

Sebastian Raschka
Vahid Mirjalili

Python

Uczenie
maszynowe

Wydanie 2

Helion 

Packt 

Tytuł oryginału: Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow, 2nd Edition

Tłumaczenie: Krzysztof Sawka

ISBN: 978-83-283-5121-9

Copyright © Packt Publishing 2017. First published in the English language under the title 'Python Machine Learning - Second Edition – (9781787125933)'.

Polish edition copyright © 2019 by Helion SA

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 jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie 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 właścicieli.

Autor oraz Helion SA 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 Helion SA nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion SA

ul. Kościuszki 1c, 44-100 GLIWICE

tel. 32 231 22 19, 32 230 98 63

e-mail: 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/pythu2>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

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

Spis treści

Informacje o autorach	11
Informacje o recenzentach	13
Wstęp	15
Rozdział 1. Umożliwianie komputerom uczenia się z danych	23
Tworzenie inteligentnych maszyn służących do przekształcania danych w wiedzę	24
Trzy różne rodzaje uczenia maszynowego	24
Prognozowanie przyszłości za pomocą uczenia nadzorowanego	25
Rozwiązywanie problemów interaktywnych za pomocą uczenia przez wzmacnianie	28
Odkrywanie ukrytych struktur za pomocą uczenia nienadzorowanego	29
Wprowadzenie do podstawowej terminologii i notacji	30
Strategia tworzenia systemów uczenia maszynowego	32
Wstępne przetwarzanie — nadawanie danym formy	32
Trenowanie i dobór modelu predykcyjnego	34
Ewaluacja modeli i przewidywanie wystąpienia nieznanych danych	34
Wykorzystywanie środowiska Python do uczenia maszynowego	35
Instalacja środowiska Python i pakietów z repozytorium Python Package Index	35
Korzystanie z platformy Anaconda i menedżera pakietów	36
Pakiety przeznaczone do obliczeń naukowych, analizy danych i uczenia maszynowego	36
Podsumowanie	37
Rozdział 2. Trenowanie prostych algorytmów uczenia maszynowego w celach klasyfikacji	39
Sztuczne neurony — rys historyczny początków uczenia maszynowego	40
Formalna definicja sztucznego neuronu	41
Reguła uczenia perceptronu	43
Implementacja algorytmu uczenia perceptronu w Pythonie	45
Obiektowy interfejs API perceptronu	45
Trenowanie modelu perceptronu na zestawie danych Iris	48

Adaptacyjne neurony liniowe i zbieżność uczenia	53
Minimalizacja funkcji kosztu za pomocą metody gradientu prostego	55
Implementacja algorytmu Adaline w Pythonie	56
Usprawnianie gradientu prostego poprzez skalowanie cech	60
Wielkoskalowe uczenie maszynowe i metoda stochastycznego spadku wzdłuż gradientu	62
Podsumowanie	66
Rozdział 3. Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn	67
Wybór algorytmu klasyfikującego	68
Pierwsze kroki z biblioteką scikit-learn — uczenie perceptronu	68
Modelowanie prawdopodobieństwa przynależności do klasy za pomocą regresji logistycznej	74
Teoretyczne podłoże regresji logistycznej i prawdopodobieństwa warunkowego	74
Wyznaczanie wag logistycznej funkcji kosztu	78
Przekształcanie implementacji Adaline do postaci algorytmu regresji logistycznej	80
Uczenie modelu regresji logistycznej za pomocą biblioteki scikit-learn	84
Zapobieganie przetrenowaniu za pomocą regularyzacji	86
Wyznaczanie maksymalnego marginesu za pomocą maszyn wektorów nośnych	88
Teoretyczne podłoże maksymalnego marginesu	89
Rozwiązywanie przypadków nieliniowo rozdzielnych za pomocą zmiennych uzupełniających	90
Alternatywne implementacje w interfejsie scikit-learn	92
Rozwiązywanie nieliniowych problemów za pomocą jądra SVM	93
Metody jądrowe dla danych nierozdzielnych liniowo	93
Stosowanie sztuczki z funkcją jądra do znajdowania przestrzeni rozdzielających w przestrzeni wielowymiarowej	95
Uczenie drzew decyzyjnych	99
Maksymalizowanie przyrostu informacji — osiąganie jak największych korzyści	100
Budowanie drzewa decyzyjnego	103
Łączenie wielu drzew decyzyjnych za pomocą modelu losowego lasu	107
Algorytm k-najbliższych sąsiadów — model leniwego uczenia	109
Podsumowanie	113
Rozdział 4. Tworzenie dobrych zbiorów uczących — wstępne przetwarzanie danych	115
Kwestia brakujących danych	115
Wykrywanie brakujących wartości w danych tabelarycznych	116
Usuwanie próbek lub cech niezawierających wartości	117
Wstawianie brakujących danych	118
Estymatory interfejsu scikit-learn	119
Przetwarzanie danych kategoryzujących	119
Cechy nominalne i porządkowe	120
Tworzenie przykładowego zestawu danych	120
Mapowanie cech porządkowych	121
Kodowanie etykiet klas	121
Kodowanie „gorącojedynkowe” cech nominalnych (z użyciem wektorów własnych)	122
Rozdzielanie zestawu danych na oddzielne podzbiory uczące i testowe	124
Skalowanie cech	127
Dobór odpowiednich cech	129
Regularyzacje L1 i L2 jako kary ograniczające złożoność modelu	129
Interpretacja geometryczna regularyzacji L2	130

Rozwiązania rzadkie za pomocą regularyzacji L1	131
Algorytmy sekwencyjnego wyboru cech	135
Ocenianie istotności cech za pomocą algorytmu losowego lasu	140
Podsumowanie	142
Rozdział 5. Kompresja danych poprzez redukcję wymiarowości	143
Nienadzorowana redukcja wymiarowości za pomocą analizy głównych składowych	144
Podstawowe etapy analizy głównych składowych	144
Wydobywanie głównych składowych krok po kroku	146
Wyjaśniona wariancja całkowita	148
Transformacja cech	149
Analiza głównych składowych w interfejsie scikit-learn	152
Nadzorowana kompresja danych za pomocą liniowej analizy dyskryminacyjnej	154
Porównanie analizy głównych składowych z liniową analizą dyskryminacyjną	155
Wewnętrzne mechanizmy działania liniowej analizy dyskryminacyjnej	156
Obliczanie macierzy rozproszenia	157
Dobór dyskryminant liniowych dla nowej podprzestrzeni cech	159
Rzutowanie próbek na nową przestrzeń cech	161
Implementacja analizy LDA w bibliotece scikit-learn	161
Jądrowa analiza głównych składowych jako metoda odwzorowywania nierozdzielnych liniowo klas	163
Funkcje jądra oraz sztuczka z funkcją jądra	164
Implementacja jądrowej analizy głównych składowych w Pythonie	168
Rzutowanie nowych punktów danych	175
Algorytm jądrowej analizy głównych składowych w bibliotece scikit-learn	178
Podsumowanie	179
Rozdział 6. Najlepsze metody oceny modelu i strojenie parametryczne	181
Usprawnianie cyklu pracy za pomocą kolejkowania	181
Wczytanie zestawu danych Breast Cancer Wisconsin	182
Łączenie funkcji transformujących i estymatorów w kolejce czynności	183
Stosowanie k-krotnego sprawdzianu krzyżowego w ocenie skuteczności modelu	184
Metoda wydzielania	185
K-krotny sprawdzian krzyżowy	186
Sprawdzanie algorytmów za pomocą krzywych uczenia i krzywych walidacji	190
Diagnozowanie problemów z obciążeniem i wariancją za pomocą krzywych uczenia	190
Rozwiązywanie problemów przetrenowania i niedotrenowania za pomocą krzywych walidacji	193
Dostrajanie modeli uczenia maszynowego za pomocą metody przeszukiwania siatki	195
Strojenie hiperparametrów przy użyciu metody przeszukiwania siatki	195
Dobór algorytmu poprzez zagnieżdżony sprawdzian krzyżowy	196
Przegląd metryk oceny skuteczności	198
Odczytywanie macierzy pomyłek	198
Optymalizacja precyzji i pełności modelu klasyfikującego	200
Wykres krzywej ROC	202
Metryki zliczające dla klasyfikacji wieloklasowej	204
Kwestia dysproporcji klas	205
Podsumowanie	208

Rozdział 7. Łączenie różnych modeli w celu uczenia zespołowego	209
Uczenie zespołów	209
Łączenie klasyfikatorów za pomocą algorytmu głosowania większościowego	213
Implementacja prostego klasyfikatora głosowania większościowego	214
Stosowanie reguły głosowania większościowego do uzyskiwania prognoz	219
Ewaluacja i strojenie klasyfikatora zespołowego	221
Agregacja — tworzenie zespołu klasyfikatorów za pomocą próbek początkowych	226
Agregacja w pigułce	227
Stosowanie agregacji do klasyfikowania przykładów z zestawu Wine	228
Usprawnianie słabych klasyfikatorów za pomocą wzmocnienia adaptacyjnego	231
Wzmacnianie — mechanizm działania	232
Stosowanie algorytmu AdaBoost za pomocą biblioteki scikit-learn	236
Podsumowanie	239
Rozdział 8. Wykorzystywanie uczenia maszynowego w analizie sentymentów	241
Przygotowywanie zestawu danych IMDb movie review do przetwarzania tekstu	242
Uzyskiwanie zestawu danych IMDb	242
Przetwarzanie wstępne zestawu danych IMDb do wygodniejszego formatu	243
Wprowadzenie do modelu worka słów	244
Przekształcanie słów w wektory cech	245
Ocena istotności wyrazów za pomocą ważenia częstości termów	
— odwrotnej częstości w tekście	246
Oczyszczanie danych tekstowych	248
Przetwarzanie tekstu na znaczniki	249
Uczenie modelu regresji logistycznej w celu klasyfikowania tekstu	251
Praca z większą ilością danych — algorytmy sieciowe i uczenie pozardzeniowe	253
Modelowanie tematyczne za pomocą alokacji ukrytej zmiennej Dirichleta	256
Rozkładanie dokumentów tekstowych za pomocą analizy LDA	257
Analiza LDA w bibliotece scikit-learn	258
Podsumowanie	261
Rozdział 9. Wdrażanie modelu uczenia maszynowego do aplikacji sieciowej	263
Serializacja wyuczonych estymatorów biblioteki scikit-learn	264
Konfigurowanie bazy danych SQLite	266
Tworzenie aplikacji sieciowej za pomocą środowiska Flask	269
Nasza pierwsza aplikacja sieciowa	269
Sprawdzanie i wyświetlanie formularza	271
Przekształcanie klasyfikatora recenzji w aplikację sieciową	275
Pliki i katalogi — wygląd drzewa katalogów	277
Implementacja głównej części programu w pliku app.py	277
Konfigurowanie formularza recenzji	280
Tworzenie szablonu strony wynikowej	281
Umieszczanie aplikacji sieciowej na publicznym serwerze	282
Tworzenie konta w serwisie PythonAnywhere	283
Przesyłanie aplikacji klasyfikatora filmowego	283
Aktualizowanie klasyfikatora recenzji filmowych	284
Podsumowanie	286

Rozdział 10. Przewidywanie ciągłych zmiennych docelowych za pomocą analizy regresywnej	287
Wprowadzenie do regresji liniowej	288
Prosta regresja liniowa	288
Wielowymiarowa regresja liniowa	288
Zestaw danych Housing	290
Wczytywanie zestawu danych Housing do obiektu DataFrame	290
Wizualizowanie ważnych elementów zestawu danych	292
Analiza związków za pomocą macierzy korelacji	293
Implementacja modelu regresji liniowej wykorzystującego zwykłą metodę najmniejszych kwadratów	296
Określanie parametrów regresyjnych za pomocą metody gradientu prostego	296
Szacowanie współczynnika modelu regresji za pomocą biblioteki scikit-learn	300
Uczenie odpornego modelu regresyjnego za pomocą algorytmu RANSAC	301
Ocenianie skuteczności modeli regresji liniowej	304
Stosowanie regularyzowanych metod regresji	307
Przekształcanie modelu regresji liniowej w krzywą — regresja wielomianowa	308
Dodawanie członów wielomianowych za pomocą biblioteki scikit-learn	309
Modelowanie nieliniowych zależności w zestawie danych Housing	310
Analiza nieliniowych relacji za pomocą algorytmu losowego lasu	314
Podsumowanie	318
Rozdział 11. Praca z nieoznakowanymi danymi — analiza skupień	319
Grupowanie obiektów na podstawie podobieństwa przy użyciu algorytmu centroidów	320
Algorytm centroidów w bibliotece scikit-learn	320
Inteligentniejszy sposób dobierania pierwotnych centroidów za pomocą algorytmu k-means++	324
Klasteryzacja twarda i miękka	325
Stosowanie metody łockia do wyszukiwania optymalnej liczby skupień	327
Ujęcie ilościowe jakości klasteryzacji za pomocą wykresu profilu	328
Organizowanie skupień do postaci drzewa klastrow	333
Oddolne grupowanie skupień	333
Przeprowadzanie hierarchicznej analizy skupień na macierzy odległości	335
Dołączanie dendrogramów do mapy cieplnej	338
Aglomeracyjna analiza skupień w bibliotece scikit-learn	339
Wyznaczanie rejonów o dużej gęstości za pomocą algorytmu DBSCAN	340
Podsumowanie	345
Rozdział 12. Implementowanie wielowarstwowej sieci neuronowej od podstaw	347
Modelowanie złożonych funkcji przy użyciu sztucznych sieci neuronowych	348
Jednowarstwowa sieć neuronowa — powtórzenie	349
Wstęp do wielowarstwowej architektury sieci neuronowych	351
Aktywacja sieci neuronowej za pomocą propagacji w przód	354
Klasyfikowanie pisma odręcznego	356
Zestaw danych MNIST	357
Implementacja perceptronu wielowarstwowego	362
Trenowanie sztucznej sieci neuronowej	371
Obliczanie logistycznej funkcji kosztu	371
Ujęcie intuicyjne algorytmu wstecznej propagacji	374
Uczenie sieci neuronowych za pomocą algorytmu propagacji wstecznej	375

Zbieżność w sieciach neuronowych	378
Jeszcze słowo o implementacji sieci neuronowej	380
Podsumowanie	380
Rozdział 13. Równoległe przetwarzanie sieci neuronowych za pomocą biblioteki TensorFlow	381
Biblioteka TensorFlow a skuteczność uczenia	382
Czym jest biblioteka TensorFlow?	383
W jaki sposób będziemy poznawać bibliotekę TensorFlow?	384
Pierwsze kroki z biblioteką TensorFlow	384
Praca ze strukturami tablicowymi	386
Tworzenie prostego modelu za pomocą podstawowego interfejsu TensorFlow	387
Skuteczne uczenie sieci neuronowych za pomocą wyspecjalizowanych interfejsów biblioteki TensorFlow	391
Tworzenie wielowarstwowych sieci neuronowych za pomocą interfejsu Layers	392
Projektowanie wielowarstwowej sieci neuronowej za pomocą interfejsu Keras	395
Dobór funkcji aktywacji dla wielowarstwowych sieci neuronowych	400
Funkcja logistyczna — powtórzenie	400
Szacowanie prawdopodobieństw przynależności do klas w klasyfikacji wieloklasowej za pomocą funkcji softmax	402
Rozszerzanie zakresu wartości wyjściowych za pomocą funkcji tangensa hiperbolicznego	403
Aktywacja za pomocą prostowanej jednostki liniowej (ReLU)	405
Podsumowanie	407
Rozdział 14. Czas na szczegóły — mechanizm działania biblioteki TensorFlow	409
Główne funkcje biblioteki TensorFlow	410
Rzędy i tensory	410
Sposób uzyskania rzędu i wymiarów tensora	411
Grafy obliczeniowe	412
Węzły zastępcze	414
Definiowanie węzłów zastępczych	414
Wypełnianie węzłów zastępczych danymi	415
Definiowanie węzłów zastępczych dla tablic danych o różnych rozmiarach pakietów danych	416
Zmienne	417
Definiowanie zmiennych	417
Inicjowanie zmiennych	419
Zakres zmiennych	420
Wielokrotne wykorzystywanie zmiennych	421
Tworzenie modelu regresyjnego	423
Realizowanie obiektów w grafie TensorFlow przy użyciu ich nazw	426
Zapisywanie i wczytywanie modelu	428
Przekształcanie tensorów jako wielowymiarowych tablic danych	430
Wykorzystywanie mechanizmów przebiegu sterowania do tworzenia grafów	433
Wizualizowanie grafów za pomocą modułu TensorBoard	436
Zdobywanie doświadczenia w używaniu modułu TensorBoard	439
Podsumowanie	440

Rozdział 15. Klasyfikowanie obrazów za pomocą spłotowych sieci neuronowych	441
Podstawowe elementy spłotowej sieci neuronowej	442
Spłotowe sieci neuronowe i hierarchie cech	442
Spłot dyskretny	444
Podpróbkiwanie	452
Konstruowanie sieci CNN	454
Praca z wieloma kanałami wejściowymi/barw	454
Regularyzowanie sieci neuronowej metodą porzucania	457
Implementacja głębokiej sieci spłotowej za pomocą biblioteki TensorFlow	459
Architektura wielowarstwowej sieci CNN	459
Wczytywanie i wstępne przetwarzanie danych	460
Implementowanie sieci CNN za pomocą podstawowego interfejsu TensorFlow	461
Implementowanie sieci CNN za pomocą interfejsu Layers	471
Podsumowanie	476
Rozdział 16. Modelowanie danych sekwencyjnych za pomocą rekurencyjnych sieci neuronowych	477
Wprowadzenie do danych sekwencyjnych	478
Modelowanie danych sekwencyjnych — kolejność ma znaczenie	478
Przedstawianie sekwencji	478
Różne kategorie modelowania sekwencji	479
Sieci rekurencyjne służące do modelowania sekwencji	480
Struktura sieci RNN i przepływ danych	480
Obliczanie aktywacji w sieciach rekurencyjnych	482
Problemy z uczeniem długofalowych oddziaływań	485
Jednostki LSTM	486
Implementowanie wielowarstwowej sieci rekurencyjnej przy użyciu biblioteki TensorFlow do modelowania sekwencji	488
Pierwszy projekt — analiza sentymentów na zestawie danych IMDb za pomocą wielowarstwowej sieci rekurencyjnej	489
Przygotowanie danych	489
Wektor właściwościowy	492
Budowanie modelu sieci rekurencyjnej	494
Konstruktor klasy SentimentRNN	495
Metoda build	495
Metoda train	499
Metoda predict	500
Tworzenie wystąpienia klasy SentimentRNN	500
Uczenie i optymalizowanie modelu sieci rekurencyjnej przeznaczonej do analizy sentymentów	501
Drugi projekt — implementowanie sieci rekurencyjnej modelującej język na poziomie znaków	502
Przygotowanie danych	503
Tworzenie sieci RNN przetwarzającej znaki	506
Konstruktor	506
Metoda build	507
Metoda train	509
Metoda sample	510
Tworzenie i uczenie modelu CharRNN	512
Model CharRNN w trybie próbkowania	512
Podsumowanie rozdziału i książki	513
Skorowidz	515

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 nauczymy się korzystać z biblioteki scikit-learn, zawierającej przyjazny interfejs użytkownika umożliwiający wydajne oraz produktywnie stosowanie omawianych algorytmów.

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

- omówienie popularnych algorytmów klasyfikujących, m.in. regresji logistycznej, maszyn wektorów nośnych i drzew decyzyjnych,
- przykłady i opisy użycia biblioteki uczenia maszynowego scikit-learn zawierającej szeroki wybór algorytmów uczenia maszynowego poprzez przystępny interfejs API,
- omówienie zalet i wad klasyfikatorów w kontekście liniowych i nieliniowych granic decyzyjnych.

Wybór algorytmu klasyfikującego

Wybór odpowiedniego algorytmu klasyfikującego do rozwiązania konkretnego zadania problemowego wymaga odrobiny praktyki: każdy algorytm ma swoje cechy charakterystyczne i bazuje na określonych założeniach. Przypomnijmy twierdzenie „niedarmowego obiadu” autorstwa Davida H. Wolperta: nie istnieje taki klasyfikator, który działałby optymalnie we wszystkich możliwych sytuacjach (D.H. Wolpert, *The Lack of A Priori Distinctions Between Learning Algorithms*, „Neural Computation 8.7” 1996, 1341 – 1390). W praktyce zawsze jest zalecane porównanie skuteczności przynajmniej kilku różnych algorytmów uczenia maszynowego, dzięki czemu można wybrać model sprawujący się najlepiej w rozwiązaniu danego problemu; poszczególne zagadnienia problemowe różnią się pomiędzy sobą liczbą cech lub próbek, poziomami szumów w zestawie danych, a także liniową rozdzielnością (lub jej brakiem) klas.

Ostatecznie skuteczność klasyfikatora — jego moc obliczeniowa oraz siła predykcyjna — zależy w olbrzymim stopniu od zbioru danych przeznaczonych do uczenia. Trenowanie algorytmu uczenia maszynowego możemy podsumować w pięciu głównych etapach:

1. Dobór cech i gromadzenie przykładów uczących.
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 — uczenie perceptronu

W rozdziale 2., „Trenowanie prostych 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. 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”.

Naukę korzystania z biblioteki scikit-learn rozpoczniemy od wytrenowania modelu perceptronu przypominającego implementację omówioną w rozdziale 2., „Trenowanie prostych 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. W celach wizualizacji będziemy wykorzystywać tylko dwie **cechy kwiatów** kosaćca zawarte w bazie danych *Iris*.

Przydzielimy **długość płatk**a i **szerokość płatk**a 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
>>> print('Etykiety klas:', np.unique(y))
Etykiety klas: [0 1 2]
```

Funkcja `np.unique(y)` zwróciła trzy niepowtarzalne etykiety klas występujące w obiekcie `iris.target` i, jak możemy się przekonać, nazwy trzech gatunków kosaćca (*Iris setosa*, *Iris versicolor* oraz *Iris virginica*) są już przechowywane w postaci liczb całkowitych (tutaj 0, 1, 2). Wiele funkcji i metod klas biblioteki scikit-learn obsługuje również etykiety klas jako ciągi znaków — zalecane jest korzystanie z formatu liczb stałoprzecinkowych, w ten sposób możemy uniknąć problemów technicznych i zwiększyć wydajność obliczeniową z powodu mniejszego wykorzystania pamięci; co więcej, kodowanie etykiet klas jako liczb stałoprzecinkowych jest powszechną konwencją w wielu bibliotekach 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 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=1, stratify=y)
```

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

Zwróć uwagę, że funkcja `train_test_split` przed rozdzieleniem podzbiorów przeprowadza wewnętrzne tasowanie zestawów uczących; gdyby nie to, wszystkie przykłady z klas 0 i 1 znalazłyby się w zbiorze uczącym, a zbiór testowy składałby się z 45 przykładów należących do klasy 2. Dzięki parametrowi `random_state` określiliśmy ziarno losowości (`random_state=1`) dla wewnętrznego generatora liczb pseudolosowych, który służy do tasowania zestawów danych przed ich rozdzieleniem. Wprowadzenie ziarna o ustalonej wartości pozwala nam zachować odtwarzalność doświadczeń.

Wykorzystujemy również wbudowaną obsługę nawarstwiania (stratyfikacji), przyjmującą postać wyrażenia `stratify=y`. W omawianym kontekście nawarstwianie oznacza, że metoda `train_test_split` zwraca podzbiory uczący i testowy mające takie same proporcje etykiet klas jak wejściowy zestaw danych uczących. Możemy użyć funkcji `bincount` (biblioteka NumPy) zliczającej wystąpienia każdej wartości w danej tablicy i przekonać się, że rzeczywiście mamy do czynienia z nawarstwianiem:

```
>>> print('Liczba etykiet w zbiorze y:', np.bincount(y))
Liczba etykiet w zbiorze y: [50 50 50]
>>> print('Liczba etykiet w zbiorze y_train:', np.bincount(y_train))
Liczba etykiet w zbiorze y_train: [35 35 35]
>>> print('Liczba etykiet w zbiorze y_test:', np.bincount(y_test))
Liczba etykiet w zbiorze y_test: [15 15 15]
```

Jak pamiętamy z umieszczonego w rozdziale 2., „Trenowanie prostych 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 modułu przetwarzania wstępnego i zainicjowaliśmy nowy obiekt `StandardScaler`, który przydzieliliśmy do zmiennej `sc`. Za pomocą metody `fit` klasa `StandardScaler` oszacowała parametry μ (wartość średnia próbek) i σ (odchylenie standardowe) dla każdego wymiaru cech stanowiącego część danych uczących. Poprzez wywołanie metody `transform` dokonujemy standaryzacji danych uczących na podstawie wspomnianych parametrów μ i σ . Zwróć 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 kosaćca. Omawiany fragment kodu przedstawia się następująco:

```
>>> from sklearn.linear_model import Perceptron
>>> ppn = Perceptron(n_iter=40, eta0=0.1, random_state=1)
>>> ppn.fit(X_train_std, y_train)
```

Interfejs biblioteki `scikit-learn` przypomina nam implementację perceptronu, omówioną w rozdziale 2., „Trenowanie prostych algorytmów uczenia maszynowego w celach klasyfikacji”: po wczytaniu klasy `Perceptron` z modułu `linear_model` zainicjowaliśmy nowy obiekt `Perceptron` i wuczyliśmy model za pomocą metody `fit`. W tym przypadku parametr `eta0` jest odpowiednikiem

współczynnika uczenia η , 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 prostych 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`, aby zapewnić odtwarzalność 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 prostych 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: 3
```

Po uruchomieniu tego kodu widzimy, że perceptron nieprawidłowo klasyfikuje 3 z 45 próbek, zatem błąd nieprawidłowej klasyfikacji na zbiorze danych testowych wynosi w przybliżeniu 0,067 lub 6,7% ($3/45 \approx 0,067$).

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

1-błąd = 0,933 lub 93,3%

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: 0.93
```

Tutaj zmienna `y_test` oznacza rzeczywiste etykiety klas, a `y_pred` — przewidziane etykiety klas. Ewentualnie każdy klasyfikator w bibliotece `scikit-learn` zawiera metodę `score`, która oblicza dokładność predykcyjną klasyfikatora poprzez połączenie wywołania funkcji `predict` z `accuracy_score` tak, jak pokazano poniżej:

```
>>> print('Dokładność: %.2f' % ppn.score(X_test_std, y_test))
Dokładność: 0.93
```

Zwróć uwagę, że oceniamy skuteczność naszych modeli na podstawie zestawu testowego opisywanego w niniejszym rozdziale. W rozdziale 6., „Najlepsze metody oceny modelu i strojenie parametryczne”, poznasz przydatne techniki, w tym analizę graficzną (np. krzywe uczenia), pozwalające na wykrywanie i zapobieganie nadmiernemu dopasowaniu. **Nadmierne dopasowanie (przetrenowanie, ang. *over-fitting*)** to sytuacja, w której model prawidłowo wykrywa wzorce w danych uczących, jednak nie potrafi ich uogólnić na nieznanne dane.

Możemy w końcu wykorzystać poznaną w rozdziale 2., „Trenowanie prostych algorytmów uczenia maszynowego w celach klasyfikacji”, funkcję `plot_decision_regions` do narysowania wykresu **regionów decyzyjnych** naszego świeżo wyuczonego modelu perceptronu oraz ujrzyć 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=0.3, 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=colors[idx],
                  marker=markers[idx], label=cl,
                  edgecolor='black')

    # zaznacza próbki testowe
    if test_idx:
        # rysuje wykres wszystkich próbek
```

```

X_test, y_test = X[list(test_idx), :], y[list(test_idx)]

plt.scatter(X_test[:, 0], X_test[:, 1], c='', edgecolor='black'
            alpha=1.0, linewidth=1, marker='o', edgecolors='k'
            s=100, label='Zestaw testowy')

```

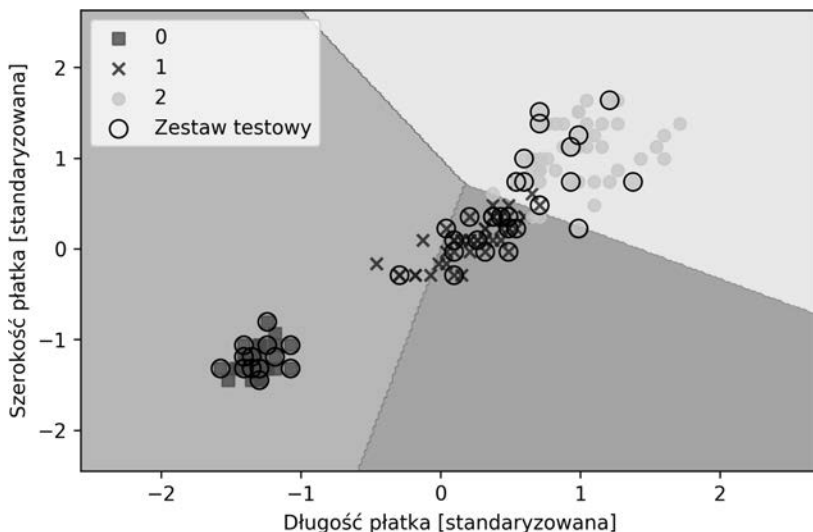
Po wprowadzeniu nieznacznej modyfikacji w funkcji `plot_decision_regions` możemy teraz zdefiniować indeksy próbek, które chcemy zaznaczyć na wykresach. Odpowiedzialny jest za to poniższy fragment kodu:

```

>>> 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 liniową granicą decyzyjną.



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

Przypominamy z rozdziału 2., „Trenowanie prostych 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 potężniejszym klasyfikatorom liniowym, które wykazują zbieżność z minimalną wartością kosztu, nawet jeśli klas nie cechuje doskonała rozdzielność liniowa.

Poszczególne funkcje i klasy biblioteki scikit-learn, w tym Perceptron, często 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 niewłaściwie 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 rozwinąć do klasyfikacji wieloklasowej, np. za pomocą techniki OvR.

Aby wyjaśnić koncepcję regresji logistycznej jako modelu probabilistycznego, wyjaśnijmy najpierw pojęcie **ilorazu szans** (ang. *odds ratio*): 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):

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

Wyrażenie $\log$ oznacza logarytm naturalny; jest to standardowa konwencja w informatyce. 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:

$$\text{logit}(p(y=1 | \mathbf{x})) = w_0 x_0 + w_1 x_1 + \dots + w_m x_m = \sum_{i=0}^m w_i x_i = \mathbf{w}^T \mathbf{x}$$

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

Interesuje nas prognozowanie prawdopodobieństwa przynależności próbki do określonej klasy, co jest odwrotnością funkcji logitowej. Mamy tu do czynienia z **sigmoidalną funkcją logistyczną**, zwaną również w skrócie 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 przykładów, $z = \mathbf{w}^T \mathbf{x} = w_0 x_0 + w_1 x_1 + \dots + w_m x_m$.

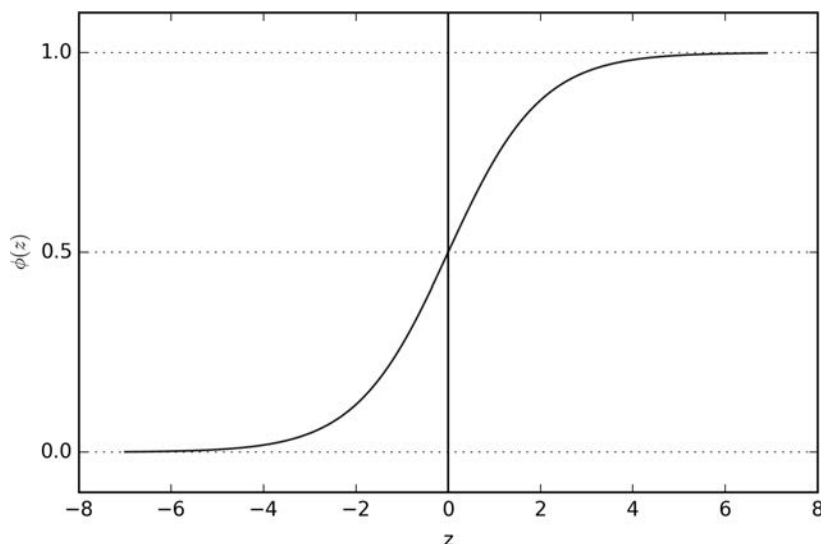
Zwróć uwagę, że podobnie jak w rozdziale 2., „Trenowanie prostych algorytmów uczenia maszynowego w celach klasyfikacji”, stosujemy konwencję, zgodnie z którą w_i oznacza obciążenie jednostkowe, stanowiące dodatkową wartość (równą 1) dla zmiennej x_i .

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.ylim(-0.1, 1.1)
>>> plt.xlabel('z')
>>> plt.ylabel('$\phi(z)$')
>>> # jednostki osi y i siatka
>>> plt.yticks([0.0, 0.5, 1.0])
>>> ax = plt.gca()
>>> ax.yaxis.grid(True)
>>> plt.show()
```

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



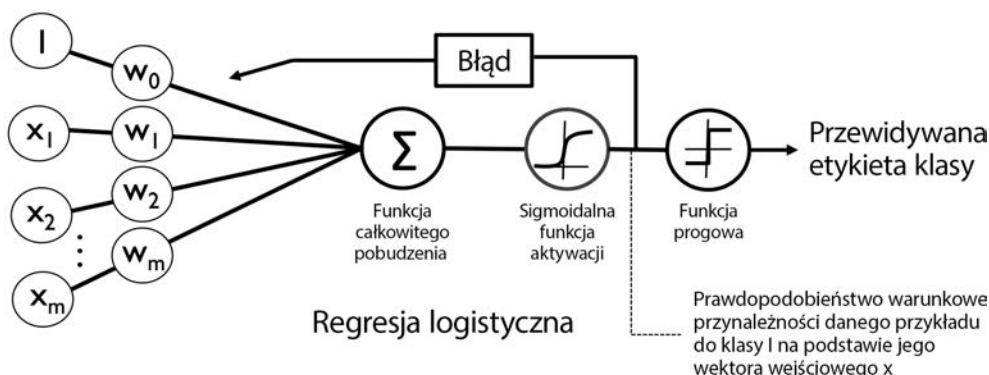
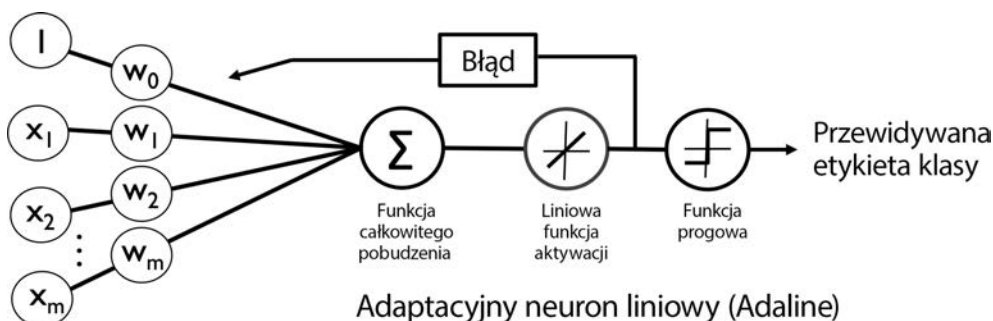
Rysunek 3.2. Wykres funkcji logistycznej

Widzimy, że funkcja $\phi(z)$ dąży do 1, gdy z dąży do nieskończoności ($z \rightarrow \infty$), ponieważ e^{-z} uzyskuje 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 informacjami umieszczonymi w rozdziale 2., „Trenowanie prostych algorytmów uczenia maszynowego 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. Na rysunku 3.3 ukazaliśmy różnice pomiędzy algorytmem Adaline a regresją logistyczną.

Wynik funkcji sigmoidalnej zostaje następnie zinterpretowany jako prawdopodobieństwo przynależności danej próbki do klasy 1, $\phi(z) = P(y=1 | \mathbf{x}; \mathbf{w})$, 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%. Zatem prawdopodobieństwo przynależności tej próbki do gatunku *setosa* możemy wyliczyć ze wzoru $P(y=0 | \mathbf{x}; \mathbf{w}) = 1 - P(y=1 | \mathbf{x}; \mathbf{w}) = 0,2$ lub 20%. Prognozowane prawdopodobieństwo może zostać następnie przekształcone na binarny wynik za pomocą funkcji progowej:

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



Rysunek 3.3. Porównanie algorytmu Adaline i modelu regresji logistycznej

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 oczywiście wiele zastosowań, w których okazuje się przydatne nie tylko przewidywanie etykiet klas, lecz również szacowanie prawdopodobieństwa przynależności określonych instancji do wyznaczonych grup (wynik funkcji sigmoidalnej przed zastosowaniem funkcji progowej). 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 wykorzystywanym modelem w medycynie.

Wyznaczanie wag logistycznej funkcji kosztu

Wiemy już, jak można wykorzystać model regresji logistycznej do prognozowania prawdopodobieństwa i etykiet klas; przyjrzyjmy się teraz pobieżnie sposobom dopasowywania parametrów tego modelu, np. wag w . W poprzednim rozdziale zdefiniowaliśmy funkcję kosztu bazującą na sumie kwadratów błędów:

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

Dokonałymi minimalizacji tej funkcji 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 prostych 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 pojedynczego przykładu uczącego:

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

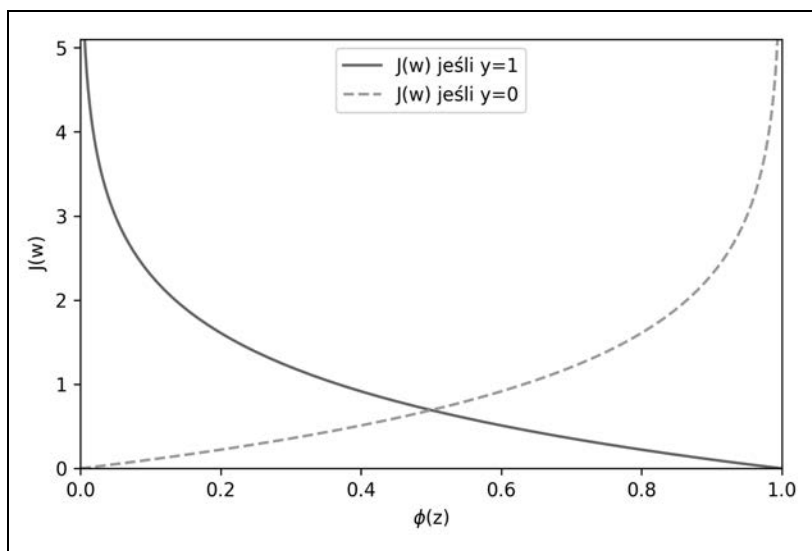
Widzimy, że pierwszy człon równania będzie wynosił 0, jeśli $y = 0$, a drugi człon osiągnie zerową wartość, gdy $y = 1$:

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

Napiszmy niewielki fragment kodu tworzący wykres ilustrujący koszt klasyfikacji jednego przykładu uczącego dla różnych wartości $\phi(z)$:

```
>>> def cost_1(z):
...     return - np.log(sigmoid(z))
>>> def cost_0(z):
...     return - np.log(1 - sigmoid(z))
>>> z = np.arange(-10, 10, 0.1)
>>> phi_z = sigmoid(z)
>>> c1 = [cost_1(x) for x in z]
>>> plt.plot(phi_z, c1, label='J(w) jeśli y=1')
>>> c0 = [cost_0(x) for x in z]
>>> plt.plot(phi_z, c0, linestyle='--', label='J(w) jeśli y=0')
>>> plt.ylim(0.0, 5.1)
>>> plt.xlim([0, 1])
>>> plt.xlabel('$\phi(z)$')
>>> plt.ylabel('J(w)')
>>> plt.legend(loc='best')
>>> plt.show()
```

Otrzymany wykres (który możemy zobaczyć na rysunku 3.4) ukazuje sigmoidalną funkcję aktywacji w osi x , w zakresie od 0 do 1 (danymi wejściowymi były wartości w przedziale od -10 do 10), a także powiązaną z nią logistyczną funkcję kosztu w osi y .



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

Jak widać, koszt dąży do 0 (linia ciągła), 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.

Przekształcanie implementacji Adaline do postaci algorytmu regresji logistycznej

Gdybyśmy chcieli samodzielnie zaimplementować algorytm regresji logistycznej, wystarczyłoby zmodyfikować funkcję kosztu J w implementacji Adaline opisanej w rozdziale 2., „Trenowanie prostych algorytmów uczenia maszynowego w celach klasyfikacji”, za pomocą nowego wzoru:

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

Wykorzystujemy go do obliczania kosztu klasyfikacji wszystkich próbek uczących w danej epoce. Ponadto musimy podmienić liniową funkcję aktywacji na funkcję sigmoidalną i przekształcić funkcję progową tak, aby zwracała etykiety klas 0 i 1, a nie -1 i 1. Jeśli wprowadzimy te trzy zmiany w kodzie algorytmu Adaline, uzyskamy działającą implementację regresji logistycznej, ukazaną poniżej:

```
class LogisticRegressionGD(object):
    """Klasyfikator regresji logistycznej wykorzystujący metodę gradientu prostego

    Parametry
    -----
    eta : zmiennoprzecinkowy
        Współczynnik uczenia (pomiędzy 0,0 a 1,0)
    n_iter : liczba całkowita
        Przebiegi po zestawie danych uczących.
    random_state : liczba całkowita
        Ziarno generatora liczb losowych do losowej inicjacji wag.

    Atrybuty
    -----
    w_ : tablica jednowymiarowa
        Wagi po dopasowaniu.
    cost_ : lista
        Wartość funkcji kosztu (suma kwadratów) w każdej epoce.

    """
    def __init__(self, eta=0.05, n_iter=100, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state

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

Parametry

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

Zwraca

self : obiekt

```

"""
rgen = np.random.RandomState(self.random_state)
self.w_ = rgen.normal(loc=0.0, scale=0.01,
                      size=1 + X.shape[1])

self.cost_ = []

for i in range(self.n_iter):
    net_input = self.net_input(X)
    output = self.activation(net_input)
    errors = (y - output)
    self.w_[1:] += self.eta * X.T.dot(errors)
    self.w_[0] += self.eta * errors.sum()

    # Zwróć uwagę, że obliczamy teraz 'koszt' logistyczny, a nie
    # sumę kwadratów błędów.
    cost = (-y.dot(np.log(output)) -
            ((1 - y).dot(np.log(1 - output))))
    self.cost_.append(cost)
    return self

def net_input(self, X):
    """Obliczanie pobudzenia całkowitego"""
    return np.dot(X, self.w_[1:]) + self.w_[0]

def activation(self, z):
    """Obliczanie logistycznej, sigmoidalnej funkcji aktywacji"""
    return 1. / (1. + np.exp(-np.clip(z, -250, 250)))

def predict(self, X):
    """Zwraca etykiety klasy po skoku jednostkowym"""
    return np.where(self.net_input(X) >= 0.0, 1, 0)
    # equivalent to:
    # return np.where(self.activation(self.net_input(X))
    #                 >= 0.5, 1, 0)

```

W trakcie dopasowywania modelu regresji logistycznej musimy pamiętać, że działa on jedynie w zadaniach klasyfikacji binarnej. Weźmy więc pod uwagę wyłącznie odmiany *Iris setosa* i *Iris versicolor* (klasy 0 i 1) i sprawdźmy skuteczność naszej implementacji:

```

>>> X_train_01_subset = X_train[(y_train == 0) | (y_train == 1)]
>>> y_train_01_subset = y_train[(y_train == 0) | (y_train == 1)]
>>> lr_gd = LogisticRegressionGD(eta=0.05,
...                               n_iter=1000,

```

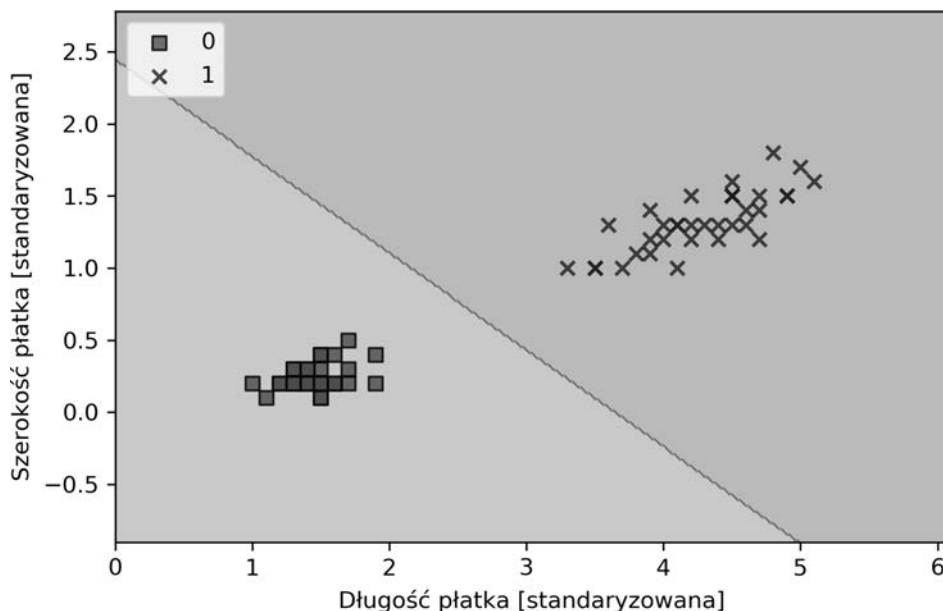
```

... random_state=1)
>>> lrgd.fit(X_train_01_subset,
...          y_train_01_subset)

>>> plot_decision_regions(X=X_train_01_subset,,
...                       y=y_train_01_subset,,
...                       classifier=lrgd)
>>> plt.xlabel('Długość płatka [standaryzowana]')
>>> plt.ylabel('Szerokość płatka [standaryzowana]')
>>> plt.legend(loc='upper left')
>>> plt.show()

```

Otrzymany wykres regionu decyzyjnego został ukazany na rysunku 3.5.



Rysunek 3.5. Region decyzyjny otrzymany za pomocą modelu regresji logistycznej

Algorytm gradientu prostego w modelu regresji logistycznej

Za pomocą analizy matematycznej 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 prostych algorytmów uczenia maszynowego w celach klasyfikacji”. Zwracamy jednak uwagę, że przedstawiony poniżej wzór pochodnej reguły uczenia wykorzystującej algorytm gradientu prostego przeznaczony jest dla Czytelników interesujących się aspektami matematycznymi wyjaśniającymi regułę uczenia gradientu prostego w modelu regresji logistycznej. Informacje zawarte w tej ramce nie są niezbędne do zrozumienia dalszej części rozdziału.

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, obliczmy 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 aktualizujemy 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 uaktualniania 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:

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_{i=1}^n (y^{(i)} - \phi(z^{(i)})) x_j^{(i)}$$

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

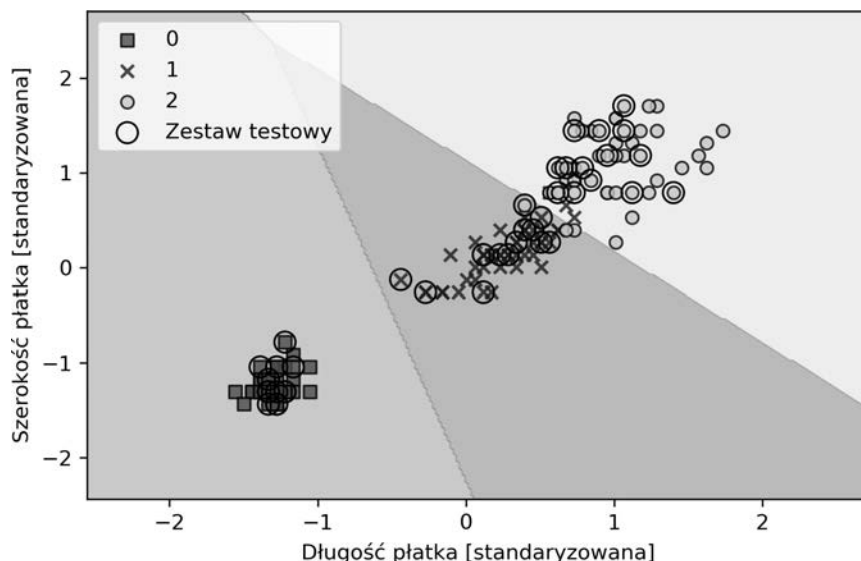
Jest to równoważne regule gradientu prostego omówionej w rozdziale 2., „Trenowanie prostych algorytmów uczenia maszynowego w celach klasyfikacji”.

Uczenie modelu regresji logistycznej za pomocą biblioteki scikit-learn

W poprzednim punkcie zajmowaliśmy się przydatnym kodem oraz zagadnieniami matematycznymi, dzięki czemu jesteśmy w stanie zrozumieć różnice koncepcyjne pomiędzy algorytmem Adaline a regresją logistyczną. Teraz nauczymy się wykorzystywać zoptymalizowaną implementację regresji logistycznej umożliwiającą również klasyfikację wieloklasową (domyślnie za pomocą techniki OvR). W poniższym fragmencie kodu użyjemy klasy `sklearn.linear_model.LogisticRegression`, jak również znanej już nam metody `fit` do wyuczenia modelu wobec wszystkich trzech klas zdefiniowanych w standaryzowanym zestawie danych Iris.

```
>>> from sklearn.linear_model import LogisticRegression
>>> lr = LogisticRegression(C=1000.0, random_state=1)
>>> 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 dopasowaniu modelu do danych uczących wygenerowaliśmy wykres regionów decyzyjnych, przykładów uczących i testowych, co zaprezentowaliśmy na rysunku 3.6.



Rysunek 3.6. Wykres wyników uczenia po zastosowaniu algorytmu regresji logistycznej wbudowanego w bibliotekę scikit-learn

Po przyjrzeniu się powyższemu listingowi wykorzystanemu do uczenia modelu `Logistic Regression` możemy się zastanawiać, do czego służy tajemniczy parametr `C`. Powrócimy do tego zagadnienia w kolejnym punkcie, w którym poruszymy temat nadmiernego dopasowania i regularyzacji. Najpierw jednak chcielibyśmy przyjrzeć się uważniej kwestii prawdopodobieństwa przynależności do danej klasy.

Możemy obliczyć prawdopodobieństwo przynależności przykładu uczącego do określonej klasy za pomocą metody `predict_proba`. Prognozujemy to prawdopodobieństwo dla trzech pierwszych przykładów w zestawie w następujący sposób:

```
>>> lr.predict_proba(X_test_std[:3, :])
```

Otrzymujemy w odpowiedzi poniższą tablicę:

```
array([[ 3.20136878e-08,  1.46953648e-01,  8.53046320e-01],
       [ 8.34428069e-01,  1.65571931e-01,  4.57896429e-12],
       [ 8.49182775e-01,  1.50817225e-01,  4.65678779e-13]])
```

Pierwszy wiersz ukazuje prawdopodobieństwa przynależności pierwszego kwiatu do poszczególnych klas, drugi – kwiatu drugiego itd. Zwróć uwagę, że zgodnie z oczekiwaniami poszczególne kolumny sumują się do wartości 1 (możesz się o tym przekonać po wpisaniu `lr.predict_proba(X_test_std[:3, :]).sum(axis=1)`). Najwyższą wartością w pierwszym rzędzie jest w przybliżeniu 0,853, co oznacza, że pierwszy przykład należy do klasy trzeciej (*Iris-virginica*) z prawdopodobieństwem 85,7%. Zatem, jak łatwo zauważyć, możemy otrzymywać przewidywane etykiety klas poprzez wyznaczenie kolumny zawierającej największą wartość w danym wierszu, np. korzystając z funkcji `argmax`:

```
>>> lr.predict_proba(X_test_std[:3, :]).argmax(axis=1)
```

Poniżej widzimy zwrócone indeksy klas (odpowiadają one kolejno odmianom: *Iris virginica*, *Iris setosa* i *Iris versicolor*):

```
array([2, 0, 0])
```

Uzyskane w powyższy sposób etykiety klas z prawdopodobieństw warunkowych zostały oczywiście wyliczone ręcznie, poprzez bezpośrednie wywołanie metody `predict`, co możemy szybko zweryfikować, tak jak pokazano poniżej:

```
>>> lr.predict(X_test_std[:3, :])
array([2, 0, 0])
```

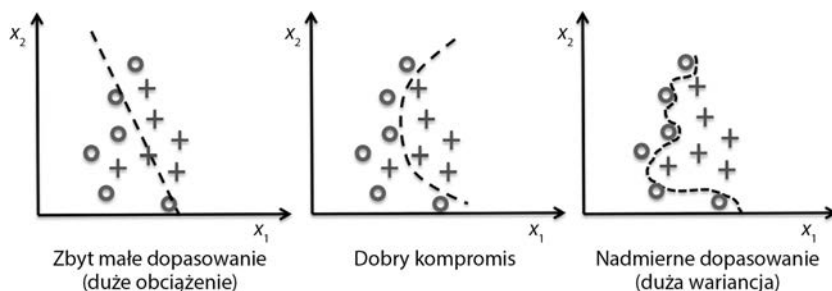
Na koniec małe ostrzeżenie dla osób chcących przewidywać etykietę klasy pojedynczego przykładu uczącego: biblioteka `scikit-learn` oczekuje na wejściu danych w formacie tablicy dwuwymiarowej; zatem musimy przekształcić pojedynczy wiersz do odpowiedniej postaci. Jednym ze sposobów konwersji tablicy jednowymiarowej do formy dwuwymiarowej jest użycie metody `reshape` z biblioteki `NumPy`, co powoduje dodanie kolejnego wymiaru:

```
>>> lr.predict(X_test_std[0, :].reshape(1, -1))
array([2])
```

Zapobieganie przetrenowaniu za pomocą regularyzacji

Jednym z częstych problemów występujących w uczeniu maszynowym jest przetrenowanie/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 przetwarzać określony zestaw danych. Zdarzają się sytuacje przeciwne, **niedotrenowanie/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 przetrenowania i niedotrenowania najłatwiej ukazać poprzez porównanie liniowej granicy decyzyjnej z bardziej złożonymi, nieliniowymi granicami, co zostało zaprezentowane na rysunku 3.7.



Rysunek 3.7. Przykłady przetrenowania i niedotrenowania

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 przewidywań 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 parametru (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 λ 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.

Aby regularyzować funkcję kosztu w modelu regresji logistycznej, wystarczy dodać odpowiedni człon regularyzacji, który posłuży do zmniejszania wag w trakcie uczenia:

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

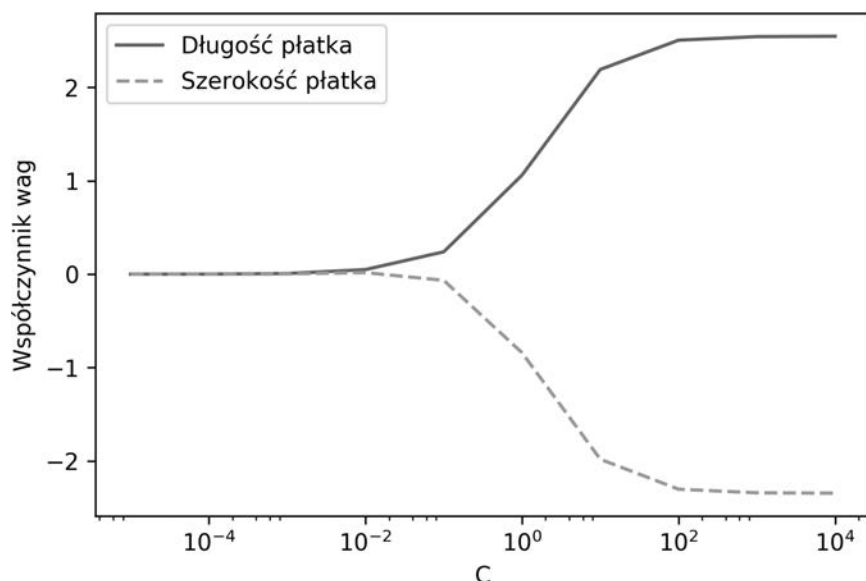
Możemy teraz za pomocą parametru regularyzacji λ kontrolować trenowanie na podstawie danych uczących przy jednoczesnym utrzymywaniu niskich wartości wag. Zwiększając wartość λ , wzmacniamy siłę regularyzacji.

Zaimplementowany w klasie `LogisticRegression` biblioteki `scikit-learn` parametr C stanowi element konwencji wywodzącej się z maszyn wektorów nośnych, którymi zajmujemy się w następnym podrozdziale. Krótko mówiąc, człon C jest bezpośrednio powiązany z parametrem λ , gdyż stanowi jego odwrotność. 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=1)
...     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 płatka')
>>> 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 wyświetlone zostały jedynie współczynniki wagowe z klasy 1 (w tym przypadku drugiej klasy w zestawie danych, czyli *Iris-versicolor*) przeciw pozostałym klasyfikatorom — pamiętaj, że do klasyfikacji wieloklasowej używamy techniki OvR.

Jak widać na rysunku 3.8, współczynniki wag maleją w sytuacji zmniejszania wartości parametru C , tzn. w trakcie zwiększania siły regularyzacji.



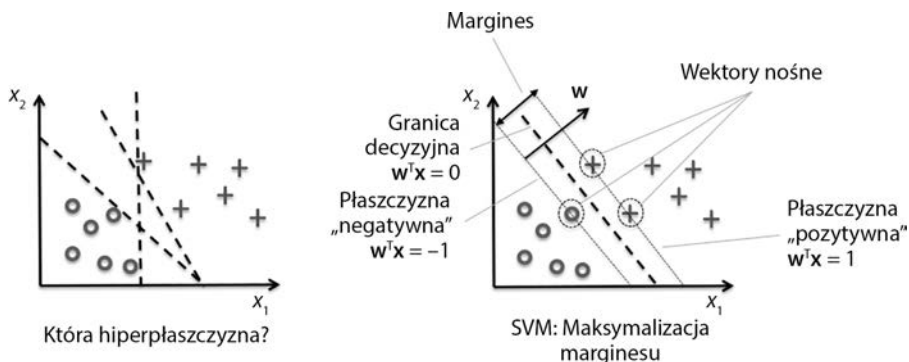
Rysunek 3.8. Wykres regularyzacji

Dokładny opis każdego algorytmu klasyfikacyjnego wykracza poza ramy niniejszej książki, dlatego wszystkim osobom pragnącym dowiedzieć się więcej na temat regresji logistycznej gorąco polecamy pozycję autorstwa dra Scotta Menarda *Logistic Regression: From Introductory to Advanced Concepts and Applications*, 2009, wydaną nakładem wydawnictwa SAGE Publications¹.

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 próbkami uczącymi (tzw. **wektorami nośnymi**). Koncepcja ta została zaprezentowana na rysunku 3.9.

¹ 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.*



Rysunek 3.9. 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, przyjrzyjmy się uważniej **pozytywnym** i **negatywnym** hiperpłaszczyznom (hiperpłaszczyznom ułożonym równoległe do granicy decyzyjnej), których wzory wyglądają następująco:

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

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

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

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

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

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

Dochodzimy więc do następującego równania:

$$\frac{\mathbf{w}^T (\mathbf{x}_{poz} - \mathbf{x}_{neg})}{\|\mathbf{w}\|} = \frac{2}{\|\mathbf{w}\|}$$

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

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

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

N oznacza tu liczbę przykładów w zestawie danych.

Te dwa wzory mówią nam, że wszystkie negatywne próbki powinny wyładować po stronie negatywnej hiperpłaszczyzny, a wszystkie próbki pozytywne — po stronie hiperpłaszczyzny pozytywnej, co możemy zapisać w krótszej postaci:

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

W rzeczywistości łatwiej minimalizować odwrotne wyrażenie $\frac{1}{2} \|\mathbf{w}\|^2$, które można obliczyć za pomocą programowania kwadratowego. Szczegółowe omówienie programowania kwadratowego wykracza poza zakres niniejszej książki. Osoby zainteresowane 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 rozdzielnych za pomocą zmiennych uzupełniających

Nie chcę wprowadzać żadnych bardziej zaawansowanych pojęć matematycznych dotyczących klasyfikacji maksymalnego marginesu, muszę jednak wspomnieć o zmiennej uzupełniającej (ang. *slack variable*) ξ , która została 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 nieliniowo rozdzielnych danych, co pozwalałoby na uzyskanie zbieżności algorytmu uczącego w obecności nieprawidłowych klasyfikacji podczas stosowania odpowiedniej metody karcenia kosztów.

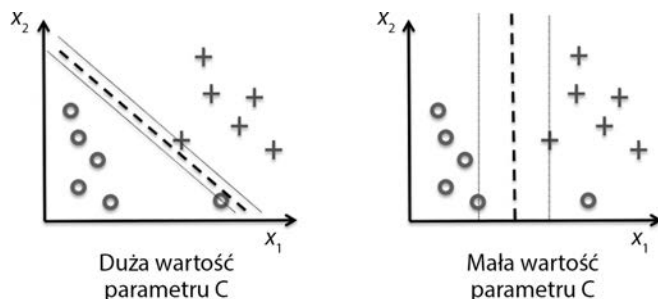
Wystarczy wstawić zmienną uzupełniającą do wzoru ograniczeń liniowych:

$$\begin{aligned} w_0 + \mathbf{w}^T \mathbf{x}^{(i)} &\geq 1 - \xi^{(i)} \text{ jeśli } y^{(i)} = 1 \\ w_0 + \mathbf{w}^T \mathbf{x}^{(i)} &\leq -1 + \xi^{(i)} \text{ jeśli } y^{(i)} = -1 \\ \text{dla } i &= 1 \dots N \end{aligned}$$

N oznacza tu liczbę przykładów w zestawie danych. Zatem nowym celem minimalizacji (zależnym od ograniczeń) staje się:

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

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 3.10.



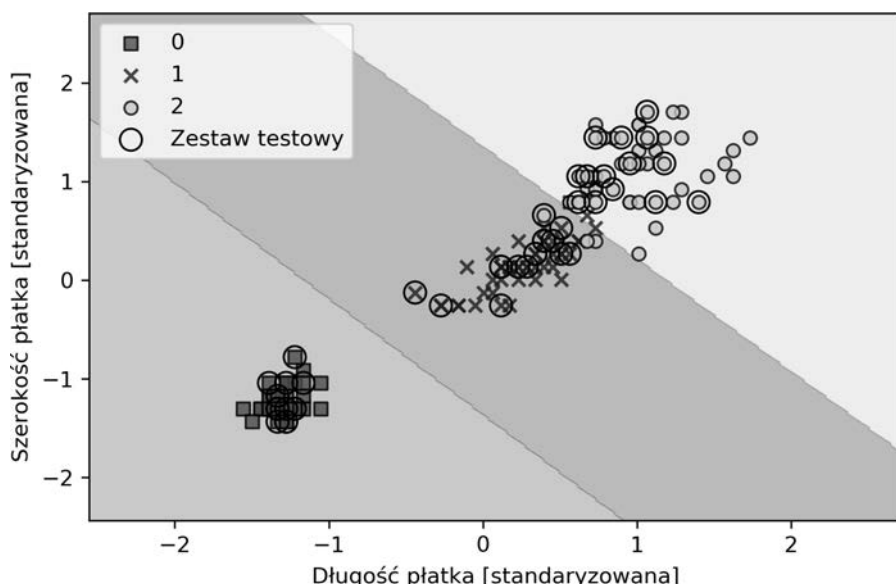
Rysunek 3.10. Wpływ zmiennej C na szerokość marginesu

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

Skoro już znamy podłoże 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=1)
>>> 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()
```

Trzy regiony decyzyjne wyznaczone po wyuczeniu klasyfikatora wobec zestawu danych Iris za pomocą algorytmu SVM zostały zaprezentowane na rysunku 3.11.



Rysunek 3.11. Efekty uczenia za pomocą modelu maszyny wektorów nośnych

Regresja logistyczna a maszyny wektorów nośnych

W praktycznych zadaniach klasyfikacji liniowa regresja logistyczna i liniowe maszyny 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, który jest z kolei bardziej nastawiony na punkty znajdujące się najbliżej granicy decyzyjnej (wektory nośne). Z drugiej strony zaletami regresji logistycznej są mniejsza złożoność modelu i 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

Używana w poprzednich podrozdziałach klasa `LogisticRegression` wykorzystuje znakomicie zoptymalizowaną w języku C/C++ bibliotekę `LIBLINEAR` stworzoną na Narodowym Uniwersytecie Tajwańskim (<http://www.csie.ntu.edu.tw/~cjlin/liblinear/>). Z kolei klasa `Perceptron` posługuje się do optymalizacji standardową implementacją algorytmu SGD. Podobnie klasa `SVC` zastosowana do uczenia modelu SVC korzysta z biblioteki `LIBSVM`, wyspecjalizowanej do obsługi maszyn 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 wzdłuż gradientu, który zaimplementowaliśmy w rozdziale 2., „Trenowanie prostych algorytmów uczenia maszynowego w celach klasyfikacji”, dla modelu Adaline. Możemy zainicjować stochastyczny spadek wzdłuż 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

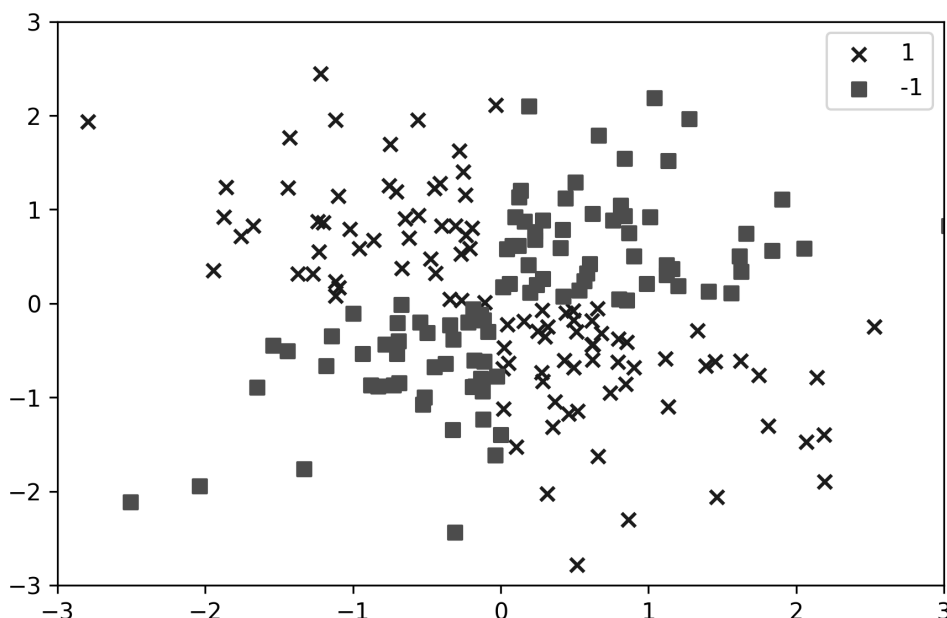
Kolejnym powodem wielkiej popularności algorytmu SVM wśród społeczności zajmującej się uczeniem maszynowym jest możliwość jego łatwej **kernelizacji** w celu rozwiązywania nieliniowych problemów klasyfikacji. Zanim zaczniemy omawiać koncepcję **jądra SVM**, zdefiniujmy i stwórzmy najpierw zbiór próbek,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

Metody jądrowe dla danych nierozdzielnych liniowo

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

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np.
>>> np.random.seed(1)
>>> 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.xlim([-3, 3])
>>> plt.ylim([-3, 3])
>>> plt.legend(loc='best')
>>> plt.show()
```

Po uruchomieniu powyższego kodu uzyskamy zbiór danych wygenerowany za pomocą bramki XOR, cechujący się losowym zaszumieniem, co zostało zaprezentowane na rysunku 3.12.



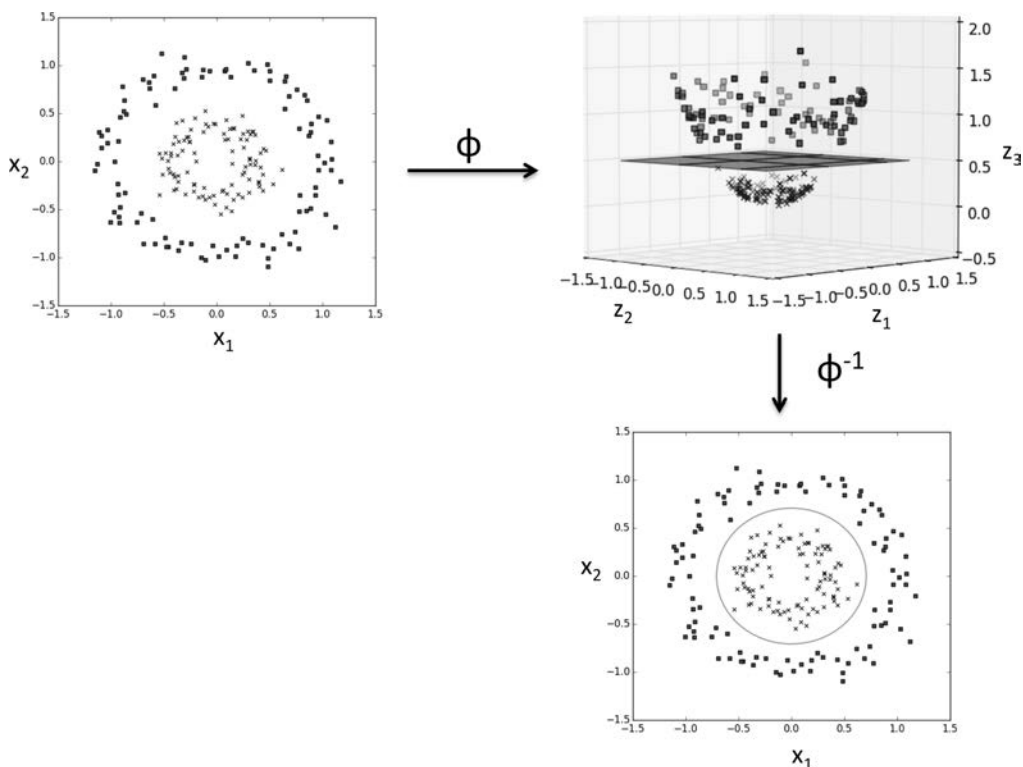
Rysunek 3.12. Zestaw danych wygenerowanych za pomocą bramki XOR

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

W metodach wykorzystujących funkcje jądrowe podstawową koncepcją radzenia sobie z tak nierozdzielniymi liniowo danymi jest utworzenie nieliniowych kombinacji pierwotnych cech, które za pomocą funkcji mapowania ϕ będą rzutowane na przestrzeń mającą więcej wymiarów, gdzie staną się liniowo separowalne. Jak widać na rysunku 3.13, możemy przekształcić dwuwymiarowy zbiór danych na nową, trójwymiarową przestrzeń cech, gdzie klasy stają się rozdzielne poprzez poniższe rzutowanie:

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

Uzyskujemy w ten sposób możliwość rozdzielenia dwóch klas widocznych na rysunku 3.13 za pomocą liniowej hiperpłaszczyzny, która zostaje przekształcona w nieliniową granicę decyzyjną po rzutowaniu do pierwotnej przestrzeni cech.



Rysunek 3.13. Wykorzystanie funkcji mapowania do utworzenia nieliniowej granicy decyzyjnej

Stosowanie sztuczki z funkcją jądra do znajdowania przestrzeni rozdzielających w przestrzeni wielowymiarowej

Aby rozwiązać nieliniowy problem za pomocą algorytmu SVM, za pomocą funkcji mapowania ϕ 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 liniowego 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 $\phi(\mathbf{x}^{(i)})^T \phi(\mathbf{x}^{(j)})$. By zaoszczędzić kosztownej operacji bezpośredniego obliczania tego iloczynu skalarnego pomiędzy dwoma punktami, definiujemy tzw. **funkcję jądra**:

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

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

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

Często jest ono upraszczane do postaci:

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

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).

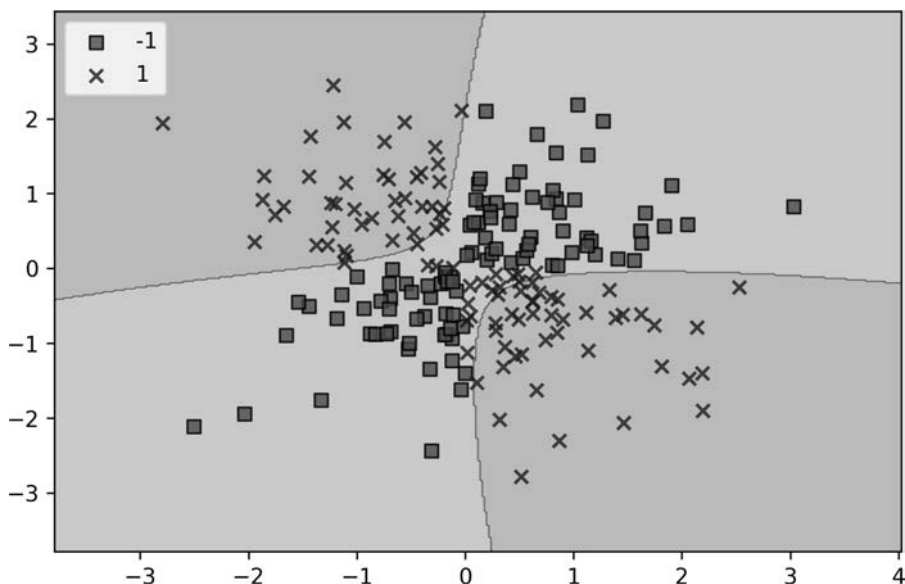
Skoro już znamy ogólne założenia sztuczki z funkcją jądra, sprawdźmy, czy jesteśmy w stanie wytrenować jądro SVM w taki sposób, żeby skutecznie wyrysowało nieliniową granicę decyzyjną rozdzielającą nasze próbki wygenerowane za pomocą bramki XOR. Wykorzystamy tu ponownie klasę SVC i zastąpimy parametr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

```
>>> svm = SVC(kernel='rbf', random_state=1, gamma=0.10, 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 3.14, jądro SVM względnie dobrze rozdziela dane XOR.

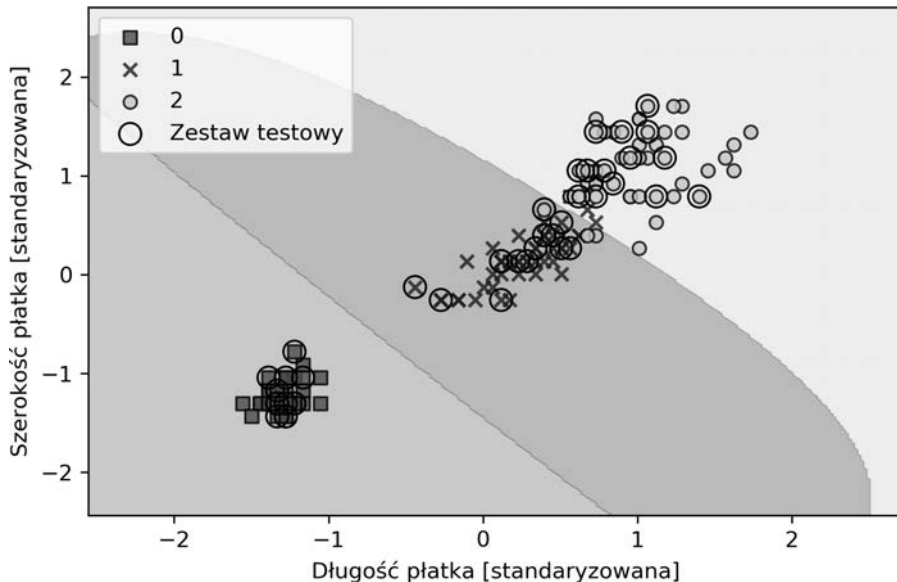
Parametr γ (zdefiniowany jako `gamma=0.1`) możemy uznać za obszar graniczny sfery Gaussa. Jeżeli zwiększymy wartość zmiennej γ , podniesiemy również wpływ (zasięg) próbek uczących, co doprowadzi do sztywniejszych i bardziej pofałdowanych granic decyzyjnych. Aby lepiej zrozumieć ten parametr, zastosujemy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

```
>>> svm = SVC(kernel='rbf', random_state=1, gamma=0.2, 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()
```



Rysunek 3.14. Granica decyzyjna wygenerowana za pomocą jądra SVM

Wybraliśmy względnie małą wartość parametru γ , dlatego uzyskamy dość elastyczne granice decyzyjne, co zostało zilustrowane na rysunku 3.15.

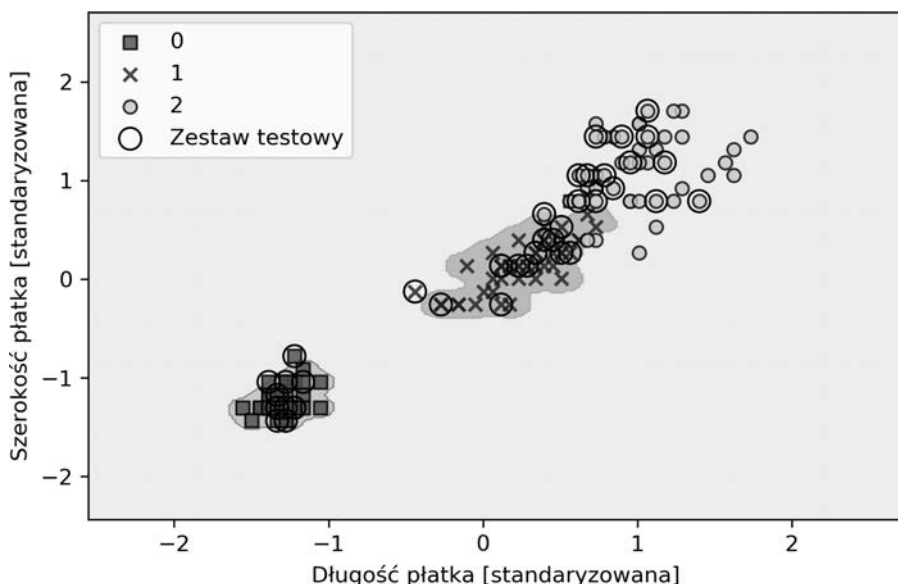


Rysunek 3.15. Nieliniowe granice decyzyjne uzyskane za pomocą niskiej wartości parametru γ

Zwiększmy teraz wartość parametru γ i zobaczmy, jaki będzie to miało wpływ na granice decyzyjne:

```
>>> svm = SVC(kernel='rbf', random_state=1, 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 3.16 widzimy, że po wprowadzeniu dużej wartości parametru γ granice decyzyjne wokół klas 0 i 1 stają się znacznie bardziej zwarte.



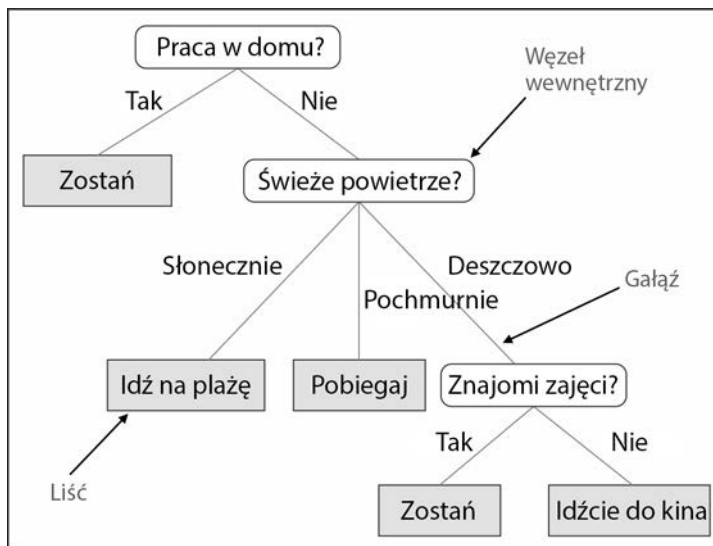
Rysunek 3.16. Wpływ zwiększenia wartości parametru γ na granice decyzyjne

Chociaż tak skonfigurowany model znakomicie dopasowuje dane uczące, klasyfikator prawdopodobnie będzie generował duży błąd nieprawidłowej klasyfikacji wobec nieznanymi danych. Optymalizacja parametru γ odgrywa również niebagatelną rolę w kontroli nadmiernego dopasowania.

Uczenie drzew decyzyjnych

Modele **drzew decyzyjnych** (ang. *decision tree*) są atrakcyjne, jeżeli zadbamy 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 3.17, w którym wykorzystujemy drzewo decyzyjne do ustalenia zajęć w danym dniu.



Rysunek 3.17. Przykład drzewa decyzyjnego

Na podstawie cech zestawu uczącego model drzewa decyzyjnego wykorzystuje szereg pytań do określania etykiet klas próbek. Na rysunku 3.17 widzimy drzewo decyzyjne działające w zbiorze kategorii, nie 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 $\geq 2,8$ cm?”.

Za 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 rozwiązanie to często skutkuje powstawaniem dużych, wielowęzłowych drzew, co może z łatwością prowadzić do przetrenowania. Z tego powodu zazwyczaj chcemy **przycinać** drzewo poprzez ustawienie 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: nadrzędnego węzła i j -tego węzła potomnego, I stanowi miarę **zanieczyszczenia**, N_p definiuje całkowitą liczbę próbek w węźle nadrzędnym, a N_j — w j -tym węźle potomnym. Jak widać, przyrost informacji to nic innego, jak różnica pomiędzy zanieczyszczeniem węzła nadrzędnego 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 nadrzędny rozgałęzia się na dwa węzły potomne: D_{lewy} i D_{prawy} :

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

W binarnych drzewach decyzyjnych trzema najpowszechniej wykorzystywanymi 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 próbkami 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 ($c = 2$):

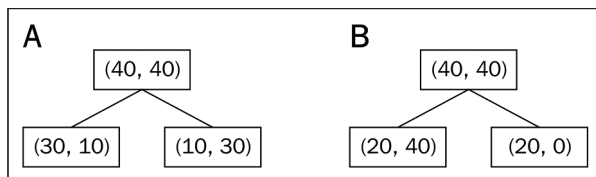
$$I_G(t) = 1 - \sum_{i=1}^c 0,5^2 = 0,5$$

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 wewnątrz węzła. Wyjaśnię to na przykładzie dwóch możliwych scenariuszy rozgałęziania (rysunek 3.18).



Rysunek 3.18. Dwie możliwości rozgałęziania binarnego drzewa decyzyjnego

Zaczynamy od zestawu danych D_p znajdującego się w węźle nadrzędnym D_p . Na zbiór tych danych składa się 40 próbek z klasy 1 i 40 próbek z klasy 2, które rozdzielamy 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):

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

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

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

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

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

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

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

Jednak wskaźnik Giniego bardziej sprzyja rozgałęzieniu ze scenariusza B ($IG_G = 0,16$) niż ze scenariusza A ($IG_G = 0,125$), gdyż rzeczywiście jest **czystsze**:

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

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

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

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

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

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

$$B: IG_G = 0,5 - \frac{6}{8}0,4 - 0 = 0,16$$

Analogicznie kryterium entropii również 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=5, 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 3.19.

Budowanie drzewa decyzyjnego

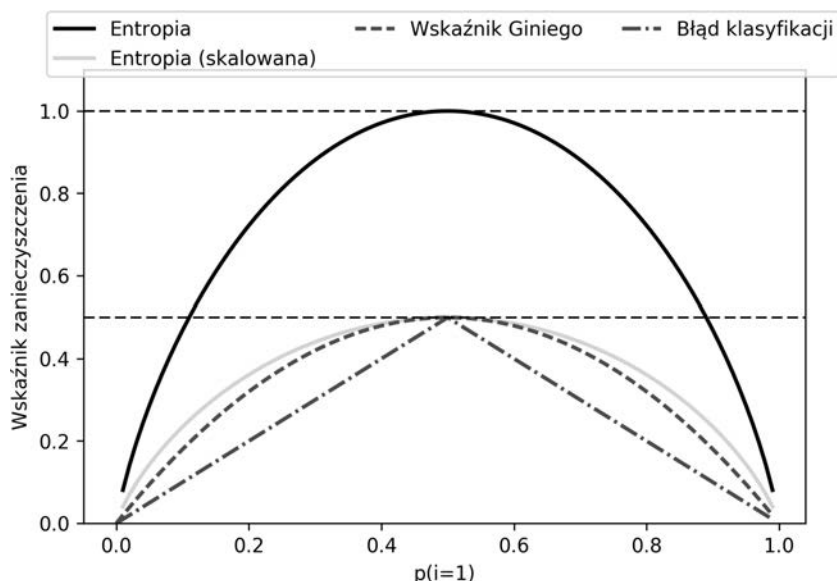
Drzewa decyzyjne generują skomplikowane granice decyzyjne poprzez podzielenie przestrzeni cech na prostokątne obszary. Musimy jednak zachować ostrożność, ponieważ im większe drzewo, tym granice decyzyjne stają się bardziej złożone, co może doprowadzić do przetrenowania.

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='gini',
...                               max_depth=4, random_state=1)
>>> 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,

```



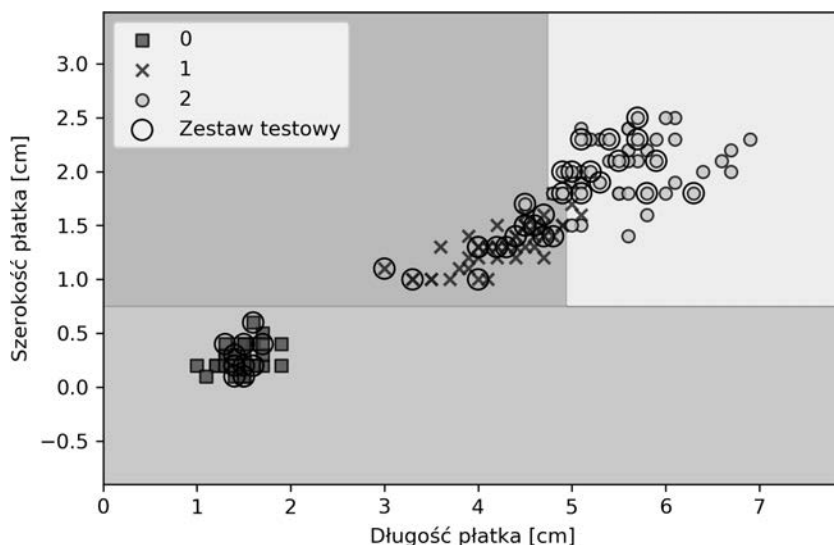
Rysunek 3.19. Porównanie indeksów zanieczyszczenia

```

... 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, równoległe do osi układu współrzędnych granice generowane przez algorytm drzewa decyzyjnego (rysunek 3.20).



Rysunek 3.20. Wykres granic decyzyjnych wygenerowanych za pomocą algorytmu drzewa decyzyjnego

Ciekawą funkcją interfejsu scikit-learn jest możliwość eksportowania drzewa decyzyjnego w formacie *.dot* po zakończeniu uczenia, np. 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. Ponadto wykorzystujemy w tym celu bibliotekę Python o nazwie pydotplus, która funkcjonalnością przypomina program GraphViz i pozwala przekształcać pliki danych *.dot* do postaci obrazu drzewa decyzyjnego. Po zainstalowaniu aplikacji GraphViz (instrukcję znajdziesz pod adresem <https://www.graphviz.org/download/>) możesz zainstalować bibliotekę pydotplus bezpośrednio poprzez instalator pip; w tym celu wpisz następującą komendę w Terminalu:

```
> pip3 install pydotplus
```

W niektórych systemach być może będziesz musiał zainstalować pakiety zależne biblioteki pydotplus ręcznie, poprzez wpisanie poniższych poleceń:

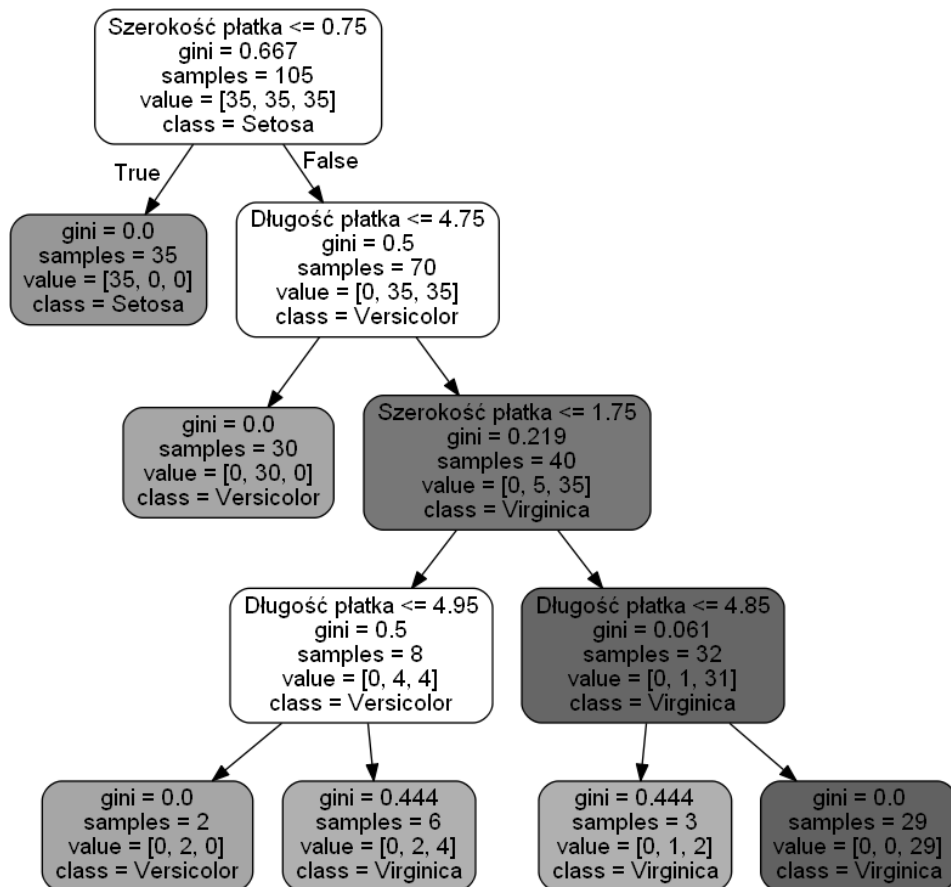
```
pip3 install graphviz
pip3 install pyparsing
```

Poniższy kod tworzy obraz naszego drzewa decyzyjnego w formacie PNG i umieszcza go w katalogu lokalnym:

```
>>> from pydotplus import graph_from_dot_data
>>> from sklearn.tree import export_graphviz
>>> dot_data = export_graphviz(tree,
...                             filled=True,
...                             rounded=True,
...                             class_names=['Setosa',
...                             'Versicolor',
...                             'Virginica'],
...                             feature_names=['Długość
płotka', 'Szerokość płotka'])
...                             out_file=None,
>>> graph = graph_from_dot_data(dot_data)
>>> graph.write_png('drzewo.png')
```

Opcja `out_file=None` pozwala nam bezpośrednio przydzielić dane w formacie *.dot* do zmiennej `dot_data` i nie musimy zapisywać pośredniego pliku *drzewo.dot* na dysku. Argumenty wyznaczone dla parametrów: `filled`, `rounded`, `class_names` i `feature_names` nie są konieczne, ale poprawiają wygląd generowanego obrazu drzewa poprzez dodanie kolorów, zaokrąglenie krawędzi pól, ukazanie nazwy etykiety klasy o największym prawdopodobieństwie przynależności w każdym węźle, a także ukazanie nazw cech w kryterium podziału. W ten sposób otrzymaliśmy obraz drzewa decyzyjnego ukazany na rysunku 3.21.

Dzięki temu obrazowi możemy teraz w bardzo wygodny sposób prześledzić poszczególne rozgałęzienia uzyskane na podstawie danych uczących. Zaczynamy u korzenia od 105 próbek i rozdzielamy je pomiędzy 2 węzły potomne — odpowiednio 35 i 70 próbek — korzystając z parametru granicznej szerokości płatka ≤ 0.75 cm. Już przy pierwszym rozgałęzieniu widzimy, że lewy węzeł jest czysty i zawiera wyłącznie próbki pochodzące z klasy Iris-setosa (wskaźnik Giniego = 0). Kolejne podziały prawego węzła dążą do rozdzielenia klas Iris-versicolor i Iris-virginica.



Rysunek 3.21. Obraz drzewa decyzyjnego wygenerowany za pomocą aplikacji Graphviz

Jak widać na rysunku 3.21 i wykresie regionu decyzyjnego, algorytm drzewa decyzyjnego znakomicie radzi sobie z rozróżnianiem odmian kosaćca. Niestety, obecnie biblioteka scikit-learn nie zawiera funkcji ręcznego przycinania drzew decyzyjnych. Jednak w powyższym kodzie wystarczyłoby zmienić wartość parametru `max_depth` na 3 i porównać uzyskane rezultaty z bieżącym modelem, ale pozostawimy to ćwiczenie zainteresowanym Czytelnikom.

Łączenie wielu drzew decyzyjnych 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 losowym lasem polega na uśrednianiu wielu (wysokich) drzew decyzyjnych, które osobno cechują się znaczną wariancją, i łączeniu ich w jeden skuteczniejszy model mający większą wydajność 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”.

Należy zauważyć, że 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.

Jeżeli nie znasz pojęć losowania **ze zwracaniem** oraz **bez zwracania**, przeprowadźmy prosty eksperyment myślowy. Załóżmy, 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 1/5. W przypadku próbkowania 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 rundzie liczby 0 zwiększa się do 1/4.

Jednak w przypadku próbkowania ze zwracaniem za każdym razem wrzucamy wylosowaną liczbę z powrotem do urny, przez co prawdopodobieństwo wyciągnięcia określonej liczby w kolejnej turze pozostaje niezmiennione; istnieje możliwość, że znowu wyciągniemy tę samą cyfrę. Innymi słowy, w próbkowaniu ze zwracaniem próbki (liczby) są od siebie wzajemnie niezależne i mają zerową kowariancję. Wyniki losowania pięciu liczb mogą wyglądać następująco:

- losowe próbkowanie bez zwracania: 2, 1, 3, 4, 0;
- losowe próbkowanie ze zwracaniem: 1, 3, 3, 4, 1.

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 szumy pochodzące z poszczególnych drzew. Jedynym 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 6., „Najlepsze metody oceny modelu i strojenie parametryczne” — takie jak rozmiar n próbek początkowych (etap 1.) oraz d , czyli liczba losowo dobieanych cech do każdego rozgałęzienia (podetap 2.1). Za pomocą rozmiaru próbek n kontrolujemy kompromis pomiędzy obciążeniem a wariancją losowego lasu.

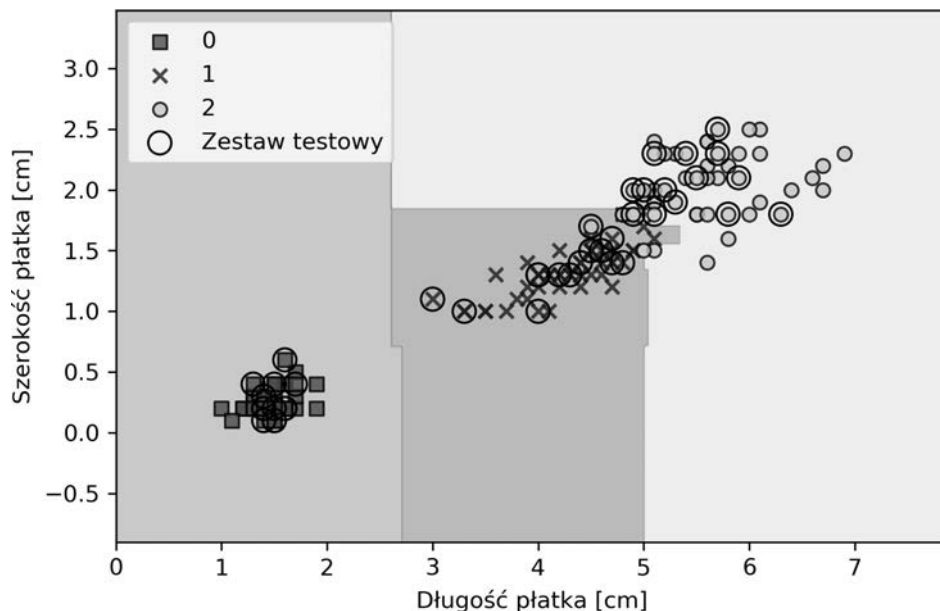
Zmniejszenie rozmiaru przykładów początkowych zwiększa zróżnicowanie pomiędzy poszczególnymi drzewami, ponieważ maleje prawdopodobieństwo wystąpienia określonego przykładu uczącego w danej próbce początkowej. Zatem ograniczenie rozmiaru próbek początkowych skutkuje zwiększeniem **losowości** lasu, co z kolei wpływa pozytywnie na zjawisko przetrenowania. Z drugiej strony mniejszy rozmiar przykładów początkowych zazwyczaj oznacza zmniejszenie skuteczności modelu losowego lasu, niewielką różnicę pomiędzy skutecznością uczenia i testowania, ale generalnie gorszą wydajność względem zbioru testowego. Z kolei zwiększenie rozmiaru przykładu początkowego może podnieść stopień przetrenowania. W miarę upływu czasu przykłady początkowe, a zatem i poszczególne drzewa decyzyjne, stopniowo upodabniają się do siebie i coraz ściślej dopasowują się do pierwotnego zestawu danych uczących.

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. Rozsądną 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, gdyż w interfejsie scikit-learn istnieje implementacja, którą możemy wykorzystać:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> forest = RandomForestClassifier(criterion='gini',
...                               n_estimators=25,
...                               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 3.22 wykres regionów decyzyjnych tworzących zestawy drzew w losowym lesie.



Rysunek 3.22. Wykres regionów decyzyjnych utworzonych za pomocą algorytmu losowego lasu

Stworzyliśmy w tym przypadku losowy las składający się z 25 drzew (parametr `n_estimators`) oraz wykorzystaliśmy wskaźnik Gini jako kryterium zanieczyszczenia do tworzenia rozgałęzień. Mimo że generujemy bardzo mały las dla naszego niewielkiego zbioru danych, w celach demonstracyjnych wprowadziliśmy 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

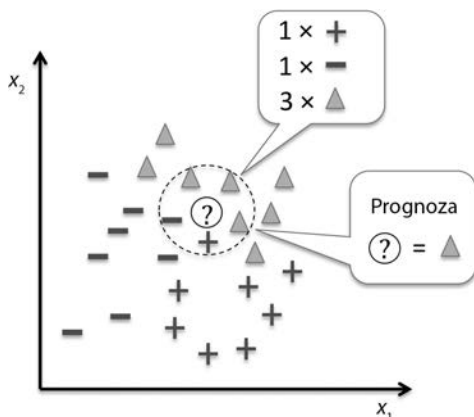
Algorytmy uczenia maszynowego możemy podzielić na modele **parametryczne** i **nieparametryczne**. Za pomocą modeli parametrycznych oszacowujemy parametry zestawu uczącego, dzięki czemu jesteśmy w stanie wytrenować funkcję, która będzie klasyfikowała nowe punkty danych bez konieczności dalszego wykorzystywania 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 3.23 pokazujemy, w jaki sposób nowy punkt danych (?) otrzymuje etykietę klasy — trójkąt — na podstawie większościowego głosowania pomiędzy pięcioma najbliższymi sąsiadami tej próbki.



Rysunek 3.23. Model k-najbliższych sąsiadów

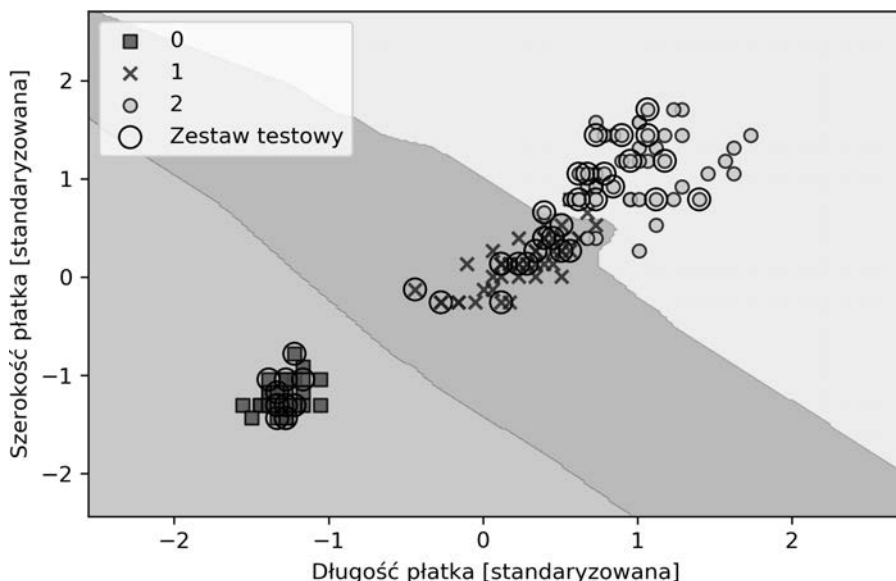
Na podstawie wybranej metryki odległości algorytm KNN wyszukuje w zestawie danych uczących k próbek znajdujących się najbliżej klasyfikowanego punktu lub wykazujących największe podobieństwo do niego. Etykieta klasy tej próbki zostaje określona poprzez większościowe głosowanie przeprowadzone pomiędzy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

Największą zaletą takiego pamięciowego algorytmu jest natychmiastowe adaptowanie się klasyfikatora w trakcie pobierania nowych danych uczących. Równowagę to jednak główna wada, polegająca na liniowym wzroście złożoności obliczeniowej wraz z liczbą próbek uczących (najgorszy 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 płotka [standaryzowana]')
>>> plt.ylabel('Szerokość płotka [standaryzowana]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Wyszukawszy za pomocą tego algorytmu pięciu sąsiadów dla naszego zestawu danych, uzyskujemy w miarę gładką granicę decyzyjną, co zostało zaprezentowane na rysunku 3.24.



Rysunek 3.24. 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 odległość od sąsiadów jest zbliżona, 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 kosaćca) 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:

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

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>.

Kłątwa wymiarowości

Należy wspomnieć, że algorytm KNN jest bardzo wrażliwy na przetrenowanie z powodu **kłątwy 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 regularyzacji jako jednego ze sposobów uniknięcia nadmiernego dopasowania. Jednak w modelach, wobec których nie jesteśmy w stanie wprowadzić regularyzacji (np. drzewach decyzyjnych i algorytmie KNN), możemy skorzystać z technik wyboru cech i redukcji wymiarowości, pozwalających zminimalizować ryzyko wystąpienia kłątwy wymiarowości. Opisanie 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 zadbaamy o właściwe interpretowanie danych. Model regresji logistycznej okazuje się przydatny nie tylko do uczenia przyrostowego za pomocą stochastycznego spadku wzdłuż 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 stają się one atrakcyjnymi modelami 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 informatywnych i rozróżnialnych cech.

W następnym 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.

Skorowidz

A

- adaptacyjne neurony liniowe, 53
- aglomeracyjna analiza skupień, 339
- agregacja, 226–228
- AI, artificial intelligence, 348
- aktualizowanie klasyfikatora, 284
- aktywacja
 - neuronu, 350
 - sieci neuronowej, 354
- algorytm
 - pełnego wiązania, 333
 - pojedynczego wiązania, 333
- algorytm
 - AdaBoost, 231, 236
 - Adaline, 56, 77
 - BPTT, 484
 - centroidów, 320
 - DBSCAN, 340, 341
 - drzewa decyzyjnego, 104, 315
 - głosowania większościowego, 213
 - gradientu prostego, 55, 82
 - imputacji EM, 259
 - jądrowej analizy głównych składowych, 178
 - k-means+ +, 324
 - k-najbliższych sąsiadów, 109, 127
 - Lancaster, 250
 - losowego lasu, 109, 140, 314
 - minimalnego filtrowania Winograda, 452
 - Portera, 250
 - propagacji wstecznej, 375
 - RANSAC, 301
 - regresji logistycznej, 80
 - rozmytych c-średnich, 325
 - sekwencyjnej selekcji wstecznej, 135
 - Snowball, 250
 - spadku wzdłuż gradientu, 62
 - uczenia perceptronu, 45
 - word2vec, 256
 - wstecznej propagacji, 374
 - wzmocnienia, 232
- algorytm
 - klasyfikujące, 68
 - sekwencyjnego wyboru cech, 135
 - sieciowe, 253
 - uczenia maszynowego, 39
 - wyboru cech, 139
- alokacja ukrytej zmiennej Dirichleta, 256
- analiza
 - dyskryminacyjna liniowa, LDA, 154
 - głównych składowych
 - głównych składowych, PCA, 144, 152, 155, 163, 171, 178, 293
 - LDA, 161, 162, 257, 258
 - nieliniowych relacji, 314
 - PCA, 144, 171, 174
 - profilu, 328
 - regresji, 26, 287, 430
 - sentymentów, 241
 - skupień, 29, 319, 320, 325, 335, 339
 - aglomeracyjna, 339
 - grafowa, 345
- API, 45
- aplikacja
 - Graphviz, 105
 - klasyfikatora filmowego, 283

aplikacje sieciowe, 263, 269
 implementacja, 277
 klasyfikator recenzji, 275
 tworzenie, 269
 umieszczanie na serwerze, 282

B

baza danych SQLite, 266
 biblioteka
 Flask, 269
 LIBLINEAR, 92, 300
 LIBSVM, 92
 NLTK, 250, 264
 NumPy, 150
 scikit-learn, 67, 300
 model regresji logistycznej, 84
 uczenie modelu, 84
 Seaborn, 292
 TensorFlow, 381, 383, 409
 biegunowość tekstu, 242
 blok rozmywania, 326
 blokada interpretera, 382
 błąd, 200
 klasyfikacji, 51, 100
 średniokwadratowy, MSE, 305, 423
 brakujące dane, 115, 118
 bramka
 wejściowa, 487
 wyjściowa, 487
 zapominająca, 487

C

cechy
 dobór, 129
 nominalne, 120
 kodowanie gorącojedynkowe, 122
 ocenianie istotności, 140
 odkrywanie, 135
 porządkowe, 120
 mapowanie, 121
 skalowanie, 127
 sztuczne, 123
 transformacja, 149
 wybór, 135
 centroid, 320
 CNN, convolutional neural networks, 441
 CSS, Cascading Style Sheets, 273
 CSV, comma-separated values, 116

częstość
 odwrotna, 246
 termów, 245

D

dane
 Iris, 31, 48
 nierozdzielne liniowo, 93, 164
 sekwencyjne, 478
 tabelaryczne, 116
 tekstowe
 oczyszczanie, 248
 definiowanie
 neuronu, 41
 zmiennych, 417
 dendrogram, 333, 337
 z mapą cieplną, 339
 diagnozowanie problemów, 190
 długość
 działki, 48
 płatka, 48
 dobór
 algorytmu, 196
 dyskryminant liniowych, 159
 funkcji aktywacji, 400
 modelu, 185
 modelu predykcyjnego, 34
 odpowiednich cech, 129
 pierwotnych centroidów, 324
 dodawanie stylu, 273
 dokładność, 200
 klasyfikacji modelu, 71
 klasyfikatora, 138
 dominanta, 214
 dostrajanie modeli, 195
 drzewo
 decyzyjne, 99, 106, 314
 budowanie, 103
 łączenie, 107
 katalogów, 277
 klastrow, 333
 dyskryminanty liniowe, 159
 dysproporcja klas, 205
 działanie
 agregacji modeli, 228
 algorytmu
 AdaBoost, 233
 DBSCAN, 341
 gradientu prostego, 55
 wstecznej propagacji, 378

biblioteki TensorFlow, 409
 estymatora, 120
 kolejki Pipeline, 185
 liniowej analizy dyskryminacyjnej, 156
 metody zespołowej, 211

E

eksploracyjna analiza danych, 292
 entropia, 100
 epoka, 44
 estymator, 120, 183
 interfejsu scikit-learn, 119
 etykieta klas, 25, 121
 ewaluacja
 klasyfikatora zespołowego, 221
 modeli, 34

F

Flask, 269
 format CSV, 116
 formularz
 recenzji, 280
 sprawdzanie, 271
 wyświetlanie, 271
 funkcja aktywacji, 350
 cumsum, 148
 decyzyjna, 41
 kosztu, 55, 78, 131, 371
 liniowa, 406
 logitowa, 75
 plot_decision_regions, 152
 podobieństwa, 96
 progowa, 350
 regularyzacji L2, 131
 ReLU, 406
 signum, 406
 skoku jednostkowego, 41, 406
 tangensa hiperbolicznego, 400, 403, 406
 ujemnego logarytmu wiarygodności, 400
 wiarygodności, 82
 funkcje
 aktywacji, 354, 400, 406
 jądra, 96, 164, 167
 logistyczne, 400, 406
 sigmoidalne, 75, 400
 transformujące, 183

G

globalna blokada interpretera, 382
 głosowanie
 większościowe, 213, 214, 219
 ze względną większością głosów, 210
 główne składowe, 144
 gradient prosty, 55, 127
 graf
 obliczeniowy, 412
 realizacja obiektów, 426
 sieci spłotowej, 466
 grafowa analizie skupień, 345
 granica decyzyjna, 97
 GRU, gated recurrent unit, 488
 grupowanie, 29
 obiektów, 320
 skupień, 333
 grupy, 29

H

hierarchie cech, 442
 hiperparametry, 59, 352
 hiperpłaszczyzna, 89

I

iloczyn
 macierzowo-wektorowy, 58
 skalarny wektorów, 41, 48, 166
 iloraz szans, 74
 implementacja
 Adaline, 80
 algorytmu Adaline, 56
 analizy LDA, 161
 jądrowej analizy, 168
 klasyfikatora głosowania większościowe, 214
 makro, 273
 perceptronu wielowarstwowego, 362
 programu, 277
 sieci
 CNN, 461, 471
 neuronowej, 380
 rekurencyjnej, 502
 wielowarstwowej neuronowej, 347
 wielowarstwowej RNN, 488
 imputacja z użyciem średniej, 118
 indeksy zanieczyszczenia, 104
 inicjowanie zmiennych, 419

instalacja środowiska Python, 35
 interfejs
 API, 45
 biblioteki TensorFlow, 391
 Keras, 395
 Layers, 392, 471
 sieci CNN, 471
 scikit-learn, 92
 estymatory, 119
 TensorFlow, 387
 implementowanie sieci CNN, 461
 istotność cech, 141

J

jądro
 radialnej funkcji bazowej, 96, 167
 SVM, 93, 97
 jądrowa analiza głównych składowych, 163, 178
 jednostka
 GRU, 488
 LSTM, 486
 obciążenia, 352
 jednowarstwowa sieć neuronowa, 349
 jednowymiarowa regresja liniowa, 288

K

kanal, 454
 katalogi, 277
 kategorie modelowania sekwencji, 479
 kernelizacja, 93
 k-krotna krosvalidacja, 185
 k-krotny sprawdzian krzyżowy, 184, 186
 metoda LOOCV, 188
 schemat, 187
 klasa
 PolynomialFeatures, 309
 SentimentRNN, 495, 500
 klasteryzacja, 29
 bazująca na prototypach, 320
 danych, 344
 hierarchiczna, 333
 spektralna, 345
 twarda i miękka, 325
 klastry, 29
 klasy
 negatywne, 26
 pozytywne, 26
 transformujące, 119

klasyfikacja, 25, 39
 binarna, 25
 miękkiego marginesu, 90
 nienadzorowana, 29
 wieloklasowa, 26, 204
 klasyfikator, 439
 słaby, 232
 uczenia maszynowego, 67
 zespołowy
 ewaluacja, 221
 strojenie, 221
 klasyfikowanie
 głosowania większościowego, 214
 obrazów, 441
 pisma odręcznego, 356
 recenzji, 275
 tekstu, 251

kłątwa wymiarowości, 112, 144, 344

kodowanie
 cech nominalnych, 122
 etykiet klas, 121
 gorącojedynkowe, 122
 kolejkovanie, 181
 komórka LSTM, 486
 kompresja danych, 30, 143, 154
 konfigurowanie
 bazy danych, 266
 formularza recenzji, 280
 konsensus próby losowej, 301
 kontaminacja modelu, 226
 koszt uczenia modelu, 427
 krzywa
 charakterystyki roboczej odbiornika, ROC, 202
 precyzji-pełności, 202
 uczenia, 190
 walidacji, 190, 193

L

las, 107
 LDA, linear discriminant analysis, 154
 leniwy klasyfikator, 109
 linia regresji, 288
 liniowa
 analiza dyskryminacyjna, LDA, 154, 155
 dyskryminacja Fishera, 155
 logistyczna funkcja kosztu, 371
 lokalne pole recepcyjne, 443
 LSTM, 486

Ł

łączenie
 klasyfikatorów, 213
 modeli, 209
 łokieć, 328

M

macierz
 błędu, 377
 korelacji, 293
 kowariancji, 159, 165
 odległości, 335
 pomyłek, 198
 rozproszenia, 157, 292
 rzadka, 123
 rzutowania, 150
 wag, 483
 wiązań, 337
 maksymalizacja
 marginesu, 88
 przyrostu informacji, 100
 mapa
 cech, 442, 455
 ciepła, 338
 mapowanie cech porządkowych, 121
 maszyna wektorów nośnych, SVM, 88, 92, 163, 196, 318
 medoid, 320
 menedżer pakietów, 36
 metoda
 build, 495, 507
 dropna, 117
 elastycznej siatki, 307
 gradientu prostego, 296
 karcenia kosztów, 90
 łokcia, 327
 minus jednego elementu, 188
 najmniejszych kwadratów, 296, 387
 porzucania, 457
 predict, 500
 predict_proba, 219
 przeszukiwania siatki, 195
 sample, 510
 średnich połączeń, 334
 toarray, 123
 train, 499, 509
 Warda, 334
 wydzielania, 185, 186

metody
 aglomeracyjne, 333
 deglomeracyjne, 333
 gradientu prostego, 55
 jądrowe, 93
 oceny modelu, 181
 regresji regularyzowanych, 307
 zespołowe, 209
 metryki zliczające, 204
 mikrośrodowisko, 269
 minimalizacja funkcji kosztu, 55
 model
 Adaline, 54, 127, 349
 CharRNN, 512
 gradientu prostego, 70
 k-najbliższych sąsiadów, 110
 leniwego uczenia, 109
 losowego lasu, 107
 neuronu, 40
 perceptronu, 45
 regresji liniowej, 289
 regresji logistycznej, 77, 84, 251
 regresyjny, 423
 worka słów, 244
 modele
 drzew decyzyjnych, 99
 nieparametryczne, 110
 parametryczne, 110
 predykcyjne, 34
 modelowanie
 danych sekwencyjnych, 478
 języka, 502
 nieliniowych zależności, 310
 prawdopodobieństwa, 74
 predykcyjne, 32, 33
 tematyczne, 256
 złożonych funkcji, 348

N

nadmierne dopasowanie, 72
 neuron McCullocha-Pittsa, 40, 348
 neurony liniowe, 53
 niedotrenowanie, 86, 193
 nieliniowe
 granice decyzyjne, 97
 zależności, 310
 NLP, natural language processing, 241
 normalizacja, 127
 wsadowa, 359
 notacja macierzowa, 31

O

- obciążenie, 191
 - jednostkowe, 41
- obiekt DataFrame, 290
- obliczanie
 - funkcji kosztu, 371
 - macierzy rozproszenia, 157
- ocena
 - istotności wyrazów, 246
 - skuteczności modelu, 184, 304
- oczyszczanie danych tekstowych, 248
- odkrywanie cech, 135, 143
- odległość Minkowskiego, 112
- odzworowanie nowego punktu danych, 178
- optymalna liczba skupień, 327

P

- pakiety, 36
- parametr regularyzacji, 87
- PCA, principal component analysis, 144
- pełność, 201
- perceptron, 43, 68
 - trenowanie modelu, 48
 - wielowarstwowy, 351, 362
- platforma Anaconda, 36
- plik app.py, 277
- pliki, 277
- pochodna cząstkowa, 82
- podobieństwo pomiędzy obiektami, 321
- podpróbkiwanie, 452
- podzbiory uczące i testowe, 124
- pojemność, 457
- porównanie
 - dopasowań, 312
 - regionów decyzyjnych, 224, 238
- porzucanie, 457
- pozytywna etykieta, 201
- pozytywne zdarzenie, 74
- prawdopodobieństwo
 - przynależności do klas, 402
 - warunkowe, 74
- precyzja, 201
- problem
 - niedotrenowania, 193
 - przetrenowania, 193
 - z obciążeniem, 190
 - z wariancją, 190
- procesor, 382
 - graficzny, 380, 382

- prognozy, 219
- projekt
 - analiza sentymentów
 - dane, 489
 - klasa SentimentRNN, 500
 - metoda build, 495
 - metoda predict, 500
 - metoda train, 499
 - model, 494
 - optymalizowanie modelu, 501
 - uczenie modelu, 501
 - wektor właściwościowy, 492
 - modelowanie języka
 - build, 507
 - dane, 503
 - konstruktor, 506
 - model CharRNN, 512
 - przetwarzanie znaków, 506
 - sample, 510
 - train, 509
 - tryb próbkowania, 512
 - uczenie, 512
 - sieci RNN
 - analiza sentymentów, 489
 - modelowanie języka, 502
- propagacja
 - jednokierunkowa, 376
 - w przód, 354
- prostowana jednostka liniowa, ReLU, 405
- próbki początkowe, 226
- przekształcanie
 - klasyfikatora, 275
 - słów, 245
 - tensorów, 430
- przestrzeń cech, 161
- przeszukiwanie siatki, 195
- przetrenowanie, 72, 86, 129, 191
- przetwarzanie
 - danych kategoryzujących, 119
 - języka naturalnego, NLP, 241
 - tekstu, 242
 - tekstu na znaczniki, 249
 - wstępne, 243
- przewidywanie, 34, 287
- przyrost informacji, 99, 100, 314
- punkt
 - graniczny, 340
 - rdzeniowy, 340
- punkty zaszumienia, 340

R

rdzeniowanie wyrazów, 250
 redukcja
 wariancji, 315
 wymiarowości, 143, 144
 redukowanie
 kłutwy wymiarowości, 138
 wymiarowości, 30
 regresja, 25, 26, 316
 grzbietowa, 307
 liniowa, 27, 288
 jednowymiarowa, 288
 wielowymiarowa, 288
 logistyczna, 74, 77, 80, 92, 251
 metodą LASSO, 307
 wielomianowa, 308
 regularyzacja, 86, 87
 L1, 129
 rozwiązania rzadkie, 131
 L2, 129
 interpretacja geometryczna, 130
 rekurencyjna sieć neuronowa, 355
 rekurencyjne sieci neuronowe, RNN, 477
 relacje
 jeden-do-wielu, 480
 wiele-do-jednego, 480
 wiele-do-wielu, 480
 ReLU, Rectified Linear Unit, 405
 repozytorium
 Python Package Index, 35
 uczenia maszynowego UCI, 182
 ROC, receiver operating characteristic, 202
 rodzaje uczenia maszynowego, 24
 rozdzielanie
 koncentrycznych kręgów, 172
 sierpowatych kształtów, 169
 rozdzielność liniowa klas, 44
 rozkładanie dokumentów tekstowych, 257
 rozmycie, 326
 rozpoznawanie opinii, 242
 równanie normalne, 301
 różniczkowanie automatyczne, 374
 rzędy, 410
 rzutowanie
 próbek, 161, 172
 punktów danych, 175

S

SBS, Sequential Backward Selection, 135
 schemat
 działania algorytmu AdaBoost, 233
 działania metody zespołowej, 211
 k-krotnego sprawdzianu krzyżowego, 187
 metody wydzielania, 186
 modelu regresji liniowej, 289
 zagnieżdżonego sprawdzianu krzyżowego, 197
 sekwencje, 478
 sekwencyjna selekcja wsteczna, 144
 serializacja wyuczonych estymatorów, 264
 sieci neuronowe
 jednowarstwowe, 349
 rekurencyjne, RNN, 355, 477
 algorytm BPTT, 484
 budowanie modelu, 494
 implementowanie, 502
 macierze wag, 483
 modelowanie sekwencji, 480
 obliczanie aktywacji, 482, 484
 optymalizowanie, 501
 problem zanikających gradientów, 485
 projekt analizy sentymentów, 489
 projekt modelowania języka, 502
 przepływ danych, 480
 sekwencje, 478
 struktura, 480
 uczenie, 501
 wielowarstwowe, 488, 489
 równoległe przetwarzanie, 381
 splotowe, CNN, 440, 441
 hierarchie cech, 442
 implementacja, 459
 implementowanie, 461, 471
 konstruowanie, 454
 lokalne pole recepcyjne, 443
 podpróbkowanie, 452
 splot dyskretny, 444
 warstwy łączące, 443
 warstwy podpróbkowania, 443
 warstwy splotowe/konwolucyjne, 443
 wielowarstwowe, 347, 459
 sigmoidalna funkcja logistyczna, 75
 silnik szablonowania Jinja2, 273
 skalowanie cech, 60, 127
 skupienia, 29, 333
 skuteczność uczenia, 382
 słownik, 415

splot
 dyskretny, 444
 w dwóch wymiarach, 449
 macierzy, 450
 splotowa sieć neuronowa, CNN, 440
 sprawdzanie algorytmów, 190
 sprawdzian krzyżowy, 184, 186
 zagnieżdżony, 196
 SQLite, 266
 standaryzacja, 60, 127
 danych, 70
 stochastyczny spadek wzdłuż gradientu, SGD,
 62, 254, 296, 351
 stosowanie algorytmu AdaBoost, 236
 strojenie
 hiperparametrów, 195
 klasyfikatora zespołowego, 221
 parametryczne, 181
 struktura
 katalogowa, 272
 sieci RNN, 480
 style CSS, 273
 suma kwadratów błędów, 55, 130, 296, 350, 388
 SVM, support vector machine, 88
 systemy uczenia maszynowego, 32
 szablon strony, 281
 szacowanie współczynnika modelu regresji, 300
 szerokość
 działki, 99
 marginesu, 91
 sztuczka z funkcją jądra, 95, 164
 sztuczna inteligencja, 24, 348

Ś

średnie odchylenie bezwzględne, 302
 środowisko Flask, 269

T

tablice, 386
 technika
 OvA, 49
 OvR, 133
 tekst
 klasyfikowanie, 251
 oczyszczanie, 248
 przetwarzanie, 242
 przetwarzanie na znaczniki, 249
 rozkładanie dokumentów, 257
 worek słów, 244

tensor, 385, 410
 trójwymiarowy, 373
 TensorFlow, 381
 funkcje aktywacji, 400
 funkcje biblioteki, 410
 grafy obliczeniowe, 412
 implementacja sieci CNN, 459
 implementacja sieci RNN, 488
 interfejsy, 391
 koszt uczenia, 427
 mechanizm działania, 409
 mechanizm przebiegu sterowania, 433
 modele regresyjne, 423
 przekształcanie tensorów, 430
 skuteczność uczenia, 382
 tworzenie grafów, 433
 wczytywanie modelu, 428
 węzły zastępcze, 414
 wizualizowanie grafów, 436
 zapisywanie modelu, 428
 zmienne, 417
 token, 244
 tokenizacja, 249
 transformacja cech, 149
 trenowanie, 34
 modelu perceptronu, 48
 sztucznej sieci neuronowej, 371
 trwałość modelu, 264
 twarda analiza skupień, 325
 tworzenie
 aplikacji sieciowej, 269
 dobrych zbiorów uczących, 115
 drzewa decyzyjnego, 103
 grafów, 433
 modelu regresyjnego, 423
 sieci CNN, 454
 strony, 274
 systemów uczenia maszynowego, 32
 szablonu strony, 281
 wielowarstwowych sieci neuronowych, 392
 zespołu klasyfikatorów, 226
 zestawu danych, 120

U

uczenie
 długofalowych oddziaływań, 485
 drzew decyzyjnych, 99
 głębokie, 352
 leniwe, 109

- maszynowe, 23
 - środowisko Python, 35
 - wielkoskalowe, 62
- modelu regresji logistycznej, 84
- nadzorowane, 25, 287
- nienadzorowane, 29, 319
- perceptronu, 43, 45, 68
- pozardzeniowe, 253
- przez wzmacnianie, 28
- przyrostowe, 63
- z pomocą minipaczek, 63
- z przykładów, 110
- zespolowe, 209
- usługa PythonAnywhere, 282
- usuwanie
 - pomijanych słów, 251
 - próbek, 117
- użycie losowego lasu, 316

W

- wagi logistycznej funkcji kosztu, 78
- walidacyjny zbiór danych, 138
- wariancja, 129, 191
 - całkowita, 148
- warstwy
 - łączące, 443
 - podpróbkowania, 443
 - splotowe/konwolucyjne, 443
- wartość
 - modalna, 214
 - NaN, 118
- ważenie, 214
 - częstości termów, 246
- wczytywanie danych, 460
- wdrażanie modelu uczenia maszynowego, 263
- wektor, 31
 - cech, 245
 - nośny, 88
 - własny, 122, 147
 - właściwościowy, 492
- wektoryzacja, 355
- wewnątrzgrupowa suma kwadratów błędów, 322, 326
- węzły zastępcze, 414
 - dla tablic, 416
- wielkoskalowe uczenie maszynowe, 62
- wielowarstwowa
 - architektura sieci neuronowych, 351
 - sieć CNN, 459
 - sieć rekurencyjna, 488, 489
- wielowymiarowa regresja liniowa, 288
- większościowe głosowanie, 210
- wizualizowanie
 - elementów zbioru, 292
 - grafów, 436
 - podsieci generatora, 438
- worek słów, 244
- wsadowa gradientu prostego, 56
- wskaźnik Giniego, 100
- współczynnik
 - dwumianowy, 212
 - korelacji liniowej Pearsona, 294
 - profilu, 328
 - uczenia, 59, 350
 - wariancji wyjaśnionej, 148
- wstawianie brakujących danych, 118
- wstępne przetwarzanie danych, 22, 115, 127, 460
- wybór
 - algorytmu klasyfikującego, 68
 - cech, 135
- wydobywanie głównych składowych, 146
- wykres
 - danych rzutowanych, 151
 - dokładności, 369
 - dokładności uczenia i walidacji, 192
 - funkcji kosztu, 79, 368
 - funkcji logistycznej, 76
 - granic decyzyjnych, 104
 - ilości informacji, 160
 - istotności cech, 141
 - klasyfikatora regresji logistycznej, 163
 - kosztu, 298
 - krzywej ROC, 202, 204
 - łokcia, 328
 - porównujący regresje, 310
 - profilu, 330, 332
 - regionów decyzyjnych, 53, 72, 109, 153, 154
 - regresji, 315
 - regresji logistycznej, 162
 - regularyzacji, 88
 - regularyzacji L1, 132
 - resztowy, 304
 - ROC klasyfikatora zespolowego, 222
 - rzutowania próbek, 172
 - skupionych danych, 321
 - współczynników wariancji, 149
 - wyników uczenia perceptronu, 73
 - wypukłej funkcji kosztu, 130
 - zbieżności algorytmu, 51
- wykrywanie brakujących wartości, 116

wyuczone estymatory, 264
wzmocnienie, 231, 232
 adaptacyjne, 231

Z

zagnieżdżony sprawdzian krzyżowy, 196
zakres zmiennych, 420
zanieczyszczenie, 100, 104
zapobieganie przetrenowaniu, 86
zbieżność, 378
 uczenia, 53
zbiór danych
 dwuwymiarowy, 26
 Housing, 290, 310
 IMDb, 241, 489
 Iris, 91, 99, 219

 losowo zaszumiony, 94
 MNIST, 357
 testowych, 34
 walidacyjny, 137
 Wine, 124, 125, 163, 228
zestaw danych, *Patrz* zbiór danych
złożoność modelu, 129
zmienna Dirichleta, 256
zmienne, 417
 definiowanie, 417
 docelowe, 288
 inicjowanie, 419
 objaśniające, 288
 uzupełniające, 90
 wielokrotne wykorzystywanie, 421
zakres, 420

PROGRAM PARTNERSKI

— GRUPY 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
Helion 

Uczenie maszynowe: oto droga do wiedzy ukrytej w oceanie danych!

Uczenie maszynowe jest wyjątkowo fascynującą dziedziną inżynierii. Coraz częściej spotykamy się z praktycznym wykorzystaniem tego rodzaju innowacyjnych technologii. Samouczące algorytmy maszynowe pozwalają na uzyskiwanie wiedzy z ogromnych ilości danych. Dla osoby planującej rozwój kariery osiągnięcie biegłości w rozwiązywaniu problemów uczenia maszynowego jest nadzwyczaj atrakcyjną ścieżką. Użycie do tego celu Pythona pozwala dodatkowo skorzystać z bardzo przystępnego, wszechstronnego i potężnego narzędzia przeznaczonego do analizowania danych naukowych.

Ta książka jest drugim, wzbogaconym i zaktualizowanym wydaniem znakomitego podręcznika do nauki o danych. Wyczerpująco opisano tu teoretyczne podwaliny uczenia maszynowego. Sporo uwagi poświęcono działaniu algorytmów uczenia głębokiego, sposobom ich wykorzystania oraz metodom unikania istotnych błędów. Dodano rozdziały prezentujące zaawansowane informacje o sieciach neuronowych: o sieciach spłotowych, służących do rozpoznawania obrazów, oraz o sieciach rekurencyjnych, znakomicie nadających się do pracy z danymi sekwencyjnymi i danymi szeregów czasowych. Poszczególne zagadnienia zostały zilustrowane praktycznymi przykładami kodu napisanego w Pythonie, co ułatwi bezpośrednie zapoznanie się z tematyką uczenia maszynowego.

W tej książce:

- struktury używane w analizie danych, uczeniu maszynowym i uczeniu głębokim
- metody uczenia sieci neuronowych
- implementowanie głębokich sieci neuronowych
- analiza sentymentów i analiza regresyjna
- przetwarzanie obrazów i danych tekstowych
- najwartościowsze biblioteki Pythona przydatne w uczeniu maszynowym

Dr Sebastian Raschka jest ekspertem w dziedzinie analizy danych i uczenia maszynowego. Naukowo zajmuje się głównie metodami obliczeniowymi w biologii statystycznej. Chętnie bierze udział w różnych projektach *open source* i wdraża nowe metody uczenia maszynowego.

Dr Vahid Mirjalili zajmuje się stosowaniem uczenia maszynowego w rozpoznawaniu obrazów i zwiększaniu prywatności przy użyciu danych biometrycznych. Projektuje też modele sieci neuronowych, które mają ułatwiać wykrywanie pieszych przez pojazdy autonomiczne.

Helion 	<i>Sprawdź nasze szkolenia!</i>	KOD KORZYŚCI Sięgnij po więcej! ▶	
 helion.pl	SZKOLENIA 	ISBN 978-83-283-5121-9	
 0 801 339900	AKADEMIA IT & BUSINESS		
 0 601 339900	WWW.SZKOLENIA.HELION.PL	9 788328 351219	
INFORMATYKA W NAJLEPSZYM WYDANIU		Cena: 99,00 zł	

Packt