O'REILLY®

Helion

# Spark

## Rozproszone uczenie maszynowe na dużą skalę

### Jak korzystać z MLlib, TensorFlow i PyTorch

Adi Polak

Tytuł oryginału: Scaling Machine Learning with Spark: Distributed ML with MLlib, TensorFlow, and PyTorch

Tłumaczenie: Radosław Meryk

ISBN: 978-83-289-1234-2



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/sparkr*
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

# Spis treści

**Przedmowa** ..... **11**

**1. Rozproszone uczenie maszynowe. Terminologia i pojęcia** ..... **19**
Etapy przepływu pracy uczenia maszynowego 22
Narzędzia i technologie w potoku uczenia maszynowego 24
Modele przetwarzania rozproszonego 25
Modele uniwersalne 26
Dedykowane modele przetwarzania rozproszonego 28
Wprowadzenie do architektury systemów rozproszonych 28
Systemy scentralizowane a zdecentralizowane 29
Modele interakcji 30
Komunikacja w środowisku rozproszonym 31
Wprowadzenie do metod uczenia zespołowego 32
Wysoka i niska stronniczość 32
Rodzaje metod zespołowych 33
Topologie szkolenia rozproszonego learner 34
Wyzwania związane z rozproszonymi systemami uczenia maszynowego 36
Wydajność 36
Zarządzanie zasobami 39
Odporność na błędy 40
Prywatność 41
Przenośność 42
Konfiguracja środowiska lokalnego 42
Środowisko samouczków z rozdziałów 2. – 6. 42
Środowisko samouczków z rozdziałów 7. – 10. 44
Podsumowanie 45

**2. Wprowadzenie do Sparka i PySparka** ..... **46**
Architektura Apache Spark 46
Wprowadzenie do PySparka 49

Podstawy Apache Spark 50
Architektura oprogramowania 50
PySpark a programowanie funkcyjne 56
Uruchamianie kodu PySparka 57
Ramki DataFrame biblioteki pandas kontra ramki DataFrame systemu Spark 57
Scikit-Learn kontra MLlib 58
Podsumowanie 59

**3. Zarządzanie cyklem życia eksperymentu uczenia maszynowego za pomocą MLflow ..... 60**
Wymagania dotyczące zarządzania cyklem życia uczenia maszynowego 61
Czym jest MLflow? 62
Komponenty oprogramowania platformy MLflow 63
Użytkownicy platformy MLflow 64
Komponenty platformy MLflow 65
MLflow Tracking 65
MLflow Projects 68
MLflow Models 69
MLflow Model Registry 70
Korzystanie z platformy MLflow w rozwiązaniach dużej skali 71
Podsumowanie 74

**4. Pozyskiwanie danych, wstępne przetwarzanie i statystyki opisowe ..... 75**
Pozyskiwanie danych za pomocą Sparka 76
Przetwarzanie obrazów 77
Przetwarzanie danych tabelarycznych 79
Wstępne przetwarzanie danych 79
Przetwarzanie wstępne a właściwe 80
Po co wstępnie przetwarzać dane? 80
Struktury danych 81
Typy danych MLlib 81
Przetwarzanie wstępne z wykorzystaniem transformatorów MLlib 83
Wstępne przetwarzanie danych obrazów 89
Zapisywanie danych i unikanie problemu małych plików 92
Statystyki opisowe: poznawanie danych 94
Obliczanie statystyk 95
Statystyki opisowe z wykorzystaniem obiektu Summarizer Sparka 95
Skośność danych 98
Korelacja 98
Podsumowanie 102

**5. Inżynieria cech ..... 103**
Cechy i ich wpływ na modele uczenia maszynowego 104
Narzędzia do cechowania w bibliotece MLlib 107
Ekstraktory 108
Selektory 108
Przykład: Word2Vec 109
Proces cechowania obrazów 111
Wykonywanie działań na obrazach 112
Wyodrębnianie cech za pomocą API Sparka 114
Proces cechowania tekstu 119
Worek słów 120
TF-IDF 121
n-gramy 122
Techniki dodatkowe 122
Wzbogacanie zbioru danych 123
Podsumowanie 124

**6. Szkolenie modeli za pomocą biblioteki MLlib platformy Spark ..... 125**
Algorytmy 125
Nadzorowane uczenie maszynowe 127
Klasyfikacja 127
Regresja 131
Nienadzorowane uczenie maszynowe 136
Wydobywanie częstych wzorców 136
Klasteryzacja 136
Ocena 139
Ewaluatory nadzorowane 140
Ewaluatory nienadzorowane 142
Hiperparametry i eksperymenty dostrajania 143
Budowanie siatki parametrów 143
Podział danych na zbiory szkoleniowe i testowe 144
Walidacja krzyżowa: lepszy sposób testowania modeli 145
Potoki uczenia maszynowego 146
Budowa potoku 148
Jak działa podział dla API Pipeline? 148
Utrwalanie 149
Podsumowanie 149

**7. Łączenie Sparka z frameworkami uczenia głębokiego ..... 150**
Podejście oparte na danych i dwóch klastrach 153
Implementacja dedykowanej warstwy dostępu do danych 155
Cechy DAL 155
Wybór warstwy DAL 157

Czym jest Petastorm? 157
SparkDatasetConverter 159
Petastorm jako magazyn Parquet 163
Projekt Hydrogen 164
Barierowy tryb wykonania 165
Harmonogramowanie z uwzględnieniem akceleratorów 166
Wprowadzenie do API Horovod Estimator 167
Podsumowanie 168

**8. Rozproszone uczenie maszynowe z wykorzystaniem TensorFlow ..... 170**
Przegląd podstawowych wywołań API biblioteki TensorFlow 171
Czym jest sieć neuronowa? 173
Role i obowiązki w procesie klastra TensorFlow 174
Ładowanie danych Parquet do zbioru danych TensorFlow 175
Strategie rozproszonego uczenia maszynowego TensorFlow 177
ParameterServerStrategy 179
CentralStorageStrategy: jedna maszyna, wiele procesorów 181
MirroredStrategy: jedna maszyna, wiele procesorów, lokalna kopia 181
MultiWorkerMirroredStrategy: wiele maszyn, tryb synchroniczny 182
TPUStrategy 186
Co się zmienia po zmianie strategii? 186
Szkoleniowe interfejsy API 187
API Keras 187
Niestandardowa pętla szkoleniowa 191
API Estimator 193
Połączmy kropki 194
Rozwiązywanie problemów 196
Podsumowanie 197

**9. Rozproszone uczenie maszynowe z wykorzystaniem frameworka PyTorch ..... 198**
Przegląd podstaw frameworka PyTorch 199
Graf obliczeniowy 199
Mechanika frameworka PyTorch i związane z nim pojęcia 201
Strategie rozproszonego szkolenia modeli frameworka PyTorch 204
Wprowadzenie do podejścia rozproszonego wykorzystywanego przez framework PyTorch 205
Rozproszone i równoległe szkolenie danych (DDP) 206
Szkolenie rozproszone oparte na RPC 207
Topologie komunikacji frameworka PyTorch (c10d) 215
Do czego można wykorzystać niskopoziomowe wywołania API frameworka PyTorch? 223

Ładowanie danych za pomocą frameworka PyTorch i biblioteki Petastorm 224
Rozwiązywanie problemów podczas korzystania z biblioteki Petastorm i frameworka PyTorch w środowisku rozproszonym 227
Enigma niedopasowanych typów danych 227
Tajemnica marudnych węzłów roboczych 228
Czym PyTorch różni się od TensorFlow? 229
Podsumowanie 230

**10. Wzorce wdrażania modeli uczenia maszynowego ..... 231**
Wzorce wdrażania 232
Wzorzec 1. Prognozy zbiorcze 232
Wzorzec 2. Model w ramach usługi 233
Wzorzec 3. Model jako usługa 234
Decydowanie o wykorzystywanym wzorcu 235
Wymagania dotyczące oprogramowania produkcyjnego 236
Monitorowanie modeli uczenia maszynowego w produkcji 239
Dryf danych 240
Dryf modelu, dryf koncepcji 243
Przesunięcie dziedziny rozkładu (długi ogon) 244
Jakie wskaźniki należy monitorować w produkcji? 244
W jaki sposób wykorzystać system monitorowania do mierzenia zmian? 245
Jak to wygląda w systemie produkcyjnym? 247
Produkcyjna pętla sprzężenia zwrotnego 248
Wdrażanie z wykorzystaniem biblioteki MLlib 248
Produkcyjne potoki uczenia maszynowego ze strukturalnym przesyłaniem strumieniowym 249
Wdrażanie z wykorzystaniem biblioteki MLflow 251
Definiowanie wrappera MLflow 251
Wdrażanie modelu jako mikrousługi 254
Ładowanie modelu jako funkcji UDF platformy Spark 255
Jak pracować nad systemem w sposób iteracyjny? 255
Podsumowanie 256

**Skorowidz ..... 259**

ROZDZIAŁ 6.

# Szkolenie modeli za pomocą biblioteki MLlib platformy Spark

Po zapoznaniu się ze sposobami zarządzania eksperymentami związanymi z uczeniem maszynowym, danymi i inżynierią cech nadszedł czas na przeszkolenie wybranych modeli.

Co to dokładnie oznacza? **Szkolenie** modelu to proces dostosowywania lub modyfikacji parametrów modelu w celu poprawy jego wydajności. Chodzi o to, aby zasilić model uczenia maszynowego danymi szkoleniowymi, dzięki którym model nauczy się rozwiązywać określone zadanie — na przykład dowie się, jak sklasyfikować obiekt na zdjęciu jako kota na podstawie identyfikacji jego „kocich" cech.

W tym rozdziale dowiesz się, jak działają algorytmy uczenia maszynowego, kiedy użyć jakiego narzędzia, jak zweryfikować model i, co najważniejsze, jak zautomatyzować proces za pomocą API `Pipelines` biblioteki MLlib Sparka.

Na wysokim poziomie ten rozdział obejmuje następujące kwestie:

- Podstawowe algorytmy uczenia maszynowego Sparka.
- Nadzorowane uczenie maszynowe z wykorzystaniem uczenia maszynowego Sparka.
- Nienadzorowane uczenie maszynowe z wykorzystaniem mechanizmów uczenia maszynowego Sparka.
- Ocena modelu i jego testowanie.
- Hiperparametry i dostrajanie modelu.
- Korzystanie z potoków uczenia maszynowego platformy Spark.
- Utrwalanie modeli i potoków na dysku.

## Algorytmy

Zacznijmy od algorytmów, zasadniczej części operacji związanych ze szkoleniem modeli. Dane wejściowe algorytmu uczenia maszynowego to przykładowe dane, a dane wyjściowe algorytmu to model. Celem algorytmu jest uogólnienie problemu i wyodrębnienie zestawu logiki umożliwiającej prognozowanie i podejmowanie decyzji bez konieczności bezpośredniego programowania. Algorytmy mogą opierać się na statystykach, optymalizacji matematycznej, wykrywaniu wzorców itd. Biblioteka

MLlib Sparka zapewnia rozproszone implementacje szkolenia dla klasycznych **nadzorowanych** algorytmów uczenia maszynowego, takich jak klasyfikacja, regresja i rekomendacje. Obejmuje również implementacje **nienadzorowanych** algorytmów uczenia maszynowego bez nadzoru, takich jak klasteryzacja i wyodrębnianie wzorców, które są często wykorzystywane do wykrywania anomalii.

Warto zauważyć, że w chwili pisania tego tekstu nie było parytetu funkcji między interfejsami API opartymi na MLlib RDD (*https://oreil.ly/r0MAS*) i opartymi na `DataFrame` (*https://oreil.ly/lQOLE*), dlatego mogą zdarzyć się przypadki, w których potrzebna funkcjonalność będzie dostępna wyłącznie w interfejsie API opartym na RDD. Jednym z przykładów jest rozkład wartości osobliwych (ang. *singular value decomposition* — SVD).

Jak wybrać odpowiedni algorytm do zadania? Wybór zawsze zależy od wyznaczonych celów i danych.

W tym rozdziale omówię wiele algorytmów i związane z nimi przypadki użycia. Z kolei tematy związane z uczeniem głębokim, integracją z PyTorch, jak również dotyczące rozproszonych strategii TensorFlow zostaną omówione w rozdziałach 7. i 8.

Chciałabym zwrócić uwagę na fakt, że egzemplarz modelu MLlib ma dedykowaną funkcjonalność do dokumentowania parametrów. Sposób natychmiastowego uzyskania dostępu do dokumentacji poszczególnych parametrów po utworzeniu egzemplarza modelu ilustruje poniższy przykład kodu:

```
import pprint
pp = pprint.PrettyPrinter(indent=4)
params = model.explainParams()
pp.pprint(params)
```

Przykładowe dane wyjściowe funkcji `model.explainParams` pokazano na listingu 6.1. Ponieważ jest to model `GaussianMixture` (omówiony we fragmencie poświęconym `Gaussian Mixture` w punkcie „Klasteryzacja”), zawiera opisy parametrów dostępnych do dostrajania z tym typem modelu. Jest to świetne narzędzie, które pomoże Ci zapoznać się z algorytmami MLlib i dowiedzieć się więcej o każdym z nich oraz zwracanych przez niego wynikach.

*Listing 6.1. Przykład estetycznego wyświetlania parametrów modelu GaussianMixture*

```
('aggregationDepth: suggested depth for treeAggregate (>= 2). (default: 2)\n'
 'featuresCol: features column name. (default: features, current: '
 'selectedFeatures)\n'
 'k: Number of independent Gaussians in the mixture model. Must be > 1. '
 '(default: 2, current: 42)\n'
 'maxIter: max number of iterations (>= 0). (default: 100, current: 100)\n'
 'predictionCol: prediction column name. (default: prediction)\n'
 'probabilityCol: Column name for predicted class conditional probabilities. '
 'Note: Not all models output well-calibrated probability estimates! These '
 'probabilities should be treated as confidences, not precise probabilities. '
 '(default: probability)\n'
 'seed: random seed. (default: 4621526457424974748, current: 10)\n'
 'tol: the convergence tolerance for iterative algorithms (>= 0). (default: '
 '0.01, current: 0.01)\n'
 'weightCol: weight column name. If this is not set or empty, we treat all '
 'instance weights as 1.0. (undefined)')
```

Po omówieniu podstaw przejdziemy do omawiania technik nadzorowanego uczenia maszynowego. Zaczynajmy!

# Nadzorowane uczenie maszynowe

Wszystkie algorytmy nadzorowane oczekują występowania w danych kolumny `label` oznaczającej etykietę danych. Dzięki temu algorytm może sam się „zwalidować" w fazie uczenia i oszacować swoją wydajność. Innymi słowy, algorytm wykorzystuje kolumnę `label` do korygowania swoich decyzji. W fazie testowania, dzięki porównywaniu prognoz modelu z rzeczywistymi wynikami, wykorzystujemy ją do oceny jakości algorytmu. Etykieta może być zmienną dyskretną (kategorialną) — czyli konkretną wartością ze zbioru wszystkich możliwych wartości, na przykład jabłko w przypadku kategoryzowania jabłek i pomarańczy. Może być też zmienną ciągłą, na przykład wzrost bądź wiek osoby. Ta różnica określa zadania, jakie chcemy rozwiązać za pomocą modelu: klasyfikację czy regresję.

W niektórych przypadkach sama etykieta może być zbiorem etykiet. O tej opcji porozmawiamy w następnym punkcie.

## Klasyfikacja

**Klasyfikacja** to zadanie polegające na obliczeniu prawdopodobieństwa przynależności punktu danych do odrębnych kategorii bądź klas poprzez zbadanie jego cech wejściowych. Wynikiem tego procesu jest prognozowanie prawdopodobieństwa przynależności punktu danych do każdej możliwej kategorii. Ze względu na istnienie algorytmu **regresji logistycznej**, często używanego do klasyfikacji binarnej, wiele osób myli regresję z klasyfikacją. Podczas gdy w wyniku regresji logistycznej wyznaczamy prawdopodobieństwo dla dyskretnej klasy, tak jak w przypadku algorytmów klasyfikacyjnych, inne algorytmy regresji służą do prognozowania ciągłych wartości liczbowych. Należy zwrócić uwagę na tę różnicę!

Istnieją trzy rodzaje klasyfikacji:

**Binarna**
: Każda dana wejściowa jest klasyfikowana w jednej z dwóch klas (tak lub nie, prawda lub fałsz itp.).

**Wieloklasowa**
: Każde dana wejściowa jest przyporządkowywana do jednej klasy ze zbioru złożonego z więcej niż dwóch klas.

**Wieloetykietowa**
: W praktyce do każdej danej wejściowej można przypisać wiele etykiet. Na przykład zdanie może mieć dwie klasyfikacje nastrojów, *zadowolony* i *spełniony*. Spark nie ma wbudowanej obsługi takiej klasyfikacji. Aby z niej skorzystać, należy przeszkolić każdy klasyfikator osobno i połączyć wyniki.

Ponadto na proces klasyfikacji wpływa rozkład klas w danych szkoleniowych. Mówi się, że etykiety danych są **niezrównoważone**, gdy dane wejściowe są nierównomiernie rozmieszczone pomiędzy klasami. Z taką sytuacją często można się spotkać w przypadkach użycia związanych z wykrywaniem oszustw i diagnostyką medyczną. W takich scenariuszach należy, o ile to możliwe, rozważyć poszczególne cechy i przypisać im odpowiednie wagi. Niezrównoważenie może również pojawić się w zbiorach szkoleniowych, walidacyjnych i testowych: aby zapewnić pożądane wyniki, wszystkie trzy

zbiory muszą zostać zrównoważone. Bliżej problemowi scenariuszy klasyfikacji wieloetykietowej zajmę się w kolejnych punktach.

### Algorytmy klasyfikacji biblioteki MLlib

Algorytmy klasyfikacji oczekują indeksowanej etykiety (często w przedziale `[0,1]`) oraz wektora indeksowanych cech. Interfejsy API służące do przekształcania cech kategorialnych na indeksy, takie jak `StringIndexer` i `VectorIndexer`, zostały omówione w rozdziale 4.

Biblioteka MLlib implementuje kilka popularnych algorytmów klasyfikacji, wymienionych w tabeli 6.1. Wzorzec nazwy klasy to zazwyczaj {*nazwa*}`Classifier` lub po prostu {*nazwa*}. Po szkoleniu klasyfikatory tworzą model o odpowiedniej nazwie: {*nazwa*}`ClassificationModel` lub {*nazwa*}`Model`. Na przykład klasyfikator `GBTClassifier` z biblioteki MLlib odpowiada modelowi `GBTClassification➥Model`, natomiast klasyfikator `NaiveBayes` odpowiada modelowi `NaiveBayesModel`.

*Tabela 6.1. Algorytmy klasyfikacji MLlib*

| Wywołanie API | Wykorzystanie |
|---|---|
| `LogisticRegression` | Klasyfikator binarny i wieloklasowy. Można go przeszkolić na danych strumieniowych za pomocą interfejsu API opartego na RDD. Oczekuje indeksowanej etykiety i wektora indeksowanych cech. |
| `DecisionTreeClassifier` | Klasyfikator binarny i wieloklasowy w postaci drzewa decyzyjnego. Oczekuje indeksowanej etykiety i wektora indeksowanych cech. |
| `RandomForestClassifier` | Klasyfikator binarny i wieloklasowy. Las losowy to grupa lub zestaw pojedynczych drzew decyzyjnych, z których każde jest przeszkolone na dyskretnych wartościach. Oczekuje indeksowanej etykiety i wektora indeksowanych cech. |
| `GBTClassifier` | Klasyfikator drzew ze wzmocnionym gradientem binarnym (obsługiwany na platformie Spark w wersji 3.1.1 i nowszych). Podobnie jak klasyfikator `RandomForestClassifier`, jest to zbiór drzew decyzyjnych. Proces szkolenia tego klasyfikatora jest jednak inny. W rezultacie można go również wykorzystać do regresji. Oczekuje indeksowanej etykiety i wektora indeksowanych cech. |
| `MultilayerPerceptronClassifier` | Klasyfikator wieloklasowy oparty na sztucznej sieci neuronowej ze sprzężeniem zwrotnym. Oczekuje warstwy rozmiarów, wektora poindeksowanych cech i poindeksowanych etykiet. |
| `LinearSVC` | Klasyfikator maszynowy liniowych wektorów nośnych (binarny). Oczekuje indeksowanej etykiety i wektora indeksowanych cech. |
| `OneVsRest` | Służy do redukcji klasyfikacji wieloklasowej do klasyfikacji binarnej przy użyciu strategii „jeden kontra wszyscy”. Oczekuje klasyfikatora binarnego, wektora indeksowanych cech i indeksowanych etykiet. |
| `NaiveBayes` | Klasyfikator wieloklasowy, uważany za wydajny, ponieważ obsługuje tylko jedno przejście przez dane szkoleniowe. Oczekuje wartości `double` dla wagi punktu danych (w celu skorygowania skośnego rozkładu etykiet), indeksowanych etykiet i wektora indeksowanych cech. Zwraca prawdopodobieństwo dla każdej etykiety. |
| `FMClassifier` | Klasyfikator maszyn binarnych do faktoryzacji. Oczekuje indeksowanych etykiet i wektora indeksowanych cech. |

### Implementacja obsługi klasyfikacji wieloetykietowej

Biblioteka MLlib nie ma wbudowanej obsługi klasyfikacji wieloetykietowej, ale ten problem można obejść na kilka sposobów:

1. Wyszukaj inne narzędzie, które wykonuje taką klasyfikację i które może przeprowadzać szkolenie na dużym zbiorze danych.
2. Przeprowadź szkolenie klasyfikatorów binarnych dla każdej etykiety i wygeneruj klasyfikacje wieloetykietowe poprzez uruchomienie dla każdej z nich prognozy istotnej i nieistotnej.
3. Zastanów się, jak wykorzystać istniejące narzędzia poprzez podzielenie zadania na części i rozwiązanie każdego podzadania niezależnie, a następnie połączenie wyników za pomocą kodu.

Dobra wiadomość, jeśli chodzi o pierwszą opcję, jest taka, że zarówno biblioteka PyTorch, jak i TensorFlow obsługują algorytmy klasyfikacji wieloetykietowej. Dzięki temu, w przypadku użycia wielu etykiet, możemy skorzystać z ich możliwości.

Druga opcja może być wykorzystana przez inżynierów AI lub doświadczonych programistów Sparka. Możesz skorzystać z bogatego interfejsu API pozwalającego na wykonanie następujących kroków:

1. Dodaj do istniejącej ramki `DataFrame` wiele kolumn, z której każda reprezentuje określoną etykietę. Na przykład jeśli pierwotnie ramka `DataFrame` zawierała jedynie kolumny `id`, `zdanie` i `nastrój`, możesz dodać każdą kategorię nastroju jako osobną kolumnę. Wiersz z wartością `[zadowolony]` w kolumnie `nastrój` otrzymałby wartość `1.0` w nowej kolumnie o nazwie `jest_zadowolony` i `0.0` w kolumnach `jest_obojętny`, `jest_spełniony` i `jest_smutny`; wiersz zawierający w kolumnie `nastrój` wartość `[zadowolony, obojętny]` otrzymałby wartość `1.0` w kolumnach `jest_zadowolony` i `jest_obojętny` oraz `0.0` w pozostałych. W ten sposób można sklasyfikować zdanie jako należące do wielu kategorii. Ideę tę zaprezentowano na rysunku 6.1.

```
+---+--------------------+--------------------+---------------+-------------+--------------+-----------+
| id|             zdanie|             nastrój|jest_zadowolony|jest_obojętny|jest_spełniony|jest_smutny|
+---+--------------------+--------------------+---------------+-------------+--------------+-----------+
|  0|Cześć Myślę, że p...|        [zadowolony]|            1.0|          0.0|           0.0|        0.0|
|  1|Wszystko,czego ch...|          [obojętny]|            0.0|          1.0|           0.0|        0.0|
|  2|Wreszcie zrozumia...|[spełniony, zadow...|            1.0|          0.0|           1.0|        0.0|
|  3|Jeszcze jedno zda...|[zadowolony, oboj...|            1.0|          1.0|           0.0|        0.0|
|  4|Dlaczego wcześnie...|  [smutny, obojętny]|            0.0|          1.0|           0.0|        1.0|
|  5|           Tak, mogę|        [zadowolony]|            1.0|          0.0|           0.0|        0.0|
+---+--------------------+--------------------+---------------+-------------+--------------+-----------+
```

*Rysunek 6.1. Przykład danych wyjściowych ramki DataFrame dla klasyfikacji wieloetykietowej*

2. Kontynuuj proces cechowania dla każdej etykiety. W repozytorium GitHub tej książki (*https://oreil.ly/smls-git*) jest dostępny kod wykorzystujący ramki `HashingTF`, `IDF` i inne metody pozwalające na przygotowanie ramki `DataFrame` do uczenia klasyfikatora, tak jak pokazano na listingu 6.2.

*Listing 6.2. Ramka DataFrame gotowa do szkolenia pierwszego klasyfikatora dla etykiety zadowolony*

```
+----------------------------------------+--------------+
|cechy                                   |zadowolony_lbl|
+----------------------------------------+--------------+
|(65536,[16887,26010],[0.0,0.0])         |0.0           |
|(65536,[575871,[0.0])                   |1.0           |
|(65536,[34782,397581,[0.0,0.0])         |0.0           |
|(65536,[11730,34744,49304],[0.0,0.0,0.0])|0.0           |
|(65536,[],[])                           |1.0           |
+----------------------------------------+--------------+
only showing top 5 rows
```

3. Zbuduj klasyfikator binarny dla każdej etykiety. Poniższy fragment kodu pokazuje, jak zbudować klasyfikator `LogisticRegression` po przekształceniach polegających na dodawaniu kolumn i indeksowaniu:

```
from pyspark.ml.classification import LogisticRegression

happy_lr = LogisticRegression(maxIter=10, labelCol="zadowolony_lbl")
happy_lr_model = happy_lr.fit(train_df)
```

   Ten sam proces należy zastosować do wszystkich pozostałych etykiet.

Należy zapamiętać, że potok uczenia maszynowego składa się z większej liczby kroków, na przykład obejmuje ocenę wyników z wykorzystaniem testowego zestawu danych oraz utworzonych wcześniej klasyfikatorów.

Aby przetestować model, należy wywołać funkcję `transform` na testowej ramce `DataFrame`:

```
result = happy_lr_model.transform(test_dataframe)
```

Wynik testowania modelu pokazano na listingu 6.3.

*Listing 6.3. Macierz korelacji Spearmana*

```
Macierz korelacji Spearmana:
DenseMatrix([[ 1.        , -0.41076061, -0.22354106,  0.03158624],
             [-0.41076061,  1.        , -0.15632771,  0.16392762],
             [-0.22354106, -0.15632771,  1.        , -0.09388671],
             [ 0.03158624,  0.16392762, -0.09388671,  1.        ]])
```

Jest to omówiony w rozdziale 4. obiekt `DenseMatrix`, pozwalający zrozumieć wynik prognozy `LogisticRegression`. Można go znaleźć w kolumnie `rawPrediction` (wektor liczb `double` oznaczających miarę pewności dla każdej z możliwych etykiet) ramki `DataFrame` prognoz. Za nią, w kolumnie `probability`, następuje wektor prawdopodobieństw (prawdopodobieństwo warunkowe dla każdej klasy) i sama prognoza w kolumnie `prediction`. Należy pamiętać, że nie wszystkie modele zwracają dokładne prawdopodobieństwa; dlatego z wektora prawdopodobieństwa należy korzystać ostrożnie.

## Co z niezrównoważonymi etykietami klas?

Jak wspomniano wcześniej, niezrównoważone dane w zadaniach klasyfikacyjnych mogą stwarzać problem. Istnienie jednej etykiety klasy z bardzo dużą liczbą obserwacji i drugiej z bardzo małą liczbą obserwacji może spowodować powstanie modelu stronniczego. Stronniczość będzie dotyczyła

etykiety klasy z większą liczbą obserwacji, ponieważ statystycznie jest ona bardziej dominująca w zbiorze danych szkoleniowych.

Stronniczość mogła być wprowadzona na różnych etapach rozwoju modelu. Do stronniczości w decyzjach podejmowanych przez model mogą prowadzić niewystarczające dane, niespójny proces zbierania danych oraz złe praktyki postępowania z danymi. Nie będziemy się tutaj zagłębiać w sposoby rozwiązywania istniejących problemów ze stronniczością modelu. Zamiast tego skoncentrujemy się na strategiach pracy ze zbiorem danych w celu złagodzenia potencjalnych źródeł stronniczości. Mogą to być następujące strategie:

1. Filtrowanie bardziej reprezentatywnych klas i próbkowanie ich w celu zmniejszenia liczby wpisów w całym zbiorze danych.
2. Korzystanie z algorytmów zespołowych opartych na drzewach decyzyjnych, takich jak `GBT` `↪Classifier`, `GBTRegressor` i `RandomForestClassifier`. Podczas procesu uczenia w tych algorytmach można ustawić dedykowany parametr `featureSubsetStrategy`. Dla tego parametru są dostępne wartości `auto`, `all`, `sqrt`, `log2` i `onethird`. Przy domyślnym ustawieniu `auto` algorytm wybiera najlepszą strategię na podstawie wybranego zbioru cech. W każdym węźle drzewa algorytm przetwarza losowy podzbiór cech i wykorzystuje wynik do zbudowania kolejnego węzła. Wykonuje iterację zgodnie z tą samą procedurą, aż do zakończenia korzystania ze wszystkich zbiorów danych. Jest to przydatne ze względu na losowe podejście do parametrów, ale w zależności od rozkładu obserwacji w wynikowym modelu nadal może występować stronniczość. Wyobraź sobie, że masz zbiór danych dotyczący jabłek i pomarańczy zawierający 99 jabłek i 1 pomarańczę. Załóżmy, że w losowym procesie algorytm wybiera partię 10 jednostek. Wybrana partia będzie zawierać co najwyżej 1 pomarańczę i 9 lub 10 jabłek. Rozkład jest mocno przekrzywiony w stronę jabłek, zatem model prawdopodobnie w 100% przypadków będzie prognozował jabłko. Jest to poprawna prognoza w przypadku zbioru danych szkoleniowych, ale dla danych produkcyjnych może być zupełnie nietrafiona. Więcej informacji na ten temat można przeczytać w dokumentacji (*https://oreil.ly/6sFNX*).

Oto jak ustawić strategię:

```
from pyspark.ml.classification import RandomForestClassifier
# przeszkolenie modelu RandomForestClassifier ze strategią podzbioru
# z dedykowaną cechą
rf = RandomForestClassifier(labelCol="label", featuresCol="cechy",
                            featureSubsetStrategy="log2")
model = rf.fit(train_df)
```

## Regresja

Czas nauczyć się regresji! Zadanie to jest również znane jako **analiza regresji** — proces szacowania zależności między jedną lub większą liczbą zmiennych zależnych a jedną lub większą liczbą zmiennych niezależnych. Wartości zmiennych niezależnych powinny umożliwiać prognozowanie wartości zmiennych zależnych. Jeśli tak nie jest, możesz użyć wywołań API omówionych w rozdziale 5., aby wybrać tylko te cechy, które dodają wartość.

Ogólnie rzecz biorąc, można wyróżnić trzy rodzaje regresji:

**Prosta**

Jest tylko jedna zmienna niezależna i jedna zmienna zależna: jedna wartość do szkolenia i jedna do prognozowania.

**Wielokrotna**

W tego rodzaju regresji występuje jedna zmienna zależna do prognozowania oraz wiele zmiennych niezależnych do szkolenia i wprowadzania danych.

**Wielozmienna**

Podobnie jak w przypadku klasyfikacji wieloetykietowej, istnieje wiele zmiennych do prognozowania oraz wiele zmiennych niezależnych do szkolenia i wprowadzania danych. Wejście i wyjście są wektorami wartości liczbowych.

Wiele algorytmów wykorzystywanych do klasyfikacji używa się również do zadań regresji prostej i regresji wielokrotnej. To dlatego, że obsługują one zarówno dyskretne, jak i ciągłe prognozy liczbowe.

W chwili pisania tego tekstu nie był dostępny dedykowany interfejs API do regresji wielozmiennej, dlatego konieczne jest zaprojektowanie systemu w taki sposób, aby obsługiwał ten przypadek użycia. Proces jest podobny do tego, który wykorzystywaliśmy do klasyfikacji wieloetykietowej: przygotowanie danych, niezależne szkolenie każdego wariantu, testowanie i dostrajanie wielu modeli, a na koniec zebranie prognoz.

Aby lepiej zapoznać się z regresją, korzystamy w prognozowaniu ze zbioru danych emisji $CO_2$ pojazdów z serwisu Kaggle (*https://oreil.ly/GND1E*). W tym celu uwzględnimy takie cechy jak firma, model samochodu, pojemność silnika, rodzaj paliwa czy zużycie paliwa.

Jak wkrótce się przekonasz, praca z danymi w celu rozwiązania takiego problemu wymaga cechowania, oczyszczania i sformatowania danych w celu dopasowania do algorytmu.

Zbiór danych zawiera 13 kolumn. Aby przyspieszyć proces indeksowania i obliczania skrótów, skorzystamy z selektora `FeatureHasher` tylko w odniesieniu do cech ciągłych. Selektor ten wymaga podania charakteru cech liczbowych — tzn. wskazania, czy są dyskretne, czy ciągłe:

```
from pyspark.ml.feature import FeatureHasher

cols_only_continuous = ["Zużycie paliwa w mieście (l/100 km)",
                        "Zużycie paliwa na autostradzie (l/100 km)",
                        "Zużycie paliwa jazda mieszana (l/100 km)"]
hasher = FeatureHasher(outputCol="skróty_cech",
                       inputCols=cols_only_continuous)
co2_data = hasher.transform(co2_data)
```

Zwróćmy uwagę na przekazywanie listy w roli argumentu `inputCols`. Dzięki temu wielokrotne wykorzystanie kodu staje się łatwiejsze, a tworzony kod jest czystszy!

Kolumna `skróty_cech` jest typu `SparseVector`. Przyjrzyjmy się listingowi 6.4. Ze względu na złożoność funkcji haszującej otrzymaliśmy wektor o rozmiarze 262 144.

*Listing 6.4. Wektor rzadki kolumny hashed_features*

```
+-----------------------------------------+
|skróty_cech                              |
+-----------------------------------------+
|(262144,[38607,109231,228390],[0.0,9.9,6.7]) |
|(262144,[38607,109231,228390],[0.0,11.2,7.7])|
|(262144,[38607,109231,228390],[0.0,6.0,5.8]) |
|(262144,[38607,109231,228390],[0.0,12.7,9.1])|
|(262144,[38607,109231,228390],[0.0,12.1,8.7])|
+-----------------------------------------+
only showing top 5 rows
```

W powyższym kodzie jest wiele do poprawienia, ponieważ większość wektorów to wektory rzadkie, które mogą być bezużyteczne. Nadszedł więc czas, aby wybrać cechy automatycznie:

```
from pyspark.ml.feature import UnivariateFeatureSelector

selector = UnivariateFeatureSelector(outputCol="wybraneCechy",
                                     featuresCol="skróty_cech",
                                     labelCol="co2")

selector.setFeatureType("continuous")
selector.setLabelType("continuous")
model = selector.fit(co2_data_train)
output = model.transform(co2_data_test)
```

Selektor zmniejsza liczbę cech z 262 144 do 50.

Zauważ, że w gruncie rzeczy za pomocą wywołania `FeatureHasher` zwiększyliśmy wymiary. To dlatego, że nie znormalizowaliśmy wcześniej danych, aby ułatwić wycofanie się z eksperymentu. W rzeczywistych przypadkach użycia przed obliczaniem skrótu (tzw. haszowaniem) najlepiej znormalizować dane.

Kolejny krok polega na zbudowaniu modelu uczenia maszynowego. Biblioteka MLlib udostępnia do wyboru wiele algorytmów, na przykład `AFTSurvivalRegression`, `DecisionTreeRegressor` i `GBTRegressor` (pełną listę znajdziesz w dokumentacji — *https://oreil.ly/IYkDB*). AFT to skrót od *accelerated failure time* (dosłownie: przyspieszony czas awarii). Ten algorytm można wykorzystać do ustalenia czasu, przez jaki maszyna przetrwa w fabryce. Algorytm `DecisionTreeRegressor` działa najlepiej na cechach kategorialnych obejmujących skończoną liczbę kategorii. W rezultacie nie jest w stanie, jak inne regresory, prognozować wartości niezaobserwowanych. Algorytm `GBTRegressor` implementuje regresor drzew wzmocnionych gradientem, wykorzystujący zbiór drzew decyzyjnych szkolonych w sposób seryjny. Dane szkoleniowe są dzielone na zbiór danych szkoleniowych i zbiór danych walidacyjnych. Zbiór walidacyjny służy do zmniejszenia błędu w każdej iteracji algorytmu po danych szkoleniowych.

Czym ten algorytm różni się od przedstawionego wcześniej klasyfikatora `RandomForestClassifier`? Główna różnica polega na tym, że algorytm `GBT` buduje jedno drzewo naraz, co pomaga poprawić błędy popełnione w poprzednim drzewie. Z kolei algorytm lasu losowego buduje drzewa losowo i równolegle: każdy podzbiór węzłów roboczych tworzy własne drzewo, które jest następnie zbierane w węźle głównym, gdzie dane wyjściowe węzłów roboczych służą do utworzenia ostatecznego modelu. Zarówno `GBTRegressor`, jak i `RandomForestClassifier` obsługują funkcje ciągłe i kategorialne.

W następnym przykładzie wypróbujemy algorytm `GBTRegressor` z biblioteki MLlib, aby sprawdzić jego wyniki. Chociaż ze względu na sekwencyjny charakter algorytmu szkolenie może trwać nieco dłużej, funkcja optymalizacji powinna pomóc w uzyskaniu dokładniejszych wyników:

```
from pyspark.ml.regression import GBTRegressor
# zdefiniowanie klasyfikatora
gbtr = GBTRegressor(maxDepth=3, featuresCol="wybraneCechy", labelCol="CO2")
# zbudowanie modelu
model = gbtr.fit(input_data)
# skorzystanie z modelu
test01 = model.transform(test_data)
```

Teraz, gdy już mamy model, możemy pozyskać dane, które wykorzystamy do jego szkolenia. Trzeba także sprawdzić, czy nie doszło do nadmiernego dopasowania. Jeśli prognozy modelu `test01` są w stu procentach dokładne, istnieje duże prawdopodobieństwo nadmiernego dopasowania. Choć może się to zdarzyć również przy niższej dokładności, to za każdym razem, gdy dokładność zbliża się do stu procent, trzeba zachować czujność. Więcej o ocenie modeli opowiem w podrozdziale „Potoki uczenia maszynowego”. Na razie przyjrzyjmy się próbce z kolumny `prediction` pokazanej na listingu 6.5.

*Listing 6.5. Prognozowana a rzeczywista emisja $CO_2$ pojazdów*

```
+----------+----------+-------------+----+------------------+
|Typ paliwa|     Model|Klasa pojazdu| CO2|        prediction|
+----------+----------+-------------+----+------------------+
|       AS5|       ILX|      COMPACT|33.0| 32.87984310695771|
|        M6|       ILX|      COMPACT|29.0|28.261976730819185|
|       AV7|ILX HYBRID|      COMPACT|48.0| 49.88632059287859|
|       AS6|   MDX 4WD|  SUV - SMALL|25.0|24.864078951152344|
|       AS6|   RDX AWD|  SUV - SMALL|27.0| 26.95552579785164|
+----------+----------+-------------+----+------------------+
only showing top 5 rows
```

Jak widać, w kolumnie `prediction` zostały wygenerowane punkty danych, które są bardzo podobne do rzeczywistych danych zestawionych w kolumnie `CO2`. Na przykład w pierwszym wierszu prognoza wynosiła `32,879`, a rzeczywista wartość $CO_2$ to 33,0. Błąd w tym przypadku jest na akceptowalnym poziomie i dotyczy to także pozostałych wierszy. Można na tej podstawie wyciągnąć wniosek, że uczenie algorytmu zmierza we właściwym kierunku. Prognozowane wyniki nie są identyczne z wartościami rzeczywistymi (co oznacza, że istnieje małe prawdopodobieństwo nadmiernego dopasowania), ale są do nich zbliżone. Jak wspomniałam wcześniej, aby zmierzyć ogólną skuteczność modelu, nadal trzeba przeprowadzić statystyczne testy ewaluacyjne.

Biblioteka MLlib obsługuje także inne algorytmy uczenia maszynowego, które można wykorzystać do rozwiązania tego problemu. Jednym z nich jest `FMRegression` (FM oznacza *factorized machines*). Model opiera się na algorytmie spadku gradientowego (ang. *gradient descent algorithm*) z dedykowaną funkcją straty, zwaną także **funkcją optymalizacji**. Algorytm spadku gradientowego to iteracyjny algorytm optymalizacji. Iteruje po danych w poszukiwaniu reguł lub definicji, które powodują minimalną utratę dokładności. Teoretycznie wydajność algorytmu poprawia się z każdą iteracją do czasu, aż funkcja straty osiągnie wartość optymalną.

Maksymalna liczba iteracji algorytmu `FMRegression` jest domyślnie ustawiona na 100, ale można to zmienić za pomocą funkcji `setMaxIter`. Algorytm wykorzystuje funkcję optymalizacji `SquaredError`.

Funkcja `SquaredError` implementuje funkcję `MSE`, która w każdej iteracji oblicza ogólny średni błąd kwadratowy. Właśnie tę wielkość algorytm stara się zmniejszyć: sumę kwadratów „odległości" pomiędzy wartością rzeczywistą a wartością prognozowaną w danej iteracji. Funkcja `MSE` przy standardowych założeniach modeli liniowych jest uznawana za niestronniczy estymator wariancji błędu.

Oprócz algorytmu `FMRegression` istnieje również klasyfikator `FMClassifier`. Najważniejszą różnicę między nimi stanowi funkcja straty. Klasyfikator wykorzystuje funkcję `LogisticLoss`, czasami nazywaną **utratą entropii** (ang. *entropy loss*) lub **utratą loga** (ang. *log loss*). Funkcja `LogisticLoss` jest używana również w algorytmie `LogisticRegression`. W tej książce nie będziemy zagłębiać się w teorię, która się za tym kryje. Temu tematowi poświęcono wiele książek wprowadzających do zagadnień uczenia maszynowego. Jedną z nich jest wydana przez O'Reilly książka Hali Nelson *Essential Math for AI — https://oreil.ly/ess-math-ai*. Warto jednak znać podobieństwa i różnice między algorytmami klasyfikacji i regresji.

## Systemy rekomendacji

Systemy rekomendacji są często uczone z wykorzystaniem zbioru danych o filmach, na przykład MovieLens (*https://movielens.org*), którego celem jest polecanie filmów użytkownikom na podstawie tego, co lubili inni użytkownicy i (lub) preferencji użytkownika co do określonych gatunków. Systemy rekomendacji są wykorzystywane w wielu platformach internetowych — na przykład w systemach e-commerce, takich jak Amazon, lub platformach streamingowych, takich jak Netflix. Bazują na *uczeniu się reguł asocjacyjnych*, gdzie zadaniem algorytmu jest nauczenie się powiązań między filmami a użytkownikami.

Na wysokim poziomie, w zależności od dostępnych danych (metadane o użytkownikach i treści oraz dane o interakcjach pomiędzy użytkownikami a treścią), można je podzielić na trzy kategorie:

**Oparte na treści**

Algorytmy wykorzystują dostępne metadane dotyczące treści i użytkowników, w tym treści, które użytkownik oglądał wcześniej i które ocenił, ulubione gatunki, gatunki filmowe itp., i na podstawie tych informacji tworzą rekomendacje. Można je zaimplementować za pomocą mechanizmu reguł, niekoniecznie z wykorzystaniem technik uczenia maszynowego.

**Wspólne filtrowanie**

W tym przypadku nie są dostępne żadne metadane dotyczące filmów i użytkowników. Dostępna jest jedynie macierz interakcji definiująca interakcje między użytkownikami a treścią (tzn. filmy, które obejrzał lub ocenił każdy z użytkowników). Aby przedstawić rekomendację, algorytm wyszukuje podobieństwa między interakcjami użytkownika.

**Sieci neuronowe**

Do formułowania rekomendacji wykorzystywane są sieci neuronowe utworzone na podstawie metadanych dotyczących użytkowników i treści oraz macierzy interakcji.

## Algorytm ALS wspólnego filtrowania

Biblioteka MLlib dostarcza dobrze udokumentowane rozwiązanie filtrowania zespołowego o nazwie `ALS` (ang. *alternating least squares* — dosłownie: metoda naprzemiennych najmniejszych

kwadratów). Jej celem jest uzupełnienie brakujących wartości w macierzy interakcji użytkownik-element. Zapewnia również rozwiązanie dla scenariuszy zimnego startu, w których użytkownik jest nowy w systemie i nie ma wcześniejszych danych, które można by wykorzystać do sformułowania zaleceń. Więcej na ten temat możesz przeczytać w dokumentacji MLlib (*https://oreil.ly/66Vyt*).

# Nienadzorowane uczenie maszynowe

Algorytmy nienadzorowane stosuje się w przypadku, gdy dane nie mają etykiet, a mimo to, bez znajomości pożądanego wyniku, chcemy automatycznie znajdować interesujące wzorce, prognozować zachowania lub obliczać podobieństwa. Te algorytmy mogą być stosowane w ramach procedury ekstrakcji cech zamiennie z algorytmami nadzorowanymi. Typowe zadania nienadzorowanego uczenia maszynowego obejmują wyodrębnianie częstych wzorców i klasteryzację. Przyjrzyjmy się, jak można obsłużyć te zadania za pomocą biblioteki MLlib.

## Wydobywanie częstych wzorców

Wydobywanie częstych wzorców należy do kategorii **uczenia reguł asocjacyjnych** (ang. *association rule learning*), opartej na identyfikowaniu reguł w danych w celu odkrywania relacji między zmiennymi. Algorytmy wydobywania reguł asocjacyjnych zazwyczaj najpierw szukają częstych elementów w zbiorze danych, a następnie częstych par lub zestawów elementów (na przykład elementów, które często są oglądane lub kupowane razem). Reguły opierają się na podstawowej strukturze **poprzednika** (jeżeli) i **następnika** (to).

Biblioteka MLlib udostępnia dwie funkcje do wyodrębniania częstych wzorców, które można wykorzystać w roli procedur wstępnego przetwarzania w silnikach rekomendacji, na przykład wyodrębnianie znaczących wzorców z korpusu tekstu w celu sprawdzenia opinii użytkowników o filmie: `FPGrowth` i `PrefixSpan`. W tym rozdziale skoncentruję się na algorytmach klasteryzacji, ponieważ można ich używać samodzielnie. Jednak w wielu przypadkach, aby uzyskać końcowy wynik, trzeba skorzystać z więcej niż jednego algorytmu. Więcej informacji na temat algorytmów wyodrębniania częstych wzorców można znaleźć w dokumentacji MLlib (*https://oreil.ly/VQQCQ*).

## Klasteryzacja

**Klasteryzacja** to technika grupowania służąca do odkrywania ukrytych relacji między punktami danych. Klasteryzację powszechnie wykorzystuje się do segmentacji klientów, przetwarzania i wykrywania obrazów, filtrowania spamu, wykrywania anomalii itp.

W procesie klasteryzacji każdy element jest przypisywany do grupy zdefiniowanej przez jej środek. Prawdopodobieństwo przynależności elementu do grupy oblicza się na podstawie jego odległości od środka. Algorytmy zazwyczaj próbują zoptymalizować model poprzez zmianę środkowych punktów grup.

Nazwy algorytmów grupowania często zawierają literę *k*, na przykład *k*-najbliższych sąsiadów (*k*-NN) i *k*-średnich (ang. *k-means*). Znaczenie litery *k* jest różne dla różnych algorytmów. Często reprezentuje ona liczbę predefiniowanych klastrów (tematów). Algorytmy MLlib mają domyślną wartość

całkowitą *k*. Można ją ustawić za pomocą metody `setK` lub przekazać jako parametr. Niektóre algorytmy wymagają, aby dane miały ustawiony parametr `weightCol` — w szczególności algorytmy `KMeans`, `GaussianMixture`, `PowerIterationClustering` i `BisectingMeans` z biblioteki MLlib oczekują w zbiorze danych szkoleniowych nieujemnej wartości `weightCol`, reprezentującej wagi punktów danych względem środka klastra. Jeśli określony punkt danych ma dużą wagę i jest stosunkowo daleko od centrum klastra, „koszt" jego przetwarzania przez funkcję optymalizacji (innymi słowy, strata dla tego punktu) jest wysoki. Aby zmniejszyć ogólną stratę, algorytm dąży do zminimalizowania strat we wszystkich punktach danych poprzez przesunięcie środka klastra, jeśli to możliwe, bliżej takich punktów danych.

Prawie wszystkie algorytmy klasteryzacji wymagają wartości początkowych (wyjątkiem jest `Power↪IterationClustering`). Wartość ziarna (ang. *seed*) służy do losowej inicjalizacji zbioru środkowych punktów klastra (pomyśl o współrzędnych *x* i *y*), a przy każdej iteracji algorytmu środki są aktualizowane na podstawie funkcji optymalizacji.

Po zapoznaniu się z zagadnieniami klasteryzacji możemy powrócić do pierwotnego celu polegającego na przewidywaniu emisji $CO_2$ i sprawdzić, czy potrafimy zidentyfikować podobieństwa między takimi kolumnami jak rodzaj paliwa, zużycie paliwa czy liczba cylindrów. Biblioteka MLlib udostępnia pięć algorytmów klasteryzacji. Przyjrzyjmy się im, aby zobaczyć, które z nich mogą być odpowiednie do tego zadania:

`LDA`

: `LDA` (ang. *Latent Dirichlet Allocation*) to algorytm statystyczny ogólnego przeznaczenia stosowany w biologii ewolucyjnej, biomedycynie i w zadaniach przetwarzania języka naturalnego. Oczekuje wektora reprezentującego liczbę pojedynczych słów w dokumencie. Ponieważ w przykładowym scenariuszu skupiamy się na takich zmiennych jak rodzaj paliwa, algorytm LDA nie pasuje do tych danych.

`GaussianMixture`

: Algorytm `GaussianMixture` jest często używany do identyfikacji obecności grupy w większej grupie. W przykładowym kontekście algorytm ten może się przydać do identyfikacji podgrup różnych klas w obrębie grupy poszczególnych producentów samochodów, na przykład klasy samochodów kompaktowych w grupie Audi i grupie Bentley. Wiadomo jednak, że algorytm `GaussianMixture` słabo radzi sobie z danymi wielowymiarowymi, co utrudnia algorytmowi osiągnięcie zbieżności do uzyskania satysfakcjonujących wniosków. Mówi się, że dane są wielowymiarowe, gdy liczba obiektów (kolumn) jest bliska liczbie obserwacji (wierszy) lub większa od niej. Na przykład dane są uważane za wielowymiarowe, jeśli w zbiorze danych jest pięć kolumn i cztery wiersze. W świecie dużych zbiorów danych istnienie takich zbiorów danych jest mniej prawdopodobne.

`KMeans`

: Ze względu na prostotę i wydajność `KMeans` jest najpopularniejszym algorytmem klasteryzacji. Pobiera w roli danych wejściowych grupę klas, tworzy losowe środki danych i rozpoczyna iterację po punktach danych i centrach, mając na celu zgrupowanie podobnych punktów danych i znalezienie ich optymalnych punktów środkowych. Algorytm zawsze jest zbieżny, ale jakość wyników zależy od liczby klastrów (*k*) i liczby iteracji.

`BisectingKMeans`

Algorytm `BisectingKMeans` opiera się na algorytmie `KMeans` z hierarchią grup. Obsługuje obliczanie odległości na dwa sposoby: euklidesowy (ang. *euclidean*) lub kosinusowy (ang. *cosine*). Model można przedstawić jako drzewo z grupami liści. Po rozpoczęciu szkolenia istnieje tylko węzeł główny. W celu optymalizacji modelu w każdej iteracji węzły są dzielone na dwie części. Jest to świetna opcja, jeśli chcesz stworzyć reprezentację grup i podgrup.

`PowerIterationClustering`

Algorytm `PowerIterationClustering` (PIC) implementuje algorytm Lina i Cohena (*https://oreil.ly/-Gu9c*). To skalowalna i wydajna opcja grupowania wierzchołków grafu na podstawie podobieństw par jako właściwości krawędzi. Należy zwrócić uwagę, że tego algorytmu nie można używać w potokach Sparka, ponieważ nie implementuje on wzorca `Estimator` lub `Transformer` (więcej informacji na ten temat można znaleźć w podrozdziale „Potoki uczenia maszynowego").

Doskonale! Teraz, gdy wiesz, jakie masz opcje do dyspozycji, możemy wypróbować jedną z nich. Ponieważ przykładowy zbiór danych ma tylko 11 kolumn i znacznie więcej wierszy danych, wybierzemy algorytm `GaussianMixture`. Wykorzystamy zbiór danych *$CO_2$ Emission by Vehicles* po jego wstępnym przetworzeniu i zastosowaniu inżynierii cech we wszystkich kolumnach, włącznie z kolumną `label` (kompleksowy samouczek znajdziesz w pliku *ch06_gm_pipeline.ipynb* w repozytorium książki, dostępnym w serwisie GitHub — *https://oreil.ly/Dl9nO*).

Wartość *k* w tym przypadku oznacza liczbę producentów samochodów w zbiorze danych. Do wyodrębnienia tej wartości używamy wywołania `distinct().count()`:

```
dataset.select("Marka").distinct().count()
```

Wynik to 42. To dość interesująca liczba. Przekażemy ją do konstruktora razem z ustawieniami kilku innych parametrów:

```
from pyspark.ml.clustering import GaussianMixture
gm = GaussianMixture(k=42, tol=0.01, seed=10,
                     featuresCol="wybraneCechy", maxIter=100)
model = gm.fit(dataset)
```

Gdy już mamy model, możemy uzyskać obiekt podsumowania (`summary`), który go reprezentuje:

```
summary = model.summary
```

Obiekt podsumowania mają wszystkie algorytmy klasteryzacji i klasyfikacji. W przypadku klasteryzacji obiekt zawiera prognozowane punkty środkowe klastrów, przekształcone prognozy, rozmiar klastra (tzn. liczbę obiektów w każdym klastrze) oraz dedykowane parametry specyficzne dla algorytmu.

Na przykład dzięki uruchomieniu metody `distinct().count()` obiektu `summary` możemy się dowiedzieć, do ilu grup algorytm osiągnął zbieżność:

```
summary.cluster.select("prediction").distinct().count()
```

W tym przypadku uzyskaliśmy wartość 17. Możemy teraz spróbować zmniejszyć wartość *k*, aby sprawdzić, czy uzyskamy lepszą zbieżność, lub spróbować zwiększyć liczbę iteracji, aby sprawdzić, czy ma to wpływ na tę liczbę. Oczywiście im więcej iteracji wykonuje algorytm, tym przetwarzanie zajmuje więcej czasu. W związku z tym, jeśli korzystamy z dużego zbioru danych, powinniśmy

ostrożnie dobierać parametry. Liczbę potrzebnych iteracji należy określić metodą prób i błędów. Należy pamiętać o dodaniu tych informacji do testów eksperymentalnych oraz o wypróbowaniu różnych miar wydajności.

Innym sposobem pomiaru wydajności modelu jest sprawdzenie wartości `logLikelihood`, reprezentującej statystyczną istotność różnicy między znalezionymi przez model grupami:

```
summary.logLikelihood
```

Przy 200 iteracjach otrzymaliśmy około 508 076 punktów wiarygodności. Wartości te nie są znormalizowane i trudno je bezpośrednio porównać. Jednak wyższy wynik wskazuje na większe prawdopodobieństwo powiązania egzemplarza z klastrem. Dlatego skorzystanie z wartości `logLikelihood` jest dobrym sposobem porównania wydajności jednego modelu z innym na tych samych danych, ale niekoniecznie samodzielnej oceny wydajności modelu. Jest to jeden z powodów, dla których ważne jest zdefiniowanie z góry celów eksperymentów. Aby dowiedzieć się więcej na ten temat, polecam lekturę wydanej przez O'Reilly książki Petera Bruce'a, Andrew Bruce'a i Petera Gedecka *Practical Statistics for Data Scientists* (*https://oreil.ly/prac-stats*).

Załóżmy, że kontynuujemy poszukiwania z wykorzystaniem parametru `maxIter = 200` i otrzymaliśmy tę samą liczbę różnych prognoz: 17. Na tej podstawie zdecydowaliśmy się zmienić *k* na 17.

Następnym krokiem, aby upewnić się, że klastry nie zawierają zerowych punktów danych, może być sprawdzenie rozmiarów klastrów:

```
summary.clusterSizes
```

Powyższe wywołanie zwraca wynik pokazany poniżej, przy czym poszczególne indeksy reprezentują indeks grupy:

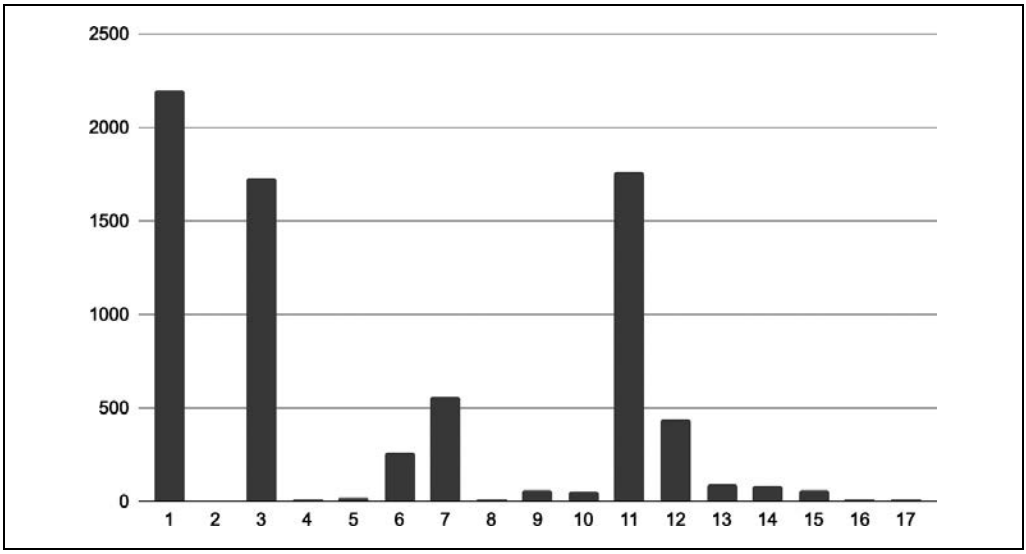
```
[2200, 7, 1733, 11, 17, 259, 562, 12, 63, 56, 1765, 441, 89, 88, 61, 13, 8]
```

Ponieważ wartość `clusterSizes` jest typu `Array`, do utworzenia histogramu wartości możemy użyć takich narzędzi jak `numpy` i `matplotlib`. Wynikowy rozkład wielkości grup (klastrów) przedstawiono na rysunku 6.2.

# Ocena

Faza oceny (ang. *evaluation*) jest istotną częścią procesu uczenia maszynowego. W ten sposób szacujemy wydajność modelu. Biblioteka MLlib ma sześć ewaluatorów, z których każdy implementuje klasę abstrakcyjną Sparka `Evaluator`. Można je z grubsza podzielić na dwie grupy: nadzorowane i nienadzorowane.

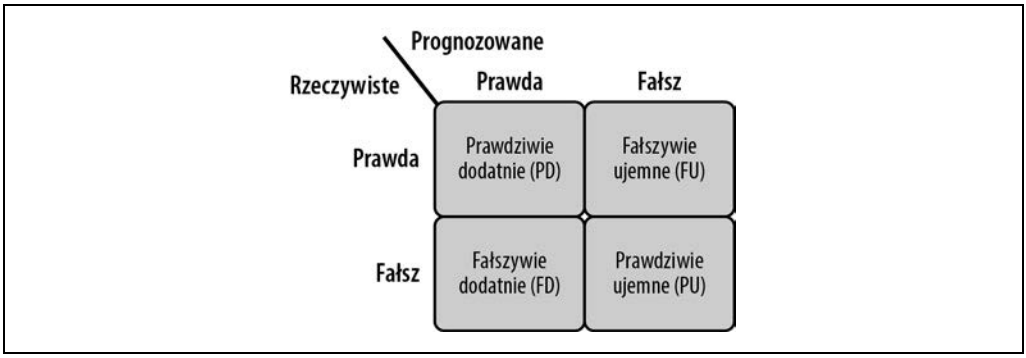
`Evaluator` to klasa, która pozwala stwierdzić, jak radzi sobie określony model według wskazanych kryteriów oceny uczenia maszynowego.




*Rysunek 6.2. Histogram rozmiarów klastrów*

## Ewaluatory nadzorowane

W uczeniu nadzorowanym dostępne są etykiety danych testowych, dzięki czemu można stworzyć wiele wskaźników szacowania wydajności. Aby to było możliwe, estymator najpierw oblicza **macierz pomyłek** (ang. *confusion matrix*), która zawiera porównanie etykiet prognozowanych z rzeczywistymi. W przypadku zadania klasyfikacji binarnej wynik mógłby wyglądać tak jak pokazano na rysunku 6.3, gdzie każde pole jest wartością z przedziału `[0, rozmiar_zbioru_danych]`.




*Rysunek 6.3. Binarna macierz pomyłek*

Wartości *Prawda* i *Fałsz* oznaczają dokładność prognozy, a wyniki dodatnie i ujemne oznaczają binarne prognozy (mogą one również być reprezentowane przez wartości 1 i 0). Istnieją cztery kategorie wyników:

**Prawdziwie dodatnie (PD)**

Etykieta *Prawda*, w przypadku, gdy prognoza również przewiduje prawdę.

**Prawdziwie ujemne (PU)**

Etykieta *Fałsz*, w przypadku, gdy prognoza również przewiduje fałsz.

**Fałszywie dodatnie (FD)**

Etykieta *Fałsz*, w przypadku, gdy prognoza przewiduje prawdę.

**Fałszywie ujemne (FU)**

Etykieta *Prawda*, w przypadku, gdy prognoza przewiduje fałsz.

Estymatory Sparka pozwalają zapewnić wiele wskaźników na podstawie tych wartości. Szczegółowe informacje można znaleźć w dokumentacji (*https://oreil.ly/knYwj*).

W przypadku próby zastosowania tego samego procesu do klasyfikacji *wieloklasowej* i *wieloetykietowej* macierz pomyłek odpowiednio się powiększy, aby uwzględnić wszystkie możliwości etykiet. W przypadku danych niezrównoważonych zwykle trzeba napisać własny ewaluator. Można to zrobić za pomocą rozbudowanego API Sparka.

Oprócz bazowej klasy `Evaluator` Spark udostępnia następujące interfejsy API do obliczania wskaźników wydajności:

`BinaryClassificationEvaluator`

Binarny ewaluator oczekujący kolumn wejściowych `rawPrediction`, `label` i `weight` (opcjonalnie). Można go wykorzystać do wyznaczenia obszaru pod krzywymi ROC (ang. *receiver operating characteristic*) oraz PR (ang. *precision-recall*).

`MulticlassClassificationEvaluator`

Ewaluator klasyfikacji wieloklasowej, który oczekuje kolumn wejściowych `prediction`, `label`, `weight` (opcjonalnie) oraz `probabilityCol` (tylko dla `logLoss`). Obejmuje dedykowane metryki, takie jak `precisionByLabel`, `recallByLabel`, oraz specjalną macierz `hammingLoss`, która oblicza część etykiet, które zostały przewidziane błędnie. Przy porównywaniu modeli klasyfikacji binarnej i wieloklasowej należy pamiętać, że oceny precyzji (ang. *precision*), przywołania (ang. *recall*) i F1 są przeznaczone dla modeli klasyfikacji binarnej; dlatego lepiej jest porównać wskaźniki `hammingLoss` (strata Hamminga) oraz `accuracy` (dokładność).

`MultilabelClassificationEvaluator`

Ewaluator klasyfikacji wieloetykietowej (wprowadzony w Sparku 3.0). W chwili pisania tego tekstu ten ewaluator nadal był uznawany za eksperymentalny. Oczekuje dwóch kolumn wejściowych: `prediction` i `label`. Na ich podstawie zwraca dedykowane metryki, takie jak `microPrecision`, `microRecall`, oznaczające średnie wartości ze wszystkich klas prognoz.

Funkcjonalność w fazie eksperymentalnej zwykle nadal jest w fazie rozwoju i nie jest gotowa do pełnego wdrożenia. W większości przypadków takie funkcjonalności obejmują zaledwie kilka modułów i są publikowane głównie po to, aby umożliwić programistom zdobycie wiedzy i uwag na potrzeby przyszłego rozwoju oprogramowania. W technologiach open source funkcjonalności eksperymentalne często mogą być wykorzystywane w kodzie produkcyjnym. Trzeba jednak pamiętać, że takie funkcjonalności mogą się zmienić w przyszłych wersjach oprogramowania, a autorzy nie zobowiązują się do dalszego ich wspierania ani przekształcania w wersje dopracowane.

`RegressionEvaluator`

Ten ewaluator oczekuje kolumn wejściowych `prediction`, `label` i `weight` (opcjonalnie) i na ich podstawie generuje takie metryki jak `mse` (średni błąd kwadratowy odległości między etykietami prognozowanymi a rzeczywistymi) i `rmse` (pierwiastek z poprzedniej wartości).

`RankingEvaluator`

Ten ewaluator (dodany w Sparku 3.0, w chwili pisania tego tekstu znajduje się w fazie eksperymentalnej) oczekuje kolumn wejściowych `prediction` i `label`. Często jest używany do oceny rankingu wyników wyszukiwania. Korzysta ze zmiennej *k*, którą można ustawić w celu uzyskania macierzy średnich z *k* pierwszych wyników. Pomyśl o systemie rekomendacji filmów, który może wyświetlać 5 lub 10 rekomendacji: średnie będą się zmieniać w zależności od liczby zwróconych rekomendacji, a ocena wyników pomoże Ci podjąć świadomą decyzję o wyborze filmu. Wyniki tego ewaluatora opierają się na interfejsie API `RankingMetrics` korzystającym ze struktury `RDD` biblioteki MLlib. Więcej informacji na ten temat można znaleźć w dokumentacji (*https://oreil.ly/vobZH*).

## Ewaluatory nienadzorowane

Biblioteka MLlib dostarcza również ewaluator dla technik uczenia nienadzorowanego. Dzięki wykorzystaniu możliwości bibliotek `TensorFlow`, `PyTorch` lub innych bibliotek umożliwiających przetwarzanie ramek `DataFrame` Sparka można uzyskać dostęp do jeszcze większej liczby opcji. Do oceny wyników klasteryzacji można wykorzystać dostępny w bibliotece MLlib ewaluator `Clustering` ↪`Evaluator`. Oczekuje on dwóch kolumn wejściowych: `prediction` i `features` oraz opcjonalnej kolumny `weight`. Ewaluator oblicza **miarę sylwetki** (ang. *silhouette measure*), dla której można wybrać jedną z dwóch miar odległości: kwadratową euklidesową (`squaredEuclidean`) i kosinusową (`cosinus`). Miara sylwetki to ocena spójności i poprawności klastrów. Aby ją wykonać, algorytm oblicza odległości pomiędzy każdym punktem danych a innymi punktami danych w klastrze i porównuje z odległością od punktów w innych klastrach. Wyniki oznaczają średnie wartości sylwetki wszystkich punktów na podstawie wagi punktów.

Powróćmy do przykładu klasyfikacji. Za pomocą algorytmów `GaussianMixture` i `KMeans` możemy ocenić modele tak, aby podejmować lepsze decyzje. Kod z przykładów zamieszczonych w tym rozdziale można znaleźć w repozytorium książki w serwisie GitHub (*https://oreil.ly/smls-git6*):

```
from pyspark.ml.evaluation import ClusteringEvaluator
evaluator = ClusteringEvaluator(featuresCol='wybraneCechy')
evaluator.setPredictionCol("prediction")

print("kmeans: "+str(evaluator.evaluate(kmeans_predictions)))
print("GM: "+ str(evaluator.evaluate(gm_predictions)))
```

Domyślną metodą obliczania odległości jest kwadratowa odległość euklidesowa. Otrzymaliśmy następujące wyniki:

```
kmeans: 0.7264903574632652
GM: -0.1517797715036008
```

Jak można stwierdzić, która z uzyskanych miar jest lepsza? Ewaluator ma dedykowaną funkcję o nazwie `isLargerBetter`, która pozwala nam to ocenić:

```
evaluator.isLargerBetter()
```

W naszym przypadku ta funkcja zwraca `True`, co sugeruje, że dla danych z tego przykładu lepiej sprawdza się algorytm `KMeans`. Na tym jednak nie koniec — przyjrzyjmy się teraz odległości kosinusowej:

```
evaluator.setDistanceMeasure("cosine")
print("kmeans: "+str(evaluator.evaluate(kmeans_predictions)))
print("GM: "+ str(evaluator.evaluate(gm_predictions)))
```

Oto uzyskane wyniki:

```
kmeans: 0.05987140304400901
GM: -0.19012403274289733
```

Algorytm `KMeans` nadal sprawdza się lepiej, ale w tym przypadku różnica jest znacznie mniej wyraźna. Dzieje się tak prawdopodobnie ze względu na sposób implementacji modeli; na przykład algorytm `KMeans` korzysta z odległości euklidesowej jako domyślnej miary odległości do klasteryzacji i dlatego przy ocenie modelu na podstawie kwadratowej odległości euklidesowej ogólnie sprawdza się lepiej. Innymi słowy, ważne jest, aby ostrożnie podchodzić do interpretacji takich wskaźników. W celu zapewnienia większej przejrzystości możemy połączyć proces oceny z dostosowaniem zbiorów danych testowych i szkoleniowych, algorytmów i ewaluatorów. Czas na dostrajanie!

# Hiperparametry i eksperymenty dostrajania

Istnieją narzędzia, które pozwalają przeprowadzać wiele eksperymentów, tworzyć wiele modeli i automatycznie wskazywać ten, który jest najlepszy. Właśnie nimi zajmiemy się w tym podrozdziale!

Wszystkie procesy uczenia maszynowego, aby można je było wykorzystać do dokładnego prognozowania zdarzeń w rzeczywistym świecie, wymagają dostrajania i eksperymentowania. Można to osiągnąć poprzez podzielenie zbioru danych na wiele zbiorów szkoleniowych i testowych i (lub) modyfikację parametrów algorytmów.

## Budowanie siatki parametrów

Na przykład w poniższym kodzie, za pomocą metody `ParamGridBuilder().addGrid`, budujemy siatkę parametrów (`param`). Skorzystanie z tej metody pozwoliło zdefiniować wiele maksymalnych wartości iteracji na potrzeby budowania modeli *k*-średnich przy użyciu obu dostępnych metryk odległości:

```
from pyspark.ml.tuning import ParamGridBuilder

grid = ParamGridBuilder().addGrid(kmeans.maxIter, [20,50,100])
        .addGrid(kmeans.distanceMeasure, ['euclidean','cosine']).build()
```

Siatka parametrów to siatka lub tabela parametrów z dyskretną liczbą wartości w każdej z nich, których w ramach procesu uczenia można używać do iterowania po różnych kombinacjach wartości parametrów w poszukiwaniu optymalnych wartości. `ParamGridBuilder` to narzędzie, które pozwala szybciej zbudować taką siatkę. Można jej używać z dowolną funkcją MLlib, która pobiera tablicę parametrów. Zanim przejdziemy dalej, dostosujemy parametry ewaluatora poprzez dodanie do niego dedykowanej siatki:

```
grid = ParamGridBuilder().addGrid(kmeans.maxIter, [20,50,100])
        .addGrid(kmeans.distanceMeasure, ['euclidean','cosine'])
        .addGrid(evaluator.distanceMeasure, ['euclidean','cosine']).build()
```

## Podział danych na zbiory szkoleniowe i testowe

Następnie, aby losowo podzielić dane na zbiór szkoleniowy i zbiór testowy, które zostaną wykorzystane do oceny poszczególnych kombinacji parametrów, użyjemy obiektu `TrainValidationSplit`:

```
from pyspark.ml.tuning import TrainValidationSplit
tvs = TrainValidationSplit(estimator=kmeans, estimatorParamMaps=grid,
                           evaluator=evaluator, collectSubModels=True, seed=42)
tvs_model = tvs.fit(data)
```

Obiekt `TrainValidationSplit` domyślnie wykorzystuje 75% danych do uczenia i 25% do testowania. Aby to zmienić, należy podczas inicjalizacji ustawić parametr `trainRatio`. Obiekt `TrainValidationSplit` jest estymatorem, zatem implementuje metodę `fit` i zwraca transformer. Zmienna `tvs_model` reprezentuje najlepszy model zidentyfikowany po zweryfikowaniu różnych kombinacji parametrów. Można także zlecić obiektowi `TrainValidationSplit` zebranie wszystkich podmodeli, które działały gorzej, zamiast zatrzymania tylko najlepszego. W tym celu należy ustawić parametr `collectSub` ↪`Models` na `True`, jak pokazano w powyższym przykładzie.

Podczas korzystania z parametru `collectSubModels` należy zachować ostrożność. Rozważ korzystanie z tej opcji, jeśli w ramach Twojego obciążenia chcesz porównać wiele modeli uczenia maszynowego lub gdy uzyskasz podobne wyniki dla metryk walidacji i chcesz zachować wszystkie modele, aby móc kontynuować eksperymenty w celu zidentyfikowania najlepszego. Szczegółowe informacje na temat dostępu do podmodeli i powodów, dla których należy zachować przy tym ostrożność, można znaleźć w ramce „Jak uzyskać dostęp do różnych modeli?”.

Po czym można poznać, że został wybrany najlepszy model? Sprawdźmy metryki walidacji:

```
tvs_model.validationMetrics
```

Za pomocą tego wywołania można uzyskać pogląd na wydajność poszczególnych eksperymentów zgodnie z oceną wykorzystanego ewaluatora. Wyniki pokazano na listingu 6.6.

*Listing 6.6. Wyniki metryk walidacji*

```
[0.04353869289393124,
 0.04353869289393124,
 0.6226612814858505,
 0.6226612814858505,
 0.04353869289393124,
 0.04353869289393124,
 0.6226612814858505,
 0.6226612814858505,
 0.04353869289393124,
 0.04353869289393124,
 0.6226612814858505,
 0.6226612814858505]
```

Należy pamiętać, że wszystkie parametry wykorzystywane w procesie można dodać do obiektu `ParamGridBuilder`. Za pomocą tej funkcji można także ustawić określone kolumny dla etykiet i cech. Jeśli masz do dyspozycji słownik parametrów lub listę par (*`parametr`*, *`wartość`*), zamiast dodawać je jeden po drugim za pomocą metody `addGrid`, możesz skorzystać z funkcji `baseOn`, która za kulisami uruchamia za Ciebie pętlę `foreach`.

### Jak uzyskać dostęp do różnych modeli?

Jak pamiętasz, ustawiliśmy opcję `collectSubModels=True`. W ten sposób wszystkie modele zostały zapisane. Dostęp do nich można uzyskać z egzemplarza `subModels` za pomocą następującego fragmentu kodu:

```
arr_models = tvs_model.subModels
```

Należy jednak pamiętać, że ta operacja zbiera w sterowniku wszystkie informacje od egzekutorów. Jeśli pracujesz z dużym zestawem danych, mogą powstać wyjątki związane z brakiem pamięci. Zatem chociaż jest to świetne rozwiązanie do celów edukacyjnych, zdecydowanie powinieneś unikać korzystania z tej funkcji w rzeczywistych przypadkach!

`arr_models` to egzemplarz tablicy Pythona, w której są przechowywane modele i niektóre metadane (rysunek 6.4). Indeks tablicy odpowiada indeksowi pokazanej wcześniej tablicy metryk walidacji.

```
[KMeansModel: uid=KMeans_0695dc19e9ae, k=2, distanceMeasure=euclidean, numFeatures=50,
 KMeansModel: uid=KMeans_0695dc19e9ae, k=2, distanceMeasure=euclidean, numFeatures=50,
 KMeansModel: uid=KMeans_0695dc19e9ae, k=2, distanceMeasure=cosine, numFeatures=50,
 KMeansModel: uid=KMeans_0695dc19e9ae, k=2, distanceMeasure=cosine, numFeatures=50,
 KMeansModel: uid=KMeans_0695dc19e9ae, k=2, distanceMeasure=euclidean, numFeatures=50,
 KMeansModel: uid=KMeans_0695dc19e9ae, k=2, distanceMeasure=euclidean, numFeatures=50,
 KMeansModel: uid=KMeans_0695dc19e9ae, k=2, distanceMeasure=cosine, numFeatures=50,
 KMeansModel: uid=KMeans_0695dc19e9ae, k=2, distanceMeasure=cosine, numFeatures=50,
 KMeansModel: uid=KMeans_0695dc19e9ae, k=2, distanceMeasure=euclidean, numFeatures=50,
 KMeansModel: uid=KMeans_0695dc19e9ae, k=2, distanceMeasure=euclidean, numFeatures=50,
 KMeansModel: uid=KMeans_0695dc19e9ae, k=2, distanceMeasure=cosine, numFeatures=50,
 KMeansModel: uid=KMeans_0695dc19e9ae, k=2, distanceMeasure=cosine, numFeatures=50]
```

*Rysunek 6.4. Przykład tablicy subModels*

Egzemplarz `tvs_model` ma dostęp do wszystkich podmodeli. Korzystanie z tego egzemplarza do testowania lub prognozowania pozwala wybrać spośród wszystkich utworzonych podmodeli model o najlepszej wydajności. Gdy, tak jak w prezentowanym przypadku, wiele modeli pozwala uzyskać równie wysoką wydajność, wybierany jest pierwszy z listy.

## Walidacja krzyżowa: lepszy sposób testowania modeli

Jedną z operacji, której nie można wykonać za pomocą obiektu `TrainValidationSplit`, jest wypróbowanie wielu kombinacji podziału danych. W tym celu MLlib udostępnia obiekt `CrossValidator` — estymator, który implementuje *k*-krotną walidację krzyżową. Jest to technika, która dzieli zbiór danych na zestaw nienakładających się na siebie, losowo podzielonych „fałd", używanych oddzielnie w roli zbiorów szkoleniowego i testowego.

Taki podział jest kosztowną obliczeniowo operacją, ponieważ korzystanie z niej polega na szkoleniu *k* razy całej mapy parametrów modeli. Innymi słowy, jeśli `numFolds` wynosi 3 i mamy jedną siatkę parametrów z 2 wartościami, to przeszkolimy 6 modeli uczenia maszynowego. Przy `numFolds=3` i poprzedniej siatce parametrów dla ewaluatora i algorytmu będziemy szkolić model `numFolds` razy rozmiar siatki, czyli 12. Łącznie więc przeszkolimy 36 modeli uczenia maszynowego.

Oto jak można to zdefiniować:

```
from pyspark.ml.tuning import CrossValidator
cv = CrossValidator(estimator=kmeans, estimatorParamMaps=grid,
                    evaluator=evaluator, collectSubModels=True,
                    parallelism=2, numFolds=3)
cv_model = cv.fit(data)
```

Podobnie jak obiekt `TrainValidationSplit` ma parametr `validationMetrics`, obiekt `CrossValidator` ↪`Model` ma parametr `avgMetrics`, którego można użyć do pobierania metryk szkoleniowych. Parametr ten przechowuje średnią ocenę dla wszystkich kombinacji siatki parametrów. Parametr `numFold.parallelism` służy do równoległej oceny modeli. Ustawienie tego parametru na 1 powoduje wykonanie oceny sekwencyjnej. Parametr `parallelism` ma to samo znaczenie w obiekcie `TrainValidationSplit`. Uruchomienie metody `cv_model.avgMetrics` zwraca wynik pokazany na listingu 6.7.

*Listing 6.7. Wyniki średniej oceny metryk szkolenia modelu*

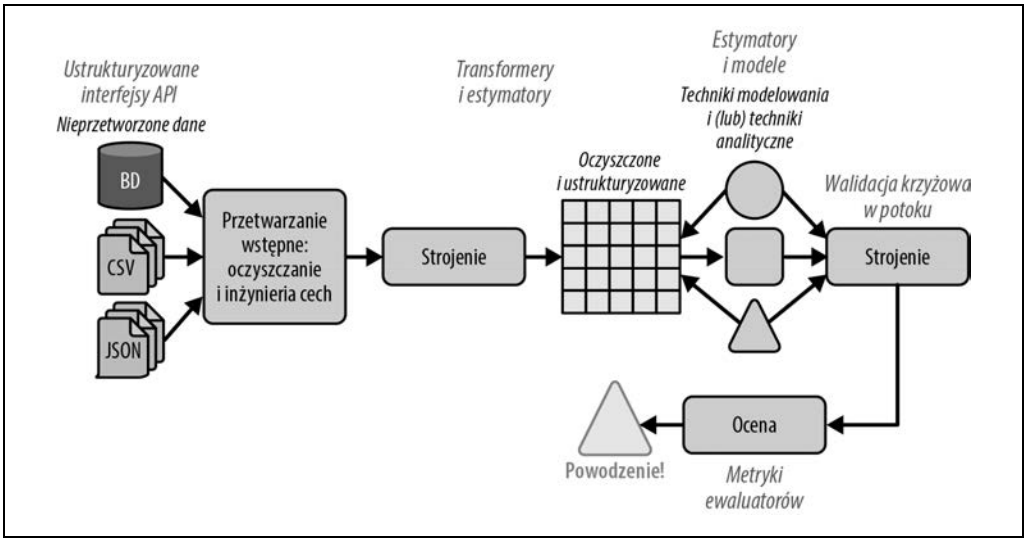
```
[0.057746040674997036,
 0.057746040674997036,
 0.5811536043895275,
 0.5811536043895275,
 0.057746040674997036,
 0.057746040674997036,
 0.5811536043895275,
 0.5811536043895275,
 0.057746040674997036,
 0.057746040674997036,
 0.5811536043895275,
 0.5811536043895275]
```

Każda komórka w tablicy `avgMetrics` jest obliczana za pomocą następującego równania:

$$averageEval = \frac{sum(EvalForFold)}{NumOfFold}$$

# Potoki uczenia maszynowego

W tym podrozdziale przedstawiono pojęcie potoku uczenia maszynowego, który jest zbudowany z różnych, omówionych wcześniej elementów składowych: cechowania, uczenia modelu, oceny modelu i dostrajania modelu. Kompletny proces: pozyskiwanie danych, wstępne przetwarzanie i czyszczenie, przekształcanie danych, budowanie i dostrajanie modelu oraz jego ocena zwizualizowano na rysunku 6.5.




*Rysunek 6.5. Przepływ pracy w uczeniu maszynowym na platformie Spark*

API `Pipelines` obejmuje dwa główne komponenty zbudowane na bazie ramek danych (`DataFrame`) i zestawów danych (`DataSet`):

`Transformer`

Funkcja, która w jakiś sposób przekształca dane. Jak dowiedziałeś się w rozdziale 4., transformery w Sparku to algorytmy lub funkcje, które pobierają ramki `DataFrame` w roli danych wejściowych i zwracają nowe ramki `DataFrame` zawierające pożądane kolumny. Jak przekonasz się w poniższym przykładzie, transformery mogą również przyjmować parametry Sparka. W ten sposób możesz wpływać na samą funkcję, włącznie z określaniem nazw kolumn wejściowych i wyjściowych. Dostarczanie parametrów tymczasowych daje elastyczność oceny i testowania wielu odmian modelu z wykorzystaniem tego samego obiektu.

Wszystkie transformery implementują metodę `Transformer.transform`.

`Estimator`

Obiekt dostarczający abstrakcje dowolnego algorytmu dopasowującego dane lub przeprowadzającego na nich szkolenie. Wynikiem działania estymatora jest model lub transformer. Na przykład algorytm uczenia, taki jak `GaussianMixture`, jest estymatorem, a wywołanie jego metody `fit` szkoli model `GaussianMixtureModel`, który jest transformerem. Estymator, podobnie jak transformer, może przyjmować dane wejściowe w postaci parametrów.

Wszystkie estymatory implementują funkcję `Estimator.fit`.

Powróćmy do przykładu prognozowania emisji $CO_2$. W tym przykładzie obiekt `UnivariateFeatureSelector` jest estymatorem. Zgodnie z definicją parametrów `outputCol` i `featuresCol` pobierają kolumnę o nazwie `hashed_features` i tworzą nową ramkę `DataFrame` z dołączoną nową kolumną o nazwie `selectedFeatures`:

```
selector = UnivariateFeatureSelector(outputCol="wybraneCechy",
                                     featuresCol="skróty_cech",
                                     labelCol="CO2")
```

```
model_select = selector.fit(data)
transformed_data = model_select.transform(data)
```

Wywołanie metody `fit` estymatora tworzy egzemplarz obiektu `UnivariateFeatureSelectorModel`, który przypisujemy do zmiennej `model_select`. Obiekt `model_select` jest teraz transformerem, którego możemy użyć do stworzenia nowej ramki danych z dołączoną kolumną `selectedFeatures`.

Zarówno estymatory, jak i transformery same w sobie są bezstanowe. Oznacza to, że gdy zostaną wykorzystane do stworzenia modelu lub innego egzemplarza transformera, nie zmienią ani nie zachowają żadnych informacji o danych wejściowych. Zachowają jedynie parametry i reprezentację modelu.

W przypadku uczenia maszynowego oznacza to, że stan modelu nie zmienia się w czasie. Dlatego w przypadku uczenia maszynowego online (uczenia adaptacyjnego), w którym nowe dane są dostępne w kolejności sekwencyjnej w czasie rzeczywistym i są wykorzystywane do aktualizacji modelu, konieczne będzie użycie biblioteki PyTorch, TensorFlow lub innej platformy, która obsługuje te funkcjonalności.

## Budowa potoku

W uczeniu maszynowym utworzenie i ocena modelu często wymagają wykonania wielu sekwencyjnych kroków. Biblioteka MLlib udostępnia dedykowany obiekt `Pipeline` umożliwiający konstruowanie sekwencji etapów uruchamianych jako całość. Potok `MLlib` jest estymatorem z dedykowanym parametrem `stages`. Każdy etap wchodzący w skład tego parametru jest transformerem lub estymatorem.

Wcześniej w tym rozdziale zdefiniowaliśmy obiekt haszowania (`hasher`), selektor (`selector`) i algorytm `GaussianMixture` (`gm`). Teraz połączymy je w potok. W tym celu przypiszemy do parametru `stages` tablicę:

```
from pyspark.ml import Pipeline
pipeline = Pipeline(stages=[hasher, selector, gm])
# dopasowanie potoku do danych szkoleniowych
model = pipeline.fit(data)
```

Należy pamiętać, aby ustawić kolumny wejściowe i wyjściowe zgodnie z kolejnością etapów! Na przykład kolumny wyjściowe obiektu `hasher` mogą stanowić dane wejściowe dla faz obiektów `selector` lub `gm`. Należy dążyć do generowania tylko tych kolumn, które będą potrzebne.

Niepoprawne zainicjowanie potoków może spowodować zgłoszenie wielu wyjątków. Jeśli dodajesz poszczególne fazy na bieżąco, pamiętaj o zainicjowaniu właściwości `stages` pustą listą w następujący sposób:

```
Pipeline(stages=[])
```

## Jak działa podział dla API Pipeline?

Ponieważ egzemplarz potoku jest estymatorem, możemy przekazać potok do dowolnej funkcji, która przyjmuje estymator w roli argumentu. Obejmuje to wszystkie opisane wcześniej funkcje realizujące podział zbioru danych, na przykład `CrossValidator`.

Oto przykład, jak to działa:

```
from pyspark.ml import Pipeline
pipeline = Pipeline(stages=[hasher,selector, gm])

cv = CrossValidator(estimator=pipeline, estimatorParamMaps=grid,
                    evaluator=evaluator, collectSubModels=True,
                    numFolds=3)
```

Jak widać, jest to całkiem proste!

# Utrwalanie

Ważną częścią potoku uczenia maszynowego jest utrwalanie wyników poprzez zapisywanie ich na dysku. Dzięki temu możliwe jest wdrożenie modelu w środowisku testowym lub produkcyjnym, udostępnienie go współpracownikom lub po prostu zapisanie w celu późniejszego wykorzystania. Biblioteka MLlib zapewnia tę funkcjonalność za pomocą metody `.write().save(`*`ścieżka_do_modelu`*`)` dla wszystkich swoich modeli, w tym `PipelineModel`:

```
path = "/cv_model"
cv_model.write().save(path)
```

Aby załadować model MLlib z dysku, musisz znać klasę modelu użytą do jego zapisania. W naszym przypadku algorytm `CrossValidator` tworzy obiekt `CrossValidatorModel`. Właśnie tego modelu używamy do ładowania modelu za pomocą funkcji `load`:

```
from pyspark.ml.tuning import CrossValidatorModel
read_model_from_disk = CrossValidatorModel.load(path)
```

Wykonanie tej metody spowodowało załadowanie modelu do pamięci i teraz jest on gotowy do użycia.

Możesz także wyeksportować model do formatu przenośnego, na przykład ONNX, a następnie skorzystać ze środowiska wykonawczego ONNX w celu jego uruchomienia. Trzeba jednak pamiętać, że nie wszystkie modele w bibliotece MLlib to obsługują. Ten i inne formaty zostaną omówione w rozdziale 8.

# Podsumowanie

W tym rozdziale przedstawiłam wprowadzenie do nadzorowanych i nienadzorowanych algorytmów uczenia maszynowego MLlib. Omówiłam zagadnienia ich szkolenia i oceny oraz konstruowania potoku umożliwiającego ustrukturyzowaną pracę zespołową. W rozdziale zamieściłam wiele informacji i wniosków na temat pracy z biblioteką MLlib. Warto do tego wrócić, gdy dowiesz się więcej o uczeniu maszynowym i platformie Spark.

W kolejnych rozdziałach dowiesz się, jak wykorzystać całą dotychczasową pracę i poszerzyć możliwości uczenia maszynowego Sparka dzięki połączeniu go z innymi frameworkami, takimi jak PyTorch i TensorFlow.

# Skorowidz


## A

abstrakcja RDD, 53
akcelerator AI, 39
algorytm SVM, 105
algorytmy
  klasyfikacji MLlib, 128
  uczenia głębokiego, 39
  uczenia maszynowego, 125
Apache
  Arrow, 114
  Spark, 46
    abstrakcje danych, 53
    architektura oprogramowania, 50
    architektura rozproszona, 48
    biblioteka MLlib, 58
    biblioteka scikit-learn, 58
    interfejsy API, 53
    obiekty DataFrame, 54, 57
    pozyskiwanie danych, 76
    przetwarzanie danych tabelarycznych, 79
    przetwarzanie obrazów, 77
    wyodrębnianie cech, 114
API
  Estimator, 187, 193
  Horovod Estimator, 167
  Keras, 187
  Pipeline, 148
  SparkDatasetConverter, 159
  TF Core, 187
architektura
  Apache Spark, 46
  klient-serwer, 30
  peer-to-peer, P2P, 30, 35
  rozproszona geograficznie, 30
autograd, 201, 203
  rozproszony, 208, 214
automatyczne różniczkowanie, 200

## B

bariera, 27
biblioteka
  MLflow, 251
  MLlib, 58
  pandas, 57
  Petastorm, 157
  PySpark, 49
  PyTorch, 198
  scikit-learn, 58
  SQLAlchemy, 73
  TensorFlow, 170

## C

cecha
  surowa, 104
  wywnioskowana, 104
cechowanie, featurization, 104, 107
  obrazów, 111
    działania na obrazach, 112
    użycie API Sparka, 114
  tekstu, 119
    metoda bag-of-words, 120
    metoda TF-IDF, 121
    n-gramy, 122
    skanowanie, 122
cechy DAL, 155
cykl życia
  modelu uczenia maszynowego, 60, 231
  oprogramowania, 61

## D


DAL, data access layer, 155
  cechy, 155
  wybór warstwy, 157
dane
  liczbowe, 87, 89
  obrazów, 89
    cechowanie, 111
  Parquet, 175
  tabelaryczne, 79
  tekstowe, 84
    cechowanie, 119
DataFrame, 54, 57, 88
  biblioteki pandas, 117
  Sparka, 117
DDP, distributed data-paralel, 205, 206
deserializacja, 114
dopasowanie
  nadmierne, 32
  niedostateczne, 32
dostrajanie, 143
dryf
  danych, 240
  koncepcji, 243
  modelu, 243
drzewa decyzyjne, 34


## E


eksperymenty dostrajania, 143
ekstraktory, 108
epoka, 67
estymator, 104
ewaluatory
  nadzorowane, 140
  nienadzorowane, 142


## F


funkcja
  blokująca, 210
  haszująca, 104
  optymalizacji, 134
  remote, 210
  straty, 174
funkcje UDF, 115
futura, 210


## G


graf
  obliczeniowy, 199
  sieci neuronowej, 173


## H


harmonogramowanie rozproszonego systemu, 166
hiperparametry, 36, 143
Hydrogen, 164
  barierowy tryb wykonania, 165
  harmonogramowanie, 166


## I


interfejsy API
  wykorzystywane do szkolenia, 187
inżynieria cech, 103
  techniki, 106
  wymagania, 106


## K


klasteryzacja, 136
klastry TensorFlow, 174
klasyfikacja, 127
komunikacja
  asynchroniczna, 31
  synchroniczna, 31
konfiguracja środowiska lokalnego, 42
korelacja, 98
  Pearsona, 99
  Spearmana, 100
kwalifikacja, 75


## M


macierz pomyłek, confusion matrix, 140
magazyn
  cech, feature store, 123
  Parquet, 163
MapReduce, 26
marshaling danych, 80
miara sylwetki, silhouette measure, 142
MLflow, 62, 251
  definiowanie wrappera, 251
  funkcja UDF Sparka, 255

komponenty, 63, 65
Model Registry, 70
rejestrowanie modeli, 70
Models, 69
Projects, 68
Tracking, 65
pamięć masowa, 72
rejestrowanie przebiegów, 66
tabele bazy danych, 73
użytkownicy platformy, 64
wzorzec
Model jako usługa, 254
Model w ramach usługi, 255
zarządzanie cyklem życia, 60
MLlib, 58, 81, 248
algorytmy klasyfikacji, 128
cechowanie, 107
dane tekstowe, 84
dwa klastry, 153
ewaluatory nadzorowane, 140
ewaluatory nienadzorowane, 142
potoki produkcyjne, 249
szkolenie modeli, 125
transformatory, 83
typy danych, 81
wywołania API selektorów, 109
MobileNetV2, 188
monoid, 151
MPI, Message Passing Interface, 26

## N

normalizacja danych, 107

## O

obrazy, 77, 89
cechowanie, 111
etykiety, 89
format, 77
format Parquet, 93
kompresja, 93
przetwarzanie, 112
wyodrębnianie rozmiaru, 91
oprogramowanie produkcyjne, 236
optymalizator, 204
rozproszony, 208, 214

## P

pamięć współdzielona, 27
pandas
dekorator pandas_udf, 115
funkcje UDF, 115
ramki DataFrame, 57
parametry, 28, 143
Petastorm, 157, 194, 224
API SparkDatasetConverter, 159
jako magazyn Parquet, 163
ładowanie danych, 224
pętla sprzężenia zwrotnego, 248
platforma MLflow, 62
potoki uczenia maszynowego, 146
pozyskiwanie, ingestion, 75
danych, 76
profilowanie, 172
programowanie funkcyjne, 56
projekt Hydrogen, 164
propagacja
w przód, 165, 173
wsteczna, backpropagation, 39, 165, 174
przepływ pracy, 147
przesunięcie dziedziny rozkładu, 244
przetwarzanie
danych tabelarycznych, 79
obrazów, 77
rozproszone, 25
model barierowy, 27
MapReduce, 26
MPI, 26
pamięć współdzielona, 27
serwer parametrów, 28, 35
równoległe zadań, 36
właściwe, processing, 80
wstępne, preprocessing, 75, 80
danych obrazów, 89
transformatory MLlib, 83
Py4J, 49
PySpark, 49
tworzenie schematu, 51
uruchamianie kodu, 57
PyTorch, 198
autograd, 201
autograd rozproszony, 214
graf obliczeniowy, 199

PyTorch
komunikacja
peer-to-peer, P2P, 222
zbiorowa, c10d, 206, 215
ładowanie danych, 224
marudne węzły robocze, 228
obiekt
AutogradMeta, 203
Variable, 203
porównanie z TensorFlow, 230
rozproszony optymalizator, 214
szkolenie
oparte na RPC, 205, 207
rozproszone, 204
równoległe, DDP, 205, 206
tensory, 201
typy danych, 227
uruchamianie zdalne, 208, 209
wywołania API, 219
niskopoziomowe, 223

## R

rasteryzacja, 77
RDD, resilient distributed dataset, 53
referencje, 211
regresja, 131
RPC, 180, 207

## S

scikit-Learn, 58
selektory, 108
serwer parametrów, 28, 35, 175
sieć neuronowa, 173
MobileNetV2, 188
skalowanie
w pionie, 25
w poziomie, 25
skośność, skewness, 97
spadek gradientu, 107
Spark
funkcja UDF, 255
SparkDatasetConverter, 159
sprzężenie zwrotne, 248
statystyki opisowe, 94
obiekt Summarizer, 95
strategia
CentralStorageStrategy, 181
MirroredStrategy, 181
MultiWorkerMirroredStrategy, 182
ParameterServerStrategy, 179
TPUStrategy, 186
strategie szkolenia rozproszonego, 177
struktury danych, 81
systemy
rekomendacji, 135
rozproszone, 28
architektura, 30
komunikacja, 31
scentralizowane, 29
topologie, 28
zdecentralizowane, 29
szkolenie modeli
użycie MLlib, 125

## T

tensor, 201
TensorFlow, 170
algorytmy optymalizacji, 203
API Estimator, 193
API Keras, 187
API niestandardowych pętli szkoleniowych, 191
błędy konwersji, 196
klastry, 174
ładowanie danych Parquet, 175
porównanie z PyTorch, 230
strategia
CentralStorageStrategy, 181
MirroredStrategy, 181
MultiWorkerMirroredStrategy, 182
ParameterServerStrategy, 179
TPUStrategy, 186
wykonywanie
w grafie, 171
zachłanne, 171
wywołania API, 171
testowanie modeli, 145
topologie
fizyczne, 28
logiczne, 28
transformatory, 83
cech kategorialnych, 86
danych tekstowych, 84
dodatkowe, 90
liczbowe, 89

typy danych
MLlib, 81
Pythona, 53

## U

uczenie federacyjne, federated learning, 41
uczenie głębokie, 39
frameworki, 150
uczenie maszynowe
algorytmy, 125
cykl życia, 60, 231
eksperymenty dostrajania, 143
hiperparametry, 143
inżynieria cech, 103
monitorowanie, 239
dryf danych, 240
dryf koncepcji, 243
dryf modelu, 243
mierzenie zmian, 245
przesunięcie dziedziny, 244
system produkcyjny, 247
wskaźniki, 244
nadzorowane, 127
klasyfikacja, 127
regresja, 131
narzędzia, 24
nienadzorowane, 136
klasteryzacja, 136
wydobywanie częstych wzorców, 136
ocena, 139
przepływ pracy, 22, 147
rozproszone, 25
odporność na błędy, 40
prywatność, 41
przenośność, 42
strategie TensorFlow, 177
użycie PyTorch, 198
użycie TensorFlow, 170
wydajność, 36
zarządzanie zasobami, 39
stronniczość, 32
technologie, 24
utrwalanie, 149
wdrażanie, 232
etapy, 255
MLflow, 251
MLlib, 248
oprogramowanie produkcyjne, 236
wzorce, 232–234, 254, 255
wielkoskalowe, 25
zastosowania, 21
zespołowe
metody, 33
scentralizowane, 34, 35
zdecentralizowane drzewa decyzyjne, 34
uczenie reguł asocjacyjnych, association rule learning, 136
uczenie transferowe, transfer learning, 123, 188
utrata
entropii, entropy loss, 135
loga, log loss, 135

## W

walidacja krzyżowa, 145
warstwa dostępu do danych, DAL, 155
wdrażanie, 231
iteracyjne, 255
oprogramowanie produkcyjne, 236
sprzężenie zwrotne, 248
wzorzec
Model jako usługa, 234, 254
Model w ramach usługi, 233, 255
Prognozy zbiorcze, 232
wybieranie, 235
wnioskowanie, inferencing, 123
Word2Vec, 109
współbieżność
danych, 36–38
modeli, 36–38, 207
potoków, 207
wyjątek braku pamięci, 119

## Z

zapisywanie danych, 92
zbiory
danych, 123
szkoleniowe, 144
testowe, 144
zdalne
referencje, RRefs, 208, 211
wywołania procedur, RPC, 180, 207

**Właśnie takiej książki społeczność Sparka wyczekuje od dekady!**

**Andy Petrella,** autor książki *Fundamentals of Data Observability*

Jeśli chcesz dostosować swoją pracę do większych zbiorów danych i bardziej złożonych kodów, potrzebna Ci jest znajomość technik rozproszonego uczenia maszynowego. W tym celu warto poznać frameworki Apache Spark, PyTorch i TensorFlow, a także bibliotekę MLlib. Biegłość w posługiwaniu się tymi narzędziami przyda Ci się w całym cyklu życia oprogramowania — nie tylko ułatwi współpracę, ale również tworzenie powtarzalnego kodu.

Dzięki tej książce nauczysz się holistycznego podejścia, które zdecydowanie usprawni współpracę między zespołami. Najpierw zapoznasz się z podstawowymi informacjami o przepływach pracy związanych z uczeniem maszynowym przy użyciu Apache Spark i pakietu PySpark. Nauczysz się też zarządzać cyklem życia eksperymentów dla potrzeb uczenia maszynowego za pomocą biblioteki MLflow. Z kolejnych rozdziałów dowiesz się, jak od strony technicznej wygląda korzystanie z platformy uczenia maszynowego. W książce znajdziesz również opis wzorców wdrażania, wnioskowania i monitorowania modeli w środowisku produkcyjnym.

## Najciekawsze zagadnienia:

- cykl życia uczenia maszynowego i MLflow
- inżynieria cech i przetwarzanie wstępne za pomocą Sparka
- szkolenie modelu i budowa potoku
- budowa systemu danych z wykorzystaniem uczenia głębokiego
- praca TensorFlow w trybie rozproszonym
- skalowanie systemu i tworzenie jego wewnętrznej architektury

**Adi Polak** jest doświadczoną inżynierką, wiceprezeską do spraw programistów w firmie Treeverse, członkinią wielu grup eksperckich. Bierze udział w organizowaniu takich konferencji jak Data + AI Summit by Databricks, Current by Confluent i Scale by the Bay. Doświadczenie w uczeniu maszynowym zdobywała, prowadząc badania dla wielu firm z listy Fortune 500.

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-1234-2
9 788328 912342
Cena: 74,90 zł