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

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.

Spis treści

ROZDZIAŁ 3

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

```
>>> from sklearn import datasets
>>> import numpy as np
>>> iris = datasets.load_iris()
>>> X = iris.data[:, [2, 3]]
>>> y = iris.target
>>> 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 wyuczyliśmy model za pomocą metody `fit`. W tym przypadku parametr `eta0` jest odpowiednikiem

współczynnika uczenia `eta`, który stosowaliśmy w poprzedniej implementacji perceptronu, a poprzez parametr `n_iter` definiujemy liczbę epok (przebiegów po zestawie danych uczących).

Jak pamiętamy z rozdziału 2., „Trenowanie 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 ≈ 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. *overfitting*) to sytuacja, w której model prawidłowo wykrywa wzorce w danych uczących, jednak nie potrafi ich uogólnić na nieznane dane.

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

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

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

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

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

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

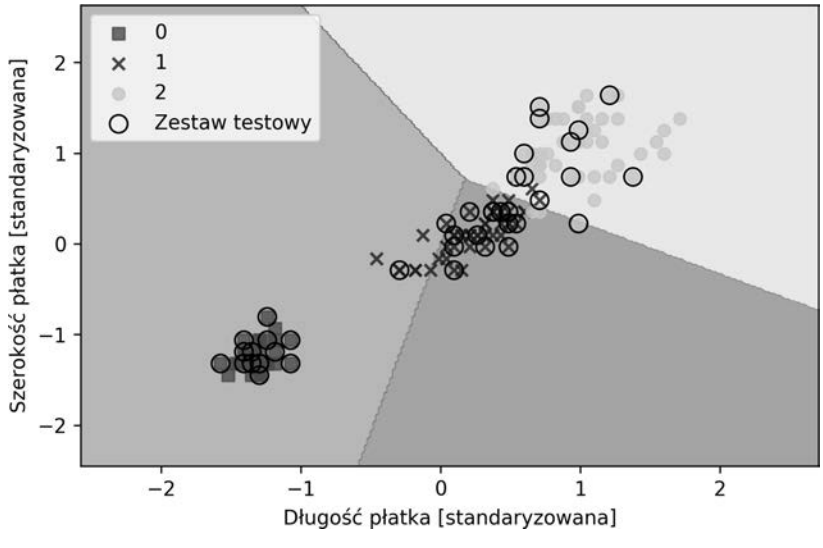
```
X_test, y_test = X[list(test_idx), :], y[list(test_idx)]

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:

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

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

Interesuje nas prognozowanie prawdopodobieństwa przynależności próbki do określonej klasy, co jest odwrotnością funkcji logitowej. Mamy tu do czynienia z **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, $z = \boldsymbol{w}^T \boldsymbol{x} = w_0 x_0 + w_1 x_1 + \cdots + w_m x_m$.

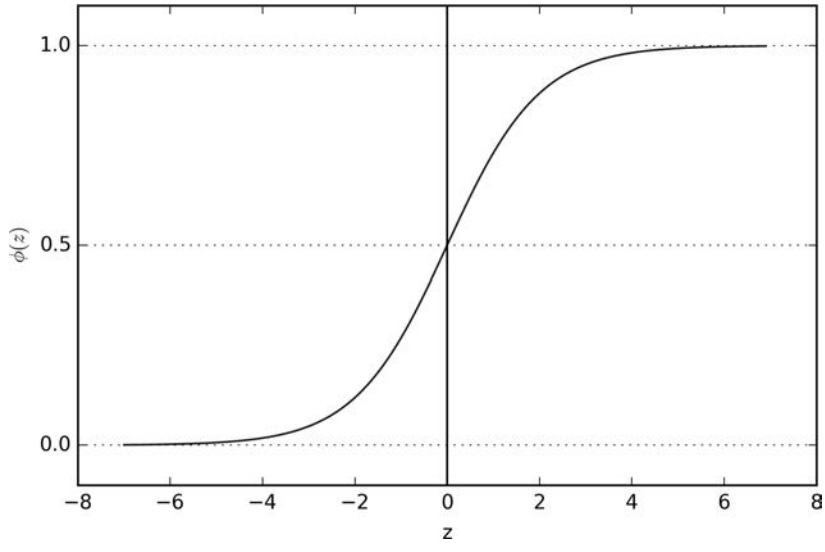
> Zwróć uwagę, że podobnie jak w rozdziale 2., „Trenowanie prostych algorytmów uczenia maszynowego w celach klasyfikacji", stosujemy konwencję, zgodnie z którą w_0 oznacza obciążenie jednostkowe, stanowiące dodatkową wartość (równą 1) dla zmiennej x_0.

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.



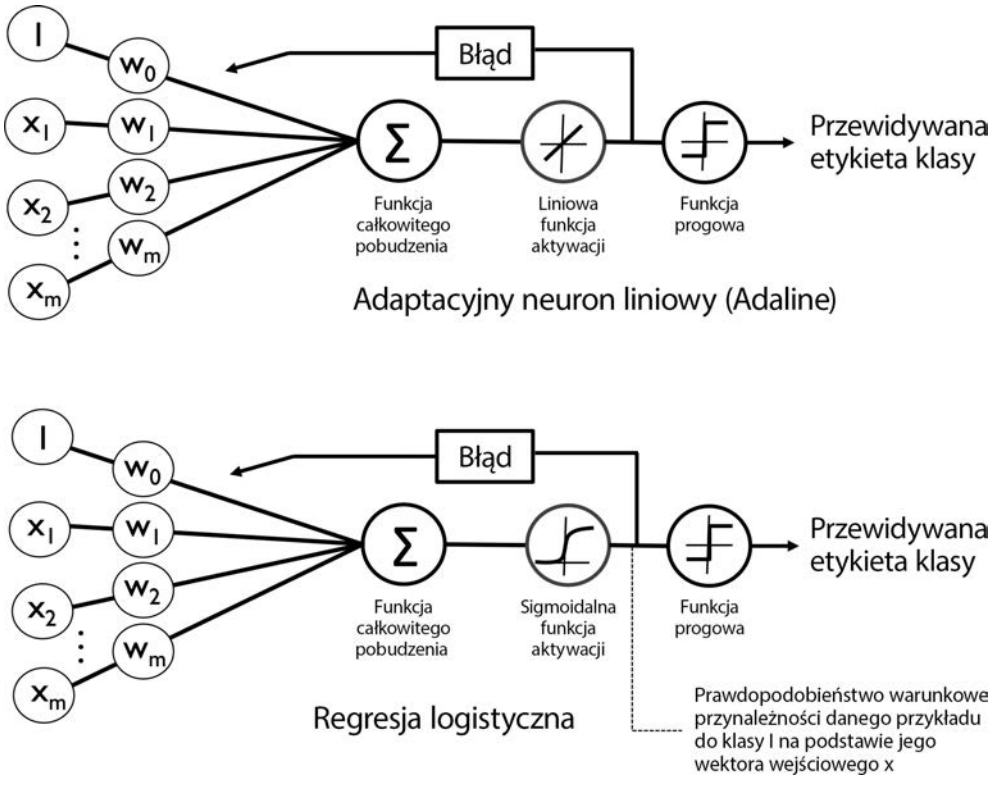
Rysunek 3.2. Wykres funkcji logistycznej

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 $P(y = 0 \mid \boldsymbol{x}; \boldsymbol{w}) = 1 - P(y = 1 \mid \boldsymbol{x}; \boldsymbol{w}) = 0{,}2$ 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 rzeczywiście 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:

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

Dokonaliśmy 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:

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

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

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

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”:

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

Aby lepiej pojąć tę funkcję kosztu, obliczmy koszt pojedynczego przykładu uczącego:

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

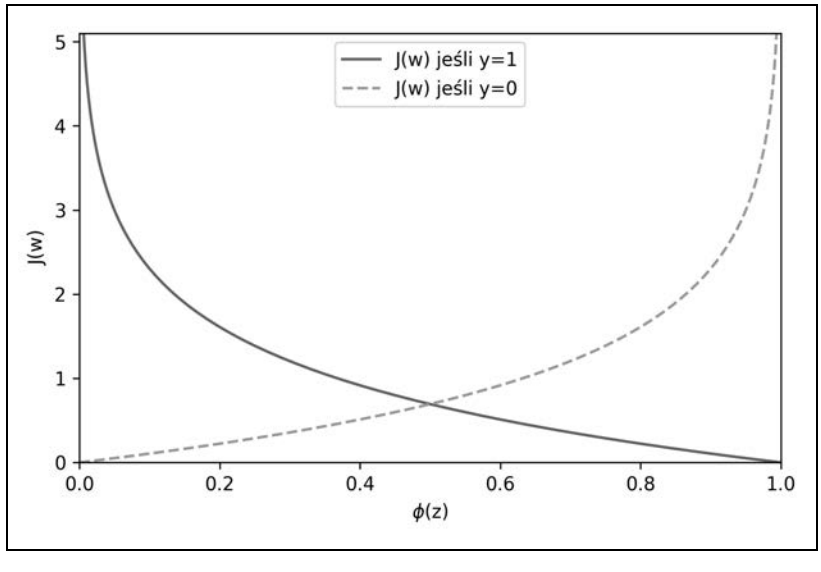
Widzimy, że pierwszy człon równania będzie wynosił 0, jeśli $y = 0$, a drugi człon osiągnie zerową wartość, gdy $y = 1$:

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

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:

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

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 etykietę 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)
```

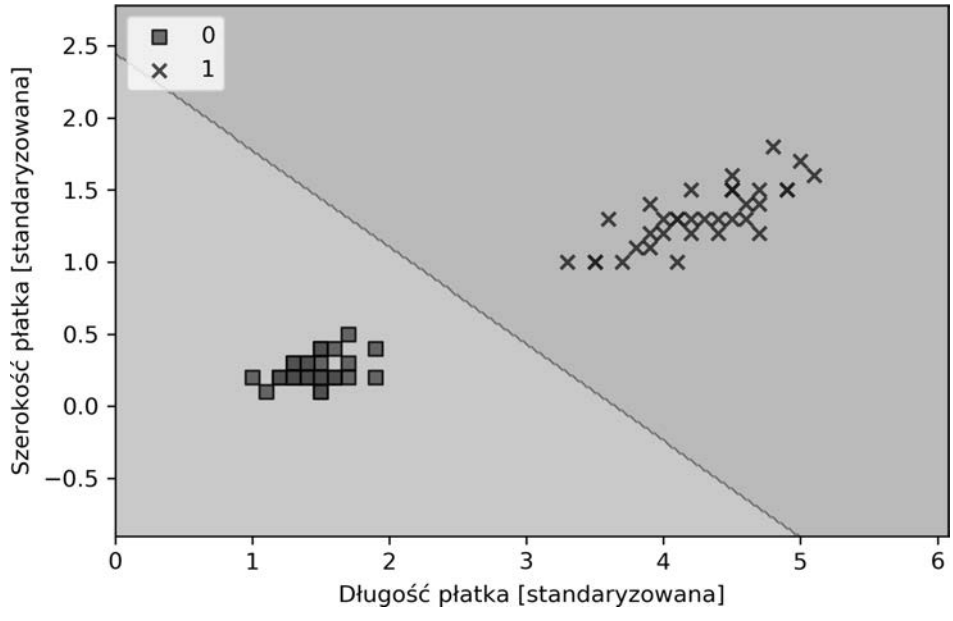
W trakcie dopasowywania modelu regresji logistycznej musimy pamiętać, że działa on jedynie w zadaniach klasyfikacji binarnej. Weźmy więc pod uwagę wyłącznie odmiany *Iris setosa* i *Iris versicolor* (klasy 0 i 1) i sprawdźmy skuteczność naszej implementacji:

```
>>> X_train_01_subset = X_train[(y_train == 0) | (y_train == 1)]
>>> y_train_01_subset = y_train[(y_train == 0) | (y_train == 1)]
>>> lrgd = 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:

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

Zanim przejdziemy dalej, obliczmy również pochodną cząstkową funkcji sigmoidalnej:

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

$$= \phi(z)(1-\phi(z))$$

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

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

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

$$= \left(y(1-\phi(z)) - (1-y)\phi(z) \right) x_j =$$

$$= \left(y - \phi(z) \right) x_j$$

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

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

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

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

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

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

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

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

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

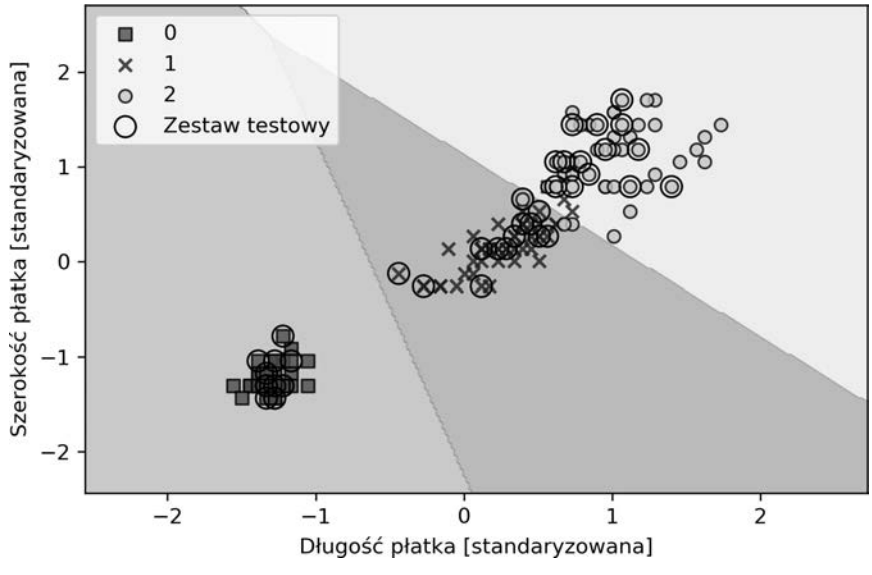
Jest to równoważne regule gradientu prostego omówionej w rozdziale 2., „Trenowanie prostych algorytmów uczenia maszynowego w celach klasyfikacji".

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])
```

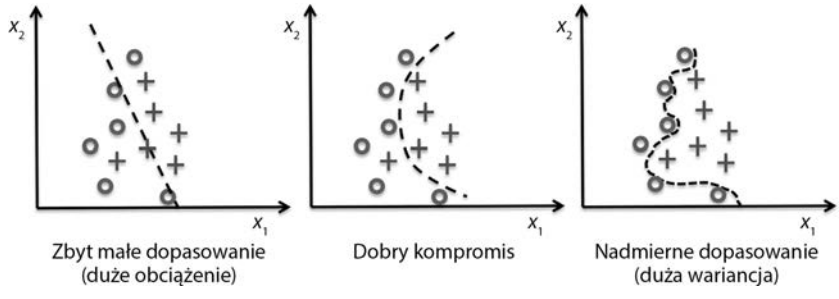
Na koniec małe ostrzeżenie dla osób chcących przewidywać etykietę klasy pojedynczego przykładu uczącego: biblioteka scikit-learn oczekuje na wejściu danych w formacie tablicy dwuwymiarowej; zatem musimy przekształcić pojedynczy wiersz do odpowiedniej postaci. Jednym ze sposobów konwersji tablicy jednowymiarowej 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:

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

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:

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

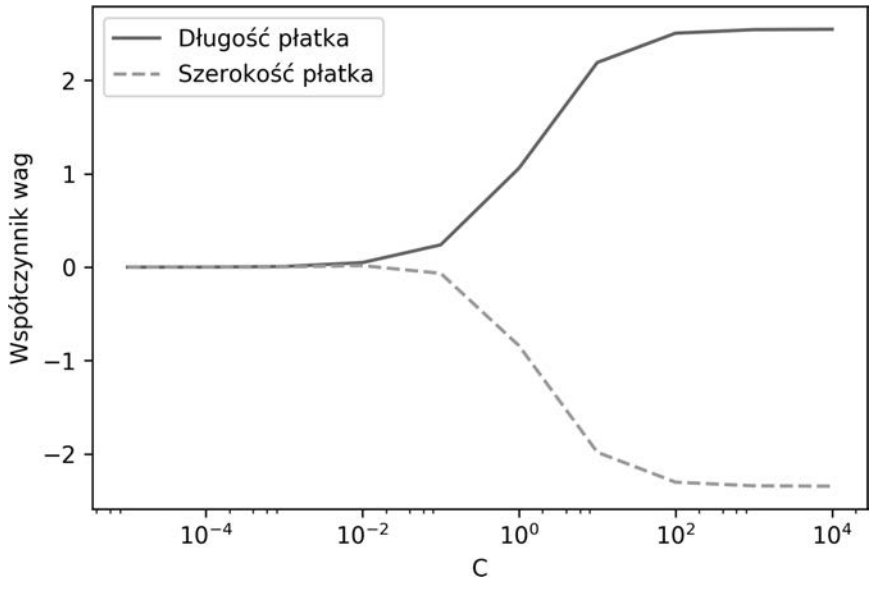
Możemy teraz za pomocą parametru regularyzacji λ kontrolować trenowanie na podstawie danych uczących przy jednoczesnym utrzymywaniu niskich wartości wag. Zwiększając wartość λ, wzmacniamy siłę regularyzacji.

Zaimplementowany w klasie `LogisticRegression` biblioteki scikit-learn parametr C stanowi element konwencji wywodzącej się z maszyn wektorów nośnych, którymi zajmiemy 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.



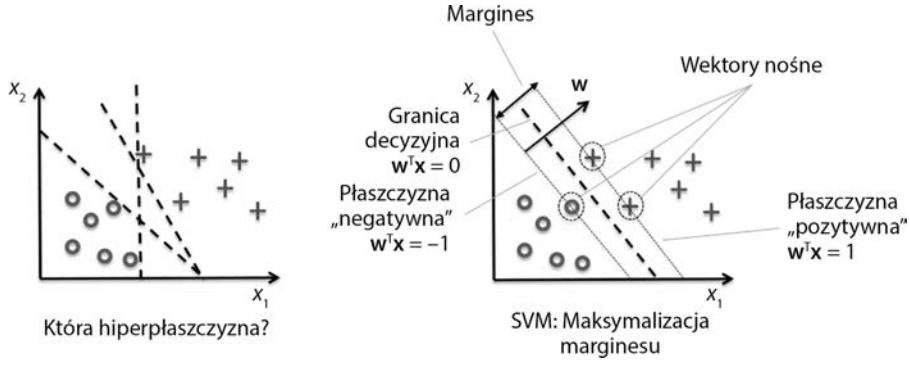
Rysunek 3.8. Wykres regularyzacji

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 Publications[1].

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 równolegle do granicy decyzyjnej), których wzory wyglądają następująco:

$$w_0 + \boldsymbol{w}^T \boldsymbol{x}_{poz} = 1 \tag{1}$$

$$w_0 + \boldsymbol{w}^T \boldsymbol{x}_{neg} = -1 \tag{2}$$

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

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

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

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

Dochodzimy więc do następującego równania:

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

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

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

$$w_0 + \boldsymbol{w}^T \boldsymbol{x}^{(i)} \geq 1 \text{ jeśli } y^{(i)} = 1$$
$$w_0 + \boldsymbol{w}^T \boldsymbol{x}^{(i)} \leq -1 \text{ jeśli } y^{(i)} = -1$$
$$\text{dla } i = 1... N$$

N oznacza tu liczbę przykładów w zestawie danych.

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

$$y^{(i)}\left(w_0 + \boldsymbol{w}^T \boldsymbol{x}^{(i)}\right) \geq 1 \, \forall_i$$

W rzeczywistości łatwiej minimalizować odwrotne wyrażenie $\frac{1}{2}\|\boldsymbol{w}\|^2$, 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.

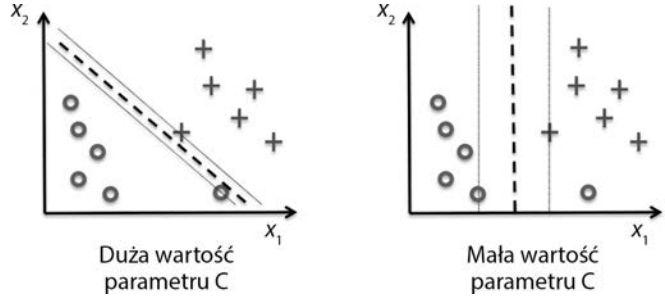
Wystarczy wstawić zmienną uzupełniającą do wzoru ograniczeń liniowych:

$$w_0 + \boldsymbol{w}^T \boldsymbol{x}^{(i)} \geq 1 - \xi^{(i)} \text{ jeśli } y^{(i)} = 1$$
$$w_0 + \boldsymbol{w}^T \boldsymbol{x}^{(i)} \leq -1 + \xi^{(i)} \text{ jeśli } y^{(i)} = -1$$
$$\text{dla } i = 1... N$$

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

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

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.



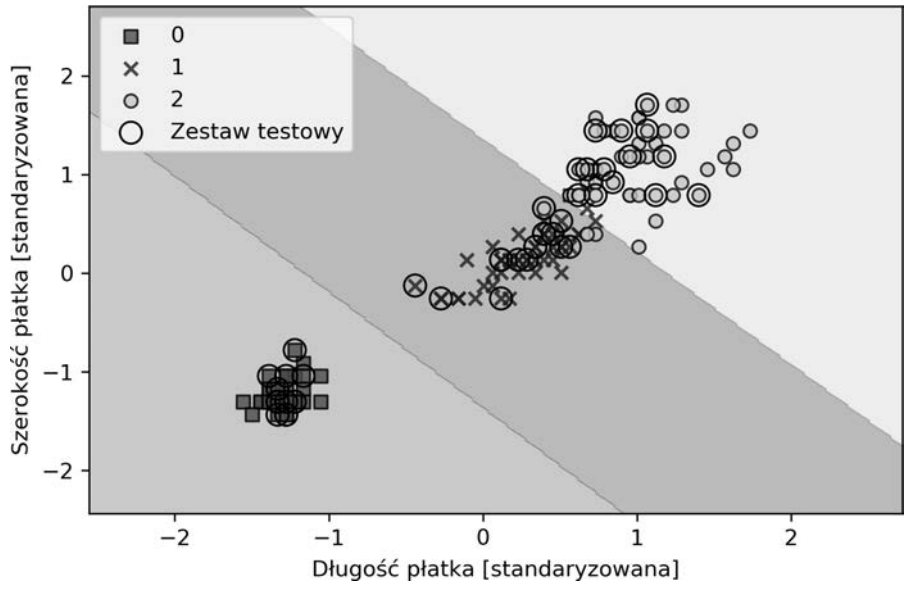
Rysunek 3.10. Wpływ zmiennej C na szerokość marginesu

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

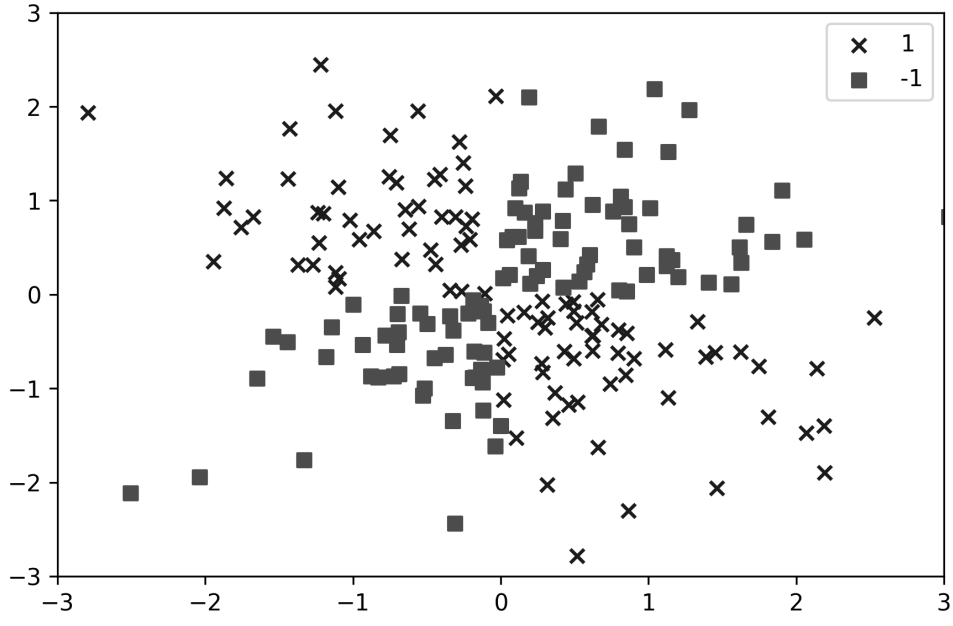
Kolejnym powodem wielkiej popularności algorytmu SVM wśród społeczności zajmującej się uczeniem maszynowym jest możliwość jego łatwej **kernelizacji** w celu rozwiązywania nieliniowych problemów klasyfikacji. Zanim zaczniemy omawiać koncepcję **jądra SVM**, zdefiniujmy i stwórzmy najpierw zbiór próbek,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



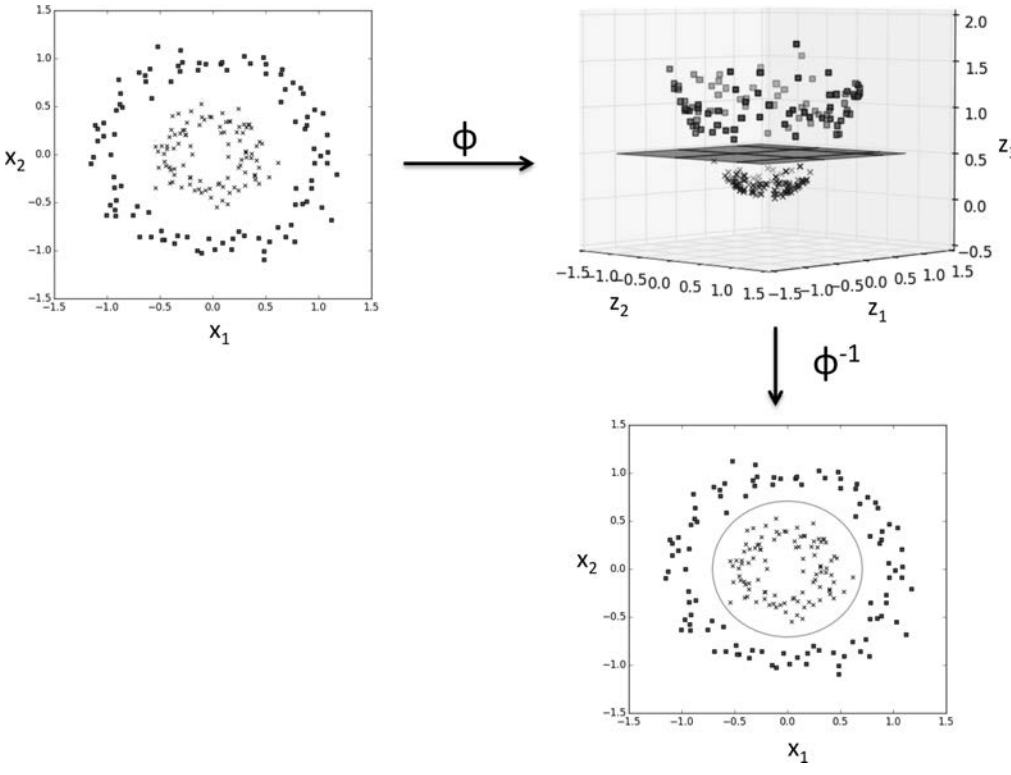
Rysunek 3.12. Zestaw danych wygenerowanych za pomocą bramki XOR

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 nierozdzielnymi 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 $\boldsymbol{x}^{(i)T}\boldsymbol{x}^{(j)}$ iloczynem funkcji $\phi\left(x^{(i)}\right)^T\phi\left(x^{(j)}\right)$. By zaoszczędzić kosztownej operacji bezpośredniego obliczania tego iloczynu skalarnego pomiędzy dwoma punktami, definiujemy tzw. **funkcję jądra**:

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

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

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

Często jest ono upraszczane do postaci:

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

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).

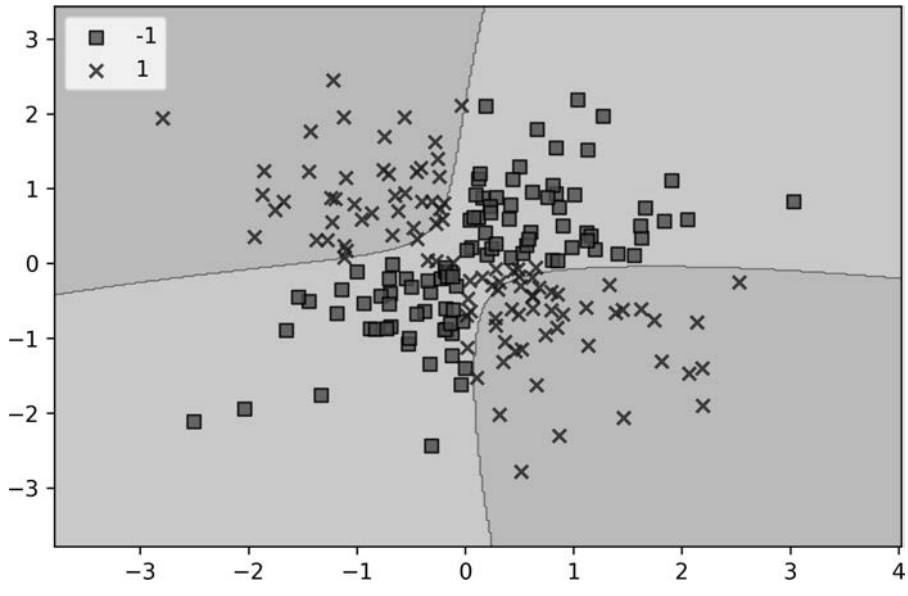
Skoro już znamy ogólne założenia sztuczki z funkcją jądra, sprawdźmy, czy jesteśmy w stanie wytrenować jądro SVM w taki sposób, żeby skutecznie wyrysowało nieliniową granicę decyzyjną rozdzielającą nasze próbki wygenerowane za pomocą bramki XOR. Wykorzystamy tu ponownie klasę `SVC` i zastąpimy parametr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

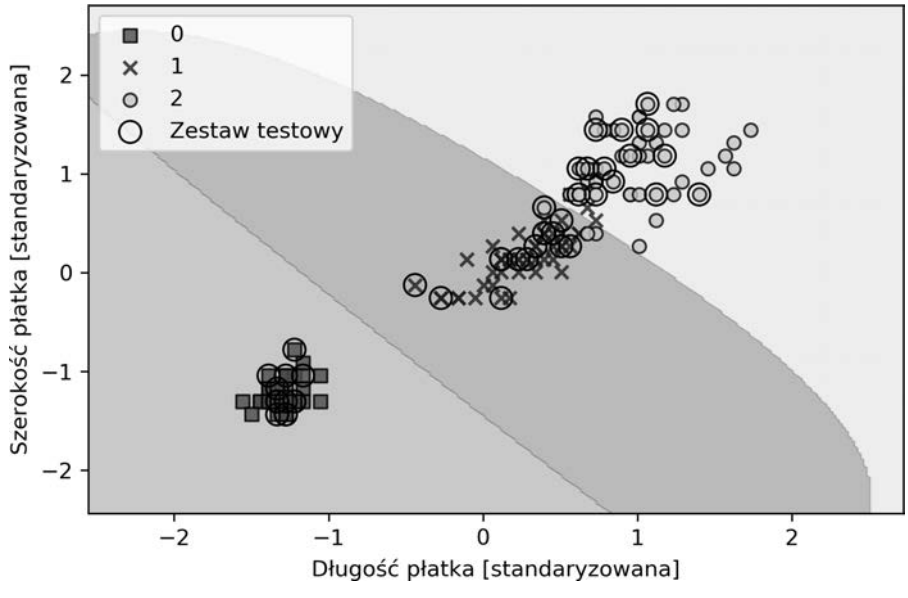
Parametr γ (zdefiniowany jako `gamma=0.1`) możemy uznać za obszar graniczny sfery Gaussa. Jeżeli zwiększymy wartość zmiennej γ, podniesiemy również wpływ (zasięg) próbek uczących, co doprowadzi do sztywniejszych i bardziej pofałdowanych granic decyzyjnych. Aby lepiej zrozumieć ten parametr, zastosujmy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

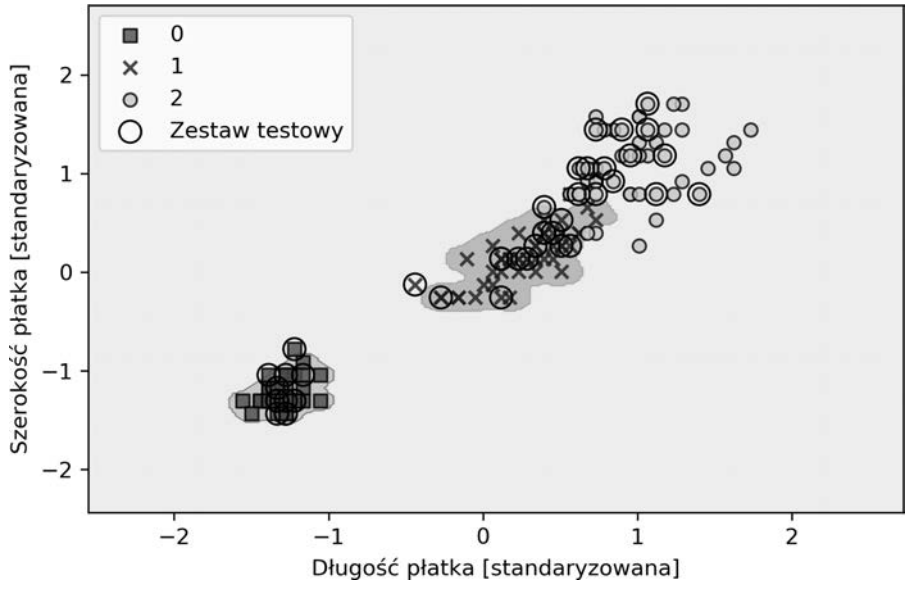


Rysunek 3.15. Nieliniowe granice decyzyjne uzyskane za pomocą niskiej wartości parametru γ

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.



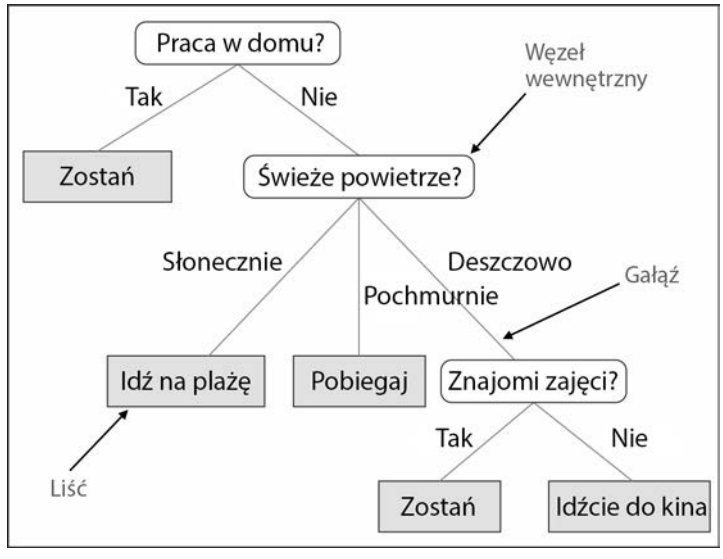
Rysunek 3.16. Wpływ zwiększenia wartości parametru γ na granice decyzyjne

Chociaż tak skonfigurowany model znakomicie dopasowuje dane uczące, klasyfikator prawdopodobnie będzie generował duży błąd nieprawidłowej klasyfikacji wobec nieznanych 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, nic jednak nie stoi na przeszkodzie, aby dostosować je do liczb rzeczywistych, takich jak tworzące zbiór danych Iris. Możemy np. po prostu zdefiniować wartość graniczną dla osi rzędnych (**szerokość działki**) i zadać binarne pytanie: „Czy szerokość działki $\geq$ 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}:

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

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). Zacznijmy od definicji entropii dla wszystkich **niepustych** klas $(p(i \mid t) \neq 0)$:

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

Wyrażenie $p(i \mid t)$ 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 $p(i=1 \mid t) = 1$ lub $p(i=0 \mid t) = 0$. Przy jednorodnym rozkładzie klas $p(i=1 \mid t) = 0{,}5$ i $p(i=0 \mid t) = 0{,}5$ 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:

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

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

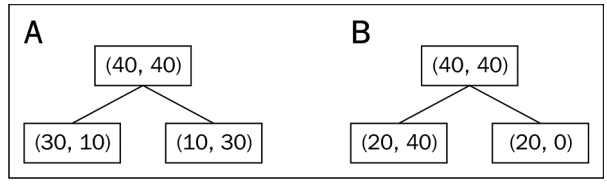
$$I_G(t) = 1 - \sum_{i=1}^{c} 0,5^2 = 0,5$$

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:

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

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):

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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()
```

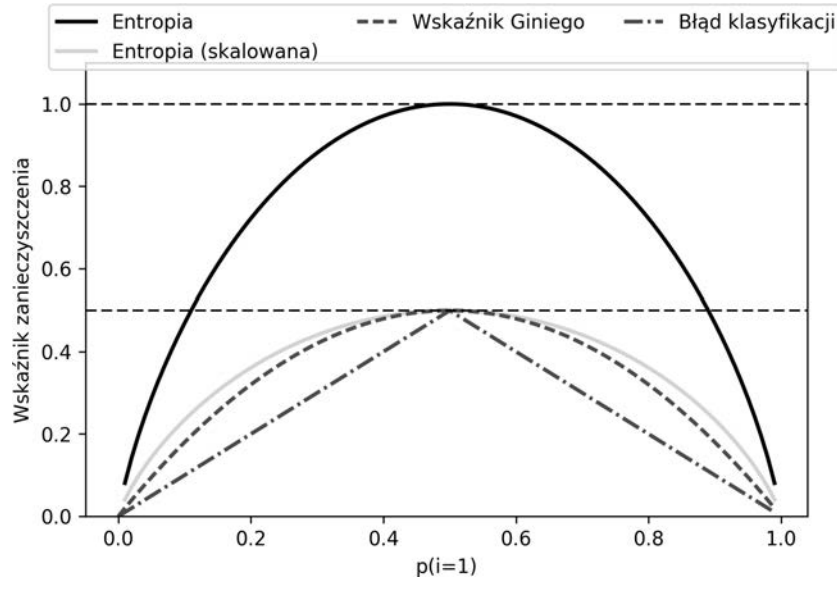
Wygenerowany wykres możemy zobaczyć na rysunku 3.19.

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:

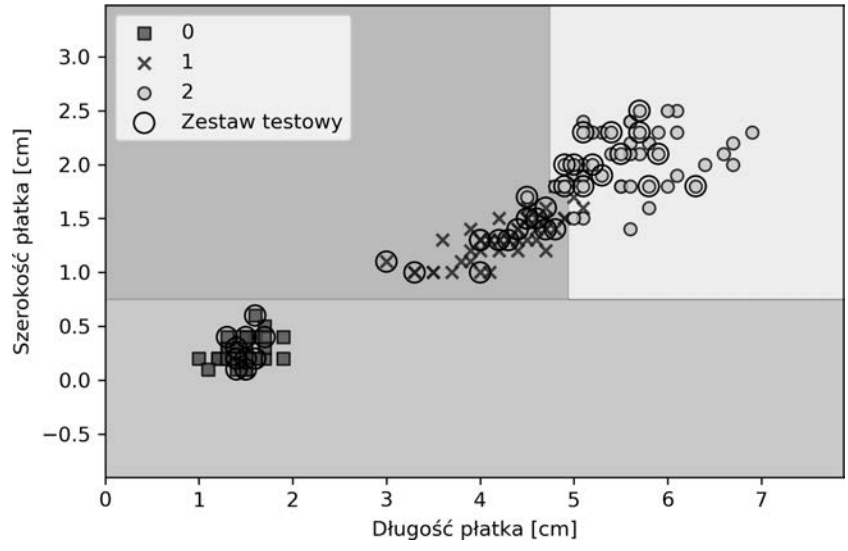
```
>>> from sklearn.tree import DecisionTreeClassifier
>>> tree = DecisionTreeClassifier(criterion='gini',
...                               max_depth=4, random_state=1)
>>> tree.fit(X_train, y_train)
>>> X_combined = np.vstack((X_train, X_test))
>>> y_combined = np.hstack((y_train, y_test))
>>> plot_decision_regions(X_combined, 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
```

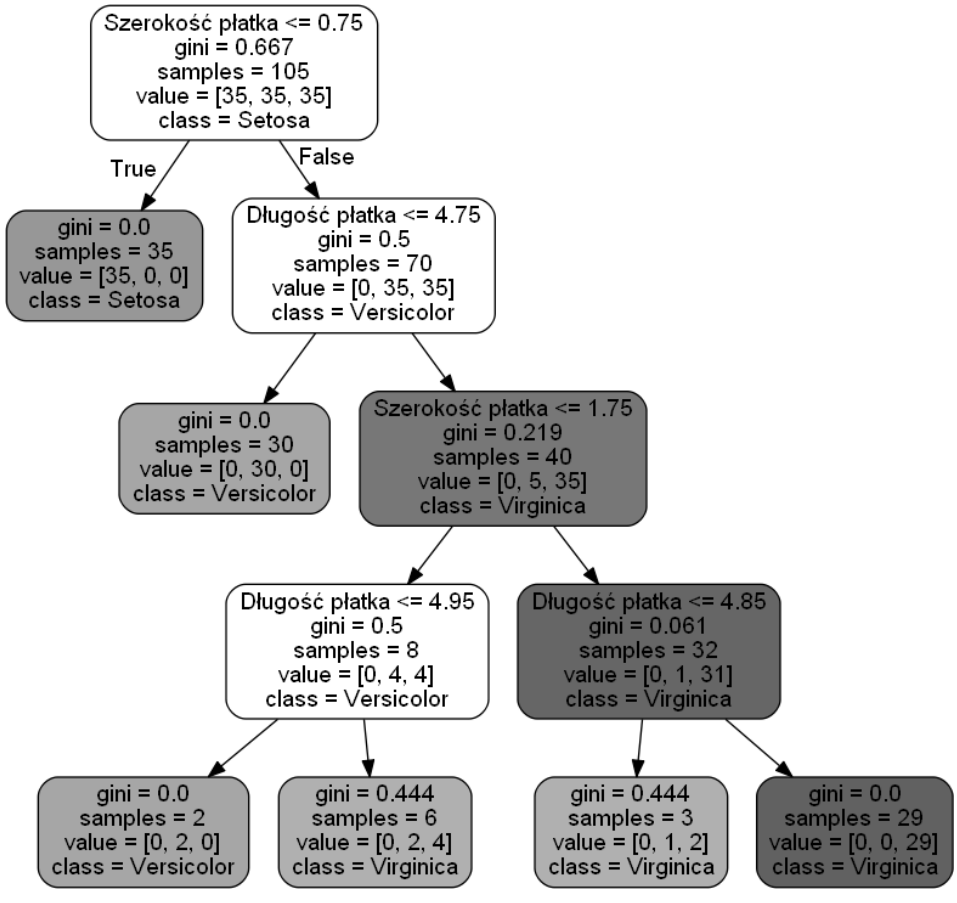
> W niektórych systemach być może będziesz musiał zainstalować pakiety zależne biblioteki `pydotplus` ręcznie, poprzez wpisanie poniższych poleceń:
>
> ```
> pip3 install graphviz
> pip3 install pyparsing
> ```

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
płatka', 'Szerokość płatka'])
...                                                  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 niezmienione; 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 dobieranych cech do każdego rozgałęzienia (podetap 2.1). Za pomocą rozmiaru próbek n kontrolujemy kompromis pomiędzy obciążeniem a wariancją losowego lasu.

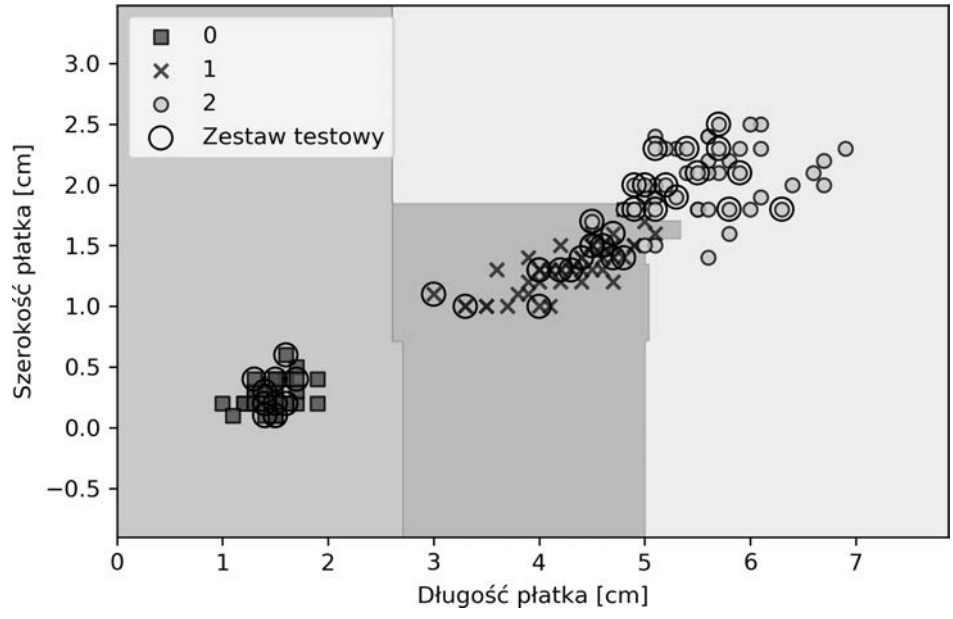
Zmniejszenie rozmiaru przykładów początkowych zwiększa zróżnicowanie pomiędzy poszczególnymi drzewami, ponieważ maleje prawdopodobieństwo wystąpienia określonego przykładu uczącego w danej próbce początkowej. Zatem ograniczenie rozmiaru próbek początkowych skutkuje zwiększeniem **losowości** lasu, co 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 Giniego 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

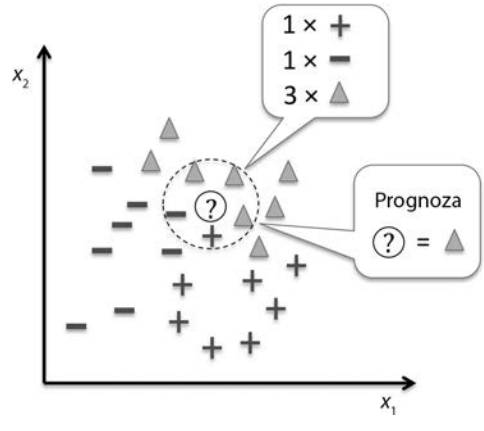
Algorytmy uczenia maszynowego możemy podzielić na modele **parametryczne** i **nieparametryczne**. Za pomocą modeli parametrycznych oszacowujemy parametry zestawu uczącego, dzięki czemu jesteśmy w stanie wytrenować funkcję, która będzie klasyfikowała nowe punkty danych bez konieczności dalszego wykorzystywania 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

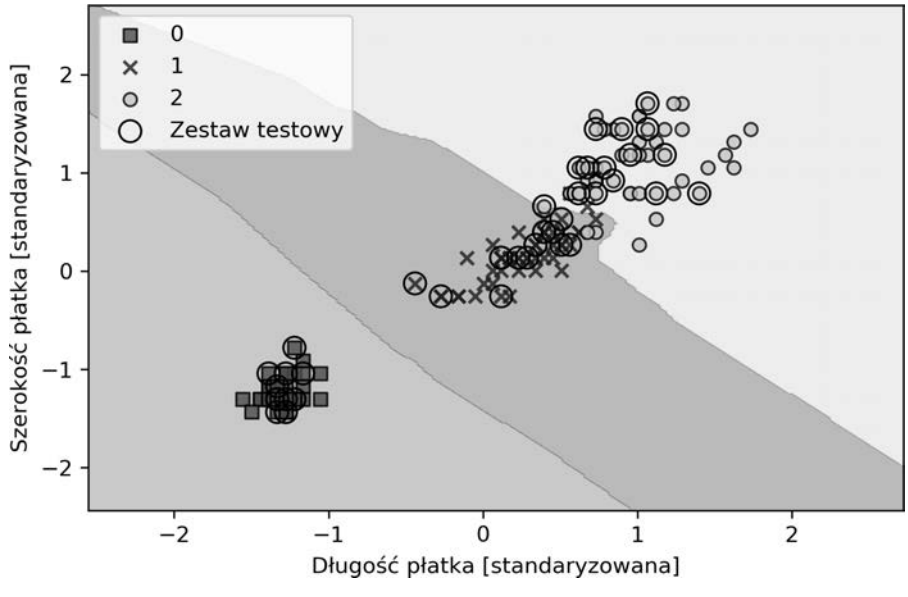
Na podstawie wybranej metryki odległości algorytm KNN wyszukuje w zestawie danych uczących k próbek znajdujących się najbliżej klasyfikowanego punktu lub wykazujących największe podobieństwo do niego. Etykieta klasy tej próbki zostaje określona poprzez większościowe głosowanie przeprowadzone pomiędzy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

Największą zaletą takiego pamięciowego algorytmu jest natychmiastowe adaptowanie się klasyfikatora w trakcie pobierania nowych danych uczących. Równoważy to jednak główna wada, polegająca na liniowym wzroście złożoności obliczeniowej wraz z liczbą próbek uczących (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 płatka [standaryzowana]')
>>> plt.ylabel('Szerokość płatka [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:

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

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*.

Klątwa wymiarowości

Należy wspomnieć, że algorytm KNN jest bardzo wrażliwy na przetrenowanie z powodu **klątwy 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 klątwy wymiarowości. Opisem tych rozwiązań zajmiemy się w następnym rozdziale.

Podsumowanie

W tym rozdziale była mowa o wielu różnych algorytmach uczenia maszynowego, które są wykorzystywane do rozwiązywania liniowych oraz nieliniowych problemów. Dowiedzieliśmy się, że model drzew decyzyjnych jest wyjątkowo atrakcyjny, pod warunkiem że zadbamy 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

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

Z

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 splotowych, 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 regresywna
- 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.

ISBN 978-83-283-5121-9