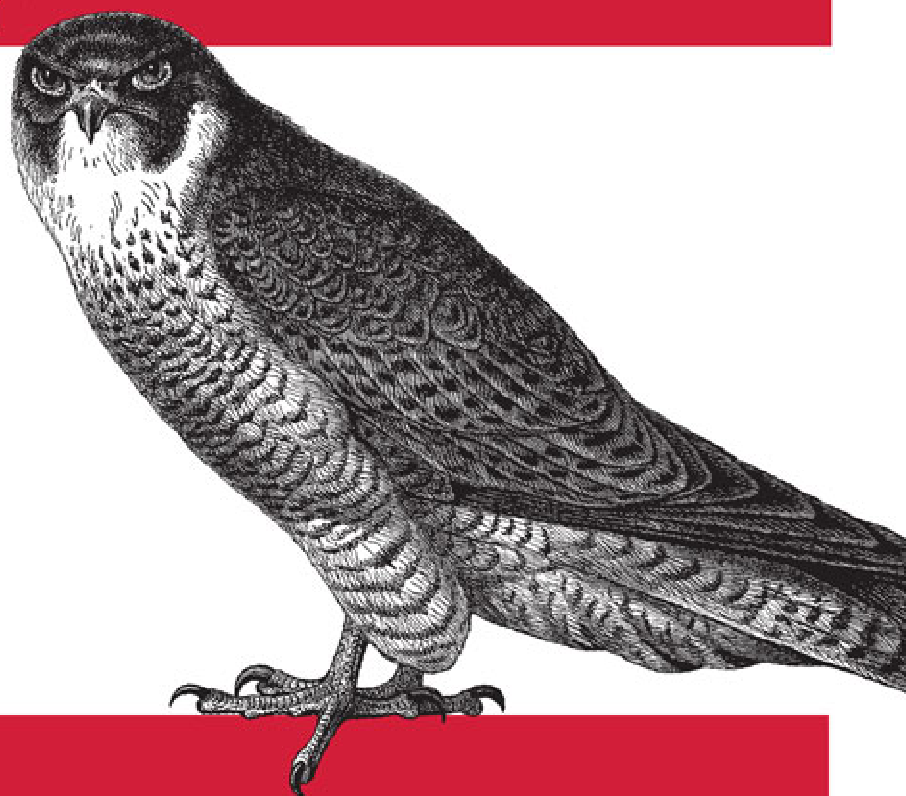
O'REILLY®



Spark

Zaawansowana analiza danych

ANALIZA OGROMNYCH ZBIORÓW DANYCH
NIE MUSI BYĆ WOLNA!

Sandy Ryza, Uri Laserson,
Sean Owen, Josh Wills

Tytuł oryginału: Advanced Analytics with Spark

Tłumaczenie: Andrzej Watrak

ISBN: 978-83-283-1461-0

© 2016 Helion S.A.

Authorized Polish translation of the English edition of Advanced Analytics with Spark, ISBN 9781491912768 © 2015 Sandy Ryza, Uri Laserson, Sean Owen, and Josh Wills.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

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

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

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

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Przedmowa 9

Słowo wstępne 11

1. Analiza wielkich zbiorów danych 13
Wyzwania w nauce o danych 15
Przedstawiamy Apache Spark 16
O czym jest ta książka 18

2. Wprowadzenie do analizy danych za pomocą Scala i Spark 21
Scala dla badaczy danych 22
Model programowania w Spark 23
Wiązanie rekordów danych 23
Pierwsze kroki — powłoka Spark i kontekst SparkContext 24
Przesyłanie danych z klastra do klienta 29
Wysyłanie kodu z klienta do klastra 32
Tworzenie list danych i klas wyboru 33
Agregowanie danych 36
Tworzenie histogramów 38
Statystyki sumaryzacyjne ciągłych wartości 39
Tworzenie współdzielonego kodu wyliczającego statystyki sumaryczne 40
Prosty wybór zmiennych i ocena zgodności rekordów 44
Następny krok 45

3. Rekomendowanie muzyki i dane Audioscrobbler 47
Zbiór danych 48
Algorytm rekomendacyjny wykorzystujący metodę naprzemiennych najmniejszych kwadratów 49
Przygotowanie danych 51

Kup ksi k Pole ksi k

Utworzenie pierwszego modelu 54
Wyrywkowe sprawdzanie rekomendacji 56
Ocena jakości rekomendacji 57
Obliczenie metryki AUC 59
Dobór wartości hiperparametrów 60
Przygotowanie rekomendacji 62
Dalsze kroki 63

4. Prognozowanie zalesienia za pomocą drzewa decyzyjnego 65
Szybkie przejście do regresji 65
Wektory i cechy 66
Przykłady treningowe 67
Drzewa i lasy decyzyjne 68
Dane Covtype 70
Przygotowanie danych 71
Pierwsze drzewo decyzyjne 72
Hiperparametry drzewa decyzyjnego 76
Regulacja drzewa decyzyjnego 77
Weryfikacja cech kategorialnych 79
Losowy las decyzyjny 81
Prognozowanie 83
Dalsze kroki 83

5. Wykrywanie anomalii w ruchu sieciowym metodą grupowania według k-średnich 85
Wykrywanie anomalii 86
Grupowanie według k-średnich 86
Włamania sieciowe 87
Dane KDD Cup 1999 87
Pierwsza próba grupowania 88
Dobór wartości k 90
Wizualizacja w środowisku R 93
Normalizacja cech 94
Zmienne kategorialne 96
Wykorzystanie etykiet i wskaźnika entropii 97
Grupowanie w akcji 98
Dalsze kroki 100

Kup książkę Poleć książkę

6. Wikipedia i ukryta analiza semantyczna 101
Macierz słowo – dokument 102
Pobranie danych 104
Analiza składni i przygotowanie danych 104
Lematyzacja 105
Wyliczenie metryk TF-IDF 106
Rozkład według wartości osobliwych 108
Wyszukiwanie ważnych pojęć 110
Wyszukiwanie i ocenianie informacji za pomocą niskowymiarowej reprezentacji danych 113
Związek dwóch słów 114
Związek dwóch dokumentów 115
Związek słowa i dokumentu 116
Wyszukiwanie wielu słów 117
Dalsze kroki 118

7. Analiza sieci współwystępowań za pomocą biblioteki GraphX 121
Katalog cytowań bazy MEDLINE — analiza sieci 122
Pobranie danych 123
Analiza dokumentów XML za pomocą biblioteki Scala 125
Analiza głównych znaczników i ich współwystępowań 126
Konstruowanie sieci współwystępowań za pomocą biblioteki GraphX 128
Struktura sieci 131
Połączone komponenty 131
Rozkład stopni wierzchołków 133
Filtrowanie krawędzi zakłócających dane 135
Przetwarzanie struktury EdgeTriplet 136
Analiza przefiltrowanego grafu 138
Sieci typu „mały świat” 139
Kliki i współczynniki klastrowania 139
Obliczenie średniej długości ścieżki za pomocą systemu Pregel 141
Dalsze kroki 145

8. Geoprzestrzenna i temporalna analiza tras nowojorskich taksówek 147
Pobranie danych 148
Przetwarzanie danych temporalnych i geoprzestrzennych w systemie Spark 148
Przetwarzanie danych temporalnych za pomocą bibliotek JodaTime i NScalaTime 149

Kup ksi k Pole ksi k

Przetwarzanie danych geoprzestrzennych za pomocą Esri Geometry API i Spray 150
Użycie interfejsu API Esri Geometry 151
Wprowadzenie do formatu GeoJSON 152
Przygotowanie danych dotyczących kursów taksówek 154
Obsługa dużej liczby błędnych rekordów danych 155
Analiza danych geoprzestrzennych 158
Sesjonowanie w systemie Spark 161
Budowanie sesji — dodatkowe sortowanie danych w systemie Spark 162
Dalsze kroki 165

9. Szacowanie ryzyka finansowego metodą symulacji Monte Carlo 167
Terminologia 168
Metody obliczania wskaźnika VaR 169
Wariancja-kowariancja 169
Symulacja historyczna 169
Symulacja Monte Carlo 169
Nasz model 170
Pobranie danych 171
Wstępne przetworzenie danych 171
Określenie wag czynników 174
Losowanie prób 176
Wielowymiarowy rozkład normalny 178
Wykonanie testów 179
Wizualizacja rozkładu zwrotów 181
Ocena wyników 182
Dalsze kroki 184

10. Analiza danych genomicznych i projekt BDG 187
Rozdzielenie sposobów zapisu i modelowania danych 188
Przetwarzanie danych genomicznych za pomocą wiersza poleceń systemu ADAM 190
Format Parquet i format kolumnowy 195
Prognozowanie miejsc wiązania czynnika transkrypcyjnego na podstawie danych ENCODE 197
Odczytywanie informacji o genotypach z danych 1000 Genomes 203
Dalsze kroki 204

Kup ksi k Pole ksi k

11. **Analiza danych neuroobrazowych za pomocą pakietów PySpark i Thunder 205**
Ogólne informacje o pakiecie PySpark 206
Budowa pakietu PySpark 207
Ogólne informacje i instalacja biblioteki pakietu Thunder 209
Ładowanie danych za pomocą pakietu Thunder 210
Podstawowe typy danych w pakiecie Thunder 214
Klasyfikowanie neuronów za pomocą pakietu Thunder 216
Dalsze kroki 221

A **Więcej o systemie Spark 223**
Serializacja 224
Akumulatory 225
System Spark i metody pracy badacza danych 226
Formaty plików 228
Podprojekty Spark 229
MLlib 229
Spark Streaming 230
Spark SQL 230
GraphX 230

B **Nowy interfejs MLlib Pipelines API 231**
Samo modelowanie to za mało 231
Interfejs API Pipelines 232
Przykład procesu klasyfikacji tekstu 233

Skorowidz 236

Kup ksi k Pole ksi k

Kup książkę Poleć książkę

ROZDZIAŁ 3.

Rekomendowanie muzyki i dane Audioscrobbler

Sean Owen

De gustibus non est disputandum.
(O gustach się nie dyskutuje).

Gdy ktoś pyta, czym się zajmuję, aby zarobić na życie, moja odpowiedź wprost: „Nauką o danych" czy „Uczeniem maszynowym" robi wrażenie, ale zazwyczaj wywołuje pełne zdumienia spojrzenia. Nic dziwnego, bo sami badacze danych mają kłopoty z określeniem, co te terminy oznaczają. Czy przechowywanie mnóstwa danych? Przetwarzanie ich? Prognozowanie wartości? Przejdę od razu do odpowiedniego przykładu:

„OK, wiesz, że księgarnia Amazon rekomenduje książki podobne do tej, którą kupiłeś, prawda? Tak! To właśnie to!".

Z praktycznego punktu widzenia **system rekomendacyjny** jest przykładem zastosowania na szeroką skalę algorytmu uczenia maszynowego, który każdy rozumie, a większość użytkowników księgarni Amazon go zna. Jest to powszechnie znana rzecz, ponieważ systemy rekomendacyjne są stosowane wszędzie, począwszy od serwisów społecznościowych, po strony z filmami i sklepy internetowe. Widzimy je również bezpośrednio w akcji. Wiemy, że komputer pobiera piosenki w serwisie Spotify, ale nie wiemy, jak serwer poczty Gmail decyduje, czy odebrana wiadomość jest spamem.

Działanie systemu rekomendacyjnego jest bardziej intuicyjnie zrozumiałe niż funkcjonowanie innych algorytmów uczenia maszynowego. System taki jest wręcz fascynujący. Wszyscy wiemy, że gust muzyczny jest sprawą osobistą i niewytłumaczalną, ale systemy rekomendacyjne zaskakująco dobrze sobie radzą z wyszukiwaniem utworów, o których nawet nie wiemy, że będą nam się podobać.

Ponadto, w branży muzycznej czy filmowej, gdzie systemy te są powszechnie stosowane, dość łatwo stwierdzić, dlaczego polecana pozycja pasuje do historii utworów wybieranych przez danego słuchacza. Jednak nie wszystkie algorytmy klasyfikujące odpowiadają temu opisowi. Na przykład maszyna wektorów nośnych jest zbiorem współczynników i nawet zawodowcom trudno określić, co oznaczają liczby wygenerowane podczas prognozowania wartości.

Czas zatem przymierzyć się do trzech kolejnych rozdziałów, szczegółowo opisujących najważniejsze algorytmy uczenia maszynowego w systemie Spark. Niniejszy rozdział poświęcony jest systemom

Kup ksi k Pole ksi k

rekomendacyjnym, a konkretnie rekomendowaniu muzyki. Stanowi on przystępne wprowadzenie do praktycznego zastosowania systemu Spark, biblioteki MLlib i kilku pojęć z dziedziny uczenia maszynowego, które będą opisane w następnych rozdziałach.

Zbiór danych

W tym rozdziale wykorzystane są dane udostępnione przez system **Audioscrobbler**. Jest to pierwszy system rekomendacyjny, wykorzystywany przez serwis *last.fm*, założony w roku 2002, będący jednym z pierwszych serwisów internetowych udostępniających strumieniową transmisję audycji radiowych. System Audioscrobbler oferował otwarty interfejs API do „scrobblowania" (wysyłania) listy utworów wykonawcy wybranego przez użytkownika. Informacje te posłużyły do zbudowania skutecznego systemu rekomendującego muzykę. Strona przyciągnęła miliony użytkowników, ponieważ zewnętrzne aplikacje i strony internetowe wysyłały informacje o odsłuchiwanych utworach z powrotem do systemu rekomendacyjnego.

W tamtym czasie systemy rekomendacyjne w większości przypadków ograniczały swoje działanie do uczenia się ocen. Zazwyczaj były zatem postrzegane jako narzędzia wykorzystujące dane wejściowe typu „Robert dał Prince'owi ocenę 3,5 gwiazdki".

Dane udostępnione przez Audioscrobbler są interesujące, ponieważ rekordy zawierają jedynie informację typu „Robert słuchał utworu Prince'a". Jest to informacja uboższa niż ocena. Jeżeli Robert słuchał jakiegoś utworu, nie oznacza to, że mu się on podobał. Ja czy Ty również możemy przypadkowo posłuchać utworu jakiegoś wykonawcy, który nie ma dla nas znaczenia, a nawet rozpocząć odtwarzanie całego albumu i wyjść z pokoju.

Jednak użytkownicy znacznie rzadziej oceniają utwory, niż je odtwarzają. Zbiór tych ostatnich danych jest znacznie większy, obejmuje większą liczbę użytkowników i wykonawców, jak również zawiera w sumie więcej informacji niż zbiór danych z ocenami, mimo że pojedynczy rekord zawiera w sobie mniej danych. Tego typu dane są często nazywane **niejawnymi informacjami zwrotnymi**, ponieważ określenie związku pomiędzy użytkownikiem a wykonawcą jest efektem ubocznym innych czynności niż jawne wystawienie oceny czy kliknięcie symbolu kciuka w górę.

Próbkę danych udostępnioną przez serwis *last.fm* w roku 2005 można pobrać w postaci spakowanego archiwum (*http://www-etud.iro.umontreal.ca/~bergstrj/audioscrobbler_data.html*). Wewnątrz niego znajduje się kilka plików. Główny zbiór danych znajduje się w pliku *user_artist_data.txt*. Zawiera on około 141 000 unikatowych identyfikatorów użytkowników i 1,6 miliona unikatowych identyfikatorów wykonawców. Zarejestrowanych zostało około 24,2 miliona przypadków odsłuchania przez danego słuchacza utworu danego wykonawcy oraz liczby odsłuchań.

Ponadto w pliku *artist_data.txt* znajdują się nazwy wykonawców odpowiadających każdemu identyfikatorowi. Zwróć uwagę, że z chwilą rozpoczęcia odtwarzania utworu aplikacja wyświetla nazwę wykonawcy. Nazwa ta może być niestandardowa i zawierać błędy, ale takie przypadki można wykryć dopiero na późniejszym etapie analizy. Na przykład nazwy „Budka Sufler", „Budka S" i „budka suflera" mogą mieć różne identyfikatory w pliku danych, choć oczywiście oznaczają tego samego wykonawcę. Dlatego wśród danych znajduje się również plik *artist_alias.txt*, w którym powiązane są identyfikatory błędnie zapisanych lub zmienionych nazw wykonawców z ich identyfikatorami podstawowymi.

Kup ksi k Pole ksi k

Algorytm rekomendacyjny wykorzystujący metodę naprzemiennych najmniejszych kwadratów

Musimy wybrać algorytm rekomendacyjny odpowiedni dla danych z niejawnymi ocenami. Dane zawierają wyłącznie powiązania pomiędzy użytkownikami a wykonawcami. Nie ma w nich informacji o użytkownikach ani wykonawcach, z wyjątkiem nazw tych ostatnich. Potrzebny jest algorytm, który będzie uczył się preferencji użytkowników bez sięgania do ich atrybutów ani atrybutów wykonawców. Tego typu algorytmy nazywane są zazwyczaj **filtrowaniem kolaboratywnym** (*http://en.wikipedia.org/wiki/Collaborative_filtering*). Na przykład twierdzenie, że dwóch użytkowników ma podobne preferencje muzyczne, ponieważ są w tym samym wieku, nie jest przykładem filtrowania kolaboratywnego. Natomiast dobrym przykładem jest twierdzenie, że dwóch użytkowników lubi ten sam utwór, ponieważ obaj wielokrotnie słuchali podobnych.

Przykładowy zbiór danych jest duży, obejmuje kilkadziesiąt milionów rekordów zawierających liczby odsłuchań utworów. Z drugiej strony jednak zbiór jest mały i ubogi, ponieważ zawiera skąpe informacje. Średnio każdy użytkownik odtwarzał piosenki 171 wykonawców spośród 1,6 miliona wszystkich. Niektórzy użytkownicy słuchali utworów tylko jednego wykonawcy. Nam potrzebny jest algorytm, który przygotuje rekomendacje nawet dla takich użytkowników. Przecież każdy użytkownik bez wyjątku musi zacząć od odsłuchania tylko jednego utworu!

Ponadto potrzebny jest algorytm skalowalny, zarówno pod względem obsługi dużego modelu danych, jak i szybkiego przygotowywania rekomendacji. Zazwyczaj wymaga się, aby rekomendacje były przygotowywane niemal na bieżąco, tj. w ciągu kilku sekund, a nie na drugi dzień.

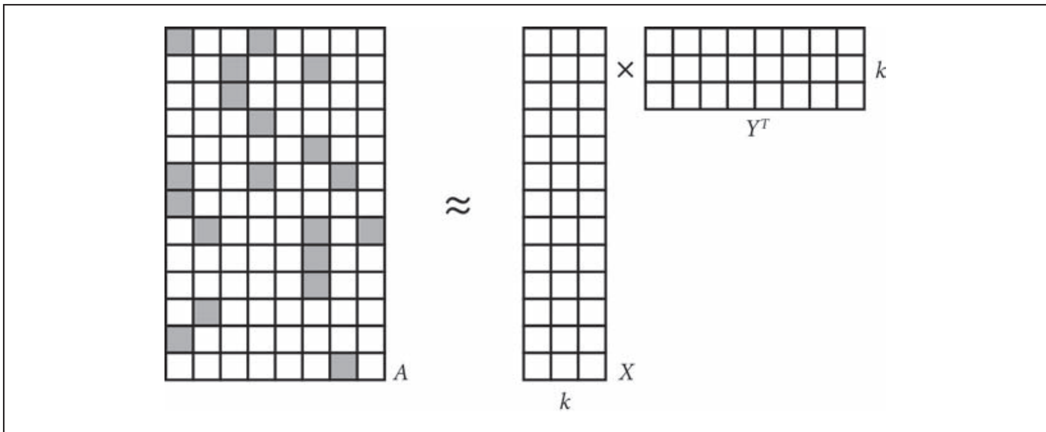
W tym przykładzie zostanie wykorzystany jeden z algorytmów należących do tzw. **modeli zmiennych ukrytych** (*http://pl.wikipedia.org/wiki/Analiza_czynnikowa*). Przeznaczeniem tych algorytmów jest wyjaśnienie *obserwowanych interakcji* pomiędzy dużą liczbą użytkowników i produktów na podstawie relatywnie małej liczby *niezaobserwowanych, ukrytych przyczyn*. Odpowiada to opisaniu preferencji użytkowników i twórczości wykonawców za pomocą kilkudziesięciu gatunków muzyki (choć preferencje nie są określone ani bezpośrednio dostępne) i wyjaśnieniu na tej podstawie, dlaczego miliony użytkowników kupują określone albumy spośród tysięcy dostępnych.

Mówiąc dokładniej, w tym rozdziale wykorzystamy pewien rodzaj **faktoryzacji macierzy** (*http://en.wikipedia.org/wiki/Non-negative_matrix_factorization*). W algorytmach tego typu użytkownicy i wykonawcy są opisani za pomocą dużej macierzy A, która w wierszu i oraz kolumnie j zawiera określone dane, gdy użytkownik i słuchał utworu wykonawcy j. Macierz A jest niezapełniona — większość komórek zawiera zera, ponieważ dane są dostępne tylko dla nielicznych kombinacji użytkownik – wykonawca. Macierz A można zapisać jako iloczyn dwóch mniejszych macierzy, X i Y. Macierze te są bardzo wąskie — obie zawierają wprawdzie wiele wierszy, ponieważ macierz A ma wiele wierszy i kolumn, ale tylko kilka kolumn (k). Tych k kolumn odpowiada ukrytym zmiennym, które zostaną użyte do wyjaśnienia zależności między danymi.

Faktoryzacja macierzy będzie przybliżona, ponieważ liczba k jest mała, jak pokazuje rysunek 3.1.

Wspomniane algorytmy są niekiedy nazywane **algorytmami uzupełniania macierzy**, ponieważ oryginalna macierz A może być bardzo rzadko wypełniona, natomiast iloczyn XY^T daje macierz gęsto wypełnioną. Bardzo mało elementów, o ile w ogóle jakiekolwiek, zawiera wartości 0, dlatego iloczyn

Kup ksi k Pole ksi k



Rysunek 3.1. Faktoryzacja macierzy

ten jest jedynie przybliżeniem macierzy *A*. W tym modelu tworzone są („uzupełniane”) wartości również w tych elementach oryginalnej macierzy *A*, w których brakuje wartości (tj. zawierających wartość 0).

Na szczęście, w naszym przypadku algebra liniowa w elegancki i bezpośredni sposób podąża za intuicją. Dwie macierze mają po jednym wierszu zawierającym informacje odpowiednio o użytkownikach i wykonawcach. Wiersze zawierają niewiele, tj. *k* wartości. Każda wartość odpowiada ukrytej zmiennej w modelu danych. Zatem wiersze opisują, jak ściśle użytkownicy i wykonawcy są skojarzeni za pomocą ukrytych cech, co przekłada się na preferencje użytkowników i gatunki muzyki wykonawców. Pełne oszacowanie gęsto wypełnionej macierzy interakcji użytkownik – wykonawca jest zatem po prostu iloczynem macierzy użytkownik – cecha oraz cecha – wykonawca.

Jest jednak zła wiadomość: równanie $A = XY^T$ generalnie nie ma rozwiązania w ogóle, ponieważ macierze *X* i *Y* nie są na tyle duże (mówiąc językiem matematycznym, mają za mały *rząd*), aby dokładnie oddać macierz *A*. Jednak tak naprawdę jest to dobra wiadomość. Macierz *A* jest tylko małą próbką wszystkich możliwych interakcji. W pewien sposób zakładamy, że macierz ta jest niezwykle rzadko wypełniona, a więc jest trudnym w zrozumieniu obrazem prostszej rzeczywistości, którą można dobrze opisać za pomocą małej liczby (*k*) czynników. Wyobraź sobie puzzle z rysunkiem kota. Ostateczny obraz jest prosty do opisania: kot. Jeżeli jednak jest dostępnych tylko kilka elementów układanki, trudno opisać obraz.

Iloczyn XY^T powinien jak najbliżej odpowiadać macierzy *A*, jest on przecież niezbędny do dalszej analizy. Wynik iloczynu nie odzwierciedla dokładnie oryginalnej macierzy i nie powinien tego robić. Kolejną złą wiadomością jest brak możliwości znalezienia najlepszych macierzy *X* i *Y* jednocześnie. Dobra wiadomość jest taka, że trywialnym zadaniem jest znalezienie najlepszej macierzy *X*, jeżeli znana jest macierz *Y*, i odwrotnie. Jednak na początku nie jest znana żadna z macierzy!

Na szczęście, istnieją algorytmy umożliwiające wydostanie się z tego impasu i znalezienie satysfakcjonującego rozwiązania. Mówiąc dokładniej, w tym rozdziale do wyliczenia macierzy *X* i *Y* zostanie zastosowany algorytm **naprzemiennych najmniejszych kwadratów** (ang. ***Alternating Least Squares***, **ALS**). Metoda ta została spopularyzowana w publikacjach *Collaborative Filtering for Implicit Feedback Datasets* (*http://yifanhu.net/PUB/cf.pdf*) i *Large-scale Parallel Collaborative Filtering for the Netflix*

Kup książkę Poleć książkę

Prize (*http://www.grappa.univ-lille3.fr/~mary/cours/stats/centrale/reco/paper/MatrixFactorizationALS.pdf*), poświęconych konkursowi ogłoszonemu przez serwis Netflix (*http://en.wikipedia.org/wiki/Netflix_Prize*). W bibliotece MLlib systemu Spark algorytm ALS jest oparty na pomysłach wziętych z obu publikacji.

Macierz Y nie jest znana, ale można ją zainicjować wektorami wierszy zawierającymi wyłącznie losowe wartości. Następnie, wykorzystując prostą algebrę liniową, można dla danych macierzy A i Y znaleźć najlepszą macierz X. W rzeczywistości wyliczenie każdego wiersza i macierzy X jako funkcji macierzy Y i jednego wiersza tabeli A jest trywialnym zadaniem. Ponieważ obliczenia są od siebie niezależne, można je wykonywać równolegle, co doskonale usprawnia przetwarzanie danych na szeroką skalę:

$$A_i Y (Y^T Y)^{-1} = X_i$$

Osiągnięcie pełnej zgodności obu stron równania nie jest możliwe, więc rzeczywistym celem jest zminimalizowanie wartości wyrażenia $|A_i Y (Y^T Y)^{-1} = X_i|$ albo sumy kwadratów różnic między wartościami z obu tabel. Stąd w nazwie algorytmu znalazły się słowa „najmniejsze kwadraty”. W praktyce równania nie da się rozwiązać przez odwrócenie macierzy, jednak szybką i bardziej bezpośrednią metodą jest na przykład **dekompozycja QR** (*http://en.wikipedia.org/wiki/QR_decomposition*). Powyższe równanie opisuje teoretyczny sposób wyliczania wektora wierszy macierzy.

Tę samą operację można wykonać w celu wyliczenia każdego wiersza Y_j na podstawie macierzy X, następnie znów macierzy X na podstawie Y itd. Stąd w nazwie algorytmu pojawia się słowo „naprzemienne”. Jest tylko mały problem: macierz Y jest tworzona z losowych wartości! Macierz X zostanie wyliczona optymalnie, ale na podstawie fikcyjnych danych w macierzy Y. Na szczęście, gdy proces będzie powtarzany, macierze X i Y ustabilizują się i ostatecznie będą zawierały akceptowalne wartości.

Algorytm ALS nieco się komplikuje, gdy trzeba faktoryzować macierz zawierającą dane niejawne. Nie jest bezpośrednio faktoryzowana macierz A, ale macierz P, zawierająca wartości 0 i 1 w miejscach, gdzie macierz A zawiera odpowiednio wartości zerowe i dodatnie. Wartości z macierzy A są wykorzystywane później jako wagi. Ten temat wykracza poza zakres niniejszej książki, a jego poznanie nie jest niezbędne do zrozumienia stosowania opisywanego algorytmu.

Dodatkowo algorytm ALS wykorzystuje również nieliczność danych wejściowych. Dzięki zastosowaniu prostej, zoptymalizowanej algebry liniowej i jej natury równoległego przetwarzania danych algorytm działa bardzo sprawnie przy dużej ilości danych. Głównie z tego powodu stanowi on temat niniejszego rozdziału, a poza tym jest to jedyny algorytm rekomendacyjny obecnie zaimplementowany w bibliotece MLlib systemu Spark!

Przygotowanie danych

Skopiuj wszystkie trzy pliki danych do systemu HDFS. W tym rozdziale przyjęte jest założenie, że pliki będą zapisane w katalogu */user/ds/*. Otwórz system Spark poleceniem `spark-shell`. Pamiętaj, że przetwarzanie danych będzie wymagało wyjątkowo dużej ilości pamięci. Jeżeli przetwarzasz dane lokalnie, a nie w klastrze, prawdopodobnie będziesz musiał użyć parametru `--driver-memory 6g` w celu skonfigurowania odpowiednio dużej ilości pamięci niezbędnej do przetworzenia danych.

Kup ksi k Pole ksi k

Pierwszym krokiem w procesie budowania modelu jest poznanie dostępnych danych, a następnie rozłożenie ich lub przetransformowanie na format umożliwiający analizę w systemie Spark.

Jednym z małych ograniczeń implementacji algorytmu ALS w bibliotece MLlib jest konieczność określenia identyfikatorów użytkowników i elementów w postaci nieujemnych 32-bitowych liczb całkowitych. Oznacza to, że nie można używać identyfikatorów większych niż stała `Integer.MAX_VALUE`, czyli 2147483647. Czy dostępne dane spełniają ten warunek? Zamień plik na zbiór RDD danych typu `String` za pomocą metody `textFile` z kontekstu `SparkContext`:

```
val rawUserArtistData = sc.textFile("hdfs:///user/ds/user_artist_data.txt")
```

Domyślnie zbiór RDD zawiera jedną partycję w każdym bloku HDFS. Ponieważ analizowany plik zajmuje ok. 400 MB miejsca w systemie HDFS, zostanie podzielony na trzy do sześciu partycji, zależnie od wielkości bloku HDFS. Zazwyczaj jest to wystarczający podział, ale zadania uczenia maszynowego, takie jak algorytm ALS, wymagają wykonania operacji bardziej intensywnych obliczeniowo niż zwykłe przetwarzanie pliku. Lepszym rozwiązaniem może być podzielenie danych na mniejsze części, tj. na większą liczbę partycji. Dzięki temu system Spark mógłby wykorzystać większą liczbę rdzeni procesorów do równoległej pracy nad problemem. W powyższej metodzie można określić dodatkowy argument wyznaczający inną, większą liczbę partycji. Argument ten może na przykład być równy liczbie rdzeni procesorów w Twoim klastrze.

Każdy wiersz pliku zawiera identyfikator użytkownika, identyfikator wykonawcy i liczbę odtworzeń utworów. Dane rozdzielone są spacjami. W celu wyliczenia statystyk dla identyfikatorów użytkowników podzielimy wiersze zgodnie ze spacjami, a pierwszy element (indeksy elementów zaczynają się od 0) zamienimy na liczbę. Metoda `stats()` zwraca obiekt zawierający statystyki, takie jak wartości minimalna i maksymalna. Podobną operację wykonamy dla identyfikatorów wykonawców:

```
rawUserArtistData.map(_.split(' ')(0).toDouble).stats()
rawUserArtistData.map(_.split(' ')(1).toDouble).stats()
```

Wyliczone statystyki pokazują, że maksymalne wartości identyfikatorów użytkowników i wykonawców są równe odpowiednio 2443548 i 10794401. Są to wartości znacznie mniejsze od 2147483647. W przypadku tych identyfikatorów nie są wymagane żadne dodatkowe transformacje.

W tym przykładzie warto byłoby później poznać nazwiska wykonawców odpowiadające enigmatycznym identyfikatorom. Potrzebna informacja jest zawarta w pliku *artist_data.txt*. Tym razem identyfikator wykonawcy i jego nazwa rozdzielone są znakiem tabulacji. Jednakże zwykły podział wiersza na pary `(Int, String)` nie powiedzie się:

```
val rawArtistData = sc.textFile("hdfs:///user/ds/artist_data.txt")
val artistByID = rawArtistData.map { line =>
  val (id, name) = line.span(_ != '\t')
  (id.toInt, name.trim)
}
```

W tym przypadku metoda `span()` dzieli wiersz w miejscu pierwszego wystąpienia znaku tabulacji. Następnie pierwsza część jest konwertowana na liczbę, ponieważ zawiera identyfikator wykonawcy, a druga jest traktowana jako nazwa wykonawcy (po usunięciu białych znaków, czyli znaku tabulacji). Kilka wierszy w pliku jest uszkodzonych. Nie zawierają one znaku tabulacji albo zawierają znak nowego wiersza. Wiersze te powodują zgłoszenie wyjątku `NumberFormatException` i w rzeczywistości nie zawierają żadnych powiązań.

Kup ksi k Pole ksi k

Jednakże funkcja `map()` musi zwracać dokładnie jedną wartość dla każdej danej wejściowej, więc w tym przypadku nie można jej użyć. Za pomocą metody `filter()` można byłoby usunąć wadliwe wiersze, ale w ten sposób algorytm podziału wiesza byłby stosowany dwukrotnie. Funkcję `flatMap()` można stosować w przypadku, gdy każdy element jest powiązany z jedną wartością, z kilkoma lub nie jest powiązany z żadną, ponieważ funkcja ta po prostu „spłaszcza" kolekcje zawierające kilka lub niezawierające żadnych wyników i tworzy jeden wielki zbiór RDD. Funkcja ta działa poprawnie z kolekcjami języka Scala i z klasą `Option`. Klasa ta reprezentuje wartość, która może istnieć tylko opcjonalnie. Przypomina prostą kolekcję zawierającą tylko wartości 1 i 0, odpowiadające podklasom `Some` i `None`. Skoro zatem funkcja `flatMap` w poniższym kodzie może równie dobrze zwrócić pustą listę typu `List` lub listę zawierającą jeden element, uzasadnione będzie użycie zamiast tych list prostszych i bardziej czytelnych klas `Some` i `None`:

```
val artistByID = rawArtistData.flatMap { line =>
  val (id, name) = line.span(_ != '\t')
  if (name.isEmpty) {
    None
  } else {
    try {
      Some((id.toInt, name.trim))
    } catch {
      case e: NumberFormatException => None
    }
  }
}
```

Plik *artist_alias.txt* zawiera powiązane identyfikatory nazw wykonawców, które mogą być błędnie zapisane lub różnić się od nazwy skojarzonej z podstawowym identyfikatorem. Każdy wiersz zawiera dwa identyfikatory rozdzielone znakiem tabulacji. Plik ten jest dość mały, zawiera około 200 000 rekordów. Warto zamienić go na kolekcję `Map`, w której „złe" identyfikatory wykonawców są powiązane z „dobrymi", a nie na zbiór RDD, zawierający wyłącznie pary identyfikatorów. Podobnie jak poprzednio, z jakiegoś powodu w kilku wierszach brakuje pierwszego identyfikatora, więc wiersze te zostaną pominięte:

```
val rawArtistAlias = sc.textFile("hdfs:///user/ds/artist_alias.txt")
val artistAlias = rawArtistAlias.flatMap { line =>
  val tokens = line.split('\t')
  if (tokens(0).isEmpty) {
    None
  } else {
    Some((tokens(0).toInt, tokens(1).toInt))
  }
}.collectAsMap()
```

Na przykład w pierwszym rekordzie powiązane są identyfikatory 6803336 i 1000010. Można je wykorzystać do przeszukania zbioru RDD zawierającego nazwy wykonawców:

```
artistByID.lookup(6803336).head
artistByID.lookup(1000010).head
```

Ten rekord ewidentnie wiąże nazwy „Aerosmith (unplugged)" i „Aerosmith".

Kup książkę Poleć książkę

Utworzenie pierwszego modelu

Choć zbiór danych osiągnął niemal ostateczną formę umożliwiającą zastosowanie algorytmu ALS zaimplementowanego w bibliotece MLlib, wymaga wykonania jeszcze dwóch dodatkowych, niewielkich transformacji. Po pierwsze, trzeba wykorzystać zbiór danych z aliasami wykonawców do przekonwertowania wszystkich identyfikatorów dodatkowych (jeżeli istnieją) na identyfikatory podstawowe. Po drugie, dane muszą być przekształcone w obiekty typu `Rating`, implementujące rekordy zawierające dane użytkownik – produkt – wartość. Obiekt `Rating`, mimo swej nazwy, nadaje się do przetwarzania danych niejawnych. Zwróć również uwagę, że w interfejsie API biblioteki MLlib stosowane jest pojęcie „produkt", które w naszym przypadku oznacza wykonawcę. Wykorzystywany model w żaden sposób nie jest związany z rekomendowaniem produktów czy też — jak w tym przypadku — rekomendowaniem produktów dla użytkowników:

```
import org.apache.spark.mllib.recommendation._

val bArtistAlias = sc.broadcast(artistAlias)

val trainData = rawUserArtistData.map { line =>
  val Array(userID, artistID, count) = line.split(' ').map(_.toInt)
  val finalArtistID =
    bArtistAlias.value.getOrElse(artistID, artistID) ❶
  Rating(userID, finalArtistID, count)
}.cache()
```

❶ Przyjmij alias wykonawcy, jeżeli jest dostępny. W przeciwnym wypadku przyjmij jego oryginalny identyfikator.

Utworzony wcześniej obiekt `artistAlias` może być wykorzystywany bezpośrednio przez funkcję `map()`, mimo że jest to obiekt `Map` utworzony lokalnie na komputerze sterującym. Kod będzie działał poprawnie, ponieważ obiekt ten będzie automatycznie kopiowany przed wykonaniem każdego zadania. Jednak obiekt nie jest mały, zajmuje ok. 15 megabajtów pamięci, a w formie serializowanej co najmniej kilka megabajtów. Ponieważ na jednej maszynie JVM wykonywanych jest wiele zadań, przesyłanie i zapisywanie wielu kopii obiektu jest marnotrawstwem zasobów.

Zamiast tego utworzymy **zmienną rozgłaszaną** (ang. ***broadcast variable*** — *http://spark.apache.org/docs/latest/programming-guide.html#broadcast-variables*) o nazwie `bArtistAlias`, skojarzoną z obiektem `artistAlias`. Dzięki niej system Spark będzie wysyłał i umieszczał w pamięci każdego komputera w klastrze tylko jedną kopię obiektu. W przypadku wykonywania tysięcy zadań, wielu równolegle na każdym komputerze, można w ten sposób znacznie zmniejszyć ruch sieciowy i oszczędzić pamięć.

Wywołanie metody `cache()` stanowi wskazówkę dla systemu Spark, aby zbiór RDD po przetworzeniu został tymczasowo zapisany w pamięci klastra. Jest to przydatna opcja, ponieważ algorytm ALS jest iteracyjny, a dane są zazwyczaj odczytywane 10 lub więcej razy. Bez wykonania tej operacji zbiór RDD byłby wielokrotnie wyliczany na podstawie oryginalnych danych za każdym razem, kiedy byłby potrzebny! Zakładka *Storage* (zapisane dane) w interfejsie graficznym systemu Spark, pokazana na rysunku 3.2, pokazuje, jaka część zbioru RDD została zapisana i ile zajmuje pamięci. W tym przypadku zbiór zajmuje niemal 900 MB pamięci w całym klastrze.

Kup ksi k Pole ksi k

Zmienne rozgłaszane

Gdy system Spark wykonuje jakiś proces, tworzy binarne reprezentacje wszystkich informacji niezbędnych do wykonania zadania, czyli **domknięcia** (ang. ***closure***) funkcji, które mają być wykonane. Domknięcie zawiera wszystkie struktury danych z procesu sterującego, do których odwołuje się dana funkcja. Spark wysyła domknięcie do wszystkich komputerów w klastrze.

Zmienne rozgłaszane przydają się w sytuacjach, gdy wiele zadań musi mieć dostęp do tych samych (stałych) struktur danych. Zmienne te rozszerzają zwykłą obsługę domknięć przez zadania o następujące funkcjonalności:

- Zapisywanie danych w postaci podstawowych obiektów Java w pamięci każdego komputera, dzięki czemu dane nie muszą być deserializowane przez każde zadanie.
- Zapisywanie danych w pamięci, do wykorzystania przez różne zadania i procesy.

Rozważmy dla przykładu aplikację do analizy języka mówionego, wykorzystującą duży słownik języka polskiego. Rozgłoszenie tego słownika umożliwia przesłanie go tylko raz do wszystkich komputerów:

```
val dict = ...
val bDict = sc.broadcast(dict)
...
def query(path: String) = {
  sc.textFile(path).map(l => score(l, bDict.value))
  ...
}
```

Storage Level	Cached Partitions	Fraction Cached	Size in Memory
Memory Deserialized 1x Replicated	120	100%	886.8 MB

Rysunek 3.2. Zakładka Storage w graficznym interfejsie systemu Spark, zawierająca informacje o pamięci zajętej przez zbiory RDD

Teraz możemy zbudować model:

```
val model = ALS.trainImplicit(trainData, 10, 5, 0.01, 1.0)
```

Powyższy kod tworzy zmienną `model` typu `MatrixFactorizationModel`. Operacja ta prawdopodobnie będzie wykonywana przez kilka minut lub dłużej, w zależności od zastosowanego klastra. W porównaniu do kilku innych modeli uczenia maszynowego, które w ostatecznej formie składają się z zaledwie kilku parametrów czy współczynników, ten model jest ogromny. Zawiera wektor 10 wartości dla każdego użytkownika i produktu w modelu, których w naszym przypadku jest ponad 1,7 miliona. Model zawiera macierze użytkownik – cecha i produkt – cecha jako osobne zbiory RDD.

Aby zobaczyć niektóre wektory cech, uruchom podany niżej kod. Zwróć uwagę, że wektor cech jest tabelą typu `Array` złożoną z 10 elementów, a tabele nie są standardowo wyświetlane w czytelnej postaci. W poniższym kodzie wektor jest zamieniany na czytelną postać za pomocą metody `mkString()`, powszechnie stosowanej w języku Scala do łączenia elementów kolekcji w rozdzielony separatorami ciąg znaków:

```
model.userFeatures.mapValues(_.mkString(", ")).first()
...
(4293,-0.3233030601963864, 0.31964527593541325,
```


Kup ksi k Pole ksi k

```
0.49025505511361034, 0.09000932568001832, 0.4429537767744912,
0.4186675713407441, 0.8026858843673894, -0.4841300444834003,
-0.12485901532338621, 0.19795451025931002)
```

W Twoim przypadku wynikowe wartości będą inne. Ostateczny model zależy od losowo wygenerowanych wektorów cech.

Pozostałe argumenty metody `trainImplicit()` są *hiperparametrami*, których wartości mają wpływ na jakość rekomendacji przedstawianych przez model. Hiperparametry zostaną opisane później. Pierwsze ważne pytanie brzmi: czy ten model jest dobry? Czy przygotowuje dobre rekomendacje?

Wyrywkowe sprawdzanie rekomendacji

Najpierw musimy sprawdzić, badając dane użytkownika i wybierane przez niego utwory, czy przygotowywane dla niego rekomendacje wykonawców są zgodnie z naszą intuicją. Weźmy dla przykładu użytkownika o identyfikatorze 2093760. Wyszukajmy identyfikatory wykonawców, których utworów słuchał ten użytkownik, i wyświetlmy odpowiadające im nazwy. Cała operacja będzie polegała na wyszukiwaniu identyfikatorów wykonawców na podstawie wejściowego identyfikatora użytkownika, następnie filtrowaniu danych według identyfikatorów wykonawców i na koniec zebraniu i wyświetleniu posortowanych nazw:

```
val rawArtistsForUser = rawUserArtistData.map(_.split(' ')).
  filter { case Array(user,_,_) => user.toInt == 2093760 } ❶

val existingProducts =
  rawArtistsForUser.map { case Array(_,artist,_) => artist.toInt }.
  collect().toSet ❷

artistByID.filter { case (id, name) =>
  existingProducts.contains(id)
}.values.collect().foreach(println) ❸

...
David Gray
Blackalicious
Jurassic 5
The Saw Doctors
Xzibit
```

❶ Wyszukanie wierszy danych z identyfikatorem użytkownika 2093760.

❷ Zebranie unikatowych identyfikatorów wykonawców.

❸ Odfiltrowanie danych o wykonawcach, wyszukanie oraz wyświetlenie ich nazw.

Utwory wyszukanych wykonawców tworzą mieszankę gatunków mainstream pop i hip-hop. Jesteś fanem zespołu Jurassic 5? Pamiętaj, że dane pochodzą z roku 2005. Jeżeli Cię to interesuje, grupa Saw Doctors jest bardzo znanym zespołem rockowym z Irlandii.

Teraz możemy wykonać coś w rodzaju zarekomendowania pięciu wykonawców wybranemu użytkownikowi:

Kup ksi k   Pole ksi k

```
val recommendations = model.recommendProducts(2093760, 5)
recommendations.foreach(println)

...
Rating(2093760,1300642,0.02833118412903932)
Rating(2093760,2814,0.027832682960168387)
Rating(2093760,1037970,0.02726611004625264)
Rating(2093760,1001819,0.02716011293509426)
Rating(2093760,4605,0.027118271894797333)
```

Wynik składa się z obiektów `Rating` zawierających (taki sam) identyfikator użytkownika, identyfikator wykonawcy i liczbę. Liczba ta jest wartością pola `rating`, ale nie oznacza szacowanej oceny. W tego typu algorytmie ALS jest to enigmatyczna wartość z przedziału od 0 do 1. Im większa jest ta liczba, tym lepsza rekomendacja. Nie jest to współczynnik prawdopodobieństwa, ale w zależności od tego, jak bliska jest ta wartość liczbie 1 lub 0, można ocenić, czy dany użytkownik zainteresuje się danym wykonawcą, czy nie.

Po wybraniu z rekomendacji identyfikatorów wykonawców możemy w podobny sposób wyszukać ich nazwy:

```
val recommendedProductIDs = recommendations.map(_.product).toSet

artistByID.filter { case (id, name) =>
  recommendedProductIDs.contains(id)
}.values.collect().foreach(println)

...
Green Day
Linkin Park
Metallica
My Chemical Romance
System of a Down
```

Wynik jest mieszanką gatunków pop punk i metal. Na pierwszy rzut oka rekomendacje te nie wyglądają zachęcająco. Są to wprawdzie popularni wykonawcy, jednak raczej nie odpowiadają preferencjom wybranego użytkownika.

Ocena jakości rekomendacji

Oczywiście, opisana wyżej ocena wyników jest subiektywna. Trudno innej osobie ocenić, jak trafione są przygotowane rekomendacje. Co więcej, niemożliwe jest dokonanie przez człowieka oceny nawet niewielkiej próbki wyników.

Rozsądne zatem jest przyjęcie założenia, że użytkownicy wybierają utwory popularnych wykonawców, a unikają utworów wykonawców niepopularnych. Zatem utwory wybierane przez użytkownika dają częściową ocenę, jak bardzo „dobra” lub „zła” jest rekomendacja danego wykonawcy. Jest to problematyczne założenie, ale chyba najlepsze, jakie można przyjąć, nie dysponując innymi danymi. Na przykład, użytkownik o identyfikatorze 2093760 może lubić wielu innych wykonawców niż pięciu wyżej wymienionych, a spośród 1,7 miliona pozostałych wykonawców, których utworów dany użytkownik nie zna, kilku może być dla niego interesujących. Nie wszystkie rekomendacje są zatem „złe”.

Kup ksi k Pole ksi k

A gdyby tak oceniać algorytm rekomendacyjny według jego możliwości umieszczania dobrych wykonawców na wysokich pozycjach listy rekomendacji? Jest to jeden z podstawowych parametrów, który można zastosować w systemach oceniających, takich jak systemy rekomendacyjne. Problem jednak polega na tym, że „dobry" wykonawca jest zdefiniowany jako „ten, którego utworów dany użytkownik słuchał", a system rekomendacyjny już otrzymał te informacje. Może on zatem po prostu zwrócić listę wykonawców, których użytkownik już słuchał, i rekomendacje te będą idealne. Nie będą to jednak przydatne wyniki, przede wszystkim dlatego, że zadaniem systemu rekomendacyjnego jest wyszukiwanie wykonawców, których użytkownik jeszcze nie słuchał.

Aby wyniki były użyteczne, niektóre dane o wybieranych wykonawcach muszą zostać wyodrębnione i ukryte przed algorytmem ALS tworzącym model. Później wyodrębnione dane można potraktować jako zbiór trafionych rekomendacji dla każdego użytkownika, ale nie mogą one być wcześniej przekazane systemowi rekomendacyjnemu. System rekomendacyjny powinien ocenić wszystkich wykonawców w modelu, po czym oceny te należy porównać z ocenami wyodrębnionych wykonawców. W idealnym przypadku system powinien wybrać wykonawców znajdujących się na początku listy.

Następnie można ocenić system rekomendacyjny, porównując rankingi wszystkich wyodrębnionych wykonawców z pozostałymi. (W praktyce polega to na sprawdzaniu jedynie próbki takich par wartości, ponieważ potencjalnie może ich być bardzo dużo). Odsetek par, w których wyodrębnieni wykonawcy zajmują wysokie pozycje, stanowi ocenę rekomendacji. Wartość 1,0 oznacza najlepszą, a 0,0 najgorszą ocenę. Wartość 0,5 jest oczekiwaną średnią wartością ocen losowo wybranych wykonawców.

Opisana metryka jest bezpośrednio związana z koncepcją pozyskiwania informacji, zwaną **krzywą ROC** (ang. ***Receiver Operating Characteristic***, charakterystyka operacyjna odbiornika — *http://en.wikipedia.org/wiki/Receiver_operating_characteristic*). Metryka opisana w poprzednim akapicie odpowiada polu poniżej krzywej ROC. Jest to tzw. **pole AUC** (ang. ***Area Under the Curve***, obszar pod krzywą), który można interpretować jako prawdopodobieństwo, że losowo wybrana dobra rekomendacja będzie znajdowała się na wyższej pozycji niż losowo wybrana zła rekomendacja.

Metryka AUC jest również wykorzystywana do oceny systemów klasyfikujących. Jest ona zaimplementowana w bibliotece MLlib w klasie `BinaryClassificationMetrics`, zawierającej odpowiednie metody. Na potrzeby systemu rekomendacyjnego wyliczymy wartość AUC *dla każdego użytkownika* i uśrednimy wyniki. Wynikowa wartość będzie nieco inna. Nazwijmy ją „średnią wartością AUC".

Inne metryki, właściwe dla systemu pozycjonującego wartości, są zaimplementowane w klasie `RankingMetrics`. Są to: precyzja (ang. *precision*), powtórzenie (ang. *recall*) i średnia precyzja (ang. *mean average precision*, MAP). Metryka MAP jest często stosowana i lepiej opisuje jakość najlepszych rekomendacji. Jednak w naszym przypadku będziemy stosować metrykę AUC, ponieważ stanowi ona ogólną ocenę jakości wyników całego modelu.

W rzeczywistości proces wyodrębniania pewnych danych w celu wybrania odpowiedniego modelu i oceny jego dokładności jest powszechnie stosowaną praktyką we wszystkich algorytmach uczenia maszynowego. Zazwyczaj dane są dzielone na trzy podzbiory: ćwiczebny, sprawdzający krzyżowy (ang. *cross-validation*, CV) i testowy. Dla uproszczenia, w naszym przykładzie zostaną wykorzystane tylko dwa podzbiory: ćwiczebny i sprawdzający. Wystarczą one do wybrania modelu. W rozdziale 5. metoda ta zostanie rozszerzona o wykorzystanie podzbioru testowego.

Kup ksi k Pole ksi k

Obliczenie metryki AUC

Implementacja funkcji obliczającej metrykę AUC znajduje się w kodzie źródłowym dołączonym do książki. Kod jest skomplikowany i nie prezentujemy go tutaj, jednak niektóre jego szczegóły są opisane w komentarzach. Argumentami funkcji jest podzbiór sprawdzający „dobrych”, czyli „trafionych” wykonawców dla każdego użytkownika oraz funkcja prognozująca. Funkcja ta przekłada każdą parę wartości użytkownik – wykonawca na obiekt typu `Rating` zawierający informacje o użytkowniku, wykonawcy i prognozowaną ocenę, której wysoka wartość oznacza wyższą pozycję w rekomendowanej liście wykonawców.

Aby użyć tej funkcji, należy dane wejściowe podzielić na podzbiory ćwiczebny i sprawdzający. Model ALS będzie tworzony tylko na podstawie podzbioru ćwiczebnego, natomiast podzbiór sprawdzający zostanie użyty do oceny modelu. W tym przypadku 90% danych będzie stanowiło podzbiór ćwiczebny, a pozostałe 10% podzbiór sprawdzający:

```
import org.apache.spark.rdd._

def areaUnderCurve(
  positiveData: RDD[Rating],
  bAllItemIDs: Broadcast[Array[Int]],
  predictFunction: (RDD[(Int,Int)] => RDD[Rating])) = {
  ...
}

val allData = buildRatings(rawUserArtistData, bArtistAlias) ❶
val Array(trainData, cvData) = allData.randomSplit(Array(0.9, 0.1))
trainData.cache()
cvData.cache()

val allItemIDs = allData.map(_.product).distinct().collect() ❷
val bAllItemIDs = sc.broadcast(allItemIDs)

val model = ALS.trainImplicit(trainData, 10, 5, 0.01, 1.0)
val auc = areaUnderCurve(cvData, bAllItemIDs, model.predict)
```

❶ Funkcja zdefiniowana w załączonym kodzie źródłowym.

❷ Usunięcie duplikatów danych i zebranie ich na komputerze sterującym.

Zwróć uwagę, że jednym z argumentów funkcji `areaUnderCurve()` jest inna funkcja. W tym przypadku jest nią metoda `predict()` klasy `MatrixFactorizationModel`, ale za chwilę zostanie zamieniona na inną, alternatywną funkcję.

Zwrócony wynik jest równy ok. 0,96. Czy to dobry rezultat? Oczywiście tak — jest to liczba większa niż 0,5, czyli oczekiwana średnia wartości losowych rekomendacji. Wynik jest bliski liczbie 1,0, czyli maksymalnej ocenie. Ogólnie przyjmuje się, że wartość AUC większa niż 0,9 jest wysoką oceną.

Opisana ocena może być ponownie wyliczona z innym ćwiczebnym podzbiorem 90% danych. Wynikowa średnia wartość AUC może być lepszym oszacowaniem skuteczności algorytmu w przypadku użytych danych. W rzeczywistości powszechnie stosowaną praktyką jest dzielenie danych na *k* podzbiorów podobnej wielkości, wybranie *k* – 1 podzbiorów do utworzenia jednego podzbioru ćwiczebnego i weryfikacja wyniku na podstawie pozostałego podzbioru. Proces ten powtarza się *k* razy, każdorazowo wybierając inne podzbiory. Metoda ta jest nazywana ***k*-krotnym**

Kup ksi k Pole ksi k

sprawdzianem krzyżowym (*http//pl.wikipedia.org/wiki/Sprawdzian_krzyżowy*). Dla uproszczenia nie jest ona zaimplementowana w prezentowanych przykładach, jednak niektóre jej elementy są dostępne w bibliotece MLlib w funkcji pomocniczej `MLUtils.kFold()`.

Przydatne jest ocenienie skuteczności opisanego algorytmu z zastosowaniem prostszego podejścia. Rozważmy dla przykładu rekomendację wykonawców najczęściej słuchanych przez wszystkich użytkowników w ogóle. Nie jest to spersonalizowane podejście, ale jest proste i prawdopodobnie skuteczne. Zdefiniujmy prostą funkcję prognozującą i wyliczmy wartość AUC:

```
def predictMostListened(
  sc: SparkContext,
  train: RDD[Rating])(allData: RDD[(Int,Int)]) = {

val bListenCount = sc.broadcast(
  train.map(r => (r.product, r.rating)).
    reduceByKey(_ + _).collectAsMap()
  )
  allData.map { case (user, product) =>
    Rating(
      user,
      product,
      bListenCount.value.getOrElse(product, 0.0)
    )
  }
}

val auc = areaUnderCurve(
  cvData, bAllItemIDs, predictMostListened(sc, trainData))
```

Powyższy kod jest kolejnym interesującym przykładem składni języka Scala, gdzie zdefiniowana jest funkcja przyjmująca dwie listy argumentów. Wywołanie tej funkcji i przekazanie jej pierwszych dwóch argumentów powoduje utworzenie *częściowo stosowanej funkcji*, która sama przyjmuje argument (`allData`) w celu zwrócenia prognozowanych wyników. Wynikiem funkcji `predictMostListened`
`↪(sc,trainData)` jest *inna funkcja*.

W tym przypadku wynik jest równy 0,93. Oznacza to, że zgodnie z przyjętą metryką niespersonalizowane rekomendacje są bardzo trafione. Miło jest widzieć, że zbudowany model jest lepszy niż użyty w tej prostej metodzie. Czy może być jeszcze lepszy?

Dobór wartości hiperparametrów

Do tej pory wartości hiperparametrów wykorzystanych do zbudowania modelu opartego na klasie `MatrixFactorizationModel` zostały dobrane bez żadnych wyjaśnień. Nie są one wyliczane w ramach algorytmu i muszą być podane w kodzie wywołującym klasę. Argumenty metody `ALS.trainImplicit()` były następujące:

`rank` = 10
: Liczba ukrytych zmiennych modelu, czyli liczba *k* oznaczająca liczbę kolumn w macierzach użytkownik – cecha i produkt – cecha. W nietrywialnych przypadkach jest to również rząd tych macierzy.

Kup ksi k Pole ksi k

`iterations` = 5
: Liczba iteracji wykonywanych podczas faktoryzacji macierzy. Wykonanie dużej liczby iteracji trwa dłużej, ale skutkuje lepszą faktoryzacją.

`lambda` = 0,01
: Parametr standardowego nadmiernego dopasowania. Duża wartość ogranicza dopasowanie, jednak może pogorszyć dokładność faktoryzacji.

`alpha` = 1,0
: Argument wpływający na względną wagę obserwowanych i nieobserwowanych interakcji użytkownik – produkt podczas faktoryzacji.

Argumenty `rank`, `lambda` i `alpha` można uznać za *hiperparametry* modelu. (Argument `iterations` stanowi raczej ograniczenie ilości zasobów zajmowanych podczas faktoryzacji). Nie są to wartości, które są umieszczane w macierzach w klasie `MatrixFactorizationModel` — są to po prostu jej *parametry*, określane w algorytmie. Te hiperparametry są stosowane zamiast parametrów procesu tworzącego model.

Wartości użyte w powyższym opisie niekoniecznie są optymalne. Dobór odpowiednich wartości hiperparametrów jest najczęściej spotykanym problemem w algorytmach uczenia maszynowego. Najbardziej podstawowym sposobem doboru wartości jest testowanie różnych ich kombinacji i ocena metryki uzyskiwanej dla każdej z nich, a na koniec wybranie kombinacji, która daje w wyniku najlepszą metrykę.

W poniższym przykładzie sprawdzanych jest osiem możliwych kombinacji dla następujących wartości argumentów: `rank` = 10 i 50, `lambda` = 1,0 i 0,0001 oraz `alpha` = 1,0 i 40,0. Wartości te również zostały przyjęte odgórnie, ale dobrane tak, aby obejmowały szeroki zakres wartości parametrów. Wyniki obliczeń zostały wyświetlone według malejącej wartości AUC:

```
val evaluations =
  for (rank <- Array(10, 50);
       lambda <- Array(1.0, 0.0001);
       alpha <- Array(1.0, 40.0)) ❶
    yield {
      val model = ALS.trainImplicit(trainData, rank, 10, lambda, alpha)
      val auc = areaUnderCurve(cvData, bAllItemIDs, model.predict)
       ((rank, lambda, alpha), auc)
    }

evaluations.sortBy(_._2).reverse.foreach(println) ❷

...
((50,1.0,40.0),0.9776687571356233)
((50,1.0E-4,40.0),0.9767551668703566)
((10,1.0E-4,40.0),0.9761931539712336)
((10,1.0,40.0),0.976154587705189)
((10,1.0,1.0),0.9683921981896727)
((50,1.0,1.0),0.9670901331816745)
((10,1.0E-4,1.0),0.9637196892517722)
((50,1.0E-4,1.0),0.9543377999707536)
```

❶ Potrójnie zagnieżdżona pętla.

❷ Sortowanie wyników malejąco według drugiej wartości (AUC) i ich wyświetlenie.

Kup ksi k Pole ksi k

Przedstawiona tu składnia jest przykładem tworzenia zagnieżdżonych pętli w języku Scala. Jest to pętla oparta na zmiennej `alpha`, zagnieżdżona w pętli opartej na zmiennej `lambda`, zagnieżdżonej w pętli opartej na zmiennej `rank`.

Interesujące jest to, że wartość 40 parametru `alpha` wydaje się lepsza niż 1. (Co ciekawe, wartość 40 została zaproponowana jako domyślna w jednej z wspomnianych wcześniej publikacji poświęconych algorytmowi ALS). Efekt ten można traktować jako wskazówkę, że model jest skuteczniejszy, gdy bardziej koncentruje się na utworach, które użytkownik już słyszał, niż na tych, których jeszcze nie zna.

Nieco lepiej wyglądają również wyższe wartości argumentu `lambda`. Wydaje się, że model jest dość wrażliwy na nadmierne dopasowanie, więc potrzebuje wyższych wartości tego argumentu w celu ograniczenia zbyt dokładnego dopasowania na podstawie nielicznych danych dla każdego użytkownika. Nadmierne dopasowanie zostanie omówione dokładniej w rozdziale 4.

Liczba cech raczej nie robi większej różnicy. Wartość 50 pojawia się w kombinacjach argumentów dających zarówno wysokie, jak i niskie oceny, aczkolwiek bezwzględne wartości tych ocen nie zmieniają się znacząco. Może to oznaczać, że właściwa liczba cech jest w rzeczywistości większa niż 50 i użyte tu wartości są raczej zbyt małe.

Oczywiście, opisany proces można powtarzać dla innych zakresów i większej liczby wartości. Jest to sposób doboru wartości hiperparametrów metodą brutalnej siły. Jednak w praktyce, gdy klastry obejmujące terabajty pamięci i setki rdzeni procesorów nie są rzadkością, a platformy takie jak Spark mogą wykonywać zadania równolegle i wykorzystywać dostępną pamięć, sposób ten staje się całkiem realny do zastosowania.

Zrozumienie znaczenia hiperparametrów nie jest bezwzględnie wymagane, aczkolwiek pomocna jest znajomość ich normalnych zakresów wartości, aby przeszukiwana przestrzeń parametrów nie była zbyt duża ani zbyt mała.

Przygotowanie rekomendacji

Jakie rekomendacje przygotuje nowy model dla użytkownika 2093760, gdy użyjemy najlepszej kombinacji hiperparametrów?

```
50 Cent
Eminem
Green Day
U2
[unknown]
```

Podobnie rekomendacja dwóch wykonawców gatunku hip-hop ma jakiś sens. Wynik `[unknown]` (nieznany) oczywiście nie oznacza wykonawcy. Po sprawdzeniu oryginalnych danych okazuje się, że wartość ta pojawia się 429 447 razy, co plasuje ją niemal w pierwszej setce wykonawców. Jest to domyślna wartość przyjmowana w przypadku, gdy nie ma danych o wykonawcy odsłuchanego utworu. Jest ona prawdopodobnie wysyłana przez jakąś aplikację serwisu. To nie jest przydatna informacja i należy ją usunąć z danych przed kolejną analizą. Jest to przykład, jak iteracyjna jest analiza danych w praktyce, gdy na każdym etapie odkrywane są nowe cechy danych.

Kup ksi k Pole ksi k

Opisany model może być użyty do przygotowywania rekomendacji dla wszystkich użytkowników. W zależności od ilości danych i wydajności klastra można utworzyć proces, który będzie przeliczał model i przygotowywał rekomendacje co godzinę lub częściej.

Jednak obecnie implementacja algorytmu ALS w bibliotece MLlib systemu Spark nie umożliwia przygotowywania rekomendacji dla wszystkich użytkowników. Można to robić dla każdego użytkownika z osobna, ale wykonanie krótkotrwałego rozproszonego zadania trwa kilka sekund. Sposób ten może być odpowiedni w przypadku, gdy trzeba szybko przygotowywać rekomendacje dla niewielkiej grupy użytkowników. Poniższy kod przygotowuje i wyświetla rekomendacje dla 100 wybranych użytkowników:

```
val someUsers = allData.map(_.user).distinct().take(100) ❶
val someRecommendations =
  someUsers.map(userID => model.recommendProducts(userID, 5)) ❷
someRecommendations.map(
  recs => recs.head.user + " -> " + recs.map(_.product).mkString(", ") ❸
).foreach(println)
```

❶ Skopiowanie danych 100 (różnych) użytkowników na komputer sterujący.

❷ Metoda `map()` jest tu wykonywana lokalnie.

❸ Metoda `mkString()` tworzy ciąg znaków złożony z elementów kolekcji rozdzielonych separatorami.

W tym przypadku rekomendacje są po prostu wyświetlane. Równie łatwo mogą być zapisane w zewnętrznym systemie, na przykład bazie HBase, umożliwiającej szybkie przeszukiwanie danych.

Co ciekawe, opisany proces można wykorzystać do rekomendowania użytkowników wykonawcom. Można w ten sposób uzyskać odpowiedź na pytanie: „Którzy ze 100 użytkowników mogą być najbardziej zainteresowani nowym albumem wykonawcy X?”. W tym celu podczas przygotowywania danych wejściowych należy jedynie zamienić miejscami pola z identyfikatorami użytkowników i wykonawców:

```
rawUserArtistData.map { line =>
  ...
  val userID = tokens(1).toInt ❶
  val artistID = tokens(0).toInt ❷
  ...
}
```

❶ Odczytanie identyfikatora wykonawcy jako „użytkownika”.

❷ Odczytanie identyfikatora użytkownika jako „wykonawcy”.

Dalsze kroki

Oczywiście, można spędzić jeszcze wiele czasu na regulacji parametrów modelu oraz wyszukiwaniu i poprawianiu anomalii w danych wejściowych, takich jak wykonawca `[unknown]`.

Na przykład po krótkiej analizie liczby odtworzeń utworów okazuje się, że użytkownik 2064012 słuchał wykonawcy 4468 niesamowitą liczbę 439 771 razy! Wykonawca ten to nieszczególnie popularny zespół „System of a Down”, grający muzykę z gatunku metalu alternatywnego. Wykonawca ten pojawiał się we wcześniejszych rekomendacjach. Przy założeniu, że utwór trwa średnio 4 minuty,

Kup ksi k
Pole ksi k

oznacza to, że hity takie jak „Chop Suey!” czy „B.Y.O.B.” były w sumie odtwarzane przez ponad 33 lata! Ponieważ zespół ten zaczął nagrywać płyty w roku 1998, w celu osiągnięcia powyższego wyniku cztery lub pięć utworów musiałoby być jednocześnie odtwarzanych przez 7 lat. To widocznie jest spam albo błąd w danych. Jest to kolejny przykład problemów, jakie trzeba rozwiązywać w praktyce.

Algorytm ALS nie jest jedynym algorytmem rekomendacyjnym. Obecnie jest to jedyny algorytm zaimplementowany w bibliotece MLlib systemu Spark. Jednak biblioteka ta obsługuje odmianę tego algorytmu, wykorzystującą jawne dane. Korzysta się z niej w bardzo podobny sposób, z tym wyjątkiem, że model tworzy się za pomocą metody `ALS.train()`. Algorytm ten jest odpowiedni dla danych opartych na rankingach, a nie na liczbach odsłuchań. Takimi danymi są na przykład oceny w skali od 1 do 5 wykonawców wystawiane przez użytkowników. Wynikowe pole `rating` obiektu `Rating` zwracane przez różne metody rekomendacyjne zawiera szacowaną ocenę.

W miarę upływu czasu w bibliotece MLlib lub innych będą pojawiały się również inne algorytmy rekomendacyjne.

W praktyce systemy rekomendacyjne często muszą przygotowywać dane w czasie rzeczywistym, ponieważ są stosowane w systemach sprzedaży internetowej, gdzie trzeba przygotowywać rekomendacje w czasie, gdy użytkownik przegląda strony z produktami. Jak wspomniałem wcześniej, wstępne przygotowanie i zapisanie rekomendacji w systemie NoSQL jest dobrym sposobem przygotowywania rekomendacji na szeroką skalę. Jednym z mankamentów tej metody jest konieczność wstępnego przygotowania rekomendacji dla wszystkich użytkowników, którzy będą ich wkrótce potrzebować, a może to być potencjalnie każdy użytkownik. Na przykład jeżeli stronę odwiedza dziennie tylko 10 000 spośród miliona użytkowników, wstępne przygotowywanie rekomendacji dla miliona użytkowników każdego dnia stanowi w 99% stracony czas.

Lepiej byłoby przygotowywać rekomendacje na bieżąco, w miarę potrzeb. W tym przykładzie do przygotowania rekomendacji dla jednego użytkownika wykorzystywaliśmy klasę `MatrixFactori↪zationModel` i była to operacja wykonywana w sposób rozproszony. Trwała ona kilka sekund, ponieważ powyższa klasa jest wyjątkowo duża i w rzeczywistości stanowi rozproszony zbiór danych. Tak nie jest w przypadku innych modeli, które mają znacznie lepsze osiągi. W projektach takich jak Oryx 2 podejmowane są próby implementacji algorytmów przygotowujących rekomendacje na żądanie w czasie rzeczywistym z wykorzystaniem bibliotek takich jak MLlib, poprzez implementację efektywnego przetwarzania danych zapisanych w pamięci.

Kup ksi k Pole ksi k

Skorowidz

A

agregowanie danych, 36
akcja
  collect, 30
  count, 30
  countByValue, 39
  saveAsTextFile, 30
  textFile, 30
akumulatory, 225
algorytm
  k-średnie++, 92
  LSA, 113
  naprzemiennych najmniejszych kwadratów, 50
  rekomendacyjny, 49
algorytmy
  uzupełniania macierzy, 49
  w bibliotece MLlib, 229
alokacja ukrytej zmiennej Dirichleta, 119
analiza
  danych, 21
  danych genomicznych, 187
  danych geoprzestrzennych, 158
  danych neuroobrazowych, 205
  dokumentów XML, 125
  głównych składowych, 93
  głównych znaczników, 126
  przefiltrowanego grafu, 138
  semantyczna, 101
  sieci, 122
  sieci współwystępowań, 121
  składni, 104
  tras taksówek, 147
  wielkich zbiorów, 13
anomalie, 86
  w ruchu sieciowym, 85
Apache Avro, 188
Apache Spark, 16
API Esri Geometry, 151
API Pipelines, 231
aplikacja Spark, 223
architektura pakietu PySpark, 208
atrybut
  krawędzi, 129
  wierzchołka, 129
Audioscrobbler, 47
Avro, 228

B

badacze danych, 22
baza
  ENCODE, 197
  MEDLINE, 122
biblioteka
  GraphX, 121, 128
  JodaTime, 149
  matplotlib, 206
  MLlib, 18, 108, 229, 231
  nltk, 206
  NScalaTime, 149
  pakietu Thunder, 209
  pandas, 206
  statsmodels, 206
  Scala, 125
błędne rekordy danych, 155
BSP, bulk-synchronous parallel, 141
budowa pakietu PySpark, 207
budowanie sesji, 162

C

cecha, feature, 66, 152
cechy kategorialne, 96
centroid, 87
CVaR, conditional value at risk, 168
czynnik
  rynkowy, 168, 170
  transkrypcyjny, 197

D

dane
  1000 Genomes, 203
  Audioscrobbler, 47
  Covtype, 70
  dotyczące kursów taksówek, 154
  ENCODE, 197
  genomiczne, 190
  GeoJSON, 152
  geoprzestrzenne, 148, 158
  KDD Cup, 87
  RDD, 37, 39
  temporalne, 148
DBSCAN, 100
dekompozycja QR, 51
dobór wartości k, 90
domknięcie, closure, 55
domniemanie typu, 27
drzewo decyzyjne, 65, 68, 72–78

Kup ksi k   Pole ksi k

E

entropia, 77, 97
Esri Geometry API, 150
etap zadania, 224
etykieta, 97

F

faktoryzacja macierzy, 50
filtrowanie, 33
  kolaboratywne, 49
  krawędzi, 135
format
  GeoJSON, 152
  kolumnowy, 195
  Parquet, 195
formaty plików, 228
fragment, 28
funkcja
  gęstości prawdopodobieństwa, PDF, 169
  isHeader, 32
  parse, 155

G

geometria, geometry, 152
graf, 230
  współwystępowań, 129
grupowanie, 88, 98
  według k-średnich, 85, 86

H

hiperparametry, 60
  drzewa decyzyjnego, 76
histogram, 38
HPC, high-performance computing, 14

I

indeks giełdowy, 168
  NASDAQ, 170
informacje
  o genotypach, 203
  o pakiecie PySpark, 206
  zwrotne, 48
instalacja biblioteki pakietu Thunder, 209
instrument, 168
interaktywna analiza, 154
interfejs
  API Pipelines, 232
  MLlib Pipelines API, 231
interpreter REPL, 26, 29
IPython Notebook, 208
iteracje, 15

J

jądrowe szacowanie gęstości, 176
język
  definicji interfejsów, 188
  Python, 206
  Scala, 30

K

KDD Cup, 87
k-krotne sprawdzanie krzyżowe, 60
klasa
  Graph, 131
  RDD, 43
  Source, 159
klastrowanie, 139
klasy wyboru, 33, 36
klasyfikowanie
  neuronów, 216
  tekstu, 233
klik, 139
kodowanie 1 z n, 71
kolekcje, 38
kompilacja kodu, 29
konfiguracja pakietu PySpark, 208
kontekst SparkContext, 24
krawędzie zakłócające dane, 135

L

lasy decyzyjne, 68
LDA, Latent Dirichlet Allocation, 119
lematyzacja, 105
lista danych, 33
losowa projekcja 3D, 95
losowanie prób, 176
losowy las decyzyjny, 66, 81
LSA, Latent Semantic Analysis, 101

Ł

ładowanie danych, 210

M

macierz, 50
  słowo – dokument, 102
MEDLINE, 122
metoda
  computeSVD, 109
  DNase-seq, 197
  innerJoin, 132
  naprzemiennych najmniejszych kwadratów, 49
  symulacji Monte Carlo, 167
  wikiXmlToPlainText, 111
metody
  obliczania wskaźnika VaR, 169
  pracy badacza danych, 226
metryka AUC, 59
metryki
  jakości grupowania, 100
  TF-IDF, 106
mieszany model Gaussa, 100
model programowania, 23
modele zmiennych ukrytych, 49
modelowanie danych, 188
moduł Spark Examples, 233
muzyka, 47

N

nakładka PySpark, 22
narzędzie
  Spray, 153
  Maven, 29
  PySpark, 206
nauka o danych, 15
niejawne informacje zwrotne, 48
niskowymiarowa reprezentacja danych, 113
normalizacja cech, 94

Kup ksi k

Pole ksi k

O

obiekty towarzyszące, 41
obliczenie
  metryki AUC, 59
  średniej długości ścieżki, 141
obsługa błędnych rekordów, 155
ocena
  jakości rekomendacji, 57
  wyników, 182
odchylenie standardowe, 217
odległość Mahalanobisa, 100
określenie wag czynników, 174

P

pakiet
  PySpark, 205
  Thunder, 205, 210
    instalacja, 209
    klasyfikowanie neuronów, 216
    ładowanie danych, 210
    typy danych, 214
Parquet, 195, 228
pary danych, 39
PDF, probability density function, 169
piksele, 211
platforma
  Avro, 188
  RPC, 189
plik
  covtype.data.gz, 71
  covtype.info, 71
  kddcup.data.gz, 88
  StatsWithMissing.scala, 40
pobranie danych, 104, 123, 148, 171
podprojekt
  GraphX, 230
  Spark SQL, 229
  Spark Streaming, 230
portfel, 168
proces klasyfikacji tekstu, 233
prognozowanie, 83
projekt
  1000 Genomes, 203
  BDG, 187
przestrzeń
  dokumentów, 109
  słów, 109
przesyłanie danych, 29
przetwarzanie danych
  genomicznych, 190
  geoprzestrzennych, 150
  temporalnych, 148, 149
przyciskanie predykatu, 195
przygotowanie danych, 51, 71, 104
punkty danych, 86

R

RDD, Resilient Distributed Dataset, 17, 27
regresja, 65
regulacja drzewa decyzyjnego, 77
rekomendacje, 56, 62
  ocena jakości, 57
  sprawdzanie, 56
REPL, 25
repozytorium Machine Learning Repository, 24
rozkład
  chi-kwadrat, 136
  stopni wierzchołków, 133
  według wartości osobliwych, SVD, 93, 102, 108
  zwrotów, 181
rozproszone przetwarzanie danych, BSP, 141
ryzyko finansowe, 167

S

Scala, 21
serializacja danych, 189, 224
sesjonowanie, 161
sieci typu „mały świat”, 139
SLA, service level agreement, 16
sortowanie danych, 162
Spark, 21
  model programowania, 23
SparkContext, 24
spodziewany niedobór, 168
sposoby zapisu, 188
spójność grafu, 131
sprawdzanie rekomendacji, 56
statystyki, 39
  chi-kwadrat, 136
  sumaryczne, 40
stopnie wierzchołków, 134
stosowanie pakietu Thunder, 209
stosy, 211
strata, 168
struktura
  EdgeTriplet, 136
  sieci, 131
SVD, singular value decomposition, 102
symulacja
  historyczna, 169
  Monte Carlo, 167, 169
system
  ADAM, 190
  Audioscrobbler, 48
  HDFS, 93
  MapReduce, 16
  Pregel, 141
  rekomendacyjny, 47
  Spark, 148, 161, 223
szacowanie ryzyka finansowego, 167

Ś

środkowe wartości klastrów, 220
środowisko
  IPython Notebook, 208
  klastrowe HPC, 14
  R, 93

T

teoria grafów, 121
test chi-kwadrat, 135
TF-IDF, 102
transformacje
  szerokie, 223
  wąskie, 223
tworzenie
  histogramów, 38
  klas wyboru, 33
  list danych, 33
  multigrafów, 131
  pierwszego modelu, 54
  współdzielonego kodu, 40

Kup ksi k   Pole ksi k

U

uczenie
  maszynowe, 85
  nadzorowane, 66
  nienadzorowane, 85
ukryta analiza semantyczna, LSA, 101
Using Scala, 26
uzupełnianie macierzy, 49

V

VaR, value at risk, 167

W

wagi czynników, 174
wariancja-kowariancja, 169
wartości osobliwe, 108
wartość zagrożona ryzykiem, 167
wektory, 66
  cech, 67, 69
weryfikacja cech kategorialnych, 79
wiązanie
  czynnika transkrypcyjnego, 197, 200
  rekordów danych, 23
wielkie zbiory danych, 13
wielowymiarowy rozkład normalny, 178
wizualizacja, 93
  rozkładu zwrotów, 181
włamania sieciowe, 87
właściwości, properties, 152
woksele, 211, 220
wskaźnik
  entropii, 97
  VaR, 169
współczynnik
  klastrowania sieci, 139, 140
  lokalnego klastrowania, 140
  Silhouette, 100
  błędów k-średnich, 219
wstępne przetworzenie danych, 171
wybór zmiennych, 44
wykrywanie anomalii, 85, 86
wyliczenie metryk TF-IDF, 106
wysyłanie kodu, 32
wyszukiwanie
  informacji, 113
  pojęć, 110
  wielu słów, 117

Z

zalesienie, 65
zanieczyszczenie Gini, 77
zapisywanie danych, 37
zawody KDD Cup, 87
zbiór
  danych, 48
  partycji, 28
  RDD, 27, 106
    instrumentów, 179
    parametrów, 179
zgodność rekordów, 44
zmienne, 67
  kategorialne, 96
  rozgłaszane, 54
związek
  dwóch dokumentów, 115
  dwóch słów, 114
  słowa i dokumentu, 116
zwrot, 168

Kup ksi k   Pole ksi k

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

http://program-partnerski.helion.pl

O'REILLY®

Poznaj potencjał i wydajność Apache Spark!

Apache Spark to darmowy, zaawansowany szkielet i silnik pozwalający na szybkie przetwarzanie oraz analizę ogromnych zbiorów danych. Prace nad tym projektem rozpoczęły się w 2009 roku, a już rok później Spark został udostępniony użytkownikom. Jeżeli potrzebujesz najwyższej wydajności w przetwarzaniu informacji, jeżeli chcesz uzyskiwać odpowiedzi na trudne pytania niemalże w czasie rzeczywistym, Spark może spełnić Twoje oczekiwania.

Sięgnij po tę książkę i przekonaj się, czy tak jest w rzeczywistości. Autor porusza tu zaawansowane kwestie związane z analizą statystyczną danych, wykrywaniem anomalii oraz analizą obrazów. Jednak zanim przejdziesz do tych tematów, zapoznasz się z podstawami — wprowadzeniem do analizy danych za pomocą języka Scala oraz Apache Spark. Nauczysz się też przeprowadzać analizę semantyczną i zobaczysz, jak w praktyce przeprowadzić analizę sieci współwystępowań za pomocą biblioteki GraphX. Na koniec dowiesz się, jak przetwarzać dane geoprzestrzenne i genomiczne, a także oszacujesz ryzyko metodą symulacji Monte Carlo. Książka ta pozwoli Ci na wykorzystanie potencjału Apache Spark i zaprzęgnięcie go do najtrudniejszych zadań!

Dzięki tej książce:

- nauczysz się podstaw języka Scala
- poznasz zaawansowane możliwości Apache Spark
- przeprowadzisz analizę danych geoprzestrzennych
- wykorzystasz biblioteki GraphX oraz Esri Geometry
- przekonasz się, że analiza ogromnych zbiorów danych nie musi być czasochłonna!

Sandy Ryza — starszy analityk w firmie Cloudera, aktywnie zaangażowany w projekt Apache Spark.

Uri Laserson — starszy analityk w firmie Cloudera, zaangażowany w obsługę języka Python w środowisku Hadoop.

Sean Owen — dyrektor działu analiz w firmie Cloudera, uczestnik projektu Apache Spark.

Josh Wills — starszy menedżer działu analiz w firmie Cloudera, inicjator prac nad pakietem Apache Crunch.

Helion

38000 numer katalogowy

księgarnia internetowa
http://helion.pl

zamówienia telefoniczne
0 801 339900
0 601 339900

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

ISBN 978-83-283-1461-0

cena: 49,00 zł