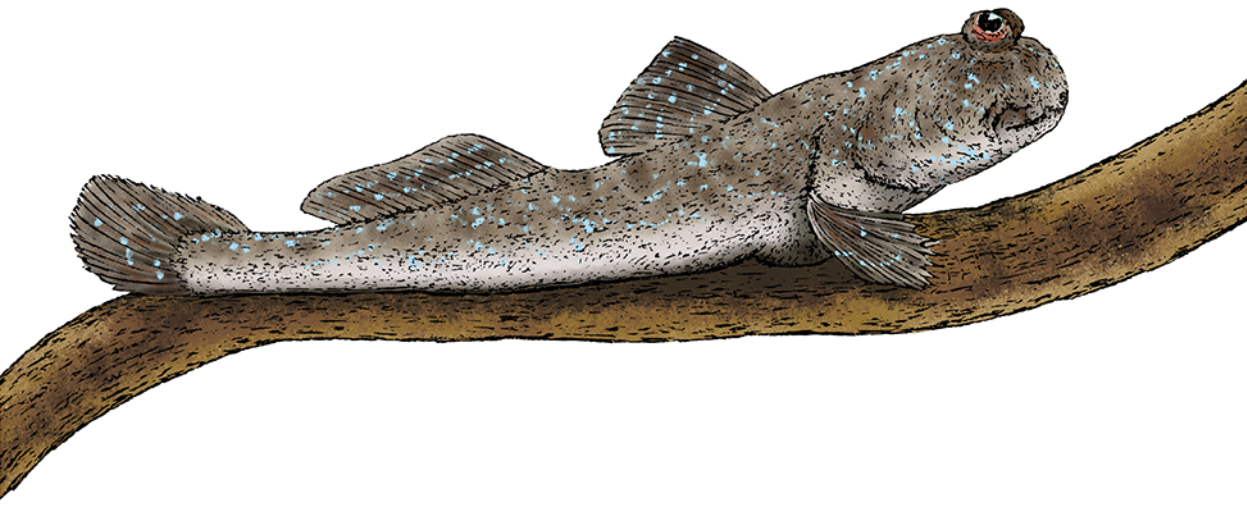
O'REILLY®

Zaawansowana analiza danych w PySpark

Metody przetwarzania informacji na szeroką skalę z wykorzystaniem Pythona i systemu Spark



Akash Tandon, Sandy Ryza,
Uri Laserson, Sean Owen, Josh Wills

Helion

Tytuł oryginału: Advanced Analytics with PySpark: Patterns for Learning from Data at Scale Using Python and Spark

Tłumaczenie: Andrzej Watrak

ISBN: 978-83-8322-069-7

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

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/zaanpy.zip

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)

Printed in Poland.

Spis treści

Słowo wstępne **7**

1. Analiza wielkich zbiorów danych **11**
Praca z wielkimi zbiorami danych 12
Przedstawiamy Apache Spark i PySpark 14
Komponenty 14
PySpark 15
Ekosystem 16
Spark 3.0 17
PySpark i wyzwania w nauce o danych 17
O czym jest ta książka 18

2. Wprowadzenie do analizy danych za pomocą PySpark **19**
Architektura systemu Spark 21
Instalacja interfejsu PySpark 22
Przygotowanie danych 24
Analiza danych za pomocą struktury DataFrame 28
Szybkie statystyki zbiorcze w strukturze DataFrame 32
Przestawienie i przekształcenie struktury DataFrame 34
Złączenie struktur DataFrame i wybór cech 36
Ocena modelu 37
Dalsze kroki 39

3. Rekomendowanie muzyki i dane Audioscrobbler **40**
Zbiór danych 41
Wymagania dla systemu rekomendacyjnego 43
Algorytm naprzemiennych najmniejszych kwadratów 45
Przygotowanie danych 46
Utworzenie pierwszego modelu 48
Wyrywkowe sprawdzanie rekomendacji 52

Ocena jakości rekomendacji 53
Obliczenie wskaźnika AUC 55
Dobór wartości hiperparametrów 56
Przygotowanie rekomendacji 58
Dalsze kroki 60

4. Prognozowanie zalesienia za pomocą drzewa decyzyjnego 61
Drzewa i lasy decyzyjne 62
Przygotowanie danych 64
Pierwsze drzewo decyzyjne 68
Hiperparametry drzewa decyzyjnego 74
Regulacja drzewa decyzyjnego 75
Weryfikacja cech kategorialnych 79
Losowy las decyzyjny 81
Prognozowanie 84
Dalsze kroki 84

5. Wykrywanie anomalii w ruchu sieciowym metodą grupowania według k-średnich 86
Grupowanie według k-średnich 87
Wykrywanie anomalii w ruchu sieciowym 88
Dane KDD Cup 1999 88
Pierwsza próba grupowania 89
Dobór wartości k 92
Wizualizacja w środowisku R 94
Normalizacja cech 97
Zmienne kategorialne 99
Wykorzystanie etykiet i wskaźnika entropii 101
Grupowanie w akcji 102
Dalsze kroki 104

6. Wikipedia, algorytmy LDA i Spark NLP 105
Algorytm LDA 106
Algorytm LDA w interfejsie PySpark 106
Pobranie danych 107
Spark NLP 108
Przygotowanie środowiska 109
Przekształcenie danych 109
Przygotowanie danych za pomocą biblioteki Spark NLP 111
Metoda TF-IDF 114

Wyliczenie wskaźników TF-IDF 115
Utworzenie modelu LDA 116
Dalsze kroki 118

7. Geoprzestrzenna i temporalna analiza tras nowojorskich taksówek 119
Przygotowanie danych 120
Konwersja ciągów znaków na znaczniki czasu 122
Obsługa błędnych rekordów danych 124
Analiza danych geoprzestrzennych 125
Wprowadzenie do formatu GeoJSON 125
Biblioteka GeoPandas 126
Sesjonowanie w interfejsie PySpark 129
Budowanie sesji — dodatkowe sortowanie danych w systemie Spark 130
Dalsze kroki 132

8. Szacowanie ryzyka finansowego 133
Terminologia 134
Metody obliczania wskaźnika VaR 134
Wariancja-kowariancja 135
Symulacja historyczna 135
Symulacja Monte Carlo 135
Nasz model 136
Pobranie danych 137
Przygotowanie danych 137
Określenie wag czynników 140
Losowanie prób 142
Wielowymiarowy rozkład normalny 144
Wykonanie testów 145
Wizualizacja rozkładu zwrotów 148
Dalsze kroki 148

9. Analiza danych genomicznych i projekt BDG 150
Rozdzielenie sposobów zapisu i modelowania danych 151
Przygotowanie pakietu ADAM 153
Przetwarzanie danych genomicznych za pomocą pakietu ADAM 154
Konwersja formatów plików za pomocą poleceń pakietu ADAM 155
Pozyskanie danych genomicznych przy użyciu interfejsu PySpark i pakietu ADAM 155
Prognozowanie miejsc wiązania czynnika transkrypcyjnego na podstawie danych ENCODE 160
Dalsze kroki 164

10. Określanie podobieństwa obrazów za pomocą głębokiego uczenia i algorytmu PySpark LSH 166
PyTorch 167
Instalacja 167
Przygotowanie danych 168
Skalowanie obrazów za pomocą PyTorch 168
Wektoryzacja obrazów za pomocą modelu głębokiego uczenia 169
Osadzenie obrazów 169
Import osadzeń obrazów do pakietu PySpark 171
Określanie podobieństwa obrazów za pomocą algorytmu PySpark LSH 172
Wyszukiwanie najbliższych sąsiadów 173
Dalsze kroki 176

11. Zarządzanie cyklem uczenia maszynowego za pomocą platformy MLflow 177
Cykl uczenia maszynowego 177
Platforma MLflow 179
Śledzenie eksperymentów 180
Zarządzanie modelami uczenia maszynowego i udostępnianie ich 182
Tworzenie i stosowanie projektów za pomocą modułu MLflow Projects 185
Dalsze kroki 187

ROZDZIAŁ 2.

Wprowadzenie do analizy danych za pomocą PySpark

Python jest najczęściej używanym językiem w analizie danych. Perspektywa wykonywania obliczeń statystycznych i tworzenia aplikacji internetowych przy użyciu tego języka zapoczątkowała wzrost jego popularności w 2010 r. Zaowocowało to powstaniem rozwojowego ekosystemu narzędzi, często określanego mianem PyData, i aktywnej społeczności użytkowników zajmujących się analizą danych. Stąd wynika również duża popularność interfejsu PySpark. Naukowcy zajmujący się analizą danych, znający język Python, mogą przy wsparciu szerokiej rzeszy innych użytkowników skutecznie stosować rozproszone przetwarzanie danych przy użyciu systemu Spark. Z tego samego powodu zdecydowaliśmy się wykorzystać interfejs PySpark w opisanych przykładach.

Trudno wyjaśnić, jak skuteczne jest oczyszczanie i analizowanie danych w jednym środowisku, niezależnie od miejsca, w którym dane są przechowywane i przetwarzane. Jest to coś, czego musisz sam doświadczyć i co musisz poznać, a my mamy nadzieję, że nasze przykłady oddadzą choć część tej magii, którą my czuliśmy, gdy zaczynaliśmy używać interfejsu PySpark. Na przykład jest on kompatybilny z biblioteką *pandas*, jedną z najpopularniejszych w ekosystemie PyData. Wykorzystamy tę właściwość w dalszej części rozdziału.

W tym rozdziale poznasz na przykładzie oczyszczania danych niezwykle użyteczną strukturę `DataFrame` interfejsu PySpark. Jest to abstrakcja dla zbiorów danych o regularnej budowie, w których rekordy są wierszami złożonymi z kolumn, a kolumny zawierają dane ściśle określonych typów. Struktura `DataFrame` w systemie Spark jest odpowiednikiem tabeli w relacyjnej bazie danych. Należy jednak pamiętać, że pomimo podobieństwa nazw struktura ta w systemie Spark jest zupełnie innym tworem niż `pandas.DataFrame`, ponieważ reprezentuje zbiór danych rozproszony w klastrze, a nie dane lokalne, gdzie wszystkie wiersze znajdują się w jednym komputerze. Są wprawdzie pewne podobieństwa, do których możesz być przyzwyczajony, w zastosowaniach struktury `DataFrame` i roli, jaką odgrywa ona w bibliotece *pandas* i języku R, jednak ma ona również wiele właściwości charakterystycznych wyłącznie dla systemu Spark. Dlatego traktuj ją z otwartym umysłem.

Oczyszczanie danych to pierwszy, często najważniejszy krok w realizacji każdego projektu w nauce o danych. Wiele wartościowych analiz nie jest wykonywanych z powodu fundamentalnych problemów z jakością danych lub użytych narzędzi, które fałszowały wyniki analizy lub powodowały, że badacze widzieli w nich coś, czego w rzeczywistości nie było. Czy można więc wyobrazić sobie lepszy sposób wprowadzenia do obróbki danych za pomocą interfejsu PySpark i struktury `DataFrame` niż ćwiczenie z oczyszczania danych?

Najpierw opiszemy podstawy interfejsu PySpark i przećwiczymy go na przykładowych danych z repozytorium Machine Learning Repository Uniwersytetu Kalifornijskiego w Irvine w Stanach Zjednoczonych. Ponownie uzasadnimy, dlaczego PySpark jest właściwym wyborem w nauce o danych i przedstawimy jego model programowania. Następnie skonfigurujemy interfejs w lokalnym systemie i klastrze i przeanalizujemy zbiór danych przy użyciu struktury `DataFrame`. Analiza danych za pomocą interfejsu PySpark polega głównie na pracy z powyższą strukturą, dlatego bądź gotów na dokładne zapoznanie się z nią. Przygotujesz się w ten sposób do kolejnych rozdziałów, w których zagłębimy się w różne algorytmy uczenia maszynowego.

Aby realizować zadania związane z analizą danych, nie musisz dokładnie wiedzieć, jak system Spark działa od środka. Jednak znajomość podstawowych pojęć dotyczących jego architektury ułatwi Ci pracę z interfejsem PySpark i podejmowanie właściwych decyzji podczas pisania kodu. Tym się właśnie zajmiemy w następnym podrozdziale.

Struktura DataFrame i zbiory RDD

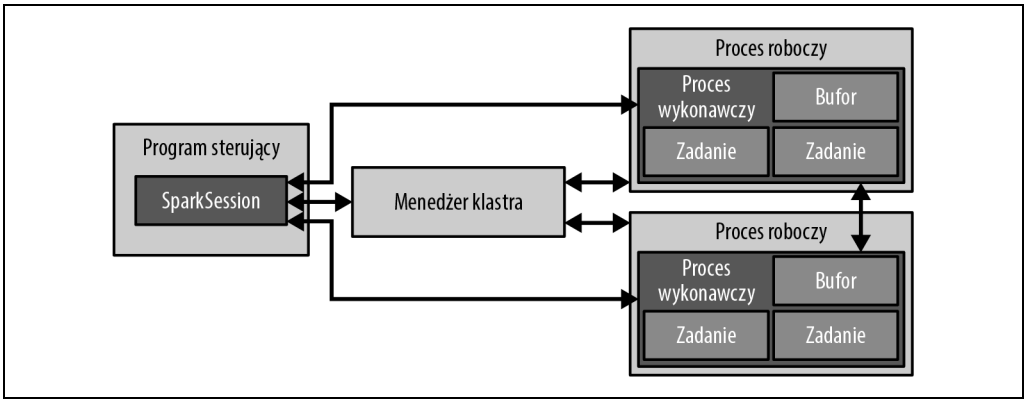
Zbiór RDD (ang. *Resilient Distributed Dataset*, stały, rozproszony zbiór danych) to fundamentalne pojęcie stosowane w systemie Spark, oznaczające zbiór obiektów, które można rozdzielać między wiele komputerów w klastrze. Partycje można przetwarzać równolegle za pomocą niskopoziomowego interfejsu API, oferującego transformacje i akcje. W chwili powstania systemu Spark była to główna abstrakcja oferowana użytkownikom. Z tym modelem wiążą się jednak pewne problemy. Przede wszystkim wszelkie obliczenia wykonywane na bazie zbioru RDD są nieprzezroczyste dla silnika Spark Core. Dlatego nie ma wbudowanej optymalizacji, którą można byłoby stosować. W przypadku korzystania z interfejsu PySpark problem staje się jeszcze bardziej dotkliwy. Nie będziemy jednak się zagłębiać w szczegóły architektoniczne, ponieważ wykraczają one poza zakres tej książki.

W wersji systemu Spark 1.3 wprowadzono strukturę `DataFrame`. Jest to rozproszona, przechowywana w pamięci tabela o nazwanym schemacie, w której wszystkie kolumny mają nazwy i przechowują dane określonych typów, np. liczby całkowite, ciągi znaków, macierze, mapy, liczby rzeczywiste, daty, znaczniki czasu itp. Z naszego punktu widzenia struktura ta jest tablicą. Istnieje również zestaw typowych operacji (łączenie, agregowanie), które można na niej wykonywać. Dzięki temu Spark jest w stanie skonstruować odpowiedni plan wykonania operacji, co przekłada się na większą wydajność przetwarzania danych w porównaniu ze zbiorem RDD.

W tej książce skupimy się wyłącznie na strukturze `DataFrame`. Jeśli chcesz zgłębić różnice między tą strukturą a zbiorem RDD, doskonałym źródłem informacji jest książka *Learning Spark* (wyd. O'Reilly).

Kod wykorzystujący interfejs PySpark i strukturę `DataFrame` powinien cechować się wydajnością porównywalną z kodem napisanym w języku Scala. Funkcje UDF i zbiory RDD pogarszają wydajność.

Architektura systemu Spark



Rysunek 2.1. Architektura systemu Spark

Rysunek 2.1 przedstawia architekturę systemu Spark i jego wysokopoziomowe komponenty. Aplikacje są zestawami niezależnych procesów uruchamianych lokalnie lub w klastrze. Każda aplikacja składa się z programu sterującego, menedżera klastra i zestawu procesów wykonawczych. Program sterujący jest centralnym komponentem odpowiedzialnym za rozdzielanie zadań między procesy wykonawcze i jest tylko jeden. Skalowanie aplikacji polega na zwiększaniu liczby procesów roboczych. Menedżer klastra po prostu zarządza zasobami.

Spark jest rozproszonym systemem do równoległego przetwarzania danych. Im większa liczba partycji, tym większe zrównoleglenie operacji. Partycjonowanie pozwala osiągnąć wydajne zrównoleglenie. Dzięki podziałowi danych na części, czyli partycje, procesy wykonawcze przetwarzają tylko te dane, które znajdują się blisko nich, minimalizując w ten sposób obciążenie sieci. Oznacza to, że każdemu procesowi jest przypisana partycja, którą przetwarza. Należy o tym pamiętać podczas podejmowania decyzji o partycjonowaniu systemu.

Programowanie w systemie Spark zaczyna się od uzyskania dostępu do zbioru danych, zwykle zapisanego w określonym formacie, np. Parquet, i będącego częścią rozproszonego, stałego systemu plików, takiego jak na przykład Hadoop Distributed File System (HDFS) lub chmura AWS S3. Tworzenie programu w systemie Spark zazwyczaj polega na wykonaniu kilku związanych ze sobą kroków:

1. Zdefiniowanie transformacji wykonywanych na danych wejściowych.
2. Wykonanie operacji zapisujących przetworzone dane na trwałych nośnikach lub zwracających wyniki do lokalnej pamięci komputera z programem sterującym. W idealnym przypadku operacje te są wykonywane przez procesy robocze, widoczne po prawej stronie rysunku 2.1.
3. Lokalne przetworzenie wyników uzyskanych w środowisku rozproszonym. Ułatwia to podjęcie decyzji o następnych transformacjach i operacjach do wykonania.

Należy pamiętać, że wszystkie abstrakcje wyższego rzędu w interfejsie PySpark opierają się na pierwotnej idei systemu Spark: interakcji zapisywania i przetwarzania danych. Świadomość tej zasady pozwala skutecznie stosować system Spark w analizie danych.

Teraz zainstaluj i skonfiguruj interfejs PySpark na swoim komputerze, abyś mógł rozpocząć analizę danych. Jest to jednorazowe ćwiczenie, po którym będziesz mógł uruchamiać przykładowe kody zawarte w rozdziale tym i w następnych.

Instalacja interfejsu PySpark

W prezentowanych w tej książce przykładowych kodach przyjęte jest założenie, że masz dostęp do wersji systemu Spark 3.1.1 lub nowszej. Na potrzeby opisanych przykładów zainstalujesz interfejs PySpark dostępny w repozytorium PyPI (*https://pypi.org/project/pyspark*).

Wpisz następujące polecenie:

```
$ pip3 install pyspark
```

Pamiętaj, że interfejs PySpark wymaga wersji języka Java 8 lub nowszej. Możesz również zainstalować komponenty SQL, ML i MLib, których będziesz potrzebował później:

```
$ pip3 install pyspark[sql,ml,mllib]
```

Podczas instalowania interfejsu z repozytorium PyPI są pomijane biblioteki wymagane do uruchamiania kodu napisanego w językach Scala, Java i R. Pełna wersja instalacyjna jest dostępna na stronie systemu Spark (*https://spark.apache.org/downloads.html*). Instrukcja instalacji systemu na lokalnym komputerze i w klastrze znajduje się w dokumentacji (*https://spark.apache.org/docs/latest*).

Teraz możesz wpisać polecenie `pyspark-shell`, uruchamiające interpreter REPL (ang. *read-eval-print loop*, pętla wczytaj-wykonaj-pokaż) dla języka Python, zawierający kilka rozszerzeń charakterystycznych dla systemu Spark. Jest on podobny do powłoki Python lub IPython, z której być może korzystałeś. Jeżeli jednak będziesz wykonywał przykłady na własnym komputerze, możesz uruchomić lokalny klaster Spark, podając parametr `local[N]`, w którym `N` oznacza liczbę uruchamianych wątków. Zamiast tej liczby możesz użyć symbolu `*`, oznaczającego liczbę rdzeni procesora dostępnych w Twoim komputerze. Na przykład w celu uruchomienia klastra wykorzystującego osiem wątków na ośmiordzeniowym komputerze użyj następującego polecenia:

```
$ spark-shell --master local[*]
```

Aplikacja Spark jest często określana mianem **klastra**. Jest to jednak logiczna abstrakcja, inna niż klaster fizyczny (zestaw komputerów).

Jeżeli masz dostęp do klastra z uruchomioną wersją środowiska Hadoop obsługującą funkcjonalność YARN, możesz uruchamiać zadania Spark, wysyłając parametr `yarn` do menedżera Spark:

```
$ pyspark-shell --master yarn --deploy-mode client
```

W pozostałych przykładach w tej książce w poleceniu `pyspark-shell` nie będzie wykorzystywany argument `--master`, jednak należy go stosować odpowiednio do używanego środowiska.

Aby w pełni wykorzystać zasoby dostępne w powłoce Spark, musisz określić dodatkowe argumenty ww. polecenia. Lista dostępnych argumentów pojawi się po wpisaniu polecenia `pyspark --help`. Na przykład jeżeli Spark będzie uruchamiany na lokalnym komputerze, możesz użyć argumentu `--driver-memory 2g`, umożliwiającego przydzielenie lokalnemu procesowi 2 gigabajtów pamięci. Konfiguracja pamięci dla funkcjonalności YARN jest bardziej skomplikowana, a odpowiednie argumenty, na przykład `--executor-memory`, są opisane w dokumentacji systemu Spark (*https://spark.apache.org/docs/latest/running-on-yarn.html*) w części poświęconej powyższej funkcjonalności.

System Spark oficjalnie obsługuje cztery rodzaje klastrów: samodzielny, YARN, Kubernetes i Mesos. Więcej informacji na ten temat można znaleźć w dokumentacji na stronie *https://spark.apache.org/docs/latest/cluster-overview.html*.

Po wpisaniu powyższych poleceń pojawi się wiele komunikatów wyświetlanych podczas inicjalizacji systemu Spark, ale powinien pojawić się również rysunek ASCII z kilkoma dodatkowymi komunikatami i znakiem zachęty:

```
Python 3.6.12 |Anaconda, Inc.| (default, Sep 8 2020, 23:10:56)
[GCC 7.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.0.1
      /_/
Using Python version 3.6.12 (default, Sep 8 2020 23:10:56)
SparkSession available as 'spark'.
```

Wpisz w powłoce polecenie `:help`. Pojawi się pytanie o uruchomienie trybu interaktywnej pomocy lub o wyświetlenie informacji o konkretnych obiektach Pythona. Oprócz informacji dotyczącej polecenia `:help` widoczny jest komunikat `SparkSession available as spark` (Sesja Spark jest dostępny jako obiekt spark). Jest to informacja o obiekcie `SparkSession` stanowiącym punkt wejścia do wszystkich operacji i danych w systemie Spark. Idź dalej i wpisz polecenie `spark`:

```
spark
...
<pyspark.sql.session.SparkSession object at DEADBEEF>
```

Interpreter REPL wyświetlił nazwę obiektu w postaci tekstowej. W przypadku obiektu `SparkSession` jest to po prostu jego nazwa wraz z szesnastkowym adresem w pamięci (ciąg `DEADBEEF` oznacza adres; widoczna tu wartość zmienia się po każdorazowym uruchomieniu powłoki). W interaktywnej powłoce instancję `SparkSession` tworzy program sterujący, natomiast w aplikacji Spark należy ją utworzyć samodzielnie.

Począwszy od wersji Spark 2.0, obiekt `SparkSession` jest głównym punktem wejścia do wszystkich operacji i danych. Za jego pośrednictwem można również uzyskać dostęp do wcześniej stosowanych punktów wejścia, takich jak `SparkContext`, `SQLContext`, `HiveContext`, `SparkConf` i `StreamingContext`.

Co można zrobić ze zmienną `spark`? `SparkSession` to obiekt, z którym skojarzone są różne metody. Można je zobaczyć w powłoce PySpark, wpisując nazwę zmiennej, po niej kropkę i naciskając klawisz tabulacji:

```
 spark.[Tab]
...
spark.Builder(                  spark.conf
spark.newSession(               spark.readStream
spark.stop(                     spark.udf
spark.builder                   spark.createDataFrame(
spark.range(                    spark.sparkContext
spark.streams                   spark.version
spark.catalog                   spark.getActiveSession(
spark.read                      spark.sql(
spark.table(
```

Spośród wszystkich metod obiektu `SparkSession` najczęściej będziemy używać tych, które tworzą strukturę `DataFrame`.

Po skonfigurowaniu interfejsu PySpark możesz przygotować zbiór danych i zacząć go przetwarzać za pomocą dostępnego interfejsu API. Tym się właśnie zajmiemy w następnym podrozdziale.

Przygotowanie danych

Repozytorium Machine Learning Repository Uniwersytetu Kalifornijskiego w Irvine jest rewelacyjnym źródłem różnych interesujących (i bezpłatnych) danych wykorzystywanych w badaniach i dydaktyce. Dane, które będziemy analizować, zostały użyte w badaniu nad wiązaniem rekordów wykonanym w jednym z niemieckich szpitali w 2010 r. Baza zawiera kilka milionów par rekordów z informacjami o pacjentach. Rekordy powiązane są według różnych pól, na przykład imion, nazwisk, adresów lub dat urodzenia. Każdemu z pól została przypisana ocena zgodności od 0,0 do 1,0, określająca podobieństwo ciągów znaków. Następnie rekordy zostały odpowiednio oznaczone ręcznie, w zależności od tego, czy dotyczyły tej samej osoby, czy nie. Oryginalne wartości samych pól, wykorzystane do utworzenia zbioru danych, zostały usunięte w celu ochrony danych osobowych pacjentów, natomiast wartości liczbowe, oceny zgodności pól i oznaczenia poszczególnych par rekordów (czy są zgodne, czy nie) zostały opublikowane na potrzeby przeprowadzania badań nad wiązaniem rekordów.

Aby pobrać dane z repozytorium, wpisz w powłoce systemu operacyjnego następujące polecenia:

```
$ mkdir linkage
$ cd linkage/
$ curl -L -o donation.zip http://bit.ly/1Aoywaq
$ unzip donation.zip
$ unzip 'block_*.zip'
```

Jeżeli masz pod ręką klaster Hadoop, możesz utworzyć katalog w systemie HDFS na blok danych i skopiować tam pliki z danymi:

```
$ hadoop dfs -mkdir linkage
$ hadoop dfs -put block_*.csv linkage
```

Wiązanie rekordów danych

Ogólny charakter problemu jest następujący: mamy dużą liczbę rekordów danych pochodzących z jednego lub kilku źródeł i istnieje prawdopodobieństwo, że część rekordów dotyczy tych samych podmiotów, na przykład klientów, pacjentów, adresów firmy czy wydarzeń. Każdy rekord ma pewną liczbę atrybutów, na przykład nazwisko, adres, datę urodzenia, a my musimy wykorzystać te atrybuty w celu znalezienia rekordów opisujących te same podmioty. Niestety, wartości tych atrybutów nie są idealne: mają różne formaty, zawierają błędy, brakuje w nich pewnych informacji. Wszystko to powoduje, że prosty test jakości atrybutów powoduje, że nie zostanie wykryta duża liczba duplikatów rekordów. Rozważmy dla przykładu listę firm pokazaną w tabeli 2.1.

Tabela 2.1. Problem wiązania rekordów danych

Nazwa	Adres	Miasto	Województwo	Telefon
Kawa u Jana, sklep	aleja Zachodzącego Słońca 12	Gdynia-Orłowo	Pom.	(58)-123-45-67
Kawa u Jana	al. Zachodzącego Słońca 12	Gdynia	Pom.	123-45-67
Polskie Kawiarnie, lokal 1234	al. Zachodzącego Słońca 30	Gdynia	Pom.	(22)-123-45-67
Polskie Kawiarnie, biuro	al. Zachodzącego Słońca 30	Gdynia	Pomorskie	(22)-123-45-67

Pierwsze dwa rekordy w powyższej tabeli opisują ten sam mały sklep z kawą, mimo że błąd we wpisie sugeruje, że są to dwa sklepy w różnych miastach (Gdynia-Orłowo i Gdynia). Natomiast dwa następne rekordy w rzeczywistości dotyczą różnych siedzib tej samej sieci handlowej, które przez przypadek mają ten sam adres: pierwszy z nich określa adres sklepu, a drugi adres biura. Oba rekordy zawierają oficjalny numer telefonu głównej siedziby firmy w Warszawie.

Powyższy przykład pokazuje, dlaczego wiązanie rekordów jest takie trudne: choć obie pary rekordów wyglądają podobnie, kryteria zastosowane do określenia, czy są to duplikaty, czy nie, są różne. Człowiek potrafi te rekordy rozpoznać i określić na pierwszy rzut oka, ale nauczyć komputer tej umiejętności jest bardzo trudno.

Problem, który przeanalizujemy w tym rozdziale, jest w literaturze i w praktyce określany na różne sposoby: wyodrębnianie informacji, deduplikacja rekordów, łączenie i oczyszczanie danych, oczyszczanie listy. W pozostałej części rozdziału będziemy to zagadnienie określać **wiązaniem rekordów danych**.

Do utworzenia struktury `DataFrame` zawierającej nasz zbiór rekordów użyj obiektu `SparkSession`. Konkretnie mówiąc, wywołaj metodę `csv()` klasy `Reader`:

```
prev = spark.read.csv("linkage/block*.csv")
...
prev
...
DataFrame[_c0: string, _c1: string, _c2: string, _c3: string,...
```

Przyjęliśmy założenie, że wszystkie kolumny w pliku CSV są typu `string`, a ich nazwy to `_c0`, `_c1`, `_c2` itd. Wywołując w powłoce interpretera metodę `show()` struktury `DataFrame`, można wyświetlić jej nagłówek:

```
prev.show(2)
...
+-----+-----+-------------+-------------+-------------+-------------+------+------+
|  _c0|  _c1|          _c2|          _c3|          _c4|          _c5|   _c6|   _c7|
+-----+-----+-------------+-------------+-------------+-------------+------+------+
| id_1| id_2| cmp_fname_c1|cmp_fname_c2|cmp_lname_c1|cmp_lname_c2|cmp_sex|cmp_bd|
| 3148| 8326|            1|            ?|            1|            ?|     1|     1|
|14055|94934|            1|            ?|            1|            ?|     1|     1|
|33948|34740|            1|            ?|            1|            ?|     1|     1|
|  946|71870|            1|            ?|            1|            ?|     1|     1|
```

Zgodnie z oczekiwaniami pierwszy wiersz struktury `DataFrame` zawiera nazwy kolumn, z których składa się plik CSV. Niektóre kolumny zawierają znak zapytania. Należy go traktować jako brak wartości.

Idealnie byłoby, gdyby system Spark nie tylko określił poprawne nazwy kolumn, ale też typy zawartych w nich danych. Na szczęście klasa `Reader` oferuje opcje realizujące tę funkcjonalność. Ich pełna lista jest dostępna w dokumentacji interfejsu PySpark (*https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/dataframe.html*). Na razie powiązane dane będziemy odczytywać i interpretować w następujący sposób:

```
parsed = spark.read.option("header", "true").option("nullValue", "?").\
         option("inferSchema", "true").csv("linkage/block*.csv")
```

Po wywołaniu metody `show()` można się przekonać, że kolumnom zostały nadane poprawne nazwy, a znaki zapytania zastąpione wartościami `null`. Aby się dowiedzieć, jakie typy danych zostały przyjęte w poszczególnych kolumnach, należy wyświetlić schemat struktury `DataFrame`, jak niżej:

```
parsed.printSchema()
...
root
|-- id_1: integer (nullable = true)
|-- id_2: integer (nullable = true)
|-- cmp_fname_c1: double (nullable = true)
|-- cmp_fname_c2: double (nullable = true)
...
```

Każda instancja klasy `Column` zawiera nazwę kolumny, najściślejszy typ odpowiadający danym zawartym we wszystkich rekordach oraz pole logiczne wskazujące, czy kolumna może zawierać wartości `null`, domyślnie przyjmujące wartość `true`. Aby poprawnie wywnioskować typ danych, Spark musi przeanalizować zbiór w *dwóch* przebiegach: jeden raz w celu określenia typów danych we wszystkich kolumnach i drugi raz w celu właściwego ich przetworzenia. W razie potrzeby pierwszy przebieg może wykonać, wykorzystując próbkę danych.

Jeżeli schemat danych, który można zastosować dla danego pliku, jest znany zawczasu, można utworzyć instancję klasy `pyspark.sql.types.StructType` i przekazać ją klasie `Reader` za pomocą funkcji `schema()`. W przypadku bardzo dużego zbioru danych można w ten sposób znacznie zwiększyć wydajność operacji, ponieważ system Spark nie będzie musiał wykonywać dodatkowego przebiegu w celu określenia typu danych w poszczególnych kolumnach.

Poniższy przykład ilustruje definiowanie schematu za pomocą klas `StructType` i `StructField`:

```
from pyspark.sql.types import *
schema = StructType([StructField("id_1", IntegerType(), False),
```

```
    StructField("id_2", StringType(), False),
    StructField("cmp_fname_c1", DoubleType(), False)])

spark.read.schema(schema).csv("...")
```

Schemat można również zdefiniować za pomocą instrukcji DDL (ang. *Data Definition Language*, język definicji danych):

```
schema = "id_1 INT, id_2 INT, cmp_fname_c1 DOUBLE"
```

Formaty i źródła danych

System Spark zawiera klasy `DataFrameReader` i `DataFrameWriter` umożliwiające odczytywanie i zapisywanie struktur `DataFrame` w różnych formatach. Nie licząc opisanego wyżej pliku CSV, strukturalne dane można odczytywać z następujących źródeł:

`parquet`
: Najpopularniejszy, kolumnowy format danych (domyślny).

`orc`
: Inny kolumnowy format danych.

`json`
: Format umożliwiający domniemanie schematu, podobnie jak w przypadku pliku CSV.

`jdbc`
: Relacyjna baza danych, z którą połączenie jest nawiązywane w standardzie JDBC (ang. *Java Database Connectivity*, łącze do bazy danych w języku Java).

`avro`
: Strumieniowe źródło danych, np. Apache Kafka, umożliwiające wydajną serializację i deserializację komunikatów.

`text`
: Format wiążący poszczególne wiersze ze strukturą `DataFrame` zawierającą tylko jedną kolumnę typu `string`.

`image`
: Pliki obrazów, odczytywane z katalogu i umieszczane w strukturze `DataFrame` zawierającej jedną kolumnę w schemacie obrazów.

`libsvm`
: Popularny format plików tekstowych, zawierających rozrzedzone dane opatrzone etykietami.

`binary`
: Pliki binarne, przekształcane w osobne wiersze struktury `DataFrame` (nowość wprowadzona w systemie Spark 3.0).

`xml`
: Prosty format tekstowy opisujący strukturalną informację, np. dokumenty, dane, konfigurację, książki (dostępny w pakiecie `spark-xml`).

Metody klasy `DataFrameReader` wywołuje się za pośrednictwem metody `read()` instancji klasy `SparkSession`. Dane z pliku odczytuje się za pomocą kombinacji metod `format()` i `load()` lub uproszczonych metod obsługujących wbudowane formaty:

```
d1 = spark.read.format("json").load("file.json")
d2 = spark.read.json("file.json")
```

W powyższym przykładzie zmienne `d1` i `d2` zawierają te same dane w formacie JSON. Wszystkie formaty mają własne opcje, ustawiane za pomocą metody `option()`, użytej wcześniej do odczytania pliku CSV.

Aby zapisać dane, należy użyć klasy `DataFrameWriter` i metody `write()` w strukturze `DataFrame`. Klasa ta obsługuje te same formaty danych, co `DataFrameReader`. Poniższe wiersze kodu zapisują zawartość zmiennej `d1` w pliku w formacie `parquet`:

```
d1.write.format("parquet").save("file.parquet")
d1.write.parquet("file.parquet")
```

Przy próbie zapisania struktury `DataFrame` w istniejącym pliku system Spark domyślnie zgłasza błąd. Reakcję systemu w takich sytuacjach można kontrolować za pomocą klasy `DataFrameWriter` i metody `mode()`. Istniejący plik można zastąpić nowym, dopisać do niego dane lub zaniechać zapisywania i pozostawić plik bez zmian:

```
d2.write.format("parquet").mode("overwrite").save("file.parquet")
```

Tryb określa się za pomocą ciągów `"overwrite"`, `"append"` lub `"ignore"`.

Struktura `DataFrame` zawiera kilka metod umożliwiających odczytywanie danych z klastra za pomocą interpretera REPL na lokalnym komputerze. Najprostszą z nich jest `first()`, zwracająca pierwszy element struktury:

```
parsed.first()
...
Row(id_1=3148, id_2=8326, cmp_fname_c1=1.0, cmp_fname_c2=None,...
```

Metoda ta przydaje się przy sprawdzaniu poprawności danych w zbiorze. My jednak jesteśmy zainteresowani uzyskaniem większej liczby próbek `DataFrame` do analizy. Jeżeli struktura zawiera niewielką liczbę rekordów, można za pomocą metody `toPandas()` lub `collect()` wyświetlić jej całą zawartość w postaci tablicy. W przypadku bardzo dużych struktur `DataFrames` użycie powyższych metod może być niebezpieczne, skutkować przepełnieniem pamięci i zgłoszeniem wyjątku. Ponieważ nie wiemy jeszcze, jak duży jest nasz zbiór powiązanych danych, wstrzymajmy się z użyciem tych metod.

W kilku kolejnych częściach rozdziału, w celu lepszego oczyszczenia i przeanalizowania powiązanych rekordów, będziemy naprzemiennie rozwijać kod na lokalnym komputerze i przetwarzać dane w klastrze. Jeżeli jednak potrzebujesz chwili, aby ochłonąć po wpisaniu kodu z mnóstwem niesamowitych funkcjonalności, przyjmiemy to ze zrozumieniem.

Analiza danych za pomocą struktury DataFrame

Interfejs API struktury `DataFrame` zawiera pakiet potężnych narzędzi, dobrze znanych badaczom danych używającym języków Python i SQL. W tym podrozdziale opiszemy te narzędzia i ich wykorzystanie do wiązania rekordów.

Transformacje i akcje

Operacja tworzenia struktury `DataFrame` nie skutkuje rozproszonym przetworzeniem danych w klastrze. Struktury te są raczej logicznymi zbiorami reprezentującymi pośrednie etapy procesu przetwarzania danych. Operacje, które system Spark wykonuje na danych, można zakwalifikować do dwóch grup: transformacji i akcji.

Transformacje są realizowane leniwie. Oznacza to, że ich wyniki nie są wyliczane natychmiast, tylko ustawiane w kolejce do wyliczenia. Dzięki temu system Spark może optymalizować plany zapytań. Rozproszone przetwarzanie danych odbywa się dopiero po zainicjowaniu odpowiedniej akcji na strukturze. Na przykład akcja `count()` zwraca liczbę obiektów w strukturze:

```
df.count()
...
15
```

Akcja `collect()` zwraca obiekt typu `Array` (tablicę) zawierający wszystkie obiekty typu `Row` (wiersz) zawarte w strukturze `DataFrame`. Tablica jest umieszczana w pamięci, a nie w klastrze:

```
df.collect()
[Row(id='12', department='sales'), ...
```

Akcje nie tylko zwracają wyniki lokalnym procesom. Na przykład akcja `save()` zapisuje zawartość struktury `DataFrame` w trwałej pamięci:

```
df.write.format("parquet").("user/ds/mynumbers")
```

Należy pamiętać, że argumentem klasy `DataFrameReader` jest nazwa katalogu zawierającego pliki tekstowe. Oznacza to, że w powyższym przykładzie zadanie Spark będzie odwoływać się do katalogu *mynumbers*.

Spójrzmy na schemat przetworzonej struktury `DataFrame` i kilka pierwszych wierszy danych:

```
parsed.printSchema()
...
root
 |-- id_1: integer (nullable = true)
 |-- id_2: integer (nullable = true)
 |-- cmp_fname_c1: double (nullable = true)
 |-- cmp_fname_c2: double (nullable = true)
 |-- cmp_lname_c1: double (nullable = true)
 |-- cmp_lname_c2: double (nullable = true)
 |-- cmp_sex: integer (nullable = true)
 |-- cmp_bd: integer (nullable = true)
 |-- cmp_bm: integer (nullable = true)
 |-- cmp_by: integer (nullable = true)
 |-- cmp_plz: integer (nullable = true)
 |-- is_match: boolean (nullable = true)
...

parsed.show(5)
...
+------+----+------------+------------+------------+------------+.....
| id_1| id_2|cmp_fname_c1|cmp_fname_c2|cmp_lname_c1|cmp_lname_c2|.....
+-----+-----+------------+------------+------------+------------+.....
| 3148| 8326|         1.0|        null|         1.0|        null|.....
|14055|94934|         1.0|        null|         1.0|        null|.....
```

```
|33948|34740|          1.0|        null|          1.0|        null|.....
|  946|71870|          1.0|        null|          1.0|        null|.....
|64880|71676|          1.0|        null|          1.0|        null|.....
```

- Pierwsze dwa pola zawierają identyfikatory będące liczbami całkowitymi, oznaczającymi odpowiadające sobie rekordy danych pacjentów.
- Następnych dziewięć pól zawiera liczby rzeczywiste (niektórych może brakować) oznaczające ocenę zgodności danych w różnych polach rekordów, na przykład w nazwisku pacjenta, dacie urodzenia czy adresie. Liczby całkowite są stosowane wtedy, gdy dopuszczalna jest jedynie pełna zgodność (1) lub jej brak (0), natomiast liczby zmiennoprzecinkowe, gdy zgodność może być częściowa.
- Ostatnie pole typu logicznego (zawierające wartość `true` lub `false`) zawiera informację, czy rekordy danych w parze są ze sobą zgodne.

Naszym celem jest opracowanie prostego klasyfikatora, który na podstawie istniejących wyników zgodności rekordów pacjentów będzie prognozował, czy dany rekord jest zgodny. Zacznijmy od określenia liczby rekordów, z którymi mamy do czynienia. Użyjemy do tego celu metody `count()`:

```
parsed.count()
...
5749132
```

Ten zbiór danych jest na tyle mały, że zmieści się w pamięci jednego węzła w klastrze, a nawet lokalnego komputera, jeżeli nie masz dostępu do klastra.

Do tej pory podczas przetwarzania danych Spark za każdym razem otwierał plik, analizował wiersze i wykonywał żądaną akcję, na przykład wyświetlał kilka pierwszych wierszy danych lub zliczał rekordy. Przy każdej kolejnej analizie Spark będzie wykonywał te same operacje, nawet jeżeli zbiór zostanie przefiltrowany lub zagregowany do niewielkiej liczby rekordów. Oznacza to, że zasoby obliczeniowe nie będą wykorzystywane optymalnie. Dlatego dane po przetworzeniu należy zapisać w wynikowej postaci w klastrze, aby nie trzeba było ich przetwarzać na nowo przy każdej operacji. System Spark jest przygotowany na taką ewentualność. Utworzoną strukturę `DataFrame` można za pomocą jej metody `cache()` zapisać w pamięci komputera. Zróbmy to teraz:

```
parsed.cache()
```

Po zapisaniu danych kolejną rzeczą, którą powinniśmy sprawdzić, są liczby zgodnych i niezgodnych rekordów:

```
from pyspark.sql.functions import col

parsed.groupBy("is_match").count().orderBy(col("count").desc()).show()
...
+--------+-------+
|is_match|  count|
+--------+-------+
|   false|5728201|
|    true|  20931|
+--------+-------+
```

Zamiast pisać funkcję wyodrębniającą kolumnę `is_match`, po prostu wywołujemy w powłoce REPL metodę `groupBy()` struktury `DataFrame` z nazwą kolumny w argumencie. Następnie wywołujemy metodę `count()`, która zlicza rekordy w każdej grupie. Na koniec sortujemy wyniki według kolumny

count w kolejności malejącej i wyświetlamy je w czytelnej formie za pomocą metody show(). System wewnętrznie określa optymalny sposób uzyskania i wyświetlenia wyników. Ten przykład pokazuje, jak prosto, szybko i czytelnie można analizować dane za pomocą systemu Spark.

Zwróć uwagę, że istnieją dwa sposoby odwoływania się do kolumny danych w strukturze DataFrame: za pomocą nazwy, tak jak np. w metodzie groupBy("is_match"), lub obiektu Column, który w powyższym przykładzie zwróciła metoda col(), wywołana z nazwą kolumny count w argumencie. W większości przypadków oba sposoby są dobre. Tutaj użyliśmy metody col(), aby następnie wywołać metodę desc() obiektu reprezentującego kolumnę count.

Metody agregujące dane w strukturze DataFrame

Struktura DataFrame posiada oprócz count() metodę agg(), która użyta wraz z metodami zawartymi w kolekcji pyspark.sql.functions umożliwia wyliczanie bardziej zaawansowanych zagregowanych danych, takich jak suma, wartości minimalna i maksymalna oraz odchylenie standardowe. Na przykład, aby wyliczyć średnią i odchylenie standardowe wartości w kolumnie cmp_sex, należy użyć następującego polecenia:

```
from pyspark.sql.functions import avg, stddev

parsed.agg(avg("cmp_sex"), stddev("cmp_sex")).show()
+-----------------+--------------------+
|     avg(cmp_sex)|stddev_samp(cmp_sex)|
+-----------------+--------------------+
|0.955001381078048|  0.2073011111689795|
+-----------------+--------------------+
```

Pamiętaj, że domyślnie system Spark wylicza odchylenie standardowe z próby. Dostępna jest jednak metoda stddev_pop() wyliczająca odchylenie standardowe z populacji.

Zapewne zauważyłeś, że nazwy metod w strukturze DataFrame są podobne do klauzul SQL. To nie jest przypadek. Każdą strukturę można traktować jako tabelę w bazie danych i formułować zapytania zgodnie z dobrze znaną, potężną składnią języka SQL. Najpierw trzeba podać nazwę, jaką silnik Spark SQL powinien przypisać obiektowi parsed, ponieważ „parsed", czyli taka jak nazwa obiektu, nie jest dostępna:

```
parsed.createOrReplaceTempView("linkage")
```

Przeanalizowana struktura jest tabelą *przejściową*, dostępną tylko w trakcie sesji REPL. Jednak za pomocą zapytań Spark SQL można również przetwarzać tabele zapisane trwale w systemie HDFS. Warunkiem jest skonfigurowanie systemu Spark tak, aby łączył się z magazynem metadanych Apache Hive, zawierającym schematy i lokalizacje ustrukturyzowanych zbiorów danych.

Po zarejestrowaniu tymczasowej tabeli w silniku Spark SQL można ją przetworzyć np. za pomocą następującego zapytania:

```
spark.sql("""
  SELECT is_match, COUNT(*) cnt
  FROM linkage
  GROUP BY is_match
  ORDER BY cnt DESC
```

```
""").show()
...
+--------+-------+
|is_match|    cnt|
+--------+-------+
|   false|5728201|
|    true|  20931|
+--------+-------+
```

Po utworzeniu instancji klasy `SparkSession` można za pomocą klasy `Builder` i jej metody `enableHive` ↪`Support()` przełączyć silnik Spark SQL z domyślnego trybu zgodnego z ANSI 2003 w tryb HiveQL.

Podłączenie silnika Spark SQL do magazynu Hive

Silnik Spark SQL można podłączyć do magazynu Hive za pomocą pliku *hive-site.xml* lub zapytania HiveQL. W tym celu należy wywołać metodę `enableHiveSupport()` klasy `Builder`, będącej częścią klasy `SparkSession`:

```
spark_session = SparkSession.builder.master("local[4]").\
                                  enableHiveSupport().getOrCreate()
```

Wszystkie tabele w magazynie Hive można traktować jako struktury `DataFrame` i przetwarzać je za pomocą zapytań Spark SQL. Uzyskane wyniki można zapisywać w magazynie Hive, aby je później przetwarzać za pomocą narzędzi takich jak Hive, Apache Impala lub Presto.

Czy do analizowania danych za pomocą interfejsu PySpark należy używać zapytań Spark SQL, czy metod struktury `DataFrame`? Oba sposoby mają swoje zalety i wady. Język SQL jest popularny, a proste zapytania są czytelne. Ponadto za pomocą sterowników JDBC/ODBC można przetwarzać dane zapisane w bazach, np. PostgreSQL, i narzędziach typu Tableau. Wada języka SQL polega jednak na tym, że trudno jest za jego pomocą definiować zaawansowane, wieloetapowe analizy w dynamiczny, czytelny i testowalny sposób. W tych obszarach przewagę ma interfejs API struktury `DataFrame`. W tej książce stosowany jest zarówno język Spark SQL, jak i interfejs API struktury. Jako ćwiczenie pozostawiamy Ci weryfikację dokonanych przez nas wyborów i wykonanie opisanych operacji odmiennym sposobem.

Aby uzyskać takie wskaźniki jak liczba czy średnia wartości, można wywołać osobno odpowiednie metody struktury `DataFrame`. Jednak interfejs PySpark oferuje lepszy sposób wyliczania zagregowanych danych. Tym zagadnieniem zajmiemy się w następnym podrozdziale.

Szybkie statystyki zbiorcze w strukturze DataFrame

Wiele rodzajów analiz można równie dobrze przeprowadzić za pomocą zapytań SQL, jak i interfejsu API struktury `DataFrame`. Jest jednak kilka operacji, których wykonanie za pomocą języka SQL jest żmudne. Jedną z nich, szczególnie przydatną, jest wyliczenie minimum, maksimum, średniej oraz odchylenia standardowego wszystkich niepustych wartości w zadanej kolumnie. W interfejsie PySpark metoda realizująca tę operację ma nazwę `describe()`, taką samą jak w bibliotece *pandas*:

```
summary = parsed.describe()
...
summary.show()
```

Obiekt `summary` typu `DataFrame` zawiera kolumny odpowiadające poszczególnym zmiennym zawartym w obiekcie `parsed` oraz kolumnę również o nazwie `summary`, w której znajdują się ciągi `count`, `mean`, `stddev`, `min` i `max` opisujące wartości znajdujące się w poszczególnych wierszach i wymienionych wyżej kolumnach. Za pomocą metody `select()` można wyświetlić wybrane kolumny, dzięki czemu łatwiej jest je odczytywać i porównywać:

```
summary.select("summary", "cmp_fname_c1", "cmp_fname_c2").show()
+-------+------------------+------------------+
|summary|      cmp_fname_c1|      cmp_fname_c2|
+-------+------------------+------------------+
|  count|           5748125|            103698|
|   mean|0.7129024704436274|0.9000176718903216|
| stddev|0.3887583596162788|0.2713176105782331|
|    min|               0.0|               0.0|
|    max|               1.0|               1.0|
+-------+------------------+------------------+
```

Zwróć uwagę na różnice między wartościami znajdującymi się w wierszu `count` i kolumnach `cmp_fname_c1` oraz `cmp_fname_c2`. Niemal wszystkie rekordy zawierają niepuste pola `cmp_fname_c1`, natomiast niepustych pól `cmp_fname_c2` jest niecałe 2%. Aby zbudować przydatny klasyfikator, musimy oprzeć się na wartościach, które są obecne w niemal wszystkich rekordach, chyba że ich brak również jest ważną informacją dotyczącą zgodności rekordów.

Mając ogólne rozeznanie w rozkładzie zmiennych w danych, sprawdźmy, jak są one skorelowane z wartościami w kolumnie `is_match`. Zatem następnym krokiem będzie wyliczenie tych samych statystyk zbiorczych dla podzbiorów struktury `DataFrame`, odpowiadających zgodnym i niezgodnym rekordom. Przefiltrujmy dane za pomocą metody `where()` i zapytania SQL lub za pomocą `Column`, a następnie użyjmy metody `describe()` z wynikowymi strukturami `DataFrame`:

```
matches = parsed.where("is_match = true")
match_summary = matches.describe()

misses = parsed.filter(col("is_match") == False)
miss_summary = misses.describe()
```

Ciąg znaków umieszczony w argumencie metody `where()` zawiera warunek stosowany w klauzuli `WHERE` w języku SQL. Natomiast w argumencie metody `filter()` jest użyty obiekt reprezentujący kolumnę `is_match` z operatorem `==` i wartością logiczną `False`, ponieważ jest tu wykorzystana składnia języka Python, a nie SQL. Zwróć uwagę, że metoda `where()` jest równoważna metodzie `filter()`. W powyższym przykładzie można pierwszą wywołać tak samo jak drugą (i odwrotnie) i kod będzie działał tak samo.

Teraz porównajmy obiekty `match_summary` i `miss_summary` typu `DataFrame` i sprawdźmy, jak zmieniają się rozkłady wartości w zależności od tego, czy rekordy są zgodne, czy nie. Choć użyty zbiór jest mały, przeprowadzenie tego rodzaju porównania jest dość żmudne. Trzeba dokonać transpozycji obu tabel, tj. zamienić wiersze na kolumny, aby następnie je złączyć i przeanalizować zbiorcze statystyki. Tego typu przekształcenie badacze danych nazywają **przestawieniem** (ang. *pivoting*). W następnym podrozdziale dowiesz się, jak to się robi.

Przestawienie i przekształcenie struktury DataFrame

Transpozycji struktury `DataFrame` można dokonać, korzystając wyłącznie z funkcji oferowanych przez interfejs PySpark. Jest jednak jeszcze inny sposób. Interfejs umożliwia konwertowanie struktury `DataFrame` z formatu Spark na stosowany w bibliotece *pandas* i odwrotnie. Najpierw przekonwertujemy strukturę na format biblioteki *pandas*, następnie odpowiednio ją przekształcimy i przekonwertujemy z powrotem na format Spark. Biblioteka umieszcza struktury `DataFrame` w pamięci komputera, ale można bezpiecznie wykonać powyższą operację, ponieważ obiekty `summary`, `match_summary` i `miss_summary` są dość małe. W następnych rozdziałach do przekształcania większych zbiorów danych będziemy korzystać z funkcji systemu Spark.

Konwersja struktury `DataFrame` do formatu biblioteki *pandas* i z powrotem jest możliwa dzięki projektowi Apache Arrow, którego celem jest wydajne przesyłanie danych pomiędzy maszyną JVM (ang. *Java Virtual Machine*, wirtualna maszyna Java) a procesem Python. Biblioteka PyArrow jest zależnością komponentu Spark SQL, instalowanego za pomocą polecenia `pip install pyspark[sql]`.

Przekształć obiekt `summary` w strukturę `DataFrame` biblioteki *pandas*:

```
summary_p = summary.toPandas()
```

Teraz możesz używać metod biblioteki, zawartych w obiekcie `summary_p`:

```
summary_p.head()
...
summary_p.shape
...
(5,12)
```

Za pomocą dobrze znanych metod wykonaj transpozycję, aby zamienić wiersze na kolumny:

```
summary_p = summary_p.set_index('summary').transpose().reset_index()
...
summary_p = summary_p.rename(columns={'index':'field'})
...
summary_p = summary_p.rename_axis(None, axis=1)
...
summary_p.shape
...
(11,6)
```

Pomyślnie przekształciłeś obiekt `summary_p`, będący strukturą `DataFrame` biblioteki *pandas*. Teraz zamień go za pomocą metody `SparkSession.createDataFrame()` w strukturę `DataFrame` systemu Spark:

```
summaryT = spark.createDataFrame(summary_p)
...
summaryT.show()
...
+------------+-------+------------------+------------------+---+------+
|       field|  count|              mean|            stddev|min|   max|
+------------+-------+------------------+------------------+---+------+
|        id_1|5749132|  33324.48559643438| 23659.859374488064|  1| 99980|
|        id_2|5749132|  66587.43558331935| 23620.487613269695|  6|100000|
```

```
|cmp_fname_c1|5748125| 0.7129024704437266|0.38875835961628014|0.0|   1.0|
|cmp_fname_c2| 103698| 0.9000176718903189| 0.2713176105782334|0.0|   1.0|
|cmp_lname_c1|5749132| 0.3156278193080383| 0.3342336339615828|0.0|   1.0|
|cmp_lname_c2|   2464| 0.3184128315317443|0.36856706620066537|0.0|   1.0|
|     cmp_sex|5749132|  0.955001381078048|0.20730111116897781|  0|     1|
|      cmp_bd|5748337|0.22446526708507172|0.41722972238462636|  0|     1|
|      cmp_bm|5748337|0.48885529849763504| 0.4998758236779031|  0|     1|
|      cmp_by|5748337| 0.2227485966810923| 0.4160909629831756|  0|     1|
|     cmp_plz|5736289|0.00552866147434343|0.07414914925420046|  0|     1|
+------------+-------+-------------------+-------------------+---+------+
```

To jeszcze nie koniec. Wyświetl schemat obiektu `summaryT`:

```
summaryT.printSchema()
...
root
 |-- field: string (nullable = true)
 |-- count: string (nullable = true)
 |-- mean: string (nullable = true)
 |-- stddev: string (nullable = true)
 |-- min: string (nullable = true)
 |-- max: string (nullable = true)
```

W schemacie zawartym w obiekcie `summary`, utworzonym za pomocą metody `describe()`, wartości w poszczególnych kolumnach są ciągami znaków. Zatem aby przeanalizować statystyki zbiorcze, musimy je zamienić na liczby zmiennoprzecinkowe:

```
from pyspark.sql.types import DoubleType
for c in summaryT.columns:
   if c == 'field':
     continue
   summaryT = summaryT.withColumn(c, summaryT[c].cast(DoubleType()))
...
summaryT.printSchema()
...
root
 |-- field: string (nullable = true)
 |-- count: double (nullable = true)
 |-- mean: double (nullable = true)
 |-- stddev: double (nullable = true)
 |-- min: double (nullable = true)
 |-- max: double (nullable = true)
```

Wiesz już, jak transponować strukturę `DataFrame` zawierającą zbiorcze statystyki. Zaimplementuj więc opisaną procedurę w funkcji, która będzie przetwarzała obiekty `match_summary` i `miss_summary`:

```
from pyspark.sql import DataFrame
from pyspark.sql.types import DoubleType

def pivot_summary(desc):
  # Konwersja do struktury DataFrame biblioteki pandas
  desc_p = desc.toPandas()
  # Transpozycja
  desc_p = desc_p.set_index('summary').transpose().reset_index()
  desc_p = desc_p.rename(columns={'index':'field'})
  desc_p = desc_p.rename_axis(None, axis=1)
  # Konwersja do struktury DataFrame systemu Spark
  descT = spark.createDataFrame(desc_p)
```

```
    # Konwersja zawartości kolumn z ciągów znaków na liczby zmiennoprzecinkowe
    for c in descT.columns:
        if c == 'field':
          continue
        else:
          descT = descT.withColumn(c, descT[c].cast(DoubleType()))
    return descT
```

Teraz w powłoce Spark przetwórz obiekty `match_summary` i `miss_summary` za pomocą funkcji `pivot_summary()`:

```
match_summaryT = pivot_summary(match_summary)
miss_summaryT = pivot_summary(miss_summary)
```

Po pomyślnej transpozycji struktur `DataFrame` możesz je złączyć i porównać. Zrobisz to w następnym podrozdziale. Wybierzesz również odpowiednie cechy do zbudowania modelu.

Złączenie struktur DataFrame i wybór cech

Do tej pory używaliśmy języka Spark SQL i metod struktury `DataFrame` wyłącznie do filtrowania i agregowania rekordów w zbiorze danych. Jednak za pomocą tych narzędzi można również łączyć struktury (wewnętrznie, zewnętrznie lewo-, prawo- i obustronnie). Struktura `DataFrame` zawiera wprawdzie przeznaczoną do tego celu metodę `join()`, ale często lepiej jest użyć bardziej czytelnej składni Spark SQL, szczególnie gdy łączone struktury zawierają wiele kolumn o takich samych nazwach i trzeba je wyraźnie rozróżniać w wyrażeniach.

Utwórz tymczasowe widoki obiektów `match_summaryT` i `miss_summaryT`, następnie złącz je według kolumny `field` i porównaj wybrane statystyki w wynikowych wierszach:

```
match_summaryT.createOrReplaceTempView("match_desc")
miss_summaryT.createOrReplaceTempView("miss_desc")
spark.sql("""
   SELECT a.field, a.count + b.count total, a.mean - b.mean delta
   FROM match_desc a INNER JOIN miss_desc b ON a.field = b.field
   WHERE a.field NOT IN ("id_1", "id_2")
   ORDER BY delta DESC, total DESC
""").show()
...
+------------+---------+--------------------+
|       field|    total|               delta|
+------------+---------+--------------------+
|     cmp_plz|5736289.0|  0.9563812499852176|
|cmp_lname_c2|   2464.0|  0.8064147192926264|
|      cmp_by|5748337.0|  0.7762059675300512|
|      cmp_bd|5748337.0|   0.775442311783404|
|cmp_lname_c1|5749132.0|  0.6838772482590526|
|      cmp_bm|5748337.0|  0.5109496938298685|
|cmp_fname_c1|5748125.0|  0.2854529057460786|
|cmp_fname_c2| 103698.0| 0.09104268062280008|
|     cmp_sex|5749132.0|0.032408185250332844|
+------------+---------+--------------------+
```

Rzetelna ocena powinna charakteryzować się dwiema właściwościami: jej wartości dla rekordów zgodnych i niezgodnych powinny znacznie się różnić (a więc różnica między średnimi wartościami ocen powinna być duża), a ponadto powinna być dostępna dla większości par rekordów w przetwarzanych

danych. Zatem cecha cmp_fname_c2 nie jest zbyt przydatna — bardzo często brakuje jej wartości, a różnica między jej średnimi wartościami dla rekordów zgodnych i niezgodnych jest relatywnie niewielka — 0,09 w przedziale od 0 do 1. Cecha cmp_sex również nie jest pomocna. Choć jest dostępna dla wszystkich par rekordów, różnica między jej wartościami średnimi wynosi zaledwie 0,03.

Natomiast cechy cmp_plz i cmp_by są doskonałe: istnieją niemal we wszystkich parach rekordów, a pomiędzy ich wartościami średnimi jest duża różnica (ponad 0,77 dla obu ocen). Cechy cmp_bd, cmp_lname_c1 i cmp_bm również wydają się przydatne: zazwyczaj są dostępne w zbiorze danych i jest znaczna różnica między ich wartościami średnimi dla rekordów zgodnych i niezgodnych.

Cechy cmp_fname_c1 i cmp_lname_c2 trudno opisać: pierwsza niemal nie rozróżnia rekordów (różnica między wartościami średnimi wynosi zaledwie 0,28), choć jest zazwyczaj dostępna dla poszczególnych par, natomiast druga wykazuje dużą różnicę między wartościami średnimi, ale niemal zawsze jej brakuje. Nie jest oczywiste, jakie warunki muszą być spełnione, aby wykorzystać te cechy w modelu opartym na przyjętych danych.

Na razie zbudujemy prosty model oceniający podobieństwo par rekordów na podstawie sum zdecydowanie dobrych cech: cmp_plz, cmp_by, cmp_bd, cmp_lname_c1 i cmp_bm. Dla niewielu rekordów, w których brakuje powyższych cech, przyjmiemy zamiast wartości null wartość 0. Ogólną ocenę wiarygodności naszego prostego modelu uzyskamy, tworząc strukturę DataFrame zawierającą oceny i informacje o zgodności rekordów, a następnie sprawdzając, jak skutecznie rozróżniane są rekordy zgodne i niezgodne przy różnych wartościach progowych ocen.

Ocena modelu

Nasza funkcja oceniająca będzie sumować wartości pięciu cech: cmp_lname_c1, cmp_plz, cmp_by, cmp_bd i cmp_bm. Wykorzystamy funkcję expr() zawartą w kolekcji pyspark.sql.functions. Funkcja ta zamienia podany w argumencie ciąg znaków na odpowiednią kolumnę lub kilka kolumn.

Utwórz odpowiedni ciąg znaków:

```
good_features = ["cmp_lname_c1", "cmp_plz", "cmp_by", "cmp_bd", "cmp_bm"]
...
sum_expression = " + ".join(good_features)
...
sum_expression
...
'cmp_lname_c1 + cmp_plz + cmp_by + cmp_bd + cmp_bm'
```

Teraz możesz wykorzystać ciąg sum_expression do wyliczenia oceny. Na potrzeby wyliczenia sumy cech brakujące wartości uzupełnij zerami za pomocą metody DataFrame.fillna():

```
from pyspark.sql.functions import expr
scored = parsed.fillna(0, subset=good_features).\
                            withColumn('score', expr(sum_expression)).\
                            select('score', 'is_match')
...
scored.show()
...
```

```
+-----+--------+
|score|is_match|
+-----+--------+
|  5.0|    true|
|  5.0|    true|
|  5.0|    true|
|  5.0|    true|
|  5.0|    true|
|  5.0|    true|
|  4.0|    true|
...
```

Ostatnim krokiem w tworzeniu funkcji oceniającej jest określenie progowej wartości oceny, po przekroczeniu której można stwierdzić, że dwa rekordy są zgodne. Jeżeli próg będzie za wysoki, wtedy zgodne rekordy będą błędnie kwalifikowane jako niezgodne (tzw. przypadki *fałszywe negatywne*). Natomiast przy zbyt niskim progu rekordy niezgodne będą błędnie kwalifikowane jako zgodne (przypadki *fałszywe pozytywne*). W każdym nietrywialnym modelu trzeba uwzględnić pewien odsetek obu rodzajów przypadków, a wyznaczenie wartości progowej sprowadza się do oszacowania ich kosztów w sytuacji, w której model będzie stosowany.

Podczas określania wartości progowej pomocna jest **tabela kontyngencji**, zwana również **tabelą krzyżową**, zawierająca liczby rekordów, których oceny są wyższe/niże od tej wartości, oraz liczby rekordów zgodnych i niezgodnych. Ponieważ nie wiemy jeszcze, jakiej wartości progowej musimy użyć, napiszmy funkcję, której argumentami będą oceniana struktura `DataFrame` oraz przyjęta wartość progowa. Funkcja ta za pomocą metod struktury `DataFrame` będzie wyliczać tabelę krzyżową:

```
def crossTabs(scored: DataFrame, t: DoubleType) -> DataFrame:
    return  scored.selectExpr(f"score >= {t} as above", "is_match").\
                groupBy("above").pivot("is_match", ("true", "false")).\
                count()
```

Zwróć uwagę, że za pomocą metody `selectExpr()` jest dynamicznie wyliczana wartość pola o nazwie `above` (powyżej) w zależności od argumentu `t`. Wykorzystana jest tu składnia f-ciągu w Pythonie. Jeżeli ciąg znaków zostanie poprzedzony literą `f`, wówczas zmienne o zadanych nazwach zostaną zastąpione ich wartościami (kolejna przydatna sztuczka). Po zdefiniowaniu pola tworzymy tabelę krzyżową za pomocą użytych wcześniej metod `groupBy()`, `pivot()` i `count()`.

Jeżeli przyjmiemy wysoką wartość progową równą 4,0, odpowiadającą średniej z pięciu cech równej 0,8, odfiltrujemy niemal wszystkie niezgodne rekordy i pozostawimy ponad 90% zgodnych:

```
crossTabs(scored, 4.0).show()
...
+-----+-----+-------+
|above|  rue|  false|
+-----+-----+-------+
| true|20871|    637|
|false|   60|5727564|
+-----+-----+-------+
```

Jeżeli przyjmiemy mniejszą wartość progową równą 2,0, wtedy zachowamy wszystkie zgodnie rekordy, ale kosztem znacznego udziału w wyniku rekordów niezgodnych (prawa górna komórka):

```
crossTabs(scored, 2.0).show()
...
```

```
+-----+-----+-------+
|above| true|  false|
+-----+-----+-------+
| true|20931| 596414|
|false| null|5131787|
+-----+-----+-------+
```

Mimo że dla łagodniejszej wartości progowej liczba błędnych wyników jest większa, niżbyśmy sobie tego życzyli, jest to skuteczniejszy filtr, ponieważ usuwa 90% niezgodnych rekordów z rozważanego zbioru danych i pozostawia w nim wszystkie zgodne rekordy. Choć jest to całkiem dobry wynik, można to zrobić jeszcze lepiej. Sprawdź, czy można wykorzystać inne cechy (zarówno z brakującymi, jak i dostępnymi wartościami) i utworzyć funkcję, która skutecznie zidentyfikuje zgodne rekordy, a błędnych wyników będzie mniej niż 100.

Dalsze kroki

Jeżeli ten rozdział stanowił Twoje pierwsze doświadczenie z przygotowywaniem danych i ich analizą za pomocą interfejsu PySpark, mam nadzieję, że przekonałeś się, jak potężne jest to narzędzie. Jeżeli używałeś wcześniej języka Python i systemu Spark, liczymy na to, że polecisz ten rozdział znajomym i kolegom, również jako wprowadzenie do potężnych możliwości tych produktów.

Naszym celem w tym rozdziale było przekazanie Ci informacji niezbędnych do zrozumienia i wykonania pozostałych przykładów w tej książce. Jeżeli jesteś osobą, która najlepiej uczy się na praktycznych przykładach, Twoim kolejnym krokiem będzie zapoznanie się z następnymi rozdziałami, w których przedstawimy bibliotekę uczenia maszynowego MLlib, zaprojektowaną dla systemu Spark.

PySpark: systemowa odpowiedź na problemy inżyniera danych!

Potrzeby w zakresie analizy dużych zbiorów danych i wyciągania z nich użytecznych informacji stale rosną. Spośród dostępnych narzędzi przeznaczonych do tych zastosowań szczególnie przydatny jest PySpark — interfejs API systemu Spark dla języka Python. Apache Spark świetnie się nadaje do analizy dużych zbiorów danych, a PySpark skutecznie ułatwia integrację Sparka ze specjalistycznymi narzędziami PyData. By jednak można było w pełni skorzystać z tych możliwości, konieczne jest zrozumienie interakcji między algorytmami, zbiorami danych i wzorcami używanymi w analizie danych.

Oto praktyczny przewodnik po wersji 3.0 systemu Spark, metodach statystycznych i rzeczywistych zbiorach danych. Omówiono w nim zasady rozwiązywania problemów analitycznych za pomocą interfejsu PySpark, z wykorzystaniem dobrych praktyk programowania w systemie Spark. Po lekturze można bezproblemowo zagłębić się we wzorce analityczne oparte na popularnych technikach przetwarzania danych, takich jak klasyfikacja, grupowanie, filtrowanie i wykrywanie anomalii, stosowane w genomice, bezpieczeństwie systemów IT i finansach. Dodatkowym plusem są opisy wykorzystania przetwarzania obrazów i języka naturalnego. Zaletą jest też szereg rzeczywistych przykładów dużych zbiorów danych i ich zaawansowanej analizy.

Dzięki książce poznasz:

- model programowania w ekosystemie Spark
- podstawowe metody stosowane w nauce o danych
- pełne implementacje analiz dużych publicznych zbiorów danych
- konkretne przypadki użycia narzędzi uczenia maszynowego
- kod, który łatwo dostosujesz do swoich potrzeb

Akash Tandon jest inżynierem danych i przedsiębiorcą, a także współzałożycielem i dyrektorem technicznym firmy Looppanel.

Sandy Ryza kieruje rozwojem projektu Dagster i jest współautorem kodu systemu Apache Spark.

Uri Laserson jest założycielem i dyrektorem technicznym firmy Patch Biosciences.

Sean Owen jest twórcą systemu Apache Spark i członkiem Project Management Committee. Interesuje się uczeniem maszynowym i nauką o danych.

Josh Wills jest inżynierem oprogramowania w firmie WeaveGrid.

Helion

helion.pl

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

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-8322-069-7

9 788383 220697

Cena: 69,00 zł