

Sebastian Raschka, Vahid Mirjalili

Python

Machine learning
i deep learning

Biblioteki scikit-learn
i TensorFlow 2

Helion 

Packt 

Tytuł oryginału: Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow 2, 3rd Edition

Tłumaczenie: Krzysztof Sawka

ISBN: 978-83-283-7001-2

Copyright © Packt Publishing 2019. First published in the English language under the title 'Python Machine Learning - Third Edition - (9781789955750)'

Polish edition copyright © 2021 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/pythu3>

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	13
Informacje o recenzentach	15
Wstęp	17
Rozdział 1. Umożliwianie komputerom uczenia się z danych	27
Tworzenie inteligentnych maszyn służących do przekształcania danych w wiedzę	28
Trzy różne rodzaje uczenia maszynowego	28
Przewidywanie przyszłości za pomocą uczenia nadzorowanego	29
Rozwiązywanie problemów interaktywnych za pomocą uczenia przez wzmacnianie	32
Odkrywanie ukrytych struktur za pomocą uczenia nienadzorowanego	33
Wprowadzenie do podstawowej terminologii i notacji	35
Notacja i konwencje używane w niniejszej książce	35
Terminologia uczenia maszynowego	37
Strategia tworzenia systemów uczenia maszynowego	38
Wstępne przetwarzanie — nadawanie danym formy	38
Trenowanie i dobór modelu predykcyjnego	39
Ewaluacja modeli i przewidywanie wystąpienia nieznanych danych	40
Wykorzystywanie środowiska Python do uczenia maszynowego	40
Instalacja środowiska Python i pakietów z repozytorium Python Package Index	41
Korzystanie z platformy Anaconda i menedżera pakietów	41
Pakiety przeznaczone do obliczeń naukowych, analizy danych i uczenia maszynowego	42
Podsumowanie	42

Rozdział 2. Trenowanie prostych algorytmów uczenia maszynowego w celach klasyfikacji	45
Sztuczne neurony — rys historyczny początków uczenia maszynowego	46
Definicja formalna sztucznego neuronu	47
Reguła uczenia perceptronu	49
Implementacja algorytmu uczenia perceptronu w Pythonie	51
Obiektowy interfejs API perceptronu	51
Trenowanie modelu perceptronu na zestawie danych Iris	54
Adaptacyjne neurony liniowe i zbieżność uczenia	60
Minimalizacja funkcji kosztu za pomocą metody gradientu prostego	61
Implementacja algorytmu Adaline w Pythonie	63
Usprawnianie gradientu prostego poprzez skalowanie cech	67
Wielkoskalowe uczenie maszynowe i metoda stochastycznego spadku wzdłuż gradientu	69
Podsumowanie	73
Rozdział 3. Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn	75
Wybór algorytmu klasyfikującego	76
Pierwsze kroki z biblioteką scikit-learn — uczenie perceptronu	76
Modelowanie prawdopodobieństwa przynależności do klasy za pomocą regresji logistycznej	82
Regresja logistyczna i prawdopodobieństwo warunkowe	82
Wyznaczanie wag logistycznej funkcji kosztu	86
Przekształcanie implementacji Adaline do postaci algorytmu regresji logistycznej	88
Uczenie modelu regresji logistycznej za pomocą biblioteki scikit-learn	92
Zapobieganie przetrenowaniu za pomocą regularyzacji	94
Wyznaczanie maksymalnego marginesu za pomocą maszyn wektorów nośnych	97
Teoretyczne podłoże maksymalnego marginesu	98
Rozwiązywanie przypadków nieliniowo rozdzielnych za pomocą zmiennych uzupełniających	99
Alternatywne implementacje w interfejsie scikit-learn	101
Rozwiązywanie nieliniowych problemów za pomocą jądra SVM	102
Metody jądrowe dla danych nierozdzielnych liniowo	102
Stosowanie sztuczki z funkcją jądra do znajdowania przestrzeni rozdzielających w przestrzeni wielowymiarowej	104
Uczenie drzew decyzyjnych	107
Maksymalizowanie przyrostu informacji — osiągnięcie jak największych korzyści	108
Budowanie drzewa decyzyjnego	112
Łączenie wielu drzew decyzyjnych za pomocą modelu losowego lasu	115
Algorytm k-najbliższych sąsiadów — model leniwego uczenia	119
Podsumowanie	122
Rozdział 4. Tworzenie dobrych zestawów danych uczących — wstępne przetwarzanie danych	125
Kwestia brakujących danych	126
Wykrywanie brakujących wartości w danych tabelarycznych	126
Usuwanie przykładów uczących lub cech niezawierających wartości	127

Wstawianie brakujących danych	128
Estymatory interfejsu scikit-learn	129
Przetwarzanie danych kategoryalnych	130
Kodowanie danych kategoryalnych za pomocą biblioteki pandas	131
Mapowanie cech porządkowych	131
Kodowanie etykiet klas	132
Kodowanie „gorącojedynkowe” cech nominalnych (z użyciem wektorów własnych)	133
Rozdzielanie zestawu danych na oddzielne podzbiory uczący i testowy	136
Skalowanie cech	138
Dobór odpowiednich cech	140
Regularyzacje L1 i L2 jako kary ograniczające złożoność modelu	141
Interpretacja geometryczna regularyzacji L2	141
Rozwiązania rzadkie za pomocą regularyzacji L1	143
Algorytmy sekwencyjnego wyboru cech	146
Ocenianie istotności cech za pomocą algorytmu losowego lasu	151
Podsumowanie	154
Rozdział 5. Kompresja danych poprzez redukcję wymiarowości	155
Nienadzorowana redukcja wymiarowości za pomocą analizy głównych składowych	156
Podstawowe etapy analizy głównych składowych	156
Wydobywanie głównych składowych krok po kroku	158
Wyjaśniona wariancja całkowita	160
Transformacja cech	161
Analiza głównych składowych w interfejsie scikit-learn	164
Nadzorowana kompresja danych za pomocą liniowej analizy dyskryminacyjnej	167
Porównanie analizy głównych składowych z liniową analizą dyskryminacyjną	167
Wewnętrzne mechanizmy działania liniowej analizy dyskryminacyjnej	169
Obliczanie macierzy rozproszenia	169
Dobór dyskryminant liniowych dla nowej podprzestrzeni cech	171
Rzutowanie przykładów na nową przestrzeń cech	173
Implementacja analizy LDA w bibliotece scikit-learn	174
Jądrowa analiza głównych składowych	
jako metoda odwzorowywania nierozdzielnych liniowo klas	176
Funkcje jądra oraz sztuczka z funkcją jądra	177
Implementacja jądrowej analizy głównych składowych w Pythonie	181
Rzutowanie nowych punktów danych	188
Algorytm jądrowej analizy głównych składowych w bibliotece scikit-learn	191
Podsumowanie	192
Rozdział 6. Najlepsze metody oceny modelu i strojenie parametryczne	195
Usprawnianie cyklu pracy za pomocą kolejkowania	195
Wczytanie zestawu danych Breast Cancer Wisconsin	196
Łączenie funkcji transformujących i estymatorów w kolejce czynności	197
Stosowanie k-krotnego sprawdzianu krzyżowego w ocenie skuteczności modelu	198
Metoda wydzielania	199
K-krotny sprawdzian krzyżowy	200

Sprawdzanie algorytmów za pomocą krzywych uczenia i krzywych walidacji	204
Diagnozowanie problemów z obciążeniem i wariancją za pomocą krzywych uczenia	204
Rozwiązywanie problemów przetrenowania i niedotrenowania za pomocą krzywych walidacji	208
Dostrajanie modeli uczenia maszynowego za pomocą metody przeszukiwania siatki	209
Strojenie hiperparametrów przy użyciu metody przeszukiwania siatki	210
Dobór algorytmu poprzez zagnieżdżony sprawdzian krzyżowy	211
Przegląd wskaźników oceny skuteczności	213
Odczytywanie macierzy pomyłek	213
Optymalizacja precyzji i pełności modelu klasyfikującego	215
Wykres krzywej ROC	217
Wskaźniki zliczające dla klasyfikacji wieloklasowej	220
Kwestia dysproporcji klas	220
Podsumowanie	223
Rozdział 7. Łączenie różnych modeli w celu uczenia zespołowego	225
Uczenie zespołów	225
Łączenie klasyfikatorów za pomocą algorytmu głosowania większościowego	229
Implementacja prostego klasyfikatora głosowania większościowego	230
Stosowanie reguły głosowania większościowego do uzyskiwania prognoz	235
Ewaluacja i strojenie klasyfikatora zespołowego	237
Agregacja — tworzenie zespołu klasyfikatorów za pomocą próbek początkowych	242
Agregacja w pigułce	243
Stosowanie agregacji do klasyfikowania przykładów z zestawu Wine	244
Usprawnianie słabych klasyfikatorów za pomocą wzmocnienia adaptacyjnego	248
Wzmacnianie — mechanizm działania	248
Stosowanie algorytmu AdaBoost za pomocą biblioteki scikit-learn	252
Podsumowanie	255
Rozdział 8. Wykorzystywanie uczenia maszynowego w analizie sentymentów	257
Przygotowywanie zestawu danych IMDb movie review do przetwarzania tekstu	258
Uzyskiwanie zestawu danych IMDb	258
Przetwarzanie wstępne zestawu danych IMDb do wygodniejszego formatu	259
Wprowadzenie do modelu worka słów	260
Przekształcanie słów w wektory cech	261
Ocena istotności wyrazów za pomocą ważenia częstości termów — odwrotnej częstości w tekście	262
Oczyszczanie danych tekstowych	264
Przetwarzanie tekstu na znaczniki	266
Uczenie modelu regresji logistycznej w celu klasyfikowania tekstu	268
Praca z większą ilością danych — algorytmy sieciowe i uczenie pozardzeniowe	270
Modelowanie tematyczne za pomocą alokacji ukrytej zmiennej Dirichleta	273
Rozkładanie dokumentów tekstowych za pomocą analizy LDA	274
Analiza LDA w bibliotece scikit-learn	274
Podsumowanie	277

Rozdział 9. Wdrażanie modelu uczenia maszynowego do aplikacji sieciowej	279
Serializacja wyuczonych estymatorów biblioteki scikit-learn	280
Konfigurowanie bazy danych SQLite	283
Tworzenie aplikacji sieciowej za pomocą środowiska Flask	285
Nasza pierwsza aplikacja sieciowa	285
Sprawdzanie i wyświetlanie formularza	287
Przekształcanie klasyfikatora recenzji w aplikację sieciową	293
Pliki i katalogi — wygląd drzewa katalogów	295
Implementacja głównej części programu w pliku app.py	296
Konfigurowanie formularza recenzji	298
Tworzenie szablonu strony wynikowej	299
Umieszczanie aplikacji sieciowej na publicznym serwerze	301
Tworzenie konta w serwisie PythonAnywhere	301
Przesyłanie aplikacji klasyfikatora filmowego	302
Aktualizowanie klasyfikatora recenzji filmowych	303
Podsumowanie	305
Rozdział 10. Przewidywanie ciągłych zmiennych docelowych za pomocą analizy regresyjnej	307
Wprowadzenie do regresji liniowej	308
Prosta regresja liniowa	308
Wielowymiarowa regresja liniowa	308
Zestaw danych Housing	310
Wczytywanie zestawu danych Housing do obiektu DataFrame	310
Wizualizowanie ważnych elementów zestawu danych	312
Analiza związków za pomocą macierzy korelacji	313
Implementacja modelu regresji liniowej wykorzystującego zwykłą metodę najmniejszych kwadratów	315
Określanie parametrów regresyjnych za pomocą metody gradientu prostego	316
Szacowanie współczynnika modelu regresji za pomocą biblioteki scikit-learn	319
Uczenie odpornego modelu regresyjnego za pomocą algorytmu RANSAC	321
Ocenianie skuteczności modeli regresji liniowej	324
Stosowanie regularyzowanych metod regresji	327
Przekształcanie modelu regresji liniowej w krzywą — regresja wielomianowa	329
Dodawanie członów wielomianowych za pomocą biblioteki scikit-learn	329
Modelowanie nieliniowych zależności w zestawie danych Housing	331
Analiza nieliniowych relacji za pomocą algorytmu losowego lasu	334
Regresja przy użyciu drzewa decyzyjnego	334
Regresja przy użyciu losowego lasu	336
Podsumowanie	339
Rozdział 11. Praca z nieoznakowanymi danymi — analiza skupień	341
Grupowanie obiektów na podstawie podobieństwa przy użyciu algorytmu centroidów	342
Algorytm centroidów w bibliotece scikit-learn	342
Inteligentniejszy sposób dobierania pierwotnych centroidów za pomocą algorytmu k-means++	346
Twarda i miękka analiza skupień	347

Stosowanie metody łockia do wyszukiwania optymalnej liczby skupień	349
Ujęcie ilościowe jakości analizy skupień za pomocą wykresu profilu	351
Organizowanie skupień do postaci drzewa skupień	355
Oddolne grupowanie skupień	356
Przeprowadzanie hierarchicznej analizy skupień na macierzy odległości	357
Dołączanie dendrogramów do mapy cieplnej	360
Aglomeracyjna analiza skupień w bibliotece scikit-learn	361
Wyznaczanie rejonów o dużej gęstości za pomocą algorytmu DBSCAN	363
Podsumowanie	368
Rozdział 12. Implementowanie wielowarstwowej sieci neuronowej od podstaw	369
Modelowanie złożonych funkcji przy użyciu sztucznych sieci neuronowych	370
Jednowarstwowa sieć neuronowa — powtórzenie	371
Wstęp do wielowarstwowej architektury sieci neuronowych	373
Aktywacja sieci neuronowej za pomocą propagacji w przód	376
Klasyfikowanie pisma odręcznego	379
Zestaw danych MNIST	379
Implementacja perceptronu wielowarstwowego	385
Trenowanie sztucznej sieci neuronowej	395
Obliczanie logistycznej funkcji kosztu	395
Wyjaśnienie algorytmu wstecznej propagacji	398
Uczenie sieci neuronowych za pomocą algorytmu propagacji wstecznej	399
Zbieżność w sieciach neuronowych	402
Jeszcze słowo o implementacji sieci neuronowej	404
Podsumowanie	404
Rozdział 13. Równoległe przetwarzanie sieci neuronowych za pomocą biblioteki TensorFlow	407
Biblioteka TensorFlow a skuteczność uczenia	408
Wyzwania związane z wydajnością	408
Czym jest biblioteka TensorFlow?	409
W jaki sposób będziemy poznawać bibliotekę TensorFlow?	411
Pierwsze kroki z biblioteką TensorFlow	411
Instalacja modułu TensorFlow	411
Tworzenie tensorów w TensorFlow	412
Manipulowanie typem danych i rozmiarem tensora	413
Przeprowadzanie operacji matematycznych na tensorach	414
Dzielenie, nawarstwianie i łączenie tensorów	415
Tworzenie potoków wejściowych za pomocą tf.data, czyli interfejsu danych TensorFlow	416
Tworzenie obiektów Dataset z istniejących tensorów	417
Łączenie dwóch tensorów we wspólny zestaw danych	418
Potasuj, pogrupuj, powtórz	419
Tworzenie zestawu danych z plików umieszczonych w lokalnym magazynie dyskowym	422
Pobieranie dostępnych zestawów danych z biblioteki tensorflow_datasets	425
Tworzenie modelu sieci neuronowej za pomocą modułu TensorFlow	430
Interfejs Keras (tf.keras)	430
Tworzenie modelu regresji liniowej	431

Uczenie modelu za pomocą metod <code>.compile()</code> i <code>.fit()</code>	435
Tworzenie perceptronu wielowarstwowego klasyfikującego kwiaty z zestawu danych Iris	436
Ocena wytrenowanego modelu za pomocą danych testowych	439
Zapisywanie i wczytywanie wyuczonego modelu	440
Dobór funkcji aktywacji dla wielowarstwowych sieci neuronowych	440
Funkcja logistyczna — powtórzenie	441
Szacowanie prawdopodobieństw przynależności do klas w klasyfikacji wieloklasowej za pomocą funkcji softmax	443
Rozszerzanie zakresu wartości wyjściowych za pomocą funkcji tangensa hiperbolicznego	444
Aktywacja za pomocą prostowanej jednostki liniowej (ReLU)	446
Podsumowanie	448
Rozdział 14. Czas na szczegóły — mechanizm działania biblioteki TensorFlow	449
Cechy kluczowe TensorFlow	450
Grafy obliczeniowe TensorFlow: migracja do wersji TensorFlow 2	451
Grafy obliczeniowe	451
Tworzenie grafu w wersji TensorFlow 1.x	452
Migracja grafu do wersji TensorFlow 2	453
Wczytywanie danych wejściowych do modelu: TensorFlow 1.x	453
Wczytywanie danych wejściowych do modelu: TensorFlow 2	454
Poprawianie wydajności obliczeniowej za pomocą dekoratorów funkcji	455
Obiekty Variable służące do przechowywania i aktualizowania parametrów modelu	456
Obliczanie gradientów za pomocą różniczkowania automatycznego i klasy GradientTape	460
Obliczanie gradientów funkcji straty w odniesieniu do zmiennych modyfikowalnych	460
Obliczanie gradientów w odniesieniu do tensorów niemodyfikowalnych	462
Przechowywanie zasobów na obliczanie wielu gradientów	462
Upraszczenie implementacji popularnych struktur za pomocą interfejsu Keras	463
Rozwiązywanie problemu klasyfikacji XOR	466
Zwiększenie możliwości budowania modeli za pomocą interfejsu funkcyjnego Keras	471
Implementowanie modeli bazujących na klasie Model	472
Pisanie niestandardowych warstw Keras	473
Estymatory TensorFlow	476
Praca z kolumnami cech	477
Uczenie maszynowe za pomocą gotowych estymatorów	481
Stosowanie estymatorów w klasyfikacji zestawu pisma odręcznego MNIST	484
Tworzenie niestandardowego estymatora z istniejącego modelu Keras	486
Podsumowanie	488
Rozdział 15. Klasyfikowanie obrazów za pomocą głębokich spłotowych sieci neuronowych	489
Podstawowe elementy spłotowej sieci neuronowej	490
Spłotowe sieci neuronowe i hierarchie cech	490
Spłot dyskretny	492
Warstwy próbkowania	501

Implementowanie sieci CNN	502
Praca z wieloma kanałami wejściowymi/barw	503
Regularyzowanie sieci neuronowej metodą porzucania	506
Funkcje straty w zadaniach klasyfikacji	509
Implementacja głębokiej sieci splotowej za pomocą biblioteki TensorFlow	511
Architektura wielowarstwowej sieci CNN	511
Wczytywanie i wstępne przetwarzanie danych	512
Implementowanie sieci CNN za pomocą interfejsu Keras	513
Klasyfikowanie płci na podstawie zdjęć twarzy za pomocą sieci splotowej	518
Wczytywanie zestawu danych CelebA	519
Przekształcanie obrazów i dogenerowanie danych	520
Uczenie modelu CNN jako klasyfikatora płci	525
Podsumowanie	530
Rozdział 16. Modelowanie danych sekwencyjnych za pomocą rekurencyjnych sieci neuronowych	533
<hr/>	
Wprowadzenie do danych sekwencyjnych	534
Modelowanie danych sekwencyjnych — kolejność ma znaczenie	534
Przedstawianie sekwencji	535
Różne kategorie modelowania sekwencji	536
Sieci rekurencyjne służące do modelowania sekwencji	537
Mechanizm zapętlenia w sieciach rekurencyjnych	537
Obliczanie aktywacji w sieciach rekurencyjnych	539
Rekurencja w warstwie ukrytej a rekurencja w warstwie wyjściowej	542
Problemy z uczeniem długofalowych oddziaływań	544
Jednostki LSTM	546
Implementowanie wielowarstwowej sieci rekurencyjnej przy użyciu biblioteki TensorFlow do modelowania sekwencji	548
Pierwszy projekt — przewidywanie sentymentów na recenzjach z zestawu danych IMDb	548
Drugi projekt — modelowanie języka na poziomie znaków w TensorFlow	561
Przetwarzanie języka za pomocą modelu transformatora	572
Wyjaśnienie mechanizmu samouwagi	573
Wieloblokowy mechanizm uwagi i komórka transformatora	575
Podsumowanie	577
Rozdział 17. Generatywne sieci przeciwstawne w zadaniach syntetyzowania nowych danych	579
<hr/>	
Wprowadzenie do generatywnych sieci przeciwstawnych	580
Autokodery	580
Modele generatywne syntetyzujące nowe dane	582
Generowanie nowych prób za pomocą sieci GAN	584
Funkcje straty generatora i dyskryminatora w modelu GAN	585
Implementowanie sieci GAN od podstaw	587
Uczenie modeli GAN w środowisku Google Colab	587
Implementacja sieci generatora i dyskryminatora	590
Definiowanie zestawu danych uczących	593
Uczenie modelu GAN	595

Poprawianie jakości syntetyzowanych obrazów za pomocą sieci GAN:	603
splotowej i Wassersteina	
Splot transponowany	603
Normalizacja wsadowa	605
Implementowanie generatora i dyskryminatora	607
Wskaźniki odmienności dwóch rozkładów	613
Praktyczne stosowanie odległości EM w sieciach GAN	616
Kara gradientowa	617
Implementacja sieci WGAN-DP służącej do uczenia modelu DCGAN	617
Załamanie modu	622
Inne zastosowania modeli GAN	623
Podsumowanie	624
Rozdział 18. Uczenie przez wzmacnianie jako mechanizm podejmowania decyzji w skomplikowanych środowiskach	625
<hr/>	
Wprowadzenie: uczenie z doświadczenia	626
Filozofia uczenia przez wzmacnianie	626
Definicja interfejsu agent-środowisko w systemie uczenia przez wzmacnianie	628
Podstawy teoretyczne uczenia przez wzmacnianie	629
Procesy decyzyjne Markowa	629
Wyjaśnienie matematyczne procesów decyzyjnych Markowa	630
Terminologia uczenia przez wzmacnianie: zwrot, strategia i funkcja wartości	633
Programowanie dynamiczne za pomocą równania Bellmana	636
Algorytmy uczenia przez wzmacnianie	637
Programowanie dynamiczne	638
Uczenie przez wzmacnianie metodą Monte Carlo	641
Uczenie metodą różnic czasowych	643
Implementacja naszego pierwszego algorytmu uczenia przez wzmacnianie	646
Wprowadzenie do pakietu OpenAI Gym	646
Rozwiązywanie problemu świata blokowego za pomocą Q-uczenia	654
Krótko o algorytmie Q-uczenia głębokiego	658
Podsumowanie rozdziału i książki	665

Trenowanie prostych algorytmów uczenia maszynowego w celach klasyfikacji

W tym rozdziale wykorzystamy dwa z najwcześniejszych algorytmów klasyfikacyjnych: modele **perceptronu** oraz **adaptacyjnego neuronu liniowego**. Zaczniemy od zaimplementowania krok po kroku perceptronu w środowisku Python oraz uczenia go klasyfikacji różnych odmian kosańca na podstawie danych z zestawu Iris. W ten sposób lepiej zrozumiemy koncepcję algorytmów klasyfikacyjnych oraz dowiemy się, jak można je skutecznie wdrożyć za pomocą Pythona.

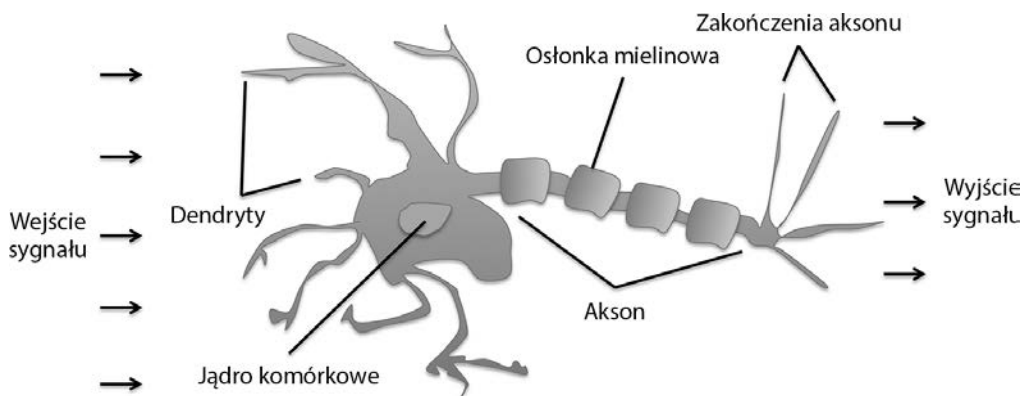
Następnie przyjrzymy się podstawom optymalizacji przy użyciu adaptacyjnych neuronów liniowych, gdyż stanowi to wstęp do stosowania bardziej zaawansowanych klasyfikatorów przechowywanych w bibliotece scikit-learn, co zostało omówione w rozdziale 3., „Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn”.

Zajmiemy się w tym rozdziale omówieniem następujących zagadnień:

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

Sztuczne neurony — rys historyczny początków uczenia maszynowego

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



Rysunek 2.1. Model uproszczonego neuronu

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

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

Definicja formalna sztucznego neuronu

W ujęciu matematycznym możemy analizować koncepcję **sztucznych neuronów** w kontekście zadania klasyfikacji binarnej, w której dla uproszczenia odnosimy się do dwóch klas: 1 (klasy pozytywnej) oraz -1 (klasy negatywnej). Następnie definiujemy **funkcję decyzyjną** ($\phi(z)$), na którą składa się liniowa kombinacja określonych wartości wejściowych \mathbf{x} oraz powiązanego z nimi wektora wag \mathbf{w} , gdzie z nosi nazwę całkowitego pobudzenia układu $z = w_1x_1 + w_2x_2 + \dots + w_mx_m$:

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

Jeżeli całkowite pobudzenie z danej przykładowej wartości $\mathbf{x}^{(i)}$ jest wyższe od zdefiniowanej wartości progowej θ , to przewidujemy, że dany obiekt przynależy do klasy pozytywnej 1, w przeciwnym wypadku — do klasy negatywnej -1. W algorytmie perceptronu funkcja decyzyjna $\phi(\cdot)$ stanowi odmianę **funkcji skoku jednostkowego**:

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

Możemy dla uproszczenia przenieść wartość progową θ na lewą stronę równania i zdefiniować początkową wagę jako $w_0 = -\theta$, a $x_0 = 1$, dzięki czemu całkowite pobudzenie z przybierze prostszą postać

$$z = w_0x_0 + w_1x_1 + \dots + w_mx_m = \mathbf{w}^T \mathbf{x}$$

Natomiast:

$$\phi(z) = \begin{cases} 1 & \text{jeśli } z \geq 0 \\ -1 & \text{jeśli } z < 0 \end{cases}$$

W literaturze specjalistycznej próg ujemny, czyli waga $w_0 = -\theta$, jest zazwyczaj nazywany **obciążeniem jednostkowym** (ang. *bias unit*).

Podstawy algebry liniowej: iloczyn skalarny i macierz transponowana

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

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

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

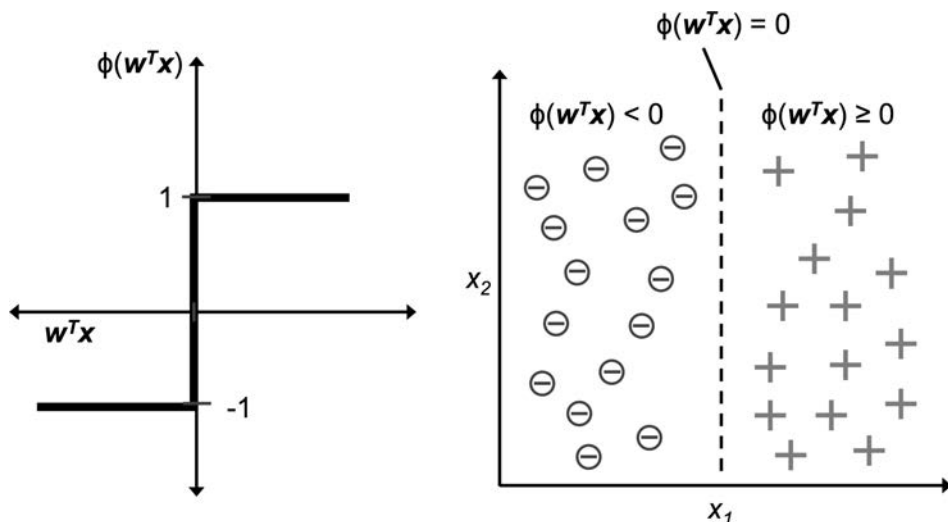
Ponadto operację transponowania można przeprowadzić również wobec macierzy, dzięki czemu następuje w niej zamiana wierszy z kolumnami:

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

Zwróć uwagę, że operacja transponowania jest zdefiniowana wyłącznie dla macierzy; jednak w kontekście uczenia maszynowego odnosimy się do macierzy $n \times 1$ i $1 \times m$, chociaż mówimy o „wektorze”.

W tej książce będziemy wykorzystywać jedynie najprostsze pojęcia z algebry liniowej, jeżeli jednak chcesz odświeżyć sobie pamięć, polecamy znakomity skrypt *Linear Algebra Review and Reference* autorstwa Zico Koltera, który można bezpłatnie przejrzeć na stronie http://www.cs.cmu.edu/~zkolter/course/linalg/linalg_notes.pdf¹.

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



Rysunek 2.2. Zastosowanie całkowitego pobudzenia w uczeniu maszynowym

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

Reguła uczenia perceptronu

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

- Wprowadź wagi o wartości 0 lub niewielkich, losowych wartościach.
- Dla każdego przykładu uczącego $\mathbf{x}^{(i)}$:
 - a) Oblicz wartość wyjściową \hat{y} .
 - b) Zaktualizuj wagi.

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

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

Wartość aktualizacji wagi w_j (lub jej zmiany), którą zapisujemy jako w_j , jest wyliczana za pomocą reguły uczenia perceptronu w następujący sposób:

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

gdzie η jest **współczynnikiem uczenia** (ang. *learning rate*; zazwyczaj jest to stała przyjmująca wartości w zakresie od 0,0 do 1,0), $y^{(i)}$ stanowi **rzeczywistą etykietę klas** i -tego przykładu uczącego, natomiast $\hat{y}^{(i)}$ to **przewidywana etykieta klas**. Bardzo istotną jest informacją, że wszystkie wagi w wektorze wag są jednocześnie aktualizowane, co oznacza, że nie przeliczamy ponownie wartości $\hat{y}^{(i)}$, dopóki nie zaktualizujemy wszystkich wag o odpowiednie wartości aktualizacji, Δw_j .

Zapis aktualizacji dla dwuwymiarowego zestawu danych możemy zdefiniować następująco:

$$\Delta w_0 = \eta(y^{(i)} - \text{wyjście}^{(i)})$$

$$\Delta w_1 = \eta(y^{(i)} - \text{wyjście}^{(i)})x_1^{(i)}$$

$$\Delta w_2 = \eta(y^{(i)} - \text{wyjście}^{(i)})x_2^{(i)}$$

Zanim zaimplementujemy model perceptronu w Pythonie, przeprowadźmy mały eksperyment myślowy ukazujący piękno prostoty tej reguły uczenia. W dwóch scenariuszach, w których perceptron we właściwy sposób przewiduje etykietę klas, wagi pozostają niezmienione, gdyż wartości aktualizacji są równe 0:

$$y^{(i)} = -1, \quad \hat{y}^{(i)} = -1, \quad \Delta w_j = \eta(-1 - (-1))x_j^{(i)} = 0 \quad (1)$$

$$y^{(i)} = 1, \quad \hat{y}^{(i)} = 1, \quad \Delta w_j = \eta(1 - 1)x_j^{(i)} = 0 \quad (2)$$

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

$$y^{(i)} = 1, \quad \hat{y}^{(i)} = -1, \quad \Delta w_j = \eta(1 - (-1))x_j^{(i)} = \eta(2)x_j^{(i)} \quad (3)$$

$$y^{(i)} = -1, \quad \hat{y}^{(i)} = 1, \quad \Delta w_j = \eta(-1 - 1)x_j^{(i)} = \eta(-2)x_j^{(i)} \quad (4)$$

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

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

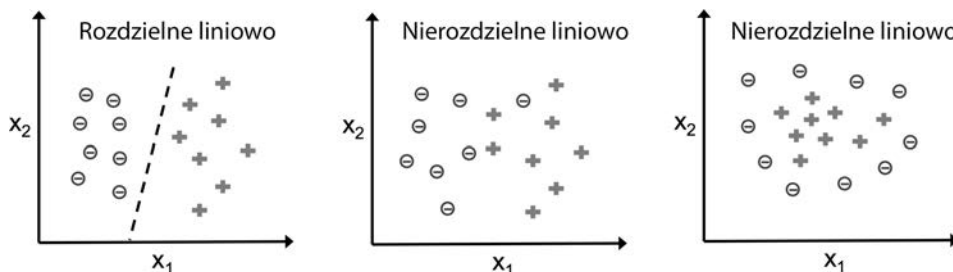
Załóżmy, że $x_j^{(i)} = 0,5$, a my ten przykład nieprawidłowo sklasyfikowaliśmy jako -1 . W takim przypadku zwiększamy wagę o 1, przez co całkowite pobudzenie $x_j^{(i)} \times w_j$ będzie silniejsze w sytuacji ponownego natrafienia na ten przykład, dzięki czemu z większym prawdopodobieństwem zostanie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

$$\Delta w_j = (1 - (-1))0,5 = (2)0,5 = 1$$

Aktualizacja wagi jest wprost proporcjonalna do wartości $x_j^{(i)}$. Załóżmy, że mamy kolejny przykład, $x_j^{(i)} = 2$, który został nieprawidłowo zaklasyfikowany jako -1 . W tej sytuacji przesuwamy granicę decyzyjną w jeszcze większym stopniu po to, aby przykład został następnym razem właściwie zaklasyfikowany:

$$\Delta w_j = (1^{(i)} - (-1)^{(i)})2^{(i)} = (2)2^{(i)} = 4$$

Zwróć uwagę, że zbieżność perceptronu zostaje zapewniona jedynie wtedy, gdy dwie klasy są liniowo rozdzielne (rysunek 2.3), a współczynnik uczenia jest wystarczająco mały (zainteresowane osoby znajdą matematyczne wyprowadzenie dowodu w moich materiałach wykładowych na stronie https://sebastianraschka.com/pdf/lecture-notes/stat479ss19/L03_perceptron_slides.pdf). Jeżeli nie można oddzielić dwóch klas za pomocą liniowej granicy decyzyjnej, możemy ustalić maksymalną liczbę przebiegów (**epok**) algorytmu z wykorzystaniem danych uczących i (lub) próg tolerancji nieprawidłowych klasyfikacji — w przeciwnym wypadku perceptron wiecznie aktualizowałby wagi.

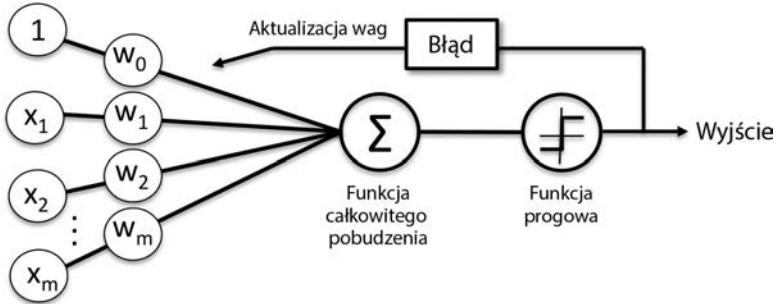


Rysunek 2.3. Rozdzielność liniowa klas

Kod źródłowy

Pliki kodu źródłowego są dostępne do pobrania pod adresem <ftp://ftp.helion.pl/przyklady/pythu3.zip> lub w oryginale pod adresem <https://github.com/rasbt/python-machine-learning-book-3rd-edition>.

Zanim przejdziemy do implementacji algorytmów w Pythonie, przyjrzyjmy się rysunkowi 2.4, który przedstawia diagram podsumowujący ogólną koncepcję, jaką kryje się za modelem perceptronu.



Rysunek 2.4. Ogólny model perceptronu

Na rysunku 2.4 widzimy schemat ukazujący sposób, w jaki perceptron otrzymuje dane wejściowe x i łączy je z wagami w w celu obliczenia funkcji całkowitego pobudzenia. Wynik jest następnie przekazywany funkcji progowej, która generuje wartość binarną -1 lub $+1$ — prognozowaną etykietę klas danego przykładu. W trakcie fazy uczenia dane wyjściowe są wykorzystywane do obliczenia błędu predykcji i aktualizowania wag.

Implementacja algorytmu uczenia perceptronu w Pythonie

W poprzednim podrozdziale poznaliśmy mechanizm działania perceptronu Rosenblatta; zaimplementujmy go teraz w Pythonie, a następnie przetestujmy na zestawie danych Iris, który wprowadziliśmy w rozdziale 1., „Umożliwienie komputerom uczenia się z danych”.

Obiektowy interfejs API perceptronu

Wykorzystamy strategię programowania obiektowego i zdefiniujemy interfejs perceptronu jako klasę języka Python, pozwalającą na inicjowanie nowych obiektów Perceptron, które będą uczyć się przy użyciu metody `fit`. Z kolei do prognozowania wykorzystamy osobną metodę — `predict`. Zgodnie z konwencją będziemy dodawać podkreślnik (`_`) do atrybutów, które nie są tworzone w momencie inicjalizowania obiektu, lecz w chwili wywoływania przez inne metody, np. `self.w_`.

Dodatkowe zasoby opisujące biblioteki naukowe Pythona

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

- **NumPy**: https://sebastianraschka.com/pdf/books/dlb/appendix_f_numpy-intro.pdf
- **pandas**: <https://pandas.pydata.org/pandas-docs/stable/10min.html>
- **Matplotlib**: <https://matplotlib.org/tutorials/introductory/usage.html>

Poniżej prezentujemy implementację perceptronu w Pythonie:

```
import numpy as np

class Perceptron(object):
    """Klasyfikator — perceptron.

    Parametry
    -----
    eta : zmiennoprzecinkowy
        Współczynnik uczenia (w przedziale pomiędzy 0.0 a 1.0).
    n_iter : liczba całkowita
        Liczba przebiegów po zestawach uczących.
    random_state : liczba całkowita
        Ziarno generatora liczb losowych służące do inicjowania losowych wag.

    Atrybuty
    -----
    w_ : jednowymiarowa tablica
        Wagi po dopasowaniu.
    errors_ : lista
        Liczba nieprawidłowych klasyfikacji (aktualizacji) w każdej epoce.

    """
    def __init__(self, eta=0.01, n_iter=50, 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_przykładów, n_cech]
            Wektory uczące, gdzie n_przykładów
            oznacza liczbę przykładów, a
            n_cech — liczbę cech.
        y : tablicopodobny, wymiary = [n_przykładów]
            Wartości docelowe.

        Zwraca
        -----
        self : obiekt
        """
```

```

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

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

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

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

```

Dzięki powyższej implementacji perceptronu możemy teraz inicjować nowe obiekty Perceptron mające wyznaczone współczynnik uczenia eta oraz liczbę epok (przebiegów po danych uczących) — `n_iter`.

Dzięki metodzie `fit` wprowadzamy wagi w obiekcie `self.w_` do wektora R^{m+1} , gdzie m oznacza liczbę wymiarów (cech) zestawu danych, do której dodajemy 1 dla pierwszego elementu w tym wektorze, czyli obciążenia jednostkowego. Przypominamy, że wspomniany element, `self.w_[0]`, reprezentuje omówione wcześniej tzw. obciążenie jednostkowe.

Zwróć również uwagę, że wektor ten zawiera małe losowe liczby wygenerowane za pomocą rozkładu normalnego o odchyleniu standardowym 0,01, przy użyciu funkcji `rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])`, gdzie `rgen` jest generatorem liczb losowych biblioteki NumPy, dla którego wyznaczyliśmy określone ziarno, dzięki czemu w razie potrzeby możemy odtwarzać uzyskiwane wyniki.

Należy pamiętać, że nie inicjalizujemy wag z wartością zerową dlatego, że współczynnik uczenia η (eta) ma wpływ na wynik klasyfikacji jedynie wtedy, gdy początkowe wartości wag są niezerowe. Jeżeli wszystkie zainicjowane wagi mają wartość 0, zmienia się tylko skala wektora, nie jego kierunek. Jeżeli znasz się na trygonometrii, weź pod uwagę wektor $v1 = [1\ 2\ 3]$, w którym kąt pomiędzy nim a wektorem $v2 = 0,5 \times v1$ miałby wartość równą 0, co zostało ukazane za pomocą poniższego fragmentu kodu:

```

>>> v1 = np.array([1, 2, 3])
>>> v2 = 0.5 * v1
>>> np.arccos(v1.dot(v2) / (np.linalg.norm(v1) *
...           np.linalg.norm(v2)))
0.0

```

W tym przypadku np.arccos jest arcus cosinusem, natomiast np.linalg.norm stanowi funkcję obliczającą długość wektora (nie ma znaczenia, czy do wygenerowania liczb losowych wykorzystasz rozkład normalny, czy równomierny albo inną wartość odchylenia standardowego; pamiętaj tylko, że zależy nam na małych wartościach, aby uniknąć omówionych wcześniej własności wektorów zerowych).

Indeksowanie tablic w NumPy

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

Po zainicjalizowaniu wag metoda `fit` analizuje każdy przykład zestawu danych uczących i aktualizuje wartości wag zgodnie z omówioną wcześniej regułą uczenia perceptronu.

Etykiety klas są prognozowane poprzez metodę `predict`, która zostaje wywołana w metodzie `fit` w czasie uczenia po to, aby przewidzieć etykietę klas dla aktualizacji wag, jednak może być także wykorzystywana do predykcji etykiet klas nowych danych po wytrenowaniu modelu. Do tego w liście `self.errors_` zliczamy liczbę nieprawidłowych klasyfikacji w czasie każdej epoki, dzięki czemu możemy później przeanalizować wydajność perceptronu w procesie nauki. Wykorzystywana w metodzie `net_input` funkcja np.dot oblicza iloczyn skalarny wektorów $w^T x$.

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

Trenowanie modelu perceptronu na zestawie danych Iris

W celu przetestowania naszej implementacji perceptronu ograniczymy analizy i przykłady z dalszej części rozdziału do dwóch zmiennych cech (wymiarów). Reguła uczenia perceptronu nie ogranicza się wyłącznie do dwóch wymiarów, ale w celach dydaktycznych dzięki uwzględnieniu tylko dwóch cech (długości działki i długości płatka) będziemy w stanie zwizualizować rejony decyzyjne wyuczonego modelu na wykresie punktowym.

Poza tym ze względów praktycznych wybraliśmy tylko dwa gatunki kosańca (Setosa i Versicolor) z zestawu danych Iris: pamiętaj, że perceptron jest klasyfikatorem binarnym. Można go jednak rozszerzyć do klasyfikacji wieloklasowej — np. poprzez technikę **OvA** (ang. *One versus All* — jeden przeciw wszystkim).

Metoda OvA służąca do klasyfikacji wieloklasowej

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

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

```
>>> import os
>>> import pandas as pd
>>> s = os.path.join('https://archive.ics.uci.edu', 'ml',
...                 'machine-learning-databases',
...                 'iris', 'iris.data')
>>> print('Adres URL:', s)
URL: https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data
>>> df = pd.read_csv(s,
...                 header=None,
...                 encoding='utf-8')
>>> df.tail()
```

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

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

Wczytywanie zestawu danych Iris

Kopię zestawu danych Iris (a także wszystkich pozostałych zestawów danych wykorzystywanych w tej książce) znajdziesz w przykładowym kodzie dołączonym do książki, dzięki czemu możesz z niego korzystać także, gdy będziesz odłączyć od internetu lub serwer UCI (<https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data>) będzie w danym momencie niedostępny. Przykładowo, aby wczytać zestaw danych Iris z katalogu lokalnego, wystarczy zastąpić wiersz:

```
df = pd.read_csv('https://archive.ics.uci.edu/ml/'
                'machine-learning-databases/iris/iris.data',
                header=None, encoding='utf-8')

wierszem

df = pd.read_csv('Twoja/ścieżka/lokalna/do/pliku/iris.data',
                header=None, encoding='utf-8')
```

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

W analogiczny sposób ze zbioru 100 przykładów uczących wydzielamy pierwszą kolumnę cech (*Długość działki*) i trzecią kolumnę (*Długość płatka*) — uzyskane wartości przydzielamy do macierzy cech *x*, którą jesteśmy w stanie wyświetlić jako dwuwymiarowy wykres punktowy:

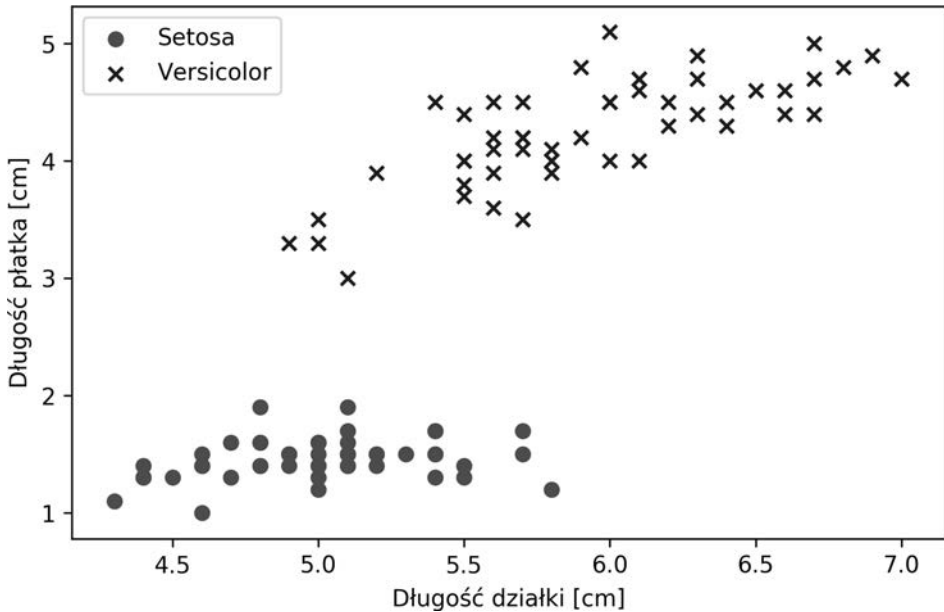
```
>>> import matplotlib.pyplot as plt
>>> import numpy as np

>>> # wybieramy odmiany setosa i versicolor
>>> y = df.iloc[0:100, 4].values
>>> y = np.where(y == 'Iris-setosa', -1, 1)

>>> # wybieramy długość działki i długość płatka
>>> X = df.iloc[0:100, [0, 2]].values

>>> # generowanie wykresu danych
>>> plt.scatter(X[:50, 0], X[:50, 1],
...            color='red', marker='o', label='Setosa')
>>> plt.scatter(X[50:100, 0], X[50:100, 1],
...            color='blue', marker='x', label='Versicolor')
>>> plt.xlabel('Długość działki [cm]')
>>> plt.ylabel('Długość płatka [cm]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

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



Rysunek 2.6. Graficzne przedstawienie zestawu danych uczących

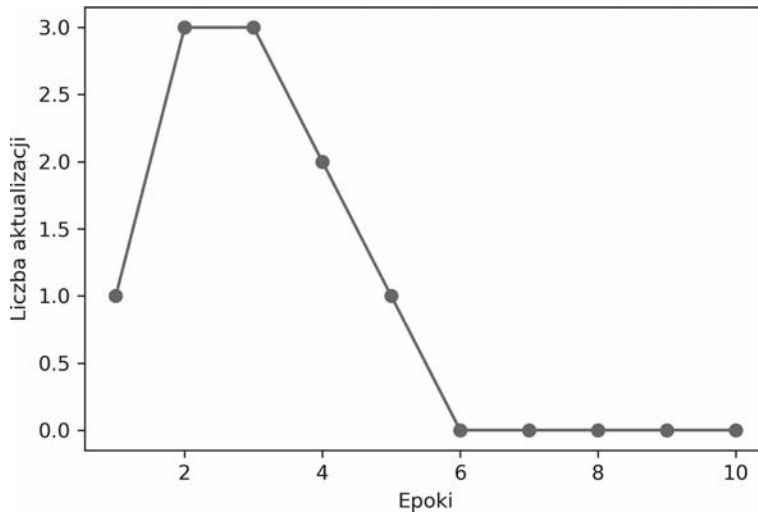
Powyższy wykres pokazuje rozkład przykładów kwiatów tworzących zestaw Iris wzdłuż dwóch osi cech (mierzonych w centymetrach): długości płatkę i długości działki. W tej dwuwymiarowej podprzestrzeni cech widzimy, że liniowa granica decyzyjna powinna wystarczyć do rozdzielenia odmiany *setosa* od *versicolor*.

Zatem taki klasyfikator liniowy jak perceptron powinien klasyfikować bez zarzutu próbki umieszczone w zestawie danych Iris.

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

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

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



Rysunek 2.7. Wykres zbieżności algorytmu

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

```
from matplotlib.colors import ListedColormap

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

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

    # rysuje wykres powierzchni decyzyjnej
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # rysuje wykres przykładów
    for idx, c1 in enumerate(np.unique(y)):
        plt.scatter(x=X[y == c1, 0], y=X[y == c1, 1],
                  alpha=0.8, c=colors[idx],
                  marker=markers[idx], label=c1,
                  edgecolor='black')
```

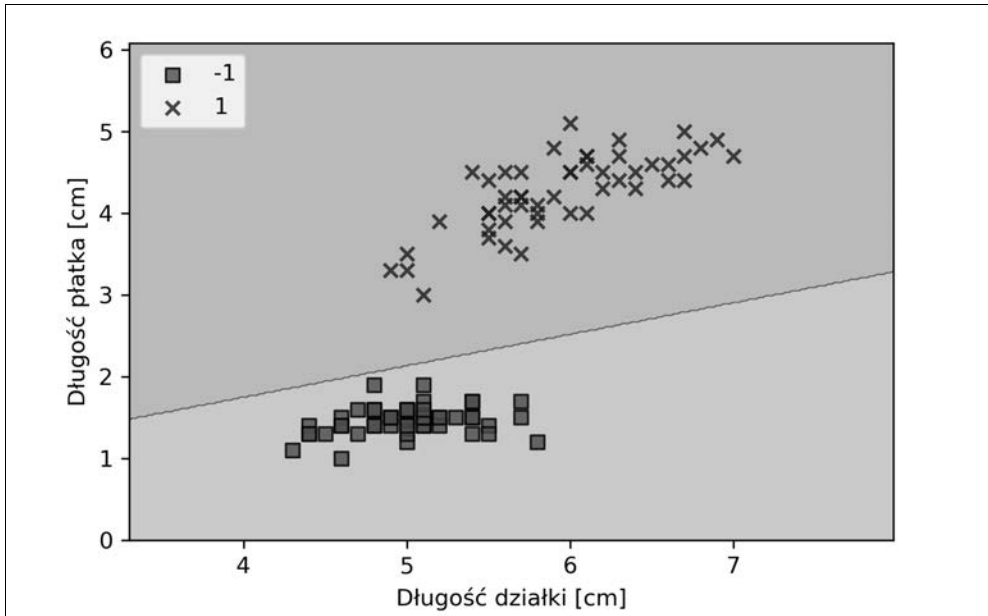
Najpierw definiujemy liczbę barw (colors) i znaczników (markers), a następnie tworzymy mapę kolorów z listy barw za pomocą klasy ListedColormap. Określamy teraz wartości minimalne i maksymalne dwóch cech i używamy tak uzyskanych wektorów cech do utworzenia pary tablic xx1 oraz xx2 za pomocą funkcji meshgrid. Uczyliśmy nasz klasyfikator na dwóch wymiarach cech, dlatego musimy zmodyfikować tablice xx1 i xx2 oraz stworzyć macierz zawierającą taką samą liczbę kolumn jak zbiór uczący, dzięki czemu będziemy w stanie zastosować metodę predict do przewidywania etykiet klas z odpowiednich elementów tablic.

Po przekształceniu przewidywanych etykiet klas do postaci tabelarycznej (o takich samych wymiarach jak tabele xx1 i xx2) będziemy mogli narysować wykres konturowy, stosując funkcję contourf, która dopasowuje kolory do różnych regionów decyzyjnych dla każdej prognozowanej klasy w tablicy:

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

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

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



Rysunek 2.8. Wykres regionów decyzyjnych

Zbieżność perceptronu

Chociaż perceptron perfekcyjnie sklasyfikował dwie odmiany kosaćca, konwergencja stanowi jeden z największych problemów omawianego modelu. Rosenblatt dowiódł matematycznie, że reguła uczenia perceptronu wykazuje zbieżność, jeśli dwie klasy mogą zostać rozdzielone liniową hiperpłaszczyzną. Jeśli nie można idealnie odseparować tych klas wspomnianą granicą decyzyjną, wagi będą cały czas aktualizowane, chyba że ustalimy maksymalną liczbę epok. Osoby zainteresowane zagadnieniem znajdą podsumowanie dowodu matematycznego w moich materiałach wykładowych: https://sebastianraschka.com/pdf/lecture-notes/stat479ss19/L03_perceptron_slides.pdf.

Adaptacyjne neurony liniowe i zbieżność uczenia

W tym podrozdziale przyjrzymy się kolejnej odmianie jednowarstwowej sieci neuronowej: **AD**aptacyjnemu **L**iniowemu **NE**uronowi (**ADALINE**). Model Adaline został zaprezentowany zaledwie kilka lat po algorytmie perceptronu Rosenblatta przez Bernarda Widrowa i jego doktoranta Tedda Hoffa; adaptacyjny neuron liniowy można uznać za twórcze rozwinięcie koncepcji perceptronu (B. Widrow i in., *An adaptive „Adaline” neuron using chemical „memistors”*, „Number Technical Report” 1553-2, Stanford Electron. Labs, Stanford, California, October 1960).

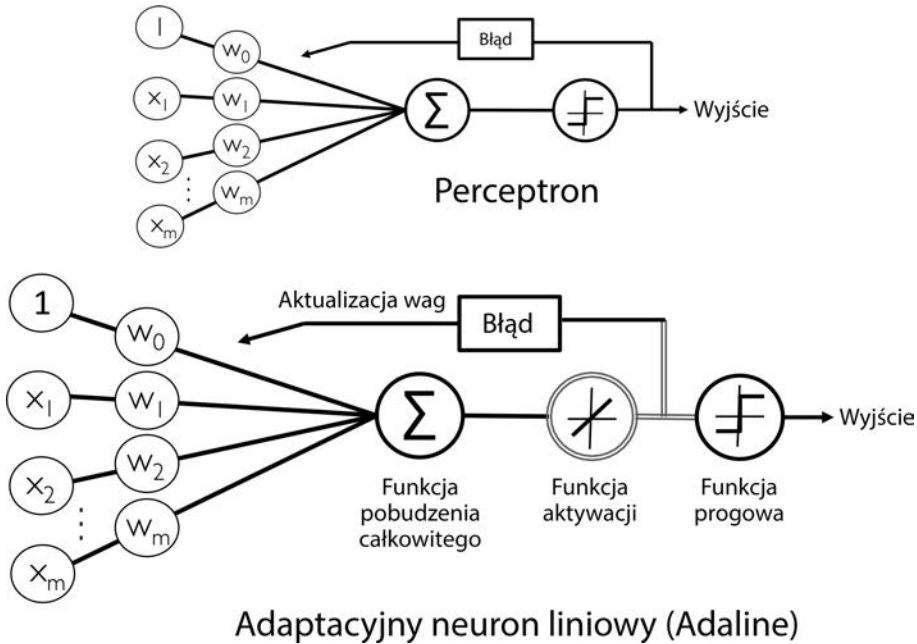
Algorytm Adaline jest szczególnie interesujący, gdyż zaprezentowane w nim zostały kluczowe koncepcje definiowania i minimalizowania ciągłych funkcji kosztu. Stanowi to podstawę zrozumienia bardziej zaawansowanych algorytmów klasyfikujących, takich jak regresja logistyczna, maszyny wektorów nośnych, a także modeli regresji omówionych w dalszych rozdziałach.

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

$$\phi(w^T x) = w^T x.$$

Liniowa funkcja aktywacji służy do obliczania wag, natomiast ciągle będziemy korzystać z funkcji progowej odpowiedzialnej za ostateczną prognozę, przypominającej omówioną wcześniej funkcję skoku jednostkowego.

Na rysunku 2.9 zostały zaprezentowane główne różnice pomiędzy perceptronem a algorytmem Adaline.



Rysunek 2.9. Porównanie modelu perceptronu i Adaline

Jak widać na rysunku 2.9, algorytm Adaline porównuje rzeczywiste etykiety klas z wartościami ciągłymi funkcji liniowej aktywacji do wyliczenia błędu modelu i aktualizowania wag. Z kolei perceptron porównuje rzeczywiste etykiety klas z prognozowanymi etykietami.

Minimalizacja funkcji kosztu za pomocą metody gradientu prostego

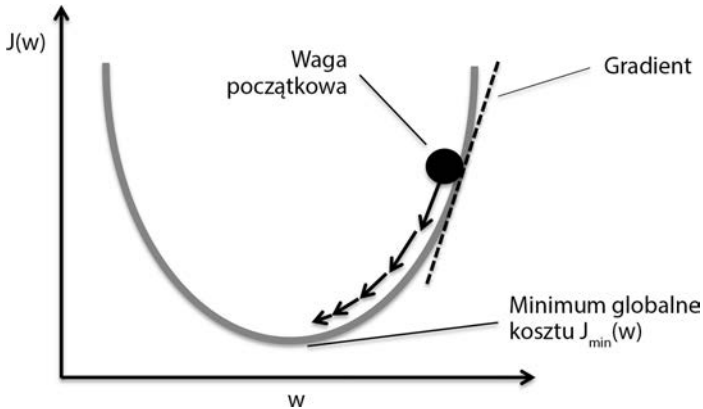
Jednym z najistotniejszych zadań w algorytmach nadzorowanego uczenia maszynowego jest zdefiniowana **funkcja celu**, która będzie optymalizowana w procesie nauki. Funkcja celu często przyjmuje postać **funkcji kosztu**, którą pragniemy zminimalizować. W przypadku modelu Adaline możemy wyznaczyć funkcję kosztu J , wyznaczającą wagi za pomocą **sumy kwadratów błędów** (ang. *sum of squared errors* — **SSE**) pomiędzy wyliczonym wynikiem a rzeczywistą etykietą klas:

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

Wartość $\frac{1}{2}$ została dodana jedynie dla naszej wygody, gdyż łatwiej nam będzie w ten sposób wyprowadzić gradient funkcji kosztu (straty) w odniesieniu do parametrów wag, o czym przekonamy się w dalszej części rozdziału. Główną zaletą tej liniowej funkcji aktywacji jest — w przeciwieństwie do funkcji skoku jednostkowego — umożliwienie różniczkowania funkcji kosztu. Inną przydatną własnością jest wypukłość funkcji kosztu; pozwala nam ją

wykorzystywać bardzo prosty, ale potężny algorytm optymalizacyjny, zwany **gradientem prostym** (ang. *gradient descent*); umożliwia on znajdowanie wag minimalizujących funkcję kosztu klasyfikującą przykłady zawarte w zbiorze danych Iris.

Na rysunku 2.10 widzimy, że działanie algorytmu gradientu prostego możemy przyrównać do **schodzenia z góry** aż do osiągnięcia lokalnego lub globalnego minimum kosztu. W każdej iteracji kierujemy się w przeciwną stronę gradientu, a rozmiar kolejnego kroku jest określany przez wartości współczynnika uczenia, jak również przez nachylenie gradientu.



Rysunek 2.10. Schemat poglądowy działania algorytmu gradientu prostego

Za pomocą gradientu prostego możemy zaktualizować wagi poprzez wykonanie kroku w kierunku przeciwnym do gradientu $\nabla J(\mathbf{w})$ naszej funkcji kosztu $J(\mathbf{w})$:

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

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

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

Aby wyliczyć gradient funkcji kosztu, musimy obliczyć pochodną cząstkową tej funkcji przy uwzględnieniu każdej wagi w_j :

$$\frac{\partial J}{\partial w_j} = -\sum_i (y^{(i)} - \phi(z^{(i)})) x_j^{(i)}$$

Zatem będziemy mogli zapisać aktualizację wagi w_j jako:

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

Aktualizujemy jednocześnie wszystkie wagi, dlatego reguła uczenia Adaline przyjmuje postać

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

Wyprowadzenie pochodnej błędu kwadratowego

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

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

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

Implementacja algorytmu Adaline w Pythonie

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

```

class AdalineGD(object):
    """Klasyfikator — ADaptacyjny Liniowy NEuron.

    Parametry
    -----
    eta : zmiennoprzecinkowy
        Współczynnik uczenia (w zakresie pomiędzy 0.0 i 1.0).
    n_iter : liczba całkowita
        Liczba przebiegów po zestawie uczącym.
    random_state : liczba całkowita
        Ziarno generatora liczb losowych służące do inicjowania
        losowych wag.

    Atrybuty
  
```

```

-----
w_ : jednowymiarowa tablica
     Wagi po dopasowaniu.
cost_ : lista
       Suma kwadratów błędów (wartość funkcji kosztu) w każdej epoce.

"""
def __init__(self, eta=0.01, n_iter=50, random_state=1):
    self.eta = eta
    self.n_iter = n_iter
    self.random_state=random_state

def fit(self, X, y):
    """Trenowanie za pomocą danych uczących.

    Parametry
    -----
    X : {tablicopodobny}, wymiary = [n_przykładów, n_cech]
        Wektory uczenia,
        gdzie n_przykładów oznacza liczbę przykładów, a
        n_cech — liczbę cech.
    y : tablicopodobny, wymiary = [n_przykładów]
        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()
        cost = (errors**2).sum() / 2.0
        self.cost_.append(cost)
    return self

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

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

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

```


W przeciwieństwie do modelu perceptronu nie aktualizujemy tutaj wag po ocenieniu każdego przykładu uczącego, lecz obliczamy gradient na podstawie całego zestawu danych uczących poprzez operację `self.eta * errors.sum()` dla obciążenia jednostkowego (wagi zerowej) i `self.eta * X.T.dot(errors)` dla wag od 1 do m , gdzie `X.T.dot(errors)` jest **iloczynem macierzo-wektorowym** macierzy cech z wektorem błędów.

Zwróć uwagę, że metoda `activation` nie ma żadnego wpływu na kod, ponieważ jest to zwykła funkcja tożsamościowa. Wprowadziliśmy tu funkcję aktywacji (obliczoną za pomocą metody `activation`) po to, aby ukazać ogólny mechanizm przepływu informacji poprzez pojedynczą warstwę sieci neuronowej: cechy z danych wejściowych, pobudzenia całkowitego, aktywacji i wyjścia.

W następnym rozdziale zajmiemy się klasyfikatorem regresji logistycznej, w którym stosowana jest nieliniowa, niezerowa funkcja aktywacji. Dowiemy się, że model regresji logistycznej jest ściśle powiązany z modelem Adaline, a jedyna różnica dotyczy jej funkcji aktywacji i kosztu.

Podobnie jak w implementacji perceptronu, gromadzimy wartości kosztu w liście `self.cost_`, aby sprawdzić zbieżność algorytmu po zakończeniu uczenia.

Iloczyn macierzowy

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

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

Zwróć uwagę, że w powyższym równaniu mnożymy macierz przez wektor, a taka operacja nie jest zdefiniowana matematycznie. Pamiętaj jednak, że stosujemy tu konwencję, zgodnie z którą powyższy wektor jest traktowany jako macierz 3×1 .

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

Hiperparametry perceptronu

Zarówno współczynnik uczenia η (eta), jak i liczba epok (`n_iter`) to tzw. **hiperparametry** (lub parametry strojenia) implementacji perceptronu oraz adaptacyjnego neuronu liniowego. W rozdziale 6., „Najlepsze metody oceny modelu i dostrajania hiperparametrów”, poznamy inne techniki umożliwiające automatyczne wyszukiwanie wartości różnych hiperparametrów zapewniających optymalną skuteczność modelu klasyfikacji.

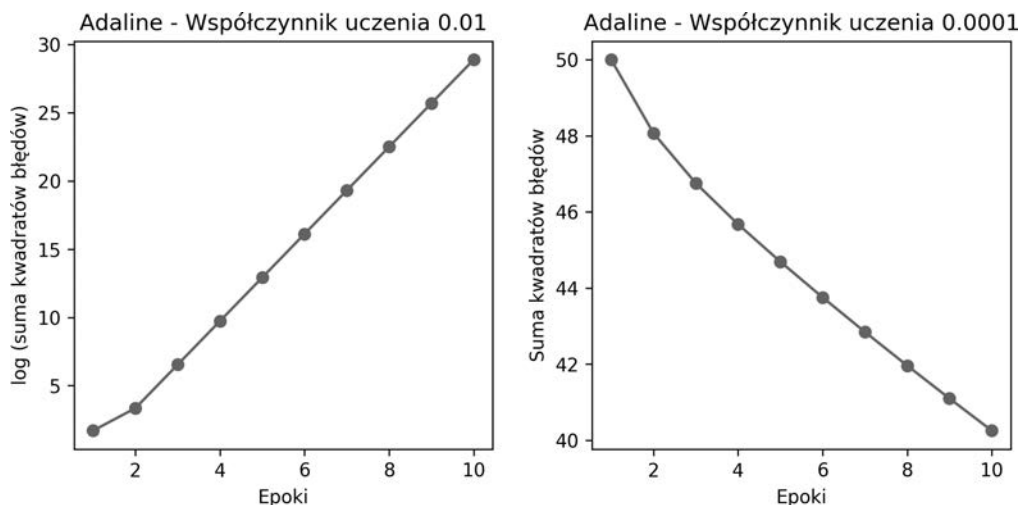
Stwórzmy teraz wykres kosztów dla liczby epok przy założeniu dwóch różnych wartości współczynnika uczenia:

```

>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))
>>> ada1 = AdalineGD(n_iter=10, eta=0.01).fit(X, y)
>>> ax[0].plot(range(1, len(ada1.cost_) + 1),
...           np.log10(ada1.cost_), marker='o')
>>> ax[0].set_xlabel('Epoki')
>>> ax[0].set_ylabel('Log (suma kwadratów błędów)')
>>> ax[0].set_title('Adaline - Współczynnik uczenia 0,01')
>>> ada2 = AdalineGD(n_iter=10, eta=0.0001).fit(X, y)
>>> ax[1].plot(range(1, len(ada2.cost_) + 1),
...           ada2.cost_, marker='o')
>>> ax[1].set_xlabel('Epoki')
>>> ax[1].set_ylabel('Suma kwadratów błędów')
>>> ax[1].set_title('Adaline - Współczynnik uczenia 0,0001')
>>> plt.show()

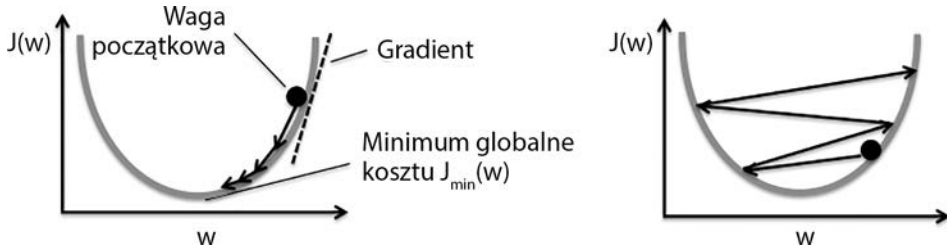
```

Jak widać na wykresach zaprezentowanych na rysunku 2.11, natrafiamy na dwa różne problemy. Wykres po lewej przedstawia sytuację, gdy dobieramy zbyt dużą wartość współczynnika uczenia. Zamiast minimalizacji funkcji kosztu następuje powiększanie błędu wraz z każdą epoką, ponieważ *przeskakujemy nad* minimum globalnym. Z drugiej strony widzimy, że na prawym wykresie koszt maleje, ale dobrana wartość $\eta = 0,0001$ jest tak mała, że algorytm musiałby wykonać mnóstwo przebiegów, żeby uzyskać zbieżność z globalnym minimum kosztu.



Rysunek 2.11. Skutek doboru niewłaściwej wartości współczynnika uczenia

Na rysunku 2.12 pokazujemy, co by się stało, gdybyśmy zmienili wartość określonego parametru wagi w celu minimalizacji funkcji kosztu J . Na lewym wykresie widoczny jest przypadek prawidłowo dobranego współczynnika uczenia — koszt maleje stopniowo, dzięki czemu dążymy do minimum globalnego. Z kolei prawy wykres pokazuje, co się dzieje, gdy dobieramy zbyt dużą wartość współczynnika uczenia i roz mijamy się z minimum globalnym.



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

Usprawnianie gradientu prostego poprzez skalowanie cech

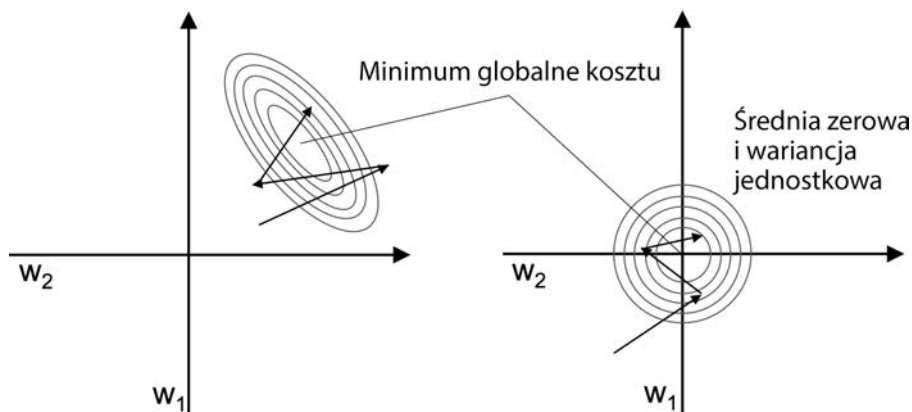
Wiele omawianych w tej książce algorytmów uczenia maszynowego wymaga jakiejś formy skalowania cech w celu uzyskania optymalnej skuteczności, co zostanie dokładniej omówione w rozdziale 3., „Stosowanie klasyfikatorów uczenia maszynowego za pomocą biblioteki scikit-learn”, i rozdziale 4., „Tworzenie dobrych zestawów danych uczących — wstępne przetwarzanie danych”.

Metoda gradientu prostego stanowi jeden z wielu algorytmów, w których przydatne okazuje się skalowanie cech. W tym punkcie zastosujemy metodę skalowania zwaną **standaryzacją**, gdyż potraktowane nią dane uzyskują własności standardowego rozkładu normalnego: średnią o wartości zero i wariancję jednostkową. Taka procedura normalizacji przyśpiesza proces uzyskiwania zbieżności; nie sprawia to jednak, że pierwotny zestaw danych uzyskuje rozkład normalny. Standaryzacja przesuwaa średnią każdej cechy w taki sposób, że zostaje wyśrodkowana do wartości 0, a każda cecha zawiera odchylenie standardowe równe 1 (wariancja jednostkowa). Przykładowo w celu standaryzacji j -tej cechy wystarczy odjąć średnią przykładowo μ_j od każdego przykładowo uczącego i podzielić ją przez jego odchylenie standardowe σ_j :

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

Tutaj x_j jest wektorem składającym się z wartości j -tej cechy wszystkich przykładowo uczących n , a taka technika standaryzacji może zostać zastosowana wobec każdej cechy j w naszym zestawie danych.

Standaryzacja pomaga w nauce gradientu prostego m.in. dlatego, że funkcja optymalizująca musi wykonać mniej przebiegów w celu znalezienia optymalnego rozwiązania (globalnego minimum kosztu), co zostało zaprezentowane na rysunku 2.13; obydwa wykresy ukazują powierzchnię kosztu jako funkcję dwóch przykładowych wag w zagadnieniu klasyfikacji dwuwymiarowej.



Rysunek 2.13. Porównanie modelu gradientu prostego bez standaryzacji (lewy wykres) ze standaryzacją (prawy wykres)

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

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

Po wprowadzeniu standaryzacji ponownie wyuczymy model Adaline i sprawdzimy, czy algorytm będzie zbieżny po wykonaniu niewielkiej liczby przebiegów przy współczynniku uczenia o wartości $\eta = 0,01$:

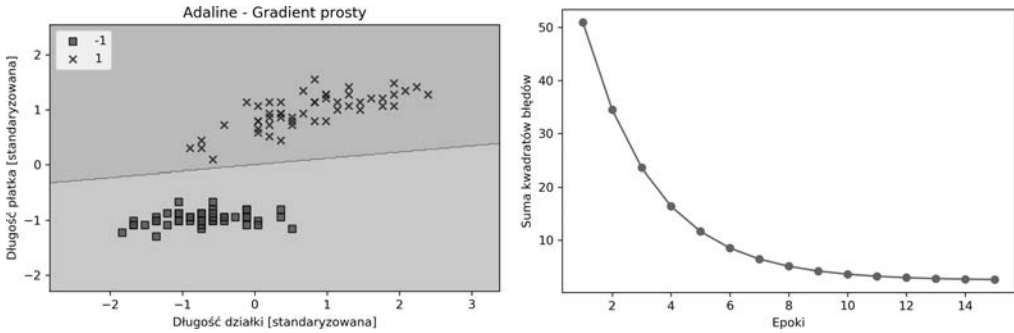
```
>>> ada_gd = AdalineGD(n_iter=15, eta=0.01)
>>> ada_gd.fit(X_std, y)

>>> plot_decision_regions(X_std, y, classifier=ada_gd)
>>> plt.title('Adaline - Gradient prosty')
>>> plt.xlabel('Długość działki [standaryzowana]')
>>> plt.ylabel('Długość płątki [standaryzowana]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()

>>> plt.plot(range(1, len(ada_gd.cost_) + 1), ada_gd.cost_, marker='o')
>>> plt.xlabel('Epoki')
>>> plt.ylabel('Suma kwadratów błędów')
>>> plt.tight_layout()
>>> plt.show()
```

Po uruchomieniu powyższego kodu powinniśmy ujrzeć wykresy regionów decyzyjnych oraz malejącego kosztu, pokazane na rysunku 2.14.

Jak widać, algorytm Adaline stał się zbieżny po uczeniu się na standaryzowanych cechach przy stosowaniu współczynnika uczenia $\eta = 0,01$. Zauważ jednak, że suma kwadratów błędów pozostaje niezerowa pomimo właściwego sklasyfikowania wszystkich przykładów kwiatów.



Rysunek 2.14. Zbieżność algorytmu Adaline po standaryzacji cech

Wielkoskalowe uczenie maszynowe i metoda stochastycznego spadku wzdłuż gradientu

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

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

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

Aktualizujemy wagi przyrostowo dla każdego przykładu uczącego:

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

Chociaż algorytm SGD można uznawać za aproksymację gradientu prostego, zazwyczaj umożliwia on znacznie szybsze uzyskanie zbieżności, gdyż wagi są częściej aktualizowane. Każdy gradient jest wyliczany na podstawie pojedynczej próbki uczącej, dlatego powierzchnia błędów generuje większe szumy niż w gradiencie prostym, co również ma znaczenie, gdyż dzięki temu algorytm SGD łatwiej może ignorować płytkie minima lokalne w przypadku, gdy pracujemy z nieliniowymi funkcjami kosztu, o czym przekonamy się w rozdziale 12., „Implementowanie wielowarstwowej, sieci neuronowej od podstaw”. Aby uzyskać satysfakcjonujące wyniki za pomocą algorytmu SGD, bardzo ważne jest zaprezentowanie algorytmowi

danych uczących w przypadkowej kolejności, ponadto chcemy przed każdą epoką przetasować zestaw danych uczących, aby uniknąć cykliczności.

Korygowanie współczynnika uczenia w trakcie trenowania modelu

W implementacjach stochastycznego spadku wzdłuż gradientu niezmienny współczynnik uczenia η często jest zastępowany adaptacyjnym współczynnikiem uczenia, którego wartość maleje wraz z upływem czasu; może on przybrać np. następującą postać:

$$\frac{c_1}{\lfloor \text{liczba iteracji} \rfloor + c_2}$$

gdzie c_1 i c_2 są stałymi. Zwracamy uwagę, że algorytm ten nie osiąga minimum globalnego, lecz dociera do jego bardzo zbliżonych wartości. Dzięki adaptacyjnemu współczynnikowi uczenia możemy jeszcze bardziej zbliżyć się do minimum kosztu.

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

Metoda gradientu prostego z użyciem minigrup

Kompromisem pomiędzy metodą gradientu prostego a algorytmem SGD jest tzw. **uczenie za pomocą minigrup** (ang. *mini-batch learning*). Rozwiązanie to można rozpatrywać jako stosowanie metody wsadowej gradientu do mniejszych podzbiorów danych uczących — np. 32 przykładów uczących. Zaletą uczenia za pomocą minigrup jest znacznie szybsza konwergencja w porównaniu z gradientem prostym z powodu częstszych aktualizacji wag. Do tego metoda ta pozwala zastępować pętlę for używaną na przykładach uczących w **stochastycznym spadku wzdłuż gradientu** operacjami wektorowymi umożliwiającymi stosowanie technik algebry liniowej (np. implementację sumy ważonej za pomocą iloczynu skalarnego), co jeszcze bardziej poprawia skuteczność obliczeniową danego algorytmu uczenia.

Wcześniej zaimplementowaliśmy regułę uczenia Adaline wykorzystującą metodę gradientu prostego, dlatego wystarczy wprowadzić do niej kilka modyfikacji, żeby algorytm zaczął aktualizować wagi metodą SGD. Teraz wewnątrz metody `fit` będziemy aktualizować wagi po sprawdzeniu każdego przykładu uczącego. Następnie zaimplementujemy dodatkową metodę `partial_fit`, która nie inicjuje od nowa wag w przypadku uczenia w locie. Aby sprawdzić zbieżność algorytmu po treningu, będziemy obliczać koszt jako średni koszt przykładów uczących w każdej epoce. Ponadto dodamy możliwość tasowania (`shuffle`) danych uczących przed rozpoczęciem każdej epoki, dzięki czemu unikniemy cykliczności podczas optymalizowania funkcji kosztu; parametr `random_state` służy do generowania wartości losowej dla zachowania większej odtwarzalności:

```
from numpy.random import seed

class AdalineSGD(object):
    """Klasyfikator — ADaptacyjny Liniowy NEuron.
```

Parametry

eta : zmiennoprzecinkowy

Współczynnik uczenia (w zakresie pomiędzy 0.0 i 1.0).

n_iter : liczba całkowita

Liczba przebiegów po zestawie uczącym.

shuffle : wartość boolowska (domyślnie: True)

Jeżeli jest ustalona wartość True,

tasuje dane uczące przed każdą epoką w celu zapobiegnięcia cykliczności.

random_state : liczba całkowita

Ziarno generatora liczb losowych służące do inicjowania

losowych wag.

Atrybuty

w_ : jednowymiarowa tablica

Wagi po dopasowaniu.

cost_ : lista

Suma kwadratów błędów (wartość funkcji kosztu) ze wszystkich przykładów uczących w każdej epoce.

"""

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

```
def fit(self, X, y):
```

""" Dopasowanie danych uczących.

Parametry

X : {tablicopodobny}, wymiary = [n_przykładów, n_cech]

Wektory uczące, gdzie n_przykładów

oznacza liczbę przykładów, a

n_cech określa liczbę cech.

y : tablicopodobny, wymiary = [n_przykładów]

Wartości docelowe.

Zwraca

self : obiekt

"""

```
self._initialize_weights(X.shape[1])
self.cost_ = []
for i in range(self.n_iter):
    if self.shuffle:
        X, y = self._shuffle(X, y)
    cost = []
    for xi, target in zip(X, y):
        cost.append(self._update_weights(xi, target))
```

```

        avg_cost = sum(cost) / len(y)
        self.cost_.append(avg_cost)
    return self

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

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

def _initialize_weights(self, m):
    """Inicjuje wagi, przydzielając im małe, losowe wartości"""
    self.rngen = np.random.RandomState(self.random_state)
    self.w_ = self.rngen.normal(loc=0.0, scale=0.01,
                                size=1 + m)
    self.w_initialized = True

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

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

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

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

```

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

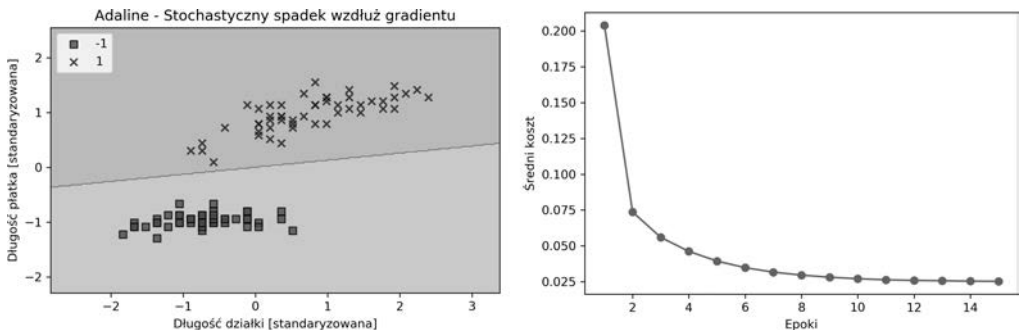
Teraz możemy wykorzystać metodę `fit` do trenowania klasyfikatora `AdalineSGD` i wyświetlić wyniki nauki za pomocą funkcji `plot_decision_regions`:

```
>>> ada_sgd = AdalineSGD(n_iter=15, eta=0.01, random_state=1)
>>> ada_sgd.fit(X_std, y)

>>> plot_decision_regions(X_std, y, classifier=ada_sgd)
>>> plt.title('Adaline - Stochastyczny spadek wzdłuż gradientu')
>>> plt.xlabel('Długość działki [standaryzowana]')
>>> plt.ylabel('Długość płatka [standaryzowana]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()

>>> plt.plot(range(1, len(ada_sgd.cost_) + 1), ada_sgd.cost_, marker='o')
>>> plt.xlabel('Epoki')
>>> plt.ylabel('Średni koszt')
>>> plt.tight_layout()
>>> plt.show()
```

Wykresy uzyskane po uruchomieniu powyższego kodu zostały zaprezentowane na rysunku 2.15.



Rysunek 2.15. Uczenie przy zastosowaniu metody stochastycznego spadku wzdłuż gradientu

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

Podsumowanie

W tym rozdziale przyjrzelśmy się uważnie podstawowym koncepcjom klasyfikatorów liniowych stosowanych w uczeniu nadzorowanym. Po zaimplementowaniu perceptronu dowiedzieliśmy się, jak można wydajnie uczyć adaptacyjne neurony liniowe poprzez wektoryzację gradientu prostego, a także jak można wykorzystać metodę SGD do uczenia w locie.

Gdy już potrafimy implementować proste klasyfikatory w Pythonie, jesteśmy gotowi na następny rozdział, w którym wykorzystamy bibliotekę uczenia maszynowego scikit-learn do tworzenia bardziej zaawansowanych i potężniejszych klasyfikatorów, powszechnie używanych zarówno na uczelniach, jak i w przemyśle.

Użyta przez nas metoda obiektowa implementowania algorytmów perceptronu i Adaline pomoże w zrozumieniu interfejsu API scikit-learn; wykorzystuje on te same koncepcje, które omówiliśmy w tym rozdziale: metody `fit` i `predict`. Na ich podstawie nauczymy się wykorzystywać regresję logistyczną do modelowania prawdopodobieństwa przynależności do klas, a także używać maszyny wektorów nośnych podczas pracy z nieliniowymi granicami decyzyjnymi. Do tego zaprezentujemy odmienną klasę algorytmów uczenia nadzorowanego (algorytmy drzew), które są często łączone w potężne zespoły klasyfikatorów.

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 głębokie z Pythonem: zrozum i zastosuj!

Uczenie maszynowe jest jedną z najbardziej fascynujących technologii naszych czasów — rozwojem jego najróżniejszych zastosowań zajmują się tacy giganci jak Google, Facebook, Apple, Amazon czy IBM. Uczenie maszynowe otwiera zupełnie nowe możliwości i powoli staje się nieodzowne: wystarczy wymienić asystenty głosowe w smartfonach, chatboty pomagające klientom w wyborze produktu, a także sieci ułatwiające podejmowanie decyzji o inwestycjach giełdowych, filtrujące niechciane wiadomości e-mail czy wspomagające diagnostykę medyczną.

Oto obszerny przewodnik po uczeniu maszynowym i uczeniu głębokim w Pythonie. Zawiera dokładne omówienie najważniejszych technik uczenia maszynowego oraz staranne wyjaśnienie zasad rządzących tą technologią. Poszczególne zagadnienia zilustrowano mnóstwem objaśnień, wizualizacji i przykładów, co znakomicie ułatwia zrozumienie materiału i sprawne rozpoczęcie samodzielnego budowania aplikacji i modeli, takich jak te służące do klasyfikacji obrazów, odkrywania ukrytych wzorców czy wydobywania dodatkowych informacji z danych. Wydanie trzecie zostało zaktualizowane — znalazł się w nim opis biblioteki TensorFlow 2 i najnowszych dodatków do biblioteki scikit-learn. Dodano również wprowadzenie do dwóch nowatorskich technik: uczenia przez wzmocnienie i budowy generatywnych sieci przeciwnastawnych (GAN).

W książce między innymi:

- platformy, modele i techniki uczenia maszynowego
- wykorzystanie biblioteki scikit-learn i TensorFlow
- sieci neuronowe, sieci GAN i inne
- przygotowywanie danych dla modeli uczenia maszynowego
- ocena i strojenie modeli
- analizy: regresyjna, skupień i sentymentów

Dr Sebastian Raschka zajmuje się rozwojem nowych technik uczenia głębokiego, pozwalających na rozwiązywanie problemów w dziedzinie biometriki. Od wielu lat programuje w Pythonie.

Dr Vahid Mirjalili specjalizuje się w wielkoskalowych symulacjach obliczeniowych struktur molekularnych oraz zastosowaniach uczenia maszynowego w rozpoznawaniu obrazów i biometrice.

	<i>Sprawdź nasze szkolenia!</i>	KOD KORZYŚCI Sięgnij po więcej! ▶	
 helion.pl	 AKADEMIA IT & BUSINESS WWW.SZKOLENIA.HELION.PL	ISBN 978-83-283-7001-2	
 0 801 339900		9 788328 370012	
 0 601 339900			
INFORMATYKA W NAJLEPSZYM WYDANIU		Cena: 119,00 zł	

Packt