Mahout in Action

Sean Owen
Robin Anil
Ted Dunning
Ellen Friedman

MEAP

MANNING

MEAP Edition
Manning Early Access Program

Table of Contents

1 Meet Mahout

This chapter covers:

- What Mahout is
- A glimpse of recommender engines, clustering, classification in the real world
- Setting up Mahout

As you may have guessed from the title, this book is about putting a particular tool, Mahout, to effective use in real life. And what is Mahout?

Mahout is an open source *machine learning* library from Apache. The algorithms it implements fall under the broad umbrella of "machine learning," or "collective intelligence." This can mean many things, but at the moment for Mahout it means primarily collaborative filtering / recommender engines, clustering, and classification.

It is *scalable*. Mahout aims to be the machine learning tool of choice when the data to be processed is very large, perhaps far too large for a single machine. In its current incarnation, these scalable implementations are written in Java, and some portions are built upon Apache's Hadoop distributed computation project.

It is a *Java library*. It does not provide a user interface, a pre-packaged server, or installer. It is a framework of tools intended to be used and adapted by developers.

1.1 Is Mahout for Me?

You may be wondering – is this a project and a book for me?

If you are seeking a textbook on machine learning, no. This book does not attempt to fully explain the theory and derivation of the various algorithms and techniques presented. Some familiarity with these machine learning techniques and related concepts like matrix and vector math is useful in reading this book, but not assumed.

If you are developing modern, intelligent applications, then the answer is yes. This book provides a practical rather than theoretical treatment of these techniques, along with complete examples and recipes for solutions. It develops some insights gleaned by experienced practitioners in the course of demonstrating how Mahout can be deployed to solve problems.

If you are a researcher in artificial intelligence, machine learning and related areas – yes. Chances are your biggest obstacle is translating new algorithms into practice. Mahout provides a fertile framework for testing and deploying new large-scale algorithms. This book is an express ticket to functioning machine learning systems on top of complex distributed computing frameworks.

If you are a leading a product team or startup that will leverage machine learning to create a competitive advantage, this book is also for you. Through real-world examples, it will plant ideas about the many ways these techniques may be deployed. It will also help your scrappy technical team jump directly to a cost-effective implementation that can handle volumes of data previously only realistic for organizations with large technology resources.

Finally, you may be wondering how to *say* "Mahout" – it should rhyme with "trout." It is a Hindi word that refers to an elephant driver, and to explain that one, here's a little history. Mahout began life in 2008 as a subproject of Apache's Lucene project, which provides the well-known open-source search engine of the same name. Lucene provides advanced implementations of search, text mining and information retrieval techniques. In the universe of Computer Science, these concepts are adjacent to machine learning techniques like clustering and, to an extent, classification. So, some of the work of the Lucene committers that fell more into these machine learning areas was spun off into its own subproject. Soon after, Mahout absorbed the "Taste" open-source collaborative filtering project.

As of April 2010, Mahout has become a top-level Apache project in its own right.

Much of Mahout's work has been to not only implement these algorithms conventionally, in an efficient and scalable way, but also to convert some of these algorithms to work at scale on top of Hadoop. Hadoop's mascot is an elephant, which at last explains the project name!

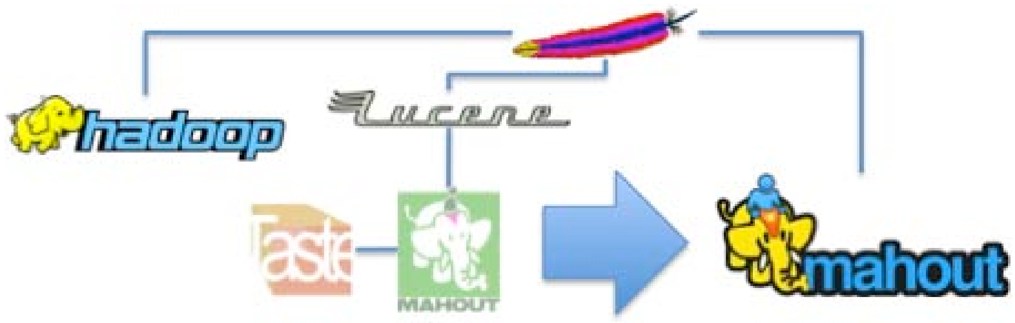


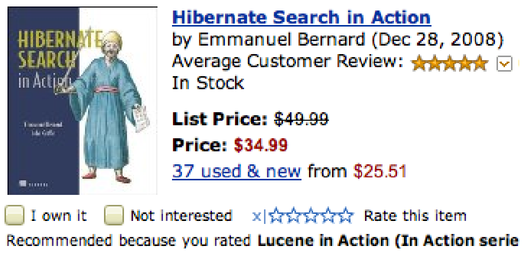
Figure 1.1 Mahout and its related projects

Mahout incubates a number of techniques and algorithms, many still in development or in an experimental phase. At this early stage in the project's life, three core themes are evident: collaborative filtering / **recommender engines**, clustering, and classification. Chances are that if you are reading this, you are already aware of the interesting potential of these three families of techniques. But just in case, read on.

1.2 Recommender Engines

Recommender engines are the most immediately recognizable machine learning technique in use today. We've all seen services or sites that attempt to recommend books or movies or articles based on our past actions. They try to infer tastes and preferences and identify unknown items that are of interest:

- Amazon.com is perhaps the most famous commerce site to deploy recommendations. Based on purchases and site activity, Amazon recommends books and other items likely to be of interest. See figure 1.1.
- Netflix similarly recommends DVDs that may be of interest, and famously offered a $1,000,000 prize to researchers that could improve the quality of their recommendations.
- Dating sites like Líbímseti (discussed later) can even recommend people to people.
- Social networking sites like Facebook use variants on recommender techniques to identify people most likely to be an as-yet-unconnected friend.



Hibernate Search in Action
by Emmanuel Bernard (Dec 28, 2008)
Average Customer Review: ★★★★★
In Stock

List Price: ~~$49.99~~
Price: $34.99
37 used & new from $25.51

I own it · Not interested · Rate this item
Recommended because you rated **Lucene in Action (In Action serie**

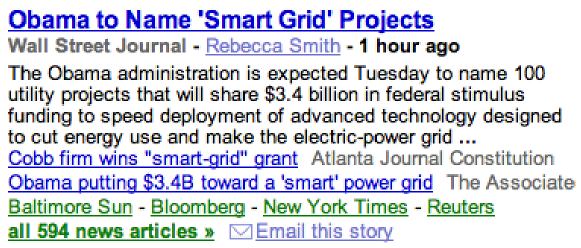
Figure 1.1 A recommendation from Amazon. Based on past purchase history and other activity of customers like the user, Amazon considers this to be something the user is interested in. It can even tell the user something similar that he or she has bought or liked that in part caused the recommendation.

1.3 Clustering

Clustering turns up in less apparent but equally well-known contexts. As its name implies, clustering techniques attempt to group a large number of things together into clusters that share some similarity. It is a way to discover hierarchy and order in a large or hard-to-understand data set, and in that way reveal interesting patterns or make the data set easier to comprehend.

Google News groups news articles according to their topic using clustering techniques in order to present news grouped by logical story, rather than a raw listing of all articles. Figure 1.2 below illustrates this.
Search engines like Clusty group search results for similar reasons

Consumers may be grouped into segments (clusters) using clustering techniques based on attributes like income, location, and buying habits.



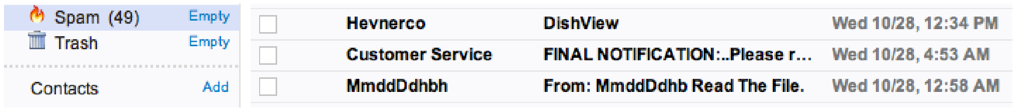
Obama to Name 'Smart Grid' Projects
Wall Street Journal - Rebecca Smith - **1 hour ago**
The Obama administration is expected Tuesday to name 100 utility projects that will share $3.4 billion in federal stimulus funding to speed deployment of advanced technology designed to cut energy use and make the electric-power grid ...
Cobb firm wins "smart-grid" grant Atlanta Journal Constitution
Obama putting $3.4B toward a 'smart' power grid The Associate
Baltimore Sun - Bloomberg - New York Times - Reuters
all 594 news articles » Email this story

Figure 1.2. A sample news grouping from Google News. A detailed snippet from one representative story is displayed, and links to a few other similar stories within the cluster for this topic are shown. Links to all the rest of the stories that clustered together in this topic are available too.

1.4 Classification

Classification techniques decide how much a thing is or isn't part of some type or category, or, does or doesn't have some attribute. Classification is likewise ubiquitous, though even more behind-the-scenes. Often these systems “learn” by reviewing many instances of items of the categories in question in order to deduce classification rules. This general idea finds many applications:

Yahoo! Mail decides whether incoming messages are spam, or not, based on prior emails and spam reports from users, as well as characteristics of the e-mail itself. A few messages classified as spam are shown in figure 1.3.
Picasa (http://picasa.google.com/) and other photo management applications can decide when a region of an image contains a human face.
Optical character recognition software classifies small regions of scanned text into individual characters by classifying the small areas as individual characters.
Apple’s Genius feature in iTunes reportedly uses classification to classify songs into potential playlists for users



☐	Hevnerco	DishView	Wed 10/28, 12:34 PM
☐	Customer Service	FINAL NOTIFICATION:..Please r...	Wed 10/28, 4:53 AM
☐	MmddDdhbh	From: MmddDdhb Read The File.	Wed 10/28, 12:58 AM

Figure 1.3 Spam messages as detected by Yahoo! Mail. Based on reports of email spam from users, plus other analysis, the system has learned certain attributes that usually identify spam. For example, messages mentioning “viagra” are frequently spam – as are those with clever misspellings like “v1agra”. The presence of such terms are an example of an attribute that a spam classifier can learn.

1.5 Scaling up

Each of these techniques works best when provided with a large amount of good input data. In some cases, these techniques must work not only on large amounts of input, but must produce results quickly. These factors quickly make scalability a major issue.

Picasa may have hosted over half a billion photos even three years ago, according to some crude estimates[1]. This implies millions of new photos per day that must be analyzed. The analysis of one photo by itself is not a large problem, though it is repeated millions of times. But, the learning phase can require information from each of the billions of photos simultaneously -- a computation on a scale that is not feasible for a single machine.

According to a similar analysis, Google News sees about a 3.5 million new news articles per day. Although this by itself is not a large amount, consider that these articles must be clustered, along with other recent articles, in minutes in order to become available in a timely manner.

The subset of rating data that Netflix published for the Netflix Prize contained 100 million ratings[2]. Since this was just the data released for contest purposes, presumably, the total amount of data that Netflix actually has and must process to create recommendations is many times larger!

These techniques are necessarily deployed in contexts where the amount of input is large – so large, that it is not feasible to process it all on one computer, even a powerful one. So, nobody implementing these techniques can ignore issues of scale. This is why Mahout makes scalability a top priority, and, why this book will focus, in a way that others don't, on dealing with large data sets effectively.

1.5.1 MapReduce and Hadoop

Some of Mahout makes use of Apache's Hadoop project, which includes an open-source, Java-based implementation of the MapReduce (http://labs.google.com/papers/mapreduce.html) distributed computing framework popularized and used internally at Google. MapReduce is a programming paradigm that at first sounds odd, or too simple to be powerful. The MapReduce paradigm applies to problems where the input is a set of key-value pairs. A "map" function turns these key-value pairs into other intermediate key-value pairs. A "reduce" function merges in some way all values for each intermediate key, to produce output. Actually, many problems can be framed as a MapReduce problem, or a series of them. And, the paradigm lends itself quite well to parallelization: all of the processing is independent, and so can be split across many machines. Rather than reproduce a full explanation of MapReduce here, we refer you to tutorials such as the one provided by Hadoop (http://hadoop.apache.org/common/docs/current/mapred_tutorial.html).

Hadoop implements the MapReduce paradigm, which is no small feat, even given how simple MapReduce sounds. It manages storage of the input, intermediate key-value pairs, and output; this data could potentially be massive, and, must be available to many worker machines, not just stored locally on one. It manages partitioning and data transfer between worker machines. It handles detection of and recovery from individual machine failure. Understanding how much work goes on behind the scenes will help prepare you for how relatively complex using Hadoop can seem. It's not just a library you add to your project. It's several components, each with libraries and (several) standalone server processes, which might be run on several machines. Operating processes based on Hadoop is not simple, but, investing in a scalable, distributed implementation can pay dividends later: because your data may grow exponentially to great sizes before you know it, this sort of scalable implementation is a way to future-proof your application.

[1] http://blogoscoped.com/archive/2007-03-12-n67.html
[2] http://archive.ics.uci.edu/ml/machine-learning-databases/netflix/

Later, this book will try to cut through some of that complexity to get you running on Hadoop fast, at which point you can explore the finer points and details of operating full clusters, and tuning the framework. Because this is complex framework that needs a great deal of computing power is becoming so popular, it's not surprising that cloud computing providers are beginning to offer Hadoop-related services. For example Amazon offers Elastic MapReduce (http://aws.amazon.com/elasticmapreduce/), a service which manages a Hadoop cluster, provides the computing power, and puts a friendlier interface on the otherwise complex task of operating and monitoring a large-scale job with Hadoop.

1.6 Setting up Mahout

Without further delay, let's look ahead to engaging Mahout in practice. You will need to assemble some tools before you can "play along at home" as we present some code in the coming chapters. We assume you are comfortable with Java development already.

Mahout and its associated frameworks are Java-based and therefore platform-independent, so you should be able to use it with any platform that can run a modern JVM. At times, we will need to give examples or instructions that will vary from platform to platform. In particular, command-line commands are somewhat different in a Windows shell than in a FreeBSD `tcsh` shell. We will use commands and syntax that work with `bash`, a shell found on most Unix-like platforms. This is the default on most Linux distributions, Mac OS X, many Unix variants, and Cygwin (a popular Unix-like environment for Windows). Windows users who wish to use the Windows shell are the most likely to be inconvenienced by this. Still, it should be simple to interpret and translate the listings given in this book to work for you.

1.6.1 Java and IDE

Java is likely already installed on your personal computer if you have done any Java development so far. Note that Mahout requires Java 6. If in doubt, open a terminal and type `java -version`. If the reported version does not begin with "1.6", you need to also install Java 6.

Windows and Linux users can find a Java 6 JVM from Sun at http://java.sun.com. Apple provides a Java 6 JVM for Mac OS X 10.5 and 10.6. If it does not appear that Java 6 is being used, open "Java Preferences" under `/Applications/Utilities`. This will allow you to select Java 6 as the default.

Most people will find it quite a bit easier to edit, compile and run the many examples we will see with the help an IDE; this is strongly recommended. Eclipse (http://www.eclipse.org) is the most popular, free Java IDE. Installing and configuring Eclipse is beyond the scope of this book, but you should spend some time becoming familiar with it before proceeding. IntelliJ IDEA (http://www.jetbrains.com/idea/index.html) is another powerful and popular IDE, with a free "community" version now available.

For example, IDEA can create a new project from an existing Maven model; by specifying the root directory of the Mahout source code upon creating a project, it will automatically configure and present the entire project in an organized manner. It's then possible, for example, drop the source code found throughout this book under the `core/src/`… source root, and run it from within IDE with one click -- the details of dependencies and compilation are managed automatically. This should prove far easier than attempting to compile and run manually.

1.6.2 Installing Maven

As with many Apache projects, Mahout's build and release system is built around Maven (http://maven.apache.org). Maven is a command-line tool that manages compiling code, packaging release, generating documentation, and publishing formal releases. Although it has some superficial resemblances to the also-popular Ant build tool, it is not the same. Ant is a flexible, lower-level scripting language, and Maven is a higher-level tool more purpose-built for release management.

Because Mahout uses Maven, you should install Maven yourself. Mac OS X users will be pleased to find that Maven should already be installed. If not, install Apple's Developer Tools. Type `mvn --version` on the command line. If you successfully see a version number, and the version is at least 2.2, you are ready to go. If not, you should install a local copy of Maven.

Users of Linux distributions with a decent package management system may be able to use it to quickly obtain a recent version of Maven. Otherwise, standard procedure would be to download a binary distribution, unpack it to a common location such as `/usr/local/maven`, then edit bash's configuration file, `~/.bashrc`, to include a line like `export PATH=/usr/local/maven/bin:$PATH`. This will ensure that the `mvn` command is always available.

If you are using an IDE like Eclipse or IntelliJ, it already includes Maven integration. Refer to its documentation to learn how to enable the Maven integration. This will make working with Mahout in an IDE much simpler, as the IDE can use the project's Maven configuration file (`pom.xml`) to instantly configure and import the project.

1.6.3 Installing Mahout

Mahout is still in development. This book was written to work with the 0.4 release of Mahout. This release and others may be downloaded by following instructions at http://lucene.apache.org/mahout/releases.html; the archive of source code may be unpacked anywhere that is convenient on your computer.

Because Mahout is changing frequently, and bug fixes and improvements are added regularly, it may be useful in practice to use a later release (or even the latest, unreleased code from Subversion. See http://lucene.apache.org/mahout/developer-resources.html). Future point releases should be backwards-compatible with the examples in this book.

Once you have obtained the source, either from Subversion or from a release archive, create a new project for Mahout in your IDE. This is IDE-specific; refer to its documentation for particulars of how this is accomplished. It will be easiest to use your IDE's Maven integration to simply import the Maven project from the `pom.xml` file in the root of the project source.

Once configured, you can easily create a new source directory within this project to hold sample code that will be introduced in upcoming chapters. With the project properly configured, you should be able to compile and run the code transparently with no further effort.

1.6.4 Installing Hadoop

For some activities later in this book, you will need your own local installation of Hadoop. You do not need a cluster of computers to run Hadoop. Setting up Hadoop is not difficult, but not trivial. Rather than repeat the procedures, we direct you to obtain a recent copy of Hadoop (version 0.20.x at the time of this writing) from http://hadoop.apache.org/common/releases.html, and then set up Hadoop for "pseudo-distributed" operation by following the quick start documentation currently found at http://hadoop.apache.org/common/docs/current/quickstart.html.

1.7 Summary

Mahout is a young, open-source, scalable machine learning library from Apache, and this book is a practical guide to using Mahout to solve real problems with machine learning techniques. In particular, we will soon explore recommender engines, clustering, and classification. If you're a researcher familiar with machine learning theory and looking for a practical how-to guide, or a developer looking to quickly learn best practices from practitioners, this book is for you.

These techniques are no longer merely theory: we've noted already well-known examples of recommender engines, clustering, and classification deployed in the real world: e-commerce, e-mail, videos, photos and more involve large-scale machine learning.

And, we've noted the vast amount of data sometimes employed with these techniques – scalability is a uniquely persistent concern in this area. We took a first look at MapReduce and Hadoop and how they power some of the scalability that Mahout provides.

Because this will be a hands-on, practical book, we've set up to begin working with Mahout right away. At this point, you should have assembled the tools you will need to work with Mahout and be ready for action. Because we promised that this book would be practical, let that wrap up the opening remarks now and get on to some real code with Mahout. Read on!

2 Introducing Recommenders

This chapter covers:

- A first look at a `Recommender` in action
- Evaluating accuracy of recommender engines
- Evaluating an engine’s precision and recall
- Evaluating a recommender on a real data set: GroupLens

Each day we form opinions about things we like, don't like, and don't even care about. It happens unconsciously. You hear a song on the radio and either notice it because it's catchy, or because it sounds awful – or maybe don't notice it at all. The same thing happens with t-shirts, salads, hairstyles, ski resorts, faces, and television shows.

Although people's tastes vary, they do follow patterns. People tend to like things that are similar to other things they like. Because I love bacon-lettuce-and-tomato sandwiches, you can guess I would enjoy a club sandwich, which is mostly the same sandwich, with turkey. Likewise, people tend to like things that similar people like. When a friend entered design school, she saw that just about every other design student owned a Macintosh computer – which was no surprise, as she was already a lifetime Mac user.

We can probably use these patterns predict these likes and dislikes. If we put a stranger in front of you and asked whether you thought she liked the third Lord of the Rings film, you might have nothing better than a guess. But, if she tells us she loved the first two films in the series, you'd be shocked if she didn't like the third as well. On the other hand, if she says she hated the films, or asks, “Lord of the what?” you'd rightly guess the third film is not on her favorites list.

Recommendation is all about predicting these patterns of taste, and using them to discover new and desirable things you didn’t already know about.

2.1 What is recommendation?

You picked up this book from the shelf for a reason. Maybe you saw it next to other books you know and find useful, and figure the bookstore has put it there since people who like those books tend to like this one too. Maybe you saw this book on the shelf of a coworker, who you know shares your interest in machine learning, or perhaps he recommended it to you directly.

In later chapters, we will explore some of the ways people make recommendations and discover new things -- and of course how these processes are implemented in software with Mahout. We've already mentioned a few strategies: to discover items we may like, we could look to what people with similar tastes seem to like. On the other hand, we could figure out what items are like the ones we already like, again by looking to others' apparent preferences. These describe the two broadest categories of recommender engine algorithms: "user-based" and "item-based" recommenders.

2.1.1 Collaborative filtering versus content-based recommendation

Strictly speaking, these are examples of "collaborative filtering" -- producing recommendations based on, and only based on, knowledge of users' relationships to items. These techniques require no knowledge of the properties of the items themselves. This is, in a way, an advantage. This recommender framework couldn't care less whether the "items" are books, theme parks, flowers, or even other people, since nothing about their attributes enters into any of the input.

There are other approaches based on the attributes of items, and are generally referred to as "content-based" recommendation techniques. For example, if a friend recommended this book to you because it's a Manning book, and the friend likes other Manning books, then the friend is engaging in something more like content-based recommendation. The thought is based on an attribute of the books: the publisher. The Mahout recommender framework does not directly implement these techniques, though it offers some ways to inject item attribute information into its computations. As such, it might technically be called a collaborative filtering framework.

There's nothing wrong with these techniques; on the contrary, they can work quite well. They are necessarily domain-specific approaches, and would be hard to meaningfully codify into a framework. To build an effective content-based book recommender, one would have to decide which attributes of a book -- page count, author, publisher, color, font -- are meaningful, and to what degree. None of this knowledge translates into any other domain; recommending books this way doesn't help in recommend pizza toppings.

For this reason, Mahout will not have much to say about this sort of recommendation. These ideas can be built into, and on top of, what Mahout provides; an example of this will follow in a later chapter, where we build a recommender for a dating site. Also later, after introducing the implementations that Mahout provides for collaborative filtering-based recommenders, we'll discuss their relation to content-based approaches in more detail.

2.1.2 Recommenders hit the mainstream

Most people have by now seen recommendations implemented in practice on sites like Amazon or Netflix: based on browsing and purchase history, the web site will produce a list of products that it believes may appeal to you. This sort of recommender engine has been around since the 1990s, but until recently has been the domain of fancy researchers with big computers. As these techniques have become more mainstream, demand for them has increased, and supply of open-source implementations has as well. This, along with increasingly accessible and cost-effective computing power, means that recommender engines are becoming more accessible and widely used.

In fact, recommender techniques aren't just for recommending things like DVDs to customers. The approach is general enough to estimate the strength of associations between many things. One could recommend customers to DVDs using the same techniques – estimate which customer might like a certain DVD the most. In a social network, a recommender could recommend people to people.

2.2 Running a first recommender engine

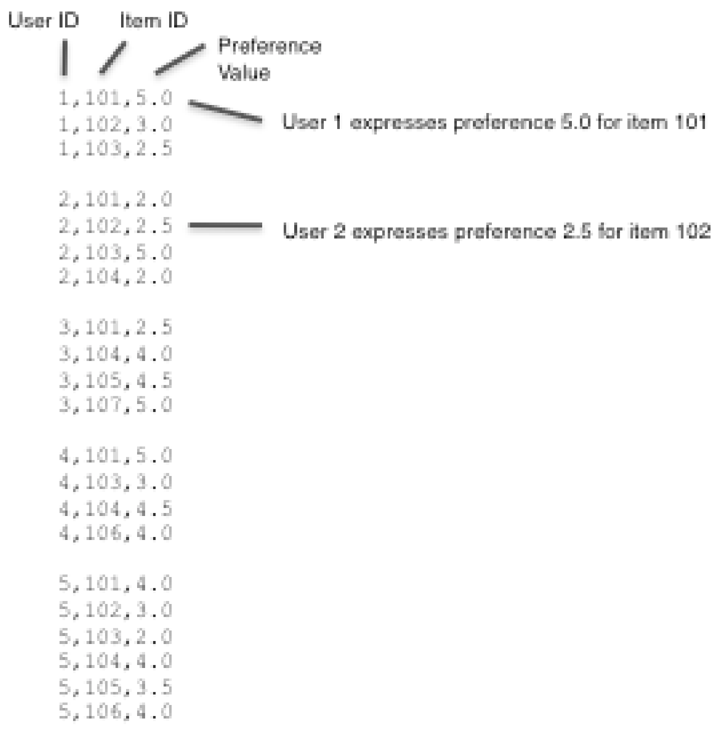
Mahout contains a recommender engine – several types, in fact, beginning with conventional user-based and item-based recommenders. It also includes implementations based on “slope-one” techniques, a new and efficient approach. You will also find experimental, preliminary implementations based on the singular value decomposition (SVD) and more. In the upcoming chapters, we will review the observations above in the context of Mahout and some real-world examples. We will look at how to represent data, tour the available recommender algorithms, evaluate the effectiveness of recommendations, tune and customize the recommender for a particular problem, and finally look at distributing the computation.

2.2.1 Creating the input

To explore recommendations in Mahout we will start with a trivial example. First, we need input to the recommender, data on which to base recommendations. This takes the form of “preferences” in Mahout-speak. Because the recommender engines that are most familiar involve recommending items to users, it will be most convenient to talk about preferences as associations from users to items – though as noted above, these users and items could be anything. A preference consists of a user ID and an item ID, and usually a number expressing the strength of the user's preference for the item. IDs in Mahout are always numbers, integers in fact. The preference value could be anything, as long as larger values mean stronger positive preferences. For instance, these values might be ratings on a scale of 1 to 5, where the user has assigned “1” to items she can't stand, and “5” to her favorites.

Create a text file containing data about users, cleverly named “1” to “5”, and their preferences for four books, which we will call “101” through “104”. In real-life, these might be customer IDs and product IDs from a company database; Mahout doesn’t literally require that the users and items be named with numbers! We'll write it down in simple comma-separated-value format. Copy the following into a file and save it as intro.csv:

Listing 2.1 Recommender input file intro.csv


User ID
Item ID
Preference Value
User 1 expresses preference 5.0 for item 101
User 2 expresses preference 2.5 for item 102

```
1,101,5.0
1,102,3.0
1,103,2.5

2,101,2.0
2,102,2.5
2,103,5.0
2,104,2.0

3,101,2.5
3,104,4.0
3,105,4.5
3,107,5.0

4,101,5.0
4,103,3.0
4,104,4.5
4,106,4.0

5,101,4.0
5,102,3.0
5,103,2.0
5,104,4.0
5,105,3.5
5,106,4.0
```

With some study, we notice some trends. Users 1 and 5 seem to have similar tastes. They both like book 101, like 102 a little less, and like 103 less still. The same goes for users 1 and 4, as they seem to like 101 and 103 identically (no word on how user 4 likes 102 though). On the other hand, users 1 and 2 have tastes that seem to run counter – 1 likes 101 while 2 doesn't, and 1 likes 103 while 2 is just the opposite. Users 1 and 3 don't overlap much – the only book both express a preference for is 101. See figure 2.1 to perhaps visualize the relations, both positive and negative, between users and items.

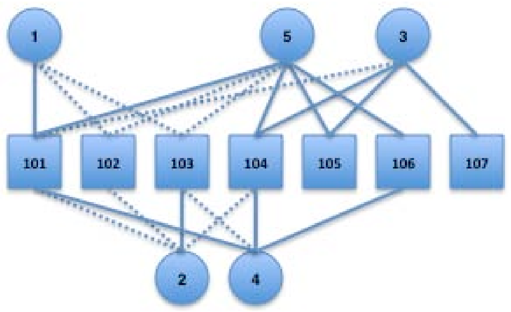


Figure 2.1 Relationships between users 1 to 5 and items 101 to 107. Dashed lines represent associations that seem negative -- the user does not seem to like the item much, but expresses a relationship to the item.

2.2.2 Creating a Recommender

So what book might we recommend to user 1? Not 101, 102 or 103 – he already knows about these books, apparently, and recommendation is about discovering new things. Intuition suggests that because users 4 and 5 seem similar to 1, we should recommend something that user 4 or user 5 likes. That leaves 104, 105 and 106 as possible recommendations. On the whole, 104 seems to be the most liked of these possibilities, judging by the preference values of 4.5 and 4.0 for item 104. Now, run the following code:

Listing 2.2 A simple user-based recommender program with Mahout

```
package mia.recommender.ch02;

import org.apache.mahout.cf.taste.impl.model.file.*;
import org.apache.mahout.cf.taste.impl.neighborhood.*;
import org.apache.mahout.cf.taste.impl.recommender.*;
import org.apache.mahout.cf.taste.impl.similarity.*;
import org.apache.mahout.cf.taste.model.*;
import org.apache.mahout.cf.taste.neighborhood.*;
import org.apache.mahout.cf.taste.recommender.*;
import org.apache.mahout.cf.taste.similarity.*;
import java.io.*;
import java.util.*;

class RecommenderIntro {

  public static void main(String[] args) throws Exception {

    DataModel model = new FileDataModel(new File("intro.csv")); A

    UserSimilarity similarity = new PearsonCorrelationSimilarity(model);
    UserNeighborhood neighborhood =
      new NearestNUserNeighborhood(2, similarity, model);

    Recommender recommender = new GenericUserBasedRecommender(
        model, neighborhood, similarity); B

    List<RecommendedItem> recommendations =
        recommender.recommend(1, 1); C

    for (RecommendedItem recommendation : recommendations) {
      System.out.println(recommendation);
    }

  }

}
```

A Load the data file
B Create the recommender engine
C For user 1, recommend 1 item

For brevity, through several more chapters of examples that follow, we will omit the imports, class declaration, and method declaration, and instead repeat only the program statements themselves. To help visualize the relationship between these basic components, see figure 2.2. Not all Mahout-based recommenders will look like this -- some will employ different components with different relationships. But this gives a sense of what’s going on in our example.

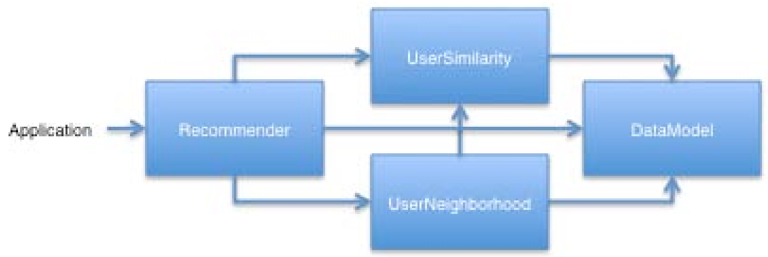


Figure 2.2 Simplified illustration of component interaction in a Mahout user-based recommender

While we will discuss each of these components in much more detail in the next two chapters, we can summarize the role of each component now. A `DataModel` implementation stores and provides access to all the preference, user and item data needed in the computation. A `UserSimiliarity` implementation provides some notion of how similar two users are; this could be based on one of many possible metrics or calculations. A `UserNeighborhood` implementation defines a notion of a group of users that are most similar to a given user. Finally, a `Recommender` implementation pulls all these components together to recommend items to users, and related functionality.

2.2.3 Analyzing the output

Compile and run this using your favorite IDE. The output of running the program in your terminal or IDE should be: RecommendedItem[item:104, value:4.257081]

We asked for one top recommendation, and got one. The recommender engine recommended book 104 to user 1. Further, it says that the recommender engine did so because it estimated user 1’s preference for book 104 to be about 4.3, and that was the highest among all the items eligible for recommendations.

That’s not bad. We didn't get 107, which was also recommendable, but only associated to a user with different tastes. We picked 104 over 106, and this makes sense when you note that 104 is a bit more highly rated overall. Further, we got a reasonable estimate of how much user 1 likes item 104 – something between the 4.0 and 4.5 that users 4 and 5 expressed.

The right answer isn't obvious from looking at the data, but the recommender engine made some decent sense of it and returned a defensible answer. If you got a pleasant tingle out of seeing this simple program give a useful and non-obvious result from a small pile of data, then the world of machine learning is for you!

For clear, small data sets, producing recommendations is as trivial as it appears above. In real life, data sets are huge, and they are noisy. For example, imagine a popular news site recommending news articles to readers. Preferences are inferred from article clicks. But, many of these “preferences” may be bogus – maybe a reader clicked an article but didn't like it, or, had clicked the wrong story. Perhaps many of the clicks occurred while not logged in, so can’t be associated to a user. And, imagine the size of the data set – perhaps billions of clicks in a month.

Producing the right recommendations from this data and producing them quickly are not trivial. Later we will present the tools Mahout provides to attack a range of such problems by way of case studies. They will show how standard approaches can produce poor recommendations or take a great deal of memory and CPU time, and, how to configure and customize Mahout to improve performance.

2.3 Evaluating a Recommender

A recommender engines is a tool, a means to answer the question, “what are the best recommendations for a user?” Before investigating the answers, we should investigate the question. What exactly is a good recommendation? And how will we know when a recommender is producing them? The remainder of this chapter pauses to explore evaluation of a recommender, because this is a tool that will be useful when we begin looking at specific recommender systems.

The best possible recommender would be a sort of psychic that could somehow know, before you do, exactly how much you would like every possible item that you've not yet seen or expressed any preference for. A recommender that could predict all your preferences exactly would merely present all other items ranked by your future preference and be done. These would be the best possible recommendations.

And indeed most recommender engines operate by trying to do just this, estimating ratings for some or all other items. So, one way of evaluating a recommender's recommendations is to evaluate the quality of its estimated preference values – that is, evaluating how closely the estimated preferences match the actual preferences.

2.3.1 Training data and scoring

Those “actual preferences” don't exist though. Nobody knows for sure how you'll like some new item in the future (including you). This can be simulated to a recommender engine by setting aside a small part of the real data set as test data. These test preferences are not present in the training data fed into a recommender engine under evaluation -- which is all data except the test data. Instead, the recommender is asked to estimate preference for the missing test data, and estimates are compared to the actual values.

From there, it is fairly simple to produce a kind of “score” for the recommender. For example we could compute the average difference between estimate and actual preference. With a score of this type, lower is better, because that would mean the estimates differed from the actual preference values by less. 0.0 would mean perfect estimation -- no difference at all between estimates and actual values.

Sometimes the root-mean-square of the differences is used: this is the square root of the average of the squares of the differences between actual and estimated preference values. Again, lower is better.

	Item 1	Item 2	Item 3
Actual	3.0	5.0	4.0
Estimate	3.5	2.0	5.0
Difference	0.5	3.0	1.0
Average Difference	= (0.5 + 3.0 + 1.0) / 3 = 1.5		
Root Mean Square	=√((0.52 + 3.02 + 1.02) / 3) = 1.8484		

Table 2.1 An illustration of the average difference, and root mean square calculation

Above, the table shows the difference between a set of actual and estimated preferences, and how they are translated into scores. Root-mean-square more heavily penalizes estimates that are way off, as with item 2 here, and that is considered desirable by some. For example, an estimate that's off by 2 whole stars is probably more than twice as “bad” as one off by just 1 star. Because the simple average of differences is perhaps more intuitive and easy to understand, we'll use it in upcoming examples.

2.3.2 Running RecommenderEvaluator

Let's revisit the example code and instead evaluate the simple recommender we created, on our simple data set:

Listing 2.3 Configuring and running an evaluation of a Recommender

```
RandomUtils.useTestSeed(); A
DataModel model = new FileDataModel(new File("intro.csv"));

RecommenderEvaluator evaluator =
  new AverageAbsoluteDifferenceRecommenderEvaluator();

RecommenderBuilder builder = new RecommenderBuilder() { B
  @Override
  public Recommender buildRecommender(DataModel model)
      throws TasteException {
    UserSimilarity similarity = new PearsonCorrelationSimilarity(model);
    UserNeighborhood neighborhood =
      new NearestNUserNeighborhood(2, similarity, model);
    return
      new GenericUserBasedRecommender(model, neighborhood, similarity);
  }
};

double score = evaluator.evaluate(
    builder, null, model, 0.7, 1.0); C
System.out.println(score);
```

A Used only in examples for repeatable result
B Builds the same Recommender as above
C Use 70% of data to train; test with other 30%

Most of the action happens in `evaluate()`. Inside, the `RecommenderEvaluator` handles splitting the data into a training and test set, builds a new training `DataModel` and `Recommender` to test, and compares its estimated preferences to the actual test data.

Note that we don't pass a `Recommender` to this method. This is because, inside, the method will need to build a `Recommender` around a newly created training `DataModel`. So we must provide an object that can build a `Recommender` from a `DataModel` – a `RecommenderBuilder`. Here, it builds the same implementation that we tried in the first chapter.

2.3.3 Assessing the result

This program prints the result of the evaluation: a score indicating how well the `Recommender` performed. In this case you should simply see: `1.0`. Even though a lot of randomness is used inside the evaluator to choose test data, the result should be consistent because of the call to `RandomUtils.useTestSeed()`, which forces the same random choices each time. This is only used in such examples, and unit tests, to guarantee repeatable results. Don't use it in your real code.

What this value means depends on the implementation we used – here, `AverageAbsoluteDifferenceRecommenderEvaluator`. A result of 1.0 from this implementation means that, on average, the recommender estimates a preference that deviates from the actual preference by 1.0.

A value of 1.0 is not great, on a scale of 1 to 5, but there is so little data here to begin with. Your results may differ as the data set is split randomly, and hence the training and test set may differ with each run.

This technique can be applied to any `Recommender` and `DataModel`. To use root-mean-square scoring, replace `AverageAbsoluteDifferenceRecommenderEvaluator` with the implementation `RMSRecommenderEvaluator`.

Also, the `null` parameter to `evaluate()` could instead be an instance of `DataModelBuilder`, which can be used to control how the training `DataModel` is created from training data. Normally the default is fine; it may not be if you are using a specialized implementation of `DataModel` in your deployment. A `DataModelBuilder` is how you would inject it into the evaluation process.

The `1.0` parameter at the end controls how much of the overall input data is used. Here it means "100%." This can be used to produce a quicker, if less accurate, evaluation by using only a little of a potentially huge data set. For example, `0.1` would mean 10% of the data is used and 90% is ignored. This is quite useful when rapidly testing small changes to a `Recommender`.

2.4 Evaluating precision and recall

We could also take a broader view of the recommender problem: we don't have to estimate preference values to produce recommendations. It's not always necessary to present estimated preference values to users. In many cases, all we want is an ordered list of recommendations, from best to worst. In fact, in some cases we don't care much about the exact ordering of the list – a set of a few good recommendations is fine.

Taking this more general view, we could also apply classic information retrieval metrics to evaluate recommenders: precision and recall. These terms are typically applied to things like search engines, which return some set of best results for a query out of many possible results.

A search engine should not return irrelevant results in the top results, although it should strive to return as many relevant results as possible. “Precision” is the proportion of top results that are relevant, for some definition of relevant. “Precision at 10” would be this proportion judged from the top 10 results. “Recall” is the proportion of all relevant results included in the top results. See figure 2.3 for a visualization of these ideas.

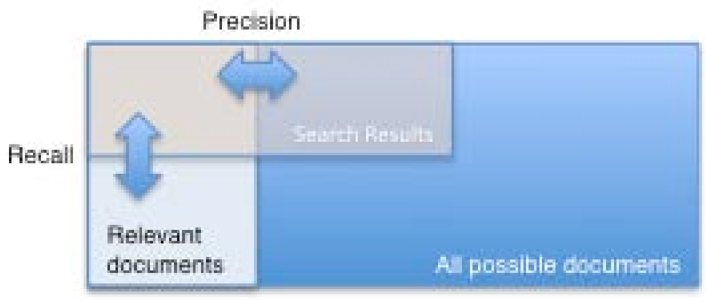


Figure 2.3 An illustration of precision and recall in the context of search results

These terms can easily be adapted to recommenders: precision is the proportion of top recommendations that are good recommendations, and recall is the proportion of good recommendations that appear in top recommendations. We’ll define “good” in the next section.

2.4.1 Running RecommenderIRStatsEvaluator

Again, Mahout provides a fairly simple way to compute these values for a `Recommender`:

Listing 2.4 Configuring and running a precision and recall evaluation

```
RandomUtils.useTestSeed();
   DataModel model = new FileDataModel(new File("intro.csv"));

   RecommenderIRStatsEvaluator evaluator =
     new GenericRecommenderIRStatsEvaluator();
   RecommenderBuilder recommenderBuilder = new RecommenderBuilder() {
     @Override
     public Recommender buildRecommender(DataModel model)
         throws TasteException {
       UserSimilarity similarity = new PearsonCorrelationSimilarity(model);
       UserNeighborhood neighborhood =
         new NearestNUserNeighborhood(2, similarity, model);
       return
         new GenericUserBasedRecommender(model, neighborhood, similarity);
     }
   };
   IRStatistics stats = evaluator.evaluate(
       recommenderBuilder, null, model, null, 2,
       GenericRecommenderIRStatsEvaluator.CHOOSE_THRESHOLD,
       1.0); A

   System.out.println(stats.getPrecision());
   System.out.println(stats.getRecall());
```

A Evaluate precision and recall at 2

Without the call to `RandomUtils.useTestSeed()`, the result you see would vary significantly due to random selection of training data and test data, and because the data set is so small here. But with the call, the result ought to be:

```
0.75
1.0
```

Precision at 2 is 0.75; on average about a three quarters of recommendations were "good." Recall at 2 is 1.0; all good recommendations are among those recommended.

But what exactly is a "good" recommendation here? Here, we actually asked the framework to decide. We didn't give it a definition. Intuitively, the most highly preferred items in the test set are the good recommendations, and the rest aren't.

Listing 2.5 User 5's preference in test data set

```
5,101,4.0
5,102,3.0
5,103,2.0
5,104,4.0
5,105,3.5
5,106,4.0
```

Look at user 5 in our simple data set again. Let's imagine we withheld as test data the preferences for items 101, 102 and 103. The preference values for these are 4.0, 3.0 and 2.0. With these values missing from the training data, we would hope that a recommender engine recommends 101 before 102, and 102 before 103, because we know this is the order in which user 5 prefers these items. But would it be a good idea to recommend 103? It's last on the list; user 5 doesn't seem to like it much. Book 102 is just average. Book 101 looks reasonable as its preference value is well above average. Maybe we'd say 101 is a good recommendation; 102 and 103 are valid, but not good recommendations.

And this is the thinking that the `RecommenderEvaluator` employs. When not given an explicit threshold that divides good recommendations from bad, the framework will pick a threshold, per user, that is equal to the user's average preference value μ plus one standard deviation σ:

threshold = $\mu + \sigma$

If you've forgotten your statistics, don't worry. This says we're taking items whose preference value is not merely a little more than average (μ), but above average by a significant amount (σ). In practice this means that about the 16% of items that are most highly preferred are considered "good" recommendations to make back to the user. The other arguments to this method are similar to those discussed before and are more fully documented in the project javadoc.

2.5 Evaluating the GroupLens data set

With these tools in hand, we will be able to discuss not only the speed, but also the quality of recommender engines that we create and modify. Although examples with large amounts real data are still a couple chapters away, we'll take a moment to quickly evaluate performance on a small data set.

2.5.1 Extracting the recommender input

GroupLens (http://grouplens.org/) is a research project that provides several data sets of different sizes, each derived from real users' ratings of movies. It is one of several large, real-world data sets available, and we will explore more of them in this book. From grouplens.org, locate and download the "100K data set", currently accessible at http://www.grouplens.org/node/73. Unarchive the file you download, and within, find the file called `ua.base`. This is a tab-delimited file with user IDs, item IDs, ratings (preference values), and some additional information.

Will this file work? Tabs, not commas, separate its field, and it includes an extra field of information at the end as well. Yes, the file will work with `FileDataModel` as-is. Return to the previous code in listing 2.3 where we built a `RecommenderEvaluator`, and, try passing in the location of `ua.base` instead of the small data file we constructed. Run it again. This time, evaluation should take a couple minutes, as it's now based on 100,000 preference values instead of a handful.

At the end, you should get a number around 0.9. That's not bad, though somehow being off by almost a whole point on a scale of 1 to 5 doesn't sound great. Perhaps the particular `Recommender` implementation we tried isn't quite the best for this kind of data?

2.5.2 Experimenting with other Recommenders

Let's test-drive a "slope-one" recommender on this data set, a simple algorithm that we will discuss in the upcoming chapter on recommender algorithms themselves. It's as easy as replacing the `RecommenderBuilder` with one that uses `org.apache.mahout.cf.taste.impl.recommender.slopeone.SlopeOneRecommeder`, like so:

Listing 2.6 Changing the evaluation program to run a SlopeOneRecommender

```
RecommenderBuilder recommenderBuilder = new RecommenderBuilder() {
  @Override
  public Recommender buildRecommender(DataModel model) throws TasteException {
    return new SlopeOneRecommender(model);
  }
};
```

Run the evaluation again. You should find it is both much quicker, and, produces an evaluation result around 0.748. That's a move in the right direction.

This is not to say slope-one is always better or faster. Each algorithm has its own characteristics and properties that can interact in hard-to-predict ways with a given data set. Slope-one happens to be quick to compute recommendations at runtime, but takes significant time to pre-compute its internal data structures before it can start, for example. The user-based recommender we tried first could be faster and more accurate on other data sets. We will explore the relative virtues of each algorithm in an upcoming chapter.

It highlights how important testing and evaluation on real data are – and how relatively painless it can be with Mahout. Soon we will be evaluating many recommenders.

2.6 Summary

In this chapter we introduced the idea of a recommender engine. We even created small input to a simple Mahout `Recommender`, ran it through a simple computation and explained the results.

We then took time to look at evaluating the quality of a recommender engine's output, before proceeding, because we will need to do this frequently in the coming chapters. This chapter covered evaluating the accuracy of a `Recommender`'s estimated preferences, as well as traditional precision and recall metrics as applied to recommendations. Finally we tried evaluating a real data set from GroupLens and observed how evaluations can be used to empirically discover improvements to a recommender engine.

Before studying recommender engines in detail, we need to spend some time with another foundational concept in Mahout in the next chapter: representation of data.

3 Representing Data

This chapter covers:

- How Mahout represents recommender data
- DataModel implementations and usage
- Data without preference values

The quality of recommendations is largely determined by the quantity and quality of data. “Garbage in, garbage out” was never more true than here. Likewise, recommender algorithms are data-intensive and runtime performance is greatly affected by quantity of data and its representation. This section explores key classes in Mahout for representing and accessing recommender-related data.

3.1 Representing Preferences

The input to a recommender engine is preference data -- who likes what, and how much. So, the input to Mahout recommenders is simply a set of user ID, item ID, preference value tuples – a large set, of course. Sometimes, even preference values are omitted.

3.1.1 The Preference object

A `Preference` is the most basic abstraction, representing a single user ID, item ID, and a preference value. One object represents one user's preference for one item. `Preference` is an interface, and the implementation one is most likely to use is `GenericPreference`. For example the following creates a representation of user 123’s preference value of 3.0 for item 456: `new GenericPreference(123, 456, 3.0f)`.

How is a set of `Preferences` represented? If you gave reasonable answers like `Collection<Preference>` or `Preference[]`, you'd be wrong in most cases in the Mahout APIs. Collections and arrays turn out to be quite inefficient for representing large numbers of `Preference` objects. If you’ve never investigated the overhead of an Object in Java, prepare to be shocked!

A single `GenericPreference` contains 20 bytes of useful data: an 8-byte user ID (Java `long`), 8-byte item ID (`long`), and 4-byte preference value (`float`). The object’s existence entails a startling amount of overhead: 28 bytes! The actual amount of overhead varies depending on the JVM's implementation; this figure was taken from Apple's 64-bit Java 6 VM for Mac OS X 10.6. This includes an 8-byte reference to the object, and, due to `Object` overhead and other alignment issues, another 20 bytes of space within the representation of the object itself. Hence a `GenericPreference` object already consumes 140% more memory than it needs to, just due to overhead.

What can be done? In the recommender algorithms, it is common to need a collection of all preferences associated to one user, or one item. In such a collection, the user ID, or item ID, will be identical for all `Preference` objects, which seems redundant.

3.1.2 PreferenceArray and implementations

Enter `PreferenceArray`, an interface whose implementations represent a collection of preferences with an array-like API. For example, `GenericUserPreferenceArray` represents all preferences associated to one user. Internally, it maintains a single user ID, an array of item IDs, and an array of preference values. The marginal memory required per preference in this representation is then only 12 bytes (one more 8-byte item ID and 4-byte preference value in an array). Compare this to the approximately 48 bytes required for a full `Preference` object. The four-fold memory savings alone justifies this special implementation, but it also provides a small performance win, as far fewer objects must be allocated and examined by the garbage collector. Compare figures 3.1 and 3.2 to understand how the savings is accomplished.

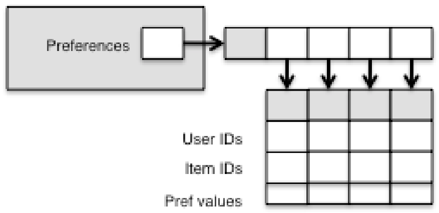


Figure 3.1 A less-efficient representation of preferences using an array of Preference objects. Gray areas represent, roughly, Object overhead. White areas are data, including Object references.

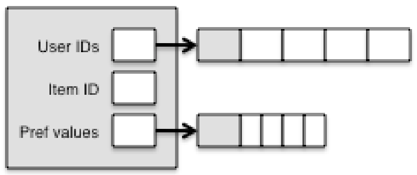


Figure 3.2 A more efficient representation using GenericUserPreferenceArray.

The code below shows typical construction and access of a `PreferenceArray`:

Listing 3.1 Setting preference values in a PreferenceArray

```
PreferenceArray user1Prefs = new GenericUserPreferenceArray(2);
user1Prefs.setUserID(0, 1L); A
```

```
user1Prefs.setItemID(0, 101L);
user1Prefs.setValue(0, 2.0f); B
user1Prefs.setItemID(1, 102L);
user1Prefs.setValue(1, 3.0f); C
Preference pref = user1Prefs.get(1); D
```

A Sets user ID for all preferences
B User 1 expresses preference 2.0 for item 101 now
C User 1 expresses preference 3.0 for 102
D Materializes a Preference for item 102

There exists, likewise, an implementation called `GenericItemPreferenceArray`, which encapsulates all preferences associated to an item, rather than user. Its purpose and usage are entirely analogous.

3.2 Speeding up collections

So, wonderful, Mahout has already reinvented an “array of Java objects,” you are thinking. Buckle up, because that’s not the end of it. Did we mention scale was important? Hopefully you are already persuaded that the amount of data we will face with these techniques is unusually huge, and may merit unusual responses.

The reduced memory requirement that `PreferenceArray` and its implementations bring is well worth its complexity. Cutting memory requirements by 75% isn’t just saving a couple megabytes -- it’s saving tens of gigabytes of memory at reasonable scale. That’s the difference between fitting and not fitting on your existing hardware, maybe. It’s the difference between having to invest in a lot more RAM and maybe a new 64-bit system and not having to. That’s a small but real energy savings. It matters.

3.2.1 FastByIDMap and FastIDSet

You won’t be surprised to hear that the Mahout recommenders make heavy use of typical data structures like maps and sets, but do not use the normal Java Collections implementations like `TreeSet` and `HashMap`. Instead, throughout the implementation and API you will find `FastByIDMap` and `FastIDSet`. These are something like a `Map` and `Set`, but specialized explicitly and only for what Mahout recommenders need. They reduce memory footprint rather than significantly increase in performance.

None of this should be construed as a criticism of the Java Collections framework. On the contrary, they are well designed for their purpose of being effective in a wide range of contexts. They cannot make many assumptions about usage patterns. Mahout’s needs are much more specific, and stronger assumptions about usage are available. The key differences are:

- Like `HashMap`, `FastByIDMap` is hash-based. It uses linear probing, rather than separate chaining, to handle hash collisions. This avoids the need for an additional `Map.Entry` object per entry; as we’ve discussed, `Objects` consume a surprising amount of memory.
- Keys and members are always long primitives in Mahout recommenders, not `Objects`. Using long keys saves memory and improves performance.
- The `Set` implementation is not implemented using a `Map` underneath
- `FastByIDMap` can act like a cache, as it has a notion of “maximum size”; beyond this size,

infrequently-used entries will be removed when new ones are added

The storage difference is significant: FastIDSet requires about 14 bytes per member on average, compared to 84 bytes for HashSet. FastByIDMap consumes about 28 bytes per entry, compared to again about 84 bytes per entry for HashMap. It goes to show that when one can make stronger assumptions about usage, significant improvements are possible – here, largely in memory requirements. Given the volume of data in question for recommender systems, these custom implementations more than justify themselves. So, where are these clever classes used?

3.3 In-memory DataModels

The abstraction that encapsulates recommender input data in Mahout is DataModel. Implementations of DataModel provide efficient access to data required by various recommender algorithms. For example, a DataModel can provide a count or list of all user IDs in the input data, or provide all preferences associated to an item, or a count of all users who express a preference for a set of item IDs. Here we will focus on some of the highlights; a more detailed account of DataModel's API can be found in the online javadoc documentation.

3.3.1 GenericDataModel

The simplest implementation available is an in-memory implementation, GenericDataModel. It is appropriate when you want to construct your data representation in memory, programmatically, rather than base it on an existing external source of data such as a file or relational database. It simply accepts preferences as inputs, in the form of a FastByIDMap mapping user IDs to PreferenceArrays with data for those users.

Listing 3.2 Defining input data programmatically with GenericDataModel

```
FastByIDMap<PreferenceArray> preferences =
  new FastByIDMap<PreferenceArray>();
PreferenceArray prefsForUser1 = new GenericUserPreferenceArray(10); A
prefsForUser1.setUserID(0, 1L);
prefsForUser1.setItemID(0, 101L); B
prefsForUser1.setValue(0, 3.0f); B
prefsForUser1.setItemID(1, 102L);
prefsForUser1.setValue(1, 4.5f);
… (8 more)

preferences.put(1L, prefsForUser1); C

DataModel model = new GenericDataModel(preferences); D
```

A Set up PreferenceArray for user 1
B Add the first of 10 preferences
C Attach user 1's preference to input
D Create the DataModel

How much memory does a GenericDataModel use? The number of preferences stored dominates memory consumption. Some empirical testing reveals that it consumes about 28 bytes of Java heap space per preference. This includes all data and other supporting data structures like indexes. You can try this if you like, as well: load a `GenericDataModel`, call `System.gc()` a few times, then compare the result of `Runtime.totalMemory()` and `Runtime.freeMemory()`. This is crude, but should give a reasonable estimate of how much memory the data is consuming.

3.3.1 File-based data

You will not typically use `GenericDataModel` directly. Instead, you will likely encounter it via `FileDataModel` – which reads data from a file and stores the resulting preference data in memory, in a `GenericDataModel`.

Just about any reasonable file will do – we already saw an example of such a file in the first section, where we produced a simple comma-separated-value file where each line contained one datum: user ID, item ID, preference value. Tab-separated files will work too. `zipped` and `gzipped` files will also work, if their names end in ".zip" or ".gz", respectively. It's a good idea to store this data in a compressed format, because it can be huge, and compresses well.

3.3.2 Refreshable components

While we're talking about loading data, let's talk about reloading data, and the `Refreshable` interface, which several components in the Mahout recommender-related classes implement. It exposes a single method, `refresh(Collection<Refreshable>)`. It simply requests that the component reload, recompute and otherwise refresh its own state, based on the latest input data available, after asking its dependencies to do likewise. For example, a `Recommender` will likely call `refresh()` on the `DataModel` on which it is based before recomputing its own internal indexes of the data. Cyclical dependencies and shared dependencies are managed intelligently, as illustrated in figure 3.3.

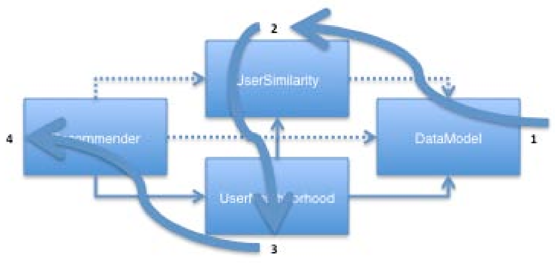


Figure 3.3 An illustration of dependencies in a simple user-based recommender system, and the order in which components refresh their data structures.

Note that `FileDataModel` will only reload data from the underlying file when asked to do so. It will not automatically detect updates or regularly attempt to reload the file's contents, for performance reasons. This is what the `refresh()` method is for. We probably don't want to just cause a `FileDataModel` to refresh, but also any objects that depends on its data. For this reason, you will almost surely call `refresh()` on a Recommender in practice:

Listing 3.3 Triggering refresh of a recommender system

```
DataModel dataModel = new FileDataModel(new File("input.csv");
Recommender recommender = new SlopeOneRecommender(dataModel);
...
recommender.refresh(null); A
```

A Refreshes the DataModel, then itself

Because scale is a pervasive theme of this book, here we should emphasize another useful feature of `FileDataModel`: "update files". Data changes, and usually the data that changes is only a tiny subset of all the data – maybe even just a few new data points, in comparison to a billion existing ones. Pushing around a brand new copy of a file containing a billion preferences just to push a few updates is wildly inefficient.

3.3.3 Update files

`FileDataModel` supports update files. These are just more data files which are read after the main data file, and overwrite any previously read data. New preferences are added; existing ones are updated. "Deletes" are handled by providing an empty preference value string.

For example, consider the following update file.

Listing 3.4 Sample update file

```
1,108,3.0
1,103,
```

This says, "update (or create) user 1's preference for item 108, and set the value to 3.0" and "remove user 1's preference for item 103".

These update files must simply exist in the same directory as the main data file, and their names must begin with the same prefix, up to the first period. If the main data file is `foo.txt.gz`, then update files might be named `foo.1.txt.gz` and `foo.2.txt.gz`. Yes, they may be compressed.

3.4 Database-based data

Sometimes data is just too large to fit into memory. Once the data set is several tens of millions of preferences, memory requirements grow to several gigabytes. This amount of memory may be unavailable in some contexts.

It is possible to store and access preference data from a relational database; Mahout supports this. Several classes in Mahout's recommender implementation will attempt to take advantage by pushing computations into the database for performance.

Note that running a recommender engine from data in a database will be much slower, by orders of magnitude, than using in-memory data representations. It's no fault of the database; properly tuned and configured, a modern database is excellent at indexing and retrieving information efficiently, but the overhead of retrieving, marshalling, serializing, transmitting and deserializing result sets is still much greater than the overhead of reading data from optimized in-memory data structures. This adds up quickly for recommender algorithms, which are data intensive. It may yet be desirable in cases where there is no choice, or, where the data set is not huge and reusing an existing table of data is desirable for integration purposes.

3.4.1 JDBC and MySQL

Preference data is accessed via JDBC, using implementations of `JDBCDataModel`. At the moment, the only concrete subclass of `JDBCDataModel` is one written for use with MySQL 5.x: `MySQLJDBCDataModel`. It may well work with older versions of MySQL, or even other databases, as it tries to use standard ANSI SQL where possible. It is not difficult to create variations, as needed, to use database-specific syntax and features. Here, we will explore the MySQL implementation here to illustrate.

Table 3.1 Illustration of default table schema for 'taste_preferences' in MySQL

user_id	**item_id**	**preference**
BIGINT NOT NULL INDEX PRIMARY KEY	BIGINT NOT NULL INDEX	FLOAT NOT NULL

By default, the implementation assumes that all preference data exists in a table called `taste_preferences`, with a column for user IDs named `user_id`, column for item IDs named `item_id`, and column for preference values named `preference`.

3.4.2 Configuring via JNDI

It also assumes that the database containing this table is accessible via a `DataSource` object registered to JNDI[3] name `jdbc/taste`. What is JNDI, you may be asking? If you are using a recommender engine in a web application, and are using a servlet container like Tomcat or Resin, then you are likely already using it indirectly. If you are configuring your database details through the container (such as through Tomcat's `server.xml` file) then you will find that typically makes this configuration available as a `DataSource` in JNDI. You can configure a database as `jdbc/taste` with details about the database that the `JDBCDataModel` ought to use. Here's a snippet suitable for use with Tomcat:

Listing 3.5 Configuring a JNDI DataSource in Tomcat

```
<Resource
  name="jdbc/taste"
  auth="Container"
  type="javax.sql.DataSource"
  username="user"
  password="password"
  driverClassName="com.mysql.jdbc.Driver"
  url="jdbc:mysql://localhost:3306/mydatabase"/>
```

These default names can be overridden to reflect your environment. You don't have to name your database and column exactly as above.

[3] Java Naming and Directory Interface; a key part of Sun's J2EE (Java 2 Enterprise Edition) specification

3.4.3 Configuring programmatically

You also don't have to use JNDI directly and can instead pass a DataSource in directly. Here's a full example of configuring a MySQLJDBCDataModel, including use of the MySQL Connector/J (http://www.mysql.com/products/connector/) driver and DataSource with customized table and column names:

Listing 3.6 Configuring a DataSource programmatically

```
MysqlDataSource dataSource = new MysqlDataSource();
dataSource.setServerName("my_user");
dataSource.setUser("my_password");
dataSource.setPassword("my_database_host");
JDBCDataModel dataModel = new MySQLJDBCDataModel(
  dataSource, "my_prefs_table", "my_user_column",
  "my_item_column", "my_pref_value_column");
```

This is all that's needed to use data in a database for recommendations. You've now got a DataModel compatible with all the recommender components! However, as the documentation for MySQLJDBCDataModel makes clear, producing the recommendations efficiently requires proper configuration of the database and the driver. In particular:

The user ID and item ID columns should be non-nullable, and must be indexed.

The primary key must be a composite of user ID and item ID.

Select data types for the columns that correspond to Java's long and float types. In MySQL, these are BIGINT and FLOAT.

Look to tuning the buffers and query caches (see javadoc)

When using MySQL's Connector/J driver, set driver parameters such as cachePreparedStatements to true. Again, see the javadoc for suggested values.

This certainly covers the basics of working with DataModels in Mahout's recommender engine framework. One significant variant on these implementations should be discussed: representing data when there are no preference values. This may sound strange, because it seems like preference values are the core of the input data required by a recommender engine. Sometimes ignoring some data helps.

3.5 Ignoring preference values

Less is more, they say. Sometimes this is true about the input to a recommender engine. More data is generally better -- if it is "good" data. Unfortunately, sometimes preference values are noisy, and simply forgetting the particular values is useful. At least, sometimes, it doesn't hurt.

We aren't talking about forgetting all associations between users and items, for then we would have no data at all. We're talking about ignoring the purported strength of the preference. For example, rather than consider what movies you all have seen and how you've rated them to recommend a new movie, we might do as well to simply consider what movies you have seen. Rather than know "user 1 expresses preference 4.5 for movie 103", we might try forgetting the 4.5 and taking, as input, data like "user 1 is associated to movie 103." Figure 3.4 attempts to illustrate the difference.

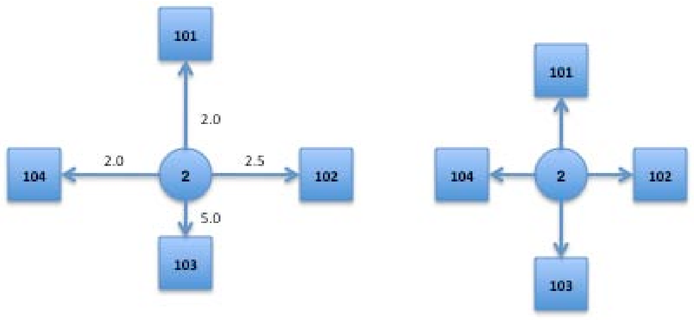


Figure 3.4 An illustration of user relationships to items with preference values (left) and “boolean data”, without preference values (right)

In Mahout-speak, we will call these “boolean preferences,” for lack of a better term, because an association can have one of two values: exists, or doesn’t exist. We do not mean that the data consists of “yes” and “no” preferences for items, that each datum expresses whether a user “likes” or “dislikes” an item. This would give three states for every possible user-item association: likes, dislikes, or nothing at all.

3.5.1 When to ignore values

Why would one do this, ignore preference values? Most commonly, this happens when preference values aren’t available to begin with. For example, imagine a news site recommending articles to users based on previously viewed articles. A “view” establishes some association between the user and item, but that’s about all that is available. It is not common for users to rate articles. It’s not even common for users to do anything more than view an article. All that’s known in this case is which articles the user is associated to, and little more.

This might be beneficial in a context where liking and not-liking an item are relatively similar states, at least when compared with having no association at all. Remember the example about the fellow who doesn’t like Rachmaninoff? There is a vast world of music out there, some of which he’s never even heard of (like Norwegian death metal). That he even knows Rachmaninoff enough to dislike it indicates an association to this composer, even a possible preference for things like it, that’s significant when considered in comparison to the vast world of things he doesn’t even know about. Although he might rate Rachmaninoff a “1” and Brahms a “5”, if pressed to do so, in reality these both communicate something similar. Forgetting the actual ratings, therefore, reflects that fact and may even make for better recommendations.

You may object that this is the user’s fault. Shouldn’t he think of Rachmaninoff as a “4”, because it’s stuff like Norwegian death metal that’s conceptually a “1” to him? Maybe so, but that’s life. This only underscores the fact that input is often problematic. You may also object that, although this reasoning stands up when recommending music taken from all genres, that we’d probably do worse by forgetting this data if we were just recommending from classical composers. This is true; a good solution for one domain does not always translate to others.

3.5.2 In-memory representations without preference values

Not having preference values dramatically simplifies the representation of preference data, and this enables better performance and significantly lower memory usage. As we saw, Mahout Preference objects store the preference value as a 4-byte float in Java. At least, not having preference values in memory ought to save 4 bytes per preference values. Indeed, repeating the same rough testing as before shows the overall memory consumption per preference drops by about 4 bytes to 24 bytes on average.

We get this value by testing the twin of `GenericalDataModel`, called `GenericBooleanPrefDataModel`. This is likewise an in-memory `DataModel` implementation, but one which internally does not store preference values. In fact it simply stores associations as `FastIDSets` -- for example, one for each user, to represent the item IDs that that user is associated to. No preference values are found.

Because it is also a `DataModel`, it is a drop-in replacement for `GenericDataModel`. Some methods of DataModel will be faster with this new implementation, such as `getItemIDsForUser()`, because the implementation already has this readily available. Some will be slower such as `getPreferencesFromUser()`, because the new implementation does not use `PreferenceArrays` and must materialize one to implement the method.

You may wonder what `getPreferenceValue()` returns, because there is no such thing to this implementation? It doesn't throw `UnsupportedOperationException`; it returns the same fixed, artificial value in all cases: 1.0. This is important to note, because components that rely on a preference value will still get one from this `DataModel`. These preference values are artificial and fixed, which can cause some subtle issues, as we will soon see.

Let's observe, by returning to the GroupLens example from the last chapter. Here is the same code snippet that we began with, but set up to use a `GenericBooleanPrefDataModel`:

Listing 3.7 Creating and evaluating with boolean data

```
DataModel model = new GenericBooleanPrefDataModel(
  new FileDataModel(new File("ua.base"))); A

RecommenderEvaluator evaluator =
  new AverageAbsoluteDifferenceRecommenderEvaluator();

  RecommenderBuilder builder = new RecommenderBuilder() {
    @Override
    public Recommender buildRecommender(DataModel model)
        throws TasteException {
      UserSimilarity similarity = new PearsonCorrelationSimilarity(model);
      UserNeighborhood neighborhood =
        new NearestNUserNeighborhood(10, similarity, model);
      return
        new GenericUserBasedRecommender(model, neighborhood, similarity);
    }
  };

  DataModelBuilder modelBuilder = new DataModelBuilder() {
    @Override
    public DataModel buildDataModel(
        FastByIDMap<PreferenceArray> trainingData) {
      return new GenericBooleanPrefDataModel(
        GenericBooleanPrefDataModel.toDataMap(trainingData)); B
    }
  };

  double score = evaluator.evaluate(
    recommenderBuilder, modelBuilder, model, 0.9, 1.0);
  System.out.println(score);
```

A Use GenericBooleanPrefDataModel, based on same data
B Build a GenericBooleanPrefDataModel here too

The twist here is the `DataModelBuilder`. This is the way we can control how the evaluation process will construct its `DataModel` for training data, rather than let it construct a simple `GenericDataModel`. `GenericBooleanPrefDataModel` takes its input in a slightly different way -- a bunch of `FastIDSets` rather than `PreferenceArrays` -- and the convenience method `toDataMap()` exists to translate between the two. Before proceeding to the next section, we suggest you try running this code.

3.5.3 Selecting compatible implementations

To be clear: the following section is largely about what not to do!

You may be surprised to see the evaluation result is `NaN`, or “not a number”. Here, it means that the evaluation couldn’t come up with any data on which to base a score at all! If you were to debug, you would find that the recommender is estimating all preferences to be `NaN` too. And if you dug deeper still, you would find that no neighborhoods of similar users can be found for any user, and this is because the `PearsonCorrelationSimilarity` metric is returning `NaN` as the similarity between every pair of users -- unknown!

This highlights a specific point about this Pearson correlation, which we will cover in the next chapter. It makes a more general point as well. The specific problem here is that we’re applying a similarity metric based on preference values, the Pearson correlation, in a situation where there aren’t any real preference values. The Pearson correlation between two data sets will be undefined if the two data sets are simply the same value, repeated[4]. And here, the `DataModel` pretends that all preference values are 1.0.

More generally, not every implementation will work well with every other, even though components are implementing a set of standard interfaces for interchangeability. The “incompatibility” here was clear from the evaluation; some other interactions are subtler.

3.5.4 Applying appropriate similarity metrics

We can fix the immediate problem by applying a more suitable similarity metric. `LogLikelihoodSimilarity` is one such implementation, because it is not based on actual preference values. We’ll discuss these similarity metrics later. Plug it in, in place of `PearsonCorrelationSimilarity`. The result is, at least, a number: 0.0. Wow, that means perfect prediction!

[4] The Pearson correlation is a ratio of the covariance of the two data sets, to their standard deviations, and when all data are 1, both of these values are 0, giving a correlation of 0/0, which is certainly “not a number” as far as Java is concerned.

Not quite. We're evaluating the average difference between estimated and actual preference, in a world where every preference value is 1. Of course the result is 0; the test itself is invalid because it will only ever result in 0.

However a precision and recall evaluation is still valid. Let's try it.

Listing 3.8 Evaluating precision and recall with boolean data

```
DataModel model = new GenericBooleanPrefDataModel(
    new FileDataModel(new File("ua.base")));

RecommenderIRStatsEvaluator evaluator =
  new GenericRecommenderIRStatsEvaluator();
RecommenderBuilder recommenderBuilder = new RecommenderBuilder() {
  @Override
  public Recommender buildRecommender(DataModel model) {
    UserSimilarity similarity = new LogLikelihoodSimilarity(model);
    UserNeighborhood neighborhood =
      new NearestNUserNeighborhood(10, similarity, model);
    return new GenericBooleanPrefUserBasedRecommender(
        model, neighborhood, similarity);
  }
};
DataModelBuilder modelBuilder = new DataModelBuilder() {
  @Override
  public DataModel buildDataModel(FastByIDMap<PreferenceArray> trainingData) {
    return new GenericBooleanPrefDataModel(
      GenericBooleanPrefDataModel.toDataMap(trainingData));
  }
};
IRStatistics stats = evaluator.evaluate(
    recommenderBuilder, modelBuilder, model, null, 10,
    GenericRecommenderIRStatsEvaluator.CHOOSE_THRESHOLD,
    1.0);
System.out.println(stats.getPrecision());
System.out.println(stats.getRecall());
```

The result is about 15.5% for both precision and recall. That's not great; recall that this means only about 1 in 6 recommendations returned are "good" and about 1 in 6 good recommendations are returned.

This is traceable to a third problem, illustrated here. Preference values are still lurking in one place here: `GenericUserBasedRecommender`. Of course, it still orders its recommendations based on estimate preference, but these values are all 1.0. The ordering is therefore essentially random. So, we introduce `GenericBooleanPrefUserBasedRecommender` (yes, that's about as long as the class names will get). This variant will produce a more meaningful ordering in its recommendations. It weights items that are associated to many other similar users, and to users that are more similar, more heavily. It does not produce a weighted average. So, try substituting this implementation and run the code again. The result is 18% or so. Better, but barely. This strongly suggests this isn't a terribly effective recommender system for this data. Our purpose here isn't to "fix" this, merely to look at how to effectively deploy "boolean" data in Mahout recommenders.

Boolean variants of the other `DataModels` we've seen so far exist as well. `FileDataModel` will automatically use a `GenericBooleanPrefDataModel` internally, if its input data contains no preference values (lines of the form `userID,itemID` only). Similarly, `MySQLBooleanPrefDataModel` is suitable for use with a database table without a preference value column. It's otherwise entirely analogous. This implementation in particular can take advantage of many more shortcuts in the database to improve performance.

Finally, if you're wondering if you can mix boolean and non-boolean data: no. In such a case, it's desirable to treat the data set as having preference values, since some preference values do exist. Those missing an actual preference value can and should be inferred by some means, even if it's as simple as filling in the simple average of all existing preference values as a placeholder.

3.6 Summary

In this chapter we looked at how preference data is represented in a Mahout recommender. This includes Preference objects, but also specialized array and collection-like implementations like `PreferenceArray` and `FastByIDMap`. These specializations exist largely to reduce memory usage.

We looked at `DataModels`, which are the abstraction for recommender input as a whole. `GenericDataModel` stores data in memory, as does `FileDataModel`, after reading input from a file. `JDBCDataModel` and implementations exist to support data based on a relational database table; we examined integration with MySQL in particular.

Finally we looked at how all this changes when the input data does not contain preference values -- only user-item associations. Sometimes this is all that is available, and, it certainly requires less storage. We looked at subtle complications that this sort of data model can cause when used with other standard components, such as `PearsonCorrelationSimilarity`, which are not suitable for this kind of input. We examined several such problems and fixed them in order to get a functioning recommender based on boolean input data.

4 Making Recommendations

This chapter covers

- User-based recommenders, in depth
- Similarity metrics
- Item-based and other recommenders

Having thoroughly discussed evaluating recommenders and representing the data input to recommender, we are at last qualified to examine recommenders themselves in detail. This is where the real action begins.

4.1 User-based recommendation

If you've seen a recommender algorithm explained, chances are it was a user-based recommender algorithm. This is the approach described in some of the earliest research in the field. The label "user-based" is somewhat imprecise, as any recommender algorithm is based on user- and item-related data. The defining characteristic of a user-based recommender algorithm is that it is based upon some notion of similarities between users. In fact, you've probably encountered this type of "algorithm" in everyday life.

4.1.1 When recommendation goes wrong

Have you ever received a CD as a gift? I did, as a young teenage boy, by well-meaning adults. One of these adults seem to have headed down to the local music store and cornered an employee, where the following scene unfolded:

> ADULT: I am looking for a CD for a teenager.
>
> EMPLOYEE: OK, what does this teenager like?
>
> ADULT: Oh, you know, what all the young kids like these days.

EMPLOYEE: What kind of music or bands?

ADULT: It's all noise to me. I don't know.

EMPLOYEE: Uh, well… I guess lots of young people are buying this boy band album here by “New 2 Town”?

ADULT: Sold!

You can guess the result. Needless to say, they instead give me gift certificates now. I am afraid this example of user-based recommendation gone wrong has played out many times. What happened? The intuition was sound: because teenagers have relatively related tastes in music, one teenager would be more likely to enjoy an album that other teenagers have enjoyed. Basing recommendations on similarity among people is quite reasonable.

Of course, recommending an album from a band that teenage girls swoon over probably isn’t the best thing for a teenage boy. The error here was that the similarity metric wasn't effective. Yes, teenagers as a group have relatively homogenous tastes: you're more likely to find pop songs than zydeco or classical music. But, the similarity is too weak to be useful: it’s not true that teenage girls have enough in common with teenage boys when it comes to music to form the basis of a recommendation.

4.1.2 When recommendation goes right

Let's rewind our scenario and imagine how it could have gone better:

ADULT: I am looking for a CD for a teenage boy.

EMPLOYEE: OK, what does he like?

ADULT: Oh, you know, he likes what all the young kids like these days.

EMPLOYEE: What kind of music or bands?

ADULT: I don't know, but his best friend is always wearing a “Bowling In Hades” t-shirt.

EMPLOYEE: Ah yes, a very popular nu-metal band from Cleveland. Well, we do have the new Bowling In Hades best-of album over here, “Impossible Split: The Singles 1997-2000”...

Now that's better. The recommendation was based on the assumption that two good friends share somewhat similar taste in music. That’s more reasonable. With a reliable similarity metric in place, the outcome is probably better. It's far more likely that these best friends share a love for Bowling In Hades than any two random teenagers. Here's another way it could have gone better:

ADULT: I am looking for a CD for a teenage boy.

EMPLOYEE: OK, what does he like?

ADULT: Oh, you know, he likes what all the young kids like these days.

EMPLOYEE: What kind of music or bands?

ADULT: “Music”? Ha, well, I wrote down the bands from posters on his bedroom wall. The Skulks, Rock Mobster, the Wild Scallions... mean anything to you?

EMPLOYEE: I see, well, my kid is into some of those albums too. And he won't stop talking about some new album from Diabolical Florist, so maybe...

Well done, adults. Now, they've inferred a similarity based directly on tastes in music. Because the two kids in question both prefer some of the same bands, it stands to reason they'll each like a lot in the rest of each other’s collections. That’s even better reasoning than guessing their tastes are similar because they’re friends. They’ve actually based their idea of similaritybetween the two teenagers on observed tastes in music. This is the essential logic of a user-based recommender system.

4.2 Exploring the user-based recommender

If we let these two adults keep going, they'd further refine their reasoning. Why base the choice of gift on just one other kid's music collection? How about finding several other similar kids? They would pay attention to which kids seemed most similar – most same posters and t-shirts and CDs scattered on top of stereos – and look at which bands seemed most important to those most similar kids, and figure those make the best gift.

4.2.1 The algorithm

The user-based recommender algorithm comes out of this intuition. We might describe the process of recommending items to some user, denoted by u, like so:

```
for every item i that u has no preference for yet
  for every other user v that has a preference for i
    compute a similarity s between u and v
    incorporate v's preference for i, weighted by s, into a running average
return the top items, ranked by weighted average
```

The outer loop simply suggests we should consider every known item (that the user hasn’t already expressed a preference for -- they’re already well aware of those items and what they think of them) as a candidate for recommendation. The inner loop suggests we look to any other user who has expressed a preference for this candidate item, and see what his or her preference value for it was. In the end, we average these values to come up with an estimate -- a weighted average, that is. We weight each preference value in the average by how similar that user is to our target user. The more similar a user, the more heavily we weight his or her preference value.

It would be terribly slow to examine every item. In reality, first, a “neighborhood” of most similar users it computed first, and only items known to those users are considered

```
for every other user v
  compute a similarity s between u and v
  retain the top users, ranked by similarity, as a “neighborhood” n
for every item i that some user in n has a preference for,
      but that u has no preference for yet
  for every other user v in n that has a preference for i
    compute a similarity s between u and v
    incorporate v's preference for i, weighted by s, into a running average
```

The primary difference is that we find the similar users first, and see what those most-similar users are interested in first, and then take those items as our candidates. The rest is the same. This is the standard user-based recommender algorithm.

4.2.2 Implementing the algorithm with GenericUserBasedRecommender

We have already seen a user-based recommender in action, in the very first example. Let's return to it, in order to explore the components in use and see how well it performs.

Listing 4.1 Revisiting of a simple user-based Recommender system

```
DataModel model = new FileDataModel(new File("intro.csv"));
UserSimilarity similarity = new PearsonCorrelationSimilarity(model);
UserNeighborhood neighborhood =
  new NearestNUserNeighborhood(2, similarity, model);
Recommender recommender =
  new GenericUserBasedRecommender(model, neighborhood, similarity);
```

`UserSimilarity` encapsulates some notion of similarity amongst users. And, `UserNeighborhood` encapsulates some notion of a group of most-similar users. These are necessary components of the standard user-based recommender algorithm.

There isn't only one possible notion of similarity – we already discussed a few real-world ideas of similarity above. There are also many ways to define a neighborhood of most similar users: the 5 most similar? 20? Users with a similarity above a certain value? To illustrate these, imagine you’re creating a guest list for your wedding. You want to invite your closest friends and family to this special occasion, but you have far more friends and family than your budget will allow. Would you decide who is and isn’t invited by picking a size first -- say 50 people -- and picking your 50 closest friends and family? Is 50 the right number, or 40 or 100? Or would you invite everyone who you consider “close”? Should you only invite your “really close” friends? Which one will give the best wedding party? This is analogous to the decision you make when deciding how to pick a neighborhood of similar users.

Plug in new ideas of similarity and you get quite different results. You can begin to see that there is not merely one way to produce recommendations – and we're still looking at one facet of one approach that can be adjusted. As you will see, Mahout is not one recommender engine at all, but an assortment of components that may be plugged together and customized to create an ideal recommender for a particular domain. Here we’ve put together the following components:

- Data model implemented via `DataModel`

- User-user similarity metric implemented via UserSimilarity
- User neighborhood definition implemented via UserNeighborhood
- Recommender engine implemented via a Recommender: here, GenericUserBasedRecommender

As you will see through the rest of the book, getting good results, and getting them fast, is inevitably a long process of experimentation and refinement.

4.2.3 Exploring with GroupLens

Let's return to the GroupLens data set and up the ante. This time we will use 100 times more data. We promised scale, right? Return to http://grouplens.org and download the 10 million rating data set, which is currently available at http://www.grouplens.org/node/73. Unpack it locally and locate the ratings.dat file inside.

For whatever reason, the format of this data is different from the 100,000 rating data set we had used before. Whereas its ua.base file was ready for use with FileDataModel, this data set's ratings.dat file is not. It would be simple to use standard command-line text-processing utilities to convert it to a comma-separated form, and in general, this is the best approach. Writing custom code to convert the file format, or a custom DataModel, is tedious and error prone.

Luckily, in this particular case there's an easier solution: Mahout's examples module includes the custom implementation GroupLensDataModel, which extends FileDataModel to read this file. Make sure you have included the code under the examples/ directory in your project in your IDE. Then, swap out FileDataModel for this alternative:

Listing 4.2 Updating to use a custom DataModel for GroupLens

```
DataModel model = new GroupLensDataModel(new File("ratings.dat"));
UserSimilarity similarity = new PearsonCorrelationSimilarity(model);
UserNeighborhood neighborhood =
  new NearestNUserNeighborhood(100, similarity, model);
Recommender recommender =
  new GenericUserBasedRecommender(model, neighborhood, similarity);
LoadEvaluator.runLoad(recommender);
```

Run this, and the first thing you will likely encounter is an OutOfMemoryError. Ah, a first sighting of issues of scale. By default, Java will not grow its heap past a certain modest size. We need to increase the amount of heap space available to Java.

This is a good first opportunity to discuss what can be done to improve performance by tuning the JVM. Refer to Appendix A at this point for a more in-depth discussion of JVM tuning.

4.3 Exploring user neighborhoods

Let's next evaluate the recommender accuracy. Yes, we've done this before; we'll present the boilerplate evaluation code one more time, but, going forward, we figure you've got the hang of it and can construct and run evaluations on your own.

Now, we look at possibilities for configuring and modifying the neighborhood implementation. Remember, we're also using 100 times more data as well.

Listing 4.3 Running an evaluation on the simple Recommender

```
DataModel model = new GroupLensDataModel(new File("ratings.dat"));
RecommenderEvaluator evaluator =
  new AverageAbsoluteDifferenceRecommenderEvaluator();
RecommenderBuilder recommenderBuilder = new RecommenderBuilder() {
  @Override
  public Recommender buildRecommender(DataModel model) throws TasteException {
    UserSimilarity similarity = new PearsonCorrelationSimilarity(model);
    UserNeighborhood neighborhood =
      new NearestNUserNeighborhood(100, similarity, model);
    return new GenericUserBasedRecommender(model, neighborhood, similarity);
  }
};
double score = evaluator.evaluate(recommenderBuilder, null, model, 0.95, 0.05);
System.out.println(score);
```

Note how the final parameter to `evaluate()` is 0.05. This means only 5% of all the data is used for evaluation. This is purely for convenience; evaluation is a time-consuming process and using this full data set, could take hours to complete. For purposes of quickly evaluating changes, it's convenient to reduce this value. We shouldn’t push it down too far, as using too little data might compromise the accuracy of the evaluation result. The parameter 0.95 simply says to build a model to evaluate with 95% of the data, and then test with the remaining 5%. After running this, your evaluation result will vary, but should likely be around 0.89.

4.3.1 Fixed-size neighborhoods

At the moment, the recommendations are derived from a neighborhood of the 100 most similar users (see use of `NearestNUserNeighborhood` with neighborhood size 100). We’ve arbitrarily decided that we will always use the 100 users whose similarity is greatest in order to make recommendations. What if this were 10? We'd base recommendations on fewer similar users, but would exclude some less-similar users from consideration.

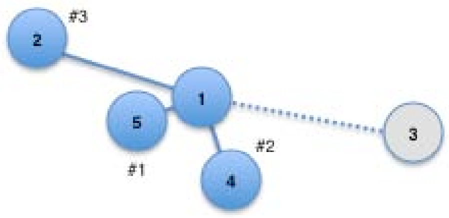


Figure 4.1 An illustration of defining a neighborhood of most similar users by picking a fixed number of closest neighbors. Here, distance illustrates similarity: farther means less similar. In this picture, neighborhood around user 1 is chosen to consist of the three most similar users: 5, 4, and 2.

Try this change -- replace 100 with 10. The result of the evaluation, the average difference between estimated and actual preference value, is 0.98 or so. Recall that larger evaluation values are worse, so that's a move in the wrong direction. The most likely explanation is that 10 users are too few. It's likely that the eleventh and twelfth most similar users and so on add value. They are still quite similar, and, are associated to items that the first 10 most similar users weren't.

We could go the other direction, and try a neighborhood of 500 users; the result drops to 0.75, which is of course better. We could evaluate many values and figure out the optimal setting for this data set. For brevity, we won't, but rather continue musing about the recommender. The lesson is that there is no magic value; some experimentation with real data is necessary to tune your recommender.

4.3.2 Threshold-based neighborhood

What if we don't want to build a neighborhood of the n most similar users, but rather try to pick the "pretty similar" users and ignore everyone else? We could pick a similarity threshold and take any users that are at least that similar.

The threshold should be between -1 and 1, since all similarity metrics return similarity values in this range. At the moment, we use a standard Pearson correlation as the similarity metric. Those familiar with this correlation would likely agree that a value of 0.7 or above is a "high correlation" and constitutes a sensible definition of "pretty similar." So, we now switch to use `ThresholdUserNeighborhood`. It's as simple as changing one line to `new ThresholdUserNeighborhood(0.7, similarity, model)` where we have created the `UserSimilarity` implementation in our evaluation code.

Now the evaluator scores our recommender at 0.84. What if we make the neighborhood more selective by choosing a threshold of 0.9? The score worsens to 0.92; it's likely that the same explanation applies. How about 0.5? The score improves to 0.78. We will use a threshold-based neighborhood with threshold 0.5 for the examples that follow.

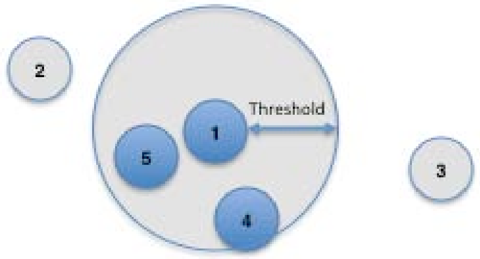


Figure 4.2 An illustration of defining a neighborhood of most-similar users with a similarity threshold.

Again, you would likely want to explore many more values on real data to determine an optimum, but already we've improved estimation accuracy by about 15% with some simple tinkering.

4.4 Exploring similarity metrics

We continue the survey of user-based recommenders by examining changes to the most important part: the `UserSimilarity` implementation. A user-based recommender relies most of all on this component. Without a reliable and effective notion of which users are similar to others, this approach falls apart.

4.4.1 Pearson correlation-based similarity

So far, we have used the `PearsonCorrelationSimilarity`, which is a similarity metric based on the Pearson correlation. The Pearson correlation is a number between -1 and 1. It measures the tendency of two series of numbers, paired up one-to-one, to move together. That is to say, it measures how much a number in one series to be relatively large when the corresponding number in the other series is high, and vice versa. To be exact, it measures the tendency of the numbers to move together proportionally, such that there is a roughly linear relationship between the values in one series and the other. When this tendency is high, the correlation is close to 1. When there appears to be little relationship at all, the value is near 0. When there appears to be an opposing relationship -- one series' numbers are high exactly when the other series' numbers are low -- the value is near -1.

This concept, widely used in statistics, can be applied to users to measure their similarity. We use it to measure the tendency of two users' preference values to move together – to be relatively high, or relatively low, on the same items. For an example, look back to the first sample data file we created:

Listing 4.4 Restatement of simple recommender input file

```
1,101,5.0
1,102,3.0
1,103,2.5

2,101,2.0
2,102,2.5
2,103,5.0
2,104,2.0

3,101,2.5
3,104,4.0
3,105,4.5
3,107,5.0

4,101,5.0
4,103,3.0
4,104,4.5
4,106,4.0

5,101,4.0
5,102,3.0
5,103,2.0
5,104,4.0
5,105,3.5
5,106,4.0
```

We noted that users 1 and 5 seem similar since their preferences seem to run together. On items 101, 102, and 103, they roughly agree: 101 is the best, 102 somewhat less good, and 103 isn't desirable. By the same reasoning users 1 and 2 are not so similar. Note that we can't really include items 104 through 106 in our reasoning about user 1, since we don't know anything about user 1's preference for 104 through 106. As far as we're concerned, the similarity computation can only operate on items that both users have expressed a preference for. In an upcoming section, we'll look at what happens when we infer "missing" preference values like this.

The Pearson correlation captures these notions, as can be seen from the table 4.2. We don't reproduce the details of the computation here; refer to other sources for a complete explanation of how the correlation is computed.

Table 4.2 This table shows the Pearson correlation between user 1 and other users (note that a user's correlation with itself is always 1.0) based on the three items that user 1 has in common with the others.

	Item 101	Item 102	Item 103	Correlation with User 1
User 1	5.0	3.0	2.5	1.000
User 2	2.0	2.5	5.0	-0.764
User 3	2.5	-	-	-
User 4	5.0	-	3.0	1.000
User 5	4.0	3.0	2.0	0.945

4.4.2 Pearson correlation problems

While the results are indeed intuitive, the Pearson correlation has some quirks in the context of recommender engines. It doesn't take into account the number of items in which two users' preferences overlap, which is probably a weakness in the context of recommender engines. Two users that have seen 200 of the same movies, for instance, even if they don't often agree on ratings, are probably more similar than two users who have only ever seen 2 movies in common. This issue appears in a small way in the data above; note that users 1 and 5 have both expressed preferences for all three items, and seem to have similar tastes. Yet, users 1 and 4 have a higher correlation of 1.0, based on only two overlapping items. This seems a bit counterintuitive.

If two users overlap in only one item, no correlation can be computed, because of how the computation is defined. This is why no correlation can be computed between users 1 and 3. This could be an issue for small or sparse data sets, in which users item sets rarely overlap. Or, one could also view it as a benefit: two users that overlap in only one item are, intuitively, not very similar anyway.

The correlation is also undefined if either series of preference values are all identical -- we noted this problem before. For example, if user 5 had expressed a preference of 3.0 for all three items above, we could not compute a similarity between 1 and 5 since the Pearson correlation would be undefined. This is likewise most probably an issue when users rarely overlap with others in the items they've expressed any preference for.

While the Pearson correlation commonly appears with recommenders in early research papers (see http://lucene.apache.org/mahout/taste.html), and appears in introductory books on recommenders, it is not necessarily a good first choice. It's not necessarily bad, either; it simply bears understanding how it works.

4.4.3 Employing weighting

`PearsonCorrelationSimilarity` provides an "extension" to the standard computation, called weighting, that mitigates one of the issues above. The Pearson correlation does not reflect, directly, the number of items over which it is computed. For our purposes that would be desirable: when based on more information, the resulting correlation would be a more reliable result. In order to reflect this, we'd like to push positive correlation values towards 1.0, and negative values towards -1.0, when the correlation is based on more items. Alternatively, you could imagine pushing the correlation values towards some mean preference value when the correlation is based on fewer items; the effect would be similar, but the implementation somewhat more complex as it would require tracking what the mean preference value is for pairs of users.

In listing 4.3, passing the value `Weighting.WEIGHTED` to the constructor of `PearsonCorrelationSimilarity` as the second argument does this. It will cause the resulting correlation to be pushed towards 1.0, or -1.0, depending on how many data points where used to compute the correlation value. A quick re-run of the evaluation framework reveals that, in this case, this setting improves the score slightly to 0.77.

4.4.4 Defining similarity by Euclidean distance

Let's try `EuclideanDistanceSimilarity` -- swap in a different implementation by simply changing the `UserSimilarity` implementation used in listing 4.3 to `new EuclideanDistanceSimilarity(model)` instead.

This implementation is based on the "distance" between users. This idea makes sense if you think of users as points in a space of many dimensions (as many dimensions are there are items), whose coordinates are preference values. This similarity metric computes the Euclidean distance[5] d between two such user "points". This value alone does not constitute a valid similarity metric, because larger values would mean more distant, and therefore less similar, users. We need the value to be smaller when users are more similar. Therefore, the implementation actually returns 1 / (1+d). You can verify that when the distance is 0 (users have identical preferences) the result is 1, decreasing to 0 as d increases. This similarity metric never returns a negative value, but larger values still mean more similarity.

Table 4.3 This table shows the Euclidean "distance" between user 1 and other users, and resulting similarity scores.

	Item 101	Item 102	Item 103	Distance	Similarity to User 1
User 1	5.0	3.0	2.5	0.000	1.000
User 2	2.0	2.5	5.0	3.937	0.203
User 3	2.5	-	-	2.500	0.286
User 4	5.0	-	3.0	0.500	0.667
User 5	4.0	3.0	2.0	1.118	0.472

[5] Recall this is the square root of the sum of squares of the differences in coordinates

After changing our last example to use `EuclideanDistanceSimilarity`, the result is 0.75 – it happens to be a little better in this case, but barely. Note that we were able compute some notion of similarity for all pairs of users here, whereas the Pearson correlation couldn't produce an answer for users 1 and 3. This is good, on the one hand, though the result is based on one item in common, which could be construed as undesirable. We note that this implementation also has the same possibly counterintuitive behavior: users 1 and 4 have a higher similarity than users 1 and 5.

4.4.5 Adapting the cosine measure similarity

The cosine measure similarity is another similarity metric that depends on envisioning user preferences as like points in space. Hold in mind the example above, of user preferences as points in an n-dimensional space. Imagine two lines from the origin, or point (0,0,…,0), to each of these two points. When the two users are similar, they will have similar ratings, and so will be relatively close in space -- at least, they'll be in roughly the same direction from the origin. The angle formed between these two lines will be relatively small. In contract, when the two users are dissimilar, their points will be distant, and likely in different directions from the origin, forming a wide angle.

This angle can be used as the basis for a similarity metric, in the same way we used a distance to form a similarity metric above. In this case, we take the cosine of the angle as the similarity value. If you're rusty on trigonometry, all you need to remember to understand this is that the cosine value is always between -1 and 1 and that the cosine of a small angle is near 1, and the cosine of a large angle near 180 degrees is close to -1. This is good, since we want small angles to map to high similarity, near 1, and large angles to map to near -1.

You may be searching for something like "`CosineMeasureSimilarity`" in Mahout. You've actually already found it but under an unexpected name: `PearsonCorrelationSimilarity`. The cosine measure similarity and Pearson correlation aren't the same thing, but, if you bother to work out the math, they actually reduce to the same computation when the two series of input values each have a mean of 0 ("centered").

The cosine measure similarity is commonly referenced in research on collaborative filtering. You can employ this similarity metric too by simply using `PearsonCorrelationSimilarity`.

4.4.6 Defining similarity by relative rank with the Spearman correlation

The Spearman correlation is an interesting variant on the Pearson correlation, for our purposes. Rather than compute a correlation based on the original preference values, it computes a correlation based on the relative rank of preference values. Imagine that, for each user, we find his or her least preferred item and overwrite its preference value with a "1". Then we change the next-least-preferred item's preference value to "2", and so on. To illustrate this, imagine if you were rating movies and gave your least-preferred movie 1 star, the next-least favorite 2 stars, and so on. Then, we compute a Pearson correlation on the transformed values. This is the Spearman correlation.

This process loses some information. While it preserves the essence of the preference values -- their ordering -- it removes information about exactly how much more each item was liked than the last. This may or may not be a good idea; it is somewhere between keeping preference values and forgetting them entirely, two options we've already looked at.

Table 4.4 below shows the resulting Spearman correlations. Its simplifications on this already-simple data set result in some extreme values: in fact, all correlations are 1 or -1 here, depending on whether a user's preference values run with or counter to user 1's preferences here. As with the Pearson correlation, no value can be computed between users 1 and 3.

Table 4.4 This table shows the preference values transformed into rank, and the resulting Spearman correlation between user 1 and each of the other users.

	Item 101	Item 102	Item 103	Correlation to User 1
User 1	3.0	2.0	1.0	1.0
User 2	1.0	2.0	3.0	-1.0
User 3	1.0	-	-	-
User 4	2.0	-	1.0	1.0
User 5	3.0	2.0	1.0	1.0

`SpearmanCorrelationSimilarity` implements this idea. You could try using this as the `UserSimilarity` in the evaluator code we've been using so far. Run it, and take a long coffee break. Turn in for the night. It won't finish anytime soon. This implementation is far slower because it must do some non-trivial work to compute and store these ranks, and is orders of magnitude slower. The Spearman correlation-based similarity metric is expensive to compute, and is therefore possibly of academic interest more than practical use. For some small data sets, it may be desirable.

It's a fine time to introduce one of many caching wrapper implementations available in Mahout. `CachingUserSimilarity` is a `UserSimilarity` implementation that wraps another `UserSimilarity` implementation and caches its results. That is, it delegates computation to another, given implementation, and remembers those results internally. Later when asked for a user-user similarity value that was previously computed, it can answer immediately rather than delegate to the given implementation again to compute. In this way, one can add on caching to any similarity implementation. When the cost of performing a computation is relatively high, as here, it can be worthwhile to employ. The cost, of course, is memory consumed by the cache. So, instead, try using:

Listing 4.5 Employing caching with a UserSimilarity implementation

```
UserSimilarity similarity = new CachingUserSimilarity(
    new SpearmanCorrelationSimilarity(model), model);
```

It's also advisable to decrease the amount of test data from 5% to 1% by increasing the `trainingPercentage` argument to `evaluate()` from 0.95 to 0.99. It would also be wise to decrease the evaluation percentage from 5% to 1% by changing the last parameter from 0.05 to 0.01. This will allow the evaluation to finish in more like tens of minutes. The result should be near 0.80. Again, broad conclusions are difficult to draw: on this particular data set, it was not quite as effective as other similarity metrics.

4.4.7 Ignoring preference values in similarity with the Tanimoto coefficient

Interestingly, there are also `UserSimilarity` implementations that ignore preference values entirely. They don't care whether a user expresses a high or low preference for an item – only that the user expresses a preference at all. How can this be a good idea? If preference values are good data, then ignoring them seems like a bad idea that hurts performance. But we saw that it didn't necessarily hurt at all. This should serve as an additional warning that more data is not necessarily better.

`TanimotoCoefficientSimilarity` is one such implementation, based on (surprise) the Tanimoto coefficient. This value is also known as the Jaccard coefficient. It is the number of items that both of two users express some preference for, divided by the number of items that either user expresses some preference for, as illustrated in figure 4.4.

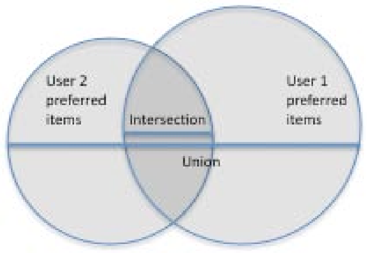


Figure 4.3 The Tanimoto coefficient is the ratio of the size of the intersection, or overlap in two users' preferred items (dark area), to the union of the users' preferred items (dark and light areas together).

In other words, it is the ratio of the size of the intersection to the size of the union of their preferred items. It has the required properties: when two users' items completely overlap, the result is 1.0. When they have nothing in common, it's 0.0. The value is never negative, but that's OK. If we wished, we could expand the results into the range -1 to 1 with some simple math: similarity = 2 • similarity - 1. It won't matter to the framework.

Table 4.5 This table shows the similarity values between user 1 and other users, computed using the Tanimoto coefficient. Note that preference values themselves are omitted, as they are not used in the computation.

	Item 101	**Item 102**	**Item 103**	**Item 104**	**Item 105**	**Item 106**	**Item 107**	**Similarity to User 1**
User 1	X	X	X					1.0
User 2	X	X	X	X				0.75
User	X			X	X		X	0.17

3								
User 4	X		X	X		X		0.4
User 5	X	X	X	X	X	X		0.5

Note that this similarity metric does not depend only on the items that both have some preference for, but that either has some preference for. Hence, all seven items appear in our calculation, unlike before.

You're likely to use this similarity metric if, and only if, your underlying data contains only "boolean" preferences, and you have no preference values to begin with. If you do have preference values, presumably it is because you believe they are more signal than noise. You would usually do better with a metric that uses this information. In our GroupLens data set, using this metric gives a slightly worse score of 0.82.

4.4.8 Computing smarter similarity with a log-likelihood test

Log-likelihood-based similarity is similar to the Tanimoto coefficient-based similarity, though more difficult to understand intuitively. It is also a metric that does not take account of individual preference values. The math involved in computing this similarity metric is beyond the scope of this book to explain. It is also based on the number of items in common between two users, but, its value is more an expression of how unlikely it is for two users to have so much overlap, given the total number of items out there and the number of items each user has a preference for.

To illustrate, consider two movie fans who have each seen and rated several moves, but, have only both seen "Star Wars" and "Casablanca". Are they similar? If they have each seen hundreds of movies, it wouldn't mean much. Many people have seen these movies, and, if these two have seen many movies but only managed to overlap in these two, they're probably not similar. On the other hand, if each user has seen just a few movies, and these two were on both users' lists, then it would seem to imply they're similar people, when it comes to movies; the overlap would be significant.

The Tanimoto coefficient already encapsulates some of this thinking, since it looks at the ratio of the size of the intersection of their interests to the union. The log-likelihood is computing something slightly different. It is trying to assess how unlikely it is that the overlap between the two users is just due to chance. That is to say, two dissimilar users will no doubt happen to rate a couple movies in common; two similar users will show an overlap that looks quite unlikely to be mere chance. With some statistical tests, this similarity metric attempts to find just how strongly unlikely it is that two users have no resemblance in their tastes; the more unlikely, the more similar we figure the two are. This requires looking at a little more than mere intersection and union of their preferred items.

Table 4.6 This table shows the similarity values between user 1 and other users, computed using the log-likelihood similarity metric.

	Item 101	**Item 102**	**Item 103**	**Item 104**	**Item 105**	**Item 106**	**Item 107**	**Similarity to User 1**

User 1	X	X	X					0.90
User 2	X	X	X	X				0.84
User 3	X			X	X		X	0.55
User 4	X		X	X		X		0.16
User 5	X	X	X	X	X	X		0.55

Using a log-likelihood-based similarity metric is as easy as inserting `new LogLikelihoodSimilarity` in listing 4.3, as before.

While it's hard to generalize, log-likelihood-based similarity will probably outperform Tanimoto coefficient-based similarity. It is, in a sense, a more intelligent metric. Re-running the evaluation shows that, at least for our data set and recommender, it improves performance over `TanimotoCoefficientSimilarity`, to 0.73.

4.4.8 Inferring preferences

We noted above that sometimes too little data is a problem. In a few cases, for example, the Pearson correlation was unable to compute any similarity value at all since some pairs of users overlap in only one item. The Pearson correlation can't take account of preference values for items which only one user has expressed a preference either.

What if we "filled in the blanks" with some default value? For example, we could pretend that each user has rated every item by inferring preferences for items for which the user hasn't explicitly expressed a preference. This sort of strategy is enabled via the `PreferenceInferrer` interface, which at the moment has one implementation, `AveragingPreferenceInferrer`. This implementation computes the average preference value for each user and fills in this average as the preference value for any item not already associated to the user. It can be enabled on a `UserSimilarity` implementation with a call to `setPreferenceInferrer()`.

While this strategy is available, it is in practice not usually helpful. It is provided primarily because it is mentioned in early research papers on recommender engines. In theory, making up information purely based on existing information isn't adding anything. It certainly does slow down computations drastically. It is available for experimentation, but will likely not be useful when applied to real data sets.

4.5 Item-based recommendation

We've looked at user-based recommenders -- not one recommender, but tools to build a nearly limitless number of variations on the basic user-based approach, by plugging in different and differently configured components into the implementation.

Yet there are other approaches to recommendation, and next we will look at item-based recommenders. This section will be shorter, since several of the components we've seen already (data models, similarity implementations) still apply to item-based recommenders.

Item-based recommendation is derived from how similar items are to items, instead of users to users. To illustrate, return to the pair we left in the music store, doing their best to pick an album that a teenage boy would like. Imagine yet another line of reasoning they could have adopted:

ADULT: I am looking for a CD for a teenage boy.

EMPLOYEE: OK, what does he like?

ADULT: Oh, you know, he likes what all the young kids like these days.

EMPLOYEE: What kind of music or bands?

ADULT: He wears a Bowling In Hades t-shirt all the time and seems to have all of their albums. Anything else you'd recommend?

EMPLOYEE: Well, about everyone I know that likes Bowling In Hades seems to like the new Rock Mobster album.

This sounds reasonable. Is this different from previous examples? Yes. The record store employee is recommending an item that is similar to something we already know the boy likes. This is not the same as before, where the question was, "who is similar to the boy, and what do they like?" Here the question is, "what is similar to what the boy likes?"

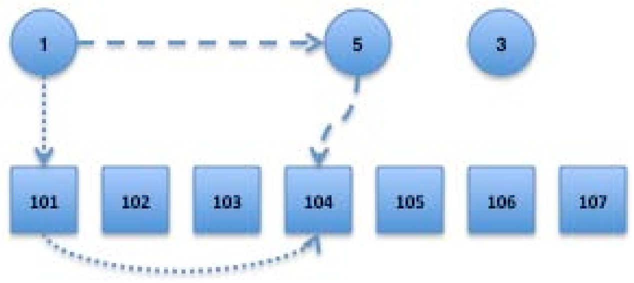


Figure 4.4 A basic illustration of the difference between user-based and item-based recommendation: user-based recommendation (large dashes) finds similar users, and sees what they like. Item-based recommendation (short dashes) sees what the user likes, then finds similar items.

4.5.1 The algorithm

The algorithm will feel familiar, having seen user-based recommenders already:

```
for every item i that u has no preference for yet
  for every item j that u has a preference for
    compute a similarity s between i and j
    add u's preference for j, weighted by s, to a running average
return the top items, ranked by weighted average
```

The third line shows how it is based on item-item similarities, not user-user similarities as before. The algorithms are similar, but not entirely symmetric. They do have notably different properties. For instance, the running time of an item-based recommender scales up as the number of items increases, whereas a user-based recommender's running time goes up as the number of users increases.

This suggests one reason that you might choose an item-based recommender: if the number of users is relatively low compared to the number of items, the performance advantage could be significant.

Also, items are typically less subject to change than users. When items are things like DVDs, we expect that over time, as we acquire more data, that our estimates of the similarities between items converge. We have no reason to expect them to change radically or frequently. Some of the same may be said of users, but, users can change over time and new knowledge of users is likely to come in bursts of new information that must be digested quickly. To connect this to the last example, it's likely that Bowling in Hades albums and Rock Mobster albums will remain as similar to each other next year as today. However, it's a lot less likely that the same fans mentioned above will have the same tastes next year, and so, their similarities will change more.

We observe this in order to argue that if item-item similarities are more fixed, then they are better candidates for precomputation. Precomputing similarities takes work, but of course speeds up recommendations at run time. This could be desirable in contexts where delivering recommendations quickly at run time is essential -- think about a news site which must potentially deliver recommendations immediately with each news article view.

4.5.2 Exploring the item-based recommender

Let's insert a simple item-based recommender into our familiar evaluation framework, using the following code. Here we're deploying `GenericItemBasedRecommender` rather than `GenericUserBasedRecommender`, and it requires a different and simpler set of dependencies.

Listing 4.6 The core of a basic item-based recommender

```
@Override
public Recommender buildRecommender(DataModel model)
    throws TasteException {
  ItemSimilarity similarity = new PearsonCorrelationSimilarity(model);
  return new GenericItemBasedRecommender(model, similarity);
}
```

`PearsonCorrelationSimilarity` still works here, because it also implements the `ItemSimilarity` interface, which is entirely analogous to the `UserSimilarity` interface that we've already seen. It implements the same notion of similarity, based on the Pearson correlation, but between items instead of users. That is, it compares series of preferences expressed by many users, for one item, rather than by one user for many items.

GenericItemBasedRecommender is simpler. It only needs a DataModel and ItemSimilarity -- no “ItemNeighborhood”. You might wonder at the apparent asymmetry. Recall that the item-based recommendation process already begins with a limited number of starting points: the items that the user in question already expresses a preference for. This is analogous to the neighborhood of similar users that the user-based approach first identifies. It doesn’t make sense in the second half of the algorithm to compute neighborhoods around each of the user’s preferred items.

You are invited to experiment with different similarity metrics, as above. Not all of the implementations of UserSimilarity that we have seen so far also implement ItemSimilarity. By now, you’ll already know how to evaluate the accuracy of this item-based recommender when using various similarity metrics on our now-familiar GroupLens data set. Results are reproduced below for convenience.

Table 4.7 Evaluation result under various ItemSimilarity metrics

Implementation	**Similarity**
PearsonCorrelationSimilarity	0.75
PearsonCorrelationSimilarity + weighting	0.75
EuclideanDistanceSimilarity	0.76
EuclideanDistanceSimilarity + weighting	0.78
TanimotoCoefficientSimilarity	0.77
LogLikelihoodSimilarity	0.77

One thing you may notice is this recommender setup runs significantly faster. This is not surprising, given that the data set has about 70,000 users and 10,000 items. We noted that item-based recommenders would generally be faster when there are fewer items than users. You may, as a result, wish to increase the percentage of data used in the evaluation to 20% or so (pass 0.2 as the final argument to evaluate()). This should result in a more reliable evaluation. Note there is little apparent difference among these implementations on this data set.

4.6 Slope-one recommender

Did you like the movie “Carlito's Way”? Most people who liked this movie, it seems, also liked another film starring Al Pacino – like “Scarface”. But people tend to like Scarface a bit more. We'd imagine most people that think of Carlito's Way as a four-star movie would give Scarface five stars. So if you told me you thought Carlito's Way was a three-star movie, I might guess you'd give Scarface four stars – one more than the other film.

If you agree with this sort of reasoning, you will like the slope-one recommender (http://en.wikipedia.org/wiki/Slope_One). It estimates preferences for new items based on average difference in preference value (“diffs”) between a new item and the other items the user prefers.

For example, let's say that we know that, on average, people rate Scarface higher by 1.0 than Carlito's Way. Let's also say we find everyone rates Scarface the same as The Godfather, on average.

And now, we are presented with a user who rates Carlito's Way 2.0, and The Godfather 4.0. What do we estimate his preference for Scarface would be?

Based on Carlito's Way, we'd guess 2.0 + 1.0 = 3.0. Based on The Godfather, we'd guess 4.0 + 0.0 = 4.0. Taking a simple average of the two, we'd guess 3.5. This is the essence of the slope-one recommender approach.

4.6.1 The algorithm

Its name comes from the fact that the recommender algorithm starts with the assumption that there is some linear relationship between the preference values for one item and another, that we can in general estimate the preferences for some item Y based on the preferences for item X, via some linear function like Y = mX + b. Then, the slope-one recommender makes the additional simplifying assumption that m=1: “slope one”. We're left attempting to find b = Y-X, the (average) difference in preference value, for every pair of items.

So, the algorithm consists of a significant preprocessing phase, in which all item-item preference value differences are computed:

```
for every item i
  for every other item j
    for every user u expressing preference for both i and j
      add the difference in u’s preference for i and j to an average
```

And then, the recommendation algorithm becomes:

```
for every item i the user u expresses no preference for
  for every item j that user u expresses a preference for
    find the average preference difference between j and i
    add this diff to u’s preference value for j
    add this to a running average
return the top items, ranked by these averages
```

The average diffs over the small sample recommender input we have been using throughout the book are showing in table 4.8.

Table 4.8 Average difference in preference value between all pairs of items. Cells along the diagonal are 0.0. Cells in the bottom left are simply the negative of their counterparts across the diagonal. Hence these are not represented explicitly. Some diffs don’t exist, such as 102-107, since no user expressed a preference for both 102 and 107.

	Item 101	Item 102	Item 103	Item 104	Item 105	Item 106	Item 107
Item 101		-0.833	0.875	0.25	0.75	-0.5	2.5
Item 102			0.333	0.25	0.5	1.0	-
Item 103				0.167	1.5	1.5	-
Item 104					0.0	-0.25	1.0
Item 105						0.5	0.5

Item 106		-
Item 107		

Slope-one is attractive because the on-line portion of the algorithm is fast. Like an item-based recommender, its performance does not depend upon the number of users in the data model. It depends only upon the average preference difference between every pair of items, which can be pre-computed. Further, its underlying data structure can be efficiently updated: when a preference changes, it's simple to update relevant diff values. In contexts where preferences may change quickly, this is an asset.

Note that the memory requirements necessary to store all of these item-item differences in preference value grow as the square of the number of items. Twice as many items means four times the memory!

4.6.2 Slope-one in practice

We can easily try the slope-one recommender by simply employing the code below. Note that the slope-one recommender takes no similarity metric as a necessary argument: `new SlopeOneRecommender(model)`.

After running a standard evaluation using, again, the GroupLens 10M ratings data set, you'll get a result near 0.65. That's the best yet. Indeed, the simple slope-one approach works well in many cases. This algorithm does not make use of a similarity metric, unlike the other approaches we have looked at. It has relatively few "knobs" to twiddle.

Like the Pearson correlation, the simplest form of the slope-one algorithm has a vulnerability: item-item diffs are given equal weighting regardless of how "reliable" they are, how much data they are based upon. Let's say only one user in the history of movie watching has rated both Carlito's Way and The Notebook. It's possible; they're quite different films. We could compute a diff for these two films. Would it be as useful as the diff we compute between Carlito's Way and The Godfather, averaged over thousands of users? It sounds unlikely. The latter diff is probably more reliable since it is an average over a higher count of users.

Again, we can employ some form of weighting to improve recommendations by taking some account of this. `SlopeOneRecommender` offers two types of weighting: weighting based on count, and on standard deviation. Recall that slope-one estimates preference values by adding diffs to all of the user's current preference values, and then averaging all of those results together to form an estimate. Count weighting will weight more heavily those elements based on diffs that are based on more data, more users who have expressed a preference for both items in question. In particular, the average becomes a weighted average, where the diff "count" is the weight -- the number of users on which the diff is based.

Similarly, standard deviation weighting will weight according to the standard deviation of difference in preference value. Lower standard deviation means higher weighting. If the difference in preference value between two films is very consistent across many users, it seems more reliable and should be given more weight. If it varies considerably from user to user, then it should be deemphasized.

These variants turn out to be enough of a good idea that they are enabled by default. You already used this strategy when you ran the evaluation above. We could disable them to see the effect:

Listing 4.7 Selecting no weighting with a SlopeOneRecommender

```
DiffStorage diffStorage = new MemoryDiffStorage(
    model, Weighting.UNWEIGHTED, false, Long.MAX_VALUE));
return new SlopeOneRecommender(
 model,
 Weighting.UNWEIGHTED,
 Weighting.UNWEIGHTED,
 diffStorage);
```

The result is 0.67 -- only slightly worse on this data set.

4.6.3 DiffStorage and memory considerations

Slope-one does have its price, as we noted: memory consumption. In fact, if you tweak the evaluation to use even 10% of all data (about 100,000 ratings), even a 1 gigabyte heap won't be enough. The diffs are used so frequently, and it's so relatively expensive to compute them, that they do need to be computed and stored ahead of time. But, keeping them all in memory can get expensive.

Storage of diffs is encapsulated separately in implementations of `DiffStorage`. We've been using, by default, `MemoryDiffStorage` so far. Not surprisingly, this implementation keeps diffs in memory. It offers one constructor parameter that can trade off some accuracy for slightly less memory consumption: `compactAverages`. This will cause the implementation to use smaller primitive data types to store count, average and standard deviation.

It's worth a try if pressed for memory, but, by that point you will want to look to storing the diffs externally, such as in a database. Fortunately, implementations like `MySQLJDBCDiffStorage` exist for this purpose. It must be used in conjunction with a JDBC-backed `DataModel` implementation like `MySQLJDBCDataModel`, as seen in listing 4.8:

Listing 4.8 Creating a JDBC-backed DiffStorage

```
AbstractJDBCDataModel model = new MySQLJDBCDataModel();
DiffStorage diffStorage = new MySQLJDBCDiffStorage(model);
Recommender recommender = new SlopeOneRecommender(
    model, Weighting.WEIGHTED, Weighting.WEIGHTED, diffStorage);
```

As with `MySQLJDBCDataModel`, the table name and column names used by `MySQLJDBCDiffStorage` can be customized via constructor parameters.

4.6.4 Distributing the precomputation

Precomputing the item-item diffs is significant work. While it is more likely that the size of your data will cause problems with memory requirements before the time required to compute these diffs becomes problematic, you might be wondering if there are ways to distribute this computation to complete faster. Diffs can be updated easily at runtime in response to new information, so, a relatively infrequent offline precomputation process is feasible in this model.

Distributing the diff computation via Hadoop is supported. We will wait until a later chapter, where we introduce all of the Hadoop-related recommender support in Mahout, to explore this process.

4.7 New and experimental recommenders

Mahout also contains implementations of other approaches to recommendation. The three implementations presented briefly below are newer: the implementation may still be evolving, or, the technique may be more recent and experimental. All are worthy ideas, which may yet be useful for use or modification. We won't spend much time here, since they are less central to Mahout's current recommender offerings, but still deserve mention.

4.7.1 Singular value decomposition-based recommenders

Among the most intriguing of these implementations is `SVDRecommender`, based on the singular value decomposition, or SVD. This is an important technique in linear algebra that pops up in machine-learning techniques. Fully understanding it requires some advanced matrix math and understanding of matrix factorization, but this is not necessary to appreciate the SVD's application to recommenders. It is beyond the scope of this book, since there are nearly entire books on the linear algebra behind the SVD, and the SVD algorithm itself.

To attempt to explain the intuition beyond what the SVD does for recommenders, let's say you ask a friend what sort of music she likes, and she lists the following artists:

- Brahms
- Chopin
- Miles Davis
- Tchaikovsky
- Louis Armstrong
- Schumann
- John Coltrane
- Charlie Parker

She might as well have summarized that she likes "classical" and "jazz" music. That communicates less precise information, but not a great deal less. From either statement, you could (probably correctly) infer that she would appreciate Beethoven more than the classic rock band Deep Purple.

Of course, recommender engines operate in a world of many specific data points, not generalities. The input is user preferences for a lot of particular items -- more like the list above rather than our summary. It would be nice to operate on a smaller set of data, all else equal, for reasons of performance. If, for example, iTunes could base its Genius recommendations based not on billions of individual song ratings, but instead millions of ratings of genres, obviously it would be faster -- and, as a basis for recommending music, might not be much worse.

Here, the SVD is the magic that can do the equivalent of the summarization above. It boils down the world of user preferences for individual items to a world of user preferences for more general and less numerous "features" (like genre, above). This is, potentially, a much smaller set of data.

While this process loses some information, it can sometimes improve recommendation results. The process "smooths" the input in useful ways. For example, imagine two car enthusiasts. One loves Corvettes, and the other loves Camaros. We'd like to recommend cars to them. These enthusiasts have similar tastes: both love a Chevrolet sports car. However, in a typical data model for this problem, these two cars would be different items. Without any overlap in their preferences, these two users would be deemed unrelated. However, an SVD-based recommender would perhaps find the similarity. The SVD output may contain features that correspond to concepts like “Chevrolet” or “sports car”, to which both users would be associated. And from the overlap in features, a similarity could be computed.

Using the `SVDRecommender` is as simple as: `new SVDRecommender(model, 10, 10)`. The first numeric argument is the number of features that the SVD should target. There’s no right answer for this; it would be equivalent to the number of genres we might condense someone’s musical taste into, in the previous example. The second argument is the number of “training steps” to run. Think of this as controlling the amount of time it should spend producing this summary; larger values mean longer training.

This approach can give good results (0.66 on our GroupLens data set). At the moment, the major issue with the implementation is that it computes the SVD in memory. This requires the entire data set to fit in memory, and it’s precisely when this isn’t the case that this technique is appealing, since it can “shrink” the input without compromising output quality significantly. In the future, this algorithm will be reimplemented in terms of Hadoop, wherein the necessarily massive SVD computation can be distributed across multiple machines. It is not yet available at this stage of Mahout’s evolution.

4.7.2 Linear interpolation item-based recommendation

This is a somewhat different take on item-based recommendation, implemented as `KnnItemBasedRecommender`. “Knn” is short for “k nearest neighbors”, which is an idea we already saw in the context of `NearestNUserNeighborhood`. This was a `UserNeighborhood` implementation that selected a fixed number of most similar users as a neighborhood of similar users. The algorithm does use the concept of a user neighborhood, but in a different way.

This recommender algorithm still estimates preference values by means of a weighted average of the items the user already has a preference for, but, the weights are not the results of some similarity metric. Instead, the algorithm calculates the optimal set of weights to use between all pairs of items, by means of some linear algebra -- here’s where the linear interpolation comes in. Yes, it is possible to just optimize the weights with some mathematical wizardry.

In reality, it would be very expensive to compute this across all pairs of items, so instead, it first calculates a neighborhood of items most similar to the target item, the one for which a preference is being estimated. It chooses the n nearest neighbors, in much the same way that `NearestNUserNeighborhood` did. One can try this recommender as seen in listing 4.9:

Listing 4.9 Deploying KnnItemBasedRecommender

```
ItemSimilarity similarity = new LogLikelihoodSimilarity(model);
Optimizer optimizer = new NonNegativeQuadraticOptimizer();
return new KnnItemBasedRecommender(model, similarity, optimizer, 10);
```

This will cause the recommender to use a log-likelihood similarity metric to calculate nearest-10 neighborhoods of items. And, it will use a quadratic programming-based strategy to calculate the linear. The details of this are outside the scope of the book.

The implementation is quite functional, but in its current form, is also slow on moderately sized data sets. It should be viewed as viable for small data sets, or for study and extension. On the GroupLens data set, it yields an evaluation result of 0.87.

4.7.3 Cluster-based recommendation

This approach is best thought of as a variant on user-based recommendation. Here, instead of recommending items to users, we recommend items to clusters of similar users. This entails a preprocessing phase, in which all users are partitioned into clusters. Recommendations are then produced for each cluster, such that the recommended items are most interesting to the largest number of users.

The upside of this approach is that recommendation is fast at runtime -- since most everything is precomputed. One could argue that the recommendations are less personal this way, since recommendations are computed for a group rather than an individual. It may be more effective at producing recommendations for new users, with little preference data available. As long as the user can be attached to a reasonably relevant cluster, the recommendations ought to be as good as they will be when more is known about the user.

The name comes from the fact that the algorithm repeatedly joins most-similar clusters into larger clusters, and this implicitly organizes users into a sort of hierarchy, or tree.

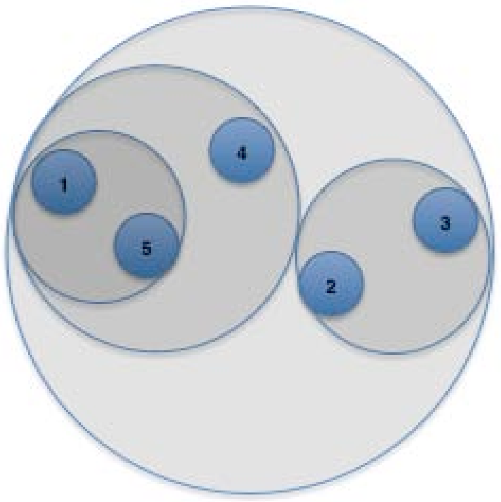


Figure 4.5 An illustration of clustering. Users 1 and 5 are clustered together first, as are 2 and 3, as they are closest. 4 is then clustered with the 1-5 cluster to create a larger cluster, one step up in the “tree”.

Unfortunately, the clustering takes a long time, which you will see if you attempt to run the code in the following listing, which employs a `TreeClusteringRecommender` to implement this idea.

Listing 4.10 Creating a cluster-based recommender

```
UserSimilarity similarity = new LogLikelihoodSimilarity(model);
ClusterSimilarity clusterSimilarity =
    new FarthestNeighborClusterSimilarity(similarity);
return new TreeClusteringRecommender(model, clusterSimilarity, 10);
```

Similarity between users is, as usual, defined by a `UserSimilarity` implementation. Similarity between two clusters of users is defined by a `ClusterSimilarity` implementation. Currently, two implementations are available: one which defines cluster similarity as the similarity between the two most similar user pair, one chosen from each cluster, and another which defines it as the similarity between the two least similar users.

Both are reasonable; in both cases the risk is that one outlier on the edge of a cluster distorts the notion of cluster similarity. Two clusters whose members are on average “distant” but happen to be close at one edge would be considered quite close by the most-similar-user rule, which is implemented by `NearestNeighborClusterSimilarity`. The least-similar-user rule, implemented above as the `FarthestNeighborClusterSimilarity` above, likewise may consider two fairly close clusters to be distant from one another, if each contains an outlier far away from the opposite cluster.

A third approach, to define cluster similarity as the “distance” between the center, or mean, of each cluster, is also possible, though not yet implemented in this part of Mahout.

4.8 Comparing to content-based recommenders

As mentioned in an earlier chapter, “content-based” recommendation is a broad and often-mentioned approach to recommendation, which takes into account the content or attributes of items. For this reason, it is similar to yet distinct from collaborative filtering approaches, which are based on user associations to items only, and treat items as black boxes without attributes. While Mahout largely does not implement content-based approaches, it does offer some opportunities to make use of item attributes in recommendation computations.

4.8.1 Finding content-based recommending in collaborative filtering

For example, consider an online bookseller, who stocks multiple editions of some books. This seller might recommend books to its customers. Its items are books, of course, and it might naturally define a book according to its ISBN number (unique product identifier). However, for a popular public-domain book like Jane Eyre, there may be many printings by different publishers of the same text, under different ISBN numbers. It seems more natural to recommend books based on its text, rather than its particular edition -- do you care more about reading “Jane Eyre” or “Jane Eyre as printed by ACME Publications in 1993 in paperback”? Rather than treat various publications of Jane Eyre as distinct items, it might be more useful to think of the book itself, the text, as the item, and recommend all editions of this book equally. This would be, in a sense, content-based recommendation. By treating the underlying text of a book product, which is its dominant attribute, as the “item” in a collaborative filtering sense, and then applying collaborative filtering techniques with Mahout, they would be engaging in a form of content-based recommendations.

Or, recall that item-based recommenders require some notion of similarity between two given items. This similarity is encapsulated by an `ItemSimilarity` implementation. So far we’ve seen implementations that derive similarity from user preferences only -- this is classic collaborative filtering. However, there’s no reason the implementation could not be based on item attributes. For example, a movie recommender might define an item (movie) similarity as a function of movie attributes like genre, director, actors and actresses, and year of release. Using such an implementation within a traditional item-based recommender would also be an example of content-based recommendation.

4.8.2 Looking deeper into content-based recommendation

Taking this a step further, imagine content-based recommendation as a generalization of collaborative filtering. In collaborative filtering, computations are based on preferences, which are user-item associations. But what drives these user-item associations? It's likely that users have implicit preferences for certain item attributes, which come out in their preferences for certain items and not others. For example, if your friend told you she likes the albums Led Zeppelin I, Led Zeppelin II and Led Zeppelin III, you might well guess she is actually expressing a preference for an attribute of these items: the band Led Zeppelin. By discovering these associations, and discovering attributes of items, it's possible to construct recommender engines based on these more nuanced understandings of user-item associations.

These techniques come to resemble search and document retrieval techniques: asking what items a user might like based on user-attribute associations and item attributes resembles retrieving search results based on query terms and occurrence of terms in documents. While Mahout's recommender support does not yet embrace these techniques, it is a natural direction for future versions to address.

4.9 Comparing to model-based recommenders

Another future direction for Mahout is model-based recommendation. This family of techniques attempts to build some model of user preferences, based on existing preferences, and then infer new preferences. These techniques generally fall into the broader category of collaborative filtering, as they typically derive from user preferences only.

The "model" might be a probabilistic picture of users' preferences, in the form of a Bayesian network for example. The algorithm then attempts to judge the probability of liking an item given its knowledge of all user preferences, and ranks recommendations accordingly.

Association rule learning can be applied in a similar sense to recommendations. By learning "rules" such as "when a user prefers item X and item Y, he or she will prefer item Z" from the data, and judging confidence in the reliability of such rules, a recommender can put together the most likely set of new, preferred items.

Cluster-based recommenders might be considered a type of model-based recommender. The clusters represent a model of how users group together and therefore how their preferences might run the same way. In this limited sense, Mahout supports model-based recommenders. However, this is an area that is still largely under construction in Mahout as of this writing.

4.10 Summary

In this chapter, we thoroughly explored the core recommender algorithms offered by Mahout. We started by explaining the general user-based recommender algorithm in terms of real-world reasoning.

From there, we looked at how this algorithm is realized in Mahout, as `GenericUserBasedRecommender`. Many pieces of this generic approach can be customized, such as the definition of user similarity and user neighborhood.

We looked at the "classic" user similarity metric, based on the Pearson correlation, noted some possible issues with this approach, and responses such as weighting. We looked at similarity metrics based on the Euclidean distance, Spearman correlation, Tanimoto coefficient and a log-likelihood ratio.

Then, we looked at the other canonical recommendation technique, item-based recommendation, as implemented by `GenericItemBasedRecommender`. It is conceptually quite similar and reuses some concepts already covered in the context of user-based recommender, such as the Pearson correlation.

Next, we examined a slope-one recommender, a unique and relatively simple approach to recommendation based on average differences in preference values between items. It requires significant precomputation and storage for these diffs, and so we explored how to store these both in memory and in a database.

Last, we looked briefly at a few newer, more experimental implementations currently in the framework. These include implementations based on the singular value decomposition, linear interpolation, and clustering. These may be useful for small data sets, or academic interest, as they are still a work in progress.

The key parameters and features for each implementation are summarized in table 4.9 below.

Table 4.9 Summary of available recommender implementations, their key input parameters, and key features to consider when choosing an implementation.

Implementation	Key Parameters	Key Features
GenericUserBasedRecommender	User similarity metric	"Conventional" implementation
	Neighborhood definition and size	Fast when number of users is relatively smaller
GenericItemBasedRecommender	Item similarity metric	Fast when number of items is relatively smaller
		Useful when an external notion of item similarity is available
SlopeOneRecommender	Diff storage strategy	Recommendations and updates fast at runtime
		Requires large precomputation
		Suitable when number of items is relatively small
SVDRecommender	Number of features	Good results
		Requires large precomputation
KnnItemBasedRecommender	Number of means ("k")	Good when number of items is relatively smaller
	Item similarity metric	
	Neighborhood size	
TreeClusteringRecommender	Number of clusters	Recommendations are fast at runtime
	Cluster similarity definition	Requires large precomputation
	User similarity metric	

Good when number of users is relatively smaller

We are done introducing Mahout's recommender engine support. Now we're ready to examine even larger and more realistic data sets, from the practitioner's perspective. You might wonder why we've made little mention of Hadoop yet. Hadoop is a powerful tool, and necessary when dealing with massive data sets, where one must make use of many machines. This has drawbacks: such computations are massive, resource-intensive, and complete in hours, not milliseconds. We will reach Hadoop in the last chapter in this section. First, in the next chapter, we will explore productionizing a recommender engine based on Mahout that fits onto one machine, one that can respond to requests for recommendations in a fraction of a second and incorporate updates immediately.

5 Taking Recommenders to Production

This chapter covers

- Analyzing data from a real dating site
- Designing and refining a recommender engine solution
- Deploying a web-based recommender service in production

So far, we have toured the recommender algorithms and variants that Mahout provides. We've seen how to evaluate the accuracy and performance of a recommender. Now, we will apply all of this to a real data set, taken from a dating site, to create an effective recommender engine from scratch based on data. Then we'll take the recommender the final step, to a deployable production-ready web service.

There is no one standard approach to building a recommender for given data and a given problem domain. The data must at least represent associations between users and items -- where "users" and "items" might be many things. Adapting the input to recommender algorithms is usually quite a problem-specific process. Discovering the best recommender engine to apply to the input data is likewise specific to each context. It inevitably involves hands-on exploration, experimentation, and evaluation on real problem data.

This chapter will present one end-to-end example that suggests the process you might take to develop a recommender system for your data set. We will try an approach, collect data, try to understand the results, and repeat many times. Many approaches won't get us anywhere, but that's good information as well. This "brute force" approach is appropriate, since it's relatively painless to evaluate an approach, and because here, as in other problem domains, it's not at all clear what the right approach is from just looking at the data.

5.1 Dating data from libimseti.cz

We will use a new data set, derived from the Czech dating site Líbímseti (http://libimseti.cz/). Users of this site are able to rate other users' profiles, on a scale of 1 to 10. A 1 means "NELÍBÍ", or "dislike", and a 10 means “LÍBÍ”, or “like”. From the presentation of profiles on the site, we infer that users of such a site are expressing some assessment of the profiled user’s appeal, attractiveness, and “dateability”. A great deal of this data has been anonymized, made available for research[6], and published by Vaclav Petricek (http://www.occamslab.com/petricek/data/). Because we will be using the data in this chapter, please obtain a copy of the data from this link[7].

With 17,359,346 ratings in the data set, this is almost twice as big as our previous data set. It contains users’ explicit ratings for “items”, where items are here other people’s user profiles. That means a recommender system built on this data will be recommending people to people. It’s a reminder to think broadly about recommenders, which aren’t limited to recommending objects like books and DVDs.

5.1.1 Analyzing the input data

The first step is analyzing what data is available to work with, and beginning to form ideas about which recommender algorithm could be suitable to use with it. The `ratings.dat` file in the archive you downloaded is a simple comma-delimited file containing user ID, profile ID, and rating. Each line represents one user’s rating of one other user’s profile. The data is purposely obfuscated, so we can’t assume that the user IDs are real user IDs from the site. Profiles are user profiles, and so this data represents users’ ratings of other users. One might suppose that user IDs and profile IDs are comparable here, that user ID 1 and profile ID 1 are the same user. This does not appear to be the case, likely for reasons of anonymity. We can’t make this assumption about the input.

There are 135,359 unique users in the data, who together rated 168,791 unique user profiles. Because the number of users and items are about the same, neither user-based nor item-based recommendation is obviously more efficient. If there had been a great deal more profiles than users, then an item-based recommender would have been relatively slower. Slope-one can be applied here, even though its memory requirements scale up quickly as the number of items. As we will see, its memory requirements can be limited.

We also note that the data set has been pre-processed in a way: no users that produced less than 20 ratings are included. In addition, users who seem to have rated every profile with the same value are also excluded, presumably because it may be spam, or an unserious attempt at rating. The data we do have comes from users who bothered to make a number of ratings; presumably, their input is useful, and not “noisy”, compared to the ratings of less-engaged users.

This input is already formatted for use with Mahout’s `FileDataModel`. The user and profile IDs are numeric, and, the file is already comma-delimited with fields in the required order: user ID, item ID, preference value.

5.1.2 Incorporating gender information

The data set provides another interesting set of data: the gender of the user for many of the profiles in the data set. We are not given the gender of all profiles; in `gender.dat`, several lines end in “U” which means “unknown”. We are also not given the gender of the users in the data set -- just the profiles.

[6] See also http://www.occamslab.com/petricek/papers/dating/brozovsky07recommender.pdf
[7] Neither the site nor publisher of the data endorse or are connected with this book.

However that means we know something more about each of our items. Male profiles are much more similar to one another than female profiles -- at least, in the context of being recommended as potential dates. If we see that most or all of a user's ratings are for male profiles, it stands to reason that the user will rate male profiles as far more desirable dates than female. We might view this information as the basis for an item-item similarity metric.

This isn't a perfect assumption. Without becoming too sidetracked on sensitive issues of sexuality, we note that some users of the site may enjoy rating profiles of a gender they are not interested in dating, for fun. Some users may legitimately have some romantic interest in both genders. In fact, the very first two ratings in `ratings.dat` are from one user, and yet appear to be for profiles of different genders.

It's important to account for gender in a dating site recommender engine like this; it would be quite bad to recommend a female to a user interested only in males -- this would surely be viewed as a bad recommendation, and to some, offensive. This restriction is important, but doesn't fit neatly into the standard recommender algorithms. Later in the chapter, we'll examine how to inject this information as both a filter, and a similarity metric.

5.2 Finding an effective recommender

To create a complete recommender engine for Líbímseti data, we will need to choose from among the many implementations we've seen already. Our recommender ought to be both fast and produce good recommendations. Of those two, it's better to focus on producing good recommendations first, and then look to performance. After all, what's the use in producing bad answers quickly?

We can't possibly deduce the right implementation from looking at the data; some empirical testing is needed. Armed with an evaluation framework, we set about collecting some data.

5.2.1 User-based recommenders

User-based recommenders are a natural first stop. We can use several different similarity metrics and neighborhood definitions. To get some sense of what works and doesn't, we can try many combinations. The result of some such experimenting in our test environment is summarized in tables 5.1 and 5.2, and figures 5.1 and 5.2.

Table 5.1 Average absolute difference in estimated and actual preference, when evaluating a user-based recommender using one of several similarity metrics, and using a nearest-n user neighborhood

n =	1	2	4	8	16	32	64	128
Euclidean	1.17	1.12	1.23	1.25	1.25	1.33	1.48	1.43
Pearson	1.30	1.19	1.27	1.30	1.26	1.35	1.38	1.47
Log-likelihood	1.33	1.38	1.33	1.35	1.33	1.29	1.33	1.49
Tanimoto	1.32	1.33	1.43	1.32	1.30	1.39	1.37	1.41

Table 5.2 Average absolute difference in estimated and actual preference, when evaluating a user-based recommender using one of two similarity metrics, and using a threshold-based user neighborhood

t =	0.95	0.9	0.85	0.8	0.75	0.7
Euclidean	1.33	1.37	1.39	1.43	1.41	1.47
Pearson	1.47	1.4	1.42	1.4	1.38	1.37
Log-likelihood	1.37	1.46	1.56	1.52	1.51	1.43
Tanimoto	NaN	NaN	NaN	NaN	NaN	NaN

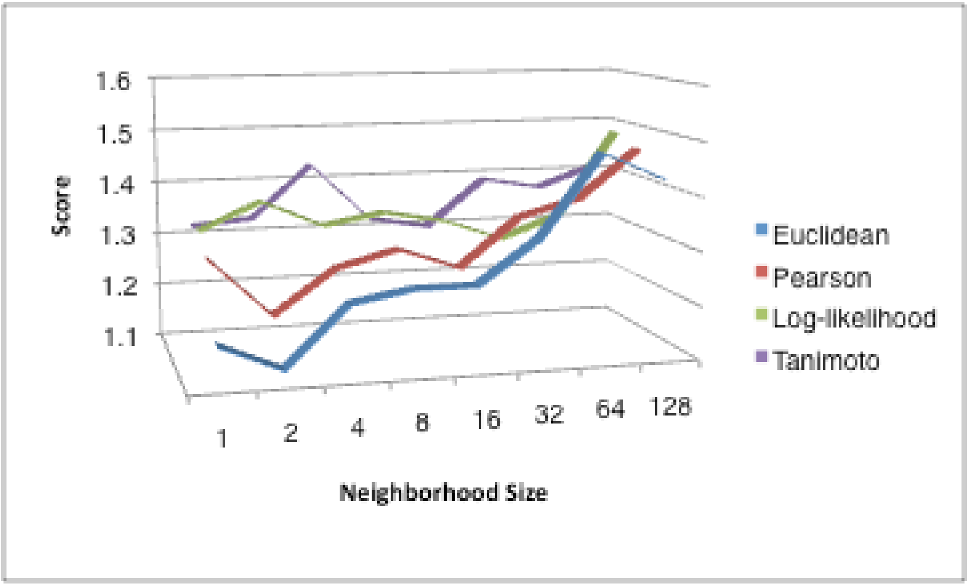


Figure 5.1 Visualization of values in table 5.1.

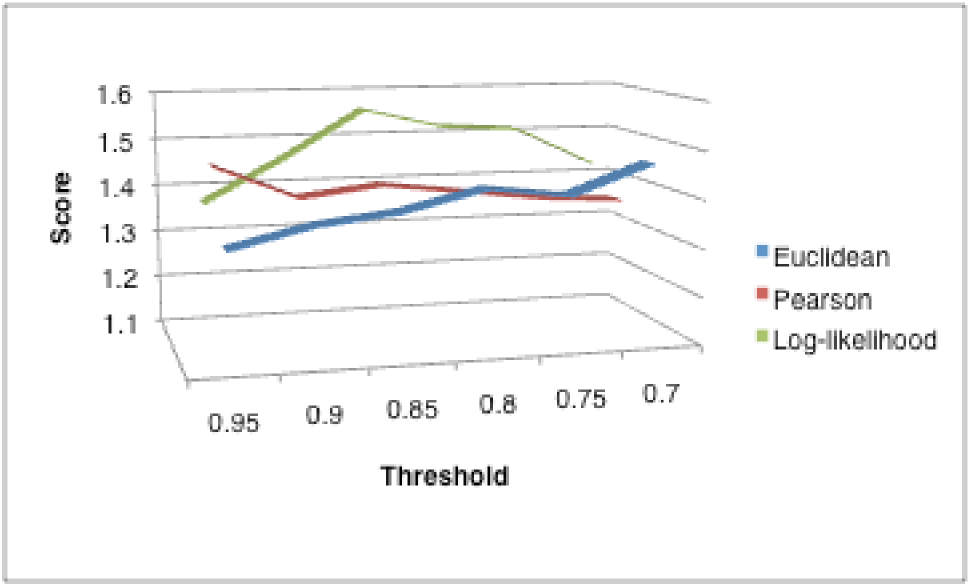


Figure 5.2 Visualization of table 5.2.

These scores aren’t bad. These recommenders are estimating user preferences to within 1.12 to 1.56 points on a scale of 1 to 10, on average.

There are some trends here, even though some individual evaluation results vary from that trend. It looks like the Euclidean distance similarity metric may be a little better than Pearson, though their results are quite similar. It also appears that using a small neighborhood is better than a large one; the best evaluations occur when using a neighborhood of two people! Maybe users’ preferences are truly quite personal, and incorporating too many others in the computation doesn’t help.

What explains the “NaN” result for the Tanimoto coefficient-based similarity metric? It is listed here to highlight a subtle point about this methodology. Although all similarity metrics return a value between -1 and 1, and return higher values to indicate greater similarity, it’s not true that any given value “means” the same thing for each similarity metric. For example, 0.5 from a Pearson correlation-based metric indicates moderate similarity. However, 0.5 for the Tanimoto coefficient indicates significant similarity between two users: of all items known to either of them, half are known to both.

Even though thresholds of 0.7 to 0.95 were reasonable values to test for the other metrics, these are quite high for a Tanimoto coefficient-based similarity metric. In each case, the bar was set so high that no user neighborhood was established in any test case! Here, we might have more usefully tested thresholds from, say, 0.4 on down. In fact, with a threshold of 0.3, the best evaluation score approaches 1.2.

Similarly, although we see an apparent best value for n in the nearest-n user neighborhood data, we don’t quite see the same in the threshold-based user neighborhood results. For example, the Euclidean-distance-based similarity metric seems to be producing better results as the threshold increases. Perhaps the most valuable users to include in the neighborhood have a Euclidean-based similarity of over 0.95. What happens at 0.99? 0.999? The evaluation result goes down to about 1.35; not bad, but not apparently the best recommender.

We leave it as an exercise to the dedicated reader to continue looking for even better configurations. For our purposes, we will take the current best solution to be:

- User-based recommender
- Euclidean distance similarity metric
- Nearest-2 neighborhood

5.2.2 Item-based recommenders

Item-based recommenders involve fewer choices: we need only choose an item similarity metric. We can easily try each similarity metric and see what works best. Again, table 5.3 summarizes the outcome.

Table 5.3 Average absolute differences in estimated and actual preference, when evaluating an item-based recommender using several different similarity metrics.

	Score
Euclidean	2.36
Pearson	2.32
Log-likelihood	2.38
Tanimoto	2.40

Scores are notably worse here; the average error, or difference between estimated and actual preference value, has roughly doubled to over 2. For this data, the item-based approach isn't as effective, for some reason. We could speculate as to why. Before, we computed similarities between users in a user-based approach, based on how users rated other users' profiles. Now, we're computing similarity between user profiles based on how other users rated that profile. Maybe this isn't as meaningful -- maybe ratings tell us more about the rater than the rated profile. Whatever the explanation, it seems clear from these results that item-based recommendation isn't the best choice here.

5.2.3 Slope-one recommender

Recall that the slope-one recommender constructs a "diff" for most item-item pairs in the data model. With 168,791 items (profiles) here, this means storing potentially 28 billion diffs -- far too much to fit in memory. Storing these diffs in a database is possible, but will greatly slow performance. In fact, we have another option, which is to ask the framework to limit the number of diffs stored to perhaps ten million, as seen in listing 5.1. It will attempt to choose the most useful diffs to keep. "Most useful" here means those diffs between a pair of items that turn up most often together in the list of items associated to a user. For example, if items A and B appear in the preferences of hundreds of users, the average diff in their preference values is likely significant, and useful. If A and B only appear together in the preferences of one user, it sounds more like a fluke than a piece of data worth storing.

Listing 5.1 Limiting memory consumed by MemoryDiffStorage

```
DiffStorage diffStorage = new MemoryDiffStorage(
    model, Weighting.WEIGHTED, true, 10000000L);
return new SlopeOneRecommender(
    model, Weighting.WEIGHTED, Weighting.WEIGHTED, diffStorage);
```

Indeed, from examining the log output, this keeps memory consumption to about 1.5GB. You'll also notice again how fast slope-one is; on the workstation used for testing, we saw average recommendation times under 10 milliseconds, compared to 200 milliseconds or so for other algorithms.

The evaluation result is about 1.41. This is not a bad result, but not quite as good a result as we observed with user-based recommenders above. It is likely not worth pursuing slope-one for this particular data set.

5.2.4 Evaluating precision and recall

Above, we did experiment with a Tanimoto coefficient-based similarity metric and log-likelihood-based metric, and as we know these are metrics which do not use preference values. However, we did not yet examine recommenders that completely ignore rating values. Such recommenders can't be evaluated in the same way -- there are no estimated preference values to evaluate against real values, because there are no preference values at all. It's possible to examine precision and recall of such recommenders versus the current best solution: user-based recommender, Euclidean distance metric, and nearest-2 neighborhood.

We can evaluate the precision and recall of this recommender engine as seen in previous chapters, using a `RecommenderIRStatsEvaluator`. It reveals that precision and recall at 10 are about 3.6% and 5%, respectively. This seems low: the recommender rarely recommends the users' own top-rated profiles, when those top-rated profiles are removed. In this context, that's not obviously a bad thing. It's conceivable that a user might see plenty of "perfect 10s" on such a dating site, and perhaps has only ever encountered and rated some of them. It could be that the recommender is suggesting even more desirable profiles than the user has seen! Certainly, this is what the recommender is communicating, that the users' top-rated profiles aren't usually the ones they would like most, were they to actually review every profile in existence.

The other explanation, of course, is that the recommender isn't functioning well. However we know this recommender is fairly good at estimating preference values, usually estimating ratings within about 1 point on a 10-point scale. So this explanation could be valid.

An interesting thing happens when we switch to ignore rating data by using `GenericBooleanPrefDataModel`, `GenericBooleanPrefUserBasedRecommender`, and an appropriate similarity metric like `LogLikelihoodSimilarity`. Precision and recall increase to over 22% in this case. Similar results are seen with `TanimotoCoefficientSimilarity`. It seems better on the surface; what the result says it that this sort of recommender engine is better at recommending back those profiles which the user might already have encountered. If we had reason to believe users had in fact reviewed a large proportion of all profiles, then their actual top ratings would be a strong indicator of what the "right" answers are. This does not seem to be the case on a dating site with hundreds of thousands of profiles.

In other contexts, a high precision and recall figure may be important. Here, it does not seem to be as important. For our purposes here, we will move forward with the previous user-based recommender, with Euclidean distance similarity and nearest-2 neighborhood, instead of opting to switch to one of these other recommenders.

5.2.5 Evaluating Performance

Finally we should look at the runtime performance of this recommender engine that we have identified. Because we intend to call it in real-time, it would do little good to produce a recommender that needs minutes to compute a recommendation!

The `LoadEvaluator` class can be used, as before, to assess per-recommendation runtime. We ran this recommender on the data set with flags "`-server -d64 -Xmx2048m -XX:+UseParallelGC -XX:+UseParallelOldGC`" and found an average recommendation time of 218 milliseconds on our test machine. The application consumes only about a gigabyte of heap at runtime. Whether or not these values are acceptable or not will depend on application requirements and available hardware. These figures seem reasonable for many applications.

5.3 Injecting domain-specific information

So far we've not taken advantage of any domain-specific knowledge here. We have used the user-profile rating data as if it could be anything at all -- ratings for books or cars or fruit. Here we look at how we could incorporate additional information we have in this domain to improve recommendation.

5.3.1 Employing a custom item similarity metric

Because we know the gender of many profiles, we could create a simple similarity metric for pairs of profiles based only on gender. Profiles are items, so this would be an `ItemSimilarity` in the framework. For example, we could call two male or two female profiles "very similar" and assign them a similarity of 1.0. We could say the similarity between a male and female profile is -1.0. Finally, we could assign a 0.0 to profile pairs where the gender of one or both is unknown.

The idea is simple, perhaps overly simplistic. It would be fast, but would discard all rating-related information from the metric computation. For the sake of experimentation, let's try it out with an item-based recommender.

Listing 5.2 A gender-based item similarity metric

```
public class GenderItemSimilarity implements ItemSimilarity {

  private final FastIDSet men;
  private final FastIDSet women;

  public GenderItemSimilarity(FastIDSet men, FastIDSet women) {
    this.men = men;
    this.women = women;
  }

  @Override
  public double itemSimilarity(long profileID1, long profileID2) {
    Boolean profile1IsMan = isMan(profileID1);
    if (profile1IsMan == null) {
      return 0.0;
    }
    Boolean profile2IsMan = isMan(profileID2);
    if (profile2IsMan == null) {
      return 0.0;
    }
    return profile1IsMan == profile2IsMan ? 1.0 : -1.0;
  }

  private Boolean isMan(long profileID) {
    if (men.contains(profileID)) {
      return Boolean.TRUE;
    }
    if (women.contains(profileID)) {
      return Boolean.FALSE;
    }
    return null;
  }

  @Override
  public void refresh(Collection<Refreshable> alreadyRefreshed) {
    // do nothing
  }

}
```

We can pair this `ItemSimilarity` metric with a standard `GenericItemBasedRecommender`, as before, and evaluate its accuracy. The concept is interesting, but the result here is not better than with other metrics: 2.35. If we had more information available, such as the interests and hobbies expressed on each profile, we could construct a more meaningful similarity metric that might yield better results.

This example, however, illustrates the main advantage of item-based recommenders: it provides a means to incorporate information about items themselves, which is commonly available in recommender problems. From the evaluation results, you also perhaps noticed how this kind of recommender is fast when based on such an easy-to-compute similarity metric; on our test machine, recommendations were produced in about 15 milliseconds on average.

5.3.2 Recommending based on content

If you blinked, you might have missed it -- we just saw an example of content-based recommendation in the last section. Above, we added a notion of item similarity that was not based on user preferences, but that was based on attributes of the item itself.

This is a simple but legitimate instance of a content-based recommendation technique. As stated above, it's a powerful addition to pure collaborative filtering approaches, which are based only on user preferences. We can usefully inject our knowledge about items (here, people) to augment the user preference data we have, and hopefully produce better recommendations.

Unfortunately the item similarity metric above is specific to the problem domain at hand. This metric doesn't help recommendations in other domains: recommending food, or movies, or travel destinations. This is why it's not part of the framework. But, it is a feasible and powerful approach any time you have domain-specific knowledge beyond user preferences about how items are related.

5.3.3 Modifying recommendations with IDRescorer

You may have observed an optional, final argument to the `Recommender.recommend()` method of type `IDRescorer`; instead of calling `recommend(long userID, int howMany)`, you can call `recommend(long userID, int howMany, IDRescorer rescorer)`. These objects show up in several parts of the Mahout recommender-related APIs. Implementations can transform values used in the recommender engine to other values based on some logic, or else exclude an entity from consideration in some process. For example, an `IDRescorer` may be used to arbitrarily modify a `Recommender`'s estimated preference value for an item. It can also remove an item from consideration entirely.

For example, suppose you were recommending books to a user on an e-commerce site. The user in question is currently browsing mystery novels. So, when recommending books to that user at that moment, you might wish to boost estimated preference values for all mystery novels. You may also wish to ensure that no out-of-stock books are recommended. An `IDRescorer` can help you do this. Below in listing 5.3 is an `IDRescorer` implementation that encapsulates this logic in term some classes from this fictitious bookseller:

Listing 5.3 Example IDRescorer that omits out-of-stock books and boosts a genre

```
public class GenreRescorer implements IDRescorer {

  private final Genre currentGenre;

  public GenreRescorer(Genre currentGenre) {
    this.currentGenre = currentGenre;
  }

  public double rescore(long itemID, double originalScore) {
    Book book = BookManager.lookupBook(itemID); A
    if (book.getGenre().equals(currentGenre)) {
      return originalScore * 1.2; B
    }
    return originalScore; C
  }

  public boolean isFiltered(long itemID) {
    Book book = BookManager.lookupBook(itemID);
    return book.isOutOfStock(); D
  }
}
```

A Assume we have some BookManager with this method available
B Boost estimated preference for matching genre books by 20%
C Don't change anything else
D Filter out books that are not in stock now

The `rescore()` method boosts estimated preference value for mystery novels. The `isFiltered()` method demonstrates the other use of `IDRescorer`: it ensures that no out-of-stock books are considered for recommendation. This is merely an example, and not relevant to our dating site. Let's turn to apply this idea with the extra data we do have: gender.

5.3.4 Incorporating gender in an IDRescorer

We can use an `IDRescorer` to filter out "items", or user profiles, for users whose gender may not be of romantic interest. We can do this by first guessing the user's preferred gender by examining the gender of profiles rated so far. Then, we filter out profiles of the opposite gender, as seen in listing 5.4.

Listing 5.4 Gender-based rescoring implementation

```
public class GenderRescorer implements IDRescorer {

  private final FastIDSet men;
  private final FastIDSet women;
  private final FastIDSet usersRateMoreMen; A
  private final FastIDSet usersRateLessMen;
  private final boolean filterMen;

  public GenderRescorer(FastIDSet men,
                        FastIDSet women,
                        FastIDSet usersRateMoreMen,
                        FastIDSet usersRateLessMen,
                        long userID, DataModel model)
      throws TasteException {
    this.men = men;
    this.women = women;
    this.usersRateMoreMen = usersRateMoreMen;
    this.usersRateLessMen = usersRateLessMen;
    this.filterMen = ratesMoreMen(userID, model);
  }

  public static FastIDSet[] parseMenWomen(File genderFile)
      throws IOException { B
    FastIDSet men = new FastIDSet(50000);
    FastIDSet women = new FastIDSet(50000);
    for (String line : new FileLineIterable(genderFile)) {
      int comma = line.indexOf(',');
      char gender = line.charAt(comma + 1);
      if (gender == 'U') {
        continue;
      }
      long profileID = Long.parseLong(line.substring(0, comma));
      if (gender == 'M') {
        men.add(profileID);
      } else {
        women.add(profileID);
      }
    }
    men.rehash(); C
    women.rehash();
    return new FastIDSet[] { men, women };
  }

  private boolean ratesMoreMen(long userID, DataModel model)
      throws TasteException {
    if (usersRateMoreMen.contains(userID)) {
      return true;
    }
    if (usersRateLessMen.contains(userID)) {
      return false;
    }
    PreferenceArray prefs = model.getPreferencesFromUser(userID);
    int menCount = 0;
    int womenCount = 0;
    for (int i = 0; i < prefs.length(); i++) {
      long profileID = prefs.get(i).getItemID();
      if (men.contains(profileID)) {
        menCount++;
      } else if (women.contains(profileID)) {
        womenCount++;
      }
    }
    boolean ratesMoreMen = menCount > womenCount; D
    if (ratesMoreMen) {
      usersRateMoreMen.add(userID);
    } else {
      usersRateLessMen.add(userID);
    }
    return ratesMoreMen;
  }

  @Override
  public double rescore(long profileID, double originalScore) {
    return isFiltered(profileID) ? Double.NaN : originalScore; E
  }

  @Override
  public boolean isFiltered(long profileID) {
    return filterMen ? men.contains(profileID) : women.contains(profileID);
  }

}
```

A Cache assessment of which users rate more male profiles
B Will be called separately later
C Optimizes data structure again for fast access
D Users rating more men probably like male profiles
E Return NaN for profiles that should be excluded

A few things are happening in this code example. The method `parseMenWomen()` will parse `gender.dat` and create two sets of profile IDs -- those that are known to be men, and those known to be women. This is parsed separately from any particular instance of `GenderRescorer` since these sets will be reused many times. `ratesMoreMen()` will be used to determine and remember whether a user seems to rate more male or female profiles. These results are cached in two additional sets. Instances of this `GenderRescorer` will then simply filter out men, or women, as appropriate, by returning `NaN` from `rescore()`, or `true` from `isFiltered()`.

This ought to have some small but helpful effect on the quality of recommendations. Presumably, women who rate male profiles are already being recommended male profiles, because they will be most similar to other women who rate male profiles, and will be recommended those profiles. This mechanism will ensure this, by filtering female profiles from results. It will cause the Recommender to not even attempt to estimate these women's preference for female profiles because such an estimate is quite a guess, and wrong. Of course, the effect of this `IDRescorer` is limited by the quality of data available: we only know the gender of about half of the profiles.

5.3.5 Building a custom Recommender around an IDRescorer

It will be useful for our purposes here to wrap up our entire, current recommender engine, plus the new `IDRescorer`, into one implementation. This will become necessary in the next section when we need to deploy one self-contained recommender engine to production. Listing 5.5 shows a Recommender implementation that contains inside it the user-based recommender engine we've identified as best suited to our data set.

Listing 5.5 Complete recommender implementation for Líbímseti

```
  public class LibimsetiRecommender implements Recommender {

    private final Recommender delegate;
    private final DataModel model;
    private final FastIDSet men;
    private final FastIDSet women;

    public LibimsetiRecommender() throws TasteException, IOException {
```

```
    this(new FileDataModel(
        RecommenderWrapper.readResourceToTempFile("ratings.dat")); A
  }

  public LibimsetiRecommender(DataModel model)
      throws TasteException, IOException {
    UserSimilarity similarity = new EuclideanDistanceSimilarity(model); B
    UserNeighborhood neighborhood =
        new NearestNUserNeighborhood(2, similarity, model);
    delegate =
        new GenericUserBasedRecommender(model, neighborhood, similarity);
    this.model = model;
    FastIDSet[] menWomen = GenderRescorer.parseMenWomen(
        RecommenderWrapper.readResourceToTempFile("gender.dat"));
    men = menWomen[0];
    women = menWomen[1];
  }

  @Override
  public List<RecommendedItem> recommend(long userID, int howMany)
      throws TasteException {
    IDRescorer rescorer = new GenderRescorer(men, women, userID, model); C
    return delegate.recommend(userID, howMany, rescorer);
  }

  @Override
  public List<RecommendedItem> recommend(long userID,
                                         int howMany,
                                         IDRescorer rescorer)
      throws TasteException {
    return delegate.recommend(userID, howMany, rescorer);
  }

  @Override
  public float estimatePreference(long userID, long itemID)
      throws TasteException {
    IDRescorer rescorer = new GenderRescorer(men, women, userID, model); D
    return (float) rescorer.rescore(
        itemID, delegate.estimatePreference(userID, itemID));
  }

  @Override
  public void setPreference(long userID, long itemID, float value)
      throws TasteException {
    delegate.setPreference(userID, itemID, value); E
  }

  @Override
  public void removePreference(long userID, long itemID)
      throws TasteException {
    delegate.removePreference(userID, itemID);
  }

  @Override
  public DataModel getDataModel() {
    return delegate.getDataModel();
  }

  @Override
```

```
    public void refresh(Collection<Refreshable> alreadyRefreshed) {
      delegate.refresh(alreadyRefreshed);
    }

  }
```

A Will need readResourceToTempFile() when deploying in production
B Construct the same user-based recommender inside
C Force all recommendations to use our GenderRescorer
D Rescore estimated preferences too
E Delegate everything else to underlying user-based recommender

This is a tidy, self-contained packaging of the recommender engine. We can evaluate the entire thing, as before. The result is about 1.18: virtually unchanged, though we might feel better with this mechanism in place that ought to avoid some seriously undesirable recommendations. Running time has increased to 500 milliseconds or so. The rescoring has added significant overhead. For our purposes, we will accept this tradeoff and continue forward with `LibimsetiRecommender` as our final implementation for this dating site.

5.4 Recommending to anonymous users

Since we are talking about recommenders in practice, this is a good place to discuss how to handle a common real-world issue: recommending to users that aren't users yet. What can be done, for instance, for the new user browsing products in an e-commerce web site? This anonymous user has no browsing or purchase history, let alone an ID, as far as the site is concerned. It is nevertheless valuable to be able to recommend products to such a user.

One approach is to not bother personalizing the recommendations. That is, when presented with a new user, present a general predefined list of products to recommend. It's simple, and usually better than nothing.

At the other end of the spectrum, a site could promote such anonymous users to real users on first visit, and assign an ID and track his or her activity merely based on a web session. This is also works, though potentially explodes the number of users, who by definition may never return and for whom little information exists.

5.4.1 Temporary users with PlusAnonymousUserDataModel

The recommender framework offers a simple way to temporarily add an anonymous user's information into the `DataModel`: `PlusAnonymousUserDataModel`. This approach treats anonymous users like real users, but only for as long as it takes to make recommendations. They are never added to or known to the real underlying `DataModel`. It is a wrapper around any existing `DataModel` and is simply a drop-in replacement

This class has a spot for one temporary user, and can hold preferences for one such user at a time. As such, a `Recommender` based on this class must only operate on one anonymous user at a time.

Listing 5.6 presents `LibimsetiWithAnonymousRecommender`, which extends the previous `LibimsetiRecomender` with a method that can recommend to an anonymous user. It takes preferences as input rather than a user ID, of course.

Listing 5.6 Anonymous user recommendation for Líbímseti

```
  public class LibimsetiWithAnonymousRecommender
```

```
    extends LibimsetiRecommender {

  private final PlusAnonymousUserDataModel plusAnonymousModel;

  public LibimsetiWithAnonymousRecommender()
      throws TasteException, IOException {
    this(new FileDataModel(
        RecommenderWrapper.readResourceToTempFile("ratings.dat")));
  }

  public LibimsetiWithAnonymousRecommender(DataModel model)
      throws TasteException, IOException {
    super(new PlusAnonymousUserDataModel(model)); A
    plusAnonymousModel =
        (PlusAnonymousUserDataModel) getDataModel();
  }

  public synchronized List<RecommendedItem> recommend( B
      PreferenceArray anonymousUserPrefs, int howMany)
      throws TasteException {
    plusAnonymousModel.setTempPrefs(anonymousUserPrefs);
    List<RecommendedItem> recommendations =
        recommend(PlusAnonymousUserDataModel.TEMP_USER_ID, howMany, null);
    plusAnonymousModel.setTempPrefs(null);
    return recommendations;
  }

  public static void main(String[] args) throws Exception {
    PreferenceArray anonymousPrefs =
        new GenericUserPreferenceArray(3); D
    anonymousPrefs.setUserID(0,
        PlusAnonymousUserDataModel.TEMP_USER_ID);
    anonymousPrefs.setItemID(0, 123L);
    anonymousPrefs.setValue(0, 1.0f);
    anonymousPrefs.setItemID(1, 123L);
    anonymousPrefs.setValue(1, 3.0f);
    anonymousPrefs.setItemID(2, 123L);
    anonymousPrefs.setValue(2, 2.0f);
    LibimsetiWithAnonymousRecommender recommender =
        new LibimsetiWithAnonymousRecommender();
    List<RecommendedItem> recommendations =
        recommender.recommend(anonymousPrefs, 10);
    System.out.println(recommendations);
  }

}
```

A Wraps the underlying DataModel
B Note synchronization
C TEMP_USER_ID is anonymous user's "ID"
D Example anonymous user prefs

This implementation otherwise walks and talks like a `Recommender` and may be used to recommend to real users as well.

5.4.2 Aggregating anonymous users

Finally, we note that it is also possible to treat all anonymous users as if they are one user. This simplifies things. Rather than track those potential users browsing a site separately and storing their browsing histories individually, one could think of all such users as like one big “tire-kicking” user. This depends upon the assumption that all such users behave meaningfully similarly.

At any time, the technique above can produce recommendations for the anonymous user. This is fast. In fact, since the result is the same for all anonymous users, the set of recommendations can be stored and recomputed periodically instead of upon every request. In a sense, this variation nearly reduces to not personalizing recommendations, and just presenting anonymous users with a fixed set of recommendations.

5.5 Creating a web-enabled service

Creating a recommender that runs in your IDE is fine, but chances are you are interested in deploying this recommender in a production application. Of course, if your application is written in Java, you can directly include the Mahout library and your implementation, and call to the `Recommender` implementation however you like. This is quite flexible.

5.5.1 Constructing a servlet container

However, you may wish to deploy a recommender as a stand-alone component of your application architecture, rather than embed it inside your application code. It is common for services to be exposed over the web, via simple HTTP or web services protocols like SOAP. In this scenario, a recommender is deployed as a web-accessible service as an independent component in a web container, or even as its own server process. This adds complexity, but it allows other applications written in other languages, or running on other machines, to access the service.

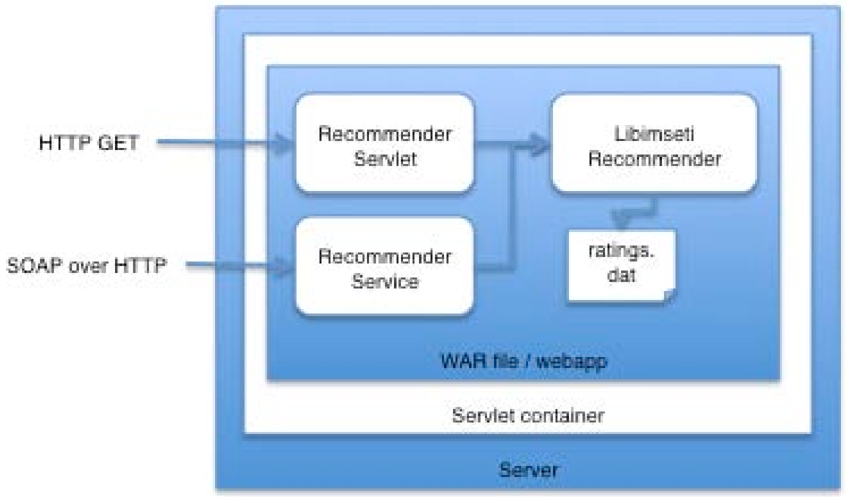


Figure 5.3 Automated WAR packaging of a recommender and deployment in a servlet container

Fortunately, Mahout makes it simple to bundle your `Recommender` implementation into a deployable WAR (web archive) file. Such a component can be readily deployed into any Java Servlet container, such as Tomcat (http://tomcat.apache.org/) or Resin (http://www.caucho.org/resin/). This WAR file, illustrated in figure 5.3, wraps up your `Recommender` implementation and exposes it via a simple servlet-based HTTP service, `RecommenderServlet`, and as an Apache Axis-powered web service using SOAP over HTTP, `RecommenderService`.

5.5.2 Packaging a WAR file

The compiled code, plus data file, will need to be packaged into a JAR file first. Chances are you have already compiled this code with your IDE, which has placed the compiled `.class` files into some output directory -- we'll call it `out`. Copy the data set's `ratings.dat` and `gender.dat` files into this same output directory, then make a JAR file with a command like "`jar cf libimseti.jar -C out/ .`".

In the `taste-web/` module directory, place the `libimseti.jar` file into the `lib/` subdirectory. Also, edit `recommender.properties` to name our recommender as the one that will be deployed. If you used the same package as the code listing above, then the right value is "`mia.recommender.libimseti.LibimsetiRecommender`".

Now execute "`mvn package`". You should find a `.war` file in the `target/` subdirectory named "`mahout-taste-webapp-0.4-SNAPSHOT.war`" (the version number may be higher if, by the time you read this, Mahout has published further releases). This is suitable for immediate deployment in a servlet container like Tomcat. In fact, this can be dropped in to Tomcat's `webapps/` directory without further modification to produce a working web-based instance of your recommender. Note that the name of the `.war` file will become part of the URL used to access the services; you may therefore wish to rename it to something shorter like "`mahout.war`".

5.5.3 Testing deployment

Alternatively, if you like, you can easily test this without bothering to set up Tomcat by using Maven's built-in Jetty plugin. Jetty (http://www.mortbay.org/jetty/) is an embeddable servlet container, which serves a function similar to that of Tomcat or Resin.

Before firing up a test deployment, you'll need to ensure that your local Mahout installation has been compiled and made available to Maven. Execute "`mvn install`" from the top-level Mahout directory and take a coffee break, since this will cause Maven to download other dependencies, compile, and run tests, all of which takes ten minutes or so. This only needs to be done once.

Having packaged the WAR file above, execute "`export MAVEN_OPTS=-Xmx2048m`" to ensure Maven and Jetty have plenty of heap space available, then from the `taste-web/` directory, "`mvn jetty:run-war`". This will start up the web-enabled recommender services on port 8080 on your local machine.

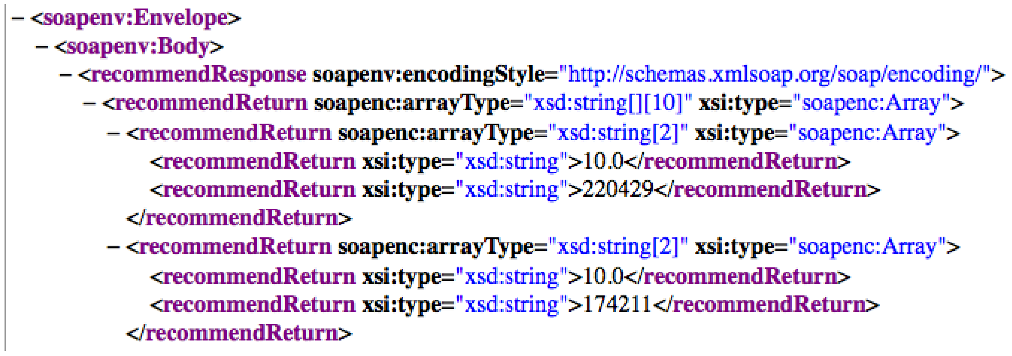
In your web browser, navigate to the URL `http://localhost:8080/mahout-taste-webapp/RecommenderServlet?userID=1` to retrieve recommendations for user ID 1. This is precisely how an external application could access recommendations from your recommender engine, by issuing an HTTP GET request for this URL and parsing the simple text result: recommendations, one estimated preference value and item ID per line, with best preference first.

Listing 5.7 Output of a GET to RecommenderServlet

```
10.0 174211
10.0 143717
10.0 220429
10.0 60679
10.0 215481
10.0 136297
9.0  192791
9.0  157343
9.0  152029
9.0  164233
9.0  207661
8.0  209192
7.0  208516
7.0  196605
7.0  2322
7.0  213682
7.0  205059
7.0  118631
7.0  208304
7.0  212452
```

To explore the more formal SOAP-based web service API that is available, access `http://localhost:8080/mahout-taste-webapp/RecommenderService.jws?wsdl` to see the WSDL (Web Services Definition Language) file that defines the input and output of this web service. It exposes a simplified version of the `Recommender` API. This web services description file can be consumed by most web service client tools, to automatically understand and provide access to the API.

If interested in trying the service directly in a browser, access `http://localhost:8080/mahout-taste-webapp/RecommenderService.jws?method=recommend&userID=1&howMany=10` to see the SOAP-based reply from the service. It is the same set of results, just presented as a SOAP response.



```
- <soapenv:Envelope>
  - <soapenv:Body>
    - <recommendResponse soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      - <recommendReturn soapenc:arrayType="xsd:string[][10]" xsi:type="soapenc:Array">
        - <recommendReturn soapenc:arrayType="xsd:string[2]" xsi:type="soapenc:Array">
            <recommendReturn xsi:type="xsd:string">10.0</recommendReturn>
            <recommendReturn xsi:type="xsd:string">220429</recommendReturn>
          </recommendReturn>
        - <recommendReturn soapenc:arrayType="xsd:string[2]" xsi:type="soapenc:Array">
            <recommendReturn xsi:type="xsd:string">10.0</recommendReturn>
            <recommendReturn xsi:type="xsd:string">174211</recommendReturn>
          </recommendReturn>
```

Figure 5.4 Browser rendering of the SOAP response from RecommenderService

Normally, at this point, you would be sanity-checking the results. Put yourself in your users' shoes -- do the recommendations make sense? Here, we don't know who the users are or what the profiles are like, so we can't do much to interpret the results intuitively. This would not be true when developing your own recommender engine, where a look at the actual recommendations would likely give insight into problems or opportunities for refinement. This would lead to more cycles of experimentation and modification to make the results match the most appropriate answers for your problem domain.

5.6 Updating and monitoring the Recommender

Now you have a live web-based recommender service running, but it's not a static, fixed system. It's natural to think about how service will be updated and monitored in production.

5.6.1 Updating recommender data directly

Of course, the data on which recommendations are based changes constantly in a real recommender engine system. Standard `DataModel` implementations will automatically use the most recent data available from your underlying data source, so, at a high level, there is nothing special that needs to be done to cause the recommender engine to incorporate new data. For example, if you had based your recommender engine on data in a database, by using a `JDBCDataModel`, then by just updating the underlying database table with new data, the recommender engine would begin using that data.

However, for performance, many components cache information and intermediate computations. These caches update eventually, but, this means that new data does not necessarily immediately affect recommendations. It is possible to force all caches to clear by calling `Recommender.refresh()`, and, this can be done by invoking the `refresh` method on the SOAP-based interface that is exposed by the web application harness. If needed, this can be invoked by other parts of your enterprise architecture.

5.6.2 Updating file-based data

File-based preference data, accessed via a `FileDataModel`, deserves some special mention. The file can be updated or overwritten in order to deploy updated information; `FileDataModel` will shortly thereafter notice the update and reload the file.

This can be slow, and memory-intensive, as both the old and new model will be in memory at the same time. Now is a good time to recall "update files," introduced in an earlier chapter. Instead of replacing or updating the main data file, it is more efficient to add update files representing recent updates. The update files are like "diffs" and when placed in the same directory as the main data file and named appropriately, will be detected and applied quickly to the in-memory representation of the preference data.

For example, an application might each hour locate all preference data created, deleted or changed in the last 60+ minutes, create an update file, and copy it alongside the main data file. Recall also that for efficiency, all of these files may be compressed.

5.6.3 Monitoring performance

Monitoring the health of this recommender service is straightforward, even if support for monitoring is outside the scope of Mahout itself. Any monitoring tool that can check the health of a web-based service, accessed via HTTP, can easily check that the recommender service is live by accessing the service URL and verifying a valid answer is returned. Such tools can and should also monitor the time it takes to answer these requests and create an alert if performance suddenly degrades. Normally, the time to compute a recommendation is quite consistent and should not vary greatly.

5.7 Summary

In this chapter, we took an in-depth look at a real, large data set made available from the Czech dating site Líbímseti. It provides 17 million ratings of over a hundred thousand profiles on the site from over a hundred thousand users. We set out to create a recommender for this site that could recommend profiles, or people, to its users.

We tried most of the recommender approaches seen so far with this data set and used evaluation techniques to choose an implementation that seemed to produce the best recommendations: a user-based recommender using a Euclidean distance-based similarity metric and nearest-2 neighborhood definition.

From there, we explored mixing in additional information from the data set: gender of the users featured in many of the profiles. We tried creating an item similarity metric based on this data. We met the `IDRescorer` interface, a practical tool that can be used to modify results in ways specific to one problem domain. We achieved a small improvement by using an `IDRescorer` to take account of gender and exclude recommendations from the gender that does not apparently interest the user.

Having tested performance and found that it performs acceptably (about 500ms per recommendation) we constructed a deployable version of our recommender engine, and automatically created a web-enabled application around it using Mahout. We briefly examined how to deploy and access this component via HTTP and SOAP.

Finally we reviewed how to update, at runtime, the recommender's underlying data.

This concludes the journey from data to production-ready recommender service. This implementation can comfortably digest this data set of 17 million ratings on one machine and produce recommendations in real time. What happens when the data outgrows one machine? In the next chapter, we'll examine how to handle a much larger data set with Hadoop.

6 Distributing Recommendation Computations

This chapter covers

- Analyzing a massive data set from Wikipedia
- Producing recommendations with Hadoop and distributed algorithms
- Pseudo-distributing existing non-distributed recommenders

We've looked at increasingly large data sets since the beginning of this book: from tens of preferences, to 100,000, to 10 million and then 17 million. This is still only medium-sized in the world of recommenders. In this chapter, we'll up the ante again by tackling a larger data set of 130 million "preferences" in the form of article-to-article links from Wikipedia's massive corpus[8]. In this data set, both users and items are articles, which also demonstrates how recommenders can be useful applied to less conventional contexts.

While 130 million preferences is still a manageable size for demonstration purposes, it is of such a scale that a single machine would have trouble processing recommendations from it in the way we've seen to date. We will need to deploy a new species of recommender algorithm, using a distributed computing approach based on the MapReduce paradigm and Hadoop.

6.1 Analyzing the massive Wikipedia data set

Wikipedia (http://wikipedia.org) is a well-known online encyclopedia whose contents may be edited and maintained by users. It reports that in May 2010 it contained over 3.2M articles written in English alone. The Freebase Wikipedia Extraction project (http://download.freebase.com/wex/) estimates the size of just the English articles to be about 42GB. Being web-based, Wikipedia articles can and do link to one

[8] Readers of earlier drafts will recall the subject of this chapter was the Netflix Prize data set. This data set is no longer officially distributed for legal reasons, and so is no longer a suitable example data set.

another. It is these links that are of interest. We will think of articles as “users”, and articles that an article points to as “items” that the source article “likes”.

Fortunately, we do not even have to download Freebase’s well-organized Wikipedia extract and parse out all these links. Researcher Henry Haselgrove has already extracted all article links and published just this information at http://users.on.net/~henry/home/wikipedia.htm. This further filters out links to ancillary resources like article discussion pages, images, and such. This data set has also represented articles by their numeric ID rather than title, which is helpful, since Mahout treats all users and items as numeric IDs.

Before continuing, download and extract `links-simple-sorted.zip`.

6.1.1 Analyzing the data set

This link data set contains 130,160,392 links from 5,706,070 articles, to 3,773,865 distinct other articles. Note that there is no explicit preference or rating; there are just associations from articles to articles. These are “boolean preferences” in the language of the framework. Associations are one-way; a link from A to B does not imply any association from B to A. There are not significantly more items than users or vice versa, so neither a user-based nor item-based algorithms suggests itself as better from an performance perspective. If using an algorithm that involves a similarity metric, one that does not depend on preference values is appropriate, like `LogLikelihoodSimilarity`.

How may we intuitively understand what the data means, and what shall we expect from the recommendations? A link from article A to B implies that B provides information related to A, typically background information on entities or ideas referenced in the article. A recommender system built on this data will recommend articles that are pointed to by other articles which also point to some of the same articles that A points to. These other articles might be interpreted as articles that A should link to, but does not. They could be articles that are simply also of interest to a reader of A. In some cases, the recommendations may reveal interesting or serendipitous associations that are not even implied by article A.

6.1.2 Struggling with scale

Deploying a non-distributed recommender engine based on this data could prove difficult. The data alone would consume about 2GB of JVM heap space with Mahout, and overall heap would likely need to be 2.5GB. On some 32-bit platforms and JVMs, this actually exceeds the maximum heap size that can be selected. This means a 64-bit machine would be required, if not immediately then soon. Depending on the algorithm, recommendation time could increase to over one second, which begins to be a long time for a “real-time” recommender engine supporting a modern web application.

With enough hardware, this could perform acceptably. But what happens when the input grows to a few billion preferences, and heap requirements top 32GB? And beyond that? For a time, one could combat scale by throwing out progressively more of the “noise” data to keep its size down. Judging what is noise begins to be a problem of accuracy and scale in its own right.

It’s unfashionable these days to be unable to cope with data beyond some scale, to have some hard limit on what your system can handle. Computing resources are readily available in large quantities; the problem here is putting enough computing resources into one box. It is disproportionately expensive to make a large machine even larger, as compared to obtaining more small machines. This massive single machine becomes a single point of failure. And, it may be hard to find any efficient way to take advantage of its expensive power when not in use by the recommender engine process.

This Wikipedia link data set size represents about the practical upper limit of how much data can be thrown at a Mahout-based real-time recommender on reasonable server hardware -- and it's not even that big by modern standards. Beyond about this scale, a new approach is needed.

6.1.2 Evaluating benefits and drawbacks of distributing computations

A solution lies in using many small machines, not one big one, for all the reasons that using one big machine is undesirable. An organization may own and operate, already, many small machines available that aren't fully utilized, and whose extra capacity could be used towards computing recommendations. Furthermore, the resources of many machines are readily available these days through cloud computing providers like Amazon's EC2 service (http://aws.amazon.com).

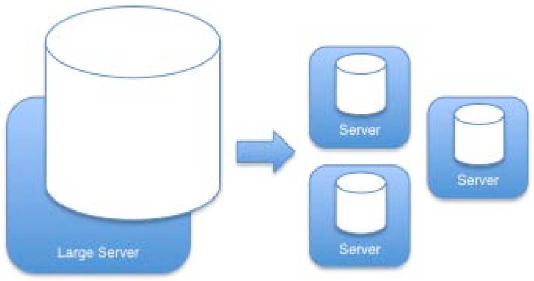


Figure 6.1 Distributed computation helps by breaking up a problem too big for one server into pieces that several smaller servers can handle.

Distributing a recommendation computation radically changes the recommender engine problem. Every algorithm we've seen so far computes recommendations as a function of, in theory, every single preference value. To recommend new links for a single article from the Wikipedia link data set, we would need access to all article-to-article links; the computation could draw on any of them. However, at large scale, access to all or even most of the data is not possible at any one time, because of its sheer size. All of the approaches we've seen so far go out the window, at least in anything like their current form. Distributed recommender engine computations are a whole new ball game.

To be clear, distributing a computation doesn't make it more efficient. On the contrary, it usually makes it require significantly more resources. For instance, moving data between many small machines consumes network resources. The computation must often be structured in a way that involves computing and storing many intermediate results, which could take significant processing time to serialize, store, and deserialize later. The software that orchestrates these operations consumes non-trivial memory and processing power.

It should be noted that such large, distributed computations are necessarily performed offline, not in real time in response to user requests. Even small computations of this form take at least minutes, not milliseconds, to complete. Commonly, recommendations would be recomputed at regular intervals, stored, and returned to the user at runtime.

However, these approaches offer a means to complete a recommender engine computation at scales where non-distributed computations cannot even start due to lack of resources on a single machine. Because distributed computations can leverage bits of resources from many machines, they offer the possibility of using spare, unused resources from existing machines rather than dedicated machines. Finally, distributed computations can allow a computation to complete earlier -- even though it might take more raw processing time. Say a distributed computation takes twice as much CPU time as its non-distributed counterpart. If 10 CPUs work on the computation, it will complete 5 times faster than a non-distributed version, which can only take advantage of one machine's resources.

6.2 Distributing an item-based algorithm

For problems of this scale, it is desirable and necessary to deploy a distributed approach to produce recommendations. First, we will sketch out a distributed variation on the item-based recommender approach we have already seen. It will be similar in some ways to the non-distributed item-based recommender algorithm we have already examined. But it will certainly look different, because the non-distributed algorithm does not fully translate to the distributed world. Then we will use Hadoop to run the algorithm.

6.2.1 Constructing a co-occurrence matrix

The algorithm we will use is best explained, and implemented, in terms of simple matrix operations. If the last time you touched matrices was in a math textbook years ago, don't worry: the trickiest operation you'll need to recall is matrix multiplication. There will be no determinants, row reduction, or eigenvalues here.

Recall that the item-based implementations we've seen so far rely on an ItemSimilarity implementation, which provides some notion of the degree of similarity between any pair of items. Imagine computing a similarity for every pair of items and putting the results into a giant matrix. It would be a square matrix, with a number of rows and columns equal to the number of items in the data model. Each row (and each column) would express similarities between one particular item and all other items. It will be useful to think of these rows and columns as vectors, in fact. It would be symmetric across the diagonal as well; because the similarity between items X and Y is the same as the similarity between items Y and X, the entry in row X and column Y would equal the entry in row Y and column X.

We need something like this for the algorithm: a "co-occurrence matrix". Instead of similarity between every pair of items, we will instead compute the number of times each pair of items occurs together in some user's list of preferences, in order to fill out the matrix. For instance, if there are 9 users who express some preference for both items X and Y, then X and Y co-occur 9 times. Two items that never appear together in any user's preferences have a co-occurrence of 0. And, conceptually, each item co-occurs with itself every time any user expresses a preference for it, though this count will not be useful.

Co-occurrence is like similarity; the more two items turn up together, the more related or similar they probably are. So, the co-occurrence matrix plays a role like that of ItemSimilarity in the item-based algorithm we saw before.

	101	102	103	104	105	106	107
101	5	3	4	4	2	2	1
102	3	3	3	2	1	1	0
103	4	3	4	3	1	2	0
104	4	2	3	4	2	2	1
105	2	1	1	2	2	1	1
106	2	1	2	2	1	2	0
107	1	0	0	1	1	0	1

Table 6.1 The co-occurrence matrix for items in simple example data set. The first row and column are labels and not part of the matrix.

Producing the matrix is a simple matter of counting. Note that the entries in the matrix are not affected by preference values. These values will enter the computation later. Table 6.1 shows the co-occurrence matrix for the small example set of preference values that we have been using throughout the book. As advertised, it is symmetric across the diagonal. There are 7 items, and the matrix is a 7x7 square matrix. The values on the diagonal, it turns out, will not be of use to the algorithm, but they are included for completeness.

6.2.2 Computing user vectors

The next step in converting our previous recommender approaches to a matrix-based distributed computation is to conceive of a user's preferences as a vector. We already did this, in a way, when discussing the Euclidean-distance-based similarity metric, where users were thought of as points in space, and similarity based on the distance between them.

Likewise, in a data model with n items, we can think of user preferences as like a vector over n dimensions, one dimension for each item. The user's preference values for items are the values in the vector. Items that the user expresses no preference for map to a 0 value in the vector. Such a vector is typically quite sparse, and mostly zeroes, because users typically express a preference for only a small subset of all items.

For example, in our small example data set, user 3's preferences correspond to the vector [2.0, 0.0, 0.0, 4.0, 4.5, 0.0, 5.0]. To produce recommendations, we will need such a vector for each user.

6.2.3 Producing the recommendations

To compute recommendations for user 3, we merely multiply this vector, as a column vector, with the co-occurrence matrix.

	101	102	103	104	105	106	107		U3		R
101	5	3	4	4	2	2	1		2.0		40.0
102	3	3	3	2	1	1	0		0.0		18.5
103	4	3	4	3	1	2	0	x	0.0	=	24.5
104	4	2	3	4	2	2	1		4.0		40.0
105	2	1	1	2	2	1	1		4.5		26.0
106	2	1	2	2	1	2	0		0.0		16.5
107	1	0	0	1	1	0	1		5.0		15.5

Table 6.2 Multiplying the co-occurrence matrix with user 3's preference vector (U3) to produce a vector that leads to recommendations, R.

Take a moment to review how matrix multiplication works, if needed (http://en.wikipedia.org/wiki/Matrix_multiplication). The product of the co-occurrence matrix and a user vector is itself a vector whose dimension is also equal to the number of items. The values in this resulting vector, R, lead us directly to recommendations: the highest values in R correspond to the best recommendations.

Table 6.2 shows this multiplication for user 3 and our small example data set, and the resulting vector R. We will ignore the values in rows of R corresponding to items 101, 104, 105 and 107 because these are not eligible for recommendation: user 3 already expresses a preference for these items. Of the remaining items, the entry for item 103 is highest, with value 24.5, and would therefore be the top recommendation, followed by 102 and 106.

6.2.4 Understanding the results

Let's pause to understand what happened above. Why do higher values in R correspond to better recommendations? Computing each entry in R is analogous to computing an estimated preference for one item, but, why is that value like an estimated preference?

Recall that computing, for example, the third entry in R entails computing the dot product between the third row vector of the matrix, and column vector U3. This is the sum of the products of each corresponding pair of entries in the vectors: 4(2.0) + 3(0.0) + 4(0.0) + 3(4.0) + 1(4.5) + 2(0.0) + 0(5.0) = 24.5

That third row contains co-occurrences between item 103 and all other items. Intuitively, if item 103 co-occurs with many items that user 3 expresses a preference for, then it is probably something that user 3 would like. The formula above sums the products of co-occurrences and preference values.

When item 103's co-occurrences overlap a lot with highly preferred items, the sum contains products of large co-occurrences and large preference values. That makes the sum larger, which is the value of the entry in R. This is why larger values in R correspond to good recommendations.

Note that the values in R do not represent an estimated preference value -- they're far too large, for one. These could be normalized into estimated preference values with some additional computation, if desired. But for our purposes, normalization doesn't matter, since we are mostly concerned with the ordering of recommendations, not the exact values on which the ordering depends.

6.2.5 Towards a distributed implementation

This is all very interesting, but what about this algorithm is more suitable for large-scale distributed implementation? The elements of this algorithm each involve only a subset of all data at any one time. For example, creating user vectors is merely a matter of collecting all preference values for one user and constructing a vector. Counting co-occurrences only requires examining one vector at a time. Computing the resulting recommendation vector only requires loading one row or column of the matrix at a time. Further, many elements of the computation just rely on collecting related data into one place efficiently -- for example, creating user vectors from all the individual preference values. The MapReduce paradigm was designed for computations with exactly these features.

6.3 Implementing a distributed algorithm with Hadoop

Now that we've sketched the algorithm, we can translate it into a form that can be implemented with MapReduce and Hadoop. Hadoop, as we've noted, is a popular distributed computing framework that includes two components of interest: a distributed file system, HDFS, and an implementation of the MapReduce paradigm.

For purposes of this chapter, we will use the Hadoop APIs found in version 0.19.x of the framework. The code presented below can be found in its complete form within Mahout, and should be runnable with Hadoop 0.19.x or 0.20.x. It may not work with later versions, as these APIs are being phased out.

6.3.1 Introducing MapReduce

MapReduce is a way of thinking about and structuring computations in a way that makes them amenable to distributing over many machines. The shape of a MapReduce computation is as follows:

1. Input is assembled in the form of many key-value (K1,V1) pairs, typically as input files on an HDFS instance
2. A "map" function is applied to each (K1,V1) pair, which results in zero or more key-value pairs of a different kind (K2,V2)
3. All V2 for each K2 are combined
4. A "reduce" function is called for each K2 and all its associated V2, which results in zero or more key-value pairs of yet a different kind (K3,V3), output back to HDFS

This may sound like an odd pattern for a computation. As it happens, many problems can be fit into this structure, or a series of them chained together. Problems framed in this way may then be efficiently distributed with Hadoop and HDFS.

6.3.2 Translating to MapReduce: Generating user vectors

In our case, the computation begins with the unarchived links data file as input. It lines are not of the form "userID,itemID,preference" that we have worked with in the past. Instead they are of the form "userID: itemID1 itemID2 itemID3 ...". This file is placed onto an HDFS instance in order to be available to Hadoop -- more on how this is done a few sections later.

The first MapReduce we use to implement this will construct user vectors:

1. Input files are treated as (`Long`,`String`) pairs by the framework, where the `Long` key is a position in the file and `String` value is the line of the text file.
2. Each line is parsed into user ID and several item IDs by a map function. The function emits new key-value pairs: user ID mapped to item ID, for each item ID
3. The framework collects all item IDs that were mapped to each user ID together.
4. A reduce function constructors a `Vector` from all item IDs for the user, and outputs the user ID mapped to the user's preference vector. All values in this vector are 0 or 1.

An implementation of this idea may be found in Listing 6.1 and Listing 6.2, below, as an implementation of both Hadoop's MapReduce `Mapper` and `Reducer` interfaces. This is typical of MapReduce computations, to have an implementation consist of a related pair of classes like this. These are all we need to implement the process above; Hadoop will take care of the rest.

Listing 6.1 Mapper which parses Wikipedia link file into ItemPrefWritables for each user

```
public class WikipediaToItemPrefsMapper extends MapReduceBase implements
    Mapper<LongWritable,Text,VLongWritable,VLongWritable> {

  public void map(LongWritable key,
                  Text value,
                  OutputCollector<VLongWritable,VLongWritable> output,
                  Reporter reporter) throws IOException {
    Matcher m = Pattern.compile("(\\d+)").matcher(value.toString());
    m.find(); A
    VLongWritable userID = new VLongWritable(Long.parseLong(m.group()));
    VLongWritable itemID = new VLongWritable();
    while (m.find()) {
      itemID.set(Long.parseLong(m.group()));
      output.collect(userID, itemID); B
    }
  }
}
```

A Locate user ID
B Emit user / item pair for each item ID

Listing 6.2 Reducer which produces Vectors from a user's item preferences

```
public class ToUserVectorReducer extends MapReduceBase implements
    Reducer<VLongWritable,VLongWritable,VLongWritable,VectorWritable> {

  public void reduce(VLongWritable userID,
                Iterator<VLongWritable> itemPrefs,
                OutputCollector<VLongWritable,VectorWritable> output,
                Reporter reporter) throws IOException {
    Vector userVector = new RandomAccessSparseVector(Integer.MAX_VALUE, 100); B
    while (itemPrefs.hasNext()) {A
      VLongWritable itemPref = itemPrefs.next();
      userVector.set(itemPref.get(), 1.0f); C
    }
    output.collect(userID, new VectorWritable(userVector));
  }
}
```

A Iterate over all item-preference pairs for a user
B Create an empty, reasonably-sized sparse vector
C Set dimension “item ID” to item’s preference value

These are simplified versions of the real implementation in Mahout, for illustration. They do not include optimizations and configuration options, but they would run and produce usable output.

6.3.3 Translating to MapReduce: Calculating co-occurrence

The next phase of the computation is another MapReduce that uses the output of the first MapReduce to compute co-occurrences.

1. Input is user IDs mapped to `Vectors` of user preferences -- the output of the last MapReduce.
2. The map function determines all co-occurrences from one user’s preferences, and emits one pair of item IDs for each co-occurrence -- item ID mapped to item ID and a count of 1. Both mappings, from one item ID to the other and vice versa, are recorded.
3. The framework collects, for each item, all co-occurrences mapped from that item.
4. The reducer tallies up all counts, for each item ID, all co-occurrences that it receives and constructs a new `Vector`, which represents all co-occurrences for one item with count of number of times they have co-occurred. These can be used as the rows -- or columns -- of the co-occurrence matrix.

The output of this phase is in fact the co-occurrence matrix. Again, Listing 6.3 and Listing 6.4 provide a simplified look at how this is implemented in Mahout on top of Hadoop. Again we have a pair of related implementations, of `Mapper` and `Reducer`.

Listing 6.3 Mapper component of co-occurrence computation

```
public class UserVectorToCooccurrenceMapper extends MapReduceBase
    implements Mapper<VLongWritable,RandomAccessSparseVectorWritable,
                      IntWritable,EntityCountWritable> {

  public void map(VLongWritable userID,
                  RandomAccessSparseVectorWritable userVector,
                  OutputCollector<IntWritable,EntityCountWritable> output,
                  Reporter reporter) throws IOException {
    Iterator<Vector.Element> it = userVector.get().iterateNonZero(); A
    IndexIndexWritable entityEntity = new IndexIndexWritable();
    IntWritable one = new IntWritable(1);
    while (it.hasNext()) {
      int index1 = it.next().index();
      Iterator<Vector.Element> it2 = userVector.iterateNonZero();
      while (it2.hasNext()) {
        int index2 = it2.next().index();
        if (index1 != index2) {
          entityEntity.set(index1, index2);
          output.collect(entityEntity, one); B
        }
      }
    }
  }
```

A Only necessary to iterate over the non-zero elements
B Record count of 1

Listing 6.4 Reducer component of co-occurrence computation

```
public class UserVectorToCooccurrenceReducer extends MapReduceBase implements
    Reducer<IndexIndexWritable,IntWritable,IntWritable,VectorWritable> {

  private int lastItem1ID = Integer.MIN_VALUE;
  private int lastItem2ID = Integer.MIN_VALUE;
  private Vector cooccurrenceRow = null;
  private int count = 0;

  public void reduce(IndexIndexWritable entityEntity,
                     Iterator<IntWritable> counts,
                     OutputCollector<IntWritable,VectorWritable> output,
                     Reporter reporter) throws IOException {

    int item1ID = entityEntity.getAID();
    int item2ID = entityEntity.getBID();
    if (item1ID == lastItem1ID) {
      if (item2ID == lastItem2ID) {
        count += CooccurrenceCombiner.sum(counts); A
      } else {
        if (cooccurrenceRow == null) {
          cooccurrenceRow = new RandomAccessSparseVector(Integer.MAX_VALUE);
        }
        cooccurrenceRow.set(item2ID, count); B
        lastItem2ID = item2ID;
        count = CooccurrenceCombiner.sum(counts);
      }
    } else {
      if (cooccurrenceRow != null) {
        output.collect(new IntWritable(lastItem1ID),
                       new VectorWritable(cooccurrenceRow)); C
      }
      lastItem1ID = item1ID;
      lastItem2ID = item2ID;
      cooccurrenceRow = null;
      count = CooccurrenceCombiner.sum(counts);
    }
  }
}
```

A Accumulate counts for item 1 / 2
B Record counts for item 1 / 2
C Done, record entire item 1 vector

Above in the mapper, note how we output two item IDs and a count of “1” every time two items co-occur. That may seem redundant, and it is. However, it’s done to enable an optimization in Hadoop known as a combiner. A combiner is like a mini-reduce phase that runs after a mapper. It can combine multiple map outputs into one that represents the same information, before storing and sending to reducers. For example, imagine items 123 and 456 co-occur ten times. Normally, the map phase would output ten `EntityCountWritables`, each recording that 123 and 456 co-occurred once. The combiner can combine these ten records into one, with count 10.

This useful detail is not applicable in all situations, but is perfect for this situation. It is implemented in Mahout, and can be seen in the source code, for the interested. With the co-occurrence matrix in hand, we can proceed to the final computation of recommendations.

6.3.4 Translating to MapReduce: Rethinking matrix multiplication

We are ready to use MapReduce to multiply the user vectors computed in step 1, and the co-occurrence matrix from step 2, to produce a recommendation vector from which we may derive recommendations.

However we will perform the multiplication in a different way that is more efficient here, and more naturally fits the shape of a MapReduce computation. We will not perform conventional matrix multiplication, wherein each row is multiplied against the user vector (as a column vector), to produce one element in the result R:

```
for each row i in the co-occurrence matrix
     compute dot product of row vector i with the user vector
     assign dot product to ith element of R
```

Why depart from the algorithm we all learned in school? The reason is purely performance, and this is a good opportunity to examine the kind of thinking necessary to achieve performance at scale when designing large matrix and vector operations. The conventional algorithm necessarily touches the entire co-occurrence matrix, since it needs to perform a vector dot product with each row. Anything that touches the entire input is “bad” here since the input may be staggeringly large and not even available locally. Instead, we note that matrix multiplication can be accomplished as a function of the co-occurrence matrix columns:

```
assign R to be the zero vector
for each column i in the co-occurrence matrix
     multiply column vector i by the ith element of the user vector
     add this vector to R
```

Take a moment to convince yourself that this is also a correct way to define this matrix multiplication, with a small example perhaps. So far, this isn’t an improvement: it also touches the entire co-occurrence matrix, by column.

However, note that wherever element i of the user vector is 0, we can skip the loop iteration entirely, because the product will just be the zero vector and does not affect the result. So, this loop need only execute for each non-zero element of the user vector. The number of columns loaded will be equal to the number of preferences that the user expresses, which is far smaller than the total number of columns.

And, expressed this way, we can distribute the computation efficiently. Column vector i can be output along with all elements it needs to be multiplied against. The products can be computed and saved independently of handling of all other column vectors.

6.3.5 Translating to MapReduce: Matrix multiplication by partial products

We already have the columns of the co-occurrence matrix from an earlier step. Because the matrix is symmetric, the rows and columns are identical, so we can use this output as either rows or columns, conceptually. These columns are keyed by item ID. We must multiply each by every non-zero preference value for that item, across all user vectors. That is, we need to map item IDs to a user ID and preference value, and then collect them together in a reducer. After multiplying the co-occurrence column by each value, we have a vector that forms part of the final recommender vector R for one user.

The difficult part here is that we want to combine two different kinds of data in one computation: co-occurrence column vectors, and user preference values. This isn't by nature possible in Hadoop, since values in a reducer can be of one `Writable` type only. We can get around this by crafting `Writable` that contains either one or the other type of data: a `VectorOrPrefWritable`. While it may be viewed as a hack, it may be valuable or necessary in designing a distributed computation to bend some rules to achieve an elegant, efficient computation.

So, the mapper phase here will actually contain two mappers, each producing different types of reducer input:

5. Input for mapper 1 is the co-occurrence matrix: item IDs as keys, mapped to columns as `Vector`s.
6. The map function simply echoes its input, but with the `Vector` wrapped in a `VectorOrPrefWritable`.
7. Input for mapper 2 is again the user vectors: user IDs as keys, mapped to preference Vectors
8. For each non-zero value in the user vector, the map function outputs item ID mapped to the user ID and preference value (here, all non-zero values are 1), wrapped in a `VectorOrPrefWritable`
9. The framework collects together, by item ID, the co-occurrence column and all user ID / preference value pairs.
10. The reducer unpacks this input and performs all multiplications with the co-occurrence column vector. (Here, since values are 1, we can skip the multiplication.) For each user ID pair, it outputs as a `Vector` the product, which is part of the user's recommendation vector R.

Listing 6.5 Wrapping co-occurrence columns

```
public class CooccurrenceColumnWrapperMapper extends MapReduceBase
    implements Mapper<IntWritable,VectorWritable,
                      IntWritable,VectorOrPrefWritable> {

  public void map(IntWritable key,
                  VectorWritable value,
                  OutputCollector<IntWritable,VectorOrPrefWritable> output,
                  Reporter reporter) throws IOException {
    output.collect(key, new VectorOrPrefWritable(value.get()));
  }
}
```

Listing 6.5 shows the co-occurrence columns being simply wrapped in `VectorOrPrefWritable`.

Listing 6.6 Splitting user vectors

```
public class UserVectorSplitterMapper extends MapReduceBase
    implements Mapper<VLongWritable,VectorWritable,
                      IntWritable,VectorOrPrefWritable> {

  public void map(VLongWritable key,
                  VectorWritable value,
                  OutputCollector<IntWritable,VectorOrPrefWritable> output,
                  Reporter reporter) throws IOException {
    long userID = key.get();
    Vector userVector = value.get();
    Iterator<Vector.Element> it = userVector.iterateNonZero();
    while (it.hasNext()) {
      Vector.Element e = it.next();
      int itemIndex = e.index();
      float preferenceValue = (float) e.get();
      itemIndexWritable.set(itemIndex);
      output.collect(new IntWritable(itemIndex),
                     new VectorOrPrefWritable(userID, preferenceValue));
    }
  }
}
```

In Listing 6.6, user vectors are “split” into their individual preference values, and output, mapped by item ID rather than user ID.

Listing 6.7 Computing partial recommendation vectors

```
public class PartialMultiplyReducer extends MapReduceBase implements
  Reducer<IntWritable,VectorOrPrefWritable,VLongWritable,VectorWritable> {

 public void reduce(IntWritable key,
           Iterator<VectorOrPrefWritable> values,
           final OutputCollector<VLongWritable,VectorWritable> output,
           Reporter reporter) throws IOException {

  OpenLongFloatHashMap savedValues = new OpenLongFloatHashMap();
  Vector cooccurrenceColumn = null;
  final int itemIndex = key.get();
```

```
      final VLongWritable userIDWritable = new VLongWritable();
      final VectorWritable vectorWritable = new VectorWritable();
      vectorWritable.setWritesLaxPrecision(true);

      while (values.hasNext()) {

        VectorOrPrefWritable value = values.next();
        if (value.getVector() == null) {

          long userID = value.getUserID(); A
          float preferenceValue = value.getValue();

          if (cooccurrenceColumn == null) { B
            savedValues.put(userID, preferenceValue);
          } else { C
            Vector partialProduct = cooccurrenceColumn; D
            partialProduct.set(itemIndex, Double.NEGATIVE_INFINITY); E
            userIDWritable.set(userID);
            vectorWritable.set(partialProduct);
            output.collect(userIDWritable, vectorWritable);
          }

        } else {

          cooccurrenceColumn = value.getVector(); F

          final Vector theColumn = cooccurrenceColumn;
          savedValues.forEachPair(new LongFloatProcedure() {
            public boolean apply(long userID, float value) {
              Vector partialProduct = theColumn.times(value);
              partialProduct.set(itemIndex, Double.NEGATIVE_INFINITY);
              userIDWritable.set(userID);
              vectorWritable.set(partialProduct);
              try {
                output.collect(userIDWritable, vectorWritable); G
              } catch (IOException ioe) {
                throw new IllegalStateException(ioe);
              }
              return true;
            }
          });
          savedValues.clear();
        }
      }

    }
  }
```

A Then it's a user ID / preference
B Co-occurrence column vector not yet seen
C Have column vector so multiply
D Normally, multiply by preference value
E Makes sure the item isn't recommended
F Found the column vector
G Output product for all saved values

Most of the complexity of listing 6.7, which shows the output of the two previous mappers being multiplied, comes from the fact that it's not known which value will contain the co-occurrence vector. Until it's seen, the user IDs and preference values must be stored temporarily for later multiplication with the vector, once it appears.

6.3.6 Translating to MapReduce: Making recommendations

At last, we just need to assemble the pieces of the recommendation vector for each user and make recommendations. Listing 6.8 shows this in action.

11. The input for the mapper is the output of the previous step: user IDs as keys mapped to a part of that user's recommendation vector, as a `Vector`.
12. The mapper merely passes through these keys and values.
13. The framework collects all of the partial vectors for each user ID.
14. The reducer sums all partial vectors for a user ID to produce that user's recommendation vector. The highest values in the vector are the best recommendations and are output.

Listing 6.8 Producing recommendations from vector

```
public class AggregateAndRecommendReducer extends MapReduceBase
    implements Reducer<VLongWritable,VectorWritable,
                       VLongWritable,RecommendedItemsWritable> {

  public void reduce(VLongWritable key,
                     Iterator<VectorWritable> values,
                     OutputCollector<VLongWritable,
                                     RecommendedItemsWritable> output,
                     Reporter reporter) throws IOException {
    Vector recommendationVector = values.next().get();
    while (values.hasNext()) { A
      recommendationVector = recommendationVector.plus(values.next().get());
    }

    Queue<RecommendedItem> topItems =
      new PriorityQueue<RecommendedItem>(10,
        Collections.reverseOrder(
          ByValueRecommendedItemComparator.getInstance())); B

    Iterator<Vector.Element> recommendationVectorIterator =
        recommendationVector.iterateNonZero();
    while (recommendationVectorIterator.hasNext()) {
      Vector.Element element = recommendationVectorIterator.next();
      int index = element.index();
      if (topItems.size() < 10) {
        long theItemID = indexItemIDMap.get(index);
        topItems.add(new GenericRecommendedItem(
          theItemID, (float) element.get()));
      } else if (element.get() > topItems.peek().getValue()) {
        long theItemID = indexItemIDMap.get(index);
        topItems.add(new GenericRecommendedItem(
          theItemID, (float) element.get()));
        topItems.poll();
      }
    }

    List<RecommendedItem> recommendations =
      new ArrayList<RecommendedItem>(topItems.size());
    recommendations.addAll(topItems);
    Collections.sort(recommendations,
                     ByValueRecommendedItemComparator.getInstance()); C
    output.collect(key, new RecommendedItemsWritable(recommendations));
  }
}
```

A Build the recommendation vector by summing
B Find the top 10 highest values
C Output recommendations in order

The output ultimately exists as one or more files stored on an HDFS instance, as a compressed text file; the lines in the text file are of the form:

```
3    [103:24.5,102:16.5,106:16.5]
```

Each user ID is followed by a comma-delimited list of item IDs that have been recommended (followed by a colon and the corresponding entry in the recommendation vector, for what it is worth). This output can be retrieved from HDFS, parsed, and used in an application. Note that the output from Mahout will be compressed, to save space, using gzip.

6.4 Running MapReduces with Hadoop

Now we're ready to try out this implementation on the Wikipedia links data set. Although Hadoop is a framework for running a computation across clusters of potentially thousands of machines, we will start by showing how to run a Hadoop computation on a cluster of one machine: yours.

6.4.1 Setting up Hadoop

As mentioned in the opening chapter, you will need to download a recent of copy of Hadoop from http://hadoop.apache.org/common/releases.html. Version 0.20.x is most recent and recommended at the time of this writing. Follow the setup directions at http://hadoop.apache.org/common/docs/current/quickstart.html and configure for what it calls "pseudo-distributed" operation. Before running the Hadoop daemons with `bin/start-all.sh`, make one additional change: in `conf/mapred-site.xml`, add a new property named "mapred.child.java.opts" with value "-Xmx1024m". This will enable Hadoop workers to use up to 1GB of heap memory. You can stop following the setup instructions after running all the Hadoop daemons.

You are now running a complete Hadoop cluster on your local machine, including an instance of the HDFS distributed file system. We need to put the input onto HDFS to make it available to Hadoop. You may wonder why, if the data is readily available on the local file system, it needs to be copied again into HDFS. Recall that in general, Hadoop is a framework run across many machines, so, any data it uses needs to be available not to one machine but many. HDFS is an entity that can make data available to these many machines. Copy the input to HDFS with "`bin/hadoop fs -put links-simple-sorted.txt input/input.txt`".

Computing recommendations for every article in the data set would take a long time, because we are running on only one machine and incurring all the overhead of the distributed computing framework. We can ask the implementation in Mahout to compute recommendation for, say, just one user. Create a file containing only the number "3" on a single line. Save it as `users.txt`. This is a list of the articles for which we will generate recommendations -- here, one, for testing purposes. Place in into HDFS as well with "`bin/hadoop fs -put users.txt input/users.txt`".

6.4.2 Running recommendations with Hadoop

The glue that binds together the various `Mapper` and `Reducer` components we've seen so far is `org.apache.mahout.cf.taste.hadoop.item.RecommenderJob`. It can be found within the Mahout source distribution. It configures and invokes the series of MapReduce jobs we have discussed.

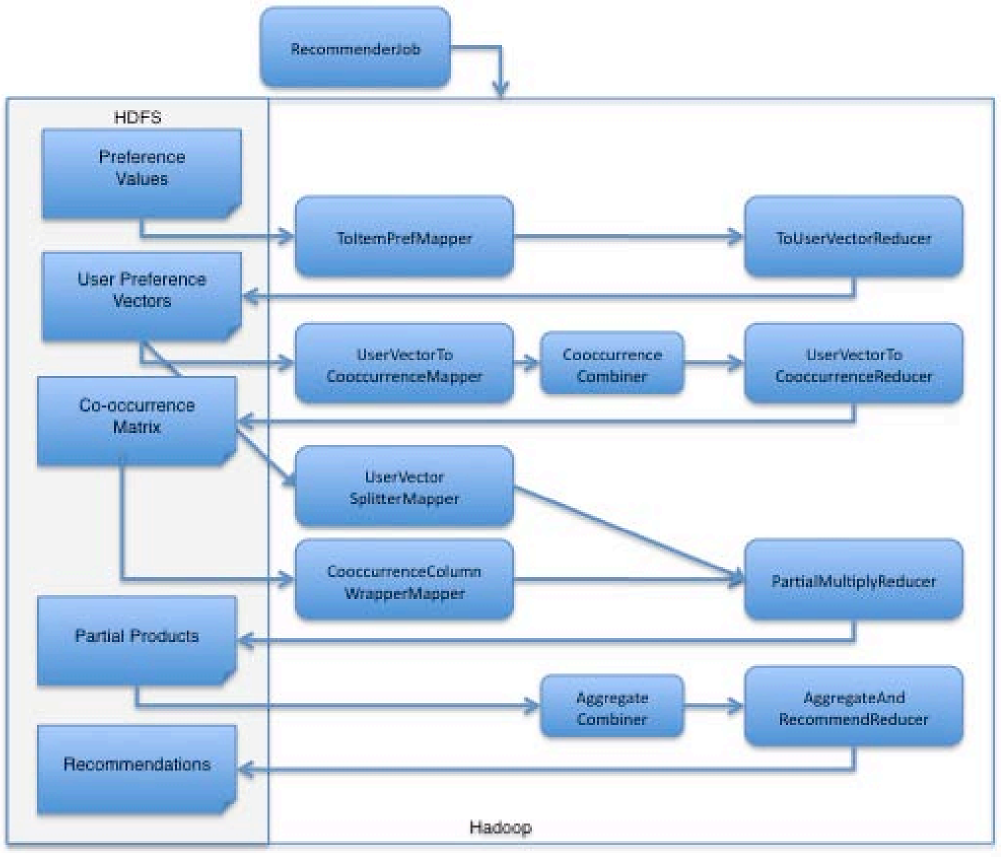


Figure 6.2 The relation between RecommenderJob, the three MapReduces it invokes, and the data that they read to and write from HDFS

In order to run it, and allow Hadoop to run these jobs, we need to compile all of this code into one `.jar` file along with all of the code it depends upon. This can be accomplished easily by running "`mvn` clean package" from the core/ directory in the Mahout distribution. This will produce a file like target/mahout-core-0.x-SNAPSHOT.job, which is in reality a .jar file.

Now, kick it all off with:

```
bin/hadoop jar target/mahout-core-0.x-SNAPSHOT.job
  org.apache.mahout.cf.taste.hadoop.item.RecommenderJob
  -Dmapred.input.dir=input/input.txt
  -Dmapred.output.dir=output --usersFile input/users.txt --booleanData
```

Hadoop will take over and begin running the series of jobs. It will take many hours, because only one machine (yours) is being deployed to complete the computations. Even with a small army of machines, don't expect results in minutes; the overhead of initializing the cluster, distributing the data and executable code, and marshalling the results, is non-trivial. If you are patient enough to let it complete, you should find the results on HDFS under the output/ directory. It will be contained in a single file called part-00000.

Copy the result back to your local file system with "bin/hadoop fs -get output/part-00000". This can be examined and used as desired. Congratulations, that's it, you've produced recommendations with a fully distributed framework (on a cluster of one machine). Don't forget to shut down Hadoop with "bin/stop-all.sh" when done.

6.4.3 Configuring mappers and reducers

One important point deserves mention here. Above, we let Hadoop default to running just one map and one reduce worker at once. This is appropriate since we're running on just one machine. In general, when launching this job on a cluster of many machines, one worker is of course too little. On a real cluster, this can be controlled with command-line arguments like "-Dmapred.map.tasks=X -Dmapred.reduce.tasks=Y". Setting both equal to the total number of cores available in the cluster is a good place to start. For example, if your cluster has five quad-core machines, set both to 20.

6.5 Pseudo-distributing a Recommender

Earlier, we saw how to create, test and operate a variety of non-distributed recommender engines with Mahout, on one machine. In this chapter, we saw how to run one fully distributed recommender computation using a quite different approach. There is a middle ground, however, for applications that want to use multiple machines, but want to use an existing non-distributed implementation.

This might be the case for applications that have already developed a customized, effective implementation using the non-distributed framework. Such a Recommender implementation is likely, as with all the non-distributed implementations we've seen, intimately bound to a DataModel to do its work, and assumes efficient, random access to all available data. It might be hard or impossible to reinvent it in a fully distributed form.

For these situations, Mahout offers a "pseudo-distributed" recommender engine framework. It is merely a Hadoop-based harness that can run several independent, non-distributed instances of a given recommender engine in parallel. As such, it is an easy way to "port" a stock, non-distributed algorithm to use many machines. This facility does not actually parallelize the computation in any sense; it only manages operation of multiple non-distributed instances. Performance is the same as when running a non-distributed instance directly. However this allows you to run n instances of the recommender, on n machines, each producing 1/n of all the recommendations required, in a total of 1/n the time it would take one machine to finish.

The disadvantage to this approach is a scalability limitation: a non-distributed computation remains limited in the amount of data it can handle by the resources of the machine(s) that it runs on. That is, a computation that can't fit on one large machine still won't fit when sent to n independent large machines. Pseudo-distributing the computation does not change this.

6.5.1 Running a pseudo-distributed Recommender on Hadoop

There are no new algorithms or code to introduce here; the pseudo-distributed recommender engine framework in Mahout runs the `Recommenders` we've already seen, but on Hadoop. Conceptually, it uses Hadoop to split the set of users across n machines, copy the input data to each, and then run one `Recommender` on each machine to process recommendations for a subset of users.

The process is the same as before. With Hadoop set up and running, copy the preferences input file into HDFS. If you wish to try out this framework, choose the input from a data set we have studied already, such as `ua.base` from the GroupLens 100K data set. (The Wikipedia links data set will be too large to use with a non-distributed implementation.) Place `ua.base` into HDFS under, for example, `input/ua.base`.

We will need to give the framework the name of a `Recommender` implementation that it can instantiate and use. The only requirement is that the implementation provides a constructor that takes a single argument, a `DataModel`. With this, the framework can do the rest. Typically, you would supply a customized `Recommender` that you had created for your application here; for testing purposes, `SlopeOneRecommender` will do because it can be instantiated with only a `DataModel` as configuration.

Create `mahout.jar` as above. As it happens, this .jar file already contains `SlopeOneRecommender`, because it is a standard Mahout implementation. However, were you to use your own implementation, you would need to add it and any of its dependent classes into the .jar file as well. This can be accomplished with "`jar uf mahout.jar -C [classes directory]`", where the classes directory is the location where your IDE or build tool output the compiled version of your code.

Finally, run the job:

```
bin/hadoop jar target/mahout-core-0.x-SNAPSHOT.job
  org.apache.mahout.cf.taste.hadoop.pseudo.RecommenderJob
  -Dmapred.input.dir=input/ua.base
  -Dmapred.output.dir=output
  --recommenderClassName
  org.apache.mahout.cf.taste.impl.recommender.slopeone.SlopeOneRecommender
```

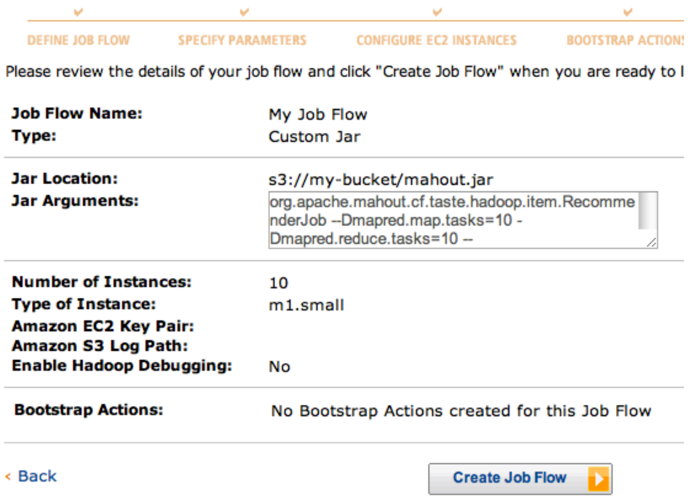
As before, you will find the output in HDFS in the `output/` directory. That's all there is to it; if your input is of such a scale that truly distributed algorithms are not required, then the pseudo-distributed recommender framework is a quick and easy way to utilize more computing power to produce recommendations faster.

6.6 Looking beyond first steps with recommendations

The portion of Mahout's recommendation engine introduced in this chapter is, as we go to press, still quite under construction, so refer to the latest documentation and code in conjunction with this reference book. The techniques described above are also by no means the best or only way to distribute a recommender computation; they are merely the first that the framework provides. Look to Mahout to provide more options as it evolves. On that note, we conclude with some thoughts about where to go next from here in your thinking and investigation of recommender engines.

6.6.1 Running in the cloud

Don't have a hundred machines lying around on which to run these big distributed computations? Fortunately, today, service providers allow you to rent storage and computing time from a computing cloud.



Please review the details of your job flow and click "Create Job Flow" when you are ready to l

Job Flow Name:	My Job Flow
Type:	Custom Jar
Jar Location:	s3://my-bucket/mahout.jar
Jar Arguments:	org.apache.mahout.cf.taste.hadoop.item.RecommenderJob --Dmapred.map.tasks=10 -Dmapred.reduce.tasks=10 --
Number of Instances:	10
Type of Instance:	m1.small
Amazon EC2 Key Pair:	
Amazon S3 Log Path:	
Enable Hadoop Debugging:	No
Bootstrap Actions:	No Bootstrap Actions created for this Job Flow

Figure 6.3 Amazon's AWS Elastic MapReduce console

Amazon's Elastic MapReduce service (http://aws.amazon.com/elasticmapreduce/) is one such service. It uses Amazon's S3 storage service instead of a pure HDFS instance for storing data in the cloud. After uploading your `.jar` file and data to S3, you can invoke a distributed computation using their AWS Console by supplying the same arguments we used to invoke the computation on the command line earlier.

After logging in to the main AWS Console, select the Amazon Elastic MapReduce tab. Choose to "Create New Job Flow". Give the new flow whatever name you like and specify "Run your own application". Choose the "Custom jar" type and continue. Specify the location on S3 where the `.jar` file resides; this will be an `s3:` URI, not unlike "`s3://my-bucket/target/mahout-core-0.x-SNAPSHOT.job`".

The job arguments will be the same as when running on the command line; here it will certainly be necessary to configure the number of mappers and reducers. The number of mappers and reducers can be tuned to your liking; as above, we recommend starting with a number equal to the number of virtual cores you reserve for the computation. While any instance type can be used, we recommend starting with the "regular" types unless there is reason to choose something else: small, large or extra-large. The number and type of instances is selected on the next AWS Console screen.

If your input data is extremely large, some recommender jobs such as that in `org.apache.mahout.cf.taste.hadoop.item` may need a more RAM per mapper or reducer. In this case you may have to choose a high-memory instance type. You may also opt for a high-CPU instance type; the risk is that the jobs will spend enough time reading and writing data to S3 that these instances' speedy CPUs will go mostly unused. Therefore the conventional instance types are a good place to start. If using the "small" instance type, which has 1 virtual core per instance, then simply set the number of mappers and reducers equal to the number of instances you will select.

You may leave other options untouched unless you have reason to set them. This is the essence of running a recommender job on Elastic MapReduce; refer to Amazon's documentation for more information about how to monitor, stop, and debug such jobs. While Amazon AWS uses Hadoop version 0.18.3 at the time of this writing, it should still be compatible with Mahout-related Hadoop code, even though Mahout is developed against version 0.20.x.

6.6.2 Imagining unconventional uses of recommendations

Although the Mahout recommender engine APIs are phrased in terms of "users" and "items", the framework does not actually assume that users are people and items are objects like books and DVDs. We already applied recommender engines to a dating site's data to recommend people to people for example. What other ways can recommender engines be applied? We provide some ideas to fire your imagination:

- Recommend users to items: By simply swapping item IDs for user IDs, a recommender engine's output instead suggests which users might be most interested in a given item.
- Think broadly about "items": given associations from users to places, times, usage patterns, or other people, you can recommend the same back to them.
- Find most similar items. Item-based recommender implementations in Mahout make it easy to find a set of most similar items, which could be useful to present to users as well.
- Think broadly about preference values. It's unusual to be able to collect explicit preference values from users. Think about what you can infer from the data you do have about users' relations to things.
- Think about more than just one "user" and "item": you can recommend to pairs of users by thinking of a pair of users as a "user". You can recommend, say, an item and place by taking both of them together as an "item".

Mahout does not offer particular, special support for these use cases, though all can be implemented on top of Mahout. This is a possible direction in which Mahout could grow, or in which specialized third-party projects might appear. In particular, the problem of inferring implicit ratings based on user behavior and other data is a fascinating and important problem in its own right, but not one that Mahout addresses.

6.7 Summary

In this chapter, we took a brief look at the large data set based on Wikipedia article links. With 130M "preferences", it is large enough to require a different, distributed approach to produce recommendations.

We discussed the tradeoffs inherent in moving from one machine and a non-distributed algorithm to a large distributed computation on a cluster of machines. Then we briefly introduced the MapReduce paradigm and its implementation in Hadoop as a way to manage such distributed computations.

We translated the item-based recommender algorithm we saw before into a different distributed implementation, which relies on matrix and vector operations to discover the best recommendations. We returned to the Wikipedia data set, prepared it for use with Hadoop, and walked through creating recommendations for this data set on a local Hadoop and HDFS instance.

Finally, we examined pseudo-distributed recommender computations with Mahout: running several independent instances of non-distributed `Recommender` implementations on Hadoop.

This concludes the coverage of recommender engines in Mahout. It has been intended as a gentle introduction to one aspect of machine learning, which gradually evolved from small input and non-distributed computation to large-scale distributed computation. Now, we move to discuss clustering and classification with Mahout, which entails more complex machine learning theory and more intense use of distributed computing. With recommender engines under your belt, you're ready to engage these topics. Read on.

7
Introduction to Clustering

This chapter covers:

- A hands-on look at `Clustering` in action
- Understanding the notion of similarity
- Running a simple clustering example in Mahout
- The various distance measures used for clustering

Birds of a feather flock together. As human beings, we tend to associate with like-minded people. We have a great mental ability for finding repeating patterns, and we continually associate what we see, hear, smell or taste to things that are already in our memory. For example, the taste of honey reminds us more of the taste of sugar than salt. So we group together the things that taste like sugar and honey and call them "sweet". Without even knowing what "sweet" tastes like, we know that all the sugary things in the world are similar and of the same group. However, we know how different they are from all the things belonging to the salty group. Unconsciously, we group together tastes into such "clusters". So, in nature we have clusters of sugary things and salty things, with each group having hundreds of items in it.

In nature, we observe many other types of groups. Consider apes versus monkeys, which are both kinds of primates. All monkeys share some traits like short height, long tail, and flat nose. On the other hand, apes are characterized by their large size, long arms, and bigger head. Apes look different from monkeys, but both are fond of bananas. So it is entirely up to us to think of apes and monkeys as two different groups, or as a single group of banana-loving primates. Therefore, what we consider as a cluster entirely depends on the traits we choose for measuring the similarity between items (in this case, primates).

So what is the process of clustering all about? Suppose you were given the keys to a library containing thousands of books. However, in this library the books are arranged in no particular order. Readers entering your library would have to sweep through all the books one by one to find a particular one. Not only is this cumbersome, and slow, but tedious as well.

Sorting the books alphabetically by title would be a vast improvement -- for readers searching for a book by title, that is. What if most people were simply browsing, or researching a general subject? A grouping of the books by topics would more useful than an alphabetical ordering.

How would you even begin this grouping? Having just taken over this job, you aren't even sure what all the books are about -- surfing, romance, or even topics you haven't encountered before? To group the books by topic, you could lay down all the books in a line and start reading them one by one. When you encounter a book whose content is similar to a previous book, you could go back and stack them together. At the end, you would have some hundreds of stacks of books instead of thousands.

Good work -- this was your first clustering experiene. If a hundred topic groups were too large, you could go back to the beginning of the line and repeat the process with stacks until you got stacks that start looking quite different from one another.

7.1 What is clustering?

Clustering is all about organizing similar items into groups from a given collection of items. These clusters could be thought of as a set of items similar to each other in some ways but dissimilar from the items belonging to other clusters. Clustering a collection involves:

- an algorithm, the method used to group the books together
- a notion of both similarity and dissimilarity -- above we relied on your assessment of which books belonged in an existing stack and which should start a new one
- a stopping condition. In the librarian example, this might have been the point beyond books can't be stacked anymore, or when the stacks are already quite dissimilar.

Until now we have thought of clustering items as stacking them. Really, we were just grouping them. Conceptually, clustering is more like looking at which items form "near" groups and just circling them. Take look at Figure 7.1. The figure shows clustering of points in a standard X-Y plane. Each circle represents one cluster, containing several points. In this simple example, this is obviously the best clustering of points into 3 clusters based on distance. Circles are good way to think of clusters, since clusters are also defined by a center point and radius. The center of the circle is called the centroid, or mean (average), of that cluster. It is the point whose coordinates are the average of the x and y coordinates of all points in the cluster.

In later chapters, we will explore some of the methods that are popularly used for clustering data – and the way they are implemented in software in Mahout. The strategy in the librarian examples was to merge stacks of books until some threshold was reached. The number of clusters formed in this case depended on the data -- based on the number of books and threshold, we might have ended up with 100, 20, or even just 1 cluster. A more popular strategy is to set a target number of clusters, instead of a threshold, and then find the best grouping with that constraint. Later we wille explain this and other variations in detail.

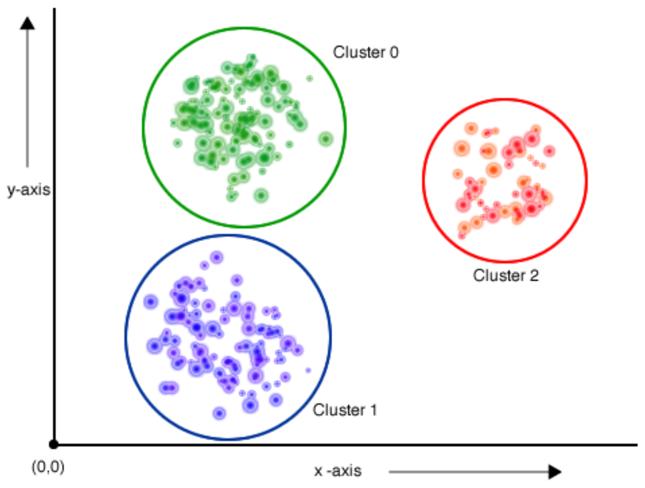


Figure 7.1 Points in an x-y plane. Circles represent the clusters. The points on the plane could be viewed as 3 logical groups. Clustering algorithms helps surface those groups to you.

7.2 Measuring the similarity of Items

The most important issue in clustering is finding a function that quantifies the similarity between any two data points as a number. Note that we are using the terms "item" and "point" interchangeably in this whole book. Both refer to a unit of data we wish to cluster.

In the X-Y plane example, the measure of similarity (or "similarity metric") for the points was the Euclidean distance between two points. The librarian example had no such clear, mathematical measure and instead relied entirely on the wisdom of the librarian to judge book similarity. That surely doesn't work for us, since we need a metric that can be implemented on a computer.

One possible metric could be based on the number of words common to two books' titles. So "Harry Potter: The Philosopher's Stone" and "Harry Potter: The Prisoner of Azkaban" have three words in common: "Harry", "Potter" and "The". But, even though the book "The Lord of the Rings: The Two Towers" is similar to the Harry Potter series, this measure of similarity doesn't capture that. We should alter the similarity measure to take account of the contents of the book itself. We could assemble word counts for each book, and when the counts are close for many words, judge the books similar.

Unfortunately, that is easier said than done. Not only do these books have hundreds of pages of text, but this sort of measure is confounded by features of English. The most frequent words in these English-language texts are words like "a", "an", and "the" which invariably occur frequently in both books, but say little about the book similarity.

To combat this effect, we could use numeric weights in the computation, and apply low weights to these words to reduce their effect on the similarity value. We should give less weight to words that occur across many books, and more weight to words that are found in few books. We should also weight words occurring more often in a particular book because those words strongly suggest the content of the books – like “magic” in the case of Harry Potter

Once we give a weight value to each word in a book, we can say the similarity between two books is the sum over all words of the similarity of those two words’ counts in the two books times their weighting. This is a decent measure, if the books are of equal length. What if one book is 300 pages long and the other 1000 pages long? Surely, the larger book will have a larger count of words, in general. We have to ensure that the weight of words should be relative to the length of the text. A popular method called Tf-Idf (term frequency - inverse document frequency) weighting does this quickly and effectively. We will cover Tf-Idf and others variations of it in detail in a later chapter.

7.3 Hello World: Running a simple clustering example

Mahout contains various implementations of clustering, like K-means, fuzzy K-means, and meanshift to name a few. In upcoming chapters we will review each of the clustering algorithms in Mahout and their real world applications. We will look at how to represent the data, run various algorithms, tune their parameters, and how to customize clustering to fit real world problems.

7.3.1 Creating the input

First, let us try a simple example, which clusters points in two dimensions like the one we saw in figure 7.1. First, we need to input the points in a plane.

We start by creating a list of points to cluster. Mahout clustering algorithms takes input in a particular binary format called SequenceFile, from Hadoop. The input encodes `Vectors`, each of which represents one point. We have three steps to input the data for Mahout clustering – firstly, you need to preprocess your data, then creates vectors from them, and finally save them in the SequenceFile format and input that to algorithm. In the case of points, no preprocessing is necessary as they are already vectors in the 2-dimensional plane. So we will need to convert them to a Vector class and save them as SequenceFile. We will give an overview of what SequenceFile can do but, a discourse into the details of the implementation and the format is beyond the scope of this book. For more details you can look at Hadoop in Action written by Chuck Lam or read the documentation from the hadoop website.

Listing 7.1 Sample input to our first clustering example

```
(1,1)
(2,1)
(1,2)
(2,2)
(3,3)
(8,8)
(8,9)
(9,8)
(9,9)
```

Examine the sample input given above. Figure 7.2 draws them on the X-Y plane. Two clusters stand out clearly; one cluster contains five points in the (1, 1), (3, 3) rectangular region, and other contains points in the (8, 8), (9, 9) rectangular region. The first step is to convert this input to a Mahout-readable format.

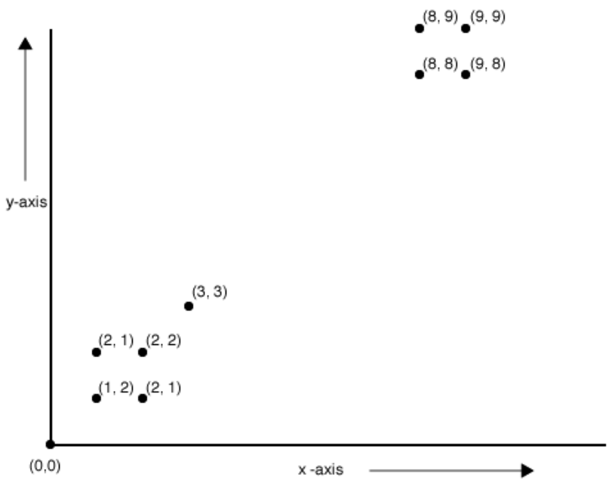


Figure 7.2 Plot of the input points in the x-y plane

We will need to represent these points as Vectors in Mahout. When you hear "vector" you may be recalling your high school physics course, where vectors were arrows and directions, not single points in space. For our purposes in machine learning, the term "vector" just refers to an ordered list of numbers, which is all a point or physics vector is anyway. Vectors have a number of dimensions (above, 2 dimensions) and a numeric value for each dimension.

Appendix A explains the Vector interface and its implementations in some detail; refer to it as needed to better understand how Mahout represents vectors. The details are not yet critical to our example, however.

In listing 7.2, we show clustering of 2-dimensional points using Mahout. The function getPoints converts the given set of input points to RandomAccessSparseVector format. Once the vectors are generated, they are written in the SequenceFile format for the clustering algorithms in mahout to read. The function writePointsToFile shows how it is done.

7.3.2 Using Mahout Clustering

Once the input is ready, we can cluster those nine points. In this example, we use the k-means clustering algorithm. The k-means clustering algorithm takes the following input parameters:

- The SequenceFile containing the input vectors.
- The SequenceFile containing the initial cluster centers. In our case we have seeded 2 clusters, hence 2 centers.

- The similarity measure to be used. We are using EuclideanDistanceMeasure as the measure of similarity. We will be exploring other kinds of similarity measures later in this chapter.
- The convergenceThreshold, if in a particular iteration, any centers of the clusters do not change beyond that threshold, then no further iterations are done.
- The number of iterations to be done.
- The number of reducers to be used. We will be using only 1. This is the value determining the parallelism of the execution. When we run this algorithm on a hadoop cluster, we will see how useful this parameter is.
- The Vector implementation used in the input files.

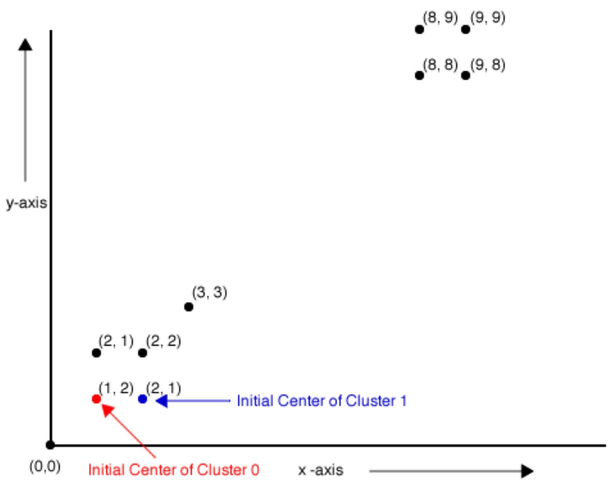


Figure 7.3 Marking the initial clusters is an important step in k-means clustering.

We have everything we need except the initial set of cluster centers. Since we are trying to generate two clusters from the nine points, we have to add two points in the initial set of centers as shown in Figure 7.3. This set serves as the best guess of the cluster center for the K-means algorithm. Of course, we can observe that these guesses aren't very good; both clearly fall within one of the apparent clusters. However in non-trivial examples there would be no way to know beforehand where the clusters like. There are various methods to estimate the centers of the clusters. Canopy clustering algorithm can do this estimation in a fast and efficient manner.

Even if the estimated centers are way off, the K-means algorithm would re-adjust it at the end of each iteration by computing the average center or the centroid of all points in the cluster. To demonstrate this corrective nature of K-means, we shall start with center points taken close together at (1, 1) and (2, 1).

Listing 7.2 SimpleKMeansClustering.java

```
public static final double[][] points = { {1, 1}, {2, 1}, {1, 2},
    {2, 2}, {3, 3}, {8, 8}, {9, 8}, {8, 9}, {9, 9}};

public static void writePointsToFile(List<Vector> points,
    String fileName, FileSystem fs, Configuration conf)
    throws IOException {
  Path path = new Path(fileName);
  SequenceFile.Writer writer = new SequenceFile.Writer(fs, conf,
      path, LongWritable.class, VectorWritable.class);
  long recNum = 0;
  VectorWritable vec = new VectorWritable();
  for (Vector point : points) {
    vec.set(point);
    writer.append(new LongWritable(recNum++), vec);
  }
  writer.close();
}


public static List<Vector> getPoints(double[][] raw) {
  List<Vector> points = new ArrayList<Vector>();
  for (int i = 0; i < raw.length; i++) {
    double[] fr = raw[i];

    Vector vec = new RandomAccessSparseVector("vector: "
        + String.valueOf(i), fr.length);

    vec.assign(fr);
    points.add(vec);
  }
  return points;
}



public static void main(String args[]) throws Exception {

  int k = 2;                                                     #1

  List<Vector> vectors = getPoints(points);

  File testData = new File("testdata");                          #2
  if (!testData.exists()) {
    testData.mkdir();
  }
  testData = new File("testdata/points");
  if (!testData.exists()) {
    testData.mkdir();
  }
  Configuration conf = new Configuration();
  FileSystem fs = FileSystem.get(conf);
  writePointsToFile(vectors, "testdata/points/file1", fs, conf);


  Path path = new Path("testdata/clusters/part-00000");       #3
  SequenceFile.Writer writer = new SequenceFile.Writer(fs, conf,
     path, Text.class, Cluster.class);

  for (int i = 0; i < k; i++) {
    Vector vec = vectors.get(i);

    Cluster cluster = new Cluster(vec, i);
    cluster.addPoint(cluster.getCenter());
    writer.append(new Text(cluster.getIdentifier()), cluster);
  }
  writer.close();


  KMeansDriver.runJob("testdata/points", "testdata/clusters",            #4
     "output", EuclideanDistanceMeasure.class.getName(), 0.001,
     10, 1);

  SequenceFile.Reader reader = new SequenceFile.Reader(fs,
     new Path("output/points/part-00000"), conf);

  Text key = new Text();
  Text value = new Text();
  while (reader.next(key, value)) {                                        #5
    System.out.println(key.toString() + " belongs to cluster "
       + value.toString());
  }
  reader.close();
}
```

#1 The number of clusters to be formed
#2 Create the input directories for the data
#3 Write the initial centers
#4 Run the K-means algorithm
#5 Read the output file and output the vector name and the cluster id it belongs to.

To get a clear picture of what we did in the sample code, take a look at the flow in figure 7.4.

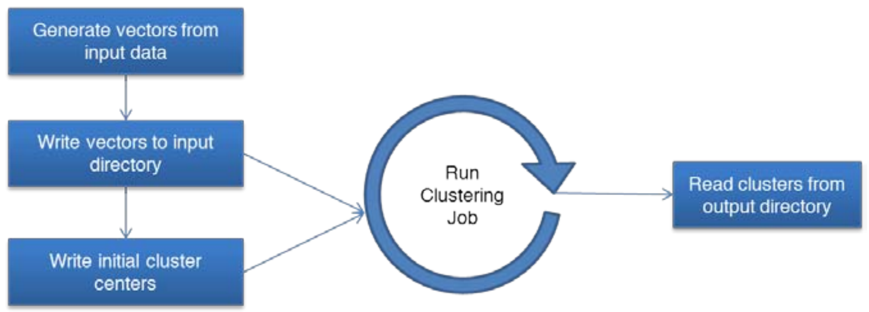

Generate vectors from input data
Write vectors to input directory
Write initial cluster centers
Run Clustering Job
Read clusters from output directory

Figure 7.4 Flow of the hello world example for clustering.

7.3.3 Analyzing the output

Compile and run this code using your favorite IDE or from the command line. Make sure you add all the Mahout dependency JAR files to the classpath. See the section on Maven compilation for more idea on packaging of your code with mahout classes and its dependencies.

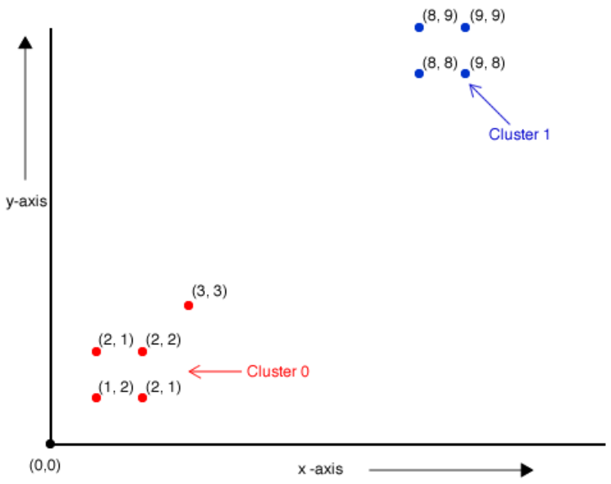
Since our data is small, in about 5-10 seconds, you will get the following output

```
vector: 0 belongs to cluster 0
vector: 1 belongs to cluster 0
vector: 2 belongs to cluster 0
vector: 3 belongs to cluster 0
vector: 4 belongs to cluster 0
vector: 5 belongs to cluster 1
vector: 6 belongs to cluster 1
vector: 7 belongs to cluster 1
vector: 8 belongs to cluster 1
```

We had marked each vector with a string identifier to uniquely identify it. This mechanism allows users to attach a unique identifier to each unit of data. This helps to evaluate and reconstruct the clusters later. As you see in figure 7.5, the algorithm was able to readjust the center of the cluster 1 from (2, 1) to (8.5, 8.5) – the centroid of all points in cluster 1.

In this simple example, Mahout clustered the points into two sets quickly and with great precision. Real world data is not as simple. With millions of such input vectors, each having millions of dimensions, clustering becomes quite non-trivial. Quality and performance issues arise. It will be difficult to decide question like how many clusters to produce, or what kind similarity measure should to choose. Tuning performance and even evaluating the quality of the clusters will need attention. Getting the perfect clustering is a never-ending task.

Mahout clustering implementations are configurable enough to fit the needs of most any clustering problem -- the question is of course which configuration is best! We will go into detail about various



parameters and the effect they have on clustering in the chapter Clustering Algorithms in Mahout. We will also examine some real world scenarios and show some techniques to improve both the clustering quality and performance.

Figure 7.5 The output of our hello world k-means clustering program. Even with distant centers, K-means algorithm was able to correctly iterate and correct the center based on Euclidean distance measure.

7.4 Exploring distance measures

In the above example, we used EuclideanDistanceMeasure to calculate the distance between points. While it proved to be an effective measure in generating the clusters we wanted, there are other similarity measure implementations in the Mahout clustering package. Aptly named as DistanceMeasure implementations, these classes calculate the distance between two vectors according to some definition of “distance”. Shorter distances indicate more similarity between the vectors and vice-versa; similarity and distance are related concepts.

7.4.1 Euclidean distance measure

The Euclidean distance, which we’ve already seen, is the simplest of all distance measures. It is the most intuitive and matches our normal idea of “distance”. For example, given two points in a plane, the Euclidean distance measure could be calculated by using a ruler to measure the distance between them. Mathematically, Euclidean distance between two n-dimensional vectors (a1, a2, … , an) and B (b1, b2, … ,bn) is:

$$d = \sqrt{(a1-b1)2 + (a2-b2)2 + \ldots + (an-bn)2}$$

The Mahout class that implements this measure is EuclideanDistanceMeasure.

7.4.2 Squared Euclidean distance measure

Just as the name suggests, this distance measure's value is just the square of the value returned by the Euclidean distance measure. For n-dimentional vectors (a1, a2, … , an) and (b1, b2, … ,bn) the distance becomes:

d = (a1-b1)2 + (a2-b2)2 + … + (an-bn)2

The Mahout class that implements this measure is SquaredEuclideanDistanceMeasure.

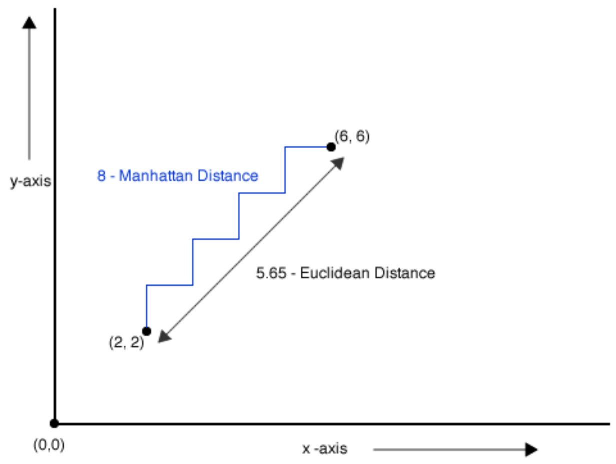


Figure 7.6 Difference between Euclidean and Manhattan distance measure. Euclidean distance measure gives 5.65 as the distance between (2, 2) and (6, 6) where as Manhattan distance is 8.0

7.4.3 Manhattan distance measure

Unlike Euclidean distance, under the Manhattan distance measure, the distance between any two points is the sum of the absolute differences of their coordinates. Figure 7.6 compares the Euclidean distance and Manhattan distance between two points in the X-Y plane. This distance measure takes its name from the grid-like layout of streets in Manhattan. As any New Yorker knows, you can't walk from 2nd Avenue and 2nd Street to 6th Avenue and 6th Street by walking straight through buildlings. The real distance walked is 4 blocks up and 4 blocks over. Mathematically, Manhattan distance between two n-dimentional vectors (a1, a2, … , an) and (b1, b2, … , bn) is:

d = (a1-b1) + (a2-b2) + … + (an-bn)

The Mahout class that implements this measure is ManhattanDistanceMeasure.

7.4.4 Cosine distance measure

The cosine distaince measure requires us to again think of points as like vectors from the origin to those points. These vectors form an angle θ between them, as illustrated in Figure 7.7.

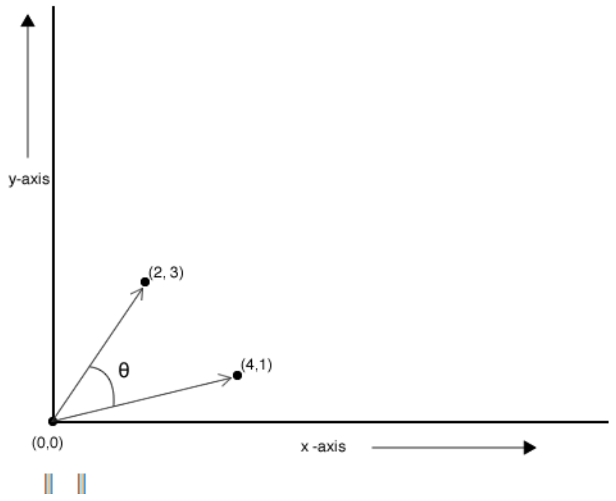


Figure 7.7 Cosine angle between the vectors (2,3) and (4, 1) as calculated from the origin

When this angle is small, the vectors must be pointing in somewhat the same direction, and so in some sense the points are “close”. The cosine distance just computes the cosine of this angle, which is near 1 when the angle is small, and decreases as it gets larger. It subtracts the cosine value from 1 in order to give a proper distance, which is 0 when close and larger otherwise.

The formula for cosine distance between n-dimensional vectors (a1, a2, … , an) and (b1, b2, … ,bn) is:

$$d = 1 - (a1b1 + a2b2 + \ldots + anbn) / (\sqrt{(a12 + a22 + \ldots + an2)}\sqrt{(b12 + b22 + \ldots + bn2)})$$

Note, this doesn’t account for the length of the two vectors; all that matters are that the points are in the same direction from the origin. Also note that the cosine distance measure ranges from 0.0 (two vectors along the same direction) to 2.0 (two vectors along opposite directions). The Mahout class that implements this measure is CosineDistanceMeasure.

7.4.5 Tanimoto distance measure

Cosine distance measure disregards the lengths of both vectors. This may work well for some data sets, but it will lead to poor clustering in others where the relative lengths of the vectors contain valuable information. For example, consider three vectors A (1.0, 1.0), B (3.0, 3.0) and C (3.5, 3.5). Even though they point in the same direction, the cosine distance is 0.0 for any two of these vectors. Cosine distance does not capture the fact that B and C are in a sense closer. The Euclidean distance measure would reflect this, but it doesn’t take account of the angle between the vectors, the fact that they’re “in the same direction”. We might want, at times, a distance measure that reflects both.

Tanimoto distance measure, also known as Jaccard's distance measure, captures the information about angle and the relative distance between the points. The formula for the Tanimoto distance between two n-dimentional vectors (a1, a2, … , an) and (b1, b2, … , bn) is:

p = (a1b1 + a2b2 + … + anbn)
d = 1 - p / (√(a12 + a22 + … + an2) + √(b12 + b22 + … + bn2) - p)

7.4.6 Weighted distance measure

Mahout also provides a WeightedDistanceMeasure class, and implementations of Euclidean and Manhattan distance measures using it. Weighted distance measure is an advanced feature in Mahout that allows you to give weights to different dimensions to either increase or decrease the effect of a dimension on the value of the distance measure. The weights in a WeightedDistanceMeasure need to be serialized to a file in a Vector format.

For example, when calculating distance between points in the X-Y plane, suppose we wished to make the x coordinate twice as significant. We would do so by doubling all x values, conceptually. To do this with a weighted distance measure, we would construct a weight `Vector` with value 2.0 in the 0th index (for x) and 1.0 the 1st index (for y). This will affect distance measures differently, but will in general make the distance value more sensitive to difference in x value.

7.5 Hello World Again! Trying out various distance measures

We will run the hello world K-means clustering example using Euclidean, Manhattan, Cosine and Tanimoto distance measure, with k=2 (producing two clusters). The results of various runs are tabulated in Table 7.1

The cosine distance measure clustering appears puzzling. From Figure 7.2, we see that only the point (2, 1) was at an angle greater than 45° from the x axis. The clustering algorithm chose to put all other points, at 45° and below, in one cluster. This doesn't mean that cosine distance measure is bad, but only that it doesn't work well on this data set. In domains such as text clustering, for instance, it can work well.

SquaredEuclideanDistanceMeasure actually increased the number of iterations. This is because absolute distance values became larger when using that measure, and we ran our algorithm using the same small value for the convergenceThreshold. So, it took a couple of iterations more for the convergence to occur.

Distance Measure	Number of iterations	Vectors[9] in Cluster 0	Vectors in Cluster 1
EuclideanDistanceMeasure	3	0, 1, 2, 3, 4	5, 6, 7, 8
SquaredEuclideanDistanceMeasure	5	0, 1, 2, 3, 4	5, 6, 7, 8

[9] These are the names/ids of the vectors as given in the Clustering Hello World source code

ManhattanDistanceMeasure	3	0, 1, 2, 3, 4	5, 6, 7, 8
CosineDistanceMeasure	1	1	0, 2, 3, 4, 5, 6, 7, 8
TanimotoDistanceMeasure	3	0, 1, 2, 3, 4	5, 6, 7, 8

Table 7.1 Result of clustering using various distance measures

In future chapters we will see more clustering methods and show how each of them is suited for various kinds of data, and optimize them using various distance measures for both speed and quality.

7.5 Summary

In this chapter, we introduced the idea of clustering. We used an intuitive approach to cluster books in a library. We formalized notions of clustering using points in two dimensions. We created a simple set of points in the plane and ran a simple K-means clustering example using EuclideanDistanceMeasure.

We then explained the various distance measures found in Mahout. Armed with these, we re-ran our example and compared the clusters generated using each of the distance measures.

Before studying clustering algorithms in detail, we need to spend some time with another foundational concept in Mahout in the next chapter: Representating Data.

8 Representing Data

This chapter covers:

- Representing data as a `Vector`
- Converting text documents into `Vector` form
- Normalizing data representations

This chapter explores ways of converting different kinds of objects into Vectors. A `Vector` is a very simplified representation of data that can help clustering algorithms understand the object and help compute that similarity with another object. To get good clustering, we need to understand the techniques in vectorization: the process of representing objects as vectors.

In the last chapter, we got a taste of clustering. Books were clustered together based on their similarity in words, and points in a two-dimensional plane were clustered together based on the distance between them. In reality, clustering could be applied to any kind of object provided we can distinguish similar and dissimilar items. Images could be clustered based on their colors, shapes in the image or both. We could cluster photographs to perhaps try to distinguish photos of animals from those of humans. We could even cluster species of animal by their average size, weight, number of legs and so on to discover groupings automatically.

As humans, we can cluster these objects because we understand them, and ”just know” what is similar and what isn’t. Computers unfortunately have no such intuition. So the clustering of anything via algorithms starts with representing the object in a way that can be read by computers.

It turns out that it is quite practical, and flexible, to think of objects in terms of their measurable features or attributes. For example, above, we identified size and weight as salient features that could help produce some notion of animal similarity. Each object (animal) has a numeric value for this feature.

So, we want to describe objects as sets of values, each associated to one of a set of distinct features, or dimensions -- does this sound familiar? We’ve all but described a vector again. While we’re accustomed to thinking of vectors as arrows or points in space, they’re just ordered lists of values. As such, they can easily represent objects.

We've already talked about how to cluster vectors in the previous chapter. But, how do we represent vectors in Mahout? And how do we get from objects to vectors in the first place? That's what we will see in the next section.

8.1 Representing vectors

You might have encountered the word "vector" in many contexts. In physics, a vector denotes the direction and magnitude of a force, or the velocity of a moving object like a car. In mathematics, a vector is simply a point in space. Both these concepts have the same representation. In two dimensions, any of these are represented as an ordered list of values, one for each dimension, like "(4, 3)". Both representations are illustrated in Figure 8.1. We often name the first dimension "x" and the second "y" when dealing with two dimensions, but this won't matter for our purposes in Mahout. As far as we're concerned, a vector can have two, three or ten thousand dimensions. The first one is dimension 0, the next is dimension 1 and so on.

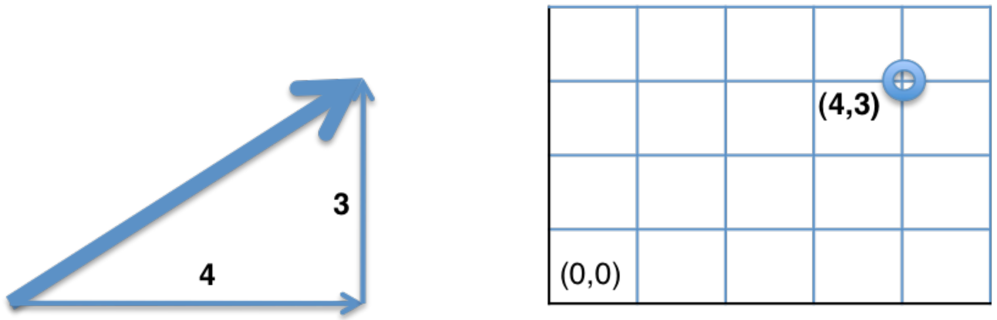


Figure 8.1 In physics, the vector can be thought of ray with a start point, direction and length and represents quantities like velocity and acceleration. In geometry or space, the vector is just a point denoted by weights along each dimension. The direction and magnitude of the vector is by default assumed to be a ray from the origin (0,0).

8.1.1 Understanding the difference between dense and sparse vectors

Vectors, as we've described them so far, are just an ordered list of values indexed by their dimension: a number. So, you may already have imagined one natural way to represent a vector in a programming language like Java: an array of numbers (doubles). In such a representation, the vector's value at dimension i would just be the value at array index i. This is a good way to represent a vector -- in some situations. We will call this a "dense" vector representation.

What's the alternative, then, to a "dense" vector representation? It is a "sparse" representation, but to understand the motivation for this alternative, we must point out a difference between the vectors you may be used to from physics and math, and those that are used to represent objects for classification.

It is common for a vector, for our purposes, to have a large number of dimensions, and to have no value in several of these many dimensions. "No value" here is like the programming concept of "null", but is represented as a zero value for a dimension in the vector. In physics or math, it's unusual to contemplate vectors with hundreds of dimensions, and, it's also unusual to think of vectors with mostly zero values, but this is a common sight when vectors are used for classification.

For such a vector, an array-based representation seems inefficient. The array would consist of mostly zeroes, with an occasional non-zero value. It would be more reasonable to only represent those dimensions with a non-zero value, not all of them. When dealing with vectors with millions of dimensions, with mostly zero values, the inefficiency of a dense representation becomes acute.

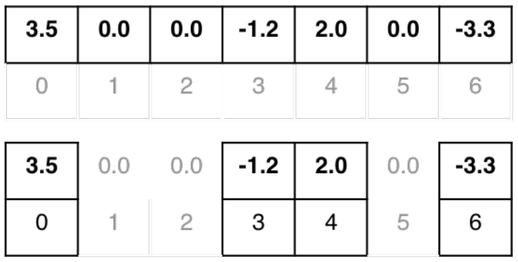


Figure 8.2 A dense and sparse vector representation. A dense vector is a represented as a full array, so it needs to store only the double values. Sparse vector saves on the space occupied by the zero valued cells but has an integer sized overhead for every non-zero double value.

Enter "sparse" vectors, which are backed by something more like a Java `Map`, mapping dimensions with a non-zero value to their values. While the memory required to store each dimension's value is higher than with a dense, array-backed representation, such a representation is superior when the number of non-zero dimensions is relatively low. Figure 8.2 illustrates the differences between a dense and sparse vector.

In Mahout, these ideas about vector representation are implemented as three different classes, each optimized for different scenarios. These `Vector` implementation classes are `DenseVector`, `RandomAccessSparseVector`, and `SequentialAccessSparseVector`.

`DenseVector` is backed by an array of doubles. Such a representation is quite memory efficient when a vector has few non-zero values. It allows quick access to any dimension's value, and quick iteration over all dimensions' values in order.

In `RandomAccessSparseVector`, the vector's values are stored in `HashMap`-like structure, where keys are `ints` and values are `doubles`. Only dimensions with non-zero values are stored, which improves memory-efficiency when a vector has many non-zero dimensions. Accessing a dimension's value is slightly slower as compared to a dense vector; iterating over dimensions in order is, however, much slower.

Compare this with `SequentialAccessSparseVector, where` the vector is represented with parallel arrays of `ints` and `doubles`. Due to this, the iteration over the vector in order by dimension is fast. But, random lookups and insertion of values are slower than with `RandomAccessSparseVector`.

These three implementations provide Mahout algorithms with an implementation whose performance characteristics suit the nature of the data, and way in which it is accessed. The choice of the implementation depends on the algorithm. If the algorithm does a lot of random insertions and updates of a vector's values, then an implementation with fast random access like `DenseVector` or

`RandomAccessSparseVector` would be appropriate. On the other hand, for an algorithm like K-Means clustering which calculates the magnitude of the vectors repeatedly, the sequential-access implementation will perform faster than the random-access sparse vector.

8.1.2 Transforming data into vectors

To cluster objects, those objects first must be converted into vectors, or “vectorized”. The vectorization process is unique to each type of data. Since we are dealing with clustering in this section, we will talk about this data transformation with clustering in mind. We hope that by now the representation of an object as an n-dimensional vector of some kind is easy to accept. Objects must first be construed as a vector having as many dimensions as the number of its features. Let’s understand this more with an example.

Say, we want to cluster a bunch of apples. They are of different shapes, different sizes, and different shades of red, yellow and green as shown in Figure 8.3. We define a distance measure, which says that two apples are similar if they differ in few features, and by a small amount. So a small, round, red apple is more similar to a small, round, green one than a large, ovoid green one.

The process of vectorization starts with assigning features to a dimension. Let’s say weight is feature (dimension) 0, color is 1, and size is 2. So the vector of a small round red apple looks like `[0 => 100 gram, 1 => red, 2 => small]`. But this “vector” doesn’t have all the numeric values yet, and it needs to.

For dimension 0, we need to express weight as a number. This could simply be the measured weight in grams or kilograms. Size, the dimension 2 doesn’t necessarily mean the same as weight. For all we know, the green apple could be denser than the red apple due to the freshness. Density/volume could be used provided we have the instrument to measure the same. Size on the other hand could even be user perceived numbers. Small sized apple could be of size value 1, medium could be 2, and large 3.

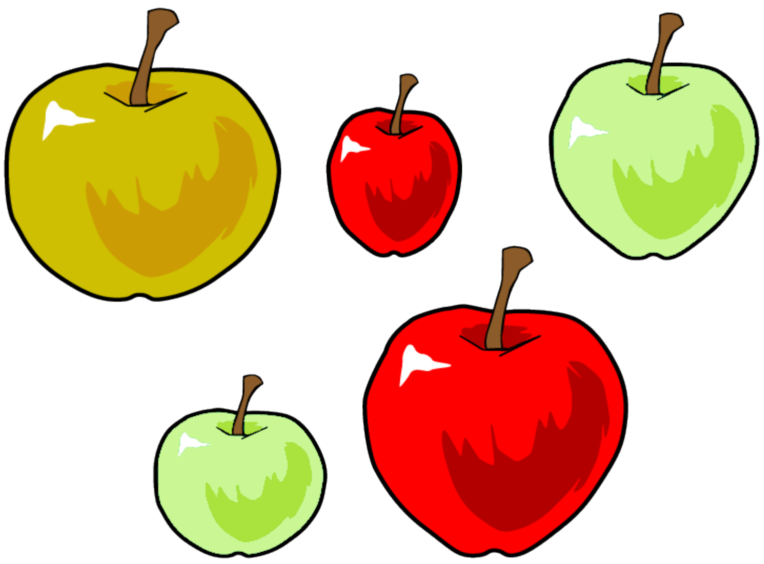


Figure 8.1 Apples of different sizes and colors needs to be converted into an appropriate vector form. The trick is to figure out how the different features of the apples translate into a decimal value.

What about color, the dimension 1? We could arbitrarily assign numbers to it, like red = 0.0, green = 1.0, yellow = 2.0. This is a crude representation; it will work in many cases but fails to reflect the fact that yellow is a color between red and green in the visible spectrum. We could fix that by changing mappings, but perhaps better would be to use something like the wavelength of the color (400nm - 650nm). This maps color to a meaningful and objective dimension value. Using these measures as properties of the apple, the vectors for some apples are described in table 8.1.

Apple	Weight (Kg) (0)	Color (1)	Size (2)	Vector
Small round green	0.11	510	1	[0.11, 510, 1]
Large oval red	0.23	650	3	[0.23, 650, 3]
Small Elongated red	0.09	630	1	[0.09, 630, 1]
Large round yellow	0.25	590	3	[0.25, 590, 3]
Medium oval green	0.18	520	2	[0.18, 510, 2]

Table 8.1 Set of apples of different weight, sizes and colors converted to vectors

If we weren't interested in clustering apples based on similarity in color shades, we could have kept each color in different dimensions. That is, red would be dimension one, green the dimension three, and yellow in the fourth dimension. If the apple is red, then red will have value 1 and the others zero. So, we could store these vectors in a sparse format and the distance measure would consider only the presence of non-zero value in these dimensions and cluster together those apples, which are of the same color.

One possible problem with our chosen mappings to dimension values is that dimension 1's values are much larger. If we applied a simple distance-based metric to determine similarity between these vectors, color differences would dominate the result. A relatively small color difference of 20nm is treated as equal to a huge size difference of 20cm. weighting the different dimensions solves this problem.

The importance of weighting is discussed in Section 8.2 where we try to generate vectors from text documents. The words in the document do not represent the document object to the same extend. The weighting technique helps magnify the weights of more important words and shrinks the least important ones.

Having established how to encode apples as vectors, we look at how in particular one prepares vectors for consumption by Mahout. An implementation of `Vector` is instantiated and filled in for each object; then, all `Vectors` are written to a file in the `SequenceFile` format, which is read by the Mahout algorithms. `SequenceFile` is a format from the Hadoop library, and encodes a series of key-value pairs. Keys must implement `WritableComparable` from Hadoop and values must implement `Writable`. These are Hadoop's equivalent of the Java's own `Comparable` and `Serializable` interfaces.

For our example, we will use the vector's name or description as a key, and the vector itself as the value. Mahout's Vector classes do not implement the `Writable` interface to avoid coupling them directly to Hadoop. However the `VectorWritable` wrapper class may be used to wrap a `Vector` and make it `Writable`. The Mahout `Vector` can be written to the `SequenceFile` using the `VectorWritable` class as shown in listing 8.1.

Listing 8.1 ApplesToVectors.java

```
public class ApplesToVectors {
 public static void main(String args[]) throws Exception {
    List<NamedVector> apples = new ArrayList<NamedVector>();

    NamedVector apple;
    apple = new NamedVector(                    A
        new DenseVector(new double[] {0.11, 510, 1}),
        "Small round green apple");
    apples.add(apple);
    apple = new NamedVector(
      new DenseVector(new double[] {0.2, 650, 3}),
        "Large oval red apple");
    apples.add(apple);
    apple = new NamedVector(
      new DenseVector(new double[] {0.09, 630, 1}),
        "Small elongated red apple");
    apples.add(apple);
    apple = new NamedVector(
      new DenseVector(new double[] {0.25, 590, 3}),
        "Large round yellow apple");
    apples.add(apple);
    apple = new NamedVector(
      new DenseVector(new double[] {0.18, 520, 2}),
        "Medium oval green apple");
    apples.add(apple);

    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(conf);

    Path path = new Path("appledata/apples");
    SequenceFile.Writer writer = new SequenceFile.Writer(fs, conf,
        path, Text.class, VectorWritable.class);
    VectorWritable vec = new VectorWritable();
    for (NamedVector vector : apples) {
      vec.set(vector);                                      B
      writer.append(new Text(vector.getName()), vec);
    }
    writer.close();

    SequenceFile.Reader reader = new SequenceFile.Reader(fs,
        new Path("appledata/apples"), conf);

    Text key = new Text();
    VectorWritable value = new VectorWritable();
    while (reader.next(key, value)) {
      System.out.println(key.toString() + " " + value.get().asFormatString());
    }
    reader.close();
  }
}
```

A Wrap the vector inside a NamedVector to assign a string name to it
B VectorWritable class helps serialize the vector data into the SequenceFile

Thus the process of selecting the features of an object and mapping them into a real number is known as feature selection. Since the basic data structure used in Mahout is vectors, the process of encoding features as a vector is named vectorization. Any kind of object can be converted to a vector form using reasonable approximations of the feature values, like it was done for apples. But now we turn to vectorizing one particularly interesting type of object: text documents.

8.2 Representing text documents as vectors

Text content in the digital form is exploding. The Google search engine alone indexes over 20 billion web documents. That's just a fraction of the publicly crawl-able information. The estimated size of text data (both public and private) could go well beyond petabytes range: that's a 1 followed by 15 zeros. There is a huge opportunity here for machine learning algorithms like clustering and classification to figure out structure and meaning in such an unstructured world and learning the art of text vectorization is the first step into it.

"Vector space model (VSM)" is the term for the common way of vectorizing text documents. First, imagine the set of all words that could be encountered in a series of documents being vectorized. This set might be all words that appear at least once in any of the documents. Imagine each word is assigned a number, which is the dimension it will occupy in document vectors.

For example, if the word “horse” is assigned to the 39,905th index of the vector, then the word “horse” will correspond to the 39,905th dimension of document vectors. A document’s vectorized form merely consists, then, of the number of times each word occurs in the document, stored the vector as its value along that word’s dimension. The dimension of these document vectors can be very large. The maximum dimensions possible is called the cardinality of the vector. Since the counts of all possible words or tokens are unimaginably large, text vectors are usually assumed to have infinite dimensions.

The value of the vector dimension for a word is usually the number of occurrence of the word in the document. This is known as term frequency weighting (TF). Note that values in a vector are also referred to as “weights” in this field; you may see references to “weighting” instead of values. The number of unique words that appear in one document is typically small compared to the number of unique words that appear in any document in a collection being processed. Hence, these high-dimension document vectors are quite sparse.

In clustering, we frequently try to find the similarity between two documents based on a distance measure. In typical English-language documents, the most frequent words will be “a”, “an”, “the”, “who”, “what”, “are”, “is”, “was” and so on. Such words are called stop-words. If we calculate the distance between two document vectors using any distance measure, we see that the distance value is dominated by the weights of these frequent words.

This is the same problem we noted before with apples and color. This effect is undesirable because it implies that two documents are similar mostly because words like “a”, “an”, and “the” occur in both. But, intuitively, we think of two documents as similar if they talk about similar topics, and words that signal a topic are usually the rare words like “enzyme” or “legislation” or “jordan” etc. This makes simple term-frequency based weighting undesirable for clustering and for applications where document similarity is to be calculated. Fortunately, weighting can be modified with a very simple but effect trick to fix these shortcomings as seen in the following sub-section.

8.2.1 Improving weighting with TF-IDF

Term frequency - inverse document frequency (TF-IDF) weighting is a widely used improvement on simple term frequency weighting. The “IDF” part is the improvement; instead of simply using term frequency as values in the vector, this value is multiplied by the inverse of the term’s document frequency. That is, its value is reduced to the extent that the word occurs frequently across documents.

To illustrate this, say that a document has words w1, w2, ..., wn with frequency f1, f2, …, fn. The term frequency (TFi) of a word wi is the frequency fi.

To calculate the inverse document frequency, first, the document frequency (DF) for each word is calculated. Document frequency is simply the number of documents the word occurs in. The number of times a word occurs in a document is not counted in document frequency. The inverse document frequency or IDFi for a word wi is:

$$IDF_i = 1 / DF_i$$

If a word occurs frequently in a collection of documents, its DF value is large and its IDF value is small. DF can be very large, and so the IDF value can be very small -- so small that it risks. In such cases its best to normalize the IDF score by multiplying it by a constant number. Usually we multiply it by the document count (N) and thus the IDF equation will look like:

IDFi = N / DFi

Therefore the weight of a word in a document vector is:

wi = TFi * IDFi = TFi * N / DFi

The IDF value in the above form is still not ideal, as it masks the effect of TF on the final term weight. To reduce this problem, a usual practice is to use the logarithm of the IDF value instead:

IDFi = log (N / DFi)

Thus the TF-IDF weight for a word wi becomes:

wi = TFi * log (N / DFi)

That is, the document vector will have this value at the dimension for word i. This is the classic TF-IDF weighting. Stop words get a small weight, and the terms that occur infrequently get a large weight. The “important” words or the topic words usually have a high TF and somewhat large IDF and so the product of the two becomes a larger value, thereby giving more importance to these words in the vector produced.

The basic assumption of vector space model is that the words are dimensions and therefore are orthogonal to each other. In other words, VSM assumes that occurrence of words are independent of each other, in the same sense that a point’s x coordinate is entirely independent of its y coordinate, in two dimensions. We know this is wrong in many cases. For example the word “Cola” has higher probability of occurrence along with the word “Coco” and therefore these words are not truly independent. Other models try to consider word dependencies. One well-known technique is Latent Semantic Indexing (LSI). LSI detects dimensions that seem to go together and merges them into a single one. Due to the reduction in dimension, this speeds up clustering computations. It improves the quality of clustering, as there is now a single good feature for the document object that dominates grouping really well. At the time of writing, Mahout does not yet implement this feature. However, TF-IDF has proved to work remarkably well even with the independence assumption. Mahout currently provides a solution to the problem of word dependencies using a method called collocation or n-gram generation, which is described in the following sub-section.

8.2.2 Accounting for word dependencies with n-gram collocations

A group of words in a sequence is called an n-gram. A single word can be called a unigram. Two words like “Coca Cola” can be considered a single unit and called a bigram. Three and more terms can be called trigrams, 4-grams, 5-grams and so on and so forth. Classic TF-IDF weighting assumes that the words occur independently of other words. The vectors created using this method usually lack the ability to identify key features of the document, which may be dependent.

To circumvent this problem, Mahout implements techniques to identify groups of words that have an unusually high probability of occurring together, such as “Martin Luther King Jr” or “Coca Cola”. Instead of creating vectors where dimensions map to single words (unigrams), we could as easily create vectors where dimensions map to bigrams -- or even both. TF-IDF can then work its magic as before.

From a sentence of multiple words, we can generate all n-grams by selecting sequential blocks of n words. This exercise will generate many n-grams, most of which do not represent a meaningful unit. For example, from the sentence “It was the best of times it was the worst of times”, we can generate the following bigrams:

“It was”

“was the”

“the best”

“best of”

“of times”

“times it”

“it was”

“was the”

“the worst”

“worst of”

“of times”

Some of these are good features (“the best”, “the worst”) for generating document vectors, but some of them aren’t (“was the”). If we combine the unigrams and bigrams from a document and generate weights using TF-IDF, we will end up with large vectors with many meaningless bigrams having large weights on account of their large IDF. This is quite undesirable. Mahout solves this by passing the n-grams through something called a log-likelihood test, which can determine whether two words occurred together rather by chance, or because they form a significant unit. It selects the most significant ones and prunes away the least significant ones. Using the remaining n-grams, TF-IDF weighting scheme is applied and vectors are produced. In this way, significant bigrams like “Coca Cola” can be more properly accounted for in a TF-IDF weighting.

In Mahout, text documents are converted to vectors using TF-IDF weighting and n-gram collocation using the `DictionaryVectorizer` class. In the next section we will show how starting from a directory full of documents one can create TF-IDF weighted vectors.

8.3 Generating vectors from documents

Now we examine two important tools that generate vectors from text documents. The first is the class `SequenceFilesFromDirectory`, which generates an intermediate document representation in `SequenceFile` format from text documents under a directory structure.

The second, `SparseVectorsFromSequenceFiles` uses the text documents in the `SequenceFile` format to convert the documents to vectors using either TF or TF-IDF weighting with n-gram generation. The intermediate `SequenceFile` is keyed by document ID; the value is the document text content. So starting from a directory of text documents with each file containing a full document, we will show how to convert them to vectors.

For the purpose of this example we will use the Reuters 21578 news collection[10]. It is a widely used dataset for machine learning research. The data was originally collected and labeled by Carnegie Group,

[10] http://www.daviddlewis.com/resources/testcollections/reuters21578/

Inc. and Reuters, Ltd. in the course of developing the CONSTRUE text categorization system. The Reuters 21578 collection is distributed in 22 files, each of which contains 1000 documents, except the last (reut2-021.sgm) that contains 578 documents.

The files are in SGML format, which is similar to XML. We could create a parser for the SGML files and write the document ID and document text into SequenceFiles, and use the vectorization tool above to convert them to vectors. However, a much quicker way is to re-use the Reuters parser given in the Lucene benchmark JAR file. Since its bundled along with Mahout, all we need to do is change to the examples/ directory under Mahout and run the class org.apache.lucene.benchmark.utils.ExtractReuters. Before doing this, download the Reuters collection from the website[11] and extract it in the reuters/ folder under examples/. Run the Reuters extraction code from the examples directory as follows:

```
mvn -e -q exec:java
-Dexec.mainClass="org.apache.lucene.benchmark.utils.ExtractReuters"
-Dexec.args="reuters/ reuters-extracted/"
```

Using the extracted folder, run the SequenceFileFromDirectory class. We can use the launcher script from the mahout root directory to do the same:

```
bin/mahout seqdirectory -c UTF-8
-i examples/reuters-extracted/ -o reuters-seqfiles
```

This will write Reuters articles in the SequenceFile format. Now the only step left is to convert this data to vectors. For that run the SparseVectorsFromSequenceFiles class using the Mahout launcher script:

```
bin/mahout seq2sparse -i reuters-seqfiles/ -o reuters-vectors -w
```

TIP

In Mahout, the -w flag is used to denote whether or not to overwrite the output folder. Since Mahout deals with huge datasets, it takes time to generate the output for each algorithm. This flag will prevent accidental deletion of any output that took hours to produce.

The seq2sparse command in the Mahout launcher script reads the Reuters data from SequenceFile and writes the vector generated by the dictionary based vectorizer to the output folder using the default options as given in Table 8.2. Inspect the folder produced using the command line:

```
$ls reuters-vectors/
dictionary.file-0
tfidf/
tokenized-documents/
vectors/
wordcount/
```

[11] http://www.daviddlewis.com/resources/testcollections/reuters21578/reuters21578.tar.gz

In the output folder we find a dictionary file and four directories. The dictionary file keeps the mapping between a term and its integer ID. This file is useful when reading the output of different algorithms, so we need to retain it. The other folders are intermediate folders generated during vectorization process, which happens in multiple steps, or MapReduce jobs.

In the first step, the text documents are tokenized or in other words: split into individual words using the Lucene `StandardAnalyzer` and stored in the `tokenized-documents/` folder. The word counting step, or the n-gram generation step (in this case only unigrams), iterates through the tokenized documents and generates a set of important words from the collection. The third step converts the tokenized documents into vectors using just the term-frequency weight, thus creating TF vectors. By default, the vectorizer uses the TF-IDF weighting, so two more steps happen after this: the document-frequency (DF) counting job, and the TF-IDF vector creation. The TF-IDF weighted vectorized documents are found in the `tfidf/vectors/` folder. For most applications, we need just this folder and the dictionary file.

Option	Flag	Description	Default Value
Overwrite (bool)	-w	If set, the output folder is overwritten. If not set, the output folder is created if the folder doesn't exist. If the output folder does exist, the job fails and an error is thrown. Default is unset.	N/A
Lucene Analyzer name (String)	-a	The class name of the analyzer to use	org.apache.lucene. analysis.standard.S tandardAnalyzer
Chunk size (int)	-chunk	The chunk size in megabytes. For large document collections (sizes in GBs and TBs) we will not be able to load the entire dictionary into memory during vectorization. So we split theexport MAVEN_OPTS=-Xmx1024m dictionary into chunks of the specified size and perform vectorization in multiple stages. Its recommended to keep this size to 80% of the Java heap size of the Hadoop child nodes to prevent the vectorizer from hitting the heap limit	100
Weighting (String)	-wt	The weighting scheme to use. tf for term frequency based weighting and tfidf for TF-IDF based weighting	tfidf
Minimum support (int)	-s	The minimum frequency of the term in the entire collection so as to be considered as a part of the dictionary file. Terms with lesser frequency are ignored	2

Minimum document frequency (int)	-minDF	The minimum number of documents the term should occur so as to be considered as a part of the dictionary file. Any term with lesser frequency is ignored	1
Max document frequency percentage (int)	-x	This is a mechanism to prune out high frequency terms or the stopwords. Any word that occurs in more than the specified percentage of documents out of the total number of documents in the collection is ignored from being a part of the dictionary	99
N-Gram size (int)	-ng	The max size of ngrams to be selected from the collection of documents.	1
Minimum Log Likelihood Ratio (LLR) (float)	-ml	This is a flag that works only when ngram size is greater than one. Very significant ngrams have large scores ~ 1000. Lesser significant ones have lower scores. While there is no specific method on how this value is chosen, the rule of thumb dictates that n-grams with LLR value < 1.0 are irrelevant.	1.0
Normalization (float)	-n	The normalization value to use in the Lp space. A detailed explanation of normalization is given in Section 8.4. Default scheme is not to normalize the weights	0
Number of reducers (int)	-nr	The number of reducer tasks to execute in parallel. This flag is useful when running dictionary vectorizer on a Hadoop cluster. Setting this to the maximum number of nodes in the cluster gives maximum performance. Setting this value higher than the number of cluster nodes lead to a slight decrease in performance. For more explanation read Hadoop documentation on setting the optimium number of reducers	1
Create sequential access sparse vectors (bool)	-seq	If set, the output vectors are created as `SequentialAccessSparseVectors`. By default the dictionary vectorizer generates `RandomAccessSparseVectors`. The former gives higher performance on certain algorithms like k-means and SVD due to the sequential nature of vector operations. By default the flag is unset.	N/A

Table 8.2 Flags of Mahout dictionary based vectorizer and their default values. To launch this run mahout laucher script as $MAHOUT_HOME/bin/mahout seq2sparse

Table 8.2 details all the important flags used in the dictionary-based vectorizer. Let us revisit the Reuters `SequenceFiles` and generate a vector dataset using non-default values. We will use the following non-default flag values:

- `org.apache.lucene.analysis.WhitespaceAnalyzer` to tokenize words based on the white-space characters between them. (`-a`)
- -Chunk size of 200MB. This value won't produce any effect on Reuters, as the dictionary sizes are usually in the range 1MB. (`-chunk`)
- Weighting method as "tfidf". (`-wt`)
- Minimum support 5.(`-s`)
- Minimum document frequency 3. (`-minDF`)
- Maximum document frequency percentage of 90% to prune away high frequency words aggressively. (`-x`)
- N-Gram size of 2 to generate both unigrams and bigrams. (`-ng`)
- Minimum value of log-likelihood ratio (LLR) is 50 to keep only very significant bigrams. (`-ml`)
- Normalization flag is unset (we will get back to this flag in the next section)
- Create `SequentialAccessSparseVectors` flag set (`-seq`)

Run the vectorizer using the above options in the Mahout launcher script

```
bin/mahout seq2sparse -i reuters-seqfiles/ -o reuters-vectors-bigram -w
-a org.apache.lucene.analysis.WhitespaceAnalyzer
-chunk 200 -wt tfidf -s 5 -md 3 -x 90 -ng 2 -ml 50 –seq
```

The dictionary file sizes from this vectorization job have increased from 654K to 1.2MB. Though we pruned away more unigrams based on frequency, we added almost double the amount of bigrams even after filtering using LLR threshold value. The dictionary size goes upto 2MB upon including trigrams. At least it has only grown linearly as we move from 2-grams to 3-grams and onwards; this is attributable to the LLR-based filtering process. Without this, the dictionary size would have grown exponentially.

At this point, you would be almost ready to try any clustering algorithm Mahout has to offer. There is just one more concept in text vectorization that is important to understand: normalization, which we explore next.

8.4 When normalization is needed

Normalization, here, is a process of cleaning up edge cases, data with unusual characteristics that skew results disproportionally. For example, when calculating similarity between documents based on some distance measure, it's not uncommon that some particular document shows up as quite similar to all the other documents. On closer inspection, we usually find that this happens because the document is large, and its vector has many non-zero dimensions, causing it to be "close" to many smaller documents. Somehow, we need to negate the effect of varying sizes of the vectors while calculating similarity. This process of decreasing the magnitude of large vectors and increasing the magnitude of smaller vectors is called normalization.

In Mahout, normalization uses what is known in statistics as a “p-norm”. For example, the p-norm of a 3-dimensional Vector [x, y, z] is:

$$[x/(|x|p + |y|p + |z|p)1/p, y/(|x|p + |y|p + |z|p)1/p, z/(|x|p + |y|p + |z|p)1/p]$$

The expression $(|x|p + |y|p + |z|p)1/p$ is known as the norm of a vector; here, we have merely divided each dimension’s value by this number. The parameter p here could be any value greater than zero. The 1-norm, or “Manhattan norm”, of a vector is the vector divided by the sum of the weights of all the dimensions:

$$[x/(|x| + |y| + |z|), y/(|x| + |y| + |z|), z/(|x| + |y| + |z|)]$$

The 2-norm, or “Euclidean norm” is the vector divided by the magnitude of the vector -- this magnitude is the “length” of the vector as we are accustomed to understanding it:

$$[x/\surd(x2 + y2 + z2), y/\surd(x2 + y2 + z2), z/\surd(x2 + y2 + z2)]$$

The infinite norm is simply the vector divided by the weight of the largest magnitude dimension:

$$[x/max(|x|, |y|, |z|), y/max(|x|, |y|, |z|), z/max(|x|, |y|, |z|)]$$

The norm power (p) to choose depends upon the type of operations done on the vector. If the distance measure used is Manhattan distance measure, the 1-norm will often yield better results with the data. Similarly, if the cosine of Euclidean distance measure is being used to calculate similarity, the 2-norm version of the vectors yields better results. That is to say, the normalization ought to relate to the notion of “distance” used in the similarity metric, for best results.

Note that the p in p-norm can be any rational number, so 3/4, 5/3, 7/5 are all valid powers of normalization. In the dictionary vectorizer the power is set using the –norm flag. A value “INF” means infinite norm. Generating the 2-normalized bigram vectors is as easy as running the Mahout launcher using the `seq2sparse` command with the `–n` flag set to 2:

```
bin/mahout seq2sparse -i reuters-seqfiles/ -o reuters-normalized-bigram -w
-a org.apache.lucene.analysis.WhitespaceAnalyzer
-chunk 200 -wt tfidf -s 5 -md 3 -x 90 -ng 2 -ml 50 –seq –n 2
```

Normalization improves the quality of clustering a little. Further refinement in the quality of clustering is achieved by the use to problem specific distance measures and appropriate algorithms. In the next chapter, we will take you on an elephant-ride past the various clustering algorithms in Mahout.

8.5 Summary

In this chapter we learned about the most important data representation scheme used by machine learning algorithms like clustering, the `Vector` format. There are two types of `Vector` implementations in Mahout, sparse and dense vectors. Dense vectors are implemented by the DenseVector class; RandomAccessSparseVector is a sparse implementation designed for applications requiring fast random reads and the SequentialAccessSparseVector is designed for applications required fast sequential reads.

We learned how to map important features of an object like an apple to numerical values and thereby create vectors representing different types of apples. The vectors were then written to and read form a SequenceFile, which is the format used by all the clustering algorithms in Mahout.

Text documents are frequently used in context of clustering. We saw how text documents could be represented as Vectors using the Vector Space Model. The TF-IDF weighting scheme proved to be a simple and elegant way to remove the negative impact of stop words during clustering. The assumption of independence of words in the classic TF-IDF weighting scheme removes some important features from text, but the collocation based n-gram generation in Mahout solves this problem to a great extent by identifying significant groups of words using a log likelihood ratio test. We saw how the Mahout dictionary-based vectorizer converted the Reuters news collections to vector with ease.

Finally we saw that the length of text documents negatively affects the quality of distance measures. The p-normalization method implemented in the dictionary vectorizer solves this problem by re-adjusting the weights of the vector by dividing by the p-norm of the vector.

Using the Reuters vector dataset, we can do clustering with different techniques, each having its pros and cons. We will explore these techniques in the next chapter on clustering algorithms.

9 Clustering Algorithms in Mahout

This chapter covers:

- K-Means clustering
- Centroid generation using Canopy clustering and K-Means++
- Fuzzy K-Means clustering, Dirichlet process clustering
- Topic modeling using LDA as a variant of clustering

Now that we know how input data is represented as `Vector`s and how `SequenceFile`s are created for input to the clustering algorithms, we are ready to explore the various clustering algorithms that Mahout provides. There are many clustering algorithms in Mahout, and some work well for a given dataset while others don't. K-Means is a very generic clustering algorithm, which can be molded easily to fit almost all situations. It's also simple to understand and can easily be executed on parallel computers.

Therefore, before going into the details of various clustering algorithms, it's best to get hands on experience using the K-Means algorithm. Then it becomes easier to understand the shortcomings and pitfalls and see how other techniques, though not so generic can help achieve better clustering of data. Simultaneously, we will use K-Means algorithm to cluster news articles and improve the quality using other techniques. Along the way, we will create a clustering pipeline for a news aggregation website to get a better feel of the real world problems in clustering. Finally, we will explore Latent Dirichlet Allocation (LDA) an algorithm, which closely resembles clustering, but achieves something far more interesting. There is a lot to cover, so let's not waste any time and jump right into the world of clustering through the K-means algorithm.

9.1 K-Means clustering

K-Means is to clustering as Vicks is to cough syrup. It's a simple algorithm and is more than 50 years old. Stuart Lloyd first proposed the standard algorithm in 1957 as a technique for pulse code modulation. However, it wasn't until 1982 before it got published[12]. It's widely used as a clustering

[12] Original Paper: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.131.1338

algorithm in many fields of science. The algorithm requires the user to set the number of clusters k as the input parameter.

9.1.1 Why only k?

K-Means algorithm puts a hard limitation on the number of clusters, k. This limitation might put a doubtful question mark on the quality of this method. Fear not, as this algorithm has proven to work very well for a wide range of real world problems over the last 25+ years of its existence. Even if the estimate of the value k is sub-optimal, the clustering quality is not affected much by it.

Say, we are clustering news articles to get top-level categories like politics, science and sports. For that we might want to choose a small value of k, which is in the range 10 – 20. If fine-grained topics are needed, a larger value of k like 50-100 is necessary. Say, there are one million news articles in our database and we are trying to find out groups of articles talking about the same story. The number of such related stories would be much smaller than the entire corpus maybe in the range of 100 articles per cluster. This means, we need a k value of 10000 to generate such a distribution. This will surely test the scalability of clustering and this is where Mahout shines at its best.

For good quality clustering using K-Means, we will need to estimate the value of k. An approximate way of estimating k is to figure it out based on the data we have, and the size of clusters we need. In the case above, if there are around 500 news articles published about every story, we should be starting our clustering with a k value like 2,000.

This is a crude way of estimating the number of clusters. Nevertheless, K-Means algorithm generates decent clustering even with this approximation. The type of distance measure used mainly determines the quality of K-Means clusters. In Chapter 7, we mentioned the various kinds of distance measures in Mahout. It's worthwhile to revise them to understand how it affects examples in this chapter.

9.1.2 All you need to know about K-Means

Let look at K-Means algorithm in detail. Suppose we have n points, which we need to cluster into k groups. K-Means algorithm will start with an initial set of k centroid points. The algorithm does multiple rounds of the processing and refines this centroid location till the iteration max-limit criterion is reached or until the centroids converge to a fixed point from which it doesn't move very much. A single K-Means iteration is illustrated clearly in Figure 9.1. The actual algorithm is a series of such iteration, till it encounters the criteria above.

There are two steps in this algorithm. The first step finds the points, which are nearest to each centroid point and assigns them to that specific cluster. The second step recalculates the centroid point using the average of the coordinates of all the points in that cluster. Such a two-step algorithm is a classic case of what is known as EM Algorithm (Expectation Maximization)[13]. The algorithm is a two-step process, which is processed repeatedly until convergence is reached. The first step, known as the expectation (E) step finds the expected points associated with a cluster. The second step known as the maximization (M) step improves the estimation of cluster center using the knowledge from the E step. A complete discourse on expectation maximization is beyond the scope of this book, but plenty of explanations and resources on EM are found online[14].

---

[13] http://en.wikipedia.org/wiki/Expectation-maximization_algorithm
[14] http://www.cc.gatech.edu/~dellaert/em-paper.pdf, Gives an easier explanation on EM Algorithm in terms of lower bound maximization.

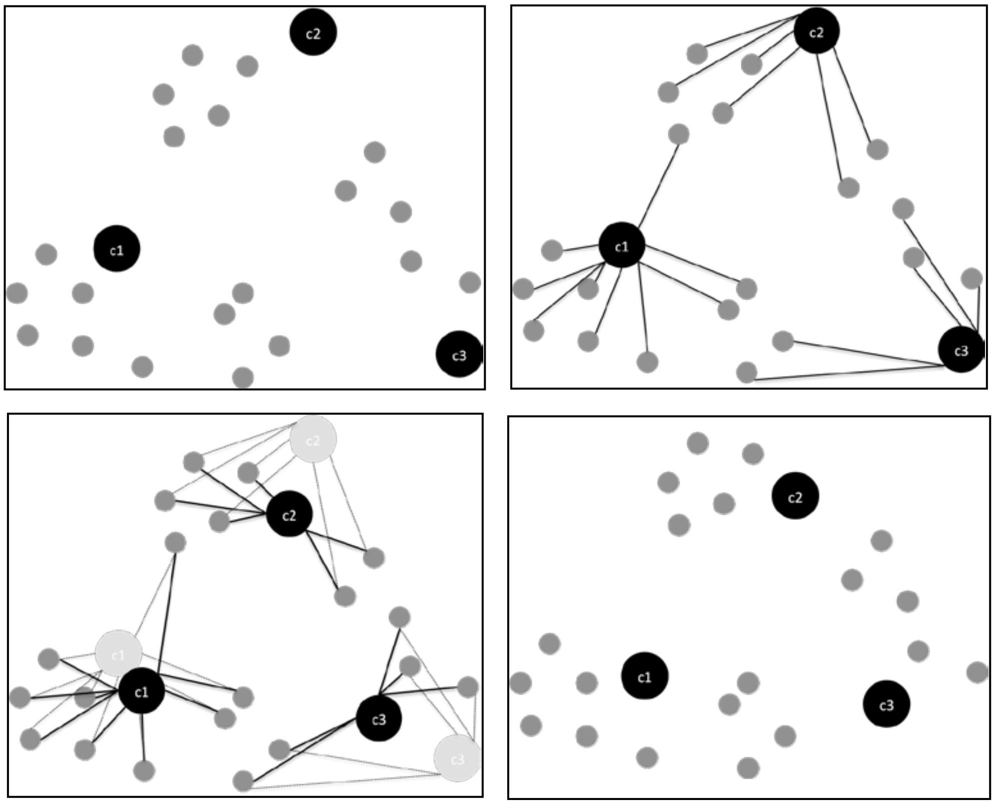


Figure 9.1 K-Means clustering in action. Starting with 3 random points as centroids (top-left), the Map stage (top-right) assigns each point to the cluster nearest to it. In the reduce stage (bottom-left), the associated points are averaged out to produce the new location of the centroid, leaving us with the final configuration (bottom-right). After each iteration, the final configuration is fed back in to the same loop till the centroids converge.

Now that we have understood K-Means technique, let's meet the important K-Means related classes in Mahout and run a simple clustering example.

9.1.3 Running K-Means clustering

The K-Means clustering algorithm is run using either the `KMeansClusterer` or the `KMeansDriver` class. The former one does an in-memory clustering of the points while the latter is an entry point to launch K-Means as a Map/Reduce job. Both methods can be run like a regular Java program and can read and write data from the disk. They can also be executed on a Hadoop cluster reading and writing data to a distributed file system.

For this example, we are going to use a random point generator function to create the points. It generates the points in the Vector format as a normal distribution around a given center. The points are scattered around in a natural manner. These points are going to be clustered using the in-memory K-Means clustering implementation in Mahout.

The `generateSamples` function in the listing 9.1 below takes a center say `(1,1)`, the standard deviation `(2)`, and creates a set of n (400) random points around the center, which behaves like a normal distribution. Similarly we will create two other sets with centers `(1, 0)` and `(0, 2)` and standard deviation `0.5` and `0.1` respectively. In listing 9.1, we ran the `KMeansClusterer` using the following parameters:

- The input points are in the `List<Vector>` format
- The `DistanceMeasure` is `EuclideanDistanceMeasure`
- The threshold of convergence is `0.01`
- The number of clusters k is `3`
- The clusters were chosen using a `RandomSeedGenerator` as in the hello-world example of Chapter 7

9.1 In-memory clustering example using the K-Means algorithm

```
private static void generateSamples(List<Vector> vectors, int num,
  double mx, double my, double sd) {
     for (int i = 0; i < num; i++) {
       sampleData.add(new DenseVector(
         new double[] {
           UncommonDistributions.rNorm(mx, sd),
           UncommonDistributions.rNorm(my, sd)
         }
       ));
     }
   }
public static void KMeansExample() {
  List<Vector> sampleData = new ArrayList<Vector>();

  generateSamples(sampleData, 400, 1, 1, 3);                  #1
  generateSamples(sampleData, 300, 1, 0, 0.5);
  generateSamples(sampleData, 300, 0, 2, 0.1);

  List<Vector> randomPoints = RandomSeedGenerator.chooseRandomPoints(
    points, k);
     List<Cluster> clusters = new ArrayList<Cluster>();

     int clusterId = 0;
     for (Vector v : randomPoints) {
       clusters.add(new Cluster(v, clusterId++));
     }

     List<List<Cluster>> finalClusters = KMeansClusterer.clusterPoints(
       points, clusters, new EuclideanDistanceMeasure(), 3, 0.01); #2
     for(Cluster cluster : finalClusters.get(finalClusters.size() - 1)) {
    System.out.println("Cluster id: " + cluster.getId() + " center: "
                           + cluster.getCenter().asFormatString());   #3
     }
   }
```

Cueball

#1 Generate 3 sets of points each with a different center and standard deviation

#2 Run `KMeansClusterer` using the `CosineDistanceMeasure`

#3 Read the center of the cluster and print it.

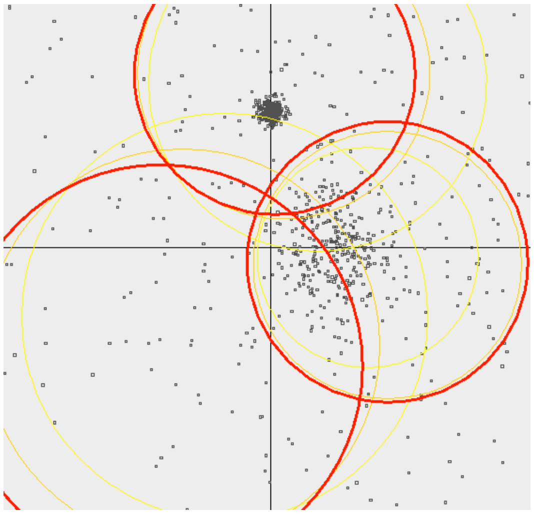


Figure 9.2 K-Means Clustering. We start with k as 3 and try to cluster 3 normal distributions we have generated. The thin lines denote the clusters estimated in previous iterations, here we can clearly see the clusters shifting

The `DisplayKMeans` class kept in the “examples” folder of the Mahout code is a great tool to visualize the algorithm in a 2-dimensional plane. It shows how the clusters shift their position after each iteration. It is also a great example of how clustering is done using `KMeansClusterer`. Just run the `DisplayKMeans` as a Java Swing application and view the output of the example as given in Figure 9.2.

Note that the K-Means in-memory clustering implementation works with list of `Vector` objects. The amount of memory used by this program depends on the total size of all the vectors. The sizes of clusters are larger as compared to the size of the vectors in the case of sparse vectors or the same size for dense vectors. As a rule of thumb, assume that number of vectors that could be fit in memory equals the number of data points + k centers. If the data is huge, we cannot run this implementation.

This is where Map/Reduce shines. Using Map/Reduce infrastructure, we can split this clustering algorithm to run on multiple machines, with each Mapper getting a subset of the points and nearest cluster computed in a streaming fashion.

The Map/Reduce version is designed to run on a Hadoop cluster. Nevertheless, it runs quite efficiently without it. Mahout is compiled against Hadoop code; that means, we could run the same implementation without a Hadoop cluster directly from within Java or from the command line.

UNDERSTANDING THE K-MEANS CLUSTERING MAP/REDUCE JOB

In Mahout, the Map/Reduce version of K-Means algorithm is instantiated using the `KMeansDriver` class. The class has just a single entry point - the `runJob` method. We have already seen K-Means in action in Chapter 7. The K-Means clustering algorithm takes the following input parameters:

- The `SequenceFile` containing the input `Vectors`

- The SequenceFile containing the initial Cluster centers
- The similarity measure to be used. We will use EuclideanDistanceMeasure as the measure of similarity and experiment with the others later
- The convergenceThreshold, if in an iteration, each centroid does not move a distance more than this value, no further iterations are done and clustering stops
- The number of iterations to be done. This is a hard limit; the clustering stops if this threshold is reached
- The number of reducers to be used. This value determines the parallelism in the execution of the job. One a single machine a value of 1 is set. When we run this algorithm on a Hadoop cluster, we will show how useful a parameter this is

Mahout algorithms never modify the input directory. This gives us the flexibility to experiment with the various parameters of the algorithm. From a Java code, we can call the entry point as given in listing 9.2 to initiate clustering of data from the file-system.

Listing 9.2 The K-Means clustering job entry point

```
KMeansDriver.runJob(inputVectorFilesDirPath, clusterCenterFilesDirPath,
     outputDir, EuclideanDistanceMeasure.class.getName(),
     convergenceThreshold, numIterations, numReducers);
```

TIP

Mahout reads and writes data using the Hadoop FileSystem class. This provides seamless access to both the local file system (via java.io) and the distributed file systems like HDFS, S3FS (using internal Hadoop classes). This way the same code that works on the local system, will also work on the Hadoop file system on the cluster, provided the path to the Hadoop configuration files are correctly set in the environment variables. In Mahout, the shell script, bin/mahout finds the Hadoop configuration files automatically from the environment variable $HADOOP_CONF

We will use the SparseVectorsFromSequenceFile tool to vectorize documents stored in SequenceFile to vectors. Refer to the vectorization section 8.3 to know more about this tool. Since K-Means algorithm needs the user to input the k initial centroids, the Map/Reduce version needs us to input the path on the file system where these k centroids are kept. To generate the centroid file, we can write a custom logic to select the centroids as we did in the hello world example in listing 7.2 or let Mahout generate the random k centroids for us as detailed next.

RUNNING K-MEANS JOB USING RANDOM SEED GENERATOR

Let's run K-Means clustering over the vectors generated from the Reuters-21578 news collection as described in section 8.3. The collection was converted to a Vector dataset and weighted using Tf-Idf measure. Reuters' collection has many topic categories. Therefore, we will set k as 20 and try to see how K-Means can cluster the broad topics in the collection. For running K-Means clustering, our mandatory checklist includes:

- The Reuters dataset in the Vector format
- The RandomSeedGenerator that will seed the initial centroids

- The SquaredEuclideanDistanceMeasure
- A large value of convergenceThreshold (1.0), since we are using the squared value of the Euclidean distance measure
- The maxIterations set as 10
- The number of reducer set as 1
- The number of clusters k set as 20

If we use the DictionaryVectorizer to convert text into vectors with more than one reducer, the dataset of vectors in SequenceFile format are usually found split into multiple chunks. KMeansDriver reads all the files from the input directory assuming they are SequenceFiles. So, don't bother about the split chunks of vectors.

The same is true for the folder having the initial centroids. The centroids may be written in multiple SequenceFile files and Mahout takes care of reading through all of them. This feature is particularly useful when having an online clustering system where data is inserted in real time. Instead of appending to the already existing file, a new chunk can be created independently and written into, without affecting the algorithm.

CAUTION

KMeansDriver accepts an initial cluster centroid folder as a parameter. It expects a SequenceFile full of centroids only if the –k parameter is not set. If the parameter is specified, the driver class will erase the folder and write randomly selected k points to a SequenceFile there.

KMeansDriver is also the main entry point to launch the K-Means clustering of Reuters-21578 news collection. From the Mahout examples directory execute the Mahout launcher from the shell with *"kmeans"* as the program name. This driver class will randomly select k cluster centroids using RandomSeedGenerator and then run the K-Means clustering algorithm:

```
$ bin/mahout kmeans -i reuters-vectors -c reuters-initial-clusters  \
-o reuters-kmeans-clusters \
-m org.apache.mahout.common.distance.SquaredEuclideanDistanceMeasure \
-r 1 -d 1.0 -k 20
```

We are using maven Java execution plug-in to execute K-Means clustering with the required command line arguments. The argument –k 20 is set implies that the centroids are randomly generated using RandomSeedGenerator and written to the input clusters folder.

TIP

We can see the complete details of the command line flags and the usage of any Mahout package by setting the –h or --help command line flag.

In the command-line, the number of reducers –r 1 parameter or the distance measure SquaredEuclideanDistanceMeasure need not be mentioned as they are set by default. Once the command is executed, clustering iterations will run one by one. Be patient and wait for the centroids to converge. An inspection of Hadoop counters printed at the end of a Map/Reduce can tell how many of the centroids have converged as specified by the threshold:

```
…
…
INFO: Counters: 14
May 5, 2010 2:52:35 AM org.apache.hadoop.mapred.Counters log
INFO:   Clustering
May 5, 2010 2:52:35 AM org.apache.hadoop.mapred.Counters log
INFO:     Converged Clusters=6
May 5, 2010 2:52:35 AM org.apache.hadoop.mapred.Counters log
…
…
```

It takes a couple of minutes to run clustering over the Reuters data with the above parameters. Had the clustering been done in-memory, it would have finished in under a minute. The same algorithm over the same data as Map/Reduce job takes a couple of minutes. This increase in timing is caused by the overhead of the Hadoop library. The library takes does many checks before starting any map or reduce task. However, once it starts, Hadoop mappers and reducers run at full speed. This overhead slows down the performance on a single system. On a cluster, the negative effect of this starting delay is negated by the reduction in processing time due to the parallelism.

Lets get back to the console where K-Means is running. After multiple Map/Reduce jobs, the K-Means clusters converge and clustering end and the points and cluster mappings are written to the output folder.

TIP

When we deal with terabytes of data that can't be fit in memory, the Map/Reduce version is able to scale to the size by keeping the data on the Hadoop distributed file system and running the algorithm on large clusters. So, if the data is small, and fits in the RAM, use the in-memory implementation. If the data grows and reaches a point where it can't fit it into the memory anymore, we will have to start using the Map/Reduce version and think of moving the computation to a Hadoop cluster. Check out the appendix C to find out more on setting up a Hadoop cluster on a Linux box.

The K-Means clustering implementation creates two types of directories in the output folder. The clusters directory "`clusters-`" is formed at the end of each iteration, which has the information about the clusters like centroid, standard deviation and other things. The `clusteredPoints` directory, on the other hand has the final mapping from cluster-id to document-id. This data is generated as per the cluster information from the output of the last Map/Reduce operation. The directory listing of the output folder looks something like this:

```
$ ls –l reuters-kmeans-clusters
drwxr-xr-x  4 user  5000  136 Feb 1 18:56 clusters-0
drwxr-xr-x  4 user  5000  136 Feb 1 18:56 clusters-1
drwxr-xr-x  4 user  5000  136 Feb 1 18:56 clusters-2
…
drwxr-xr-x  4 user  5000  136 Feb 1 18:59 clusteredPoints
```

The `clusters-0` folder is generated after the first iteration, the `clusters-1` folder after the second iteration and so on. Now that the clustering is done, we need a way to inspect the clusters and see how they are formed. Mahout has a utility called the `org.apache.mahout.utils.clustering.ClusterDumper` that can read the output of any clustering algorithm and show the top terms in each cluster and the documents belonging to that cluster. To execute cluster dumper run the following:

```
$ bin/mahout clusterdump -dt sequencefile \
-d reuters-vectors/dictionary.file-* \
-s  reuters-kmeans-clusters/clusters-19 -b 0
```

The Cluster dumper takes dictionary file as the input. This is used to convert the feature ids or dimensions of the `Vector` into words. Running `ClusterDumper` on the output folder corresponding to the last iteration produces an output similar to the one given below.

```
Id: 11736:
     Top Terms: debt, banks, brazil, bank, billion, he, payments, billion dlrs,
interest, foreign
Id: 11235:
     Top Terms: amorphous, magnetic, metals, allied signal, 19.39, corrosion, allied,
molecular, mode, electronic components
…
Id: 20073:
     Top Terms: ibm, computers, computer, att, personal, pc, operating system, intel,
machines, dos
```

The reason why it is different for different runs is that a random seed generator was used to select the k centroids. The output depends heavily on the selection of these centers. Inspecting the output above, the cluster with id `11736` has top words like bank, brazil, billion, debt etc. Most of the articles that belong to this cluster talk about news associated with these words. Note that the cluster with id `20073` talks about computers, ibm, att, pc etc. The news articles associated with that cluster evidently talks about computers and related companies.

Thus, we have achieved a decent clustering using a distance measure like `SquaredEuclideanDistanceMeasure`. However, it took us 15+ iterations to get there. What's peculiar about text data is that two documents that are similar in content don't necessarily need to have the same length. The Euclidean distance between two similar document of different sizes and about the same topic is quite large. That is, the Euclidean distance is affected more by the difference in the number of words between the two documents, and less by the words common to both of them. Visit the Euclidean distance equation from section 7.4.1 and try to understand its behavior by experimenting with it.

The reasons stated above makes Euclidean distance measurement a misfit for text documents. Take look at a cluster in the Cluster dumper output. This shows a cluster that was created because of the Euclidean distance metric:

```
Id: 20978:
     Top Terms: said, he, have, market, would, analysts, he said, from,
which, has
```

This cluster really doesn't make any sense, especially with words like "said", "he" or "the". To really get good clustering for a given dataset, we have to experiment with the different distance measures available in Mahout as given in section 7.4 and see how it performs on the data we have

We now know that Cosine and Tanimoto measures work well for text documents since they depend more on the common words and less by the un-common words. The only way to evaluate that is to try it on our Reuters dataset and see. Let's run K-Means with `CosineDistanceMeasure`:

```
$ bin/mahout kmeans -i reuters-vectors -c reuters-initial-clusters  \
-o reuters-kmeans-clusters \
-m org.apache.mahout.common.distance.CosineDistanceMeasure \
-r 1 -d 0.1 -k 20
```

Note that convergence threshold was set to 0.1, instead of the default value of 0.5 as cosine distances lie in between 0 and 1. When the program runs, one peculiar behavior is noticeable: the clustering speed slowed down a bit due to the extra calculation involved when using cosine distance, but the whole clustering converges within a few iterations as compared to 15+ used by squared Euclidean distance measure. This clearly indicates that cosine distance gives a better notion of similarity between text documents than Euclidean distance. Once clustering finishes, run the `ClusterDumper` against the output and inspect some of the top words in each cluster. Some of the interesting clusters are shown below:

```
Id: 3475:name:
     Top Terms: iranian, iran, iraq, iraqi, news agency, agency, news, gulf, war,
offensive
Id: 20861:name:
     Top Terms: crude, barrel, oil, postings, crude oil, 50 cts, effective, raises,
bbl, cts
```

Experiment with Mahout K-Means and find out the combination of `DistanceMeasure` and `convergenceThreshold` that gives the best clustering for the given problem. Try them on various kinds of data and see how things behave. Explore the various distance measures in Mahout or try and make one on your own. Though K-Means runs impeccably well using randomly seeded clusters, the final centroid locations still depend on their initial positions.

K-Means algorithm is an optimization technique. Given the initial conditions, K-Means tries to put the centers at their optimal position. But it is a greedy optimization, which causes it to find the local minima. There can be other centroids positions that satisfy the convergence property and some of them might be better than the result we just got. Though, we may never find the perfect clusters, we can apply two powerful techniques that will takes us closer to it. They are called Canopy clustering and K-Means++ and they are discussed in the following section.

9.1.4 Finding the perfect k using approximate clustering

For many real-world clustering problems, the number of clusters is not known beforehand, like the grouping of books in the library example from Chapter 7. A class of techniques known as approximate clustering algorithms can estimate the number of clusters as well as the approximate location of the centroids from a given dataset. Two notable methods that do this are Canopy generation and K-Means++ algorithm.

An algorithm that finds the number of clusters! Exciting? Well, hold on! They don't just magically run and find the solution for the clustering problem. They still have to be told what size clusters to look for and they will find the number of clusters that have such a size approximately.

REASON FOR HAVING A PERFECT SET OF K CENTROIDS

K-Means algorithm in Mahout generates the SequenceFile containing the k vectors using the `RandomSeedGenerator` class as we saw earlier. While random centroid generation is fast, there is no guarantee that it will generate good estimates for centroids of the k clusters. Centroid estimation affects the run time of K-Means a lot. Good estimates help the algorithm to converge faster and use less number of passes over the data. We will see two techniques to select k as well as the centroid vectors for K-Means – Canopy generation and K-Means++

9.1.5 Seeding K-Means centroids using Canopy generation

Canopy generation also known as Canopy clustering is a fast approximate clustering technique. It's used to divide the input set of points into overlapping clusters known as canopies. The word "canopy" by definition is an enclosure. For us it is nothing but an enclosure of points or just a cluster. Canopy clustering tries to estimate the approximate cluster centroids or the canopy centroids using two distance thresholds T1 and T2, with T1>T2.

Canopy clustering strength lies in its ability to create clusters very very fast. It can do this with a single pass over the data. But its strength is also its weakness. This algorithm may not give accurate and precise clusters. But, it can give the optimal number of clusters without even specifying the number of clusters k like in K-Means.

The algorithm uses a fast distance measure and two distance thresholds T1 and T2, with T1>T2. It begins with a dataset of points and an empty list of canopies. It just iterates over the dataset, creating canopies in the process. During each iteration, it removes a point from the dataset and adds a canopy into the list with that point as the center. It loops through the rest of the points one by one. With each one, it calculates the distances to all the canopy centers in the list. If the distance of the point to any canopy center is within T1, it is added into that canopy. If the distance is within T2, it is removed from the list and thereby prevented from forming a new canopy in the subsequent loops. We repeat this process until the list is empty.

We prevent all points close to an already existing canopy (distance < T2) from being the center of a new canopy. We really don't want the formation of another redundant canopy in close proximity. Figure 9.3 illustrates the canopies created using this method. The clusters formed depends only on the choice of distance thresholds.

UNDERSTANDING CANOPY GENERATION ALGORITHM

The canopy generation algorithm is executed using the `CanopyClusterer` or the `CanopyDriver` class. The former one does an in-memory clustering of the points while the latter is an implementation of it as Map/Reduce jobs. These jobs can be run like a regular Java program and can read and write data from the disk. They can also be run on a Hadoop cluster reading and writing data to a distributed file system.

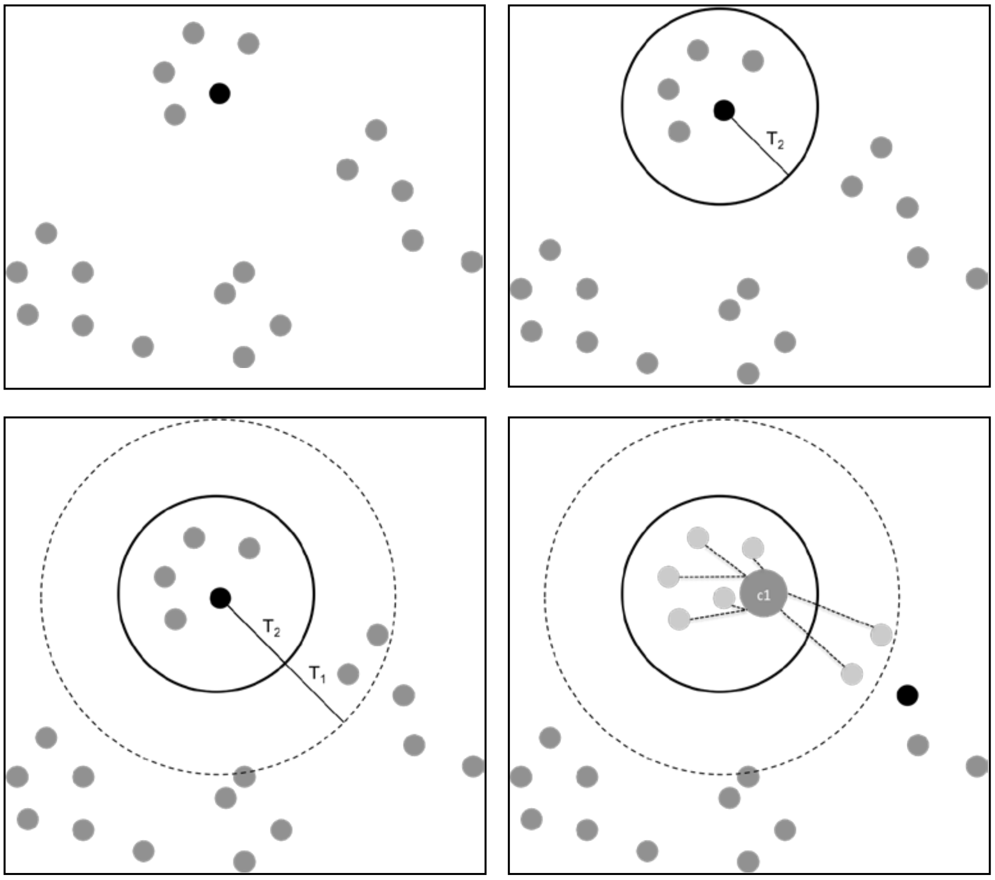


Figure 9.3 Canopy Clustering. If we start with a point (top left) and mark it as part of a canopy, then all the points within a distance T2 (top right) are removed from the dataset and prevented from becoming a new canopy. The points within outer circle (bottom-right) are also put in the same canopy but they are allowed to be part of other canopies. This assigning process is done in a single pass on a Mapper. The Reducer computes average of the centroid (bottom right) and merges close canopies.

We are going to use the same random point generator function as earlier to create vectors in scattered in the 2-dimensional plane like in a normal distribution. In listing 9.3, we ran the in-memory version of Canopy using the `CanopyClusterer` with the following parameters:

- The input `Vector` data in the `List<Vector>` format
- The `DistanceMeasure` is `EuclideanDistanceMeasure`
- The value of T1 is `3.0`
- The value of T2 is `1.5`

9.3 In-memory example of Canopy generation algorithm

```
public static void CanopyExample() {
  List<Vector> sampleData = new ArrayList<Vector>();

  generateSamples(sampleData, 400, 1, 1, 2);                    #1
  generateSamples(sampleData, 300, 1, 0, 0.5);
  generateSamples(sampleData, 300, 0, 2, 0.1);


  List<Canopy> canopies = CanopyClusterer.createCanopies(
    points, new EuclideanDistanceMeasure(), 3.0, 1.5);

     for(Canopy canopy : canopies) {
    System.out.println("Canopy id: " + canopy.getId() + " center: "
                          + canopy.getCenter().asFormatString());   #3
     }
   }
```

#1 Generate 3 sets of points with different parameters
#2 Run CanopyClusterer using the EuclideanDistanceMeasure
#3 Read the center of the canopy and print it.

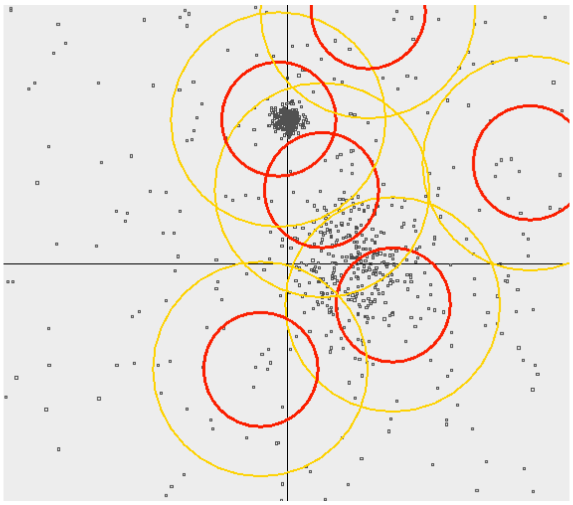


Figure 9.4 An example of in-memory Canopy Generation visualized using the DisplayCanopy class. We start with T1=3.0 and T2=1.5 and try to cluster the 3 normal distributions that are synthetically generated.

The `DisplayCanopy` class in the “examples” folder of the Mahout code displays a set of points in a 2-dimensional plane and shows how the canopy generation is done using in-memory `CanopyClusterer`. A typical output of the `DisplayCanopy` is given in the figure 9.4 on the next page.

Canopy clustering is non parametric with respect to the number of cluster centroids. The number of centroids formed depends only on the choice of distance measure, T1 and T2. The Canopy in-memory clustering implementation works with list of Vector objects just like the K-Means implementation. If the dataset is huge, we can never run this algorithm on a single machine and would have to rely on the Map/Reduce job. The Map/Reduce version of canopy clustering implementation does a slight approximation as compared to the in-memory one and thus produces a slightly different set of canopies for the same input data. This is nothing to be alarmed about when the data is huge. The output canopy clustering is a great starting point for K-Means, which improves clustering due to the increased precision of the initial centroids as compared to random selection.

Using the canopies we generated above, we can assign points to the nearest canopy center, thus in theory cluster the set of points. This is called canopy clustering instead of canopy generation. In Mahout, the CanopyDriver class does both canopy centroid generation and an optional clustering if the runClustering parameter is set to true. Next, we will try and run Canopy generation on the Reuters collection and figure out the value of k.

RUNNING CANOPY GENERATION ALGORITHM TO SELECT K CENTROIDS

We are going to generate Canopy centroids from the Reuters Vector dataset. For the centroid generation, we will use the distance measure as EuclideanDistanceMeasure and the threshold values t1=2000 and t2=1500. Remember that Euclidean distance measure gives very large distance values for sparse document vectors so large values for t1 and t2 are necessary to get meaningful clusters.

The distance threshold values t1 and t2 that we chose above produces less than 50 centroid points for the Reuters collection. We estimated these threshold values after running the CanopyDriver multiple times over the input data. Due to the fast nature of canopy clustering, we are at liberty to experiment with various parameters and are able to see the results much quicker than we would have had if we were using expensive techniques like K-Means. To run canopy generation over Reuters; execute the canopy program using the Mahout launcher as follows:

```
$bin/mahout canopy -i reuters-vectors -o reuters-canopy-centroids \
-m org.apache.mahout.common.distance.EuclideanDistanceMeasure \
-t1 1500 -t2 2000
```

Within a minute CanopyDriver will generate the centroids in the output folder. We can inspect the Canopy centroids using the cluster dumper utility as we did for K-Means earlier in this chapter. Next, we will use this set of centroids to improve K-Means clustering.

IMPROVING K-MEANS CLUSTERING USING CANOPY CENTERS

We are ready to run the K-Means clustering algorithm using the canopy centroids we just generated. For that, all we need to do is to set the clusters parameter (-c) to this folder and remove the –k command line parameter in the KMeansDriver. Remember that, if –k flag is set, the RandomSeedGenerator will overwrite the canopy centroid folder. We will be using the TanimotoDistanceMeasure in K-Means to get clusters as follows:

```
$bin/mahout kmeans -i reuters-vectors -o reuters-kmeans-clusters \
-m org.apache.mahout.common.distance.TanimotoDistanceMeasure  \
-c reuters-canopy-centroids -d 0.1 –w
```

After the clustering is done, use the `ClusterDumper` to inspect the clusters. Some of them are listed below:

```
Id: 21523:name:
     Top Terms:
tones, wheat, grain, said, usda, corn, us, sugar, export, agriculture
Id: 21409:name:
     Top Terms:
stock, share, shares, shareholders, dividend, said, its, common, board, company
Id: 21155:name:
     Top Terms:
oil, effective, crude, raises, prices, barrel, price, cts, said, dlrs
Id: 19658:name:
     Top Terms:
drug, said, aids, inc, company, its, patent, test, products, food
Id: 21323:name:
     Top Terms:
             7-apr-1987, 11, 10, 12, 07, 09, 15, 16, 02, 17
```

Note the last cluster shown above. While the others seem to be great topic groups, the last one looks meaningless. However, the clustering would have grouped these as occur together. Another issue is that words like "its", "said" etc that occur in these clusters are also useless from a language standpoint. The algorithm simply doesn't know that. Therefore, any clustering algorithm can generate good clustering provided the highest weighted features of the vector represent good features of the document.

In sections 8.3 and 8.4, we saw how Tf-Idf and normalization gave higher weights to the important features and lower weight to the stop words, but from time to time such spurious clusters do surface. A quick and effective way to solve such a problem is to remove these words from ever occurring as features in the document `Vector`. In the case study in section 9.1.5, we will show how we fix this using a custom Lucene `Analyzer` class.

Canopy clustering is a good approximate clustering technique. However, it suffers from memory problem. If the distance thresholds are close, too many canopies get generated. This increases RAM usage in the Mapper and hence might hit out of memory error while running on a large dataset with a bad set of thresholds. K-Means++ solves the problem in a more elegant manner as we see next.

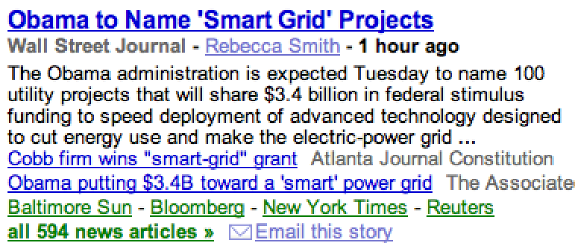
9.1.6 K-Means++: Clustering reloaded

TODO

We are going to put all the learning we did till now to create a clustering module for a news website. We choose a news website as it best represents a dynamic system where content needs to be organized and with very good precision. Clustering can help solve the issues related to such content systems.

9.1.7 Case study: Clustering news articles using K-Means

In this case study we are going to assume that we are in charge of a fictional news aggregation website called "AllMyNews.com". A person who comes to the website tries to search using keywords to find the content they are looking for. If they see an interesting article, they have to use the words in the article to search for related articles, or they drill down to the news category and explore news articles there. Usually we rely on human editors to find related items and help categorize and cross-link the whole website. If articles are coming in at tens of thousands per day, human intervention might prove too expensive. Enter clustering. Using clustering, we may be able to find related stories automatically and thus be able to give the user a better browsing experience.



Obama to Name 'Smart Grid' Projects
Wall Street Journal - Rebecca Smith - 1 hour ago
The Obama administration is expected Tuesday to name 100 utility projects that will share $3.4 billion in federal stimulus funding to speed deployment of advanced technology designed to cut energy use and make the electric-power grid ...
Cobb firm wins "smart-grid" grant Atlanta Journal Constitution
Obama putting $3.4B toward a 'smart' power grid The Associate
Baltimore Sun - Bloomberg - New York Times - Reuters
all 594 news articles » Email this story

Figure 9.5 An example of related-articles functionality taken from the Google News website. The links to similar stories within the cluster are shown at the bottom in bold. The top related articles are shown as links above that.

To minimize the human intervention we are going to use K-Means clustering to implement such a feature. Look at figure 9.5 for an example of what the feature would look like in practice. For news story on the website, we will show to the user, the list of all related news articles.

Lets see how clustering solves this problem. For any given article, we can store the cluster in which the articles reside. When a user requests for articles related to the one he is reading, we will pick out all the articles in the cluster and sort them based on the distance to the given article and present it to the user. Though this is a great starting design for a news-clustering system. It just doesn't solve all issues completely. Lets list down some real life problems one might face in such a dynamics:

- There are articles coming in every minute and the website needs to refresh its clusters and indexes.
- There might be multiple stories breaking out at the same time so we would require separate clusters for them, thus we need to add more centroids incrementally every time this happens.
- The quality of the text content is questionable as there are multiple sources feeding in the data. So, we need to have mechanisms to cleanup the content when doing feature selection.

We start with an efficient K-Means clustering implementation to cluster news articles offline. Here the word "offline" means we will write the documents into `SequenceFiles` and start the clustering as a backend process. In the coming chapters, we will modify this case study add various advanced techniques using Mahout and help solve issues related to speed and quality.

Finally at the end of the clustering section, we will show a working, tuned and scalable clustering framework for a live website like "AllMyNews.com" that can be adapted for different applications. We will not go into details of how storage of news data is done. We will assume for simplicity document storage and retrieval blocks can't be replaced easily by database read/write code. The listing 9.4 shows the code that clusters news articles from `SequenceFiles` and listing 9.5 shows a custom Lucene `Analyzer` class, which prunes away non-alphabetic features from the data.

9.4 News clustering using Canopy generation and K-Means Clustering

```
public class NewsKMeansClustering {
  public static void main(String args[]) throws Exception {
```

```
int minSupport = 2;
int minDf = 5;
int maxDFPercent = 80;
int maxNGramSize = 2;
int minLLRValue = 50;
int reduceTasks = 1;
int chunkSize = 200;
int norm = 2;
boolean sequentialAccessOutput = true;

String inputDir = "inputDir";
File inputDirFile = new File(inputDir);
if (!inputDirFile.exists()) {
  inputDirFile.mkdir();
}
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);

SequenceFile.Writer writer = new SequenceFile.Writer(fs, conf,
    new Path(inputDir, "documents.seq"), Text.class, Text.class);

for (Document d : Database) {                       #1
  writer.append(new Text(d.getID()), new Text(d.contents()));
}

writer.close();
String outputDir = "newsClusters";
HadoopUtil.overwriteOutput(outputDir);

String tokenizedPath = outputDir +
    DocumentProcessor.TOKENIZED_DOCUMENT_OUTPUT_FOLDER;
MyAnalyzer analyzer = new MyAnalyzer();             #2
DocumentProcessor.tokenizeDocuments(inputDir, analyzer.getClass()
    .asSubclass(Analyzer.class), tokenizedPath);    #3

DictionaryVectorizer.createTermFrequencyVectors(tokenizedPath,
  outputDir, minSupport, maxNGramSize, minLLRValue, reduceTasks,
  chunkSize, sequentialAccessOutput);
TFIDFConverter.processTfIdf(
  outputDir + DictionaryVectorizer.DOCUMENT_VECTOR_OUTPUT_FOLDER,
  outputDir + TFIDFConverter.TFIDF_OUTPUT_FOLDER, chunkSize, minDf,
  maxDFPercent, norm, sequentialAccessOutput);       #4

String vectorsFolder = outputDir + TFIDFConverter.TFIDF_OUTPUT_FOLDER
                       + "/vectors";
String canopyCentroids = outputDir + "/canopy-centroids";
String clusterOutput = outputDir + "/clusters";

CanopyDriver.runJob(vectorsFolder, canopyCentroids,  #5
  ManhattanDistanceMeasure.class.getName(), 2000, 1800);
KMeansDriver.runJob(vectorsFolder, canopyCentroids, clusterOutput,
  TanimotoDistanceMeasure.class.getName(), 0.01, 10, 1);   #6



SequenceFile.Reader reader = new SequenceFile.Reader(fs, new Path(
  clusterOutput + "/points/part-00000"), conf);

Text key = new Text();
```

```
      Text value = new Text();
      while (reader.next(key, value)) {                          #7
        System.out.println(key.toString() + " belongs to cluster "
                           + value.toString());
        // Write code here to save the cluster mapping to the database
      }
      reader.close();
    }
  }
```

#1 Replace with the code that fetches data from a DB/File
#2 Add a custom Lucene Analyzer - MyAnalyzer
#3 Tokenize document for the DictionaryVectorizer
#4 Calculate Tf-Idf vectors from the tokenized documents using bigrams
#5 Run canopy centroid generation job to get cluster centroids
#6 Run K-Means algorithm to cluster the documents
#7 Read the mapping table of Vector to Cluster and save them

9.5 A custom Lucene Analyzer that filters non alphabetic tokens

```
  public class MyAnalyzer extends Analyzer {

    private final CharArraySet stopSet;
    private final Pattern alphabets = Pattern.compile("[a-z]+");

    public MyAnalyzer() {
      stopSet = (CharArraySet) StopFilter.makeStopSet(StopAnalyzer.ENGLISH_STOP_WORDS);
    }

    public MyAnalyzer(CharArraySet stopSet) {
      this.stopSet = stopSet;
    }

    @Override
    public TokenStream tokenStream(String fieldName, Reader reader) {
      TokenStream result = new StandardTokenizer(Version.LUCENE_CURRENT, reader);
      result = new StandardFilter(result);
      result = new LowerCaseFilter(result);
      result = new StopFilter(true, result, stopSet);       #1

      TermAttribute termAtt = (TermAttribute) result.addAttribute(TermAttribute.class);
      StringBuilder buf = new StringBuilder();
      try {
        while (result.incrementToken()) {
          if (termAtt.termLength() < 3) continue;          #2
          String word = new String(termAtt.termBuffer(), 0, termAtt.termLength());
          Matcher m = alphabets.matcher(word);

          if (m.matches()) {                              #3
            buf.append(word).append(" ");
          }
        }
      } catch (IOException e) {
        e.printStackTrace();
      }
```

```
        return new WhiteSpaceTokenizer(new StringReader(buf.toString()));
      }
    }
```

#1 Use couple of Lucene filters
#2 Remove word having a length of three characters or less
#3 Consider only words made of alphabets

The NewsKMeansClustering example is straightforward. The documents are fetched and written to the input directory. From these, we create vectors from unigrams and bigrams that contain only alphabets. Using the generated Vectors as input, we run the Canopy centroid generation job to create the seed set of centroids for K-Means clustering algorithm. Finally, at the end of K-Means clustering, we read the output and save it to the database. The next section looks at the other algorithms in Mahout that takes us further than K-Means.

9.2 Beyond K-Means: An overview of clustering techniques

K-Means produces very rigid clustering, for example a news article, which talks about influence of politics in biotechnology, could be clustered either along with the politics document or with the biotechnology document but not with both. Since we are trying to tune the related articles feature, we might also need the overlapping or fuzzy information. We also might need to model the point distribution of our data. This is not something K-Means was designed to do. K-Means is just one type of clustering. There are many other clustering algorithms is designed on different principle, which we will see next.

9.2.1 Different kinds of clustering problems

Recall that clustering is simply a process of putting things into groups. To do more than just this simple grouping, we need to first understand the different kinds of problems in clustering. These problems and their solutions fall mainly into four categories as follows:

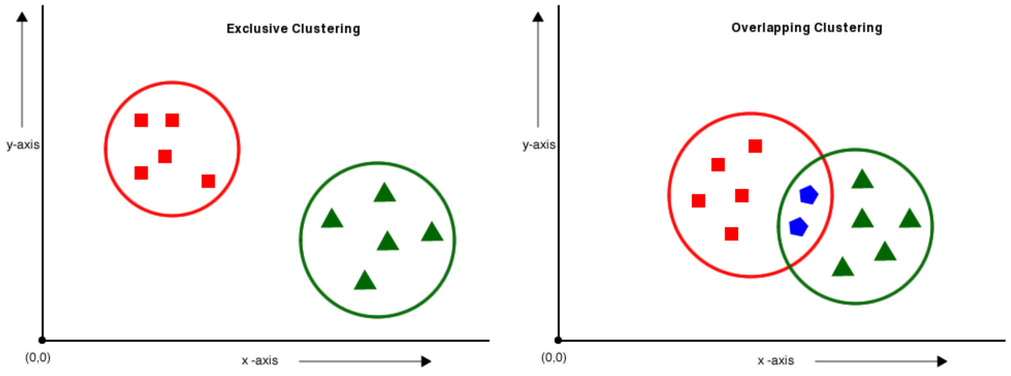


Figure 9.6 Exclusive clustering v/s Overlapping clustering with two centers. In the former, squares and triangles have

their own cluster and each one belong only to one cluster. While in overlapping clustering, some shapes like pentagon can belong to both the clusters with some probability so they are part of both clusters, instead of having a cluster of their own.

EXCLUSIVE CLUSTERING

In exclusive clustering, an item belongs exclusively to one cluster, not several. Recall the librarian example in Chapter 7, where we associated a book like Harry Potter to the cluster containing books of the fiction genre. There, Harry Potter exclusively belonged to the fiction cluster. K-Means as we saw does this exclusive clustering. So if the clustering problem demands this behavior, K-Means will usually do the trick.

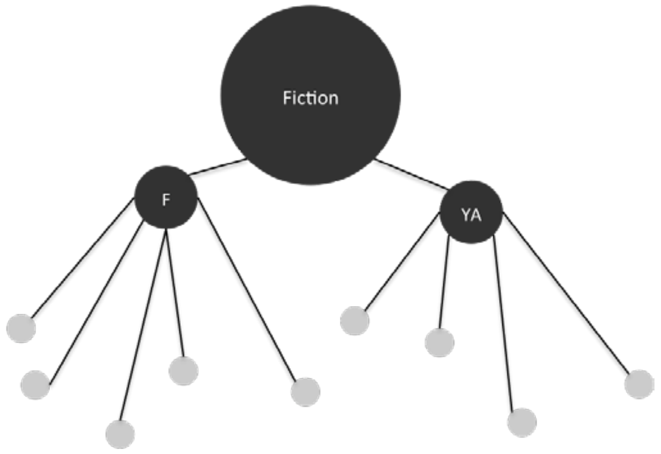
OVERLAPPING CLUSTERING

What If we wanted to do non-exclusive clustering: that is, put Harry Potter not only in fiction but also in a “young adult” cluster as well as under “fantasy”. An overlapping clustering algorithm like Fuzzy K-Means achieves this easily. Moreover, Fuzzy K-Means also tells the degree with which an object is associated with a cluster. So Harry potter might be inclined more towards the “fantasy” cluster than the “young adult” cluster. The difference between exclusive and overlapping clustering is illustrated in figure 9.6.

HIERARCHICAL CLUSTERING

Now, assume a situation where we have two clusters of books, one on “fantasy” and the other on “space travel”. Harry Potter is in the cluster of fantasy books. However, these two clusters, “space travel” and “fantasy”, could be visualized as sub-clusters of “fiction”. Hence, we can construct the “fiction” cluster by merging these and other similar clusters. “fiction” and “fantasy” has a parent child hierarchy, and hence the name Hierarchical clustering.

Similarly, we could keep grouping clusters into bigger and bigger ones. At a certain point, the clusters would be so large and so generic that they’d be useless as groupings. Nevertheless, this is a useful method of clustering: merging small clusters until it becomes undesirable to do so. Methods that uncover such a systematic tree-like hierarchy from a given data collection are called Hierarchical-clustering algorithms.

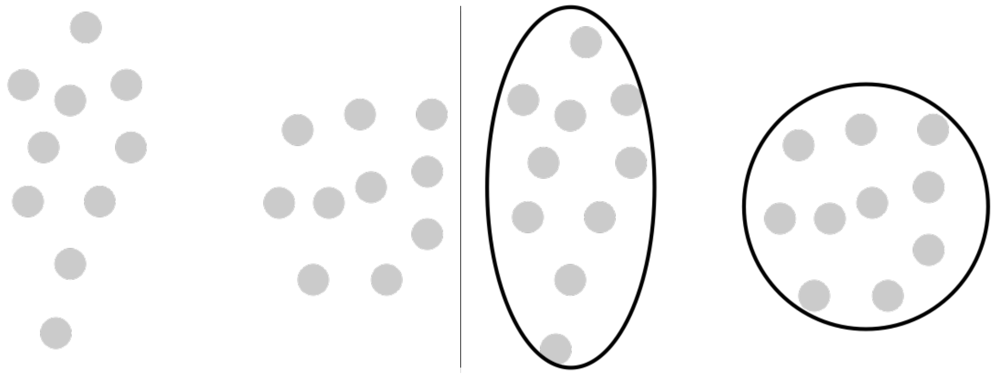


9.7 Hierarchical clustering. A bigger cluster groups two or more smaller clusters in the form of a tree like hierarchy.

Recall the librarian example in chapter 7. We did a crude form of hierarchical clustering when we simply stacked books based the similarity we felt when we read them.

PROBABILISTIC CLUSTERING

A probabilistic model is usually a distribution of a set of points in the n-dimensional plane, and usually they have a characteristic shape. There are certain probabilistic models that fit known data patterns. Therefore, such clustering algorithms try to fit a probabilistic model over a dataset and try to adjust the model parameter to correctly fit the data. Mostly such correct fit never happens. Instead, these algorithms give a percentage match or a probability value, which tells how much a fit the model is to the cluster.



9.8 A simplified view of probabilistic clustering. The initial set of points (left). On the right: the first set of points matches an elongated elliptical model where as the second one is more symmetric.

To explain how this fitting happens, let's look at a 2-d example in Figure 9.8. Say, we somehow know that all points in a plane are distributed in various regions with an elliptical shape. However, we don't know the center and radius or axes of these regions. We will choose an elliptical model and try to fit it to the data. We will move, stretch or contract each ellipse to best fit a region. We will do this for all the regions. This is called model-based clustering. A typical example of this type is the Dirichlet Process clustering algorithm, which does fitting based on a model provided by the user. We will see this clustering algorithm in action in section 9.4 of this chapter. Before we get there, we need to understand how different clustering algorithms are grouped based on their strategy.

9.2.1 Different clustering approaches

Different algorithms in clustering take different approaches. We can look at these approaches in a categorical manner as follows:

- Fixed number of centers
- Bottom-up approach
- Top-down approach

There are many other clustering algorithms that have other unique ways of clustering. You may never encounter them in Mahout, as at the time of writing they are not scalable on large datasets. Instead, we will explore the different algorithms based on the above three approaches, next.

FIXED NUMBER OF CENTERS

These methods fix the number of clusters ahead of time. The count of clusters is typically denoted by the letter k, which originated from the k of the K-Means algorithm. The idea is to start with k and to modify these k cluster centers to better fit the data. Once converged, the points in the dataset are assigned to the centroid closest to it.

Fuzzy K-Means is another example of an algorithm, which requires a fixed number of clusters. Unlike K-Means, which does exclusive clustering, Fuzzy K-Means does overlapping clustering.

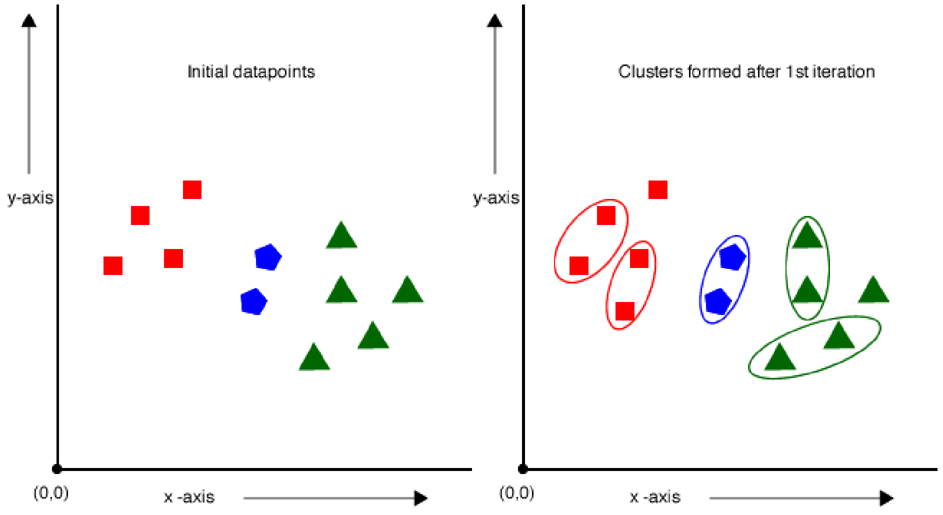


Figure 9.9 Bottom up clustering approach. After every iteration, the clusters are merged to produce larger and larger clusters till it is infeasible to merge based on the given distance measure.

BOTTOM-UP APPROACH: FROM POINTS TO CLUSTERS VIA GROUPING

When we have a set of points in n-dimensions, we can do two things. We can assume that all points belong to a single cluster and start dividing the cluster into smaller clusters, or we can assume that each of the data point begins in its own cluster and start grouping them iteratively. The former is called a top-down approach and the latter is called a bottom-up approach. The bottom-up clustering algorithms work as follows:

From a set of points in an n-dimensional space, the algorithm finds the pairs of points close to each other and merges them into one cluster. This merge is done only if the distance between them is below a certain threshold value. If not, those points are left alone. We repeat this process of merging the clusters using the distance measure till nothing can be merged anymore.

TOP-DOWN APPROACH: SPLITTING THE GIANT CLUSTER

We start with all points belonging to a single cluster i.e. a giant cluster. Then we divide this giant cluster into smaller clusters. This is known as a top-down approach. The aim here is to find the best possible way to split this giant cluster into two smaller clusters. These clusters are divided repeatedly until we get clusters, which are meaningful. This is based on some distance measure criterion.

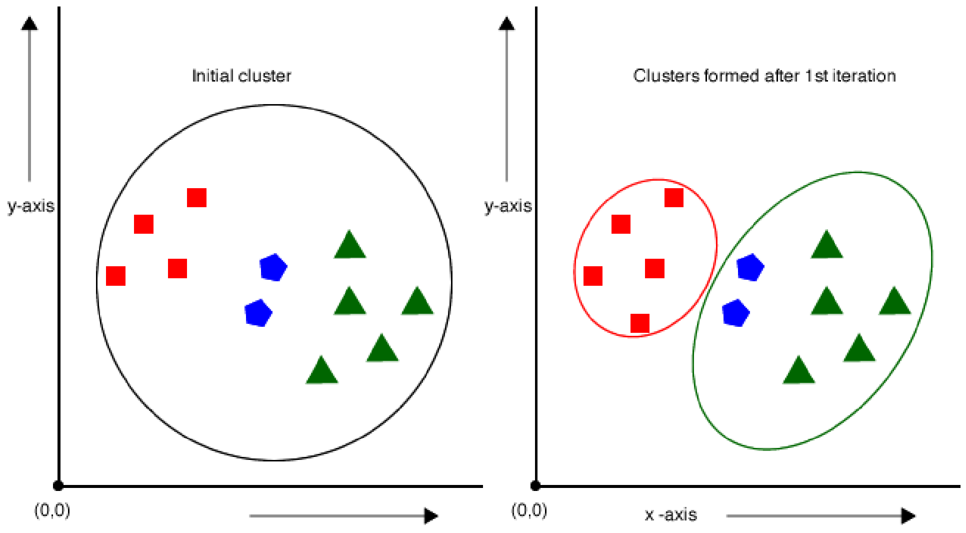


Figure 9.10 Top down clustering approach. During each iteration, the clusters are divided into two by finding the best splitting till we get the clusters we desire.

Though this is very straightforward, finding the best possible split for a set of n-dimensional points is not too easy. Moreover, most of these algorithms cannot be easily reduced into the map-reduce form and so Mahout doesn't have them now. An example of a top down algorithm is Spectral clustering. In Spectral clustering, the splitting is decided by finding the line/plane that cuts the data into two sets with a larger margin between the two sets.

The beauty of top down and bottom up approaches are that they don't require the user to input the cluster size. This means that in a dataset where we have no idea about the distribution of the points, both types of algorithms output clusters based solely on the similarity metric. This works quite well in many applications. These approaches are still being researched upon. Even though Mahout has no implementations of these methods, other specialized algorithms implemented in Mahout can run as Map/Reduce jobs without specifying the number of clusters as explained below.

The lack of hierarchical clustering algorithms in Map/Reduce is easily circumvented by the smart use of K-Means, Fuzzy K-Means, and Dirichlet clustering. To get the hierarchy, start with small number of clusters (k) and repeat clustering with increasing values of k. Alternately, we can start with large number of centroids and start clustering the cluster centroids with decreasing value of k. This mimics the hierarchical clustering behavior while making full use of the scalable nature of Mahout implementations.

The next section deals with the Fuzzy K-Means algorithm in detail. We will be using it to improve our related articles implementation for our news website allmynews.com.

9.3 Fuzzy K-Means clustering

As the name says, this algorithm does a fuzzy form of K-Means clustering. Instead of exclusive clustering in K-Means, Fuzzy K-Means tries to generate overlapping clusters from the dataset. In the academic community, it's also known by the name Fuzzy C-Means algorithm. We can think of it as an extension of K-Means. K-Means tries to find the hard clusters (a point belonging to one cluster) where as, Fuzzy K-Means discovers the soft clusters. In a soft cluster, any point can belong to more than one cluster with a certain affinity value towards each. This affinity is proportional to the distance of point to the centroid of the cluster. Like K-Means, Fuzzy K-Means works on those objects that can be represented in n-dimensional vector space and has a distance measure defined.

9.3.1 Running Fuzzy K-Means clustering

The algorithm above is available in `FuzzyKMeansClusterer` or the `FuzzyKMeansDriver` class. Like others, the former is an in-memory implementation and the latter Map/Reduce. We are going to use the same random point generator function we used earlier in order to create the points scattered in the 2-dimensional plane. In listing 9.6, we ran the in-memory version using the `FuzzyKMeansClusterer` with the following parameters:

- The input `Vector` data in the `List<Vector>` format.
- The `DistanceMeasure` is `EuclideanDistanceMeasure`.
- The threshold of convergence is `0.01`
- The number of clusters k is `3`
- The fuzziness parameter m is `3.` This parameter will be explained later in section 9.3.2

9.6 In-memory clustering example of Fuzzy K-Means clustering

```
public static void FuzzyKMeansExample() {
  List<Vector> sampleData = new ArrayList<Vector>();

  generateSamples(sampleData, 400, 1, 1, 3);                  #1
  generateSamples(sampleData, 300, 1, 0, 0.5);
  generateSamples(sampleData, 300, 0, 2, 0.1);

  List<Vector> randomPoints = RandomSeedGenerator.chooseRandomPoints(
    points, k);
     List<SoftCluster> clusters = new ArrayList<SoftCluster>();

     int clusterId = 0;
     for (Vector v : randomPoints) {
       clusters.add(new SoftCluster(v, clusterId++));
     }

     List<List<SoftCluster>> finalClusters = FuzzyKMeansClusterer
.clusterPoints(points, clusters, new EuclideanDistanceMeasure(),
    0.01, 3, 10); #2
```

```
        for(SoftCluster cluster : finalClusters.get(finalClusters.size() - 1)) {
       System.out.println("Fuzzy Cluster id: " + cluster.getId()
           + " center: " + cluster.getCenter().asFormatString());   #3
        }

      }
```

#1 Generate 3 sets of points using different parameters
#2 Run FuzzyKMeansClusterer
#3 Read the center of the fuzzy-clusters and print it.

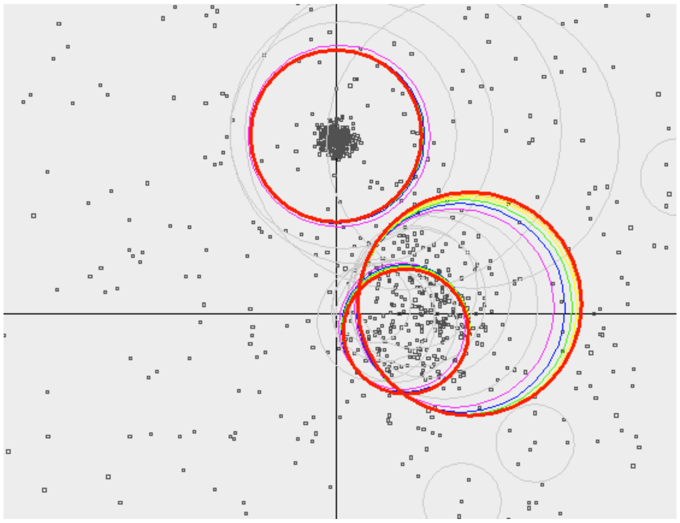


Figure 9.11 Fuzzy K-Means clustering. The clusters look like they are overlapping each other and the degree of overlap is decided by the fuzziness parameter.

The `DisplayFuzzyKMeans` class in the “examples” folder of the Mahout code is a good tool to visualize this algorithm on a 2-dimensional plane. `DisplayFuzzyKMeans` runs as a Java swing application and produces an output as given in the figure 9.11.

MAP/REDUCE IMPLEMENTATION OF FUZZY K-MEANS

Before running the Map/Reduce implementation lets create a checklist for running Fuzzy K-Means clustering against the Reuters dataset like we did for K-Means. We have:

- The dataset in the `Vector` format.
- The `RandomSeedGenerator` to seed the initial k clusters.
- The distance measure is `SquaredEuclideanDistanceMeasure`.
- A large value of `convergenceThreshold –d 1.0`, as we are using the squared value of the distance measure.
- The `maxIterations` is the default value of `–x 10`

- The coefficient of normalization or the fuzziness factor, a value greater than `1.0`, which will be explained in the section 9.3.2, `–m`

To run the Fuzzy K-Means clustering over the input data, use the Mahout launcher using the “fkmeans” program name as follows:

```
$bin/mahout fkmeans
-i reuters-vectors -c reuters-fkmeans-centroids
-o reuters-fkmeans-clusters -d 1.0 -k 21 -m 2 -w
-dm org.apache.mahout.common.distance.SquaredEuclideanDistanceMeasure
```

Like K-Means, `FuzzyKMeansDriver` will automatically run the `RandomSeedGenerator` if the number of clusters (k) flag is set. Once the random centroids are generated, Fuzzy K-Means clustering will use it as the input set of k centroids. The algorithm runs multiple iterations over the dataset until the centroids converges, each time creating the output in the folder `cluster-*`. Finally, it runs another job, which generates the probabilities of a point belonging to a particular cluster based on the distance measure and the fuzziness parameter (`m`).

Before we get into details of the fuzziness parameter, it’s a good idea to inspect the clusters using the `ClusterDumper` tool. `ClusterDumper` shows the top words of the cluster as per the centroid. To get the actual mapping of points to the clusters, we need to read the SequenceFiles in the `points/` folder. Each entry in the sequence file has a key, which is the identifier of the vector, and a value, which is the list of cluster centroids with an associated numerical value, which tell us how well the point, belongs to that particular centroid.

9.3.2 How fuzzy is too fuzzy

Fuzzy K-Means has a parameter `m` called the fuzziness factor. Like the K-Means Fuzzy K-Means loops over the dataset and instead of assigning vectors to the nearest centroids, it calculates the degree of association of the point to each of the clusters. Say for a vector (V) if d1, d2, … , dk are the distances towards each of the k cluster centroids. The degree of association (`u1`) of vector (V) to the first cluster (C1) is calculated as

```
u1  = 1/((d1/d1)^(2/(m-1))+ (d1/d2)^(2/(m-1)) + … + (d1/dk)^(2/(m-1)))
```

Similarly, we can calculate the degree of belonging to other clusters by replacing d1 in the numerators of the denominator expression by d2, d3 and so on. It’s clear from the expression that m should be greater than 1, or else the denominator of the fraction becomes zero and things break down.

If we choose a value of `m` as 2, we will see that all degrees of association for any point sums up to one. If on the other hand, m comes very close to 1, like 1.000001, more importance would be given to that centroid closest to the vector. So, the Fuzzy K-Means algorithm starts behaving more like K-Means algorithm, as `m` gets closer to 1. If `m` increases, the fuzziness of the algorithm increases and we begin to see more and more overlap.

The Fuzzy-K-Means algorithm is also found to converge better and faster than a standard K-Means algorithm.

9.3.3 Case study: clustering news articles using Fuzzy K-Means

The related articles functionality will certainly be richer with knowledge of partial overlap. The partial score will help rank the related articles by their relatedness to the cluster. In listing 9.7, we will modify our case study example to use Fuzzy K-Means algorithm and retrieve the fuzzy cluster membership information.

9.7 News clustering using Fuzzy K-Means clustering

```
public class NewsFuzzyKMeansClustering {
  public static void main(String args[]) throws Exception {
    …
    …
    …
    float fuzzificationFactor = 2.0f;
    String vectorsFolder = outputDir + TFIDFConverter.TFIDF_OUTPUT_FOLDER
                           + "/vectors";
    String canopyCentroids = outputDir + "/canopy-centroids";
    String clusterOutput = outputDir + "/clusters";

    CanopyDriver.runJob(vectorsFolder, canopyCentroids,  #1
      ManhattanDistanceMeasure.class.getName(), 2000, 1800);
    FuzzyMeansDriver.runJob(vectorsFolder, canopyCentroids, clusterOutput,
      TanimotoDistanceMeasure.class.getName(), 0.01, 10, 1,
 fuzzificationFactor);                            #2



    SequenceFile.Reader reader = new SequenceFile.Reader(fs, new Path(
      clusterOutput + "/points/part-00000"), conf);

    Text key = new Text();
    Text value = new Text();
    while (reader.next(key, value)) {                    #3
      for (int i = 0; i < value.getClusters().length; i++) {
        System.out.println(key.toString() + " belongs to cluster " #4
          + value.getClusters()[i].getIdentifier() + " with probability "
          + value.getProbabilities()[i]);
      }
      // Write code here to save the cluster mapping to our database
    }
    reader.close();
  }
}
```

#1 Run canopy generation job to get cluster centroids
#2 Run Fuzzy K-Means to cluster the documents
#3 Read the mapping table of vector to Fuzzy K-Means output
#4 Print the clusters and the probabilities of association

They Fuzzy K-Means algorithm gave us a way to do a much needed refinement for our related articles code. Now we know, by what degree a point belongs to a cluster. Using this information we can find top clusters the point belongs to and use the degree to find the weighted score of articles. This way we negate the strictness of overlapping clustering and give better related-articles for documents lying on the boundaries of a cluster.

Next, let’s learn another clustering algorithm in Mahout. Unlike the ones we have seen until now, this one produces a lot of information about the cluster and how points are distributed within it. It is called Dirichlet process clustering.

9.4 Model based Clustering

The complexities of clustering algorithms have increased progressively in this chapter. We started with K-Means, a very fast clustering algorithm. Then, we saw how we captured partial clustering membership using Fuzzy K-Means. We also learned to optimize the clustering using centroid generation algorithms like Canopy clustering and K-Means++. What more do we want to know about these clusters? How do we understand the structures within the data better? To do this, we may need a method that is completely different from the algorithms we described above. Model based clustering methods help alleviate these problems. Before learning what model based clustering is, we need to see some of the issues faced by K-Means and other related algorithms.

9.4.1 Fallacies of K-Means

Say we wanted to cluster our dataset into some k clusters. We have learned how we can run K-Means and get the clusters quickly. K-Means works well because it can always divide clusters easily using a linear distance. What if we knew that the clusters are based on a normal distribution and are mixed together and overlapping each other? Can we use this information to improve clustering using K-Means? Here, we might be better off with a Fuzzy K-Means clustering.

What if the clusters themselves are not in a normal distribution? What if the clusters are having an ovoid shape? Neither K-Means nor Fuzzy K-Means knows how to use this information to improve the clustering. Before we answer these questions, let’s first see an example where K-Means clustering fails to describe a simple distribution of data.

ASYMMETRICAL NORMAL DISTRIBUTION

We are going to run K-Means clustering using points generated from an asymmetrical normal distribution. What it means is, instead of the points being scattered in 2-dimensions around a point in a circular area, we are going to make the point-generator generate clusters of points having different standard deviations in different directions. This creates an ellipsoidal area where the points are concentrated. We will now run the in-memory K-Means implementation over this data.

Figure 9.12 shows the ellipsoidal or asymmetric distribution of points and the clusters generated by K-Means. It is clear that K-Means is not powerful enough to figure out the distribution of these points.

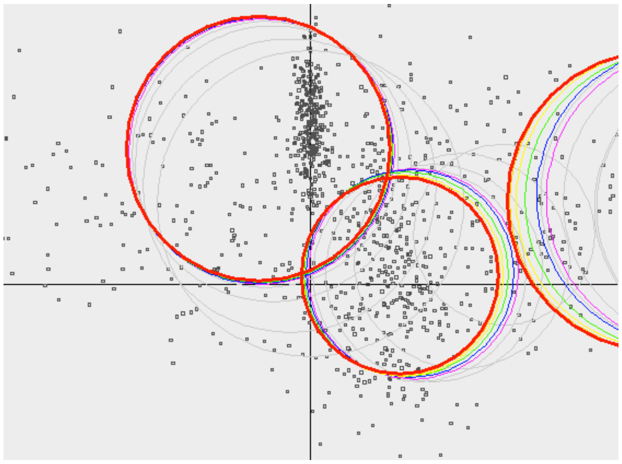


Figure 9.12 Running K-Means clustering over asymmetric normal distribution of points. The points are scattered in an oval -shaped area instead of a circular one. Clearly K-Means is not a perfect fit for the data and these clusters don't make any.

Another issue with K-Means is its requirement of the k number of seed centroids. Most of the time, we end up over-estimating this number. Finding the optimal value of k is not an easy task unless there is a clear idea about the data, which happens very rarely. Even by doing canopy generation or K-Means++, we need to tune the distance measure to make these algorithms improve the estimate of k. What if there was a better way to find the number of clusters? That's something where model-based clustering proves to be useful.

ISSUES WITH CLUSTERING REAL WORLD DATA

Think of the following real-world clustering problem where we want to cluster a population of people based on their movie preferences to find like-minded people. We can estimate the number of clusters in such a population by counting different genre of movies.

Some of the clusters that we find here are: people who like action movies, people who like romantic movies, people who like comedy and so on. This is not a very good estimation as there are tons of exceptions. For example: there are clusters of people who like only gangster movies, not other action movies. They form a sub cluster under the action cluster. With such a complex mixing of clusters, we never get the information of a small cluster since a bigger cluster always subsumes it. The only way to improve this situation is to somehow understand that movie preferences of a population of people are hierarchical in nature.

If we had known this earlier, we would have used a hierarchical clustering method to cluster the people better. But those methods cannot capture the overlap. So all the clustering algorithms we have seen before does not capture the hierarchy and the overlap at the same time. How can we use a method to uncover all these information? That’s also something that is tackled by the model-based clustering.

9.4.3 Dirichlet processes clustering

Mahout has a model based clustering algorithm known by the name Dirichlet processes clustering. The word “Dirichlet” refers to a family of probability distributions defined by a German mathematician Johann Peter Gustav Lejeune Dirichlet. The Dirichlet process clustering performs something known as mixture modeling using calculations based on the Dirichlet distribution. The whole process might sound very complicated without a deeper understanding of Dirichlet distributions, but the idea is very simple.

Say, we knew that our data-points are concentrated in an area like a circle and well distributed within it and we have a model that explains this behavior. We test whether our data fits the model by reading through our vectors and calculating the probability of the model being a fit to the data. It is like saying that the region of concentration of points looks more like a circular model, with some greater degree of confidence. It could also say that the region looks less like a triangle, another model, due to lesser probability of fit of the data with the triangle. If we find a fit, we know the structure of our data. Note, that circles and triangles are used here as a tool for visualizing this algorithm. They are not be mistaken for a probabilistic model on which this algorithm works.

Dirichlet process clustering is implemented as a Bayesian clustering algorithm in Mahout. What that means is that the algorithm doesn’t just want to give one explanation of the data, rather it wants to give lots of explanations. This is like saying, the region A is like a circle, the region B is like a triangle, together region A and region B is like a polygon and so on. In reality, these regions are statistical distributions like the normal distribution that was seen earlier in the chapter.

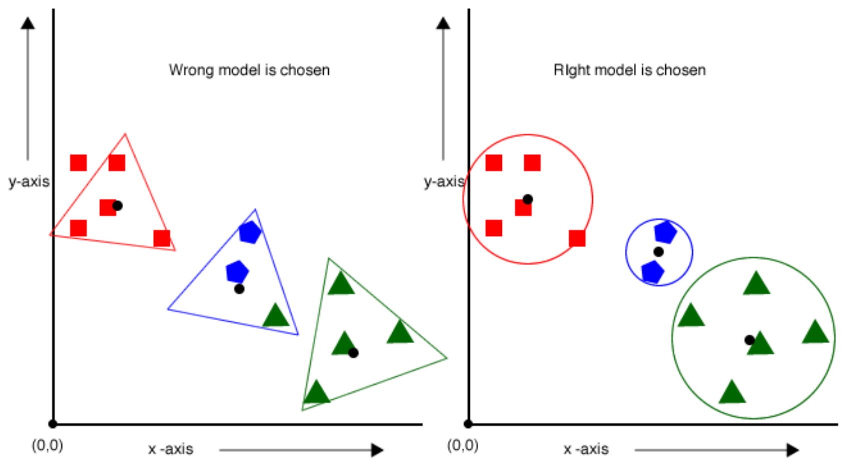


Figure 9.13 Dirichlet clustering. The models are made to fit the given dataset as best as it can to describe it. The right model will fit the data better and tells the number of clusters in the dataset which correlates with the model.

We will come to various model distributions a little later in this section, but a full discourse on them are out of scope of this book. Next, Lets understand how Dirichlet implementation in Mahout works

UNDERSTANDING DIRICHLET PROCESS CLUSTERING ALGORITHM

The Dirichlet process clustering starts with a dataset of points and a `ModelDistribution`. Think of `ModelDistribution` as a class that generates different models. We create an empty model and try to assign points to it. When this happens, the model grows or shrinks its parameters in a crude manner to try and fit the data. One it does this for all points, it re-estimates the parameters of the model precisely using all the points and the partial probability of the point belonging to the model.

At the end of each pass, we get some number of samples that contains the probabilities, models and assignment of points to models. These samples could be regarded as a cluster and they give us information about the models and its parameters. They also give us information about the shape and size of the model. Moreover, by examining the number of models in each sample that actually has some points assigned to it, we can get information about how many models (clusters) our the data supports. Also, by examining how often two points are assigned to the same model, we can get an approximate measure of how these points are explained by the same model. Such soft-membership information is a side product of using model-based clustering. Dirichlet process clustering is able to capture the partial probabilities of points towards various models.

9.4.4 Running a model based clustering example

The Dirichlet process based clustering is implemented in the `DirichletClusterer` class as in-memory and in `DirichletDriver` as a Map/Reduce job. We are going to use the `generateSamples` function we saw earlier in this chapter to create our vectors in a random fashion. Note that the Dirichlet clustering implementation is generic enough to put in any type of distribution and any data type. The `Model` implementations in Mahout use the `VectorWritable` type; hence, we will be using that as the default type in our clustering code. We are going to run Dirichlet process clustering using the following parameters:

- The input `Vector` data in the `List<VectorWritable>` format.
- The `NormalModelDistribution` as the model distribution we are trying to fit our data on.
- The alpha value of the Dirichlet distribution `1.0`
- The number of models to start with `numModels` is `10`
- The `thin` and `burn` intervals as `2` and `2`.

These points will be scattered around a specified center point like the normal distribution. The code snippet is shown below in listing 9.8.

9.8 Dirichlet clustering using normal distribution

```
List<VectorWritable> sampleData = new ArrayList<VectorWritable>();

generateSamples(sampleData, 400, 1, 1, 3);                  #1
generateSamples(sampleData, 300, 1, 0, 0.5);
generateSamples(sampleData, 300, 0, 2, 0.1);

DirichletClusterer<VectorWritable> dc =
  new DirichletClusterer<VectorWritable>(
    sampleData,
       new NormalModelDistribution(
         new VectorWritable(new DenseVector(2))),
       1.0, 10, 2, 2);
List<Model<VectorWritable>[]> result = dc.cluster(20);    #2
```

#1 Generate 3 sets of points each with different parameters
#2 Run Dirichlet Clusterer using the NormalModelDistribution

In the above example, we generated some sample points using a normal distribution and tried to fit the normal model distribution over our data. The parameters of the algorithm decide the speed and quality of convergence.

Here, alpha is a smoothing parameter. It allows a smooth transition of the models before and after the re-sampling happens. A higher value makes the transition slower and so clustering would try and over-fit the models. Lower value causes the clustering to merge models more quickly and hence tries to under-fit the model.

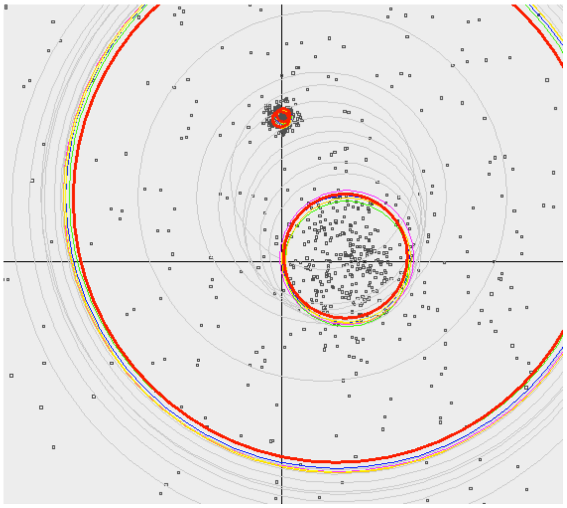


Figure 9.14 Dirichlet clustering with Normal Distribution using the DisplayNDirichlet class in Mahout examples folder

The `thin` and the `burn` intervals are used to decrease the memory usage of the clustering. The `burn` parameter decides the number of iterations to complete before saving the first set of models for the dataset. The `thin` parameter decides the number of iterations to skip between saving such a `Model` configuration.

The motivation of having these parameters is that we generally do many iterations to reach convergence and the initial states are not worth exploring. During the initial stages, the counts of Models are extremely high and they provide no real value for us. So, we skip (`thin/burn`) them to save memory. The final state achieved using the `DisplayNDirichlet` clustering example is given in

Figure 9.14. It's found in the examples folder of Mahout along with examples to plenty other model distributions and their clustering.

The kind of clusters we got here is different from the output of K-Means clustering we got in Section 9.2. Dirichlet process clustering did something that K-Means could not do which was to actually identify the 3 clusters exactly the way we generated it. Any other algorithm would have just tried to cluster things into overlapping groups or hierarchical groups.

This is just the tip of the iceberg. To show the awesome power of Model based clustering, we are going to repeat this example based on something more difficult than a normal distribution.

ASYMMETRIC NORMAL DISTRIBUTION

Normal distribution is asymmetrical when the standard deviations of points along different dimensions are different. This gives it an ellipsoidal shape. When we ran K-Means clustering on this distribution in section 9.4.1, we saw how it broke down miserably. Now we will attempt to cluster the same set of points using Dirichlet clustering with another model distribution class, the `AsymmetricSampledNormalDistribution`. We will run Dirichlet process clustering using the asymmetric normal model on the set of 2-d points that has different standard deviation along the x and y directions. The output of the clustering is shown in Figure 9.15.

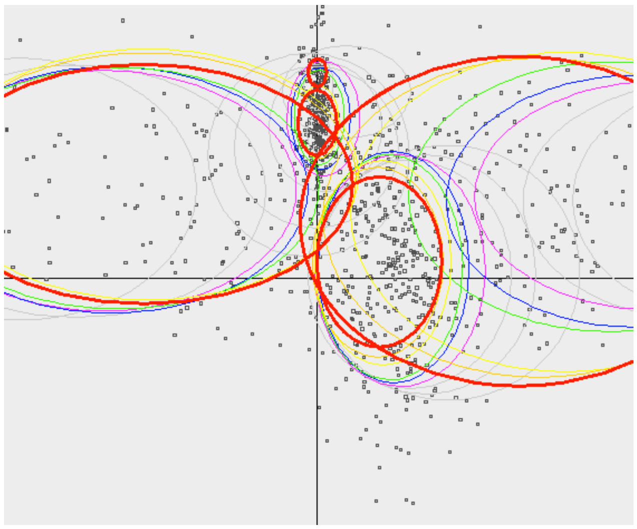


Figure 9.15 Dirichlet clustering with Asymmetrical Normal Distribution using the Display2dASNDirichlet class. The thick line denotes the final state and the thin lines are the states in previous iterations

Even though the number of clusters formed seems to have increased, model based clustering was able to find the asymmetric model and fit them to the data much better than any of the other algorithms. A better value of alpha might have improved this. The other model-distributions implemented in Mahout are `L1ModelDistribution` and `SampledNormalModelDisribution`. A discussion on them is too advanced for a book that gives an introduction clustering. Mahout documentation will explain their usage more. Next, we will try to launch the Map/Reduce version of Dirichlet clustering.

MAP/REDUCE VERSION OF DIRICHLET CLUSTERING

Like other implementations, in Mahout, Dirichlet clustering is also focused on scaling with huge datasets. The Map/Reduce version of Dirichlet clustering is implemented in the `DirichletDriver` class. The Dirichlet job could be run from the command line on the Reuters dataset. Lets get to our checklist for running a Dirichlet clustering Map/Reduce job:

- The Reuters dataset in the `Vector` format
- The model distribution class `–d` defaults to `NormalModelDistribution`
- The model distribution prototype `Vector` class. The class that becomes the type for all vectors created in the job `–p` defaults to `SequentialAccessSparseVector`
- The `alpha0` value for the distribution, `-m 1.0`
- The number of clusters to start the clustering with `–k 60`.
- The number of iterations to run the algorithm `–x 10`

Launch the algorithm over the dataset using the Mahout launcher with program name as “dirichlet” as follows:

```
bin/mahout dirichlet
-i examples/reuters-vectors/
-o reuters-dirichlet-clusters -k 60 -x 10 -m 1.0
-d org.apache.mahout.clustering.dirichlet.models.NormalModelDistribution
-p org.apache.mahout.math.SequentialAccessSparseVector
```

After each iteration of Dirichlet process clustering, the job writes the state in the output folder as subfolders with pattern `state-*`. Your can read then using SequenceFile reader and get the centroid and the standard deviation values for each model. Based on this we assign vectors to each cluster at the end of clustering.

Dirichlet process clustering is a powerful way of getting quality clusters using the knowledge of data distribution models. In Mahout, we have made the algorithm as a pluggable framework where different models can be created and tested on. As the models becomes more complex, there is a chance of things slowing down on huge datasets. At this point, we will have to fall back on the other clustering algorithms. However, seeing the output of Dirichlet process clustering, we can clearly take a decision on whether the algorithm we choose should be fuzzy or rigid, overlapping or hierarchical, or whether the distance measure is Manhattan or cosine and the threshold for convergence. The Dirichlet process clustering is more of a data understanding tool while being a great data clustering one.

9.5 Topic Modeling using Latent Dirichlet Allocation (LDA)

Till now, we have thought of documents as a set of term with some weights assigned to it. In real life, we think of news articles or any other text document as a set of topics. These topics are fuzzy in nature. On rare occasions, they are ambiguous. Most of the time when we read a text, we somehow associate it to a set of topics. If someone asks, “Hey Bob, what was that news article all about?” We will naturally say “Well, it talked about the US war against terrorism” instead of telling him what words were actually used in the document.

Think of a topic like “Dog”. There are plenty of texts on the topic “Dog” each describing various things. The frequently occurring words in such documents would be “dog”, “woof”, “puppy”, “bark”, “bow”, “chase”, “loyal”, “friend” to name a few. Some of these words “bow”, “bark” etc are ambiguous, as they are found in other topics like “arrow and bow” or “bark of a tree”. Still, we can say that all these words are in the topic “Dog”, some with more probability than others. Similarly, a topic like “Cat” has frequently occurring words like “cat”, “kitten”, “meow”, “purr”, and “fur-ball”.

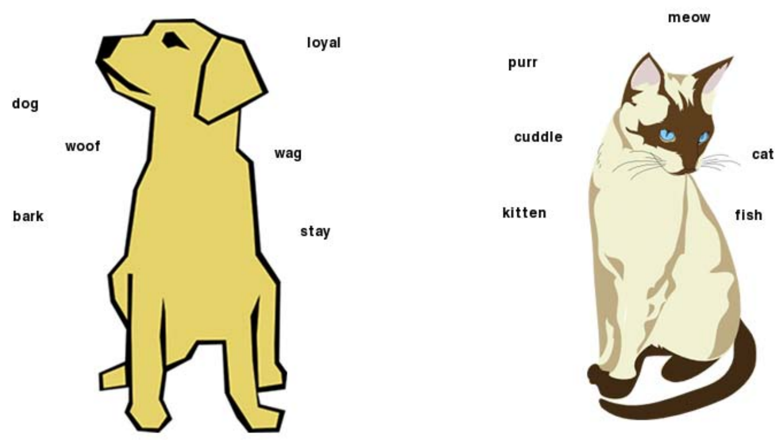


Figure 9.14 The topics “Dog” and “Cat” and the words that occur frequently in them.

If we were asked to find out these topics in a particular set of documents, our natural instinct now would be to use clustering. We would modify our clustering code to work with word vectors instead of document vectors we have been using until now. A word vector is nothing but a vector for each word where the features would be ids of the other words that occur along with it in the corpus and the weights would be the number of documents they occur together in.

Once we have such a vector, we could simply run one of the clustering algorithms, figure out the clusters of words, and call them as topics. Though this seems very simple, the amount of processing required to create the word vector is quite high. Still we can cluster words that occur together, call them a topic, and then calculate the probabilities of the word occurring to each topic.

LDA is more than just this clustering. If two words having the same meaning or form don’t occur together, then clustering will not be able to associate correlation between those two based on other instances. This is where LDA shines.

Now lets extend this problem. Say, we have a set of observations (documents and the words in it). Can we find out the hidden groups of features (topics) to explain these observations? LDA clusters features into hidden groups or topics in a very efficient manner.

9.5.1 Understanding LDA

Firstly, don't confuse LDA with the concept of Linear Discriminant Analysis. That is also known by the short name LDA. Linear Discriminant Analysis is a method used in classification where as Latent Dirichlet Allocation is a form of clustering.

LDA is a generative model like the Dirichlet process clustering. We start with a known model and try to explain the data by refining the parameters to fit the model to the data. LDA does this by assuming that the whole corpus has some k number of topics and each document is a talking about these k topics. Therefore, the document is considered a mixture of topics with different probabilities.

TIP

Machine learning algorithms come in two flavors - generative or discriminative. Algorithms like K-Means or hierarchical clustering which tries to split the data into k groups based on a distance metric are generally called discriminative. The example of the discriminative type is the SVM classifier, which we will learn about in the Classification part of this book. In Dirichlet clustering, the model tweaked to fit the data, and just using the parameters of the model, we can generate the data on which it fits. Hence, it is called a generative model.

How is it better than just clustering words? LDA is much powerful than standard clustering as it can jointly cluster words into "topics" and documents into mixtures of topics. Suppose there is a document about the Olympics, which has words like "gold", "medal", "run", "sprint" and another document about the 100m sprints in Asian games and has words like "winner", "gold", and "sprint". LDA can infer a model where the first document is considered as the mix of two topics one about sports and has words like "winner, "gold", "medal" and other about the 100m run and has words like "run", "sprint". LDA can find the probability with which each of the topics generate the respective documents. The topics themselves are a distribution of the probabilities of words. Therefore, the topic "sports" may have the word "run" with a lower probability than in the "100m sprint".

The LDA algorithm works similar to Dirichlet clustering. It starts with an empty topic model. It then reads all the documents in the Mapper in parallel and calculates the probability of each topic for each word in the document. Once this is done, the counts of these probabilities are sent to the reducer where they are summed and the whole model is normalized. We run this process repeatedly until the model starts explaining the documents better: that is, the sum of the (log) probabilities stop changing. The degree of change is decided by a convergence threshold parameter, similar to the threshold we found in K-Means clustering. Instead of the relative change in centroid, LDA estimates how well the model fits the data. If the likelihood value does not change above this threshold, we stop the iteration.

9.5.2 Tuning the parameters of LDA

Before running the LDA implementation in Mahout, we need understand the two parameters in LDA that gives a big impact on the runtime and quality. The first of these is the number of topics. Like, K-Means, we need to figure it out from the data that we have. Lower value for the number of topics usually gives us broader topics like science, sports, politics etc and engulfs words spanning multiple sub-topics. Large number of topics gives us focused or niche ones like "quantum physics", "laws of reflection".

A large number of topics also mean that the algorithm needs lengthier passes to estimate the word distribution for all the topics. This can be a serious slow down. A good rule of thumb is to choose a value that makes sense for a particular use case. Since, Mahout LDA is written as a Map/Reduce job, it can be run in large Hadoop clusters. We can speed up the algorithm by adding more servers if there is such need.

Second parameter is the number of words in the corpus, which is also the cardinality of the vectors. It determines the size of the matrices used in the LDA Mapper. The Mapper constructs a matrix of the size - number of topics multiplied by document length, which is number of words or features in the corpus. If we need to speed up LDA, apart from decreasing the number of topics, we also need to keep the features to a minimum. If we need to find the complete probability distribution of all the words over topics, we should leave this parameter alone. Instead, if we are interested only in finding the topic model containing only the keywords from a large corpus, we can simply prune away the high frequency words in the corpus while creating vectors.

We can lower the value of the max-document-frequency-percentage parameter (`--maxDFPercent`) in the dictionary-based vectorizer. A value of 70 removes all words that occur in more than 70% of the documents.

9.5.3 Case Study: Finding topics in News documents

We will run the Mahout LDA over the Reuters dataset. First, we run the dictionary vectorizer, create Tf-Idf vectors, and use them as input for the `LDADriver`. The high frequency words are pruned to speed up the calculation. In this example, we will model 10 topics from the Reuter vectors. The entry point `LDADriver` takes the following parameters:

- Input directory containing `Vectors`
- Output directory to write the LDA states after every iteration
- Number of topics to model `–k 10`
- Number of features in the corpus `–v`
- Topic smoothing parameter (uses the default value of `50/number of topics`)
- Limit on the maximum number of iterations (`--maxIter 20`)

The number of features in the corpus (`-v`) can be easily found by counting the number of entries in the dictionary file located in the vectorizer folder. We can use the `SequenceFileDumper` utility to find the number of dictionary entries as described in chapter 8. We will run the LDA algorithm from the command line as follows:

```
bin/mahout lda
-i reuters-vectors
-o reuters-lda-sparse
-k 10 -v 7000 --maxIter 20 –w
```

LDA will run 20 iterations or stop when the estimation converges. The state of the model after each iteration is written in the output directory as folders beginning with `state-`. Mahout has an output reader for LDA under the utils directory for reading the topic and word probabilities from the output state directory.

`org.apache.mahout.clustering.lda.LDAPrintTopics` is the main entry point for the utility. We can see the top 5 words of each topic model from the state folder of any iteration as follows:

```
bin/mahout org.apache.mahout.clustering.lda.LDAPrintTopics
-s reuters-lda-sparse/state-20/
-d reuters-vectors/dictionary-file-*
-dt sequencefile –w 5
```

The output for this example is shown in Table 9.1. Note that only 5 topics are shown from the 10.

Topic 0	Topic 1	Topic 2	Topic 3	Topic 4
wheat	south	loans	trading	said
7-apr-1987	said	president	exchange	inc
agriculture	oil	bank	market	its
export	production	chairman	dollar	corp
tonnes	energy	debt	he	company

Table 9.1 Top 5 words in selected topics from LDA topic modeling of Reuters news data.

LDA was able to distill some very diverse set of topics from the Reuters collection. Still there are some undesired words like “7-apr-1987”, “said”, “he” etc. LDA treats these words similar to any other word in the collection. So, more number of iterations is usually necessary to find better topic models.

The unwanted words don’t go away easily because of the high frequency with which they occur. It is found that these words belong to any topic with a higher probability than the keywords. This is clear if we try to examine the documents talking about these topics in the corpus. However, words like “said”, “he” etc did not go even after pruning high frequency words using the dictionary-vectorizer. Can LDA do something better? Yes, it can!

One parameter we didn’t tweak in this run was the topic smoothing parameter (`-a`). Since text data is very noisy, it induces error in the LDA estimation. LDA can work around it by increasing smoothing value to increase the effect of keywords that occur infrequently. Doing this decreases the effect of the high frequency words, as well. This causes LDA to take more number of iterations to produce a meaningful topic model.

By default, LDA keeps this smoothing parameter as `50/numTopics`. In our sample run, it was `5`. Let us increase the smoothing value to say `20`, and re-rerun LDA. After the iterations finish, inspect the output using the `LDAPrintTopics` class.

Topic 0	Topic 1	Topic 2	Topic 3	Topic 4
production	year	said	stock	vs
tonnes	growth	banks	corp	mln
price	foreign	have	securities	cts

oil	last	analysts	inc	net
department	billion	market	reuter	loss

Table 9.2 Top 5 words in selected topics from LDA topic modeling of Reuters data after increased smoothing is applied.

The output is displayed in table 9.2. Effects of high frequency words are still there, but the topics look like they have become more coherent.

9.5.4 Applications of Topic Modeling

Topic modeling output files are of the (key, value) format (`IntPairWritable, DoubleWritable`). The key is a pair of integers, first being the topic id and the second the feature id. The value is the likelihood of the word being in the model. We can use these models for many practical purposes:

- Use them as centroids and associate documents to the nearest center using any distance measure
- Assign label to them and use them as models for classification again using some distance measure
- Topic collections can be visualized as related tag clouds similar to Digg and Del.icio.us. We will explore more on this in our chapter on case studies
- Visualize topics across time. We model topics in news articles by month or by year. We can see trends in topics over time. An interesting experiment is the topic modeling of science across time. If we look closer, we will see that the most mentioned words in science journals of 1890s was about steam engine, in 1940s about atomic research, in 1990s about polymer and semiconductor devices. The experiment is explained in this website: http://www.cs.princeton.edu/~blei/topicmodeling.html
- Words in topic models can be used to improve search coverage. Using this information, a person can search for “Cola”, and get results for the queries “Coca-Cola” and “Pepsi” along with it

LDA is an algorithm, which can uncover interesting clusters and word relationship from a corpus. People are still trying to discover ways to fully utilize all this information. Mahout LDA helps us analyze millions of documents over large number of servers. Since it runs very fast, it is easy to experiment with it. We will explore LDA in a case study in Chapter 12 and show how it is used to boost the related document framework we are trying show.

9.6 Summary

In this chapter, we saw the clustering algorithms Mahout had to offer. The chapter started with the various categories of clustering algorithms based on their clustering strategy and are summarized table 9.3.

Algorithms	In-memory implementation	Map/Reduce implementation	Fixed clusters	Partial membership
K-Means	KMeansClusterer	KMeansDriver	Y	N
Canopy	CanopyClusterer	CanopyDriver	N	N
K-Means++	KMeansPlusClusterer	KMeansPlusDriver	N	N
Fuzzy K-Means	FuzzyKMeansClusterer	FuzzyKMeansDriver	Y	Y
Dirichlet process	DirichletClusterer	DirichletDriver	N	Y
LDA	N/A	LDADriver	Y	Y

Table 9.3 A summary of the different clustering algorithms in mahout, the entry-point classes, and their properties.

The Mahout implementation of the popular K-Means algorithm works great for small and big datasets. A good estimation of the centroids of the clusters made clustering faster. Due to this reason, we explored ways to improve centroid estimation. The Canopy clustering and K-Means++ algorithms did fast and approximate clustering of the data and estimated the centroid of the clusters approximately. By using these centroids as starting point, K-Means iterations were found to converge much faster than before. We saw the various parameters in K-Means and used it to create a clustering module for a news website. Using the distance measure classes in Mahout, we were able to tune the news-clustering module to get better quality of clusters for text data.

Fuzzy K-Means clustering gives more information related to partial membership of a document into various clusters and Fuzzy K-Means has better convergence properties than just K-Means. We tuned our clustering module to use Fuzzy K-Means to help identify this soft membership information. Due to the limitation of fixing a k value in K-Means and Fuzzy K-Means, we explored other options and found model-based clustering algorithm to be a good replacement for both of them.

Model based clustering algorithm in Mahout, the Dirichlet process clustering did not just assign points into a set of clusters. It was able to explain how well the model fit the data as well as the distribution of points in the cluster. This algorithm was able to describe the clusters in some very difficult dataset where previous methods failed. Dirichlet process clustering proved to be a powerful tool to describe such data.

Finally, we looked at LDA a recent advancement in the area of clustering which was able to model the data into mixture of topics. These topics are not only clusters of documents but also a probabilistic distribution of words. LDA could jointly cluster the set of words into topics and make the set of documents a mixture of topics. LDA opened up new possibilities where we are able to identify connections between various words purely from the observed text corpus.

The actual insight of what works best for our data comes with experimentation. We have powerful tools in the Mahout clustering package, which are built on top of Hadoop that gives us the power to scale to data of any size by simply adding more machines to the cluster.

The next few chapters will be focused more on tuning a clustering algorithm for speed and quality. Over the way, we will refine our news clustering code, and finally demonstrate the related-articles feature in action. We will also explore some very interesting problems as case studies, which the clustering algorithms in Mahout help solve. Next, in Chapter 9, we will learn about some lesser known tools and techniques present in Mahout to help understand and improve the quality of clustering.