 1.0, facecolor='1.0', alpha=1.0, ls='dotted')
>>> plt.axhline(y=0.5, ls='dotted', color='k')
>>> plt.yticks([0.0, 0.5, 1.0])
>>> plt.ylim(-0.1, 1.1)
>>> plt.xlabel('z')
>>> plt.ylabel('$\phi(z)$')
>>> plt.show()
```

Po uruchomieniu powyższego kodu naszym oczom powinien ukazać się wykres zaprezentowany na rysunku 3.2.

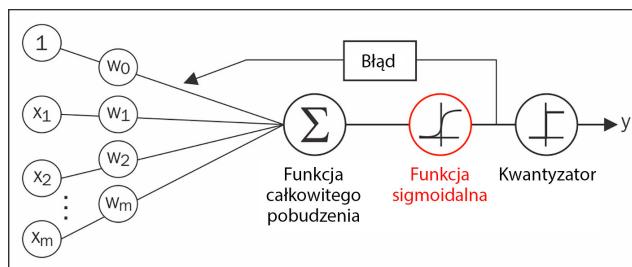
Widzimy, że funkcja  $\phi(z)$  dąży do  $1$ , gdy  $z$  dąży do nieskończoności ( $z \rightarrow \infty$ ), ponieważ  $e^{-z}$  uzupełnia bardzo małe wartości wraz ze wzrostem wartości  $z$ . Analogicznie funkcja  $\phi(z)$  dąży do  $0$ , gdy  $z \rightarrow -\infty$  wskutek dużych wartości mianownika. Podsumowując, omawiana funkcja sigmoidalna przyjmuje wartości w postaci liczb rzeczywistych i przekształca je w wartości z przedziału  $[0, 1]$ , a punkt przecięcia krzywej z osią rzędnych następuje w sytuacji, gdy  $\phi(z) = 0.5$ .

Aby pojąć model regresji logistycznej w bardziej intuicyjny sposób, możemy powiązać go z implementacją Adaline opisaną w rozdziale 2., „Trenowanie algorytmów uczenia maszynowego”.



Rysunek 3.2. Wykres funkcji logistycznej

w celach klasyfikacji". W przypadku adaptacyjnego neuronu liniowego korzystaliśmy z funkcji tożsamościowej  $\phi(z) = z$  jako funkcji aktywacji. W modelu regresji logistycznej rolę funkcji aktywacji przejmuje funkcja sigmoidalna, co zostało ukazane na rysunku 3.3.



Rysunek 3.3. Schemat modelu regresji logistycznej

Wynik funkcji sigmoidalnej zostaje następnie następnie zinterpretowany jako prawdopodobieństwo przynależności danej próbki do klasy 1,  $\phi(z) = P(y = 1 | x; w)$ , gdzie  $x$  to cechy tej próbki pomnożone przez wagi  $w$ . Jeśli np. obliczymy wartość funkcji  $\phi(z) = 0,8$  dla danej próbki kwiatu, to szansa na przynależność tej próbki do klasy *Iris-versicolor* wynosi 80%. Analogicznie prawdopodobieństwo przynależności tej próbki do gatunku *setosa* możemy wyliczyć ze wzoru  $P(y = 0 | x; w) = 1 - P(y = 1 | x; w) = 0,2$  lub 20%. Prognozowane prawdopodobieństwo może zostać następnie przekształcone na binarny wynik za pomocą kwantyzatora (funkcji skoku jednostkowego):

$$\hat{y} = \begin{cases} 1 & \text{jeśli } \phi(z) \geq 0,5 \\ 0 & \text{jeśli } \phi(z) < 0,5 \end{cases}$$

Jeśli przyjrzymy się widocznemu na rysunku 3.3 wykresowi funkcji sigmoidalnej, zauważymy, że powyższe założenie jest równoznaczne:

$$\hat{y} = \begin{cases} 1 & \text{jeśli } z \geq 0 \\ 0 & \text{jeśli } z < 0 \end{cases}$$

Istnieje rzeczywiście wiele zastosowań, w których okazuje się przydatne nie tylko przewidywanie etykiet klas, lecz również szacowanie prawdopodobieństwa przynależności określonych instancji do wyznaczonych grup. Przykładowo regresja logistyczna jest używana w prognozowaniu pogody do przewidywania opadów w danym dniu, ale także do określania szansy wystąpienia deszczu. W analogiczny sposób metoda ta pozwala wyznaczyć prawdopodobieństwo występowania danej choroby u pacjenta na podstawie znanych objawów, dlatego właśnie regresja logistyczna jest bardzo chętnie wykorzystywany modelem w medycynie.

## Wyznaczanie wag logistycznej funkcji kosztu

Wiemy już, jak można wykorzystać model regresji logistycznej do prognozowania prawdopodobieństwa i etykiet klas. Przyjrzymy się teraz pobiieżnie parametrom tego modelu, np. wagom  $w$ . W poprzednim rozdziale zdefiniowaliśmy funkcję kosztu bazującą na sumie kwadratów błędów:

$$J(\mathbf{w}) = \sum_i \frac{1}{2} (\phi(z^{(i)}) - y^{(i)})^2$$

Dokonaliśmy minimalizacji w celu wyuczenia wag  $w$  dla modelu Adaline. Aby wyprowadzić funkcję kosztu dla regresji logistycznej, zdefiniujemy najpierw wiarygodność  $L$ , którą będziemy chcieli maksymalizować w trakcie tworzenia modelu przy założeniu, że wszystkie próbki znajdujące się w zbiorze danych są od siebie wzajemnie niezależne. Wzór wygląda następująco:

$$L(\mathbf{w}) = P(\mathbf{y} \mid \mathbf{x}; \mathbf{w}) = \prod_{i=1}^n P(y^{(i)} \mid x^{(i)}; \mathbf{w}) = \prod_{i=1}^n (\phi(z^{(i)}))^{y^{(i)}} (1 - \phi(z^{(i)}))^{1-y^{(i)}}$$

W praktyce łatwiej jest maksymalizować logarytm (naturalny) równania, co możemy nazwać zlogarytmowaną funkcją wiarygodności:

$$l(\mathbf{w}) = \log L(\mathbf{w}) = \sum_{i=1}^n [y^{(i)} \log(\phi(z^{(i)})) + (1 - y^{(i)}) \log(1 - \phi(z^{(i)}))]$$

Po pierwsze, wprowadzenie funkcji logarytmicznej zmniejsza ryzyko niedomiaru obliczeniowego, co może występować w przypadku bardzo małych wartości wiarygodności. Po drugie, możemy przekształcić iloczyn czynników w ich sumę, przez co łatwiej nam będzie obliczyć pochodną tej funkcji, co być może pamiętasz z lekcji matematyki.

Do maksymalizacji zlogarytmowanej funkcji wiarygodności możemy teraz wykorzystać jakiś algorytm optymalizacji, np. wzrostu gradientu. Ewentualnie przedstawmy zlogarytmowaną funkcję wiarygodności jako funkcję kosztu  $J$ , którą będziemy w stanie zminimalizować za pomocą algorytmu gradientu prostego, omówionego w rozdziale 2., „Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji”:

$$J(\mathbf{w}) = \sum_{i=1}^n [-y^{(i)} \log(\phi(z^{(i)})) - (1-y^{(i)}) \log(1-\phi(z^{(i)}))]$$

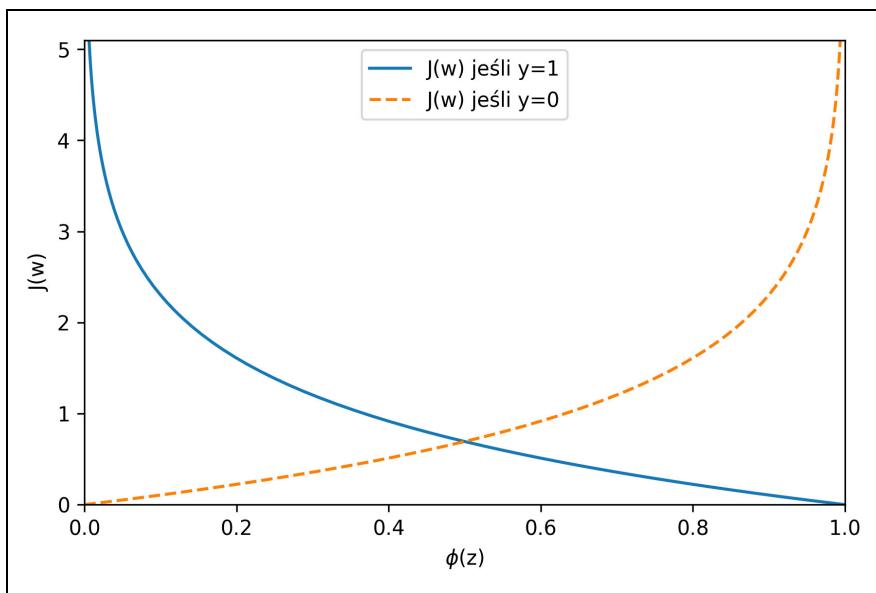
Aby lepiej pojąć tę funkcję kosztu, obliczmy koszt jednopróbkowego wystąpienia:

$$J(\phi(z), y; \mathbf{w}) = -y \log(\phi(z)) - (1-y) \log(1-\phi(z))$$

Widzimy, że pierwszy człon równania będzie wynosił 0, jeśli  $y = 0$ , a drugi człon osiągnie zerową wartość, gdy  $y = 1$ :

$$J(\phi(z), y; \mathbf{w}) = \begin{cases} -\log(\phi(z)) & \text{jesli } y = 1 \\ -\log(1-\phi(z)) & \text{jesli } y = 0 \end{cases}$$

Na rysunku 3.4 przedstawiłem koszt klasyfikacji jednopróbkowego wystąpienia dla różnych wartości  $\phi(z)$ .



Rysunek 3.4. Wykres funkcji kosztu jednopróbkowej instancji dla różnych wartości funkcji  $\phi(z)$

Jak widać, koszt dąży do 0 (ciągła niebieska linia), jeżeli poprawnie przewidzimy, że próbka należy do klasy 1. Analogicznie możemy zauważyć, że dla osi y koszt również dąży do 0, jeżeli

poprawnie będziemy prognozować  $y = 0$  (linia przerywana). Jeśli jednak predykcje są błędne, koszt będzie dążył ku nieskończoności. Morał z tego taki, że karcimy niewłaściwe prognozy, zwiększając wartość funkcji kosztu.

## Uczenie modelu regresji logistycznej za pomocą biblioteki scikit-learn

Gdybyśmy chcieli samodzielnie zaimplementować algorytm regresji logistycznej, wystarczyłoby zmodyfikować funkcję kosztu  $J$  w implementacji Adaline opisanej w rozdziale 2., „Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji”, w następujący sposób:

$$J(\mathbf{w}) = -\sum_i y^{(i)} \log(\phi(z^{(i)})) + (1 - y^{(i)}) \log(1 - \phi(z^{(i)}))$$

Z pomocą powyższego wzoru możemy obliczyć koszt klasyfikacji wszystkich próbek uczących w danej epoce, a zatem stworzyć działający model regresji logistycznej. Biblioteka scikit-learn zawiera jednak znakomicie zoptymalizowaną wersję tego algorytmu, która od razu obsługuje klasyfikację wieloklasową, dlatego zrezygnujemy z własnoręcznej implementacji kodu i wykorzystamy klasę `sklearn.linear_model.LogisticRegression`, a do tego znamą już nam metodę `fit` do uczenia modelu na standaryzowanym zestawie danych uczących:

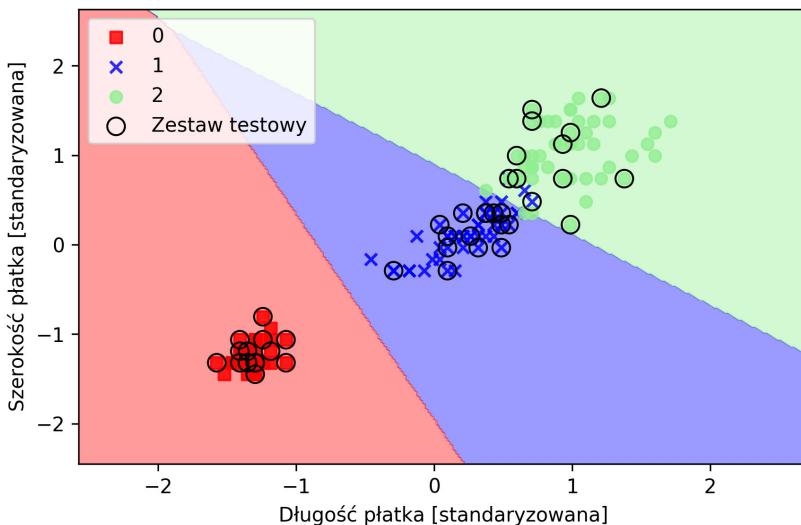
```
>>> from sklearn.linear_model import LogisticRegression
>>> lr = LogisticRegression(C=1000.0, random_state=0)
>>> lr.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                         y_combined, classifier=lr,
...                         test_idx=range(105,150))
>>> plt.xlabel('Długość płatka [standaryzowana]')
>>> plt.ylabel('Szerokość płatka [standaryzowana]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Po wytrenowaniu modelu na danych uczących wygenerowaliśmy wykres regionów decyzyjnych, próbek uczących i testowych, który został ukazany na rysunku 3.5.

Po przyjrzeniu się powyższemu listingowi wykorzystanemu do uczenia modelu `LogisticRegression` możemy się zastanawiać, do czego służy tajemniczy parametr  $C$ . Niebawem powrócimy do tego zagadnienia, najpierw jednak chciałbym w kolejnym ustępie poruszyć temat nadmiernego dopasowania i regularyzacji.

Mogliśmy dodatkowo przewidzieć prawdopodobieństwo przynależności określonych instancji do wyznaczonych grup za pomocą metody `predict_proba`. Mogliśmy np. prognozować prawdopodobieństwo natrafienia w pierwszej kolejności na próbce Iris-setosa:

```
>>> lr.predict_proba(X_test_std[0,:])
... .reshape(1, -1)
```



Rysunek 3.5. Wykres wyników uczenia po zastosowaniu algorytmu regresji logistycznej wbudowanego w bibliotekę scikit-learn

Metoda ta zwraca nam następującą tablicę:

```
array([[ 2.05743774e-11,    6.31620264e-02,   9.36837974e-01]])
```

Dzięki tej tablicy wiemy, że istnieje niemal 93,7% szansy na to, że kolejna próbka będzie należała do klasy Iris-virginica, 6,3% prawdopodobieństwa na to, że będzie nią Iris-versicolor, a szanse na pojawienie się próbki Iris-setosa są znikome.

Możemy udowodnić, że aktualizacja wag za pomocą metody gradientu prostego w regresji logistycznej jest w istocie taka sama jak w równaniu wykorzystanym w implementacji Adaline omówionej w rozdziale 2., „Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji”. Obliczmy najpierw pochodną cząstkową zlogarytmowanej funkcji wiarygodności dla  $j$ -tej wagi:

$$\frac{\partial}{\partial w_j} l(\mathbf{w}) = \left( y \frac{1}{\phi(z)} - (1-y) \frac{1}{1-\phi(z)} \right) \frac{\partial}{\partial w_j} \phi(z)$$

Zanim przejdziemy dalej, policzmy również pochodną cząstkową funkcji sigmoidalnej:

$$\begin{aligned} \frac{\partial}{\partial z} \phi(z) &= \frac{\partial}{\partial z} \frac{1}{1+e^{-z}} = \frac{1}{(1+e^{-z})^2} e^{-z} = \frac{1}{1+e^{-z}} \left( 1 - \frac{1}{1+e^{-z}} \right) = \\ &= \phi(z)(1-\phi(z)) \end{aligned}$$

Teraz możemy podstawić w pierwszym wzorze  $\frac{\partial}{\partial z}\phi(z) = \phi(z)(1 - \phi(z))$ , dzięki czemu otrzymamy:

$$\begin{aligned} & \left( y \frac{1}{\phi(z)} - (1-y) \frac{1}{1-\phi(z)} \right) \frac{\partial}{\partial w_j} \phi(z) = \\ & = \left( y \frac{1}{\phi(z)} - (1-y) \frac{1}{1-\phi(z)} \right) \phi(z)(1 - \phi(z)) \frac{\partial}{\partial w_j} z = \\ & = (y(1 - \phi(z)) - (1-y)\phi(z))x_j = \\ & = (y - \phi(z))x_j \end{aligned}$$

Pamiętaj, że celem jest znalezienie wag maksymalizujących zlogarytmowaną funkcję wiarygodności, dzięki czemu będziemy mogli aktualizować wagi w następujący sposób:

$$w_j := w_j + \eta \sum_{i=1}^n (y^{(i)} - \phi(z^{(i)}))x_j^{(i)}$$

Aktualizujemy jednocześnie wszystkie wagi, dlatego możemy zapisać regułę aktualniania w ogólnej postaci:

$$\mathbf{w} := \mathbf{w} + \Delta \mathbf{w}$$

Definiujemy  $\Delta \mathbf{w}$  jako:

$$\Delta \mathbf{w} = -\eta \nabla l(\mathbf{w})$$

Maksymalizowanie zlogarytmowanej funkcji wiarygodności jest równoznaczne z minimalizowaniem zdefiniowanej wcześniej funkcji kosztu  $J$ , zatem możemy zapisać regułę aktualizacji gradientu prostego w poniższy sposób:

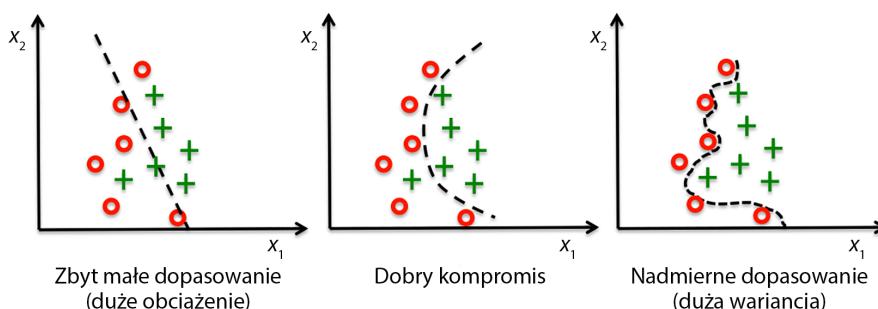
$$\begin{aligned} \Delta w_j &= -\eta \frac{\partial J}{\partial w_j} = \eta \sum_{i=1}^n (y^{(i)} - \phi(z^{(i)}))x_j^{(i)} \\ \mathbf{w} &:= \mathbf{w} + \Delta \mathbf{w}, \quad \Delta \mathbf{w} = -\eta \nabla l(\mathbf{w}) \end{aligned}$$

Jest to równoważne reguły gradientu prostego omówionej w rozdziale 2., „Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji”.

## Zapobieganie nadmiernemu dopasowaniu za pomocą regularyzacji

Jednym z częstych problemów występujących w uczeniu maszynowym jest nadmierne dopasowanie — model sprawdza się dobrze na danych uczących, ale nie uogólnia wyuczonej reguły na nieznane dane (dane testowe). Model cechujący się przetrenowaniem bywa również nazywany modelem o dużej wariancji, gdyż często wywołuje je zbyt duża liczba parametrów, przez co algorytm jest zbyt złożony, aby przetwarzanie określony zestaw danych. Zdarzają się sytuacje przeciwne, **zbyt małe dopasowanie** (ang. *underfitting*), definiowane także jako model o dużym obciążeniu, co oznacza, że algorytm nie jest wystarczająco skomplikowany, aby wychwytywać skutecznie wzorce w zestawie danych uczących, a w konsekwencji jest mało wydajny wobec nieznanych informacji.

Do tej pory zajmowaliśmy się wyłącznie modelami klasyfikacji liniowej, ale problem nadmiernego i zbyt małego dopasowania najłatwiej ukazać na przykładzie bardziej złożonych, nielinowych granic decyzyjnych, zaprezentowanych na rysunku 3.6.



Rysunek 3.6. Przykłady nadmiernego i zbyt małego dopasowania

Wariancja służy do mierzenia jednorodności (lub różnorodności) modelu prognozowania dla danego wystąpienia próbki w sytuacji wielokrotnego uczenia modelu na np. różnych podzbiorach zestawu uczącego. Możemy stwierdzić wtedy, że model jest wrażliwy na losowość danych uczących. Obciążenie stanowi przeciwieństwo wariancji: pozwala ono mierzyć ogólną niezgodność przewidywania z właściwymi wartościami po wielokrotnym wytrenowaniu modelu na różnych zestawach danych uczących; wartość obciążenia stanowi miarę błędu systematycznego niezależnego od losowości.

Jednym ze sposobów znalezienia dobrego kompromisu pomiędzy wariancją a obciążeniem jest dostrojenie złożoności modelu poprzez regularyzację. Jest to bardzo dobra metoda pozwalająca na kontrolowanie współliniowości (dużej wzajemnej korelacji cech), odfiltrowanie szumów oraz zapobieganie przetrenowaniu. Istotą regularyzacji jest wprowadzenie dodatkowych danych (obciążenia), karcących olbrzymie wartości wag. Najpopularniejszą formą regularyzacji jest tzw. **regularyzacja L2**, zwana także czasami rozpadem wag, którą można sformułować następująco:

$$\frac{\lambda}{2} \|\mathbf{w}\|^2 = \frac{\lambda}{2} \sum_{j=1}^m w_j^2$$

Tutaj  $\lambda$  to tzw. parametr regularyzacji.

Regularyzacja stanowi kolejny powód, dla którego skalowanie cech (np. standaryzacja) jest takie istotne. Aby regularyzacja mogła zostać właściwie przeprowadzona, musimy sprawić, żeby wszystkie cechy zostały dostosowane do jednolitej skali.

W celu przeprowadzenia regularyzacji wystarczy dodać odpowiedni czynnik do uprzednio zdefiniowanej funkcji kosztu, który posłuży do zmniejszania wag:

$$J(\mathbf{w}) = \sum_{i=1}^n \left[ -y^{(i)} \log(\phi(z^{(i)})) - (1 - y^{(i)}) \log(1 - \phi(z^{(i)})) \right] + \frac{\lambda}{2} \|\mathbf{w}\|^2$$

Możemy teraz za pomocą parametru regularyzacji  $\lambda$  kontrolować trenowanie na podstawie danych uczących przy jednoczesnym utrzymywaniu niskich wartości wag. Zwiększając wartość  $\lambda$ , wzmacniamy siłę regularyzacji.

Zaimplementowany w klasie `LogisticRegression` biblioteki scikit-learn parametr  $C$  stanowi element konwencji wywodzącej się z maszyn wektorów nośnych, którymi zajmiemy się w następnym podrozdziale. Krótko mówiąc, parametr  $C$  stanowi odwrotność parametru  $\lambda$ :

$$C = \frac{1}{\lambda}$$

Możemy więc przekształcić regularyzowaną funkcję kosztu dla modelu regresji logistycznej w następujący sposób:

$$J(\mathbf{w}) = C \left[ \sum_{i=1}^n \left( -y^{(i)} \log(\phi(z^{(i)})) - (1 - y^{(i)}) \log(1 - \phi(z^{(i)})) \right) \right] + \frac{1}{2} \|\mathbf{w}\|^2$$

Zatem zmniejszanie odwrotności regularyzacji  $C$  oznacza, że zwiększamy siłę regularyzacji, co możemy zaobserwować, rysując wykres ścieżki regularyzacji L2 dla dwóch współczynników wag:

```
>>> weights, params = [], []
>>> for c in np.arange(-5, 5):
...     lr = LogisticRegression(C=10**c, random_state=0)
...     lr.fit(X_train_std, y_train)
...     weights.append(lr.coef_[1])
...     params.append(10**c)
>>> weights = np.array(weights)
>>> plt.plot(params, weights[:, 0],
...            label='Długość pątaka')
>>> plt.plot(params, weights[:, 1], linestyle='--',
```

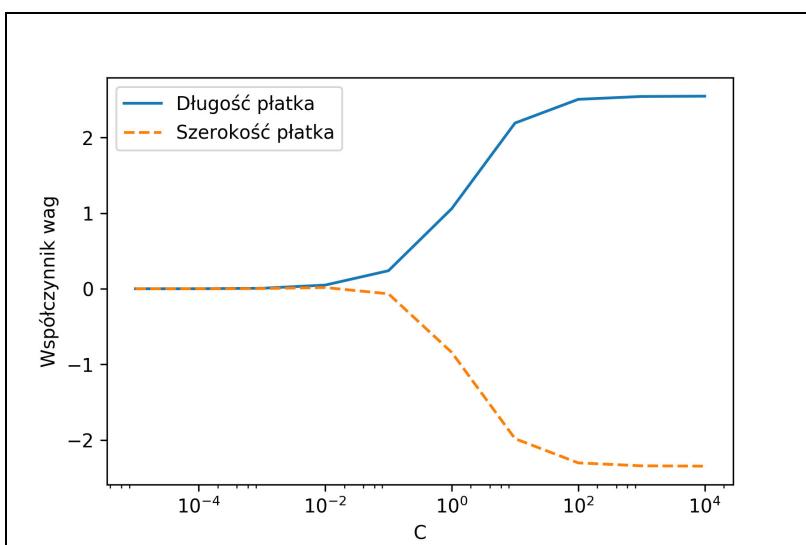
```

...      label='Szerokość płatka')
>>> plt.ylabel('Współczynnik wag')
>>> plt.xlabel('C')
>>> plt.legend(loc='upper left')
>>> plt.xscale('log')
>>> plt.show()

```

Dzięki powyższemu kodowi wytrenowaliśmy 10 modeli logistycznych, korzystając z różnych wartości parametru  $C$ . W celu zachowania jasności wykresu wyświetcone zostały jedynie współczynniki wagowe z klasy 1 przeciw pozostałym klasyfikatorom. Pamiętaj, że do klasyfikacji wieloklasowej używamy techniki OvR.

Jak widać na rysunku 3.7, współczynniki wag maleją w sytuacji zmniejszania wartości parametru  $C$ , tzn. w trakcie zwiększania siły regularyzacji.



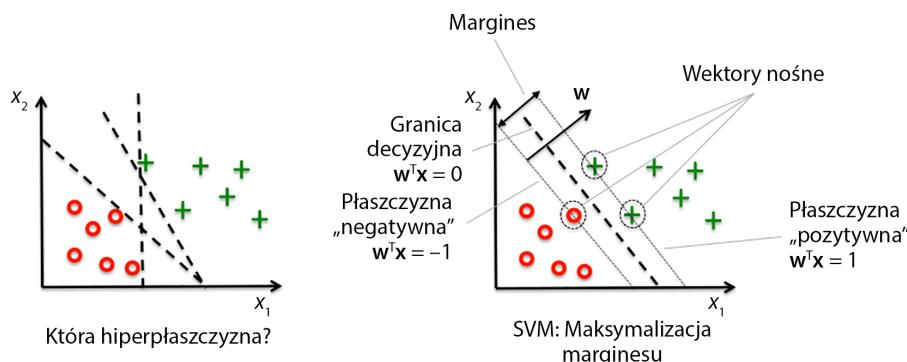
Rysunek 3.7. Wykres regularyzacji

Dokładny opis każdego algorytmu klasyfikacyjnego wykracza poza ramy niniejszej książki, dlatego wszystkim osobom pragnącym dowiedzieć się więcej na temat regresji logistycznej gorąco polecam pozycję autorstwa dra Scotta Menarda *Logistic Regression: From Introductory to Advanced Concepts and Applications*, wydaną nakładem wydawnictwa SAGE Publications<sup>1</sup>.

<sup>1</sup> Potężnym źródłem informacji na omawiany temat napisanym w języku polskim jest książka *Modele regresji logistycznej. Zastosowania w medycynie, naukach przyrodniczych i społecznych* autorstwa Andrzeja Stanisza (StatSoft Polska, Kraków 2016) — przyp. tłum.

# Wyznaczanie maksymalnego marginesu za pomocą maszyn wektorów nośnych

Kolejnym potężnym i szeroko stosowanym algorytmem uczenia jest **maszyna wektorów nośnych** (ang. *support vector machine* — SVM), którą możemy uznać za rozwinięcie modelu perceptronu. Dzięki algorytmowi perceptronu minimalizowaliśmy błędy nieprawidłowej klasyfikacji. Z kolei podstawowym celem optymalizacyjnym modelu SVM jest maksymalizacja **marginesu**. Parametr ten definiujemy jako odległość pomiędzy hiperprzestrzenią rozdzielającą (granicą decyzyjną) a najbliższymi próbami uczącymi (tzw. **wektorami nośnymi**). Koncepcja ta została zaprezentowana na rysunku 3.8.



Rysunek 3.8. Model maszyny wektorów nośnych

## Teoretyczne podłoże maksymalnego marginesu

Dążymy do uzyskania granic decyzyjnych o szerokich marginesach, ponieważ takie modele są bardziej odporne na błędy uogólniania w porównaniu z algorytmami o wąskich marginesach, cechują się bowiem większą podatnością na przetrenowanie. Aby intuicyjnie pojąć koncepcję maksymalizacji marginesu, przyjrzymy się uważniej **pozytywnym i negatywnym** hiperpłaszczyznom (hiperpłaszczyznom ułożonym równolegle do granicy decyzyjnej), których wzory wyglądają następująco:

$$w_0 + \mathbf{w}^T \mathbf{x}_{pos} = 1 \quad (1)$$

$$w_0 + \mathbf{w}^T \mathbf{x}_{neg} = -1 \quad (2)$$

Gdy odejmiemy równanie (2) od równania (1), otrzymamy:

$$\Rightarrow \mathbf{w}^T (\mathbf{x}_{pos} - \mathbf{x}_{neg}) = 2$$

Możemy dokonać normalizacji o długość wektora  $w$ , który definiujemy jako:

$$\|w\| = \sqrt{\sum_{j=1}^m w_j^2}$$

Dochodzimy więc do następującego równania:

$$\frac{w^T(x_{pos} - x_{neg})}{\|w\|} = \frac{2}{\|w\|}$$

Lewą stronę powyższego równania interpretujemy jako odległość pomiędzy hiperplaszczyzną pozytywną a negatywną, czyli tzw. margines, który chcemy maksymalizować.

Zadaniem algorytmu SVM jest maksymalizacja tego marginesu poprzez maksymalizowanie wyrażenia  $\frac{2}{\|w\|}$  przy takim ograniczeniu, że próbki mają być poprawnie klasyfikowane, co można zapisać w poniższy sposób:

$$\begin{aligned} w_0 + w^T x^{(i)} &\geq 1 \text{ jeśli } y^{(i)} = 1 \\ w_0 + w^T x^{(i)} &< -1 \text{ jeśli } y^{(i)} = -1 \end{aligned}$$

Te dwa wzory mówią nam, że wszystkie negatywne próbki powinny wylądować po stronie negatywnej hiperplaszczyzny, a wszystkie próbki pozytywne — po stronie hiperplaszczyzny pozytywnej. Możemy to również napisać w krótszej postaci:

$$y^{(i)} (w_0 + w^T x^{(i)}) \geq 1 \quad \forall_i$$

W rzeczywistości łatwiej minimalizować odwrotne wyrażenie  $\frac{1}{2} \|w\|^2$ , które można obliczyć za pomocą programowania kwadratowego. Szczegółowe omówienie programowania kwadratowego wykracza poza zakres niniejszej książki, ale zainteresowane osoby mogą dowiedzieć się więcej o maszynie wektorów nośnych z książki Vladimira Vapnika *The Nature of Statistical Learning Theory*, wydanej nakładem wydawnictwa Springer Science & Business Media, lub znakomitego artykułu Chrisa J.C. Burgesa *A Tutorial on Support Vector Machines for Pattern Recognition* („Data Mining and Knowledge Discovery” 1998, nr 2 (2), s. 121 – 167).

## Rozwiązywanie przypadków nieliniowo rozdzielnnych za pomocą zmiennych uzupełniających

Nie chcę wprowadzać żadnych bardziej zaawansowanych pojęć matematycznych dotyczących klasyfikacji marginesu, muszę jednak wspomnieć o zmiennej uzupełniającej (ang. *slack variable*)  $\xi$ . Została ona zaproponowana w 1995 roku przez Vladimira Vapnika i stała się podstawą tzw. klasyfikacji miękkiego marginesu (ang. *soft-margin classification*). Motywacją wprowadzenia tej zmiennej

była potrzeba „uelastycznienia” liniowych ograniczeń podczas analizowania nielinowo rozdzielnego danych, co pozwalałoby na uzyskanie zbieżności algorytmu uczącego w obecności nieprawidłowych klasyfikacji podczas stosowania odpowiedniej metody karcenia kosztów.

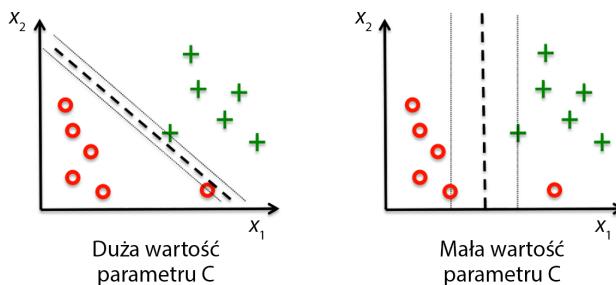
Wystarczy wstawić zmienną uzupełniającą do wzoru ograniczeń liniowych:

$$\begin{aligned} \mathbf{w}^T \mathbf{x}^{(i)} &\geq 1 - \xi^{(i)} \text{ jeśli } y^{(i)} = 1 \\ \mathbf{w}^T \mathbf{x}^{(i)} &\leq -1 + \xi^{(i)} \text{ jeśli } y^{(i)} = -1 \end{aligned}$$

Zatem nowym celem minimalizacji (zależnym od powyższych ograniczeń) staje się:

$$\frac{1}{2} \|\mathbf{w}\|^2 + C \left( \sum_i \xi^{(i)} \right)$$

Za pomocą zmiennej  $C$  możemy teraz kontrolować karę za niewłaściwą klasyfikację. Duże wartości  $C$  odpowiadają wysokim karom za błędy, z kolei przy niskich wartościach tej zmiennej nie będziemy tak rygorystyczni wobec nieprawidłowych klasyfikacji. Dzięki parametrowi  $C$  jesteśmy w stanie regulować szerokość marginesu, a zatem dostrajać kompromis pomiędzy obciążeniem a wariancją tak, jak pokazano na rysunku 3.9.



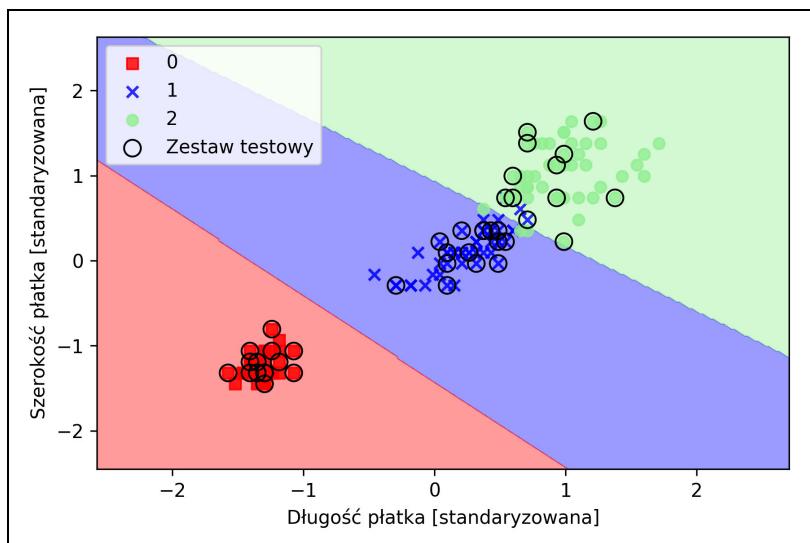
Rysunek 3.9. Wpływ zmiennej  $C$  na szerokość marginesu

Koncepcja ta jest powiązana z omawianą w kontekście regresji logistycznej regularizacją, w której obniżanie wartości parametru  $C$  zwiększa obciążenie i zmniejsza wariancję modelu.

Skoro już znamy podłożę teoretyczne liniowego modelu SVM, nauczmy maszynę wektorów nośnych klasyfikowania poszczególnych gatunków kosaćca na podstawie zbioru danych Iris:

```
>>> from sklearn.svm import SVC
>>> svm = SVC(kernel='linear', C=1.0, random_state=0)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                         y_combined, classifier=svm,
...                         test_idx=range(105,150))
>>> plt.xlabel('Długość płatka [standardyzowana]')
>>> plt.ylabel('Szerokość płatka [standardyzowana]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Regiony decyzyjne wyznaczone za pomocą algorytmu SVM zostały zaprezentowane na rysunku 3.10.



Rysunek 3.10. Efekty uczenia za pomocą modelu maszyny wektorów wektorów nośnych

#### Regresja logistyczna a model SVM

W praktycznych zadaniach klasyfikacji liniowej regresja logistyczna i liniowe maszyny wektorów wektorów nośnych często dają bardzo podobne rezultaty. Algorytm regresji logistycznej stara się zmaksymalizować wiarygodność danych uczących, przez co jest bardziej wrażliwy na odstające elementy zbiorów niż model SVM. Maszyny wektorów wektorów nośnych są kolej bardziej nastawione na punkty znajdujące się najbliżej granicy decyzyjnej (wektory nośne). Z drugiej strony zaletą regresji logistycznej jest mniejsza złożoność modelu, a zatem większa łatwość implementacji. Do tego modele regresji logistycznej można aktualizować w prosty sposób, co stanowi ważną informację w przypadku pracy z danymi strumieniowymi.

## Alternatywne implementacje w interfejsie scikit-learn

Używane w poprzednich podrozdziałach klasy Perceptron i LogisticRegression wykorzystują znakomicie zoptymalizowaną w języku C/C++ bibliotekę LIBLINEAR stworzoną na Narodowym Uniwersytecie Tajwańskim (<http://www.csie.ntu.edu.tw/~cjlin/liblinear/>). Podobnie klasa SVC zastosowana do uczenia modelu SVC korzysta z biblioteki LIBSVM, wyspecjalizowanej do obsługi maszyn wektorów wektorów nośnych (<http://www.csie.ntu.edu.tw/~cjlin/libsvm/>).

Przewagą bibliotek LIBLINEAR i LIBSVM nad natywnymi implementacjami środowiska Python jest umożliwianie niezwykle szybkiego uczenia na dużej liczbie klasyfikatorów liniowych. Czasami jednak nasze zbiory danych są zbyt duże, aby pomieściły się w pamięci komputera. Dlatego

interfejs scikit-learn zawiera również alternatywne implementacje w klasie `SGDClassifier`, która obsługuje uczenie przyrostowe poprzez metodę `partial_fit`. Filozofia kryjąca się za klasą `SGDClassifier` przypomina algorytm stochastycznego spadku wzduż gradientu, który zaimplementowaliśmy w rozdziale 2., „Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji”, dla modelu Adaline. Możemy zainicjować stochastyczny spadek wzduż gradientu dla perceptronu, regresji logistycznej i maszyny wektorów nośnych (z domyślnymi parametrami) w następujący sposób:

```
>>> from sklearn.linear_model import SGDClassifier
>>> ppn = SGDClassifier(loss='perceptron')
>>> lr = SGDClassifier(loss='log')
>>> svm = SGDClassifier(loss='hinge')
```

## Rozwiązywanie nieliniowych problemów za pomocą jądra SVM

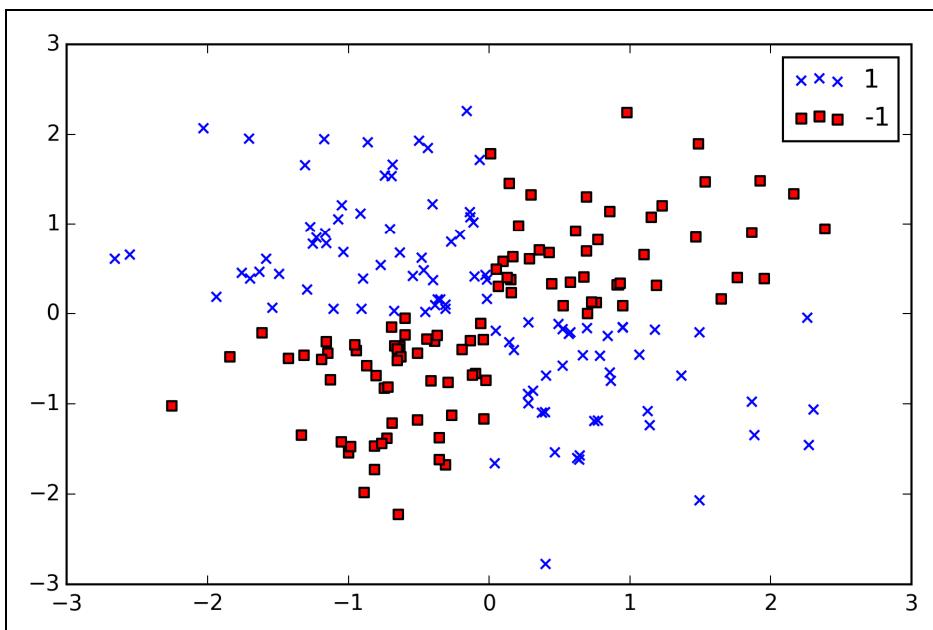
Kolejnym powodem wielkiej popularności algorytmu SVM wśród społeczności zajmującej się uczeniem maszynowym jest możliwość jego łatwej **kernelizacji** w celu rozwiązywania nieliniowych problemów klasyfikacji. Zanim zaczniemy omawiać koncepcję **jądra SVM**, zdefiniujmy i stwórzmy najpierw zbiór próbek, żeby się przekonać, jak wygląda problem nieliniowej klasyfikacji.

Za pomocą poniższego kodu stworzymy prosty zestaw danych przy zastosowaniu bramki XOR — wykorzystamy funkcję `logical_xor` będącą częścią biblioteki NumPy — gdzie przydzielimy 100 próbkom klasę 1, a kolejnym 100 punktom danych etykietę -1:

```
>>> np.random.seed(0)
>>> X_xor = np.random.randn(200, 2)
>>> y_xor = np.logical_xor(X_xor[:, 0] > 0, X_xor[:, 1] > 0)
>>> y_xor = np.where(y_xor, 1, -1)

>>> plt.scatter(X_xor[y_xor==1, 0], X_xor[y_xor==1, 1],
...               c='b', marker='x', label='1')
>>> plt.scatter(X_xor[y_xor== -1, 0], X_xor[y_xor== -1, 1],
...               c='r', marker='s', label=' -1')
>>> plt.ylim(-3.0)
>>> plt.legend()
>>> plt.show()
```

Po uruchomieniu powyższego kodu uzyskamy zbiór danych wygenerowany za pomocą bramki XOR, cechujący się losowym zaszuwaniem, co zostało zaprezentowane na rysunku 3.11.



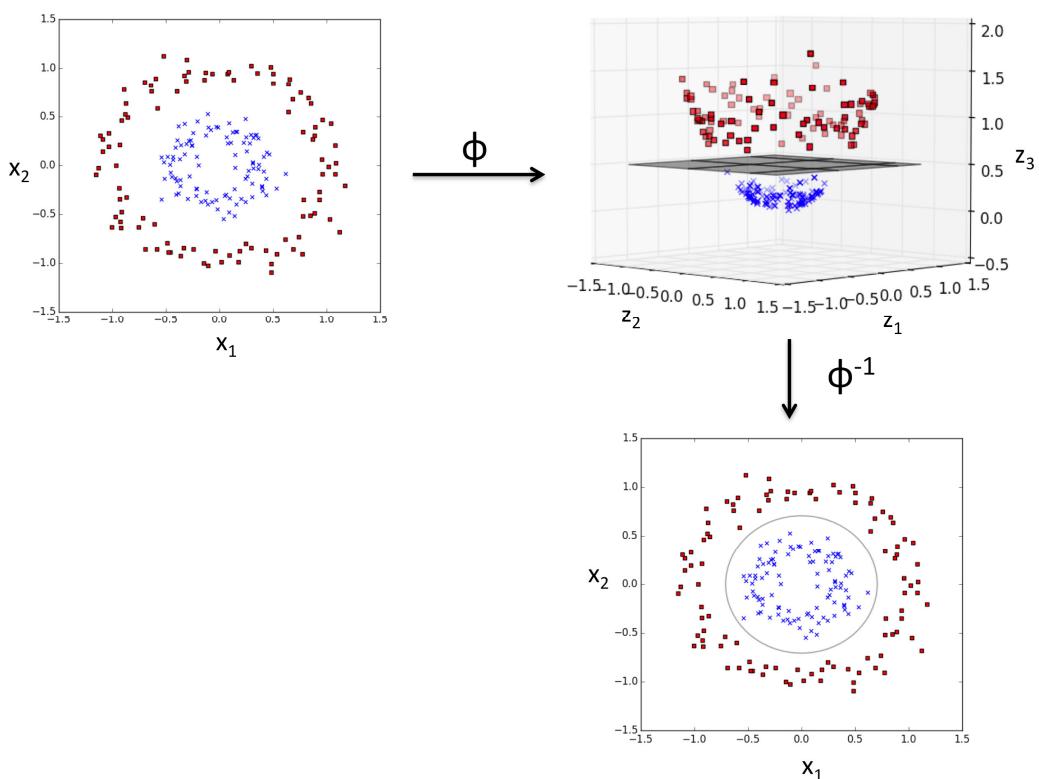
Rysunek 3.11. Zestaw danych wygenerowanych za pomocą bramki XOR

Z oczywistych względów nie jesteśmy w stanie rozdzielić tych próbek na pozytywną i negatywną klasę za pomocą liniowej hiperplanej pełniącej funkcję regionu decyzyjnego w algorytmach regresji logistycznej lub liniowego modelu SVM.

W metodach wykorzystujących funkcje jądrowe podstawową koncepcją radzenia sobie z tak nierozdzielonymi liniowo danymi jest utworzenie nieliniowych kombinacji pierwotnych cech, które za pomocą funkcji mapowania  $\phi(\cdot)$  będą rzutowane na przestrzeń mającą więcej wymiarów, gdzie staną się liniowo separowalne. Jak widać na rysunku 3.12, możemy przekształcić dwuwymiarowy zbiór danych na nową, trójwymiarową przestrzeń cech, gdzie klasy stają się rozdzielne poprzez poniższe rzutowanie:

$$\phi(x_1 \cdot x_2) = (z_1, z_2, z_3) = (x_1, x_2, x_1^2 + x_2^2)$$

Uzyskujemy w ten sposób możliwość rozdzielenia dwóch klas widocznych na rysunku 3.12 za pomocą liniowej hiperplanej, która zostaje przekształcona w nieliniową granicę decyzyjną po rzutowaniu do pierwotnej przestrzeni cech.



Rysunek 3.12. Wykorzystanie funkcji mapowania do utworzenia nieliniowej granicy decyzyjnej

## Stosowanie sztuczki z funkcją jądra do znajdowania przestrzeni rozdzielających w przestrzeni o większej liczbie wymiarów

Aby rozwiązać nieliniowy problem za pomocą algorytmu SVM, za pomocą funkcji mapowania  $\phi(\cdot)$  przenosimy dane uczące na przestrzeń cech o wyższej liczbie wymiarów, a następnie uczymy liniowy model maszyny wektorów nośnych klasyfikowania danych w tej nowej przestrzeni. Możemy potem użyć tej samej funkcji mapowania do przenoszenia nowych, nieznanych danych, które będą klasyfikowane za pomocą modelu SVM.

Jednym z problemów tego typu mapowania jest konieczność przeznaczania olbrzymiej mocy obliczeniowej do tworzenia nowych cech, zwłaszcza gdy mamy do czynienia z wielowymiarowymi danymi. W tym momencie przydaje się tzw. sztuczka z funkcją jądra (ang. *kernel trick*). Nie zagłębialiśmy się w tematykę stosowania programowania kwadratowego w uczeniu algorytmu SVM, w praktyce jednak wystarczy zastąpić iloczyn skalarny  $\mathbf{x}^{(i)T} \mathbf{x}^{(j)}$  iloczynem funkcji

$\phi(x^{(i)})^T \phi(x^{(j)})$ . By zaoszczędzić kosztowej operacji bezpośredniego obliczania tego iloczynu skalarnego pomiędzy dwoma punktami, definiujemy tzw. funkcję jądra:

$$k(x^{(i)}, x^{(j)}) = \phi(x^{(i)})^T \phi(x^{(j)})$$

Do najczęściej używanych jader należy **jądro radialnej funkcji bazowej** (ang. *Radial Basis Function kernel* — RBF kernel), nazywane także jądem gaussowskim:

$$k(x^{(i)}, x^{(j)}) = \exp\left(-\frac{\|x^{(i)} - x^{(j)}\|^2}{2\sigma^2}\right)$$

Często jest ono upraszczane do postaci:

$$k(x^{(i)}, x^{(j)}) = \exp\left(-\gamma \|x^{(i)} - x^{(j)}\|^2\right)$$

Tutaj  $\gamma = \frac{1}{2\sigma^2}$  jest swobodnym parametrem, który będziemy optymalizować.

**Funkcję jądra** możemy w dużym przybliżeniu uznać za **funkcję podobieństwa** pomiędzy parą próbek. Symbol minusa przekształca pomiar odległości w ocenę podobieństwa, która, z powodu postaci wykładniczej funkcji, będzie się mieściła w zakresie od 1 (takie same próbki) do 0 (próbki zupełnie do siebie niepodobne).

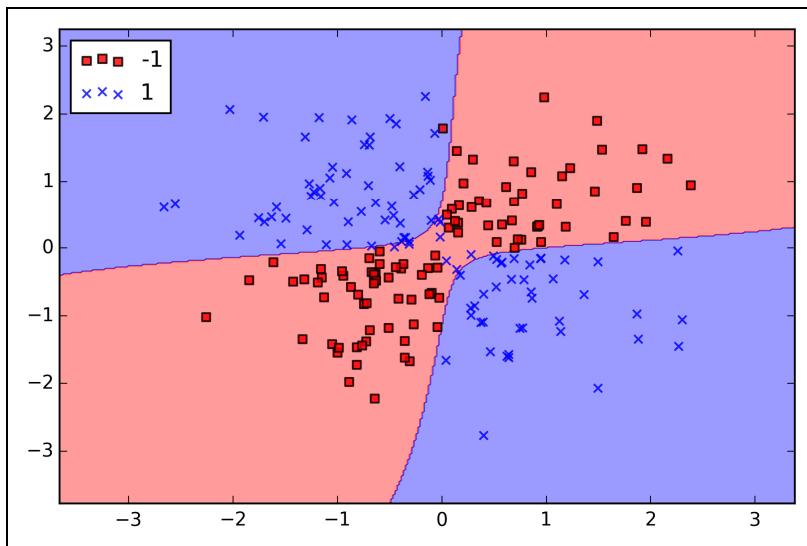
Skoro już znamy ogólne założenia sztuczki z funkcją jądra, sprawdźmy, czy jesteśmy w stanie wytrenować jądro SVM w taki sposób, żeby skutecznie wyrysowało nieliniową granicę decyzyjną rozdzielającą nasze próbki wygenerowane za pomocą bramki XOR. Wykorzystamy tu ponownie klasę SVC i zastąpimy parametr `kernel='linear'` wartością `kernel='rbf'`:

```
>>> svm = SVC(kernel='rbf', random_state=0, gamma=0.1, C=10.0)
>>> svm.fit(X_xor, y_xor)
>>> plot_decision_regions(X_xor, y_xor, classifier=svm)
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Jak widać na rysunku 3.13, jądro SVM względnie dobrze rozdziela dane XOR.

Parametr  $\gamma$  (zdefiniowany jako `gamma=0.1`) możemy uznać za obszar graniczny sfery Gaussa. Jeżeli zwiększymy wartość zmiennej  $\gamma$ , podniemyśmy również wpływ (zasięg) próbek uczących, co doprowadzi do sztywniejszych granic decyzyjnych. Aby lepiej zrozumieć ten parametr, zastosujmy jądro RBF SVM do naszego zestawu danych Iris:

```
>>> svm = SVC(kernel='rbf', random_state=0, gamma=0.2, C=1.0)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
```



Rysunek 3.13. Granica decyzyjna wygenerowana za pomocą jądra SVM

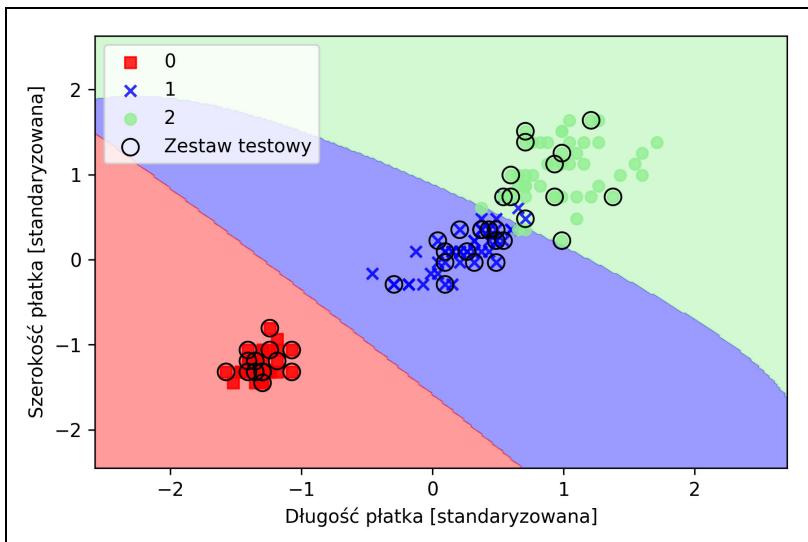
```
...                                     y_combined, classifier=svm,
...                                     test_idx=range(105,150))
>>> plt.xlabel('Długość płatka [standaryzowana]')
>>> plt.ylabel('Szerokość płatka [standaryzowana]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Wybraлиśmy względnie małą wartość parametru  $\gamma$ , dlatego uzyskamy dość elastyczne granice decyzyjne, co zostało zilustrowane na rysunku 3.14.

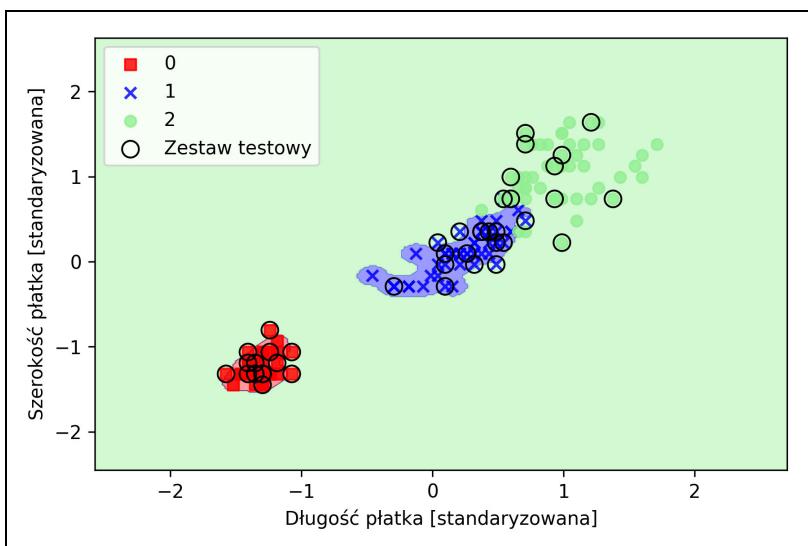
Zwiększymy teraz wartość parametru  $\gamma$  i zobaczymy, jaki będzie to miało wpływ na granice decyzyjne:

```
>>> svm = SVC(kernel='rbf', random_state=0, gamma=100.0, C=1.0)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                         y_combined, classifier=svm,
...                         test_idx=range(105,150))
>>> plt.xlabel('Długość płatka [standaryzowana]')
>>> plt.ylabel('Szerokość płatka [standaryzowana]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Na rysunku 3.15 widzimy, że po wprowadzeniu dużej wartości parametru  $\gamma$  granice decyzyjne wokół klas 0 i 1 stają się znacznie ciasniejsze.



Rysunek 3.14. Nieliniowe granice decyzyjne uzyskane za pomocą niskiej wartości parametru  $\gamma$



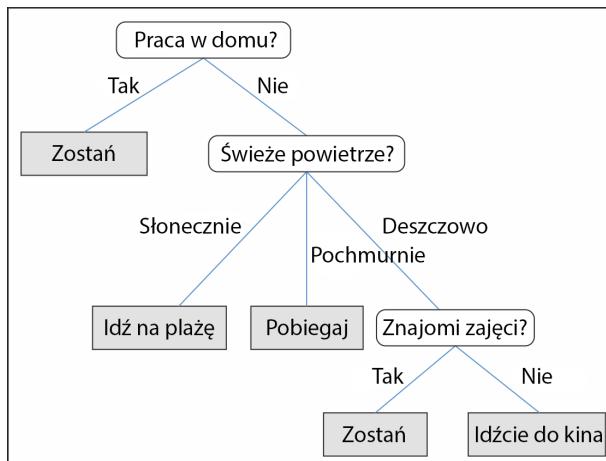
Rysunek 3.15. Wpływ zwiększenia wartości parametru  $\gamma$  na granice decyzyjne

Chociaż tak skonfigurowany model znakomicie dopasowuje dane uczące, klasyfikator prawdopodobnie będzie generował duży błąd nieprawidłowej klasyfikacji wobec nieznanych danych, dlatego optymalizacja parametru  $\gamma$  odgrywa również niebagatelną rolę w kontroli nadmiernego dopasowania.

## Uczenie drzew decyzyjnych

Modele drzew decyzyjnych (ang. *decision tree*) są atrakcyjne, jeżeli zadbane o odpowiednią interpretację danych. Jak sama nazwa wskazuje, możemy rozważać ten model jako klasyfikowanie danych poprzez podejmowanie decyzji na podstawie szeregu odpowiedzi.

Rozważmy przykład ukazany na rysunku 3.16, w którym wykorzystujemy drzewo decyzyjne do ustalenia zajęć w danym dniu.



Rysunek 3.16. Przykład drzewa decyzyjnego

Na podstawie cech zestawu uczącego model drzewa decyzyjnego wykorzystuje szereg pytań do określania etykiet klas próbek. Na rysunku 3.16 zostało przedstawione drzewo decyzyjne działające w zbiorze kategorii, nic jednak nie stoi na przeszkodzie, aby dostosować je do liczb rzeczywistych, takich jak tworzące zbiór danych Iris. Możemy np. po prostu zdefiniować wartość graniczną dla osi rzędnych (**szerokość działki**) i zadać binarne pytanie: „Czy szerokość działki 2,8 cm?”.

Z pomocą algorytmu decyzyjnego tworzymy korzeń drzewa i rozdzielamy dane wobec cechy mającej największy **przyrost informacji** (ang. *information gain* — IG; ten parametr zostanie dokładniej opisany w następnym ustępie). Poprzez wielokrotne iteracje możemy powtarzać procedurę rozdzielania danych w każdym potomnym węźle, aż uzyskamy same liście. Oznacza to, że wszystkie próbki w danym węźle przynależą do tej samej klasy. W praktyce rozwiązywanie to często skutkuje powstawaniem dużych, wielowęzłowych drzew, co może z łatwością prowadzić do przetrenowania. Z tego powodu zazwyczaj chcemy **przycinać** drzewo poprzez ustalenie granicy jego maksymalnej wysokości.

# Maksymalizowanie przyrostu informacji

## — osiąganie jak największych korzyści

Aby móc rozdzielać węzły zawierające najbardziej informatywne cechy, musimy zdefiniować funkcję celu, którą będziemy optymalizować za pomocą algorytmu uczenia drzewa. W tym przypadku naszą funkcją celu jest maksymalizacja przyrostu informacji w każdym rozgałęzieniu, co możemy sformułować następująco:

$$IG(D_p, f) = I(D_p) - \sum_{j=1}^m \frac{N_j}{N_p} I(D_j)$$

Parametr  $f$  to cecha, na podstawie której zostanie przeprowadzone rozgałęzianie,  $D_p$  i  $D_j$  są zestawami danych odpowiednio: nadzawanego węzła i  $j$ -tego węzła potomnego,  $I$  stanowi miarę zanieczyszczenia,  $N_p$  definiuje całkowitą liczbę próbek w węźle nadzawanym, a  $N_j$  — w  $j$ -tym węźle potomnym. Jak widać, przyrost informacji to nic innego, jak różnica pomiędzy zanieczyszczeniem węzła nadzawanego a sumą zanieczyszczeń węzłów potomnych — im niższe zanieczyszczenie tych drugich, tym większy przyrost informacji. Jednak dla uproszczenia i w celu ograniczenia kombinatorycznej przestrzeni przeszukiwania w większości bibliotek (w tym również scikit-learn) jest stosowana implementacja binarnych drzew. Oznacza to, że węzeł nadzawany rozgałęzia się na dwa węzły potomne:  $D_{lewy}$  i  $D_{prawy}$ :

$$IG(D_p, f) = I(D_p) - \frac{N_{lewy}}{N_p} I(D_{lewy}) - \frac{N_{prawy}}{N_p} I(D_{prawy})$$

W binarnych drzewach decyzyjnych trzema najpowszechniej wykorzystywanyimi miarami zanieczyszczenia (kryteriami rozgałęzień) są **wskaźnik Giniego** (ang. *Gini impurity*;  $I_G$ ), **entropia** ( $I_H$ ) oraz **błąd klasyfikacji** ( $I_E$ ). Zaczniemy od definicji entropii dla wszystkich **niepustych** klas  $p(i|t) \neq 0$ :

$$I_H(t) = -\sum_{i=1}^c p(i|t) \log_2 p(i|t)$$

Wyrażenie  $p(i|t)$  oznacza tu proporcję pomiędzy próbками należącymi do klasy  $i$  w danym węźle  $t$ . Z tego wynika, że entropia będzie wynosiła 0, jeśli wszystkie próbki w węźle będą należały do tej samej klasy, natomiast maksymalną wartość osiągnie wtedy, gdy będziemy mieli do czynienia z jednorodnym rozkładem klas. Przykładowo w binarnej konfiguracji klas entropia jest równa 0, gdy  $p(i=1|t)=1$  lub  $p(i=0|t)=0$ . Przy jednorodnym rozkładzie klas  $p(i=1|t)=0,5$  i  $p(i=0|t)=0,5$  wartość entropii wynosi 1. Możemy więc powiedzieć, że poprzez kryterium entropii próbujemy zmaksymalizować wzajemne informacje w drzewie.

Z kolei wskaźnik Giniego możemy interpretować jako kryterium służące do minimalizowania prawdopodobieństwa nieprawidłowej klasyfikacji:

$$I_G(t) = \sum_{i=1}^c p(i|t)(1-p(i|t)) = 1 - \sum_{i=1}^c p(i|t)^2$$

Podobnie jak w przypadku entropii, wskaźnik Giniego uzyskuje największą wartość, gdy klasy są między sobą idealnie wymieszane; np. dla binarnej konfiguracji klas ( $c = 2$ ):

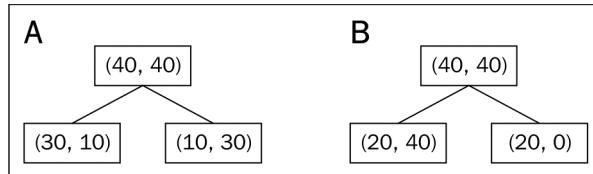
$$I_G(t) = 1 - \sum_{i=1}^2 0,5^2 = 0,5$$

W praktyce jednak zarówno wskaźnik Giniego, jak i entropia generują zazwyczaj bardzo podobne wyniki i często nie warto marnować czasu na ocenianie drzew za pomocą różnych kryteriów zanieczyszczeń, lepiej zaś eksperymentować z różnymi wartościami granicy przycinania.

Kolejną miarą zanieczyszczenia jest błąd klasyfikacji:

$$I_E(t) = 1 - \max \{p(i|t)\}$$

Jest to kryterium przydatne do przycinania, lecz nie zalecane do rozwijania drzewa, ponieważ wykazuje mniejszą czułość na zmiany w rozkładzie prawdopodobieństwa klas wewnętrz węzła. Wyjaśnię to na przykładzie dwóch możliwych scenariuszy rozgałęziania (rysunek 3.17).



Rysunek 3.17. Dwie możliwości rozgałęziania binarnego drzewa decyzyjnego

Zaczynamy od zestawu danych  $D_p$  znajdującego się w węźle nadziedzonym  $D_p$ . Na zbiór tych danych składa się 40 próbek z klasy 1 i 40 próbek z klasy 2, które rozdzielimy na dwa węzły potomne:  $D_{lewy}$  i  $D_{prawy}$ . Wartość przyrostu informacji obliczonego za pomocą błędu klasyfikacji jest taka sama ( $IG_E = 0,25$ ) w obydwu przypadkach (A i B):

$$I_E(D_p) = 1 - 0,5 = 0,5$$

$$A : I_E(D_{lewy}) = 1 - \frac{3}{4} = 0,25$$

$$A : I_E(D_{prawy}) = 1 - \frac{3}{4} = 0,25$$

$$A : IG_E = 0,5 - \frac{4}{8}0,25 - \frac{4}{8}0,25 = 0,25$$

$$B : I_E(D_{lewy}) = 1 - \frac{4}{5} = \frac{1}{3}$$

$$B : I_E(D_{\text{prawy}}) = 1 - 1 = 0$$

$$B : IG_E = 0,5 - \frac{6}{8} \times \frac{1}{3} - 0 = 0,25$$

Jednak wskaźnik Giniego bardziej sprzyja rozgałęzieniu ze scenariusza B ( $IG_G = 0,1\overline{6}$ ) niż ze scenariusza A ( $IG_G = 0,125$ ), gdyż rzeczywiście jest **czystsze**:

$$I_G(D_p) = 1 - (0,5^2 + 0,5^2) = 0,5$$

$$A : I_G(D_{\text{lewy}}) = 1 - \left( \left( \frac{3}{4} \right)^2 + \left( \frac{1}{4} \right)^2 \right) = \frac{3}{8} = 0,375$$

$$A : I_G(D_{\text{prawy}}) = 1 - \left( \left( \frac{1}{4} \right)^2 + \left( \frac{3}{4} \right)^2 \right) = \frac{3}{8} = 0,375$$

$$A : IG_G = 0,5 - \frac{4}{8} 0,375 - \frac{4}{8} 0,375 = 0,125$$

$$B : I_G(D_{\text{lewy}}) = 1 - \left( \left( \frac{2}{6} \right)^2 + \left( \frac{4}{6} \right)^2 \right) = \frac{4}{9} = 0,\overline{4}$$

$$B : I_G(D_{\text{prawy}}) = 1 - (1^2 + 0^2) = 0$$

$$B : IG_G = 0,5 - \frac{6}{8} 0,\overline{4} - 0 = 0,1\overline{6}$$

Analogicznie kryterium entropii sprzyja bardziej scenariuszowi B ( $IG_H = 0,31$ ) niż scenariuszowi A ( $IG_H = 0,19$ ):

$$I_H(D_p) = -(0,5 \log_2(0,5) + 0,5 \log_2(0,5)) = 1$$

$$A : I_H(D_{\text{lewy}}) = -\left( \frac{3}{4} \log_2\left(\frac{3}{4}\right) + \frac{1}{4} \log_2\left(\frac{1}{4}\right) \right) = 0,81$$

$$A : I_H(D_{\text{prawy}}) = -\left( \frac{1}{4} \log_2\left(\frac{1}{4}\right) + \frac{3}{4} \log_2\left(\frac{3}{4}\right) \right) = 0,81$$

$$A : IG_H = 1 - \frac{4}{8} 0,81 - \frac{4}{8} 0,81 = 0,19$$

$$B : I_H(D_{\text{lewy}}) = -\left( \frac{2}{6} \log_2\left(\frac{2}{6}\right) + \frac{4}{6} \log_2\left(\frac{4}{6}\right) \right) = 0,92$$

$$B : I_H(D_{\text{prawy}}) = 0$$

$$B : IG_H = 1 - \frac{6}{8} 0,92 - 0 = 0,31$$

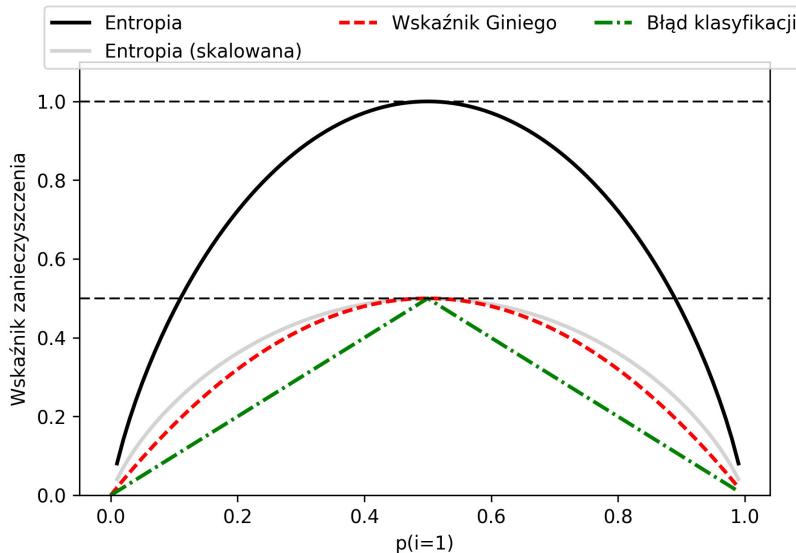
Porównajmy teraz wzrokowo omówione kryteria zanieczyszczeń — narysujmy wykres wskaźników zanieczyszczeń przy zakresie prawdopodobieństwa [0, 1] dla klasy 1. Zwróć uwagę, że dodamy również skalowaną wersję entropii (**entropia/2**), dzięki czemu przekonamy się, że wskaźnik Giniego daje wartości pośrednie pomiędzy entropią a błędem klasyfikacji. Do narysowania wykresu wykorzystamy następujący fragment kodu:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> def gini(p):
...     return (p)*(1 - (p)) + (1 - p)*(1 - (1-p))
>>> def entropy(p):
...     return - p*np.log2(p) - (1 - p)*np.log2((1 - p))
>>> def error(p):
...     return 1 - np.max([p, 1 - p])
>>> x = np.arange(0.0, 1.0, 0.01)
>>> ent = [entropy(p) if p != 0 else None for p in x]
>>> sc_ent = [e*0.5 if e else None for e in ent]
>>> err = [error(i) for i in x]
>>> fig = plt.figure()
>>> ax = plt.subplot(111)
>>> for i, lab, ls, c, in zip([ent, sc_ent, gini(x), err],
...                           ['Entropia', 'Entropia (skalowana)',
...                            'Wskaźnik Giniego',
...                            'Błąd klasyfikacji'],
...                           ['-', '--', '-.', '-.'],
...                           ['black', 'lightgray',
...                            'red', 'green', 'cyan']):
...     line = ax.plot(x, i, label=lab,
...                     linestyle=ls, lw=2, color=c)
>>> ax.legend(loc='upper center', bbox_to_anchor=(0.5, 1.15),
...             ncol=3, fancybox=True, shadow=False)
>>> ax.axhline(y=0.5, linewidth=1, color='k', linestyle='--')
>>> ax.axhline(y=1.0, linewidth=1, color='k', linestyle='--')
>>> plt.ylim([0, 1.1])
>>> plt.xlabel('p(i=1)')
>>> plt.ylabel('Wskaźnik zanieczyszczenia')
>>> plt.show()
```

Wygenerowany wykres możemy zobaczyć na rysunku 3.18.

## Budowanie drzewa decyzyjnego

Drzewa decyzyjne generują skomplikowane granice decyzyjne poprzez podzielenie przestrzeni cech na prostokątne obszary. Musimy jednak zachować ostrożność, ponieważ im większe drzewo, tym granice decyzyjne stają się bardziej złożone, co może doprowadzić do przetrenowania.



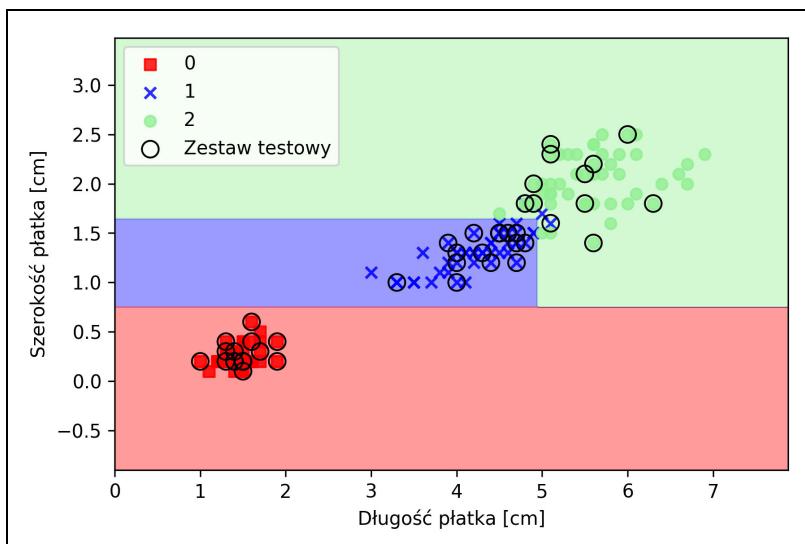
Rysunek 3.18. Porównanie indeksów zanieczyszczenia

Korzystając z interfejsu scikit-learn, stworzymy teraz drzewo decyzyjne o maksymalnej wysokości 3, a na kryterium zanieczyszczenia dobierzemy entropię. Skalowanie cech w tym przypadku przydaje się do poprawy wizualizacji, pamiętaj jednak, że nie jest ono wymagane w algorytmach drzew decyzyjnych. Kod, z którego skorzystamy, został zaprezentowany poniżej:

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> tree = DecisionTreeClassifier(criterion='entropy',
...                                 max_depth=3, random_state=0)
>>> tree.fit(X_train, y_train)
>>> X_combined = np.vstack((X_train, X_test))
>>> y_combined = np.hstack((y_train, y_test))
>>> plot_decision_regions(X_combined, y_combined,
...                         classifier=tree, test_idx=range(105,150))
>>> plt.xlabel('Długość płatka [cm]')
>>> plt.ylabel('Szerokość płatka [cm]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Po uruchomieniu powyższego kodu naszym oczom ukażą się klasyczne, równolegle do osi układu współrzędnych granice generowane przez algorytm drzewa decyzyjnego (rysunek 3.19).

Ciekawą funkcją interfejsu scikit-learn jest możliwość eksportowania drzewa decyzyjnego w formacie *.dot* po zakończeniu uczenia za pomocą aplikacji Graphviz. Program ten jest bezpłatnie dostępny na stronie <http://www.graphviz.org/> i można go pobrać m.in. na systemy Linux, Windows oraz Mac OS X.



Rysunek 3.19. Wykres granic decyzyjnych wygenerowanych za pomocą algorytmu drzewa decyzyjnego

Najpierw tworzymy plik `.dot` w interfejsie scikit-learn, korzystając z funkcji `export_graphviz` umieszczonej w podmodule `tree`:

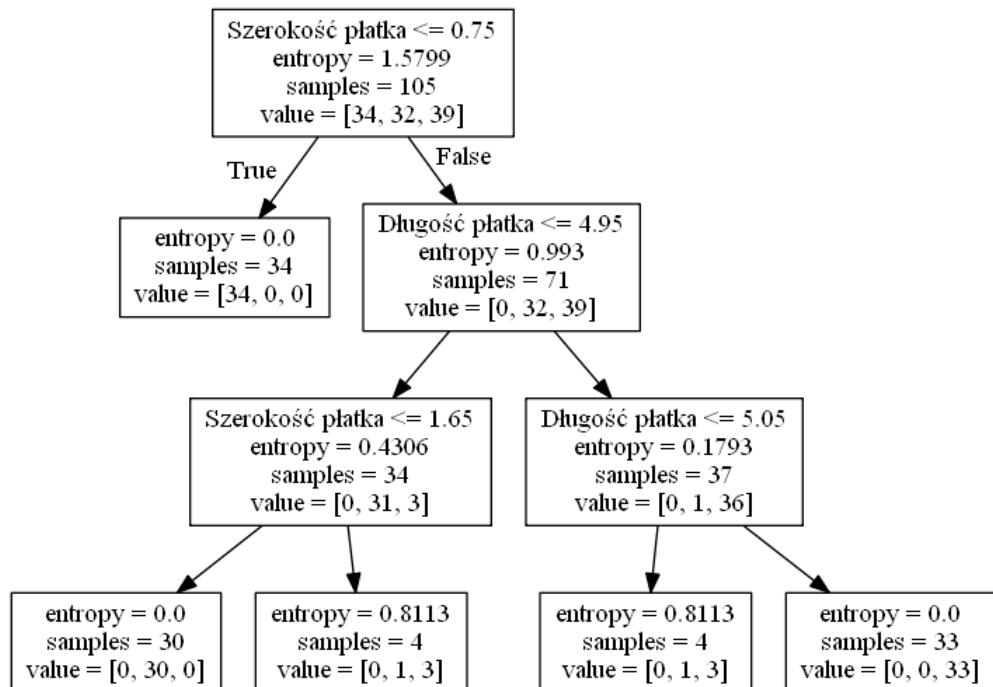
```
>>> from sklearn.tree import export_graphviz
>>> export_graphviz(tree,
...                   out_file='drzewo.dot',
...                   feature_names=['Długość płatka', 'Szerokość płatka'])
```

Po zainstalowaniu aplikacji Graphviz możemy przekonwertować plik `drzewo.dot` w obraz PNG (rysunek 3.20) poprzez wpisanie poniższego polecenia w wierszu poleceń (w katalogu zawierającym zapisany plik `drzewo.dot`)<sup>2</sup>:

```
> dot -Tpng drzewo.dot -o drzewo.png
```

Dzięki wygenerowanemu wykresowi drzewa, widocznemu na rysunku 3.20, możemy teraz w bardzo wygodny sposób prześledzić poszczególne rozgałęzienia uzyskane na podstawie danych uczących. Zaczynamy u korzenia od 105 próbek i rozdzielimy je pomiędzy 2 węzły potomne — odpowiednio 34 próbki i 71 próbek — korzystając z parametru granicznej szerokości płatka  $\leq 0,75$  cm. Już przy pierwszym rozgałęzieniu widzimy, że lewy węzeł jest czysty i zawiera wyłącznie próbki pochodzące z klasy *Iris-setosa* (entropia = 0). Kolejne podziały prawego węzła dążą do rozdzielenia klas *Iris-versicolor* i *Iris-virginica*.

<sup>2</sup> Począwszy od wersji 2.31 aplikacji Graphviz, w czasie instalacji nie są modyfikowane rejestrory oraz zmieniona środowiskowa PATH. Aby móc korzystać z tego programu poprzez środowisko Python/Anaconda lub wiersz poleceń, należy własnoręcznie dodać ścieżkę do aplikacji w zmiennej PATH (dla najnowszej wersji programu Graphviz, domyślnej ścieżki instalacji oraz systemu Windows 10 Anniversary wpis ten wygląda następująco: `C:\Program Files (x86)\Graphviz2.38\bin`) — przyp. tłum.



Rysunek 3.20. Obraz drzewa decyzyjnego wygenerowany za pomocą aplikacji Graphviz

## Łączenie słabych klasyfikatorów w silne klasyfikatory za pomocą modelu losowego lasu

W ciągu ostatniej dekady metoda losowych lasów (ang. *random forest*) zyskała znaczną popularność w środowisku uczenia maszynowego, ponieważ odznacza się dobrą skutecznością klasyfikacji, skalowalnością i łatwością stosowania. Intuicyjnie możemy rozumieć losowy las jako zespół drzew decyzyjnych. Koncepcja kryjąca się za uczeniem zespołów polega na łączeniu słabych klasyfikatorów (ang. *weak learners*) w jeden skuteczniejszy model — silny klasyfikator (ang. *strong learner*), mający mniejszy błąd uogólniania oraz wykazujący niższą wrażliwość na przetrenowanie. Algorytm losowego lasu można rozpisać na cztery proste etapy:

1. Wprowadź losowanie  $n$  próbek początkowych (ang. *bootstrap*; losowo dobierz  $n$  próbek z zestawu uczącego i wstaw za nie próbki zastępcze).
2. Wygeneruj drzewo decyzyjne na podstawie próbek początkowych. W każdym węźle:
  - a) Dobierz losowo  $d$  cech i nie zastępuj ich innymi.
  - b) Rozdziel węzeł za pomocą cechy gwarantującej najlepsze rozgałęzienie pod kątem funkcji celu (np. maksymalizując przyrost informacji).

3. Powtórz kroki 2. i 3.  $k$ -krotnie.
4. Zbierz prognozy otrzymane z każdego drzewa i przydzielaj próbkom etykiety klas poprzez **większościowe głosowanie**. Technika większościowego głosowania zostanie dokładniej opisana w rozdziale 7., „Łączenie różnych modeli w celu uczenia zespołowego”.

W porównaniu z uczeniem pojedynczych drzew decyzyjnych różnica pojawia się na etapie 2.: nie oceniamy wszystkich cech w celu określenia najlepszego rozgałęzienia drzewa, lecz dokonujemy tego jedynie na losowym podzbiorze cech.

Chociaż algorytm lasów losowych nie umożliwia interpretowania wyników w takim stopniu, jak podczas stosowania pojedynczych drzew decyzyjnych, wielką zaletą omawianego modelu jest mniejsze znaczenie doboru odpowiednich wartości hiperparametrów. Zazwyczaj nie musimy przycinać losowego lasu, ponieważ model zespołu jest dość odporny na szумy pochodzące z poszczególnych drzew. Jednym parametrem, który powinien nas interesować, jest  $k$  — liczba drzew (na etapie 3.) mających tworzyć las. Przeważnie im większa liczba drzew, tym lepsza skuteczność klasyfikatora okupiona mocą obliczeniową.

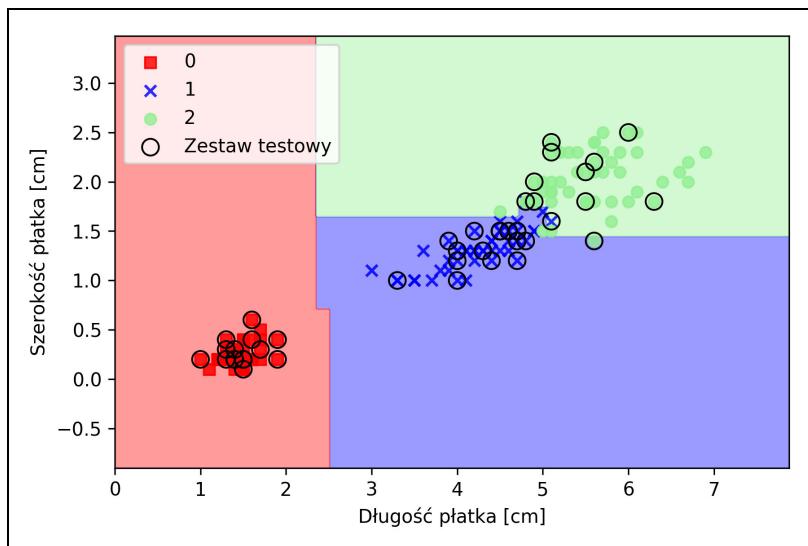
Zazwyczaj jest to rzadko wykorzystywane, ale pozostałe hiperparametry losowego lasu również można optymalizować — za pomocą technik omówionych w rozdziale 5., „Komprezja danych poprzez redukcję wymiarowości” — takie jak rozmiar  $n$  próbek początkowych (etap 1.) oraz  $d$ , czyli liczba losowo dobieranych cech do każdego rozgałęzienia (podetap 2.1). Za pomocą rozmiaru próbek  $n$  kontrolujemy kompromis pomiędzy obciążeniem a wariancją losowego lasu. Wprowadziwszy większą wartość  $n$ , zmniejszamy losowość, a zatem narażamy algorytm na nadmierne dopasowanie. Z drugiej strony możemy ograniczyć ryzyko przetrenowania, podając mniejszą wartość  $n$ , co odbija się negatywnie na skuteczności modelu. W większości przypadków wprowadzenia implementacji `RandomForestClassifier` w interfejsie scikit-learn rozmiar próbek początkowych jest równy liczbie próbek w pierwotnym zestawie uczącym, co najczęściej gwarantuje dobry kompromis pomiędzy obciążeniem a wariancją. W przypadku liczby cech  $d$  chcemy dobrać wartość, która jest mniejsza od całkowitej liczby cech z zbiorze danych uczących. Rozsądnią wartością domyślną wprowadzoną w interfejsie scikit-learn i innych implementacjach jest  $d = \sqrt{m}$ , gdzie  $m$  to liczba cech w zbiorze uczącym.

Na szczęście nie musimy samodzielnie tworzyć klasyfikatora losowego lasu z pojedynczych drzew; w interfejsie scikit-learn istnieje implementacja, którą możemy wykorzystać:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> forest = RandomForestClassifier(criterion='entropy',
...                                     n_estimators=10,
...                                     random_state=1,
...                                     n_jobs=2)
>>> forest.fit(X_train, y_train)
>>> plot_decision_regions(X_combined, y_combined,
...                         classifier=forest, test_idx=range(105,150))
>>> plt.xlabel('Długość płatka [cm]')
```

```
>>> plt.ylabel('Szerokość płatka [cm]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Po uruchomieniu powyższego kodu powinniśmy ujrzeć ukazany na rysunku 3.21 wykres regionów decyzyjnych tworzących zestawy drzew w losowym lesie.



Rysunek 3.21. Wykres regionów decyzyjnych utworzonych za pomocą algorytmu losowego lasu

Stworzyliśmy w tym przypadku losowy las składający się z 10 drzew (parametr `n_estimators`) oraz wykorzystaliśmy entropię jako kryterium zanieczyszczenia do tworzenia rozgałęzień. Mimo że generujemy bardzo mały las dla naszego niewielkiego zbioru danych, w celach demonstracyjnych wprowadziłem parametr `n_jobs`, który służy do współbieżnego uczenia modelu przy użyciu wielu rdzeni procesora (w naszym przykładzie dwóch).

## Algorytm k-najbliższych sąsiadów — model leniwego uczenia

Ostatnim algorytmem uczenia nadzorowanego, jakim się zajmiemy w tym rozdziale, jest **klasyfikator k-najbliższych sąsiadów** (ang. *k-nearest neighbor classifier* — KNN), który jest o tyle interesujący, że całkowicie różni się od wcześniej omówionych modeli.

KNN jest typowym przykładem **leniwego klasyfikatora** (ang. *lazy learner*). Nazwa leniwy nie odnosi do prostoty algorytmu, lecz do tego, że nie uczy się on funkcji dyskryminacyjnej na podstawie danych uczących, lecz stara się „zapamiętać” cały zbiór próbek.

## Modele parametryczne a nieparametryczne

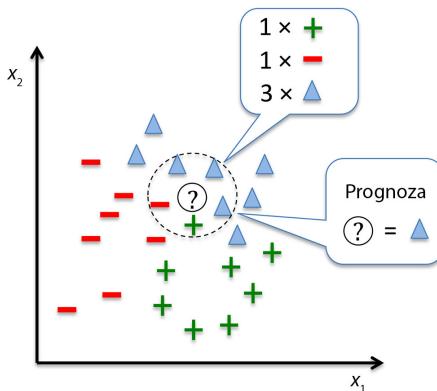
Algorytmy uczenia maszynowego możemy podzielić na modele **parametryczne** i **nieparametryczne**. Za pomocą modeli parametrycznych oszacowujemy parametry zestawu uczącego, dzięki czemu jesteśmy w stanie wytrenować funkcję, która będzie klasyfikowała nowe punkty danych bez konieczności dalszego wykorzystywania pierwotnego zbioru uczącego. Typowymi przykładami modeli parametrycznych są perceptron, regresja logistyczna oraz liniowa maszyna SVM. Z drugiej strony algorytmu nieparametrycznego nie da się opisać za pomocą ustalonego zestawu parametrów, a ich liczba wzrasta wraz z danymi uczącymi. Dotychczas omówiliśmy dwa przykładowe modele nieparametryczne: klasyfikator drzewa decyzyjnego/losowego lasu oraz algorytm jądra SVM.

Algorytm KNN należy do podkategorii modeli nieparametrycznych zwanej **uczeniem z przykładów** (ang. *instance-based learning*). Modele tego typu charakteryzują się zapamiętywaniem zestawu danych uczących, natomiast leniwe uczenie stanowi szczególny przypadek uczenia z przykładów, gdyż w tym przypadku koszt uczenia wynosi 0.

Algorytm KNN jest sam w sobie bardzo prosty i można go podsumować następująco:

1. Wybierz jakąś wartość parametru  $k$  i metrykę odległości.
  2. Znajdź  $k$  najbliższych sąsiadów próbki, którą chcesz sklasyfikować.
  3. Przydziel etykietę klasy poprzez głosowanie większościowe.

Na rysunku 3.22 pokazuję, w jaki sposób nowy punkt danych (?) otrzymuje etykietę klasy — trójkąt — na podstawie większościowego głosowania pomiędzy pięcioma najbliższymi sąsiadami tej próbki.



Rysunek 3.22. Model k-najbliższych sąsiadów

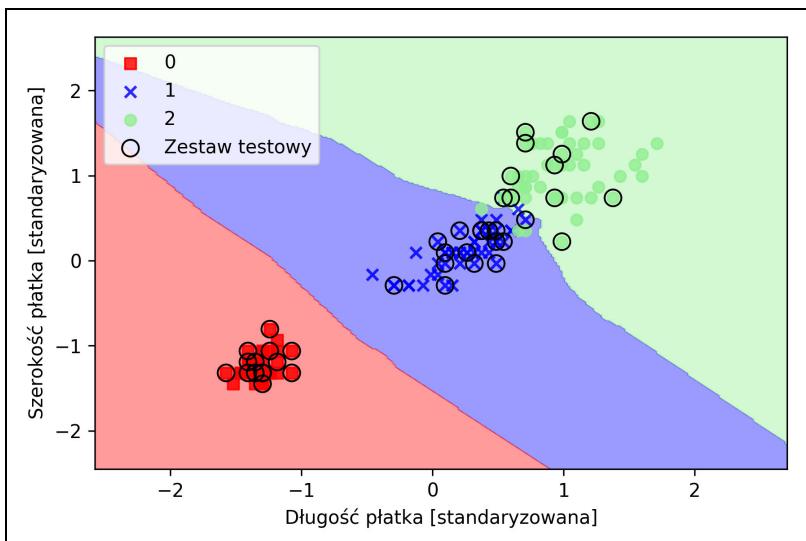
Na podstawie wybranej metryki odległości algorytm KNN wyszukuje w zestawie danych uczących  $k$  próbek znajdujących się najbliżej klasyfikowanego punktu lub wykazujących największe podobieństwo do niego. Etykieta klasy tej próbki zostaje określona poprzez większościowe głosowanie przeprowadzone pomiędzy  $k$  najbliższych sąsiadów.

Największą zaletą takiego pamięciowego algorytmu jest natychmiastowe adaptowanie się klasyfikatora w trakcie pobierania nowych danych uczących. Równoważy to jednak główna wada, polegająca na liniowym wzroście złożoności obliczeniowej wraz z liczbą próbek uczących (naj-gorszy scenariusz) — wyjątkiem jest niska wymiarowość (liczba cech) zbioru danych oraz zwiększenie skuteczności algorytmu poprzez stosowanie zoptymalizowanych struktur danych, takich jak drzew KD (J.H. Friedman, J.L. Bentley i R. Finkel, *An Algorithm for Finding Best Matches in Logarithmic Expected Time*, „ACM Transactions on Mathematical Software [TOMS]” 1977, nr 3 (3), s. 209 – 226). Poza tym nie możemy odrzucać żadnych danych uczących, ponieważ w tym algorytmie nie występuje proces **uczenia**. Zatem w przypadku dużych zbiorów danych pojawia się problem z pojemnością nośników.

Zaimplementujmy teraz model KNN poprzez interfejs scikit-learn oraz przy użyciu metryki euklidesowej:

```
>>> from sklearn.neighbors import KNeighborsClassifier
>>> knn = KNeighborsClassifier(n_neighbors=5, p=2,
...                                metric='minkowski')
>>> knn.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std, y_combined,
...                        classifier=knn, test_idx=range(105,150))
>>> plt.xlabel('Długość płatka [standaryzowana]')
>>> plt.ylabel('Szerokość płatka [standaryzowana]')
>>> plt.show()
```

Wyszukawszy za pomocą tego algorytmu pięciu sąsiadów dla naszego zestawu danych, uzyskujemy w miarę gładką granicę decyzyjną, co zostało zaprezentowane na rysunku 3.23.



Rysunek 3.23. Granica decyzyjna wygenerowana za pomocą algorytmu KNN

W przypadku takiej samej liczby głosów podczas głosowania większościowego algorytm KNN zaimplementowany w interfejsie scikit-learn faworyzuje sąsiadów znajdujących się bliżej próbki. Jeśli odległości do sąsiadów są zbliżone, algorytm wybiera pierwszą etykietę klasy występującą w zestawie danych uczących.

Właściwy dobór parametru  $k$  stanowi podstawę w uzyskaniu równowagi pomiędzy przetrenowaniem i zbyt małym dopasowaniem. Musimy się także upewnić, że wybieramy metrykę odległości dopasowaną do cech zestawu danych. Często dla próbek przyjmujących wartości liczb rzeczywistych (np. podawanych w centymetrach wymiarów kwiatów kosańca) stosowana jest prosta metryka euklidesowa. Jeśli jednak z niej korzystamy, musimy również dokonać standaryzacji danych, dzięki czemu każda cecha będzie odpowiednio wyskalowana do odległości. Wykorzystana w powyższym przykładzie odległość Minkowskiego (minkowski) stanowi uogólnienie metryki euklidesowej i miejskiej, które można zapisać następującym wzorem:

$$d(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \sqrt[p]{\sum_k |x_k^{(i)} - x_k^{(j)}|^p}$$

Jeżeli za  $p$  podstawimy 2, otrzymamy odległość euklidesową, a jeśli  $p = 1$  — to będziemy mieli do czynienia z metryką miejską. Interfejs scikit-learn zawiera mnóstwo różnych metryk odległości (parametr `metric`). Ich listę znajdziesz pod adresem <http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.DistanceMetric.html>.

### Kłatywa wymiarowości

Należy wspomnieć, że algorytm KNN jest bardzo wrażliwy na przetrenowanie z powodu **kłatywy wymiarowości** (ang. *curse of dimensionality*). Jest to zjawisko, w którym przestrzeń cech wraz ze wzrostem liczby wymiarów zestawu danych uczących o ustalonym rozmiarze staje się coraz bardziej rozległa. Możemy to sobie wyobrazić w taki sposób, że w wielowymiarowej przestrzeni nawet najbliżsi sąsiedzi znajdują się zbyt daleko, aby uzyskać za ich pomocą dobre oszacowanie.

W podrozdziale dotyczącym regresji logistycznej poruszyliśmy temat regularizacji jako jednego ze sposobów uniknięcia nadmiernego dopasowania. Jednak w modelach, wobec których nie jesteśmy w stanie wprowadzić regularizacji (np. drzewach decyzyjnych i algorytmie KNN), możemy skorzystać z technik wyboru cech i redukcji wymiarowości, pozwalających zminimalizować ryzyko wystąpienia kłatywy wymiarowości. Opisem tych rozwiązań zajmiemy się w następnym rozdziale.

## Podsumowanie

W tym rozdziale była mowa o wielu różnych algorytmach uczenia maszynowego, które są wykorzystywane do rozwiązywania liniowych oraz nieliniowych problemów. Dowiedzieliśmy się, że model drzew decyzyjnych jest wyjątkowo atrakcyjny, pod warunkiem że zadbane o właściwe interpretowanie danych. Model regresji logistycznej okazuje się przydatny nie tylko do uczenia przyrostowego za pomocą stochastycznego spadku wzduż gradientu, lecz także pozwala

przewidywać wystąpienie konkretnego zdarzenia. Maszyny wektorów nośnych to potężne modele liniowe, które za pomocą sztuczki z funkcją jądra można wykorzystać do rozwiązywania nieliniowych zagadnień, jednak w celu uzyskania dobrych przewidywań trzeba dostroić w nich dużą liczbę parametrów. Pod tym względem przeciwieństwem maszyn SVM są metody zespołowe, takie jak algorytm losowego lasu, w których nie trzeba dopasowywać wielu hiperparametrów oraz które nie są tak podatne na przetrenowanie jak pojedyncze drzewo decyzyjne, co sprawia, że staje się on atrakcyjnym modelem do rozwiązywania wielu praktycznych problemów. Alternatywnym rozwiązaniem dla klasyfikacji jest klasyfikator k-najbliższych sąsiadów, gdyż wykorzystujemy w nim mechanizm leniwego uczenia — tworzenie prognoz bez uprzedniego uczenia modelu, jednak wymagające większej mocy obliczeniowej.

Od wyboru odpowiedniego algorytmu uczenia ważniejsze są dane dostępne w zestawie uczącym. Żaden algorytm nie będzie w stanie dobrze prognozować wyników bez dostępu do informacyjnych i rozróżnialnych cech.

W kolejnym rozdziale omówimy ważne zagadnienia dotyczące wstępnego przetwarzania danych, wyboru cech oraz redukcji wymiarowości, co jest potrzebne do konstruowania potężnych modeli uczenia maszynowego. Z kolei z rozdziału 6., „Najlepsze metody oceny modelu i strojenie parametryczne”, dowiemy się, jak można oceniać i porównywać skuteczność modeli oraz zaczniemy stosować przydatne sztuczki, ułatwiające dokładne strojenie różnych algorytmów.

# Tworzenie dobrych zbiorów uczących — wstępne przetwarzanie danych

Jakość danych oraz ilość zawartych w nich użytecznych informacji stanowią główne czynniki decydujące o skuteczności trenowania algorytmu uczenia maszynowego. Jest więc nieodzowne, abyśmy sprawdzali i wstępnie przetwarzali zestaw danych przed jego wczytaniem do algorytmu uczenia. W tym rozdziale poznamy najważniejsze techniki wstępnego przetwarzania danych, dzięki którym łatwiej nam będzie tworzyć dobre modele uczenia maszynowego.

Zajmiemy się w niniejszym rozdziale następującymi zagadnieniami:

- usuwanie/wstawianie brakujących wartości w zbiorze danych,
- dopasowywanie kategoryzujących danych do algorytmów uczenia maszynowego,
- dobór odpowiednich cech podczas konstruowania modelu.

## Kwestia brakujących danych

Dość powszechnym problemem w branży informatycznej jest wynikający z różnych powodów brak co najmniej jednej wartości charakteryzującej analizowane próbki. Przyczyną może być np. jakiś błąd w procesie gromadzenia informacji, niemożność wykorzystania pewnych form

pomiarów albo podczas wypełniania formularza pewne pola mogły zostać pozostawione puste. Zazwyczaj widzimy **brakujące dane** jako puste komórki w tabeli danych lub jako określone symbole zastępcze, takie jak `NaN` (ang. *Not a Number* — w wolnym tłumaczeniu „nie jest liczbą”).

Niestety, jeżeli zignorujemy takie brakujące wartości, większość współczesnych narzędzi obliczeniowych nie będzie sobie w stanie z nimi poradzić lub zacznie generować nieprzewidywalne wyniki. Dlatego zanim przejdziemy do dalszej analizy danych, zawsze musimy w jakiś sposób zająć się tymi brakującymi danymi. Niebawem przejdziemy do omówienia kilku technik rozwiązywających kwestię brakujących danych, najpierw jednak stworzymy prostą, przykładową ramkę danych w formacie **CSV** (ang. *comma-separated values* — wartości rozdzielone przecinkiem), aby lepiej zrozumieć omawiany problem:

```
>>> import pandas as pd
>>> from io import StringIO
>>> csv_data = '''A,B,C,D
...           1.0,2.0,3.0,4.0
...           5.0,6.0,,8.0
...           10.0,11.0,12.0,''''
>>> # jeżeli korzystasz ze środowiska Python 2.7, musisz
>>> # przekonwertować ciąg znaków do standardu unicode:
>>> # csv_data = unicode(csv_data)
>>> df = pd.read_csv(StringIO(csv_data))
>>> df
   A    B    C    D
0  1    2    3    4
1  5    6  NaN  8
2  10   11   12  NaN
```

Z pomocą powyższego kodu, poprzez funkcję `read_csv` wczytujemy dane w formacie CSV do obiektu `DataFrame`; w rezultacie dwie brakujące próbki zostały zastąpione wyrażeniem `NaN`. Funkcja `StringIO` została wprowadzona jedynie w celu lepszego zilustrowania przykładu. Pozwala nam ona na wczytywanie ciągów znaków związanych z tabelą `csv_data` do obiektu `DataFrame` tak, jakby była ona zwyczajnym plikiem `.csv` zapisanym na dysku.

Przy większych tabelach `DataFrame` ręczne wyszukiwanie brakujących danych jest bardzo uciążliwe; w takim przypadku możemy wykorzystać metodę `isnull` do wygenerowania obiektu `DataFrame` zdefiniowanego przez wartości boolowskie wskazujące, czy dana komórka zawiera wartość liczbową (`False`) lub jest pusta (`True`). Następnie, używając metody `sum`, algorytm może zwrócić liczbę brakujących wartości w poszczególnych kolumnach:

```
>>> df.isnull().sum()
A    0
B    0
C    1
D    1
dtype: int64
```

W ten sposób możemy zliczać brakujące wartości w kolumnach; w kolejnych ustępach poznamy różne strategie radzenia sobie z brakującymi danymi.

Interfejs scikit-learn został stworzony w taki sposób, aby współpracował z tablicami biblioteki NumPy, jednak czasami wygodniej jest wstępnie przetwarzać dane za pomocą obiektu DataFrame biblioteki pandas. Przed wczytaniem tablicy DataFrame do estymatora scikit-learn zawsze możemy ją podejrzeć przy użyciu atrybutu values:

```
>>> df.values
array([[ 1.,  2.,  3.,  4.],
       [ 5.,  6., nan,  8.],
       [10., 11., 12., nan]])
```

## Usuwanie próbek lub cech niezawierających wartości

Jednym z najprostszych sposobów poradzenia sobie z brakującymi danymi jest całkowite usunięcie odpowiednich cech (kolumn) lub próbek (wierszy) ze zbioru danych; możemy bardzo łatwo pozbyć się takich wierszy za pomocą metody dropna:

```
>>> df.dropna()
      A   B   C   D
0   1   2   3   4
```

W podobny sposób jesteśmy w stanie pozbyć się kolumn, w których znajduje się przynajmniej jedna wartość NaN; wystarczy przydzielić argumentowi axis wartość 1:

```
>>> df.dropna(axis=1)
      A   B
0   1   2
1   5   6
2  10  11
```

Metoda dropna obsługuje również kilka innych przydatnych parametrów:

```
# usuwa jedynie wiersze, w których wszystkie kolumny mają wartość NaN
>>> df.dropna(how='all')
```

```
# usuwa wiersze, w których znajduje się mniej niż 4 próbki
>>> df.dropna(thresh=4)
```

```
# usuwa wyłącznie wiersze, w których wartość NaN pojawia się jedynie w określonych
# kolumnach (w tym przypadku w kolumnie C)
>>> df.dropna(subset=['C'])
```

Chociaż usuwanie brakujących danych wydaje się wygodnym rozwiążaniem, wiążą się z nim pewne niedogodności; np. możemy usunąć zbyt wiele próbek, przez co rzetelna analiza stanie się niemożliwa. Z kolei jeśli usuniemy zbyt dużo kolumn (cech), pojawi się niebezpieczeństwo

utraty wartościowych informacji, dzięki którym klasyfikator rozróżnia poszczególne klasy. Dlatego właśnie w następnym podrozdziale przyjrzymy się najczęściej stosowanym alternatywnym rozwiązaniom zaprojektowanym do radzenia sobie z brakującymi danymi: technikom interpolacji.

## Wstawianie brakujących danych

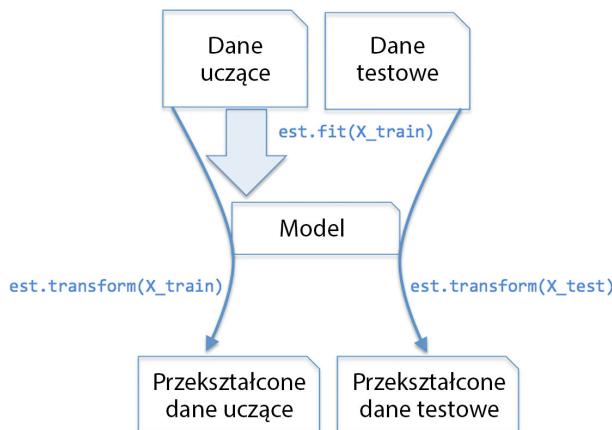
Często usuwanie próbek lub całych kolumn okazuje się po prostu niewykonalne, ponieważ grozi utratą cennych informacji. W takiej sytuacji możemy wprowadzić różnorakie techniki interpolacji, pozwalające oszacować wartości brakujących próbek na podstawie dostępnych danych uczących. Jedną z najpopularniejszych metod interpolacji jest **imputacja z użyciem średniej** (ang. *mean imputation*), polegająca na zastępowaniu brakującej wartości średnią wyliczoną na podstawie całej kolumny cech. W interfejsie scikit-learn rozwiązanie to jest bardzo wygodnie zaimplementowane, ponieważ wystarczy użyć klasy `Imputer`:

```
>>> from sklearn.preprocessing import Imputer
>>> imr = Imputer(missing_values='NaN', strategy='mean', axis=0)
>>> imr = imr.fit(df)
>>> imputed_data = imr.transform(df.values)
>>> imputed_data
array([[ 1.,  2.,  3.,  4.],
       [ 5.,  6.,  7.5,  8.],
       [ 10., 11., 12.,  6.]])
```

Zastępujemy tutaj każdą wartość `NaN` odpowiednią średnią, która jest osobno wyliczana dla poszczególnych kolumn. Gdybyśmy zmienili parametr `axis=0` na `axis=1`, wyliczana byłaby średnia z wierszy. Inne wartości, które można wstawić dla parametru `strategy`, to `median` lub `most_frequent`. W tym drugim przypadku brakujące dane są zastępowane najczęściej pojawiającymi się próbками w zestawie. Jest to bardzo przydatne podczas zastępowania wartości kategoryzujących cech.

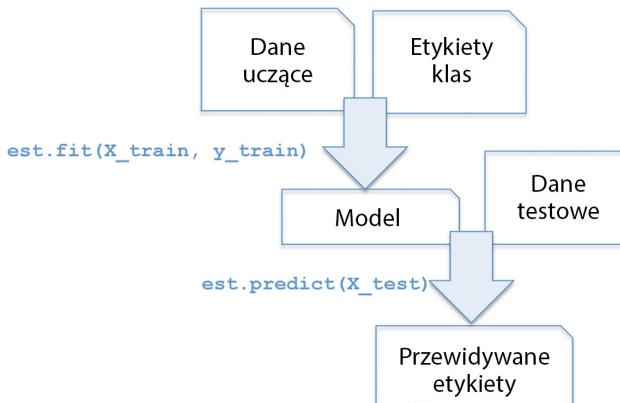
## Estymatory interfejsu scikit-learn

W poprzednim ustępie wykorzystaliśmy klasę `Imputer` (będącą częścią biblioteki scikit-learn) do zastępowania brakujących danych w zbiorze. Klasa `Imputer` należy do kategorii tzw. klas **transformujących**, które są używane do przekształcania danych. Dwiema podstawowymi metodami tego rodzaju estymatorów są `fit` i `transform`. Metoda `fit` służy do dopasowywania parametrów poprzez dane uczące, natomiast metoda `transform` wykorzystuje te parametry do modyfikowania danych. Przekształcana tabela danych musi zawierać taką samą liczbę cech jak tabela danych wykorzystana do uczenia algorytmu. Rysunek 4.1 ilustruje sposób, w jaki dopasowana do danych uczących klasa transformująca zostaje użyta do przekształcania tego zestawu uczącego, jak również nowego zbioru danych testowych.



Rysunek 4.1. Mechanizm działania klasy transformujączej

Klasyfikatory, wykorzystywane przez nas w rozdziale 3., „Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn”, przynależą do kategorii tzw. estymatorów biblioteki scikit-learn, a ich interfejs API bardzo przypomina klasę transformującą. Estymatory wykorzystują metodę predict, lecz możemy również używać metody transform, o czym przekonamy się w dalszej części rozdziału. Być może pamiętasz też, że stosowaliśmy metodę fit do trenowania parametrów modelu podczas uczenia estymatorów procesu klasyfikacji. Jednak w zadaniach uczenia nadzorowanego wprowadzamy dodatkowo etykiety klas w celu dopasowania modelu, które następnie są używane do prognozowania nowych próbek poprzez metodę predict, co zostało ukazane na rysunku 4.2.



Rysunek 4.2. Mechanizm działania estymatora

# Przetwarzanie danych kategoryzujących

Do tej pory pracowaliśmy jedynie z wartościami numerycznymi. Często jednak rzeczywiste zestawy danych zawierają przynajmniej jedną kolumnę cech kategoryzujących. Definiując dane kategoryzujące, musimy dokonać dalszego rozróżnienia na cechy **nominalne** i **porządkowe**. Cechy porządkowe to takie wartości kategoryzujące, które możemy wykorzystać do sortowania lub porządkowania informacji. Przykładowo **rozmiar koszulki** to cecha porządkowa, ponieważ możemy określić kolejność rozmiarów  $XL > L > M$ . Z drugiej strony cechy nominalne nie definiują kolejności próbek i, korzystając z poprzedniego przykładu, możemy za tego typu cechę uznać **kolor koszulki**, ponieważ zazwyczaj stwierdzenie, że *czerwony* jest większy od *niebieskiego*, nie ma sensu.

Przed przystąpieniem do omówienia różnych technik zajmujących się danymi kategoryzującymi stworzymy nową przykładową ramkę danych ukazującą istotę zagadnienia:

```
>>> import pandas as pd
>>> df = pd.DataFrame([
...     ['Zielony', 'M', 10.1, 'klasa1'],
...     ['Czerwony', 'L', 13.5, 'klasa2'],
...     ['Niebieski', 'XL', 15.3, 'klasa1']])
>>> df.columns = ['Kolor', 'Rozmiar', 'Cena', 'Etykieta klas']
>>> df
      Kolor   Rozmiar    Cena   Etykieta klas
0   Zielony        M  10.1        klasa1
1   Czerwony       L  13.5        klasa2
2   Niebieski     XL  15.3        klasa1
```

Jak widać, wynikowa tabela DataFrame składa się z cechy nominalnej (**Kolor**), porządkowej (**Rozmiar**) i numerycznej (**Cena**). W ostatniej kolumnie są przechowywane etykiety klas (założymy, że stworzyliśmy zestaw danych przeznaczonych do uczenia nadzorowanego). Omawiane w tej książce algorytmy klasyfikacji nie korzystają z informacji porządkowych zawartych w etykietach klas.

## Mapowanie cech porządkowych

Aby się upewnić, że algorytm uczenia poprawnie interpretuje cechy porządkowe, musimy przekształcić wartości kategoryzujące występujące jako ciągi znaków w postać liczb całkowitych. Niestety, nie istnieje żadna funkcja, która automatycznie przeniosłaby właściwy szynk etykiet z kolumny **Rozmiar**. Z tego powodu musimy własnoręcznie zdefiniować mapowanie cech. W poniższym przykładzie zakładamy, że znamy różnice pomiędzy cechami, np.  $XL = L+1 = M+2$ :

```
>>> size_mapping = {
...     'XL': 3,
...     'L': 2,
...     'M': 1}
```

```
>>> df['Rozmiar'] = df['Rozmiar'].map(size_mapping)
>>> df
   Kolor    Rozmiar    Cena   Etykieta klas
0  Zielony        1  10.1      klasa1
1  Czerwony       2  13.5      klasa2
2  Niebieski      3  15.3      klasa1
```

Gdybyśmy chcieli przekształcić liczby całkowite z powrotem na ich znakową reprezentację w którymś z późniejszych etapów, wystarczy zdefiniować słownik do odwrotnego mapowania `inv_size_mapping = {v: k for k, v in size_mapping.items()}`. Możemy go następnie zastosować za pomocą metody `map` wobec zmodyfikowanej kolumny cech, podobnie jak wcześniej wykorzystaliśmy słownik `size_mapping`.

## Kodowanie etykiet klas

Wiele algorytmów uczenia maszynowego wymaga, aby etykiety klas były kodowane w postaci liczb całkowitych. Większość estymatorów klasyfikacji w interfejsie scikit-learn zawiera wewnętrzne mechanizmy przekształcające etykiety klas w liczby całkowite, jednak zawsze dobrym rozwiązaniem jest własnoręczne przygotowanie tych etykiet jako tablic z liczbami całkowitymi w celu uniknięcia potencjalnych problemów technicznych. Do kodowania etykiet klas możemy wykorzystać mechanizm podobny do omówionego w poprzednim ustępie mapowania cech porządkowych. Należy pamiętać, że etykiety klas **nie są** cechami porządkowymi i nie ma znaczenia, jaką liczbę całkowitą przydzielimy do danego ciągu znaków. Możemy więc zwyczajnie ponumerować etykiety klas, począwszy od wartości 0:

```
>>> import numpy as np
>>> class_mapping = {label:idx for idx,label in
...                           enumerate(np.unique(df['Etykieta klas']))}
>>> class_mapping
{'klasa1': 0, 'klasa2': 1}
```

Teraz możemy wykorzystać słownik mapowania do przekształcenia etykiet klas w liczby całkowite:

```
>>> df['Etykieta klas'] = df['Etykieta klas'].map(class_mapping)
>>> df
   Kolor    Rozmiar    Cena   Etykieta klas
0  Zielony        1  10.1          0
1  Czerwony       2  13.5          1
2  Niebieski      3  15.3          0
```

Możemy odwrócić parę klucz – wartość w słowniku mapowania, aby przywrócić etykietom klas pierwotne wartości w postaci ciągów znaków:

```
>>> inv_class_mapping = {v: k for k, v in class_mapping.items()}
>>> df['Etykieta klas'] = df['Etykieta klas'].map(inv_class_mapping)
>>> df
```

	Kolor	Rozmiar	Cena	Etykieta klas
0	Zielony	1	10.1	klasa1
1	Czerwony	2	13.5	klasa2
2	Niebieski	3	15.3	klasa1

Uzyskamy ten sam efekt, gdy skorzystamy z wygodnej klasy LabelEncoder zaimplementowanej bezpośrednio w bibliotece scikit-learn:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> class_le = LabelEncoder()
>>> y = class_le.fit_transform(df['Etykieta klas'].values)
>>> y
array([0, 1, 0])
```

Zwróć uwagę, że metodę `fit_transform` stanowi jedynie skrót do oddzielnego wywołania metod `fit` i `transform`, a metodę `inverse_transform` możemy wykorzystać do przekształcania liczb całkowitych z powrotem do reprezentacji etykiet klas w postaci ciągów znaków:

```
>>> class_le.inverse_transform(y)
array(['klasa1', 'klasa2', 'klasa1'], dtype=object)
```

## Kodowanie „gorącojedynkowe” cech nominalnych (z użyciem wektorów własnych)

W poprzednim ustępie zastosowaliśmy proste rozwiązywanie odwzorowywania przy użyciu słownika w celu przekształcenia wartości cechy porządkowej w liczbę całkowitą. Estymatory biblioteki scikit-learn nie przetwarzają etykiet klas w żadnym jasno określonym porządku, dlatego wprowadziliśmy klasę `LabelEncoder` do przekształcania ciągów znaków etykiet w liczby całkowite. Mogłoby się wydawać, że w podobny sposób jesteśmy w stanie przekształcić próbki znajdujące się w nominalnej kolumnie `Kolor`:

```
>>> X = df[['Kolor', 'Rozmiar', 'Cena']].values
>>> color_le = LabelEncoder()
>>> X[:, 0] = color_le.fit_transform(X[:, 0])
>>> X
array([[1, 1, 10.1],
       [2, 2, 13.5],
       [0, 3, 15.3]], dtype=object)
```

Po uruchomieniu powyższego kodu pierwsza kolumna w tabeli NumPy przechowuje teraz nowe wartości cechy `Kolor`, które zostają zakodowane w następujący sposób:

- Niebieski → 0
- Zielony → 1
- Czerwony → 2

Gdybyśmy w tym momencie się zatrzymali i przesłali tabelę do klasyfikatora, popełnilibyśmy jeden z najczęstszych błędów spotykanych podczas przetwarzania danych kategoryzujących. Czy widzisz, w czym tkwi problem? Mimo że wartości kolorów nie pojawiają się w określonej kolejności, algorytm uczący będzie teraz zakładał, że *zielony* jest większy od *niebieskiego*, a *czerwony* od *zielonego*. To założenie jest niewłaściwe, a jednak algorytm może generować poprawne wyniki, chociaż będą dalekie od optymalnych.

Popularnym rozwiązaniem tego problemu jest zastosowanie techniki zwanej **kodowaniem „gorącojedynkowym”** (ang. *one-hot encoding*). Koncepcją kryjącą się za tą metodą jest wprowadzenie **sztucznej cechy** (ang. *dummy feature*) dla każdej unikatowej wartości w kolumnie cechy nominalnej. W naszym przypadku możemy przekonwertować cechę *Kolor* w trzy nowe cechy: *Niebieski*, *Zielony* i *Czerwony*. Wykorzystamy wartości binarne do wskazywania danego koloru próbki; np. niebieska próbka może zostać zakodowana jako *Niebieski*=1, *Zielony*=0, *Czerwony*=0. Aby dokonać takiej transformacji, możemy wykorzystać klasę *OneHotEncoder* zaimplementowaną w module *scikit-learn.preprocessing*:

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> ohe = OneHotEncoder(categorical_features=[0])
>>> ohe.fit_transform(X).toarray()
array([[ 0. ,  1. ,  0. ,  1. , 10.1],
       [ 0. ,  0. ,  1. ,  2. , 13.5],
       [ 1. ,  0. ,  0. ,  3. , 15.3]])
```

Po zainicjowaniu klasy *OneHotEncoder* zdefiniowaliśmy pozycję kolumny, w której znajduje się zmienna przekształcana za pomocą parametru *categorical\_features* (zauważmy, że *Kolor* jest pierwszą kolumną w macierzy cech *X*). Domyślnie po użyciu metody *transform* klasa *OneHotEncoder* zwraca macierz rzadką, ale w celu lepszej wizualizacji przykładowu przekształciliśmy tę tablicę w macierz regularną (**gęstą**) poprzez metodę *toarray*. Macierz rzadka stanowi wydajniejszy sposób na przechowywanie dużych ilości danych, a także jest obsługiwana przez wiele funkcji biblioteki *scikit-learn*, co okazuje się szczególnie przydatne, gdy zawiera ona dużo zer. Gdybyśmy nie chcieli korzystać z metody *toarray*, moglibyśmy zainicjować kodowanie jako *OneHotEncoder(..., sparse=False)*, dzięki czemu generowana byłaby gęsta macierz.

Jeszcze wygodniejszym sposobem tworzenia takich sztucznych cech za pomocą kodowania „gorącojedynkowego” jest wykorzystanie metody *get\_dummies* zaimplementowanej w bibliotece *pandas*. Po jej zastosowaniu wobec tabeli *DataFrame* zostaną przekształcone jedynie kolumny zawierające ciągi znaków, natomiast wszystkie pozostałe cechy nie ulegną zmianie:

```
>>> pd.get_dummies(df[['Cena', 'Kolor', 'Rozmiar']])
   Cena  Rozmiar  Kolor_Niebieski  Kolor_Zielony  Kolor_Czerwony
0  10.1        1            0            1            0
1  13.5        2            0            0            1
2  15.3        3            1            0            0
```

# Rozdzielanie zestawu danych na podzbiory uczące i testowe

W rozdziałach 1., „Umożliwianie komputerom uczenia się z danych”, i 3., „Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn”, побieżnie wyjaśniłem koncepcję rozdzielania zestawu danych na osobne podzbiory próbek uczących i testowych. Przypominam, że zestaw testowy rozumiemy jako **ostateczny sprawdzian** naszego modelu przed zastosowaniem go w prawdziwym świecie. W tym podrozdziale przygotujemy kolejny zestaw danych — **Wine**. Po wstępny przetworzeniu próbek poznamy różne techniki służące do wyboru cech, a zatem redukowania wymiarowości.

Zestaw danych Wine stanowi kolejny otwarty zbiór informacji dostępny w repozytorium uczenia maszynowego UCI<sup>1</sup> (<https://archive.ics.uci.edu/ml/datasets/Wine>); składa się ze 178 próbek win opisywanych 13 cechami (własnościemi chemicznymi).

Za pomocą biblioteki pandas bezpośrednio wczytamy zestaw danych Wine:

```
>>> df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/machinelearning-databases/wine/wine.data', header=None)
>>> df_wine.columns = ['Etykieta klas', 'Alkohol',
...                   'Kwas jabłkowy', 'Popiół',
...                   'Zasadowość popiołu', 'Magnez',
...                   'Całk. zaw. fenoli', 'Flawonoidy',
...                   'Fenole nieflawonoidowe',
...                   'Proantocyjaniny',
...                   'Intensywność koloru', 'Odcień',
...                   'Transmitancja 280/315 nm',
...                   'Prolina']
>>> print('Etykiety klas', np.unique(df_wine['Etykieta klas']))
Etykiety klas [ 1  2  3]
>>> df_wine.head()
```

Umieszczonej w zbiorze danych **Wine** 13 różnych cech opisujących właściwości chemiczne 178 próbek win zostało ukazanych na rysunku 4.3.

	Etykieta klas	Alkohol	Kwas jabłkowy	Popiół	Zasadowość popiołu	Magnez	Całk. zaw. fenoli	Flawonoidy	Fenole nieflawonoidowe	Proantocyjaniny	Intensywność koloru	Odcień	Transmitancja 280/315 nm	Prolina
0	1	14,23	1,71	2,43	15,6	127	2,80	3,06	0,28	2,29	5,64	1,04	3,92	1065
1	1	13,20	1,78	2,14	11,2	100	2,65	2,76	0,26	1,28	4,38	1,05	3,40	1050
2	1	13,16	2,36	2,67	18,6	101	2,80	3,24	0,30	2,81	5,68	1,03	3,17	1185
3	1	14,37	1,95	2,50	16,8	113	3,85	3,49	0,24	2,18	7,80	0,86	3,45	1480
4	1	13,24	2,59	2,87	21,0	118	2,80	2,69	0,39	1,82	4,32	1,04	2,93	735

Rysunek 4.3. Opis pięciu pierwszych próbek zawartych w zbiorze danych Wine

<sup>1</sup> University of California, Irvine, czyli Uniwersytet Kalifornijski w Irvine — przyp. tłum.

Próbki przynależą do jednej z trzech klas: 1, 2 i 3, co stanowi odniesienie do odmian winorośli rosnących w różnych regionach Włoch.

Wygodnym sposobem losowego rozdzielenia tego zestawu danych na podzbiory danych testowych i uczących jest wykorzystanie funkcji `train_test_split` umieszczonej w podmoduле `model_selection`<sup>2</sup> interfejsu scikit-learn:

```
>>> from distutils.version import LooseVersion as Version
>>> from sklearn import __version__ as sklearn_version

>>> if Version(sklearn_version) < '0.18':
>>>     from sklearn.cross_validation import train_test_split
>>> else:
>>>     from sklearn.model_selection import train_test_split

>>> X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values
>>> X_train, X_test, y_train, y_test = \
...         train_test_split(X, y, test_size=0.3, random_state=0)
```

Najpierw przydzieliliśmy reprezentację 13 kolumn cech do zmiennej `X`, a etykiety klas z pierwszej kolumny do zmiennej `y`. Następnie użyliśmy funkcji `train_test_split` do losowego rozdzielenia wartości `X` i `y` pomiędzy osobne podzbiory uczące i testowe. Ustanowiony parametr `test_size=0.3`, przydzieliliśmy 30% próbek wina do zmiennych `X_test` i `y_test`, a pozostałe 70% próbek zostało przeniesionych do `X_train` i `y_train`.

Jeżeli rozdzielamy zestaw testowy na podzbiory uczące i testowe, musimy brać pod uwagę fakt, że rezygnujemy z części cennych informacji, z których algorytm uczenia mógłby skorzystać. Z tego powodu nie chcemy umieszczać zbyt wielu informacji w zbiorze testowym. Z drugiej strony: im jest on mniejszy, tym mniej redukujemy dokładność oszacowania błędu uogólniania. Dzielenie zbioru danych na zestawy uczące i testowe polega na znalezieniu złotego środka. W praktyce najczęściej wykorzystywanymi proporcjami pomiędzy danymi uczącymi a testowymi są 60:40, 70:30 lub 80:20, w zależności od początkowego rozmiaru zestawu. Jednak w przypadku dużych zbiorów danych często spotykane i odpowiednie są takie proporcje jak 90:10 czy 99:1. Zamiast odrzucania wykorzystanych danych testowych po wyuczeniu i ocenie modelu dobrym pomysłem jest ponowne wytrenowanie klasyfikatora w celu uzyskania optymalnej skuteczności.

## Skalowanie cech

Skalowanie cech jest o tyle ważnym etapem **wstępnego przetwarzania danych**, co równie łatwo zapominanym. Drzewa decyzyjne i lasy losowe to jedne z niewielu algorytmów uczenia maszynowego, w których nie musimy martwić się skalowaniem danych. Jednakże zdecydowana

---

<sup>2</sup> W starszych wersjach biblioteki scikit-learn funkcja ta występowała w podmodule `cross_validation` — przyp. tłum.

większość algorytmów uczenia maszynowego i optymalizacyjnych działa znacznie skuteczniej, jeżeli cechy są dopasowane do jednakowej skali, o czym mieliśmy okazję się przekonać w rozdziale 2., „Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji”, przy okazji implementacji algorytmu **gradientu prostego**.

Znaczenie skalowania cech możemy zrozumieć poprzez bardzo prosty przykład. Założymy, że mamy do czynienia z dwiema cechami, z których jedna jest mierzona w skali 1:10, a druga — 1:100 000. Jeśli przypomnimy sobie funkcję sumy kwadratów błędów stosowaną w modelu **Adaline** (rozdział 2., „Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji”), łatwo sobie uświadomimy, że algorytmowi temu najwięcej czasu zajmowałoby optymalizowanie wag z powodu większych błędów wynikających z drugiej cechy. Innym przykładem może być algorytm **k-najbliższych sąsiadów** (KNN) wykorzystujący miarę odległości euklidesowej; obliczane dystanse pomiędzy próbками zostaną zdominowane przez wartości drugiej cechy.

Istnieją dwie popularne metody sprowadzania różnych cech do wspólnej skali: **normalizacja** i **standaryzacja**. Pojęcia te są dość luźno stosowane w różnych dziedzinach nauki, a ich znaczenie zależy od kontekstu. Gdy mówimy o normalizacji, najczęściej mamy na myśli przeskłowanie cech do zakresu [0, 1], co stanowi specjalny przypadek skalowania min. – max. Aby przeprowadzić normalizację danych, wystarczy przeskalać każdą kolumnę cech wobec wartości krańcowych — tu nową wartość  $x_{\text{norm}}^{(i)}$  próbki  $x^{(i)}$  możemy obliczyć następująco:

$$x_{\text{norm}}^{(i)} = \frac{x^{(i)} - x_{\min}}{x_{\max} - x_{\min}}$$

Tutaj  $x^{(i)}$  oznacza daną próbke,  $x_{\min}$  — najmniejszą wartość próbki w kolumnie cech, a  $x_{\max}$  — największą wartość.

Tego typu procedura skalowania jest zaimplementowana w bibliotece scikit-learn i możemy ją zobaczyć w poniższym fragmencie kodu:

```
>>> from sklearn.preprocessing import MinMaxScaler
>>> mms = MinMaxScaler()
>>> X_train_norm = mms.fit_transform(X_train)
>>> X_test_norm = mms.transform(X_test)
```

Chociaż normalizacja wykorzystująca skalowanie min. – max. jest powszechnie używaną techniką, przydatną w sytuacji, gdy potrzebujemy wartości w ograniczonych przedziałach, standaryzacja okazuje się znacznie praktyczniejszym rozwiązaniem w wielu algorytmach uczenia maszynowego. Wynika to z faktu, że mnóstwo modeli liniowych, takich jak omówione w rozdziale 3., „Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn”, algorytmy regresji logistycznej czy maszyn wektorów nośnych inicjują wagi o zerowej lub losowej, bardzo zbliżonej do zera wartości. Poprzez standaryzację wprowadzamy do kolumn cech średnią 0 przy odchyleniu standardowym 1, dzięki czemu otrzymujemy rozkład normalny, ułatwiający uczenie wag. Do tego w procesie standaryzacji zachowujemy przydatne informacje

na temat odstających próbek, a on sam zmniejsza wrażliwość algorytmu na te krańcowe przypadki, w przeciwieństwie do skalowania min. – max., gdzie dane zostają przeskalowane do ograniczonego zakresu wartości.

Procedurę standaryzacji możemy zapisać w postaci wzoru:

$$x_{std}^{(i)} = \frac{x^{(i)} - \mu_x}{\sigma_x}$$

Zmienna  $\mu_x$  oznacza tutaj średnią próbek z danej kolumny cech, a  $\sigma_x$  jest związanym z nią odchyleniem standardowym.

W tabeli 4.1 ukazalem różnicę pomiędzy dwiema powszechnie stosowanymi technikami skalowania funkcji — standaryzacją i normalizacją — przy użyciu prostego zestawu danych składających się z liczb 0 – 5.

**Tabela 4.1.** Różnica pomiędzy normalizacją i standaryzacją dla przykładowego zbioru danych

Dane wejściowe	Standaryzowane	Normalizowane
0,0	-1,336306	0,0
1,0	-0,801784	0,2
2,0	-0,267261	0,4
3,0	0,267261	0,6
4,0	0,801784	0,8
5,0	1,336306	1,0

Interfejs scikit-learn zawiera także klasę umożliwiającą standaryzowanie danych:

```
>>> from sklearn.preprocessing import StandardScaler
>>> stdsc = StandardScaler()
>>> X_train_std = stdsc.fit_transform(X_train)
>>> X_test_std = stdsc.transform(X_test)
```

Również w tym przypadku muszę zaznaczyć, że wprowadzamy klasę StandardScaler tylko raz wobec danych uczących, a następnie wykorzystujemy te wyniki do przekształcania danych testowych lub każdej nowej próbki.

## Dobór odpowiednich cech

Jeżeli zauważymy, że model sprawuje się znacznie lepiej wobec danych uczących niż testowych, może to oznaczać, że algorytm uległ nadmiernemu dopasowaniu; algorytm dopasował parametry zbyt blisko wyników z danych uczących, żeby móc je uogólnić do rzeczywistych

próbek — w takim przypadku mówimy, że model cechuje **duża wariancja**. Przyczyną przetrenowania jest zbyt duża złożoność modelu wobec wykorzystywanych danych uczących. Problem ten możemy rozwiązać poprzez:

- nagromadzenie większej ilości danych uczących,
- wprowadzenie kary za złożoność poprzez regularyzację,
- dobór prostszego modelu zawierającego mniej parametrów,
- zmniejszenie wymiarowości danych.

Dodateknie większej liczby danych uczących nie zawsze jest możliwe. W następnym rozdziale poznamy przydatną technikę pozwalającą sprawdzić, czy dodatkowy zapas danych uczących jest w stanie w czymkolwiek pomóc. W kolejnych ustępach przyjrzymy się popularnym sposobom redukowania przetrenowania poprzez regularyzację oraz zmniejszenie wymiarowości za pomocą wyboru cech.

## Regularizacja L1

Pamiętamy z rozdziału 3., „Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn”, że **regularizacja L2** stanowi jeden ze sposobów zmniejszania kompleksowości modelu poprzez karzenie dużych, pojedynczych wag; normę wektora wagi  $w$  zdefiniowaliśmy następująco:

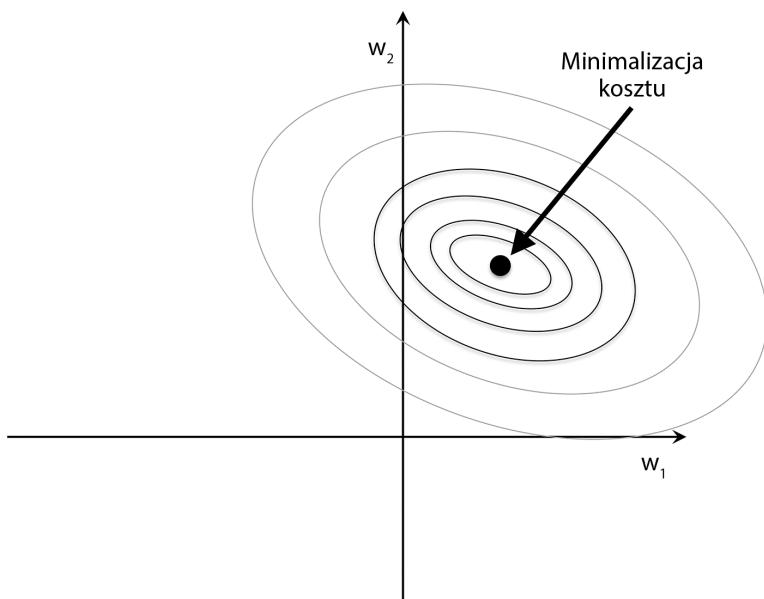
$$\text{L2} : \|\boldsymbol{w}\|_2^2 = \sum_{j=1}^m w_j^2$$

Inna metoda redukcji złożoności modelu jest powiązana z **regularizacją L1**:

$$\text{L1} : \|\boldsymbol{w}\|_1 = \sum_{j=1}^m |w_j|$$

Zastąpiliśmy tu po prostu sumy kwadratów wag sumą ich wartości bezwzględnych. W przeciwieństwie do regularizacji L2 regularizacja L1 generuje rzadkie wektory cech; większość wag będzie równa 0. Rzadkość bywa przydatna w przypadku wielowymiarowych zestawów danych o dużej liczbie nieważnych cech, zwłaszcza gdy istnieje więcej nieistotnych wymiarów niż próbki. W takim znaczeniu regularizację L1 możemy uznać za techniki wyboru cech.

Aby lepiej zrozumieć zaletę rzadkości podczas korzystania z regularizacji L1, cofniemy się nieco i przyjrzyjmy geometrycznej interpretacji regularizacji. Narysujmy wykres konturowy wypukłej funkcji kosztu dla dwóch współczynników wag:  $w_1$  i  $w_2$ . Będziemy rozpatrywać funkcję kosztu **sumy kwadratów błędów**, wykorzystywaną w modelu Adaline opisany w rozdziale 2., „Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji”, gdyż jest to wykres prostszy do wygenerowania i bardziej symetryczny od wykresu funkcji kosztu regresji logistycznej; jednak omawiane koncepcje pasują do obydwu typów funkcji. Pamiętajmy, że naszym celem jest znalezienie takiej kombinacji współczynników wag, która zminimalizuje funkcję kosztu dla danych uczących, co zostało przedstawione na rysunku 4.4 (punkt wyznaczający środek elips).



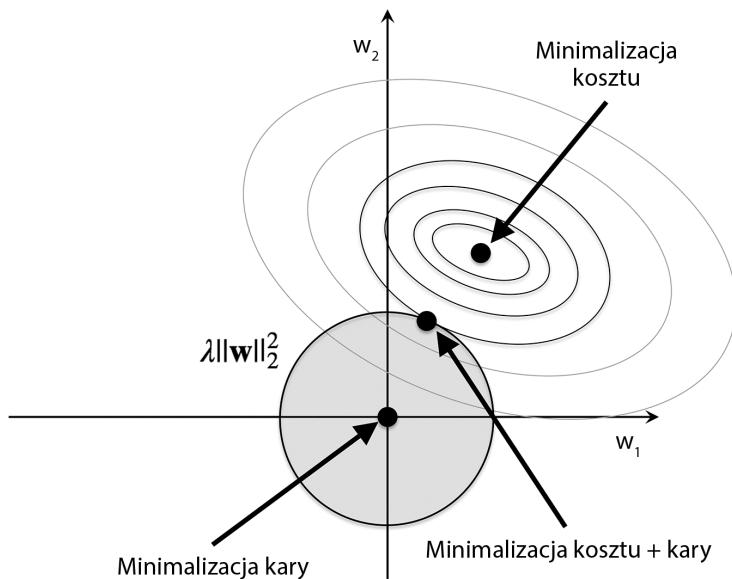
Rysunek 4.4. Wykres wypukłej funkcji kosztu dla współczynników wag  $w_1$  i  $w_2$

W takim kontekście regularyzację możemy uznać za dodatkową karę nakładaną na funkcję kosztu w celu faworyzowania mniejszych wag; innymi słowy, karcimy duże wagi.

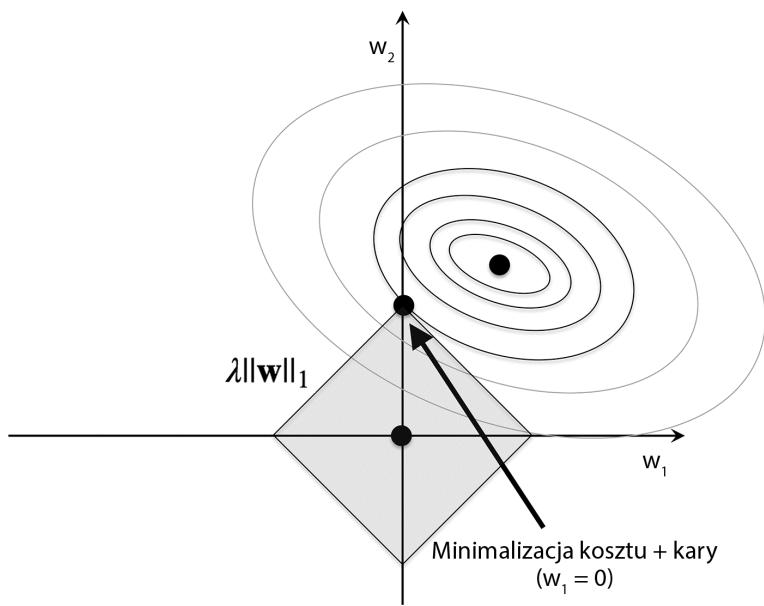
Zatem zwiększając siłę regularyzacji za pomocą parametru  $\lambda$ , zmniejszamy wartości wag w kierunku zera i ograniczamy zależność modelu od danych uczących. Koncepcja ta została ukazana na rysunku 4.5.

Warunek kwadratowej regularyzacji L2 jest widoczny na rysunku 4.5 jako wypełniony okrąg. Współczynniki wag nie mogą przekroczyć budżetu regularyzacji — suma współczynników wag nie może wykroczyć poza zacieniowany obszar. Z drugiej strony ciągle chcemy zminimalizować funkcję kosztu. Pamiętając o ograniczeniu, staramy się wybrać punkt, w którym kraniec okręgu styka się z zarysem funkcji kosztu. Im większa wartość parametru regularyzacji  $\lambda$ , tym szybciej wzrasta obszar funkcji kosztu objęty karą, co prowadzi do zmniejszania okręgu L2. Jeżeli np. parametr  $\lambda$  będzie dążył do nieskończoności, to współczynniki wag będą maleć do zera, gdyż będą ograniczone do samego środka okręgu. Podsumujmy moral pływający z tego przykładu: naszym celem jest minimalizacja sumy nieobarzczonej karą funkcji kosztu i warunku kary, co można interpretować jako dodawanie obciążenia oraz dobór prostszego modelu w celu zmniejszenia wariancji z powodu niedostatku danych uczących.

Przyjrzyjmy się teraz regularyzacji L1 i zjawisku rzadkości. Ogólna koncepcja tej metody jest podobna do regularyzacji L2. W tym jednak przypadku mamy do czynienia z sumą wartości bezwzględnych wag (przypominam, że warunek L2 jest funkcją kwadratową), co możemy wyrysować na wykresie jako **budżet** romboidalny, zaprezentowany na rysunku 4.6.



Rysunek 4.5. Funkcja regularizacji L2 nałożona na funkcję kosztu



Rysunek 4.6. Wykres regularyzacji L1

Widzimy na rysunku 4.6, że zarys funkcji kosztu styka się z wierzchołkiem rombu L1, gdy  $w_1 = 0$ . Krawędzie budżetu L1 są wyraźnie zaznaczone, dlatego jest bardzo prawdopodobne, że optimum — tj. punkt zetknięcia się elipsy funkcji kosztu z granicą rombu regularizacji — znajduje

się w punkcie przecięcia z którąś osią, co sprzyja rozrzedzeniu. Opis matematyczny przyczyń takiego sprzyjania rozrzedzeniu przez regularyzację L1 wykracza poza ramy niniejszej książki. Jeżeli interesuje Cię porównanie obydwu typów regularizacji, znakomity opis znajdziesz w podrozdziale 3.4 książki *The Elements of Statistical Learning*, której autorami są Trevor Hastie, Robert Tibshirani i Jerome Friedman (wydawnictwo Springer).

W interfejsie scikit-learn dla modeli wykorzystujących regularyzację L1 wystarczy wprowadzić wartość 11 parametru `penalty`, aby uzyskać rozrzedzenie:

```
>>> from sklearn.linear_model import LogisticRegression
>>> LogisticRegression(penalty='l1')
```

Po zastosowaniu standaryzowanego (L1) algorytmu regresji logistycznej wobec zestawu danych Wine otrzymujemy następujące wyniki:

```
>>> lr = LogisticRegression(penalty='l1', C=0.1)
>>> lr.fit(X_train_std, y_train)
>>> print('Dokładność dla danych uczących:', lr.score(X_train_std, y_train))
Dokładność dla danych uczących: 0.983870967742
>>> print('Dokładność dla danych testowych:', lr.score(X_test_std, y_test))
Dokładność dla danych testowych: 0.981481481481
```

Obydwa pomiary (każdy z nich przekracza wartość 98%) nie wskazują na przetrenowanie modelu. Sprawdzenie warunków przecięcia za pomocą atrybutu `lr.intercept_` powoduje zwrócenie trójelementowej tablicy:

```
>>> lr.intercept_
array([-0.38379237, -0.1580855 , -0.70047966])
```

Dopasowujemy obiekt `LogisticRegression` do wieloklasowego zestawu danych, dlatego domyślnie zostaje wykorzystana technika **OvR (jeden przeciw reszcie)**, w której pierwszy punkt przecięcia należy do modelu dopasowującego klasę 1 przeciw klasom 2 i 3; drugi punkt przecięcia należy do modelu dopasowującego klasę 2 przeciw klasom 1 i 3; a trzeci punkt przecięcia należy do modelu dopasowującego klasę 3 do klas 1 i 2:

```
>>> lr.coef_
array([[ 0.280,  0.000,  0.000, -0.0282,  0.000,
         0.000,  0.710,  0.000,  0.000,  0.000,
         0.000,  0.000,  1.236],
       [-0.644, -0.0688 , -0.0572,  0.000,  0.000,
        0.000,  0.000,  0.000,  0.000, -0.927,
        0.060,  0.000, -0.371],
       [ 0.000,  0.061,  0.000,  0.000,  0.000,
        0.000, -0.637,  0.000,  0.000,  0.499,
        -0.358, -0.570,  0.000
      ]])
```

Wygenerowana za pomocą atrybutu `lr.coef_` tablica zawiera trzy wiersze współczynników wag — po jednym wektorze wag na każdą klasę. Każdy wiersz składa się z 13 wag, z których każda została przemnożona przez odpowiednią cechę 13-wymiarowego zestawu danych Wine, co pozwoliło wyliczyć całkowite pobudzenie układu:

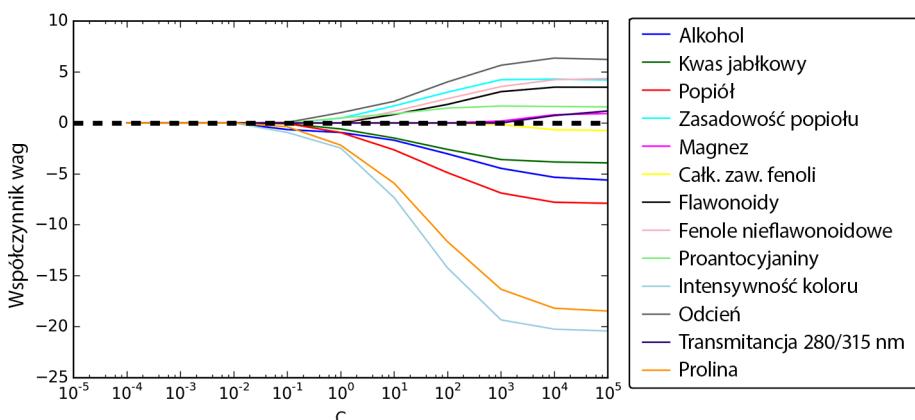
$$z = w_1 x_1 + \cdots + w_m x_m + b = \sum_{j=0}^m x_j w_j + b = \mathbf{w}^T \mathbf{x} + b$$

Widzimy, że wektory wag są rozrzedzone, co oznacza, że zawierają tylko kilka niezerowych wartości. W wyniku regularizacji L1 (stanowiącej jedną z metod wyboru cech) wytrenowaliśmy model w dużej mierze odporny na wpływ potencjalnie nieistotnych cech analizowanego zestawu danych. Współczynnik  $b$  oznacza składową obciążenia (która biblioteka scikit-learn przechowuje osobno od współczynników wag).

Wygenerujmy na koniec wykres ścieżki regularizacji, czyli współczynników wag różnych cech dla poszczególnych wartości sily regularyzacji:

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> ax = plt.subplot(111)
>>> colors = ['blue', 'green', 'red', 'cyan',
...             'magenta', 'yellow', 'black',
...             'pink', 'lightgreen', 'lightblue',
...             'gray', 'indigo', 'orange']
>>> weights, params = [], []
>>> for c in np.arange(-4, 6):
...     lr = LogisticRegression(penalty='l1',
...                             C=10**c,
...                             random_state=0)
...     lr.fit(X_train_std, y_train)
...     weights.append(lr.coef_[1])
...     params.append(10**c)
>>> weights = np.array(weights)
>>> for column, color in zip(range(weights.shape[1]), colors):
...     plt.plot(params, weights[:, column],
...               label=df_wine.columns[column+1],
...               color=color)
>>> plt.axhline(0, color='black', linestyle='--', linewidth=3)
>>> plt.xlim([10**(-5), 10**5])
>>> plt.ylabel('Współczynnik wag')
>>> plt.xlabel('C')
>>> plt.xscale('log')
>>> plt.legend(loc='upper left')
>>> ax.legend(loc='upper center',
...            bbox_to_anchor=(1.38, 1.03),
...            ncol=1, fancybox=True)
>>> plt.show()
```

Wykres pokazany na rysunku 4.7 daje nam więcej informacji na temat działania regularizacji L1. Jak widać, wszystkie wagi cech będą równe 0, jeżeli będziemy karać model dużą wartością parametru regularizacji ( $C < 0,1$ );  $C$  jest odwrotnością parametru regularizacji .



Rysunek 4.7. Wykres ścieżki regularizacji

## Algorytmy sekwencyjnego wyboru cech

Alternatywnym sposobem zmniejszenia złożoności modelu i uniknięcia przetrenowania jest **redukcja wymiarowości** poprzez wybór cech, co jest skuteczne zwłaszcza w przypadku nieregularyzowanych modeli. Istnieją dwie podstawowe metody redukowania wymiarowości: **wybór cech** i **odkrywanie cech**. Za pomocą wyboru cech możemy wyznaczać podzbiory pierwotnych cech. W przypadku odkrywania cech pozyskujemy ze zbioru cech informacje pozwalające stworzyć nową podprzestrzeń cech. W tym ustępie zapoznamy się z rodziną klasycznych algorytmów doboru cech. Z kolei w rozdziale 5., „Kompresja danych poprzez redukcję wymiarowości”, omówimy różnorodne techniki odkrywania cech pozwalające na przeniesienie zbioru danych do mniejszej podprzestrzeni cech.

Algorytmy sekwencyjnego wyboru cech należą do rodziny zachłannych algorytmów przeszukiwania wykorzystywanych do zmniejszania początkowej **d-wymiarowej** przestrzeni cech do podprzestrzeni **k-wymiarowej**, gdzie  $k < d$ . Celem algorytmów wyboru cech jest automatyczne wyznaczanie podzbioru cech, które są najściślej powiązane z analizowanym problemem, co pozwala na poprawę skuteczności obliczeniowej lub zmniejszenie błędu uogólniania w modelu poprzez usuwanie nieistotnych cech/szumu — przydaje się to w algorytmach niewykorzystujących regularizacji. Sztandarowym przykładem algorytmu sekwencyjnego wyboru cech jest algorytm **sekwencyjnej selekcji wstecznnej** (ang. *Sequential Backward Selection* — **SBS**), w którym staramy się zredukować wymiarowość początkowego podzbioru cech przy minimalnym rozpadzie skuteczności klasyfikatora, a zatem poprawie skuteczności obliczeniowej. W pewnych sytuacjach algorytm SBS może nawet pozytywnie wpływać na siłę predykcyjną przetrenowanego modelu.

Algorytmy zachłanne dokonują lokalnie optymalnych wyborów na każdym etapie przeszukiwania kombinatorycznego i zazwyczaj uzyskują niższe od optymalnych rozwiązania problemu, w przeciwieństwie do algorytmów wyczerpującego przeszukiwania, które oszacowują wszystkie możliwe kombinacje i zawsze znajdują optymalne wyniki. W praktyce jednak przeszukiwanie wyczerpujące pochłania mnóstwo zasobów obliczeniowych, natomiast algorytmy zachłanego wyszukiwania stanowią prostsze, mniej wymagające wydajnościowo rozwiązanie.

Mechanizm działania algorytmu SBS jest zasadniczo całkiem prosty: algorytm sekwencyjnie usuwa cechy z zapełnionego podzbioru cech aż do wprowadzenia odpowiedniej ich liczby do nowego podzbioru. Aby określić, które cechy mają zostać usunięte na każdym etapie, musimy zdefiniować minimalizowaną funkcję kryterialną  $J$ . Wartością kryterialną wyliczaną przez tę funkcję może być np. różnica w skuteczności klasyfikatora przed pozbyciem się danej cechy i po jej usunięciu. Następnie możemy tak zdefiniować usuwaną funkcję, że będzie maksymalizowała tę wartość kryterialną; albo, mówiąc bardziej intuicyjnie, na końcu każdego etapu eliminujemy cechę, której usunięcie w najmniejszym stopniu obniża skuteczność modelu. Na podstawie takiej definicji algorytmu SBS możemy jego działanie rozpisać na cztery proste etapy:

1. Inicjacja algorytmu przy zadanym parametrze  $k = d$ , gdzie  $d$  oznacza wymiarowość pełnej przestrzeni cech  $\mathbf{x}_d$ .
2. Określenie cechy  $x^-$  maksymalizującej funkcję kryterialną  $x^- = \arg \max_{\mathbf{x} \in \mathbf{X}_k} J(\mathbf{X}_k - \mathbf{x})$ ,
3. Usunięcie cechy  $x^-$  ze zbioru cech:  $\mathbf{x}_{k-1} := \mathbf{X}_k - \mathbf{x}^- ; k := k-1$ .
4. Zakończenie działania, jeżeli  $k$  jest równe liczbie wymaganych cech, w przeciwnym wypadku powrót do etapu 2.

Dokładny opis kilku różnych algorytmów sekwencyjnego wyszukiwania cech znajdziesz w artykule *Comparative Study of Techniques for Large Scale Feature Selection*, którego autorami są Francesc Ferri, Pavel Pudil, Mohamad Hatef i Josef Kittler („Pattern Recognition in Practice IV” 1994, s. 403 – 413).

Niestety, algorytm SBS nie został jeszcze zaimplementowany w bibliotece scikit-learn. Nie jest jednak skomplikowany, dlatego możemy go samodzielnie stworzyć w Pythonie:

```
from sklearn.base import clone
from itertools import combinations
import numpy as np
if Version(sklearn_version) < '0.18':
    from sklearn.cross_validation import train_test_split
else:
    from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

class SBS():
    def __init__(self, estimator, k_features,
                 scoring=accuracy_score,
```

```

test_size=0.25, random_state=1):
    self.scoring = scoring
    self.estimator = clone(estimator)
    self.k_features = k_features
    self.test_size = test_size
    self.random_state = random_state

def fit(self, X, y):
    X_train, X_test, y_train, y_test = \
        train_test_split(X, y, test_size=self.test_size,
                         random_state=self.random_state)

    dim = X_train.shape[1]
    self.indices_ = tuple(range(dim))
    self.subsets_ = [self.indices_]
    score = self._calc_score(X_train, y_train,
                             X_test, y_test, self.indices_)
    self.scores_ = [score]

    while dim > self.k_features:
        scores = []
        subsets = []

        for p in combinations(self.indices_, r=dim-1):
            score = self._calc_score(X_train, y_train,
                                     X_test, y_test, p)
            scores.append(score)
            subsets.append(p)

        best = np.argmax(scores)
        self.indices_ = subsets[best]
        self.subsets_.append(self.indices_)
        dim -= 1

        self.scores_.append(scores[best])
    self.k_score_ = self.scores_-[-1]

    return self

def transform(self, X):
    return X[:, self.indices_]

def _calc_score(self, X_train, y_train,
               X_test, y_test, indices):
    self.estimator.fit(X_train[:, indices], y_train)
    y_pred = self.estimator.predict(X_test[:, indices])
    score = self.scoring(y_test, y_pred)
    return score

```

W powyższej implementacji zdefiniowaliśmy parametr `k_features` określający liczbę pożądanego cech, które mają zostać zwrócone. Domyślnie wykorzystujemy funkcję biblioteki scikit-learn `accuracy_score` do oceny skuteczności modelu i estymatora klasyfikacji wobec podzbiorów cech. Wewnątrz pętli `while` (w metodzie `fit`) oceniane są podzbiory cech tworzone przez funkcję `itertools.combinations`, a następnie redukowane do momentu osiągnięcia pożądanej wymiarowości. W każdej iteracji punktacja dokładności najlepszego podzbioru jest zapisywana na liście `self.scores_` na podstawie wewnętrznie utworzonego zestawu testowego `X_test`. Będziemy później korzystać z tej punktacji do oceniania wyników. Indeksy kolumn ostatecznego podzbioru cech zostają przydzielone do obiektu `self.indices_`, który po potraktowaniu metodą `transform` zwraca nową tabelę danych z wybranymi cechami. Zwróci uwagę, że zamiast jawnego wyliczenia funkcji kryterialnej w metodzie `fit` po prostu usuwaliśmy cechy niewystępujące w optymalnym podzbiorze cech.

Sprawdźmy teraz działanie implementacji algorytmu SBS wykorzystującej klasyfikator KNN:

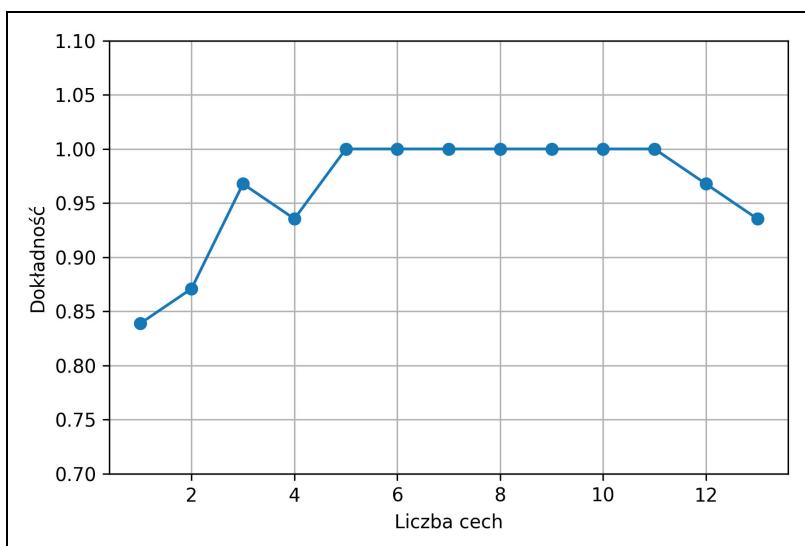
```
>>> from sklearn.neighbors import KNeighborsClassifier
>>> import matplotlib.pyplot as plt
>>> knn = KNeighborsClassifier(n_neighbors=2)
>>> sbs = SBS(knn, k_features=1)
>>> sbs.fit(X_train_std, y_train)
```

Chociaż nasza implementacja już rozdziela zestaw danych na podzbiory danych uczących i testowych wewnętrz metodę `fit`, musimy dostarczyć algorytmowi próbki potrzebne do utworzenia zbioru `X_train`. Metoda `fit` wygeneruje wtedy nowe podzbiory uczące służące do uczenia i testowania (walidacji), dlatego jest on często nazywany **walidacyjnym zbiorem danych**. *Jest to niezbędne rozwiązanie: w ten sposób sprawiamy, że pierwotny zestaw testowy nie zostaje dołączony do danych uczących.*

Pamiętaj, że nasz algorytm SBS na każdym etapie zapisuje punktację najlepszego podzbioru cech, dlatego przejdźmy do ciekawszej części implementacji i wygenerujmy wykres klasyfikatora KNN obliczanego wobec zestawu walidacyjnego. Kod wygląda następująco:

```
>>> k_feat = [len(k) for k in sbs.subsets_]
>>> plt.plot(k_feat, sbs.scores_, marker='o')
>>> plt.ylim([0.7, 1.1])
>>> plt.ylabel('Dokładność')
>>> plt.xlabel('Liczba cech')
>>> plt.grid()
>>> plt.show()
```

Jak widać na rysunku 4.8, dokładność klasyfikatora KNN analizującego próbki walidacyjne poprawiała się wraz ze zmniejszaniem liczby cech, co prawdopodobnie ma związek z redukowaniem **klątwy wymiarowej** omówionej w rozdziale 3., „*Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn*”, przy okazji opisu algorytmu k-najbliższych sąsiadów. Ponadto widzimy, że klasyfikator osiągnął 100-procentową dokładność dla wartości  $k = [5, 6, 7, 8, 9, 10, 11]$ .



Rysunek 4.8. Wpływ liczby cech na dokładność klasyfikatora

Dla zaspokojenia ciekawości sprawdźmy, które pięć cech gwarantuje taką dobrą skuteczność podczas analizowania danych walidacyjnych:

```
>>> k5 = list(sbs.subsets_[8])
>>> print(df_wine.columns[1:][k5])
Index(['Alkohol', 'Kwas jabłkowy', 'Zasadowość popiołu', 'Odcień', 'Prolina'],
      dtype='object')
```

Z pomocą powyższego kodu odczytaliśmy indeksy kolumn pięcioelementowego podzbioru cech z dziewiątej pozycji atrybutu `sbs.subsets_`, po czym wyświetliśmy odpowiadające tym indeksom nazwy cech umieszczone w obiekcie DataFrame zawierającym dane tabeli Wine.

Sprawdźmy teraz skuteczność klasyfikatora KNN wobec pierwotnego zestawu testowego:

```
>>> knn.fit(X_train_std, y_train)
>>> print('Dokładność dla danych uczących:', knn.score(X_train_std, y_train))
Dokładność dla danych uczących: 0.983870967742
>>> print('Dokładność dla danych testowych:', knn.score(X_test_std, y_test))
Dokładność dla danych testowych: 0.944444444444
```

W powyższym kodzie wykorzystaliśmy pełny zestaw cech i otrzymaliśmy  $\sim 98,4\%$  dokładności dla danych uczących. Jednak dokładność dla danych testowych okazała się nieco niższa ( $\sim 94,4\%$ ), co stanowi objaw niewielkiego przetrenowania. Wprowadźmy teraz wspomniany wcześniej pięcioelementowy podzióbior cech i sprawdźmy skuteczność algorytmu KNN:

```
>>> knn.fit(X_train_std[:, k5], y_train)
>>> print('Dokładność dla danych uczących:',
...       knn.score(X_train_std[:, k5], y_train))
```

```
Dokładność dla danych uczących: 0.959677419355
>>> print('Dokładność dla danych testowych:',
...         knn.score(X_test_std[:, k5], y_test))
Dokładność dla danych testowych: 0.962962962963
```

Przy użyciu mniej niż połowy dostępnych cech w zestawie danych Wine dokładność przewidawań dla zbioru testowego wzrosła niemal o 2%. Zmniejszyliśmy również nadmierne dopasowanie, gdyż dokładność dla danych uczących (~96%) znacznie zbliżała się do dokładności dla danych testowych (~96,3%).

#### Algorytmy wyboru cech w bibliotece scikit-learn

Interfejs scikit-learn zawiera znacznie więcej algorytmów wyboru cech. Wśród nich możemy wymienić rekurencyjną eliminację wsteczną bazującą na wagach cech, metody drzew polegające na doborze cech pod kątem ich istotności, a także jednoczynnikowe testy statystyczne. Głębsza analiza tych metod wykracza poza zakres tej książki, ale ich dobre podsumowanie wraz z graficznymi przykładami można znaleźć pod adresem [http://scikit-learn.org/stable/modules/feature\\_selection.html](http://scikit-learn.org/stable/modules/feature_selection.html).

## Ocenianie istotności cech za pomocą algorytmu losowego lasu

W poprzednich podrozdziałach nauczyliśmy się wykorzystywać regularyzację L1 do ignorowania nieistotnych cech w modelu regresji logistycznej, a także używać algorytmu SBS w doborze cech. Innym przydatnym sposobem wybierania korzystnych cech z zestawu danych jest utworzenie losowego lasu — technika ta została omówiona w rozdziale 3., „Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn”. Za pomocą losowego lasu jesteśmy w stanie mierzyć ważność cechy jako uśredniony spadek zanieczyszczeń obliczony dla wszystkich drzew decyzyjnych w lesie bez konieczności definiowania założeń, czy dane są liniowo rozdzielne lub nie. Implementacja algorytmu losowego lasu dostępna w bibliotece scikit-learn gromadzi informacje o istotności cech za nas, dzięki czemu możemy uzyskać do nich dostęp za pomocą atrybutu `feature_importances` już po wyuczeniu obiektu `RandomForestClassifier`. Dzięki poniższemu kodowi stworzymy las składający się z 10 000 drzew na podstawie zestawu danych Wine, a następnie ocenimy wszystkie 13 cech pod względem ich metryki ważności. Pamiętasz zapewne z rozdziału 3., „Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn”, że w modelach wykorzystujących algorytmy drzew nie musimy nic standaryzować ani normalizować. Wspomniany fragment kodu został zaprezentowany poniżej:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> feat_labels = df_wine.columns[1:]
>>> forest = RandomForestClassifier(n_estimators=10000,
...                                   random_state=0,
```

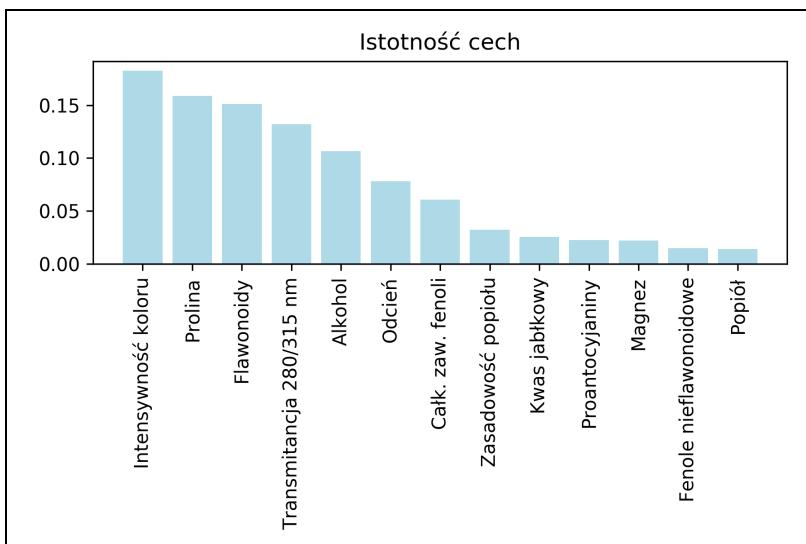
```

...
n_jobs=-1)
>>> forest.fit(X_train, y_train)
>>> importances = forest.feature_importances_
>>> indices = np.argsort(importances)[::-1]
>>> for f in range(X_train.shape[1]):
...     print("%2d) %-*s %f" % (f + 1, 30,
...                           feat_labels[indices[f]],
...                           importances[indices[f]]))
1) Intensywność koloru          0.182483
2) Prolina                      0.158610
3) Flawonoidy                   0.150948
4) Transmitancja 280/315 nm    0.131987
5) Alkohol                      0.106589
6) Odcień                       0.078243
7) Całk. zaw. fenoli           0.060718
8) Zasadowość popiołu          0.032033
9) Kwas jabłkowy                0.025400
10) Proantocyjaniny            0.022351
11) Magnez                      0.022078
12) Fenole nieflawonoidalowe   0.014645
13) Popiół                       0.013916
>>> plt.title('Istotność cech')
>>> plt.bar(range(X_train.shape[1]),
...           importances[indices],
...           color='lightblue',
...           align='center')
>>> plt.xticks(range(X_train.shape[1]),
...             feat_labels[indices], rotation=90)
>>> plt.xlim([-1, X_train.shape[1]])
>>> plt.tight_layout()
>>> plt.show()

```

Wskutek uruchomienia powyższego kodu stworzyliśmy wykres, na którym zostały poszerowane cechy zawarte w zestawie danych Wine pod kątem względnej istotności; zwrócić uwagę, że ukazane na rysunku 4.9 wartości zostały znormalizowane i po zsumowaniu są równe 1.

Podsumowując, na podstawie średniej redukcji zanieczyszczeń w 10 000 drzewach okazuje się, że najbardziej rozróżniającą cechą w zestawie danych Wine jest intensywność koloru wina. Co ciekawe, trzy najlepiej oceniane cechy z rysunku 4.9 znalazły się również wśród podzbioru pięciu najwydajniejszych cech dobranych przez algorytm SBS, z którego skorzystaliśmy w poprzednim ustępie. Jednak pod względem interpretowania danych algorytmy losowego lasu cechują się pewnym **potencjalnym problemem**, o którym należy wiedzieć. Jeśli co najmniej dwie cechy są ze sobą ściśle powiązane, jedna z nich może zostać wysoko oceniona, podczas gdy informacje zawarte w drugiej (lub pozostałych) mogą zostać nie do końca uchwycone przez algorytm. Z drugiej strony nie mamy powodu do zmartwień, jeżeli jesteśmy zainteresowani wyłącznie skutecznością predykcyjną modelu, a nie interpretowaniem istotności cech. Warto



Rysunek 4.9. Wykres istotności poszczególnych cech zestawu danych Wine

dodać, że interfejs scikit-learn zawiera również zaimplementowaną metodę `SelectFromModel`<sup>3</sup>, która po zakończeniu uczenia dobiera cechy na podstawie wartości progowej wyznaczonej przez użytkownika, co okazuje się przydatne w sytuacji, gdy chcemy skorzystać z obiektu `RandomForestClassifier` jako z selekcjonera cech oraz etapu pośredniego pozwalającego łączyć różne techniki wstępnego przetwarzania danych z estymatorem, co zostanie zaprezentowane w rozdziale 6., „Najlepsze metody oceny modelu i strojenie parametryczne”. Możemy np. ustawić wartość progową wynoszącą 0,15, aby zmniejszyć zestaw cech do trzech najważniejszych: *Intensywność koloru*, *Prolina* i *Flawonoidy*:

```
>>> if Version(sklearn_version) < '0.18':
>>>     X_selected = forest.transform(X_train, threshold=0.15)
>>> else:
>>>     from sklearn.feature_selection import SelectFromModel
>>>     sfm = SelectFromModel(forest, threshold=0.15, prefit=True)
>>>     X_selected = sfm.transform(X_train)

>>> X_selected.shape
(124, 3)
```

<sup>3</sup> Zastępuje ona starszą metodę `Transform` — przyp. tłum.

## Podsumowanie

Rozpoczęliśmy ten rozdział od przyjrzenia się przydatnym technikom umożliwiającym radzenie sobie z brakującymi danymi. Zanim prześlemy dane do algorytmu uczenia maszynowego, musimy również we właściwy sposób zakodować zmienne kategoryzujące. Nauczyliśmy się także mapować wartości cech nominalnych i porządkowych na postać liczb całkowitych.

Ponadto ogólnie omówiliśmy zagadnienie regularyzacji L1, która służy do zapobiegania przetrenowaniu modelu poprzez zmniejszenie jego złożoności. Poznaliśmy też alternatywny sposób usuwania nieistotnych cech — algorytm sekwencyjnego wyboru cech — wybierający najistotniejsze cechy z zestawu danych.

Z następnego rozdziału dowiesz się więcej na temat kolejnego sposobu redukowania wymiarowości: odkrywania cech. Za jego pomocą będziemy w stanie kompresować cechy do podprzestrzeni o mniejszej liczbie wymiarów bez potrzeby ich usuwania, jak to ma miejsce w algorytmie wyboru cech.



# Kompresja danych poprzez redukcję wymiarowości

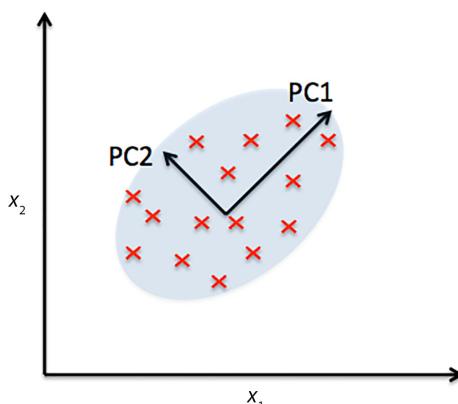
W rozdziale 4., „Tworzenie dobrych zbiorów uczących — wstępne przetwarzanie danych”, poznaliśmy różne sposoby redukowania wymiarowości zestawu danych za pomocą odmiennych technik wyboru cech. Alternatywnym rozwiązaniem dla wyboru cech w celu zmniejszenia wymiarowości jest **odkrywanie cech** (ang. *feature extraction*). W tym rozdziale przyjrzymy się trzem głównym mechanizmom umożliwiającym podsumowywanie informacji zawartych w zbiorze danych poprzez ich przeniesienie do podprzestrzeni o mniejszej liczbie wymiarów niż pierwotna przestrzeń. Kompresja danych stanowi istotne zagadnienie w dziedzinie uczenia maszynowego, gdyż pozwala nam na przechowywanie i analizowanie coraz większej ilości danych generowanych każdego dnia. Zajmiemy się teraz omówieniem następujących tematów:

- analiza głównych składowych w celu kompresowania nienadzorowanych danych,
- liniowa analiza dyskryminacyjna jako technika nadzorowanej redukcji wymiarowości w celu zmaksymalizowania rozdzielności klas,
- nieliniowa redukcja wymiarowości za pomocą jądrowej analizy głównych składowych.

# Nienadzorowana redukcja wymiarowości za pomocą analizy głównych składowych

Możemy wykorzystać odkrywanie cech, podobnie jak ich wybór, do zmniejszania liczby cech w zbiorze danych. Jednak stosując algorytmy wyboru cech (takie jak **sekwencyjna selekcja wsteczna**), przechowywaliśmy ciągle wszystkie cechy, natomiast dzięki odkrywaniu cech przekształcamy/rzutujemy dane na nową przestrzeń cech. W kontekście redukcji wymiarowości odkrywanie cech możemy interpretować jako metodę kompresji danych przeprowadzaną pod kątem zachowania jak największej ilości użytecznych informacji. Odkrywanie cech najczęściej służy do zwiększenia skuteczności obliczeniowej, pomaga jednak również ograniczać **kłatwę wymiarowości** — zwłaszcza jeżeli działały na niregularyzowanych danych.

**Analiza głównych składowych** (ang. *principal component analysis* — PCA) jest techniką nienadzorowanej, liniowej transformacji, powszechnie wykorzystywanej w różnych dziedzinach, najczęściej w celu redukowania wymiarowości. Inne popularne zastosowania tej metody to m.in. badawcza analiza danych, usuwanie szumu w transakcjach giełdowych, a także bioinformatyczna analiza genomów oraz ekspresji genów. Analiza PCA umożliwia identyfikowanie wzorców danych na podstawie korelacji pomiędzy cechami. Krótko mówiąc, zadaniem analizy głównych składowych jest wyszukiwanie kierunków maksymalnej wariancji w wielowymiarowej przestrzeni i rzutowanie ich na nową podprzestrzeń zawierającą tyle samo (lub mniej) wymiarów, co pierwotna przestrzeń cech. Osie ortogonalne (główne składowe) rzutowane w nowej podprzestrzeni symbolizują kierunki maksymalnej wariancji przy wprowadzonym ograniczeniu mówiącym, że nowe osie cech są wobec siebie ułożone prostopadle, co zostało zaprezentowane na rysunku 5.1. Na widocznym wykresie  $x_1$  i  $x_2$  są pierwotnymi osiami cech, a  $PC1$  i  $PC2$  to główne składowe.



Rysunek 5.1. Graficzna ilustracja głównych składowych wyznaczających maksimum wariancji

W przypadku stosowania analizy PCA do redukowania wymiarowości tworzymy  $d \times k$ -wymiarową macierz transformacji  $\mathbf{W}$ , która umożliwi nam rzutowanie wektora próbek  $\mathbf{x}$  na nową,  $k$ -wymiarową podprzestrzeń cech, mającą mniej wymiarów niż pierwotna,  $d$ -wymiarowa przestrzeń cech:

$$\mathbf{x} = [x_1, x_2, \dots, x_d], \quad \mathbf{x} \in \mathbb{R}^d$$

$$\downarrow \mathbf{x}\mathbf{W}, \quad \mathbf{W} \in \mathbb{R}^{d \times k}$$

$$\mathbf{z} = [z_1, z_2, \dots, z_k], \quad \mathbf{z} \in \mathbb{R}^k$$

W wyniku rzutowania pierwotnych  $d$ -wymiarowych danych na nową  $k$ -wymiarową podprzestrzeń (zazwyczaj  $k < d$ ) pierwsza główna składowa wyznaczy największą możliwą wariancję, a wszystkie następne składowe będą również cechować się największą możliwą wariancją, pod warunkiem że będą nieskorelowane (ortogonalne) z pozostałymi głównymi składowymi. Zwróci uwagę, że kierunki wyznaczane metodą PCA są bardzo czułe na skalowanie danych i, jeśli cechy były mierzone w różnych skalach (a nam zależy na ujednoliceniu ich ważności), musimy przed uruchomieniem algorytmu przeprowadzić standaryzację cech.

Zanim przejdziemy do dokładniejszego opisu algorytmu PCA, podsumujmy jego podstawowe etapy:

1. Standaryzowanie  $d$ -wymiarowego zestawu danych.
2. Stworzenie macierzy kowariancji.
3. Rozkład macierzy kowariancji na jej wektory własne i wartości własne.
4. Wybór  $k$  wektorów własnych odpowiadających  $k$  największym wartościom własnym, gdzie  $k$  oznacza wymiarowość nowej podprzestrzeni cech ( $k \leq d$ ).
5. Utworzenie macierzy rzutowania  $\mathbf{W}$  z  $k$  „najlepszych” wektorów własnych.
6. Przekształcenie  $d$ -wymiarowego początkowego zbioru danych  $X$  za pomocą macierzy rzutowania  $\mathbf{W}$  w celu uzyskania nowej,  $k$ -wymiarowej podprzestrzeni cech.

## Wyjaśniona wariancja całkowita

W tym ustępie zajmiemy się pierwszymi czterema etapami analizy głównych składowych: standaryzowaniem danych, utworzeniem macierzy kowariancji, rozkładem tej macierzy na jej wektory własne i wartości własne, a także szeregowaniem wartości własnych tak, aby pasowały do wektorów własnych.

Wczytamy najpierw zestaw danych **Wine**, przy którym już pracowaliśmy w rozdziale 4., „Tworzenie dobrych zbiorów uczących — wstępne przetwarzanie danych”:

```
>>> import pandas as pd
>>> df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/machinelearning-
databases/wine/wine.data', header=None)
```

Teraz rozdzielimy wczytane dane na oddzielny podzbior danych uczących (70% próbek) oraz testowych (30% próbek) i dokonamy standaryzacji przy użyciu wariancji jednostkowej w celu wyeliminowania rozbieżności:

```
>>> from distutils.version import LooseVersion as Version
>>> from sklearn import __version__ as sklearn_version
>>> if Version(sklearn_version) < '0.18':
>>>     from sklearn.cross_validation import train_test_split
>>> else:
>>>     from sklearn.model_selection import train_test_split
>>> from sklearn.preprocessing import StandardScaler
>>> X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values
>>> X_train, X_test, y_train, y_test = \
...         train_test_split(X, y,
...                         test_size=0.3, random_state=0)
>>> sc = StandardScaler()
>>> X_train_std = sc.fit_transform(X_train)
>>> X_test_std = sc.transform(X_test)
```

Po przeprowadzeniu niezbędnych etapów wstępnego przetwarzania danych możemy przejść do kolejnej czynności: stworzenia macierzy kowariancji. Symetryczna macierz kowariancji o wymiarach  $d \times d$ , gdzie  $d$  oznacza liczbę wymiarów w zestawie danych, przechowuje pary kowariancji łączących poszczególne cechy; np. kowariancję pomiędzy dwiema cechami  $x_j$  i  $x_k$  na poziomie populacji możemy wyliczyć za pomocą poniższego wzoru:

$$\sigma_{jk} = \frac{1}{n} \sum_{i=1}^n (x_j^{(i)} - \mu_j)(x_k^{(i)} - \mu_k)$$

Parametry  $\mu_j$  i  $\mu_k$  stanowią wartości średnie dla cech, odpowiednio,  $j$  i  $k$ . Zauważ, że te wartości średnie po przeprowadzeniu standaryzacji zestawu danych wynoszą 0. Dodatnia kowariancja pomiędzy dwiema cechami wskazuje na to, że wartości sparowanych cech wspólnie rosną lub maleją, natomiast wartość ujemna kowariancji mówi nam, że wartości cech zmieniają się w przeciwnych kierunkach. Możemy np. zapisać macierz kowariancji trzech cech w następujący sposób (znak  $\Sigma$  oznacza tu greczką literę sigma, a nie symbol sumowania):

$$\Sigma = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_2^2 & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_3^2 \end{bmatrix}$$

Wektory własne macierzy kowariancji symbolizują główne składowe (kierunki maksymalnej wariancji), natomiast związane z nimi wartości własne definiują rozmiar tych składowych. W przypadku zestawu danych Wine możemy otrzymać 13 wektorów własnych i wartości własnych przechowywanych w wektorze kowariancji o wymiarze  $13 \times 13$ .

Wczytajmy teraz wartości własne z macierzy kowariancji. Na pewno pamiętasz z lekcji matematyki, że wektor własny  $v$  spełnia następujący warunek:

$$\Sigma v = \lambda v$$

Parametr  $\lambda$  jest tu wartością skalarną; tzw. wartością własną. Samodzielne obliczanie wektorów własnych i wartości własnych jest poniekąd żmudnym i pracochłonnym zadaniem, dlatego skorzystamy z funkcji `linalg.eig` należącej do biblioteki NumPy do wydobycia par własnych macierzy kowariancji wyliczonej dla zestawu danych Wine:

```
>>> import numpy as np
>>> cov_mat = np.cov(X_train_std.T)
>>> eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)
>>> print('\nWartości własne\n' % eigen_vals)
Wartości własne
[ 4.8923083 2.46635032 1.42809973 1.01233462 0.84906459
 0.60181514
 0.52251546 0.08414846 0.33051429 0.29595018 0.16831254 0.21432212
 0.2399553 ]
```

Z pomocą funkcji `numpy.cov` obliczyliśmy macierz kowariancji ustandaryzowanego zestawu danych uczących. Z kolei funkcja `linalg.eig` wykonała proces rozkładu własnego tej macierzy, w wyniku czego otrzymaliśmy wektor (`eigen_vals`) składający się z 13 wartości własnych, a także macierz  $13 \times 13$  przechowującą wektory własne w postaci kolumn (`eigen_vecs`).

Chociaż funkcja `numpy.linalg.eig` została zaprojektowana tak, aby rozkładać niesymetryczne macierze kwadratowe, w pewnych szczególnych przypadkach może zwracać bardzo złożone wartości własne.

Do rozkładu macierzy hermitowskich służy podobna funkcja, `numpy.linalg.eigh`, która cechuje się większą stabilnością numeryczną podczas przetwarzania macierzy symetrycznych, takich jak macierz kowariancji; funkcja ta zawsze zwraca wartości własne w formie liczb rzeczywistych.

Chcemy zmniejszyć wymiarowość zestawu danych, kompresując go do nowej podprzestrzeni cech, dlatego interesuje nas jedynie podzbiór wektorów własnych (głównych składowych) zawierających najwięcej informacji (wariancję). Wartości własne definiują rozmiar wektorów własnych, dlatego musimy uszeregować je w malejącej kolejności; szukamy  $k$  największych wektorów na podstawie sparowanych z nimi wartości własnych. Zanim jednak uzyskamy te  $k$  wektorów własnych zawierających najwięcej informacji, narysujmy wykres **współczynników wariancji wyjaśnionej** dla wartości własnych.

Współczynnik wariancji wyjaśnionej dla wartościowej  $\lambda_j$  to nic innego, jak iloraz tej wartościowej i całkowitej sumy wszystkich wartości własnych:

$$\frac{\lambda_j}{\sum_{j=1}^d \lambda_j}$$

Z pomocą funkcji `cumsum` (stanowiącej część biblioteki NumPy) możemy obliczyć łączną sumę wyjaśnionych wariancji, po czym przy użyciu funkcji `step` (przynależnej do biblioteki `matplotlib`) wygenerujemy:

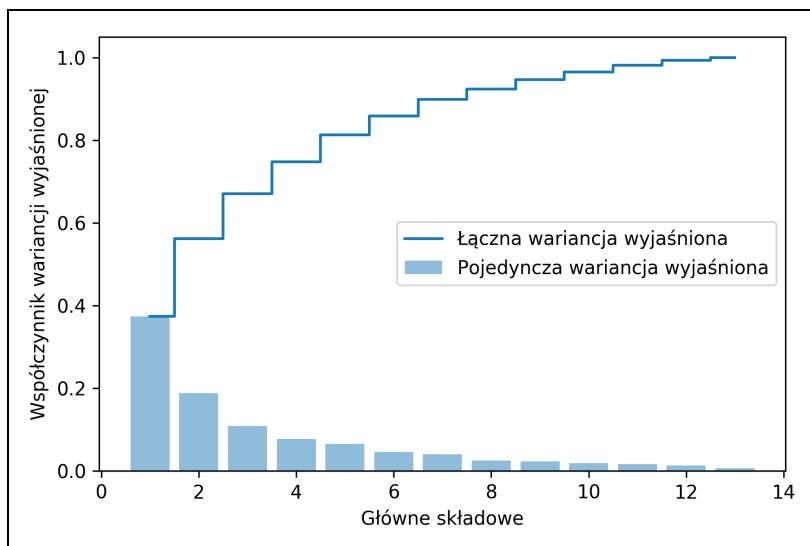
```

>>> tot = sum(eigen_vals)
>>> var_exp = [(i / tot) for i in
...             sorted(eigen_vals, reverse=True)]
>>> cum_var_exp = np.cumsum(var_exp)

>>> import matplotlib.pyplot as plt
>>> plt.bar(range(1,14), var_exp, alpha=0.5, align='center',
...         label='Pojedyncza wariancja wyjaśniona')
>>> plt.step(range(1,14), cum_var_exp, where='mid',
...           label='Łączna wariancja wyjaśniona')
>>> plt.ylabel('Współczynnik wariancji wyjaśnionej')
>>> plt.xlabel('Główne składowe')
>>> plt.legend(loc='best')
>>> plt.show()

```

Jak widać na rysunku 5.2, sama pierwsza główna składowa stanowi 40% wariancji. Ponadto suma dwóch pierwszych głównych składowych daje niemal 60% wariancji analizowanego zestawu danych.



Rysunek 5.2. Wykres współczynników wariancji wyjaśnionej

Chociaż wykres wariancji wyjaśnionej przypomina nam wykres istotności cech wygenerowany za pomocą algorytmu losowego lasu w rozdziale 4., „Tworzenie dobrych zbiorów uczących — wstępne przetwarzanie danych”, musimy pamiętać, że metoda PSA jest nienadzorowana, co oznacza, że informacje na temat etykiet klas są ignorowane. Algorytm losowego lasu wykorzystuje informacje o przynależności określonych instancji do wyznaczonych grup w celu obliczania zanieczyszczenia węzłów, natomiast wariancja mierzy rozkład wartości wzduż osi cech.

## Transformacja cech

Po skutecznym rozłożeniu macierzy kowariancji na pary własne możemy zająć się trzema pozostałymi etapami rzutowania zestawu danych **Wine** na nową podprzestrzeń głównych składowych. W niniejszym ustępie uszeregujemy pary własne w kierunku malejącym wartości własnych, z wyselekcjonowanych wektorów własnych utworzymy macierz rzutowania i wykorzystamy ją do przeniesienia do podprzestrzeni składającej się z mniejszej liczby wymiarów.

Zaczniemy od uszeregowania par własnych zgodnie z malejącymi wartościami własnymi:

```
>>> eigen_pairs =[(np.abs(eigen_vals[i]),eigen_vecs[:,i])
...                 for i in range(len(eigen_vals))]
>>> eigen_pairs.sort(key=lambda k: k[0], reverse=True)
```

Teraz wybieramy dwa wektory własne, które łącznie dają największą wartość wariancji (ok. 60%). Zwróć uwagę, że dobieram jedynie dwa wektory własne w celu lepszego zilustrowania przykładu, gdyż w dalszej części rozdziału wygenerujemy dwuwymiarowy wykres punktowy dla tych danych. W praktyce liczbę głównych składowych wyznacza się na podstawie kompromisu pomiędzy mocą obliczeniową a skutecznością klasyfikatora:

```
>>> w= np.hstack((eigen_pairs[0][1][:, np.newaxis],
...                  eigen_pairs[1][1][:, np.newaxis]))
>>> print('Macierz W:\n',w)
Macierz W:
[[ 0.14669811  0.50417079]
 [-0.24224554  0.24216889]
 [-0.02993442  0.28698484]
 [-0.25519002 -0.06468718]
 [ 0.12079772  0.22995385]
 [ 0.38934455  0.09363991]
 [ 0.42326486  0.01088622]
 [-0.30634956  0.01870216]
 [ 0.30572219  0.03040352]
 [-0.09869191  0.54527081]
 [ 0.30032535 -0.27924322]
 [ 0.36821154 -0.174365 ]
 [ 0.29259713  0.36315461]]
```

Powyższy kod tworzy macierz rzutowania **W** o wymiarze  $13 \times 2$  z dwóch największych wektorów własnych. Za pomocą tej macierzy możemy teraz przenieść próbkę **x** (mającą postać wektora  $1 \times 13$ ) w podprzestrzeń PCA i uzyskać dwuwymiarowy wektor próbki **x'** składający się z dwóch nowych cech:

$$\mathbf{x}' = \mathbf{x} \mathbf{W}$$

```
>>> X_train_std[0].dot(w)
array([ 2.59891628,  0.00484089])
```

W podobny sposób przekształcamy cały zestaw danych uczących o rozmiarze  $124 \times 13$  na dwie główne składowe — obliczając iloczyn skalarny macierzy:

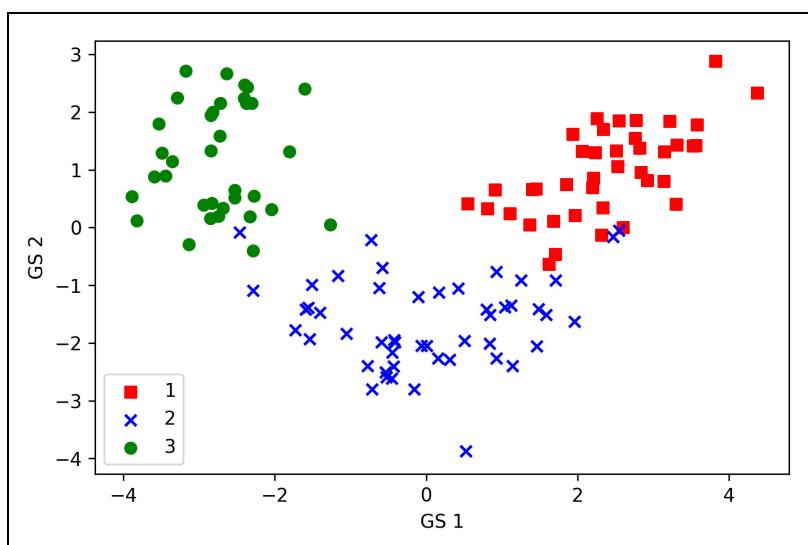
$$\mathbf{X}' = \mathbf{X}\mathbf{W}$$

```
>>> X_train_pca = X_train_std.dot(w)
```

Teraz możemy zwizualizować nasz przekształcony zestaw danych Wine, przechowywany w postaci macierzy  $124 \times 2$ , na dwuwymiarowym wykresie punktowym:

```
>>> colors = ['r', 'b', 'g']
>>> markers = ['s', 'x', 'o']
>>> for l, c, m in zip(np.unique(y_train), colors, markers):
...     plt.scatter(X_train_pca[y_train==l, 0],
...                 X_train_pca[y_train==l, 1],
...                 c=c, label=l, marker=m)
>>> plt.xlabel('GS 1')
>>> plt.ylabel('GS 2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

Jak widać na wykresie (rysunek 5.3), dane zostały bardziej rozproszone wzduż osi  $x$  — pierwszej głównej składowej — niż wzduż drugiej głównej składowej (osi  $y$ ), co jest zgodne z wykresem współczynnika wyjaśnionej wariancji zaprezentowanym na rysunku 5.2. Patrząc na rysunek 5.3, możemy z łatwością stwierdzić, że klasyfikator powinien bez większego trudu rozdzielać analizowane klasy.



Rysunek 5.3. Wykres danych rzutowanych w podprzestrzeni dwóch głównych składowych o największej wariancji

Chociaż w celach lepszego zilustrowania przykładu zakodowaliśmy informację o etykietach klas, nie możemy zapominać, że analiza PCA jest techniką nienadzorowaną, niewykorzystującą informacji zawartych w etykietach klas.

## Analiza głównych składowych w interfejsie scikit-learn

Dosyć rozwlekłe omówienie analizy PCA z poprzedniego ustępu pozwoliło nam zrozumieć jej wewnętrzne mechanizmy działania, teraz zaś zajmiemy się kwestią wykorzystania klasy PCA zaimplementowanej w bibliotece scikit-learn. Jest to kolejna z dziedzin klas transformujących, w których najpierw dopasowujemy model za pomocą danych uczących, a następnie przekształcamy zarówno dane uczące, jak i testowe za pomocą tych samych parametrów modelu. Sprawdźmy teraz działanie tej klasy PCA na naszym zestawie danych Wine — sklasyfikujmy przekształcone próbki za pomocą algorytmu regresji logistycznej i zwizualizujmy regiony decyzyjne dzięki zdefiniowanej w rozdziale 2., „Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji”, funkcji `plot_decision_regions`:

```
from matplotlib.colors import ListedColormap

def plot_decision_regions(X, y, classifier, resolution=0.02):

    # konfiguruje generator znaczników i mapę kolorów
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

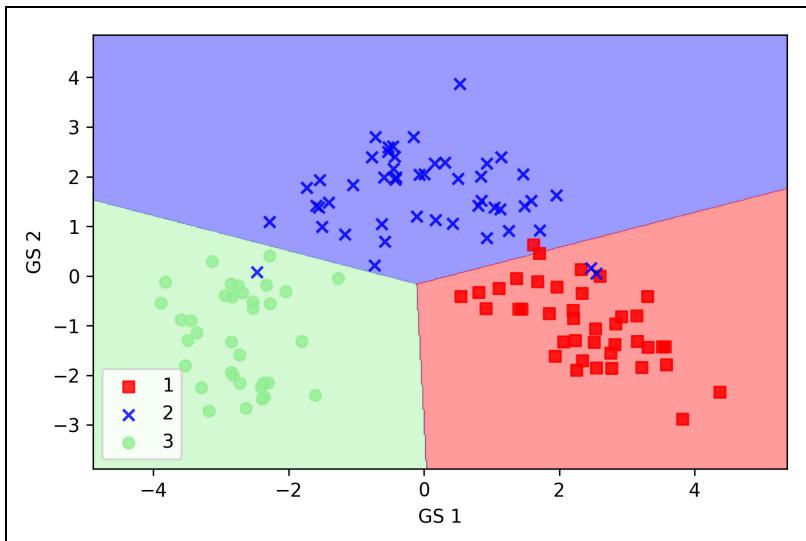
    # rysuje wykres powierzchni decyzyjnej
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # rysuje wykres próbek
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                    alpha=0.8, c=cmap(idx),
                    marker=markers[idx], label=cl)

>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=2)
>>> lr = LogisticRegression()
>>> X_train_pca = pca.fit_transform(X_train_std)
>>> X_test_pca = pca.transform(X_test_std)
```

```
>>> lr.fit(X_train_pca, y_train)
>>> plot_decision_regions(X_train_pca, y_train, classifier=lr)
>>> plt.xlabel('GS1')
>>> plt.ylabel('GS2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

Po uruchomieniu powyższego kodu naszym oczom powinien ukazać się wykres regionów decyzyjnych dla modelu uczącego zredukowanego do dwóch osi głównych składowych (rysunek 5.4).

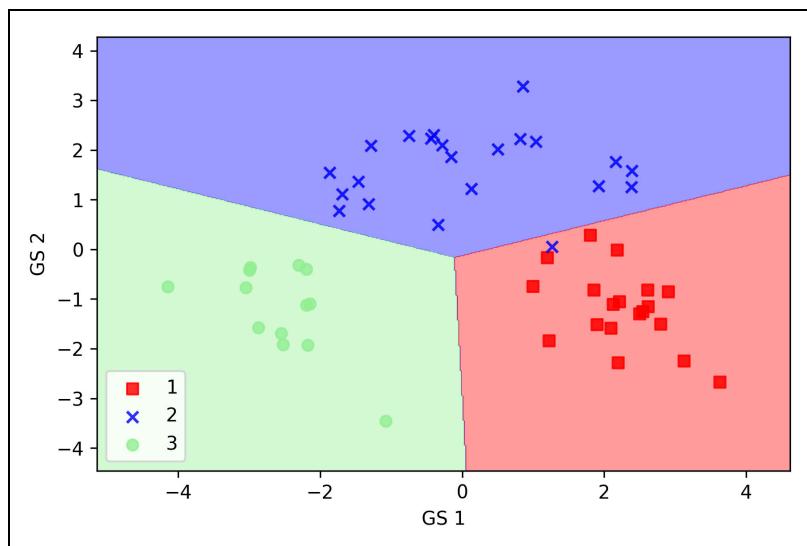


Rysunek 5.4. Wykres regionów decyzyjnych dla próbek rzutowanych na przestrzeń dwóch głównych składowych

Jeżeli porównamy rzutowanie PCA utworzone za pomocą biblioteki scikit-learn z naszą własnoręczną implementacją, zauważmy, że wykres z rysunku 5.4 stanowi lustrzane odbranie wykresu zilustrowanego na rysunku 5.3. Zwrót uwagi, że nie jest za to odpowiedzialny błąd którejś implementacji, lecz wynika to z faktu, że w zależności od funkcji obliczającej czynniki własne wektory własne mogą mieć różne znaki. Gdybyśmy chcieli „odwrócić” wykres, wystarczyłoby przemnożyć dane przez -1 (choćż nie ma to żadnego znaczenia dla analizy danych); zauważmy, że wektory własne są zazwyczaj skalowane do jednostek o skoku równym 1. Gwoli dopełnienia narysujmy wykres regionów decyzyjnych wyliczonych przez algorytm regresji logistycznej dla przekształconego zestawu danych testowych, żeby się przekonać, czy poszczególne klasy są właściwie rozdzielane:

```
>>> plot_decision_regions(X_test_pca, y_test, classifier=lr)
>>> plt.xlabel('GS 1')
>>> plt.ylabel('GS 2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

Po uruchomieniu powyższego fragmentu kodu przekonujemy się, że algorytm regresji logistycznej sprawdza się zupełnie nieźle w tej niewielkiej, dwuwymiarowej podprzestrzeni cech i nieprawidłowo klasyfikuje tylko jedną próbkę (rysunek 5.5).



Rysunek 5.5. Wykres regionów decyzyjnych dla próbek testowych rzutowanych na nową podprzestrzeń cech

Jeżeli interesują nas współczynniki wariancji wyjaśnionej dla różnych głównych składowych, wystarczy zainicjować klasę PCA z ustawioną wartością `None` dla parametru `n_components`, co spowoduje przechowywanie wszystkich głównych składowych; teraz możemy uzyskać dostęp do współczynnika wariancji wyjaśnionej za pomocą atrybutu `explained_variance_ratio`:

```
>>> pca = PCA(n_components=None)
>>> X_train_pca = pca.fit_transform(X_train_std)
>>> pca.explained_variance_ratio_
array([ 0.37329648,  0.18818926,  0.10896791,  0.07724389,
       0.06478595,
       0.04592014,  0.03986936,  0.02521914,  0.02258181,  0.01830924,
       0.01635336,  0.01284271,  0.00642076])
```

Zauważ, że ustawiamy `n_components=None` w czasie inicjowania klasy PCA, dzięki czemu zostają zwrócone wszystkie główne składowe w uporządkowanej kolejności i unikamy w ten sposób redukcji wymiarowości.

# Nadzorowana kompresja danych za pomocą liniowej analizy dyskryminacyjnej

Liniowa analiza dyskryminacyjna (ang. *linear discriminant analysis* — LDA) bywa stosowana jako technika odkrywania cech do poprawiania skuteczności obliczeniowej i zmniejszenia przetrenowania wynikającego z kłaty wymiarowości w nieregularyzowanych modelach.

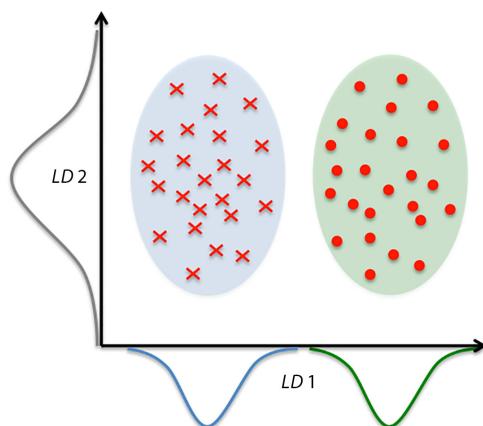
Ogólna koncepcja analizy LDA bardzo przypomina technikę PCA; w przypadku analizy głównych składowych staramy się znaleźć ortogonalne osie składowych określających maksymalną wariancję zestawu danych; z kolei celem liniowej analizy dyskryminacyjnej jest zdefiniowanie takiej podprzestrzeni cech, która umożliwiałaby optymalne rozdzielenie klas. Obydwa typy analiz należą do kategorii technik transformacji liniowej pozwalających na zmniejszenie wymiarowości zbioru danych; analiza PCA jest nienadzorowana, a LDA — nadzorowana. Dlatego intuicja może nam podpowiadać, że to drugie rozwiązanie nadaje się bardziej do odkrywania cech w modelach klasyfikacji. Jednak Aleix M. Martinez stwierdził, że wstępne przetwarzanie danych za pomocą modelu PCA w pewnych przypadkach daje lepsze wyniki klasyfikacji w procesie rozpoznawania obrazów, np. gdy każda klasa składa się z niewielkiej liczby próbek (A.M. Martinez i A.C. Kak, *PCA Versus LDA, Pattern Analysis and Machine Intelligence*, „IEEE Transactions” 2001, nr 23 (2), s. 228 – 233).

Technika LDA jest czasami nazywana liniową analizą dyskryminacyjną Fishera, ale Ronald A. Fisher pierwotnie sformułował liniową dyskryminację Fishera dla zagadnień klasyfikacji dwuklasowej w 1936 roku (R.A. Fisher, *The Use of Multiple Measurements in Taxonomic Problems*, „Annals of Eugenics” 1936, nr 7 (2), s. 179 – 188). Liniowa dyskryminacja Fishera została uogólniona do problemów wieloklasowych (przy założeniu równorzędnych kowariancji klas oraz rozkładzie normalnym klas) przez Calyampudi Radhakrishnę Rao w 1948 roku; to właśnie ten model znamy jako liniową dyskryminację klas (C.R. Rao, *The Utilization of Multiple Measurements in Problems of Biological Classification*, „Journal of the Royal Statistical Society. Series B (Methodological)” 1948, nr 10 (2), s. 159 – 203).

Na rysunku 5.6 prezentuję koncepcję analizy LDA dla problemu dwuklasowego. Próbki należące do klasy 1 zostały oznaczone krzyżykami, a do klasy 2 — kropkami.

Jak widzimy na rysunku 5.6, liniowa dyskryminanta może dobrze rozdzielać objęte rozkładem normalnym klasy wzduż osi  $x$  (*LD 1*). Chociaż przykładowa dyskryminanta liniowa wzduż osi  $y$  (*LD 2*) zawiera znaczną część wariancji zestawu danych, nie jest ona zbyt skuteczna, ponieważ nie ukazuje żadnych informacji rozróżniających poszczególne klasy.

Jednym z założeń analizy LDA jest rozkład normalny danych. Zakładamy również, że poszczególne klasy mają takie same macierze kowariancji, a cechy są od siebie statystycznie niezależne. Jednak nawet pomimo znacznego naruszenia któregoś z tych założeń redukcja wymiarowości zapewniana przez liniową analizę dyskryminacyjną zazwyczaj sprawuje się wystarczająco dobrze (R.O. Duda, P.E. Hart i D.G. Stork, *Pattern Classification. 2nd Edition*, New York: Wiley, 2001).



Rysunek 5.6. Ilustracja zasady działania liniowej analizy dyskryminacyjnej

Zanim przystąpimy do omówienia mechanizmów działania analizy LDA, podsumujmy główne etapy algorytmu:

1. Standaryzowanie  $d$ -wymiarowego zestawu danych ( $d$  jest liczbą cech).
2. Obliczenie  $d$ -wymiarowego wektora średnich dla każdej klasy.
3. Utworzenie międzyklasowej macierzy rozproszenia  $S_B$  oraz wewnętrzklasowej macierzy rozproszenia  $S_W$ .
4. Wyliczenie wektorów własnych i powiązanych z nimi wartości własnych z macierzy  $S_W^{-1}S_B$ .
5. Wybór  $k$  wektorów własnych odpowiadających  $k$  największym wartościom własnym w celu utworzenia  $d \times k$ -wymiarowej macierzy transformacji  $\mathbf{W}$ ; wektory własne tworzą kolumny tej macierzy.
6. Rzutowanie próbek na nową podprzestrzeń cech za pomocą macierzy transformacji  $\mathbf{W}$ .

Podczas stosowania analizy LDA zakładamy, że cechy mają rozkład normalny i są od siebie wzajemnie niezależne. Ponadto algorytm liniowej analizy dyskryminacyjnej wymaga, aby macierze kowariancji poszczególnych klas były identyczne. Jednak nawet pomimo znacznego naruszenia któregoś z tych założeń redukcja wymiarowości zapewniana przez liniową analizę dyskryminacyjną zazwyczaj sprawuje się wystarczająco dobrze (R.O. Duda, P.E. Hart i D.G. Stork, *Pattern Classification. 2nd Edition*, New York: Wiley, 2001).

## Obliczanie macierzy rozproszenia

Na początku rozdziału dokonaliśmy standaryzacji cech zestawu danych Wine, dlatego możemy pominać pierwszy etap i przejść do obliczenia wektorów średnich, za pomocą których utworzymy wewnętrzklasowe i międzyklasowe macierze rozproszenia. Każdy wektor średnich  $\mathbf{m}_i$  przechowuje uśrednioną wartość cechy  $\mu_m$  w odniesieniu do próbki klasy  $i$ :

$$\mathbf{m}_i = \frac{1}{n_i} \sum_{x \in D_i}^c \mathbf{x}_m$$

Otrzymujemy w ten sposób trzy wektory średnich:

$$\mathbf{m}_i = \begin{bmatrix} \mu_{i,Alkohol} \\ \mu_{i,Kwas\;jablkowy} \\ \vdots \\ \mu_{i,Prolina} \end{bmatrix}^T = i \in \{1, 2, 3\}$$

```
>>> np.set_printoptions(precision=4)
>>> mean_vecs = []
>>> for label in range(1,4):
...     mean_vecs.append(np.mean(
...         X_train_std[y_train==label], axis=0))
...     print('UW %s: %s\n' %(label, mean_vecs[label-1]))
UW 1: [ 0.9259 -0.3091  0.2592 -0.7989  0.3039  0.9608  1.0515 -0.6306
0.5354
 0.2209  0.4855  0.798  1.2017]

UW 2: [-0.8727 -0.3854 -0.4437  0.2481 -0.2409 -0.1059  0.0187 -0.0164
0.1095
 -0.8796  0.4392  0.2776 -0.7016]

UW 3: [ 0.1637  0.8929  0.3249  0.5658 -0.01 -0.9499 -1.228  0.7436
-0.7652
 0.979 -1.1698 -1.3007 -0.3912]
```

Korzystając z wektorów średnich, możemy teraz obliczyć wewnątrzklasową macierz rozproszenia  $\mathbf{S}_W$ :

$$\mathbf{S}_W = \sum_{i=1}^c \mathbf{S}_i$$

Dokonujemy tego, sumując pojedyncze macierze rozproszenia  $\mathbf{S}_i$  poszczególnych klas  $i$ :

```
>>> d = 13 # liczba cech
>>> S_W = np.zeros((d, d))
>>> for label,mv in zip(range(1,4), mean_vecs):
...     class_scatter = np.zeros((d, d))
...     for row in X_train[y_train == label]:
...         row, mv = row.reshape(d, 1), mv.reshape(d, 1)
...         class_scatter += (row-mv).dot((row-mv).T)
...     S_W += class_scatter
```

```
>>> print('Wewnętrzklasowa macierz rozproszenia: %sx%s'
...      % (S_W.shape[0], S_W.shape[1]))
Wewnętrzklasowa macierz rozproszenia: 13x13
```

Podczas obliczania macierzy rozproszenia przyjmujemy założenie, że etykiety klas w zestawie danych uczących mają rozkład normalny. Jeżeli jednak sprawdzimy liczbę etykiet klas, przekonamy się, że to założenie nie jest spełnione:

```
>>> print('Rozkład etykiet klas: %s'
...      % np.bincount(y_train)[1:])
Rozkład etykiet klas: [40 49 35]
```

Chcemy zatem wyskalować pojedyncze macierze rozproszenia  $S_i$  przed ich zsumowaniem do macierzy  $S_W$ . Podzieliwszy macierze rozproszenia przez liczbę klas  $N_i$ , możemy się przekonać, że obliczanie macierzy rozproszenia przebiega w istocie tak samo jak wyliczanie macierzy kowariancji  $\sigma_{ij}$ . Macierz kowariancji stanowi znormalizowaną wersję macierzy rozproszenia:

$$\Sigma_i = \frac{1}{N_i} S_W = \frac{1}{N_i} \sum_{x \in D_i}^c (x - \mathbf{m}_i)(x - \mathbf{m}_i)^T$$

```
>>> d = 13 # liczba cech
>>> S_W = np.zeros((d, d))
>>> for label, mv in zip(range(1, 4), mean_vecs):
...     class_scatter = np.cov(X_train_std[y_train==label].T)
...     S_W += class_scatter
>>> print('Skalowana wewnętrzklasowa macierz rozproszenia: %sx%s'
...       % (S_W.shape[0], S_W.shape[1]))
Skalowana wewnętrzklasowa macierz rozproszenia: 13x13
```

Po wyliczeniu skalowanej wewnętrzklasowej macierzy rozproszenia (lub macierzy kowariancji) możemy przejść do następnego etapu i obliczyć międzyklasową macierz rozproszenia  $S_B$ :

$$S_B = \sum_{i=1}^c N_i (\mathbf{m}_i - \mathbf{m})(\mathbf{m}_i - \mathbf{m})^T$$

Tutaj  $\mathbf{m}$  oznacza całkowitą średnią wyliczoną również z próbek znajdujących się we wszystkich klasach:

```
>>> mean_overall = np.mean(X_train_std, axis=0)
>>> d = 13 # liczba cech
>>> S_B = np.zeros((d, d))
>>> for i, mean_vec in enumerate(mean_vecs):
...     n = X_train[y_train==i+1, :].shape[0]
...     mean_vec = mean_vec.reshape(d, 1)
...     mean_overall = mean_overall.reshape(d, 1)
...     S_B += n * (mean_vec - mean_overall).dot(
...                 (mean_vec - mean_overall).T)
print('Międzyklasowa macierz rozproszenia: %sx%s'
...     % (S_B.shape[0], S_B.shape[1]))
Międzyklasowa macierz rozproszenia: 13x13
```

## Dobór dyskryminant liniowych dla nowej podprzestrzeni cech

Pozostałe etapy analizy LDA są podobne do analogicznych kroków w metodzie PCA. Nie będziemy jednak dokonywać rozkładu macierzy kowariancji, lecz rozwiążemy uogólniony problem wartości własnych dla macierzy  $S_W^{-1}S_B$ :

```
>>> eigen_vals, eigen_vecs = \
... np.linalg.eig(np.linalg.inv(S_W).dot(S_B))
```

Po wyliczeniu par własnych możemy uszeregować wartości własne w malejącej kolejności:

```
>>> eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:,i])
...                 for i in range(len(eigen_vals))]
>>> eigen_pairs = sorted(eigen_pairs,
...                       key=lambda k: k[0], reverse=True)
>>> print('Wartości własne w malejącej kolejności:\n')
>>> for eigen_val in eigen_pairs:
...     print(eigen_val[0])
```

*Wartości własne w malejącej kolejności:*

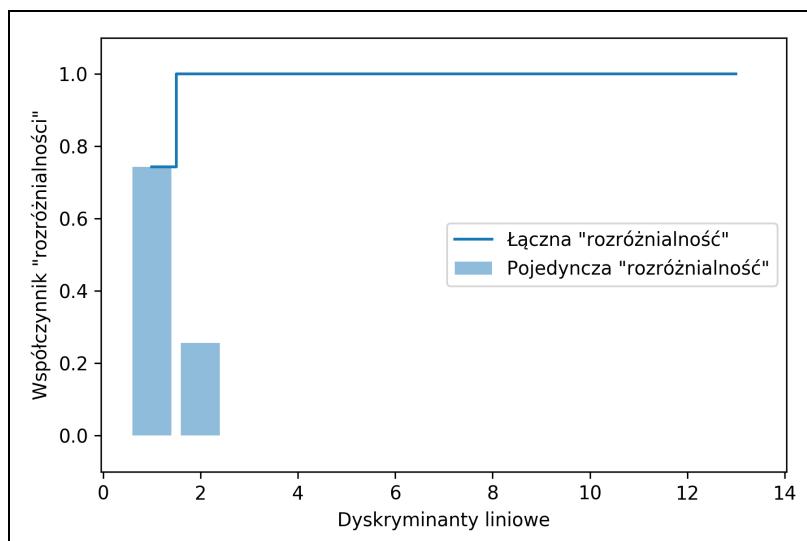
```
452.721581245
156.43636122
8.11327596465e-14
2.78687384543e-14
2.78687384543e-14
2.27622032758e-14
2.27622032758e-14
1.97162599817e-14
1.32484714652e-14
1.32484714652e-14
1.03791501611e-14
5.94140664834e-15
2.12636975748e-16
```

W analizie LDA liczba liniowych dyskryminant wynosi co najwyżej  $c-1$ , gdzie  $c$  jest liczbą etykiet klas, gdyż międzyklasowa macierz rozproszenia  $S_B$  stanowi sumę  $c$  macierzy pierwszego lub niższego rzędu. Istotnie, widzimy, że mamy tylko dwie niezerowe wartości własne (wartości własne 3 – 13 nie są równe 0, ale wynika to z arytmetyki zmienoprzecinkowej biblioteki NumPy). Zwróci uwagę, że w rzadko występującej sytuacji doskonalej współliniowości (wszystkie punkty reprezentujące powiązane próbki ulozone są w linii prostej) macierz kowariancji byłaby pierwszego rzędu, w wyniku czego otrzymalibyśmy tylko jeden wektor własny zawierający jedną niezerową wartość własną.

Aby zmierzyć rozmiar rozróżniających informacji przechowywanych w liniowych dyskryminantach (wektorach własnych), wygenerujemy wykres dyskryminant dla malejących wartości własnych, podobnie jak tego dokonaliśmy w podrozdziale poświęconym analizie PCA. Dla uproszczenia nazwiemy informacje przenoszone przez te wektory **rozsźnialnością**.

```
>>> tot = sum(eigen_vals.real)
>>> discr = [(i / tot) for i in sorted(eigen_vals.real, reverse=True)]
>>> cum_discr = np.cumsum(discr)
>>> plt.bar(range(1, 14), discr, alpha=0.5, align='center',
...           label='Pojedyncza "rozsźnialność"')
>>> plt.step(range(1, 14), cum_discr, where='mid',
...           label='Łączna "rozsźnialność"')
>>> plt.ylabel('Współczynnik "rozsźnialności"')
>>> plt.xlabel('Dyskryminanty liniowe')
>>> plt.ylim([-0.1, 1.1])
>>> plt.legend(loc='best')
>>> plt.show()
```

Jak widać na rysunku 5.7, dwie pierwsze dyskryminanty liniowe zawierają niemal 100% przydatnych informacji przechowywanych w zestawie danych **Wine**.



Rysunek 5.7. Wykres ilości informacji przechowywanych w dwóch pierwszych dyskryminantach liniowych

Złączmy teraz kolumny dwóch wektorów własnych zawierających najwięcej informacji rozróżniających w celu utworzenia macierzy transformacji **W**:

```
>>> w = np.hstack((eigen_pairs[0][1][:, np.newaxis].real,
...                  eigen_pairs[1][1][:, np.newaxis].real))
>>> print('Macierz W:\n', w)
Macierz W:
```

```
[[ 0.0662 -0.3797]
[-0.0386 -0.2206]
[ 0.0217 -0.3816]
[-0.184 0.3018]
[ 0.0034 0.0141]
[-0.2326 0.0234]
[ 0.7747 0.1869]
[ 0.0811 0.0696]
[-0.0875 0.1796]
[-0.185 -0.284 ]
[ 0.066 0.2349]
[ 0.3805 0.073 ]
[ 0.3285 -0.5971]]
```

## Rzutowanie próbek na nową przestrzeń cech

Z pomocą utworzonej w poprzednim ustępie macierzy transformacji  $W$  możemy przekształcić próbki, mnożąc dwie macierze:

$$X' = XW$$

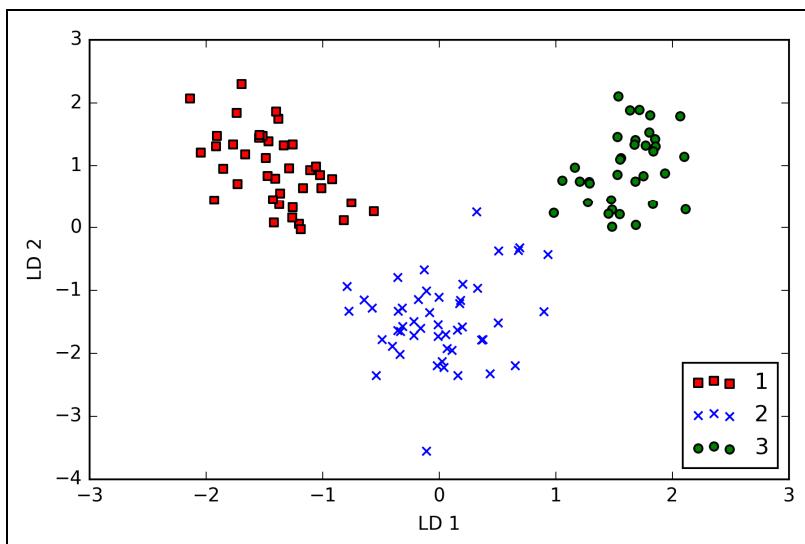
```
>>> X_train_lda = X_train_std.dot(w)
>>> colors = ['r', 'b', 'g']
>>> markers = ['s', 'x', 'o']
>>> for l, c, m in zip(np.unique(y_train), colors, markers):
...     plt.scatter(X_train_lda[y_train==l, 0]*(-1),
...                 X_train_lda[y_train==l, 1]*(-1),
...                 c=c, label=l, marker=m)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower right')
>>> plt.show()
```

Widzimy na uzyskanym wykresie (rysunek 5.8), że trzy klasy zestawu danych Wine są teraz liniowo rozdzielne w nowej podprzestrzeni cech.

## Implementacja analizy LDA w bibliotece scikit-learn

Samodzielna implementacja analizy LDA pozwoliła nam zajrzeć w mechanizmy rządzące omawianym procesem oraz ułatwiła zrozumienie różnic pomiędzy liniową analizą dyskryminacyjną a analizą głównych składowych. Zajmijmy się teraz klasą LDA zaimplementowaną w interfejsie scikit-learn:

```
>>> if Version(sklearn_version) < '0.18':
...     from sklearn.lda import LDA
... else:
```



Rysunek 5.8. Wykres próbek rzutowanych na nowej podprzestrzeni cech

```
>>> from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
->as LDA
>>> lda = LDA(n_components=2)
>>> X_train_lda = lda.fit_transform(X_train_std, y_train)
```

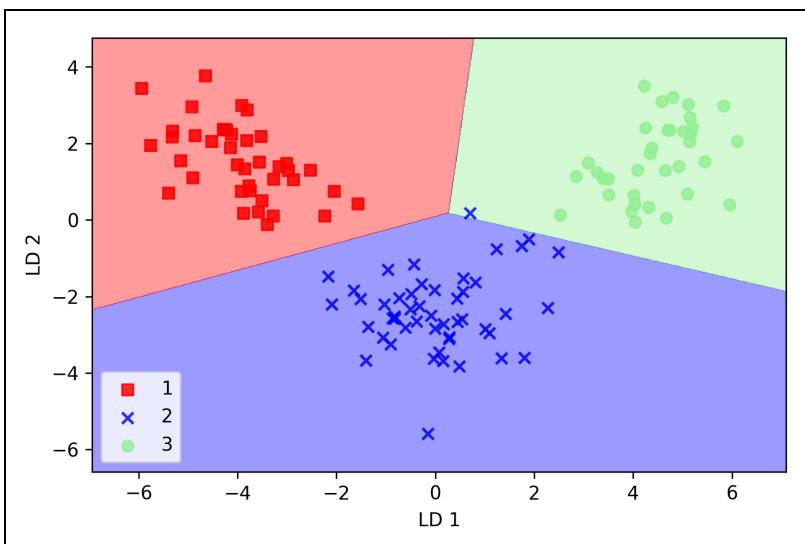
Sprawdźmy, jak algorytm regresji logistycznej poradzi sobie z klasyfikacją zestawu danych przeniesionego na podprzestrzeń o mniejszej liczbie wymiarów po przeprowadzeniu transformacji LDA:

```
>>> lr = LogisticRegression()
>>> lr = lr.fit(X_train_lda, y_train)
>>> plot_decision_regions(X_train_lda, y_train, classifier=lr)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

Przyglądając się wykresowi zaprezentowanemu na rysunku 5.9, możemy się przekonać, że model regresji logistycznej nieprawidłowo klasyfikuje jedną próbke z klasy 2.

Zmniejszając siłę regularyzacji, prawdopodobnie jesteśmy w stanie przesunąć granice decyzyjne w taki sposób, że algorytm regresji logistycznej poprawnie sklasyfikowałby wszystkie próbki uczące. Zobaczmy jednak, jak sobie poradzi z zestawem danych testowych:

```
>>> X_test_lda = lda.transform(X_test_std)
>>> plot_decision_regions(X_test_lda, y_test, classifier=lr)
>>> plt.xlabel('LD 1')
```



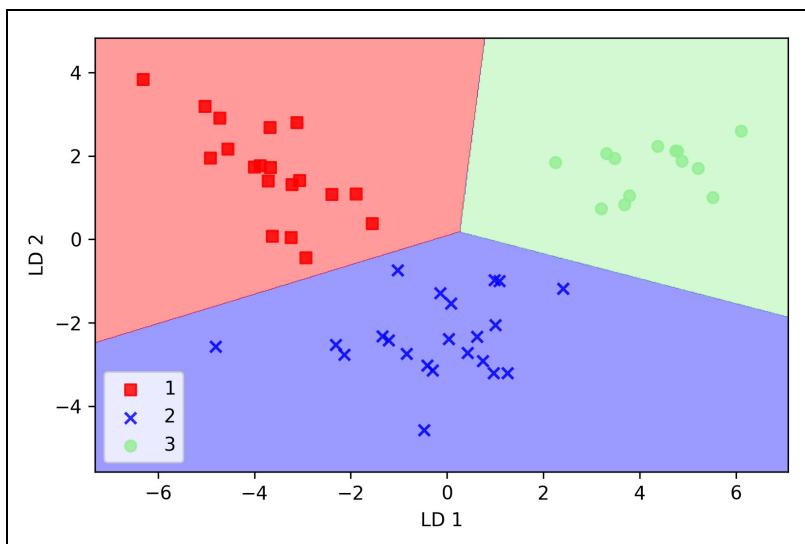
Rysunek 5.9. Wykres regresji logistycznej klasyfikującej próbki rzutowane na przestrzeń opracowaną za pomocą analizy LDA

```
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

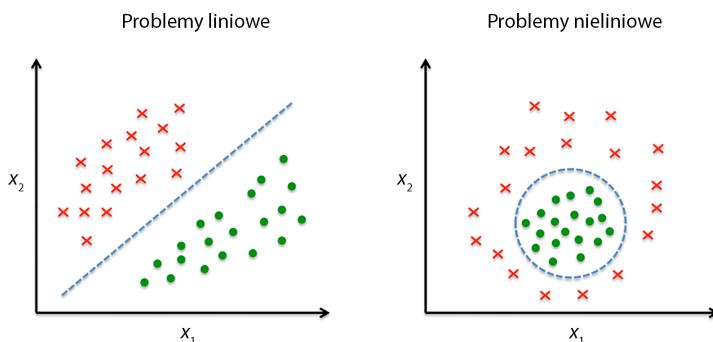
Wykres na rysunku 5.10 dowodzi, że algorytm regresji logistycznej wykazuje doskonałą skuteczność w klasyfikowaniu próbek zestawu testowego wyłącznie na podstawie dwuwymiarowej podprzestrzeni cech i nie potrzebuje w tym celu wszystkich 13 cech zbioru danych Wine.

## Jądrowa analiza głównych składowych jako metoda odwzorowywania nierozdzielnych liniowo klas

Wiele algorytmów uczenia maszynowego bazuje na założeniu liniowej rozdzielnosci wprowadzanych danych. Wiemy już, że model perceptronu do uzyskania zbieżności wymaga wręcz doskonale rozdzielnych liniowo danych uczących. W innych dotyczcych omówionym algorytmach zakładamy, że brak doskonalej rozdzielnosci liniowej próbek spowodowany jest zaszu-mieniem układu: możemy wymienić tu choćby model Adaline, regresji logistycznej oraz (standardową) maszynę wektorów nośnych. Jednak w przypadku problemów nieliniowych, dość często występujących w prawdziwym świecie, techniki transformacji liniowej służące do redukcji wymiarowości, takie jak PCA czy LDA nierzaz okazują się nieskuteczne (rysunek 5.11). W tym



Rysunek 5.10. Wykres klasyfikatora regresji logistycznej analizującego próbki testowe rzutowane na dwuwymiarową podprzestrzeń cech za pomocą analizy LDA



Rysunek 5.11. Wykresy danych rozdzielnych i nierozdzielnych liniowo

podrozdziale zajmiemy się kernelizowaną wersją analizy PCA, czyli **jądrową analizą głównych składowych** (ang. *kernel PCA*), która koncepcyjnie przypomina nieco jądrową maszynę wektorów nośnych, opisaną w rozdziale 3., „*Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn*”. Dzięki jądrowej analizie PCA nauczymy się rzutować nierozdzielne liniowo dane na nową podprzestrzeń o mniejszej liczbie wymiarów, nadającą się idealnie dla klasyfikatorów liniowych.

## Funkcje jądra oraz sztuczka z funkcją jądra

Jak pamiętamy z opisu jądrowych maszyn SVM w rozdziale 3., „Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn”, możemy rozwiązywać nieliniowe problemy, rzutując dane na nową, mającą większą liczbę wymiarów od pierwotnej, przestrzeń cech, gdzie klasy stają się liniowo rozdzielne. Aby przenieść próbki  $x \in \mathbb{R}^d$  do tej  $k$ -wymiarowej podprzestrzeni, zdefiniowaliśmy nieliniową funkcję odwzorowującą  $\phi$ :

$$\phi: \mathbb{R}^d \rightarrow \mathbb{R}^k \quad (k >> d)$$

Funkcję  $\phi$  możemy interpretować w taki sposób, że tworzy ona nieliniowe kombinacje pierwotnych cech, które są następnie rzutowane z początkowego,  $d$ -wymiarowego zbioru danych na większą,  $k$ -wymiarową przestrzeń cech. Przykładowo, mając dany wektor cech  $x \in \mathbb{R}^d$  ( $x$  jest wektorem kolumnowym składającym się z  $d$  cech) opisany dwoma wymiarami ( $d = 2$ ), możemy go rzutować na potencjalną trójwymiarową przestrzeń w następujący sposób:

$$\begin{aligned} x &= [x_1, x_2]^T \\ &\downarrow \phi \\ z &= [x_1^2, \sqrt{2x_1x_2}, x_2^2]^T \end{aligned}$$

Innymi słowy, za pomocą jądrowej analizy PCA przeprowadzamy nieliniowe odwzorowywanie danych, umożliwiające ich rzutowanie na przestrzeń o większej liczbie wymiarów, z kolei poprzez standardową analizę głównych składowych powracamy do przestrzeni zawierającej mniej wymiarów, gdzie próbki zostają następnie rozdzielone przez liniowy klasyfikator (pod warunkiem że próbki te można rozdzielić pod względem gęstości w przestrzeni początkowej). Jest to jednak proces bardzo złożony obliczeniowo, dlatego wprowadzamy **sztuczkę z funkcją jądra**. Dzięki niej jesteśmy w stanie wyliczać podobieństwo pomiędzy dwoma wielowymiarowymi wektorami cech umieszczonymi w pierwotnej przestrzeni cech.

Zanim przeanalizujemy rolę sztuczki z funkcją jądra w rozwiązyaniu tego kosztownego pod względem obliczeniowym problemu, przyjrzyjmy się **standardowej** analizie PCA omówionej na początku niniejszego rozdziału. Kowariancję pomiędzy dwiema cechami  $j$  i  $k$  obliczyliśmy przy użyciu poniższego wzoru:

$$\sigma_{jk} = \frac{1}{n} \sum_{i=1}^n (x_j^{(i)} - \mu_j)(x_k^{(i)} - \mu_k)$$

Poprzez standaryzację cech uzyskują one uśrednioną wartość 0 (np.  $\mu_j = 0$  i  $\mu_k = 0$ ); możemy uprościć powyższy wzór:

$$\sigma_{jk} = \frac{1}{n} \sum_{i=1}^n x_j^{(i)} x_k^{(i)}$$

Zwróć uwagę, że powyższe równanie dotyczy kowariancji dwóch cech; zapiszmy teraz ogólny wzór służący do obliczania **macierzy kowariancji** :

$$\Sigma = \frac{1}{n} \sum_{i=1}^n \mathbf{x}_j^{(i)} \mathbf{x}_k^{(i)T}$$

Bernhard Schölkopf dokonał uogólnienia tego rozwiązania (B. Schölkopf, A. Smola i K.R. Müller, *Kernel Principal Component Analysis*, 1997, s. 583 – 588), przez co możemy za pomocą funkcji  $\phi$  podmienić iloczyny skalarne próbek w pierwotnej przestrzeni cech na nieliniowe kombinacje cech:

$$\Sigma = \frac{1}{n} \sum_{i=1}^n \phi(\mathbf{x}^{(i)}) \phi(\mathbf{x}^{(i)})^T$$

W celu uzyskania wektorów własnych — głównych składowych — z tej macierzy kowariancji musimy rozwiązać następujące równanie:

$$\Sigma \mathbf{v} = \lambda \mathbf{v}$$

$$\Rightarrow \frac{1}{n} \sum_{i=1}^n \phi(\mathbf{x}^{(i)}) \phi(\mathbf{x}^{(i)})^T \mathbf{v} = \lambda \mathbf{v}$$

$$\Rightarrow \mathbf{v} = \frac{1}{n\lambda} \sum_{i=1}^n \phi(\mathbf{x}^{(i)}) \phi(\mathbf{x}^{(i)})^T \mathbf{v} = \frac{1}{n} \sum_{i=1}^n \mathbf{a}^{(i)} \phi(\mathbf{x}^{(i)})$$

Tutaj  $\lambda$  i  $\mathbf{v}$  to wartości własne i wektory własne macierzy kowariancji  $\Sigma$ . Wartości wektora  $\mathbf{a}$  użyjemy poprzez wydobycie wektorów własnych z macierzy jądra (podobieństwa)  $\mathbf{K}$ , o czym przekonamy się w dalszej części rozdziału.

Macierz jądra wprowadzamy w następujący sposób:

Zapiszmy najpierw macierz kowariancji w notacji macierzowej, gdzie  $\phi(\mathbf{X})$  oznacza macierz o wymiarach  $n \times k$ :

$$\Sigma = \frac{1}{n} \sum_{i=1}^n \phi(\mathbf{x}^{(i)}) \phi(\mathbf{x}^{(i)})^T = \frac{1}{n} \phi(\mathbf{X})^T \phi(\mathbf{X})$$

Teraz możemy zapisać równanie wektora własnego:

$$\mathbf{v} = \frac{1}{n} \sum_{i=1}^n \mathbf{a}^{(i)} \phi(\mathbf{x}^{(i)}) = \lambda \phi(\mathbf{X})^T \mathbf{a}$$

Ponieważ  $\mathbf{v} = \mathbf{v}$ , otrzymujemy:

$$\frac{1}{n} \phi(\mathbf{X})^T \phi(\mathbf{X}) \phi(\mathbf{X})^T \mathbf{a} = \lambda \phi(\mathbf{X})^T \mathbf{a}$$

Po przemnożeniu obydwu stron równania przez  $\phi(\mathbf{X})$  uzyskujemy następujący rezultat:

$$\begin{aligned} \frac{1}{n} \phi(\mathbf{X}) \phi(\mathbf{X})^T \phi(\mathbf{X}) \phi(\mathbf{X})^T \mathbf{a} &= \lambda \phi(\mathbf{X}) \phi(\mathbf{X})^T \mathbf{a} \\ \Rightarrow \frac{1}{n} \phi(\mathbf{X}) \phi(\mathbf{X})^T \mathbf{a} &= \lambda \mathbf{a} \\ \Rightarrow \frac{1}{n} \mathbf{K} \mathbf{a} &= \lambda \mathbf{a} \end{aligned}$$

$\mathbf{K}$  jest naszą macierzą podobieństwa (jądra):

$$\mathbf{K} = \phi(\mathbf{X}) \phi(\mathbf{X})^T$$

Jak pamiętamy z rozdziału 3., „Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn”, sztuczkę z funkcją jądra wykorzystujemy w celu uniknięcia jawnego obliczania par iloczynów skalarnych próbek  $\mathbf{x}$  wewnątrz funkcji  $\phi$  i dlatego wprowadzamy funkcję jądra  $\mathbf{K}$ , dzięki której nie musimy jawnie wyliczać wektorów własnych:

$$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \phi(\mathbf{x}^{(i)})^T \phi(\mathbf{x}^{(j)})$$

Krótko mówiąc, w wyniku jądrowej analizy głównych składowych otrzymujemy próbki już rzutowane na odpowiednią podprzestrzeń wektorów własnych; w standardowej analizie PCA natomiast tworzyliśmy w tym celu macierz transformacji. Funkcję jądra (zwaną również **jądrem**) możemy interpretować jako funkcję obliczającą iloczyn skalarny dwóch wektorów — miarę podobieństwa.

Najpopularniejszymi funkcjami jądra są:

- wielomianowa funkcja jądra:
- $$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = (\mathbf{x}^{(i)T} \mathbf{x}^{(j)} + \theta)^p$$
- gdzie  $\theta$  oznacza parametr progowy, a  $p$  wykładnik potęgi wyznaczany przez użytkownika;
- jądro tangensa hiperbolicznego (sigmoidalna funkcja jądra):

$$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \tanh(\eta \mathbf{x}^{(i)T} \mathbf{x}^{(j)} + \theta)$$

- **jądro radialnej funkcji bazowej** (ang. *radial basis function* — **RBФ**), czyli jądro gaussowskie, które wykorzystamy w przykładowej implementacji:

$$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp\left(-\frac{\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2}{2\sigma^2}\right)$$

co możemy również zapisać następująco:

$$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp(-\gamma \|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2)$$

W ramach podsumowania zdefiniujmy poszczególne etapy implementacji jądra RBF w analizie głównych składowych:

1. Obliczamy macierz jądra (podobieństwa)  $k$  za pomocą wzoru:

$$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp\left(-\gamma \|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2\right)$$

Dokonujemy tego dla każdej pary próbek:

$$\mathbf{K} = \begin{bmatrix} k(\mathbf{x}^{(1)}, \mathbf{x}^{(1)}) & k(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}) & \dots & k(\mathbf{x}^{(1)}, \mathbf{x}^{(n)}) \\ k(\mathbf{x}^{(2)}, \mathbf{x}^{(1)}) & k(\mathbf{x}^{(2)}, \mathbf{x}^{(2)}) & \dots & k(\mathbf{x}^{(2)}, \mathbf{x}^{(n)}) \\ \vdots & \vdots & \ddots & \vdots \\ k(\mathbf{x}^{(n)}, \mathbf{x}^{(1)}) & k(\mathbf{x}^{(n)}, \mathbf{x}^{(2)}) & \dots & k(\mathbf{x}^{(n)}, \mathbf{x}^{(n)}) \end{bmatrix}$$

Jeżeli nasz zestaw danych składałby się np. ze 100 próbek uczących, symetryczna macierz jądra par podobieństw miałaby wymiar  $100 \times 100$ .

2. Centrujemy macierz jądra  $k$  przy użyciu następującego równania:

$$\mathbf{K}' = \mathbf{K} - \mathbf{1}_n \mathbf{K} - \mathbf{K} \mathbf{1}_n + \mathbf{1}_n \mathbf{K} \mathbf{1}_n$$

Tutaj  $\mathbf{I}_n$  oznacza  $n \times n$ -wymiarową macierz (o takim samym rozmiarze jak macierz jądra), w której wszystkie wartości są równe  $\frac{1}{n}$ .

3. Na podstawie odpowiednich wartości własnych (uszeregowanych w malejącej kolejności) dobieramy  $k$  największych wektorów własnych wycentrowanej macierzy jądra. W przeciwnieństwie do standardowej analizy PCA tutaj wektory własne nie są osiami głównych składowych, lecz próbками rzutowanymi na te osie.

Być może zastanawiasz się teraz, dlaczego w etapie 2. wycentrowaliśmy macierz jądra. Założyliśmy uprzednio, że pracujemy na standaryzowanych danych, a podczas tworzenia macierzy kowariancji wszystkie cechy mają średnioną wartość 0, po czym za pomocą funkcji  $\phi$  zastąpiliśmy iloczyny skalarne nieliniowymi kombinacjami cech. Z tego powodu centrowanie macierzy jądra staje się koniecznością, gdyż nie wyliczamy w jawnym sposób nowej przestrzeni cech i nie mamy gwarancji, że nowo utworzona podprzestrzeń będzie również miała środek w punkcie 0 układu współrzędnych.

W kolejnym ustępie sprawdzimy wymienione powyżej etapy w akcji i zaimplementujemy jądrową analizę PCA w Pythonie.

# Implementacja jądrowej analizy głównych składowych w Pythonie

W poprzednim ustępie omówiliśmy mechanizm działania jądrowej analizy PCA. Teraz zajmiemy się implementacją jądra RBF w Pythonie — zaprogramujemy trzy etapy podsumowujące algorytm jądrowej analizy głównych składowych. Dzięki pomocniczym funkcjom bibliotek SciPy i NumPy stworzymy naprawdę prostą implementację jądra PCA:

```
from scipy.spatial.distance import pdist, squareform
from scipy import exp
from scipy.linalg import eigh
import numpy as np
def rbf_kernel_pca(X, gamma, n_components):
```

=====

*Implementacja jądra RBF w analizie PCA.*

*Parametry*

-----  
*X: {typ ndarray biblioteki NumPy}, wymiary = [n\_próbek, n\_cech]*

*gamma: liczby zmiennoprzecinkowe*

*Parametr strojenia jądra RBF*

*n\_components: liczby całkowite*

*Liczba zwracanych głównych składowych*

*Zwraca*

-----  
*X\_pc: {typ ndarray biblioteki NumPy}, wymiary = [n\_próbek, k\_cech]*  
*Rzutowany zestaw danych*

=====

*# oblicza kwadraty odległości euklidesowych par*

*# w zestawie danych o rozmiarze MxN*

*sq\_dists = pdist(X, 'squared')*

*# przekształca wyliczone odległości na macierz kwadratową*  
*mat\_sq\_dists = squareform(sq\_dists)*

*# oblicza symetryczną macierz jądra*  
*K = exp(-gamma \* mat\_sq\_dists)*

*# centruje macierz jądra*  
*N = K.shape[0]*  
*one\_n = np.ones((N,N)) / N*  
*K = K - one\_n.dot(K) - K.dot(one\_n) + one\_n.dot(K).dot(one\_n)*

```
# wydobywa pary własne z centrowanej macierzy jądra
# funkcja numpy.eigh zwraca je w malejącej kolejności
eigvals, eigvecs = eigh(K)

# wybiera k największych wektorów własnych (rzutowanych próbek)
X_pc = np.column_stack((eigvecs[:, -i]
                         for i in range(1, n_components + 1)))
return X_pc
```

Jedną z wad stosowania jądrowej analizy PCA typu RBF do redukowania wymiarowości jest konieczność określenia parametru  $\gamma$  z góry. Dobieramy odpowiednią wartość tego parametru metodą prób i błędów, a najlepiej dokonać tego, korzystając z któregoś z algorytmów strojenia parametrów, np. przeszukiwania siatki (algorytmy te zostały omówione w rozdziale 6., „Najlepsze metody oceny modelu i strojenie parametryczne”).

## Przykład 1. Rozdzielenie sierpowatych kształtów

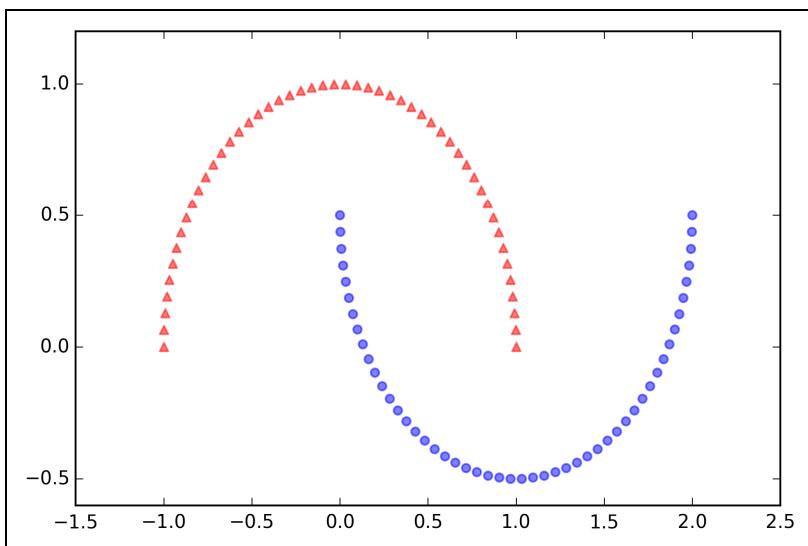
Przetestujmy nasz algorytm rbf\_kernel\_pca na nieliniowych zbiorach danych. Zaczniemy od utworzenia dwuwymiarowego zestawu 100 próbek tworzących na wykresie dwa sierpowate kształty:

```
>>> from sklearn.datasets import make_moons
>>> X, y = make_moons(n_samples=100, random_state=123)
>>> plt.scatter(X[y==0, 0], X[y==0, 1],
...                 color='red', marker='^', alpha=0.5)
>>> plt.scatter(X[y==1, 0], X[y==1, 1],
...                 color='blue', marker='o', alpha=0.5)
>>> plt.show()
```

W celach poglądowych próbki symbolizowane trójkątami należą do jednej klasy, a oznaczone kółkami stanowią zbiór należący do drugiej klasy (rysunek 5.12).

Jest oczywiste, że te dwa kształty nie są liniowo rozdzielne, a naszym celem jest **oddzielenie** ich za pomocą jądrowej analizy PCA w taki sposób, aby zbiór danych można było zaprezentować klasyfikatorowi liniowemu. Najpierw jednak przekonajmy się, jak ten zestaw danych będzie się prezentował w podprzestrzeni głównych składowych obliczonych za pomocą standardowej analizy PCA:

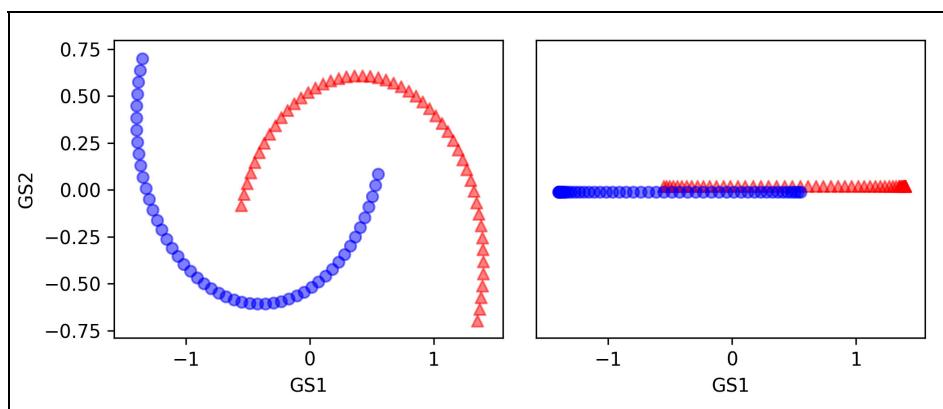
```
>>> from sklearn.decomposition import PCA
>>> scikit_pca = PCA(n_components=2)
>>> X_spca = scikit_pca.fit_transform(X)
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_spca[y==0, 0], X_spca[y==0, 1],
...                 color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_spca[y==1, 0], X_spca[y==1, 1],
...                 color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_spca[y==0, 0], np.zeros((50,1))+0.02,
...                 color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_spca[y==1, 0], np.zeros((50,1))-0.02,
...                 color='blue', marker='o', alpha=0.5)
... 
```



Rysunek 5.12. Wykres przykładowych danych nieliniowych mających sierpowaty kształt

```
>>> ax[0].set_xlabel('GS1')
>>> ax[0].set_ylabel('GS2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('GS1')
>>> plt.show()
```

Gdy spojrzymy na rysunek 5.13, stanie się jasne, że liniowy klasyfikator miałby olbrzymie problemy z przetwarzaniem danych przekształconych za pomocą standardowej analizy PCA.



Rysunek 5.13. Wykresy danych nieliniowo rozdzielnych rzutowanych na przestrzeń utworzoną poprzez standardową analizę głównych składowych

Zwróć uwagę, że na wykresie zawierającym tylko jedną główną składową (rysunek 5.13, po prawej), przesunąłem nieznacznie próbki z klasy 1 (trójkąty) do góry, a próbki z klasy 2 (kółka) w dół, aby lepiej ukazać wzajemne nakładanie się poszczególnych klas.

Pamiętaj, że analiza PCA należy do metod nienadzorowanych i, w przeciwieństwie do analizy LDA, nie wykorzystuje informacji zawartych w etykietach klas do zmaksymalizowania wariancji. W naszym przykładzie dodałem podział na trójkąty i kółka wyłącznie po to, aby zwizualizować nakładanie się klas.

Wypróbujmy teraz funkcję jądrowej analizy PCA (`rbf_kernel_pca`):

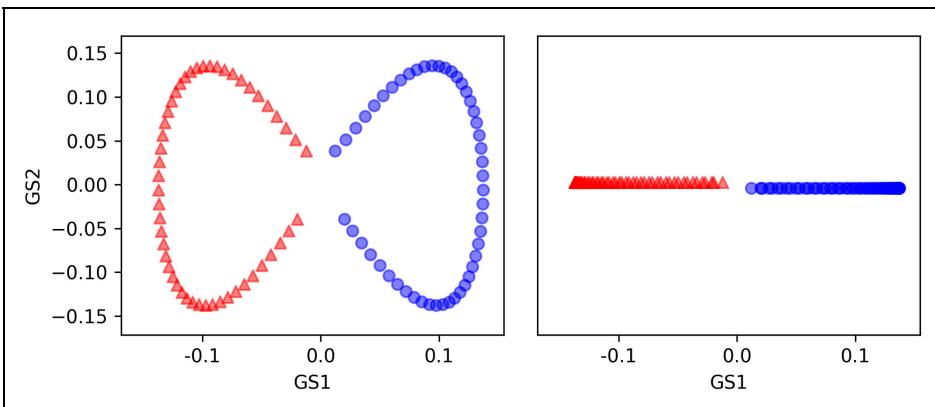
```
>>> from matplotlib.ticker import FormatStrFormatter
>>> X_kpca = rbf_kernel_pca(X, gamma=15, n_components=2)
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_kpca[y==0, 0], X_kpca[y==0, 1],
...                 color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_kpca[y==1, 0], X_kpca[y==1, 1],
...                 color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==0, 0], np.zeros((50,1))+0.02,
...                 color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==1, 0], np.zeros((50,1))-0.02,
...                 color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('GS1')
>>> ax[0].set_ylabel('GS2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('GS1')
>>> ax[0].xaxis.set_major_formatter(FormatStrFormatter('%0.1f'))
>>> ax[1].xaxis.set_major_formatter(FormatStrFormatter('%0.1f'))
>>> plt.show()
```

Jak widać na rysunku 5.14, obydwie klasy (trójkąty i kółka) stały się idealnie rozdzielne liniowo, dzięki czemu można je wykorzystać do uczenia algorytmów klasyfikujących.

Nie istnieje, niestety, uniwersalna wartość parametru  $\gamma$ , która byłaby skuteczna dla różnych zestawów danych. Znalezienie odpowiedniej wartości tego parametru dla określonego zbioru danych wymaga eksperymentowania. W rozdziale 6., „Najlepsze metody oceny modelu i strojenie parametryczne”, przybliżę Ci techniki pozwalające na zautomatyzowanie procesów optymalizacji takich parametrów strojeniowych. Na razie będziemy korzystać z wyliczonej przez mnie wartości parametru  $\gamma$ , która daje **dobre** wyniki.

## Przykład 2. Rozdzielenie koncentrycznych kręgów

W poprzednim ustępie ukazałem sposób rozdzielania klas tworzących na wykresie sierpowate kształty za pomocą jądrowej analizy PCA. Wkładamy dużo wysiłku, żeby zrozumieć działanie tej metody, dlatego przyjrzyjmy się kolejnemu ciekawemu przykładowi nieliniowego problemu: koncentrycznym kręgom.



Rysunek 5.14. Wykresy rzutowania próbek na podprzestrzeń wygenerowaną za pomocą jądrowej analizy głównych składowych

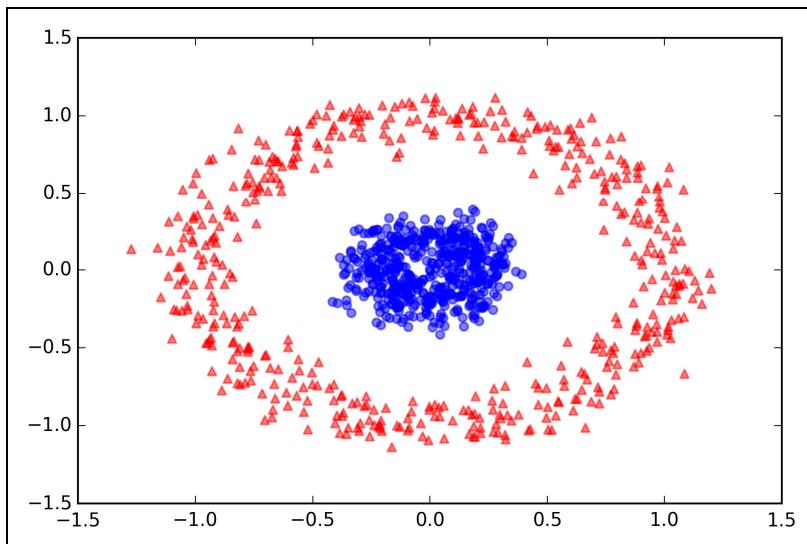
Poniżej prezentuję odpowiedni fragment kodu:

```
>>> from sklearn.datasets import make_circles
>>> X, y = make_circles(n_samples=1000,
...                      random_state=123, noise=0.1, factor=0.2)
>>> plt.scatter(X[y==0, 0], X[y==0, 1],
...               color='red', marker='^', alpha=0.5)
>>> plt.scatter(X[y==1, 0], X[y==1, 1],
...               color='blue', marker='o', alpha=0.5)
>>> plt.show()
```

Również tutaj zakładamy dwuklasowy problem, w którym trójkąty reprezentują próbki przynależne do jednej klasy, a kółka — dane z innej klasy (rysunek 5.15).

Stwórzmy najpierw wykres rzutowania danych na podprzestrzeń utworzoną w wyniku standardowej analizy głównych składowych, którą następnie porównamy z rezultatami jądrowej analizy PCA:

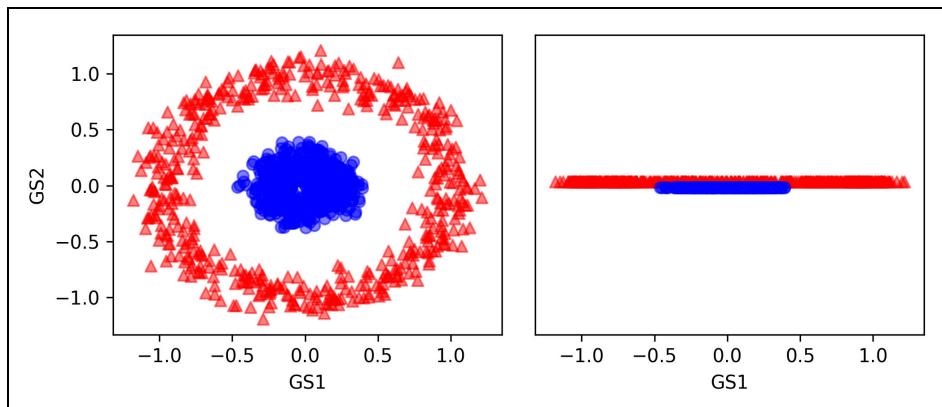
```
>>> scikit_pca = PCA(n_components=2)
>>> X_spca = scikit_pca.fit_transform(X)
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_spca[y==0, 0], X_spca[y==0, 1],
...                 color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_spca[y==1, 0], X_spca[y==1, 1],
...                 color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_spca[y==0, 0], np.zeros((500,1))+0.02,
...                 color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_spca[y==1, 0], np.zeros((500,1))-0.02,
...                 color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('GS1')
>>> ax[0].set_ylabel('GS2')
>>> ax[1].set_ylim([-1, 1])
```



Rysunek 5.15. Przykład nieliniowo rozdzielnych klas tworzących na wykresie koncentryczne kręgi

```
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('GS1')
>>> plt.show()
```

Jak widać na rysunku 5.16, standardowa analiza PCA nie jest w stanie uzyskać rezultatów nadającecych się do liniowej klasyfikacji.

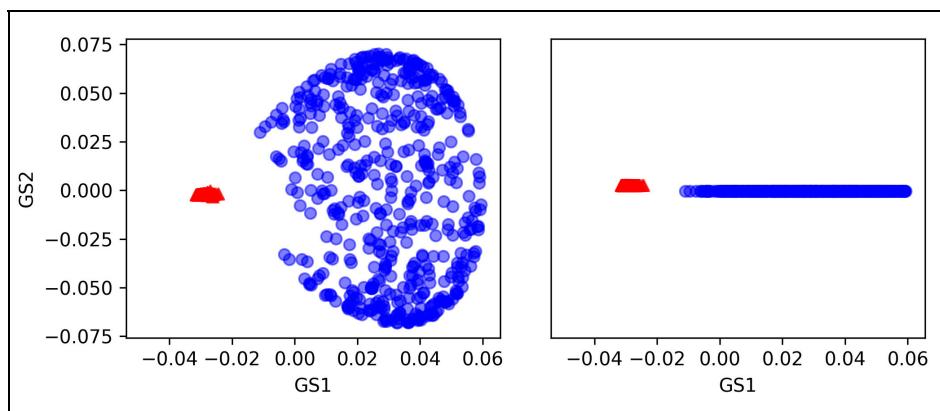


Rysunek 5.16. Próbki rzutowane na podprzestrzeń wygenerowaną poprzez standardową analizę PCA

Zobaczmy, czy będziemy mieli więcej szczęścia po zastosowaniu jądrowej analizy PCA przy odpowiednio dobranym parametrze  $\gamma$ :

```
>>> X_kpca = rbf_kernel_pca(X, gamma=15, n_components=2)
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_kpca[y==0, 0], X_kpca[y==0, 1],
...                 color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_kpca[y==1, 0], X_kpca[y==1, 1],
...                 color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==0, 0], np.zeros((500,1))+0.02,
...                 color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==1, 0], np.zeros((500,1))-0.02,
...                 color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('GS1')
>>> ax[0].set_ylabel('GS2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('GS1')
>>> plt.show()
```

Również w tym przypadku jądrowa analiza PCA rzutowała dane na nową podprzestrzeń, w której obydwie klasy stały się liniowo rozdzielne (rysunek 5.17).



Rysunek 5.17. Wynik rzutowania próbek na podprzestrzeń wygenerowaną poprzez jądrową analizę PCA

## Rzutowanie nowych punktów danych

W omówionych powyżej przykładach stosowania jądrowej analizy PCA rzutowaliśmy pojedynczy zestaw danych na nową podprzestrzeń cech. Jednak w prawdziwym świecie możemy mieć do czynienia z większą liczbą zbiorów danych, np. z danymi uczącymi i testowymi, a zazwyczaj także z nowymi próbками pojawiającymi się już po stworzeniu i ocenie modelu. W nimiejszym ustępie nauczymy się rzutować punkty danych nienależące do zestawu danych uczących.

Jak pamiętamy z opisu standardowej analizy PCA, rzutujemy dane poprzez obliczenie iloczynu skalarnego pomiędzy macierzą transformacji a wprowadzonymi próbками; kolumny macierzy

rzutowania to  $k$  największych wektorów własnych ( $v$ ) uzyskanych za pomocą macierzy kowariancji. Pojawia się następujące pytanie: w jaki sposób możemy przenieść tę koncepcję na jądrową analizę PCA? Jeżeli zastanowimy się nad mechanizmem działania jądrowej analizy głównych składowych, przypomnimy sobie, że otrzymywaliśmy wektor własny ( $a$ ) z wycentrowanej macierzy jądra (nie z macierzy kowariancji), co oznacza, że analizowane próbki są od razu rzutowane na oś głównej składowej  $v$ . Zatem jeśli chcemy rzutować nową próbkę  $x'$  na tę oś głównej składowej, musimy przeprowadzić następujące obliczenia:

$$\phi(x')^T v$$

Na szczęście możemy wykorzystać sztuczkę z funkcją jądra, dzięki czemu nie będziemy musieli jawnie wycalczać rzutowania  $\phi(x')^T v$ . Warto jednak zauważyć, że jądrowana analiza PCA w przeciwieństwie do standardowej analizy głównych składowych jest metodą pamięciową, co oznacza, że musimy wykorzystywać pierwotny zestaw uczący podczas każdorazowego rzutowania nowych próbek. Musimy obliczać jądro (podobieństwo) RBF pomiędzy  $i$ -tą próbką zbioru danych uczących a nową próbką  $x'$ :

$$\begin{aligned}\phi(x')^T v &= \sum_i a^{(i)} \phi(x')^T \phi(x^{(i)}) = \\ &= \sum_i a^{(i)} k(x', x^{(i)})^T\end{aligned}$$

Tutaj wektory własne  $a$  i wartości własne  $\lambda$  macierzy jądra  $K$  spełniają następujący warunek:

$$Ka = \lambda a$$

Po wyliczeniu podobieństwa pomiędzy nowymi próbками a danymi z zestawu uczącego musimy znormalizować wektor własny  $a$  za pomocą jego wartości własne. Zmodyfikujmy więc uprzednio zaimplementowaną funkcję `rbf_kernel_pca` w taki sposób, aby zwracane były również wartości macierzy jądra:

```
from scipy.spatial.distance import pdist, squareform
from scipy import exp
from scipy.linalg import eigh
import numpy as np

def rbf_kernel_pca(X, gamma, n_components):
    """
    Implementacja jądra RBF w analizie PCA.

    Parametry
    -----
    X: {typ ndarray biblioteki NumPy}, wymiary = [n_próbek, n_cech]

    gamma: liczby zmiennoprzecinkowe
    Parametr strojenia jądra RBF
    
```

`n_components`: liczby całkowite  
*Liczba zwracanych głównych składowych*

*Zwroca*

`X_pc`: {typ ndarray biblioteki NumPy}, wymiary = [n\_próbek, k\_cech]

*Rzutowany zestaw danych*

`lambdas`: lista  
*Wartości własne*

.....

*# oblicza kwadraty odległości euklidesowych par*

*# w zestawie danych o rozmiarze MxN*

`sq_dists = pdist(X, 'squared_euclidean')`

*# przekształca wyliczone odległości na macierz kwadratową*  
`mat_sq_dists = squareform(sq_dists)`

*# oblicza symetryczną macierz jądra*  
`K = exp(-gamma * mat_sq_dists)`

*# centruje macierz jądra*  
`N = K.shape[0]`  
`one_n = np.ones((N,N)) / N`  
`K = K - one_n.dot(K) - K.dot(one_n) + one_n.dot(K).dot(one_n)`

*# wydobywa pary własne z centrowanej macierzy jądra*  
*# funkcja numpy.eigh zwraca je w malejącej kolejności*  
`eigvals, eigvecs = eigh(K)`

*# wybiera k największych wektorów własnych (rzutowanych próbek)*  
`alphas = np.column_stack((eigvecs[:, :-i]`  
`for i in range(1, n_components+1)))`

*# wybiera odpowiednie wartości własne*  
`lambdas = [eigvals[-i] for i in range(1, n_components+1)]`

`return alphas, lambdas`

Stwórzmy teraz nowy zestaw danych widoczny na wykresie jako sierpowate kształty i rzutujmy go na jednowymiarową podprzestrzeń za pomocą zaktualizowanej implementacji jądrowej analizy PCA:

```
>>> X, y = make_moons(n_samples=100, random_state=123)
>>> alphas, lambdas = rbf_kernel_pca(X, gamma=15, n_components=1)
```

Aby upewnić się, że nasz kod rzeczywiście rzutuje nowe dane, założymy, że 26. punkt w nowo utworzonym zbiorze danych jest nową próbką  $x'$ ; sprawdźmy teraz, czy ten punkt będzie rzutowany na podprzestrzeń cech:

```
>>> x_new = X[25]
>>> x_new
array([ 1.8713187 ,  0.00928245])
>>> x_proj = alphas[25] # oryginalne rzutowanie
>>> x_proj
array([ 0.07877284])
>>> def project_x(x_new, X, gamma, alphas, lambdas):
...     pair_dist = np.array([np.sum(
...         (x_new-row)**2) for row in X])
...     k = np.exp(-gamma * pair_dist)
... return k.dot(alphas / lambdas)
```

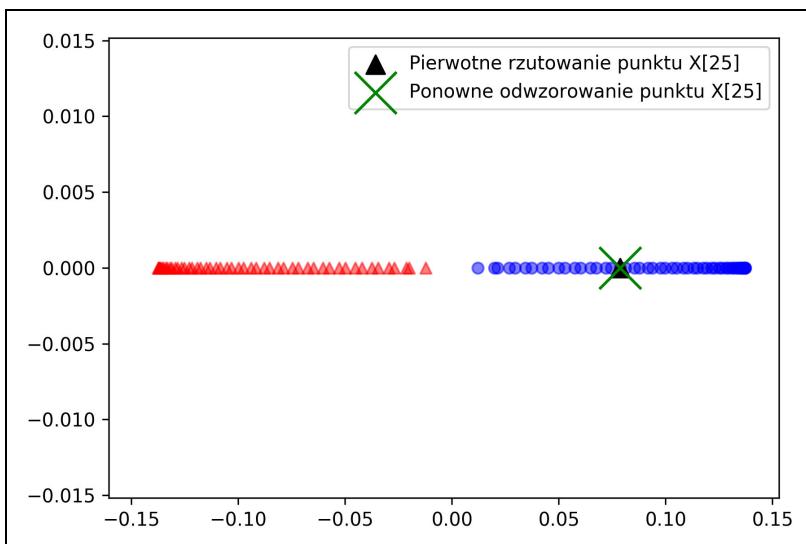
Z pomocą poniższego fragmentu kodu będziemy w stanie odtworzyć pierwotne rzutowanie. Dzięki funkcji `project_x` możemy rzutować również wszelkie nowe próbki danych. Omawiany fragment wygląda następująco:

```
>>> x_reproj = project_x(x_new, X,
...                         gamma=15, alphas=alphas, lambdas=lambdas)
>>> x_reproj
array([ 0.07877284])
```

Na koniec zwizualizujmy to rzutowanie wobec pierwszej głównej składowej:

```
>>> plt.scatter(alphas[y==0, 0], np.zeros((50)),
...               color='red', marker='^', alpha=0.5)
>>> plt.scatter(alphas[y==1, 0], np.zeros((50)),
...               color='blue', marker='o', alpha=0.5)
>>> plt.scatter(x_proj, 0, color='black',
...               label='Pierwotne rzutowanie punktu X[25]',
...               marker='^', s=100)
>>> plt.scatter(x_reproj, 0, color='green',
...               label='Ponowne odwzorowanie punktu X[25]',
...               marker='x', s=500)
>>> plt.legend(scatterpoints=1)
>>> plt.show()
```

Widzimy na rysunku 5.18, że próbka  $x'$  została poprawnie odwzorowana wobec pierwszej głównej składowej.



Rysunek 5.18. Odwzorowanie nowego punktu danych na podprzestrzeni pierwszej głównej składowej

## Algorytm jądrowej analizy głównych składowych w bibliotece scikit-learn

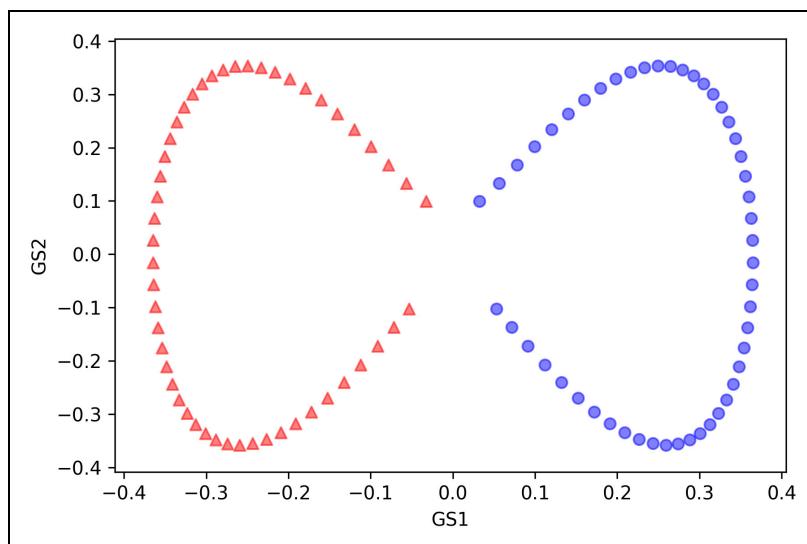
Dla naszej wygody możemy korzystać z implementacji jądrowej analizy głównych składowych zawartej w podmoduле `sklearn.decomposition`. Sposób jej używania przypomina standardową klasę PCA, a funkcję jądra możemy zdefiniować za pomocą parametru `kernel`:

```
>>> from sklearn.decomposition import KernelPCA
>>> X, y = make_moons(n_samples=100, random_state=123)
>>> scikit_kpca = KernelPCA(n_components=2,
...                           kernel='rbf', gamma=15)
>>> X_skernpca = scikit_kpca.fit_transform(X)
```

Aby się przekonać, czy uzyskujemy wyniki zgodne z naszą samodzielnią implementacją jądrowej analizy PCA, stwórzmy wykres przekształconych danych „sierpowatych” rzutowanych na podprzestrzeń dwóch pierwszych głównych składowych:

```
>>> plt.scatter(X_skernpca[y==0, 0], X_skernpca[y==0, 1],
...               color='red', marker='^', alpha=0.5)
>>> plt.scatter(X_skernpca[y==1, 0], X_skernpca[y==1, 1],
...               color='blue', marker='o', alpha=0.5)
>>> plt.xlabel('GS1')
>>> plt.ylabel('GS2')
>>> plt.show()
```

Wykres ukazany na rysunku 5.19 dowodzi, że klasa KernelPCA jest zgodna z naszą własną implementacją.



Rysunek 5.19. Wykres danych rzutowanych na przestrzeń o zredukowanej liczbie wymiarów wygenerowanej za pomocą klasy KernelPCA biblioteki scikit-learn

Interfejs scikit-learn zawiera również zaawansowane techniki nieliniowej redukcji wymiarowości, których opis wykracza jednak poza ramy niniejszej książki. Wyczerpujące podsumowanie bieżących algorytmów redukowania wymiarowości w bibliotece scikit-learn wraz z graficznymi przykładami znajdziesz pod adresem <http://scikit-learn.org/stable/modules/manifold.html>.

## Podsumowanie

W tym rozdziale poznaliśmy trzy różne podstawowe techniki redukowania wymiarowości pozwalające na odkrywanie cech: standardową analizę PCA, analizę LDA oraz jądrową analizę głównych składowych. Dzięki standardowej analizie głównych składowych rzutowaliśmy dane na podprzestrzeń liczącą sobie mniejszą liczbę wymiarów od pierwotnej przestrzeni cech, co pozwala nam na maksymalizowanie wariancji wzduż ortogonalnych osi cech przy jednoczesnym ignorowaniu etykiet klas. Liniowa analiza dyskryminacyjna jest, w przeciwieństwie do analizy PCA, techniką nadzorowaną redukcji wymiarowości, co oznacza, że są w niej wykorzystywane zawarte w danych treningowych informacje o klasach do maksymalizowania rozdzielcości klas w liniowej przestrzeni cech. Rozdział zakończyliśmy opisem nieliniowego ekstraktora cech — jądrowej analizy PCA. Dzięki sztuczce z funkcją jądra i tymczasowemu rzutowaniu próbek na

przestrzeń o większej liczbie wymiarów mogliśmy ostatecznie skompresować zbiory danych zawierające nieliniowe cechy do mniejszej podprzestrzeni, w której poszczególne klasy stawały się liniowo rozdzielne.

Po zapoznaniu się z tymi niezbędnymi technikami wstępnego przetwarzania danych jesteśmy już gotowi do nauki ich skutecznego wdrażania oraz oceniania skuteczności różnych modeli, czym zajmiemy się w następnym rozdziale.

# Najlepsze metody oceny modelu i strojenie parametryczne

W poprzednich rozdziałach omówiłem najważniejsze algorytmy uczenia maszynowego służące do klasyfikacji próbek oraz sposoby przekształcania danych przed ich wczytaniem do modelu. Nadszedł czas, aby poznać najlepsze metody tworzenia skutecznych modeli uczenia maszynowego poprzez dostrajanie algorytmów i ocenę skuteczności modelu! W niniejszym rozdziale poruszymy następujące tematy:

- uzyskiwanie uczciwych oszacowań skuteczności modelu,
- diagnozowanie najczęstszych problemów trapiących algorytmy uczenia maszynowego,
- strojenie modeli uczenia maszynowego,
- ocenianie modeli predykejnych za pomocą różnych metryk skuteczności.

## Usprawnianie cyklu pracy za pomocą kolejkowania

W czasie stosowania różnych technik wstępniego przetwarzania danych, takich jak **standardyzacja** w celu skalowania cech (rozdział 4., „Tworzenie dobrych zbiorów uczących — wstępne przetwarzanie danych”) czy **analiza głównych składowych** pozwalająca na kompresję danych (rozdział 5., „Kompresja danych poprzez redukcję wymiarowości”), dowiedzieliśmy się, że musimy wielokrotnie korzystać z parametrów uzyskanych podczas dopasowywania i uczenia modelu

po to, aby wstępnie przygotować wszelkie nowe dane, np. próbki należące do oddzielnego zestawu danych testowych. W tym podrozdziale nauczymy się używać niezwykle przydatnego narzędzia, klasy `Pipeline` stanowiącej część biblioteki scikit-learn. Umożliwia nam ona konfigurowanie modelu np. poprzez dobór dowolnej liczby etapów transformacji danych, dzięki czemu jeszcze łatwiej będą przeprowadzane analizy nowych danych.

## Wczytanie zestawu danych Breast Cancer Wisconsin

W tym rozdziale będziemy pracować na zestawie danych **Breast Cancer Wisconsin**, zawierającym 569 próbek **łagodnych** i **złośliwych** komórek nowotworowych. Pierwsze dwie kolumny tego zbioru danych to unikatowe identyfikatory próbek oraz rozpoznanie ( $M = \text{Złośliwa}$ ,  $B = \text{Łagodna}$ )<sup>1</sup>. Kolumny 3 – 32 zawierają 30 cech wyliczonych na podstawie cyfrowych obrazów jądra komórkowego; można te cechy wykorzystać do stworzenia modelu przewidującego, czy dana próbka jest złośliwa lub łagodna. Podobnie jak w przypadku pozostałych zestawów danych, Breast Cancer Wisconsin jest częścią **repozytorium uczenia maszynowego UCI**, a więcej informacji na jego temat znajdziesz pod adresem [https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+\(Diagnostic\)](https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)).

Wczytamy teraz zbiór danych i rozdzielimy go na dane uczące oraz testowe:

1. Najpierw za pomocą biblioteki pandas wczytamy zestaw danych bezpośrednio ze strony UCI:

```
>>> import pandas as pd
>>> df = pd.read_csv('https://archive.ics.uci.edu/ml/machinelearning-
databases/breast-cancer-wisconsin/wdbc.data',
header=None)
```

2. Przydzielimy teraz 30 cech do tabeli `x` utworzonej za pomocą biblioteki NumPy. Dzięki klasie `LabelEncoder` przekształcimy etykiety klas z łańcuchów znaków ( $M$  i  $B$ ) w liczby całkowite:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> X = df.loc[:, 2:].values
>>> y = df.loc[:, 1].values
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
```

Po zakodowaniu etykiet klas (diagnoz) w tabeli `y` złośliwe guzy przynależą do klasy 1, a nowotwory łagodne są reprezentowane jako klasa 0. Możemy to sprawdzić, wywołując metodę `transform` klasy `LabelEncoder` wobec dwóch sztucznych etykiet:

```
>>> le.transform(['M', 'B'])
array([1, 0])
```

3. Zanim zajmiemy się kolejkowaniem czynności w naszym modelu, rozdzielimy zestaw danych na 80% próbek uczących i 20% próbek testowych:

<sup>1</sup> Od „malignant” i „benign” — przyp. tłum.

```
>>> from distutils.version import LooseVersion as Version
>>> from sklearn import __version__ as sklearn_version

>>> if Version(sklearn_version) < '0.18':
>>>     from sklearn.cross_validation import train_test_split
>>> else:
>>>     from sklearn.model_selection import train_test_split

>>> X_train, X_test, y_train, y_test = \
...     train_test_split(X, y, test_size=0.20, random_state=1)
```

## Łączenie funkcji transformujących i estymatorów w kolejce czynności

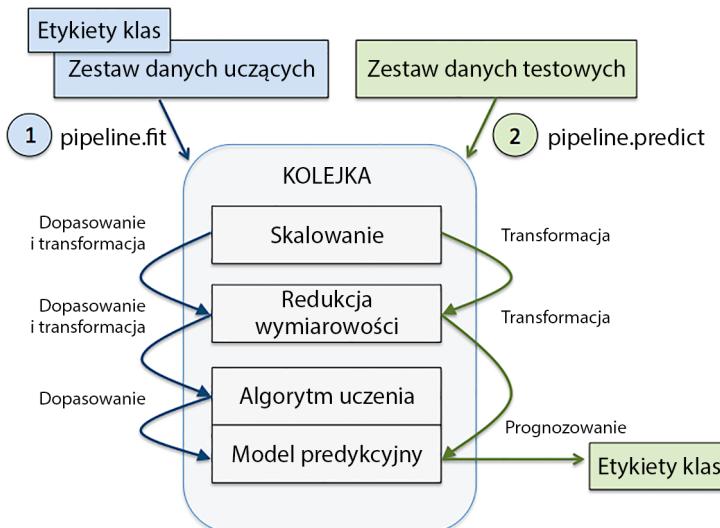
Z poprzedniego rozdziału dowiedzieliśmy się, że w wielu algorytmach uczenia maszynowego w celu uzyskania maksymalnej skuteczności wymagane jest skalowanie wprowadzanych cech. Dlatego musimy przeprowadzić standaryzację kolumn w zestawie Breast Cancer Wisconsin przed przesłaniem ich do klasyfikatora liniowego, np. modelu regresji logistycznej. Ponadto założymy, że chcemy skompresować nasze dane z początkowych 30 wymiarów do dwuwymiarowej podprzestrzeni za pomocą **analizy głównych składowych** (PCA), techniki odkrywania cech opisanej w rozdziale 5., „Kompresja danych poprzez redukcję wymiarowości”. Zamiast osobno dopasowywać i przekształcać dane uczące i testowe, możemy umieścić obiekty StandardScaler, PCA i LogisticRegression w kolejce:

```
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.decomposition import PCA
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.pipeline import Pipeline
>>> pipe_lr = Pipeline([('scl', StandardScaler()),
...                     ('pca', PCA(n_components=2)),
...                     ('clf', LogisticRegression(random_state=1))])
>>> pipe_lr.fit(X_train, y_train)
>>> print('Dokładność testu: %.3f' % pipe_lr.score(X_test, y_test))
Dokładność testu: 0.947
```

Danymi wejściowymi obiektu Pipeline jest lista krotek. Pierwszą wartością każdej krotki jest dowolny identyfikator (ciąg znaków), za pomocą którego uzyskujemy dostęp do poszczególnych elementów kolejki, o czym przekonamy się w dalszej części rozdziału, natomiast drugim elementem krotki jest transformator lub estymator biblioteki scikit-learn.

Etapami pośrednimi w kolejce są klasy transformujące, zamyka ją natomiast estymator. W powyższym fragmencie kodu stworzyliśmy kolejkę składającą się z dwóch pośrednich etapów: funkcji transformujących StandardScaler i PCA, a klasyfikator liniowy LogisticRegression jest ostatnią funkcją. Po uruchomieniu metody fit w kolejce pipe\_lr funkcja StandardScaler przeprowadziła operacje fit i transform na danych uczących i te przekształcone próbki zostały

przekazane kolejnemu obiektowi w kolejce — klasie PCA. Podobnie jak w przypadku funkcji StandardScaler, obiekt PCA wykonał czynności fit i transform na przeskalowanych danych uczących, po czym przekazał zmodyfikowane próbki do ostatniego elementu kolejki — estymatora LogisticRegression. Warto zauważyć, że możemy wstawiać dowolną liczbę etapów pośrednich w kolejce; nie istnieją żadne ograniczenia. Mechanizm działania kolejek został podsumowany na rysunku 6.1.



Rysunek 6.1. Ogólny mechanizm działania kolejki Pipeline

## Stosowanie k-krotnego sprawdzianu krzyżowego w ocenie skuteczności modelu

Jednym z kluczowych aspektów tworzenia modelu uczenia maszynowego jest oszacowanie jego skuteczności wobec nieznanych danych. Założymy, że wytrenowaliśmy nasz model na zestawie danych uczących i wykorzystujemy te same dane do sprawdzenia jego działania na prawdziwych próbkach. Pamiętamy z podrozdziału „Zapobieganie nadmiernemu dopasowaniu za pomocą regularyzacji” w rozdziale 3., „Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn”, że model może być niewystarczająco dopasowany (duże obciążenie) z powodu zbyt małej złożoności lub przetrenowany (duża wariancja) w wyniku zbyt wysokiego stopnia skomplikowania. Aby określić dopuszczalny kompromis pomiędzy obciążeniem a wariancją, musimy ostrożnie ocenić działanie modelu. W tym ustępie poznamy przydatne techniki sprawdzianu krzyżowego: metodę wydzielania oraz k-krotną kroswalidację, które pomagają w uzyskaniu rzetelnych oszacowań błędów uogólniania modelu, tj. skuteczności modelu wobec nieznanych danych.

## Metoda wydzielania

Klasycznym i bardzo popularnym sposobem szacowania skuteczności uogólniania modeli uczenia maszynowego jest metoda wydzielania (ang. *holdout cross-validation*). Rozdzielimy w niej początkowy zestaw danych na osobne podzbiory danych uczących i testowych — ten pierwszy służy do trenowania modelu, a drugi do szacowania jego skuteczności. Jednak w typowych zastosowaniach uczenia maszynowego interesuje nas również strojenie i porównywanie różnych ustawień parametrów w celu dalszej poprawy skuteczności w prognozowaniu wyników nieznanych próbek. Proces ten nazywamy **doborem modelu** — pojęcie to odnosi się do danego problemu klasyfikacji, dla którego chcemy znaleźć **optymalne** wartości parametrów strojenia (zwanych również **hiperparametrami**). Jeżeli jednak na etapie doboru modelu będziemy ciągle wykorzystywać ten sam zestaw danych testowych, stanie się on w końcu składową danych uczących i dlatego zwiększymy ryzyko jego przetrenowania. Pomimo tego problemu wiele osób stosuje dane testowe podczas doboru modelu, a to nie jest dobre rozwiązanie.

Skuteczniejszym sposobem wykorzystania metody wydzielania w doborze modelu jest podzielenie danych na trzy zestawy: zbiór uczący, zbiór walidacyjny i zbiór testowy. Zestaw danych uczących służy do trenowania różnych modeli, których skuteczność jest następnie sprawdzana na próbkach walidacyjnych. Zaletą wyznaczania zestawu testowego z próbek nieznanych modelowi na etapie uczenia i doboru modelu jest uzyskanie bardziej neutralnego oszacowania zdolności uogólniania klasyfikacji. Na rysunku 6.2 widzimy ogólny schemat metody wydzielania, w której zestaw danych walidacyjnych jest ciągle wykorzystywany do oceny skuteczności wyuczonego modelu przy użyciu różnych wartości hiperparametrów. Gdy już jesteśmy zadowoleni z dostrojenia wartości parametrów, możemy przejść do oszacowania skuteczności modelu wobec danych testowych.



Rysunek 6.2. Schemat metody wydzielania

Wadą metody wydzielania jest czułość oszacowania skuteczności na sposób podziału danych uczących na podzbiory uczące i walidujące; oszacowanie będzie się różniło dla różnych próbek. W kolejnym ustępie zapoznamy się z bardziej wszechstronną techniką szacowania skuteczności — k-krotnego sprawdzianu krzyżowego — gdzie powtarzamy  $k$ -krotnie proces wydzielania wobec  $k$  podzbiorów danych uczących.

## K-krotny sprawdzian krzyżowy

W k-krotnym sprawdzianie krzyżowym (ang.  *$k$ -fold cross-validation*) losowo rozdzielimy zestaw danych uczących na  $k$  niezastępowanych podzbiorów, gdzie  $k-1$  podzbiorów jest wykorzystywanych do uczenia modelu, a tylko jeden do jego testowania. Proces ten jest powtarzany  $k$ -krotnie, dzięki czemu otrzymujemy  $k$  modeli i oszacowań skuteczności.

Jeżeli nie znasz pojęć losowania ze zwracaniem oraz bez zwracania, przeprowadźmy prosty eksperyment myślowy. Założymy, że gramy w jakąś grę losową, w której dobieramy liczby z urny. Zaczynamy z urną przechowującą pięć unikatowych cyfr: 0, 1, 2, 3 i 4, a w każdej turze losujemy tylko jedną z nich. W pierwszej rundzie prawdopodobieństwo wyciągnięcia określonej liczby z urny wynosi  $\frac{1}{5}$ . W przypadku próbowania bez zwracania nie wrzucamy z powrotem tego numeru do urny po zakończeniu tury. W wyniku tego prawdopodobieństwo wylosowania danej cyfry z puli pozostałych cyfr w następnej rundzie ściśle zależy od wyniku poprzedniej tury. Jeżeli np. w urnie pozostał zbiór liczb: 0, 1, 2 i 4, prawdopodobieństwo wyciągnięcia w kolejnej kolejce liczby 0 zwiększa się do  $\frac{1}{4}$ .

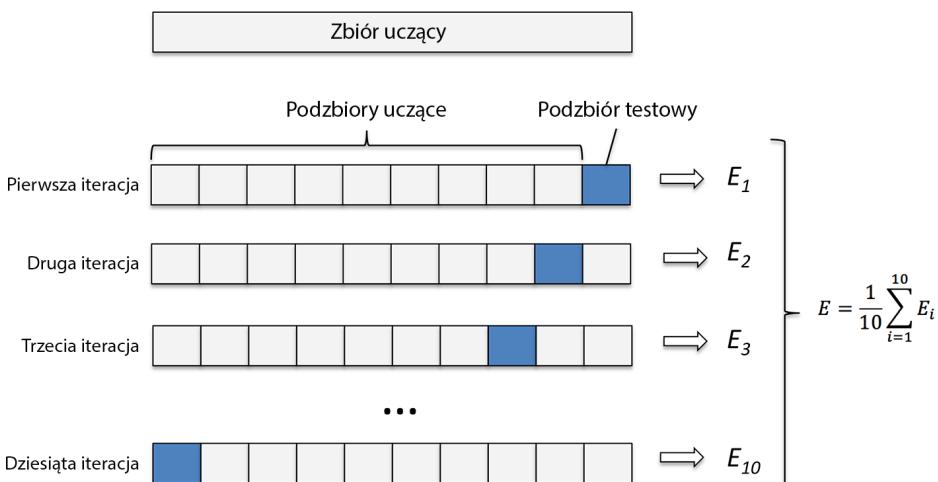
Jednak w przypadku próbowania ze zwracaniem za każdym razem wrzucamy wylosowaną liczbę z powrotem do urny, przez co prawdopodobieństwo wyciągnięcia określonej liczby w kolejnej turze pozostaje niezmienne; istnieje możliwość, że znowu wyciągniemy tę samą cyfrę. Innymi słowy, w próbowaniu ze zwracaniem próbki (liczby) są od siebie wzajemnie niezależne i mają zerową kowariancję. Wyniki losowania pięciu liczb mogą wyglądać następująco:

- losowe próbowanie bez zwracania: 2, 1, 3, 4, 0;
- losowe próbowanie ze zwracaniem: 1, 3, 3, 4, 1.

Obliczamy następnie średnią skuteczność modeli na podstawie różnych, niezależnych od siebie podzbiorów, przez co w konsekwencji uzyskujemy oszacowanie skuteczności, które jest mniej wrażliwe na podział danych od metody wydzielania. Zazwyczaj wykorzystujemy k-krotny sprawdzian krzyżowy do strojenia modelu, tj. wyszukiwania optymalnych wartości hiperparametrów, gwarantujących zadowalającą wydajność uogólniania. Po określeniu tych hiperparametrów ponownie uczymy model, tym razem na całym zestawie danych uczących, i otrzymujemy ostateczne oszacowanie skuteczności przy użyciu niezależnego zestawu testowego.

K-krotny sprawdzian krzyżowy stanowi technikę próbowania bez zwracania, dlatego każda próbka będzie występowała tylko raz w zestawie uczącym lub testowym, dzięki czemu oszacowanie skuteczności modelu jest obarczone mniejszą wariancją niż w metodzie wydzielania. Rysunek 6.3 przedstawia podsumowanie mechanizmu k-krotnego sprawdzianu krzyżowego dla  $k = 10$ . Zestaw danych uczących został podzielony na 10 podzbiorów i w trakcie 10. iteracji

9 podzbiorów zostanie wykorzystanych w procesie uczenia, a 1 będzie stanowił zbiór testowy. Do tego oszacowanie skuteczności  $E_i$  (np. dokładność lub błąd klasyfikacji) każdego podzbioru służy następnie do obliczenia średniej oszacowanej skuteczności  $E$  modelu.



Rysunek 6.3. Schemat k-krotnego sprawdzianu krzyżowego dla 10 wydzielonych podzbiorów

Standardowa wartość parametru  $k$  wynosi 10, co w większości zastosowań stanowi rozsądny wybór. Jeśli jednak pracujesz na względnie małych zestawach danych, warto zwiększyć liczbę podzbiorów. Po zwiększeniu wartości parametru  $k$  więcej danych uczących będzie wykorzystywanych w każdej iteracji, przez co zmniejszymy obciążenie modelu w czasie szacowania wydajności uogólniania. Duże wartości parametru  $k$  wydłużają jednak czas działania algorytmu k-krotnego sprawdzianu krzyżowego i obarczają oszacowania większą wariancją, gdyż podzbiory uczące będą do siebie bardziej podobne. Z drugiej strony podczas pracy z dużymi zbiorami danych możemy wprowadzić mniejszą wartość parametru  $k$  (np.  $k = 5$ ) i ciągle otrzymywać dokładne oszacowanie średniej skuteczności modelu przy jednoczesnym zmniejszeniu pochłaniającej mocy obliczeniowej przeznaczonej na dopasowywanie i ocenę modelu wobec poszczególnych podzbiorów.

Specjalnym przypadkiem k-krotnego sprawdzianu krzyżowego jest metoda **minus jednego elementu** (ang. *leave-one-out method* — LOO). W sprawdzaniu LOO wprowadzamy taką samą liczbę podzbiorów jak próbek uczących ( $k = n$ ), więc w każdej iteracji tylko jedna próbka służy do testowania modelu. Jest to rozwiązanie zalecane podczas pracy z bardzo małymi zbiorami danych.

Nieznaczna modyfikacją standardowej k-krotnej kroswalidacji jest warstwowy (stratyfikowany) k-krotny sprawdzian krzyżowy, który lepiej sobie radzi z oszacowaniem obciążenia i wariancji, zwłaszcza w przypadkach nierównomiernych proporcji pomiędzy klasami, co zostało udowodnione przez Rona Kohaviego i in. (R. Kohavi i in., *A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection* [w:] *Ijcai*, t. 14, 1995, s. 1137 – 1145). W stratyfikowanej

walidacji krzyżowej proporcje pomiędzy klasami zostają zachowane w każdym podzbiorze, dzięki czemu poszczególne podzbiory reprezentują proporcje klas w zestawie uczącym, co pokażemy przy użyciu obiektu `StratifiedKFold` (stanowiącego część biblioteki scikit-learn):

```
>>> import numpy as np
>>> if Version(sklearn_version) < '0.18':
>>>     from sklearn.cross_validation import StratifiedKFold
>>> else:
>>>     from sklearn.model_selection import StratifiedKFold
>>> if Version(sklearn_version) < '0.18':
>>>     kfold = StratifiedKFold(y=y_train,
...                           n_folds=10,
...                           random_state=1)
>>> else:
>>>     kfold = StratifiedKFold(n_splits=10,
...                           random_state=1).split(X_train, y_train)
>>> scores = []
>>> for k, (train, test) in enumerate(kfold):
...     pipe_lr.fit(X_train[train], y_train[train])
...     score = pipe_lr.score(X_train[test], y_train[test])
...     scores.append(score)
...     print('Podzbiór: %s, Rozkład klasy: %s, Dokładność: %.3f' % (k+1,
...           np.bincount(y_train[train]), score))
Podzbiór: 1, Rozkład klasy: [256 153], Dokładność: 0.891
Podzbiór: 2, Rozkład klasy: [256 153], Dokładność: 0.978
Podzbiór: 3, Rozkład klasy: [256 153], Dokładność: 0.978
Podzbiór: 4, Rozkład klasy: [256 153], Dokładność: 0.913
Podzbiór: 5, Rozkład klasy: [256 153], Dokładność: 0.935
Podzbiór: 6, Rozkład klasy: [257 153], Dokładność: 0.978
Podzbiór: 7, Rozkład klasy: [257 153], Dokładność: 0.933
Podzbiór: 8, Rozkład klasy: [257 153], Dokładność: 0.956
Podzbiór: 9, Rozkład klasy: [257 153], Dokładność: 0.978
Podzbiór: 10, Rozkład klasy.: [257 153], Dokładność: 0.956
>>> print('Dokładność sprawdzianu: %.3f +/- %.3f' %
...           np.mean(scores), np.std(scores)))
Dokładność sprawdzianu: 0.950 +/- 0.029
```

Najpierw inicjujemy iterator `StratifiedKFold` z modułu `sklearn.model_selection`; etykiety klas zestawu uczącego wstawiamy do obiektu `y_train`, a liczbę podzbiorów definiujemy za pomocą parametru `n_folds`. Podczas wprowadzenia iteratora `kfold` odpowiedzialnego za pętle przebiegów po  $k$  podzbiorów wykorzystaliśmy indeksy zwrócone w obiekcie `train` w celu dopasowania kolejki utworzonej na początku rozdziału. Dzięki kolejce `pipe_lr` upewniliśmy się, że w każdej iteracji próbki są właściwie skalowane (w tym przypadku standaryzowane). Następnie użyliśmy indeksów `test` do obliczenia dokładności modelu, a jej wyniki umieściliśmy na liście `scores`, co pozwoliło nam na wyliczenie średniej dokładności i odchylenia standardowego oszacowania.

Powyższy kod bardzo ładnie ilustruje działanie k-krotnego sprawdzianu krzyżowego, ale biblioteka scikit-learn zawiera także funkcję oceniającą, pozwalającą na skuteczniejszą ewaluację naszego modelu przy użyciu warstwowej krosvalidacji:

```
>>> if Version(sklearn_version) < '0.18':
>>>     from sklearn.cross_validation import cross_val_score
>>> else:
>>>     from sklearn.model_selection import cross_val_score
>>> scores = cross_val_score(estimator=pipe_lr,
...                           X=X_train,
...                           y=y_train,
...                           cv=10,
...                           n_jobs=1)
>>> print('Wyniki dokładności sprawdzianu: %s' % scores)
Wyniki dokładności sprawdzianu: [ 0.89130435  0.97826087  0.97826087
0.91304348  0.93478261  0.97777778
0.93333333  0.95555556  0.97777778
0.95555556]
>>> print('Dokładność sprawdzianu: %.3f +/- %.3f' % (np.mean(scores),
...           np.std(scores)))
Dokładność sprawdzianu: 0.950 +/- 0.029
```

Niezwykłe przydatną cechą funkcji `cross_val_score` jest możliwość rozdzielenia procesu oceny poszczególnych podzbiorów pomiędzy kilka procesorów (lub rdzeni procesora). Jeżeli wyznaczymy wartość 1 parametru `n_jobs`, tylko jeden procesor będzie obliczał skuteczność modelu (jak to miało miejsce w poprzednim przykładzie). Jednakże po zmianie wartości tego parametru na 2 możemy rozdzielić 10 podzbiorów sprawdzianu krzyżowego pomiędzy 2 procesory (jeżeli komputer jest wyposażony w taką ich liczbę), z kolei dzięki parametrowi `n_jobs=-1` zaprzegamy do pracy wszystkie procesory znajdujące się w komputerze.

Szczegółowy opis szacowania wariancji dla uogólnionej skuteczności w sprawdzianie krzyżowym wykracza poza zakres niniejszej książki, jednak więcej informacji na ten temat znajdziesz w znakomitym artykule autorstwa Marianne Markatou i in. (M. Markatou, H. Tian, S. Biswas i G.M. Hripcak, *Analysis of Variance of Cross-validation Estimators of the Generalization Error*, „Journal of Machine Learning Research” 2005, nr 6, s. 1127 – 1168).

Möżesz również zapoznać się z innymi technikami krosvalidacyjnymi, np. ze sprawdzianem krzyżowym z wykorzystaniem reguły .632 (ang. *.632 bootstrap cross-validation*, B. Efron i R. Tibshirani, *Improvements on Cross-validation: The 632+ Bootstrap Method*, „Journal of the American Statistical Association” 1997, nr 92 (438), s. 548 – 560).

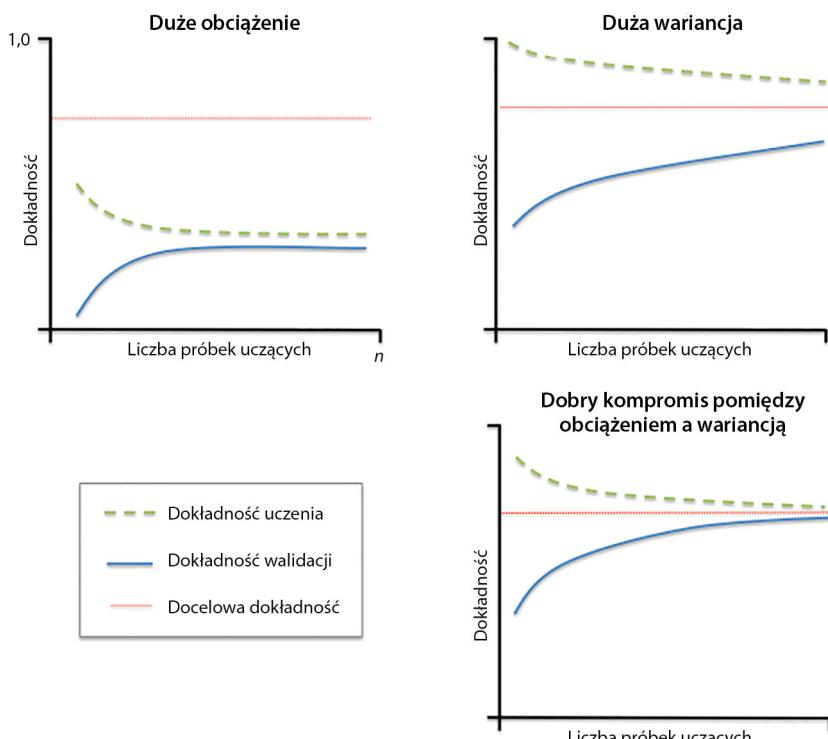
# Sprawdzanie algorytmów za pomocą krzywych uczenia i krzywych walidacji

W tym podrozdziale przyjrzymy się dwóm prostym, ale bardzo potężnym narzędziom diagnostycznym, które służą do usprawniania skuteczności algorytmu uczącego: **krzywym uczeniu i krzywym walidacji**. Dowiemy się, w jaki sposób wykorzystywać krzywe uczenia do diagnostowania algorytmu uczenia mającego problem z przetrenowaniem (dużą wariancją) lub niedostatecznym dopasowaniem (dużym obciążeniem). Omówimy ponadto krzywe walidacji pozwalające na rozwiązywanie wielu popularnych usterek algorytmów uczenia maszynowego.

## Diagnozowanie problemów z obciążeniem i wariancją za pomocą krzywych uczenia

Jeżeli model jest zbyt złożony dla danego zestawu danych — zawiera zbyt wiele stopni swobody lub parametrów — dochodzi często do jego przetrenowania wobec danych uczących, co objawia się niską jakością klasyfikowania nieznanych próbek. Często wystarczy powiększyć zestaw danych uczących, żeby zmniejszyć nadmierne dopasowanie modelu. W praktyce jednak nie zawsze jest to tanie lub wykonalne rozwiązanie. Utworzyszwy wykres dokładności uczenia i walidacji modelu jako funkcji rozmiaru zestawu uczącego, z łatwością możemy się przekonać, czy dany algorytm ma dużą wariancję lub obciążenie, a także czy dolożenie większej liczby próbek pomoże rozwiązać problem. Zanim jednak przejdziemy do tworzenia odpowiednich wykresów w interfejsie scikit-learn, rozważmy dwa często spotykane problemy z modelami, ukazane na rysunku 6.4.

Wykres w lewej górnej części rysunku 6.4 przedstawia model o wysokim obciążeniu. Cechującego niskie wartości zarówno dokładności uczenia, jak i walidacji, co wskazuje na niedostateczne dopasowanie do danych uczących. Najczęściej stosowanymi rozwiązaniami jest w tym przypadku zwiększenie liczby parametrów modelu (np. poprzez dodanie lub utworzenie nowych cech) oraz zmniejszenie stopnia regularyzacji (w klasyfikatorach SVM lub regresji logistycznej). Z kolei model zilustrowany na wykresie w prawej górnej części rysunku 6.4 ma za dużą wariancję, co można rozpoznać po dużej przerwie pomiędzy krzywymi dokładności uczenia i walidacji. Możemy wyeliminować takie przetrenowanie, dodając więcej próbek uczących lub redukując złożoność modelu np. poprzez zwiększenie wartości parametru regularizacji; w przypadku nieregularyzowanych modeli pomaga również zmniejszenie liczby cech poprzez ich wybór (rozdział 4., „Tworzenie dobrych zbiorów uczących — wstępne przetwarzanie danych”) lub odkrywanie (rozdział 5., „Kompresja danych poprzez redukcję wymiarowości”). Zauważmy, że wprowadzanie kolejnych danych uczących zmniejsza ryzyko przetrenowania. Jednak ten sposób nie zawsze okazuje się skuteczny, np. wtedy, gdy zbiór uczący jest bardzo zaszumiony lub model już jest w dużej mierze zoptymalizowany.



Rysunek 6.4. Graficzne przedstawienie przetrenowania, niedostatecznego dopasowania modelu oraz właściwego kompromisu pomiędzy obciążeniem a wariancją

W kolejnych ustępach nauczymy się rozwiązywać wspomniane problemy za pomocą krzywych walidacji, najpierw jednak przekonajmy się, w jaki sposób można wykorzystywać funkcję krzywej nauczania zawartą w bibliotece scikit-learn do oceny modelu:

```
>>> import matplotlib.pyplot as plt
>>> if Version('sklearn_version') < '0.18':
>>>     from sklearn.learning_curve import learning_curve
>>> else:
>>>     from sklearn.model_selection import learning_curve
>>> pipe_lr = Pipeline([
...         ('sc1', StandardScaler()),
...         ('clf', LogisticRegression(
...             penalty='l2', random_state=0)))
>>> train_sizes, train_scores, test_scores = \
...     learning_curve(estimator=pipe_lr,
...                     X=X_train,
...                     y=y_train,
...                     train_sizes=np.linspace(0.1, 1.0, 10),
...                     cv=10,
...                     n_jobs=1)
```

```

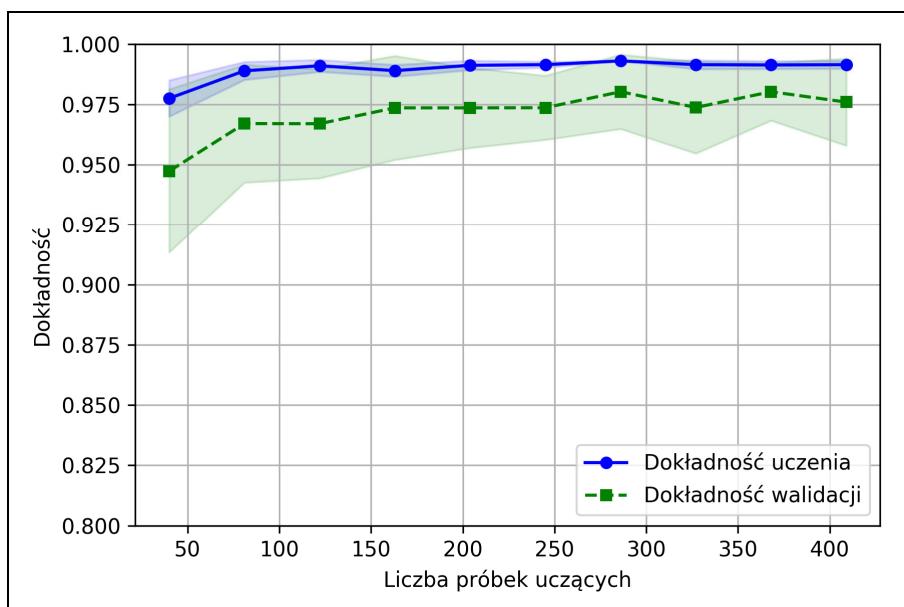
>>> train_mean = np.mean(train_scores, axis=1)
>>> train_std = np.std(train_scores, axis=1)
>>> test_mean = np.mean(test_scores, axis=1)
>>> test_std = np.std(test_scores, axis=1)
>>> plt.plot(train_sizes, train_mean,
...             color='blue', marker='o',
...             markersize=5,
...             label='Dokładność uczenia')
>>> plt.fill_between(train_sizes,
...                     train_mean + train_std,
...                     train_mean - train_std,
...                     alpha=0.15, color='blue')
>>> plt.plot(train_sizes, test_mean,
...             color='green', linestyle='--',
...             marker='s', markersize=5,
...             label='Dokładność walidacji')
>>> plt.fill_between(train_sizes,
...                     test_mean + test_std,
...                     test_mean - test_std,
...                     alpha=0.15, color='green')
>>> plt.grid()
>>> plt.xlabel('Liczba próbek uczących')
>>> plt.ylabel('Dokładność')
>>> plt.legend(loc='lower right')
>>> plt.ylim([0.8, 1.0])
>>> plt.show()

```

Po uruchomieniu powyższego kodu naszym oczom ukaże się wykres zaprezentowany na rysunku 6.5.

Poprzez parametr `train_sizes` funkcji `learning_curve` podajemy bezwzględną lub względną liczbę próbek uczących wykorzystanych do wygenerowania krzywych uczenia. W naszym przypadku określamy go następująco: `train_sizes=np.linspace(0.1, 1.0, 10)`, co oznacza, że użytych zostaje 10 równomiernie rozłożonych zestawów danych uczących. Domyślnie funkcja `learning_curve` wykorzystuje warstwowy k-krotny sprawdzian krzyżowy do obliczenia dokładności walidacji, a my poprzez parametr `cv` ustawiamy wartość  $k = 10$ . Później po prostu obliczamy uśrednione dokładności ze zwróconych wyników dla różnych rozmiarów zestawu testowego, co uwieńczyliśmy wykresem wygenerowanym za pomocą funkcji `plot`. Ponadto dodaliśmy odchylenie standardowe uśrednionych dokładności poprzez funkcję `fill_between`, co pozwoliło zaprezentować wariancję oszacowania.

Jak widać na wykresie z rysunku 6.5, nasz model sprawuje się całkiem nieźle wobec danych testowych. Jednak istnieje możliwość nieznacznego przetrenowania, na co wskazuje obecność niewielkiego, ale widocznego odstępu pomiędzy krzywymi dokładności uczenia i dokładności walidacji.



Rysunek 6.5. Wykres dokładności uczenia i walidacji dla zestawu danych Breast Cancer Wisconsin

## Rozwiązywanie problemów nadmiernego i niewystarczającego dopasowania za pomocą krzywych walidacji

Krzywe walidacji stanowią przydatne narzędzie poprawy skuteczności modelu, gdyż mogą być wykorzystane do eliminacji przetrenowania/niedostatecznego dopasowania algorytmu. Są one powiązane z krzywymi uczenia, nie tworzymy jednak wykresu dokładności uczenia i walidacji jako funkcji liczby próbek, lecz wobec wartości parametrów modelu, np. odwrotności parametru regularizacji  $C$  w przypadku regresji logistycznej. Zobaczmy, jak będzie wyglądał taki wykres:

```
>>> if Version(sklearn_version) < '0.18':
>>>     from sklearn.learning_curve import validation_curve
>>> else:
>>>     from sklearn.model_selection import validation_curve
>>> param_range = [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]
>>> train_scores, test_scores = validation_curve(
...             estimator=pipe_lr,
...             X=X_train,
...             y=y_train,
...             param_name='clf_C',
...             param_range=param_range,
...             cv=10)
>>> train_mean = np.mean(train_scores, axis=1)
```

```

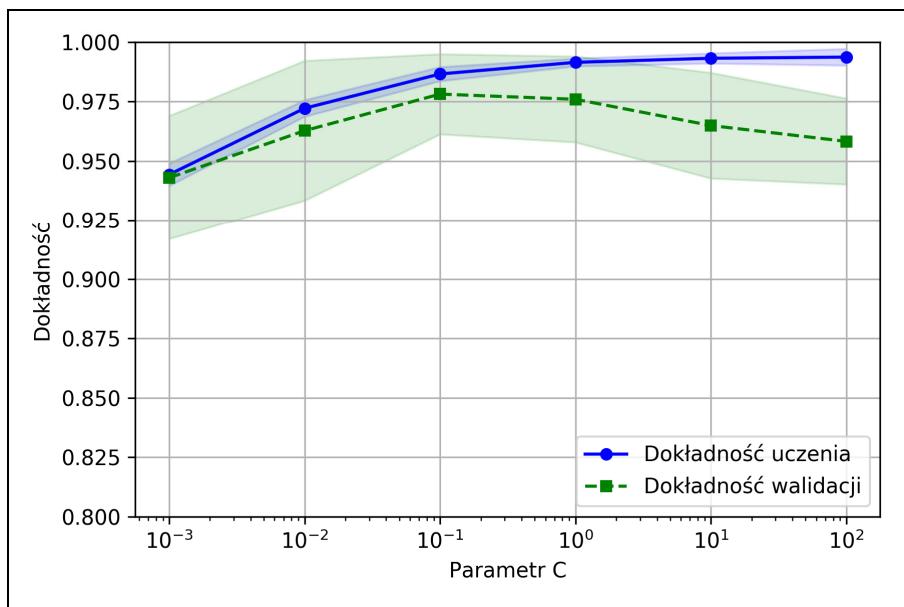
>>> train_std = np.std(train_scores, axis=1)
>>> test_mean = np.mean(test_scores, axis=1)
>>> test_std = np.std(test_scores, axis=1)
>>> plt.plot(param_range, train_mean,
...             color='blue', marker='o',
...             markersize=5,
...             label='Dokładność uczenia')
>>> plt.fill_between(param_range, train_mean + train_std,
...                     train_mean - train_std, alpha=0.15,
...                     color='blue')
>>> plt.plot(param_range, test_mean,
...             color='green', linestyle='--',
...             marker='s', markersize=5,
...             label='Dokładność walidacji')
>>> plt.fill_between(param_range,
...                     test_mean + test_std,
...                     test_mean - test_std,
...                     alpha=0.15, color='green')
>>> plt.grid()
>>> plt.xscale('log')
>>> plt.legend(loc='lower right')
>>> plt.xlabel('Parametr C')
>>> plt.ylabel('Dokładność')
>>> plt.ylim([0.8, 1.0])
>>> plt.show()

```

Z pomocą powyższego kodu otrzymaliśmy wykres krzywej walidacji dla parametru C (rysunek 6.6).

Podobnie jak w przypadku funkcji `learning_curve`, funkcja `validation_curve` domyślnie wykorzystuje warstwowy k-krotny sprawdzian krzyżowy do szacowania skuteczności modelu klasyfikacji. Wewnątrz funkcji `validation_curve` zdefiniowaliśmy oceniany parametr. W tym przykładzie wybraliśmy C — odwrotność parametru regularyzacji stosowanego w klasyfikatorze `LogisticRegression` — użyliśmy zapisu '`clf_C`', aby uzyskać dostęp do obiektu klasy `LogisticRegression` wewnątrz kolejki zadań przy określonym zakresie wartości wyznaczonym za pomocą parametru `param_range`. Analogicznie jak w przypadku omówionego w poprzednim ustępie przykładu z krzywą uczenia, wygenerowaliśmy wykres uśrednionych krzywych dokładności uczenia i walidacji wraz z ich odchyleniami standardowymi.

Pomimo że różnice w dokładności dla poszczególnych wartości parametru C są subtelne, możemy zauważyc, że naszemu modelowi zaczyna nieco brakować dopasowania z danymi przy zwiększeniu siły regularyzacji (małe wartości parametru C). Jednak duże wartości parametru C oznaczają zmniejszenie siły regularyzacji, przez co model staje się delikatnie przetrenowany. W omawianym przykładzie optymalna wartość wynosi mniej więcej C=0.1.

Rysunek 6.6. Wykres krzywej walidacji dla parametru  $C$ 

## Dostrajanie modeli uczenia maszynowego za pomocą metody przeszukiwania siatki

W domenie uczenia maszynowego występują dwa rodzaje parametrów: część parametrów jest dopasowywana za pomocą danych uczących (np. wagi w regresji logistycznej), natomiast parametry algorytmu uczenia są optymalizowane oddzielnie. Ta druga kategoria jest nazywana parametrami strojenia (lub hiperparametrami) modelu i należą do nich m.in. parametr **regularyzacji** w regresji logistycznej lub **wysokość drzewa decyzyjnego**.

W poprzednim podrozdziale użyliśmy krzywych walidacji do poprawienia skuteczności modelu poprzez dostrojenie jednego z jego parametrów strojenia. Teraz zapoznamy się z potężną techniką optymalizacji hiperparametrycznej zwaną metodą **przeszukiwania siatki** (ang. *grid search*), która powoduje zwiększenie skuteczności modelu poprzez znalezienie **optymalnej kombinacji wartości hiperparametrów**.

# Strojenie hiperparametrów przy użyciu metody przeszukiwania siatki

Koncepcja kryjąca się za metodą przeszukiwania siatki jest bardzo prosta — mamy do czynienia z prymitywną, wyczerpującą techniką wyszukiwania, w której wprowadzamy listę wartości różnych parametrów, a komputer ocenia skuteczność modelu dla każdej kombinacji tych wartości w celu otrzymania optymalnej konfiguracji:

```
>>> from sklearn.svm import SVC
>>> if Version(sklearn_version) < '0.18':
>>>     from sklearn.grid_search import GridSearchCV
>>> else:
>>>     from sklearn.model_selection import GridSearchCV
>>> pipe_svc = Pipeline([('scaler', StandardScaler()),
...                      ('clf', SVC(random_state=1))])
>>> param_range = [0.0001, 0.001, 0.01, 0.1, 1.0, 10.0, 100.0, 1000.0]
>>> param_grid = [{ 'clf__C': param_range,
...                  'clf__kernel': ['linear']},
...                 { 'clf__C': param_range,
...                  'clf__gamma': param_range,
...                  'clf__kernel': ['rbf']}]
>>> gs = GridSearchCV(estimator=pipe_svc,
...                     param_grid=param_grid,
...                     scoring='accuracy',
...                     cv=10,
...                     n_jobs=-1)
>>> gs = gs.fit(X_train, y_train)
>>> print(gs.best_score_)
0.978021978022
>>> print(gs.best_params_)
{'clf__C': 0.1, 'clf__kernel': 'linear'}
```

Z pomocą powyższego kodu zainicjalizowaliśmy zawarty w module `sklearn.model_selection` obiekt `GridSearchCV`, dzięki któremu wyuczyliśmy i dostroiliśmy kolejkę **maszyny wektorów nośnych (SVM)**. Przydzieliśmy do parametru `param_grid` listę słowników zawierającą hiperparametry, które chcemy stroić. Dla liniowej maszyny SVM oceniliśmy jedynie odwrotny parametr regularyzacji `C`; w przypadku funkcji jądra SVM skupiliśmy się na parametrach `C` i `gamma`. Przypominam, że parametr `gamma` jest specyficzny dla funkcji jądra SVM. Po wykorzystaniu danych uczących w procesie przeszukiwania siatki otrzymaliśmy w atrybutie `best_score_` wynik najbardziej zoptymalizowanego modelu, a najlepiej sprawujące się wartości parametrów możemy sprawdzić poprzez atrybut `best_params_`. W omawianym przykładzie liniowy model maszyny SVM o wartości parametru '`clf__C`' = 0.1 daje największą dokładność walidacji: 97,8%.

Możemy w końcu użyć niezależnego zestawu danych testowych do oszacowania skuteczności najlepszego wyszukanego modelu, co jest możliwe poprzez wprowadzenie atrybutu `best_estimator_` obiektu `GridSearchCV`:

```
>>> clf = gs.best_estimator_
>>> clf.fit(X_train, y_train)
>>> print('Dokładność testu: %.3f' % clf.score(X_test, y_test))
Dokładność testu: 0.965
```

Chociaż technika przeszukiwania siatki stanowi potężne narzędzie wyszukujące optymalny zestaw parametrów, ocena wszystkich możliwych kombinacji parametrów jest również bardzo wymagająca obliczeniowo. Alternatywnym sposobem próbkowania różnych kombinacji parametrycznych w bibliotece scikit-learn jest przeszukiwanie losowe (ang. *randomized search*). Za pomocą klasy RandomizedSearchCV jesteśmy w stanie uzyskać losowe kombinacje parametrów z rozkładów prób przy zdefiniowanym budżecie.Więcej informacji i opisów zastosowań tej klasy znajdziesz pod adresem <http://scikit-learn.org/stable/modules/grid-search.html#randomized-parameter-optimization>.

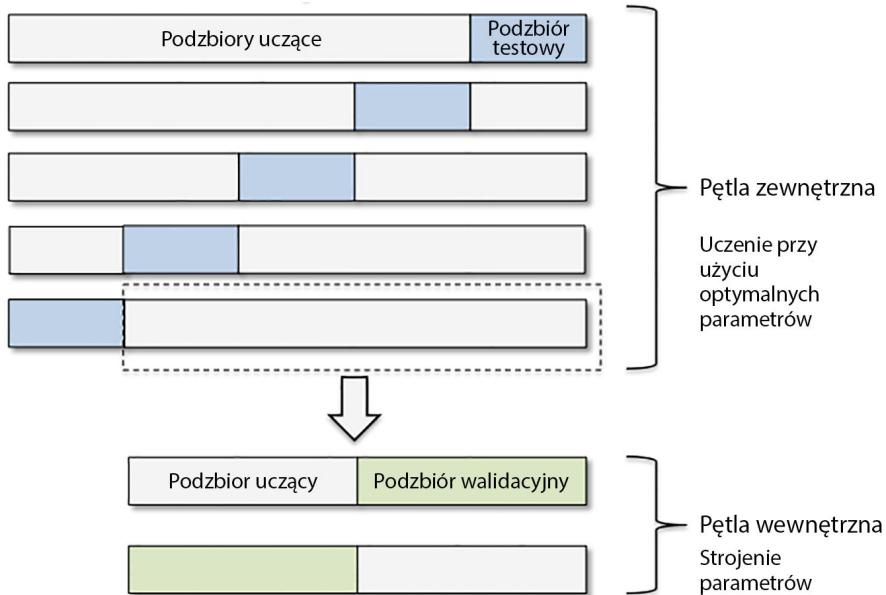
## Dobór algorytmu poprzez zagnieżdżony sprawdzian krzyżowy

Jak już zdążyliśmy się przekonać, połączenie k-krotnego sprawdzianu krzyżowego z metodą przeszukiwania siatki daje bardzo dobre rezultaty w dostrajaniu skuteczności modelu uczenia maszynowego poprzez zróżnicowanie jego wartości hiperparametrycznych. Jeśli jednak chcemy dobierać same algorytmy uczenia, lepszym rozwiązaniem okazuje się stosowanie zagnieżdżonego sprawdzianu krzyżowego (ang. *nested cross-validation*); w interesującym artykule na temat roli obciążenia w szacowaniu błędu Sudhir Varma i Richard Simon stwierdzili, że podczas korzystania ze wspomnianej metody prawdziwy błąd szacowania niemal wcale nie jest obarczony obciążeniem pochodząącym z zestawu testowego (S. Varma i R. Simon, *Bias in Error Estimation When Using Cross-validation in Model Selection*, „BMC Bioinformatics” 2006, nr 7 (1), s. 91).

W metodzie zagnieżdzonej walidacji krzyżowej mamy do czynienia z zewnętrzna pętlą k-krotnego sprawdzianu krzyżowego służącą do rozdzielania danych na podzbiory uczące i testowe oraz z pętlą wewnętrzną wykorzystywaną do doboru modelu poprzez stosowanie k-krotnego sprawdzianu krzyżowego wobec podzbioru uczącego. Po wybraniu modelu zostaje sprawdzona jego skuteczność za pomocą podzbioru testowego. Na rysunku 6.7 prezentuję koncepcję zagnieżdzonej kroswalidacji zawierającej pięć zewnętrznych podzbiorów i dwa podzbiory wewnętrzne — rozwiązanie to okazuje się przydatne w przypadku dużych zbiorów danych, gdy istotne znaczenie ma moc obliczeniowa; ta konkretna odmiana zagnieżdżonego sprawdzianu krzyżowego znana jest także jako **kroswalidacja 5x2**.

Dzięki interfejsowi scikit-learn możemy zaimplementować metodę zagnieżdżonego sprawdzianu krzyżowego w następujący sposób:

```
>>> gs = GridSearchCV(estimator=pipe_svc,
...                     param_grid=param_grid,
...                     scoring='accuracy',
...                     cv=2,
```



Rysunek 6.7. Schemat zagnieżdzonego sprawdzianu krzyżowego

```
...           n_jobs=-1)
>>> scores = cross_val_score(gs, X_train, y_train, scoring='accuracy',
...                           cv=5)
>>> print('Dokładność sprawdzianu krzyżowego: %.3f +/- %.3f' %
...       np.mean(scores), np.std(scores)))
Dokładność sprawdzianu krzyżowego: 0.965 +/- 0.025
```

Zwrócona uśredniona dokładność sprawdzianu krzyżowego pozwala nam w miarę dobrze przewidzieć, co się będzie działo z modelem po dostrojeniu jego hiperparametrów i sprawdzeniu go wobec nieznanych danych. Możemy np. wykorzystać zagnieżdżony sprawdzian krzyżowy do porównania maszyny SVM z prostym algorytmem drzewa decyzyjnego; dla uproszczenia dostroimy jedynie parametr wysokości drzewa:

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> gs = GridSearchCV(
...     estimator=DecisionTreeClassifier(random_state=0),
...     param_grid=[
...         {'max_depth': [1, 2, 3, 4, 5, 6, 7, None]}],
...     scoring='accuracy',
...     cv=5)
>>> scores = cross_val_score(gs,
...                           X_train,
...                           y_train,
...                           scoring='accuracy',
...                           cv=2)
```

```
>>> print('Dokładność sprawdzianu krzyżowego: %.3f +/- %.3f' % (
...     np.mean(scores), np.std(scores)))
Dokładność sprawdzianu krzyżowego: 0.921 +/- 0.029
```

Jak widać po przeprowadzeniu sprawdzianu krzyżowego, skuteczność modelu SVM (97,8%) jest znacznie większa od skuteczności algorytmu drzewa decyzyjnego (90,8%). Zatem możemy oczekwać, że lepszym rozwiązaniem byłoby klasyfikowanie nowych danych pochodzących z tej samej populacji próbek, co dany zestaw danych.

## Przegląd metryk oceny skuteczności

Do tej pory ocenialiśmy model na podstawie jego dokładności, co stanowi przydatną metrykę do sprawdzania jego ogólnej skuteczności. Istnieje jednak kilka innych metryk skuteczności, które można wykorzystać do sprawdzania istotności modelu, takich jak wyniki **precyzji**, **pełności** i **parametru F1**.

### Odczytywanie macierzy pomyłek

Zanim zajmiemy się szczegółami różnych metryk, przyjrzyjmy się tzw. macierzy pomyłek (ang. *confusion matrix*) — tabeli pozwalającej na określanie skuteczności algorytmu uczenia. Jest to macierz kwadratowa, w której zliczane są wyniki przewidywań klas: **prawdziwie pozytywna** (ang. *true positive*), **fałszywie pozytywna** (ang. *false positive*), **prawdziwie negatywna** (ang. *true negative*) i **fałszywie negatywna** (ang. *false negative*). Macierz pomyłek została ukazana na rysunku 6.8.

		Przewidywana klasa	
		P	N
Rzeczywista klasa	P	Prawdziwie pozytywne (PP)	Fałszywie negatywne (FN)
	N	Fałszywie pozytywne (FP)	Prawdziwie negatywne (PN)

Rysunek 6.8. Macierz pomyłek

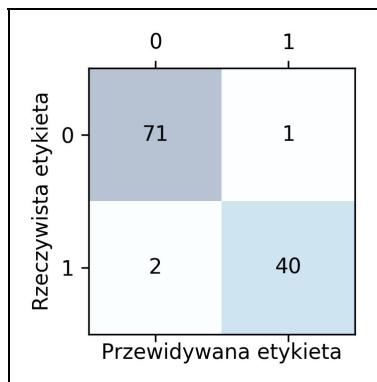
Metryki te można z łatwością obliczyć, ręcznie porównując etykiety klas rzeczywiste z przewidywanymi, ale interfejs scikit-learn zawiera wygodną funkcję `confusion_matrix`, którą możemy wykorzystać w następujący sposób:

```
>>> from sklearn.metrics import confusion_matrix
>>> pipe_svc.fit(X_train, y_train)
>>> y_pred = pipe_svc.predict(X_test)
>>> confmat = confusion_matrix(y_true=y_test, y_pred=y_pred)
>>> print(confmat)
[[71  1]
 [ 2 40]]
```

Dane zawarte w wygenerowanej tabeli informują nas o różnych rodzajach błędów popełnionych przez klasyfikator podczas analizy zestawu testowego, co możemy wyświetlić w graficzny sposób za pomocą funkcji `matshow` biblioteki `matplotlib`:

```
>>> fig, ax = plt.subplots(figsize=(2.5, 2.5))
>>> ax.matshow(confmat, cmap=plt.cm.Blues, alpha=0.3)
>>> for i in range(confmat.shape[0]):
...     for j in range(confmat.shape[1]):
...         ax.text(x=j, y=i,
...                 s=confmat[i, j],
...                 va='center', ha='center')
>>> plt.xlabel('Przewidywana etykieta')
>>> plt.ylabel('Rzeczywista etykieta')
>>> plt.show()
```

Ukazana na rysunku 6.9 macierz pomyłek powinna być teraz łatwiejsza do zinterpretowania.



Rysunek 6.9. Graficzne przedstawienie macierzy pomyłek za pomocą funkcji `matshow`

Zakładając w tym przykładzie, że klasa 1 (nowotwory złośliwe) jest pozytywną, nasz model poprawnie sklasyfikował 71 próbek należących do klasy 0 (prawdziwie negatywne) oraz 40 próbek do klasy 1 (prawdziwie pozytywne). Jednocześnie jedna próbka z klasy 0 została nieprawidłowo zaklasyfikowana do klasy 1 (fałszywie pozytywna) oraz algorytm przewidział, że dwie próbki są łagodne, mimo że w rzeczywistości reprezentują nowotwór złośliwy (fałszywie negatywne). W kolejnym ustępie wyjaśnię, w jaki sposób możemy wykorzystać te informacje do obliczania różnych metryk błędów.

## Optymalizacja precyzji i pełności modelu klasyfikującego

Zarówno błąd (*BLĄD*), jak i dokładność (*DOK*) przewidywań dostarczają informacji na temat liczby nieprawidłowo sklasyfikowanych próbek. Błąd możemy interpretować jako iloraz sumy wszystkich fałszywych przewidywań przez sumę wszystkich prognoz:

$$BLĄD = \frac{FP + FN}{FP + FN + PP + PN}$$

Dokładność predykcji możemy wyliczyć bezpośrednio z błędem:

$$DOK = \frac{PP + PN}{FP + FN + PP + PN} = 1 - BLĄD$$

**Odsetek prawdziwie pozytywnych (OPP; ang. *true positive rate*)** i **odsetek fałszywie pozytywnych (OFP; ang. *false positive rate*)** to metryki skuteczności użyteczne zwłaszcza w przypadku niezrównoważonych klas:

$$OFP = \frac{FP}{N} = \frac{FP}{FP + PN}$$

$$OPP = \frac{PP}{P} = \frac{PP}{FN + PP}$$

Przykładowo w diagnozowaniu nowotworów najważniejsze jest wykrywanie złośliwych guzów w celu jak najszybszej pomocy chorym osobom. Równie istotne jest jednak zmniejszenie liczby łagodnych nowotworów nieprawidłowo sklasyfikowanych jako złośliwe (fałszywie pozytywne), żeby nie stresować niepotrzebnie pacjentów. W przeciwnieństwie do OFP odsetek prawdziwie pozytywnych zawiera przydatne informacje dotyczące pozytywnych (ważnych) próbek, które zostały poprawnie zidentyfikowane w puli wszystkich pozytywnych danych (**P**).

**Precyzja (PRE; ang. *precision*)** i **pełność (PEŁ; ang. *recall*)** to metryki powiązane z odsetkiem prawdziwie pozytywnych oraz prawdziwie negatywnych, w rzeczywistości zaś ta druga stanowi synonim OPP:

$$PRE = \frac{PP}{PP + FP}$$

$$PEŁ = OPP = \frac{PP}{P} = \frac{PP}{FN + PP}$$

W praktyce często wykorzystywana jest kombinacja precyzji i pełności, tzw. **wynik F1** (ang. *F1 score*):

$$F1 = 2 \frac{PRE \times PEŁ}{PRE + PEŁ}$$

Wymienione metryki zliczające są zaimplementowane w bibliotece scikit-learn i możemy je importować z modułu `sklearn.metrics` w sposób ukazany w poniższym fragmencie kodu:

```
>>> from sklearn.metrics import precision_score
>>> from sklearn.metrics import recall_score, f1_score
>>> print('Precyza: %.3f' % precision_score(
...         y_true=y_test, y_pred=y_pred))
Precyza: 0.976
>>> print('Pełność: %.3f' % recall_score(
...         y_true=y_test, y_pred=y_pred))
Pełność: 0.952
>>> print('F1: %.3f' % f1_score(
...         y_true=y_test, y_pred=y_pred))
F1: 0.964
```

Możemy ponadto stosować inne metryki niż dokładność, definiując parametr `scoring` w klasie `GridSearchCV`. Pełną listę różnych wartości dostępnych dla parametru `scoring` znajdziesz pod adresem [http://scikit-learn.org/stable/modules/model\\_evaluation.html](http://scikit-learn.org/stable/modules/model_evaluation.html).

Pamiętaj, że w interfejsie scikit-learn pozytywna klasa jest oznaczona etykietą 1. Jeżeli chcemy zdefiniować inną **pozytywną etykietę**, możemy stworzyć własny algorytm zliczający (ang. `scorer`) za pomocą funkcji `make_scoring`, a następnie przekazać ją jako argument parametru `scoring`:

```
>>> from sklearn.metrics import make_scoring, f1_score
>>> scorer = make_scoring(f1_score, pos_label=0)
>>> gs = GridSearchCV(estimator=pipe_svc,
...                     param_grid=param_grid,
...                     scoring=scorer,
...                     cv=10)
```

## Wykres krzywej ROC

Wykresy krzywej charakterystyki roboczej odbiornika (ang. *receiver operating characteristic* — **ROC**) są bardzo dobrymi narzędziami doboru modelu klasyfikującego, bazującymi na jego skuteczności obliczonej na podstawie odsetka fałszywie pozytywnych i prawdziwie pozytywnych, które wyliczamy poprzez przesunięcie progu decyzyjnego klasyfikatora. Przekątną krzywej ROC możemy interpretować jako losowe zgadywanie, a skuteczność modeli klasyfikujących znajdujących się pod obszarem tej przekątnej jest uznawana za gorszą od zgadywania. Idealny klasyfikator znajdowałby się w lewym górnym rogu wykresu ( $OPP = 1$ ,  $OFP = 0$ ). Na podstawie tej krzywej możemy obliczyć tzw. **obszar pod krzywą ROC** (ang. *area under the curve* — **AUC**) opisujący skuteczność modelu klasyfikatora.

Jesteśmy również w stanie obliczać **krzywe precyzyj–pełności** dla różnych progów prawdopodobieństwa klasyfikatora. Odpowiedzialna za to funkcja jest również zaimplementowana w bibliotece scikit-learn, a jej dokumentację znajdziemy na stronie [http://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision\\_recall\\_curve.html](http://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_curve.html).

Dzięki poniższemu fragmentowi kodu wygenerujemy wykres krzywej ROC klasyfikatora wykorzystującego jedynie dwie cechy z zestawu danych Breast Cancer Wisconsin do przewidywania, czy nowotwór jest złośliwy lub łagodny. Będziemy korzystać z uprzednio zdefiniowanej kolejki regresji logistycznej, ale zwiększymy poziom trudności klasyfikacji, przez co przebieg krzywej ROC będzie dla nas bardziej interesujący. Z tego samego powodu zmniejszymy również liczbę podzbiorów w validatorze StratifiedKFold do trzech. Oto omawiany kod:

```
>>> from sklearn.metrics import roc_curve, auc
>>> from scipy import interp
>>> pipe_lr = Pipeline([('sc1', StandardScaler()),
...                     ('pca', PCA(n_components=2)),
...                     ('clf', LogisticRegression(penalty='l2',
...                                     random_state=0,
...                                     C=100.0))])
>>> X_train2 = X_train[:, [4, 14]]
>>> if Version(sklearn_version) < '0.18':
...     cv = StratifiedKFold(y_train,
...                           n_folds=3,
...                           random_state=1)
>>> else:
...     cv = list(StratifiedKFold(n_splits=3,
...                               random_state=1).split(X_train, y_train))
>>> fig = plt.figure(figsize=(7, 5))
>>> mean_tpr = 0.0
>>> mean_fpr = np.linspace(0, 1, 100)
>>> all_tpr = []

>>> for i, (train, test) in enumerate(cv):
...     probas = pipe_lr.fit(X_train2[train],
>>> y_train[train]).predict_proba(X_train2[test])
...     fpr, tpr, thresholds = roc_curve(y_train[test],
...                                         probas[:, 1],
...                                         pos_label=1)
...     mean_tpr += interp(mean_fpr, fpr, tpr)
...     mean_tpr[0] = 0.0
...     roc_auc = auc(fpr, tpr)
...     plt.plot(fpr,
...               tpr,
...               lw=1,
...               label='Podzbiór nr %d (obszar = %.2f)' %
...                   (i+1, roc_auc))
>>> plt.plot([0, 1],
...           [0, 1],
...           linestyle='--',
...           color=(0.6, 0.6, 0.6),
...           label='Losowe zgadywanie')
>>> mean_tpr /= len(cv)
>>> mean_tpr[-1] = 1.0
>>> mean_auc = auc(mean_fpr, mean_tpr)
```

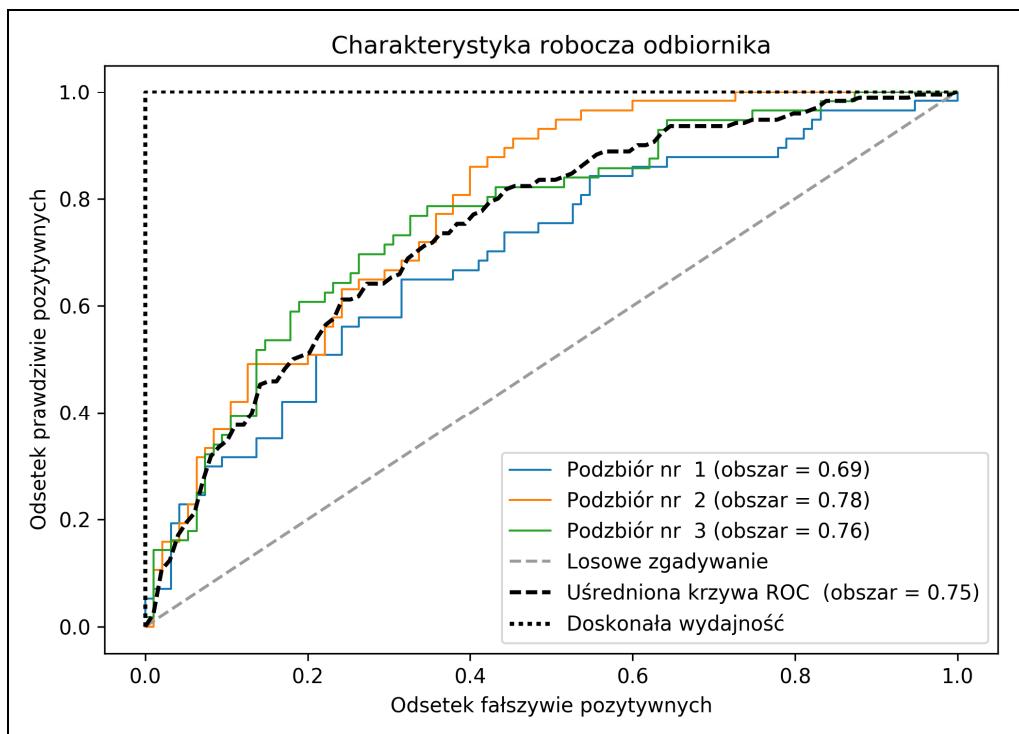
```
>>> plt.plot(mean_fpr, mean_tpr, 'k--',
...             label='Uśredniona krzywa ROC (obszar = %0.2f)' % mean_auc, lw=2)
>>> plt.plot([0, 0, 1],
...             [0, 1, 1],
...             lw=2,
...             linestyle=':',
...             color='black',
...             label='Doskonała skuteczność')
>>> plt.xlim([-0.05, 1.05])
>>> plt.ylim([-0.05, 1.05])
>>> plt.xlabel('Odsetek fałszywie pozytywnych')
>>> plt.ylabel('Odsetek prawdziwie pozytywnych')
>>> plt.title('Charakterystyka robocza odbiornika')
>>> plt.legend(loc="lower right")
>>> plt.show()
```

W powyższym kodzie zastosowaliśmy znaną nam już klasę `StratifiedKFold` i oddziennie dla każdej iteracji podzbiorów obliczyliśmy krzywą ROC klasyfikatora `LogisticRegression` (stającego częścią kolejki `pipe_lr`) przy użyciu funkcji `roc_curve` umieszczonej w module `sklearn.metrics`. Do tego dokonaliśmy interpolacji uśrednionej krzywej ROC z trzech podzbiorów za pomocą funkcji `interp` (wzięliśmy ją z biblioteki SciPy), z kolei funkcja `auc` posłużyła nam do wyznaczenia przekątnej granicznej. W rezultacie widzimy na rysunku 6.10, że istnieje pewna rozbieżność wariancji pomiędzy trzema podzbiorami, a uśredniona wartość krzywej ROC (0,75) mieści się pomiędzy wartością idealnego algorytmu (1,0) a losowym zgadywaniem (0,5).

Jeżeli interesuje nas jedynie wynik AUC, jesteśmy w stanie bezpośrednio importować funkcję `roc_auc_score` z podmodułu `sklearn.metrics`. Poniższy fragment kodu wylicza wynik obszaru pod krzywą ROC wobec niezależnego zestawu danych testowych po wyuczeniu algorytmu na dwuwymiarowym zbiorze uczącym:

```
>>> pipe_lr = pipe_lr.fit(X_train2, y_train)
>>> y_pred2 = pipe_lr.predict(X_test[:, [4, 14]])
>>> from sklearn.metrics import roc_auc_score
>>> from sklearn.metrics import accuracy_score
>>> print('Obszar pod krzywą ROC: %.3f' % roc_auc_score(
...         y_true=y_test, y_score=y_pred2))
Obszar pod krzywą ROC: 0.662
>>> print('Dokładność: %.3f' % accuracy_score(
...         y_true=y_test, y_pred=y_pred2))
Dokładność: 0.711
```

Sprawdzenie skuteczności klasyfikatora jako obszaru pod krzywą ROC pozwala obserwować jego skuteczność wobec niezrównoważonych próbek. Wynik dokładności możemy interpretować jako pojedynczy punkt graniczny krzywej ROC, ale Andrew P. Bradley udowodnił, że obszar pod krzywą ROC i metryka dokładności zazwyczaj dają bardzo zbliżone wyniki (A.P. Bradley, *The Use of the Area Under the ROC Curve in the Evaluation of Machine Learning Algorithms*, „Pattern Recognition” 1997, nr 30 (7), s. 1145 – 1159).



Rysunek 6.10. Wykres ROC dla kilku podzbiorów klasyfikowanych metodą regresji logistycznej

## Metryki zliczające dla klasyfikacji wieloklasowej

Metryki zliczające omówione w poprzednich ustępcach są charakterystyczne dla systemów klasyfikacji binarnej. Biblioteka scikit-learn zawiera jednak również metody **makro-** i **mikrouśredniania**, pozwalające na stosowanie tych metryk w problemach wieloklasowych poprzez klasyfikację typu **jeden przeciw wszystkim**. Uśrednianie mikroskopowe polega na zliczaniu średniej wartości każdej prognozy (prawdziwie pozytywnej, prawdziwie negatywnej, fałszywie pozytywnej oraz fałszywie negatywnej) w systemie. Przykładowo wartość mikro średniej precyzji w  $k$ -klasowym systemie jest obliczana następująco:

$$PRE_{mikro} = \frac{PP_1 + \dots + PP_k}{PP_1 + \dots + PP_k + FP_1 + \dots + FP_k}$$

Z kolei wartość makro średniej to po prostu uśredniona wartość sumy precyzji wszystkich systemów:

$$PRE_{makro} = \frac{PRE_1 + \dots + PRE_k}{k}$$

Mikroskopowe uśrednianie jest pozyteczne w przypadku, gdy chcemy w taki sam sposób ważyć każdą instancję lub prognozę, podczas gdy uśrednianie makroskopowe równorzęduśnie waży wszystkie klasy w celu ogólnej oceny skuteczności klasyfikatora pod względem najczęściej występujących etykiet klas.

Jeżeli używamy binarnych metryk skuteczności interfejsu scikit-learn do oceny wieloklasowych modeli, domyślnie stosowany jest wariant znormalizowanej lub ważonej średniej makroskopowej. Makroskopowa średnia ważona jest wyliczana poprzez ważenie wyników każdej etykiety klas przez liczbę prawdziwych wystąpień. Średnia taka przydaje się wtedy, gdy mamy do czynienia z niezrównoważeniem klas, tzn. niejednakową liczbą wystąpień dla poszczególnych etykiet.

Wspomnialem już, że makroskopowa średnia ważona jest stosowana domyślnie wobec problemów wieloklasowych w bibliotece scikit-learn, ale możemy wybrać inną metodę uśredniania za pomocą parametru `average` wewnętrz różnych funkcji zliczających wyniki importowanych z modułu `sklearn.metrics`, np. funkcji `precision_score` lub `make_scorer`:

```
>>> pre_scorer = make_scorer(score_func=precision_score,
...                             pos_label=1,
...                             greater_is_better=True,
...                             average='micro')
```

## Podsumowanie

Na początku niniejszego rozdziału poznaliśmy sposób łączenia różnych technik transformacji danych wejściowych i klasyfikatorów za pomocą wygodnej funkcji kolejkowania, pozwalającej na skuteczniejsze trenowanie i ocenianie modeli uczenia maszynowego. Następnie wykorzystaliśmy kolejkowanie do przeprowadzenia k-krotnego sprawdzianu krzyżowego — jednej z podstawowych technik doboru i oceny modelu. Stworzyliśmy wykresy krzywych uczenia i walidacji, które zawierają wiele informacji na temat najczęstszych problemów związanych z algorytmami uczenia maszynowego, takich jak przetrenowanie lub niewystarczające dopasowanie. Dzięki metodzie przeszukiwania siatki byliśmy w stanie jeszcze bardziej dostroić działanie modelu. Rozdział zakończyliśmy omówieniem macierzy pomyłek oraz różnych metryk skuteczności przydatnych w dokładniejszej optymalizacji algorytmów pod kątem określonego zadania problemowego. Teraz powinniśmy być wystarczająco dobrze wyposażeni w najważniejsze techniki pozwalające na skuteczne tworzenie modeli nadzorowanych klasyfikatorów.

W następnym rozdziale zapoznamy się z metodami zespołowymi — umożliwiającymi łączenie wielu modeli i algorytmów klasyfikujących w celu dalszego zwiększenia skuteczności systemu uczenia maszynowego.

# Łączenie różnych modeli w celu uczenia zespołowego

W poprzednim rozdziale skoncentrowaliśmy się na najlepszych rozwiązaniach służących do strojenia i oceny różnych modeli klasyfikujących. Teraz wykorzystamy zdobytą wiedzę i przyjrzymy się metodom tworzenia zestawów klasyfikatorów, które często wykazują znacznie lepsze właściwości predykcyjne od pojedynczych algorytmów. Nauczymy się m.in.:

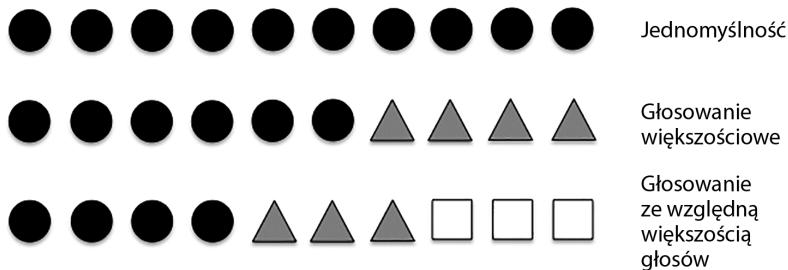
- uzyskiwać prognozy na podstawie większościowego głosowania,
- zmniejszać przetrenowanie modelu poprzez tworzenie losowych kombinacji zestawu uczącego,
- budować potężne modele ze **słabych klasyfikatorów**, które uczą się na własnych błędach.

## Uczenie zespołów

Celem **metod zespołowych** (ang. *ensemble methods*) jest łączenie różnych klasyfikatorów w jeden metaklasyfikator wykazujący większą skuteczność uogólniania niż każdy ze składowych algorytmów. Założmy, że uzyskalismy prognozy od 10 ekspertów; metody zespołowe pozwalają na strategiczne łączenie tych przewidywań w taki sposób, że uzyskamy jedną wspólną predykcję, dokładniejszą i pewniejszą niż pochodzące od poszczególnych ekspertów. Przekonamy się w dalszej części rozdziału, że istnieje kilka różnych sposobów tworzenia zespołów klasyfikatorów.

W niniejszym podrozdziale przyjrzymy się podstawowym mechanizmom działania zespołów oraz dowiemy się, dlaczego oferują one zazwyczaj bardzo dobrą skuteczność uogólniania.

Skupimy się na najpopularniejszych metodach zespołowych wykorzystujących zasadę **większościowego głosowania** (ang. *majority voting*). Jest to zjawisko polegające na wyborze etykiety klas przewidzianej przez większość klasyfikatorów, tj. takiej, która uzyskała ponad 50% głosów. Dla uściślenia — pojęcie większościowego głosowania dotyczy wyłącznie binarnych konfiguracji klas. Łatwo jednak przekształcić zasadę głosowania większościowego na zadania wieloklasowe, wtedy zostaje ona przemianowana na **głosowanie ze względną większością głosów** (ang. *plurality voting*). W tym przypadku wybieramy etykietę klas, która otrzymała najwięcej głosów (dominante). Rysunek 7.1 przedstawia koncepcję głosowania większościowego i ze względną większością głosów dla zespołu 10 klasyfikatorów — poszczególne rodzaje znaków (trójkąt, kwadrat i kółko) reprezentują różne etykiety klas.



Rysunek 7.1. Porównanie głosowania większościowego z głosowaniem ze względną większością głosów

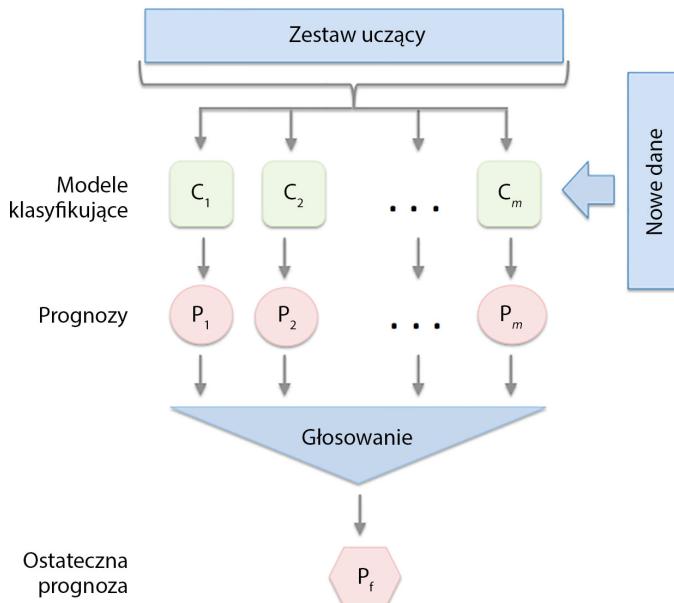
Korzystając z zestawu uczącego, najpierw uczymy  $m$  klasyfikatorów ( $C_1, \dots, C_m$ ). W zależności od używanej techniki zespół może być skonstruowany z różnych algorytmów klasyfikacji, np. drzew decyzyjnych, maszyny wektorów nośnych, klasyfikatorów regresji logistycznej itd. Ewentualnie możemy również wykorzystać ten sam bazowy algorytm uczenia dopasowujący się do różnych podzbiorów zestawu uczącego. Znakiomym przykładem jest tu algorytm losowego lasu, łączący różne klasyfikatory drzew decyzyjnych. Na rysunku 7.2 zaprezentowałem ogólną koncepcję metody zespołowej wykorzystującej głosowanie większościowe.

Aby przewidzieć etykietę klasy za pomocą prostego głosowania większościowego lub ze względną większością głosów, łączymy prognozy etykiet klas pochodzące z każdego pojedynczego klasyfikatora  $C_j$ , a następnie wybieramy etykietę  $\hat{y}$ , która otrzymała najwięcej głosów:

$$\hat{y} = \text{moda} \{C_1(\mathbf{x}), C_2(\mathbf{x}), \dots, C_m(\mathbf{x})\}$$

Przykładowo w zadaniu klasyfikacji binarnej, gdzie  $klasa1 = -1$ , a  $klasa2 = +1$ , możemy zapisać przewidywanie wyznaczone głosowaniem większościowym w następujący sposób:

$$C(\mathbf{x}) = \text{sgn} \left[ \sum_j^m C_j(\mathbf{x}) \right] = \begin{cases} 1 & \text{jeśli } \sum_i C_j(\mathbf{x}) \geq 0 \\ -1 & \text{jeśli } \sum_i C_j(\mathbf{x}) < 0 \end{cases}$$



Rysunek 7.2. Schemat działania metody zespołowej z wykorzystaniem głosowania większościowego

W celu wyjaśnienia przyczyn znakomitej skuteczności metod zespołowych wykorzystamy proste reguły kombinatoryki. Zakładamy w poniższym przykładzie, że wszystkie  $n$  bazowych algorytmów klasyfikacji binarnej mają taką samą stopę błędu  $\varepsilon$ . Ponadto klasyfikatory są od siebie wzajemnie niezależne, a stopy błędu nie są ze sobą skorelowane. W takiej sytuacji możemy sformułować prawdopodobieństwo popełnienia błędu przez zespół bazowych klasyfikatorów jako funkcję masy prawdopodobieństwa dla rozkładu dwumianowego:

$$P(y \geq k) = \sum_{k=0}^n \binom{n}{k} \varepsilon^k (1-\varepsilon)^{n-k} = \varepsilon_{\text{zespoł}}$$

Tutaj  $\binom{n}{k}$  jest współczynnikiem dwumianowym (**n po k**). Innymi słowy, obliczamy prawdopodobieństwo wystąpienia błędnej prognozy wygenerowanej przez zespół. Przyjrzyjmy się teraz praktycznemu przykładowi, w którym mamy 11 podstawowych klasyfikatorów ( $n = 11$ ) oraz stopę błędu na poziomie 0,25 ( $\varepsilon = 0,25$ ):

$$P(y \geq k) = \sum_{k=6}^{11} \binom{11}{k} 0,25^k (1-0,25)^{11-k} = 0,034$$

Jak widać, po spełnieniu wszystkich założeń stopa błędu całego zespołu (0,034) jest znacznie niższa od analogicznej wartości poszczególnych klasyfikatorów (0,25). Zwrót uwagę, że dla uproszczenia wyniki połowy z  $n$  klasyfikatorów są uznawane za błędne, w rzeczywistości zaś jest to

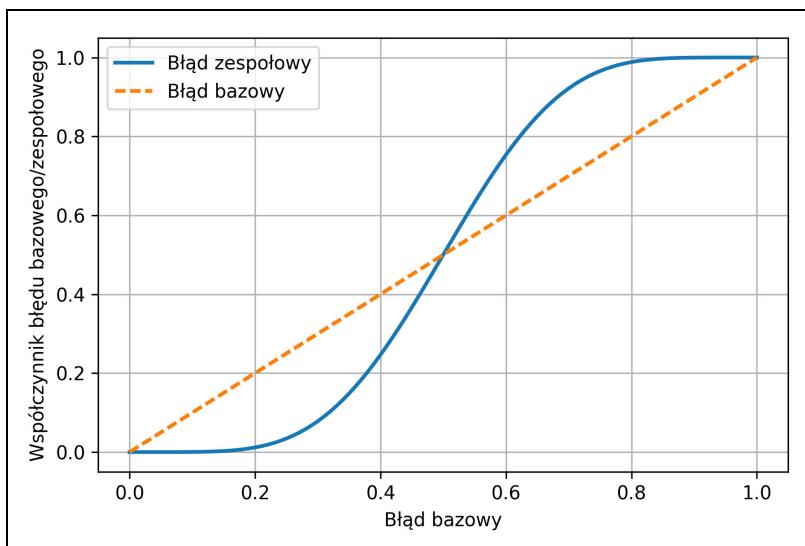
prawdziwe jedynie w 50% sytuacji. Aby porównać taki wyidealizowany klasyfikator zespołowy z klasyfikatorem bazowym dla różnych wartości stopy błędu, zaimplementujmy funkcję masy prawdopodobieństwa w Pythonie:

```
>>> from scipy.misc import comb
>>> import math
>>> def ensemble_error(n_classifier, error):
...     k_start = int(math.ceil(n_classifier / 2.0))
...     probs = [comb(n_classifier, k) *
...               error**k *
...               (1-error)**(n_classifier - k)
...             for k in range(k_start, n_classifier + 1)]
...     return sum(probs)
>>> ensemble_error(n_classifier=11, error=0.25)
0.034327507019042969
```

Po zaimplementowaniu funkcji `ensemble_error` możemy wyliczyć stopy błędu dla różnych zakresów bazowych stóp błędu (w przedziale od 0,0 – 1,0), żeby zwizualizować na wykresie relacje pomiędzy tymi dwoma parametrami:

```
>>> import numpy as np
>>> error_range = np.arange(0.0, 1.01, 0.01)
>>> ens_errors = [ensemble_error(n_classifier=11, error=error)
...                 for error in error_range]
>>> import matplotlib.pyplot as plt
>>> plt.plot(error_range, ens_errors,
...            label='Błąd zespołowy',
...            linewidth=2)
>>> plt.plot(error_range, error_range,
...            linestyle='--', label='Błąd bazowy',
...            linewidth=2)
>>> plt.xlabel('Błąd bazowy')
>>> plt.ylabel('Współczynnik błędu bazowego/zespołowego')
>>> plt.legend(loc='upper left')
>>> plt.grid()
>>> plt.show()
```

Jak widać na rysunku 7.3, prawdopodobieństwo wystąpienia błędu w zespole jest zawsze mniejsze od ryzyka wygenerowania błędu przez pojedyncze klasyfikatory bazowe w przypadku, gdy te poszczególne algorytmy są skuteczniejsze od losowego zgadywania ( $\epsilon < 0,5$ ). Zwróć uwagę, że osi y reprezentuje zarówno błąd bazowy (linia przerywana), jak i błąd zespołowy (linia ciągła).



Rysunek 7.3. Wykres porównania stopy błędu algorytmu bazowego ze stopą błędu zespołu klasyfikatorów

## Implementacja prostego klasyfikatora wykorzystującego głosowanie większościowe

Po krótkim wprowadzeniu w świat uczenia zespołów możemy przejść do „rozgrzewki” i zaimplementować w Pythonie prosty klasyfikator wykorzystujący głosowanie większościowe. Chociaż omawiany algorytm może być stosowany również w zagadnieniach wieloklasowych poprzez głosowanie ze względną większością głosów, dla uproszczenia będziemy mówić o **głosowaniu większościowym**, zgodnie z utartym w literaturze schematem.

Implementowany przez nas algorytm pozwoli nam na łączenie różnych klasyfikatorów dla naszej wygody powiązanych z poszczególnymi wagami. Naszym celem jest stworzenie potężniejszego metaklasyfikatora równoważącego wady poszczególnych algorytmów składowych w odniesieniu do danego zestawu próbek. W matematycznym ujęciu możemy sformułować ważone głosowanie większościowe następująco:

$$\hat{y} = \arg \max_i \sum_{j=1}^m w_j \chi_A(C_j(\mathbf{x}) = i)$$

Tutaj  $w_j$  jest wagą powiązaną z klasyfikatorem bazowym  $C_j$ ,  $\hat{y}$  to prognozowana przez zespół etykieta klas,  $\chi_A$  (grecka litera **chi**) stanowi funkcję charakterystyczną [ $C_j(x) = i \in A$ ], natomiast  $A$  symbolizuje zbiór unikatowych etykiet klas. W przypadku równych wag możemy uproszczyć powyższe równanie i zapisać je w sposób przedstawiony poniżej:

$$\hat{y} = \text{moda}\{C_1(\mathbf{x}), C_2(\mathbf{x}), \dots, C_m(\mathbf{x})\}$$

Aby lepiej zrozumieć koncepcję **ważenia**, przeanalizujemy konkretny przykład. Założymy, że stworzyliśmy zespół złożony z trzech bazowych klasyfikatorów  $C_j$  ( $j \in \{0, 1\}$ ) i chcemy przewidzieć etykietę klas danej próbki  $x$ . Dwa klasyfikatory bazowe klasyfikują ją do klasy 0, a jeden z nich ( $C_3$ ) prognozuje, że próbka należy do klasy 1. Jeżeli nadamy przewidywaniom każdego klasyfikatora takie same wagi, to zgodnie z głosowaniem większościowym zespół uzna, że próbka jest częścią klasy 0:

$$\begin{aligned} C_1(x) &\rightarrow 0, C_2(x) \rightarrow 0, C_3(x) \rightarrow 1 \\ \hat{y} &= \text{moda}\{0, 0, 1\} = 0 \end{aligned}$$

Wyznaczmy teraz wagę 0,6 kwalifikatorowi  $C_3$ , a algorytmom  $C_1$  i  $C_2$  — wartość 0,2.

$$\begin{aligned} \hat{y} &= \arg \max_i \sum_{j=1}^m w_j \chi_A(C_j(\mathbf{x}) = i) = \\ &= \arg \max_i [0, 2 \times i_0 + 0, 2 \times i_0 + 0, 6 \times i_1] = 1 \end{aligned}$$

Zgodnie z intuicją wyczuwamy, że skoro  $3 \times 0,2 = 0,6$ , to prognoza wyznaczona przez klasyfikator  $C_3$  ma trzykrotnie większą wagę od przewidywań algorytmów  $C_1$  i  $C_2$ . Możemy zapisać to następująco:

$$\hat{y} = \text{moda}\{0, 0, 1, 1, 1\} = 1$$

Do przełożenia koncepcji ważonego głosowania większościowego na kod Pythona możemy wykorzystać funkcję biblioteki NumPy: `argmax` i `bincount`:

```
>>> import numpy as np
>>> np.argmax(np.bincount([0, 0, 1],
...                      weights=[0.2, 0.2, 0.6]))
1
```

Z rozdziału 3., „Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn”, wiemy, że niektóre klasyfikatory stanowiące część interfejsu scikit-learn mogą zwracać prawdopodobieństwo prognozowanej etykiety klas również za pomocą metody `predict_proba`. Korzystanie z prawdopodobieństwa przewidywań zamiast z etykiet klas w głosowaniu większościowym bywa przydatne w przypadku, gdy klasyfikatory tworzące zespół są dobrze skalibrowane. Taką zmodyfikowaną wersję głosowania większościowego możemy sformułować tak, jak zaprezentowałem poniżej:

$$\hat{y} = \arg \max_i \sum_{j=1}^m w_j p_{ij}$$

W powyższym wzorze  $p_{ij}$  oznacza przewidywane prawdopodobieństwo  $j$ -tego klasyfikatora dla  $i$ -tej etykiety klas.

Kontynuując opisywany przykład, założymy, że mamy do czynienia z zagadnieniem klasyfikacji liniowej, w której etykiety klas są oznaczone jako  $i \in \{0, 1\}$ , a zespół składa się z trzech klasyfikatorów:  $C_j$  ( $j \in \{1, 2, 3\}$ ). Powiedzmy, że klasyfikator  $C_1$  zwraca następujące prawdopodobieństwa przynależności próbki  $x$  do wyznaczonej grupy:

$$C_1(x) \rightarrow [0, 0, 9, 0, 1], C_2(x) \rightarrow [0, 8, 0, 2], C_3(x) \rightarrow [0, 4, 0, 6]$$

Obliczmy teraz pojedyncze prawdopodobieństwa prognoz:

$$\begin{aligned} p(i_0|x) &= 0,2 \times 0,9 + 0,2 \times 0,8 + 0,6 \times 0,4 = 0,58 \\ p(i_1|x) &= 0,2 \times 0,1 + 0,2 \times 0,2 + 0,6 \times 0,6 = 0,42 \\ \hat{y} &= \arg \max_i [p(i_0|x), p(i_1|x)] = 0 \end{aligned}$$

Do zaimplementowania ważonego głosowania większościowego bazującego na prawdopodobieństwie wystąpienia klas użyjemy funkcji `np.average` i `np.argmax`:

```
>>> ex = np.array([[0.9, 0.1],
...                 [0.8, 0.2],
...                 [0.4, 0.6]])
>>> p = np.average(ex, axis=0, weights=[0.2, 0.2, 0.6])
>>> p
array([ 0.58,  0.42])
>>> np.argmax(p)
0
```

Poskładajmy wszystko w całość, aby zaimplementować obiekt `MajorityVoteClassifier` w Pythonie:

```
from sklearn.base import BaseEstimator
from sklearn.base import ClassifierMixin
from sklearn.preprocessing import LabelEncoder
from sklearn.externals import six
from sklearn.base import clone
from sklearn.pipeline import _name_estimators
import numpy as np
import operator

class MajorityVoteClassifier(BaseEstimator,
                            ClassifierMixin):
```

""" Klasyfikator zespołowy (głosowanie większościowe)

Parametry

-----  
classifiers : tablicopodobny, wymiary = [n\_klasyfikatorów]  
Różne klasyfikatory tworzące zespół

vote : łańcuch znaków, {'classlabel', 'probability'}

Domyślnie: 'classlabel'

Jeżeli jest wybrany argument 'classlabel', to prognoza  
jest przeprowadzana przy użyciu funkcji  
argmax wobec etykiet klas. W przeciwnym wypadku  
('probability') wynik funkcji argmax wobec sumy  
prawdopodobieństw zostaje użyty do prognozowania  
etykiety klas (zalecane dla skalibrowanych klasyfikatorów).

weights : tablicopodobne, wymiary = [n\_klasyfikatorów]

Opcjonalny, domyślnie: None

Po wprowadzeniu listy wartości typu `int` lub `float`  
klasyfikatory są ważone pod kątem ważności. Jeśli  
'weights=None', wykorzystywane są takie same wagи.

"""

```
def __init__(self, classifiers,  
            vote='classlabel', weights=None):  
  
    self.classifiers = classifiers  
    self.named_classifiers = {key: value for  
                             key, value in  
                             _name_estimators(classifiers)}  
    self.vote = vote  
    self.weights = weights  
  
def fit(self, X, y):  
    """ Dopasowywanie klasyfikatorów.
```

Parametry

-----  
X : {tablicopodobny, macierz rzadka},  
 wymiary = [n\_próbek, n\_cech]  
 Macierz próbek uczących.

y : tablicopodobny, wymiary = [n\_próbek]  
 Wektor docelowych etykiet klas.

Zwraca

-----  
self : obiekt

```
"""
# Dzięki klasie LabelEncoder etykiety klas rozpoczynają się
# od wartości 0, co jest bardzo ważne podczas wywołania np. argmax
# w self.predict
self.lablenc_ = LabelEncoder()
self.lablenc_.fit(y)
self.classes_ = self.lablenc_.classes_
self.classifiers_ = []
for clf in self.classifiers:
    fitted_clf = clone(clf).fit(X,
                                 self.lablenc_.transform(y))
    self.classifiers_.append(fitted_clf)
return self
```

Wstawilem do kodu wiele komentarzy, żeby ułatwić zrozumienie poszczególnych fragmentów. Zanim jednak zaimplementujemy pozostałe metody, przyjrzymy się najpierw wierszom kodu, które na pierwszy rzut oka mogą budzić wątpliwości. Skorzystaliśmy z nadrzędnych klas `BaseEstimator` i `ClassifierMixin`, aby za darmo uzyskać pewne podstawowe funkcje, w tym metody `get_params` i `set_params`, służące do konfigurowania i zwracania parametrów klasyfikatora; jak również metodę `score` wyliczającą dokładność prognoz. Importowaliśmy też pakiet `six` po to, aby obiekt `MajorityVoteClassifier` był kompatybilny z Pythonem w wersji 2.7.

Dodamy teraz metodę `predict` przewidującą etykietę klas poprzez głosowanie większościowe bazujące na etykietach klas (jeśli zainicjujemy nowy obiekt `MajorityVoteClassifier` z parametrem `vote='classlabel'`). Możemy ewentualnie zainicjować klasyfikator zespołowy z parametrem `vote='probability'`, dzięki czemu etykiety klas będą prognozowane na podstawie prawdopodobieństwa przynależności określonej instancji do wyznaczonych grup. Ponadto wstawimy metodę `predict_proba` zwracającą średnie wartości prawdopodobieństwa, co przyda nam się do obliczenia obszaru pod krzywą ROC.

```
def predict(self, X):
    """
    Prognozowanie etykiet klas dla próbki X.

    Parametry
    -----
    X : tablicopodobny, macierz rzadka},
        wymiary = [n_próbek, n_cech]
        Macierz próbek uczących.

    Zwraca
    -----
    maj_vote : tablicopodobny, wymiary = [n_próbek]
        Przewidywane etykiety klas.

    """
    if self.vote == 'probability':
        maj_vote = np.argmax(self.predict_proba(X),
                             axis=1)
```

```

else: # wartość 'classlabel'

    # zbiera wyniki z wywołań metody clf.predict
    predictions = np.asarray([clf.predict(X)
                               for clf in
                               self.classifiers_]).T

    maj_vote = np.apply_along_axis(
        lambda x:
            np.argmax(np.bincount(x,
                                  weights=self.weights)),
        axis=1,
        arr=predictions)
    maj_vote = self.lablenc_.inverse_transform(maj_vote)
    return maj_vote

def predict_proba(self, X):
    """ Prognozowania prawdopodobieństwa przynależności próbki X do danej klasy.

    Parametry
    -----
    X : {tablicopodobny, macierz rzadka},
        wymiary = [n_próbek, n_cech]
        Wektory uczenia, gdzie n_próbek oznacza
        liczbę próbek, a n_cech — liczbę cech.

    Zwraca
    -----
    avg_proba : tablicopodobny,
        wymiary = [n_próbek, n_klas]
        Ważone, uśrednione prawdopodobieństwo
        wystąpienia każdej klasy na daną próbkę.

    """
    probas = np.asarray([clf.predict_proba(X)
                         for clf in self.classifiers_])
    avg_proba = np.average(probas,
                           axis=0, weights=self.weights)
    return avg_proba

def get_params(self, deep=True):
    """ pobiera nazwy parametrów klasyfikatora dla klasy GridSearch"""
    if not deep:
        return super(MajorityVoteClassifier,
                     self).get_params(deep=False)
    else:
        out = self.named_classifiers.copy()
        for name, step in \
            six.iteritems(self.named_classifiers):

```

```

for key, value in six.iteritems(
    step.get_params(deep=True)):
    out['%s__%s' % (name, key)] = value
return out

```

Zwróć również uwagę, że zdefiniowaliśmy własną, zmodyfikowaną wersję metody `get_params`, wykorzystującą funkcję `_name_estimators` do uzyskiwania dostępu do parametrów poszczególnych składowych zespołu. Może się to wydawać początkowo dość skomplikowane, ale stanie się całkowicie zrozumiałe w dalszej części rozdziału, gdy zaczniemy używać przeszukiwania siatki do strojenia hiperparametrów.

Implementacja obiektu `MajorityVoteClassifier` nadaje się świetnie do zilustrowania działania zespołów klasyfikatorów, ale umieściłem w bibliotece scikit-learn bardziej zaawansowaną wersję klasyfikatora wykorzystującego głosowanie większościowe. Jest to zamieszczona w wersji 0.17 interfejsu klasa `sklearn.ensemble.VotingClassifier`.

## Łączenie różnych algorytmów w celu klasyfikacji za pomocą głosowania większościowego

Najwyższa pora, aby sprawdzić działanie stworzonego przez nas klasyfikatora `MajorityVoteClassifier`. Przygotujmy jednak najpierw zestaw danych, na których będziemy mogli przetestować skuteczność modelu. Potrafimy już wczytywać zestawy danych z plików CSV, dlatego pojedziemy na skróty i załadujemy zbiór danych **Iris** z modułu danych interfejsu scikit-learn. Poza tym wybierzemy tylko dwie cechy — szerokość działki i długość płatka, aby nieco utrudnić proces klasyfikacji. Obiekt `MajorityVoteClassifier` uogólnia rozwiązania również do wieloklasowych problemów, mimo to do obliczenia **obszaru pod krzywą ROC** będziemy klasyfikować próbki kwiatów pochodzące wyłącznie z dwóch klas: **Iris-versicolor** i **Iris-virginica**. Poniżej prezentuję gotowy fragment kodu:

```

>>> from sklearn import datasets
>>> from distutils.version import LooseVersion as Version
>>> from sklearn import __version__ as sklearn_version
>>> if Version(sklearn_version) < '0.18':
>>>     from sklearn.cross_validation import train_test_split
>>> else:
>>>     from sklearn.model_selection import train_test_split
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.preprocessing import LabelEncoder
>>> iris = datasets.load_iris()
>>> X, y = iris.data[50:, [1, 2]], iris.target[50:]
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)

```

Zauważmy, że biblioteka scikit-learn wykorzystuje metodę predict\_proba (jeśli istnieje taka możliwość) do obliczania wyniku obszaru pod krzywą ROC. W rozdziale 3., „Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn”, pokazałem, w jaki sposób prawdopodobieństwa przynależności do klas są wyliczane w modelach regresji logistycznej. W drzewach decyzyjnych prawdopodobieństwo jest obliczane za pomocą wektora częstotliwości tworzonego dla każdego węzła na etapie uczenia. Wektor ten gromadzi częstotliwości pojawiania się poszczególnych etykiet klas na podstawie rozkładu etykiet klas w danym węźle. Następnie częstotliwości te są normalizowane tak, że po zsumowaniu dają wartość 1. W podobny sposób są zbierane etykiety klas w modelu k-najbliższych sąsiadów, dzięki czemu zwarcane są znormalizowane częstości występowania etykiet klas w tym algorytmie. Choćiąż tego typu znormalizowane prawdopodobieństwa przypominają analogiczne wartości wyliczane w modelu regresji logistycznej, pamiętajmy, że nie uzyskujemy ich z funkcji masy prawdopodobieństwa.

Rozdzielimy próbki z zestawu danych Iris na 50% danych uczących i 50% danych testowych:

```
>>> X_train, X_test, y_train, y_test = \
...     train_test_split(X, y,
...                     test_size=0.5,
...                     random_state=1)
```

Z pomocą zestawu danych uczących wytrenujemy teraz trzy rodzaje klasyfikatorów — model regresji logistycznej, drzewa decyzyjnego oraz k-najbliższych sąsiadów — i dzięki zbiorowi danych testowych sprawdzimy skuteczność każdego z nich metodą dziesięciokrotnego sprawdzianu krzyżowego, a dopiero w następnej kolejności połączymy je w jeden klasyfikator zespołowy:

```
>>> from distutils.version import LooseVersion as Version
>>> from sklearn import __version__ as sklearn_version
>>> if Version(sklearn_version) < '0.18':
>>>     from sklearn.cross_validation import cross_val_score
>>> else:
>>>     from sklearn.model_selection import cross_val_score
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.tree import DecisionTreeClassifier
>>> from sklearn.neighbors import KNeighborsClassifier
>>> from sklearn.pipeline import Pipeline
>>> import numpy as np
>>> clf1 = LogisticRegression(penalty='l2',
...                             C=0.001,
...                             random_state=0)
>>> clf2 = DecisionTreeClassifier(max_depth=1,
...                                 criterion='entropy',
...                                 random_state=0)
>>> clf3 = KNeighborsClassifier(n_neighbors=1,
...                             p=2,
...                             metric='minkowski')
>>> pipe1 = Pipeline([('sc', StandardScaler()),
...                   ['clf', clf1]])
>>> pipe3 = Pipeline([('sc', StandardScaler())],
```

```

...          ['clf', clf3]])
>>> clf_labels = ['Regresja logistyczna', 'Drzewo decyzyjne',
    'K-najbliżsi sąsiedzi']
>>> print('Dziesięciokrotny sprawdzian krzyżowy:\n')
>>> for clf, label in zip([pipe1, clf2, pipe3], clf_labels):
...     scores = cross_val_score(estimator=clf,
...                               X=X_train,
...                               y=y_train,
...                               cv=10,
...                               scoring='roc_auc')
>>> print("Obszar pod krzywą ROC: %0.2f (+/- %0.2f) [%s]" %
...           (scores.mean(), scores.std(), label))

```

Poniżej prezentuję otrzymany rezultat, z którego wynika, że skuteczność predykcyjna poszczególnych klasyfikatorów daje bardzo zbliżone rezultaty:

*Dziesięciokrotny sprawdzian krzyżowy:*

*Obszar pod krzywą ROC: 0.92 (+/- 0.20) [Regresja logistyczna]  
 Obszar pod krzywą ROC: 0.92 (+/- 0.15) [Drzewo decyzyjne]  
 Obszar pod krzywą ROC: 0.93 (+/- 0.10) [K-najbliżsi sąsiedzi]*

Zastanawiasz się być może, dlaczego proces uczenia modelu regresji logistycznej i k-najbliższych sąsiadów wprowadziliśmy jako część **kolejki**. Przyczyna tego, jak zostało omówione w rozdziale 3., „Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn”, wynika z faktu, że obydwa algorytmy (wykorzystujące metrykę odległości euklidesowych), w przeciwieństwie do modelu drzewa decyzyjnego, są wrażliwe na skalę prezentowanych próbek. Pomimo że wszystkie cechy zestawu danych Iris są mierzone w tej samej skali (centymetrach), dobrym zwyczajem jest praca na standaryzowanych cechach.

Przejdźmy do ciekawszej części i połączmy poszczególne klasyfikatory tak, aby brały udział w głosowaniu większościowym wewnątrz obiektu `MajorityVoteClassifier`:

```

>>> mv_clf = MajorityVoteClassifier(
...                           classifiers=[pipe1, clf2, pipe3])
>>> clf_labels += ['Głosowanie większościowe']
>>> all_clf = [pipe1, clf2, pipe3, mv_clf]
>>> for clf, label in zip(all_clf, clf_labels):
...     scores = cross_val_score(estimator=clf,
...                               X=X_train,
...                               y=y_train,
...                               cv=10,
...                               scoring='roc_auc')
...     print("Obszar pod krzywą ROC: %0.2f (+/- %0.2f) [%s]" %
...           (scores.mean(), scores.std(), label))
Obszar pod krzywą ROC: 0.92 (+/- 0.20) [Regresja logistyczna]
Obszar pod krzywą ROC: 0.92 (+/- 0.15) [Drzewo decyzyjne]
Obszar pod krzywą ROC: 0.93 (+/- 0.10) [K-najbliżsi sąsiedzi]
Obszar pod krzywą ROC: 0.97 (+/- 0.10) [Głosowanie większościowe]

```

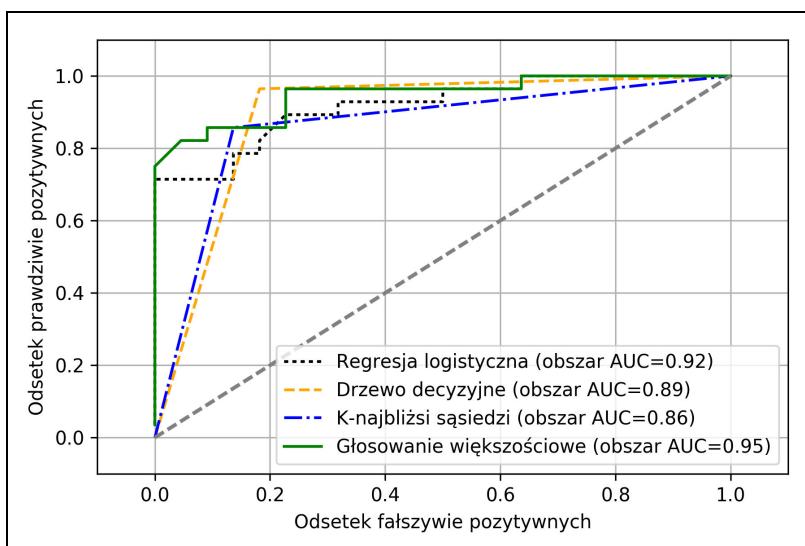
Jak widać, skuteczność obiektu `MajorityVoteClassifier` jest znacznie wyższa od skuteczności pojedynczych klasyfikatorów testowanych za pomocą dziesięciokrotnego sprawdzianu krzyżowego.

## Ewaluacja i strojenie klasyfikatora zespołowego

Przejdziemy teraz do obliczania krzywych ROC przy użyciu danych testowych, aby sprawdzić, jak klasyfikator `MajorityVoteClassifier` radzi sobie z nieznanymi danymi. Pamiętasz zapewne, że zestaw testowy nie jest wykorzystywany na etapie doboru modelu; jego jedyne zadanie to niczym nieobciążone oszacowanie skuteczności uogólniania systemu klasyfikującego. Odpowiedzialny jest za to następujący fragment kodu:

```
>>> from sklearn.metrics import roc_curve
>>> from sklearn.metrics import auc
>>> colors = ['black', 'orange', 'blue', 'green']
>>> linestyles = [':', '--', '-.', '-']
>>> for clf, label, clr, ls \
...     in zip(all_clf, clf_labels, colors, linestyles):
...     # przy założeniu, że etykieta klasy pozytywnej to 1
...     y_pred = clf.fit(X_train,
...                       y_train).predict_proba(X_test)[:, 1]
...     fpr, tpr, thresholds = roc_curve(y_true=y_test,
...                                      y_score=y_pred)
...     roc_auc = auc(x=fpr, y=tpr)
...     plt.plot(fpr, tpr,
...               color=clr,
...               linestyle=ls,
...               label='%s (obszar auc = %0.2f)' % (label, roc_auc))
>>> plt.legend(loc='lower right')
>>> plt.plot([0, 1], [0, 1],
...           linestyle='--',
...           color='gray',
...           linewidth=2)
>>> plt.xlim([-0.1, 1.1])
>>> plt.ylim([-0.1, 1.1])
>>> plt.grid()
>>> plt.xlabel('Odsetek fałszywie pozytywnych')
>>> plt.ylabel('Odsetek prawdziwie pozytywnych')
>>> plt.show()
```

Z zaprezentowanego na rysunku 7.4 wykresu krzywych ROC wynika jasno, że klasyfikator zespołowy bardzo dobrze sobie radzi z danymi testowymi (obszar AUC = 0,95), podczas gdy algorytm k-najbliższych sąsiadów okazuje się nadmiernie dopasowany (obszar AUC dla danych uczących = 0,93, obszar AUC dla danych testowych = 0,86).



Rysunek 7.4. Wykres ROC klasyfikatora zespołowego

W celach klasyfikacji podaliśmy jedynie dwie cechy zestawu danych, dlatego byłoby dla nas bardzo interesujące sprawdzenie, jak wyglądają regiony decyzyjne wygenerowane przez nasz klasyfikator zespołowy. Chociaż nie musimy standaryzować cech uczących przed dopasowywaniem modelu (algorytmy regresji logistycznej i k-najbliższych sąsiadów wykonują to automatycznie), przeprowadzimy standaryzację zestawu uczącego po to, aby regiony decyzyjne generowane przez model drzewa decyzyjnego prezentowały się w tej samej skali co regiony decyzyjne z pozostałych algorytmów. Poniżej znajdziesz odpowiedni fragment kodu:

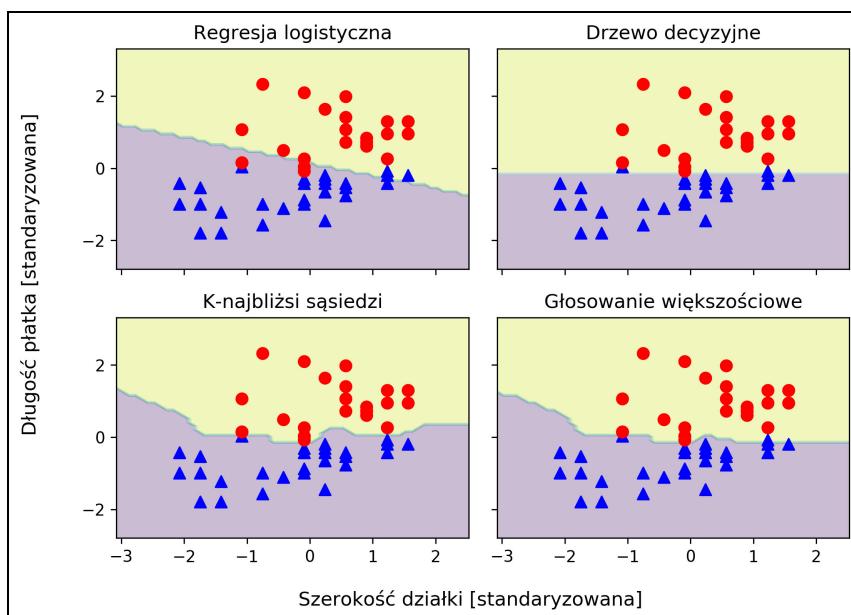
```
>>> sc = StandardScaler()
>>> X_train_std = sc.fit_transform(X_train)
>>> from itertools import product
>>> x_min = X_train_std[:, 0].min() - 1
>>> x_max = X_train_std[:, 0].max() + 1
>>> y_min = X_train_std[:, 1].min() - 1
>>> y_max = X_train_std[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                      np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(nrows=2, ncols=2,
...                         sharex='col',
...                         sharey='row',
...                         figsize=(7, 5))
>>> for idx, clf, tt in zip(product([0, 1], [0, 1]),
...                           all_clf, clf_labels):
...     clf.fit(X_train_std, y_train)
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx[0], idx[1]].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx[0], idx[1]].scatter(X_train_std[y_train==0, 0],
```

```

...
X_train_std[y_train==0, 1],
...
c='blue',
marker='^',
s=50)
...
axarr[idx[0], idx[1]].scatter(X_train_std[y_train==1, 0],
...
X_train_std[y_train==1, 1],
...
c='red',
marker='o',
s=50)
...
axarr[idx[0], idx[1]].set_title(tt)
>>> plt.text(-3.5, -4.5,
...
s='Szerokość działki [standaryzowana]',
...
ha='center', va='center', fontsize=12)
>>> plt.text(-10.5, 4.5,
...
s='Długość pątka [standaryzowana]',
...
ha='center', va='center',
...
fontsize=12, rotation=90)
>>> plt.show()

```

Co ciekawe (ale również oczekiwane), regiony decyzyjne klasyfikatora zespołowego okazują się hybrydą regionów decyzyjnych z poszczególnych składowych modeli (rysunek 7.5). Na pierwszy rzut oka granica decyzyjna klasyfikatora bazującego na głosowaniu większościowym przypomina analogiczną granicę utworzoną przez algorytm k-najbliższych sąsiadów. Widzimy jednak, że dla **szerokości działki  $\geq 1$**  przebieg linii granicznej jest prostopadły wobec osi  $y$ , zupełnie jak w modelu drzewa decyzyjnego.



Rysunek 7.5. Porównanie regionów decyzyjnych generowanych przez pojedyncze klasyfikatory i ich zespół

Zanim nauczymy się dostrajać poszczególne klasyfikatory składowe w celu zoptymalizowania klasyfikacji zespołowej, wywołajmy metodę `get_params`, dzięki czemu dowiemy się, w jaki sposób uzyskujemy dostęp do poszczególnych parametrów obiektu `GridSearch`:

```
>>> mv_clf.get_params()
{'decisiontreeclassifier': DecisionTreeClassifier(class_weight=None,
criterion='entropy', max_depth=1,
max_features=None, max_leaf_nodes=None, min_samples_
leaf=1,
min_samples_split=2, min_weight_fraction_leaf=0.0,
random_state=0, splitter='best'),
'decisiontreeclassifier_class_weight': None,
'decisiontreeclassifier_criterion': 'entropy',
[...]
'decisiontreeclassifier_random_state': 0,
'decisiontreeclassifier_splitter': 'best',
'pipeline-1': Pipeline(steps=[('sc', StandardScaler(copy=True, with_
mean=True, with_std=True)), ('clf', LogisticRegression(C=0.001, class_
weight=None, dual=False, fit_intercept=True,
intercept_scaling=1, max_iter=100, multi_class='ovr',
penalty='l2', random_state=0, solver='liblinear',
tol=0.0001,
verbose=0))]),
'pipeline-1_clf': LogisticRegression(C=0.001, class_weight=None,
dual=False, fit_intercept=True,
intercept_scaling=1, max_iter=100, multi_class='ovr',
penalty='l2', random_state=0, solver='liblinear',
tol=0.0001,
verbose=0),
'pipeline-1_clf_C': 0.001,
'pipeline-1_clf_class_weight': None,
'pipeline-1_clf_dual': False,
[...]
'pipeline-1_sc_with_std': True,
'pipeline-2': Pipeline(steps=[('sc', StandardScaler(copy=True, with_
mean=True, with_std=True)), ('clf', KNeighborsClassifier(algorithm='au
to', leaf_size=30, metric='minkowski',
metric_params=None, n_neighbors=1, p=2,
weights='uniform'))]),
'pipeline-2_clf': KNeighborsClassifier(algorithm='auto', leaf_
size=30, metric='minkowski',
metric_params=None, n_neighbors=1, p=2,
weights='uniform'),
'pipeline-2_clf_algorithm': 'auto',
[...]
'pipeline-2_sc_with_std': True}
```

Na podstawie wartości zwracanych przez metodę `get_params` możemy uzyskiwać dostęp do atrybutów poszczególnych klasyfikatorów. W celach demonstracyjnych skonfigurujemy teraz odwrotny parametr regularizacji C algorytmu regresji logistycznej, a także wysokość drzewa decyzyjnego poprzez przeszukiwanie siatki. Odpowiedzialny jest za to poniższy kod:

```
>>> if Version(sklearn_version) < '0.18':
>>>     from sklearn.cross_validation import GridSearchCV
>>> else:
>>>     from sklearn.model_selection import GridSearchCV
>>> params = {'decisiontreeclassifier__max_depth': [1, 2],
...             'pipeline-1__clf__C': [0.001, 0.1, 100.0]}
>>> grid = GridSearchCV(estimator=mv_clf,
...                      param_grid=params,
...                      cv=10,
...                      scoring='roc_auc')
>>> grid.fit(X_train, y_train)
```

Po zakończeniu przeszukiwania siatki możemy wyświetlić kombinacje różnych wartości hiperparametrów oraz uśrednione wyniki obszaru pod krzywą ROC obliczone dzięki dziesięciokrotнемu sprawdzianowi krzyżowemu. Wykorzystamy do tego poniższy kod:

```
>>> if Version(sklearn_version) < '0.18':
...     for params, mean_score, scores in grid.grid_scores_:
...         print("%0.3f +/- %0.2f %r"
...               % (mean_score, scores.std() / 2.0, params))

>>> else:
...     cv_keys = ('mean_test_score', 'std_test_score','params')

...     for r, _ in enumerate(grid.cv_results_['mean_test_score']):
...         print("%0.3f +/- %0.2f %r"
...               % (grid.cv_results_[cv_keys[0]][r],
...                   grid.cv_results_[cv_keys[1]][r] / 2.0,
...                   grid.cv_results_[cv_keys[2]][r]))
0.967+/-0.05 {'pipeline-1__clf__C': 0.001, 'decisiontreeclassifier__max_depth': 1}
0.967+/-0.05 {'pipeline-1__clf__C': 0.1, 'decisiontreeclassifier__max_depth': 1}
1.000+/-0.00 {'pipeline-1__clf__C': 100.0, 'decisiontreeclassifier__max_depth': 1}
0.967+/-0.05 {'pipeline-1__clf__C': 0.001, 'decisiontreeclassifier__max_depth': 2}
0.967+/-0.05 {'pipeline-1__clf__C': 0.1, 'decisiontreeclassifier__max_depth': 2}
1.000+/-0.00 {'pipeline-1__clf__C': 100.0, 'decisiontreeclassifier__max_depth': 2}
>>> print('Najlepsze parametry: %s' % grid.best_params_)
Najlepsze parametry: {'pipeline-1__clf__C': 100.0,
```

```
'decisiontreeclassifier_max_depth': 1}
>> print('Dokładność: %.2f' % grid.best_score_)
Dokładność: 1.00
```

Jak widać, osiągamy najlepsze rezultaty przy niskiej sile regularyzacji ( $C = 100$ ), z kolei wysokość drzewa zdaje się nie mieć żadnego wpływu na skuteczność modelu, co stanowi sugestię, że granice decyzyjne są odpowiednio dobrane do rozdzielania danych. Przypominam, że nawet dwukrotne wykorzystanie tego samego zestawu danych testowych do oceny modelu jest złym rozwiązaniem, dlatego nie będziemy w tym ustępie szacować skuteczności uogólniania po dostrojeniu hiperparametrów. Zamiast tego przejdziemy od razu do alternatywnej techniki uczenia zespołowego — **agregacji**.

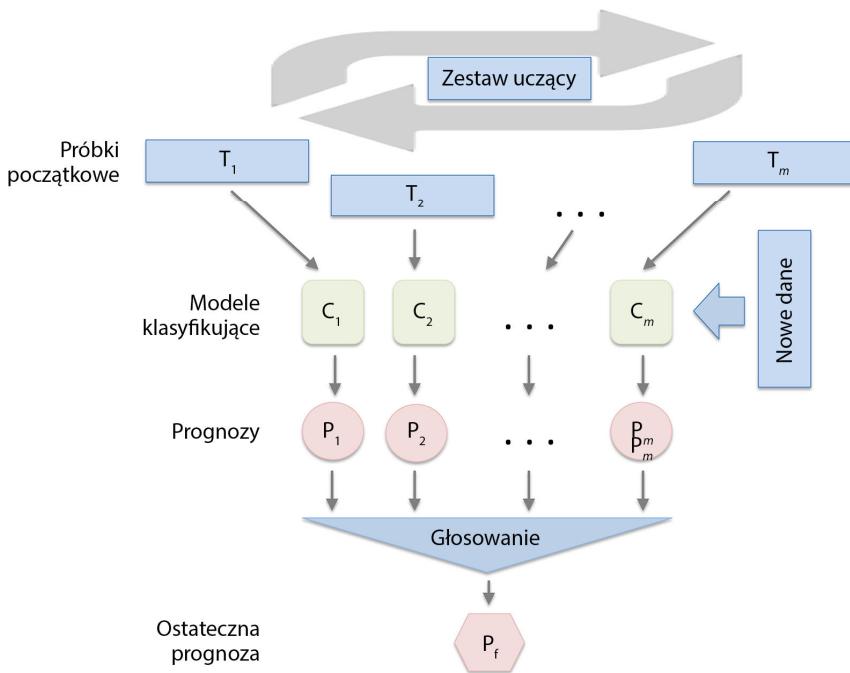
Zaimplementowana przez nas metoda głosowania większościowego zwana jest czasami **kontaminacją modelu** (ang. *stacking*). Jednak algorytm kontaminacyjny przeważnie jest łączony z modelem regresji logistycznej, gdzie służy do przewidywania ostatecznej etykiety klas przy użyciu prognoz poszczególnych klasyfikatorów zespołu jako danych wejściowych, co zostało dokładniej opisane w artykule *Stacked Generalization* autorstwa Davida H. Wolperta („Neural Networks” 1992, nr 5 (2), s. 241 – 259).

## Agregacja — tworzenie zespołu klasyfikatorów za pomocą próbek początkowych

**Agregacja** (ang. *bagging*) stanowi technikę uczenia zespołowego ściśle powiązaną z obiektem `MajorityVoteClassifier` zaimplementowanym w poprzednim podrozdziale, o czym możemy się przekonać, spoglądając na rysunek 7.6.

Nie wykorzystujemy tu jednak tego samego zbioru danych testowych do uczenia poszczególnych składowych zespołu, lecz tworzymy próbki początkowe (ang. *bootstrap samples*; podzbiór losowych próbek ze zwracaniem) z pierwotnego zestawu danych uczących; stąd wzięła się angielska nazwa „*bagging*” — „*bootstrap aggregating*”, czyli w wolnym tłumaczeniu **agregacja próbek wstępnych**. Żeby lepiej zrozumieć zasadę działania agregacji, przeanalizujmy przykład zaprezentowany na rysunku 7.7. Widzimy na nim siedem różnych wystąpień uczących (oznaczonych indeksami od 1 do 7), które w każdej turze agregacji są losowo dobierane ze zwracaniem. Próbki początkowe są następnie używane do trenowania klasyfikatora  $C_j$  — najczęściej nieprzycinanego drzewa decyzyjnego.

Agregacja jest również powiązana z klasyfikatorem losowego lasu, omówionym w rozdziale 3., „*Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn*”. W rzeczywistości losowe lasy stanowią szczególny przypadek agregacji, w którym wykorzystujemy również podzbiory losowych cech do uczenia poszczególnych drzew decyzyjnych. Koncepcja agregacji



Rysunek 7.6. Schemat techniki agregacji modeli

Indeksy próbek	Agregacja: tura 1.	Agregacja: tura 2.	...
1	2	7	...
2	2	3	...
3	1	2	...
4	3	1	...
5	7	1	...
6	2	7	...
7	4	7	...

Przykład działania agregacji modeli:

Przykład działania agregacji modeli:

Przykład działania agregacji modeli:

Rysunek 7.7. Przykład działania agregacji modeli

została zaproponowana przez Leo Breimana w raporcie technicznym z 1994 roku; udowodnił on także, że metoda ta poprawia dokładność niestabilnych modeli i zmniejsza stopień przetrenowania. Jeżeli interesuje Cię technika agregacji, polecam zapoznanie się z artykułem jej twórcy (L. Breiman, *Bagging Predictors*, „Machine Learning” 1996, nr 24 (2), s. 123 – 140), dostępnym bezpłatnie w internecie.

Aby sprawdzić działanie agregacji, stworzmy bardziej skomplikowany problem klasyfikacji przy użyciu zestawu danych **Wine**, z którego korzystaliśmy już w rozdziale 4., „Tworzenie dobrych zbiorów uczących — wstępne przetwarzanie danych”. W tym przypadku interesują nas wyłączenie klasy 2 i 3, a także dobieramy dwie cechy: *Alkohol* i *Odcień*.

```
>>> import pandas as pd
>>> df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/machinelearning-
databases/wine/wine.data', header=None)
>>> df_wine.columns = ['Etykieta klas', 'Alkohol',
...                     'Kwas jabłkowy', 'Popiół',
...                     'Zasadowość popiołu', 'Magnez',
...                     'Całk. zaw. fenoli', 'Flawonoidy',
...                     'Fenole nieflawonoidalowe',
...                     'Proantocyjaniny',
...                     'Intensywność koloru', 'Odcień',
...                     'Transmitancja 280/315 nm',
...                     'Prolina']
>>> df_wine = df_wine[df_wine['Etykieta klas'] != 1]
>>> y = df_wine[['Etykieta klas']].values
>>> X = df_wine[['Alkohol', 'Odcień']].values
```

Zakodujmy teraz etykiety klas w postać binarną i rozzielmy zestaw danych na 60% próbek uczących i 40% próbek testowych:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> if Version(sklearn_version) < '0.18':
>>>     from sklearn.cross_validation import train_test_split
>>> else:
>>>     from sklearn.model_selection import train_test_split
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
>>> X_train, X_test, y_train, y_test = \
...         train_test_split(X, y,
...                           test_size=0.40,
...                           random_state=1)
```

Algorytm *BaggingClassifier* jest już zaimplementowany w bibliotece scikit-learn, dlatego możemy go importować z podmodułu *ensemble*. Naszym bazowym klasyfikatorem będzie tu nieprzycinane drzewo decyzyjne — stworzymy zespół składający się z 500 drzew decyzyjnych trenowanych za pomocą różnych próbek początkowych:

```
>>> from sklearn.ensemble import BaggingClassifier
>>> tree = DecisionTreeClassifier(criterion='entropy',
...                                 max_depth=None,
...                                 random_state=1)
>>> bag = BaggingClassifier(base_estimator=tree,
...                           n_estimators=500,
...                           max_samples=1.0,
...                           max_features=1.0,
...                           bootstrap=True,
```

```
...
bootstrap_features=False,
...
n_jobs=1,
random_state=1)
```

Wyliczymy teraz dokładność przewidywań na podstawie danych uczących i testowych w celu porównania skuteczności klasyfikatora agregacji ze skutecznością pojedynczego, nieprzycinaneego drzewa decyzyjnego:

```
>>> from sklearn.metrics import accuracy_score
>>> tree = tree.fit(X_train, y_train)
>>> y_train_pred = tree.predict(X_train)
>>> y_test_pred = tree.predict(X_test)
>>> tree_train = accuracy_score(y_train, y_train_pred)
>>> tree_test = accuracy_score(y_test, y_test_pred)
>>> print('Dokładność drzewa decyzyjnego dla danych uczących/testowych %.3f/%.3f'
...      % (tree_train, tree_test))
Dokładność drzewa decyzyjnego dla danych uczących/testowych 1.000/0.833
```

Na podstawie uzyskanych wyników dokładności możemy stwierdzić, że nieprzycinane drzewo decyzyjne prawidłowo przewiduje etykiety klas wszystkich elementów zbioru uczącego; jednak znacznie mniejsza skuteczność wobec podzbioru testowego wskazuje na dużą wariancję (przetrenowanie) modelu:

```
>>> bag = bag.fit(X_train, y_train)
>>> y_train_pred = bag.predict(X_train)
>>> y_test_pred = bag.predict(X_test)
>>> bag_train = accuracy_score(y_train, y_train_pred)
>>> bag_test = accuracy_score(y_test, y_test_pred)
>>> print('Dokładność agregacji dla danych uczących/testowych %.3f/%.3f'
...      % (bag_train, bag_test))
Dokładność agregacji dla danych uczących/testowych 1.000/0.896
```

Chociaż dokładność dla danych uczących jest taka sama w przypadku drzewa decyzyjnego i agregacji modeli (wynik 1,0), to widzimy, że klasyfikator agregacji wykazuje nieznacznie większą skuteczność uogólniania wobec zestawu testowego. Porównajmy teraz regiony decyzyjne używane za pomocą obydwu metod:

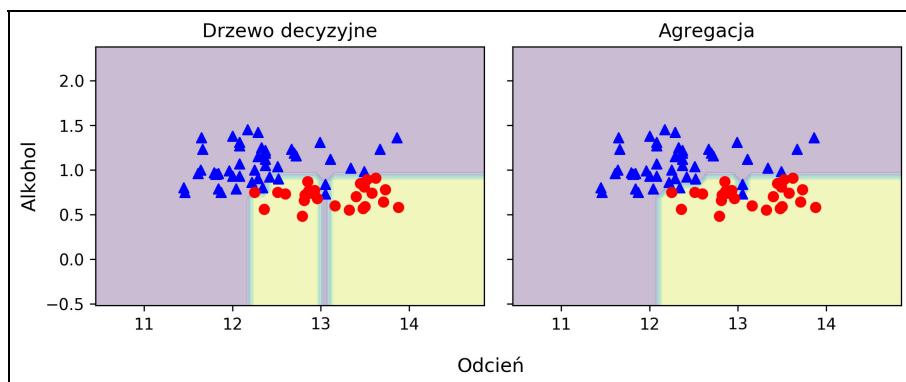
```
>>> x_min = X_train[:, 0].min() - 1
>>> x_max = X_train[:, 0].max() + 1
>>> y_min = X_train[:, 1].min() - 1
>>> y_max = X_train[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                      np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(nrows=1, ncols=2,
...                         sharex='col',
...                         sharey='row',
...                         figsize=(8, 3))
>>> for idx, clf, tt in zip([0, 1],
...                           [tree, bag],
```

```

...             ['Drzewo decyzyjne', 'Agregacja']):
...     clf.fit(X_train, y_train)
...
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx].scatter(X_train[y_train==0, 0],
...                         X_train[y_train==0, 1],
...                         c='blue', marker='^')
...     axarr[idx].scatter(X_train[y_train==1, 0],
...                         X_train[y_train==1, 1],
...                         c='red', marker='o')
...     axarr[idx].set_title(tt)
>>> axarr[0].set_ylabel('Alkohol', fontsize=12)
>>> plt.text(10.2, -1.2,
...            s='Odcień',
...            ha='center', va='center', fontsize=12)
>>> plt.show()

```

Jak widzimy na rysunku 7.8, rozdzielone granice decyzyjne trójwzglęwowego drzewa decyzyjnego nabierają jednolitego kształtu w przypadku agregacji.



Rysunek 7.8. Porównanie granic decyzyjnych wygenerowanych przez nieprzycinane drzewo decyzyjne i agregację modeli

W tym ustępie przeanalizowaliśmy zaledwie bardzo prosty przykład agregacji. W praktyce bardziej złożone zadania klasyfikacji oraz wielowymiarowość zestawów danych mogą z łatwością doprowadzić do przetrenowania pojedynczego drzewa decyzyjnego — właśnie wtedy technika agregacji okazuje się najbardziej przydatna. Na koniec warto zauważyć, że algorytm agregacji może skutecznie zredukować wariancję modelu. Jednak nie nadaje się on do zmniejszania obciążenia, dlatego dobieramy w tym celu zespoły klasyfikatorów cechujących się małym obciążeniem, takich jak właśnie nieprzycinane drzewa decyzyjne.

# Usprawnianie słabych klasyfikatorów za pomocą wzmacnienia adaptacyjnego

Przyjrzymy się teraz technice wzmacniania (ang. *boosting*) ze szczególnym uwzględnieniem jej najpowszechniejszej implementacji — AdaBoost (skrót od wyrażenia „ADAptive BOOS-Ting”, czyli **wzmocnienie adaptacyjne**).

Pomysłodawcą techniki AdaBoost jest Robert E. Schapire (*The Strength of Weak Learnability*, „Machine Learning” 1990, nr 5 (2), s. 197 – 227). Po zaprezentowaniu algorytmu przez Roberta E. Schapire'a i Yoava Freunda na 13. Konferencji IMCL w 1996 roku stał się on jedną z najbardziej rozpowszechnionych metod zespołowych (Y. Freund, R.E. Schapire i in., *Experiments with a New Boosting Algorithm [w:] ICML*, t. 96, 1996, s. 148 – 156). W 2003 roku Schapire i Freund otrzymali Nagrodę Goedla — prestiżowe wyróżnienie dla najznakomitszych publikacji w dziedzinie informatyki teoretycznej — za swoje przełomowe badania.

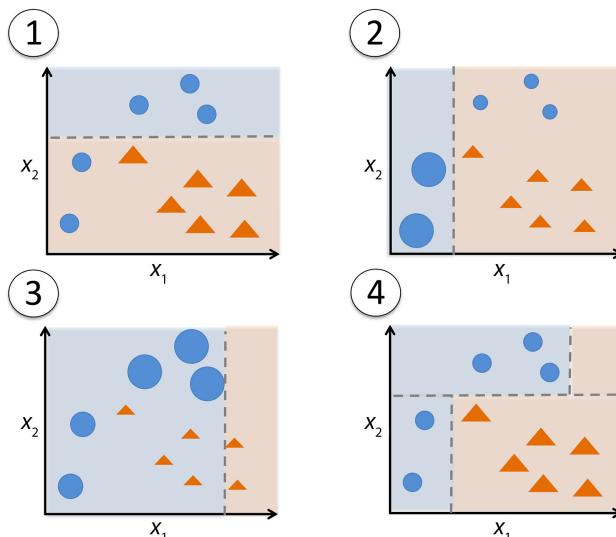
W przypadku wzmacniania zespół składa się z bardzo prostych klasyfikatorów, często nazywanych **słabyimi klasyfikatorami** (ang. *weak learner*), których skuteczność przewidywania jest tylko nieznacznie lepsza od losowego zgadywania. Typowy przykład słabego klasyfikatora stanowi pień drzewa decyzyjnego. Podstawową koncepcją wzmacniania jest koncentracja na trudnych do klasyfikowania próbkach uczących w celu poprawy skuteczności całego zespołu. W przeciwieństwie do agregacji algorytm wzmacnienia wykorzystuje losowe podzbiory danych uczących, które są pobierane z zestawu uczącego bez zwracania. Pierwotną procedurę wzmacniania możemy podzielić na cztery zasadnicze etapy:

1. Stworzenie losowego podzbioru próbek uczących  $d_1$  bez zwracania danych ze zbioru uczącego  $D$  i uczenie słabego klasyfikatora  $C_1$ .
2. Stworzenie drugiego losowego podzbioru  $d_2$  bez zwracania danych z zestawu uczącego i dodanie 50% nieprawidłowo sklasyfikowanych próbek w kroku 1. w celu trenowania słabego klasyfikatora  $C_2$ .
3. Określenie podzbioru próbek uczących  $d_3$  w zestawie treningowym  $D$ , wobec których klasyfikatory  $C_1$  i  $C_2$  są nieskuteczne, i trenowanie trzeciego słabego klasyfikatora —  $C_3$ .
4. Połączenie klasyfikatorów:  $C_1$ ,  $C_2$  i  $C_3$  za pomocą głosowania większościowego.

Leo Breiman stwierdził (L. Breiman, *Bias, Variance and Arcing Classifiers*, 1996), że wzmocnienie może prowadzić do większej redukcji zarówno obciążenia, jak i wariancji niż w przypadku agregacji modeli. W praktyce jednak takie algorytmy wzmacniania jak AdaBoost są również znane z dużej wariancji, tj. tendencji do nadmiernego dopasowania do danych uczących (G. Raetsch, T. Onoda i K.R. Mueller, *An Improvement of Adaboost to Avoid Overfitting*, Międzynarodowa Konferencja Neuronowego Przetwarzania Informacji<sup>1</sup>, Citesser, 1998).

<sup>1</sup> Proceedings of the International Conference on Neural Information Processing — przyp. tłum.

W przeciwieństwie do klasycznej procedury wzmacniania algorytm AdaBoost wykorzystuje pełny zestaw danych uczących do trenowania słabego klasyfikatora; po każdej iteracji wagi próbek uczących są modyfikowane w celu utworzenia skutecznego klasyfikatora uczącego się na pomyłkach zespołu słabych klasyfikatorów. Zanim przejdziemy do analizy szczegółów algorytmu AdaBoost, spójrzmy na rysunek 7.9, aby lepiej zrozumieć podstawowy mechanizm jego działania.



Rysunek 7.9. Schemat działania algorytmu AdaBoost

Przeanalizujmy rysunek 7.9 krok po kroku. Na wykresie 1. widzimy zestaw uczący dla klasyfikacji binarnej — wszystkie próbki mają tu jednakowe wagi. Na podstawie tych próbek uczących generowana jest granica decyzyjna (zaznaczona przerywaną linią) służąca do klasyfikowania próbek do dwóch klas (trójkątów i kółek), jak również, w miarę możliwości, do minimalizowania funkcji kosztu (w przypadku zespołów drzew decyzyjnych — zanieczyszczenia). W kolejnej turze (wykres 2.) wyznaczamy większe wagi dwóm uprzednio niewłaściwie sklasyfikowanym próbkom (kółkom). Do tego obniżamy wagi prawidłowo wyznaczonych próbek. Kolejna granica decyzyjna będzie teraz bardziej skoncentrowana na próbках uczących o największych wagach, tj. danych najtrudniejszych do sklasyfikowania. Słaby klasyfikator ukazany na wykresie 2. nieprawidłowo klasyfikuje trzy różne próbki z klasy oznaczonej kółkami, co oznacza, że w kolejnej iteracji (wykres 3.) uzyskują nowe, większe wagi. Przy założeniu, że omawiany proces wzmacniania składa się jedynie z trzech tur, przychodzi w końcu czas na połączenie trzech słabych klasyfikatorów wytrenowanych na trzech podzbiorach uczących o różnych wagach i klasyfikację wykorzystującą głosowanie większościowe, co widzimy na wykresie 4.

Gdy już wiemy, na czym polega podstawa działania algorytmu AdaBoost, możemy dokładniej go przeanalizować z pomocą pseudokodu. Dla przejrzystości mnożenie elementów będziemy oznaczać krzyżykiem ( $\times$ ), a iloczyn skalarny dwóch wektorów kropką ( $\cdot$ ). Poszczególne etapy algorytmu wyglądają następująco:

1. Wyznacz wektor  $\mathbf{w}$  zawierający jednakowe wagi, gdzie  $\sum_i w_i = 1$ .
2. W  $j$ -tej turze z  $m$  iteracji wykonaj następujące czynności:
3. Wyucz ważony, słaby klasyfikator:  $C_j = \text{ucz}(\mathbf{X}, \mathbf{y}, \mathbf{w})$ .
4. Prognozuj etykiety klas:  $\hat{\mathbf{y}} = \text{prognozuj}(C_j, \mathbf{X})$ .
5. Oblicz ważoną stopę błędu:  $\varepsilon = \mathbf{w} \cdot (\hat{\mathbf{y}} \neq \mathbf{y})$ .
6. Wylicz współczynnik:  $\alpha_j = 0,5 \log \frac{1-\varepsilon}{\varepsilon}$ .
7. Zaktualizuj wagi:  $\mathbf{w} := \mathbf{w} \times \exp(-\alpha_j \times \hat{\mathbf{y}} \times \mathbf{y})$ .
8. Znormalizuj wagi tak, aby ich suma dawała wartość 1:  $\mathbf{w} := \mathbf{w} / \sum_i w_i$ .
9. Oblicz ostateczną prognozę:  $\hat{\mathbf{y}} = (\sum_{j=1}^m (\alpha_j \times \text{prognozuj}(C_j, \mathbf{X})) > 0)$ .

Zwrócić uwagę, że wyrażenie  $(\hat{\mathbf{y}} \neq \mathbf{y})$  w etapie 5. dotyczy wektora składającego się z zer i jedynek, w którym wartość 1 jest przydzielana w przypadku nieprawidłowej prognozy, a wartość 0 — prawidłowej.

Pomimo że algorytm AdaBoost nie wydaje się zbyt skomplikowany, przeanalizujmy jakiś przykład, w którym zestaw uczący składa się z 10 próbek zaprezentowanych w tabeli 7.1.

**Tabela 7.1.** Zestaw danych uczących wykorzystanych w przykładzie wyjaśniającym działanie algorytmu AdaBoost

Indeksy próbek	x	y	Wagi	$\hat{y}(x=3,0)?$	Poprawnie?	Zaktualizowane wagi
1	1,0	1	0,1	1	Tak	0,072
2	2,0	1	0,1	1	Tak	0,072
3	3,0	1	0,1	1	Tak	0,072
4	4,0	-1	0,1	-1	Tak	0,072
5	5,0	-1	0,1	-1	Tak	0,072
6	6,0	-1	0,1	-1	Tak	0,072
7	7,0	1	0,1	-1	Nie	0,167
8	8,0	1	0,1	-1	Nie	0,167
9	9,0	1	0,1	-1	Nie	0,167
10	10,0	-1	0,1	-1	Tak	0,072

W pierwszej kolumnie znajdują się indeksy próbek uczących. Druga kolumna zawiera wartości cech poszczególnych próbek przy założeniu, że mamy do czynienia z jednowymiarowym zbiorem danych. Trzecia kolumna przedstawia prawdziwe etykiety klas  $y_i$  dla każdej próbki uczącej  $x_i$ , gdzie  $y_i \in \{1, -1\}$ . Początkowe wagi są widoczne w czwartej kolumnie; wszystkie mają jednakowe wartości i są tak znormalizowane, że ich suma daje wartość 1. Z tego powodu w wektorze wag  $\mathbf{w}$  każda waga  $w_i$  w 10 próbkach jest równa 0,1. Przewidywane etykiety klas  $\hat{\mathbf{y}}$

umieszczono w kolumnie piątej; zakładamy przy okazji, że naszym kryterium podziału na klasy jest  $x \leq 3,0$ . W ostatniej kolumnie natomiast widzimy zaktualizowane wagи wyliczone na podstawie reguł zdefiniowanych w pseudokodzie.

Obliczenie zaktualizowanych wag może początkowo wydawać się dość złożoną czynnością, dlatego przeprowadzmy ten proces krok po kroku. Rozpoczniemy od wyliczenia ważonej stopy błędu (tak jak zostało opisane w etapie 5. pseudokodu):

$$\begin{aligned}\varepsilon &= 0,1 \times 0 + 0,1 \times 1 + 0,1 \times 1 + \\ &+ 0,1 \times 1 + 0,1 \times 0 = \frac{3}{10} = 0,3\end{aligned}$$

Wyliczamy teraz współczynnik  $\alpha_j$  (etap 6.), który zostaje wykorzystany w kroku 7. przy aktualizacji wag, jak również podczas określania ostatecznego wyniku za pomocą głosowania większościowego:

$$\alpha_j = 0,5 \log\left(\frac{1-\varepsilon}{\varepsilon}\right) \approx 0,424$$

Po wyliczeniu współczynnika  $\alpha_j$  możemy przejść do aktualizacji wektora wag za pomocą następującego wzoru:

$$\mathbf{w} := \mathbf{w} \times \exp(-\alpha_j \times \hat{\mathbf{y}} \times \mathbf{y})$$

Wyrażenie  $\hat{\mathbf{y}} \times \mathbf{y}$  oznacza tu mnożenie elementów pomiędzy wektorami etykiet prognozowanych i rzeczywistych. Zatem jeśli prognoza  $\hat{y}_i$  jest poprawna, iloczyn  $\hat{y}_i \times y_i$  będzie dodatni, dzięki czemu będziemy w stanie zmniejszyć  $i$ -tą wagę, gdyż  $\alpha_j$  ma również wartość dodatnią:

$$0,1 \times \exp(-0,424 \times 1 \times 1) \approx 0,065$$

W analogiczny sposób zmodyfikujemy  $i$ -tą wagę, jeżeli prognoza  $\hat{y}_i$  okazuje się nieprawidłowa:

$$0,1 \times \exp(-0,424 \times 1 \times (-1)) \approx 0,153$$

Albo:

$$0,1 \times \exp(-0,424 \times (-1) \times 1) \approx 0,153$$

Po zaktualizowaniu każdej wagi w wektorze wag przeprowadzamy ich normalizację, dzięki czemu ich suma jest równa 1 (etap 8.):

$$\mathbf{w} := \frac{\mathbf{w}}{\sum_i w_i}$$

W naszym przykładzie  $\sum_i w_i = 7 \times 0,065 + 3 \times 0,153 = 0,914$ .

Zatem każda waga przydzielona prawidłowo sklasyfikowanej próbce zostanie zmniejszona z początkowej wartości 0,1 o  $0,065/0,914 \approx 0,071$  w następnej iteracji. Z kolei waga nieprawidłowo sklasyfikowanej próbki zostanie zwiększona o  $0,153/0,914 \approx 0,167$ .

Tak wygląda algorytm AdaBoost w pigułce. Przejedźmy do części praktycznej — uczenia klasifikatora zespołowego AdaBoost za pomocą interfejsu scikit-learn. Wykorzystamy w tym celu znany nam już zestaw danych Wine. Dzięki atrybutowi `base_estimator` wytrenujemy obiekt `AdaBoostClassifier` zawierający 500 drzew decyzyjnych:

```
>>> from sklearn.ensemble import AdaBoostClassifier
>>> tree = DecisionTreeClassifier(criterion='entropy',
...                                 max_depth=1,
...                                 random_state=0)
>>> ada = AdaBoostClassifier(base_estimator=tree,
...                            n_estimators=500,
...                            learning_rate=0.1,
...                            random_state=0)
>>> tree = tree.fit(X_train, y_train)
>>> y_train_pred = tree.predict(X_train)
>>> y_test_pred = tree.predict(X_test)
>>> tree_train = accuracy_score(y_train, y_train_pred)
>>> tree_test = accuracy_score(y_test, y_test_pred)
>>> print('Dokładność drzewa decyzyjnego dla danych uczących/testowych
%.3f/.3f'
...      % (tree_train, tree_test))
Dokładność drzewa decyzyjnego dla danych uczących/testowych 0.845/0.854
```

Jak widać, przycinane drzewo decyzyjne wykazuje zbyt małe dopasowanie do danych uczących — w przeciwieństwie do algorytmu nieprzycinanego drzewa.

```
>>> ada = AdaBoostClassifier()
>>> ada.fit(X_train, y_train)
>>> y_train_pred = ada.predict(X_train)
>>> y_test_pred = ada.predict(X_test)
>>> ada_train = accuracy_score(y_train, y_train_pred)
>>> ada_test = accuracy_score(y_test, y_test_pred)
>>> print('Dokładność algorytmu AdaBoost dla danych uczących/testowych
%.3f/.3f'
...      % (ada_train, ada_test))
Dokładność algorytmu AdaBoost dla danych uczących/testowych 1.000/0.875
```

Z kolei tutaj widzimy, że model AdaBoost poprawnie klasyfikuje wszystkie próbki uczące, a także wykazuje nieco większą niż algorytm drzewa decyzyjnego skuteczność w przypadku danych testowych. Możemy jednak również dostrzec, że próbując zmniejszyć obciążenie modelu, jednocześnie zwiększyliśmy jego wariancję.

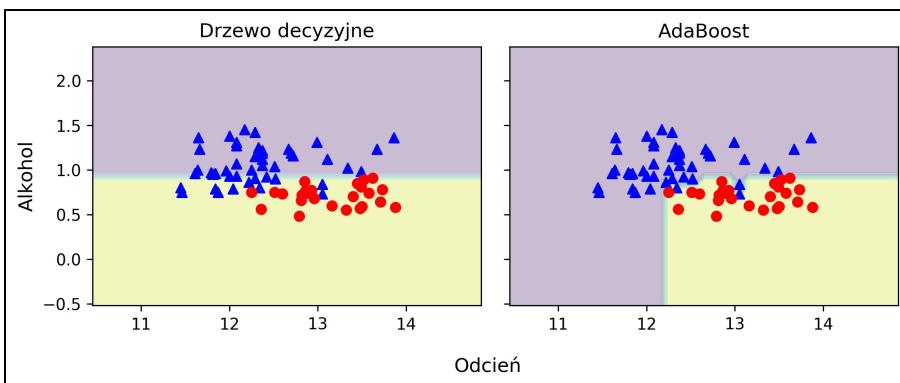
Mimo że znowu wykorzystaliśmy prosty przykład w celach demonstracyjnych, możemy stwierdzić, że klasifikator AdaBoost ma nieco lepszą skuteczność od algorytmu drzewa decyzyjnego i dokładność zbliżoną do omawianego w poprzednim podrozdziale modelu agregacji. Nie możemy

jednak zapominać, że dobór modelu na podstawie wielokrotnego wykorzystywania tego samego zestawu testowego jest złym rozwiązańiem. Oszacowanie skuteczności uogólniania może być natyżby optymistyczne, co dokładniej wyjaśnione w rozdziale 6., „Najlepsze metody oceny modelu i strojenie parametryczne”.

Przyjrzyjmy się w końcu regionom decyzyjnym generowanym przez model AdaBoost:

```
>>> x_min = X_train[:, 0].min() - 1
>>> x_max = X_train[:, 0].max() + 1
>>> y_min = X_train[:, 1].min() - 1
>>> y_max = X_train[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                         np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(1, 2,
...                         sharex='col',
...                         sharey='row',
...                         figsize=(8, 3))
>>> for idx, clf, tt in zip([0, 1],
...                           [tree, ada],
...                           ['Drzewo decyzyjne', 'AdaBoost']):
...     clf.fit(X_train, y_train)
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx].scatter(X_train[y_train==0, 0],
...                        X_train[y_train==0, 1],
...                        c='blue',
...                        marker='^')
...     axarr[idx].scatter(X_train[y_train==1, 0],
...                        X_train[y_train==1, 1],
...                        c='red',
...                        marker='o')
...     axarr[idx].set_title(tt)
...     axarr[0].set_ylabel('Alkohol', fontsize=12)
>>> plt.text(10.2, -1.2,
...            s='Odcień',
...            ha='center',
...            va='center',
...            fontsize=12)
>>> plt.show()
```

Przeglądając się regionom decyzyjnym ukazanym na rysunku 7.10, możemy stwierdzić, że granice wyznaczane przez model AdaBoost są znacznie bardziej skomplikowane niż granice generowane przez drzewo decyzyjne. Do tego łatwo zauważać, że algorytm ten rozdziela przestrzeń cech w bardzo podobny sposób do modelu agregacji omówionego w poprzednim podrozdziale.



Rysunek 7.10. Porównanie regionów decyzyjnych generowanych przez przyjęte drzewo decyzyjne i algorytm AdaBoost

Na koniec naszej opowieści o technikach zespołowych warto zwrócić uwagę, że uczenie tego typu zwiększa złożoność obliczeniową w porównaniu z pojedynczymi klasyfikatorami. W praktyce musimy za każdym razem dobrze przemyśleć, czy warto znacznie obciążać moc obliczeniową kosztem skromnego (często) przyrostu skuteczności predykcyjnej.

Nieraz przytaczanym przykładem takiego kompromisu jest słynna **miliondolarowa nagroda Netflixsa**, którą laureaci zdobyli właśnie dzięki algorytmom zespołowych. Szczegóły tego algorytmu zostały opisane w eseju autorstwa Andreasa Toeschera, Michaela Jahrera i Roberta M. Bella (*The Bigchaos Solution to the Netflix Grand Prize*, dokumentacja Netflixsa na temat nagrody, 2009), dostępnym pod adresem [http://www.stat.osu.edu/~dmsl/GrandPrize2009\\_BPC\\_BigChaos.pdf](http://www.stat.osu.edu/~dmsl/GrandPrize2009_BPC_BigChaos.pdf). Chociaż twórcy algorytmu otrzymali główną nagrodę, firma Netflix nigdy go nie zaimplementowała z powodu złożoności wręcz uniemożliwiającej jego działanie na rzeczywistych danych. Cytując dokładnie słowa pracowników serwisu (<http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html>):

„[...] zmierzzone przez nas zyski dokładności nie usprawiedliwiały wysiłku inżynierijnego wymaganego do wdrożenia ich w środowisko produkcyjne”.

## Podsumowanie

W tym rozdziale przyjrzaliśmy się niektórym z najpopularniejszych i najczęściej stosowanych technik uczenia zespołowego. Dzięki metodom zespołowym łączymy różne modele klasyfikujące, co pozwala niwelować wady poszczególnych składowych, a w rezultacie nieraz używać stabilne i skuteczne modele, które są bardzo atrakcyjne w zastosowaniach przemysłowych, jak również w różnego rodzaju konkursach.

Na początku rozdziału zaimplementowaliśmy w Pythonie obiekt `MajorityVoteClassifier`, służący do łączenia różnych algorytmów klasyfikujących. Następnie przeanalizowaliśmy technikę agregacji modeli, zmniejszającą wariancję modelu poprzez losowanie próbek początkowych z zestawu uczącego oraz łączącą wyniki poszczególnych klasyfikatorów składowych poprzez głosowanie większościowe. Na koniec dotarliśmy do algorytmu AdaBoost, bazującego na słabych klasyfikatorach w znaczący sposób uczących się na własnych błędach.

Do tej pory zajmowaliśmy się opisem różnych algorytmów uczenia maszynowego, a także technik ich strojenia i oceny. W kolejnym rozdziale weźmiemy na celownik specyficzne zastosowanie uczenia maszynowego — analizę sentymentów, która zyskała na znaczeniu i popularności w erze internetu i mediów społecznościowych.



# Wykorzystywanie uczenia maszynowego w analizie sentymen-tów

W erze internetu i mediów społecznościowych opinie, recenzje i rekomendacje stanowią cenne źródło informacji dla nauk politycznych i biznesu. Dzięki współczesnym technologiom jesteśmy w stanie niezwykle skutecznie gromadzić i analizować te dane. W niniejszym rozdziale skoncentrujemy się na **przetwarzaniu języka naturalnego** (ang. *natural language processing* — NLP), zwanym **analizą sentymen-tów**, oraz dowiemy się, w jaki sposób używać uczenia maszynowego do klasyfikowania tekstu na podstawie jego bieguności: nastawienia autora. W kolejnych podrozdziałach zajmiemy się następującymi zagadnieniami:

- oczyszczanie i przygotowywanie danych tekstowych,
- tworzenie wektorów cech z dokumentów tekstowych,
- trenowanie modelu uczenia maszynowego w celu klasyfikowania pozytywnych i negatywnych recenzji filmów,
- praca z dużymi zbiorami danych tekstowych przy użyciu uczenia pozardzeniowego.

## Zestaw danych IMDb movie review

Analiza sentymen-tów (ang. *sentiment analysis*), zwana czasami także **rozpoznawaniem opinii** (ang. *opinion mining*), stanowi popularny dział przetwarzania języka naturalnego; jej celem jest analizowanie **biegunowości** tekstu. Często sprowadza się to do klasyfikowania danych tekstowych na podstawie wyrażonych w nich opinii lub emocji autora dotyczących określonego tematu.

W niniejszym rozdziale będziemy pracować na bardzo dużym zestawie recenzji filmowych pochodzących z serwisu **IMDb** (ang. *Internet Movie Database* — internetowa filmowa baza danych) zebranych przez Andrew L. Maasa i in. (A.L. Maas, R.E. Daly, P.T. Pham, D. Huang, A.Y. Ng i C. Potts, *Learning Word Vectors for Sentiment Analysis*, 49. doroczne spotkanie stowarzyszenia ACL (Association for Computational Linguistics): „Human Language Technologies”, Portland, Oregon, USA, czerwiec 2011, s. 142 – 150). Ten zbiór recenzji filmowych składa się z 50 000 bieżących recenzji oznaczonych jako **pozytywne** albo **negatywne**; w tym przypadku poprzez „pozytywne” rozumiemy filmy, które otrzymały w serwisie IMDb przynajmniej sześć gwiazdek w dziesięciostopniowej skali, natomiast „negatywne” dotyczą recenzji przyznających danemu tytułu co najwyżej pięć gwiazdek. W kolejnych podrozdziałach nauczymy się z tego zestawu recenzji wydobywać przydatne informacje, które posłużą nam do utworzenia modelu uczenia maszynowego prognozującego, czy danemu autorowi film się podobał lub nie.

Skompresowane archiwum zestawu recenzji (80,2 MB) jest dostępne pod adresem <http://ai.stanford.edu/~amaas/data/sentiment/> w postaci pliku .tar:

- Jeżeli pracujesz na systemie Linux lub Mac OS X, otwórz nowe okno terminala, za pomocą polecenia cd przejdź do katalogu *download* i wpisz komendę tar -zxf aclImdb\_v1.tar.gz, aby rozpakować archiwum zestawu recenzji.
- Użytkownicy systemu Windows mogą pobrać bezpłatny archiwizator (np. 7-Zip — <http://www.7-zip.org/>), służący do wypakowania zawartości archiwum.

Po rozpakowaniu archiwum przejdziemy do połączenia poszczególnych danych tekstowych w pojedynczy plik CSV. Za pomocą poniższego fragmentu kodu będziemy wczytywać recenzje filmów do obiektu DataFrame (część biblioteki pandas), co na standardowym komputerze stacjonarnym zajmuje ok. 10 minut. W celu śledzenia postępów oraz szacowanego czasu ukończenia procesu wykorzystamy pakiet **PyPrind** (<https://pypi.python.org/pypi/PyPrind/>), który kilka lat temu stworzyłem właśnie do tego typu zadań. Bibliotekę PyPrind zainstalujemy za pomocą polecenia pip install pyprind.

```
>>> import pyprind
>>> import pandas as pd
>>> import os
>>> pbar = pyprind.ProgBar(50000)
>>> labels = {'pos':1, 'neg':0}
>>> df = pd.DataFrame()
>>> for s in ('test', 'train'):
...     for l in ('pos', 'neg'):
...         path ='./aclImdb/%s/%s' % (s, l)
...         for file in os.listdir(path):
...             with open(os.path.join(path, file), 'r') as infile:
...                 txt = infile.read()
...                 df = df.append([[txt, labels[l]]], ignore_index=True)
...                 pbar.update()
>>> df.columns = ['Recenzja', 'Sentyment']
0%                                         100%
[#####] | ETA[sec]: 0.000
Total time elapsed: 725.001 sec
```

Z pomocą powyższego kodu najpierw zainicjalizujemy nowy pasek postępu `pbar` mający 50 000 iteracji, co jest równe liczbie dokumentów tekstowych, które wczytujemy. Poprzez zagnieżdżone pętle `for` algorytm sprawdzał podkatalogi `train` i `test` w głównym katalogu `aclImdb` i wczytywał poszczególne pliki tekstowe z podkatalogów `pos` i `neg`, które ostatecznie dołączymy do obiektu `DataFrame` — wraz z etykietami klas (1 = pozytywna, 0 = negatywna).

Etykiety klas w tym zestawie danych są uporządkowane, dlatego teraz je przetasujemy za pomocą funkcji `permutation` z podmodułu `np.random` — przyda nam się to podczas rozdzielania zbioru danych na próbki uczące i testowe w dalszej części rozdziału (gdy będziemy przesyłać strumieniowo dane bezpośrednio z lokalnego nośnika danych). Dla własnej wygody zapiszemy również zebrane i przetasowane recenzje filmów w pliku CSV:

```
>>> import numpy as np
>>> np.random.seed(0)
>>> df = df.reindex(np.random.permutation(df.index))
>>> df.to_csv('./filmy_dane.csv', index=False)
```

Będziemy korzystać z tych danych w dalszej części rozdziału, dlatego sprawdźmy szybko, czy udało nam się zachować je we właściwym formacie poprzez odczytanie pliku CSV i wyświetlenie fragmentu trzech pierwszych próbek:

```
>>> df = pd.read_csv('./filmy_dane.csv')
>>> df.head(3)
```

Jeżeli korzystasz z plików kodu źródłowego w aplikacji Jupyter Notebook, Twoim oczom powinny ukazać się teraz trzy pierwsze próbki z zestawu danych, zaprezentowane w tabeli 8.1.

**Tabela 8.1.** Pierwsze trzy próbki zestawu danych IMDb movie review

	Recenzja	Sentyment
0	In 1974, the teenager Martha Moxley (Maggie Gr...	1
1	OK... so... I really like Kris Kristofferson a...	0
2	***SPOILER*** Do not read this, if you think a...	0

## Wprowadzenie do modelu worka słów

Pamiętamy z rozdziału 4., „Tworzenie dobrych zbiorów uczących — wstępne przetwarzanie danych”, że przed przekazaniem danych kategoryzujących, takich jak tekst lub słowa, do algorytmu uczenia maszynowego, musimy je przekształcić do postaci numerycznej. W tym podrozdziale przeanalizujemy model **worka słów** (ang. *bag-of-words model*) modyfikujący tekst w formę numerycznych wektorów cech. Mechanizm działania tego algorytmu nie jest zbyt skomplikowany i można go podzielić na następujące etapy:

1. Tworzymy słownik unikatowych znaczników (tokenów) — np. wyrazów — z całego zbioru danych tekstowych.

- Konstruujemy wektor cech z każdego tekstu zawierającego zliczone wystąpienia poszczególnych wyrazów znajdujących się w danym dokumencie tekstowym.

Unikatowe wyrazy w każdym dokumencie stanowią jedynie niewielki podzbiór wszystkich elementów słownika modelu worka słów, dlatego wektory cech będą składać się głównie z zer (stąd nazywamy je **rzadkimi**). Nie martw się, jeśli brzmi to dość abstrakcyjnie; w dalszej części rozdziału zaprezentuję krok po kroku proces tworzenia prostego modelu worka słów.

## Przekształcanie słów w wektory cech

Aby stworzyć model worka słów na podstawie wystąpień wyrazów w poszczególnych danych tekstowych, możemy wykorzystać klasę CountVectorizer zaimplementowaną w interfejsie scikit-learn. Jak się za chwilę przekonamy, klasa ta przyjmuje tablicę danych tekstowych (np. dokumenty albo pojedyncze zdania) i tworzy za nas model worka słów:

```
>>> import numpy as np
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> count = CountVectorizer()
>>> docs = np.array([
...     'Słońce jest dziś promienne',
...     'Pogoda jest dziś wiosenna',
...     'Słońce jest dziś promienne i pogoda jest dziś wiosenna',
...     'a jeden i jeden to jest dwa'])
>>> bag = count.fit_transform(docs)
```

Z pomocą metody `fit_transform` w klasie CountVectorizer stworzyliśmy właśnie słownik modelu worka słów i przekształciliśmy poniższe zdania w rzadki wektor cech:

- Słońce grzeje dziś mocno
- Pogoda jest dziś wiosenna
- Słońce grzeje dziś mocno i pogoda jest dziś wiosenna, a jeden i jeden daje dwa

Wyświetlmy teraz zawartość słownika w celu lepszego zrozumienia działania omawianego modelu:

```
>>> print(count.vocabulary_)
{'słońce': 6, 'jest': 3, 'dziś': 1, 'promienne': 5, 'pogoda': 4, 'wiosenna':
 8, 'jeden': 2, 'to': 7, 'dwa': 0}
```

Jak widać powyżej, wyrazy są przechowywane w słowniku Pythona, w którym poszczególne wyrazy otrzymują indeksy liczbowe. Spójrzmy na wygenerowane przez nas wektory cech:

```
>>> print(bag.toarray())
[[0 1 0 1 0 1 1 0 0]
 [0 1 0 1 1 0 0 0 1]
 [1 2 2 3 1 1 1 1 1]]
```

Pozycja każdego indeksu w ukazanych wektorach cech odpowiada liczbom całkowitym przechowywanym jako elementy słownika CountVectorizer; np. pierwsza cecha o indeksie 0 reprezentuje zliczenia wyrazu „dwa”, który pojawia się wyłącznie w ostatnim zdaniu, natomiast słowo „jest” na czwartej pozycji (indeks 3 w wektorach cech) występuje we wszystkich trzech zdaniach. Wartości przechowywane w wektorach cech są również nazywane **częstością termów** (ang. *raw term frequency*):  $tf(t, d)$  — liczbą wystąpień termu  $t$  w dokumencie  $d$ .

Kolejność występowania elementów w utworzonym przez nas modelu worka słów nosi miano modelu **unigramowego (1-gramowego)** — każdy element lub znacznik w słowniku reprezentuje pojedynczy wyraz. Uogólniając, sąsiadujące ze sobą elementy w modelach NLP — wyrazy, litery lub symbole — są również nazywane **n-gramami**. Dobór wartości  $n$  w modelu n-gramów zależy od tego, co chcemy osiągnąć; np. Ioannis Kanaris i in. dowiedli, że n-gramy o rozmiarze 3 lub 4 dają dobre wyniki w przypadku filtrowania spamu poczty e-mail (I. Kanaris, K. Kanaris, I. Houvardas i E. Stamatos, *Words vs Character N-Grams for Anti-Spam Filtering*, „International Journal on Artificial Intelligence Tools” 2007, nr 16 (6), s. 1047 – 1067). Podsumujmy reprezentację n-gramów: 1-gramowego i 2-gramowego na przykładzie naszego pierwszego tekstu („Słońce jest dziś promienne”):

- Model **1-gramowy**: „Słońce”, „jest”, „dziś”, „promienne”.
- Model **2-gramowy**: „Słońce jest”, „jest dziś”, „dziś promienne”.

Klasa CountVectorizer w bibliotece scikit-learn umożliwia dobór różnych modeli n-gramów za pomocą parametru `ngram_range`. Domyślnie jest skonfigurowana reprezentacja unigramowa, ale możemy ją zmienić na reprezentację 2-gramową, inicjując nowe wystąpienie obiektu CountVectorizer z parametrem `ngram_range=(2, 2)`.

## Ocena istotności wyrazów za pomocą ważenia częstotliwości termów — odwrotnej częstotliwości w tekście

W czasie analizowania danych tekstowych często natrafiamy na wyrazy z obydwu klas pojawiające się w wielu różnych danych tekstowych. Takie często występujące słowa zazwyczaj nie zawierają żadnych przydatnych ani rozróżniających informacji. W tym ustępie poznamy użyteczną technikę zwaną **ważeniem częstotliwości termów — odwrotną częstotliwością w tekście** (ang. *term frequency — inverse document frequency — tf-idf*), za pomocą której zmniejszamy wagę takich mniej istotnych wyrazów w wektorach cech. Parametr tf-idf możemy zdefiniować jako iloczyn **częstotliwości termów** przez **odwrotną częstotliwość w tekście**:

$$tf\text{-}idf(t,d) = tf(t,d) \times idf(t,d)$$

Tutaj  $tf(t, d)$  jest wprowadzoną w poprzednim ustępie częstotliwością termów, natomiast odwrotną częstotliwość w tekście  $idf(t, d)$  definiujemy następująco:

$$idf(t,d) = \log \frac{n_d}{1 + df(d,t)}$$

gdzie  $n_d$  oznacza całkowitą liczbę danych tekstowych, a  $df(d, t)$  — liczbę danych tekstowych  $d$  zawierających term  $t$ . Zwrót uwagę, że wstawienie stałej 1 do mianownika nie jest konieczne i służy do przydzielania niezerowej wartości termom znajdującym się we wszystkich próbkach uczących; algorytm gwarantuje, że niewielkie częstości danych tekstowych nie będą otrzymywać zbyt dużej wagi.

Biblioteka scikit-learn zawiera jeszcze jedną klasę transformującą, `TfidfTransformer`, przyjmującą częstości termów przechowywane w klasie `CountVectorizer` i przekształcające je w wartości *tf-idf*:

```
>>> from sklearn.feature_extraction.text import TfidfTransformer
>>> tfidf = TfidfTransformer()
>>> np.set_printoptions(precision=2)
>>> print(tfidf.fit_transform(count.fit_transform(docs)).toarray())
[[ 0.        0.43      0.        0.43      0.        0.56      0.56      0.        0.      ]
 [ 0.        0.43      0.        0.43      0.56      0.        0.        0.        0.56]
 [ 0.28      0.33      0.56      0.49      0.21      0.21      0.21      0.28      0.21]]
```

Jak wiemy z poprzedniego ustępu, wyraz „jest” ma największą częstość termu w trzecim dokumencie, przez co stanowi najczęściej występujące słowo. Jednak po przekształceniu tego samego wektora cech za pomocą algorytmu tf-idf widzimy, że słowo to jest powiązane teraz ze względnie niską wartością tf-idf (0,49) w trzecim dokumencie, ponieważ występuje również w pozostałych dwóch dokumentach tekstowych, zatem jest mało prawdopodobne, że jest ono nośnikiem przydatnych, rozróżniających informacji. Z drugiej strony wyraz „jeden” (trzeci element wektorów cech) ma także częstotliwość równą 2 w trzecim dokumencie, ale większą wartość tf-idf (0,56), gdyż pojawia się wyłącznie w tym dokumencie, co sprawia, że wyraz ten cechuje większą rozróżnialność.

Gdybyśmy jednak ręcznie policzyli wartości tf-idf poszczególnych termów w wektorach cech, okazałoby się, że klasa `TfidfTransformer` oblicza je nieco inaczej niż **standardowe**, omówione wcześniej wzory. Równanie na odwrotną częstość tekstów zaimplementowane w bibliotece scikit-learn wygląda następująco:

$$\text{idf}(t,d) = \log \frac{1 + n_d}{1 + \text{df}(d,t)}$$

Z kolei wzór na model tf-idf został zaimplementowany tak jak poniżej:

$$\text{tf-idf}(t,d) = \text{tf}(t,d) \times (\text{idf}(t,d) + 1)$$

Zazwyczaj częstości termów są normalizowane jeszcze przed obliczeniem wartości tf-idf, ale klasa `TfidfTransformer` normalizuje wyniki tf-idf bezpośrednio. Domyślnie (`norm='l2'`) przeprowadzana jest normalizacja L2, która zwraca wektor o długości 1, poprzez podzielenie nieznormalizowanego wektora cech  $v$  przez normę L2:

$$v_{norm} = \frac{v}{\|v\|_2} = \frac{v}{\sqrt{v_1^2 + v_2^2 + \dots + v_n^2}} = \frac{v}{\left(\sum_{i=1}^n v_i^2\right)^{1/2}}$$

Aby upewnić się, że rozumiemy mechanizm działania klasy `TfidfTransformer`, przyjrzyjmy się przykładowi i wyliczmy wartość tf-idf wyrazu „jest” w trzecim dokumencie.

Wyraz „jest” ma w trzecim dokumencieczęstość termu równą 3 ( $tf = 3$ ), jego częstość występowania w tekście również wynosi 3, ponieważ znajduje się we wszystkich trzech zdaniach ( $df = 3$ ). Możemy więc wyliczyć parametr  $idf$  następująco:

$$idf("jest", d3) = \log \frac{1+3}{1+3} = 0$$

Teraz w celu obliczenia wartości tf-idf wystarczy dodać 1 do odwrotnej częstości w dokumencie i pomnożyć wynik przez częstość termu:

$$tf\text{-}idf("jest", d3) = 3 \times (0 + 1) = 3$$

Po przeprowadzeniu analogicznych obliczeń dla pozostałych termów w trzecim dokumencie uzyskamy następujący wektor tf-idf: [1,69, 2, 3,39, 3, 1,29, 1,29, 1,29, 1,69 i 1,29]. Widać jednak, że wartości w tym wektorze cech różnią się od wyników uzyskanych za pomocą klasy `TfidfTransformer`. Musimy jeszcze tylko przeprowadzić normalizację L2, którą wykonamy w sposób zaprezentowany poniżej:

$$\begin{aligned} tf\text{-}idf_{norm} &= \frac{[1,69, 2, 3,39, 3, 1,29, 1,29, 1,29, 1,69, 1,29]}{\sqrt{[1,69^2, 2^2, 3,39^2, 3^2, 1,29^2, 1,29^2, 1,29^2, 1,69^2, 1,29^2]}} = \\ &= [0,28, 0,33, 0,56, 0,49, 0,21, 0,21, 0,21, 0,28, 0,21] \\ tf\text{-}idf_{norm}("jest", d3) &= 0,45 \end{aligned}$$

Jak widać, uzyskane wyniki są zgodne z rezultatami zwracanymi przez klasę `TfidfTransformer`. Skoro już wiemy, jak są wyliczane wartości tf-idf, możemy przejść do dalszej części rozdziału i wykorzystać omówione koncepcje w odniesieniu do naszego zestawu recenzji filmów.

## Oczyszczanie danych tekstowych

W poprzednich ustępach poszerzyliśmy naszą wiedzę o takie zagadnienia jak model worka słów, częstość termów i parametr tf-idf. Jednakże pierwszą istotną czynnością — jeszcze przed utworzeniem modelu worka słów — jest oczyszczenie danych tekstowych poprzez usunięcie wszelkich niepotrzebnych znaków. Aby zrozumieć, dlaczego jest to takie ważne, wyświetlimy 50 ostatnich znaków z pierwszego tekstu w przetasowanym zbiorze danych:

```
>>> df.loc[0, 'Recenzja'][-50:]
'is seven.<br /><br />Title (Brazil): Not Available'
```

Jak widać, zwrócony został tekst składający się ze znaczników języka HTML, znaków interpunkcyjnych oraz z innych symboli nieliterowych. Znaczniki języka HTML nie są przydatne pod względem semantycznym, ale znaki interpunkcyjne w pewnych kontekstach przetwarzania języka naturalnego mogą zawierać użyteczne informacje. Dla uproszczenia pozbędziemy się wszelkich znaków interpunkcyjnych, ale pozostawimy **emotikony**, takie jak symbol „; )”, ponieważ są bardzo przydatne w analizie sentymentów. Żeby tego dokonać, użyjemy biblioteki **wyrażeń regularnych** (ang. *regular expression — regex*), re, zgodnie z poniższym przykładem:

```
>>> import re
>>> def preprocessor(text):
...     text = re.sub('<[^>]*>', '', text)
...     emoticons = re.findall('(?:.;|=)(?:-)?(?:\:)|\(|D|P|', text)
...     text = re.sub('[\W]+', ' ', text.lower()) + \
...             ''.join(emoticons).replace('-', '')
...     return text
```

Za pomocą pierwszego wyrażenia regularnego < [ ^ > ] \* > próbujemy usunąć wszystkie znaczniki języka HTML występujące w zbiorze danych. Wielu programistów nie pochwala tego typu rozwiązania w stosunku do składni HTML, ale w tym przypadku powinno ono wystarczyć do **oczyszczenia** naszego zbioru recenzji. Po pozbiciu się znaczników HTML wprowadziliśmy nieco bardziej skomplikowane wyrażenia regularne mające na celu wyszukiwanie emotikonów, które będą tymczasowo przechowywane w obiekcie `emoticons`. W kolejnym etapie usuneliśmy wszelkie niewyrazowe znaki z tekstu poprzez wyrażenie regularne [\W] +, przekształciliśmy tekst w taki sposób, aby składał się z samych małych liter, a na końcu przetworzonego tekstowego ciągu znaków dodaliśmy wspomniany już tymczasowy obiekt `emoticons`. Ponadto usuneliśmy z emotikonów symbol **nosa** (-) dla ich ujednolicenia.

Wyrażenia regularne stanowią bardzo skuteczny i wygodny sposób wyszukiwania znaków tekście, ale opanowanie ich obsługi wymaga wiele nauki. Niestety, ich dokładny opis wykracza poza ramy niniejszej książki. Znajdziesz jednak znakomite wprowadzenie na ich temat w serwisie Google Developers (<https://developers.google.com/edu/python/regular-expressions>), a także wśród oficjalnej dokumentacji modułu re (<https://docs.python.org/3.4/library/re.html>).

Chociaż emotikony dodane na końcu oczyszczonego tekstu nie wyglądają zbyt elegancko, kolejność wyrazów nie ma znaczenia w modelu worka słów, jeśli nasz słownik składa się wyłącznie z jednowyrazowych znaczników. Zanim jednak przejdziemy do omówienia rozdzielania danych tekstowych na pojedyncze termy, słowa lub znaczniki, upewnijmy się, że nasza funkcja wstępniego przetwarzania tekstu działa należycie:

```
>>> preprocessor(df.loc[0, 'Recenzja'][-50:])
'is seven title brazil not available'
>>> preprocessor("</a>Przeprowadźmy :) mały :( test :-)!")
'przeprowadźmy mały test :( :)'
```

Zróbmy jeszcze jedną rzecz: skoro w kolejnych ustępach będziemy korzystać z oczyszczonego tekstu, zastosujmy funkcję `preprocessor` wobec wszystkich recenzji zawartych w obiekcie Data Frame:

```
>>> df['Recenzja'] = df['Recenzja'].apply(preprocessor)
```

## Przetwarzanie tekstu na znaczniki

Po skutecznym przygotowaniu zestawu danych musimy pomyśleć nad sposobem podziału ciała tekstu na pojedyncze elementy. Jedeną z metod tzw. **tokenizacji** jest rozdzielenie tekstu na poszczególne wyrazy poprzez analizę odstępów w oczyszczonym tekście:

```
>>> def tokenizer(text):
...     return text.split()
>>> tokenizer('biegacze lubią biegać i dlatego oni biegają')
['biegacze', 'lubią', 'biegać', 'i', 'dlatego', 'oni', 'biegają']
```

W kontekście tokenizacji kolejną przydatną techniką jest **rdzeniowanie wyrazów** (ang. *word stemming*), dzięki czemu zostaje wyznaczony rdzeń wyrazu służący do stworzenia sieci powiązań pomiędzy słowami. Pierwszy algorytm analizy słowotwórczej został zaproponowany przez Martina F. Portera w 1979 roku i dlatego jest nazywany **algorytmem Portera** (M.F. Porter, *An Algorithm for Suffix Stripping*, „Program: Electronic Library and Information Systems” 1980, nr 14 (3), s. 130 – 137). Biblioteka NLTK (ang. *Natural Language Toolkit* — przybornik języka naturalnego; [www.nltk.org](http://www.nltk.org)) w Pythonie zawiera algorytm Portera; za chwilę go przetestujemy.

W celu zainstalowania interfejsu NLTK można skorzystać z komendy `pip install nltk`.

```
>>> from nltk.stem.porter import PorterStemmer
>>> porter = PorterStemmer()
>>> def tokenizer_porter(text):
...     return [porter.stem(word) for word in text.split()]
>>> tokenizer_porter('runners like running and thus they run')
['runner', 'like', 'run', 'and', 'thu', 'they', 'run']
```

Biblioteka NLTK nie stanowi głównej części tego rozdziału, ale i tak polecam wszystkim zainteresowanym odwiedzenie oficjalnej strony interfejsu, jak również zapoznanie się z podręcznikiem jego obsługi, który można bezpłatnie przeglądać na stronie <http://www.nltk.org/book>.

Dzięki klasie `PorterStemmer` zmodyfikowaliśmy funkcję `tokenizer` w taki sposób, że zredukuwała wyrazy do ich rdzenia słowotwórczego, co zaprezentowałem w powyższym przykładzie, w którym z wyrazu „running” został wydzielony rdzeń „run”<sup>1</sup>.

<sup>1</sup> Zaimplementowany w Pythonie algorytm Portera nie za dobrze sobie radzi z polskimi wyrazami, dla tego wyjątkowo pozostawiłem oryginalne zdanie tutaj i w następnym przykładzie — przyp. tłum.

Algorytm Portera jest prawdopodobnie najprostszym i najstarszym algorytmem rdzeniującym. Do innych algorytmów z tej klasy należą m.in. nowszy algorytm **Snowball** (inaczej Porter2 lub „angielski” algorytm rdzeniowania) oraz algorytm **Lancaster** (algorytm Paice/Husk), które są szybsze, ale agresywniejsze od metody Portera. Również one zostały zaimplementowane w pakiecie NLTK (<http://www.nltk.org/api/nltk.stem.html>).

Jak mieliśmy okazję przekonać się w powyższym przykładzie, algorytm Portera generuje czasami również nieistniejące słowa (np. „thu” od wyrazu „thus”), istnieje jednak technika, zwana **lematyzacją**, której zadaniem jest wyznaczanie form kanonicznych (poprawnych gramatycznie, tzw. **lematów**) poszczególnych wyrazów. Proces ten jednak wymaga większej mocy obliczeniowej w porównaniu z rdzeniowaniem i okaże się w praktyce, że zarówno rdzeniowanie, jak i lematyzacja mają niewielki wpływ na skuteczność klasyfikowania tekstu (M. Toman, R. Tesar i K. Jezek, *Influence of Word Normalization on Text Classification*, „Proceedings on InSciT” 2006, s. 354 – 358).

Zanim przejdziemy do następnej części rozdziału, w której będziemy trenować model uczenia maszynowego za pomocą metody worka słów, przyjrzyjmy się jeszcze jednej przydatnej technice, zwanej **usuwaniem pomijanych słów** (ang. *stop-word removal*). Słownami pomijanymi nazywamy wyrazy występujące powszechnie we wszystkich rodzajach tekstów i niezawierające (lub zawierające bardzo mało) przydatnych informacji pozwalających na rozróżnianie poszczególnych klas tekstu. Przykładami takich słów są „**jest**”, „**i**”, „**ma**”. Usuwanie pomijanych słów jest użyteczne, gdy mamy do czynienia z nieprzetworzonymi lub ze znormalizowanymi częściami termów, a nie z wynikami tf-idf, gdyż w tej ostatniej technice często występujące słowa mają już zmniejszone wagę.

Aby usunąć pomijane słowa z wykorzystywanych przez nas recenzji filmów, wykorzystamy zbiór 127 angielskich wyrazów umieszczony w bibliotece NLTK; uzyskujemy do nich dostęp, wywoławszy funkcję `nltk.download`:

```
>>> import nltk  
>>> nltk.download('stopwords')
```

Po pobraniu zestawu pomijanych słów możemy go wczytać i wdrożyć do kodu w następujący sposób:

```
>>> from nltk.corpus import stopwords  
>>> stop = stopwords.words('english')  
>>> [w for w in tokenizer_porter('a runner likes running and runs a  
...                                     lot')[-10:] if w not in stop]  
['runner', 'like', 'run', 'run', 'lot']
```

# Uczenie modelu regresji logistycznej w celu klasyfikowania tekstu

W tym podrozdziale wytrenujemy model regresji logistycznej w celu rozdzielenia recenzji filmów na pozytywne i negatywne. Najpierw podzielimy obiekt DataFrame zawierający oczyszczone teksty na 25 000 próbek uczących i tyle samo danych testowych:

```
>>> X_train = df.loc[:25000, 'Recenzja'].values
>>> y_train = df.loc[:25000, 'Sentyment'].values
>>> X_test = df.loc[25000:, 'Recenzja'].values
>>> y_test = df.loc[25000:, 'Sentyment'].values
```

Teraz wykorzystamy obiekt GridSearchCV (dokładnie pięciokrotny sprawdzian krzyżowy) do znalezienia optymalnego zestawu parametrów dla naszego modelu regresji logistycznej:

```
>>> from distutils.version import LooseVersion as Version
>>> from sklearn import __version__ as sklearn_version
>>> if Version(sklearn_version) < '0.18':
>>>     from sklearn.grid_search import GridSearchCV
>>> else:
>>>     from sklearn.model_selection import GridSearchCV
>>> from sklearn.pipeline import Pipeline
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> tfidf = TfidfVectorizer(strip_accents=None,
...                         lowercase=False,
...                         preprocessor=None)
>>> param_grid = [{ 'vect__ngram_range': [(1,1)],
...                  'vect__stop_words': [stop, None],
...                  'vect__tokenizer': [tokenizer,
...                                     tokenizer_porter],
...                  'clf__penalty': ['l1', 'l2'],
...                  'clf__C': [1.0, 10.0, 100.0]},
...                 { 'vect__ngram_range': [(1,1)],
...                  'vect__stop_words': [stop, None],
...                  'vect__tokenizer': [tokenizer,
...                                     tokenizer_porter],
...                  'vect__use_idf':[False],
...                  'vect__norm':[None],
...                  'clf__penalty': ['l1', 'l2'],
...                  'clf__C': [1.0, 10.0, 100.0]}
...                ]
>>> lr_tfidf = Pipeline([('vect', tfidf),
...                      ('clf',
...                       LogisticRegression(random_state=0))])
>>> gs_lr_tfidf = GridSearchCV(lr_tfidf, param_grid,
```

```

...
scoring='accuracy',
...
cv=5, verbose=1,
n_jobs=1)
>>> gs_lr_tfidf.fit(X_train, y_train)

```

Po zainicjowaniu obiektu `GridSearchCV` i zdefiniowaniu klasy `param_grid` ograniczyliśmy się do określonej liczby kombinacji parametrów, ponieważ zarówno liczba wektorów cech, jak i rozmiar słownika wymagają dużej mocy obliczeniowej; w tym przypadku przeszukiwanie siatki na standardowym komputerze stacjonarnym może zająć do 11 godzin.

W powyższym fragmencie kodu zastąpiliśmy klasy `CountVectorizer` i `TfidfTransformer` klasą `TfidfVectorizer`, która stanowi połączenie dwóch pierwszych algorytmów transformujących. Obiekt `param_grid` zawiera dwa słowniki. W pierwszym z nich wykorzystaliśmy domyślną konfigurację klasy `TfidfVectorizer` (`use_idf=true`, `smooth_idf=True` i `norm='L2'`) do wyliczenia wartości tf-idf; w drugim słowniku zmodyfikowaliśmy te parametry (`use_idf=False`, `smooth_idf=False` i `norm=None`) w celu wytrenowania modelu przy użyciu standardowej częstości termów. Ponadto w przypadku samego modelu regresji logistycznej trenujemy modele za pomocą regularizacji L1 i L2 poprzez parametr kary, po czym porównujemy różne siły regularizacji, definiując zakres wartości parametru `C`.

Po zakończeniu przeszukiwania siatki możemy zerknąć na zbiór najlepszych parametrów:

```

>>> print('Zestaw najlepszych parametrów: %s' % gs_lr_tfidf.best_params_)
Zestaw najlepszych parametrów: {'clf_C': 10.0, 'vect_stop_words': None,
'clf_penalty': 'l2', 'vect_tokenizer': <function tokenizer at
0x7f6c704948c8>, 'vect_ngram_range': (1, 1)}

```

Jak widać, najlepsze wyniki przeszukiwania siatki uzyskaliśmy za pomocą standardowego algorytmu tokenizacji, bez udziału algorytmu Portera, biblioteki pomijanych słów, a w obecności wyników tf-idf w połączeniu z klasyfikatorem regresji logistycznej wykorzystującym regularyzację L2 o sile  $C = 10$ .

Korzystając z najlepszego modelu uzyskanego za pomocą przeszukiwania siatki, sprawdźmy wyniki średnionej dokładności pięciokrotnego sprawdzianu krzyżowego dla zestawu uczącego oraz dokładność klasyfikacji dla danych testowych:

```

>>> print('Dokładność sprawdzianu krzyżowego: %.3f'
...      % gs_lr_tfidf.best_score_)
Dokładność sprawdzianu krzyżowego: 0.897
>>> clf = gs_lr_tfidf.best_estimator_
>>> print('Dokładność wobec danych testowych: %.3f'
...      % clf.score(X_test, y_test))
Dokładność wobec danych testowych: 0.899

```

Wyniki te mówią, że nasz model uczenia maszynowego prognozuje przynależność recenzji filmu do klasy pozytywnej lub negatywnej z niemal 90-procentową dokładnością.

Bardzo popularnym algorytmem do klasyfikowania tekstu pozostaje naiwny klasyfikator bayesowski (ang. *naive Bayes classifier*), który jest bardzo często stosowany w filtrach antyspamowych. Tego typu klasyfikatory są niezwykle łatwe w implementacji, skuteczne obliczeniowo i świetnie się spisują wobec względnie małych zbiorów danych w porównaniu z innymi algorytmami. Omówienie naiwnych klasyfikatorów bayesowskich wykraca poza ramy niniejszej książki, ale zainteresowane osoby mogą zapoznać się z moim bezpłatnym artykułem omawiającym wspomniany temat, dostępnym w serwisie ArXiv (S. Raschka, *Naïve Bayes and Text Classification I — Introduction and Theory*, „Computing Research Repository” (CoRR) 2014, abs/1410.5329, <https://arxiv.org/pdf/1410.5329v3.pdf>).

## Praca z większą ilością danych — algorytmy sieciowe i uczenie pozardzeniowe

Zapewne zauważyłeś, że niektóre algorytmy z poprzednich podrozdziałów są dość kosztowne obliczeniowo podczas np. tworzenia wektorów cech dla zbioru 50 000 recenzji filmowych (zwłaszcza na etapie przeszukiwania siatki). W wielu rzeczywistych zastosowaniach uczenia maszynowego miewamy do czynienia z jeszcze większymi zestawami danych, które mogą nawet nie mieścić się w pamięci komputera. Nie każdy ma dostęp do superkomputerów, dlatego zapoznamy się teraz z techniką zwaną uczeniem pozardzeniowym (ang. *out-of-core learning*), pozwalającą na pracowanie z tak olbrzymimi zbiorami danych.

W rozdziale 2., „Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji”, wprowadziliśmy koncepcję **stochastycznego spadku wzduż gradientu** — algorytmu optymalizującego, w którym wagę modelu są aktualizowane kolejno, jedna próbka po drugiej. W tym podrozdziale wykorzystamy funkcję `partial_fit` z klasy `SGDClassifier` do strumieniowego przesyłania tekstu bezpośrednio z lokalnego dysku oraz trenowania modelu regresji logistycznej za pomocą niewielkich podzbiorów danych tekstowych.

Zdefiniujmy najpierw funkcję `tokenizer`, która oczyści nieprzetworzone dane tekstowe w naszym pliku `filmy_dane.csv`, a następnie wydzieli znaczniki słów przy jednoczesnym usunięciu pomijanych słów.

```
>>> import numpy as np
>>> import re
>>> from nltk.corpus import stopwords
>>> stop = stopwords.words('english')
>>> def tokenizer(text):
...     text = re.sub('<[^>]*>', '', text)
...     emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)', text.lower())
...     text = re.sub('[\W]+', ' ', text.lower())
...     + ''.join(emoticons).replace('-', ' ')
...     tokenized = [w for w in text.split() if w not in stop]
...     return tokenized
```

Stworzymy następnie funkcję `stream_docs` każdorazowo wczytującą i zwracającą jeden tekst:

```
>>> def stream_docs(path):
...     with open(path, 'r', encoding='utf-8') as csv:
...         next(csv) # pomija nagłówek
...         for line in csv:
...             text, label = line[:-3], int(line[-2])
...             yield text, label
```

Aby się przekonać, czy funkcja `stream_docs` działa właściwie, wczytajmy pierwszy tekst z pliku `filmy_dane.csv`, w wyniku czego powinna zostać wyświetlona krotka składająca się z tekstu recenzji i odpowiedniej etykiety klas:

```
>>> next(stream_docs(path='./filmy_dane.csv'))
("In 1974, the teenager Martha Moxley ... ",1)
```

Przechodzimy teraz do funkcji `get_minibatch`, która pobiera strumień danych tekstowych z funkcji `stream_docs` i zwraca określoną liczbę danych tekstowych zdefiniowaną parametrem `size`:

```
>>> def get_minibatch(doc_stream, size):
...     docs, y = [], []
...     try:
...         for _ in range(size):
...             text, label = next(doc_stream)
...             docs.append(text)
...             y.append(label)
...     except StopIteration:
...         return None, None
...     return docs, y
```

Niestety, nie możemy wykorzystać klasy `CountVectorizer` w uczeniu pozardzeniowym, ponieważ wymaga ona wczytania całego słownika w pamięci. Poza tym klasa `TfidfVectorizer` musi przechowywać w pamięci wszystkie wektory cech zestawu danych uczących po to, aby wyliczać odwrotne częstości w tekstach. Jednak w bibliotece scikit-learn dostępna jest jeszcze jedna przydatna klasa do przetwarzania tekstu — `HashingVectorizer`. Jest ona niezależna od danych i wykorzystuje sztuczkę z obliczaniem skrótów wykorzystującą 32-bitowy algorytm Murmur-Hash3 autorstwa Austin Appleby'ego (<https://sites.google.com/site/murmurhash/>).

```
>>> from sklearn.feature_extraction.text import HashingVectorizer
>>> from sklearn.linear_model import SGDClassifier
>>> vect = HashingVectorizer(decode_error='ignore',
...                           n_features=2**21,
...                           preprocessor=None,
...                           tokenizer=tokenizer)
>>> clf = SGDClassifier(loss='log', random_state=1, n_iter=1)
>>> doc_stream = stream_docs(path='./filmy_dane.csv')
```

Dzięki powyższemu fragmentowi kodu zainicjowaliśmy klasę `HashingVectorizer` wraz z funkcją `tokenizer` i wyznaczyliśmy liczbę cech równą  $2^{21}$ . Poza tym ponownie uruchomiliśmy klasyfikator regresji logistycznej, wprowadziwszy wartość `log` do parametru `loss` klasy `SGDClassifier` — poprzez wyznaczenie dużej liczby cech w obiekcie `HashingVectorizer` zmniejszamy prawdopodobieństwo pojawiania się konfliktów skrótów, ale jednocześnie zwiększamy liczbę współczynników w modelu regresji logistycznej.

Teraz robi się naprawdę ciekawie. Po skonfigurowaniu wszystkich uzupełniających funkcji możemy rozpocząć proces uczenia pozardzeniowego, korzystając z poniższego kodu:

```
>>> import pyprind
>>> pbar = pyprind.ProgBar(45)
>>> classes = np.array([0, 1])
>>> for _ in range(45):
...     X_train, y_train = get_minibatch(doc_stream, size=1000)
...     if not X_train:
...         break
...     X_train = vect.transform(X_train)
...     clf.partial_fit(X_train, y_train, classes=classes)
...     pbar.update()
0%                                         100%
[########################################] | ETA[sec]: 0.000
Total time elapsed: 50.063 sec
```

Do oszacowania postępów algorytmu uczenia wykorzystaliśmy pakiet PyPrind. Zainicjowaliśmy pasek postępów dla 45 iteracji, a w pętli `for` wykonaliśmy 45 przebiegów podzbiorów danych tekstowych, z których każdy składa się z 1000 próbek.

Po zakończeniu przyrostowego procesu uczenia użyjemy pozostałych 5000 danych tekstowych do oceny skuteczności modelu:

```
>>> X_test, y_test = get_minibatch(doc_stream, size=5000)
>>> X_test = vect.transform(X_test)
>>> print('Dokładność: %.3f' % clf.score(X_test, y_test))
Dokładność: 0.868
```

Jak widać, dokładność modelu wynosi 87%, czyli nieznacznie mniej w porównaniu z wynikami osiągniętymi przez omówiony w poprzednim podrozdziale model wykorzystujący przeszukiwanie siatki w celu optymalizacji hiperparametrów. Jednak uczenie pozardzeniowe jest bardzo skuteczne — pod względem wykorzystania pamięci i przeprowadzenie całego procesu zajęło niecałą minutę. Wykorzystajmy pozostałe 5000 próbek do zaktualizowania modelu:

```
>>> clf = clf.partial_fit(X_test, y_test)
```

Jeżeli zamierzasz przejść bezpośrednio do rozdziału 9., „Wdrażanie modelu uczenia maszynowego do aplikacji sieciowej”, zalecam pozostawienie bieżącej sesji Pythona otwartej. W następnym rozdziale nauczymy się zapisywać na dysku dopiero co wyuczony model oraz wdrażać go w aplikacjach sieciowych.

Chociaż najczęściej stosowanym modelem klasyfikowania tekstu jest algorytm worka słów, w jego zakresie nie wchodzi analiza zdań ani gramatyki. Popularnym rozszerzeniem tego modelu jest **alokacja ukrytej zmiennej Dirichleta** (ang. *latent Dirichlet allocation*), która stanowi klasyczny przykład algorytmu uwzględniającego ukrytą semantykę wyrazów (D.M. Blei, A.Y. Ng i M.I. Jordan, *Latent Dirichlet Allocation*, „The Journal of Machine Learning Research” 2003, nr 3, s. 993 – 1022).

Nowocześniejszą odmianą modelu worka słów jest algorytm **word2vec**, wydany przez firmę Google w 2013 roku (T. Mikolov, K. Chen, G. Corrado i J. Dean, *Efficient Estimation of Word Representations in Vector Space*, arXiv preprint arXiv: 1301.3781, 2013). Model ten jest algorytmem nienadzorowanego uczenia bazującym na sieciach neuronowych, w którym są podejmowane próby automatycznego określania relacji pomiędzy wyrazami. Podstawową koncepcją jest umieszczanie słów cechujących się podobieństwem znaczeniowym w grupach podobieństw; za pomocą inteligentnego rozmieszczenia wektora tego model ten może „odkrywać” pewne wyrazy poprzez proste operacje na wektorach, np. **król–mężczyzna + kobieta = królowa**.

Pod adresem <https://code.google.com/archive/p/word2vec/> znajdziesz pierwotną implementację algorytmu word2vec w języku C, a także przydatne odnośniki do powiązanych źródeł i alternatywnych implementacji.

## Podsumowanie

W tym rozdziale nauczyliśmy się wykorzystywać algorytmy uczenia maszynowego do klasyfikowania tekstu na podstawie ich bieguności, co stanowi podstawowe zadanie w analizie sentymentów (dziedziny przetwarzania języka naturalnego). Nie tylko dowiedzieliśmy się, w jaki sposób kodować tekst do postaci wektora cech przy użyciu modelu worka słów, lecz również poznaliśmy mechanizm ważenia częstości termów pod kątem istotności dzięki zastosowaniu częstości termów — odwrotnej częstości w tekstach.

Z powodu niejednokrotnie olbrzymich wektorów cech w czasie analizy praca z danymi tekstowymi bywa bardzo wymagająca obliczeniowo; w ostatnim podrozdziale omówiłem mechanizm uczenia pozardzeniowego (przyrostowego), dzięki któremu jesteśmy w stanie wytrenować algorytm uczenia maszynowego bez potrzeby wczytywania całego zestawu danych do pamięci komputera.

W kolejnym rozdziale wykorzystamy stworzony tu klasyfikator tekstu i dowiemy się, w jaki sposób można go umieścić w aplikacji sieciowej.

# Wdrażanie modelu uczenia maszynowego do aplikacji sieciowej

W poprzednich rozdziałach poznaliśmy wiele różnych koncepcji i algorytmów z dziedziny uczenia maszynowego, pomagających w lepszym i skuteczniejszym podejmowaniu decyzji. Jednak techniki uczenia maszynowego nie ograniczają się wyłącznie do lokalnych analiz i danych, gdyż istnieje możliwość przekształcenia ich w silnik predycyjny usług sieciowych. Do bardzo popularnych i przydatnych zastosowań modeli uczenia maszynowego w internecie należą np. wykrywanie spamu, silniki wyszukiwarek, systemy polecające w portalach multimedialnych lub sklepach internetowych, a także wiele innych.

W niniejszym rozdziale nauczymy się wdrażać modele uczenia maszynowego do aplikacji sieciowych, w których algorytmy będą nie tylko klasyfikować próbki, lecz również uczyć się w czasie rzeczywistym. Zajmiemy się następującymi zagadnieniami:

- zapisywanie bieżącego stanu wytrenowanego modelu uczenia maszynowego,
- wykorzystywanie baz danych SQLite do przechowywania próbek,
- projektowanie aplikacji sieciowej za pomocą popularnego środowiska Flask,
- umieszczenie aplikacji uczenia maszynowego na publicznym serwerze sieciowym.

# Serializacja wyuczonych estymatorów biblioteki scikit-learn

Trenowanie modelu uczenia maszynowego może być bardzo wymagające obliczeniowo, o czym przekonaliśmy się w rozdziale 8., „Wykorzystywanie uczenia maszynowego w analizie sentymentów”. Nie chcemy, oczywiście, uczyć modelu od nowa za każdym razem po zamknięciu interpretera Pythona, lecz analizować nowe dane lub aktualizować aplikację sieciową, prawda? Jednym ze sposobów zachowania **trwałości modelu** jest zastosowanie domyślnie wbudowanego modułu pickle (<https://docs.python.org/3.4/library/pickle.html>), który umożliwia serializowanie i deserializowanie struktur obiektowych Pythona do postaci zwięzłego kodu bajtowego, dzięki czemu jesteśmy w stanie zapisywać bieżący stan klasyfikatora i wczytywać go w razie potrzeby bez konieczności każdorazowego trenowania go na danych uczących. Zanim uruchomisz poniższy kod, wytrenuj najpierw pozardzeniowy model regresji logistycznej, omówiony w ostatniej części poprzedniego rozdziału, tak, że będzie już przygotowany w bieżącej sesji Pythona:

```
>>> import pickle
>>> import os
>>> dest = os.path.join('klasyfikator_filmowy', 'pkl_objects')
>>> if not os.path.exists(dest):
...     os.makedirs(dest)
>>> pickle.dump(stop,
...             open(os.path.join(dest, 'stopwords.pkl'), 'wb'),
...             protocol=4)
>>> pickle.dump(clf,
...             open(os.path.join(dest, 'classifier.pkl'), 'wb'),
...             protocol=4)
```

Z pomocą powyższego kodu stworzyliśmy katalog *klasyfikator\_filmowy*, w którym będziemy przechowywać pliki i dane aplikacji sieciowej. Wewnątrz tego folderu wstawiliśmy podkatalog *pkl\_objects* zawierający serializowane obiekty Pythona na dysku lokalnym. Następnie poprzez metodę `dump` przeprowadziliśmy serializację wyuczonego modelu regresji logistycznej, a także słownik pomijanych wyrazów z biblioteki NLTK, dzięki czemu nie musimy jej instalować na serwerze. Pierwszym przyjmowanym argumentem w metodzie `dump` jest obiekt, na którym ma być wykonana serializacja, natomiast drugi argument to plik, w którym Python będzie zapisywał serializowane dane. Wewnątrz funkcji `open` widzimy argument `wb`, służący do otwierania pliku w trybie binarnym; skonfigurowaliśmy również parametr `protocol=4`, w którym wybraliśmy najnowszy i najskuteczniejszy protokół serializacji (dodany w wersji 3.4 Pythona; jeżeli ta wersja protokołu generuje jakieś problemy, upewnij się, że masz zainstalowaną najnowszą wersję środowiska Python. Ewentualnie możesz wybrać starszą wersję protokołu).

Nasz model regresji logistycznej zawiera kilka tablic interfejsu NumPy (np. wektor `wag`), a skuteczniejszym sposobem ich serializacji jest zastosowanie alternatywnej biblioteki `joblib`. W celu zagwarantowania kompatybilności ze środowiskiem serwerowym korzystamy ze standardowej biblioteki `pickle`. Więcej informacji na temat biblioteki `joblib` znajdziesz na stronie <https://pypi.python.org/pypi/joblib>.

Nie musimy serializować obiektu `HashingVectorizer`, gdyż nie ma potrzeby jego uczenia. Zamiast tego utworzymy nowy skrypt Pythona, za pomocą którego będziemy importować wektoryzator do bieżącej sesji Pythona. Skopiuj poniższy fragment kodu i zapisz go w katalogu `klasyfikator_filmowy` jako `vectorizer.py`:

```
from sklearn.feature_extraction.text import HashingVectorizer
import re
import os
import pickle

cur_dir = os.path.dirname(__file__)
stop = pickle.load(open(
    os.path.join(cur_dir,
    'pkl_objects',
    'stopwords.pkl'), 'rb'))

def tokenizer(text):
    text = re.sub('<[^>]*>', '', text)
    emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)', text.lower())
    text = re.sub('[\W]+', ' ', text.lower()) \
        + ' '.join(emoticons).replace('-', '')
    tokenized = [w for w in text.split() if w not in stop]
    return tokenized

vect = HashingVectorizer(decode_error='ignore',
                        n_features=2**21,
                        preprocessor=None,
                        tokenizer=tokenizer)
```

Po przeprowadzeniu serializacji obiektów Pythona i utworzeniu pliku `vectorizer.py` dobrze byłoby uruchomić ponownie interpreter Pythona lub jądro aplikacji Jupyter Notebook, aby sprawdzić, czy deserializacja jest wykonywana bez problemów. Pragnę jednak zwrócić uwagę, że deserializacja danych pochodzących z nieznanych źródeł stanowi potencjalne zagrożenie bezpieczeństwa, ponieważ moduł `pickle` nie jest zabezpieczony przeciwko złośliwemu kodowi. Z poziomu wiersza poleceń/terminala przejdź do katalogu `klasyfikator_filmowy`, rozpoczętaj nową sesję Pythona i uruchom poniższy kod, aby sprawdzić, czy możesz importować wektoryzator i deserializować model uczenia maszynowego:

```
>>> import pickle
>>> import re
>>> import os
>>> from vectorizer import vect
>>> clf = pickle.load(open(
...     os.path.join('pkl_objects',
...                 'classifier.pkl'), 'rb'))
```

Gdy już uda nam się wczytać wektoryzator i deserializować klasyfikator, możemy użyć tych obiektów do wstępnego przetwarzania próbek i prognozowania zawartego w nich ładunku emocjonalnego:

```
>>> import numpy as np
>>> label = {0:'Negatywna', 1:'Pozytywna'}
>>> example = ['I love this movie']
>>> X = vect.transform(example)
>>> print('Prognoza: %s\nPrawdopodobieństwo: %.2f%%' %\
... (label[clf.predict(X)[0]],\
... np.max(clf.predict_proba(X))*100))
Prognoza: Pozytywna
Prawdopodobieństwo: 91.56%
```

Nasz klasyfikator zwraca etykiety klas w postaci liczb całkowitych, dlatego zdefiniowaliśmy prosty słownik Pythona przekształcający te wartości na odpowiedni ciąg znaków. Następnie zastosowaliśmy obiekt `HashingVectorizer` do przekształcenia prostego dokumentu tekowego na wektor wyrazów `X`. Na końcu wykorzystaliśmy metodę `predict` klasyfikatora regresji logistycznej do prognozowania etykiet klas, a także metodę `predict_proba` w celu zwrócenia prawdopodobieństwa poprawności przewidywań. Zwróć uwagę, że metoda `predict_proba` zwraca tablicę zawierającą wartość prawdopodobieństwa dla każdej unikatowej etykiety klas. Etykieta klas o największym prawdopodobieństwie odpowiada etykiecie klasy zwróconej przez wywołanie metody `predict`, dlatego użyliśmy funkcji `np.max` do wyświetlenia prawdopodobieństwa przewidywanej klasy.

## Konfigurowanie bazy danych SQLite

W tym podrozdziale stworzymy prostą bazę danych **SQLite** (ang. *SQLite database*), w której będą zbierane dodatkowe opinie użytkowników aplikacji sieciowej na temat jakości prognoz. Opinie te możemy następnie wykorzystać do zaktualizowania modelu klasyfikującego. SQLite to bezpłatny silnik bazodanowy, niewymagający oddzielnego serwera do działania, dlatego stanowi on idealne rozwiązanie dla mniejszych projektów i prostych aplikacji sieciowych. Zasadniczo bazę danych SQLite możemy uznać za pojedynczy, samodzielny plik bazodanowy, umożliwiający bezpośredni dostęp do przechowywanych danych. Co więcej, silnik SQLite nie wymaga konfiguracji systemu i jest obsługiwany przez wszystkie najważniejsze systemy operacyjne. Jest uznawany za bardzo stabilną platformę i używają go największe firmy, takie jak Google, Mozilla, Adobe, Apple, Microsoft i in. Jeżeli chcesz dowiedzieć się więcej na temat silnika SQLite, polecam odwiedzenie jego oficjalnej strony — <http://www.sqlite.org/>.

Na szczęście zgodnie z filozofią **baterie w komplecie** jedną ze standardowych bibliotek Pythona jest interfejs `sqlite3`, pozwalający na pracę z bazami danych SQLite (więcej informacji na temat tej biblioteki znajdziesz na stronie <https://docs.python.org/3.4/library/sqlite3.html>).

Z pomocą poniższego kodu stworzymy nową bazę danych wewnątrz katalogu *klasyfikator\_filmowy* i umieścimy w niej dwie przykładowe recenzje:

```
>>> import sqlite3
>>> import os
>>> conn = sqlite3.connect('reviews.sqlite')
>>> c = conn.cursor()
>>> c.execute('CREATE TABLE Recenzja_db' \
...           ' (Recenzja TEXT, Sentyment INTEGER, Data TEXT)')
>>> example1 = 'I love this movie'
>>> c.execute("INSERT INTO Recenzja_db" \
...           " (Recenzja, Sentyment, Data) VALUES" \
...           " (?, ?, DATETIME('now'))", (example1, 1))
>>> example2 = 'I disliked this movie'
>>> c.execute("INSERT INTO Recenzja_db" \
...           " (Recenzja, Sentyment, Data) VALUES" \
...           " (?, ?, DATETIME('now'))", (example2, 0))
>>> conn.commit()
>>> conn.close()
```

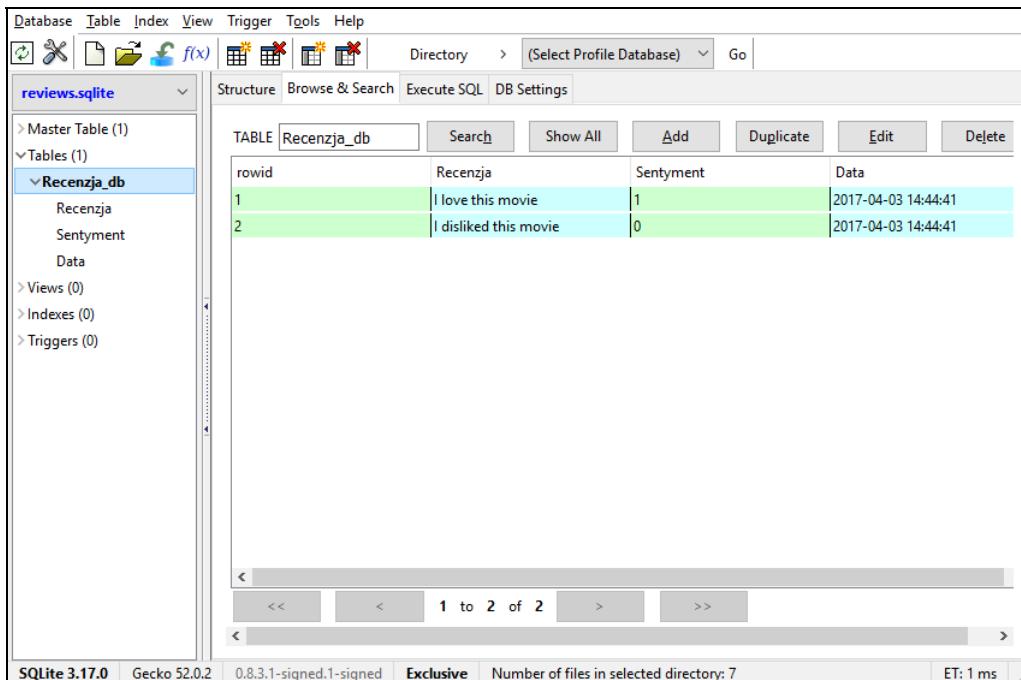
Zgodnie z powyższym kodem stworzyliśmy połączenie (`conn`) z plikiem bazodanowym poprzez wywołanie metody `connect` (część biblioteki `sqlite3`), co spowodowało utworzenie nowego pliku `reviews.sqlite` w katalogu *klasyfikator\_filmowy* (jeżeli jeszcze plik ten nie istniał). Zwróć uwagę, że silnik SQLite nie zawiera funkcji podmieniania zawartości w istniejących tabelach; jeżeli chcesz uruchomić ponownie kod, musisz własnoręcznie usunąć plik bazodanowy z eksploratora plików. Następnie za pomocą metody `cursor` tworzymy kursor umożliwiający poruszanie się po rekordach bazodanowych przy użyciu potężnej składni SQL. Dzięki pierwszemu wywołaniu funkcji `execute` stworzyliśmy nową tabelę bazodanową, *Recenzja\_db*. To w niej będziemy przechowywać dane. Oprócz tabeli wygenerowaliśmy również trzy kolumny: *Recenzja*, *Sentyment* i *Data*. Zapisaliśmy w nich dwie przykładowe recenzje filmów oraz odpowiadające im etykiety klas (sentymenty). Polecenie `DATETIME('now')` służy do dodawania daty i godziny do wpisów tabeli. Oprócz znaczników czasowych wprowadziliśmy też znaki zapytania (?), dzięki którym teksty recenzji (`example1` i `example2`) oraz etykiety klas (1 i 0) stają się argumentami pozycyjnymi dla metody `execute` (jako elementy krotki). Na końcu zastosowaliśmy metodę `commit` do zapisania zmian wprowadzonych w bazie danych i zakończyliśmy połączenie dzięki metodzie `close`.

Aby sprawdzić, czy wpisy zostały poprawnie zapisane w tabeli bazodanowej, nawiążemy ponownie połączenie z bazą danych i użyjemy komendy `SELECT` do wyświetlenia wszystkich krotek utworzonych od 2015 roku aż do teraz:

```
>>> conn = sqlite3.connect('reviews.sqlite')
>>> c = conn.cursor()
>>> c.execute("SELECT * FROM Recenzja_db WHERE Data" \
...           " BETWEEN '2015-01-01 00:00:00' AND DATETIME('now'))"
>>> results = c.fetchall()
>>> conn.close()
```

```
>>> print(results)
[('I love this movie', 1, '2017-04-03 14:44:41'), ('I disliked this
movie', 0, '2017-04-03 14:44:41')]
```

Ewentualnie możemy użyć bezpłatnego dodatku do przeglądarki Firefox — **SQLite Manager** (dostępnego pod adresem <https://addons.mozilla.org/en-US/firefox/addon/sqlite-manager/>), cechującego się mifym dla oka interfejsem, ukazanym na rysunku 9.1.



Rysunek 9.1. Dodatek przeglądarki Firefox — SQLite Manager

## Tworzenie aplikacji sieciowej za pomocą środowiska Flask

Po przygotowaniu w poprzednim podrozdziale kodu klasyfikującego recenzje filmów możemy zająć się omówieniem podstaw środowiska Flask, które posłuży nam do napisania aplikacji sieciowej. Pierwotnie zostało ono wydane przez Armina Ronachera w 2010 roku i od tamtego czasu zyskało olbrzymią popularność; przykładowymi aplikacjami bazującymi na środowisku Flask są LinkedIn oraz Pinterest. Środowisko Flask jest napisane w Pythonie, dlatego osoby programujące w tym języku znajdą tu wygodny interfejs umożliwiający wdrażanie utworzonego kodu (jak choćby naszego klasyfikatora filmowego).

Flask bywa nazywany **mikrośrodowiskiem** (ang. *microframework*), co oznacza, że jego jądro jest bardzo proste i skromne, lecz łatwo można je rozszerzyć za pomocą dodatkowych bibliotek. Choć nauka obsługi tej biblioteki nie jest tak skomplikowana jak innych popularnych interfejsów Pythona (np. Django), zachęcam do zapoznania się z jej dokumentacją dostępną pod adresem <http://flask.pocoo.org/docs/0.10/>.

Jeżeli biblioteka Flask nie jest domyślnie umieszczona wraz ze środowiskiem Python, możesz ją bez problemu zainstalować z poziomu wiersza poleceń<sup>1</sup>:

```
pip install flask
```

## Nasza pierwsza aplikacja sieciowa

W tym podrozdziale stworzymy bardzo prostą aplikację sieciową w celu lepszego zrozumienia interfejsu Flask, a dopiero później przejdziemy do implementacji klasyfikatora filmowego. Najpierw stworzymy drzewo katalogów:

```
flask_pierwsza_aplikacja_1\  
    app.py  
    templates\  
        first_app.html
```

Plik *app.py* będzie zawierał główny kod wykonywany przez interpreter Pythona i będzie nam służył do uruchamiania aplikacji sieciowej. W katalogu *templates* interfejs Flask będzie szukał statycznych plików HTML, a następnie wyświetlał je w przeglądarce. Spójrzmy na zawartość pliku *app.py*:

```
from flask import Flask, render_template  
  
app = Flask(__name__)  
  
@app.route('/')  
def index():  
    return render_template('first_app.html')  
  
if __name__ == '__main__':  
    app.run()
```

W tym przypadku uruchamiamy naszą aplikację w postaci pojedynczego modułu, dlatego zainicjowaliśmy nowe wystąpienie obiektu Flask z argumentem *\_\_name\_\_*, co oznacza, że podkatalog z szablonami (*templates*) znajduje się w tej samej lokacji. Następnie wykorzystujemy dekorator *@app.route ('/')*, by określić adres URL uruchamiający funkcję *index*. Funkcja ta służy tutaj

<sup>1</sup> W czasie tłumaczenia książki najnowsza stabilna wersja była oznaczona numerem 0.12 — *przyp. tłum.*

<sup>2</sup> W przypadku innych systemów operacyjnych niż Windows (np. Linux, Mac OS X) w systemie plików używane są przeciwkośniki (/) — *przyp. tłum.*

jedynie do wyświetlania pliku *first\_app.html*, umieszczonego w katalogu *templates*. Na końcu użyliśmy funkcji *run*, dzięki której aplikacja jest uruchamiana na serwerze wtedy, gdy skrypt zostaje bezpośrednio uruchomiony w interpreterze Pythona, co sobie zagwarantowaliśmy instrukcją `if __name__ == '__main__'`.

Zobaczmy teraz, co kryje się w pliku *first\_app.html*. Jeżeli nie jesteś jeszcze zaznajomiony ze składnią języka HTML, znajdziesz przydatne porady i kursy na stronie <http://www.kurshtml.edu.pl/html/zielony.html> (lub, jeśli znasz język angielski, pod adresem <https://developer.mozilla.org/en-US/docs/Web/HTML>).

```
<!doctype html>
<html>
  <head>
    <title>Pierwsza aplikacja</title>
  </head>
  <body>
    <div>Hej! Oto moja pierwsza aplikacja napisana we Flasku!</div>
  </body>
</html>
```

Wypełniliśmy tu nasz pusty szablon pliku HTML elementem *div* (elementem blokowym) zawierającym zdanie: *Hej! Oto moja pierwsza aplikacja napisana we Flasku!* Interfejs Flask umożliwia uruchamianie aplikacji na poziomie lokalnym, co jest bardzo przydatne podczas jej tworzenia i testowania, zanim wdrożymy ją do publicznego serwera. Zobaczmy, jak wygląda nasza pierwsza aplikacja — wpisz poniższe polecenie wewnętrz katalogu *flask\_pierwsza\_aplikacja\_1*:

```
python3 ap.py
```

W wierszu poleceń powinna pojawić się następująca informacja:

```
* Running on http://127.0.0.1:5000/
```

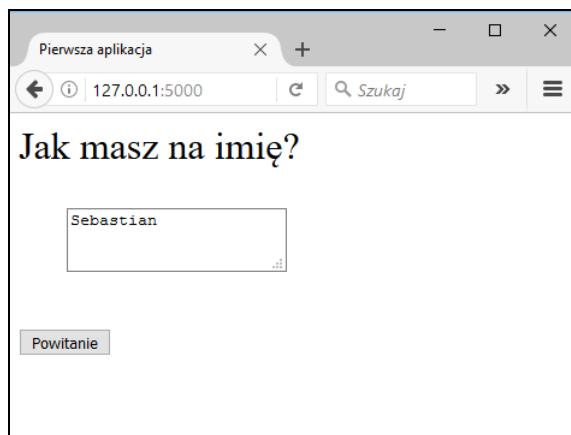
Jest to adres naszego lokalnego serwera. Możemy go teraz wpisać w przeglądarce, aby wyświetlić utworzoną aplikację. Jeżeli wszystko zrobiliśmy jak należy, naszym oczom ukaże się bardzo prosta strona internetowa z napisem: *Hej! Oto moja pierwsza aplikacja napisana we Flasku!*

## Sprawdzanie i wyświetlanie formularza

W tym podrozdziale dodamy do naszej aplikacji sieciowej proste elementy formularza, aby nauczyć się zbierania danych od użytkownika za pomocą biblioteki **WTForms** (<https://wtforms.readthedocs.io/en/latest/>), którą możemy zainstalować przy użyciu interfejsu `pip`:

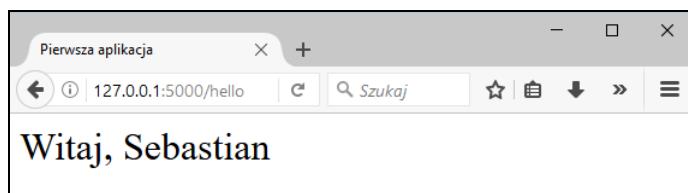
```
pip install wtforms
```

Druga wersja aplikacji będzie zachęcać użytkownika do podania swojego imienia w polu tekstowym, co zostało ukazane na rysunku 9.2.



Rysunek 9.2. Ekran główny drugiej wersji aplikacji

Po wcisnięciu przycisku zgłoszenia (*Powitanie*) i sprawdzeniu formularza zostanie wyświetlona nowa strona z imieniem użytkownika (rysunek 9.3).



Rysunek 9.3. Ekran przywitania użytkownika

Struktura katalogów w nowej aplikacji wygląda następująco:

```
flask_pierwsza_aplikacja_2\
    app.py
    static\
        style.css
    templates\
        _formhelpers.html
        first_app.html
        hello.html
```

Poniżej prezentuję zawartość zmodyfikowanego pliku *app.py*:

```
from flask import Flask, render_template, request
from wtforms import Form, TextAreaField, validators

app = Flask(__name__)

class HelloForm(Form):
    sayhello = TextAreaField('', [validators.DataRequired()])
```

```

@app.route('/')
def index():
    form = HelloForm(request.form)
    return render_template('first_app.html', form=form)

@app.route('/hello', methods=['POST'])
def hello():
    form = HelloForm(request.form)
    if request.method == 'POST' and form.validate():
        name = request.form['sayhello']
        return render_template('hello.html', name=name)
    return render_template('first_app.html', form=form)

if __name__ == '__main__':
    app.run(debug=True)

```

Dzięki bibliotece WTForms rozszerzyliśmy funkcję `index` o pole tekstowe, które wstawimy na stronie początkowej za pomocą klasy `TextAreaField`, sprawdzającej automatycznie, czy użytkownik wprowadził prawidłowe dane. Do tego zdefiniowaliśmy nową funkcję `hello`, wyświetlającą stronę `hello.html` po sprawdzeniu poprawności danych w formularzu. Wykorzystaliśmy tu metodę POST do przeniesienia informacji z formularza do ciała wiadomości. Na koniec dodaliśmy argument `debug=True` wewnętrz metody `app.run`, dzięki czemu uaktywniliśmy debugger biblioteki Flask. Jest to bardzo przydatna funkcja podczas projektowania nowych aplikacji sieciowych.

Zaimplementujemy teraz ogólne makro w pliku `_formhelpers.html` za pomocą silnika szablonowania **Jinja2**; będzie ono później importowane do pliku `first_app.html`, gdzie posłuży do wyświetlania pola tekowego:

```

{% macro render_field(field) %}
<dt>{{ field.label }}
<dd>{{ field(**kwargs)|safe }}
{% if field.errors %}
<ul class=errors>
{% for error in field.errors %}
<li>{{ error }}</li>
{% endfor %}
</ul>
{% endif %}
</dd>
</dt>
{% endmacro %}

```

Dokładna analiza składni języka szablonowania stosowanego w bibliotece Jinja2 wykracza poza zakres niniejszej książki, jednak bogatą dokumentację na jej temat znajdziesz pod adresem <http://jinja.pocoo.org/>.

Przyszedł czas na zdefiniowanie prostego pliku **kaskadowych arkuszy stylów** (ang. *Cascading Style Sheets — CSS*), `style.css`, w celu zaprezentowania możliwości modyfikowania stron HTML.

Poniższy kod, podwajający rozmiar czcionek w wyświetlanym tekście, musimy zapisać w katalogu *static*, gdyż jest to domyślny folder, w którym biblioteka Flask szuka statycznych plików, takich jak właśnie format CSS. Kod ten wygląda następująco:

```
body {
    font-size: 2em;
}
```

Poniżej umieściłem treść zmodyfikowanego pliku *first\_app.html* — posłuży ona do wyświetlania pola tekstowego, w którym użytkownik będzie mógł wpisać swoje imię:

```
<!doctype html>
<html>
    <head>
        <title>Pierwsza aplikacja</title>
        <link rel="stylesheet" href="{{ url_for('static',
            filename='style.css') }}">
    </head>
    <body>

        {% from "_formhelpers.html" import render_field %}

        <div>Jak masz na imię?</div>
        <form method=post action="/hello">
            <dl>
                {{ render_field(form.sayhello) }}
            </dl>
            <input type=submit value='Powitanie' name='submit_btn'>
        </form>
    </body>
</html>
```

W nagłówku pliku *first\_app.html* wczytaliśmy plik CSS. Powinien teraz ulec zmianie rozmiar elementów tekstowych na naszej stronie. W ciele strony wprowadziliśmy makro formularza z pliku *\_formhelpers.html* i wyświetliśmy formularz *sayhello*, zdefiniowany w pliku *app.py*. Ponadto wstawiliśmy do tego formularza przycisk pozwalający na wysłanie danych wprowadzanych przez użytkownika w polu tekstowym.

Został nam jeszcze do przygotowania plik *hello.html*, którego zawartość jest wyświetlana za pomocą wiersza `render_template('hello.html', name=name)` wewnętrz funkcji *hello* zdefiniowanej w skrypcie *app.py* — strona *hello.html* pojawia się na ekranie po przesłaniu danych za pomocą formularza. Umieścmy w tym pliku następujący kod:

```
<!doctype html>
<html>
    <head>
        <title>Pierwsza aplikacja</title>
        <link rel="stylesheet" href="{{ url_for('static',
            filename='style.css') }}">
```

```
</head>
<body>

<div>Witaj, {{ name }}</div>
</body>
</html>
```

Po skonfigurowaniu zmodyfikowanej aplikacji możemy ją za pomocą poniższego polecenia uruchomić lokalnie w głównym katalogu aplikacji i sprawdzić wygląd strony, wkleiwszy do przeglądarki adres <http://127.0.0.1:5000/>:

```
python3 ap.py
```

Osobom niezaznajomionym z tworzeniem aplikacji sieciowych pewne rozwiązania mogą wydawać się w pierwszej chwili bardzo skomplikowane. W takim przypadku polecam przygotowanie powyżej omówionej aplikacji na dysku lokalnym i przeanalizowanie poszczególnych plików. Przekonasz się, że interfejs Flask w istocie jest znacznie prostszy, niż wygląda na pierwszy rzut oka. Ponadto nie zapominaj o znakomitej dokumentacji tej biblioteki i przykładowych aplikacjach, dostępnych na stronie <http://flask.pocoo.org/docs/0.10/>.

## Przekształcanie klasyfikatora recenzji w aplikację sieciową

Skoro zapoznaliśmy się już z podstawami tworzenia aplikacji sieciowych we Flasku, pójdzmy dalej i zaimplementujmy nasz klasyfikator do programu sieciowego. W tym podrozdziale skonstruujemy aplikację, w której użytkownik zostanie najpierw poproszony o podzielenie się opinią dotyczącą filmu (rysunek 9.4).

Po przesłaniu recenzji użytkownik ujrzy nową stronę przedstawiającą prognozowaną etykietę klas i prawdopodobieństwo wyznaczenia właściwej etykiety. Do tego będzie miał do dyspozycji dwa przyciski pozwalające na **potwierdzenie** lub **zaprzeczenie** poprawności przewidywań, co zostało zaprezentowane na rysunku 9.5.

Jeżeli użytkownik kliknie któryś z wymienionych przycisków (*Prawidłowa* lub *Nieprawidłowa*), model klasyfikujący zostanie zaktualizowany zgodnie z opinią użytkownika. Oprócz tego zapiszemy tę recenzję wraz z odpowiadającą jej etykietą klas w bazie danych SQLite. Kolejną stroną, jaką zobaczy użytkownik po kliknięciu któregoś z tych przycisków, jest **ekran podziękowania** zawierający przycisk *Prześlij kolejną recenzję*, odsyłający użytkownika na stronę początkową. Na rysunku 9.6 widzimy stronę z podziękowaniem.

Zanim zajmiemy się implementacją omawianej aplikacji, polecam zapoznanie się z działającą wersją demonstracyjną umieszczoną na stronie <http://raschkas.pythonanywhere.com/>, dzięki czemu będzie Ci łatwiej zrozumieć, co chcemy osiągnąć w tym rozdziale.

Rozpoczniemy od zapoznania się z drzewem katalogów tej aplikacji, zaprezentowanym na rysunku 9.7.

The screenshot shows a web browser window titled "Filmy - klasyfikacja". The address bar displays "127.0.0.1:5000". The main content area contains the following text:

**Opisz swoje odczucia na temat filmu:**

I love ~~this~~ movie!

**Prześlij recenzję**

Rysunek 9.4. Formularz wpisywania recenzji filmu

The screenshot shows a web browser window titled "Filmy - klasyfikacja" with the URL "127.0.0.1:5000/results". The main content area contains the following text:

**Twoja recenzja filmu:**

I love this movie!

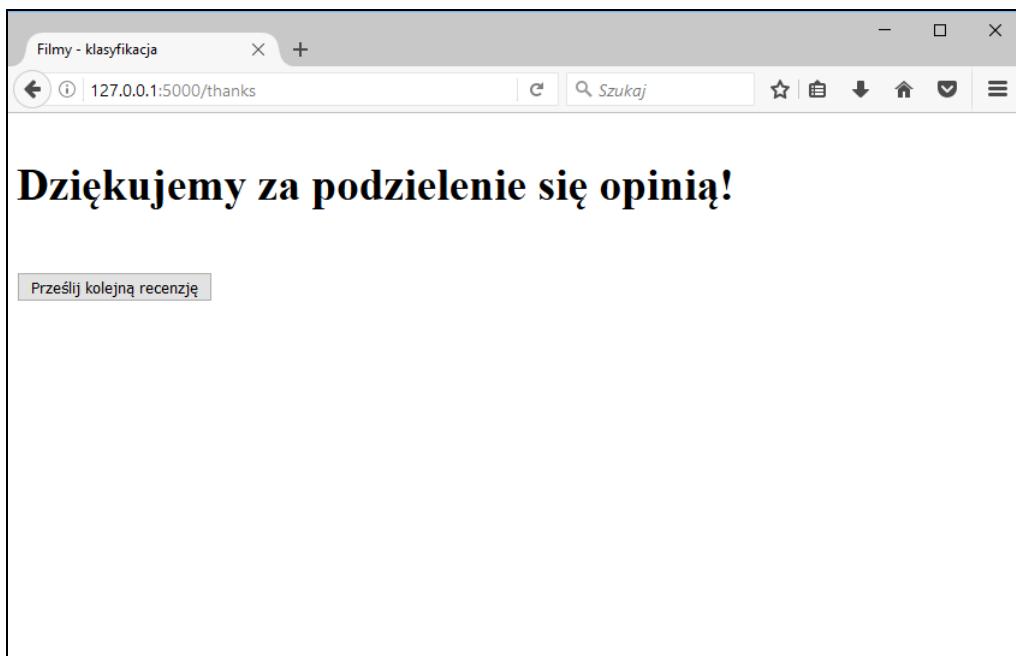
**Prognoza:**

Ta recenzja jest **pozytywna** (prawdopodobieństwo: 82.52%).

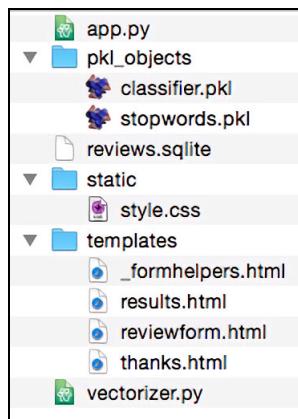
**Prawidłowa** **Nieprawidłowa**

**Prześlij kolejną recenzję**

Rysunek 9.5. Ekran przewidywania sentymentu



Rysunek 9.6. Ekran podziękowania



Rysunek 9.7. Drzewo katalogów aplikacji klasyfikator\_filmowy

W poprzednim podrozdziale stworzyliśmy plik *vectorizer.py*, bazę danych *reviews.sqlite*, a także podkatalog *pkl\_objects* wraz z jego zawartością.

Plik *app.py* w głównym katalogu aplikacji to skrypt Pythona zawierający nasz kod napisany przy użyciu interfejsu Flask, natomiast utworzoną wcześniej bazę danych *reviews.sqlite* wykorzystamy do przechowywania opinii przesłanych przez użytkowników. W katalogu *templates* znaj-

dziemy szablony HTML przetwarzane przez bibliotekę Flask i wyświetlane w przeglądarce, z kolei podkatalog *static* składa się z prostego pliku CSS definiującego wygląd aplikacji.

Plik *app.py* jest dość duży, dlatego rozbijemy jego analizę na dwie części. Pierwsza z nich służy do importowania modułów Pythona oraz potrzebnych obiektów, a także wprowadza deserializację i konfigurację modelu klasyfikującego:

```
from flask import Flask, render_template, request
from wtforms import Form, TextAreaField, validators
import pickle
import sqlite3
import os
import numpy as np.

# importuje obiekt HashingVectorizer z lokalnego katalogu
from vectorizer import vect

app = Flask(__name__)

##### przygotowuje klasyfikator
cur_dir = os.path.dirname(__file__)
clf = pickle.load(open(os.path.join(cur_dir,
                                    'pkl_objects/classifier.pkl'), 'rb'))
db = os.path.join(cur_dir, 'reviews.sqlite')

def classify(document):
    label = {0: 'negatywna', 1: 'pozytywna'}
    X = vect.transform([document])
    y = clf.predict(X)[0]
    proba = clf.predict_proba(X).max()
    return label[y], proba

def train(document, y):
    X = vect.transform([document])
    clf.partial_fit(X, [y])

def sqlite_entry(path, document, y):
    conn = sqlite3.connect(path)
    c = conn.cursor()
    c.execute("INSERT INTO Recenzja_db (Recenzja, Sentyment, Data)\"\
    " VALUES (?, ?, DATETIME('now'))", (document, y))
    conn.commit()
    conn.close()
```

Powyższy fragment skryptu *app.py* wygląda całkiem znajomo. Importowaliśmy obiekt `HashingVectorizer` i deserializowaliśmy klasyfikator regresji logistycznej. Następnie zdefiniowaliśmy funkcję `classify` zwracającą przewidywaną etykietę klas, jak również prawdopodobieństwo wyznaczenia prawidłowej klasy dla danego tekstu. Funkcji `train` użyjemy do aktualizacji klasyfikatora na podstawie danego tekstu i jego etykiety klas. Funkcja `sqlite_entry` pozwala

na zapisanie recenzji filmu w bazie danych wraz z jej etykietą klas i znacznikiem czasowym. Zwróć uwagę, że w momencie ponownego uruchomienia aplikacji obiekt `c1f` zostanie przywrócony do pierwotnego, serializowanego stanu. Na końcu rozdziału nauczymy się wykorzystywać dane zawarte w bazie danych SQLite w celu trwałe aktualizacji klasyfikatora.

Zrozumienie drugiej części skryptu `app.py` również nie powinno nam przysporzyć żadnych trudności:

```
app = Flask(__name__)
class ReviewForm(Form):
    moviereview = TextAreaField('',
                                [validators.DataRequired(),
                                 validators.length(min=15)])

@app.route('/')
def index():
    form = ReviewForm(request.form)
    return render_template('reviewform.html', form=form)

@app.route('/results', methods=['POST'])
def results():
    form = ReviewForm(request.form)
    if request.method == 'POST' and form.validate():
        review = request.form['moviereview']
        y, proba = classify(review)
        return render_template('results.html',
                              content=review,
                              prediction=y,
                              probability=round(proba*100, 2))
    return render_template('reviewform.html', form=form)

@app.route('/thanks', methods=['POST'])
def feedback():
    feedback = request.form['feedback_button']
    review = request.form['review']
    prediction = request.form['prediction']

    inv_label = {'negatywna': 0, 'pozytywna': 1}
    y = inv_label[prediction]
    if feedback == 'Nieprawidłowa':
        y = int(not(y))
    train(review, y)
    sqlite_entry(db, review, y)
    return render_template('thanks.html')

if __name__ == '__main__':
    app.run(debug=True)
```

Zdefiniowaliśmy klasę `ReviewForm` tworzącą obiekt `TextAreaField` wyświetlany za pomocą szablonu `reviewform.html` (pierwsza strona wyświetlana przez aplikację). Z kolei ten szablon jest przetwarzany przez funkcję `index`. Za pomocą parametru `validators.length(min=15)` wymagamy od użytkownika, aby napisał recenzję składającą się z co najmniej 15 znaków. Wewnątrz funkcji `results` zbieramy treść przesłanego formularza recenzji, którą następnie przekazujemy klasyfikatorowi w celu prognozowania sentymenu zawartego w tekście, po czym wyniki zostaną wyświetcone w wygenerowanym szablonie `results.html`.

Pozornie funkcja `feedback` wydaje się być dość skomplikowana. W rzeczywistości pobiera przewidawaną etykietę klas z szablonu `results.html` (jeżeli użytkownik wcisnął któryś z przycisków — *Prawidłowa* albo *Nieprawidłowa*), a następnie przekształca prognozowany sentymenit z powrotem na liczbę całkowitą, która zostanie użyta do zaktualizowania klasyfikatora za pomocą funkcji `train` — zaimplementowanej w pierwszej części skryptu `app.py`. Poza tym, jeżeli użytkownik kliknął któryś przycisk, przez funkcję `sqlite_entry` zostanie utworzony nowy wpis w bazie danych, a następnie wyświetlona strona `thanks.html` w celu wyrażenia wdzięczności za pozostawienie opinii.

Spójrzmy teraz na szablon `reviewform.html`, definiujący stronę początkową naszej aplikacji:

```
<!doctype html>
<html>
<head>
    <title>Filmy - klasyfikacja</title>
</head>
<body>

<h2>Opisz swoje odczucia na temat filmu:</h2>

{% from "_formhelpers.html" import render_field %}

<form method=post action="/results">
    <dl>
        {{ render_field(form.moviereview, cols='30', rows='10') }}
    </dl>
    <div>
        <input type=submit value='Prześlij recenzję' name='submit_btn'>
    </div>
</form>

</body>
</html>
```

Importowaliśmy tu obiekt `_formhelpers.html` zdefiniowany w ustępie „Sprawdzanie i wyświetlanie formularza”. Funkcja `render_field` zostaje użyta do wyświetlania obiektu `TextAreaField`, za pomocą którego użytkownik może napisać recenzję filmu i wysłać ją po wcisnięciu przycisku *Prześlij recenzję* zlokalizowanego na dole strony. Pole tekstowe `TextAreaField` składa się z 30 kolumn i 10 wierszy.

Kolejny szablon, *results.html*, prezentuje się nieco bardziej interesująco:

```
<!doctype html>
<html>
  <head>
    <title>Filmy - klasyfikacja</title>
    <link rel="stylesheet" href="{{ url_for('static',
      filename='style.css') }}">
  </head>
  <body>

    <h3>Twoja recenzja filmu:</h3>
    <div>{{ content }}</div>

    <h3>Prognoza:</h3>
    <div>Ta recenzja jest <strong>{{ prediction }}</strong>
      (probability: {{ probability }}%).</div>

    <div class='button'>
      <form action="/thanks" method="post">
        <input type=submit value='Prawidłowa' name='feedback_button'>
        <input type=submit value='Nieprawidłowa' name='feedback_button'>
        <input type=hidden value='{{ prediction }}' name='prediction'>
        <input type=hidden value='{{ content }}' name='review'>
      </form>
    </div>

    <div class='button'>
      <form action="/">
        <input type=submit value='Prześlij kolejną recenzję'>
      </form>
    </div>

  </body>
</html>
```

Najpierw wstawiliśmy przeslaną recenzję oraz wyniki przewidywań w odpowiednich polach: {{ content }}, {{ prediction }} i {{ probability }}. Być może zauważyleś, że już drugi raz wprowadziliśmy zmienne zastępcze {{ content }} i {{ prediction }} w formularzu zawierającym przyciski *Prawidłowa* i *Nieprawidłowa*. Stosujemy to obejście do przesłania tych wartości z powrotem na serwer w celu aktualizacji klasyfikatora oraz zachowania recenzji, gdy użytkownik kliknie jeden z tych przycisków. Poza tym na początku pliku importowaliśmy plik CSS (*style.css*). Jego treść jest całkiem nieskomplikowana; ogranicza szerokość składników aplikacji sieciowej do 600 pikseli i przesuwa przyciski *Prawidłowa* i *Nieprawidłowa* umieszczone w elemencie button w dół o 20 pikseli:

```
body{
  width:600px;
}
```

```
.button{
    padding-top: 20px;
}
```

Treść tego pliku jest całkowicie dowolna, dlatego możesz ją tak zmieniać, aby dostosować wygląd aplikacji według własnego uznania.

Ostatnim implementowanym przez nas plikiem HTML jest szablon *thanks.html*. Jak sama nazwa wskazuje, posłuży on nam do wygenerowania grzecznego **podziękowania** użytkownikowi, który wcisnie przycisk *Prawidłowa* lub *Nieprawidłowa*. Ponadto wstawiamy również na dole strony przycisk *Prześlij kolejną recenzję*, przenoszącą użytkownika na stronę początkową. Zawartość pliku *thanks.html* wygląda następująco:

```
<!doctype html>
<html>
    <head>
        <title>Filmy - klasyfikacja</title>
    </head>
    <body>

        <h3>Dziękujemy za opinię!</h3>
        <div id='button'>
            <form action="/">
                <input type=submit value='Prześlij kolejną recenzję'>
            </form>
        </div>

    </body>
</html>
```

Nadszedł czas, aby uruchomić aplikację lokalnie z poziomu wiersza poleceń, za pomocą poniższego polecenia, a następnie nauczymy się wdrażać ją na publiczny serwer:

```
python3 app.py
```

Po zakończeniu testowania aplikacji zawsze pamiętajmy o usunięciu argumentu `debug=True` z polecenia `app.run()` w skrypcie *ap.py*.

## Umieszczanie aplikacji sieciowej na publicznym serwerze

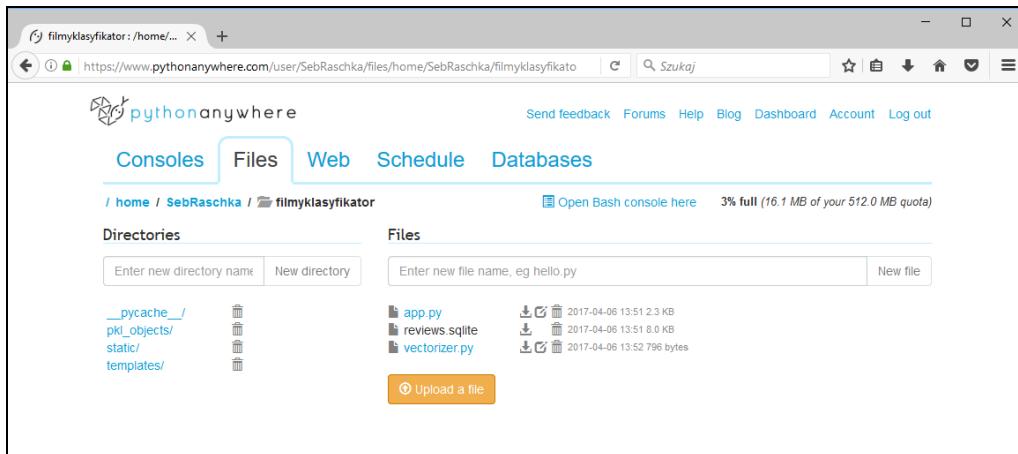
Po przetestowaniu naszej aplikacji na poziomie lokalnym jesteśmy gotowi, aby wdrożyć ją na publicznym serwerze. W tym celu wykorzystamy usługę **PythonAnywhere** specjalizującą się w przechowywaniu aplikacji napisanych w Pythonie, przez co zarządzanie stronami staje się

znacznie prostsze. Co więcej, mamy możliwość założenia podstawowego konta pozwalającego na bezpłatne zarządzanie jedną aplikacją sieciową.

W celu utworzenia nowego konta PythonAnywhere odwiedzamy stronę <https://www.pythonanywhere.com/>, a następnie klikamy odnośnik *Pricing & signup* umieszczony w prawym górnym rogu ekranu. Teraz wciskamy przycisk *Create a Beginner account* i podajemy nazwę użytkownika, hasło i poprawny adres e-mail. Po przeczytaniu warunków użytkowania i zgodzeniu się na ich przestrzeganie stajemy się właścicielami nowego konta.

Niestety, w zakresie podstawowego konta nie mamy możliwości dostępu do zdalnego serwera za pomocą protokołu SSH z poziomu wiersza poleceń. Dlatego musimy zarządzać aplikacją sieciową poprzez interfejs serwisu PythonAnywhere. Zanim jednak wyślemy pliki na serwer, musimy stworzyć nową aplikację sieciową dla naszego konta. Po kliknięciu przycisku *Dashboard* znajdującego się w prawej górnej części ekranu uzyskujemy dostęp do panelu sterowania. Przechodzimy teraz do widocznej na górze zakładki *Web*, a na następnym ekranie klikamy przycisk *Add a new web app*, co pozwoli nam na utworzenie nowej aplikacji (*Flask/Python 3.4*), którą nazwijmy *filmyklasyfikator*.

Po utworzeniu nowej aplikacji kierujemy się do zakładki *Files*, do której możemy przesyłać pliki z lokalnego katalogu za pomocą interfejsu PythonAnywhere. Po skopiowaniu zawartości katalogu *zaktualizowany\_klasifikator\_filmowy* będziemy przechowywać na koncie wszystkie pliki i katalogi potrzebne do działania aplikacji (rysunek 9.8).



Rysunek 9.8. Aplikacja *filmyklasyfikator* przesłana na serwer PythonAnywhere

Na koniec wróćmy jeszcze do zakładki Web i kliknijmy przycisk *Reload <nazwa\_użytkownika>.pythonanywhere.com*, żeby zaakceptować wprowadzone zmiany i odświeżyć informacje o aplikacji sieciowej. Teraz nasza aplikacja powinna normalnie działać i być dostępna pod adresem *<nazwa\_użytkownika>.pythonanywhere.com*.

Serwery sieciowe są, niestety, bardzo wrażliwe na najmniejsze nawet problemy z przesyłanymi aplikacjami. Jeżeli masz kłopoty z uruchomieniem aplikacji w serwisie PythonAnywhere i wyświetlają się informacje o błędach w przeglądarce, możesz je przejrzeć po kliknięciu zakładki *Web* — dzięki temu łatwiej zdiagnozujesz przyczynę problemu.

## Aktualizowanie klasyfikatora recenzji filmowych

Nasz model predykcyjny jest na bieżąco aktualizowany za każdym razem, gdy użytkownik potwierdza dokładność przewidywań lub jej zaprzecza, ale aktualizacje obiektu `clf` zostaną wyzerowane w przypadku zawieszenia lub ponownego uruchomienia serwera. Gdy ponownie uruchomimy aplikację sieciową, obiekt `clf` zostanie zainicjowany z serializowanego pliku `classifier.pkl`. Jednym ze sposobów zachowania aktualizacji jest serializacja obiektu `clf` po każdej aktualizacji modelu. Jest to jednak bardzo nieskuteczne obliczeniowo rozwiązywanie z powodu rosnącej liczby użytkowników; grozi uszkodzeniem serializowanego pliku w przypadku, gdyby wielu użytkowników naraz przesyłało swoje wyniki. Alternatywną możliwością jest aktualizowanie modelu predykcyjnego za pomocą danych zbieranych w bazie danych SQLite. Możemy np. pobrać tę bazę danych z serwisu PythonAnywhere, zaktualizować lokalnie obiekt `clf` i przesłać zaktualizowany plik na serwer. W celu lokalnej aktualizacji klasyfikatora na komputerze stworzymy plik `update.py` wewnętrz katalogu `zaktualizowany_klasifikator_filmowy`:

```
import pickle
import sqlite3
import numpy as np
import os

# importuje klasę HashingVectorizer z lokalnego katalogu
from vectorizer import vect

def update_model(db_path, model, batch_size=10000):

    conn = sqlite3.connect(db_path)
    c = conn.cursor()
    c.execute('SELECT * from Recenzja_db')

    results = c.fetchmany(batch_size)
    while results:
        data = np.array(results)
        X = data[:, 0]
        y = data[:, 1].astype(int)

        classes = np.array([0, 1])
        X_train = vect.transform(X)
        model.partial_fit(X_train, y, classes=classes)
        results = c.fetchmany(batch_size)

    conn.close()
```

```

    return model

    cur_dir = os.path.dirname(__file__)

    clf = pickle.load(open(os.path.join(cur_dir,
                                         'pkl_objects',
                                         'classifier.pkl'), 'rb'))
    db = os.path.join(cur_dir, 'reviews.sqlite')

    update_model(db_path=db, model=clf, batch_size=10000)

    # usuń komentarze z poniższych wierszy, jeśli masz pewność,
    # że chcesz trwale aktualizować plik classifier.pkl

    # pickle.dump(clf, open(os.path.join(cur_dir,
    #                                     'pkl_objects', 'classifier.pkl'), 'wb')
    #             , protocol=4)

```

Funkcja `update_model` pobiera wpisy z bazy SQLite w pakietach o rozmiarze 10 000 próbek, jeśli jest dostępnych w niej tyle recenzji. Ewentualnie możemy przesyłać wpisy pojedynczo, zastępując metodę `fetchmany` wyrażeniem `fetchone`, co stanowi nader nieskuteczne rozwiązanie. Z kolei wprowadzenie `fetchall` może być kłopotliwe w przypadku bardzo dużych baz danych przekraczających rozmiarem pojemność pamięci komputera lub serwera.

Po utworzeniu skryptu `update.py` możemy go przesłać do katalogu `filmyklasyfikator` w serwisie PythonAnywhere oraz importować funkcję `update_model` w głównym skrypcie `app.py`, żeby klasyfikator za każdym razem po zresetowaniu aplikacji był aktualizowany przy użyciu bazy danych. W tym celu wystarczy dodać poniższy wiersz kodu na początku pliku `app.py`:

```
# importuje z lokalnego katalogu funkcję aktualizowania
from update import update_model
```

Musimy teraz wywołać funkcję `update_model` w ciele głównej aplikacji:

```
...
if __name__ == '__main__':
    clf = update_model(db_path="db", model=clf, batch_size=10000)
...

```

## Podsumowanie

W tym rozdziale poruszyliśmy wiele przydatnych i praktycznych tematów poszerzających naszą teoretyczną wiedzę na temat uczenia maszynowego. Nauczyliśmy się serializować wyuczony model oraz wczytywać go do późniejszego użytku. Do tego utworzyliśmy bazę danych SQLite stanowiącą skuteczny magazyn danych, po czym stworzyliśmy aplikację sieciową pozwalającą na podzielenie się klasyfikatorem recenzji filmowych z resztą świata.

Książka ta jest dla nas wehikułem pozwalającym poruszać się po krańcu pojęć z zakresu uczenia maszynowego, najlepszych rozwiązań oraz nadzorowanych modeli klasyfikacji. W kolejnym rozdziale przyjrzymy się kolejnej podkategorii uczenia nadzorowanego — analizie regresywnej — metodzie pozwalającej na przewidywanie wynikowych zmiennych w sposób ciągły, w przeciwieństwie do kategoryzujących etykiet klas spotykanych w dotychczas omówionych modelach klasyfikacji.



# Przewidywanie ciągłych zmiennych docelowych za pomocą analizy regresywnej

W poprzednich rozdziałach poznaliśmy wiele podstawowych koncepcji **uczenia nadzorowanego** i nauczyliśmy się trenować różne modele klasyfikowania próbek, pozwalające prognozować przydziały grupowe lub zmienne kategoryzujące. Teraz skoncentrujemy na kolejnej kategorii uczenia nadzorowanego: **analizie regresywnej**.

Modele regresywne pozwalają na przewidywanie docelowych zmiennych w sposób **ciągły**, przez co techniki te stanowią obiekt zainteresowania w wielu dziedzinach nauki oraz przemysłu; możemy dzięki nim m.in. wyjaśniać relację pomiędzy różnymi zmiennymi, określać trendy oraz tworzyć prognozy. Dobrym przykładem jest prognozowanie obrotów firmy w przyszłych miesiącach.

W niniejszym rozdziale przyjrzymy się głównym założeniom modeli regresywnych i zajmiemy się następującymi tematami:

- przeglądanie i wizualizacja zestawów danych,
- różne metody implementacji modeli regresji liniowej,
- trenowanie odpornych modeli regresywnych uwzględniających odstające próbki,
- ocenianie modeli regresywnych oraz diagnozowanie najczęstszych problemów,
- dopasowywanie modeli regresywnych do nieliniowych danych.

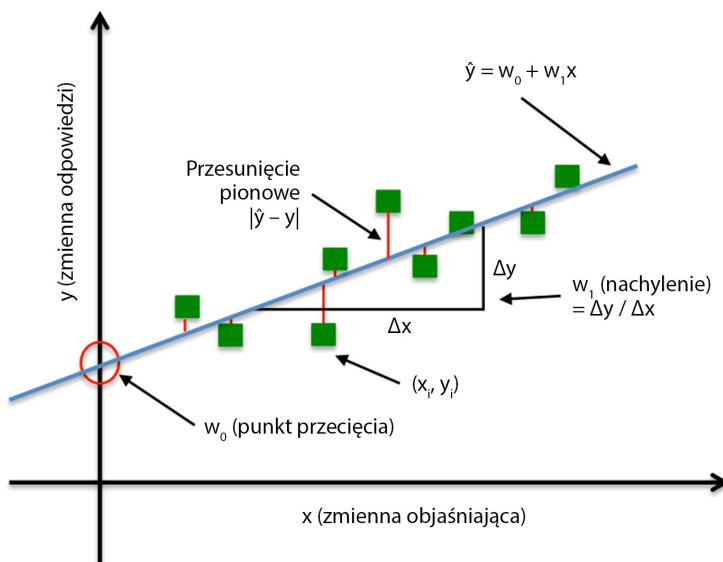
# Wprowadzenie do prostego modelu regresji liniowej

Celem prostej (jednowymiarowej; ang. *univariate*) regresji liniowej jest tworzenie modelu relacji pomiędzy pojedynczą cechą (zmienną objaśniającą  $x$ ) a generowaną w sposób ciągły **odpowiedzią** (zmienną docelową  $y$ ). Równanie modelu liniowego z jedną zmienną objaśniającą wygląda następująco:

$$y = w_0 + w_1 x$$

Waga  $w_0$  symbolizuje punkt przecięcia z osią  $y$ , a  $w_1$  stanowi współczynnik zmiennej objaśniającej. Naszym zadaniem jest uczenie wag modelu liniowego w sposób umożliwiający opis relacji pomiędzy zmienną objaśniającą a docelową, co pozwala w dalszej kolejności na przewidywanie odpowiedzi nowych zmiennych objaśniających niebędących częścią zestawu danych uczących.

Na podstawie powyższego równania możemy zdefiniować regresję liniową jako wyszukiwanie najlepiej dopasowanej prostej przebiegającej pomiędzy punktami danych, co zostało zaprezentowane na rysunku 10.1.



Rysunek 10.1. Schemat modelu regresji liniowej

Taka optymalna prosta jest nazywana **linią regresji**, a pionowe odcinki wyznaczające odległość próbek od tej linii nazywamy **przesunięciami** albo **wartościami resztowymi (rezydualnymi)** — są to błędy predykcji.

Specjalnym przypadkiem modelu zawierającego jedną zmienną objaśniającą jest **prosta regresja liniowa** (ang. *simple linear regression*), możemy jednak, oczywiście, uogólnić analizę regresywną do wielu zmiennych objaśniających, stąd ten proces jest nazywany **wielowymiarową regresją liniową** (ang. *multiple linear regression*):

$$y = w_0x_0 + w_1x_1 + \dots + w_mx_m = \sum_{i=0}^m w_i x_i = w^T x$$

W tym przypadku waga  $w_0$  wyznacza punkt przecięcia z osią  $y$  dla  $x_0 = 1$ .

## Zestaw danych Housing

Zanim przystąpimy do implementacji naszego pierwszego modelu regresji liniowej, wprowadzimy nowy zestaw danych — **Housing** — zawierający informacje o domach na przedmieściach Bostonu, zebrane przez Davida Harrisona i Daniela L. Rubenfelda w 1978 roku. Zbiór ten można bezpłatnie pobrać z **repozytorium uczenia maszynowego UCI**, a znajdziemy go pod adresem <https://archive.ics.uci.edu/ml/datasets/Housing>.

Cechy opisujące 506 zawartych w zestawie próbek zostały opisane poniżej (zgodnie z informacjami podanymi na stronie repozytorium):

- **CRIM:** współczynnik przestępcości per capita na każde miasteczko,
- **ZN:** odsetek działek przekraczających 25 000 stóp kwadratowych ( $\approx 2533 \text{ m}^2$ ),
- **INDUS:** odsetek terenów przeznaczonych na przemysł niedetaliczny na każde miasteczko,
- **CHAS:** zmienna zerojedynkowa określająca rzekę Charles (przyjmuje wartość 1, gdy na danym terenie znajduje się koryto rzeki),
- **NOX:** stężenie tlenków azotu (w częściach na 10 000 000),
- **RM:** średnia liczba pomieszczeń na dom,
- **AGE:** odsetek zamieszkałych budynków wybudowanych przed 1940 rokiem,
- **DIS:** ważona odległość od pięciu bostońskich urzędów pracy,
- **RAD:** wskaźnik dostępności do głównych arterii komunikacyjnych,
- **TAX:** pełna wartość podatku od nieruchomości na każde 10 000 dolarów,
- **PTRATIO:** stosunek liczby uczniów do nauczycieli na każde miasteczko,
- **B:** parametr wyliczany ze wzoru  $1000(Bk - 0,63)^2$ , gdzie  $Bk$  oznacza odsetek osób pochodzenia afroamerykańskiego zamieszkiujących dane miasteczko,
- **LSTAT:** odsetek ubogiej części społeczeństwa,
- **MEDV:** mediana wartości zamieszkałych domów wyrażona w tysiącach dolarów.

W dalszej części rozdziału wykorzystamy cechę *MEDV* (ceny domów) jako naszą zmienną docelową — zmienną, której wartości chcemy przewidywać na podstawie przynajmniej jednej z 13 pozostałych zmiennych objaśniających. Zanim przejdziemy dalej, wczytajmy cały zestaw danych do obiektu DataFrame:

```
>>> import pandas as pd
>>> df = pd.read_csv('https://archive.ics.uci.edu/ml/
    ↪machine-learningdatabases/housing/housing.data',
...                 header=None, sep='\s+')
>>> df.columns = ['CRIM', 'ZN', 'INDUS', 'CHAS',
...                 'NOX', 'RM', 'AGE', 'DIS', 'RAD',
...                 'TAX', 'PTRATIO', 'B', 'LSTAT', 'MEDV']
>>> df.head()
```

Aby upewnić się, że próbki zostały prawidłowo wczytane, wyświetlamy pięć pierwszych wierszy zestawu danych, co zostało ukazane w tabeli 10.1.

**Tabela 10.1.** Pięć pierwszych wierszy zestawu danych Housing

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
<b>0</b>	0.00632	18	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
<b>1</b>	0.02731	0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
<b>2</b>	0.02729	0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
<b>3</b>	0.03237	0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
<b>4</b>	0.06905	0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33	36.2

## Wizualizowanie ważnych elementów zestawu danych

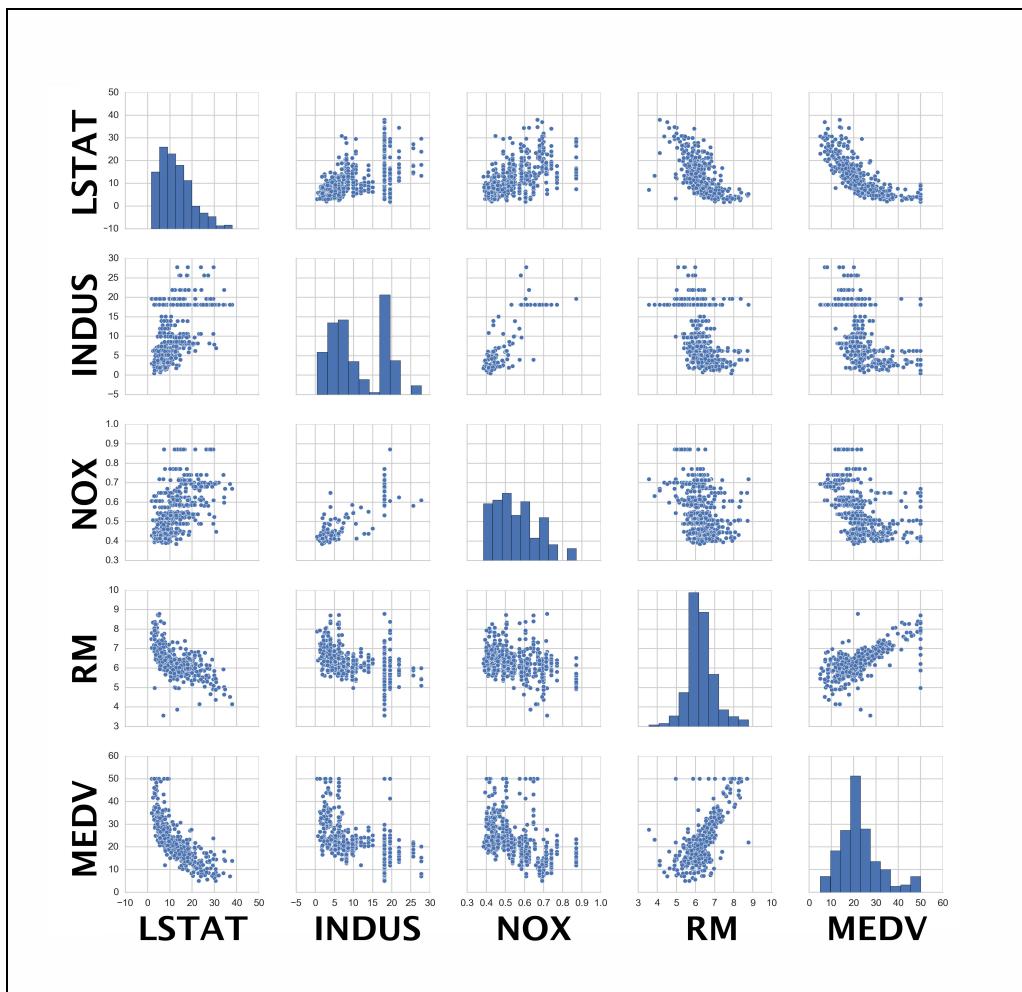
Eksploracyjna analiza danych (ang. *exploratory data analysis* — EDA) to ważny i zalecany pierwszy etap poprzedzający trening modelu uczenia maszynowego. W dalszej części podręcznika będziemy korzystać z prostych, ale użytecznych technik zawartych w graficznym narzędziu analizy EDA, dzięki którym łatwiej nam będzie wykrywać obecność odstających próbek, rozkład danych oraz relacje pomiędzy poszczególnymi cechami.

Najpierw utworzymy **macierz rozproszenia** (ang. *scatterplot matrix*) prezentującą w jednym miejscu związki pomiędzy parami poszczególnych cech zestawu danych. W tym celu wykorzystamy funkcję `pairplot` stanowiącą część biblioteki `seaborn` (<http://seaborn.pydata.org/>) — interfejsu środowiska Python bazującego na bibliotece `matplotlib`, umożliwiającego tworzenie wykresów statystycznych:

```
>>> import matplotlib.pyplot as plt
>>> import seaborn as sns
>>> sns.set(style='whitegrid', context='notebook')
```

```
>>> cols = ['LSTAT', 'INDUS', 'NOX', 'RM', 'MEDV']
>>> sns.pairplot(df[cols], size=2.5)
>>> plt.show()
```

Jak widzimy na rysunku 10.2, macierz wykresów stanowi przydatne graficzne podsumowanie zależności występujących w zbiorze danych.



Rysunek 10.2. Macierz wykresów dla pięciu cech zestawu Housing

Z powodu ograniczonego miejsca i chęci zachowania czytelności stworzyliśmy wykresy tylko pięciu kolumn zestawu danych: *LSTAT*, *INDUS*, *NOX*, *RM* i *MEDV*. Jeżeli jednak masz ochotę poznać pozostałe zależności, możesz wygenerować macierz rozproszenia dla wszystkich cech obiektu DataFrame.

Po importowaniu biblioteki seaborn ulega modyfikacji domyślny wygląd wykresów matplotlib w bieżącej sesji Pythona. Jeżeli nie zamierzasz korzystać z ustawień wyglądu biblioteki seaborn, możesz je zresetować za pomocą poniższego polecenia:

```
>>> sns.reset_orig()
```

Dzięki tej macierzy rozproszenia możemy szybko oszacować rozkład danych oraz obecność odstających próbek. Widzimy np., że istnieje liniowy związek pomiędzy cechami *RM* i *MEDV* (piąta kolumna, czwarty rząd). Co więcej, histogram (wykres kolumnowy w prawym dolnym rogu macierzy) informuje nas, że rozkład zmiennej *MEDV* jest normalny, ale zawiera ona kilka odstających próbek.

Zwróć uwagę, że, wbrew powszechnemu mniemaniu, uczenie modelu regresji liniowej nie wymaga rozkładu normalnego w przypadku zmiennych objaśniających ani docelowych. Założenie normalności jest wymagane jedynie wobec określonych testów statystycznych oraz testów hipotez, których opis wykracza poza ramy niniejszej książki (D.C. Montgomery, E.A. Peck i G.G. Vining, *Introduction to Linear Regression Analysis*, John Wiley and Sons, 2012, s. 318 – 319).

W celu ilościowego określenia zależności pomiędzy cechami stworzymy teraz macierz korelacji. Jest ona ściśle powiązana z macierzą kowariancji omówioną w rozdziale 4., „Tworzenie dobrych zbiorów uczących — wstępne przetwarzanie danych”, w ramach opisu **analizy głównych składowych (PCA)**. Możemy mianowicie uznać macierz korelacji za przeskalowaną wersję macierzy kowariancji. W rzeczywistości macierz korelacji daje te same wyniki, co macierz kowariancji obliczona ze standaryzowanych danych.

Macierz korelacji jest macierzą kwadratową składającą się ze **współczynników korelacji liniowej Pearsona** (ang. *Pearson product-moment correlation coefficients*), często zwanych **współczynnikami r-Pearsona**, mierzących liniową zależność pomiędzy parami cech. Wartości tych współczynników mieszczą się w zakresie pomiędzy  $-1$  a  $1$ . Dwie cechy mają idealną korelację pozytywną, jeśli  $r = 1$ , nie są ze sobą skorelowane przy wartości  $r = 0$ , a osiągają doskonałą korelację negatywną dla  $r = -1$ . Jak już wspomniałem, współczynnik korelacji Pearsona można z łatwością wyliczyć jako kowariancję dwóch cech  $x$  i  $y$  (licznik) podzieloną przez iloczyn ich odchyлеń standardowych (mianownik):

$$r = \frac{\sum_{i=1}^n [(x^{(i)} - \mu_x)(y^{(i)} - \mu_y)]}{\sqrt{\sum_{i=1}^n (x^{(i)} - \mu_x)^2} \sqrt{\sum_{i=1}^n (y^{(i)} - \mu_y)^2}} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$$

Parametr  $\mu$  oznacza tu wartości średnie próbek dla danej cechy,  $\sigma_{xy}$  stanowi kowariancję cech  $x$  i  $y$ , natomiast  $\sigma_x$  i  $\sigma_y$  to odchylenia standardowe tych cech.

Możemy udowodnić, że kowariancja standaryzowanych cech jest w istocie równa ich liniowemu współczynnikowi korelacji.

Przeprowadźmy najpierw standaryzację cech  $x$  i  $y$ , a wyniki oznaczmy jako  $x'$  i  $y'$ :

$$x' = \frac{x - \mu_x}{\sigma_x}, \quad y' = \frac{y - \mu_y}{\sigma_y}$$

Przypominam, że obliczamy kowariancję (populację) dwóch cech w następujący sposób:

$$\sigma_{xy} = \frac{1}{n} \sum_i^n (x^{(i)} - \mu_x)(y^{(i)} - \mu_y)$$

Standaryzacja wyśrodkowuje zmienną cech wobec wartości 0, dlatego możemy teraz wyliczyć kowariancję skalowanych cech przy użyciu wzoru:

$$\sigma'_{xy} = \frac{1}{n} \sum_i^n (x' - 0)(y' - 0)$$

W wyniku podstawienia uzyskujemy poniższy rezultat:

$$\frac{1}{n} \sum_i^n \left( \frac{x - \mu_x}{\sigma_x} \right) \left( \frac{y - \mu_y}{\sigma_y} \right)$$

$$\frac{1}{n \cdot \sigma_x \sigma_y} \sum_i^n (x^{(i)} - \mu_x)(y^{(i)} - \mu_y)$$

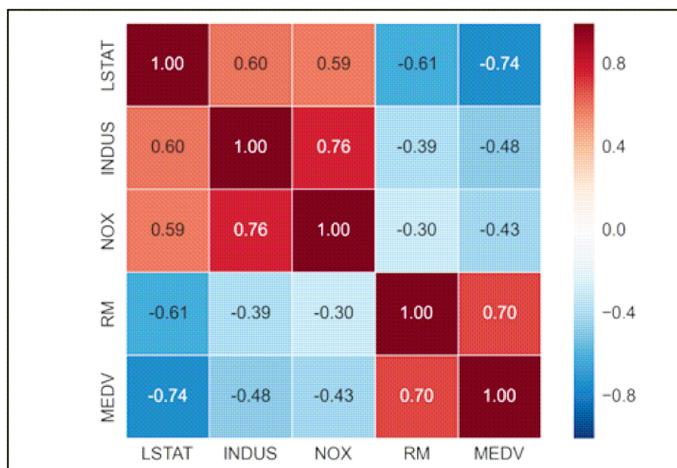
Możemy uprościć to do postaci:

$$\sigma'_{xy} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$$

W poniższym przykładzie wykorzystamy funkcję biblioteki NumPy, corrcoef, wobec pięciu uprzednio zwizualizowanych cech zestawu danych Housing, po czym za pomocą funkcji biblioteki seaborn, heatmap, wygenerujemy macierz rozproszenia w postaci mapy cieplnej:

```
>>> import numpy as np
>>> cm = np.corrcoef(df[cols].values.T)
>>> sns.set(font_scale=1.5)
>>> hm = sns.heatmap(cm,
...                   cbar=True,
...                   annot=True,
...                   square=True,
...                   fmt='.2f',
...                   annot_kws={'size': 15},
...                   yticklabels=cols,
...                   xticklabels=cols)
>>> plt.show()
```

Jak widać na rysunku 10.3, dzięki macierzy rozproszenia uzyskujemy kolejny przydatny wykres graficzny, pomagający dobrać cechy na podstawie ich wzajemnych powiązań.



Rysunek 10.3. Mapa cieplna pięciu cech zestawu Housing

W celu wycuczenia modelu regresji liniowej koncentrujemy się na cechach mających dużą korelację z naszą zmienną docelową *MEDV*. Dzięki widocznej na rysunku 10.3 mapie cieplnej wiemy, że zmienna *MEDV* wykazuje największą korelację z cechą *LSTAT* (-0,74). Jeśli jednak spojrzymy na macierz rozproszenia (rysunek 10.2), przekonamy się, że pomiędzy tymi zmiennymi istnieje wyraźna nieliniowa zależność. Z drugiej strony korelacja cech *RM* i *MEDV* również jest dość duża (0,7) i, jeśli wziąć pod uwagę występującą pomiędzy nimi liniową zależność, zmienna *RM* wydaje się bardzo dobrze nadawać do roli zmiennej objaśniającej, dzięki czemu możemy w następnym podrozdziale zająć się omówieniem koncepcji modelu prostej regresji liniowej.

## Implementacja modelu regresji liniowej wykorzystującego zwykłą metodę najmniejszych kwadratów

Na początku tego rozdziału stwierdziłem, że regresję liniową możemy postrzegać jako sposób wyszukiwania optymalnej prostej przechodzącej pomiędzy próbami danych uczących. Nie zdefiniowałem jednak, co mam na myśli poprzez **optymalną** ani nie wspomniałem o różnych technikach uczenia takiego modelu. W kolejnych ustępach uzupełnię te brakujące elementy układanki za pomocą **zwykłej metody najmniejszych kwadratów** (ang. *ordinary least squares* — **OLS**), służącej do szacowania parametrów linii regresyjnej poprzez minimalizację sumy kwadratów pionowych odległości (wartości rezydualnych/błędów) pomiędzy prostą a próbami.

## Określanie parametrów regresywnych za pomocą metody gradientu prostego

Wróćmy do implementacji ADaptacyjnego Liniowego NEuronu (**ADALINE**) omówionej w rozdziale 2., „Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji”; pamiętamy, że sztuczny neuron korzysta z liniowej funkcji aktywacji oraz że zdefiniowaliśmy funkcję kosztu  $J(\cdot)$ , którą minimalizowaliśmy w celu aktualizowania wag za pomocą algorytmów optymalizujących, takich jak metoda **gradientu prostego (GD)** czy **stochastyczny spadek wzduż gradientu (SGD)**. W modelu Adaline funkcją kosztu jest **suma kwadratów błędów (SSE)**. Jest ona identyczna z funkcją kosztu OLS:

$$J(w) = \frac{1}{2} \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2$$

Parametr  $\hat{y}$  to przewidywana wartość  $\hat{y} = w^T x$  (wartość  $1/2$  została użyta jedynie dla naszej wygody w celu wyprowadzenia reguły aktualizowania gradientu prostego). Zasadniczo regresję liniową **OLS** możemy postrzegać jako model Adaline pozbawiony funkcji skoku jednostkowego, dzięki czemu uzyskujemy docelowe wartości ciągle zamiast etykiet klas  $-1$  lub  $1$ . Zademonstruję to podobieństwo, usuwając funkcję skoku jednostkowego z zaprezentowanej w rozdziale 2., „Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji”, implementacji Adaline; w ten sposób otrzymamy nasz pierwszy model regresji liniowej:

```
class LinearRegressionGD(object):

    def __init__(self, eta=0.001, n_iter=20):
        self.eta = eta
        self.n_iter = n_iter

    def fit(self, X, y):
        self.w_ = np.zeros(1 + X.shape[1])
        self.cost_ = []

        for i in range(self.n_iter):
            output = self.net_input(X)
            errors = (y - output)
            self.w_[1:] += self.eta * X.T.dot(errors)
            self.w_[0] += self.eta * errors.sum()
            cost = (errors**2).sum() / 2.0
            self.cost_.append(cost)
        return self

    def net_input(self, X):
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def predict(self, X):
        return self.net_input(X)
```

Jeśli nie pamiętasz, na czym polega reguła aktualizowania wag — wykonywania kroku w kierunku przeciwnym do gradientu — zajrzyj do podrozdziału opisującego model Adaline w rozdziale 2., „Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji”.

W celu sprawdzenia działania regresora `LinearRegressionGD` wykorzystamy cechę *RM* (liczba pomieszczeń) jako zmienną objaśniającą w modelu przewidywania zmiennej *MEDV* (cen domów). Do tego przeprowadzimy standaryzację zmiennych, dzięki czemu uzyskamy większą zbieżność z algorytmem gradientu prostego. Wykorzystamy do tego poniższy kod:

```
>>> X = df[['RM']].values
>>> y = df['MEDV'].values
>>> from sklearn.preprocessing import StandardScaler
>>> sc_x = StandardScaler()
>>> sc_y = StandardScaler()
>>> X_std = sc_x.fit_transform(X)
>>> y_std = sc_y.fit_transform(y[:, np.newaxis]).flatten()
>>> lr = LinearRegressionGD()
>>> lr.fit(X_std, y_std)
```

W rozdziale 2., „Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji”, wyjaśniłem, że w czasie używania algorytmów optymalizujących, takich jak gradient prosty, zawsze warto generować wykres kosztu jako funkcji liczby epok (przebiegów algorytmu po zestawie danych uczących) w celu sprawdzenia zbieżności. Przejedźmy więc od razu do rzeczy — stworzymy wykres kosztu w funkcji epok i zobaczymy, czy model regresji liniowej osiągnął konwergencję:

```
>>> plt.plot(range(1, lr.n_iter+1), lr.cost_)
>>> plt.ylabel('Suma kwadratów błędów')
>>> plt.xlabel('Epoka')
>>> plt.show()
```

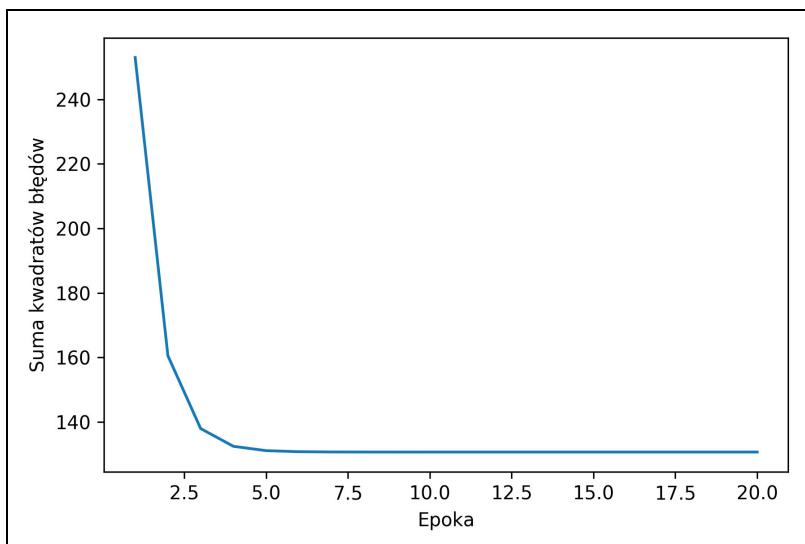
Jak widać na rysunku 10.4, algorytm gradientu prostego stał się zbieżny po pięciu epokach.

Zobaczmy teraz dopasowanie prostej regresywnej do danych uczących. Dokonamy tego, definiując prostą funkcję pomocniczą, która generuje wykres punktowy próbek uczących oraz dodaje tę linię:

```
>>> def lin_regplot(X, y, model):
...     plt.scatter(X, y, c='blue')
...     plt.plot(X, model.predict(X), color='red')
...     return None
```

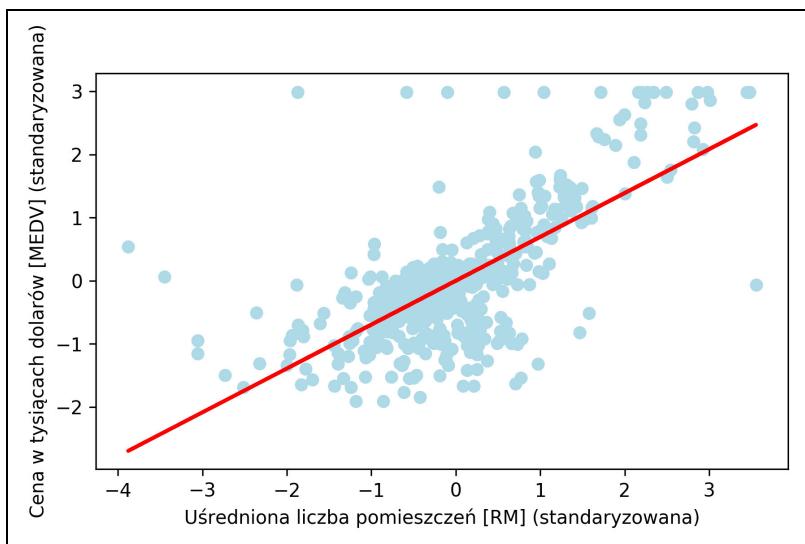
Użyjemy funkcji `lin_regplot` do narysowania wykresu liczby pomieszczeń w funkcji cen domów:

```
>>> lin_regplot(X_std, y_std, lr)
>>> plt.xlabel('Uśredniona liczba pomieszczeń [RM] (standaryzowana)')
>>> plt.ylabel('Cena w tysiącach dolarów [MEDV] (standaryzowana)')
>>> plt.show()
```



Rysunek 10.4. Wykres kosztu w funkcji epok dla modelu regresji liniowej

Zgodnie z wykresem ukazanym na rysunku 10.5 prosta regresji liniowej odzwierciedla ogólny trend wzrostu cen domów wraz z liczbą pomieszczeń.



Rysunek 10.5. Wykres zależności cen domów od liczby pomieszczeń z nałożoną prostą regresji liniowej

Powyzsza obserwacja jest całkiem logiczna, jednak z danych na wykresie wynika również, że w wielu przypadkach liczba pomieszczeń nie tłumaczy dobrze cen domów. W dalszej części rozdziału pokażę, w jaki sposób przedstawić ilościowo skuteczność modelu regresywnego.

Co ciekawe, w osi  $y = 3$  widzimy szereg punktów tworzących prostą, co może oznaczać, że ceny niektórych domów zostały zawyżone. W pewnych zastosowaniach ważną rolę odgrywać może również podawanie przewidywanych zmiennych w ich pierwotnej skali. Aby dopasować otrzymywane wyniki cen do osi *Cena w tysiącach dolarów*, wystarczy wprowadzić metodę `inverse_transform` będącą częścią klasy `StandardScaler`:

```
>>> num_rooms_std = sc_x.transform(np.array([[5.0]]))
>>> price_std = lr.predict(num_rooms_std)
>>> print("Cena w tysiącach dolarów: %.3f" % \
...       sc_y.inverse_transform(price_std))
Cena w tysiącach dolarów: 10.840
```

W powyższym fragmencie kodu wykorzystaliśmy wyuczony model regresji liniowej do prognozowania ceny pięciopokojowych domów. Zgodnie z wyliczeniami modelu dom taki jest wart 10 840 dolarów.

Na marginesie warto wspomnieć, że technicznie rzecz biorąc, jeżeli pracujemy na standaryzowanych zmiennych, nie musimy aktualizować wag, ponieważ w takim przypadku punkt przecięcia z osią  $y$  zawsze jest równy 0. Możemy to szybko potwierdzić, wyświetlając wartości wag:

```
>>> print('Nechylenie: %.3f' % lr.w_[1])
Nechylenie: 0.695
>>> print('Punkt przecięcia: %.3f' % lr.w_[0])
Punkt przecięcia: -0.000
```

## Szacowanie współczynnika modelu regresji za pomocą biblioteki scikit-learn

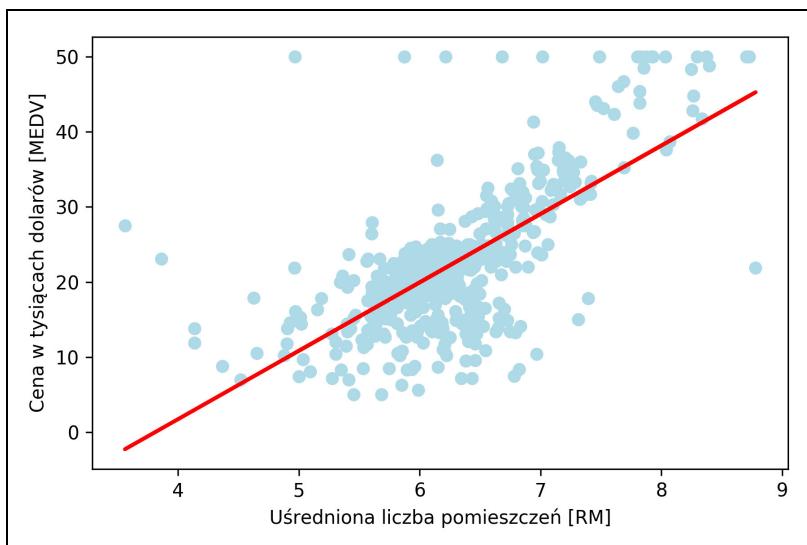
W poprzednim podrozdziale zaimplementowaliśmy działający model analizy regresywnej. Jednak w rzeczywistych zastosowaniach mogą nas interesować skuteczniejsze implementacje, takie jak obiekt biblioteki scikit-learn `LinearRegression` wykorzystujący bibliotekę **LIBLINEAR** oraz zaawansowane algorytmy optymalizujące działające skuteczniej na niestandardyzowanych zmiennych. Jest to wręcz pożądane w niektórych sytuacjach:

```
>>> from sklearn.linear_model import LinearRegression
>>> slr = LinearRegression()
>>> slr.fit(X, y)
>>> print('Nechylenie: %.3f' % slr.coef_[0])
Nechylenie: 9.102
>>> print('Punkt przecięcia: %.3f' % slr.intercept_)
Punkt przecięcia: -34.671
```

Po uruchomieniu powyższego fragmentu kodu algorytm `LinearRegression` wyuczony na niestandardyzowanych zmiennych *RM* i *MEDV* uzyskał odmienne współczynniki modelu w porównaniu z wartościami otrzymanymi w poprzednim podrozdziale. Porównajmy je z implementacją gradientu prostego, tworząc wykres *MEDV* w funkcji *RM*:

```
>>> lin_regplot(X, y, slr)
>>> plt.xlabel('Uśredniona liczba pomieszczeń [RM]')
>>> plt.ylabel('Cena w tysiącach dolarów [MEDV]')
>>> plt.show()
```

Dzięki powyższemu fragmentowi kodu otrzymaliśmy wykres danych uczących wraz z prostą reprezentującą wytrenowany model (rysunek 10.6) i widzimy, że rezultat jest identyczny jak w przypadku implementacji gradientu prostego.



Rysunek 10.6. Wykres modelu regresji liniowej uzyskanego za pomocą klasy LinearRegression

Rozwiązańiem alternatywnym dla korzystania z bibliotek uczenia maszynowego jest mechanizm w postaci jawnego rozwiązywania zwykłej metody najmniejszych kwadratów, wykorzystujący układ równań nielinijnych (metodę tę można znaleźć w większości podręczników statystyki):

$$w = (X^T X)^{-1} X^T y$$

W Pythonie możemy zaimplementować tę metodę następująco:

```
# dodaje wektor kolumnowy zawierający "jedynki"
>>> Xb = np.hstack((np.ones((X.shape[0], 1)), X))
>>> w = np.zeros(X.shape[1])
>>> z = np.linalg.inv(np.dot(Xb.T, Xb))
>>> w = np.dot(z, np.dot(Xb.T, y))
>>> print('Nachylenie: %.3f' % w[1])
Nachylenie: 9.102
>>> print('Punkt przecięcia: %.3f' % w[0])
Punkt przecięcia: -34.671
```

Zaletą tego rozwiązania jest pewność znalezienia optymalnego rozwiązania w analityczny sposób. Jeżeli jednak pracujemy na bardzo dużych zestawach danych, proces odwrócenia macierzy w tym wzorze (zwany czasami **równaniem normalnym**) może być zbyt kosztowny obliczeniowo lub macierz próbek może być osobienna (nieodwzracalna), dlatego w pewnych sytuacjach lepsze okazują się metody przyrostowe.

Osobom zainteresowanym sposobami otrzymywania równań normalnych polecam zapoznanie się z rozdziałem 6, *The Classical Linear Regression Model*, stanowiącym część wykładów autorstwa dra Stephena Pollocka dla Uniwersytetu w Leicester, umieszczonym pod adresem <http://www.le.ac.uk/users/dsgp1/COURSES/MESOMET/ECMETXT/06mesmet.pdf>.

## Uczenie odpornego modelu regresywnego za pomocą algorytmu RANSAC

Wyniki modeli regresji liniowej zostają często wypaczone przez odstające próbki (ang. *outliers*). W pewnych sytuacjach niewielki podzbior danych może mieć olbrzymi wpływ na szacowane wartości współczynników modelu. Istnieje wiele testów statystycznych wykrywających obecność odstających próbek, ich omówienie jednak wykracza poza zakres niniejszej książki. Z kolei usuwanie takich odstających próbek zawsze wymaga naszego osądów jako analityków danych, jak również wiedzy z zakresu badanej dziedziny.

Zamiast wyrzucać odstające próbki, przyjrzymy się odpornemu modelowi regresji wykorzystującemu algorytm **konsensusu próby losowej** (ang. *RANdom SAmple Consensus — RANSAC*), dopasowującemu model do podzbioru tzw. **danych nieodstających** (ang. *inliers*).

Możemy podsumować mechanizm działania przyrostowego algorytmu RANSAC w następujący sposób:

1. Wybierz losową liczbę próbek do roli danych nieodstających i dopasuj model.
2. Przetestuj wszystkie pozostałe punkty danych wobec wyuczonego modelu i dodaj wszystkie próbki mieszczące się w zakresie tolerancji do podzbioru danych nieodstających.
3. Wytrenuj ponownie model przy użyciu kompletnego podzbioru danych odstających.
4. Oszacuj błąd wyuczonego modelu dla danych odstających.
5. Zakończ algorytm, jeśli jego skuteczność przekracza wartość progową zdefiniowaną przez użytkownika lub jeśli została osiągnięta ustalona liczba iteracji; w przeciwnym wypadku wróć do etapu 1.

<sup>1</sup> Z kolei wśród materiałów dostępnych w języku polskim warto zajrzeć do skryptu *Analiza regresji i analiza wariancji* napisanego przez prof. Aleksandra Zaigrajewa dla Uniwersytetu Mikołaja Kopernika w Toruniu (<http://www-users.mat.umk.pl/~alzaig/araw.pdf>) — przyp. tłum.

Umieścimy teraz nasz liniowy model wewnątrz algorytmu RANSAC przy użyciu obiektu RANSACRegressor (część interfejsu scikit-learn):

```
>>> from distutils.version import LooseVersion as Version
>>> from sklearn import __version__ as sklearn_version
>>> from sklearn.linear_model import RANSACRegressor
>>> if Version(sklearn_version) < '0.18':
...     ransac = RANSACRegressor(LinearRegression(),
...                             max_trials=100,
...                             min_samples=50,
...                             residual_metric=lambda x: np.sum(np.abs(x),
...                                axis=1),
...                             residual_threshold=5.0,
...                             random_state=0)
>>> else:
...     ransac = RANSACRegressor(LinearRegression(),
...                             max_trials=100,
...                             min_samples=50,
...                             loss='absolute_loss',
...                             residual_threshold=5.0,
...                             random_state=0)
>>> ransac.fit(X, y)
```

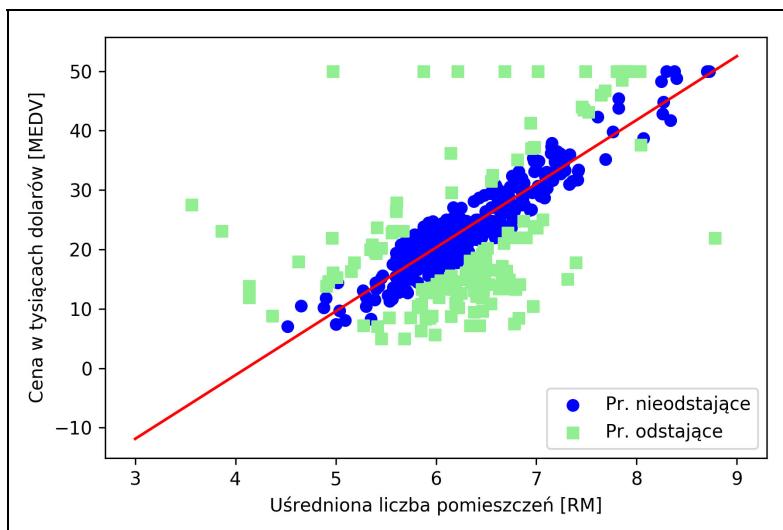
Wyznaczamy algorytmowi RANSACRegressor maksymalną liczbę iteracji równą 100 i za pomocą parametru `min_samples=50` definiujemy minimalną liczbę losowo dobieranych próbek (co najmniej 50). Dzięki parametrowi `residual_metric` wprowadzamy wywoływalną funkcję `lambda`, której zadaniem jest wyliczanie bezwzględnej pionowej odległości pomiędzy wyznaczoną prostą regresywną a punktami próbek. Wprowadzając wartość 5.0 w parametrze `residual_threshold`, dopuszczaemy do podzbioru danych nieodstających jedynie te próbki, których odległość od prostej mieści się w zakresie pięciu jednostek, co się bardzo dobrze sprawdza w przypadku naszego zestawu próbek. Domyślnie biblioteka scikit-learn wykorzystuje oszacowanie MAD w trakcie dobierania progu próbek nieodstających, gdzie skrót **MAD** oznacza **średnie odchylenie bezwzględne** (ang. *Median Absolute Deviation*) docelowych wartości `y`. Jednak wybór odpowiedniej wartości progowej dla próbek nieodstających zależy w dużej mierze od analizowanego problemu, co stanowi jedną z wad algorytmu RANSAC. W ciągu ostatnich lat zaprojektowano wiele różnych sposobów pozwalających na automatyczny dobór właściwej wartości progowej. Szczegółowe informacje na ten temat znajdziesz w artykule autorstwa Roberta Toldo i Andrei Fusiello, *Automatic Estimation of the Inlier Threshold in Robust Multiple Structures Fitting* [w:] *Image Analysis and Processing — ICIAP 2009*, Springer, 2009, s. 123 – 131).

Gdy już wyuczyszmy model RANSAC, zobaczymy, jak wyglądają nasze odstające i nieodstające próbki wyznaczone przez algorytm regresji liniowej, poprzez ich umieszczenie na wykresie wraz z prostą regresywną:

```
>>> inlier_mask = ransac.inlier_mask
>>> outlier_mask = np.logical_not(inlier_mask)
>>> line_X = np.arange(3, 10, 1)
>>> line_y_ransac = ransac.predict(line_X[:, np.newaxis])
```

```
>>> plt.scatter(X[inlier_mask], y[inlier_mask],
...                 c='blue', marker='o', label='Pr. nieodstające')
>>> plt.scatter(X[outlier_mask], y[outlier_mask],
...                 c='lightgreen', marker='s', label='Pr. odstające')
>>> plt.plot(line_X, line_y_ransac, color='red')
>>> plt.xlabel('Uśredniona liczba pomieszczeń [RM]')
>>> plt.ylabel('Cena w tysiącach dolarów [MEDV]')
>>> plt.legend(loc='lower right')
>>> plt.show()
```

Zgodnie z wykresem zaprezentowanym na rysunku 10.7 model regresji liniowej został wytrewnowany za pomocą wykrytego zestawu próbek nieodstających, symbolizowanych kółkami.



**Rysunek 10.7.** Wykres modelu regresji liniowej z zaznaczonymi próbками odstającymi (kwadraty) i nieodstającymi (kółka)

Po wyświetleniu wartości nachylenia i punktu przecięcia modelu przekonamy się, że prosta regresji liniowej nieznacznie różni się od wyniku uzyskanego w modelu niewykorzystującym algorytmu RANSAC:

```
>>> print('Nachylenie: %.3f' % ransac.estimator_.coef_[0])
Nachylenie: 9.621
>>> print('Punkt przecięcia: %.3f' % ransac.estimator_.intercept_)
Punkt przecięcia: -37.137
```

Dzięki algorytmowi RANSAC zmniejszyliśmy potencjalny wpływ odstających próbek na nasz zestaw danych, nie wiemy jednak, czy to rozwiązanie będzie miało pozytywny wpływ na skuteczność predykcyjną wobec nieznanych danych. Dlatego w następnym podrozdziale zajmiemy się dyskusją na temat oceny różnych typów modelu regresywnego, co stanowi zasadniczy element tworzenia układów prognozujących.

## Ocenianie skuteczności modeli regresji liniowej

W poprzednim podrozdziale nauczyliśmy się trenować model regresywny za pomocą danych uczących. Wiemy jednak z poprzednich podrozdziałów, że niezmiernie istotnym etapem jest testowanie modelu wobec danych niewykorzystanych na etapie uczenia, co pozwala uzyskać wiarygodne oszacowanie jego skuteczności.

Jak pamiętamy z rozdziału 6., „Najlepsze metody oceny modelu i strojenie hiperparametryczne”, chcemy rozdzielić nasz zestaw danych na podzbiory próbek uczących i testowych — te pierwsze służą do trenowania modelu, a drugie pozwalają ocenić jego skuteczność uogólniania wobec nieznanych danych. Przejdziemy teraz z prostego modelu regresji na model wielowymiarowy, który wyuczymy za pomocą wszystkich próbek zestawu danych.

```
>>> from distutils.version import LooseVersion as Version
>>> from sklearn import __version__ as sklearn_version
>>> if Version(sklearn_version) < '0.18':
...     from sklearn.cross_validation import train_test_split
>>> else:
...     from sklearn.model_selection import train_test_split
>>> X = df.iloc[:, :-1].values
>>> y = df['MEDV'].values
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=0.3, random_state=0)
>>> slr = LinearRegression()
>>> slr.fit(X_train, y_train)
>>> y_train_pred = slr.predict(X_train)
>>> y_test_pred = slr.predict(X_test)
```

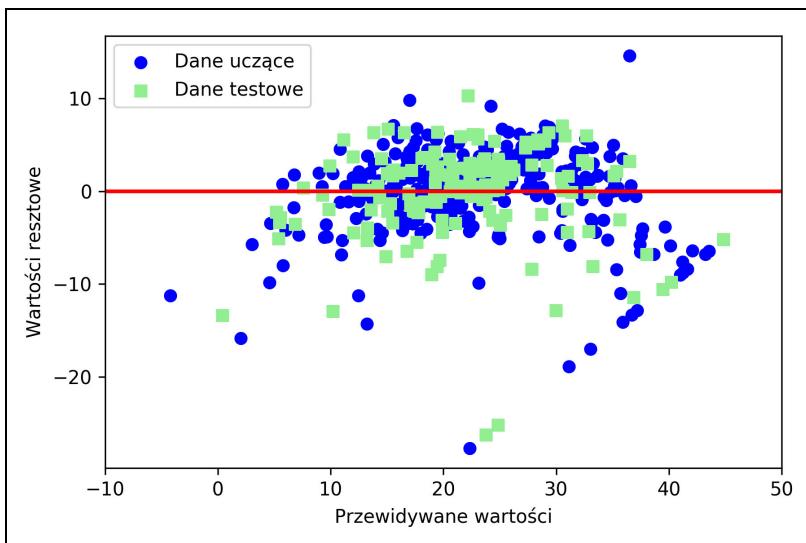
W naszym modelu wykorzystujemy wiele zmiennych objaśniających, dlatego nie jesteśmy w stanie wyświetlić prostej (a właściwie hiperplasycznej) regresji liniowej na dwuwymiarowym wykresie, ale możemy wygenerować wykres wartości resztowych (równolegleych do osi  $y$  odległości pomiędzy przewidywanymi a rzeczywistymi wartościami) w funkcji prognozowanych wartości w celu zdiagnozowania modelu regresji. Takie **wykresy resztowe** (rezydualne) są powszechnie stosowaną metodą analizy graficznej pozwalającą na diagnozowanie skuteczności modeli regresywnych, wykrywającą nieliniowość oraz odstające próbki, a także sprawdzającą, czy rozkład błędów jest normalny.

Za pomocą poniższego kodu, odejmując rzeczywiste zmienne docelowe od przewidywanych odpowiedzi, stworzymy wykres resztowy:

```
>>> plt.scatter(y_train_pred, y_train_pred - y_train,
...               c='blue', marker='o', label='Dane uczące')
>>> plt.scatter(y_test_pred, y_test_pred - y_test,
...               c='lightgreen', marker='s', label='Dane testowe')
>>> plt.xlabel('Przewidywane wartości')
```

```
>>> plt.ylabel('Wartości resztowe')
>>> plt.legend(loc='upper left')
>>> plt.hlines(y=0, xmin=-10, xmax=50, lw=2, color='red')
>>> plt.xlim([-10, 50])
>>> plt.show()
```

Po uruchomieniu powyższego kodu naszym oczom powinien ukazać się zaprezentowany na rysunku 10.8 wykres resztowy, w którym przez początek układu współrzędnych przebiega prosta równoległa do osi  $x$ .



Rysunek 10.8. Wykres resztowy analizowanego modelu regresji

W przypadku idealnej predykcji wartości resztowe byłyby zerowe, co prawdopodobnie nigdy nie ma miejsca w rzeczywistych oraz praktycznych zastosowaniach. Spodziewalibyśmy się jednak po dobrym modelu regresji, że błędy będą losowo występować zgodnie z rozkładem normalnym, a wartości resztowe również będą losowo rozmiieszczane wokół środkowej linii. Jeżeli na wykresie resztowym będziemy dostrzegać jakieś wzorce, to znaczy, że nasz model nie jest w stanie uchwycić jakichś informacji objaśniających, które zostają przeniesione na wartości resztowe, co możemy w nieznacznym stopniu dostrzec na rysunku 10.8. Ponadto wykres resztowy pozwala wyznaczać próbki odstające, czyli w tym przypadku punkty znacznie oddalone od środkowej prostej.

Innym przydatnym pomiarem ilościowym skuteczności modelu jest tzw. **błąd średniokwadratowy** (ang. *mean squared error* — MSE), czyli uśredniona wartość funkcji kosztu SSE, którą minimalizujemy podczas trenowania modelu regresji liniowej. Algorytm MSE przydaje się do porównywania różnych modeli regresywnych lub w trakcie ich dostrajania za pomocą przeszukiwania siatki/sprawdzianu krzyżowego:

$$MSE = \frac{1}{n} \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2$$

Uruchom następujący kod:

```
>>> from sklearn.metrics import mean_squared_error
>>> print('MSE na próbkach uczących: %.3f, testowych: %.3f' %
...     mean_squared_error(y_train, y_train_pred),
...     mean_squared_error(y_test, y_test_pred)))
```

Zobaczmy, że wynik uzyskany metodą *MSE* dla próbek uczących wynosi 19,96, natomiast dla danych testowych osiąga znacznie wyższą wartość — 27,2 — co wskazuje na przetrenowanie modelu.

Czasami bardziej opłacalne jest wykorzystanie współczynnika determinacji ( $R^2$ ), który możemy zdefiniować jako standaryzowaną wersję metody *MSE*, co pozwala na lepszą interpretację skuteczności modelu. Innymi słowy, współczynnik  $R^2$  stanowi ulamek wariancji odpowiedzi, zawarty w modelu. Wartość współczynnika determinacji obliczamy w następujący sposób:

$$R^2 = 1 - \frac{SSE}{SST}$$

*SSE* oznacza tu sumę kwadratów błędów, a *SST* całkowitą sumę kwadratów

$$SST = \sum_{i=1}^n (y^{(i)} - \mu_y)^2,$$

czyli, krótko mówiąc, wariancję odpowiedzi.

Sprawdźmy, czy współczynnik  $R^2$  rzeczywiście stanowi przeskalowaną wersję metody *MSE*:

$$\begin{aligned} R^2 &= 1 - \frac{SSE}{SST} \\ &= 1 - \frac{\frac{1}{n} \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2}{\frac{1}{n} \sum_{i=1}^n (y^{(i)} - \mu_y)^2} \\ &= 1 - \frac{MSE}{Var(y)} \end{aligned}$$

Dla zestawu danych uczących współczynnik  $R^2$  osiąga wartości w zakresie od 0 do 1, ale w przypadku próbek testowych może uzyskiwać wartość ujemną. Jeżeli  $R^2 = 1$ , to model jest idealnie dopasowany do danych przy jednoczesnej wartości  $MSE = 0$ .

Po sprawdzeniu współczynnika  $R^2$  dla danych uczących uzyskujemy wartość 0,765, czyli nie jest źle. Jednak ten sam współczynnik dla danych testowych daje wartość zaledwie 0,673, o czym możemy się przekonać, uruchomiony poniższy fragment kodu:

```
>>> from sklearn.metrics import r2_score
>>> print('Współczynnik R^2 dla danych uczących: %.3f, testowych: %.3f' %
...      (r2_score(y_train, y_train_pred),
...      r2_score(y_test, y_test_pred)))
```

## Stosowanie regularyzowanych metod regresji

Zgodnie z informacjami zawartymi w rozdziale 3., „Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn”, regularyzacja stanowi jeden ze sposobów radzenia sobie z przetrenowaniem poprzez wprowadzenie dodatkowych informacji, a zatem zmniejszanie wag parametrów modelu (zwiększenie kary za jego złożoność). Najpopularniejszymi metodami regularyzowanej regresji liniowej są **regresja grzbietowa**, **regresja metodą LASSO** (ang. *Least Absolute Shrinkage and Selection Operator* — w wolnym tłumaczeniu operator najmniejszej bezwzględnej redukcji i wyboru) oraz **metoda elastycznej siatki**.

Regresja grzbietowa (ang. *ridge regression*) to model regularyzacji typu L2, w którym dodajemy kwadrat sumy wag do funkcji kosztu obliczonej metodą najmniejszych kwadratów:

$$J(w)_{\text{grzbietowa}} = \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2 + \lambda \|w\|_2^2$$

Tutaj:

$$\text{L2: } \lambda \|w\|_2^2 = \lambda \sum_{j=1}^m w_j^2$$

Wraz ze wzrostem hiperparametru zwiększamy siłę regularizacji i zmniejszamy wagi modelu. Zwróć uwagę, że nie regularyzujemy wagi początkowej (punktu przecięcia)  $w_0$ .

Alternatywną metodą prowadzącą do modeli rzadkich jest algorytm LASSO. W zależności od siły regularizacji pewne wagi mogą uzyskać zerową wartość, co sprawia, że omawiana technika przydaje się w nadzorowanym wyborze cech:

$$J(w)_{\text{LASSO}} = \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2 + \lambda \|w\|_1$$

Tutaj:

$$\text{L1: } \lambda \|w\|_1 = \lambda \sum_{j=1}^m |w_j|$$

Jednak ograniczeniem algorytmu LASSO jest dobór co najwyżej  $n$  zmiennych, jeśli  $m > n$ . Kompromis pomiędzy regresją grzbietową a algorytmem LASSO stanowi technika elastycznej siatki (ang. *elastic net*), w której kara L1 służy do generowania rzadkości, a kara L2 pozwala przewyciążyć pewne ograniczenia algorytmu LASSO, takie jak liczba dobieranych zmiennych.

$$J(w)_{\text{ElastycznaSiatka}} = \sum_{i=1}^n \left( y^{(i)} - \hat{y}^{(i)} \right)^2 + \lambda_1 \sum_{j=1}^m w_j^2 + \lambda_2 \sum_{j=1}^m |w_j|$$

Wszystkie te regularyzowane modele regresji są dostępne w bibliotece scikit-learn, a sposób ich użycia przypomina stosowanie klasycznego modelu regresji; jedyną różnicą jest konieczność zdefiniowania siły regularyzacji za pomocą parametru  $\lambda$ , optymalizowanego np. poprzez k-krotny sprawdzian krzyżowy.

Regresję grzbietową inicjujemy w następujący sposób:

```
>>> from sklearn.linear_model import Ridge
>>> ridge = Ridge(alpha=1.0)
```

Zwróć uwagę, że siłę regularyzacji kontrolujemy tu za pomocą parametru `alpha`, przypominającego w dużej mierze parametr  $\lambda$ . W podobny sposób możemy inicjować regresor LASSO stanowiący część podmodułu `linear_model`:

```
>>> from sklearn.linear_model import Lasso
>>> lasso = Lasso(alpha=1.0)
```

Została nam jeszcze implementacja klasy `ElasticNet`, w której możemy zdefiniować stosunek kar L1 do L2:

```
>>> from sklearn.linear_model import ElasticNet
>>> lasso = ElasticNet(alpha=1.0, l1_ratio=0.5)
```

Jeżeli np. wyznaczymy wartość 1.0 parametru `l1_ratio`, regresor `ElasticNet` przyjmie postać regresji LASSO. Więcej szczegółowych informacji na temat różnych implementacji regresji liniowej znajdziesz w dokumentacji pod adresem [http://scikit-learn.org/stable/modules/linear\\_model.html](http://scikit-learn.org/stable/modules/linear_model.html).

## Przekształcanie modelu regresji liniowej w krzywą — regresja wielomianowa

W poprzednich podrozdziałach zakładaliśmy istnienie liniowej zależności pomiędzy zmiennymi objaśniającymi a zmiennymi odpowiedzi. Jednym ze sposobów rozwiązania sytuacji, w której naruszone zostało założenie liniowości, jest wprowadzenie modelu regresji wielomianowej poprzez dodanie wielomianowych członów:

$$y = w_0 + w_1 x + w_2 x^2 + \dots + w_d x^d$$

Symbol  $d$  oznacza tu stopień wielomianu. Regresja wielomianowa służy głównie do modelowania nieliniowych zależności, mimo to uznajemy ją za wielowymiarowy model regresji z powodu obecności współczynników liniowych  $w$ .

Przeanalizujemy teraz sposób wykorzystania klasy transformującej `PolyomialFeatures` z biblioteki scikit-learn do dodania wyrażenia kwadratowego ( $d = 2$ ) w prostym problemie regresywnym z jedną zmienną objaśniającą, a następnie porównamy dopasowanie wielomianowe z liniowym. Poszczególne etapy wyglądają następująco:

1. Dodanie wyrażenia wielomianowego drugiego stopnia:

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> X = np.array([258.0, 270.0, 294.0,
...                 320.0, 342.0, 368.0,
...                 396.0, 446.0, 480.0,
...                 586.0])[:, np.newaxis]
>>> y = np.array([236.4, 234.4, 252.8,
...                 298.6, 314.2, 342.2,
...                 360.8, 368.0, 391.2,
...                 390.8])
>>> lr = LinearRegression()
>>> pr = LinearRegression()
>>> quadratic = PolynomialFeatures(degree=2)
>>> X_quad = quadratic.fit_transform(X)
```

2. Wyuczenie prostego modelu regresji liniowej w celach porównawczych:

```
>>> lr.fit(X, y)
>>> X_fit = np.arange(250, 600, 10)[:, np.newaxis]
>>> y_lin_fit = lr.predict(X_fit)
```

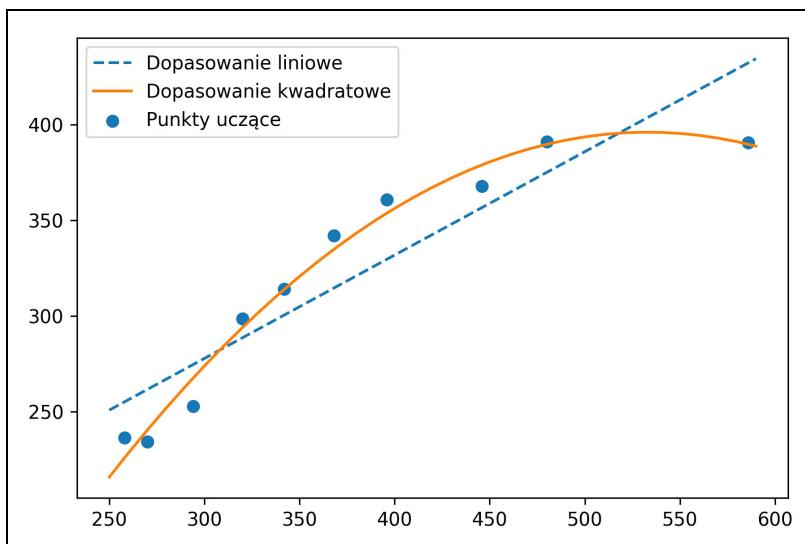
3. Wyuczenie wielowymiarowego modelu regresji na przekształconych cechach dostosowanych do regresji wielomianowej:

```
>>> pr.fit(X_quad, y)
>>> y_quad_fit = pr.predict(quadratic.fit_transform(X_fit))
```

4. Wygenerowanie wynikowego wykresu:

```
>>> plt.scatter(X, y, label='Punkty uczące')
>>> plt.plot(X_fit, y_lin_fit,
...             label='Dopasowanie liniowe', linestyle='--')
>>> plt.plot(X_fit, y_quad_fit,
...             label='Dopasowanie kwadratowe')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Widzimy na uzyskanym wykresie (rysunek 10.9), że dopasowanie kwadratowe znacznie skuteczniej niż dopasowanie liniowe wykrywa powiązania pomiędzy danymi objaśniającymi i objaśnianymi.



Rysunek 10.9. Wykres porównujący regresję liniową z regresją wielomianową

```
>>> y_lin_pred = lr.predict(X)
>>> y_quad_pred = pr.predict(X_quad)
>>> print('Liniowe trenowanie metodą MSE: %.3f, kwadratowe: %.3f' %
... mean_squared_error(y, y_lin_pred),
... mean_squared_error(y, y_quad_pred))
Liniowe trenowanie metodą MSE: 569.780, kwadratowe: 61.330
>>> print('Liniowe trenowanie metodą R^2: %.3f, kwadratowe: %.3f' %
... r2_score(y, y_lin_pred),
... r2_score(y, y_quad_pred))
Liniowe trenowanie metodą R^2: 0.832, kwadratowe: 0.982
```

Jak widać po uruchomieniu powyższego kodu, wartość wyniku MSE spadła z 570 (dopasowanie liniowe) do 61 (dopasowanie kwadratowe), a współczynnik determinacji przedstawia większe dopasowanie modelu kwadratowego ( $R^2 = 0,982$ ) od modelu liniowego ( $R^2 = 0,832$ ) w tym konkretnym przykładzie.

## Modelowanie nieliniowych zależności w zestawie danych Housing

Gdy już wiemy, jak przygotowywać nieliniowe cechy do wykrywania nieliniowych zależności, możemy przejść do rzeczywistego przykładu i wykorzystać zdobyte informacje na zestawie danych **Housing**. Dzięki poniższemu kodowi wymodelujemy relacje pomiędzy cenami domów a parametrem *LSTAT* (odsetek uboższej części populacji) za pomocą wielomianów drugiego (kwadratowych) i trzeciego (sześciennych) stopnia oraz porównamy wyniki z rezultatami modelu liniowego.

Oto wspomniany kod:

```

>>> X = df[['LSTAT']].values
>>> y = df['MEDV'].values
>>> regr = LinearRegression()

# tworzy wielomianowe cechy
>>> quadratic = PolynomialFeatures(degree=2)
>>> cubic = PolynomialFeatures(degree=3)
>>> X_quad = quadratic.fit_transform(X)
>>> X_cubic = cubic.fit_transform(X)

# dopasowanie liniowe
>>> X_fit = np.arange(X.min(), X.max(), 1)[:, np.newaxis]
>>> regr = regr.fit(X, y)
>>> y_lin_fit = regr.predict(X_fit)
>>> linear_r2 = r2_score(y, regr.predict(X))

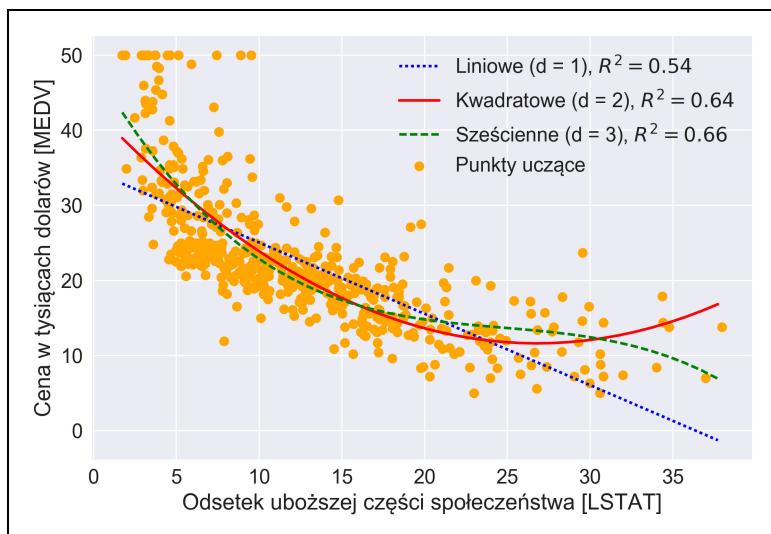
# dopasowanie kwadratowe
>>> regr = regr.fit(X_quad, y)
>>> y_quad_fit = regr.predict(quadratic.fit_transform(X_fit))
>>> quadratic_r2 = r2_score(y, regr.predict(X_quad))

# dopasowanie sześciennne
>>> regr = regr.fit(X_cubic, y)
>>> y_cubic_fit = regr.predict(cubic.fit_transform(X_fit))
>>> cubic_r2 = r2_score(y, regr.predict(X_cubic))

# tworzy wynikowy wykres
>>> plt.scatter(X, y,
...                 label='Punkty uczące',
...                 color='lightgray')
>>> plt.plot(X_fit, y_lin_fit,
...                 label='Liniowe (d=1), $R^2=% .2f$',
...                 % linear_r2,
...                 color='blue',
...                 lw=2,
...                 linestyle=':')
>>> plt.plot(X_fit, y_quad_fit,
...                 label='Kwadratowe (d=2), $R^2=% .2f$',
...                 % quadratic_r2,
...                 color='red',
...                 lw=2,
...                 linestyle='--')
>>> plt.plot(X_fit, y_cubic_fit,
...                 label='Sześciennne (d=3), $R^2=% .2f$',
...                 % cubic_r2,
...                 color='green',
...                 lw=2,
...                 linestyle='--')
```

```
>>> plt.xlabel('Odsetek uboższej części społeczeństwa [LSTAT]')
>>> plt.ylabel('Cena w tysiącach dolarów [MEDV]')
>>> plt.legend(loc='upper right')
>>> plt.show()
```

Zgodnie z wykresem zaprezentowanym na rysunku 10.10 dopasowanie sześcienne wyłapuje związek pomiędzy cenami domów a cechą *LSTAT* skuteczniej niż dopasowanie liniowe lub kwadratowe. Pamiętajmy jednak, że wraz z dodatkowym wielomianowym cech wzrasta złożoność modelu, a zatem ryzyko przetrenowania. Dlatego w praktyce zawsze zaleca się ocenę skuteczności modelu na osobnym zestawie danych testowych w celu oszacowania skuteczności uogólniania.



Rysunek 10.10. Porównanie dopasowania liniowego, kwadratowego i sześciennego na zestawie danych Housing

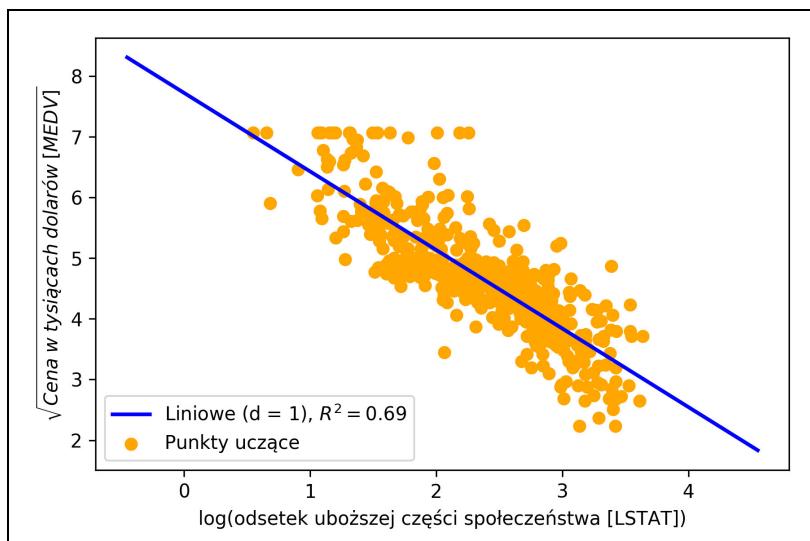
Poza tym cechy wielomianowe nie zawsze stanowią najlepszy wybór w modelowaniu nielinowych relacji. Wystarczy spojrzeć np. na wykres zależności *MEDV* – *LSTAT*, aby stwierdzić, że w tym przypadku transformacja logarytmiczna zmiennej *LSTAT* oraz pierwiastkowanie cechy *MEDV* pozwoliłyby rzutować dane na liniową przestrzeń cech, umożliwiającą liniowe dopasowanie modelu. Sprawdźmy tę hipotezę za pomocą poniższego kodu:

```
# przekształca cechy
>>> X_log = np.log(X)
>>> y_sqrt = np.sqrt(y)

# dopasowuje cechy
>>> X_fit = np.arange(X_log.min()-1,
...                     X_log.max()+1, 1)[:, np.newaxis]
>>> regr = regr.fit(X_log, y_sqrt)
>>> y_lin_fit = regr.predict(X_fit)
>>> linear_r2 = r2_score(y_sqrt, regr.predict(X_log))
```

```
# tworzy wykres
>>> plt.scatter(X_log, y_sqrt,
...                 label='Punkty uczące',
...                 color='lightgray')
>>> plt.plot(X_fit, y_lin_fit,
...                 label='Liniowe (d=1), $R^2=% .2f$' % linear_r2,
...                 color='blue',
...                 lw=2)
>>> plt.xlabel('log (odsetek uboższej części społeczeństwa [LSTAT])')
>>> plt.ylabel('$\sqrt{\text{Cena}} \text{ w tysiącach dolarów [MEDV]}$')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

Po przekształceniu zmiennych objaśniających do postaci logarytmicznej oraz spierwiastkowaniu zmiennych odpowiedzi jesteśmy w stanie wykrywać relacje pomiędzy dwiema cechami za pomocą prostej regresji liniowej w skuteczniejszy sposób ( $R^2 = 0,69$ ) niż przy użyciu wcześniej wykorzystanych wielomianowych przekształceń cech (rysunek 10.11).



Rysunek 10.11. Wykres regresji liniowej wobec danych rzutowanych na nową przestrzeń cech

## Analiza nieliniowych relacji za pomocą algorytmu losowego lasu

W niniejszym podrozdziale zajmiemy się omówieniem regresywnego **losowego lasu**, który znacznie różni się od pozostałych modeli regresji omówionych w tym rozdziale. Losowy las, czyli zespół wielu **drzew decyzyjnych**, możemy interpretować jako zlepek różnych funkcji lini-

wych, w przeciwieństwie do globalnych modeli regresji liniowej i wielomianowej opisanych w poprzednich podrozdziałach. Innymi słowy, za pomocą algorytmu drzew decyzyjnych rozdzielimy przestrzeń danych wejściowych na mniejsze podregiony, które są łatwiej zarządzane.

## Regresja przy użyciu drzewa decyzyjnego

Zaletą algorytmu drzewa decyzyjnego jest brak wymogów dotyczących transformacji cech w przypadku pracy na nieliniiowych danych. Pamiętamy z rozdziału 3., „Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn”, że rozwijaliśmy drzewo decyzyjne, dzieląc jego węzły w sposób przyrostowy aż do oczyszczenia jego liści lub spełnienia granicznego kryterium. W trakcie klasyfikowania próbek za pomocą algorytmu drzewa decyzyjnego zdefiniowaliśmy entropię jako miarę zanieczyszczenia węzłów, co pozwalało nam określić cechy, których rozdzielanie generowało maksymalny **przyrost informacji (IG)**; dla podziału binarnego możemy sformułować to następująco:

$$G(D_p, x_i) = I(D_p) - \frac{N_{\text{lewy}}}{N_p} I(D_{\text{lewy}}) - \frac{N_{\text{prawy}}}{N_p} I(D_{\text{prawy}})$$

Tutaj  $x$  oznacza rozdzielaną cechę,  $N_p$  — liczbę próbek w nadzędnym węźle,  $I$  — funkcję zanieczyszczenia,  $D_p$  — podzbiór danych uczących w nadzędnym węźle, a  $D_{\text{lewy}}$  i  $D_{\text{prawy}}$  — podzbiory próbek uczących w potomnych węzłach po dokonaniu podziału, odpowiednio: lewym i prawym. Przypominam, że naszym celem jest znalezienie podziału cech maksymalizującego przyrost informacji, czyli, innymi słowy, chcemy określić taki podział cech, który będzie redukował zanieczyszczenie potomnych węzłów. W rozdziale 3., „Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn”, miarę zanieczyszczenia była **entropia**, stanowiąca dobre kryterium klasyfikacji. Aby dostosować model drzewa decyzyjnego do regresji, w miejsce entropii jako miary zanieczyszczenia węzła  $t$  wstawimy błąd średniokwadratowy (MSE):

$$I(t) = MSE(t) = \frac{1}{N_t} \sum_{i \in D_t} (y^{(i)} - \hat{y}_t)^2$$

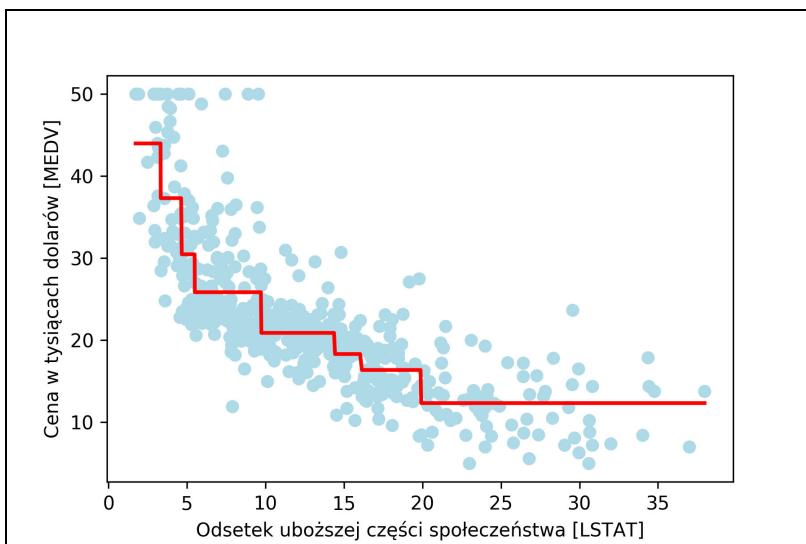
W powyższym wzorze  $N_t$  jest liczbą próbek uczących w węźle  $t$ ,  $D_t$  oznacza podzbiór uczący w węźle  $t$ ,  $y^{(i)}$  stanowi rzeczywistą wartość docelową, a  $\hat{y}_t$  — przewidywaną wartość docelową (wartość średnią próbek):

$$\hat{y}_t = \frac{1}{N_t} \sum_{i \in D_t} y^{(i)}$$

W kontekście regresji przy użyciu drzewa decyzyjnego wartość MSE jest często określana jako wariancja wewnętrzwęzlowa, dlatego kryterium rozdzielania bywa nazywane **redukcją wariancji**. Zobaczmy, jak wygląda linia dopasowania w drzewie decyzyjnym; wykorzystamy w tym celu klasę `DecisionTreeRegressor` zaimplementowaną w bibliotece scikit-learn i wymodelujemy nielinowy związek zmiennych **MEDV** i **LSTAT**:

```
>>> from sklearn.tree import DecisionTreeRegressor  
>>> X = df[['LSTAT']].values  
>>> y = df['MEDV'].values  
>>> tree = DecisionTreeRegressor(max_depth=3)  
>>> tree.fit(X, y)  
>>> sort_idx = X.flatten().argsort()  
>>> lin_regplot(X[sort_idx], y[sort_idx], tree)  
>>> plt.xlabel('Odsetek uboższej części społeczeństwa [LSTAT]')  
>>> plt.ylabel('Cena w tysiącach dolarów [MEDV]')  
>>> plt.show()
```

Jak widać na rysunku 10.12, algorytm drzewa decyzyjnego uchwycił ogólny trend kształtuający dane. Ograniczeniem tego modelu jest jednak brak możliwości uchwycenia ciągłości i rozróżnialności pożądanej prognozy. Poza tym musimy ostrożnie dobierać wartość wysokości drzewa po to, aby uniknąć przetrenowania lub niewystarczającego dopasowania modelu; w naszym przypadku wysokość drzewa równa 3 wydaje się spełniać swoje zadanie.



Rysunek 10.12. Wykres regresji przy użyciu algorytmu drzewa decyzyjnego

Przeanalizujmy teraz odporniejszą metodę dopasowywania regresyjnych drzew: losowe lasy.

## Regresja przy użyciu losowego lasu

Jak już wiemy z rozdziału 3., „Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn”, algorytm losowego lasu jest techniką zespołową, w której łączymy wiele drzew decyzyjnych. Losowy las wykazuje zazwyczaj większą skuteczność uogólniania od pojedynczych drzew z powodu losowości pozwalającej zmniejszyć wariancję modelu. Do innych zalet losowego lasu należą mniejsza wrażliwość na odstające dane oraz w dużej mierze uni-

zależnienie od strojenia hiperparametrycznego. Zazwyczaj jedynym parametrem, z którym musimy w tym przypadku eksperymentować jest liczba drzew w zespole. Podstawowy algorytm losowego lasu wykorzystywany w regresji jest niemal identyczny jak algorytm klasyfikujący, używany przez nas w rozdziale 3., „Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn”. Jedyną różnicą jest wprowadzenie błędu średniokwadratowego określającego wzrost poszczególnych drzew, natomiast prognozowana zmienna docelowa jest wyliczana jako uśredniona predykcja z całego lasu.

Użyjmy teraz wszystkich cech zestawu danych Housing do wyuczenia regresywnego algorytmu losowego lasu na 60% próbek, po czym sprawdźmy jego skuteczność wobec pozostałych 40% danych. Odpowiedzialny jest za to poniższy kod:

```
>>> X = df.iloc[:, :-1].values
>>> y = df['MEDV'].values
>>> X_train, X_test, y_train, y_test = \
...     train_test_split(X, y,
...                     test_size=0.4,
...                     random_state=1)

>>> from sklearn.ensemble import RandomForestRegressor
>>> forest = RandomForestRegressor(
...                 n_estimators=1000,
...                 criterion='mse',
...                 random_state=1,
...                 n_jobs=-1)
>>> forest.fit(X_train, y_train)
>>> y_train_pred = forest.predict(X_train)
>>> y_test_pred = forest.predict(X_test)
>>> print('MSE na danych uczących: %.3f, testowych: %.3f' % (
...     mean_squared_error(y_train, y_train_pred),
...     mean_squared_error(y_test, y_test_pred)))
>>> print('Współczynnik R^2 dla danych uczących: %.3f, testowych: %.3f' % (
...     r2_score(y_train, y_train_pred),
...     r2_score(y_test, y_test_pred)))
MSE na danych uczących: 1.642, testowych: 11.635
Współczynnik R^2 dla danych uczących: 0.960, testowych: 0.871
```

Widzimy, niestety, że algorytm losowego lasu dąży ku przetrenowaniu na danych uczących. Mimo to całkiem dobrze sobie radzi z ukazaniem zależności pomiędzy zmienną objaśniającą a docelową ( $R^2 = 0,871$  na zbiorze danych testowych).

Na koniec przyjrzyjmy się wartościom resztowym przewidywań:

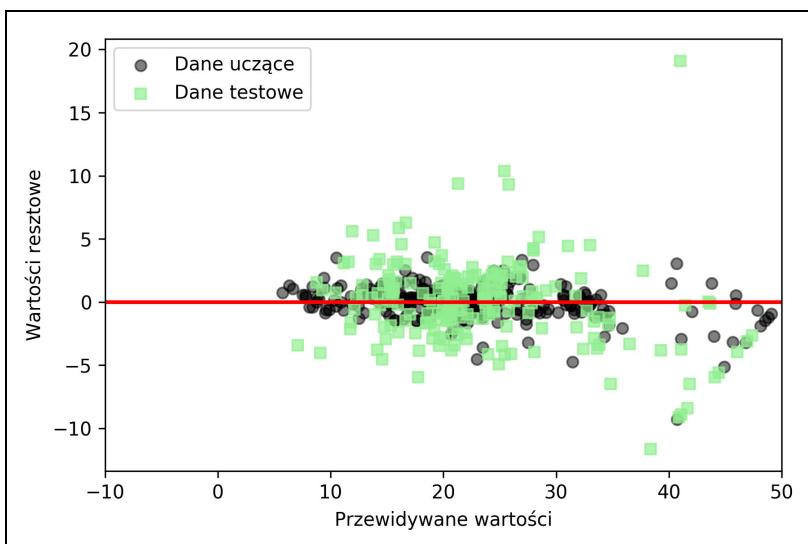
```
>>> plt.scatter(y_train_pred,
...               y_train_pred - y_train,
...               c='black',
...               marker='o',
...               s=35,
...               alpha=0.5,
```

```

...           label='Dane uczące')
>>> plt.scatter(y_test_pred,
...                 y_test_pred - y_test,
...                 c='lightgreen',
...                 marker='s',
...                 s=35,
...                 alpha=0.7,
...                 label='Dane testowe')
>>> plt.xlabel('Przewidywane wartości')
>>> plt.ylabel('Wartości resztowe')
>>> plt.legend(loc='upper left')
>>> plt.hlines(y=0, xmin=-10, xmax=50, lw=2, color='red')
>>> plt.xlim([-10, 50])
>>> plt.show()

```

Wiemy już dzięki współczynnikowi  $R^2$ , że model jest lepiej dopasowany do danych uczących niż testowych, na co wskazują odstające próbki wzdłuż osi  $y$ . Ponadto rozkład wartości resztowych nie wydaje się być całkowicie losowy wokół środkowej prostej, co stanowi dla nas informację, że model ten nie wyłapuje wszystkich informacji objaśniających. Z drugiej strony widoczny na rysunku 10.13 wykres wartości resztowych wskazuje na znacznie większą skuteczność tego modelu od modelu liniowego, którego wykres wygenerowaliśmy we wcześniejszej części rozdziału (rysunek 10.8).



Rysunek 10.13. Wykres resztowy dla algorytmu losowego lasu

W rozdziale 3., „Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn”, poruszyliśmy również temat sztuczki z funkcją jądra, którą w celach klasyfikowania stosowaliśmy wraz z maszyną wektorów nośnych (SVM) — technika przydatna podczas pracy na nieliniiowych danych. Maszyny SVM mogą być również częścią nieliniiowych zadań regresywnych, jednak zagadnienie to wykracza poza ramy niniejszej książki. Osoby zainteresowane wykorzystywaniem maszyn wektorów nośnych w regresji mogą zapoznać się ze znakomitym artykułem autorstwa Steve'a R. Gunna (*Support Vector Machines for Classification and Regression*, ISIS Technical Report, 1998, nr 14). Regresor SVM jest również zaimplementowany w interfejsie scikit-learn, a więcej informacje na jego temat znajdziesz pod adresem <http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html#sklearn.svm.SVR>.

## Podsumowanie

Na początku tego rozdziału nauczyliśmy się używać prostej regresji liniowej w celu modelowania związków pomiędzy pojedynczą zmienną objaśniającą a ciągłą zmienną odpowiedzi. Następnie przyjrzelismy się przydatnej technice analizy danych objaśniających pozwalającej na wyszukiwanie wzorców oraz anomalii wśród danych, co stanowi bardzo ważny etap początkowy w tworzeniu modeli predykcyjnych.

Stworzyliśmy nasz pierwszy model, implementując regresję liniową przy użyciu gradientowej metody optymalizacyjnej. Dowiedzieliśmy się następnie, jak wykorzystywać regresywne modele liniowe biblioteki scikit-learn oraz implementować odporną technikę regresji (RANSAC) jako sposób ograniczania wpływu odstających danych. W celu ocenienia skuteczności predykcyjnej modeli regresywnych wyliczaliśmy błąd średniokwadratowy oraz związany z nim współczynnik  $R^2$ . Ponadto poznaliśmy użyteczną metodę graficznego diagnozowania problemów dotykających modele regresji: wykres resztowy.

Po omówieniu sposobu wprowadzenia regularyzacji do modeli regresywnych w celu zmniejszenia złożoności modelu i uniknięcia przetrenowania przeszliśmy do analizy kilku różnych modeli pozwalających na badanie zależności nieliniiowych, m.in. wielomianowej transformacji cech i regresorów losowego lasu.

Do tej pory szczegółowo zajmowaliśmy się opisem uczenia nadzorowanego, klasyfikacji i analizy regresywnej. W następnym rozdziale przejdziemy do kolejnego interesującego działu uczenia maszynowego: uczenia nienadzorowanego. Nauczymy się wykorzystywać analizę skupień do wykrywania ukrytych struktur wśród danych przy braku zmiennych docelowych.



# Praca z nieoznakowanymi danymi — analiza skupień

W poprzednich rozdziałach wykorzystywaliśmy techniki uczenia nadzorowanego do tworzenia modeli uczenia maszynowego za pomocą danych zawierających już zdefiniowaną odpowiedź — etykiety klas były dostępne w danych uczących. W tym rozdziale zmienimy nieco obszar naszych zainteresowań i przyjrzymy się analizie skupień, jednej z technik **uczenia nienadzorowanego**, pozwalającej na wykrywanie ukrytych struktur w danych niezawierających jawnej odpowiedzi. Zadaniem klasteryzacji jest wyszukiwanie naturalnych zgrupowań danych — elementy danej grupy wykazują wzajemnie większe podobieństwo niż do składowych z innych skupień.

Klasteryzacja z powodu swojej badawczej natury stanowi fascynujące zagadnienie, a w tym rozdziale zapoznamy się z następującymi pojęciami umożliwiającymi organizowanie danych w sensowne struktury:

- wykrywanie centrów podobieństwa za pomocą popularnego algorytmu centroidów,
- stosowanie metod wstępujących do budowania hierarchicznych drzew,
- identyfikowanie dowolnych kształtów obiektów przy użyciu algorytmów gęstościowych.

# Grupowanie obiektów na podstawie podobieństwa przy użyciu algorytmu centroidów

W tym podrozdziale przyjrzymy się jednemu z najpowszechniejszych algorytmów analizy skupień (**grupowania, klasteryzacji**; ang. *clustering*) — **algorytmowi centroidów (k-średnich; ang. k-means)**, który jest równie często używany w pracach naukowych, jak w przemyśle. Klasteryzacja jest techniką umożliwiającą wyszukiwanie grup podobnych obiektów, czyli obiektów wykazujących między sobą ściślejsze zależności niż z obiektem stanowiącym składowe innych grup. Przykładami biznesowych zastosowań analizy skupień są grupowanie dokumentów, muzyki i filmów pod względem tematyki, a także wyszukiwanie klientów o podobnych zainteresowaniach na podstawie historii zakupów (trzon silników wyszukiwarek).

Przekonamy się niebawem, że algorytm centroidów w porównaniu z innymi algorytmami klasteryzacji jest nie tylko niezwykle łatwy w implementacji, lecz również bardzo skuteczny obliczeniowo, co może wyjaśniać jego olbrzymią popularność. Algorytm k-średnich należy do kategorii grupowania bazującego na prototypach. W dalszej części rozdziału przeanalizujemy też dwie inne techniki analizy skupień — hierarchiczną i gęstościową. Klasteryzacja **bazująca na prototypach** (ang. *prototype-based clustering*) polega na tym, że każda grupa jest reprezentowana przez prototyp stanowiący albo **centroid** (wartość średnia) podobnych punktów o cechach ciągłych, albo **medoid** (najbardziej reprezentatywny lub najczęściej występujący punkt) w przypadku cech kategoryzujących. Algorytm centroidów świetnie radzi sobie z wykrywaniem grup o kulistym kształcie, ale jedną z jego wad jest konieczność odgórnego zdefiniowania liczby klastrów  $k$ . Niewłaściwy dobór parametru  $k$  to w efekcie niska skuteczność algorytmu. W dalszej części rozdziału poznamy również **metodę łokcia i wykresy profilu**, które są przydatnymi technikami w ocenie jakości analizy skupień, ułatwiającymi określenie optymalnej liczby klastrów ( $k$ ).

Chociaż można stosować analizę centroidów wobec wielowymiarowych próbek, w poniższych przykładach będziemy korzystać z prostego, dwuwymiarowego zestawu danych w celu zachowania przejrzystości:

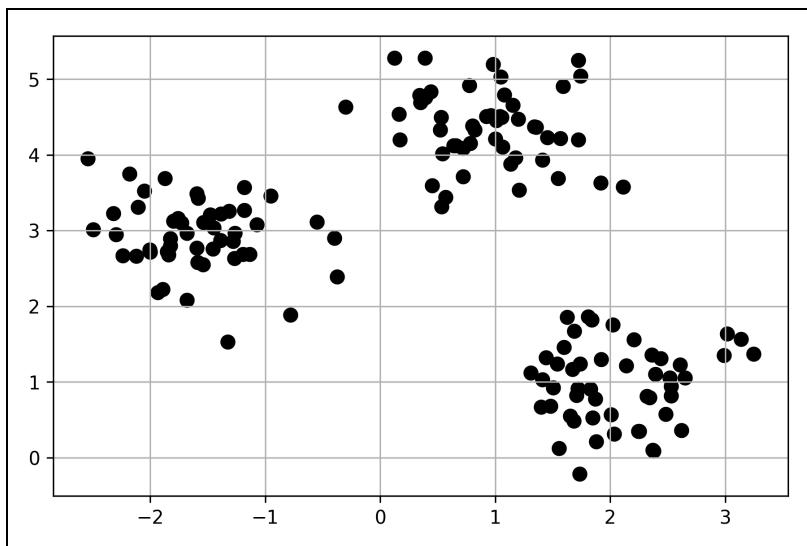
```
>>> from sklearn.datasets import make_blobs  
>>> X, y = make_blobs(n_samples=150,  
...                   n_features=2,  
...                   centers=3,  
...                   cluster_std=0.5,  
...                   shuffle=True,  
...                   random_state=0)  
>>> import matplotlib.pyplot as plt  
>>> plt.scatter(X[:,0],  
...               X[:,1],  
...               c='black',  
...               marker='o',
```

```

...           s=50)
>>> plt.grid()
>>> plt.show()

```

Stworzyliśmy właśnie zbiór danych składający się ze 150 losowych punktów ogólnie zgrupowanych w trzech regionach o większej gęstości, co możemy zobaczyć na rysunku 11.1.



Rysunek 11.1. Wykres wygenerowanego zbioru skupionych danych

W rzeczywistych zastosowaniach analizy skupień nie mamy żadnych pewnych informacji kategoryzujących na temat tych danych; gdybyśmy mieli, wykorzystalibyśmy algorytm uczenia nadzorowanego. Zatem naszym celem jest pogrupowanie próbek na podstawie podobieństw definiowanych przez cechy; osiągniemy go za pomocą algorytmu centroidów, który możemy podsumować w czterech następujących etapach:

1. Losowo dobierz  $k$  centroidów z punktów danych; staną się one początkowymi punktami środkowymi klastrów.
2. Przydziel każdą próbke do najbliższego centroida  $\mu^j, j \in \{1, \dots, k\}$ .
3. Przesuń centroid do środka przydzielonego doń zgrupowania próbek.
4. Powtarzaj etapy 2. i 3. aż do osiągnięcia stałej wartości przynależności skupienia albo do osiągnięcia maksymalnej liczby iteracji zdefiniowanej przez użytkownika.

Kolejne pytanie, jakie się nasuwa, brzmi: **w jaki sposób mierzymy podobieństwo pomiędzy obiektyami?** Możemy zdefiniować podobieństwo jako przeciwnieństwo odległości, a często stosowaną metryką odległości wobec zgrupowanych próbek o cechach ciągłych jest **kwadrat odległości euklidesowej** pomiędzy dwoma punktami  $x$  i  $y$  w **m-wymiarowej** przestrzeni:

$$d(\mathbf{x}, \mathbf{y})^2 = \sum_{j=1}^m (x_j - y_j)^2 = \|\mathbf{x} - \mathbf{y}\|_2^2$$

Zwróć uwagę, że w powyższym wzorze indeks  $j$  odnosi się do  $j$ -tego wymiaru (kolumny cech) próbki  $x$  i  $y$ . W pozostałej części podrozdziału indeksy górne  $i$  oraz  $j$  będą oznaczały odpowiednio: indeks próbki oraz indeks skupienia.

Na podstawie tej metryki odległości euklidesowej możemy opisać algorytm centroidów jako proste zagadnienie optymalizacji, przyrostową metodę minimalizacji **wewnętrzgrupowej sumy kwadratów błędów** (ang. *within-cluster sum of squared errors*), zwaną także **bezwładnością klastra** (ang. *cluster inertia*):

$$SSE = \sum_{i=1}^n \sum_{j=1}^k w^{(i,j)} \|\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(j)}\|_2^2$$

Parametr  $\boldsymbol{\mu}^{(j)}$  reprezentuje punkt (centroid) klastra  $j$ , a  $w^{(i,j)} = 1$ , jeżeli próbka  $x^{(i)}$  znajduje się w skupieniu  $j$ , w przeciwnym wypadku  $w^{(i,j)} = 0$ .

Gdy już znamy podstawowy mechanizm działania algorytmu centroidów, zastosujmy go wobec naszego przykładowego zestawu danych, implementując klasę `KMeans`, która stanowi część modułu `cluster` (w bibliotece scikit-learn):

```
>>> from sklearn.cluster import KMeans
>>> km = KMeans(n_clusters=3,
...                 init='random',
...                 n_init=10,
...                 max_iter=300,
...                 tol=1e-04,
...                 random_state=0)
>>> y_km = km.fit_predict(X)
```

W powyższym fragmencie kodu wyznaczyliśmy liczbę pożądanych klastrów równą 3; określenie liczby skupień *a priori* stanowi jedno z ograniczeń algorytmu centroidów. Wprowadziliśmy parametr `n_init=10`, co oznacza, że uruchomiliśmy 10 niezależnych algorytmów skupień zawierających różne losowe centroidy, dzięki czemu wybrany zostanie ostateczny model cechujący się najmniejszą wartością sumy kwadratów błędów. Poprzez parametr `max_iter` określamy maksymalną liczbę iteracji dla każdego pojedynczego przebiegu (ich łączna liczba wynosi 300). Zwróć uwagę, że implementacja algorytmu centroidów w interfejsie scikit-learn zostaje wcześniej zatrzymana, jeżeli staje się zbieżna przed osiągnięciem maksymalnej liczby iteracji.

Istnieje jednak możliwość, że algorytm nie stanie się zbieżny w określonym przebiegu, co może stanowić problem (kosztowny obliczeniowo), jeżeli przyjmiemy względnie duże wartości parametru `max_iter`. Jednym ze sposobów rozwiązania tego problemu jest wybór większych wartości parametru `tol`, który kontroluje zakres tolerancji dla zmian wewnętrzgrupowej sumy kwadratów błędów stanowiącej wyznacznik zbieżności. W powyższym kodzie wybraliśmy zakres tolerancji o wartości `1e-04` (= 0,0001).

## Algorytm k-means++

Do tej pory zajmowaliśmy się klasycznym modelem k-srednich wykorzystującym losowe próbki do wyznaczania wstępnych centroidów, w wyniku czego mamy czasami do czynienia z nieprawidłowym grupowaniem lub powolną zbieżnością przy niewłaściwie dobranych początkowych centroidach. Możemy temu zapobiec, wielokrotnie uruchamiając algorytm centroidów na zestawie danych i wybierając najlepiej sprawujący się model (o najniższej wartości SSE). Inną strategią jest rozmieszczenie początkowych centroidów jak najdalej od siebie za pomocą algorytmu **k-means++**, co zapewnia lepsze i spójniejsze wyniki niż klasyczny algorytm k-srednich (D. Arthur i S. Vassilvitskii, *k-means++: The Advantages of Careful Seeding* [w:] *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, Society for Industrial and Applied Mathematics, 2007, s. 1027 – 1035).

Inicjację algorytmu k-means++ możemy opisać następująco:

1. Stwórz pusty zbiór  $M$  przechowujący  $k$  wybranych centroidów.
2. Losowo dobierz pierwszy centroid  $\mu^{(0)}$  z dostępnych próbek i umieść go w zbiorze  $M$ .
3. Dla każdej próbki  $x^{(i)}$  niebędącej częścią zbioru  $M$  znajdź minimalny kwadrat odległości  $d(x^{(i)}, M)^2$  od każdego centroidu umieszczonego w zestawie danych  $M$ .
4. W celu losowego doboru następnego centroidu  $\mu^{(p)}$  użyj ważonego rozkładu prawdopodobieństwa: 
$$\frac{d(\mu^{(p)}, M)^2}{\sum_i d(x^{(i)}, M)^2}.$$
5. Powtarzaj etapy 2. i 3. aż do znalezienia  $k$  centroidów.
6. Wykonaj dalsze obliczenia za pomocą klasycznego algorytmu centroidów.

Aby wykorzystać algorytm k-means++ wraz z obiektem KMeans wystarczy do parametru `init` wstawić wartość `k-means++` (domyślna konfiguracja) zamiast `random`.

Kolejnym problemem wynikającym ze stosowania algorytmu centroidów jest możliwość wyznaczenia pustych skupień. Zauważ, że nie mamy z tym do czynienia w przypadku algorytmów k-medoidów lub rozmytych c-srednich, którymi zajmiemy się w dalszej części rozdziału. Musimy jednak rozważyć ten problem w kontekście bieżącej implementacji algorytmu k-srednich. Jeżeli skupienie jest puste, algorytm będzie wyszukiwał próbkę znajdująca się najdalej od centroida tego pustego klastra, a następnie wyznaczy ten najdalszy punkt jako nowy centroid.

W czasie wykorzystywania algorytmu centroidów wobec rzeczywistych danych za pomocą metryki odległości euklidesowej musimy się upewnić, że wszystkie cechy są mierzone w tej samej skali i w razie potrzeby zastosować standaryzację lub normalizację min.-max.

Po przewidzeniu etykiet skupienia  $y_{km}$  i omówieniu niebezpieczeństw związanych z algorytmem centroidów zobaczymy, jak wyglądają klastry zidentyfikowane przez ten algorytm dla zestawu danych, oraz sprawdzmy, gdzie się znajdują poszczególne centroidy. Ich wartości są przechowywane w atrybutie `centers_` wyuczonego obiektu KMeans:

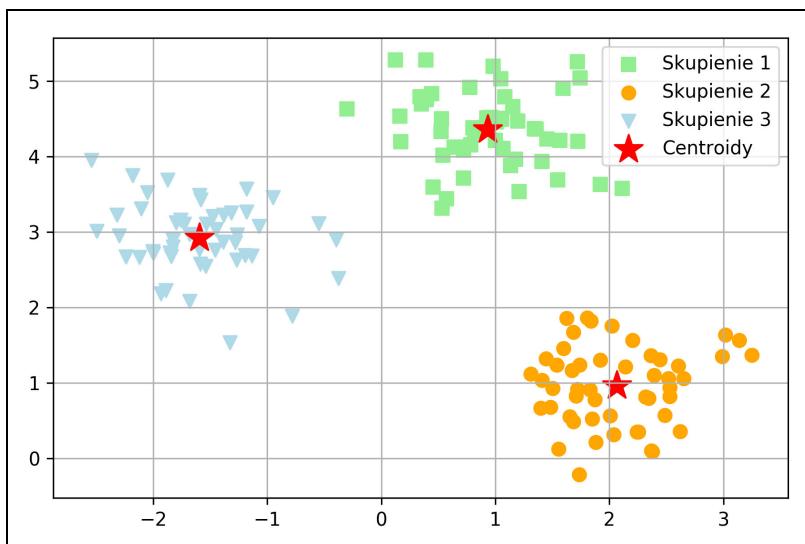
```
>>> plt.scatter(X[y_km==0,0],
...                 X[y_km==0,1],
...                 s=50,
...                 c='lightgreen',
...                 marker='s',
...                 label='Skupienie 1')
>>> plt.scatter(X[y_km==1,0],
...                 X[y_km==1,1],
...                 s=50,
...                 c='orange',
...                 marker='o',
...                 label='Skupienie 2')
>>> plt.scatter(X[y_km==2,0],
...                 X[y_km==2,1],
...                 s=50,
...                 c='lightblue',
...                 marker='v',
...                 label='Skupienie 3')
>>> plt.scatter(km.cluster_centers_[:,0],
...                 km.cluster_centers_[:,1],
...                 s=250,
...                 marker='*',
...                 c='red',
...                 label='Centroidy')
>>> plt.legend()
>>> plt.grid()
>>> plt.show()
```

Widzimy na rysunku 11.2, że algorytm rozmiścił trzy centroidy w środku każdego okręgu, co w przypadku naszego zestawu danych okazuje się rozsądny sposobem grupowania.

Chociaż algorytm centroidów dobrze się spisał wobec naszego przykładowego zestawu danych, należy wspomnieć o jego kilku ograniczeniach. Jedną z jego wad jest konieczność odgórnego wyznaczenia liczby skupień  $k$ , co może sprawiać problem w niektórych rzeczywistych zastosowaniach, zwłaszcza w przypadku wielowymiarowych zestawów danych, których nie można zwizualizować. Inną właściwością tego algorytmu jest brak hierarchiczności i nakładania się klastrów, zakładamy także, że w każdym klastrze znajduje się przynajmniej jeden element.

## Klasteryzacja twarda i miękka

**Twarda analiza skupień** (ang. *hard clustering*) opisuje rodzinę algorytmów, w której każda próbka stanowi element wyłącznie jednego klastra; przykładem jest omówiony w poprzednim podrozdziale algorytm centroidów. Z drugiej strony mamy do czynienia z algorytmami **miękkiej**



Rysunek 11.2. Klastry oraz centroidy wyznaczone za pomocą algorytmu k-średnich

(rozmytej) klasteryzacji (ang. *soft clustering* a. *fuzzy clustering*), w których dana próbka może przynależeć do kilku skupień. Popularnym przykładem miękkiej analizy skupień jest algorytm rozmytych c-średnich (ang. *fuzzy c-means* — FCM; zwany także algorytmem miękkich k-średnich). Pierwotna koncepcja tego algorytmu została ukuta już w latach 70. ubiegłego wieku, gdy Joseph C. Dunn zaproponował wczesną wersję rozmytej klasteryzacji w celu poprawienia skuteczności algorytmu centroidów (J.C. Dunn, *A Fuzzy Relative of the Isodata Process and its Use in Detecting Compact Well-separated Clusters*, 1973). Niemal dekadę później James C. Bezdek opublikował poprawioną wersję algorytmu rozmytej klasteryzacji, którą znamy obecnie jako algorytm FCM (J.C. Bezdek, *Pattern Recognition with Fuzzy Objective Function Algorithms*, Springer Science & Business Media, 2013).

Procedura FCM w znacznym stopniu przypomina algorytm centroidów. Zastępujemy tu jednak bezwzględny przydział do klastra prawdopodobieństwem przynależności każdej próbki do pozostałych skupień. W przypadku algorytmu k-średnich możemy wyrazić przydział danej próbki  $x$  do skupienia poprzez rządki wektor zawierający wartości dwójkowe:

$$\begin{bmatrix} \mu^{(1)} \rightarrow 0 \\ \mu^{(2)} \rightarrow 1 \\ \mu^{(3)} \rightarrow 0 \end{bmatrix}$$

Pozycja o wartości 1 mówi nam, że dana próbka jest przydzielona do centroida klastra  $\mu^{(j)}$  (przy założeniu, że  $k = 3, j \in \{1, 2, 3\}$ ). Z kolei wektor przydziału w algorytmie FCM można przedstawić następująco:

$$\begin{bmatrix} \boldsymbol{\mu}^{(1)} \rightarrow 0,1 \\ \boldsymbol{\mu}^{(2)} \rightarrow 0,85 \\ \boldsymbol{\mu}^{(3)} \rightarrow 0,05 \end{bmatrix}$$

Każda wartość mieści się w zakresie  $[0, 1]$  i reprezentuje prawdopodobieństwo przydziału do danego centroida klastra. Suma przydziałów dla danej próbki jest zawsze równa 1. Podobnie jak w przypadku klasycznego algorytmu k-średnich, algorytm FCM możemy podsumować w czterech etapach:

1. Zdefiniuj liczbę centroidów ( $k$ ) i losowo rozdziel każdemu punktowi przydziały do skupień.
2. Oblicz centroidy skupień  $\boldsymbol{\mu}^{(j)}, j \in \{1, \dots, k\}$ .
3. Zaktualizuj dla każdego punktu przydziały do skupień.
4. Powtarzaj etapy 2. i 3. aż do uzyskania stałych wartości współczynników przydziałów lub do osiągnięcia zdefiniowanej przez użytkownika maksymalnej liczby iteracji.

Funkcja celu w algorytmie FCM — zapisywana symbolem  $J_m$  — bardzo przypomina **wewnętrzgrupową sumę kwadratów błędów**, którą minimalizujemy w algorytmie centroidów:

$$J_m = \sum_{i=1}^n \sum_{j=1}^k w^{m(i,j)} \left\| \mathbf{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2^2, m \in [1, \infty)$$

Zwróć jednak uwagę, że wskaźnik przydziału  $w^{(i,j)}$  nie przyjmuje wartości binarnych, w przeciwieństwie do algorytmu centroidów ( $w^{(i,j)} \in \{0, 1\}$ ), lecz rzeczywiste prawdopodobieństwo przydziału do danego skupienia ( $w^{(i,j)} \in [0, 1]$ ). Być może zauważysz również, że dodaliśmy kolejny wykładnik do wyrażenia  $w^{(i,j)}$ ; wykładnik  $m$  przyjmujący wartości równe lub większe od 1 (zazwyczaj  $m = 2$ ) jest tzw. **blokiem rozmywania** (ang. *fuzziness coefficient*), nazywanym również **fuzyfikatorem**, który kontroluje stopień **rozmycia**. Im większa wartość współczynnika  $m$ , tym przydział do skupienia  $w^{(i,j)}$  staje się coraz mniejszy, co prowadzi do rozmytych klastrów. Samo prawdopodobieństwo przydziału do skupienia możemy zapisać następująco:

$$w^{(i,j)} = \left[ \sum_{p=1}^k \left( \frac{\left\| \mathbf{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \mathbf{x}^{(i)} - \boldsymbol{\mu}^{(p)} \right\|_2} \right)^{\frac{2}{m-1}} \right]^{-1}$$

Jeżeli np. wybraliśmy trzy centra skupień wykorzystane w powyższym przykładzie, możemy wyliczyć prawdopodobieństwo przydziału próbki  $\mathbf{x}^{(i)}$  do klastra  $\boldsymbol{\mu}^{(j)}$  w sposób przedstawiony poniżej:

$$w^{(i,j)} = \left[ \left( \frac{\left\| \mathbf{x}^{(i)} - \boldsymbol{\mu}^{(1)} \right\|_2}{\left\| \mathbf{x}^{(i)} - \boldsymbol{\mu}^{(1)} \right\|_2} \right)^{\frac{2}{m-1}} + \left( \frac{\left\| \mathbf{x}^{(i)} - \boldsymbol{\mu}^{(2)} \right\|_2}{\left\| \mathbf{x}^{(i)} - \boldsymbol{\mu}^{(2)} \right\|_2} \right)^{\frac{2}{m-1}} + \left( \frac{\left\| \mathbf{x}^{(i)} - \boldsymbol{\mu}^{(3)} \right\|_2}{\left\| \mathbf{x}^{(i)} - \boldsymbol{\mu}^{(3)} \right\|_2} \right)^{\frac{2}{m-1}} \right]^{-1}$$

Z kolei środek skupienia  $\mu^{(j)}$  wyliczamy jako wartość średnią wszystkich próbek w danym skupieniu ważoną za pomocą stopnia przynależności do tego klastra:

$$\boldsymbol{\mu}^{(j)} = \frac{\sum_{i=1}^n w^{m(i,j)} \mathbf{x}^{(i)}}{\sum_{i=1}^n w^{m(i,j)}}$$

Już na pierwszy rzut oka widać, że każda kolejna iteracja algorytmu FCM staje się coraz bardziej kosztowna obliczeniowo w porównaniu z algorytmem centroidów. Jednak technika ta zazwyczaj wymaga mniejszej liczby iteracji do osiągnięcia zbieżności. Niestety, obecnie algorytm FCM nie jest zaimplementowany w bibliotece scikit-learn. Wykazano za to, że zarówno algorytm centroidów, jak i FCM uzyskują bardzo zbliżone wyniki, co zostało zaprezentowane w artykule autorstwa Soumiego Ghosha i Sanjaya K. Dubeya (*Comparative Analysis of k-means and Fuzzy c-means Algorithms*, IJACSA, t. 4, 2013, s. 35 – 38).

## Stosowanie metody łokcia do wyszukiwania optymalnej liczby skupień

Jednym z głównych wyzwań nienadzorowanego uczenia jest nieznajomość ostatecznej odpowiedzi. Nie mamy w zestawie danych etykiet klas pozwalających na stosowanie technik opisanych w rozdziale 6., „Najlepsze metody oceny modelu i strojenie parametryczne”, pozwalających na ocenę skuteczności modelu nadzorowanego. Dlatego w celu ilościowego ujęcia jakości grupowania musimy korzystać z wewnętrznych metryk — takich jak wewnętrzgrupowa suma kwadratów błędów (znieksztalcenie) omówiona na początku niniejszego rozdziału — co pozwala na porównanie skuteczności różnych klasteryzacji metodą centroidów. Na szczęście nie musimy jawnie wyliczać znieksztalcenia, ponieważ po wytrenowaniu modelu KMeans możemy skorzystać z atrybutu `inertia_`:

```
>>> print('Znieksztalcenie: %.2f' % km.inertia_)
Znieksztalcenie: 72.48
```

Znając wewnętrzgrupową sumę kwadratów błędów, możemy teraz skorzystać z narzędzia graficznego, tzw. **metody łokcia** (ang. *elbow method*), w celu oszacowania optymalnej liczby klastrów  $k$  dla danego zadania. Intuicja podpowiada, że jeżeli wartość  $k$  rośnie, to maleje znieksztalcenie. Wynika to z faktu, że próbki będą bliżej centroidów, do których zostały przydzielone. Celem metody łokcia jest określenie wartości parametru  $k$ , dla której znieksztalcenie zaczyna najszyciej wzrastać; stanie się to bardziej zrozumiałe, gdy wygenerujemy wykres znieksztalcenia dla różnych wartości parametru  $k$ :

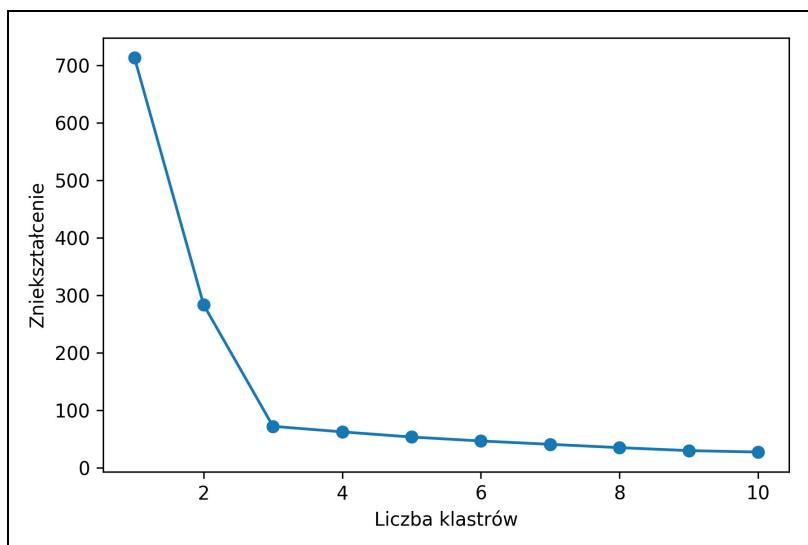
```
>>> distortions = []
>>> for i in range(1, 11):
...     km = KMeans(n_clusters=i,
...                  init='k-means++',
...                  n_init=10,
...                  max_iter=300,
```

```

...                     random_state=0)
>>> km.fit(X)
>>> distortions.append(km.inertia_)
>>> plt.plot(range(1,11), distortions, marker='o')
>>> plt.xlabel('Liczba skupień')
>>> plt.ylabel('Zniekształcenie')
>>> plt.show()

```

Jak widać na wykresie z rysunku 11.3, łokieć pojawia się przy wartości  $k = 3$ , co stanowi potwierdzenie, że ta liczba skupień jest optymalna dla analizowanego zestawu danych.



Rysunek 11.3. Wykres łokcia dla analizowanego zestawu danych

## Ujęcie ilościowe jakości klasteryzacji za pomocą wykresu profilu

Kolejną metryką wewnętrzną pozwalającą na ocenę jakości analizy skupień jest **analiza profilu** (ang. *silhouette analysis*); można ją wprowadzać również do innych algorytmów klasteryzacji, którymi zajmiemy się w dalszej części rozdziału. Analiza profilu często przyjmuje postać narzędzia graficznego, które generuje wykres pokazujący stopień upakowania próbek w danym skupieniu. Aby obliczyć **współczynnik profilu** pojedynczej próbki w naszym zestawie danych, należy wykonać następujące czynności:

1. Oblicz spójność klastra  $a^{(i)}$  jako średnią odległość pomiędzy próbką  $x^{(i)}$  a wszystkimi pozostałymi elementami skupienia.
2. Oblicz odstęp klastra  $b^{(i)}$  od najbliższego sąsiadującego skupienia jako średnią odległość pomiędzy próbką  $x^{(i)}$  a wszystkimi próbками należącymi do sąsiadującej grupy.

3. Oblicz profil  $s^{(i)}$  jako różnicę pomiędzy spójnością klastra a jego odstępem, dzieloną przez większą wartość jednego z tych dwóch parametrów, zgodnie z poniższym równaniem:

$$s^{(i)} = \frac{b^{(i)} - a^{(i)}}{\max\{b^{(i)}, a^{(i)}\}}$$

Wartości współczynnika profilu mieszczą się w zakresie od -1 do 1. Na podstawie powyższego wzoru widzimy, że przyjmuje on wartość 0, gdy odstęp i spójność klastra są takie same ( $b^{(i)} = a^{(i)}$ ). Ponadto zbliżamy się do wartości idealnego profilu 1, jeśli  $b^{(i)} >> a^{(i)}$ , ponieważ parametr  $b^{(i)}$  mówi nam, jak bardzo dana próbka nie pasuje do pozostałych skupień, a  $a^{(i)}$  informuje nas o stopniu podobieństwa tej próbki do innych elementów wspólnego klastra.

Współczynnik profilu jest dostępny w bibliotece scikit-learn jako klasa `silhouette_samples` umieszczona w module `metrics`; możemy również dodatkowo importować klasę `silhouette_scores` — służy ona do wyliczania uśrednionego współczynnika profilu wobec wszystkich próbek, co odpowiada zastosowaniu funkcji `numpy.mean(silhouette_samples(...))`. Za pomocą poniższego kodu stworzymy wykres współczynników profilu dla algorytmu centroidów o parametrze  $k = 3$ :

```
>>> km = KMeans(n_clusters=3,
...                 init='k-means++',
...                 n_init=10,
...                 max_iter=300,
...                 tol=1e-04,
...                 random_state=0)
>>> y_km = km.fit_predict(X)

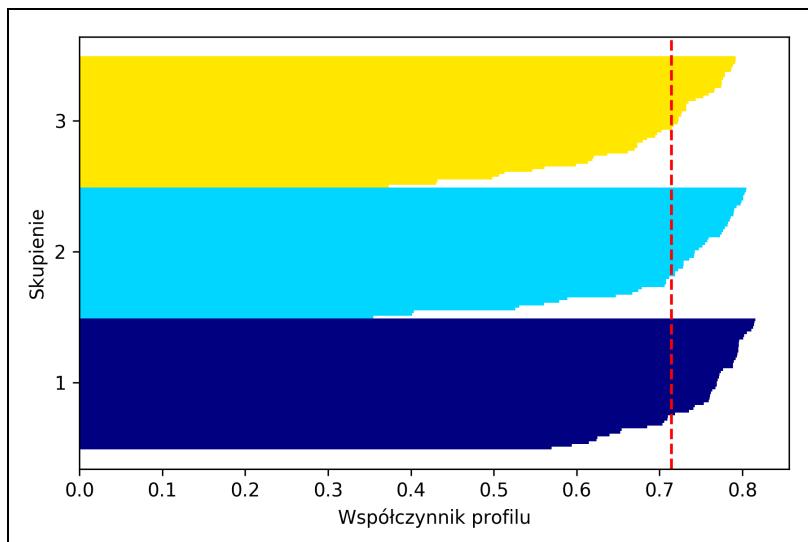
>>> import numpy as np
>>> from matplotlib import cm
>>> from sklearn.metrics import silhouette_samples
>>> cluster_labels = np.unique(y_km)
>>> n_clusters = cluster_labels.shape[0]
>>> silhouette_vals = silhouette_samples(X,
...                                         y_km,
...                                         metric='euclidean')
>>> y_ax_lower, y_ax_upper = 0, 0
>>> yticks = []
>>> for i, c in enumerate(cluster_labels):
...     c_silhouette_vals = silhouette_vals[y_km == c]
...     c_silhouette_vals.sort()
...     y_ax_upper += len(c_silhouette_vals)
...     color = cm.jet(float(i) / n_clusters)
...     plt.barh(range(y_ax_lower, y_ax_upper),
...              c_silhouette_vals,
...              height=1.0,
...              edgecolor='none',
```

```

...
        color=color)
...
    yticks.append((y_ax_lower + y_ax_upper) / 2)
    y_ax_lower += len(c_silhouette_vals)
>>> silhouette_avg = np.mean(silhouette_vals)
>>> plt.axvline(silhouette_avg,
...                 color="red",
...                 linestyle="--")
>>> plt.yticks(yticks, cluster_labels + 1)
>>> plt.ylabel('Skupienie')
>>> plt.xlabel('Współczynnik profilu')
>>> plt.show()

```

Dzięki wykresowi profilu (rysunek 11.4) jesteśmy w stanie szybko oszacować rozmiary różnych klastrów i wyznaczyć skupienia zawierające **odstające elementy**.



Rysunek 11.4. Wykres profilu dla zestawu danych składających się z trzech skupień ( $k = 3$ )

Jak widać na rysunku 11.4, nasze współczynniki profilu mają wartość znacznie przekraczającą 0, co wskazuje na właściwie przeprowadzoną klasteryzację. Ponadto w celu podsumowania skuteczności analizy skupień umieściliśmy na wykresie uśredniony współczynnik profilu (linia przerywana).

Przekonajmy się teraz, jak będzie wyglądał wykres profilu dla **nieprawidłowo** przeprowadzonej analizy skupień; w tym celu wstawimy do algorytmu k-średnich tylko dwa centroidy:

```

>>> km = KMeans(n_clusters=2,
...               init='k-means++',
...               n_init=10,
...               max_iter=300,
...               tol=1e-04,

```

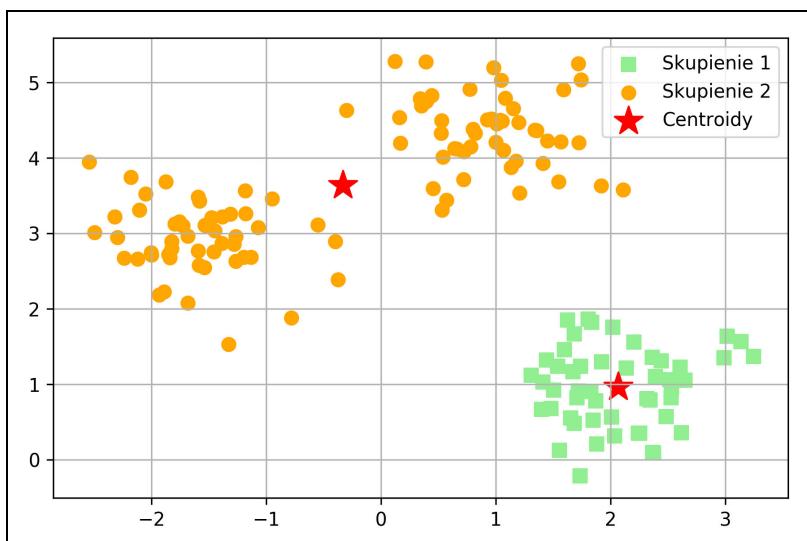
```

...           random_state=0)
>>> y_km = km.fit_predict(X)

>>> plt.scatter(X[y_km==0,0],
...                 X[y_km==0,1],
...                 s=50, c='lightgreen',
...                 marker='s',
...                 label='Skupienie 1')
>>> plt.scatter(X[y_km==1,0],
...                 X[y_km==1,1],
...                 s=50,
...                 c='orange',
...                 marker='o',
...                 label='Skupienie 2')
>>> plt.scatter(km.cluster_centers_[:,0],
...                 km.cluster_centers_[:,1],
...                 s=250,
...                 marker='*',
...                 c='red',
...                 label='Centroidy')
>>> plt.legend()
>>> plt.grid()
>>> plt.show()

```

Zgodnie z wykresem na rysunku 11.5 jeden z centroidów został wyznaczony pomiędzy dwoma z trzech zgrupowań próbek. Chociaż taka analiza skupień nie jest zupełną porażką, z pewnością nie można jej nazwać optymalną.



Rysunek 11.5. Wykres analizy skupień dla dwóch centroidów ( $k = 2$ )

Stworzymy teraz wykres profilu, aby sprawdzić rezultat analizy skupień. Przypominam, że podczas pracy na rzeczywistych danych zazwyczaj nie mamy luksusu wizualizowania próbek na dwuwymiarowych wykresach, ponieważ analizowane dane przeważnie są wielowymiarowe:

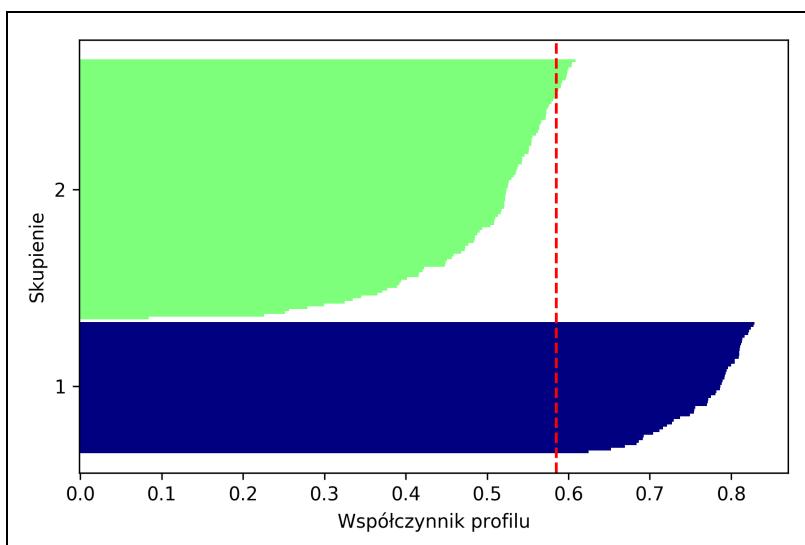
```
>>> cluster_labels = np.unique(y_km)
>>> n_clusters = cluster_labels.shape[0]
>>> silhouette_vals = silhouette_samples(X,
...                                         y_km,
...                                         metric='euclidean')
>>> y_ax_lower, y_ax_upper = 0, 0
>>> yticks = []
>>> for i, c in enumerate(cluster_labels):
...     c_silhouette_vals = silhouette_vals[y_km == c]
...     c_silhouette_vals.sort()
...     y_ax_upper += len(c_silhouette_vals)
...     color = cm.jet(i / n_clusters)
...     plt.barh(range(y_ax_lower, y_ax_upper),
...              c_silhouette_vals,
...              height=1.0,
...              edgecolor='none',
...              color=color)
...     yticks.append((y_ax_lower + y_ax_upper) / 2)
...     y_ax_lower += len(c_silhouette_vals)
>>> silhouette_avg = np.mean(silhouette_vals)
>>> plt.axvline(silhouette_avg, color="red", linestyle="--")
>>> plt.yticks(yticks, cluster_labels + 1)
>>> plt.ylabel('Skupienie')
>>> plt.xlabel('Współczynnik profilu')
>>> plt.show()
```

Jak widać na rysunku 11.6, poszczególne profile różnią się zarówno długością, jak i szerokością, co stanowi kolejny dowód na daleką od optymalnej skuteczność klasteryzacji.

## Organizowanie skupień do postaci drzewa klastrów

W niniejszym podrozdziale poznamy alternatywny sposób analizy skupień: **klasteryzację hierarchiczną** (ang. *hierarchical clustering*). Jedną z zalet algorytmów hierarchicznego grupowania jest możliwość generowania **dendrogramów** (graficznego przedstawienia binarnej, hierarchicznej analizy skupień), pomagających w interpretowaniu wyników poprzez tworzenie istotnych taksonomii. Kolejną przydatną własnością metod hierarchicznych jest brak konieczności odgórnego definiowania liczby skupień.

Dwie głównymi technikami wykorzystywanyimi w hierarchicznej analizie skupień są metody **aglomeracyjne** (ang. *agglomerative*) i **deglomeracyjne** (ang. *divisive*). W klasteryzacji deglo-



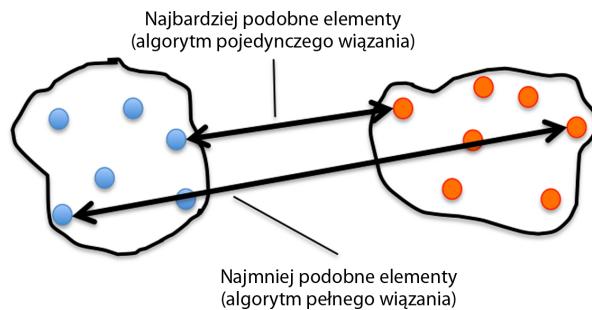
Rysunek 11.6. Wykres profilu dla niewłaściwie przeprowadzonej analizy skupień ( $k = 2$ )

meracyjnej zaczynamy od jednego klastra zawierającego w sobie wszystkie pozostałe skupienia i stopniowo dzielimy go na mniejsze grupy, aż do osiągnięcia stanu, w którym każdy klaster będzie zawierał tylko jedną próbkę. W tym podrozdziale zajmiemy się klasteryzacją aglomeracyjną stanowiącą przeciwieństwo grupowania deglomeracyjnego. W tym przypadku każda próbka stanowi osobne skupienie i dążymy do łączenia par najbliższych elementów, aż do uzyskania jednego wielkiego klastra.

W aglomeracyjnej analizie skupień stosowane są dwa standardowe algorytmy: **pojedynczego wiązania (najbliższego sąsiedztwa; ang. single linkage)** oraz **pełnego wiązania (najdalszego sąsiedztwa; ang. complete linkage)**. Za pomocą algorytmu pojedynczego wiązania wyszukujemy dwa punkty we wszystkich parach skupień (po jednym punkcie na każdy klaster), cechujących się najmniejszą odlegością pomiędzy klastrami. Zasada działania algorytmu pełnego wiązania jest podobna, z tym że łączymy w nim klastry, dla których została wyliczona największa odległość. Koncepcja obydwu algorytmów została zaprezentowana na rysunku 11.7.

Innymi powszechnie stosowanymi algorytmami w aglomeracyjnej analizie skupień są **metoda średnich połączeń (ang. average linkage)** oraz **metoda Warda (ang. Ward's linkage)**. W algorytmie średnich połączeń łączymy pary klastrów na podstawie minimalnych uśrednionych odległości pomiędzy wszystkimi elementami grupy w obydwu klastach. Z kolei w metodzie Warda łączone są te skupienia, dla których następuje najmniejszy wzrost wewnętrzgrupowej sumy kwadratów błędów.

W tym podrozdziale skoncentrujemy się na klasteryzacji aglomeracyjnej przy użyciu algorytmu pełnego wiązania. Jest to procedura przyrostowa, którą możemy podzielić na następujące etapy:



Rysunek 11.7. Schematyczne przedstawienie działania algorytmów pojedynczego wiązania i pełnego wiązania

1. Oblicz macierz odległości dla wszystkich próbek.
2. Stwórz reprezentację każdego elementu jako oddzielnego klastra.
3. Połącz dwa najbliższe klastry na podstawie odległości pomiędzy dwoma najbardziej różniącymi się (oddalonymi) elementami.
4. Zaktualizuj macierz odległości.
5. Powtarzaj kroki 2. – 4., dopóki nie pozostanie tylko jedno skupienie.

Przeanalizujemy teraz sposób obliczania macierzy odległości (etap 1.). Najpierw jednak wygenerujmy losowe próbki, na których sprawdzimy działanie algorytmu. Rzędy będą symbolizować wyniki obserwacji (identyfikatory od 0 do 4), a kolumny będą przedstawiać różne cechy (X, Y i Z) naszych próbek:

```
>>> import pandas as pd
>>> import numpy as np
>>> np.random.seed(123)
>>> variables = ['X', 'Y', 'Z']
>>> labels = ['ID_0', 'ID_1', 'ID_2', 'ID_3', 'ID_4']
>>> X = np.random.random([5,3])*10
>>> df = pd.DataFrame(X, columns=variables, index=labels)
>>> df
```

Po uruchomieniu powyższego kodu naszym oczom powinien ukazać się zaprezentowany w tabeli 11.1 obiekt DataFrame zawierający losowo wygenerowane próbki.

Tabela 11.1. Obiekt DataFrame zawierający losowo wygenerowane próbki

	X	Y	Z
ID_0	6.964692	2.861393	2.268515
ID_1	5.513148	7.194690	4.231065
ID_2	9.807642	6.848297	4.809319
ID_3	3.921175	3.431780	7.290497
ID_4	4.385722	0.596779	3.980443

## Przeprowadzanie hierarchicznej analizy skupień na macierzy odległości

Aby wyliczyć macierz odległości zawierającą dane wejściowe dla algorytmu hierarchicznej klasteryzacji, wykorzystamy funkcję `pdist` będącą składową podmodułu `spatial.distance` (w bibliotece SciPy):

```
>>> from scipy.spatial.distance import pdist, squareform
>>> row_dist = pd.DataFrame(squareform(
...     pdist(df, metric='euclidean')),
...     columns=labels, index=labels)
>>> row_dist
```

Dzięki powyższemu fragmentowi kodu wyliczyliśmy odległość euklidesową pomiędzy wszystkimi parami próbek na podstawie cech X, Y i Z. Taka zwarta macierz odległości — zwrócona przez funkcję `pdist` — stanowi wartość wejściową funkcji `squareform`, która posłużyła nam do utworzenia macierzy symetrycznej zawierającej odległości pomiędzy parami poszczególnych punktów, co zostało zaprezentowane w tabeli 11.2.

**Tabela 11.2.** Symetryczna macierz odległości wyliczona za pomocą funkcji `pdist` i `squareform`

	ID_0	ID_1	ID_2	ID_3	ID_4
ID_0	0.000000	4.973534	5.516653	5.899885	3.835396
ID_1	4.973534	0.000000	4.347073	5.104311	6.698233
ID_2	5.516653	4.347073	0.000000	7.244262	8.316594
ID_3	5.899885	5.104311	7.244262	0.000000	4.382864
ID_4	3.835396	6.698233	8.316594	4.382864	0.000000

Przeprowadzimy teraz aglomerację klastrów metodą pełnego wiązania za pomocą funkcji `linkage` stanowiącej część podmodułu `cluster.hierarchy` (biblioteka SciPy), która zwraca tzw. **macierz wiązania** (ang. *linkage matrix*).

Zanim jednak wywołamy funkcję `linkage`, zapoznajmy się dokładnie z jej dokumentacją:

```
>>> from scipy.cluster.hierarchy import linkage
>>> help(linkage)
[...]
Parametry:
y : tablica typu ndarray
Zwarta lub nadmiarowa macierz odległości. Zwarta
macierz odległości jest jednowymiarową tablicą zawierającą
górny trójkąt macierzy odległości. W takiej postaci
zwraca ją funkcja pdist. Można ewentualnie przekazać zbiór m
wektorów obserwacji w n wymiarach jako tablicę m x n.
```

**method** : ciąg znaków, opcjonalny

Rodzaj wykorzystywanego algorytmu wiązania. Dokładny opis znajduje się w sekcji Linkage Methods.

**metric** : ciąg znaków, opcjonalny

Stosowana metryka odległości. Listę akceptowanych metryk odległości znajdziesz w opisie funkcji `distance.pdist`.

Zwroca:

**Z** : tablica typu ndarray

Hierarchiczna klasteryzacja zakodowana w postaci macierzy wiązania.

[...]

Z opisu funkcji wynika, że możemy wykorzystać zwartą macierz odległości (trójkątną górną) wygenerowaną przez funkcję `pdist` jako atrybut wejściowy. Możemy również ewentualnie wykorzystać początkową tablicę zawierającą dane i zastosować metrykę `euclidean` jako argument w funkcji `linkage`. Nie powinniśmy jednak wykorzystywać zdefiniowanej wcześniej kwadratowej macierzy odległości, ponieważ uzyskalibyśmy pomiary odległości odmienne od oczekiwanych. W ramach podsumowania tego zagadnienia zaprezentowałem poniżej trzy możliwe scenariusze:

- **Nieprawidłowe podejście:** wykorzystujemy tu kwadratową macierz odległości. Kod wygląda następująco:

```
>>> from scipy.cluster.hierarchy import linkage
>>> row_clusters = linkage(row_dist,
...                         method='complete',
...                         metric='euclidean')
```

- **Prawidłowe podejście:** wprowadzamy tu zwartą macierz odległości. Używamy do tego poniższego kodu:

```
>>> row_clusters = linkage(pdist(df, metric='euclidean'),
...                         method='complete')
```

- **Prawidłowe podejście:** stosujemy pierwotną macierz danych. Oto fragment kodu:

```
>>> row_clusters = linkage(df.values,
...                         method='complete',
...                         metric='euclidean')
```

Aby dokładniej przeanalizować wyniki klasteryzacji, przekształcimy je w obiekt `DataFrame` (najlepiej będą się prezentować w aplikacji Jupyter Notebook):

```
>>> pd.DataFrame(row_clusters,
...                 columns=['Etykieta rzędu 1',
...                           'Etykieta rzędu 2',
...                           'Odległość',
...                           'Liczba elementów klastra'],
...                 index=['Klaster %d' %(i+1) for i in
...                        range(row_clusters.shape[0])])
```

Jak widać w tabeli 11.3, macierz wiązania składa się z kilku rzędów — każdy z nich symbolizuje jedno połączenie skupień. Pierwsze dwie kolumny zawierają wymienione najmniej podobne elementy w każdym klastrze, natomiast w trzeciej kolumnie widzimy podane odległości pomiędzy tymi elementami. Ostatnia kolumna zwraca liczbę składowych przynależnych do poszczególnych skupień.

Tabela 11.3. Macierz wiązania

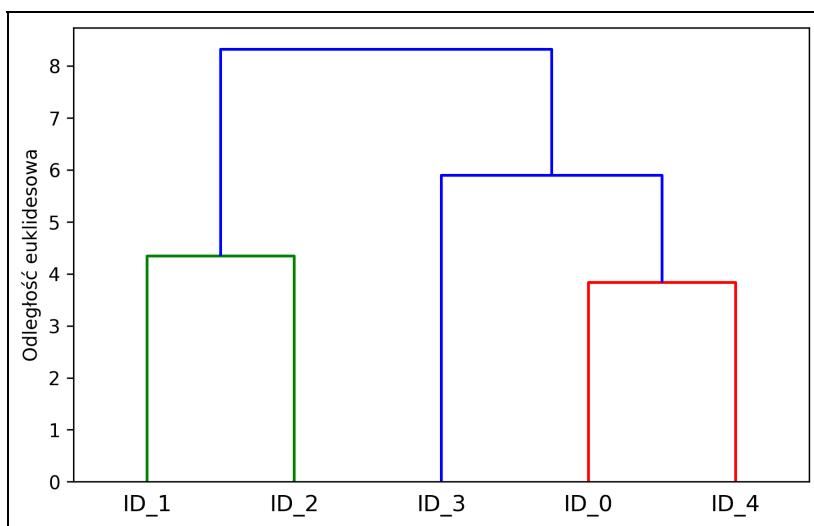
	Etykieta rzędu 1	Etykieta rzędu 2	Odległość	Liczba elementów klastra
Klaster 1	0	4	3.835396	2
Klaster 2	1	2	4.347073	2
Klaster 3	3	5	5.899885	3
Klaster 4	6	7	8.316594	5

Po obliczeniu macierzy wiązania możemy na własne oczy zobaczyć wyniki w postaci dendrogramu:

```
>>> from scipy.cluster.hierarchy import dendrogram
# jeżeli chcesz wygenerować dendrogram w kolorze czarnym (część 1/2):
# from scipy.cluster.hierarchy import set_link_color_palette
# set_link_color_palette('black')
>>> row_dendr = dendrogram(row_clusters,
...                           labels=labels,
...                           # jeżeli chcesz wygenerować dendrogram w kolorze czarnym
...                           # (część 2/2):
...                           color_threshold=np.inf
... )
>>> plt.tight_layout()
>>> plt.ylabel('Odległość euklidesowa')
>>> plt.show()
```

Po uruchomieniu powyższego kodu zauważysz, że poszczególne rozgałęzienia dendrogramu zostały oznaczone różnymi kolorami (rysunek 11.8). Kolorystyka jest dobrana na podstawie listy kolorów przechowywanej w interfejsie matplotlib, a poszczególne barwy są cyklicznie zmieniane wraz ze wzrostem wartości progowych odległości pomiędzy poszczególnymi skupieniami. Jeżeli chcesz, aby rozgałęzienia dendrogramu były wyświetlane w czarnym kolorze, wystarczy usunąć komentarze z zaznaczonych fragmentów kodu.

Dendrogram przedstawia różne klastry utworzone w trakcie aglomeracyjnej, hierarchicznej analizy skupień; widzimy np., że próbki  $ID\_0$  oraz  $ID\_4$  (a zaraz po nich  $ID\_1$  i  $ID\_2$ ) są do siebie najbardziej podobne pod względem euklidesowej metryki odległości.



Rysunek 11.8. Dendrogram przykładowego zbioru danych

## Dołączanie dendrogramów do mapy cieplnej

W praktycznych zastosowaniach dendrogramy są często wykorzystywane wspólnie z **mapą cieplną** (ang. *heat map*), służącą do przedstawiania poszczególnych wartości macierzy próbek przy użyciu oznaczeń kolorystycznych. Dowiemy się teraz, w jaki sposób przyłączyć dendrogram do mapy cieplnej oraz jak odpowiednio uporządkować na niej rzędy tabeli.

Powiązanie dendrogramu z mapą cieplną nie jest jednak takie proste, dlatego przyjrzymy się tej procedurze krok po kroku:

1. Tworzymy nowy obiekt `figure`, a za pomocą atrybutu `add_axes` definiujemy położenie osi *x* i *y*, szerokość oraz wysokość dendrogramu; ponadto obracamy go o 90 stopni zgodnie z ruchem wskazówek zegara. Osiągniemy to przy użyciu poniższego kodu:

```
>>> fig = plt.figure(figsize=(8,8), facecolor='white')
>>> axd = fig.add_axes([0.09,0.1,0.2,0.6])
>>> row_dendr = dendrogram(row_clusters, orientation='right')
# uwaga: dla biblioteki matplotlib w wersji >= v1.5.1, zmień parametr orientation='left'
```

2. Sortujemy dane w początkowym obiekcie `DataFrame` zgodnie z etykietami klasteryzacji, do których uzyskujemy dostęp z poziomu dendrogramu (słownika Pythona) za pomocą klucza `leaves`. Kod wygląda następująco:

```
>>> df_rowclust = df.ix[row_dendr['leaves'][:-1]]
```

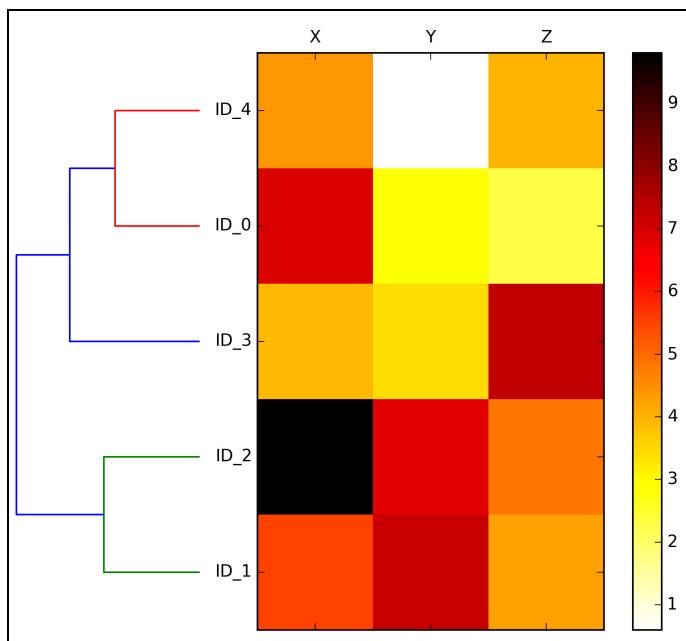
3. Tworzymy teraz mapę cieplną z posortowanej tabeli `DataFrame` i wstawiamy ją tuż obok dendrogramu:

```
>>> axm = fig.add_axes([0.23,0.1,0.6,0.6])
>>> cax = axm.matshow(df_rowclust,
...                      interpolation='nearest', cmap='hot_r')
```

4. Na koniec zmodyfikujemy wygląd mapy cieplnej, ukrywając jednostki oraz proste osi współrzędnych. Dodamy poza tym legendę, a poszczególne jednostki osi  $x$  i  $y$  oznaczymy odpowiednimi etykietami. Służy do tego następujący fragment kodu:

```
>>> axd.set_xticks([])
>>> axd.set_yticks([])
>>> for i in axd.spines.values():
...     i.set_visible(False)
>>> fig.colorbar(cax)
>>> axm.set_xticklabels([''] + list(df_rowclust.columns))
>>> axm.set_yticklabels([''] + list(df_rowclust.index))
>>> plt.show()
```

Po wykonaniu powyższych czynności naszym oczom powinna się ukazać mapa cieplna z dołączonym dendrogramem (rysunek 11.9).



Rysunek 11.9. Dendrogram z dołączoną mapą cieplną

Jak widać, kolejność rzędów na mapie cieplnej odpowiada klasteryzacji próbek na dendrogramie. Oprócz tego prostego dendrogramu w podsumowaniu naszego zestawu danych pomagają legenda mapy cieplnej oraz dołączone etykiety cech.

## Aglomeracyjna analiza skupień w bibliotece scikit-learn

Przekonaliśmy się już, w jaki sposób wykonać aglomeracyjną, hierarchiczną klasteryzację w bibliotece SciPy. Istnieje jednak implementacja tej techniki w interfejsie scikit-learn — `AgglomerativeClustering`, w której możemy dobrać liczbę zwracanych skupień. Rozwiązanie to przydaje się w sytuacji, gdy chcemy przyciąć drzewo klasteryzacji. Po wprowadzeniu wartości 2 w parametrze `n_clusters` utworzymy dwa skupienia próbek za pomocą algorytmu pełnego wiązania, w którym wykorzystamy metrykę odległości euklidesowej:

```
>>> from sklearn.cluster import AgglomerativeClustering
>>> ac = AgglomerativeClustering(n_clusters=2,
...                                affinity='euclidean',
...                                linkage='complete')
>>> labels = ac.fit_predict(X)
>>> print('Etykiety skupień: %s' % labels)
Etykiety skupień: [0 1 1 0 0]
```

Zgodnie z wektorem etykiet skupień próbki: pierwsza, czwarta i piąta ( $ID\_0$ ,  $ID\_3$  oraz  $ID\_4$ ) zostały przydzielone do jednego skupienia (0), a próbki druga i trzecia ( $ID\_1$  i  $ID\_2$ ) — do drugiego skupienia (1), co jest zgodne z wynikami zaobserwowanymi na dendrogramie.

## Wyznaczanie rejonów o dużej gęstości za pomocą algorytmu DBSCAN

Istnieje zbyt wiele algorytmów klasteryzacji, żeby móc je wszystkie opisać w niniejszej książce, ale przyjrzyjmy się jeszcze jednej technice analizy skupień: **algorytmowi DBSCAN** (ang. *density-based spatial clustering of applications with noise* — gęstościowa klasteryzacja przestrzenna z uwzględnieniem szumu). Gęstość w algorytmie DBSCAN jest zdefiniowana jako liczba punktów w określonym promieniu .

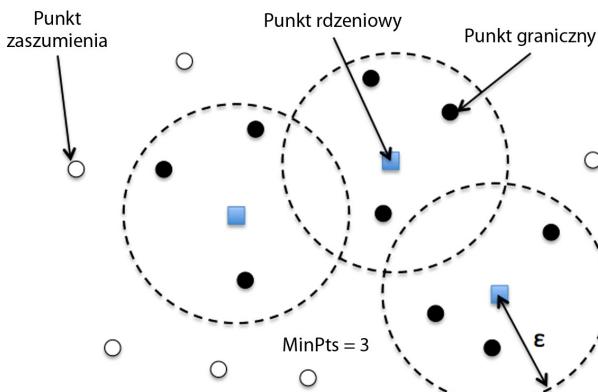
W tej technice każda próbka (punkt) otrzymuje specjalną etykietę wyznaczaną na podstawie następujących kryteriów:

- punkt jest uznawany za **rdzeniowy** (ang. *core point*), jeżeli w określonym od niego promieniu występuje co najmniej minimalna liczba (`MinPts`) sąsiadujących punktów w określonym promieniu ;
- punkt **graniczny** (ang. *border point*) ma w promieniu mniej sąsiadujących punktów od wartości minimalnej, ale znajduje się w promieniu punktu rdzeniowego;
- wszystkie pozostałe punkty niebędące rdzeniowymi ani granicznymi są uznawane za punkty **zaszumienia** (ang. *noise point*).

Po wyznaczeniu punktów rdzeniowych, granicznych i zaszumienia algorytm DBSCAN przeprowadza dwie proste czynności:

1. Tworzy oddzielne skupienie dla każdego punktu rdzeniowego lub połączonych grup punktów rdzeniowych (punkty rdzeniowe są ze sobą połączone, jeżeli odległość pomiędzy nimi nie przekracza długości promienia  $\epsilon$ ).
  2. Przydziela każdy punkt graniczny do klastra zawierającego odpowiedni punkt rdzeniowy.

Zanim przejdziemy do implementacji algorytmu DBSCAN, zaprezentuję podsumowanie naszej wiedzy na jego temat na rysunku 11.10.



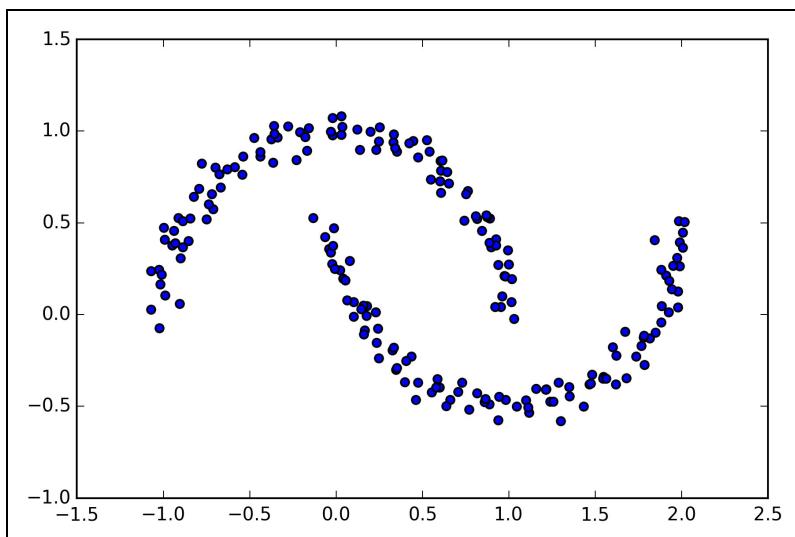
Rysunek 11.10. Mechanizm działania algorytmu DBSCAN

Jedną z głównych zalet algorytmu DBSCAN jest, w przeciwieństwie do algorytmu centroidów, brak założenia kulistości kształtu klastrów. Co więcej, od algorytmów centroidów i hierarchicznych DBSCAN odróżnia się brakiem konieczności przydzielania wszystkich punktów do skupień oraz możliwością usuwania szumów.

Przejdźmy do bardziej poglądowego przykładu; stworzymy nowy zestaw danych tworzących na wykresie sierpowate kształty i porównamy na nim skuteczność klasteryzacji centroidów, hierarchicznej oraz algorytmu DBSCAN:

```
>>> from sklearn.datasets import make_moons  
>>> X, y = make_moons(n_samples=200,  
...                      noise=0.05,  
...                      random_state=0)  
>>> plt.scatter(X[:,0], X[:,1])  
>>> plt.show()
```

Jak widać na wykresie z rysunku 11.11, wygenerowaliśmy dwie półksiężycowe grupy, z których każda zawiera po 100 punktów.



Rysunek 11.11. Wykres przykładowego zestawu danych tworzącego półksiężycowe struktury

Zacznijmy od algorytmu centroidów oraz klasteryzacji metodą pełnego wiązania, aby sprawdzić, czy te omówione wcześniej metody analizy skupień poradzą sobie z rozpoznaniem sierpowatych struktur jako osobnych klastrów. Posłuży nam do tego następujący kod:

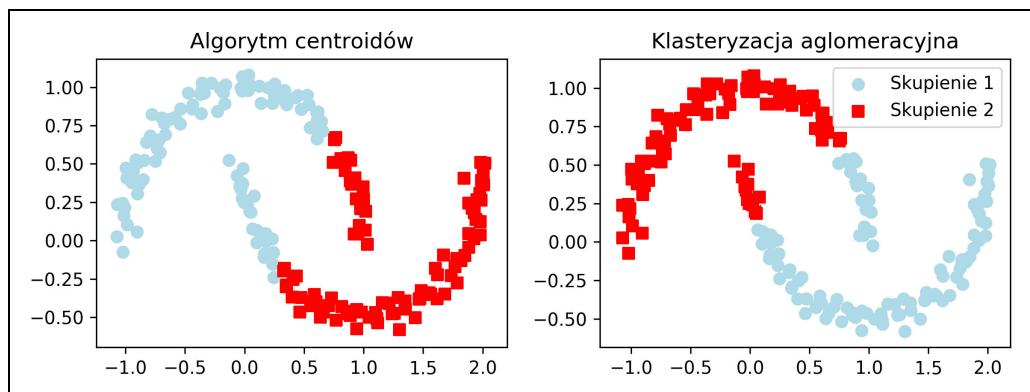
```
>>> f, (ax1, ax2) = plt.subplots(1, 2, figsize=(8,3))
>>> km = KMeans(n_clusters=2,
...                 random_state=0)
>>> y_km = km.fit_predict(X)
>>> ax1.scatter(X[y_km==0,0],
...               X[y_km==0,1],
...               c='lightblue',
...               marker='o',
...               s=40,
...               label='Skupienie 1')
>>> ax1.scatter(X[y_km==1,0],
...               X[y_km==1,1],
...               c='red',
...               marker='s',
...               s=40,
...               label='Skupienie 2')
>>> ax1.set_title('Algorytm centroidów')
>>> ac = AgglomerativeClustering(n_clusters=2,
...                                 affinity='euclidean',
...                                 linkage='complete')
>>> y_ac = ac.fit_predict(X)
>>> ax2.scatter(X[y_ac==0,0],
...               X[y_ac==0,1],
```

```

...           c='lightblue',
...           marker='o',
...           s=40,
...           label='Skupienie 1')
>>> ax2.scatter(X[y_ac==1,0],
...                 X[y_ac==1,1],
...                 c='red',
...                 marker='s',
...                 s=40,
...                 label='Skupienie 2')
>>> ax2.set_title('Klasteryzacja aglomeracyjna')
>>> plt.legend()
>>> plt.show()

```

Na podstawie rezultatów zaprezentowanych na rysunku 11.12 możemy stwierdzić, że algorytm centroidów nie poradził sobie z rozdzieleniem dwóch skupień, podobnie jak te skomplikowane kształty stanowią wyzwanie dla algorytmu hierarchicznej klasteryzacji.



Rysunek 11.12. Wykresy wynikowe zastosowania algorytmów centroidów i aglomeracyjnych w celu klasteryzacji półksiężycowatych kształtów

Pozostało nam już tylko sprawdzić działanie algorytmu DBSCAN (klasteryzacji gęstościowej) wobec testowego zestawu danych:

```

>>> from sklearn.cluster import DBSCAN
>>> db = DBSCAN(eps=0.2,
...               min_samples=5,
...               metric='euclidean')
>>> y_db = db.fit_predict(X)
>>> plt.scatter(X[y_db==0,0],
...               X[y_db==0,1],
...               c='lightblue',
...               marker='o',
...               s=40,
...

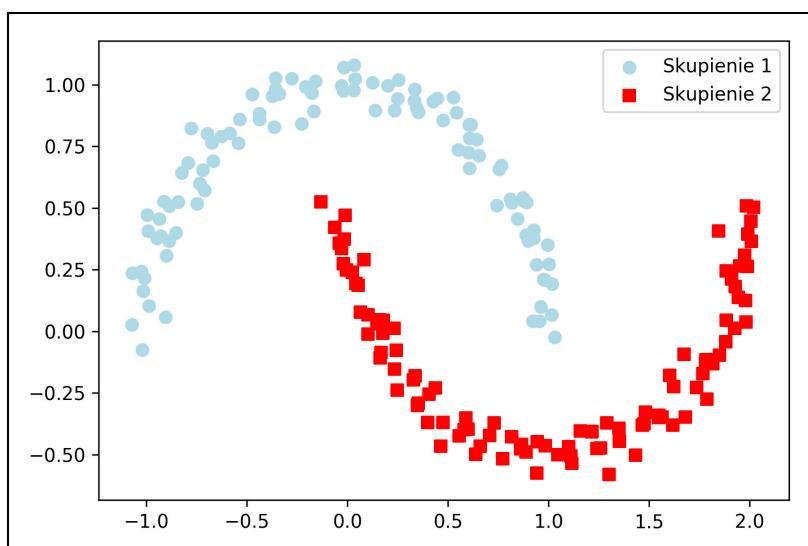
```

```

...           label='Skupienie 1')
>>> plt.scatter(X[y_db==1,0],
...                 X[y_db==1,1],
...                 c='red',
...                 marker='s',
...                 s=40,
...                 label='Skupienie 2')
>>> plt.legend()
>>> plt.show()

```

Algorytm DBSCAN w skuteczny sposób wykrywa sierpowate kształty, co pokazuje jedną z zalet tej metody (klasteryzacja danych tworzących niestandardowe kształty). Zostało to ukazane na rysunku 11.13.



Rysunek 11.13. Klasteryzacja danych tworzących sierpowate struktury za pomocą algorytmu DBSCAN

Nie możemy jednak zapominać o pewnych wadach algorytmu DBSCAN. Wraz ze wzrostem liczby cech w zestawie danych — przy uwzględnieniu stałego rozmiaru zestawu próbek uczących — wzmacnieniu ulega negatywny wpływ **kłatywy wymiarowości**. Jest to odczuwalne zwłaszcza w przypadku stosowania metryki odległości euklidesowej. Nie jest to jednak problem specyficzny wyłącznie dla algorytmu DBSCAN; dotyczy również innych modeli klasteryzacji wykorzystujących metrykę euklidesową, np. algorytmy centroidów czy aglomeracyjne. Poza tym w algorytmie DBSCAN mamy do czynienia z dwoma hiperparametrami ( $\text{MinPts}$  i  $\varepsilon$ ), które musimy zoptymalizować, jeśli zależy nam na dobrych rezultatach. Znalezienie dobrej kombinacji tych parametrów może być kłopotliwe w przypadku, gdy różnice w gęstości zestawu danych są względnie duże.

Dotychczas omówiliśmy trzy fundamentalne kategorie algorytmów analizy skupień: klasteryzację bazującą na prototypach (algorytm centroidów), hierarchiczną klasteryzację aglomeracyjną oraz klasteryzację gęstościową (reprezentowaną przez algorytm DBSCAN). Chciałbym jednak wspomnieć jeszcze o czwartej klasie bardziej złożonych algorytmów grupowania, które zostały pominięte w tym rozdziale: **grafowej analizie skupień** (ang. *graph-based clustering*). Prawdopodobnie najpopularniejszą rodziną takich algorytmów są algorytmy **klasteryzacji spektralnej** (ang. *spectral clustering algorithms*). Istnieje wiele różnych implementacji spektralnej analizy skupień, ale ich wspólną cechą jest stosowanie wektorów własnych macierzy podobieństwa do wykrywania współzależności w klastrach. Opis klasteryzacji spektralnej wykracza poza ramy niniejszej książki, ale zainteresowane osoby mogą zapoznać się ze znakomitym artykułem autorstwa Ulrike von Luxburg stanowiącym wprowadzenie do wspomnianego tematu (U. von Luxburg, *A Tutorial on Spectral Clustering*, „Statistics and Computing” 2007, nr 17 (4), s. 395 – 416), dostępnym bezpłatnie w serwisie arXiv: <https://arxiv.org/pdf/0711.0189v1.pdf>.

Zwróć uwagę, że w praktyce nie zawsze wiadomo, który algorytm będzie sprawiał się najlepiej wobec określonego zestawu danych, zwłaszcza jeżeli te dane są wielowymiarowe, co nierzaz utrudnia albo uniemożliwia ich wizualizację. Trzeba również podkreślić, że skuteczna klasteryzacja nie zależy wyłącznie od algorytmu lub jego hiperparametrów. Liczy się również dobór właściwej metryki odległości, a także duża wiedza z analizowanej dziedziny pozwalająca na skuteczniejsze eksperymentowanie z konfiguracją modelu.

## Podsumowanie

W tym rozdziale poznaliśmy trzy różne algorytmy analizy skupień pomagające w wykrywaniu ukrytych struktur lub informacji w danych. Rozpoczęliśmy od techniki bazującej na prototypach — algorytmie centroidów — w którym skupienia próbek przybierają kolisty kształt, w zależności od liczby zdefiniowanych centroidów. Klasteryzacja stanowi nienadzorowaną metodę uczenia, dlatego nie mamy do dyspozycji rzeczywistych etykiet pozwalających na ocenę skuteczności modelu. Z tego powodu przyjrzaliśmy się kilku wewnętrzny metrykom skuteczności, takim jak metoda łokcia czy analiza profilu, które ułatwiają ilościowe ujęcie jakości klasteryzacji.

Przeszliśmy następnie do odmiennej techniki analizy skupień: hierarchicznej, aglomeracyjnej klasteryzacji. W tym przypadku nie musimy odgórnie definiować liczby skupień, a wyniki możemy przedstawiać graficznie w postaci dendrogramu, co znacznie ułatwia interpretację rezultatów. Ostatnim z poznanych przez nas w tym rozdziale algorytmów jest DBSCAN, algorytm grupujący punkty na podstawie lokalnej gęstości, wykrywający odstające próbki oraz wyznaczający niestandardowe kształty skupień.

Po takiej wyprawie w dziedzinę uczenia nienadzorowanego możemy w końcu zawitać do kraju jednych z najciekawszych algorytmów nadzorowanego uczenia maszynowego: wielowarstwowych sieci neuronowych. Obecnie kategoria ta przeżywa swój renesans, a sieci neuronowe ponownie stanowią najbardziej ekscytujące zagadnienie w świecie uczenia maszynowego.

Dzięki niedawno zaprojektowanym algorytmom uczenia głębskiego, sieci neuronowe są uznawane za najnowocześniejszą technologię przydającą się do rozwiązywania wielu skomplikowanych zadań, takich jak rozpoznawanie obrazu lub mowy. W rozdziale 12., „Trenowanie sztucznych sieci neuronowych w rozpoznawaniu obrazu”, stworzymy od podstaw własną wielowarstwową sieć neuronową. Z kolei w rozdziale 13., „Równoległe przetwarzanie sieci neuronowych za pomocą biblioteki Theano”, wprowadzimy do użytku potężne technologie pozwalające na skuteczniejsze uczenie złożonych konstrukcji sieciowych.

# Trenowanie sztucznych sieci neuronowych w rozpoznawaniu obrazu

Być może zauważyleś, że tematyka **uczenia głębokiego** (ang. *deep learning*) zyskuje ostatnio coraz większą popularność i stanowi bez wątpienia najbardziej intrugującą dział uczenia maszynowego. Proces uczenia głębokiego możemy zdefiniować jako zestaw algorytmów zaprojektowanych w celu jak najsłuszniejszego trenowania wielowarstwowych **sztucznych sieci neuronowych** (ang. *artificial neural networks*). W tym rozdziale poznamy podstawowe pojęcia dotyczące tych sieci neuronowych, dzięki czemu zdobędziesz wiedzę pozwalającą na eksplorację tej jakże ekscytującej dziedziny uczenia maszynowego; a także narzędzia w postaci zaawansowanych, ciągle rozwijanych bibliotek Pythona przeznaczonych do uczenia głębokiego.

W tym rozdziale zajmiemy się następującymi zagadnieniami:

- podstawy teoretyczne wielowarstwowych sieci neuronowych,
- uczenie sieci neuronowych pod względem rozpoznawania obrazu,
- implementacja potężnego algorytmu wstecznej propagacji,
- usuwanie błędów z implementacji sieci neuronowych.

# Modelowanie złożonych funkcji przy użyciu sztucznych sieci neuronowych

Naszą podróż po krainie uczenia maszynowego rozpoczęliśmy (w rozdziale 2., „Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji”) od algorytmów uczących bazujących na sztucznych neuronach. Twory te stanowią elementy budulcowe wielowarstwowych sieci neuronowych, którym poświęcimy uwagę w niniejszym rozdziale. Podstawowe założenia i koncepcje dotyczące sieci neuronowych bazują na hipotezach i modelach wyjaśniających działanie ludzkiego mózgu w czasie rozwiązywania złożonych problemów. Model sieci neuronowych zyskuje od pewnego czasu coraz większą popularność, ale pierwsze badania na jego temat były przeprowadzane już w latach 40. ubiegłego wieku, gdy Warren McCulloch i Walter Pitts jako pierwsi opisali teoretyczny model działania komórki nerwowej. Jednak w kolejnych dziesięcioleciach po opracowaniu w latach 50. modelu perceptronu Rosenblatta (bazującego na modelu **neuronu McCullocha-Pittsa**) zainteresowanie tym tematem stopniowo malało, gdyż nikt nie był w stanie opracować dobrego sposobu uczenia wielowarstwowych sieci neuronowych. Fascynacja tym zagadnieniem odzyla dopiero w 1986 roku, gdy David E. Rumelhart, Geoffrey E. Hinton i Ronald J. Williams odkryli (ponownie) i popularyzowali algorytm **wstecznej propagacji** (ang. *backpropagation algorithm*) pozwalający na skuteczniejsze uczenie sieci neuronowych; zajmiemy się jego omówieniem w dalszej części rozdziału (D.E. Rumelhart, G.E. Hinton i R.J. Williams, *Learning Representations by Back-propagating Errors*, „Nature” 1986, nr 323 (6008), s. 533 – 536).

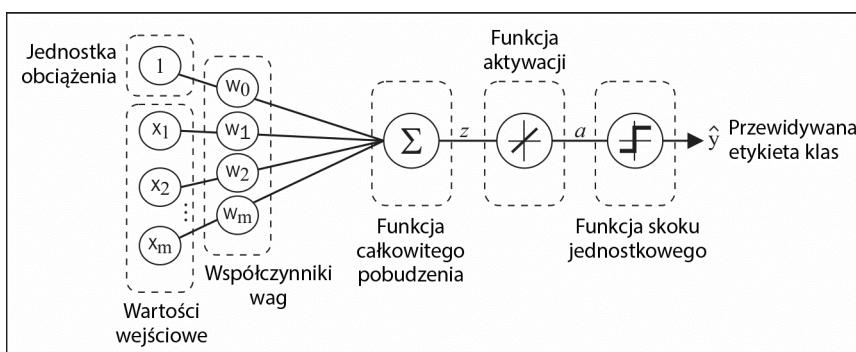
W ciągu ostatniego dziesięciolecia dokonano wielu przełomowych odkryć prowadzących do powstania tzw. algorytmów uczenia głębokiego, które są wykorzystywane do konstruowania detektorów **cech** (ang. *feature detector*) z nieoznaczonych danych, umożliwiających wstępne uczenie głębokich sieci neuronowych — składających się z wielu warstw. Sieci neuronowe nie tylko stanowią obiekt akademickich rozważań, lecz również budzą zainteresowanie wielkich koncernów, takich jak Facebook, Microsoft czy Google, które inwestują mnóstwo środków w badania nad sztucznymi sieciami neuronowymi i uczeniem głębokim. Obecnie złożone sieci neuronowe wspomagane algorytmami uczenia głębokiego uznawane są za najnowocześniejsze zdobyczne techniki służące do rozwiązywania skomplikowanych problemów, np. rozpoznawania obrazu czy głosu. Dobrymi przykładami produktów spotykanych w codziennym życiu, a wykorzystujących algorytmy uczenia głębokiego są silnik wyszukiwania obrazów firmy Google oraz Tłumacz Google — aplikacja mobilna automatycznie rozpoznająca tekst zamieszczony na rysunkach, tłumacząca go w czasie rzeczywistym na ponad 20 języków (<https://research.googleblog.com/2015/07/how-google-translate-squeezes-deep.html>).

Obecnie rozwijanych jest mnóstwo innych fascynujących zastosowań głębokich sieci neuronowych przez największe firmy technologiczne, jak silnik DeepFace z firmy Facebook, umożliwiający oznaczanie osób na zdjęciach (Y. Taigman, M. Yang, M. Ranzato i L. Wolf, *DeepFace: Closing the gap to human-level performance in face verification*, „Computer Vision and Pattern Recognition CVPR”, 2014 IEEE Conference, s. 1701 – 1708), a także DeepSpeech firmy Baidu, służący do obsługi zapytań w języku mandaryńskim (A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates i in., *DeepSpeech:*

*Scaling up end-to-end speech recognition*, arXiv preprint arXiv: 1412.5567, 2014). Do tego w przemyśle farmaceutycznym zaczęto ostatnio wykorzystywać techniki uczenia głębokiego do wykrywania leków i przewidywania działania nowych toksyn, a przeprowadzane analizy udowadniają, że metody te znacznie przewyższają skutecznością tradycyjne badania przesiewowe (T. Unterthiner, A. Mayr, G. Klambauer i S. Hochreiter, *Toxicity prediction using deep learning*, arXiv preprint arXiv: 1503.01445, 2015).

## Jednowarstwowa sieć neuronowa — powtórzenie

Rozdział ten jest w całości poświęcony wielowarstwowym sieciom neuronowym — mechanizmowi ich działania oraz sposobowi ich uczenia w celu rozwiązywania złożonych problemów. Zanim jednak przejdziemy do omówienia architektury wielowarstwowej sieci neuronowej, przypomnijmy sobie pewne pojęcia dotyczące jednowarstwowych sieci neuronowych, którymi zajmowaliśmy się w rozdziale 2., „Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji”, mianowicie algorytmu **ADaptacyjnego Liniowego NEuronu (ADALINE)**, zaprezentowanego na rysunku 12.1.



Rysunek 12.1. Jeszcze jedno spojrzenie na model ADALINE

W rozdziale 2., „Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji”, zaimplementowaliśmy algorytm Adaline przeprowadzający klasyfikację binarną i wykorzystaliśmy algorytm optymalizacyjny (**gradientu prostego**) do wyuczenia współczynników wag modelu. W każdej **epoce** (przebiegu algorytmu po zestawie danych uczących) aktualizowaliśmy wektor wag  $\mathbf{w}$  za pomocą następującej reguły uczenia:

$$\mathbf{w} := \mathbf{w} + \Delta \mathbf{w}, \quad \text{gdzie} \quad \Delta \mathbf{w} = -\eta \nabla J(\mathbf{w})$$

Innymi słowy, wyliczaliśmy gradient na podstawie całego zestawu uczącego i aktualizowaliśmy wagi modelu, wykonując krok w kierunku przeciwnym do gradientu  $\nabla J(\mathbf{w})$ . W celu znalezienia optymalnych wag modelu przeprowadziliśmy optymalizację funkcji celu zdefiniowaną jako funkcja kosztu — suma kwadratów błędów (SSE) —  $J(\mathbf{w})$ . Do tego przemnażaliśmy gradient przez **współczynnik uczenia**  $\eta$ , który ostrożnie dobraliśmy tak, aby zrównoważyć szybkość trenowania z ryzykiem pominięcia globalnego minimum funkcji kosztu.

Zgodnie z optymalizacją gradientu prostego po każdej epoce aktualizowaliśmy jednocześnie wszystkie wagi i w następujący sposób zdefiniowaliśmy pochodną cząstkową dla każdej wagi  $w_j$  wektora wag  $\mathbf{w}$ :

$$\frac{\partial}{\partial w_j} J(\mathbf{w}) = - \sum_i (y^{(i)} - a^{(i)}) x_j^{(i)}$$

Tutaj  $y^{(i)}$  oznacza docelową etykietę klas danej próbki  $x^{(i)}$ , natomiast  $a^{(i)}$  jest **aktywacją** neuronu — funkcją liniową w specjalnym przypadku modelu Adaline. Do tego zdefiniowaliśmy **funkcję aktywacji**  $\phi(\cdot)$ :

$$\phi(z) = z = a$$

W omawianym przypadku całkowite pobudzenie  $z$  stanowi liniową kombinację wag łączących warstwę wejściową z wyjściową:

$$z = \sum_j w_j x_j = \mathbf{w}^T \mathbf{x}$$

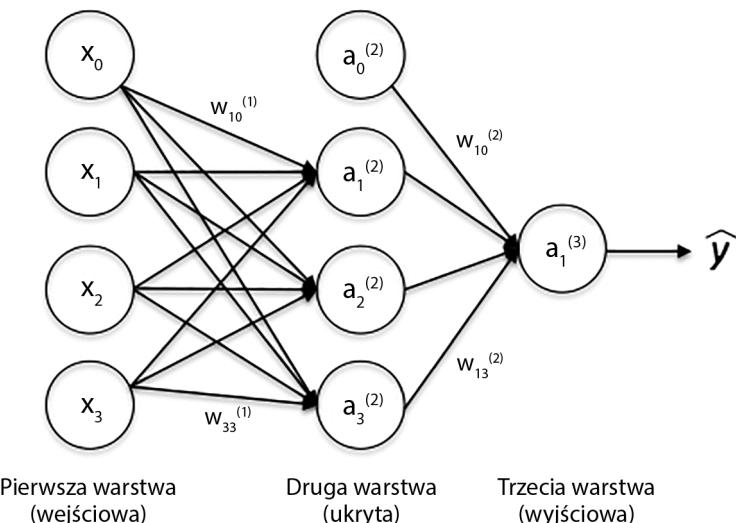
Funkcja aktywacji  $\phi(z)$  posłużyła nam do obliczania aktualizacji gradientu, natomiast wprowadziliśmy jeszcze **funkcję progową** (Heaviside'a), która przekształca odpowiedzi ciągłe na binarne etykiety klas służące do tworzenia prognoz:

$$\hat{y} = \begin{cases} 1 & \text{jeśli } g(z) \geq 0 \\ -1 & \text{jeśli } g(z) < 0 \end{cases}$$

Zwróć uwagę, że model Adaline nazywamy siecią jednowarstwową, mimo że składa się z dwóch warstw: wejściowej i wyjściowej. Wynika to z faktu, że istnieje tylko jedno połączenie pomiędzy obydwoma warstwami.

## Wstęp do wielowarstwowej architektury sieci neuronowych

W tym podrozdziale nauczymy się łączyć wiele pojedynczych neuronów w **wielowarstwową jednokierunkową sieć neuronową** (ang. *multi-layer feedforward neural network*); jest to specjalny przypadek sieci często nazywany również **wielowarstwowym perceptronem** (ang. *multi-layer perceptron* — **MLP**). Rysunek 12.2 przedstawia koncepcję trójwarstwowego modelu MLP; zawiera on warstwę wejściową, **warstwę ukrytą** oraz warstwę wyjściową. Jednostki warstwy ukrytej są w pełni połączone z jednostkami warstwy wejściowej, a jednostki warstwy wyjściowej — z jednostkami warstwy ukrytej. Jeżeli taki model ma więcej niż jedną warstwę ukrytą, nazywamy go **głęboką** siecią neuronową.



Rysunek 12.2. Model wielowarstwowej sieci neuronowej

W celu stworzenia głębszej architektury sieci MLP możemy dodawać kolejne warstwy ukryte. Możemy uznać liczbę jednostek i warstw w sieci neuronowej za dodatkowe **hiperparametry**, które chcemy zoptymalizować pod kątem określonego problemu za pomocą sprawdzianu krzyżowego, omówionego w rozdziale 6., „Najlepsze metody oceny modelu i strojenie parametryczne”.

Jednak gradienty błędu wyliczane za pomocą omawianego w dalszej części rozdziału algorytmu wstecznej propagacji stopniowo maleją wraz ze wzrostem liczby warstw w sieci. Problem **zanikającego gradientu** (ang. *vanishing gradient*) znacznie utrudnia uczenie modelu. Z tego powodu zaprojektowano specjalne algorytmy wstępniego uczenia głębszych sieci neuronowych — jest to właśnie **uczenie głębokie**.

Widzimy na rysunku 12.2, że oznaczamy  $i$ -tą jednostkę aktywacji w  $l$ -tej warstwie jako  $a_i^{(l)}$ , natomiast jednostki aktywacji  $a_0^{(1)}$  i  $a_0^{(2)}$  to **jednostki obciążenia** (ang. *bias units*) mające wartość 1. Aktywacja jednostek w warstwie wejściowej to nic innego jak ich wartości wejściowe wraz z jednostką obciążenia:

$$a^{(1)} = \begin{bmatrix} a_0^{(1)} \\ a_1^{(1)} \\ \vdots \\ a_m^{(1)} \end{bmatrix} = \begin{bmatrix} 1 \\ x_1^{(i)} \\ \vdots \\ x_m^{(i)} \end{bmatrix}$$

Każda jednostka w warstwie  $l$  jest połączona ze wszystkimi jednostkami warstwy  $l+1$  za pomocą współczynnika wag. Przykładowo połączenie pomiędzy  $k$ -tą jednostką w warstwie  $l$  z  $j$ -tą jednostką warstwy  $l+1$  można zapisać jako  $w_{j,k}^{(l)}$ . Zwróci uwagę, że indeks górnny  $i$  w wyrażeniu

$x_m^{(i)}$  oznaczają  $i$ -tą próbkę, nie warstwę. W dalszej części podrozdziału będziemy często pomijać indeks górny  $i$  w celu zachowania przejrzystości.

Pojedyncza jednostka w warstwie wyjściowej wystarczy do binarnego klasyfikowania próbek, ale na rysunku 12.2 przedstawiłem uogólnioną postać wielowarstwowej sieci neuronowej, pozwalającą na przeprowadzanie wieloklasowej klasyfikacji poprzez generalizację techniki **jeden przeciw wszystkim (OvA)**. Aby lepiej zrozumieć mechanizm działania tego modelu, przypomnijmy sobie wprowadzoną w rozdziale 4., „Tworzenie dobrych zbiorów uczących — wstępne przetwarzanie danych”, gorącojedynkową reprezentację zmiennych kategoryzujących. Możemy np. w następujący sposób zakodować trzy etykiety klas ze znanego już nam zestawu danych Iris ( $0 = \text{Setosa}$ ,  $1 = \text{Versicolor}$ ,  $2 = \text{Virginica}$ ):

$$0 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, 1 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, 2 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

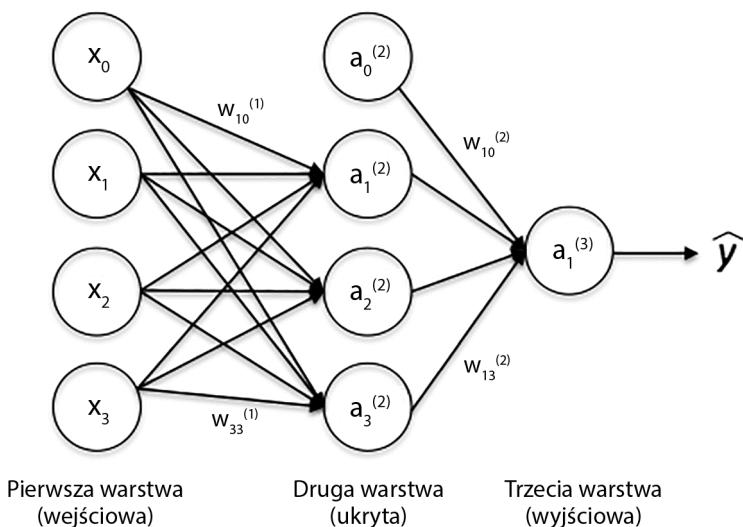
Taka „zerojedynkowa” reprezentacja wektorowa pozwala nam wykonywać czynności klasyfikujące na dowolnej liczbie unikatowych etykiet klas zawartych w zestawie danych uczących.

Osobom dopiero zapoznającym się z sieciami neuronowymi taka notacja (indeksy dolne i górne) może początkowo wydawać się nieco zagmatwana. Być może zastanawiasz się, dlaczego zapisalem  $w_{j,k}^{(l)}$ , a nie  $w_{k,j}^{(l)}$  przy wyznaczaniu współczynnika wag łączącego  $k$ -tą jednostkę warstwy  $l$  z  $j$ -tą jednostką warstwy  $l+1$ . To, co teraz wydaje się nieco dziwne, stanie się całkowicie zrozumiałe w dalszej części rozdziału, gdy będziemy przeprowadzać wektoryzację modelu sieci neuronowej. Zapiszemy np. wagi łączące warstwę wejściową z ukrytą jako macierz  $\mathbf{W}(1) \in \mathbb{R}^{h \times [m+1]}$ , gdzie  $h$  oznacza liczbę ukrytych jednostek, a  $m+1$  — liczbę jednostek wejściowych wraz z jednostką obciążenia. Musimy przyswoić sobie tę notację w celu zrozumienia informacji zawartych w dalszej części rozdziału, dlatego rysunek 12.3 stanowi opisową ilustrację uproszczonego wielowarstwowego perceptronu 3-4-3.

## Aktywacja sieci neuronowej za pomocą propagacji w przód

Zajmiemy się teraz omówieniem procesu **propagacji w przód** (ang. *forward propagation*) służącego do wyliczania wyniku w modelu MLP. Aby zrozumieć jego rolę w kontekście uczenia perceptronu wielowarstwowego, podzielimy czynność uczenia modelu na trzy podstawowe etapy:

1. Począwszy od warstwy wejściowej, rozsyłamy do przodu wzorce danych uczących wzduż całej sieci w celu uzyskania odpowiedzi.
2. Na podstawie otrzymanego wyniku obliczamy błąd, który chcemy zminimalizować za pomocą funkcji kosztu opisanej w dalszej części rozdziału.
3. Rozprzestrzeniamy wstecz otrzymany błąd, wyliczamy jego pochodną przy uwzględnieniu wszystkich wag, a następnie aktualizujemy model.



Rysunek 12.3. Uproszczony model wielowarstwowego perceptronu (3-4-3)

W końcu, po powtórzeniu tych czynności dla wielu epok oraz wytrenowaniu wag modelu MLP, możemy wykorzystać propagację w przód w celu obliczenia odpowiedzi, a także wprowadzić funkcję progową po to, aby otrzymać przewidywane etykiety klas w notacji „gorącojedynkowej”, omówionej w poprzednim podrozdziale.

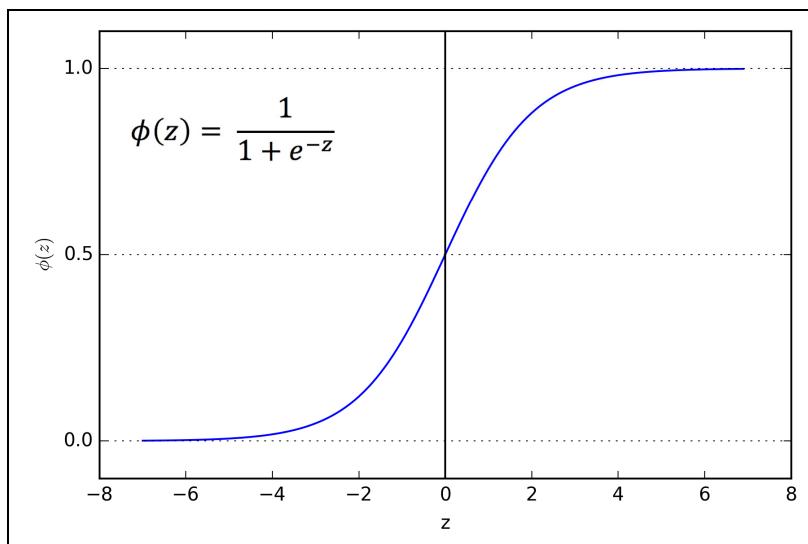
Przyjrzyjmy się teraz poszczególnym etapom procesu propagacji w przód dążącym do wygenerowania odpowiedzi na podstawie wzorców z danych uczących. Każda jednostka warstwy ukrytej jest powiązana ze wszystkimi jednostkami warstwy wejściowej, dlatego możemy obliczyć aktywację  $a_1^{(2)}$  w następujący sposób:

$$\begin{aligned} z_1^{(2)} &= a_0^{(1)} w_{1,0}^{(1)} + a_1^{(1)} w_{1,1}^{(1)} + \dots + a_m^{(1)} w_{1,m}^{(1)} \\ a_1^{(2)} &= \phi(z_1^{(2)}) \end{aligned}$$

Tutaj  $z_1^{(2)}$  jest pobudzeniem całkowitym, a  $\phi(\cdot)$  — funkcją aktywacji, która musi być różniczkowalna w celu uczenia metodą gradientową wag łączących neurony. Abyśmy mogli rozwiązywać złożone problemy, takie jak rozpoznawanie obrazu, musimy wprowadzić do modelu MLP nielinowe funkcje aktywacji, np. **logistyczną (sigmoidalną)** funkcję aktywacji, którą stosowaliśmy w modelu **regresji logistycznej** (rozdział 3., „Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn”):

$$\phi(z) = \frac{1}{1 + e^{-z}}$$

Jak pamiętamy, funkcja sigmoidalna przyjmuje kształt krzywej s-kształtnej odwzorowującej przekształcenie całkowitego pobudzenia  $z$  w rozkład logistyczny miesiący się w przedziale od 0 do 1; krzywa ta przechodzi przez oś  $y$  w punkcie  $z = 0$ , co możemy zobaczyć na rysunku 12.4.



Rysunek 12.4. Wykres funkcji sigmoidalnej — powtóżenie

Model MLP stanowi klasyczny przykład **jednokierunkowej** (ang. *feed forward*) sztucznej sieci neuronowej. Termin „jednokierunkowa” oznacza, że każda warstwa stanowi warstwę wejściową dla kolejnej warstwy, bez użycia zapętleń; jej przeciwieństwem jest omówiona w dalszej części rozdziału **rekurencyjna sieć neuronowa** (ang. *recurrent neural network*). Z kolei termin „wielowarstwowy perceptron” wydaje się nieco nie na miejscu, ponieważ sztuczne neurony w tym modelu są zazwyczaj jednostkami sigmoidalnymi, a nie **perceptronami**. Możemy uznawać neurony modelu MLP za jednostki regresji logistycznej zwracające wartości ciągłe w przedziale pomiędzy 0 i 1.

W celu zachowania czytelności i dużej skuteczności używanego kodu będziemy zapisywać aktywację w zwięzlejszej postaci, przy użyciu podstawowych pojęć z algebry liniowej, co pozwoli nam na **wektoryzację** naszej implementacji w bibliotece NumPy i uniknięciu wielu zagnieżdzonych oraz kosztownych obliczeniowo pętli **for**:

$$\begin{aligned} z^{(2)} &= W^{(1)} a^{(1)} \\ a^{(2)} &= \phi(z^{(2)}) \end{aligned}$$

Zapis  $h$ , stosowany poniżej, należy rozumieć jako  $h+1$ , gdyż uwzględniamy również jednostkę obciążenia (należy tu także uwzględnić zgodność wymiarów wektorów).

Tutaj  $a^{(1)}$  oznacza nasz  $[m+1] \times 1$ -wymiarowy wektor cech dla próbki  $x^{(i)}$  wraz z jednostką obciążenia.  $W^{(1)}$  jest  $h \times [m+1]$ -wymiarową macierzą wag, gdzie  $h$  definiuje liczbę ukrytych jednostek w sieci neuronowej. Po przeprowadzeniu mnożenia macierzowo-wektorskiego otrzymujemy  $h \times 1$ -wymiarowy wektor całkowitego pobudzenia  $z^{(2)}$  umożliwiający obliczenie aktywacji  $a^{(2)}$  (gdzie  $a^{(2)} \in \mathbb{R}^{h \times 1}$ ). Możemy to uogólnić na wszystkie  $n$  próbki uczących:

$$\mathbf{Z}^{(2)} = \mathbf{W}^{(1)} \left[ \mathbf{A}^{(1)} \right]^T$$

$\mathbf{A}^{(1)}$  jest tu  $n \times [m+1]$ -wymiarową macierzą, a w wyniku mnożenia dwóch macierzy otrzymamy  $h \times n$ -wymiarową macierz całkowitego pobudzenia  $\mathbf{Z}^{(2)}$ . Na koniec przemnażamy wszystkie wartości macierzy całkowitego pobudzenia przez funkcję aktywacji  $\phi(\cdot)$  i wyliczamy w ten sposób  $h \times n$ -wymiarową macierz aktywacji  $\mathbf{A}^{(2)}$  dla następnej warstwy (w naszym przykładzie wyjściowej):

$$\mathbf{A}^{(2)} = \phi(\mathbf{Z}^{(2)})$$

W analogiczny sposób możemy zapisać zwektryzowaną postać aktywacji warstwy wyjściowej:

$$\mathbf{Z}^{(3)} = \mathbf{W}^{(2)} \mathbf{A}^{(2)}$$

Mnożymy tu  $t \times h$ -wymiarową macierz  $\mathbf{W}^{(2)}$  ( $t$  oznacza liczbę jednostek wyjściowych) przez  $h \times n$ -wymiarową macierz  $\mathbf{A}^{(2)}$  w celu uzyskania  $t \times n$ -wymiarowej macierzy  $\mathbf{Z}^{(3)}$  (jej kolumny tworzą wartości wyjściowe dla każdej próbki).

Pozostało nam już tylko wprowadzić sigmoidalną funkcję aktywacji, aby otrzymywać wynikowe wartości ciągłe:

$$\mathbf{A}^{(3)} = \phi(\mathbf{Z}^{(3)}), \quad \mathbf{A}^{(3)} \in \mathbb{R}^{t \times n}$$

## Klasyfikowanie pisma odręcznego

Cały poprzedni podrozdział poświęciliśmy na opis teoretycznych podstaw sieci neuronowych, co może być nieco przytłaczające dla osób dopiero rozpoczynających przygodę z tą dziedziną uczenia maszynowego. Zanim przejdziemy do omówienia algorytmu aktualizującego wagę w modelu MLP (algorytmu wstępnej propagacji), oderwijmy się na chwilę od teorii i sprawdźmy, jak się spisuje wielowarstwowa sieć neuronowa w akcji.

Teoria sieci neuronowych jest dość skomplikowanym zagadnieniem, dlatego chciałbym polecić dwa dodatkowe źródła informacji opisujące szczegółowo część tematów poruszanych w niniejszym rozdziale:

T. Hastie, J. Friedman i R. Tibshirani, *The Elements of Statistical Learning*, t. 2, Springer, 2009.

C.M. Bishop i in., *Pattern Recognition and Machine Learning*, t. 1, Springer, New York 2006.

W tym podrozdziale wyuczyszmy nasz pierwszy model wielowarstwowej sieci neuronowej klasyfikującej zapisane odręcznie cyfry przechowywane w popularnym zestawie danych **MNIST** (ang. *Mixed National Institute of Standards and Technology* — w wolnym tłumaczeniu mieszany zestaw danych Narodowego Instytutu Standaryzacji i Technologii), stworzonym przez Yanna LeCuna i in.; zbiór ten stanowi powszechnie stosowany wzorzec dla algorytmów uczenia maszynowego (Y. LeCun, L. Bottou, Y. Bengio i P. Haffner, *Gradient-based Learning Applied to Document Recognition*, „Proceedings of the IEEE” November 1998, nr 86 (11), s. 2278 – 2324).

## Zestaw danych MNIST

Zestaw danych MNIST jest dostępny publicznie pod adresem <http://yann.lecun.com/exdb/mnist/> i składa się z czterech części:

- **zbior obrazów uczących:** *train-image-idx3-ubyte.gz* (spakowany plik 9,9 MB, po rozpakowaniu 47 MB; 60 000 próbek),
- **zbior etykiet zestawu uczącego:** *train-labels-idx1-ubyte.gz* (spakowany plik 29 kB, po rozpakowaniu 60 kB; 60 000 etykiet),
- **zbior obrazów testowych:** *t10k-images-idx3-ubyte.gz* (spakowany plik 1,6 MB; po rozpakowaniu 7,8 MB; 10 000 próbek),
- **zbior etykiet zestawu testowego:** *t10k-labels-idx1-ubyte.gz* (spakowany plik 5 kB; po rozpakowaniu 10 kB; 10 000 etykiet).

Zestaw danych MNIST został utworzony z dwóch zbiorów danych amerykańskiego **Narodowego Instytutu Standaryzacji i Technologii (NIST)**. Na dane uczące składają się ręcznie zapisane cyfry pochodzące od 250 osób — połowa próbek została utworzona przez uczniów szkoły średniej, a druga połowa przez pracowników amerykańskiego Biura Cenzusowego. Taka sama proporcja dotyczy próbek tworzących zestaw danych testowych.

Po pobraniu plików zalecam ich rozpakowanie za pomocą narzędzia gzip (użytkownicy Uniksa/Linuksa) z poziomu terminalu — wystarczy użyć poniższego polecenia w katalogu zawierającym pliki MNIST:

```
gzip *ubyte.gz -d
```

Z kolei osoby pracujące w systemie Windows mogą do rozpakowania tych plików użyć dowolnego archiwizatora. Obrazy są przechowywane w formacie bajtowym i będziemy je wczytywać do tablic NumPy, które następnie posłużą nam do uczenia i testowania implementacji MLP:

```
import os
import struct
import numpy as np

def load_mnist(path, kind='train'):
    """wczytuje dane MNIST umieszczone w katalogu roboczym"""
    labels_path = os.path.join(path,
        '%s-labels.idx1-ubyte' % kind)
    # ...
```

```
% kind)
images_path = os.path.join(path,
                           '%s-images.idx3-ubyte'
                           % kind)

with open(labels_path, 'rb') as lbpath:
    magic, n = struct.unpack('>II',
                             lbpath.read(8))
    labels = np.fromfile(lbpath,
                         dtype=np.uint8)

with open(images_path, 'rb') as imgpath:
    magic, num, rows, cols = struct.unpack(">IIII",
                                            imgpath.read(16))
    images = np.fromfile(imgpath,
                         dtype=np.uint8).reshape(len(labels), 784)
return images, labels
```

Funkcja `load_mnist` zwraca dwie tablice; pierwsza z nich to  $n \times m$ -wymiarowa tablica NumPy (`images`), gdzie  $n$  oznacza liczbę próbek, a  $m$  — cech. Zestaw danych uczących składa się z 60 000 próbek, a danych testowych z 10 000 cyfr. Poszczególne rysunki mają rozmiar 28×28 pikseli; każdemu pikselowi przyporządkowana jest wartość odcienia szarości. W naszym przypadku rozwijamy te rysunki na jednowymiarowe wektory symbolizujące rzędy w macierzy rysunku (784 punkty na każdy rzęd lub rysunek). Druga tabela (`labels`) zwracana przez funkcję `load_mnist` zawiera odpowiednie zmienne docelowe — etykiety klas (liczby całkowite od 0 do 9) dla rysunków odręcznie pisanych cyfr.

Sposób wczytywania rysunku może na początku wydawać się nieco dziwny:

```
magic, n = struct.unpack('>II', lbpath.read(8))
labels = np.fromfile(lbpath, dtype=np.int8)
```

Aby zrozumieć, co oznaczają te dwa wiersze kodu, spójrzmy na opis zestawu danych umieszczony na stronie MNIST:

[przesunięcie]	[typ]	[wartość]	[opis]
0000	32-bitowa liczba całkowita	0x00000801 (2049)	magiczna liczba
→(najpierw bajt MSB)			
0004	32-bitowa liczba całkowita	60000	liczba elementów
0008	bajt bez znaku	??	etykieta
0009	bajt bez znaku	??	etykieta
.....			
xxxx	bajt bez znaku	??	etykieta

Za pomocą powyższych dwóch wierszy kodu najpierw wczytujemy **magiczną liczbę** (ang. *magic number*), stanowiącą opis protokołu pliku, a także **liczbę elementów** (`n`) z bufora pliku; dopiero teraz następuje wczytywanie kolejnych bajtów do tabeli NumPy za pomocą metody `fromfile`. Wartość `>II` w parametrze `fmt`, który przekazaliśmy jako argument do funkcji `struct.unpack` składa się z dwóch części:

- >: jest to grubokońcowość (definiuje kolejność przechowywania sekwencji bajtów); jeżeli nie spotkałeś się wcześniej z pojęciem **grubokońcowości** (ang. *big-endian*) lub **cienkokońcowości** (ang. *small-endian*), warto zapoznać się z artykułem *Endianness* na Wikipedii (<https://en.wikipedia.org/wiki/Endianness>), a po polsku z artykułem *ByteOrder — kolejność bajtów* ze strony <http://blog.malcom.pl/2013/byteorder-kolejnosc-bajtow.html>.
- I: to bezznakowa liczba całkowita.

Po uruchomieniu poniższego kodu wczytamy 60 000 instancji uczących, jak również 10 000 wystąpień testowych z katalogu *mnist*, w którym rozpakowaliśmy pobrane pliki:

```
>>> X_train, y_train = load_mnist('mnist', kind='train')
>>> print('Rzędy: %d, kolumny: %d'
...      % (X_train.shape[0], X_train.shape[1]))
Rzędy: 60000, kolumny: 784
>>> X_test, y_test = load_mnist('mnist', kind='t10k')
>>> print('Rzędy: %d, kolumny: %d'
...      % (X_test.shape[0], X_test.shape[1]))
Rzędy: 10000, kolumny: 784
```

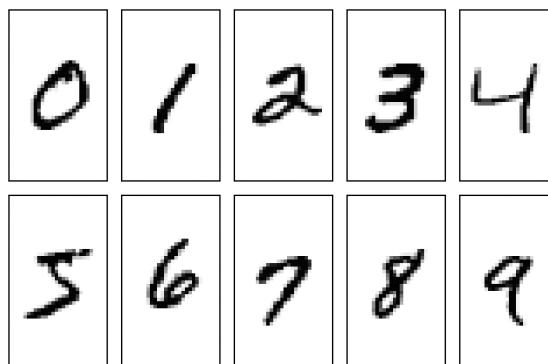
Warto byloby sprawdzić, jak wyglądają obrazy z zestawu danych MNIST, dlatego wizualizujemy przykładowe próbki poszczególnych cyfr arabskich, przekształcając 784-pikselowe wektory z macierzy cech w pierwotne obrazy o rozmiarze  $28 \times 28$  pikseli, które wyświetlimy za pomocą funkcji *imshow*:

```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(nrows=2, ncols=5, sharex=True,
...                         sharey=True,)
>>> ax = ax.flatten()
>>> for i in range(10):
...     img = X_train[y_train == i][0].reshape(28, 28)
...     ax[i].imshow(img, cmap='Greys', interpolation='nearest')
>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```

Naszym oczom powinien ukazać się wykres składający się z 10 obrazów przedstawiających poszczególne cyfry arabskie (rysunek 12.5).

Poza tym możemy stworzyć wykres różnych odmian tej samej cyfry, żeby sprawdzić, jak poszczególne rysunki różnią się pomiędzy sobą:

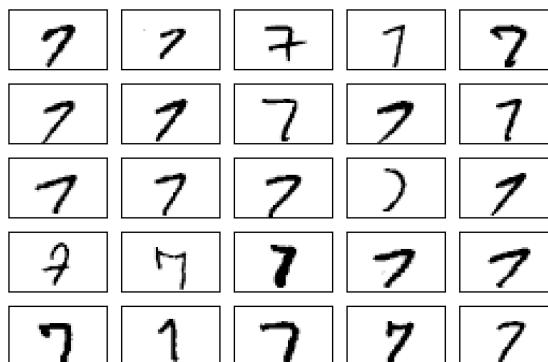
```
>>> fig, ax = plt.subplots(nrows=5,
...                         ncols=5,
...                         sharex=True,
...                         sharey=True,)
>>> ax = ax.flatten()
>>> for i in range(25):
```



Rysunek 12.5. Wykres zawierający przykładowe, odręcznie napisane cyfry arabskie

```
...     img = X_train[y_train == 7][i].reshape(28, 28)
...     ax[i].imshow(img, cmap='Greys', interpolation='nearest')
>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```

Po uruchomieniu powyższego kodu zostanie wyświetcone pierwsze 25 wariacji cyfry 7 (rysunek 12.6).



Rysunek 12.6. Wykres różnych odmian jednej cyfry

Możemy ewentualnie zapisać obrazy MNIST do plików CSV, aby móc pracować na nich w aplikacjach nieobsługujących formatu bajtowego. Chciałbym jednak zwrócić uwagę, że pliki CSV zabierają znacznie więcej miejsca na dysku:

- *obrazy\_uczące.csv* — 1,1 GB,
- *etykiety\_uczące.csv* — 1,4 MB,
- *obrazy\_testowe.csv* — 187 MB,
- *etykiety\_testowe.csv* — 144 kB.

Jeżeli zdecydujesz się na zapisanie zestawu danych do formatu CSV, uruchom poniższy kod w bieżącej sesji Pythona po wczytaniu danych do tablic NumPy:

```
>>> np.savetxt('obrazy_uczące.csv', X_train,
...             fmt='%i', delimiter=',')
>>> np.savetxt('etykiety_uczące.csv', y_train,
...             fmt='%i', delimiter=',')
>>> np.savetxt('obrazy_testowe.csv', X_test,
...             fmt='%i', delimiter=',')
>>> np.savetxt('etykiety_testowe.csv', y_test,
...             fmt='%i', delimiter=',')
```

Po zapisaniu plików CSV możemy wczytać je do środowiska Python za pomocą funkcji `genfromtxt`:

```
>>> X_train = np.genfromtxt('obrazy_uczące.csv',
...                         dtype=int, delimiter=',')
>>> y_train = np.genfromtxt('etykiety_uczące.csv',
...                         dtype=int, delimiter=',')
>>> X_test = np.genfromtxt('obrazy_testowe.csv',
...                         dtype=int, delimiter=',')
>>> y_test = np.genfromtxt('etykiety_testowe.csv',
...                         dtype=int, delimiter=',')
```

Wczytanie danych z plików CSV zajmuje jednak o wiele więcej czasu, dlatego zalecam pozostawanie w miarę możliwości przy oryginalnym formacie bajtowym.

## Implementacja wielowarstwowego perceptronu

Zajmiemy się teraz implementacją modelu MLP składającego się z jednej warstwy wejściowej, ukrytej i wyjściowej; posłużymy się nim do klasyfikowania obrazów z zestawu danych MNIST. Starałem się stworzyć jak najbardziej przejrzysty i prosty kod, ale może się on wydawać na początku nieco skomplikowany, dlatego zachęcam do pobrania kodu źródłowego ze strony wydawnictwa Helion, gdyż został on opatrzony komentarzami, a podświetlone elementy składni ułatwiają analizę algorytmu. Jeżeli nie używasz cyfrowych notatników uzupełniających książkę, zalecam skopiowanie kodu do skryptu Pythona w bieżącym katalogu roboczym, np. do pliku `siećneuronowa.py`, który możesz następnie importować do bieżącej sesji Pythona przy użyciu następującego polecenia:

```
from siećneuronowa import NeuralNetMLP
```

Kod zawiera elementy, których jeszcze nie omawialiśmy, takie jak algorytm wstępnej propagacji, ale jego większa część przypomina w dużej mierze implementację modelu Adaline opisaną w rozdziale 2., „Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji”, a także fragmenty wykorzystujące przeanalizowany na początku rozdziału algorytm jednokierunkowej propagacji. Nie martw się, jeżeli nie wszystko będzie od razu zrozumiałe; wieloma fragmentami

kodu zajmiemy się w dalszej części rozdziału. Teraz jednak umieszczenie całego kodu powinno nam bardziej pomóc w zrozumieniu czekającego nas jeszcze opisu teoretycznego:

```

import numpy as np
from scipy.special import expit
import sys

class NeuralNetMLP(object):
    def __init__(self, n_output, n_features, n_hidden=30,
                 l1=0.0, l2=0.0, epochs=500, eta=0.001,
                 alpha=0.0, decrease_const=0.0, shuffle=True,
                 minibatches=1, random_state=None):
        np.random.seed(random_state)
        self.n_output = n_output
        self.n_features = n_features
        self.n_hidden = n_hidden
        self.w1, self.w2 = self._initialize_weights()
        self.l1 = l1
        self.l2 = l2
        self.epochs = epochs
        self.eta = eta
        self.alpha = alpha
        self.decrease_const = decrease_const
        self.shuffle = shuffle
        self.minibatches = minibatches

    def _encode_labels(self, y, k):
        onehot = np.zeros((k, y.shape[0]))
        for idx, val in enumerate(y):
            onehot[val, idx] = 1.0
        return onehot

    def _initialize_weights(self):
        w1 = np.random.uniform(-1.0, 1.0,
                               size=self.n_hidden*(self.n_features + 1))
        w1 = w1.reshape(self.n_hidden, self.n_features + 1)
        w2 = np.random.uniform(-1.0, 1.0,
                               size=self.n_output*(self.n_hidden + 1))
        w2 = w2.reshape(self.n_output, self.n_hidden + 1)
        return w1, w2

    def _sigmoid(self, z):
        # wartość zmiennej expit wynosi  $1.0/(1.0 + \exp(-z))$ 
        return expit(z)

    def _sigmoid_gradient(self, z):
        sg = self._sigmoid(z)
        return sg * (1 - sg)

```

```

def _add_bias_unit(self, X, how='column'):
    if how == 'column':
        X_new = np.ones((X.shape[0], X.shape[1]+1))
        X_new[:, 1:] = X
    elif how == 'row':
        X_new = np.ones((X.shape[0]+1, X.shape[1]))
        X_new[1:, :] = X
    else:
        raise AttributeError('Atrybut `how` musi mieć wartość `column` lub
                             `row`')
    return X_new

def _feedforward(self, X, w1, w2):
    a1 = self._add_bias_unit(X, how='column')
    z2 = w1.dot(a1.T)
    a2 = self._sigmoid(z2)
    a2 = self._add_bias_unit(a2, how='row')
    z3 = w2.dot(a2)
    a3 = self._sigmoid(z3)
    return a1, z2, a2, z3, a3

def _L2_reg(self, lambda_, w1, w2):
    return (lambda_/2.0) * (np.sum(w1[:, 1:] ** 2) \
                           + np.sum(w2[:, 1:] ** 2))

def _L1_reg(self, lambda_, w1, w2):
    return (lambda_/2.0) * (np.abs(w1[:, 1:]).sum() \
                           + np.abs(w2[:, 1:]).sum())

def _get_cost(self, y_enc, output, w1, w2):
    term1 = -y_enc * (np.log(output))
    term2 = (1 - y_enc) * np.log(1 - output)
    cost = np.sum(term1 - term2)
    L1_term = self._L1_reg(self.l1, w1, w2)
    L2_term = self._L2_reg(self.l2, w1, w2)
    cost = cost + L1_term + L2_term
    return cost

def _get_gradient(self, a1, a2, a3, z2, y_enc, w1, w2):
    # propagacja wsteczna
    sigma3 = a3 - y_enc
    z2 = self._add_bias_unit(z2, how='row')
    sigma2 = w2.T.dot(sigma3) * self._sigmoid_gradient(z2)
    sigma2 = sigma2[1:, :]
    grad1 = sigma2.dot(a1)
    grad2 = sigma3.dot(a2.T)

    # regularyzacja
    grad1[:, 1:] += self.l2 * w1[:, 1:]

```

```

grad1[:, 1:] += self.l1 * np.sign(w1[:, 1:])
grad2[:, 1:] += self.l2 * w2[:, 1:]
grad2[:, 1:] += self.l1 * np.sign(w2[:, 1:])

return grad1, grad2

def predict(self, X):
    a1, z2, a2, z3, a3 = self._feedforward(X, self.w1, self.w2)
    y_pred = np.argmax(z3, axis=0)
    return y_pred

def fit(self, X, y, print_progress=False):
    self.cost_ = []
    X_data, y_data = X.copy(), y.copy()
    y_enc = self._encode_labels(y, self.n_output)

    delta_w1_prev = np.zeros(self.w1.shape)
    delta_w2_prev = np.zeros(self.w2.shape)

    for i in range(self.epochs):

        # współczynnik uczenia adaptacyjnego
        self.eta /= (1 + self.decrease_const*i)

        if print_progress:
            sys.stderr.write(
                '\rEpoka: %d/%d' % (i+1, self.epochs))
            sys.stderr.flush()

        if self.shuffle:
            idx = np.random.permutation(y_data.shape[0])
            X_data, y_enc = X_data[idx], y_enc[:,idx]

        mini = np.array_split(range(
            y_data.shape[0]), self.minibatches)
        for idx in mini:

            # propagacja jednokierunkowa
            a1, z2, a2, z3, a3 = self._feedforward(
                X_data[idx], self.w1, self.w2)
            cost = self._get_cost(y_enc=y_enc[:, idx],
                                  output=a3,
                                  w1=self.w1,
                                  w2=self.w2)
            self.cost_.append(cost)

            # oblicza gradient za pomocą wstecznej propagacji
            grad1, grad2 = self._get_gradient(a1=a1, a2=a2,
                                              a3=a3, z2=z2,

```

```

        y_enc=y_enc[:, idx],
        w1=self.w1,
        w2=self.w2)

    # aktualizuje wagи
    delta_w1, delta_w2 = self.eta * grad1,\
                         self.eta * grad2
    self.w1 -= (delta_w1 + (self.alpha * delta_w1_prev))
    self.w2 -= (delta_w2 + (self.alpha * delta_w2_prev))
    delta_w1_prev, delta_w2_prev = delta_w1, delta_w2

    return self

```

Zainicjujmy nowy model MLP 784-50-10, sieć neuronową składającą się z 784 jednostek wejściowych (`n_features`), 50 jednostek ukrytych (`n_hidden`) i 10 jednostek wyjściowych (`n_output`):

```

>>> nn = NeuralNetMLP(n_output=10,
...                     n_features=X_train.shape[1],
...                     n_hidden=50,
...                     l2=0.1,
...                     l1=0.0,
...                     epochs=1000,
...                     eta=0.001,
...                     alpha=0.001,
...                     decrease_const=0.00001,
...                     shuffle=True,
...                     minibatches=50,
...                     random_state=1)

```

Zauważysz zapewne, że w naszej implementacji modelu MLP wprowadziliśmy pewne nowe parametry, które teraz zostaną omówione:

- `l2`: parametr regularyzacji L2 pozwalający na zmniejszenie stopnia przetrenowania; gdybyśmy zastąpili go wartością `l1`, otrzymalibyśmy parametr  $\lambda$  regularyzacji L1;
- `epochs`: liczba przebiegów algorytmu po zestawie danych uczących;
- `eta`: współczynnik uczenia  $\eta$ ;
- `alpha`: parametr dla uczenia momentowego, określający część wartości poprzedniego gradientu dodawaną do zaktualizowanych wag w celu przyśpieszenia nauki:  

$$\Delta\mathbf{w}_t = \eta \nabla J(\mathbf{w}_t) + \alpha \Delta\mathbf{w}_{t-1}$$
 (gdzie  $t$  oznacza bieżący etap/bieżącą epokę);
- `decrease_const`: stała redukcji  $d$  stanowiąca element adaptacyjnego współczynnika uczenia  $n$ , malejącego z upływem czasu w celu uzyskania lepszej zbieżności:  $\eta / 1 + t \times d$ ;
- `shuffle`: przed każdą epoką tasujemy zestaw danych uczących po to, aby zapobiec cykliczności przebiegów;
- `minibatches`: rozdzielanie danych uczących na  $k$  podzbiorów w każdej epoce. W celu szybszego uczenia gradient jest dla każdego podzbioru obliczany oddziennie (nie jest wyliczany dla całego zbioru próbek uczących).

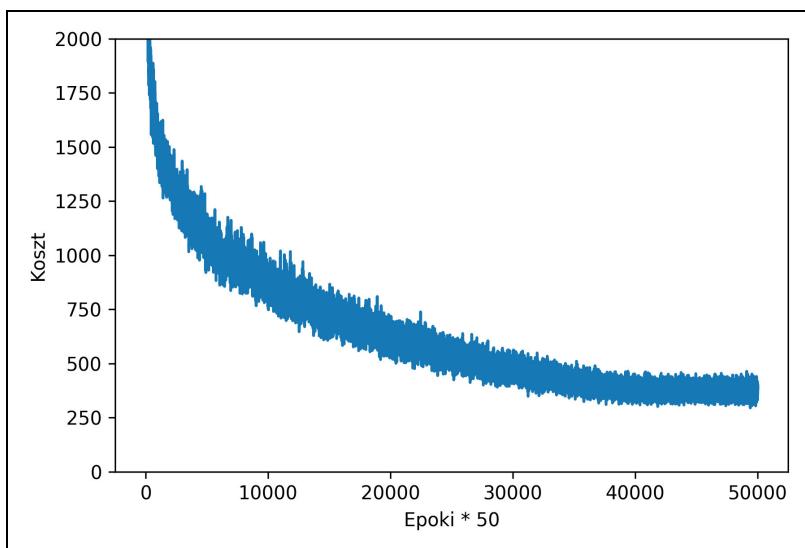
Wyuczymy teraz model MLP za pomocą 60 000 próbek tworzących przetasowany zbiór danych uczących MNIST. Przed uruchomieniem kodu zwróć uwagę, że uczenie sieci neuronowej może zająć 10 – 30 minut na standardowym komputerze stacjonarnym:

```
>>> nn.fit(X_train, y_train, print_progress=True)
Epoka: 1000/1000
```

Podobnie jak w przypadku implementacji modelu Adaline, zachowujemy koszt wyliczony dla każdej epoki na liście `cost_`, którą możemy teraz wizualizować, by się upewnić, czy algorytm optymalizujący uzyskał zbieżność. Stworzymy wykres co pięćdziesiątego etapu dla każdego z 50 podzbiorów danych uczących (50 podzbiorów×1000 epok). Wykorzystamy do tego poniższy kod:

```
>>> plt.plot(range(len(nn.cost_)), nn.cost_)
>>> plt.ylim([0, 2000])
>>> plt.ylabel('Koszt')
>>> plt.xlabel('Epoki * 50')
>>> plt.tight_layout()
>>> plt.show()
```

Jak widać na rysunku 12.7, wykres funkcji kosztu wygląda na bardzo zaszumiony. Wynika to z faktu, że uczyliśmy naszą sieć neuronową za pomocą podzbiorów próbek uczących, co stanowi jedną z odmian stochastycznego spadku wzduż gradientu.



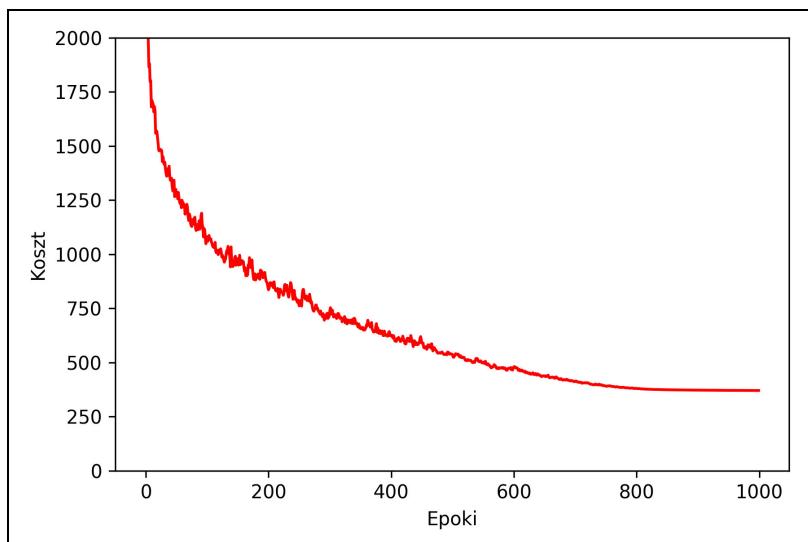
Rysunek 12.7. Wykres funkcji kosztu dla wyuczonej sieci neuronowej

Możemy wyczytać z tego wykresu, że algorytm optymalizujący osiągnął zbieżność mniej więcej po 800 epokach ( $40\ 000/50 = 800$ ), mimo to wygenerujmy wygładzony wykres kosztu w funkcji epok poprzez uśrednienie interwałów podzbiorów. Pozwoli nam na to następujący fragment kodu:

```
>>> batches = np.array_split(range(len(nn.cost_)), 1000)
>>> cost_ary = np.array(nn.cost_)
>>> cost_avgs = [np.mean(cost_ary[i]) for i in batches]

>>> plt.plot(range(len(cost_avgs)),
...           cost_avgs,
...           color='red')
>>> plt.ylim([0, 2000])
>>> plt.ylabel('Koszt')
>>> plt.xlabel('Epoki')
>>> plt.tight_layout()
>>> plt.show()
```

Wykres na rysunku 12.8 jest bardziej wyrazisty i możemy na nim dokładnie zobaczyć, że algorytm stał się zbieżny po wykonaniu 800 przebiegów.



Rysunek 12.8. Wykres funkcji kosztu po uśrednieniu interwałów podzbiorów

Sprawdźmy teraz skuteczność modelu, obliczając dokładność przewidywań:

```
>>> y_train_pred = nn.predict(X_train)
>>> acc = np.sum(y_train == y_train_pred, axis=0) / X_train.shape[0]
>>> print('Dokładność wobec danych uczących: %.2f%%' % (acc * 100))
Dokładność wobec danych uczących: 97.59%
```

Nasz model poprawnie klasyfikuje większość danych uczących, ale czy algorytm będzie skutecznie uogólniał działanie również wobec nieznanych próbek? Wyliczmy dokładność dla 10 000 obrazów testowych:

```
>>> y_test_pred = nn.predict(X_test)
>>> acc = np.sum(y_test == y_test_pred, axis=0) / X_test.shape[0]
>>> print('Dokładność wobec danych testowych: %.2f%%' % (acc * 100))
Dokładność wobec danych testowych: 95.62%
```

Na podstawie niewielkiej rozbieżności pomiędzy dokładnością dla danych uczących i testowych możemy stwierdzić, że nasz model jest jedynie w nieznacznym stopniu przetrenowany. W celu dalszego dostrajania modelu moglibyśmy zmienić liczbę ukrytych jednostek, wartości parametrów regularyzacji, współczynnik uczenia, wartość stałej redukcji lub współczynnik adaptacyjnego uczenia za pomocą technik opisanych w rozdziale 6., „Najlepsze metody oceny modelu i strojenie parametryczne” (potraktuj to jako zadanie domowe).

Sprawdźmy, z jakimi cyframi ma problem nasz model MLP:

```
>>> miscl_img = X_test[y_test != y_test_pred][:25]
>>> correct_lab = y_test[y_test != y_test_pred][:25]
>>> miscl_lab = y_test_pred[y_test != y_test_pred][:25]

>>> fig, ax = plt.subplots(nrows=5,
...                         ncols=5,
...                         sharex=True,
...                         sharey=True,)
>>> ax = ax.flatten()
>>> for i in range(25):
...     img = miscl_img[i].reshape(28, 28)
...     ax[i].imshow(img,
...                  cmap='Greys',
...                  interpolation='nearest')
...     ax[i].set_title('%d r: %d p: %d'
...                     % (i+1, correct_lab[i], miscl_lab[i]))
>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```

Powinniśmy teraz ujrzeć macierz składającą się z 25 obrazów cyfr (rysunek 12.9). Pierwsza liczba w opisie rysunków stanowi indeks analizowanego obrazu, druga wskazuje na rzeczywistą etykietę klas (*r*), a trzecia przedstawia prognozowaną etykietę klas (*p*).

Jak widać, rozpoznanie niektórych z tych cyfr może stanowić wyzwanie nawet dla ludzkiego mózgu. Przykładowo cyfra 9 jest klasyfikowana przez algorytm jako 3 lub 8, jeśli dolna część cyfry jest haczykowato zagięta (obrazki nr 4, 18 i 20).

1) r: 5 p: 6	2) r: 4 p: 9	3) r: 2 p: 3	4) r: 9 p: 8	5) r: 5 p: 3
				
6) r: 8 p: 7	7) r: 4 p: 2	8) r: 6 p: 0	9) r: 8 p: 9	10) r: 9 p: 7
				
11) r: 2 p: 7	12) r: 5 p: 3	13) r: 5 p: 0	14) r: 2 p: 7	15) r: 5 p: 3
				
16) r: 2 p: 3	17) r: 6 p: 0	18) r: 9 p: 8	19) r: 3 p: 5	20) r: 9 p: 3
				
21) r: 8 p: 2	22) r: 4 p: 1	23) r: 8 p: 3	24) r: 7 p: 1	25) r: 8 p: 2
				

Rysunek 12.9. Wykres niektórych z niewłaściwie sklasyfikowanych cyfr

## Trenowanie sztucznej sieci neuronowej

Skoro już się przekonaliśmy, jak działa sieć neuronowa, i przyjrzyliśmy się jej kodowi, możemy skoncentrować się na opisie niektórych koncepcji użytych do uczenia wag, takich jak logistyczna funkcja kosztu czy algorytm wstecznej propagacji.

## Obliczanie logistycznej funkcji kosztu

Logistyczna funkcja kosztu, którą zaimplementowaliśmy jako metodę `_get_cost`, jest w rzeczywistości bardzo prosta do zrozumienia, ponieważ w istocie mamy do czynienia z tą samą funkcją kosztu, którą zajmowaliśmy się przy okazji omawiania regresji logistycznej w rozdziale 3., „Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn”.

$$J(\mathbf{w}) = -\sum_{i=1}^n y^{(i)} \log(a^{(i)}) + (1 - y^{(i)}) \log(1 - a^{(i)})$$

Parametr  $a^{(i)}$  oznacza tutaj sigmoidalną aktywację  $i$ -tej jednostki w którejś z warstw; obliczamy ją jako funkcję propagacji jednokierunkowej:

$$a^{(i)} = \phi(z^{(i)})$$

Dodajmy teraz wyrażenie **regularyzacji** umożliwiające zmniejszenie stopnia przetrenowania. Jak zapewne pamiętasz z poprzednich rozdziałów, wyrażenia regularyzacji L2 i L1 możemy zdefiniować następująco (pamiętaj, że nie regularizujemy jednostek obciążenia):

$$L2 = \lambda \|\mathbf{w}\|_2^2 = \lambda \sum_{j=1}^m w_j^2 \quad \text{oraz} \quad L1 = \lambda \|\mathbf{w}\|_1^1 = \lambda \sum_{j=1}^m |w_j|$$

W naszej implementacji wykorzystujemy obydwie formy regularyzacji, ale dla uproszczenia skupimy się jedynie na wyrażeniu L2. Poniższe koncepcje jednak dotyczą również regularyzacji L1. Po dodaniu wyrażenia regularyzacji L2 do logistycznej funkcji kosztu otrzymujemy następujące równanie:

$$J(\mathbf{w}) = - \left[ \sum_{i=1}^n y^{(i)} \log(a^{(i)}) + (1 - y^{(i)}) \log(1 - a^{(i)}) \right] + \frac{\lambda}{2} \|\mathbf{w}\|_2^2$$

Nasz model MLP służy do klasyfikacji wieloklasowej, dlatego powyższa funkcja zwraca wektor wyjściowy składający się z  $t$  elementów, które musimy porównać z  $t \times 1$ -wymiarowym wektorem docelowym przekształconym do postaci „gorącojedynkowej”. Przykładowo aktywacja trzeciej warstwy i docelowej klasy (w tym przypadku klasy 2) dla określonej próbki może wyglądać tak, jak przedstawiono poniżej:

$$a^{(3)} = \begin{bmatrix} 0,1 \\ 0,9 \\ \vdots \\ 0,3 \end{bmatrix}, \quad y = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

Musimy zatem uogólnić logistyczną funkcję kosztu na wszystkie jednostki aktywacji  $j$  w naszej sieci. Zatem funkcja kosztu (bez wyrażenia regularyzacji) przybiera następującą postać:

$$J(\mathbf{w}) = - \sum_{i=1}^n \sum_{j=1}^t y_j^{(i)} \log(a_j^{(i)}) + (1 - y_j^{(i)}) \log(1 - a_j^{(i)})$$

Indeks górny  $i$  stanowi wskaźnik danej próbki w zestawie danych uczących.

Zaprezentowane poniżej uogólnione wyrażenie regularizacji może początkowo wydawać się dość skomplikowane, ale wyliczamy tu jedynie sumę wszystkich wag w warstwie  $l$  (nie uwzględniamy tu jednostki obciążenia), które dodaliśmy do pierwszej kolumny:

$$\begin{aligned} J(\mathbf{w}) &= - \left[ \sum_{i=1}^n \sum_{j=1}^t y_j^{(i)} \log(\phi(z_j^{(i)})) + (1 - y_j^{(i)}) \log(1 - \phi(z_j^{(i)})) \right] + \\ &+ \frac{\lambda}{2} \sum_{l=1}^{L-1} \sum_{i=1}^u \sum_{j=1}^{u_{l+1}} (w_{j,i}^{(l)})^2 \end{aligned}$$

Poniższe wyrażenie jest karą wyliczoną z regularizacji L2:

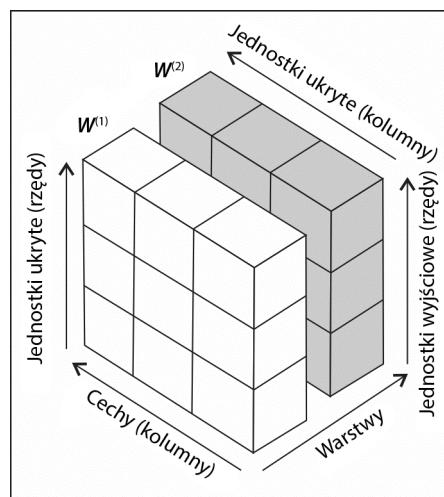
$$\frac{\lambda}{2} \sum_{l=1}^{L-1} \sum_{i=1}^u \sum_{j=1}^{u_{l+1}} (w_{j,i}^{(l)})^2$$

Przypominam, że naszym celem jest minimalizacja funkcji kosztu  $J(\mathbf{w})$ . Z tego powodu musimy wyliczyć pochodną cząstkową macierzy  $\mathbf{W}$  przy uwzględnieniu każdej wagi we wszystkich warstwach sieci:

$$\frac{\partial}{\partial w_{j,i}^{(l)}} J(\mathbf{W})$$

W dalszej części rozdziału będziemy omawiać algorytm wstecznej propagacji, pozwalający na obliczanie tych pochodnych cząstkowych w celu minimalizacji funkcji kosztu.

Zwróć uwagę, że tablica  $\mathbf{W}$  składa się z kilku macierzy. W przypadku wielowarstwowego perceptronu zawierającego jedną warstwę ukrytą występuje macierz wag  $\mathbf{W}^{(l)}$  łącząca warstwy wejściową z ukrytą, a także macierz wag  $\mathbf{W}^{(2)}$  stanowiąca połączenie warstw ukrytej z wyjściową. Macierz  $\mathbf{W}$  możemy wyobrazić sobie tak, jak zaprezentowano na rysunku 12.10.



Rysunek 12.10. Intuicyjny schemat budowy macierzy  $\mathbf{W}$

Rysunek 12.10 zawiera uproszczony schemat, dlatego może się wydawać, że macierze  $\mathbf{W}^{(l)}$  i  $\mathbf{W}^{(2)}$  składają się z tej samej liczby rzędów i kolumn; zdarza się tak niemal wyłącznie wtedy, gdy zainicjujemy model MLP zawierający po tyle samo jednostek ukrytych i wyjściowych oraz cech wejściowych.

Jeżeli ciągle wydaje Ci się to niejasne, poczekaj do następnego podrozdziału, w którym wyjaśnię dokładniej wymiarowość macierzy  $\mathbf{W}^{(l)}$  i  $\mathbf{W}^{(2)}$  w kontekście algorytmu wstecznej propagacji.

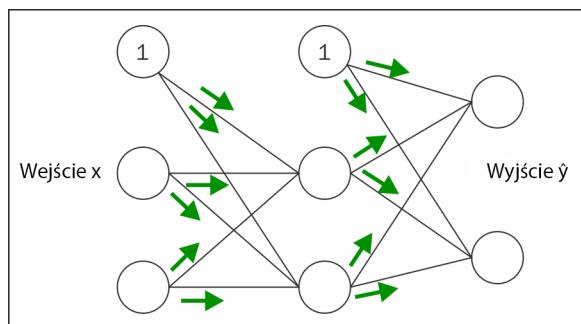
## Uczenie sieci neuronowych za pomocą algorytmu wstecznej propagacji

Zajmiemy się teraz analizą algorytmu wstecznej propagacji pozwalającego na bardzo skuteczne aktualizowanie wag w sieci neuronowej. W zależności od stopnia znajomości aparatu matematycznego niektórym Czytelnikom poniższe wzory mogą wydawać się na początku dość złożone. Wiele osób woli podejść systemowe — analizować poszczególne równania krok po kroku — co pozwala im zrozumieć działanie algorytmu. Jeżeli jednak wolisz poznać algorytm wstecznej propagacji bez zagłębiania się w notację matematyczną, zalecam zajrzenie w pierwszej kolejności do następnego podrozdziału, „Ujęcie intuicyjne algorytmu wstecznej propagacji”, a następnie powrót tutaj.

Wiemy już z poprzedniego podrozdziału, jak wyliczać koszt w postaci różnicy pomiędzy aktywacją ostatniej warstwy a docelową etykietą klas. Zobaczmy teraz, w jaki sposób algorytm wstecznej propagacji aktualizuje wagę w modelu MLP; zaimplementowaliśmy go jako metodę `_get_gradient`. Jak pamiętamy z początku niniejszego rozdziału, musimy najpierw przeprowadzić propagację jednokierunkową (w przód) po to, aby uzyskać wartość aktywacji warstwy wyjściowej, co sformułowaliśmy następująco:

$$\begin{aligned} \mathbf{Z}^{(2)} &= \mathbf{W}^{(1)} \left[ \mathbf{A}^{(1)} \right]^T && \text{(całkowite pobudzenie warstwy ukrytej)} \\ \mathbf{A}^{(2)} &= \phi(\mathbf{Z}^{(2)}) && \text{(aktywacja warstwy ukrytej)} \\ \mathbf{Z}^{(3)} &= \mathbf{W}^{(2)} \mathbf{A}^{(2)} && \text{(całkowite pobudzenie warstwy wyjściowej)} \\ \mathbf{A}^{(3)} &= \phi(\mathbf{Z}^{(3)}) && \text{(aktywacja warstwy wyjściowej)} \end{aligned}$$

Podsumowując, rozsyłamy w jednym kierunku wejściowe cechy poprzez widoczne na rysunku 12.11 połączenia sieci.



Rysunek 12.11. Schemat jednokierunkowej propagacji w sieci neuronowej

W algorytmie wstecznej propagacji przesyłamy błąd z prawej strony w lewo (od wyjścia do wejścia). Rozpoczynamy od obliczenia wektora błędu dla warstwy wyjściowej:

$$\delta^{(3)} = a^{(3)} - y$$

Parametr  $y$  stanowi tu wektor rzeczywistych etykiet klas.

Następnie wyliczamy błąd dla warstwy ukrytej:

$$\delta^{(2)} = (\mathbf{W}^{(2)})^T \delta^{(3)} * \frac{\partial \phi(z^{(2)})}{\partial z^{(2)}}$$

Wyrażenie  $\frac{\partial \phi(z^{(2)})}{\partial z^{(2)}}$  to po prostu pochodna sigmoidalnej funkcji aktywacji, którą zaimplementowaliśmy jako obiekt `_sigmoid_gradient`:

$$\frac{\partial \phi(z^{(2)})}{\partial z^{(2)}} = (a^{(2)} * (1 - a^{(2)}))$$

Zwróć uwagę, że w tym kontekście symbol gwiazdki (\*) oznacza iloczyn elementów wektora.

Dla osób zainteresowanych sposobem otrzymania pochodnej funkcji aktywacji zamieściłem poniżej wyrowadzenie wzorów krok po kroku:

$$\begin{aligned}\phi'(z) &= \frac{\partial}{\partial z} \left( \frac{1}{1 + e^{-z}} \right) = \\ &= \frac{e^{-z}}{(1 + e^{-z})^2} = \\ &= \frac{1 + e^{-z}}{(1 + e^{-z})^2} - \left( \frac{1}{1 + e^{-z}} \right)^2 = \\ &= \frac{1}{(1 + e^{-z})} - \left( \frac{1}{1 + e^{-z}} \right)^2 = \\ &= \phi(z) - (\phi(z))^2 = \\ &= \phi(z)(1 - \phi(z)) = \\ &= a(1 - a)\end{aligned}$$

Aby lepiej zrozumieć, w jaki sposób obliczamy wyrażenie  $\delta^{(3)}$ , przyjrzyjmy się dokładniej temu procesowi. W jednym z powyższych równań wykorzystaliśmy do mnożenia transpozycję  $(\mathbf{W}^{(2)})^T$  macierzy  $\mathbf{W}^{(2)}$  o wymiarach  $t \times h$ ; parametr  $t$  jest liczbą wyjściowych etykiet klas, a  $h$  — liczbą ukrytych jednostek. Uzyskaliśmy macierz transponowaną  $(\mathbf{W}^{(2)})^T$  o wymiarach  $h \times t$  wraz z  $t \times l$ -wymiarowym wektorem  $\delta^{(3)}$ . Następnie przeprowadziliśmy mnożenie czynników pomiędzy  $(\mathbf{W}^{(2)})^T \delta^{(3)}$  a wyrażeniem  $(a^{(2)} * (1 - a^{(2)}))$ , które również jest  $t \times l$ -wymiarowym wektorem. W końcu, po wyliczeniu wyrażeń  $\delta$ , możemy zapisać wzór na pochodną funkcji kosztu:

$$\frac{\partial}{\partial w_{i,j}^{(l)}} J(\mathbf{W}) = a_j^{(l)} \delta_i^{(l+1)}$$

Teraz musimy zliczyć pochodną cząstkową każdego  $j$ -tego węzła w warstwie  $l$  oraz  $i$ -ty błąd z warstwy  $l+1$ :

$$\Delta_{i,j}^{(l)} := \Delta_{i,j}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$$

Pamiętaj, że musimy wyliczyć  $\Delta_{i,j}^{(l)}$  dla każdej próbki w zestawie danych uczących. Z tego powodu łatwiej zaimplementować tę operację w zwiękotyzowanej postaci, tak jak to zrobiliśmy w modelu MLP:

$$\Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)} (\mathbf{A}^{(l)})^T$$

Po zliczeniu pochodnych cząstkowych możemy dodać wyrażenie regularizacji:

$$\Delta^{(l)} := \Delta^{(l)} + \lambda^{(l)} \text{ (pomijamy jednostkę obciążenia)}$$

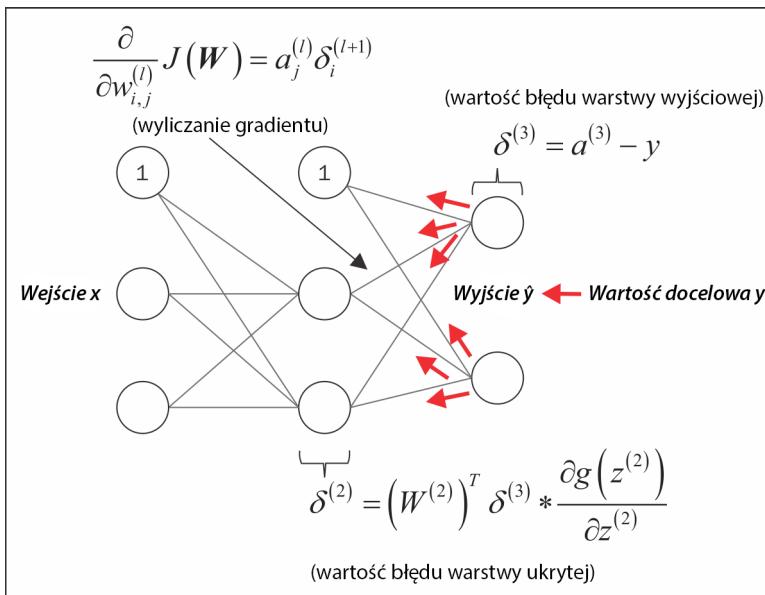
Na koniec, już po obliczeniu gradientów, możemy zaktualizować wagi, wykonując krok w kierunku przeciwnym do gradientu:

$$\mathbf{W}^{(l)} := \mathbf{W}^{(l)} - \eta \Delta^{(l)}$$

Na rysunku 12.12 umieściłem podsumowanie działania algorytmu wstecznej propagacji.

## Ujęcie intuicyjne algorytmu wstecznej propagacji

Algorytm wstecznej propagacji został opracowany już niemal 30 lat temu, mimo to ciągle pozostaje jedną z najskuteczniejszych technik uczenia sztucznych sieci neuronowych. W tym podrozdziale postaram się wyjaśnić bardziej obrazowo działanie tego fascynującego algorytmu.



Rysunek 12.12. Schemat działania algorytmu wstecznej propagacji

W swojej istocie algorytm wstecznej propagacji stanowi bardzo skuteczny obliczeniowo sposób wyliczania pochodnych złożonej funkcji kosztu. Naszym zadaniem jest użycie tych pochodnych do aktualizowania współczynników wag stanowiących parametry wielowarstwowej sieci neuronowej. Wyzwaniem w tym przypadku jest zazwyczaj olbrzymia liczba tych współczynników w wielowymiarowej przestrzeni cech. W przeciwieństwie do funkcji kosztu, omówionych w poprzednich rozdziałach, powierzchnia błędu w funkcji kosztu sieci neuronowej nie jest ani wypukła, ani gładka. Wielowymiarowa powierzchnia kosztu zawiera mnóstwo nierówności (lokalnych minimów), którymi musimy się zająć, zanim wyszukamy globalne minimum funkcji.

Być może znasz z lekcji matematyki regułę łańcuchową. Pozwala ona na różniczkowanie złożonej, zagnieżdżonej funkcji, takiej jak  $f(g(x)) = y$ , którą rozbijamy na główne składowe:

$$\frac{\partial y}{\partial x} = \frac{\partial f}{\partial g} \frac{\partial g}{\partial x}$$

W kontekście algebry informatycznej zaprojektowano szereg technik (zwanych **automatycznym różniczkowaniem**) bardzo skutecznie rozwiązujecych tego typu problemy. Jeżeli chcesz się dowiedzieć więcej na temat automatycznego różniczkowania w dziedzinie uczenia maszynowego, polecam zapoznanie się z następującym artykułem: A.G. Baydin i B.A. Pearlmutter, *Automatic Differentiation of Algorithms for Machine Learning*, arXiv preprint arXiv: 1404.7456, 2014 — jest on bezpłatnie dostępny pod adresem <https://arxiv.org/pdf/1404.7456.pdf>.

Różniczkowanie automatyczne występuje w dwóch formach: metody **w przód** (ang. *forward*) i **odwrotne** (ang. *reverse*). Algorytm propagacji wstecznej stanowi specjalny przypadek

odwrotnego różniczkowania automatycznego. Podstawowym problemem różniczkowania automatycznego w przód jest duży koszt obliczeniowy kalkulowania pochodnej funkcji złożonej, ponieważ w każdej warstwie musimy przemnażać duże macierze (macierze Jacobiego), które na koniec mnożymy jeszcze przez wektor w celu uzyskania wyniku. W metodzie odwrotnej przeprowadzamy obliczenia w przeciwnym kierunku (od prawej do lewej): najpierw przemnażamy macierz przez wektor, dzięki czemu otrzymujemy następny wektor, który mnożymy przez kolejną macierz itd. Iloczyn macierzowo-wektorowy jest znacznie skuteczniejszy obliczeniowo od mnożenia dwóch macierzy, dlatego właśnie algorytm wstępnej propagacji stanowi jedno z najpopularniejszych rozwiązań uczenia sieci neuronowych.

## Usuwanie błędów w sieciach neuronowych za pomocą sprawdzania gradientów

Implementacje sztucznych sieci neuronowych bywają bardzo skomplikowane i zawsze warto **samodzielnie** sprawdzać, czy algorytm wstępnej propagacji został właściwie zaprogramowany. W tym podrozdziale zapoznamy się z prostą procedurą zwaną **sprawdzaniem gradientów** (ang. *gradient checking*), która umożliwia porównanie gradientów sieci wyliczonych analitycznie z gradientami obliczonymi metodami numerycznymi. Metoda ta nie jest specyficzna dla sieci neuronowych ze sprzężeniem w przód i może być wykorzystana w wielu innych architekturach gradientowych sieci neuronowych. Nawet jeżeli zamierzasz wprowadzić prostsze algorytmy wykorzystujące optymalizację gradientową, np. regresję liniową, regresję logistyczną czy maszyny wektorów nośnych, zawsze warto sprawdzić, czy gradienty zostały poprawnie obliczone.

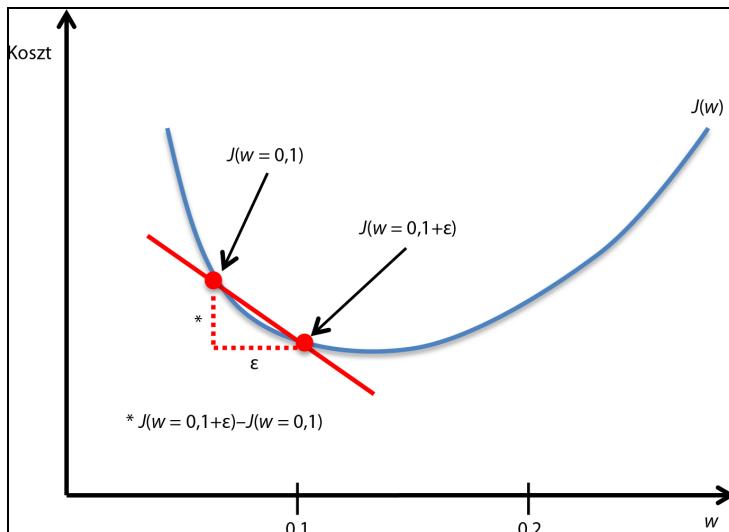
W poprzednich podrozdziałach zdefiniowaliśmy funkcję kosztu  $J(\mathbf{W})$ , gdzie  $\mathbf{W}$  stanowi macierz współczynników wag danej sieci neuronowej. Zwrót uwagę, że funkcja  $J(\mathbf{W})$  jest — oględzie mówiąc — „wieloczęściową” macierzą składającą się z macierzy  $\mathbf{W}(1)$  i  $\mathbf{W}(2)$  w modelu wielowarstwowego perceptronu zawierającego jedną jednostkę ukrytą. Zdefiniowaliśmy  $\mathbf{W}(1)$  jako  $h \times [m+1]$ -wymiarową macierz łączącą warstwę wejściową z warstwą ukrytą, gdzie  $h$  wyznacza liczbę ukrytych jednostek, a  $m$  — liczbę cech (jednostek wejściowych). Z kolei macierz  $\mathbf{W}(2)$  łączy warstwę ukrytą z warstwą wyjściową i ma wymiary  $t \times h$  ( $t$  oznacza liczbę jednostek wyjściowych). Następnie obliczyliśmy pochodną funkcji kosztu dla wagi  $w_{i,j}^{(l)}$  za pomocą wzoru:

$$\frac{\partial}{\partial w_{i,j}^{(l)}} J(\mathbf{W})$$

Pamiętaj, że aktualizujemy wagi, wykonując krok w kierunku przeciwnym do gradientu. W metodzie sprawdzania gradientu porównujemy otrzymany analitycznie wynik gradientu z wynikami uzyskanymi numerycznie:

$$\frac{\partial}{\partial w_{i,j}^{(l)}} J(\mathbf{W}) \approx \frac{J(w_{i,j}^{(l)} + \varepsilon) - J(w_{i,j}^{(l)})}{\varepsilon}$$

Parametr  $\varepsilon$  przyjmuje tu zazwyczaj bardzo małe wartości, np. rzędu  $1e-5$  (jest to bardziej zwięzła notacja oznaczająca wartość  $0,00001$ ). Możemy sobie wyobrazić taką aproksymację metodą różnic skończonych jako nachylenie siecznej łączącej dwa punkty funkcji kosztu: wagi  $w$  oraz  $w + \varepsilon$  (obydwa punkty przyjmują wartości skalarne), co zostało zilustrowane na rysunku 12.13. Dla uproszczenia pomijamy na rysunku indeksy górne i dolne.



Rysunek 12.13. Metoda sprawdzania gradientów na wykresie funkcji kosztu

Jeszcze lepszym rozwiązańm pozwalającym na dokładniejszą aproksymację gradientu jest wyliczenie symetrycznego (lub centralnego) ilorazu różnicowego wyrażonego poniższym wzorem:

$$\frac{J(w_{i,j}^{(l)} + \varepsilon) - J(w_{i,j}^{(l)} - \varepsilon)}{2\varepsilon}$$

Zazwyczaj przybliżona różnica pomiędzy wyliczonym numerycznie gradientem  $J'_n$  a jego analitycznym odpowiednikiem  $J'_a$  jest wyliczana jako wektor znormalizowany metodą L2. Z praktycznych względów rozmiieszczamy wartości wyliczonych macierzy gradientu w jednowymiarowych wektorach, dzięki czemu łatwiej nam jest wyliczyć błąd (różnicę pomiędzy wektorami gradientów):

$$blqd = \|J'_n - J'_a\|_2$$

Problem polega na tym, że błąd jest zależny od skali (jeżeli wektory wag są małe, to małe błędy zyskują większe znaczenie). Dlatego zalecane jest wyliczanie znormalizowanej różnicy:

$$względny\ blqd = \frac{\|J'_n - J'_a\|_2}{\|J'_n\|_2 + \|J'_a\|_2}$$

Chcemy, aby wartość względnego błędu pomiędzy gradientem numerycznym a analitycznym była jak najmniejsza. Przed wprowadzeniem implementacji sprawdzania gradientów musimy jeszcze zastanowić się nad jednym szczegółem: jaka jest dopuszczalna wartość progowa błędu pozwalająca na pozytywne rozpatrzenie sprawdzania gradientów? Zależy ona od złożoności architektury sieci neuronowej. Zgodnie z niepisana zasadą: im więcej dodajemy ukrytych warstw, tym większa różnica pojawi się pomiędzy gradientem numerycznym a analitycznym przy właściwej implementacji algorytmu wstępnej propagacji. W tym rozdziale zaimplementowaliśmy względnie nieskomplikowaną architekturę sieci neuronowej, dlatego będziemy dość surowi względem wartości granicznej i zdefiniujemy następujące zasady:

- wartość względnego błędu  $\leq 1e-7$  oznacza, że model działa jak należy;
- wartość względnego błędu  $\leq 1e-4$  oznacza, że model nie jest całkowicie prawidłowy i powinniśmy go sprawdzić;
- wartość względnego błędu  $> 1e-4$  oznacza, że coś jest prawdopodobnie nie tak z naszym kodem.

Po ustaleniu powyższych reguł możemy przystąpić do implementacji sprawdzania gradientów. W tym celu przerobimy utworzoną wcześniej klasę NeuralNetMLP i wstawimy do niej poniższą metodę:

```
def _gradient_checking(self, X, y_enc, w1,
                      w2, epsilon, grad1, grad2):
    """ Sprawdzanie gradientu (jedynie w celu wyszukiwania błędów).

    Zwraca
    ------
    relative_error : wartość zmiennoprzecinkowa
        Względny błąd pomiędzy numerycznymi aproksymacjami
        gradientów a wartościami gradientów otrzymanymi za pomocą
        algorytmu wstępnej propagacji.

    """
    num_grad1 = np.zeros(np.shape(w1))
    epsilon_ary1 = np.zeros(np.shape(w1))
    for i in range(w1.shape[0]):
        for j in range(w1.shape[1]):
            epsilon_ary1[i, j] = epsilon
            a1, z2, a2, z3, a3 = self._feedforward(
                X,
                w1 - epsilon_ary1,
                w2)
            cost1 = self._get_cost(y_enc,
                                  a3,
                                  w1-epsilon_ary1,
                                  w2)
            a1, z2, a2, z3, a3 = self._feedforward(
                X,
                w1 + epsilon_ary1,
                w2)
```

```

cost2 = self._get_cost(y_enc,
                      a3,
                      w1 + epsilon_ary1,
                      w2)
num_grad1[i, j] = (cost2 - cost1) / (2 * epsilon)
epsilon_ary1[i, j] = 0

num_grad2 = np.zeros(np.shape(w2))
epsilon_ary2 = np.zeros(np.shape(w2))
for i in range(w2.shape[0]):
    for j in range(w2.shape[1]):
        epsilon_ary2[i, j] = epsilon
        a1, z2, a2, z3, a3 = self._feedforward(
            X,
            w1,
            w2 - epsilon_ary2)
        cost1 = self._get_cost(y_enc,
                               a3,
                               w1,
                               w2 - epsilon_ary2)
        a1, z2, a2, z3, a3 = self._feedforward(
            X,
            w1,
            w2 + epsilon_ary2)
        cost2 = self._get_cost(y_enc,
                               a3,
                               w1,
                               w2 + epsilon_ary2)
        num_grad2[i, j] = (cost2 - cost1) / (2 * epsilon)
        epsilon_ary2[i, j] = 0

num_grad = np.hstack((num_grad1.flatten(),
                      num_grad2.flatten()))
grad = np.hstack((grad1.flatten(), grad2.flatten()))
norm1 = np.linalg.norm(num_grad - grad)
norm2 = np.linalg.norm(num_grad)
norm3 = np.linalg.norm(grad)
relative_error = norm1 / (norm2 + norm3)
return relative_error

```

Kod metody `_gradient_checking` nie wygląda na zbyt skomplikowany — i sam radzę, żeby utrzymywać go w jak najprostszej postaci. Naszym celem jest określenie dokładności wyliczeń gradientów, dlatego musimy się upewnić, że nie wprowadzimy do algorytmu sprawdzania gradientów żadnych dodatkowych błędów wynikających z utworzenia skutecznego, ale złożonego kodu. Musimy teraz wprowadzić niewielką modyfikację w metodzie `fit`. Dla zachowania przejrzystości pominąłem w poniższym fragmencie początek funkcji `fit`, a jedynie wiersze, które powinniśmy do niej wstawić, umieściłem pomiędzy komentarzami `## początek sprawdzania gradientów` i `## koniec sprawdzania gradientów`:

```

class MLPGradientCheck(object):
    [...]
    def fit(self, X, y, print_progress=False):
        [...]

            # oblicza gradient za pomocą algorytmu wstępnej propagacji
            grad1, grad2 = self._get_gradient(
                a1=a1,
                a2=a2,
                a3=a3,
                z2=z2,
                y_enc=y_enc[:, idx],
                w1=self.w1,
                w2=self.w2)

    ## początek sprawdzania gradientów
    grad_diff = self._gradient_checking(
        X=X[idx],
        y_enc=y_enc[:, idx],
        w1=self.w1,
        w2=self.w2,
        epsilon=1e-5,
        grad1=grad1,
        grad2=grad2)

    if grad_diff <= 1e-7:
        print('OK: %s' % grad_diff)
    elif grad_diff <= 1e-4:
        print('Ostrzeżenie: %s' % grad_diff)
    else:
        print('PROBLEM: %s' % grad_diff)

    ## koniec sprawdzania gradientów

    # aktualizuje wagi; [alpha * delta_w_prev]
    # dla uczenia momentowego
    delta_w1 = self.eta * grad1
    delta_w2 = self.eta * grad2
    self.w1 -= (delta_w1 +
                (self.alpha * delta_w1_prev))
    self.w2 -= (delta_w2 +
                (self.alpha * delta_w2_prev))
    delta_w1_prev = delta_w1
    delta_w2_prev = delta_w2

return self

```

Zakładając, że nazwaliśmy zmodyfikowaną klasę wielowarstwowego perceptronu `MLPGradientCheck`, możemy teraz zainicjować model MLP zawierający 10 ukrytych warstw. Poza tym wyłączamy regularyzację, uczenie adaptacyjne oraz uczenie momentowe. Do tego wprowadzamy

standardowy algorytm gradientu prostego, zmieniając wartość parametru `minibatches` na 1. Posłuży nam do tego poniższy kod:

```
>>> nn_check = MLPGradientCheck(n_output=10,
                                 n_features=X_train.shape[1],
                                 n_hidden=10,
                                 l2=0.0,
                                 l1=0.0,
                                 epochs=10,
                                 eta=0.001,
                                 alpha=0.0,
                                 decrease_const=0.0,
                                 minibatches=1,
                                 random_state=1)
```

Jedną z wad sprawdzania gradientów jest jego olbrzymia kosztowność obliczeniowa. Uczenie sieci neuronowej przy użyciu algorytmu sprawdzania gradientów jest tak powolne, że w rzeczywistości chcemy z niego korzystać jedynie w celu wyszukiwania błędów. Z tego powodu dość powszechną praktyką jest stosowanie tego modelu wyłącznie wobec kilku próbek uczących (w naszym przykładzie pięciu). Sprawdźmy wyniki:

```
>>> nn_check.fit(X_train[:5], y_train[:5], print_progress=False)
OK: 2.56712936241e-10
OK: 2.94603251069e-10
OK: 2.37615620231e-10
OK: 2.43469423226e-10
OK: 3.37872073158e-10
OK: 3.63466384861e-10
OK: 2.22472120785e-10
OK: 2.33163708438e-10
OK: 3.44653686551e-10
OK: 2.17161707211e-10
```

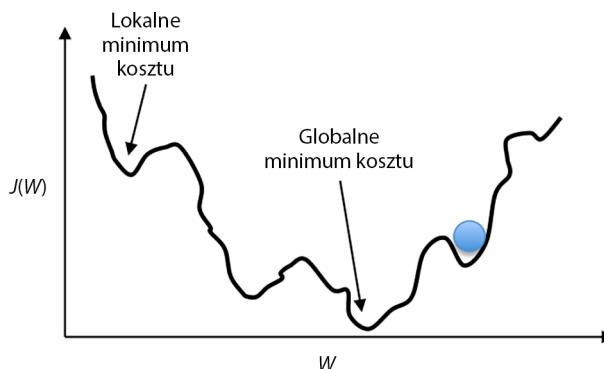
Jak widać, nasz wielowarstwowy perceptron uzyskuje w teście znakomite wyniki.

## Zbieżność w sieciach neuronowych

Być może zastanawiasz się, dlaczego podczas trenowania sieci neuronowej w klasyfikowaniu odreźnie zapisywanych znaków zastosowaliśmy nie klasyczny algorytm gradientu prostego, lecz uczenie za pomocą podzbiorów danych uczących (ang. *minibatch learning*). Przypomnijmy sobie dyskusję na temat stochastycznego spadku wzduż gradientu używanego podczas uczenia przyrostowego (ang. *online learning*). W trakcie uczenia przyrostowego każdorazowo obliczamy gradient na podstawie pojedynczego przebiegu uczenia ( $k = 1$ ) w celu aktualizowania wag. Pomimo że jest to rozwiązaństochastyczne, często otrzymujemy za jego pomocą bardzo dokładne wyniki, a jednocześnie uzyskujemy zbieżności znacznie szybciej niż przy standardowym algorytmie gradientu prostego. Uczenie za pomocą podzbiorów próbek stanowi specjalny

przypadek stochastycznego spadku wzduż gradientu, w którym wyliczamy gradient na podstawie podzbioru  $k$  stanowiącego część zestawu  $n$  próbek uczących ( $1 < k < n$ ). Uczenie za pomocą podzbiorów ma tę przewagę nad uczeniem przyrostowym, że możemy w nim wykorzystać zwięktryzowane implementacje do poprawy skuteczności obliczeniowej. Można jednak aktualizować wagi znacznie szybciej niż przy użyciu algorytmu gradientu prostego. Uczenie za pomocą podzbiorów danych możemy porównać do przewidywania zwycięzcy wyborów prezydenckich na podstawie ankiety przeprowadzonej na reprezentatywnej grupie społeczeństwa.

Ponadto wprowadziliśmy dodatkowe parametry strojenia, takie jak stała redukcji czy współczynnik uczenia adaptacyjnego. Powodem jest to, że sieci neuronowe są znacznie trudniejsze do wyuczenia niż proste algorytmy, takie jak Adaline, regresja logistyczna czy maszyny wektorów nośnych. W wielowarstwowych sieciach neuronowych mamy zazwyczaj do czynienia z setkami, tysiącami, a nawet milionami wag, które należy zoptymalizować. Niestety, wynikowa funkcja kosztu ma nierówny przebieg i algorytm optymalizacyjny może bardzo łatwo ugręźnąć w lokalnych minimach, co zostało przedstawione na rysunku 12.14.



Rysunek 12.14. Algorytm (niebieskie kółko) zablokowany w lokalnym minimum funkcji kosztu

Zwrócić uwagę, że taki opis jest maksymalnie uproszczony, ponieważ sieci neuronowe są wielowymiarowe; w rzeczywistości zwizualizowanie rzeczywistej powierzchni kosztu okazuje się niemożliwe. Na rysunku 12.14 pokazuję jedynie wykres funkcji kosztu dla pojedynczej wagi w osi  $x$ . Chcę jednak w ten sposób wyjaśnić, że nie możemy dopuścić do utknięcia algorytmu w lokalnych minimach. Jeśli zwiększymy wartość współczynnika uczenia, łatwiej nam będzie ich unikać. Z drugiej strony, wybierając zbyt dużą wartość współczynnika uczenia, ryzykujemy pominięcie globalnego minimum. Na początku inicjujemy losowe wartości wag, dlatego w owym momencie rozwiązanie problemu optymalizacyjnego jest zazwyczaj beznadziejnie niewłaściwe. Zdefiniowana wcześniej stała redukcji pomaga nam w początkowej fazie szybko „schodzić” po powierzchni funkcji kosztu, a dzięki współczynnikiowi uczenia adaptacyjnego łatwiej algorytmowi trafić na minimum globalne.

# Inne architektury sieci neuronowych

W tym rozdziale opisałem jedną z najpopularniejszych reprezentacji sieci neuronowej wykorzystującej sprzęt w przód — wielowarstwowy perceptron. Sieci neuronowe obecnie stanowią jedną z najprężniej rozwijających się dziedzin uczenia maszynowego i powstało wiele innych architektur, których opis wykracza poza zakres niniejszej książki. Osoby poszukujące wiedzy na temat sieci neuronowych i algorytmów uczenia głębokiego powinny zapoznać się z pozycją: Y. Bengio, *Learning Deep Architectures for AI, „Foundations and Trends in Machine Learning”* 2009, nr 2 (1), s. 1 – 127. Publikacja ta jest obecnie dostępna bezpłatnie pod adresem [http://www.iro.umontreal.ca/~bengioy/papers/fml\\_book.pdf](http://www.iro.umontreal.ca/~bengioy/papers/fml_book.pdf).

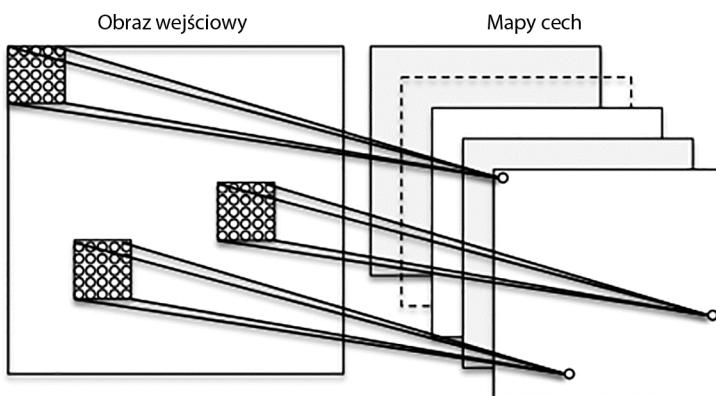
Sieci neuronowe stanowią temat na osobną książkę, ale przyjrzyjmy się побieżnie jeszcze dwóm rodzajom architektur: **splotowym sieciom neuronowym** oraz **rekurencyjnym sieciom neuronowym**.

## Splotowe sieci neuronowe

Splotowe (konwolucyjne) sieci neuronowe (ang. *convolutional neural networks* — CNN) uzyskały popularność w zastosowaniach graficznych z powodu dobrej skuteczności w klasyfikowaniu obrazów. Obecnie sieci CNN stanowią jedną z najpopularniejszych architektur stosowanych w uczeniu głębokim. Kluczową cechą w splotowych sieciach neuronowych jest tworzenie wielu warstw **detektorów cech** (ang. *feature detectors*), analizujących rozmieszczenie przestrzenne pikseli w wejściowym obrazie. Zauważmy, że istnieje wiele różnych odmian sieci CNN. My zajmiemy się jedynie opisem ogólnego schematu działania tej architektury. Jeżeli chcesz dowiedzieć się więcej na ten temat, polecam zapoznanie się z dorobkiem Yanna LeCuna (<http://yann.lecun.com/>) — jednego z twórców architektury CNN. W szczególności dobrym wprowadzeniem do splotowych sieci neuronowych są następujące pozycje:

- Y. LeCun, L. Bottou, Y. Bengio i P. Haffner, *Gradient-based Learning Applied to Document Recognition*, „Proceeding of the IEEE” 1998, nr 86 (11), s. 2278 – 2324.
- P.Y. Simard, D. Steinkraus i J.C. Platt, *Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis*, IEEE 2003, s. 958.

Jak pamiętamy z implementacji wielowarstwowego perceptronu, rozkładaliśmy obrazy na wektory cech, które w pełni łączłyśmy z warstwą ukrytą — informacje przestrzenne nie były kodowane w tej architekturze sieci neuronowej. W architekturze CNN wykorzystujemy **pola recepcyjne** (ang. *receptive field*) do łączenia warstwy wejściowej z mapą cech. Pola recepcyjne możemy sobie wyobrazić jako nakładające się na siebie okienka, które przesuwamy po pikselach obrazu wejściowego w celu utworzenia mapy cech. Stopień przesunięcia okna oraz jego rozmiar stanowią dodatkowe hiperparametry, które musimy odgórnie zdefiniować. Proces tworzenia **mapy cech** (ang. *feature map*) jest nazywany **splataniem (konwolucją)**. Przykład takiej **warstwy splotowej** (ang. *convolutional layer*), łączącej wejściowe piksele z każdą jednostką mapy cech, został zaprezentowany na rysunku 12.15.



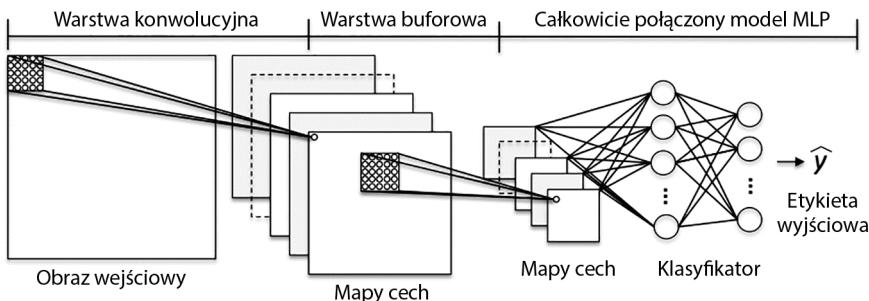
Rysunek 12.15. Splatanie pikseli w konwolucyjnej sieci neuronowej

Należy zauważyć, że detektory cech są kopiami, co oznacza, że pola recepcyjne rzutujące cechy do jednostek tworzących kolejną warstwę mają identyczne wagi. Wynika to z założenia, że jeżeli detektor cech przydaje się w jednej części obrazu, to może też być użyteczny w innym jego obszarze. Przydatnym skutkiem ubocznym tego rozwiązania jest znaczne ograniczenie liczby parametrów, które należy zoptymalizować. Pozwalamy, żeby różne fragmenty obrazu były przedstawiane na różne sposoby, dlatego sieci CNN świetnie spisują się w rozpoznawaniu obiektów o różnych rozmiarach i lokalizacji na zdjęciu. W przeciwnieństwie do przykładu z zestawem danych MNIST nie musimy się tu przejmować koniecznością skalowania czy wyśrodkowania analizowanych obrazów.

W architekturze CNN po warstwie konwolucyjnej występuje **warstwa buforowa** (ang. *pooling layer*), zwana też czasami **warstwą podpróbkowania** (ang. *sub-sampling layer*). W warstwie buforowej łączymy sąsiadujące detektory cech w celu zmniejszenia liczby cech przekazywanych do następnej warstwy. W tym przypadku buforowanie możemy sobie wyobrazić jako prostą metodę odkrywania cech, w której bierzemy uśrednioną lub maksymalną wartość fragmentu obrazu reprezentowanego przez sąsiadujące cechy, a następnie przesyłamy ją do kolejnej warstwy. W celu stworzenia głębszej, konwolucyjnej sieci neuronowej tworzymy wiele warstw — naprzemiennie konwolucyjnych i buforowych — po czym łączymy ją z wielowarstwowym perceptronem w celu przeprowadzenia klasyfikacji. Zostało to ukazane na rysunku 12.16.

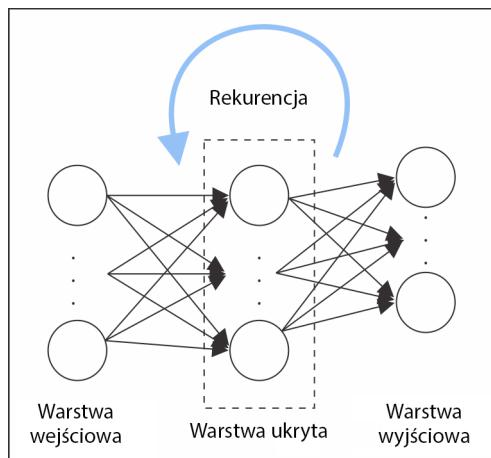
## Rekurencyjne sieci neuronowe

**Rekurencyjne sieci neuronowe** (ang. *recurrent neural networks* — **RNN**) możemy uznać za jednokierunkowe sieci neuronowe (propagacja w przód), w których zaimplementowano również pętle sprzężenia zwrotnego (wsteczną propagację w czasie). W sieciach RNN neurony są pobudzane wyłącznie przez określony czas, po czym następuje ich (tymczasowa) dezaktywacja. Z kolei przekazują one sygnał do kolejnych neuronów, które zostają pobudzone w późniejszym



Rysunek 12.16. Schemat działania splotowej sieci neuronowej

czasie. Mówiąc w skrócie, rekurencyjne sieci neuronowe przypominają model MLP z dodatkową zmienią funkcji czasu (rysunek 12.17). Dzięki obecności tej zmiennej czasowej oraz dynamicznej strukturze w sieci tej wykorzystywane są nie tylko bieżące sygnały wejściowe, lecz również dane wprowadzone wcześniej.



Rysunek 12.17. Schemat działania rekurencyjnej sieci neuronowej

Sieci RNN osiągają bardzo dobre rezultaty w rozpoznawaniu mowy oraz odręcznego pisma, a także tłumaczeniu języków, ale bardzo trudno jest je wytrenować. Wynika to z faktu, że nie ma prostego sposobu wstępnej propagacji błędów warstwa po warstwie; musimy dodatkowo uwzględniać dodatkowy czynnik czasowy, który zwielokrotnia problem zanikania i potęgowania gradientu. W 1997 roku Juergen Schmidhuber wraz ze współpracownikami opracował obejście wspomnianego problemu: tzw. **jednostki długiej pamięci krótkotrwałej** (ang. *Long Short Term Memory units — LSTM*); S. Hochreiter i J. Schmidhuber, *Long Short-term Memory*, „Neural Computation” 1997, nr 9 (8), s. 1735 – 1780.

Pragnę jednak zauważyć, że istnieje mnóstwo różnych odmian sieci RNN, których dokładny opis wykracza poza ramy niniejszej książki.

## Jeszcze słowo o implementacji sieci neuronowej

Zastanawiasz się pewnie, po co tak bardzo skoncentrowałem się na teorii, gdy ostatecznie zaimplementowaliśmy jedynie prostą, wielowarstwową sieć neuronową służącą do klasyfikowania odręcznie pisanych cyfr, zamiast skorzystać w Pythonie z jawnej biblioteki uczenia maszynowego. Jedną z przyczyn jest to, że w trakcie pisania tej książki nie istniała jeszcze implementacja modelu MLP w interfejsie scikit-learn. Co ważniejsze, my (adepci sztuki uczenia maszynowego) powinniśmy rozumieć chociaż w podstawowym zakresie stosowane przez nas algorytmy, aby móc je wykorzystywać prawidłowo i skutecznie.

Skoro już wiemy, jak działają jednokierunkowe sieci neuronowe, jesteśmy gotowi zapoznać się z bardziej zaawansowanymi bibliotekami Pythona bazującymi na interfejsie NumPy, takimi jak Theano (<http://deeplearning.net/software/theano/>), umożliwiająca skuteczniejsze projektowanie sieci neuronowych. Została ona omówiona w rozdziale 13., „Równoległe przetwarzanie sieci neuronowych za pomocą biblioteki Theano”. W ciągu kilku minionych lat interfejs Theano zdobył dużą popularność wśród analityków uczenia maszynowego, którzy używają go do tworzenia głębokich sieci neuronowych z powodu możliwości optymalizowania wyrażeń matematycznych w obliczeniach przeprowadzanych na wielowymiarowych tablicach przy użyciu procesorów graficznych (ang. *Graphical Processing Units — GPU*).

Na stronie <http://deeplearning.net/software/theano/tutorial/index.html#tutorial> znajdziesz duży zbiór poradników i instrukcji używania biblioteki Theano.

Istnieje również wiele interesujących, aktywnie rozwijanych bibliotek służących do uczenia sieci neuronowych w Theano, z którymi warto się zapoznać:

- *Pylearn2* (<http://deeplearning.net/software/pylearn2/>),
- *Lasagne* (<https://lasagne.readthedocs.io/en/latest/>),
- *Keras* (<https://keras.io/>).

## Podsumowanie

W niniejszym rozdziale poznaliśmy najważniejsze pojęcia z zakresu wielowarstwowych, sztucznych sieci neuronowych, stanowiących obecnie najpopularniejszy dział uczenia maszynowego. W rozdziale 2., „Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji”, rozpoczęliśmy naszą podróż od prostej, jednowarstwowej architektury sieci neuronowej, którą w tym rozdziale połączylismy w wieloneuronową strukturę pozwalającą na rozwiązywanie złożonych problemów, takich jak rozpoznawanie odręcznie zapisywanych cyfr. Odkryliśmy tajemnice algorytmu wstępnej propagacji — jednego z elementów budulcowych wielu modeli głębokich

sieci neuronowych. Gdy już poznaliśmy ten algorytm, mogliśmy przejść do aktualizowania wag w tak złożonej sieci neuronowej. Dodaliśmy ponadto kilka modyfikacji, takich jak uczenie za pomocą podzbiorów danych uczących, a także współczynnik uczenia adaptacyjnego, które zwiększą skuteczność uczenia sieci neuronowych.

# Równoległe przetwarzanie sieci neuronowych za pomocą biblioteki Theano

W poprzednim rozdziale skupiliśmy się na teoretycznym opisie działania jednokierunkowych sztucznych sieci neuronowych i wielowarstwowych perceptronów. Właściwe zrozumienie aparatu matematycznego stanowiącego trzon algorytmów uczenia maszynowego jest bardzo istotną kwestią, gdyż pozwala na skuteczne i, przede wszystkim, **poprawne** stosowanie tych potężnych technik. W trakcie czytania niniejszej książki poświęciliśmy mnóstwo czasu na poznawanie najlepszych rozwiązań stosowanych w dziedzinie uczenia maszynowego i nawet zaimplementowaliśmy samodzielnie od podstaw kilka algorytmów. W tym rozdziale możesz sobie pozwolić na chwilę odpoczynku, gdyż pragnę Cię zabrać w ekscytującą podróż ukazującą jedną z najpotężniejszych bibliotek wykorzystywanych przez adeptów uczenia maszynowego do niezwykle skutecznego trenowania głębokich sieci neuronowych oraz eksperymentowania z ich różnymi konfiguracjami. Obecnie w większości wypadków w tym celu są wykorzystywane komputery zawierające bardzo wydajne **procesory graficzne** (ang. *Graphics Processing Units — GPU*). Jeżeli interesuje Cię kwestia uczenia głębokiego — obecnie najpopularniejszego zagadnienia

z dziedziny uczenia maszynowego — ten rozdział został napisany specjalnie dla Ciebie. Nie martw się jednak, jeśli nie masz dostępu do procesorów graficznych; w tym rozdziale ich stosowanie nie jest koniecznością.

Zanim przejdziemy do głównego dania, zobaczymy, czym będziemy się zajmować w niniejszym rozdziale:

- pisanie zoptymalizowanego kodu uczenia maszynowego w bibliotece Theano,
- wybór odpowiedniej funkcji aktywacji w sztucznej sieci neuronowej,
- szybkie i łatwe eksperymentowanie przy użyciu biblioteki Keras.

## Tworzenie, komplikowanie i uruchamianie wyrażeń w interfejsie Theano

W tym podrozdziale przyjrzymy się potężnemu narzędziu Theano, zaprojektowanemu w celu jak najskuteczniejszego trenowania modeli uczenia maszynowego napisanych w Pythonie. Początki tej biblioteki sięgają 2008 roku, powstała w laboratorium **LISA** (fr. *Laboratoire d'Informatique des Systèmes Adaptatifs*; obecnie **MILA** — ang. *Montreal Institute for Learning Algorithms*; <https://mila.umontreal.ca/en/>), jej autorem jest Yoshua Bengio.

Zanim przejdziemy do omówienia istoty biblioteki Theano oraz tego, w jaki sposób pomaga ona w przyśpieszaniu zadań uczenia maszynowego, przeanalizujmy wyzwania, którym musimy stawić czoła podczas uruchamiania bardzo złożonych obliczeń na komputerze. Szczęśliwie dla nas wydajność procesorów jest cały czas rozwijana, co pozwala nam na trenowanie coraz bardziej skomplikowanych i potężnych systemów uczenia maszynowego, dzięki czemu uzyskujemy coraz większą skuteczność predykcyjną tworzonych przez nas modeli. Nawet najtańszy komputer osobisty zawiera obecnie kilka rdzeni procesora. W poprzednich rozdziałach mogliśmy się przekonać, że wiele funkcji interfejsu scikit-learn pozwala nam na rozdzielenie czynności obliczeniowych na wiele rdzeni procesora jednocześnie. Niestety, domyślnie działanie algorytmów napisanych w Pythonie jest ograniczone do jednego rdzenia z powodu **globalnej blokady interpretera** (ang. *Global Interpreter Lock* — **GIL**). Choćbyśmy wzięli pod uwagę możliwość korzystania z biblioteki `multiprocessing` służącej do wielowątkowości, pamiętajmy, że nawet zaawansowane komputery osobiste rzadko kiedy mają więcej niż 8 – 16 rdzeni.

Wróćmy na chwilę do omówionej w poprzednim rozdziale bardzo prostej implementacji perceptronu wielowarstwowego zawierającego tylko jedną warstwę ukrytą, złożoną z 50 jednostek. Już w tym modelu musielibyśmy zoptymalizować mniej więcej 1000 wag w celu klasyfikowania nieskomplikowanych obrazów. Rysunki znajdujące się w bazie danych MNIST nie są zbyt duże ( $28 \times 28$  pikseli) i możemy sobie jedynie wyobrazić gwałtowny wzrost liczby parametrów po dodaniu kolejnych warstw lub pracę z obrazami o większej gęstości pikseli. Model taki szybko przerósłby możliwości pojedynczego rdzenia procesora. Rodzi się w tym momencie pytanie, w jaki sposób możemy sobie skutecznie poradzić z tym problemem. Oczywistą odpowiedzią

w tym przypadku są procesory graficzne — prawdziwe bolidy wśród jednostek obliczeniowych. Możemy wyobrazić sobie kartę graficzną jako komputerek wewnątrz komputera. Wielką zaletą kart graficznych jest również ich względnie niska cena w stosunku do najnowocześniejszych procesorów głównych, co zostało zaprezentowane w tabeli 13.1.

**Tabela 13.1.** Porównanie parametrów współczesnego procesora głównego (CPU) i graficznego (GPU); dane aktualne na 20 sierpnia 2015 roku

Parametry	Procesor Intel® Core™ i7-5960X Extreme Edition	NVIDIA GeForce® GTX™ 980 Ti
Podstawowa częstotliwość taktowania	3,0 GHz	1,0 GHz
Liczba rdzeni	8	2816
Przepustowość pamięci	68 GB/s	336,5 GB/s
Liczba operacji zmiennoprzecinkowych	354 gigaflop	5632 gigaflop
Cena	1000 \$	700 \$

Źródła danych:

<http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-980-ti/specifications>,  
[http://ark.intel.com/products/82930/Intel-Core-i7-5960X-Processor-Extreme-Edition-20M-Cache-up-to-3\\_50-GHz](http://ark.intel.com/products/82930/Intel-Core-i7-5960X-Processor-Extreme-Edition-20M-Cache-up-to-3_50-GHz).

Za 70% ceny nowoczesnego procesora głównego otrzymujemy kartę graficzną mającą 450 razy więcej rdzeni i wykonującą około piętnastokrotnie więcej operacji zmiennoprzecinkowych na sekundę. Dlaczego więc mielibyśmy nie wykorzystywać procesorów graficznych w uczeniu maszynowym? Problem polega na tym, że pisanie kodu działającego na procesorze graficznym nie jest takie proste, jak uruchomienie procedur Pythona w interpreterze. Istnieją pewne pakiety, takie jak CUDA czy OpenCL, umożliwiające korzystanie z procesora graficznego, ale tworzenie w nich algorytmów uczenia maszynowego nie jest zbyt wygodnym rozwiązaniem. Dobra informacja jest taka, że w celu ułatwienia nam życia została stworzona biblioteka Theano!

## Czym jest Theano?

Czym właściwie jest Theano — językiem programowania, kompilatorem czy biblioteką Pythona? Okazuje się, że po trochu pełni każdą z wymienionych funkcji. Interfejs Theano został zaprojektowany z myślą o bardzo skutecznym implementowaniu, komplikowaniu i ocenianiu wyrażeń matematycznych ze szczególnym naciskiem na wielowymiarowe tablice (tensory). Istnieje w nim możliwość uruchamiania kodu na zwykłych procesorach, jednak maksymalną wydajność osiąga, wykorzystując olbrzymią przepustowość pamięci oraz liczbę obliczeń zmiennoprzecinkowych na sekundę, zapewniane przez karty graficzne. Za pomocą Theano możemy również z łatwością uruchomić wielowątkowe przetwarzanie kodu przy użyciu współdzielonej pamięci. W 2010 roku twórcy interfejsu Theano stwierdzili, że kod uruchomiony na zwykłym procesorze był wykonywany 1,8-krotnie szybciej niż przetwarzany w bibliotece NumPy, a przy użyciu karty graficznej różnica szybkości pomiędzy bibliotekami Theano a NumPy była aż jedenastokrotna (J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley i Y. Bengio, *Theano: A CPU and GPU Math Compiler in Python*, Proc. 9th Python in

Science Conf 2010, s. 1 – 7). Pamiętaj, że te informacje pochodzą z 2010 roku i przez ten czas biblioteka Theano była stale rozwijana, podobnie jak możliwości kart graficznych.

Jaki związek więc istnieje pomiędzy bibliotekami Theano i NumPy? Silnik Theano bazuje na bibliotece NumPy i obydwa interfejsy mają bardzo zbliżoną składnię, przez co używanie tego pierwszego nie stanowi problemu dla osób zaznajomionych z działaniem drugiego. Gwoli ściśleści, interfejs Theano nie jest jedynie „biblioteką NumPy na sterydach”, jak określają go niektórzy ludzie, lecz ma również wiele podobieństw z pakietem SymPy (<http://www.sympy.org/en/index.html>) służącym do przeprowadzania obliczeń symbolicznych (algebry symbolicznej). Wiemy już z poprzednich rozdziałów, że w bibliotece NumPy opisujemy zmienne oraz ich wzajemne powiązania; napisany kod jest wykonywany sekwencyjnie, wiersz po wierszu. Z kolei w Theano definiujemy najpierw problem oraz sposób jego analizowania. Następnie kod zostaje zoptymalizowany i skompilowany pod kątem środowiska C/C++ lub, jeżeli chcemy korzystać z dobodziejstw procesora graficznego, CUDA/OpenCL. W celu wygenerowania zoptymalizowanego kodu biblioteka Theano musi znać zakres problemu; wyobraź to sobie jako drzewo operacji (lub graf wyrażeń symbolicznych). Zwróć uwagę, że pakiet ten jest wciąż aktywnie rozwijany, dlatego w kolejnych wersjach dodawane są nowe funkcje oraz regularnie wprowadzane poprawki i usprawnienia. W tym rozdziale poznamy podstawy obsługi biblioteki Theano i dowiemy się, jak ją wykorzystywać w procesie uczenia maszynowego. Cechują ją spore rozmiary oraz mnogość zaawansowanych funkcji, dlatego omówienie całości wykracza poza zakres niniejszej książki. Jeżeli jednak chcesz się dowiedzieć więcej na temat omawianego interfejsu, polecam znakomitą dokumentację dostępną pod adresem <http://deeplearning.net/software/theano/>.

## Pierwsze kroki z Theano

Poznamy teraz podstawy obsługi biblioteki Theano. W zależności od konfiguracji systemu zazwyczaj wystarczy użyć instalatora pip i pobrać bibliotekę Theano z repozytorium PyPI przy użyciu następującej komendy wpisanej w wierszu polecenia:

```
pip install Theano
```

Jeżeli podczas instalacji pakietu natrafisz na problemy, polecam zapoznanie się z wymaganiami platformy i systemu operacyjnego dostępnymi na stronie <http://deeplearning.net/software/theano/install.html>. Zauważ, że wszystkie fragmenty kodu zamieszczone w tym rozdziale możesz uruchamiać na zwykłym procesorze; korzystanie z procesora graficznego nie jest obowiązkowe, ale warto go używać, aby wykorzystać w pełni potencjał biblioteki Theano. Jeżeli masz do dyspozycji kartę graficzną obsługującą architektury CUDA lub OpenCL, przeczytaj informacje zawarte pod adresem [http://deeplearning.net/software/theano/tutorial/using\\_gpu.html#using-gpu](http://deeplearning.net/software/theano/tutorial/using_gpu.html#using-gpu) w celu właściwej konfiguracji środowiska pracy.

Mechanizm działania interfejsu Theano bazuje na tzw. tensorach pozwalających na ewaluację symbolicznych wyrażeń matematycznych. Tensory możemy zdefiniować jako uogólnienie wielkości skalarnych, wektorów, macierzy itd. Ścisłe mówiąc, wielkość skalarna jest uznawana za tensor zerowego rzędu, wektor jako tensor pierwszego rzędu, macierz — drugiego rzędu, a macierze występujące w trójwymiarowej przestrzeni — za tensorы trzeciego rzędu. W ramach rozgrzewki

uzejmamy prostych wielkości skalarnych umieszczonych w module tensor do obliczenia całkowitego pobudzenia  $z$  przykładowej próbki  $x$  w jednowymiarowym zbiorze danych przy danej wadze  $w_1$  i obciążeniu  $w_0$ :

$$z = x_1 \times w_1 + w_0$$

Wspomniany kod wygląda następująco:

```
>>> import theano
>>> from theano import tensor as T

# inicjuje
>>> x1 = T.scalar()
>>> w1 = T.scalar()
>>> w0 = T.scalar()
>>> z1 = w1 * x1 + w0

# kompiluje
>>> net_input = theano.function(inputs=[w1, x1, w0],
...                                outputs=z1)

# uruchamia
>>> print('Całkowite pobudzenie: %.2f' % net_input(2.0, 1.0, 0.5))
Całkowite pobudzenie: 2.50
```

Nie było tak strasznie, prawda? Podczas pisania kodu w interfejsie Theano musimy pamiętać o trzech prostych czynnościach: zdefiniowaniu **symboli** (obiektów Variable), skompilowaniu kodu i jego uruchomieniu. Na etapie inicjacji zdefiniowaliśmy trzy symbole:  $x_1$ ,  $w_1$  i  $w_0$ , służące do wyliczenia wartości  $z_1$ . Następnie skompilowaliśmy funkcję `net_input` wyliczającą całkowite pobudzenie  $z_1$ .

Pisząc kod, musimy zwrócić szczególną uwagę na pewną rzecz: typ zmiennych (`dtype`). Dla jednych może być to błogosławieństwo, a dla innych kłata, ale w bibliotece Theano musimy definiować typ używanych zmiennych jako 32-/64-bitowe liczby całkowite lub zmiennoprzecinowe, co w olbrzymim stopniu wpływa na skuteczność kodu. Przyjrzyjmy się dokładniej tym typom zmiennych.

## Konfigurowanie środowiska Theano

Obecnie nie ma znaczenia, czy korzystamy z systemu Mac OS X, Windows czy Linuksa — obsługujemy aplikacje i oprogramowanie głównie za pomocą 64-bitowej architektury pamięci. Jeżeli jednak pragniemy przyśpieszyć obliczanie wyrażeń matematycznych na procesorach graficznych, często musimy polegać na starszym, 32-bitowym adresowaniu pamięci. Jest to jak dotąd jedyna obsługiwana architektura obliczeniowa w bibliotece Theano. W dalszej części rozdziału wyjaśnię, w jaki sposób właściwie skonfigurować ten pakiet. Jeżeli interesują Cię bardziej zaawansowane możliwości konfiguracyjne, sprawdź oficjalną dokumentację Theano: <http://deeplearning.net/software/theano/library/config.html>.

Podeczas implementowania algorytmów uczenia maszynowego najczęściej pracujemy na liczbach zmiennoprzecinkowych (ang. *floating point*). Domyślnie zarówno biblioteka NumPy, jak i Theano wykorzystują wartości zmiennoprzecinkowe podwójnej precyzji (`float64`). W praktyce jednak bardzo przydaje się możliwość zmieniania typu danych pomiędzy `float64` (wykłpy procesor) a `float32` (procesor graficzny) podeczas prototypowania algorytmu (pod zwykły procesor) i jego uruchamiania na karcie graficznej. Aby np. poznać domyślną konfigurację wartości zmiennoprzecinkowych w Theano, możemy wpisać następujące polecenie w interpreterze Pythona:

```
>>> print(theano.config.floatX)
float64
```

Jeżeli nie zmodyfikowałeś żadnych ustawień Theano podeczas instalacji, powinien wyświetlić Ci się domyślnie typ danych `float64`. Możemy go jednak bezproblemowo zmienić w bieżącej sesji Pythona na `float32` za pomocą komendy:

```
>>> theano.config.floatX = 'float32'
```

Zwróć uwagę, że chociaż obecnie musimy wprowadzać typ danych `float32` podeczas używania karty graficznej, główne procesory obsługują zarówno formaty `float32`, jak i `float64`. Dlatego jeżeli chcemy zmienić domyślne ustawienia na poziomie globalnym, możemy zmodyfikować zmienną `THEANO_FLAGS` w terminalu wiersza polecenia (Bash):

```
export THEANO_FLAGS="floatX=float32"
```

Możemy ewentualnie zastosować te zmiany jedynie wobec określonego skryptu Pythona, co zostało ukazane poniżej:

```
set THEANO_FLAGS="floatX=float32" & python Twój_skrypt.py
```

Wiemy już, w jaki sposób zdefiniować domyślne typy danych zmiennoprzecinkowych, żeby zmaksymalizować korzyści płynące ze stosowania biblioteki Theano na procesorach graficznych. Zobaczmy teraz, w jaki sposób możemy wybierać rodzaj wykorzystywanego procesora. Poniższa komenda służy do wyświetlania aktualnie używanego podzespołu:

```
>>> print(theano.config.device)
cpu
```

Zalecam pozostawienie `cpu` jako domyślnej wartości, dzięki czemu prototypowanie i usuwanie błędów z kodu będzie łatwiejsze. Możesz np. uruchomić kod Theano na głównym procesorze w postaci skryptu z poziomu wiersza polecenia:

```
set THEANO_FLAGS="device=cpu" & set THEANO_FLAGS="floatX=float64" & python
➥Twój_skrypt.py
```

Natomiast jeżeli już zaimplementowaliśmy kod i chcemy jak najskuteczniej uruchomić go na procesorze graficznym, możemy użyć poniższej komendy bez konieczności wprowadzania modyfikacji w samym kodzie:

```
set THEANO_FLAGS="device=gpu" & set THEANO_FLAGS="floatX=float32" & python
➥Twój_skrypt.py
```

Wygodnym rozwiązaniem jest również utworzenie pliku `.theanorc` w głównym katalogu, co pozwoli na utrwalenie omawianych zmian. Przykładowo, jeżeli chcesz zawsze korzystać z typu danych `float32` i procesora graficznego, stwórz plik `.theanorc` zawierający odpowiednią konfigurację. Dokonasz tego za pomocą polecenia:

```
echo -e "\n[global]\nfloatX=float32\ndevice=gpu\n" >> ~/.theanorc
```

Jeżeli nie chcesz korzystać z terminalu systemu Mac OS X lub linuksowego, możesz stworzyć samodzielnie plik `.theanorc`, wpisując w dowolnym edytorze tekstu następującą zawartość:

```
[global]
floatX=float32
device=gpu
```

Skoro wiemy już, jak skonfigurować bibliotekę Theano przy uwzględnieniu dostępnych podzespołów komputera, możemy przejść do analizowania obsługi bardziej zaawansowanych struktur tablicowych.

## Praca ze strukturami tablicowymi

Z tego punktu dowiemy się, w jaki sposób używać struktur tablicowych w bibliotece Theano za pomocą modułu `tensor`. Po uruchomieniu poniższego kodu utworzymy macierz o wymiarach  $2 \times 3$  i wyliczymy sumę kolumn, używając zoptymalizowanych wyrażeń tensorowych:

```
>>> import numpy as np

# inicjuje
# jeżeli uruchamiasz Theano w trybie 64-bitowym,
# musisz zmienić fmatrix na dmatrix
>>> x = T.fmatrix(name='x')
>>> x_sum = T.sum(x, axis=0)

# kompliluje
>>> calc_sum = theano.function(inputs=[x], outputs=x_sum)

# uruchamia (lista w Pythonie)
>>> ary = [[1, 2, 3], [1, 2, 3]]
>>> print('Suma kolumn:', calc_sum(ary))
Suma kolumn: [ 2.  4.  6.]

# uruchamia (tablica NumPy)
>>> ary = np.array([[1, 2, 3], [1, 2, 3]],
...                 dtype=theano.config.floatX)
>>> print('Suma kolumn:', calc_sum(ary))
Suma kolumn: [ 2.  4.  6.]
```

Jak już wiemy, podczas korzystania z biblioteki Theano musimy pamiętać jedynie o trzech podstawowych czynnościach: zdefiniowaniu zmiennej, skompilowaniu kodu i jego uruchomieniu. Powyższy przykład pokazuje, że Theano współpracuje zarówno z typem `list` Pythona, jak i `numpy.ndarray` biblioteki NumPy.

Zwróć uwagę, że użyliśmy opcjonalnego argumentu nazwy (w tym wypadku `x`) podczas tworzenia zmiennej tensorowej `fmatrix`, co przydaje się w czasie wyszukiwania błędów w kodzie lub wyświetlenia wykresów. Gdybyśmy np. chcieli wyświetlić wartość `x` bez dodania parametru `name`, funkcja `print` zwróciłaby typ tensora (`TensorType`):

```
>>> print(x)
<TensorType(float32, matrix)>
```

Z kolei w wyniku inicjowania zmiennej `TensorVariable` wraz z argumentem `name='x'` (jak w powyższym przykładzie) funkcja `print` zwróciłaby:

```
>>> print(x)
x
```

Dostęp do obiektu `TensorType` uzyskujemy za pomocą metody `type`:

```
>>> print(x.type())
<TensorType(float32, matrix)>
```

Biblioteka Theano zawiera także bardzo inteligentny system zarządzania pamięcią, przyśpieszający jej działanie poprzez wielokrotne wykorzystywanie poszczególnych komórek. Dokładniej mówiąc, interfejs Theano wykorzystuje przestrzeń pamięci wśród różnych podzespołów — głównych procesorów i kart graficznych; śledzi zmiany w pamięci, przypisując aliasy do poszczególnych buforów. Przenalizujmy teraz zmienną `shared` umożliwiającą rozprzestrzenianie dużych obiektów (tablic) i zapewniającą dostęp do różnych funkcji odczytu oraz zapisu, co pozwala na aktualizowanie tych obiektów już po komplikacji. Szczegółowy opis zarządzania pamięcią przez bibliotekę Theano wykracza poza ramy niniejszej książki, dlatego polecam zapoznanie się z aktualnymi informacjami na ten temat: <http://deeplearning.net/software/theano/tutorial/aliasing.html>.

```
# inicjuje
>>> x = T.fmatrix('x')
>>> w = theano.shared(np.asarray([[0.0, 0.0, 0.0]]),
...                      dtype=theano.config.floatX)
>>> z = x.dot(w.T)
>>> update = [[w, w + 1.0]]

# kompliluje
>>> net_input = theano.function(inputs=[x],
...                                updates=update,
...                                outputs=z)

# uruchamia
>>> data = np.array([[1, 2, 3]],
```

```

...           dtype=theano.config.floatX)
>>> for i in range(5):
...     print('z%d:' % i, net_input(data))
z0: [[ 0.]]
z1: [[ 6.]]
z2: [[ 12.]]
z3: [[ 18.]]
z4: [[ 24.]]

```

Jak widać, współdzielenie pamięci w bibliotece Theano jest naprawdę proste: w powyższym przykładzie zdefiniowaliśmy zmienną `update`, w której zadeklarowaliśmy aktualizację tablicy `w` polegającą na przyroście jej składowych o wartość 1 po każdej iteracji pętli `for`. Po zdefiniowaniu aktualizowanego obiektu oraz sposobu jego aktualizacji przekazaliśmy odpowiednie informacje do parametru `update` stanowiącego część kompilatora `theano.function`.

Kolejną sprytną sztuczką w bibliotece Theano jest wykorzystanie zmiennej `givens` do wstawiania wartości w graf jeszcze przed etapem kompilowania kodu. W ten sposób redukujemy liczbę transferów z pamięci RAM procesora do karty graficznej, co pozwala na przyśpieszenie działania algorytmów uczenia wykorzystujących współdzielone zmienne. Gdybyśmy użyli parametru `inputs` w module `theano.function`, dane byłyby wielokrotnie przenoszone z głównego procesora do karty graficznej, np. podczas wielu przebiegów (epok) algorytmu po zestawie danych w czasie optymalizacji metodą gradientu prostego. Dzięki zmiennej `givens` możemy przechowywać zbiór danych w procesorze graficznym, jeżeli dane będą się mieściły (np. podczas uczenia metodą miniaturowych podzbiorów). Odpowiedzialny za to kod wygląda następująco:

```

# inicjuje
>>> data = np.array([[1, 2, 3]],
...                   dtype=theano.config.floatX)
>>> x = T.fmatrix('x')
>>> w = theano.shared(np.asarray([[0.0, 0.0, 0.0]],
...                               dtype=theano.config.floatX))
...
>>> z = x.dot(w.T)
>>> update = [[w, w + 1.0]]


# kompiluje
>>> net_input = theano.function(inputs=[],
...                                updates=update,
...                                givens={x: data},
...                                outputs=z)


# uruchamia
>>> for i in range(5):
...     print('z:', net_input())
z: [[ 0.]]
z: [[ 6.]]
z: [[ 12.]]
z: [[ 18.]]
z: [[ 24.]]

```

Jak widać w powyższym kodzie, atrybut `givens` stanowi również słownik Pythona rzutujący nazwę zmiennej na rzeczywisty obiekt środowiska Python. W omawianym przypadku ustalamy tę nazwę podczas definiowania obiektu `fmatrix`.

## Przejdźmy do konkretów — implementacja regresji liniowej w Theano

Skoro już nieco zaznajomiliśmy się z biblioteką Theano, przeanalizujmy praktyczny przykład i zaimplementujmy regresję zwykłą metodą najmniejszych kwadratów (**OLS**). Szybką powtórkę z analizy regresywnej znajdziesz w rozdziale 10., „Przewidywanie ciągłych zmiennych docelowych za pomocą analizy regresywnej”.

Zacznijmy od utworzenia małego, jednowymiarowego zestawu danych, składającego się z 10 próbek uczących:

```
>>> X_train = np.asarray([[0.0], [1.0],
...                         [2.0], [3.0],
...                         [4.0], [5.0],
...                         [6.0], [7.0],
...                         [8.0], [9.0]],
...                         dtype=theano.config.floatX)
>>> y_train = np.asarray([1.0, 1.3,
...                         3.1, 2.0,
...                         5.0, 6.3,
...                         6.6, 7.4,
...                         8.0, 9.0],
...                         dtype=theano.config.floatX)
```

Zwróć uwagę, że do konstruowania tablic NumPy wykorzystujemy atrybut `theano.config.floatX`, dzięki czemu w razie potrzeby możemy przełączać się pomiędzy procesorem głównym a graficznym.

Zaimplementujmy teraz funkcję aktualizującą wagi modelu regresji liniowej za pomocą funkcji sumy kwadratów błędów (funkcji kosztu). Zwróć uwagę, że  $w_0$  to jednostka obciążenia (przejęcie z osią  $y$  następuje w punkcie  $x = 0$ ). Wykorzystamy do tego następujący kod:

```
import theano
from theano import tensor as T
import numpy as np

def train_linreg(X_train, y_train, eta, epochs):
    costs = []
    # inicjuje tablice
    eta0 = T.fscalar('eta0')
    y = T.fvector(name='y')
    X = T.fmatrix(name='X')
```

```
w = theano.shared(np.zeros(
    shape=(X_train.shape[1] + 1),
    dtype=theano.config.floatX),
    name='w')

# oblicza funkcję kosztu
net_input = T.dot(X, w[1:]) + w[0]
errors = y - net_input
cost = T.sum(T.pow(errors, 2))

# wykonuje aktualizację gradientu
gradient = T.grad(cost, wrt=w)
update = [(w, w - eta0 * gradient)]

# kompliluje model
train = theano.function(inputs=[eta0],
                        outputs=cost,
                        updates=update,
                        givens={X: X_train,
                                y: y_train,})

for _ in range(epochs):
    costs.append(train(eta))

return costs, w
```

W powyższym przykładzie wykorzystaliśmy bardzo przydatną funkcję grad. Wylicza ona automatycznie pochodną wyrażenia **przy uwzględnieniu** jego parametrów, które przekazaliśmy w postaci atrybutu wrt.

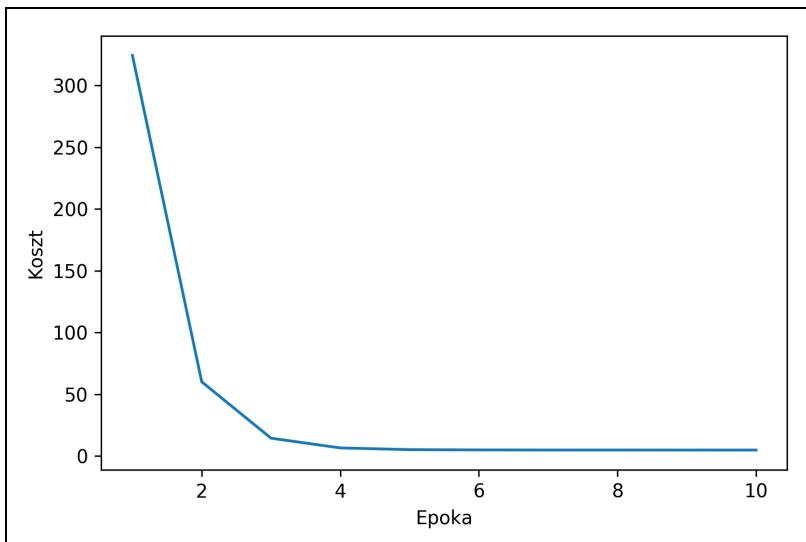
Po zaimplementowaniu funkcji uczącej wytrenujmy nasz model regresji liniowej i sprawdźmy zbieżność algorytmu, obserwując wartości **funkcji SSE** (sumy kwadratów błędów):

```
>>> import matplotlib.pyplot as plt
>>> costs, w = train_linreg(X_train, y_train, eta=0.001, epochs=10)
>>> plt.plot(range(1, len(costs)+1), costs)
>>> plt.tight_layout()
>>> plt.xlabel('Epoka')
>>> plt.ylabel('Koszt')
>>> plt.show()
```

Jak widać na rysunku 13.1, algorytm stał się zbieżny z danymi już po pięciu epokach.

Jak na razie wszystko pięknie; analiza funkcji kosztu pokazuje, że z zestawu danych stworzyliśmy działający model regresji liniowej. Skompilujmy teraz nową funkcję przewidującą wyniki na podstawie wejściowych cech:

```
def predict_linreg(X, w):
    Xt = T.matrix(name='X')
    net_input = T.dot(Xt, w[1:]) + w[0]
```



Rysunek 13.1. Wykres zbieżności algorytmu regresji liniowej napisanego przy użyciu biblioteki Theano

```
predict = theano.function(inputs=[Xt],
                         givens={w: w},
                         outputs=net_input)

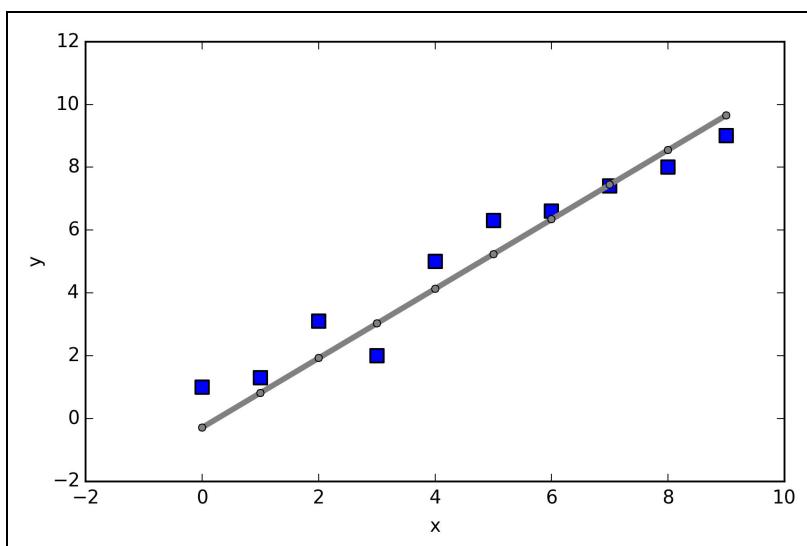
return predict(X)
```

Implementacja funkcji predict zgodnie z trzema fazami wymaganymi przez bibliotekę Theano (definiowanie, komplikowanie, uruchamianie) nie jest zbyt trudna. Stwórzmy teraz wykres dopasowania modelu regresji liniowej do danych uczących:

```
>>> plt.scatter(X_train,
...                 y_train,
...                 marker='s',
...                 s=50)
>>> plt.plot(range(X_train.shape[0]),
...             predict_linreg(X_train, w),
...             color='gray',
...             marker='o',
...             markersize=4,
...             linewidth=3)
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.show()
```

Na podstawie wykresu ukazanego na rysunku 13.2 możemy stwierdzić, że nasz model został odpowiednio wyuczony wobec punktów danych.

Implementacja prostego modelu regresywnego stanowi dobre ćwiczenie w obsłudze interfejsu Theano. Jednak naszym głównym celem jest wykorzystanie tej biblioteki, tj. tworzenie potężnych,



Rysunek 13.2. Wykres dopasowania modelu regresji liniowej do danych uczących

sztucznych sieci neuronowych. Mamy teraz do dyspozycji wszelkie narzędzia niezbędne do wdrożenia wielowarstwowego perceptronu opisanego w rozdziale 12., „Trenowanie sztucznych sieci neuronowych w rozpoznawaniu obrazu”, za pomocą pakietu Theano. Byłoby to jednak dość nudne, nieprawdaż? Dlatego przyjrzymy się jednej z moich ulubionych bibliotek uczenia głębokiego bazujących na interfejsie Theano, pozwalającej na bardzo wygodne eksperymentowanie z sieciami neuronowymi. Zanim jednak przejdziemy do omówienia biblioteki Keras, zastanówmy się nad kwestią doboru funkcji aktywacji dla sieci neuronowych.

## Dobór funkcji aktywacji dla jednokierunkowych sieci neuronowych

Dotychczas w ramach zachowania przejrzystości wykorzystywaliśmy jedynie sigmoidalną funkcję aktywacji w kontekście wielowarstwowych, jednokierunkowych sieci neuronowych; używaliśmy jej zarówno w warstwie ukrytej, jak i wyjściowej podczas omawiania implementacji wielowarstwowego neuronu w rozdziale 12., „Trenowanie sztucznych sieci neuronowych w rozpoznawaniu obrazu”. Nazywamy ją funkcją sigmoidalną — zgodnie z wypracowanym w literaturze użusem — jednak jej precyzyjnieszym określeniem jest funkcja logistyczna lub funkcja ujemnego logarytmu wiarygodności. W kolejnych punktach poznasz alternatywne funkcje sigmoidalne przydatne w implementacjach wielowarstwowych sieci neuronowych.

Z technicznego punktu widzenia możemy wykorzystać w wielowarstwowej sieci neuronowej dowolną funkcję jako funkcję aktywacji, dopóki pozwala ona na rozróżnianie klasyfikowanych

próbek. Możemy nawet stosować liniowe funkcje aktywacji, np. zastosowaną w modelu Adaline (rozdział 2., „Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji”). W praktyce jednak liniowe funkcje aktywacji nie są zbyt przydatne zarówno w warstwie ukrytej, jak i wyjściowej, gdyż w typowej sieci neuronowej pragniemy wprowadzić nieliniowość pozwalającą na rozwiązywanie złożonych zadań problemowych. Suma funkcji liniowych zawsze daje, bądź co bądź, również funkcję liniową.

Użyta przez nas w poprzednim rozdziale logistyczna funkcja aktywacji prawdopodobnie w największym stopniu odwzorowuje działanie rzeczywistej komórki nerwowej: możemy ją sobie wyobrazić jako prawdopodobieństwo pobudzenia neuronu. Logistyczna funkcja aktywacji staje się jednak nieprzydatna w przypadku wielu ujemnych danych wejściowych, ponieważ wówczas jej wartość byłaby zbliżona do zera. Jeśli wartość funkcji sigmoidalnej zbliża się do zera, sieć neuronowa uczy się bardzo powoli i zwiększa się ryzyko utknięcia algorytmu w lokalnym minimum funkcji kosztu na etapie trenowania. Dlatego analitycy często wybierają funkcję **tangensa hiperbolicznego** jako funkcję pobudzenia w warstwach ukrytych. Zanim przejdziemy do omówienia funkcji tangensa hiperbolicznego, przypomnijmy sobie pokrótko podstawowe informacje o funkcji logistycznej i przyjrzyjmy się uogólnieniu ułatwiającemu jej stosowanie w wieloklasowych zadaniach klasyfikacji.

## Funkcja logistyczna — powtóżenie

Jak już wspomniałem we wstępnie do niniejszego podrozdziału, funkcja logistyczna, powszechnie zwana **funkcją sigmoidalną**, jest w rzeczywistości specjalnym przypadkiem właściwej funkcji sigmoidalnej. Pamiętamy z podrozdziału dotyczącego regresji logistycznej w rozdziale 3., „Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn”, że możemy używać funkcji logistycznej do modelowania prawdopodobieństwa przynależności próbki  $x$  do klasy pozytywnej (klasy 1) w klasyfikacji binarnej:

$$\phi_{\text{logistyczna}}(z) = \frac{1}{1 + e^{-z}}$$

Zmienna skalarna  $z$  jest tu zdefiniowana jako całkowite pobudzenie:

$$z = w_0x_0 + \dots + w_mx_m = \sum_{j=0}^m w_jx_j = \mathbf{w}^T \mathbf{x}$$

Zwróć uwagę, że  $w_0$  jest jednostką obciążenia (punkt przecięcia z osią  $y$ ,  $x_0 = 1$ ). Czas na konkretny przykład; załóżmy, że mamy do czynienia z modelem analizującym dwuwymiarowy punkt danych  $x$  oraz zawierającym poniższe współczynniki wag przydzielone do wektora  $w$ :

```
>>> X = np.array([[1, 1.4, 1.5]])
>>> w = np.array([0.0, 0.2, 0.4])

>>> def net_input(X, w):
...     z = X.dot(w)
```

```

...     return z

>>> def logistic(z):
...     return 1.0 / (1.0 + np.exp(-z))

>>> def logistic_activation(X, w):
...     z = net_input(X, w)
...     return logistic(z)

>>> print('P(y=1|x) = %.3f'
...       % logistic_activation(X, w)[0])
P(y=1|x) = 0.707

```

Jeżeli wyliczymy całkowite pobudzenie i wykorzystamy je do aktywacji logistycznego neuronu przy użyciu otrzymanych wartości cech oraz współczynników wag, otrzymamy wynik 0,707, który interpretujemy jako 70,7% prawdopodobieństwa na przynależność próbki  $x$  do klasy pozytywnej. W rozdziale 12., „Trenowanie sztucznych sieci neuronowych w rozpoznawaniu obrazu”, używaliśmy techniki kodowania „gorącojedynkowego” do obliczania wartości w warstwie wyjściowej składającej się z wielu jednostek aktywacji logistycznej. Za chwilę jednak przekonamy się, że warstwa wyjściowa zawierająca wiele jednostek aktywacji logistycznej generuje absurdalne, nieinterpretowalne rezultaty:

```

# W : tablica, wymiary = [n_jednostek_wyjsciowych, n_jednostek_ukrytych+1]
#      Macierz wag łączących warstwę ukrytą z warstwą wyjściową.
# zwróć uwagę, że pierwsza kolumna (W[:,0]) zawiera jednostki obciążenia
>>> W = np.array([[1.1, 1.2, 1.3, 0.5],
...                 [0.1, 0.2, 0.4, 0.1],
...                 [0.2, 0.5, 2.1, 1.9]])

# A : tablica, wymiary = [n_ukrytych+1, n_próbek]
#      Aktywacja warstwy ukrytej.
# zwróć uwagę, że pierwszy element (A[0][0] = 1) jest jednostką obciążenia
>>> A = np.array([[1.0],
...                 [0.1],
...                 [0.3],
...                 [0.7]])

# Z : tablica, wymiary = [n_jednostek_wyjsciowych, n_próbek]
#      Calkowite pobudzenie warstwy wyjściowej.
>>> Z = W.dot(A)
>>> y_probas = logistic(Z)
>>> print('Prawdopodobieństwa:\n', y_probas)
Prawdopodobieństwa:
[[ 0.87653295]
 [ 0.57688526]
 [ 0.90114393]]

```

Jak widać, prawdopodobieństwo przynależności danej próbki do klasy 1 wynosi niemal 88%, do klasy 2 — niemal 58%, a do klasy 3 — 90%. Jest to, oczywiście, niezrozumiałe, ponieważ intuicja

podpowiada nam, że suma poszczególnych prawdopodobieństw cząstkowych powinna wynosić 100%. W rzeczywistości nie jest to duży problem, jeżeli używamy modelu do prognozowania etykiet klas, a nie prawdopodobieństw przynależności określonych instancji do wyznaczonych grup:

```
>>> y_class = np.argmax(Z, axis=0)
>>> print('Przewidywana etykieta klas: %d' % y_class[0])
Przewidywana etykieta klas: 2
```

W pewnych sytuacjach jednak przydają się sensowne wartości prawdopodobieństwa przynależności do klas podczas wieloklasowych predykcji. W dalszej części rozdziału przyjrzymy się uogólnieniu funkcji logistycznej, tzw. **funkcji softmax**, która świetnie nadaje się do wspomnianego zadania.

## Szacowanie prawdopodobieństw w klasyfikacji wieloklasowej za pomocą znormalizowanej funkcji wykładniczej

Znormalizowana funkcja wykładnicza (ang. *softmax function*) stanowi uogólnienie funkcji logistycznej, pozwalające na wyliczanie sensownych prawdopodobieństw przynależności do klas w wieloklasowych konfiguracjach (wielomianowa regresja logistyczna). W funkcji softmax prawdopodobieństwo przynależności danej próbki o całkowitym pobudzeniu  $z$  do  $i$ -tej klasy jest wyliczane za pomocą wyrażenia normalizującego występującego w mianowniku — sumy wszystkich  $M$  funkcji liniowych:

$$P(y = i | z) = \phi_{\text{softmax}}(z) = \frac{e^z}{\sum_{m=1}^M e^z}$$

Zobaczmy, jak się sprawuje znormalizowana funkcja wykładnicza w Pythonie:

```
>>> def softmax(z):
...     return np.exp(z) / np.sum(np.exp(z))

>>> def softmax_activation(X, w):
...     z = net_input(X, w)
...     return softmax(z)

>>> y_probas = softmax(Z)
>>> print('Prawdopodobieństwa:\n', y_probas)
Prawdopodobieństwa:
[[ 0.40386493]
 [ 0.07756222]
 [ 0.51857284]]
>>> y_probas.sum()
1.0
```

Jak widać, prawdopodobieństwa przynależności do klas sumują się teraz do wartości 1, co jest zgodne z przewidywaniami. Warto również zwrócić uwagę, że prawdopodobieństwo przynależności do klasy 2 jest bliskie zeru, ponieważ istnieje olbrzymia rozbieżność pomiędzy  $z_1$  a  $\max(z)$ . Mimo to prognozowana etykieta klas jest taka sama jak w przypadku funkcji logistycznej. Możemy intuicyjnie myśleć o funkcji softmax jako o **znormalizowanej** funkcji logistycznej, przydatnej do uzyskiwania wiarygodnych prognoz przynależności określonych instancji do wyznaczonych grup w środowiskach wieloklasowych.

```
>>> y_class = np.argmax(Z, axis=0)
>>> print('Przewidywana etykieta klas:
... %d' % y_class[0])
Przewidywana etykieta klas: 2
```

## Rozszerzanie zakresu wartości wyjściowych za pomocą funkcji tangensa hiperbolicznego

Kolejną często stosowaną funkcją sigmoidalną w warstwach ukrytych sztucznych sieci neuronowych jest funkcja **tangensa hiperbolicznego** (ang. *hyperbolic tangent; tanh*), którą możemy uznać za przeskalowaną wersję funkcji logistycznej:

$$\phi_{\text{tanh}}(z) = 2 \times \phi_{\text{logistyczna}}(2 \times z) - 1 = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$\phi_{\text{logistyczna}}(z) = \frac{1}{1 + e^{-z}}$$

Przewagą funkcji tangensa hiperbolicznego nad funkcją logistyczną jest szerszy zakres wartości wyjściowych oraz obecność przedziału otwartego  $(-1, 1)$ , co poprawia zbieżność algorytmu wstępnej propagacji (C.M. Bishop, *Neural Networks for Pattern Recognition*, Oxford University Press, 1995, s. 500 – 501). Z kolei funkcja logistyczna zwraca sygnał wyjściowy mieszący się w przedziale otwartym  $(0, 1)$ . Porównajmy wykresy obydwu omawianych typów funkcji:

```
>>> import matplotlib.pyplot as plt

>>> def tanh(z):
...     e_p = np.exp(z)
...     e_m = np.exp(-z)
...     return (e_p - e_m) / (e_p + e_m)

>>> z = np.arange(-5, 5, 0.005)
>>> log_act = logistic(z)
>>> tanh_act = tanh(z)

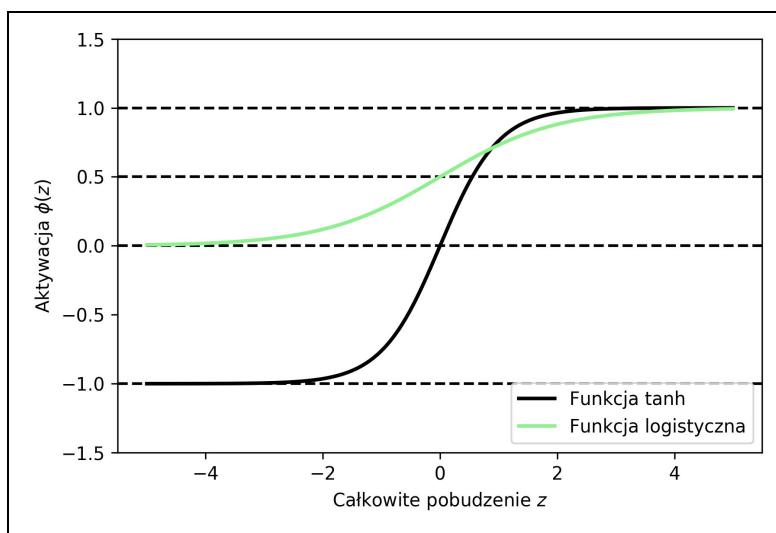
>>> plt.ylim([-1.5, 1.5])
>>> plt.xlabel('Całkowite pobudzenie $z$')
>>> plt.ylabel('Aktywacja $\phi(z)$')
>>> plt.axhline(1, color='black', linestyle='--')
```

```
>>> plt.axhline(0.5, color='black', linestyle='--')
>>> plt.axhline(0, color='black', linestyle='--')
>>> plt.axhline(-1, color='black', linestyle='--')

>>> plt.plot(z, tanh_act,
...             linewidth=2,
...             color='black',
...             label='Funkcja tanh')
>>> plt.plot(z, log_act,
...             linewidth=2,
...             color='lightgreen',
...             label='Funkcja logistyczna')

>>> plt.legend(loc='lower right')
>>> plt.tight_layout()
>>> plt.show()
```

Jak widać na rysunku 13.3, kształty obydwu funkcji sigmoidalnych są do siebie podobne, ale funkcja tanh ma dwukrotnie większą przestrzeń wartości wyjściowych od funkcji logistycznej.



Rysunek 13.3. Porównanie wykresów funkcji tangensa hiperbolicznego i logistycznej

Zauważ, że w celach poglądowych zaimplementowaliśmy funkcje `logistic` i `tanh` dość rozklekle. W praktyce te same rezultaty uzyskamy, korzystając z funkcji `tanh` biblioteki NumPy:

```
>>> tanh_act = np.tanh(z)
```

Ponadto funkcja `logistic` jest dostępna w module `special` biblioteki SciPy:

```
>>> from scipy.special import expit
>>> log_act = expit(z)
```

Skoro już wiemy nieco więcej na temat różnych funkcji aktywacji powszechnie wykorzystywanych w sztucznych sieciach neuronowych, możemy zakończyć ten podrozdział podsumowaniem różnych ich rodzajów, których używaliśmy w niniejszej książce (tabela 13.2).

**Tabela 13.2.** Powszechnie rodzaje funkcji aktywacji

Funkcja aktywacji	Wzór	Przykład	Wykres jednowymiarowy
Skoku jednostkowego (Heaviside'a)	$\phi(z) = \begin{cases} 0, & z < 0 \\ 0,5, & z = 0 \\ 1, & z > 0 \end{cases}$	Odmiana perceptronu	
Signum	$\phi(z) = \begin{cases} -1, & z < 0 \\ 0, & z = 0 \\ 1, & z > 0 \end{cases}$	Odmiana perceptronu	
Liniowa	$\phi(z) = z$	Adaline, regresja liniowa	
Odcinkowo liniowa	$\phi(z) = \begin{cases} -1, & z \geq \frac{1}{2} \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2} \\ 0, & z \leq -\frac{1}{2} \end{cases}$	Maszyna wektorów nośnych	
Logistyczna (sigmoidalna)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Regresja logistyczna, wielowarstwowa sieć neuronowa	
Tangensa hiperbolicznego	$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	Wielowarstwowa sieć neuronowa	

## Skuteczne uczenie sieci neuronowych za pomocą biblioteki Keras

W tym podrozdziale poznamy jedną z najnowszych bibliotek ułatwiających trenowanie sieci neuronowych — Keras. Jej historia sięga początku 2015 roku; obecnie jest jedną z najpopularniejszych i najczęściej używanych bibliotek, które bazują na pakiecie Theano, pozwalającą

na zaprzegnięcie procesora graficznego do uczenia sieci neuronowych. Ma bardzo intuicyjny interfejs umożliwiający implementowanie sieci neuronowych przy użyciu zaledwie kilku wierszy kodu. Po zainstalowaniu interfejsu Theano możesz wdrożyć na swój komputer bibliotekę Keras z repozytorium PyPI, wpisawszy w wierszu poleceń następującą komendę:

```
pip install Keras
```

Więcej informacji na temat biblioteki Keras znajdziesz na jej oficjalnej stronie: <https://keras.io/>.

Aby się przekonać, jak wygląda uczenie sieci neuronowej przy użyciu pakietu Keras, zaimplementujemy model wielowarstwowego perceptronu służący do klasyfikowania odreźnie pisanych cyfr, tworzących opisany w poprzednim rozdziale zestaw danych MNIST. Zestaw ten jest dostępny pod adresem <http://yann.lecun.com/exdb/mnist/> w czterech plikach:

- *train-images-idx3-ubyte.gz*: zestaw obrazów uczących (9 912 422 bajty),
- *train-labels-idx1-ubyte.gz*: zestaw etykiet danych uczących (28 881 bajtów),
- *t10k-images-idx3-ubyte.gz*: zestaw obrazów testowych (1 648 877 bajtów),
- *t10k-labels-idx1-ubyte.gz*: zestaw etykiet testowych (4542 bajty).

Po pobraniu i rozpakowaniu skompresowanych plików umieszczamy je w podkatalogu *mnist* stworzonym w bieżącym katalogu roboczym, co pozwoli nam na wczytanie danych uczących i testowych za pomocą poniższej funkcji:

```
import os
import struct
import numpy as np

def load_mnist(path, kind='train'):
    """wczytuje dane MNIST umieszczone w katalogu roboczym"""
    labels_path = os.path.join(path,
                               '%s-labels.idx1-ubyte'
                               % kind)
    images_path = os.path.join(path,
                               '%s-images.idx3-ubyte'
                               % kind)

    with open(labels_path, 'rb') as lbpath:
        magic, n = struct.unpack('>II',
                                 lbpath.read(8))
        labels = np.fromfile(lbpath,
                             dtype=np.uint8)

    with open(images_path, 'rb') as imgpath:
        magic, num, rows, cols = struct.unpack(">IIII",
                                                imgpath.read(16))
        images = np.fromfile(imgpath,
                            dtype=np.uint8).reshape(len(labels), 784)

    return images, labels
X_train, y_train = load_mnist('mnist', kind='train')
```

```

print('Wiersze: %d, kolumny: %d' % (X_train.shape[0], X_train.shape[1]))
Wiersze: 60000, kolumny: 784
X_test, y_test = load_mnist('mnist', kind='t10k')
print('Wiersze: %d, kolumny: %d' % (X_test.shape[0], X_test.shape[1]))
Wiersze: 10000, kolumny: 784

```

W dalszej części rozdziału będziemy krok po kroku analizować wykorzystujące bibliotekę Keras pliki kodu źródłowego, które możemy uruchamiać bezpośrednio w interpreterze Pythona. Jeżeli jednak chcesz w tym celu skorzystać z mocy procesora graficznego, możesz wstawić kod do skryptu Pythona lub ściągnąć odpowiedni plik ze strony wydawnictwa Helion. Aby uruchomić skrypt Pythona na procesorze graficznym, wprowadź następujące polecenie w katalogu zawierającym plik *mnist\_keras\_mlp.py*:

```

set THEANO_FLAGS="mode=FAST_RUN" & set THEANO_FLAGS="device=gpu" & set
THEANO_FLAGS="floatX=float32" & python mnist_keras_mlp.py

```

Kolejnym etapem przygotowywania danych uczących jest przekształcenie obrazów MNIST do formatu 32-bitowego:

```

>>> import theano
>>> theano.config.floatX = 'float32'
>>> X_train = X_train.astype(theano.config.floatX)
>>> X_test = X_test.astype(theano.config.floatX)

```

Musimy teraz przekonwertować etykiety klas (liczby całkowite 1 – 9) do postaci „gorąco-edykowej”. Na szczęście biblioteka Keras jest zaopatrzona w odpowiednie narzędzie<sup>1</sup>:

```

>>> from keras.utils import np_utils
>>> print('Pierwsze trzy etykiety: ', y_train[:3])
Pierwsze trzy etykiety: [5 0 4]
>>> y_train_ohe = np_utils.to_categorical(y_train)

```

<sup>1</sup> Biblioteka Keras używa domyślnie silnika TensorFlow, co uniemożliwia korzystanie z omawianego kodu. Aby zmienić silnik na Theano, należy najpierw znaleźć lub utworzyć plik *keras.json*. Zlokalizujemy go, wpisawszy w interpreterze Pythona następujący fragment kodu:

```

import os
print(os.path.expanduser('~'))
C:\\Users\\Nazwa_użytkownika'

```

Dzięki powyższemu kodowi zostaje wyświetlony katalog główny, w którym powinien znajdować się podkatalog *keras* zawierający plik *keras.json*. W przypadku braku tego pliku należy go stworzyć i wstawić następującą zawartość:

```

{
    "floatx": "float32",
    "epsilon": 1e-07,
    "backend": "theano",
    "image_data_format": "channels_last"
}

```

— przyp. tłum.

```
>>> print('\nPierwsze trzy etykiety (zakodowane „gorącojedynkowo”):\n',
y_train_ohe[:3])
Pierwsze trzy etykiety (zakodowane „gorącojedynkowo”):
[[ 0.  0.  0.  0.  0.  1.  0.  0.  0.]
 [ 1.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  0.  0.  0.]]
```

Przechodzimy teraz do ciekawszej części, w której zaimplementujemy sieć neuronową. Wykorzystamy tu tę samą architekturę co w rozdziale 12., „Trenowanie sztucznych sieci neuronowych w rozpoznawaniu obrazu”. Z tym że jednostki logistyczne warstwy ukrytej zastąpimy funkcjami aktywacji (tangensa hiperbolicznego), a funkcję logistyczną warstwy wyjściowej — znormalizowaną funkcją wykładniczą, a także wstawimy dodatkową warstwę ukrytą. Czynności te nie stanowią problemu dla biblioteki Keras, o czym się przekonasz, uruchomiony poniższy kod:

```
>>> from keras.models import Sequential
>>> from keras.layers.core import Dense
>>> from keras.optimizers import SGD

>>> np.random.seed(1)

>>> model = Sequential()
>>> model.add(Dense(input_dim=X_train.shape[1],
...                   units=50,
...                   kernel_initializer='uniform',
...                   activation='tanh'))

>>> model.add(Dense(input_dim=50,
...                   units=50,
...                   kernel_initializer='uniform',
...                   activation='tanh'))

>>> model.add(Dense(input_dim=50,
...                   units=y_train_ohe.shape[1],
...                   kernel_initializer='uniform',
...                   activation='softmax'))

>>> sgd = SGD(lr=0.001, decay=1e-7, momentum=.9)
>>> model.compile(loss='categorical_crossentropy', optimizer=sgd,
metrics=['accuracy'])
```

Najpierw za pomocą klasy `Sequential` inicjujemy nowy model jednokierunkowej sieci neuronowej. Następnie dodajemy do niego dowolną liczbę warstw. Pamiętajmy jednak, że pierwsza warstwa jest wejściowa, dlatego musimy się upewnić, że parametr `input_dim` będzie miał taką samą wartość jak liczba cech (kolumn) w zestawie uczącym (w naszym przypadku 768). Musimy także zapewnić taką samą liczbę jednostek wyjściowych (`units`) i wejściowych (`input_dim`) w dwóch sąsiadujących ze sobą warstwach. W omawianym przykładzie dodaliśmy dwie warstwy ukryte zawierające po 50 jednostek oraz po 1 jednostce obciążenia. Zwróć uwagę, że jednostki obciążenia w całkowicie połączonych sieciach w bibliotece Keras są inicjowane z wartością 0. Jest to różnica w porównaniu z implementacją modelu MLP, opisaną w rozdziale 12., „Trenowanie

sztucznych sieci neuronowych w rozpoznawaniu obrazu” — wówczas przydzieliśmy jednostkom obciążenia wartość 1, co jest powszechniejszą (ale niekoniecznie lepszą) konwencją.

Również liczba jednostek w warstwie wyjściowej powinna się zgadzać z liczbą unikatowych etykiet klas — liczbą kolumn w „gorącojedynkowo” zakodowanej tablicy etykiet klas. Przed komplikacją modelu musimy jeszcze skonfigurować optymalizator. W omawianym przykładzie wybraliśmy optymalizację metodą stochastycznego spadku wzduż gradientu, z którym zapoznaliśmy się już w poprzednich rozdziałach. Ponadto możemy zdefiniować wartości stałej rozkładu wag oraz uczenia momentowego w celu skonfigurowania szybkości uczenia w każdej metodzie zgodnie z informacjami zawartymi w rozdziale 12., „Trenowanie sztucznych sieci neuronowych w rozpoznawaniu obrazu”. Na koniec wyznaczamy funkcję koszta (lub straty) jako `categorical_crossentropy`. Entropia krzyżowa (binarna) to nic innego jak funkcja koszta w modelu regresji logistycznej, a kategoryzująca entropia krzyżowa stanowi jej uogólnienie dla wieloklasowego prognozowania za pomocą algorytmu softmax. Po skompilowaniu modelu możemy go wytrainować przy użyciu metody `fit`. Stosujemy tu metodę stochastycznego gradientu, która bazuje na podzbiorach danych składających się z 300 próbek uczących. Trenujemy model MLP przez 50 epok i możemy obserwować wyniki optymalizacji funkcji `cost`, wyznaczyszy atrybut `verbose=1`. Bardzo przydatny jest parametr `validation_split`, ponieważ rezerwuje on 10% danych uczących (w tym wypadku 6000 próbek) na walidację modelu po każdej epoce, dzięki czemu możemy na bieżąco sprawdzać, czy sieć neuronowa ulega przetrenowaniu.

```
>>> model.fit(X_train, y_train_ohe,
...             epochs=50,
...             batch_size=300,
...             verbose=1,
...             validation_split=0.1)
Train on 54000 samples, validate on 6000 samples
Epoch 0
54000/54000 [=====] - 1s - loss: 2.2290 -
acc: 0.3592 - val_loss: 2.1094 - val_acc: 0.5342
Epoch 1
54000/54000 [=====] - 1s - loss: 1.8850 -
acc: 0.5279 - val_loss: 1.6098 - val_acc: 0.5617
Epoch 2
54000/54000 [=====] - 1s - loss: 1.3903 -
acc: 0.5884 - val_loss: 1.1666 - val_acc: 0.6707
Epoch 3
54000/54000 [=====] - 1s - loss: 1.0592 -
acc: 0.6936 - val_loss: 0.8961 - val_acc: 0.7615
...
Epoch 49
54000/54000 [=====] - 1s - loss: 0.1907 -
acc: 0.9432 - val_loss: 0.1749 - val_acc: 0.9482
```

Wyświetlanie wartości funkcji koszta w trakcie uczenia jest niezwykle użyteczne, ponieważ możemy szybko zauważyc, czy w czasie trenowania maleje koszt, oraz zatrzymać działanie algorytmu w celu dalszego dostrajania hiperparametrów.

W celu przewidywania etykiet klas możemy użyć metody `predict_classes`, która zwraca bezpośrednio etykiety klas jako liczby całkowite:

```
>>> y_train_pred = model.predict_classes(X_train, verbose=0)
>>> print('Pierwsze trzy prognozy: ', y_train_pred[:3])
Pierwsze trzy prognozy: [5 0 4]
```

Sprawdźmy w końcu dokładność modelu wobec danych uczących i testowych:

```
>>> train_acc = np.sum(
...     y_train == y_train_pred, axis=0) / X_train.shape[0]
>>> print('Dokładność wobec danych uczących: %.2f%%' % (train_acc * 100))
Dokładność wobec danych uczących: 94.51%

>>> y_test_pred = model.predict_classes(X_test, verbose=0)
>>> test_acc = np.sum(y_test == y_test_pred,
...     axis=0) / X_test.shape[0]
print('Dokładność wobec danych testowych: %.2f%%' % (test_acc * 100))
Dokładność wobec danych testowych: 94.39%
```

Zwróci uwagę, że mamy tu do czynienia jedynie z bardzo prostą siecią neuronową bez zoptymalizowanych parametrów strojenia. Jeżeli chcesz lepiej poznać bibliotekę Keras, nie krępuj się i pobaw się wartościami współczynnika uczenia, uczenia momentowego, stałej rozkładu wag oraz liczby jednostek ukrytych.

Biblioteka Keras jest znakomitym narzędziem do implementowania i testowania sieci neuronowych, ale istnieje wiele innych „nakładek” na Theano, o których warto wspomnieć. Wybitnym przykładem jest biblioteka Pylearn2 (<http://deeplearning.net/software/pylearn2>), zaprojektowana w laboratorium LISA (obecnie MILA) w Montrealu. Może Cię również zainteresować biblioteka Lasagne (<https://github.com/Lasagne/Lasagne>), jeżeli preferujesz minimalistyczne, ale rozszerzalne interfejsy gwarantujące większą kontrolę nad kodem pisany w pakiecie Theano.

## Podsumowanie

Mam nadzieję, że podobał Ci się ostatni rozdział podróży po fascynującym świecie uczenia maszynowego. W trakcie lektury niniejszej książki przyjrzaliśmy się wszystkim najważniejszym gałęziom tej dziedziny i teraz bez większego problemu powinieneś wdrażać omówione techniki do rozwiązywania rzeczywistych problemów.

Rozpoczęliśmy naszą podróż od pobicznego przeglądu różnych rodzajów uczenia maszynowego: nadzorowanego, nienadzorowanego i ze wzmacnieniem. Zapoznaliśmy się z kilkoma różnymi algorytmami uczącymi służącymi do klasyfikowania danych, począwszy od jednowarstwowych sieci neuronowych w rozdziale 2., „Trenowanie algorytmów uczenia maszynowego w celach klasyfikacji”. Następnie w rozdziale 3., „Stosowanie klasyfikatorów uczenia maszynowego za

pomocą biblioteki scikit-learn”, zajęliśmy się bardziej zaawansowanymi algorytmami klasyfikującymi, a w rozdziałach 4., „Tworzenie dobrych zbiorów uczących — wstępne przetwarzanie danych”, i 5., „Kompresja danych poprzez redukcję wymiarowości”, poznaliśmy najważniejsze aspekty kolejkowania algorytmów uczenia maszynowego. Pamiętaj, że nawet najbardziej złożone algorytmy są zawsze ograniczane przez informacje zawarte w danych uczących. W rozdziale 6., „Najlepsze metody oceny modelu i strojenie parametryczne”, nauczyliśmy się wykorzystywać najlepsze rozwiązania pozwalające na tworzenie i ocenianie modeli uczenia maszynowego, co stanowi kolejny niezmiernie istotny aspekt tej dziedziny. Jeżeli pojedynczy algorytm nie jest w stanie osiągnąć pożąданej przez nas skuteczności, czasami warto projektować modele zespołowe, generujące prognozy metodą głosowania większościowego. Zajęliśmy się tą kwestią w rozdziale 7., „Łączenie różnych modeli w celu uczenia zespołowego”. Z kolei w rozdziale 8., „Wykorzystywanie uczenia maszynowego w analizie sentymentów”, wykorzystaliśmy uczenie maszynowe do analizowania obecnie prawdopodobnie najczęściejowej formy informacji dominującej w internetowych mediach społecznościowych: dokumentów tekstowych. Techniki uczenia maszynowego jednak nie ograniczają się wyłącznie do analizowania lokalnych danych, a w rozdziale 9., „Wdrażanie modelu uczenia maszynowego do aplikacji sieciowej”, mogliśmy się przekonać, w jaki sposób należy wdrożyć model uczenia maszynowego do aplikacji sieciowej i podzielić się nim z resztą świata. Przez większość czasu koncentrowaliśmy się na zadaniach klasyfikacji — chyba najpopularniejszym zastosowaniu tej dziedziny. Tymczasem jej możliwości są znacznie większe! W rozdziale 10., „Przewidywanie ciągłych zmiennych docelowych za pomocą analizy regresywnej”, przeanalizowaliśmy kilka algorytmów analizy regresywnej, pozwalających na przewidywanie wyjściowych zmiennych ciągłych. Kolejnym fascynującym działem uczenia maszynowego jest analiza skupień, pomagająca w wykrywaniu ukrytych struktur danych, nawet w przypadku, gdy dane uczące same w sobie nie mają pozytycznych informacji. Temu zagadnieniu poświęciłem rozdział 11., „Praca z nieoznakowanymi danymi — analiza skupień”.

Dwa ostatnie rozdziały zawierały krótkie wprowadzenie do najpiękniejszych i najbardziej ekscytujących algorytmów uczenia maszynowego: sztucznych sieci neuronowych. Chociaż szczegółowy opis uczenia głębokiego wykracza poza rama niniejszej książki, mam nadzieję, że udało mi się rozpalić w Tobie ciekawość wystarczającą do śledzenia nowinek z tej gałęzi uczenia maszynowego. Jeżeli zamierzasz zostać zawodowym badaczem uczenia maszynowego albo po prostu chcesz być na bieżąco z najnowszymi osiągnięciami w tej dziedzinie, zalecam zapoznanie się z publikacjami wiodących ekspertów, takich jak Geoff Hinton (<http://www.cs.toronto.edu/~hinton/>), Andrew Ng (<http://www.andrewng.org/>), Yann LeCun (<http://yann.lecun.com/>), Juergen Schmidhuber (<http://people.idsia.ch/~juergen/>) czy Yoshua Bengio ([http://www.iro.umontreal.ca/~bengioy/yoshua\\_en/index.html](http://www.iro.umontreal.ca/~bengioy/yoshua_en/index.html)). Zachęcam również do zapisania się na listy mailingowe projektów scikit-learn, Theano i Keras oraz udziału w interesujących dyskusjach dotyczących wymienionych bibliotek, a także całokształtu uczenia maszynowego. Czekam z niecierpliwością na Ciebie! Jeżeli masz jakiekolwiek pytania odnośnie do niniejszej książki lub szukasz porad na temat uczenia maszynowego, zawsze chętnie służę pomocą.

Mam nadzieję, że nasza wspólna podróż po krainie uczenia maszynowego była dla Ciebie warta zachodu oraz że zdobyleś wiele nowych i przydatnych umiejętności, które umożliwią Ci rozwój kariery i które jeszcze nieraz wykorzystasz w prawdziwym świecie.



# Skorowidz

## A

agent, 29, 30  
agregacja, 221, 223, 224  
akson, 42  
algebra  
    liniowa, 32, 43, 342  
    symboliczna, 378  
algorytm, 35, *Patrz też:* klasyfikator, metoda  
    AdaBoost, 226, 227, 228  
        implementacja, 230  
        skuteczność, 230  
    Adaline, 54, 55, 60, 122, 158, 283, 337, 338,  
        369  
        funkcja kosztu, 55, 57  
        implementacja, 57, 59, 77, 81  
    AgglomerativeClustering, 328  
    agregacji, *Patrz:* agregacja  
    alokacji ukrytej zmiennej Dirichleta, 250  
    analizy słów-twarzcej, 243, 244  
    BaggingClassifier, 223  
    błąd generalizacji, *Patrz:* błąd generalizacji  
    centroidów, 308, 309, 311, 313, 329, 330, 331,  
        333  
        implementacja, 310  
        liczba skupień, 312  
        standaryzacja, 311  
        zbieżność, 310  
    DBSCAN, 328, 329, 331, 332, 333  
    dopasowanie nadmierne, *Patrz:* dopasowanie  
        nadmierne  
    dyskryminacji liniowej, *Patrz:* algorytm LDA  
        Fishera, 150  
    elastycznej siatki, 294  
    eliminacji wstępnej, 134  
    FCM, *Patrz:* algorytm rozmytych c-srednich

głosowania większościowego,  
    *Patrz:* głosowanie większościowe  
gradientu prostego, 57, 56, 60, 62, 63, 368  
iteracyjny spadek wzduż gradientu, *Patrz:*  
    algorytm stochastycznego spadku wzduż  
    gradientu  
k-krotnego sprawdzianu krzyżowego, 180, 182,  
    183, 193, 214, 245, 246  
    dokładność, 194  
    implementacja, 184, 185, 193  
    warstwowy, 183  
k-krotnej krossvalidacji, *Patrz:* algorytm  
    k-krotnego sprawdzianu krzyżowego  
klasteryzacji spektralnej, 333  
klasyfikacyjny, 41, 76  
k-means++, 311  
k-medoidów, 311  
k-najbliższych sąsiadów, *Patrz:* algorytm KNN  
KNN, 106, 107, 122, 132, 214, 215  
    implementacja, 108  
    przetrenowanie, 109  
konsensusu próby losowej, *Patrz:* algorytm  
    RANSAC  
kontaminacyjny, 221  
k-srednich, *Patrz:* algorytm centroidów  
    miękkich, *Patrz:* algorytm rozmytych  
    c-srednich  
Lancaster, 244  
LASSO, 294, 295  
LDA, 150, 151, 154, 155  
    implementacja, 156  
LinearRegression, 286  
LOO, 183  
losowego lasu, *Patrz:* las losowy  
łączenie, 203  
metryka, *Patrz:* metryka

## algorytm

- minimalizacji wewnętrzgrupowej sumy kwadratów błędów, 310, 315
- minus jednego elementu, *Patrz:* algorytm LOO MLP, 338, 340, 341, 342, 358, 370, 372, 376, 387 implementacja, 348, 352 przetrenowanie, 355, 356 trenowanie, 353 warstwa ukryta, 338, 339, 358, 359, 360 warstwa wejściowa, 338 warstwa wyjściowa, 338, 358, 359
- MSE, 292, 293 najmniejszych kwadratów, *Patrz:* algorytm OLS neuronu liniowego adaptacyjnego, 41 nieparametryczny, 107 o dużej wariancji, 84, 124 o dużym obciążeniu, 84 OLS, 282, 283, 287, 384, 385 dopasowanie, 284
- Paice/Husk, *Patrz:* algorytm Lancaster parametryczny, 107 PCA, 140, 147, 150, 155, 167, 169, 170, 179, 280 implementacja, 147, 148 jądrowy, 159, 160, 162, 164, 165, 167, 169, 171, 174
- perceptronu, 41, 54, 75, 87, 91, 107 implementacja, 47, 48 progowego, 44, 45, 46 Rosenblatta, 336 trenowanie, 50, 51, 71, 72 wielowarstwowego, *Patrz:* algorytm MLP Portera, 243, 244 propagacji w przód, 340, 341, 359 propagacji wstecznej, 336, 339, 358, 359, 362 ujęcie intuicyjne, 361 przeszukiwania siatki, 191, 192, 193, 220, 246 RANSAC, 288, 290 implementacja, 289 regresji liniowej, *Patrz:* regresja liniowa regresji logistycznej, 76, 77, 79, 90, 91, 107, 122, 158, 204, 214, 215, 221, 245, 249, 341, 363, 369 implementacja, 81 rozmytych średnich, 311, 313, 314 sąsiedztwa najbliższego, *Patrz:* algorytm wiązania pojedynczego najdalszego, *Patrz:* algorytm wiązania pełnego SBS, 129, 130, 132, 140

sekwencyjnego wyboru cech, 129, 130

- sekwencyjnej selekcji wstępnej, *Patrz:* algorytm SBS skuteczność, 35, 36, 70, 180, 195, 197 badanie, 71 Snowball, 244 stochastycznego spadku wzduż gradientu, 62, 63, 91, 247, 368 implementacja, 63, 66 stopa błędu, *Patrz:* stopa błędu strojenie, *Patrz:* dobór modelu SVM, *Patrz:* maszyna wektorów nośnych tf-idf, 239, 240, 245 trenowanie, 70 waga, 207, 208, 209 ważenia częstości termów — odwrotna częstość w tekście, *Patrz:* algorytm tf-idf wiarygodność, 79, 90, *Patrz też:* funkcja wiarygodności wiązania pełnego, 321, 323, 330 pojedynczego, 321 word2vec, 250 worka słów, *Patrz:* worek słów wrażliwy na losowość danych uczących, 84 wstępniego uczenia głębskich sieci neuronowych, 339 wyboru cech, 134, 140 wybór, 70 wyczerpującego przeszukiwania, 130 wydzielania, 180, 181, 182 wzmacniania, *Patrz:* wzmacnianie zachłanny, 129, 130 zachowanie trwałości modelu, 252, 271 zbieżność, 51, 52, 54, 60, 89, 284 Anaconda, 37 analiza dyskryminacyjna liniowa, *Patrz:* algorytm LDA EDA, 278 eksploracyjna danych, *Patrz:* analiza EDA głównych składowych, *Patrz:* algorytm PCA graficzna, 73 PCA, *Patrz:* algorytm PCA profilu, 316 regresji, 28 sentymentów, 235, 237, 238, 239, 241, 242, 247, 250 skupień, 30, 307, 308, *Patrz też:* klasteryzacja aglomeracyjna, 321, 323, 328, 333 analiza profilu, *Patrz:* analiza profilu

- gęstościowa, 308, 333
  - grafowa, 333
  - hierarchiczna, 308, 320, 323, 329, 331
  - miękka, 313
  - na macierzy odległości, 323, 324
  - skupienie puste, 311
  - skuteczność, 318
  - spektralna, 333
  - twarda, 312
  - aplikacja
    - Graphviz, *Patrz:* Graphviz
    - Jupyter Notebook, 253, 324
    - sieciowa, 262
      - publikowanie, 269, 270
      - serializacja, 271
      - tworzenie, 257, 258
      - uruchamianie, 269
    - szachowa, 30
  - architektura SIMD, *Patrz:* SIMD
  - artificial neural network, *Patrz:* sieć neuronowa sztuczna
  - average linkage, *Patrz:* metoda średnich połączeń
- B**
- backpropagation algorithm, *Patrz:* algorytm propagacji wstecznej
  - bagging, *Patrz:* agregacja
  - bag-of-words model, *Patrz:* worek słów
  - Baidu DeepSpeech, 336
  - batch gradient descent, *Patrz:* algorytm wsadowy gradientu prostego
  - baza danych SQLite, 254
    - aktualizacja, 271
    - w Pythonie, 254
  - Bezdek James, 313
  - bias unit, *Patrz:* jednostka obciążenia
  - biblioteka
    - BLAS, 49
    - Flask, *Patrz:* Flask
    - Jinja2, 260
    - joblib, 252
    - Keras, 373, 393, 394, 395, 396, 398
    - LAPACK, 49
    - Lasagne, 373, 398
    - LIBLINEAR, 90, 286
    - LIBSVM, 90
    - matplotlib, 37, 38, 47, 196
      - lista kolorów, 325
      - wykres, 280
- NLTK, 243, 244
  - NumPy, 36, 37, 47, 49, 59, 91, 113, 143, 178, 252, 281, 342, 378, 380, 382
  - pandas, 37, 38, 47, 50, 113, 119, 178, 236
  - pickle, 252
  - Pylearn2, 373, 398
  - PyPrind, 236, 249
  - rozszerzeń, 36
  - scikit-learn, 36, 38, 60, 70, 71, 75, 81, 85, 105, 134, 147, 156, 175, 178, 185, 187, 201, 214, 286, 289, 296, 317
    - estymator, 115, 118
    - interfejs, 72, 102, 113
  - SciPy, 36, 37, 323
  - seaborn, 278, 280
  - sqlite3, 254
  - SymPy, 378
  - Theano, 373, 376, 377, 378, 384
    - adresowanie pamięci, 379
    - instalacja, 378
    - konfiguracja, 379
    - obiekt Variable, 379
    - struktura tablicowa, 381
    - typ zmiennych, 379, 380
    - współdzielenie pamięci, 383
    - zarządzanie pamięcią, 382, 383
    - zmienna givens, 383
  - WTForms, 258, 260
  - blok rozmywania, 314
  - błąd, 197
  - generalizacji, 36
  - klasyfikacji, 51, 73, 98
  - predykcji, 276
  - stopa, *Patrz:* stopa błędu
  - suma kwadratów, *Patrz:* SSE
  - średniokwadratowy, *Patrz:* MSE
  - boosting, *Patrz:* wzmacnianie
  - bootstrap sample, *Patrz:* próbka początkowa
  - border point, *Patrz:* punkt graniczny
  - bramka logiczna, 42
  - XOR, 91, 94
  - Breiman Leo, 222, 226
- C**
- Cascading Style Sheet, *Patrz:* CSS
  - cecha
    - detektor, *Patrz:* detektor cech
    - istotność, *Patrz:* cecha ważność
    - mapa, *Patrz:* mapa cech

- cecha  
nominalna, 116, 118, 119  
numeryczna, 116  
odkrywanie, 129, 139, 140, 150, 179  
porządkowa, 116  
    mapowanie, 116  
skalowanie, 121, 122  
    normalizacja, *Patrz*:  
        standaryzacja  
sztuczna, 119  
usuwanie, 130  
ważność, 134, 135, 144  
wybór, 129
- centroid, 308, 309  
cienkokońcowość, 346
- cluster inertia, *Patrz*: kłaster bezwładności
- clustering, *Patrz*: analiza skupień, klasteryzacja
- CNN, *Patrz*: sieć neuronowa splotowa
- complete linkage, *Patrz*: algorytm wiązania pełnego
- confusion matrix, *Patrz*: macierz pomyłek
- convolutional neural networks, *Patrz*: sieć neuronowa splotowa
- core point, *Patrz*: punkt rdzeniowy
- CPU  
    cena, 377  
    liczba zmiennoprzecinkowa, 380
- CSS, 260, 261
- curse of dimensionality, *Patrz*: kłata wymiarowości
- D**
- dane  
    analiza eksploracyjna, *Patrz*: analiza EDA  
    baza, *Patrz*: baza danych  
    brakujące, 111, 113  
        usuwanie, 113  
        wstawianie, *Patrz*: interpolacja  
    cecha, *Patrz*: cecha  
    interpolacja, *Patrz*: interpolacja  
    jakość, 111  
    kategoryzujące, 116  
    kompresja, 139, 140, 143  
    liniowo nierozdzielne, *Patrz*: klasa  
        nierozdzielna liniowo  
    NaN, 112  
    nieodstające, 288, 289
- nieoznakowane, 30
- odstające, *Patrz*: próbka odstająca  
przetwarzanie wstępne, 34, 121, 177
- ramka, 112
- reprezentatywne, *Patrz*: medoid
- rozkład normalny, 150, 153
- rzutowanie, 156, 157, 158, 159, 160, 162, 167, 168, 169, 170
- serializacja, *Patrz*: serializacja
- standaryzacja, *Patrz*: standaryzacja
- szum, *Patrz*: szum
- tekstowe, *Patrz*: tekst
- testowe, 35, 71, 84, 120, 121, 178, 181, 182, 223  
    próbka początkowa, *Patrz*: próbka początkowa  
    uczące, 27, 35, 71, 120, 121, 124, 178, 181, 182, 223  
        dobór modelu, 36  
        losowość, 84  
        przyrostowe, 63  
    validacyjne, 132, 181, 182, *Patrz*: dane testowe wielowymiarowe, 31  
wizualizacja, 31, 37, *Patrz też*: wykres zbiór  
    Breast Cancer Wisconsin, 178, 199  
    Housing, 277, 281, 297, 303  
    IMDb, 236  
    Iris, 31, 33, 34, 50, 71, 94, 213  
    MNIST, 344, 346, 348, 376, 395  
    Wine, 120, 134, 141, 145, 147, 223  
decision tree, *Patrz*: drzewo decyzyjne
- deep learning, *Patrz*: uczenie maszynowe głębokie
- DeepSpeech, 336
- dendrogram, 320, 325, 326, 327
- dendryt, 42
- density-based spatial clustering of applications with noise, *Patrz*: algorytm DBSCAN
- deserializacja, 252, 253
- detektor cech, 336, 370, 371
- dimensionality reduction, *Patrz*: redukcja wymiarowości
- dobór modelu, 181, 182
- dopasowanie  
    nadmierne, 73, 81, 84, 123, 188, *Patrz też*: wariancja  
    zapobieganie, 84, 186, 189  
    zbyt małe, 84, 186  
    zapobieganie, 189

drzewo  
 decyzyjne, 97, 194, 204, 214, 215, 220, 223  
 binarne, 98  
 korzeń, 97  
 pień, 226  
 przetrenowanie, 101, 224, 225  
 przycinanie, 97, 99  
 skuteczność, 224  
 tworzenie, 101  
 węzeł, 98  
 wykres, 103  
 zespół, *Patrz:* las losowy  
 katalogów, 257, 262  
 operacji, 378  
 dummy feature, *Patrz:* cecha sztuczna  
 Dunn Joseph, 313  
 dyskryminanta liniowa, 155  
 analiza dyskryminacyjna, *Patrz:* algorytm LDA

**E**

e regression, *Patrz:* regresja grzbietowa  
 elbow method, *Patrz:* metoda łokcia  
 ensemble method, *Patrz:* metoda zespołowa  
 entropia, 98, 99, 102  
 krzyżowa, 397  
 epoka, 46, 48, 52, 59, 284, 337  
 etykieta klasy, 27, 44, 97, 153, 204, 237  
 kodowanie, 117  
 prognozowanie, 390  
 przewidywanie, 27, 79, 204, 221  
 rozkład w węźle, 214  
 exploratory data analysis, *Patrz:* analiza EDA

**F**

F1 score, *Patrz:* wynik F1  
 Facebook, 336  
 FCM, *Patrz:* algorytm rozmytych c-średnich  
 feature detector, *Patrz:* detektor cech  
 feature extraction, *Patrz:* cecha odkrywanie  
 filtr antyspamowy, 27, 239, 247, 251  
 Fisher Ronald, 150  
 Flask, 256, 257, 262  
 debugger, 260  
 fold cross-validation, *Patrz:* algorytm k-krotnego sprawdzianu krzyżowego  
 formularz, 258, 261  
 forward propagation, *Patrz:* propagacja w przód

Freund Yoav, 226  
 funkcja  
 aktywacji, 42, 43, 55, 78, 283, 341, 360, 387, 393  
 liniowa, 388  
 celu, 55, 98  
*Heaviside'a*, 393  
 jądra, 92, 94, 162, 174  
 sigmoidalna, 162  
 sztuczka, 93, 94, 160, 171, 305  
 wielomianowa, 162  
 kosztu, 55, 57, 79, 80, 81, 283, 337, 353, 362, 384  
 gradient, 56  
 logistyczna, 356, 357  
 minimalizowanie, 83, 125, 227, 358  
 minimum globalne, 362, 369  
 minimum lokalne, 362, 369  
 OLS, 283  
 regularyzowana, 85  
 sumy kwadratów błędów, 124  
 kryterialna, 130  
 krzywej nauczania, 187  
 logistyczna, 387, 388, 389, 393, *Patrz też:*  
 funkcja sigmoidalna  
 uogólnienie, 390  
 znormalizowana, 391  
 logitowa, 76  
 mapowania, 92, 93  
 masy prawdopodobieństwa, 205, 206, 214  
 nagrody, 29  
 odcinkowo liniowa, 393  
 pobudzenia, 388  
 podobieństwa, 94  
 radialna bazowa, 94  
 RBF, *Patrz:* jądro radialnej funkcji bazowej  
 sigmoidalna, 77, 341, 342, 387, 388, 393  
 Signum, 393  
 skokowa *Heaviside'a*, 43, *Patrz też:* funkcja skoku jednostkowego  
 skoku jednostkowego, 43, 44, 55, 393  
 s-ksztaltna, *Patrz:* funkcja sigmoidalna  
 softmax, 390, 391, 396, 397  
 tangensa hiperbolicznego, 388, 391, 393, 396  
 ujemnego logarytmu wiarygodności, 387  
 wiarygodności, 79, 80, 83  
 wykładnicza znormalizowana, *Patrz:* funkcja softmax  
 Fusiello Andrea, 289  
 fuzyfikator, *Patrz:* blok rozmywania

fuzziness coefficient, *Patrz*: blok rozmywania  
fuzzy c-means, *Patrz*: algorytm rozmytych  
c-srednich

## G

Galton Francis, 29  
GIL, 376  
Gini impurity, *Patrz*: wskaźnik Giniego  
Global Interpreter Lock, *Patrz*: GIL  
głosowanie  
    większościowe, 204, 213, 215, 221  
    implementacja, 207, 209  
    ze względną większością głosów, 204  
Google, 336  
    silnik wyszukiwania obrazów, 336  
    Tłumacz, *Patrz*: Tłumacz Google  
GPU, 373, 375, 380  
    cena, 377  
    liczba zmiennoprzecinkowa, 380  
gradient

    prosty, *Patrz*: algorytm gradientu prostego  
    sprawdzanie, 363, 364, 365  
        implementacja, 365, 366  
        koszt, 368  
    zanikający, 339  
gradient checking, *Patrz*: gradient sprawdzanie  
gradient descent, *Patrz*: algorytm gradientu  
    prostego  
graf wyrażeń symbolicznych, 378  
Graphical Processing Units, *Patrz*: GPU  
Graphviz, 102  
grid search, *Patrz*: algorytm przeszukiwania siatki  
grubokońcowość, 346  
grupowanie, *Patrz*: analiza skupień, klasteryzacja  
GUI, 394, 395

## H

hard clustering, *Patrz*: analiza skupień twarda  
Harrison David, 277  
heat map, *Patrz*: mapa cieplna  
hierarchical clustering, *Patrz*: klasteryzacja  
    hierarchiczna  
Hinton Geoffrey, 336  
hiperparametr, 35, 181, 191, 339  
    adaptacyjnego neuronu liniowego, 59  
    implementacji perceptronu, 59  
    lasu losowego, 105  
    strojenie, 191, 192, 193

hiperplaszczyzna  
    liniowa, 92  
    negatywna, 87, 88  
    odległość, 88, *Patrz też*: margines  
    pozytywna, 87, 88  
hiperprzestrzeń, 87  
histogram, 31  
Hoff Tedd, 54  
holdout cross-validation, *Patrz*: algorytm  
    wydzielania  
hyperbolic tangent, *Patrz*: funkcja tangensa  
    hiperbolicznego

## I

IG, *Patrz*: przyrost informacji  
iloczyn  
    macierzowo-wektorowy, 59, 343, 363  
    skalarny  
        wektorów, *Patrz*: wektor iloczyn skalarny  
    iloraz szans, 76  
imputacja z użyciem średniej, 114  
information gain, *Patrz*: przyrost informacji  
inlier, *Patrz*: dane nieodstające  
instance-based learning, *Patrz*: uczenie  
    maszynowe z przykładów  
interpolacja, 114

## J

jadro  
    gaussowskie, 94, 162  
    radialnej funkcji bazowej, 94, 162, 165  
        implementacja, 164, 165  
RBF, *Patrz*: jadro radialnej funkcji bazowej  
SVM, 91, 94  
tangensa hiperbolicznego, *Patrz*: funkcja jądra  
    sigmoidalna  
jednostka  
    aktywacji, 339  
    długiej pamięci krótkotrwałej, *Patrz*: LSTM  
    obciążenia, 339  
    ukryta, 358  
język  
    HTML, 258  
    interpretowany, 36  
mandaryński, 336  
naturalny przetwarzanie, *Patrz*: analiza  
    sentymentów  
Python, *Patrz*: Python

**K**

- Kanaris Ioannis, 239  
 kara, 89, 125  
 kaskadowy arkusz stylów, *Patrz:* CSS  
 kernel PCA, *Patrz:* algorytm PCA jądrowy  
 kernel trick, *Patrz:* funkcja jądra sztuczka  
 kernelizacja, 91  
 k-krotny sprawdzian krzyżowy, *Patrz:* algorytm k-krotnego sprawdzianu krzyżowego  
 klasa  
   BaggingClassifier, 223  
   BaseEstimator, 211  
   ClassifierMixin, 211  
   CountVectorizer, 238, 239, 246, 248  
   DataFrame, 236  
   ElasticNet, 295  
   etykieta, *Patrz:* etykieta klasy  
   GridSearchCV, 198  
   HashingVectorizer, 248, 249  
   Imputer, 114  
   LabelEncoder, 118, 178, 211  
   LDA, 156  
   LogisticRegression, 85  
   MajorityVoteClassifier, 213, 221  
   model.LogisticRegression, 81  
   negatywna, 28, 50  
   nierozdzielna liniowo, 46, 75, 76, 89, 91, 92, 159  
   OneHotEncoder, 119  
   PCA, 147, 149  
   Perceptron, 72, 90  
   Perceptron i LogisticRegression, 90  
   Pipeline, 178  
   PolynomialFeatures, 296  
   PorterStemmer, 243  
   pozytywna, 28, 50  
   RandomizedSearchCV, 193  
   rozdzielna liniowo, 44, 46, 54, 70, 76  
   Sequential, 396  
   SGDClassifier, 91, 247  
   silhouette\_samples, 317  
   silhouette\_scores, 317  
   sklearn.ensemble.VotingClassifier, 213  
   StandardScaler, 72, 123  
   StratifiedKFold, 200  
   SVC, 90  
   TextAreaField, 260  
   TfidfTransformer, 240, 241, 246  
   TfidfVectorizer, 246, 248  
   transformująca, 114  
 klaster, 30  
   bezwładność, 310  
   spójność, 316  
 klasteryzacja, 30, 307, 308, *Patrz też:* analiza skupień  
   bazująca na prototypach, 308, 333  
   hierarchiczna, 320  
   rozmyta, 313  
 klasyfikacja, 27  
   binarna, 27, 28, 42, 76, 204, 227, 337  
   rozszerszania, 50  
   dokładność, 35, 73  
   marginesu, *Patrz:* margines klasyfikacja miękkiego marginesu, 88  
   wieloklasowa, 28, 72, 357  
   wielowymiarowa, 50  
 klasyfikator, *Patrz też:* algorytm bayesowski naiwny, 247  
   leniwy, 106  
   łączenie, 203  
   silny, 104  
   słaby, 104, 226, 227  
   kłatywa wymiarowości, 109, 132, 332  
   ograniczanie, 140, 150  
 k-nearest neighbor classifier, *Patrz:* algorytm KNN  
   kodowanie gorącojedynkowe, 119, 340, 341  
 Kohavi Ron, 183  
 kolejka, 179, 215  
   maszyny wektorów nośnych, 192  
 kontaminacja modelu, *Patrz:* algorytm kontaminacyjny  
 konwolucja, 370, *Patrz też:* sieć neuronowa splotowa  
 korelacja, 280  
 kowariancja, 160, 280, 281  
   macierz, *Patrz:* macierz kowariancji  
   wartość ujemna, 142  
 kroswalidacja 5x2, 193  
 krotka, 179  
 krzywa  
   charakterystyki roboczej odbiornika,  
     *Patrz:* krzywa ROC  
   precyzji-pełności, 198  
   ROC, 198, 199, 200, 211, 213, 220  
   sigmoidalna, 77  
   uczenia, 73, 186, 187, 188  
   walidacji, 186, 187, 188, 189, 190  
 kwantyzator, 55  
 kwiat kosańca, 32

## L

las losowy, 104, 105, 134, 135, 221  
hiperparametr, 105  
regresywny, 300, 301, 302, 303  
przetrenowanie, 303  
skuteczność, 302  
latent Dirichlet allocation, *Patrz:* algorytm alokacji ukrytej zmiennej Dirichleta  
lazy learner, *Patrz:* klasyfikator leniwy  
Least Absolute Shrinkage and Selection Operator, *Patrz:* algorytm LASSO  
leave-one-out method, *Patrz:* algorytm LOO  
LeCun Yann, 344, 370  
lemat, 244  
lematyzacja, 244  
liczba  
magiczna, 345  
zmiennoprzecinkowa, 380  
linear discriminant analysis, *Patrz:* algorytm LDA  
linia regresji, 276  
liniowa dyskryminacja Fishera, *Patrz:* algorytm dyskryminacji liniowej Fishera  
linkage matrix, *Patrz:* macierz wiązania  
LinkedIn, 256  
logistic regression, *Patrz:* algorytm regresji logistycznej  
Long Short Term Memory unit, *Patrz:* LSTM  
losowanie  
bez zwracania, 182, 226  
ze zwracaniem, 182, 221  
LSTM, 373

## M

machine learning, *Patrz:* uczenie maszynowe  
macierz, 33  
dwuwymiarowa, 378  
hermitowska, 143  
Jacobiego, 363  
korelacji, 280  
kowariancji, 141, 142, 151, 153, 161, 280  
odległości, 323, 324  
podobieństwa, 333  
pomyłek, 195, 196  
rozproszenia, 151, 152, 153, 278, 280  
rzutowania, 141, 145, 171  
symetryczna, 142, 143  
trójwymiarowa, 378  
wiązania, 323, 325

MAD, 289  
magic number, *Patrz:* liczba magiczna  
majority voting, *Patrz:* głosowanie większościowe  
makrouśrednianie, 201, 202  
mapa  
cech, 371  
cieplna, 282, 326, 327  
mapowanie, 117  
margines, 87, 88  
Markatou Marianthi, 185  
Martinez Aleix, 150  
Maslow Abraham, 35  
maszyna wektorów nośnych, 85, 87, 90, 91, 93, 107, 122, 158, 194, 204, 305, 363, 369  
jądro, *Patrz:* jądro SVM  
kolejka, 192  
programowanie kwadratowe, 93  
trenowanie, 89  
McCulloch Warren, 42, 336  
McCulloch-Pitts neuron, *Patrz:* neuron MCP  
mean imputation, *Patrz:* interpolacja imputacja z użyciem średniej  
mean squared error, *Patrz:* MSE  
Median Absolute Deviation, *Patrz:* MAD  
medoid, 308  
metaklasyfikator, 203  
metoda, *Patrz też:* algorytm, klasyfikator  
aglomeracyjna, 320  
deglomeracyjna, 320  
łokcia, 308, 315  
średnich połączeń, 321  
transform, 72  
Warda, 321  
zespołowa, 203, 204, 213, 302  
implementacja, 207, 209  
skuteczność, 205  
stopa błędu, *Patrz:* stopa błędu strojenie, 216, 219, 220  
usprawnianie, 226  
waga, 207, 208, 209  
złożoność obliczeniowa, 232  
metryka, 35, 195  
odległości, 309  
odsetek  
falszywie pozytywnych, 197  
prawdziwie pozytywnych, 197  
pełność, 197  
precyzja, 197  
skuteczności, 197

Microsoft, 336  
 mikrośrodowisko, 257  
 mikrouśrednianie, 201, 202  
 mini-batch learning, *Patrz:* uczenie maszynowe z pomocą minipaczek  
 model, *Patrz:* algorytm  
 modelowanie predyktacyjne, 33  
 MSE, 292, 301  
 multi-layer feedforward neural network, *Patrz:* sieć neuronowa wielowarstwowa jednokierunkowa  
 multi-layer perceptron, *Patrz:* algorytm MLP  
 multiple linear regression, *Patrz:* regresja liniowa wielowymiarowa

**N**

nagroda, 29, 30  
 naïve Bayes classifier, *Patrz:* klasyfikator bayesowski naiwny  
 NaN, *Patrz:* dane:NaN  
 natural language processing, *Patrz:* analiza sentymentów  
 neuron  
     liniowy adaptacyjny, 54, *Patrz też:* algorytm Adaline  
     McCullocha-Pittsa, *Patrz:* neuron MCP  
     MCP, 42, 44, 336  
 n-gram, 239  
 NLP, *Patrz:* analiza sentymentów  
 noise point, *Patrz:* punkt zaszumienia  
 normalizacja, 122, 123  
 notacja  
     gorącojedynkowa, 119, 340, 341  
     macierzowa, 32  
     wektorowa, 32

**O**

obciążenie, 84, 89, 180, 186, 226  
 szacowanie błędu, 193  
 odchylenie bezwzględne średnie, *Patrz:* MAD  
 odds ratio, *Patrz:* iloraz szans  
 one-hot encoding, *Patrz:* kodowanie gorącojedynkowe  
 online learning, *Patrz:* uczenie maszynowe przyrostowe  
 opinion mining, *Patrz:* analiza sentymentów  
 optymalizacja hiperparametryczna, 35  
 ordinary least squares, *Patrz:* algorytm OLS

outlier, *Patrz:* próbka odstająca  
 overfitting, *Patrz:* dopasowanie nadmierne

**P**

pakiet, *Patrz:* biblioteka  
 Pearson product-moment correlation coefficient, *Patrz:* współczynnik korelacji liniowej Pearsona  
 Pinterest, 256  
 pismo odręczne  
     rozpoznawanie, *Patrz:* rozpoznawanie pisma odręcznego  
 Pitts Walter, 42, 336  
 plik  
     app.py, 257, 261, 264, 265  
     style.css, 260  
     vectorizer.py, 253  
 plurality voting, *Patrz:* głosowanie ze względnią większością głosów  
 podobieństwo, 309, 333  
 pole tekstowe, 258, 260  
 Pollock Stephen, 288  
 Porter Martin, 243  
 prawdopodobieństwo, 390  
     szacowanie, 79, 81  
 preprocessing, *Patrz:* dane przetwarzanie wstępne  
 principal component analysis, *Patrz:* algorytm PCA  
 procesor  
     główny, *Patrz:* CPU  
     graficzny, *Patrz:* GPU  
     wybór, 380  
 program marketingowy, 30  
 propagacja  
     w przód, 340, 341, 359  
     wsteczna, *Patrz:* algorytm propagacji wstecznej  
 protokół SSH, 270  
 prototype-based clustering, *Patrz:* klasteryzacja oparta na prototypach  
 próbka  
     nieodstająca, 288  
     odstająca, 280, 288, 290  
     początkowa, 221, 223  
     rdzeniowa, *Patrz:* punkt rdzeniowy, *Patrz:* punkt rdzeniowy  
     współczynnik profilu, *Patrz:* współczynnik profilu  
 próbkiwanie, *Patrz:* losowanie przeszukiwanie losowe, 193

przetrenowanie, *Patrz*: dopasowanie nadmierne przyrost informacji, 97

maksymalizowanie, 98

punkt

graniczny, 328

rdzeniowy, 328

zaszumienia, 328

Python

Anaconda, *Patrz*: Anaconda

globalna blokada interpretera, 376

instalacja, 36, 37

PythonAnywhere, 269

błedy, 271

konto, 270

panel sterowania, 270

## R

radial basis function, *Patrz*: jądro radialnej funkcji bazowej

Radial Basis Function kernel, *Patrz*: jądro radialnej funkcji bazowej

random forest, *Patrz*: las losowy

RANDom SAmple Consensus, *Patrz*: algorytm RANSAC

randomized search, *Patrz*: przeszukiwanie losowe

Rao Calyampudi Radhakrishna, 150

raw term frequency, *Patrz*: term częstość

RBF kernel, *Patrz*: jądro radialnej funkcji bazowej

receiver operating characteristic, *Patrz*: krzywa ROC

recurrent neural networks, *Patrz*: sieć

neuronowa rekurencyjna

redukcja wymiarowości, 31, 34, 109, 124, 129, 140, 150

nieliniowa, 31

odkrywanie cech, 129

wybór cech, 129

regresja, 27, 28, 29

grzbietowa, 294, 295

liniowa, 29, 276, 286, 296, 299, 301, 363, 384, 385

implementacja, 282

OLS, *Patrz*: algorytm OLS

prosta, 276, 277

przesunięcie, 276, 282, 289, 291, 303

regularyzowana, 294

skuteczność, 285, 291, 292

uczenie, 280

wartość resztowa, 276, 291

wielowymiarowa, 277, 291

logistyczna, *Patrz*: algorytm regresji

logistycznej

przy użyciu drzewa decyzyjnego, 301

przy użyciu lasu losowego, *Patrz*: las losowy regresywny

wielomianowa, 296, 299, 301

regular expression, *Patrz*: wyrażenie regularne regularyzacja, 81, 84, 85, 89, 109, 124, 125, 157, 189, 294, 356, 357

L1, 124, 246

implementacja, 127

rzadkie wektory cech, 124, 125, 127, 128 L2, 84, 124, 246, 294, 352, *Patrz też*: rozpad wag

ścieżka, *Patrz*: ścieżka regularyzacji

regulator, 29, 30

reguła

łańcuchowa, 362

Rosenblatta, *Patrz*: reguła uczenia perceptronu uczenia perceptronu, 42, 44

Widrowa-Hoffa, 55, *Patrz też*: algorytm Adaline reinforcement learning, *Patrz*: uczenie maszynowe przez wzmacnianie

RNN, *Patrz*: sieć neuronowa rekurencyjna

Ronacher Armin, 256

Rosenblatt Frank, 42

rozkład normalny standardowy, 34

rozpad wag, 84, *Patrz też*: regularyzacja L2 rozpoznawanie opinii, *Patrz*: analiza sentymentów osób, 336

pisma odręcznego, 28, 343, 346, 348

rozróżnialność, 155

równanie normalne, 288

różniczkowanie automatyczne, 362

Rubenfeld Daniel, 277

Rumelhart David, 336

## S

scatterplot matrix, *Patrz*: macierz rozproszenia

Schapire Robert, 226

Schölkopf Bernhard, 161

sentiment analysis, *Patrz*: analiza sentymentów

Sequential Backward Selection, *Patrz*: algorytm SBS

serializacja, 252, 253, 271

sfera Gaussa, 94

sieć neuronowa

ADALINE, 54

implementacja, 373

- jednowarstwowa, 54, 337, 338  
 konwolucyjna, *Patrz:* sieć neuronowa splotowa  
 rekurencyjna, 342, 371, 372  
 splotowa, 370  
     warstwa buforowa, 371  
     warstwa detektora cech, 370, 371  
     warstwa podpróbkowania, 371  
     warstwa splotowa, 371  
 sztuczna, 335, 336  
     implementacja, 363  
     jednokierunkowa, 342  
     trenowanie, 356  
     usuwanie błędów, 363, 364, 365, 368  
     zbieżność, 368  
 teoria, 343  
 wielowarstwowa, 336, 340, 343, 369, 387  
     jednokierunkowa, 338  
 silhouette analysis, *Patrz:* analiza profilu SIMD, 49  
 Simon Richard, 193  
 simple linear regression, *Patrz:* regresja liniowa prosta  
 Single Instruction, Multiple Data, *Patrz:* SIMD  
 single linkage, *Patrz:* algorytm wiązania pojedynczego  
 skupienie, 30  
 slack variable, *Patrz:* zmienna uzupełniająca słownik, 238  
     odwrotnego mapowania, 117  
 soft clustering, *Patrz:* analiza skupień miękka  
 soft-margin classification, *Patrz:* klasyfikacja miękkiego marginesu  
 softmax function, *Patrz:* funkcja softmax spam, 251  
     filtrowanie, 27, 239, 247  
 spectral clustering algorithm, *Patrz:* algorytm klasteryzacji spektralnej splatanie, 370  
 SQLite Manager, 256  
 SSE, 55, 57, 79, 124, 283, 292, 311, 337, 385  
 stacking, *Patrz:* algorytm kontaminacyjny standaryzacja, 60, 72, 85, 122, 123, 141, 142, 179, 215, 217, 281  
 stochastic gradient descent, *Patrz:* algorytm stochastycznego spadku wzduż gradientu stopa błędu, 205  
 stop-word removal, *Patrz:* tekst słowo pomijane strong learner, *Patrz:* klasyfikator silny Sum of Squared Errors, *Patrz:* SSE suma kwadratów błędów, *Patrz:* SSE supervised learning, *Patrz:* uczenie maszynowe nadzorowane support vector machine, *Patrz:* maszyna wektorów nośnych SVM, *Patrz:* maszyna wektorów nośnych sygnał nagrody, 29, 30 wyjściowy, *Patrz też:* wynik ciągły, 27, 28 symbol, 379 system polecający, 236, 251, 262 szachy, 30 szansa, 76, *Patrz też:* zdarzenie pozytywne iloraz, *Patrz:* iloraz szans sztuczna inteligencja, 26, 42 szum, 70 filtrowanie, 84
- ## Ś
- ścieżka regularyzacji, 128 środowisko C/C++, 378 CUDA/OpenCL, 378 Flask, *Patrz:* Flask
- ## T
- tabela DataFrame, 112, 116, 245, 322 tablica danych tekstowych, 238 NumPy, 344 technika OvA, *Patrz:* technika OvR OvR, 50, 72, 76, 127, 340 tekst analiza, 239 biegunowość, 235 forma kanoniczna, 244 lematyzacja, *Patrz:* lematyzacja oczyszczenie, 241 emotikony, 242 znacznik HTML, 242 znak interpunkcyjny, 242 postać numeryczna, 237, 238 przesyłanie strumieniowe, 247, 248 przetwarzanie na znaczniki, 243 rdzeniowanie wyrazów, 243, 244 słowo pomijane, 244

tensor, 378  
term, 239, 240  
    częstość, 239, 241  
    ważenie, 239  
    znormalizowana, 240, 244  
wartość  
    niezerowa, 240  
    term frequency — inverse document  
        frequency, *Patrz*: algorytm tf-idf  
test statystyczny jednocyfrowy, 134  
tf-idf, *Patrz*: algorytm tf-idf  
Tłumacz Google, 336  
token, 237  
tokenizacja, 243, 246, 247, 249  
Toldo Robert, 289  
training data, *Patrz*: dane uczące  
transpozycja macierzy, 43  
twierdzenie Wolperta, 35

## U

uczenie maszynowe, 25  
    głębokie, 335, 336, 339, 376  
        w przemyśle farmaceutycznym, 337  
    historia, 42  
    leniwe, *Patrz*: klasyfikator leniwy  
    nadzorowane, 26, 27, 29, 30, 42, 115, 275  
        klasyfikacja, *Patrz*: klasyfikacja  
    nienadzorowane, 26, 307  
        redukcja wymiarowości, *Patrz*: redukcja  
            wymiarowości  
    nieparametryczne, 107  
    parametryczne, 107  
    pozadzeniowe, 247, 248  
        implementacja, 249  
    przez wzmacnianie, 26, 29, 30  
    przyrostowe, 63, 91, 368  
    z pomocą minipaczek, 63  
    z przykładów, 107  
underfitting, *Patrz*: dopasowanie zbyt małe  
unsupervised learning, *Patrz*: uczenie maszynowe  
    nienadzorowane  
usługa PythonAnywhere, *Patrz*: PythonAnywhere  
uśrednianie mikroskopowe, 201, 202

## V

vanishing gradient, *Patrz*: gradient zanikający  
Vapnik Vladimir, 88  
Varma Sudhir, 193

## W

Ward's linkage, *Patrz*: metoda Warda  
wariancja, 84, 89, 124, 141, 143, 180, 186, 226,  
    *Patrz też*: dopasowanie nadmierne  
    jednostkowa, 142  
    maksymalizacja, 145  
    wewnętrzowa, 301  
    wyjaśniona współczynnik, *Patrz*: współczynnik  
        wariancji wyjaśnionej  
wartość  
    kryterialna, 130  
    średnia, *Patrz*: centroid  
    własna, 141, 142, 143, 154, 161  
ważenie częstości termów — odwrotna częstość  
    w tekście, *Patrz*: algorytm tf-idf  
weak learner, *Patrz*: klasyfikator słaby, *Patrz*:  
    klasyfikator słaby  
wektor, 33  
    częstotliwości węzła, 214  
element, 33  
iloczyn skalarny, 43, 59  
nośny, 87, 90  
rzadki, 238  
średnich, 151, 152  
własny, 141, 142, 143, 155, 161  
    macierzy podobieństwa, 333  
Widrow Bernard, 54  
wielowątkowość, 376, 377  
wiersz poleceń, 269  
Williams Ronald, 336  
within-cluster sum of squared errors, *Patrz*:  
    algorytm minimalizacji wewnętrzgrupowej

- wariancji wyjaśnionej, 143, 144  
zmiennej objaśniającej, *Patrz*  
**wykres**  
krzywej walidacji, 190  
profilu, 308  
punktowy, 31  
resztowy, 292  
ROC, 198, 199, 200  
współczynników  
profilu, 317, 318, 320  
wariancji wyjaśnionej, 143, 144  
**wynik**, *Patrz*: sygnał wyjściowy  
F1, 197  
fałszywie negatywny, 195, 201  
fałszywie pozytywny, 195, 197, 201  
prawdziwie negatywny, 195, 201  
prawdziwie pozytywny, 195, 197, 201  
przewidywanie, 275  
**wyrażenie**  
matematyczne, 377  
regularne, 242  
symboliczne, 378  
wyszukiwarka, 251, 308  
wzmacnianie, 226, 227  
adaptacyjne, 226
- Z**
- zasada**  
większościowego głosowania, *Patrz*: głosowanie  
większościowe  
**zdarzenie pozytywne**, 76  
**zmienna**  
Dirichleta, 250  
docelowa, 276, 278  
objaśniająca, 28, 29, 276, 278, 291  
współczynnik, *Patrz*: współczynnik  
zmiennej objaśniającej  
objaśniana, 28, 29  
prognozowana, *Patrz*: zmienna objaśniana  
prognozująca, *Patrz*: zmienna objaśniająca  
typ, 379  
uzupełniająca, 88

# PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW  
w działający bankomat!

**Dowiedz się więcej i dołącz już dzisiaj!**

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA

 Helion SA