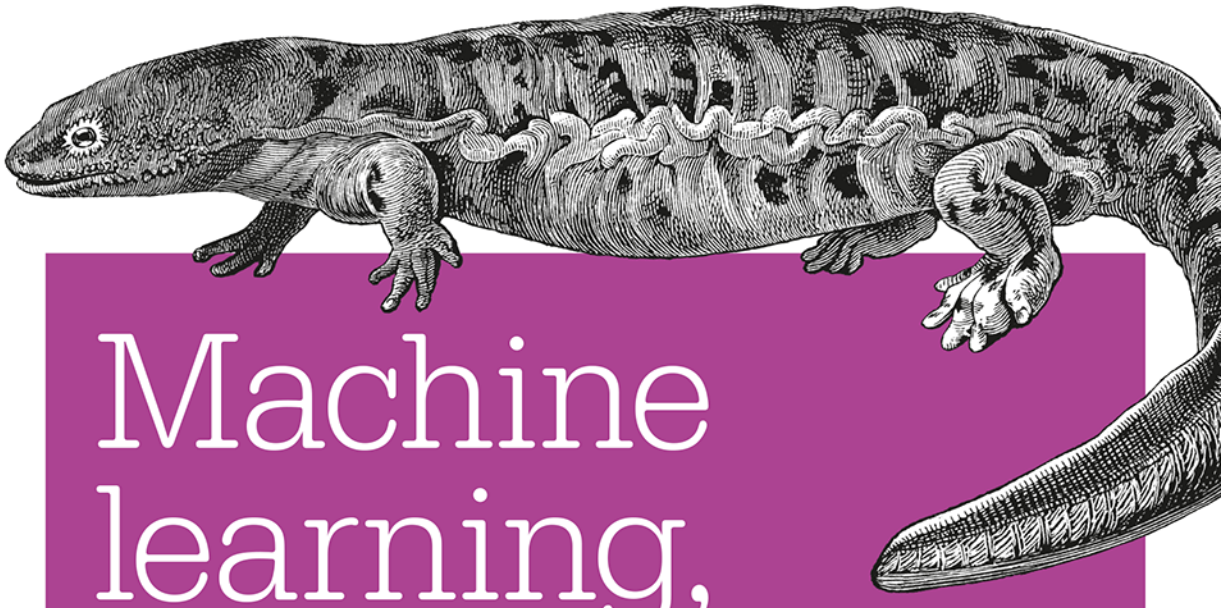
O'REILLY®



Machine learning, Python i data science

WPROWADZENIE

Helion

Andreas C. Müller, Sarah Guido

Tytuł oryginału: Introduction to Machine Learning with Python: A Guide for Data Scientists

Tłumaczenie: Michał Sternik

ISBN: 978-83-8322-751-1

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/malepv
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 2.

Nadzorowane uczenie maszynowe

Jak wspomnieliśmy wcześniej, nadzorowane uczenie maszynowe jest jednym z najczęściej używanych i najskuteczniejszych rodzajów uczenia maszynowego. W tym rozdziale szczegółowo opiszemy nadzorowane uczenie maszynowe i omówimy kilka popularnych algorytmów nadzorowanego uczenia maszynowego. Zastosowanie nadzorowanego uczenia maszynowego widzieliśmy już w rozdziale 1.: klasyfikowanie irysów do kilku gatunków przy użyciu fizycznych pomiarów kwiatów.

Pamiętaj, że uczenie nadzorowane jest używane zawsze, gdy chcemy prognozować określony wynik na podstawie danych wejściowych, a mamy przykłady par danych wejściowych/wyjściowych. Tworzymy model uczenia maszynowego z tych par danych wejścia/wyjścia, które składają się na nasz zestaw uczący. Celem jest tworzenie dokładnych prognoz dla nowych, nigdy wcześniej niewidzianych danych. Nadzorowane uczenie maszynowe często wymaga działań wykonywanych przez ludzi w celu zbudowania zestawu uczącego, ale późniejsze działania są już automatyczne, co często przyspiesza skądinąd pracochłonne lub niewykonalne zadanie.

Klasyfikacja i regresja

Istnieją dwa główne typy nadzorowanych problemów uczenia maszynowego, zwane *klasyfikacją* i *regresją*.

W klasyfikacji celem jest stworzenie prognozy *etykiety klasy*, którą wybiera się z predefiniowanej listy możliwości. W rozdziale 1. posłużyliśmy się przykładem zaklasyfikowania irysów do jednego z trzech możliwych gatunków. Klasyfikacja zostaje czasami podzielona na *klasyfikację binarną*, co jest szczególnym przypadkiem rozróżnienia dokładnie dwóch klas, oraz *klasyfikację wieloklasową*, która obejmuje więcej niż dwie klasy. Możesz myśleć o klasyfikacji binarnej jako o próbie odpowiedzi na pytanie „tak" lub „nie". Dzielenie wiadomości e-mail na spam i te, które nim nie są, to przykład problemu z klasyfikacją binarną. W tym zadaniu z klasyfikacją binarną pytanie brzmiałoby: czy ta wiadomość e-mail jest spamem?

W klasyfikacji binarnej często mówimy, że jedna klasa jest klasą *pozytywną*, a druga *negatywną*. W tym przypadku pozytyw nie oznacza korzyści ani dodanej wartości, a raczej to, czym jest przedmiot badania. Dlatego przy wyszukiwaniu spamu „pozytywny" może oznaczać, że coś do niego należy. To, którą z tych dwóch klas nazywa się pozytywną, jest często kwestią subiektywną i specyficzną dla danej dziedziny.

Z drugiej strony zadanie dotyczące irysa jest przykładem problemu klasyfikacji wieloklasowej. Innym przykładem jest prognozowanie języka witryny na podstawie zamieszczonego w niej tekstu. W tym przypadku klasy byłyby predefiniowaną listą możliwych języków.

W przypadku zadań regresyjnych celem jest prognozowanie liczby ciągłej lub, mówiąc językiem programistów, *liczby zmiennoprzecinkowej* (w matematyce *liczby rzeczywistej*). Prognozowanie rocznego dochodu danej osoby na podstawie jej edukacji, wieku i miejsca zamieszkania jest przykładem zadania regresyjnego. Przy przewidywaniu dochodu prognozowana wartość jest *kwotą* i może być dowolną liczbą z danego zakresu. Innym przykładem zadania regresyjnego jest prognozowanie plonów z upraw kukurydzy przy danych atrybutach, takich jak poprzednie plony, pogoda i liczba osób pracujących w gospodarstwie. W tym przypadku wynik też może być dowolną liczbą.

Łatwym sposobem rozróżnienia zadań klasyfikacyjnych i regresyjnych jest określenie, czy w wynikach zachodzi jakaś ciągłość. Jeśli istnieje ciągłość między możliwymi wynikami, problem jest problemem regresji. Rozważmy prognozowanie rocznego dochodu. W danych wyjściowych występuje wyraźna ciągłość. Różnica między dochodami kogoś, kto rocznie zarabia 40 000 dolarów, a dochodami osoby zarabiającej 40 001 dolarów nie jest namacalna, mimo że są to różne kwoty; nie jest istotne czy nasz algorytm prognozuje 39 999 USD czy 40 001 USD, podczas gdy powinien prognozować 40 000 USD.

Natomiast w przypadku zadania rozpoznania języka strony internetowej (co stanowi problem klasyfikacyjny) granica jest wyraźna. Witryna internetowa jest albo w jednym języku, albo w innym. Języki nie mogą się różnić w *jakimś stopniu*, albo jest to angielski, albo francuski[1].

Uogólnianie, nadmierne dopasowanie i niedopasowanie

W uczeniu nadzorowanym chodzi o zbudowanie modelu na podstawie danych uczących, a następnie tworzenie dokładnych prognoz na temat nowych, nieznanych danych, które mają te same cechy co użyty zestaw uczący. Jeśli model dokładnie prognozuje na nowych danych, mówimy, że może *uogólniać* od zestawu uczącego do zestawu testowego. Chcemy zbudować model, który będzie uogólniać jak najdokładniej.

Zwykle budujemy model w taki sposób, aby mógł trafnie prognozować na zestawie uczącym. Jeśli zestawy uczące i testowe mają wystarczająco dużo cech wspólnych, oczekujemy, że model będzie również dokładnie uogólniał dane z zestawu testowego. Jednak są pewne przypadki, w których może się to nie udać. Przykładowo, jeśli pozwolimy sobie na budowanie bardzo złożonych modeli, na zestawie uczącym możemy być tak dokładni, jak nam się podoba.

Aby zilustrować ten punkt, weźmy pod uwagę zmyślony przykład. Powiedzmy, że początkujący badacz danych chce prognozować, czy klient kupi łódź, na podstawie danych o wcześniejszych nabywcach i osobach, o których wiadomo, że nie są zainteresowane jej kupnem[2]. Celem jest wysyłanie reklam w wiadomościach e-mail do tych osób, które prawdopodobnie dokonają zakupu, a nie do odbiorców niezainteresowanych kupnem łodzi.

[1] Prosimy lingwistów o wybaczenie tak uproszczonego przedstawienia języków jako odrębnych i stałych bytów.

[2] W prawdziwym świecie jest to trudny problem. Chociaż wiemy, że inni klienci jeszcze nie kupili od nas łodzi, być może kupili ją od kogoś innego lub nadal oszczędzają i planują kupić ją w przyszłości.

Załóżmy, że mamy takie dane dotyczące klientów, jakie przedstawiono w tabeli 2.1.

Tabela 2.1. Przykładowe dane dotyczące klientów

Wiek	Liczba posiadanych samochodów	Posiada dom	Liczba dzieci	Stan cywilny	Posiada psa	Kupił łódź
66	1	tak	2	wdowiec/wdowa	nie	tak
52	2	tak	3	żonaty	nie	tak
22	0	nie	0	żonaty	tak	nie
25	1	nie	1	kawaler	nie	nie
44	0	nie	2	rozwiedziony	tak	nie
39	1	tak	2	żonaty	tak	nie
26	1	nie	2	kawaler	nie	nie
40	3	tak	1	żonaty	tak	nie
53	2	tak	2	rozwiedziony	nie	tak
64	2	tak	3	rozwiedziony	nie	nie
58	2	tak	2	żonaty	tak	tak
33	1	nie	1	kawaler	nie	nie

Po dłuższym przyjrzeniu się danym początkujący analityk danych wymyśla taką zasadę: „Jeśli klient ma więcej niż 45 lat, ma mniej niż 3 dzieci lub nie jest rozwiedziony, chce kupić łódź". Na pytanie, jak dobrze działa ta jego reguła, odpowiada: jest w 100% dokładna! I rzeczywiście, w przypadku danych zawartych w tabeli reguła jest całkowicie precyzyjna. Istnieje wiele możliwych reguł, które moglibyśmy wymyślić i które doskonale odpowiadałyby na pytanie, czy ktoś z tego zestawu danych chciał kupić łódź. Żaden wiek nie pojawia się dwukrotnie w danych, więc można powiedzieć, że tylko osoby w wieku 66, 52, 53 lub 58 lat chcą kupić łódź. Chociaż możemy stworzyć wiele reguł, które dobrze działają na tych danych, pamiętaj, że nie jesteśmy zainteresowani prognozowaniem dla tego zestawu danych; znamy już odpowiedzi dla tych klientów. Chcemy wiedzieć, jakie jest prawdopodobieństwo, że *nowi klienci* kupią łódź. Zamierzamy więc znaleźć regułę, która sprawdzi się dobrze w przypadku nowych klientów, a osiągnięcie 100-procentowej dokładności w zestawie uczącym nam w tym nie pomaga. Nie możemy oczekiwać, że reguła, którą wymyślił analityk danych, będzie działać bardzo dobrze w przypadku nowych klientów. Wydaje się zbyt skomplikowana i jest poparta za małym zestawem danych. Przykładowo część reguły „lub nie jest rozwiedziony" dotyczy tylko jednego klienta.

Jedyną miarą tego, czy algorytm będzie działał dobrze na nowych danych, jest jego ocena w zestawie testowym. Jednak intuicyjnie[3] spodziewamy się, że proste modele lepiej uogólniają nowe dane. Gdyby zasada brzmiała: „Osoby powyżej 50. roku życia chcą kupić łódź", a to wyjaśniałoby zachowanie wszystkich klientów, zaufalibyśmy jej bardziej niż zasadzie dotyczącej dzieci i stanu cywilnego z wyłączeniem wieku. Dlatego zawsze chcemy znaleźć najprostszy model. Tworzenie modelu, który jest zbyt złożony w stosunku do zasobu posiadanych informacji, co zrobił przykładowy początkujący analityk danych, nazywa się *nadmiernym dopasowaniem* (ang. *overfitting*). Występuje ono, gdy model jest zbyt ściśle dopasowany do specyfiki zestawu uczącego i działa dobrze na

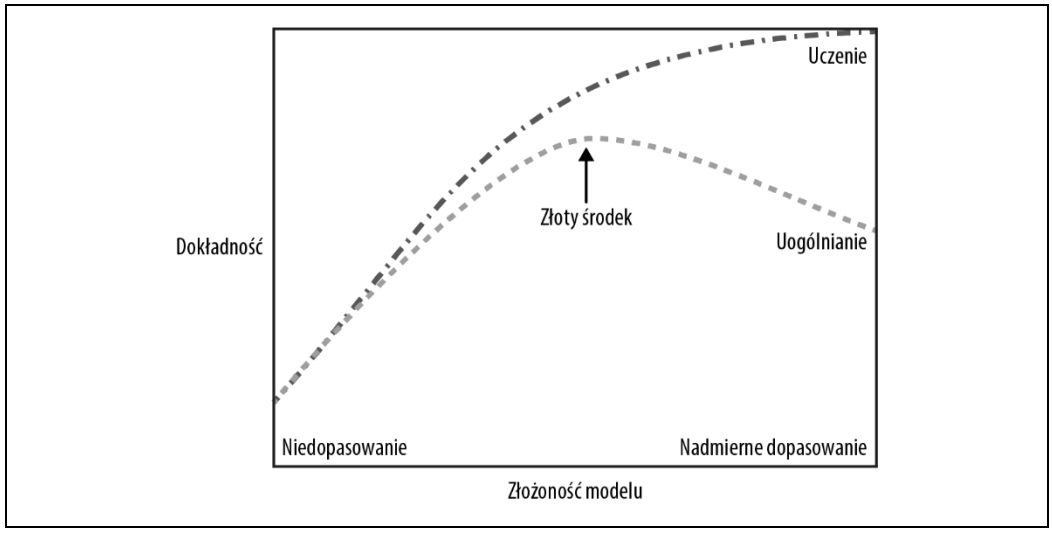
[3] A także udowodnić matematycznie.

zestawie uczącym, ale nie potrafi uogólnić nowych danych. Z drugiej strony, jeśli model jest zbyt prosty — np. „Każdy, kto jest właścicielem domu, kupuje łódź” — może nie uchwycić wszystkich aspektów i zmienności danych, a model będzie działał źle nawet w przypadku danych uczących. Wybór zbyt prostego modelu nazywa się *niedopasowaniem* (ang. *underfitting*).

Im bardziej złożony model, tym lepiej będziemy w stanie prognozować na danych uczących. Jeśli jednak będzie zbyt złożony, będziemy się zbytnio koncentrować na każdym indywidualnym punkcie danych w zestawie uczącym, a model nie uogólni się dobrze na nowe dane.

Pomiędzy nimi znajduje się złoty środek, który zapewnia najlepszą wydajność generalizacji. To jest właśnie ten model, który chcemy znaleźć.

Kompromis między nadmiernym a niedostatecznym dopasowaniem pokazano na rysunku 2.1.



Rysunek 2.1. Kompromis między złożonością modelu a uczeniem i dokładnością testów

Relacja złożoności modelu do rozmiaru zestawu danych

Należy zauważyć, że złożoność modelu jest ściśle związana ze zmiennością danych wejściowych zawartych w zestawie danych uczących: im większa jest różnorodność punktów danych, które zawiera zestaw, tym bardziej złożony model można stworzyć bez ryzyka nadmiernego dopasowania. Zwykle gromadzenie większej liczby punktów danych zapewnia większą różnorodność, więc rozbudowane zestawy danych pozwalają na tworzenie bardziej złożonych modeli. Jednak zwykłe powielenie tych samych punktów danych lub zebranie bardzo podobnych danych nie pomoże.

Wróćmy do przykładu sprzedaży łodzi. Jeśli weźmiemy pod uwagę 10 000 więcej wierszy danych, które dotyczą klientów, i wszystkie z nich będą zgodne z zasadą „Jeśli klient ma więcej niż 45 lat i ma mniej niż 3 dzieci lub nie jest rozwiedziony, to chce kupić łódź”, to będziemy o wiele bardziej skłonni uwierzyć, że jest to dobra zasada, niż wtedy, gdy została opracowana przy użyciu tylko 12 wierszy, które przedstawiono w tabeli 2.1.

Posiadanie większego zbioru danych i tworzenie odpowiednio bardziej złożonych modeli może często zdziałać cuda w przypadku nadzorowanych zadań uczenia maszynowego. W tej książce skupimy się na pracy z zestawami danych o stałych rozmiarach. W prawdziwym świecie często możesz wcześniej zdecydować, ile danych chcesz zebrać, co może być bardziej korzystne od ulepszania i dostrajania modelu. Nigdy nie lekceważ potęgi większego zasobu danych.

Nadzorowane algorytmy uczenia maszynowego

Omówimy teraz najpopularniejsze algorytmy uczenia maszynowego i wyjaśnimy, jak się uczą na podstawie danych i jak tworzą prognozy. Opiszemy również, w jaki sposób koncepcja złożoności modelu działa dla każdego z tych modeli, i przedstawimy przegląd tego, jak każdy algorytm buduje model. Przeanalizujemy mocne i słabe strony każdego algorytmu oraz podpowiemy, do jakiego rodzaju danych najlepiej go zastosować. Wyjaśnimy również znaczenie najważniejszych parametrów i opcji[4]. Wiele algorytmów zawiera klasyfikację i wariant regresji, opiszemy tutaj obie kwestie.

Szczegółowe zapoznawanie się z opisami każdego algorytmu nie jest konieczne, ale zrozumienie modeli pozwoli lepiej poznać różne sposoby działania algorytmów uczenia maszynowego. Ten rozdział może być również używany jako przewodnik i jeśli nie będziesz mieć pewności co do działania któregoś z algorytmów, możesz do niego wrócić.

Przykładowe zestawy danych

Użyjemy kilku zestawów danych do zilustrowania różnych algorytmów. Niektóre zestawy danych będą małe i syntetyczne (tj. zmyślone), zaprojektowane w celu uwypuklenia określonych aspektów algorytmów. Inne będą dużymi, rzeczywistymi przykładami.

Przykładem syntetycznego zestawu danych klasyfikacji dwuklasowej jest zestaw danych `forge`, który ma dwie cechy. Poniższy kod tworzy wykres punktowy (przedstawiony na rysunku 2.2), który wizualizuje wszystkie punkty danych w tym zestawie danych. Pierwsza cecha znajduje się na osi *x*, a druga na osi *y* wykresu. Jak zawsze w przypadku wykresów punktowych, każdy punkt danych jest reprezentowany jako jedna kropka. Kolor i kształt kropki wskazują na jego klasę:

`In[2]:`

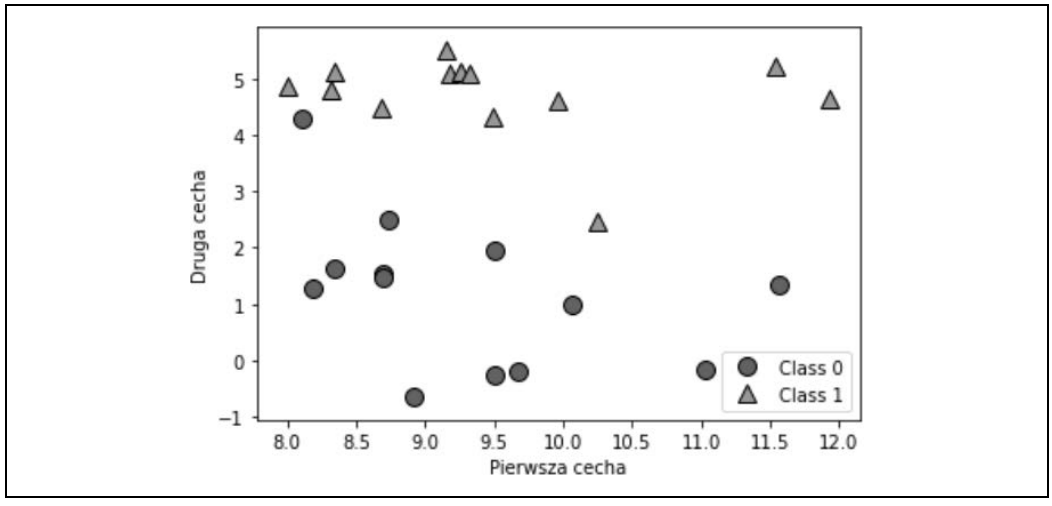
```
# wygeneruj zestaw danych
    X, y = mglearn.datasets.make_forge()

# stwórz wykres
    mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
    plt.legend(["Class 0", "Class 1"], loc=4)
    plt.xlabel("Pierwsza cecha")
    plt.ylabel("Druga cecha")
    print("X.shape: {}".format(X.shape))
```

`Out[2]:`

```
    X.shape: (26, 2)
```

[4] Omówienie ich wszystkich wykracza poza zakres książki, dlatego po więcej informacji odsyłamy do dokumentacji biblioteki `scikit-learn` (*http://scikit-learn.org/stable/documentation*).



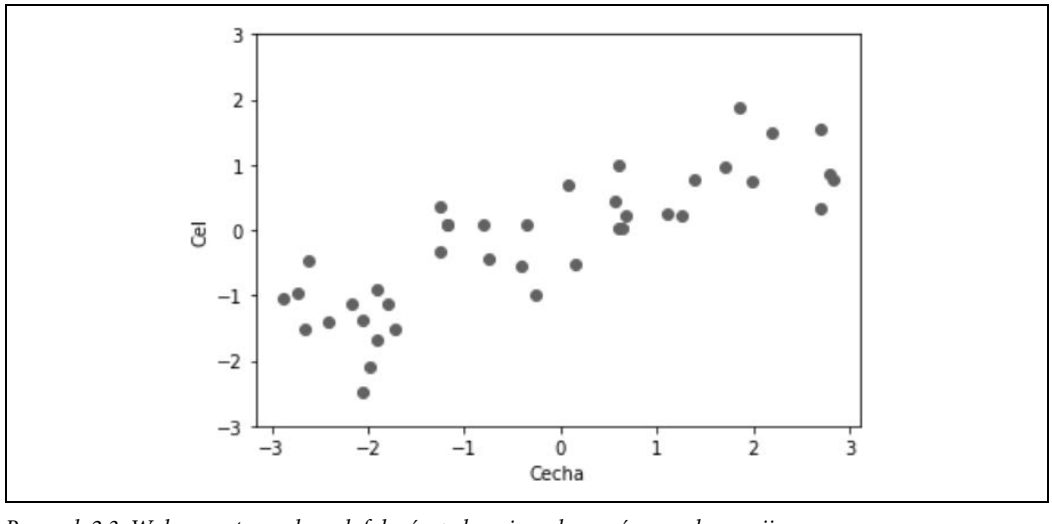
Rysunek 2.2. Wykres punktowy zestawu danych forge

Jak widać po zawartości `X.shape`, ten zestaw danych składa się z 26 punktów danych i 2 cech.

Aby zilustrować algorytmy regresji, użyjemy zestawu danych syntetycznych `wave`. Zestaw danych `wave` ma jedną cechę wejściową i ciągłą zmienną docelową (lub *odpowiedź*), którą chcemy modelować. Na rysunku 2.3 przedstawiono wykres, który zawiera pojedynczą cechę na osi *x* i cel regresji (wynik) na osi *y*:

`In[3]:`

```
X, y = mglearn.datasets.make_wave(n_samples=40)
plt.plot(X, y, 'o')
plt.ylim(-3, 3)
plt.xlabel("Cecha")
plt.ylabel("Cel")
```



Rysunek 2.3. Wykres zestawu danych fal; oś x pokazuje cechę, a oś y — cel regresji

Używamy tych bardzo prostych, niskowymiarowych zestawów danych, ponieważ możemy je łatwo zwizualizować — wydrukowana strona ma dwa wymiary, więc dane z więcej niż dwiema cechami są trudne do pokazania. Jakakolwiek intuicja wywodząca się z zestawów danych z kilkoma cechami (zwanych również *niskowymiarowymi* zestawami danych) może nie dotyczyć zestawów danych z wieloma cechami (*wielowymiarowych* zestawów danych). Dopóki masz to na uwadze, sprawdzanie algorytmów na niskowymiarowych zestawach danych może być bardzo pouczające.

Uzupełnimy te małe syntetyczne zbiory danych o dwa rzeczywiste ich zestawy, które zawarto w bibliotece `scikit-learn`. Jednym z nich jest zestaw danych, który dotyczy przypadków raka piersi z Wisconsin (Breast Cancer — w skrócie `cancer`) i w którym zarejestrowano kliniczne pomiary guzów raka piersi. Każdy guz jest oznaczony jako łagodny (w przypadku guzów nieszkodliwych) lub złośliwy (w przypadku guzów nowotworowych), a zadaniem jest nauczenie się prognozowania, czy guz jest złośliwy, na podstawie pomiarów tkanki.

Dane można załadować za pomocą funkcji `load_breast_cancer` z biblioteki `scikit-learn`:

`In[4]:`

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
print("cancer.keys(): \n{}".format(cancer.keys()))
```

`Out[4]:`

```
cancer.keys():
dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names', 'filename'])
```

Zestawy danych zawarte w bibliotece `scikit-learn` są zwykle przechowywane jako obiekty klasy `Bunch`, które zawierają pewne informacje o zestawie danych, a także rzeczywiste dane. O obiektach klasy `Bunch` musisz wiedzieć, że zachowują się jak słowniki, z dodatkową korzyścią w postaci możliwości uzyskania dostępu do wartości z użyciem kropki (np. `bunch.key` zamiast `band['key']`).

Zestaw składa się z 569 punktów danych, z których każdy zawiera 30 cech:

`In[5]:`

```
print("Kształt danych cancer: {}".format(cancer.data.shape))
```

`Out[5]:`

```
Kształt danych cancer: (569, 30)
```

Z tych 569 punktów danych 212 jest oznaczonych jako złośliwe (ang. *meligant*), a 357 jako łagodne (ang. *benign*):

`In[6]:`

```
print("Liczba próbek na klasę:\n{}".format(
    {n: v for n, v in zip(cancer.target_names, np.bincount(cancer.target))}))
```

`Out[6]:`

```
Liczba próbek na klasę:
{'benign': 357, 'malignant': 212}
```

Aby uzyskać opis znaczenia semantycznego każdej cechy, możemy spojrzeć na atrybut `feature_names`:

In[7]:

```
print("Nazwy cech:\n{}".format(cancer.feature_names))
```

Out[7]:

```
Nazwy cech:
['mean radius' 'mean texture' 'mean perimeter' 'mean area'
'mean smoothness' 'mean compactness' 'mean concavity'
'mean concave points' 'mean symmetry' 'mean fractal dimension'
'radius error' 'texture error' 'perimeter error' 'area error'
'smoothness error' 'compactness error' 'concavity error'
'concave points error' 'symmetry error' 'fractal dimension error'
'worst radius' 'worst texture' 'worst perimeter' 'worst area'
'worst smoothness' 'worst compactness' 'worst concavity'
'worst concave points' 'worst symmetry' 'worst fractal dimension']
```

Jeśli chcesz poznać więcej informacji o danych, przeczytaj zawartość `cancer.DESCR`.

Będziemy również korzystać z rzeczywistego zestawu danych regresji — Boston Housing. Związane z nim zadanie polega na prognozowaniu mediany wartości domów w kilku dzielnicach Bostonu w latach 70. XX wieku przy użyciu takich informacji jak wskaźnik przestępczości, bliskość rzeki Charles, dostępność autostrad itd. Zestaw danych zawiera 506 punktów danych, opisanych przez 13 cech:

In[8]:

```
from sklearn.datasets import load_boston
boston = load_boston()
print("Kształt danych: {}".format(boston.data.shape))
```

Out[8]:

```
Kształt danych: (506, 13)
```

Tak jak poprzednio, możesz uzyskać więcej informacji o zestawie danych, jeśli przeczytasz atrybut `DESCR` obiektu `boston`. Rozszerzymy ten zestaw danych na nasze potrzeby, poza 13 cechami wejściowymi w postaci pomiarów weźmiemy pod uwagę wszystkie produkty (zwane także *interakcjami*) między cechami. Innymi słowy, uwzględnimy cechy nie tylko w postaci wskaźnika przestępczości i dostępności autostrad, ale także w postaci iloczynu współczynnika przestępczości i dostępności autostrady. Włączenie takiej pochodnej cechy nazywa się *inżynierią cech*, którą omówimy bardziej szczegółowo w rozdziale 4. Ten wyprowadzony zestaw danych można załadować za pomocą funkcji `load_extended_boston`:

In[9]:

```
X, y = mglearn.datasets.load_extended_boston()
print("X.shape: {}".format(X.shape))
```

Out[9]:

```
X.shape: (506, 104)
```

Wynikowe 104 cechy to 13 oryginalnych funkcji wraz ze zbiorem 91 możliwych kombinacji 2 funkcji w ramach tych 13[5].

[5] Nazywa się to współczynnikiem dwumianowym, czyli liczbą kombinacji k elementów, które można wybrać ze zbioru n elementów. Często jest to zapisywane jako $\binom{n}{k}$ i wymawiane jako „n brane po k" — w tym przypadku „13 brane po 2".

Użyjemy tych zestawów danych do wyjaśnienia i zilustrowania właściwości różnych algorytmów uczenia maszynowego. Na razie jednak przejdźmy do samych algorytmów. Po pierwsze, wrócimy do algorytmu *k*-najbliższych sąsiadów (*k*-NN — z ang. *k-nearest neighbours*), który widzieliśmy w poprzednim rozdziale.

k-najbliższych sąsiadów

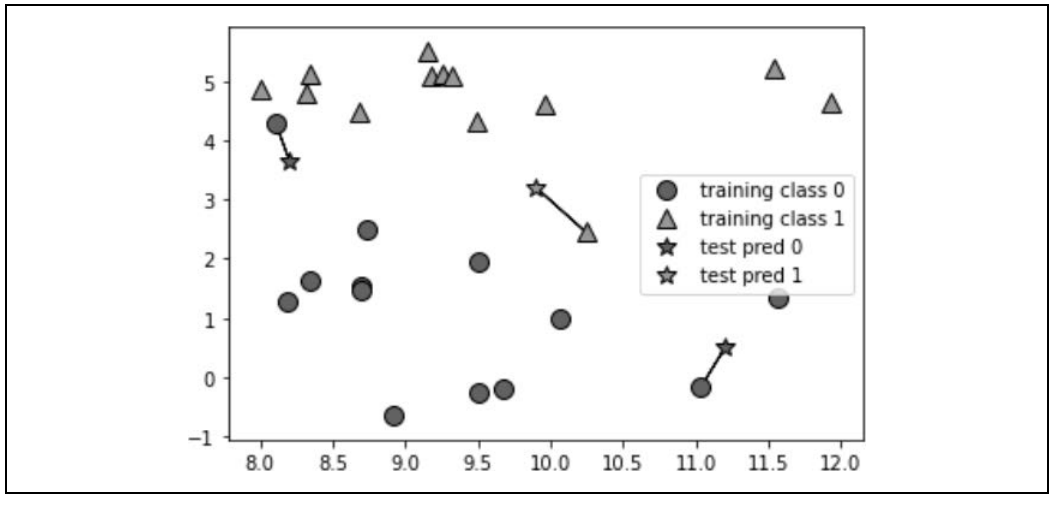
To prawdopodobnie najprostszy algorytm uczenia maszynowego. Budowanie modelu polega wyłącznie na przechowywaniu zestawu danych uczących. Aby prognozować nowy punkt danych, *k*-NN wyszukuje najbliższe punkty danych w uczącym zestawie danych — jego „najbliższych sąsiadów".

Klasyfikacja k-sąsiadów

W swojej najprostszej wersji algorytm *k*-NN bierze pod uwagę tylko dokładnie jednego najbliższego sąsiada, będącego najbliższym punktem z danych uczących względem punktu, dla którego chcemy tworzyć prognozę. Jest ona wtedy po prostu znanym wynikiem dla tego punktu uczącego. Przykład przypadku klasyfikacji w zestawie danych `forge` zilustrowano na rysunku 2.4:

`In[10]:`

```
mglearn.plots.plot_knn_classification(n_neighbors=1)
```



Rysunek 2.4. Prognozy stworzone przez model jednego najbliższego sąsiada na zestawie danych forge

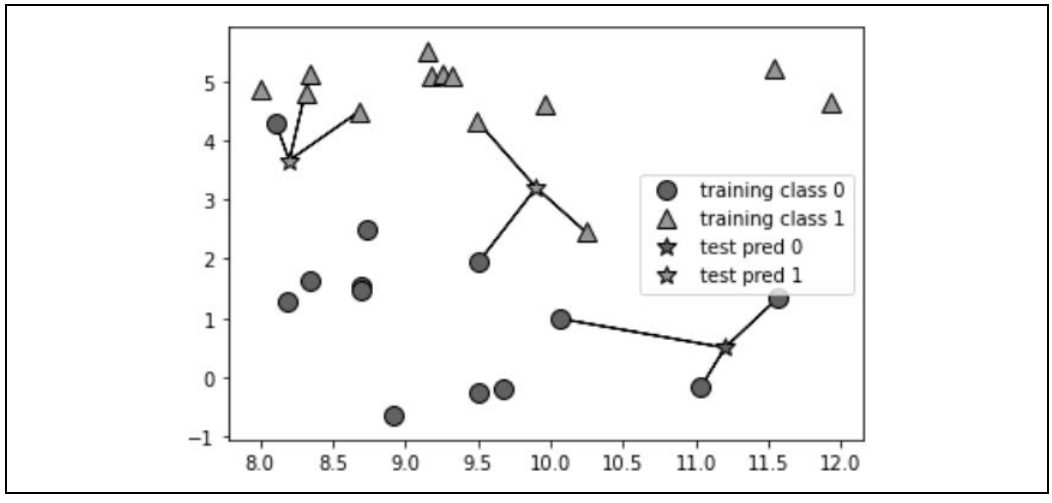
Na tym wykresie dodaliśmy trzy nowe punkty danych, zilustrowane jako gwiazdy. Dla każdego z nich zaznaczyliśmy najbliższy punkt z zestawu uczącego. Wynik prognozowania algorytmu jednego najbliższego sąsiada to etykieta tego punktu (na co wskazuje kolor gwiazdki).

Zamiast uwzględniać tylko najbliższego sąsiada, możemy wziąć pod uwagę dowolną liczbę sąsiadów, *k*. Stąd bierze się nazwa algorytmu *k*-najbliższych sąsiadów. Gdy uwzględnimy więcej niż jednego sąsiada, to aby przypisać etykietę, używamy mechanizmu *głosowania*. Oznacza to, że dla każdego punktu testowego liczymy, ilu sąsiadów należy do klasy 0, a ilu do klasy 1. Następnie przypisujemy

klasę, która występuje częściej: innymi słowy, klasę, która jest w większości wśród *k*-najbliższych sąsiadów. Zastosowanie algorytmu trzech najbliższych sąsiadów przedstawiono na rysunku 2.5:

In[11]:

```
mglearn.plots.plot_knn_classification(n_neighbors=3)
```



Rysunek 2.5. Prognozy stworzone przez model trzech najbliższych sąsiadów na zestawie danych forge

Również w tym przypadku wynik prognozy jest zaznaczony kolorem gwiazdki. Widać, że prognoza dla nowego punktu danych w lewym górnym rogu nie jest taka sama jak ta, gdy używaliśmy tylko jednego sąsiada.

Na tej ilustracji przedstawiono problem z klasyfikacją binarną, ale tę metodę można zastosować do zestawów danych z dowolną liczbą klas. W przypadku większej liczby klas liczymy, ilu sąsiadów należy do każdej klasy, i ponownie prognozujemy najczęściej występującą klasę.

Teraz spójrzmy, jak możemy zastosować algorytm *k*-najbliższych sąsiadów za pomocą biblioteki `scikit-learn`. Aby móc ocenić wydajność uogólniania, najpierw dzielimy nasze dane na zestaw uczący i testowy, jak omówiono w rozdziale 1.:

In[12]:

```
from sklearn.model_selection import train_test_split
X, y = mglearn.datasets.make_forge()
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

Następnie importujemy i tworzymy klasę. Jest to moment, w którym możemy ustawić parametry, takie jak liczba sąsiadów do użycia. Tutaj ustawiamy ją na 3:

In[13]:

```
from sklearn.neighbors import KNeighborsClassifier
clf = KNeighborsClassifier(n_neighbors=3)
```

Teraz dopasowujemy klasyfikator przy użyciu zestawu. W przypadku klasy `KNeighborsClassifier` oznacza to przechowywanie zestawu danych, dzięki czemu możemy do obliczeń podczas prognozowania użyć sąsiadów:

In[14]:

```
clf.fit(X_train, y_train)
```

Aby tworzyć prognozy na danych testowych, wywołujemy metodę `predict`. Dla każdego punktu danych w zestawie testowym oblicza się jego najbliższych sąsiadów w zestawie uczącym i spośród nich typuje najbardziej powszechną klasę:

In[15]:

```
print("Prognozy na zestawie testowym: {}".format(clf.predict(X_test)))
```

Out[15]:

```
Prognozy na zestawie testowym: [1 0 1 0 1 0 0]
```

Aby ocenić, jak dobrze nasz model uogólnia, możemy wywołać metodę `score` z danymi testowymi wraz z etykietami testowymi:

In[16]:

```
print("Dokładność w zestawie testowym: {:.2f}".format(clf.score(X_test, y_test)))
```

Out[16]:

```
Dokładność w zestawie testowym: 0.86
```

Widzimy, że nasz model jest dokładny mniej więcej w 86%, co oznacza, że model prawidłowo prognozował klasę dla 86% próbek w testowym zestawie danych.

Analiza KNeighborsClassifier

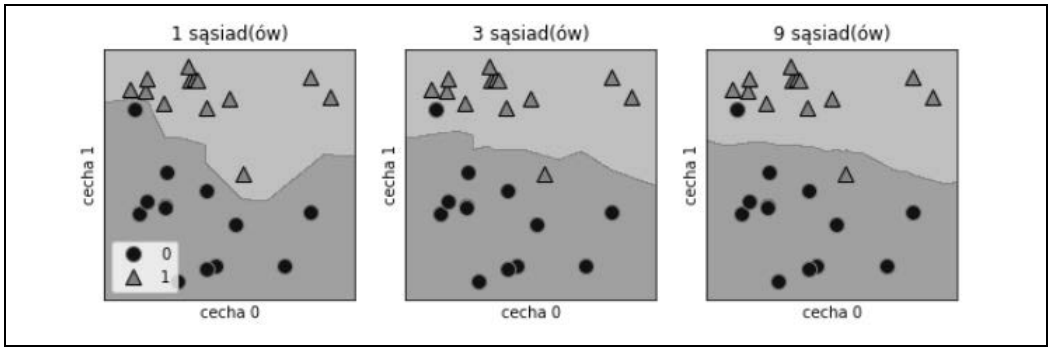
W przypadku dwuwymiarowych zestawów danych możemy również zilustrować prognozowanie dla wszystkich możliwych punktów testowych na płaszczyźnie xy. Płaszczyznę kolorujemy zgodnie z klasą, która byłaby przypisana do punktu w tym regionie. To pozwala nam zobaczyć *granicę decyzyjną*, oddzielającą miejsce, w którym algorytm przypisuje klasę 0, od tego, gdzie przypisuje klasę 1.

Uruchomienie poniższego kodu utworzy wizualizacje granic decyzyjnych dla jednego, trzech i dziewięciu sąsiadów, co przedstawiono na rysunku 2.6:

In[17]:

```
fig, axes = plt.subplots(1, 3, figsize=(10, 3))

for n_neighbors, ax in zip([1, 3, 9], axes):
    # metoda fit zwraca obiekt self, więc możemy utworzyć instancję
    # i dopasować ją do jednej linii
    clf = KNeighborsClassifier(n_neighbors=n_neighbors).fit(X, y)
    mglearn.plots.plot_2d_separator(clf, X, fill=True, eps=0.5, ax=ax, alpha=.4)
    mglearn.discrete_scatter(X[:, 0], X[:, 1], y, ax=ax)
    ax.set_title("{} sąsiad(ów)".format(n_neighbors))
    ax.set_xlabel("cecha 0")
    ax.set_ylabel("cecha 1")
axes[0].legend(loc=3)
```



Rysunek 2.6. Granice decyzyjne utworzone przez model najbliższych sąsiadów dla różnych wartości n_neighbors

Jak widać po lewej stronie rysunku, użycie jednego sąsiada skutkuje utworzeniem granicy decyzyjnej, która jest ściśle zgodna z danymi uczącymi. Uwzględnianie coraz większej liczby sąsiadów prowadzi do gładszej granicy decyzyjnej. Ta zaś odpowiada prostszemu modelowi. Innymi słowy, użycie kilku sąsiadów prowadzi do dużej złożoności modelu (jak pokazano po prawej stronie rysunku 2.1), a użycie wielu sąsiadów skutkuje małą złożonością modelu (jak pokazano po lewej stronie rysunku 2.1). Jeśliby wziąć pod uwagę skrajny przypadek, w którym liczba sąsiadów jest liczbą wszystkich punktów danych w zestawie uczącym, każdy punkt testowy miałby dokładnie tych samych sąsiadów (wszystkie punkty uczące) i wszystkie prognozy byłyby takie same: klasa, która występuje najczęściej w zestawie uczącym.

Zbadajmy, czy możemy potwierdzić związek między złożonością modelu a uogólnieniem, o którym mówiliśmy wcześniej. Zrobimy to na rzeczywistym zestawie danych Breast Cancer. Rozpoczynamy od podzielenia zestawu danych na zestawy uczący i testowy. Następnie oceniamy jakość uczenia i testujemy wydajność z różną liczbą sąsiadów. Wyniki przedstawiono na rysunku 2.7:

In[18]:

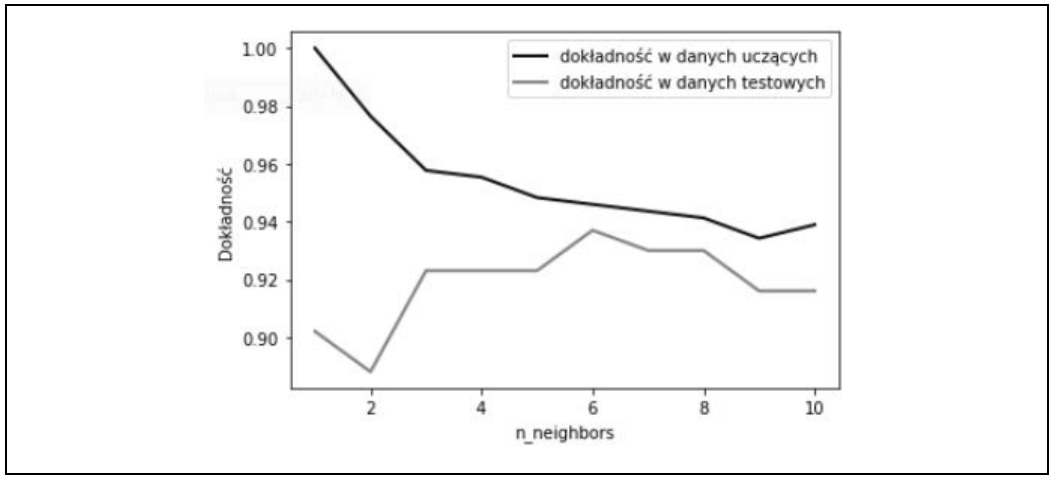
```
from sklearn.datasets import load_breast_cancer

cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=66)

training_accuracy = []
test_accuracy = []
# spróbuj n_neighbors od 1 do 10
neighbors_settings = range(1, 11)

for n_neighbors in neighbors_settings:
    # zbuduj model
    clf = KNeighborsClassifier(n_neighbors=n_neighbors)
    clf.fit(X_train, y_train)
    # zapisz dokładność zestawu uczącego
    training_accuracy.append(clf.score(X_train, y_train))
    # zapisz dokładność uogólniania
    test_accuracy.append(clf.score(X_test, y_test))

plt.plot(neighbors_settings, training_accuracy, label="dokładność w danych uczących")
plt.plot(neighbors_settings, test_accuracy, label="dokładność w danych testowych")
plt.ylabel("Dokładność")
plt.xlabel("n_neighbors")
plt.legend()
```



Rysunek 2.7. Porównanie dokładności uczenia i testu jako funkcji n_neighbors

Wykres przedstawia dokładność zestawu uczącego i testowego na osi *y* w porównaniu z ustawieniem `n_neighbors` na osi *x*. Chociaż wykresy w świecie rzeczywistym rzadko są bardzo gładkie, nadal możemy rozpoznać niektóre cechy nadmiernego i niedostatecznego dopasowania (zwróć uwagę, że ponieważ wzięcie pod uwagę mniejszej liczby sąsiadów powoduje wygenerowanie bardziej złożonego modelu, to wykres jest odwrócony poziomo w stosunku do ilustracji na rysunku 2.1). Prognoza na zestawie uczącym jest idealna, jeśli weźmiemy pod uwagę jednego najbliższego sąsiada. Ale gdy weźmiemy pod uwagę więcej sąsiadów, model staje się prostszy, a dokładność uczenia spada. Dokładność zestawu testowego przy korzystaniu z jednego sąsiada jest niższa niż w przypadku większej liczby sąsiadów, co wskazuje, że użycie jednego najbliższego sąsiada prowadzi do modelu, który jest zbyt złożony. Z drugiej strony w razie użycia 10 sąsiadów model jest zbyt prosty, a jego wydajność — jeszcze gorsza. Najlepsza wydajność jest gdzieś pośrodku, przy, powiedzmy, sześciu sąsiadach. Mimo wszystko warto pamiętać o skali wykresu. Najgorsza wydajność występuje przy mniej więcej 88-procentowej dokładności, co wciąż może być akceptowalne.

Regresja k-sąsiadów

Istnieje również wariant regresji algorytmu *k*-najbliższych sąsiadów. Ponownie zacznijmy od jednego najbliższego sąsiada, tym razem z wykorzystaniem zestawu danych `wave`. Dodaliśmy na osi *x* trzy punkty danych testowych w postaci zielonych gwiazd. Wynik prognozy przy użyciu pojedynczego sąsiada jest po prostu wartością dla najbliższego sąsiada. Na rysunku 2.8 pokazano je jako niebieskie gwiazdy:

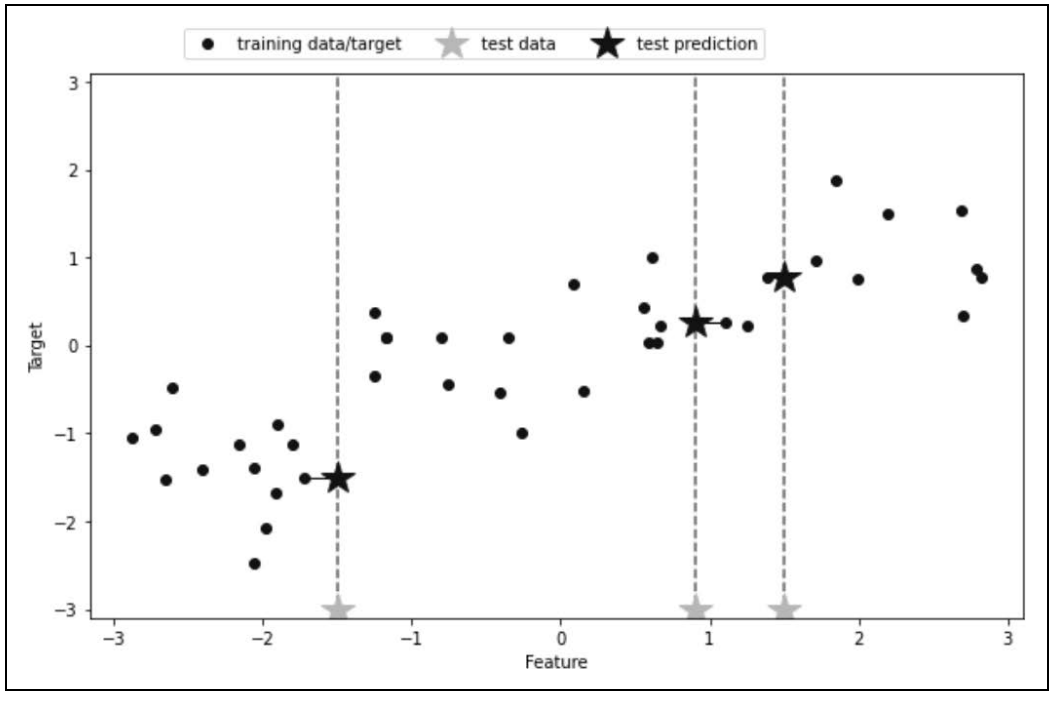
`In[19]:`

```
mglearn.plots.plot_knn_regression(n_neighbors=1)
```

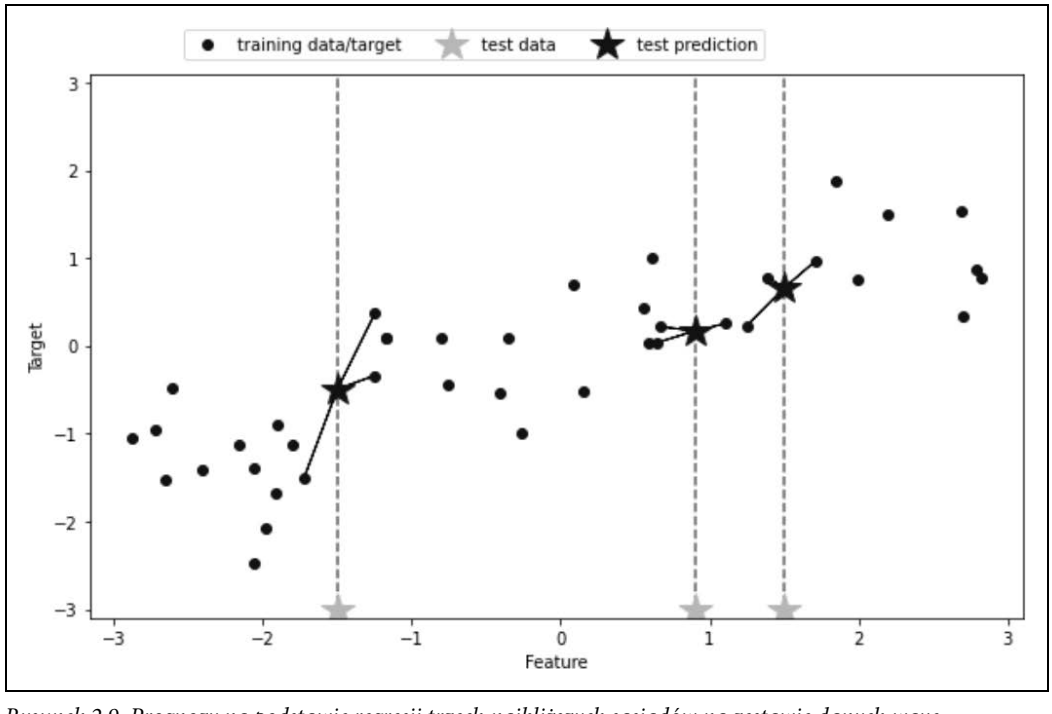
Tak jak wcześniej, do regresji możemy użyć więcej niż jednego najbliższego sąsiada. W przypadku korzystania z wielu najbliższych sąsiadów wynik prognozy to średnia wartości odpowiednich sąsiadów, jak pokazano na rysunku 2.9:

`In[20]:`

```
mglearn.plots.plot_knn_regression(n_neighbors=3)
```



Rysunek 2.8. Prognozy na podstawie regresji jednego najbliższego sąsiada na zestawie danych wave



Rysunek 2.9. Prognozy na podstawie regresji trzech najbliższych sąsiadów na zestawie danych wave

Algorytm regresji *k*-najbliższych sąsiadów jest zaimplementowany w klasie `KNeighborsRegressor` w `scikit-learn`. Używa się jej podobnie jak klasy `KNeighborsClassifier`:

In[21]:

```
from sklearn.neighbors import KNeighborsRegressor
X, y = mglearn.datasets.make_wave(n_samples=40)
# podziel zestaw danych wave na zestaw uczący i zestaw testowy
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
# utwórz instancję modelu i ustaw do rozważenia 3 sąsiadów
reg = KNeighborsRegressor(n_neighbors=3)
# dopasuj model przy użyciu danych uczących i celów uczących
reg.fit(X_train, y_train)
```

Teraz możemy wykonać prognozę na zestawie testowym:

In[22]:

```
print("Prognozy dotyczące zestawu testowego:\n{}".format(reg.predict(X_test)))
```

Out[22]:

```
Prognozy dotyczące zestawu testowego:
[-0.054 0.357 1.137 -1.894 -1.139 -1.631 0.357 0.912 -0.447 -1.139]
```

Możemy również ocenić model metodą punktacji, która dla regresorów zwraca wynik R^2. Wynik R^2, znany również jako współczynnik determinacji, jest miarą jakości prognozy dla modelu regresji i daje wynik od 0 do 1. Wartość 1 odpowiada doskonałej prognozie, a 0 — modelowi stałemu, który tylko prognozuje średnią odpowiedzi zestawu uczącego, `y_train`:

In[23]:

```
print("Zestaw testowy R^2: {:.2f}".format(reg.score(X_test, y_test)))
```

Out[23]:

```
Zestaw testowy R^2: 0.83
```

W tym przypadku wynik wynosi 0,83, co wskazuje na stosunkowo dobre dopasowanie modelu.

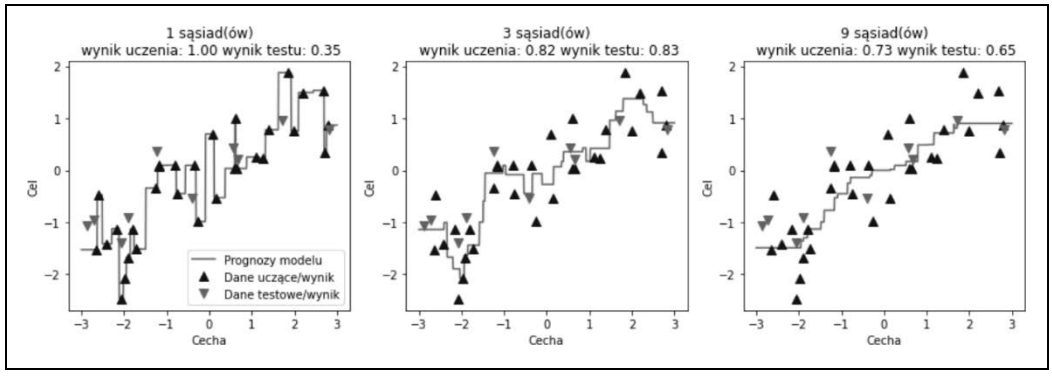
Analiza KNeighborsRegressor

W przypadku naszego jednowymiarowego zestawu danych możemy zobaczyć, jak wyglądają prognozy dla wszystkich możliwych wartości cech, co zilustrowano na rysunku 2.10. Aby to zrobić, tworzymy testowy zestaw danych, który składa się z wielu punktów na prostej:

In[24]:

```
fig, axes = plt.subplots(1, 3, figsize=(15, 4))
# stwórz 1000 punktów danych, równo rozmieszczonych pomiędzy –3 i 3
line = np.linspace(-3, 3, 1000).reshape(-1, 1)
for n_neighbors, ax in zip([1, 3, 9], axes):
    # prognozuj z użyciem 1 sąsiada, 3 sąsiadów lub 9 sąsiadów
    reg = KNeighborsRegressor(n_neighbors=n_neighbors)
    reg.fit(X_train, y_train)
    ax.plot(line, reg.predict(line))
    ax.plot(X_train, y_train, '^', c=mglearn.cm2(0), markersize=8)
    ax.plot(X_test, y_test, 'v', c=mglearn.cm2(1), markersize=8)

    ax.set_title(
        "{} sąsiad(ów)\n wynik uczenia: {:.2f} wynik testu: {:.2f}".format(
            n_neighbors, reg.score(X_train, y_train),
            reg.score(X_test, y_test)))
    ax.set_xlabel("Cecha")
    ax.set_ylabel("Cel")
axes[0].legend(["Prognozy modelu", "Dane uczące/wynik",
    "Dane testowe/wynik"], loc="best")
```



Rysunek 2.10. Porównanie prognoz regresji najbliższych sąsiadów dla różnych wartości n_neighbors

Jak widać na wykresie, gdy używamy tylko jednego sąsiada, każdy punkt w zestawie ma oczywisty wpływ na prognozę, a prognozy wartości przechodzą przez wszystkie punkty danych. Prowadzi to do bardzo niepewnych prognoz. Uwzględnienie większej liczby sąsiadów prowadzi do wykresów prognoz o łagodniejszym przebiegu, ale te również nie pasują do danych uczących.

Mocne i słabe strony i parametry

W zasadzie istnieją dwa ważne parametry klasy klasyfikatora `KNeighbors`: liczba sąsiadów i sposób pomiaru odległości między punktami danych. W praktyce używanie niewielkiej liczby sąsiadów, np. trzech lub pięciu, zazwyczaj działa dobrze, ale z pewnością należy dostosować ten parametr. Wybór właściwej wartości wykracza nieco poza zakres tej książki. Domyślnie używana jest odległość euklidesowa, która sprawdza się w wielu ustawieniach.

Do mocnych stron algorytmu *k*-NN należy to, że model jest bardzo łatwy do zrozumienia i często zapewnia rozsądną wydajność bez konieczności wprowadzania zbyt wielu korekt. Korzystanie z tego algorytmu jest dobrą metodą wstępnego wypróbowania przed rozważeniem bardziej zaawansowanych technik. Budowanie modelu najbliższych sąsiadów zwykle jest szybkie, ale gdy zestaw uczący jest bardzo duży (pod względem liczby funkcji lub próbek), prognozowanie może być powolne. Przy korzystaniu z algorytmu *k*-NN ważne jest, aby wstępnie przetworzyć dane (jak opisano w rozdziale 3.). To podejście zwykle nie działa dobrze w przypadku zestawów danych z wieloma cechami (setką lub więcej), a szczególnie źle działa przy zestawach danych, w których większość cech ma wartość 0 przez większość czasu (tzw. **rzadkie zestawy danych**).

Tak więc, chociaż algorytm najbliższych sąsiadów jest łatwy do zrozumienia, nie jest często używany w praktyce ze względu na powolne prognozowanie i niezdolność do obsługi wielu funkcji. Metoda, którą omówimy dalej, nie ma żadnej z tych wad.

Modele liniowe

Modele liniowe — klasa modeli szeroko stosowanych w praktyce i wnikliwie badanych w ciągu ostatnich kilku dekad — są stosowane od ponad 100 lat. Służą do prognozowania cech wejściowych przy użyciu **funkcji liniowej**, co wkrótce wyjaśnimy.

Modele liniowe do regresji

W przypadku regresji ogólny wzór prognozowania dla modelu liniowego wygląda następująco:

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + \ldots + w[p] * x[p] + b$$

gdzie *x[0]*, …, *x[p]* oznacza cechy (w tym przykładzie liczba cech to *p*) pojedynczego punktu danych, *w* i *b* są parametrami uczonego modelu, a $\hat{y}$ oznacza prognozę stworzoną przez model. Dla zestawu danych z pojedynczą cechą wzór przybierze taką postać:

$$\hat{y} = w[0] * x[0] + b$$

Z lekcji matematyki w szkole średniej możesz pamiętać to jako równanie prostej. W powyższym równaniu *w[0]* reprezentuje nachylenie, a *b* jest przesunięciem względem osi *y*. W przypadku większej liczby elementów parametr *w* zawiera nachylenia dla osi wszystkich elementów. Alternatywnie możesz myśleć o prognozowanej odpowiedzi jako o ważonej sumie cech wejściowych, z wagami (które mogą być ujemne) opisanymi przez parametry *w*.

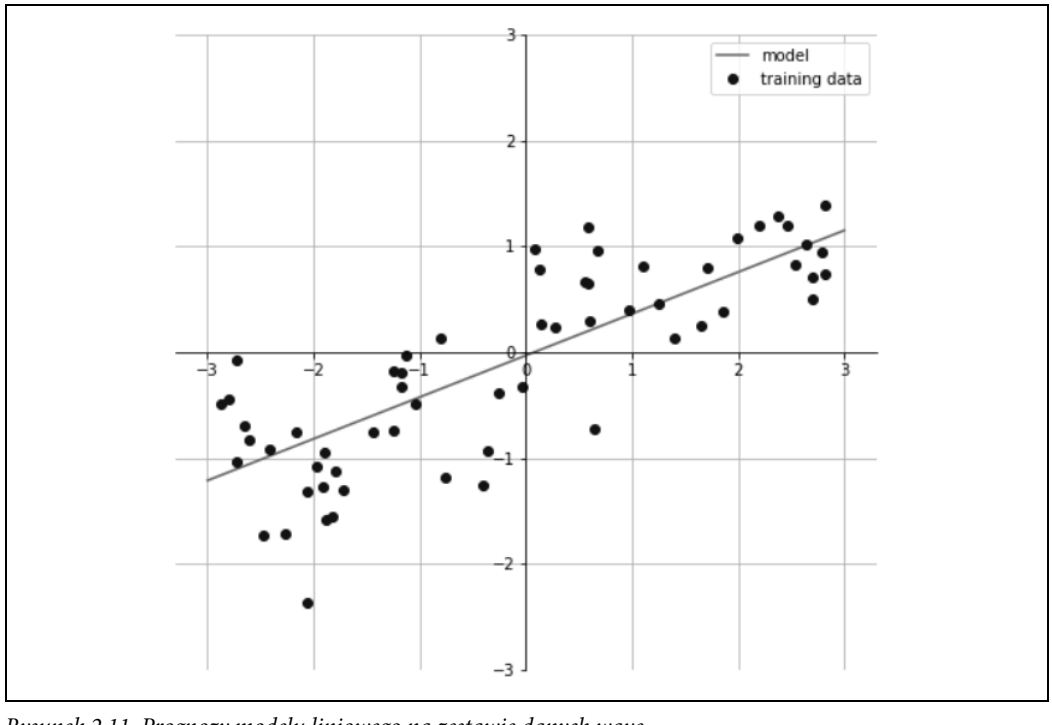
Próba poznania parametrów *w[0]* i *b* w naszym jednowymiarowym zestawie danych `wave` może spowodować utworzenie wykresu podobnego do tego, który przedstawiono na rysunku 2.11:

`In[25]:`

```
mglearn.plots.plot_linear_regression_wave()
```

`Out[25]:`

```
w[0]: 0.393906 b: -0.031804
```



Rysunek 2.11. Prognozy modelu liniowego na zestawie danych wave

Dodaliśmy do wykresu oś współrzędnych, aby ułatwić zrozumienie przebiegu funkcji. Patrząc na `w[0]`, widzimy, że nachylenie powinno wynosić około 0,4, co możemy potwierdzić na wykresie. Punkt przecięcia to miejsce, w którym wykres prognozy powinien przecinać oś *y*: jest to nieco poniżej 0, co można również sprawdzić na wykresie.

Liniowe modele regresji można scharakteryzować jako modele regresji, w których wykres prognozy dla pojedynczej cechy ma postać prostej, płaszczyzny w przypadku używania dwóch cech lub hiperpłaszczyzny w wyższych wymiarach (tzn. przy użyciu większej liczby cech).

Jeśli porównasz prognozy wykonane przy użyciu linii prostej z prognozami dokonanymi przy użyciu klasy `KNeighborsRegressor`, którą przedstawiono na rysunku 2.10, to użycie prostej do prognozowania wydaje się bardzo ograniczające. Wygląda na to, że wszystkie drobne szczegóły danych zostały utracone. W pewnym sensie to prawda. Jest to mocne (i nieco nierealistyczne) założenie, że nasz cel *y* jest liniową kombinacją cech. Jednak spojrzenie na dane jednowymiarowe daje nieco wypaczoną perspektywę. W przypadku zestawów danych z wieloma cechami modele liniowe mogą być bardzo potężnym narzędziem. Szczególnie jeśli masz więcej funkcji niż punktów danych uczących, każdy cel *y* można idealnie modelować (na zestawie uczącym) jako funkcję liniową[6].

Istnieje wiele różnych modeli liniowych regresji. Różnica polega na tym, w jaki sposób parametry modelu *w* i *b* są uczone z danych uczących oraz w jaki sposób można kontrolować złożoność modelu. Przyjrzymy się teraz najpopularniejszym modelom liniowym regresji.

Regresja liniowa (inaczej zwykła metoda najmniejszych kwadratów)

Regresja liniowa lub *metoda najmniejszych kwadratów* (OLS — ang. *ordinary least squares*) jest najprostszą i najbardziej klasyczną metodą regresji liniowej. Regresja liniowa znajduje parametry *w* i *b*, które minimalizują średni kwadratowy błąd między prognozami a rzeczywistymi celami regresji *y* w zestawie uczącym. Średni kwadrat błędu to suma kwadratów różnic między prognozami a wartościami rzeczywistymi. Regresja liniowa nie ma parametrów, co jest zaletą, ale nie ma też możliwości kontrolowania złożoności modelu. Oto kod powodujący utworzenie modelu, który można zobaczyć na rysunku 2.11:

`In[26]:`

```
from sklearn.linear_model import LinearRegression
X, y = mglearn.datasets.make_wave(n_samples=60)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
lr = LinearRegression().fit(X_train, y_train)
```

Parametry „nachylenia” (*w*), zwane również wagami lub *współczynnikami*, są przechowywane w atrybucie `coef_`, podczas gdy przesunięcie lub punkt przecięcia z osią (*b*) — w atrybucie `intercept_`:

`In[27]:`

```
print("lr.coef_: {}".format(lr.coef_))
print("lr.intercept_: {}".format(lr.intercept_))
```

`Out[27]:`

```
lr.coef_: [ 0.394]
lr.intercept_: -0.031804343026759746
```

[6] Łatwo to dostrzec, jeśli znasz algebrę liniową.

Możesz zauważyć dziwnie wyglądający znak podkreślenia na końcu nazw atrybutów `coef_` i `intercept_`. W bibliotece `scikit-learn` wszystko, co pochodzi z danych uczących, przechowuje się w atrybutach, które kończą się znakiem podkreślenia. Stosuje się go po to, żeby odróżnić je od parametrów ustawianych przez użytkownika.

Atrybut `intercept_` zawsze ma wartość pojedynczej liczby zmiennoprzecinkowej, natomiast wartością atrybutu `coef_` jest tablica NumPy z jednym wpisem na każdą funkcję wejściową. Ponieważ w zestawie danych `wave` mamy tylko jedną cechę wejściową, w tablicy `lr.coef_` znajduje się tylko jeden wpis.

Przyjrzyjmy się wydajności zestawów uczącego i testowego:

`In[28]:`

```
print("Wynik zestawu uczącego: {:.2f}".format(lr.score(X_train, y_train)))
print("Wynik zestawu testowego: {:.2f}".format(lr.score(X_test, y_test)))
```

`Out[28]:`

```
Wynik dla zestawu uczącego: 0.67
Wynik dla zestawu testowego: 0.66
```

R^2 wynoszące około 0,66 to nie jest zbyt dobry wynik, ale widzimy, że w przypadku zestawu uczącego i testowego wyniki wydajności są zbliżone. Oznacza to, że prawdopodobnie model jest niedopasowany, a nie nadmiernie dopasowany. Ponieważ model jest bardzo prosty (inaczej: ograniczony), w przypadku tego jednowymiarowego zestawu danych istnieje niewielkie ryzyko nadmiernego dopasowania. Jednak przy zestawach danych o wyższych wymiarach (co oznacza zestawy danych z wieloma cechami) modele liniowe stają się potężniejsze i istnieje większe prawdopodobieństwo ich nadmiernego dopasowania. Przyjrzyjmy się, jak klasa `LinearRegression` działa na bardziej złożonym zestawie danych, takim jak zestaw danych Boston Housing. Pamiętaj, że ten zestaw danych zawiera 506 próbek i 105 pochodnych funkcji. Najpierw ładujemy zestaw danych i dzielimy go na zestaw uczący i testowy. Następnie, tak jak poprzednio, budujemy model regresji liniowej:

`In[29]:`

```
X, y = mglearn.datasets.load_extended_boston()
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
lr = LinearRegression().fit(X_train, y_train)
```

Kiedy porównamy wyniki zestawu uczącego i testowego, stwierdzimy, że prognozy w przypadku zestawu uczącego są bardzo dokładne, ale współczynnik R^2 w zestawie testowym ma znacznie gorszą wartość:

`In[30]:`

```
print("Wynik dla zestawu uczącego: {:.2f}".format(lr.score(X_train, y_train)))
print("Wynik dla zestawu testowego: {:.2f}".format(lr.score(X_test, y_test)))
```

`Out[30]:`

```
Wynik dla zestawu uczącego: 0.95
Wynik dla zestawu testowego: 0.61
```

Ta rozbieżność między wynikami na zestawie uczącym i testowym wskazuje wyraźnie, że doszło do nadmiernego dopasowania, dlatego powinniśmy spróbować znaleźć model, który pozwoli nam kontrolować złożoność. Jedną z najczęściej stosowanych alternatyw dla standardowej regresji liniowej jest regresja grzbietowa, którą opisujemy dalej.

Regresja grzbietowa

Regresja grzbietowa też jest liniowym modelem regresji, więc wzór używany do prognozowania jest taki sam jak w przypadku zwykłej metody najmniejszych kwadratów. Jednak w regresji grzbietowej współczynniki (*w*) dobiera się nie tylko tak, aby uzyskiwać dobre prognozy na danych uczących, ale też aby przyjmować dodatkowe ograniczenia. Chcemy również, aby wielkość współczynników była jak najmniejsza; innymi słowy, wszystkie wartości współczynników *w* powinny być bliskie 0. Intuicyjnie oznacza to, że każda cecha powinna mieć jak najmniejszy wpływ na wynik (co przekłada się na małe nachylenie), ale bez pogorszenia jakości prognoz. Takie ograniczenie jest przykładem *regularyzacji*. Regularyzacja oznacza wyraźne ograniczenie modelu w taki sposób, aby uniknąć nadmiernego dopasowania. Jej szczególny rodzaj, który jest używany w regresji grzbietowej, nazywamy regularyzacją L2[7].

Regresja grzbietowa jest zaimplementowana w klasie `linear_model.Ridge`. Zobaczmy, jak dobrze działa na rozszerzonym zestawie danych Boston Housing:

`In[31]:`

```
from sklearn.linear_model import Ridge
ridge = Ridge().fit(X_train, y_train)
print("Wynik dla zestawu uczącego: {:.2f}".format(ridge.score(X_train, y_train)))
print("Wynik dla zestawu testowego: {:.2f}".format(ridge.score(X_test, y_test)))
```

`Out[31]:`

```
Wynik dla zestawu uczącego: 0.89
Wynik dla zestawu testowego: 0.75
```

Jak widać, wynik dla zestawu uczącego w przypadku użycia modelu `Ridge` jest *niższy* niż przy `LinearRegression`, natomiast wynik dla zestawu testowego jest wyższy. Jest to zgodne z naszymi oczekiwaniami. W przypadku regresji liniowej nadmiernie dopasowaliśmy model do danych. Model `Ridge` jest bardziej ograniczony, więc jest mniej prawdopodobne, że nastąpi nadmierne dopasowanie. Mniej złożony model oznacza gorszą wydajność w zestawie uczącym, ale lepsze uogólnianie. Ponieważ interesuje nas tylko wykonanie uogólnień, powinniśmy wybrać model `Ridge` zamiast `LinearRegression`.

`Ridge` jest kompromisem między prostotą modelu (współczynniki bliskie 0) a jego wydajnością w zestawie uczącym. Jak duży nacisk w modelu ma być położony na prostotę w porównaniu z wydajnością w zestawie uczącym, może zostać określone przez użytkownika za pomocą parametru `alfa`. W poprzednim przykładzie użyliśmy domyślnego parametru `alpha=1.0`. Nie da nam to jednak najlepszego kompromisu. Optymalne ustawienie parametru `alfa` zależy od konkretnego zestawu danych, którego używamy. Zwiększenie wartości parametru `alfa` wymusza konieczność większego zbliżenia wartości współczynników do 0, co zmniejsza wydajność zestawu uczącego, ale może pomóc uogólniać, np.:

`In[32]:`

```
ridge10 = Ridge(alpha=10).fit(X_train, y_train)
print("Wynik dla zestawu uczącego: {:.2f}".format(ridge10.score(X_train, y_train)))
print("Wynik dla zestawu testowego: {:.2f}".format(ridge10.score(X_test, y_test)))
```

`Out[32]:`

```
Wynik dla zestawu uczącego: 0.79
Wynik dla zestawu testowego: 0.64
```

[7] Model Ridge wprowadza funkcję kary do normy współczynników L2, inaczej: odległości euklidesowej *w*.

Zmniejszenie współczynnika `alfa` pozwala na mniejsze ograniczenie współczynników, co oznacza, że posuwamy się w prawo na wykresie, który przedstawiono na rysunku 2.1. W przypadku bardzo małych wartości `alfa` współczynniki są ledwo ograniczone i otrzymujemy model, który przypomina regresję liniową:

In[33]:

```
ridge01 = Ridge(alpha=0.1).fit(X_train, y_train)
print("Wynik dla zestawu uczącego: {:.2f}".format(ridge01.score(X_train, y_train)))
print("Wynik dla zestawu testowego: {:.2f}".format(ridge01.score(X_test, y_test)))
```

Out[33]:

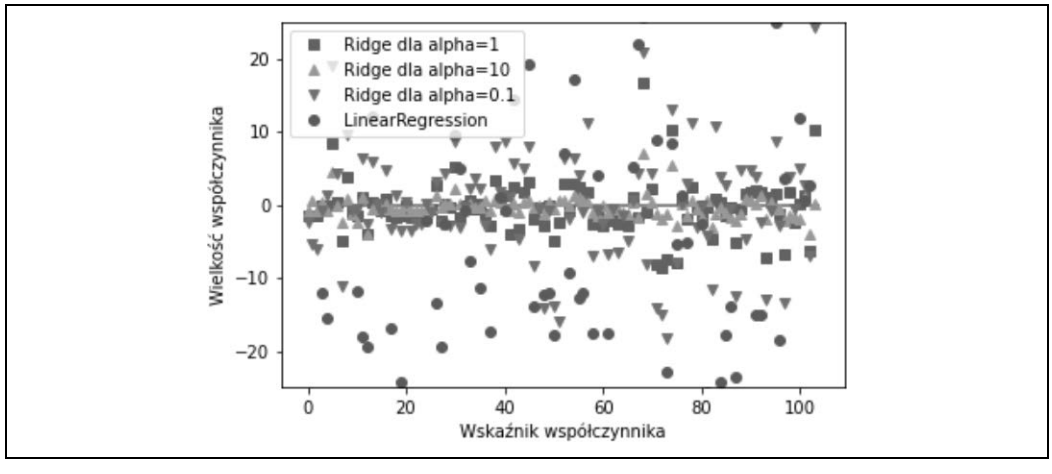
```
Wynik dla zestawu uczącego: 0.93
Wynik dla zestawu testowego: 0.77
```

W tym przypadku współczynnik `alfa=0,1` wydaje się działać dobrze. Moglibyśmy spróbować jeszcze bardziej zmniejszyć jego wartość, aby poprawić uogólnianie. Na razie skupmy się na tym, jak parametr `alfa` wpływa na złożoność modelu, co pokazano na rysunku 2.1. Metody prawidłowego doboru parametrów omówimy w rozdziale 5.

Bardziej wartościowy wgląd w to, jak parametr `alfa` zmienia model, możemy uzyskać poprzez sprawdzenie atrybutu `coef_attribute` w modelach o różnych wartościach `alfa`. Wyższa wartość tego parametru oznacza bardziej ograniczony model. Spodziewamy się więc, że wartości `coef_attribute` będą niższe dla wysokiej wartości `alfa`, a wyższe dla niskiej wartości tego parametru. Potwierdza to wykres, który umieszczono na rysunku 2.12:

In[34]:

```
plt.plot(ridge.coef_, 's', label="Ridge dla alpha=1")
plt.plot(ridge10.coef_, '^', label="Ridge dla alpha=10")
plt.plot(ridge01.coef_, 'v', label="Ridge dla alpha=0.1")
plt.plot(lr.coef_, 'o', label="LinearRegression")
plt.xlabel("Wskaźnik współczynnika")
plt.ylabel("Wielkość współczynnika")
plt.hlines(0, 0, len(lr.coef_))
plt.ylim(-25, 25)
plt.legend()
```



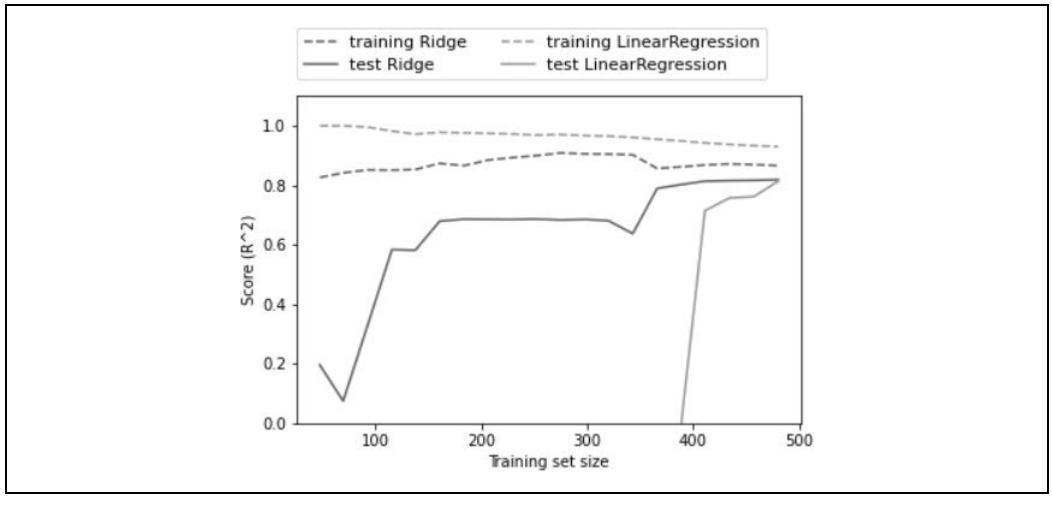
Rysunek 2.12. Porównanie wielkości współczynników dla regresji grzbietowej z różnymi wartościami regresji alfa i liniowej

Na powyższym wykresie oś *x* oznacza wpisy `coef_`, gdzie: `x=0` pokazuje współczynnik powiązany z pierwszą cechą, `x=1` współczynnik powiązany z drugą cechą itd., aż do `x=100`. Oś *y* oznacza wartości liczbowe dla odpowiednich wartości współczynników. Najważniejszym wnioskiem jest to, że dla `alfa=10` współczynniki są przeważnie w granicach od –3 do 3. Wartości współczynników dla modelu `Ridge` z `alfa=1` są nieco wyższe. Wartości współczynników odpowiadające `alfa=0,1` nadal są tak wysokie, a współczynniki w przypadku regresji liniowej bez żadnej regularyzacji (czyli gdyby przyjąć `alfa=0`) mają tak wysokie wartości, że nie mieszczą się na wykresie.

Innym sposobem zrozumienia wpływu regularyzacji jest ustalenie stałej wartości `alfa` i regulacja zasobu dostępnych danych uczących. Na potrzeby rysunku 2.13 wydzieliliśmy część z zestawu danych Boston Housing i dokonaliśmy oceny regresji liniowej i grzbietowej (gdzie `alfa=1`) na podzbiorach o rosnącym rozmiarze (wykresy, na których przedstawiono wydajność modelu w funkcji rozmiaru zestawu danych, nazywamy *krzywymi uczenia się*):

`In[35]:`

```
mglearn.plots.plot_ridge_n_samples()
```



Rysunek 2.13. Krzywe uczenia się dla regresji grzbietowej i regresji liniowej w zestawie danych Boston Housing

Jak można się spodziewać, wynik uczenia jest wyższy niż wynik testu dla wszystkich rozmiarów zestawu danych, zarówno dla regresji grzbietowej, jak i liniowej. Ponieważ model grzbietowy jest uregulowany, wynik jego uczenia jest niższy niż wynik uczenia dla regresji liniowej na całej planszy. Jednak wynik testu dla modelu grzbietowego jest lepszy, szczególnie dla małych podzbiorów danych. W przypadku mniej niż 400 punktów danych regresja liniowa niczego się nie nauczy. W miarę jak coraz więcej danych jest dostępnych dla modelu, oba modele stają się lepsze, a model regresji liniowej na końcu dogania model regresji krawędziowej. Wniosek z tego taki, że przy wystarczającej ilości danych uczących regularyzacja staje się mniej ważna, a przy wystarczającej ilości danych regresje grzbietowa i liniowa będą miały taką samą wydajność (fakt, że dzieje się to tutaj, gdy używa się pełnego zestawu danych, jest po prostu przypadkiem). Innym interesującym aspektem, który przedstawiono na rysunku 2.13, jest spadek wydajności uczenia w przypadku regresji liniowej. Jeśli dodanych zostanie więcej danych, modelowi trudniej będzie nadmiernie dopasować lub zapamiętać dane.

Model Lasso

Alternatywnym dla Ridge modelem do uregulowania regresji liniowej jest Lasso. Podobnie jak w przypadku regresji grzbietowej, użycie go ogranicza współczynniki do bliskich 0, ale w nieco inny sposób, zwany regularyzacją L1[8]. Konsekwencją regularyzacji L1 jest to, że podczas korzystania z modelu Lasso niektóre współczynniki są dokładnie **równe 0**. Oznacza to, że model całkowicie ignoruje niektóre funkcje. Można to postrzegać jako formę automatycznego wyboru cech. Posiadanie niektórych współczynników równych dokładnie 0 często ułatwia interpretację modelu i może ujawnić jego najważniejsze cechy.

Zastosujmy model Lasso do rozszerzonego zestawu danych Boston Housing:

In[36]:

```
from sklearn.linear_model import Lasso
lasso = Lasso().fit(X_train, y_train)
print("Wynik dla zestawu uczącego: {:.2f}".format(lasso.score(X_train, y_train)))
print("Wynik dla zestawu testowego: {:.2f}".format(lasso.score(X_test, y_test)))
print("Liczba użytych cech: {}".format(np.sum(lasso.coef_ != 0)))
```

Out[36]:

```
Wynik dla zestawu uczącego: 0.29
Wynik dla zestawu testowego: 0.21
Liczba użytych cech: 4
```

Jak widać, model Lasso radzi sobie nie najlepiej, zarówno na zestawie uczącym, jak i testowym. Oznacza to, że jest niedopasowany, i okazuje się, że użyto tylko 4 ze 105 funkcji. Podobnie jak w przypadku modelu Ridge, Lasso również ma parametr regularyzacji `alfa`, który kontroluje odchylenie współczynników od 0. W poprzednim przykładzie użyliśmy domyślnej wartości `alpha=1.0`. Aby zmniejszyć niedopasowanie, spróbujmy zmniejszyć parametr `alfa`. W takim przypadku musimy również zwiększyć domyślne ustawienie `max_iter` (maksymalną liczbę iteracji do uruchomienia):

In[37]:

```
# zwiększamy domyślne ustawienie "max_iter",
# w innym przypadku model poinformuje, że należy zwiększyć max_iter
lasso001 = Lasso(alpha=0.01, max_iter=100000).fit(X_train, y_train)
print("Wynik dla zestawu uczącego: {:.2f}".format(lasso001.score(X_train, y_train)))
print("Wynik dla zestawu testowego: {:.2f}".format(lasso001.score(X_test, y_test)))
print("Liczba użytych cech: {}".format(np.sum(lasso001.coef_ != 0)))
```

Out[37]:

```
Wynik dla zestawu uczącego: 0.90
Wynik dla zestawu testowego: 0.77
Liczba użytych cech: 33
```

Niższa wartość parametru `alfa` pozwoliła nam dopasować bardziej złożony model, który działał lepiej na danych testowych. Wydajność jest nieco lepsza niż przy użyciu modelu Ridge i używamy tylko 33 ze 105 funkcji. To sprawia, że model ten jest potencjalnie łatwiejszy do zrozumienia.

Jeśli jednak ustawimy zbyt niską wartość parametru `alfa`, ponownie usuwamy efekt regularyzacji i zostajemy z modelem nadmiernie dopasowanym, z wynikiem podobnym do uzyskanego przy użyciu modelu `LinearRegression`:

[8] Lasso wprowadza funkcję kary do normy L1 wektora współczynników — lub, innymi słowy, sumę bezwzględnych wartości współczynników.

In[38]:

```
lasso00001 = Lasso(alpha=0.0001, max_iter=100000).fit(X_train, y_train)
print("Wynik dla zestawu uczącego: {:.2f}".format(lasso00001.score(X_train, y_train)))
print("Wynik dla zestawu testowego: {:.2f}".format(lasso00001.score(X_test, y_test)))
print("Liczba użytych cech: {}".format(np.sum(lasso00001.coef_ != 0)))
```

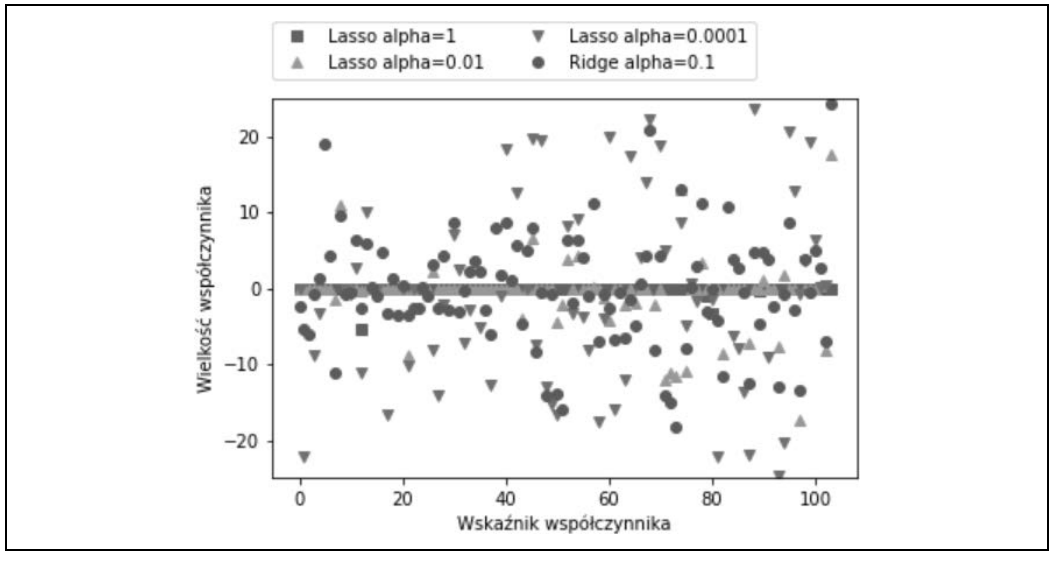
Out[38]:

```
Wynik dla zestawu uczącego: 0.95
Wynik dla zestawu testowego: 0.64
Liczba użytych cech: 94
```

Ponownie możemy określić współczynniki różnych modeli, podobnie jak w przypadku przykładu z rysunku 2.12. Wynik pokazano na rysunku 2.14:

In[39]:

```
plt.plot(lasso.coef_, 's', label="Lasso alpha=1")
plt.plot(lasso001.coef_, '^', label="Lasso alpha=0.01")
plt.plot(lasso00001.coef_, 'v', label="Lasso alpha=0.0001")
plt.plot(ridge01.coef_, 'o', label="Ridge alpha=0.1")
plt.legend(ncol=2, loc=(0, 1.05))
plt.ylim(-25, 25)
plt.xlabel("Wskaźnik współczynnika")
plt.ylabel("Wielkość współczynnika")
```



Rysunek 2.14. Porównanie wielkości współczynników dla regresji Lasso z różnymi wartościami alfa i regresji grzbietowej

W przypadku `alfa=1` widzimy, że większość współczynników ma wartość zerową (co już wiedzieliśmy) oraz że pozostałe współczynniki też są małe. Po zmniejszeniu `alfa` do 0,01 otrzymujemy rozwiązanie, które pokazano jako zielone kropki, co powoduje, że większość cech przyjmuje wartość 0. Ustawiwszy wartość `alfa=0,00001`, otrzymujemy model, który jest dość nieregularny, z większością współczynników różnych od 0, do tego z dużymi wartościami. Dla porównania najlepsze rozwiązanie w przypadku modelu Ridge pokazano w kolorze turkusowym. Model Ridge z wartością `alfa=0,1` ma podobną wydajność predykcyjną jak model Lasso z wartością `alfa=0,01`, ale przy użyciu modelu Ridge wszystkie współczynniki są niezerowe.

W praktyce regresja grzbietowa jest zwykle pierwszym wyborem między tymi dwoma modelami. Jeśli jednak masz dużą liczbę funkcji i oczekujesz, że tylko kilka z nich będzie brana pod uwagę, Lasso może być lepszym wyborem. Podobnie, jeśli wymagany jest model łatwy do zinterpretowania, model Lasso będzie łatwiejszy do zrozumienia, ponieważ wybierze tylko podzbiór funkcji wejściowych. W bibliotece `scikit-learn` udostępniono także klasę `ElasticNet`, która łączy funkcje kary Lasso i Ridge. W praktyce ta kombinacja działa najlepiej, ale wymaga dostosowywania dwóch parametrów: jednego dla regularyzacji L1, a drugiego dla regularyzacji L2.

Modele liniowe do klasyfikacji

Modele liniowe są również powszechnie stosowane do klasyfikacji. Spójrzmy najpierw na klasyfikację binarną. W tym przypadku predykcja jest wykonywana przy użyciu wzoru:

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + ... + w[p] * x[p] + b > 0$$

Wzór wygląda bardzo podobnie do wzoru dla regresji liniowej, ale zamiast po prostu zwracać ważoną sumę cech, ustalamy wartość prognozowaną na 0. Jeśli funkcja jest mniejsza od 0, prognozujemy klasę –1; jeśli jest większa od 0, prognozujemy klasę +1. Ta zasada prognozowania jest wspólna dla wszystkich modeli liniowych do klasyfikacji. Istnieje wiele różnych sposobów na znalezienie współczynników (w) i przecięcia (b).

W przypadku liniowych modeli regresji wynik jest liniową funkcją cech: linią, płaszczyzną lub hiperpłaszczyzną (w wyższych wymiarach). W przypadku modeli liniowych do klasyfikacji *granica decyzyjna* ma postać liniowej funkcji danych wejściowych. Innymi słowy, (binarny) klasyfikator liniowy to klasyfikator, który oddziela dwie klasy za pomocą linii, płaszczyzny lub hiperpłaszczyzny. W tym podrozdziale poznamy na to przykłady.

Istnieje wiele algorytmów uczenia modeli liniowych. Różnią się na dwa sposoby:

- Sposób, w jaki mierzą, jak dobrze dana kombinacja współczynników i punkt przecięcia pasuje do danych uczących.
- Czy i jakiego rodzaju regularyzacji używają.

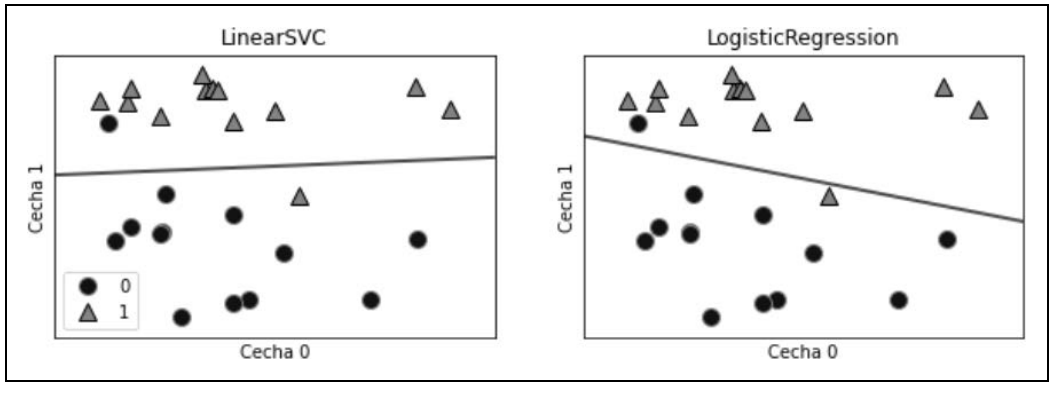
Różne algorytmy na różne sposoby dokonują pomiaru tego, co oznacza „dobrze pasuje do zestawu uczącego". Niestety, z technicznych przyczyn matematycznych nie jest możliwe dostosowanie w i b tak, aby zminimalizować liczbę błędnych klasyfikacji generowanych przez algorytmy. Na potrzeby nasze i wielu innych zastosowań różne wybory dla pierwszej pozycji z powyższej listy (zwane *funkcjami strat*) nie mają większego znaczenia.

Dwa najpopularniejsze algorytmy klasyfikacji liniowej to *regresja logistyczna*, zaimplementowana w klasie `linear_model.LogisticRegression`, oraz *maszyny liniowych wektorów nośnych* (liniowe SVM — ang. *support-vector machines*), zaimplementowane w klasie `svm.LinearSVC` (SVC oznacza klasyfikację wektorów nośnych). Pomimo swojej nazwy model `LogisticRegression` jest algorytmem klasyfikacyjnym, a nie algorytmem regresji, i nie należy go mylić z modelem `LinearRegression`.

Możemy zastosować modele `LogisticRegression` i `LinearSVC` do zestawu danych `forge` i zwizualizować granicę decyzyjną, która została wyznaczona przez modele liniowe, co przedstawiono na rysunku 2.15:

`In[40]:`

```
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC
X, y = mglearn.datasets.make_forge()
fig, axes = plt.subplots(1, 2, figsize=(10, 3))
for model, ax in zip([LinearSVC(), LogisticRegression()], axes):
    clf = model.fit(X, y)
    mglearn.plots.plot_2d_separator(clf, X, fill=False, eps=0.5, ax=ax, alpha=.7)
    mglearn.discrete_scatter(X[:, 0], X[:, 1], y, ax=ax)
    ax.set_title("{}".format(clf.__class__.__name__))
    ax.set_xlabel("Cecha 0")
    ax.set_ylabel("Cecha 1")
axes[0].legend()
```



Rysunek 2.15. Granice decyzyjne liniowej SVM i regresji logistycznej na zestawie danych forge z parametrami domyślnymi

Na powyższym rysunku pierwsza cecha zestawu danych `forge` znajduje się na osi *x*, a druga na osi *y*. Wyświetlamy granice decyzyjne znalezione odpowiednio przez `LinearSVC` i `LogisticRegression` jako linie proste, które oddzielają obszar sklasyfikowany jako klasa 1 na górze od obszaru sklasyfikowanego jako klasa 0 na dole. Innymi słowy, każdy nowy punkt danych, który leży powyżej czarnej linii, zostanie sklasyfikowany przez odpowiedni klasyfikator jako klasa 1, a każdy punkt leżący poniżej czarnej linii zostanie zaklasyfikowany jako klasa 0.

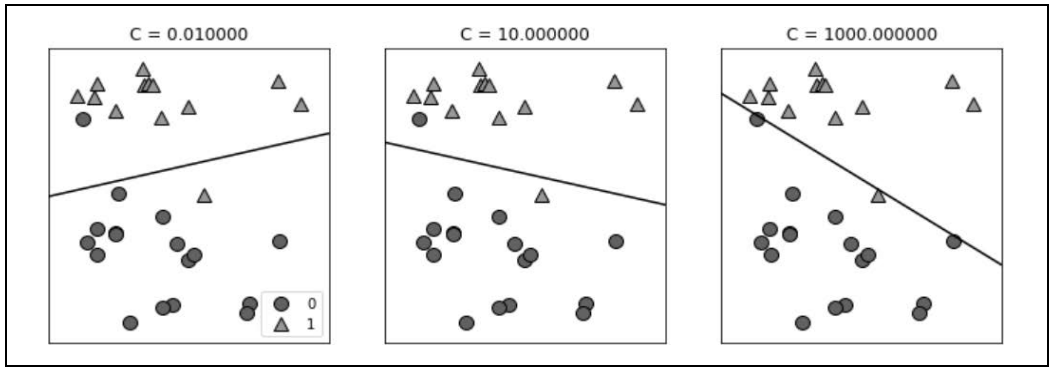
Oba modele mają podobne granice decyzyjne. Zauważ, że oba błędnie klasyfikują dwa punkty. Domyślnie oba modele stosują regularyzację L2, w taki sam sposób jak model `Ridge` dla regresji.

W przypadku `LogisticRegression` i `LinearSVC` parametr kompromisu, który określa siłę regularyzacji, nazywa się `C`, a wyższe wartości `C` odpowiadają *mniejszej* regularyzacji. Innymi słowy, gdy używasz wysokiej wartości parametru `C`, `LogisticRegression` i `LinearSVC`, staraj się jak najlepiej dopasować zestaw uczący, podczas gdy przy niskich wartościach parametru `C` modele kładą większy nacisk na znalezienie wektora współczynników (w), który jest bliski 0.

Jest jeszcze jeden interesujący aspekt działania parametru `C`. Użycie niskich wartości `C` spowoduje, że algorytmy będą próbowały dostosować się do „większości" punktów danych, podczas gdy użycie wyższej wartości `C` podkreśla znaczenie prawidłowego sklasyfikowania każdego pojedynczego punktu danych. Zastosowanie `LinearSVC` zilustrowano na rysunku 2.16:

In[41]:

```
mglearn.plots.plot_linear_svc_regularization()
```



Rysunek 2.16. Granice decyzyjne liniowej maszyny SVM na zestawie danych forge dla różnych wartości C

Po lewej stronie mamy bardzo małą wartość parametru `C` odpowiadającą dużej regularyzacji. Większość punktów klasy 0 znajduje się w górnej części wykresu, a większość punktów klasy 1 w części dolnej. Silnie uregulowany model wybiera relatywnie poziomą linię, co powoduje błędne sklasyfikowanie dwóch punktów. Na środkowym wykresie parametr `C` ma nieco wyższą wartość, a przechylenie granicy decyzyjnej modelu sprawia, że jest bardziej skupiony na dwóch błędnie sklasyfikowanych próbkach. Wreszcie, po prawej stronie, bardzo wysoka wartość parametru `C` w modelu powoduje mocne przechylenie granicy decyzyjnej, a wszystkie punkty klasy 0 są teraz poprawnie sklasyfikowane. Jeden z punktów klasy 1 nadal jest błędnie sklasyfikowany, ponieważ niemożliwe jest poprawne sklasyfikowanie wszystkich punktów w tym zestawie danych za pomocą linii prostej.

Model przedstawiony po prawej stronie najlepiej klasyfikuje wszystkie punkty, ale może nie oddać dobrze ogólnego układu klas. Innymi słowy, ten model jest prawdopodobnie nadmiernie dopasowany. Podobnie jak w przypadku regresji, modele liniowe do klasyfikacji mogą wydawać się bardzo restrykcyjne w przestrzeniach niskowymiarowych i dopuszczać jedynie granice decyzyjne, które są liniami prostymi lub płaszczyznami. Wielowymiarowe modele liniowe do klasyfikacji stają się bardzo potężne, a ochrona przed nadmiernym dopasowaniem staje się coraz ważniejsza przy rozważaniu większej liczby funkcji.

Przeanalizujmy bardziej szczegółowo model `LinearLogistic` w zestawie danych Breast Cancer:

In[42]:

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
cancer.data, cancer.target, stratify=cancer.target, random_state=42)
logreg = LogisticRegression().fit(X_train, y_train)
print("Wynik dla zestawu uczącego: {:.3f}".format(logreg.score(X_train, y_train)))
print("Wynik dla zestawu testowego: {:.3f}".format(logreg.score(X_test, y_test)))
```

Out[42]:

```
Wynik dla zestawu uczącego: 0.953
Wynik dla zestawu testowego: 0.958
```

Domyślna wartość `C=1` zapewnia całkiem dobre wyniki, z 95-procentową dokładnością zarówno w zestawie uczącym, jak i testowym. Ponieważ jednak w przypadku zestawu uczącego i testowego wyniki są zbliżone, prawdopodobnie model jest niedopasowany. Spróbujmy zwiększyć parametr `C`, żeby pasował do bardziej elastycznego modelu:

In[43]:

```
logreg100 = LogisticRegression(C=100).fit(X_train, y_train)
print("Wynik dla zestawu uczącego: {:.3f}".format(logreg100.score(X_train, y_train)))
print("Wynik dla zestawu testowego: {:.3f}".format(logreg100.score(X_test, y_test)))
```

Out[43]:

```
Wynik dla zestawu uczącego: 0.972
Wynik dla zestawu testowego: 0.965
```

Użycie wartości `C=100` skutkuje większą dokładnością zestawu uczącego, a także nieznacznie zwiększoną dokładnością zestawu testowego, co potwierdza naszą tezę, że bardziej złożony model powinien działać lepiej.

Aby zbadać, co się stanie, jeśli użyjemy jeszcze bardziej regularyzowanego modelu niż domyślne `C=1`, możemy ustawić wartość `C=0,01`:

In[44]:

```
logreg001 = LogisticRegression(C=0.01).fit(X_train, y_train)
print("Wynik dla zestawu uczącego: {:.3f}".format(logreg001.score(X_train, y_train)))
print("Wynik dla zestawu testowego: {:.3f}".format(logreg001.score(X_test, y_test)))
```

Out[44]:

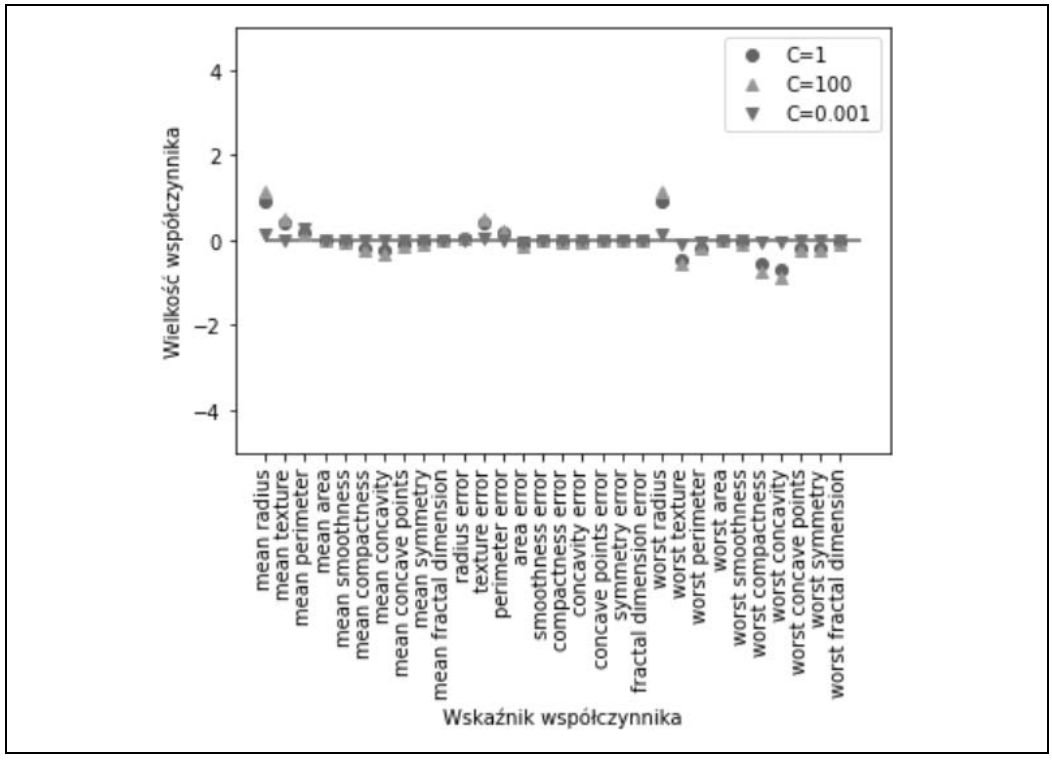
```
Wynik dla zestawu uczącego: 0.934
Wynik dla zestawu testowego: 0.930
```

Zgodnie z oczekiwaniami, gdy wyjdziemy od już niedopasowanego modelu i będziemy się przesuwać bardziej w lewo wzdłuż skali, którą przedstawiono na rysunku 2.1, dokładność spadnie w stosunku do parametrów domyślnych zarówno w przypadku badania na zestawie uczącym, jak i testowym.

Na koniec przyjrzyjmy się współczynnikom wyuczonym przez modele z trzema różnymi ustawieniami parametru regularyzacji `C` (rysunek 2.17):

In[45]:

```
plt.plot(logreg.coef_.T, 'o', label="C=1")
plt.plot(logreg100.coef_.T, '^', label="C=100")
plt.plot(logreg001.coef_.T, 'v', label="C=0.001")
plt.xticks(range(cancer.data.shape[1]), cancer.feature_names, rotation=90)
plt.hlines(0, 0, cancer.data.shape[1])
plt.ylim(-5, 5)
plt.xlabel("Wskaźnik współczynnika")
plt.ylabel("Wielkość współczynnika")
plt.legend()
```



Rysunek 2.17. Współczynniki wyuczone przez regresję logistyczną na zestawie danych Breast Cancer dla różnych wartości parametru C

Ponieważ model `LogisticRegression` domyślnie stosuje regularyzację L2, wynik wygląda podobnie do uzyskanego przez model `Ridge` na rysunku 2.12. Silniejsza regularyzacja zbliża wartości współczynników do 0, chociaż wartości współczynników nigdy nie będą równe 0. Po bliższym przyjrzeniu się wykresowi możemy również dostrzec interesujący efekt w przypadku trzeciego współczynnika w „średnim zakresie". Dla wartości parametru `C=100` i `C=1` współczynnik jest ujemny, natomiast dla `C=0,001` — dodatni, o wielkości nawet większej niż w przypadku `C=1`. Podczas interpretacji takiego modelu można by pomyśleć, że współczynnik wskazuje nam, z którą klasą powiązana jest cecha. Przykładowo można by pomyśleć, że duży zakres, w którym dochodzi do niespójności, wiąże się z tym, że próbka jest „złośliwa". Jednak zmiana znaku wartości współczynnika w „średnim zakresie" oznacza, że w zależności od tego, na który model patrzymy, wysoki „średni zakres" może być uznany za taki, który wskazuje na „łagodność" lub „złośliwość" próbki. Ten przykład pokazuje, że współczynniki modeli liniowych należy zawsze interpretować z przymrużeniem oka.

Jeśli zależy nam na łatwiejszym do zinterpretowania modelu, pomocne może być użycie regularyzacji L1, ponieważ ogranicza ona model do skorzystania tylko z kilku cech. Oto wykres współczynników i dokładności klasyfikacji dla regularyzacji L1 (rysunek 2.18):

In[46]:

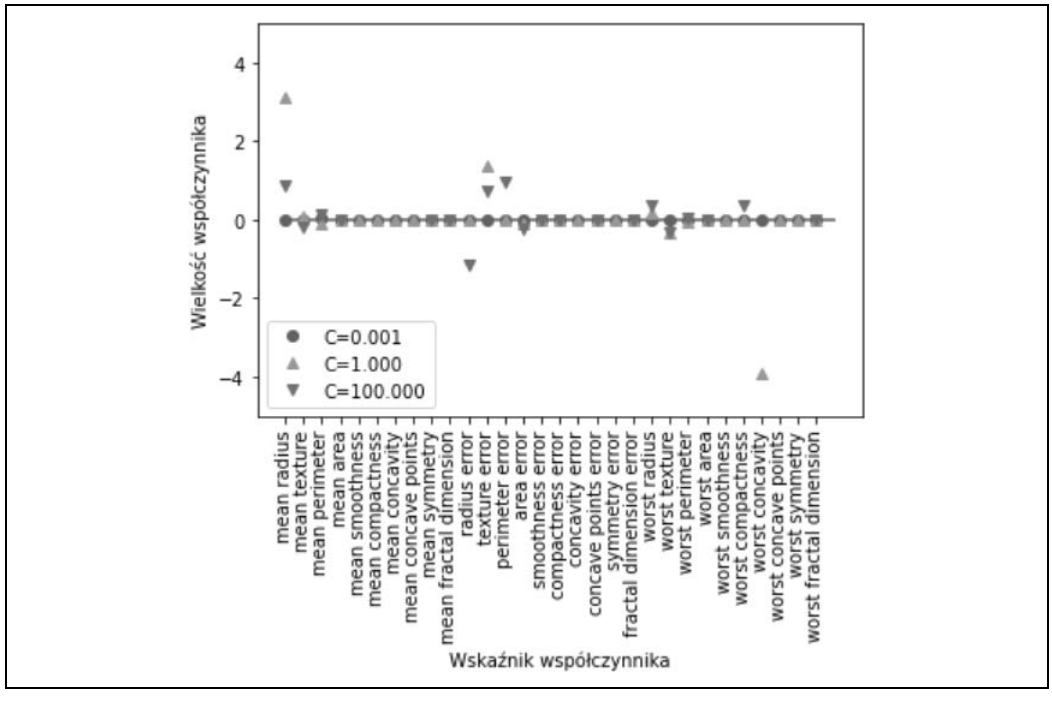
```
for C, marker in zip([0.001, 1, 100], ['o', '^', 'v']):
    lr_l1 = LogisticRegression(C=C, solver='liblinear', penalty="l1").fit(X_train, y_train)
    print("Dokładność dla danych uczących modelu l1 logreg z C={:.3f}: {:.2f}".format(C,
    ↪lr_l1.score(X_train, y_train)))
    print("Dokładność dla danych testowych modelu l1 logreg z C={:.3f}: {:.2f}".format(C,
    ↪lr_l1.score(X_test, y_test)))
    plt.plot(lr_l1.coef_.T, marker, label="C={:.3f}".format(C))

plt.xticks(range(cancer.data.shape[1]), cancer.feature_names, rotation=90)
plt.hlines(0, 0, cancer.data.shape[1])
plt.xlabel("Wskaźnik współczynnika")
plt.ylabel("Wielkość współczynnika")

plt.ylim(-5, 5)
plt.legend(loc=3)
```

Out[46]:

```
Dokładność dla danych uczących modelu l1 logreg z C=0.001: 0.91
Dokładność dla danych testowych modelu l1 logreg z C=0.001: 0.92
Dokładność dla danych uczących modelu l1 logreg z C=1.000: 0.96
Dokładność dla danych testowych modelu l1 logreg z C=1.000: 0.96
Dokładność dla danych uczących modelu l1 logreg z C=100.000: 0.99
Dokładność dla danych testowych modelu l1 logreg z C=100.000: 0.98
```



Rysunek 2.18. Współczynniki wyuczone przez regresję logistyczną z funkcją kary L1 w zestawie danych Breast Cancer dla różnych wartości parametru C

Jak widać, istnieje wiele podobieństw między modelami liniowymi do klasyfikacji binarnej a modelami liniowymi do regresji. Podobnie jak w przypadku regresji, główną różnicą między modelami jest parametr `penalty`, który wpływa na regularyzację i na to, czy model będzie używał wszystkich dostępnych cech, czy wybierze tylko ich podzbiór.

Modele liniowe dla klasyfikacji wieloklasowej

Wiele liniowych modeli klasyfikacji służy wyłącznie do klasyfikacji binarnej i nie obejmuje przypadku wieloklasowego w naturalny sposób (z wyjątkiem regresji logistycznej). Powszechną techniką rozszerzania algorytmu klasyfikacji binarnej na algorytm klasyfikacji wieloklasowej jest podejście *jeden kontra reszta*. W podejściu jeden kontra reszta model binarny jest uczony dla każdej klasy w taki sposób, żeby odróżnić ją od wszystkich innych klas, w wyniku czego powstaje tyle modeli binarnych, ile jest klas. W celu prognozy wszystkie klasyfikatory binarne są uruchamiane w punkcie testowym. Klasyfikator, który ma najwyższy wynik w swojej pojedynczej klasie, „wygrywa", a ta etykieta klasy jest zwracana jako prognoza.

Posiadanie jednego klasyfikatora binarnego na klasę skutkuje posiadaniem jednego wektora współczynników (w) i jednego punktu przecięcia z osią (b) dla każdej klasy. Ta klasa, dla której wynik podanej tutaj formuły ufności klasyfikacji jest najwyższy, jest przypisaną etykietą klasy:

$$w[0] * x[0] + w[1] * x[1] + ... + w[p] * x[p] + b$$

Zasady matematyczne, które są podstawą wieloklasowej regresji logistycznej, różnią się nieco od podejścia jeden kontra reszta, ale ich zastosowanie również daje jeden wektor współczynników i punkt przecięcia z osią na klasę; używają także tych samych metod prognozowania.

Zastosujmy metodę jeden kontra reszta do prostego zestawu danych klasyfikacji trzech klas. Używamy dwuwymiarowego zestawu danych, w którym każda klasa jest określona przez dane pobrane z rozkładu Gaussa, jak przedstawiono na rysunku 2.19:

`In[47]:`

```
from sklearn.datasets import make_blobs

X, y = make_blobs(random_state=42)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Cecha 0")
plt.ylabel("Cecha 1")
plt.legend(["Klasa 0", "Klasa 1", "Klasa 2"])
```

Teraz uczymy klasyfikator `LinearSVC` na zestawie danych:

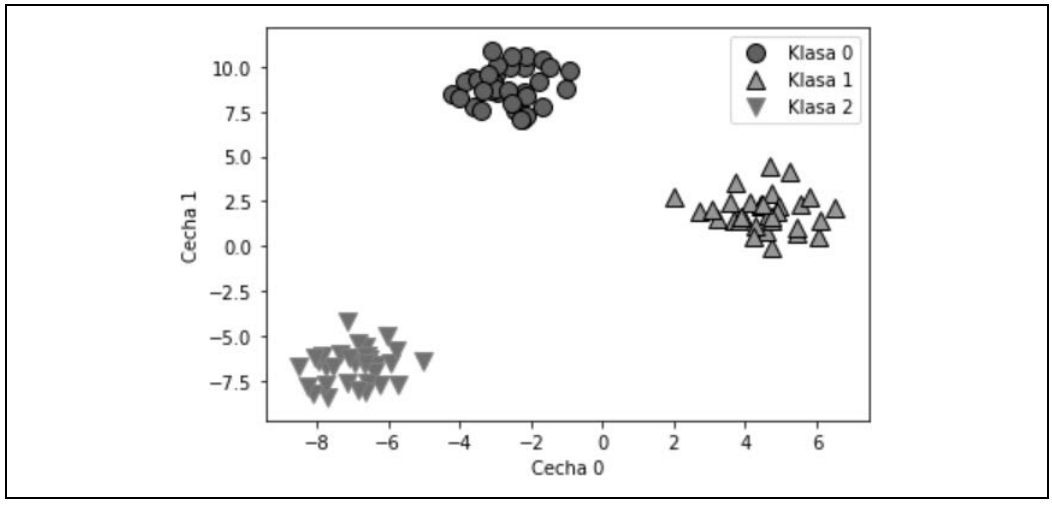
`In[48]:`

```
linear_svm = LinearSVC().fit(X, y)
print("Kształt współczynnika: ", linear_svm.coef_.shape)
print("Kształt przecięcia: ", linear_svm.intercept_.shape)
```

`Out[48]:`

```
Kształt współczynnika: (3, 2)
Kształt przecięcia: (3,)
```

Widzimy, że kształt atrybutu `coef_` to `(3, 2)`, co oznacza, że każdy jego wiersz zawiera wektor współczynników dla jednej z trzech klas, a każda kolumna zawiera wartość współczynnika dla określonej cechy (w tym zestawie danych są dwie). Atrybut `intercept_` przechowuje teraz jednowymiarową tablicę, w której znajdują się punkty przecięcia dla każdej klasy.

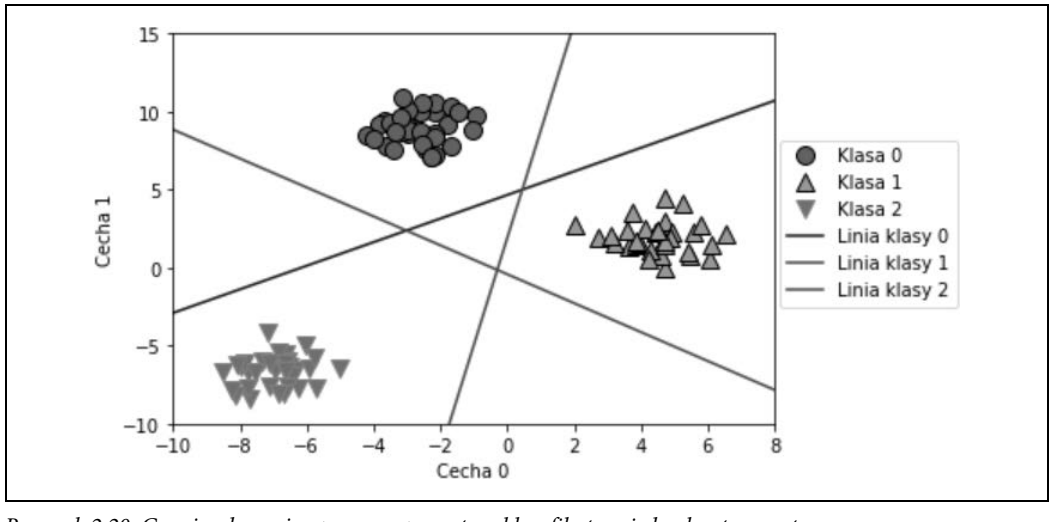


Rysunek 2.19. Dwuwymiarowy zestaw danych, który zawiera trzy klasy

Linie podane przez trzy klasyfikatory binarne przedstawiono na rysunku 2.20:

In[49]:

```
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
line = np.linspace(-15, 15)
for coef, intercept, color in zip(linear_svm.coef_, linear_svm.intercept_, ['b', 'r', 'g']):
    plt.plot(line, -(line * coef[0] + intercept) / coef[1], c=color)
plt.ylim(-10, 15)
plt.xlim(-10, 8)
plt.xlabel("Cecha 0")
plt.ylabel("Cecha 1")
plt.legend(['Klasa 0', 'Klasa 1', 'Klasa 2', 'Linia klasy 0', 'Linia klasy 1', 'Linia klasy
↪2'], loc=(1.01, 0.3))
```



Rysunek 2.20. Granice decyzyjne poznane przez trzy klasyfikatory jeden kontra reszta

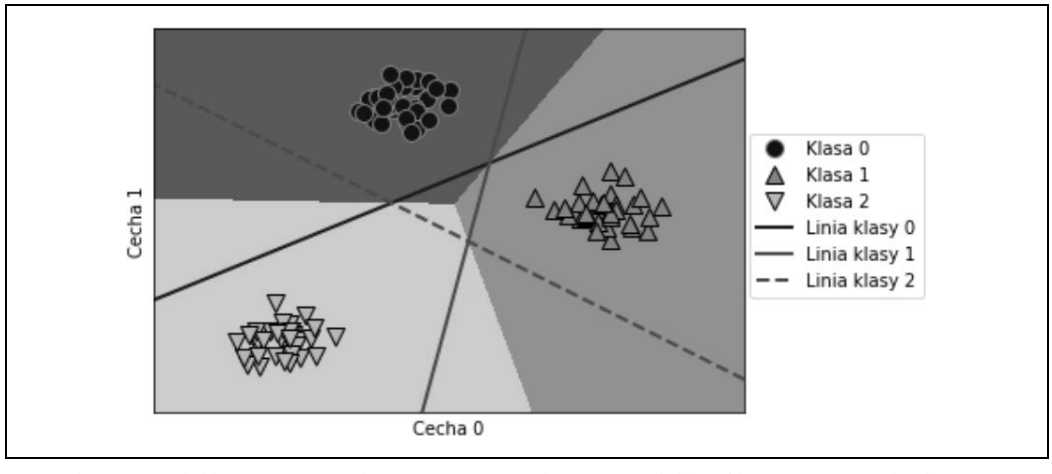
Widać, że wszystkie punkty z danych uczących należące do klasy 0 znajdują się powyżej linii, która odpowiada klasie 0, co oznacza, że znajdują się po stronie „klasy 0" tego klasyfikatora binarnego. Punkty w klasie 0 znajdują się powyżej linii, która odpowiada klasie 2, co oznacza, że są klasyfikowane przez binarny klasyfikator dla klasy 2 jako „reszta". Punkty, które należą do klasy 0, znajdują się po lewej stronie linii odpowiadającej klasie 1, co oznacza, że klasyfikator binarny dla klasy 1 również klasyfikuje je jako „reszta". Dlatego każdy punkt w tym obszarze zostanie sklasyfikowany jako klasa 0 przez klasyfikator ostateczny (wynik formuły ufności klasyfikacji dla klasyfikatora 0 jest większy od 0, a dla pozostałych dwóch klas mniejszy od 0).

A co z trójkątem w środku wykresu? Wszystkie trzy klasyfikatory binarne klasyfikują punkty w tym miejscu jako „reszta". Do jakiej klasy zostałby przypisany punkt, który się tam znajduje? Odpowiedzią jest ta klasa, która ma najwyższą wartość dla formuły klasyfikacyjnej: klasa najbliższej linii.

Kolejny przykład, który zilustrowano na rysunku 2.21, przedstawia przykład prognozy dla wszystkich obszarów przestrzeni 2D:

In[50]:

```
mglearn.plots.plot_2d_classification(linear_svm, X, fill=True, alpha=.7)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
line = np.linspace(-15, 15)
for coef, intercept, color in zip(linear_svm.coef_, linear_svm.intercept_, ['b', 'r', 'g']):
    plt.plot(line, -(line * coef[0] + intercept) / coef[1], c=color)
plt.legend(['Klasa 0', 'Klasa 1', 'Klasa 2', 'Linia klasy 0', 'Linia klasy 1', 'Linia klasy
↪2'], loc=(1.01, 0.3))
plt.xlabel("Cecha 0")
plt.ylabel("Cecha 1")
```



Rysunek 2.21. Wieloklasowe granice decyzyjne wyprowadzone z trzech klasyfikatorów typu jeden kontra reszta

Mocne, słabe punkty i parametry

Głównym parametrem modeli liniowych jest parametr regularyzacji, który w modelach regresji nazywamy `alfa`, a w przypadku `LinearSVC` i `LogisticRegression` używamy nazwy `C`. Duże wartości `alfa` lub małe wartości `C` oznaczają proste modele. Dostrojenie tych parametrów jest dość ważne, w szczególności w przypadku modeli regresji. Zwykle parametry `C` i `alfa` są wyszukiwane na skali logarytmicznej. Inną decyzją, jaką musisz podjąć, jest to, czy chcesz użyć regularyzacji L1, czy L2.

Jeśli zakładasz, że właściwie tylko kilka z Twoich cech jest ważnych, użyj regularyzacji L1. W przeciwnym razie musisz domyślnie ustawić regularyzację L2. Regularyzacja L1 może być również przydatna, jeśli ważna jest interpretacja modelu. Łatwiej wyjaśnić, które cechy są ważne dla modelu i jakie są skutki tych funkcji, ponieważ zostanie użyte tylko kilka funkcji.

Modele liniowe można bardzo szybko wyuczyć i równie szybko tworzyć z ich użyciem prognozy. Skalują się do bardzo dużych zestawów danych i dobrze pracują na rzadkich danych. Jeśli Twoje dane składają się z setek tysięcy lub milionów próbek, możesz chcieć je zbadać za pomocą dostępnej w modelu `LogisticRegression` i `Ridge` opcji `solver='sag'`, która może być szybsza niż domyślna w przypadku dużych zestawów danych. Inne opcje to klasa `SGDClassifier` i klasa `SGDRegressor`, w których zaimplementowano jeszcze bardziej skalowalne wersje opisanych tutaj modeli liniowych.

Kolejną zaletą modeli liniowych jest to, że przy użyciu wzorów, które widzieliśmy wcześniej dla regresji i klasyfikacji, stosunkowo łatwo jest zrozumieć, w jaki sposób tworzone są prognozy. Niestety, często nie jest do końca jasne, dlaczego współczynniki są takie, jakie są. Jest to szczególnie ważne, jeśli zestaw danych ma wysoce skorelowane funkcje; w takich przypadkach współczynniki mogą być trudne do interpretacji.

Modele liniowe zazwyczaj działają dobrze, gdy liczba cech jest duża w porównaniu z liczbą próbek. Są również często używane w bardzo dużych zestawach danych, po prostu dlatego, że nie jest możliwe uczenie innych modeli. Jednak w przestrzeniach o niższych wymiarach inne modele mogą dawać lepszą wydajność uogólniania. Kilka przykładów, w których modele liniowe zawodzą, omówiono w sekcji „Maszyny wektorów nośnych”, który znajduje się na stronie 88.

Łączenie metod

Metoda `fit` wszystkich modeli `scikit-learn` zwraca instancję danej klasy (`self`). Pozwala to na napisanie kodu, który jest podobny do poniższego i którego już w tym rozdziale intensywnie używaliśmy:

`In[51]:`

```
# w jednym wierszu utwórz instancję modelu i wywołaj jego metodę fit
logreg = LogisticRegression().fit(X_train, y_train)
```

W powyższym przykładzie, aby przypisać wyuczony model do zmiennej `logreg`, użyto wartości zwracanej przez metodę `fit` (czyli `self`). Ta konkatenacja wywołań metod (tutaj `__init__`, a następnie `fit`) jest znana jako *łączenie metod*. Innym powszechnym zastosowaniem łączenia metod w `scikit-learn` jest wywołanie w jednym wierszu metod `fit` i `predict`:

`In[52]:`

```
logreg = LogisticRegression()
y_pred = logreg.fit(X_train, y_train).predict(X_test)
```

W jednym wierszu można nawet jednocześnie stworzyć instancję modelu oraz wywołać metody `fit` i `predict`:

`In[53]:`

```
y_pred = LogisticRegression().fit(X_train, y_train).predict(X_test)
```

Ta bardzo krótka metoda pisania kodu nie jest jednak idealna. W jednym wierszu wywołujemy wiele różnych metod, co może utrudniać czytanie kodu. Dodatkowo dopasowany model regresji logistycznej nie jest przechowywany w żadnej zmiennej, więc nie możemy go sprawdzić ani użyć do prognozowania innych danych.

Naiwne klasyfikatory Bayesa

Naiwne klasyfikatory Bayesa to rodzina klasyfikatorów, które są dość podobne do modeli liniowych, omówionych w poprzednim podrozdziale. Są jednak jeszcze szybsze w uczeniu. Ceną za tę wydajność jest to, że naiwne modele Bayesa często zapewniają wydajność generalizacji, która jest nieco gorsza niż w przypadku klasyfikatorów liniowych, takich jak `LogisticRegression` i `LinearSVC`.

Powodem, dla którego naiwne modele Bayesa są tak wydajne, jest to, że uczą się parametrów, biorąc pod uwagę każdą funkcję z osobna i zbierając proste statystyki dla każdej klasy z każdej funkcji. Istnieją trzy rodzaje naiwnych klasyfikatorów Bayesa, które zaimplementowano w bibliotece `scikit-learn`: `GaussianNB`, `BernoulliNB` i `MultinomialNB`. Klasyfikator `GaussianNB` można zastosować do dowolnych danych ciągłych, natomiast klasyfikator `BernoulliNB` zakłada dane binarne, a `MultinomialNB` przyjmuje dane zliczeniowe (to znaczy, że każda cecha reprezentuje jakąś liczbę całkowitą, np. to, jak często słowo występuje w zdaniu). Klasyfikatory `BernoulliNB` i `MultinomialNB` są najczęściej używane w klasyfikacji danych tekstowych.

Klasyfikator `BernoulliNB` służy do zliczania, jak często wartość każdej cechy każdej z klas jest różna od 0. Najłatwiej zrozumieć to na przykładzie:

`In[54]:`

```
X = np.array([[0, 1, 0, 1],
    [1, 0, 1, 1],
    [0, 0, 0, 1],
    [1, 0, 1, 0]])
y = np.array([0, 1, 0, 1])
```

Kod widoczny w powyższym przykładzie tworzy cztery punkty danych, z których każdy ma cztery cechy binarne. Istnieją dwie klasy, 0 i 1. Dla klasy 0 (pierwszy i trzeci punkt danych) w dwóch przypadkach pierwsza cecha ma wartość 0 i niezerową wartość 0 razy, druga cecha w pierwszym punkcie ma wartość 0, a w trzecim punkcie niezerową itd. Te same liczby są następnie obliczane dla punktów danych w drugiej klasie. Zasadę zliczania niezerowych wpisów na klasę przedstawiono w poniższym listingu:

`In[55]:`

```
counts = {}
for label in np.unique(y):
    # iteruj po każdej klasie
    # zlicz (sumę) wpisów po jednej na cechę
    counts[label] = X[y == label].sum(axis=0)
print("Sumy cech:\n{}".format(counts))
```

`Out[55]:`

```
Sumy cech:
{0: array([0, 1, 0, 2]), 1: array([2, 0, 2, 1])}
```

Pozostałe dwa naiwne modele Bayesa, `MultinomialNB` i `GaussianNB`, różnią się nieco pod względem rodzajów obliczanych statystyk. `MultinomialNB` uwzględnia średnią wartość każdej cechy dla każdej klasy, a `GaussianNB` przechowuje średnią wartość, a także odchylenie standardowe każdej cechy dla każdej klasy.

W celu wykonania prognozy ze statystykami dla każdej z klas porównany zostaje punkt danych, a wynikiem prognozy jest najlepiej pasująca klasa. Co ciekawe, prowadzi to do formuły prognozowania, która ma taką samą postać jak w modelach liniowych (patrz „Modele liniowe do klasyfikacji" na stronie 59), tak samo w przypadku `MultinomialNB`, jak i `BernoulliNB`. Niestety, w przypadku naiwnych modeli Bayesa atrybut `coef_` ma nieco inne znaczenie niż w modelach liniowych, w tym atrybut `coef_` nie jest tym samym, co parametr *w*.

Mocne i słabe strony oraz parametry

Modele `MultinomialNB` i `BernoulliNB` mają jeden parametr `alfa`, który kontroluje złożoność modelu. Parametr `alfa` działa w taki sposób, że algorytm dodaje do danych `alfa` wiele wirtualnych punktów danych, które mają dodatnie wartości dla wszystkich funkcji. Powoduje to „wygładzenie" statystyk. Duża wartość `alfa` oznacza większe wygładzanie, co skutkuje mniej złożonymi modelami. Wydajność algorytmu jest stosunkowo odporna na ustawienie parametru `alfa`, co oznacza, że jego wartość nie wpływa znacznie na wydajność. Jednak dostrojenie parametru `alfa` zwykle poprawia nieco dokładność.

Model `GaussianNB` jest najczęściej używany w przypadku danych o bardzo dużych wymiarach, a pozostałe dwa warianty naiwnych klasyfikatorów Bayesa są szeroko stosowane w przypadku rzadkich danych zliczeniowych, takich jak tekst. Model `MultinomialNB` zazwyczaj działa lepiej niż `BernoulliNB`, szczególnie w przypadku zestawów danych ze stosunkowo dużą liczbą funkcji niezerowych (tj. dużych dokumentów).

Naiwne modele Bayesa mają wiele tych samych mocnych i słabych stron co modele liniowe. Szybko się je uczy i szybko dokonują predykcji, a procedura uczenia jest łatwa do zrozumienia. Modele działają bardzo dobrze z wysokowymiarowymi, rzadkimi zestawami danych i są stosunkowo odporne na zmiany wartości parametrów. Naiwne modele Bayesa to świetne modele bazowe, często używane do pracy na bardzo dużych zestawach danych, gdy uczenie modelu może zająć zbyt dużo czasu, nawet w przypadku modelu liniowego.

Drzewa decyzyjne

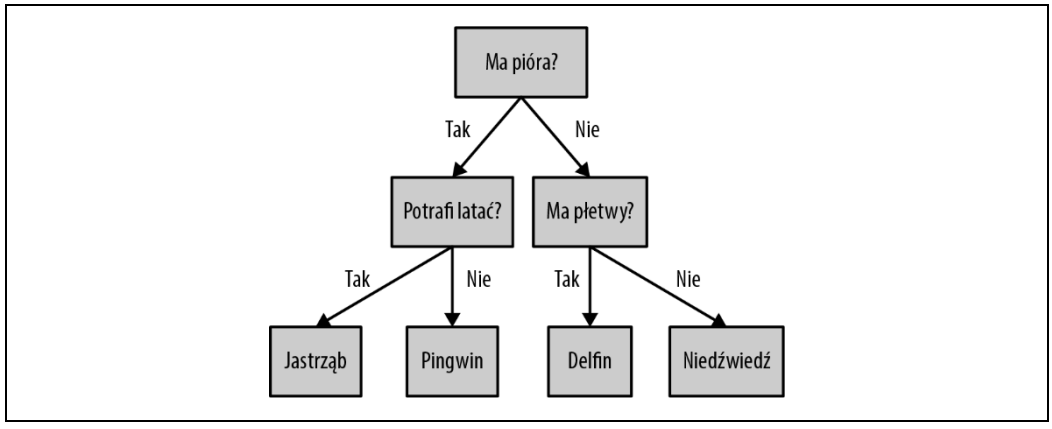
Drzewa decyzyjne są szeroko stosowanymi modelami zadań klasyfikacyjnych i regresyjnych. Zasadniczo uczą się hierarchii pytań typu „jeśli/to", które prowadzą do podjęcia decyzji.

Są one podobne do tych, które możesz zadawać w grze 20 pytań. Wyobraź sobie, że chcesz rozróżnić czworo zwierząt: niedźwiedzie, jastrzębie, pingwiny i delfiny. Twoim celem jest rozpoznanie zwierzęcia po zadaniu jak najmniejszej liczby pytań typu „jeśli/to". Możesz zacząć od pytania, czy zwierzę ma pióra, co zawęża liczbę możliwych zwierząt do dwóch. Jeśli odpowiedź brzmi „tak", możesz zadać inne pytanie, które pomoże Ci odróżnić jastrzębie od pingwinów. Przykładowo możesz zapytać, czy zwierzę potrafi latać. Jeśli nie ma piór, możliwy wybór spośród zwierząt to delfiny i niedźwiedzie i trzeba będzie zadać pytanie, które pozwoli na ich rozróżnienie — np. czy zwierzę ma płetwy.

Taką serię pytań można wyrazić w postaci drzewa decyzyjnego, jak pokazano na rysunku 2.22:

`In[56]:`

```
mglearn.plots.plot_animal_tree()
```



Rysunek 2.22. Drzewo decyzyjne, które pozwala rozróżnić kilka gatunków zwierząt

Na powyższej ilustracji każdy węzeł w drzewie przedstawia pytanie lub jest węzłem końcowym (zwanym także *liściem*), który zawiera odpowiedź. Krawędzie łączą odpowiedzi na pytanie z następnym pytaniem, które zostałoby zadane.

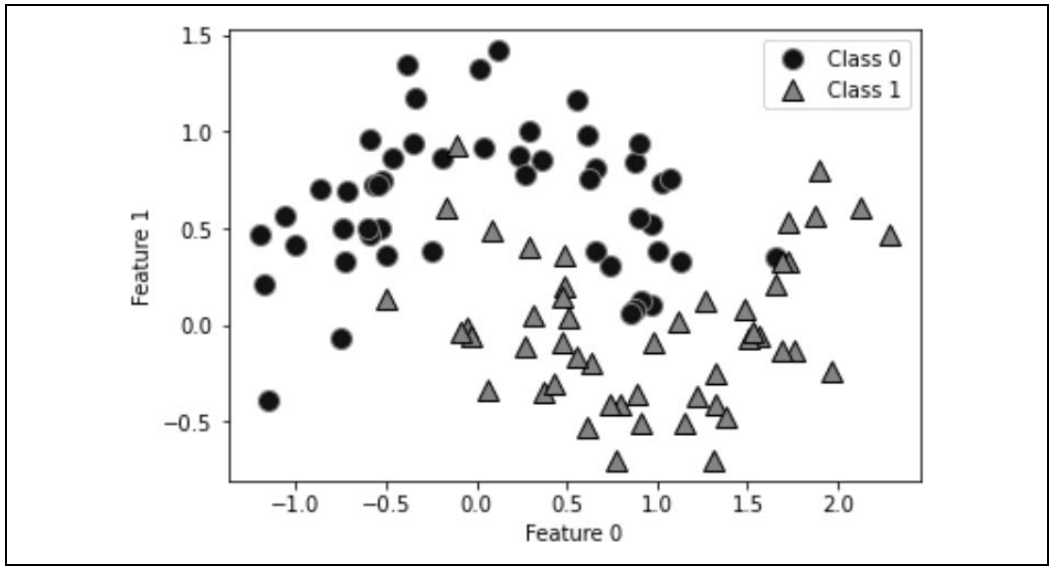
Mówiąc w żargonie uczenia maszynowego — zbudowaliśmy model, który rozróżnia cztery klasy zwierząt (jastrzębie, pingwiny, delfiny i niedźwiedzie) przy użyciu trzech cech: „ma pióra", „potrafi latać" i „ma płetwy". Zamiast tworzyć je ręcznie, możemy uczyć te modele na podstawie danych, korzystając z uczenia nadzorowanego.

Budowanie drzew decyzyjnych

Przejdźmy przez proces budowania drzewa decyzyjnego dla zestawu danych klasyfikacyjnych 2D, które przedstawiono na rysunku 2.23. Zestaw danych składa się z dwóch kształtów półksiężyców, a każda klasa zawiera 75 punktów danych. Zestaw danych będziemy nazywać `two_moons`.

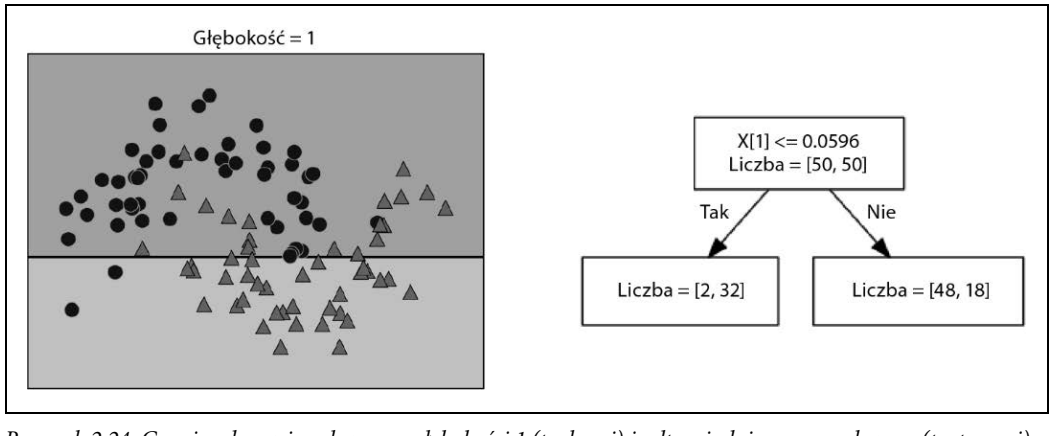
Uczenie drzewa decyzyjnego oznacza nauczenie go sekwencji pytań „jeśli/to", dzięki którym najszybciej zwróci prawidłową odpowiedź. W przypadku uczenia maszynowego pytania te nazywane są *testami* (nie należy ich mylić z zestawem testowym, czyli danymi, których używamy do testowania, aby sprawdzić, jak można uogólnić model). Zwykle dane nie są dostarczane w postaci binarnych cech, które można rozpoznać dzięki odpowiedzi na jakieś pytanie „tak" lub „nie", jak w przykładzie ze zwierzętami. Zamiast tego są reprezentowane jako cechy ciągłe, takie jak w zestawie danych 2D, który pokazano na rysunku 2.23. Testy używane na danych ciągłych mają postać „Czy wartość cechy *i* jest większa niż wartość *a*?".

Aby zbudować drzewo, algorytm przeszukuje wszystkie możliwe testy i znajduje ten, który zawiera najwięcej informacji o zmiennej docelowej. Pierwszy wybrany test przedstawiono na rysunku 2.24. Najwięcej informacji można uzyskać po podzieleniu zestawu danych w pionie przy `x[1]=0,0596`; to najlepiej oddzieli punkty z klasy 1 od punktów z klasy 2. Najwyższy węzeł, zwany także *korzeniem*, reprezentuje cały zestaw danych, składający się z 75 punktów należących do klasy 0 i 75 punktów z klasy 1. Podział jest wykonywany przez sprawdzenie, czy `x[1]<=0,0596`, co wskazuje czarna linia.

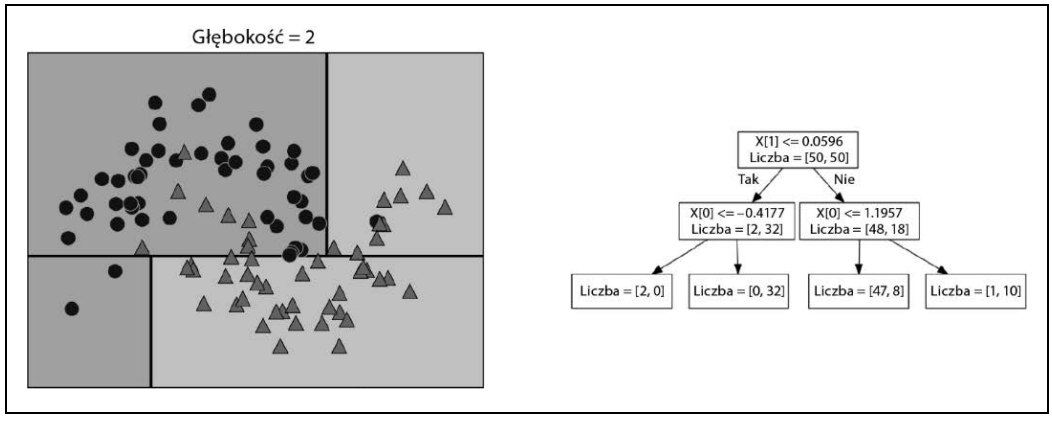


Rysunek 2.23. Zestaw danych o dwóch księżycach, na którym zostanie zbudowane drzewo decyzyjne

Jeśli test zwróci odpowiedź pozytywną, punkt jest przypisywany do lewego węzła, który zawiera 2 punkty należące do klasy 0 i 32 punkty z klasy 1. W przeciwnym razie punkt jest przypisywany do węzła prawego, który zawiera 48 punktów należących do klasy 0 oraz 18 punktów należących do klasy 1. Te dwa węzły odpowiadają górnym i dolnym regionom, które pokazano na rysunku 2.24. Mimo że pierwszy podział prawidłowo rozdzielił dwie klasy, dolny region nadal zawiera punkty należące do klasy 0, a górny — punkty należące do klasy 1. Możemy zbudować dokładniejszy model, powtarzając proces poszukiwania najlepszego testu w obu regionach. Następny podział najbardziej pouczający dla lewego i prawego regionu jest oparty na `x[0]`, co pokazano na rysunku 2.25.



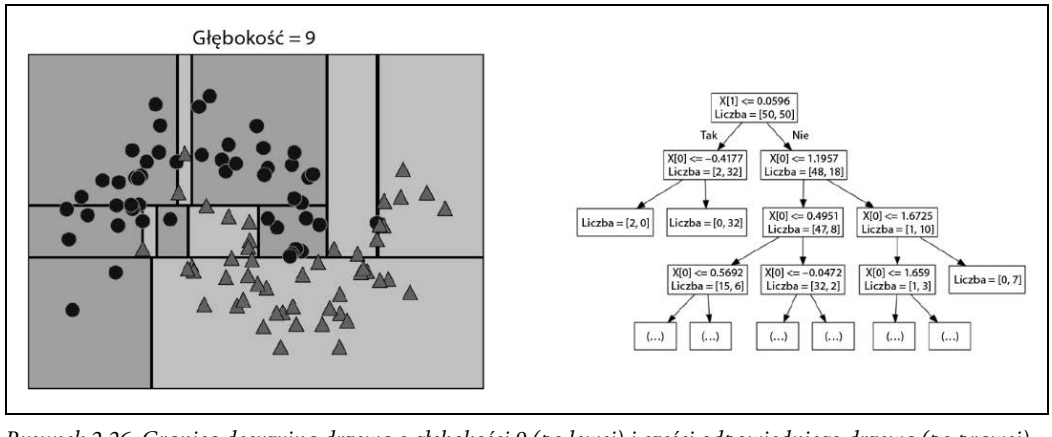
Rysunek 2.24. Granica decyzyjna drzewa o głębokości 1 (po lewej) i odpowiadającego mu drzewa (po prawej)



Rysunek 2.25. Granica decyzyjna drzewa o głębokości 2 (po lewej) i odpowiadające jej drzewo decyzyjne (po prawej)

Ten proces rekurencyjny tworzy binarne drzewo decyzji, w którym każdy węzeł zawiera test. Alternatywnie możesz myśleć o każdym teście jako o dzieleniu części danych, które są obecnie rozpatrywane, wzdłuż jednej osi. Daje to pogląd na algorytm jako budowanie hierarchicznej partycji. Ponieważ każdy test dotyczy tylko jednej cechy, regiony w wynikowym podziale zawsze mają granice równoległe do osi.

Rekursywne partycjonowanie danych jest powtarzane tak długo, aż każdy region w partycji (każdy liść w drzewie decyzyjnym) zawiera tylko jedną wartość docelową (pojedynczą klasę lub pojedynczą wartość regresji). Liść drzewa, który zawiera punkty danych, które mają tę samą wartość docelową, nazywamy punktem *czystym*. Ostateczne partycjonowanie tego zestawu danych pokazano na rysunku 2.26.



Rysunek 2.26. Granica decyzyjna drzewa o głębokości 9 (po lewej) i części odpowiedniego drzewa (po prawej); pełne drzewo jest dość duże i trudne do wizualizacji

Prognozowanie nowego punktu danych polega na sprawdzeniu, w którym regionie podziału przestrzeni cech znajduje się punkt, a następnie prognozowaniu celu większości punktów (lub pojedynczego celu w przypadku czystej liście) w tym regionie. Region można znaleźć, przechodząc

przez drzewo od korzenia i idąc w lewo lub w prawo, w zależności od tego, czy test zwróci odpowiedź „tak”, czy „nie”.

Dzięki użyciu dokładnie tej samej techniki możliwe jest również użycie drzew do zadań regresji. Aby dokonać predykcji, przeszukujemy drzewo na podstawie testów w każdym węźle i znajdujemy liść, w który wpada nowy punkt danych. Dane wyjściowe dla tego punktu danych to średni cel dla punktów uczących w tym liściu.

Kontrolowanie złożoności drzew decyzyjnych

Zazwyczaj budowanie drzewa w sposób opisany tutaj i kontynuowanie, aż wszystkie liście staną się czyste, prowadzi do modeli, które są bardzo złożone i nadmiernie dopasowane do danych uczących. Obecność czystych liści oznacza, że drzewo w zestawie uczącym jest w 100% dokładne; każdy punkt danych w zestawie uczącym znajduje się w liściu, który ma poprawną klasę większości. Po lewej stronie rysunku 2.26 można zobaczyć nadmierne dopasowanie. Regiony określone jako należące do klasy 1 znajdują się pośrodku wszystkich punktów należących do klasy 0. Z drugiej strony wokół punktu należącego do klasy 0 po prawej stronie znajduje się mały pasek rozpoznany jako klasa 0. Granica decyzyjna nie wygląda tak, jakbyśmy się mogli spodziewać, koncentruje się w dużej mierze na pojedynczych punktach odstających, które znajdują się daleko od innych punktów w tej klasie.

Istnieją dwie powszechne strategie zapobiegania nadmiernemu dopasowaniu: wczesne zatrzymywanie tworzenia drzewa (zwane także *przycinaniem wstępnym*) lub budowanie drzewa, ale następnie usuwanie lub zwijanie węzłów, które zawierają niewiele informacji (nazywane *późniejszym przycinaniem* lub po prostu *przycinaniem*). Możliwe kryteria wstępnego przycinania obejmują ograniczenie maksymalnej głębokości drzewa, ograniczenie maksymalnej liczby liści lub wymaganie minimalnej liczby punktów w węźle, aby móc je dalej dzielić.

Drzewa decyzyjne w bibliotece `scikit-learn` zaimplementowano w klasach `DecisionTreeRegressor` i `DecisionTreeClassifier`. Biblioteka `scikit-learn` realizuje tylko wstępne przycinanie, a nie późniejsze przycinanie.

Przyjrzyjmy się bardziej szczegółowo wpływowi wstępnego przycinania na zestaw danych Breast Cancer. Jak zawsze importujemy zestaw danych i dzielimy go na część uczącą i testową. Następnie budujemy model, używając domyślnego ustawienia pełnego rozwinięcia drzewa (powiększanie drzewa do momentu, aż wszystkie liście będą czyste). Ustawiamy w drzewie parametr `random_state`, który jest używany do wewnętrznego rozstrzygania remisów:

`In[58]:`

```
from sklearn.tree import DecisionTreeClassifier
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
cancer.data, cancer.target, stratify=cancer.target, random_state=42)
tree = DecisionTreeClassifier(random_state=0)
tree.fit(X_train, y_train)
print("Dokładność w zestawie uczącym: {:.3f}".format(tree.score(X_train, y_train)))
print("Dokładność w zestawie testowym: {:.3f}".format(tree.score(X_test, y_test)))
```

`Out[58]:`

```
Dokładność w zestawie uczącym: 1.000
Dokładność w zestawie testowym: 0.937
```

Zgodnie z oczekiwaniami dokładność zestawu uczącego wynosi 100% — ponieważ liście są czyste, drzewo było na tyle głębokie, że mogło doskonale zapamiętać wszystkie etykiety danych uczących. Dokładność zestawu testowego jest nieco gorsza niż w przypadku modeli liniowych, które oglądaliśmy wcześniej. Miały one około 95% dokładności.

Jeśli nie ograniczymy głębokości drzewa decyzyjnego, może się ono stać dowolnie głębokie i złożone. Nieprzycięte drzewa są zatem podatne na nadmierne dopasowanie i nie uogólniają dobrze nowych danych. Teraz zastosujmy wstępne przycinanie drzewa, które zatrzyma jego rozwijanie, zanim idealnie dopasuje się do danych uczących. Jedną z opcji jest zaprzestanie budowania drzewa po osiągnięciu określonej głębokości. W tym przypadku należy ustawić parametr `max_depth=4`, co oznacza, że można zadać tylko cztery kolejne pytania (por. rysunki 2.24 i 2.26). Ograniczenie głębokości drzewa zmniejsza nadmierne dopasowanie. Prowadzi to do mniejszej dokładności w zestawie uczącym, ale większej dokładności w zestawie testowym:

`In[59]:`

```
tree = DecisionTreeClassifier(max_depth=4, random_state=0)
tree.fit(X_train, y_train)
print("Dokładność w zestawie uczącym: {:.3f}".format(tree.score(X_train, y_train)))
print("Dokładność w zestawie testowym: {:.3f}".format(tree.score(X_test, y_test)))
```

`Out[59]:`

```
Dokładność w zestawie uczącym: 0.988
Dokładność w zestawie testowym: 0.951
```

Analiza drzew decyzyjnych

Korzystając z funkcji `export_graphviz` z modułu `tree`, możemy wygenerować grafikę, która przedstawia drzewo. Uruchomienie tej funkcji spowoduje zapisanie pliku w formacie *.dot*, który jest formatem pliku tekstowego do przechowywania wykresów. Ustawiliśmy opcję kolorowania węzłów w celu odzwierciedlenia większości klas w każdym węźle oraz — aby drzewo było właściwie opisane — przekazaliśmy nazwy klas i cech:

`In[61]:`

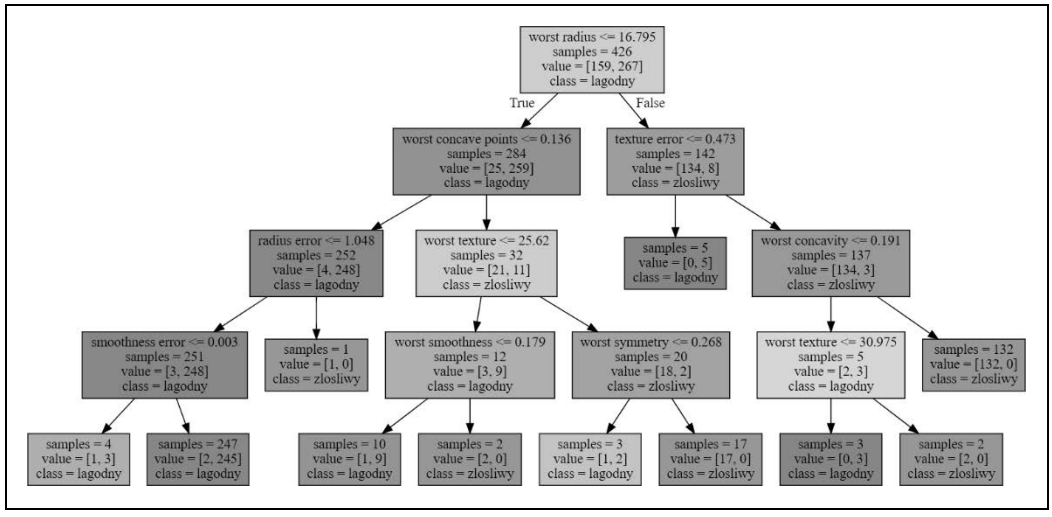
```
from sklearn.tree import export_graphviz
export_graphviz(tree, out_file="drzewo.dot", class_names=["zlosliwy", "lagodny"],
feature_names=cancer.feature_names, impurity=False, filled=True)
```

Możemy odczytać ten plik i wygenerować jego wizualizację, używając modułu `graphviz` (można użyć dowolnego programu, który potrafi czytać pliki z rozszerzeniem *.dot*), jak pokazano na rysunku 2.27:

`In[61]:`

```
import graphviz
with open("drzewo.dot") as f:
    dot_graph = f.read()
graphviz.Source(dot_graph)
```

Wizualizacja drzewa zapewnia doskonały, dogłębny obraz tego, w jaki sposób algorytm tworzy prognozy, i jest dobrym przykładem algorytmu uczenia maszynowego, który można łatwo wyjaśnić laikom. Jednak nawet w przypadku drzewa, które ma głębokość równą 4, jak w tym przypadku, może stać się ono nieco przytłaczające. Głębsze drzewa (często występują takie o głębokości 10) są jeszcze trudniejsze do opanowania. Jedną z pomocnych metod sprawdzania drzewa jest ustalenie,



Rysunek 2.27. Wizualizacja drzewa decyzyjnego zbudowanego na zestawie danych Breast Cancer

która ścieżka jest obierana przy klasyfikacji większości danych. Parametr `samples` widoczny w każdym węźle na rysunku 2.27 oznacza liczbę próbek w danym węźle, a wartość parametru `value` oddaje liczbę próbek na klasę. Jeśli przyjrzymy się gałęziom po prawej stronie drzewa, widzimy, że negatywny wynik dla warunku `worst radius <= 16,795` wystąpił w przypadku 8 łagodnych, ale też w przypadku 134 złośliwych próbek. Aby oddzielić te 8 łagodnych próbek, reszta gałęzi z tej strony drzewa stosuje drobniejsze rozróżnienia. Spośród 142 próbek, które zostały zakwalifikowane na prawą stronę drzewa podczas początkowego podziału, prawie wszystkie z nich (132) trafiają do liścia po prawej stronie.

Po lewej stronie drzewa znalazło się 25 złośliwych i 259 łagodnych próbek, które spełniają warunek `worst radius > 16,795`. Prawie wszystkie łagodne próbki trafiają do drugiego liścia od lewej, a większość pozostałych liści zawiera bardzo mało próbek.

Ważność cech w drzewach

Patrzenie na drzewo jako całość, aby podsumować jego działanie, może być trudne, ale możemy wyprowadzić kilka przydatnych właściwości, które mogą w tym pomóc. Najczęściej używanym podsumowaniem jest *ważność cechy*, określająca, jak ważna jest każda cecha dla decyzji, którą podejmuje drzewo. Jest to liczba z przedziału od 0 do 1 dla każdej funkcji, gdzie 0 oznacza „w ogóle nie używany”, a 1 oznacza „doskonale prognozuje cel”. Istotności cech zawsze sumują się do 1:

`In[62]:`

```
print("Ważność cech:\n{}".format(tree.feature_importances_))
```

`Out[62]:`

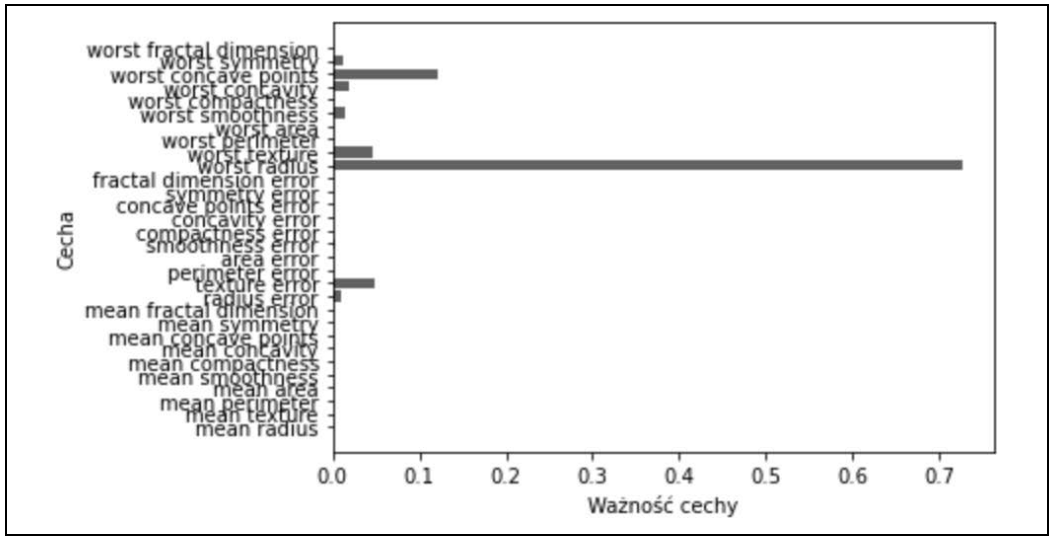
```
Ważność cech:
[0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.01019737 0.04839825
 0.         0.         0.0024156  0.         0.         0.
 0.         0.         0.72682851 0.0458159  0.         0.
 0.0141577  0.         0.018188   0.1221132  0.01188548 0.        ]
```

Możemy wizualizować ważność cech podobnie, jak wizualizujemy współczynniki w modelu liniowym, co przedstawiono na rysunku 2.28:

In[63]:

```
def plot_feature_importances_cancer(model):
    n_features = cancer.data.shape[1]
    plt.barh(range(n_features), model.feature_importances_, align='center')
    plt.yticks(np.arange(n_features), cancer.feature_names)
    plt.xlabel("Ważność cechy")
    plt.ylabel("Cecha")

plot_feature_importances_cancer(tree)
```



Rysunek 2.28. Istotność funkcji obliczona na podstawie drzewa decyzyjnego uzyskanego z zestawu danych Breast Cancer

Uważny czytelnik zauważy, że cecha użyta na szczycie drzewa („worst radius") jest zdecydowanie najważniejsza. Potwierdza to naszą obserwację podczas analizy drzewa, że pierwszy poziom już dość dobrze oddziela dwie klasy.

Jeśli jednak cecha ma niską wartość współczynnika `feature_importance`, nie oznacza to, że nie zawiera żadnych informacji. Oznacza tylko, że obiekt nie został wybrany przez drzewo prawdopodobnie dlatego, że inny element przechowuje te same informacje.

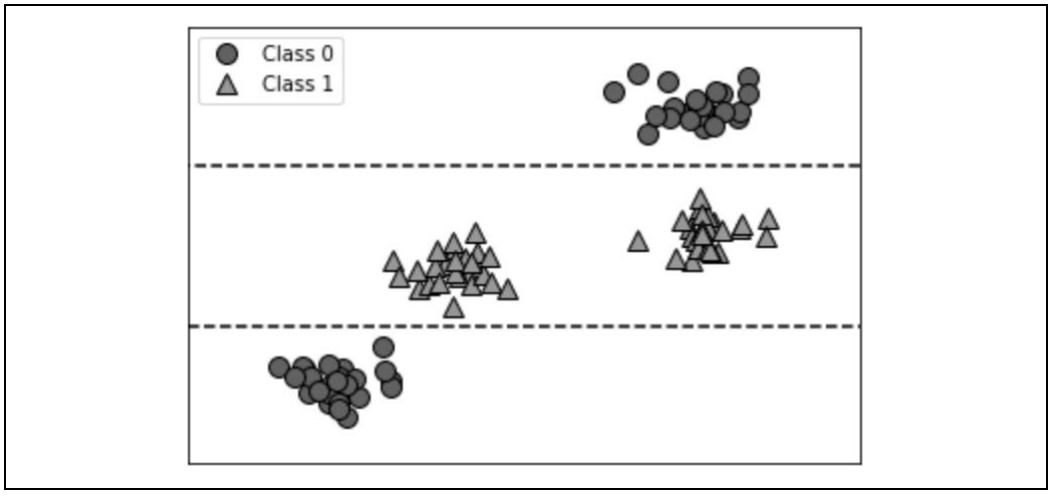
W przeciwieństwie do współczynników w modelach liniowych ważność cech zawsze jest dodatnia i nie przechowuje informacji o tym, na którą klasę cecha wskazuje. Ważność cech mówi nam, że „worst radius" jest ważny, ale nie komunikuje, czy jego duża wartość wskazuje na to, czy próbka jest łagodna, czy złośliwa. W rzeczywistości może nie być takiej prostej zależności między cechami a klasą, jaką widać w przykładzie przedstawionym na rysunkach 2.29 i 2.30:

In[64]:

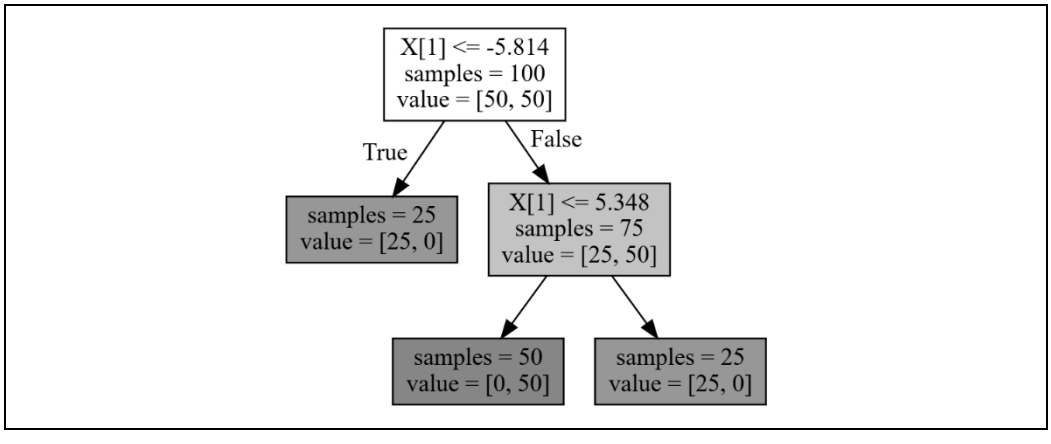
```
tree = mglearn.plots.plot_tree_not_monotone()
display(tree)
```

Out[64]:

```
Feature importances: [ 0. 1.]
```



Rysunek 2.29. Dwuwymiarowy zestaw danych, w którym obiekt na osi y ma niekonsekwentną zależność z etykietą klasy i granicami decyzyjnymi wyznaczonymi przez drzewo decyzyjne



Rysunek 2.30. Drzewo decyzyjne stworzone na podstawie danych przedstawionych na rysunku 2.29

Na wykresie przedstawiono zestaw danych z dwiema cechami i dwiema klasami. Wszystkie informacje są zawarte w `X[1]`, a `X[0]` w ogóle nie jest używane. Jednak relacja między `X[1]` a klasą wyjściową nie jest monotonna, co oznacza, że nie możemy powiedzieć, że „wysoka wartość `X[0]` oznacza klasę 0, a mała wartość oznacza klasę 1 (lub odwrotnie).

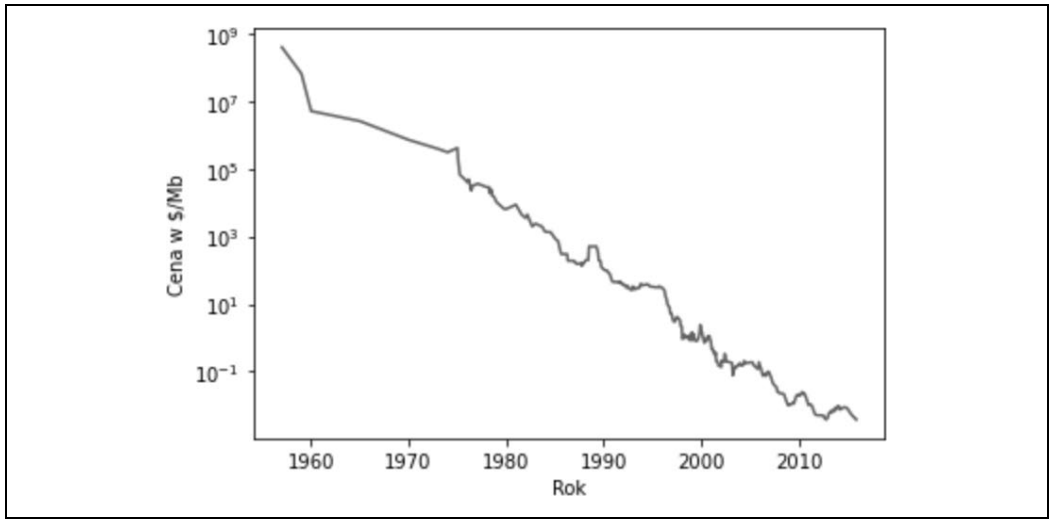
Chociaż skupiliśmy się na drzewach decyzyjnych do klasyfikacji, to wszystko, co omówiliśmy, jest tak samo prawdziwe w przypadku drzew decyzyjnych dla regresji, jak zaimplementowano w klasie `DecisionTreeRegressor`. Zastosowanie i analiza drzew regresji są bardzo podobne do tej z drzew klasyfikacyjnych. Chcemy jednak zwrócić uwagę na jedną szczególną właściwość stosowania modeli opartych na drzewach do regresji. Model `DecisionTreeRegressor` (i wszystkie inne modele regresji oparte na drzewie) nie jest w stanie ekstrapolować ani prognozować poza zakresem danych uczących.

Przyjrzyjmy się temu bardziej szczegółowo, korzystając z zestawu danych historycznych cen pamięci komputera (RAM). Na rysunku 2.31 przedstawiono zestaw danych z datą na osi *x* i ceną 1 megabajta pamięci RAM w danym roku na osi *y*:

In[65]:

```
import pandas as pd
ram_prices = pd.read_csv("data/ram_price.csv")

plt.semilogy(ram_prices.date, ram_prices.price)
plt.xlabel("Rok")
plt.ylabel("Cena w $/Mb")
```



Rysunek 2.31. Historyczny rozwój ceny pamięci RAM, wykreślony na skali logarytmicznej

Zwróć uwagę na logarytmiczną skalę osi *y*. Przy logarytmicznej skali wykresu relacja wydaje się być prawie liniowa, a więc poza pewnymi nierównościami powinna być stosunkowo łatwa do prognozowania.

Sporządzimy prognozę na lata po 2000 roku. Użyjemy do tego danych historycznych do tego momentu z datą jako jedyną cechą. Porównamy dwa proste modele: `DecisionTreeRegressor` i `LinearRegression`. Przeskalujemy ceny za pomocą logarytmu, aby zależność była względnie liniowa. Nie ma to znaczenia dla modelu `DecisionTreeRegressor`, ale powoduje dużą różnicę w przypadku `LinearRegression` (omówimy to bardziej szczegółowo w rozdziale 4.). Po wyuczeniu modeli i wykonaniu prognoz stosujemy mapę wykładniczą, aby cofnąć transformację logarytmiczną. Prognozujemy na całym zestawie danych do celów wizualizacji, ale do oceny ilościowej weźmiemy pod uwagę tylko testowy zestaw danych:

In[66]:

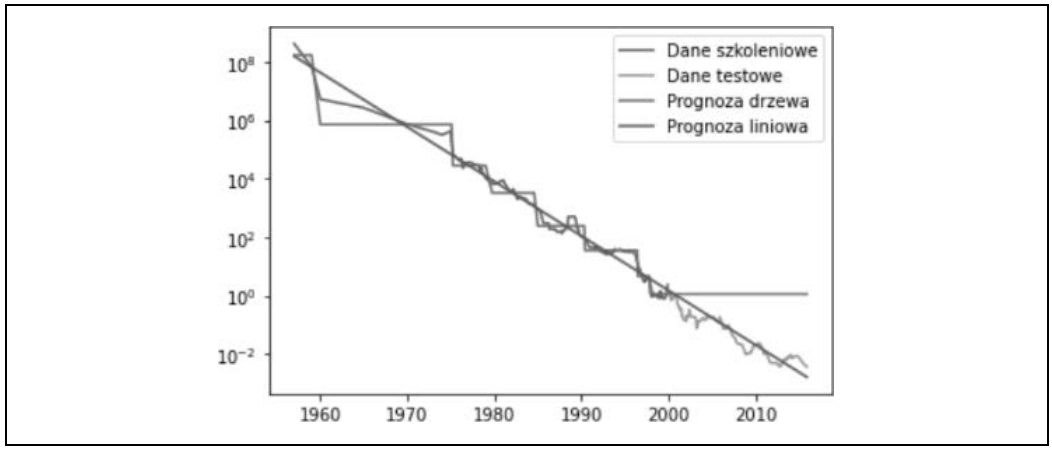
```
from sklearn.tree import DecisionTreeRegressor
# użyj danych historycznych do prognozowania cen po 2000 roku
data_train = ram_prices[ram_prices.date < 2000]
data_test = ram_prices[ram_prices.date >= 2000]
# twórz prognozy ceny na podstawie daty
X_train = data_train.date[:, np.newaxis]
```

```
# aby uzyskać prostszą relację danych do celu, używamy transformacji logarytmicznej
y_train = np.log(data_train.price)
tree = DecisionTreeRegressor().fit(X_train, y_train)
linear_reg = LinearRegression().fit(X_train, y_train)
# twórz prognozy na wszystkich danych
X_all = ram_prices.date[:, np.newaxis]
pred_tree = tree.predict(X_all)
pred_lr = linear_reg.predict(X_all)
# cofnij transformację logarytmiczną
price_tree = np.exp(pred_tree)
price_lr = np.exp(pred_lr)
```

Na rysunku 2.32 przedstawiono porównanie prognoz drzewa decyzyjnego i modelu regresji liniowej z prawdą podstawową:

In[67]:

```
plt.semilogy(data_train.date, data_train.price, label="Dane uczące")
plt.semilogy(data_test.date, data_test.price, label="Dane testowe")
plt.semilogy(ram_prices.date, price_tree, label="Prognoza drzewa")
plt.semilogy(ram_prices.date, price_lr, label="Prognoza liniowa")
plt.legend()
```



Rysunek 2.32. Porównanie prognoz wykonanych przez model liniowy i prognoz wykonanych przez drzewo regresji na danych o cenach pamięci RAM

Różnica między modelami jest dość uderzająca. Tak jak widzieliśmy, model liniowy aproksymuje dane za pomocą linii. Linia ta zapewnia całkiem dobrą prognozę dla danych testowych (dla lat po 2000 roku), a jednocześnie wychwytuje niektóre drobniejsze różnice zarówno w danych uczących, jak i testowych. Z drugiej strony model drzewa zapewnia doskonałe prognozy w przypadku danych uczących; nie ograniczyliśmy złożoności drzewa, więc model nauczył się całego zestawu danych. Jednak po opuszczeniu zakresu, dla którego model ma dane, prognozuje on po prostu ostatni znany punkt. Drzewo nie ma możliwości generowania „nowych" odpowiedzi poza tym, co było widoczne w danych uczących. Ta wada dotyczy wszystkich modeli opartych na drzewach[9].

[9] W rzeczywistości możliwe jest tworzenie bardzo dobrych prognoz za pomocą modeli opartych na drzewach (np. by przewidzieć, czy cena wzrośnie, czy spadnie). Celem tego przykładu nie było pokazanie, że drzewa są złym modelem szeregów czasowych, ale zilustrowanie określonej właściwości sposobu, w jaki drzewa tworzą prognozy.

Mocne, słabe strony i parametry

Jak wspomniano wcześniej, parametrami sterującymi złożonością modelu w drzewach decyzyjnych są parametry wstępnie przycięte, które zatrzymują budowanie drzewa, zanim zostanie ono w pełni opracowane. Aby zapobiec nadmiernemu dopasowaniu, zwykle wystarczy wybrać jedną ze strategii wstępnego przycinania — ustawienie wartości `max_depth`, `max_leaf_nodes` lub `min_samples_leaf`.

Drzewa decyzyjne mają dwie zalety w stosunku do wielu algorytmów, które omówiliśmy do tej pory: wynikowy model może być łatwo wizualizowany i zrozumiały dla laików (przynajmniej w przypadku mniejszych drzew), a algorytmy są całkowicie niezmienne niezależnie od ilości danych. Ponieważ w przypadku algorytmów drzewa decyzyjnego każda cecha jest przetwarzana osobno, a możliwe podziały danych nie zależą od skali, nie jest potrzebne żadne przetwarzanie wstępne, takie jak normalizacja czy standaryzacja funkcji. Drzewa decyzyjne działają dobrze szczególnie w przypadku cech, które są w zupełnie różnych skalach lub stanowią mieszankę funkcji binarnych i ciągłych.

Główną wadą drzew decyzyjnych jest to, że nawet przy korzystaniu z przycinania wstępnego mają tendencję do nadmiernego dopasowania i wykonują prognozy ze słabą wydajnością. Dlatego w większości zastosowań zamiast pojedynczego drzewa decyzyjnego zwykle używane są metody zespolone, które omówimy dalej.

Zespoły drzew decyzyjnych

Zespół (ang. *ensemble*) to metoda łączenia wielu modeli uczenia maszynowego w celu tworzenia bardziej zaawansowanych modeli. W literaturze dotyczącej uczenia maszynowego znajduje się wiele modeli należących do tej kategorii, ale istnieją dwa modele zespołowe, które okazują się skuteczne w przypadku szerokiego zakresu zestawów danych do klasyfikacji i regresji. Oba wykorzystują drzewa decyzyjne jako elementy składowe: lasy losowe i drzewa decyzyjne z gradientem.

Lasy losowe

Główną wadą drzew decyzyjnych jest to, że mają tendencję do nadmiernego dopasowywania się do danych uczących. Lasy losowe to jeden ze sposobów na rozwiązanie tego problemu. Las losowy to zasadniczo zestaw drzew decyzyjnych, w których każde różni się nieco od pozostałych. Idea, która za nim stoi, polega na tym, że każde drzewo może stosunkowo dobrze prognozować, ale prawdopodobnie będzie nadmiernie dopasowywać się do części danych. Jeśli zbudujemy wiele drzew, z których wszystkie działają dobrze i są na różne sposoby nadmiernie dopasowane, to możemy zmniejszyć nadmierne dopasowanie, uśredniając ich wyniki. To zmniejszenie nadmiernego dopasowania przy zachowaniu zdolności predykcyjnej drzew można wykazać za pomocą rygorystycznej matematyki.

Aby wdrożyć tę strategię, musimy zbudować wiele drzew decyzyjnych. Każde drzewo powinno wykonywać prognozowanie w zadowalający sposób i różnić się od innych drzew. Lasy losowe biorą swoją nazwę od wstrzykiwania losowości w strukturę drzewa w celu zapewnienia, że każde drzewo będzie inne. Istnieją dwa sposoby, w jakie wstrzykuje się losowość w drzewa losowe: wybranie punktów danych użytych do zbudowania drzewa oraz wybranie cech w każdym teście podziału. Przyjrzyjmy się bliżej temu procesowi.

Budowanie lasów losowych. Aby zbudować model lasu losowego, musisz zdecydować, ile drzew chcesz zbudować (parametr `n_estimators` klasy `RandomForestRegressor` lub `RandomForestClassifier`). Powiedzmy, że ma ich być 10. Drzewa te zostaną zbudowane całkowicie niezależnie od siebie, a algorytm, aby upewnić się, że są one różne, dla każdego z nich dokona różnych losowych wyborów. Aby zbudować drzewo, najpierw bierzemy tzw. *próbkę bootstrap* naszych danych. Oznacza to, że z naszych punktów danych `n_samples` wielokrotnie wybieramy losowo próbkę bez jej usuwania z zestawu (co oznacza, że ta sama próbka może być pobierana wiele razy), `n_samples` razy. Spowoduje to utworzenie zestawu danych, który jest tak duży jak oryginalny zestaw danych, ale niektórych punktów danych w nim będzie brakować (około jednej trzeciej), a niektóre zostaną powtórzone.

Aby to zilustrować, powiedzmy, że chcemy utworzyć próbkę bootstrap z listy `["a", "b", "c", "d"]`. Możliwym przykładem ładowania początkowego byłoby `['b', 'd', 'd', 'c']`. Inną możliwą próbką byłoby `["d", "a", "d", "a"]`.

Następnie na podstawie nowo utworzonego zestawu danych budowane jest drzewo decyzyjne. Jednak algorytm drzewa decyzyjnego, który opisaliśmy, jest nieco zmodyfikowany. Zamiast szukać najlepszego testu dla poszczególnych węzłów, w każdym z nich algorytm losowo wybiera podzbiór cech i szuka najlepszego możliwego testu, który obejmuje jedną z nich. Liczba wybranych cech jest kontrolowana przez parametr `max_features`. Ten wybór podzbioru cech jest powtarzany oddzielnie w każdym węźle, tak że każdy węzeł w drzewie może podjąć decyzję z wykorzystaniem innego podzbioru cech.

Próbkowanie metodą bootstrap prowadzi do tego, że każde drzewo decyzyjne w lesie losowym jest budowane na nieco innym zestawie danych. Ze względu na wybór cech w poszczególnym węźle, każdy podział w każdym drzewie działa na innym ich podzbiorze. Razem te dwa mechanizmy dają pewność, że wszystkie drzewa w lesie losowym są różne.

Krytycznym parametrem w tym procesie jest `max_features`. Jeśli ustawimy wartość `max_features` na taką samą jak `n_features`, to każdy podział będzie mógł przejrzeć wszystkie cechy w zestawie danych, a do wyboru funkcji losowość nie zostanie wprowadzona (jednak pozostaje losowość wynikająca z metody bootstrap). Jeśli ustawimy `max_features` na 1, to podziały nie będą miały żadnego wyboru, którą cechę testować, i mogą przeszukiwać tylko różne progi dla wybranej losowo cechy. Dlatego ustawienie wysokiej wartości parametru `max_features` oznacza, że drzewa w lesie losowym będą dość podobne i będą łatwo dopasowywać się do danych przy użyciu najbardziej charakterystycznych cech. Niska wartość parametru `max_features` oznacza, że drzewa w lesie losowym będą zupełnie inne i że każde z nich będzie musiało być bardzo głębokie, aby dobrze dopasować się do danych.

Aby prognozować na podstawie lasu losowego, algorytm najpierw wykonuje prognozy dla każdego drzewa w lesie. Przy regresji, aby otrzymać ostateczną prognozę, możemy uśrednić te wyniki. W przypadku klasyfikacji stosuje się strategię „miękkiego głosowania". Oznacza to, że każdy algorytm wykonuje „miękką" prognozę, podając prawdopodobieństwo dla każdej możliwej etykiety wyjściowej. Prawdopodobieństwa prognozowane przez wszystkie drzewa są uśredniane, a wynikiem prognozowania jest klasa o najwyższym prawdopodobieństwie.

Analiza lasów losowych. Zastosujmy las losowy składający się z pięciu drzew do zestawu danych `two_moons`, który badaliśmy wcześniej:

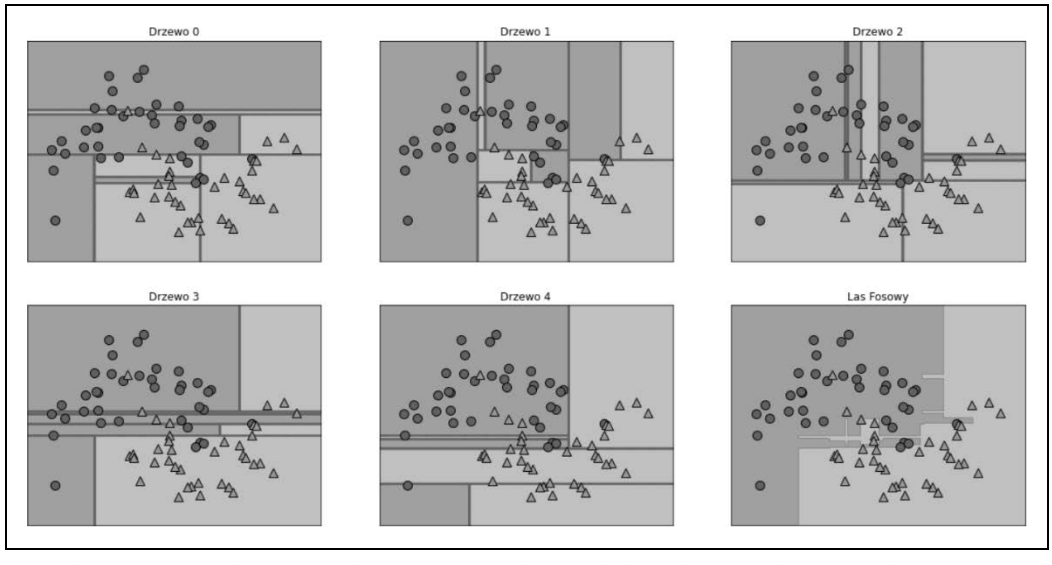
In[68]:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=100, noise=0.25, random_state=3)
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, random_state=42)
forest = RandomForestClassifier(n_estimators=5, random_state=2)
forest.fit(X_train, y_train)
```

Drzewa zbudowane jako część lasu losowego są przechowywane w atrybucie `estimator_`. Wizualizację granic decyzyjnych, których nauczyło się każde drzewo, wraz z ich zagregowaną prognozą wykonaną przez las przedstawiono na rysunku 2.33:

In[69]:

```
fig, axes = plt.subplots(2, 3, figsize=(20, 10))
for i, (ax, tree) in enumerate(zip(axes.ravel(), forest.estimators_)):
ax.set_title("Drzewo {}".format(i))
mglearn.plots.plot_tree_partition(X_train, y_train, tree, ax=ax)
mglearn.plots.plot_2d_separator(forest, X_train, fill=True, ax=axes[-1, -1], alpha=.4)
axes[-1, -1].set_title("Las Fosowy")
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
```



Rysunek 2.33. Granice decyzyjne znalezione przez pięć losowych drzew decyzyjnych i granica decyzyjna uzyskana przez uśrednienie ich prognozowanych prawdopodobieństw

Wyraźnie widać, że granice decyzyjne wyuczone przez pięć drzew są całkiem różne. Ponieważ ze względu na próbkowanie metodą bootstrap niektóre z wykreślonych tutaj punktów uczących nie zostały uwzględnione w zestawach uczących drzew, każde z nich popełnia błędy.

Las losowy nadmiernie dopasowuje się w mniejszym stopniu niż jakiekolwiek drzewo z osobna i zapewnia znacznie bardziej intuicyjne granice decyzyjne. W każdej prawdziwej aplikacji użylibyśmy znacznie więcej drzew (często setek lub tysięcy), co doprowadziłoby do jeszcze gładszych granic.

Jako inny przykład zastosujmy las losowy, który składa się ze 100 drzew w zestawie danych Breast Cancer:

In[70]:

```
X_train, X_test, y_train, y_test = train_test_split(
cancer.data, cancer.target, random_state=0)
forest = RandomForestClassifier(n_estimators=100, random_state=0)
forest.fit(X_train, y_train)
print("Dokładność w danych uczących: {:.3f}".format(forest.score(X_train, y_train)))
print("Dokładność w danych testowych: {:.3f}".format(forest.score(X_test, y_test)))
```

Out[70]:

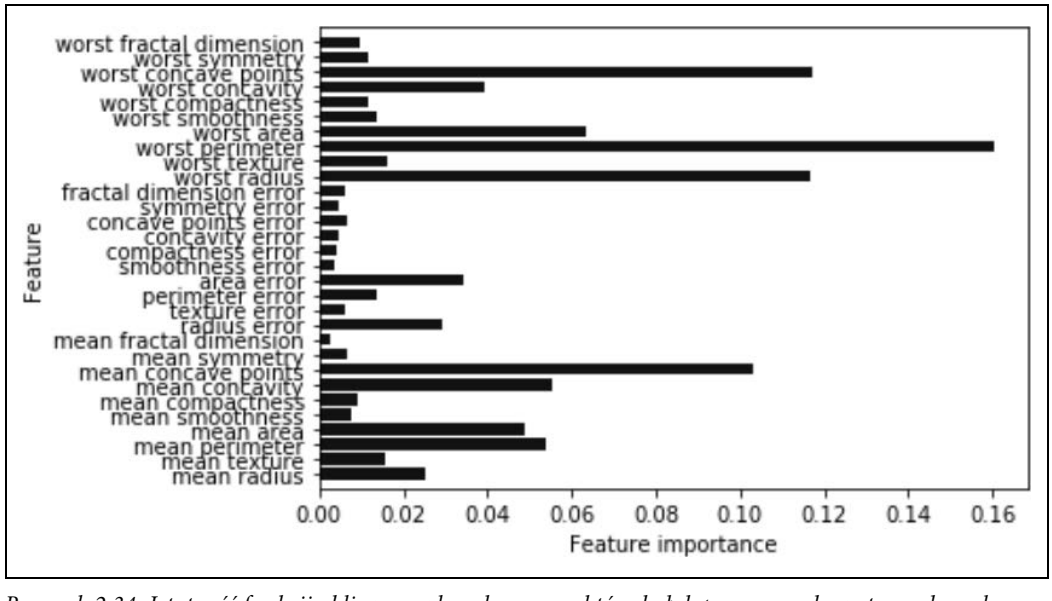
```
Dokładność w danych uczących: 1.000
Dokładność w danych testowych: 0.972
```

Las losowy daje nam dokładność 97%, czyli lepszą niż modele liniowe lub pojedyncze drzewo decyzyjne, i to bez dostrajania żadnych parametrów. Mogliśmy dostosować ustawienie `max_features` lub zastosować wstępne przycinanie, tak jak zrobiliśmy to dla pojedynczego drzewa decyzyjnego. Jednak często domyślne parametry lasu losowego działają już całkiem dobrze.

Podobnie jak w przypadku drzewa decyzyjnego, las losowy dostarcza wartości ważności cech, które są obliczane poprzez agregację ważności cech na drzewach w lesie. Zwykle ważność cech zapewniana przez las losowy jest bardziej wiarygodna niż wartości zapewniane przez pojedyncze drzewa. Przyjrzyjmy się rysunkowi 2.34:

In[71]:

```
plot_feature_importances_cancer(forest)
```



Rysunek 2.34. Istotność funkcji obliczona z lasu losowego, który był dopasowany do zestawu danych Breast Cancer

Jak widać, las losowy nadaje niezerowe znaczenie zdecydowanie większej liczbie cech niż pojedyncze drzewo. Podobnie jak w przypadku pojedynczego drzewa decyzyjnego, las losowy również przywiązuje dużą wagę do „worst radius", ale wybiera „worst perimeter" jako cechę, która wnosi najwięcej informacji. Losowość w budowaniu lasu losowego zmusza algorytm do rozważenia wielu możliwych wyjaśnień, w wyniku czego las losowy przedstawia znacznie szerszy obraz danych niż pojedyncze drzewo.

Mocne i słabe strony oraz parametry. Lasy losowe do regresji i klasyfikacji należą obecnie do najczęściej stosowanych metod uczenia maszynowego. Są bardzo wydajne, zwykle działają dobrze bez intensywnego dostrajania parametrów i nie wymagają skalowania danych.

Zasadniczo lasy losowe mają wszystkie zalety drzew decyzyjnych, a przy tym nadrabiają niektóre ich braki. Sytuacja, w której potrzebna jest zwarta reprezentacja procesu decyzyjnego, to jedyna, w której nadal opłaca się używać drzew decyzyjnych. Zasadniczo niemożliwe jest szczegółowe zinterpretowanie dziesiątek lub setek drzew, a drzewa w lasach losowych są zwykle głębsze niż drzewa decyzyjne (ze względu na wykorzystanie podzbiorów cech). Dlatego jeśli chcesz zwizualizować, jak przebiega prognozowanie, osobom, które nie są ekspertami, pojedyncze drzewo decyzyjne może być lepszym wyborem. Chociaż tworzenie lasów losowych na dużych zestawach danych może być nieco czasochłonne, można je łatwo rozkładać na wiele rdzeni procesora w komputerze. Jeśli używasz procesora wielordzeniowego (który znajduje się w prawie każdym współczesnym komputerze), to aby dostosować liczbę używanych rdzeni, możesz użyć parametru `n_jobs`. Użycie większej liczby rdzeni procesora spowoduje przyspieszenie w sposób liniowy (przy użyciu dwóch rdzeni uczenie lasu losowego będzie dwukrotnie szybsze), ale ustawienie parametru `n_jobs` na wartość większą niż liczba rdzeni nie pomoże. Aby używać wszystkich rdzeni w komputerze, można ustawić `n_jobs=-1`.

Należy pamiętać, że lasy losowe są naturalnie losowe, a ustawienie różnych stanów losowych (lub całkowity brak ustawienia parametru `random_state`) może drastycznie zmienić budowany model. Im więcej drzew jest w lesie, tym będzie on odporniejszy na ustawienie stanu losowego. Jeśli chcesz uzyskać powtarzalne wyniki, ważne jest, aby dobrze ustawić parametr `random_state`.

Lasy losowe zwykle nie radzą sobie dobrze z bardzo wielowymiarowymi, rzadkimi danymi, takimi jak dane tekstowe. W przypadku tego rodzaju danych bardziej odpowiednie mogą być modele liniowe. Lasy losowe zwykle działają dobrze nawet w przypadku bardzo dużych zestawów danych, a uczenie można z łatwością przeprowadzić równolegle na wielu rdzeniach procesora w potężnym komputerze. Jednak lasy losowe wymagają więcej pamięci oraz wolniej się uczą i prognozują niż modele liniowe. Jeśli czas i pamięć są w aplikacji ważne, warto zamiast tego użyć modelu liniowego.

Ważnymi parametrami do dostosowania są `n_estimators`, `max_features` i najprawdopodobniej opcje wstępnego przycinania, takie jak `max_depth`. W przypadku parametru `n_estimators` większa wartość jest zawsze lepsza. Uśrednienie większej liczby drzew dzięki ograniczeniu nadmiernego dopasowania da bardziej wytrzymały zespół modeli. Jednak nie odbywa się to bez strat, więcej drzew potrzebuje więcej pamięci i więcej czasu na uczenie. Powszechną zasadą jest budowanie „tylu, na ile masz czasu i pamięci".

Jak opisano wcześniej, parametr `max_features` określa, jak losowe jest każde drzewo, a mniejsza wartość parametru `max_features` zmniejsza nadmierne dopasowanie. Ogólnie rzecz biorąc,

dobrą zasadą jest używanie wartości domyślnych: `max_features=sqrt(n_features)` do klasyfikacji i `max_features=log2(n_features)` do regresji. Dodanie parametru `max_features` lub `max_leaf_nodes` może czasami poprawić wydajność. Może również drastycznie zmniejszyć wymagania dotyczące pamięci i czasu potrzebnego na uczenie i prognozowanie.

Drzewa regresji ze wzmocnieniem gradientowym (maszyny ze wzmocnieniem gradientowym)

Drzewo regresji ze wzmocnieniem gradientowym to kolejna metoda, która łączy w zespół wiele drzew decyzyjnych w celu stworzenia silniejszego modelu. Pomimo słowa „regresja" w nazwie modeli tych można użyć zarówno do regresji, jak i klasyfikacji. W przeciwieństwie do metody lasów losowych wzmacnianie gradientowe polega na budowaniu drzew w sposób seryjny, w którym każde drzewo stara się poprawić błędy poprzedniego. Domyślnie w drzewach regresji wzmocnionej gradientem nie ma losowości; zamiast tego stosowane jest silne przycinanie wstępne. Drzewa wzmocnione gradientem często używają bardzo płytkich drzew, o głębokości od 1 do 5, co powoduje, że model jest mniejszy pod względem pamięci i przyspiesza wykonywanie prognoz.

Główną ideą wzmocnienia gradientowego jest połączenie wielu prostych modeli (w tym kontekście nazywanych *słabymi uczniami*), takich jak płytkie drzewa. Każde drzewo może zapewnić dobrą jakość prognoz tylko dla części danych, dlatego aby iteracyjnie poprawiać wydajność, wciąż dodawane są kolejne drzewa.

Drzewa wzmocnione gradientem są często używane w zwycięskich pracach w konkursach uczenia maszynowego i szeroko stosowane w przemyśle. Generalnie wykazują nieco większą wrażliwość na ustawienia parametrów niż lasy losowe, ale mogą zapewnić lepszą dokładność, jeśli parametry są ustawione poprawnie.

Oprócz wstępnego przycinania i liczby drzew w zespole innym ważnym parametrem wzmocnienia gradientu jest parametr `learning_rate`, który kontroluje, jak bardzo każde drzewo próbuje poprawić błędy poprzednich drzew. Wyższy wskaźnik uczenia się (ang. *learning rate*) oznacza, że każde drzewo może wprowadzać silniejsze poprawki, co pozwala na bardziej złożone modele. Aby zwiększyć złożoność modelu, można ustawić większą wartość parametru `n_estimators`, co spowoduje dodanie większej liczby drzew do zespołu, a tym samym większą szansę na poprawienie błędów w zestawie danych uczących.

Poniżej przedstawiono przykład użycia modelu `GradientBoostingClassifier` w zestawie danych Breast Cancer. Domyślnie używanych jest 100 drzew o maksymalnej głębokości 3 i współczynniku uczenia się wynoszącym 0,1:

`In[72]:`

```
from sklearn.ensemble import GradientBoostingClassifier
X_train, X_test, y_train, y_test = train_test_split(
cancer.data, cancer.target, random_state=0)
gbrt = GradientBoostingClassifier(random_state=0)
gbrt.fit(X_train, y_train)
print("Dokładność w danych uczących: {:.3f}".format(gbrt.score(X_train, y_train)))
print("Dokładność w danych testowych: {:.3f}".format(gbrt.score(X_test, y_test)))
```

`Out[72]:`

```
Dokładność w danych uczących: 1.000
Dokładność w danych testowych: 0.958
```

Ponieważ dokładność zestawu uczącego wynosi 100%, prawdopodobnie model będzie nadmiernie dopasowany. Aby zmniejszyć nadmierne dopasowanie, moglibyśmy zastosować silniejsze przycinanie wstępne, ograniczając maksymalną głębokość, lub obniżyć szybkość uczenia:

In[73]:

```
gbrt = GradientBoostingClassifier(random_state=0, max_depth=1)
gbrt.fit(X_train, y_train)
print("Dokładność w danych uczących: {:.3f}".format(gbrt.score(X_train, y_train)))
print("Dokładność w danych testowych: {:.3f}".format(gbrt.score(X_test, y_test)))
```

Out[73]:

```
Dokładność w danych uczących: 0.991
Dokładność w danych testowych: 0.972
```

In[74]:

```
gbrt = GradientBoostingClassifier(random_state=0, learning_rate=0.01)
gbrt.fit(X_train, y_train)
print("Dokładność w danych uczących: {:.3f}".format(gbrt.score(X_train, y_train)))
print("Dokładność w danych testowych: {:.3f}".format(gbrt.score(X_test, y_test)))
```

Out[74]:

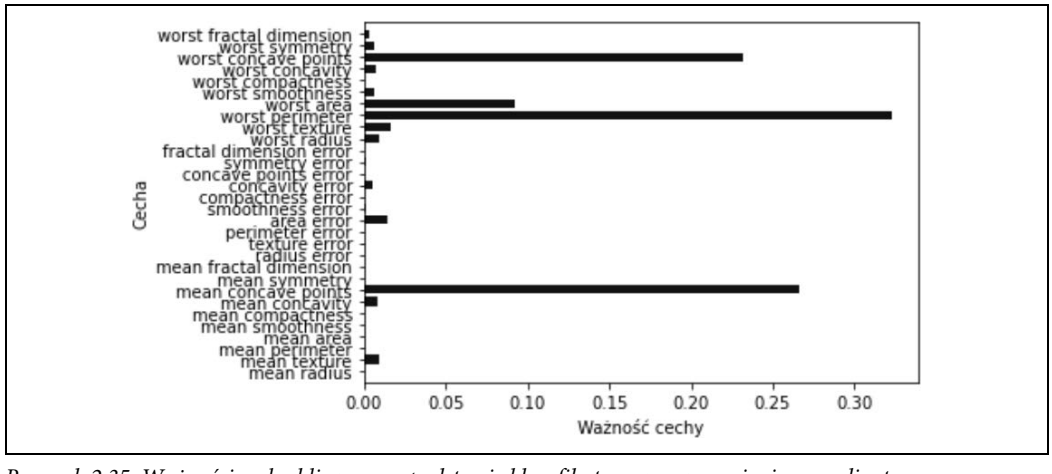
```
Dokładność w danych uczących: 0.988
Dokładność w danych testowych: 0.965
```

Zgodnie z oczekiwaniami obie metody zmniejszania złożoności modelu zmniejszyły dokładność w danych z zestawu uczącego. W tym przypadku obniżenie maksymalnej głębokości drzew zapewniło istotną poprawę modelu, natomiast obniżenie szybkości uczenia się tylko nieznacznie zwiększyło wydajność uogólniania.

Jeśli chodzi o inne modele oparte na drzewie decyzyjnym, to aby uzyskać lepszy wgląd w nasz model, możemy ponownie zwizualizować znaczenie cech, jak przedstawiono na rysunku 2.35. Ponieważ użyliśmy 100 drzew, sprawdzenie ich wszystkich jest niepraktyczne, nawet jeśli wszystkie mają głębokość 1:

In[75]:

```
gbrt = GradientBoostingClassifier(random_state=0, max_depth=1)
gbrt.fit(X_train, y_train)
plot_feature_importances_cancer(gbrt)
```



Rysunek 2.35. Ważności cech obliczone na podstawie klasyfikatora ze wzmocnieniem gradientu, dopasowanego do zestawu danych Breast Cancer

Możemy zauważyć, że znaczenie cech drzew ze wzmocnionym gradientem jest nieco podobne do ważności cech lasów losowych, chociaż wzmocnienie gradientu całkowicie zignorowało niektóre cechy.

Ponieważ zarówno wzmocnienie gradientu, jak i lasy losowe działają dobrze na podobnych rodzajach danych, powszechnym podejściem jest w pierwszej kolejności wypróbowanie lasów losowych, które działają dość solidnie. Jeśli lasy losowe działają dobrze, ale liczy się czas prognozowania lub ważne jest, aby wycisnąć jak najwięcej dokładności z modelu uczenia maszynowego, to w takiej sytuacji często może pomóc przejście na model ze wzmocnieniem gradientu.

Jeśli wzmocnienie gradientu ma być zastosowane do problemu ze znaczną ilością danych, warto przyjrzeć się bibliotece `xgboost` i jej interfejsowi w języku Python, która w momencie pisania na wielu zestawach danych jest szybsza (i czasami łatwiejsza do dostrojenia) niż implementacja modelu ze wzmocnieniem gradientowym dostępna w bibliotece `scikit-learn`.

Mocne i słabe strony oraz parametry. Drzewa decyzyjne ze wzmocnieniem gradientowym należą do najpotężniejszych i najpowszechniej stosowanych modeli nadzorowanego uczenia maszynowego. Ich główną wadą jest to, że wymagają starannego dostrojenia parametrów oraz że uczenie może zająć dużo czasu. Podobnie jak w przypadku innych modeli opartych na drzewach, algorytm działa dobrze bez skalowania i na połączeniu cech binarnych i ciągłych. Tak jak w przypadku innych modeli opartych na drzewach, często nie działa dobrze przy rozrzedzonych danych wielowymiarowych.

Głównymi parametrami modeli drzew ze wzmocnieniem gradientowym są liczba drzew, ustawiana wartością parametru `n_estimators`, i parametr `learning_rate`, którego wartość kontroluje stopień, w jakim każde drzewo może korygować błędy poprzednich drzew. Te dwa parametry są ze sobą silnie powiązane, ponieważ niższa wartość współczynnika `learning_rate` oznacza, że do zbudowania modelu o podobnej złożoności potrzeba więcej drzew. W przeciwieństwie do lasów losowych, gdy wyższa wartość `n_estimators` zawsze jest lepsza, zwiększenie wartości parametru `n_estimators` w modelu ze wzmacnianiem gradientowym prowadzi do bardziej złożonego modelu, co może skutkować nadmiernym dopasowaniem. Powszechną praktyką jest dopasowywanie parametru `n_estimators` w zależności od czasu i ilości pamięci, a następnie wypróbowywanie różnych wartości parametru `learning_rate`.

Innym ważnym parametrem, który służy do zmniejszenia złożoności każdego drzewa, jest `max_depth` (lub alternatywnie `max_leaf_nodes`). W przypadku modeli ze wzmocnieniem gradientowym zwykle ustawia się bardzo małą wartość parametru `max_depth`, często nie większą niż 5.

Maszyny wektorów nośnych

Kolejnym typem nadzorowanych modeli, które omówimy, są maszyny wektorów nośnych z jądrem. W podrozdziale „Modele liniowe do klasyfikacji” na stronie 59 do klasyfikacji użyliśmy maszyn liniowych wektorów nośnych. Kernelizowane maszyny wektorów nośnych (często nazywane po prostu SVM) to rozszerzenie pozwalające na bardziej złożone modele, które nie są zdefiniowane

w prosty sposób przez hiperpłaszczyzny w przestrzeni danych wejściowych. Chociaż istnieją maszyny wektorów nośnych do klasyfikacji i do regresji, ograniczymy się do przypadku klasyfikacji, który zaimplementowano w modelu SVC. Podobnie w modelu SVR zaimplementowano koncepcje, które dotyczą obsługi regresji wektorowej.

Matematyka stojąca za maszynami wektorów nośnych jest dość skomplikowana i wykracza poza zakres tej książki. Szczegóły można znaleźć w pierwszym rozdziale książki Hastiego, Tibshiraniego i Friedmana — *The Elements of Statistical Learning* (*http://statweb.stanford.edu/~tibs/ElemStatLearn/*). Postaramy się jednak przybliżyć ideę tej metody.

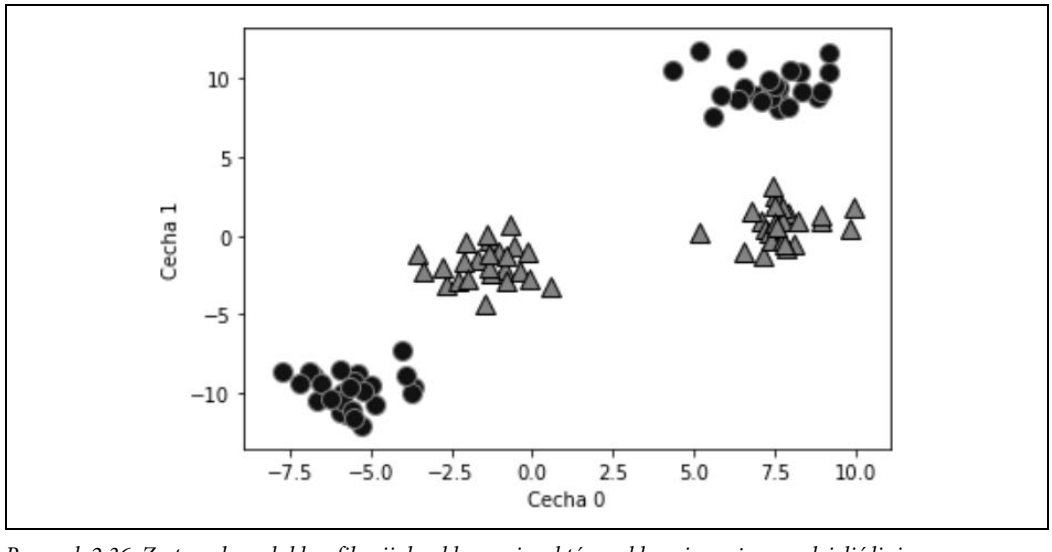
Modele liniowe i cechy nieliniowe

Jak widać na rysunku 2.15, ponieważ linie i hiperpłaszczyzny mają ograniczoną elastyczność, modele liniowe mogą być dość ograniczone w przestrzeniach o małych wymiarach. Jednym ze sposobów uelastycznienia modelu liniowego jest dodanie większej liczby elementów — np. poprzez dodanie interakcji lub wielomianów elementów wejściowych.

Przyjrzyjmy się syntetycznemu zestawowi danych (rysunek 2.36), którego użyliśmy w podrozdziale „Ważność cech w drzewach” na stronie 76, co przedstawiono na rysunku 2.29:

In[76]:

```
X, y = make_blobs(centers=4, random_state=8)
y = y % 2
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Cecha 0")
plt.ylabel("Cecha 1")
```

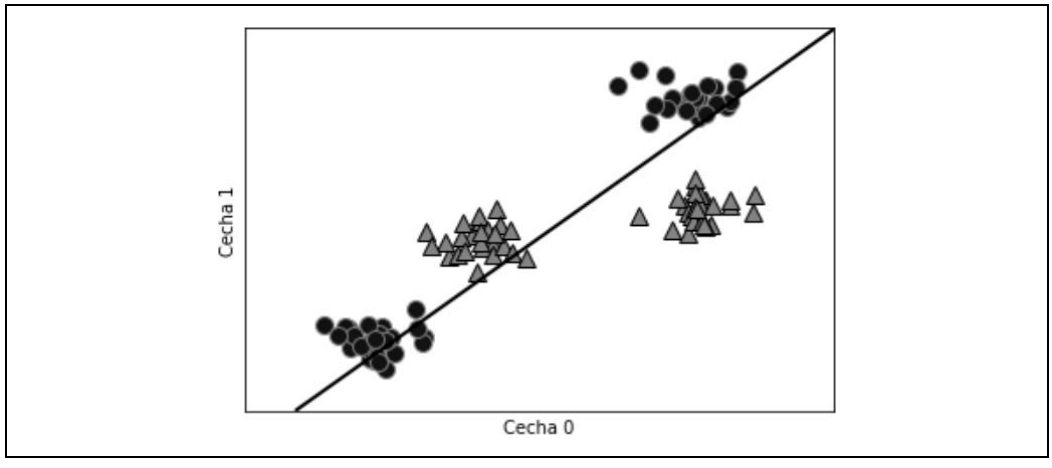


Rysunek 2.36. Zestaw danych klasyfikacji dwuklasowej, w którym klas nie można rozdzielić liniowo

Model liniowy do klasyfikacji może oddzielać punkty tylko za pomocą linii i nie będzie w stanie dokonać poprawnej klasyfikacji na zestawie danych, który przedstawiono na rysunku 2.37:

In[77]:

```
from sklearn.svm import LinearSVC
linear_svm = LinearSVC().fit(X, y)
mglearn.plots.plot_2d_separator(linear_svm, X)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Cecha 0")
plt.ylabel("Cecha 1")
```



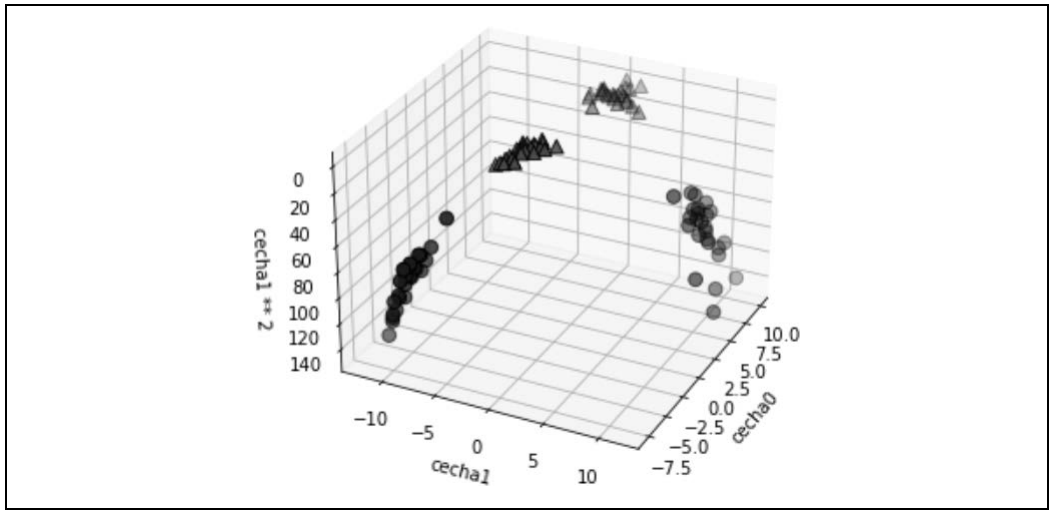
Rysunek 2.37. Granica decyzyjna znaleziona przez liniowy model SVM

Rozwińmy zestaw cech wejściowych, powiedzmy, przez dodanie cechy `cecha1 ** 2`, czyli kwadratu drugiej cechy jako nowej cechy. Zamiast przedstawiać każdy punkt danych jako punkt dwuwymiarowy (`cecha0`, `cecha1`), przedstawiamy go teraz jako punkt trójwymiarowy (feature0, feature1, feature1 ** 2)[10]. Tę nową reprezentację zilustrowano na trójwymiarowym wykresie punktowym (rysunek 2.38):

In[78]:

```
# dodaj pierwszą cechę podniesioną do kwadratu
X_new = np.hstack([X, X[:, 1:] ** 2])
from mpl_toolkits.mplot3d import Axes3D, axes3d
figure = plt.figure()
# zwizualizuj w 3D
ax = Axes3D(figure, elev=-152, azim=-26)
# umieść na wykresie wszystkie punkty z y = 0, a następnie wszystkie z y = 1
mask = y == 0
ax.scatter(X_new[mask, 0], X_new[mask, 1], X_new[mask, 2], c='b',
           cmap=mglearn.cm2, s=60, edgecolor='k')
ax.scatter(X_new[~mask, 0], X_new[~mask, 1], X_new[~mask, 2], c='r', marker='^',
➥cmap=mglearn.cm2, s=60, edgecolor='k')
ax.set_xlabel("cecha0")
ax.set_ylabel("cecha1")
ax.set_zlabel("cecha1 ** 2")
```

[10] Wybraliśmy tę konkretną cechę do dodania w celach demonstracyjnych. Wybór nie jest szczególnie ważny.



Rysunek 2.38. Rozszerzenie zestawu danych, który pokazano na rysunku 2.37, utworzone przez dodanie trzeciej cechy, wyprowadzonej z cechy1

W powyższej, nowej reprezentacji danych jest teraz możliwe rozdzielenie dwóch klas za pomocą modelu liniowego, płaszczyzny w trzech wymiarach. Możemy to potwierdzić, dopasowując model liniowy do rozszerzonych danych, jak przedstawiono na rysunku 2.39:

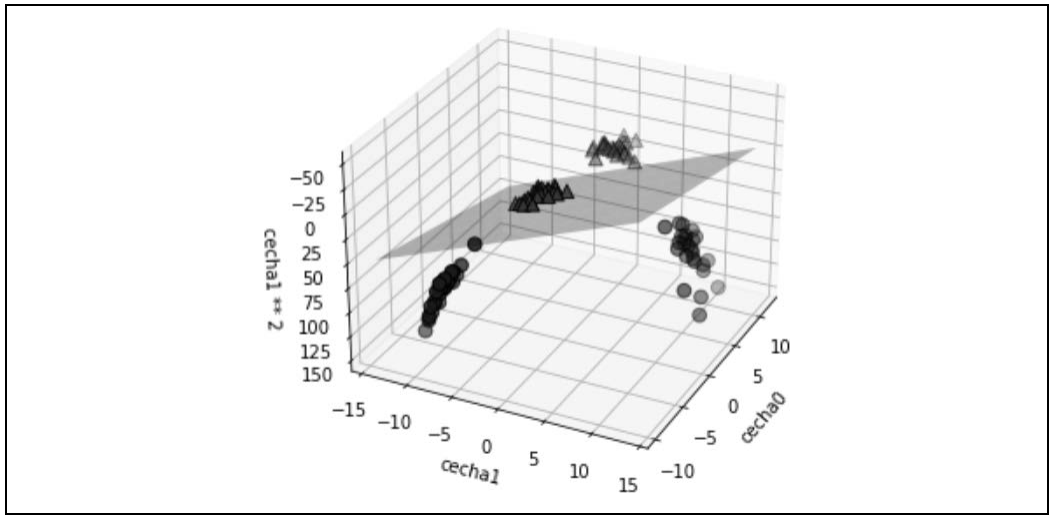
In[79]:

```
linear_svm_3d = LinearSVC().fit(X_new, y)
coef, intercept = linear_svm_3d.coef_.ravel(), linear_svm_3d.intercept_
# pokaż liniową granicę decyzyjną
figure = plt.figure()
ax = Axes3D(figure, elev=-152, azim=-26)
xx = np.linspace(X_new[:, 0].min() - 2, X_new[:, 0].max() + 2, 50)
yy = np.linspace(X_new[:, 1].min() - 2, X_new[:, 1].max() + 2, 50)
XX, YY = np.meshgrid(xx, yy)
ZZ = (coef[0] * XX + coef[1] * YY + intercept) / -coef[2]
ax.plot_surface(XX, YY, ZZ, rstride=8, cstride=8, alpha=0.3)
ax.scatter(X_new[mask, 0], X_new[mask, 1], X_new[mask, 2], c='b', cmap=mglearn.cm2, s=60)
ax.scatter(X_new[~mask, 0], X_new[~mask, 1], X_new[~mask, 2], c='r', marker='^',
➥cmap=mglearn.cm2, s=60)
ax.set_xlabel("cecha0")
ax.set_ylabel("cecha1")
ax.set_zlabel("cecha0 ** 2")
```

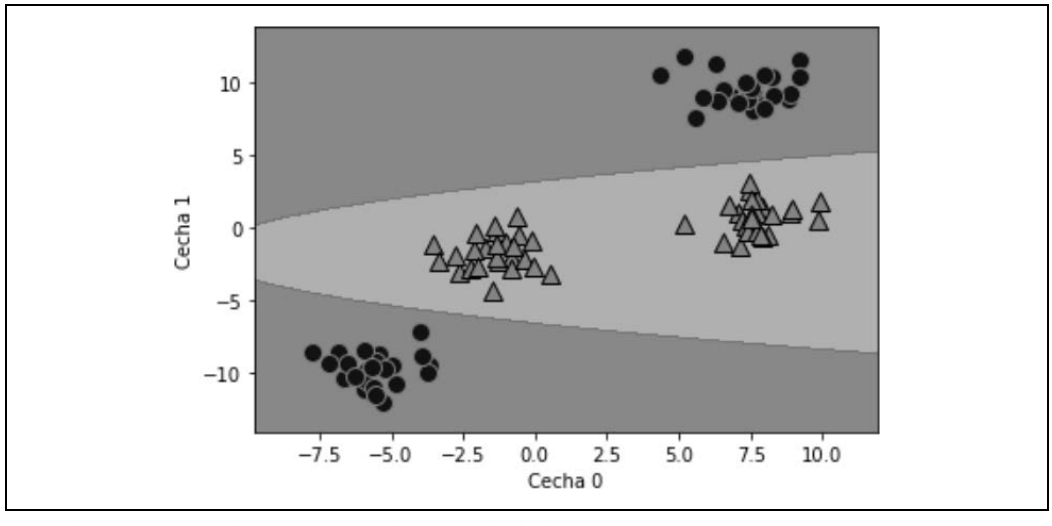
Jako funkcja oryginalnych cech liniowy model SVM nie jest już liniowy. To nie jest linia, raczej elipsa, jak widać na wykresie, który przedstawiono na rysunku 2.40:

In[80]:

```
ZZ = YY ** 2
dec = linear_svm_3d.decision_function(np.c_[XX.ravel(), YY.ravel(), ZZ.ravel()])
plt.contourf(XX, YY, dec.reshape(XX.shape), levels=[dec.min(), 0, dec.max()],
➥cmap=mglearn.cm2, alpha=0.5)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Cecha 0")
plt.ylabel("Cecha 1")
```



Rysunek 2.39. Granica decyzyjna znaleziona przez liniowy SVM w rozszerzonym trójwymiarowym zestawie danych



Rysunek 2.40. Granica decyzyjna z rysunku 2.39 jako funkcja dwóch oryginalnych cech

Kernel trick

Dowiedzieliśmy się, że dodanie nieliniowych cech do reprezentacji naszych danych może znacznie zwiększyć wydajność modeli liniowych. Jednak często nie wiemy, które cechy warto dodać, a dodanie wielu cech (takich jak wszystkie możliwe interakcje w 100-wymiarowej przestrzeni cech) może spowodować, że obliczenia będą bardzo kosztowne. Na szczęście istnieje sprytna sztuczka matematyczna, która pozwala uczyć klasyfikator w większej przestrzeni bez faktycznego obliczania nowej, prawdopodobnie bardzo dużej reprezentacji cech. Jest to znane jako *sztuczka jądra* (ang. *kernel trick*) i polega na bezpośrednim obliczeniu odległości (dokładniej iloczynu skalarnego) punktów danych dla rozszerzonej reprezentacji cech, bez faktycznego obliczania rozszerzenia.

Istnieją dwa sposoby odwzorowania danych w przestrzeni o wyższym wymiarze, powszechnie używanej z maszynami wektorów pomocniczych: jądro wielomianowe, które oblicza wszystkie możliwe wielomiany do pewnego stopnia pierwotnych cech (takich jak `cecha1 ** 2 * cecha2 ** 5`), oraz jądro radialnej funkcji bazowej (RBF — ang. *radial basis function*), znane również jako jądro Gaussa. Jest ono nieco trudniejsze do wyjaśnienia, ponieważ odpowiada przestrzeni cech w nieskończonych wymiarach. Jednym ze sposobów wyjaśnienia jądra Gaussa jest to, że bierze ono pod uwagę wszystkie możliwe wielomiany wszystkich stopni, ale znaczenie cech maleje wraz z wyższymi stopniami[11].

W praktyce jednak szczegóły matematyczne stojące za modelem SVM jądra nie są tak ważne i to, jak SVM z jądrem RBF podejmuje decyzję, można podsumować dość łatwo — zrobimy to w następnym podrozdziale.

SVM

Aby zaprezentować granicę decyzyjną między dwiema klasami, maszyna SVM uczy się, jak ważny jest każdy z punktów danych uczących. Zwykle dla określenia granicy decyzyjnej ma znaczenie tylko podzbiór punktów uczących: tych, które leżą na granicy między klasami. Nazywane są one *wektorami nośnymi* i to od nich bierze się nazwa maszyny wektorów nośnych.

Aby prognozować nowy punkt, mierzona jest odległość do każdego z wektorów nośnych. Decyzja o klasyfikacji jest podejmowana na podstawie odległości do wektora nośnego i ważności wektorów nośnych, które zostały wyuczone (przechowywane w atrybucie `dual_coef_` obiektu `SVC`).

Odległość między punktami danych jest mierzona przez jądro Gaussa:

$$k_{rbf}(x_1, x_2) = \exp(\gamma \|x_1 - x_2\|^2)$$

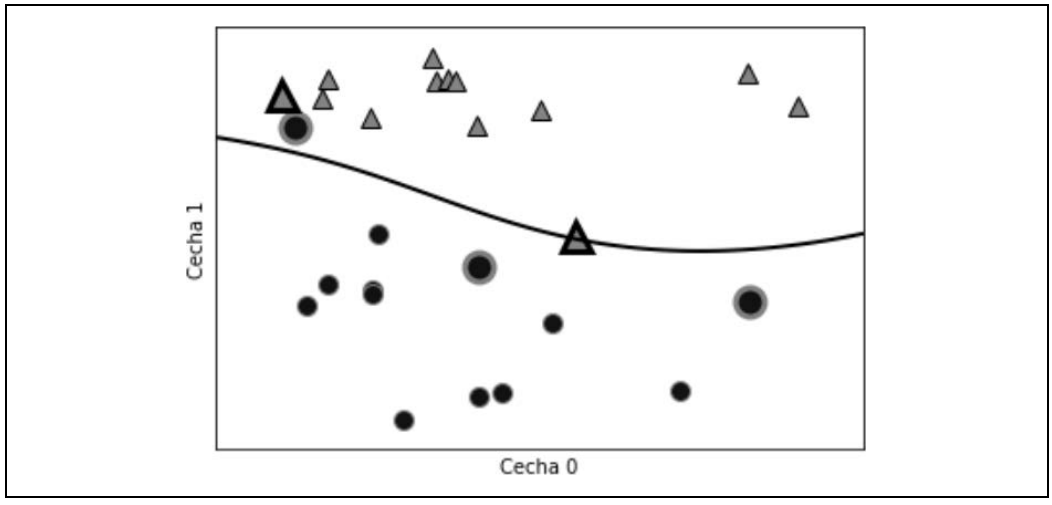
x_1 i x_2 to punkty danych, $\|x_1 - x_2\|$ oznacza odległość euklidesową, a γ (gamma) to parametr, który kontroluje szerokość jądra Gaussa.

Na rysunku 2.41 przedstawiono wynik uczenia maszyny wektorów nośnych na dwuwymiarowym, dwuklasowym zestawie danych. Granicę decyzyjną zaznaczono na czarno, a wektory nośne to większe punkty z szerokim konturem. Kod przedstawiony w poniższym listingu służy do wyuczenia maszyny SVM na zestawie danych `forge` i utworzenia wykresu, który pokazano na rysunku 2.41:

In[81]:

```
from sklearn.svm import SVC
X, y = mglearn.tools.make_handcrafted_dataset()
svm = SVC(kernel='rbf', C=10, gamma=0.1).fit(X, y)
mglearn.plots.plot_2d_separator(svm, X, eps=.5)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
# stwórz wykres wektorów pomocniczych
sv = svm.support_vectors_
# etykiety klas wektorów nośnych są podane za pomocą podwójnych współczynników
sv_labels = svm.dual_coef_.ravel() > 0
mglearn.discrete_scatter(sv[:, 0], sv[:, 1], sv_labels, s=15, markeredgewidth=3)
plt.xlabel("Cecha 0")
plt.ylabel("Cecha 1")
```

[11] Wynika to z rozszerzenia mapy wykładniczej Taylora.



Rysunek 2.41. Granica decyzyjna i wektory nośne znalezione przez SVM z jądrem RBF

W tym przypadku SVM daje bardzo gładką i nieliniową (nieprostą) granicę. Dostosowaliśmy tutaj dwa parametry: parametr `C` i parametr `gamma`, które szczegółowo omówimy poniżej.

Dostrajanie parametrów SVM

Parametr `gamma`, który podano we wzorze przedstawionym w poprzedniej sekcji, kontroluje szerokość jądra Gaussa. Określa skalę tego, co oznacza, że punkty są blisko siebie. Parametr `C` jest parametrem regularyzacji, podobnym do używanego w modelach liniowych. Ogranicza znaczenie każdego punktu (a dokładniej ich współczynnika `dual_coef_`).

Przyjrzyjmy się, co się stanie, gdy zmienimy te parametry (rysunek 2.42):

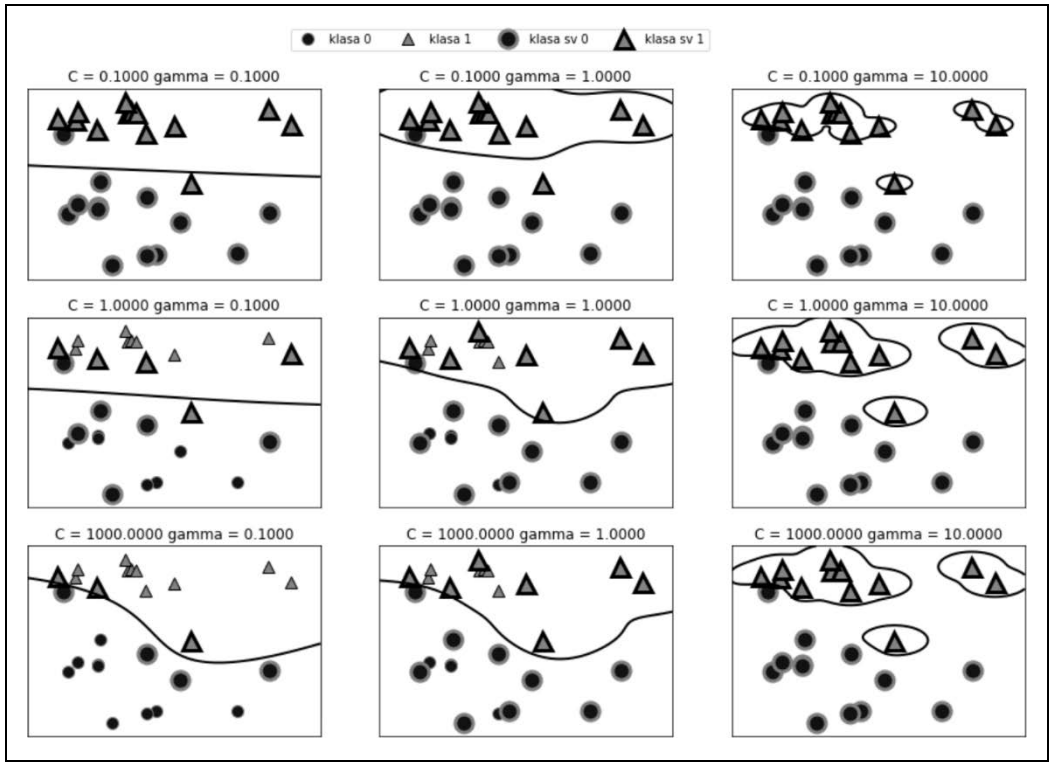
`In[82]:`

```
fig, axes = plt.subplots(3, 3, figsize=(15, 10))

for ax, C in zip(axes, [-1, 0, 3]):
    for a, gamma in zip(ax, range(-1, 2)):
        mglearn.plots.plot_svm(log_C=C, log_gamma=gamma, ax=a)

axes[0, 0].legend(["klasa 0", "klasa 1", "klasa sv 0", "klasa sv 1"],
                  ncol=4, loc=(.9, 1.2))
```

Kiedy przyjrzymy się rysunkowi od lewej do prawej, możemy zauważyć zwiększenie wartości parametru `gamma` od `0,1` do 10. Mała wartość parametru `gamma` oznacza duży promień dla jądra Gaussa, co z kolei oznacza, że uznaje się wiele punktów za leżące blisko siebie. Znajduje to odzwierciedlenie w bardzo płynnych granicach decyzyjnych po lewej stronie i granicach, które koncentrują się bardziej na pojedynczych punktach, po prawej stronie rysunku. Niska wartość parametru `gamma` mówi, że granica decyzyjna będzie się zmieniać powoli, co oznacza, że model będzie miał małą złożoność, a wysoka wartość parametru `gamma` poskutkuje utworzeniem bardziej złożonego modelu.



Rysunek 2.42. Granice decyzyjne i wektory pomocnicze dla różnych ustawień parametrów C i gamma

Jeśli obejrzeć rysunek od góry do dołu, widać zwiększenie wartości parametru `C` z `0,1` do `1000`. Podobnie jak w przypadku modeli liniowych, mała wartość parametru `C` oznacza bardzo ograniczony model, w którym każdy punkt danych może mieć mocno ograniczony wpływ. Widać, że w lewym górnym rogu granica decyzyjna wygląda prawie liniowo, a błędnie sklasyfikowane punkty nie mają prawie żadnego wpływu na linię. Jak pokazano w prawym dolnym rogu rysunku, zwiększenie wartości parametru `C` pozwala tym punktom mieć silniejszy wpływ na model i powoduje zakrzywienie granicy decyzji, co umożliwia ich poprawne sklasyfikowanie.

Zastosujmy SVM z jądrem RBF do zestawu danych Breast Cancer. Domyślnie `C=1`, a `gamma=1/n_features`:

`In[83]:`

```
X_train, X_test, y_train, y_test = train_test_split(
cancer.data, cancer.target, random_state=0)
svc = SVC()
svc.fit(X_train, y_train)
print("Dokładność w danych uczących: {:.2f}".format(svc.score(X_train, y_train)))
print("Dokładność w danych testowych: {:.2f}".format(svc.score(X_test, y_test)))
```

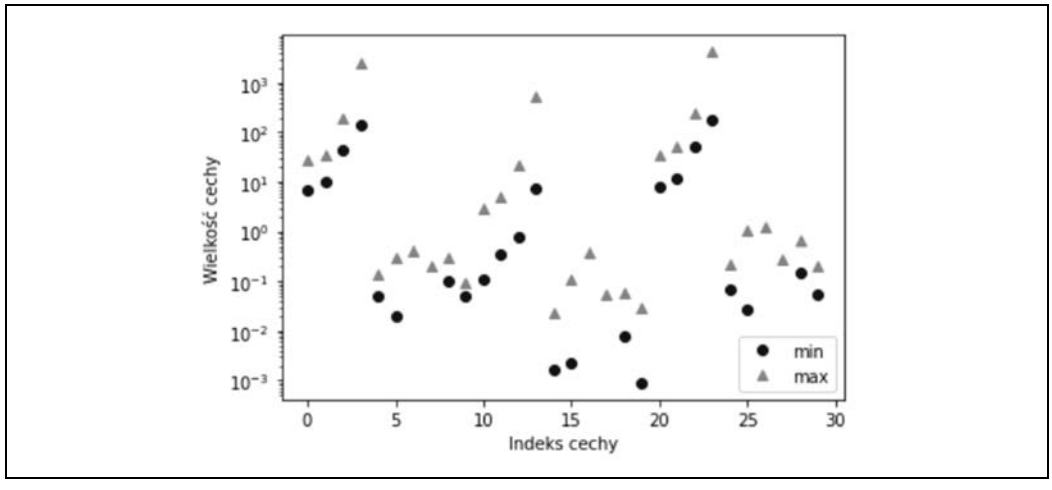
`Out[83]:`

```
Dokładność w danych uczących: 1.00
Dokładność w danych testowych: 0.63
```

Model jest nadmiernie dopasowany w znacznym stopniu, ma doskonały wynik dla zestawu uczącego i tylko 63% dokładności dla zestawu testowego. Chociaż maszyny SVM często działają całkiem dobrze, są bardzo wrażliwe na ustawienia parametrów i skalowanie danych. W szczególności wymagają, aby wszystkie cechy różniły się w podobnej skali. Spójrzmy na minimalne i maksymalne wartości dla każdej cechy, wykreślone w przestrzeni logarytmicznej, co przedstawiono na rysunku 2.43:

In[84]:

```
plt.plot(X_train.min(axis=0), 'o', label="min")
plt.plot(X_train.max(axis=0), '^', label="max")
plt.legend(loc=4)
plt.xlabel("Indeks cechy")
plt.ylabel("Wielkość cechy")
plt.yscale("log")
```



Rysunek 2.43. Zakresy cech dla zestawu danych Breast Cancer (zwróćmy uwagę, że oś y jest w skali logarytmicznej)

Na podstawie tego wykresu możemy określić, że cechy w zestawie danych Breast Cancer mają zupełnie różne rzędy wielkości. W przypadku innych modeli (np. modeli liniowych) stanowi to nieznaczny problem, ale ma druzgocące skutki dla SVM. Przyjrzyjmy się kilku sposobom rozwiązania tego problemu.

Wstępne przetwarzanie danych dla maszyn SVM

Jednym ze sposobów na rozwiązanie tego problemu jest przeskalowanie każdej cechy tak, aby wszystkie były w przybliżeniu w tej samej skali. Powszechną metodą przeskalowywania maszyn SVM jest skalowanie danych w taki sposób, aby wszystkie funkcje znajdowały się w przedziale od 0 do 1. Więcej szczegółów na temat tego, jak to zrobić, podano przy okazji użycia metody przetwarzania wstępnego `MinMaxScaler`|, opisanej w rozdziale 3. Na razie zróbmy to „ręcznie”:

In[85]:

```
# oblicz minimalną wartość dla cechy w zestawie uczącym
min_on_training = X_train.min(axis=0)
# obliczyć zakres każdej cechy (max – min) w zestawie uczącym
range_on_training = (X_train - min_on_training).max(axis=0)
```

```
# odejmij min, a następnie podziel przez zakres,
# min = 0 i max = 1 dla każdej cechy
X_train_scaled = (X_train - min_on_training) / range_on_training
print("Minimum dla każdej cechy\n{}".format(X_train_scaled.min(axis=0)))
print("Maximum dla każdej cechy\n {}".format(X_train_scaled.max(axis=0)))
```

Out[85]:

```
Minimum dla każdej cechy
[ 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
Maximum dla każdej cechy
[ 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

In[86]:

```
# użyj TEJ SAMEJ transformacji na zestawie testowym,
# używaj min i zakresu zestawu uczącego (szczegóły w rozdziale 3.)
X_test_scaled = (X_test - min_on_training) / range_on_training
```

In[87]:

```
svc = SVC()
svc.fit(X_train_scaled, y_train)
print("Dokładność w danych uczących: {:.3f}".format(
svc.score(X_train_scaled, y_train)))
print("Dokładność w danych testowych: {:.3f}".format(svc.score(X_test_scaled, y_test)))
```

Out[87]:

```
Dokładność w danych uczących: 0.948
Dokładność w danych testowych: 0.951
```

Skalowanie danych spowodowało ogromną różnicę! Obecnie model jest niedopasowany, wyniki uczenia i zestawu testowego są dość podobne, ale mniej zbliżają się do 100% dokładności. Teraz, aby dopasować bardziej złożony model, możemy spróbować zwiększyć parametr `C` lub `gamma`, np.:

In[88]:

```
svc = SVC(C=1000)
svc.fit(X_train_scaled, y_train)
print("Dokładność w danych uczących: {:.3f}".format(
svc.score(X_train_scaled, y_train)))
print("Dokładność w danych testowych: {:.3f}".format(svc.score(X_test_scaled, y_test)))
```

Out[88]:

```
Dokładność w danych uczących: 0.988
Dokładność w danych testowych: 0.972
```

W tym przypadku zwiększenie parametru `C` pozwala nam znacznie ulepszyć model, co da 97,2% dokładności.

Mocne i słabe strony oraz parametry

Maszyny wektorów nośnych to potężne modele, które dobrze radzą sobie z różnymi zestawami danych. Maszyny SVM pozwalają na stosowanie złożonych granic decyzyjnych, nawet jeśli dane mają tylko kilka cech. Działają dobrze w przypadku danych niskowymiarowych i wielowymiarowych (tj. kilku i wielu cech), ale nie skalują się dobrze z liczbą próbek. Przetwarzanie danych, które

zawierają do 10 000 próbek, na maszynie SVM może działać dobrze, ale praca z zestawami danych o rozmiarze 100 000 lub większym może być trudna pod względem czasu wykonywania i wykorzystania pamięci.

Inną wadą maszyn SVM jest to, że wymagają starannego wstępnego przetwarzania danych i dostrajania parametrów. Dlatego w dzisiejszych czasach większość ludzi w wielu aplikacjach używa modeli opartych na drzewach, takich jak lasy losowe lub wzmacnianie gradientowe (które wymagają niewielkiego przetwarzania wstępnego lub wcale go nie wymagają). Ponadto modele SVM są trudne do zbadania; zrozumienie, dlaczego wykonano określoną prognozę, może być trudne, tak samo jak wyjaśnienie modelu osobie niebędącej ekspertem.

Mimo to warto wypróbować maszyny SVM, zwłaszcza jeśli wszystkie cechy reprezentują pomiary w podobnych jednostkach (np. wszystkie oznaczają intensywność pikseli) i mają podobną skalę.

Ważnymi parametrami w maszynach SVM są parametr regularyzacji `C`, wybór jądra oraz parametry specyficzne dla jądra. Chociaż skupiliśmy się głównie na jądrze RBF, w bibliotece `scikit-learn` są dostępne inne opcje. Jądro RBF ma tylko jeden parametr, `gamma`, który jest odwrotnością szerokości jądra Gaussa. Parametry `gamma` i `C` kontrolują złożoność modelu, a im większe mają wartości, tym bardziej złożony jest model. Dlatego dobre ustawienia tych dwóch parametrów są zwykle silnie skorelowane, a parametry `C` i `gamma` powinny być regulowane razem.

Sieci neuronowe (głębokie uczenie)

Rodzina algorytmów znanych jako sieci neuronowe niedawno odrodziła się pod nazwą głębokie uczenie (ang. *deep learning*). Głębokie uczenie jest niezwykle obiecujące w wielu aplikacjach uczenia maszynowego, a jego algorytmy są często bardzo dokładnie dostosowywane do konkretnego przypadku użycia. Omówimy tylko niektóre, stosunkowo proste metody, a mianowicie *perceptrony wielowarstwowe* do klasyfikacji i regresji, które mogą służyć jako punkt wyjścia dla bardziej zaawansowanych metod głębokiego uczenia. Wielowarstwowe perceptrony (MLP — ang. *multilayer perceptrons*) są również znane jako (standardowe) sieci neuronowe typu *feed-forward* lub czasami po prostu jako sieci neuronowe.

Model sieci neuronowej

MLP można postrzegać jako uogólnienie modeli liniowych, które wykonują wiele etapów przetwarzania w celu prognozowania.

Pamiętaj, że sposób prognozowania regresora liniowego można opisać wzorem:

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + \ldots + w[p] * x[p] + b$$

To znaczy, że $\hat{y}$ jest sumą cech wejściowych od *x[0]* do *x[p]*, ważoną poznanymi współczynnikami od *w[0]* do *w[p]*. Wizualizację graficzną przedstawiono na rysunku 2.44:

In[89]:

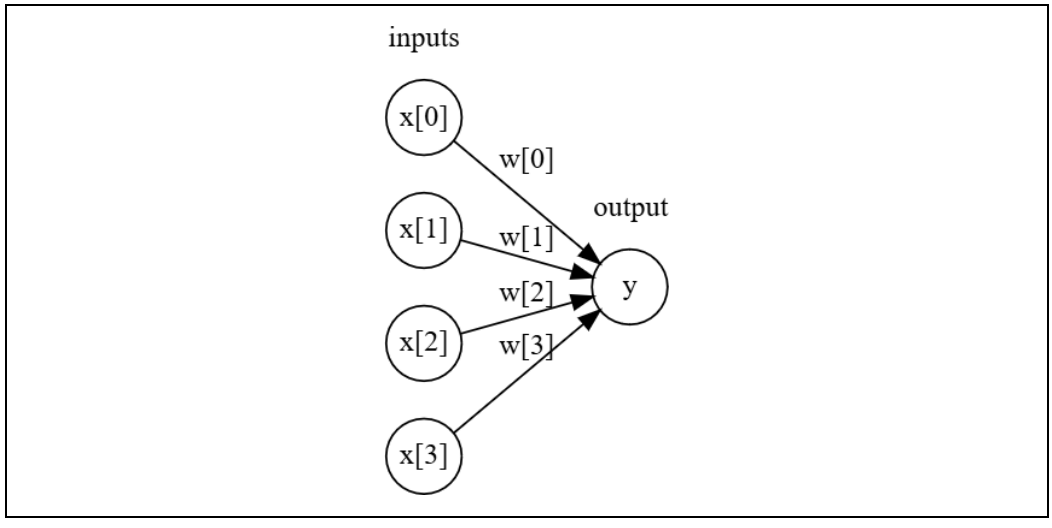
```
display(mglearn.plots.plot_logistic_regression_graph())
```

Każdy węzeł po lewej stronie reprezentuje cechę wejściową, linie łączące reprezentują wyuczone współczynniki, a węzeł po prawej reprezentuje dane wyjściowe, które są ważoną sumą nakładów.

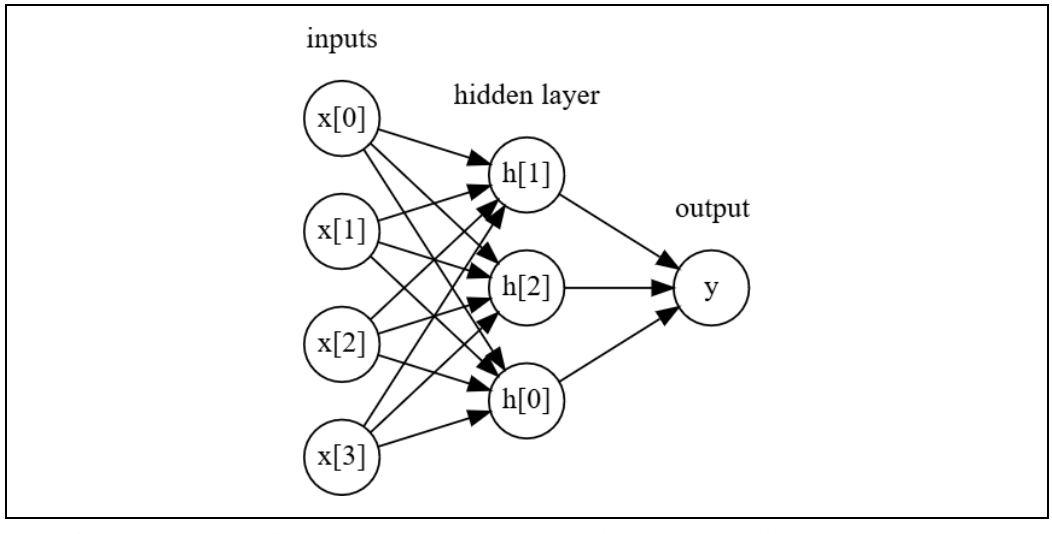
W przypadku MLP proces obliczania sum ważonych jest powtarzany wiele razy, najpierw następuje obliczenie ukrytych jednostek, które reprezentują pośredni etap przetwarzania i aby uzyskać ostateczny wynik, są ponownie łączone za pomocą sum ważonych, jak przedstawiono na rysunku 2.45:

`In[90]:`

```
display(mglearn.plots.plot_single_hidden_layer_graph())
```



Rysunek 2.44. Wizualizacja regresji logistycznej — cechy wejściowe i prognozy zostały przedstawione jako węzły, a współczynniki są połączeniami między węzłami



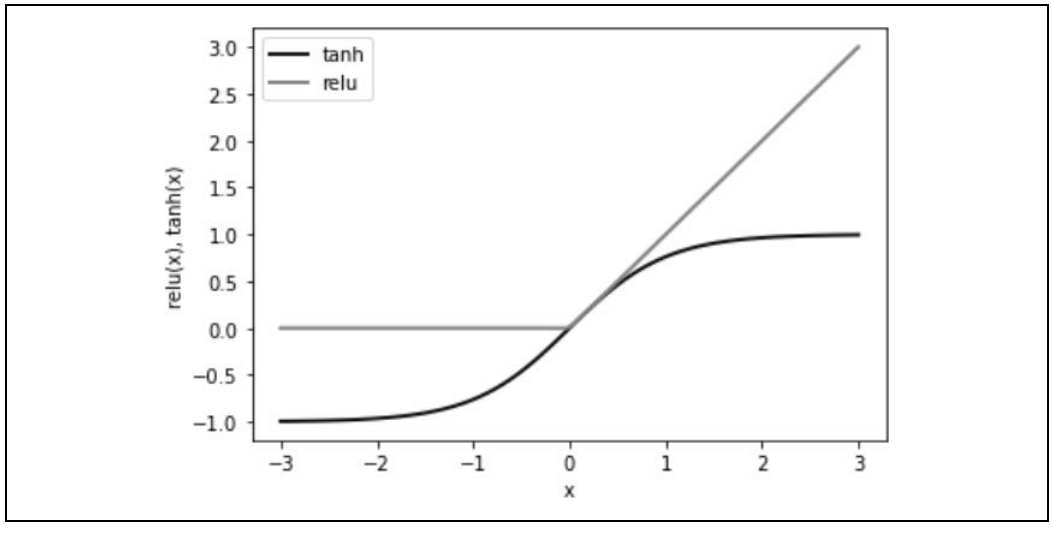
Rysunek 2.45. Ilustracja wielowarstwowego perceptronu z pojedynczą warstwą ukrytą

Ten model musi się nauczyć o wiele więcej współczynników (które nazywamy też wagami): jeden między wszystkimi poszczególnymi danymi wejściowymi a każdą ukrytą jednostką (która tworzy *warstwę ukrytą*) i jeden między każdą jednostką w warstwie ukrytej a danymi wyjściowymi.

Obliczanie serii sum ważonych jest matematycznie tym samym, co obliczanie tylko jednej sumy ważonej, więc aby ten model był lepszy niż model liniowy, musimy zastosować jedną dodatkową sztuczkę. Po obliczeniu sumy ważonej dla każdej ukrytej jednostki do wyniku stosowana jest funkcja nieliniowa — zwykle *rektyfikująca nieliniowość* (znana również jako rektyfikowana jednostka liniowa lub relu) lub *tangens hyperbolicus* (tanh). Wynik tej funkcji jest następnie używany w sumie ważonej, która oblicza wynik $\hat{y}$. Obie funkcje przedstawiono na rysunku 2.46. Relu odcina wartości poniżej 0, a tanh nasyca się do –1 dla niskich wartości wejściowych i do +1 dla wysokich wartości wejściowych. Każda z funkcji nieliniowych pozwala sieci neuronowej uczyć się znacznie bardziej skomplikowanych funkcji niż model liniowy:

In[91]:

```
line = np.linspace(-3, 3, 100)
plt.plot(line, np.tanh(line), label="tanh")
plt.plot(line, np.maximum(line, 0), label="relu")
plt.legend(loc="best")
plt.xlabel("x")
plt.ylabel("relu(x), tanh(x)")
```



Rysunek 2.46. Funkcja aktywacji stycznej hiperbolicznej i rektyfikowana liniowa funkcja aktywacji

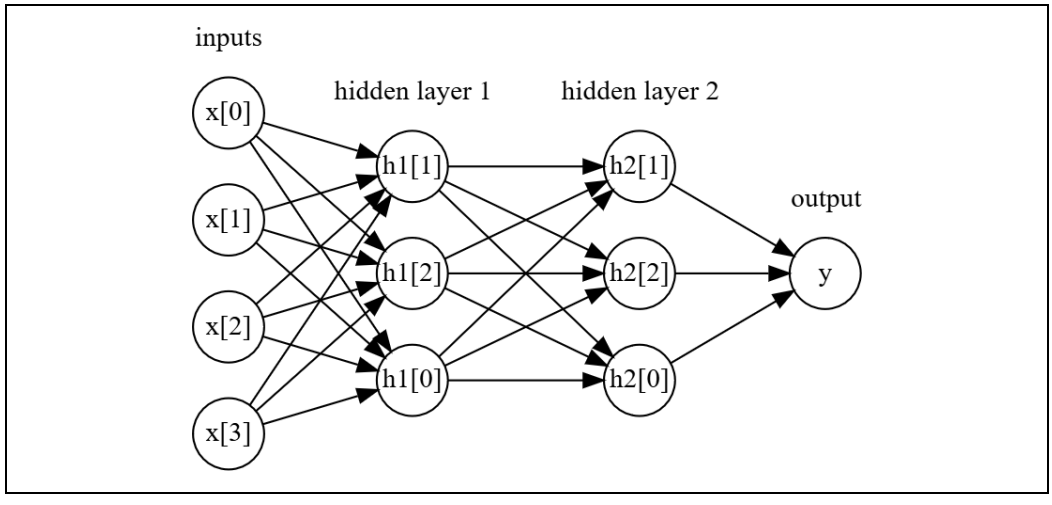
W przypadku małej sieci neuronowej, którą pokazano na rysunku 2.45, pełny wzór na obliczenie $\hat{y}$ w razie regresji wyglądałby tak, jak przedstawiono w poniższym listingu (przy użyciu nieliniowości tanh):

```
h[0] = tanh(w [0,0] * x[0] + w[1,0] * x[1] + w[2,0] * x[2] + w[3,0] * x[3])
h[1] = tanh(w [0,0] * x[0] + w[1,0] * x[1] + w[2,0] * x[2] + w[3,0] * x[3])
h[2] = tanh(w [0,0] * x[0] + w[1,0] * x[1] + w[2,0] * x[2] + w[3,0] * x[3])
ŷ = v[0] * h[0] + v[1] * h[1] + v[2] * h[2]
```

w oznacza wagę między danymi wejściowymi *x* i ukrytą warstwą *h*, natomiast *v* oznacza wagę między ukrytą warstwą *h* a danymi wyjściowymi $\hat{y}$. Wagi *v* i *w* są pobierane z danych, *x* to cechy wejściowe, $\hat{y}$ to obliczone dane wyjściowe, a *h* to obliczenia pośrednie. Ważnym parametrem, który użytkownik musi ustawić, jest liczba węzłów w warstwie ukrytej. Może być ich nawet 10 dla bardzo małych lub prostych zestawów danych i nawet 10 000 dla bardzo złożonych danych. Możliwe jest również dodanie dodatkowych warstw ukrytych, jak pokazano na rysunku 2.47:

In[92]:

```
mglearn.plots.plot_two_hidden_layer_graph()
```



Rysunek 2.47. Wielowarstwowy perceptron z dwiema ukrytymi warstwami

Właśnie od dużych sieci neuronowych, które składają się z wielu z warstw obliczeniowych, wziął się termin uczenie głębokie.

Dostrajanie sieci neuronowych

Poprzez zastosowanie `MLPClassifier` do zestawu danych `two_moons`, którego używaliśmy wcześniej w tym rozdziale, możemy przyjrzeć się działaniu MLP. Wynik zastosowania przedstawiono na rysunku 2.48:

In[93]:

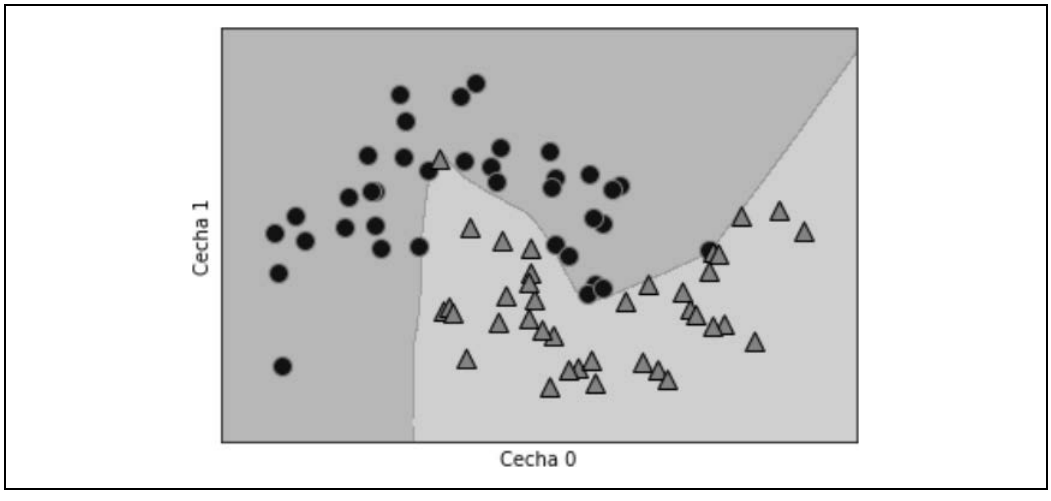
```
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=100, noise=0.25, random_state=3)
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, random_state=42)
mlp = MLPClassifier(algorithm='l-bfgs', random_state=0).fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Cecha 0")
plt.ylabel("Cecha 1")
```

Jak widać, sieć neuronowa nauczyła się nieliniowej, ale stosunkowo gładkiej granicy decyzyjnej. Do jej utworzenia użyliśmy argumentu `algorithm='l-bfgs'`, który omówimy później.

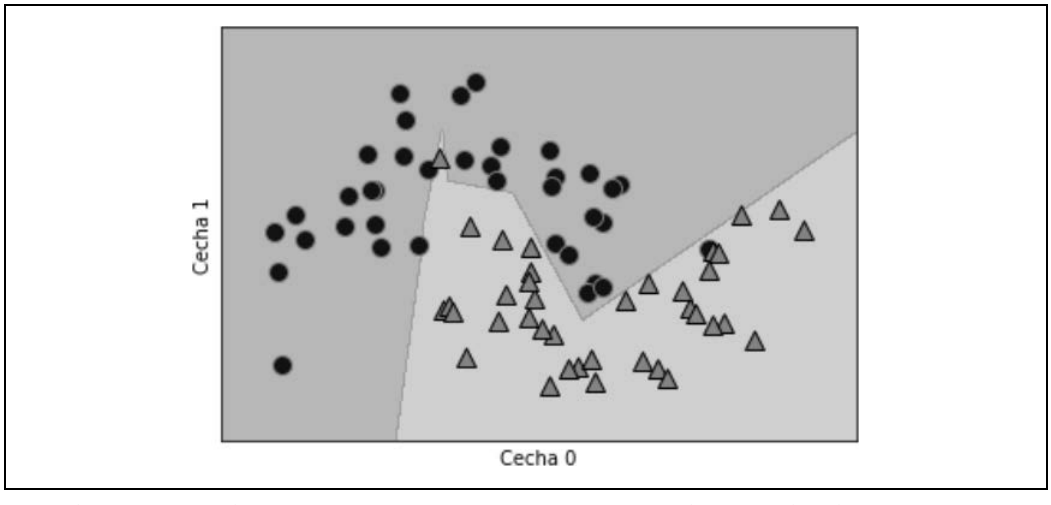
Domyślnie MLP wykorzystuje 100 ukrytych węzłów, co jest dość wysoką liczbą jak na tak mały zestaw danych. Możemy zmniejszyć liczbę (co zmniejszy złożoność modelu) i nadal uzyskiwać dobry wynik, jak przedstawiono na rysunku 2.49:

In[94]:

```
mlp = MLPClassifier(algorithm='l-bfgs', random_state=0, hidden_layer_sizes=[10])
mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Cecha 0")
plt.ylabel("Cecha 1")
```



Rysunek 2.48. Granica decyzyjna wyuczona przez sieć neuronową ze 100 ukrytymi jednostkami w zestawie danych two_moons



Rysunek 2.49. Granica decyzyjna wyuczona przez sieć neuronową z 10 ukrytymi jednostkami w zestawie danych two_moons

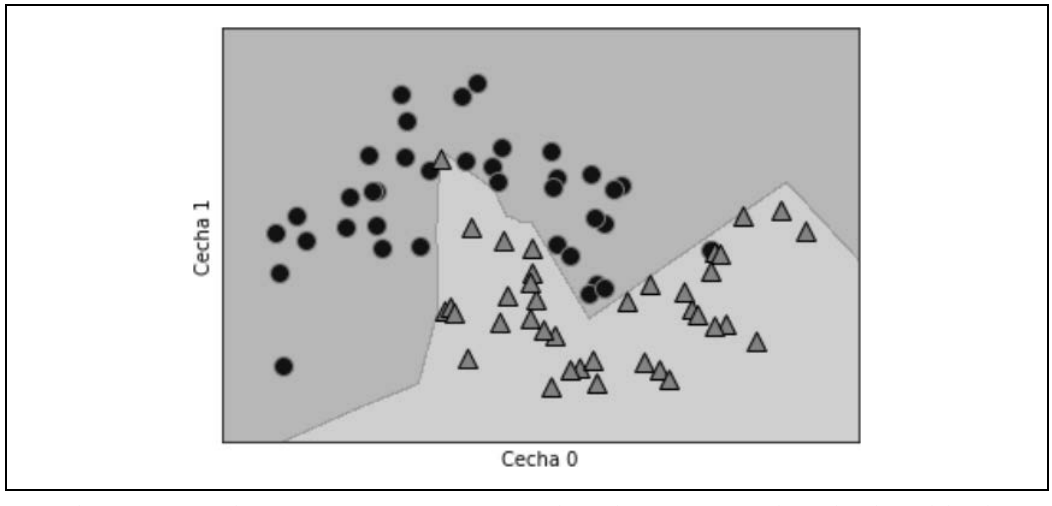
Przy tylko 10 ukrytych jednostkach granica decyzyjna wygląda na nieco bardziej nierówną. Domyślną nieliniowością jest relu, którą pokazano na rysunku 2.46. W przypadku pojedynczej warstwy ukrytej oznacza to, że funkcja decyzyjna będzie się składać z 10 odcinków prostych. Jeśli chcemy uzyskać gładszą granicę decyzyjną, możemy dodać więcej ukrytych jednostek (jak pokazano na rysunku 2.49), dodać drugą ukrytą warstwę (jak widać na rysunku 2.50) lub użyć nieliniowości tanh (co przedstawiono na rysunku 2.51):

In[95]:

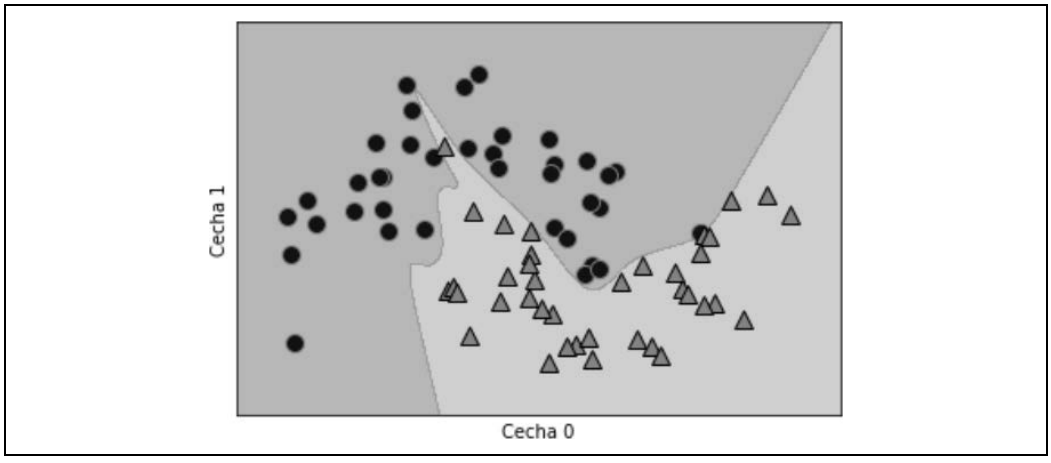
```
# używaj dwóch ukrytych warstw, po 10 jednostek każda
mlp = MLPClassifier(algorithm='l-bfgs', random_state=0, hidden_layer_sizes=[10, 10])
mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Cecha 0")
plt.ylabel("Cecha 1")
```

In[96]:

```
# używaj dwóch ukrytych warstw, po 10 jednostek każda, tym razem z nieliniowością tanh
mlp = MLPClassifier(algorithm='l-bfgs', activation='tanh',
random_state=0, hidden_layer_sizes=[10, 10])
mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Cecha 0")
plt.ylabel("Cecha 1")
```



Rysunek 2.50. Granica decyzyjna poznana przy użyciu 2 ukrytych warstw po 10 ukrytych jednostek każda, z prostą funkcją aktywacji

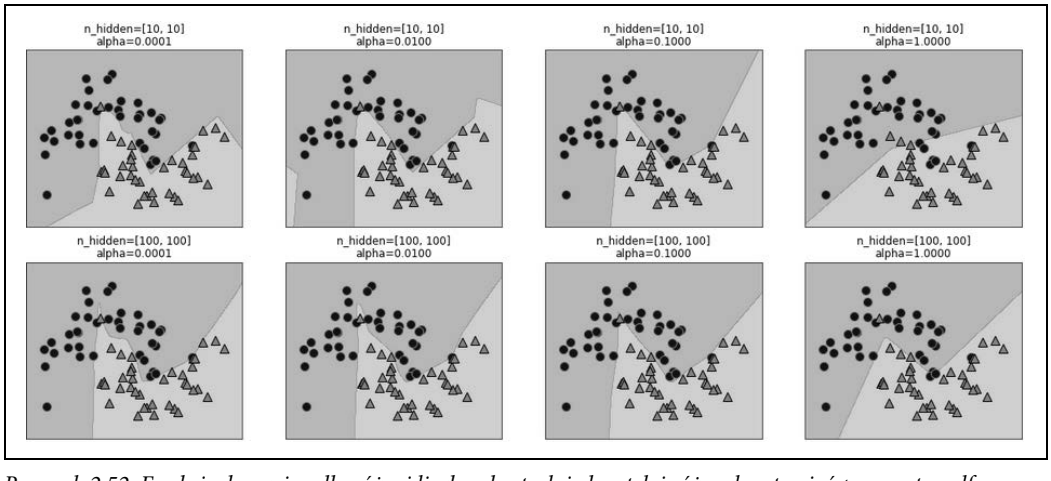


Rysunek 2.51. Granica decyzyjna poznana przy użyciu 2 ukrytych warstw po 10 ukrytych jednostek każda, z funkcją aktywacji tanh

Możemy również kontrolować złożoność sieci neuronowej, używając kary `l2` do zmniejszenia wagi do 0, tak jak to zrobiliśmy w przypadku regresji grzbietowej i liniowych klasyfikatorów. Parametr, który służy do tego w modelu `MLPClassifier`, nazywa się `alfa` (jak w modelach regresji liniowej) i jest domyślnie ustawiony na bardzo niską wartość (niewielka regularyzacja). Wpływ różnych wartości `alfa` na zestaw danych `two_moons` przy użyciu dwóch ukrytych warstw po 10 lub 100 jednostek pokazano na rysunku 2.52:

`In[97]:`

```
fig, axes = plt.subplots(2, 4, figsize=(20, 8))
for axx, n_hidden_nodes in zip(axes, [10, 100]):
for ax, alpha in zip(axx, [0.0001, 0.01, 0.1, 1]):
mlp = MLPClassifier(algorithm='l-bfgs', random_state=0, hidden_layer_sizes=[n_hidden_nodes,
↪n_hidden_nodes], alpha=alpha)
mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3, ax=ax)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train, ax=ax)
ax.set_title("n_hidden=[{}, {}]\nalpha={:.4f}".format(n_hidden_nodes, n_hidden_nodes, alpha))
```



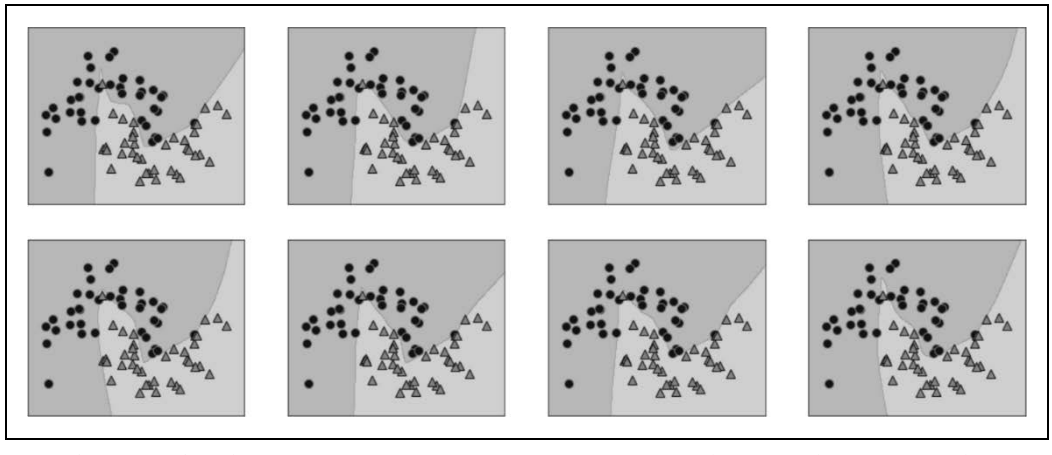
Rysunek 2.52. Funkcje decyzyjne dla różnej liczby ukrytych jednostek i różnych ustawień parametru alfa

Zapewne zdajesz już sobie sprawę, że istnieje wiele sposobów na kontrolowanie złożoności sieci neuronowej: liczba ukrytych warstw, liczba jednostek w każdej ukrytej warstwie i regularyzacja (`alfa`). W rzeczywistości jest ich jeszcze więcej, ale nie będziemy ich tutaj omawiać.

Ważną właściwością sieci neuronowych jest to, że ich wagi są ustawiane losowo przed rozpoczęciem uczenia, a ta losowa inicjalizacja ma wpływ na wyuczony model. Oznacza to, że nawet przy zastosowaniu dokładnie tych samych parametrów możemy uzyskiwać bardzo różne modele, używając różnych losowo zainicjalizowanych modeli. Jeśli sieci są duże, a ich złożoność — odpowiednio dobrana, nie powinno to mieć większego wpływu na dokładność, ale warto o tym pamiętać (szczególnie w przypadku mniejszych sieci). Wykresy kilku modeli wyuczonych z tymi samymi ustawieniami parametrów przedstawiono na rysunku 2.53:

In[98]:

```
fig, axes = plt.subplots(2, 4, figsize=(20, 8))
for i, ax in enumerate(axes.ravel()):
    mlp = MLPClassifier(algorithm='l-bfgs', random_state=i, hidden_layer_sizes=[100, 100])
    mlp.fit(X_train, y_train)
    mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3, ax=ax)
    mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train, ax=ax)
```



Rysunek 2.53. Funkcje decyzyjne nauczone z tymi samymi parametrami, ale różnymi losowymi inicjalizacjami

Aby lepiej zrozumieć działanie sieci neuronowej na rzeczywistych danych, zastosujmy klasyfikator MLP do zestawu Breast Cancer. Zaczynamy od parametrów domyślnych:

In[99]:

```
print("Maksymalne dane dotyczące raka dla każdej cechy:\n{}".format(cancer.data.max(axis=0)))
```

Out[99]:

```
Maksymalne dane dotyczące raka dla każdej cechy:
[ 28.110 39.280 188.500 2501.000 0.163 0.345 0.427
0.201 0.304 0.097 2.873 4.885 21.980 542.200
0.031 0.135 0.396 0.053 0.079 0.030 36.040
49.540 251.200 4254.000 0.223 1.058 1.252 0.291
0.664 0.207]
```

In[100]:

```
X_train, X_test, y_train, y_test = train_test_split(
cancer.data, cancer.target, random_state=0)
mlp = MLPClassifier(random_state=42)
mlp.fit(X_train, y_train)
print("Dokładność w zestawie uczącym: {:.2f}".format(mlp.score(X_train, y_train)))
print("Dokładność w zestawie testowym: {:.2f}".format(mlp.score(X_test, y_test)))
```

Out[100]:

```
Dokładność w zestawie uczącym: 0.92
Dokładność w zestawie testowym: 0.90
```

Dokładność modelu MLP jest całkiem dobra, ale nie tak dobra jak innych modeli. Podobnie jak we wcześniejszym przykładzie z SVC, jest to prawdopodobnie spowodowane skalowaniem danych. Sieci neuronowe również oczekują, że wszystkie funkcje wejściowe będą się różnić w podobny sposób i najlepiej, aby miały średnią równą 0 i wariancję równą 1. Aby dane spełniały te wymagania, musimy je przeskalować. Jeszcze raz zrobimy to ręcznie, ale w rozdziale 3. wprowadzimy klasę `StandardScaler`, która wykona to automatycznie:

In[101]:

```
# oblicz średnią wartość dla cechy w zestawie uczącym
mean_on_train = X_train.mean(axis=0)
# oblicz odchylenie standardowe każdej cechy w zestawie uczącym
std_on_train = X_train.std(axis=0)
# odejmij średnią i skaluj przez odwrotne odchylenie standardowe
# później mean = 0 i std = 1
X_train_scaled = (X_train - mean_on_train) / std_on_train
# użyj TEJ SAMEJ transformacji (przy użyciu mean i sdt) na zestawie testowym
X_test_scaled = (X_test - mean_on_train) / std_on_train
mlp = MLPClassifier(random_state=0)
mlp.fit(X_train_scaled, y_train)
print("Dokładność w zestawie uczącym: {:.3f}".format(mlp.score(X_train_scaled, y_train)))
print("Dokładność w zestawie testowym: {:.3f}".format(mlp.score(X_test_scaled, y_test)))
```

Out[101]:

```
Dokładność w zestawie uczącym: 0.991
Dokładność w zestawie testowym: 0.965
ConvergenceWarning:
Stochastic Optimizer: Maximum iterations reached and the optimization hasn't converged yet.
```

Po wykonaniu skalowania wyniki są znacznie lepsze i mogą już konkurować z innymi modelami. Otrzymaliśmy jednak od modelu ostrzeżenie, że osiągnięto maksymalną liczbę iteracji. Jest to część algorytmu do uczenia modelu `adam`, która mówi nam, że powinniśmy zwiększyć liczbę iteracji:

In[102]:

```
mlp = MLPClassifier(max_iter=1000, random_state=0)
mlp.fit(X_train_scaled, y_train)
print("Dokładność w zestawie uczącym: {:.3f}".format(mlp.score(X_train_scaled, y_train)))
print("Dokładność w zestawie testowym: {:.3f}".format(mlp.score(X_test_scaled, y_test)))
```

Out[102]:

```
Dokładność w zestawie uczącym: 0.995
Dokładność w zestawie testowym: 0.965
```

Zwiększenie liczby iteracji spowodowało tylko zwiększenie wydajności w zestawie uczącym, a nie wydajność uogólniania. Mimo to model radzi sobie całkiem dobrze. Ponieważ wciąż istnieje pewna różnica między wydajnością na zestawie uczącym a testowym, to aby uzyskać lepszą wydajność generalizacji, możemy spróbować zmniejszyć złożoność modelu. Zwiększymy parametr `alfa` (dość znacznie, bo z 0,0001 na 1), aby mocniej uregulować wagi cech:

In[103]:

```
mlp = MLPClassifier(max_iter=1000, alpha=1, random_state=0)
mlp.fit(X_train_scaled, y_train)
print("Dokładność w zestawie uczącym: {:.3f}".format(mlp.score(X_train_scaled, y_train)))
print("Dokładność w zestawie testowym: {:.3f}".format(mlp.score(X_test_scaled, y_test)))
```

Out[103]:

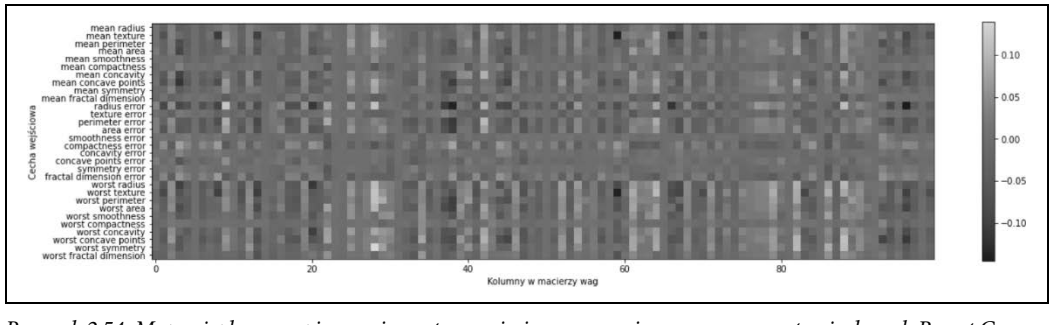
```
Dokładność w zestawie uczącym: 0.988
Dokładność w zestawie testowym: 0.972
```

Dzięki temu uzyskaliśmy wydajność porównywalną do najlepszych jak dotąd modeli[12].

Choć przeanalizowanie wszystkiego, czego sieć neuronowa się nauczyła, jest możliwe, zwykle bywa to znacznie trudniejsze niż analiza modelu liniowego lub modelu drzewa. Jednym ze sposobów introspekcji tego, czego sieć się nauczyła, jest przyjrzenie się wagom w modelu. Możesz to zobaczyć w galerii przykładów biblioteki `scikit-learn` (*http://scikit-learn.org/stable/auto_examples/neural_networks/plot_mnist_filters.html*). W przypadku zestawu danych Breast Cancer może to być nieco trudne do zrozumienia. Wagi cech, których model nauczył się podczas podłączania danych wejściowych do pierwszej ukrytej warstwy, przedstawiono na wykresie na rysunku 2.54 — wiersze odpowiadają 30 cechom wejściowym, a kolumny 100 ukrytym jednostkom; jasne kolory reprezentują duże wartości dodatnie, a kolory ciemne wskazują wartości ujemne:

In[104]:

```
plt.figure(figsize=(20, 5))
plt.imshow(mlp.coefs_[0], interpolation='none', cmap='viridis')
plt.yticks(range(30), cancer.feature_names)
plt.xlabel("Kolumny w macierzy wag")
plt.ylabel("Cecha wejściowa")
plt.colorbar()
```



Rysunek 2.54. Mapa cieplna wag pierwszej warstwy w sieci neuronowej wyuczona na zestawie danych Breast Cancer

[12] Uważny czytelnik zauważy, że wiele z dobrze działających modeli osiągnęło taką samą dokładność: 0,972. Oznacza to, że wszystkie modele popełniają dokładnie taką samą liczbę błędów, czyli cztery. Jeśli porównasz te prognozy, możesz nawet zobaczyć, że popełniają dokładnie te same błędy! Może to być konsekwencją tego, że zbiór danych jest bardzo mały, lub wynikać z tego, że dane punkty naprawdę różnią się od pozostałych.

Jednym z możliwych wniosków, jakie możemy wyciągnąć, jest to, że cechy, które mają bardzo małe wagi dla wszystkich ukrytych jednostek, są „mniej ważne" dla modelu. Widzimy, że „mean smoothness" (średnia gładkość) i „mean compactness" (średnia zwartość), oprócz cech znalezionych między „smoothness error" (błędem gładkości) a „fractal dimension error" (błędem wymiaru fraktalnego), mają stosunkowo niską wagę w porównaniu z innymi cechami. Może to oznaczać, że są to mniej ważne cechy lub być może nie przedstawiliśmy ich w sposób, który mogłaby wykorzystać sieć neuronowa.

Moglibyśmy również zwizualizować wagi, które łączą warstwę ukrytą z warstwą wyjściową, ale są one jeszcze trudniejsze do zinterpretowania.

Chociaż modele `MLPClassifier` i `MLPRegressor` zapewniają łatwe w użyciu interfejsy dla najpopularniejszych architektur sieci neuronowych, wychwytują tylko niewielki podzbiór tego, co jest w nich możliwe. Jeśli interesuje Cię praca z bardziej elastycznymi lub większymi modelami, zachęcamy do wyjścia poza bibliotekę `scikit-learn` i sięgnięcia do fantastycznych bibliotek głębokiego uczenia, które są dostępne w sieci. Najbardziej znanymi bibliotekami dla języka Python są `keras`, `lasagna` i `tensor-flow`. `lasagna` opiera się na bibliotece `theano`, a `keras` może używać `tensor-flow` lub `theano`. Biblioteki te zapewniają znacznie bardziej elastyczny interfejs do budowy sieci neuronowych i śledzenia szybkiego postępu w badaniach głębokiego uczenia. Wszystkie popularne biblioteki głębokiego uczenia pozwalają również na użycie wysokowydajnych procesorów graficznych (GPU), których biblioteka `scikit-learn` nie obsługuje. Korzystanie z procesorów graficznych pozwala nam przyspieszyć obliczenia na współczynnikach od 10 do 100 razy i są one niezbędne do stosowania metod głębokiego uczenia do dużych zestawów danych.

Mocne i słabe strony oraz parametry

Sieci neuronowe są najnowocześniejszymi modelami w wielu zastosowaniach uczenia maszynowego. Jedną z ich głównych zalet jest to, że potrafią przechwytywać informacje zawarte w dużych ilościach danych i budować niezwykle złożone modele. Jeśli ma się wystarczająco dużo czasu na obliczenia, odpowiednią ilość danych i starannie dostrojone parametry, to sieci neuronowe zazwyczaj pokonują inne algorytmy uczenia maszynowego (w jakości wykonywania zadań klasyfikacji i regresji).

Mają też wady. Sieci neuronowe — zwłaszcza te duże i potężne — często wymagają dużo czasu na uczenie. Wymagają również starannego wstępnego przetwarzania danych, jak widzieliśmy we wcześniej omówionych przykładach. Podobnie jak maszyny SVM, najlepiej sprawdzają się w przypadku danych „jednorodnych", gdy wszystkie cechy mają podobne znaczenie. Natomiast w przypadku danych, które mają bardzo różne rodzaje cech, lepiej działać mogą modele oparte na drzewach decyzyjnych. Dostrajanie parametrów sieci neuronowej jest również sztuką samą w sobie. W naszych eksperymentach ledwo otarliśmy się o ogrom możliwych sposobów dostosowania modeli sieci neuronowych i sposobów ich uczenia.

Szacowanie złożoności w sieciach neuronowych. Najważniejszymi parametrami są liczba warstw i liczba ukrytych jednostek na warstwę. Zacznij od jednej lub dwóch ukrytych warstw i dodawaj kolejne w razie potrzeby. Liczba węzłów na warstwę ukrytą jest często podobna do liczby cech wejściowych, ale rzadko kiedy jest wyższa niż 7000.

Myśląc o złożoności modelu sieci neuronowej, powinniśmy brać pod uwagę liczbę wyuczonych wag lub współczynników. Jeśli masz binarny zestaw danych klasyfikacji ze 100 cechami i 100 ukrytych jednostek, to między danymi wejściowymi a pierwszą ukrytą warstwą jest 100*100 = 10 000 wag. Między warstwą ukrytą a warstwą wyjściową występuje również 100*1 = 100 wag, co daje w sumie około 10 100 wag. Jeśli dodasz drugą ukrytą warstwę ze 100 ukrytymi jednostkami, pojawi się kolejne 100*100 = 10 000 wag między pierwszą a drugą ukrytą warstwą, co da w sumie 20 100 wag. Jeśli zamiast tego użyjesz jednej warstwy z 1000 ukrytych jednostek, to uczysz model 100*1000 = 100 000 wag od warstwy wejściowej do ukrytej i 1000*1 między warstwą ukrytą a wyjściową, co daje łącznie 101 000. Jeśli dodasz drugą ukrytą warstwę, dodasz 1000*1000 = 1 000 000 wag, co daje olbrzymią sumę 1 101 000 — 50 razy większą niż model z dwiema ukrytymi warstwami o rozmiarze 100.

Typowym sposobem dostosowania parametrów w sieci neuronowej rozpoczyna się od stworzenia sieci, która jest wystarczająco duża, aby była nadmiernie dopasowana — przy upewnieniu się, że faktycznie może się ona tego nauczyć. Następnie, gdy już wiesz, że model może się nauczyć danych uczących, to aby poprawić wydajność generalizacji, dodaj regularyzację, zmniejszając sieć lub zwiększając parametr `alfa`.

W naszych eksperymentach skupialiśmy się głównie na definicji modelu: liczbie warstw i węzłów na warstwę, regularyzacji i nieliniowości. To definiuje model, którego chcemy się nauczyć. Pojawia się również pytanie, *jak* nauczyć się modelu, czyli algorytmu, który służy do uczenia parametrów i który ustawia się za pomocą parametru `algorithm`. Istnieją dwie łatwe w użyciu opcje dla parametru `algorithm`. Wartość domyślna to `'adam'`, która działa dobrze w większości sytuacji, ale jest dość wrażliwa na skalowanie danych (dlatego ważne jest, aby zawsze skalować dane do średniej 0 i wariancji jednostkowej). Druga opcja to `'l-bfgs'`, który jest dość uniwersalny, ale w przypadku większych modeli lub większych zestawów danych jego uczenie może zająć dużo czasu. Istnieje również bardziej zaawansowana opcja `'sgd'`, z której korzysta wielu badaczy zajmujących się głębokim uczeniem. Opcja `'sgd'` zawiera wiele dodatkowych parametrów, które należy dostroić, aby uzyskać najlepsze wyniki. Wszystkie te parametry i ich definicje można znaleźć w instrukcji obsługi. Kiedy zaczynasz pracę z MLP, zalecamy trzymać się opcji `'adam'` i `'l-bfgs'`.

fit resetuje model

Ważną właściwością modeli dostępnych w bibliotece `scikit-learn` jest to, że wywołanie metody `fit` zawsze zresetuje wszystko, czego model wcześniej się nauczył. Jeśli więc utworzysz model na jednym zestawie danych, a następnie ponownie wywołasz metodę `fit` dla innego zestawu danych, model „zapomni” wszystko, czego się nauczył z pierwszego zestawu danych. Metodę `fit` modelu można wywoływać wiele razy, a wynik zawsze będzie taki sam jak w przypadku „nowego” modelu.

Szacunki niepewności na podstawie klasyfikatorów

Kolejną użyteczną częścią interfejsu biblioteki `scikit-learn`, o której jeszcze nie pisaliśmy, jest zdolność klasyfikatorów do zapewniania oszacowań niepewności prognoz. Zapewne często interesuje Cię nie tylko to, jaka klasa jest przez klasyfikator rozpoznana dla określonego punktu testowego, ale także jak pewne jest, że to właściwa klasa. W praktyce różnego rodzaju błędy prowadzą do bardzo różnych wyników w rzeczywistych zastosowaniach. Wyobraź sobie aplikację, która

testuje wyniki badań w celu rozpoznania raka. Sformułowanie fałszywie pozytywnej prognozy może prowadzić do poddania pacjenta dodatkowym badaniom, a fałszywie negatywna prognoza może prowadzić do zaniechania leczenia poważnej choroby. Bardziej szczegółowo zajmiemy się tym tematem w rozdziale 6.

W bibliotece `scikit-learn` istnieją dwie różne funkcje, których można użyć do szacowania niepewności z klasyfikatorów: `decision_function` i `predict_proba`. Większość klasyfikatorów (ale nie wszystkie) zawiera co najmniej jedną z nich, a wiele zawiera obie. Przyjrzyjmy się, co te dwie funkcje zwrócą w przypadku syntetycznego dwuwymiarowego zestawu danych — aby to zrobić, stworzymy klasyfikator `GradientBoostingClassifier`, który ma zarówno `decision_function`, jak i `predict_proba`:

In[105]:

```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.datasets import make_blobs, make_circles
X, y = make_circles(noise=0.25, factor=0.5, random_state=1)
# w celach ilustracyjnych zmieniamy nazwy klas na „niebieski" i „czerwony"
y_named = np.array(["niebieski", "czerwony"])[y]
# możemy wywołać funkcję train_test_split z dowolną liczbą tablic;
# wszystko zostanie podzielone w spójny sposób
X_train, X_test, y_train_named, y_test_named, y_train, y_test = train_test_split(X, y_named,
↪y, random_state=0)
# zbuduj model wzmocnienia gradientu
gbrt = GradientBoostingClassifier(random_state=0)
gbrt.fit(X_train, y_train_named)
```

Funkcja decyzyjna

W przypadku binarnej klasyfikacji wartość zwracana przez `decision_function` jest kształtu `(n_samples,)` i dla każdej próbki zwraca jedną liczbę zmiennoprzecinkową:

In[106]:

```
print("X_test.shape: {}".format(X_test.shape))
print("Kształt funkcji decyzyjnej: {}".format(
gbrt.decision_function(X_test).shape))
```

Out[106]:

```
X_test.shape: (25, 2)
Kształt funkcji decyzyjnej: (25,)
```

Ta wartość oznacza stopień, w jakim model uważa, że punkt danych należy do klasy „pozytywnej", w tym przypadku do klasy 1. Wartości dodatnie oznaczają preferencję dla klasy pozytywnej, a wartości ujemne wskazują na preferencje dla klasy „negatywnej" (innej):

In[107]:

```
# pokaż kilka pierwszych wpisów, które zwraca funkcja decision_function
print("Funkcja decyzyjna:\n{}".format(gbrt.decision_function(X_test)[:6]))
```

Out[107]:

```
Funkcja decyzyjna:
[ 4.136 -1.683 -3.951 -3.626 4.29 3.662]
```

Możemy odzyskać prognozę, patrząc tylko na znak wartości, którą zwraca funkcja decyzyjna:

In[108]:

```
print("Wyniki funkcji decyzyjnej:\n{}".format(
gbrt.decision_function(X_test) > 0))
print("Prognozy:\n{}".format(gbrt.predict(X_test)))
```

Out[108]:

```
Wyniki funkcji decyzyjnej:
[False  True  True  True False False  True False False False  True False
 False  True False  True  True  True False False False False False  True
  True]
Prognozy:
['czerwony' 'niebieski' 'niebieski' 'niebieski' 'czerwony' 'czerwony'
 'niebieski' 'czerwony' 'czerwony' 'czerwony' 'niebieski' 'czerwony'
 'czerwony' 'niebieski' 'czerwony' 'niebieski' 'niebieski' 'niebieski'
 'czerwony' 'czerwony' 'czerwony' 'czerwony' 'czerwony' 'niebieski'
 'niebieski']
```

W przypadku klasyfikacji binarnej klasa „negatywna” zawsze jest pierwszym wpisem w atrybucie `classes_`, a klasa „pozytywna” jest drugim wpisem w atrybucie `classes_`. Jeśli więc chcesz w pełni odtworzyć wynik funkcji `predict`, musisz użyć atrybutu `classes_`:

In[109]:

```
# przekształć typ bool w wartości 0 i 1
greater_zero = (gbrt.decision_function(X_test) > 0).astype(int)
# użyj 0 i 1 jako indeksów atrybutu classes_
pred = gbrt.classes_[greater_zero]
# pred jest tym samym, co wynik gbrt.predict
print("pred jest równy prognozą: {}".format(
np.all(pred == gbrt.predict(X_test))))
```

Out[109]:

```
pred jest równy prognozą: True
```

Zakres `decision_function` może być dowolny i zależy od danych i parametrów modelu:

In[110]:

```
decision_function = gbrt.decision_function(X_test)
print("Minimum funkcji decyzyjnej: {:.2f} maksimum: {:.2f}".format(np.min(decision_function),
➥np.max(decision_function)))
```

Out[110]:

```
Minimum funkcji decyzyjnej: -7.69 maksimum: 4.29
```

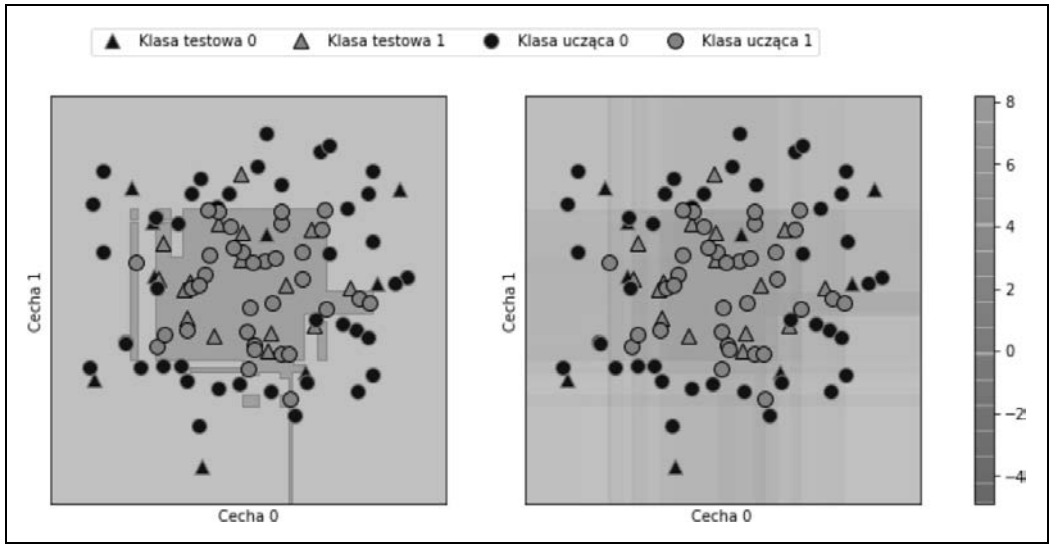
To arbitralne skalowanie sprawia, że wynik `decision_function` często jest trudny do zinterpretowania.

W poniższym przykładzie dla wszystkich punktów na płaszczyźnie 2D wykreślono `decision_function` za pomocą kodowania kolorami, obok wizualizacji granicy decyzyjnej. Na rysunku 2.55 punkty uczące przedstawiono w postaci kół, a testowe w postaci trójkątów:

In[111]:

```
fig, axes = plt.subplots(1, 2, figsize=(13, 5))
mglearn.tools.plot_2d_separator(gbrt, X, ax=axes[0], alpha=.4, fill=True, cm=mglearn.cm2)
scores_image = mglearn.tools.plot_2d_scores(gbrt, X, ax=axes[1], alpha=.4, cm=mglearn.ReBl)
```

```
for ax in axes:
    # umieść punkty uczące i testowe na wykresie
    mglearn.discrete_scatter(X_test[:, 0], X_test[:, 1], y_test, markers='^', ax=ax)
    mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train, markers='o', ax=ax)
    ax.set_xlabel("Cecha 0")
    ax.set_ylabel("Cecha 1")
cbar = plt.colorbar(scores_image, ax=axes.tolist())
axes[0].legend(["Klasa testowa 0", "Klasa testowa 1", "Klasa ucząca 0", "Klasa ucząca 1"],
↪ncol=4, loc=(.1, 1.1))
```



Rysunek 2.55. Granica decyzyjna (po lewej) i funkcja decyzyjna (po prawej) dla modelu ze wzmocnieniem gradientu na dwuwymiarowym przykładowym zestawie danych

Zakodowanie pewności klasyfikatora oraz prognozowanego wyniku dostarcza dodatkowych informacji. Jednak w tej wizualizacji trudno jest dostrzec granicę między tymi dwiema klasami.

Prognozy prawdopodobieństw

Wynik funkcji `predict_proba` oznacza prawdopodobieństwo dla każdej klasy i często jest łatwiejszy do zrozumienia niż wynik funkcji decyzyjnej. W przypadku klasyfikacji binarnej zawsze ma kształt `(n_samples, 2)`:

`In[112]:`

```
print("Kształt prawdopodobieństw: {}".format(gbrt.predict_proba(X_test).shape))
```

`Out[112]:`

```
Kształt prawdopodobieństw: (25, 2)
```

Pierwszy wpis w każdym wierszu oznacza oszacowane prawdopodobieństwo dla pierwszej klasy, a drugi wpis to oszacowane prawdopodobieństwo dla drugiej klasy. Ponieważ jest to prawdopodobieństwo, wynik funkcji `predict_proba` zawsze mieści się między 0 a 1, a suma wpisów dla obu klas zawsze wynosi 1:

In[113]:

```
# pokaż kilka pierwszych wpisów z predict_proba
print("Prognozowane prawdopodobieństwa:\n{}".format(gbrt.predict_proba(X_test[:6])))
```

Out[113]:

```
Prognozowane prawdopodobieństwa:
[[0.984 0.016]
 [0.157 0.843]
 [0.019 0.981]
 [0.026 0.974]
 [0.986 0.014]
 [0.975 0.025]]
```

Ponieważ prawdopodobieństwa dla dwóch klas sumują się do 1, dla dokładnie jednej klasy pewność będzie większa niż 50%. To właśnie ta klasa jest wynikiem prognozy[13].

W danych wyjściowych ostatniego przykładu widać, że klasyfikator dla większości punktów prognozuje ze względnie dużą pewnością. To, jak dobrze niepewność klasyfikatora faktycznie odzwierciedla niepewność w danych, zależy od modelu i parametrów. Model, który jest bardziej dopasowany, ma tendencję do tworzenia pewniejszych prognoz, nawet jeśli są błędne. Model o mniejszym stopniu złożoności zwykle swoje prognozy tworzy mniej pewnie. Model nazywany jest *skalibrowanym*, jeśli raportowana niepewność faktycznie odpowiada temu, z jaką dokładnością wykonywane są prognozy — w modelu skalibrowanym prognoza wykonana z 70-procentową pewnością byłaby poprawna w 70% przypadków.

W poniższym przykładzie (rysunek 2.56) ponownie pokazano granicę decyzyjną w zestawie danych obok prawdopodobieństw klas dla klasy 1:

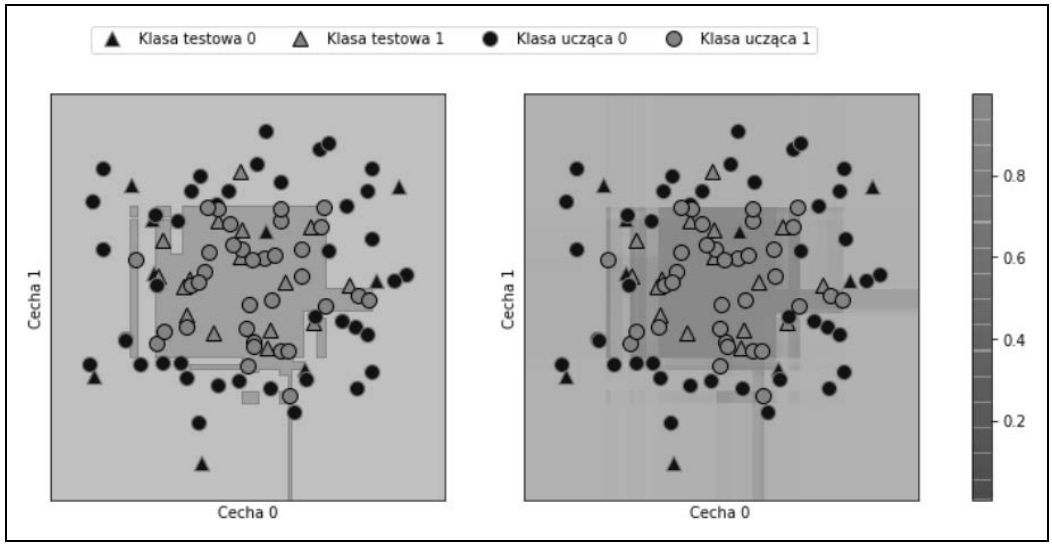
In[114]:

```
fig, axes = plt.subplots(1, 2, figsize=(13, 5))
mglearn.tools.plot_2d_separator(gbrt, X, ax=axes[0], alpha=.4, fill=True, cm=mglearn.cm2)
scores_image = mglearn.tools.plot_2d_scores(gbrt, X, ax=axes[1], alpha=.5, cm=mglearn.ReBl,
↪function='predict_proba')
for ax in axes:
    # umieść punkty uczące i testowe na wykresie
    mglearn.discrete_scatter(X_test[:, 0], X_test[:, 1], y_test, markers='^', ax=ax)
    mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train, markers='o', ax=ax)
    ax.set_xlabel("Cecha 0")
    ax.set_ylabel("Cecha 1")
cbar = plt.colorbar(scores_image, ax=axes.tolist())
axes[0].legend(["Klasa testowa 0", "Klasa testowa 1", "Klasa ucząca 0", "Klasa ucząca 1"],
↪ncol=4, loc=(.1, 1.1))
```

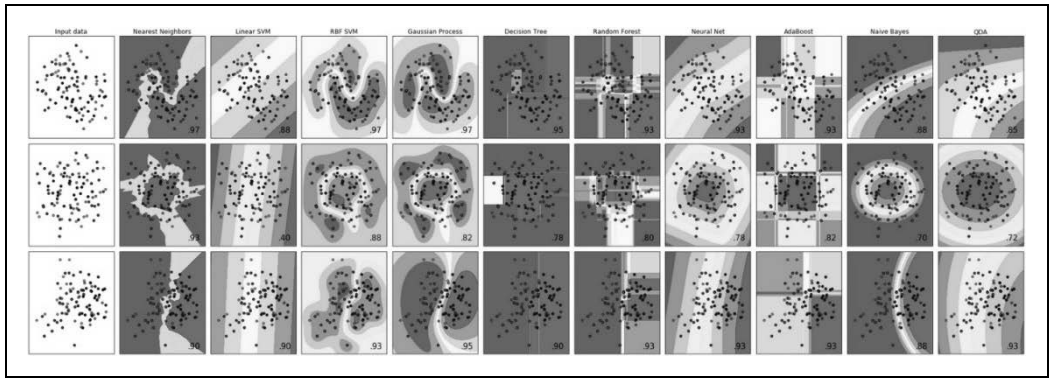
Granice na tym wykresie są znacznie dokładniej zdefiniowane, a małe obszary niepewności — wyraźnie widoczne.

Na stronie biblioteki `scikit-learn` (*http://bit.ly/2cqCYx6*) znajduje się świetne porównanie wielu modeli i tego, jak wyglądają ich szacunki niepewności. Zachęcamy do przejrzenia zamieszczonego tam przykładu, który również umieszczono na rysunku 2.57.

[13] Ponieważ prawdopodobieństwa są liczbami zmiennoprzecinkowymi, jest mało prawdopodobne, aby oba były równe dokładnie 0,500. Jeśli jednak tak się stanie, wynik prognozy jest zwracany losowo.



Rysunek 2.56. Granica decyzyjna (po lewej) i prognozowane prawdopodobieństwa dla modelu wzmocnienia gradientu pokazanego na rysunku 2.55



Rysunek 2.57. Porównanie kilku dostępnych w bibliotece scikit-learn klasyfikatorów na syntetycznych zestawach danych (rysunek pochodzi ze strony http://scikit-learn.org/)

Niepewność w klasyfikacji wieloklasowej

Do tej pory omówiliśmy tylko szacunki niepewności w klasyfikacji binarnej. Ale metody `decision_function` i `predict_proba` działają również w ustawieniu wieloklasowym. Zastosujmy je do zestawu danych Iris, który jest trzyklasowym zestawem danych klasyfikacyjnych:

In[115]:

```
from sklearn.datasets import load_iris
iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target,
random_state=42)
gbrt = GradientBoostingClassifier(learning_rate=0.01, random_state=0)
gbrt.fit(X_train, y_train)
```

In[116]:

```
print("Kształt funkcji decyzyjnej: {}".format(gbrt.decision_function(X_test).shape))
# umieść na wykresie kilka pierwszych wpisów z funkcji decyzyjnej
print("Funkcja decyzyjna:\n{}".format(gbrt.decision_function(X_test)[:6, :]))
```

Out[116]:

```
Kształt funkcji decyzyjnej: (38, 3)
Funkcja decyzyjna:
[[-0.529 1.466 -0.504]
 [ 1.512 -0.496 -0.503]
 [-0.524 -0.468 1.52 ]
 [-0.529 1.466 -0.504]
 [-0.531 1.282 0.215]
 [ 1.512 -0.496 -0.503]]
```

W przypadku wieloklasowym funkcja `decision_function` ma kształt `(n_samples, n_classes)`, a każda kolumna zapewnia „wynik pewności" dla każdej klasy, w której duży wynik oznacza, że klasa jest bardziej prawdopodobna, a niski wynik oznacza, że klasa jest mniej prawdopodobna. Możesz odzyskać prognozy na podstawie tych wyników, znalazłszy maksymalny wpis dla każdego punktu danych:

In[117]:

```
print("Maksymalny argument funkcji decyzyjnej:\n{}".format(
np.argmax(gbrt.decision_function(X_test), axis=1)))
print("Prognozy:\n{}".format(gbrt.predict(X_test)))
```

Out[117]:

```
Maksymalny argument funkcji decyzyjnej:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1 0]
Prognozy:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1 0]
```

Dane wyjściowe funkcji `predict_proba` mają ten sam kształt `(n_samples, n_classes)`. Również w tym przypadku suma prawdopodobieństw możliwych klas dla każdego punktu danych wynosi 1:

In[118]:

```
# wyświetl pierwsze kilka wpisów z wyniku funkcji predict_proba
print("Prognozowane prawdopodobieństwa:\n{}".format(gbrt.predict_proba(X_test)[:6]))
# wyświetl sumy wszystkich wierszy, które wynoszą 1
print("Sumy: {}".format(gbrt.predict_proba(X_test)[:6].sum(axis=1)))
```

Out[118]:

```
Prognozowane prawdopodobieństwa:
[[ 0.107 0.784 0.109]
 [ 0.789 0.106 0.105]
 [ 0.102 0.108 0.789]
 [ 0.107 0.784 0.109]
 [ 0.108 0.663 0.228]
 [ 0.789 0.106 0.105]]
Sumy: [ 1. 1. 1. 1. 1. 1.]
```

Możemy odzyskać prognozy, obliczywszy `argmax` z `predict_proba`:

`In[119]:`

```
print("Argmax prognozowanych
prawdopodobieństw:\n{}".format(np.argmax(gbrt.predict_proba(X_test), axis=1)))
print("Prognozy:\n{}".format(gbrt.predict(X_test)))
```

`Out[119]:`

```
Argmax prognozowanych prawdopodobieństw:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1 0]
Prognozy:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1 0]
```

Podsumowując, funkcje `predict_proba` i `decision_function` zawsze mają kształt (`n_samples`, `n_classes`) — wyjątkiem jest `decision_function` w specjalnym przypadku binarnym. Funkcja decyzyjna ma w nim tylko jedną kolumnę, odpowiadającą klasie „pozytywnej" `classes_[1]`. Dzieje się tak głównie z powodów historycznych.

Gdy istnieje `n_classes` kolumn, możesz odzyskać prognozę, obliczywszy `argmax` w kolumnach. Ale musisz uważać, jeżeli Twoje klasy są ciągami znaków lub liczbami całkowitymi, które nie są kolejne i zaczynają się od 0. Jeśli chcesz porównać wyniki otrzymane z predykcją z wynikami uzyskanymi za pomocą funkcji decyzyjnej lub `predict_proba`, upewnij się, że używasz atrybutu `classes_` klasyfikatora, aby uzyskać rzeczywiste nazwy klas:

`In[120]:`

```
logreg = LogisticRegression()
# zaprezentuj każdy cel poprzez nazwę klasy w zestawie danych Iris
named_target = iris.target_names[y_train]
logreg.fit(X_train, named_target)
print("unikalne klasy w danych uczących: {}".format(logreg.classes_))
print("prognozy: {}".format(logreg.predict(X_test)[:10]))
argmax_dec_func = np.argmax(logreg.decision_function(X_test), axis=1)
print("argmax funkcji decyzyjnej: {}".format(argmax_dec_func[:10]))
print("argmax razem z classes_: {}".format(logreg.classes_[argmax_dec_func][:10]))
```

`Out[120]:`

```
unikalne klasy w danych uczących: ['setosa' 'versicolor' 'virginica']
prognozy: ['versicolor' 'setosa' 'virginica' 'versicolor' 'versicolor'
'setosa' 'versicolor' 'virginica' 'versicolor' 'versicolor']
argmax funkcji function: [1 0 2 1 1 0 1 2 1 1]
argmax razem z classes_: ['versicolor' 'setosa' 'virginica' 'versicolor' 'versicolor' 'setosa'
↪'versicolor' 'virginica' 'versicolor' 'versicolor']
```

Podsumowanie i przegląd

Zaczęliśmy ten rozdział od omówienia złożoności modelu, po czym omówiliśmy *uogólnianie*, czyli zdolność modelu do prognozowania na nowych, niepoznanych wcześniej danych. Następnie przeszliśmy do definicji niedopasowania, która oznacza, że model nie może uchwycić zmian obecnych w danych uczących, oraz definicji nadmiernego dopasowania, dotyczącej modelu zbytnio skupiającego się na danych uczących, przez co nie jest w stanie bardzo dobrze uogólniać nowych danych.

Następnie omówiliśmy szeroką gamę modeli uczenia maszynowego do klasyfikacji i regresji, a także to, jakie są ich zalety i wady oraz jak dla każdego z nich kontrolować złożoność modelu. Widzieliśmy, że aby osiągnąć dobrą wydajność w przypadku wielu algorytmów, ważne jest ustawienie odpowiednich parametrów. Niektóre algorytmy są również wrażliwe na sposób, w jaki przedstawiamy dane wejściowe, a w szczególności na sposób skalowania funkcji. Dlatego ślepe zastosowanie algorytmu do zestawu danych bez zrozumienia założeń przyjętych przez model i znaczenia ustawień parametrów rzadko prowadzi do uzyskania dokładnego modelu.

W tym rozdziale zawarto wiele informacji o algorytmach, ale znajomość wszystkich szczegółów nie jest konieczna do lektury następnych rozdziałów. Jednak pewna wiedza na temat opisanych tutaj modeli — i tego, których z nich należy użyć w określonej sytuacji — jest ważna dla skutecznego zastosowania uczenia maszynowego w praktyce. W poniższym krótkim podsumowaniu opisano, kiedy należy używać każdego modelu:

Najbliżsi sąsiedzi

Modele dobre do małych zestawów danych, dobre jako punkt odniesienia, łatwe do wyjaśnienia.

Modele liniowe

Pierwsze algorytmy, które należy wypróbować, dobre w przypadku bardzo dużych zestawów danych oraz danych wielowymiarowych.

Naiwne klasyfikatory Bayesa

Tylko do klasyfikacji. Szybsze niż modele liniowe, dobre w przypadku bardzo dużych zestawów danych i danych wielowymiarowych. Często mniej dokładne niż modele liniowe.

Drzewa decyzyjne

Bardzo szybkie, nie wymagają skalowania danych, można je łatwo zwizualizować i wyjaśnić.

Lasy losowe

Prawie zawsze działają lepiej niż pojedyncze drzewo decyzyjne, są bardzo solidne i potężne. Nie potrzebują skalowania danych. Niezbyt dobre w przypadku rzadkich danych o bardzo dużych wymiarach.

Drzewa decyzyjne ze wzmocnieniem gradientowym

Zazwyczaj nieco dokładniejsze niż lasy losowe. Potrzebują więcej czasu na uczenie, ale szybciej tworzą prognozy niż lasy losowe i zajmują mniej pamięci. Wymagają więcej dostrajania parametrów niż lasy losowe.

Maszyny wektorów nośnych

Potężne w przypadku średnich zestawów danych funkcji o podobnym znaczeniu. Wymagają skalowania danych wrażliwych na parametry.

Sieci neuronowe

Mogą budować bardzo złożone modele, szczególnie dla dużych zestawów danych. Wrażliwe na skalowanie danych i dobór parametrów. Duże modele potrzebują dużo czasu na uczenie.

Zazwyczaj dobrym pomysłem podczas pracy z nowym zestawem danych jest rozpoczęcie od prostego modelu, takiego jak model liniowy, naiwny klasyfikator Bayesa lub najbliższego sąsiada, i zorientowanie się, jak się sprawdzają. Po dokładniejszym poznaniu danych można rozważyć przejście na algorytm, który może tworzyć bardziej złożone modele, takie jak lasy losowe, drzewa decyzyjne ze wzmocnieniem gradientowym, maszyny SVM lub sieci neuronowe.

Masz już zatem pojęcie o tym, jak stosować, dostrajać i analizować modele, które tutaj omówiliśmy. W tym rozdziale skupiliśmy się na przypadku klasyfikacji binarnej, ponieważ jest on zwykle najłatwiejszy do zrozumienia. Większość przedstawionych algorytmów jednak ma warianty klasyfikacji i regresji, a wszystkie algorytmy klasyfikacyjne obsługują klasyfikację binarną i wieloklasową. Spróbuj zastosować dowolny z tych algorytmów do wbudowanych w bibliotekę `scikit-learn` zestawów danych, takich jak `boston_housing` lub `diabetes` w przypadku regresji albo zestaw danych `digits` w przypadku klasyfikacji wieloklasowej. Bawienie się algorytmami w różnych zestawach danych pozwoli lepiej zrozumieć, jak długo muszą się uczyć, jak łatwo jest analizować modele i jak wrażliwe są one na reprezentację danych.

Chociaż przeanalizowaliśmy konsekwencje różnych ustawień parametrów dla algorytmów, które badaliśmy, to zbudowanie modelu faktycznie dobrze uogólniającego nowe dane w produkcji jest nieco trudniejsze. W rozdziale 6. opisano, jak automatycznie znaleźć dobre parametry i jak prawidłowo je dostosować.

Zanim jednak do tego przejdziemy, zajmiemy się bardziej szczegółowo uczeniem i przetwarzaniem wstępnym bez nadzoru, co zostało opisane w następnym rozdziale.

O'REILLY®

Python i uczenie maszynowe: programowanie do zadań specjalnych!

Uczenie maszynowe kojarzy się z dużymi firmami i rozbudowanymi zespołami. Prawda jest taka, że obecnie można samodzielnie budować zaawansowane rozwiązania uczenia maszynowego i korzystać do woli z olbrzymich zasobów dostępnych danych. Trzeba tylko mieć pomysł i... trochę podstawowej wiedzy. Tymczasem większość opracowań na temat uczenia maszynowego i sztucznej inteligencji wymaga biegłości w zaawansowanej matematyce. Utrudnia to naukę tego zagadnienia, mimo że uczenie maszynowe jest coraz powszechniej stosowane w projektach badawczych i komercyjnych.

Ta praktyczna książka ułatwi Ci rozpoczęcie wdrażania rozwiązań rzeczywistych problemów związanych z uczeniem maszynowym. Zawiera przystępne wprowadzenie do uczenia maszynowego i sztucznej inteligencji, a także sposoby wykorzystania Pythona i biblioteki scikit-learn, uwzględniające potrzeby badaczy i analityków danych oraz inżynierów pracujących nad aplikacjami komercyjnymi. Zagadnienia matematyczne ograniczono tu do niezbędnego minimum, zamiast tego skoncentrowano się na praktycznych aspektach algorytmów uczenia maszynowego. Dokładnie opisano, jak konkretnie można skorzystać z szerokiej gamy modeli zaimplementowanych w dostępnych bibliotekach.

Najważniejsze zagadnienia:

- podstawowe informacje o uczeniu maszynowym
- najważniejsze algorytmy uczenia maszynowego
- przetwarzanie danych w uczeniu maszynowym
- ocena modelu i dostrajanie parametrów
- łańcuchy modeli i hermetyzacja przepływu pracy
- przetwarzanie danych tekstowych

Dr Andreas C. Müller – zajmował się uczeniem maszynowym aplikacji rozpoznawania obrazów w Amazonie, później dołączył do Center for Data Science na New York University. Jest jednym z głównych autorów biblioteki scikit-learn i kilku innych pakietów uczenia maszynowego.

Sarah Guido – jest analitykiem danych. Pracowała w kilku w start-upach. Jest ceniona za znakomite wystąpienia na prestiżowych konferencjach.

Helion

helion.pl

HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-8322-751-1

9 788383 227511

Cena: 79,00 zł