Suresh Kumar Gorakala

Building Recommendation Engines

Understand your data and user preferences to make intelligent, accurate, and profitable decisions

Packt

Building Recommendation Engines

Understand your data and user preferences to make intelligent, accurate, and profitable decisions

Suresh Kumar Gorakala

BIRMINGHAM - MUMBAI

Building Recommendation Engines

Copyright © 2016 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: December 2016

Production reference: 1231216

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.
ISBN 978-1-78588-485-6

www.packtpub.com

Credits

Author

Suresh Kumar Gorakala

Reviewers

Vikram Dhillon

Vimal Romeo

Commissioning Editor

Veena Pagare

Acquisition Editor

Tushar Gupta

Content Development Editor

Manthan Raja

Technical Editor

Dinesh Chaudhary

Copy Editor

Manisha Sinha

Project Coordinator

Nidhi Joshi

Proofreader

Safis Editing

Indexer

Mariammal Chettiyar

Graphics

Disha Haria

Production Coordinator

Arvindkumar Gupta

About the Author

Suresh Kumar Gorakala is a Data scientist focused on Artificial Intelligence. He has professional experience close to 10 years, having worked with various global clients across multiple domains and helped them in solving their business problems using Advanced Big Data Analytics. He has extensively worked on Recommendation Engines, Natural language Processing, Advanced Machine Learning, Graph Databases. He previously co-authored Building a Recommendation System with R for Packt Publishing. He is passionate traveler and is photographer by hobby.

I would like to thank my wife for putting up with my late-night writing sessions and all my family members for supporting me over the months. I also give deep thanks and gratitude to Barathi Ganesh, Raj Deepthi, Harsh and my colleagues who without their support this book quite possibly would not have happened. I would also like to thank all the mentors that I've had over the years. Without learning from these teachers, there is not a chance I could be doing what I do today, and it is because of them and others that I may not have listed here that I feel compelled to pass my knowledge on to those willing to learn. I would also like to thank all the reviewers and project managers of the book to make it a reality.

About the Reviewers

Vikram Dhillon is a software developer, a bioinformatics researcher, and a software coach at the Blackstone LaunchPad in the University of Central Florida. He has been working on his own startup involving healthcare data security of late. He lives in Orlando and regularly attends development meetups and hackathons. He enjoys spending his spare time reading about new technologies, such as the Blockchain and developing tutorials for machine learning in game design. He has been involved in open-source projects for over five years and writes about technology and startups at `opsbug.com`

Vimal Romeo is a data science at Ernst and Young, Rome. He holds a master's degree in Big Data Analytics from Luiss Business School, Rome. He also holds an MBA degree from XIME ,India and a bachelor's degree in computer science and engineering from CUSAT, India. He is an author at MilanoR which is a blog related to the R language.

I would like to thank my mom – Mrs Bernadit and my brother - Vibin for their continuous support. I would also like to thank my friends – Matteo Amadei, Antonella Di Luca, Asish Mathew and Eleonora Polidoro who supported me during this process. A special thanks to Nidhi Joshi from Packt Publishing for keeping me motivated during the process.

www.PacktPub.com

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

https://www.packtpub.com/mapt

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Love You Mom

Table of Contents

Preface 1

Chapter 1: Introduction to Recommendation Engines 7

- **Recommendation engine definition** 7
- **Need for recommender systems** 10
- **Big data driving the recommender systems** 10
- **Types of recommender systems** 11
  - Collaborative filtering recommender systems 11
  - Content-based recommender systems 13
  - Hybrid recommender systems 14
  - Context-aware recommender systems 16
- **Evolution of recommender systems with technology** 17
  - Mahout for scalable recommender systems 17
  - Apache Spark for scalable real-time recommender systems 18
    - Neo4j for real-time graph-based recommender systems 20
- **Summary** 22

Chapter 2: Build Your First Recommendation Engine 23

- **Building our basic recommendation engine** 25
  - Loading and formatting data 25
  - Calculating similarity between users 29
  - Predicting the unknown ratings for users 30
- **Summary** 38

Chapter 3: Recommendation Engines Explained 39

- **Evolution of recommendation engines** 40
- **Nearest neighborhood-based recommendation engines** 42
  - User-based collaborative filtering 44
  - Item-based collaborative filtering 47
  - Advantages 49
  - Disadvantages 50
- **Content-based recommender systems** 50
  - User profile generation 54
  - Advantages 56
  - Disadvantages 56
- **Context-aware recommender systems** 56

Context definition 58
Pre-filtering approaches 61
Post-filtering approaches 62
Advantages 62
Disadvantages 63
Hybrid recommender systems 63
Weighted method 63
Mixed method 64
Cascade method 64
Feature combination method 64
Advantages 65
Model-based recommender systems 65
Probabilistic approaches 66
Machine learning approaches 66
Mathematical approaches 66
Advantages 67
Summary 67
Chapter 4: Data Mining Techniques Used in Recommendation Engines 68
Neighbourhood-based techniques 69
Euclidean distance 70
Cosine similarity 71
Jaccard similarity 74
Pearson correlation coefficient 75
Mathematic model techniques 78
Matrix factorization 78
Alternating least squares 81
Singular value decomposition 82
Machine learning techniques 84
Linear regression 84
Classification models 86
Linear classification 87
KNN classification 88
Support vector machines 90
Decision trees 93
Ensemble methods 96
Random forests 96
Bagging 97
Boosting 98
Clustering techniques 100
K-means clustering 101

Dimensionality reduction 103
Principal component analysis 104
Vector space models 108
Term frequency 109
Term frequency inverse document frequency 110
Evaluation techniques 113
Cross-validation 114
Regularization 115
Root-mean-square error (RMSE) 115
Mean absolute error (MAE) 116
Precision and recall 117
Summary 119
Chapter 5: Building Collaborative Filtering Recommendation Engines 120
Installing the recommenderlab package in RStudio 121
Datasets available in the recommenderlab package 122
Exploring the Jester5K dataset 123
Description 123
Usage 123
Format 124
Details 124
Exploring the dataset 126
Exploring the rating values 127
Building user-based collaborative filtering with recommenderlab 128
Preparing training and test data 129
Creating a user-based collaborative model 129
Predictions on the test set 131
Analyzing the dataset 133
Evaluating the recommendation model using the k-cross validation 135
Evaluating user-based collaborative filtering 137
Building an item-based recommender model 141
Building an IBCF recommender model 142
Model evaluation 145
Model accuracy using metrics 147
Model accuracy using plots 148
Parameter tuning for IBCF 151
Collaborative filtering using Python 154
Installing the required packages 155
Data source 155
Data exploration 156
Rating matrix representation 158

Creating training and test sets 160
The steps for building a UBCF 161
User-based similarity calculation 161
Predicting the unknown ratings for an active user 162
User-based collaborative filtering with the k-nearest neighbors 163
Finding the top-N nearest neighbors 163
Item-based recommendations 165
Evaluating the model 166
The training model for k-nearest neighbors 167
Evaluating the model 167
Summary 168
Chapter 6: Building Personalized Recommendation Engines 169
Personalized recommender systems 170
Content-based recommender systems 170
Building a content-based recommendation system 171
Content-based recommendation using R 172
Dataset description 175
Content-based recommendation using Python 184
Dataset description 186
User activity 189
Item profile generation 193
User profile creation 195
Context-aware recommender systems 199
Building a context-aware recommender systems 200
Context-aware recommendations using R 201
Defining the context 202
Creating context profile 203
Generating context-aware recommendations 205
Summary 207
Chapter 7: Building Real-Time Recommendation Engines with Spark 208
About Spark 2.0 209
Spark architecture 210
Spark components 212
Spark Core 212
Structured data with Spark SQL 212
Streaming analytics with Spark Streaming 213
Machine learning with MLlib 213
Graph computation with GraphX 214
Benefits of Spark 215
Setting up Spark 215

About SparkSession 216
Resilient Distributed Datasets (RDD) 217
About ML Pipelines 218
Collaborative filtering using Alternating Least Square 220
Model based recommender system using pyspark 223
MLlib recommendation engine module 225
The recommendation engine approach 225
Implementation 226
Data loading 226
Data exploration 228
Building the basic recommendation engine 233
Making predictions 234
User-based collaborative filtering 236
Model evaluation 237
Model selection and hyperparameter tuning 238
Cross-Validation 239
CrossValidator 239
Train-Validation Split 239
Setting the ParamMaps/parameters 242
Setting the evaluator object 243
Summary 244
Chapter 8: Building Real-Time Recommendations with Neo4j 245
Discerning different graph databases 246
Labeled property graph 248
Understanding GraphDB core concepts 248
Neo4j 250
Cypher query language 250
Cypher query basics 251
Node syntax 251
Relationship syntax 251
Building your first graph 252
Creating nodes 253
Creating relationships 254
Setting properties to relations 256
Loading data from csv 259
Neo4j Windows installation 261
Installing Neo4j on the Linux platform 263
Downloading Neo4j 263
Setting up Neo4j 264
Starting Neo4j from the command line 264
Building recommendation engines 267

Loading data into Neo4j 268
Generating recommendations using Neo4j 272
Collaborative filtering using the Euclidean distance 273
Collaborative filtering using Cosine similarity 279
Summary 282
Chapter 9: Building Scalable Recommendation Engines with Mahout 283
Mahout – a general introduction 284
Setting up Mahout 285
The standalone mode – using Mahout as a library 285
Setting Mahout for the distributed mode 293
Core building blocks of Mahout 295
Components of a user-based collaborative recommendation engine 296
Building recommendation engines using Mahout 300
Dataset description 300
User-based collaborative filtering 303
Item-based collaborative filtering 306
Evaluating collaborative filtering 309
Evaluating user-based recommenders 310
Evaluating item-based recommenders 311
SVD recommenders 314
Distributed recommendations using Mahout 315
ALS recommendation on Hadoop 316
The architecture for a scalable system 321
Summary 322
Chapter 10: What Next - The Future of Recommendation Engines 323
Future of recommendation engines 324
Phases of recommendation engines 324
Phase 1 – general recommendation engines 325
Phase 2 – personalized recommender systems 326
Phase 3 – futuristic recommender systems 328
End of search 330
Leaving the Web behind 332
Emerging from the Web 333
Next best actions 334
Use cases to look out for 335
Smart homes 335
Healthcare recommender systems 336
News as recommendations 336
Popular methodologies 337

Serendipity 337
Temporal aspects of recommendation engines 338
A/B testing 340
Feedback mechanism 341
Summary 341
Index 342

Preface

Building Recommendation Engines is a comprehensive guide for implementing Recommendation Engines such as collaborative filtering, content based recommendation engines, context aware recommendation engines using R, Python, Spark, Mahout, Neo4j technologies. The book covers various recommendation engines widely used across industries with their implementations. This book also covers a chapter on popular datamining techniques commonly used in building recommendations and also discuss in brief about the future of recommendation engines at the end of the book.

What this book covers

`Chapter 1`, *Introduction to Recommendation Engines*, will be a refresher to Data Scientists and an introduction to the beginners of recommendation engines. This chapter introduces popular recommendation engines that people use in their day-to-day lives. Popular recommendation engine approaches available along with their pros and cons are covered.

`Chapter 2`, *Build Your First Recommendation Engine*, is a short chapter about how to build a movie recommendation engine to give a head start for us before we take off into the world of recommendation engines.

`Chapter 3`, *Recommendation Engines Explained*, is about different recommendation engine techniques popularly employed, such as user-based collaborative filtering recommendation engines, item-based collaborative filtering, content-based recommendation engines, context-aware recommenders, hybrid recommenders, model-based recommender systems using Machine Learning models and mathematical models.

`Chapter 4`, *Data Mining Techniques Used in Recommendation Engines*, is about various Machine Learning techniques used in building recommendation engines such as similarity measures, classification, regression, and dimension reduction techniques. This chapter also covers evaluation metrics to test the recommendation engine's predictive power.

`Chapter 5`, *Building Collaborative Filtering Recommendation Engines*, is about how to build user-based collaborative filtering and item-based collaborative filtering in R and Python. We'll also learn about different libraries available in R and Python that are extensively used in building recommendation engines.

`Chapter 6`, *Building Personalized Recommendation Engines*, is about how to build personalized recommendation engines using R and Python and the various libraries used for building content-based recommender systems and context-aware recommendation engines.

`Chapter 7`, *Building Real-Time Recommendation Engines with Spark*, is about the basics of Spark and MLlib required for building real-time recommender systems.

`Chapter 8`, *Building Real-Time Recommendation Engines with Neo4j*, is about the basics of graphDB and Neo4j concepts and how to build real-time recommender systems using Neo4j.

`Chapter 9`, *Building Scalable Recommendation Engines with Mahout*, is about the basic building blocks of Hadoop and Mahout required for building scalable recommender systems. It also covers the architecture we use to build scalable systems and a step-by-step implementation using Mahout and SVD.

`Chapter 10`, *What Next?*, is the final chapter explaining the summary of what we have learned so far: best practices that are employed in building the decision-making systems and where the future of the recommender systems are set to move.

What you need for this book

To get started with different implementations of recommendation engines in R, Python, Spark, Neo4j, Mahout we need the following software:

Chapter number	**Software required (With version)**	**Download links to the software**	**OS required**
2,4,5	R studio Version 0.99.489	`https://www.rstudio.com/products/rstudio/download/`	WINDOWS 7+/Centos 6
2,4,5	R version 3.2.2	`https://cran.r-project.org/bin/windows/base/`	WINDOWS 7+/Centos 6
5,6,7	Anaconda 4.2 for Python 3.5	`https://www.continuum.io/downloads`	WINDOWS 7+/Centos 6
8	Neo4j 3.0.6	`https://neo4j.com/download/`	WINDOWS 7+/Centos 6
7	Spark 2.0	`https://spark.apache.org/downloads.html`	WINDOWS 7+/Centos 6
9	Hadoop 2.5 -Mahout 0.12	`http://hadoop.apache.org/releases.html` `http://mahout.apache.org/general/downloads.html`	WINDOWS 7+/Centos 6
7,9,8	Java 7/Java 8	`http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html`	WINDOWS 7+/Centos 6

Who this book is for

This book caters to beginners and experienced data scientists looking to understand and build complex predictive decision-making systems, recommendation engines using R, Python, Spark, Neo4j, and Hadoop.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can include other contexts through the use of the include directive."

A block of code is set as follows:

```
export MAHOUT_HOME = /home/softwares/ apache-mahout-distribution-0.12.2
export MAHOUT_LOCAL = true #for standalone mode
export PATH = $MAHOUT_HOME/bin
export CLASSPATH = $MAHOUT_HOME/lib:$CLASSPATH
```

Any command-line input or output is written as follows:

```
[cloudera@quickstart ~]$ hadoop fs -ls
Found 1 items
drwxr-xr-x - cloudera cloudera 0 2016-11-14 18:31 mahout
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Clicking the **Next** button moves you to the next screen."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support`and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at https://github.com/PacktPublishing/building-recommendation-engines. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from http://www.packtpub.com/sites/default/files/downloads/BuildingRecommendationEngines_ColorImages.pdf.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to https://www.packtpub.com/books/content/supportand enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Introduction to Recommendation Engines

How do we buy things? How do we make decisions in our day-to-day lives? We ask our friends or relatives for suggestions before making decisions. When it comes to making decisions online about buying products, we read reviews about the products from anonymous users, compare the products' specifications with other similar products and then we make our decisions to buy or not. In an online world, where information is growing at an exponential rate, looking for valid information will be a challenge. Buying the confidence of the user for the search results will be a much more challenging task. Recommender systems come to our rescue to provide relevant and required information.

The popularity of implementing recommendation engines comes as a result of their successful implementation by big players on the Internet. Some real-world examples include suggestions for products on Amazon, friends' suggestions on social applications such as Facebook, Twitter, and LinkedIn, video recommendations on YouTube, news recommendations on Google News, and so on. These successful implementations of recommendation engines have shown the way for other areas such as the travel, healthcare, and banking domains.

Recommendation engine definition

Recommendation engines, a branch of information retrieval and artificial intelligence, are powerful tools and techniques to analyze huge volumes of data, especially product information and user information, and then provide relevant suggestions based on data-mining approaches.

In technical terms, a recommendation engine problem is *to develop a mathematical model or objective function which can predict how much a user will like an item*.

If $U = \{users\}$, $I = \{items\}$ then F = *Objective function* and measures the usefulness of item I to user U, given by:

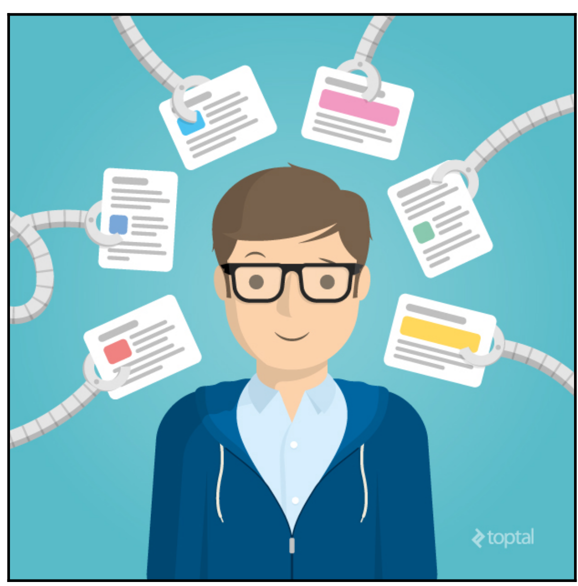
$$F{:}U\ X\ I \rightarrow R$$

Where $R = \{recommended\ items\}$.

For each user u, we want to choose the item i that maximizes the objective function:

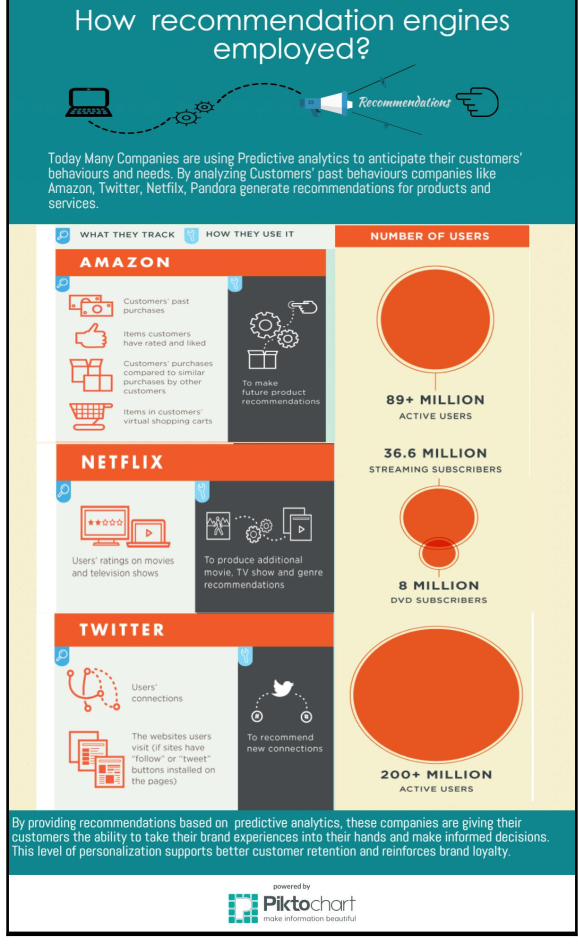
$$\mathbf{u} \in \boldsymbol{U}, \boldsymbol{I}'_{\boldsymbol{u}} = \boldsymbol{argmax_u u(u, i)}$$

The main goal of recommender systems is to provide relevant suggestions to online users to make better decisions from many alternatives available over the Web. A better recommender system is directed more toward personalized recommendations by taking into consideration the available digital footprint of the user, such as user-demographic information, transaction details, interaction logs, and information about a product, such as specifications, feedback from users, comparison with other products, and so on, before making recommendations:



Pic credits: toptal

Building a good recommendation engine poses challenges to both the actors of the system, namely, the consumers and sellers. From a consumer perspective, receiving relevant suggestions from a trusted source is critical for decision making. So the recommendation engine needs to build in such a way that it buys the confidence of the consumers. From a seller perspective, generating relevant recommendations to consumers at a personalized level is more important. With the rise of online sales, the big players are now collecting large volumes of transactional interaction logs of users to analyze the user behaviors more deeply than ever. Also, the need to recommend in real time is adding to the challenge. With advancements in technology and research, recommendation engines are evolving to overcome these challenges based on big-data analysis and artificial intelligence. The following diagram illustrates how organizations employ recommendation engines:



Need for recommender systems

Given the complexity and challenges in building recommendation engines, a considerable amount of thought, skill, investment, and technology goes into building recommender systems. Are they worth such an investment? Let us look at some facts:

- Two-thirds of movies watched by Netflix customers are recommended movies
- 38% of click-through rates on Google News are recommended links
- 35% of sales at Amazon arise from recommended products
- ChoiceStream claims that 28% of people would like to buy more music, if they find what they like

Big data driving the recommender systems

Of late, recommender systems are successful in impacting our lives in many ways. One such obvious example of this impact is how our online shopping experience has been redefined. As we browse through e-commerce sites and purchase products, the underlying recommendation engines respond immediately, in real time, with various relevant suggestions to consumers. Regardless of the perspective, from business player or consumer, recommendation engines have been immensely beneficial. Without a doubt, big data is the driving force behind recommender systems. A good recommendation engine should be reliable, scalable, highly available, and be able to provide personalized recommendations, in real time, to the large user base it contains.

A typical recommendation system cannot do its job efficiently without sufficient data. The introduction of big data technology enabled companies to capture plenty of user data, such as past purchases, browsing history, and feedback information, and feed it to the recommendation engines to generate relevant and effective recommendations in real time. In short, even the most advanced recommender system cannot be effective without the supply of big data. The role of big data and improvements in technology, both on the software and hardware front, goes beyond just supplying massive data. It also provides meaningful, actionable data fast, and provides the necessary setup to quickly process the data in real time.

Source:
`http://www.kdnuggets.com/2015/10/big-data-recommendation-systems-change-lives.html`.

Types of recommender systems

Now that we have defined recommender systems, their objective, usefulness, and the driving force behind recommender systems, in this section, we introduce different types of popular recommender systems in use.

Collaborative filtering recommender systems

Collaborative filtering recommender systems are basic forms of recommendation engines. In this type of recommendation engine, filtering items from a large set of alternatives is done collaboratively by users' preferences.

The basic assumption in a collaborative filtering recommender system is that if two users shared the same interests as each other in the past, they will also have similar tastes in the future. If, for example, user A and user B have similar movie preferences, and user A recently watched *Titanic,* which user B has not yet seen, then the idea is to recommend this unseen new movie to user B. The movie recommendations on Netflix are one good example of this type of recommender system.

There are two types of collaborative filtering recommender systems:

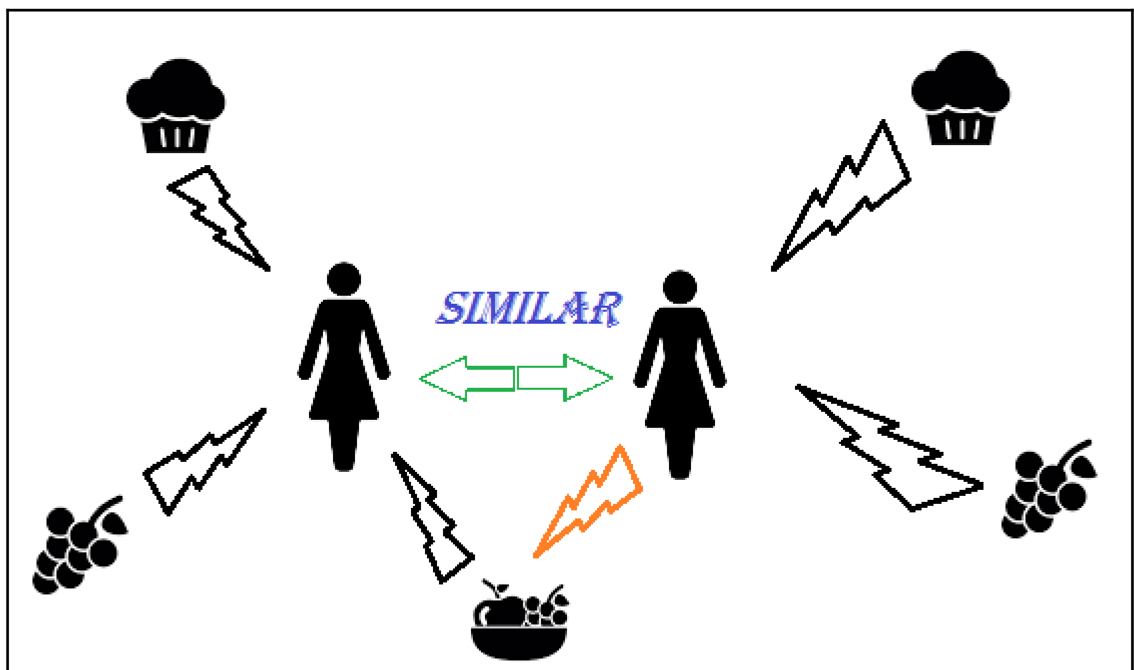
- **User-based collaborative filtering**: In user-based collaborative filtering, recommendations are generated by considering the preferences in the user's neighborhood. User-based collaborative filtering is done in two steps:
    - Identify similar users based on similar user preferences
    - Recommend new items to an active user based on the rating given by similar users on the items not rated by the active user.
- **Item-based collaborative filtering**: In item-based collaborative filtering, the recommendations are generated using the neighbourhood of items. Unlike user-based collaborative filtering, we first find similarities between items and then recommend non-rated items which are similar to the items the active user has rated in past. Item-based recommender systems are constructed in two steps:
    - Calculate the item similarity based on the item preferences
    - Find the top similar items to the non-rated items by active user and recommend them

We will learn in depth about these two forms of recommendations in `Chapter 3`, *Recommendation Engines Explained*.

While building collaborative filtering recommender systems, we will learn about the following aspects:

- How to calculate the similarity between users?
- How to calculate the similarity between items?
- How recommendations are generated?
- How to deal with new items and new users whose data is not known?

The advantage of collaborative filtering systems is that they are simple to implement and very accurate. However, they have their own set of limitations, such as the *Cold Start* problem, which means, collaborative filtering systems fails to recommend to the first-time users whose information is not available in the system:



Content-based recommender systems

In collaborative filtering, we consider only user-item-preferences and build the recommender systems. Though this approach is accurate, it makes more sense if we consider user properties and item properties while building recommendation engines. Unlike in collaborative filtering, we use item properties and user preferences to the item properties while building content-based recommendation engines.

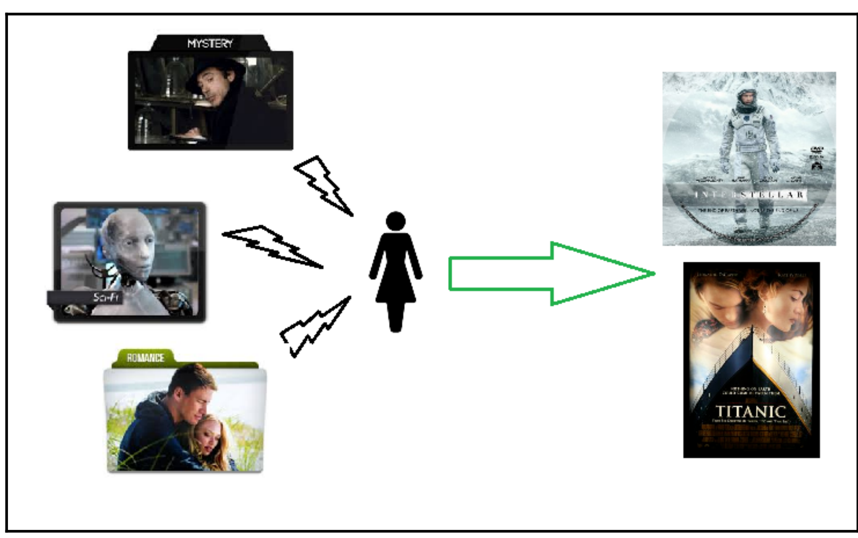
As the name indicates, a content-based recommender system uses the content information of the items for building the recommendation model. A content recommender system typically contains a user-profile-generation step, item-profile-generation step- and model-building step to generate recommendations for an active user. The content-based recommender system recommends items to users by taking the content or features of items and user profiles. As an example, if you have searched for videos of Lionel Messi on YouTube, then the content-based recommender system will learn your preference and recommend other videos related to Lionel Messi and other videos related to football.

In simpler terms, the system recommends items similar to those that the user has liked in the past. The similarity of items is calculated based on the features associated with the other compared items and is matched with the user's historical preferences.

While building a content-based recommendation system, we take into consideration the following questions:

- How do we choose content or features of the products?
- How do we create user profiles with preferences similar to that of the product content?
- How do we create similarity between items based on their features?
- How do we create and update user profiles continuously?

The preceding considerations will be explained in `Chapter 3`, *Recommendation Engines Explained*. This technique doesn't take into consideration the user's neighborhood preferences. Hence, it doesn't require a large user group's preference for items for better recommendation accuracy. It only considers the user's past preferences and the properties/features of the items. In `Chapter 3`, *Recommendation Engines Explained*, we will learn about this system in detail, and also its pros and cons:



Hybrid recommender systems

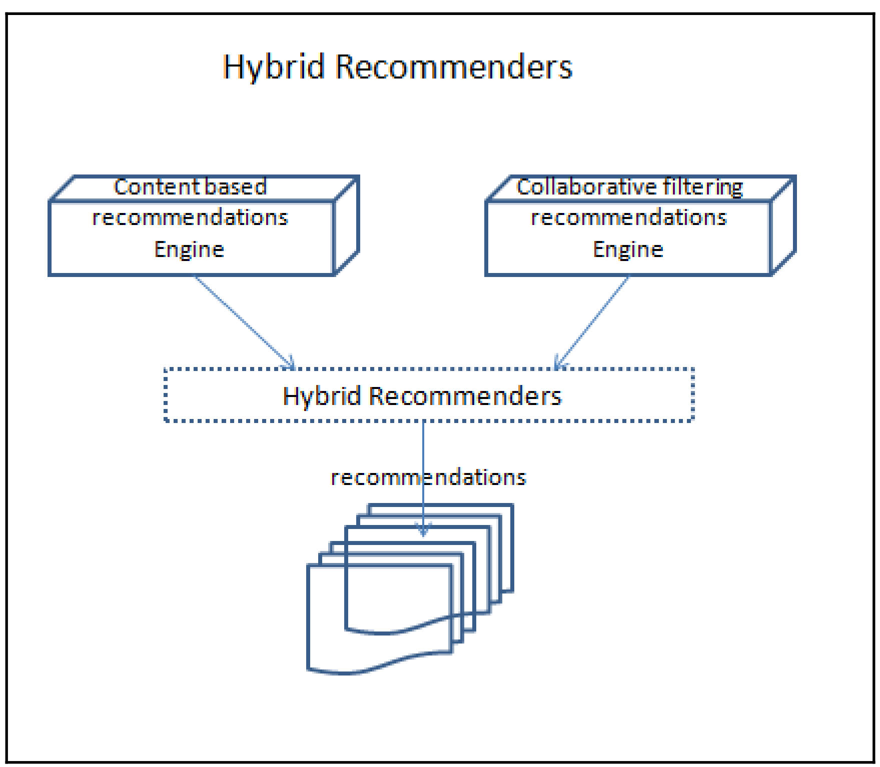
This type of recommendation engine is built by combining various recommender systems to build a more robust system. By combining various recommender systems, we can replace the disadvantages of one system with the advantages of another system and thus build a more robust system. For example, by combining collaborative filtering methods, where the model fails when new items don't have ratings, with content-based systems, where feature information about the items is available, new items can be recommended more accurately and efficiently.

For example, if you are a frequent reader of news on Google News, the underlying recommendation engine recommends news articles to you by combining popular news articles read by people similar to you and using your personal preferences, calculated using your previous click information. With this type of recommendation system, collaborative filtering recommendations are combined with content-based recommendations before pushing recommendations.

Before building a hybrid model, we should consider the following questions:

- What recommender techniques should be combined to achieve the business solution?
- How should we combine various techniques and their results for better predictions?

The advantage of hybrid recommendation engines is that this approach will increase the efficiency of recommendations compared to the individual recommendation techniques. This approach also suggests a good mix of recommendations to the users, both at the personalized level and at the neighborhood level. In `Chapter 3`, *Recommendation Engines Explained*, we will learn more about hybrid recommendations:



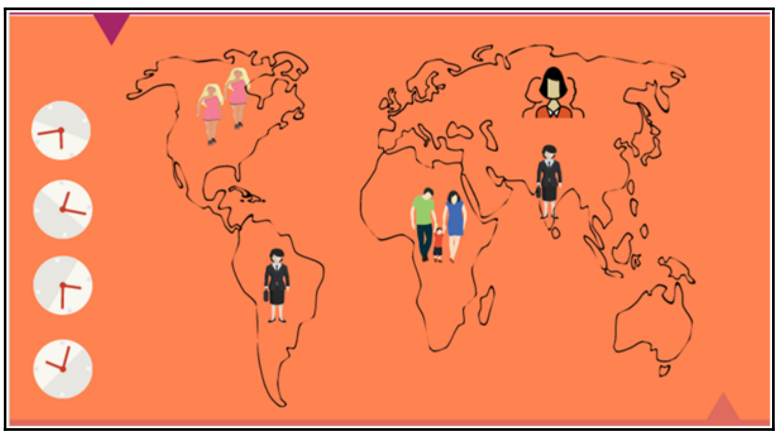
Context-aware recommender systems

Personalized recommender systems, such as content-based recommender systems, are inefficient; they fail to suggest recommendations with respect to context. For example, assume a lady is very fond of ice-cream. Also assume that this lady goes to a cold place. Now there is high chance that a personalized recommender system suggests a popular ice-cream brand. Now let us ask our self a question: is it the right thing to suggest an ice-cream to a person in a cold place? Rather, it makes sense to suggest a coffee. This type of recommendation, which is personalized and context-aware is called a context-aware recommender systems. In the preceding example, place is the context.

User preferences may differ with the context, such as time of day, season, mood, place, location, options offered by the system, and so on. A person at a different location at a different time with different people may need different things. A context-aware recommender system takes the context into account before computing or serving recommendations. This recommender system caters for the different needs of people differently in different contexts.

Before building a context-aware model, we should consider the following questions:

- How should we define the contexts to be used in the recommender system?
- What techniques should be used to build recommendations to achieve the business solution?
- How do we extract context the preferences of the users with respect to the products?
- What techniques should we use to combine the context preferences with user-profile preferences to generate recommendations?



The preceding image shows how different people, at different times and places, and with different company, need different dress recommendations.

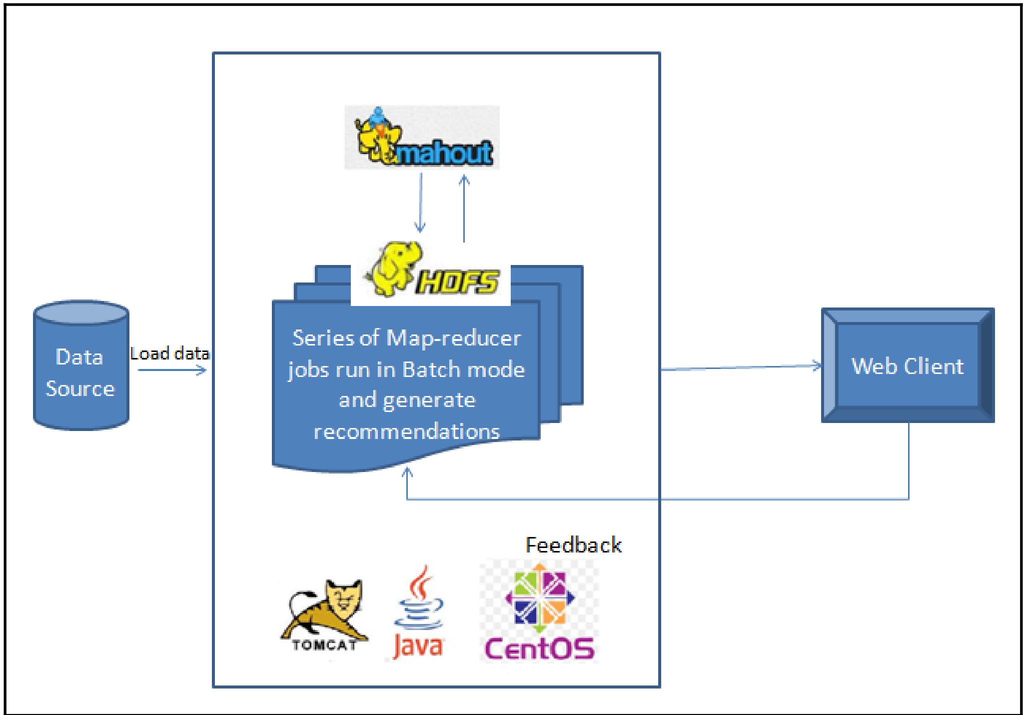
Evolution of recommender systems with technology

With the advancements in technology, research, and infrastructure, recommender systems have been evolving rapidly. Recommender systems are moving away from simple similarity-measure-based approaches, to machine-learning approaches, to very advanced approaches such as deep learning. From a business angle, both customers and organizations are looking toward more personalized recommendations to be catered for immediately. Building personalized recommenders to cater to the large user base and products, we need sophisticated systems, which can scale easily and respond fast. The following are the types of recommendations that can help solve this challenge.

Mahout for scalable recommender systems

As stated earlier, big data primarily drives recommender systems. The big-data platforms enabled researchers to access large datasets and analyze data at the individual level, paving paths for building personalized recommender systems. With increase in Internet usage and a constant supply of data, efficient recommenders not only require huge data, but also need infrastructure which can scale and have minimum downtime. To realize this, big-data technology such as the Apache Hadoop ecosystem provided the infrastructure and platform to supply large data. To build recommendation systems on this huge supply of data, **Mahout**, a machine-learning library built on the Hadoop platform enables us to build scalable recommender systems. Mahout provides infrastructure to build, evaluate, and tune the different types of recommendation-engine algorithms. Since Hadoop is designed for offline batch processing, we can build offline recommender systems, which are scalable. In `Chapter 9`, *Building Scalable Recommendation Engines with Mahout*, we further see how to build scalable recommendation engines using Mahout.

The following figure displays how a scalable recommender system can be designed using Mahout:

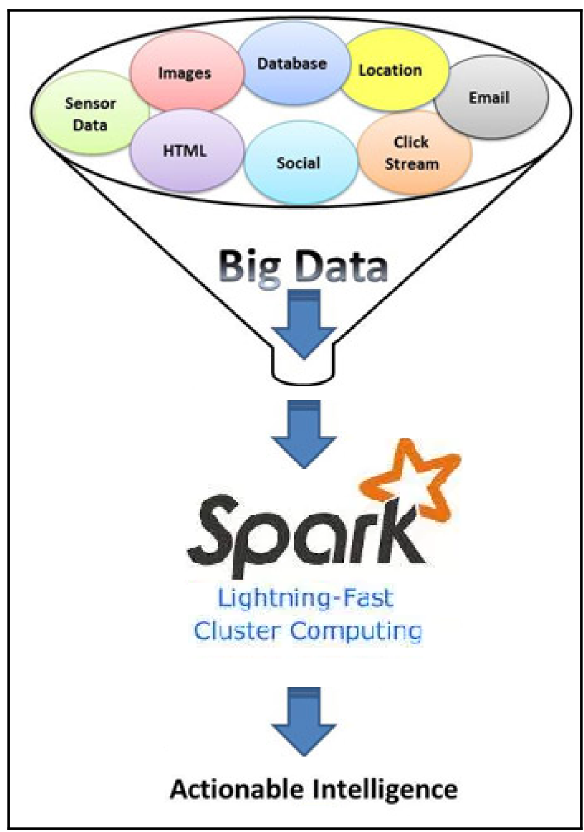


Apache Spark for scalable real-time recommender systems

We have seen many times, on any of the e-commerce sites, the *You may also like* feature. This is a deceptively simple phrase that encapsulates a new era in customer relationship management delivered in real time. Business organizations started investing in such systems, which can generate recommendations personalized to the customers and can deliver them in real time. Building such a system will not only give good returns on investment but also, efficient systems will buy the confidence of the users. Building a scalable real-time recommender system will not only capture users' purchase history, product information, user preferences, and extract patterns and recommend products, but will also respond instantly based on user online interactions and multi-criteria search preferences.

This ability makes compelling suggestions requiring a new generation of technology. This technology has to consider large databases of users' previous purchasing history, their preferences, and online interaction information such as in-page navigation data and multi-criteria searches, and then analyzes all this information in real time and responds accurately according to the current and long-term needs of the users. In this book, we have considered in-memory and graph-based systems, which are capable of handling large-scale, real-time recommender systems.

Most popular recommendation engine collaborative filtering requires considering the entirety of users and product information while generating recommendations. Assume a scenario where we have 1 million user ratings on 10,000 products. In order to build a system to handle such heavy computations and respond online, we require a system that is big-data compatible and processes data in-memory. The key technology in enabling scalable, real-time recommendations is Apache Spark Streaming, a technology that leverages scalability of big data and generates recommendations in real time, and processes data in-memory:



Neo4j for real-time graph-based recommender systems

Graph databases have revolutionized the way people discover new products, information, and so on. In the human mind, we remember people, things, places, and so on, as graphs, relations, and networks. When we try to fetch information from these networks, we directly go to a required connection or graph and fetch information accurately. In a similar fashion, graph databases allow us to store user and product information in graphs as nodes and edges (relations). Searching in a graph database is fast. In recent times, recommender systems powered by graph databases have allowed organizations to build suggestions which are personalized and accurate in real time.

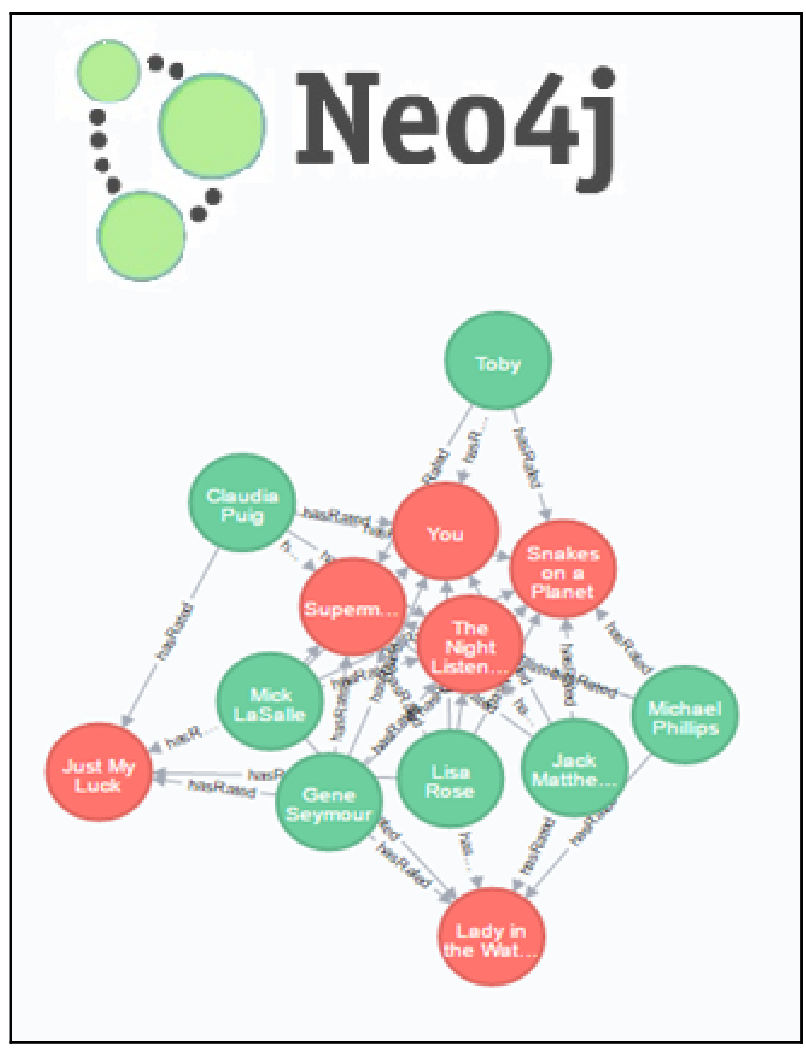
One of the key technologies enabling real-time recommendations using graph databases is Neo4j, a kind of NoSQL graph database that can easily outperform any other relational and NoSQL system in providing customer insights and product trends.

A NoSQL database, popularly known as *not only SQL*, provides a new way of storing and managing data other than in relational format that is row and columns such as columnar, graph, key-value pair store of data. This new way of storing and managing data enables us to build scalable and real-time systems.

A graph database mainly consists of nodes and edges, wherein nodes represent the entities and edges the relations between them. The edges are directed lines or arrows that connect the nodes. In the preceding image, the circles are the nodes, which represent the entities, and the lines connecting the nodes are called edges, which represent relationships. The orientation of arrows follows the flow of information. By presenting all nodes and links of the graph, it helps users to have a global view of the structure.

The following image shows user-movie-rating information representation in graph form. Green and red circles indicate nodes representing users and movies, respectively. The ratings given by users to movies are represented as edges showing the relationship between users and movies. Each node and relation may contain properties to store further details of the data.

On this graph representation, we apply concepts of graph theory to generate recommendations in real time, as the retrieval and searching is very fast. In `Chapter 8`, *Building Real Time Recommendation Engines with Neo4j*, we deal with building real-time recommendation engines using Neo4j:



Summary

In this chapter, we got introduced to various types of popular recommendation engines such as collaborative filtering, content-based recommendation engines, hybrid recommenders, context-aware systems, scalable recommenders, and graph-based, real-time recommenders.

We also learned how big data is fuelling the rise of recommendation engines and some real-world use-cases employed by major IT giants. In `Chapter 3`, *Recommendation Engines Explained*, we will learn more about these recommendations in detail. In `Chapter 2`, *Build Your First Recommendation Engine*, we learn how to build a basic recommendation engine using R.

2
Build Your First Recommendation Engine

In the previous chapter, we had an introduction to various types of recommendation engines, which we will be building in the subsequent chapters. Now that we have got introduced to recommendation engines, let's build our first recommendation engine using R.

Before we proceed further for implementation, let's have a brief discussion about the required software and packages for building our first recommendation using R.

For this exercise, we have used the R 3.2.2 version and RStudio 0.99 or above. For installation and setup of R and RStudio, please refer to the software and hardware list section of the book.

The R packages we have used for this exercise are as follows:

- `dplyr`
- `data.table`
- `reshape2`

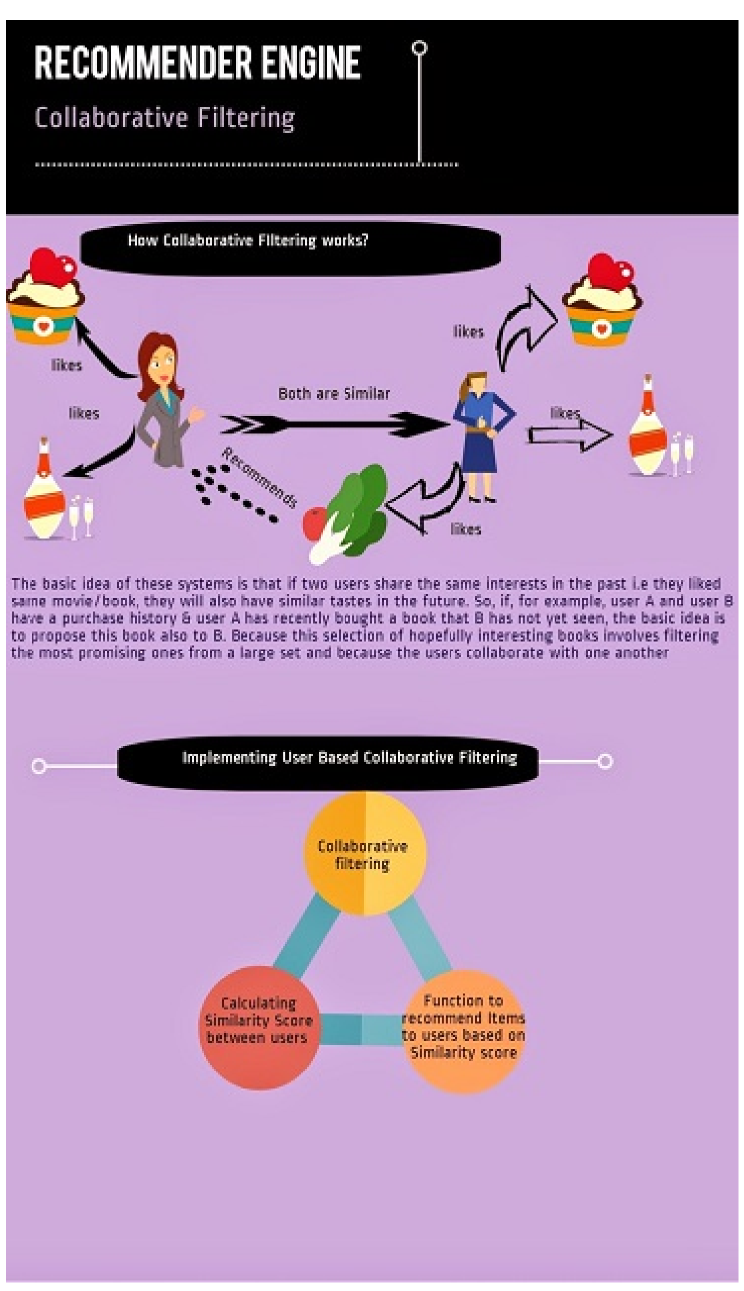
Installing a R package is given by the following codes:

```
#for online installation
Install.packages("dplyr")
```

For offline installation, first download the required `gz` file from CRAN Repository to a local folder, and then execute the following code:

```
install.packages("path/to/file/dplyr_0.5.0.tar.gz", repos=NULL)
```

The recommendation engine we are going to build is based on the collaborative filtering approach. As explained in `Chapter 1`, *Introduction to Recommendation Engines*, is based on the user's neighbourhood, as explained in the following figure:

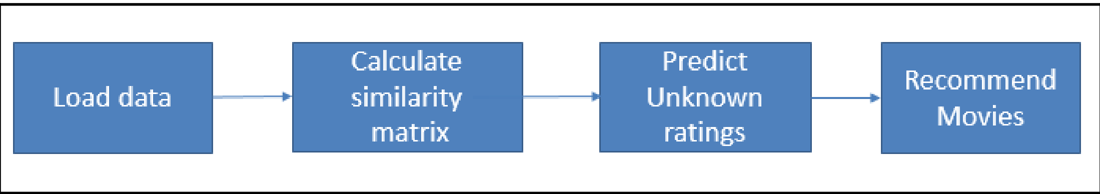


Building our basic recommendation engine

The steps to build our basic recommendation engine are as follows:

1. Loading and formatting data.
2. Calculating similarity between users.
3. Predicting the unknown ratings for users.
4. Recommending items to users based on user-similarity score.

These steps can be seen in the following diagram:



Loading and formatting data

The dataset used for this chapter can be downloaded from `https://raw.githubusercontent.com/sureshgorakala/RecommenderSystems_R/master/movie_rating.csv`.

The dataset chosen for the chapter is a movie-rating dataset containing ratings for six movies given by six users on a scale of 0 to 5:

```
critic,title,rating
Jack Matthews,Lady in the Water,3.0
Jack Matthews,Snakes on a Plane,4.0
Jack Matthews,You Me and Dupree,3.5
Jack Matthews,Superman Returns,5.0
Jack Matthews,The Night Listener,3.0
Mick LaSalle,Lady in the Water,3.0
Mick LaSalle,Snakes on a Plane,4.0
Mick LaSalle,Just My Luck,2.0
Mick LaSalle,Superman Returns,3.0
Mick LaSalle,You Me and Dupree,2.0
Mick LaSalle,The Night Listener,3.0
Claudia Puig,Snakes on a Plane,3.5
Claudia Puig,Just My Luck,3.0
Claudia Puig,You Me and Dupree,2.5
Claudia Puig,Superman Returns,4.0
Claudia Puig,The Night Listener,4.5
Lisa Rose,Lady in the Water,2.5
Lisa Rose,Snakes on a Plane,3.5
Lisa Rose,Just My Luck,3.0
Lisa Rose,Superman Returns,3.5
Lisa Rose,The Night Listener,3.0
Lisa Rose,You Me and Dupree,2.5
Toby,Snakes on a Plane,4.5
Toby,Superman Returns,4.0
Toby,You Me and Dupree,1.0
Gene Seymour,Lady in the Water,3.0
Gene Seymour,Snakes on a Plane,3.5
Gene Seymour,Just My Luck,1.5
Gene Seymour,Superman Returns,5.0
Gene Seymour,You Me and Dupree,3.5
Gene Seymour,The Night Listener,3.0
```

Before we load the data, let me explain a few things about the data. The dataset chosen is a comma-separated file having movie ratings from 1 to 5 in steps of 5 given by six users on six movies. Not all critics have rated all the titles in the dataset.

Our objective is to build a recommendation engine that recommends unknown movies to users based on the ratings of similar users.

Loading the data from a `csv` file in R is given by `read.csv()`:

```
ratings = read.csv("~/movie_rating.csv")
```

The first six rows of the data can be viewed using `head()`, an inbuilt function in R:

```
head(ratings)
```

```
> head(ratings)
          critic               title rating
1 Jack Matthews  Lady in the Water    3.0
2 Jack Matthews  Snakes on a Plane    4.0
3 Jack Matthews  You Me and Dupree    3.5
4 Jack Matthews   Superman Returns    5.0
5 Jack Matthews The Night Listener    3.0
6  Mick LaSalle  Lady in the Water    3.0
```

To see the dimensions of the dataset, we use `dim()`, an inbuilt function in R:

```
dim(ratings)
[1] 31  3
```

To see the structure of the input data, we may use the `str()` function in R, as follows:

```
Str(ratings)
```

```
'data.frame':   31 obs. of  3 variables:
 $ critic: Factor w/ 6 levels "Claudia Puig",..: 3 3 3 3 3 5 5 5 5 5 ...
 $ title : Factor w/ 6 levels "Just My Luck",..: 2 3 6 4 5 2 3 1 4 6 ...
 $ rating: num  3 4 3.5 5 3 3 4 2 3 2 ...
```

We see that we have a dataset containing 31 observations and three variables such as critic, title, and rating. Also, we see that six critics have rated six movies. The ratings are between 1 and 5.

To see the levels of the attributes of a variable, we use `levels()` in R:

```
> levels(ratings$critic)
[1] "Claudia Puig"  "Gene Seymour"  "Jack Matthews" "Lisa Rose"     "Mick LaSalle"  "Toby"
> levels(ratings$title)
[1] "Just My Luck"        "Lady in the Water"   "Snakes on a Plane"   "Superman Returns"    "The Night Listener" "You Me and Dupree"
> sort(unique(ratings$rating),decreasing = F)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

To build a recommender system, we would be requiring a matrix where rows contain users, columns contain items, and the cells contain the ratings given by users to the items.

The next step is to arrange the data in a format that is useful to build the recommendation engine. The current data contains a row containing critic, title, and rating. This has to be converted to matrix format containing critics as rows, title as columns, and ratings as the cell values.

The following code helps us achieve this. We use the `acast()` function available in `reshape2` package. The `reshape2` package is a R package popularly used for restructuring data. The `acast()` function in `reshape2` package casts a data frame to matrix representation.

The `cast` function takes the ratings dataset as input, `title` as row attribute, `critic` as column attribute, and `rating` as value.

```
#data processing and formatting
movie_ratings = as.data.frame(acast(ratings, title~critic,
    value.var="rating"))
```

The transformed data can be viewed as follows:

```
View(movie_ratings)
```

	Claudia Puig	Gene Seymour	Jack Matthews	Lisa Rose	Mick LaSalle	Toby
Just My Luck	3.0	1.5	NA	3.0	2	NA
Lady in the Water	NA	3.0	3.0	2.5	3	NA
Snakes on a Plane	3.5	3.5	4.0	3.5	4	4.5
Superman Returns	4.0	5.0	5.0	3.5	3	4.0
The Night Listener	4.5	3.0	3.0	3.0	3	NA
You Me and Dupree	2.5	3.5	3.5	2.5	2	1.0

From the formatted data, we see that `Toby` has rated three movies. `Lisa Rose`, `Mick LaSalle`, and `Gene Seymour` have rated all the movies. `Claudia Puig`, and `Jack Matthews` have not rated one movie each. Here, let's revisit our objective, which is defined at the beginning of the section: we shall recommend to critics movies that they have not rated, based on similar users. For example, we shall recommend movies to `Toby` based on the ratings provided by other critics similar to `Toby`.

Calculating similarity between users

This is a very important step as we need to recommend the previously unseen movies based on the ratings given to these movies by other similar critics. There are various similarity measures, such as Euclidean distance, cosine distance, Pearson coefficient, Jaccard distance, and so on. The details of these measures or similarity metrics are explained in detail in the `Chapter 4`, *Data Mining Techniques Used in Recommendation Engines*.

In this chapter, we'll use correlation as the similarity measure between users. The reason for choosing correlation is that correlation represents the association two items or how closely two item vectors covary or are related to each other. So for this chapter, we have chosen the correlation value as the measure of similarity between two items in a matrix.

In R, we have the `cor()` function to find correlation between variables in a dataset. The following code calculates the similarity between critics:

While finding similarity between Toby and other critics,use the `use="complete.obs"` attribute, of the `cor()` function to consider complete observations:

```
sim_users = cor(movie_ratings[,1:6],use="complete.obs")
View(sim_users)
```

	Claudia Puig	Gene Seymour	Jack Matthews	Lisa Rose	Mick LaSalle	Toby
Claudia Puig	1.0000000	0.7559289	0.9285714	0.9449112	0.6546537	0.8934051
Gene Seymour	0.7559289	1.0000000	0.9449112	0.5000000	0.0000000	0.3812464
Jack Matthews	0.9285714	0.9449112	1.0000000	0.7559289	0.3273268	0.6628490
Lisa Rose	0.9449112	0.5000000	0.7559289	1.0000000	0.8660254	0.9912407
Mick LaSalle	0.6546537	0.0000000	0.3273268	0.8660254	1.0000000	0.9244735
Toby	0.8934051	0.3812464	0.6628490	0.9912407	0.9244735	1.0000000

From the preceding code, we observe that `Lisa Rose` is very similar to `Toby` with `0.99` and `Mick LaSalle` with `0.92`.

Predicting the unknown ratings for users

In this section, we will predict the unrated movies of Toby using the ratings given by similar users. The following are the steps to achieve this:

1. Extract the titles which Toby has not rated.
2. For these titles, separate all the ratings given by other critics.
3. Multiply the ratings given for these movies by all critics other than Toby with the similarity values of critics with Toby.
4. Sum up the total ratings for each movie, and divide this summed up value with the sum of similarity critic values.

Before we go into the code, let's learn a bit about the `data.table` package and the `setDT()` method we have used in the following code.

`Data.table` is a popular R package that provides an enhanced `data.frame` version, which allows us to do manipulations on data with lightening speed. Another advantage of the `data.table` package is that it can handle very large datasets up to 100 GB data in RAM. Various operations, such as creating a data table, an enhanced version of data frame, sub-setting data, manipulating the data, joins etc.

For this exercise, we have made use of the `setDT()` method available in `data.table`. The `set*` functions in `data.table` help manipulate input data by reference instead of value, that is, while transforming data, there won't be any physical copy of the data.

The preceding explanation is written as code, as follows:

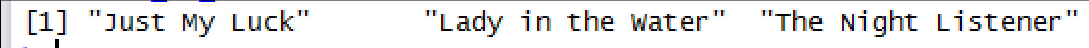
1. Extract the titles which Toby has not rated. We have used the `setDT()` function available in the `data.table` package to extract the non-rated titles and create a `data.table` and `data.frame` object, `rating_critic`. The `setDT()` method extracts column values and corresponding row names and creates a two-dimension `data.frame` or `data.table` object:

```
rating_critic  = setDT(movie_ratings[colnames(movie_ratings)
    [6]],keep.rownames = TRUE)[]
names(rating_critic) = c('title','rating')
View(rating_critic)
```

	title	rating
1	Just My Luck	NA
2	Lady in the Water	NA
3	Snakes on a Plane	4.5
4	Superman Returns	4.0
5	The Night Listener	NA
6	You Me and Dupree	1.0

2. Isolate the non-rated movies from the aforementioned list:

```
titles_na_critic =
    rating_critic$title[is.na(rating_critic$rating)]
titles_na_critic
```



```
[1] "Just My Luck"       "Lady in the Water"   "The Night Listener"
```

Please note that the `is.na()` function is used to filter out NA values.

Take the ratings based on the original dataset and subset all the critics who have rated the aforementioned shown movies.

In the following code, `%in%` acts as the where condition in SQL:

```
ratings_t =ratings[ratings$title %in% titles_na_critic,]
View(ratings_t)
```

critic	title	rating
Jack Matthews	Lady in the Water	3.0
Jack Matthews	The Night Listener	3.0
Mick LaSalle	Lady in the Water	3.0
Mick LaSalle	Just My Luck	2.0
Mick LaSalle	The Night Listener	3.0
Claudia Puig	Just My Luck	3.0
Claudia Puig	The Night Listener	4.5
Lisa Rose	Lady in the Water	2.5
Lisa Rose	Just My Luck	3.0
Lisa Rose	The Night Listener	3.0
Gene Seymour	Lady in the Water	3.0
Gene Seymour	Just My Luck	1.5
Gene Seymour	The Night Listener	3.0

To the aforementioned data frame, now let's add a new variable, `similarity`, using the similarity values of each `critic` w.r.t Toby:

```
x = (setDT(data.frame(sim_users[,6]),keep.rownames = TRUE)[])
names(x) = c('critic','similarity')
ratings_t =  merge(x = ratings_t, y = x, by = "critic", all.x =
    TRUE)
View(ratings_t)
```

critic	title	rating	similarity
Claudia Puig	Just My Luck	3.0	0.8934051
Claudia Puig	The Night Listener	4.5	0.8934051
Gene Seymour	Lady in the Water	3.0	0.3812464
Gene Seymour	Just My Luck	1.5	0.3812464
Gene Seymour	The Night Listener	3.0	0.3812464
Jack Matthews	Lady in the Water	3.0	0.6628490
Jack Matthews	The Night Listener	3.0	0.6628490
Lisa Rose	Lady in the Water	2.5	0.9912407
Lisa Rose	Just My Luck	3.0	0.9912407
Lisa Rose	The Night Listener	3.0	0.9912407
Mick LaSalle	Lady in the Water	3.0	0.9244735
Mick LaSalle	Just My Luck	2.0	0.9244735
Mick LaSalle	The Night Listener	3.0	0.9244735

3. Multiply rating with similarity value, and add the resultant as a new variable, `sim_rating`:

```
ratings_t$sim_rating = ratings_t$rating*ratings_t$similarity
    ratings_t
```

critic	title	rating	similarity	sim_rating
Claudia Puig	Just My Luck	3.0	0.8934051	2.6802154
Claudia Puig	The Night Listener	4.5	0.8934051	4.0203232
Gene Seymour	Lady in the Water	3.0	0.3812464	1.1437393
Gene Seymour	Just My Luck	1.5	0.3812464	0.5718696
Gene Seymour	The Night Listener	3.0	0.3812464	1.1437393
Jack Matthews	Lady in the Water	3.0	0.6628490	1.9885469
Jack Matthews	The Night Listener	3.0	0.6628490	1.9885469
Lisa Rose	Lady in the Water	2.5	0.9912407	2.4781018
Lisa Rose	Just My Luck	3.0	0.9912407	2.9737221
Lisa Rose	The Night Listener	3.0	0.9912407	2.9737221
Mick LaSalle	Lady in the Water	3.0	0.9244735	2.7734204
Mick LaSalle	Just My Luck	2.0	0.9244735	1.8489469
Mick LaSalle	The Night Listener	3.0	0.9244735	2.7734204

4. Sum up all the rating values for each title calculated in the preceding step, and then divide this summed up value for each title with the sum of similarity values of each critic, that is, for the `Just My Luck` title, the rating for `Toby` is calculated by summing up all the `sim_rating` values for `Just My Luck` divided by the sum of similarity values of all the critics who have rated the `Just My Luck` title:

   (2.6802154+0.5718696+2.9737221+1.8489469)/(0.8934051+0.3812464+0.9912407+0.9244735) = 2.530981

   The preceding calculation for all the titles are done in R using two functions available in the `dplyr` package, `group_by()`, and `summarise()`.

The `dplyr` package is an R package used for data manipulations. This package is very useful, like `data.table`; it comes in very handy for exploratory analysis and data manipulation.
The `summarise()` function is available in the `dply` package for summarizing results. The `group_by()` function is used to group data by one or more variables.
The `%>%` operator available in the `dply` package is a very handy function used to group multiple codes together. In the following code, we are using the `%>%` code to group the `group_by()` and `summarise()` functions together and compute the results without writing intermediate results:

```
result = ratings_t %>% group_by(title) %>%
    summarise(sum(sim_rating)/sum(similarity))
result
Source: local data frame [3 x 2]
title sum(sim_rating)/sum(similarity)
(fctr) (dbl)
1 Just My Luck 2.530981
2 Lady in the Water 2.832550
3 The Night Listener 3.347790
```

You can see the calculated or predicted ratings for all the three titles not rated by Toby. Now you can recommend these new titles, the ratings for which are greater than the average ratings given by Toby. For example, the `mean` rating given by Toby to three titles is given by the following code:

```
mean(rating_critic$rating,na.rm = T)
3.166667
```

Now that we know the average rating by Toby is `3.16`, we can recommend movies with ratings greater than the mean values. From the predicted values, we can recommend the movie `The Night Listener`, which is above his mean value.
The aforementioned generating recommendations for all the users can be easily extended by writing a function as follows:

```
generateRecommendations <- function(userId){
rating_critic = setDT(movie_ratings[colnames(movie_ratings)
    [userId]],keep.rownames = TRUE)[]
names(rating_critic) = c('title','rating')
titles_na_critic =
    rating_critic$title[is.na(rating_critic$rating)]
ratings_t =ratings[ratings$title %in% titles_na_critic,]
```

```
#add similarity values for each user as new variable
x = (setDT(data.frame(sim_users[,userId]),keep.rownames = TRUE)
    [])
names(x) = c('critic','similarity')
ratings_t = merge(x = ratings_t, y = x, by = "critic", all.x =
    TRUE)
#mutiply rating with similarity values
ratings_t$sim_rating = ratings_t$rating*ratings_t$similarity
#predicting the non rated titles
result = ratings_t %>% group_by(title) %>%
    summarise(sum(sim_rating)/sum(similarity))
return(result)
}
```

Making predictions now for each of the users will be very easy and is shown next:

```
> generateRecommendations(1)
Source: local data frame [1 x 2]

               title sum(sim_rating)/sum(similarity)
              (fctr)                           (dbl)
1 Lady in the Water                         2.856137
> generateRecommendations(2)
Source: local data frame [0 x 2]

Variables not shown: title (fctr), sum(sim_rating)/sum(similarity) (lgl)
> generateRecommendations(3)
Source: local data frame [1 x 2]

          title sum(sim_rating)/sum(similarity)
         (fctr)                           (dbl)
1 Just My Luck                         2.409926
> generateRecommendations(4)
Source: local data frame [0 x 2]

Variables not shown: title (fctr), sum(sim_rating)/sum(similarity) (lgl)
> generateRecommendations(5)
Source: local data frame [0 x 2]

Variables not shown: title (fctr), sum(sim_rating)/sum(similarity) (lgl)
> generateRecommendations(6)
Source: local data frame [3 x 2]

                title sum(sim_rating)/sum(similarity)
               (fctr)                           (dbl)
1        Just My Luck                         2.530981
2   Lady in the Water                         2.832550
3 The Night Listener                          3.347790
> |
```

Kudos to us for having built our first and basic recommender system. Let's put together the entire code we have done till now. The following is the full version of the code:

```
library(reshape2)
library(data.table)
library(dplyr)
#data loading
ratings = read.csv("C:/Users/Suresh/Desktop/movie_rating.csv")
#data processing and formatting
movie_ratings = as.data.frame(acast(ratings, title~critic,
value.var="rating"))
#similarity calculation
sim_users = cor(movie_ratings[,1:6],use="complete.obs")
#sim_users[colnames(sim_users) == 'Toby']
sim_users[,6]
#predicting the unknown values
#seperating the non rated movies of Toby
rating_critic =
setDT(movie_ratings[colnames(movie_ratings)[6]],keep.rownames = TRUE)[]
names(rating_critic) = c('title','rating')
titles_na_critic = rating_critic$title[is.na(rating_critic$rating)]
ratings_t =ratings[ratings$title %in% titles_na_critic,]
#add similarity values for each user as new variable
x = (setDT(data.frame(sim_users[,6]),keep.rownames = TRUE)[])
names(x) = c('critic','similarity')
ratings_t = merge(x = ratings_t, y = x, by = "critic", all.x = TRUE)
#mutiply rating with similarity values
ratings_t$sim_rating = ratings_t$rating*ratings_t$similarity
#predicting the non rated titles
result = ratings_t %>% group_by(title) %>%
summarise(sum(sim_rating)/sum(similarity))
#function to make recommendations
generateRecommendations <- function(userId){
rating_critic =
setDT(movie_ratings[colnames(movie_ratings)[userId]],keep.rownames =
TRUE)[]
names(rating_critic) = c('title','rating')
titles_na_critic = rating_critic$title[is.na(rating_critic$rating)]
ratings_t =ratings[ratings$title %in% titles_na_critic,]
#add similarity values for each user as new variable
x = (setDT(data.frame(sim_users[,userId]),keep.rownames = TRUE)[])
names(x) = c('critic','similarity')
ratings_t = merge(x = ratings_t, y = x, by = "critic", all.x = TRUE)
#mutiply rating with similarity values
ratings_t$sim_rating = ratings_t$rating*ratings_t$similarity
#predicting the non rated titles
result = ratings_t %>% group_by(title) %>%
summarise(sum(sim_rating)/sum(similarity))
```

```
return(result)
}
```

Summary

Congratulations! We have built a very basic recommendation engine using R. We have seen the step-by-step approach of building a recommendation engine. In the following chapters, we will learn about different types of recommendation engines and their implementations in various technologies such, as Spark, Mahout, Neo4j, R, and Python. In the next chapter, we will learn about the various types of recommendation engines in depth.

3
Recommendation Engines Explained

In `Chapter 2`, *Build Your First Recommendation Engine,* we learned how to build a basic recommender system using R. With introductions to various recommender systems in `Chapter 1`, *Introduction to Recommendation Engines,* we have got a fair idea about what a recommender system is and why a recommender system is important in the current age of data explosion. In this chapter we will learn about various types of recommender systems in detail. This chapter explains Neighborhood similarity-based recommendations, personalized recommendation engines, model-based recommender systems, and hybrid recommendation engines.

The following are the different subtypes of recommender system covered in this chapter:

- Neighborhood-based recommendation engines:
    - User-based collaborative filtering
    - Item-based collaborative filtering
- Personalized recommendation engines:
    - Content-based recommendation engines
    - Context-aware recommendation engines
- Model-based recommendation engines:
    - ML-based recommendation engines
    - Classification – SVM/KNN
    - Matrix Factorization
    - Singular value decomposition
    - Alternating Least Squares
    - Hybrid recommendation engines

Evolution of recommendation engines

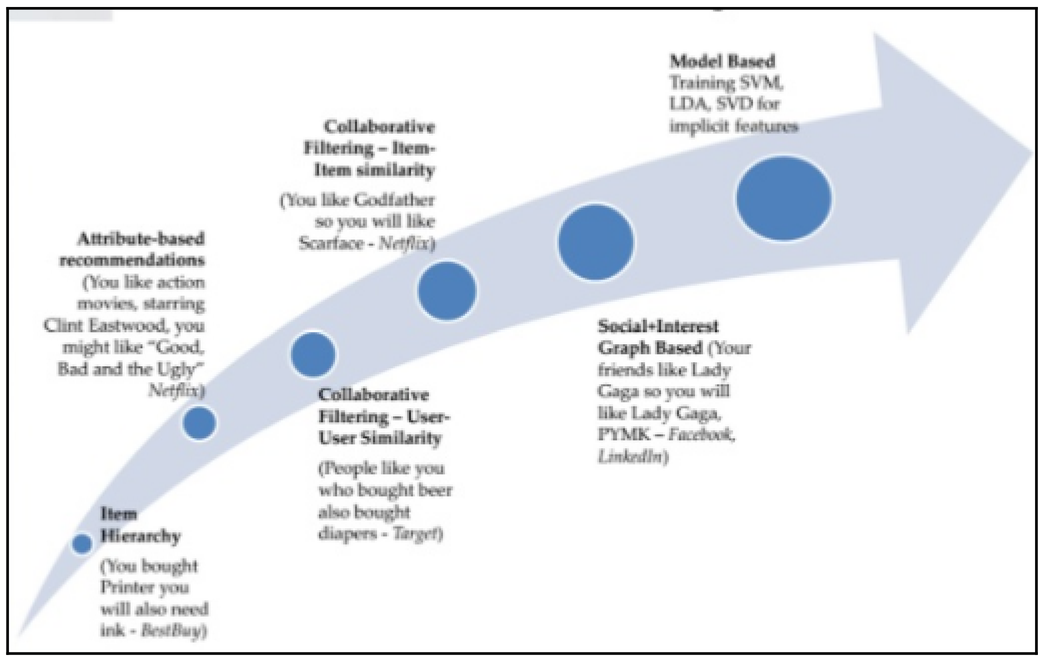
Over the years, recommender systems have evolved, from basic nearest neighborhood methods to personalized recommenders to context-aware recommendations, from batch-mode recommendations to real-time recommendations, from basic heuristic approaches such as similarity calculation to more accurate, complex machine-learning approaches.

In the early stages of these recommender systems, only user ratings on products were used for generating recommendations. At this time, researchers used only the available ratings information. They simply applied heuristic approaches such as similarity calculation using Euclidean distances, the Pearson coefficient, cosine similarity, and so on. These approaches were well received and surprisingly they perform quite well even today.

This first generation of recommendation engines is called collaborative filtering or neighborhood method recommenders. Though they perform very well, these recommenders come with their own set of limitations such as cold-start problems; that is to say, they failed to recommend products to new users with no rating information and recommend new products with no ratings to the users. Also these recommenders failed to handle scenarios where the data is very sparse, so user ratings on products are much less.

In order to overcome these limitations, new approaches have been developed. For example, in order to handle very large user-rating with high data sparsity, mathematical approaches such as Matrix Factorization and singular value decomposition methods have been used.

To handle the cold-start problem, new approaches such as content-based recommendation systems have been developed. These recommender systems opened the door to many more opportunities such as personalized recommenders systems, which enabled them to recommend products to each user on an individual level. In this approach, instead of rating information, user personal preferences and product features are considered.



In the beginning, similarity calculations were used in content-based recommenders, but with advancements in technology and infrastructure more advanced methods such as machine-learning models have replaced the heuristic methods. These new machine models have improved the accuracy of the recommendations.

Though content-based recommenders have solved many of the shortcomings of collaborative filtering, these have their own inherent shortcomings such as serendipity, that is to say not being able to recommend new items outside the user's preference scope, which collaborative filtering can do.

To solve this problem, researchers started combining different recommendation models to come up with hybrid recommendation models, which are much more powerful than any of the individual models.

With personal successful implementations of personalized recommendation engines, people started extending the personalization to other dimensions called contexts, such as the addition of location, time, group, and so on, and changed the set of recommendations with each context.

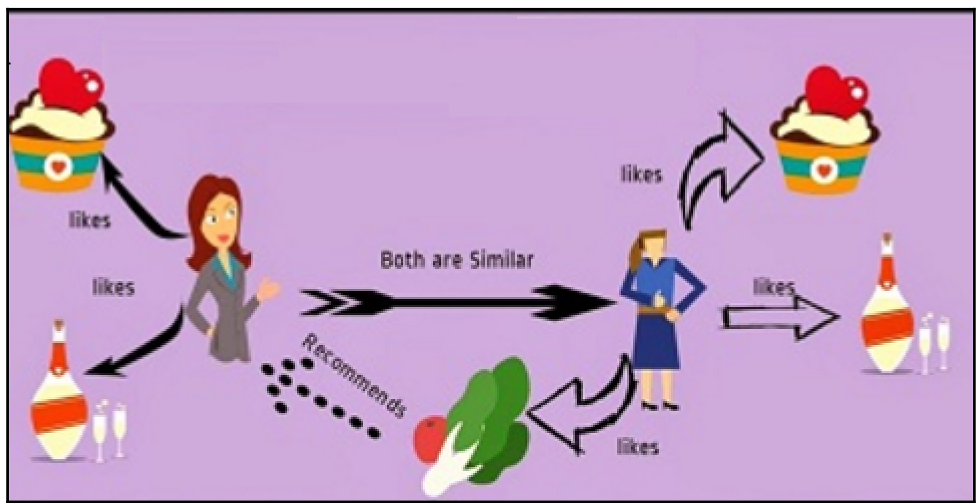
With advancements in technology such as big data ecosystems, in-memory analytic tools such as Apache Spark, and recommendations in real time, the capability of handling very large databases has become possible.

Currently we are moving into more personalization of aspects such as temporal dimension and ubiquitous ways of recommendation.

In the technology aspect the recommendations are moving from machine-learning approaches to more advanced neural network deep-learning approaches.

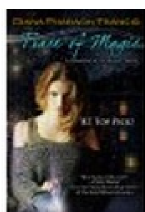
Nearest neighborhood-based recommendation engines

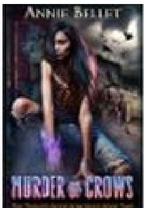
As the name suggests, neighborhood-based recommender systems considers the preferences or likes of the user community or users of the neighborhood of an active user before making suggestions or recommendations to the active user. The idea for neighborhood-based recommenders is very simple: given the ratings of a user, find all the users similar to the active user who had similar preferences in the past and then make predictions regarding all unknown products that the active user has not rated but are being rated in their neighborhood:



While considering the preferences or tastes of neighbors, we first calculate how similar the other users are to the active user and then unrated items from the user community are recommended to the user following predictions. Here the active user is the person to whom the system is serving recommendations. Since similarity calculations are involved, these recommender systems are also called similarity-based recommender systems. Also, since preferences or tastes are considered collaboratively from a pool of users, these recommender systems are also called collaborative filtering recommender systems. In these types of systems, the main actors are the users, products, and user's preference information such as rating/ranking/liking towards the products.

The following image is an example from Amazon showing a neighborhood case:





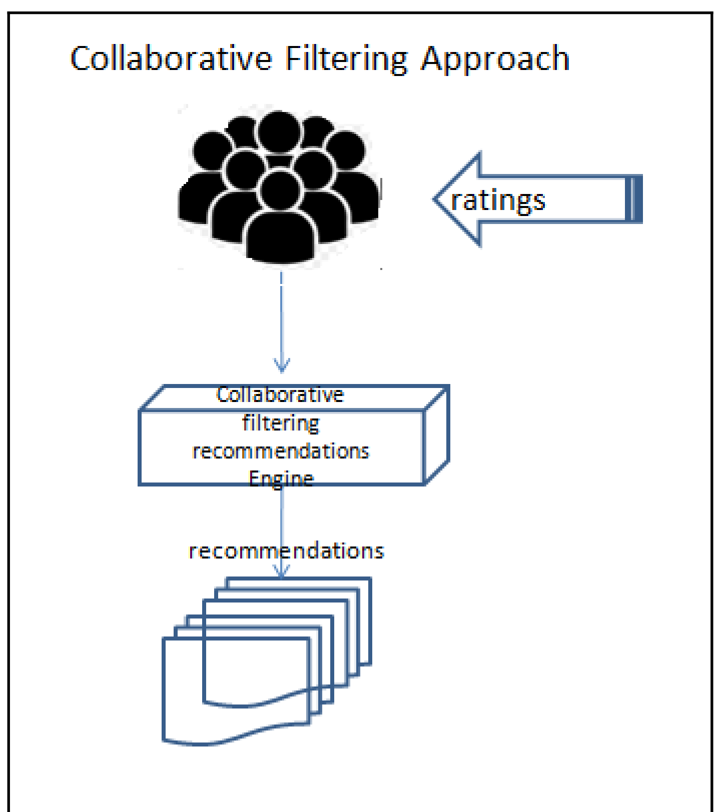
These heuristic-based methods are based on the following assumptions:

- People with similar preferences in the past have similar preferences in the future
- People's preferences will remain stable and consistent in the future

The collaborative filtering systems come in two flavors:

- User-based collaborative filtering
- Item-based collaborative filtering

These neighborhood methods are employed when we have only the users' interaction data of the products, such as ratings, like/unlike, view/not. Unlike content-based recommendations, which will be explained in the next section, they do not consider any features of the products or personal preferences of the user for the products:



User-based collaborative filtering

As previously mentioned, the basic intuition behind user-based collaborative filtering systems is that people with similar tastes in the past will like similar items in future as well. For example, if user A and user B have very similar purchase histories and if user A buys a new book which user B has not yet seen then we can suggest this book to user B as they have similar tastes.

Let us try to understand user-based collaborative filtering with an example:

Problem Statement: Imagine we have a dataset used in `Chapter 2`, *Build Your First Recommendation Engine* containing the reviewers' ratings given to movies on a movie review site. The task at hand is to recommend movies to the reviewers:

Movie/User	Claudia Puig	Gene Seymour	Jack Matthews	Lisa Rose	Mick LaSalle	Toby
Just My Luck	3	1.5		3	2	
Lady in the Water		3	3	2.5	3	
Snakes on a Plane	3.5	3.5	4	3.5	4	4.5
Superman Returns	4	5	5	3.5	3	4
The Night Listener	4.5	3	3	3	3	
You Me and Dupree	2.5	3.5	3.5	2.5	2	1

Before we learn the recommendation approach, the first step is to analyze the data at hand. Let us analyze the data step-by-step as follows:

- A collection of users who have interacted with the application
- A movie catalog of all the available movies
- We have individual users' ratings of movies

Note that each user has not rated all of the movies but only a few movies from the entire catalog.

The first step is to find similar users for an active user and then suggest new movies that this active user has not seen but similar users have seen.

This can be summarized in two steps:

1. Calculate the similarity between users using the rating information of the movies.
2. For each active user, consider the movies that are not rated by them but rated by other users. Predict the unknown ratings for the non-rated movies for the active user.

In the preceding tabular data, let us recommend new movies to our active user, Jack Mathews:

1. The first step is to look for similar users to Jack. We observe by looking at the dataset that Gene Seymour and Mick Lasalle are very similar to Jack Mathews.
2. The similarity between users is calculated based on the ratings given by users. The most common approaches for calculating the similarity are Euclidean distance and the Pearson correlation coefficient.
3. For now we choose Euclidean distance to find the similarity calculation given by the equation that follows:

$$\text{Euclidean Distance}(x,y) = \sqrt{\sum_{i=1}^{n} |x_i - y_i|^2}$$

The intuition behind using Euclidean distance is that we represent users, movies, and ratings as points in a vector space with users on the *x* axis, movies on the *y* axis, and ratings as points in vector space. Now that we have projected our data into vector space, similarity or closeness between two points can be calculated using Euclidean distance, and the Pearson correlation coefficient. The detailed explanation for the similarity measures will be explained in `Chapter 4`, *Data Mining Techniques Used in Recommendation Engines*.

	Claudia Puig	Gene Seymour	Jack Matthews	Lisa Rose	Mick LaSalle	Toby
Claudia Puig	1	0.7559289	0.9285714	0.9449112	0.6546537	0.8934051
Gene Seymour	0.7559289	1	0.9449112	0.5	0	0.3812464
Jack Matthews	0.9285714	0.9449112	1	0.7559289	0.3273268	0.662849
Lisa Rose	0.9449112	0.5	0.7559289	1	0.8660254	0.9912407
Mick LaSalle	0.6546537	0	0.3273268	0.8660254	1	0.9244735
Toby	0.8934051	0.3812464	0.662849	0.9912407	0.9244735	1

Using the previous equation we can calculate the similarity between all the reviewers as shown in the table. We observe from the table that our active user, Toby, is most similar to Lisa Rose.

As a second step we predict the ratings for the unknown movie *Just My Luck* for Jack by calculating the weighted average of the ratings given by other reviewers for *Just My Luck* given in the following:

The rating for Jack for the *Just my Luck* movie is given by the following:

*(3*0.9285+1.5*0.944+3*0.755+2*0.327)/(0.8934051+0.3812464+0.9912407+0.9244735)= 2.23*

In the above equation, we have multiplied the similarity values of all reviewers with Jack by the ratings given by them to the *Just My Luck* movie and then summed all the values. This total sum is divided by the sum of similarity values to normalize the final rating. Similarly we can predict unknown movie ratings for all the reviews and then recommendations can be made.

Item-based collaborative filtering

In item-based collaborative filtering recommender systems, unlike user-based collaborative filtering, we use similarity between items instead of similarity between users. The basic intuition for item-based recommender systems is that if a user liked item A in the past they might like item B, which is similar to item A:

In user-based collaborative filtering, there are a few downs sides:

- The system suffers with performance if the user ratings are very sparse, which is very common in the real world where users will rate only a few items from a large catalog
- The computing cost for calculating the similarity values for all the users is very high if the data is very large
- If user profiles or user inputs change quickly then we have to re-compute the similarity values that come with a high computational cost

Item-based recommendation engines handle these shortcomings by calculating similarity between items or products instead of calculating similarity between users, thereby reducing the computational cost. Since the item catalog doesn't change rapidly, we don't have to re-compute calculations very often.

As with a user-based collaborative filtering approach there are two steps for an item-based collaborative approach:

1. Calculating the similarity between items.
2. Predicting the ratings for the non-rated item for an active user by making use of previous ratings given to other similar items.

The most common similarity measure used for this approach is cosine similarity. Cosine similarity calculates the similarity between two n-dimensional vectors by the angle between them in the vector space. Cosine similarity is given by the following equation:

$$sim(\vec{a}, \vec{b}) = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}| * |\vec{b}|}$$

When applying cosine similarity to recommender systems, we consider the item column as the n-dimensional vector and the similarity between two items as the angle between them. The smaller the angle, the more similar the items.

For example, in the previous dataset, if we want to predict the rating for Toby for the movie *Lady in the Water*, first we have to identify movies similar to *Lady in the Water*. Using the previous cosine equation we can calculate the similarity for all the items. The following table shows the similarity values for all the movies:

Item-based similarity is calculated only for co-rated items.

	Just My Luck	Lady in the Water	Snakes on a Plane	Superman Returns	The Night Listener	You Me and Dupree
Just My Luck	1.0000000	0.6339001	0.7372414	0.7194516	0.8935046	0.7598559
Lady in the Water	0.6339001	1.0000000	0.7950515	0.8149529	0.7977412	0.8897565
Snakes on a Plane	0.7372414	0.7950515	1.0000000	0.9779829	0.8585983	0.9200319
Superman Returns	0.7194516	0.8149529	0.9779829	1.0000000	0.8857221	0.9680784
The Night Listener	0.8935046	0.7977412	0.8585983	0.8857221	1.0000000	0.9412504
You Me and Dupree	0.7598559	0.8897565	0.9200319	0.9680784	0.9412504	1.0000000

From the preceding table, we see that *You, me and Dupree* is the most similar to *Lady in the Water (0.8897565)*.

We now predict the rating for the *Lady in the Water* movie by calculating the weighted sum of ratings assigned to movies similar to *Lady in the Water* by Toby. That is to say, we take the similarity score of *Lady in the Water* for each movie rated by Toby, multiply it by the corresponding rating, and sum up all the scores for all the rated movies. This final sum is divided by the total sum of similarity scores of *Lady in the Water* given as follows:

Rating for *Lady in the Water*:

$(0.795*4.5 + 0.814*4 + 0.889*1)/(0.795+0.814+0.889) = 3.09$

Similarly we can calculate ratings for all other users for movies using the preceding equation. In `Chapter 4`, *Data Mining Techniques Used in Recommendation Engines*, we deal with other similarity metrics that can be used in item-based recommendations.

Advantages

- Easy to implement
- Neither the content information of the products nor the users' profile information is required for building recommendations
- New items are recommended to users giving a surprise factor to the users

Disadvantages

- This approach is computationally expensive as all the user, product, and rating information is loaded into the memory for similarity calculations.
- This approach fails for new users where we do not have any information about the users. This problem is called the cold-start problem.
- This approach performs very poorly if we have little data.
- Since we do not have content information about users or products, we cannot generate recommendations accurately based on rating information only.

Content-based recommender systems

In the previous section, we saw that the recommendations were generated by considering only the rating or interaction information of the products by the users, that is to say that suggesting new items for the active user is based on the ratings given to those new items by similar users to the active user.

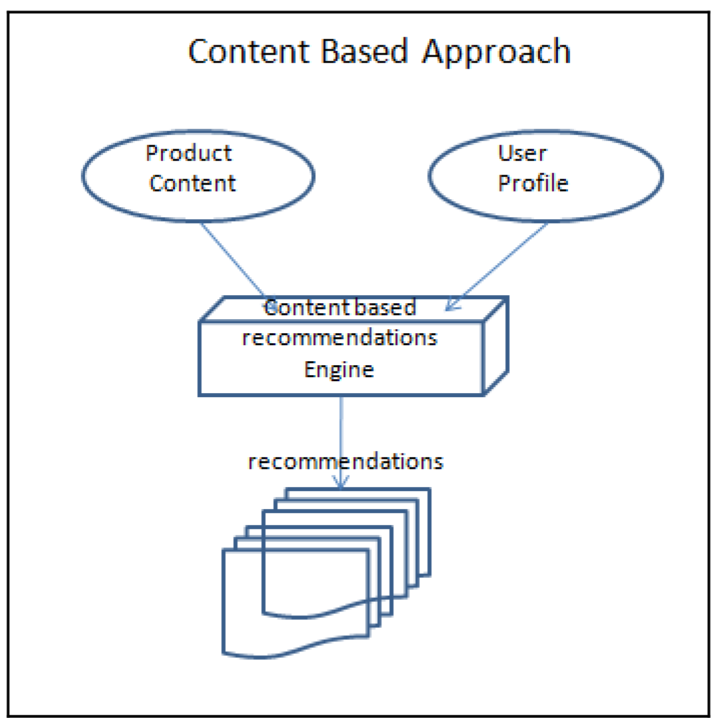
Let's take the case of a person who has given a 4-star rating to a movie. In a collaborative filtering approach we only consider this rating information for generating recommendations. In real life, a person rates a movie based on the features or content of the movie such as its genre, actor, director, story, and screenplay. Also the person watches a movie based on their personal choices. When we are building a recommendation engine to target users at a personal level, the recommendations should not be based on the tastes of other similar people but should be based on the individual users' tastes and the contents of the products.

A recommendation that is targeted at a personalized level and that considers individual preferences and contents of the products for generating recommendations is called a content-based recommender system.

Another motivation for building content-based recommendation engines is that they solve the cold-start problem that new users face in the collaborative filtering approach. When a new user comes, based on the preferences of the person we can suggest new items that are similar to their tastes.

Building content-based recommender systems involves three main steps, as follows:

1. Generating content information for products.
2. Generating a user profile and preferences with respect to the features of the products.
3. Generating recommendations and predicting a list of items that the user might like:



Item profile generation: In this step, we extract the features that represent the product. Most commonly the content of the products is represented in the vector space model with product names as rows and features as columns. Usually the content of the products will either be structured data or unstructured data. Structured data will be obtained from the databases; unstructured features would include the reviews, tags, or textual properties associated in websites. In the item profile generation step, we have to extract relevant features and their relative importance score associated with the product.

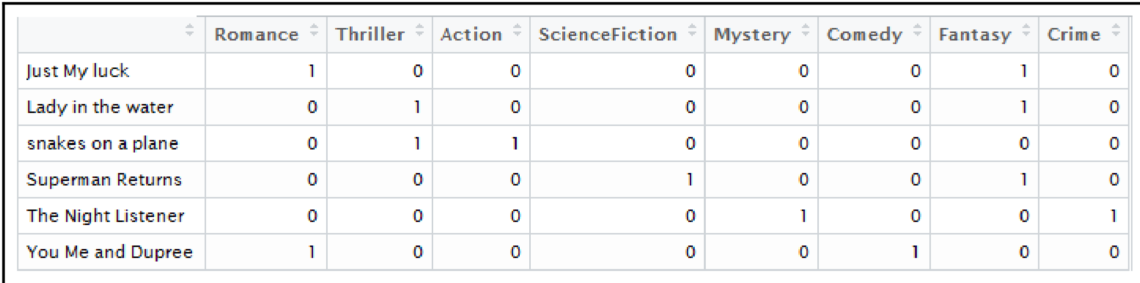
For generating the item profile we use the **term frequency inverse document frequency** (**tf-idf**) for calculating the feature relative importance associated with the item. Since we represent the item features in vector representation, we may use tf-idf, which will be explained in detail in `Chapter 4`, *Data Mining Techniques Used in Recommendation Engines*.

Let us try to better understand with an example. As we've already mentioned, for content-based recommendation engines, we require additional content information about Movies, as follows:

Movies	Genre
Just My luck	Romance
Lady in the water	Thriller
snakes on a plane	Action
Superman Returns	ScienceFiction
The Night Listener	Mystery
You Me and Dupree	Comedy

The first thing we have to do is to create an item profile using tf-idf, by means of the following steps:

Create a term frequency matrix containing the frequency count of each term in each document; that is to say, in our case, the presence of each genre in each movie. The number 1 represents the presence of the genre and 0 represents the absence of the genre:



	Romance	Thriller	Action	ScienceFiction	Mystery	Comedy	Fantasy	Crime
Just My luck	1	0	0	0	0	0	1	0
Lady in the water	0	1	0	0	0	0	1	0
snakes on a plane	0	1	1	0	0	0	0	0
Superman Returns	0	0	0	1	0	0	1	0
The Night Listener	0	0	0	0	1	0	0	1
You Me and Dupree	1	0	0	0	0	1	0	0

The next step is to create inverse document frequency given by the following formula:

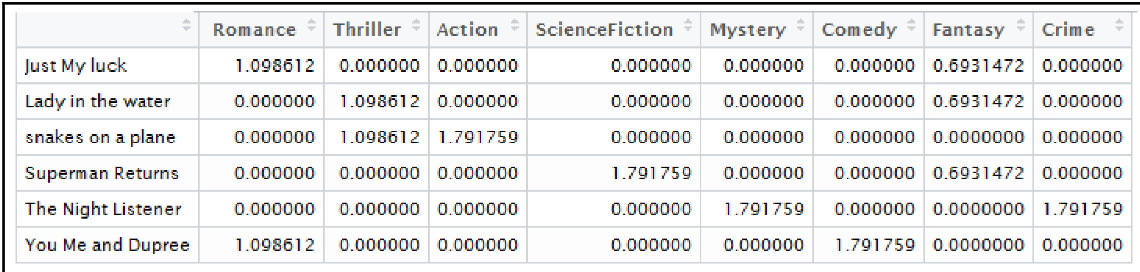
Idf = Log(total number of documents/document frequency)

Here, the total number of documents is the number of movies, and the document frequency is the total number of times they have occurred in all the documents:

Romance	Thriller	Action	ScienceFiction	Mystery	Comedy	Fantasy	Crime
1.0986123	1.0986123	1.7917595	1.7917595	1.7917595	1.7917595	0.6931472	1.7917595

The final step is to create a *tf-idf* matrix given by the following formula:

*tf*idf*

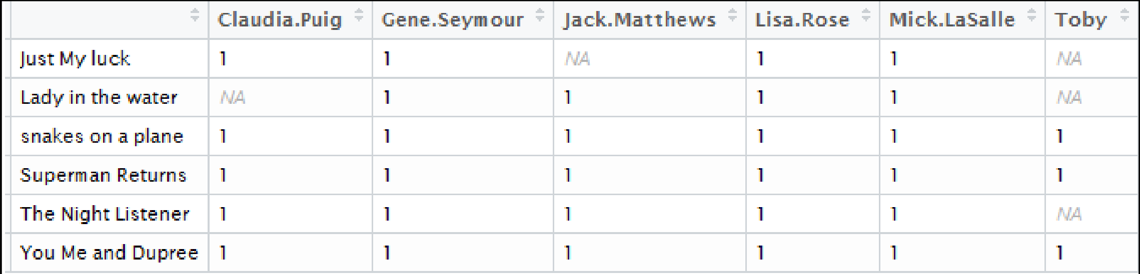


	Romance	Thriller	Action	ScienceFiction	Mystery	Comedy	Fantasy	Crime
Just My luck	1.098612	0.000000	0.000000	0.000000	0.000000	0.000000	0.6931472	0.000000
Lady in the water	0.000000	1.098612	0.000000	0.000000	0.000000	0.000000	0.6931472	0.000000
snakes on a plane	0.000000	1.098612	1.791759	0.000000	0.000000	0.000000	0.0000000	0.000000
Superman Returns	0.000000	0.000000	0.000000	1.791759	0.000000	0.000000	0.6931472	0.000000
The Night Listener	0.000000	0.000000	0.000000	0.000000	1.791759	0.000000	0.0000000	1.791759
You Me and Dupree	1.098612	0.000000	0.000000	0.000000	0.000000	1.791759	0.0000000	0.000000

User profile generation

In this step, we build the user profile or preference matrix matching the product content. In general we build the user profile or features that are in common with the product content as it makes more sense to compare both user and item profiles and calculate the similarity between them.

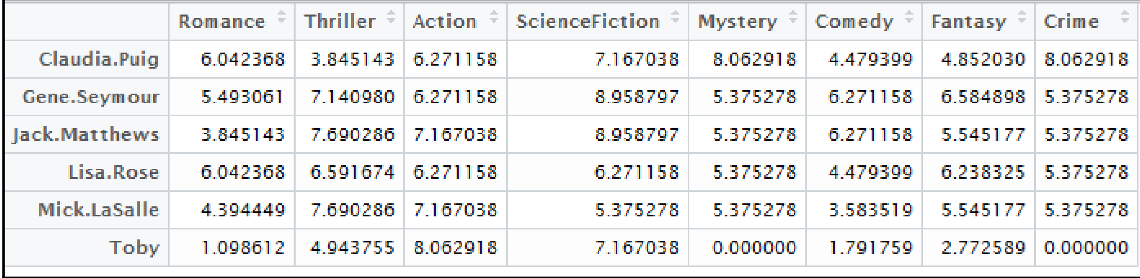
Let's consider the following dataset showing the viewed history of each user. If there is a value of 1 in the matrix cell, it means that the user has seen the movie. This information gives us their preference of movies:



	Claudia.Puig	Gene.Seymour	Jack.Matthews	Lisa.Rose	Mick.LaSalle	Toby
Just My luck	1	1	NA	1	1	NA
Lady in the water	NA	1	1	1	1	NA
snakes on a plane	1	1	1	1	1	1
Superman Returns	1	1	1	1	1	1
The Night Listener	1	1	1	1	1	NA
You Me and Dupree	1	1	1	1	1	1

From the preceding information, we will create a user profile that can be used to compare with the item profile; that is to say, we now create a user profile that contains the user preference of the item features, to genres, in our case. Dot product between the tf-idf and user preference matrix will give the user affinity for each of the genres, as shown in the following table:

dotProduct(Tf-idf, userPreference matrix)



	Romance	Thriller	Action	ScienceFiction	Mystery	Comedy	Fantasy	Crime
Claudia.Puig	6.042368	3.845143	6.271158	7.167038	8.062918	4.479399	4.852030	8.062918
Gene.Seymour	5.493061	7.140980	6.271158	8.958797	5.375278	6.271158	6.584898	5.375278
Jack.Matthews	3.845143	7.690286	7.167038	8.958797	5.375278	6.271158	5.545177	5.375278
Lisa.Rose	6.042368	6.591674	6.271158	6.271158	5.375278	4.479399	6.238325	5.375278
Mick.LaSalle	4.394449	7.690286	7.167038	5.375278	5.375278	3.583519	5.545177	5.375278
Toby	1.098612	4.943755	8.062918	7.167038	0.000000	1.791759	2.772589	0.000000

With user profiles and item profiles at hand, the next step would be to estimate the degree to which the user will prefer each of the items. We can now use cosine similarity to compute the user preference for each of the items. In our example, the cosine similarity between user and item profiles gives the following results:

cosineSimilarity(userProfile,ItemProfile)

	Just My luck	Lady in the water	snakes on a plane	Superman Returns	The Night Listener	You Me and Dupree
Claudia.Puig	0.8919446	0.8826889	0.8057865	0.8173293	0.7461213	0.7964116
Gene.Seymour	0.9478958	0.9442628	0.8729500	0.8891199	0.8219716	0.8696591
Jack.Matthews	0.8879721	0.9029502	0.8005526	0.8502198	0.7538020	0.8210059
Lisa.Rose	0.9478958	0.9442628	0.8729500	0.8891199	0.8219716	0.8696591
Mick.LaSalle	0.9478958	0.9442628	0.8729500	0.8891199	0.8219716	0.8696591
Toby	0.6232739	0.6408335	0.5219196	0.5785800	0.4712180	0.5430630

From the preceding table we can conclude that the greater the cosine angle the more likely the user is to like a movie and hence it can be recommended to the user.

Now that we have made the recommendations, let us take a step back from gathering user preference data. Usually there are two ways of capturing user data; these are as follows:

- Explicitly ask the user for their preferences regarding the product's features, and store them.
- Implicitly capture the user interaction data on products such as browsing history, rating history, and purchase history, and build the user preferences to the product features. In `Chapter 4`, *Data Mining Techniques Used in Recommendation Engines*, and `Chapter 5`, *Building Collaborative Filtering Recommendation Engines*, we build recommendation engines using explicit and implicit user activity examples.

The approach that we have followed until now for building a content-based recommendation system is based on similarity calculation. We may also apply supervised machine-learning approaches such as classification for predicting the most probable products the user might like.

Recommender systems using machine learning or any other mathematical, statistical models to generate recommendations are called model-based systems. In classification-based approaches which fall under model-based recommender systems, first we build a machine-learning model by using a user profile and item profile to predict if a user likes/dislikes an item. Supervised classification tasks such as logistic regression, KNN-classification methods, probabilistic methods, and so on, can be used. Model-based recommendation engines are discussed in the next section.

Advantages

- Content-based recommender systems target at an individual level
- Recommendations are generated using the user preferences alone rather than the user community as with collaborative filtering
- These approaches can be employed in real time as the recommendation model doesn't need to load all the data for processing or generating recommendations
- Accuracy is high compared to collaborative approaches as they deal with the content of the products instead of rating information alone
- The cold-start problem can be easily handled

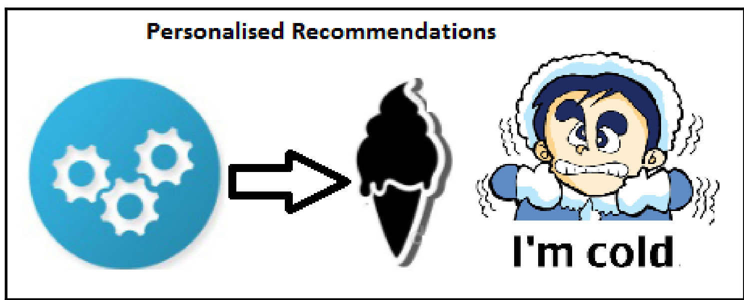
Disadvantages

- As the system is more personalized the generated recommendations will become narrowed down to only user preferences when more user information comes into the system
- As a result, no new products that are not related to the user preferences will be shown to the user
- The user will not be able to look at what is happening around them or what's trending

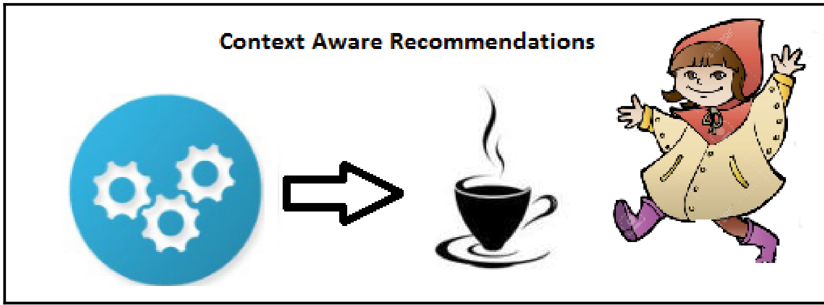
Context-aware recommender systems

Over the years there has been an evolution in recommender systems from neighborhood approaches to personalized recommender systems that are targeted to individual users. These personalized recommender systems have become a huge success as this is useful at end user level, and for organizations these systems become catalysts to increase their business.

Though the personalized recommender systems were targeted at the individual user level and provided recommendations based on the personal preferences of the users, there was scope to refine the systems. For example, the same person in different places might have different requirements. Likewise, the same person has different requirements at different times:



Our intelligent recommender systems should be evolved enough to cater to the needs of the users for different places, at different times. The recommender system should be robust enough to suggest cotton shirts to a person during summer and leather jackets during winter. Similarly, based on the time of day, suggesting good restaurants serving a person's personal choice of breakfast or dinner would be very helpful. These kinds of recommender systems that consider location, time, mood, and so on, that defines the context of the user and suggests personalized recommendations, are called **context aware recommender systems**:



The preceding image illustrates a recommendation engine suggesting coffee in cold weather.

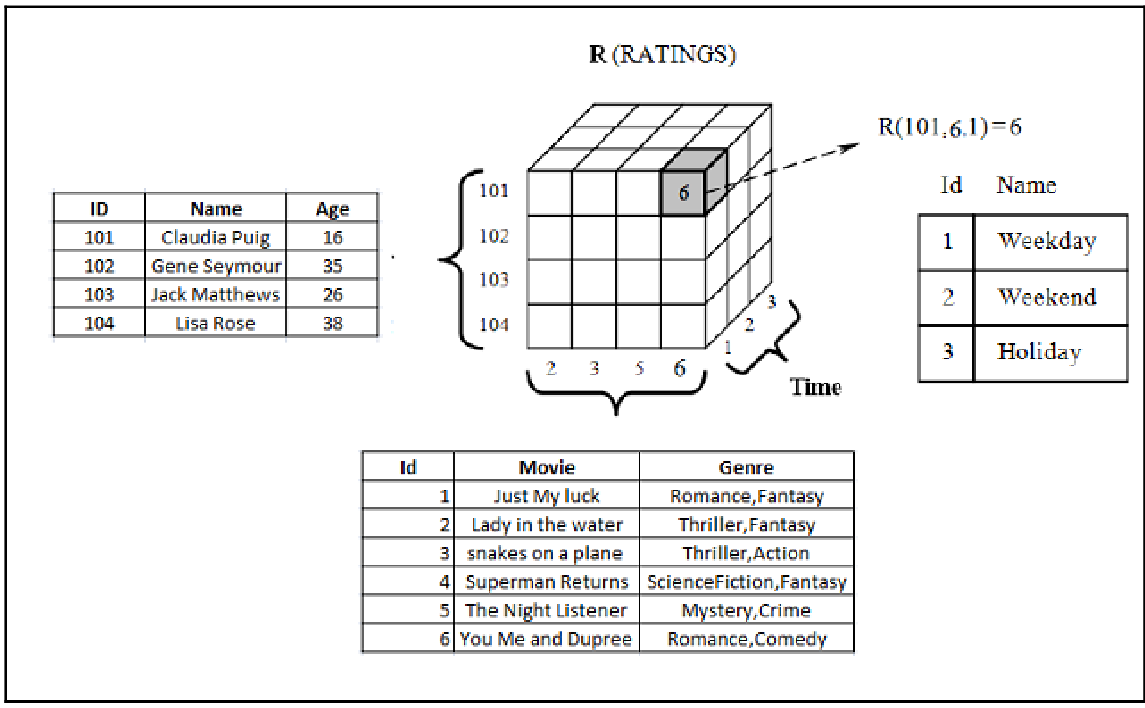
Context definition

So what exactly is context? In general, context represents the present state of the user. The context of a user can be anything such as place, time, day, season, mood, device, whether the user is alone, at the office, on vacation, with family, with friends, life events, and so on. Since people will have different needs in different contexts, the recommendation systems can capture the context information of the user and refine their suggestions accordingly.

For example, a travel vacation system may consider season, place, and time as context for refining the suggestions; an e-commerce website can consider the life event and user purchases for context aware recommendations, and a food website may consider time of day and place information while recommending restaurants.

How are context aware systems designed? Until now we have considered recommendations as a two-dimensional problem, that is to say user preferences and item representations. With the inclusion of context as a new dimension we can build context aware recommendations as a three-dimensional problem:

Recommendations = User x Item x Context



ID	Name	Age
101	Claudia Puig	16
102	Gene Seymour	35
103	Jack Matthews	26
104	Lisa Rose	38

Id	Name
1	Weekday
2	Weekend
3	Holiday

Id	Movie	Genre
1	Just My luck	Romance,Fantasy
2	Lady in the water	Thriller,Fantasy
3	snakes on a plane	Thriller,Action
4	Superman Returns	ScienceFiction,Fantasy
5	The Night Listener	Mystery,Crime
6	You Me and Dupree	Romance,Comedy

Let us revisit the same example as we did in the content-based recommendations, where we considered the user profile and the item profile to generate the user ratings for each of the items based on user preferences by computing the similarity between the user profile and the item profile. Now, in context aware systems, we include context to generate the rankings for items with respect to user preference and context.

For example, we can assume our recommendation has captured the movie watching patterns of the user for a weekday, weekend, and holiday. From this context information, we extract the affinity of each user for the movie contents. For example, consider the following preferences for TOBY for each of the contexts for the movie contents:

	Romance	Thriller	Action	ScienceFi	Mystery	Comedy	Fantasy	Crime
weekday	0.3	0	0.2	0	0	0.5	0	0
weekend	0.4	0	0	0.3	0	0	0.3	0
Holiday	0	0.5	0	0.4	0	0	0.1	0

Preferences of TOBY	1.09	4.94	8.06	7.16	0	1.79	2.77	0

Let us first create a user profile for TOBY for each of the contexts for all the movie content. A dot product between the context matrix and user profile matrix gives us the user profile for all the contexts:

Dotproduct(user profile, context matrix) for TOBY:

		Romance	Thriller	Action	ScienceFi	Mystery	Comedy	Fantasy	Crime
TOBY CONTEXT	weekday	0.32958	0	0.35835	0	0	0.89588	0	0
	weekend	0.43944	0	0	0.53753	0	0	0.20794	0
	Holiday	0	0.54931	0	0.7167	0	0	0.06931	0

We have now calculated the preference of TOBY for each context for the movie context. The next step would be to calculate the ranking of each movie for TOBY for all the contexts.

Cosine similarity (contextual movie content preference matrix, item profile):

	Just My luck	Lady in the water	snakes on a plane	Superman Returns	The Night Listener	You Me and Dupree
weekday	0.27337511	0.000000	0.2996171	0.0000000	0	0.918004
weekend	0.66588719	0.153096	0.0000000	0.7952170	0	0.316934
Holiday	0.04083948	0.553805	0.3170418	0.7656785	0	0.000000

Now we have the context level ranking for movies for TOBY, we can suggest movies based on the context.

From the preceding example, we understood that context-aware recommender systems are content-based recommenders with the inclusion of a new dimension called context. In context-aware systems, recommendations are generated in two steps, as follows:

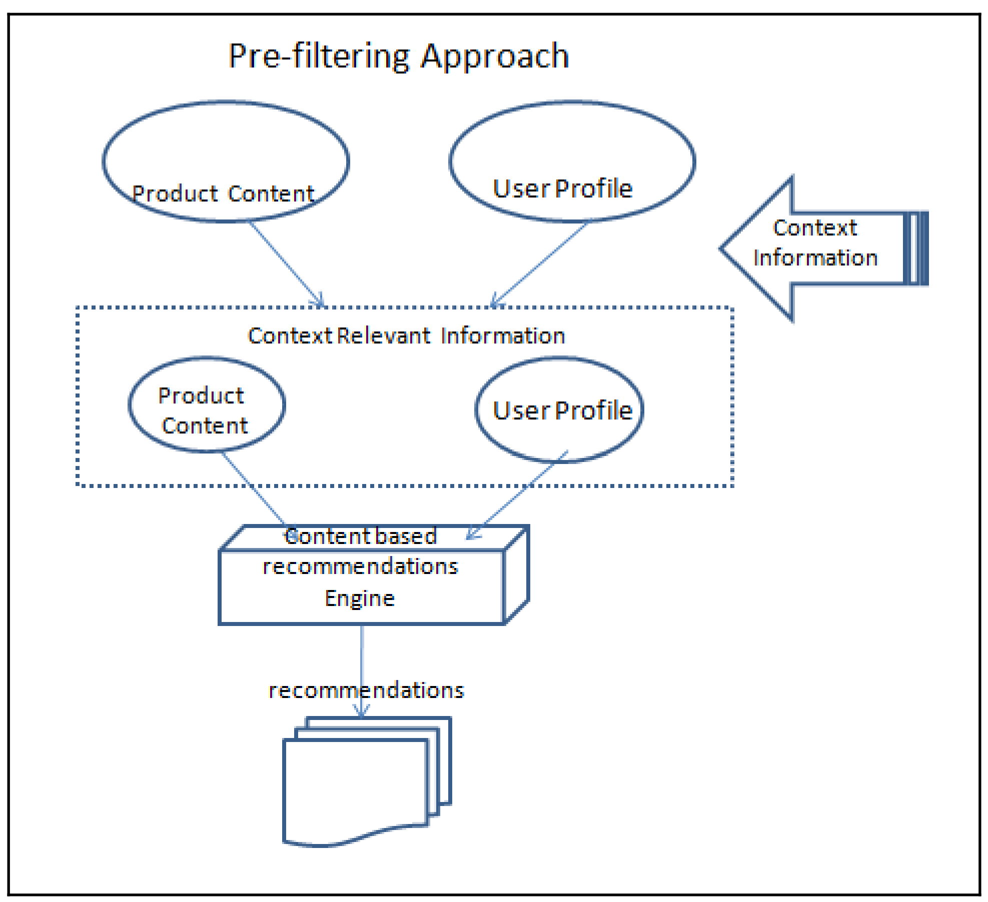
1. Generate a list of recommendations of the products for each user based on the user's preferences; that is, content-based recommendations.
2. Filter out the recommendations that are specific to a current context.

The most common approaches for building context-aware recommender systems are as follows:

- Post-filtering approaches
- Pre-filtering approaches

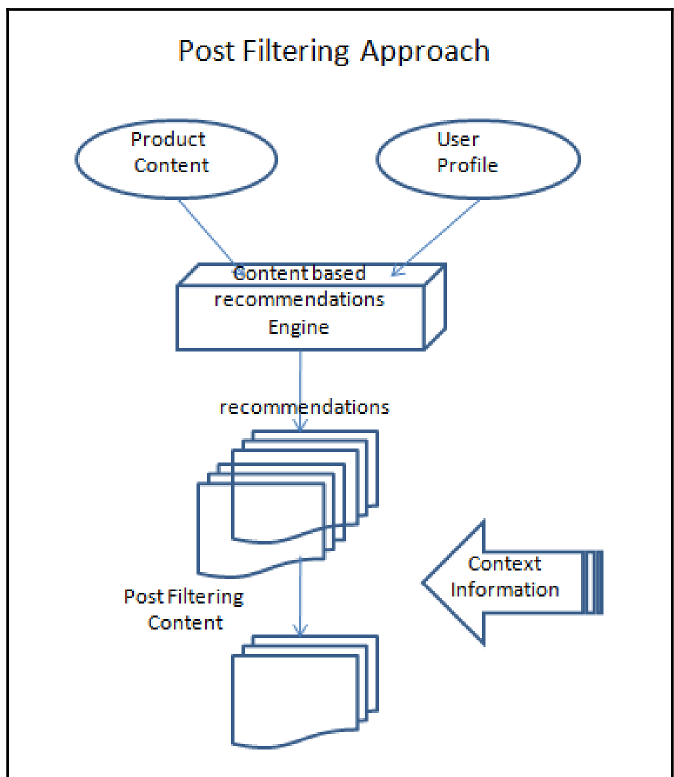
Pre-filtering approaches

In the pre-filtering approach, context information is applied to the user profile and product content. This step will filter out all the non-relevant features, and final personalized recommendations are generated on the remaining feature set. Since the filtering of features is performed before generating personalized recommendations, these are called pre-filtering approaches:



Post-filtering approaches

In post-filtering approaches, firstly personalized recommendations are generated based on the user profile and the product catalogue, then the context information is applied to filter out the relevant products to the user for the current context:



Advantages

- Context aware systems are much more advanced than the personalized content-based recommenders as these systems will be constantly in sync with user movements and generate recommendations as per the current context
- These systems have more of a real-time nature

Disadvantages

- Serendipity or surprise factor, as with other personalized recommenders, will be missing in these types of recommendation as well

Hybrid recommender systems

Collaborative filtering systems and content-based recommender systems are effective and cater to a wide range of needs. They have quite successful implementations but each independently has its own limitations. Research has started moving in the direction of combining both collaborative filtering and content-based recommendations. This new type of recommender system formed by combining collaborative filtering with content-based methods is called a hybrid recommender system.

The choice of combining the different recommendation approaches is up to the researcher or the person implementing the hybrid recommendation engine based on the problem statement and business needs.

The most common approaches followed for building a hybrid system are as follows:

- Weighted method
- Mixed method
- Switching method
- Cascade method
- Feature combination method
- Feature augmentation
- Meta-level

Weighted method

In this method the final recommendations would be the combination, mostly linear, of recommendation results of all the available recommendation engines. At the beginning of the deployment of this weighted hybrid recommendation engine, equal weights will be given to each of the results from available recommendation engines, and gradually the weights will be adjusted by evaluating the responses from the users to recommendations.

Mixed method

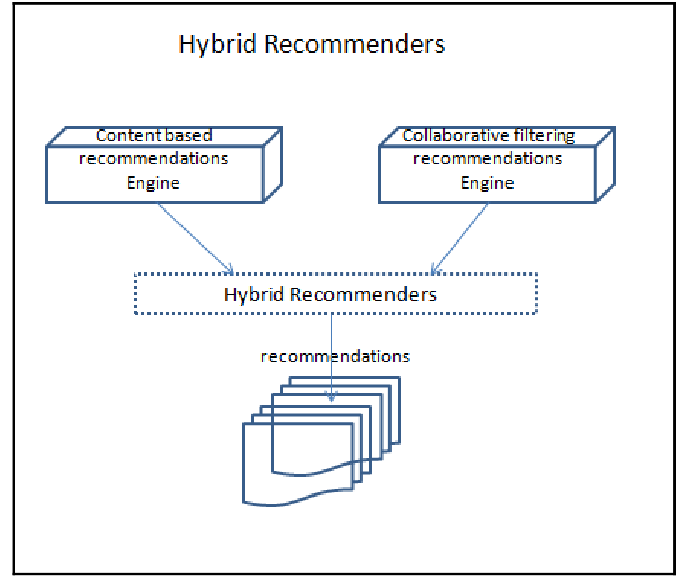
The mixed method is applicable in places where we can mix results from all the available recommenders. These are mostly employed in places where it is not feasible to achieve a score for a product by all the available recommender systems because of data sparsity. Hence recommendations are generated independently and are mixed before being sent to the user.

Cascade method

In this approach, recommendations are generated using collaborative filtering. The content-based recommendation technique is applied and then final recommendations / a ranked list will be given as the output.

Feature combination method

The feature combination method, in which we combine the features of different recommender systems and final recommendation approach, is applied on the combined feature sets. In this technique, we combine both User-Item preference features extracted from content based recommender systems and, User-Item ratings information, and consider a new strategy to build Hybrid recommender systems.



Advantages

- Issues such as the cold-start problem and data sparsity can be handled
- These systems are much more robust and scalable than any of the individual models
- A combination of methods leads to an improvement in accuracy

Model-based recommender systems

Till now we have been focusing on neighborhood approaches which involve similarity calculations between users or products for collaborative filtering approaches or represent the user and item contents in a vector space model, and find similarity measures to identify items similar to the preferences of the users. The main objective of the similarity-based approaches is to calculate the weights of the preferences of users for the products or product content and then use these feature weights for recommending items.

These approaches have been very successful over the years and even today. But these approaches have their own limitations. Since entire data has to be loaded into the environment for similarity calculations, these approaches were also known as memory-based models. These memory-based models are very slow to respond in real-time scenarios when the amount of data is very large as all the data has to be loaded. Another limitation is that the weights calculated are not learned automatically as with machine-learning applications. The cold-start problem is another common limitation that memory-based or neighborhood-based methods suffer from.

In order to address these limitations, researchers started to apply more advanced methods to improve the performance of the recommendation engines such as probabilistic models, machine-learning models such as supervised and unsupervised models, and matrix approaches such as Matrix Factorization and single value decomposition. In the model-based approaches, using available historical data, a model is built with weights learned automatically. New predictions regarding the products will be made using the learned weights and then the final results ranked in a specific order before making recommendations.

Probabilistic approaches

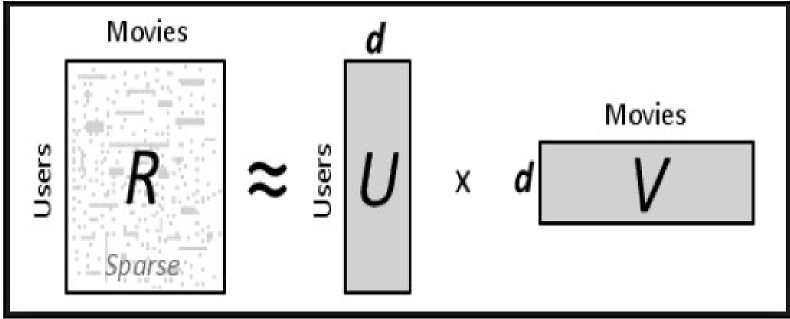
In a probabilistic approach, we build a probability model using the prior probabilities from the available data, and a ranked list of recommendations is generated by calculating the probability of liking/disliking of a product for each user. Most commonly the Naïve Bayes method is used in probabilistic approaches, which is a simple but powerful technique.

Machine learning approaches

As explained in content-based recommender systems, we can consider a recommendation problem as a machine-learning problem. Using historical user and product data, we can extract features and output classes and then build a machine learning model. A final ranked list of product recommendations is generated using the generated model. Many machine-learning approaches such as logistic regression, KNN classification, decision trees, SVM, clustering, and so on, can be used. These machine-learning approaches are applied for collaborative, content based, context aware, and hybrid recommender systems. In `Chapter 4`, *Data Mining Techniques Used in Recommendation Engines*, we learn in detail about each of the machine-learning approaches.

Mathematical approaches

In these approaches, we assume that the ratings or interaction information of users on products are simple matrices. On these matrices we apply mathematical approaches to predict the missing ratings for the users. The most commonly used approaches are the Matrix Factorization model and single valued decomposition models:



By applying matrix decomposition approaches we assume that we decompose the original rating matrix (R) into two new matrices (U, V) that represent the latent features of the users and movies.

In mathematical terms, we can decompose a matrix into two low rank matrices. In the preceding example, matrix R is decomposed into matrices U and V. Now when we multiply back U and V we get the original matrix R approximately. This concept is used in recommendation engines for filling up the unknown ratings in the original rating matrix, and recommendations are then ranked and suggested to the users.

In `Chapter 4`, *Data Mining Techniques Used in Recommendation Engines* we discuss these two approaches in more detail.

Advantages

- Model-based recommendations are much more accurate than the heuristic-based approaches such as neighborhood methods
- In heuristic methods the weights of products / product content is more static, whereas in model-based recommendations, the weights are established through auto-learning
- The model-based approach extracts many unseen patterns using data-driven approaches

Summary

In this chapter, we have learned about popular recommendation engine techniques such as collaborative filtering, content-based recommendations, context aware systems, hybrid recommendations, and model-based recommendation systems, with their advantages and disadvantages. There are different similarity methods such as cosine similarity, Euclidean distance, and the Pearson coefficient. Subcategories within each of the recommendations are also explained.

In the next chapter, we learn about different data-mining techniques such as Neighborhood methods, machine learning methods used in recommendation engines, and their evaluation techniques such as RMSE and, Precision-Recall.

4

Data Mining Techniques Used in Recommendation Engines

Data mining techniques lie at the heart of recommendation engines. These data mining techniques help us extract patterns, group users, calculate similarities, predict preferences, handle sparse input data, evaluate recommendation models, and so on. In the previous chapter, we have learned about recommendation engines in detail. Though we did not get into the implementation of recommendation engines, we learned the theory behind the different types of recommendations engines, such as neighborhood-based, personalized, contextual recommenders, hybrids, and so on. In this chapter, we shall look into the popular data mining techniques currently used in building recommendation engines. The reason for dedicating a separate chapter to this is that we will come across many techniques while implementing recommendation engines in the subsequent chapters.

This chapter is broadly divided into the following sections:

- Neighbourhood-based techniques
    - Euclidean distance
    - Cosine similarity
    - Jaccard similarity
    - Pearson correlation coefficient
- Mathematical modelling techniques
    - Matrix factorization
    - Alternating Least Squares
    - Singular value decomposition

- Machine learning techniques
    - Linear regression
    - Classification models
- Clustering techniques
    - K-means clustering
- Dimensionality reduction
    - Principal component analysis
- Vector space models
    - Term frequency
    - Term frequency-inverse document frequency
- Evaluation techniques
    - Root-mean-square error
    - Mean absolute error
    - Precision and recall

Each section is explained with the basic technique and its implementation in R.

Let's start refreshing the basics that are mostly commonly used in recommendation engines.

Neighbourhood-based techniques

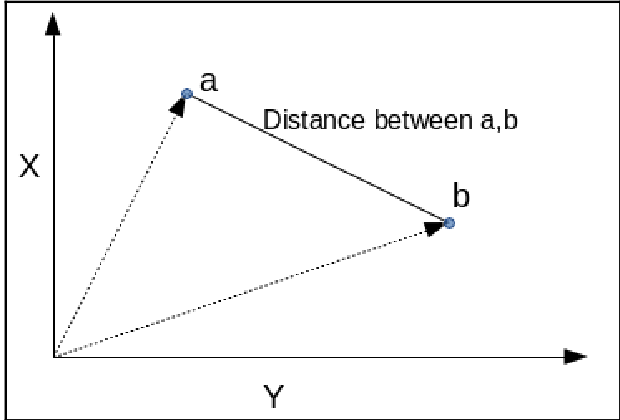
As introduced in the previous chapters, the neighbourhood methods are very simple techniques, which are used right from the beginning of building recommendation engines. These are the oldest yet most widely used approaches, even today. The popularity of these widely used approaches is because of their accuracy in generating recommendations. We know that almost every recommender system works on the concept of similarity between items or users. These neighbourhood methods consider the available information between two users or items as two vectors, and simple mathematic calculation is applied between these two vectors to see how close they are. In this section, we will discuss the following neighbourhood techniques:

- Euclidean distance
- Cosine similarity
- Jaccard Similarity
- Pearson correlation coefficient

Euclidean distance

Euclidean distance similarity is one of the most common similarity measures used to calculate the distance between two points or two vectors. It is the path distance between two points or vectors in vector space.

In the following diagram, we see the path distance between the two vectors, a and b, as Euclidean distance:



Euclidean distance is based on the Pythagoras's theorem to calculate the distance between two points.

The **Euclidean distance** between two points or objects (point *x* and point *y*) in a dataset is defined by the following equation:

$$\text{Euclidean Distance}(x, y) = \sqrt{\sum_{i=1}^{n} |x_i - y_i|^2}\ .$$

Here, *x* and *y* are two consecutive data points, and n is the number of attributes for the dataset.

How is Euclidean distance applied in recommendation engines?

Consider a rating matrix containing user IDs as rows, item IDs as columns, and the preference values as cell values. The Euclidean distance between two rows gives us the user similarity, and the Euclidean distance between two columns gives us the item similarity. This measure is used when the data is comprised of continuous values.

The R script for calculating the Euclidean distance is as follows:

```
x1 <- rnorm(30)
x2 <- rnorm(30)
Euc_dist = dist(rbind(x1,x2) ,method="euclidean")
```

```
> x1
 [1]  0.7824548 -0.2623895  0.5276719  1.2552186 -0.9803315 -0.4561338  2.4051567
 [8]  0.6858002 -0.8711695 -0.2618928  0.2973917  0.8448787  0.2188954 -1.2323462
[15]  0.9133412 -0.4238214 -0.5814376 -0.2448999  1.1896259 -0.9937443  0.6576142
[22] -0.1357882 -0.5627333 -0.8575745  0.2385076  0.7217603 -1.7579127 -0.7489078
[29] -0.3605539 -0.7173789
> x2
 [1]  0.36918961 -0.85669259 -0.66356226  0.70927104 -0.24235742  0.68548041  0.97911641
 [8]  0.19732953 -0.83348519  0.38272366 -1.61543924  2.31314283  1.44765481 -0.77416639
[15]  1.20584033 -0.94992148 -0.73585753 -1.32329554  0.10810163 -0.62878243  1.22097185
[22]  0.33721922 -0.03807742 -0.55773028  0.68864984  1.26823921 -0.94928127  0.88784091
[29]  0.81162258  1.37679405
> Euc_dist = dist(rbind(x1,x2),method = "euclidean")
> Euc_dist
         x1
x2 5.259711
> |
```

Cosine similarity

Cosine similarity is a measure of similarity between two vectors of an inner product space that measures the cosine of the angle between them; it's given by the following equation:

$$\text{similarity} = \cos(\theta) = \frac{A \cdot B}{\|A\| \|B\|} :$$

Let *a* be a vector (*a1*, *a2*, *a3*, *a4*) and *b* be another vector (*b1*, *b2*, *b3*, *b4*). The dot product between these vectors *a* and *b* is as follows:

a.b = a1b1 + a2b2 + a3b3 + a4b4

The resultant will be a single value, a scalar constant. What does it mean to take the dot product between two vectors? To answer this question, let's define the geometric definition of dot product between two vectors:

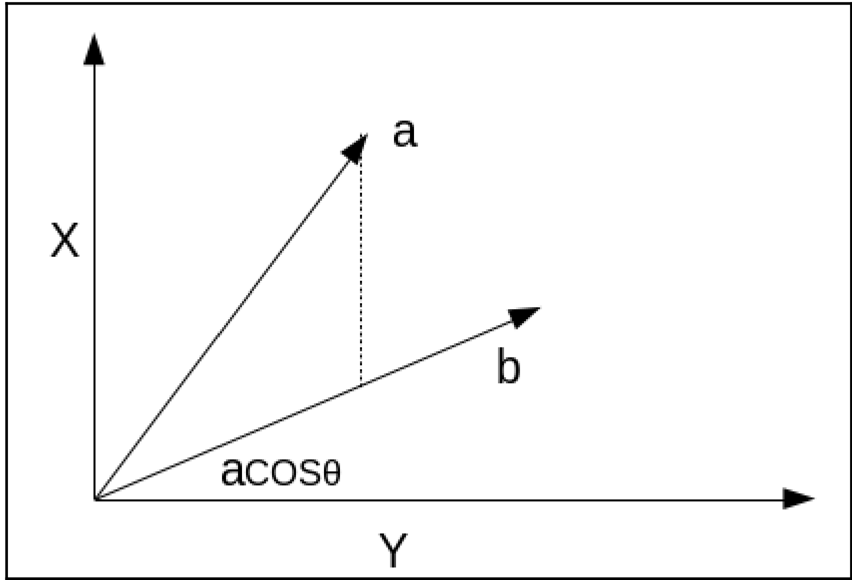
$$\vec{a} \cdot \vec{b} = \|\vec{a}\| \|\vec{b}\| \cos\theta$$

On rearranging the preceding equation, we get the following:

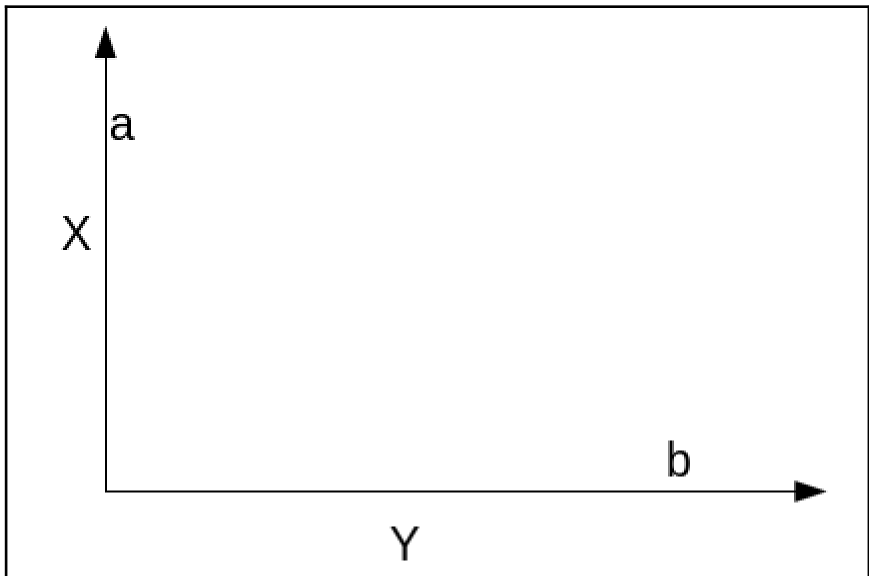
$$\vec{a} \cdot \vec{b} = \|\vec{b}\| \|\vec{a}\| \cos\theta$$

In the earlier equation, *cosθ* is the angle between two vectors, and *acosθ* is the projection of vector A onto vector B.

The visual vector space representation of the dot product between two vectors can be shown as follows:

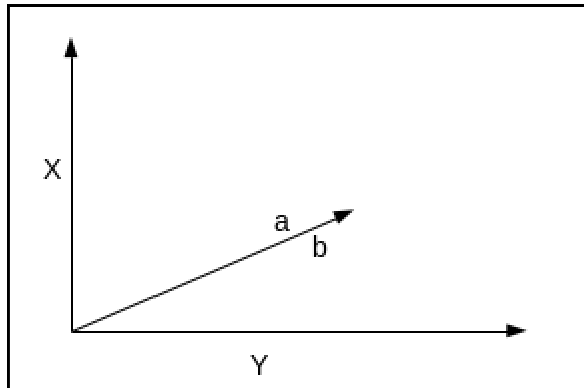


When cos angle between the two vectors is 90 degrees, the *cos 90* will become zero, and the whole dot product will be zero, that is, they will be orthogonal to each other. The logical conclusion we can infer is that they are very far from each other:



When we reduce the cosine angle between the two vectors, their orientation will look very similar to each other's.

When the angle between the two vectors is zero, *cos 0* will be 1, and both the vectors will lie on each other, as shown in the following image. Thus, we can say that the two vectors are similar to each other in terms of their orientation:



So on summarizing, we can conclude that when we compute the cosine angle between two vectors, the resultant scalar value will indicate how close the two vectors are with each other in terms of orientation:

$$\vec{a} \cdot \vec{b} = \|\vec{a}\|\|\vec{b}\| \cos\theta$$

$$\cos\theta = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\|\|\vec{b}\|}$$

Now let's revisit our original question: what does the dot product mean? When we take the dot product between two vectors, the resultant scalar value represents the cosine angle between them. If the scalar is zero, the two vectors are orthogonal and unrelated. If the scalar is 1, the two vectors are similar.

Now, how is this applied in recommendation engines?

As mentioned earlier, consider a rating matrix containing user IDs as rows and item IDs as columns. We can assume each row as user vectors and each column as item vectors.

The cosine angle between row vectors will give the user similarity, and cosine angle between column vectors give the item similarity.

The R script for calculating the cosine distance is as follows:

```
vec1 = c( 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 )
vec2 = c( 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0 )
library(lsa)
cosine(vec1,vec2)
```

Here, *x* is matrix containing all the variables in a dataset; the cosine function is available in the `lsa` package. The `lsa` is a text mining package available in r used for discovering latent features or topics within the text. This package provides `cosine()` method to calculate cosine angle between two vectors.

Jaccard similarity

Jaccard simlarity is another type of similarity measure used in recommendation engines. The **Jaccard similarity** coefficient is calculated as the ratio of the intersection of features to the union of features between two users or items.

Mathematically speaking, if *A* and *B* are two vectors, the Jaccard similarity is given by the following equation:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

The Jaccard similarity coefficient metric is a statistic used to find the similarity and diversity in sample sets. Since users and items can be represented as vectors or sets, we can easily apply the Jaccard coefficient to recommender systems in order to find similarity between users or items.

The R script for calculating the Jaccard similarity is as follows:

```
vec1 = c( 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 )
vec2 = c( 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0 )
library('clusteval')
cluster_similarity(vec1, vec2, similarity = "jaccard")
```

The `clusteval` package in r is a popular package for evaluating clustering techniques. `Cluster_similarity()` method provides a very good implementation for calculating Jaccard similarity.

Pearson correlation coefficient

Another way of finding the aforementioned similarity is to find the correlation between two vectors. Instead of using the distance measures as a way of finding similarity among vectors, we use the correlation between vectors in this approach.

The Pearson correlation coefficient can be computed as follows:

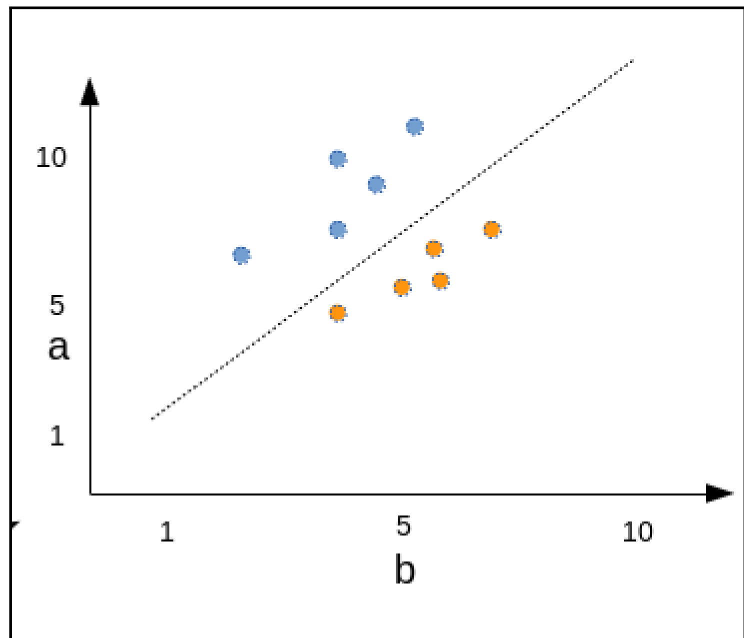
$$r = r_{xy} = \frac{1}{n-1} \sum_{i=1}^{n} \left(\frac{x_i - \bar{x}}{s_x} \right) \left(\frac{y_i - \bar{y}}{s_y} \right)$$

Here, *r* is the correlation coefficient, *n* is the total number of data points, x_i is the ith vector point of the x vector, y_i is the ith vector point of the y vector, *x-bar* is the mean of vector x, *y-bar* is the mean of vector y, s_x is the standard deviation of vector x, *and* s_y is the standard deviation of vector y.

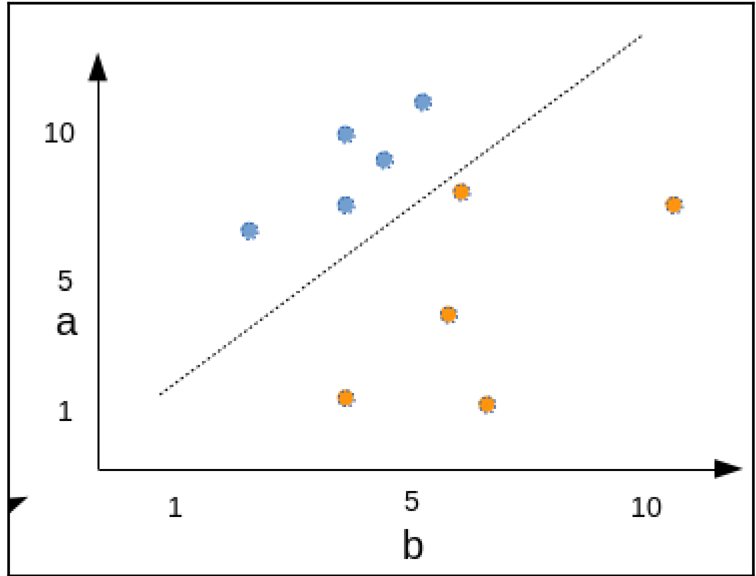
Another way of computing the correlation coefficient between two variables is by dividing the covariance of the two variables by the product of their standard deviations, given by ρ (rho):

$$\rho_{X,Y} = \frac{\mathrm{cov}(X,Y)}{\sigma_X \sigma_Y}$$

Let's understand this with an example, as shown in the following image. We plot the values of two vectors a, b; it is natural to assume that if all the points of the vectors vary together, there exists a positive relation among them. This tendency to vary together, or covariance, in simple terms, can be called correlation. Take a look at the following diagram:



Let's now examine the following image. We can observe that the vectors are not varying together, and the corresponding points are scattered randomly. So the tendency to vary together, or covariance, is less or in other less correlation:



From the similarity calculation aspect, we can conclude that the more the correlation between two vectors, the more similar they are.

Now, how is the Pearson correlation coefficient applied in recommendation engines?

As mentioned earlier, consider a rating matrix containing user IDs as rows and item IDs as columns. We can assume each row as user vectors and each column as item vectors.

The correlation coefficient between the row vectors will give the user similarity, and the correlation coefficient between the column vectors will give the item similarity using the following equation.

The R script is given by the following equation:

```
Coef = cor(mtcars, method="pearson")
```

Here, `mtcars` is the dataset.

Mathematic model techniques

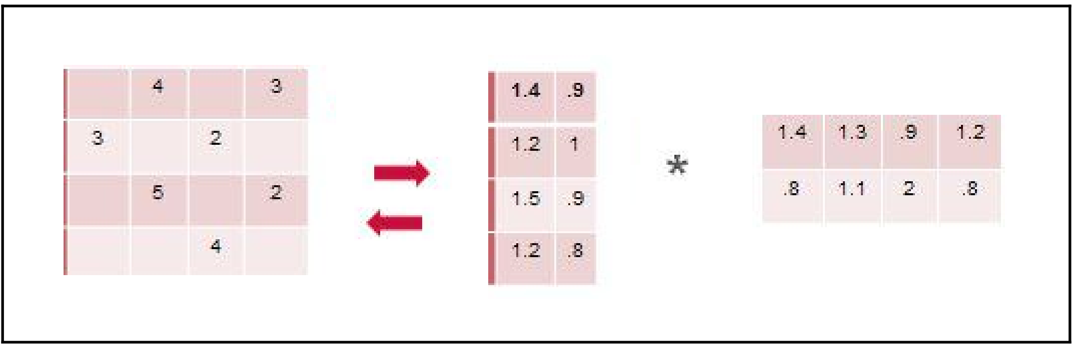
Mathematical models such as matrix factorization and SVD have proved to be very accurate when it comes to building recommendation engines over the similarity calculation measures. Another advantage is their ability to scale down easily also allowed to design the systems easily. In this chapter, we will learn about the mathematical models as explained next.

Matrix factorization

A matrix can be decomposed into two low rank matrices, which when multiplied back will result in a single matrix approximately equal to the original matrix.

Let's say that *R*, a rating matrix of size *U X M* can be decomposed into two low rank matrices, *P* and *Q*, of size *U X K* and *M X K* respectively, where *K* is called the rank of the matrix.

In the following example, the original matrix of size *4 X 4* is decomposed into two matrices, *P (4 X 2)* and *Q (4 X 2)*; multiplying back *P* and *Q* will bring me the original matrix of size *4 X 4* and values approximately equal to those of the original matrix:



One of the major advantages of the matrix factorization method is that we can compute the empty cells in the original matrix, *R*, using the dot product between the low-rank matrices *P*, *Q*. This is given by the following equation:

$$\hat{r}_{ij} = p_i^T q_j = \sum_{k=1}^{k} p_{ik} q_{kj}$$

When we apply the previous equation, we can reproduce the original matrix, *R*, with all the empty cells filled.

In order to make the predicted values as close as to the original matrix as possible, we have to minimize the difference between the original values and the predicted values, which is also also known as error. The error between the original value and predicted value can be given by the following equation:

$$e_{ij}^2 = (r_{ij} - \hat{r}_{ij})^2 = (r_{ij} - \sum_{k=1}^{K} p_{ik} q_{kj})^2$$

In order to minimize the aforementioned error term and reproduce the original matrix as closely as possible, we have to use gradient descend technique-an algorithm to find out optimal parameters of an objective function and minimize the function in an iterative manner, introduce Regularization term to the equation.

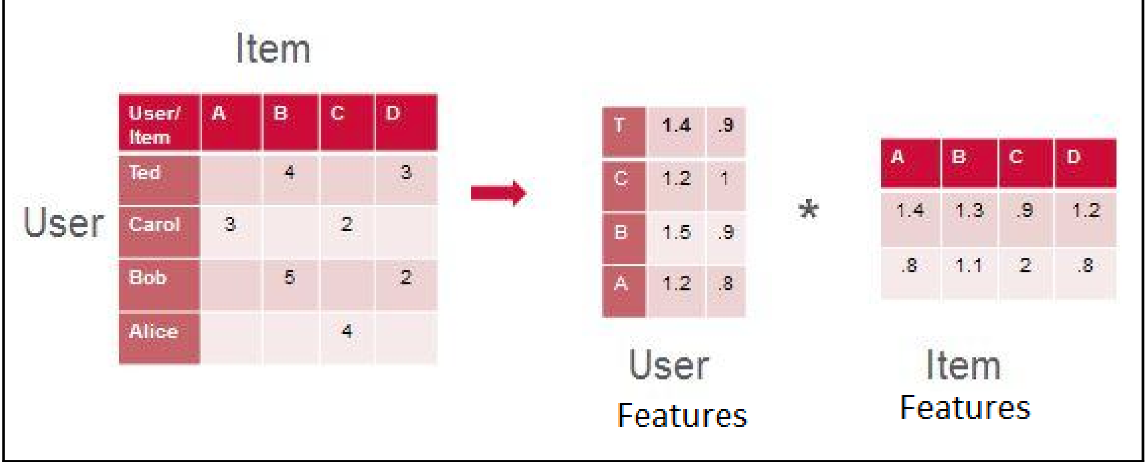
How is matrix factorization applied to recommendation engines?

This is core question, which we are more interested in rather than the mathematics involved in matrix factorization. We will see how we can apply matrix factorization techniques in building recommendation engines.

Recall the core tasks in building recommendation engines: finding similar users or items, then predicting the non-rated preferences, and finally recommending new items to active users. In short, we are predicting the non-rated item preferences. Recall this is what matrix factorization does: predicting the empty cells in the original rating matrix.

Now, how do we justify our approach of applying matrix decomposition to low-rank matrices to recommendation engines? To answer this question, we will discuss how users rate movies. People rate movies because of the story or actors or the genre of the movie, that is, users rate items because of the features of the items. When given a rating matrix containing user IDs, item IDs, and rating values, we can make an assumption that users will have some inherent preferences toward rating items, and the items will also have inherent features that help users rate them. These features of users and items are called **latent features**.

Considering the earlier assumption, we apply the matrix factorization technique to the rating matrix the two low-rank matrices, which are assumed to be the user latent feature matrix and item latent feature matrix:



Considering these assumptions, researchers started applying matrix factorization techniques in building recommender systems. The advantage of the matrix factorization methods is that since it is a machine-learning model, the feature weightages are learned overtime, improving the model accuracy:

The following code explains the implementation of Matrix Factorization using `nmf` package in R:

```
#MF
library(recommenderlab)
data("MovieLense")
dim(MovieLense)

#applying MF using NMF
mat  = as(MovieLense,"matrix")
mat[is.na(mat)] = 0
res = nmf(mat,10)
res

#fitted values
r.hat <- fitted(res)
dim(r.hat)

p <- basis(res)
dim(p)
q <- coef(res)
dim(q)
```

Alternating least squares

Recall the error minimization equation in the previous section. Upon introducing a regularization term to avoid over fitting, the final error term would look like the following equation:

$$\min_{q^*,p^*} \sum_{(u,i)\in\kappa} (r_{ui} - q_i^T p_u)^2 + \lambda(\| q_i \|^2 + \| p_u \|^2)$$

In order to optimize the preceding equation, there are two popular techniques:

- **Stochastic gradient descent** (**SGD**): A mini-batch optimizing technique, similar to gradient descend, for finding optimal parameters in large-scale data or sparse data.
- **Alternating least squares**(**ALS**): The main advantage of ALS method over SGD is that it can be easily parallelized on distributed platforms.

In this section, we will look into the ALS method.

The preceding equation involves two unknowns, which we need to solve. Since two unknowns are involved, the aforementioned equation is a non-convex problem. If we fix one of the unknown term constants, this optimization problem will become quadratic and can be solved optimally.

Alternating least squares is an iterative method, which involves computing one feature vector term using the least squares function by fixing the other feature vector term constant until we solve the preceding equation optimally.

In order to compute the user feature vector, we fix the item feature vector as a constant and solve for least squares. Similarly, while computing the item feature vector, we fix the user feature vector as a constant and solve for least squares.

Following this approach, we are able to convert a non-convex problem into a quadratic one, which can be solved optimally.

Most of the open source distributed platforms such as Mahout and Spark use the ALS method to implement scalable recommender systems for their ability to be parallelized.

Singular value decomposition

Singular value decomposition (**SVD**) is another very popular matrix factorization method. In simple terms, an SVD method decomposes a real matrix A of size m x n into three matrices, U, Σ ,V, which satisfy the following equation:

$$A = U \times \Sigma \times V^T$$

(m x n) (m x r) (r x r) (r x n)

Where **U** is (m x r) matrix
V is (n x r) matrix
Σ is (r x r) matrix

In the previous equation, *r* is called the rank of the matrix, *A*. *U*, *V* are orthogonal matrices, and Σ is a diagonal matrix having all singular values of the matrix *A*. The values of the *U* and *V* are real if *A* is a real matrix. The values of matrix Σ are positive and real and are available in a decreasing order.

SVD can also be used as a dimensionality reduction technique, following two steps:

- Choose a rank *k* that is less than *r*.
- Recompute or subsize the *U*, Σ ,*V* matrices to *(m x k)*, *(k x k)*, *(k x n)*.

U is (m xk) matrix
V is (n xk) matrix
Σ is (kx k) matrix

The matrices obtained by applying SVD are very much applicable to recommender systems as they provide the best low-rank approximations of the original matrix. How do we apply the SVD approach to recommendation? Let's take a rating matrix R of size m x n containing many empty cells. Similar to matrix factorization, our objective is to compute an approximate rating matrix as close as possible o the original matrix with the missing values being predicted.

Applying SVD on R will produce three matrices, *U*, Σ, *V*, of sizes, let's say, *m x r*, *r x r*, *r x n*. Here, *U* represents the user latent feature vector representations, *V* represents the item latent feature vector representations, and Σ represents the independent feature representation, r, of user and items. By setting the value of the independent feature representation to a value *k* less than *r*, we are choosing the k optimal latent features, thereby reducing the size of the matrix. The k-value can be chosen using the cross-validation approach as the value of *k* defines the performance of the model.

A simpler method for choosing the value *k* as Σ is to take a diagonal matrix that contains singular values in a descending order, choose the values in the diagonal that have higher values, and eliminate very less diagonal values.

After choosing the k-value, we now resize or choose the first k-column in each of the matrices *U*, Σ ,*V*. This step will render matrices *U*, Σ ,*V* of size *m x k*, *k x k*, and *k x n* respectively, please refer to the below image. After we resize the matrices, we move ahead to the final step.

In the final step, we will compute the dot products of the following series of matrices in order to calculate the approximate rating matrix $\hat{A}$:

Step 1: $\underset{(m \times k)}{U \otimes \Sigma^{1/2}}$

Step 2: $\underset{(k \times n)}{\Sigma^{1/2} \otimes V^T}$

Step 3: $\underset{(m \times n)}{\hat{A}} = \underset{(m \times k)}{U \otimes \Sigma^{1/2}} \otimes \underset{(k \times n)}{\Sigma^{1/2} \otimes V^T}$

Below code snippet shows SVD implementations in R, the following code creates a sample matrix and then SVD is applied, using `svd()` available in base package in r, on the sample data to create 3 matrices, dot product between three matrices will get back our approximate original matrix.

```
sampleMat <- function(n) { i <- 1:n; 1 / outer(i - 1, i, "+") }
original.mat <- sampleMat(9)[, 1:6]
(s <- svd(original.mat))
D <- diag(s$d)
#  X = U D V'
s$u %*% D %*% t(s$v)
```

Please refer to `Chapter 7`, *Building Real-Time Recommendation Engines with Spark* for ALS implementation in Spark-python.

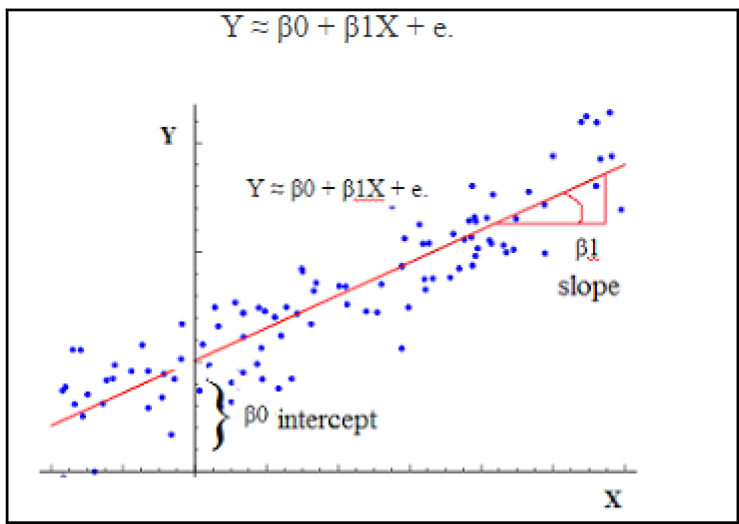
Machine learning techniques

In this section, we will learn about the most important or the most frequently used machine learning techniques, which are widely used in building recommendation engines.

Linear regression

Linear regression may be treated as a simple, popular, and the foremost approach for solving prediction problems. We employ linear regression where our objective is to predict the future outcomes, given input features and the output label is a continuous variable.

In linear regression, given historical input and output data, the model will try to find out the relation between independent feature variables and the dependent output variable given by the following equation and diagram:



Here, *y* represents the output continuous dependent variable, *x* represents independent feature variables, *β0* and *β1* are the unknowns or feature weights, *e* represents the error.

Using the **ordinary least squares (OLS)** approach, we will estimate the unknowns in the preceding equation. We shall not go deep into the linear regression approach, but here we will discuss how we can use linear regression in recommendation engines.

One of the core tasks in recommendation engines is to make predictions for non-rated items for users. For example, in case of item-based recommendation engines, the prediction for item *i* by user *u* is done by computing the sum of ratings given by user *u* to items similar to item *i*. Then each rating is weighted by its similarity value:

$$P_{u,i} = \frac{\sum_{\text{all similar items, N}}(s_{i,N} * R_{u,N})}{\sum_{\text{all similar items, N}}(|s_{i,N}|)}$$

Instead of using this weighted average approach to make the predictions, we can use the linear regression approach to calculate the preference values for user *u* for item *i*. While using regression approach, instead of using the original rating values of similar items, we use their approximate rating values based on the linear regression model. For example, to predict the rating for item *i* by user *u*, we can use the following equation:

$$\bar{R'_N} = \alpha \bar{R_i} + \beta + \epsilon$$

Linear regression using R is given by the following code:

```
library(MASS)
data("Boston")
set.seed(0)
which_train <- sample(x = c(TRUE, FALSE), size = nrow(Boston),
                      replace = TRUE, prob = c(0.8, 0.2))
train <- Boston[which_train, ]
test <- Boston[!which_train, ]
lm.fit =lm(medv~. ,data=train )
summary(lm.fit)

Call:
lm(formula = medv ~ ., data = train)

Residuals:
     Min       1Q   Median       3Q      Max
-15.2631  -2.7614  -0.5243   1.7867  24.6306
```

```
Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  39.549376   5.814446   6.802 3.82e-11 ***
crim         -0.090720   0.040872  -2.220  0.02701 *
zn            0.050080   0.015307   3.272  0.00116 **
indus         0.032339   0.070343   0.460  0.64596
chas          2.451235   0.992848   2.469  0.01397 *
nox         -18.517205   4.407645  -4.201 3.28e-05 ***
rm            3.480574   0.469970   7.406 7.91e-13 ***
age           0.012625   0.015786   0.800  0.42434
dis          -1.470081   0.223349  -6.582 1.48e-10 ***
rad           0.322494   0.077050   4.186 3.51e-05 ***
tax          -0.012839   0.004339  -2.959  0.00327 **
ptratio      -0.972700   0.148454  -6.552 1.77e-10 ***
black         0.008399   0.003153   2.663  0.00805 **
lstat        -0.592906   0.058214 -10.185  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 4.92 on 396 degrees of freedom
Multiple R-squared:  0.7321,     Adjusted R-squared:  0.7233
F-statistic: 83.26 on 13 and 396 DF,  p-value: < 2.2e-16

#predict new values
pred = predict(lm.fit,test[,-14])
```

The `lm()` function, available in `stats` package I R, usually used to fit linear regression models.

Classification models

Classification models fall into the category of the supervised form of machine learning. These models are usually employed in prediction problems, where the response is binary or multiclass labels. In this chapter, we will discuss many types of classification models, such as logistic regression, KNN classification, SVM, decision trees, random forests, bagging, and boosting. Classification models play a very crucial role in recommender systems. Though classification models don't play a great role in neighbourhood methods, they play a very important role in building personalized recommendations, contextual aware systems, and hybrid recommenders. Also, we can apply classification models to the feedback information about the recommendations, which can further be used for calculating the weightages for user features.

Linear classification

Logistic regression is the most common among classification models. **Logistic regression** is also known as **linear classification** as it is very similar to linear regression, except that in regression, the output label is continuous, whereas in linear classification, the output label is class variable. In regression, the model is a least squares function, whereas in logistic regression, the prediction model is a logit function given by the following equation:

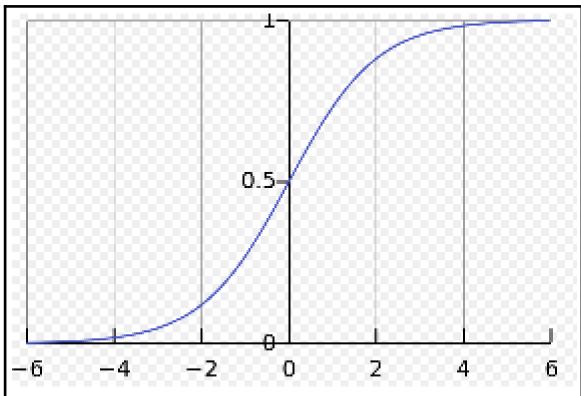
$$F(x) = \frac{1}{1 + e^{-\mathbf{x}}}$$

$$\mathbf{x} = \beta_0 + \beta_1 x$$

$$F(x) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x)}}$$

In the preceding equation, *e* is the natural logarithm, *x* is the input variable, β_0 is the intercept, and β_1 is the weight of variable x.

We may interpret the preceding equation as the conditional probability of response variable against the linear combination of input variables. The logit function allows to take any continuous variable and gives response in the range of *(0,1)*, as illustrated in the following diagram:



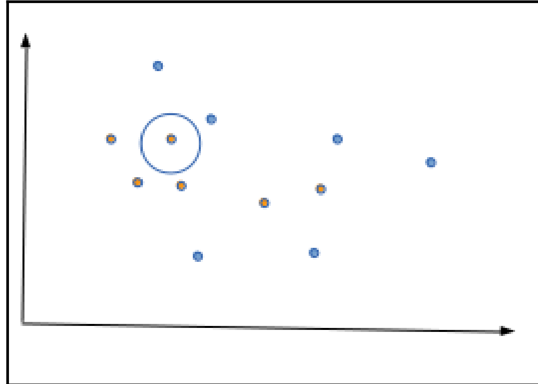
The logistic regression using R is given as follows:

```
set.seed(1)
x1 = rnorm(1000)            # sample continuous variables
x2 = rnorm(1000)
z = 1 + 4*x1 + 3*x2         # data creation
pr = 1/(1+exp(-z))          # applying logit function
y = rbinom(1000,1,pr)       # bernoulli response variable

  #now feed it to glm:
df = data.frame(y=y,x1=x1,x2=x2)
glm( y~x1+x2,data=df,family="binomial")
```

The `glm()` function in R is used to fit generalized linear models, popularly employed for classification problems.

KNN classification

The k nearest neighbors classification is popularly known as the KNN classification. This is one of the most popular classification techniques. The basic concept in KNN classification is that the algorithm considers k nearest items surrounding a particular data point and tries to classify this data point into one of the output labels based on its k-nearest data points. Unlike other classification techniques such as logistic regression, SVM, or any other classification algorithms, KNN classification is a non-parametric model, which doesn't involves any parameter estimation. The k in KNN is the number of nearest neighbors to be considered:



Consider 10 data points. We need to classify a test data point, highlighted in the preceding diagram, into one of two classes, blue or orange. In this example, we classify the test data point using the KNN classification. Let's say k is 4; it means that by considering four data points surrounding the active data point, we need to classify it by performing the following steps:

- As a first step, we need to calculate the distance of each point from the test data point.
- Identify the top four closest data points to the test data point.
- Using the voting mechanism, assign the majority class label count to the test data point.

The KNN classification works well in case of highly non-linear problems. Though this method works well in most cases, this method being a non-parametric approach cannot find the feature importance or weightages.

Similar to the KNN classification, there is a regression version of KNN, which can be used to predict the continuous output labels.

Both the KNN classification and egression methods find their wide applicability in collaborative filtering recommender systems.

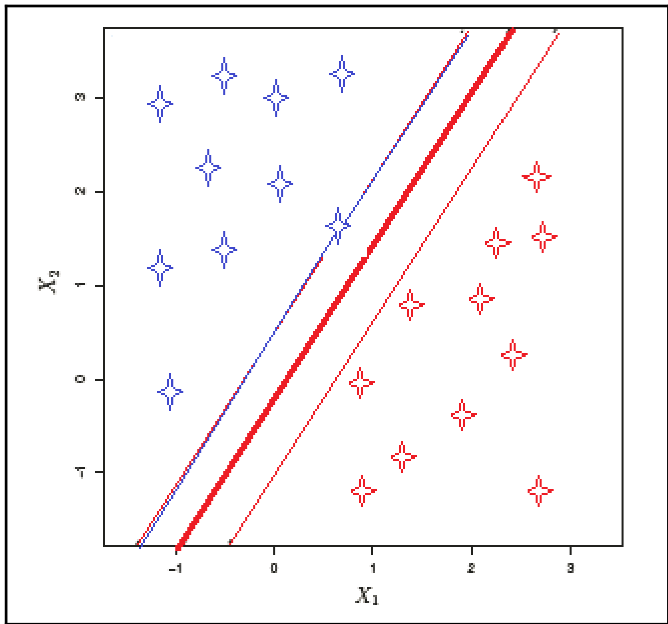
The following code snippet shows KNN classification using R, in the below code snippet we are using `knn3()` available in `caret` package for fitting KNN classification and `sample_n()` available in `dplyr` package to select random rows from a dataframe.

```
data("iris")
library(dplyr)
iris2 = sample_n(iris, 150)
train = iris2[1:120,]
test = iris2[121:150,]
cl = train$Species
library(caret)
fit <- knn3(Species~., data=train, k=3)
predictions <- predict(fit, test[,-5], type="class")
table(predictions, test$Species)
```

Support vector machines

Support vector machines algorithms are a form of supervised learning algorithms employed for solving classification problems. SVM is generally treated as one of the best algorithms for dealing with classification problems. Given a set of training examples, where each of the data points falls into one of two categories, an SVM training algorithm builds a model that assigns new data points into one category or the other. This model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a margin that is as wide as possible, as shown in the following figure. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall on. In this section, we will go through an overview and implementation of SVMs without going into mathematical details.

When SVM is applied to a p-dimensional dataset, the data is mapped to a p-1 dimensional hyper plane, and the algorithm finds a clear boundary with sufficient margin between classes. Unlike other classification algorithms that also create a separating boundary for classifying data points, SVM tries to choose a boundary that has maximum margin for separating the classes, as shown in the following figure:



Consider a two-dimensional dataset having two classes, as shown in the previous figure. Now, when the SVM algorithm is applied, firstly it checks whether a one-dimensional hyper plane exists to map all the data points. If the hyper plane exists, the linear classifier creates a decision boundary with a margin to separate the classes. In the preceding figure, the thick redline is the decision boundary, and the thinner blue and red lines are the margins of each class from the boundary. When new test data is used to predict the class, the new data falls into one of the two classes.

The following are a few key points to be noted:

- Though an infinite number of hyper planes can be created, SVM chooses only one hyper plane that has maximum margin, that is, the separating hyper plane that is farthest from the training observations.
- This classifier is only dependent on the data points that lie on the margins of the hyper plane, that is, on thin margins in the figure but not on other observations in the dataset. These points are called support vectors.
- The decision boundary is affected only by the support vectors but not by other observations located away from the boundaries, that is, if we change the data points other than the support vectors, there will not be any effect on the decision boundary, but if the support vectors are changed, the decision boundary changes.
- A large margin on the training data will also have a large margin on the test data so as to classify the test data correctly.
- Support vector machines also perform well with non-linear datasets. In this case, we use radial kernel functions.

See below for R implementation of SVM on the `iris` dataset. We use the `e1071` package to run SVM. In R, the `SVM()` function contains the implementation of Support Vector Machines present in the `e1071` package.

The cross-validation method is used to evaluate the accuracy of predictive models before testing future unseen data.

We can see that the SVM method is called with the `tune()` method, which performs cross validation and runs the model on different values of the cost parameters:

```
library(e1071)
data(iris)
sample = iris[sample(nrow(iris)),]
train = sample[1:105,]
test = sample[106:150,]
tune =tune(svm,Species~.,data=train,kernel ="radial",scale=FALSE,ranges
=list(cost=c(0.001,0.01,0.1,1,5,10,100)))
tune$best.model

Call:
best.tune(method = svm, train.x = Species ~ ., data = train, ranges =
list(cost = c(0.001,
    0.01, 0.1, 1, 5, 10, 100)), kernel = "radial", scale = FALSE)

Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  radial
       cost:  10
      gamma:  0.25

Number of Support Vectors:  25

summary(tune)

Parameter tuning of 'svm':
- sampling method: 10-fold cross validation
- best parameters:
 cost
   10
- best performance: 0.02909091
- Detailed performance results:
   cost      error dispersion
1 1e-03 0.72909091 0.20358585
2 1e-02 0.72909091 0.20358585
3 1e-01 0.04636364 0.08891242
4 1e+00 0.04818182 0.06653568
5 5e+00 0.03818182 0.06538717
6 1e+01 0.02909091 0.04690612
7 1e+02 0.07636364 0.08679584

cost =10 is chosen from summary result of tune variable
model =svm(Species~.,data=train,kernel ="radial",cost=10,scale=FALSE)
```

The `tune$best.model`tells us that the model works best with cost parameter as `10` and the total number of support vectors as `25`:

```
pred = predict(model,test)
```

Decision trees

`Decision trees` is a simple, fast, and tree-based supervised learning algorithm for solving classification problems. Though not very accurate when compared to other logistic regression methods, this algorithm comes in handy while dealing with recommender systems.

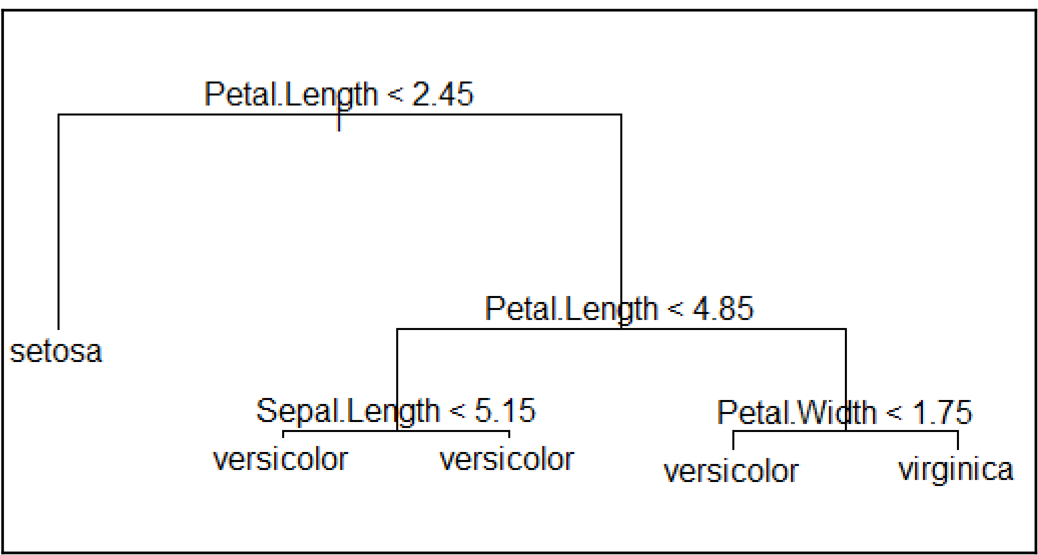
Let's define the decision trees with an example. Imagine a situation where you have to predict the class of flower based on its features such as petal length, petal width, sepal length and sepal width. We apply the decision trees methodology to solve this problem:

1. We consider the entire data at the start of the algorithm.
2. Now we choose a proper question/variable to divide the data into two parts. In our case, we choose to divide the data based on petal length > 2.45 and <= 2.45. This separates flower class `setosa` from the rest of the classes.
3. We further divide the data having petal length > 2.45, based on same variable with petal length < 4.5 and >= 4.5 as shown in the following diagram.
4. This splitting of the data will be further divided by narrowing down the data space until we reach a point where all the bottom points represent the response variables or where further logical split cannot be done on the data.

In the following decision tree diagram, we have one root node, four internal nodes where data split occurred, five terminal nodes where data split cannot be done further, and they are defined as follows:

- Petal Length < 2.5 as root node
- Petal length < 2.5, petal length < 4.85, sepal length < 5.15, and petal width < 1.75 are called internal nodes
- Final nodes having the class of the flowers are called terminal nodes
- The lines connecting the nodes are called the branches of the tree

While predicting responses on new data using the aforementioned built model, each of the new data points is taken through each of the nodes, a question is asked, and a logical path is taken to reach its logical class:



Take a look at the decision tree implementation in R on the `iris` dataset using tree package available from CRAN.

The summary of the mode given next tells us that the misclassification rate is 0.0381, indicating that the model is very accurate:

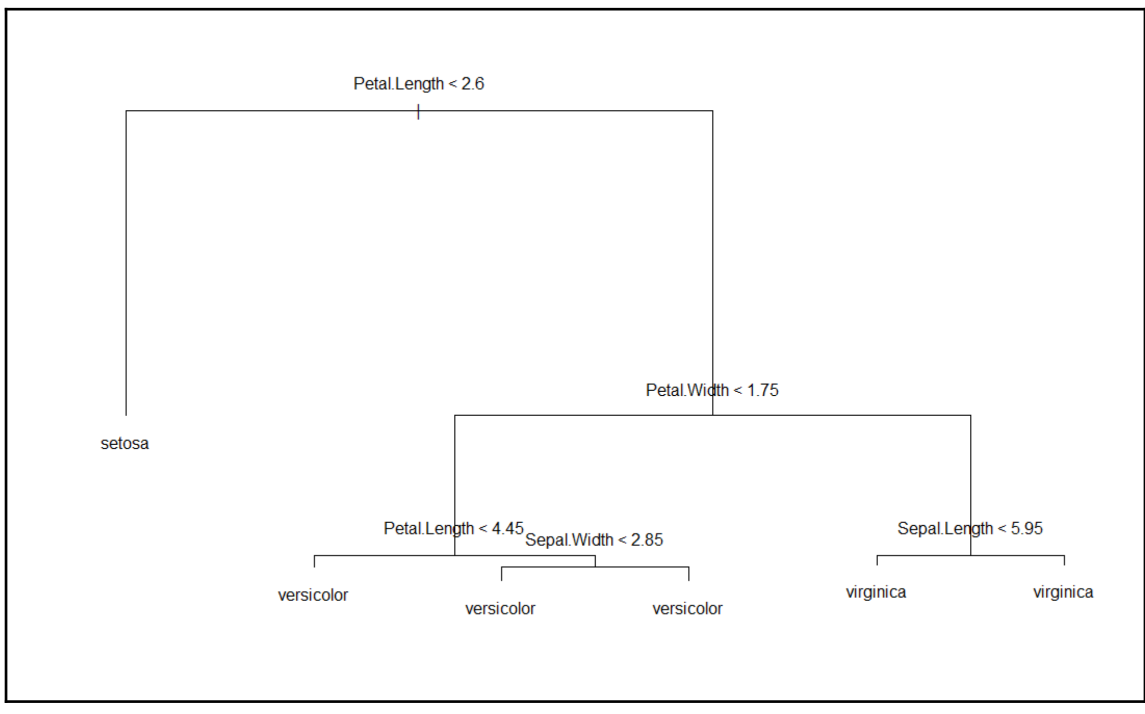
```
library(tree)
data(iris)
sample = iris[sample(nrow(iris)),]
train = sample[1:105,]
test = sample[106:150,]
model = tree(Species~.,train)
summary(model)
```

```
Classification tree:
tree(formula = Species ~ ., data = train)
Number of terminal nodes:  6
Residual mean deviance:  0.1471 = 14.56 / 99
Misclassification error rate: 0.0381 = 4 / 105
```

Below code shows plotting the decision tree:

```
plot(model) #plot trees
text(model) #apply text
```

The following code displays the decision tree model:




```
pred = predict(model,test[,-5],type="class")
```

The following image displays the prediction values made using `pred()` method:

```
> pred
 [1] versicolor virginica  versicolor versicolor versicolor setosa     virginica
 [8] setosa     virginica  versicolor setosa     setosa     setosa     setosa
[15] versicolor versicolor versicolor setosa     versicolor versicolor virginica
[22] virginica  virginica  setosa     setosa     versicolor virginica  versicolor
[29] versicolor virginica  virginica  virginica  versicolor virginica  versicolor
[36] virginica  versicolor setosa     virginica  setosa     virginica  versicolor
[43] versicolor versicolor setosa
Levels: setosa versicolor virginica
```

Ensemble methods

In data mining, we use ensemble methods, which refer to using multiple learning algorithms to obtain better predictive results than applying any single learning algorithm on any statistical problem. This section deals with an overview of popular ensemble methods such as bagging, boosting, and random forests.

Random forests

Random forests refer to improvised supervised algorithm than bootstrap aggregation or bagging method though built on a similar approach. Unlike selecting all the variables in all the B samples generated using the bootstrap technique in bagging, we select only a few predictor variables randomly from total variables for each of the B samples, and then these samples are trained with the models. Predictions are made by averaging the result of each model. The number of predictors in each sample is decided using the formula $m = \sqrt{p}$, where p is the total variable count in the original dataset.

This approach removes the condition of dependency of strong predictor in the dataset as we intentionally select fewer variables than all the variables for every iteration.

- This approach also decorrelates variables resulting in less variability in the model, hence more reliability.

Take a look at the following R implementation of random forests on the iris dataset using the `randomForest` package available from CRAN:

```
library(randomForest)
data(iris)
sample = iris[sample(nrow(iris)),]
train = sample[1:105,]
test = sample[106:150,]
model =randomForest(Species~.,data=train,mtry=2,importance
=TRUE,proximity=TRUE)
```

The following image will display the model details for random forest built above:

```
> model

Call:
 randomForest(formula = Species ~ ., data = train, mtry = 2, importance = TRUE,      proximity = TRUE)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 2

        OOB estimate of  error rate: 6.67%
Confusion matrix:
           setosa versicolor virginica class.error
setosa         40          0         0  0.00000000
versicolor      0         28         3  0.09677419
virginica       0          4        30  0.11764706
```

```
pred = predict(model,newdata=test[,-5])
```

```
> pred
        96        105        138         99         39         37        149        106         29
versicolor  virginica  virginica versicolor     setosa     setosa  virginica  virginica     setosa
        66         61         70        140         83        126         77         53        102
versicolor versicolor versicolor  virginica versicolor  virginica versicolor versicolor  virginica
       135         82        103         52        146         58         67         19         87
 virginica versicolor  virginica versicolor  virginica versicolor versicolor     setosa versicolor
         5        124         57         42         68        100        145         32          6
    setosa  virginica versicolor     setosa versicolor versicolor  virginica     setosa     setosa
       139         21         86        148        130        108         47         98         73
 virginica     setosa versicolor  virginica  virginica  virginica     setosa versicolor  virginica
Levels: setosa versicolor virginica
```

Bagging

Bagging is also known as **bootstrap aggregating**. It is designed to improve the stability and accuracy of machine learning algorithms. It helps in avoiding overfitting and reduces variance. This is mostly used with decision trees.

Bagging involves randomly generating bootstrap samples, random sample with replacement, from the dataset and training the models individually. Predictions are then made by aggregating or averaging all the response variables.

For example, consider a dataset (Xi, Yi) where i=1 …n, contains n data points. The following are the steps to perform bagging on this dataset:

- Now randomly select B samples with replacement from the original dataset using the bootstrap technique.
- Next, train the B samples with regression/classification models independently, and then predictions are made on the test set by averaging the responses from all the B models generated in case of regression or selecting the most-occurring class among B samples in case of classification.

Boosting

Unlike in bagging where multiple copies of bootstrap samples are created and a new model is fitted for each copy of dataset and all the individual models are combined to create a single predictive model, in boosting, each new model is built using information from previously built models. Boosting can be understood as an iterative method involving two steps:

1. A new model is built on the residuals of the previous models instead of response variable.
2. Now the residuals are calculated from this model and updated to the residuals used in previous step.

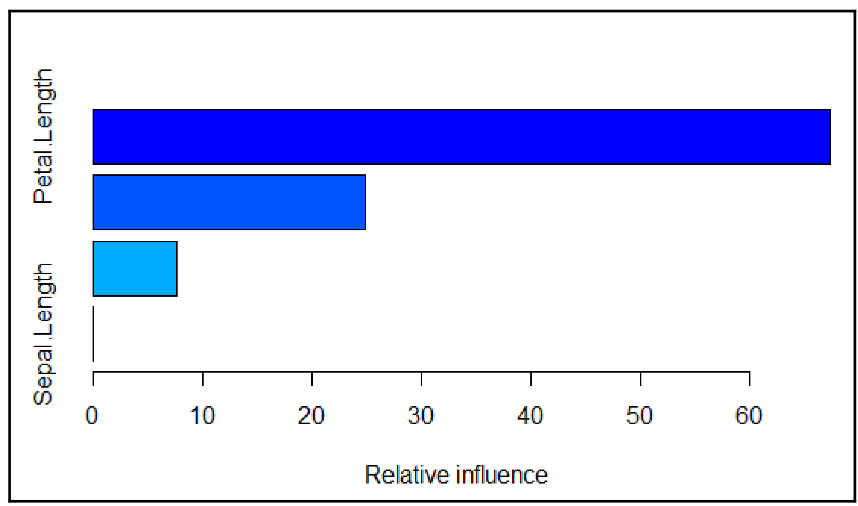
The preceding two steps are repeated multiple iterations allowing each new model to learn from its previous mistakes, thereby improving the model accuracy.

The following code snippet shows us gradient boosting using R, `gbm()` package in r generally used to perform various regression tasks:

```
library(gbm)
data(iris)
sample = iris[sample(nrow(iris)),]
train = sample[1:105,]
test = sample[106:150,]
model =
gbm(Species~.,data=train,distribution="multinomial",n.trees=5000,interactio
n.depth=4)
summary(model)
```

```
> summary(model)
                      var   rel.inf
Petal.Length Petal.Length 67.440852
Petal.Width   Petal.Width 24.942084
Sepal.Width   Sepal.Width  7.617065
Sepal.Length Sepal.Length  0.000000
```

The following image shows them model summary visually, showing the relative importance of each feature:



The preceding summary states the relative importance of the variables of the model.

```
pred = predict(model,newdata=test[,-5],n.trees=5000)
```

```
> pred[1:5,,]
         setosa versicolor virginica
[1,]  5.647443  -2.951628 -4.964130
[2,] -5.238890  -3.222812  4.295997
[3,] -5.289086   3.447595 -3.277463
[4,] -5.288114   2.690219 -2.389940
[5,] -5.245599  -1.588168  3.026419
```

Pick the response with the highest probability from the resulting `pred` matrix, by doing `apply(pred, 1, which.max)` on the vector output from prediction.

```
p.pred <- apply(pred,1,which.max)
```

```
> p.pred
 [1] 1 3 2 2 3 1 1 1 1 3 1 3 2 2 2 3 2 1 2 1 3 2 3 1 1 2 3 1 2 1 2 2 1 3 3 1 2 1 1 3 2 2 3 2 2
>
```

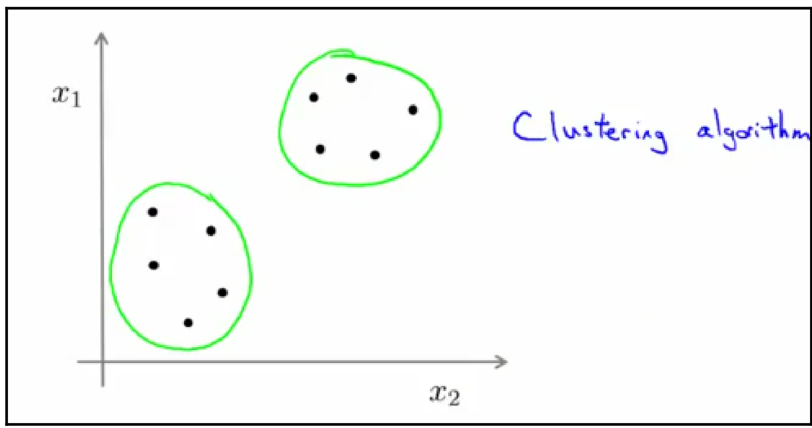
In the preceding code snippet, the output value for the `predict()` function is used in the `apply()` function to pick the response with highest probability among each row in the pred matrix, and the resultant output from the `apply()` function is the prediction for the response variable.

Clustering techniques

Cluster analysis is the process of grouping objects together in a way that objects in one group are more similar than objects in other groups.

For example, identifying and grouping clients with similar booking activities on travel portal, as shown in the following figure.

In the preceding example, each group is called a **cluster**, and each member (data point) of the cluster behaves similar to its group members:



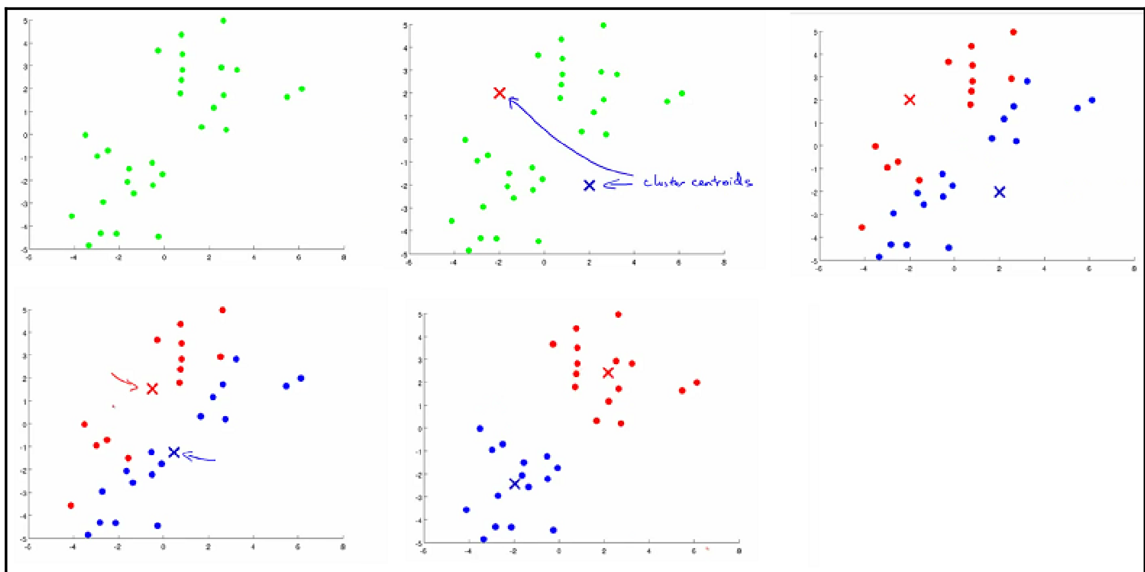
Cluster analysis is an unsupervised learning method. In supervised methods such as regression analysis, we have input variables and response variables; we fit a statistical model to the input variables to predict the response variable, whereas in unsupervised learning methods, we do not have any response variable to predict; we only have input variables. Instead of fitting a model to the input variables to predict the response variable, we just try to find patterns within the dataset. There are three popular clustering algorithms: hierarchical cluster analysis, k-means cluster analysis, and two-step cluster analysis. In this section, we will learn about k-means clustering.

K-means clustering

K-means is an unsupervised, iterative algorithm where k is the number of clusters to formed from the data. Clustering is achieved in two steps, as follows:

- **The cluster assignment step**: In this step, we randomly choose two cluster points (red dot & green dot) and assign each data point to one of the two cluster points, whichever is closer to it (Take a look at the top part of the following figure).
- **The move centroid step:** In this step, we take the average of the points of all the examples in each group and move the centroid to the new position, that is, the mean position calculated (Take a look at the bottom part of the following image).

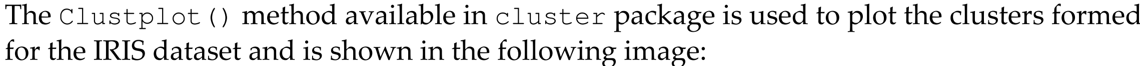
The preceding steps are repeated until all the data points are grouped into two groups and the mean of the data points at the end of the move centroid step doesn't change:



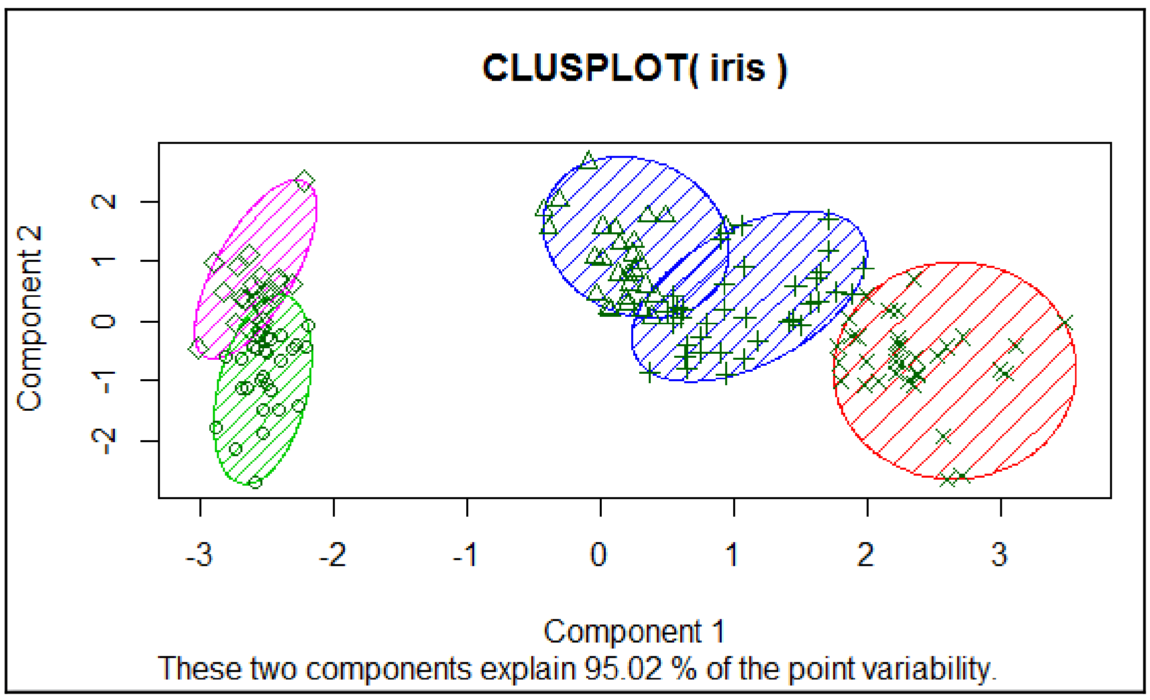
The previous figure shows how a clustering algorithm works on data to form clusters. Take a look at the following R implementation of k-means clustering on the iris dataset.

The k-means clustering using R is as follows:

```
library(cluster)
data(iris)
iris$Species = as.numeric(iris$Species)
kmeans<- kmeans(x=iris, centers=5)
clusplot(iris,kmeans$cluster, color=TRUE, shade=TRUE,labels=13, lines=0)
```



The `Clustplot()` method available in `cluster` package is used to plot the clusters formed for the IRIS dataset and is shown in the following image:

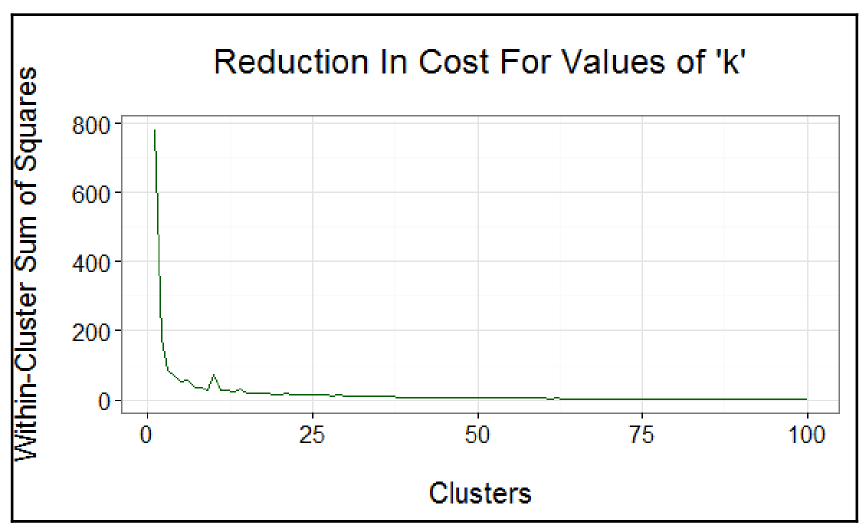


The previous figure shows the formation of clusters on the iris data, and the clusters account for 95% of the data. In the previous example, the number of clusters k value is selected using the elbow method.

The following code snippet explains implementation of k-means clustering, which is shown in the next screenshot:

```
library(cluster)
library(ggplot2)
data(iris)
iris$Species = as.numeric(iris$Species)
cost_df <- data.frame()
for(i in 1:100){
kmeans<- kmeans(x=iris, centers=i, iter.max=50)
cost_df<- rbind(cost_df, cbind(i, kmeans$tot.withinss))
}
names(cost_df) <- c("cluster", "cost")
#Elbow method to identify the idle number of Cluster
#Cost plot
```

```
ggplot(data=cost_df, aes(x=cluster, y=cost, group=1)) +
theme_bw(base_family="Garamond") +
geom_line(colour = "darkgreen") +
theme(text = element_text(size=20)) +
ggtitle("Reduction In Cost For Values of 'k'\n") +
xlab("\nClusters") + ylab("Within-Cluster Sum of Squares\n")
```



From the previous figure, we can observe that the direction of the cost function is changed at cluster number 5, hence we choose 5 as our number of clusters k. Since the number of optimal clusters is found at elbow of the graph, we call it the **elbow method**.

Dimensionality reduction

One of the most commonly faced problems while building recommender systems is high dimensional and sparse data. Many times, we face a situation where we have a large set of features and less number of data points. In such situations, when we fit a model to the dataset, the predictive power of the model would be lower. This scenario is often termed as the curse of dimensionality. In general, adding more data points or decreasing the feature space, also known as dimensionality reduction, often reduces the effects of curse of dimensionality. In this section, we will discuss principal component analysis, a popular dimensionality reduction technique to reduce the effects of the curse of dimensionality.

Principal component analysis

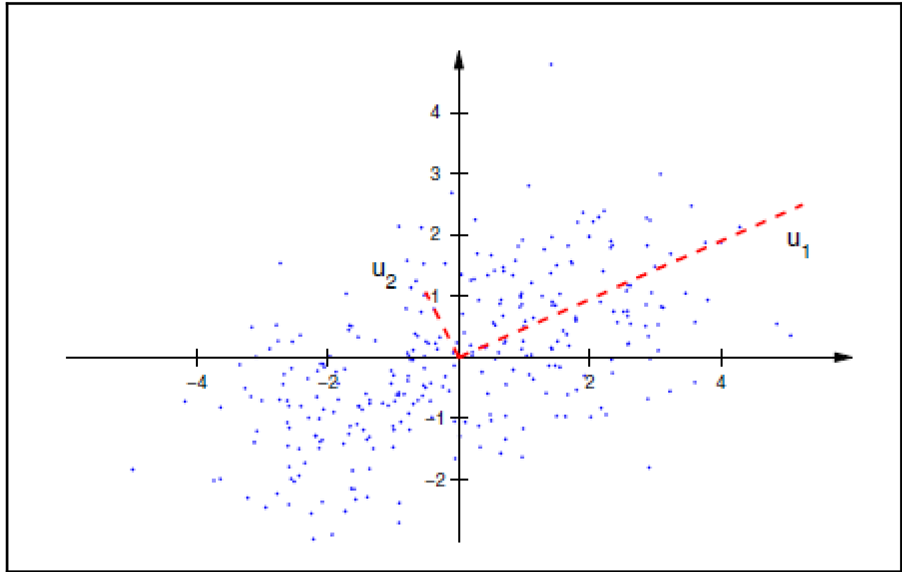
Principal component analysis (**PCA**) is a classical statistical technique for dimensionality reduction. PCA algorithm transforms the data with high dimensional space to a space with fewer dimensions. The algorithm linearly transforms m-dimensional input space to n-dimensional (n<m) output space, with the objective of minimizing the amount of information/variance lost by discarding (m-n) dimensions. PCA allows us to discard the variables/features that have less variance.

Technically speaking, PCA uses orthogonal projection of highly correlated variables to a set of values of linearly uncorrelated variables called principal components. The number of principal components is less than or equal to the number of original variables. This linear transformation is defined in such a way that the first principal component has the largest possible variance, that is, it accounts for as much of the variability in the data as possible by considering highly correlated features, and each succeeding component, in turn, has the highest variance by using the features that are less correlated with the first principal component, which is orthogonal to the preceding component.

Let's understand this in simple terms. Assume that we have a three-dimensional data space with two features more correlated with each other than with the third. We now want to reduce the data to a two-dimensional space using PCA.

The first principal component is created in such a way that it explains maximum variance using the two correlated variables along the data. In the following figure, the first principal component (the bigger line) is along the data explaining most variance. To choose the second principal component, we need to choose another line that has the highest variance, is uncorrelated, and is orthogonal to the first principal component, The implementation and technical details of PCA is out of scope of this book, so we will discuss how it used in R.

The following image explains the spatial representation of principal components:



We illustrate PCA using the `USArrests` dataset. The `USArrests` dataset contains crime related statistics, such as `Assault`, `Murder`, `Rape`, `UrbanPop` per 100,000 residents in 50 states in the U.S..

PCA implementation in R is as follows:

```
data(USArrests)
head(states)
[1] "Alabama"     "Alaska"      "Arizona"     "Arkansas"    "California"
"Colorado"

names(USArrests)
[1] "Murder"    "Assault"   "UrbanPop"  "Rape"
```

Let's use the `apply()` function to the `USArrests` dataset row-wise to calculate the variance to see how each variable is varying:

```
apply(USArrests , 2, var)

Murder    Assault   UrbanPop       Rape
  18.97047 6945.16571  209.51878   87.72916
```

We observe that `Assault` has the most variance. It is important to note at this point that scaling the features is a very important step while applying PCA.

Apply PCA after scaling the feature, as follows:

```
pca =prcomp(USArrests , scale =TRUE)

pca
Standard deviations:
[1] 1.5748783 0.9948694 0.5971291 0.4164494

Rotation:
                PC1        PC2        PC3         PC4
Murder   -0.5358995  0.4181809 -0.3412327  0.64922780
Assault  -0.5831836  0.1879856 -0.2681484 -0.74340748
UrbanPop -0.2781909 -0.8728062 -0.3780158  0.13387773
Rape     -0.5434321 -0.1673186  0.8177779  0.08902432
```

Now let's understand the components of PCA output:

```
names(pca)
[1] "sdev"     "rotation" "center"   "scale"    "x"
```

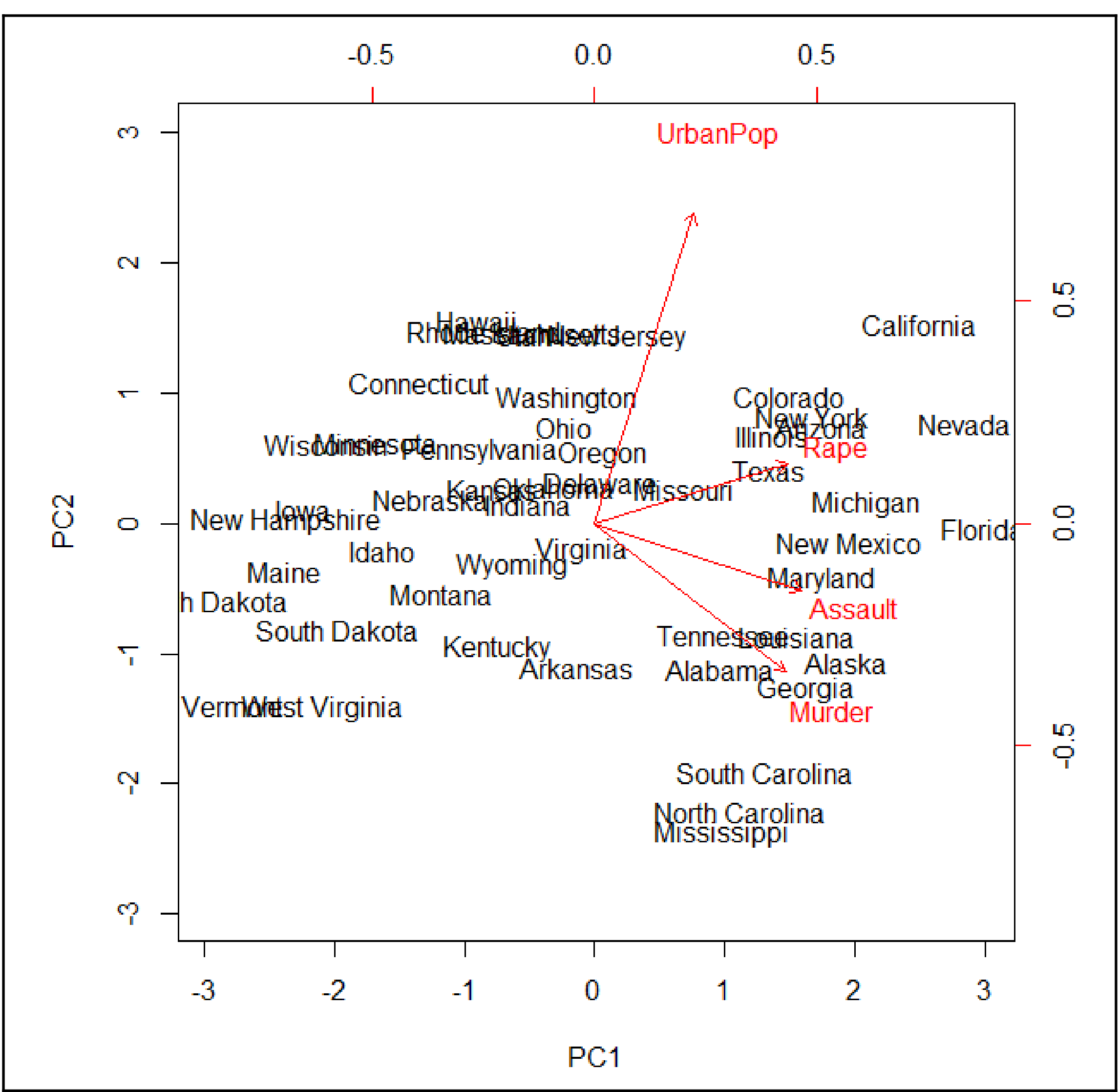
`Pca$rotation` contains the principal component loadings matrix, which explains the proportion of each variable along each principal component.

Now let's learn how to interpret the results of PCA using a biplot graph. Biplot is used to show the proportions of each variable along the two principal components.

The following code changes the directions of the biplot; if we don't include the following two lines, the plot will be a mirror image of the following one:

```
pca$rotation=-pca$rotation
pca$x=-pca$x
biplot (pca , scale =0)
```

The following image shows a plot showing principal components for the dataset:



In the previous figure, known as biplot, we can see the two principal components (PC1, PC2) of the `USArrests` dataset. The red arrows represent the loading vectors, which shows how the feature space varies along the principal component vectors.

From the plot, we observe that the first principal component vector, PC1, more or less places equal weight to three features: rape, assault, and murder. This means that these three features are more correlated with each other than with the UrbanPop feature. The second principal component, PC2, places more weight on UrbanPop than and is less correlated with the remaining three features.

Vector space models

Vector space models are algebraic models most commonly used in text analysis applications for representing text documents using the words as vectors. This is widely used in information retrieval applications. In text analysis, let's say we want to find the similarity between two sentences. How do we go about this? We know that to compute similarity measure metric, the data should be all numeric. When it comes to a sentence we have words rather than numerals. Vectors space models allow us to represent the words present in the sentences in numeric form so that we can apply any of the similarity calculation metrics, such as cosine similarity.

This representation of sentences of words in numeric form can be done in two popular ways:

- Term frequency
- Term frequency inverse document frequency

Let's understand the previously mentioned approaches with an example:

- **Sentence 1**: THE CAT CHASES RAT.
- **Sentence 2**: THE DOG CHASES CAT.
- **Sentence 3**: THE MAN WALKS ON MAT.

Given three sentences, our objective is to find the similarity between the sentences. It is clear that we cannot directly apply similarity metrics such as cosine directly. So now let's learn how to represent them in numeric format.

As a general notation, each sentence in vector space model is known as a **document**.

Term frequency

Term frequency simply means the frequency of the word in a document. To find the frequency, we need to perform the following steps:

1. The first step is to find all the unique keywords present in all the documents, represented as V:

   V = {THE, CAT, CHASES, RAT, DOG, MAN, WALKS, ON, MAT}

2. The next step is to create vectors of documents, shown as follows:

   D1 = {THE, CAT, CHASES, RAT}

   D2 = {THE, DOG, CHASES, CAT}

   D3 = {THE, MAN, WALKS, ON, MAT}

3. In this step, we have to count the term frequency of all the terms in each document:

   D1 = {(THE,1),(CAT,1),(CHASES,1),(RAT,1)}

   D2 = {(THE,1),(DOG,1),(CHASES,1),(CAT,1)}

   D3 = {(THE,1),(MAN,1),(WALKS,1),(ON,1),(MAT,1)}

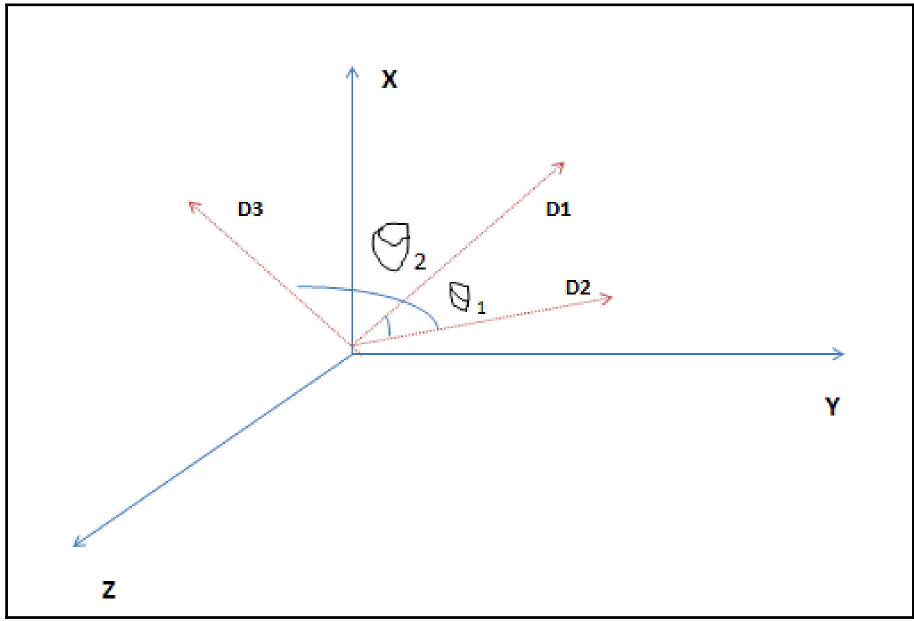
4. Now we will create a term-document matrix with document IDs as rows, unique terms as columns, and term frequency as cell values, as follows:

	THE	CAT	CHASES	RAT	DOG	MAN	WALKS	ON	MAT
D1	1	1	1	1	0	0	0	0	0
D2	1	1	1		1	0	0	0	0
D3	1	0	0	0	0	1	1	1	1

Take a while and understand what is happening here: we have put 1 in places where the word occurs in the sentences and 0 where the word does not.

Now observe that we have represented our documents in the form of a numerical matrix using the term frequency in each document.

Now, in this term-matrix document, also known as TDM, we can directly apply similarity metrics such as cosine similarity.



The previous figure visually represents the similarity between the documents after calculating the cosine angle. From the figure, we can infer that the angle between D1 and D2 is less and that between D1 and D3 is more, indicating that D1 is more similar to D2 than D3.

Term frequency inverse document frequency

The earlier approach is also known as the bag of words approach where we just have to find the frequency of each term in each document and numerically represent it in a TDM. But inherently, there is a flaw in this approach. This approach gives more weightage or importance to the terms that occur more frequently and less importance to the rarely occurring terms. It is important to understand that if a term occurs more frequently in most document collections, that term will not contribute as a differentiator in identifying a document. Similarly, a term which occurs more frequently in a document and less frequently in the whole of the document collection will contribute as a differentiator in identifying a particular document. This scaling down of weightage to more frequently occurring terms in the document collection and scaling up the weightage to more frequently occurring terms in document but less frequently in all of the document collection can be achieved using **term frequency inverse document frequency** (**tf-idf**).

The *tf-idf* can be computed as a product of term frequency of document and inverse document frequency of the term:

tf-idf = tf X idf

Here, *idf* is defined as follows:

idf = log(D/(1+ n(d,t)))

Here, *D* is the total number of document collections, and *n(d,t)* is the number of times a term *t* occurs in all the documents.

Let's compute TDM in terms of *tf-idf* for the previous set of documents (*D1*, *D2*, *D3*):

1. Take the TDM and calculate the term frequency of each term in each of the documents. This is the same as what we have done in the TF section:

	THE	CAT	CHASES	RAT	DOG	MAN	WALKS	ON	MAT
D1	1	1	1	1	0	0	0	0	0
D2	1	1	1		1	0	0	0	0
D3	1	0	0	0	0	1	1	1	1

2. In this step, we need to calculate the **document frequency** (**DF**), that is, how many times a term occurred in all the document collection. For example, let's calculate the DF for the term THE. The term is present in all the three documents, hence its DF will be 3. Similarly, for the term CAT the DF is 2:

DF	3	2	2	1	1	1	1	1	1

3. In this step, we shall calculate the inverse of document frequency (IDF) using the aforementioned formula for IDF.
    - So for the term THE, the idf will be given by: idf(THE) = log(3/(1+3)) = -0.12494

IDF	-0.12494	0	0	0.477121	0.477121	0.477121	0.477121	0.477121	0.477121

4. Compute tf-idf as the product of tf and idf for each of the terms in the whole document collection, as follows:
    - For example, for the term RAT in D1, the tf-idf will be computed as *1 X 0.4777121 = 0.4777121*

	THE	CAT	CHASES	RAT	DOG	MAN	WALKS	ON	MAT
D1	-0.12494	0	0	0.477121	0	0	0	0	0
D2	-0.12494	0	0	0	0.477121	0	0	0	0
D3	-0.12494	0	0	0	0	0.477121	0.477121	0.477121	0.477121

Now that we have calculated the *tf-idf*, we can compare the previous TDM based on *tf-df* with the TDM with *tf*. The main comparisons we can draw are in the differences in weightages for each of the terms in TDM. More frequent words across the document collection have got less weightage compared to the rarely occurred words in the document.

Now, on top of this TDM based on *tf-idf* representation, we can directly apply the similarity metric measurements.

In this section, we learned about the vector space models and *tf*, *tf-idf* concepts, which are widely used in text analysis. Now the real question is: how do we apply these techniques in recommendation engines?

Many a time, while building content-based recommendation engines, we will be getting the user preference data in text format or the features of the items as text. In such cases we might able to apply the aforementioned techniques to represent the text data as numerical vectors.

Also many times we need to find the feature importance or feature weightage to the item features while building personalized content-based recommendation engines. In such cases, vector space models concepts are very useful.

The following code snippet shows how to calculate `tfidf` in R. In the code, we use `TermDocumentMatrix()` and `weightTFidf()` to calculate the term document matrix and `tfidf` respectively available in the `tm` package in R. The `inspect()` method is used to attain the results:

```
library(tm)
data(crude)
tdm <- TermDocumentMatrix(crude,control=list(weighting =   function(x)
weightTfIdf(x, normalize =TRUE), stopwords = TRUE))
inspect(tdm)
```

The following screenshot just shows a very small portion of the large document term:

```
"for        0.000000000 0.000000000 0.000000000 0.000000000 0.000000000 0.000000000 0.000000000
"growth     0.000000000 0.000000000 0.000000000 0.000000000 0.000000000 0.000000000 0.000000000
"if         0.000000000 0.000000000 0.000000000 0.000000000 0.000000000 0.000000000 0.000000000
"is         0.000000000 0.000000000 0.015238202 0.000000000 0.000000000 0.046137890 0.000000000
"may        0.000000000 0.000000000 0.019825358 0.000000000 0.000000000 0.000000000 0.000000000
"none       0.000000000 0.000000000 0.000000000 0.000000000 0.000000000 0.000000000 0.054457838
"opec       0.000000000 0.000000000 0.000000000 0.000000000 0.000000000 0.000000000 0.000000000
"opec's     0.000000000 0.000000000 0.000000000 0.000000000 0.000000000 0.000000000 0.000000000
"our        0.000000000 0.021082576 0.000000000 0.000000000 0.000000000 0.000000000 0.000000000
"the        0.000000000 0.000000000 0.000000000 0.000000000 0.000000000 0.000000000 0.000000000
"there      0.000000000 0.000000000 0.000000000 0.000000000 0.000000000 0.000000000 0.000000000
"they       0.000000000 0.000000000 0.000000000 0.000000000 0.000000000 0.000000000 0.000000000
"this       0.000000000 0.000000000 0.000000000 0.000000000 0.000000000 0.000000000 0.000000000
```

Evaluation techniques

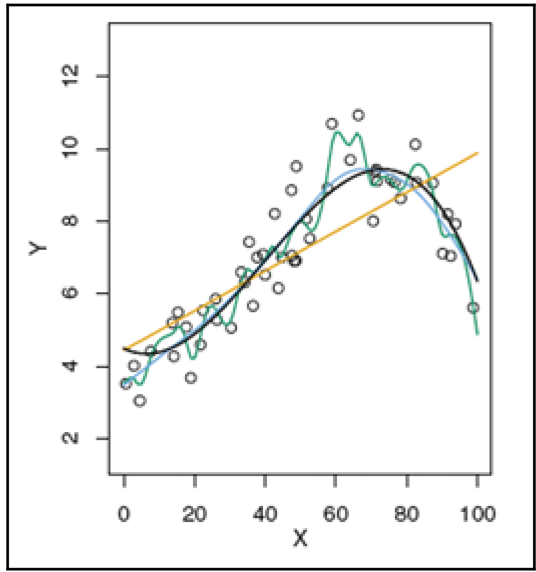
In the previous sections, we saw various data mining techniques used in recommender systems. In this section, we will learn how to evaluate models built using data mining techniques. The ultimate goal for any data analytics model is to perform well on future data. This objective can be achieved only if we build a model, which is efficient and robust during the development stage.

While evaluating any model, the most important things we need to consider are as follows:

- Whether the model is overfitting or underfitting
- How well the model fits the future data or the test data

Underfitting, also known as bias, is a scenario in which the model doesn't even perform well on training data; this means that we are fitting a less robust model to the data; for example, letting the data be distributed non-linearly and fitting it with a linear model. From the following image, we see that data is non-linearly distributed. Assume that we have fitted a linear model (orange line). In this case, during the model-building stage itself, the prediction power will be low.

Overfitting is a scenario in which the model performs well on training data but does really bad on test data. This scenario arises when the model memorizes the data pattern rather than learning from the data; for example, letting the data be distributed non-linearly and fitting a complex model (green line). In this case, we observe that the model is fitted very close to the data distribution, taking care of every up and down. In this case, the model is most likely to fail on previously unseen data:



The preceding figure shows simple, complex, and appropriately fitted models training data. The green fit represents overfitting, the orange line represents underfitting, the black and blue lines represent the appropriate model, which is a trade-off between the underfit and the overfit.

Any fitted model is evaluated to avoid the aforementioned scenarios using cross-validation, regularization, pruning, model comparisons, ROC curves, confusion matrix, and so on.

Cross-validation

This is very popular technique for model evaluation for almost all models. In this technique, we divide the original data into multiple folds/sets (say 5) of training dataset and test dataset. At each fold iteration the model is built using the training dataset and evaluated using the test dataset. This process is repeated for all the folds. The test errors are calculated for very iteration. The average test error is calculated to generalize the model accuracy at the end of all the iterations.

Cross-validation implementation is explained in `Chapter 5`, *Building Collaborative Filtering Recommendation Engines*.

Regularization

In this technique, the data variables are penalized to reduce the complexity of the model with the objective of minimizing the cost function. There are the two most popular regularization techniques: Ridge Regression and Lasso Regression. In both the techniques, we try to reduce the variable coefficients to zero so less number of variables will fit the data optimally.

The popular evaluation metrics for recommendation engines are as follows:

- Root-mean-square error (RMSE)
- Mean absolute error (MAE)
- Precision and recall

Root-mean-square error (RMSE)

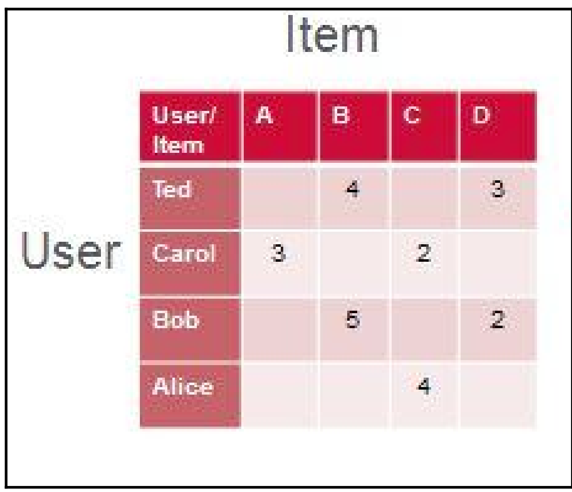
Root-mean-square error is one of the most popular, frequently used, simple measures to find the accuracy of a model. In a general sense, it is the difference between the actual and predicted values. By definition, it is the squared root of mean square error, as given by the following equation:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(X_{act} - X_{pred})^2}{n}}$$

Here, X_{act} refers to the observed values, and X_{pred} refers to the predicted values.

How is RMSE applicable to recommendation engines?

One of the core tasks in recommendation engines is to predict the preference values for non-rated items for particular users. We use many approaches discussed in previous sections to predict these non-rated preference values. Consider the following rating matrix used for building a recommendation model. Assume that our recommendation engine model has predicted all the empty cells in the following figure let they be represented as r hat. Also assume that we know the actual values of these predicted empty cells; let they be represented as r.



User/ Item	A	B	C	D
Ted		4		3
Carol	3		2	
Bob		5		2
Alice			4	

Now use the values of *r hat* and *r* in the preceding equation to calculate the model accuracy of the predictive power of the recommendation engine.

Mean absolute error (MAE)

Another popular evaluation technique for the data-mining model is **mean absolute error** (**MAE**). This evaluation metric is very similar to RMSE and is given by the following equation:

$$\mathrm{MAE} = \frac{1}{n}\sum_{i=1}^{n} |x_i - y_i|$$

This is a very simple measure computed as mean error between predicted and actual values. MAE is applied in recommendation engines as a way to evaluate the model.

Precision and recall

After we have deployed a recommendation engine in production, we will be interested only if the suggested recommendations are accepted by the users. How do we measure the effectiveness of the recommendation engine in terms of whether the model is generating valid recommendations? To measure the effectiveness, we can borrow the precision-recall evaluation technique, a popular technique in evaluating a classification model. The preceding discussion about whether a served recommendation is useful or not for a user can be treated as the binary class label of a classification model, and then we can calculate the precision-recall.

To understand precision-recall, we should understand a few more metrics that go together with precision-recall, such as true positive, true negative, false positive, and true negative.

To build what is popularly known as a confusion matrix, as shown next, let's take an example of online news recommending site, which contains 50 web pages.

Let's say we have generated 35 recommendations to user A. Of these, A has clicked on 25 suggested web pages, and 10 web pages are non-clicked. Now with this information, we create a table with the number of clicks, as follows:

- In the top-left column, enter the count of suggested links that A has clicked on
- In the top-right column, enter the count of suggested links that A has not clicked on
- In the bottom-left column, enter the count of links that A has clicked on but have not been suggested
- In the bottom-right column, enter the count of links that A has not clicked on and have not been suggested:

		Preferred	
		TRUE	FALSE
Recommended	POSITIVE	25	10
	NEGATIVE	5	10

- The top-left count is called **true positive** (**tf**), which indicates the count of all the responses where the actual response is positive and the model predicted as positive

- The top-right count is called **false positive** (**fp**), which indicates the count of all the responses where the actual response is negative but the model predicted as positive, in other words, a **FALSE ALARM**
- The bottom-left count is called **false negative** (**fn**), which indicates the count of all the responses where the actual response is positive but the model predicted as negative; in general, we call it **A MISS**
- The bottom-right count is called **true negative** (**tn**), which indicates the count of all the responses where the actual response is negative and the model predicted negative.

Take a look at the following table:

		ACTUAL	
		TRUE	**FALSE**
PREDICTED	**POSITIVE**	*TRUE POSITIVE*	*FALSE POSITIVE*
	NEGATIVE	*FALSE NEGATIVE*	*TRUE NEGATIVE*

Using the preceding information, we shall compute precision-recall metrics, as follows:

$$\text{Precision} = \frac{\#\text{tp}}{\#\text{tp} + \#\text{fp}}$$

$$\text{Recall (True Positive Rate)} = \frac{\#\text{tp}}{\#\text{tp} + \#\text{fn}}$$

Precision is calculated by true-positive divided by the sum of true-positive and false-positive. Precision indicates what per cent of the total recommendations are useful.

Recall is calculated by true-positive divided by the sum of true positive and false negative. Recall indicates of the total recommendations what percentage of recommendations is useful.

Precision and recall are both required while evaluating a recommendation model. Sometimes we would be interested in generating good recommendations with high precision, and other times we would be interested in generating recommendations with high recall. But the problem with these two is that if we focus on improving one metric, the other metric suffers. We need to choose an optimal trade-off between precision and recall based on our requirements. The implementations for precision-recall is covered in `Chapter 5`, *Building Collaborative Filtering Recommendation Engines*.

Summary

In this chapter, we saw various data-mining steps that are popularly used in building recommendation engines. We started by learning similarity calculations, such as Euclidean distance measures, followed by mathematical models, such as matrix factorization techniques. Then we covered supervised and unsupervised machine learning techniques, such as regression, classification, clustering techniques, and dimensionality reduction techniques. In the last sections of the chapter, we covered how information retrieval methods from natural language processing, such as vector space models, can be used in recommendation engines. We concluded the chapter by covering popular evaluating metrics. Till now we have covered theoretical background required for building recommendation engines. In the next chapter, we will learn building collaborative filtering recommendation engines in R and Python.

5

Building Collaborative Filtering Recommendation Engines

In this chapter, we learn how to implement collaborative filtering recommendation systems using popular data analysis programming languages, R and Python. We will learn how to implement user-based collaborative filtering and item-based collaborative filtering in R and Python programming languages.

In this chapter, we learn about:

- The Jester5k dataset we will be using for this chapter
- Exploring the dataset and understanding the data
- Recommendation engine packages/libraries available in R and Python
- Building user-based collaborative filtering in R
- Building item-based collaborative filtering in R
- Building user-based collaborative filtering in Python
- Item-based collaborative filtering in Python
- Evaluating the model

The **recommenderlab**, R package is a framework for developing and testing recommendation algorithms including user-based collaborative filtering, item-based collaborative filtering, SVD, and association rule-based algorithms, which are used to build recommendation engines. This package also provides basic infrastructure or mechanisms to develop our own recommendation engine methodology.

Installing the recommenderlab package in RStudio

The following code snippet, will install the `recommenderlab` package into RStudio, if it is not available:

```
if(!"recommenderlab" %in% rownames(installed.packages())){
install.packages("recommenderlab")}
```

First the r-environment checks if there are any previous installations of the recommender lab package, if none are found then it installs as shown below:

```
Loading required package: recommenderlab
Error in .requirePackage(package) :
  unable to find required package 'recommenderlab'
In addition: Warning message:
In library(package, lib.loc = lib.loc, character.only = TRUE,
logical.return = TRUE,  :
  there is no package called 'recommenderlab'
Loading required package: recommenderlab
install.packages("recommenderlab")
Installing package into 'path to installation folder/R/win-library/3.2'
(as 'lib' is unspecified)
trying URL
'https://cran.rstudio.com/bin/windows/contrib/3.2/recommenderlab_0.2-0.zip'
Content type 'application/zip' length 1405353 bytes (1.3 MB)
downloaded 1.3 MB
package 'recommenderlab' successfully unpacked and MD5 sums checked
```

The following code snippet using `library()` loads the `recommenderlab` package into the r-environment:

```
library(recommenderlab)

Loading required package: Matrix
Loading required package: arules

Attaching package: 'arules'

The following objects are masked from 'package:base':

    abbreviate, write

Loading required package: proxy

Attaching package: 'proxy'
```

```
The following object is masked from 'package:Matrix':

    as.matrix

The following objects are masked from 'package:stats':

    as.dist, dist

The following object is masked from 'package:base':

    as.matrix

Loading required package: registry
```

To get help on the `recommenderlab` package using the help function, run the following command in Rstudio:

```
help(package = "recommenderlab")
```

Check the help page for details on the usage of the package by clicking on the links provided:

Lab for Developing and Testing Recommender Algorithms

Documentation for package 'recommenderlab' version 0.2-0

- DESCRIPTION file.
- User guides, package vignettes and other documentation.

Help Pages

A B C D E F G H I J L M N P R S T misc

-- A --

avg	Class "evaluationResults": Results of the Evaluation of a Single Recommender Method
avg-method	Class "evaluationResultList": Results of the Evaluation of a Multiple Recommender Methods
avg-method	Class "evaluationResults": Results of the Evaluation of a Single Recommender Method

-- B --

Datasets available in the recommenderlab package

Like any other package available in R, `recommenderlab` also comes with default datasets. Run the following command to show the available packages:

```
data_package <- data(package = "recommenderlab")
data_package$results[,c("Item","Title")]
```

Item	Title
Jester5k	Jester dataset (5k sample)
JesterJokes (Jester5k)	Jester dataset (5k sample)
MSWeb	Anonymous web data from www.microsoft.com
MovieLense	MovieLense Dataset (100k)
MovieLenseMeta (MovieLense)	MovieLense Dataset (100k)

Out of all the available datasets, we have chosen to use the `Jester5k` dataset for implementing user-based collaborative filtering and item-based collaborative filtering recommendation engines using R.

Exploring the Jester5K dataset

In this section, we shall explore the `Jester5K` dataset as follows:

Description

The dataset contains a sample of 5000 users from the Jester Online Joke Recommender System anonymous ratings data, collected between April 1999 and May 2003.

Usage

```
data(Jester5k)
```

Format

The format of `Jester5k` is: `Formal class 'realRatingMatrix' [package "recommenderlab"]`.

The format of `JesterJokes` is a vector of character strings.

Details

`Jester5k` contains a *5000 x 100* rating matrix (5000 users and 100 jokes) with ratings between -10.00 and +10.00. All selected users have rated 36 or more jokes.

The data also contains the actual jokes in `JesterJokes`.

The number of ratings present in the real-rating matrix is represented as follows:

```
nratings(Jester5k)

[1] 362106

Jester5k
5000 x 100 rating matrix of class 'realRatingMatrix' with 362106 ratings.
```

You can display the class of the rating matrices by running the following command:

```
class(Jester5k)
[1] "realRatingMatrix"
attr(,"package")
[1] "recommenderlab"
```

The `recommenderlab` package efficiently stores the rating information in a compact way. Usually, rating matrices are sparse matrices. For this reason, the `realRatingMatrix` class supports a compact storage of sparse matrices.

Let's compare the size of `Jester5k` with the corresponding R matrix to understand the advantage of the real rating matrix, as follows:

```
object.size(Jester5k)
4633560 bytes
#convert the real-rating matrix into R matrix
object.size(as(Jester5k,"matrix"))
4286048 bytes
object.size(as(Jester5k, "matrix"))/object.size(Jester5k)
0.925001079083901 bytes
```

We observe that the real-rating matrix stores `0.92` times less space than the R matrix. For collaborative filtering methods, which are memory-based models, where all the data is loaded into the memory while generating recommendations, storing data efficiently is very important. The `recommenderlab` package does this job efficiently.

`The recommenderlab` package exposes many functions which can be operated on using the rating matrix object. Run the following command to see the available methods:

```
methods(class = class(Jester5k))
```

[	dimnames<-	Recommender
binarize	dissimilarity	removeKnownRatings
calcPredictionAccuracy	evaluationScheme	rowCounts
calcPredictionAccuracy	getData.frame	rowMeans
colCounts	getList	rowSds
colMeans	getNormalize	rowSums
colSds	getRatings	sample
colSums	getTopNLists	show
denormalize	image	similarity
dim	normalize	
dimnames	nratings	

Run the following commands to see the available recommendation algorithms in the `recommenderlab` package:

```
names(recommender_models)
```

Models
IBCF_realRatingMatrix
RERECOMMEND_realRatingMatrix
UBCF_realRatingMatrix
POPULAR_realRatingMatrix
RANDOM_realRatingMatrix
SVD_realRatingMatrix
SVDF_realRatingMatrix

The following code snippet displays the same result as in the previous image, `lapply()` function applies the function to all the elements of the list, in our case, for each of the items in the `recommender_models` object, `lapply` will extract the description and display the results as follows:

```
lapply(recommender_models, "[[", "description")
$IBCF_realRatingMatrix
[1] "Recommender based on item-based collaborative filtering (real data)."

$POPULAR_realRatingMatrix
[1] "Recommender based on item popularity (real data)."

$RANDOM_realRatingMatrix
[1] "Produce random recommendations (real ratings)."

$RERECOMMEND_realRatingMatrix
[1] "Re-recommends highly rated items (real ratings)."

$SVD_realRatingMatrix
[1] "Recommender based on SVD approximation with column-mean imputation
(real data)."

$SVDF_realRatingMatrix
[1] "Recommender based on Funk SVD with gradient descend (real data)."

$UBCF_realRatingMatrix
[1] "Recommender based on user-based collaborative filtering (real data)."
```

Exploring the dataset

In this section let's explore the data in more detail. To find the dimensions of the data and the type of data, run the following commands:

There are `5000` users and `100` items:

```
dim(Jester5k)

[1] 5000  100
```

The data is of R Matrix:

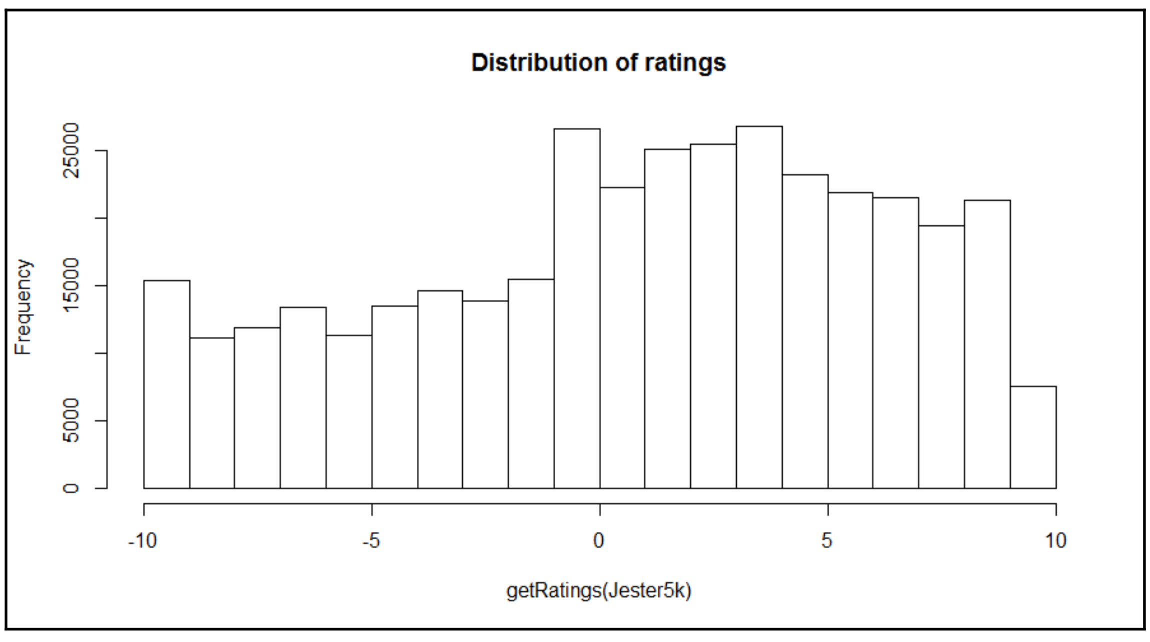
```
class(Jester5k@data)

[1] "dgCMatrix"
attr(,"package")
[1] "Matrix"
```

Exploring the rating values

The following code snippet will help us understand the rating values distribution:

Rating distribution is given as:

```
hist(getRatings(Jester5k), main="Distribution of ratings")
```



The preceding image shows the frequency of the ratings available from the `Jester5K` dataset. We can observe that the negative ratings are more or less of uniform distribution or the same frequency, and the positive ratings are of a higher frequency and are declining towards the right of the plot. This may attribute to the bias induced by the ratings given by the users.

Building user-based collaborative filtering with recommenderlab

Run the following code in order to load the `recommenderlab` library and data into the R environment:

```
library(recommenderlab)
data("Jester5k")
```

Let's look at the sample rating data of the first six users on the first 10 jokes. Run the following command:

```
head(as(Jester5k,"matrix")[,1:10])
```

	j1	j2	j3	j4	j5	j6	j7	j8	j9	j10
u2841	7.91	9.17	5.34	8.16	-8.74	7.14	8.88	-8.25	5.87	6.21
u15547	-3.20	-3.50	-9.56	-8.74	-6.36	-3.30	0.78	2.18	-8.40	-8.79
u15221	-1.70	1.21	1.55	2.77	5.58	3.06	2.72	-4.66	4.51	-3.06
u15573	-7.38	-8.93	-3.88	-7.23	-4.90	4.13	2.57	3.83	4.37	3.16
u21505	0.10	4.17	4.90	1.55	5.53	1.50	-3.79	1.94	3.59	4.81
u15994	0.83	-4.90	0.68	-7.18	0.34	-4.32	-6.17	6.12	-5.58	5.44

We have looked at exploring the data in the previous section so we will get directly to building a user-based collaborative recommender system.

This section is divided as follows:

- Building a base recommender model for benchmarking by splitting the data into 80% training data and 20% test data.
- Evaluating the recommender model using a k-fold cross-validation approach model
- Parameter tuning for the recommender model

Preparing training and test data

For building and evaluating a recommender model, we need training data and test data. Run the following command to create it:

We use the seed function for generating reproducible results:

```
set.seed(1)
which_train <- sample(x = c(TRUE, FALSE), size = nrow(Jester5k),replace =
TRUE, prob = c(0.8, 0.2))
head(which_train)
[1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE
```

The previous code creates a logical object with an equal length to the number of users. True indexes will be part of the train set and false indexes will be part of the test set.

```
rec_data_train <- Jester5k[which_train, ]
rec_data_test <- Jester5k[!which_train, ]

dim(rec_data_train)
[1] 4004  100

dim(rec_data_test)
[1] 996  100
```

Creating a user-based collaborative model

Now, let's create a recommendation model on the whole data of `Jester5k`. Before that, let's explore the recommender models available and their parameters in the `recommenderlab` package as follows:

```
recommender_models <- recommenderRegistry$get_entries(dataType =
"realRatingMatrix")

recommender_models
```

```
$IBCF_realRatingMatrix
Recommender method: IBCF
Description: Recommender based on item-based collaborative
filtering (real data).
Parameters:
   k method normalize normalize_sim_matrix alpha na_as_zero
1 30 Cosine    center                FALSE   0.5      FALSE

$POPULAR_realRatingMatrix
Recommender method: POPULAR
Description: Recommender based on item popularity (real data).
Parameters:
  normalize aggregationRatings aggregationPopularity
1    center         <function>            <function>

$RANDOM_realRatingMatrix
Recommender method: RANDOM
Description: Produce random recommendations (real ratings).
Parameters: None

$RERECOMMEND_realRatingMatrix
Recommender method: RERECOMMEND
Description: Re-recommends highly rated items (real ratings).
Parameters:
  randomize minRating
1         1        NA

$SVD_realRatingMatrix
Recommender method: SVD
Description: Recommender based on SVD approximation with column-
mean imputation (real data).
Parameters:
   k maxiter normalize
1 10     100    center

$SVDF_realRatingMatrix
Recommender method: SVDF
Description: Recommender based on Funk SVD with gradient descend
(real data).
Parameters:
   k gamma lambda min_epochs max_epochs min_improvement normalize
verbose
1 10 0.015  0.001         50        200           1e-06    center
FALSE

$UBCF_realRatingMatrix
Recommender method: UBCF
Description: Recommender based on user-based collaborative
filtering (real data).
Parameters:
  method nn sample normalize
1 cosine 25  FALSE    center
```

Image we just saw displays the 6 different recommender models available and its parameters.

Run the following code to build a user-based collaborative filtering model:

```
recc_model <- Recommender(data = rec_data_train, method = "UBCF")
recc_model

Recommender of type 'UBCF' for 'realRatingMatrix'
learned using 4004 users.
recc_model@model$data

4004 x 100 rating matrix of class 'realRatingMatrix' with 289640 ratings.
Normalized using center on rows.
```

The `recc_model@model$data` object contains the rating matrix. The reason for this is that UBCF is a lazy-learning technique, which means that it needs to access all the data to perform a prediction.

Predictions on the test set

Now that we have built the model, let's predict the recommendations on the test set. For this we will use the `predict()` function available in the library. We generate 10 recommendations per user. See the following code for the predictions:

```
n_recommended <- 10
recc_predicted <- predict(object = recc_model,newdata = rec_data_test, n =
n_recommended)
recc_predicted
Recommendations as 'topNList' with n = 10 for 996 users.

#Let's define list of predicted recommendations:
rec_list <- sapply(recc_predicted@items, function(x){
  colnames(Jester5k)[x]
})
```

The resultant object is a list type given by the following code:

```
class(rec_list)
[1] "list"
```

The first two recommendations are given as follows:

```
rec_list [1:2]
$u21505
 [1] "j81"  "j73"  "j83"  "j75"  "j100" "j80"  "j72"  "j95"  "j87"  "j96"

$u5809
 [1] "j97" "j93" "j76" "j78" "j77" "j85" "j89" "j98" "j91" "j80"
```

We can observe that for user `u21505`, the top 10 recommendations are given as `j81, j73, j83, ... j96`.

The following image shows the recommendations for four users:

Users	Recommended Jokes
u21505	"j81" "j73" "j83" "j75" "j100" "j80" "j72" "j95" "j87" "j96"
u5809	"j97" "j93" "j76" "j78" "j77" "j85" "j89" "j98" "j91" "j80"
u12519	"j98" "j100" "j80" "j93" "j99" "j87" "j76" "j89" "j84" "j96"
u12094	"j89" "j96" "j78" "j94" "j88" "j86" "j87" "j93" "j91" "j99"

Let's see how many recommendations are generated for all the test users by running the following code:

```
number_of_items = sort(unlist(lapply(rec_list, length)),decreasing = TRUE)
table(number_of_items)

0   1   2   3   4   5   6   7   8   9  10
286   3   2   3   3   1   1   1   2   3 691
```

From the above results, we see that for `286` users, zero recommendations were generated. The reason is that they have rated all the movies in the original dataset. For 691 users, 10 ratings for each of them has been generated, the reason is that in the original dataset, they have not rated for any movies. Other users who have received 2, 3, 4, and so on recommendations means that they have recommended very few movies.

Analyzing the dataset

Before we evaluate the model, let's take one step back and analyze the data. By analyzing the number of ratings given by all the users for the jokes, we can observe that there are 1422 people who have rated all 100 jokes, which seems to be unusual as there are very few people who have rated 80 to 99 jokes. Further analyzing the jokes we find that, there are 221, 364, 312, and 131 users who have rated 71, 72, 73, and 74 jokes respectively which seems to be unusual compared to other joke ratings.

Run the following code to extract the number of ratings given to each joke:

```
table(rowCounts(Jester5k))
```

36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
114	77	80	78	74	70	74	75	80	81	70	60	69	61	62	47	42	52	52	48	56	54	34	43
60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83
48	41	42	42	41	53	51	39	39	29	87	221	364	312	131	48	36	19	33	32	38	20	19	18
84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100							
15	23	21	12	12	11	14	16	10	12	12	11	16	13	8	16	1422							

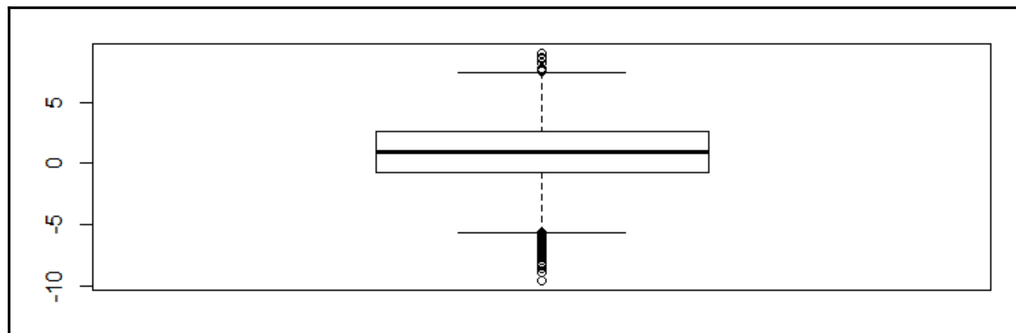
For the next step, let's remove the records of users who have rated 80 or more jokes as follows:

```
model_data = Jester5k[rowCounts(Jester5k) < 80]
dim(model_data)
[1] 3261  100
```

The dimension has been reduced from `5000` to `3261` records.

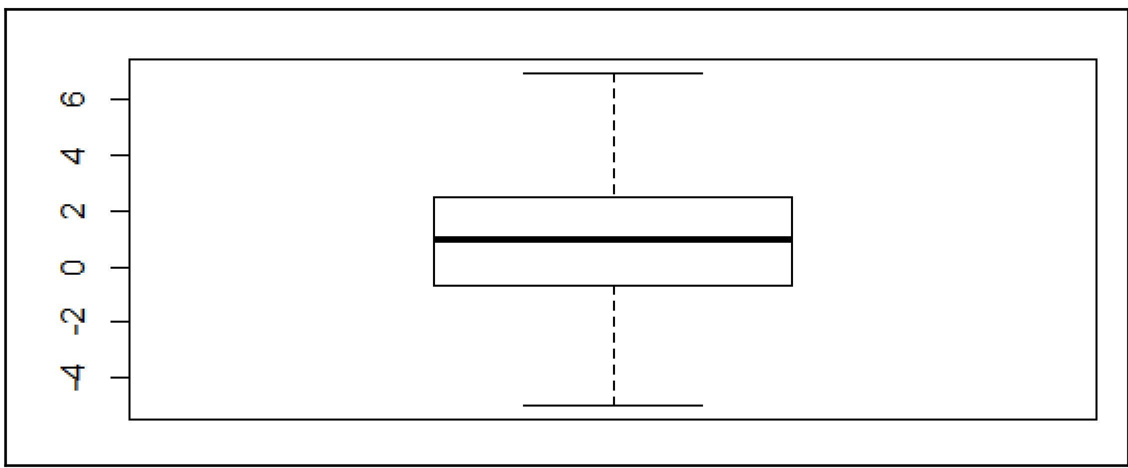
Now let's analyze the average ratings given by each user. A boxplot shows us the average distribution of the joke ratings.

```
boxplot(model_data)
```



The preceding image shows us that there are very few ratings that deviate from normal behavior. From the preceding image we see that the average ratings that are above 7 (approximately) and below -5 (approximately) are kind of outliers and are less in number. Let's see the count by running the following code:

```
boxplot(rowMeans(model_data [rowMeans(model_data)>=-5 &
rowMeans(model_data)<= 7]))
```

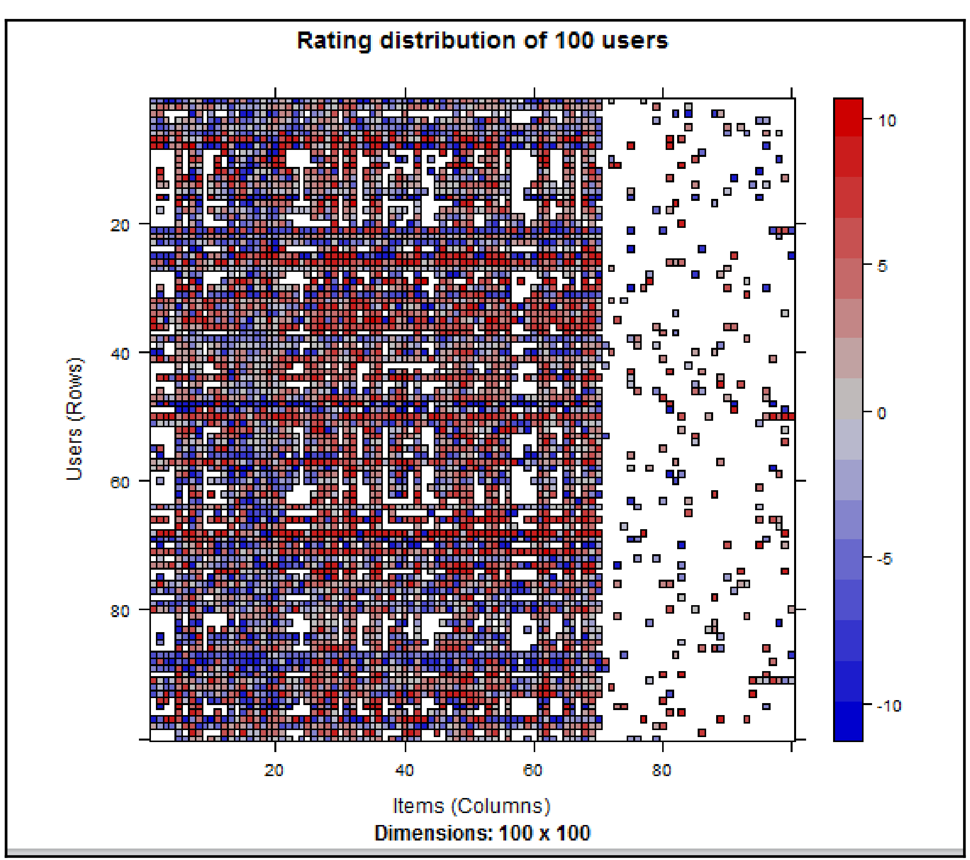


Dropping users who have given very low average ratings and very high average ratings.

```
model_data = model_data [rowMeans(model_data)>=-5 & rowMeans(model_data)<=
7]
dim(model_data)
[1] 3163  100
```

Let's examine the rating distribution of the first 100 users in the data as follows:

```
image(model_data, main = "Rating distribution of 100 users")
```



Evaluating the recommendation model using the k-cross validation

The `recommenderlab` package provides an infrastructure for evaluating the models using the `evaluationScheme()` function. By definition, from the Cran website, evaluationScheme *creates an evaluationScheme object from a data set. The scheme can be a simple split into training and test data, k-fold cross-evaluation or using k independent bootstrap samples*.

The following are the arguments for the `evaluationScheme()` function:

data	data as ratingMatrix
method	Split/Cross-validation/bootstrap
k	number of nearest neighbours to be considered for similarity calculation
goodRating	Minimum value for considering as good rating
given	Minimum number of records each row should contain

We use cross-validation method to split the data, for example the 5-fold cross-validation approach divides the training data into five smaller sets where four sets will be used for training the model and the remaining one set is used for evaluating the model. Let's define the parameters into minimum good ratings, number of folds for cross-validation method, and split method as follows:

```
items_to_keep <- 30
rating_threshold <- 3
n_fold <- 5 # 5-fold
eval_sets <- evaluationScheme(data = model_data, method = "cross-
validation",train = percentage_training, given = items_to_keep, goodRating
= rating_threshold, k = n_fold)

Evaluation scheme with 30 items given
Method: 'cross-validation' with 5 run(s).
Good ratings: >=3.000000
Data set: 3163 x 100 rating matrix of class 'realRatingMatrix' with 186086
ratings.
```

Let's examine the size of the five sets formed by the cross-validation approach as follows:

```
size_sets <- sapply(eval_sets@runsTrain, length)
 size_sets
[1] 2528 2528 2528 2528 2528
```

In order to extract the sets, we need to use `getData()`. There are three sets:

- **train**: This is the training set
- **known**: This is the test set, with the item used to build the recommendations
- **unknown**: This is the test set, with the item used to test the recommendations

Let's take a look at the training set in the following code:

```
getData(eval_sets, "train")
2528 x 100 rating matrix of class 'realRatingMatrix' with 149308 ratings.
```

Evaluating user-based collaborative filtering

Now let's evaluate the models, let's set the parameters `model_to_evaluate` with user-based collaborative filtering and `model_parameters` with `NULL` for using default settings as follows:

```
model_to_evaluate <- "UBCF"
model_parameters <- NULL
```

The next step is to build the recommender model using the recommender() function as follows:

```
eval_recommender <- Recommender(data = getData(eval_sets, "train"),method =
model_to_evaluate, parameter = model_parameters)

Recommender of type 'UBCF' for 'realRatingMatrix'
learned using 2528 users
```

We have seen that the user-based recommender model has been learned with a training data of `2528 users`. Now we can predict the known ratings in `eval_sets` and evaluate the results with unknown sets as described earlier.

Before making the predictions for the known ratings, we have to set the number of items to be recommended. Next, we have to provide the test set to the `predict()` function for prediction. The prediction of ratings is done by running the following command:

```
items_to_recommend <- 10
eval_prediction <- predict(object = eval_recommender, newdata
=getData(eval_sets, "known"), n = items_to_recommend, type = "ratings")

eval_prediction
635 x 100 rating matrix of class 'realRatingMatrix' with 44450 ratings
```

Executing the `predict()` function will take time because the user-based collaborative filtering approach is memory-based and a lazy learning technique implemented at run time, to show that the whole dataset is loaded during the prediction.

Now we shall evaluate the predictions with the unknown sets and estimate the model accuracy with metrics such as precision, recall, and F1 measure. Run the following code to calculate the model accuracy metrics by calling the `calcPredicitonAccuracy()` method:

```
eval_accuracy <- calcPredictionAccuracy(  x = eval_prediction, data =
getData(eval_sets, "unknown"), byUser = TRUE)
head(eval_accuracy)
           RMSE       MSE      MAE
u17322 4.536747 20.582076 3.700842
u13610 4.609735 21.249655 4.117302
u5462  4.581905 20.993858 3.714604
u1143  2.178512  4.745912 1.850230
u5021  2.664819  7.101260 1.988018
u21146 2.858657  8.171922 2.194978
```

By setting `byUser = TRUE`, we are calculating the model accuracy for each user. Taking the average will give us the overall accuracy as follows:

```
apply(eval_accuracy,2,mean)
     RMSE       MSE       MAE
 4.098122 18.779567  3.377653
```

By setting `byUser=FALSE`, in the previous `calcPredictionAccuracy()` we can calculate the overall model accuracy given by the following:

```
eval_accuracy <- calcPredictionAccuracy(  x = eval_prediction, data =
getData(eval_sets, "unknown"), byUser =
    FALSE)

eval_accuracy
    RMSE       MSE       MAE
 4.372435 19.118191  3.431580
```

In the previous approach, we evaluated the model accuracy using the **root mean squared error** (**RMSE**), and **mean absolute error** (**MAE**), but we can also evaluate the model accuracy using precision/recall For this, we use the `evaluate()` function and then the result of the `evaluate()` method is used to create a confusion matrix containing precision/recall/f1 measures as follows:

```
results <- evaluate(x = eval_sets, method = model_to_evaluate, n = seq(10,
100, 10))
```

```
UBCF run fold/sample [model time/prediction time]
 1  [0.01sec/19.64sec]
 2  [0.02sec/19.6sec]
 3  [0.02sec/19.68sec]
 4  [0.02sec/19.45sec]
 5  [0.03sec/19.01sec]
```

```
head(getConfusionMatrix(results)[[1]])
```

```
         TP        FP        FN        TN precision    recall       TPR
FPR
10  6.63622  3.363780 10.714961 49.285039 0.6636220 0.4490838 0.4490838
0.05848556
20 10.03150  9.968504  7.319685 42.680315 0.5015748 0.6142384 0.6142384
0.17854766
30 11.20787 18.792126  6.143307 33.856693 0.3735958 0.6714050 0.6714050
0.34877101
40 11.91181 28.088189  5.439370 24.560630 0.2977953 0.7106378 0.7106378
0.53041204
50 12.96850 37.031496  4.382677 15.617323 0.2593701 0.7679658 0.7679658
0.70444585
60 14.82362 45.176378  2.527559  7.472441 0.2470604 0.8567522 0.8567522
0.85919995
```

The first four columns contain the true-false positives/negatives, and they are as follows:

- **True Positives** (**TP**): These are recommended items that have been rated correctly
- **False Positives** (**FP**): These are recommended items that haven't been rated
- **False Negatives** (**FN**): These are not recommended items that have been rated
- **True Negatives** (**TN**): These are not recommended items that haven't been rated

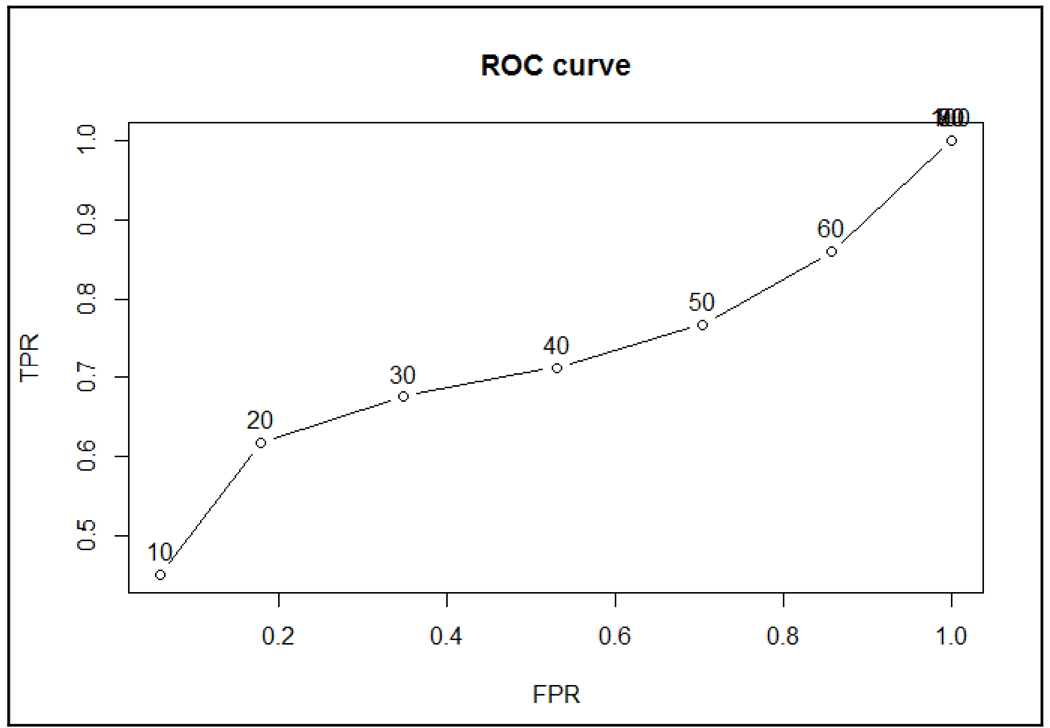
A perfect (or overfitted) model would have only `TP` and `TN`.

If we want to take account of all the splits at the same time, we can just sum up the indices as follows:

```
columns_to_sum <- c("TP", "FP", "FN", "TN")
indices_summed <- Reduce("+", getConfusionMatrix(results))[,
columns_to_sum]
head(indices_summed)
         TP        FP       FN        TN
10 32.59528  17.40472 53.22520 246.77480
20 49.55276  50.44724 36.26772 213.73228
30 55.60787  94.39213 30.21260 169.78740
40 59.04724 140.95276 26.77323 123.22677
50 64.22205 185.77795 21.59843  78.40157
60 73.67717 226.32283 12.14331  37.85669
```

Since summarizing the model is difficult by referring to the above table, we can use a `ROC curve` to evaluate the model. Use `plot()` to build the ROC plot as follows:

```
plot(results, annotate = TRUE, main = "ROC curve")
```



The preceding plot shows the relation between **True Positive Rate** (**TPR**) and **False Positive Rate** (**FPR**), but we have to choose the values in such a way that we give a trade-off between TPR and FPR. In our case, we observe that *nn=30* is a very good trade-off since when considering neighbors of 30 we have TPR closer to *0.7*, FPR is *0.4* and when moving to *nn=40* the TPR is still close to *0.7* but the FPR has been changed to *0.4.* This means that the False Positive Rate has been increased.

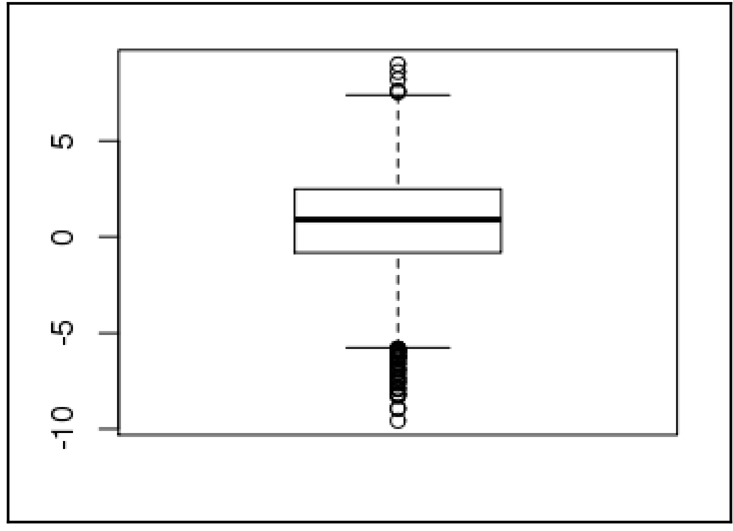
Building an item-based recommender model

As with UBCF, we use the same `Jester5k` dataset for the item-based recommender system. In this section, we do not explore the data as we have already done so in the previous section. We first remove the user data of those who have rated all the items and also those records who have rated more than `80` as follows:

```
library(recommenderlab)
data("Jester5k")
model_data = Jester5k[rowCounts(Jester5k) < 80]
model_data
[1] 3261  100
```

Now let's see how the average ratings are distributed for each user:

```
boxplot(rowMeans(model_data))
```



The following code snippet calculates the average ratings given by each user and identifies users who have given extreme ratings – either very high ratings or very low ratings:

From the below results, we observe that there are 19 records with very high average ratings and 79 records with very low ratings, compared with the majority of users:

```
dim(model_data[rowMeans(model_data) < -5])
[1]  79 100
dim(model_data[rowMeans(model_data) > 7])
[1]  19 100
```

Of the total `3261` records, only `98` records had much less than the average and much more than the average ratings, so we removed these from our dataset as follows:

```
model_data = model_data [rowMeans(model_data)>=-5 & rowMeans(model_data)<=
7]
model_data
[1] 3163  100
```

From here, we divide the sections as follows:

- Building the IBCF recommender model using the training and test data.
- Evaluating the model
- Parameter tuning

Building an IBCF recommender model

The first step in building any recommender model is to prepare the training data. Previously, we have prepared the data required for building the model by removing outlying data. Now run the following code to divide the available data into two sets: 80% training set and 20% test set. We build the recommender model using the training data and generate the recommendations on the test set.

The following code first creates a logical object of the same length as the original dataset containing 80% elements as TRUE and 20% as a test:

```
which_train <- sample(x = c(TRUE, FALSE), size = nrow(model_data),
 replace = TRUE, prob = c(0.8, 0.2))
class(which_train)
[1] "logical"
head(which_train)
[1] TRUE TRUE TRUE TRUE TRUE TRUE
```

Then we use the logical object in the `model_data` to generate the training set as follows:

```
model_data_train <- model_data[which_train, ]
dim(model_data_train)
[1] 2506  100
```

Then we use the logical object in the `model_data` to generate the test set as follows:

```
model_data_test <- model_data[!which_train, ]
dim(model_data_test)
[1] 657 100
```

Now that we have prepared the training set and the test set, let's train the model and generate the top recommendations on the test set.

For model building, as mentioned in the UBCF section, we use the same `recommender()` function available in the `recommenderlab` package. Run the following code to train the model with training data.

Set the parameters for the `recommender()` function. We set the model to evaluate as `"IBCF"` and `k=30`. `k` is the number of neighbors to be considered while calculating the similarity values as follows:

```
model_to_evaluate <- "IBCF"

model_parameters <- list(k = 30)
```

The following code snippet shows building the recommendation engine model using the `recommender()` function and its input parameters such as input data, model to evaluate the parameters, and the k parameter:

```
model_recommender <- Recommender(data = model_data_train,method =
model_to_evaluate, parameter = model_parameters)
```

The IBCF model object is created as a `model_recommender`. This model is trained and learned using the `2506` training set we created earlier as follows:

```
model_recommender
Recommender of type 'IBCF' for 'realRatingMatrix'  learned using 2506
users.
```

Now that we have created the model, let's explore the model bit. We use `getModel()` available in the `recommenderlab` to extract the model details as follows:

```
model_details = getModel(model_recommender)
str(model_details)
List of 10
 $ description         : chr "IBCF: Reduced similarity matrix"
 $ sim                 :Formal class 'dgCMatrix' [package
"Matrix"] with 6 slots
  .. ..@ i       : int [1:3000] 1 2 3 9 10 23 24 32 36 37 ...
  .. ..@ p       : int [1:101] 0 20 41 66 101 126 162 192 226 262
...
  .. ..@ Dim     : int [1:2] 100 100
  .. ..@ Dimnames:List of 2
  .. .. ..$ : chr [1:100] "j1" "j2" "j3" "j4" ...
  .. .. ..$ : chr [1:100] "j1" "j2" "j3" "j4" ...
  .. ..@ x       : num [1:3000] 0.1785 0.1735 0.0448 0.0833 0.0272
...
  .. ..@ factors : list()
 $ k                   : num 30
 $ method              : chr "Cosine"
 $ normalize           : chr "center"
 $ normalize_sim_matrix: logi FALSE
 $ alpha               : num 0.5
 $ na_as_zero          : logi FALSE
 $ minRating           : logi NA
 $ verbose             : logi FALSE
```

From the above results, the important parameters to note are `k` value, the default similarity value, and method, `cosine similarity`.

The final step is to generate the recommendations on the test set. Run the following code on the test set and generate the recommendations.

`items_to_recommend` is the parameter to set the number of recommendations to be generated for each user:

```
items_to_recommend <- 10
```

Call the `predict()` method available in the reocommenderlab package to predict the unknowns in the test set:

```
model_prediction <- predict(object = model_recommender, newdata =
model_data_test, n = items_to_recommend)

model_prediction
Recommendations as 'topNList' with n = 10 for 657 users.

print(class(model_prediction))
[1] "topNList"
attr(,"package")
[1] "recommenderlab"
```

We can get the slot details of the prediction object using the `slotNames()` method:

```
slotNames(model_prediction)
[1] "items"      "itemLabels" "n"
```

Let's have a look of the predictions generated for the first user in the test set:

```
model_prediction@items[[1]]
[1]  89  76  72  87  93 100  97  80  94  86
```

Let's add the item labels to each of the predictions:

```
recc_user_1  = model_prediction@items[[1]]

jokes_user_1 <- model_prediction@itemLabels[recc_user_1]

jokes_user_1
[1] "j89"  "j76"  "j72"  "j87"  "j93"  "j100" "j97"  "j80"  "j94"  "j86"
```

Model evaluation

Let's take one step back to evaluate the recommender model before we generate the predictions. As we saw in UBCF, we can use the available `evaluationScheme()` method. We use the cross-validation setting to generate the training and test sets. Then we make predictions on each test set and evaluate the model accuracy.

Run the following code to generate the training and test sets.

`n_fold` defines the 4-fold cross-validation, that divides the data into 4 sets; 3 training sets and 1 test set:

```
n_fold <- 4
```

`items_to_keep` defines the minimum number of items to use to generate recommendations:

```
items_to_keep <- 15
```

`rating_threshold` defines the minimum rating which is considered as a good rating:

```
rating_threshold <- 3
```

`evaluationScheme` method creates the test sets:

```
eval_sets <- evaluationScheme(data = model_data, method = "cross-
validation",k = n_fold, given = items_to_keep, goodRating
=rating_threshold)
size_sets <- sapply(eval_sets@runsTrain, length)
size_sets
[1] 2370 2370 2370 2370
```

Set the `model_to_evaluate` to set the recommender method to be used. `model_parameters` defines the model parameters such as the number of neighbors to be considered while computing the similarity using cosine. For now we will set it as `NULL` in order to make the model choose the default values, as follows:

```
model_to_evaluate <- "IBCF"
model_parameters <- NULL
```

Use the `recommender()` method to generate the model. Let's understand each parameter of the `recommender()` method:

`getData` extracts the training data from `eval_sets` and passes it on to the `recommender()` method as follows:

```
getData(eval_sets,"train")
2370 x 100 rating matrix of class 'realRatingMatrix' with 139148 ratings
```

Since we are using 4-folds cross-validation, the `recommender()` method uses the three sets from `eval_sets` for training and the remaining one set for testing/ evaluating the model as follows:

```
eval_recommender <- Recommender(data = getData(eval_sets, "train"),method =
model_to_evaluate, parameter = model_parameters)
#setting the number of items to be set for recommendations
items_to_recommend <- 10
```

Now we use the built model to make predictions on a `"known"` dataset from `eval_sets`. As seen before, we use the `predict()` method to generate the predictions as follows:

```
eval_prediction <- predict(object = eval_recommender, newdata =
getData(eval_sets, "known"), n = items_to_recommend, type = "ratings")

class(eval_prediction)
[1] "realRatingMatrix"
attr(,"package")
[1] "recommenderlab"
```

Model accuracy using metrics

Until now, the procedure has been the same as for making the initial predictions, now we will see how to evaluate the model accuracy for the predictions made on the "known" set of test data from `eval_sets`. As we saw in the UBCF section, we use the `calcPredictionAccuracy()` method to calculate the prediction accuracy.

We use the `calcPredictionAccuracy()` method and pass the `"unknown"` dataset available in the `eval_sets` as follows:

```
eval_accuracy <- calcPredictionAccuracy(x = eval_prediction, data =
getData(eval_sets, "unknown"), byUser = TRUE)

head(eval_accuracy)
           RMSE      MSE      MAE
u238   4.625542 21.39564  4.257505
u17322  4.953789  24.54003  3.893797
u5462  4.685714   21.95591  4.093891
u13120   4.977421  24.77472  4.261627
u12519   3.875182  15.01703  2.750987
u17883 7.660785 58.68762 6.595489
```

Using `byUser = TRUE` in the previous method calculates the accuracy for each user. In the table above we can see that for user – `u238` the `RMSE` is `4.62` and `MAE` is `4.25`

If we want to see the accuracy of the whole model, just calculate the mean of each column, that is to say the average for all the users as follows:

```
apply(eval_accuracy,2,mean)
  RMSE      MSE      MAE
 4.45511 21.94246  3.56437
```

By setting `byUser=FALSE` we can calculate the model accuracy for the whole model:

```
eval_accuracy <- calcPredictionAccuracy(x = eval_prediction, data =
getData(eval_sets, "unknown"), byUser = FALSE)

eval_accuracy
     RMSE       MSE       MAE
 4.672386 21.831190  3.555721
```

Model accuracy using plots

Now we can see the model accuracy using Precision-Recall, ROC curves, and precision/recall curves. These curves help us to decide the trade-off between Precision-Recall while choosing the parameters we use for the recommender models, IBCF in our case.

We use the `evaluate()` method and then set the n value which defines the number of nearest neighbors while calculating the similarities between items as follows:

Running the following evaluate method makes the model run four times for each dataset:

```
results <- evaluate(x = eval_sets, method = model_to_evaluate, n =
seq(10,100,10))
IBCF run fold/sample [model time/prediction time]
 1  [0.145sec/0.327sec]
 2  [0.139sec/0.32sec]
 3  [0.139sec/0.32sec]
 4  [0.137sec/0.322sec]
```

Let's see the model accuracy at each fold:

```
results@results[1]
```

```
          TP        FP          FN          TN precision    recall
TPR
10   2.923077 7.076923 14.48675914 60.5132409 0.2923077 0.1735253
0.1735253
20   6.508197 13.491803 10.90163934 54.0983607 0.3254098 0.3967540
0.3967540
30   9.388398 20.611602 8.02143758 46.9785624 0.3129466 0.5636088
0.5636088
40 11.419924 28.580076  5.98991173 39.0100883 0.2854981 0.6759040
0.6759040
50 13.119798 36.880202  4.29003783 30.7099622 0.2623960 0.7702400
0.7702400
60 14.602774 45.397226  2.80706179 22.1929382 0.2433796 0.8503495
0.8503495
70 15.876419 54.116015 1.53341740 13.4741488 0.2268279 0.9184538
0.9184538
80 16.941992 63.017654  0.46784363  4.5725095 0.2118578 0.9764965
0.9764965
90  17.398487 67.229508  0.01134931  0.3606557 0.2054721 0.9992819
0.9992819
100 17.398487 67.229508  0.01134931  0.3606557 0.2054721 0.9992819
0.9992819
          FPR
10  0.1037828
20  0.1969269
40  0.4184371
50  0.5414312
60  0.6679105
70  0.7979464
80  0.9314466
90  0.9947886
100 0.9947886
```

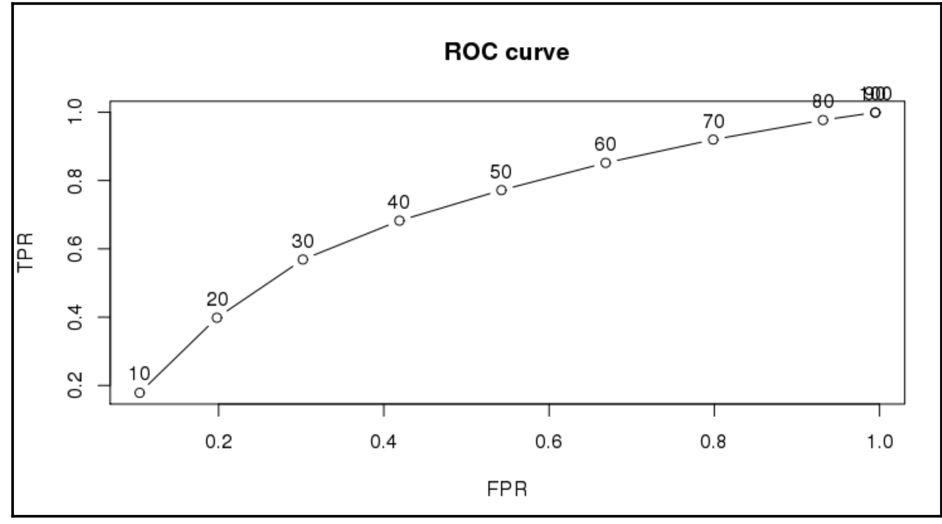
Let's sum up all the 4-fold results using the following code:

```
columns_to_sum <- c("TP", "FP", "FN", "TN","precision","recall")
indices_summed <- Reduce("+", getConfusionMatrix(results))[,
columns_to_sum]
```

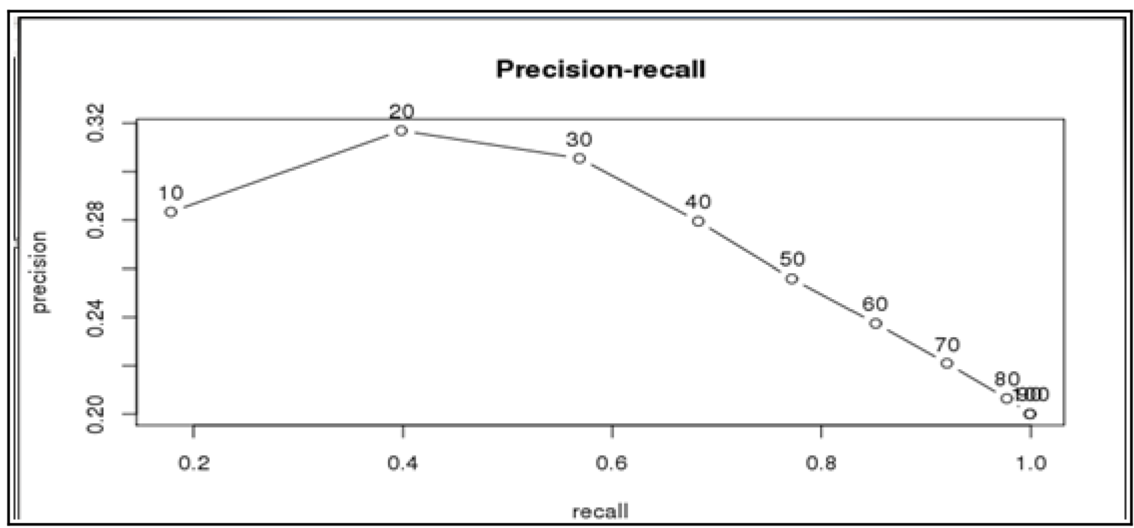
```
head(indices_summed)
         TP        FP       FN        TN precision    recall
10 11.33291 28.66709 56.47415 243.52585 1.1332913 0.7137918
20 25.35687 54.64313 42.45019 217.54981 1.2678436 1.5929334
30 36.65700 83.34300 31.15006 188.84994 1.2219000 2.2746946
40 44.71248 115.28752 23.09458 156.90542 1.1178121 2.7293984
50 51.17528 148.82472 16.63178 123.36822 1.0235057 3.0871703
60 56.98613 183.01387 10.82093 89.17907 0.9497688 3.4062870
```

From the previous table, we can observe that the model accuracy, Precision-Recall values are good for n values of 30 and 40. The same results can be visually inferred using ROC curves and Precision-Recall plots as follows:

```
plot(results, annotate = TRUE, main = "ROC curve")
```



```
plot(results, "prec/rec", annotate = TRUE, main = "Precision-recall")
```



Parameter tuning for IBCF

While building the IBCF model there are a few places where we can choose the optimal values before we generate recommendations for building a final model:

- We have to choose, optimal number of neighbors for calculating the similarities between items
- Similarity method to be used, whether it is the cosine or Pearson method

See the following steps:

First set the different k-values:

```
vector_k <- c(5, 10, 20, 30, 40)
```

Use `lapply` to generate different models using the cosine method and different values of k:

```
 model1 <- lapply(vector_k, function(k,l){ list(name = "IBCF", param =
list(method = "cosine", k = k)) })
names(model1) <- paste0("IBCF_cos_k_", vector_k)
names(model1) [1] "IBCF_cos_k_5" "IBCF_cos_k_10" "IBCF_cos_k_20"
"IBCF_cos_k_30" [5] "IBCF_cos_k_40" #use Pearson method for similarities
model2 <- lapply(vector_k, function(k,l){ list(name = "IBCF", param =
list(method = "pearson", k = k)) })
```

```
names(model2) <- paste0("IBCF_pea_k_", vector_k)
names(model2) [1] "IBCF_pea_k_5" "IBCF_pea_k_10" "IBCF_pea_k_20"
"IBCF_pea_k_30" [5] "IBCF_pea_k_40"
#now let's combine all the methods:
models = append(model1,model2)
```

$IBCF_cos_k_5	$IBCF_cos_k_30	$IBCF_pea_k_5	$IBCF_pea_k_20	$IBCF_pea_k_40
$IBCF_cos_k_5$name	$IBCF_cos_k_30$name	$IBCF_pea_k_5$name	$IBCF_pea_k_20$name	$IBCF_pea_k_40$name
[1] "IBCF"	[1] "IBCF"	[1] "IBCF"	[1] "IBCF"	[1] "IBCF"
$IBCF_cos_k_5$param	$IBCF_cos_k_30$param	$IBCF_pea_k_5$param	$IBCF_pea_k_20$param	$IBCF_pea_k_40$param
$IBCF_cos_k_5$param$method	$IBCF_cos_k_30$param$method	$IBCF_pea_k_5$param$method	$IBCF_pea_k_20$param$method	$IBCF_pea_k_40$param$method
[1] "cosine"	[1] "cosine"	[1] "pearson"	[1] "pearson"	[1] "pearson"
$IBCF_cos_k_5$param$k	$IBCF_cos_k_30$param$k	$IBCF_pea_k_5$param$k	$IBCF_pea_k_20$param$k	$IBCF_pea_k_40$param$k
[1] 5	[1] 30	[1] 5	[1] 20	[1] 40
$IBCF_cos_k_10	$IBCF_cos_k_40	$IBCF_pea_k_10	$IBCF_pea_k_30	$IBCF_cos_k_20
$IBCF_cos_k_10$name	$IBCF_cos_k_40$name	$IBCF_pea_k_10$name	$IBCF_pea_k_30$name	$IBCF_cos_k_20$name
[1] "IBCF"	[1] "IBCF"	[1] "IBCF"	[1] "IBCF"	[1] "IBCF"
$IBCF_cos_k_10$param	$IBCF_cos_k_40$param	$IBCF_pea_k_10$param	$IBCF_pea_k_30$param	$IBCF_cos_k_20$param
$IBCF_cos_k_10$param$method	$IBCF_cos_k_40$param$method	$IBCF_pea_k_10$param$method	$IBCF_pea_k_30$param$method	$IBCF_cos_k_20$param$method
[1] "cosine"	[1] "cosine"	[1] "pearson"	[1] "pearson"	[1] "cosine"
$IBCF_cos_k_10$param$k	$IBCF_cos_k_40$param$k	$IBCF_pea_k_10$param$k	$IBCF_pea_k_30$param$k	$IBCF_cos_k_20$param$k
[1] 10	[1] 40	[1] 10	[1] 30	[1] 20

Set the total number of recommendations to be generated:

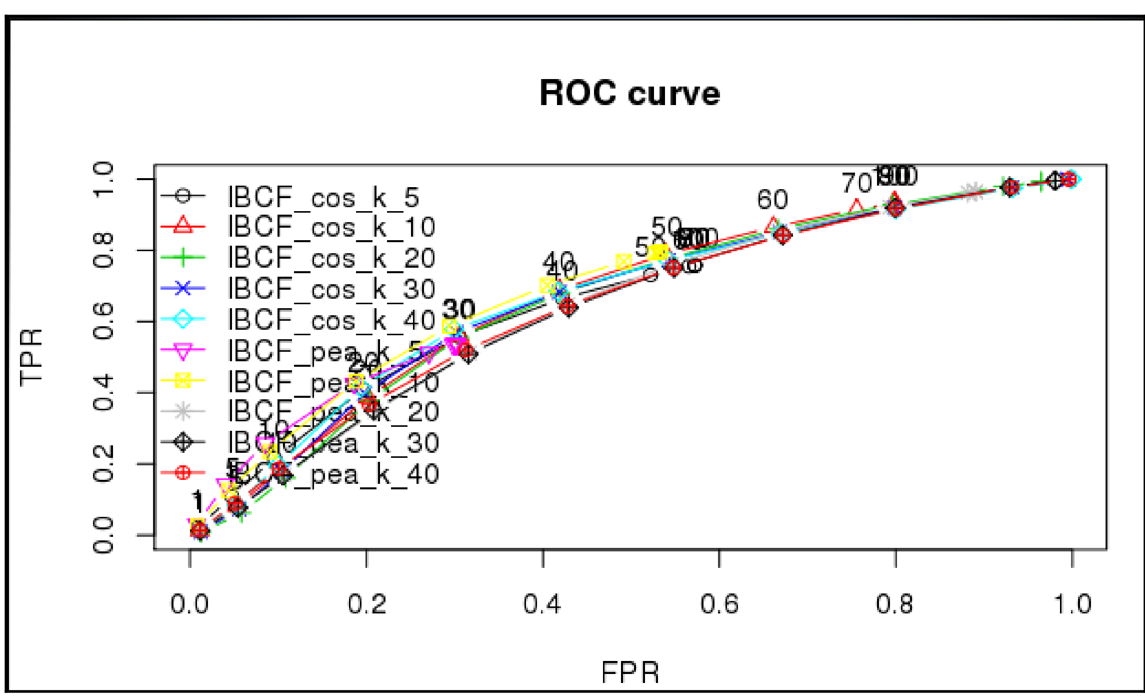
```
n_recommendations <- c(1, 5, seq(10, 100, 10))
```

Call the evaluate method to the build 4-fold methods:

```
 list_results <- evaluate(x = eval_sets, method = models, n=
n_recommendations)
IBCF run fold/sample [model time/prediction time] 1 [0.139sec/0.311sec] 2
[0.143sec/0.309sec] 3 [0.141sec/0.306sec] 4 [0.153sec/0.312sec]
IBCF run fold/sample [model time/prediction time] 1 [0.141sec/0.326sec] 2
[0.145sec/0.445sec] 3 [0.147sec/0.387sec] 4 [0.133sec/0.439sec]
IBCF run fold/sample [model time/prediction time] 1 [0.14sec/0.332sec] 2
[0.16sec/0.327sec] 3 [0.139sec/0.331sec] 4 [0.138sec/0.339sec] IBCF run
fold/sample [model time/prediction time] 1 [0.139sec/0.341sec] 2
[0.157sec/0.324sec] 3 [0.144sec/0.327sec] 4 [0.133sec/0.326sec]
```

Now that we have got the results, let's plot and choose the optimal parameters as follows:

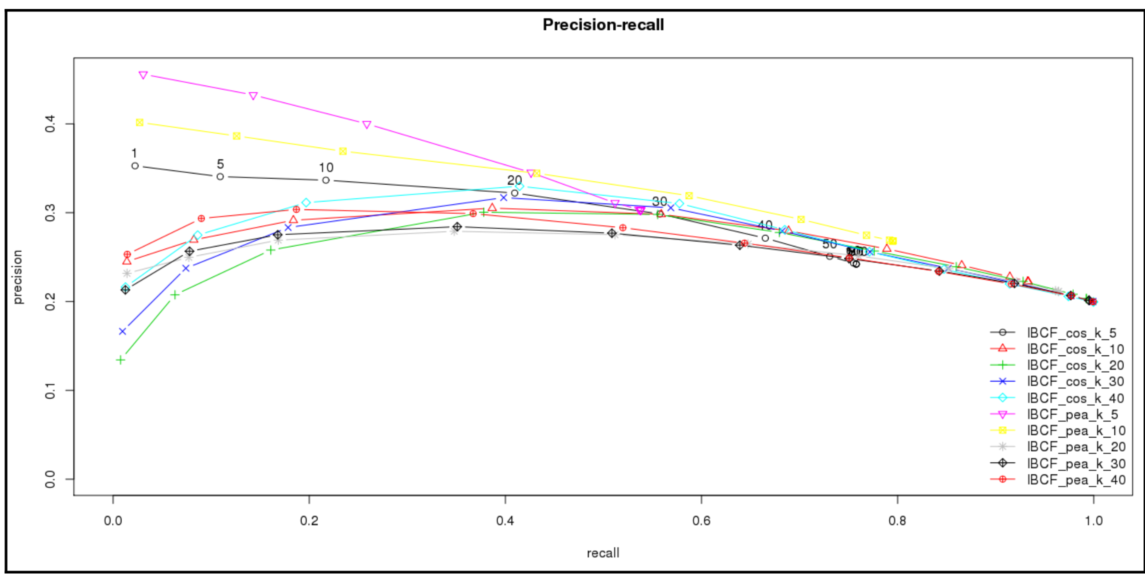
```
plot(list_results, annotate = c(1,2), legend = "topleft")
title("ROC curve")
```



From the preceding plot, the best methods are IBCF with cosine similarity with *n = 30*, and the next best is the Pearson method with *n = 40*.

Let's confirm this with the `Precision-Recall` curve as follows:

```
plot(list_results, "prec/rec", annotate = 1, legend = "bottomright")
title("Precision-recall")
```



From the above plot we see that the best Precision-Recall ratio is achieved when the number of recommendations = 30 with cosine similarity and *n=40*. Another good model is achieved with the Pearson similarity method and *n=10*.

Collaborative filtering using Python

In the previous section we saw implementations of user-based recommender systems and item-based recommender systems using the R package, `recommenderlab`. In this section, we see UBCF and IBCF implementation using the Python programming language.

For this section, we use the MovieLens 100k dataset, which contains 943 user ratings on 1682 movies. Unlike in R, in Python we do not have a proper Python package dedicated to building recommender engines, at least the neighborhood-based recommenders such as user-based/item-based recommenders.

We have the Crab Python package available but it is not actively supported. So I thought of building a recommender engine using scientific packages in Python such as NumPy, sklearn, and Pandas.

Installing the required packages

For this section, please make sure you have the following system requirements:

- Python 3.5
- Pandas 1.9.2 – Pandas is an open source, BSD-licensed library providing high-performance, easy-to-use data structures, and data analysis tools for the Python programming language.
- NumPy 1.9.2 – NumPy is the fundamental package for scientific computing with Python.
- sklearn 0.16.1

The best way to install the preceding packages is to install, Anaconda distribution which will install all the required packages such as Python, Pandas, and Numpy. Anaconda can be found at: `https://www.continuum.io/downloads`

Data source

The MovieLens 100k data can be downloaded from the following link:

`http://files.grouplens.org/datasets/movielens/ml-100k.zip`

Let's get started with implementing user-based collaborative filtering. Assuming we have downloaded the data into our local system, let's load the data into a Python environment.

We load the data using the Pandas package and the `read_csv()` method by passing two parameters, path and separator as follows:

```
path = "~/udata.csv"
df = pd.read_csv(path, sep='\t')
```

The data will be loaded as a DataFrame, a table-like data structure that can easily be used for data handling and manipulation tasks.

```
type(df)
<class 'pandas.core.frame.DataFrame'>
```

Let's see the first six results of the data frame to have a look at how data seems to be using the `head()` method available in the Pandas DataFrame object:

```
df.head()
 UserID  ItemId   Rating  Timestamp
0     196      242       3  881250949
1     186      302       3  891717742
2      22      377       1  878887116
3     244       51       2  880606923
4     166      346       1  886397596
```

Let's see the column names of the data frame, `df` using the columns attributes. The result of the following code snippet shows that there are four columns: `UserID`, `ItemId`, `Rating`, `Timestamp`, and that it is of the object datatype:

```
df.columns
Index([u'UserID', u'ItemId ', u'Rating', u'Timestamp'], dtype='object')
```

Let's see the size of the data frame by calling the shape attribute; we observe that we have 100k records with 4 columns:

```
df.shape
(100000, 4)
```

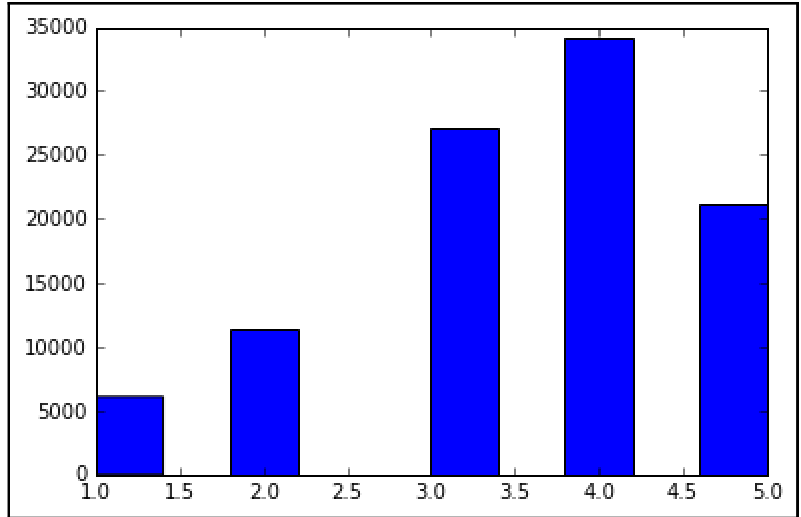
Data exploration

In this section, we will explore the MovieLens dataset and also prepare the data required for building collaborative filtering recommendation engines using python.

Let's see the distribution of ratings using the following code snippet:

```
import matplotlib.pyplot as plt
plt.hist(df['Rating'])
```

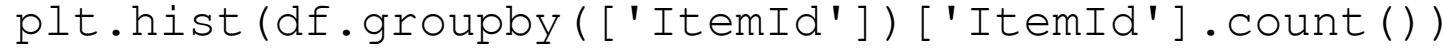
From the following image we see that we have more movies with 4 star ratings:



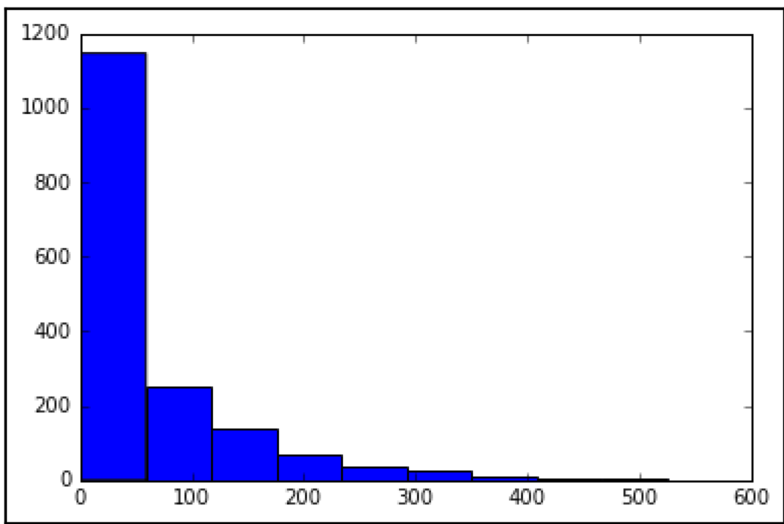
Using the following code snippet, we shall see the counts of ratings by applying the `groupby()` function and the `count()` function on DataFrame:

```
In [21]: df.groupby(['Rating'])['UserID'].count()
Out[21]:
Rating
1      6110
2     11370
3     27145
4     34174
5     21201
Name: UserID, dtype: int64
```

The following code snippet shows the distribution of movie views. In the following code we apply the `count()` function on DataFrame:



```
plt.hist(df.groupby(['ItemId'])['ItemId'].count())
```



From the previous image, we can observe that the starting ItemId has more ratings than later movies.

Rating matrix representation

Now that we have explored the data, let's represent the data in a rating matrix form so that we can get started with our original task of building a recommender engine.

For creating a rating matrix, we make use of NumPy package capabilities such as arrays and row iterations in a matrix. Run the following code to represent the data frame in a rating matrix:

In the following code, first we extract all the unique user IDs and then we check the length using the shape parameter.

Create a variable of `n_users` to find the total number of unique users in the data:

```
n_users = df.UserID.unique().shape[0]
```

Create a variable `n_items` to find the total number of unique movies in the data:

```
n_items = df['ItemId '].unique().shape[0]
```

Print the counts of unique users and movies:

```
print(str(n_users) + ' users')
943 users

print(str(n_items) + ' movies')
1682 movies
```

Create a zero value matrix of size (*n_users x n_items*) to store the ratings in the cell of the matrix ratings:

```
ratings = np.zeros((n_users, n_items))
```

For each tuple in the DataFrame, `df` extracts the information from each column of the row and stores it in the rating matrix cell value as follows:

```
for  row in df.itertuples():
ratings[row[1]-1, row[2]-1] = row[3]
```

Run the loop and the whole DataFrame movie ratings information will be stored in the matrix ratings of the `numpy.ndarray` type as follows:

```
type(ratings)
<type 'numpy.ndarray'>
```

Now let's see the dimensions of the multidimensional array 'ratings' using the shape attribute as follows:

```
ratings.shape
(943, 1682)
```

Let's see the sample data for how a ratings multidimensional array looks by running the following code:

```
ratings
array([[ 5.,  3.,  4., ...,  0.,  0.,  0.],
       [ 4.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       ...,
       [ 5.,  0.,  0., ...,  0.,  0.,  0.],
```

```
        [ 0.,  0.,  0., ...,  0.,  0.,  0.],
        [ 0.,  5.,  0., ...,  0.,  0.,  0.]])
```

We observe that the rating matrix is sparse as we see a lot of zeros in the data. Let's determine the `sparsity` in the data, by running the following code:

```
sparsity = float(len(ratings.nonzero()[0]))
sparsity /= (ratings.shape[0] * ratings.shape[1])
sparsity *= 100
print('Sparsity: {:4.2f}%'.format(sparsity))
Sparsity: 6.30%
```

We observe that the sparsity is `6.3%` that is to say that we only have rating information for `6.3%` of the data and for the others it is just zeros. Also please note that, the `0` value we see in the rating matrix doesn't represent the rating given by the user, it just means that they are empty.

Creating training and test sets

Now that we have a ratings matrix, let's create a training set and test set to build the recommender model using a training set and evaluate the model using a test set.

To divide the data into training and test sets, we use the `sklearn` package's capabilities. Run the following code to create training and test sets:

Load the `train_test_split` module into the python environment using the following import functionality:

```
from sklearn.cross_validation import train_test_split
```

Call the `train_test_split()` method with a test size of `0.33` and random seed of `42`:

```
 ratings_train, ratings_test = train_test_split(ratings,test_size=0.33,
random_state=42)
```

Let's see the dimensions of the train set:

```
 ratings_train.shape (631, 1682)
#let's see the dimensions of the test set
ratings_test.shape (312, 1682)
```

For user-based collaborative filtering, we predict that a user's rating for an item is given by the weighted sum of all other users' ratings for that item, where the weighting is the cosine similarity between each user and the input user.

The steps for building a UBCF

The steps for building a UBCF are:

- Creating a similarity matrix between the users
- Predicting the unknown rating value of item *i* for an active user *u* by calculating the weighted sum of all the users' ratings for the item.

Here the weighting is the cosine similarity calculated in the previous step between the user and neighboring users.

- Recommending the new items to the users.

User-based similarity calculation

The next step is to create pairwise similarity calculations for each user in the rating matrix, that is to say we have to calculate the similarity of each user with all the other users in the matrix. The similarity calculation we choose here is cosine similarity. For this, we make use of pairwise distance capabilities to calculate the cosine similarity available in the `sklearn` package as follows:

```
import numpy as np
import sklearn

#call cosine_distance() method available in sklearn metrics module
by passing the ratings_train set.
The output will be a distance matrix.
dist_out = 1-
sklearn.metrics.pairwise.cosine_distances(ratings_train)

# the type of the distance matrix will be the same type of the
rating matrix
type(dist_out)
<type 'numpy.ndarray'>
#the dimensions of the distance matrix will be a square matrix of
size equal to the number of users.
dist_out.shape
(631, 631)
```

Let's see a sample dataset of the distance matrix:

```
dist_out
```

```
array ([[ 1.        ,| 0.36475764,  0.44246231, ...,  0.02010641,
          0.33107929, 0.25638518],
        [ 0.36475764, 1.        ,  0.42635255, ..., 0.06694419,
          0.27339314, 0.22337268],
        [ 0.44246231, 0.42635255, 1.        , ..., 0.06675756,
          0.25424373, 0.22320126],
        ...,
        [ 0.02010641, 0.06694419, 0.06675756, ..., 1.        ,
          0.04853428, 0.05142508],
        [ 0.33107929, 0.27339314, 0.25424373, ..., 0.04853428,
          1.        ,  0.1198022],
        [ 0.25638518, 0.22337268, 0.22320126, ..., 0.05142508,
          0.1198022, 1.    ]])
```

Predicting the unknown ratings for an active user

As previously mentioned, the unknown values can be calculated for all the users by taking the dot product between the distance matrix and the rating matrix and then normalizing the data with the number of ratings as follows:

```
user_pred = dist_out.dot(ratings_train) /
np.array([np.abs(dist_out).sum(axis=1)]).T
```

Now that we have predicted the unknown ratings for use in the training set, let's define a function to check the error or performance of the model. The following code defines a function for calculating the root mean square error (RMSE) by taking the predicted values and original values. We use `sklearn` capabilities for calculating RMSE as follows:

```
from sklearn.metrics import mean_squared_error
def get_mse(pred, actual):
    #Ignore nonzero terms.
    pred = pred[actual.nonzero()].flatten()
    actual = actual[actual.nonzero()].flatten()
    return mean_squared_error(pred, actual)
```

We call the `get_mse()` method to check the model prediction error rate as follows:

```
get_mse(user_pred, ratings_train)
7.8821939915510031
```

We see that the model accuracy or RMSE is `7.8`. Now let's run the same `get_mse()` method on the test data and check the accuracy as follows:

```
get_mse(user_pred, ratings_test)
8.9224954316965484
```

User-based collaborative filtering with the k-nearest neighbors

If we observe the RMSE values in the above model, we can see that the error is a bit higher. The reason may be that we have chosen all the users' rating information while making the predictions. Instead of considering all the users, let's consider only the top-N similar users' ratings information and then make the predictions. This may result in improving the model accuracy by eliminating some biases in the data.

To explain in a more elaborate way; in the previous code we predicted the ratings of the users by taking the weighted sum of the ratings of all users, instead we first chose the top-N similar users for each user and then the ratings were calculated by considering the weighted sum of the ratings of these top-N users.

Finding the top-N nearest neighbors

Firstly, for computational easiness, we shall choose the top five similar users by setting a variable, *k*.

k=5

We use the k-nearest neighbors method to choose the top five nearest neighbors for an active user. We will see this in action shortly. We choose sklearn.knn capabilities for this task as follows:

```
from sklearn.neighbors import NearestNeighbors
```

Define the `NearestNeighbors` object by passing k and the similarity method as parameters:

```
neigh = NearestNeighbors(k,'cosine')
```

Fit the training data to the `nearestNeighbor` object:

```
neigh.fit(ratings_train)
```

Calculate the top five similar users for each user and their similarity values, that is the distance values between each pair of users:

```
top_k_distances,top_k_users = neigh.kneighbors(ratings_train,
return_distance=True)
```

We can observe below that the resultant `top_k_distances` ndarray contains similarity values and top five similar users for each users in the training set:

```
top_k_distances.shape
(631, 5)
top_k_users.shape
(631, 5)
```

Let's see the top five users that are similar to user 1 in the training set:

```
top_k_users[0]
array([  0,  82, 511, 184, 207], dtype=int64)
```

The next step would be to choose only the top five users for each user and use their rating information while predicting the ratings using the weighted sum of all of the ratings of these top five similar users.

Run the following code to predict the unknown ratings in the training data:

```
user_pred_k = np.zeros(ratings_train.shape)
for i in range(ratings_train.shape[0]):
    user_pred_k[i,:] =
top_k_distances[i].T.dot(ratings_train[top_k_users][i])
/np.array([np.abs(top_k_distances[i].T).sum(axis=0)]).T
```

Let's see the data predicted by the model as follows:

```
user_pred_k.shape
(631, 1682)

user_pred_k
```

The following image displays the results for `user_pred_k`:

```
array ([[ 3.25379713, 1.75556855, 0.        , ..., 0.        ,
          0.        ,  0.        ],
        [ 1.48370298, 0.        ,  1.24948776, ..., 0.        ,
          0.        ,  0.        ],
        [ 1.01011767, 0.73826825, 0.7451635, ..., 0.        ,
          0.        ,  0.        ],      ...,
        [ 0.        , 0.        ,  0.        , ..., 0.        ,
          0.        ,  0.        ],
        [ 0.74469557, 0.        ,  0.        , ..., 0.        ,
          0.        ,  0.        ],
        [ 1.9753676, 0.         ,  0.        , ...,  0.        ,
          0.        ,  0.        ]])
```

Now let's see if the model has improved or not. Run the get_mse() method defined earlier as follows:

```
get_mse(user_pred_k, ratings_train)
8.9698490022546036
get_mse(user_pred_k, ratings_test)
11.528758029255446
```

Item-based recommendations

IBCF is very similar to UBCF but with very minor changes in how we use the rating matrix.

The first step is to calculate the similarities between movies, as follows:

Since we have to calculate the similarity between movies, we use movie count as `k` instead of user count:

```
k = ratings_train.shape[1]
neigh = NearestNeighbors(k,'cosine')
```

We fit the transpose of the rating matrix to the `NearestNeighbors` object:

```
neigh.fit(ratings_train.T)
```

Calculate the cosine similarity distance between each movie pair:

```
top_k_distances,top_k_users = neigh.kneighbors(ratings_train.T,
return_distance=True)
top_k_distances.shape
(1682, 1682)
```

The next step is to predict the movie ratings using the following code:

```
item__pred = ratings_train.dot(top_k_distances) /
np.array([np.abs(top_k_distances).sum(axis=1)])
item__pred.shape
(631, 1682)
item__pred
```

The following image shows the result for `item_pred`:

```
array ([[ 0.        , 0.51752631,  0.60019695, ...,  2.31664301,
         2.34134745, 2.46671096],
       [ 0.        , 0.31976603, 0.37168534, ..., 1.34680571,
         1.34897863, 1.44314592],
       [ 0.        , 0.50619664, 0.58685005, ..., 2.5337623,
         2.57055505, 2.74749235],
       ...,
       [ 0.        , 0.08945322, 0.10271303, ..., 0.41949597,
         0.41995047, 0.45733339],
       [ 0.        , 0.25785693, 0.29819614, ..., 1.30767892,
         1.32470838, 1.41198324],
       [ 0.        , 0.07197376, 0.08524505, ..., 0.25523416,
         0.25259761, 0.26155752]])
```

Evaluating the model

Now let's evaluate the model using the `get_mse()` method we have defined by passing the prediction ratings and the training and test set as follows:

```
get_mse(item_pred, ratings_train)
11.130000188318895
get_mse(item_pred,ratings_test)
12.128683035513326
```

The training model for k-nearest neighbors

Run the following code to calculate the distance matrix for the top 40 nearest neighbors and then calculate the weighted sum of ratings by the top 40 users for all the movies. If we closely observe the code, it is very similar to what we have done for UBCF. Instead of passing `ratings_train` as is, we transpose the data matrix and pass to the previous code as follows:

```
k = 40
neigh2 = NearestNeighbors(k,'cosine')
neigh2.fit(ratings_train.T)
top_k_distances,top_k_movies = neigh2.kneighbors(ratings_train.T,
return_distance=True)

#rating prediction - top k user based
pred = np.zeros(ratings_train.T.shape)
for i in range(ratings_train.T.shape[0]):
    pred[i,:] =
top_k_distances[i].dot(ratings_train.T[top_k_users][i])/np.array([np.abs(to
p_k_distances[i]).sum(axis=0)]).T
```

Evaluating the model

The follow code snippet calculates the mean squared error for the training and test set. We can observe that the training error is 11.12 whereas the test error is 12.12.

```
get_mse(item_pred_k, ratings_train)
11.130000188318895
get_mse(item_pred_k,ratings_test)
12.128683035513326
```

Summary

In this chapter, we have explored building collaborative filtering approaches such as user-based and item-based approaches in R and Python, the popular data mining programming languages. The recommendation engines are built on MovieLens, and Jester5K datasets available online.

We have learnt about how to build the model, choose data, explore the data, create training and test sets, and evaluate the models using metrics such as RMSE, Precision-Recall, and ROC curves. Also, we have seen how to tune parameters for model improvements.

In the next chapter, we will be covering personalized recommendation engines such as content-based recommendation engines and context-aware recommendation engines using R and Python.

6
Building Personalized Recommendation Engines

Recommendation engines have been evolving very fast, with a lot of research also going into this field. Big multinational companies are investing huge amounts of money into this field. As mentioned earlier, right from the earlier models of recommendation engines such as collaborative filtering, these systems have been a huge success. With more and more revenues being generated through recommendation engines and more and more people using the Internet for their shopping needs, reading news, or for getting information related to health, business organizations have seen huge business in tapping this available user activities on the Internet. With the increase in the number of users of recommendation engines, and with more and more applications being powered by recommendation engines, users also started asking for personalized suggestions rather than community-based recommendations. This requirement of the user community was taken as the new challenge, and personalized recommendation engines have been built for providing suggestions at a personal level.

Almost all the industry domains are currently building recommendation engines that can recommend at personalized levels.

The following are a few personalized recommendations:

- Personalized news recommendations–Google News
- Personalized health-care systems
- Personalized travel recommendation systems
- Personalized recommendations on Amazon
- Personalized movie recommendations on YouTube

The following is the screenshot of personalized recommendations:

In the `Chapter 3`, *Recommendation Engines Explained*, we learned in detail about content-based recommender systems and context-aware recommender systems. In this chapter, we will recall these topics in brief and then move ahead to build content-based and context-aware recommender systems.

Personalized recommender systems

In this chapter, we will learn about two flavours of personalized recommenders:

- Content-based recommender systems
- Context-aware recommender systems

Content-based recommender systems

Building collaborative filtering is relatively easy. In the fifth chapter, we learned about building collaborative filtering recommender systems. While building those systems, we just considered the ratings given to a product and the information about whether a product is liked or not. With this minimal information, we built the systems. To many people's surprise, these systems performed very well. But these systems had their own limitations, such as the cold start problem explained in the previous chapters.

Assume a case of a user, Nick, giving five-star rating to a movie, say *Titanic.* What could have made Nick give that rating? May be the story of the film, the actors in the movie, the background score, or the screenplay. These preferences for these features made Nick rate the movie. Wouldn't including this internal information of preferences for the product/features make more sense while building recommendations?

In collaborative filtering, the basic assumption is that people with similar taste in the past will have similar taste in the future. If we closely observe, this assumption may not apply in all cases. For example, if my neighbors have rated the thriller movie *The Exorcist highly*, that movie should not be suggested to me since I have a preference for romantic movies. I should instead get *Titanic,* which is of the romance genre, as a suggestion. I do not always have the same taste as my neighbors; I would be happy if I got suggestions solely based on my preferences and actions. Businesses have seen a lot of business opportunities in implementing these types of recommendations, known as personalized recommender systems, at an individual level.

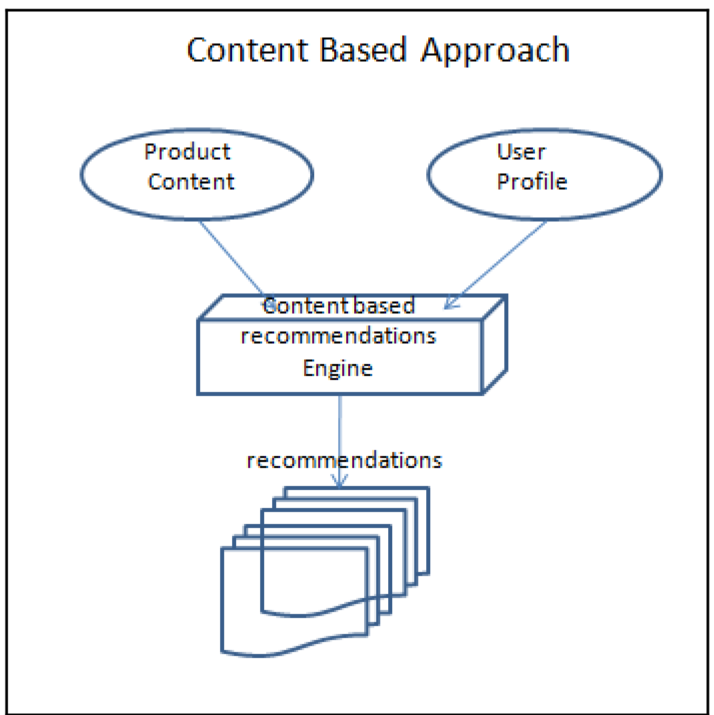
Building a content-based recommendation system

In content-based recommender systems, we use the content information of both users and items while building recommendation engines. A typical content-based recommender system will perform the following steps:

1. Generate user profiles.
2. Generate item profile.
3. Generate the recommendation engine model.
4. Suggest the top N recommendations.

We first generate user and item profiles from the available information. A profile typically contains preferences for the features of items and users (refer to `Chapter 3`, *Recommendation Engines Explained* for details). Once the profiles are created, we choose a method to build the recommendation engine model. Many data-mining techniques such as classification, text similarity approaches such as *tf-idf* similarity, and Matrix factorization models can be applied for building content-based recommendation engines.

We can even employ multiple recommendation engine models and build hybrid recommendation engines to serve as content-based recommendations. A typical content recommender is depicted in the following figure:

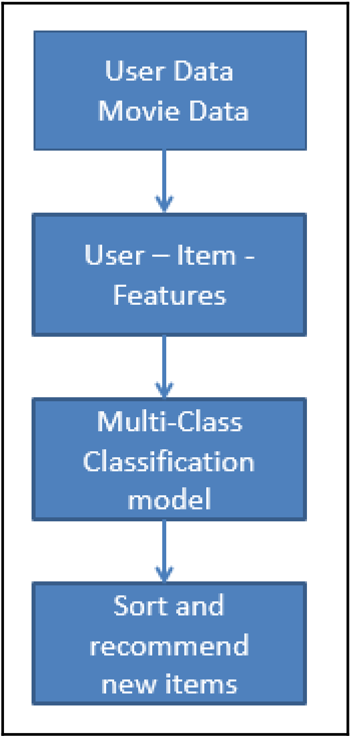


Content-based recommendation using R

Let's start building a personalized recommendation engine in R. We choose the MovieLens dataset to build our system. In the previous section, we refreshed the concepts of content-based recommender systems. There are multiple ways we can build personalized recommenders; in this section, we will use the multiclass classification approach to build our basic content-based recommendation engine.

Using the classification approach, we are trying to build a model-based recommendation engine. Most recommender systems–either collaborative filtering or content-based–use neighbourhood methods to build the recommenders. Let's explore how we can use a supervised machine-learning approach to build the recommendation engines.

Before we start writing the code, let's discuss the steps for building the personalized recommender system. The following figure shows the order of steps we would be following to achieve our objective:



The first step would always be to gather the data and pull it into the programming environment so that we may apply further steps. For our use case, we download the MovieLens dataset containing three sets of data, as defined next:

- Ratings data containing userID, itemID, rating, timestamp
- User data containing the user information, such as userID, age, gender, occupation, ZIP code, and so on
- Movie data containing a certain movie's information, such as movieID, release date, URL, genre details, and so on

The second step would be preparing the data required to build the classification models. In this step, we extract the required features of the users and class labels to build the classification model:

MovieId	UserId	Rating	Gender	Occupation	Unknown	Action	Adventure	Animation	Children	Comedy	Crime
1	1	5	M	technician	0	0	0	1	1	1	0
1	117	4	M	student	0	0	0	1	1	1	0
1	429	3	M	student	0	0	0	1	1	1	0
1	919	4	M	other	0	0	0	1	1	1	0
1	457	4	F	salesman	0	0	0	1	1	1	0
1	468	5	M	engineer	0	0	0	1	1	1	0
1	17	4	M	programmer	0	0	0	1	1	1	0
1	892	5	M	other	0	0	0	1	1	1	0
1	16	5	M	entertainment	0	0	0	1	1	1	0
1	580	3	M	student	0	0	0	1	1	1	0
1	268	3	M	engineer	0	0	0	1	1	1	0
1	894	4	M	educator	0	0	0	1	1	1	0
1	535	3	F	educator	0	0	0	1	1	1	0

- For our example case, we define the ratings (1 to 5) given by the users as class labels, such as 1-3 rating as 0 and 4-5 rating as 1. Thus, we will build a two-class classification model. Our model will predict the class label, given the input features for a given user.

You might be wondering why we are choosing binary classification instead of multiclass classification. The choice of model is left to the person building the recommender system; in our case, with the dataset we have chosen, binary class classification fits better than a multiclass classification. Readers are encouraged to try multiclass-classification for your understanding.

- We choose user demography and item features from user data and item data to form the features of our binary classification model. We extended the `User_item_rating` data by including features such as genre information for the movie rated by the user, user personal information such as age, gender, occupation, and so on. The final features and class labels can be seen in the preceding figure.

The third step will be to build the binary classification model. We will choose the RandomForest algorithm to build the class.

The fourth and final step will be to generate the top-N recommendations for the users. For our example, we take a test user and predict the class labels for the movie that he has not rated earlier and send the top-N movies, which have higher probability ratings predicted by our classification model.

Please note that the choice of generating the top-N recommendations are left to the choice of the users.

Let's implement the aforementioned steps using R. In this section, we will go through a step-by-step implementation of content-based recommendation using R.

Dataset description

For this exercise, we use two MovieLens dataset files–one is a ratings file containing ratings given to 943 to 1682 movies on a scale of 1-5, and the second is an item dataset file containing content information, that is, information about the movie genre, movie name, movie ID, URLs, and so on.

The MovieLens dataset can be downloaded from following URL:
`http://grouplens.org/datasets/movielens/`

Loading ratings data into R environment using `read.csv()` function available in R:

```
raw_data = read.csv("~/udata.csv",sep="\t",header=F)
Adding column names to the dataframe
colnames(raw_data) = c("UserId","MovieId","Rating","TimeStamp")
```

This code removes the last column from the DataFrame:

```
ratings = raw_data[,1:3]
```

See the first five lines of the data, we use `head()` function as follows:

```
head(ratings)
```

See the columns of the rating data frame using `names()` function.

See the descriptions of the ratings function using `str()` function. All the results of the three mentioned functions are shown as follows:

```
> head(ratings)
  UserId MovieId Rating
1    196     242      3
2    186     302      3
3     22     377      1
4    244      51      2
5    166     346      1
6    298     474      4
> names(ratings)
[1] "UserId"  "MovieId" "Rating"
> str(ratings)
'data.frame':   100000 obs. of  3 variables:
 $ UserId : int  196 186 22 244 166 298 115 253 305 6 ...
 $ MovieId: int  242 302 377 51 346 474 265 465 451 86 ...
 $ Rating : int  3 3 1 2 1 4 2 5 3 3 ...
>
```

The following code loads item data into the R environment using the `read.csv()` function available in R:

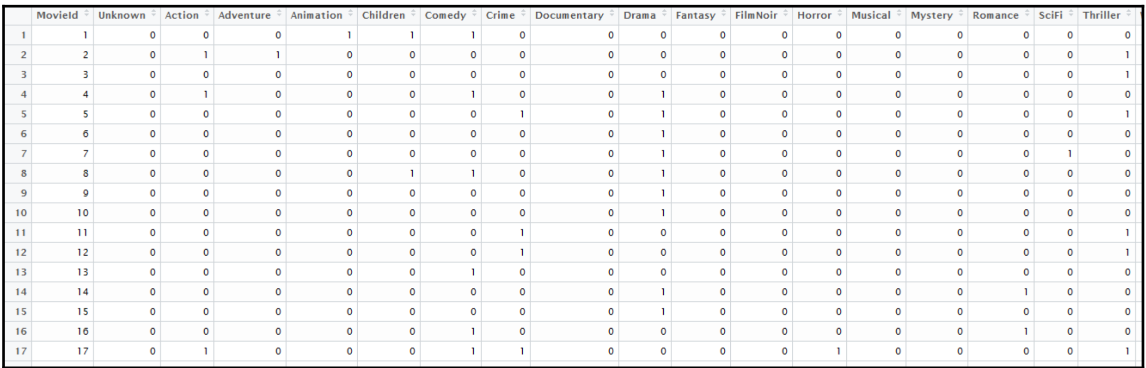
```
movies =
read.csv("C:/Suresh/R&D/packtPublications/reco_engines/drafts/personalRecos
/uitem.csv",sep="|",header=F)
```

Next, we add columns to the data frame:

```
colnames(movies) =
c("MovieId","MovieTitle","ReleaseDate","VideoReleaseDate","IMDbURL","Unknow
n","Action","Adventure","Animation","Children","Comedy","Crime","Documentar
y","Drama","Fantasy","FilmNoir","Horror","Musical","Mystery","Romance","Sci
Fi","Thriller","War","Western")
```

Then we remove unwanted data; for this exercise we are keeping only the genre information only:

```
movies = movies[,-c(2:5)]
View(movies)
```



	MovieId	Unknown	Action	Adventure	Animation	Children	Comedy	Crime	Documentary	Drama	Fantasy	FilmNoir	Horror	Musical	Mystery	Romance	SciFi	Thriller
1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0
2	2	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
3	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
4	4	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0
5	5	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1
6	6	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
7	7	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0
8	8	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0
9	9	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
10	10	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
11	11	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
12	12	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
13	13	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
14	14	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0
15	15	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
16	16	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0
17	17	0	1	0	0	0	1	1	0	0	0	0	1	0	0	0	0	1

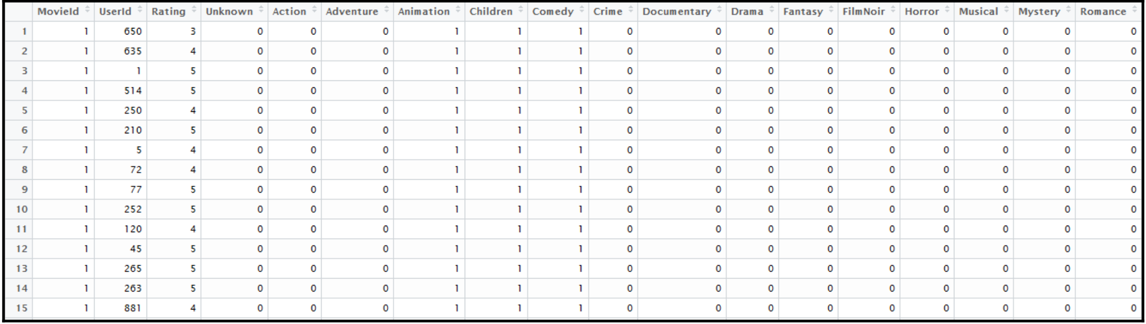
The description of movies is given by `str(movies)`.The column names can be seen using `names(movies)`:

```
> names(movies)
 [1] "MovieId"     "Unknown"     "Action"      "Adventure"   "Animation"   "Children"
 [7] "Comedy"      "Crime"       "Documentary" "Drama"       "Fantasy"     "FilmNoir"
[13] "Horror"      "Musical"     "Mystery"     "Romance"     "SciFi"       "Thriller"
[19] "War"         "Western"
> str(movies)
'data.frame':   1682 obs. of  20 variables:
 $ MovieId    : int  1 2 3 4 5 6 7 8 9 10 ...
 $ Unknown    : int  0 0 0 0 0 0 0 0 0 0 ...
 $ Action     : int  0 1 0 1 0 0 0 0 0 0 ...
 $ Adventure  : int  0 1 0 0 0 0 0 0 0 0 ...
 $ Animation  : int  1 0 0 0 0 0 0 0 0 0 ...
 $ Children   : int  1 0 0 0 0 0 0 1 0 0 ...
 $ Comedy     : int  1 0 0 1 0 0 0 1 0 0 ...
 $ Crime      : int  0 0 0 0 1 0 0 0 0 0 ...
 $ Documentary: int  0 0 0 0 0 0 0 0 0 0 ...
 $ Drama      : int  0 0 0 1 1 1 1 1 1 1 ...
 $ Fantasy    : int  0 0 0 0 0 0 0 0 0 0 ...
 $ FilmNoir   : int  0 0 0 0 0 0 0 0 0 0 ...
 $ Horror     : int  0 0 0 0 0 0 0 0 0 0 ...
 $ Musical    : int  0 0 0 0 0 0 0 0 0 0 ...
 $ Mystery    : int  0 0 0 0 0 0 0 0 0 0 ...
 $ Romance    : int  0 0 0 0 0 0 0 0 0 0 ...
 $ SciFi      : int  0 0 0 0 0 0 1 0 0 0 ...
 $ Thriller   : int  0 1 1 0 1 0 0 0 0 0 ...
 $ War        : int  0 0 0 0 0 0 0 0 0 1 ...
 $ Western    : int  0 0 0 0 0 0 0 0 0 0 ...
>
```

The next step is to create feature profiles of customers to build a classification model. We should extend the rating data frame containing userID, movieID, and rating with the movie properties, as shown next.

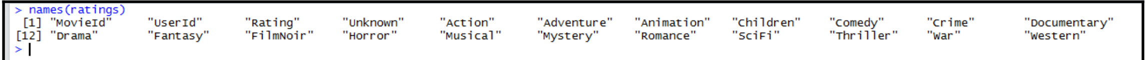
In the following code, we use `merge()` to perform a join function to merge ratings data with item data:

```
ratings = merge(x = ratings, y = movies, by = "MovieId", all.x = TRUE)

View(ratings)
```



	MovieId	UserId	Rating	Unknown	Action	Adventure	Animation	Children	Comedy	Crime	Documentary	Drama	Fantasy	FilmNoir	Horror	Musical	Mystery	Romance
1	1	650	3	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0
2	1	635	4	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0
3	1	1	5	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0
4	1	514	5	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0
5	1	250	4	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0
6	1	210	5	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0
7	1	5	4	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0
8	1	72	4	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0
9	1	77	5	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0
10	1	252	5	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0
11	1	120	4	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0
12	1	45	5	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0
13	1	265	5	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0
14	1	263	5	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0
15	1	881	4	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0

Let's see the columns names using `names()` method:



```
> names(ratings)
 [1] "MovieId"     "UserId"      "Rating"      "Unknown"     "Action"      "Adventure"   "Animation"   "Children"    "Comedy"      "Crime"       "Documentary"
[12] "Drama"       "Fantasy"     "FilmNoir"    "Horror"      "Musical"     "Mystery"     "Romance"     "SciFi"       "Thriller"    "War"         "Western"
> |
```

Now we create the class labels for each record of the profile we just created. We shall create a binary class label for each of the ratings so that 1-3 ratings will be labelled as 0 and 4-5 ratings as 1. The following code does this conversion for us. We use the `lapply()` function to reshape the ratings:

The following code manages the conversion of numerical ratings to binary categorical variable:

```
nrat = unlist(lapply(ratings$Rating,function(x)
{
  if(x>3) {return(1)}
  else {return(0)}
}))
```

Next, we combine the newly created rating categorical rating variable – `nrat` – with the original rating data frame ratings using `cbind()`:

```
ratings = cbind(ratings,nrat)
```

```
> head(ratings)
  MovieId UserId Rating Unknown Action Adventure Animation Children Comedy Crime Documentary Drama Fantasy FilmNoir Horror Musical Mystery Romance SciFi Thriller
1       1    650      3       0      0         0         1        1      1     0           0     0       0        0      0       0       0       0     0        0
2       1    635      4       0      0         0         1        1      1     0           0     0       0        0      0       0       0       0     0        0
3       1      1      5       0      0         0         1        1      1     0           0     0       0        0      0       0       0       0     0        0
4       1    514      5       0      0         0         1        1      1     0           0     0       0        0      0       0       0       0     0        0
5       1    250      4       0      0         0         1        1      1     0           0     0       0        0      0       0       0       0     0        0
6       1    210      5       0      0         0         1        1      1     0           0     0       0        0      0       0       0       0     0        0
  War Western nrat
1   0       0    0
2   0       0    1
3   0       0    1
4   0       0    1
5   0       0    1
6   0       0    1
> |
```

In the preceding figure, we can observe the new rating binary class, `nrat`.

Now let's observe the variables that will be going into the model building stage using the `apply()` function by applying `table()` to each column, as shown next:

```
apply(ratings[,-c(1:3,23)],2,function(x)table(x))
```

```
> apply(ratings[,-c(1:3,23)],2,function(x)table(x))
  Unknown Action Adventure Animation Children Comedy Crime Documentary Drama Fantasy FilmNoir Horror Musical Mystery Romance SciFi Thriller   War Western
0   99990  74411     86247     96395    92818  70168 91945       99242 60105   98648    98267  94683   95046   94755   80539 87270    78128 90602   98146
1      10  25589     13753      3605     7182  29832  8055         758 39895    1352     1733   5317    4954    5245   19461 12730    21872  9398    1854
> |
```

From the preceding results, we can observe that the number of zeroes is very high when compared to the number of 1s; so let's remove this variable from our feature list. Also, let's remove the rating variable, as we have created a new variable `nrat`:

```
scaled_ratings = ratings[,-c(3,4)]
```

We shall now standardize or center the data by using the `scale()` function available in R before we build the model as shown in the following code snippet. Standardizing will adjust data in different scales to common a scale. The scale function will apply centering by removing column means on the each of corresponding column:

```
scaled_ratings=scale(scaled_ratings[,-c(1,2,21)])
scaled_ratings = cbind(scaled_ratings,ratings[,c(1,2,23)])
```

```
> head(scaled_ratings)
      Action  Adventure Animation Children  Comedy      Crime Documentary      Drama   Fantasy   FilmNoir    Horror    Musical    Mystery    Romance      SciFi
1 -0.5864161 -0.3993232  5.170975 3.594937 1.53365 -0.2959828 -0.08739462 -0.8147076 -0.117069 -0.1327985 -0.236971 -0.2283016 -0.2352716 -0.4915609 -0.3819263
2 -0.5864161 -0.3993232  5.170975 3.594937 1.53365 -0.2959828 -0.08739462 -0.8147076 -0.117069 -0.1327985 -0.236971 -0.2283016 -0.2352716 -0.4915609 -0.3819263
3 -0.5864161 -0.3993232  5.170975 3.594937 1.53365 -0.2959828 -0.08739462 -0.8147076 -0.117069 -0.1327985 -0.236971 -0.2283016 -0.2352716 -0.4915609 -0.3819263
4 -0.5864161 -0.3993232  5.170975 3.594937 1.53365 -0.2959828 -0.08739462 -0.8147076 -0.117069 -0.1327985 -0.236971 -0.2283016 -0.2352716 -0.4915609 -0.3819263
5 -0.5864161 -0.3993232  5.170975 3.594937 1.53365 -0.2959828 -0.08739462 -0.8147076 -0.117069 -0.1327985 -0.236971 -0.2283016 -0.2352716 -0.4915609 -0.3819263
6 -0.5864161 -0.3993232  5.170975 3.594937 1.53365 -0.2959828 -0.08739462 -0.8147076 -0.117069 -0.1327985 -0.236971 -0.2283016 -0.2352716 -0.4915609 -0.3819263
    Thriller        War   Western MovieId UserId nrat
1 -0.5291012 -0.3220673 -0.137441       1    650    0
2 -0.5291012 -0.3220673 -0.137441       1    635    1
3 -0.5291012 -0.3220673 -0.137441       1      1    1
4 -0.5291012 -0.3220673 -0.137441       1    514    1
5 -0.5291012 -0.3220673 -0.137441       1    250    1
6 -0.5291012 -0.3220673 -0.137441       1    210    1
```

Now let's get into building the model using the `randomForest` algorithm for binary classification. Before that, let's divide the data into training and test sets with an 80:20 split.

The following code will first create a randomize index object of all the data. Then we use this indexes to divide the train and test sets.

```
set.seed(7)
which_train <- sample(x = c(TRUE, FALSE), size = nrow(scaled_ratings),
                      replace = TRUE, prob = c(0.8, 0.2))
model_data_train <- scaled_ratings[which_train, ]
model_data_test <- scaled_ratings[!which_train, ]
```

```
> dim(model_data_test)
[1] 19955    21
> dim(model_data_train)
[1] 80045    21
```

Now let's build the model using `randomForest` algorithm from the library `randomForest`:

In the following code snippet, we are converting the integer `nrat` variable to factor format.

```
library(randomForest)
fit = randomForest(as.factor(nrat)~., data = model_data_train[,-c(19,20)])
```

We can see the details of the model build, fit, by just typing, `fit`:

```
> fit

Call:
 randomForest(formula = as.factor(nrat) ~ ., data = model_data_train[,      -c(19, 20)])
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 4

        OOB estimate of  error rate: 39.98%
Confusion matrix:
      0     1 class.error
0 14390 21315   0.5969752
1 10686 33654   0.2410014
```

In the previous code snippet we have used the `randomforest()` method with default values. For random forests we have two parameters which can be tuned for optimal performance; **mtry** is number of samples at each tree split, **ntree** is number of decision trees to be grown. Using parameter tuning and cross-validation approaches, we should choose optimal parameters.

We can also see the summary of the model using `summary()`, as shown next:

```
> summary(fit)
                Length Class  Mode
call                 3 -none- call
type                 1 -none- character
predicted        80045 factor numeric
err.rate          1500 -none- numeric
confusion            6 -none- numeric
votes           160090 matrix numeric
oob.times        80045 -none- numeric
classes              2 -none- character
importance          18 -none- numeric
importanceSD         0 -none- NULL
localImportance      0 -none- NULL
proximity            0 -none- NULL
ntree                1 -none- numeric
mtry                 1 -none- numeric
forest              14 -none- list
y                80045 factor numeric
test                 0 -none- NULL
inbag                0 -none- NULL
terms                3 terms  call
```

Now, let's see how the model performs on the test set:

```
predictions <- predict(fit, model_data_test[,-c(19,20,21)], type="class")
```

```
> predictions[0:20]
  1   8  19  23  24  25  29  36  39  45  48  49  50  73  80  81  82  93  99 107
  1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
Levels: 0 1
```

Let's evaluate the model using the Precision-Recall method:

```
#building confusion matrix
cm = table(predictions,model_data_test$nrat)
(accuracy <- sum(diag(cm)) / sum(cm))
(precision <- diag(cm) / rowSums(cm))
recall <- (diag(cm) / colSums(cm))
```

```
> cm

predictions    0    1
          0 3541 2738
          1 5379 8297
> (accuracy <- sum(diag(cm)) / sum(cm))
[1] 0.5932348
> (precision <- diag(cm) / rowSums(cm))
        0         1
0.5639433 0.6066832
> recall <- (diag(cm) / colSums(cm))
> recall
        0         1
0.3969731 0.7518804
>
```

With the preceding results, we are quite happy with a 60% precision rate and a 75% recall rate. Now we move ahead to generate the top-N recommendations to a user ID (943) by performing the following steps:

1. Create a DataFrame containing all the movies not rated by the active user (user id: 943 in our case).

```
#extract distinct movieids
totalMovieIds = unique(movies$MovieId)
#see the sample movieids using tail() and head() functions:
```

```
> head(totalMovieIds)
[1] 1 2 3 4 5 6
> tail(totalMovieIds)
[1] 1677 1678 1679 1680 1681 1682
>
```

```
#a function to generate dataframe which creates non-rated
  movies by active user and set rating to 0;
nonratedmoviedf = function(userid){
  ratedmovies = raw_data[raw_data$UserId==userid,]$MovieId
  non_ratedmovies = totalMovieIds[!totalMovieIds %in%
    ratedmovies]
   df = data.frame(cbind(rep(userid),non_ratedmovies,0))
   names(df) = c("UserId","MovieId","Rating")
   return(df)
}

#let's extract non-rated movies for active userid 943
activeusernonratedmoviedf = nonratedmoviedf(943)
```

```
> head(activeusernonratedmoviedf)
  UserId MovieId Rating
1    943       1      0
2    943       3      0
3    943       4      0
4    943       5      0
5    943       6      0
6    943       7      0
>
```

2. Build a profile for this active user DataFrame:

```
activeuserratings = merge(x = activeusernonratedmoviedf, y =
  movies, by = "MovieId", all.x = TRUE)
```

```
> head(activeuserratings)
  MovieId UserId Rating Unknown Action Adventure Animation Children Comedy Crime Documentary Drama Fantasy FilmNoir Horror Musical Mystery Romance SciFi Thriller
1       1    943      0       0      0         0         1        1      1     0           0     0       0        0      0       0       0       0     0        0
2       3    943      0       0      0         0         0        0      0     0           0     0       0        0      0       0       0       0     0        1
3       4    943      0       0      1         0         0        0      1     0           0     1       0        0      0       0       0       0     0        0
4       5    943      0       0      0         0         0        0      0     1           0     1       0        0      0       0       0       0     0        1
5       6    943      0       0      0         0         0        0      0     0           0     1       0        0      0       0       0       0     0        0
6       7    943      0       0      0         0         0        0      0     0           0     1       0        0      0       0       0       0     1        0
  War Western
1   0       0
2   0       0
3   0       0
4   0       0
5   0       0
6   0       0
>
```

3. Predict ratings, sort and generate 10 recommendations:

```
#use predict() method to generate predictions for movie ratings
  by the active user profile created in the previous step.
predictions <- predict(fit, activeuserratings[,-c(1:4)],
  type="class")
#creating a dataframe from the results
recommend = data.frame(movieId =
  activeuserratings$MovieId,predictions)
#remove all the movies which the model has predicted as 0 and
  then we can use the remaining items as more probable movies
    which might be liked by the active user.
recommend = recommend[which(recommend$predictions == 1),]
```

With this step, we have when extending or improving finished building the content-based recommendation engine using a classification model. Before we move into the next section, I would like to make a clear point that the choice of the model and class label features is up to the reader to extend or improve the model.

As mentioned earlier, we should use cross-validation approach to choose optimal parameters so as to improve the model accuracy.

Content-based recommendation using Python

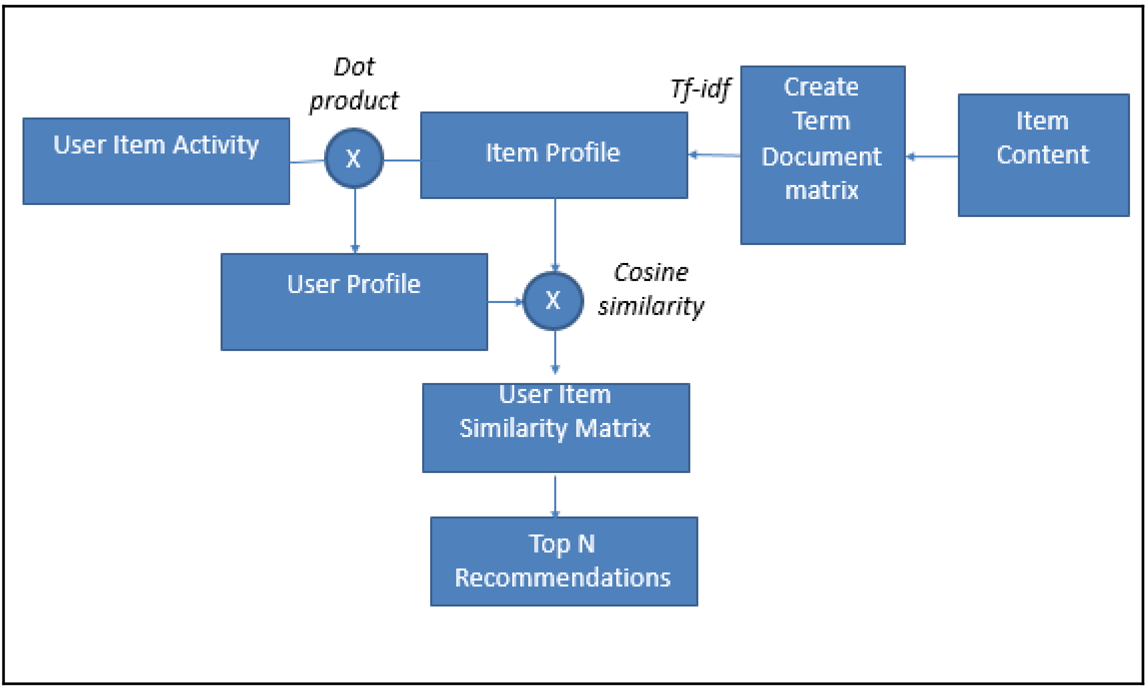
In the previous section, we built a model-based content recommendation engine using R. In this section, we will build content recommendations using another approach, using the Python `sklearn`, `NumPy`, and `pandas` packages.

Let's recall the steps for building a content-based system discussed in the beginning of the chapter:

1. Item profile generation
2. User profile generation
3. Recommendation engine model generation
4. Generation of the top-N recommendations

In this section, we shall learn in detail how to build content following the aforementioned steps using Python:

The design of the approach is shown in the following figure:



- **Item profile creation**: In this step, we create a profile for each item using the content information we have about the items. The item profile is usually created using a widely-used information retrieval technique called tf-idf. In `Chapter 4`, *Data Mining Techniques for Recommendation Engines*, we explained *tf-idf* in detail. To recap, the tf-idf value gives the relative importance of features with respect to all the items or documents.
- **User profile creation**: In this step, we take the user activity dataset and preprocess the data into a proper format to create a user profile. We should remember that, in a content-based recommender system, the user profile is created with respect to the item content, that is, we have to extract or compute the preferences of the user for the item content or item features. Usually, a dot product between user activity and item profile gives us the user profile.

- **Recommendation engine model generation**: Now that we have the user profile and item profile in hand, we will proceed to build a recommendation model. Computing a cosine similarity between the user profile and item profile gives us the affinity of the user to each of the items.
- **Generation of the top-N recommendations**: In the final step, we shall sort the user-item preferences based on the values calculated in the previous step and then suggest the top-N recommendations.

Now we will proceed toward the implementation of the aforementioned steps in Python.

Dataset description

In this section, we will use the Anonymous Microsoft Web Dataset to build a content-based recommendation system. The objective of this section is to recommend websites to an active user, based on his previous web browsing history.

MS Web Dataset refers to the web logs of the website `www.microsoft.com` accessed by 38,000 anonymous users. For each of the users, the dataset consists of lists about data of all the websites visited by the users in a time frame of one week.

The dataset can be downloaded from the following URL:

`https://archive.ics.uci.edu/ml/datasets/Anonymous+Microsoft+Web+Data`

For the sake of simplicity, from now on, we will refer to the website areas with the term *items.* There are 5,000 users, and they are represented by sequential numbers between 10,001 and 15,000. Items are represented by numbers between 1,000 and 1,297, even if they are less than 298.

The dataset is an unstructured text file. Each record contains several fields between two and six. The first field is a letter defining what the record contains. There are three main types of records, which are as follows:

- **Attribute (A)**: This is the description of the website area
- **Case (C)**: This is the case for each user, containing its ID
- **Vote (V)**: This is the vote lines for the case

The first column case record is followed by the userID/caseID. The third column contains the user ID/vote given to the website area. The fourth column contains the description of the website area, and the fifth column consists of the URL of the website area.

The following image shows a small set of original data:

```
I,4,"www.microsoft.com","created by getlog.pl"
T,1,"VRoot",0,0,"VRoot"
N,0,"0"
N,1,"1"
T,2,"Hide1",0,0,"Hide"
N,0,"0"
N,1,"1"
A,1277,1,"NetShow for PowerPoint","/stream"
A,1253,1,"MS Word Development","/worddev"
A,1109,1,"TechNet (World Wide Web Edition)","/technet"
A,1038,1,"SiteBuilder Network Membership","/sbnmember"
A,1205,1,"Hardware Supprt","/hardwaresupport"
A,1076,1,"NT Workstation Support","/ntwkssupport"
A,1100,1,"MS in Education","/education"
A,1229,1,"Uruguay","/uruguay"
A,1172,1,"Belgium","/belgium"
A,1173,1,"Microsoft OnLine Institute","/moli"
A,1283,1,"Cinemainia","/cinemania"
A,1167,1,"Windows Hardware Testing","/hwtest"
A,1290,1,"Activate the Internet Conference","/devmovies"
A,1193,1,"Office Developer Support","/offdevsupport"
A,1153,1,"Venezuela","/venezuela"
A,1013,1,"Visual Basic Support","/vbasicsupport"
A,1241,1,"India","/india"
A,1169,1,"MS Project","/msproject"
A,1260,1,"Exchange Trial","/trial"
A,1063,1,"Intranet Strategy","/intranet"
A,1252,1,"Community Affairs","/giving"
```

Our target is to suggest that each user explores some areas of the website that they haven't explored yet.

The following is the list of packages we will be using for this exercise:

```
import pandas as pd
import numpy as np
import scipy
import sklearn
```

Loading the data:

```
path = "~/anonymous-msweb.test.txt"
import pandas as pd
```

Use `read.csv()` function available in pandas package to read the data:

```
raw_data = pd.read_csv(path,header=None,skiprows=7)
raw_data.head()
```

```
In [2]: raw_data.head()
Out[2]:
   0     1  2                                 3                   4
0  A  1277  1           NetShow for PowerPoint             /stream
1  A  1253  1             MS Word Development            /worddev
2  A  1109  1  TechNet (World Wide Web Edition)            /technet
3  A  1038  1     SiteBuilder Network Membership         /sbnmember
4  A  1205  1                  Hardware Supprt  /hardwaresupport
```

Let's see more sample data to have a much clearer idea:

```
In [6]: raw_data
Out[6]:
       0      1      2                                 3                 4
0      A   1277      1            NetShow for PowerPoint           /stream
1      A   1253      1               MS Word Development          /worddev
2      A   1109      1  TechNet (World Wide Web Edition)          /technet
3      A   1038      1     SiteBuilder Network Membership        /sbnmember
4      A   1205      1                   Hardware Supprt  /hardwaresupport
5      A   1076      1             NT Workstation Support      /ntwkssupport
6      A   1100      1                   MS in Education        /education
7      A   1229      1                           Uruguay          /uruguay
8      A   1172      1                           Belgium          /belgium
9      A   1173      1        Microsoft OnLine Institute             /moli
10     A   1283      1                         Cinemainia        /cinemania
11     A   1167      1          Windows Hardware Testing           /hwtest
12     A   1290      1   Activate the Internet Conference        /devmovies
...   ..    ...    ...                               ...               ...
20455  V   1004      1                               NaN               NaN
20456  C  14992  14992                               NaN               NaN
20457  V   1001      1                               NaN               NaN
20458  V   1034      1                               NaN               NaN
20459  V   1004      1                               NaN               NaN
20460  C  14993  14993                               NaN               NaN
20461  V   1010      1                               NaN               NaN
20462  V   1004      1                               NaN               NaN
20463  C  14994  14994                               NaN               NaN
```

We can observe the following from the preceding figure:

- The first column contains three types of values: **A/V/C**, where A represents case ID, **V** represents the user, and **C** represents the case IDs that the user has accessed
- The second column contains IDs to represent users and items
- The third column contains the description of website area
- The fourth contains the URL for the website area on the website

To make an item profile, we use the rows containing **A** in the first column, and to create a user activity or dataset, we use the rows which don't contain **A** in the first column.

Let's get started with profile generation.

Before we proceed toward profile generation, we will have to format the user activity data; the following section explains how to create a user activity dataset.

User activity

In this section, we will create a user-item rating matrix containing users as rows, items as columns, and the value as the cells. Here, the value is either `0` or `1`, indicating `1` if the user has accessed the web page, else `0`:

First we filter only records that don't contain `"A"` in the first column:

```
user_activity = raw_data.loc[raw_data[0] != "A"]
```

Next, we assign then we remove unwanted columns from the dataset:

```
user_activity = user_activity.loc[:, :1]
```

Assigning names to the columns of `user_activity` DataFrame:

```
user_activity.columns = ['category','value']
```

The following code shows the sample `user_activity` data:

```
In [85]: user_activity.head(15)
Out[85]:
    category  value
294        C  10001
295        V   1038
296        V   1026
297        V   1034
298        C  10002
299        V   1008
300        V   1056
301        V   1032
302        C  10003
303        V   1064
304        V   1065
305        V   1020
306        V   1007
307        V   1038
308        V   1026
```

To get the total unique `webid` in the dataset, see as the following code:

```
len(user_activity.loc[user_activity['category'] =="V"].value.unique())
Out[73]: 236
```

To get the unique users count, see following code:

```
len(user_activity.loc[user_activity['category'] =="C"].value.unique())
Out[74]: 5000
```

Now let's run the following code to create a user-item-rating matrix, as follows:

First, we assign variables:

```
tmp = 0
nextrow = False
```

Then we get the last index of the dataset:

```
lastindex = user_activity.index[len(user_activity)-1]
lastindex
Out[77]: 20484
```

The for loop code loops through each record and adds new columns(`'userid'`, `'webid'`) to `user_activity` data frame which shows `userid` and corresponding web activity:

```
for index,row in user_activity.iterrows():
    if(index <= lastindex ):
        if(user_activity.loc[index,'category'] == "C"):
            tmp = 0
            userid = user_activity.loc[index,'value']
            user_activity.loc[index,'userid'] = userid
            user_activity.loc[index,'webid'] = userid
            tmp = userid
            nextrow = True
        elif(user_activity.loc[index,'category'] != "C" and nextrow ==
True):
                webid = user_activity.loc[index,'value']
                user_activity.loc[index,'webid'] = webid
                user_activity.loc[index,'userid'] = tmp
                if(index != lastindex and
user_activity.loc[index+1,'category'] == "C"):
                    nextrow = False
                    caseid = 0
```

```
In [102]: user_activity.head(30)
Out[102]:
    category  value   userid    webid
294        C  10001  10001.0  10001.0
295        V   1038  10001.0   1038.0
296        V   1026  10001.0   1026.0
297        V   1034  10001.0   1034.0
298        C  10002  10002.0  10002.0
299        V   1008  10002.0   1008.0
300        V   1056  10002.0   1056.0
301        V   1032  10002.0   1032.0
302        C  10003  10003.0  10003.0
303        V   1064  10003.0   1064.0
304        V   1065  10003.0   1065.0
305        V   1020  10003.0   1020.0
306        V   1007  10003.0   1007.0
307        V   1038  10003.0   1038.0
308        V   1026  10003.0   1026.0
309        V   1052  10003.0   1052.0
310        V   1041  10003.0   1041.0
311        V   1028  10003.0   1028.0
312        C  10004  10004.0  10004.0
313        V   1004  10004.0   1004.0
314        C  10005  10005.0  10005.0
315        V   1017  10005.0   1017.0
316        V   1156  10005.0   1156.0
317        V   1004  10005.0   1004.0
318        V   1018  10005.0   1018.0
319        V   1008  10005.0   1008.0
320        V   1027  10005.0   1027.0
321        V   1009  10005.0   1009.0
322        V   1046  10005.0   1046.0
323        V   1038  10005.0   1038.0
```

Next, we remove the unwanted rows from the preceding data frame, that is, we will be removing the rows containing "C" in the category column:

```
user_activity = user_activity[user_activity['category'] == "V" ]
```

```
In [104]: user_activity.head()
Out[104]:
    category  value   userid   webid
295        V   1038  10001.0  1038.0
296        V   1026  10001.0  1026.0
297        V   1034  10001.0  1034.0
299        V   1008  10002.0  1008.0
300        V   1056  10002.0  1056.0
```

We subset the columns, and remove the first two columns, which we no longer needed:

```
user_activity = user_activity[['userid','webid']]
```

Next, we sort the data by `webid`; this is to make sure that the rating matrix generation is in good format:

```
user_activity_sort = user_activity.sort('webid', ascending=True)
```

Now, let's create a dense binary rating matrix containing user_item_rating using the following code:

First, we get the size of `webid` column:

```
sLength = len(user_activity_sort['webid'])
```

Then we add a new column, `'rating'` to the `user_activity` data frame which contains only 1:

```
user_activity_sort['rating'] = pd.Series(np.ones((sLength,)),
index=user_activity.index)
```

Next, we use pivot to create binary rating matrix:

```
ratmat = user_activity_sort.pivot(index='userid', columns='webid',
values='rating').fillna(0)
```

Finally, we create a dense matrix:

```
ratmat = ratmat.to_dense().as_matrix()
```

Item profile generation

In this section, we will be creating an item profile from the initial raw data (`raw_data`). To create item data, we will consider the data that contains `A` in the first column:

First, we filter all the records containing first column as `"A"`

```
items = raw_data.loc[raw_data[0] == "A"]
```

Then we name the columns as follows:

```
items.columns = ['record','webid','vote','desc','url']
```

To generate `item` profile we only needed two columns so we slice the dataframe as follows:

```
items = items[['webid','desc']]
```

To see the dimensions of the items, the dataframe is given like:

```
items.shape
Out[12]: (294, 2)
```

We observe that there are `294` unique `webid` in the dataset:

To check the sample of the data, we use the following code:

```
Items.head()
```

```
In [13]: items.head()
Out[13]:
   webid                              desc
0   1277            NetShow for PowerPoint
1   1253                MS Word Development
2   1109  TechNet (World Wide Web Edition)
3   1038    SiteBuilder Network Membership
4   1205                  Hardware Supprt
```

To check the count of unique `webid`, we use the following code:

```
items['webid'].unique().shape[0]
Out[117]: 294
```

We can also only those `webid` which are present in the `user_activity` data:

```
items2 = items[items['webid'].isin(user_activity['webid'].tolist())]
```

We can use the following code check type of the object

```
type(items2)
Out[123]: pandas.core.frame.DataFrame
```

We can also sort the data by `webid`:

```
items_sort = items2.sort('webid', ascending=True)
```

Let'see what we have done till now, using the `head(5)` function:

```
In [122]: items_sort.head(5)
Out[122]:
     webid                      desc
113   1000                    regwiz
40    1001           Support Desktop
278   1002    End User Produced View
102   1003            Knowledge Base
243   1004      Microsoft.com Search
```

Now, we shall create the item profile using the `tf-idf` functions available in the sklearn package. To generate `tf-idf`, we use the `TfidfVectorizer()`. The `fit_transform()` methods are in the `sklearn` package. The following code shows how we can create `tf-idf`.

In the following code, the choice of the number of features to be included depends on the dataset, and the optimal number of features can be selected by the cross-validation approach:

```
from sklearn.feature_extraction.text import TfidfVectorizer
v = TfidfVectorizer(stop_words ="english",max_features = 100,ngram_range=
(0,3),sublinear_tf =True)
x = v.fit_transform(items_sort['desc'])
itemprof = x.todense()
```

```
In [128]: itemprof
Out[128]:
matrix([[ 1.        ,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        [ 0.32213709,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        [ 0.43709646,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        ...,
        [ 0.38159493,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        [ 0.30073274,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        [ 0.36402686,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ]])
```

User profile creation

We now have item profile and user activity in hand; the dot product between these two matrices will create a new matrix with dimensions equal to # of users by # Item features.

To compute the dot product between user activity and item profile, we use the `scipy` package methods such as `linalg`, dot available.

Run the following code to compute the dot product:

```
#user profile creation
from scipy import linalg, dot
userprof = dot(ratmat,itemprof)/linalg.norm(ratmat)/linalg.norm(itemprof)

userprof
```

```
In [130]: userprof
Out[130]:
matrix([[ 0.00062937,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        [ 0.00089668,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        [ 0.00144708,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        ...,
        [ 0.00046412,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        [ 0.00067229,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        [ 0.00079067,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ]])

In [131]: userprof.shape
Out[131]: (5000, 100)
```

Activate Windows

The final step in a recommendation engine model would be to compute the active user preferences for the items. For this, we do a cosine similarity between user profile and item profile.

To compute the cosine calculations, we will be using the `sklearn` package. The following code will calculate the `cosine_similarity`:

We calculate the cosine similarity between `userprofile` an item profile:

```
import sklearn.metrics
similarityCalc = sklearn.metrics.pairwise.cosine_similarity(userprof,
itemprof, dense_output=True)
```

We can see the results of the preceding calculation as follows:

```
In [138]: similarityCalc
Out[138]:
array([[ 0.54168902,  0.17449812,  0.23677035, ...,  0.20670579,
         0.16290362,  0.19718935],
       [ 0.78844617,  0.25398775,  0.34462703, ...,  0.30086706,
         0.23711158,  0.28701558],
       [ 0.63172381,  0.20350167,  0.27612424, ...,  0.29413451,
         0.18998003,  0.22996444],
       ...,
       [ 0.56969503,  0.1835199 ,  0.24901168, ...,  0.21739274,
         0.17132595,  0.20738429],
       [ 0.49394733,  0.15911875,  0.21590263, ...,  0.1884878 ,
         0.14854613,  0.17981009],
       [ 0.86518334,  0.27870764,  0.37816858, ...,  0.33014958,
         0.26018896,  0.31494998]])

In [139]: similarityCalc.shape
Out[139]: (5000, 236)

In [140]:
```

Now, let's format the preceding results calculated as binary data `(0,1)`, as follows:

First, we convert the rating to binary format:

```
final_pred= np.where(similarityCalc>0.6, 1, 0)
```

Then we examine the final predictions of first three users:

```
In [141]: final_pred[1]

In [141]: Out[141]:
array([1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
       1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1,
       0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0,
       0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0,
       0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0,
       0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1,
       0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1,
       0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1,
       0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0,
       0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0,
       1, 0, 0, 0, 0, 0])

In [142]: final_pred[2]
Out[142]:
array([1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0,
       1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1,
       0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0,
       0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0,
       0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0,
       0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1,
       0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1,
       0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1,
       0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0,
       0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0,
       1, 0, 0, 0, 0, 0])

In [143]: final_pred[3]
Out[143]:
array([0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

Removing the zero values from the preceding results gives us the list of the probable items that can be recommended to the users:

For user `213` the recommended items are generated as follows:

```
indexes_of_user = np.where(final_pred[213] == 1)
```

In the preceding code, we are generating recommendations for the active user `213`:

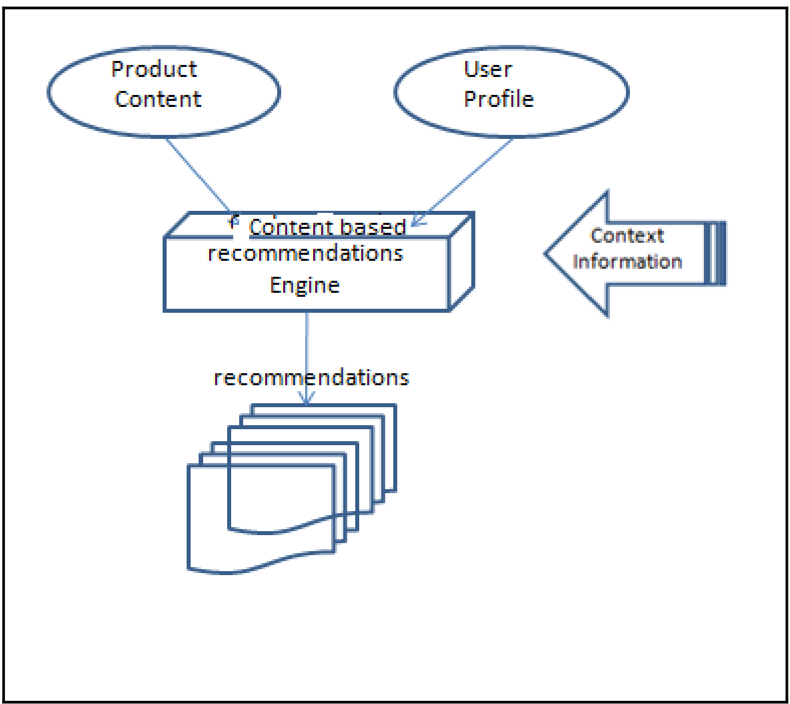
```
In [145]: indexes_of_user
Out[145]: (array([  9,  37,  68, 152], dtype=int64),)
```

Context-aware recommender systems

The next type of personalized recommender system that we will be learning here is context-aware recommender system. These recommender systems are next generation recommendations systems, which fall into the hyper-personalization category. It's natural that there won't be an end to the needs of humans. The more we get, the more we want. Though content-based recommender systems are efficient, targeted at an individual level, and consider the user's personal preferences alone while building recommendation engines, people wanted recommendation engines to be more personalized. For example, a person going on a trip alone may need a book to read whereas the same person may need beer if he is travelling with friends. Similarly, the same person might require diapers, medicines, snacks, and so on if he is going with his own family. People at different places at different times with different company have different needs. Our recommender systems should be robust enough to handle such scenarios. Such hyper personalized recommender systems, which cater to different recommendations to the same person based on his current context, are known as context-aware recommender systems.

Building a context-aware recommender systems

Building a context-aware recommender system is more like extending a content recommender system. Building a context-aware system typically involves adding the context dimension on top of content recommenders, as shown in the following figure:



In the preceding figure, we can observe that context dimension is added on top of a content-based recommendation engine model, and then recommendations are generated. As we discussed in `Chapter 3`, *Recommendation Engines Explained*, there are two popular types of approaches for building context-aware recommendations:

- Pre-filtering approaches
- Post-filtering approaches

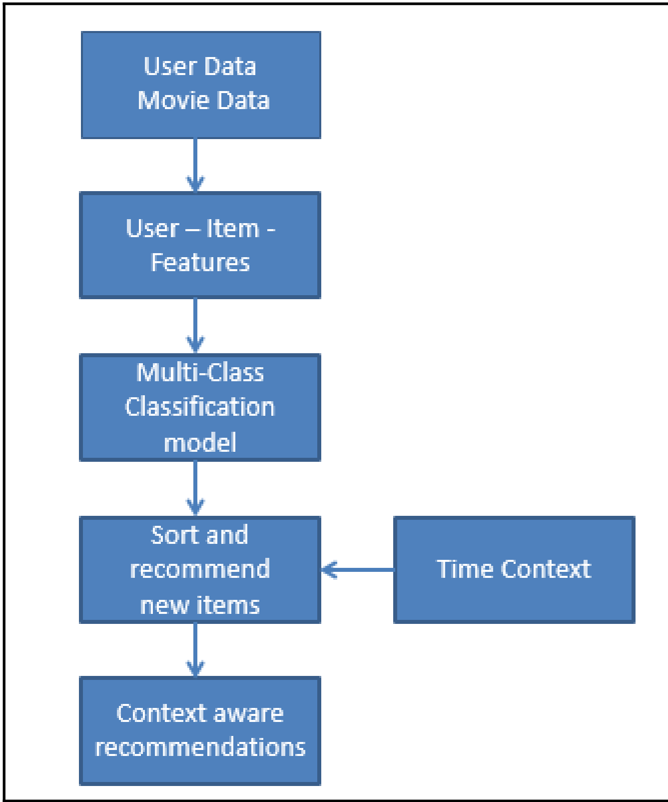
In this section, we will use post-filtering techniques to build context-aware recommender systems.

Context-aware recommendations using R

In the previous section, we built a content-based recommendation engine. In this section, we will extend the previous model to include context information and generate a context-aware recommendation engine.

The usual practice of building context-aware systems is to add a time dimension to the content-based recommendations.

The workflow is shown as follows:



Let's try building context aware systems using R. The steps for building context-aware systems in R are as follows:

1. Define context.
2. Create a context profile with respect to a user for item content.
3. Generate recommendations for a context.

Defining the context

The first step is to define the context that we will be including in our recommendations. In the previous section, we used the MovieLens dataset to build content-based recommendation engines. In the dataset, we have a time component, timestamp, in the rating data. We shall use this variable for our context-aware recommendation systems.

We will extend the R code we used while building content-based recommendations.

We load the full MovieLens ratings dataset as follows:

```
raw_data =
read.csv("C:/Suresh/R&D/packtPublications/reco_engines/drafts/personalRecos
/udata.csv",sep="\t",header=F)
colnames(raw_data) = c("UserId","MovieId","Rating","TimeStamp")
```

See the sample data using `head()` function:

```
> head(raw_data)
  UserId MovieId Rating TimeStamp
1    196     242      3 881250949
2    186     302      3 891717742
3     22     377      1 878887116
4    244      51      2 880606923
5    166     346      1 886397596
6    298     474      4 884182806
> 
```

We load movies dataset:

```
movies =
read.csv("C:/Suresh/R&D/packtPublications/reco_engines/drafts/personalRecos
/uitem.csv",sep="|",header=F)
```

Then we add column names to the movies data frame:

```
colnames(movies) =
c("MovieId","MovieTitle","ReleaseDate","VideoReleaseDate","IMDbURL","Unknow
n","Action","Adventure","Animation","Children","Comedy","Crime","Documentar
y","Drama","Fantasy","FilmNoir","Horror","Musical","Mystery","Romance","Sci
Fi","Thriller","War","Western")
```

Next, we remove unwanted columns from the data frame:

```
movies = movies[,-c(2:5)]
```

```
> head(movies)
  MovieId Unknown Action Adventure Animation Children Comedy Crime Documentary Drama Fantasy FilmNoir Horror Musical Mystery Romance SciFi Thriller War Western
1       1       0      0         0         1        1      1     0           0     0       0        0      0       0       0       0     0        0   0       0
2       2       0      1         1         0        0      0     0           0     0       0        0      0       0       0       0     0        1   0       0
3       3       0      0         0         0        0      0     0           0     0       0        0      0       0       0       0     0        1   0       0
4       4       0      1         0         0        0      1     0           0     1       0        0      0       0       0       0     0        0   0       0
5       5       0      0         0         0        0      0     1           0     1       0        0      0       0       0       0     0        1   0       0
6       6       0      0         0         0        0      0     0           0     1       0        0      0       0       0       0     0        0   0       0
> |
```

We merge the Movies and Ratings datasets using `merge()` function

```
ratings_ctx = merge(x = raw_data, y = movies, by = "MovieId", all.x = TRUE)
```

```
> head(ratings_ctx)
  MovieId UserId Rating TimeStamp Unknown Action Adventure Animation Children Comedy Crime Documentary Drama Fantasy FilmNoir Horror Musical Mystery Romance SciFi
1       1    650      3 891369759       0      0         0         1        1      1     0           0     0       0        0      0       0       0       0     0
2       1    635      4 878879283       0      0         0         1        1      1     0           0     0       0        0      0       0       0       0     0
3       1      1      5 874965758       0      0         0         1        1      1     0           0     0       0        0      0       0       0       0     0
4       1    514      5 875309276       0      0         0         1        1      1     0           0     0       0        0      0       0       0       0     0
5       1    250      4 883263374       0      0         0         1        1      1     0           0     0       0        0      0       0       0       0     0
6       1    210      5 887731052       0      0         0         1        1      1     0           0     0       0        0      0       0       0       0     0
  Thriller War Western
1        0   0       0
2        0   0       0
3        0   0       0
4        0   0       0
5        0   0       0
6        0   0       0
> |
```

The context that we want to introduce to our previous content-based recommendation is the hour of the day, that is, our recommendations will be made as per the time of the day. The set of recommendations for an active user will be different for each hour of the day. Usually, these changes in recommendations are due to the ordering of the recommendations as per the hour. We will see next how we achieve this.

Creating context profile

In the following section, we shall write code to create context profile of the user. We chose the timestamp information available in the dataset and calculate the preference value for movie genres for each user for each hour of the day. This context profile information is used for generating context aware recommendations.

We extract timestamp from the ratings dataset:

```
ts = ratings_ctx$TimeStamp
```

Then, we convert it into a `POSIXlt` date object and using hour property to extract hour of the day:

```
hours <- as.POSIXlt(ts,origin="1960-10-01")$hour
```

See below for sample data:

```
> ts = ratings_ctx$TimeStamp
> head(ts)
[1] 891369759 878879283 874965758 875309276 883263374 887731052
> hours <- as.POSIXlt(ts,origin="1960-10-01")$hour
> head(hours)
[1]  0 10  3  2  4 21
>
```

We can append the hours back on to the ratings dataset:

```
ratings_ctx = data.frame(cbind(ratings_ctx,hours))
```

```
> head(ratings_ctx)
  MovieId UserId Rating TimeStamp Unknown Action Adventure Animation Children Comedy Crime Documentary Drama Fantasy FilmNoir Horror Musical Mystery Romance SciFi
1       1    650      3 891369759       0      0         0         1        1      1     0           0     0       0        0      0       0       0       0     0
2       1    635      4 878879283       0      0         0         1        1      1     0           0     0       0        0      0       0       0       0     0
3       1      1      5 874965758       0      0         0         1        1      1     0           0     0       0        0      0       0       0       0     0
4       1    514      5 875309276       0      0         0         1        1      1     0           0     0       0        0      0       0       0       0     0
5       1    250      4 883263374       0      0         0         1        1      1     0           0     0       0        0      0       0       0       0     0
6       1    210      5 887731052       0      0         0         1        1      1     0           0     0       0        0      0       0       0       0     0
  Thriller War Western hours
1        0   0       0     0
2        0   0       0    10
3        0   0       0     3
4        0   0       0     2
5        0   0       0     4
6        0   0       0    21
>
```

Now, let's start building a context profile for a user with the user ID 943:

Extract ratings information for the active user(943) and removing UserId, MovieId, Rating, Timestamp columns, as shown as follow:

```
UCP = ratings_ctx[(ratings_ctx$UserId == 943),][,-c(2,3,4,5)]
```

```
> head(UCP)
     MovieId Action Adventure Animation Children Comedy Crime Documentary Drama Fantasy FilmNoir Horror Musical Mystery Romance SciFi Thriller War Western hours
496        2      1         1         0        0      0     0           0     0       0        0      0       0       0       0     0        1   0       0     9
1676       9      0         0         0        0      0     0           0     1       0        0      0       0       0       0     0        0   0       0     8
2210      11      0         0         0        0      0     1           0     0       0        0      0       0       0       0     0        1   0       0     9
2356      12      0         0         0        0      0     1           0     0       0        0      0       0       0       0     0        1   0       0     9
3781      22      1         0         0        0      0     0           0     1       0        0      0       0       0       0     0        0   1       0     9
3944      23      0         0         0        0      0     0           0     1       0        0      0       0       0       0     0        1   0       0     9
>
```

As a next step, we compute the columns of all the item features. This columnwise sum is used to compute the preferences for the item features for each hour of the day.

We compute the column wide sum of each column using `aggregate()` function:

```
UCP_pref = aggregate(.~hours,UCP[,-1],sum)
```

```
> head(UCP_pref)
  hours Action Adventure Animation Children Comedy Crime Documentary Drama Fantasy FilmNoir Horror Musical Mystery Romance SciFi Thriller War Western
1     0      8         7         0        1      1     0           0     3       0        0      1       0       0       1     6        0   0       1
2     6      3         2         0        0      0     0           0     2       0        0      0       0       0       1     1        1   1       0
3     8     11         5         0        0     11     4           0    10       0        0      2       0       2       5     4       10   3       0
4     9     38        20         2        8     36    14           0    41       2        0     11       4       1      24     9       23  11       7
5    10      4         1         0        0      8     0           0     1       0        0      0       0       0       2     2        1   0       1
> 
```

From the preceding figure, we can see the time preferences for each of the movie genres for the active user 943. We can observe that during the ninth hour of the day, the user watches more movies, especially action/drama/comedy movies:

We can normalize the preceding data between 0-1 using following function:

```
UCP_pref_sc = cbind(context = UCP_pref[,1],t(apply(UCP_pref[,-1], 1,
function(x)(x-min(x))/(max(x)-min(x)))))
```

```
> head(UCP_pref_sc)
     context    Action Adventure  Animation Children    Comedy     Crime Documentary     Drama    Fantasy FilmNoir    Horror    Musical    Mystery   Romance
[1,]       0 1.0000000 0.8750000 0.00000000 0.125000 0.1250000 0.0000000           0 0.3750000 0.00000000        0 0.1250000 0.00000000 0.00000000 0.1250000
[2,]       6 1.0000000 0.6666667 0.00000000 0.000000 0.0000000 0.0000000           0 0.6666667 0.00000000        0 0.0000000 0.00000000 0.00000000 0.3333333
[3,]       8 1.0000000 0.4545455 0.00000000 0.000000 1.0000000 0.3636364           0 0.9090909 0.00000000        0 0.1818182 0.00000000 0.18181818 0.4545455
[4,]       9 0.9268293 0.4878049 0.04878049 0.195122 0.8780488 0.3414634           0 1.0000000 0.04878049        0 0.2682927 0.09756098 0.02439024 0.5853659
[5,]      10 0.5000000 0.1250000 0.00000000 0.000000 1.0000000 0.0000000           0 0.1250000 0.00000000        0 0.0000000 0.00000000 0.00000000 0.2500000
         SciFi  Thriller       War   Western
[1,] 0.7500000 0.0000000 0.0000000 0.1250000
[2,] 0.3333333 0.3333333 0.3333333 0.0000000
[3,] 0.3636364 0.9090909 0.2727273 0.0000000
[4,] 0.2195122 0.5609756 0.2682927 0.1707317
[5,] 0.2500000 0.1250000 0.0000000 0.1250000
> 
```

Generating context-aware recommendations

Now that we have created the context profile for the active user, let's start generating context-aware recommendations for the user.

For this, we shall reuse the recommend object built using R, which contains content recommendations for all the users.

Let's see the recommendations made to the user 943 using the content-based system:

```
recommend$MovieId
```

```
> recommend$MovieId
  [1]   3   4   5   6   7   8  10  14  15  16  18  19  20  30  32  34  35  36  37  43  44  45  46  47  48  49
 [32]  65  66  70  71  74  75  77  78  81  83  84  86  87  88  89  90  93  99 102 103 106 107 112 113 114 115
 [63] 131 133 134 135 136 137 140 141 142 143 146 149 150 155 156 157 159 160 162 165 166 170 175 178 179 180
 [94] 199 203 206 207 209 212 213 214 220 221 223 224 236 244 245 246 253 255 256 258 262 264 267 268 270 272
[125] 280 283 285 286 287 288 289 291 292 293 296 297 299 302 303 304 305 306 307 308 309 310 311 312 313 315
[156] 324 325 326 327 329 330 331 332 333 336 337 339 340 344 345 346 347 348 351 354 355 357 359 360 361 365
[187] 379 381 382 387 389 392 394 404 408 409 411 413 416 417 418 420 424 429 432 434 436 437 438 439 440 441
[218] 448 451 454 457 458 459 460 461 462 463 464 465 466 467 469 473 474 478 479 480 481 483 484 486 487 488
[249] 499 500 501 503 504 505 506 507 509 510 511 512 513 514 515 517 518 519 520 521 522 523 525 527 528 529
[280] 535 536 537 538 542 543 544 545 547 548 551 552 553 555 556 557 558 560 561 563 565 567 571 573 574 580
[311] 589 591 592 593 594 596 598 600 601 602 603 605 606 607 608 610 611 612 615 616 617 618 619 620 621 622
[342] 632 633 634 635 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 656 657 658 659 660
[373] 669 670 671 673 674 675 676 677 678 681 682 686 690 691 692 693 694 695 696 697 698 699 701 702 703 704
[404] 711 712 713 714 715 716 723 726 727 729 730 731 733 735 736 737 738 740 741 742 744 747 749 750 753 754
[435] 766 767 768 770 772 773 774 775 776 777 778 781 782 783 784 786 787 788 789 791 799 800 803 806 807 809
[466] 817 821 822 823 826 828 832 834 835 836 837 839 842 844 846 847 848 850 851 852 855 856 857 858 860 861
[497] 869 870 872 873 874 875 877 880 882 883 884 885 886 887 888 889 891 895 896 898 899 900 902 903 904 905
```

Now, to these content recommendations, we add our time or hour of the day dimension and then generate recommendations as per the current context.

We merge recommendations and movies dataset using `merge()` function:

```
UCP_pref_content = merge(x = recommend, y = movies, by = "MovieId", all.x =
TRUE)
```

```
> head(UCP_pref_content)
  MovieId predictions Unknown Action Adventure Animation Children Comedy Crime Documentary Drama Fantasy FilmNoir Horror Musical Mystery Romance SciFi Thriller
1       3           1       0      0         0         0        0      0     0           0     0       0        0      0       0       0       0     0        1
2       4           1       0      1         0         0        0      1     0           0     1       0        0      0       0       0       0     0        0
3       5           1       0      0         0         0        0      0     1           0     1       0        0      0       0       0       0     0        1
4       6           1       0      0         0         0        0      0     0           0     1       0        0      0       0       0       0     0        0
5       7           1       0      0         0         0        0      0     0           0     1       0        0      0       0       0       0     1        0
6       8           1       0      0         0         0        1      1     0           0     1       0        0      0       0       0       0     0        0
  War Western
1   0       0
2   0       0
3   0       0
4   0       0
5   0       0
6   0       0
>
```

With the preceding step, we have computed all the required matrices, user context profile (`UCP_Pref_SC`) and user content recommendations (`UCP_Pref_content`).

Suppose we want to generate recommendations for the user at the ninth hour; we just need to perform an element wise multiplication of user content recommendations and the context row for the ninth hour of the day from the `UCP_pref_SC` object. This is given as follows:

Performing element wise multiplication for the User content recommendations and the ninth hour context preferences for the user:

```
active_user =cbind(UCP_pref_content$MovieId,(as.matrix(UCP_pref_content[,-
c(1,2,3)]) %*% as.matrix(UCP_pref_sc[4,2:19])))
```

The results can be seen as follows; we can observe that the preference for MovieId 3 is 0.5 where as for MovieId 4 the preference is 2.8

```
> head(active_user)
     [,1]      [,2]
[1,]    3 0.5609756
[2,]    4 2.8048780
[3,]    5 1.9024390
[4,]    6 1.0000000
[5,]    7 1.2195122
[6,]    8 2.0731707
> 
```

We can create a dataframe object of the prediction object:

```
active_user_df = as.data.frame(active_user)
```

Next, we add column names to the predictions object:

```
names(active_user_df) = c('MovieId','SimVal')
```

Then we sort the results:

```
FinalPredicitons_943 = active_user_df[order(-active_user_df$SimVal),]
```

Summary

In this chapter, we learned how to build content-based recommendation engines and context-aware recommendation engines using R and Python. We modelled content-based recommendation engines in two types–the classification model and the tf-idf model approaches using R and Python. To build context-aware recommendations, we simply did an element wise multiplication between content-based recommendations and context profile of the user.

In the next chapter, we will be exploring Apache Spark, to build scalable, real-time recommendation engines.

7

Building Real-Time Recommendation Engines with Spark

In this day and age, the need to build scalable real-time recommendations is increasing day by day. With more internet users using e-commerce sites for their purchases, these e-commerce sites have realized the potential of understanding the patterns of the users' purchase behavior to improve their business, and to serve their customers on a very personalized level. To build a system which caters to a huge user base and generates recommendations in real time, we need a modern, fast scalable system. **Apache Spark**, which is a special framework designed for distributed in-memory data processing, comes to our rescue. Spark applies a set of transformations and actions to distributed data to build real-time data mining applications.

In the previous chapters, we learned about implementing similarity-based collaborative filtering approaches, such as user-based collaborative filtering and content-based collaborative filtering. Though the similarity-based approaches are a huge success in commercial applications, there came into existence model-based recommender models, such as matrix factorization models, which have improved the performance of recommendation engine models. In this chapter, we will learn about the model-based approach of collaborative filtering, moving away from the heuristic-based similarity approaches. Also, we will focus on implementing the model-based collaborative filtering approach using Spark.

In this chapter, we will learn about the following:

- What is in Spark 2.0
- Setting up the pyspark environment
- Basic Spark concepts
- The MLlib recommendation engine module
- The Alternating Least Squares algorithm
- Data exploration of the Movielens-100k dataset
- Building model-based recommendation engines using ALS
- Evaluating the recommendation engine model
- Parameter tuning

About Spark 2.0

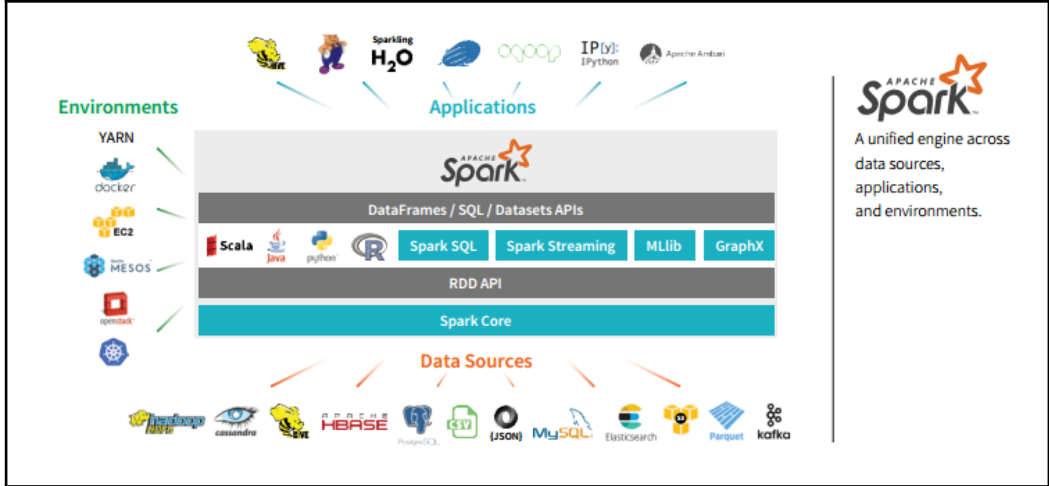
Apache Spark is a fast, powerful, easy-to-use, distributed, in-memory, and open source cluster computing framework built to perform advanced analytics. It was originally developed at UC Berkeley in 2009. Spark has been widely adopted by enterprises across a wide range of industries since its inception.

One of the main advantages of Spark is that it takes all the complexities away from us, such as resources scheduling, job submissions, executions, tracking, between-node communication, fault tolerance, and all low-level operations that are inherent features of parallel processing. The Spark framework helps us write programs to run on the clusters in parallel.

Spark can be run both as a standalone mode and as a cluster mode. Spark can be easily integrated with Hadoop platforms.

As a general-purpose computing engine, Spark with its in-memory data processing capability and easy-to-use APIs allows us to efficiently work on a wide range of large-scale data processing tasks, such as streaming applications, machine learning, or interactive SQL queries over large datasets that require iterative access.

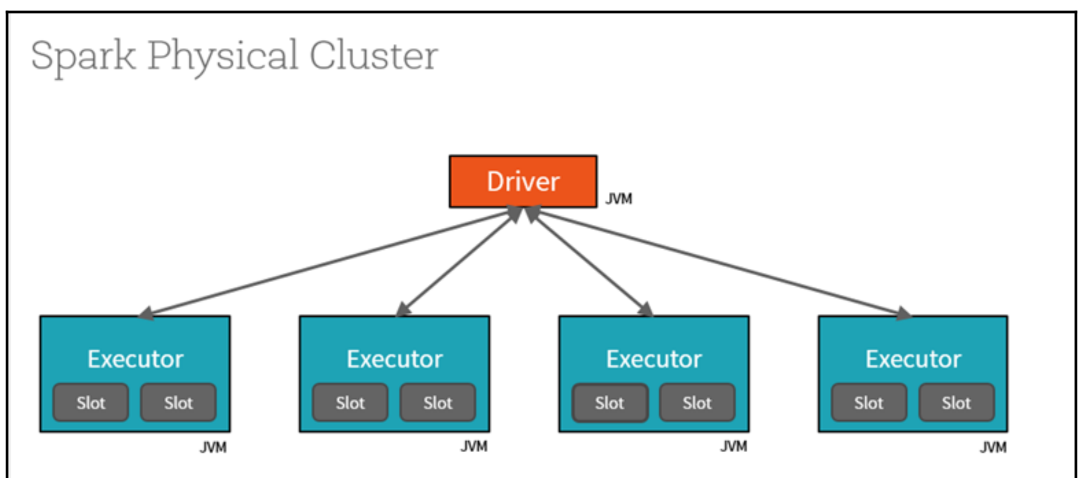
Spark can be easily integrated with many applications, data sources, storage platforms, and environments, and exposes high-level APIs in Java, Python, and R to work with. Spark has proved to be broadly useful for a wide range of large-scale data processing tasks, over and above machine learning and iterative analytics.



Credits: Databricks

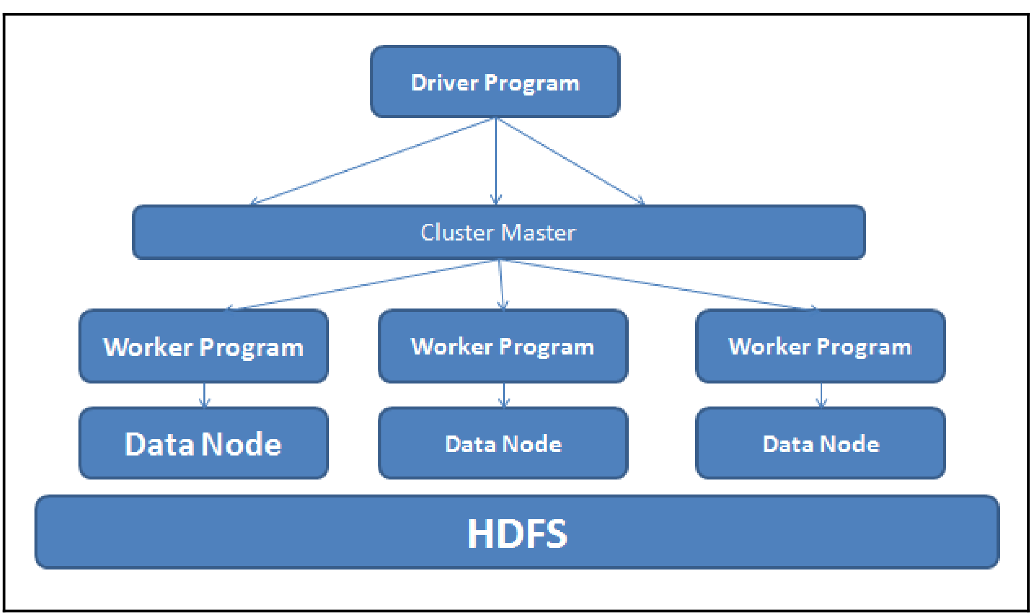
Spark architecture

The Apache Spark ecosystem contains many components to work with distributed, in-memory, and machine-learning data processing tools. The main components of Spark are discussed in the following sub-sections. Spark works on a master-slave architecture; a high-level architecture is shown in the following diagram:



Credits: Databricks

The Spark cluster works on the master-slave architecture. The Spark Core execution engine accepts requests from clients and passes them to the master node. The driver program in the master communicates with the worker node executors to get the work done as shown in the following diagram:



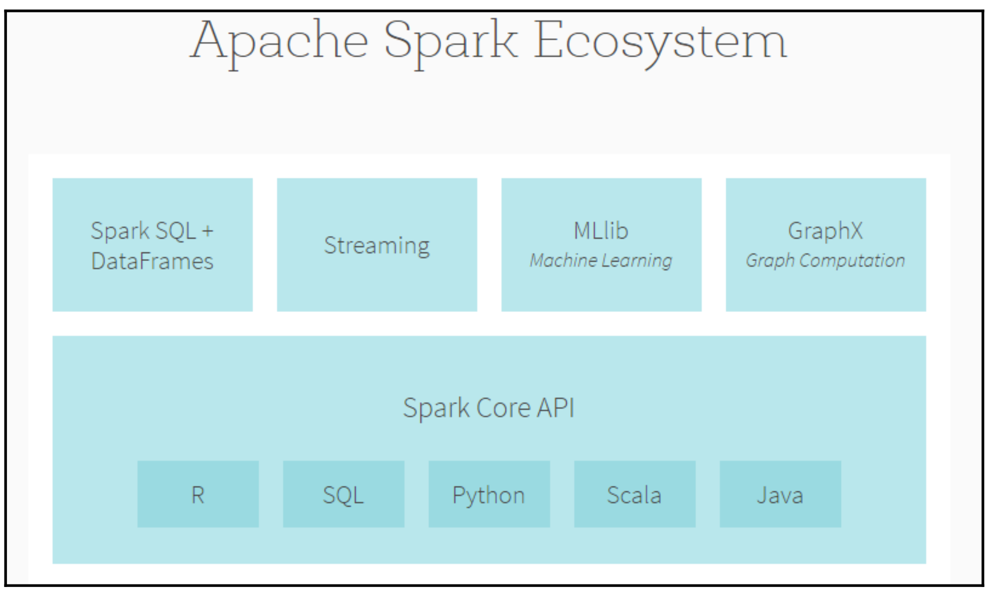
Spark driver program: The driver program acts as a master node in a Spark cluster, which hosts the SparkContext for Spark applications. It receives the client request and co-ordinates with the cluster manager which manages the worker nodes. The driver program splits the original request into tasks and schedules them to run on executors in worker nodes. All the processes in Spark are Java processes. The SparkContext creates **Resilient Distributed Datasets** (**RDD**), an immutable, distributable collection of datasets partitioned across nodes, and performs a series of transformations and actions to compute the final output. We will learn more about RDDs in the latter sections.

Worker nodes: A worker contains executors, where the actual task execution happens in the form of Java processes. Each worker runs its own Spark instance and is the main compute node in Spark. When a SparkContext is created, each worker node starts its own executors to receive the tasks.

Executors: These are the main task executioners of Spark applications.

Spark components

In this section, we will see the core components of the Spark ecosystem. The following diagram shows the Apache Spark Ecosystem:



Credits:Databricks

Spark Core

Spark Core is the core part of the Spark platform: the execution engine. All other functionalities are built on top of Spark Core. This provides all the capabilities of Spark, such as in-memory distributed computation, and fast, easy-to-use APIs.

Structured data with Spark SQL

Spark SQL is a component on top of Spark Core. It is a spark module that provides support for structured and semi-structured data.

Spark SQL provides a unified approach, to allow users to query the data objects in an interactive SQL type, such as applying select, where you can group data objects by the kind of operations through data abstraction APIs, such as DataFrames.

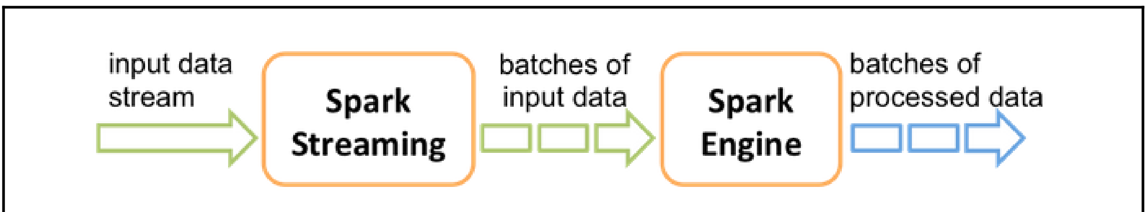
A considerable amount of time will be dedicated to data exploration, exploratory analysis, and SQL-like interactions. Spark SQL, which provides DataFrames, also acts as a distributed SQL query engine; for instance, in R, the DataFrames in Spark 2.0, the data is stored as rows and columns, access to which is allowed as an SQL table with all the structural information, such as data types.

Streaming analytics with Spark Streaming

Spark Streaming is another Spark module that enables users to process and analyze both batch and streaming data in real time, to perform interactive and analytical applications. Spark Streaming provides **Discretized Stream** (**DStream**), a high-level abstraction, to represent a continuous stream of data.

The main features of Spark Streaming API are as follows:

- Scalable
- High throughput
- Fault-tolerant
- Processes live stream of incoming data
- Can connect to real-time data sources and process real-time incoming data on the go
- Can apply complex machine learning and graph processing algorithms on streaming data



Machine learning with MLlib

MLlib is another module in Spark, built on top of Spark Core. This machine learning library is developed with an objective to make practical machine learning scalable and easy to use.

This library provides tools for data scientists, such as the following:

- Machine learning algorithms for regression, classification, clustering, and recommendation engines
- Feature extraction, feature transformation, dimensionality reduction, and feature selection
- Pipeline tools for streamlining machine learning processes for construction, evaluation, and tuning the over process of solving a machine learning problem
- Persistence of storing and loading machine learning models and pipelines
- Utilities, such as linear algebra and statistical tasks

When starting Spark 2.0, the old MLlib model is replaced with ML library, which is built with DataFrames APIs, providing more optimizations and making uniform APIs across all languages.

Graph computation with GraphX

GraphX is a new Spark API for building graph-based systems. It is a graph-parallel processing computation engine and distributed framework, which is built on top of Spark Core. This project was started with the objective of unifying the graph-parallel and data distribution framework into a single Spark API. GraphX enables users to process data both as RDDs and graphs.

GraphX provides many features, such as the following:

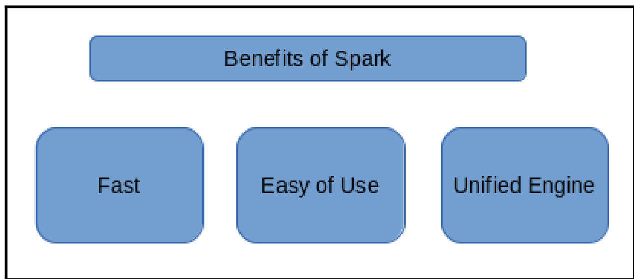
- Property graphs
- Graph-based algorithms, such as PageRank, connected components, and Graph Builders, which are used to build graphs
- Basic graph computational components, such as subgraph, joinVertices, aggregateMessages, Pregel API, and so on

Though graph models are not within the scope of this book, we will learn some fundamentals of graph applications in `Chapter 8`, *Building Real-Time Recommendations with Neo4j*.

Benefits of Spark

The main advantages of Spark are, it is fast, has in-memory framework, contains several APIs, making it very easy to use, its Unified Engine for large quantities of data, and its machine learning components. Unlike Map-Reduce model with its batch mode, which is slower and contains lot of programming, Spark is faster, with real-time and easy to code framework.

The following diagram shows the above mentioned benefits:



Setting up Spark

Spark runs on both Windows and UNIX-like systems (for example, Linux, Mac OS). It's easy to run locally on one machine; all you need is to have Java installed on your system PATH or the `JAVA_HOME` environment variable pointing to a Java installation.

Spark runs on Java 7+, Python 2.6+/3.4+ and R 3.1+. For the Scala API, Spark 2.0.0 uses Scala 2.11. You will need to use a compatible Scala version (2.11.x).

Get Spark from the downloads page of the project website:

```
http://d3kbcqa49mib13.cloudfront.net/spark-2.0.0-bin-hadoop2.7.tgz
```

Spark needs to be built against a specific version of Hadoop in order to access **Hadoop Distributed File System** (**HDFS**), as well as standard and custom Hadoop input sources.

Spark requires the Scala programming language (version 2.10.4 at the time of writing this book) in order to run. Fortunately, the prebuilt binary package comes with the Scala runtime packages included, so you don't need to install Scala separately in order to get started. However, you will need to have a **Java Runtime Environment** (**JRE**) or **Java Development Kit** (**JDK**) installed (take a look at the software and hardware list in this book's code bundle for installation instructions).

Once you have downloaded the Spark binary package, unpack the contents of the package and change it into the newly created directory by running the following commands:

```
tar xfvz spark-2.0.0-bin-hadoop2.7.tgz
cd spark-2.0.0-bin-hadoop2.7
```

Spark places user scripts to run Spark in the bin directory. You can test whether everything is working correctly by running one of the example programs included in Spark:

```
------------------
./bin/run-example org.apache.spark.examples.SparkPi
16/09/26 15:20:36 INFO DAGScheduler: Job 0 finished: reduce at
SparkPi.scala:38, took 0.845103 s
Pi is roughly 3.141071141071141
--------------------
```

You can run Spark interactively with Scala using the following command:

```
./bin/spark-shell --master local[2]
```

The `--master` option specifies the master URL for a distributed cluster, or you can use `local` to run locally with one thread or `local[N]` to run locally with N threads. You should start by using `local` for testing. For a full list of options, run the Spark shell with the `--help` option.

Source:
`http://spark.apache.org/docs/latest/`

About SparkSession

From Spark 2.0, the `SparkSession` will be the entry point for Spark applications. The `SparkSession` serves as the main interactive access point for underlying Spark functionalities and Spark programming capabilities, such as DataFrames API and Dataset API. We use `SparkSession` to create DataFrame objects.

In the earlier versions of Spark, we used to create `SparkConf`, `SparkContext`, or `SQLContext` to interact with Spark, but since Spark 2.0, this has been taken care of by `SparkSession` by encapsulating `SparkConf`, `SparkContext` automatically.

When you start Spark in the shell command, `SparkSession` is created automatically as `spark`

We can programmatically create SparkSession, as follows:

```
spark = SparkSession\
     .builder\
     .appName("recommendationEngine")\
      config("spark.some.config.option", "some-value")\
     .getOrCreate()
```

Resilient Distributed Datasets (RDD)

The core of Spark is Resilient Distributed Datasets, in short, RDD. RDD is an immutable distributed collection of objects of some datatype of your data, partitioned across nodes on your cluster. This RDD is fault-tolerant, that is, a property of the system that is able to operate continuously, even in the event of failure by reconstructing the failed partition.

In short, we can say that RDD is a distributed dataset abstraction, which allows iterative operations on very large-scale cluster systems in a fault-tolerant way.

RDDs can be created in a number of ways, such as parallelizing an existing collection of data objects, or referencing an external file system, such as HDFS:

Creating RDD from an existing data object:

```
coll = List("a", "b", "c", "d", "e")

rdd_from_coll = sc.parallelize(coll)
```

Creating RDD from a referenced file:

```
rdd_from_Text_File = sc.textFile("testdata.txt")
```

RDD supports two types of operations:

- **Transformations**: This operation creates new RDDs from existing RDDs, which are immutable
- **Actions**: This operation returns values after performing computation on the dataset

These RDD transformations are executed lazily only when the final results are required. We can recreate or recompute the RDDs any number of times, or we can persist them by caching them in memory if we know we may need them in the future.

About ML Pipelines

The ML Pipelines API in Spark 2.0 is the way to use a standard workflow when solving machine learning problems. Every machine learning problem will undergo a sequence of steps, such as the following:

1. Loading data.
2. Feature extraction.
3. Model training.
4. Evaluation.
5. Predictions.
6. Model tuning.

If we closely observe the aforementioned steps, we can see the following:

- The ML process follows a series of steps as if it is a workflow.
- Often, we require more than one algorithm while solving a machine learning problem; for example, a text classification problem might require a feature extraction algorithm for feature extraction.
- Generating predictions on test data may require many data transformations or data-preprocessing steps, which are used during model training. For example, in text classification problems, making predictions on test data involves data pre-processing steps, such as tokenization and feature extraction, before applying them to a generated classification model, which was used during model creation on training data.

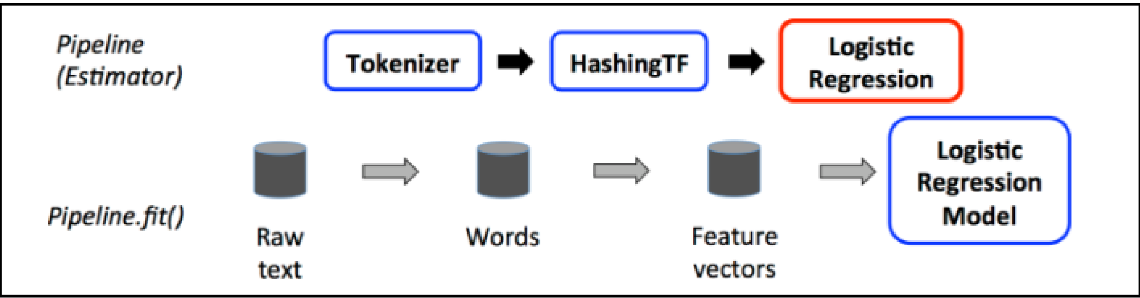
The preceding steps form one of the main motivating factors behind introducing the ML Pipeline API. The ML Pipeline module allows users to define a sequence of stages, so that it's easy to use. The API framework allows the ML process to scale on a distributed platform and accommodate very large datasets, reuse some components, and so on.

The components of the ML Pipeline module are as follows:

- **DataFrame**: As mentioned earlier, DataFrame is a way of representing the data in the Spark framework.
- **Transformers**: Transformers take input DataFrames and transform data into new DataFrames. Transformation classes contain the `transform()` method to do the transformations.

- **Estimators**: Estimators compute the final results. An Estimator class makes use of the `fit()` method to compute results.
- **Pipeline**: This is a set of Transformers and Estimators stacked as a workflow.
- **Parameters**: This refers to the set of parameters that may be used by both Transformers and Estimators.

We'll illustrate this for the simple text document workflow. The following figure is for the training time usage of a pipeline:



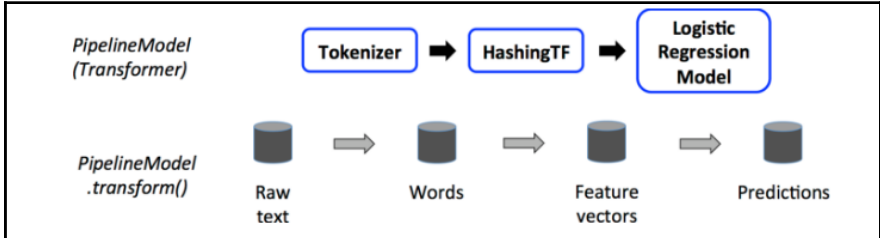
Find below the explanation for the steps shown in the preceding figure. The blue boxes are Transformers and the red box is the Estimator.

1. The Tokenizer Transformer takes the text column of a DataFrame as input and returns a new DataFrame column containing tokens.
2. The HashingTF Transformer takes in the tokens DataFrame from the previous step as input and creates new DataFrame features as output.
3. Now the LogisticRegression Estimator takes in the features DataFrame, fits a logistic regression model, and creates a PipelineModel transformer.

First we build a pipeline, which is an Estimator, then on this Pipeline we apply `fit()` method which produces a PipelineModel, a Transformer, that can be used on test data or at prediction time.

Source:
`http://spark.apache.org/docs/latest/ml-guide.html`

The following figure illustrates this usage:

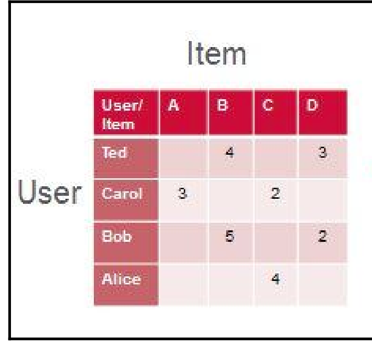


In the preceding figure, when we want to make predictions on the test data, we observe that first the test data has to pass through a series of data-preprocessing steps, which are very much identical to the aforementioned training step. After the pre-processing step is completed, the features of the test data are applied to the logistic regression model.

In order to make the data-preprocessing and feature extraction steps identical, we will pass the test data to the PipelineModel (logistic regression model) by calling the `transform()` to generate the predictions.

Collaborative filtering using Alternating Least Square

In this section, let's explain the Matrix Factorization Model (MF) and the Alternating Least Squares method. Before we get to know about the Matrix Factorization Model, we'll define the objective once again. Imagine we have ratings given to items by a number of users. Let's define the ratings given by users on items in a matrix form given by *R*, as shown in the following diagram:



Item

User/ Item	A	B	C	D
Ted		4		3
Carol	3		2	
Bob		5		2
Alice			4	

User

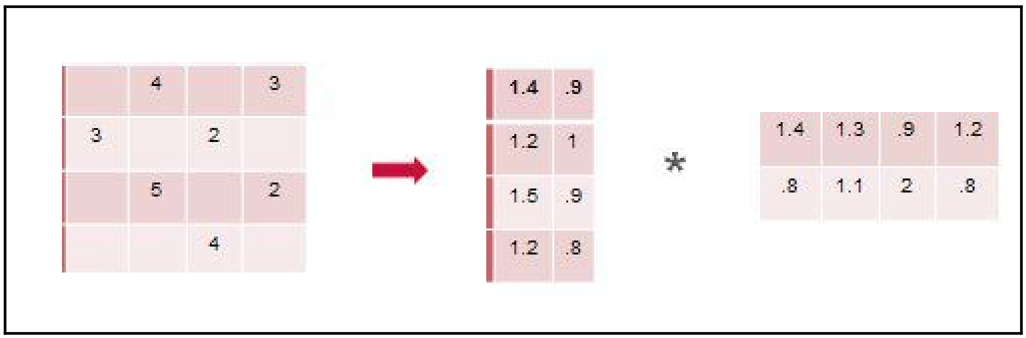
In the preceding diagram, we observe that user Ted has rated items B and D as 4 and 3 respectively. In a collaborative filtering approach, the first step before generating recommendations is to fill the empty spaces, that is, to predict the non-rated items. Once the non-rated item ratings are filled, we suggest new items to the users by ranking the newly filled items.

In the previous chapters, we have seen neighbouring methods using Euclidean distances and cosine distances to predict the missing values. In this section, we will adopt a new method to fill the missing non-rated items. This approach is called the matrix factorization method. This is a mathematical approach, which uses matrix decomposition methods. This method is explained as follows:

A matrix can be decomposed into two low rank matrices, which, when multiplied back, will result in a single matrix approximately equal to the original matrix.

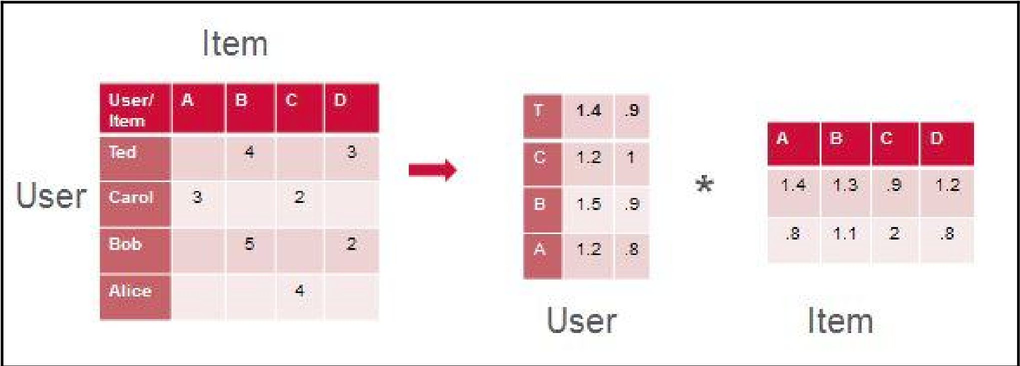
Let's say a rating matrix *R* of size *U X M* can be decomposed into two low rank matrices *P* and *Q* of size *U X K* and *M X K* respectively, where *K* is called the rank of the matrix.

In the following example, let's say the original matrix of size *4 X 4* is decomposed into two matrices: *P (4 X 2)* and *Q (4 X 2)*. Multiplying back *P* and *Q* will give us the original matrix of size *4 X 4* and values approximately equal to those of the original matrix:



The principle of matrix factorization is used in recommendation engines, to fill the non-rated items. The assumption in applying the aforementioned principle to recommendation engines is that the ratings given by users on items are based on some latent features. These latent features are applicable to both users and items, that is, a user rates an item because of some of his personal preferences for it. Also, the user rates items because of certain features of the items.

Using this assumption, when the matrix factorization method is applied to the ratings matrix, we decompose the original ratings matrix into two matrices follows as user-latent factor matrix, P, and item-latent factor matrix:



Now, let's come back to the machine learning approach; you must be wondering what the learning in this approach is. Observe the following formula:

$$\min_{q^*,p^*} \sum_{(u,i)\in\kappa} (r_{ui} - q_i^T p_u)^2 + \lambda(\| q_i \|^2 + \| p_u \|^2)$$

We have learnt that when we multiply back the two latent factor matrices, we get the approximate original matrix. Now, in order to improve the accuracy of the model, that is, to learn the optimal factor vectors, P and Q, we define an optimization function, shown in the preceding formula, which minimizes the regularized squared error between the original ratings matrix and the resultant after the product of the latent matrices. The latter part of the preceding equation is the regularization imposed to avoid over-fitting.

Alternating Least Squares is the optimization technique to minimize the aforementioned loss function. In general, we use stochastic gradient descent to optimize the loss function. For the Spark recommendation module, the ALS technique has been used for minimizing the loss function.

In the ALS method, we calculate the optimal latent factor vectors alternatively by fixing one of the two factor vectors, that is, we calculate the user latent vector by fixing the item-latent feature vector as constant and vice versa.

The main benefits of the ALS approach are as follows:

- This approach can be easily parallelized
- In most cases, we deal with sparse datasets in recommendation engine problems, and ALS is more efficient in handling sparsity compared to the stochastic gradient descent

The Spark implementation of the recommendation engine module in spark.ml has the following parameters:

- **numBlocks**: This is the number of blocks the users and items will be partitioned into in order to parallelize computation (defaults to 10)
- **rank**: This refers to the number of latent factors in the model (defaults to 10)
- **maxIter**: This is the maximum number of iterations to run (defaults to 10)
- **regParam**: This parameter specifies the regularization parameter in ALS (defaults to 0.1)
- **implicitPrefs**: This parameter specifies whether to use the explicit feedback ALS variant or the one adapted for implicit feedback data (defaults to false, which means it's using explicit feedback)
- **alpha**: This is a parameter applicable to the implicit feedback variant of ALS, which governs the baseline confidence in preference observations (defaults to 1.0)
- **nonnegative**: This parameter specifies whether or not to use non-negative constraints for least squares (defaults to false)

Model based recommender system using pyspark

Software details for the use case are as follows:

- Spark 2.0
- Python API: pyspark
- Centos 6
- Python 3.4

Start the Spark session using pyspark, as follows:

```
pyspark
```

The following screenshot shows the Spark session created by running the above `pyspark` command:

```
Python 3.4.3 (default, Sep 16 2015, 10:42:38)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-11)] on linux

Type "help", "copyright", "credits" or "license" for more
information.
Using Spark's default log4j profile: org/apache/spark/log4j-
defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel).

16/10/04 09:53:13 WARN NativeCodeLoader: Unable to load native-hadoop
library for your platform... using builtin-java classes where
applicable
16/10/04 09:53:13 WARN Utils: Your hostname, 01hw745020.tcs-
mobility.com resolves to a loopback address: 127.0.0.1; using
10.132.252.116 instead (on interface eth0)
16/10/04 09:53:13 WARN Utils: Set SPARK_LOCAL_IP if you need to bind
to another address
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.0.0
      /_/

Using Python version 3.4.3 (default, Sep 16 2015 10:42:38)

SparkSession available as 'spark'.
```

To build the recommendation engine using Spark, we make use of Spark 2.0 capabilities, such as DataFrames, RDD, Pipelines, and Transforms available in Spark MLlib, which has was explained earlier.

Unlike earlier heurist approaches, such as k-nearest neighboring approaches used for building recommendation engines, in Spark, matrix factorization methods are used for building recommendation engines and the Alternating Least Squares (ALS) method is used for generating model-based collaborative filtering.

MLlib recommendation engine module

In this section, let's learn about the different methods present in the MLlib recommendation engine module. The current recommendation engine module helps us build the model-based collaborative filtering approach using the Alternating Least Squares matrix factorization model to generate recommendations.

The main methods available for building collaborative filtering are as follows:

- `ALS()`: The `ALS()` constructor is invoked and its instance is created with all the required parameters, such as user column name, item column name, rating column name, rank, regularization parameter (regParam), maximum iterations (maxIter), and so on supplied.
- `fit()`: The `fit()` method is used to generate the model. This method takes the following parameters:
    - `dataset`: input dataset of type `pyspark.sql.DataFrame`(`http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame`)
    - `params`: this is an optional param map which contains required parameters listed above.
    - Returns: The `fit()` method returns fitted models.
- `Transform()`: The `transform()` method is used to generate the predictions. The `transform()` method takes in the following:
    - Test data (DataFrame datatype)
    - Optional additional parameters that embed previously defined parameters.
    - Returns a predictions (DataFrame object)

The recommendation engine approach

Now let's get into the actual implementation of the recommendation engine. We use the following approach to build the recommendation engine using Spark:

1. Start the Spark environment.
2. Load the data.
3. Explore the data source.
4. Use the MLlib recommendation engine module to generate the recommendations using ALS instance.

5. Generate the recommendations.
6. Evaluate the model.
7. Using the cross_validation approach, apply the parameter tuning model to tune the parameter and select the best model, and then generate recommendations.

Implementation

Like any other recommendation engine, the first step is to load the data into the analytics environment (into the Spark environment in our case). When we start the Spark environment in the 2.0 version, SparkContext and SparkSession will be created at the load time.

Before we get into the implementation part, let's review the data for a while. In this chapter, we use the MovieLens 100K Dataset to build collaborative filtering recommendation engines, both user-based and item-based. The dataset contains 943 user ratings on 1,682 movies. The ratings are on a scale of 1-5.

As a first step, we shall make use of SparkContext (sc) to load the data into the Spark environment.

Data loading

To load the data, run the below command:

```
data = sc.textFile("~/ml-100k/udata.csv")
```

Loaded data will be a spark RDD type-run the below command to find out the data type of the data object:

```
type(data)
<class 'pyspark.rdd.RDD'>
```

Total length of the data loaded is given by:

```
data.count()
100001
```

To load the first record in the loaded data:

```
data.first()
'UserID\tItemId \tRating\tTimestamp'
```

We can see that the header information is located as the first row in the data object separated by `\t`; the column names of the data object are `UserID`, `ItemId`, `Rating`, and `Timestamp`.

For our purposes, we don't require Timestamp information, so we can remove this field from the data RDD:

To check the first 5 rows of the data RDD, we use take() action method:

```
data.take(5)
['UserID\tItemId \tRating\tTimestamp', '196\t242\t3\t881250949',
'186\t302\t3\t891717742', '22\t377\t1\t878887116', '244\t51\t2\t880606923']
```

The MLlib recommendation engine module expects the data to be without any header information. So let's remove the header information, that is, remove the first line from the data RDD object, as follows:

Extract the first row from the data RDD object:

```
header = data.first()
```

Use `filter()` method and lambda expression to remove the first header row from the data. The lambda expression below is applied for each row and each row is compared with the header to check if the extracted row is the header or not. If the extracted row is found to be the header then that row is filtered:

```
data = data.filter(lambda l:l!=header)
```

Now let us check the count of the data RDD object; it has reduced from 100001 to 100000:

```
data.count()
100000
```

Now let us check the first row, we can observe that the header has been successfully removed:

```
data.first()
'196\t242\t3\t881250949'
```

Now that we have loaded the data into the Spark environment, let's format the data into a proper shape, as follows:

1. Load the required functions for building the recommendation engine, such as ALS, the Matrix Factorization Model, and the Rating function from the MLlib recommendation module.
2. Extract each row from the data RDD and split by `\t` to separate each column using the `map()` and lambda expressions.
3. In the resultant set, let's create a Rating row object for each of the lines extracted in the previous step
4. When the following expression is applied on the entire dataset, a pipelined RDD object is created:

```
from pyspark.mllib.recommendation import ALS,
MatrixFactorizationModel, Rating
ratings = data.map(lambda l: l.split('\t'))\
    .map(lambda l: Rating(int(l[0]), int(l[1]), float(l[2])))
```

Check the data type of ratings object using type:

```
type(ratings)
<class 'pyspark.rdd.PipelinedRDD'>
```

Check the first 5 records of the ratings PipelinedRDD object by running the following code:

```
ratings.take(5)
[Rating(user=196, product=242, rating=3.0), Rating(user=186, product=302,
rating=3.0), Rating(user=22, product=377, rating=1.0), Rating(user=244,
product=51, rating=2.0), Rating(user=166, product=346, rating=1.0)]
```

We can observe from the preceding result that each row in the original raw data RDD object turns into a kind of list of Rating row objects stacked into PipelinedRDD.

Data exploration

Now that we have loaded the data, let's spend some time exploring the data. Let's use the Spark 2.0 DataFrame API capabilities to explore the data:

Compute the total number of unique users by first selecting the `'user'` column and then using `distinct()` function to remove the duplicate `userId`:

```
df.select('user').distinct().show(5)
```

The following screenshot shows the results of the previous query:

```
+----+
|user|
+----+
|  26|
|  29|
| 474|
| 191|
|  65|
+----+
only showing top 5 rows
```

Total number of unique users:

```
df.select('user').distinct().count()
943
```

Total number of unique items:

```
df.select('product').distinct().count()
1682
```

Display first 5 unique products:

```
df.select('product').distinct().show(5)
```

The following screenshot shows the results of the previous query:

```
+-------+
|product|
+-------+
|    474|
|     29|
|     26|
|    964|
|   1677|
+-------+
only showing top 5 rows
```

Number of rated products by each user:

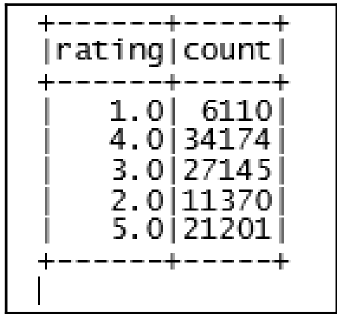
```
df.groupBy("user").count().take(5)
[Row(user=26, count=107), Row(user=29, count=34), Row(user=474, count=327),
Row(user=191, count=27), Row(user=65, count=80)]
```

The previous results explain that User 26 has rated 107 movies and user 29 has rated 34 movies.

Number of records for each rating type:

```
df.groupBy("rating").count().show()
```

The following screenshot shows the results of the previous query:

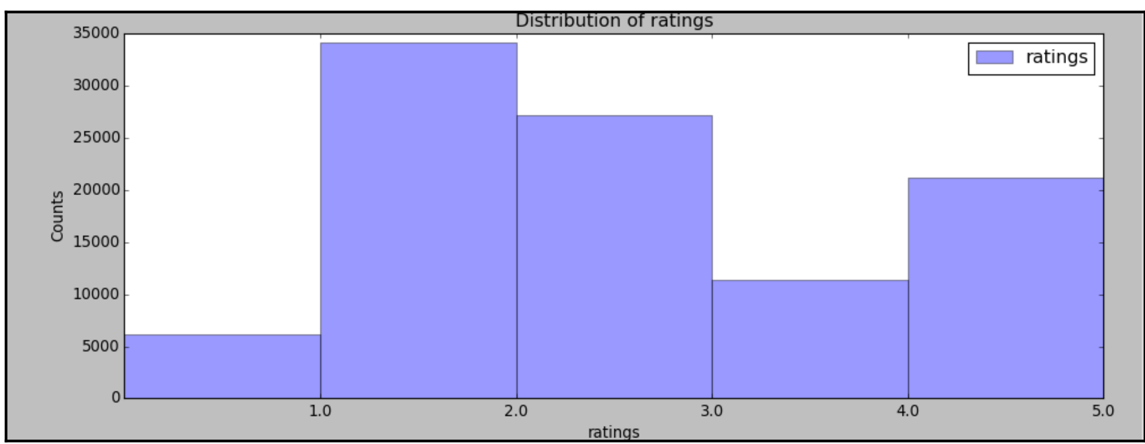


rating	count
1.0	6110
4.0	34174
3.0	27145
2.0	11370
5.0	21201

In the following code, we make use of the `numpy` scientific computing package in Python, used for working with arrays: `matplotlibe` – a visualizing package in Python:

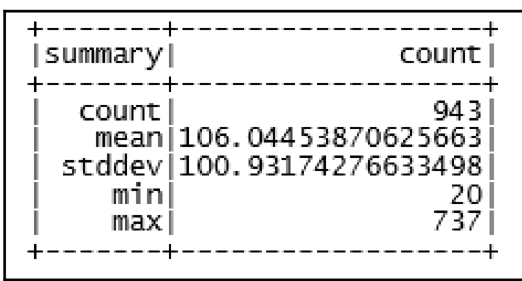
```
import numpy as np
import matplotlib.pyplot as plt
n_groups = 5
x = df.groupBy("rating").count().select('count')
xx = x.rdd.flatMap(lambda x: x).collect()
fig, ax = plt.subplots()
index = np.arange(n_groups)
bar_width = 1
opacity = 0.4
rects1 = plt.bar(index, xx, bar_width,
                 alpha=opacity,
                 color='b',
                 label='ratings')
plt.xlabel('ratings')
plt.ylabel('Counts')
plt.title('Distribution of ratings')
plt.xticks(index + bar_width, ('1.0', '2.0', '3.0', '4.0', '5.0'))
plt.legend()
```

```
plt.tight_layout()
plt.show()
```



Statistics of ratings per user:

```
df.groupBy("UserID").count().select('count').describe().show()
```



summary	count
count	943
mean	106.04453870625663
stddev	100.93174276633498
min	20
max	737

Individual counts of ratings per user:

```
df.stat.crosstab("UserID", "Rating").show()
```

```
+-------------+---+---+---+---+---+
|UserID_Rating|1.0|2.0|3.0|4.0|5.0|
+-------------+---+---+---+---+---+
|          645|  2|  2| 29| 55| 34|
|          892|  2| 13| 40| 99| 72|
|           69|  2|  3| 21| 16| 23|
|          809|  2|  2|  6|  5|  5|
|          629|  1|  8| 24| 35| 53|
|          365|  5|  9| 12| 23|  9|
|          138|  0|  1|  3| 28| 19|
|          760|  4|  7| 11| 13|  6|
|          101|  3| 19| 28| 16|  1|
|          479| 24| 14| 48| 85| 31|
|          347| 20| 25| 37| 55| 62|
|          846| 10| 46| 89|154|106|
|          909|  0|  0|  5|  7| 14|
|          333|  1|  1|  5| 13|  6|
|          628|  0|  1|  1|  3| 22|
|          249|  1|  5| 31| 63| 61|
|          893|  1|  5| 28| 18|  7|
|          518|  4|  3| 28| 18| 20|
|          468|  0| | 9| 31| 55| 48|
|          234| 14|103|205|126| 32|
+-------------+---+---+---+---+---+
only showing top 20 rows
```

Average rating given by each user:

```
df.groupBy('UserID').agg({'Rating': 'mean'}).take(5)

[Row(UserID=148, avg(Rating)=4.0), Row(UserID=463,
avg(Rating)=2.8646616541353382), Row(UserID=471,
avg(Rating)=3.3870967741935485), Row(UserID=496,
avg(Rating)=3.0310077519379846), Row(UserID=833,
avg(Rating)=3.056179775280899)]
```

Average rating per movie:

```
df.groupBy('ItemId ').agg({'Rating': 'mean'}).take(5)

[Row(ItemId =496, avg(Rating)=4.121212121212121), Row(ItemId =471,
avg(Rating)=3.6108597285067874), Row(ItemId =463,
avg(Rating)=3.859154929577465), Row(ItemId =148, avg(Rating)=3.203125),
Row(ItemId =1342, avg(Rating)=2.5)]
```

Building the basic recommendation engine

Divide the original data into training and test datasets randomly as follows, using the `randomSplit()` method:

(training, test) = ratings.randomSplit([0.8, 0.2])

Counting the number of instances in the training dataset:

```
training.count()
80154
```

Counting the number of instances in the test set:

```
test.count()
19846
```

Let's now build a recommendation engine model using the ALS algorithm available in the MLlib library of Spark.

For this, we use the following methods and parameters:

1. Load the ALS module into the Spark environment.
2. Call the `ALS.train()` method to train the model.
3. Pass the required parameters, such as rank, number of iterations (maxIter), and training data to the `ALS.train()` method.

Let's understand the parameters now:

- **Rank**: This parameter is the number of latent factors of users and items to be used in the model. The default is 10.
- **maxIter**: This is the number of iterations the model has to run. The default is 10.

Build the recommendation model using Alternating Least Squares:

Setting rank and maxIter parameters:

```
rank = 10
numIterations = 10
```

Calling `train()` method with training data, rank, maxIter params, `model = ALS.train`(training, rank, numIterations):

```
16/10/04 11:01:34 WARN BLAS: Failed to load implementation from:
com.github.fommil.netlib.NativeSystemBLAS
16/10/04 11:01:34 WARN BLAS: Failed to load implementation from:
com.github.fommil.netlib.NativeRefBLAS
16/10/04 11:01:34 WARN LAPACK: Failed to load implementation from:
com.github.fommil.netlib.NativeSystemLAPACK
16/10/04 11:01:34 WARN LAPACK: Failed to load implementation from:
com.github.fommil.netlib.NativeRefLAPACK
16/10/04 11:01:37 WARN Executor: 1 block locks were not released by TID =
122:
[rdd_221_0]
16/10/04 11:01:37 WARN Executor: 1 block locks were not released by TID =
123:
[rdd_222_0]
16/10/04 11:01:37 WARN Executor: 1 block locks were not released by TID =
124:
[rdd_221_0]
16/10/04 11:01:37 WARN Executor: 1 block locks were not released by TID =
125:
[rdd_222_0]
```

Checking the model as below, we observe that `Matrixfactorizationmodel` object is created:

```
model
```

Making predictions

Now that we have created the model, let's predict the values of ratings on the test set we created earlier.

The ALS module has provided many methods discussed in the following sections for making predictions, recommending users, and recommending items to users, user features, item features, and so on. Let's run the methods one by one.

Before we proceed to predictions, we shall first create test data in a way that is acceptable to the prediction methods, as follows:

The following code extracts each row in the test data and extracts `userID, ItemID` and puts it in testdata `PipelinedRDD` object:

```
testdata = test.map(lambda p: (p[0], p[1]))

type(testdata)
<class 'pyspark.rdd.PipelinedRDD'>
```

The following code shows the original test data sample:

```
test.take(5)

[Rating(user=119, product=392, rating=4.0), Rating(user=38, product=95,
rating=5.0), Rating(user=63, product=277, rating=4.0), Rating(user=160,
product=234, rating=5.0), Rating(user=225, product=193, rating=4.0)]
```

The following code displays the formatted data required for making predictions:

```
testdata.take(5)

[(119, 392), (38, 95), (63, 277), (160, 234), (225, 193)]
```

The prediction methods are as follows:

- `predict()`: The predict method will the predict rating for a given user and item and is given as follows:

This method is used when we want to make predictions for a combination of user and item:

```
pred_ind = model.predict(119, 392)
```

We can observe below that the prediction value for a user `119` and movie `392` is `4.3926091845289275`: just see above the original value for the same combination in test data:

```
pred_ind

4.3926091845289275
```

- `predictall()`: This method is used when we want to predict values for all the test data in one go, given as follows:

```
predictions = model.predictAll(testdata).map(lambda r: ((r[0], r[1]),
r[2]))
```

Use the following code to check the data type:

```
type(predictions)
<class 'pyspark.rdd.PipelinedRDD'>
```

Use the following code displays the first five predictions:

```
 predictions.take(5)

[((268, 68), 3.197299431949281), ((200, 68), 3.6296857016488357), ((916,
68), 3.070451877410571), ((648, 68), 2.165520614428771), ((640, 68),
3.821666263132798)]
```

User-based collaborative filtering

Now let's recommend items (movies) to users. The ALS recommendation module contains the `recommendProductsForUsers()` method to generate the top-N item recommendations for users.

The `recommendProductsForUsers()` method takes integers as the input parameter, which indicates the top-N recommendations; for example, to generate the top 10 recommendations to the users, we pass 10 as value to the `recommendProductsForUsers()` method, as follows:

```
recommedItemsToUsers = model.recommendProductsForUsers(10)
```

Use the following code shows that recommendations are generated for all the 943 users:

```
recommedItemsToUsers.count()
943
```

Let us see the recommendations for the first two users: 96 and 784:

```
recommedItemsToUsers.take(2)

[
(96, (Rating(user=96, product=1159, rating=11.251653489172302),
Rating(user=96, product=962, rating=11.1500279633824), Rating(user=96,
product=534, rating=10.527262244626867), Rating(user=96, product=916,
rating=10.066351313580977), Rating(user=96, product=390,
rating=9.976996795233937), Rating(user=96, product=901,
rating=9.564128162876036), Rating(user=96, product=1311,
rating=9.13860044421153), Rating(user=96, product=1059,
rating=9.081563794413025), Rating(user=96, product=1178,
rating=9.028685203289745), Rating(user=96, product=968,
rating=8.844312806737918)
)),
 (784, (Rating(user=784, product=904, rating=5.975314993539809),
Rating(user=784, product=1195, rating=5.888552423210881), Rating(user=784,
product=1169, rating=5.649927493462845), Rating(user=784, product=1446,
rating=5.476279163198376), Rating(user=784, product=1019,
rating=5.303140289874016), Rating(user=784, product=1242,
rating=5.267858336331315), Rating(user=784, product=1086,
rating=5.264190584020031), Rating(user=784, product=1311,
rating=5.248377920702441), Rating(user=784, product=816,
rating=5.173286729120303), Rating(user=784, product=1024,
rating=5.1253425029498985)
))
]
```

Model evaluation

Now let's evaluate the model accuracy. For this, we choose the Root Mean Squared Error method to calculate the model accuracy. We can do it either manually, as shown next, or call a defined function available in the Spark MLlib:
Create a `ratesAndPreds` object by joining the original ratings and predictions:

```
ratesAndPreds = ratings.map(lambda r: ((r[0], r[1]),
r[2])).join(predictions)
```

The following code will calculate the mean squared error:

```
MSE = ratesAndPreds.map(lambda r: (r[1][0] - r[1][1])**2).mean()

[Stage 860:>                                                  (0 + 4) / 6]

Mean Squared Error = 1.1925845065690288
```

```
from math import sqrt

rmse = sqrt(MSE)
rmse
1.092055175606539
```

Model selection and hyperparameter tuning

The most important step in any machine learning task is to use model evaluation or model selection to find the optimal parameters that the fit the data. Spark provides infrastructure to tune and model evaluation, for individual algorithms or for the entire model building pipeline. Users may tune the entire pipeline model or tune individual components of the pipeline. MLlib provides model selection tools such as `CrossValidator` class and `TrainValidationSplit` class.

The above mentioned classes require the following items:

- **Estimator** algorithm or Pipeline to tune
- **Set of ParamMaps**: parameters to choose from, sometimes called a *parameter grid* to search over
- **Evaluator**: metric to measure how well a fitted model does on held-out test data

At a high level, these model selection tools work as follows:

- They split the input data into separate training and test datasets
- For each (training and test) pair, they iterate through the set of `ParamMaps`
- For each `ParamMap`, they fit the Estimator using those parameters, get the fitted Model, and evaluate the Model's performance using the Evaluator
- They select the Model produced by the best-performing set of parameters

The MLlib supports various evaluation classes for performing evaluation tasks, such as the `RegressionEvaluator` class for regression based problems, the `BinaryClassificationEvaluator` class for binary classification problems, and the `MulticlassClassificationEvaluator` class for multiclass classification problems. For constructing a parameter grid, we can use the `paramGridBuilder` class.

Cross-Validation

The **Cross-Validation** approach is one of the most popular approaches in evaluating the datamining models and in choosing optimal parameters for building the best estimation model. MLlib offers two types of evaluation classes: the `CrossValidator` and `TrainValidationSplit` classes.

CrossValidator

The `CrossValidator` class takes the input dataset and splits it into multiple dataset folds, which can be used as training and test sets. Using these datasets, the `CrossValidator` class builds multiple models and finds optimal parameters and stores in `ParamMap`. After identifying the best `ParamMap`, the `CrossValidator` class finally computes the best model using the entire dataset. For example, let's say we choose the five-fold cross-validation; the `CrossValidator` class splits the original dataset into five sub-datasets with each sub-dataset containing training and test sets. The `CrossValidator` class chooses each fold set at a time and estimates the model parameters. Finally, `CrossValidator` computes the average of the evaluation metric to store the best parameters in `ParamMap`.

Train-Validation Split

Spark MLlib provides another class for estimating the optimal parameters using `TrainValidationSplit`. Unlike `CrossValidator`, this class estimates the optimal parameters on a single dataset. For example, the `TrainValidatorSplit` class divides the input data into trainset and test sets of size 3/4 and 1/4, and optimal parameters are chosen using these sets.

Now, let's understand the recommendation engine model we built earlier.

The tuning model is present in the MLlib of Spark 2.0 and makes use of DataFrame API features. So in order to accommodate this, our first step is to convert the original dataset ratings to DataFrame.

For conversion, we use the `sqlContext` object and the `createDataFrame()` method to convert the ratings RDD object to a `DataFrame` object, as follows:

```
type(ratings)
<class 'pyspark.rdd.PipelinedRDD'>
```

SQL Context object is created when starting the spark session using pyspark:

```
sqlContext
<pyspark.sql.context.SQLContext object at 0x7f24c94f7d68>
```

Creating a DataFrame object from the ratings rdd object as follows:

```
df = sqlContext.createDataFrame(ratings)

type(df)

<class 'pyspark.sql.dataframe.DataFrame'>
```

Display first 20 records of `dataframe` object:

```
df.show()
```

The following screenshot shows the results of the previous query:

```
+----+-------+------+
|user|product|rating|
+----+-------+------+

| 196|    242|   3.0|
| 186|    302|   3.0|
|  22|    377|   1.0|
| 244|     51|   2.0|
| 166|    346|   1.0|
| 298|    474|   4.0|
| 115|    265|   2.0|
| 253|    465|   5.0|
| 305|    451|   3.0|
|   6|     86|   3.0|
|  62|    257|   2.0|
| 286|   1014|   5.0|
| 200|    222|   5.0|
| 210|     40|   3.0|
| 224|     29|   3.0|
| 303|    785|   3.0|
| 122|    387|   5.0|
| 194|    274|   2.0|
| 291|   1042|   4.0|
| 234|   1184|   2.0|
+----+-------+------+
only showing top 20 rows
```

Creating random samples of training set and test set using `randomSplit()` method:

```
(training, test) = df.randomSplit([0.8, 0.2])
```

Load modules required for running parameter `tuningmodel`:

```
from pyspark.ml.recommendation import ALS
```

Call the ALS method available in MLlib to building the recommendation engine. The following method `ALS()` takes only column values of training data, such as UserID, ItemId, and Rating. The other parameters, such as rank, number of iterations, learning parameters, and so on will be passed as the `ParamGridBuilder` object to the cross-validation method.

As mentioned earlier, the model tuning pipeline require Estimators, a set of ParamMaps, and Evaluators. Let's create each one of them, as follows:

Estimator objects as stated earlier, estimators take algorithm or pipeline objects as input. Let us build one pipeline object as follows:

Calling ALS algorithm:

```
als = ALS(userCol="user", itemCol="product", ratingCol="rating")
als

ALS_45108d6e011beae88f4c
```

Checking the type of `als` object:

```
type(als)
<class 'pyspark.ml.recommendation.ALS'>
```

Let us see how the default parameters are set for the ALS model:

```
als.explainParams()
"alpha: alpha for implicit preference (default: 1.0)\ncheckpointInterval:
set checkpoint interval (>= 1) or disable checkpoint (-1). E.g. 10 means
that the cache will get checkpointed every 10 iterations. (default:
10)\nfinalStorageLevel: StorageLevel for ALS model factors. (default:
MEMORY_AND_DISK)\nimplicitPrefs: whether to use implicit preference
(default: False)\nintermediateStorageLevel: StorageLevel for intermediate
datasets. Cannot be 'NONE'. (default: MEMORY_AND_DISK)\nitemCol: column
name for item ids. Ids must be within the integer value range. (default:
item, current: ItemId )\nmaxIter: max number of iterations (>= 0).
(default: 10)\nnonnegative: whether to use nonnegative constraint for least
squares (default: False)\nnumItemBlocks: number of item blocks (default:
10)\nnumUserBlocks: number of user blocks (default: 10)\npredictionCol:
```

```
prediction column name. (default: prediction)\nrank: rank of the
factorization (default: 10)\nratingCol: column name for ratings (default:
rating, current: Rating)\nregParam: regularization parameter (>= 0).
(default: 0.1)\nseed: random seed. (default:
-1517157561977538513)\nuserCol: column name for user ids. Ids must be
within the integer value range. (default: user, current: UserID)"
```

From the preceding result, we observe that the model is set to its default values for rank as 10, maxIter as 10, and blocksize as 10:

Create pipeline object and setting the created als model as a stage in the pipeline:

```
from pyspark.ml import Pipeline

pipeline = Pipeline(stages=[als])
 type(pipeline)
<class 'pyspark.ml.pipeline.Pipeline'>
```

Setting the ParamMaps/parameters

Let's observe the `ALS()` method closely and logically and infer the parameters that can be used for parameter tuning:

rank: We know that rank is the number of latent features for users and items, which, by default, is 10, but if we do not have the optimal number of latent features for a given dataset, this parameter can be taken up for tuning the model by giving a range of values between 8 and 12 - the choice is left to the users. Due to the computational cost, we restrict the values to 8 -12, but readers are free to try other values.

MaxIter: MaxIter is the number of times the model is made to run; it's set to a default 10. We can select this parameter also for tuning as we do not know the optimal iterations at which the model performs well; we select between 10 and 15.

reqParams: regParams is the learning parameter set between 1 and 10.

Loading `CrossValidation` and `ParamGridBuilder` modules to create range of parameters:

```
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

paramMapExplicit = ParamGridBuilder() \
                   .addGrid(als.rank, [8, 12]) \
                   .addGrid(als.maxIter, [10, 15]) \
                   .addGrid(als.regParam, [1.0, 10.0]) \
                   .build()
```

Setting the evaluator object

As stated earlier, the evaluator object sets the evaluation metric to evaluate the model during multiple runs in the cross validation method:

Loading the `RegressionEvaluator` model:

```
from pyspark.ml.evaluation import RegressionEvaluator
```

Calling `RegressionEvaluator()` method with evaluation metric set to `rmse` and `evaluation` column set to Rating:

```
evaluatorR = RegressionEvaluator(metricName="rmse", labelCol="rating")
```

Now that we have prepared all the required objects for running the cross-validation method, that is, `Estimator`, `paramMaps`, and `Evaluator`, let's run the model.

The cross-validation method gives us the best optimal model out of all the executed models:

```
cvExplicit = CrossValidator(estimator=als, estimatorParamMaps=paramMap,
evaluator=evaluatorR,numFolds=5)
```

Running the model using `fit()` method:

```
cvModel = cvExplicit.fit(training)

[Stage 897:============================>                          (5 + 4)
/ 10]
[Stage 938:==================================================>     (9 + 1)
/ 10]
[Stage 1004:>(0 + 4) / 10][Stage 1005:> (0 + 0) / 2][Stage 1007:>(0 + 0) /
10]
[Stage 1008:>                                                       (3 + 4) /
200]


preds = cvModel.bestModel.transform(test)
evaluator = RegressionEvaluator(metricName="rmse",
labelCol="rating",predictionCol="prediction")
rmse = evaluator.evaluate(pred)
print("Root-mean-square error = " + str(rmse))
 rmse

0.924617823674082
```

Summary

In this chapter, we learned about model-based collaborative filtering using matrix factorization methods using ALS. We used the Python API to access the Spark framework and ran the ALS collaborative filtering. In the beginning of the chapter, we refreshed our knowledge of Spark with all the basics that are required to run the recommendation engines, such as what Spark is, the Spark ecosystem, components of Spark, SparkSession, DataFrames, RDD, and so on. As explained then, we explored the MovieLens data, built a basic recommendation engine, evaluated the model, and used parameter tuning to improve the model. In the next chapter, we shall learn about building recommendations using Graph database – Neo4j.

8
Building Real-Time Recommendations with Neo4j

The world we live in is a big, interconnected place. Anything and everything that exists in this world is connected together in some way. Relationships and connections exist among the entities that inhabit this world.

The human brain tries to store or extract information in the form of networks and relations. Perhaps this is a more optimal way of representing data, so that storing and retrieval of information is fast and efficient. What if we have a system that works in a similar way. We can use graphs; they are a systematic and methodical approach to representing data.

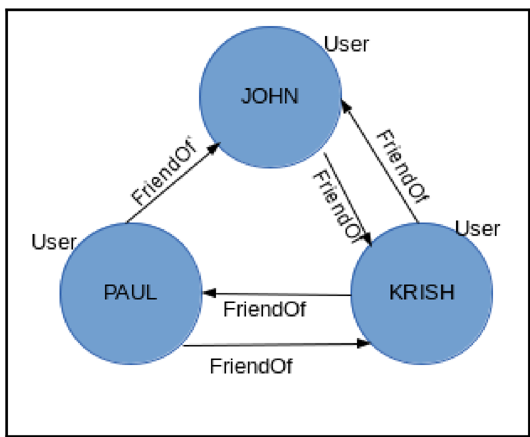
Before we move ahead with this chapter it is essential to understand the background and necessity of graphs.

Credit for the concept behind the graph theory is given to the 18th Century mathematician, Leonhard Euler, who solved the age-old problem known as The Bridges of Konigsberg, which is essentially a pathfinding problem. Although we won't look further at this problem, I suggest that readers attempt to understand how Euler has come up with a new paradigm approach in understanding and solving the problem.

Graphs are found everywhere in today's world and are one of the most efficient and natural ways of working with data.

Graph can represent how two or more real world entities, represented as nodes, are connected to each other. We also learn how each of them are related to the other, and how this helps to communicate information in a fast, efficient , visual way. Since graph systems allow us to express anything in an expressive, structured way, we can apply these systems across domains such as social networks, medicine, science and technology and many more.

To better understand graph representation, we can take an example of networking on Facebook. Let us assume there are three friends **John**, **Paul** and **Krish**, connected on Facebook. JOHN-KRISH are mutual friends, PAUL-KRISH are mutual friends and PAUL is **`FriendOf`** of JOHN. How do we represent this information? Take a look at the following diagram:



Don't we feel that the above representation is one of the most efficient and natural ways of representing data and its relations? In the previous diagram, JOHN-KRISH-PAUL are *Nodes* representing User entities, and `FriendOf` arrows are edges which represents the *relationships* between the Nodes. We can also store the demographic details of User Nodes – such as age and details of relationship (such as FriendSince) – as *Properties* in the Graphs. By applying Graph Theory concepts we can find similar Users in a Network or suggest new Friends to Users within the Friends Network. We shall learn about more on this in later sections.

Discerning different graph databases

Graph databases have revolutionized the way people discover new products and share information with one another. In the human mind, we remember people, things, places, and so on, as graphs, relations, and networks. When we try to fetch information from these networks we go directly to the required connection or graph and fetch information accurately. In a similar fashion, graph databases allow us to store the users and product information in graphs as nodes and edges (relations). Searching a graph database is fast.

A graph database is a type of NoSQL database that uses graph theory to store, map and query relationships. Graph databases excel at managing highly connected data and managing complex queries. They are mainly used for analyzing the interconnections between data. Here, the priority is given to relations, so that we don't have to bother with the foreign keys, as in the case of SQL.

Graph databases mainly consist of nodes and edges, wherein nodes represent the entities and edges the relations between them. The edges are directed lines or arrows that connect the nodes. In the preceding diagram, the circles are the nodes that represent the entities, and the lines connecting the nodes are called the edges – these represent relationships. The orientation of arrows follows the flow of information. By presenting all nodes and links of the graph, it helps users get a global view of the structure.

Neo4j, FlockDB, AllegroGraph, GraphDB, and InfiniteGraph are some of the graph databases available. Let us have a look at Neo4j, one of the most popular among them, made by Neo Technology.

Neo4j is so popular because of its strength, swiftness and scalability. It is mainly implemented in Scala and Java. It is available in both the community and enterprise editions. The enterprise edition has the same features as the community one, with additional features like enterprise-grade availability, management and scale-up and scale-out capabilities. In the case of RDBMS, the performance degrades exponentially as the number of relations increases, whereas in Neo4j it is linear. The following image shows the various graph databases:

Labeled property graph

In the introduction section, we have seen an example of a Social Network representation of three friends. This graph representation of data which contains directed connections between entities/nodes, relationships between nodes, and properties associated with nodes and relationships is called a **labeled property graph data model**.

A labeled property graph data model has the following properties:

- Graph contains nodes and relationships
- Nodes may contain properties (key-value pairs)
- Nodes may be labeled with one or more labels
- Relationships are named and directed, and always have a start and end node
- Relationships may also contain properties

Listed concepts are explained in the following section.

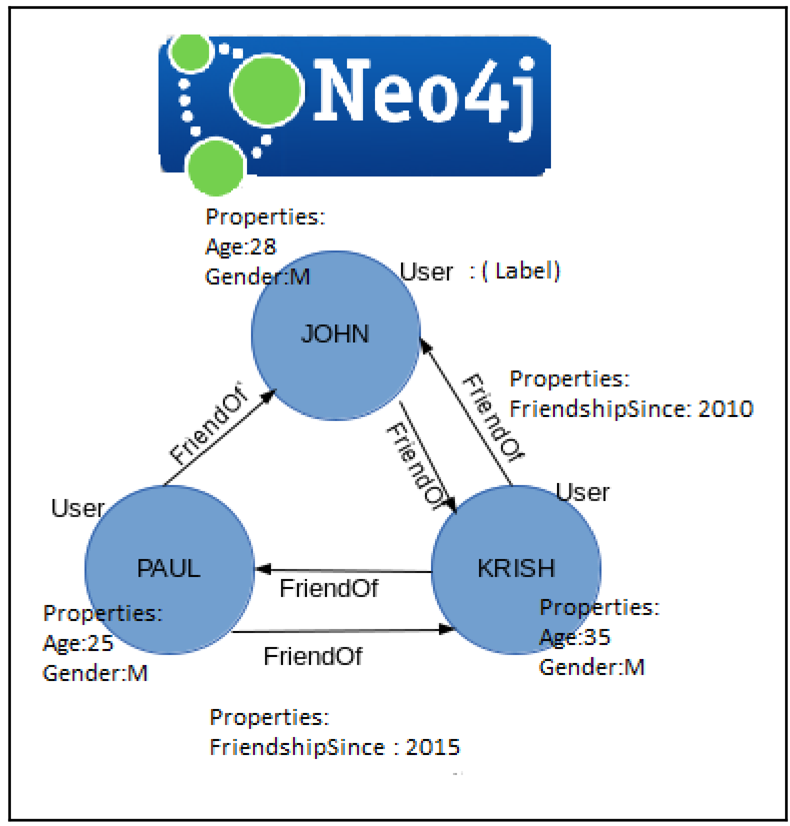
Understanding GraphDB core concepts

The following list enumerates all the elements of a graph:

- **Nodes**: Nodes are the fundamental unit of a graph. Nodes are the vertices in the graph. It mostly refers to the main object that is being referred. Nodes can contains labels and properties. From the story, we can pull three different objects and make three nodes. Two of those are for friends and the other one is for the movie.
- **Labels**: Labels are the way to differentiate between the same kinds of objects. Labels are generally given to each node with similar characteristics. Nodes can have more than one label. In the example story, we gave labels of **PERSON** and **MOVIE**. This optimized the graph traversal and also helped in logically querying the model efficiently.
- **Relationships**: Relationships are the edge between two nodes. They can be unidirectional and bidirectional. They can also contain the property for which the relationship is being created. Relationships are named and directed, and always have a start and end node. For example, there is a relationship of *Friend Of* between two friends. This shows the connection between different nodes. There is also a relation of *Has Watched* between each of the friends with the movie node.

- **Properties**: Properties are key value pairs. Properties can be used for both nodes and relationships. They are used to save the details about a particular node or relationship. In the example, the Person node has the properties of name and age. These properties are used to distinguish different nodes. Relation Has Watched also has the properties of date and rating.

In the following diagram, **JOHN**, **KRISH**, and **PAUL** are nodes that are mapped as User labels. Also, observe the edges that show relations. Both nodes and relations can have properties to further describe them:



Neo4j

Neo4j is an open-source Graph Database implemented in Java and Scala. Neo4j implements labeled property graph model efficiently. Like any other database, Neo4j provides ACID transactions, runtime fail-over and cluster support, allowing it for developing production ready applications. This graph database architecture is designed for efficient data storage and faster traversal between Nodes and relations. To work with the data for storing, retrievals and traversal, we use **CYPHER query language** which is Neo4j's query language based on patterns.

Cypher query language

Cypher is the query language for Neo4j that follows SQL-like queries. It is a declarative query language that focuses on what to retrieve from the graph, rather than how to retrieve it. We know that Neo4j property graphs consist of nodes and relationships; though these nodes and relationships are the basic building blocks, the real power of a graph database is to identify the underlying patterns that exist between nodes and relationships. This pattern extraction capability of graph databases, such as Neo4j, helps us to perform complex operations very quickly and efficiently.

Neo4j's Cypher query language is based on patterns. These patterns are used for matching underlying graph structures so that we may make use of patterns for further processing, such as building recommendation engines, in our case.

An example of extracting patterns using a Cypher query is shown later. The following Cypher query matches all *friendof* patterns between pairs of users and returns them as a graph:

Cypher Query

```
MATCH(u:User) -[f:friendof]-> (m:User)  RETURN f
```

Above Cypher query pulls up all the friendship relations between pairs of users

Cypher query basics

Before we go into building recommendations using Neo4j, let us look into the basics of Cypher query. As we mentioned earlier, Cypher is the query language for Neo4j that follows SQL-like queries. Being a declarative language, Cypher focuses on what to retrieve from the graph rather than how to retrieve it. The key principles and capabilities of Cypher are as follows:

- Cypher matches key patterns between nodes and relationships in the graph to extract information from the graph.
- Cypher has many capabilities similar to SQL such as create, delete, and update. These operations are applied to nodes and relationships to fetch information.
- Indexing and constraints similar to SQL are also present.

Node syntax

Cypher uses pairs of parentheses `()` or pairs of parenthesis with text inside to represent nodes. Furthermore, we can assign labels, and properties of nodes are given as key-value pairs.

Look at the following example to understand the concept better. In the following queries, node is represented using `()` or `(user)`, label is represented with `u`, `(u:user)` and properties of the node are assigned with key-value pairs as `(u:user{name:'Toby'})` :

```
()
(user)
(u:user)
(u:user{name:'Toby'})
```

Relationship syntax

Cypher uses `-[]->` to represent relationships between two nodes. These relationships allow developers to represent complex relations between nodes, making them easier to read or understand.

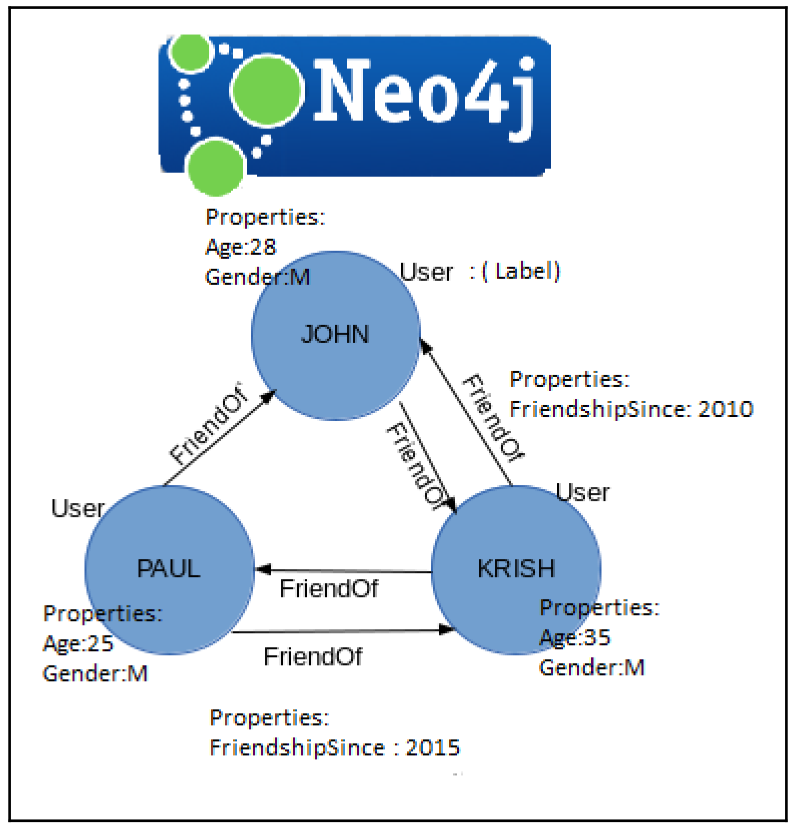
Let us look at the following example:

```
 -[]->
(user) -[f:friendof]->(user)
(user) -[f:friendof {since: 2016}]->(user)
```

In the preceding example, a `friendof` relationship is established between two user nodes and the relationship is having property `since:2016`.

Building your first graph

Now that we saw the node syntax and relationship syntax, let us practice what we have learned so far by creating a Facebook social network graph similar to the following diagram:



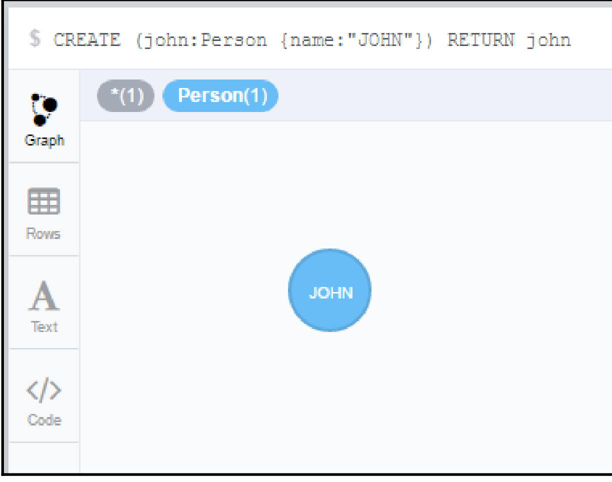
In order to create the above graph, we need following steps:

1. Create 3 nodes Person with labels JOHN, PAUL, KRISH
2. Create relationships between 3 Nodes
3. Set properties
4. Display results used with all the patterns

Creating nodes

We use the `CREATE` clause to create graph elements such as nodes and relations. The below example shows us how to create a single node Person labeled as john and having the property name:`JOHN`. When we run the below query in Neo4j browser, we get the graph as shown in the following screenshot:

```
CREATE (john:Person {name:"JOHN"})  RETURN  john
```



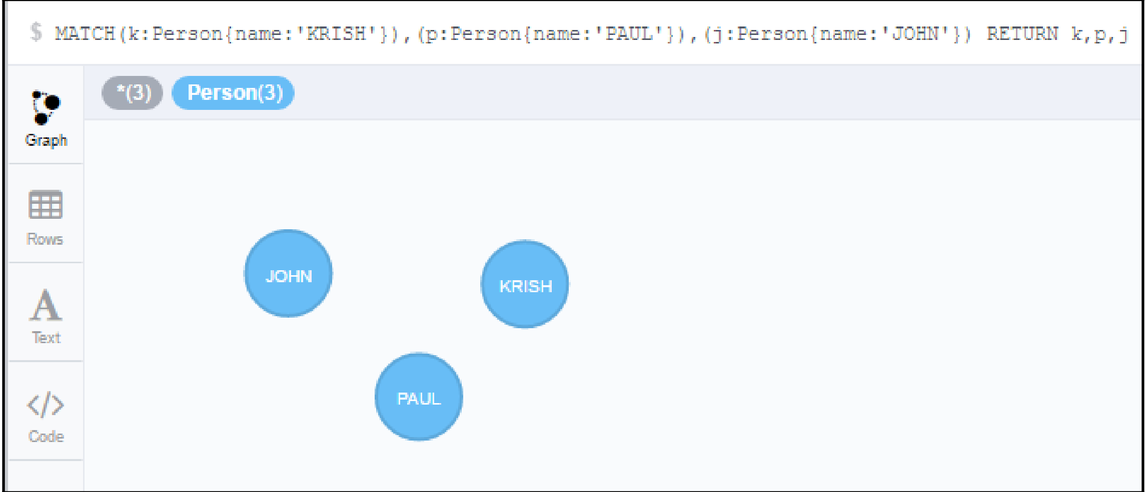
The RETURN clause helps to return the result set, namely Node – PERSON

Instead of just creating one node, we can create multiple nodes as follows:

```
CREATE (paul:Person {name:"PAUL"})
CREATE (krish:Person {name:"KRISH"})
```

Earlier code will create three nodes, and Person labelled `JOHN`, `PAUL`, `KRISH`. Let's see what we have created so far; to see the results we have to use `MATCH` clause. `MATCH` clause will check for the required patterns and return the retrieved patterns using `RETURN` clause. In the below query, `MATCH` will look for patterns such as Person nodes with labels names `k,p,j` and their corresponding labels:

```
MATCH(k:Person{name:'KRISH'}),(p:Person{name:'PAUL'}),(j:Person{name:'JOHN'
}) RETURN k,p,j
```



Creating relationships

With creation of nodes we are half done. Now, let's complete the remaining portion by creating relations.

Instructions for creating relationships are as follows:

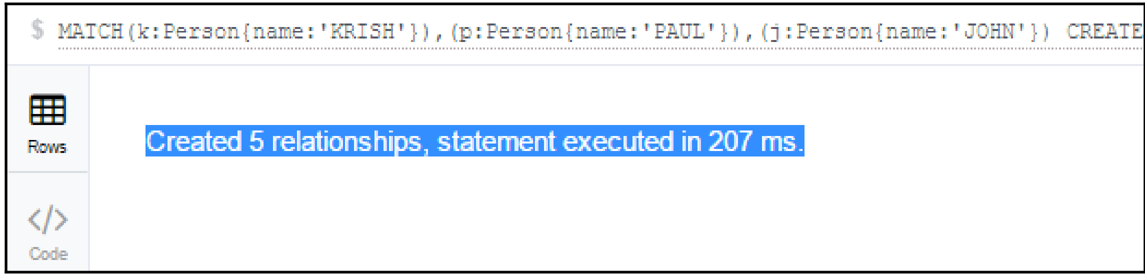
- Extract the nodes from the database using the `MATCH` clause
- Create the required relationships between the `Persons` using the `CREATE` clause

In the following query, we are extracting all the `Person` nodes and then creating relationships called `FRIENDOF` between the nodes:

```
MATCH(k:Person{name:'KRISH'}),(p:Person{name:'PAUL'}),(j:Person{name:'JOHN'
})
CREATE (k)-[:FRIENDOF]->(j)
CREATE (j)-[:FRIENDOF]->(k)
```

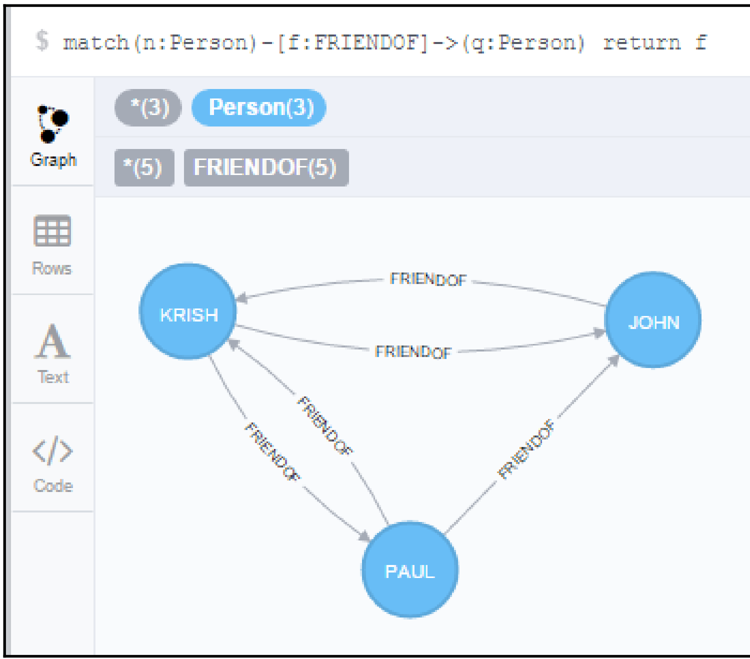
```
CREATE (p)-[:FRIENDOF]->(j)
CREATE (p)-[:FRIENDOF]->(k)
CREATE (k)-[:FRIENDOF]->(p)
```

The following screenshot shows the result displayed when we run the earlier query:



Now we have created all the required nodes and relationships. To see what we have achieved, run the following query, which displays Nodes and relationships between the nodes:

```
match(n:Person)-[f:FRIENDOF]->(q:Person) return f
```



Setting properties to relations

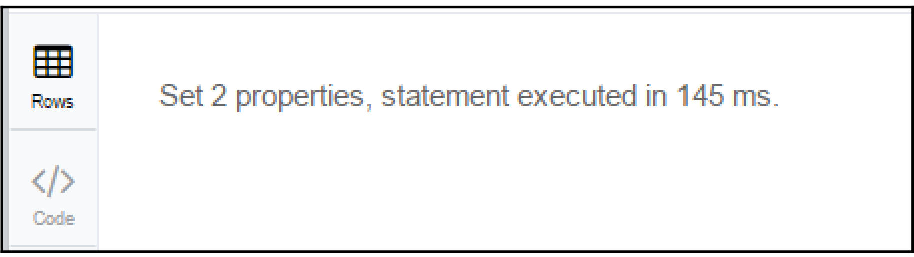
The final step is to set properties to node labels and relationships, and is explained as follows:

We use the `SET` clause to set the properties. For setting properties to relations we need to follow two steps:

1. Extract all the relations , `FRIENDOF`
2. Use the `SET` clause to set the properties to these relations

In the following example, we set the properties to the relation `FRIENDOF` between `KRISH` and `PAUL` with the property `friendsince` as follows:

```
MATCH (k:Person{name:'KRISH'})-[f1:FRIENDOF]-> (p:Person{name:'PAUL'}),
(k1:Person{name:'KRISH'})<-[f2:FRIENDOF]- (p1:Person{name:'PAUL'})
SET f1.friendsince = '2016', f2.friendsince = '2015'
```

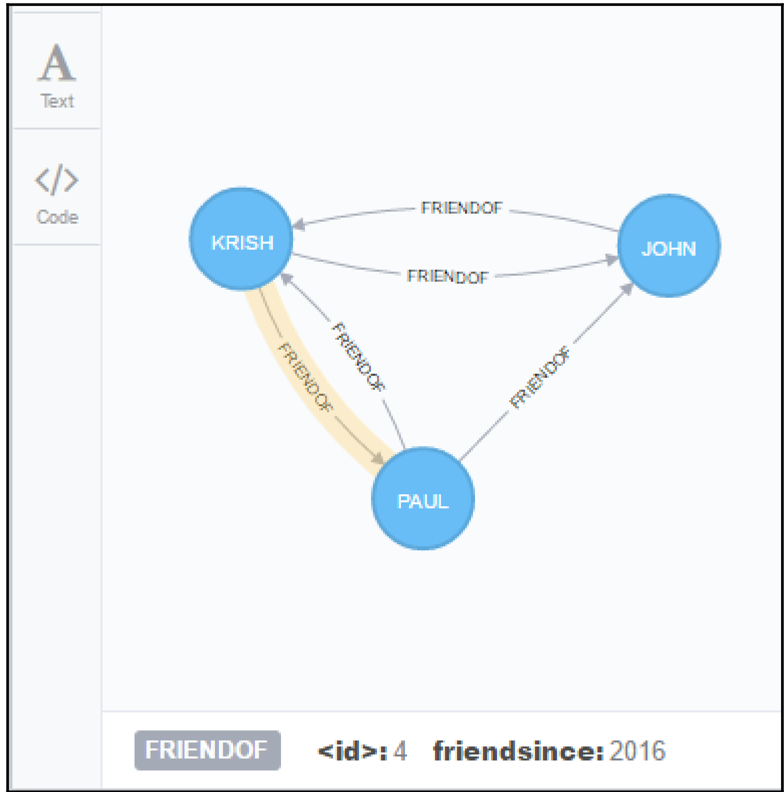


In the previous query , `()-[]->` pattern extracts relation `Krish` is `friendOfPaul` and `() <- [] -` pattern extracts relation `Paul` is `friendOf` of `Krish`.

Let's display the results so far as follows:

```
match(n:Person)-[f:FRIENDOF]->(q:Person) return f
```

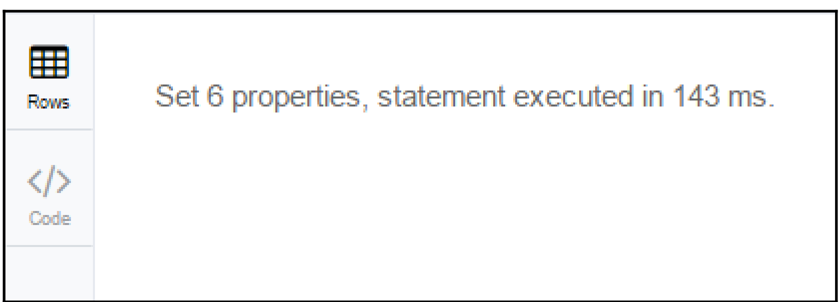
The following diagram shows the nodes, relationships and properties added in the previous query.



In the preceding diagram we can see that for `KRISH` and `PAUL` the property for the `FRIENDOF` relation has been set as `friendsince`.

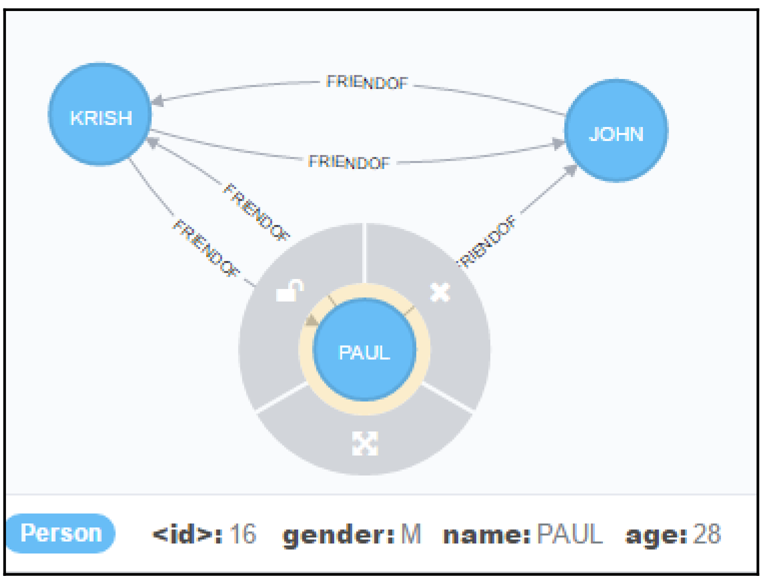
Similarly, we can set the properties to the nodes as follows:

```
MATCH(k:Person{name:'KRISH'}),(p:Person{name:'PAUL'}),(j:Person{name:'JOHN'
})
SET k.age = '26' ,p.age='28',
j.age='25',k.gender='M',p.gender='M',j.gender='M'
```



Let's verify the results here using the following query, which displays nodes, relationships, labels, properties to nodes, and relationships:

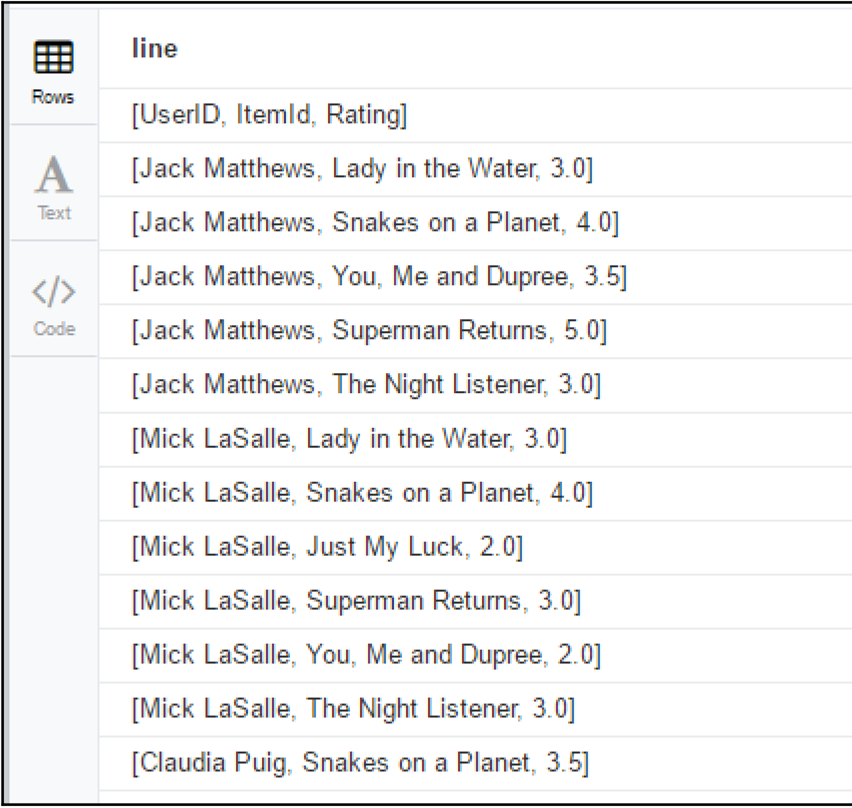
```
match(n:Person)-[f:FRIENDOF]->(q:Person) return f
```



Loading data from csv

In the previous section, we created nodes, relationships, and properties manually. Most of the time, we create nodes by loading data from csv files. To achieve this, we use the `LOAD CSV` command readily available in Neo4j, to load data into the Neo4j browser.

The following screenshot shows the dataset we will be using for this section which contains user-movie-rating data.



line
[UserID, ItemId, Rating]
[Jack Matthews, Lady in the Water, 3.0]
[Jack Matthews, Snakes on a Planet, 4.0]
[Jack Matthews, You, Me and Dupree, 3.5]
[Jack Matthews, Superman Returns, 5.0]
[Jack Matthews, The Night Listener, 3.0]
[Mick LaSalle, Lady in the Water, 3.0]
[Mick LaSalle, Snakes on a Planet, 4.0]
[Mick LaSalle, Just My Luck, 2.0]
[Mick LaSalle, Superman Returns, 3.0]
[Mick LaSalle, You, Me and Dupree, 2.0]
[Mick LaSalle, The Night Listener, 3.0]
[Claudia Puig, Snakes on a Planet, 3.5]

Query below to load csv data given below:

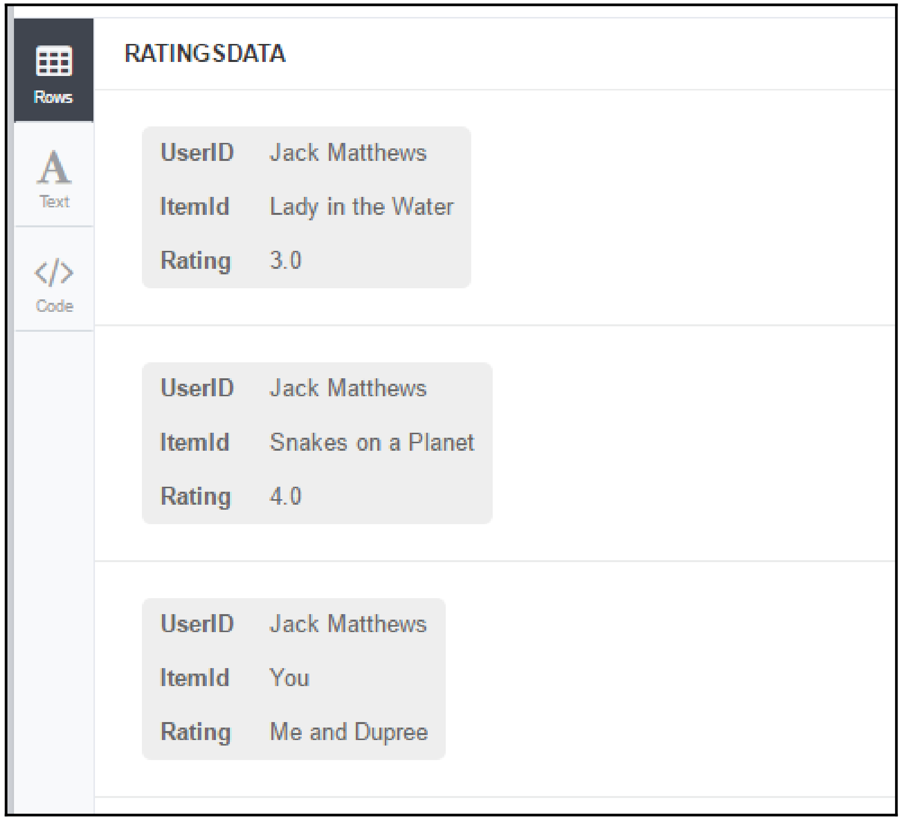
```
LOAD CSV WITH HEADERS FROM 'file:///C:/ Neo4J/test.csv' AS RATINGSDATA
RETURN RATINGSDATA
```

In the preceding query:

- The **HEADERS** keyword allows us to ask the query engine to consider the first row as header information
- The **WITH** keyword is similar to the return keyword; it separate portions of the query explicitly and allows us to define which values or variables we should carry forward to the next parts of the query
- The **AS** keyword is used to create an alias name to variables

When we run the above query, two things happen:

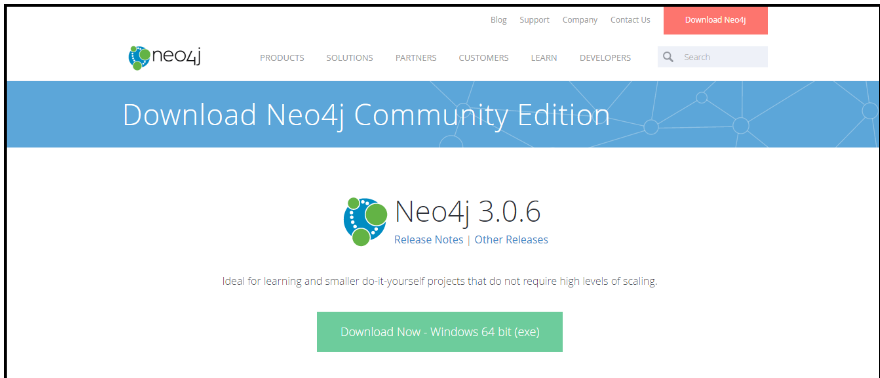
- **CSV** data will be loaded to the graph database
- The **RETURN** clause will display the loaded data, as shown in the following screenshot:



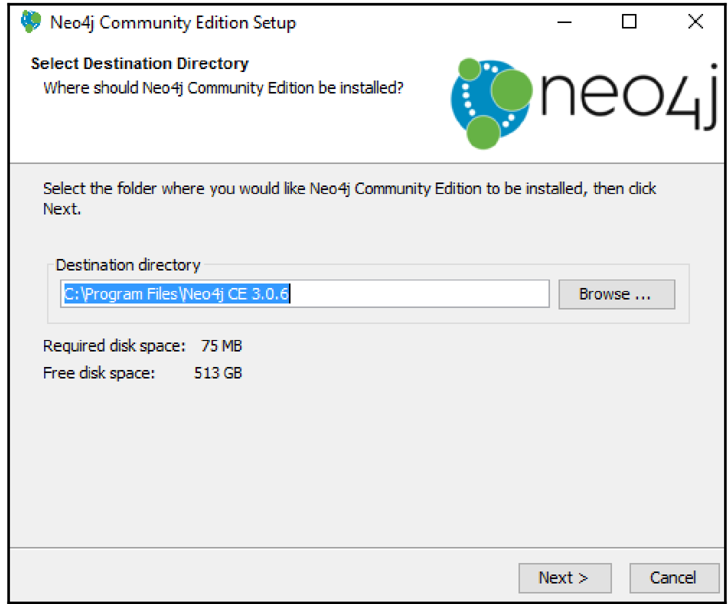
Neo4j Windows installation

In this section, we will see how to install Neo4j for Windows. We can download the Neo4j Windows installer from the following URL:

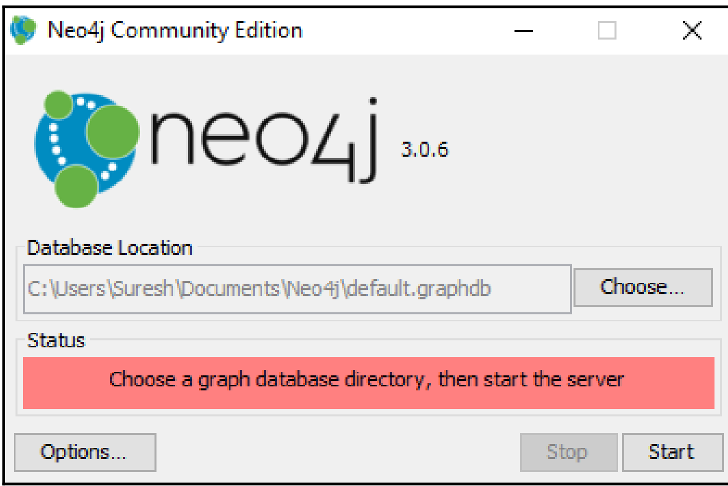
`https://neo4j.com/download/`



Once the installer is downloaded, click on the installer to get the following screen to proceed with installation:



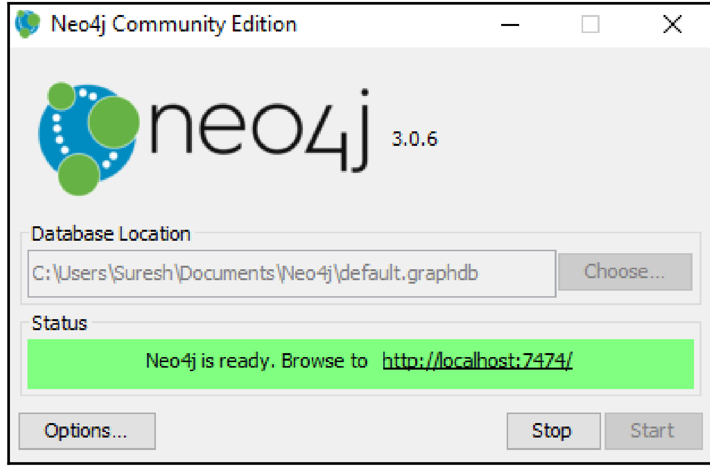
After successful installation, start the Neo4j Community Edition. For the first time you will see the following screen, asking you to choose a directory to store the graph database, and then click on **Start**:



In our case we have chosen the default directory where the `graphdb` database is created as follows:

```
C:\Users\Suresh\Documents\Neo4J\default.graphdb
```

After we click the start button, as shown in the preceding screenshot, start Neo4j will be started and will be displayed as below. We are now ready to start working on Neo4j.



Now that we have started Neo4j, we can access it from the browser by using:

```
http://localhost:7474
```

Installing Neo4j on the Linux platform

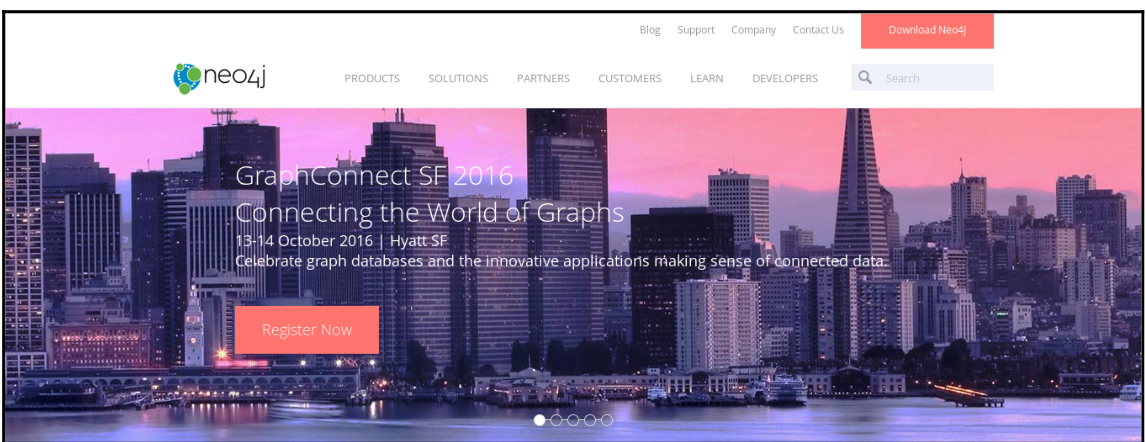
In this section we learn about downloading and installing Neo4j on the CentOS Linux platform.

Downloading Neo4j

We can download the latest version of the Neo4j 3 Linux source file from the Neo4j home page:

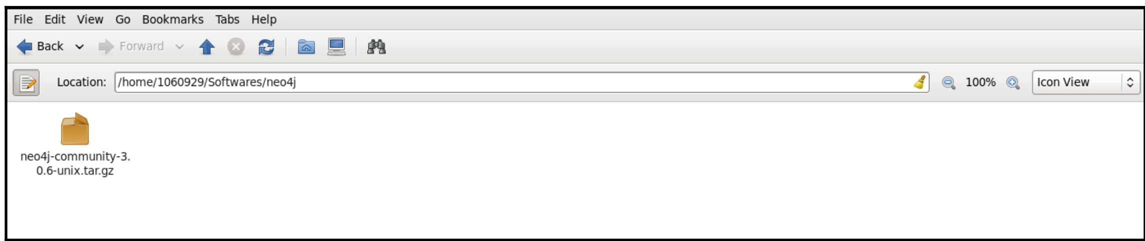
```
https://Neo4J.com/
```

Click on the **Download Neo4J** button on the page shown as follows:



Alternatively you can download it directly from the following URL: `http://info.Neo4J.com/download-thanks.html?edition=community&release=3.0.6&flavour=unix&_ga=1.171681440.1829638272.1475574249`

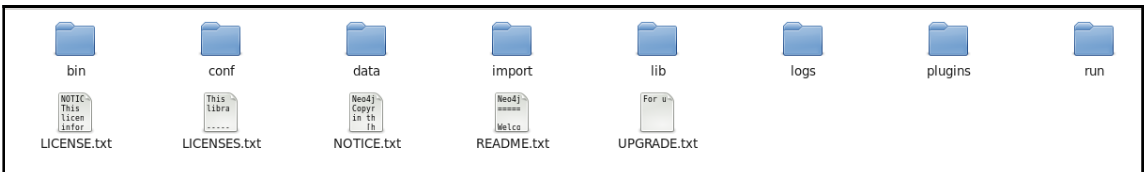
This will download a `tar` file – `Neo4J-community-3.0.6-unix.tar.gz` as shown in the following screenshot:



We can find the developer resources at `https://Neo4J.com/developer/get-started/`

Setting up Neo4j

Extract the `tar` file and you will get a folder called `Neo4J-community-3.0.6` containing the following files:



Starting Neo4j from the command line

Make sure you install Java 8 in your PC, as Neo4j 3.0 version requires Java 8. Check the Neo4j requirements before you install.

Once you have installed Java 8 then we can go ahead and run our Neo4j instance, but before that, let us set the `Neo4J` path in the `bashrc` file as follows:

```
gedit ~/.bashrc
export NEO4J_PATH=/home/1060929/Softwares/Neo4J/Neo4J-community-3.0.6
export PATH=$PATH:$NEO4J_PATH/bin
source ~/.bashrc
```

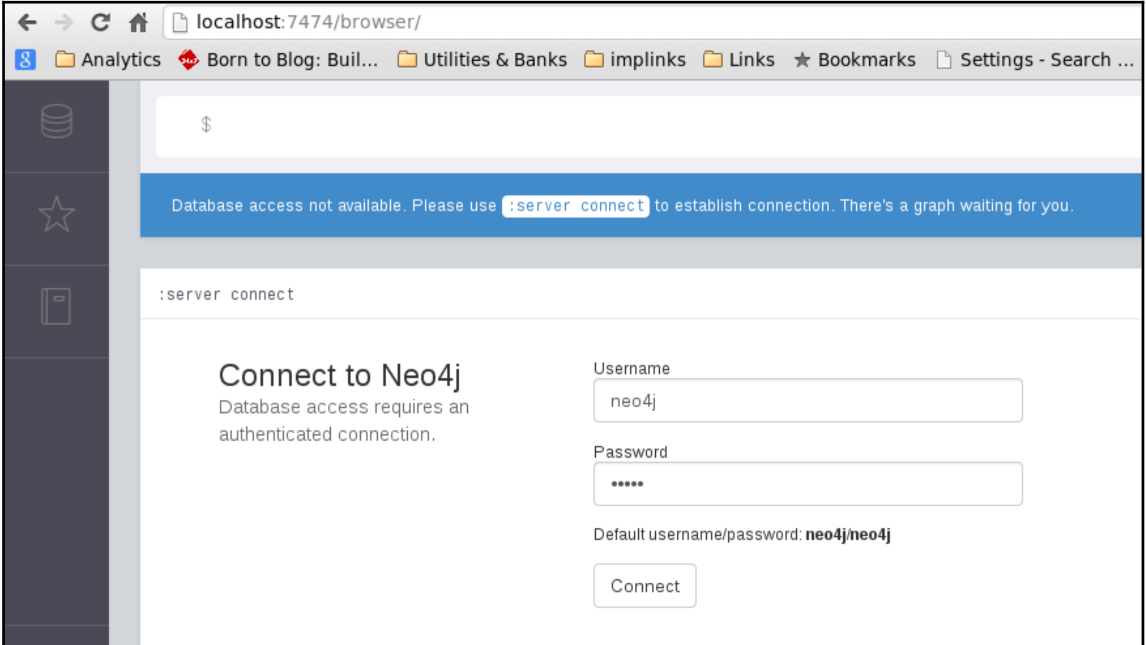
We Start the `Neo4j` in command line using the following command:

```
Neo4J start
```

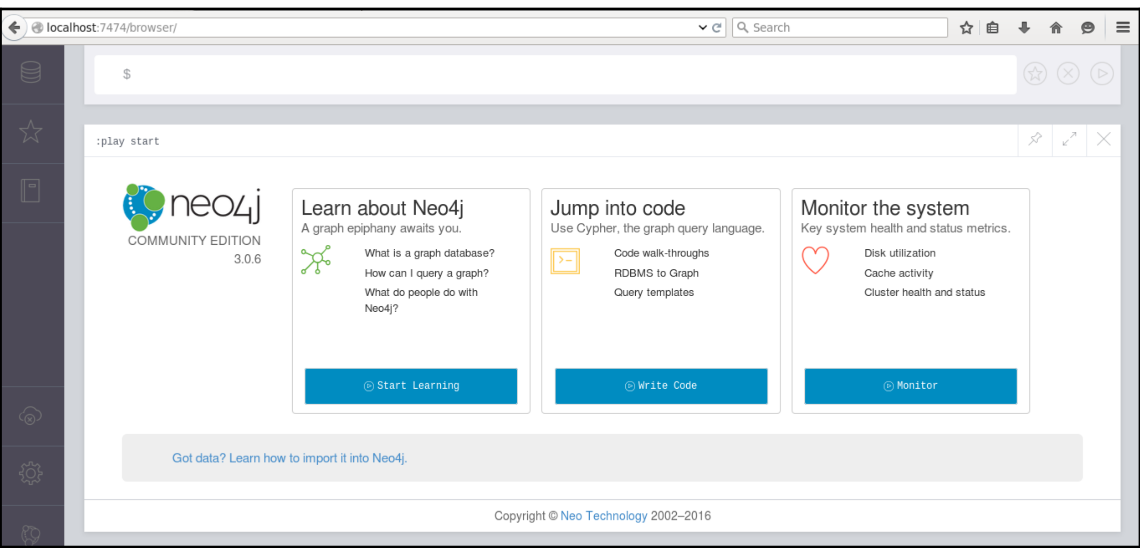
```
[1060929@01hw745020 home]$ neo4j start
Starting Neo4j.
WARNING: Max 1024 open files allowed, minimum of 40000 recommended. See the Neo4j manual.
Started neo4j (pid 1039). By default, it is available at http://localhost:7474/
There may be a short delay until the server is ready.
See /home/1060929/Softwares/neo4j/neo4j-community-3.0.6/logs/neo4j.log for current status.
[1060929@01hw745020 home]$ gedit ~/.bashrc
```

We can observe that the Neo4j has been started and we can access the graph `dbcapabilites` from the browser at `http://localhost:7474/`

For the first time, running Neo4j in the browser requires you to set the **Username** and **Password**:

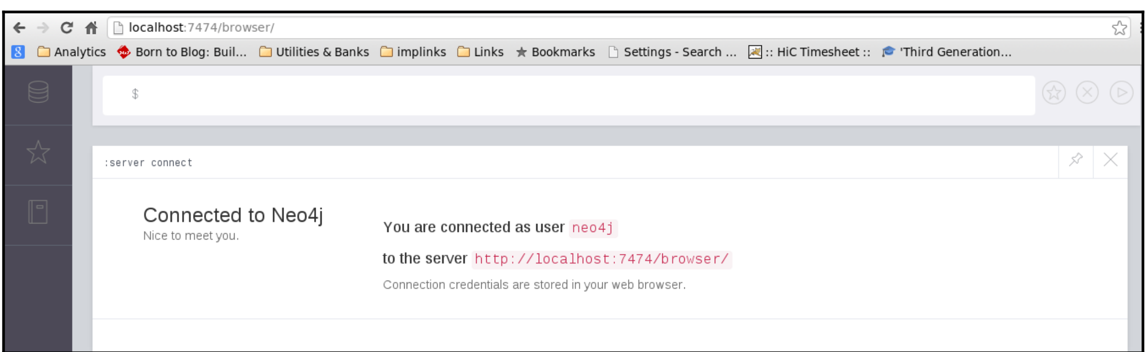


Once we have set the credentials it will redirect to the following page:



If you are using it for the first time, spend some time on the browser to get acquainted with its features and explore the different options available on the left-hand panel. Enter the following command in the browser to display the connection details:

```
:server connect
```




```
basic usage :
getting help on Neo4J in the browser:
:help
```

Building recommendation engines

In this section, we will learn how to generate collaborative filtering recommendations using three approaches. They are as follows:

- A simple count of co-rated movies
- Euclidean distance
- Cosine similarity

I would like to highlight a point at this junction. In earlier chapters, we learnt that for building recommendation engines using heuristic approaches, we used similarity calculations such as Euclidean distance/cosine distance. It is not necessary to use only these approaches; we are free to choose our own way of computing the closeness or extracting the similarity between two users just by simple counts as well, for example, similarity between two users can be extracted just by counting the number of the same movies two users have co-rated. If more movies have been co-rated by two users then we may assume that they are similar to each other. If the count of co-rated movies between two people is less then we may assume that their tastes are different.

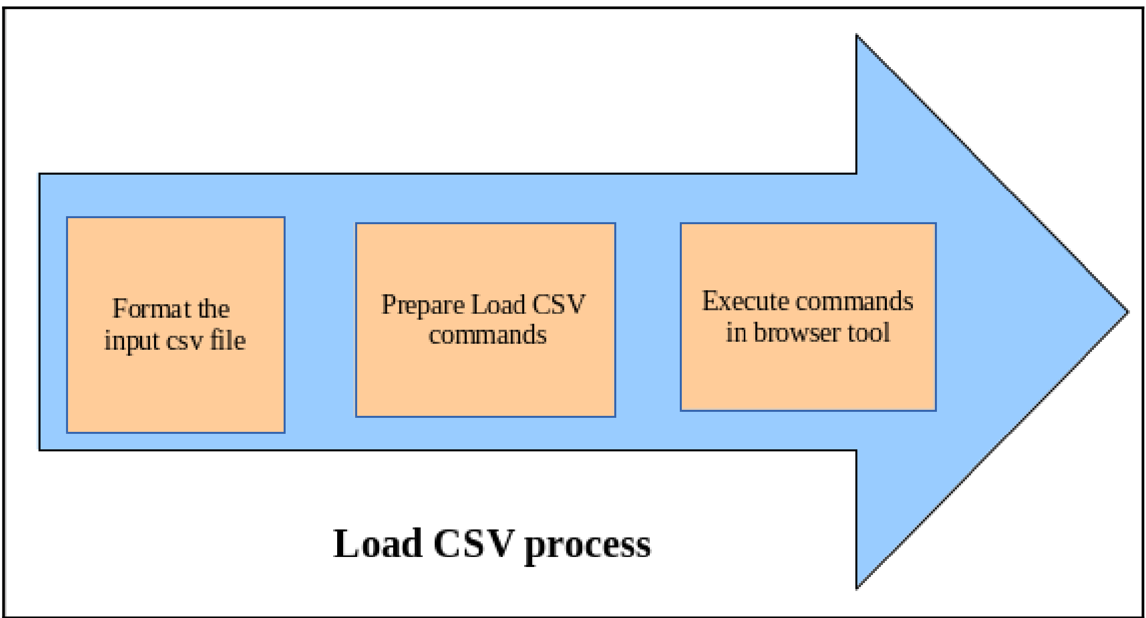
This assumption is taken to build our first recommendation engine and is explained as follows:

For building a collaborative movie recommendation engine, we will build a system based on past movie rating behavior of users. The steps we follow can be summarized as follows:

1. Loading data into an environment
2. Extracting relations and extracting similarity between users
3. Recommendation step

Loading data into Neo4j

Though we have multiple ways of loading data into Neo4j, we use the `Load CSV` option to import the data into the browser tool. The following diagram shows the workflow of the process of loading the CSV process:



The dataset we use for this section is the small sample data set containing Users-Movies-ratings, as shown in the following screenshot:

```
1   Jack Matthews,Lady in the Water,3.0
2   Jack Matthews,Snakes on a Planet,4.0
3   Jack Matthews,You, Me and Dupree,3.5
4   Jack Matthews,Superman Returns,5.0
5   Jack Matthews,The Night Listener,3.0
6   Mick LaSalle,Lady in the Water,3.0
7   Mick LaSalle,Snakes on a Planet,4.0
8   Mick LaSalle,Just My Luck,2.0
9   Mick LaSalle,Superman Returns,3.0
10  Mick LaSalle,You, Me and Dupree,2.0
11  Mick LaSalle,The Night Listener,3.0
12  Claudia Puig,Snakes on a Planet,3.5
13  Claudia Puig,Just My Luck,3.0
14  Claudia Puig,You, Me and Dupree,2.5
15  Claudia Puig,Superman Returns,4.0
16  Claudia Puig,The Night Listener,4.5
17  Lisa Rose,Lady in the Water,2.5
18  Lisa Rose,Snakes on a Planet,3.5
19  Lisa Rose,Just My Luck,3.0
20  Lisa Rose,Superman Returns,3.5
21  Lisa Rose,The Night Listener,3.0
22  Lisa Rose,You, Me and Dupree,2.5
23  Toby,Snakes on a Planet,4.5
24  Toby,Superman Returns,4.0
25  Toby,You, Me and Dupree,1.0
26  Gene Seymour,Lady in the Water,3.0
27  Gene Seymour,Snakes on a Planet,3.5
28  Gene Seymour,Just My Luck,1.5
29  Gene Seymour,Superman Returns,5.0
30  Gene Seymour,You, Me and Dupree,3.5
31  Gene Seymour,The Night Listener,3.0
32  Michael Phillips,Lady in the Water,2.5
33  Michael Phillips,Snakes on a Planet,3.0
34  Michael Phillips,Superman Returns,3.5
```

Let's load the MovieLens data into the Neo4j browser tool as follows:

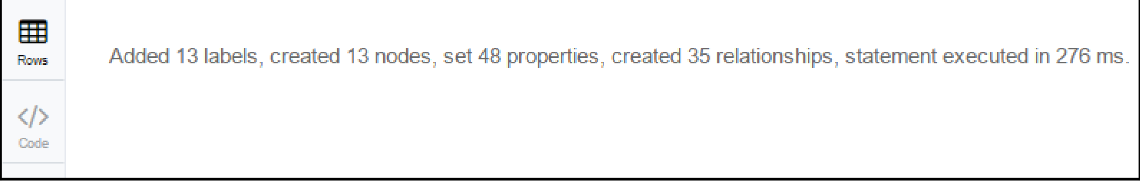
```
LOAD CSV WITH HEADERS FROM file:///ratings.csv AS line
```

Now let's create Users and Movies as nodes and the ratings given by Users to Movies as the relations.

The `MERGE` clause will find the query patterns in the data, and if it doesn't find any it will create one. In the following example below, first it look for a User Node (pattern) and then creates one if it doesn't exist. Since we have just loaded the data into GraphDB, we need to create nodes and establish relationships. Following code will first looks for the mentioned nodes and relationships; if not found it will create new nodes and relationships:

```
LOAD CSV WITH HEADERS FROM file:///C:/Neo4J/test.csv AS line MERGE (U:USER
{USERID : line.UserID})
WITH line, U
MERGE (M:MOVIE {ITEMID : line.ItemId})
WITH line,M,U
MERGE (U)-[:hasRated{RATING:line.Rating}]->(M);
```

When we run the previous query, nodes, relationships, and properties will be created as shown in the following screenshot:



Now, we shall understand each line one by one to make our understanding more clear.

Merge will create `USER` Node from `UserID` columns from the original data:

```
MERGE (U:USER {USERID : line.UserID})
```

The `With` command will take the `User` node and line object to the next part of the query as follows:

```
WITH line, U
```

Now we will create `Movie` Node using `MERGE` and `line.ItemId` object as follows:

```
MERGE (M:MOVIE {ITEMID : line.ItemId})
```

We carry forward the Movie, User nodes, and line object to the next part of the query as follows:

```
WITH line,M,U
```

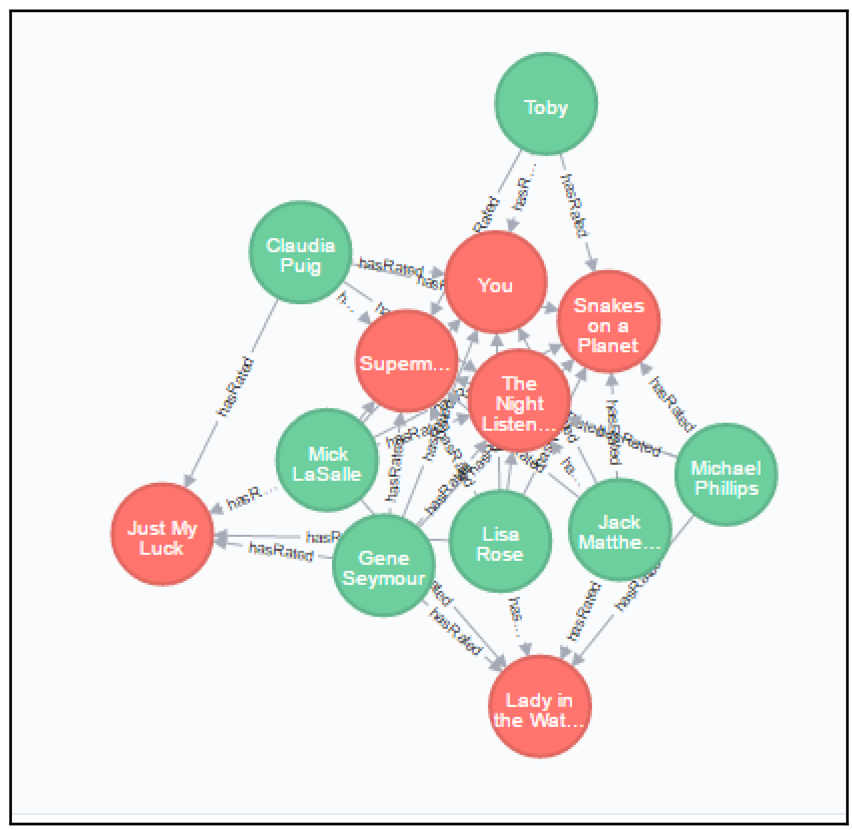
We create a relation between `USER` node and `MOVIE` node as follows:

```
MERGE (U)-[:hasRated{RATING:line.Rating}]->(M);
```

Now that we have loaded the data into Neo4j, we can visualize the movie ratings data with users, movies and ratings as follows:

```
MATCH (U:USER)-[R:hasRated]->(M:MOVIE) RETURN R
```

In the following image, all users are created in green color, and movies are created in red color. We can also see the relationships as arrows with directions.



Generating recommendations using Neo4j

We have now created all the required graphs for building our first recommendation engine using Neo4j. Let's get started.

In the following query `COUNT()` function will count the number of instances, `collect()` will.

The following screenshot will return movie recommendations to the sample user `'TOBY'`:

```
match(u1:USER)-[:hasRated]->(i1:MOVIE)<-[:hasRated]-(u2:USER)-
[:hasRated]->(i2:MOVIE)
with u1,u2, count(i1) as cnt , collect(i1) as Shareditems,i2
where not(u1-[:hasRated]->i2) and u1.USERID='Toby' and cnt> 2
return distinct i2.ITEMID as Recommendations
```

The following query shows the recommendations made to Toby when we run the earlier query:

Recommendations
Just My Luck
Lady in the Water
The Night Listener

The concept behind making recommendations in the previous query is as follows:

- Extract pair of users who have rated the same movies
- Take the count of commonly rated movies by each pair of users
- The higher the commonly rated movie count, the more similar two users are to each other
- The final step is to extract all the movies which similar users have rated, but which have not been rated by the active user, and suggest these new movies as recommendations to the active user

Let's understand the query we just saw step by step:

- In line one, for each user (say `USER1`) who has rated a movie (say `MOVIE1`), select all the users (say `USER2`) who have also rated `MOVIE1`. For this `USER2`, also extract other movies rated by him, apart from `MOVIE1`.
- In line two, we carry similar users (`u1`,`u2`), calculating the count of co-rated movies by `u1`,`u2`, and extracting shared/co-rated movies by `u1`,`u2` to the next part of the query.
- In line three, we now apply a filter where we choose those movies that are not being rated by `u1` and the count of co-rated movies greater than two.
- In line 4 we return new movies rated by similar users to `u1` as recommendations.

Collaborative filtering using the Euclidean distance

In the previous section, we saw how to build recommendation engines using a simple count-based approach for identifying similar users, and then we chose movies from similar users which the active user has not rated or recommended.

In this section, instead of computing the similarity between two users based on the simple count of co-rated movies, let us make use of the rating information and calculate the Euclidean distance, to come up with the similarity score.

The following cypher query will generate recommendations for the user, Toby, based on the Euclidean similarity:

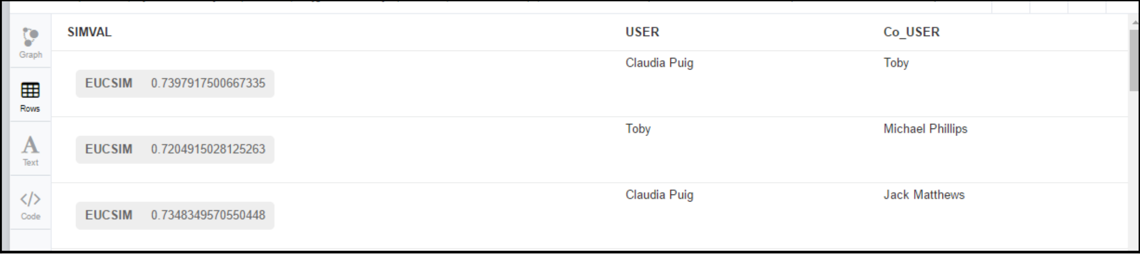
1. The first step is to extract co-rated users by movies and calculate the Euclidean distance between co-rated users as follows:

```
MATCH (u1:USER)-[x:hasRated]-> (b:MOVIE)<-[y:hasRated]-
  (u2:USER)
WITH count(b) AS CommonMovies, u1.username AS user1,
  u2.username AS user2, u1, u2,
collect((toFloat(x.RATING)-toFloat(y.RATING))^2) AS ratings,
collect(b.name) AS movies
WITH CommonMovies, movies, u1, u2, ratings
MERGE (u1)-[s:EUCSIM]->(u2) SET s.EUCSIM = 1-
  (SQRT(reduce(total=0.0, k in extract(i in ratings |
    i/CommonMovies) | total+k))/4)
```

In this code we are using `reduce()` and `extract()` to calculate the Euclidean distance. In order to apply mathematical calculations, we have changed the values to floating point numbers using the `float()` function in the following query.

To see the Euclidean distance values between pairs of users, run the below query:

```
MATCH (u1:USER)-[x:hasRated]-> (b:MOVIE)<-[y:hasRated]-
  (u2:USER)
WITH count(b) AS CommonMovies, u1.username AS user1,
  u2.username AS user2, u1, u2,
collect((toFloat(x.RATING)-toFloat(y.RATING))^2) AS ratings,
collect(b.name) AS movies
WITH CommonMovies, movies, u1, u2, ratings
MERGE (u1)-[s:EUCSIM]->(u2) SET s.EUCSIM = 1-
  (SQRT(reduce(total=0.0, k in extract(i in ratings |
    i/CommonMovies) | total+k))/4) return s as SIMVAL,
      u1.USERID as USER,u2.USERID as Co_USER;
```



SIMVAL	USER	Co_USER
EUCSIM 0.7397917500667335	Claudia Puig	Toby
EUCSIM 0.7204915028125263	Toby	Michael Phillips
EUCSIM 0.7348349570550448	Claudia Puig	Jack Matthews

2. In the second step, we calculate the Euclidean distance using the formula *sqrt(sum((R1-R2)*(R1-R2)))*, where *R1* is the rating given by `Toby` for a `movie1` and *R2* is the other co-rated user's rating for the same `movie1`, and we take the top three similar users, as follows:

```
MATCH (p1:USER {USERID:'Toby'})-[s:EUCSIM]-(p2:USER)
WITH p2, s.EUCSIM AS sim
ORDER BY sim DESC
RETURN distinct p2.USERID AS CoReviewer, sim AS similarity
```

3. The final step is to suggest or recommend non-rated movies from the top three similar users to `Toby` as follows:

```
MATCH (b:USER)-[r:hasRated]->(m:MOVIE), (b)-[s:EUCSIM]-(a:USER
  {USERID:'Toby'})
WHERE NOT((a)-[:hasRated]->(m))
WITH m, s.EUCSIM AS similarity, r.RATING AS rating
ORDER BY m.ITEMID, similarity DESC
WITH m.ITEMID AS MOVIE, COLLECT(rating) AS ratings
WITH MOVIE, REDUCE(s = 0, i IN ratings |toInt(s) +
  toInt(i))*1.0 / size(ratings) AS reco
ORDER BY recoDESC
RETURN MOVIE AS MOVIE, reco AS Recommendation
```

MOVIE	Recommendation
The Night Listener	3.3333333333333335
Lady in the Water	2.6
Just My Luck	2.25

Let us explain the preceding query in detail as follows:

1. As we explained in the first step, we extract co-rated movies by users along with their ratings as follows:

   In our example, Toby has rated three movies: Snakes on a Planet, Superman Returns, and You Me and Dupree. Now we have to extract other common users who have co-rated the same three movies as Toby. For this, we use the following query:

```
MATCH (u1:USER{USERID:'Toby'})-[x:hasRated]-> (b:MOVIE)<-
  [y:hasRated]-(u2:USER)
return u1, u2,
collect(b.ITEMID) AS CommonMovies,
collect(x.RATING) AS user1Rating,
collect(y.RATING) AS user2Rating
```

u1	u2	CommonMovies	user1Rating	user2Rating
USERID Toby	USERID Jack Matthews	[You Me and Dupree, Superman Returns, Snakes on a Planet]	[1.0, 4.0, 4.5]	[3.5, 5.0, 4.0]
USERID Toby	USERID Michael Phillips	[Superman Returns, Snakes on a Planet]	[4.0, 4.5]	[3.5, 3.0]
USERID Toby	USERID Mick LaSalle	[You Me and Dupree, Superman Returns, Snakes on a Planet]	[1.0, 4.0, 4.5]	[2.0, 3.0, 4.0]
USERID Toby	USERID Gene Seymour	[You Me and Dupree, Superman Returns, Snakes on a Planet]	[1.0, 4.0, 4.5]	[3.5, 5.0, 3.5]
USERID Toby	USERID Claudia Puig	[You Me and Dupree, Superman Returns, Snakes on a Planet]	[1.0, 4.0, 4.5]	[2.5, 4.0, 3.5]
USERID Toby	USERID Lisa Rose	[You Me and Dupree, Superman Returns, Snakes on a Planet]	[1.0, 4.0, 4.5]	[2.5, 3.5, 3.5]

2. The second step is to calculate the Euclidean distance between the ratings given to each co-rated movie by the other users, to the movies of Toby, and this is calculated using the following query:

```
MATCH (u1:USER)-[x:hasRated]-> (b:MOVIE)<-[y:hasRated]-
  (u2:USER)
WITH count(b) AS CommonMovies, u1.username AS user1,
  u2.username AS user2, u1, u2,
collect((toFloat(x.RATING)-toFloat(y.RATING))^2) AS ratings,
collect(b.name) AS movies
WITH CommonMovies, movies, u1, u2, ratings
MERGE (u1)-[s:EUCSIM]->(u2) SET s.EUCSIM = 1-
  (SQRT(reduce(total=0.0, k in extract(i in ratings |
    i/CommonMovies) | total+k))/4)
```

In the preceding query, we create and merge new relationships between each of the co-rated users to show the distance between two users, using the MERGE clause. Also, we set the property of the relationship as EUCSIM (which represents the Euclidean distance between each of the co-rated users) using the SET clause.

Now that we have created new relations and set the values of the similarity distances, let us view the results as given by the following query:

```
MATCH (p1:USER {USERID:'Toby'})-[s:EUCSIM]-(p2:USER)
WITH p2, s.EUCSIM AS sim
ORDER BY sim DESC
RETURN distinct p2.USERID AS CoReviewer, sim AS similarity
```

The following screenshot shows the similarity value for Toby with other users:

CoReviewer	similarity
Mick LaSalle	0.7834936490538904
Claudia Puig	0.7397917500667335
Lisa Rose	0.7299691375663392
Michael Phillips	0.7204915028125263
Jack Matthews	0.6047152924789525
Gene Seymour	0.585421901205575

3. The final step is to predict the non-rated movies by Toby, and then recommend the top-rating predicted items. To achieve this, we employ the following steps:

    - Extract the movies rated by similar users to Toby, but not rated by Toby himself
    - Take the ratings given for all the non-rated movies and average them, to predict the ratings that Toby might give to these movies.
    - Display the sorted results as per the predicted rating, in descending order.

   To achieve this, use the following query:

```
MATCH (b:USER)-[r:hasRated]->(m:MOVIE), (b)-[s:EUCSIM]-(a:USER
  {USERID:'Toby'})
WHERE NOT((a)-[:hasRated]->(m))
WITH m, s.EUCSIM AS similarity, r.RATING AS rating ORDER BY
  similarity DESC
WITH m.ITEMID AS MOVIE, COLLECT(rating) AS ratings
WITH MOVIE, REDUCE(s = 0, i IN ratings |toInt(s) +
  toInt(i))*1.0 / size(ratings) AS reco
ORDER BY reco DESC
RETURN MOVIE AS MOVIE, reco AS Recommendation
```

MOVIE	Recommendation
The Night Listener	3.3333333333333335
Lady in the Water	2.6
Just My Luck	2.25

Let us understand the recommendations query line-by-line as follows:

The following query fetches the patterns of all the users who are similar to Toby, and all the movies rated by similar users, as follows:

```
MATCH (b:USER)-[r:hasRated]->(m:MOVIE), (b)-[s:EUCSIM]-(a:USER
{USERID:'Toby'})
```

The `WHERE NOT` clause will filter out all the movies that have been rated by similar users but not by Toby, as follows:

```
WHERE NOT((a)-[:hasRated]->(m))
```

Movies, similarity values and ratings given by the co-users are passed to the next part of the query using the `WITH` clause and the results are ordered by descending similarity value as follows:

```
WITH m, s.EUCSIM AS similarity, r.RATING AS rating ORDER BY similarity DESC
```

After sorting the results based on the similarity values, we further allow values, such as movie name and ratings, to the next part of the query using the `WITH` clause, as follows:

```
WITH m.ITEMID AS MOVIE, COLLECT(rating) AS ratings
```

This is the main step for recommending movies to Toby, predicting the ratings for non-rated movies by Toby by taking the average of movie ratings by similar users to Toby, and using the `REDUCE` clause, as follows:

```
WITH MOVIE, REDUCE(s = 0, i IN ratings |toInt(s) + toInt(i))*1.0 /
size(ratings) AS reco
```

Finally, we sort the final results and return the top movies to Toby as follows:

```
ORDER BY recoDESC
RETURN MOVIE AS MOVIE, reco AS Recommendation
```

Collaborative filtering using Cosine similarity

Now that we have seen recommendations based on simple count and Euclidean distances for identifying similar users, let us use Cosine similarity to calculate the similarity between users.

The following query is used to create a new relation called similarity between users:

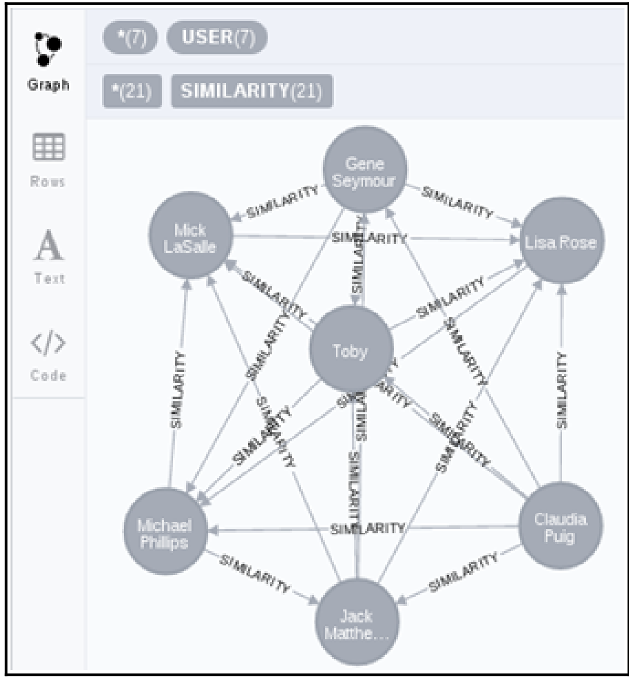
```
MATCH (p1:USER)-[x:hasRated]->(m:MOVIE)<-[y:hasRated]-(p2:USER)
WITH SUM(toFloat(x.RATING) * toFloat(y.RATING)) AS xyDotProduct,
SQRT(REDUCE(xDot = 0.0, a IN COLLECT(toFloat(x.RATING)) | xDot
+toFloat(a)^2)) AS xLength,
SQRT(REDUCE(yDot = 0.0, b IN COLLECT(toFloat(y.RATING)) | yDot +
toFloat(b)^2)) AS yLength,
p1, p2
MERGE (p1)-[s:SIMILARITY]-(p2)
SET s.similarity = xyDotProduct / (xLength * yLength)
```

Code

Set 42 properties, created 21 relationships, statement executed in 292 ms.

Let us explore the similarity values as follows:

```
match(u:USER)-[s:SIMILARITY]->(u2:USER) return s;
```



We calculate the similar users for Toby as follows:

For the active user Toby, let us display the similarity values with respect to other users as follows:

```
MATCH (p1:USER {USERID:'Toby'})-[s:SIMILARITY]-(p2:USER)
WITH p2, s.similarity AS sim
ORDER BY sim DESC
LIMIT 5
RETURN p2.USERID AS Neighbor, sim AS Similarity
```

The following image displays the results by running the previous Cypher query; the results show the similarity value for Toby, with respect to other users.

Neighbor	Similarity
Lisa Rose	0.9982743731749959
Mick LaSalle	0.9965457582448796
Claudia Puig	0.9921572222264535
Michael Phillips	0.9908301680442989
Jack Matthews	0.9856838997418295

Returned 5 rows in 42 ms.

Now let us start our recommendations of movies to Toby. The recommendation process is very similar to what we have done in the previous approach, as follows:

- Extract movies rated by similar users to Toby but not rated by Toby himself
- Take the ratings given for all the non-rated movies and average them to predict the ratings that Toby might give to these movies
- Display the sorted results as per the predicted rating, in descending order

We use the following code:

```
MATCH (b:USER)-[r:hasRated]->(m:MOVIE), (b)-[s:SIMILARITY]-(a:USER
  {USERID:'Toby'})
WHERE NOT((a)-[:hasRated]->(m))
WITH m, s.similarity AS similarity, r.RATING AS rating
ORDER BY m.ITEMID, similarity DESC
WITH m.ITEMID AS MOVIE, COLLECT(rating) AS ratings
WITH MOVIE, REDUCE(s = 0, i IN ratings |toInt(s) + toInt(i))*1.0 /
  size(ratings) AS reco
ORDER BY reco DESC
RETURN MOVIE AS MOVIE, reco AS Recommendation
```

MOVIE	Recommendation
The Night Listener	3.3333333333333335
Lady in the Water	2.6
Just My Luck	2.25

Summary

Kudos! We have created recommendation engines using the Neo4j graph database. Let us recap what we have learned in this chapter. We started the chapter by giving a very brief introduction to graphs and graph databases. We covered a very short introduction to the core Neo4j graph database concepts such as the labeled property graph model, Nodes, Labels, Relationships, Cypher query language, Patterns, Node syntax, and Relationship Syntax.

We also touched upon Cypher clauses that are useful in building recommendations, such as `MATCH` ,`CREATE` ,`LOADCSV` ,`RETURN` ,`AS` ,and `WITH`.

Then we moved onto installation and setting up Neo4j from the browser tool in the Windows and Linux platforms.

Once the entire working environment was setup to build our recommendation engines, we chose sample movie ratings data and implemented three types of collaborative filtering, such as simple distance based, Euclidean similarity based, and Cosine similarity based recommendations. In the next chapter, we will be exploring Mahout, a machine learning library available on Hadoop, for building scalable recommender systems.

9
Building Scalable Recommendation Engines with Mahout

Imagine that you have just launched an online e-commerce website to sell clothes designed by you and you are lucky enough to make your business kick-start well and make it a successful venture. With more web traffic coming to your site, the most obvious choice is to implement a recommendation engine on your website with features such as people who visited something also visited something else, items similar to the current item, and so on. Since your website is new and successful, you have implemented a recommendation engine using popular tools, such as R and Python. The recommendation functionality is deployed and works well, adding more value to the success of the business. Now with more business coming in and with an increase in your user base, the most likely problem you might face with the website is that your customers start complaining that your website is becoming slow.

Upon analyzing the root cause, the obvious reason would be that the recommender features that are added to the website are slowing down the site. This is bound to happen because of the limitation of collaborative filtering algorithms used to cater for recommendations. Every time we calculate the similarity between users, the entire user base will be loaded into the memory and the similarity values would be calculated. This operation will be fast with a small user base. Assume that with a large use base, such as one million users, the collaborative filtering model will be thrown out of the memory exception. By increasing the RAM capability, we might address this to some extent, but it still won't help us. Increasing the RAM would be bad idea as it shoots up the infrastructure cost.

The best way is to redesign the recommender engine on a distributed platform, such as Hadoop. This is where Apache Mahout will come in handy as it is an open source machine learning library built for the distributed platform, Apache Hadoop.

In this chapter, we will be covering the following sections:

- Mahout general introduction
- Setting up Mahout standalone and distributed mode
- Core building blocks of Mahout
- Building and evaluating recommendation engines with Mahout such as user-based collaborative filtering, item-based collaborative filtering, SVD recommendation engines, and ALS recommendation engines.

Mahout – a general introduction

Apache Mahout is an open source java library built on top of Apache Hadoop, which provides large-scale machine learning algorithms. Though this library was originally started with the MapReduce paradigm, the framework currently offers bindings to Apache Spark, H2O, and Apache Flink. The latest version of Mahout supports collaborative filtering recommendation engines, clustering, classification, dimensionality reduction, H2O, and spark bindings.

The major features of Mahout 0.12.2 are as follows:

- An extensible programming environment and framework for the building of scalable algorithms
- Support for Apache Spark, Apache Flink, and H2O algorithms
- Samsara, a vector Math environment similar to the R programming language

As mentioned in the previous section, though many things are possible with Mahout, we will be limiting our discussion to building recommendation engines using Mahout. Mahout provides support for both the standalone mode, where the recommendation model or application can be deployed on a single server, and the distributed mode, where the recommendation model can be deployed on a distributed platform.

Setting up Mahout

In this section, we shall look at setting up Mahout in standalone and distributed mode.

The standalone mode – using Mahout as a library

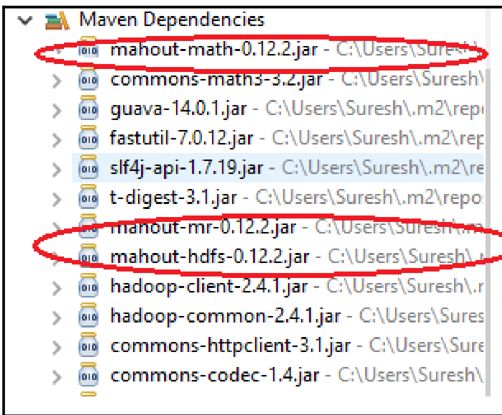
The standalone mode of Mahout usually involves two steps:

- Adding Mahout libraries to the Java application that wants to use the Mahout capabilities
- Calling Mahout recommendation engine functions to build the recommender application

Running an application that uses Mahout requires the following dependencies to be added to the `pom.xml` file of your Java Maven project:

```
18   <dependency>
19     <groupId>org.apache.mahout</groupId>
20     <artifactId>mahout-math</artifactId>
21     <version>0.12.2</version>
22 </dependency>
23 <dependency>
24     <groupId>org.apache.mahout</groupId>
25     <artifactId>mahout-mr</artifactId>
26     <version>0.12.2</version>
27 </dependency>
```

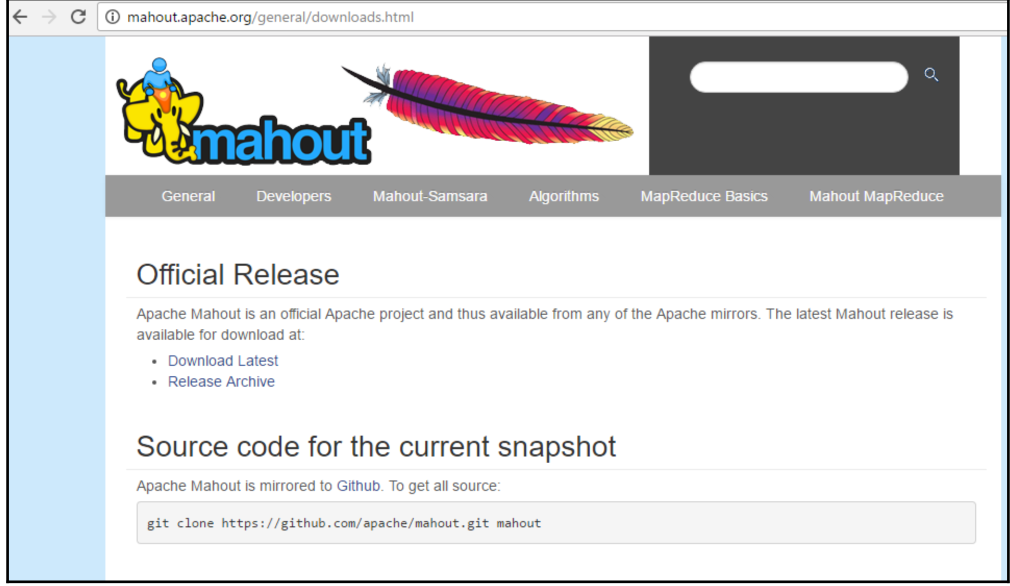
The preceding dependencies will download all the required jars or libraries required to run the Mahout functionalities, as shown in the following screenshot:



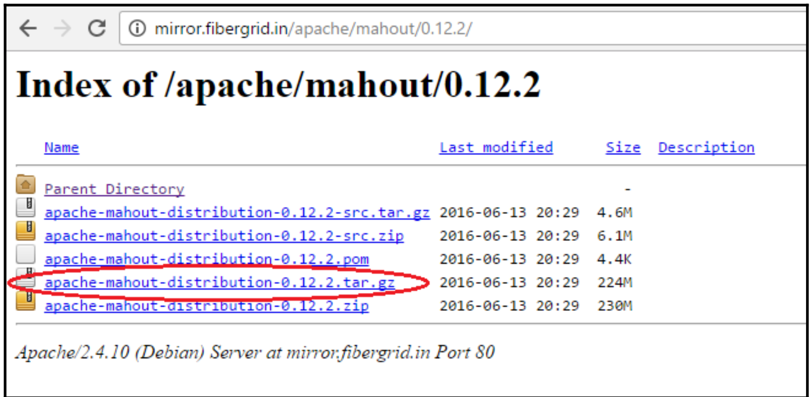
Another step is to go to the official Apache Mahout website and download the required Mahout jar files, as shown here:

The latest Mahout library can be downloaded from the Apache Mahout official website at `http://mahout.apache.org/general/downloads.html`.

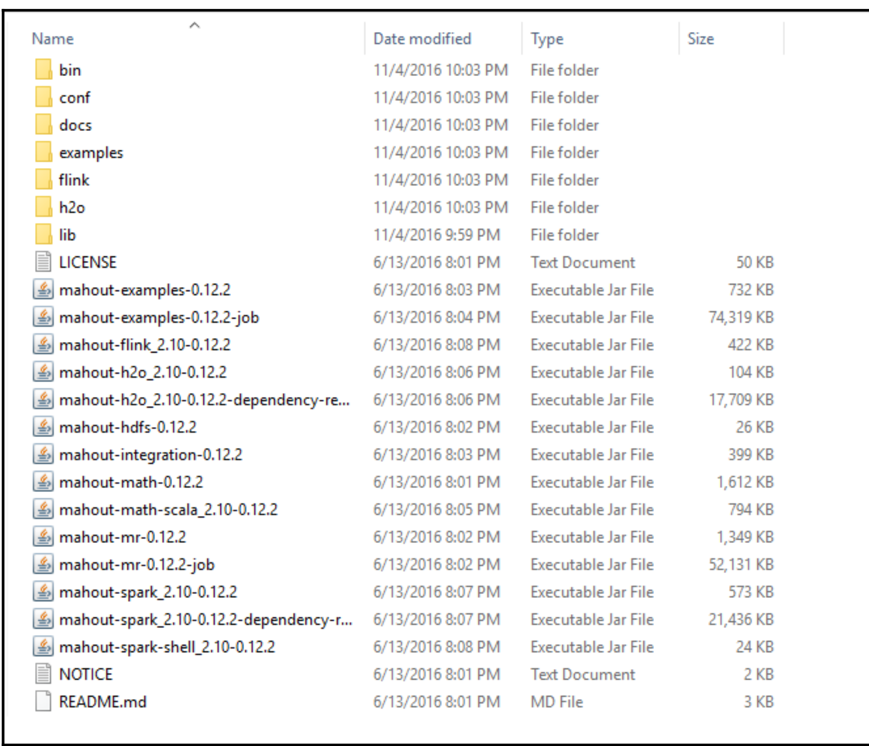
The following image shows the screenshot of the above mentioned URL:



Download the tar file(tar files are just executable) instead of source files, as we just need the jar files of Mahout to build recommendation engines:



After downloading the tar file, just extract all the files and add the required jars to the Java application:

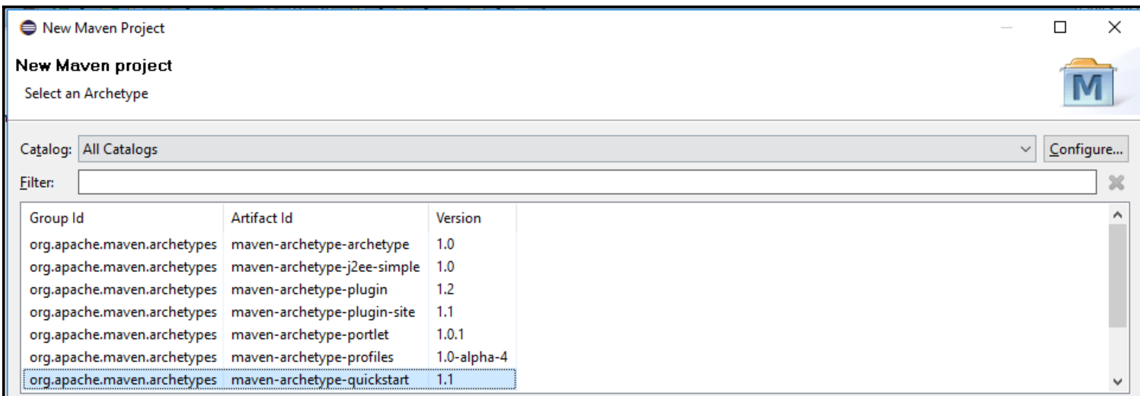


With this minimal setup, let's build a very basic recommendation engine using Java Eclipse.

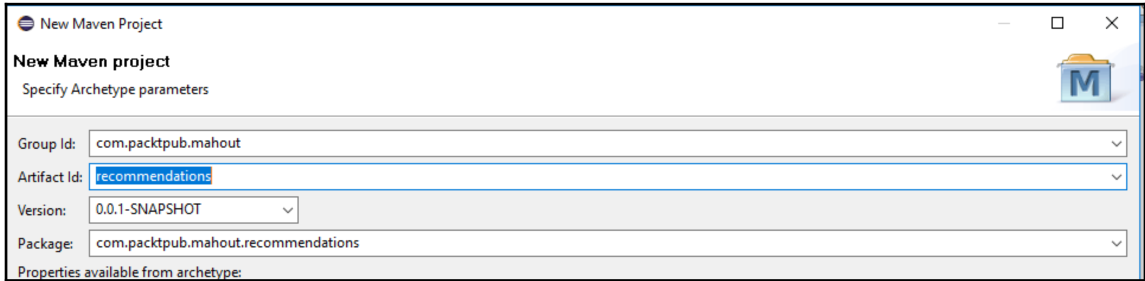
The minimal setup just requires the following steps:

1. Create a Java Maven project in Eclipse with the following attribute selection:

   The following image shows the screenshot of creating a new Maven project setup step 1:



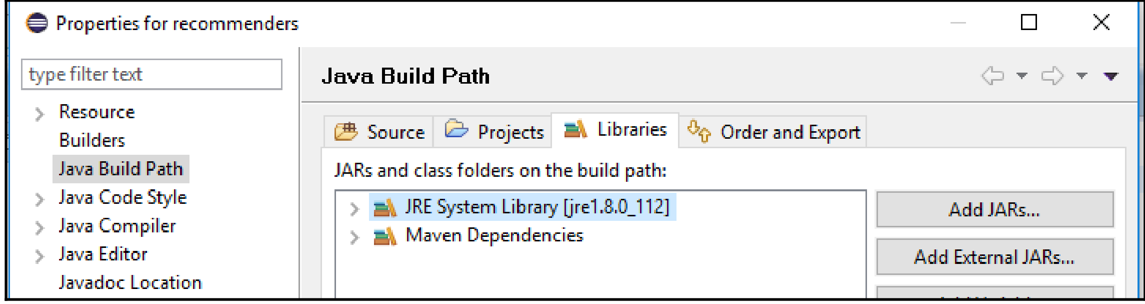
   In the following image, add the **Artifact Id** "`recommendations`":



2. A Maven project will be created with `app.java` as the default class. We can make changes in this class to build our standalone recommendation engine:

```
app.java
1  package com.packtpub.mahout.recommenders;
2
3  public class app {
4
5      public static void main(String[] args) {
6          // TODO Auto-generated method stub
7
8      }
9
10 }
```

3. Set Java runtime as 1.7 or higher, as shown in the next screenshot:

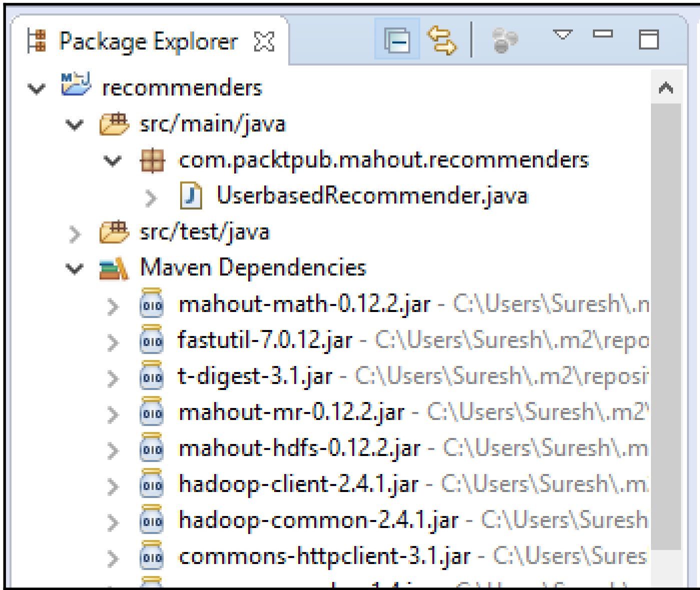


4. Set the required Maven dependencies listed as **mahout-mr**, **mahout-math**, **slf4j-log4j**, **commons-math3**, and **guava**; this will download the required jars for the application to run, as shown in the following screenshot:

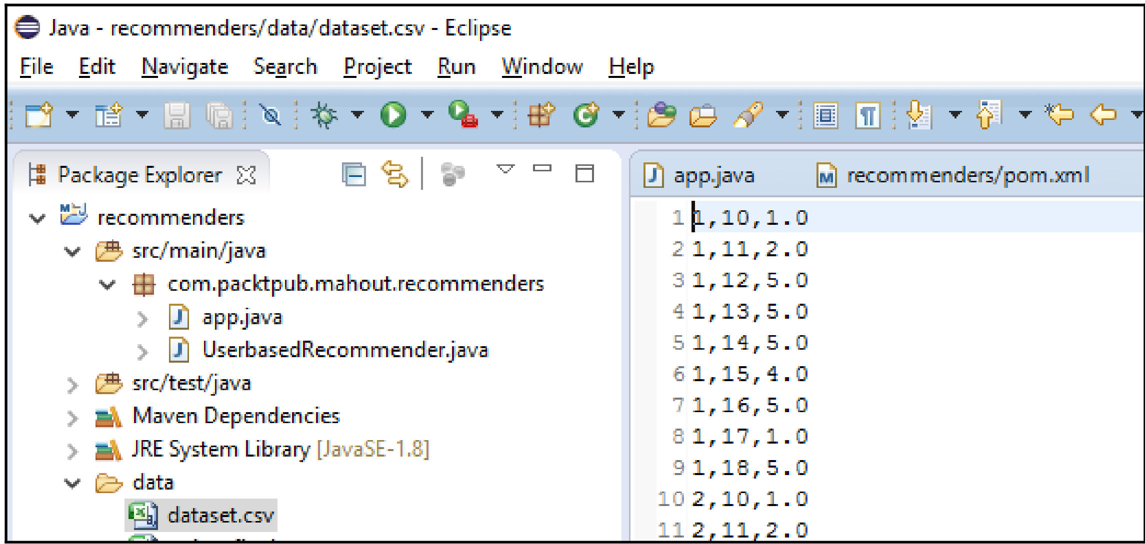
app.java | recommenders/pom.xml

```
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  <dependencies>
  <dependency>
    <groupId>org.apache.mahout</groupId>
    <artifactId>mahout-math</artifactId>
    <version>0.12.2</version>
</dependency>
<dependency>
    <groupId>org.apache.mahout</groupId>
    <artifactId>mahout-mr</artifactId>
    <version>0.12.2</version>
</dependency>
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.21</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.21</version>
</dependency>
<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>19.0</version>
</dependency>
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-math3</artifactId>
```

5. These dependencies can be seen in the following screenshot:



6. Create a folder called `data` in the project and create a sample dataset, as shown in the following screenshot:



7. Now rename `app.java` to the `UserbasedRecommender.java` file. Write the code in the java class to build the basic user-based recommender system:

```
package com.packtpub.mahout.recommenders;

import java.io.File;
import java.io.IOException;
import java.util.List;

import org.apache.mahout.cf.taste.common.TasteException;
import org.apache.mahout.cf.taste.impl.model.file.FileDataModel;
import
org.apache.mahout.cf.taste.impl.neighborhood.ThresholdUserNeighborhood;
import
org.apache.mahout.cf.taste.impl.recommender.GenericUserBasedRecommender;
import
org.apache.mahout.cf.taste.impl.similarity.PearsonCorrelationSimilarity;
import org.apache.mahout.cf.taste.model.DataModel;
import org.apache.mahout.cf.taste.neighborhood.UserNeighborhood;
import org.apache.mahout.cf.taste.recommender.RecommendedItem;
import org.apache.mahout.cf.taste.recommender.UserBasedRecommender;
import org.apache.mahout.cf.taste.similarity.UserSimilarity;

//class for generating User Based Recommendation
public class UserbasedRecommender
{
    public static void main( String[] args ) throws TasteException,
IOException
    {
    //creating data model
         DataModel model = new FileDataModel(new File("data/dataset.csv"));
    // creating pearson similarity between users
    UserSimilarity similarity = new PearsonCorrelationSimilarity(model);
         //creating user neighborhood
           UserNeighborhood neighborhood = new
ThresholdUserNeighborhood(0.1,
similarity, model);
      // creating recommender model
            UserBasedRecommender recommender = new
GenericUserBasedRecommender(model, neighborhood, similarity);
        //generating 3 recommendations for user 2
    List<RecommendedItem> recommendations = recommender.recommend(2, 3);
    for (RecommendedItem recommendation : recommendations) {
      System.out.println(recommendation);
    }
    }
}
```

Running the preceding code will generate the recommendations for user 2, as shown in the following screenshot:

```
Java - recommenders/src/main/java/com/packtpub/mahout/recommenders/UserbasedRecommender.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Problems @ Javadoc Declaration Console
<terminated> UserbasedRecommender [Java Application] C:\Program Files\Java\jre1.8.0_112\bin\javaw.exe (Nov 10, 2016, 11:12:09 PM)
log4j:WARN No appenders could be found for logger (org.apache.mahout.cf.taste.impl.model.file.FileDataModel).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
RecommendedItem[item:12, value:4.8328104]
RecommendedItem[item:13, value:4.6656213]
RecommendedItem[item:14, value:4.331242]
```

Boom! We have created our first user-based recommendation engine. Don't worry about what we have done or what's happening; everything will become clearer in the next few sections. For now, just try to understand how the Mahout library can be used in the standalone mode to build recommendation engines.

Setting Mahout for the distributed mode

We have seen how to use Mahout libraries in the standalone mode. In this section, let's see how to setup Mahout on a distributed platform, such as HDFS. The following are the requirements in order to set up Mahout:

- Java 7 and higher
- Apache Hadoop
- Apache Mahout

Setting up Java 7 and installing Hadoop is out of the scope of the is book. We can find very good resources online on how to set up Hadoop. Assuming Hadoop is already set up, follow these steps to set up Mahout:

Download and extract the latest Mahout distribution from Apache Mahout website, as explained earlier.

Let's set up environment values:

```
Export JAVA_HOME = path/to/java7 or more
export MAHOUT_HOME = /home/softwares/ apache-mahout-distribution-0.12.2
export MAHOUT_LOCAL = true #for standalone mode
export PATH = $MAHOUT_HOME/bin
export CLASSPATH = $MAHOUT_HOME/lib:$CLASSPATH
```

Unset `MAHOUT_LOCAL` in order to run it on the Hadoop cluster.

Once the environment variables are set up, use the following commands in the command line to run a recommendation engine on the distributed platform.

Using the following code, we are generating item-based recommendations using the log likelihood similarity:

```
mahout recommenditembased -s SIMILARITY_LOGLIKELIHOOD -i mahout/data.txt -o
mahout/output1 --numRecommendations 25

[cloudera@quickstart ~]$ mahout recommenditembased -s
SIMILARITY_LOGLIKELIHOOD -i mahout/data.txt -o mahout/output1 --
numRecommendations 25
MAHOUT_LOCAL is not set; adding HADOOP_CONF_DIR to classpath.
Running on hadoop, using /usr/lib/hadoop/bin/hadoop and
HADOOP_CONF_DIR=/etc/hadoop/conf
MAHOUT-JOB: /usr/lib/mahout/mahout-examples-0.9-cdh5.4.0-job.jar
16/11/10 11:05:09 INFO common.AbstractJob: Command line arguments: {--
booleanData=[false], --endPhase=[2147483647], --input=[mahout/data.txt], --
maxPrefsInItemSimilarity=[500], --maxPrefsPerUser=[10], --
maxSimilaritiesPerItem=[100], --minPrefsPerUser=[1], --
numRecommendations=[25], --output=[mahout/output1], --
similarityClassname=[SIMILARITY_LOGLIKELIHOOD], --startPhase=[0], --
tempDir=[temp]}
16/11/10 11:05:09 INFO common.AbstractJob: Command line arguments: {--
booleanData=[false], --endPhase=[2147483647], --input=[mahout/data.txt], --
minPrefsPerUser=[1], --output=[temp/preparePreferenceMatrix], --
ratingShift=[0.0], --startPhase=[0], --tempDir=[temp]}
16/11/10 11:05:10 INFO Configuration.deprecation: mapred.input.dir is
deprecated. Instead, use mapreduce.input.fileinputformat.inputdir
16/11/10 11:05:10 INFO Configuration.deprecation:
mapred.compress.map.output is deprecated. Instead, use
mapreduce.map.output.compress
16/11/10 11:05:10 INFO Configuration.deprecation: mapred.output.dir is
deprecated. Instead, use mapreduce.output.fileoutputformat.outputdir
```

```
16/11/10 11:05:11 INFO client.RMProxy: Connecting to ResourceManager at
/0.0.0.0:8032
16/11/10 11:05:20 INFO input.FileInputFormat: Total input paths to process
: 1
16/11/10 11:05:22 INFO mapreduce.JobSubmitter: number of splits:1
16/11/10 11:05:24 INFO mapreduce.JobSubmitter: Submitting tokens for job:
job_1478802142793_0003
16/11/10 11:05:42 INFO impl.YarnClientImpl: Submitted application
application_1478802142793_0003
16/11/10 11:05:52 INFO mapreduce.Job: The url to track the job:
http://quickstart.cloudera:8088/proxy/application_1478802142793_0003/
16/11/10 11:05:52 INFO mapreduce.Job: Running job: job_1478802142793_0003
16/11/10 11:16:45 INFO mapreduce.Job: Job job_1478802142793_0011 running in
uber mode : false
16/11/10 11:16:45 INFO mapreduce.Job:  map 0% reduce 0%
16/11/10 11:16:58 INFO mapreduce.Job:  map 100% reduce 0%
16/11/10 11:17:19 INFO mapreduce.Job:  map 100% reduce 100%
16/11/10 11:17:20 INFO mapreduce.Job: Job job_1478802142793_0011 completed
successfully
16/11/10 11:17:21 INFO mapreduce.Job: Counters: 49
File System Counters
-------------------------------
-------------------------------
Bytes Written=28
16/11/10 11:17:21 INFO driver.MahoutDriver: Program took 732329 ms
(Minutes: 12.205483333333333)
```

The output is as follows:

```
[cloudera@quickstart ~]$ hadoop fs -cat mahout/output1/part-r-
00000
3 [10:3.8597424]
4 [13:4.0]
```

Core building blocks of Mahout

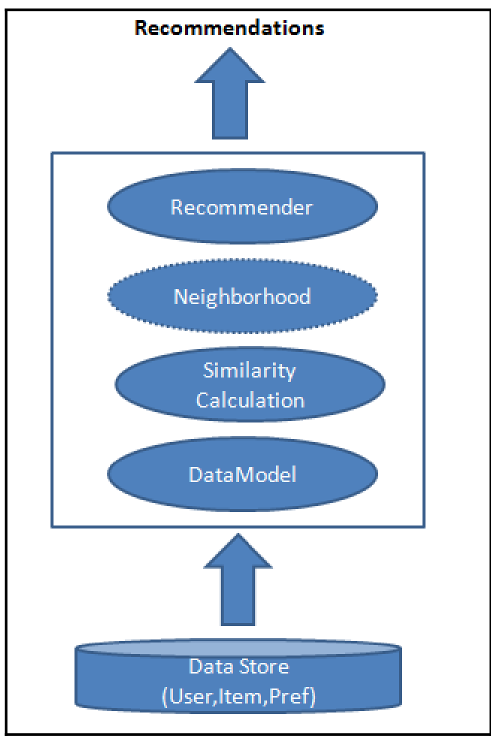
Like any other recommendation engine framework, Mahout also provides a rich set of components to build customized recommender systems that are enterprise-ready, scalable, flexible, and that perform well.

The key components of Mahout are as follows:

- DataModel
    - Similarity: UserSimilarity, ItemSimilarity
- User neighborhood
- Recommender
- Recommender evaluator

Components of a user-based collaborative recommendation engine

In this section, we shall cover the components required for building a user-based collaborative filtering system.



The components of a user-based collaborative recommendation engine are as follows:

- **DataModel**: A DataModel implementation allows us to store and provide access to the user, item, and preference data required for computation. The DataModel component allows us to pull data from the data source. Mahout provides **MySQLJDBCDataModel**, which allows us to pull data from the database via JDBC and MySQL. For the purpose of our example, we use the **FileDataModel** interface to access data from files that Mahout exposes.

  Some other DataModels exposed by Mahout are as follows:

  - **HBaseDataModel**: (`http://apache.github.io/mahout/0.10.1/docs/mahout-integration/org/apache/mahout/cf/taste/impl/model/hbase/HBaseDataModel.html`)
  - **GenericJDBCDataModel**: (`http://apache.github.io/mahout/0.10.1/docs/mahout-integration/org/apache/mahout/cf/taste/impl/model/jdbc/GenericJDBCDataModel.html`)
  - **PostgreSQLJDBCDataModel**: (`http://apache.github.io/mahout/0.10.1/docs/mahout-integration/org/apache/mahout/cf/taste/impl/model/jdbc/PostgreSQLJDBCDataModel.html`)
  - **MongoDBDataModel**: (`http://apache.github.io/mahout/0.10.1/docs/mahout-integration/org/apache/mahout/cf/taste/impl/model/mongodb/MongoDBDataModel.html`)

  Mahout expects the user data to be in the format of a userID, itemID, preference triplet. The preference values can be either continuous or Boolean. Mahout has support for both continuous and Boolean preference values. Each input triplet, containing userID, itemID, and preference, which we supply to the DataModel, will be represented in a memory-efficient **Preference object** or a **PreferenceArray** object.

- **UserSimilarity**: The UserSimilarity interface calculates the similarity between two users. The implementations of UserSimilarity return values in the range of -1.0 to 1.0 usually, with 1.0 being the perfect similarity. In previous chapters, we saw the multiple ways in which we can calculate the similarity between users, such as Euclidean Distance, Pearson Coefficient, cosine distance, and so on. There are many implementations of the UserSimilarity interface to calculate the User Similarity, which are listed as follows:
    - CachingUserSimilarity
    - CityBlockSimilarity
    - EuclideanDistanceSimilarity
    - GenericUserSimilarity
    - LogLikelihoodSimilarity
    - PearsonCorrelationSimilarity
    - SpearmanCorrelationSimilarity
    - TanimotoCoefficientSimilarity
    - UncenteredCosineSimilarity
- **ItemSimilarity**: Similar to UserSimilarity, Mahout also provides the ItemSimilarity interface, analogous to UserSimilarity, which can be used to calculate the similarity between items. The implementations of UserSimilarity return values in the range of -1.0 to 1.0 usually, with 1.0 being the perfect similarity:
    - AbstractItemSimilarity
    - AbstractJDBCItemSimilarity
    - CachingItemSimilarity
    - CityBlockSimilarity
    - EuclideanDistanceSimilarity
    - FileItemSimilarity
    - GenericItemSimilarity
    - LogLikelihoodSimilarity
    - MySQLJDBCInMemoryItemSimilarity
    - MySQLJDBCItemSimilarity
    - PearsonCorrelationSimilarity
    - SQL92JDBCInMemoryItemSimilarity
    - SQL92JDBCItemSimilarity
    - TanimotoCoefficientSimilarity
    - UncenteredCosineSimilarity

- **UserNeighborhood**: In a user-based recommender, recommendations generated for the active user are produced by finding a neighborhood of similar users. UserNeighborhood usually refers to a way to determine the neighborhood for a given active user, for example, the ten nearest neighbors to take into account while generating recommendations.

  These neighborhood classes implement the UserSimilarity interface for their operation.
  The following are the implementations of the neighborhood interface:

  - CachingUserNeighborhood
  - NearestNUserNeighborhood
  - ThresholdUserNeighborhood

- **Recommender**: A recommender is the core abstraction in Mahout. Given the `DataModel` object as the input, it produces recommendations for items to users. The implementations of the recommender interface are as follows:
  - AbstractRecommender
  - CachingRecommender
  - GenericBooleanPrefItemBasedRecommender
  - GenericBooleanPrefUserBasedRecommender
  - GenericItemBasedRecommender
  - GenericUserBasedRecommender
  - ItemAverageRecommender
  - ItemUserAverageRecommender
  - RandomRecommender
  - RecommenderWrapper
  - SVDRecommender

Building recommendation engines using Mahout

Now that we have covered the core building blocks of the Mahout recommendation engine framework, let's start building recommendations. In this section, we will look at a series of different recommendation engines implemented using the standalone mode. The recommendation engine capabilities are using implementations of the `org.apache.mahout.cf.taste.impl` package.

The recommendation engines we see in this section are as follows:

- User-based collaborative filtering
- Item-based collaborative filtering
- SVD recommenders

Dataset description

Before we get into recommender implementations, let's look at the dataset we use in this section. For this section, we use the restaurant and consumer data dataset available from the UCI machine learning dataset repository from the following URL:

`https://archive.ics.uci.edu/ml/datasets/Restaurant+%26+consumer+data`

This dataset can be used to build collaborative filtering applications using consumer preference information. The dataset, the file downloaded from the previous link, contains nine files listed in the following figure. Of all the files in this exercise, we use the `rating_final.csv` file, which contains attributes such as userID, placeID, rating, food_rating, and service_rating. But for our use cases, we only use userID, placeID, and rating. We can think of the data as a preference value given to Place by a given user.

We will have to make use of the previously created project in the setup session.
Add the input `ratings_final.csv` file to the *data* folder to the current project structure.

So first, let's preprocess the original raw data into the required format of the userID, placeID, and rating triplet. Here's the raw dataset used for this exercise:

```
 1 userID,placeID,rating,food_rating,service_rating
 2 U1077,135085,2,2,2
 3 U1077,135038,2,2,1
 4 U1077,132825,2,2,2
 5 U1077,135060,1,2,2
 6 U1068,135104,1,1,2
 7 U1068,132740,0,0,0
 8 U1068,132663,1,1,1
 9 U1068,132732,0,0,0
10 U1068,132630,1,1,1
11 U1067,132584,2,2,2
12 U1067,132733,1,1,1
13 U1067,132732,1,2,2
14 U1067,132630,1,0,1
15 U1067,135104,0,0,0
16 U1067,132560,1,0,0
17 U1103,132584,1,2,1
18 U1103,132732,0,0,2
```

The following program will prepare the required triplet dataset, implemented as follows:

- Read each line from the `ratings_final.csv` file
- Extract the first three columns
- Write the extracted columns from the previous step to a new `recoDataset.csv` file

The following java program implements the previously explained steps:

```
package com.packtpub.mahout.recommenders;

import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

import au.com.bytecode.opencsv.CSVReader;
import au.com.bytecode.opencsv.CSVWriter;

public class Preprocessdata  {

public static void main(String[] args) throws IOException {
String fileName = "data/rating_final.csv";
```

```
String csv = "data/recoDataset.csv";
CSVReader csvReader = new CSVReader(new FileReader(fileName));
String[] row = null;
List<String[]> data = new ArrayList<String[]>();
CSVWriter writer = new CSVWriter(new FileWriter(csv),
CSVWriter.DEFAULT_SEPARATOR,
CSVWriter.NO_QUOTE_CHARACTER);
while((row = csvReader.readNext()) != null) {
if(!row[0].contains("userID")){
data.add(new String[] {row[0].substring(1), row[1],row[2]});
}
}
writer.writeAll(data);
writer.close();
csvReader.close();
}

}
```

Upon running the preceding java program, the final dataset that we use to build recommendation engines will be created under the *data* folder as the `recoDataset.csv` file. The following is a sample dataset:

```
 1 1077,135085,2
 2 1077,135038,2
 3 1077,132825,2
 4 1077,135060,1
 5 1068,135104,1
 6 1068,132740,0
 7 1068,132663,1
 8 1068,132732,0
 9 1068,132630,1
10 1067,132584,2
11 1067,132733,1
12 1067,132732,1
13 1067,132630,1
14 1067,135104,0
15 1067,132560,1
16 1103,132584,1
17 1103,132732,0
18 1103,132630,1
```

Now that we have preprocessed the required data, let's start building our recommendation engines with the Mahout framework.

User-based collaborative filtering

Just for the sake of a refresher: the user-based recommender system generates recommendations based on the UserSimilarity calculation between users and then uses UserNeighborhood to choose top-N users and then generate recommendations.

Let's first execute the following code and then we shall look at the code line by line. We will use the Euclidean Distance similarity and Nearest Neighborhood methods to generate recommendations:

```
package com.packtpub.mahout.recommenders;

import java.io.File;
import java.io.IOException;
import java.util.List;

import org.apache.mahout.cf.taste.common.TasteException;
import org.apache.mahout.cf.taste.impl.model.file.FileDataModel;
import
org.apache.mahout.cf.taste.impl.neighborhood.NearestNUserNeighborhood;
import
org.apache.mahout.cf.taste.impl.recommender.GenericUserBasedRecommender;
import
org.apache.mahout.cf.taste.impl.similarity.EuclideanDistanceSimilarity;
import org.apache.mahout.cf.taste.model.DataModel;
import org.apache.mahout.cf.taste.neighborhood.UserNeighborhood;
import org.apache.mahout.cf.taste.recommender.RecommendedItem;
import org.apache.mahout.cf.taste.recommender.UserBasedRecommender;
import org.apache.mahout.cf.taste.similarity.UserSimilarity;

//class for generating User Based Recommendation
public class UserbasedRecommendations
{
    public static void main( String[] args ) throws TasteException,
IOException
    {
    //creating data model
    DataModel model = new FileDataModel(new File("data/recoDataset.csv"));
    // creating Euclidean distance similarity between users
    UserSimilarity similarity = new EuclideanDistanceSimilarity(model);
    //creating user neighborhood
    UserNeighborhood neighborhood = new NearestNUserNeighborhood(10,
```

```
similarity, model);
    // creating recommender model
    UserBasedRecommender recommender = new
GenericUserBasedRecommender(model, neighborhood, similarity);
    //generating 3 recommendations for user 1068
    List<RecommendedItem> recommendations = recommender.recommend(1068, 3);
    for (RecommendedItem recommendation : recommendations) {
      System.out.println(recommendation);
    }
    }
}
```

Running this program generates recommendations shown in the following figures. We are generating the top three user-based item recommendations to `UserId - 1068`:

From the result, we can conclude that for `UserId - 1068`, the top three recommended places along with similarity values are as follows:

```
Problems  @ Javadoc  Declaration  Console
<terminated> UserbasedRecommender [Java Application] C:\Program Files\Java\jre1.8.0_112\bin\javaw.exe (Nov 11, 2016, 11:04:44 PM)
log4j:WARN No appenders could be found for logger (org.apache.mahout.cf.taste.impl.model.file.FileDataModel).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
RecommendedItem[item:132613, value:1.2205102]
RecommendedItem[item:132667, value:1.0]
RecommendedItem[item:132584, value:0.98069793]
```

Let's now look at the code line by line; just recall the core building blocks of the Mahout recommendations section. We need DataModel, Similarity calculation, UserNeighborhood, recommender, and generating recommendations. This order is used in the previous code:

1. The code in the `UserbasedRecommender.main` method creates a data source from the `data/recoDataset.csv` CSV file using `org.apache.mahout.cf.taste.impl.model.file.FileDataModel.FileDataModel class`. This class constructor gets the `Java.io.File` instance containing the preferences data and creates the `DataModel` class instance model:

   ```
   //creating data model
   DataModel model = new FileDataModel(new
     File("data/recoDataset.csv"));
   ```

2. In this step, we create the UserSimilarity instance: the similarity calculation between all users using `org.apache.mahout.cf.taste.impl.similarity.EuclideanDistanceSimilarity class`, which takes the `FileDataModel` instance created in the previous step as the constructor parameter:

   ```
   // creating Euclidean distance similarity between users
   UserSimilarity similarity = new
     EuclideanDistanceSimilarity(model);
   ```

3. In this step, we create the UserNeighborhood instance: the neighborhood using `org.apache.mahout.cf.taste.impl.neighborhood.NearestNUserNeighborhood` class, and it takes three parameters: the number of nearest neighbors to be considered, the UserSimilarity instance-similarity, the DataModel instance which is the model created in the previous steps as inputs:

   ```
   //creating user neighborhood
   UserNeighborhood neighborhood = new
     NearestNUserNeighborhood(10, similarity, model);
   ```

4. The next step is to generate a recommender model. This is achieved using the `org.apache.mahout.cf.taste.impl.recommender.GenericUserBasedRecommender class` instance. A `GenericUserBasedRecommender` instance-the recommender is created by passing the DataModel instance model, the UserNeighborhood instance neighborhood, the UserSimilarity instance similarity as inputs to the constructer while creating the recommender object.

   ```
   // creating recommender model
   UserBasedRecommender recommender = new
     GenericUserBasedRecommender(model, neighborhood, similarity);
   ```

5. Kudos! We have created our user-based recommender system using the Euclidean Distance similarity and the `NearestNNeighborhhood` method to create a recommender model. Now the next step would be to generate recommendations; for this, we call the `recommend()` method available in the recommender object, which takes `UserId` for which the recommendations and the number of recommendations have to be generated:

   ```
   //generating 3 recommendations for user 1068
   List<RecommendedItem> recommendations =
     recommender.recommend(1068, 3);
   ```

This step has generated three item recommendations to the `UserId` 1068 along with the strength of the preference.

In our case, we generated the following recommendations:

```
item:132613, value:1.2205102
item:132667, value:1.0
item:132584, value:0.98069793
```

Item-based collaborative filtering

Item-based recommenders recommend similar items to users by considering the similarity between items instead of the similarity of users, as shown in the previous section.

The following is the given java program to build item-based collaborative filtering. We have used `LogLikelihoodSimilarity` to calculate `ItemSimilarity`, and then we used the `GenericItemBasedRecommender` class to recommend items to users. In addition, we can see how to check similar items for a given item using the `mostSimilarItems` method present in `GenericItemBasedRecommender`:

```
package com.packpub.mahout.recommendationengines;

import java.io.File;
import java.io.IOException;
import java.util.List;

import org.apache.mahout.cf.taste.common.TasteException;
import org.apache.mahout.cf.taste.impl.model.file.FileDataModel;
import
org.apache.mahout.cf.taste.impl.recommender.GenericItemBasedRecommender;
import org.apache.mahout.cf.taste.impl.similarity.LogLikelihoodSimilarity;
import org.apache.mahout.cf.taste.model.DataModel;
import org.apache.mahout.cf.taste.recommender.RecommendedItem;
import org.apache.mahout.cf.taste.similarity.ItemSimilarity;


public class ItembasedRecommendations {

public static void main(String[] args) throws TasteException, IOException {
DataModel model = new FileDataModel(new File("data/recoDataset.csv"));
    ItemSimilarity similarity = new LogLikelihoodSimilarity(model);
    GenericItemBasedRecommender recommender = new
GenericItemBasedRecommender(model, similarity);
    System.out.println("*********Recommend Items to Users********");
    List<RecommendedItem> recommendations = recommender.recommend(1068, 3);
```

```
      for (RecommendedItem recommendation : recommendations) {
        System.out.println(recommendation);
      }
       System.out.println("*********Most Similar Items********");
      List<RecommendedItem> similarItems =
  recommender.mostSimilarItems(135104, 3);
      for (RecommendedItem similarItem : similarItems) {
        System.out.println(similarItem);
      }
  }

  }
```

Running the previous program will generate three items most similar to the input item, in our case, for the placeID `135104`, the most similar placeID attributes along with the strength of the similarity is shown in the following screenshot:

```
Problems  @ Javadoc  Declaration  Console
<terminated> ItembasedRecommendations [Java Application] C:\Program Files\Java\jre1.8.0_112\bin\javaw.exe (Nov 12, 2016, 12:03:12 AM)
log4j:WARN No appenders could be found for logger (org.apache.mahout.cf.taste.impl.model.file.FileDataModel).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
RecommendedItem[item:132667, value:0.96383345]
RecommendedItem[item:132732, value:0.9602005]
RecommendedItem[item:132733, value:0.9543598]
```

Let's look at each step of the preceding program in order to understand what's happening in the preceding implementation:

1. The first step, like in the previous section, is to create the DataModel instance using the `org.apache.mahout.cf.taste.impl.model.file.FileDataModel` class:

   ```
   //we create DataModel instance - model
   DataModel model = new FileDataModel(new
     File("data/recoDataset.csv"));
   ```

2. In this step, we create the ItemSimilarity instance, the similarity calculation between all users using the `org.apache.mahout.cf.taste.impl.similarity.LogLikelihoodSimilarity` class, which takes the `FileDataModel` instance created in the previous step as the constructor parameter:

   ```
   // creating LogLikelihood distance similarity between users
   ItemSimilarity similarity = new LogLikelihoodSimilarity
     (model);
   ```

3. The next step is to generate a recommender model. This is achieved using the `org.apache.mahout.cf.taste.impl.recommender.GenericItemBasedRecommender` class instance. A `GenericItemBasedRecommender` instance recommender is created passing the DataModel instance which is the model ItemSimilarity instance-similarity as inputs to the constructer while creating the recommender object.

```
// creating recommender model
GenericItemBasedRecommender recommender = new
  GenericItemBasedRecommender(model, similarity);
```

The choice of the similarity metric is left to you; it is set as per your requirement.

4. Kudos! We have created our item-based recommender system using the `LogLikelihood` similarity to create a recommender model. Now the next step would be to generate recommendations, and for this, we call the `recommend()` method available in the recommender object, which takes `UserId` for which the recommendations and the number of recommendations have to be generated:

```
//generating 3 recommendations for user 1068
List<RecommendedItem> recommendations =
  recommender.recommend(1068, 3);
```

This step has generated three item recommendations to the UserID 1068 along with the strength of the preference.

In our case, we generated the following recommendations:

```
item:132613, value:1.2205102
item:132667, value:1.0
item:132584, value:0.98069793
```

5. Imagine that we want to see items similar to a particular item; recommender interfaces, such as the `GenericItemBasedRecommender` class in our example, provide the `mostSimilarItems()` method, which takes `UserId`, the number of items to be displayed as inputs, and extracts `similarItems` for a given item:

```
List<RecommendedItem> similarItems =
  recommender.mostSimilarItems(135104, 3);
```

In our example, the three places most similar to `PlaceId` 135104 are shown as follows:

```
item:132667, value:0.96383345
item:132732, value:0.9602005
item:132733, value:0.9543598
```

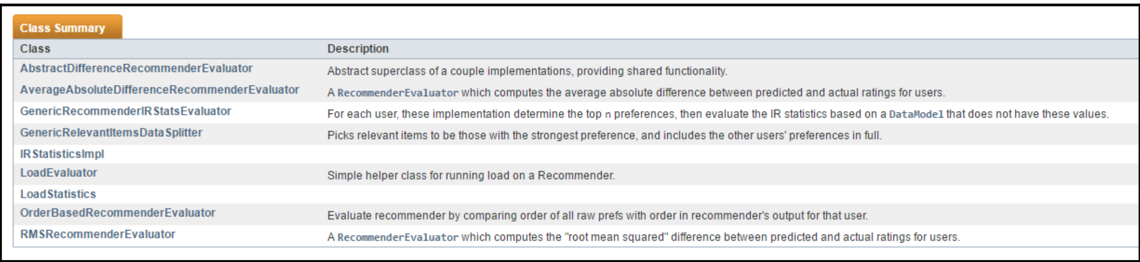
In the following section, let's evaluate the recommendations we have created so far.

Evaluating collaborative filtering

We have seen how to build recommendations using collaborative filtering approaches. But the key thing is to build efficient recommendations. Evaluating the accuracy of the recommender models – what we built – is a very crucial step in building recommendation engines. In this section, we will look at how to evaluate both user-based recommenders and item-based recommenders.

Mahout provides components that enable us to evaluate the accuracy of the recommendation models we have built so far. We can evaluate how closely our recommendation engine estimates the preferences against the actual preference values. We can instruct Mahout to use part of the original training data to set aside and use this test dataset in order to calculate the accuracy of the model.

We can use any of the following listed recommender evaluators provided by Mahout as per our requirement:



Class Summary

Class	Description
AbstractDifferenceRecommenderEvaluator	Abstract superclass of a couple implementations, providing shared functionality.
AverageAbsoluteDifferenceRecommenderEvaluator	A `RecommenderEvaluator` which computes the average absolute difference between predicted and actual ratings for users.
GenericRecommenderIRStatsEvaluator	For each user, these implementation determine the top `n` preferences, then evaluate the IR statistics based on a `DataModel` that does not have these values.
GenericRelevantItemsDataSplitter	Picks relevant items to be those with the strongest preference, and includes the other users' preferences in full.
IRStatisticsImpl	
LoadEvaluator	Simple helper class for running load on a Recommender.
LoadStatistics	
OrderBasedRecommenderEvaluator	Evaluate recommender by comparing order of all raw prefs with order in recommender's output for that user.
RMSRecommenderEvaluator	A `RecommenderEvaluator` which computes the "root mean squared" difference between predicted and actual ratings for users.

Recommender evaluation using Mahout usually requires two steps:

- Creating an instance of the `org.apache.mahout.cf.taste.impl.eval.RMSRecommenderEvaluator` class available from the preceding list, which will create the accuracy score
- Implementing the inner interface for `org.apache.mahout.cf.taste.eval.RecommenderBuilder` so as to create the recommender that the `RecommenderEvaluator` class instance from the previous step can use to produce the accuracy score

The listing shows the java implementation for user-based recommender model evaluation. For this exercise, we have used the root mean squared error evaluation technique.

Evaluating user-based recommenders

In this section, we shall see the code for evaluating the user-based recommendations we built in the previous section:

```
package com.packtpub.mahout.recommenders;

import java.io.File;
import java.io.IOException;

import org.apache.mahout.cf.taste.common.TasteException;
import org.apache.mahout.cf.taste.eval.RecommenderBuilder;
import org.apache.mahout.cf.taste.eval.RecommenderEvaluator;
import org.apache.mahout.cf.taste.impl.eval.RMSRecommenderEvaluator;
import org.apache.mahout.cf.taste.impl.model.file.FileDataModel;
import
org.apache.mahout.cf.taste.impl.neighborhood.NearestNUserNeighborhood;
import
org.apache.mahout.cf.taste.impl.recommender.GenericUserBasedRecommender;
import
org.apache.mahout.cf.taste.impl.similarity.EuclideanDistanceSimilarity;
import org.apache.mahout.cf.taste.model.DataModel;
import org.apache.mahout.cf.taste.neighborhood.UserNeighborhood;
import org.apache.mahout.cf.taste.recommender.Recommender;
import org.apache.mahout.cf.taste.similarity.UserSimilarity;

public class EvaluateUBCFRecommender {

public static void main(String[] args) throws IOException, TasteException {

DataModel model = new FileDataModel(new File("data/recoDataset.csv"));
```

```
RecommenderEvaluator evaluator = new RMSRecommenderEvaluator();
RecommenderBuilder builder = new RecommenderBuilder() {
public Recommender buildRecommender(DataModel model)
throws TasteException {
UserSimilarity similarity = new EuclideanDistanceSimilarity(model);
UserNeighborhood neighborhood =
new NearestNUserNeighborhood (10, similarity, model);
return
new GenericUserBasedRecommender (model, neighborhood, similarity);
}
};
double score = evaluator.evaluate(
builder, null, model, 0.8, 1.0);
System.out.println(score);
}

}
```

Executing the preceding program will give us the model accuracy: `0.692216091226208`.

Evaluating item-based recommenders

Below code snippet will be used in evaluating the item-based recommendations:

```
package com.packtpub.mahout.recommenders;

import java.io.File;
import java.io.IOException;

import org.apache.mahout.cf.taste.common.TasteException;
import org.apache.mahout.cf.taste.eval.RecommenderBuilder;
import org.apache.mahout.cf.taste.eval.RecommenderEvaluator;
import org.apache.mahout.cf.taste.impl.eval.RMSRecommenderEvaluator;
import org.apache.mahout.cf.taste.impl.model.file.FileDataModel;
import
org.apache.mahout.cf.taste.impl.recommender.GenericItemBasedRecommender;
import org.apache.mahout.cf.taste.impl.similarity.LogLikelihoodSimilarity;
import org.apache.mahout.cf.taste.model.DataModel;
import org.apache.mahout.cf.taste.recommender.Recommender;
import org.apache.mahout.cf.taste.similarity.ItemSimilarity;

public class EvaluateIBCFRecommender {

public static void main(String[] args) throws IOException, TasteException {

DataModel model = new FileDataModel(new File("data/recoDataset.csv"));
```

```
//RMS Recommender Evaluator
RecommenderEvaluator evaluator = new RMSRecommenderEvaluator();
RecommenderBuilder builder = new RecommenderBuilder() {
public Recommender buildRecommender(DataModel model)
throws TasteException {
ItemSimilarity similarity = new LogLikelihoodSimilarity(model);
return
new GenericItemBasedRecommender(model, similarity);
}
};
double score = evaluator.evaluate(builder, null, model, 0.7, 1.0);
System.out.println(score);

}

}
```

Executing the previous program will give us the model accuracy: `0.6041129199039021`.

Now let's look at this evaluation implementation step by step:

1. The first step is to create a DataModel instance model using the `org.apache.mahout.cf.taste.impl.model.file.FileDataModel` class:

   ```
   DataModel model = new FileDataModel(new
     File("data/recoDataset.csv"));
   ```

2. In this step, we create the `org.apache.mahout.cf.taste.impl.eval.RMSRecommenderEvaluator` instance evaluator, which will calculate the recommendation engine accuracy:

   ```
   // Recommendation engine model evaluator engine
   RecommenderEvaluator evaluator = new RMSRecommenderEvaluator();
   ```

3. In this step, we implement the `org.apache.mahout.cf.taste.eval.RecommenderBuilder` interface to create the recommender of our choice.

   Let's use the same recommender models we used for both user-based and item-based recommenders in the previous section:

   ```
   // User based recommenders
   public Recommender buildRecommender(DataModel model)
   throws TasteException {
   UserSimilarity similarity = new
     EuclideanDistanceSimilarity(model);
   UserNeighborhood neighborhood =
   ```

```
new NearestNUserNeighborhood (2, similarity, model);
return
new GenericUserBasedRecommender (model, neighborhood,
  similarity);
}
};

//Item based recommenders
public Recommender buildRecommender(DataModel model)
throws TasteException {
ItemSimilarity similarity = new LogLikelihoodSimilarity(model);
return
new GenericItemBasedRecommender(model, similarity);
}
};
```

4. Now we are ready to calculate the recommendation accuracy. For this, we use the `evaluate()` method from the evaluator instance. The `evaluate()` method does not accept a recommender instance–which we created in user-based/item-based recommenders directly–but it accepts RecommenderBuilder, created in step 3 of our examples/index, which can build the recommender to test the accuracy on top of a given DataModel.

   The `evaluate()` method takes four parameters: the recommender builder created in step 3, the DataModel object created in step 1, the DataModel builder object that we don't need for our example, the training percentage–in our case, we used 0.7 % as the training dataset and 0.3 as the test dataset–, evaluation percentage, the percentage of users to be used in evaluation.

   The `evaluate()` method returns the accuracy score of the model, which is how well the recommender-predicted preferences match the real values. Lower values indicate a better match, with 0 being the perfect match:

```
//generating 3 recommendations for user 1068
double score = evaluator.evaluate(builder, null, model, 0.7,
  1.0);
```

SVD recommenders

Similar to the item-based and user-based recommender systems explained earlier, we can also use model-based recommender implementations in Mahout, such as `SVDRecommender`, which uses matrix factorization methods to generate recommendations.

The steps are similar to previous implementations. Two important steps that need to be understood here are as follows:

- The `org.apache.mahout.cf.taste.impl.recommender.svd.ALSWRFactorizer` class, which factorizes the user rating matrix using *Alternating-Least-Squares with Weighted-λ-Regularization*. The `ALSWRFactorizer` class constructor takes parameters such as DataModel, the number of features, the regularization parameter, and the number of iterations as inputs. This `ALSWRFactorizer` class instance is passed as the input parameter to the recommender object: the `SVDRecommender` class.
- The `org.apache.mahout.cf.taste.impl.recommender.svd.SVDRecommender` class generates the recommendation model by taking `DataModel` and `ALSWRFactorizer` objects.

The rest of the other steps are very similar to what we saw in the previous examples:

The following code snippet shows how to build SVD recommender systems:

```
package com.packpub.mahout.recommendationengines;

import java.io.File;
import java.io.IOException;
import java.util.List;

import org.apache.mahout.cf.taste.common.TasteException;
import org.apache.mahout.cf.taste.impl.model.file.FileDataModel;
import org.apache.mahout.cf.taste.impl.recommender.svd.ALSWRFactorizer;
import org.apache.mahout.cf.taste.impl.recommender.svd.SVDRecommender;
import org.apache.mahout.cf.taste.model.DataModel;
import org.apache.mahout.cf.taste.recommender.RecommendedItem;


public class UserBasedSVDRecommender {

public static void main(String[] args) throws TasteException, IOException {
//MF recommender model
    DataModel model = new FileDataModel(new File("data/dataset.csv"));
```

```
    ALSWRFactorizer factorizer = new ALSWRFactorizer(model, 50, 0.065, 15);
    SVDRecommender recommender = new SVDRecommender(model, factorizer);
    List<RecommendedItem> recommendations = recommender.recommend(2, 3);
    for (RecommendedItem recommendation : recommendations) {
      System.out.println(recommendation);
    }

}

}
```

Distributed recommendations using Mahout

Up to now, we have seen how to build recommendation engines in the standalone mode. In most cases, the standalone implementations are very handy and they work quite efficiently in handling a million records provided we supply the dataset format, such as the userID, itemID, and preference triplet.

When the size of the data increases, the standalone mode might not be able to address the requirements. We need to look for ways to handle the enormous amount of data and be able to process the data to build recommendations. One approach is to port our standalone solution to the distributed mode, an example of which is Hadoop platforms.

The porting of the recommender solution to Hadoop is not straight forward, as the data will be distributed across the nodes. The memory-based models, such as neighbourhood recommenders, or model-based recommenders, such as Alternating Least Squares, requires the entire data to be available while generating the model, which will not be available on a distributed platform. Hence we need an entirely new design to build recommender systems.

Luckily, Mahout has removed the headaches in designing recommender implementations that can be distributed. These Mahout-distributed recommendation engine implementations are provided as jobs that internally run a series of map-reduce phases.

For example, Mahout-distributed recommendations using Alternating Least Squares consists of two jobs:

- A parallel matrix factorization job
- A recommendation job

The matrix factorization job takes the user-item-rating file as the input and creates the user latent matrix that is a user feature matrix and an item feature matrix.

The recommendation job uses the latent feature matrices created using the matrix factorization job and computes Top-N recommendations.

The two jobs are executed sequentially, the input data is read from HDFS, and final recommendations are written to HDFS.

In this section, we shall look at how to generate recommendations using the item-based recommendation engine and Alternating Least Squares methods using Hadoop. Let's begin.

ALS recommendation on Hadoop

To build the recommendation using ALS implementations, the following are the steps:

1. Load data to the Hadoop platform. The ALS implementation of Mahout expects the input to be a triplet: userID, itemID, and preference value (explicit rating/implicit rating).
2. Execute the ALS recommendation engine implementation job; this job will create user and item latent matrices by taking the input dataset from step 1.
3. Execute the recommender job that takes the user-item latent feature matrices created in step 2 and generate Top-N recommendations.

Let's execute all the steps one by one.

For the following exercise, we are using CDH 5 and Centos 6.
This is assuming `JAVA_HOME` is set and Mahout is installed properly.

1. Load data to the Hadoop platform as follows:

```
#create a directory to store the input data using mkdir command
[cloudera@quickstart ~]$ hadoop fs –mkdir mahout
```

Let's check whether we have created the directory properly using the `ls` command:

```
[cloudera@quickstart ~]$ hadoop fs -ls
Found 1 items
drwxr-xr-x   - cloudera cloudera          0 2016-11-14 18:31 mahout
```

Now let's load data to the HDFS using the `copyFromLocal` command:

```
hadoop fs -copyFromLocal /home/cloudera/datasets/u.data mahout
```

The input data is the MovieLens dataset that consists of one million rating data.

Let's verify that the data is loaded properly using the `ls` command:

```
[cloudera@quickstart ~]$ hadoop fs -ls mahout
Found 1 items
-rw-r--r--   1 cloudera cloudera    1979173 2016-11-14 18:32
mahout/u.data
```

Now that we have seen that the data is loaded properly, let's look at the first few records of the input data:

```
[cloudera@quickstart ~]$ hadoop fs -cat mahout/u.data |head
196 242 3 881250949
186 302 3 891717742
22 377 1 878887116
244 51 2 880606923
166 346 1 886397596
298 474 4 884182806
115 265 2 881171488
253 465 5 891628467
305 451 3 886324817
6 86 3 883603013
```

2. Create **User** and **Item latent** matrices. To create the latent feature matrices, we need to run the following commands from the command line:

```
$MAHOUT_HOME\bin\mahout parallelALS \
    --input mahout \
    --output output \
    --lambda 0.1 \
    --implicitFeedback false \
    --numFeatures 10 \
    --numIterations 1  \
    --tempDir tmp
```

   Let's look at each of the command parameters:

   - **$MAHOUT_HOME\bin\mahout**: This is the executable file that runs the underlying matrix factorization job.
   - **parallelALS**: This is the name of the algorithm to be applied on the input dataset. The `parallelALS` command invokes the underlying `ParallelALSFactorizationJob class object`, which is a map-reduce implementation of the factorization algorithms described in *Large-scale Parallel Collaborative Filtering for the Netflix Prize*.
   - **–input**: This is the HDFS input path of the input ratings data.
   - **–output**: This is the path where the output latent matrices for the user and item will be generated.
   - **–lambda**: This is the regularization parameter given in order to avoid overfitting.
   - **–alpha**: This is the confidence parameter used for implicit feedback only.
   - **–implicitFeatures**: This is the Boolean value to state whether the preference values are true or false. In our case, they are false.
   - **–numIterations**: This is the total number of times the model gets recomputed by applying the learnings from the previous model to the new model.
   - **–tempDir**: This is the path to the temporary directory where the intermediate results are written.

On executing the command we saw, three datasets are created in the *output* directory:

- **U**: This contains the user latent feature matrix
- **M**: The contains the item latent feature matrix
- **userRatings:** All the outputs are of a sequence file format.

3. Generate recommendations for all the users. This step takes the *output* results stored to HDFS from the previous step as the input, generates recommendations, and writes the final recommendations to the *recommendations output* directory on HDFS.

   The following command will invoke the `org.apache.mahout.cf.taste.hadoop.als.RecommenderJob` recommender job, which internally calls an `org.apache.mahout.cf.taste.hadoop.als.PredictionMapper` class to generate recommendations:

```
$MAHOUT_HOME\bin\mahout recommendfactorized \
       --input output/userRatings/  \
       --userFeatures output/U/ \
       --itemFeatures output/M/ \
       --numRecommendations 15 \
       --output recommendations/topNrecommendations \
       --maxRating 5
```

   Let's look at each of the parameters in detail:

   - **— input**: This is the HDFS path containing the list of the userID file to be used to generate recommendations in the sequence file format. In our example, the *output/userRatings* directory contains all the userID to be used to generate recommendations in the sequence file format; this file is the output of step 2.
   - **–userFeatures**: This is the HDFS path containing user latent features generated as the output in step 2.
   - **–itemFeatures**: This is the HDFS path containing item latent features generated as the output in step 2.
   - **–numRecommendations**: The number of recommendations to be generated per user.

- **–output recommendations**: This is the HDFS path where the final recommendations have to be generated.
- **–maxRating**: This is the maximum rating that the generated recommendations should contain.

Upon running the previous commands in the command line, recommendations are generated into the recommendations folder on HDFS, as follows:

```
[cloudera@quickstart ~]$ hadoop fs -cat
recommendations/topNrecommendations/part-m-00000 |head

1
[1536:5.0,1467:4.831182,1449:4.80844,814:4.742634,1599:4.68286
9,1398:4.649307,1629:4.570285,1639:4.562079,408:4.536842,1367:
4.528492,483:4.4752526,318:4.4236937,1500:4.4102707,1201:4.408
1335,603:4.3991466]

2
[1536:5.0,814:4.78269,1449:4.7134724,1398:4.6964526,1599:4.563
0975,1467:4.551129,169:4.437805,408:4.38913,114:4.381019,64:4.
3791966,1367:4.3718753,1064:4.3644257,483:4.350657,851:4.32865
8,318:4.325402]

3
[1536:3.842033,1642:3.749693,1467:3.6595325,1449:3.6565793,150
0:3.652883,1398:3.5153584,814:3.5086145,169:3.4643984,1651:3.4
516013,1636:3.4516013,1645:3.4516013,1650:3.4516013,114:3.4097
04,1639:3.3940363,1524:3.367357]

4
[1642:5.0,1651:5.0,1636:5.0,1650:5.0,1645:5.0,1201:5.0,1639:5.
0,1536:5.0,1449:5.0,1367:5.0,1500:5.0,483:5.0,113:5.0,1122:5.0
,1398:5.0]

5
[1536:4.328833,1449:4.1486154,814:4.09392,1599:4.0884914,1467:
4.0109262,1398:4.0078206,1500:3.8654287,1639:3.821516,1629:3.8

013735,1122:3.8007212,1463:3.7989178,318:3.7716885,483:3.76697
33,114:3.7642615,1642:3.7634466]

6
[1536:4.669815,1467:4.5344944,814:4.49133,1449:4.4251847,1599:
4.367457,1639:4.355604,1398:4.228149,1500:4.21864,1642:4.19020
7,1367:4.1796536,1463:4.175493,1452:4.0896783,1458:4.0896783,1
629:4.063499,851:4.0583687]

7
[1500:4.8419685,1122:4.636387,1536:4.611435,1449:4.604633,1467
:4.470685,1651:4.439256,1650:4.439256,1636:4.439256,1645:4.439
256,1642:4.408753,1189:4.399697,1201:4.3634996,1398:4.3018093,
169:4.25464,1450:4.2321644]

8
[1536:5.0,1467:5.0,1642:5.0,1639:5.0,814:4.9976716,1449:4.9946
73,1398:4.813181,851:4.7969217,119:4.7551465,169:4.7425375,146
3:4.7282143,1367:4.6779137,483:4.661983,1201:4.658822,1458:4.6
515884]

9
[1500:5.0,1645:4.7686267,1636:4.7686267,1650:4.7686267,1651:4.
7686267,1431:4.6634703,1491:4.651774,1558:4.6267195,1122:4.591
5747,1449:4.568913,1201:4.5492425,1175:4.515396,1643:4.506893,
1512:4.4682612,1155:4.4282117]

10
[1536:4.821219,1500:4.7425957,1449:4.724121,1467:4.6232443,112
2:4.5975246,1642:4.548042,1398:4.512008,1599:4.461757,1650:4.4
549794,1645:4.4549794,1651:4.4549794,1636:4.4549794,1189:4.440
5603,169:4.3742394,814:4.372402]
```

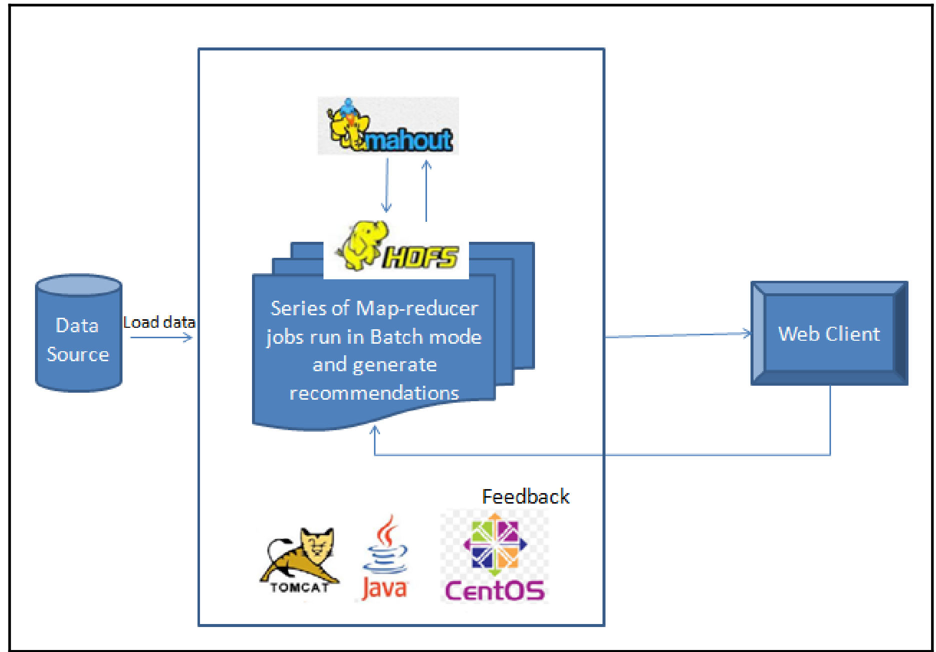
In the earlier result, we can see the first ten user recommendations in order. Each user vector contains itemID and the rating that the algorithm has predicted. While serving recommendations, we can just send recommendations as is.

Now you may get a question like this: what if I want to generate recommendations to specific users? Mahout supports such scenarios as well. Remember the input parameter in step 3? Just provide the HDFS path containing the userID which we need for recommendations. But make sure that the input path containing the userID are in a sequence file format.

The architecture for a scalable system

Taking the recommendation engine system to production is the same as any other system. The previous figure shows a very simple recommendation engine deployed on a production system:

- The production system is Centos 6 with Java 8 and the Apache Tomcat server installed on the system. CDH 5 and the Mahout 0.12 version is also installed on it so that the recommender jobs we have built so far can be deployed:



- The Java code we have written so far can be made as jar files and deployed on the production system. Schedule all the jobs at a regular interval as per our requirements.
- At a defined scheduled time, the recommender jobs start executing and the data is pulled from data sources, computes recommendation models, and generates recommendations.
- The data for the recommendation module will be read and written back to the HDFS file system.
- The frontend applications will read the final outputs from HDFS.

Summary

In this chapter, we saw how to build recommendations using Apache Mahout. We looked at how we can leverage Mahout for both the standalone and the distributed mode. We have written Java code for user-based, item-based, and SVD-based recommendation engines in the standalone mode and Alternating Least Squares recommendations in the distributed mode. We also saw how we can evaluate the recommendation engine models. In the final section, we explored a very basic system of how to take Mahout to production.

In the final chapter, we shall cover the future of recommendation engines, where the recommendation engines are heading, and the promising use cases to lookout for.

10
What Next - The Future of Recommendation Engines

Thank you for taking this wonderful journey with me so far. I hope you have got a fair idea of how to build recommendation engines using various technologies such as R, Python, Mahout, Spark, and Neo4j. We have covered recommendation engines such as neighborhood recommendations, model-based content recommendations, and context-aware, scalable, real-time, graph recommendations.

I would like to touch upon two things in the concluding chapter:

- Technological shift and motivational shift driving the future of recommendation engines
- Popular methodologies for improving the quality of recommendation engines

In the *Future of recommendation engines* section, I will summarize one of my talks about recommendation engines in a tech conference held in 2015. In the *good implementations* section, I will touch upon important methodologies to be followed while building recommendation engines.

With business organizations investing a lot in recommendation engines, researchers are freely exploring different aspects of recommendation engines and applying very advanced methods to improve their performance. It is very important for us to know the future of recommendation engines and the direction in which the research is moving so that we can apply these new techniques in our day-to-day work while building recommendation engines.

In the future of recommendation engines section, we will touch upon the technological and motivational shifts that are driving the future of recommendations. In the following section, we will learn about a few popular methodologies that every data scientist should know while building recommendation engines.

Future of recommendation engines

As we have reached the end of the book, I feel it is time to talk about the future of recommendation engines. Let us first have a small recap of what we have covered so far in the book:

- Recommendation engines in detail
- Data-mining techniques used in recommendation engines
- Collaborative filtering: similarity-based recommendations
- Model-based recommendations using R and Python
- Content-based recommender systems using R and Python
- Context-aware recommender systems using R and Python
- Scalable real-time recommender systems using Scala
- Scalable recommendation engines using Mahout
- Graph-based recommendation engines using Neo4j

Phases of recommendation engines

As explained in `Chapter 1`, *Recommendation Engines an Introduction*, if we look closely at the evolution of recommender systems, the recommendation engines have evolved in multiple directions; it is important to understand the directions in which recommendation engines are evolving to cope with futuristic situations.

We are heading into the third phase of evolution of recommendation engines, which are as follows:

- Phase 1: General recommendation engines
- Phase 2: Personalized recommendation engines
- Phase 3: Futuristic recommendation engines

Predominantly, recommendation engines focus on consumers. **Consumer** is the central objective for recommendation engines. Let's understand how these systems are evolving with respect to users. With the increase in the usage of the Internet, everyday decisions, ranging from which products to buy to which movies to watch to which restaurants to try, have led to more and more trust being put in the hands of these recommendation systems. These recommendation systems are changing the way we make up our minds, guiding us through a new digital reality, the evolution of which is bringing us closer to exactly we want, even if we ourselves don't know it yet.

Phase 1 – general recommendation engines

These recommendation engines are the earlier generation of recommendation engines. Collaborative filtering, user-based recommenders, and item-based recommenders fall into this phase of general recommendations.

As explained in `Chapter 3`, *Recommendation Engines Explained*, the collaborative filtering recommenders have become very popular, and are also very effective at recommending things to users. The following figure is symbolic of general recommendation engines:

Phase 2 – personalized recommender systems

As the information explosion started and more and more people started using the Web, leaving a good amount of digital footprints, such as search patterns, clicks, and browsing more and more, companies started looking into what, in the items or products, the user is interested in and which features of the items are making the user look for it. Companies started realizing that each person is unique and has unique tastes, and then they started catering to their need for personalized things; these are also called **content-based recommender systems**. In `Chapter 3`, *Recommendation Engines Explained*, we learned about content-based recommenders in detail. The following figure shows how personalized recommendations are given to the customer:

As systems moved to personalized systems–known as content-based recommender systems–more advanced techniques using machine learning, big data, and the cloud started calculating items more suited to the users. Methods such as matrix factorizations, **Singular Value Decomposition** (**SVD**), and regression analysis started to be employed as technology evolved.

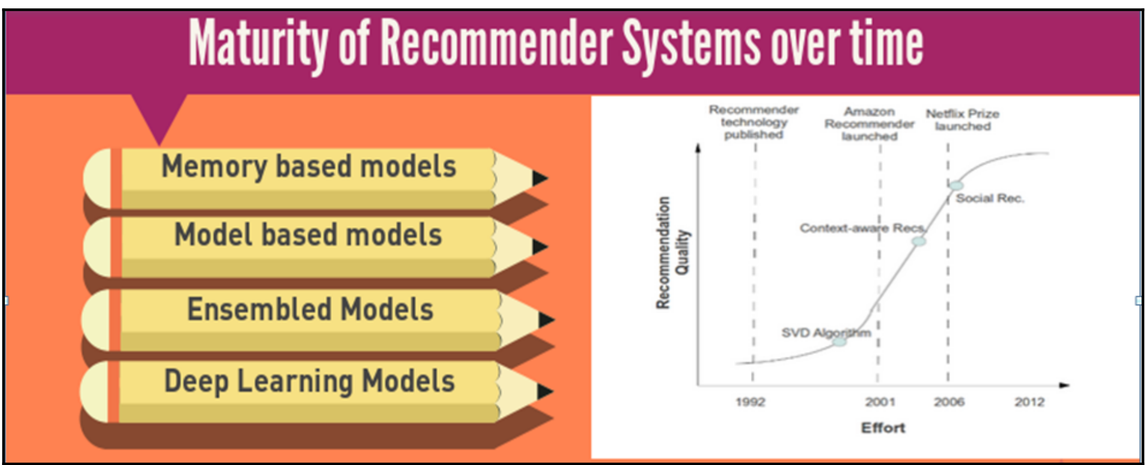
The aforementioned two methods have their own limitations to new data (the cold-start problem) and in narrowing down the information. To solve these problems, the ensemble or **hybrid recommender models** evolved, which are formed by combining one or more algorithms to achieve more accuracy.

From here, we are moving into **deep neural nets**, a very advanced level of neural-network algorithms stacked together in multiple layers, where feature engineering is being automated. One of the major difficult tasks in machine-learning models is to accurately engineer the features; so research is being done to apply deep-learning methods to recommendation engines.

For more information, refer to the following websites:

- `http://benanne.github.io/2014/08/05/spotify-cnns.html`
- `http://machinelearning.wustl.edu/mlpapers/paper_files/NIPS2013_5004.pdf`
- `https://www.quora.com/Has-there-been-any-work-on-using-deep-learning-for-recommendation-engines`

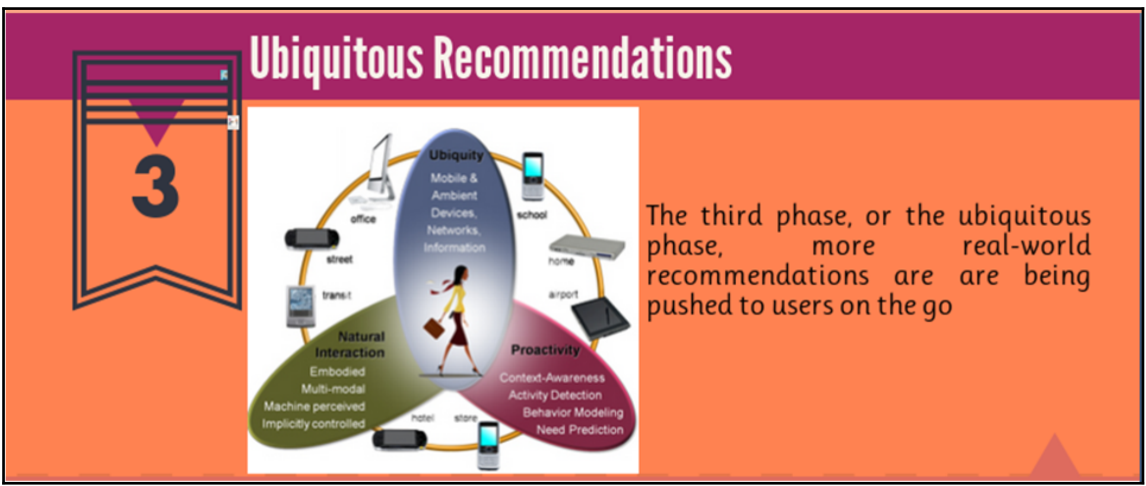
An example of implementation of deep learning in recommendation engines can be found as follows:



An S-curve for recommender systems would push the current state of the art nearly to the top of its bent.

Phase 3 – futuristic recommender systems

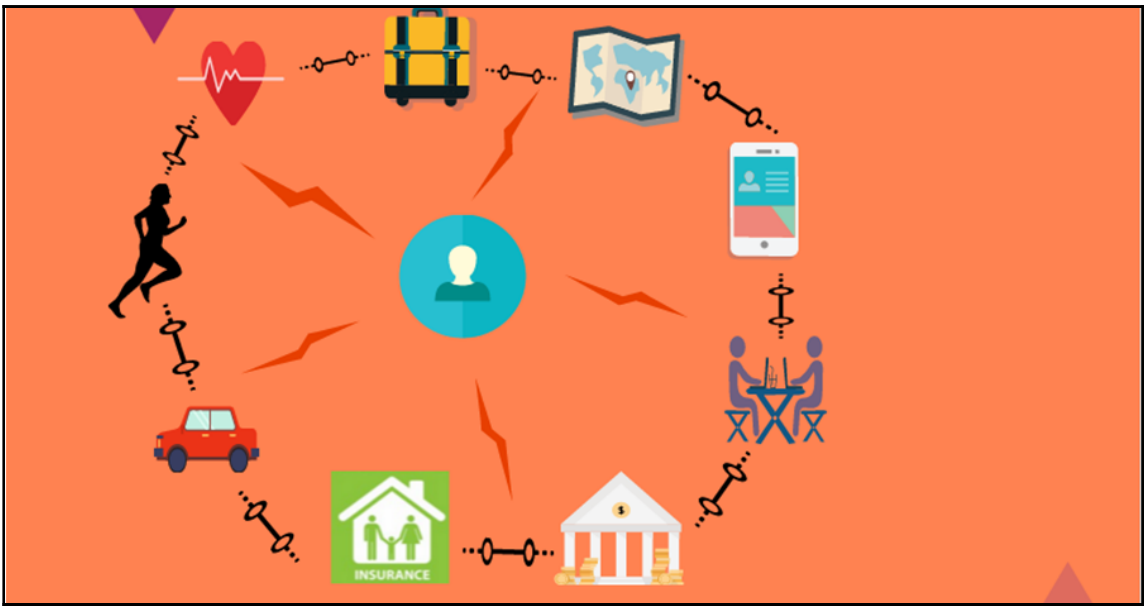
At this point, let me pause and take you to the futuristic view of recommender systems: we are moving into something called ubiquitous recommender systems. Ubiquitous recommenders start recommending things in real time based on your location, time, mood, sleep cycle, and energy output. The following figure depicts a futuristic system:



This means that where ever you go and whatever you do, the recommender systems will be watching you and will be recommending things on the go. Google, Facebook, and other major IT giants have pioneered these recommender systems and have nearly perfected them and started delivering these ubiquitous recommender systems. *Google Allo* from Google is one example of a ubiquitous recommender system:

Till now, we were getting recommendations based on some bucketing, users or items, but the future will be more tailor-made to users based on the digital footprints available. *Google Allo* is one of the best available apps, which continuously monitors you, and based on your activity on the app, it will definitely be recommending things on the go in the future. *Google Allo* is chat environment, which works as virtual assistant, assisting us with all our queries, which in turn will learn our preferences and interests over time and make smart suggestions on the go.

This is how real-time context-aware recommender systems also fall into this type of futuristic recommenders:



We are almost in a digitized world, where we are relying on the Internet for almost everything, be it related to banking, healthcare, driving cars, restaurants, travelling, personal fitness, and so on. In the near future, all companies would share information about users among themselves to create 360-degree user profiles using all the aforementioned digital footprints, and personalized, real-time, context-aware recommendations will be catered to with great accuracy. In the preceding figure, we can see how we are leaving our digital footprints and how it is possible for companies to share data with one another to generate recommendations tailored at individual levels.

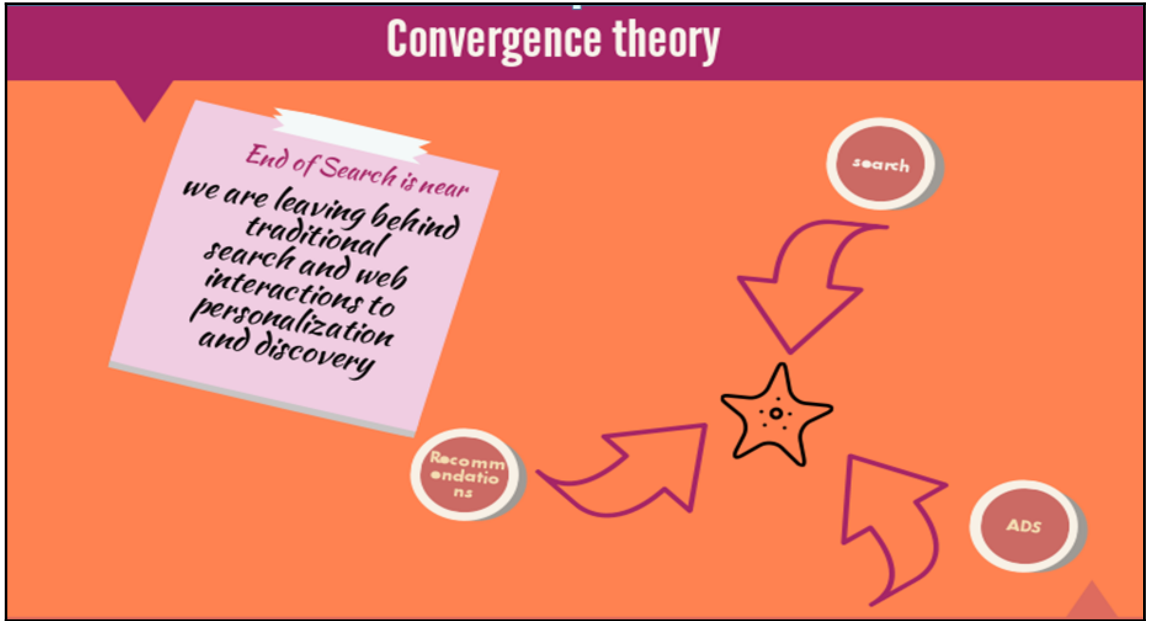
There have been shifts in motivation, which are driving the evolution of recommendation engines toward futuristic systems. A few of them are as follows:

- End of search
- Leaving the Web behind
- Emerging from the Web

Since the advent of recommendation engines, the primary focus is on customers, and even for the futuristic recommender systems, this remains the same, but the difference comes in with the increase in the number of digital footprints users are leaving. This large amount of usage of digital systems is leading customers to ask for more sophisticated solutions to make the access to information more personal than generic.

End of search

We are moving away from traditional ways of searching and web integration toward information and content discovery. Future search engines will be using a convergence theory of web *search/personalization/ads*, which will allow users to move into content discovery though recommendations rather than searches. The following figure shows convergence theory:

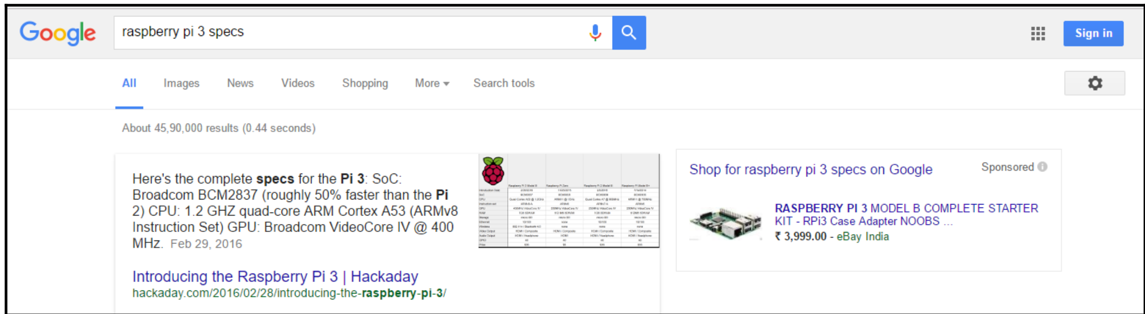


With more and more people relying on the Internet or search engines to find suitable products online, organizations are working together by sharing the data from users' online activity from different platforms, with the objective of minimizing the number of searches to find relevant information or content by generating accurate personalized recommendations based on recent activities.

I call this paradigm the **convergence theory,** where traditional search, ads, and recommendation engines are combined together to bring relevant content discovery to users at a personalized level by bringing an end to searching on the Internet.

For example, the Google search engine and YouTube have already started working toward this convergence theory to help users find relevant information on products and perform content discovery without searching explicitly.

Recently, I was looking for the Raspberry Pi 3, the latest in the series of credit-card-sized single-board computers, on an e-commerce site, to buy it for my personal work. A few days later, when I was searching for the specifications of the Raspberry Pi on Google Search, I noticed that Google had displayed ads related to the Raspberry Pi. Though alarming, it was more convenient for me because it had eliminated the task of going to the e-commerce site exclusively for purchasing. The search results are displayed in the following screenshot:



We can take YouTube as an example. The suggestions on YouTube are becoming more and more relevant, so much so that the number of times we search for required information is getting less and less; we follow the suggestions that are recommended to get more information on related topics.

In the future, we will see more and more applications making use of this convergence theory to minimize the number of searches by users, and also of recommendation engines for personalized content discovery.

Leaving the Web behind

Search engines and recommender systems are employed to enrich social experience as well as user experience by reaching out to the customers. More futuristic systems will continuously listen to their customers online and address their grievances at the earliest opportunity, even when their displeasure is being expressed on social sites. The following figure shows the motivation for futuristic recommendations – leaving the Web behind:

As an example, imagine you have expressed displeasure about your recent flight experience over Twitter by tagging the flight's official Twitter account; efforts will be made to listen to these concerns and reach out to customers personally to solve their problems. In addition to that, they may also please their customers with more personalized offers.

In the future, this approach may be followed by all organizations to listen to their customers' grievances or feedback on social platforms and reach out to customers with good recommendations or offers, which might be beneficial to both parties.

Emerging from the Web

New paradigms are evolving, such as Internet television replacing traditional television. Those times are gone when people had to wait to see their favorite programs at a predefined hour; instated people would wish to watch programs at a convenient time. Now times have changed; we are watching episodes at a time convenient to us on the Internet.

This change in the mindsets of people made business houses redesign their business models to increase their profits. For example, all the episodes of the Netflix series *House of Cards* were released together, breaking away from the traditional approach of one episode a week, which was a tremendous success and made other production houses follow suit.

The emergence of such business models is a result of the analysis of the viewing patterns of people.

Another interesting aspect of the *House of Cards* series is that the producers of *House of Cards*, Netflix, made use of big data analytics to analyze and understand the viewing patterns of its large user base, came up with a story comprising all the ingredients that viewers would love, and made a TV series. When it was released, the series was an instant hit.

This approach has made its way into other organizations to come up with more creations in order to improve customer experience.

As a result of the aforementioned shift in the motivation of recommendations, the context of recommendations is changing. Different people need different things at different times, with different people. The following figure depicts what was explained earlier:

A person going on a vacation with family might have a set of requirements that is contradictory to the set of requirements for when the person is going on a vacation with friends. Similarly, the same person may have different requirements at the same time in two different places. For example, a person travelling for business purposes to many countries might have different requirements, as per the local conditions. A person who is in a tropical country may need cotton dresses, and the same person may need woollen dresses when in cold countries. The difference in preferences is elaborated in the following figure, where we can see different people, in different countries, with different people, at different times, wearing different dresses:

The futuristic recommender systems will continuously and actively listen to their users to cater to their requirements on the go.

Next best actions

Another type of futuristic recommender system would be the systems that are sophisticated enough to predict your next move and make relevant suggestions without you asking explicitly.

TARS, from the movie *Interstellar*, could be a reality soon, which may suggest the next best actions a human should take by considering all the information surrounding the person.

Though TARS would be the most sophisticated system, humanoid robots, which are already a reality, might work as the first-generation next-best-action prediction agents.

Use cases to look out for

In this section, we will list a few promising use cases that might make you more interested in future of recommendation engines. Let's look at some good use cases for ubiquitous futuristic recommendations.

Smart homes

IoT and recommender systems together form a very powerful combination to bring about futuristic recommendation engines. A fully digitized smart home would be the best use case, where in the future, your refrigerator may suggest your monthly grocery list on your mobile while you are at work, as in following image. Similarly, IoT-enabled recommendation engines are something to be watched out for in the future:

Pic credits: http://walyou.com/smart-home-gadgets/

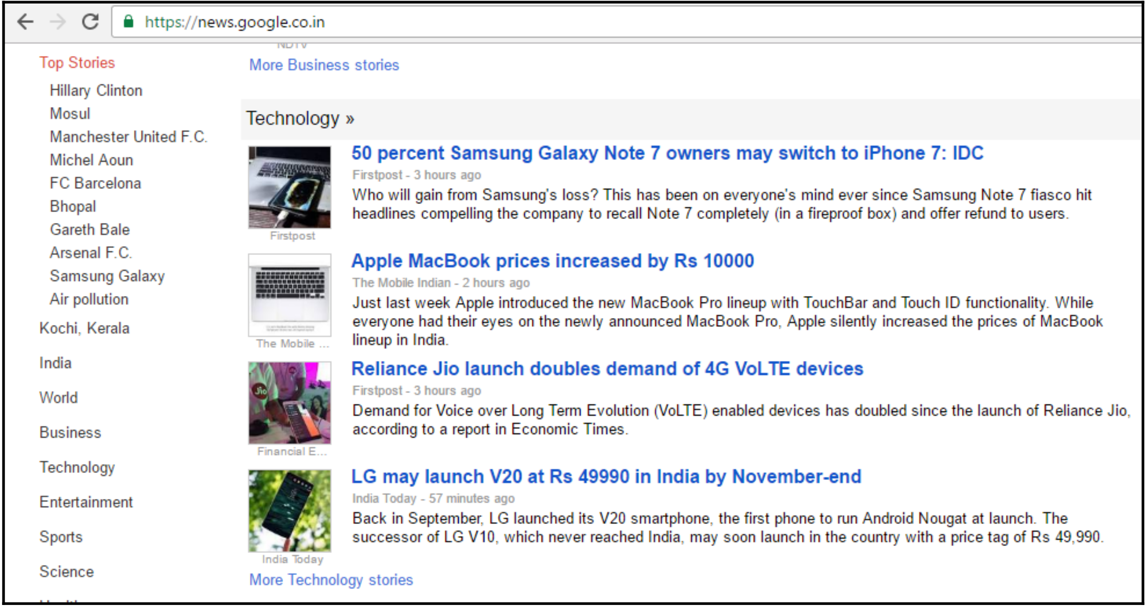
Healthcare recommender systems

Healthcare recommender systems form one of most exciting areas we have to watch out for. Researchers are focusing on how we can bring personalized healthcare to common people using advanced analytics. For instance, researchers at the University of Notre Dame have developed a system called **Collaborative Assessment and Recommendation Engine** (**CARE**), which uses a simple collaborative filtering that finds similar patients based on similarity of symptoms and generates probable risk profiles for individuals.

Consider the case of Proteus Digital Health Company, which uses an IoT-enabled device, an ingestible sensor, to track medical adherence. The device detects medication intake, tracks physiological data, and alerts the patient if he has skipped the tablet by mistake.

News as recommendations

If you observe Google News, you can see that a recommendation engine is working behind the scenes, continuously monitoring your click patterns, and combined with what is trending around you, a content-based personalized recommendation engine starts recommending news tailored for you:



Taking a leaf from this application's book, many new companies, such as Reddit and Rigg, among others, are using recommendation engines to suggest news items or articles as recommendations.

Popular methodologies

In earlier chapters, we have seen various recommendation engines. In this section, we touch upon a few popular methodologies, which are actively been employed in building recommendation engines for improving the robustness and relevance of the recommendations, such as:

- Serendipity
- Temporal aspects
- A/B testing
- Feedback mechanism

Serendipity

One of the drawbacks of recommendation engines is that the recommendation engine will push us to a corner where the items to be suggested or discovered will be entirely based on what we have looked for in the past or what we are currently looking for:

Credits: neighwhentheyrun

They work just the way horse blinkers work: they protect the horse from getting distracted from their path. The more interactions we have on the website, the narrower and closer to the users' profiles the recommendations tend to be. Is this wrong? No, absolutely not, but this is not how life works. If we turn back, most of the best discoveries of the past were made by chance. Surprises add spice to one's life. The amount of joy we get when we unexpectedly find something we need cannot be expressed in words. This feature is missing in the current paradigm of accuracy-oriented recommender systems.

By introducing serendipity and surprise to our recommender systems, we can reduce the aforementioned limitation. How do we introduce serendipity in the recommender systems?

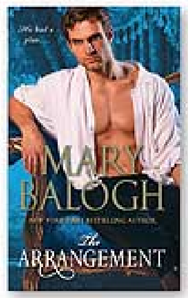
Google News, before generating personalized news-article recommendations, will combine the trending news within a region or country. This enables users to get more news, which is trending around them, and they will also become interested in such news.

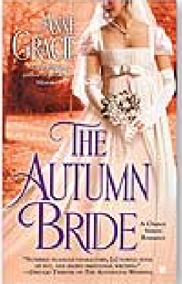
Temporal aspects of recommendation engines

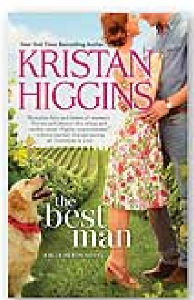
Consider the following scenario: a lady purchases items or looks for items related to pregnancy for almost 9 months. After the delivery of the baby, she will start looking for items related to a newborn. Our recommendation engines should be intelligent enough to capture this information, and as soon as the lady starts looking for items for newborns, the recommendation engine should remove the recommendations related to pregnancy as they are no longer relevant.

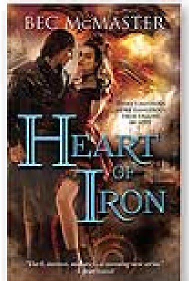
The following image shows how the non-inclusion of temporal aspects of recommendation engines still recommends romantic books to a recently-turned monk:

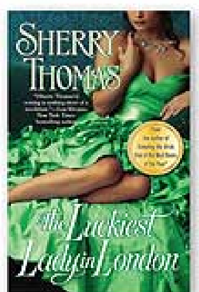
Book Recommendations based on previous interactions











Our choices are very time-specific; we may not like the things that we like today in the future. This aspect of time is often not considered while designing a recommendation engine. Our recommender systems capture every interaction of the user and accumulate large user preferences over a particular period. Since temporal information is inherent in user preferences, it is reasonable for a data scientist to exploit the temporal information to improve the relevance of the recommendation engine. A simple way of handling the temporal aspect is to give more weight to the most recent interactions, and less weight to old interactions while generating recommendations.

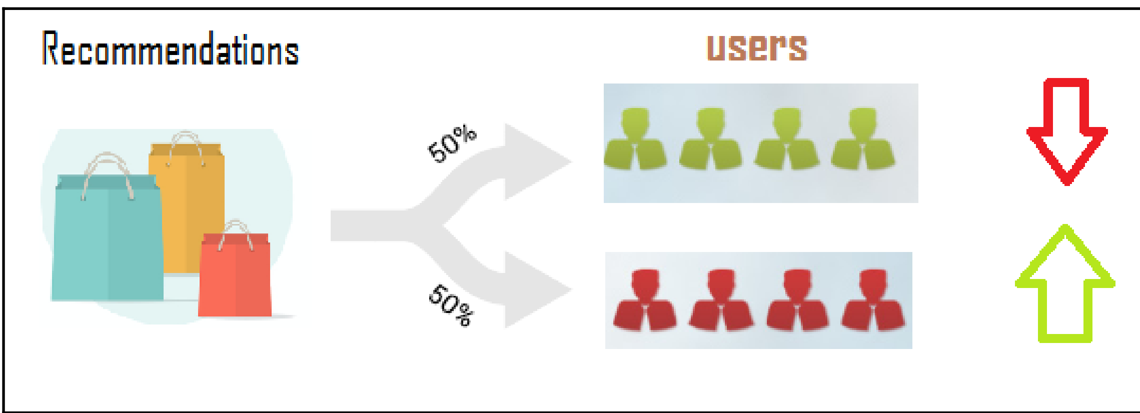
A/B testing

What is the most important thing for any data scientist? The accuracy of the machine-learning model we have built to solve the problem at hand. How do we ensure whether the model is right or not? We usually perform evaluation metrics, most likely the cross-validation approach and error/evaluation metrics while building the model, to check model accuracy before deploying the model to production. Though we apply best practices while building a model, such as **RMSE**, precision-recall and cross-validation approaches, which you learned about in previous chapters, are evaluated on historical data. Once the model is deployed in production, only then do we come to know how good the model's performance is. Usually, there isn't a single solution for a problem.

While designing a recommendation engine, we should always keep in mind the following things:

- A way to evaluate the model's performance at real time
- Always use multiple models for generating recommendations, and choose the model best suited for the user groups

The following shows how a simple A/B testing mechanism can be deployed in a production scenario:

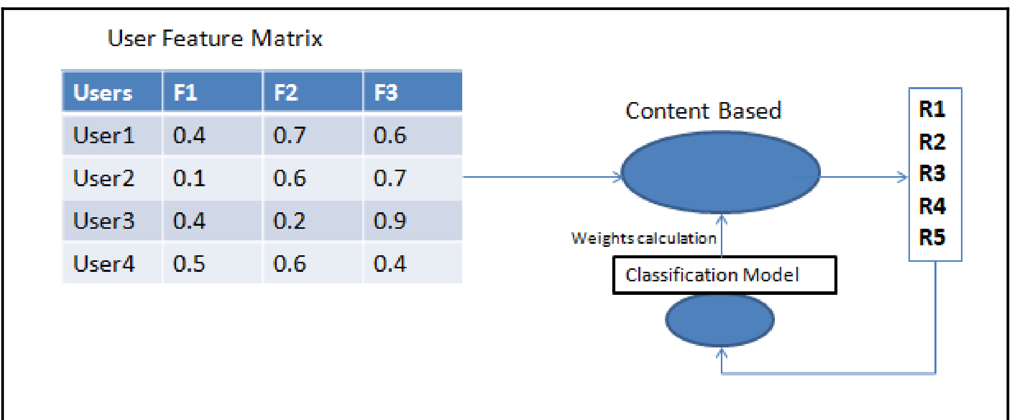


A/B testing comes to the rescue. In A/B testing, different sets of recommendations will be sent to different sets of users, and the performance of the recommendations will be evaluated at real time over a period. Though costly, A/B testing is an effective testing method to evaluate models in real time.

Feedback mechanism

Having spoken about A/B testing evaluating the performance of the recommender systems at real-time, it is very important to design the recommender systems to include a feedback mechanism. This is done to send back to the user interactions on the recommendations generated to fine-tune the model features included during model creation.

One simple approach to include the feedback mechanism is discussed as follows:


User Feature Matrix

Users	F1	F2	F3
User1	0.4	0.7	0.6
User2	0.1	0.6	0.7
User3	0.4	0.2	0.9
User4	0.5	0.6	0.4

Recall the content-based approach we used to generate recommendations. In the content-based approach, all the features are given equal weight. But we should be aware that not all features will be contributing equally to the recommendation model. To increase the accuracy of the model, we should enable a mechanism to calculate feature weights. Introduce a feedback mechanism to capture the user interactions on the recommendations, and then use this information to build a classification model to calculate the model feature weights.

Summary

In this chapter, we saw how recommendation engines are evolving and the motivations that are affecting the evolution of recommender systems, followed by a few potential use cases to watch out for. Finally, we touched upon some good methodologies, which should be considered before designing a recommender system. With this input, I'm sure that you are now equipped to confront the requirements of building future-ready recommendation engines which are self-learning, scalable, real-time, and futuristic. As mentioned in this chapter, deep learning can play a very important role in building more futuristic recommender systems.

Index

A

A/B testing 340
ALS recommendation
  implementing, on Hadoop 316, 318, 319, 320, 321
ALS() method
  maxIter 242
  rank 242
  reqParams 242
alternating least squares (ALS)
  about 81
  benefits 223
  used, for collaborative filtering 220, 221, 222, 223
Anaconda
  URL 155
Apache Mahout
  about 284
  components 295, 296
  setting up 285
  setting up, for distributed mode 293, 295
  setting, in standalone mode 285, 288, 293
  URL 286
  user-based collaborative recommendation engine 296
Apache Spark 2.0
  about 209
  architecture 210, 211
  benefits 215
  components 212
  ML Pipelines 218, 219, 220
  Resilient Distributed Datasets (RDD) 217
  setting up 215, 216
  Spark Core 212
  SparkSession 216, 217
  URL 215
Apache Spark
  about 208
  using, for scalable real-time recommender systems 19
AS keyword 260

B

boosting 98, 99
bootstrap aggregating (bagging) 97

C

classification models
  about 86
  decision trees 93, 94, 95
  ensemble methods 96
  KNN classification 88, 89
  linear classification 87, 88
  logistic regression 87
  support vector machines 90, 91, 92, 93
cluster 100
cluster analysis 100
clustering techniques
  about 100
  k-means clustering 101, 103
Collaborative Assessment and Recommendation Engine (CARE) 336
collaborative filtering recommender systems 11, 12
collaborative filtering, with Python
  building 154
  data source, downloading 155, 156
  data, exploring 156, 157, 158
  matrix representation, rating 158, 159
  packages, installing 155
  similarity, calculating 161, 162
  test sets, creating 160
  training sets, creating 160
  unknown ratings, predicting for active user 162,

163
collaborative filtering
ALS() constructor 225
fit() method 225
transform() method 225
with alternating least square (ALS) 221, 222, 223
with cosine similarity 279, 281
with Euclidean distance 273, 276, 278, 279
command line
Neo4j, starting 264, 266
commons-math3 290
consumer 325
content-based recommender systems, using Python
dataset, using 186, 189
item profile, creating 185
item profile, generating 193
recommendation engine model, generating 186
top-N recommendation, generating 186
user activity, defining 189, 192
user profile, creating 185, 195, 199
content-based recommender systems
about 13, 50, 52, 53, 170
advantages 56
building 171
building, with python 184
building, with R 172, 175
disadvantages 56
item profile generation 52
Movie Lens dataset, using 175, 178, 180, 184
user profile, generating 54, 56
context-aware recommender systems, using R
context profile, creating 203, 205
context, defining 202, 203
context-aware recommendations, generating 205, 207
context-aware recommender systems
about 16, 17, 199
building 200
building, with R 201
convergence theory 331
cosine similarity
about 71, 72, 73, 74
used, for collaborative filtering 279, 281
cross-validation 114
Cypher query language
about 250
node, syntax 251
principles 251
relationship, syntax 251

D

decision trees 93, 94, 95
deep learning
references 327
deep neural nets 327
dimensionality reduction
about 103
principal component analysis (PCA) 104, 105, 106, 107, 108
Discretized Stream (DStream) 213
document 108
driver program 211

E

elbow method 103
ensemble methods
about 96
bagging 97
boosting 98, 99, 100
random forests 96, 97
Euclidean distance
about 70, 71
used, for collaborative filtering 273, 276, 278, 279
evaluation techniques
about 113, 114
cross-validation 114
regularization 115

F

Facebook social network graph
creating 252
data, loading from csv 259, 260
nodes, creating 253
properties, setting to relations 256, 258
relationships, creating 254
False Positive Rate (FPR) 141
feedback mechanism 341

FileDataModel interface 297
futuristic recommender systems, use cases
  about 335
  healthcare recommender systems 336
  News 336
  smart homes 335
futuristic recommender systems
  about 328, 330
  future actions 334
  paradigms, emerging from Web 333
  web search, ending 330
  Web, avoiding 332

G

general recommendation engines 325
graph databases
  discerning 246
  GraphDB 248
  labeled property graph 248
  Neo4j 250
graph
  about 245
  labels 248
  nodes 248
  properties 249
  relationships 248
GraphDB 248
GraphX 214
guava 290

H

Hadoop Distributed File System (HDFS) 215
Hadoop
  ALS recommendation, implementing 316, 318, 319, 320, 321
HEADERS keyword 260
healthcare recommender systems 336
hybrid recommender models 326
hybrid recommender systems
  about 14, 15, 63
  advantages 65
  cascade method 64
  feature combination method 64
  mixed method 64
  weighted method 63

I

IBCF recommender model
  building 142, 143, 144, 145
  parameter, tuning 151, 153, 154
item-based collaborative filtering 47, 49
item-based recommender model
  accuracy, evaluating with metrics 147
  accuracy, evaluating with plots 148, 149, 150
  building 141, 142
  evaluating 145, 146, 147
  IBCF recommender model, building 142, 143, 144, 145

J

Jaccard similarity 74, 75
Java Development Kit (JDK) 215
Java Runtime Environment (JRE) 215
Jester5K dataset
  description 123
  details 124, 125, 126
  exploring 123, 126
  format 124
  rating values, exploring 127
  usage 123

K

k-cross validation
  used, for evaluating recommendation model 135, 136, 137
k-means clustering
  about 101, 103
  cluster assignment step 101
  move centroid step 101
k-nearest neighbors
  item-based recommendations 165, 166
  model, evaluating 166, 167
  top-N nearest neighbors, determining 163, 164, 165
  training model 167
  used, for user-based collaborative filtering 163
KNN classification 88, 89

L

labeled property graph
  about 248
  properties 248
latent features 79
linear classification 87, 88
linear regression 84, 85
Linux
  Neo4j, installing 263
logistic regression 87

M

machine learning techniques
  about 84
  classification models 86
  linear regression 84, 85
Mahout 0.12.2
  features 284
Mahout recommendation engine
  ALS recommendation, implementing on Hadoop 316, 318, 319, 320, 321
  building 300
  collaborative filtering 309, 310
  dataset, implementing 300, 303
  implementing, in distributed mode 315, 316
  item-based collaborative filtering 306, 308, 309
  item-based recommenders, evaluating 311, 313
  SVD recommenders, implementing 314
  user-based collaborative filtering 303, 305, 306
  user-based recommenders, evaluating 310
mahout-math 290
mahout-mr 290
Mahout
  about 17
  using, for scalable recommender systems 17, 18
mathematic model techniques
  about 78
  alternating least squares (ALS) 81
  matrix factorization 78, 79
  singular value decomposition (SVD) 82, 83
matrix factorization 78, 79
mean absolute error (MAE) 116, 139
methodologies
  A/B testing 340
  about 337
  feedback mechanism 341
  serendipity 337
  temporal aspect 338, 339
ML Pipelines
  about 218, 219, 220
  DataFrame 218
  Estimators 219
  Parameters 219
  Pipeline 219
  Transformers 218
MLlib 213, 214
MLlib recommendation engine module 225
model based recommender system
  creating, with pyspark 223, 224
model selection, recommendation engine
  Cross-Validation 239
  CrossValidator class 239
  evaluator object, setting 243
  ParamMaps/parameters, setting 242
  Train-Validation Split 239, 240, 241
model-based recommender systems
  about 65
  advantages 67
  machine learning approach 66
  mathematical approach 66
  probabilistic approach 66
movie recommendation engine
  building 267
  collaborative filtering, with cosine similarity 279, 281
  collaborative filtering, with Euclidean distance 273, 276, 278, 279
  data, loading into Neo4j 268, 271
  recommendations, generating with Neo4j 272
movie-rating dataset
  URL 25
MovieLens dataset
  URL 175
MS Web Dataset
  URL 186
MySQLJDBCDataModel 297

N

nearest neighborhood-based recommendation engines
  about 42, 44
  advantages 49
  disadvantages 50
  item-based collaborative filtering 47, 49
  user-based collaborative filtering 44, 47
neighbourhood-based techniques
  about 69
  cosine similarity 71, 72, 73, 74
  Euclidean distance 70, 71
  Jaccard similarity 74, 75
  Pearson correlation coefficient 75, 76, 77
Neo4j
  about 20, 250
  Cypher query language 250
  downloading 263
  installing, on Linux 263
  installing, on Windows 261, 262
  movie data, loading 268, 271
  recommendations, generating 272
  setting up 264
  starting, from command line 264, 266
  URL 261, 263
News 336

O

ordinary least squares (OLS) 85

P

Pearson correlation coefficient 75, 76, 77
personalized recommender systems
  about 326
  building 170
  content-based recommender systems 170
  context-aware recommender systems 199
phases, recommendation engine
  about 324
  futuristic recommender systems 328
  general recommendation engines 325
  personalized recommender systems 326
precision 117, 118, 119
Preference object 297
PreferenceArray object 297
principal component analysis (PCA) 104, 105, 106, 107, 108
pyspark
  used, for creating model based recommender system 223, 224
Python
  used, for building content-based recommender systems 184
  used, for collaborative filtering 154

R

R
  used, for building content-based recommender systems 172, 175
  used, for building context-aware recommender systems 201
random forests 96, 97
recall 117, 118, 119
recommendation engine, implementation
  about 226
  approach 225, 226
  data, exploring 228, 229, 230
  data, loading 226, 227, 228
  rating value, predicting 234, 236
  recommendation engine, building 233
recommendation engine
  building 25
  data, formatting 26, 27, 28, 29
  data, loading 26, 27, 28, 29
  definition 7, 8, 9
  evolution 40, 41
  future enhancements 324
  hyperparameter, tuning 238
  model selection 238
  model, evaluating 237
  similarity, calculating between users 29, 30
  unknown ratings, predicting for users 30, 33, 34, 37
  user-based collaborative filtering 236, 237
recommender systems
  about 56
  advantages 62
  big data 10
  collaborative filtering recommender systems 11,

12
content-based recommender systems 13, 14
context, defining 58, 60
context-aware recommender systems 16, 17
disadvantages 63
evolution 17
hybrid recommender systems 14, 15
need for 10
post filtering approach 62
pre-filtering approach 61
scalable real-time recommender systems, Apache Spark used 18
scalable recommender systems, Mahout used 17, 18
types 11
recommenderlab package
about 120
datasets 123
installing, in RStudio 121
Jester5K dataset 123
used, for building user-based collaborative filtering 128
regularization
about 115
mean absolute error (MAE) 116
precision 117, 118, 119
recall 117, 118, 119
root mean square error (RMSE) 115, 116
Resilient Distributed Datasets (RDD)
about 211, 217
Actions operation 217
Transformations operation 217
root mean square error (RMSE) 115, 116
root mean squared error (RMSE) 139
RStudio
recommenderlab package, installing 121

S

scalable real-time recommender systems
Apache Spark, used 18, 19
scalable recommender systems
Mahout, used 17, 18
scalable system
architecture 321
serendipity 337
similarity-based recommender systems 43
singular value decomposition (SVD) 82, 83, 326
slf4j-log4j 290
smart homes 335
Spark Core
about 212
GraphX 214
MLlib 213, 214
Spark SQL 212, 213
Spark Streaming 213
Spark SQL 212, 213
Spark Streaming 213
spark.ml, parameters
alpha 223
implicitPrefs 223
maxIter 223
nonnegative 223
numBlocks 223
rank 223
regParam 223
SparkSession 216
stochastic gradient descent (SGD) 81
support vector machines 90, 91, 92, 93

T

temporal aspect
about 338
handling 339
term frequency 109, 110
term frequency inverse document frequency (tf-idf) 52, 110, 111, 113
True Positive Rate (TPR) 141
types, collaborative filtering recommender systems
item-based collaborative filtering 11
user-based collaborative filtering 11

U

ubiquitous recommender systems 328
user-based collaborative filtering
about 44, 47
building, with recommenderlab package 128
dataset, analyzing 133, 134, 135
evaluating 137, 138, 139, 140, 141
model, creating 129, 131
predicting, on test set 131, 132

recommendation model, evaluating with k-cross validation 135, 136, 137
test data, preparing 129
training data, preparing 129
with k-nearest neighbors 163
user-based collaborative recommendation engine, components
about 296
DataModel 297
Item Similarity 298
Recommender 299
User Similarity 298
UserNeighborhood 299

V

vector space models
about 108
term frequency 109, 110
term frequency inverse document frequency (tf-idf) 110, 111, 113

W

Windows
Neo4j, installing 261, 262
WITH keyword 260
worker nodes 211