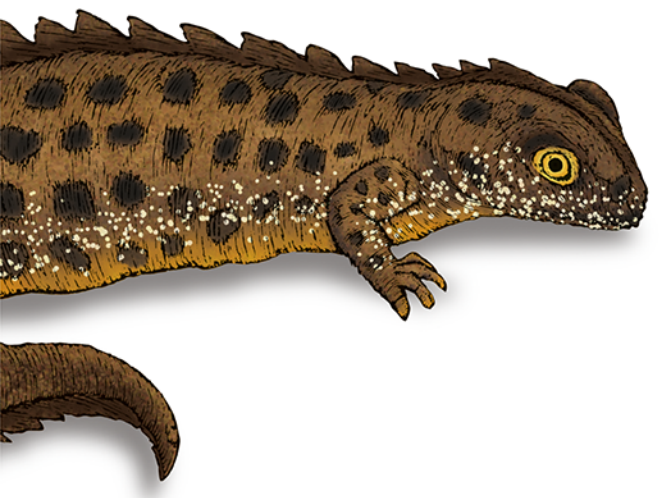
O'REILLY®

Uczenie maszynowe w Pythonie

Leksykon kieszonkowy



Helion

Matt Harrison

Tytuł oryginału: Machine Learning Pocket Reference: Working with Structured Data in Python

Tłumaczenie: Andrzej Watrak

ISBN: 978-83-283-6558-2

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

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

Printed in Poland.

Spis treści

Rozdział 3. Klasyfikacja danych: baza Titanic

W tym rozdziale opisany jest typowy proces klasyfikacji danych na przykładzie bazy Titanic (*https://oreil.ly/PjceO*). W następnych rozdziałach szczegółowo opisane są poszczególne etapy analizy danych.

Proponowany schemat projektu

Do eksploracji danych doskonale nadaje się bezpłatne środowisko Jupyter (*https://jupyter.org*), obsługujące różne języki, m.in. Python, i umożliwiające tworzenie komórek kodu i dokumentów w formacie Markdown.

Środowiska Jupyter używam w dwóch trybach. W pierwszym przeprowadzam szybką analizę danych, a w drugim przygotowuję raport w formacie Markdown zawierający komórki z kodem, który ilustruje ważne aspekty i wnioski z analizy. Jeżeli używasz innego środowiska, może być konieczne dostosowanie go do przyjętych praktyk programistycznych (usunięcie zmiennych globalnych, zastosowanie funkcji, klas itp.).

Twórcy pakietu Cookiecutter (*https://oreil.ly/86jL3*) proponują opisaną niżej strukturę projektu umożliwiającą łatwe powielanie i współdzielenie kodu.

Importowane biblioteki

W opisanym tu przykładzie wykorzystywane są głównie biblioteki `pandas` (*http://pandas.pydata.org*), `scikit-learn` (*https://scikit-learn.org*) i `Yellowbrick` (*http://www.scikit-yb.org*). Pierwsza z nich zapewnia narzędzia, za pomocą których łatwo przekształca się dane. Biblioteka `scikit-learn` umożliwia tworzenie doskonałych modeli predykcyjnych, a `Yellowbrick` służy do wizualizowania wyników i weryfikowania skuteczności modelu. Poniższy kod przedstawia niezbędne instrukcje importujące.

```
>>> import matplotlib.pyplot as plt
>>> import pandas as pd
>>> from sklearn import (
...     ensemble,
...     preprocessing,
...     tree,
... )
>>> from sklearn.metrics import (
```

```
...     auc,
...     confusion_matrix,
...     roc_auc_score,
...     roc_curve,
... )
>>> from sklearn.model_selection import (
...     train_test_split,
...     StratifiedKFold,
... )
>>> from yellowbrick.classifier import (
...     ConfusionMatrix,
...     ROCAUC,
... )
>>> from yellowbrick.model_selection import (
...     LearningCurve,
... )
```

Ostrzeżenie

W dokumentacji i przykładach w Internecie można znaleźć następującą instrukcję importującą biblioteki:

```
from pandas import *
```

Unikaj stosowania gwiazdki z instrukcją `import`. Jeżeli będziesz precyzyjnie importował biblioteki, Twój kod będzie bardziej czytelny.

Zadanie pytania

W tym przykładzie utworzymy model umożliwiający znalezienie odpowiedzi na pewne pytanie. Na podstawie charakterystyk (cech) pasażera Titanica i jego podróży określimy jego szanse przeżycia katastrofy. Przykład jest prosty, ale pozwala zademonstrować wiele etapów procesu modelowania danych.

Pytanie o szanse przeżycia jest problemem klasyfikacyjnym, w którym prognozujemy jeden z dwóch możliwych wyników: czy pasażer przeżyje katastrofę czy nie.

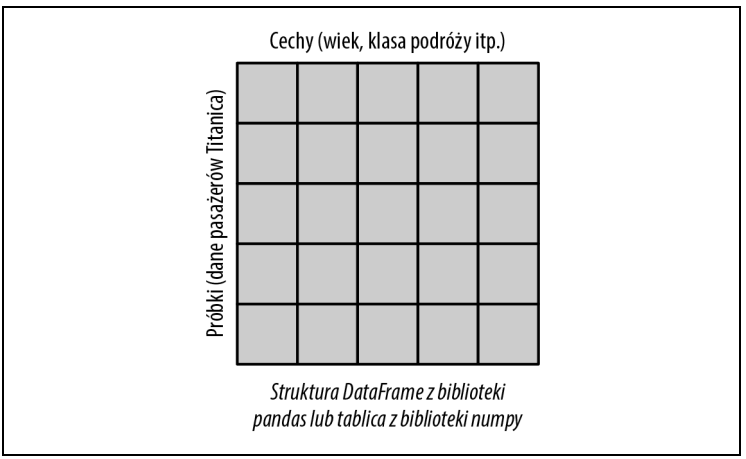
Stosowana terminologia

Model trenuje się zazwyczaj za pomocą macierzy danych. (Wykorzystuję do tego celu strukturę `DataFrames` dostępną w bibliotece `pandas`, ponieważ ma wygodne w użyciu etykiety kolumn. Dobrze sprawdza się również tablica z biblioteki `numpy`).

W nadzorowanym uczeniu, na przykład regresji lub klasyfikacji, celem jest znalezienie funkcji, która przekształca cechy w etykiety. W notacji algebraicznej funkcję tę zapisuje się w następujący sposób:

```
y = f(X)
```

Argument `X` oznacza macierz, w której poszczególnych wierszach umieszczone są **próbki danych**, czyli w tym przykładzie informacje o pasażerach. Poszczególne kolumny macierzy reprezentują **cechy** danych. Wynikiem funkcji jest wektor `y` zawierający etykiety (w tym przypadku wyniki klasyfikacji) lub wartości (wyniki regresji). Ilustruje to rysunek 3.1.



Rysunek 3.1. Układ danych strukturalnych

W powyższym opisie zostały użyte typowe terminy stosowane do opisywania danych w publikacjach naukowych, jak również w dokumentacjach bibliotek.

W programach tworzonych w języku Python próbki danych umieszcza się w zmiennej o nazwie `X`, wbrew przyjętej konwencji PEP8, według której nazwy zmiennych powinny zaczynać się od małej litery. Ten wyjątek robią wszyscy analitycy, więc gdybyś użył zmiennej o nazwie `x`, wywołałbyś zdziwienie. Etykiety, czyli cele, umieszcza się w zmiennej o nazwie `y`.

Tabela 3.1 przedstawia prosty zbiór danych złożony z dwóch próbek, z których każda ma trzy cechy.

Tabela 3.1. Próbki (wiersze) i cechy (kolumny)

pclass	age	sibsp
1	29	0
1	2	1

Zebranie danych

Napiszemy teraz kod, który będzie ładował plik Excela zawierający cechy pasażerów Titanica (wcześniej trzeba zainstalować biblioteki `pandas` i `xlrd`[1]). Dane umieszczone są w kilkunastu kolumnach. Wśród nich jest kolumna `survived` (ang. ocalał), zawierająca informację o tym, co się stało z pasażerem. Kod ładujący plik ma następującą postać:

```
>>> url = (
...     "http://biostat.mc.vanderbilt.edu/"
...     "wiki/pub/Main/DataSets/titanic3.xls"
... )
>>> df = pd.read_excel(url)
>>> orig_df = df
```

Arkusz zawiera następujące kolumny:

- *pclass* — klasa (1 — pierwsza, 2 — druga, 3 — trzecia),
- *survived* — pasażer ocalał (0 — nie, 1 — tak).
- *name* — imię i nazwisko,
- *sex* — płeć (*male* — mężczyzna, *female* — kobieta),
- *age* — wiek,
- *sibsp* — towarzysząca żona/mąż lub liczba bliźniaków,
- *parch* — liczba towarzyszących dzieci/rodziców,
- *ticket* — numer biletu,
- *fare* — cena biletu,
- *cabin* — numer kajuty,
- *embarked* — miejsce zaokrętowania (*C* = Cherbourg, *Q* = Queenstown, *S* = Southampton),
- *boat* — numer szalupy ratunkowej,
- *body* — identyfikator zwłok,
- *home.dest* — miejsce zamieszkania / cel podróży.

Za pomocą biblioteki `pandas` można załadować powyższy arkusz i przekształcić go na strukturę `DataFrame`. Zanim przystąpimy do analizy, musimy oczyścić dane i sprawdzić, czy są poprawne.

[1] Podczas ładowania pliku Excela nie będziemy odwoływać się do tej biblioteki bezpośrednio. Niejawnie korzysta z niej biblioteka `pandas`.

Oczyszczanie danych

Po załadowaniu danych należy sprawdzić, czy ich format pozwala na utworzenie modelu. W większości modeli obsługiwanych przez bibliotekę `scikit-learn` cechy muszą być liczbami (całkowitymi lub rzeczywistymi). Ponadto wiele modeli nie funkcjonuje poprawnie, jeżeli w danych występują braki (w bibliotekach `pandas` i `numpy` oznaczane symbolem `NaN`). Niektóre modele sprawdzają się lepiej, gdy dane są **znormalizowane** (tj. ich średnia jest równa 0, a odchylenie standardowe równe 1). Opisane usterki danych można korygować za pomocą bibliotek `pandas` i `scikit-learn`. Ponadto mogą pojawiać się **wyciekające dane**, czyli takie, które zawierają informacje o wyniku lub celu. Choć nie ma nic złego w tym, że tego rodzaju dane są dostępne (a w praktyce podczas tworzenia modelu zdarza się to dość często), należy je usunąć przed rozpoczęciem prognozowania wyników, ponieważ „ujawniają przyszłość".

Oczyszczanie danych może być czasochłonną czynnością, dlatego warto przy tym skorzystać z pomocy eksperta, który doradzi, co należy zrobić z odstającymi lub brakującymi danymi.

Typy danych zawartych w bazie Titanic można sprawdzić za pomocą następującego polecenia:

```
>>> df.dtypes
pclass         int64
survived       int64
name          object
sex           object
age          float64
sibsp          int64
parch          int64
ticket        object
fare         float64
cabin         object
embarked      object
boat          object
body         float64
home.dest     object
dtype: object
```

Najczęściej stosowane typy danych wykorzystywane w bibliotece `pandas` to `int64`, `float64`, `datetime64[ns]` i `object`. Typy `int64` i `float64` reprezentują dane liczbowe, `datetime64[ns]` oznacza datę i godzinę, a `object` reprezentuje ciągi znaków i ewentualnie dane innych typów.

Typy danych biblioteka `pandas` określa automatycznie podczas ładowania pliku CSV. Jeżeli dane są liczbami lub datami, wtedy przyjmowany jest typ `object`. Typy są ściśle określane podczas ładowania danych do struktury

DataFrame z arkuszy kalkulacyjnych, baz lub innych systemów. Zawsze jednak warto przeglądać załadowane dane i sprawdzać, czy typy zostały dobrane poprawnie.

Typ int64 informuje, że pomyślnie zostały załadowane liczby całkowite. Jeżeli przyjęty został typ float64, to znaczy, że może brakować niektórych wartości. Daty i ciągi znaków mogą wymagać konwersji lub przekształcenia na typy liczbowe. Kolumna zawierająca dane tekstowe o niskiej kardynalności nosi nazwę **kolumny kategorialnej**. Czasami warto na jej podstawie utworzyć dodatkową kolumnę (służy do tego metoda pd.get_dummies).

Uwaga

Wybranie typu danych int64 przez bibliotekę pandas w wersji 0.23 lub starszej oznaczało, że w kolumnie nie było brakujących wartości. Natomiast wybranie typu float64 oznaczało, że kolumna zawierała liczby rzeczywiste, całkowite i brakujące wartości (dla tego typu są one dopuszczalne). Typ object oznaczał, że kolumna zawierała ciągi znaków i ewentualnie liczby.

W wersji biblioteki pandas 0.24 został wprowadzony nowy typ Int64 (zwróć uwagę na pierwszą, wielką literę). Nie zastępuje on typu int64, natomiast obsługuje brakujące wartości.

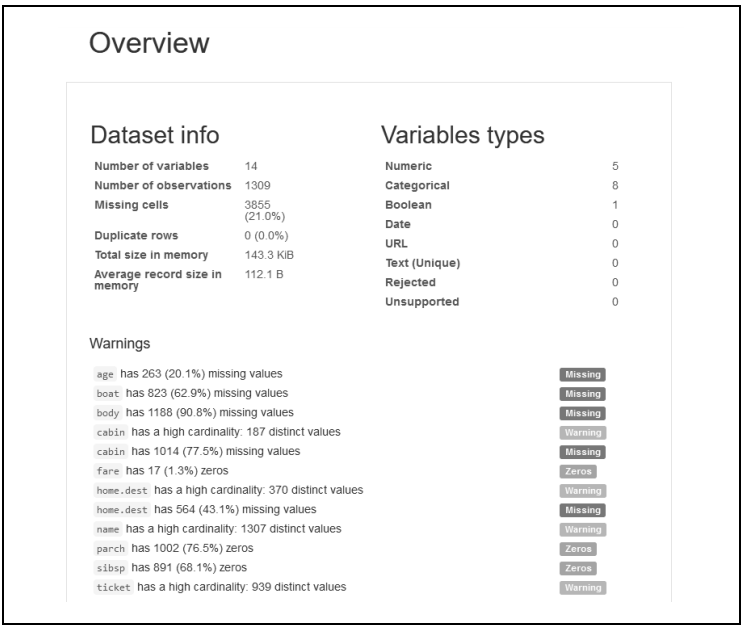
Za pomocą biblioteki pandas-profiling można utworzyć — na przykład w środowisku programistycznym — raport z profilem danych. Raport zawiera opis danych zawartych w poszczególnych kolumnach i różnego rodzaju podsumowania, na przykład histogram, wartości typowe i ekstremalne (patrz rysunek 3.2 i 3.3). Służą do tego następujące instrukcje:

```
>>> import pandas_profiling
>>> pandas_profiling.ProfileReport(df)
```

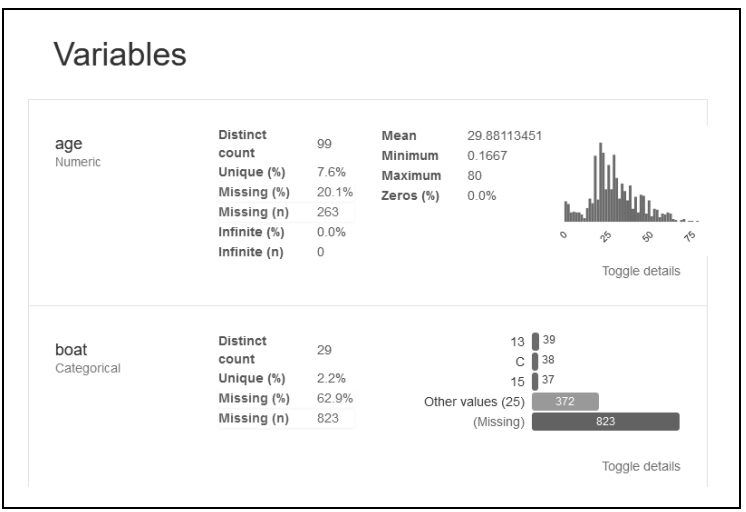
Aby sprawdzić liczbę wierszy i kolumn umieszczonych w strukturze DataFrame, należy użyć atrybutu shape:

```
>>> df.shape
(1309, 14)
```

Metoda describe zwraca podsumowanie danych oraz liczbę niepustych wartości. Domyślnie przetwarza tylko kolumny zawierające liczby. Poniżej przedstawiony jest fragment wyniku ograniczony do pierwszych dwóch kolumn:



Rysunek 3.2. Raport utworzony za pomocą biblioteki pandas-profiling



Rysunek 3.3. Szczegółowy profil danych uzyskany za pomocą biblioteki pandas-profiling

```
>>> res = df.describe().iloc[:, :2]
>>> output = res.rename(
...     {"count": "liczba", "mean": "Średnia", "std": "odch. stand."},
...     axis='index')
>>> output
                pclass     survived
liczba     1309.000000  1309.000000
Średnia       2.294882     0.381971
odch. stand.  0.837836     0.486055
min           1.000000     0.000000
25%           2.000000     0.000000
50%           3.000000     0.000000
75%           3.000000     1.000000
max           3.000000     1.000000
```

Wiersz `count` zawiera liczbę wartości innych niż `NaN`. Na tej podstawie można stwierdzić, czy kolumna zawiera komplet wartości. Minimalne i maksymalne wartości widoczne w powyższym wyniku informują, czy w danych znajdują się wartości odstające. Nie jest to jedyna forma podsumowania. Można również zwizualizować dane za pomocą histogramu lub wykresu, o czym piszę w dalszej części rozdziału.

W tym przykładzie trzeba wziąć pod uwagę fakt, że w danych brakuje niektórych wartości. Za pomocą metody `isnull` można wyszukać kolumny i wiersze zawierające niekompletne dane. Metoda ta zwraca strukturę `DataFrame`, w której każda komórka zawiera wartości `True` lub `False`, zamieniane w języku Python na liczby, odpowiednio, 1 i 0, dzięki czemu można je sumować i wyliczać odsetek braków (równy wartości średniej).

Poniższy kod wyświetla liczbę brakujących wartości w poszczególnych kolumnach:

```
>>> df.isnull().sum()
pclass          0
survived        0
name            0
sex             0
age           263
sibsp           0
parch           0
ticket          0
fare            1
cabin        1014
embarked        2
boat          823
body         1188
home.dest     564
dtype: int64
```

Wskazówka

Aby uzyskać odsetki brakujących wartości w poszczególnych kolumnach, należy zamiast metody `sum` użyć `mean`. Domyślnie obie metody działają wzdłuż osi 0, czyli indeksu. Aby uzyskać liczby brakujących wartości w poszczególnych kolumnach, należy użyć osi nr 1, jak niżej:

```
>>> df.isnull().sum(axis=1).loc[:10]
0    1
1    1
2    2
3    1
4    2
5    1
6    1
7    2
8    1
9    2

dtype: int64
```

Specjalista od przetwarzania danych może poradzić, co robić w przypadku brakujących danych. Kolumna `age` jest ważna i może dostarczyć cennych wniosków, dlatego należy ją uwzględnić w modelu. Kolumny zawierające w większości puste komórki (`cabin`, `boat` i `body`) prawdopodobnie nie zawierają ważnych danych i można je pominąć. Ponadto kolumna `body` zawiera wyciekające informacje o osobach, które nie przeżyły katastrofy. Dane te fałszowałyby wyniki, dlatego nie można ich uwzględnić w modelu. (W modelu prognozującym śmierć pasażera istnienie identyfikatora zwłok oznaczałoby, że wynik jest znany zawczasu. Model nie może takich informacji „znać" i musi prognozować wyniki na podstawie danych zawartych w innych kolumnach). Kolumna `boat` również zawiera wyciekające dane, ale odwrotne (o tym, że pasażer ocalał).

Przyjrzyjmy się kilku wierszom, w których brakuje danych. Możemy utworzyć macierz logiczną (zawierającą wartości `True` i `False`, które będą oznaczać, odpowiednio, dostępność lub brak wartości) i wykorzystać ją do wyszukania wierszy z brakującymi wartościami. Przykładowy kod może wyglądać jak niżej:

```
>>> mask = df.isnull().any(axis=1)
>>> mask.head()  # wiersze
0    True
1    True
2    True
3    True
4    True
dtype: bool
>>> df[mask].body.head()
0      NaN
```

```
1      NaN
2      NaN
3    135.0
4      NaN
Name: body, dtype: float64
```

Przetwarzaniem (lub imputacją) brakujących wartości w kolumnie `age` zajmiemy się później.

Kolumny typu `object` zazwyczaj są kategorialne (zawierają wartości tekstowe o wysokiej kategorialności lub dane różnych typów). Zakładając, że dana kolumna jest kategorialna, możemy zliczyć unikatowe wartości za pomocą metody `value_counts`, jak niżej:

```
>>> df.sex.value_counts(dropna=False)
male      843
female    466
Name: sex, dtype: int64
```

Pamiętaj, że biblioteka `pandas` pomija wartości `null` i `NaN`. Aby je uwzględnić w wynikach, należy metodę `value_counts` wywołać z argumentem `dropna=`
`↪False`, jak niżej:

```
df.embarked.value_counts(dropna=False)
S      914
C      270
Q      123
NaN      2
Name: embarked, dtype: int64
```

Z brakującymi wartościami w kolumnie `embarked` można poradzić sobie na kilka sposobów. Logicznym rozwiązaniem wydaje się przyjęcie wartości `S`, ponieważ występuje ona najczęściej. Można również zbadać dokładniej dane i sprawdzić, czy inna wartość byłaby lepsza. Jeszcze innym wyjściem jest pominięcie dwóch wierszy z brakującymi wartościami, ponieważ kolumna jest kategorialna, i utworzenie za pomocą biblioteki `pandas` kolumn pomocniczych. Ten sposób jest zastosowany w dalszej części rozdziału.

Zdefiniowanie cech

Jeżeli kolumna zawiera wartości, których wariancja lub odchylenie standardowe są równe zeru, można ją pominąć. W przykładowym zbiorze nie ma takich kolumn, ale gdyby była na przykład kolumna `human` (człowiek) zawierająca we wszystkich wierszach wartość 1, wtedy należałoby ją zignorować, ponieważ nie dostarczałaby żadnych wartościowych informacji.

Podobnie nie ma potrzeby analizowania kolumn tekstowych, w których każda wartość jest inna (chyba że przeprowadzamy analizę lingwistyczną lub z tekstu wyodrębniamy określone wartości). Przykładem takiej kolumny jest `name`. Można z niej ewentualnie wyodrębnić tytuł i przyjąć, że uzyskane dane są kategorialne.

Ponadto można pominąć kolumny zawierające wyciekające dane, tutaj `boat` i `body`, ponieważ zawierają informacje o tym, czy pasażer przeżył katastrofę.

Biblioteka `pandas` udostępnia metodę `drop`, pomijającą wiersze i kolumny. Poniżej przedstawiony jest przykład jej użycia:

```
>>> name = df.name
>>> name.head(3)
0      Allen, Miss. Elisabeth Walton
1     Allison, Master. Hudson Trevor
2       Allison, Miss. Helen Loraine
Name: name, dtype: object
>>> df = df.drop(
...     columns=[
...         "name",
...         "ticket",
...         "home.dest",
...         "boat",
...         "body",
...         "cabin",
...     ]
... )
```

W przypadku kolumn zawierających dane tekstowe, w tym przykładzie `sex` i `embarked`, należy utworzyć kolumny pomocnicze. Biblioteka `pandas` zapewnia przeznaczoną do tego celu wygodną metodę `get_dummies`:

```
>>> df = pd.get_dummies(df)
>>> df.columns
Index(['pclass', 'survived', 'age', 'sibsp',
    'parch', 'fare', 'sex_female', 'sex_male',
    'embarked_C', 'embarked_Q', 'embarked_S'],
    dtype='object')
```

Utworzone w powyższym kodzie kolumny `sex_male` i `sex_female` są idealnie odwrotnie skorelowane. Zazwyczaj należy usuwać kolumny, które są silnie ze sobą skorelowane dodatnio lub ujemnie, ponieważ w niektórych modelach mogą utrudniać ocenę ważności cech i współczynników. Poniższy kod usuwa kolumnę `sex_male`:

```
>>> df = df.drop(columns="sex_male")
```

Innym rozwiązaniem jest użycie metody `get_dummies` z argumentem `drop_first=True`:

```
>>> df = pd.get_dummies(df, drop_first=True)
```

```
>>> df.columns
Index(['pclass', 'survived', 'age', 'sibsp',
    'parch', 'fare', 'sex_male',
    'embarked_Q', 'embarked_S'],
    dtype='object')
```

Teraz można utworzyć strukturę `DataFrame` (zmienną `X`), zawierającą cechy, oraz wektor etykiet (zmienną `y`), jak w poniższym przykładzie. Za pomocą biblioteki `numpy` można też utworzyć tablicę, jednak nie będzie ona zawierała nazw kolumn.

```
>>> y = df.survived
>>> X = df.drop(columns="survived")
```

Wskazówka

Z użyciem biblioteki `pyjanitor` (*https://oreil.ly/_IWbA*) można dwa powyższe wiersze kodu zastąpić jednym, jak niżej:

```
>>> import janitor as jn
>>> X, y = jn.get_features_targets(
...     df, target_columns="survived"
... )
```

Próbkowanie danych

Każdy model należy trenować i testować na różnych danych. W przeciwnym razie nie będzie wiadomo, jak dobrze model interpretuje nieznane mu wcześniej dane. W tym przykładzie 30% danych testowych wyodrębnimy za pomocą biblioteki `scikit-learn`. Użyty niżej argument `random_state=42` usuwa czynnik losowy, aby można było porównywać różne modele danych:

```
>>> X_train, X_test, y_train, y_test = model_selection.train_test_split(
...     X, y, test_size=0.3, random_state=42
... )
```

Imputacja danych

Kolumna `age` zawiera brakujące wartości, które należy uzupełnić na podstawie danych zawartych w innych kolumnach liczbowych. Należy to zrobić tylko w treningowym zbiorze danych, a w opracowanym algorytmie wykorzystać dane testowe. Jeżeli się tego nie zrobi, pojawią się wyciekające wyniki (wprowadzające do modelu wynikowe dane). Biblioteka `fancyimpute` (*https://oreil.ly/Vlf9e*) implementuje wiele algorytmów imputacji danych. Większość z nich jednak nie jest indukcyjna, tj. nie pozwala na użycie metod `fit` i `transform`. Nie można więc ich wykorzystać do imputacji danych w nowym zbiorze na podstawie wyników uzyskanych podczas trenowania modelu.

Algorytm indukcyjny implementuje klasa `IterativeImputer` (pierwotnie znajdująca się w bibliotece `fancyimpute`, ale później przeniesiona do `scikit-learn`). Aby jej użyć, należy zaimportować specjalną, eksperymentalną bibliotekę (jeżeli używana jest biblioteka `scikit-learn` w wersji 0.21.2 lub nowszej). Ilustruje to poniższy kod:

```
>>> from sklearn.experimental import (
...     enable_iterative_imputer,
... )
>>> from sklearn import impute
>>> num_cols = [
...     "pclass",
...     "age",
...     "sibsp",
...     "parch",
...     "fare",
...     "sex_female",
... ]
>>> imputer = impute.IterativeImputer()
>>> imputed = imputer.fit_transform(
...     X_train[num_cols]
... )
>>> X_train.loc[:, num_cols] = imputed
>>> imputed = imputer.transform(X_test[num_cols])
>>> X_test.loc[:, num_cols] = imputed
```

Aby w imputacji danych wykorzystać medianę, należy użyć biblioteki `pandas`, jak niżej:

```
>>> meds = X_train.median()
>>> X_train = X_train.fillna(meds)
>>> X_test = X_test.fillna(meds)
```

Normalizacja danych

Normalizacja, czyli wstępne przetwarzanie danych, pozwala znacznie poprawić dokładność modelu, szczególnie jeżeli do określania podobieństwa danych wykorzystuje metryki odległości. (Zwróć uwagę, że model drzewa, w którym wszystkie cechy są traktowane niezależnie od siebie, nie wymaga normalizowania danych).

Normalizacja polega na przekształceniu danych tak, aby średnia wartość była równa zeru, a odchylenie standardowe równe jedności. Dzięki temu duże wartości nie są traktowane jako ważniejsze od wartości mniejszych.

W tym przykładzie wyniki uzyskane za pomocą biblioteki `numpy` umieścimy z powrotem w strukturze `DataFrame` biblioteki `pandas`, ponieważ łatwiej będzie je znormalizować (a ponadto zachowane będą nazwy kolumn). Dodatkowo nie trzeba normalizować kolumn pomocniczych, dlatego w poniższym kodzie są one pomijane:

```
>>> cols = "pclass,age,sibsp,fare".split(",")
>>> sca = preprocessing.StandardScaler()
>>> X_train = sca.fit_transform(X_train)
>>> X_train = pd.DataFrame(X_train, columns=cols)
>>> X_test = sca.transform(X_test)
>>> X_test = pd.DataFrame(X_test, columns=cols)
```

Refaktoryzacja kodu

W tym momencie należy zmienić kod. Zazwyczaj tworzę dwie funkcje. Jedna ogólnie oczyszcza dane, a druga dzieli je na zbiory treningowy i testowy. Druga funkcja dodatkowo wprowadza w obu zbiorach niezbędne, odmienne modyfikacje. Poniżej przedstawiony jest kod tych funkcji:

```
>>> def tweak_titanic(df):
...     df = df.drop(
...         columns=[
...             "name",
...             "ticket",
...             "home.dest",
...             "boat",
...             "body",
...             "cabin",
...         ]
...     ).pipe(pd.get_dummies, drop_first=True)
...     return df

>>> def get_train_test_X_y(
...     df, y_col, size=0.3, std_cols=None
... ):
...     y = df[y_col]
...     X = df.drop(columns=y_col)
...     X_train, X_test, y_train, y_test = model_selection.train_test_split(
...         X, y, test_size=size, random_state=42
...     )
...     cols = X.columns
...     num_cols = [
...         "pclass",
...         "age",
...         "sibsp",
...         "parch",
...         "fare",
...     ]
...     fi = impute.IterativeImputer()
...     X_train.loc[
...         :, num_cols
...     ] = fi.fit_transform(X_train[num_cols])
...     X_test.loc[:, num_cols] = fi.transform(
...         X_test[num_cols]
...     )
...
...     if std_cols:
...         std = preprocessing.StandardScaler()
```

```
...         X_train.loc[
...             :, std_cols
...         ] = std.fit_transform(
...             X_train[std_cols]
...         )
...         X_test.loc[
...             :, std_cols
...         ] = std.transform(X_test[std_cols])
...
...     return X_train, X_test, y_train, y_test
>>> ti_df = tweak_titanic(orig_df)
>>> std_cols = "pclass,age,sibsp,fare".split(",")
>>> X_train, X_test, y_train, y_test = get_train_test_X_y(
...     ti_df, "survived", std_cols=std_cols
... )
```

Model odniesienia

Model przetwarzający dane w naprawdę prosty sposób, staje się odniesieniem, z którym można porównywać kolejne modele. Zwróć uwagę, że dokładność modelu określona na podstawie domyślnych wyników metody score może być myląca. Jeżeli na 10 000 wyników tylko jeden jest poprawny, wtedy model z dokładnością 99% będzie zawsze prognozował błędne wyniki. Poniżej przedstawione są przykłady określania dokładności modelu:

```
>>> from sklearn.dummy import DummyClassifier
>>> bm = DummyClassifier()
>>> bm.fit(X_train, y_train)
>>> bm.score(X_test, y_test)  # Dokładność
0.5292620865139949
>>> from sklearn import metrics
>>> metrics.precision_score(
...     y_test, bm.predict(X_test)
... )
0.4027777777777778
```

Różne rodziny algorytmów

W przedstawionym niżej kodzie wykorzystywane są różne rodziny algorytmów. Zasada „nic za darmo" mówi, że nie istnieje jeden uniwersalny algorytm, odpowiedni do wszelkiego rodzaju danych. Są jednak algorytmy dobrze działające na określonych, skończonych zbiorach danych. (Obecnie w dziedzinie danych strukturalnych popularny jest algorytm wzmocnionego drzewa XGBoost).

W tym przykładzie wykorzystane są różne rodziny algorytmów. W drodze *k*-krotnej walidacji porównywane są pola pod krzywą (AUC — ang. *area under curve*) i odchylenia standardowe. Lepszy algorytm to taki, który ma nieco niższą średnią ocenę, ale znacznie mniejsze odchylenie standardowe.

Ponieważ stosowana jest *k*-krotna walidacja, do modelu trzeba wprowadzić wartości `X` i `y` obu zbiorów danych. Poniżej przedstawiony jest kod modelu:

```
>>> X = pd.concat([X_train, X_test])
>>> y = pd.concat([y_train, y_test])
>>> from sklearn import model_selection
>>> from sklearn.dummy import DummyClassifier
>>> from sklearn.linear_model import (
...     LogisticRegression,
... )
>>> from sklearn.tree import DecisionTreeClassifier
>>> from sklearn.neighbors import (
...     KNeighborsClassifier,
... )
>>> from sklearn.naive_bayes import GaussianNB
>>> from sklearn.svm import SVC
>>> from sklearn.ensemble import (
...     RandomForestClassifier,
... )
>>> import xgboost
>>> for model in [
...     DummyClassifier,
...     LogisticRegression,
...     DecisionTreeClassifier,
...     KNeighborsClassifier,
...     GaussianNB,
...     SVC,
...     RandomForestClassifier,
...     xgboost.XGBClassifier,
... ]:
...     cls = model()
...     kfold = model_selection.KFold(
...         n_splits=10, random_state=42
...     )
...     s = model_selection.cross_val_score(
...         cls, X, y, scoring="roc_auc", cv=kfold
...     )
...     print(
...         f"{model.__name__:22}  AUC: "
...         f"{s.mean():.3f} STD: {s.std():.2f}"
...     )
DummyClassifier         AUC: 0.511  STD: 0.04
LogisticRegression      AUC: 0.843  STD: 0.03
DecisionTreeClassifier  AUC: 0.761  STD: 0.03
KNeighborsClassifier    AUC: 0.829  STD: 0.05
GaussianNB              AUC: 0.818  STD: 0.04
SVC                     AUC: 0.838  STD: 0.05
RandomForestClassifier  AUC: 0.829  STD: 0.04
XGBClassifier           AUC: 0.864  STD: 0.04
```

Kontaminacja modeli

Jeżeli celem jest utworzenie jak najdokładniejszego modelu kosztem łatwości interpretowania wyników, można zastosować **kontaminację modeli**. Polega ona na prognozowaniu wartości etykiet na podstawie wyników uzyskiwanych z różnych modeli. W tym przykładzie wykorzystamy uzyskane wcześniej wyniki, połączymy ze sobą i sprawdzimy, czy tak uzyskany nowy klasyfikator jest lepszy:

```
>>> from mlxtend.classifier import (
...     StackingClassifier,
... )
>>> clfs = [
...     x()
...     for x in [
...         LogisticRegression,
...         DecisionTreeClassifier,
...         KNeighborsClassifier,
...         GaussianNB,
...         SVC,
...         RandomForestClassifier,
...     ]
... ]
>>> stack = StackingClassifier(
...     classifiers=clfs,
...     meta_classifier=LogisticRegression(),
... )
>>> kfold = model_selection.KFold(
...     n_splits=10, random_state=42
... )
>>> s = model_selection.cross_val_score(
...     stack, X, y, scoring="roc_auc", cv=kfold
... )
>>> print(
...     f"{stack.__class__.__name__}  "
...     f"AUC: {s.mean():.3f}  STD: {s.std():.2f}"
... )
StackingClassifier  AUC: 0.804  STD: 0.06
```

Jak widać, ocena wyników zwróconych przez nowy model jest nieco gorsza, podobnie jak ich odchylenie standardowe.

Utworzenie modelu

Do utworzenia modelu wykorzystamy klasyfikator drzewa losowego. Jest to elastyczny model, dający dobre wyniki w standardowej konfiguracji. Pamiętaj, że wcześniej należy go przetrenować za pomocą zbioru wyodrębnionego z oryginalnych danych. Poniżej przedstawiony jest kod modelu.

```
>>> rf = ensemble.RandomForestClassifier(
...     n_estimators=100, random_state=42
... )
>>> rf.fit(X_train, y_train)
RandomForestClassifier(bootstrap=True,
   class_weight=None, criterion='gini',
   max_depth=None, max_features='auto',
   max_leaf_nodes=None,
   min_impurity_decrease=0.0,
   min_impurity_split=None,
   min_samples_leaf=1, min_samples_split=2,
   min_weight_fraction_leaf=0.0, n_estimators=10,
   n_jobs=1, oob_score=False, random_state=42,
   verbose=0, warm_start=False)
```

Ocena modelu

Po utworzeniu modelu należy sprawdzić, jak dobrze przetwarza dane, których „nie widział" wcześniej. W tym celu należy zasilić go danymi testowymi. Metoda `score` klasyfikatora zwraca ocenę dokładności prognozowanych wyników. Oczekujemy, że wyniki uzyskane na podstawie danych testowych będą lepsze niż w przypadku danych treningowych. Poniżej przedstawiony jest kod oceniający dokładność modelu:

```
>>> rf.score(X_test, y_test)
0.7964376590330788
```

Oprócz oceny można sprawdzić precyzję modelu:

```
>>> metrics.precision_score(
...     y_test, rf.predict(X_test)
... )
0.8013698630136986
```

Ciekawą właściwością modelu drzewa jest możliwość oceniania ważności cech. Ważność oznacza, jak istotnie dana cecha wpływa na wyniki zwracane przez model. Zwróć uwagę, że po usunięciu wybranej cechy ocena nie musi proporcjonalnie się zmniejszyć, ponieważ mogą istnieć cechy, które są skorelowane z usuniętą. W takim przypadku można niektóre z nich usunąć, tak jak to zrobiliśmy z cechami `sex_male` i `sex_female`, idealnie skorelowanymi ujemnie. Poniżej przedstawiony jest kod wyświetlający ważności cech:

```
>>> for col, val in sorted(
...     zip(
...         X_train.columns,
...         rf.feature_importances_,
...     ),
...     key=lambda x: x[1],
...     reverse=True,
... )[:5]:
...     print(f"{col:10}{val:10.3f}")
```

```
age             0.277
fare            0.265
sex_female      0.240
pclass          0.092
sibsp           0.048
```

Ważność cechy jest wyliczana na podstawie jej wpływu na wielkość błędu wyników. Cecha jest ważna, jeżeli po jej usunięciu błąd modelu wzrasta.

Bardzo lubię oceniać ważność cech i interpretować prognozy za pomocą biblioteki `shap`. Biblioteka ta sprawdza się w modelach typu „czarna skrzynka", o czym piszę w dalszej części książki.

Optymalizacja modelu

Do sterowania działaniem modelu służą **hiperparametry**. Modyfikacja ich wartości wpływa na dokładność modelu. Biblioteka `scikit-learn` udostępnia klasę `GridSearchCV`, oceniającą skuteczność modelu dla różnych kombinacji parametrów i zwracającą najlepszą z nich. Uzyskane wyniki można wykorzystać do zainicjowania modelu. Ilustruje to poniższy kod:

```
>>> rf4 = ensemble.RandomForestClassifier()
>>> params = {
...     "max_features": [0.4, "auto"],
...     "n_estimators": [15, 200],
...     "min_samples_leaf": [1, 0.1],
...     "random_state": [42],
... }
>>> cv = model_selection.GridSearchCV(
...     rf4, params, n_jobs=-1
... ).fit(X_train, y_train)
>>> print(cv.best_params_)
{'max_features': 'auto', 'min_samples_leaf': 0.1,
 'n_estimators': 200, 'random_state': 42}

>>> rf5 = ensemble.RandomForestClassifier(
...     **{
...         "max_features": "auto",
...         "min_samples_leaf": 0.1,
...         "n_estimators": 200,
...         "random_state": 42,
...     }
... )
>>> rf5.fit(X_train, y_train)
>>> rf5.score(X_test, y_test)
0.7888040712468194
```

Aby zoptymalizować model przez dobranie różnych wartości wskaźników, należy w argumencie klasy `GridSearchCV` umieścić parametr `scoring`. Rozdział 12. zawiera listę wskaźników wraz z opisem.

Macierz pomyłek

Macierz pomyłek pokazuje poprawne wyniki klasyfikacji danych, jak również wyniki fałszywie pozytywne i fałszywie negatywne. Stosowanie różnych algorytmów i modyfikowanie ich parametrów pozwala optymalizować model pod kątem wybranych wskaźników. Biblioteka `scikit-learn` pozwala zilustrować macierz pomyłek w formie tekstowej, a biblioteka `Yellowbrick` — w formie graficznej (patrz rysunek 3.4). Poniżej przedstawiony jest przykładowy kod:

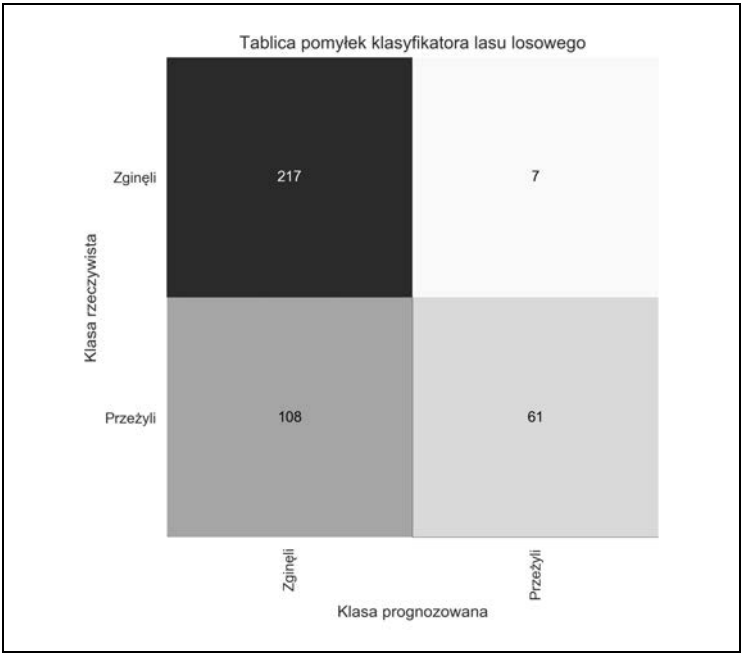
```
>>> from sklearn.metrics import confusion_matrix
>>> y_pred = rf5.predict(X_test)
>>> confusion_matrix(y_test, y_pred)
array([[196,  28],
       [ 55, 114]])
>>> mapping = {0: "Zginęli", 1: "Przeżyli"}
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> cm_viz = ConfusionMatrix(
...     rf5,
...     classes=["Zginęli", "Przeżyli"],
...     label_encoder=mapping,
... )
>>> cm_viz.score(X_test, y_test)

>>> cm_viz.ax.set(title="Tablica pomyłek klasyfikatora lasu losowego",
...     xlabel="Klasa prognozowana", ylabel="Klasa rzeczywista"
... )
>>> fig.savefig(
...     "images/mlpr_0304.png",
...     dpi=300,
...     bbox_inches="tight",
... )
```

Krzywa ROC

Do oceny klasyfikatorów często wykorzystuje się krzywą ROC (ang. *receiver operating characteristic* — charakterystyka operacyjna odbiornika), ilustrującą zależność częstości poprawnych wyników od częstości wyników błędnych. Pole pod krzywą (ang. *area under the curve* — AUC) jest wskaźnikiem wykorzystywanym do porównywania różnych klasyfikatorów (patrz rysunek 3.5). Wskaźnik AUC można wyliczyć za pomocą biblioteki `scikit-learn`. Ilustruje to poniższy kod:

```
>>> y_pred = rf5.predict(X_test)
>>> roc_auc_score(y_test, y_pred)
0.7747781065088757
```



Rysunek 3.4. Macierz pomyłek utworzona za pomocą biblioteki Yellowbrick, czytelnie ilustrująca dokładność modelu. Na osi poziomej znajdują się klasy prognozowanych wyników, a na pionowej klasy wyników rzeczywistych. Jeżeli klasyfikator jest dokładny, wtedy wyniki znajdują się w komórkach na przekątnej macierzy, a wszystkie pozostałe komórki zawierają zera

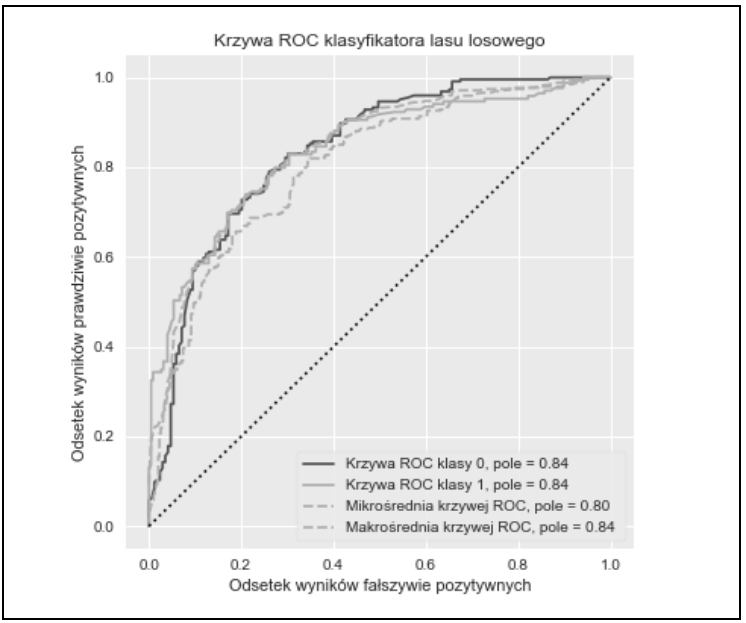
Wyniki w formie graficznej można uzyskać dzięki bibliotece `Yellowbrick`, jak niżej:

```
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> roc_viz = ROCAUC(rf5)
>>> roc_viz.score(X_test, y_test)
0.8279691030696217
>>> roc_viz.ax.set(title="Krzywa ROC klasyfikatora lasu losowego",
...                xlabel="Odsetek wyników fałszywie pozytywnych",
...                ylabel="Odsetek wyników prawdziwie pozytywnych")
>>> ax.legend(("Krzywa ROC klasy {}, pole = {:0.2f}"
...                .format(roc_viz.classes_[0], roc_viz.roc_auc[0]),
...            "Krzywa ROC klasy {}, pole = {:0.2f}"



...                .format(roc_viz.classes_[1], roc_viz.roc_auc[1]),
...            "Mikrośrednia krzywej ROC, pole = {:0.2f}"
```

```
...                 .format(roc_viz.roc_auc["micro"]),
...             "Makrośrednia krzywej ROC, pole = {:0.2f}"
...                 .format(roc_viz.roc_auc["macro"])),
...             frameon=True, loc="lower right")
>>> fig.savefig("images/mlpr_0305.png")
```

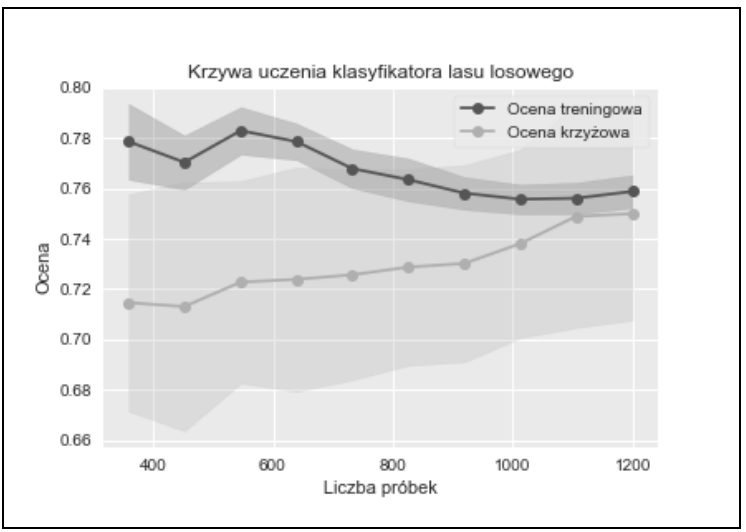


Rysunek 3.5. Krzywe ROC przedstawiające zależności częstości poprawnych wyników od częstości wyników błędnych. Im bardziej wypukła jest krzywa, tym model jest lepszy. Oceną jest pole pod krzywą. Im bliższe jest ono jedności, tym model jest dokładniejszy. Wynik mniejszy niż 0,5 oznacza, że model jest niedokładny

Krzywa uczenia

Krzywa uczenia pozwala ocenić, czy model został wystarczająco przetrenowany. Ilustruje zależność oceny od ilości danych treningowych (patrz rysunek 3.6). Jeżeli ocena krzyżowa rośnie wraz z ilością danych, należy powiększyć zbiór treningowy. Poniżej przedstawiony jest przykład wykreślenia krzywej uczenia za pomocą biblioteki `Yellowbrick`:

```
>>> import numpy as np
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> cv = StratifiedKFold(12)
>>> sizes = np.linspace(0.3, 1.0, 10)
>>> lc_viz = LearningCurve(
...     rf5,
...     cv=cv,
...     train_sizes=sizes,
...     scoring="f1_weighted",
...     n_jobs=4,
...     ax=ax,
... )
>>> lc_viz.fit(X, y)
>>> ax.legend(("Ocena treningowa", "Ocena krzyżowa"), frameon=True,
loc="best")
>>> lc_viz.ax.set(title="Krzywa uczenia klasyfikatora lasu losowego",
...     xlabel="Liczba próbek", ylabel="Ocena"
... )
>>> fig.savefig("images/mlpr_0306.png")
```



Rysunek 3.6. Krzywa uczenia pozwala ocenić, czy w miarę zwiększania ilości danych treningowych poprawia się ocena krzyżowa modelu

Wdrożenie modelu

Za pomocą modułu `pickle` można zapisywać i ładować modele danych. Aby po utworzeniu modelu uzyskać wyniki klasyfikacji lub regresji, należy użyć metody `predict`, jak w poniższym przykładzie:

```
>>> import pickle
>>> pic = pickle.dumps(rf5)
>>> rf6 = pickle.loads(pic)
>>> y_pred = rf6.predict(X_test)
>>> roc_auc_score(y_test, y_pred)
0.7747781065088757
```

Bardzo często model wdraża się w formie usługi WWW za pomocą biblioteki `Flask` (*https://palletsprojects.com/p/flask*). Dostępnych jest również wiele płatnych i darmowych produktów przeznaczonych do tego celu. Są to m.in.: Clipper (*http://clipper.ai*), Pipeline (*https://oreil.ly/UfHdP*) i Google Cloud Machine Learning Engine (*https://oreil.ly/1qYkH*).


PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA
Helion

Uczenie maszynowe: nowy wymiar analizy danych!

Uczenie maszynowe i nauka o danych szybko się rozwijają, a poszczególne techniki znajdują coraz więcej różnorodnych zastosowań. Wiedza, którą można uzyskać dzięki odpowiedniemu przygotowaniu danych i ich eksploracji, często jest bezcenna. Umiejętność ich analizy oraz wiedza o możliwych sposobach rozwiązywania problemów napotykanych podczas uczenia maszynowego są dużymi atutami i mogą być wykorzystywane w wielu gałęziach nauki, techniki i biznesu.

Z tego zwięzłego przewodnika skorzystają programiści, badacze, osoby zajmujące się nauką o danych oraz twórcy systemów sztucznej inteligencji. Znalazł się tu wyczerpujący opis procesu uczenia maszynowego i klasyfikacji danych strukturalnych. Przedstawiono też metody klastrowania danych, analizy regresji, redukcji wymiarowości oraz inne ważne zagadnienia. Prezentowane treści zostały zilustrowane uwagami, tabelami i przykładami kodu.

W książce między innymi:

- klasyfikacja, oczyszczanie i uzupełnianie braków danych
- eksploracyjna analiza danych i dobór modelu danych
- przykłady analiz regresji
- redukcja wymiarowości
- potoki w bibliotece scikit-learn

Matt Harrison programuje w Pythonie od dwudziestu lat. Wykorzystuje go do różnych zastosowań związanych z gromadzeniem i analizą danych, tworzeniem i automatyzowaniem procesów czy też budowaniem systemów sztucznej inteligencji.

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 978-83-283-6558-2

Cena: 44,90 zł