Mahout IN ACTION

Sean Owen
Robin Anil
Ted Dunning
Ellen Friedman

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Mahout in Action version 7

For more information on this and other Manning titles go to
www.manning.com

Table of Contents

Part 3 Classification

13. Introduction to classification

14. Training a classifier

15. Evaluating and tuning a classifier

16. Deploying a classifier

17. Case study: Shop it To Me

Appendices

A. JVM tuning

B. Mahout math

C. Resources

1

Meet Apache Mahout

This chapter covers:

- What Apache Mahout is, and where it came from
- 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, Apache Mahout, to effective use in real life. It has three defining qualities.

First, 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 also *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.

Finally, 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.

To set the stage, this chapter will take a brief look at the sorts of machine learning that Mahout can help you perform on your data – recommender engines, clustering, classification – by looking at some familiar real-world instances.

In preparation for hands-on interaction with Mahout throughout the book, you will also step through some necessary setup and installation that will prepare you to work with the project.

1.1 Mahout's Story

First, some background on Mahout itself is in order. You may be wondering how to *say* "Mahout" – as it is commonly Anglicized, 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.

Figure 1.1 shows some of Mahout's lineage within the Apache Foundation. As of April 2010, Mahout has become a top-level Apache project in its own right, and got a brand-new elephant rider logo to boot.

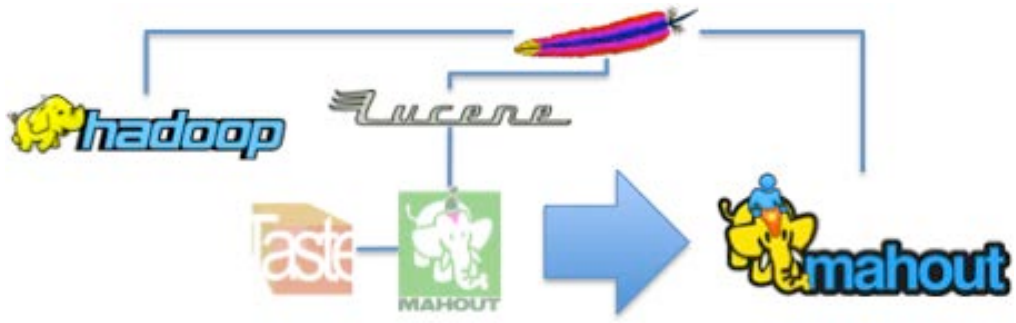


Figure 1.1 Apache Mahout and its related projects within the Apache Foundation.

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!

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*. This is by no means all that exists within Mahout, but are the most prominent and mature themes at the time of writing. These therefore are the scope of this book.

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 Mahout's Machine Learning Themes

While Mahout is, in theory, a project open to implementations of all kinds of machine learning techniques, it is in practice a project that focuses on three key areas of machine learning at the moment. These are recommender engines (collaborative filtering), clustering, and classification.

1.2.1 Recommender Engines

Recommender engines are the most immediately recognizable machine learning technique in use today. You will have 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 1.2.
- 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.

Figure 1.2 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.

As Amazon and others have demonstrated, recommenders can have concrete commercial value too, by enabling smart cross-selling opportunities. One firm reports that recommending products to users can drive an 8-12% increase in sales[1].

1.2.2 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 1.3 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.

Figure 1.3. 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.

Clustering helps discover structure, and even hierarchy, among a large collection of things which may be otherwise difficult to make sense of. Enterprises might use this to discover hidden groupings among users, or organize a large collection of documents sensibly, or discover common usage patterns for a site based on logs.

1.2.3 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:

[1] http://www.practicalecommerce.com/articles/1942-10-Questions-on-Product-Recommendations

- 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 1.4.
- 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

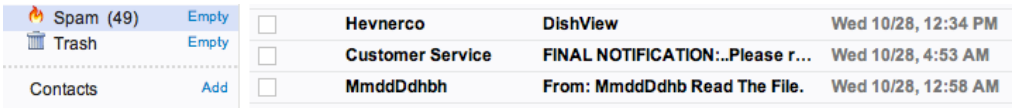


Figure 1.4 Spam messages as detected by Yahoo! Mail. Based on reports of email spam from users, plus other analysis, the system has learned certain attributes that usually identify spam. For example, messages mentioning "viagra" are frequently spam – as are those with clever misspellings like "v1agra". The presence of such terms are an example of an attribute that a spam classifier can learn.

Classification helps decide whether a new input or thing matches a previously observed pattern or not, and is often used to classify behavior or patterns as unusual. It could be used to detect suspicious network activity or fraud. It might be used to figure out when a user's message indicates frustration or satisfaction.

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. And, as mentioned before, one of Mahout's key reasons for being is to produce implementations of these techniques that do scale up to huge input.

1.3 Tackling large scale with Mahout and Hadoop

How real is the problem of scale in machine learning algorithms? Let's consider a few examples of the size of problems where you might deploy Mahout.

Consider that Picasa may have hosted over half a billion photos even three years ago, according to some crude estimates[2]. 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[3]. 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. Without an implementation such as Mahout, these would be impossible tasks. 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.

Sophisticated machine learning techniques, applied at scale, were until recently only something that large, advanced technology companies could consider using. But today computing power is cheaper than ever and more accessible via open-source frameworks like Apache's Hadoop. Mahout attempts to

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

complete the puzzle by providing quality, open-source implementations capable of solving problems at this scale, with Hadoop, and putting this into the hands of all technology organizations.

Some of Mahout makes use of Hadoop, 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.

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.

1.4 Setting up Mahout

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.

1.4.1 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 to come 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. NetBeans (http://netbeans.org/) is also a popular, free IDE. 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.

1.4.2 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.

1.4.3 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 https://cwiki.apache.org/confluence/display/MAHOUT/Downloads; 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 https://cwiki.apache.org/confluence/display/MAHOUT/Version+Control). 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.

1.4.4 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 copy of Hadoop version 0.20.2 from the Hadoop web site at

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.

1.5 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, you 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. These techniques have been deployed to solve real problems and even generate value for enterprises -- and are now accessible via Mahout.

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 this book intends to 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:

- What recommender are, within Mahout
- A first look at a `Recommender` in action
- Evaluating accuracy and quality of recommender engines
- 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 that I would enjoy a club sandwich, which is mostly the same sandwich but with turkey. Likewise, people tend to like things that similar people like.

These patterns can be used to predict such likes and dislikes. Recommendation is all about predicting these patterns of taste, and using them to discover new and desirable things you didn't already know about.

After introducing the idea of recommendation in more depth, this chapter will help you experiment with some Mahout code to run a simple recommender engine, and understand how well it works, in order to give you an immediate feel for how Mahout works in this regard.

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.

These are different, but valid strategies for discovering new things: to discover items you may like, you could look to what people with similar tastes seem to like. On the other hand, you could figure out what items are like the ones you already like, again by looking to others' apparent preferences. In fact, these describe the two broadest categories of recommender engine algorithms: "user-based" and "item-based" recommenders, both of which are well-represented within Mahout.

2.1.1 Collaborative filtering, not content-based recommendation

Strictly speaking, the scenarios above 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 does not care 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 is 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 you will build a recommender for a dating site.

For now, however, it is time to experiment with collaborative filtering within Mahout by creating some simple input and finding recommendations based on the input.

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 includes implementations of several other algorithms as well, but, for now we will explore a simple user-based recommender.

2.2.1 Creating the input

To explore recommendations in Mahout, it’s best to start with a trivial example. Input to the recommender is required – 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, simply called “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! 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
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
```

User 1 expresses preference 5.0 for item 101

User 2 expresses preference 2.5 for item 102

With some study, trends appear. 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.

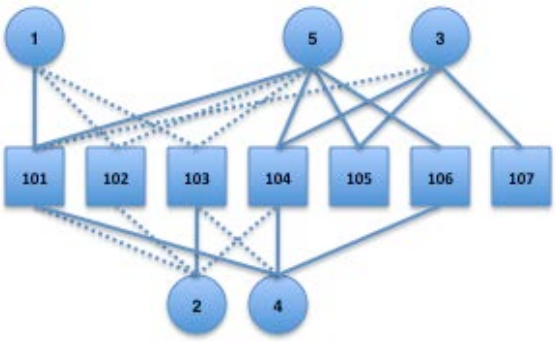


Figure 2.1 Relationships between users 1 to 5 and items 101 to 107. Dashed lines represent associations that seem negative -- the user does not seem to like the item much, but expresses a relationship to the item.

2.2.2 Creating a Recommender

So what book might you 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, it would be good to 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, code listings 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 the example.

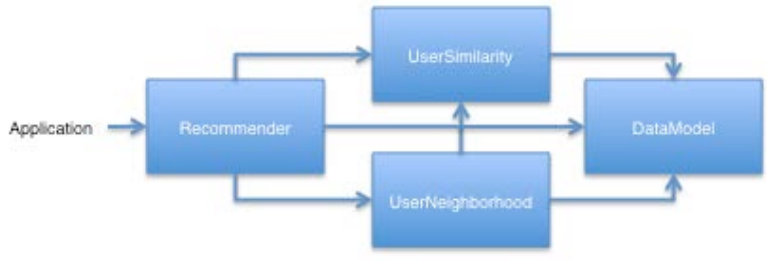


Figure 2.2 Simplified illustration of component interaction in a Mahout user-based recommender

Before discussing each of these components in 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]`

The request 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. 107 did not appear, which was also recommendable, but only associated to a user with different tastes. It picked 104 over 106, and this makes sense after noting that 104 is a bit more highly rated overall. Further, the output contained 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*, it's best to investigate the *question*. What exactly is a good recommendation? And how does one 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 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 it’s possible to 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	$=\sqrt{((0.5^2 + 3.0^2 + 1.0^2) / 3)} = 1.8484$		

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, upcoming examples will use it.

2.3.2 Running RecommenderEvaluator

Let's revisit the example code and instead evaluate the simple recommender, on this 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 there is no `Recommender` passed to this method. This is because, inside, the method will need to build a `Recommender` around a newly created training `DataModel`. So the caller must provide an object that can build a `Recommender` from a `DataModel` – a `RecommenderBuilder`. Here, it builds the same implementation that was tried earlier in this 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 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: it's not *strictly* necessary *to* estimate preference values in order to produce recommendations. It's not always essential to present estimated preference values to users. In many cases, just an ordered list of recommendations, from best to worst, is sufficient. In fact, in some cases the exact ordering of the list doesn't matter much – 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.

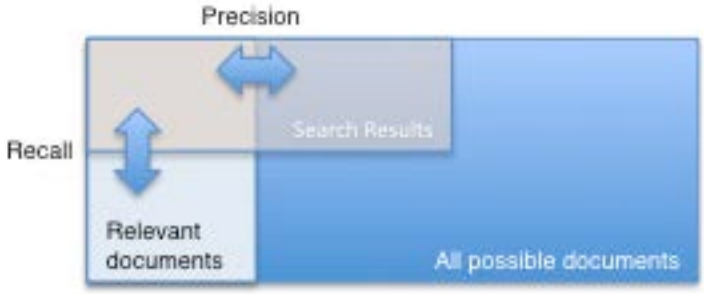


Figure 2.3 An illustration of precision and recall in the context of search results

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. The next section will define "good".

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 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? The framework was asked to decide. It didn't receive 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 this simple data set again. Let's imagine the preferences for items 101, 102 and 103 were withheld as test data. The preference values for these are 4.0, 3.0 and 2.0. With these values missing from the training data, a recommender engine ought to recommend 101 before 102, and 102 before 103, because 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 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 takes 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.

2.4.2 Problems with precision and recall

The usefulness of precision and recall tests in the context of recommenders depends entirely on how well a "good" recommendation can be defined. Above, the threshold was given, or defined by the framework. A poor choice will hurt the usefulness of the resulting score.

There's a subtler problem with these tests, though. Here, they necessarily pick the set of good recommendations from among items for which the user has already expressed some preference. But the best recommendations are of course not necessarily among those the user already knows about!

Imagine running such a test for a user who would absolutely love the little-known French cult film, "My Brother The Armoire". Let's say it's objectively a great recommendation for this user. But, the user has never heard of this movie. If a recommender actually returned this film when recommending movies, it would be penalized; the test framework can only pick good recommendations from among those in the user's set of preferences already.

The issue is further complicated when the preferences are 'boolean' and contain no preference value. There is not even a notion of relative preference on which to select a subset of good items. The best the test can do is to randomly select some preferred items as the good ones.

The test nevertheless has some use. The items a user prefers are a reasonably proxy for the best recommendations for the user, but by no means a perfect one. In the case of boolean preference data, only a precision-recall test is available anyway. It is worth understanding the test's limitations in this context.

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. Although examples with *large* amounts real data are still a couple chapters away, it's already possible 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 more will be explored in this book. From the GroupLens site, 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, which built a `RecommenderEvaluator`, and, try passing in the location of `ua.base` instead of the small data file. 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 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 will reappear 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 20ecommenderBuilder = 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 that was 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.

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, it's important to spend some time with another foundational concept for recommenders in Mahout in the next chapter: representation of data.

3

Representing Recommender Data

This chapter covers:

- How Mahout represents recommender data
- DataModel implementations and usage
- Handling 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. Having high-quality data is a good thing, and generally, having lots of it is also good. Recommender algorithms are data-intensive by nature; their computations access a great deal of information. Runtime performance is therefore greatly affected by quantity of data and its representation. Intelligent choice of data structures can affect performance by orders of magnitude, and at scale, it matters most.

This entire chapter explores key classes in Mahout for representing and accessing recommender-related data. You will get a better sense of why users and items, and their associated preferences, are represented the way they are in Mahout for efficiency and scalability. This chapter will also look in detail at the key abstraction in Mahout that provides access to this data: a `DataModel`.

Finally, we will look at problems and opportunities that arise when user and item data has no concept of ratings or preferences values – so-called “boolean preferences” – which require special handling. The next section will introduce the basic unit of recommender data, a user-item preference.

3.1 Representing Preference Data

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.

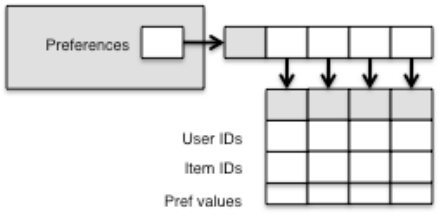


Figure 3.1 A less-efficient representation of preferences using an array of Preference objects. Gray areas represent, roughly, Object overhead. White areas are data, including Object references.

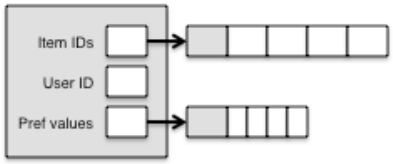


Figure 3.2 A more efficient representation using GenericUserPreferenceArray.

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.

3.1.3 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 you may 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.

3.1.4 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 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?

3.2 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. This section focuses on some of the highlights; a more detailed account of `DataModel`'s API can be found in the online javadoc documentation.

3.2.1 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.

3.2.2 File-based data

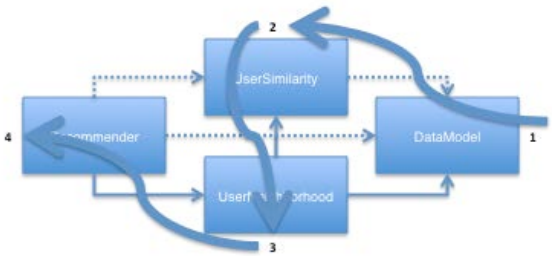
You will not typically use `GenericDataModel` directly. Instead, you will likely encounter it via `FileDataModel` – which reads data from a file and stores the resulting preference data in memory, in a `GenericDataModel`.

Just about any reasonable file will do, such as the simple comma-separated value file from the first section. 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.

3.2.3 Refreshable components

While talking about loading data, it's useful to 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. You 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.

3.2.4 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.

3.2.5 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.

3.2.6 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.

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

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

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`.

3.2.7 Configuring via JNDI

It also assumes that the database containing this table is accessible via a `DataSource` object registered to JNDI[4] 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.

3.2.8 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

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

(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_database_host");
dataSource.setUser("my_user");
dataSource.setPassword("my_password");
dataSource.setDatabaseName("my_database_name");
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 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. However sometimes a *value* is not present, or, ignoring it is actually useful.

3.3 Coping without preference values

Sometimes, the preferences that go into a recommender engine have no value. That is, a user and item are associated, but there is no notion of the strength of that association. 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.

In such cases, there is no choice; there is no preference value in the input to begin with. The techniques and advice in the remainder of the chapter still apply to these scenarios. However, it's also sometimes advantageous to ignore preference values when they do exist in the input. At least, sometimes, it doesn't hurt.

It's not a question of forgetting all associations between users and items. Rather, the idea is to ignore 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, you might do as well to simply consider what movies you have seen. Rather than know "user 1 expresses preference 4.5 for movie 103", it could be useful to forget the 4.5 and taking, as input, data like "user 1 is associated to movie 103." Figure 3.4 attempts to illustrate the difference.

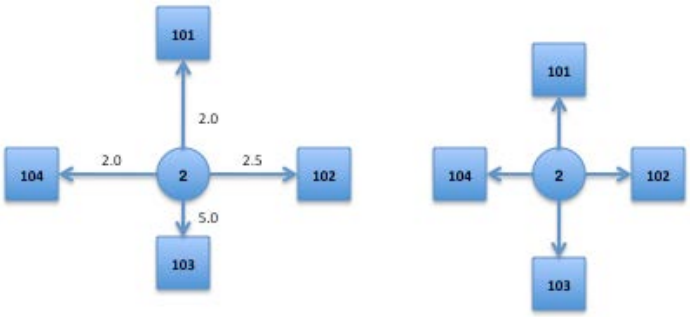


Figure 3.4 An illustration of user relationships to items with preference values (left) and "boolean data", without preference values (right)

In Mahout-speak, these are called "boolean preferences," for lack of a better term, because an association can have one of two values: exists, or doesn't exist. This does *not* mean that the data consists of "yes" and "no" preferences for items. This would give three states for every possible user-item association: likes, dislikes, or nothing at all.

3.3.1 When to ignore values

Why would one ignore preference values? It 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, forgetting this data may be worse when just recommending from classical composers. This is true; a good solution for one domain does not always translate to others.

3.3.2 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 seen, 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.

This value comes from testing the twin of `GenericDataModel`, 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.

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

Listing 3.7 Creating and evaluating with boolean data

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

RecommenderEvaluator evaluator =
  new AverageAbsoluteDifferenceRecommenderEvaluator();

RecommenderBuilder recommenderBuilder = new RecommenderBuilder() {
  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() {
  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 to 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, try running this code – it will *not* complete successfully.

3.3.3 Selecting compatible implementations

You should find that running this code results in an `IllegalArgumentException` from the `PearsonCorrelationSimilarity` constructor. This may be surprising at first: isn't `GenericBooleanPrefDataModel` also a `DataModel`, and nearly the same as GenericDataModel except that it doesn't store distinct preference values?

This similarity metric, along with `EuclideanDistanceSimilarity`, refuse to work without preference values since their results would be undefined or meaningless, producing useless results. The Pearson correlation between two data sets will be undefined if the two data sets are simply the same value, repeated[5]. Here, the `DataModel` pretends that all preference values are 1.0. Similarly, computing

[5] 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.

a Euclidean distance between users who all correspond to the same point in space -- (1.0, 1.0, …, 1.0) -- would be meaningless since all similarities would be 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. To solve the immediate problem, substitute a 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 0.0. That's excellent, since it means perfect prediction. Is it too good to be true?

Unfortunately, yes. That is 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` 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.

3.4 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 incompatibilities between this sort of data and standard components like `PearsonCorrelationSimilarity`. 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 a recommender, now it is time to examine the recommenders themselves in detail. This is where the real action begins.

Previous chapters alluded to two well-known styles of recommender algorithm, both of which are implemented in Mahout: user-based recommenders and item-based recommenders. In fact, you already encountered a user-based recommender in chapter 2. This chapter will explore the theory of these algorithms, as well as the Mahout implementation of both, in detail.

Both algorithms rely on a similarity metric, or notion of sameness between two things, whether they are users or items. There are many ways to define similarity, and so this chapter will introduce in detail your choices within Mahout. These include implementations based on the Pearson correlation, log likelihood, Spearman correlation, Tanimoto coefficient and more.

Finally, you will also become familiar with other styles of recommender algorithm implemented within Mahout, including slope-one recommenders, SVD-based recommenders, and clustering-based recommenders.

4.1 Understanding user-based recommendation

If you've seen a recommender algorithm explained before, chances are it was a user-based recommender algorithm. This is the approach described in some of the earliest research in the field, and is certainly implemented within Mahout. 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 youngster, by well-meaning adults. One of these adults evidently 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. 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 the scenario and imagine how it could have gone better:

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

EMPLOYEE: What kind of music or bands does he like?

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”...

That's better. The recommendation was based on the assumption that two good friends share somewhat similar taste in music. 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: What kind of music or bands does he like?

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...

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 similarity between 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 those two adults kept 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. (And then, they might become Mahout users!)

4.2.1 The algorithm

The user-based recommender algorithm comes out of this intuition. It is a 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 considers 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 looks 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, the values are averaged to come up with an estimate -- a weighted average, that is. Each preference value is weighted in the average by how similar that user is to the target user. The more similar a user, the more heavily his or her preference value is weighted.

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 w
  compute a similarity s between u and w
  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 similar users are found first, before seeing what those most-similar users are interested in. Those items become the candidates for recommendation. The rest is the same. This is the standard user-based recommender algorithm, and the way it is implemented in Mahout.

4.2.2 Implementing the algorithm with GenericUserBasedRecommender

The very first example showed a user-based recommender in action in Mahout. 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 – the dialog above illustrated 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 the results change considerably. You can begin to see that there is not merely one way to produce recommendations – and this is still looking at one facet of one approach that can be adjusted. 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 the following components are assembled:

- 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`

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 with 100 times more data. 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 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. The heap space available to Java must increase.

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.

4.2.4 Exploring user neighborhoods

Let's next evaluate the recommender accuracy. Below is the boilerplate evaluation code one more time; going forward, we figure you've got the hang of it and can construct and run evaluations on your own.

Now, let's look at possibilities for configuring and modifying the neighborhood implementation. Remember, this is 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. However, 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.

4.2.5 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). The decision to use the 100 users whose similarity is greatest in order to make recommendations is arbitrary. What if this were 10? Recommendations would be based on fewer similar users, but would exclude some less-similar users from consideration.

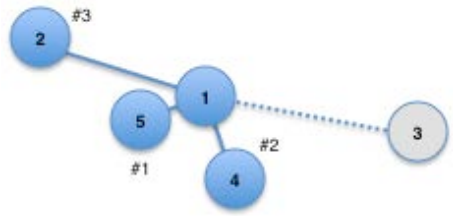


Figure 4.1 An illustration of defining a neighborhood of most similar users by picking a fixed number of closest neighbors. Here, distance illustrates similarity: farther means less similar. In this picture, neighborhood around user 1 is chosen to consist of the three most similar users: 5, 4, and 2.

Try replacing 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.

Try a neighborhood of 500 users; the result drops to 0.75, which is of course better. You could evaluate many values and figure out the optimal setting for this data set. The lesson is that there is no magic value; some experimentation with real data is necessary to tune your recommender.

4.2.6 Threshold-based neighborhood

What if you 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? It's possible to 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, the example uses 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 instead instantiate a new `ThresholdUserNeighborhood(0.7, similarity, model)`.

Now, the evaluator scores the recommender at 0.84. What if the neighborhood were more selective, using 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. Examples that follow will use a threshold-based neighborhood with threshold 0.5.

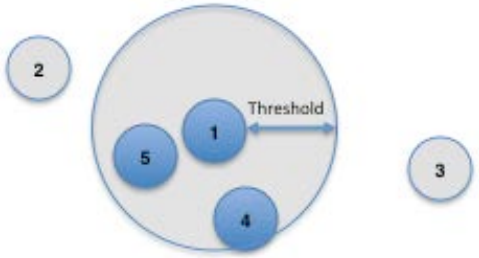


Figure 4.2 An illustration of defining a neighborhood of most-similar users with a similarity threshold.

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

4.3 Exploring similarity metrics

Another important part of user-based recommenders is 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. The same is true of the user-based recommender's cousin, the item-based recommender, which also relies on similarity. This components is so important that we will spend almost a third of the chapter investigating standard similarity metrics and how they are implemented in Mahout.

4.3.1 Pearson correlation-based similarity

So far, examples have used the implementation `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. It measures 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:

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 reasoning about user 1 can't really include items 104 through 106, since user 1's preference for 104 through 106 is not known. The similarity computation can only operate on items that both users have expressed a preference for. An upcoming section will look at what happens when "missing" preference values like this are *inferred*.

The Pearson correlation captures these notions, as can be seen from the table 4.2. The details of the computation here are not reproduced 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

4.3.2 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. For example, if user 5 had expressed a preference of 3.0 for all three items above, it would not be possible to 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[6], 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.

4.3.3 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, it's desirable 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.

4.3.4 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[7] *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. The value should 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.

[6] https://cwiki.apache.org/confluence/display/MAHOUT/Recommender+Documentation
[7] Recall this is the square root of the sum of squares of the differences in coordinates

	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

After changing the last example to use `EuclideanDistanceSimilarity`, the result is 0.75 – it happens to be a little better in this case, but barely. Note that some notion of similarity was computable 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. This implementation also has the same possibly counterintuitive behavior: users 1 and 4 have a higher similarity than users 1 and 5.

4.3.5 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 contrast, 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 a distance was used to form a similarity metric above. In this case, the cosine of the angle leads to a 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 small angles should 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, you will discover that they actually reduce to the same computation when the two series of input values each have a mean of 0 ("centered"). The Mahout implementation centers the input, so, the two are the same.

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

4.3.6 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, his or her least-preferred item's preference value is overwritten with a "1". Then the next-least-preferred item's preference value is changed to "2", and so on. To illustrate this, imagine that you were rating movies and gave your least-preferred movie 1 star, the next-least favorite 2 stars, and so on. Then, a Pearson correlation is computed 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 possibilities that have been explored before.

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 from before. 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.

4.3.7 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, 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.

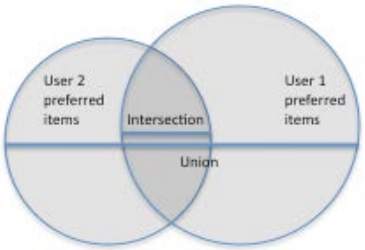


Figure 4.3 The Tanimoto coefficient is the ratio of the size of the intersection, or overlap in two users' preferred items (dark area), to the union of the users' preferred items (dark and light areas together).

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. It's possible to 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 3	X			X	X		X	0.17
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 the 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 the GroupLens data set, using this metric gives a slightly worse score of 0.82.

4.3.8 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 the two should be. 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 this data set and recommender, it improves performance over `TanimotoCoefficientSimilarity`, to 0.73.

4.4.9 Inferring preferences

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 a default value filled in for all the missing data points? For example, the framework 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.

At last, you have seen all of the user-user similarity metrics that are implemented within Mahout. This knowledge pays double dividends, because the very same implementations in Mahout also provide an analogous notion of item-item similarity. That is, the same computation can be applied to define how similar items are, not just users. This is just in time, since notion of similarity is likewise the basis of the next style of Mahout recommender implementation you will meet, the item-based recommender.

4.4 Item-based recommendation

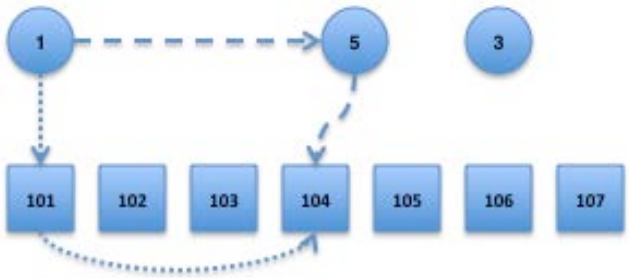
We've looked at user-based recommenders in Mahout -- not one recommender, but tools to build a nearly limitless number of variations on the basic user-based approach, by plugging in different and differently configured components into the implementation.

It's natural to look next at *item-based* recommenders. This section will be shorter, since several of the components above (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*. In Mahout, this means they are based on an `ItemSimilarity` implementation instead of `UserSimilarity`. 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: What kind of music or bands does he like?
>
> 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. It is different from previous examples, too. The record store employee is recommending an item that is similar to something they 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?"



corrections to the Author Online forum:
ɪrum.jspa?forumID=623

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.

4.4.1 The algorithm

The algorithm will feel familiar, having seen user-based recommenders already. This is also how the algorithm is implemented in Mahout.

```
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 items is relatively low compared to the number of users, the performance advantage could be significant.

Also, items are typically less subject to change than users. When items are things like DVDs, it's reasonable to expect that over time, as you acquire more data, that estimates of the similarities between items *converge*. There is 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.

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.

In Mahout, the class `GenericItemSimilarity` can be used to pre-compute and store the results from any `ItemSimilarity`. It can be applied to any of the implementations you have seen, and added to code snippets below, if you wish.

4.4.2 Exploring the item-based recommender

Next, you will insert a simple item-based recommender into the familiar evaluation framework, using the following code. It deploys a `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

```
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. 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` 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 the 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. Item-based recommenders are generally 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.

4.5 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, on average, people rate Scarface higher by 1.0 than Carlito's Way. Let's also say that everyone rates Scarface the same as The Godfather, on average. And now, there is a user who rates Carlito's Way 2.0, and The Godfather 4.0. What is a good estimate of his or her preference for Scarface?

Based on Carlito's Way, a good guess would be 2.0 + 1.0 = 3.0. Based on The Godfather, it might be 4.0 + 0.0 = 4.0. A better guess still might be the average of the two: 3.5. This is the essence of the slope-one recommender approach.

4.5.1 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 it's valid to 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". It just remains 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 from several prior examples throughout the book are shown 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!

4.5.2 Slope-one in practice

You 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 so far. 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. It's easy to compute a diff for these two films. Would it be as useful as the diff 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, some form of weighting may help 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. Disable them to see the effect:

Listing 4.7 Selecting no weighting with a SlopeOneRecommender

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

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

4.5.3 DiffStorage and memory considerations

Slope-one does have its price: memory consumption. In fact, if you tweak the evaluation to use even 10% of all data (about 100,000 ratings), even a gigabyte of heap space 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. It may become necessary to store diffs elsewhere.

Fortunately, implementations like `MySQLJDBCDiffStorage` exist for this purpose, to allow diffs to be computed and update from a database. 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.

4.5.4 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. Chapter 6 will introduce all of the Hadoop-related recommender support in Mahout, to explore this process.

4.6 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.

4.6.1 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 this 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. They want car recommendations. 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 that you 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 the 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.

4.6.2 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 also embodied in `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.

4.6.3 Cluster-based recommendation

This approach is best thought of as a variant on user-based recommendation. Here, instead of recommending items to users, items are recommended 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.

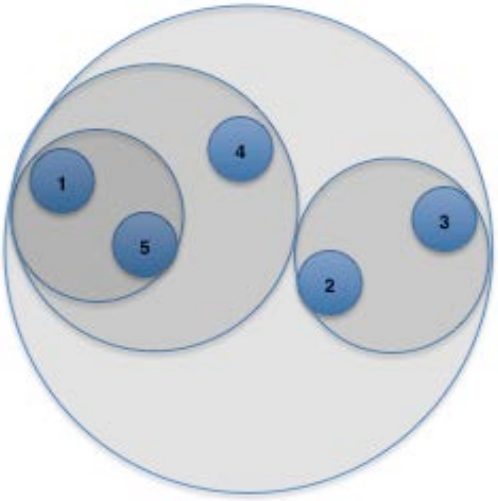


Figure 4.5 An illustration of clustering. Users 1 and 5 are clustered together first, as are 2 and 3, as they are closest. 4 is then clustered with the 1-5 cluster to create a larger cluster, one step up in the "tree".

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.

4.7 Comparison to other 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.

4.7.1 Injecting content-based techniques into Mahout

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, implementations have derived 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.

4.7.2 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.

4.8 Comparison 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.

4.9 Summary

In this chapter, we thoroughly explored the core recommender algorithms offered by Mahout. See table 4.9 for a catalog of the implementations we have seen, and their key characteristics.

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 examined another canonical recommendation technique, item-based recommendation, as implemented by `GenericItemBasedRecommender`. It 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 in Mahout, their key input parameters, and key features to consider when choosing an implementation.

Implementation	Key Parameters	Key Features
GenericUserBasedRecommender	• User similarity metric • Neighborhood definition and size	• "Conventional" implementation • 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") • Item similarity metric	• Good when number of items is relatively smaller

TreeClusteringRecommender	• Neighborhood size • Number of clusters • Cluster similarity definition • User similarity metric	• Recommendations are fast at runtime • Requires large precomputation • 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, this book has toured the recommender algorithms and variants that Apache Mahout provides, and how to evaluate the accuracy and performance of a recommender. The next step is to 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 the recommender will be turned into 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 using Mahout for your data set. You will try an approach, collect data, understand the results, and repeat many times. Many approaches won't go anywhere, but that's good information as well. This "brute force" approach is appropriate, since it's relatively painless to evaluate an approach in Mahout, 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 Analyzing example data from a dating site

This example 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". The presentation of profiles on the site suggests 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[8], and published by Vaclav Petricek (http://www.occamslab.com/petricek/data/). Please obtain a copy of the data from this link[9].

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

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

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.

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 the user IDs are not 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.

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. However, its memory requirements can be capped.

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 that is given 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.

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

However that means we know something more about each item. Male profiles are much more similar to one another than female profiles -- at least, in the context of being recommended as potential dates. If 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. This information could be the basis for an item-item similarity metric.

This isn't a perfect assumption. Without becoming sidetracked on sensitive issues of sexuality, 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 we have seen in Mahout. Later sections in this chapter will 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, it's necessary to pick a recommender implementation from Mahout. The 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?

It's impossible to deduce the right implementation from looking at the data; some empirical testing is needed. Armed with an evaluation framework, the next step is to collect some data.

5.2.1 User-based recommenders

User-based recommenders are a natural first stop. Several different similarity metrics and neighborhood definitions are available in Mahout. 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*

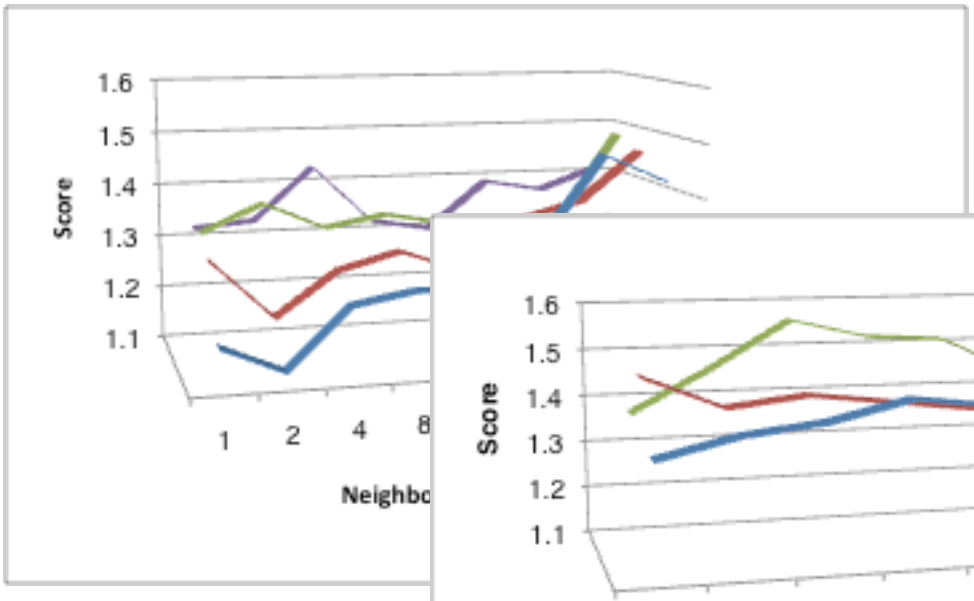


Figure 5.1 Visualization of values in table 5.1.

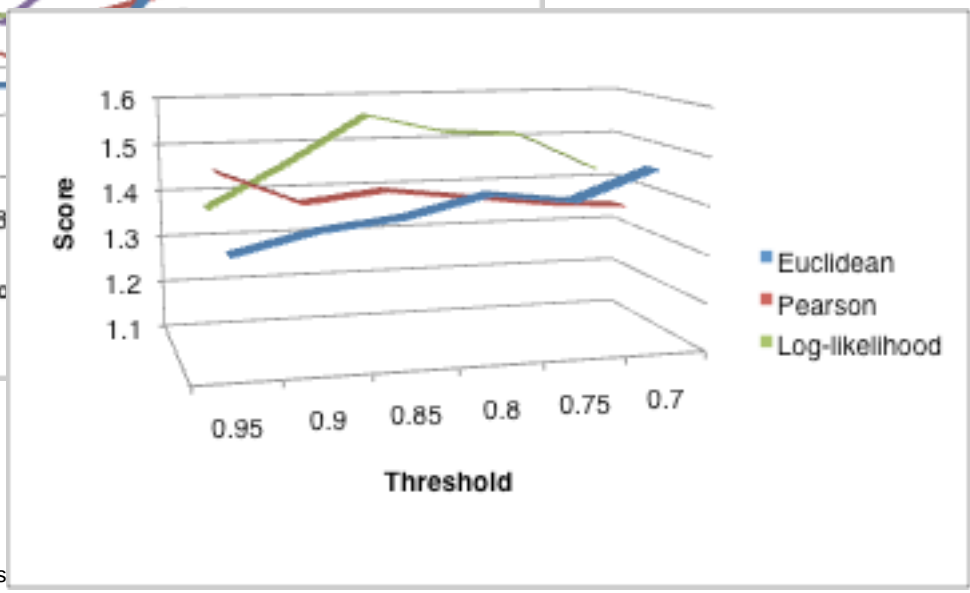


Figure 5.2 Visualization of table 5.2.

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. This is generally true, and not an artifact of how Mahout works. 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, it might have been more useful to test thresholds from, say, 0.4 on down. In fact, with a threshold of 0.3, the best evaluation score approaches 1.2.

Similarly, although there is an apparent best value for *n* in the nearest-*n* user neighborhood data, there is not quite the same best value 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.

Separately, you may continue looking for even better configurations. For our purposes here, take the current best solution in Mahout to be:

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

5.2.2 Item-based recommenders

Item-based recommenders involve just one choice: an item similarity metric. Trying each similarity metric is a straightforward way to 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. Why? Before, the algorithm computed similarities between users in a user-based approach, based on how users rated other users' profiles. Now, it's computing similarity between user profiles based on how other users rated that profile. Maybe this isn't as meaningful -- maybe ratings say

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, there is 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, 10000000L);
return new SlopeOneRecommender(
    model, Weighting.WEIGHTED, Weighting.WEIGHTED, diffStorage);
```

Indeed, from examining the Mahout 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 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, the examples experimented with a Tanimoto coefficient-based similarity metric and log-likelihood-based metric, and these are metrics which do not use preference values. However, they did not evaluate recommenders that completely ignore rating values. Such recommenders can't be evaluated in the same way -- there are no estimated preference values to compare 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.

It's possible to 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 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 rating data is ignored by using Mahout's `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 there were 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 purposes here, the book 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

It's important to look at the runtime performance of this recommender. Because it will be queried 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.

So far we have just applied standard Mahout recommenders to the data set at hand. We've had to do no customization. However, creating the best recommender system for a particular data set or site inevitably requires taking advantage of all the information you can. This in turn requires some degree of customization and specialization of standard implementations like those in Mahout to make use of the particular attributes of the problem at hand. In the next section, we will look at extending the existing Mahout implementations in small ways to take advantage of particular properties of the dating data, in order to increase the quality of recommendations.

5.3 Injecting domain-specific information

So far the recommender has not taken advantage of any domain-specific knowledge here. That is, it has taken no advantage of the fact that the ratings are, specifically, people rating people. It has used the user-profile rating data as if it could be anything at all -- ratings for books or cars or fruit. However it's often possible to incorporate additional information in the data to improve recommendation.

In the following subsections, we will look at ways to incorporate one important piece of information in this data set that has so far been unused: gender. We will create a custom `ItemSimilarity` metric based on gender, and also see how to avoid recommending users of an inappropriate gender.

5.3.1 Employing a custom item similarity metric

Because the gender of many profiles is given, you 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, call two male or two female profiles "very similar" and assign them a similarity of 1.0. Say the similarity between a male and female profile is -1.0. Finally, 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;
  }
```

```
  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;
  }

  public double[] itemSimilarities(long itemID1, long[] itemID2s) {
    double[] result = new double[itemID2s.length];
    for (int i = 0; i < itemID2s.length; i++) {
      result[i] = itemSimilarity(itemID1, itemID2s[i]);
    }
    return result;
  }

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

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

}
```

This `ItemSimilarity` metric can be paired 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 more information were available, such as the interests and hobbies expressed on each profile, it could form the basis 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 – that was an example of content-based recommendation in the last section. It had a notion of item similarity that was *not* based on user preferences, but that was based on *attributes* of the item itself. As we've said before, Mahout doesn't provide content-based recommendation implementations, but provides expansion points and APIs that let you write code to deploy it within the framework.

It's a powerful addition to pure collaborative filtering approaches, which are based only on user preferences. You can usefully inject our knowledge about items (here, people) to augment the user preference data you 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 some BookManager exists
B Boost estimate by 20%
C Don't change anything else
D Filter not-in-stock books

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 available: gender.

5.3.4 Incorporating gender in an IDRescorer

An `IDRescorer` can filter out "items", or user profiles, for users whose gender may not be of romantic interest. An implementation might work by first guessing the user's preferred gender, by examining the gender of profiles rated so far. Then, it could 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;
}

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

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

```
    }

}
```

A Cache which users rate more males
B Called separately later
C Reoptimize for fast access
D Users rating men probably like male profiles
E NaN for profiles to 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: the gender of about half of the profiles is known.

5.3.5 Packaging a custom Recommender

It will be useful for our purposes here to wrap up the entire, current recommender engine, plus the new `IDRescorer`, into one implementation. This will become necessary in the next section when one self-contained recommender engine is deployed to production. Listing 5.5 shows a Recommender implementation that contains inside it the user-based recommender engine above.

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];
  }

  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);
  }
```

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

    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));
    }

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

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

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

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

  }
```

A Need readResourceToTempFile() in production
B Construct the same user-based recommender
C All recommendations use GenderRescorer
D Rescore estimated preferences too
E Delegate rest to underlying recommender

This is a tidy, self-contained packaging of the recommender engine. If you evaluate it, the result is about 1.18: virtually unchanged, and it is good to have 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 purposes here, this tradeoff is acceptable and `LibimsetiRecommender` is the final implementation for this dating site.

5.4 Recommending to anonymous users

Another common issue you will soon run into when creating a real production-ready recommender is what to do with users that aren't registered 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 extreme 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, but not an option we will explore. Mahout is in the business of personalizing recommendations.

At the other extreme 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. It is also not an option we will explore.

Instead, we will examine two more potential approaches that are compromises between these extremes: creating "temporary users" and treating all anonymous users as like one user.

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); C
    plusAnonymousModel.clearTempPrefs();
    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 Anonymous user's "ID"
D Example anonymous user prefs

This implementation otherwise works like any other `Recommender` and may be used to recommend to real users as well.

5.4.2 Aggregating anonymous users

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 is just presenting anonymous users with a fixed set of recommendations.

5.5 Creating a web-enabled recommender

Creating a recommender that runs in your IDE is fine, but chances are you are interested in deploying this recommender in a real production application.

You may wish to deploy a recommender that you have designed and tested in Java and Mahout as a stand-alone component of your application architecture, rather than embed it inside your application's Java 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.

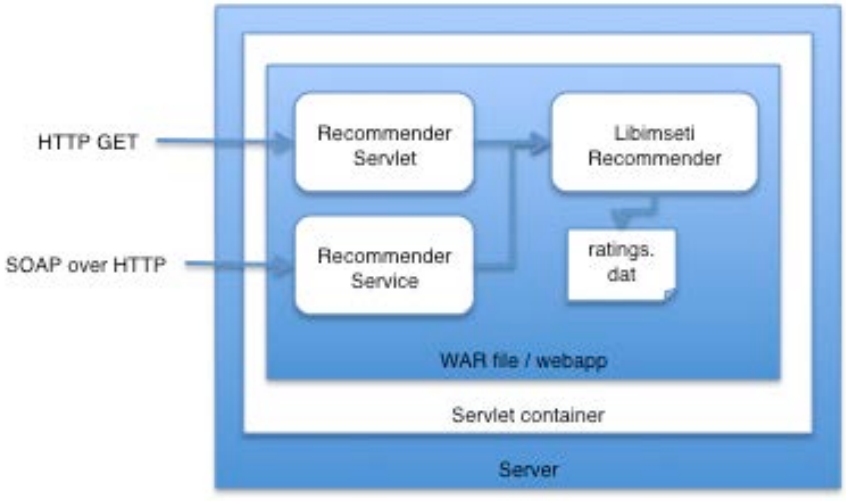


Figure 5.3 Automated WAR packaging of a recommender and deployment in a servlet container

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 -- 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.

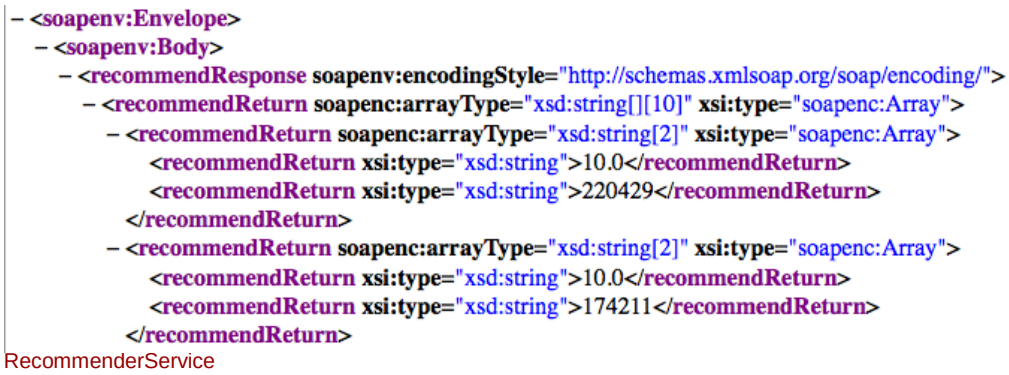


Figure 5.4 Browser rendering of the SOAP response from RecommenderService

Normally, at this point, you would be sanity-checking the results. Put yourself in your users' shoes -- do the recommendations make sense? Here, it's not possible to know who the users are or what the profiles are like, so there is no intuitive interpretation of the results. 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 that is run and then forgotten. It is a dynamic server, ingesting new information and returning answers in real time, and like any production system, it's natural to think about how service will be updated and monitored.

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.

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.

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

This book has looked at increasingly large data sets: 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. This chapter ups 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[10]. In this data set, the articles are both the users and the items, which also demonstrates how recommenders can be usefully applied, with Mahout, 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 presented so far. It calls for a new species of recommender algorithm, using a distributed computing approach from Mahout based on the MapReduce paradigm and Apache Hadoop.

We will start by examining the Wikipedia data set and understanding what it means to distribute a recommender computation. You will learn how a simple distributed recommender system is designed in a distributed environment, since it differs greatly from non-distributed implementations. You will also see how, in Mahout, this design translates into an implementation based on MapReduce and Hadoop. Finally, you will have a first occasion to run a complete Hadoop-based recommender job and see the result.

6.1 Analyzing the Wikipedia data set

We will begin here by taking a look at the Wikipedia data set, but, will see quickly why it's difficult to proceed as before due to issues of scale. We will shortly have to back up and take a look at distributing the computation to proceed any further.

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 3.2 million 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 another. It is these links that are of interest. Think of articles as "users", and articles that an article points to as "items" that the source article "likes".

Fortunately, it's not necessary 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

[10] 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.

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, from this page, download and extract `links-simple-sorted.zip`. This data set contains 130,160,392 links from 5,706,070 articles, to 3,773,865 distinct other articles. Note that there are no explicit preferences or ratings; 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 a performance perspective. If using an algorithm that involves a similarity metric, one that does not depend on preference values is appropriate, like `LogLikelihoodSimilarity`.

What is the meaning of the data, intuitively, and what is it reasonable to 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.

6.1.1 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.

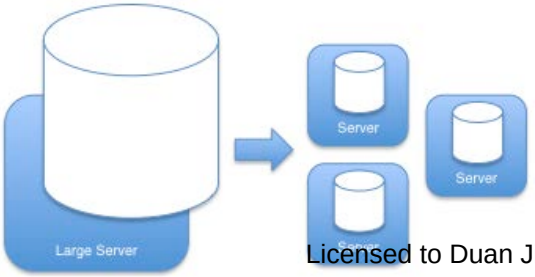
It's unfashionable these days to be unable to cope with data beyond some scale, to have some hard limit on what a system can handle. Computing resources are readily available in large quantities; the problem here is putting enough computing resources into one box. It is disproportionately expensive to 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 non-distributed 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 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, it would need access to all article-to-article links, because 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 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 Designing a distributed item-based algorithm

For problems of this scale, it is desirable and necessary to deploy a distributed approach to produce recommendations. First, this section will sketch out a distributed variation on the item-based recommender approach from before. It will be similar in some ways to that non-distributed item-based recommender. But it will certainly look different, because the non-distributed algorithm does not fully translate to the distributed world.

6.2.1 Constructing a co-occurrence matrix

The algorithm 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 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.

Something like this is needed for the algorithm: a "co-occurrence matrix". Instead of *similarity* between every pair of items, it will 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 non-distributed item-based algorithm.

	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 have been used 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 the previous recommender approaches to a matrix-based distributed computation is to conceive of a user's preferences as a vector. You 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, user preferences are 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 which 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 the 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, each user needs such a vector.

6.2.3 Producing the recommendations

To compute recommendations for user 3, 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 the small example data set, and the resulting vector R. It's possible to 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 purposes here, normalization doesn't matter, since the ordering of recommendations is the important thing, 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 MapReduce

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

The subsections that follow will introduce, one by one, the several MapReduce stages that come together into a pipeline that makes recommendations. Each individually does a little bit of the work. We will look at the inputs, outputs and purpose of each stage. Be prepared for plenty of reading; even this simple recommender algorithm will take five MapReduce stages – and this is even a simplified form of how it exists in Mahout! By the end of this section you will have seen a complete end-to-end recommender system based on Hadoop, however.

Note that this chapter will use the Hadoop APIs found in version 0.20.2 of the framework. The code presented below can be found in its complete form within Mahout, and should be runnable with Hadoop 0.20.2 or later. In particular, refer to `org.apache.mahout.cf.taste.hadoop.RecommenderJob`, which invokes the actual implementation of all of the processes below.

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.

Those unfamiliar with Hadoop should probably read and perhaps run Hadoop's short tutorial, found at http://hadoop.apache.org/common/docs/r0.20.2/mapred_tutorial.html in the current 0.20.2 release, which illustrates the basic operation of a MapReduce job in Hadoop.

6.3.2 Translating to MapReduce: Generating user vectors

In this case, the computation begins with the links data file as input. Its lines are not of the form "userID,itemID,preference". 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 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. Example: `239 / 98955: 590 22 9059`
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. Example: `98955 / 590`
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. Example: `98955 / [590:1.0, 22:1.0, 9059:1.0]`

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 that are needed 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 Mapper<LongWritable,Text,VLongWritable,VLongWritable> {
  public void map(LongWritable key,
                  Text value,
                  Context context) throws IOException, InterruptedException {
    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()));
      context.write(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
    Reducer<VLongWritable,VLongWritable,VLongWritable,VectorWritable> {
  public void reduce(VLongWritable userID,
                     Iterable<VLongWritable> itemPrefs,
                     Context context)
      throws IOException, InterruptedException {
    Vector userVector = new RandomAccessSparseVector(Integer.MAX_VALUE, 100); B
    for (VLongWritable itemPref : itemPrefs) {A
      userVector.set(itemPref.get(), 1.0f); C
    }
    context.write(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. Example: `98955 / [590:1.0,22:1.0,9059:1.0]`
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. Both mappings, from one item ID to the other and vice versa, are recorded. Example: `590 / 22`
3. The framework collects, for each item, all co-occurrences mapped from that item.
4. The reducer 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. Example: `590 / [22:3.0,95:1.0,…,9059:1.0,…]`

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 there is a pair of related implementations, of `Mapper` and `Reducer`.

Listing 6.3 Mapper component of co-occurrence computation

```
public class UserVectorToCooccurrenceMapper extends
    Mapper<VLongWritable,VectorWritable,IntWritable,IntWritable> {
  public void map(VLongWritable userID,
                  VectorWritable userVector,
                  Context context) throws IOException, InterruptedException {
    Iterator<Vector.Element> it = userVector.get().iterateNonZero(); A
    while (it.hasNext()) {
      int index1 = it.next().index();
      Iterator<Vector.Element> it2 = userVector.iterateNonZero();
      while (it2.hasNext()) {
        int index2 = it2.next().index();
        context.write(new IntWritable(index1), new IntWritable(index2)); 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
    Reducer<IntWritable,IntWritable,IntWritable,VectorWritable> {
  public void reduce(IntWritable itemIndex1,
                     Iterable<IntWritable> itemIndex2s,
                     Context context) throws IOException, InterruptedException {
    Vector cooccurrenceRow = new RandomAccessSparseVector(Integer.MAX_VALUE, 100);
    for (IntWritable intWritable : itemIndex2s) {
      int itemIndex2 = intWritable.get();
      cooccurrenceRow.set(itemIndex2, cooccurrenceRow.get(itemIndex2) + 1.0); A
    }
    context.write(itemIndex1, new VectorWritable(cooccurrenceRow)); B
  }

}
```

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

6.3.4 Translating to MapReduce: Rethinking matrix multiplication

It's now possible 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 the algorithm may derive recommendations.

However the multiplication will proceed in a different way that is more efficient here, and more naturally fits the shape of a MapReduce computation. The algorithm 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 i^th 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, 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 i^th 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, the loop iteration may be skipped 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, the algorithm 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

The columns of the co-occurrence matrix are available from an earlier step. Because the matrix is symmetric, the rows and columns are identical, so this output can be viewed as either rows or columns, conceptually. These columns are keyed by item ID. The algorithm must multiply each by every non-zero preference value for that item, across all user vectors. That is, it must 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, it produces a vector that forms part of the final recommender vector R for one user.

The difficult part here is that two different kinds of data are combined 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. A clever implementation 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:

1. Input for mapper 1 is the co-occurrence matrix: item IDs as keys, mapped to columns as `Vectors`. Example: `590` / `[22:3.0,95:1.0,…,9059:1.0,…]`
2. The map function simply echoes its input, but with the `Vector` wrapped in a `VectorOrPrefWritable`.

1. Input for mapper 2 is again the user vectors: user IDs as keys, mapped to preference Vectors. Example: `98955` / `[590:1.0,22:1.0,9059:1.0]`
2. For each non-zero value in the user vector, the map function outputs item ID mapped to the user ID and preference value, wrapped in a `VectorOrPrefWritable`. Example: `590` / `[98955:1.0]`

3. The framework collects together, by item ID, the co-occurrence column and all user ID / preference value pairs.
4. The reducer collects this information into one output record and stores it.

Listing 6.5 Wrapping co-occurrence columns

```
public class CooccurrenceColumnWrapperMapper extends
    Mapper<IntWritable,VectorWritable,
           IntWritable,VectorOrPrefWritable> {
  public void map(IntWritable key,
                  VectorWritable value,
                  Context context) throws IOException, InterruptedException {
    context.write(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
    Mapper<VLongWritable,VectorWritable,
           IntWritable,VectorOrPrefWritable> {
  public void map(VLongWritable key,
                  VectorWritable value,
                  Context context) throws IOException, InterruptedException {
    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);
      context.write(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.

Technically speaking, there is no real `Reducer` following these two `Mappers`; it is no longer possible to feed the output of two `Mappers` into a `Reducer`. Instead they are run separately, and the output passed through a no-op Reducer, and saved in two locations. These two locations can be used as input to another MapReduce, whose `Mapper` does nothing as well and whose `Reducer` collects together a co-occurrence column vector for an item, and all users and preference values for that item, into a single entity called `VectorAndPrefsWritable`. `ToVectorAndPrefReducer` implements this; for brevity, this detail is omitted.

With co-occurrence columns and user preferences in hand, the algorithm proceeds by feeding into a mapper that will output the product of the column and the user's preference, for each given user ID.

1. Input to the mapper is all co-occurrence column / user records. Example: `590 / [22:3.0,95:1.0,…,9059:1.0,…]` and `590 / [98955:1.0]`
2. Mapper outputs the co-occurrence column for each associated user times the preference value. Example: `590 / [22:3.0,95:1.0,…,9059:1.0,…]`
3. The framework collects these partial products together, by user
4. The reducer unpacks this input and sums all the vectors, which gives the user's final recommendation vector (call it R). Example: `590 / [22:4.0,45:3.0,95:11.0,…,9059:1.0,…]`

Listing 6.7 Computing partial recommendation vectors

```
public class PartialMultiplyMapper extends
    Mapper<IntWritable,VectorAndPrefsWritable,
           VLongWritable,VectorWritable> {
  public void map(IntWritable key,
                  VectorAndPrefsWritable vectorAndPrefsWritable,
                  Context context) throws IOException, InterruptedException {
    int itemIndex = key.get();
    Vector cooccurrenceColumn = vectorAndPrefsWritable.getVector();
    List<Long> userIDs = vectorAndPrefsWritable.getUserIDs();

    for (int i = 0; i < userIDs.size(); i++) {
      long userID = userIDs.get(i);
      float prefValue = prefValues.get(i);
      Vector partialProduct = cooccurrenceColumn.times(prefValue); A
      context.write(new VLongWritable(userID),
                    new VectorWritable(partialProduct));
    }
  }
}
```

This Mapper actually writes a very large amount of data. For each user-item association, it outputs one copy of an entire column of the co-occurrence matrix. That is more or less necessary; those copies must be grouped together with copies of other columns in a reducer, and summed, to produce the recommendation vector.

However one optimization comes into play at this stage: a Combiner. This is like a miniature Reducer operation (in fact, Combiners extend `Reducer` as well) which is run on map output, while it's still in memory, in an attempt to combine several of the output records into one before writing them out. This saves I/O, and is not always possible. Here, it very much is; when outputting two vectors A and B for a user, it's just as fine to output one vector, A+B, for that user -- they're all going to be summed in the end.

Listing 6.8 Combiner for partial products

```
public class AggregateCombiner extends
    Reducer<VLongWritable,VectorWritable,
            VLongWritable,VectorWritable> {
  public void reduce(VLongWritable key,
                     Iterable<VectorWritable> values,
                     Context context)
      throws IOException, InterruptedException {
    Vector partial = null;
    for (VectorWritable vectorWritable : values) {
      partial = partial == null ? vectorWritable.get() :
partial.plus(vectorWritable.get());
    }
    context.write(key, new VectorWritable(partial));
  }
}
```

Listing 6.8 shows such a Combiner, which operates on the output of `PartialMultiplyMapper`. How much I/O it will save depends on how large the map spill output buffer is, and how frequently columns for one user show up in the output of one map worker. Mahout's implementation of these jobs will attempt to reserve much more memory than usual for the mapper output by increasing the value of `io.sort.mb` to even a gigabyte.

6.3.6 Translating to MapReduce: Making recommendations

At last, the pieces of the recommendation vector must be assembled for each user so that the algorithm can make recommendations. Listing 6.9 shows this in action.

Listing 6.9 Producing recommendations from vector

```
public class AggregateAndRecommendReducer extends
    Reducer<VLongWritable,VectorWritable,
```

```
          VLongWritable,RecommendedItemsWritable> {
  public void reduce(VLongWritable key,
                     Iterable<VectorWritable> values,
                     Context context)
      throws IOException, InterruptedException {

    Vector recommendationVector = null;
    for (VectorWritable vectorWritable : values) {
      recommendationVector = recommendationVector == null ?
          vectorWritable.get() :
          recommendationVector.plus(vectorWritable.get()); A
    }

    Queue<RecommendedItem> topItems = new PriorityQueue<RecommendedItem>(
          recommendationsPerUser + 1,
          Collections.reverseOrder(
            ByValueRecommendedItemComparator.getInstance()));

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

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

A Build the recommendation vector by summing
B Find the top N 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    [103:24.5,102:18.5,106:16.5]
```

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.

At last, we have ended up with recommendations, having started with raw input from the Wikipedia data set. As you can see, even this simple recommender implementation looks quite different when designed and implemented on a distributed system like Hadoop. Even in simplified form it consists of five stages, each performing a piece of the computation or transformation along the way. The natural next step is to actually run all of this, to better understand how such an implementation is put to work in practice.

6.4 Running MapReduces with Hadoop

Now it's time 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, you will start by running a Hadoop computation on a cluster of one machine: yours. It is far simpler to set up a Hadoop cluster on your local machine for learning rather than set up a proper cluster. However, it will not be very different to run on a real cluster. First, you must set up a pseudo-distributed Hadoop cluster on your local machine, and then you will be able to experiment with running these and other MapReduces in Mahout.

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 0.20.2 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 1 gigabyte 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. You 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 you are running on only one machine and incurring all the overhead of the distributed computing framework. You 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 the algorithm 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 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 above.

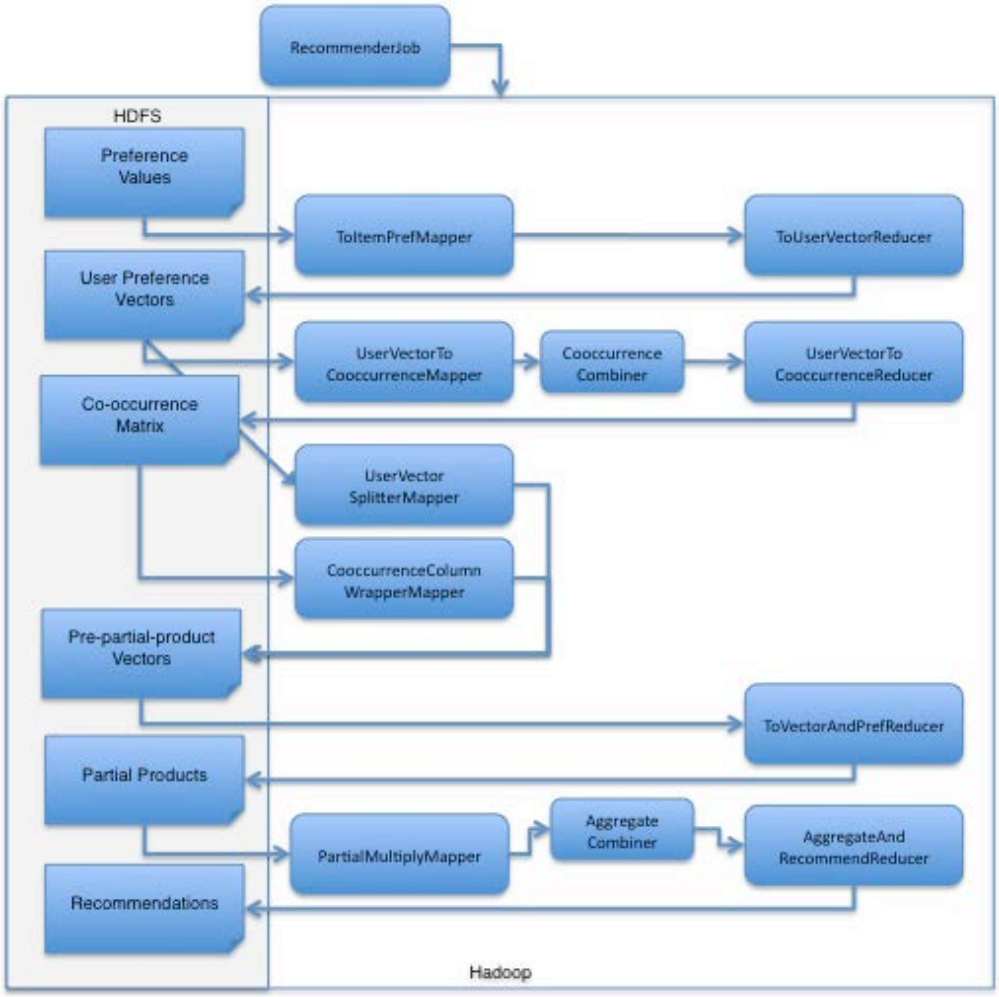


Figure 6.2 The relation between RecommenderJob, the MapReduces it invokes, and the data that they read to and write from HDFS

In order to run it, and allow Hadoop to run these jobs, you 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-0.4-SNAPSHOT.job`, which is in reality a `.jar` file. (The exact name will change with the version number of Mahout.)

Now, kick it all off with:

```
bin/hadoop jar target/mahout-core-0.4-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 marshaling the results, is non-trivial.

In fact, running recommendations for all 5.7 million or so "users" in the data set took almost 700 hours of CPU core time on a local one-machine cluster, or about half a second per recommendation. Running time is dominated by the final phase, where vectors are multiplied and added. That's more than with non-distributed recommenders, at somewhat smaller scales, but not orders of magnitude worse. At

current prices for virtual instance time, this would cost about $0.01 per 1,000 users' worth of recommendations. Of course, it need not take 700 hours to compute all these recommendation; if employing 100 worker machines, the entire process would take somewhat more than 7 hours of actual time to complete.

Back to your local instance, if you are patient enough to let it complete recommendations for just one user, you should find the results on HDFS under the `output/` directory. It will be contained in a single file called `part-r-00000`.

Copy the result back to your local file system with "`bin/hadoop fs -get output/part-r-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, you let Hadoop default to running just one map and one reduce worker at once. This is appropriate since the algorithm was 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.

You have now successfully designed and implemented a distributed algorithm on Hadoop, then set up a Hadoop cluster and run your implementation on Hadoop. The good news is that this constitutes most of the learning curve; interacting with a real Hadoop cluster is virtually the same. You are now well prepared for encounters with a number of other MapReduce-based and Hadoop-based machine learning algorithms in Mahout, which will be presented in coming chapters.

Before closing out coverage of recommenders, it's useful to examine one more Hadoop-based recommender algorithm offered by Mahout, which is a sort of hybrid between the distributed and non-distributed approaches you have seen so far.

6.5 Pseudo-distributing a Recommender

Earlier, you saw how to create, test and operate a variety of non-distributed recommender engines with Mahout, on one machine. This chapter demonstrated 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, 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.

There are no new algorithms or code to introduce here; the pseudo-distributed recommender engine framework in Mahout runs the existing `Recommenders`, 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 an input 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`.

The framework needs 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-0.4-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.

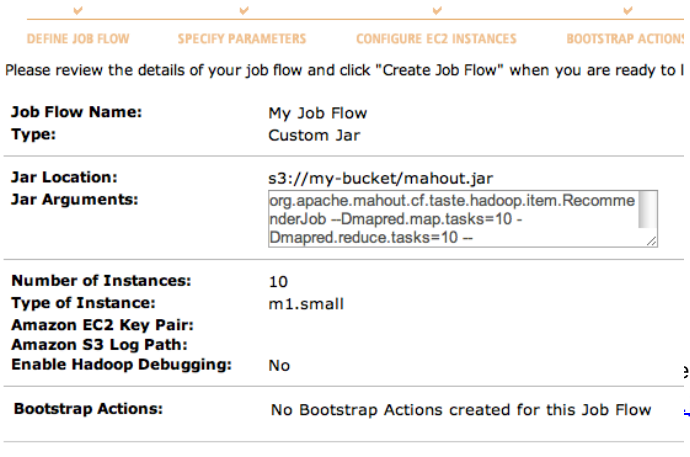
6.6 Looking beyond first steps with recommendations

The distributed portion of Mahout's recommendation engine introduced in this chapter is still quite under construction, so refer to the latest documentation and code in conjunction with this book. Look to Mahout to provide more options as it evolves. It is a work in progress from a growing community, which aims to continually improve and expand our understanding of and ability to deploy recommenders and machine learning.

Before moving on from recommenders to clustering and classifier algorithms, we conclude with a collection of brief, parting 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.



ections to the Author Online forum:

.jspa?forumID=623

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 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-0.4-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, start with a number equal to the number of virtual cores you reserve for the computation. While any instance type can be used, start 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.

6.6.2 Imagining unconventional uses of recommendations

Our discussion of recommenders is drawing to a close. Before moving on to clustering, it is worth changing gears to provide some parting thoughts on how recommenders might apply to your projects.

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. You 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 will explain this and other variations in detail.

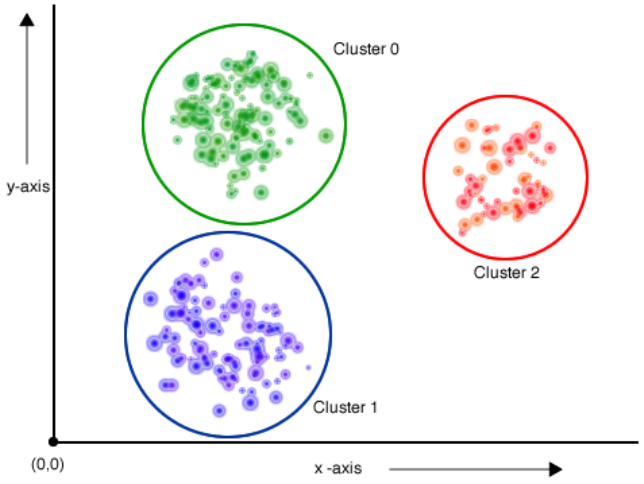


Figure 7.1 Points in an x-y plane. Circles represent the clusters. The points on the plane could be viewed as 3 logical groups. Clustering algorithms helps surface those groups to you.

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. Based on that "Harry Potter: The Philosopher's Stone" and "Harry Potter: The Prisoner of Azkaban" have three words in common: "Harry", "Potter" and "The". 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 at all. We need to 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 overlapping words, judge the books similar.

Unfortunately, that is easier said than done. Not only do these books have hundreds of pages of text, but features of English also confound this sort of measure. 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 their 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, the similarity of word counts of a particular word 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. Weighting words in free form text is a science on its own. There are many tricks like the ones we discussed above to improve the quality of word weighting. These and many other tricks are part of a popular weighting method called Tf-Idf (term frequency - inverse document frequency). We will cover Tf-Idf and its variations 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. For the hello world example, we will run the K-Means algorithm. In upcoming chapters, we will review other 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`, a format used extensively by the Hadoop library. The input encodes `Vectors`, each of which represents one point.

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 clubbed into one region of the plane, and other contains four points in another region.

We input the data for Mahout clustering in three steps – first, we need to preprocess the data. We then use the preprocessed data and create 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 need to convert them to a `Vector` class. The first step is to convert this input to a Mahout Vector.

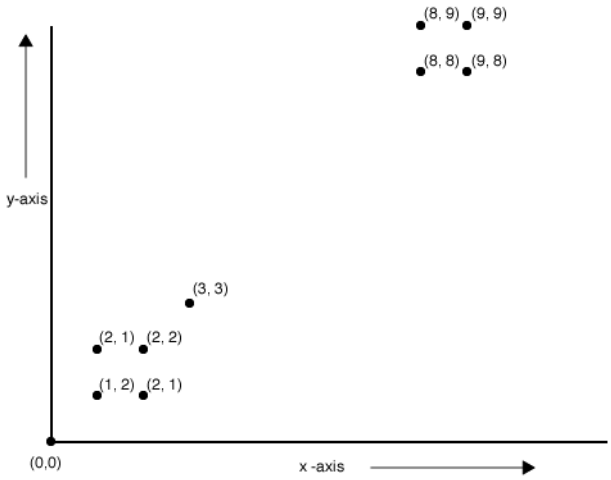


Figure 7.2 Plot of the input points in the x-y plane

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 the following sub-section, 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 this 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 this 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.

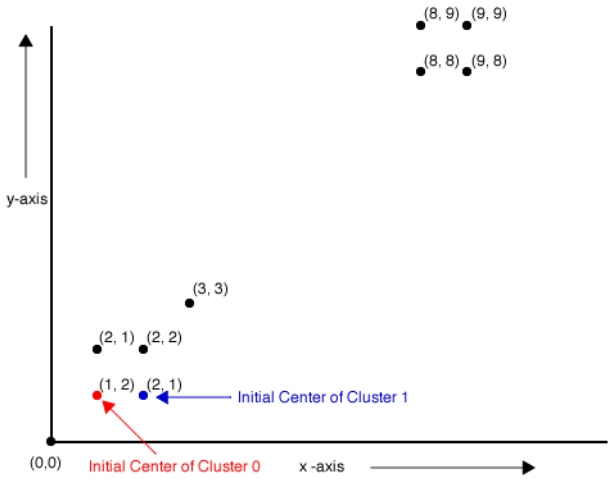


Figure 7.3 Marking the initial clusters is an important step in k-means clustering.

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. Unfortunately, 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 were 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(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);                       #2

    File testData = new File("testdata");
    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);   #3

    Path path = new Path("testdata/clusters/part-00000");
    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, new EuclideanDistanceMeasure());
      writer.append(new Text(cluster.getIdentifier()), cluster);
    }
    writer.close();

    KMeansDriver.run(conf, new Path("testdata/points"),             #4
      new Path("testdata/clusters"),
      new Path("output"), new EuclideanDistanceMeasure(), 0.001, 10,
      true, false);

    SequenceFile.Reader reader = new SequenceFile.Reader(fs,
        new Path("output/" + Cluster.CLUSTERED_POINTS_DIR
                 + "/part-m-00000"), conf);

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

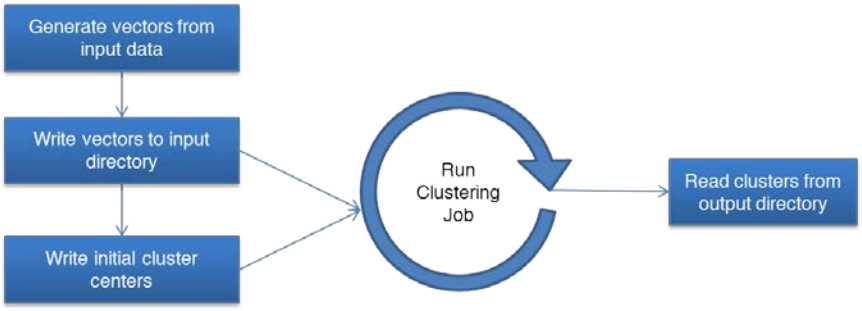


Figure 7.4 Flow of the hello world example for clustering.

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. 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 good accuracy. 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.

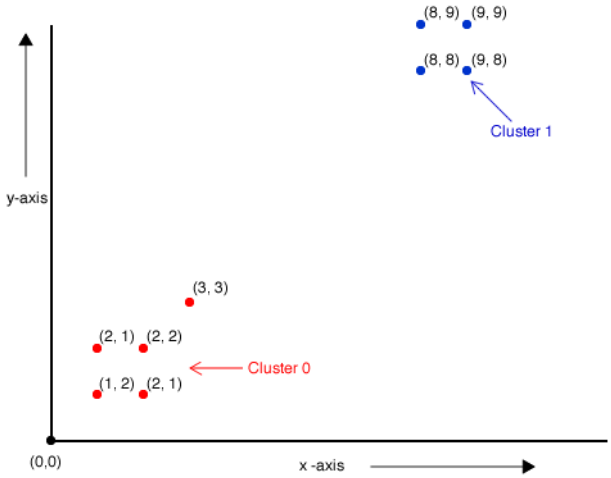


Figure 7.5 The output of our hello world k-means clustering program. Even with distant centers, K-means algorithm was able to correctly iterate and correct the center based on Euclidean distance measure.

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 (a_1, a_2, … , a_n) and B (b_1, b_2, … ,b_n) is:

$$d = \sqrt{((a_1-b_1)^2 + (a_2-b_2)^2 + \ldots + (a_n-b_n)^2)}$$

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 (a_1, a_2, … , a_n) and (b_1, b_2, … ,b_n) the distance becomes:

$$d = (a_1-b_1)^2 + (a_2-b_2)^2 + \ldots + (a_n-b_n)^2$$

The Mahout class that implements this measure is SquaredEuclideanDistanceMeasure.

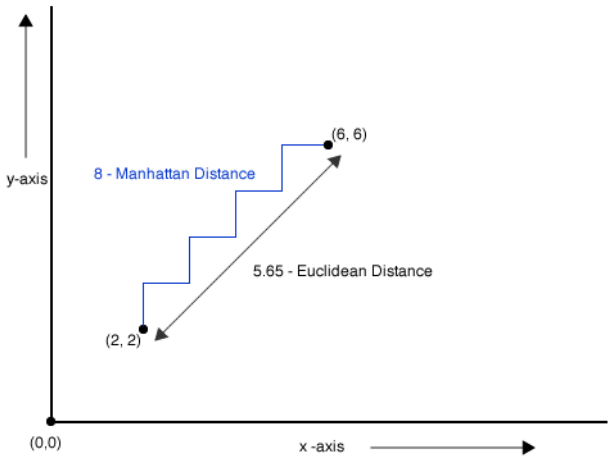


Figure 7.6 Difference between Euclidean and Manhattan distance measure. Euclidean distance measure gives `5.65` as the distance between `(2, 2)` and `(6, 6)` where as Manhattan distance is `8.0`

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 (a_1, a_2, … , a_n) and (b_1, b_2, … , b_n) is:

$$d = (a_1-b_1) + (a_2-b_2) + \ldots + (a_n-b_n)$$

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.

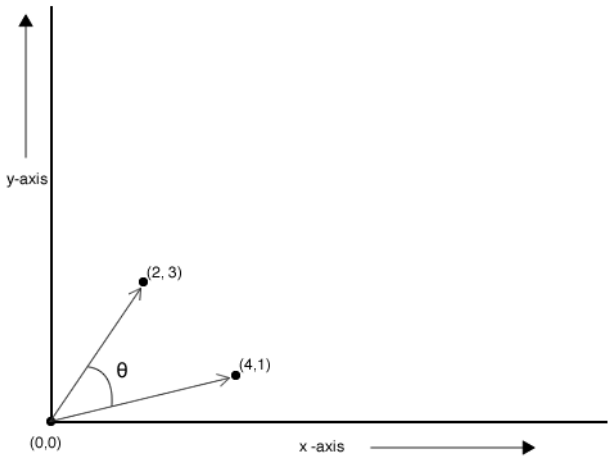


Figure 7.7 Cosine angle between the vectors (2,3) and (4, 1) as calculated from the origin

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 (a_1, a_2, … , a_n) and (b_1, b_2, … ,b_n) is:

$$d = 1 - (a_1b_1 + a_2b_2 + \ldots + a_nb_n) / (\sqrt{(a_1^2 + a_2^2 + \ldots + a_n^2)}\sqrt{(b_1^2 + b_2^2 + \ldots + b_n^2)})$$

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 (a_1, a_2, … , a_n) and (b_1, b_2, … , b_n) is:

$$p = (a_1b_1 + a_2b_2 + \ldots + a_nb_n)$$
$$d = 1 - p / (\sqrt{(a_1^2 + a_2^2 + \ldots + a_n^2)} + \sqrt{(b_1^2 + b_2^2 + \ldots + b_n^2)} - p)$$

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 0^{th} index (for x) and `1.0` the 1^{st} 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.

Table 7.1 Result of clustering using various distance measures

Distance Measure	Number of iterations	Vectors[11] 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
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

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 small 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.

[11] These are the names/ids of the vectors as given in the Clustering Hello World source code

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, and number of legs etc. 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. How do we represent the vectors in Mahout? How do we get from objects to vectors in the first place? we will see answers to these questions in the following 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.

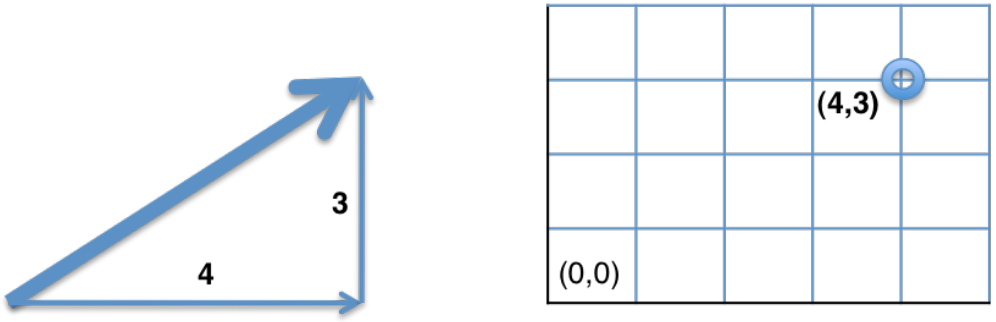


Figure 8.1 In physics, the vector can be thought of ray with a start point, direction and length and represents quantities like velocity and acceleration. In geometry or space, the vector is just a point denoted by weights along each dimension. The direction and magnitude of the vector is by default assumed to be a ray from the origin (0,0).

8.1.1 Transforming data into Vectors

In Mahout, vectors are implemented as three different classes, each optimized for different scenarios. These `Vector` implementation classes are `DenseVector`, `RandomAccessSparseVector`, and `SequentialAccessSparseVector`. Appendix B explains the differences between these clearly.

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 performs faster than the random-access sparse vector.

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. The vector of a small round red apple looks like `[0 => 100 gram, 1 => red, 2 => small]`. This "vector" however 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.

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

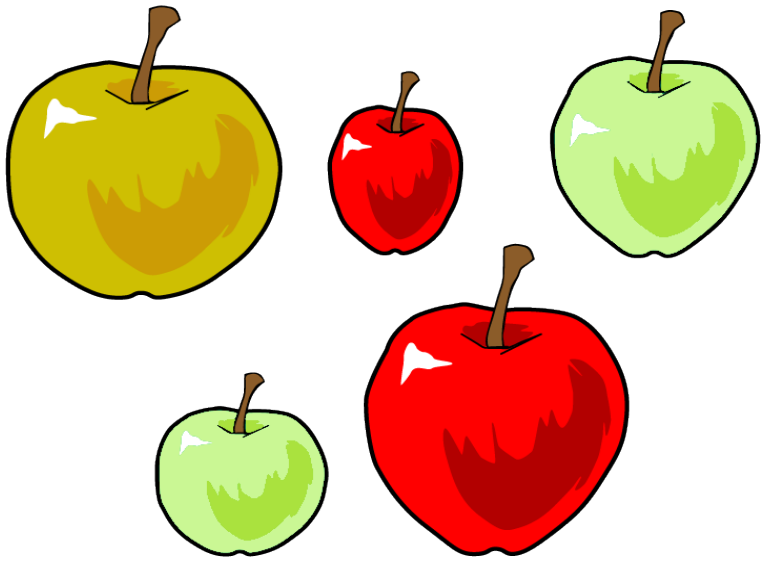


Figure 8.1 Apples of different sizes and colors needs to be converted into an appropriate vector form. The trick is to figure out how the different features of the apples translate into a decimal value.

Table 8.1 Set of apples of different weight, sizes and colors converted to vectors

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]

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 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[] {0.23, 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");


  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);
    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();                B
  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 serializes 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. 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 w_1, w_2, ..., w_n with frequency f_1, f_2, ..., f_n. The term frequency (TF_i) of a word w_i is the frequency f_i.

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 IDF_i for a word w_i 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:

$$IDF_i = N / DF_i$$

Therefore the weight of a word in a document vector is:

$$w_i = TF_i * IDF_i = TF_i * N / DF_i$$

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:

$$IDF_i = \log (N / DF_i)$$

Thus the TF-IDF weight for a word w_i becomes:

$$w_i = TF_i * \log (N / DF_i)$$

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 completely 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 because 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[12]. It is a widely used dataset for machine learning research. The data was originally collected and labeled by Carnegie Group, 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`.

[12] http://www.daviddlewis.com/resources/testcollections/reuters21578/

Before doing this, download the Reuters collection from the website[13] 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/
```

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.

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

Option	Flag	Description	Default Value
Overwrite	-w	If set, the output folder is overwritten. If not set,	N/A

[13] http://www.daviddlewis.com/resources/testcollections/reuters21578/reuters21578.tar.gz

(bool)		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.	
Lucene Analyzer name (String)	-a	The class name of the analyzer to use	org.apache.lucene.analysis.standard.StandardAnalyzer
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 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 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 flag 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 L_p 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	1

		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	
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 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 up to 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 isn't uncommon that a few documents pop up as if they are similar to all the other documents in the collection. 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:

$$[x/\sqrt{(x^2 + y^2 + z^2)}, y/\sqrt{(x^2 + y^2 + z^2)}, z/\sqrt{(x^2 + y^2 + z^2)}]$$

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
- Fuzzy K-Means clustering, Dirichlet clustering
- Topic modeling using LDA as a variant of clustering

Now that we know how input data is represented as `Vectors` and how `SequenceFiles` 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 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[14]. It's widely used as a clustering 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 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 to 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

[14] Original Paper: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.131.1338

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 us look at the 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)[15]. 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[16].

[15] http://en.wikipedia.org/wiki/Expectation-maximization_algorithm
[16] http://www.cc.gatech.edu/~dellaert/em-paper.pdf, Gives an easier explanation on EM Algorithm in terms of lower bound maximization.

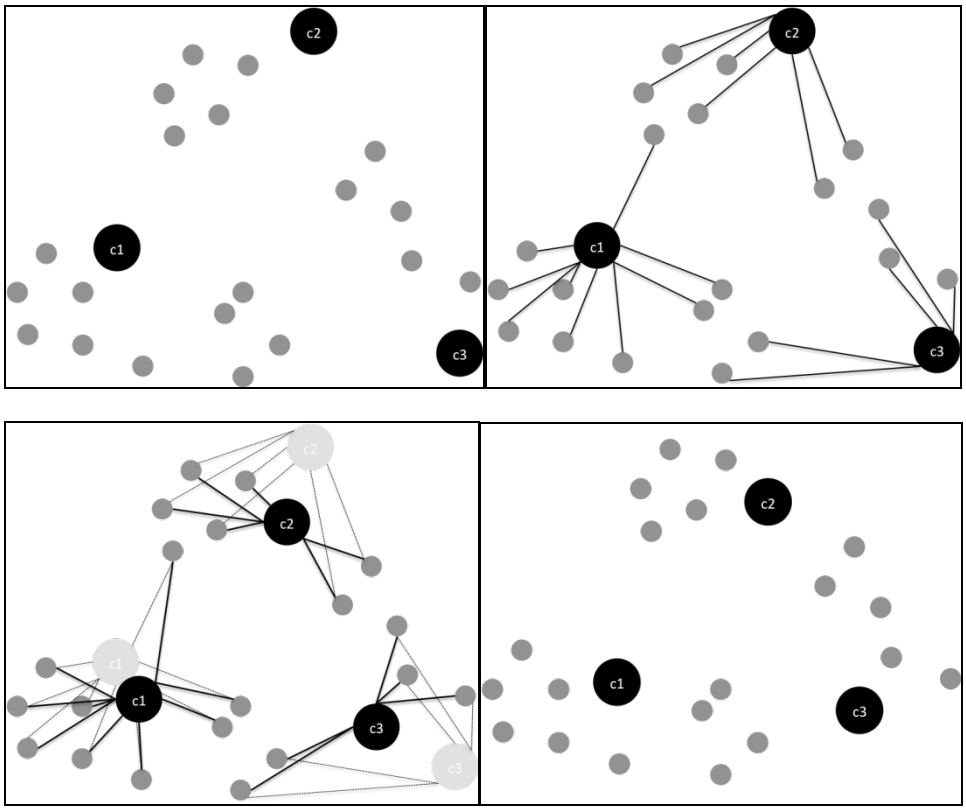


Figure 9.1 K-Means clustering in action. Starting with 3 random points as centroids (top-left), the Map stage (top-right) assigns each point to the cluster nearest to it. In the reduce stage (bottom-left), the associated points are averaged out to produce the new location of the centroid, leaving us 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 MapReduce 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 an Apache 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
```

Listing 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
     }
   }
```

#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.

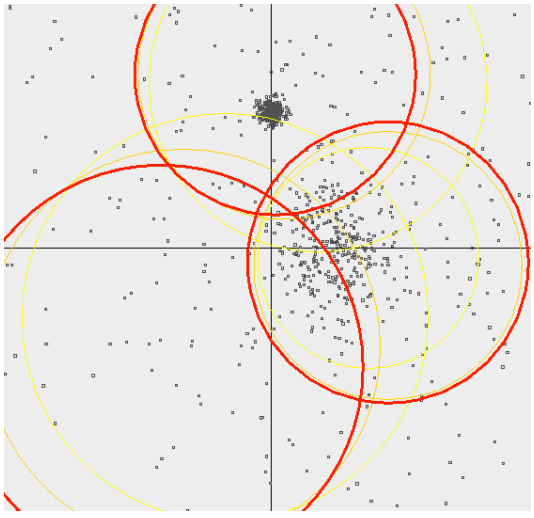


Figure 9.2 K-Means Clustering. We start with k as 3 and try to cluster 3 normal distributions we have generated. The thin lines denote the clusters estimated in previous iterations, here we can clearly see the clusters shifting

The DisplayKMeans class in Mahout's mahout-examples module 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 plus the k centers. If the data is huge, we cannot run this implementation.

This is where MapReduce shines. Using MapReduce 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 MapReduce 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 MAPREDUCE JOB

In Mahout, the MapReduce 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 MapReduce 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 MapReduce 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 MapReduce 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 MapReduce 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 MapReduce 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 MapReduce version and think of moving the computation to a Hadoop cluster. Check out 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 MapReduce 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 program 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 more than 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 documents 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 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 distance 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 over 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 one such powerful technique that will takes us closer to it.

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.

This sounds exciting, but, the algorithms 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 one such technique to select k as well as the centroid vectors for K-Means – canopy generation.

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 T_1 and T_2, with $T_1 > T_2$.

Canopy clustering strength lies in its ability to create clusters extremely quickly. 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 T_1 and T_2, with $T_1 > T_2$. 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 T_1, it is added into that canopy. If the distance is within T_2, it is removed from the list and thereby prevented from forming a new canopy in the subsequent loops. It repeats this process until the list is empty.

It prevents all points close to an already existing canopy (distance < T_2) from being the center of a new canopy. It's detrimental to form 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 MapReduce 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.

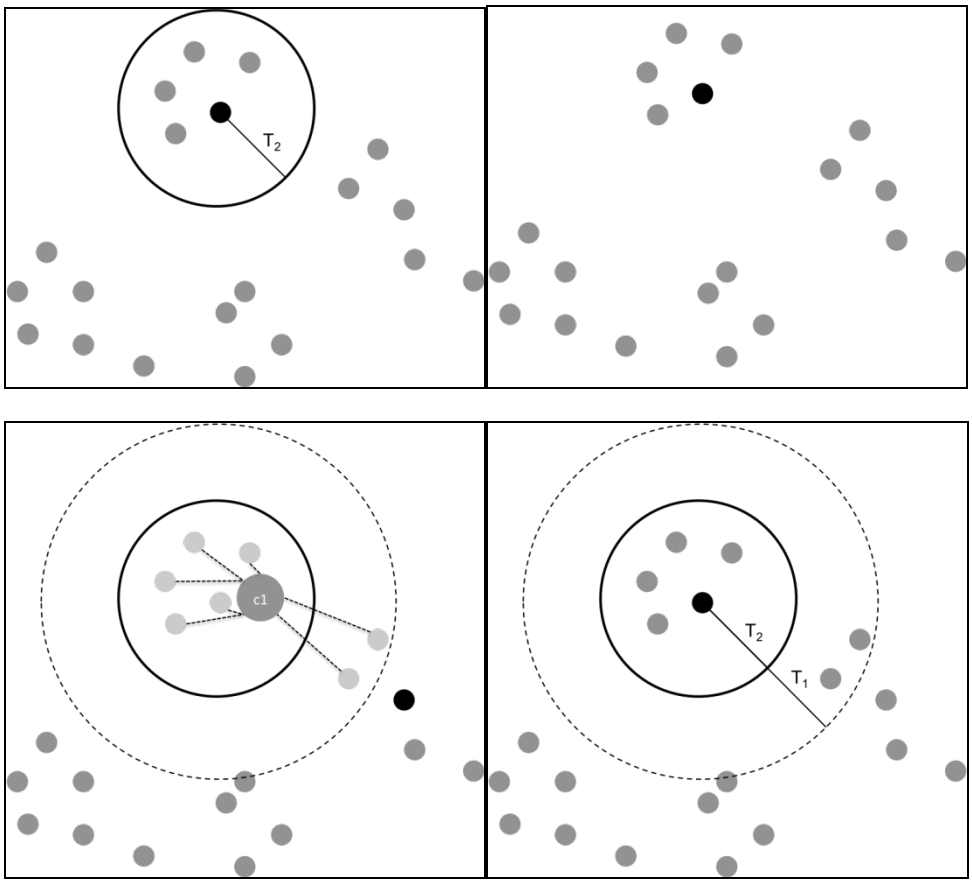


Figure 9.3 Canopy Clustering. If we start with a point (top left) and mark it as part of a canopy, then all the points within a distance T_2 (top right) are removed from the dataset and prevented from becoming a new canopy. The points within outer circle (bottom-right) are also put in the same canopy 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
```

-

Listing 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
     }
   }
```

Cueball

#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.

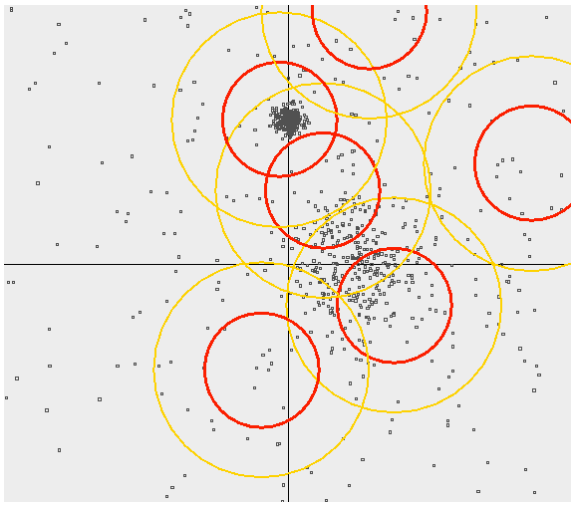


Figure 9.4 An example of in-memory Canopy Generation visualized using the DisplayCanopy class. We start with T_1=3.0 and T_2=1.5 and try to cluster the 3 normal distributions that are synthetically generated.

The `DisplayCanopy` class in Mahout's `mahout-examples` module 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 does not require the number of cluster centroids as a parameter. The number of centroids formed depends only on the choice of distance measure, T_1 and T_2. The canopy in-memory clustering implementation works with list of `Vector` objects just like the K-Means implementation. If the dataset is huge, it's not possible to run this algorithm on a single machine and the MapReduce job would be required. The MapReduce 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

The next example generates 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, it's possible to experiment with various parameters and are able to see the results much quicker than if 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" and "said" 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 next case study, 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 memory usage in the Mapper and hence might exceed available memory while running on a large dataset with a bad set of thresholds. The parameters need to be tuned to fit the dataset and the clustering problem, as we will see next.

The next example creates 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.

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.

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.

Listing 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(new Path(outputDir));
    Path tokenizedPath = new Path(outputDir,
        DocumentProcessor.TOKENIZED_DOCUMENT_OUTPUT_FOLDER);
    MyAnalyzer analyzer = new MyAnalyzer();                    #2
    DocumentProcessor.tokenizeDocuments(new Path(inputDir),
      analyzer.getClass().asSubclass(Analyzer.class),
      tokenizedPath);                                          #3

    DictionaryVectorizer.createTermFrequencyVectors(tokenizedPath,
      new Path(outputDir), conf, minSupport, maxNGramSize, minLLRValue,
      2, true, reduceTasks,
      chunkSize, sequentialAccessOutput, false);
    TFIDFConverter.processTfIdf(
      new Path(outputDir ,
     DictionaryVectorizer.DOCUMENT_VECTOR_OUTPUT_FOLDER),
      new Path(outputDir), chunkSize, minDf,
      maxDFPercent, norm, true, sequentialAccessOutput, false,
     reduceTasks);                                           #4
    Path vectorsFolder = new Path(outputDir, "tfidf-vectors");
    Path canopyCentroids = new Path(outputDir , "canopy-centroids");
    Path clusterOutput = new Path(outputDir , "clusters");

    CanopyDriver.run(vectorsFolder, canopyCentroids,
      new EuclideanDistanceMeasure(), 250, 120, false, false); #5
    KMeansDriver.run(conf, vectorsFolder,
      new Path(canopyCentroids, "clusters-0"),
      clusterOutput, new TanimotoDistanceMeasure(), 0.01,
      20, true, false);                                        #6

    SequenceFile.Reader reader = new SequenceFile.Reader(fs,
        new Path(clusterOutput
            + Cluster.CLUSTERED_POINTS_DIR + "/part-00000"), conf);

    IntWritable key = new IntWritable();
```

```
    WeightedVectorWritable value = new WeightedVectorWritable();
    while (reader.next(key, value)) {                          #7
       System.out.println(key.toString() + " belongs to cluster "
       + value.toString());
    }
    reader.close();
  }
}
```

Cueball

#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

Listing 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()));
  }
}
```

Cueball

#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 alphabetic characters. 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 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:

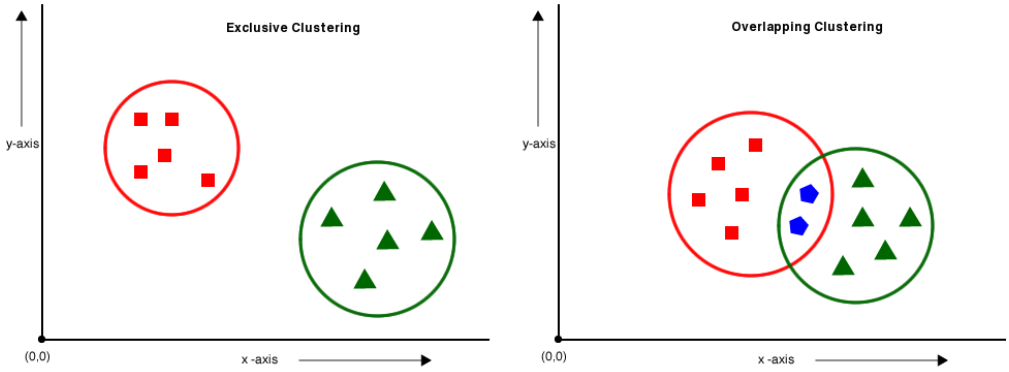


Figure 9.6 Exclusive clustering versus overlapping clustering with two centers. In the former, squares and triangles have their own cluster and each one belong only to one cluster. While in overlapping clustering, some shapes like pentagon can belong to both the clusters with some probability so they 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. The "fiction" and "fantasy" clusters have a parent-child relation, 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.

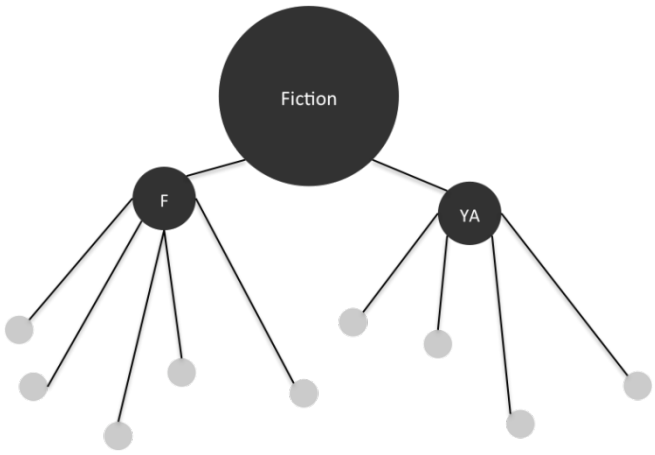


Figure 9.7 Hierarchical clustering. A bigger cluster groups two or more smaller clusters in the form of a tree like hierarchy. Recall the librarian example in chapter 7. We did a crude form of hierarchical clustering when we simply stacked books based the similarity we felt when we read them.

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.

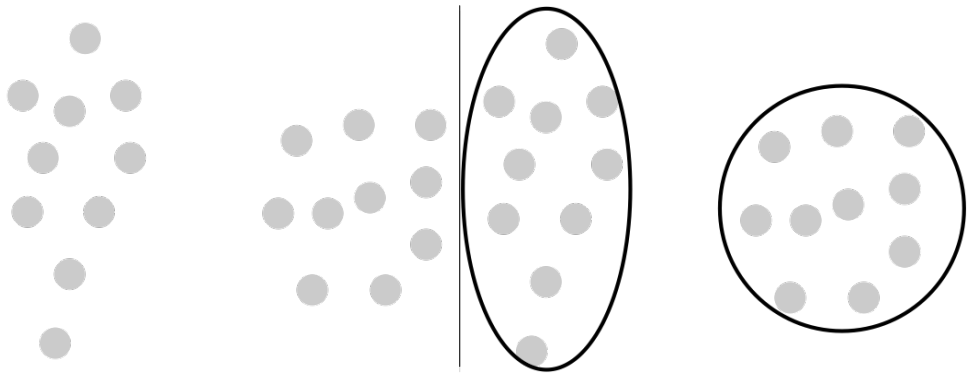


Figure 9.8 A simplified view of probabilistic clustering. The initial set of points (left). On the right: the first set of points matches an elongated elliptical model where as the second one is more symmetric.

To explain how this fitting happens, let's look at a 2-dimensional example in Figure 9.8. Say that 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 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.

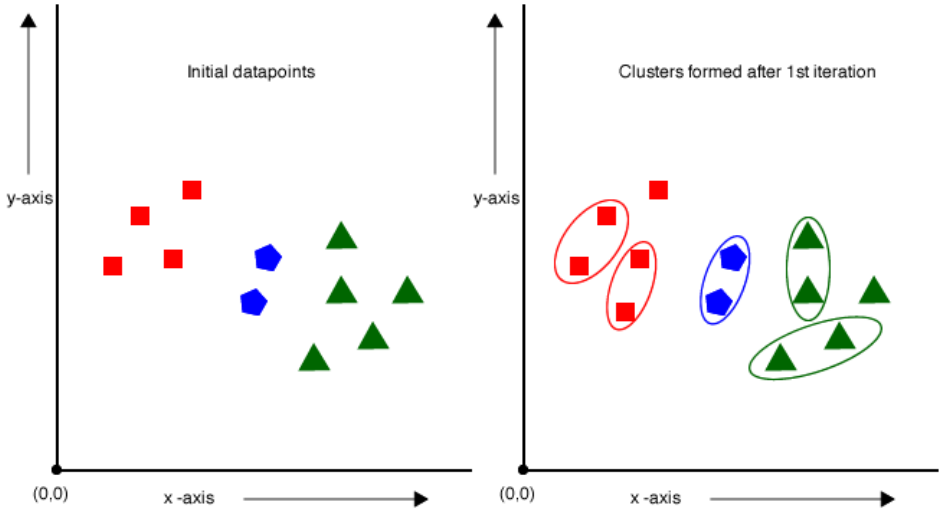


Figure 9.9 Bottom up clustering approach. After every iteration, the clusters are merged to produce larger and larger clusters till it is infeasible to merge based on the given distance measure.

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.

Though this is straightforward, finding the best possible split for a set of n-dimensional points is not 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.

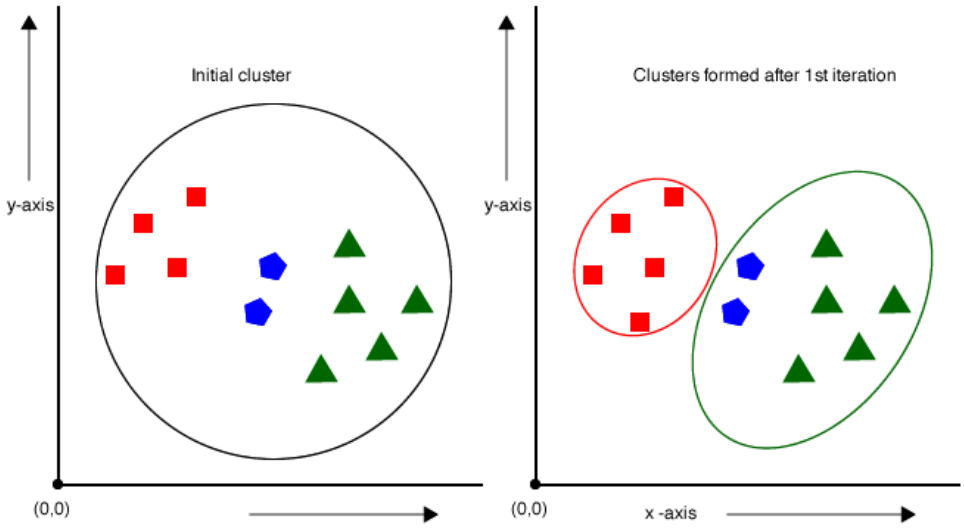


Figure 9.10 Top down clustering approach. During each iteration, the clusters are divided into two by finding the best splitting till we get the clusters we desire.

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 the distribution of the points is unclear, 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 MapReduce jobs without specifying the number of clusters as explained below.

The lack of hierarchical clustering algorithms in MapReduce 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) whereas 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 MapReduce. 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. Listing 9.6 shows 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

Listing 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.

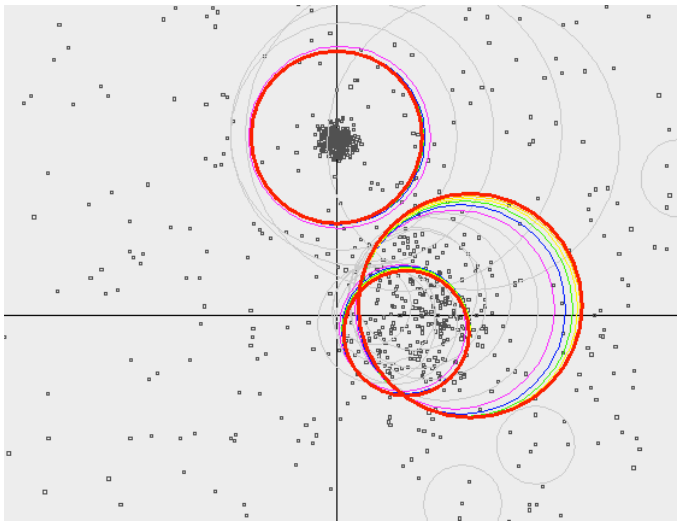


Figure 9.11 Fuzzy K-Means clustering. The clusters look like they are overlapping each other and the degree of overlap is decided by the fuzziness parameter.

The `DisplayFuzzyKMeans` class in Mahout's `mahout-examples` module 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.

MapReduce Implementation of Fuzzy K-Means

Before running the MapReduce 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 -cd 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 d_1, d_2, … , d_k are the distances towards each of the k cluster centroids. The degree of association (u_1) of vector (V) to the first cluster (C_1) is calculated as

$$u_1 = 1/((d_1/d_1)^{(2/(m-1))} + (d_1/d_2)^{(2/(m-1))} + \ldots + (d_1/d_k)^{(2/(m-1))})$$

Similarly, we can calculate the degree of belonging to other clusters by replacing d_1 in the numerators of the denominator expression by d_2, d_3 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.

Listing 9.7 News clustering using Fuzzy K-Means clustering

```
public class NewsFuzzyKMeansClustering {
  public static void main(String args[]) throws Exception {
    …

    String vectorsFolder = outputDir + "/tfidf-vectors";
    String canopyCentroids = outputDir + "/canopy-centroids";
    String clusterOutput = outputDir + "/clusters/";

    CanopyDriver.run(conf, new Path(vectorsFolder), new Path(
        canopyCentroids), new ManhattanDistanceMeasure(), 3000.0,
      2000.0, false, false);                          #1

    FuzzyKMeansDriver.run(conf, new Path(vectorsFolder), new Path(
        canopyCentroids, "clusters-0"), new Path(clusterOutput),
      new TanimotoDistanceMeasure(), 0.01, 20, 2.0f, true, true, 0.0,
      false);                                         #2

    SequenceFile.Reader reader = new SequenceFile.Reader(fs,
        new Path(clusterOutput + Cluster.CLUSTERED_POINTS_DIR
                 + "/part-m-00000"), conf);

    IntWritable key = new IntWritable();
    WeightedVectorWritable value = new WeightedVectorWritable();
    while (reader.next(key, value)) {             #3
      System.out.println("Cluster: " + key.toString() + " "
                         + value.getVector().asFormatString());
      // 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 refine the 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.

9.4 Model-based Clustering

The complexities of clustering algorithms have increased progressively in this chapter. We started with K-Means, a 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. 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.

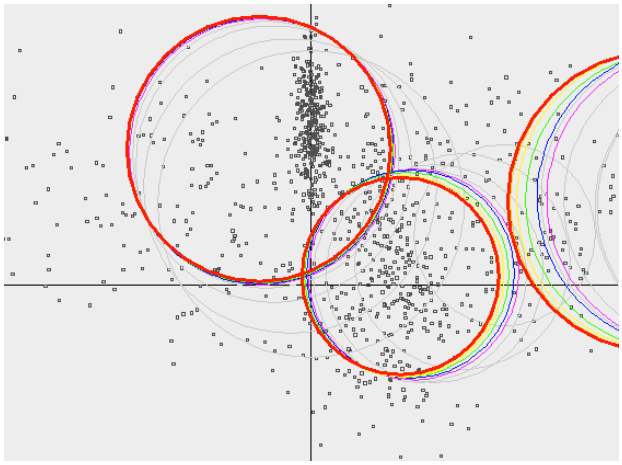


Figure 9.12 Running K-Means clustering over asymmetric normal distribution of points. The points are scattered in an oval -shaped area instead of a circular one. Clearly K-Means is not a perfect fit for the data and these clusters don't make any.

Another issue with K-Means is that is requires estimation of k, the number of seed centroids. It is usually overestimated. Finding the optimal value of k is not easy unless there is a clear idea about the data, which happens rarely. Even by doing canopy generation, 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 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 clustering

Mahout has a model-based clustering algorithm, Dirichlet clustering. The word "Dirichlet" refers to a family of probability distributions defined by a German mathematician Johann Peter Gustav Lejeune Dirichlet. The Dirichlet clustering performs something known as mixture modeling using calculations based on the Dirichlet distribution. The whole process might sound complicated without a deeper understanding of Dirichlet distributions, but the idea is simple.

Say we know 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 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.

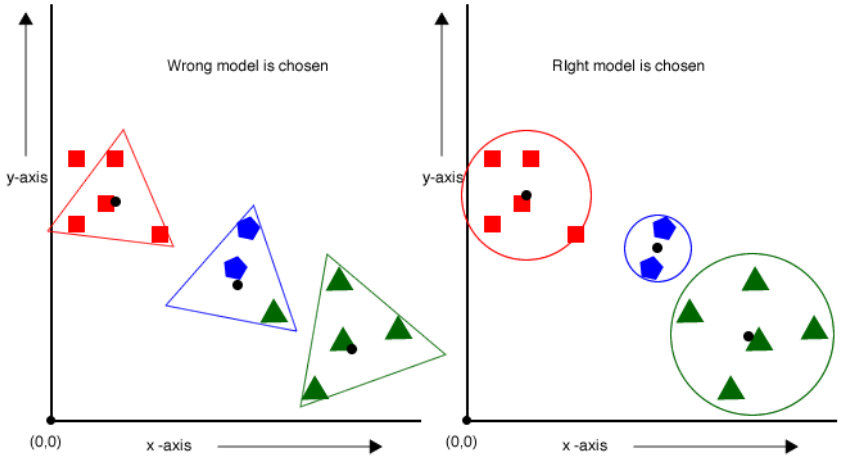


Figure 9.13 Dirichlet clustering. The models are made to fit the given dataset as best as it can to describe it. The right model will fit the data better and tells the number of clusters in the dataset which correlates with the model.

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 clustering implementation in Mahout works

UNDERSTANDING DIRICHLET CLUSTERING ALGORITHM

Dirichlet 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 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 MapReduce 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 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.

Listing 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
```

Cueball

#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.

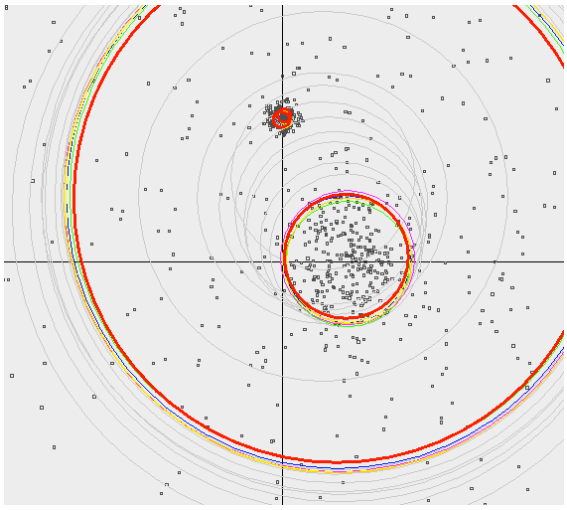


Figure 9.14 Dirichlet clustering with Normal Distribution using the DisplayNDirichlet class in Mahout examples folder

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 purpose of these parameters is that many iterations are needed 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. 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 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 clustering using the asymmetric normal model on the set of 2-dimensional points that has different standard deviation along the x and y directions. The output of the clustering is shown in Figure 9.15.

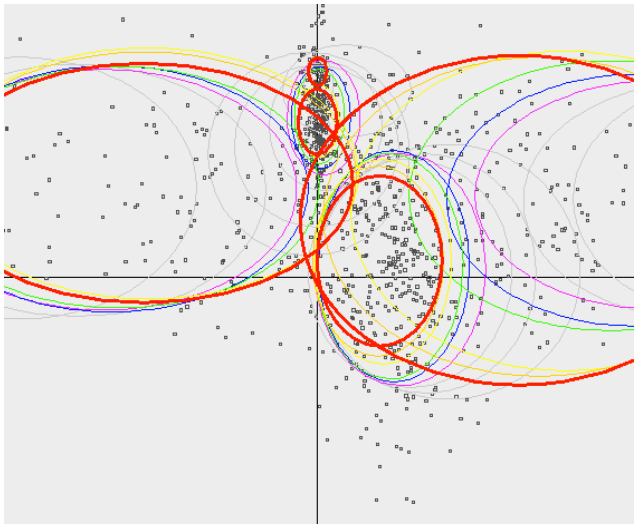


Figure 9.15 Dirichlet clustering with Asymmetrical Normal Distribution using the Display2dASNDirichlet class. The thick line denotes the final state and the thin lines are the states in previous iterations

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 MapReduce version of Dirichlet clustering.

MapReduce Version of Dirichlet Clustering

Like other implementations, in Mahout, Dirichlet clustering is also focused on scaling with huge datasets. The MapReduce 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 MapReduce job:

- The Reuters dataset in the `Vector` format
- The model distribution class `–md` defaults to `NormalModelDistribution`
- The model distribution prototype `Vector` class. The class that becomes the type for all vectors created in the job `–mp` defaults to `SequentialAccessSparseVector`
- The `alpha0` value for the distribution, `-a0 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 `dirichlet` as follows:

```
bin/mahout dirichlet
-i examples/reuters-vectors/
-o reuters-dirichlet-clusters -k 60 -x 10 –a0 1.0
-md org.apache.mahout.clustering.dirichlet.models.NormalModelDistribution
-mp org.apache.mahout.math.SequentialAccessSparseVector
```

After each iteration of Dirichlet 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 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 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 clustering is more of a data understanding tool while being a great data clustering one.

9.5 Topic Modeling using Latent Dirichlet Allocation (LDA)

Until 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, “What was that news article all about?” we will naturally say something like “it talked about the US war against terrorism” instead of telling him or her 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 might be “dog”, “woof”, “puppy”, “bark”, “bow”, “chase”, “loyal”, “friend”. Some of these words like “bow” and “bark” 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 “furball”.

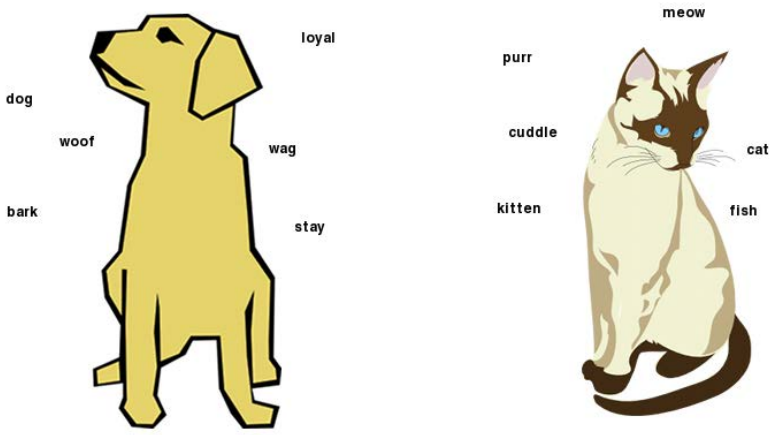


Figure 9.14 The topics “Dog” and “Cat” and the words that occur frequently in them.

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 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.

Latent Dirichlet Analysis, or 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 Latent Dirichlet Analysis

LDA is a generative model like the Dirichlet 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 chapters 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 more 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 like Dirichlet clustering. It starts with an empty topic model. It then reads all the documents in a Mapper phase 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 impact 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”, and “politics” 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 slowdown. A good rule of thumb is to choose a value that makes sense for a particular use case. Mahout’s LDA implementation is written as a MapReduce job, it can be run in large Hadoop clusters. We can speed up the algorithm by adding more workers.

The 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: the 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 maximum 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 in the `mahout-utils` module for reading the topic and word probabilities from the output state directory.

`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.

Table 9.1 Top 5 words in selected topics from LDA topic modeling of Reuters news data.

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

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" and "he" did not disappear even after pruning high frequency words using the dictionary vectorizer. Can LDA do something better?

One parameter we didn't tweak in this run was the topic smoothing parameter (`-a`). Since text data is 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.

Table 9.2 Top 5 words in selected topics from LDA topic modeling of Reuters data after increased smoothing is applied.

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

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. We could 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. Topic modeling can be used to 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.

Table 9.3 A summary of the different clustering algorithms in mahout, the entry-point classes, and their properties.

Algorithms	In-memory implementation	MapReduce implementation	Fixed clusters	Partial membership
K-Means	KMeansClusterer	KMeansDriver	Y	N
Canopy	CanopyClusterer	CanopyDriver	N	N
Fuzzy K-Means	FuzzyKMeansClusterer	FuzzyKMeansDriver	Y	Y
Dirichlet	DirichletClusterer	DirichletDriver	N	Y
LDA	N/A	LDADriver	Y	Y

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 algorithm did fast and approximate clustering of the data and estimated the centroids 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 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 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, we will learn about some lesser known tools and techniques present in Mahout to help understand and improve the quality of clustering.

10 Evaluating clustering quality

This chapter covers:

- Inspecting clustering output
- Evaluating the quality of clustering
- Improving clustering quality

We saw many types of clustering algorithms in the last chapter: K-Means, Canopy, K-Means++, Fuzzy K-Means, Dirichlet, and Latent Dirichlet Analysis. They all performed well on certain types of data and sometimes poorly on others. The most natural question that comes to our mind after every clustering job is, "how well did the algorithm perform on my data?" Analyzing the output of clustering is an important exercise. This may be done with simple command-line tools or richer GUI-based visualizations. Once the clusters are visualized and problem areas are identified, these results may be formalized into quality measures, which give numeric values that tell how good the clusters are. In this chapter we'll look at several ways to inspect, evaluate, and improve our clustering algorithms.

Tuning a clustering problem typically reduces to creating a custom similarity metric and choosing the right algorithm. The evaluation measure shows the impact of changes in the distance measure to that of clustering quality. First, we'll look at the tools and techniques that help inspect the output of clustering.

10.1 Inspecting clustering output

The primary tool in Mahout for inspecting the clustering output is called the `ClusterDumper`. It is found in the `mahout-utils` module in the Mahout codebase under the `org.apache.mahout.utils.clustering` package. `ClusterDumper` makes it easier to read the output of any clustering algorithm in Mahout. We have already seen it in action in the K-Means section of Chapter 9. We will be using it more in this chapter to understand clustering and its quality.

The `ClusterDumper` takes the input set of clusters and an optional dictionary that was generated during the conversion of the data to vectors. The full set of options for `ClusterDumper` tool is shown in table 10.1.

Table 10.1 Flags of Mahout cluster dumper tool and their default values

Option	Flag	Description	Default Value
SequenceFile dir (String)	-s	The directory containing the `SequenceFile` of the clusters	N/A
Output (String)	-o	The output file, if not specified print the output into the console	N/A
Points Directory (String)	-p	At the end of clustering, Mahout clustering algorithms produce two kinds of output. One is the set of <cluster-id, centroid> pair, other is the set <point-id, cluster-id> pair. The latter is generated when clustering finishes and usually resides in the points folder under the output. When this parameter is set to the points folder, all the points in a cluster are written to the output	N/A
JSON output (bool)	-j	If set, the centroid is written as a JSON format. Otherwise it substitutes in the terms for vector cell entries. By default this flag is unset	N/A
Dictionary (String)	-d	The path to the dictionary file which has the reverse mapping of integer id to word	N/A
Dictionary Type (String)	-dt	Format of the dictionary file. If text, then the integer id, and the terms should be tab separated. If the format is sequencefile, it should have an Integer key and a String value	text
Number of Words (int)	-n	The number of top terms to print.	10

Recall that any cluster has a centroid point, which is the average of all points in that cluster. This is true for all algorithms except model-based clustering like Dirichlet. The center in Dirichlet process clustering is not really the average of points, but a point that is the anchor to a particular group of data points.

Even then, the centroid points act as “summaries” of their clusters, and provides quick insight into the nature of each cluster. The top-weighted features of a centroid (its highest-value dimensions) are the features which indicate what that cluster is about. For text documents, the features are words and hence the top-weighted words in a centroid vector are words that indicate the meaning of documents in that cluster.

For example, the features from the document talking about “Obama” and “McCain” and “election” would have higher weight in a cluster of documents about the recent US presidential election. These top-weighted features are semantically related to each other. If not, something has gone wrong. If the top-weighted features are not related, this means the cluster has documented that did not belong together. This could be due to many reasons, most of which we will cover in this chapter.

As an example, let’s look at the output of `ClusterDumper` for two outputs of K-Means clustering: one using a Euclidean distance measure and the other using a cosine distance measure:

```
bin/mahout clusterdump
-s kmeans-output/clusters-19/
-d reuters-vectors/dictionary.file-0
-dt sequencefile -n 10
```

`ClusterDumper` outputs a block of information for each cluster including the centroid vector and top-weighted terms in the cluster. The dimensions of the vector would be translated to the corresponding word in the dictionary and printed on the screen. The output might be more easily examined as a text file, which can be accomplished with the option `-o`:

```
bin/mahout clusterdump
-s kmeans-output/clusters-19/
-o output.txt
-d reuters-vectors/dictionary.file-0
-dt sequencefile -n 10
```

The `output.txt` file can be opened on any text editor. The line containing the centroid vector for each cluster is usually large, so it is omitted in the output below. A typical sorted list of top ten terms for a cluster generated using K-Means clustering over the Reuters collection with a Euclidean distance measure looks like:

```
Top Terms:
                said                    =>   11.60126582278481
                bank                    =>   5.943037974683544
                dollar                  =>    4.89873417721519
                market                  =>   4.405063291139241
                us                      =>  4.2594936708860756
                banks                   =>  3.3164556962025316
                pct                     =>   3.069620253164557
                he                      =>   2.740506329113924
                rates                   =>  2.7151898734177213
                rate                    =>  2.7025316455696204
```

The top terms in this cluster indicate that it contains documents related to banks. A similar cluster using a cosine distance measure looks like:

```
Top Terms:
                bank                    =>  3.3784878432986294
                stg                     =>   3.122450990639185
                bills                   =>   2.440446103514419
                money                   =>   2.324820905806048
                market                  =>   2.223828649332401
                england                 =>  1.6710182027854468
                pct                     =>  1.5883359918481277
                us                      =>   1.490838685054553
                dealers                 =>  1.4752549691633745
                billion                 =>  1.3586127823991738
```

They are similar, but the second cluster appears better than the first one. The second cluster identifies “money” as an important term, where as the first cluster has a common verb “said” as the top term. The next section evaluates the quality of clustering by comparing the example above.

10.2 Analyzing clustering output

A clustering is a function, not only of the input, but several algorithms, each with parameters to tune. Therefore, when clustering goes wrong, there is no single way to “debug” it. It’s important to understand what is going on, intuitively, by analyzing clusters. So, let’s start analyzing the above output across the following three themes.

10.2.1 Distance measure and feature selection

Cosine distance measure is a better distance measure for texts as it groups documents by the highest-weighted common words between them. Here the TF-IDF-weighted vectors had higher weights for topic words. So similar documents clustered using the cosine distance measure came to have common topic words between them. This caused the cluster centroid vector to have a higher average weight for topic words than the stop-words.

The top words of the second cluster are topical and do not contain a stop-words like “said”. We encountered such words in the first cluster, including some unimportant ones. Though TF-IDF-weighted vectors were used in both instances, the way they were grouped in the first clustering gave more importance to the commonly occurring verb “said”. Even though the word “said” had a small weight, the

average weight in the cluster came out to be larger as compared to the topic words due to the equal importance Euclidean distance measure gives for any feature. So choosing a good distance measure helps ensure quality.

WEIGHTING THE FEATURES

Good clusters usually form around a few strong features, which drive a strong notion of similarity amongst the documents. That is, choosing the right features is as important as choosing the right distance measure. For unstructured text, better quality is achieved by extracting better features. There are limitations to what TF-IDF weighting can achieve. For practical applications, we might need to improve upon this weighting technique.

For other types of data, it takes some ingenuity on the part of the user to weight vectors in an optimal way. Good clustering requires more weight to be given to the important features and less weight to the unimportant ones.

Recall the "Apples to Vectors" example from Chapter 8. Numerically, color had a higher decimal value while the weight had a lower one. If we found that the weight was a better feature than color, the weights need to be scaled higher and color lower. So, the color values could be scaled to a 0-1 range and the weights to a 0-100 range. On doing so, the physical significance of the values vanish. What remains is a vector with features and weights that will cluster the apples better.

10.2.2 Inter-cluster and intra-cluster distances

Given all centroid points, the distance between all pairs of centroids could be calculated using the given distance measure and represented in a matrix. This matrix then gives a nice view of what happened in clustering, by showing how near or far the resulting clusters ended up. Inter-cluster distance is a good measure of clustering quality; good clusters probably do not have centroids that are too close, for this would indicate the clustering process is creating groups with similar features, and perhaps drawing distinctions between cluster members that are hard to support.

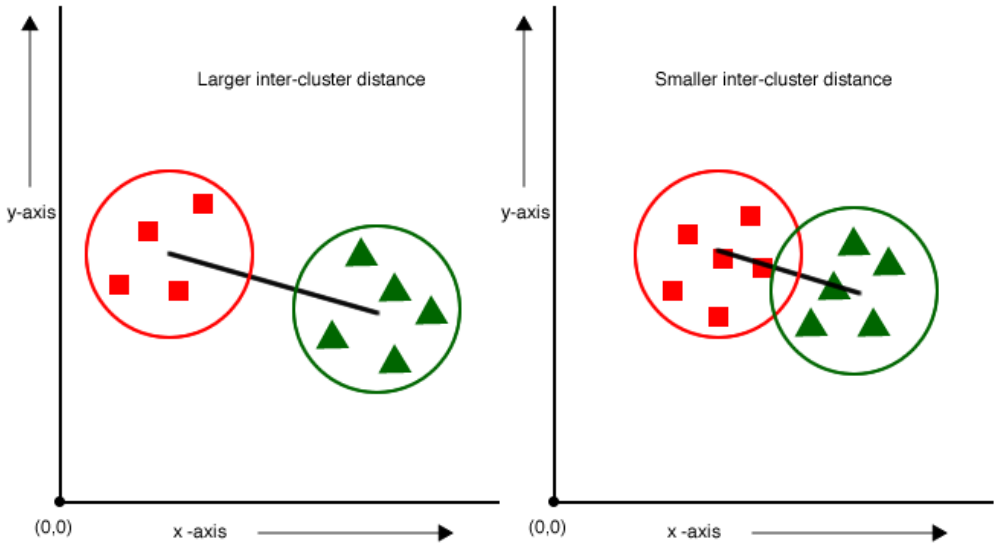


Figure 10.1 Inter-cluster distance is a measure of how well the data is separated. This is dependent on the feature weighting technique and the distance measure used.

Would it be useful to discover a news article in a cluster on "US", "president" and "election", and find that it does not exist in the cluster that talks about "candidate", "United States", and "McCain"? Maybe not; we may want clusters that are close to each other. Inter-cluster distance is a metric that correlates

with this aspect of quality. Figure 10.1 illustrates the inter-cluster distances across two different configurations of data.

We know that Euclidean distance values are larger as compared to cosine distance measure. Comparing the inter-cluster distances across two distance measure makes no sense unless they are scaled to the same range. The scaled average inter-cluster distances for all cluster centroid pairs is a better measure of clustering quality. Listing 10.1 details a simple way of calculating the inter-cluster distances from the output of clustering.

Listing 10.1 InterClusterDistances.java

```
public class InterClusterDistances {
  public static void main(String args[]) throws Exception {
    String inputFile = "reuters-kmeans-clusters/clusters-6/part-00000";  #1

    Configuration conf = new Configuration();
    Path path = new Path(inputFile);
    System.out.println("Input Path: " + path);
    FileSystem fs = FileSystem.get(path.toUri(), conf);

    List<Cluster> clusters = new ArrayList<Cluster>();

    SequenceFile.Reader reader = new SequenceFile.Reader(fs, path, conf);
    Writable key = (Writable) reader.getKeyClass().newInstance();
    Writable value = (Writable) reader.getValueClass().newInstance();

    while (reader.next(key, value)) {                              #2
      Cluster cluster = (Cluster) value;
      clusters.add(cluster);
      value = (Writable) reader.getValueClass().newInstance();
    }

    DistanceMeasure measure = new CosineDistanceMeasure();
    double max = 0;
    double min = Double.MAX_VALUE;
    double sum = 0;
    int count = 0;
    for (int i = 0; i < clusters.size(); i++) {
      for (int j = i + 1; j < clusters.size(); j++) {              #3
        double d = measure.distance(clusters.get(i).getCenter(),
            clusters.get(j).getCenter());
        min = Math.min(d, min);
        max = Math.max(d, max);
        sum += d;
        count++;
      }
    }

    System.out.println("Maximum Intercluster Distance: " + max);
    System.out.println("Minimum Intercluster Distance: " + min);
    System.out.println("Average Intercluster Distance(Scaled): "
        + (sum / count - min) / (max - min));
  }
}
```

#1 Path to the clustering output file
#2 Reading the cluster object from the SequenceFile
#3 Calculate distance measure between all pair of centroids

The listing computes the distances between all pairs of centroids and calculates the minimum, maximum and scaled inter-cluster distances. The above code produces the following output on the clustering output of `CosineDistanceMeasure` with Reuters:

```
Maximum Intercluster Distance: 0.9577057041381253
Minimum Intercluster Distance: 0.22988591806895065
Average Intercluster Distance(Scaled): 0.7702427093554679
```

When running the same code over the clusters generated using Euclidean distance measure on the Reuters, the output changes markedly:

```
Maximum Intercluster Distance: 96165.06236154583
Minimum Intercluster Distance: 2.7820472396645335
Average Intercluster Distance(Scaled): 0.09540179823852128
```

Note that the minimum and maximum inter-cluster distances were closer in magnitude when a cosine distance measure was used. This tells us that the clusters are equally spread out when using Cosine distance measure. The minimum inter-cluster distance measure was very small when using the Euclidean distance measure, indicating that at least two clusters are virtually the same or almost completely overlapping.

The average scaled inter-cluster distance shows clearly that the cosine distance measure is a better measure of similarity than Euclidean as it creates clusters that are more evenly separated. This explains why the clusters formed when using the cosine distance measure produces top terms of higher quality than when using a Euclidean distance measure.

INTRA-CLUSTER DISTANCE

Intra-cluster distance is the distance among members of a cluster, rather than distance between two different clusters. This metric gives a sense of how well the distance measure was able to bring the items together. Intra-cluster distances will be small as compared to inter-cluster distances. Figure 10.2 illustrates the hypothetical intra-cluster distances obtained when clustering using two different distance measures. A good distance measure will return a small distance between objects that are similar, and produce clusters that are "tighter" and therefore more reliably discriminated from one another.

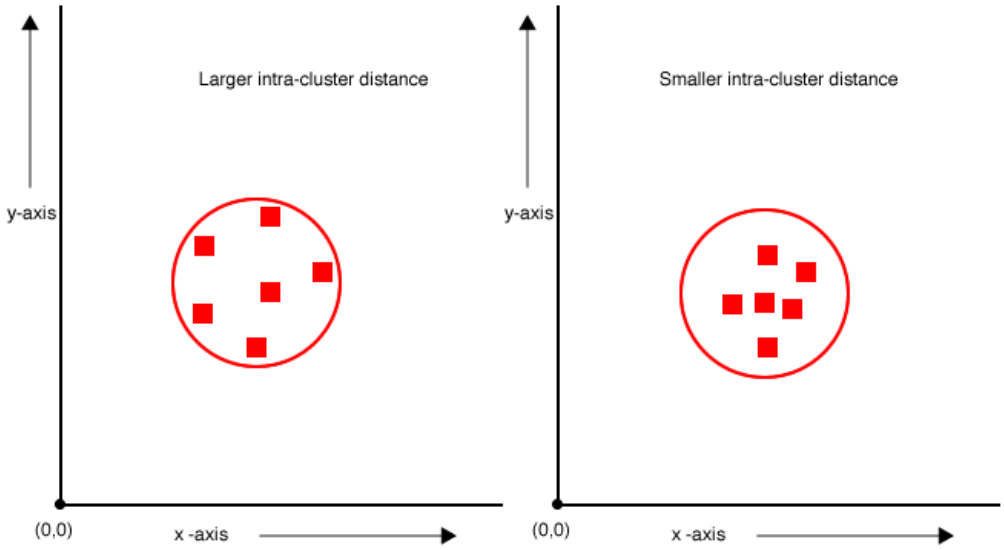


Figure 10.2 Intra-cluster distance is a measure of how well the points lie close to each other. This is dependent on two things: the penalty a distance measure gives for father objects and the smaller value it gives for closer objects. Higher the ratio of these two values, more separated out are the clusters.

10.2.3 Mixed and overlapping clusters

In some datasets, it is difficult to separate data points into good clusters. The clusters produced in such cases have low inter-cluster distances, and are of the same magnitude as that of intra-cluster distances. For them, changing distance measure or improving feature selection usually doesn't make much difference. Such datasets would require specialized algorithms like Fuzzy K-Means (to identify partial memberships) or Dirichlet process clustering (to identify a model that fits the data).

It would be worthwhile to recall the different clustering algorithms, their strengths, and the type of data they help cluster the best. The next section explores some advanced techniques that can improve the inter-cluster and intra-cluster distances.

10.3 Improving clustering quality

There are two important elements in improving the quality of clustering: improving the weights of the features in a document vector and creating a more appropriate distance measure. A good weighting technique can promote the good features of an object and an appropriate distance measure can help bring similar features together. The next two sections explain how we can create custom feature selection and distance measurement classes.

10.3.1 Improving document vector generation

A good document vector has the right kind of features, with higher weights assigned to the more important ones. In text data, there are two ways to improve the quality of a document vector: by removing noise and using a good weighting technique.

Not every text document in the world is of high quality: the sentences could be malformed, or the words might be hard to distinguish from formatting or structure. This is true for large datasets generated from the web. Most of the textual content on the Internet such as in web pages, blogs, wikis, chats, or discussion forums, is delivered as a complex collection of markup, stylesheets and script, not pure text. Texts generated from OCR (Optical Character Recognition) of scanned documents and from SMS messages are of very poor quality due to mistaken character recognition ("tne" for "the") or extreme abbreviations from slangy messaging ("c u" for "see you"). The text might be missing characters, spaces or punctuation, or contain random jargon or even unintended words and grammatical mistakes. Clustering in the presence of such noise is hard. To get optimum quality, the data first needs to be cleaned of these errors.

Some existing text analysis tools do this well. Mahout provides a hook that allows any text filtering technique to be injected into the vectorization process. This is done through something called a Lucene `Analyzer`. We already caught a glimpse of Lucene `Analyzers` in Chapter 9 when we tried to improve the news-clustering module, though we didn't explain much what it did. Creation of a custom Lucene `Analyzer` involves the following:

- Extending the interface `Analyzer`
- Overriding the `tokenStream(String field, Reader r)`

Listing 10.2 shows a custom Lucene Analyzer, which can tokenize a document, using the `StandardTokenizer`, a somewhat error resilient tokenizer implementation in Lucene.

Listing 10.2 A custom Lucene Analyzer that wraps around StandardTokenizer

```
public class MyAnalyzer extends Analyzer {  #A

  @Override
   public TokenStream tokenStream(String fieldName, Reader reader) {  #B
     TokenStream result = new StandardTokenizer(Version.LUCENE_CURRENT, reader);
     return result;
   }
}
```

#1 Any custom analyzer extends the abstract class Analyzer
#2 The tokenStream function takes a Java Reader class and returns the tokens in the stream

The custom analyzer tokenizes the input `Reader` into a stream on tokens. In the above example we are simply using the `StandardTokenizer` to tokenize the contents on the reader. Next we will try improving this `Tokenizer`. Instead of simply tokenizing words, we will also add a filter, which converts the token into the lower case. The updated class looks as follows:

Listing 10.3 MyAnalyzer with lowercase filter

```
public class MyAnalyzer extends Analyzer {

  @Override
   public TokenStream tokenStream(String fieldName, Reader reader) {
     TokenStream result = new StandardTokenizer(Version.LUCENE_CURRENT, reader);
     result = new LowerCaseFilter(result);   #1
     return result;
   }
}
```

#1 Apply a lower case filter to the tokenized stream.

`TokenFilter` applies some transformation to the underlying token stream. These filters could be chained. Other than `LowerCaseFilter`, there are `StopFilter`, `LengthFilter` and `PorterStemFilter` in Lucene. `StopFilter` skips over stopwords in the underlying stream, `LengthFilter` filters out tokens whose lengths are beyond the specified range, and finally the `PorterStemFilter` performs stemming on the underlying word. Stemming is a process of reducing a word into its base form. For example, the words “kicked” and “kicking” is both reduced to ”kick”. Using these, a custom Lucene `Analyzer` is created as in Listing 10.4

Listing 10.4 MyAnalyzer.java

```
public class MyAnalyzer extends Analyzer {

  @Override
   public TokenStream tokenStream(String fieldName, Reader reader) {
     TokenStream result = new StandardTokenizer(Version.LUCENE_CURRENT, reader);
     result = new LowerCaseFilter(result);                    #1
     result = new LengthFilter(result,3, 50);                 #2
     result = new StopFilter(result, StandardAnalyzer.STOP_WORDS_SET);   #3
     result = new PorterStemFilter(result);                    #4
     return result;
   }
}
```

#1 Apply a lower case filter to the tokenized stream.
#2 Prune the words less than 3 or greater than 50 in length
#3 Remove stop words from the list using the StopFilter
#4 Stem the words to their base form.

Using this Lucene `Analyzer`, let’s create TF-IDF vectors and cluster the data from the Reuters collections. We will borrow the code from Chapter 9 and use it with this Analyzer as shown in listing 10.5.

10.5 NewsKMeansClustering.java using MyAnalyzer

```
public class NewsKMeansClustering {
  public static void main(String args[]) throws Exception {
    int minSupport = 5;
    int minDf = 5;
    int maxDFPercent = 99;
    int maxNGramSize = 1;
    int minLLRValue = 50;
    int reduceTasks = 1;
    int chunkSize = 200;
    int norm = -1;
    boolean sequentialAccessOutput = true;

    String inputDir = "kmeans-seqfiles";
    File inputDirFile = new File(inputDir);
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(conf);
    String outputDir = "newsClusters";
    HadoopUtil.overwriteOutput(outputDir);
    String tokenizedPath = outputDir
        + DocumentProcessor.TOKENIZED_DOCUMENT_OUTPUT_FOLDER;
    MyAnalyzer analyzer = new MyAnalyzer();                          #1
    DocumentProcessor.tokenizeDocuments(inputDir, analyzer.getClass()
        .asSubclass(Analyzer.class), tokenizedPath);                #2
```

```
    DictionaryVectorizer.createTermFrequencyVectors(tokenizedPath,    #3
      outputDir, minSupport, maxNGramSize, minLLRValue, reduceTasks,
      chunkSize, sequentialAccessOutput);
    TFIDFConverter.processTfIdf(                                        #4
      outputDir + DictionaryVectorizer.DOCUMENT_VECTOR_OUTPUT_FOLDER,
      outputDir + TFIDFConverter.TFIDF_OUTPUT_FOLDER, chunkSize,
      minDf, maxDFPercent, norm, sequentialAccessOutput);

    String vectorsFolder = outputDir
                           + TFIDFConverter.TFIDF_OUTPUT_FOLDER
                           + "/vectors";
    String centroids = outputDir + "/centroids";
    String clusterOutput = outputDir + "/clusters";


    RandomSeedGenerator.buildRandom(vectorsFolder, centroids,
      20);
    KMeansDriver.runJob(vectorsFolder, centroids,                  #5
      clusterOutput, CosineDistanceMeasure.class.getName(), 0.01,
      20, 1);

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

  }
}
```

#1 Initialize MyAnalyzer
#2 Tokenize documents using MyAnalyzer
#3 Create TF Vectors
#4 Using TF Vectors create IDF Vectors
#5 Using TFIDF Vectors run K-Means with random centroids

The code is similar to code seen in Chapters 8 and 9. Instead of the `StandardAnalyzer` we used `MyAnalyzer` and ran K-Means clustering over the generate vectors. After running clustering using the Reuters data with the code above, the top ten terms in the bank cluster looks as follows:

```
Top Terms:
                billion             =>   4.766493374867332
                bank                =>  2.2411748607854296
                stg                 =>  1.8598368697035639
                monei               =>  1.7500053112049054
                mln                 =>  1.7042239766168474
                bill                =>  1.6742077991552187
                dlr                 =>  1.5460346601253139
                from                =>  1.5302046415514483
                pct                 =>  1.5265302869149873
                surplus             =>  1.3873321058744208
```

Note that the words were stemmed using the `PorterStemFilter` and the `StopwordsFilter` are virtually absent. This illustrates the power of selecting the right set of features for improving clustering.

10.3.2 Writing a custom distance measure

If the vectors are of the highest quality, the biggest improvement in cluster quality comes from the choice of an appropriate distance measure. We have seen how the cosine distance is a good distance measure for clustering text documents. To illustrate the power of a custom distance measure, we will create a different form of the cosine distance measure, which exaggerates distances: it makes big distances bigger and small distances smaller. Like an `Analyzer`, writing a custom `DistanceMeasure` involves:

- Implementing the `DistanceMeasure` Interface
- Writing the measure in the method `distance(Vector v1, Vector v2)`

Our modified cosine distance measure is shown in listing 10.6 below:

10.6 MyDistanceMeasure

```
public class MyDistanceMeasure implements DistanceMeasure {   #1
  @Override
  public double distance(Vector v1, Vector v2) {               #2
```

```
    if (v1.size() != v2.size()) {
      throw new CardinalityException();
    }
    double lengthSquaredv1 = v1.getLengthSquared();
    double lengthSquaredv2 = v2.getLengthSquared();

    double dotProduct = v2.dot(v1);

    double denominator = Math.sqrt(lengthSquaredv1)
                         * Math.sqrt(lengthSquaredv2);

    if (denominator < dotProduct) {
      denominator = dotProduct;
    }
    double distance = 1.0 - dotProduct / denominator;           #3

    return (distance * distance);                               #4
  }

  @Override
  public double distance(double centroidLengthSquare,
                         Vector centroid,
                         Vector v) {
    return distance(centroid, v);
  }

  @Override
  public void createParameters(String prefix, JobConf jobConf) {}

  @Override
  public Collection<Parameter<?>> getParameters() {
    return Collections.emptyList();
  }

  @Override
  public void configure(JobConf arg0) {}
}
```

#1 A distance measure implements the DistanceMeasure interface
#2 The distance function needs to be overridden with custom logic
#3 The distance measure computes the standard cosine distance
#4 The square of the cosine distance is returned

The `configure()` and `getParameters()` / `createParameters()` methods are used to pass arguments to the `DistanceMeasure` during runtime at each job node in a MapReduce job. The method of most interest to us is the distance function. Note, how it calculates the standard cosine distance measure and then tries to bring distances closer by squaring the cosine distance value (0 - 1.0). Some of the interesting niche topics that surfaced due to this distance measure, which were not there earlier are:

```
Top Terms:
                futures            =>  3.2517230513480757
                trading            =>    2.73124880187049
                exchange           =>  2.2752173132458906
                market             =>  2.0752484701513274
                said               =>  1.9316076421017707
                stock              =>  1.8062251904561821
                new                =>  1.5571645992558176
                traders            =>   1.531255338250137
                index              =>  1.3630116680911748
                options            =>   1.183384693735014
```

Note that this output was generated from vectors tokenized using `StandardAnalyzer`; we leave it to readers to experiment with both a custom Lucene `Analyzer` and a custom `DistanceMeasure` to get the desired clustering.

Summary

In this chapter we explored how clustering can be improved. The algorithms, the distance measures, and the type of data influence the output of clustering. The `ClusterDumper` tool can inspect output of all the

clustering algorithms: the top features of each cluster and centroid vector. This can help understand what has happened in clustering.

For large data, it's infeasible to manually inspect all clusters. Instead, inter-cluster distance and intra-cluster distance can provide a quick quantitative "score" of the clustering quality. If many clusters are too close, then it's time to move past K-Means and see how partial membership and mixture distributions can be clustered out.

Writing a custom Lucene Analyzer and a custom similarity metric can help improve the clustering quality beyond what the standard implementations can do. We saw that it is worthwhile to explore custom measures to improve clustering.

If you have read this far, you are ready to take on any type of data and tune your clustering algorithm to work well. You will soon encounter the biggest obstacle in clustering: scale. Fortunately, Mahout can crunch terabytes of data over large Hadoop clusters. Next, we will explore how the clustering algorithms in Mahout leverage the power of Hadoop. Next, we will show how you can run a clustering job over a Hadoop cluster and take your clustering to production in Chapter 11.

11

Taking clustering to production

This chapter covers

- Running a clustering job on a Hadoop cluster
- Tuning a clustering job for performance
- Batch clustering versus online clustering

We have seen how different clustering algorithms in Mahout grouped the document in the Reuters news dataset. On the way, we learned about the vector representation of data, the distance measure and various other techniques to improve quality of the clusters. One of Mahout's strength is its ability to scale. The Reuters dataset wasn't much of a challenge. So in this chapter we will set a bigger challenge for Mahout: clustering one of the largest free datasets in the world: Wikipedia – the free encyclopedia. Mahout can handle such scale because the algorithms in it are implemented as MapReduce jobs that can execute on a Hadoop cluster over hundreds and thousan[17]ds of computers.

Unfortunately, not everyone have access to such a cluster at this moment. For demonstration purposes in this chapter, we will use a subset of documents extracted from Wikipedia and try it on a small cluster, and show how speedup is achieved by adding more computers.

11.1 From zero to clustering on Hadoop in 10 minutes

Let's first look at an aspect of Hadoop's architecture. A Hadoop cluster consists of a single server called the NameNode that handles the different computers, called DataNodes, in the cluster, and synchronizes the distributed file system, HDFS. Another server called the Jobtracker manages all the MapReduce jobs and manages the computers (nodes) where the `Mapper` and `Reducer` classes execute on the cluster. On each of these nodes, another process called the TaskTracker manages the `Mapper` or `Reducer` execution request from the JobTracker. Hadoop does all this seamlessly and without any user intervention. In single-node clusters, the NameNode, JobTracker, DataNode and TaskTracker all run on the same system as separate processes talking with each other.

The version of Hadoop that Mahout is designed to work with at the moment is 0.20.2 (though it should be compatible with other recent versions). As before in Chapter 6, we will need a local Hadoop cluster set up in pseudo-distributed mode in order to proceed. The following web page is a short tutorial on setting it up:

http://hadoop.apache.org/common/docs/r0.20.2/quickstart.html

The Hadoop binary resides in the `bin/` folder of the Hadoop home directory. To view the contents of the HDFS file system, execute:

[17] 4000 node hadoop: http://developer.yahoo.net/blogs/hadoop/2008/09/scaling_hadoop_to_4000_nodes_a.html

```
bin/hadoop dfs –ls /
```

This will list all files in the root of the file system. When the cluster is started for the first time, your home directory might not exist in HDFS. So, create a `/users/<your UNIX user name>` folder:

```
bin/hadoop dfs –mkdir /users/<your UNIX username>
```

The home directory can also be listed as follows:

```
bin/hadoop dfs –ls ~
```

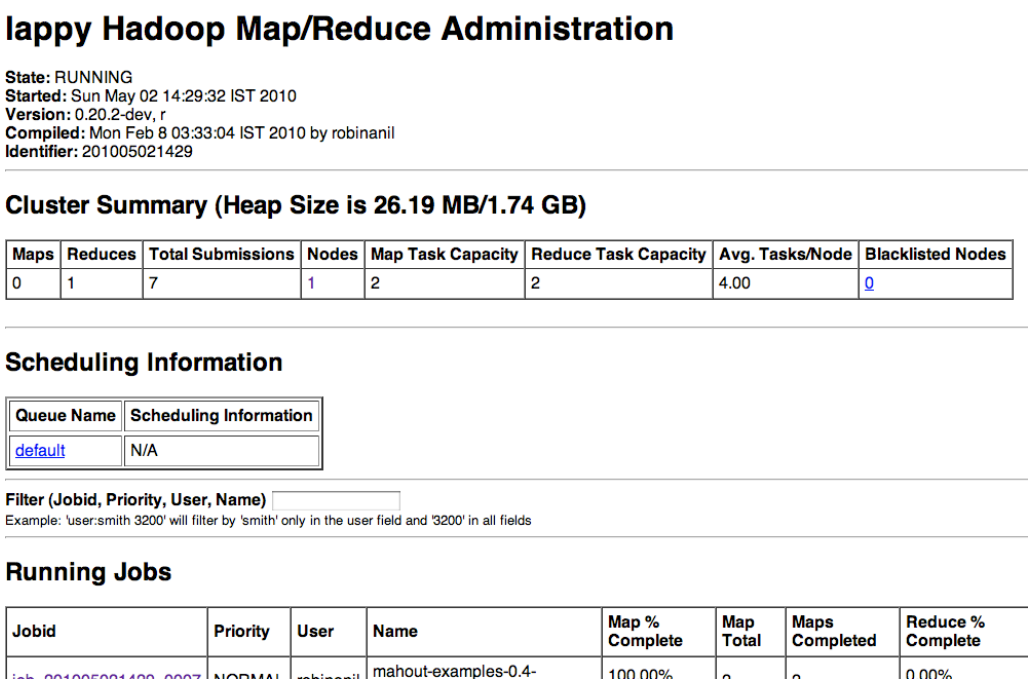
The symbol ~ expands to the home directory of the current user as in UNIX and Linux. To begin clustering of Reuters data on our pseudo-distributed cluster, prepare the Reuters `SequenceFile` containing the text data locally as given in Chapter 8 and copy it to the HDFS:

```
bin/hadoop dfs –put <path-to>/reuters-seqfiles reuters-seqfiles
```

Using this as the input, we will run the dictionary-based vectorizer on the cluster and then run K-Means clustering on it. Any Hadoop MapReduce job is executed using the `hadoop jar` command. Mahout puts all example class files and dependencies in a single JAR file under `examples/target/mahout-examples-0.4-SNAPSHOT.job`. To run the dictionary vectorizer in the local Hadoop cluster using the Reuters `SequenceFiles` as input, simply execute the Hadoop jar command with the Mahout job file as follows:

```
bin/hadoop jar examples/target/mahout-examples-0.4-SNAPSHOT.job \
  org.apache.mahout.text.SparseVectorsFromSequenceFile \
  -w –i reuters-seqfiles –o reuters-vectors
```

That's it. The clustering executes on the local Hadoop cluster. If the default Hadoop configuration was used and if the system had at least two processor cores, two mappers would be executed in parallel, with each running on one core. This can be tracked on JobTracker dashboard at http://localhost:50030 as shown in Figure 10.1.



lappy Hadoop Map/Reduce Administration

State: RUNNING
Started: Sun May 02 14:29:32 IST 2010
Version: 0.20.2-dev, r
Compiled: Mon Feb 8 03:33:04 IST 2010 by robinanil
Identifier: 201005021429

Cluster Summary (Heap Size is 26.19 MB/1.74 GB)

Maps	Reduces	Total Submissions	Nodes	Map Task Capacity	Reduce Task Capacity	Avg. Tasks/Node	Blacklisted Nodes
0	1	7	1	2	2	4.00	0

Scheduling Information

Queue Name	Scheduling Information
default	N/A

Filter (Jobid, Priority, User, Name)
Example: 'user:smith 3200' will filter by 'smith' only in the user field and '3200' in all fields

Running Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete
job_201005021429_0007	NORMAL	robinanil	mahout-examples-0.4-	100.00%	9	9	0.00%

Figure 10.1 A screenshot showing a typical Hadoop MapReduce JobTracker page. To got to the above page for the local Hadoop cluster point the browser to http://localhost:50030.

Out of the box, Hadoop executes clustering twice as fast as the local runs, which we invoked earlier in Chapters 8, 9 and 10. If more that two cores are available on the computer, the Hadoop configuration file can be modified to set the number of mapper and reducer tasks to a higher value increasing the overall speed of computation. The configuration property `mapred.map.tasks` and `mapred.reduce.tasks` in `mapred-site.xml` of the Hadoop installation needs to to be configure to the appropriate value based on the number of CPU cores.

With each additional core, the parallelism increases and processing time decreases. Once the maximum parallel tasks hit the peak for a given single node, increasing the tasks decreases performance heavily. The only way to scale more is to have multiple nodes with the same configuration: a fully distributed Hadoop cluster.

TIP

The web page http://hadoop.apache.org/common/docs/r0.20.2/cluster_setup.html gives a tutorial with step-by-step instructions for setting up a distributed Hadoop cluster. The 0.20.2 and later versions of Hadoop has configuration split across three files: `core-site.xml`, `hdfs-site.xml` and `mapred-site.xml` in the conf folder. The default configuration values are set in `core-default.xml`, `hdfs-default.xml` and `mapred-default.xml` respectively. A parameter given in the default file can be overridden in the site xml file. Tuning the MapReduce parameters usually involves editing the `mapred-site.xml`.

Executing clustering code on a distributed Hadoop cluster is no different than executing it on a single node cluster. The `bin/hadoop` script can launch the job on the cluster from the NameNode, slave nodes or any computer, which has access to the NameNode. The only requirement is that the script, when run, needs to access the correct configuration files of the cluster through the `HADOOP_CONF_DIR` environment variable.

EXECUTING JOBS ON HADOOP CLUSTER USING MAHOUT LAUNCHER

Similar to the Hadoop launcher script, Mahout provides a script `bin/mahout` to launch clustering jobs. We had used it extensively in the previous chapters to launch clustering as a single-process job. By setting the `HADOOP_HOME` and `HADOOP_CONF_DIR` environment variables, the same script could be used to launch any of the Mahout algorithms on a Hadoop cluster. The script will automatically read the Hadoop cluster configuration files and launch the Mahout job on the cluster. A typical output would be like:

```
export HADOOP_HOME=~/hadoop/
export HADOOP_CONF_DIR=~/hadoop/conf
bin/mahout kmeans –h

running on hadoop, using HADOOP_HOME=/Users/username/hadoop and
HADOOP_CONF_DIR=/Users/username/hadoop/conf
…
…
```

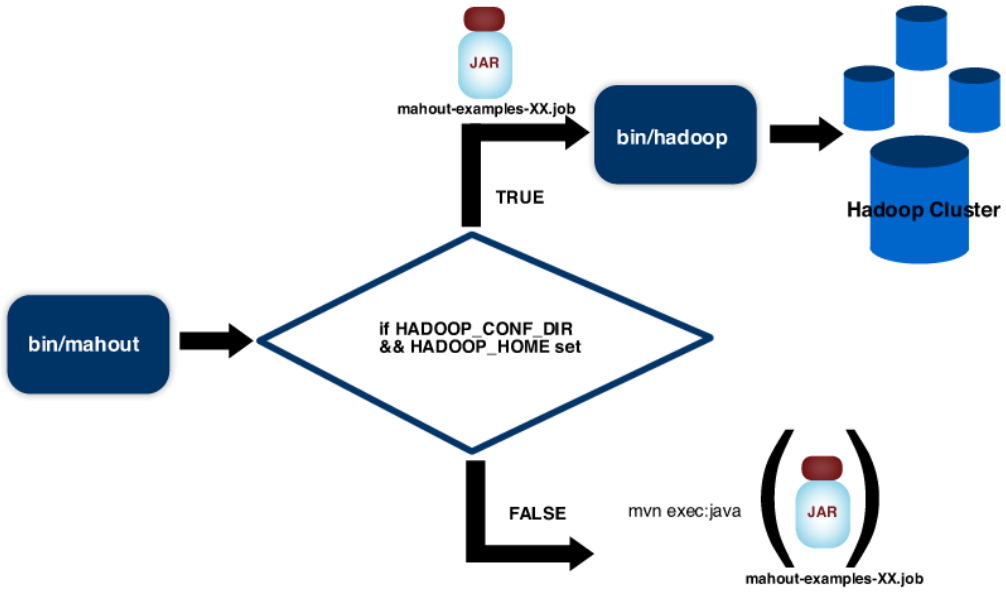


Figure 11.2 A logic diagram showing the various modes of execution of the Mahout launcher script.

The Mahout launcher script internally calls the Hadoop launcher script using the correct cluster configuration files. A schematic diagram of the same is shown in Figure 11.2. In mahout, the launcher script is the easiest way to launch any algorithm locally or on a distributed Hadoop cluster.

11.2 Tuning clustering performance

Clustering algorithms in Mahout are designed to run in parallel. Though the algorithms differ, they are similar in one respect: they all read from the `SequenceFile` of vectors in parallel in each `Mapper`. Many of these operations in the clustering algorithms are CPU-bound. That means for many operations, the vector serializations and deserializations, distance computations, and so on keep the CPU at full usage. On the other hands, some other operations are I/O bound: for example, transmitting the centroid to each `Reducer` over the network. For improving clustering performance, one needs to understand tricks to address one or the other type of performance bottleneck. For reference, the K-Means clustering algorithm is illustrated in Figure 11.3. Other clustering algorithms like Fuzzy K-Means, Dirichlet, LDA all have an architecture similar to that of K-Means and the optimizations hold true for all of them

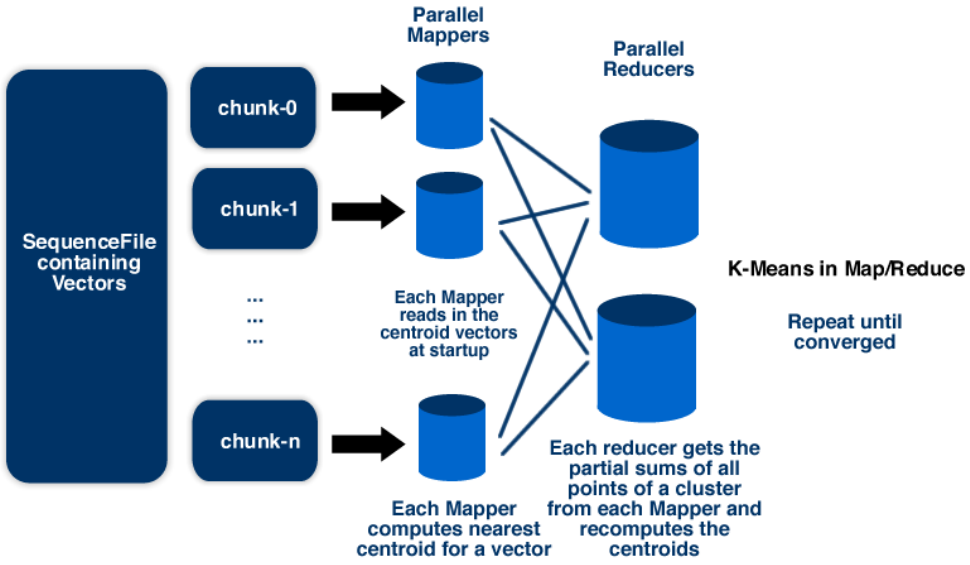


Figure 11.3 Schematic diagram of K-Means clustering iteration running as a MapReduce Job

11.2.1 Avoiding performance pitfalls in CPU-bound operations

CPU performance drops when heavily used functions start becoming slower. In clustering, the distance computation is CPU bound, making the computation faster makes the whole job faster. The following things need to be kept in mind when tackling a clustering problem.

USING AN APPROPRIATE VECTOR REPRESENTATION

`DenseVector` is generally the fastest `Vector` among the three in Mahout. Any element can be accessed quickly, and it can be iterated over in sequence efficiently. However, using `DenseVector` for vectors that are sparse can cause significant performance problems. When used to represent a vector that has many many zero elements, a `DenseVector` will wastefully store all of those elements that would otherwise have no representation in a sparse representation. This means more data is needlessly serialized, and that many zero elements are iterated over in operations that wish to ignore zero elements.

For example, say we are clustering very sparse data (the number of non-zero elements is about 1% of the cardinality) like text documents vectors. The clustering using `SparseVector` is about two times as fast `DenseVector` on the local system and about 10 times faster on distributed system. The extra performance during distributed clustering is due to the latency caused by transferring unnecessary zero bytes over the network in a `DenseVector`.

So, it's usually best to represent as a `SparseVector` because even with smaller sparseness it gives great savings in storage, deserialization, and network operation.

USING A FAST DISTANCE MEASURE:

Clustering algorithms calculate distance a large number of time, so using a fast distance measure implementation is crucial. Improvements in the distance measure's speed go straight to the "bottom line" of the overall algorithm's performance. If you implement your own distance measure, follow the best practices given below:

- Avoid cloning or instantiating a new `Vector`. Since they are heavy Java objects, cloning them degrades performance heavily
- Avoid iterating on all elements if distance measure requires only non-zero elements. Use `Vector.iterateNonZero()` iterator instead of `Vector.iterator()`
- Use `Vector.assign()`, `Vector.aggregate()` methods to efficiently traverse and modify the vector. Find out more about them in appendix B.

USING A SPARSEVECTOR TYPE BASED ON THE DISTANCE MEASURE COMPUTATION

There are two sparse vector implementations, and it pays to use an implementation appropriate to the distance measure computation. `RandomAccessSparseVector` is tuned for random lookups and `SequentialAccessSparseVector` is tuned for fast sequential access.

For example, computing a cosine distance measure entails many vector dot product operations. This requires iterating over two vectors in order, element by element. The code needs to multiply values from matching indices from both vectors. Naturally, `SequentialAccesssSparseVector` is optimal for such sequential access pattern in the distance measure and is much faster as compared to the `RandomAccessSparseVector`. So keeping all the document vectors in sequential format gives huge boost to the speed of distance-measure computation and hence improves the overall clustering performance.

> **TIP**
>
> The Mahout utils package has a utility class called the `VectorBenchmark`. It pits different types of vector operations across combinations of dense, random-access and sequential-access vectors and compares their speed. If your custom distance measure does certain types of vector operations more, then using this utility, find the vector that performs the best for that operation at a certain sparsity level. The tool is found in the `mahout-utils` module in package `org.apache.mahout.benchmark`.

11.2.2 Avoiding performance pitfalls in I/O-bound operations

I/O performance is usually affected most by the amount of data the program reads and writes. In the case of Hadoop jobs, these are confined mostly to sequential reads and writes. Decreasing the amount of bytes written improves the overall clustering performance greatly. The following things need to be kept in mind to decrease the I/O bottleneck.

USING AN APPROPRIATE VECTOR REPRESENTATION

This is quite obvious. As explained earlier, never store `SparseVectors` as `DenseVector` objects. It bloats the size on disk can creates both disk and network I/O bottleneck for the clustering application.

USING HDFS REPLICATION

Any file stored on HDFS is by default replicated three times across various nodes of the cluster. This is done to not only to guard against data loss, in case one copy on one node is lost, but also to increase performance. If data exists on only one node, then all `Mappers` and `Reducers`, which need that data, must access that one node, potentially across the network. This can be a bottleneck. Replication makes HDFS store the chunks on separate servers. This gives Hadoop choices of nodes where the computation could be initialized, thereby reducing this network I/O bottleneck. Replication can be increased, to potentially further reduce this bottleneck, but will mean more storage is used on HDFS. Consider this only if there is evidence that under-replication is creating a bottleneck.

REDUCE THE NUMBER OF CLUSTERS

Clusters are usually represented as large, dense vectors, each of which consumes considerable storage. If the clustering job is trying to find a large number of clusters (k), these vectors are sent across the network from the `Mappers` to the `Reducers`. Having a small value of k will decrease this network I/O. It also decreases the overhead of distance-measure computation in K-Means and Fuzzy K-Means where each cluster centroid increases the distance measure computation by n where n is the number of points stored on the disk.

When it's necessary to cluster data into large number of clusters, we can throw it all at Hadoop and add more computers and let it scale for itself. Alternatively, we can smartly reduce the distance measure computation and clustering time by an order of magnitude using a two-tiered approach as explained next.

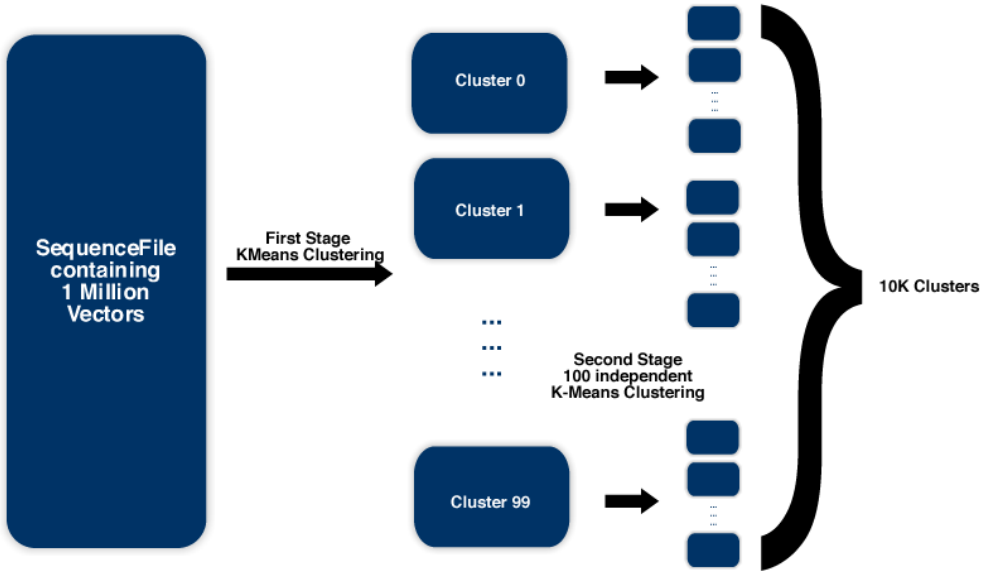


Figure 11.4 Applying K-Means in two levels. A single large MapReduce with 10,000 clusters will have $Nx10^6x10^4$ distance computations, where N is the number of iterations. On the other hand, the hierarchical breakdown of the problem gives rise to only $N_1x10^6x10^2 + N_2x10^4x10^2x10^2$ distance computation, or almost 100 times fewer distance computations.

11.3 Batch versus online clustering

Let us return to the online news portal AllMyNews.com case study from Chapter 9. We found that a related articles feature implementation could be construed as a simple clustering problem. However, clustering won't give the desired final output. Assume the portal has about a million news articles. We would have to produce clusters containing only up to, say, 100 articles, or else the list of related articles will become unusably large. But that implies producing 10,000 clusters. This is not a good way to go about this problem. Though Hadoop enables us to scale up to do this, it would consume a wasteful amount of CPU and disk, which cost money. A better way to go about this would be to split the articles into just 100 giant clusters. Then for each cluster of about 10,000 articles, cluster again to get 100 smaller clusters as illustrated in Figure 11.4.

As in past examples, we are clustering all articles in one big effort. This is known as batch clustering, or offline clustering. Now, as new articles come into the news portal, they need to be clustered too. It would be quite expensive to keep running the entire batch clustering jobs again and again just to cluster a few recent articles. Even if it were run hourly, we'd be unable to identify related articles for a new article for up to an hour. This is clearly not a desirable behavior. Enter online clustering techniques, which aim to efficiently cluster new items given clustering of existing items right away.

11.3.1 Case study: Online news clustering

Online clustering here is not truly online, truly immediate. There are algorithms that directly cluster streams of data, but have trouble scaling. Instead, consider the following technique:

- Cluster 1 million articles as above and save the cluster centroids for all clusters
- Periodically, for each new article, use canopy clustering to assign it to the cluster whose centroid is closest based on a very smal distance threshold. This ensures that articles on topics that occurred previously are associated with that topic cluster and shown instantly on the website. These documents are removed from the new document list.
- The left over documents, which are not associated with any old cluster, forms new canopies. These canopies represent new topics that appeared in the news that has little or no match with any articles that we have from the past.

- Use the new canopy centroids and cluster the articles that are not associated with any of the old clusters and add these temporary cluster centroids to our centroid list.
- Less frequently, execute the full batch clustering to re-cluster the entire set of documents. While doing so, it is useful to keep all previous cluster centroids as input to the algorithm so that clustering achieves faster convergence.

The key is to ensure that the incremental canopy clustering is done as fast as possible that the user gets related clusters within minutes of publishing the story. One dedicated node can easily take care of this. Alternately, we can delay publishing the story on the website by a couple of minutes until related-article clusters are generated.

This example shows that just because a solution can use up lots of resources easily and scale doesn't mean it should use lots of resources. We could have run full clustering on a dedicated cluster having thousands on nodes, but this would have just wasted money. A cheaper and faster solution is available with a little ingenuity.

11.3.2 Case study: Clustering Wikipedia articles

Wikipedia's articles are exported every night into XML files and made available here: http://dumps.wikimedia.org/enwiki/

This is an amazing data set. Since its being edited and organized by humans, many articles remain insufficiently or incorrectly grouped taxonomized within Wikipedia. We will try to run clustering on these articles to extract the related articles that could potentially help the editors group the related articles on the Wikipedia website.

Before attempting to cluster, we need to extract the documents and vectors from the XML format. Fortunately, Mahout has a Wikipedia dataset creator class that can read the XML format and output each individual article in a `SequenceFile` format. The extractor can also select the documents matching a certain category. For the purpose of this experiment, we will extract out all documents in the Science category. First, download the latest dump of `page-articles.xml.bz2` file from the above website. Extract it to a local directory, and upload it to the HDFS. Assuming the Hadoop environment variables are set, run the Wikipedia extractor as follows:

```
echo "science" > categories.txt
bin/mahout seqwiki –c categories.txt –i articles.xml \
  –o wikipedia-seqfiles –e
```

This will select all documents from the articles XML that have a Wikipedia category exactly matching the word "science" and write those articles to a `SequenceFile`. Thus we can run the `SparseVectorsFromSequenceFile` job over this output and create the Wikipedia TF-IDF vectors. Then we can run any clustering algorithm on top of it. This whole process will take quite a while.

Once you have successfully run clustering, you may wish to experiment with the clustering algorithm in order to observe the effect of, say, choice of vector representation. Recreate the vectors as a `SequentialAccessSparseVector` with the `–seq` flag in the dictionary vectorizer. This will run faster than the `RandomAccessSparseVector`, which is created as default.

Once you are confident enough in clustering the Wikipedia sample set, extract the full data set and vectorize it as follows as follows:

```
bin/mahout seqwiki -all –i articles.xml –o wikipedia-seqfiles
```

Unless you have access to a powerful Hadoop cluster, this may take up a whole a day on one machine. One option is to run it on the cloud, as we will see next.

RUNNING WIKIPEDIA CLUSTERING ON AMAZON ELASTIC MAP REDUCE

One cheap way to explore Mahout is to purchase Hadoop cluster time from providers "in the cloud" like Amazon's Elastic MapReduce. Section 6.6 of this book showed us quick way to access this resource. For those who are comfortable with the Amazon Elastic MapReduce (EMR) service, Mahout algorithms could be run as follows:

- Upload Mahout job file to an Amazon S3 bucket.
- Provide the path to the .job file as the .jar file.
- Give the full class name of the driver class of the algorithm that needs to be run – perhaps `org.apache.mahout.clustering.kmeans.KMeansDriver` – and the parameters
- Give the input path of the Wikipedia Vectors file stored on a S3 bucket and output path on the S3 bucket where clusters need to be written.
- Give other required parameters for the MapReduce like the number of machines. Remember, each compute unit is charged by the hour, so a large cluster quickly gets expensive
- Now run the job. Follow the same procedure to generate vectors directly using the EMR service by uploading the sequence file instead.

At the time of writing, it took an eight-node cluster about an hour to convert Wikipedia `SequenceFiles` into vectors. The clustering time depends heavily on the initial centroids, the distance-measure, and the number of clusters -- and ofcourse the number of machines available to work on the computation. Always remember, the computation is not free.

Summary

In this chapter, we took a brief look at Hadoop cluster and how a clustering job is executed on it. We also looked at the Mahout launcher script and its two modes of operation.

We discussed the various issues in clustering performance and analyzed various techniques to bring down the CPU and the I/O bottleneck of clustering. Correct vector representation and optimized distance measurement was the key to a high performance clustering application.

We translated what we learned in this section to complete the pseudo-online clustering engine for our fictitious news aggregation portal – AllMyNews.com. With a bit of ingenuity, the large clustering problem was broken down into simpler and faster one using a mix of techniques thus achieving near-real-time clustering of related articles in the system. To highlight the ability of Mahout clustering to scale, we tried to cluster the largest public database available today, the English Wikipedia. Since the process was slow, we took a quick look at how multi node clustering could be performed on the cloud using the Amazon Elastic MapReduce service.

With this, we learned to run Mahout on a Hadoop cluster and make it scale. The journey wasn't simple. We started with a simple clustering of points in a plane from which we learned about the notion of vectors and distance measures. We toured various algorithms present in Mahout and tried to apply the learnings to a real world case study, this gradually evolved from small input and non-distributed computation and finally to large-scale distributed computation. We also learned techniques to tune the algorithms for speed and quality. Next, in Chapter 12 will use all these learnings and apply them real world case studies.

12

Real-world applications of clustering

This chapter covers

- Real-world applications of clustering
- Finding like-minded people on Twitter
- Tag suggestion for artist on Last.fm using clustering

We have seen plenty of clustering algorithms in this book and understood ways to tune them. To explain these, we used the Reuters dataset, which had around 20,000 documents, each having about 1,000 to 2,000 words. This scale was not challenging. In this chapter, we will use clustering to solve three interesting problems, on much larger datasets.

First, we will attempt to use the public "tweets" from Twitter (http://twitter.com) to try to find people who tweet alike using clustering. Second, we will examine a dataset from Last.fm (http://last.fm), a popular Internet radio website, and try to generate related tags from the data. Finally, we will take the full data dump of a popular technology discussion website, StackOverflow (http://stackoverflow.com/), which has around 500,000 questions and around 200,000 users. We will explore some interesting features for the website that could be thought of as clustering problems starting with clustering of users on Twitter.

12.1 Finding similar users on Twitter

Twitter is a social networking and micro-blogging service where people broadcast short, public messages known as "tweets". These tweets are at most 140 characters long. Once posted, a tweet is instantly published on the feed of all of the user's followers (and visible publicly on Twitter's site). These tweets are sometimes annotated with keywords in a style like "#Obama" or marked as directed at another user with a syntax like "@SeanOwen".

For the purpose of this example, we will use a scrape of Twitter data having around 10 million tweets available at http://www.public.asu.edu/~mdechoud/datasets.html. These 10 million tweets are from 118,000 unique users. The compressed data is around 520MB and available under the Creative Commons License for research purposes. It is a great dataset with which to illustrate the power of clustering and to explain some finer points in feature selection.

12.1.1 Data preprocessing and feature weighting

This data must be pre-processed, put into a format usable by Mahout. First it must be converted to the `SequenceFile` format that can be read by the dictionary vectorizer. Then this must be converted into vectors using TF-IDF as the feature weighting method.

The dictionary vectorizer expects its input to have a `Text` key and value. Here, the key will be the user's Twitter username, and value will be the concatenation of all tweets from that user.

Preparing this input is a job well suited to the MapReduce paradigm. A `Mapper` reads each line of the input: username, timestamp, and the tweet text separated by tabs. It outputs the username as key and the tweet text as value. A `Reducer` receives all tweets from a particular user and joins them into one

string and outputs this as the value, with the username again the key. This data is written into a `SequenceFile`.

The `Mapper`, the `Reducer`, and the `Configuration` classes, along with a `Job` that invokes them on a Hadoop cluster, are given in listings 12.1 and 12.2. The MapReduce classes are implemented as a generic group-by-field `Mapper`.

Listing 12.1 Group-by-field Mapper

```
public class ByKeyMapper extends Mapper<LongWritable,Text,Text,Text> {
  private Pattern splitter = Pattern.compile("\t");
  private int selectedField = 2;
  private int groupByField = 0;

  @Override
  protected void map(LongWritable key, Text value,
 Context context) throws IOException,
InterruptedException {
    String[] fields = splitter.split(value.toString());                #A
    if (fields.length - 1 < selectedField ||
        fields.length - 1 < groupByField) {
      context.getCounter("Map", "LinesWithErrors").increment(1);       #B
      return;
    }

    String oKey = fields[groupByField];
    String oValue = fields[selectedField];
    context.write(new Text(oKey), new Text(oValue));                   #C

  }

}
```

A Split each line using Regex Pattern
B Increment a counter for lines with data errors
C Output the selected username and tweet

Listing 12.2 Group-by-field Reducer

```
public class ByKeyReducer extends Reducer<Text,Text,Text,Text> {
  @Override
  protected void reduce(Text key,
                        Iterable<Text> values,
                        Context context) {
    StringBuilder output = new StringBuilder();
    for (Text value : values) {                              #A
      output.append(value.toString()).append(" ");           #B
    }
    context.write(key, new Text(output.toString().trim())); #C
  }
}
```

A Read each tweet by the user
B Generate the concatenated string from tweets
C Output the username and concatenated string

Once the job executes, the tweets of a user are concatenated into a document-sized string and written out into the `SequenceFile`. Next, this output must be converted to TF-IDF vectors. Before running the dictionary-based vectorizer, it is worthwhile to understand the Twitter data more and identify some simple and helpful tweaks, which will aid the feature selection and weighting process.

12.1.2 Avoiding common pitfalls in feature selection

TF-IDF is an effective feature weighting technique that we've used several times already. If applied to individual tweets, most all term frequencies would be 1 since the text is 140 characters in length, or perhaps 25 total words. We are interested in clustering users that tweet alike, so we have instead created a dataset where a document consists of all the tweets for a user, instead of individual tweets. Applying TF-IDF to this will make more sense and the clustering will happen more meaningfully.

Directly applying TF-IDF weighting may not produce great results. It would be better to remove features that commonly occur in all tweets: stop words. We will use the dictionary vectorizer and set the maximum document frequency percentage parameter to a low value of perhaps 50%. This removes words that occur in more than 50% of documents. We will also use the collocations feature of the vectorizer to generate bigrams from the Twitter data and use them as features for clustering.

However, there is a problem. Tweets are casually written messages, not carefully edited text like that in the Reuters news collection. They contain misspellings, abbreviations, slang, and other variations. Three people tweeting the sentences below will never get clustered together because their tweets have no words in common:

HappyDad7: “Just had a refreshing bottle of Loke”
BabyBoy2010: “I love Loca Kola”
Cool_Dude9: “Dude me misssing LLokkkeee!!!!”

To solve this problem, we can discover the relationship between these alternate “spellings” using phonetic filters. These are algorithms that reduce a word to a base form that approximates the way the word is pronounced. For example, “Loke” and “Loca” might both be reduced to something like “LK”. The word “refreshing” could be reduced to something like “RFRX”. There are at least three notable implementations of a procedure like this: Soundex, Metaphone and Double Metaphone. All are implemented in the Apache Commons Codec library[18]. We will be using its `DoubleMetaphone` implementation to convert each word into a base phonetic form. Listing 12.3 below shows a sample Lucene `Analyzer` implementation using `DoubleMetaphone` in this way. Keep in mind that this will only work for English text.

Listing 12.3 Lucence Analyzer class optimized for Tweets

```
public class TwitterAnalyzer extends Analyzer {
  private DoubleMetaphone filter = new DoubleMetaphone();          #A
  @Override
  public TokenStream tokenStream(String fieldName, Reader reader) {
    final TokenStream result = new PorterStemFilter(new StopFilter(
        true, new StandardTokenizer(Version.LUCENE_CURRENT, reader),
        StandardAnalyzer.STOP_WORDS_SET));

    TermAttribute termAtt = (TermAttribute) result
        .addAttribute(TermAttribute.class);
    StringBuilder buf = new StringBuilder();
    try {
      while (result.incrementToken()) {
        String word = new String(termAtt.termBuffer(), 0, termAtt
            .termLength());
        buf.append(filter.encode(word)).append(" ");               #B

      }
    } catch (IOException e) {
      e.printStackTrace();
    }
    return new WhitespaceTokenizer(new StringReader(buf.toString()));
  }
}
```

A Initialize a single instance of the encoder
B Encode the stemmed token into phonetic root

Copy the `SequenceFile` generated in section 12.1.1 to the cluster using the Hadoop HDFS copy command. Once the data is copied to the cluster, we can run the `DictionaryVectorizer` on the Twitter `SequenceFiles` as follows:

```
bin/mahout seq2sparse -ng 2 -ml 20 -s 3 -x 50 -w \
```

[18] http://commons.apache.org/codec/apidocs/org/apache/commons/codec/language/package-tree.html

```
-i mia/Twitter_seqfiles/ -o mia/Twitter-vectors -n 2 \
-a mia.clustering.ch12.TwitterAnalyzer -seq
```

It's important to make sure the class `TwitterAnalyzer` is in the Java CLASSPATH as shown above. Vectorization may take an hour to generate the complete set of bigrams and tokenize it. The vectorized representation will be smaller than the original input.

We don't know the number of clusters we need to generate. Since there are 118,000 unique users, we might decide to create clusters of 100 users each and hence generate around 1,180 clusters. Run K-Means algorithm using the vectors we generated. One of the clusters from the output is listed below:

```
Top Terms:
        KMK                     =>   7.609467840194702
        APKM                    =>  7.5810538053512575
        TP                      =>   6.775630271434784
        KMPN                    =>   5.795418488979339
        MRFL                    =>  4.9811420202255245
        KSTS                    =>   4.082704496383667
        NFLX                    =>   4.005469512939453
        PTL                     =>  3.9200561702251435
        N                       =>  3.7717770755290987
        A                       =>  3.4771755397319795
```

Here the feature "KMK" is the output of the `DoubleMetaphone` filter for words like "comic", "comics", "komic" and so on. To see the original words, run the clustering with vectors without using any of the filters, or simply using the `StandardAnalyzer`.

We can re-run clustering, two times more, with maximum document frequency set at 50% and at 20%. The following top-terms lists are selected from the cluster having tweets about the topic "comics" from each of these runs:

```
Top Terms with -maxDFPercent=50
        comic                   =>   9.793121272867376
        comics                  =>   6.115341078151356
        con                     =>   5.015090566692931
        http                    =>   4.613376768249454
        i                       =>   4.255820257194115
        sdcc                    =>   3.927590843402978
        you                     =>   3.635386448917967
        rt                      =>  3.0831371437419546
        my                      =>  2.9564724853544524
        webcomics               =>   2.916910980686997


Top Terms with -maxDFPercent=20
        webcomics               =>  11.411304909842356
        comic                   =>  10.651778670719692
        comics                  =>  10.252182320186071
        webcomic                =>   7.813077817644392
        con                     =>   5.867863954816546
        http                    =>   4.937416185651506
        companymancomic         =>   4.899269049508231
        spudcomics              =>   4.543261228288923
        rt                      =>   4.149479137148176
        addanaccity             =>   4.056342724391392
```

The two sets of output illustrate the importance of feature selection in a noisy dataset like tweets. With maximum document frequency set at 50% of the document count, there are features like "i", "you", "my", "http", "rt". These features dominate the distance measure calculations. So, they appear with higher weights in the cluster centroid. Decreasing the document frequency threshold removed many of these words. Still, there are words like "http", "rt" that appear in the second cluster, that should ideally be removed. However setting a much lower threshold runs the risk of removing important features. The best option usually is to manually create a list of words and prune them away using the Lucene `StopFilter`: words like "bit.ly", "http", "tinyurl.com", "rt" and so on.

This is clustering using only words to calculate user similarity. Similarity may also be inferred from user interaction. For example, if user A re-tweeted (forwarded, or repeated) user B's tweet, we might count B as a feature in A's user vector, with weight equal to the number of times A re-tweeted B's tweets. This may be a good feature for clustering users that think alike.

Alternately, if the feature weight were the number of times the users mentioned each other in a tweet, adding this as a feature might negatively affect the clustering result. Such a feature is best for clustering people into closed communities that tweet with each other and not for clustering users that think alike. The best feature for one problem may not be good for others.

The clustering result will show how various tech bloggers are clustered together and how various friend groups are clustered together. We will leave it to you to inspect the results and experiment with the clustering. Next, we will take a look at another interesting problem: a tag suggestion engine for a web 2.0 radio station.

12.2 Suggesting tags for artists on Last.fm

Last.fm is a popular Internet radio site for music. Last.fm has many features; of interest here is a user's ability to tag a song or an artist with a word. For example, the band "Green Day" could be tagged with "punk", "rock", or "greenday". The band "The Corrs" might be tagged with words like "violin" and "Celtic".

We will try to implement a related tags function using clustering. This could assist users in tagging items; after entering one tag, the system could suggest other tags that seem to "go together" with that tag in some sense. For example, after tagging a song with "punk" it might suggest the tag "rock". We will explore how to do this in two ways: first, with a simple co-occurrence-based approach, then with a form of clustering.

We will use as input a dataset that records the number of times each artists was tagged with a certain word. The dataset is available for download from the web at http://blogs.sun.com/plamere/entry/open_research_the_data_lastfm. It contains raw tag counts for the 100 most frequently occurring tags that Last.fm listeners have applied to over 20,000 artists. The dataset is of the format:

```
artist-id<tab>artist-name<tab>tag-name<tab>count
```

Each line in the dataset contains an artist ID and name, a tag name and the count of times that tag was applied to the artist.

12.2.1 Tag suggestion using co-occurrence

We can use co-occurrence, or the number of times two tags appear together on some artist, as the basis of a similarity metric between tags. We saw co-occurrence first in Chapter 6, where we counted item co-occurrence. With a similarity metric, we can then cluster.

So, for each tag, we could count co-occurrence with all other tags and present the most-co-occurring tags as suggestions. There is only one problem. A frequently-occurring tag like "awesome" will co-occur with many other tags, so it would becomes a suggestion for many tags, even though it in a sense is not a helpful suggestion. Therefore, we might have to cut out high-frequency tags.

This technique however, will never suggest tags that are two or more "hops" away. That is, if tag A co-occurs with B and B co-occurs with C, this method will not suggest tag C to the person adding the tag A. A workaround is to add all the tags, which are two hops away from tag A to the suggestion list and re-rank the tags.

But, clustering takes care of this out of the box. If the tags are within a certain distance threshold, regardless of whether they co-occur together, they are clustered.

We can model this problem as a user-based clustering where the user is the tag and the item or the feature is the artist they occur with. Alternately you can think of tags as documents, the artists as words in the document, and the number of times they co-occur as the word frequency.

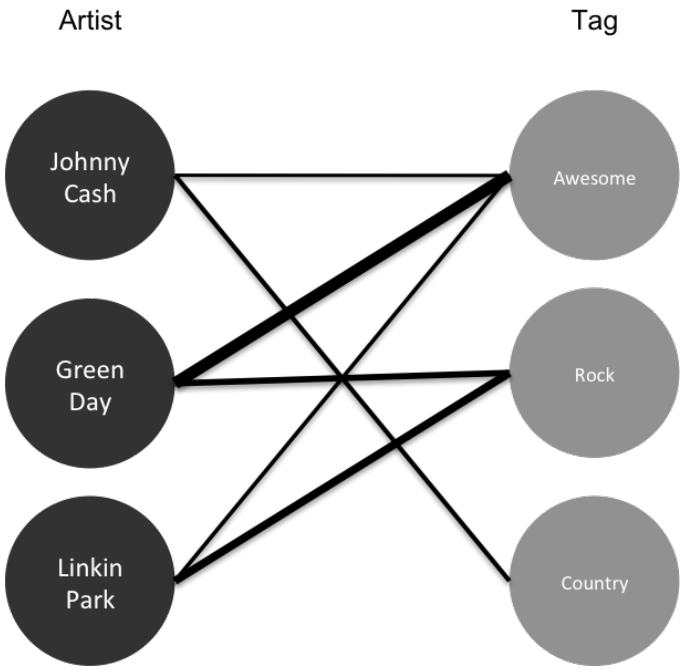


Figure 12.1 A typical bipartite (two partition). One type of nodes is for artists and the other for tags. The edges carry a weight equal to number of times tag was applied on the artist. In recommenders, these were users and items and star rating. Both forms of machine learning algorithms are more closely related than what many people realize.

The Last.fm dataset already looks like it can be directly converted to vectors. We will create a simple vectorizer, which first generates a dictionary of artists, and using the dictionary, convert the artists into integer dimensions in a MapReduce fashion. We could have used the strings as IDs, but Mahout `Vector` format only takes integer dimensions. The reason for doing this will be clear once we arrive at the code to vectorize Last.fm artists into vectors of their tags.

12.2.3 Creating a dictionary of Last.fm artists

To generate these feature vectors for the Last.fm dataset, we will employ two MapReduce jobs. The first generates the unique artists in the form of a dictionary and the second generates vectors from the data using the generated dictionary. The `Mapper` and `Reducer` class of the dictionary generation code is listed below in listing 12.4 and 12.5 below.

Listing 12.4 DictionaryMapper.java

```
public class DictionaryMapper extends
Mapper<LongWritable,Text,Text, IntWritable> {
  private Pattern splitter = Pattern.compile("<sep>")

  @Override
  protected void map(LongWritable key, Text line, Context context) {
    String[] fields = splitter.split(line.toString());
    String artist = fields[1];                                    #A

    context.write(new Text(artist), 0);                  #B
  }
}
```

A Select artist from the split line
B Emit the artist as the key

The Mapper takes a line of text and outputs the artist as key with an empty value. This is passed into the reducer as in listing 12.5.

Listing 12.5 DictionaryReducer.java

```
public class DictionaryReducer extends
Reducer<Text,IntWritable,Text,IntWritable> {
  @Override
  protected void reduce(Text artist,
                        Iterable<IntWritable> values,
                        Context context) {
    context.write(artist, 0);      #A
  }
}
```

A Reducer selects only the key

The `Mapper` simply emits the artist and the `Reducer` emits only a unique artist and discards the list of empty strings. Once the dictionary is generated, it can be read into a `HashMap`, which maps the words to a unique index as the value as shown below in listing 12.6. This is done sequentially.

Listing 12.6 VectorCreationJob.java

```
  …
  …
    Map<String,Integer> dictionary = new HashMap<String,Integer>();
    FileSystem fs = FileSystem.get(dictionaryPath.toUri(), conf);
    FileStatus[] outputFiles = fs.globStatus(
new Path(dictionaryPath, "part-*"));                    #A
    int i = 0;
    for (FileStatus fileStatus : outputFiles) {
      Path path = fileStatus.getPath();
      SequenceFile.Reader reader = new SequenceFile.Reader(fs, path,
          conf);
      Text key = new Text();
      Text value = new Text();
      while (reader.next(key, value)) {
        dictionary.put(key.toString(), Integer.valueOf(i++));   #B
      }
    }
  …
  …
```

A Select all files from the dictionary directory
B Read each unique tag and put it in the dictionary

This `HashMap` is serialized and stored in the Hadoop configuration. This serialized map goes into the next `Mapper`, which uses this as a dictionary to convert string tags to integer dimension IDs, as we will see next.

12.2.3 Converting Last.fm tags into Vectors with musicians as features

Using the dictionary generated above, we will create vectors of tags so that they can be clustered together. The vectorization process is a simple MapReduce Job. The `Mapper` selects the integer feature ID of an artist and creates a vector with a single feature. These partial vectors with one dimension are sent to the `Reducer`, which simply joins the partial `Vector` into a full `Vector`.

Listing 12.7 VectorMapper.java

```
public class VectorMapper extends
    Mapper<LongWritable,Text,Text,VectorWritable> {
  private Pattern splitter = Pattern.compile("<sep>");
  private VectorWritable writer = new VectorWritable();
  private Map<String,Integer> dictionary = new HashMap<String,Integer>();

  @Override
  protected void map(LongWritable key, Text value, Context context) {
    String[] fields = splitter.split(value.toString());

    String artist = fields[1];
    String tag = fields[2];                                  #A
    double weight = Double.parseDouble(fields[3

    NamedVector vector = new NamedVector(                    #B
        new SequentialAccessSparseVector(dictionary.size()), tag);
```

```
      vector.set(dictionary.get(artist), weight);                    #C
      writer.set(vector);

      context.write(new Text(tag), writer);                          #D
    }

    @Override
    protected void setup(Context context) {                          #E
      super.setup(context);

      Configuration conf = context.getConfiguration();

      DefaultStringifier<Map<String,Integer>> mapStringifier         #F
          = new DefaultStringifier<Map<String,Integer>>(
                conf, GenericsUtil.getClass(dictionary));
      dictionary = mapStringifier.fromString(conf.get("dictionary"));

    }
  }
```

A Select fields from the split
B Create a sequential and sparse vector
C Set the id from the dictionary with weight
D Output the partial vector with single feature
E Setup function runs before the Mapper is initialized
F Deserialize the HashMap file

The vector `Mapper` used as its `Vector` implementation the class `SequentialAccessSparseVector`. K-Means performs many sequential passes over the vector elements and this representation is best suited to that access pattern. Using the partial vectors from the `Mapper` as input, the `Reducer` simply concatenates them into a full vector as shown in Listing 12.7.

Listing 12.7 VectorReducer.java

```
public class VectorReducer extends
    Reducer<Text,VectorWritable,Text,VectorWritable> {
  private VectorWritable writer = new VectorWritable();

  protected void reduce(Text tag,
                        Iterable<VectorWritable> values,
                        Context context) {
    Vector vector = null;
    for (VectorWritable partialVector : values) {
      if (vector == null) {
        vector = partialVector.get().like();                 #A
      }
      partialVector.get().addTo(vector);                     #B
    }

    NamedVector namedVector = new NamedVector(vector, tag
        .toString());
    writer.set(namedVector);

    context.write(tag, writer);                              #C
  }
}
```

A Initialize, If not, an empty vector similar to the input
B Add all feature from the partial vector the full vector
C Output the full vector

The `VectorReducer` merges the partial vectors and converts them into a full `Vector`. The Last.fm artist vectors are ready. The next step is to run K-Means to generate the Topics.

12.2.5 Running K-Means over the Last.fm data

This process generates around 98,999 unique tags from the dataset of Last.fm. We might want to cluster this into small clusters having about 30. At first, we will try to create 2,000 tag clusters from K-Means as follows:

```
bin/mahout kmeans -i mia/lastfm_vectors/
-o mia/lastfm_topics -c mia/lastfm_centroids -k 2000 -ow
-dm org.apache.mahout.common.distance.CosineDistanceMeasure
-cd 0.01 -x 20
```

On one machine, it may take about 30 minutes to complete the iterations; since this is a CPU-bound process, adding more worker machines in the Hadoop cluster will improve its completion time. Once they have been output, the tag clusters can be inspected on the command line using the `seqdumper` utility:

```
Top Terms:
        Shania Twain                 =>   1.126984126984127
        Garth Brooks                 =>   0.746031746031746
        Sara Evans                   =>  0.6031746031746031
        Lonestar                     =>  0.5238095238095238
        SHeDaisy                     => 0.47619047619047616
        Faith Hill                   =>  0.4126984126984127
        Miranda Lambert              => 0.36507936507936506
        Dixie Chicks                 => 0.36507936507936506
        Brad Paisley                 =>  0.3492063492063492
        Lee Ann Womack               =>  0.3492063492063492

Top Terms:
        Explosions in the Sky        =>   55.19047619047619
        Joe Satriani                 =>   51.19047619047619
        Apocalyptica                 =>   50.95238095238095
        Steve Vai                    =>  43.666666666666664
        Mogwai                       =>   39.57142857142857
        Godspeed You! Black Emperor  =>   32.00100000000101
        Yann Tiersen                 =>  30.857142857142858
        Pelican                      =>  20.285714285714285
        Do Make Say Think            =>  16.904761904761905
        Liquid Tension Experiment    =>   16.61904761904762

Top Terms:
        Justin Timberlake            =>   6.739130434782608
        Disturbed                    =>   0.391304347826087
        Timbaland                    => 0.34782608695652173
        Nelly Furtado                => 0.30434782608695654
        Lustans Lakejer              =>  0.2608695652173913
        Michael Jackson              =>  0.2608695652173913
        Aaliyah                      => 0.21739130434782608
        Timbaland Magoo              => 0.21739130434782608
        Jonathan Davis               => 0.21739130434782608
        Sum 41                       => 0.21739130434782608
```

These are top artists for a given tag. These tags could be stored in a database and later queried to produce suggestions for any tag. This example showed the trick of transforming a dataset to a different view in order to apply clustering. Next example is a bit more involved and on a much larger and open dataset of a popular Q&A website: StackOverflow.com

12.3 Analyzing the StackOverflow dataset

StackOverflow[19] (http://stackoverflow.com) is an online discussion website hosting questions and answers on a wide range on programming-related topics. Users ask and answer questions, as well as vote for or against questions and answers. Users earn reputation points for interacting on the website. For example, a user gets 10 points upon receiving an "up" vote on an answer given to a question. After accumulating enough points, users earn additional abilities on the site.

Thanks to StackOverflow, all data on the website is freely available for non-commercial research. Therefore, we will explore StackOverflow as a dataset and some of the interesting problems that can be modeled as clustering: like clustering questions to find related questions or clustering users to find related users or clustering similar answers to make the answers more readable.

[19] http://www.stackoverflow.com

12.3.1 Parsing the StackOverflow dataset

The entire data of the stackoverflow.com site and its two sister sites, superuser.com and serverflow.com, are available as a single file at http://blog.stackoverflow.com/category/cc-wiki-dump. The whole dataset is around 700MB, compressed. All three datasets are similar in nature, with StackOverflow being the largest and most prominent of the three. Uncompressed, the StackOverflow dataset is around 3.5GB in size split across multiple XML files. There are separate files for posts, comments, users, votes, and badges.

Parsing XML files can be tricky, especially in a Hadoop MapReduce job. Mahout provides `org.apache.mahout.classifier.bayes.XMLInputFormat` for reading an XML file in a MapReduce job. It correctly detects each repeated block of XML and passes it to the `Mapper` as an input record. This string will have to be further parsed as a XML using any XML parsing library.

The `XMLInputFormat` needs the starting and ending tokens of the XML block. This is achieved by setting the following keys in the Hadoop `Configuration` object:

```
…
conf.set("xmlinput.start", "<row Id=");
conf.set("xmlinput.end", " />");
…
conf.setInputFormat(XmlInputFormat.class);
…
```

In the `Mapper`, set the key as `LongWritable` and value as `Text` and read the entire XML block as `Text` in the value including the start and end.

12.3.2 Finding clustering problems in StackOverflow

StackOverflow dataset contains a table of users, a table of questions, and a table of answers. There are relations between them, based on the fields user-id, question-id and answer-id, that helps us to sift and transform the data in which ever way we like. Some of the interesting clustering problems on this data follow.

CLUSTERING POSTS DATA FOR SURFACING RELATED QUESTIONS

It's useful to identify questions that concern similar topics. Users browsing a post that doesn't quite address his or her question might find a better answer among a list of related questions. This process of finding similar questions will be like that which was applied to the Reuters dataset. These are smaller than Reuters articles but larger than Twitter tweets. So, TF-IDF weighting ought to give good features. It is always good to remove high frequency words from the dataset so keep the maximum document frequency threshold at 70% or even lower.

One big challenge we will encounter while vectorizing this dataset is the lack of good tokenization of the StackOverflow questions. Many questions and answers have snippets of code from different programming languages, and the default `StandardAnalyzer` is not designed to work on such data. We will have to write a parser that takes care of braces and arrays in code and other quirks in different programming languages.

Instead of using just the question, we can bundle together both questions and the answers, as well as comments, to produce larger documents and get more features to cluster similar questions. Unlike Twitter, the spelling mistakes should not harm the quality so much due to the larger size of the content. However, adding a `DoubleMetaphone` filter would give slight improvement in quality. Since there is lot of data, both K-Means and Fuzzy K-Means would yield similar results. A better improvement in quality will only come by using LDA topics as features, but the CPU consumed by LDA may be prohibitively high for this data set.

CLUSTERING USER DATA FOR SURFACING SIMILAR USERS

Imagine you are a developer who extensively uses, for instance, the Java Messaging Service (JMS) API. It might be useful to discover people who work with JMS as well. Helping users form such communities not only improves the user experience on the website, but also drives user engagement. Again, clustering can help compute such possible communities.

For clustering users, we need a vector from the features of the user. This could be the kind of content the users posts or answers, or the kind of interaction he has with the rest of the users. Some of the features of the vector are:

- The contents of the questions or answers he or she created, which includes the n-grams from text and code snippets
- The other users who replied or commented on the post he or she created and interacted on:

Clustering users could be done using just the post content, or using just the common interaction counts, or both. Previously, when clustering tweets, only the content was used. It would be a good exercise to experiment on using the interaction features to cluster users.

In this model, a user is a document and its features are the IDs of other users who he or she interacted with, and the feature value corresponds to how much that other user interacted with him or her. This can be weighted. For example, if user 2 commented on user 1's post, this might map to a weight of 10. However, if user 2 voted on user 1's post, it might just be weighted a 2. If the vote was negative, it might receive a weight of -10 to indicate that these two users have a lower probability of wanting to interact with each other. Think of all possible interactions and assign weights to it. The weight of the feature user 2 in the vector of user 1 would be the cumulative weight of the interaction they did. With these vectors and features, try-clustering users just based on their interaction patterns and see how it works out.

12.5 Summary

In this chapter, we examined real world clustering through analysis of three datasets: Twitter, Last.fm, and Stack Overflow.

We started with the analysis of tweets by trying to cluster users that tweet alike. We preprocessed the data, converted it to vectors, and used it to successfully cluster users by their similarity in tweets. Mahout was able to scale easily to the data having 10 million tweets due to the MapReduce-based implementations.

We took a small subset of a dataset based on Last.fm tags and tried to generate a tag suggestion feature for the site. We modeled the problem as a bi-partite network of music artist on one part and user generated tags on the other with the edge weights being the number of time they co-occur together. We clustered based on tags as well. Using this output it was easy to generate the tag suggestion functionality.

Finally we examined data from StackOverflow. We discussed solving several real life problems on such a discussion forum with only clustering.

This concludes the coverage of clustering algorithms in Mahout. It has been intended as a gentle introduction to one aspect of machine learning, which started from understanding the basic structures and mathematics to solving large-scale problems on Hadoop cluster.

Now, we will move to discuss classification with Mahout, which entails more complex machine learning theory and some solid optimization techniques both distributed and non-distributed to achieve state of the art classification accuracy and performance.

13

Introduction to classification

This chapter covers

- Why Mahout is a powerful choice for classification
- Key ideas and terminology to help you get started quickly
- The workflow of a typical classification project
- A step-by-step classification example

Life often presents us with questions that are not open-ended but instead ask us to choose from a limited number of answers. This relatively simple idea forms the basis for classification, both that done by humans and done by machines. Classification relies on categorization of potential answers, and machine-based classification is the automation of such simplified decisions. The chapters in this part of the book describe how to build and train a Mahout classifier, how to use evaluation to fine-tune a classification model and how to get a classification service ready for use in production.

This chapter identifies those times when Mahout is a good approach for classification and why it offers an advantage. This chapter also explains the key ideas that help you understand what classification is. It presents the three stages in the workflow for typical classification projects and provides a step-by-step simple classification project that will let you put these basic ideas into practice.

13.1 What is classification?

Before we look at the way Mahout-based classification is done, let's delve a little deeper into the basics of what classification is, how it differs from recommendation and clustering, and the answer to the question, "Why use Mahout as your classification approach?"

To better understand what classification is, first consider familiar examples of human-based classification. Medical histories, surveys about the quality of service of a particular business, or checking off the correct box for our filing status on tax forms are all choices we make from a limited group of pre-selected answers. Sometimes the number of categories for our responses is very small, even as few as two in the case of yes/no questions. In these situations, we reduce what are potentially large and complicated responses to fit into a small number of options.

> Definition: Classification is a decision based on specific information (input) that gives a single selection (output) chosen from a short list of pre-determined potential responses.

Compare what happens when someone goes to a wine tasting to the experience in a grocery store when a person is getting wine to take to a dinner. At the wine tasting, people will ask, "What do you think of that wine?" and a wide range of answers are likely to be heard: "peppery," "overtones of fruit", "a rich finish", "sharp" "the color of my aunt's sofa" and "nice label". The question is non-specific, and the answers (output) do not form a single selection from a pre-determined short list of possibilities. This situation as stated is not classification, though it may have some similarities to classification.

Now consider the wine shopper in grocery store. Some of the same wine characteristics might matter in general to the shopper (taste, color, and so on) but in order to efficiently make the specific decision (buy/not buy) and get to dinner, most likely she or he has a mental checklist of characteristics to look for (red, $10, good with pizza, a label that is familiar) when examining each bottle, and for each bottle, makes the decision (buy/don't buy). Some detail and nuance may be lost, but simplifying the question to one that is very specific and has very few potential answers (buy, don't buy) is a practical way to make many decisions manageable. In this case, it gets the person out of the store in time for dinner. This situation is a form of classification.

Machines cannot do all the types of thinking and decision-making that humans do. When the situation is limited within specific boundaries, machines can emulate human decisions efficiently in order to carry out classification. This can work when there is a single, focused decision where the input to be considered consists a specific set of characteristics, and the output is categorical, that is to say, a single selection from a short list.

Classification algorithms are the heart of what is called predictive analytics. The goal of predictive analytics is to build automated systems that can make decisions that replicate human judgment, and classification algorithms form a fundamental tool to meet that goal.

One example of predictive analytics is spam detection. A computer uses the details of user history and features of email messages to determine whether new messages are spam or relatively welcome email. Another example is credit card fraud detection. A computer uses the recent history of an account and the details of the current transaction to determine whether the transaction is fraudulent.

How do automated systems do this? We would prefer that they do it magically, but in lieu of that, we rely on having them learn by example.

> definition: Classification systems are a form of machine learning that uses learning algorithms to provide a way for computers to make decisions based on experience and in the process to emulate certain forms of human decision making.

Classification algorithms learn by example, but they do not provide a general substitute for human judgment largely because they require carefully prepared examples of correct decisions from which to learn and require inputs that are carefully prepared so as to be tractable for the machine learning techniques used to implement these algorithms. Unlike some other uses of Mahout, machine-based classification is a form of supervised learning.

13.1.1 Differences between classification, recommendation and clustering

Classification algorithms stand in contrast to the clustering algorithms described in previous chapters because the clustering algorithms are able to decide on their own what distinctions appear to be important (in machine learning parlance, they are unsupervised learning algorithms) while classification algorithms learn to mimic examples of correct decisions (they are supervised learning algorithms). Classification algorithms are also different from recommendation algorithms. Classification algorithms are intended to make a single decision with a very limited set of possible outcomes while recommendation algorithms select and rank the best of very many alternative possibilities.

> Warning: One way to think about what classification *is* is to remind yourself of what classification *is not*: If your system is unsupervised, it *is not* classification. If your questions are open-ended or the answers are not categorical, your system *is not* classification.

The chapter has begun to answer the question, "What is classification?" Part of the answer lies in understanding what classification is used for.

13.1.2 Applications of classification

The output of a classification system is the assignment of data to one of a small pre-determined set of categories. The usefulness of classification is usually determined by how meaningful the pre-chosen categories are. By assigning data to categories, the system allows the computer or a user to act on the output.

Classification is often used for prediction or detection. In the case of credit card fraud, a classifier can be useful in predicting what transactions will later be determined to be fraudulent based on known examples of behavior for purchases and other credit card transactions. The yes/no decision of whether or not a particular transaction is fraudulent is made by assigning that transaction to one of two categories: yes, fraud or no, not fraud.

Classification is also valuable in predicting customer attrition in industries such as insurance or telecommunication. One way to design a classification system for this application is to train on data from a previous year's records for known cases of attrition or retention, which are the categories being targeted as answers. Known characteristics of customers included in the training data set are selected as the features that can best predict whether or not a customer will stay with the service. Once trained, the classification system can be used to predict the future behavior of other customers based on the known, specific characteristics, including current behavior.

Note: Keep in mind that prediction does not refer solely to the ability to foresee future events.

Prediction can be the task of correctly assigning a characteristic (a category) based on available data rather than by directly testing for the characteristic in question. Often the characteristic that is targeted for classification is itself too expensive to be determined directly. By expensive, we can mean too costly in money, too expensive in time or too dangerous.

For example, you may not want to wait a year or more to see telltale symptoms of a progressive illness such as diabetes. Instead, you might build a classification system for early indicators of the disease that will assign risk in the future without having to wait until possibly irreversible damage has occurred. Knowing a patient is pre-diabetic or likely to develop diabetes can provide options for prevention or early treatment.

Tip: In general, classification is often a way of determining the value of an expensive variable by exploiting a combination of cheap variables.

In related situations, a classification system can reduce physical danger. It is far preferable to be able to correlate data obtained by noninvasive means, such as cognitive tests or PET scans, as a way to determine the existence of Alzheimer's disease than by direct testing through autopsy on the brain, especially while the patient is still alive.

Classification is one tool used in a generally computations way that may be glued into larger procedures. For example, classification can be used as a part of a recommendation system. The value of classification for certain situations is fairly obvious. Now it is useful to think about the value of Mahout as your classification approach.

13.1.3 Why use Mahout for classification?

Mahout can be used on a wide range of classification projects, but the advantage of Mahout over other approaches becomes striking as the number of training examples gets extremely large. What "large" means can vary enormously. Up to about 100,000 examples, other classification systems can be efficient and accurate. But generally as the input exceeds 1 to 10 million training examples, it will begin to need something scalable like Mahout. Table 13.1 shows the situations in which Mahout is most appropriate.

Table 13.1 Mahout is most useful with extremely large or rapidly growing data sets where other solutions are least feasible.

System size in number of examples	Choice of classification approach
< 100,000	Traditional, non-Mahout approaches should work very well. Mahout may even be slower for training.
100,000 to 1 million	Mahout begins to be a good choice. The flexible API may make Mahout a preferred choice even

	though there is no performance advantage.
> 10 million	Mahout excels where others fail

The reason for this advantage with Mahout is that as the input data increases, the time or memory requirements for training may not increase linearly in a non-scalable system. A system that slows by a factor of two with twice the data may be acceptable, but if using five times as much data input results in the system taking 100 times as long to run, another solution must be found.

In general, the classification algorithms in Mahout require resources that increase no faster than the number of training or test examples, and in most cases, the computing resources required can be parallelized. This property allows you to trade-off the number of computers used versus the time the problem takes to solve. Figure 13.1 illustrates the advantages of scalable algorithms such as those in Mahout for medium to very large-scale classification projects.

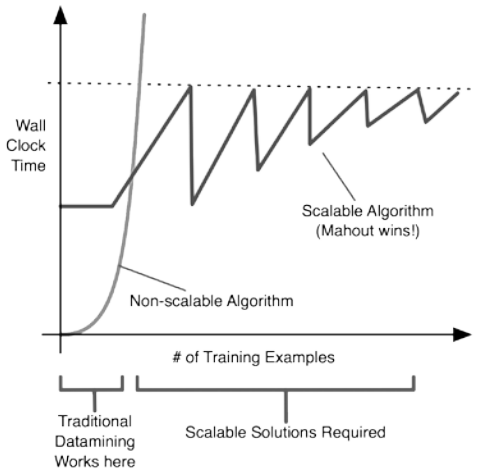


Figure 13.1 The advantages of Mahout for medium and large classification systems. The jagged line shows the effect of recruitment of new machines to increase performance. Each new machine decreases training time.

The conceptual graph in figure 13.1 shows that when the number of training examples is relatively small, traditional data mining approaches work as well or better than Mahout. But as the number of examples increase, the scalable and parallel algorithm shown for Mahout is better with regard to time.

The increased time required by non-scalable algorithms is often due to the fact that they require unbounded amounts of memory as the number of training examples grows. Extremely large data sets are becoming increasingly widespread. With the advent of more electronically stored data, the expense of data acquisition can decrease enormously, making extremely large data sets more readily available. Increased data for training is desirable because it typically improves accuracy. As the number of large data sets that require scalable learning increases, the scalable classifiers in Mahout are becoming more and more widely useful.

13.2 How classification works

This section introduces the basics of what a model is and how it is trained, tested and used in production. You will be able to distinguish predictor variables from target variables, understand how to correctly make use of records, fields and value, and know the four forms of values variables can take. In addition, you will see that Mahout classification is an example of supervised learning. The subsections define and expand the explanation of the key ideas.

With Mahout's classifiers, as with classifiers in general, there are two main phases involved in building a classification system: the creation of a model produced by a learning algorithm, and the assignment of new data to categories using the model. The selection of training data, output categories (the target), the

algorithm through which the system will learn, and the variables used as input are key steps in the first phase of building the classification system.

The basic steps in building a classification system are illustrated in figure 13.2.

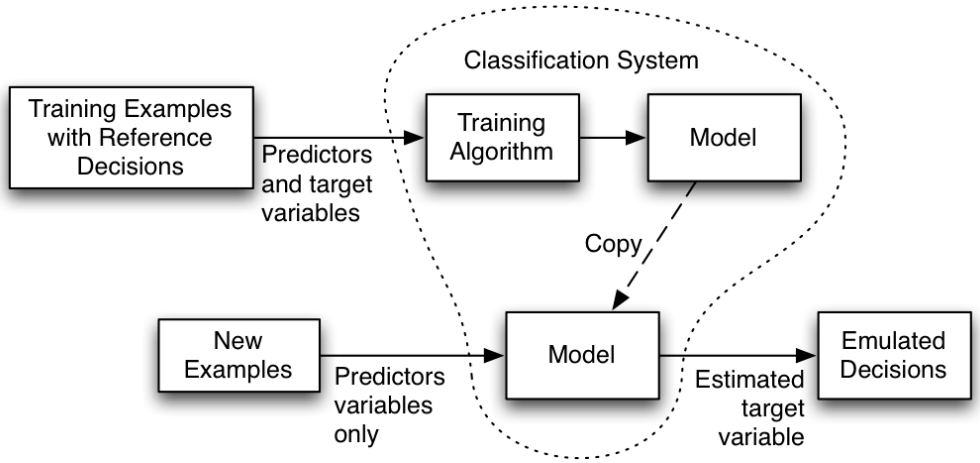


Figure 13.2. How a classification system works. Inside the dotted lasso is the heart of the classification system, a training algorithm that learns a model to emulate human decisions. A copy of the model is then used in evaluation or in production with new input examples to estimate the target variable.

The figure shows two phases of the classification process, with the upper path representing training the classification model and the lower path providing new examples for which the model will assign categories (the target variables) as a way to emulate decisions. For the training phase, input for the training algorithm consists of example data labeled with known target variables. The target variables are, of course, unknown to the model when using new examples during production. In evaluation, the value of the target variables are known, but will not be given to the model. In production, the target variable values are not known, which is why the model is built in the first place.

For testing, which is not shown explicitly in figure 13.2, new examples from held-out training data are presented to the model. The results chosen by the model are compared to known answers for the target variables in order to evaluate performance, a process described in depth throughout chapter 15.

The terminology used by different people to describe classification is highly variable. For consistency, we have limited the terms used for key ideas in the book. Many of these terms are provided in table 13.2. Note the relationship between a record and a field: the record is the repository for values related to a training example or production example, and a field is where the value of a feature is stored for each example.

Table 13.2 Terminology for the key ideas in classification

Key idea	Description
Model	A computer program that makes decisions; in classification, the output of the training algorithm is a model
Training Data	Subset of training examples labeled with the value of the target variable and used as input to the learning algorithm to produce the model
Test Data	Withheld portion of training examples given to the model without the value for the target variable (although the value is known) and used to evaluate the model
Training	Learning process that uses training data to produce a model. That model can then compute estimates of the target variable given the predictor variables as inputs.

Training example	Entity with features that will be used as input for learning algorithm
Feature	A known characteristic of a training or new example; a “feature” is equivalent to saying a “characteristic”.
Variable	In this context, a variable is equivalent to a the value of a feature or a function of several features. This usage is somewhat different from a normal variable in a computer program.
Record	A container where an example is stored; such a record is composed of fields.
Field	Part of a record that contains the value of a feature (variable)
Predictor variable	Feature selected for use as input for a classification model. Not all features need be used. Some features may be algorithmic combinations of other features.
Target variable	Feature that the classification model is attempting to estimate: the target variable is categorical and its determination is the aim of the classification system.

13.2.1 Models

A classification algorithm learns from examples in a process known as training. The output of this training process is called a model. This model is a function that can then be applied to new examples in order to produce outputs that emulate the decisions that were in the original examples. These emulated decisions are the end product of the classification system.

note The model produced by the training algorithm is itself effectively a computer program.

Figure 13.2 shows how training examples, complete with desired output values, are given to the learning part of the classification system to get a model. This model, which itself can be viewed as a computer program that makes decisions, is then copied and given new examples as input. The intention is that the model will classify the new examples to emulate the training examples used as input.

13.2.2 Training versus test versus production

Look again at the training examples shown in figure 13.2 that are used to produce a model. In practice, the training examples are typically divided into two parts. One part, known as the training data, typically consists of 80-90% of the available data. The training data are used in training to produce the model. A second part, called the test data, is then given to the model without telling it the desired answers, although they are known. This is done in order to compare the output of the model with the desired output.

Once the model performs as desired, it is typically put into production and given additional examples to classify for which the “correct” decision is unknown. Production results are usually not 100% accurate, but the quality of the output should stay consistent with what was achieved during the test run(s) unless the quality of the input data or external conditions changes.

It is common to take samples of these production examples over time in order to make more training data so that updated versions of the model can be produced. This sampling of production examples is important if external conditions might change such that the quality of the decisions made by the model begins to degrade over time. In that case, a new model can be produced and the quality of the decisions can be improved or restored to original levels.

13.2.3 Predictor variables versus target variable

A variable is a value for a feature or characteristic of an example derived by measurement or by computation. The ultimate goal of a classifier is to estimate a categorical answer to a specific question, and this answer is known as the target variable. In production, the target variable is the value being sought for each new example. In training, the target variable is known for historical data (training data) and is used to train the model.

.In classification, the predictor variables are the “clues” given to the model that are used to decide what target variable to assign to each example. Predictor variables used for classification are also known as input variables or predictors.

The characteristics of the examples to be classified can also be called features; the process of describing a particular characteristic in a way that can be used by a classification system is known as feature extraction. If the feature is chosen to be used as input for the model, the value of that feature would then be thought of as a predictor variable.

Recall the situation of the shopper buying wine for dinner. Attributes of the wine, such as “color”, “good with steak”, “good with fish”, “good with pizza” or other attributes such as price are used by the shopper to make a decision to buy or not to buy. In the purchase decision, these features of the wine are analogous to predictor variables input to the model in machine classification. In this analogy, the decision has a binary target variable, buy/don’t buy; only one of these choices will be selected for each bottle.

> Note: Both the target and predictor variables are given to the learning algorithm during training. In contrast, during test phase and in production, only the predictor variables are available to the model.

In preparation of the classification system the bulk of historical examples that are used as training data will be labeled with the value of the target variable for each example. During testing the target variables known for the withheld examples used to test the model test data will be cloaked as they are used for input to the training algorithm. A comparison of the estimated target values as produced by the model, with the known values for the target variable associated with each test example will reveal the accuracy of the classification model.

> TIP For classification, the target variable (field) must have a categorical value. The value for predictor variables can be continuous, categorical, text-like or word-like.

You can see the difference between these types of values (continuous, categorical, text-like or word-like) in section 13.2.5. Often the target variable is a binary categorical variable, meaning it has only two possible values. Spam detection is an example of a classification system with a binary categorical variable. In other cases, there could be more than two possible values for the target variable. This situation is called multi-class classification. An example of multi-class classification using 20 newsgroups data is presented in chapter 14, sections 14.4 and 14.6.

13.2.4 Records, fields and values

The examples that comprise the input for classification algorithms are normally represented in the form of records. The record can be thought of as the repository for the values that represent the training example or new examples used in production. Records typically consist of a collection of named fields, each of which has a value.

The record for each training example will have more than one field to store values for the target variable and for one or more features chosen as predictor variables. The field for the target variable in a production example is, of course, the “unknown” value that is being sought by the classification algorithm. Figure 13. 3 illustrates the idea that values for target variable and predictor variables associated with each training example and each production example are stored in a record, shown in this illustration as a box divided into fields. The output of the model is the value estimated for the target variable denoted in the diagram by a box containing just the T for target. There is one output for each input example.

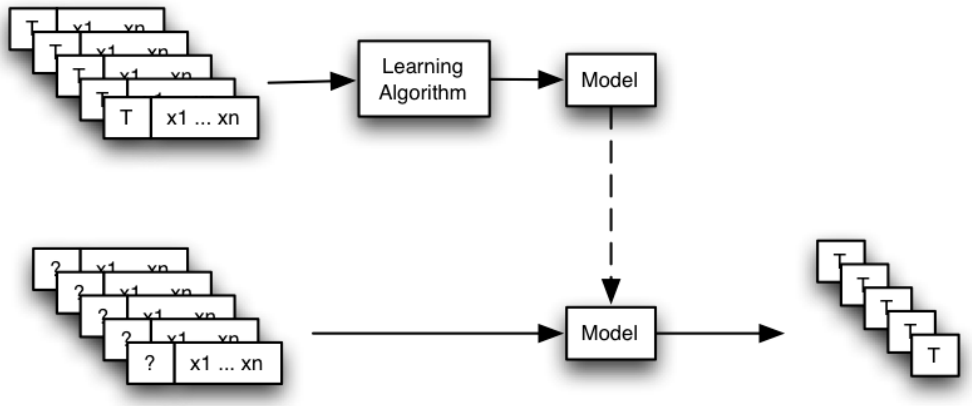


Figure 13.3 Input and output of a classification model. “T” stands for target variables; “x1...xn” indicate predictor variables. Notice that T is included with input data during training but absent from data input during production. The missing value for target variable in a production example is indicated by “?”.

Although target values can only be categorical, there is more than one type of value that can represent features used as predictor variables. We present a comparison of the different types of values for predictor variables in 13.2.5.

13.2.5 The four types of values for predictor variables

Predictor variables serve as the input for the learning algorithm, but which ones are useful depends in part on the type of value used to represent them. Details of which types of predictor values are used by a particular algorithm are covered in chapter 14, section 14.5. There generally are four types of values of predictor variables: continuous, categorical, word-like and text-like, as shown in table 13.3.

Table 13.3 Four common types of values used to represent features.

Type of Value	Description
Continuous	This type of value is a floating point value. This might be a price, a weight, a time, a value or anything else that has a numerical magnitude and where this magnitude is the key property of the value.
Categorical	A categorical value can have one of a pre-specified set of values. Typically the set of categorical values is relatively small and may be as small as two, although the set can be quite large. Boolean values are generally treated as categorical values. Another example might be a vendor id.
Word-like	A word-like value is like a categorical value, but it has an open-ended set of possible values.
Text-like	A text-like value is a sequence of word-like values, all of the same kind. Text is the classic example of a text-like value, but a list of email addresses or URL’s is also text-like.

The distinction between continuous and categorical values is often difficult. It is a common mistake to treat numeric identifiers codes as continuous values when they should really be considered categorical values. Consider, for instance, this list of values that represent zip codes (first three digits): 757, 415,

215, 809, 446. These values look numerical, and you might be tempted to consider them as continuous, but they better fit the description of categorical because each is one from a set of pre-specified values. Treating a value as continuous when it is really categorical (or vice versa) can seriously impair the accuracy of a classifier.

> tip Things that look like integers are not necessarily continuous. A key test is to imagine adding two values together or taking the log or square root of a value. If this does not make any sense, then you probably have a categorical value rather than a continuous one. Another hint is that anything with an associated unit of measurement is usually a continuous variable.

A word-like value is an extension of categorical values to the case where there are many possible values. The distinction between these two types hinges on how many values a feature could have. Even if a feature could have many values, is there a finite group of pre-specified values? In cases where you cannot *a priori* say how many possible values there are, the variable likely is word-like. It is often difficult to decide whether something should be treated as word-like or categorical. For instance, the combined make and model of a car might seem categorical, but if there is a good chance that a new make or model of a car could appear at any time so treating such a value as word-like might be a better choice.

In the occasional case in which categorical values have thousands or more possible, they are often better treated as word-like. If you already have many possible values, it is quite plausible that the set of values might not be as fixed as you might think at first. Occasionally, categorical values might have some ordering, but the classification algorithms in Mahout do not currently pay attention to such ordering.

If you have a variable whose values can be viewed as collections of word-like values, then you should consider it to be text-like, even if only a few examples might have more than one such value. Text-like values are often stored as strings and thus need a tokenizer that can convert the string representation as a sequence of word-like values.

> tip: A text-like variable can often be spotted by the fact that the components of the value can be appear in almost any combination. For this reason, a variable that could have values “General Motors”, “Ford” or “Chrysler” is not text-like even though “General Motors” appears to have two words in it. This is because values like “General Chrysler Ford” do not make sense.

Keep in mind the descriptions in table 13.3 as you examine the group of examples from difference data sources and the type of value for each one that are shown in table 13.4. The first value, the address of the sender of an email is a single entity from an open-ended set and thus is word-like. The values for spam-words or non-spam words comprise a list of words (or possible words) and are considered text-like.

Table 13.4 Sample data that illustrate all four value types. These examples are typical of features of email data. Values cover the range of all four types of predictor variables.

Name	Type	Value
from-address	word-like	`George <george@fumble-tech.com>`
in-address-book?	categorical(TRUE, FALSE)	TRUE
non-spam-words	text-like	`“Ted”, “Mahout”, “User”, “lunch”`
spam-words	text-like	`“available”`
unknown-words	continuous	0
message-length	continuous	31

One of the distinctions between classification and other Mahout capabilities such as clustering or recommendation is based on whether learning is supervised or unsupervised. Section 13.2.6 explains this terminology.

13.2.6 Supervised versus unsupervised learning

Classification algorithms are related to, but still quite different from clustering algorithms such as the k-means algorithm described in previous chapters in that classification algorithms are a form of supervised learning as opposed to unsupervised learning such as with clustering algorithms. A supervised learning algorithm is one that is given examples that contain the desired value of a target variable. Unsupervised algorithms are not given the desired answer, but instead must find something plausible on their own.

Supervised and unsupervised learning algorithms can often be usefully combined. A clustering algorithm can be used to create features to be used by a learning algorithm or the output of several classifiers can be used as features by the clustering algorithm. Moreover, clustering systems often build a model that can be used to categorize new data that works much like the model produced by a classification system. The difference lies in what data were used to produce the model. For classification, the training data include the target variables; for clustering, the training data do not include target variables.

13.3 How classification works: Work flow in a typical project

Regardless of the variation between problems, there is a pretty standard set of steps that need to be followed to build and run a classification system. This section of the chapter provides an overview of the typical workflow in classification projects, which falls into three stages: training the model, evaluating and adjusting the model to an acceptable level, and running the system in production. We introduce a toy example about color-fill in symbols using synthetic data as a way to illustrate some of the steps in the workflow. We then use an example of this type in section 13.4 to carry out a practice classification project using Mahout.

As a pre-requisite to any project, you need to identify the information you are seeking (target variable) as a feature that can be stored in appropriate form in a record and that meets the overall goal, such as identifying a fraudulent financial transaction. Sometimes you have to adjust what you choose as a target variable based on pragmatic considerations such as expense for training, privacy issues and so on. You also need to know what data are available for training and what new data will be available for running the system in order to design a useful classifier.

> Note: It is common for most of the effort required to build a classifier to be spent inventing and extracting useful features. It is also common to massively underestimate just how large the effort for this step will be.

In each stage of the process, the accuracy of the system and the usefulness of the output largely reflect the choices made originally in the selection of particular features to be used as predictors and in the categories chosen for the target variable. There is not one single, “correct” selection for each of these to be made: a variety of approaches and adjustments are possible and usually required to build an efficient and robust system. These general steps are shown in table 13.5.

Table 13.5 Workflow in a typical classification project

Stage	Step
1. Training the model	Define target variable
	Collect historical data
	Define predictor variables
	Select a learning algorithm
	Use learning algorithm to train model
2. Evaluating the model	Run test data
	Adjust input (different predictor variables and/or algorithm)

3. Using model in production	Input the new examples to estimate unknown target values
	Retrain model as needed

Each step in the process of training and evaluating the model requires a balance between selecting features for variables that are practical and affordable in cost and time and between getting an acceptable level of accuracy in the values estimated by the model.

Now we discuss what happens during the first stage of classification, training.

13.3.1 Workflow for stage 1: training the classification model

In the first stage of a classification project, you will need to have in mind a target variable of interest, and this will in turn help you select appropriate historical data for the training process and select a useful learning algorithm. These decisions go hand-in-hand. The detailed discussion that follows examines a few of the specific ways in which the choice about features influences which Mahout learning algorithms will work in your classifier.

DEFINE CATEGORIES FOR THE TARGET VARIABLE

Defining your target variable means you must define the categories for your target variable. The target variable cannot have an open-ended set of possible values. This decision, in turn, also affects your choices for learning algorithms since some algorithms are limited to binary target variables. Having fewer categories for the target variable is definitely preferable, and if you can pare the categories down to just two values, you have a few more options available to you for learning algorithms. If the target variable resists efforts to reduce it to a simple form, it may be that you need to build several classification systems, each possibly deriving some aspect of your desired target variable. In your final system, you can then combine the outputs of these classification systems in order to get the fully nuanced decision or prediction that you need.

COLLECT HISTORICAL DATA

The choice of your source for historical data is directed in part by the need to collect historical data with known values for the target variable. For some problems, there will be difficulties in determining exactly what the value of the target variable should be.

The old saying about garbage in leading to garbage out is particularly apt at this point, and great care should be taken to make sure that the value of the target variable is known accurately for the historical data. Many models have been built that faithfully replicate a flawed target variable or data collection process, and you do not want to build another one.

DEFINE PREDICTOR VARIABLES

After you have selected a useful target variable and defined the set of values it can take on, you need to define your predictor variables. These variables are the concrete encoding of the features extracted from the training and test examples. Once again you will need to examine your source of examples to make certain that they have useful features with values that can be stored in appropriate format for their fields in the record for each example. As you saw illustrated in figure 13.3, the predictor variables appear in records for the training/test data and for the production data.

> WARNING: A common mistake in selecting and defining predictor variables is to include what is known as a target leak in the predictor variables.

An important consideration in defining the predictor variables is to avoid setting up a target leak. A target leak, as the name implies, is a bug that involves providing data about the target variable unintentionally in the design of predictor variables. This bug is not to be confused with intentionally including the target variable in the record of a training example.

Target leaks can seriously affect the accuracy of the classification system. This problem can be relatively obvious when you tell the classifier that the target variable is a predictor, an error that seems obvious but still happens distressingly often. Target leaks can be subtler as well. Suppose in building a spam detector that you keep the spam and non-spam examples in separate files and then label the

examples sequentially within each file. Examples in the same file would have consecutive sequence numbers within a limited range and would have identical spam/non-spam status.

With this type of target leak, many classification algorithms will quickly figure out the sequence number ranges that correspond to spam and to non-spam and will settle on that one feature since it appears (in the training data) to be completely reliable. There lies the problem. When you give new data to a model that is based almost entirely on sequence numbers, the best that the model could do is throw up its hands because the new sequence numbers in the new examples would be outside the previously seen ranges. More likely, the model will attribute new examples to whichever range has the highest sequence numbers. Target leaks can be quite insidious and difficult to find. The best advice is: be suspicious of models that produce results too good to be true.

EXAMPLE 1: USING POSITION AS A PREDICTOR VARIABLE

Here we present the first version of a toy example with synthetic data to illustrate how to select predictor variables that will let a Mahout model accurately predict the desired target variable. The data displayed in figure 13.4 is your collection of historical data. Suppose you are in search of filled shapes (color-fill is your target variable). What features will best serve as predictor variables for this problem.

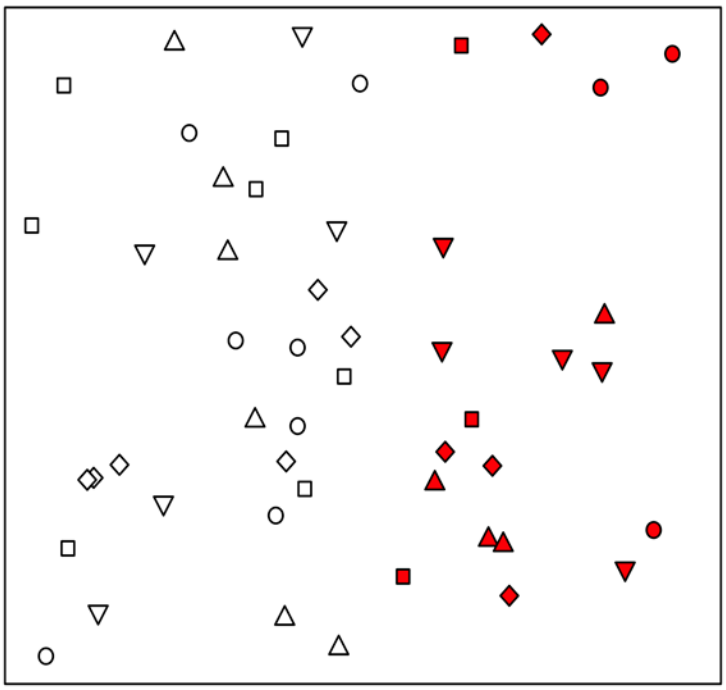


Figure 13.4 Example 1: Using position to classify color-fill.. The target variable is color fill. The features to consider as predictor variables for this historical data include shape and position. Position looks more promising as a predictor., The horizontal (x) coordinate may be sufficient., Shape does not seem to matter.

For a classification problem, you must select categories for your target variable, color fill. The target variable in this case is clearly categorical with two possible values that you could label “filled” and “unfilled”. Or you could set up the target variable as a question with yes/no responses: “Is the item filled?” Do not take anything for granted: inspect the historical data to make certain they include the target variable. Here it is obvious that they do (good!) but in real world situations, it is not always obvious.

Now you must select the features for use as predictor variables. What features can you describe appropriately? Color-fill is out (it is the target variable) but you can use position or shape as variables. You could describe position by the x and y coordinates. From a table of your data, you can create a record for

each example that includes fields for the target variable and for the predictor variables you are considering. A record for two of the training examples is shown in figure 13.5. (The complete data for this sample can be found in the examples module of the Mahout source code.)

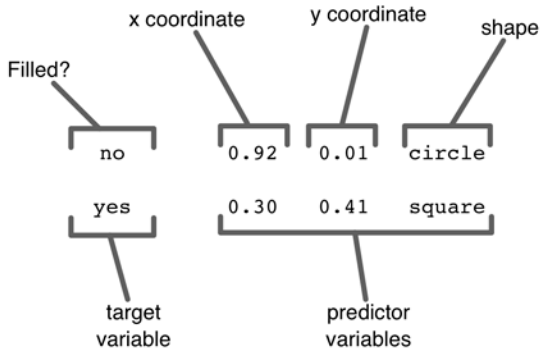


Figure 13.5 Two records for training data. The records for data shown in figure 13.4 have fields that store the values of target variable and fields for the values of the predictor variables. In this case, values relating to position (x, y coordinates) and shape have been included for each of two examples.

As you once again examine the historical data depicted in figure 13.4, you likely see that although the training data have position and shape, the x-coordinate suffices to make the differentiation between filled and unfilled symbols. Shape has no utility to the classifier in trying to determine if a shape should be labeled as filled or not filled, nor does the y-coordinate.

> tip Not all features are useful for any specific classification problem. The specific situation and question you are asking will determine which features make effective predictor variables. Learning about your data is crucial for success.

As you design your classification system, you will select those features that your experience suggests are the most likely to be effective, and the accuracy of your model will tell you if you have chosen wisely. It is not necessary to exclude all features that are not useful in differentiation, but the fewer redundant or useless features, the more likely your classifier will produce accurate results. For example 1 data shown as a record in figure 13.5, you would do best to use only the x-coordinate field for your final model.

EXAMPLE 2: DIFFERENT PREDICTOR VARIABLES ARE NEEDED WITH DIFFERENT DATA

Just because a new collection of data exhibits the same features as another group does not mean you should use the same features as before for your predictor variables. This idea is illustrated in figure 13.6. Here we see another collection of historical data that have the same features as the previous group. But in this situation, neither the x- nor y-coordinates seem helpful in predicting whether a symbol is filled or unfilled. Position is no longer useful, but shape is now a useful feature.

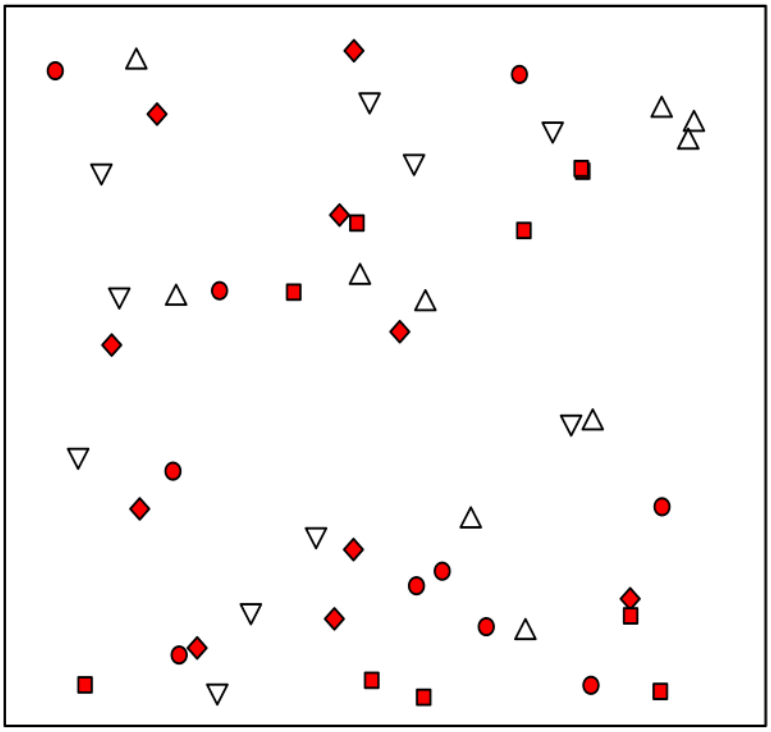


Figure 13.6 Example 2: Using shape to classify color-fill. Different data display the same features (shape and position) as example 1, but different predictor variables are needed to predict the target variable (color-fill). Position is not readily helpful, while shape is useful.

In example 2, the feature chosen for a predictor variable (shape) has three values (round, triangle, square). Had it been necessary, you could also have used orientation to further differentiate shape (square versus diamond; upwards triangle versus downward pointing triangle).

SELECT A LEARNING ALGORITHM TO TRAIN THE MODEL

In any project, you must select which algorithm to use, taking into consideration parameters such as the size of training data, the characteristics of the predictor variables and the number of categories in the target variable. Mahout classification algorithms include Naïve Bayes, Complementary Naïve Bayes, Stochastic Gradient Descent (SGD) and Random Forests. The use and behavior of Naive Bayes, Complementary Naive Bayes and SGD are described in detail in chapter 14, section 14.5.

Even in these toy examples, different algorithms have different advantages. In example 1 the training algorithm should make use of x-coordinate position to determine color fill. In example 2, shape is more useful. The x position of a point is, however, a continuous variable. Some algorithms can make good use of continuous variables. In Mahout, SGD and Random Forests do this. Other algorithms cannot use continuous variables. Naïve Bayes and Complementary Naive Bayes have this issue.

The time advantage for the parallel algorithm is obvious with an extremely large data set. Why, then, ever use the non-parallel (sequential) algorithm for classification? The answer is simply that there is a trade-off. The parallel algorithm does have considerable overhead, meaning that it takes a while to “find the car keys and get down the driveway” before it even begins processing any examples. Where you can afford the time, as with a medium-sized data set, the sequential algorithm may not only be sufficient but preferred. This trade-off is illustrated in figure 13.7 where the run-times for hypothetical sequential and parallel scalable algorithms are compared.

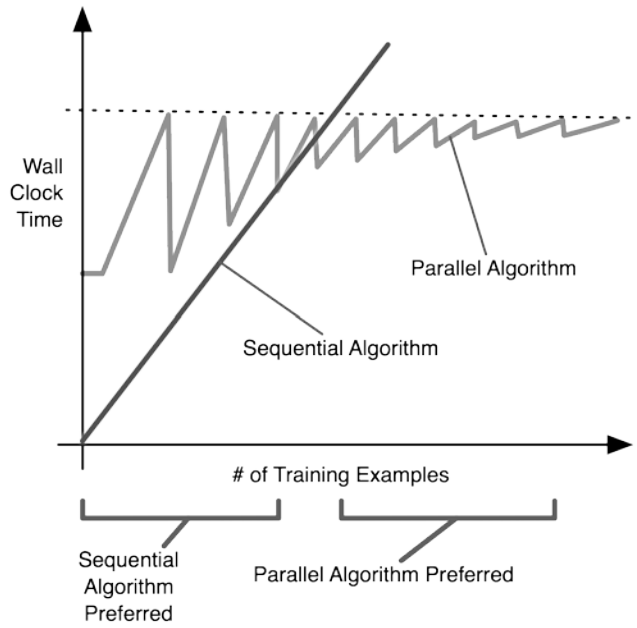


Figure 13.7 Comparing two Mahout classifier algorithms. In both cases, the algorithms are scalable as depicted in this conceptual graph.

Over the lower range of project sizes, the sequential algorithm provides a time advantage. In the midrange of size (the gap between the annotated examples) both perform equally well. As the number of training examples becomes very large, the parallel algorithm outperforms the sequential one.

As described for figure 13.1, please keep in mind that the jagged shape of the parallel algorithm shows the times at which declining performance is revived by the addition of new machines

USE LEARNING ALGORITHM TO TRAIN THE MODEL

After an appropriate set of target variables and predictor variables have been identified and encoded, and a learning algorithm has been chosen, the next step in training is to run the training algorithm in order to produce a model (these steps were depicted in figure 13.3). This model will capture the essence of what the learning algorithm is able to discern about the relationship between the predictor variables and the target variable. For example 1, the model would be something a lot like the following pseudo-code:

```
if (x > 0.5) return RED;
else return WHITE;
```

Of course, a real model that is produced by a learning algorithm is not implemented this way, but this model illustrates how simple a rule can be and still work in this toy example.

13.3.2 Workflow for stage 2: evaluating the classification model

An essential step before using the classification system in production is to find out how well it is likely to work. To do this, you must evaluate the accuracy of the model, and make large or small adjustments as needed before you begin classification. Evaluation of the trained model is almost always not a simple process. Chapter 16 provides a detailed explanation of how to go about the evaluation and fine-tuning of a model after training in preparation for running the system in production. And in the step-by-step example you will do in section 13.4, you will see the beginnings of how evaluation works.

13.3.3 Workflow for stage 3: using the model in production

Once an acceptable level of accuracy for the output of the model is achieved, classification of new data can begin. The performance of the classification system in production will depend on several factors. One of the most important is the quality of the input data. If the new data to be analyzed has inaccuracies in

the values of predictor variables or if the new data are not an appropriate match to the training data or if external conditions change over time, the quality of the output of the classification model will degrade.

> Note: In order to guard against this problem, periodic re-testing of the model is useful and retraining may be necessary.

To do retesting, you need to gather new examples and verify or generate the value of the target variable. Then you run these new examples as test data just like in the second stage and compare the results with the results on the original held-out data used in testing. If these new results are substantially worse, then something has changed and you need to consider retraining the model.

One common cause for retraining being necessary is that the incorporation of the model into your system may change the system substantially. Imagine, for example, what happens if a large bank deploys a very effective fraud detection system. Within a few months to years, the fraudsters will adapt and start using different techniques that the original model cannot detect as well. Even before these new fraud methods become a significant loss, it is very important that the bank's modelers detect the loss in accuracy and respond with updated models.

If the performance of a classification system has diminished, it is probably not necessary to discard the approach. Retraining using new and more apropos training data may be sufficient to update the model. In addition, some adjustments to the training algorithm can be useful. Ongoing evaluation is a part of retraining, as described in more detail in the discussion of model updates in chapter 16, section 16.4.1.

13.4 Step-by-step simple classification example

You have learned the basics of what classification is and how the workflow progresses in a typical project. You have seen the importance of careful selection of features to use as predictor variables depending on the question you are asking and which target variable(s) you are estimating.

Although Mahout is intended for extremely large data sets, it is helpful to start with a simple, synthetic example as you familiarize yourself with Mahout. Now you have a chance to put Mahout into action as a classifier for a small data set as a practice exercise. We will describe the data involved, how to train a model and how to tune the classifier a bit for better accuracy.

13.4.1 The Data and the Challenge

In this section you will train a classification model to assign color-fill to a collection of synthetic data similar to those in the previous discussion. For the data you will use for training, position is the key to predicting color-fill, and you will use a Mahout classification algorithm called Stochastic Gradient Descent (SGD) to train the model. Your target variable once again will be color-fill, having two categories, and with filled being the desired value. The historical data for this do-it-yourself practice project are depicted in figure 13.8 and are included as part of the Mahout distribution. To help with getting started with classification, Mahout has several such data sets built into the jar file that contains the example programs. For the simplicity throughout this worked example, we will refer to this data set as the donut data.

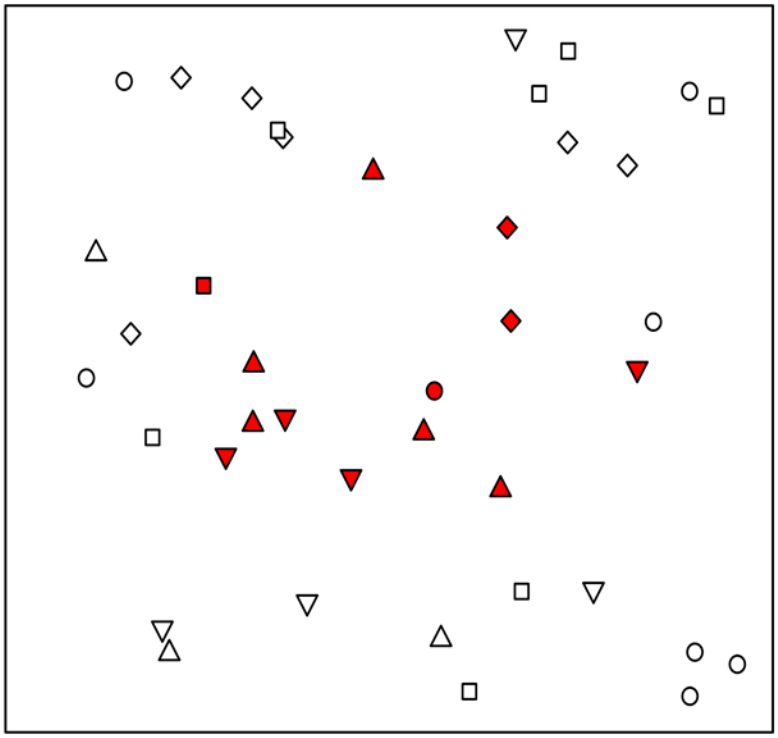


Figure 13.8 Example 3 data for a toy classification project: The “donut” data. What features are useful to find filled points?

Now you will have a chance to do a simple classification project using the “donut data” presented in figure 13.8. The target variable is color fill, and your goal is to build a system capable of finding filled points.

13.4.2 Training a model to find color-fill: preliminary thinking

As you begin to design how to train a model for your project, think about what will work as predictor variables. In previous example discussed in section 13.3.1 and depicted in figure 13.4, position rather than shape was key. A cursory inspection of figure 13.8 tells you for this simple collection of data that once again position is key, But you may suspect that neither x- nor y-coordinates nor the combination are likely to be sufficient to predict the location of filled points for this data set.

For a feature such as position, there is more than one way to consider it as a variable. Perhaps position defined as distance to a particular fixed point such as A, B or C would be useful, as shown in figure 13.9

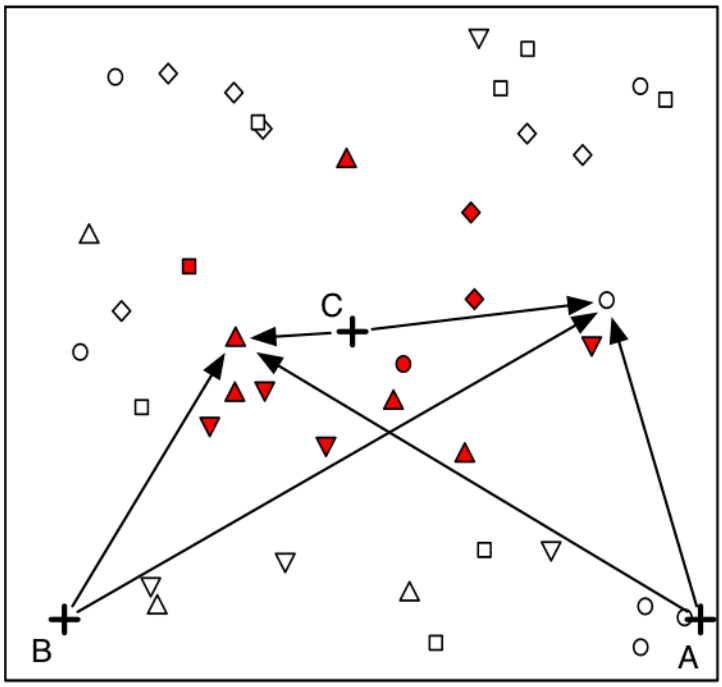


Figure 13.9 If position is the feature used as predictor variable for this “donut data” set, you must decide how bsst to designate position e to gain your particular goal, to identify filled points. Distance to C looks most useful.

While x and y coordinates are sufficient to specify any particular point, they are not useful features to predict that a point is filled. Adding the distances to points A, B and C restates position in a style that makes it easier to build an accurate classifier for these data. Distance to C promising, but you may chose to include additional predictor variables in your system. The details for how to do this are described in following sections.

13.4.3 Choosing a learning algorithm to train the model

Mahout provides a number of classification algorithms, but many are designed to handle very large data sets and as a result can be a bit cumbersome to use, at least to start. Some, on the other hand, are easy to get started with because, though still scalable, they have low overhead on small data sets.

One such low overhead method is the Stochastic Gradient Descent algorithm for logistic regression. This algorithm is a sequential (non-parallel) algorithm, but is fast, as was shown in the conceptual graph in figure 13.9. Most importantly for working with large data, the SGD algorithm uses a constant amount of memory regardless of the size of the input.

TRY RUNNING MAHOUT

First, list commands available from the command-line tool in Mahout:

```
$ $MAHOUT_HOME/bin/mahout
An example program must be given as the first argument.
Valid program names are:
  canopy: : Canopy clustering
  cat : Print a file or resource as the logistic regression models would see it
...
  runlogistic : Run a logistic regression model against CSV data
  ...
 trainlogistic : Train a logistic regression using stochastic gradient descent
 ...
```

The commands that we will be most interested in here are the `cat`, `trainlogistic` and `runlogistic` commands.

CHECK THE DATA BUILT INTO MAHOUT

To help with getting started with classification, Mahout has several datasets built in as resources in the jar file that contains the example programs. When you specify an input to the SGD algorithm, if the input you give does not specify an existing file, the training or test algorithm will read the resource instead (if a resource of that name exists, that is).

To get a listing of the contents of any of these resources, use the cat command to list the resource. For instance this command will dump the contents of the donut data set that we talked about in section 13.4.1.

```
$ bin/mahout cat donut.csv
"x","y","shape","color","k","k0","xx","xy","yy","a","b","c","bias"
0.923307513352484,0.0135197141207755,21,2,4,8,0.852496764213146,...,1
0.711011884035543,0.909141522599384,22,2,3,9,0.505537899239772,...,1
...
0.67132937326096,0.571220482233912,23,1,5,2,0.450683127402953,...,1
0.548616112209857,0.405350996181369,24,1,5,3,0.300979638576258,...,1
0.677980388281867,0.993355110753328,25,2,3,9,0.459657406894831,...,1
$
```

Note that most of the contents are not shown, we have put an ellipsis (...) where there is something missing. This file consists of a first line, which specifies the names of the fields in the data followed by more lines that contain the data itself. As you can see, there are the original predictor variables that we talked about, x, y and shape as well as the color. The values of x and y are numerical values in the range [0,1] while shape is an integer in the range [21..25]. The color is an integer that is always 1 or 2. Since subtracting a triangle from a square makes no sense, we should consider the shape variable to be categorical. Likewise, the color values should be considered numerical codes for a categorical variable with two values.

In addition to x, y, shape and color, there are additional variables present in this data to allow you to experiment with different classification schemes. These variables are described in table 13.6.

Table 13.6: Definitions of the fields for examples in the donut.csv data file.

Name	Description	Possible Values
x	The x-coordinate of a point	Numerical from 0 to 1
y	The y-coordinate of a point	Numerical from 0 to 1
shape	The shape of a point	Shape code from 21 to 25
color	Whether the point is filled or not	1=empty, 2=filled
k	The k-means cluster id derived using only x and y	Cluster code from 1 to 10
k0	The k-means cluster id derived using x, y and color	Cluster code from 1 to 10
xx	The square of the x-coordinate	Numerical from 0 to 1
xy	The product of x and y coordinates	Numerical from 0 to 1
yy	The square of the y-coordinate	Numerical from 0 to 1
a	The distance from the point (0,0)	Numerical from 0 to sqrt(2)
b	The distance from the point (1,0)	Numerical from 0 to sqrt(2)

c	The distance from the point (0.5,0.5)	Numerical from 0 to sqrt(2)/2
bias	A constant	1

BUILD A MODEL USING MAHOUT

We can build a model to determine the color field from the x and y features with the following command:

```
$ $MAHOUT_HOME/bin/mahout trainlogistic --input donut.csv \
       --output ./model \
       --target color --categories 2 \
       --predictors x y  --types numeric \
       --features 20 --passes 100 --rate 50
...
color ~ -0.157*Intercept Term + -0.678*x + -0.416*y
      Intercept Term -0.15655
                   x -0.67841
                   y -0.41587
...
```

This command specifies that the input come from the resource named `donut.csv`, that the resulting model be stored in the file `./model`, that the target variable is in the field named `color` and that it has two possible values. We also specify that the algorithm should use variables `x` and `y` as predictors, both with numerical type. The remaining options specify internal parameters for the learning algorithm. A list of all of the command line options for the `trainlogistic` program is shown in table 13.7.

Table 13.7 Command Line Options for the `trainlogistic` program

Option	What it Does
`--quiet`	Produce less status and progress output
`--input <file-or-resource>`	Use the specified file or resource as input
`--output <file-for-model>`	Put the model into the specified file
`--target <variable>`	Use the specified variable as the target
`--categories <n>`	How many categories does the target variable have?
`--predictors <v1> ... <vn>`	A list of the names of the predictor variables
`--types <t1> ... <tm>`	A list of the types of the predictor variables. Each type should be one of `numeric`, `word` or `text`. Types can be abbreviated to their first letter. If too few types are given, the last one is used again as necessary. Use `word` for categorical variables.
`--passes`	The number of times the input data should be re-examined during training. Small input files may need to be examined dozens of times. Very large input files probably don't even need to be completely examined
`--lambda`	Controls how much the algorithm tries to eliminate variables from the final model. A value of 0 indicates no effort is made. Typical values are on the order of 0.00001 or less.
`--rate`	The initial learning rate. This can be large if you have lots of data or use lots of passes because it is decreased progressively as data is examined.

`--noBias`	Do not use the built-in constant in the model (this eliminates the `Intercept Term` from the model. Occasionally this is a good idea, but generally it is not since the SGD learning algorithm can usually eliminate the intercept term if warranted.
`--features`	The size of the internal feature vector to use in building the model. A larger value here can be helpful, especially with text-like input data.

13.4.4 Improving performance of the color-fill classifier

Now that you have built a functioning, trained classification model, you need to determine how well it is performing the task of estimating color-fill condition. This evaluation stage in the process is not just a way to assign a "grade" for your project. It is your means of assessing and improving your classifier for excellent performance.

EVALUATE THE MODEL

Remember that this problem has the filled points completely surrounded by unfilled points which makes it essentially impossible for a simple model such as produced by the SGD algorithm using only `x` and `y` coordinates to classify points accurately. Indeed, the linear equation underlying this model can only produce a negative value which, in the context of logistic regression, is guaranteed to never produce a score greater than 0.5.

After the model is trained, we can run the model on the training data again to evaluate how it does (even though we know it will not do well).

```
$  bin/mahout runlogistic --input donut.csv --model ./model \
        --auc --confusion
AUC = 0.57
confusion: [[27.0, 13.0], [0.0, 0.0]]
...
```

The output here contains two values of particular interest. First, the

(which is short for Area Under the Curve, and is a widely used measure of model quality) has a value of 0.57. AUC can range from 0 for a perfectly perverse model that is always exactly wrong to 0.5 for a model that is no better than random to 1.0 for a model that is perfect. The value here of 0.57 indicates a model that is hardly better than average.

To understand why this model performs so poorly, you can look to a confusion matrix for a clue. A confusion matrix is a table that compares actual results with desired results. Along with other measurement tools, the confusion matrix is described in more detail in the evaluation chapter 15. All examples are classified at the default score threshold of 0.5 as being unfilled. This allows the classifier to be correct 2/3 of the time (27 out of 40 times), but it only manages to be correct marking everything as unfilled. This model gets the right answer much of the time but only in the same way that a stopped clock is right twice a day.

The options accepted by the `runlogistic` command are summarized in the table 13.8.

Table 13.8 Command Line options for the `runlogistic` program

Option	What it Does
`--quiet`	Produce less status and progress output
`--auc`	Print out AUC score for model versus input data after reading data
`--scores`	Print target variable value and scores for each input example

```
--threshold <t>
```
Set the threshold for confusion matrix computation to `t` (default 0.5)

```
--confusion
```
Print out confusion matrix for a particular threshold (See `--threshold`)

```
--input <input>
```
Read data records from specified file or resource

```
--model <model>
```
Read model from specified file

BUILD A MORE INTERESTING MODEL

We can get more interesting results if we use additional variables for training. For instance, this command allows the model to use x and y as well as a, b and c for building the model.

```
$ bin/mahout trainlogistic --input donut.csv --output model \
         --target color --categories 2 \
         --predictors x y a b c  --types numeric \
         --features 20 --passes 100 --rate 50
...
color ~ 11.520*Intercept Term + 3.257*x + 3.923*y + -1.702*a + -1.546*b + -38.844*c
      Intercept Term 11.51961
                   a -1.70190
                   b -1.54631
                   c -38.84422
                   x 3.25651
                   y 3.92260
...
```

Notice how this model has a large weight that is applied to the c variable as well as a large intercept term. If we ignore the other variables (which isn't entirely right to do, but the intercept and c are definitely dominant here), the output of the linear part of this model will reach a maximum value of 11.5 where c = 0 and will drop rapidly into negative territory once c > 0.3. This makes a lot of sense geometrically, of course, based on what we know about the problem. This consideration alone is a strong indication that this model is not a stopped-clock model like the one that depended on x and y alone.

SIDEBAR Logistic Regression (sidebar)

Logistic regression describes a kind of classification model in which the predictor variables are combined with linear weights and then passed through a soft-limit function that limits the output to the range from 0 to 1. Logistic regression is closely related to other models such as a perceptron (where the soft-limit is replaced by a hard-limit), neural networks (where multiple layers of linear-combination and soft limiting are used) and Naive Bayes (where the linear weights are determine strictly by feature frequencies assuming independence). Logistic regression cannot separate all possible classes, but in very high dimensional problems or where you can introduce new variables by combining other predictors this is much less of a problem. The mathematical simplicity of logistic regression allows very efficient and effective learning algorithms to be derived.

LOOK INSIDE THE MODEL

The on-disk representation of a Mahout logistic regression model such as produced here is, by the way, an ordinary text file containing a JSON data structure that you can examine. The model that resulted from the command above can be viewed without any special tools, and although it can be a bit difficult to decipher, the major outlines are clear.

```
$ cat model
{"targetVariable":"color","typeMap":{"b":"n","c":"n","a":"n","y":"n","x":
    "n"},"numFeatures":20,"useBias":true,"maxTargetCategories":2,
  "targetCategories":["2","1"],"lambda":1.0E-4,"learningRate":50.0,"lr":{
    "beta":{"rows":1,"cols":20,"data":[[0.0,-1.5463057251192813,0.0,0.0,
          0.0,11.519613974616073,-1.701904213103679,0.0,0.0,0.0,0.0,0.0,
          0.0,3.922598277596618,3.2565105593880252,0.0,0.0,0.0,0.0,
          -38.84422219106972]]},"numCategories":2,"step":4001,"mu_0":50.0,
    "decayFactor":0.999,"stepOffset":10,"forgettingExponent":-0.5,"lambda":
    1.0E-4,"sealed":true}}
```

TEST AGAIN

If we run this more interesting model on the training data, we get a very different result than with the previous model.

```
$  bin/mahout runlogistic --input donut.csv --model model \
        --auc --confusion
AUC = 1.00
confusion: [[27.0, 0.0], [0.0, 13.0]]
entropy: [[-0.0, -8.7], [-6.4, -0.0]]
```

Now we have AUC with a perfect value of 1 and we see from the confusion matrix that the model was able to classify all of the training examples perfectly.

TEST USING NEW DATA

We can run this same model on additional data found in the resource `donut-test.csv`. Since this new data were not used to train the model, there are a few surprises in store for the model.

```
$  bin/mahout runlogistic --input donut-test.csv --model model \
        --auc --confusion
AUC = 0.97
confusion: [[24.0, 2.0], [3.0, 11.0]]
entropy: [[-0.0, -0.6], [-5.4, -0.7]]
```

On this held out data, we see that the AUC has dropped to 0.97, which is still quite good. The confusion matrix shows that of the 40 new samples there are 5 incorrectly classified examples, 2 unfilled points that were marked as filled by the model (false positives) and 3 filled points that were marked by the model as unfilled (false negatives). Clearly the use of the additional variables, notably the inclusion of c, has helped the model substantially but there are still a few issues. Figure 13.10 shows what happened.

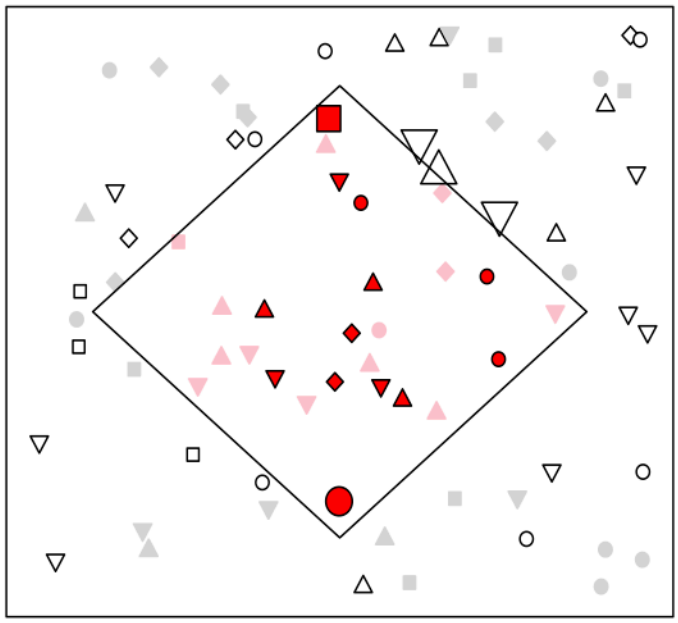


Figure 13.10 Trying the classifier on new data. The original data is shown in different shades. The new data is filled or not. The expanded symbols are the ones that the model got wrong. The square shows the region that we actually used to generate this data.

Notice the large, filled circle near the bottom of the square shown in figure 13.10. This circle is close to training dots that were not filled but not near filled training dots. Since the learning algorithm was not allowed to know that the correct region had this odd shape, it was nearly impossible for the model to get the right answer.

TRY OTHER MODELS

We can do additional experiments by selecting other sets of variables. For instance, here is what happens when we include x, y, a and b, but exclude c from the set of predictor variables

```
$ bin/mahout trainlogistic --input donut.csv --output model \
           --target color --categories 2 \
           --predictors x y a b  --types n --features 20 \
           --passes 100 --rate 50
...
color ~ 10.597*Intercept Term + 0.641*x + 17.866*y + -13.901*a + -13.155*b
      Intercept Term 10.59732
                   a -13.90088
                   b -13.15528
                   x 0.64150
                   y 17.86575
  ...
$  bin/mahout runlogistic --input donut-test.csv --model model \
             --auc --confusion
AUC = 0.98
confusion: [[27.0, 2.0], [0.0, 11.0]]
entropy: [[-0.1, -0.5], [-2.0, -1.0]]
$
```

This time, the model does slightly better on the held-out data even though its favorite variable has been withheld from it. This indicates that the information carried by c is also available in x, y, a and b. You can try lots of experiments on the parameters of the learning model as well. These experiments are the beginning of the evaluation process described in Chapter 16.

13.5 Summary

Using the toy example in the previous section, you have put to use what you have learned throughout the chapter about the first two stages of classification: how to train a model and how to evaluate it and adjust performance. If this were a real system, your model would now be ready for the third stage in classification, deploying it into production in a real world setting. The basic terminology of classification associated with fundamental ideas will now be familiar to you, and you have a solid understanding of what classification is and how it works.

Remember that a key to building a successful classifier is to carefully pose the question being asked in specific and simple fashion so that it can be answered by a limited list of pre-determined categories, the target variables. Be aware that not all features of input data are equally helpful; you must select the features to be used as predictor variables carefully and may need to try many combinations to find those useful for high performance of your classifier. And remember, pragmatism is important.

With these ideas in mind, think back to the example of wine tasting. The goal of your machine-based classifier should be to help you get you to dinner on time, not to help you make subtle esthetic judgments about the finer things in life.

As you continue through chapters 14-17, you will see how Mahout provides a powerful solution to the requirements of extremely large classification systems, particularly those with data sets above 1 million examples. Chapter 14 focuses on how to extract features in preparation for input to the learning algorithm, as a preparation to find out how to evaluate and tune a trained classifier before deploying it to production.

14

Training a classifier

- This chapter covers
  - Extracting features from text and converting them into a form Mahout can use
  - Training two Mahout classifiers using the 20 newsgroups data set
  - Selecting among the learning algorithms available with Mahout

This chapter explores the first stage in classification, training the model. Developing a classifier is a dynamic process that requires you to think creatively about the best way to describe the features of your data and to consider how they will be used by the learning algorithm that you choose in order to train your models. Some kinds of data lend themselves readily to classification; others offer a greater challenge, which can be rewarding, frustrating and interesting all at once. In this chapter you will learn how to choose and extract feature data, preprocess the data to make them classifiable, and convert them to vectors to be usable by the classifier algorithms.

Later in the chapter, we review the characteristics of the learning algorithms available for use with Mahout classification and help you decide how to choose the ones best suited to particular projects. These two steps, extracting features and choosing the algorithm, go hand-in-hand in designing and training the classifier, as this chapter will show. To help you develop an intuitive sense of the different options in planning a system and training a classification model, we provide two step-by-step examples using two different learning algorithms, Stochastic Gradient Descent (SGD) and Naïve Bayes, with the same data set, 20 newsgroups.

This chapter starts by explaining how to work with the data in your training examples.

14.1 Extracting features to build a Mahout classifier

Getting data into a form usable by a classifier is a complex and often time-consuming step, and in this section we present an overview of what is involved. In Chapter 13, we presented a simplified view of the training and use of a classification model as depicted in Figure 13.1, which showed a single step leading from a collection of training example data to input into a learning algorithm that trains the classification model. But in reality, the situation is more complex. The important details not mentioned in Chapter 13 are shown in Figure 14.1.

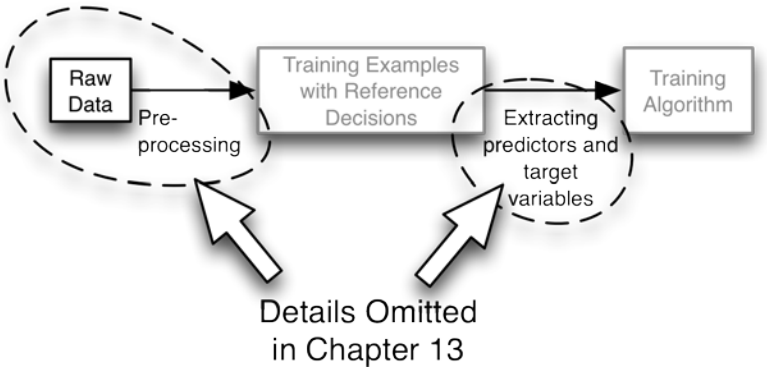


Figure 14.1 A portion of Figure 13.1 from the previous chapter is expanded here to highlight what must be done to raw data to get it ready for input to a training algorithm.

The original figure showed a single step leading from the training data to the training algorithm. In reality, raw data must be collected together and pre-processed to be classifiable training data. Once the raw data are preprocessed into classifiable form, several steps are required to select predictor and target variables and encode them as vectors, the style of input required by Mahout classifiers.

The training examples in figure 14.1 are classifiable data, not the original raw data. The processing of raw data into classifiable data is covered in this chapter and discussed in more detail in later chapters on tuning and production. Recall from Chapter 13 that the values for features to be used as predictor variables are of one of four types:

- continuous
- categorical
- word-like
- text-like

This description of the form of predictor variables is correct on the whole, but it omits the important fact that when the values are presented to any of the algorithms used to train a classifier, the data must be in the form of vectors of numbers, either in memory or in a file format that the training algorithm can read.

Compare the simplified sequence depicted in Figure 14.1 with the more detailed diagram in Figure 14.2. Original data undergo changes in order to make them presentable as the vectors required as input for the training algorithm. These changes happen in two phases: preprocessing to produce classifiable data for use as training examples and conversion of the classifiable data into vectors.

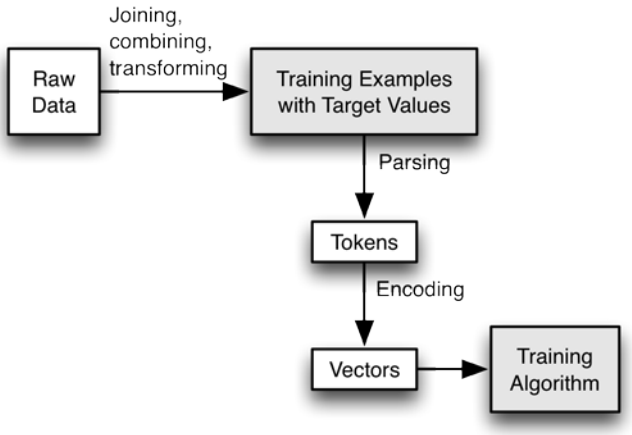


Figure 14.2 Vectors are the format that the classification algorithms require as input. In order to encode data as vectors, raw data must be collected in a single record in classifiable format, ready for parsing and vectorization.

As Figure 14.2 shows, preparation of data for the training algorithm consists of two main steps. The second step has two phases, tokenization and vectorization. For continuous variables, the text analysis may be trivial, but for other types of variables such as categorical, word-like or text-like, vectorization can be considerably involved in some cases. These two steps are compared in Table 14.1 and described in detail in subsequent sections of this chapter.

Table 14.1 Two phases in feature extraction: preprocessing raw data and text analysis with vectorization that will convert data to the vectors required by the training algorithm.

Step in Preparation	Description
Preprocessing raw data	Rearrangement of raw data into a record. Conversion of raw data into records with identical fields in one of four types: continuous, categorical, word-like, or text-like in order to be classifiable.
Converting data to vectors	Parsing and vectorization of classifiable data, either using custom code or via tools such as Lucene analyzers and Mahout vector encoders. Some Mahout classifiers also include vectorization code.

The first phase of feature extraction involves rethinking the data to identify features to use as predictor variables. To do this, first select a target variable in accordance with the goal of classification, then pick or drop features to get a mix that is worthy of an initial try. There is no exact recipe: this step calls for intelligent guessing guided by experience, and as you work through the examples in this chapter, you will be on your way towards getting that experience. Begin by learning how raw data is made classifiable.

14.2 Preprocessing raw data into classifiable data

This section offers a brief overview of preprocessing of data. The steps include collecting or rearranging data into a single record, and gleaning secondary meaning from raw data (such as converting a ZIP code to a 3-digit ZIP or using a birth date to determine age). Preprocessing is not a prominent part of the examples in this chapter, because this phase of data extraction has essentially already been done for you in the data set you will be using in the hands-on examples later. Preprocessing plays a larger role in the worked examples of Chapters 16 and 17.

14.2.1 Transforming raw data

Once you identify features to try out, they must be converted into a format that is classifiable. This task involves rearrangement of data into a single location as well as transformation of data into an appropriate and consistent form.

Note: Classifiable data consist of records with identical fields containing one of four types of data: continuous, categorical, word-like, or text-like. Each record contains the fully denormalized description of one training example.

At first glance, it may seem this step is already done: if the data look like words, they must be words, right? If the data look like numbers, they must be continuous, yes? But as you saw in chapter 13, first appearances can be a little misleading. For example, a ZIP code looks like a number but is actually a category, a label for predetermined classes. Things that contain words may be word-like, or they may best considered as categories or text. ID codes for users or products may look like numbers, categorical or word-like data, but more commonly, they should be denormalized away in favor of the characteristics of the user or product that they link to. The following example from computational marketing provides an exercise in preparing raw data for use in classification.

14.2.2 Computational marketing example

Suppose that you are trying to build a classification model that will determine whether a user will buy a particular product if they are offered it. This is not quite a recommendation system because it will classify examples using user and product characteristics rather than collective behavior of similar users.

For this example, you have a database with several tables in it as shown in Figure 14.3. The tables are typical of a highly simplified retail system. There are tables representing users, representing products, recording when products were offered or shown to users, and a table that records purchases that arise from showing a product to a user. As they stand, the data in these tables are not acceptable as training or test data for a classifier since they are spread across several tables.

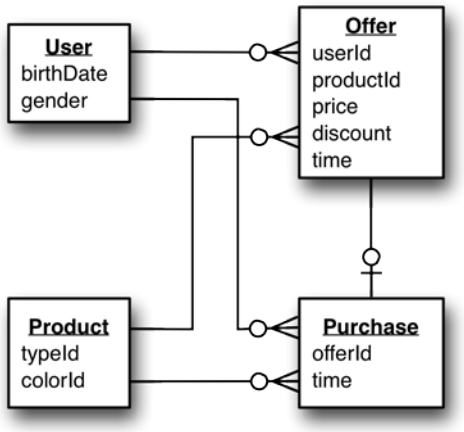


Figure 14.3. Table structure for product sales example contains different types of data. Since no record in any table contains the complete picture necessary in a training example for a classifier, this organization of raw data is not directly usable as training data.

The marketing project depicted in Figure 14.3 has a wide variety of data types. Users have demographic data such as birth date and gender, products have type, and color. Offers of products to users are recorded in the offer table, and purchases of products subsequent to an offer are recorded in the purchase table.

Figure 14.4 shows how these data might become training data for a classifier by this query. There will be one of these records for each record in the `offer` table, but note how user and product IDs have been used to join against the user and product tables. In the process of joining against the user table, the user's birth date is expressed as an age. An outer join has been used to derive the delay until purchase as well as a flag indicating whether a purchase was ever made.

Figure 14.4 To get classifiable data, training examples are constructed from several sources to denormalize everything into a single record that describes what happened. Some variables are transformed; birthDate is expressed as age, and purchase time is expressed as a delay.

In order to get the data represented in the tables in figure 14.3 into the usable form shown in figure 14.4, they need to be brought together and rearranged. To do this, you might use an SQL query like this:

```
select
  now()-birthDate as age, gender,
  typeId, colorId, price, discount, offerTime,
  ifnull(purchase.time, 0, purchase.time - offer.time) as purchaseDelay,
  ifnull(purchase.time, 0, 1) as purchased
from
  offer
  join user using (userId)
  join product using (productId)
  left outer join purchase using (offerId);
```

This query denormalizes the data stored in the separate database tables into records that contain all the necessary data. The driving table here is the `offer` table and the nature of the foreign keys for `userId`, `productId` and `offerId` force the output of this query to have exactly one record for each record in the original `offer` table. Note how the join against the purchase table is an outer join. This allows the `ifnull` expressions involving purchase.time to produce 0s if there is never a purchase.

> SIDEBAR: Sometimes age is better for classification and sometimes birth date is better. For instance, in the case of insurance data on car accidents, age will be a better variable to use because having car accidents is more related to life-stage than it is to the generation a person belongs to. On the other hand, in the case of music purchases, birth date might be more interesting because people often retain early music preferences as they get older. Their tastes often reflect those of their generation.

The records that come out of this query are now in the form of classifiable data, ready to be parsed and vectorized by the training algorithm. These next steps can be done using code you write or by putting data into a format accepted by Mahout classifiers (which contain their own parsing and vectorization code). The following section and examples focus on vectorization of the parsed, classifiable data.

14.3 Converting classifiable data into vectors

In Mahout parlance, a `Vector` is a data type that stores floating-point numbers indexed by integers. This section of the chapter teaches you how to encode data as `Vectors`, explains what feature hashing is and how the Mahout API does feature hashing. It includes how to encode the different types of values associated with variables.

You saw `Vectors` used previously in the chapters on clustering. Many kinds of classifiers, especially ones available in Mahout, are fundamentally based on linear algebra and therefore require training data to be input in the form of a `Vector`.

14.3.1 Representing data as a vector

How do you represent classifiable data as a `Vector`? There are several ways to do this, which are covered in this subsection and summarized in table 14.2.

Table 14.2 Approaches to encoding classifiable data as a vector.

Approach	Benefit	Cost	Where Used?
One vector cell per word, category or continuous value	No collisions, can be reversed easily	Requires two passes, one to assign cells, one to set values, vectors may have different length	Dumping Lucene index as `Vectors` for clustering.
Represent vectors implicitly as bags of words	One-pass, no collisions	Difficult to use linear algebra primitives, difficult to represent continuous values, must format data in special non-vector format	Naïve Bayes
Feature hashing	One pass, vector size fixed in advance, good fit to linear algebra primitives	Feature collisions, interpretation of resulting model may be tricky	`OnlineLogisticRegression` and other SGD learners

The approaches listed in Table 14.2 are used in different classifiers in Mahout. Let's begin by looking at how to encode word-like, text-like or categorical values as a `Vector`.

ONE CELL PER WORD

One way to encode classifiable data as a `Vector` involves two passes through the training data: one to determine the necessary size of the `Vector` and build a dictionary that remembers where every feature should go in the `Vector`, and then another to actually convert the data. This approach can use a simple representation for encoding training and test examples. In this representation, every continuous value and every unique word or category in categorical, text-like and word-like data is assigned a unique location in the vector representation. There is an obvious downside, however, in that passing through the training data twice has the potential for making the training of the classifier twice as computationally expensive, which can be a real problem with very large data sets. This two-pass approach is used by most of the clustering algorithms in Mahout.

VECTORS AS BAGS OF WORDS

Another approach is to carry around the feature names or names plus category, word or text values instead of carrying around a `Vector` object. This method is used primarily by Mahout classifiers like Naïve Bayes and Complementary Naïve Bayes. The advantage of this approach is that it avoids the need for a dictionary, but it means that it is difficult to make use of Mahout's linear algebra capabilities that would require known and consistent lengths for the `Vector` objects involved.

FEATURE HASHING

SGD-based classifiers avoid the need to predetermine vector size by simply picking a reasonable size and "shoehorning" the training data into vectors of that size. This approach is known as feature hashing. The shoehorning is done by picking one or more locations by using a hash of the name of the variable for continuous variables, or a hash of the name and the category name or word for categorical, text-like or word-like data.

This hashed feature approach has a distinct advantage of requiring less memory and one less pass through the training data, but it can make it much harder to reverse-engineer vectors, to determine what original feature mapped to a vector location. This is because multiple features may hash to the same location. With large vectors or with multiple locations per feature, this is not a problem for accuracy, but it can make it hard to understand what a classifier is doing.

14.3.2 Examining Mahout APIs to do feature hashing

In this section we present in detail how to encode continuous, categorical, word-like and text-like features. We also explain what feature collisions are and how they may affect your classifier.

ENCODING CONTINUOUS FEATURES

Continuous values are the easiest kind of data to encode. The value of the continuous variable gets added directly to one or more locations that are allocated for the storage of the value. The location or locations are determined by the name of the feature.

Mahout supports feature hashed encoding of continuous values through `ContinuousValueEncoder`. By default, the `ContinuousValueEncoder` only updates one location in the vector. You can specify how many locations are to be updated using its `setProbes()` method. To encode a value for one variable, a `Vector` is needed: `new RandomAccessSparseVector(lmp.getNumFeatures())`. The value is then encoded with `encoder.addToVector(value, vector)`. Note that the value being encoded is assumed to be a string. If the value being encoded is available as a `double`, then simply pass a `null` for the `String` value and use the weight as the value: `encoder.addToVector(null, value, vector)`

ENCODING CATEGORICAL AND WORD-LIKE FEATURES

Categorical features that have *n* distinct values can be encoded by storing the value 1 into one of *n* vector locations, according to which value the variable has. To decrease redundancy, it is also possible to use only *n*-1 locations and then encode the *n*th category by storing a 1 in no location at all (all zeroes).

Hashed feature encoding is similar except that the locations are scattered throughout the feature vector using the hash of the feature name and value. In addition, each category can be associated with several locations. An additional benefit of feature hashing is that the unknown and unbounded vocabularies typical of word-like variables are not a problem.

To encode a categorical or word-like value using feature hashing, create a `WordValueEncoder`. This encoder is used in the way that the `ContinuousValueEncoder` was used, except that the default number of probes is 2 and the location being updated varies according to the value specified as well as the name of the variable. There are two kinds of this encoder. The `AdaptiveWordValueEncoder` builds a dictionary on the fly to estimate word frequencies so that it can apply larger weights to rare words. The `StaticWordValueEncoder` uses a supplied dictionary or gives equal weights to all words if none is given.

Some learning algorithms are substantially helped by good word weighting, but others, like the `OnlineLogisticRegression` class described later, are not much affected by equal weights for all words.

Encoding words with equal weighting is similar to how continuous values were encoded. Here is some sample code that shows how to encode continuous variables:

```
RecordValueEncoder encoder =                        #1
    new StaticWordValueEncoder("variable-name");

for (DataRecord ex: trainingData) {
  Vector v = new RandomAccessSparseVector(10000);
  String word = ex.get("variable-name");
  encoder.addToVector(word, v);                     #2
  // use vector
}
#1 tell encoder variable name
#2 this increments a few values in v
```

In this code, a `StaticWordValueEncoder` is constructed and assigned a name that determines the seed for the random number generator, which in turn determines the locations to encode the word-like values. Then later, values are encoded. This is done by combining the name of the variable and the word being encoded to get locations to be modified by the encoding operation.

Sizing the vector requires experimentation. Larger vectors consume more memory and slow training. Smaller vectors eventually cause so many feature collisions that the learning algorithm cannot compensate.

ENCODING TEXT-LIKE FEATURES

Encoding a text-like feature is similar to encoding a word-like one except that text has many words. In fact, the example here simply adds up the vector representations for the words in the text to get a vector for the text. This approach can work reasonably well, but the more nuanced approach of counting the words first and then producing a representation that is a weighted sum of the vectors for unique words can produce better results. This second approach is illustrated later in this chapter.

SIDEBAR: Text-like data values are, in fact, ordered sequences of words, but considering the ordering of the words in text is often not necessary to get good accuracy. Simply counting the words in a text, ignoring order, is usually sufficient. That said, the best vector for text is not usually just the sum of the vectors for the words in the text weighted by the counts of the words. Instead, the word vectors are usually best weighted by some function of the number of times that the corresponding word appeared in the text. Commonly used weight functions are the square root, the logarithm, or equal weighting for all words that appear regardless of their frequency. The example in listing 14.2 uses linear weighting for simplicity, but weighting by logarithm of the count is a better starting point for most applications.

Listing 14.3 shows how to encode text as a vector by encoding all of the words in the text independently, and then producing a linearly weighted sum of the encodings of each word. This is accomplished with a `StaticWordValueEncoder`, and a way to break down or parse the text into words. Mahout provides the encoder and Lucene provides the parser.

Listing 14.3 Tokenizing and vectorizing text

```
RecordValueEncoder encoder = new StaticWordValueEncoder("text");
Analyzer analyzer = new StandardAnalyzer(Version.LUCENE_30);     #A

StringReader in = new StringReader("text to magically vectorize");
TokenStream ts = analyzer.tokenStream("body", in);
TermAttribute termAtt = ts.addAttribute(TermAttribute.class);

Vector v1 = new RandomAccessSparseVector(100);                   #B
while (ts.incrementToken()) {
  char[] termBuffer = termAtt.termBuffer();
  int termLen = termAtt.termLength();
  String w = new String(termBuffer, 0, termLen);                 #C
  encoder.addToVector(w, 1, v1);                                 #D
}
System.out.printf("%s\n", new SequentialAccessSparseVector(v1));
#A breaks text into words
#B encode into vector size 100
#C word to encode
#D add word w into vector v
```

This code produces a printable form of the vector like this:

```
{8:1.0,21:1.0,67:1.0,77:1.0,87:1.0,88:1.0}
```

A graphical form of this output is shown in figure 14.5:

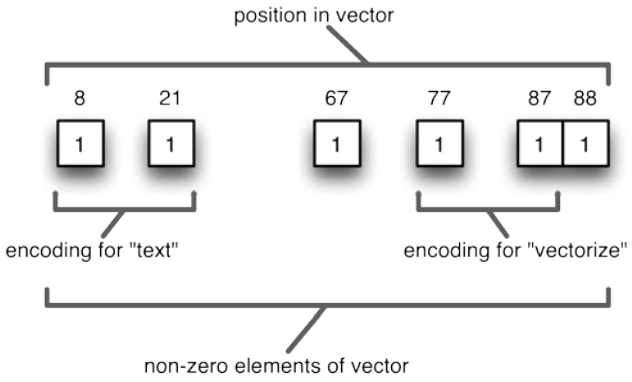


Figure 14.5 This diagram shows a vector that encodes “text to magically vectorize”. There are 6 non-zero values in this vector which has size 100. Values equal to zero are not stored and thus are not shown in results like these produced by the Lucene standard analyzer.

You can see in Figure 14.5 that locations 8 and 21 encode the word “text”, locations 77 and 87 encode “vectorize” and locations 67 and 88 encode “magically”. The word “to” is eliminated by the Lucene standard analyzer. The overall result of this process is a sparse vector, in which values equal to zero are not stored at all.

FEATURE COLLISIONS

The hashed feature representation for data can lead to collisions where different variables or words are stored in the same location. For instance, if you had used a vector with length 20 above instead of the 100-long vector shown above, you would have gotten the following result as the encoded value of “text to magically encode”: $\{1:1.0, 7:2.0, 8:2.0, 17:1.0\}$ In this vector, locations 7 and 8 have a value of 2 instead of 1 like all the non-zero cells in the vector shown in figure 14.5 had. How this happens is shown in figure 14.6.

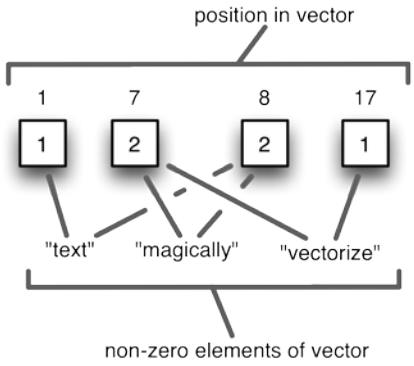


Figure 14.6. When the vector is size 20 instead of size 100, collisions occur. The terms “text” and “magically” collide at location 8 and the terms “magically” and “vectorize” collide at location 7. These collisions make it more difficult to interpret the resulting vector, but typically do not affect classification accuracy.

Dealing with feature collisions to avoid hurting performance can be fairly easy. In the vector shown in Figure 14.6, the word “text” is encoded in locations 1 and 8, “magically” in locations 7 and 8 and the word “vectorize” in locations 7 and 17. This means that “text” and “magically” share location 8 and “vectorize” and “magically” share location 7. So, locations 7 and 8 contain a value of 2 instead of 1 for all of the other non-zero values. Since each word has two locations assigned to it, the learning algorithm can learn to compensate for these collisions.

In this example, the learning algorithm compensates because the transformation from words to vector is invertible. This would not normally true if each word had a single location, because the vector dimension will almost always be smaller than the vocabulary of the text. With multiple locations, however, this vectorization technique usually works for the same reasons that Bloom filters work.

Where there is a collision between two words that have quite different meaning for the classifier, it is unlikely that the other locations for these two words will also collide and it is also unlikely that either will collide with a third word that presents the same problem. On average, therefore, this vectorization technique works well and if the vector length is chosen to be large enough, there will be no measurable loss of accuracy. Exactly how long is long enough is subject to empirical validation using the evaluation techniques presented throughout Chapter 16.

Let’s try these approaches with some real world data using the Mahout classification algorithm known as Stochastic Gradient Descent, or SGD. Later in this chapter, you will have a chance to do another version of classifying the same data set with a different Mahout algorithm. A discussion of both classes of algorithms is found in the overview of Mahout algorithms presented in section 14.5. Now let’s get started with the first version using SGD.

14.4 Classifying the 20 newsgroups data set with SGD

In this section you will build a classification model for the 20 newsgroups data set using the SGD algorithm. This project will include previewing data and doing preliminary analysis, analyzing which are the most common headers, converting the data to vector form through parsing and tokenizing and the training code for the 20 newsgroups data set project.

> sidebar: The 20 newsgroups data set is a standard data set commonly used for machine learning research. The name comes from the fact that the data are from transcripts of several months of the postings made in 20 Usenet newsgroups from the early 1990's.

This example emphasizes feature extraction and focuses largely on the second phase of feature extraction, vectorization. The 20 newsgroups data set is useful here because the first phase of feature extraction, preprocessing raw data into classifiable data, will be relatively simple, as the researchers who created the data set have already done most of the work.

14.4.1 Getting started: preview the data set

The first step in preparing a data set is to examine the data and decide which features might be useful in classifying examples into categories of the chosen target variable -- in this case each of the 20 newsgroups.

To begin, download the 20 newsgroups data set from this URL:
http://people.csail.mit.edu/jrennie/20Newsgroups/20news-bydate.tar.gz

This version of the data set splits training and test data by date and keeps all of the header lines. The date split is good because it is more like the real world, where you will want to process new data with your classifier rather than old data.

If you examine one of the files in the training data directory, such as `20news-bydate-train/sci.crypt/15524`, you will see something like this:

```
From: rdippold@qualcomm.com (Ron "Asbestos" Dippold)
Subject: Re: text of White House announcement and Q&As
Originator: rdippold@qualcom.qualcomm.com
Nntp-Posting-Host: qualcom.qualcomm.com
Organization: Qualcomm, Inc., San Diego, CA
Lines: 12

ted@nmsu.edu (Ted Dunning) writes:
>nobody seems to have noticed that the clipper chip *must* have been
>under development for considerably longer than the 3 months that
>clinton has been president.  this is not something that choosing
```

The 20 newsgroups data set consists of messages, one per file. Each file begins with header lines that specify things such as who sent the message, how long it is, what kind of software was used and the subject. A blank line follows, and then the message body follows as unformatted text.

The predictor features in this kind of data are either in the headers or in the message body. A natural step when first examining this kind of data is to count the number of times different header fields are used across all documents. This helps determine which ones are most common and thus likely to affect our classification of lots of documents. Because these documents have a simple form, a bash script like this can be used to count the header lines.

```
#!/bin/bash
export LC_ALL='C'
for file in 20news-bydate-train/*/*
do
  sed -E -e '/^$/,$d' -e 's/:.*//' -e '/^[[:space:]]/d' $file
done | sort | uniq -c | sort -nr
```

This script scans all of the files in the training data set, deletes lines after the first blank line and deletes all content in the remaining lines after the colon. In addition, header continuation lines that start with a space are deleted. The header lines that remain in the output from `sed` are then sorted, counted and sorted again in descending order by count.

The header line counts for the training examples are summarized in Table 14.3. The row marked "..." indicates that we skipped to the latter part of the list to show you some of the more bizarre examples. The last column shows what actions you might choose for each header based on its potential usefulness. The decision choices are to discard during parsing the useless headers (drop) or retain potentially useful ones (keep). We marked some headers as "Try?" to indicate some that are less clearly useful but definitely worth a try. Do use data such as these whose value is unclear as they may prove to be very helpful. Perhaps we should have marked them as "Try!".

Table 14.3 A listing of the most common headers found in the 20 newsgroups articles. A few of the less common headers are included as well.

Header	Count	Comment	Action
Subject	11314	Text	Keep
From	11314	Sender of message	Keep
Lines	11311	Number of lines in message	Keep
Organization	10841	Related to sender	Try?
Distribution	2533	Possible target leak	Try?
Nntp-Posting-Host	2453	Related to sender	Try?
NNTP-Posting-Host	2311	Same as previous except for capitalization	Try?
Reply-To	1720	Possible cue, not common, experiment with this	Try?
Keywords	926	Describes content	Keep
Article-I.D.	673	Too specific	Drop
X-Newsreader	588	Software used by sender	Drop
Summary	391	Like Keywords	Keep
Originator	291	Similar to From, but much less common	Drop
In-Reply-To	219	Probably just an id number, rare	Drop
News-Software	164	Software used by sender	Drop
Expires	113	Date specific, rare	Drop
In-reply-to	101	Probably just an id number, rare	Drop
To	80	Possible target leak, rare	Drop
X-Disclaimer	64	Noise, rare	Drop
Disclaimer	56	Possible cue due to newsgroup paranoia level, rare	Drop
...		...	
Weather	1	Bizarre header	Drop

Orginization	1	Spelling errors in headers are surprising	Drop
Oganization	1	Well, maybe not	Drop
Oanization	1		
Moon-Phase	1	Yes, really. One posting had this. Open source is wonderful.	!?!

Many headers are unlikely to be helpful, and many are too rare to have much effect. Table 14.3 marks "Subject", "From", "Lines", "Keywords", and "Summary" as potentially interesting feature headers. These occur fairly often and seem likely to relate to the content of the document. The oddball among these is "Lines", which is actually content related since some newsgroups tend to long documents and some to short.

> SIDEBAR: Preliminary analysis of data is critical to successful classification. It is sometimes "fun" because the analysis often turns up Easter eggs like the "Moon-Phase" header line above. These surprises can also be important in building a classifier because they can uncover problems in the data, or give you a key insight that simplifies the classification problem. Visualize early and visualize often.

Note that the 20 newsgroups example has been carefully prepared to be easy to use for testing classifiers. As such, it is not entirely realistic. You can see that no preprocessing is needed because all of the data are in one place and all obvious target leaks have already been removed. So you can proceed to text analysis.

14.4.2 Parsing and tokenizing features for 20 newsgroups data

The second phase of preparing data for presentation to the learning algorithm is converting classifiable data in each of the four possible types into vectors. For the 20 newsgroups data set, all of the fields besides "Lines" are text-like or word-like and all seem to be in a form that will tokenize easily using the standard Lucene tokenizer. The "Lines" field is a number and it can be parsed using the Lucene tokenizer as well. It appears that this field would be quite helpful in distinguishing news groups since the average number of lines per message for the `talk.politics.*` groups is, for example, about 60 lines while the average number of lines for documents in `misc.forsale` is only 25 lines.

14.4.3 Training code for 20 newsgroups data

Now you can build a model. For simplicity, you will use the same `OnlineLogisticRegression` learning algorithm as you used in the previous chapter for classifying dots. This time, however, you will run the classifier from Java code instead of the command line so that you can see how features are extracted and vectorized.

> Sidebar Logistic regression classifiers use a linear combination of input values whose value is compressed to the (0,1) range using the logistic function, $1 / (1+e^{-x})$. The output of a logistic regression model an often be interpreted as probability estimates. In addition, the weights used in the linear combination can be efficiently computed in an incremental fashion even when the feature vector has high dimension. This makes logistic regression a popular choice for scalable sequential learning. The features given to the logistic regression must be in numerical form, so text, words and categorical values must be encoded in vector form for use with a logistic regression model.

SETTING UP VECTOR ENCODERS

First, you need objects that will help convert text and line counts into vector values, as follows.

```
Map<String, Set<Integer>> traceDictionary =
  New TreeMap<String, Set<Integer>>();
RecordValueEncoder encoder = new StaticWordValueEncoder("body"); #A
encoder.setProbes(2);
encoder.setTraceDictionary(traceDictionary);
```

```
RecordValueEncoder bias = new ConstantValueEncoder("Intercept"); #B
bias.setTraceDictionary(traceDictionary);
RecordValueEncoder lines = new ConstantValueEncoder("Lines");    #C
lines.setTraceDictionary(traceDictionary);

Dictionary newsGroups = new Dictionary();
```

#A encoder will convert text
#B bias will implement constant value
#C include the line count

Since the `OnlineLogisticRegression` expects to get integer class IDs for the target variable during training, you will also need a dictionary to convert the target variable (the newsgroup) to an integer, which is the `newsGroups` object above.

CONFIGURE THE LEARNING ALGORITHM

You can configure the logistic regression learning algorithm as follows.

```
OnlineLogisticRegression learningAlgorithm =
    new OnlineLogisticRegression(
          20, FEATURES, new L1())          #A
        .alpha(1).stepOffset(1000)         #1
        .forgettingExponent(0.9)
        .lambda(3.0e-5)                    #2
        .learningRate(20);                 #3
```

#A number of categories and features
#1 initial power law decay is slow
#2 amount of regularization
#3 initial learning rate

Configuring the learning algorithm requires picking a number of parameter values such as the initial learning rate #1 or the amount of regularization #2. In a production model, the learning algorithm would run hundreds or thousands of times to find good values. Here, a few quick trials will suffice to find reasonable values, and this is how these values were chosen.

ACCESS DATA FILES

Next, you need a list of all of the training data files, as follows

```
List<File> files = new ArrayList<File>();
for (File newsgroup : base.listFiles()) {
  newsGroups.intern(newsgroup.getName());              #A
  files.addAll(Arrays.asList(newsgroup.listFiles()));
}
Random rand = new Random();
System.out.printf("%d training files\n", files.size());
```

#A assign 1 of N encoding to each newsgroup

This list will allow you to randomize the order that training examples are given to the learning algorithm. You can also enter all of the newsgroups' names into the dictionary. Predefining the contents of the dictionary like this ensures that the entries in the dictionary are in a stable and recognizable order. This helps when comparing results from training run to training run.

PREPARING TO TOKENIZE

Most of the data in these files are textual, so we can use Lucene to tokenize this text. Using Lucene is better than simply splitting on whitespace and punctuation because you can use the `StandardAnalyzer` class from Lucene, so that special tokens like email addresses are treated correctly.

You also will need several variables to accumulate averages for progress outputs like average log-likelihood, percent correct, document line count, as well as the number of documents process, like this

```
double averageLL = 0.0;
double averageCorrect = 0.0;
double averageLineCount = 0.0;
int k = 0;
double step = 0.0;
int[] bumps = new int[]{1, 2, 5};
```

These variables will allow you to measure progress and performance of the learning algorithm as it proceeds.

READING AND TOKENIZING THE DATA

You are ready to process the data. You will make only one pass through the data since the online logistic regression algorithm learns quickly. Making only a single pass also facilitates progress evaluation, since each document can be tested against the current state classifier before it is used as training data. To do the actual learning, documents are processed in a random order. Presenting training data in a random order helps the `OnlineLogisticRegression` converge to a solution more quickly. This is shown in Listing 14.9.

Listing 14.9 Parsing

```
for (File file : permute(files, rand)) {
  BufferedReader reader = new BufferedReader(new FileReader(file));
  String ng = file.getParentFile().getName();     #1
  int actual = newsGroups.intern(ng);
  Multiset<String> words = ConcurrentHashMultiset.create();
  double lineCount;
  String line = reader.readLine();
  while (line != null && line.length() > 0) {
    if (line.startsWith("Lines:")) {              #2
      String count = Iterables.get(onColon.split(line), 1);
      try {
        lineCount = Integer.parseInt(count);
        averageLineCount += (lineCount - averageLineCount)
            / Math.min(k + 1, 1000);
      } catch (NumberFormatException e) {
        lineCount = averageLineCount;
      }
    }
    boolean countHeader = (
        line.startsWith("From:") || line.startsWith("Subject:")||
        line.startsWith("Keywords:")|| line.startsWith("Summary:"));
    do {
      StringReader in = new StringReader(line);
      if (countHeader) {
        countWords(analyzer, words, in);    #3
      }
      line = reader.readLine();
    } while (line.startsWith(" "));
  }
  countWords(analyzer, words, reader);      #4
  reader.close();
}
```

#1 filename gives newsgroup
#2 Check for line count header
#3 Count words in important header lines
#4 Count words in body text.

The file name can be used to tell which newsgroup each document belongs to. There is a header line that gives the number of lines, which can be encoded as a feature. Then, in parsing header lines and the body of the document, you need to count the words that you find; you can use a multi-set from the Google Guava library. At least one of the documents has a bogus line count, in that case so the line count to the overall average to be defensive. In each document, the headers come first and consist of lines containing a header name, a colon, content plus possible continuation lines that start with a space. The line number and the counts of the words on selected header lines are needed as features. After processing the headers, the body of the document is passed through the Lucene Analyzer.

VECTORIZING

With the data from the document in hand, you are ready to collect all of the features into a single vector for use by the classifier learning algorithm as follows:

```
Vector v = new RandomAccessSparseVector(FEATURES);
bias.addToVector(null, 1, v);                 #1
lines.addToVector(null, lineCount / 30, v);   #2
logLines.addToVector(null, Math.log(lineCount + 1), v); #2
for (String word : words.elementSet()) {
  encoder.addToVector(word, Math.log(1 + words.count(word)), v);  #3
}
```

#1 bias term
#2 line count and log
#3 word counts, weighted by log frequency

First, encode the bias (constant) term, which always has a value of 1. This feature can be used by the learning algorithm as a threshold. Some problems cannot be solved using logistic regression without such a term. The line length is encoded both in raw form and as a logarithm. The division by 30 is done to put the line length into roughly the same range as other inputs so that learning will progress more quickly. The body of the document is encoded similarly except that the weight applied to each word in the document is based on the log of the frequency of the word in the document rather than the straight frequency. This is done because words occur more than once in a single document much more frequently than would be expected by the overall frequency of the word. The log of the frequency takes this into account.

MEASURING PROGRESS SO FAR

At this point, you should have a vector and be ready to give it to the learning algorithm. Until then, the document can still be considered to be held-out data. As such, you might as well use it to get some feedback about the (hopefully improving) accuracy of the classifier. Here, you compute two measures of accuracy, the log-likelihood and the average rate of correct classification.

Log-likelihood has a maximum value of 0 and random guessing with 20 alternatives should give a result of close to -3. Log-likelihood "gives credit" even when the classifier gets the wrong answer if the right answer is near the top and it takes credit away if it gets the right answer but has a wrong answer with nearly as high a score. These close-but-not-quite properties of log-likelihood make it a useful measure for performance testing. On the other hand, it can be difficult to explain log-likelihood to non-technical colleagues, so it is good to have a simpler measure, the average rate of correct answers, as well.

You can compute the averages of the performance metrics here using a trick known as Welford's algorithm (see Listing 14.11), which allows us to always have an estimate of the current average. A variant on Welford's algorithm is used here so that until you have processed 200 examples, a straight average is used. After that, exponential averaging is used so that you get a progressive measure of performance that eventually forgets about the early results before the classifier has learned anything.

Listing 14.11 Welford's algorithm

```
double mu = Math.min(k + 1, 200);
double ll = learningAlgorithm.logLikelihood(actual, v);  #1
averageLL = averageLL + (ll - averageLL) / mu;

Vector p = new DenseVector(20);
learningAlgorithm.classifyFull(p, v);                    #2
int estimated = p.maxValueIndex();                       #2

int correct = (estimated == actual? 1 : 0;
averageCorrect = averageCorrect + (correct - averageCorrect) / mu;  #3
```

#1 instantaneous log-likelihood
#2 get probabilities to find max
#3 compute average percent correct

In this snippet, the model is asked for some performance metrics. The model computes the log-likelihood. Then the most highly rated newsgroup is determined and then compared to the correct value and averaged to get average percent correct.

TRAINING THE SGD MODEL WITH THE ENCODED DATA

After getting progress information from the current training example, you can pass it to the learning algorithm to update the model. See Listing 14.12. If making multiple passes through the data, it would probably be a good idea to use testing examples for progress monitoring and training examples only for training rather than for both training and progress monitoring as you have done here.

Listing 14.12 Training the model

```
learningAlgorithm.train(actual, v);
k++;
```

```
int bump = bumps[(int) Math.floor(step) % bumps.length];
int scale = Math.pow(10, Math.floor(step / bumps.length));
if (k % (bump * scale) == 0) {
  step += 0.25;
  System.out.printf("%10d %10.3f %10.3f %10.2f %s %s\n",
       k, ll, averageLL, averageCorrect * 100, ng,
       newsGroups.values().get(estimated));
}
learningAlgorithm.close();
```

From there, you can output your current status. As the learning progresses, the frequency of status reports will decrease. This means that where accuracy is changing more quickly, there will be more frequent updates. Importantly, you have to tell the learning algorithm that it is done. This causes any delayed learning to take effect and cleans up any temporary structures.

14.5 Choosing the algorithm to train the classifier

The main advantage of Mahout is its robust handling of extremely large and growing data sets. The algorithms in Mahout all share scalability, but they differ from each other in other characteristics, and these differences give different advantages or drawbacks. Table 14.4 compares the algorithms used for classification within Mahout. This table, along with the rest of this section, help you decide which Mahout algorithm bests suits a particular classification problem. Keep in mind that Mahout has additional algorithms not listed here, since new ones continue to be developed.

Table 14.4 Characteristics of the Mahout learning algorithms used for classification. The algorithms differ somewhat in the overhead or cost of training, the size of the data set for which they are most efficient or the complexity of analysis they can deliver.

Size of Data Set	Mahout Algorithm	Execution Model	Characteristics
Small to Medium	Stochastic Gradient Descent (SGD) family: OnlineGradientDescent, CrossFoldLearner, AdaptiveLogisticRegression	Sequential, on-line, incremental	Uses all types of predictor variables, sleek and efficient over the appropriate data range (up to millions of training examples)
	Support Vector Machine (SVM)	Sequential	Experimental still; Sleek and efficient over the appropriate data range
Medium to Large	Naïve Bayes	Parallel	Strongly prefers text-like data; medium to high overhead for training; effective and useful for data sets too large for SGD or SVM.
	Complementary Naïve Bayes	Parallel	Somewhat more expensive to train than Naïve Bayes; effective and useful for data sets too large for SGD, but has similar limitations as Naïve Bayes
Large to Very Large	Random Forest	Parallel	Uses all types of predictor variables; high overhead for training; not widely used (yet); costly but offers complex and interesting classification. Handles non-linear and conditional

relationships in data better than other techniques.

The algorithms differ somewhat in the overhead or cost of training, the size of the data set for which they are most efficient or the complexity of analysis they can deliver.

14.5.1 Nonparallel but powerful: Using SGD and SVM

As you saw in Figures 13.1 and 13.7 of the previous chapter, the behavior of an algorithm can be powerfully scalable even if the algorithm is nonparallel. This section provides and overview of two sequential Mahout learning algorithms, Stochastic Gradient Descent (SGD) and Support Vector Machine (SVM).

THE STOCHASTIC GRADIENT DESCENT ALGORITHM

Stochastic Gradient Descent is a widely used learning algorithm in which each training example is used to tweak the model slightly to give a more correct answer for that one example. This incremental approach is repeated over many training examples. With some special tricks to decide how much to nudge the model, the model accurately classifies new data after seeing only a modest number of examples. While SGD algorithms are difficult to parallelize effectively, they are often so fast that for a wide variety of applications, parallel execution is not necessary.

Importantly, because these algorithms do the same simple operation for each training example, they require a constant amount of memory. For this reason, each training example requires roughly the same amount of work. These properties make SGD-based algorithms scalable in the sense that twice as much data takes only twice as long to process.

THE SUPPORT VECTOR MACHINE ALGORITHM

An experimental sequential implementation Support Vector Machine (SVM) algorithm, has recently been added to Mahout. SVM implementations in Mahout include Java translations of the industrial-strength but non-scalable LIBLINEAR that was previously only available in C++. The SVM implementation is still new and should be tested before deploying.

The behavior of the SVM algorithm will likely be similar to SGD in that the implementation is sequential, and that the training speed for large data sizes will probably be somewhat slower than SGD. The Mahout SVM implementation will likely share the input flexibility and linear scaling of SGD and thus will probably be a better choice than Naïve Bayes for moderate scale projects.

14.5.3 Power of the Naïve classifier: using Naïve Bayes and Complementary Naïve Bayes

The Naïve Bayes and Complementary Naïve Bayes algorithms in Mahout are parallelized algorithms that can be applied to larger data sets than are practical with SGD-based algorithms, as seen in table 14.4. Since they can work effectively on multiple machines at once, these algorithms will scale to much larger training data than will the SGD-based algorithms.

The Mahout implementation of Naïve Bayes, however, is restricted to classification based on a single text-like variable. For many problems, especially typical large data problems, this requirement is not a problem. If continuous variables are needed and cannot be quantized into word-like objects that could be lumped in with other text data, then it may not be possible to use the Naïve Bayes family of algorithms.

In addition, if the data have more than one categorical, word-like or text-like variable, it is possible to simply concatenate your variables together, disambiguating them by prefixing them in an unambiguous way. This approach may lose important distinctions, however, because the statistics of all of the words and categories get lumped together. Most text classification problems, however, should work well with Naïve Bayes or Complementary Naïve Bayes.

> Tip: If you have a more than 10 million training examples and the predictor variable is a single, text-like value, Naïve Bayes or Complementary Naïve Bayes may be your best choice of algorithm. For other types of variables, or less training data try SGD.

If the constraints of the Mahout Naïve Bayes algorithms fit your problem, they are good first choices to test. They are good candidates for data with more than 100,000 training examples and are likely to outperform sequential algorithms at 10 million training examples.

14.5.3 Strength in elaborate structure: using Random Forest

Mahout has sequential and parallel implementations of Leo Breiman's Random Forest algorithm as well. This algorithm trains an enormous number of simple classifiers, and uses a voting scheme to get a single result. The Mahout parallel implementation trains the many classifiers in the model in parallel. This approach has somewhat unusual scaling properties. Since each small classifier is trained on some of the features of all of the training examples, the memory required on each node in the cluster will scale roughly in proportion to the square root of the number of training examples. This is not quite as good as Naïve Bayes, where memory requirements are proportional to the number of unique words seen and thus are proportional to more like the logarithm of the number of training examples.

In return for this less desirable scaling property, Random Forest models have more power when it comes to problems that are difficult for logistic regression, SVM or Naïve Bayes. Typically these problems require a model to use variable interactions and discretization to handle threshold effects in continuous variables. Simpler models can handle these effects with enough time and effort developing variable transformations, but Random Forests can often deal with these problems without that effort.

Now that you are familiar with the options Mahout offers for learning algorithms used to train classifiers, it is time to put that knowledge to work. Compare what happened when you used the SGD algorithm to train a classifier for 20 newsgroups data in section 14.4 with the results that you will get using a different algorithm for the same problem.

14.6 Classifying 20 newsgroups data with Naïve Bayes

The route to classification depends in part on the Mahout classification algorithm used. Whether the algorithm is parallel or sequential, as described in the preceding section, makes a big difference. The parallel SGD algorithm and the sequential Naïve Bayes algorithm have alternative input paths. This difference, in turn, requires differ approaches to the handling of data, particularly with regard to vectorization.

This section shows how to train a Naïve Bayes model on the same 20 newsgroups data that you used in the example in section 14.4. By using a different algorithm on the same data, you will highlight the differences between sequential and parallel approaches for building a classification model. This section first guides you through the data extraction step needed to get data ready for the training algorithm and then explains how to train the model. Once done, you will be ready to begin the process of evaluating your initial model to determine if it is performing well or what changes are useful to make.

14.6.1 Getting started: data extraction for Naïve Bayes

Here you will start the process of getting data into a classifiable form and converting it to a file format for use with the Naïve Bayes algorithm. Use the same 20 newsgroups data you downloaded for the SGD example in section 14.4. The Naïve Bayes classifiers can be driven programmatically as you did with the SGD classifier, but it also has a parser built in that can be invoked from the command line. This parser accepts a variant on the data file format used by the SVMLight program in which each data record consists of a single line in the file. Each line contains the value of the target variable followed by space-delimited features, where the presence of the feature name indicates that the feature has a value of 1 and absence indicates that that feature has a value of 0. In order to use the command line version of the Naïve Bayes classifier, you have to reformat the data in the 20newsgroups data set to be in this format.

The data in the 20 newsgroups data set has one directory per newsgroup and inside each directory, one document per file. You need to scan each directory and transform each file into a single line of text that starts with the directory name and then contains all the words of the document. The Mahout program called `prepare20newsgroups` does just this. To convert the training and test data, use something like this:

```
$ bin/mahout prepare20newsgroups -p 20news-bydate-train/ \        #A
       -o 20news-train/ \                                        #B
```

```
        -a org.apache.lucene.analysis.standard.StandardAnalyzer \  #C
        -c UTF-8                                                   #D

no HADOOP_CONF_DIR or HADOOP_HOME set, running locally
INFO: Program took 3713 ms

$ bin/mahout prepare20newsgroups -p 20news-bydate-test \
        -o 20news-test \
        -a org.apache.lucene.analysis.standard.StandardAnalyzer \
        -c UTF-8
no HADOOP_CONF_DIR or HADOOP_HOME set, running locally
INFO: Program took 2436 ms
```

#A input directory
#B output directory
#C how text will be tokenized
#D character encoding

The result should be a directory called `20news-train` that contains one file per newsgroup. Inside a data file like `20news-train/misc.forsale.txt`, you see something like what is shown in Figure 14.7.

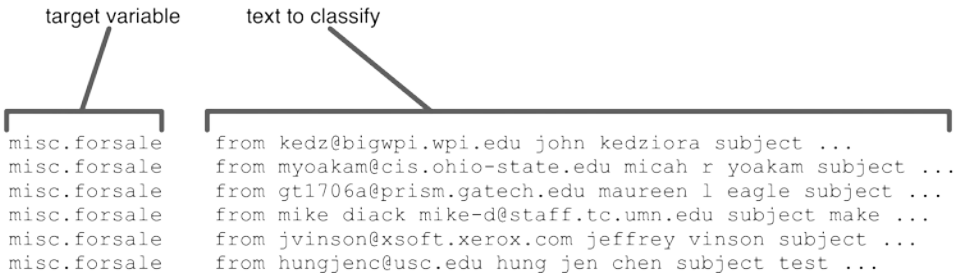


Figure 14.7 Converted training data in format preferred by Naïve Bayes program. Note that the lines displayed here have been shortened considerably.

14.6.2 Training the Naïve Bayes classifier

Up to this point you have been working with the 20 newsgroup data doing feature extraction to prepare data appropriately for input to the Naïve Bayes algorithm. With training and test data converted to the right format, you can now train a classification model.

To use with the Naïve Bayes algorithm, try this command to produce a trained model:

Listing 14.14 Training a Naïve Bayes model

```
bin/mahout trainclassifier -i 20news-train \
           -o 20news-model \
           -type cbayes \                          #A
           -ng 1 \                                 #B
           -source hdfs                            #C
...
INFO: Program took 250104 ms
```

#A type of model
#B size overlapping n-grams
#C where to get the data

The result is a model stored in the directory `20news-model`. This model consists of several files that contain the components of the model. These files are in a binary format and cannot be easily inspected directly, but you can use them to classify the test data with the help of the `testclassifier` program.

Now you have built and initially trained a model using 20 newsgroups data and the Naïve Bayes algorithm. How well is it working? You are ready to address that question by beginning to evaluate performance of your model.

14.6.3 How to test a Naïve Bayes model

In this section of the chapter you will gain experience with the second stage of classification, evaluating the performance of your newly trained model. This topic is dealt with in depth throughout Chapter 15.

Here you will experience just an introduction to the process as you do some initial testing on the newly trained model you have just built.

To run the Naïve Bayes model on the test data you can use the following command:

```
bin/mahout testclassifier -d 20news-test \
            -m 20news-model \                       #A
            -type cbayes \
            -ng 1 \
            -source hdfs \
            -method sequential                         #B
```

#A model computed during training
#B runs without Hadoop

When the test program finishes, it produces output on standard out like what appears below. The summary has raw counts of how many documents were classified correctly or incorrectly.

```
...
Summary
-------------------------------------------------------
Correctly Classified Instances          :       6398       84.9442%
Incorrectly Classified Instances        :       1134         15.0558%
Total Classified Instances              :       7532
=======================================================
```

We have omitted some noisy output and replaced it with "…". Here, the Naïve Bayes model is performing well, with a score of nearly 85% correct. The summary does not tell anything about the errors that are made other than the gross number of errors. For more details about what errors were made, we need to look to the confusion matrix in the next section of output, which looks something like the following.

```
Confusion Matrix
-------------------------------------------------------
a b c d e f g h i j k l m n o p q r s t <--Classified as
388                                      | 397 a = rec.sport.baseball
  386                                    | 396 b = sci.crypt
    396                                  | 399 c = rec.sport.hockey
      347                                | 364 d = talk.politics.guns
        377                              | 398 e = soc.religion.christian
  12      304           18               | 393 f = sci.electronics
            281     14  43        21     | 394 g = comp.os.ms-windows.misc
              313       22 16            | 390 h = misc.forsale
     26 69      83 41                    | 251 i = talk.religion.misc
        45        225 13             11  | 319 j = alt.atheism
                    334           32     | 395 k = comp.windows.x
                      367                | 376 l = talk.politics.mideast
          19 23 15      307 13           | 392 m =comp.sys.ibm.pc.hardware
                         16 335          | 385 n = comp.sys.mac.hardware
                              371        | 394 o = sci.space
                                393      | 398 p = rec.motorcycles
                               12 364    | 396 q = rec.autos
            11      22            305    | 389 r = comp.graphics
      102                           160   | 310 s = talk.politics.misc
                                      362 | 396 t = sci.med
Default Category: unknown: 20
```

Your output will look a little different because we squashed the actual output here to fit on shorter lines so you can read it. The confusion matrix shows a complete breakout of all correct and incorrect classifications so that we can see what kinds of errors the model made on the test data. Along the diagonal, we see that most newsgroups are classified fairly well, although `talk.religion.misc` and `talk.politics.misc` both stand out somewhat because of the relatively smaller number of correct classifications. About a third of the documents from `talk.politics.misc` are incorrectly classified as belonging to `talk.politics.guns`, which is at least plausible if not correct. Likewise, 69 documents from `talk.religion.misc` are classified as `soc.religion.christian`, which is, again, plausible. It is reassuring to see that the errors make a sort of sense because that indicates that the model is picking up on the actual content of the documents in some meaningful way.

It is also ironically reassuring to see that the performance of the model is not suspiciously good. For instance, if you run the same model on the same training data that it learned from, the summary looks like this:

```
bin/mahout testclassifier -d 20news-train -m 20news-model -type cbayes -ng 1 -source
hdfs -method sequential
...
Correctly Classified Instances          :      11075       97.8876%
Incorrectly Classified Instances        :        239          2.1124%
Total Classified Instances              :      11314
```

When given this familiar data, the model is able to get nearly 98% correct, which is much too good to be true on this particular problem. The best machine learning researchers only claim accuracies for their systems in the around 84-86%. We will talk more about how to evaluate models in later chapters and, in particular, how to develop an idea about how well the model really should be performing.

14.5 Summary

In working through this chapter, you have gained experience with building and training classifiers using real data from the 20 newsgroups data set. The chapter focused particularly on how to do feature extraction and how to choose the best algorithm for your particular project.

Feature extraction is a big part of building a classification system, both in time, effort, and the payoff of doing it well. Remember that not all features are equally useful for every classifier, and that you will try a variety of combinations to see what works. Raw data are not usable directly. The chapter showed in detail how to process raw data into classifiable form and how to convert that into the vectors required by the learning algorithms of Mahout.

The types of values for input variables that are acceptable are different for different for different learning algorithm, and section 14.5 provided an overview of those and other differences that distinguish one algorithm from another. You should now have a basic understanding of the benefits and costs of using the different algorithms to help guide you through making these choices in your own projects.

Finally, the chapter provided a comparison of the behavior of two different learning algorithms (SGD and Naïve Bayes) that you used to train the same data set, the 20 newsgroups. Having learned a lot about the first stage of classification, training a model, you are now ready to move on to the second stage, evaluation and tuning the performance of a trained model. Those are the topics of the next chapter.

15

Evaluating and tuning a classifier

This chapter covers

- Basic considerations in classifier evaluation
- The Mahout evaluation api
- An example of how to measure performance of an sgd classifier
- Examples of common problems and how to diagnose them
- Approaches for tuning a classifier

Because evaluation is so important, it is built in at a fundamental level in Mahout. This chapter deals mainly with stage 2 of the classification process, the evaluation and fine-tuning of a classifier to prepare for deployment and to maintain performance in production. We will include how classifiers are evaluated at a high level as well as details of the Mahout API for evaluation and an example of how to use the Mahout API. We also will present several examples that highlight how performance metrics and diagnostic capabilities of the Mahout evaluation API can be used to diagnose common problems with classifiers. We end with a discussion of classifier tuning strategies and techniques that cover the range of tuning techniques from choice of algorithm to adjustment of learning rates.

Evaluation presents several pitfalls, and this chapter also offers ways to avoid the more costly ones. Evaluation is also challenging because classification model internals can be difficult to understand.

15.1 Classifier evaluation in Mahout

Evaluation is an essential part of building a successful classifier. It is far more than a test to determine a single score of success after you have trained a model. Instead, it is an iterative process that begins during training of a classifier. Early evaluation is valuable when building classifiers because it encourages trying out many options including different algorithms, settings and sets of predictor variables. Evaluation also validates the data pre-preprocessing pipeline for errors that cause the classifier to appear to perform better than it actually will.

To evaluate classifiers, Mahout offers a variety of performance metrics. The main approaches are percent correct, confusion matrix, AUC and log-likelihood. The classifier algorithms Naïve Bayes and Complementary Naïve Bayes are best evaluated using percent correct and confusion matrix. With the algorithm SGD, any of these methods will work. AUC or log likelihood may be particularly useful as they provide insight into the model's confidence level.

This section covers how to get feedback on model performance, even while training is progressing, how to interpret the performance metrics that Mahout provides and how to integrate cost considerations into your evaluation.

15.1.1 Getting rapid feedback

Evaluation in Mahout works a bit differently than in other systems. One useful difference is its ability to provide immediate and ongoing assessment of performance by allowing a user to pass target variable scores and reference values to an evaluator one at a time to get on-the-fly performance feedback. This Mahout approach is in contrast to other systems that must process all of the scores and target values in a batch mode to evaluate performance. Mahout's ability to do on-line evaluation is convenient when you need to record progress of a learning system, but it also has practical benefits in that it allows Mahout to adapt learning parameters on the fly as learning progresses.

The ability to get in-progress metrics is nice, but what do these metrics mean?

15.1.2 Deciding what "good" means

Intuitively, most of us want a classifier to be as accurate as possible, to put items into the correct category every time. Unfortunately, this intuition does not lead to a practical evaluation scheme. In the real world, no classifier is going to be 100% accurate. In fact, very high accuracy typically indicates something is wrong with the test -- if it seems too good to be true, it probably is! So, how good is good? Useful evaluation requires realistic benchmarks for success so that you can recognize when a classifier is going to be effective.

Simple accuracy metrics such as percent correct are not necessarily the best way to evaluate and tune a system. For example, consider two hypothetical models illustrated in Table 15.1. A comparison of the two models in this table shows how a model that is "always wrong" based on a simple accuracy metric can be substantially more useful than a model that is sometimes right.

Table 15.1 Data from two hypothetical classifiers to show some of the limitations of just looking at percent correct. The answer with the highest score is in bold. Model 1 is never quite right, but may still be useful while model 2 is like a stopped clock..

	Model 1			Model 2		
Correct	**A**	**B**	**C**	**A**	**B**	**C**
A	0.45	**0.50**	0.05	0.01	0.01	**0.99**
B	**0.50**	0.45	0.05	0.01	0.01	**0.99**
C	0.05	**0.50**	0.45	0.01	0.01	**0.99**
A	0.45	**0.50**	0.05	0.01	0.01	**0.99**

This hypothetical example shows why simply looking at the percent accuracy often does not reveal the true value of a model. In this table, each row represents an example with the correct answer on the far left. Each column contains the score for each of the 3 possible answers. Model 1 is never quite "right" yet it is much more useful than Model 2 despite the higher apparent accuracy of the latter. Model 2 is scoring with a simplistic fixed rule and is correct part of the time purely at random. In contrast, Model 1 does appear to consistently eliminate one of three possibilities that is not the right answer.

For many applications, Model 1's output is probably better for many purposes than the Model 2, which gets the right answer (sometimes) while never changing what it says. For this example, Model 1 got 0% correct and Model 2 got 25% correct. In contrast, Model 1 has an average log-likelihood of -0.8 while Model 2 has the considerably worse value of -3.5. In the next section, we discuss measures of success like log-likelihood that better reflect the types of performance we would like to see.

Further, it is useful for a model to know when it is likely to be correct and when it does not have a good guess. There are performance measures that can reward this kind of self-knowledge. Model 1 above provides this information by showing it is nearly ambivalent about two choices, one of which is in fact right, while Model 2 provides seems to have no clue about whether it is right or wrong. Again, log-likelihood highlights this difference correctly.

Sometimes, however, a universal measure of performance isn't the right thing. This happens commonly when some errors are much worse than others.

15.1.3 Recognizing the difference in cost of errors

One issue that clouds the usefulness of simple accuracy metrics is that some errors cost more than others. The cost of a false positive may be much less than the cost of a false negative. For example, finding a cancer that does not exist has an expense in terms of retesting costs and anxiety, but not finding a cancer that does exist can cost a life.

A less dramatic example of the differential cost for errors of omission and commission is spam detection. The user of a spam detector is likely to grumble if spam is marked as non-spam, but a user will be furious if a non-spam message is marked as spam. Spam which is misclassified as non-spam wastes tens of seconds of the user's time for each error, but a non-spam message that is marked as spam is much more than an inconvenience. If this is frequent, the user likely will discover the problem and take action. If, however, such errors are rare, then users may not be aware of the need to compensate for their possibility, and the consequences will escalate.

The output of the Mahout classifier evaluation API was shown in action in the examples in chapters 13 and 14. The command line tools we demonstrated in those chapters provided evaluation output. In some cases, those standard diagnostics suffice, but generally you will need to use the API in your own programs to evaluate how things are going. The next section provides a detailed guide on how to use API-based evaluation in Mahout.

15.2 The classifier evaluation API

Mahout's classifier evaluation API is a collection of classes that can compute various classifier performance metrics from your code whether or not you use the Mahout classifiers. The Mahout API classes that support various classifier metrics are shown in Table 15.2. Each metric is described in more detail in the discussion throughout this section of the chapter. These topics include how to compute performance metrics in both on-line and off-line applications. The online learning algorithms also provide API access to several of these performance metrics during training.

Table 15.2 Mahout supports a variety of classifier performance metrics through multiple APIs.

Metric	Supported by class
%-correct	`ConfusionMatrix`
confusion matrix	`ConfusionMatrix, Auc`
entropy matrix	`Auc`
AUC	`Auc, OnlineAuc, CrossFoldLearner, AdaptiveLogisticRegression`
log-likelihood	`CrossFoldLearner, AdaptiveLogisticRegression`

As you can see in Table 15.2, each metric except for entropy matrix is provided in different ways by multiple classes. The capabilities are listed by class in Table 15.3.

Table 15.3 The Mahout classes that support performance evaluation for classifiers.

Class	Description
`Auc`	Computes batch AUC by sampling. Also computes confusion and entropy matrices for binary target variable.
`OnlineAuc`	Computes on-line AUC by sampling and averaging

`OnlineSummarizer`	Computes average, standard deviation, quartiles, median
`ConfusionMatrix`	Computes confusion matrix, %-correct, totals. Used by NaiveBayes for result summary.
`AbstractVectorClassifier`	Computes log-likelihood for single example. Typically used with `OnlineSummarizer`.
`CrossFoldLearner`	Uses cross-validation to get on-the-fly metrics. See `auc()` and `loglikelihood()` methods.
`AdaptiveLogisticRegression`	Maintains a pool of `CrossFoldLearner` objects to tune learning parameters. Uses the estimates of performance from these during learning to tune parameters. See `auc()` and `loglikelihood()` methods.

Note that the use of these classes varies. The learning algorithms including `AdaptiveLogisticRegression`, `CrossFoldLearner` and `AbstractVectorClassifier` all provide evaluation methods for the model being learned. In contrast, `Auc` and `OnlineAuc` compute performance metrics given scores and reference target values. Finally, the `OnlineSummarizer` computes aggregate statistics for arbitrary measurements. So now let's take a look at how to compute these metrics in detail.

15.2.1 Computation of AUC

The AUC metric is useful when evaluating a model that produces a continuous score for a binary target variable. For example, a classifier that computes a probability between 0 and 1 that an item has a certain attribute or not fits this description. The score need not be a probability estimate, and models with very different score ranges can be compared using AUC.

The meaning of AUC is not vital to understand in depth. It is sufficient to remember that AUC is the probability that a randomly chosen "yes" instance will have a higher score than a randomly chosen "no" instance. A model that produces scores with no correlation to the target variable will produce an AUC of about 0.5, while a perfect model will produce an AUC of 1. An AUC in the range of 0.7 to 0.9 is typically considered good. For fraud detection or click-through analysis this is the level that a model produces usefully accurate results. AUC can have values as low as 0 in cases where the model is actually predicting the opposite of the target variable.

Practical and theoretical considerations dictate that an AUC is best computed by holding out some subset of all data as test data, and comparing the classifier's result on the test data against the known value. Unfortunately, AUC is normally limited to models with binary target variables that produce a score rather than a hard classification. There are extensions to AUC to allow non-binary target variables, but Mahout does not support these extensions yet.

In Mahout, if scores and correct target values for test data are available, then `Auc` from the `org.apache.mahout.classifier.evaluation` package or `OnlineAuc` from the `org.apache.mahout.math.stats` package can be used to compute AUC. In both cases, you need to feed the actual value of the target variable from held-out test data and the score for that example. For both classes, the value of the AUC metric is obtained from the `auc()` method.

`OnlineAuc` differs from `Auc` in that estimates of AUC are available at any time during the loading of data with little overhead, though insertion of each element is slightly more expensive. The algorithms used in `Auc` and `OnlineAuc` provide essentially identical accuracy, but `Auc.auc()` is more expensive to run than `OnlineAuc.auc()` because it involves sorting several thousand samples.

The code in Listing 15.1 shows how scores and target variable values can be read from a file to compute AUC using both classes.

Listing 15.1 Passing data for AUC metric classes

```
Auc x1 = new Auc();
OnlineAuc x2 = new OnlineAuc();
BufferedReader in = new BufferedReader(new FileReader(inputFile));
int lineCount = 0;
String line = in.readLine();
while (line != null) {
  lineCount++;
  String[] pieces = line.split(",");
  double score = Double.parseDouble(pieces[0]);
  int target = Integer.parseInt(pieces[1]);
  x1.add(target, score);
  x2.addSample(target, score);
  if (lineCount%500 == 0) {
    System.out.printf("%10d\t%10.3f\t%10d\t%.3f\n",  #1
                lineCount, score, target, x2.auc());
  }
  line = in.readLine();
}

System.out.printf("%d lines read\n", lineCount);
System.out.printf("%10.2f = batch estimate\n", x1.auc())
System.out.printf("%10.2f = on-line estimate\n", x2.auc());
```

#1 x2 computes AUC progressively as input is read

In this code, `x1` and `x2` compute nearly the same estimate of AUC, but with `x2`, a progressive estimate #1 of the value is available as the data is being scanned. At the end #2 both `x1` and `x2` give essentially the same estimate by somewhat different methods.

A recently computed value of AUC or log-likelihood is available during training from the `auc()` and `loglikelihood()` methods of the `CrossFoldLearner` and `AdaptiveLogisticRegression` classes. Accessing these values allows comparison among models as they are trained. The `AdaptiveLogisticRegression` class does this so that it can adapt the training parameters during training.

While AUC is typically the gold standard for binary target variables and scored outputs, where there are more than two possible categories for the target variable, or where there are no scores, then we need to look for another good metric.

15.2.2 Confusion matrices and entropy matrices

Probably the most straightforward metric for unscored outputs is the confusion matrix. A confusion matrix has rows for the actual value of the target variable and columns for the output of the model. In each cell of the table is a count that tells us how often a test example from category *i* was thought by the model to be in category *j*. The confusion matrix for a good model will be dominated by counts along the diagonal. These diagonal elements correspond to cases where category *i* was correctly identified as category *i.*

Here is a confusion matrix in which each row corresponds to a possible value for the target variable.

```
=======================================================
Confusion Matrix
-------------------------------------------------------
a   b   c   d   e   f   <--Classified as
9   0   1   0   0   0    |  10      a     = one
0   9   0   0   1   0    |  10      b     = two
0   0   10  0   0   0    |  10      c     = three
0   0   1   8   1   0    |  10      d     = four
1   1   0   0   7   1    |  10      e     = five
0   0   0   0   1   9    |  10      f     = six
Default Category: one: 6
```

Each value of the target variable has a single row, labeled on the right. The possible values of the model output are arranged in columns, each labeled by a letter key that corresponds to one of the output values. The numbers show the number of examples for which the classifier produced a particular value for the target value for each possible actual value. The synthetic data used in this example are representative of an accurate classifier, as indicated by the fact that the numbers on the diagonal dominate the counts for each row.

Listing 15.2 shows how to use the `ConfusionMatrix` from the `org.apache.mahout.classifier` package to build a confusion matrix for output data that includes the true category and the model score output. This code makes two passes through the data, once to get a list of all possible values of the target variable and once to actually fill in the counts in the confusion matrix. It is common for the first pass to not be explicit because the list of symbols is known by some other method.

Listing 15.2 Building a confusion matrix

```
BufferedReader in = new BufferedReader(new FileReader(inputFile)); #A
List<String> symbols = new ArrayList<String>();
String line = in.readLine();
while (line != null) {
  String[] pieces = line.split(",");
  if (!symbols.contains(pieces[0])) {
    symbols.add(pieces[0]);
  }
  line = in.readLine();
}

ConfusionMatrix x2 = new ConfusionMatrix(symbols, "unknown");

in = new BufferedReader(new FileReader(inputFile)); #B
line = in.readLine();
while (line != null) {

  String[] pieces = line.split(",");         #C
  x2.addInstance(pieces[0], pieces[1]);
  line = in.readLine();
}
System.out.printf("%s\n\n", x2.summarize());
```

#A read the values and remember them
#B now count the pairs
#C input contains true category, model output

The confusion matrices produced by `ConfusionMatrix` depend on the outputs of a classifier being compared to each other or to a threshold in the binary case. Such a comparison forces the model to commit to a single answer. This approach loses information about how certain the model is of the answers it is giving because only the final answers count. When probability scores are available, we can avoid this loss by using a variation of a confusion matrix known as an entropy matrix.

In an entropy matrix, the rows correspond to actual target values, and the columns correspond to model outputs just as with a confusion matrix. The difference is that the elements of the matrix are the averages of the log of the probability score for each true/estimated category combination. Using the average log in this way has strong motivation in terms of mathematical principles based on approximating a probability distribution optimally. This average log probability expresses how unusual that particular classifier result is, comparing the classifier's likelihood of classifying into a certain category against how common that category is overall. A good model will have small negative numbers along the diagonal and will have large negative numbers in the off-diagonal positions. This is analogous to the way that a confusion matrix has large counts on the diagonal and small counts off the diagonal.

The entropy matrix has a useful connection with log-likelihood in that the weighted average of the diagonal entries of the entropy matrix is the log-likelihood. Log-likelihood is normally computed by taking the average of the log of the score for the target category. We discuss log-likelihood in more depth in the next section.

The `Auc` class can be used to compute both confusion matrices and entropy matrices, but currently only supports binary target variables. You can add data to the `Auc` object and print the confusion and entropy matrices like this:

```
Matrix entropy = x1.entropy();
for (int i = 0; i < entropy.rowSize(); i++) {
  for (int j = 0; j < entropy.columnSize(); j++) {
    System.out.printf("%10.2f ", entropy.get(i, j));
  }
  System.out.printf("\n");
```

```
}
```

Right now, the Auc class only works for binary target variables. Extending the Auc class to support more than just binary target variables is a feature that may appear in a future version of Mahout.

When we have a model that produces probability score outputs, the log of the score for the correct answer is a useful measure of how good the model is. If this value is averaged over a large number of held-out examples, it provides a useful single measure of model quality. This measure is known as the average log-likelihood. This term is often abbreviated as log-likelihood, at the risk of some confusion.

15.2.3 Computing average log-likelihood

Log-likelihood has the desirable mathematical property that when you can only maximize log-likelihood with an estimate of a probability distribution if you have exactly reproduced the actual probability distribution. Log-likelihood is a useful contrast with AUC. Log-likelihood can be used with multiple target categories, while AUC is limited to binary outputs. AUC, on the other hand, can accept any kind of score, while log-likelihood requires scores that reflect probability estimates. AUC has an absolute scale from 0 to 1 that can be used to compare different models while the scale for log-likelihood is open-ended.

The value of log-likelihood for a single training example is not quite useful because it varies so much from example to example, but the average over a number of examples is useful `OnlineSummarizer` can average the log-likelihood estimates from a classifier and summarize the estimates using median, mean and upper and lower quartiles like this:

```
OnlineSummarizer summarizeLogLikelihood = new OnlineSummarizer();
for (int i = 0; i < 1000; i++) {
  Example x = Example.readExample(); #1
  double ll = classifier.logLikelihood(x.target, x.features); #2
  summarizeLogLikelihood.add(ll);   #3
}

System.out.printf(
    "Average log-likelihood = %.2f (%.2f, %.2f) (25%%-ile,75%%-ile)\n",
    summarizeLogLikelihood.getMean(),
    summarizeLogLikelihood.getQuartile(1), #4
    summarizeLogLikelihood.getQuartile(2));
```

#1 get example
#2 compute log-likelihood
#3 add to summary
#4 compute and print statistics

In this code, examples are read #1 and a classifier computes the log likelihood #2 which is then passed to an `OnlineSummarizer` #3. After the input has been read, this summarize can be queried #4 to get descriptive statistics for the distribution of log-likelihood.

Log-likelihood has a maximum value of zero and no bound on how far negative it can go. For highly accurate classifiers, the value of average log-likelihood should be close to the average percent correct for the classifier times the number of target categories. For less accurate classifiers, especially those with many target categories, the average percent correct tends to go to zero, making it difficult to compare classifiers. The log-likelihood, on the other hand, can still distinguish better from worse classifiers even when the classifiers being compared are rarely or even never exactly correct which makes the average percent correct be zero.

For internal developer audiences, log-likelihood is probably a better measure for comparing models, but when presenting results to a less technically sophisticated audience, percent correct or even a confusion matrix is probably going to work better to convey your results.

Metrics like AUC and log-likelihood give us a picture of which models perform better or worse overall, but they cannot tell us why a model is working well or why it is not. In order to know that, it is necessary to look inside a model.

15.2.4 Dissecting a model

Dissecting a model is a way to see which features make a large difference in the output of the model. Mahout can do this for classes that extend `AbstractVectorClassifier` by using a `ModelDissector` from the `org.apache.mahout.classifier.sgd` package.

Remember that features used as predictor variables have to be tokenized and vectorized in order to be used by the learning algorithm. These feature vectors are among the clues that need to be examined to understand why a model is working. As input, a `ModelDissector` takes a feature vector, a trace of how the feature vector was constructed and a model. It then tweaks the feature vector and observes how the model output changes. By averaging over a number of examples, the effect of different features can be determined. When a summary is requested, the `ModelDissector` returns a list of those variables and values that exhibit the largest average effects.

This next code shows how a `ModelDissector` can be used to list the most important variables affecting the output of a model.

```
model.close();  #A

Map<String, Set<Integer>> traceDictionary =
    new HashMap<String, Set<Integer>>();
ModelDissector md = new ModelDissector(model.numCategories());

encoder.setTraceDictionary(traceDictionary);

for (... lots of examples ...) {
  traceDictionary.clear();
  Vector v = encodeFeatureVector(example, actual, leakType);
  md.update(v, traceDictionary, model);
}

List<ModelDissector.Weight> weights = md.summary(100);
for (ModelDissector.Weight w : weights) {
  System.out.printf("%s\t%.1f\n",
        w.getFeature(), w.getWeight());
}
```

#A forces model to finalize parameter weights

Here the loop that iterates over the examples is pseudo code, but the rest of the code is taken from actual working code. The key factor here is that the trace dictionary is used during encoding to record hints about how to dissect the model later. You don't have to push many of your training examples into the encoder to get lots of good hints. Generally if you have a few thousand or few tens of thousands of examples you should be pretty good to go.

The output of the `ModelDissector` is helpful in detecting what is going on in a misbehaving model. In a properly trained model, the most important variables and values should make sense to an expert in the problem domain. In a broken model, however, it is common for the most important variables to be clearly nonsensical. We will see examples of correctly working and misbehaving models in the next few sections.

15.2.5 Performance of the SGD classifier with 20 newsgroups

Chapter 14.4 showed you how to write a classifier using the SGD algorithm to categorize data from the 20 newsgroups data set. In this section we extend that example by focusing on evaluating the performance of this classification system. Here we dissect the entire learning program, complete with metrics, so you can see how the pieces work together. In following sections, we will inject typical bugs into this example so that you can see their effect on performance.

Listing 15.3 shows a program that trains an `AdaptiveLogisticRegression` model on 20 newsgroup data. It is similar to the example for training an SGD model in section 14.4.

Listing 15.3 A complete program to train a model with progress diagnostics

```
private static final Analyzer analyzer =
    new StandardAnalyzer(Version.LUCENE_30);
private static final FeatureVectorEncoder encoder =
    new StaticWordValueEncoder("body");
private static final FeatureVectorEncoder bias =
    new ConstantValueEncoder("Intercept");

public static void main(String[] args) throws IOException {
  File base = new File(args[0]);
  Dictionary newsGroups = new Dictionary();
```

```
    encoder.setProbes(2);

    AdaptiveLogisticRegression learningAlgorithm =   #1
        new AdaptiveLogisticRegression(20, FEATURES, new L1());
    learningAlgorithm.setInterval(800);              #2
    learningAlgorithm.setAveragingWindow(500);       #3

    List<File> files = Lists.newArrayList();
    for (File newsgroup : base.listFiles()) {
      newsGroups.intern(newsgroup.getName());
      files.addAll(Arrays.asList(newsgroup.listFiles()));
    }
    Collections.shuffle(files);                    #4

    int k = 0;
    for (File file : files.subList(0, 5000)) {
      String ng = file.getParentFile().getName();
      int actual = newsGroups.intern(ng);

      Vector v = encodeFeatureVector(file, leakType);   #5
      learningAlgorithm.train(actual, v);              #6
      k++;

      State<AdaptiveLogisticRegression.Wrapper> best =
              learningAlgorithm.getBest();

      if (best != null && k % 500) {
        CrossFoldLearner model = best.getPayload().getLearner(); #7
        double averageCorrect = model.percentCorrect();   #7
        double averageLL = model.logLikelihood();         #7
        System.out.printf("%d\t%.3f\t%.2f\t%s\n",
            k, averageLL, averageCorrect * 100, leakLabels[leakType % 3]);
      }
    }
    State<AdaptiveLogisticRegression.Wrapper> best =
              learningAlgorithm.getBest();
    dissect(best.getPayload().getLearner(), files); #8
  }
```

#1 Allocation and initialize learning algorithm
#2 short generation
#3 small window for averages
#4 randomize order of training
#5 encode features
#6 train model
#7 get performance of best model
#8 dissect final best model

This program starts with the allocation and configuration #1 of the learning algorithm. Here we chose an `AdaptiveLogisticRegression` for simplicity because it is not necessary to worry much about its learning parameters. You might also use a `CrossFoldLearner` if you don't mind tweaking learning parameters. The code configures the generation time #2 for the underlying evolutionary algorithm to be fairly short here because the dataset is relatively small and the algorithm will learn quickly. Similarly, we set the averaging window #3 for diagnostics to be relatively short so that we get quick indications of how the learning algorithm is doing.

In preparation for training, we need a list of all documents in the training set. We also want to order them randomly to help the algorithm learn more effectively. In order to train the model, we iterate #4 over the randomly ordered document list. Training involves conversion to vector form #5 and the actual training #6 of the model. Once the evolutionary algorithm inside the `AdaptiveLogisticRegression` gets going, it will provide the best-known model from its pool. We can use this model to get the log-likelihood and percent correct averaged over recent training examples #7.

When we complete training, we dissect the model #8. The details of this dissection are shown in Listing 15.4.

Listing 15.4 Model dissection for 20 newsgroup example

```
private static void dissect(CrossFoldLearner model, List<File> files)
    throws IOException {
  model.close();               #1

  ModelDissector md = new ModelDissector(model.numCategories());
  Map<String, Set<Integer>> traceDictionary =
           new HashMap<String, Set<Integer>>();   #2
  encoder.setTraceDictionary(traceDictionary);    #2
  bias.setTraceDictionary(traceDictionary);

  for (File file : files.subList(0, 500)) {
    traceDictionary.clear();         #3
    Vector v = encodeFeatureVector(file, leakType);
    md.update(v, traceDictionary, model);   #4
  }

  for (ModelDissector.Weight w : md.summary(100)) {      #5
    System.out.printf("%s\t%.1f\t%s\n", w.getFeature(), w.getWeight());
  }
}
```

#1 close model to finalize weights
#2 trace dictionary records feature hashing details
#3 clear dictionary occasionally to avoid memory clog
#4 pass dictionary to dissector
#5 dump dissection

In dissecting the model, we need to first make sure that the model's parameters have been finalized. Closing an on-line model does not prevent further training, but it does make sure that we have a consistent state to dissect. In dissection, we need to provide the feature encoding with a so-called trace dictionary. This records which predictor variables and values were assigned to which locations in the feature vector. This slows down the encoding by quite a bit so it isn't something to be done normally. By clearing the trace dictionary on each example, or every few examples, we can avoid letting the trace dictionary grow too large in memory. After encoding each example and recording the details of the feature extraction, we pass the details to the model dissector.

Once you have finished processing a reasonable number of training examples, you can ask the model dissector for a list of the most important variables and values. In this example, the top 100 such values are printed.

If you run this example, you will see the percent correct evolve something like what is shown in Figure 15.1.

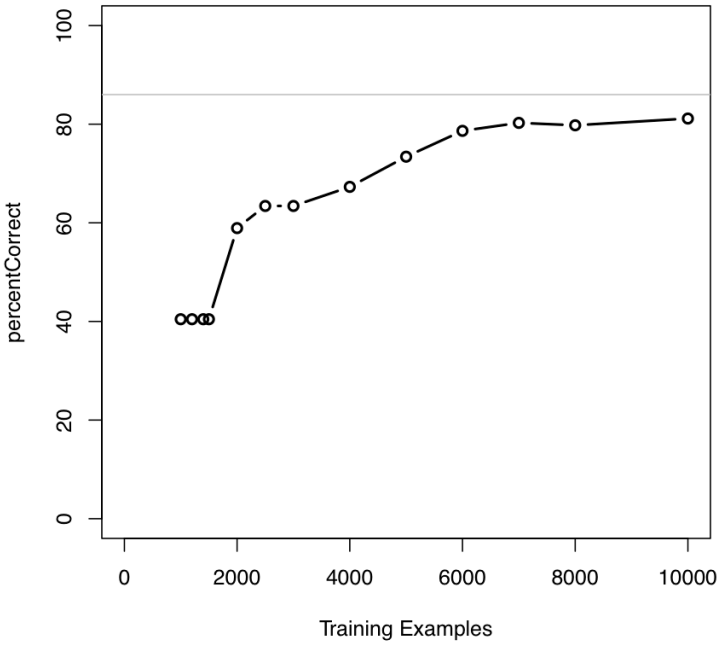


Figure 15.1 Increase in average percent correct with increasing number of training examples. The gray bar shows a reasonable maximum performance that we can expect based on the best results reported in the research literature.

If we used multiple passes through the data depicted in figure 15.1, the actual performance would likely approach the state of the art performance in the mid 80% range. Multiple passes do not improve discrepancies as much as simply training on additional data because repeating the same data does not give as much diversity in training data. Happily, too small a sample size is almost never the issue with Mahout-based classification because Mahout's claim to fame is large data. Multiple passes through the data do still improve performance of the model because the system is learning with every pass, up to a point.

Here is the output of the model dissection for a typical run of this program. The columns are the name of the feature (a single word of text in this case), the largest weight of that feature and the target variable value for that largest weight.

```
body=space  2.1    sci.space
body=sale   1.9    misc.forsale
body=car    1.9    rec.autos
body=windows1.8    comp.os.ms-windows.misc
body=mac    1.7    comp.sys.mac.hardware
body=bike   1.7    rec.motorcycles
body=apple  1.5    comp.sys.mac.hardware
body=gun    1.5    talk.politics.guns
body=baseball      1.5    rec.sport.baseball
body=graphics      1.5    comp.graphics
body=key    1.5    sci.crypt
body=x      1.4    comp.windows.x
body=dod    1.4    rec.motorcycles
body=orbit  1.4    sci.space
body=clipper1.4    sci.crypt
body=encryption    1.3    sci.crypt
body=window 1.3    comp.windows.x
body=image  1.2    comp.graphics
```

```
body=hockey 1.2     rec.sport.hockey
body=atheists       1.2     alt.atheism
```

These features are clearly pretty reasonable with the possible exception of the word “dod”. This is apparently a feature for the `rec.motorcycles` Usenet discussion group. Examination of the training texts, interestingly, shows that there was a lengthy discussion back and forth using “dod” as a nonsense word. While this feature is probably not going to help future performance after the presumably short-lived use of that word, it also is not likely to hurt.

At this point, you know how to measure how good your classifier is. That leaves you with the problem of trying to decide what those measurements mean. Let’s look at how typical problems show up in the performance metrics and how you diagnose and fix these problems.

15.3 When classifiers go bad

When working with real data and real classifiers it is almost a rule that the first attempts to build models will fail, occasionally spectacularly. Unlike normal software engineering, the failures of models are not usually as dramatic as a null pointer dereference or out-of-memory exception. Instead, a failing model can appear to produce miraculously accurate results. Such a model can also produce results so wrong that it seems the model is trying to be incorrect. It is important to be somewhat dubious of extremely good or bad results, especially if they occur early in a model’s development.

This section will help you understand how the metrics you can collect using techniques from the previous section tell you about these predictable bumps in the life of a classifier. To do that, you need to see what the common problems with classifiers are and how those problems show up in the performance metrics.

The two most common problems in building classification models are so-called target leaks and broken feature extraction. A target leak occurs when some feature in the training data provides information about the target variable in some way that will not happen in production. This is the classifier equivalent of handing out the answers to an exam along with the exam itself. Not surprisingly, a classifier can do quite well in such a case. This can happen in subtle ways that are difficult to diagnose. Bad feature extraction can also be difficult to detect unless it is catastrophically bad. We will see examples of both problems in this next section.

15.3.1 Target leaks

A target leak is a bug that results from information from the target variable being included in the predictor variables used to train a classifier. The major symptom of a target leak is performance that is just too good to be true.

To help you understand how to avoid a target leak, we have intentionally inserted one in the 20 news groups sample classifier from listing 15.3. Here is the small addition to the program that injects a target leak into our training examples

```
private static final SimpleDateFormat("MMM-yyyy");
// 1997-01-15 00:01:00 GMT
private static final long DATE_REFERENCE = 853286460;

...
  long date = (long) (1000 *
      (DATE_REFERENCE + target * MONTH + 1 * WEEK * rand.nextDouble())); #A
  Reader dateString = new StringReader(df.format(new Date(date)));
  countWords(analyzer, words, dateString);
```

#A Target variable becomes date

Unlike before, this code injects a simulated target leak into the data in the form of a date field. This date field is chosen so that all the documents from the same newsgroup appear to have come from a same month, but documents from different newsgroups come from different months. Dates, times and id numbers are common sources of target leaks, but we could have used any sort of field to inject the leak. It can even happen that the target variable itself is accidentally included as a predictor. As you might expect, such an error creates the mother of all target leaks.

When we train the model using this target leak in addition to the text, the accuracy of the model gets entirely too good. Figure 15.2 shows what happens. Especially in the case where the target leak appears

with just the subject line, accuracy approaches 100%. This is unrealistically high because the best reported result for this data set in the research literature is about 86%. In the case where the leak is added to all of the text from the subject line and the body of the message, it takes longer for the learning algorithm to sort out what is happening, but before long it still reaches implausibly good performance.

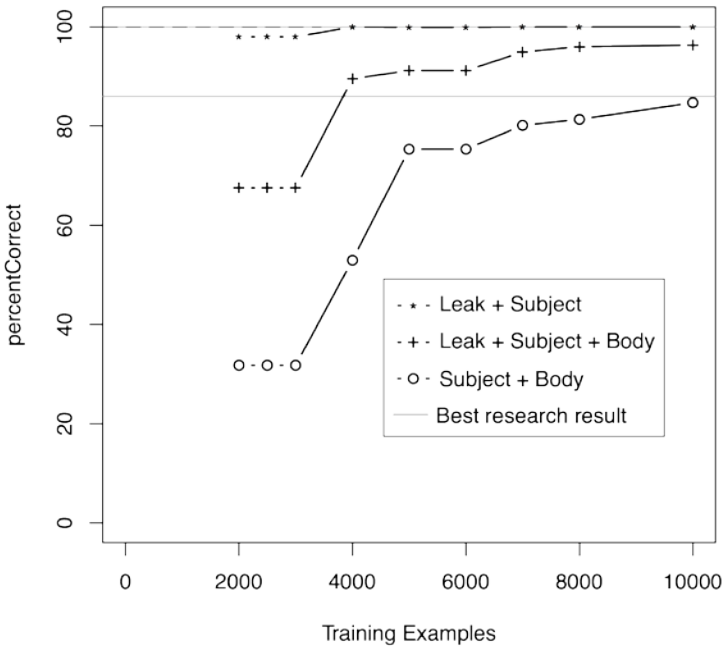


Figure 15.2 A target leak produces implausible accuracy but not always instantly.

From the results shown in figure 15.2, you can see that the two lines representing conditions with the target leak quickly approach 100% accuracy. Such an extreme level of accuracy is not plausible for this problem. The SGD model without the leak, on the other hand, only just reaches the level represented by the best academic research results for this problem. That level of performance is very good, but still plausible.

When working with real data rather than carefully studied research data, it is not necessarily possible to known how good "too good" is from the start. If the model does appear to be doing exceptionally well, or if the performance on new data is dramatically worse than predicted by cross validation, then it is worth looking at the output of the `ModelDissector`.

Here are the first few lines of the model dissection output for the text plus leak case. The columns in the output are the term, the weight and the newsgroup that the weight applies to.

```
body=feb-1998       3.7     sci.electronics
body=dec-1997       3.6     comp.graphics
body=sep-1997       3.5     sci.med
body=mar-1998       3.3     comp.sys.mac.hardware
body=jul-1997       3.3     talk.religion.misc
body=jun-1997       3.2     misc.forsale
body=apr-1998       3.2     rec.motorcycles
body=jul-1998       3.1     talk.politics.misc
body=jun-1998       3.0     comp.os.ms-windows.misc
body=mar-1997       2.9     sci.space
body=apr-1997       2.9     talk.politics.guns
body=jan-1998       2.8     soc.religion.christian
body=nov-1997       2.8     comp.sys.ibm.pc.hardware
```

```
body=may-1997       2.8     comp.windows.x
body=feb-1997       2.7     rec.sport.baseball
body=aug-1998       2.6     alt.atheism
body=may-1998       2.6     rec.autos
body=oct-1997       2.5     rec.sport.hockey
body=gun    2.1     talk.politics.guns
body=aug-1997       2.1     sci.crypt
body=windows2.0     comp.os.ms-windows.misc
body=consciously    1.8     sci.electronics
body=taber  1.8     sci.electronics
body=uw.pc.general  1.8     sci.electronics
body=hearing1.8     sci.med
body=car    1.7     rec.autos
body=sdennis@osf.org        1.7     misc.forsale
body=1024x768x24    1.7     comp.graphics
body=contradicts    1.7     talk.religion.misc
body=market 1.7     comp.graphics
body=passes 1.7     misc.forsale
body=att-14,796     1.7     talk.religion.misc
body=bars   1.7     sci.med
```

These most heavily features are dominated by dates. Other than large-scale seasonal changes like Christmas, having the date be such a strong predictor of topic is not reasonable. Having nearly all of the top 20 features be particular dates is entirely too much to believe so the only reasonable conclusion is that these features represent a target leak. Of course, in this case, we injected the target leak so nobody should be surprised to find one.

Another subtler indicator that something is wrong comes from looking the top non-leak features. The `AdaptiveLogisticRegression` adjusts the learning rate in order to optimize performance. In the case of a target leak, it is to the advantage of the algorithm to decrease the training rate fairly quickly once it has found the leak features. As such, it can freeze other extraneous features without their weights decaying to zero, as a normal model would have to do to get good performance. Thus, we see words like "taber," "market," "passes" and "bars" listed as highly weighted features, predictive of `sci.electronics`, `comp.graphics`, `misc.forsale` and `sci.med` respectively. That such features persist in the final model is an indication that learning has frozen without proper pruning.

15.3.2 Broken feature extraction

Another common problem occurs when the feature extraction is somehow broken. In contrast to target leaks, where performance is unreasonably good, broken feature extraction causes much worse performance than expected. This code below shows how a Lucene tokenizer might be used incorrectly for the tokenization of the 20 newsgroup data.

```
TokenStream ts = analyzer.tokenStream("text", in);
ts.addAttribute(TermAttribute.class);
while (ts.incrementToken()) {
  String s;
  if (insertTokenError) {
    s = ts.toString();                                  #A
  } else {
    s = ts.getAttribute(TermAttribute.class).term();  #B
  }
  words.add(s);
}
```

#A BAD! don't do this
#B Good! This is the correct way to extract a term

This error causes additional information to be retained in addition to the desired token, and this additional information makes the same word in different locations to appear as if it were two different words. That makes learning from the affected predictor hard, if not impossible.

Figure 15.3 shows how what performance looks like with the tokenization error in place. The percent correct never gets nearly above 5%, which is, of course, the result the model would get by randomly selecting a result.

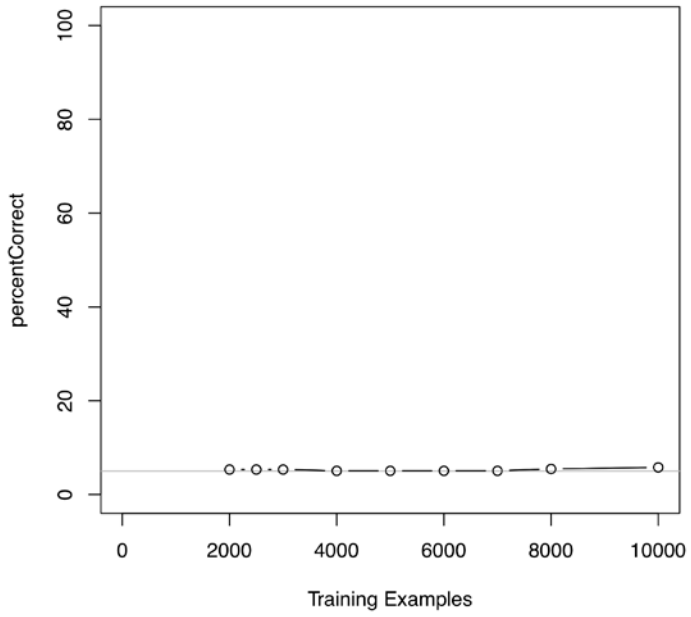


Figure 15.3 Performance with the tokenization bug never increases of the level expected from a random selection of the target variable. The gray line shows performance that would be expected by chance.

This example is extreme because all of the input data has been corrupted by bad tokenization and nothing is left that the model can use to produce good results. In practice, it is more common that only some fields are corrupted while others are fine. Such a partial corruption will lead to degraded performance if the corrupted variable would actually have helped the model, but will not necessarily lead to the flat-line performance that we see in this example since the uncorrupted variables are still available to carry some of the load.

The best way to find these kinds of problems is to compute summary statistics on the values being encoded. If the value is continuous, use an `OnlineSummarizer` to examine the mean, minimum, maximum and quartiles. These values should seem reasonable. For categorical, word-like and text-like data, you should produce counts of the tokens produced and plot these counts versus rank on a log-log scale. For the 20 newsgroup data, this gives a graph like the one shown in Figure 15.4. For word-like and text-like data, something similar should emerge for each field of your data.

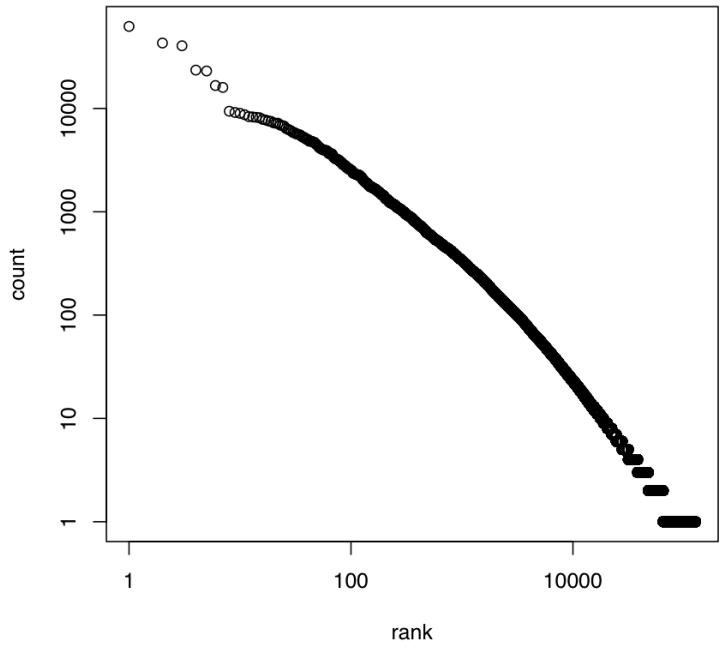


Figure 15.4. Plot of token count versus rank for correctly operating tokenizer on 20 newsgroup data. With the broken tokenizer, the counts would all have a value of 1.

In the case of our injected tokenization error, the frequencies of essentially all terms would be exactly one.

Table 15.4 shows some of the incorrect tokens that are produced if `insertTokenError` is set along with the correct values that should have been produced. Keep in mind that the parentheses, commas and numbers are all part of the incorrect token.

Table 15.4. How bad tokens can defeat feature extraction.

Erroneous Token	Correct Token
(term=elastic,startOffset=1705,endOffset=1712,...)	elastic
(term=elastic,startOffset=2064,endOffset=2071,...)	elastic
(term=failed,startOffset=1236,endOffset=1242,...)	failed
(term=failed,startOffset=245,endOffset=251,...)	failed
(term=failed,startOffset=974,endOffset=980,...)	failed

Note how repeated instances of the same token in the table appear different if too much information is inadvertently retained. This situation makes these erroneous tokens useless for categorization since the chances that they will appear in another document at exactly the same place is nearly nil.

Now that you know how some pitfalls such as target leaks and broken feature extraction can defeat a classifier, let's turn to a more optimistic topic and see how you can use evaluation to adjust and tune a classifier to improve performance.

15.4 Tuning for better performance

Beyond these steps of correcting errors in training data, there is much that can be done to improve classifier performance. Some approaches involve making the classifier's life easier by changing the problem itself. Others involve making the classifier run better on the data given. In the first case, it is possible to find new predictor variables, combinations of predictors, or transformations of predictors to help the classifier out. It is also possible to eliminate predictors that are not helping, or simplify the target variable. In the second case, learning rates can be tuned, or feature encoding strategies adjusted in the learning algorithm.

In any case, when tuning a classifier system, we need to make careful and realistic measurements of performance with and without the changes. It is not usually sufficient to examine a few ad hoc examples unless your change has made a catastrophic difference. In fact, once the system is working reasonably, most adjustments should improve or degrade performance by only small amounts, which can only be detected with accurate measurement.

15.4.1 Tuning the problem

To most people, the idea of tuning up a classification system brings to mind the idea of tuning the learning algorithm so that accuracy is highest. This assumption leads to trouble, however, because the big improvements usually result from tuning the problem instead. This can include changing the input, the output or even the interpretation of the output.

REMOVE "FLUFF" VARIABLES

Some variables don't actually help the classifier much or even at all and it may be better to simply withhold these variables from the classifier. Paradoxically, removing some information like this can improve accuracy because the learning algorithm may have less "distraction" with fewer variables and thus may learn a better classifier in the available training time. The target leak problem shown in section 15.4 was a great example of this phenomenon. In that example, the learning algorithm took considerably more time to learn the trivial target leak model when it had lots of text around to distract it. Conversely, you could say that the leak variable itself distracted the learning algorithm preventing it from noticing the really helpful variables in the subject or body.

Regularization is a more general version of this idea. Regularization is a mathematical technique that is used to avoid over-fitting by decreasing the effect of some variables or even eliminating them entirely. Regularization operates by having the learning algorithm learn to minimize error and model complexity at the same time. The mathematical function that penalizes complexity is called the "prior" in Mahout terminology.

Algorithms like SGD with regularization try to eliminate variables where possible by penalizing the inclusion of variables in the model. All else equal, these algorithms would prefer to not include a variable. Even so, it can take time for these algorithms to understand which variables can be eliminated. If dissection of a model shows that some variables are never mentioned in the model, it is probably useful to try eliminating them so that subsequent training goes faster. Likewise, it is useful to test versions of the model that are not allowed to use certain categories of variables to see what happens.

Other Mahout classifier algorithms, like Naïve Bayes and Complementary Naïve Bayes, can benefit greatly from the elimination of variables that do not carry useful information because these algorithms have no mechanism to eliminate these variables on their own via regularization.

It is relatively common to find that removing one of two variables makes no difference, but removing both seriously hurts performance. It can also happen that using a particular variable alone gives no better performance than chance decisions. It would be tempting to eliminate that variable if you noticed this. Occasionally, however, such a variable would be able to improve the performance of other variables even though it shows no predictive qualities in isolation. It is safe to say that any variable that does not appear in the dissection of the final model can be eliminated with positive effect, but most other conclusions require careful testing.

ADD NEW VARIABLES, INTERACTIONS AND DERIVED VALUES

In building a classification system, some data are easier to wrangle into usable form than other data. Difficulties in data preparation will usually lead to delays in being able to present all of the desired predictor variables to the learning algorithm. Conversely, some variables will probably be available sooner than others. In a large project, the delays in getting certain predictors can easily extend to weeks or months, and some predictors may well turn out impossible to get. The potential for these delays means that modeling and evaluation should be started early, with whatever data is available. It often happens that the initially available predictor variables are sufficient to build a deployable model. Additional variables should be tested as they become available since they may improve performance even if the early models are usable in production.

Another common source of new variables is the transformation of existing continuous variables. It can pay to insert additional versions that are the logarithm or the square of the original value. Generalized linear models such as the Mahout SGD-based models can benefit from this technique, since they otherwise notice only linear relationships between predictors and categories. Most tree-based models will not notice the difference.

Here is an example where a fictitious `TrainingExample` object is queried for a line count that is then encoded three different ways:

```
FeatureValueEncoder lineEncoder = new ConstantValueEncoder("lines");
FeatureValueEncoder logLineEncoder =
    new ConstantValueEncoder("log(lines)"); #A
FeatureValueEncoder lineSquaredEncoder =
    new ConstantValueEncoder("lines^2");    #B

for (TrainingExample example : trainingData) {
  Vector v = new RandomAccessSparseVector(50000);

  double lines = example.getLines();
  lineEncoder.addToVector((byte[]) null, lines, v);  #1
  logLineEncoder.addToVector((byte[]) null, Math.log(lines), v); #1
  lineSquaredEncoder.addToVector((byte[]) null, lines * lines, v);  #1
}
```

#A Note name change
#B ConstantValueEncoder to avoid parsing
#1 Insert three transformations

By encoding the line count #1 transformed multiple ways, we have provided the model with the ability to make certain kinds of inferences much more easily. For example, including logs of original and sale prices lets the model have access to the equivalent of a discount percentage. Giving it only the original and sale prices untransformed allows it to only see discounts in dollar terms rather than as a fraction of the original price. Which view is better is hard to say in advance so letting the learning algorithm choose can be a very good idea.

It does not make sense to add multiple versions of a variable for a Naïve Bayes model because Naïve Bayes only accepts categorical or text-like inputs. These inputs are encoded as a number of binary values, and transformations of binary variables rarely make any useful sense.

Interaction variables are another source of power for a classifier. Interaction variables are combinations of other variables. For instance, in a marketing application, it might be good to have an interaction variable that combines gender with brand name. Without this interaction variable, the model can deduce that women are more or less likely to respond to a marketing offer than men or that a particular brand is more or less popular. With the interaction of gender and brand, however, the model can say that a particular brand is more or less popular to men or more or less popular with women. Since different manufacturers specialize the brands they sell to appeal to different market segments, it is no surprise that such a variable can be very useful. With hashed feature representations, interaction variables can be added using code similar to the following

```
LuceneTextValueEncoder authorEnc = new LuceneTextValueEncoder("author");
LuceneTextValueEncoder subjectEnc = new LuceneTextValueEncoder("subject");
FeatureVectorEncoder authorPlusSubjectEnc =   #1
      new InteractionValueEncoder("author_subject", author, subject);

for (TrainingExample example : trainingData) {
```

```
  Vector v = new RandomAccessSparseVector(50000);
  String author = example.getAuthor();
  String subject = example.getSubject();
  authorEnc.addToVector(author, 1, v);         #2
  subjectEnc.addToVector(subject, 1, v);       #3
  authorPlusSubjectEnc.addToVector(author, subject, 1, v);  #4
  ...
}
#1 Create composite encoder
#2 encode author
#3 and subject
#4 and the interaction
```

Here, the interaction encoder `authorPlusSubjectEnc` is created using a reference to the author encoder `authorEnc` and the subject text encoder `subjectEnc`. The string containing the author name and subject line are then encoded as an interaction variable. The variable represents not just the author and subject, but the fact of their association, distinct from that subject as used by another author. This allows the classifier to draw different distinctions based on both who and what was said. If different authors use the same text to mean different things, the model would have no chance of learning the correct meaning without this interaction variable. But with it, the model can ascribe different meanings to exactly the same words depending on the context in which they appear. In addition, because of the inclusion of the separate encoders for author and subject, the model can also learn subject line meanings that do not depend on who authored the subject line.

Interactions are powerful, but they can dramatically increase the complexity of a classifier and make feature encoding and learning dramatically slower. Since the number of features for each training example is dramatically increased, this can also cause accuracy to degrade.

Another way to introduce transformations of existing variables is to cluster a portion of training data using something like k-means clustering applied to several of the predictor variables. This clustering leads to a new predictor: for each datum, its cluster is a categorical feature.

Similarly, it is possible to build a classifier model and then use the output of that classifier as a predictor variable for another model. The idea is that the first model will learn the general outline of the problem whereas the cascaded second model can learn how to correct errors that the first model makes. Here is a bit of code that illustrates this

```
AbstractVectorClassifier subModel =
    ModelSerializer.loadJsonFrom(new FileReader("sub-model.model"), #1
    OnlineLogisticRegression.class);
ConstantValueEncoder subModelEnc = new ConstantValueEncoder("sub");
...
for (TrainingExample example : trainingData) {
  Vector v1 = new RandomAccessSparseVector(50000); #2
  ... encode features for subModel ...
  Vector v2 = new RandomAccessSparseVector(50000);
  double subScore = subModel.classifyScalar(v1);   #3
  subModelEnc((byte[]) null, subScore, v2); #4
  ... encode other features ...
  ... train model ...
}
```

#1 read previously trained sub-model
#2 v1 is for sub-model.model
#3 compute output of sub-model
#4 encode sub-model feature

Here we assume that there is a first model already trained and waiting for us to load from the file "sub-model.model" #1. Then when training the second model, we build a feature vector to be used as input for the first model #2 and then use the first model to compute a score #3 that is then used as a feature #4 for the second model.

15.4.2 Tuning the classifier

In addition to changing the data that are presented to the classifier learning algorithm, it is possible to change the algorithm or to modify the parameters that define the algorithm. The benefit of trying these approaches is that it provides you with a variety of ways to improve performance through guided trial and error.

TRY ALTERNATIVE ALGORITHMS

It is important to test different classifier algorithms to see which algorithm does best. Mahout supports Naïve Bayes, Complementary Naïve Bayes and SGD. There are also experimental implementations of Random Forests and Support Vector Machine (SVM) available.

In addition to changing the entire class of model, it is possible to change the learning algorithm while keeping the same kind of model. In particular, SGD-based classifiers can operate the low-level learning algorithm with cross validation using `CrossFoldLearner`, or the self-adapting `AdaptiveLogisticRegression`, which adjusts the many learning parameters automatically. There is also the choice of just what the learning algorithm is optimizing, choosing from log-likelihood, AUC or mixtures of these.

Compare different algorithms based on performance, training time and classification speed. For example, try both versions of the Bayes algorithms as well as the `AdaptiveLogisticRegression`. If more than one model does reasonably well on accuracy metrics, then decide between them based on how well they integrate with the rest of the system.

TUNE THE LEARNING ALGORITHM

Particularly with the low-level SGD learning algorithms, there are many knobs that can be turned to get different results. Most of these parameters are adjusted when constructing an `AdaptiveLogisticRegression`, `CrossFoldLearner` or `OnlineLogisticRegression` using similar configuration methods. These methods are summarized in Table 15.5.

Table 15.5 Summary of configuration for SGD learning classes

Algorithm Parameter	How Configured	Online Logistic Regression	Cross Fold Learner	Adaptive Logistic Regression	Description
prior	constructor	yes	yes	yes	Regularization function to encourage sparsity and avoid over-fitting
numFeatures	constructor	yes	yes	yes	The size of the feature vector
numCategories	constructor	yes	yes	yes	How many target categories there are
alpha	setter	yes	yes		Learning rate schedule
decayExponent	setter	yes	yes		Learning rate schedule
lambda	setter	yes	yes		Controls amount of regularization
learningRate	setter	yes	yes		Learning rate schedule
stepOffset	setter	yes	yes		Learning rate schedule

The prior, number of features and number of target categories are configured in all SGD classes in the constructor. The `AdaptiveLogisticRegression` class controls the learning rate by running a number of `CrossFoldLearners` in parallel with different settings and comparing the results obtained with each of them. This means that with the `AdaptiveLogisticRegression`, the only other configuration settings needed are used to control how often the learning rate is changed and how many different parallel `CrossFoldLearners` are used to test different settings. For the `CrossFoldLearner` and `OnlineLogisticRegression`, needs to be given learning rate and also how the learning rate will decay over time. This is done using builder-style methods such as those shown here:

```
lr = new CrossFoldLearner(targetCategories, numFeatures, new L1())
          .lambda(currentLambda)
          .learningRate(initialLearningRate)
          .decayExponent(0.5)
          .stepOffset(2000);
```

Usually, the `decayExponent` and `stepOffset` are used to control how the initial learning rate is decreased over time while setting alpha is much less common.

In addition to these settings, the `CrossFoldLearner` allows setting the number cross-validation folds it uses internally. The default value of 5 is acceptable in almost all settings. Using a lower value such as 2 or 3 decreases the amount of data available for training more than most applications are willing to accept, and setting the value much higher than 5 incurs higher computation costs, since it has to have an independent learner for each fold.

The simplest approach to tuning all of these parameters is to just use the `AdaptiveLogisticRegression` with default settings. This costs quite a bit in terms of compute power since the `AdaptiveLogisticRegression` uses an internal evolutionary pool of 20 `CrossFoldLearners`, each of which internally maintains 5 `OnlineLogisticRegression` objects.

This means we pay the cost of running 100 learning algorithms at the same time. In practice, the penalty is much smaller because reading training examples from disk and encoding them as feature vectors often costs nearly as much as running these 100 learning algorithms. Moreover, in practical situations, a learning algorithm will be run with at least dozens and possibly hundreds or thousands of settings for these parameters. The `AdaptiveLogisticRegression` does all that automatically. As a side effect, data must only be encoded once. Bad options are cut-off early so no computational power is wasted on them.

15.5 Summary

At this point we have seen the major APIs in Mahout for computing classification quality metrics and have seen the basics of how to use these metrics. We have seen how common bugs affect these metrics and how to find and correct common problems with classification.

As the examples in this chapter show, problems with classifiers can be subtle. Unlike in most computer programming, even a serious error may not cause complete failure because learning algorithms often learn how to compensate for partial failures. This is both good and bad news, and makes detailed quality testing important.

Evaluation is not only an iterative process during the steps of building and tuning a high-performing classifier, it also is an important ongoing step after production launches. Evaluation of performance over time is critical to maintaining performance. External conditions may change making the model less useful or operational errors in data collection can degrade a model's effectiveness. In any case, ongoing evaluation during production is key to being able to recognize the need to re-tune or re-train a model or opt for a new approach. These issues are explored in the discussion of deployment in the next chapter.

16
Deploying a Classifier

This chapter covers

- Specifying the speed and size requirements of a classifier
- Building a large-scale classifier
- Delivering a high speed classifier
- Building and deploying a working classification server

Chapter 16 examines the issues you face in putting a classification system into real-world production. In previous chapters, we talked about how Mahout classifiers are trained, how they work and how they are evaluated, but those discussions purposely omitted many of the ways that the real world intrudes when they are put into production. In this chapter, we explain how to deploy a high-speed classifier in real-world situations.

In production, some or all of the following are true: you have a huge amount of training data, you have high throughput rates, you need fast responses, or you need integration into a large pre-existing system. Each of these characteristics implies specific architectural and design considerations that you need to take into account as you build your system. Large data require efficient ingestion and organization; high throughput requires scalable and optimized code. Integration into large systems requires a simple and clean API that plays well as a networked service. Mahout offers a powerful way to deal with big-data systems, but keep in mind that there are ways to simplify or restate problems to make tools like Mahout even more powerful.

In the next section, we give an overview of how to design, build and deploy a real-world large-scale classification.

16.1 Process for deployment in huge systems

There are a few standard steps that will help ensure success in deploying a Mahout-based classifier system. The detailed strategies for coping with each of these issues are explained in sections 16.2 through 16.4 of this chapter; this section is an overview of the main ideas as an introduction to process of deploying a huge system.

> Note: To understand the potential value of Mahout for your classification project, look for aspects of your system that are especially large or where you need it to be especially fast. Finding these “pain points” will tell you whether or not Mahout is the right solution.

16.1.1 Scope out the problem

The first step is to determine how large a problem you actually have and how fast it has to run in both training and production classification. The size of your training data will dictate whether your feature extraction will allow use of conventional tools like a relational database or whether it will need a highly

scalable feature extraction system using tools like Apache Hadoop. The specifics on size and speed requirements are explored in section 16.2 of the chapter.

The allowable time for training will determine whether you can use a sequentially-trained classifier such as SGD at all, for example. The throughput and latency requirements for the production classifier will determine whether you need a distributed classifier and whether you need to do extensive pre-computation and currying to maximize classification speed. For utmost throughput at some cost in latency, you may also decide to do batch classification.

16.1.2 Optimize feature extraction as needed

As data sizes get to be more than tens of millions of examples, it becomes more and more important to use scalable techniques such as MapReduce for feature extraction. At small to medium scale, any technique will work and proven data management and extraction technologies like relational databases will serve well.

As scale increases, you are likely to have to commit to a parallel extraction architecture such as Hadoop or a commercially supported system like AsterData, Vertica or Greenplum. If you choose to use Hadoop, you will also have to decide what general approach you will use to implement your feature extraction code. Good options include Apache Hive, Apache Pig, Apache Cascading or even raw MapReduce programs written in Java.

A key point here is that any downsampling should be done as part of feature extraction rather than vector encoding so that it can be done in parallel. Issues in real-world feature extraction for big data systems are examined in detail in section 16.3.

16.1.3 Optimize vector encoding as needed

If you have less than about 100 million training examples per classifier after feature extraction and downsampling, consider a sequential model builder such as SGD. If you have more than about 5 million training examples, you will also need to optimize your data parsing and feature vector encoding code fairly carefully. This requirement may involve parsing without string conversions, multi-threaded input conversion and caching strategies to maximize the speed that data can be fed to the learning algorithm.

For more than about 10 million training examples, you may want to consider using a `CrossFoldLearner` directly with predefined learning rate parameters. If the problem is relatively stable, you may be able to use an adaptive technique to learn good learning rate schedules and then just stick with what works well. The advantage over using an `AdaptiveLogisticRegression` is that 20 times fewer numerical operations are required in learning. The disadvantages are that the speedup is not nearly as large as it sounds and your learning chain will require more maintenance.

16.1.4 Deploy a scalable classifier service

In most applications, it is convenient to separate classifiers into a distinct service that handles requests to classify data. These services are typically conventional services based on a networked RPC protocol such as Apache Thrift or REST-based HTTP. An example of how to do build a Thrift-based classification server is in section 16.4. This is particularly appropriate where many classification operations are done on each input example such as where many items are tested for expected value.

In a few cases, the network round-trip is an unacceptable overhead and the classifier needs to be integrated as a directly called library. This happens when very small grain requests must be done at extreme rates. Since it is possible to achieve throughputs of over 50,000 requests per second with modern network service architectures, such cases are relatively rare.

16.2 Determining scale and speed requirements

The first step in building a classifier is to characterize the scale factors including the amount of training data and throughput rates that your system is required to meet. Knowing the size of the problem in various dimensions will in turn lead to several design decisions that will characterize how your system will need to be built.

16.2.1 How big is big?

If you are considering using Mahout to build a classification system, ask yourself these questions about your system and use these answers to determine which parts of your system will require big-data techniques and which can be handled conventionally:

- How many training examples are there?
- What is the batch size for classification?
- What is the required response time for classification batches?
- How many classifications per second total need to be done?
- What is the expected peak load for the system?

> Sidebar To estimate peak load, it is convenient to simply assume that a day has 20,000 seconds instead of the actual 86,400 seconds that make up a single day. Using an abbreviated day of 20,000 seconds is a rule-of-thumb that allows you to convert from average rates to peak rates at plausible levels of transaction burstiness. Estimating long-term average transaction rates is usually pretty easy, but you will need to know the peak rate to be supported when you design your system. Using this rule of thumb allows you to make a so-called "Fermi estimate" for the required number of transactions per second that your system will need to handle. Enrico Fermi was a renowned physicist who was remarkably good at making estimates like this, and his name has become attached to such back-of-the-envelope computations.

These values need not be estimated exactly. Typically numbers that are within an order of magnitude are good enough for initial system scaling and architecture. As you move into production, you will be able to refine the estimates for final tuning. Once you have an idea of the number of training examples you expect to use, you can consider whether or not Mahout is appropriate for your project and if so, which algorithms are likely to be a good choice.

IMPLICATIONS OF THE NUMBER OF TRAINING EXAMPLES

Typically, Mahout becomes interesting once you exceed 100,000 training examples. There is, however, no lower bound on how much data can be used. At 10 million training examples, most other data-mining solutions are unable to complete training while above 100 million or a billion training examples there are very few systems available that can build usable models. The number of training, along with model type, mostly determines the training time.

With a scalable system such as Mahout, you can extrapolate fairly reliably from smaller tests to determine expected training time. With sequential learning algorithms such as SGD, learning time should scale linearly with input size. In fact, many models converge to acceptable error levels long before all available input has been processed so training time grows sub-linearly with input size.

With parallel learning algorithms such as Naïve Bayes, learning time scales roughly linearly in input size divided by the number of map-reduce nodes available for learning. Since Naïve Bayes learning is coded as a batch system where all input is processed through multiple phases, it cannot be stopped early. This means that even though a partial data set might be sufficient to produce a useful model, training must process all training examples before a usable model is available.

CLASSIFICATION BATCH SIZE

It is common for a number of classifications to be done to satisfy a single request. This requirement often occurs because multiple alternatives need to be evaluated. For instance, in an ad placement system, there are typically thousands of advertisements that each have models that need to be evaluated to determine which advertisement to show on a web page. The batch size in this case would be the number of advertisements that need to be evaluated. A few applications have very large batches of more than 100,000 models.

At the other extreme, a fraud detection server might only have a single model whose value needs to be computed because the question of whether a single transaction is fraudulent involves only one decision: fraud or non-fraud. This situation is very different from an advertisement selection server where a click prediction model must be computed for each of many possible advertisements.

MAXIMUM RESPONSE TIME AND REQUIRED THROUGHPUT

The maximum response time determines how fast a single batch of classifications must be evaluated. This, in turn, defines a lower bound on the required classification rate. This rate is only a lower bound because if there are enough transactions per second, it may be necessary to support an even higher rate.

Typically, it is pretty easy to build a system that supports 1,000 classifications per second with small classification batch sizes because such a rate is small relative to what current computers can do. In fact, the Thrift-based server described later in this chapter can easily meet this requirement. Until batch sizes exceed 100 to 1,000 classifications per batch, you should be able to support roughly that transaction rate because costs will be dominated by network time. At batch sizes of 10,000 or more, response times will start to climb substantially. At larger batch sizes and with sub-second required response times, you will probably need to start scaling horizontally or restructuring how the classifications are done.

16.2.2 Balancing big versus fast

One of the advantages of using Mahout is that the trade-offs between large data and fast operation are much looser than in non-scalable systems. That said, there is almost always a trade-off to some extent between these characteristics of a system, so you will need to consider how to balance these needs. To do this, consider that one aspect of your system may have different requirements than another.

Once you have the overall characteristics of your system in mind, you can begin to outline those aspects of your deployment that will require special attention and which can be implemented more quickly. The following sections outline the most important considerations in each phase of the deployment. These considerations generally fall into those considerations that apply when training is done or after deployment during classification.

AT TRAINING TIME

Joining data into classifiable data often dominates all other aspects of training. You can use standard Hadoop tools for that, but care needs to be taken to make the joins run efficiently.

Downsampling negative cases can help enormously by making your training data much, much smaller. In fraud or click-through optimization, the number of uninteresting training examples typically vastly outnumbers the number of interesting ones (the frauds or clicks). It is critical to accurate learning to keep all of the interesting records, but many of the uninteresting records can often be randomly discarded as long as you keep enough to still dominate the interesting records. This discarding is called downsampling and is usually done as part of the joining process.

Feature encoding often dominates training time after you have your training examples. Down-sampling the training data helps with this, but there are still a number of things you can do to make the encoding process itself more efficient.

Through it all, it is important to remember that building good classification models requires many cycles of iterative refinement. You need to make sure your training pipeline makes these iterations efficient. Section 16.3 covers these architectural and tactical issues involved in building a training pipeline.

AT CLASSIFICATION TIME

When you come to integrating your classifier into your system, your overall classification throughput needs will drive exactly what you need to optimize in the classifier. Most Mahout classifiers are fast enough that this will only matter in extreme systems, but you need to check to see if you are in the extreme category. If not, you are ready to roll with the standard methods, but if you are extreme, then you will need to budget time to stretch the normal performance limits.

If you need to do small numbers of classifications per transaction and have only modest latency and throughput requirements, then it should suffice to use conventional service design with a single thread per transaction. Rest based services may be acceptable or you may need to use a somewhat higher performance network service mechanism like Thrift, Avro or Protocol Buffers. Section 16.4 has an example of how to build such a service.

If you require tens to thousands of classifications per evaluation and have more stringent latency and throughput requirements, you will need to augment the basic classification service with a multi-threaded evaluation of multiple models or inputs, but otherwise, the basic design should still suffice.

For extreme batch sizes with very large numbers of model computations per service transaction, you will need to dive into the heart of the model evaluation code in order to divide the model evaluation into portions that can be evaluated ahead of time and portions that must be evaluated at classification time. Details of such optimization are highly specific to your particular application and are largely beyond the scope of this book. An example of such an implementation using Mahout is given in Chapter 17.

16.3 Building a training pipeline for large systems

Classifications systems need training data, and scalable classification systems can consume truly enormous amounts of data. In order to feed these systems, you have to be able to process even larger volumes of data in order to pick out the bits that you can use for training your classifier.

In order to handle these huge volumes of data, there are a few things to keep in mind. The first and most important is to match your techniques to your task. If you have less than about a million training examples, for instance, almost any reasonable technique will work including conventional relational databases or even scripts that process flat file representations of your data. At 100 million training examples, relational databases can still do the necessary data preparation, but getting the data out becomes progressively more and more painful. At a billion examples and beyond, relational databases become impractical, and you will probably have to resort to map-reduce based systems like Hadoop.

A billion training examples sounds extreme, but it is surprisingly common to reach that level. Shop It To Me, the company we profile in Chapter 17 for example, sends several million emails each day, each containing hundreds of items. As a result, they generate several hundred million training examples per day, a number that rapidly turns into billions of training examples. The larger ad networks all have well above a billion ad impressions per day, each of which is a potential training example. A music streaming service with a million unique visitors per month can produce nearly a billion training examples each month.

No matter what your volume, your model training data pipeline will need to support the following capabilities:

- Data records will have to be acquired and stored coherently, typically in time-based partitions
- Data records will have to be augmented by joining against reference data of some kind
- Data records will need to be joined to target variable values
- Down-sampling of training data may be useful and should be supported
- Other kinds of filtering and projection beyond simple down sampling will be required
- Training records will need to be encoded as training features
- Trained models will have to be retained and associated with meta-data describing how to encode input records.

For the most part, all the parts of the training data pipeline can scale horizontally, but when it comes to learning a model, some of the most easily deployed Mahout models are not suitable for horizontal scaling. This means that at very high data volumes, early cleaning and processing of data can be done in a highly scalable fashion, but significant efforts may be necessary to make data encoding very efficient at the point that the model is actually trained.

This chapter will cover the most important aspects of how to build these capabilities into your data pipeline using Mahout and related technologies. We will focus here on techniques that you can use at large scales on the assumption that you will already be familiar with the techniques that you can use at smaller data scales.

16.3.1 Acquiring and retaining large scale data

Training a large-scale model means that you are going to have to deal with really big training data. As always, issues that are trivial with small-scale data often balloon to magnificent and dire proportions when working with large-scale data. Most of these problems arise from the fact that large-scale data has a significant amount of what is metaphorically called inertia. This inertia makes it hard to move or

transform large-scale data and requires that you plan out what you intend to do. In the absence of direct experience with a new class of data, this planning is impossible to do effectively. Instead, you have to fall back to following standard best practices until you have enough experience with your data.

ACQUIRING DATA

The first problem with large data is acquiring it. This sounds easy, but as with all things to do with large-scale data resources, it is harder than it looks. The primary issue at this level is to keep the data relatively compact and to maintain natural parallelism inherent in the production of the data.

For instance, if you have a farm of servers each handling thousands of transactions per second, it is probably a bad idea to use a single centralized log server. Excessive concentration is bad because it leaves you liable to single point failures and because it is very easy for the data acquisition system to back up and adversely affect the operation of the system being observed. To avoid this problem, it is crucial to build in load shedding in the data acquisition phase; it is preferable to lose data than for your production system to crash.

The second trick to acquiring large data is to make sure that you keep data intended for machines to read in machine-readable form. It is nice to have human readable logs, but extracting data from those logs is a common source of errors because of fairly loose format. It is better to keep some data in human readable form for humans to read and to keep the data you want to process in machine-readable formats.

For large scale data ingestion, there are a number of different data formats that can be used, but only a few that provide the necessary combination of compactness, fast parsing libraries and, most important, schema flexibility. Schema flexibility allows you to evolve the data that you store without losing the ability to read old data.

The two most prominent encoding formats that combine these features are Avro from the Apache and Protocol Buffers from Google. Both are well supported and both provide comparable features and flexibility. Storing data in comma or tab separated fields is commonly done, but this approach quickly leads to problems as schemas evolve due to confusion about which column of data contains which field. The confusion becomes especially severe when files with differing schemas must be handled concurrently.

PARTITIONING AND STORING DATA

As you acquire and store data, there are several organizing principles that need to be maintained to ensure that you wind up with a scalable system:

- directories should generally contain no more than a few thousand files. Larger directories can cause scaling problems because file opens and creates will begin to take longer with very large directories.
- programs should process as few files as possible both by not processing irrelevant data and by keeping files as large as practical. Avoiding irrelevant data limits the amount of data to be transferred while using large files minimizes the time lost due to disk seeks thus maximizing I/O speed.
- the directory structure used to organize files should support your most common scheme for selecting relevant training data. Organizing directories well allows you to find relevant files without sifting through all of your files.
- files should not generally be more than about 1GB in size to facilitate handling them. Files should be small enough that you can transfer them between systems in a reasonably short period of time. This limits the impact of network disruption by allowing you to repeat transfers quickly. About 1GB is a happy middle ground.

Following these principles generally leads to storing data with top-level directories that define the general category of data, with sub-directories that contain data from progressively finer time divisions.

> Tip It is generally good practice to concatenate small older files into larger files that represent large enough time periods to avoid excessively many input files for subsequent processing. If fine-grained time resolution is required for recent time periods, then files might be kept for very short periods such as 5 minute intervals, but beyond a few days, these should be accumulated into hourly and then into daily files. If the data for a single day total more than 1-2 GB, then keeping several files for each day is probably a good idea.

WORKING INCREMENTALLY

For the most part, feature extraction is not something that requires aggregation over long periods of time. That means that once you have done the necessary joins and data reduction to generate the corresponding training data for an hour's worth of data, that training data will probably not change when the next hour of data comes in. This characteristic makes it eminently feasible to collect training data incrementally rather than doing all feature extraction for the entire period of interest each time training is done.

Likewise, this incremental nature of the processing makes it possible to accumulate extracted features for short periods of time and then concatenate them for longer-term storage. It is common, for instance, to accumulate hourly training data and concatenate these hourly training files into dailies that are then kept pretty much forever. This corresponds nicely to the way that the raw data are usually kept.

When your classification system is relatively new, the addition of new features and changes in downsampling will likely mean that you will need to rebuild your training data files fairly often. Over time, the frequency of change typically decreases. Once this happens, incremental creation of training data is a big win because the lag between the arrival of data and the beginning of model training can be decreased.

16.3.2 Denormalizing and downsampling

It is common for the data that you acquire directly to require denormalization before they can be used to train a model because the original data may be scattered across many different storage formats including log files, database tables and other sources. In denormalization, data that could be less redundantly stored in separate tables connected by a key are joined together so that each record is self-contained without any references to external data such as user profile tables or product description tables. Since model training algorithms have no provision for resolving external references, their inputs have to be fully denormalized to be useful.

Denormalization is almost always done using some kind of join. With small to medium data sets, almost any method will suffice to do these joins, and tools such as relational databases are very useful. With large datasets, however, these joins become much more difficult to do efficiently. Moreover, the characteristics of your data must be taken into consideration to do these joins.

JOIN TARGET FIRST

Joins are also almost universally required to attach target variable values to the predictor variables. The target values should be attached if possible before denormalization because it is common to down-sample training records depending on the value of the target variable. By doing any downsampling before subsequent joins, it is often possible to achieve substantial savings during training because less data is processed with almost identical results.

IN-MEMORY JOINS

In some cases, training data needs to be joined to a secondary table with a limited number of rows. When this is the case, the secondary table can sometimes be loaded into memory, and the join can be done by consulting an in-memory representation of the second table. In-memory joins can be done inside a map-reduce program as part of either the mapper or the reducer, although when generating classification training data map-side joins are more common.

MERGE JOINS

Where both data sets to be joined are large, a merge join may be possible. In a merge-join in a sequential, non-parallel program, two data sets ordered on the same key can be joined in a single pass over both data sets.

In a map-reduce program, the situation is a bit more complex because the two data sets are very unlikely to be split in such a way that each map can merge data correctly. It is possible, however, to build an index on one file while sorting it. The path of the indexed data is provided as side data so that as each map starts reading a split of the unindexed data, it can seek to near the right place in the indexed data and start merging from that point.

Merge joins are advantageous when data are naturally sorted, but it is often overlooked that if only one of the inputs is sorted and indexed, then a merge join can be done in the reducer instead of in the

mapper. This choice allows you to use the map-reduce framework to sort the unindexed input. Whether this approach saves time relative to a full reduce join is something that can only be determined by experimentation.

FULL REDUCE JOINS

In terms of programming effort, the simplest way to join data is to do a full reduce join. This means that you simply read both inputs in the mapper and reduce on the key of interest. Such joins can be more expensive than merge joins because more data needs to be sorted in the Hadoop shuffle step. When you do a full reduce join, it helps to have Hadoop sort the results so that the training records come after the data being added to the training records to denormalize them. Having Hadoop do the sort allows the reducer to be written in a fully streaming fashion. Since the shuffle and sort in Hadoop are highly optimized, a brute-force full reduce join may actually be as efficient as a merge join, if not more so, especially if you are doing a reduce side merge join.

If your training data are larger than the data it is being joined to, then a full reduce join and any kind of merge join are likely to be quite comparable in speed.

16.3.3 Training pitfalls

Even assuming that your data are what you think it is, your joins are working correctly and you are parsing your data correctly, there are a few pitfalls that are very easy to run into. The most subtle and difficult to diagnose is a target leak. Also common is the use of an inappropriate encoding which leads to semantic mismatch between what you intended and how the model sees the data. Let's look at these issues in more detail.

WATCH OUT FOR TARGET LEAKS

Keeping daily training files also substantially facilitates time-shifted training that is often necessary to avoid target leaks. For instance, suppose one feature that you want to use in a targeting system is a clustering of users based on click history. If you use that clustering to train a model on the same time periods as you used to derive the clustering, you will probably find that the resulting model will do exceedingly well at predicting clicks. What is happening, however, is that your click-history clustering is probably putting everybody with no clicks into different clusters than everybody with clicks. This division means that the user cluster is a target leak in the time period when the clustering was created. A picture of how such a target leak can creep into your design is in figure 16.1.

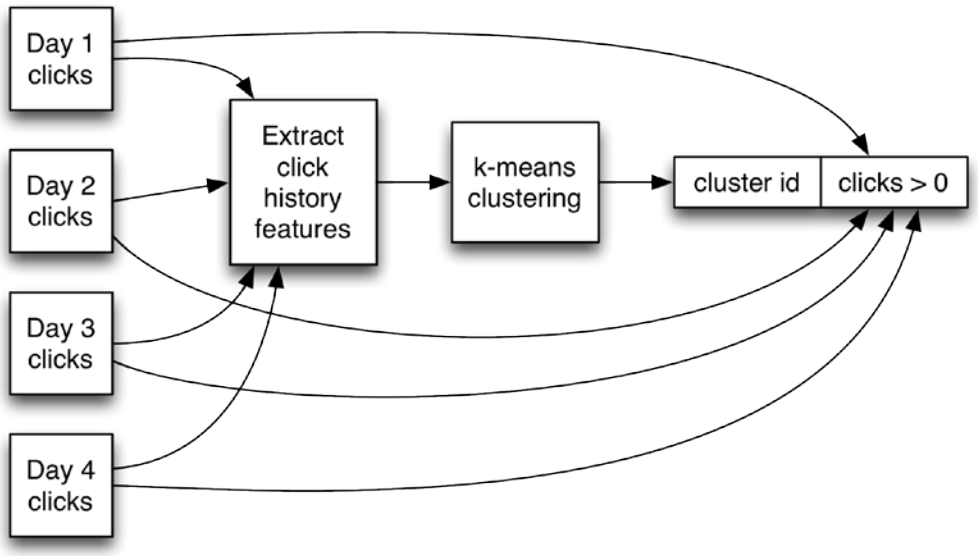


Figure 16.1 Don't do this: Click-history clustering can introduce a target leak in the training data because the target variable (clicks > 0) is based on the same data as the cluster id.

To fix this leak, we can't just delete the problematic variable. The user-history clustering could still be a valuable predictive variable. The problem here is simply that the models is not being trained on new data.

Figure 16.2 shows how this can be improved by clustering on early data and then training on more recent data unknown to the clustering algorithm. Moreover, the held-out training data can be on even more recent data.

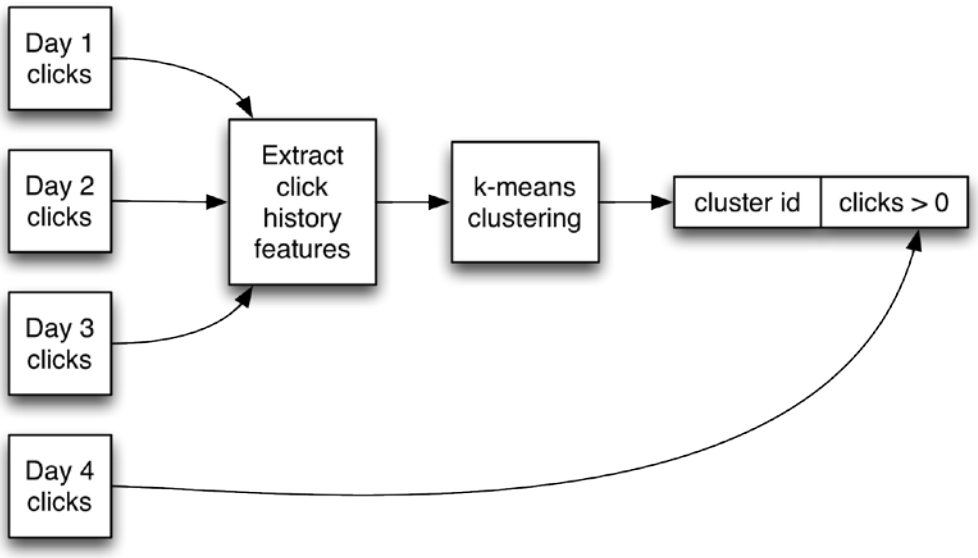


Figure 16.2 A good way to avoid a target leak: Computing click history clusters based on days 1, 2 and 3 while deriving the target variable (clicks > 0) from day 4 prevents any target leak.

Keeping data segregated by time makes holding out data this way very easy. In practice, we would use data from day 5 in this scenario as held-out data to evaluate model performance.

SEMANTIC MISMATCH DURING ENCODING

A rose by any other name would still smell as sweet, but with training data, an integer is not always what it seems. In some cases, an integer represents a quantity such as the number of hours or days between two events. In other cases, integers are used as identifiers for categories. The distinction between these cases has been covered in previous chapters, but in spite of the best intentions, it is still common for integer data to be treated incorrectly when training models. This mistake can be hard to debug because a model built with incorrectly encoded data may actually be able perform correctly a surprising amount of the time.

The key to diagnosing such problems is to review the internal structure of the resulting model to look for a few weights applied to each field. If you see lots of weights for particular values of a field, then the field is being treated as an identifier for a categorical or word-like value. If you see a single weight for the field, then it is being treated as a continuous variable. Whichever way the model is thinking of the field should match the way you think of the field.

The one exception to this sort of rule is when you have integer values that really are integers, but which only have a few distinct values. Treating such variables as a categorical value may be valuable even it is more correctly treated as a continuous value. Only experimentation can really tell you if this will work for you, but whenever you have integers with a small number of values, this trick may be worth trying.

16.3.4 Reading and encoding data at speed

For many applications, the methods for reading, parsing and encoding data as feature vectors that were discussed in previous chapters are plenty good enough. It is important to note, however, that the SGD classifiers in Mahout only support training on a single machine rather than in parallel. This limitation can lead to slow training times if you have vats of training data. A profiler would reveal that most of the time in a straightforward use of Mahout's SGD model training API is actually spent in preparing the data for the training algorithm, not in doing the actual learning itself.

To speed this up, you have to go to a lower level than is typical in Java programs. The String data type, for instance, is handy in that it supports Unicode and lots of libraries for parsing, regex matching and so on. Unfortunately, this happy generality also has costs in terms of complexity. It is common for training data to be subject to much more stringent assumptions about character set and content than is typical for general text.

Another major speed bump that training programs encounter is a copying style of processing. In this style, data are often processed by creating immutable strings that represent lines of input. These lines are segmented into lists of new immutable strings representing input fields. Then, if these fields represent text, they are further segmented into new immutable strings during tokenization. The tokens in the text may be stemmed, which involves creating yet another generation of immutable strings.

Doing all of the allocations involved in such a copy-heavy programming style costs quite a bit, and lots of people focus on reducing allocation costs by re-using data structures extensively. The fact is, however, that the real cost is not so much the cost of allocation, but the cost of copying the data over and over. An additional cost is due to the fact that constructing a new String object in Java involves creating the hash-code for each new string. Computation of the hash code costs almost as much as copying the data.

All of this copying provides us an opportunity to increase speed enormously by simply working at a lower level of abstraction and avoiding all of these copies. In this lower level style, if we do allocate new structures, we try to make sure that they reference the original data rather than copy it. These small structures are also highly ephemeral, so the cost to allocate and collect them is nearly zero.

As an example of a string copying style, here is a simple class that uses strings to parse tab or comma separated data. This class illustrates a simple, clear and somewhat slow approach to parsing this kind of data.

```
class Line {
  private static final Splitter onTabs = Splitter.on(SEPARATOR);
  private List<String> data;
  private Line(String line) {
```

```
      data = Lists.newArrayList(onTabs.split(line));
    }
    public double getDouble(int field) {
      return Double.parseDouble(data.get(field));
    }
  }
```

Obviously, important details are left out here, but the key idea is clear. The class splits a string representation of a line into a list of strings. This splitting and allocating of new strings requires that we copy the data around repeatedly. Moreover, by copying strings, we are copying two bytes per character.

Listing 16.1 shows how this code could be used to parse and encode data from a file containing CSV data.

Listing 16.1 Code to parse and encode CSV data

```
  public static void main(String[] args) throws IOException {
    FeatureVectorEncoder[] encoder = new FeatureVectorEncoder[FIELDS];
   for (int i = 0; i < FIELDS; i++) {
      encoder[i] = new ConstantValueEncoder("v" + 1);     #A
    }
    long t0 = System.currentTimeMillis();
    Vector v = new DenseVector(1000);
    BufferedReader in = new BufferedReader(new FileReader(args[1]));
    String line = in.readLine();
    while (line != null) {
      v.assign(0);                                        #B
      Line x = new Line(line);                            #C
      for (int i = 0; i < FIELDS; i++) {
        encoder[i].addToVector(x.get(i), v);              #D
      }
      line = in.readLine();
    }
    System.out.printf("\nElapsed time = %.3f s\n",
          (System.currentTimeMillis() - t0) / 1000.0);
  }
```

#A build one encoder per field
#B reset contents of vector
#C parse string
#D and encode each field

This code reads each line, one at a time, and parses each line using the `Line` class shown above. For each field found, a feature encoder is used to add the field to the feature vector. Separate encoders are used for each field. When used to parse a million lines, each with 100 data elements, the string style of parsing and encoding shown in listing 16.1 takes 75-80 seconds running on a 2.8GHz Core Duo processor. If we make few changes, you can speed this up by about a factor of at least 5. With the improvements we are about to describe below, a program that does the same work takes well under 15 seconds.

SEGMENT BYTES, NOT CHARACTERS

The first step to speed things up is to use large-scale byte I/O, not to convert bytes to `String` and definitely to avoid copying pieces of the input over and over. In the `Line` class, each line is copied from the input, then copied again when it is converted to a string and then copied again as the pieces are parsed out. All of this copying makes the program easier to understand and to debug, but it definitely slows things down quite a bit.

For many data files, avoiding the conversion to Java strings is a fine thing to do because they have a limited data format that can be expressed in an ASCII character set where each character is encoded by a single byte code. Java characters are twice as large as bytes, and moving them around requires twice as much memory bandwidth. Even better, with byte arrays, we have the option to avoid copies almost entirely. If the fields in question contain numbers, then using the normal Java primitives means that the fields will be parsed by routines that are capable of converting general Unicode strings into numbers. This conversion is harder than it sounds because basic digits appear many times in Unicode. If your data is all ASCII, conversion can be done a lot more simply than converting your data to Unicode and then converting such a general representation to a number.

Listing 16.2 shows an adaptation of the `Line` class called `FastLine` that uses a `ByteBuffer` to avoid copies. `FastLine` also parses numbers in a way that is highly specialized for parsing one and two-digit integers encoded in an ISO Latin character set.

Listing 16.2 Code for byte-level CSV parsing

```
private static class FastLine {
  private ByteBuffer base;
  private IntArrayList start = new IntArrayList();     #A
  private IntArrayList length = new IntArrayList();    #A
  private FastLine(ByteBuffer base) {
    this.base = base;
 }
 public static FastLine read(ByteBuffer buf) {
    FastLine r = new FastLine(buf);
    r.start.add(buf.position());                       #B
    int offset = buf.position();
    while (offset < buf.limit()) {
      int ch = buf.get();
      switch (ch) {
       case '\n':                            #C
         r.length.add(offset - r.start.get(r.length.size()) - 1);
         return r;
        case SEPARATOR_CHAR:
          r.length.add(offset - r.start.get(r.length.size()) - 1);
          r.start.add(offset);               #D
          break;
        default:
       }
    }
    throw new IllegalArgumentException("Not enough bytes in buffer"); #E
   }
    public double getDouble(int field) {
     int offset = start.get(field);
      int size = length.get(field);
      switch (size) {
       case 1:
          return base.get(offset) - '0';
        case 2:
          return (base.get(offset)-'0') * 10+base.get(offset + 1) - '0';
        default:
          double r = 0;
          for (int i = 0; i < size; i++) {
            r = 10 * r + base.get(offset + i) - '0';
          }
         return r;
      }
  }
}
```

#A special collections avoid boxing
#B we just remember a reference
#C remember end of line is end of field
#D start of next field
#E assume buffer has a full line

The `FastLine` class in listing 16.2 uses `IntArrayList`'s from the Mahout collections package to store offsets and lengths. This has two effects. One is to avoid boxing and unboxing of integers. A second effect is that fields are kept as references into the original byte array to avoid a copy. The parsing of the file data is based on several strong assumptions that depend on the input data being somewhat constrained. First, there is an assumption that the `ByteBuffer` always has enough data in it to complete the current line. Second, the data is assumed to use Unix-style line separators consisting of a newline character with no carriage return character. Thirdly, it is assumed that only the ASCII subset of characters is used.

These assumptions allow considerable liberties to be taken with the result that the time required for just reading and parsing the data takes less than one third the time required for the same operations in the `String` based code. `FastLine` is good, but is not the whole story.

DIRECT VALUE INTERFACE TO VALUE ENCODERS

Values can be encoded as vectors in a number of ways. For continuous variables, we can use the `ContinuousValueEncoder` and pass in the value as a string as we did in previous chapters. On the other hand, if we already have the desired value in the form of a floating-point number, we can pass in a null as the string and pass in the value in the form of the weight.

In the case of a direct byte parser, we can have access to the value of the field very quickly without converting to string and then to a floating-point representation. This makes the short-cut use of the weight field on the `ConstantValueEncoder` attractive. The code in listing 16.3 shows how this can be done.

Listing 16.3 Direct value encoding is fast

```
public static void main(String[] args) throws IOException {
  FeatureVectorEncoder[] encoder = new FeatureVectorEncoder[FIELDS];
  for (int i = 0; i < FIELDS; i++) {
    encoder[i] = new ConstantValueEncoder("v" + 1);
  }
  long t0 = System.currentTimeMillis();
  Vector v = new DenseVector(1000);
  FastLineReader in = new FastLineReader(new FileInputStream(args[1]));
  FastLine line = in.read();
  while (line != null) {
    v.assign(0);
    for (int i = 0; i < FIELDS; i++) {
      encoder[i].addToVector((byte[]) null, line.getDouble(i), v);  #A
    }
    line = in.read();
  }
  System.out.printf("\nElapsed time = %.3f s\n",
      (System.currentTimeMillis() - t0) / 1000.0);
}
```

#A note the null string value

This code is similar to the string-oriented encoding code, except for the use of `FastLine` instead of `Line`. The same encoders are constructed and lines are still read, though from a `FastLineReader` instead of from a string level input. The encoding of each field is a bit different since we let the `FastLine` class convert the byte representation to a number to avoid letting the encoder do the job.

The final version of the code using `FastLine` is about five times faster, as was mentioned earlier, but the effect of this direct value encoding trick in isolation is even more impressive. For the million lines of encoding ignoring the time required for data parsing, it takes about 4-5 seconds to encode all of the values when they are extracted from the byte representation, but takes over 40 seconds to encode the values from string form.

The overall effect of these speedups is that raw data to be classified can be read, parsed and encoded at a rate of about 15MB/s without much effort. By using several threads for reading, the aggregate conversion rate can easily match disk transfer speeds for several spindles. With more elaborate encodings including lots of interaction variables, the cost of encoding will be higher, but these improvements will still have large impacts.

Such optimizations are worthwhile if you are running large modeling runs, especially if you are using a lower level `CrossFoldLearner` or `OnlineLogisticRegression` object directly without the mediation of an `AdaptiveLogisticRegression` because these lower level learners are so fast that simply reading the training data can be the limiting factor on performance. On the other hand, using the more abstracted string-based methods allows you to write code that is much easier to write and debug and much less prone to surprises if your data files might have oddities in them.

Once you have your data read (quickly) and converted (quickly), the next performance bottleneck is likely to actually integrating your classifier into a server.

16.4 Integrating a Mahout classifier

Integrating a Mahout classifier is usually just a matter of creating a simple networked service. As such, general issues of throughput, latency, and service updates that apply to all networked services apply to a

Mahout classifier service. These issues can take on a slightly different hue with Mahout classifiers than is customary for a variety of reasons because Mahout classifiers are often integrated into applications that require extreme speed. To some extent, services based on Mahout classifiers are actually simpler than most services since they are stateless. This statelessness allows trivial horizontal scaling. This section will cover how to plan and execute the job of integrating a Mahout classifier into a service-based architecture.

16.4.1 Plan ahead: key issues for integration

Although Mahout-based services have the advantage of being simple, there are still some key things to be considered in the creation of such a service in order to use Mahout to its full potential. These key issues include best practices about the division of responsibility for the client and server, checking production data for differences from training data, designing for speed and handling updates.

DIVIDING RESPONSIBILITIES

One of the critical architectural choices you will face as you integrate classification models into a classification service is the division of responsibilities between the client side and server side of your system. A well considered division can greatly enhance performance and will help future-proof your design.

In general, it is important to ensure that the server handles encoding features and evaluating the model. There is a grey area, however, in how much feature extraction should be done on client or server sides. For instance, if the client has a user ID and a page URL but the server needs information from the user's profile and some content features based on the content of the page, then either the client or the server can do the necessary join against the user profile database and a page content or feature cache. Doing these join and feature extraction operations on the client side will leave the model server with a much more circumscribed task and will likely result in a more reliable server. Doing these join and extraction operations on the server side will make certain kinds of caching easier and will hide the details of the model from the client a bit more.

Figure 16.3 illustrates how the responsibilities can be divided.

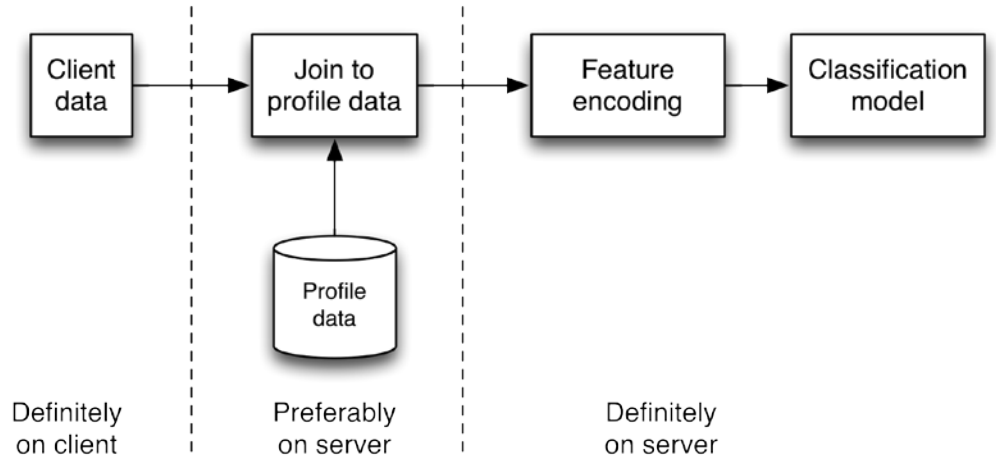


Figure 16.3. Joins with profile data such as user profiles is preferably done in the server, but these joins may be done by the client. Feature encoding should definitely be done on the classification server.

It is a good design principle for the model server to accept whatever form of data the client programs already have. If possible, the model server should not require extensive data preparations to be done by the client unless they were already being done before the model service existed. A major exception to this principle is when access to necessary join resources by the model server would create a need for access to secured resources already available to the client. In such a case, necessary joins in preparation for feature encoding may have to be done in the client.

REAL-TIME FEATURES ARE DIFFERENT

One major issue with deploying a model into production is that the data that are given to a classifier in production are often quite different from the data that are reconstructed for training a classifier. These differences are generally not intentional, and they can lead to subtle bugs and apparent poor performance of the classifier system if not handled properly.

What sort of differences are you likely to encounter? Some of these differences are due to time shifting processes similar to target leaks. For instance, suppose you are trying to predict whether a user will purchase a particular product. The predictor variables for the problem are based on the contents of the user's profile. In creating the training data for this model, it would be easy to collect user profiles, as they are at the time the training data is collected rather than as they were at the time the purchase decision was made. If the completion of the purchase process results in additional information being stored in the user profile, any model built using the time-shifted profiles is likely to behave very poorly when given non-shifted profiles.

LOGGING REQUESTS

One of the best ways to detect when production data differs from training data is to log all classification requests even before the classification system is ready for deployment. After a short period of logging, you can use any technique you prefer to extract training data for the period of time in question and compare the logged data with the extracted data. If they differ, then the training data extraction process must somehow be changed to avoid extracting time shifted training examples. Ultimately, it may be that only directly logged data can be used for training.

DESIGNING FOR SPEED

Speed considerations when building a classification often go to paradoxical extremes. When a Mahout classifier is properly integrated, extreme speeds are definitely achievable. If your service is evaluating a single model against a single input record, and the classification server does not have to do any expensive joins, the model evaluation is likely to be so fast that network overheads will dominate. The speed problem will reduce to simply finding an access method that can drive high volumes of request through the server.

For instance, a Thrift server running locally takes a bit less than 50-100µs to do a server request no matter how little the request actually does. Since a typical model evaluation requires no more than a thousand or so floating point operations, it will finish in a few tens of microseconds at most, resulting in 90% of the time being spent in overhead. The throughput will be a bit higher than this due to threading and network transit time, but it is clear that most of the time is not spent on model evaluation. In such a system, the key place to put your effort is in optimizing network latency. To achieve highest speed, you may need to integrate the model directly into your client code, avoiding server round trips entirely. This direct integration minimizes latency for each model evaluation, but it may substantially complicate model update.

> Tip If you need to evaluate one example at a time, focus on decreasing network round trip times to minimize evaluation latency and consider client-side model evaluation. For model updates, you may need some kind of server to simplify distribution of model updates rather than depending on the file being accessible as a local file.

In the middle of the speed spectrum, some uses of classifier models require many models to be evaluated at a time or allow many feature vectors to be sent together in a single server request. This batched evaluation approach can allow substantial increases in efficiency simply by virtue of the fact that network latency is amortized over more model computations. Latency is not decreased relative to the case with a single evaluation per server request, but the amount of computation that can be done per request is increased dramatically.

> Note Doing lots of model evaluations per server request is a good thing, especially in terms of throughput. Many targeting systems require model evaluations for lots of items and thus allow this optimization.

At the extreme end of the speed spectrum, systems typically have enormous numbers of inputs to be evaluated. The computational marketing system described later in this chapter, for instance, requires that tens to hundreds of thousands of products be evaluated for each user, resulting in as many model evaluations as there are items. One request is made per user, and a feature vector is constructed using user parameters, item parameters and user-item interactions. The models in that system are also fairly elaborate, involving hundreds or thousands of non-zero features for each item. In such cases, the evaluation time can be several hundred milliseconds, and computation time totally dominates all architectural decisions.

COORDINATING MODEL UPDATES

In a production system that is handling a stream of classification requests, it is fairly common for there to be an expectation of nearly 100% uptime. Since a classifier is normally stateless, this uptime requirement can be satisfied by placing a pool of classification servers behind a load balancer. Whenever the classification model is updated and released to production, it is necessary for the servers to load the new model and start using the new model for classification. In some cases, there is an additional requirement that model updates be made visible at essentially the same time across the entire pool of servers to maintain consistency between all servers.

> Tip Regardless of your exact requirements, we highly recommend using a coordination service such as Apache Zookeeper to manage model updates and to provide a reliable list of live servers. Among such services, Zookeeper is far and away the most popular.

If exactly simultaneous updates are not necessary, then the overall architecture can be quite simple. Zookeeper needs to hold a model configuration. Classification servers request notifications about changes to this configuration, and while they have a live model, they maintain indicator files that tell classification clients which servers are handling requests. Classification clients interrogate Zookeeper to determine which classification servers are available.

This coordination data in Zookeeper can be arranged as follows

```
/model-farm/
   model-to-serve
   current-servers/
     10.1.5.300
     10.1.5.301
     10.1.5.302
```

In this directory structure, the `/model-farm/model-to-serve` file contains the URL of the serialized version of the model that all servers should be using. Each live server reads this file on startup and tells Zookeeper to provide notifications of any changes. In addition, each server should poll this file every 30-60 seconds in case notifications could not be set, possibly because the file did not exist when the server was started.

When Zookeeper notifies the server that a change to model-to-serve has been made, the server should download the serialized form of the model and instantiate a new model. This new model should then be used to serve all subsequent requests. Once the new model is properly installed, the server should create a file in the `/model-farm/current-servers` directory. This file should be named with a name unique to the server and should contain, either in the name or file contents, information necessary for a client to send requests to the server. Such information might include a host name and port number.

Figure 16.4 contains a ladder diagram showing a typical time sequence for deployment of a new model.

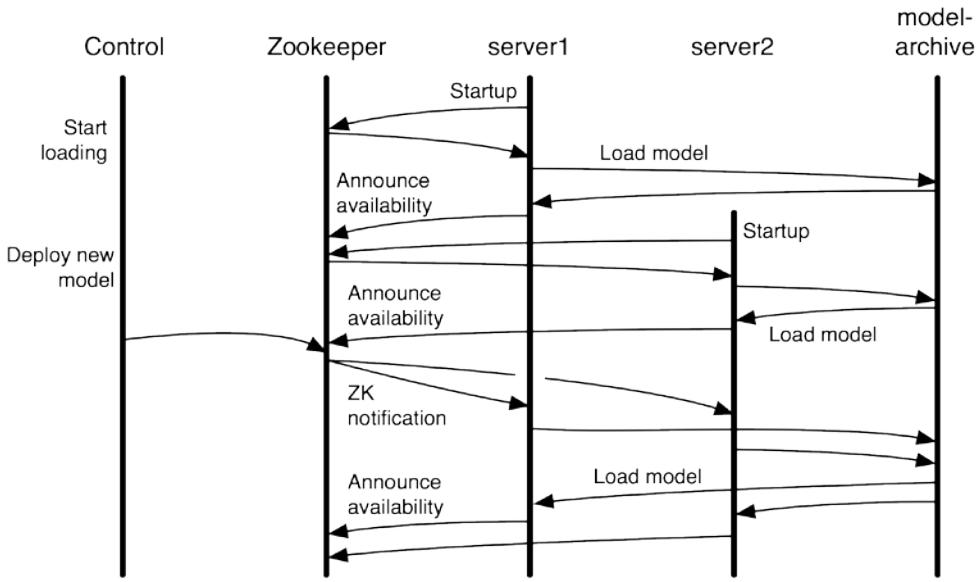


Figure 16.4. How model deployments are coordinated with Zookeeper. The control process, server1, and server2 all communicate via Zookeeper, which also notifies server1 and server2 when changes happen.

In the sequence depicted in figure 16.4 we assume that Zookeeper starts with a model specification. When server1 starts, it loads the specification from Zookeeper and then loads the model. When the model is fully loaded and ready to serve, server1 creates a file in Zookeeper to indicate it is ready. When server2 starts up a short time later, the same sequence ensues. When the control process updates Zookeeper with a new model, notifications are passed to server1 and server2, and the model download and announcement process repeats.

This process is simple to understand and easy to implement. It has significant defects, however, in that all servers run the same model, and all servers update their models essentially simultaneously. This simultaneous access could conceivably result in degraded service across the entire cluster because servers will switch to using the new model whenever they have completed the load process, which in turn could cause a short time when different servers will give different answers. If the model file is large, having all servers read it simultaneously could cause problems due to network overload.

Here is a directory structure that shows how updates can be coordinated more carefully so that a more cautious model update can be done

```
/model-farm/
   should-load/
      node-10.1.5.300
      node-10.1.5.301
      node-10.1.5.302
      node-10.1.5.303
   currently-loaded/
      node-10.1.5.300
      node-10.1.5.301
      node-10.1.5.302
      node-10.1.5.303
   model-to-serve/
      node-10.1.5.300
      node-10.1.5.301
      node-10.1.5.302
      node-10.1.5.303
```

In this directory structure, the top-level directory `model-farm` is the name of the overall model cluster. Below this, the `should-load` directory contains a single file for each server named with the address of the server. The contents of each of these files is a list of URL's and MD-5's that can be used to retrieve and verify the content of the model. Parallel to the `should-load` directory is the `currently-loaded` directory. The `currently-loaded` directory contains a single ephemeral file created by each

active model server that contains a list of the MD-5 hashes for all models that are currently loaded on that server. Finally, another sibling directory called `model-to-serve` contains one file per server node that contains the hash of the single model that should be used for all incoming requests.

In operation, each model server maintains a watch on the `should-load` directory and the `model-to-serve` file. Changes in the `should-load` directory signal that a model should be loaded or dropped, while changes to the `model-to-serve` file signal that all subsequent requests be routed to the indicated model. Whenever a model is loaded or unloaded, the corresponding ephemeral file in `currently-loaded` is updated.

This directory structure allows considerably more flexibility than the previous model at the cost of implementation complexity. This flexibility allows new models to be loaded and staged on multiple machines before making them available essentially simultaneously. It also allows loading and running a new model on only one server in order to test for stability of the new model before rolling it to all of the servers. The use of MD5 hashes allows the re-use of URLs with new content and allows servers to verify that they have loaded the intended model.

In practice, you may need to extend this model to allow control of multiple model server farms or you may want to automate how models are rolled out to a progressively more machines. It is also common to have servers update their status files with indicators about how much traffic they can handle. Clients sending requests can use these indicators when picking a server to send a request to.

Also, keep in mind that even if you train your model using an `AdaptiveLogisticRegression`, which contains many models inside you only need to save a single model from one of the underlying `CrossFoldLearners`. This is important because the `AdaptiveLogisticRegression` contains a large number of classifiers, each of which has a potentially very large coefficient matrix inside. With a large feature vector size, the size of a serialized `AdaptiveLogisticRegression` can actually be hundreds of megabytes.

16.4.2 Model serialization

As of version 0.4 of Mahout, the SGD models and the Naïve Bayes models in Mahout are serialized in different ways. The SGD models share a helper class called `ModelSerializer` that handles serialization and deserialization of all SGD models. In contrast, all Naive Bayes models are only ever serialized as a side effect of the multiple files created during training. Deserialization of Naïve Bayes models is not done using a single method, but rather by reading these multiple files back into memory explicitly. In future versions of Mahout it is likely that the `ModelSerializer` or similar class will likely be extended to handle Naive Bayes models as well as SGD based models. Until then, serialization and deserialization of Naive Bayes models will remain a tricky and highly variable process. Until a more usable serialization interface is completed, this means that any deployment of a Naive Bayes model should be done using the command line interface we described in previous chapters rather than programmatically.

USING THE MODELSERIALIZER FOR SGD MODELS

The `ModelSerializer` class provides static methods that serialize models to files and strings. The use of the class is quite simple as shown below where both a single model from an `AdaptiveLogisticRegression` as well as the complete ensemble model are saved to different files.

```
if (learningAlgorithm.getBest() != null) {
   ModelSerializer.writeBinary("best.model",
      learningAlgorithm.getBest().getPayload().getLearner());
 }
 ModelSerializer.writeBinary("complete.model", learningAlgorithm);
```

All of the different kinds of SGD models can be serialized using the same approach.

Reading a model from a file is just about as easy, as shown here

```
learningAlgorithm = ModelSerializer.loadBinary(
             new FileInputStream("complete.model"),
             AdaptiveLogisticRegression.class);

OnlineLogisticRegression bestSubModel = ModelSerializer.loadBinary(
        new FileInputStream("best.model"), OnlineLogisticRegression.class);
```

In this snippet, two uses of a `ModelSerializer` are shown. The first might be used to reload an entire `AdaptiveLogisticRegression`, complete with sub-models while the second can be used to load a single `OnlineLogisticRegression` which might be one of the component models in an `AdaptiveLogisticRegression`. Note how you have to supply the class to the `loadBinary` method so that it can know what kind of object to construct and return.

When serializing SGD models for deployment as classifiers, it is usually best to only serialize the best performing sub-model from an `AdaptiveLogisticRegression`. The resulting serialized file will be 100 times smaller than would result from serializing the entire ensemble of models contained in the `AdaptiveLogisticRegression` object. In addition, when data is to be classified the individual model is more suited to deployment since only the best sub-model from the `AdaptiveLogisticRegression` object would be used and all of the other models would be ignored.

On the other hand, if you are serializing a model so that you can continue training at a later time, then serializing the entire `AdaptiveLogisticRegression` is probably a better idea.

16.4.3 Example: Thrift-based classification server

Putting all the bits together of a working classification server can be a bit daunting. To help with that, here is an example. We have written a simplified, but complete, classification server that implements the simpler model deployment scheme described earlier. This server provides all of the basic capabilities needed to allow fully functional load-balanced classification service including some administrative capabilities.

Communication between the classification client and classification server will be handled in this example using Apache Thrift. Thrift is an Apache project that provides allows client-server applications to be built very simply. Coordination of multiple servers and between server and client will be handled in this example using Zookeeper. Zookeeper is another Apache project that provides a distributed coordination service.

In order to facilitate control of the server and to allow us to monitor the operation of the server, we use Apache Zookeeper. In this example, Zookeeper holds instructions for all of the classification servers about which model to load as well as status information from all of the servers so that we can tell which servers are up and which model they are serving. This server coordination structure is shown in figure 16.5.

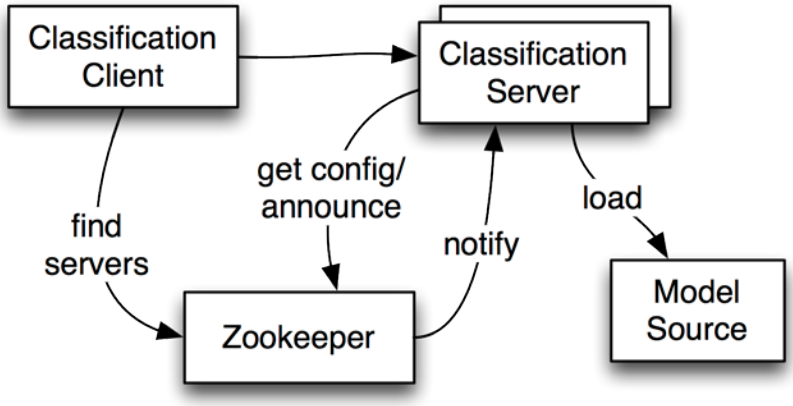


Figure 16.5. Zookeeper serves as a coordination and notification server between the classification client and classification server.

This arrangement allows the classification server to know where to find the model definition and allows the client to find all active servers.

When a server comes up, it queries Zookeeper to find out what model to load. After loading that model, it announces to the world that it is ready for traffic by writing a file into Zookeeper. Whenever the file indicating which file to load is changed, Zookeeper notifies all servers of the change.

When a client wishes to send a query to a server, it can look into Zookeeper to find which servers are currently providing service and pick one server at random.

The structure of the files in Zookeeper that coordinate everything is as follows

```
/model-service/
   model-to-serve
   current-servers/
     hostname-1
     hostname-2
```

The file called `model-to-serve` contains the URL of the model that should be loaded and used for classification by all servers. These servers will maintain a watch on this file so that they are notified as soon as this file changes. When they see a change, they will load the file to get the model URL and then they will reload the model. As soon as any model is loaded by a server, it will create an ephemeral file in the current-servers directory with the name of the server. Since this file is ephemeral, it will vanish within a few seconds if the corresponding server ever crashes or exits.

The code for the main class of this model server is shown in listing 16.4. It mostly contains the setup code for the Thrift server layer, but note how a timer is used to schedule periodic checks in Zookeeper. This should be completely redundant, but it is a good example of belt-and-suspenders coding style. We know Zookeeper should notify us of any changes, but checking in now and again makes sure we will catch up if something gets lost due to a programming error.

Listing 16.4. Main program for classification server

```
public static final String ZK_BASE = "/model-service";
public static final String ZK_CURRENT_SERVERS =
              ZK_BASE + "/current-servers";
public static final String ZK_MODEL = ZK_BASE + "/model-to-serve";
private final TServer server;
private final Logger log =
              LoggerFactory.getLogger(this.getClass());
private final ZooKeeper zk;
private final Ops modelHandler;
private Timer timer;
private String currentUrl = null;
private int version;
private Watcher modelWatcher = new Watcher() {

  ...  #1
}
public Server(int port)
throws TTransportException, IOException,
                      InterruptedException, KeeperException {
 zk = new ZooKeeper("localhost", 2181, null);     #A
  modelHandler = new Ops();                       #B
  timer = new Timer();                            #C
  timer.scheduleAtFixedRate(new TimerTask() {
  @Override
  public void run() {
   modelWatcher.process(null);
  }
},  0,  30000);
socket = new TServerSocket(port);                 #D
Classifier.Processor processor = new Classifier.Processor(modelHandler);
TProtocolFactory protocol = new TBinaryProtocol.Factory(true, true);
server = new TThreadPoolServer(processor, socket, protocol);
log.warn("Starting server on port {}", port);

server.serve();                                   #E
}
public static void main(String[] args) throws IOException,
              TtransportException, InterruptedException, KeeperException {

  new Server(7908);
```

```
}
```

#1 details in listing 16.5
#A connect to local Zookeeper
#B create classifier internal service
#C retry model load every 30 seconds
#D set up thrift server
#E start server

Most of the action is in the `modelWatcher` object. This object is an implementation of a Zookeeper Watcher and is invoked whenever a watch is triggered by the modification of the model-to-serve file. The source code for the `modelWatcher` is given in listing 16.5.

Listing 16.5. The watcher object loads the model and sets model status.

```
private Watcher modelWatcher = new Watcher() {
  @Override
  public void process(WatchedEvent watchedEvent) {
    String hostname = null;
    try {
   hostname = InetAddress.getLocalHost().getHostName();
 } catch (UnknownHostException e) {
    }
 if (hostname == null) {
  log.error("Must have hostname ... exiting");
      System.exit(1);
 }
 String url = null;
 try {
   Stat stat = new Stat();                                    #1
   byte[] urlAsBytes = zk.getData(ZK_MODEL, modelWatcher, stat);
   int latestVersion = stat.getVersion();
   url = new String(urlAsBytes, Charsets.UTF_8);
   if (currentUrl == null || latestVersion != version) {       #A

     URL modelUrl = new URL(url);
     log.warn("Loading model from " + modelUrl);
     AbstractVectorClassifier model = ModelSerializer.readBinary(
         modelUrl.openStream(),
         OnlineLogisticRegression.class);
    try {
       zk.create(ZK_CURRENT_SERVERS + "/" + hostname,       #2
                   modelUrl.toString().getBytes(Charsets.UTF_8),
                   ZooDefs.Ids.OPEN_ACL_UNSAFE, CreateMode.EPHEMERAL);
     } catch (KeeperException.NodeExistsException e) {

       zk.setData(ZK_CURRENT_SERVERS + "/" + hostname,  #3
                   modelUrl.toString().getBytes(Charsets.UTF_8), -1);
     } catch (KeeperException e) {
       log.error("Couldn't write server status file");
     }
     modelHandler.setModel(model);
     currentUrl = url;
     version = latestVersion;
     log.info("done loading version " + version);
  }
   return;
} catch (KeeperException.NoNodeException e) {      #B
  log.error("Could not find model URL in ZK file: " + ZK_MODEL, e);
  return;
} catch (KeeperException e) {
  log.error("Failed to load model due to ZK exception", e);
} catch (InterruptedException e) {
  log.error("Operation interrupted should never happen", e);
} catch (IOException e) {
  log.error("Failed to load model from " + url, e);
}
log.warn("Clearing current URL due to error");      #C
currentUrl = null;
version = -1;
  }
};
```

#1 get URL from ZK
#A check for changed URL
#2 tell ZK what model we have loaded
#3 update pre-existing file
#B if no such data on ZK
#C only get here on error

A lot of this code exists to handle exceptional conditions but the basic outline is quite simple. The backbone of the code is in the calls to `zk.getData` #1, `zk.create` #2 and `zk.setData` #3. The way that the code works is that it attempts to read the URL of the model from Zookeeper. If a URL is read but there is no current model or if the version of the Zookeeper file containing the URL is different from the one for the current model, then the model is reloaded and the status file for this server is updated with the model URL. The update is done by first trying to create the status file and if that fails to update the status file.

If the original read of the URL failed due to the model file being missing, a message is logged and a relatively normal return ensues. Other errors cause the server to null out the URL of the current model to make sure that we reload the model at the next opportunity.

The actual handling of classification requests is done by a Thrift server. The interface is defined in Thrift IDL as shown here

```
namespace java com.tdunning.ch16.generated
service Classifier {
  list<double> classify(1: string text)
}
```

This interface definition declares that the server will accept classification requests. Each request will return a list of scores, one for each possible value of the target variable.

The interface defined by the IDL above is implemented in the class `Ops`, which is shown in listing 16.6.

Listing 16.6. Implementation of classification service

```
public class Ops implements Classifier.Iface {
private static final int FEATURES = 10000;          #A

private static final FeatureVectorEncoder enc =
        new TextValueEncoder("text");               #B
volatile AbstractVectorClassifier model;            #C

public Ops() {}

@Override                                    #1
public List<Double> classify(String text) throws TException {
 Vector features = new RandomAccessSparseVector(FEATURES);
  enc.addToVector(text, features);          #D
  Vector r = model.classify(features);      #E

  List<Double> rx = Lists.newArrayList();
  for (int i = 0; i < r.size(); i++) {      #F
    rx.add(r.get(i));
  }
 return rx;
}

public void setModel(AbstractVectorClassifier model) {
  this.model = model;                        #2
}
}
```

#1 basic classification method
#A encode compatible with 20news example
#B standard text encoding, also compatible
#C volatile to allow atomic switchover
#D encode text
#E classify text
#F and convert to output format
#2 used by Zookeeper watcher to update model

The `Ops` class can classify a vector versus the currently loaded model using the `classify` method. This method is called by the Thrift server when a client request is received and performs encoding of the

text passed to it in the style of the TrainNewsGroups example program. The model is then used to compute a vector of scores and these scores are returned. The Ops class also defines a setModel method that the Zookeeper watcher class can use to load a new model when a change is observed in Zookeeper.

RUNNING THE CLASSIFICATION SERVER

To see this server in action, we can use the command line interface to Zookeeper to cause instigate behaviors. Before we start, though, we need a model to work with. The TrainNewsGroups is a handy program to get such a model from since it writes a copy of a model to /tmp every few hundred training examples. To get it to create a model, we just need to run it as previously described in Chapter 15. You can use TrainNewsGroups to get a model for the server to use, but first we need to make sure that Mahout has been compiled:

```
$ cd mahout-trunk
$ mvn -DskipTests install
```

Then we can run the TrainNewsGroups example program

```
$ cd /examples
$ export MAVEN_OPTS=-mx600m
$ export TNG="org.apache.mahout.classifier.sgd.TrainNewsGroups"
$ mvn -e exec:java -Dexec.mainClass=$TNG -Dvm.args="-mx400m" \
          -Dexec.args="../../20news-bydate-train/"
```

The result is a mass of output, the critical bits of which are shown here:

```
[INFO] [exec:java {execution: default-cli}]
11314 training files
0.00 0.00    0.00    ... 1   0.000   0.00    none
0.00 0.00    0.00    ... 2   0.000   0.00    none
0.00 0.00    0.00    ... 3   0.000   0.00    none
...
0.33 187471.00      ... 1000        -2.56237.20    none
0.33 187471.00      ... 1200        -2.56237.20    none
…
```

As a side effect we get model files stored in /tmp

```
$ ls -l /tmp/                       #D
-rw-r--r--  1 ...  1680247 Nov 21 01:16 news-group-1000.model
-rw-r--r--  1 ...  1680247 Nov 21 01:16 news-group-1200.model
-rw-r--r--  1 ...  1680247 Nov 21 01:16 news-group-1400.model
-rw-r--r--  1 ...  1680247 Nov 21 01:16 news-group-1500.model
...
-rw-r--r--  1 ...  1680247 Nov 21 01:19 news-group.model
$
```

The files with names like news-group-*.model contain the best model at each stage of learning.

Once we have some model files we can start a local copy of Zookeeper. Following the quick-start guide for Zookeeper, we just have to download and unpack the distribution, set up a single configuration file and start the server. Assuming that you have already downloaded the Zookeeper binary distribution in compressed tar file format from http://hadoop.apache.org/zookeeper/releases.html. You can configure and run a single node Zookeeper cluster like this

```
$ tar zxvf ~/Downloads/zookeeper-3.3.2.tar.gz
$ cd zookeeper-3.3.2/
$ cat > conf/zoo.cfg  <<EOF
tickTime=2000
dataDir=/var/zookeeper
clientPort=2181
EOF
$ sudo bin/zkServer.sh start
...
2010-11-21 01:31:49,947 - INFO  [main:NIOServerCnxn$Factory@143] -
binding to port 0.0.0.0/0.0.0.0:2181
```

With Zookeeper running, you can start the classification server:

```
$ cd ch16
$ mvn compile
...
$ mvn exec:java -Dexec.mainClass="com.tdunning.ch16.Server"
...
10/11/21 01:36:06 INFO zookeeper.ClientCnxn: Socket connection established to
localhost/fe80:0:0:0:0:0:0:1%1:2181, initiating session
```

```
10/11/21 01:36:06 WARN ch16.Server: Starting server on port 7908
...
10/11/21 01:36:07 ERROR ch16.Server: Could not find model URL in ZK file: /model-
service/model-to-serve
10/11/21 01:36:08 WARN ch16.Server: Starting server on port 7908
...
```

This is best done in a separate window from the one used to start Zookeeper so that you can separate the different log lines as the come out.

At this point the classification server is running, but we don't have anything in Zookeeper to tell it what model to load and use to classify requests.

To put the right things into Zookeeper, we can use the Zookeeper command line interface as shown below. Again, a new window is helpful to keep things separated.

```
$ cd zookeeper-3.3.2/
$ bin/zkCli.sh
[zk: 1] ls /
[zookeeper]
[zk: 2] create /model-service ""
Created /model-service
[zk: 3] create /model-service/model-to-serve file://localhost/tmp/news-group-1500.model
Created /model-service/model-to-serve
```

Make sure that you don't split the model path across lines. As we were talking to Zookeeper, the classification server we started earlier was checking every 30 seconds to see if Zookeeper was correctly set up. Each time it did so, it would emit the same error message about not being able to find the model URL. On the cycle after we finally create the `/model-service/model-to-serve` file, however, it will produce a different message:

```
10/11/21 01:45:04 INFO ch16.Server: Loading model from file://localhost/tmp/news-group-
1500.model
10/11/21 01:45:04 INFO ch16.Server: model loaded
```

At this point, the classification server is up and running. We can watch it respond to configuration commands in Zookeeper by changing the content of the model-to-serve file:

```
[zk: 5] set /model-service/model-to-serve file://localhost/tmp/xxx.model
```

Almost instantly after we do this, we will see the server output a warning that it can't find this bogus model. You can put things back right again with this command:

```
[zk: 6] set /model-service/model-to-serve file://localhost/tmp/news-group-1500.model
```

Almost instantly, the classification server will respond with a confirmation that it has loaded the model. If we had 10 model servers running on different machines, they would all respond just as quickly.

ACCESSING THE CLASSIFIER SERVICE

On the client side, matters are even simpler. Listing 16.7 shows how the client can get information from Zookeeper to find out what servers are live and then request one of them to do the desired classification.

Listing 16.7. How to access the sample classification server

```
public class Client {
  public static void main(String[] args) throws TException, IOException,
        InterruptedException, KeeperException {

 ZooKeeper zk = new ZooKeeper("localhost", 2181, null); #A
 List<String> servers =
          zk.getChildren(Server.ZK_CURRENT_SERVERS, false, null);
 if (servers.size() == 0) {
   throw new IllegalStateException("No servers to query");
 }

 int n = new Random().nextInt(servers.size());     #B
 String hostname = servers.get(n);
 Connection c = new Connection(hostname, 7908);   #C
 List<Double> result1 = c.classify("this is some text to
classify");
 System.out.printf("%s\n", result1);
 List<Double> result2 = c.classify("Given that the escrow " +
          "keys are generated 200 at a time on floppies, why\n" +
          "not keep them there rather than creating one huge" +
```

```
            " database that will have to\n" +
            "be guarded better than Fort Knox.");
    System.out.printf("%s\n", result2);
    c.close();
     }
   }
```

#A talk to Zookeeper to get list of live servers
#B pick a server at random
#C send request and print the result

The first thing that the client does is to contact Zookeeper. It then requests a list of the live servers and picks one at random. After that it connects to the randomly selected server and requests the server to classify two bits of text. The output of the client program should look like what this when you run it with some bits omitted here for clarity

```
$ mvn exec:java -Dexec.mainClass="com.tdunning.ch16.Client"
...
[INFO] Preparing exec:java
...
10/11/21 20:15:26 INFO zookeeper.ClientCnxn: Opening socket connection to server
localhost/0:0:0:0:0:0:0:1:2181
...
[0.03964640884739735, 0.07565438894170683, 0.05901603385987076, 0.05114326116258236,
0.050421016895119145, 0.044210065098318506, 0.04114422304836256, 0.05402402583820334,
0.05281506587810628, 0.042271194810117395, 0.07788774620593823, 0.048761793702438536,
0.03998325394587257, 0.03846382850793468, 0.04373069260414459, 0.04980658400349993,
0.04722614673439341, 0.04948828603244314, 0.04477719025270829]
#1
[0.030532890514017766, 0.25984369795811524, 0.03596784998912587, 0.05321219256232682,
0.02395166487009193, 0.033935125988576176, 0.03729692334981463, 0.04035766767461885,
0.05067830734673832, 0.03391626207189733, 0.07418249664999789, 0.06666407060825316,
 0.0424634477415305, 0.01074681122380413, 0.05212145428368258, 0.030600556819694907,
0.025032370110624636, 0.02760955132545441, 0.02696366801467259]
[INFO]
 ------------------------------------------------------------------------
[INFO]
 BUILD SUCCESSFUL
...
$
```

#1 second request

The output from the two requests appears as a list of numbers, each of which is the score of a different news group. The second request is the more interesting one because it has text that was extracted from an actual newsgroup posting. That request also shows a significantly higher score for the second newsgroup as you would expect when giving a classifier some data it can make a decision about.

Clients written like this example inherently balance load across multiple servers. Moreover, a slightly more advanced client that is written to perform multiple queries in sequence can maintain a watch on the current servers directory on Zookeeper. Whenever the watch is triggered, the client can pick an alternative server. This will cause immediate rebalancing of load as soon as any new server comes on-line or shortly after a live server exits or crashes.

16.5 Summary

Congratulations! As you complete this chapter you have successfully navigated the third and final stage in Mahout classification, deploying a large-scale classifier. Now you know how to build and train models for a classification system, how to evaluate and fine-tune your models to improve performance and how to deploy the trained classifier in huge systems. In deployment, you have seen the importance of fully scoping out the project, paying particular attention to any parts of your system that as designed could exceed what Mahout can easily do. Discovering these potential difficulties ahead of time gives you the chance to modify your design to avoid the problems or to budget time to stretch those boundaries.

Often the most intense aspect of deployment of a large-scale system is building the training pipeline. In systems of the size you will tackle with Mahout, the pipeline may offer some substantial engineering challenges. One key to doing this successfully is what you have learned about doing extremely large-scale joins to implement the pipeline in parallel.

The chapter also showed you some of the pitfalls to avoid, namely, how not to produce target leaks or create a semantic mismatch.

Finally, you have seen the core design of a robust classification service based on standard technologies like Zookeeper and Thrift. If you keep the spirit of this service design, you should be able to deploy high performance classification services reliably and repeatably. Don't be fooled by the simplicity of the design presented in this chapter. Much of the simplicity, robustness, and reliability of this system is provided by building on the firm foundation of Zookeeper. Reflect carefully before you diverge too far from this basic design idea.

As you saw, deployment of even a well-designed classifier is a complicated task when it is integrated into a larger system. The exploration of classification is concluded in the next chapter, which puts all of this together in presenting a real world case study of a large-scale classification system used by an online marketing company, ShopItToMe.

17

Case study: Shop It To Me

This chapter covers

- Speed and scale considerations of the classifier system
- How the training pipeline was constructed
- Restructuring classification for very high throughput

Up to here, the chapters on classification have presented an overview of classification plus detailed explanations of what it takes to design and train a Mahout classifier, evaluate the trained model(s) in order to adjust performance to the desired level, and deploy the classifier into large-scale systems. The current chapter puts all those topics into practice in a case-study drawn from a real-world online marketing company, called Shop It To Me,[20] that selected Mahout as their approach to classification. You will examine the issues their small engineering team faced in building and deploying a high performance Mahout-based classifier and find out the solutions they came up with.

Chapter 16 focused on the scale requirements of the very large systems that are best served by Mahout classification. Similarly, the case study presented in this chapter deals with large data sets, but it also affords a look at a system with speed requirements that are extreme, even for Mahout systems. Because of the scale and especially the speed required, some of the solutions the team designed required substantial innovations. Overall, the system described in this chapter shows how you can push Mahout classifiers farther than it appears possible at first glance.

After introducing the Shop It To Me system, this chapter explains generally how the outbound email system works and describes the data that system produces. The goal of the case study classifier is presented along with the steps taken in training the model. Then this chapter examines how the Shop It To Me team made the training and model computations fast enough to meet their business requirements. Finally, the lessons to be learned from this case study are brought together so that you can apply what you learn to projects of your own.

17.1 Introducing to Shop It To Me

In order for you to appreciate the challenges described in the case study, it is helpful to have some background information about the company. This section will answer the questions of what Shop It To Me does, what they wanted a classification system to do and why they chose Mahout to build that system. First we describe the company and their mission.

Shop It To Me is a free online personal shopper service that provides email notifications of sales for merchandise of interest to their customers. Their mission is to effectively connect a shopper with a list of sale items that are appropriate to the tastes of the customer and to do so in a timely manner. The latter is particularly important because the business often deals with products that are temporarily discounted, and

[20] http://www.shopittome.com/

if options to click are not delivered promptly to the customers, the sale items will disappear before the shopper can shop. In short, the business mission of the company is to give people what they want to see.

Shop It To Me has relationships with hundreds of retailers that allow it to know when items will go on sale. With over 3 million customers, Shop It To Me makes over 2 billion product recommendations per month in order to construct the SaleMails that they send out to the customers.

17.1.1 Why Shop It To Me chose classification

You may be wondering "why do classification?" if the business mission is essentially based on a form of recommendation, that is, predicting which sale items particular customers will want to see. The reason is that the Shop It To Me situation is a case where normal recommendation methods do not work well. Recommendation systems work well with items that are seen by many users. Classic examples include movies and music. People can be seen as related to each other by noting that they interact similarly with the same items and that similarity can be used to predict other preferences. Most importantly, the items being recommended will still be there tomorrow.

This persistence is not true for the products being recommended customers by Shop It To Me. The best sales items are, literally, here today and often gone tomorrow (if not later today). The sale items may go away faster than feedback on who is clicking on the item can even be obtained much less used productively. This ephemeral nature of items means that the recommendation system has to take into account the characteristics of items in the training data such as brand, price or color that will be shared with other items that appear later. To do this requires a different approach: the use of classification to make predictions that operate as part of a system that acts like a recommendation system. At Shop It To Me, available data include personal history of the users and characteristics of the items to be recommended. These form the predictors. There is also a good target variable as well: whether the user will click on an item.

In order to increase the utility of these recommendations, Shop It To Me has built an advanced classification system to help predict exactly which products will appeal to particular users. When constructing a SaleMail for a particular user, this classification system can compute the value of a classification model trained to predict whether this user will click on an item for each of hundreds of thousands of possible sale items. By ranking items according to the predicted probability of user interest, Shop It To Me can build SaleMails of particular interest to users.

17.1.2 Why use Mahout for the Shop It To Me classifier?

Building this system posed some particular challenges. Not only are there billions of training examples, but classifications must be done extremely fast. A back-of-the-envelope calculation shows that millions of users need model evaluations for hundreds of thousands of items giving hundreds of billions of evaluations. In order to meet the constraints of the email construction and delivery pipeline, these evaluations must be completed in just a few hours. This gives a raw classification rate requirement of tens of millions of classifications per second which is, as they say in the old country, a lot.

To meet these severe requirements with a reasonably-sized computing infrastructure, Shop It To Me has had to do some very interesting engineering. Several systems were used to produce prototype classifiers including R and Vowpal Wabbit (an open source SGD classifier from Yahoo!). With R, simply handling the training data volumes proved very difficult and good integration with the existing Ruby / JRuby infrastructure at Shop It To Me proved very difficult. Vowpal Wabbit was able to handle the data volumes for training, but integrating a Vowpal Wabbit classifier proved difficult. Mahout, on the other hand, was able to handle the training volume and integrated well with the existing infrastructure. Mahout provides key portions of the Shop It To Me system, aspects of which are described here.

> Note: Some details of system design, variable extraction and model performance are proprietary and have been omitted or glossed over.

The system outline here is broadly descriptive and does not contain an exact representation of all aspects of the Shop It To Me system. Enough details are included, however, to allow you to understand the scaling issues and solutions. Those issues start with the structure of the email system itself.

17.2 General structure of the email marketing system

The overall data flow at Shop It To Me is roughly as shown the path shown in Figure 17.1. This diagram shows how registration information is used to fill a user table as well as a brand preference table. It also shows how specialized data importer is used to find sale items from retail partners. The core of the system is the SaleMail constructor, which accesses tables containing information about current sale items, users and their expressed interests. Models including a click prediction model are used to select which items are to be included in each user's email. The items included in each email are recorded in a table of item appearances.

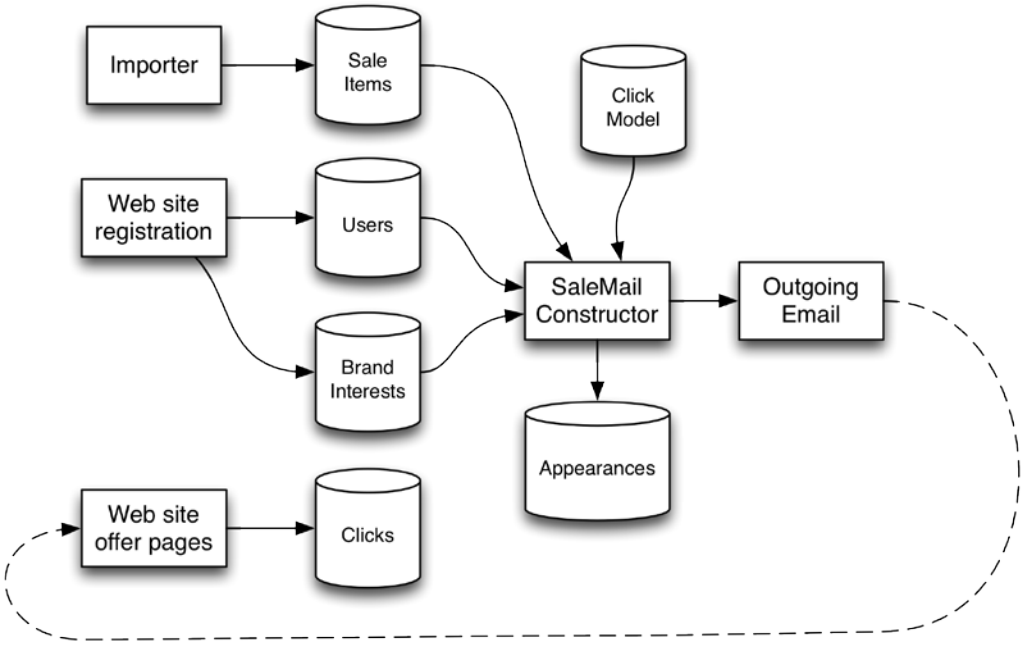


Figure 17.1 The overall data flow for the Shop It To Me email marketing system. The dashed line illustrates how users provide feedback to the system by clicking on items in their emails

As Figure 17.1 shows, users and items for sale form the data that make up the predictors for the click model used by the email generator. When users click on items in the emails that they receive, they cause web page visits to be recorded in log files. These recorded clicks are then turned into training data for the next round of modeling so that the click model can be updated. This updated model is then used for the next round of recommendations and the cycle repeats. Some of these data sets are stored in relational databases, such as the table of users, but other data sets, such as the appearances table, are stored either in large collections of log files or are stored implicitly and must be reconstructed when needed.

Figure 17.2 shows a simplified structure of the tables in the Shop It To Me system used to record the required user and item data along with the relevant actions needed to build the click model. ☐

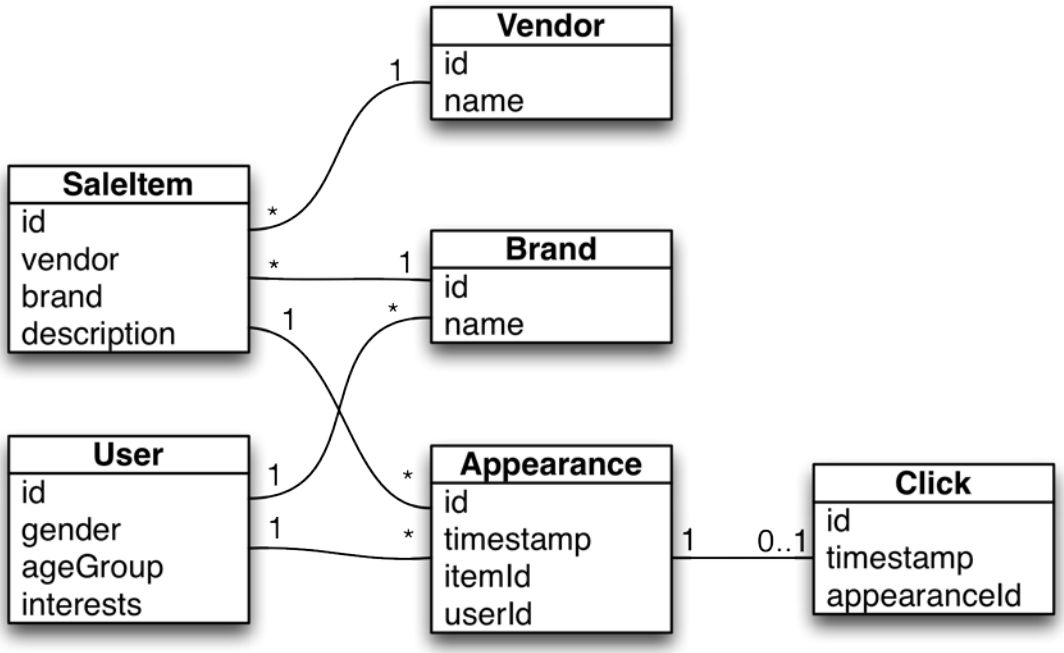


Figure 17.2 Simplified UML view of database tables used to support the data flow at Shop It To Me. Each sale item has a unique vendor and brand, but the item can appear many times. Likewise, a user may indicate interest in multiple brands, but the user will see many items presented to them. Each appearance may result in at most one click.

In Figure 17.2, denormalized views of the `User` and `SaleItem` tables are exported as Apache Hadoop files for use in creating the training data. The `Appearance` table is not actually a database table due to size. Instead it is kept in conventional files that contain logs of emails sent out. The `Appearance` and `Click` tables are both exported to Hadoop for joining. Note how these tables are kept in a highly normalized form. This is important for transactional processing and for supporting web access, but this normalization must be removed for training.

Now that you have a framework for understanding the mission of Shop It To Me and how their marketing process is structured, you are ready to tackle the challenge of designing a classification system. The first stage for Shop It To Me, as in any classification project, is to train a model.

17.3 Training the model

Training a model first requires preliminary planning. You must consider how to pose the questions being asked in order to meet your goals and what the target variables will be the output of your system. You must examine the available historical data, see what features are available for use as variables, and determine which of these will be most effective as predictor variables. For a Mahout classifier, it is particularly important to scope the proposed project for "pain points" in order to best determine how to balance speed and size requirements as you have seen in previous chapters. In this section, you will see how Shop It To Me accomplished these steps.

17.3.1 Defining the goal of the classification project

The Shop It To Me engineers approached their problem in a similar way as we outlined in previous chapters. The goal of their classification project was to produce a model that could predict whether a user would click on an item using the historical data on item appearance and the user's clicking history as training data. Predicting clicks is the target and user and item characteristics are the predictors. To be usable, however, the Shop It To Me data required significant preprocessing.

The process of creating training data is shown in Figure 17.3. This figure shows how user data is preprocessed and joined with item data to form the training data used in model training.

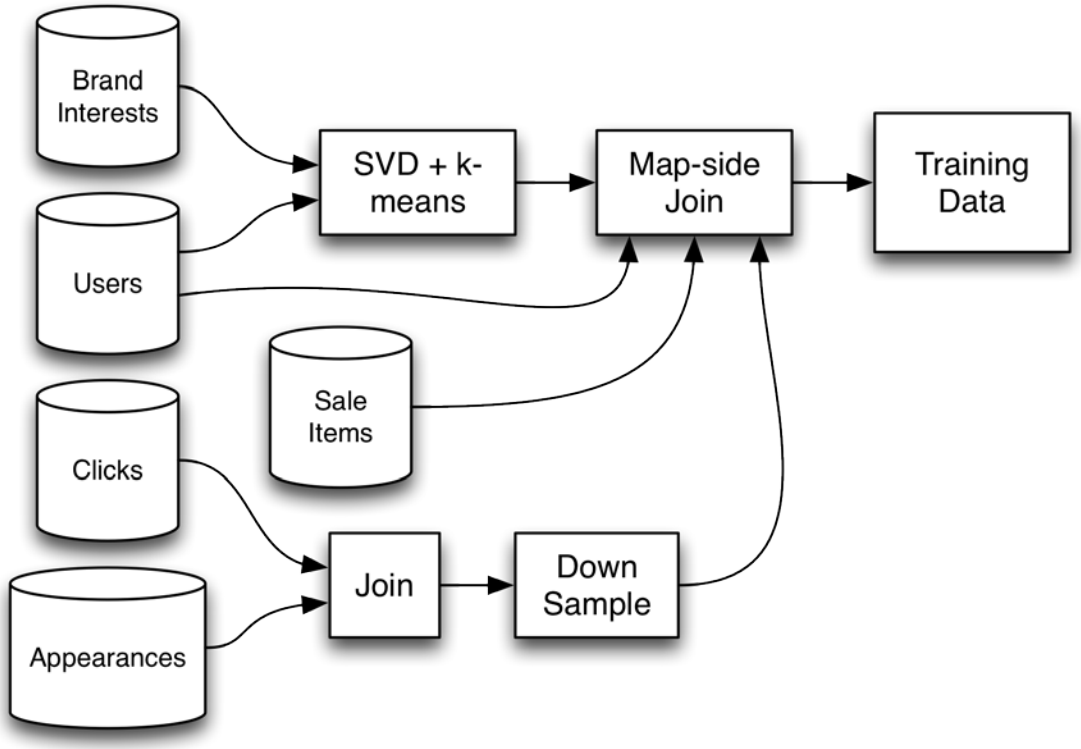


Figure 17.3 Data flow during the creation of model training data. User brand interests are used to form user clusters. In parallel, clicks and appearances are joined and selectively down-sampled. All of the data are then joined to produce the training data.

Some of the richest information available in the Shop It To Me system is the brand preference database. When users register with the Shop It To Me system, they are asked which brands they like. Their answers are stored in the brand preference database and are an excellent source of information about what users do or do not like. Unfortunately, these data are not in the form that would be usable for classification.

To rectify this, the data are first reduced using an SVD decomposition and then clustered using the k-means clustering algorithm. The SVD decomposition is a mathematical technique often used to reduce the dimensionality of observed data. This results in the assignment of all users to clusters of people who have expressed similar brand preferences. The brand preference cluster id expresses the original brand preference information in a form very suitable for modeling.

The `Appearances` table is the largest table in the entire computation, and each record in the final training data can be traced back to a unique row in the `Appearances` table. Moreover, the rows of the `Appearances` table are downsampled at different rates depending on whether or not the appearance of the item resulted in a click. Unclicked appearances are down-sampled substantially while clicked appearances are not down-sampled at all. In order to make downstream joins handle less data and thus be more efficient, clicks and appearances are joined in a full reduce join and down-sampled in the reducer before joining against user and item meta-data.

Click and appearance data are then joined against user meta-data including gender, age, general geographic location and brand preference cluster as well as against sale item meta-data including item ID, color, size and description. This second join is done using a map-side join because the user and item meta-data tables contain only a few million entries and can be kept in memory relatively easily.

This final join could alternatively have been made into a reduce-side join and integrated into the click and appearance join step. It is possible the performance of this alternative could have been improved with more experimentation.

17.3.2 Partition by time

Accumulated training data are partitioned by day to allow flexibility in terms of how much data is used for training and, most importantly, to allow test data to come from disjoint and later times than the training data. This gives a more realistic appraisal of performance because it includes real-world effects like item inventory changes. Daily aggregation also helps when it comes time to expunge old training data that is no longer used in modeling.

Time partitioning also allows training data to be created incrementally, one day at a time. This means that the large cost of building a long-term training set can be paid over a long period of time rather than delaying the build process each day.

17.3.3 Avoiding target leaks

Through the chapters on classification we have emphasized the importance of careful planning for how to do feature extraction to avoid creating target leaks. In order to avoid problems with target leaks, Shop It To Me engineers used a multi-layered segregation of data based on time. The SVD and k-means clustering of the brand preference information were based an early time period and were kept relatively stable as models were rebuilt. The resulting cluster model was applied to data from a later time period to produce the actual training data. Additional, even more recent, data were held back from the training algorithm to assess the accuracy of the trained model.

17.3.4 Learning algorithm tweaks

The default behavior of the Mahout learning algorithms was not entirely suited to the task that Shop It To Me needed to do, partly because of the way the classifier was being used as a recommender and partly because the model was difficult to train. These difficulties led the engineers there to extend the default Mahout logistic regression learning algorithm in several ways.

OPTIMIZING BASED ON PER-USER AUC

The `AdaptiveLogisticRegression` learning algorithm uses AUC as a default figure of merit to tune the hyper-parameters for binary models. In the Shop It To Me classification system, AUC is an accurate measure of how well the model predicts which user-item pairs are likely to result in a click. Unfortunately, one of the strongest signals in click-through models involves the classification of users into those who click and those who do not. Since the fundamental problem in email marketing is to rank items for each user, predicting who will click is not of much interest. What matters is the prediction of which items the user will click on if they are shown to the user. In order to help the learning algorithm find models that solve the right problem, a different figure of merit than normal was required.

In Mahout, there are two classes that can be used to compute different forms of AUC. Normal, global AUC is computed by the class `GlobalAuc` while AUC based on ranking within groups is computed using `GroupedOnlineAuc`. The `AdaptiveLogisticRegression` learning algorithm allows alternative AUC implementations and a grouping criterion to be supplied. Grouping on user or user cluster allows models that are good recommenders to be distinguished from models that simply pick users who click.

MIXED RANK AND SCORE LEARNING

Whether learning a probability directly is better than learning an ordering of results is a major current area of research. These approaches seem equivalent at first examination, but several researchers have noted that there are differences in practical situations. One recent advance is represented by the paper “Combined Ranking and Regression” by Google researcher D. Sculley. Shop It To Me engineers implemented this technique in a wrapper around Mahout code by maintaining a short history of training examples and using either the current training example or the difference between the current example and a previous one to compute the gradient for updating the model. Mahout has a largely untested implementation of this technique in the class `MixedGradient`. This technique was quite effective for Shop It To Me, but more experience is necessary to determine how often this will be true.

17.3.5 Feature vector encoding

Figure 17.4 shows a record from an idealized set of training data that might be used to build a click model. In this model, as in more realistic versions of such a record, there are three types of fields that are of interest:

- User-specific fields
- Item-specific fields
- User-item interaction fields

The user-specific fields include values for variables such as the age group, gender, and brand preference cluster. The item-specific fields include values for variables such as the vendor, brand, and item description. Of greatest benefit are the user-item interaction variables. The interaction variables enable the model to be more than a record of what items or kinds of item are most popular.

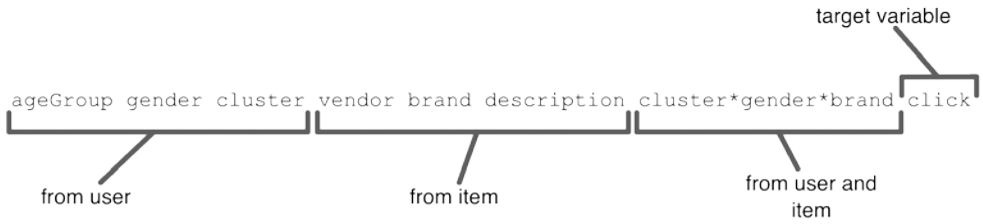


Figure 17.4 Fields in a possible training record for simplified email marketing click-through prediction model. The record contains user specific, item specific and user-item interaction variables. Real models of this sort have many more fields than shown in this conceptual example.

The interesting item here relative to our previous examples of feature encoding is the interaction variable that involves the user's brand preference cluster ID (cluster in the diagram), gender and item brand.

To encode an interaction variable using the feature hashing style recommended for SGD learning, the code needs to pick feature vector locations are that are independent of the feature vector locations used for each of the constituent features in the interaction. For example, figure 17.5 shows how we would like things to work. Assuming we have a sparse feature vector of size 100, the locations for the feature "gender = female" might be assigned locations 8 and 21 by the hashed feature encoding. Similarly, "item = dress" might be assigned locations 77 and 87 in the vector. The interaction "female x dress" needs to be assigned locations that are as statistically independent from these four locations as possible, while still being deterministically computed based on these original values.

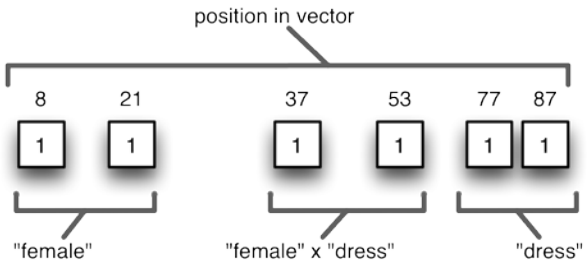


Figure 17.5 Interaction features need to be hashed to locations different from the locations of the component features.

One way that Shop It To Me engineers did this was to pick random seeds to use to combine the locations of each of the two locations. The seeds generated by the random number generator for this example were 22 and 92. To get the *i*th location for the interaction feature, use `hash[i] = seed[0] * x[i] + seed[1] * y[i]` where `x` and `y` contain the feature locations for "woman" and "dress" respectively.

This method is closely related to double hashing as used in Bloom filters and can be shown to provide reasonably good, independent locations for the composite features. Listing 17.1 provides a sample encoder with one implementation of an interaction encoder that is based on this idea.

Listing 17.1 Sample encoder showing one implementation of an interaction encoder

```
public class CategoryInteractionEncoder {
  private int[] seeds;                              #1
  private CategoryFeatureEncoder[] encoders;    #2
  private int probes = 2;
  CategoryInteractionEncoder(int seed, CategoryFeatureEncoder... enc) {
    Random r = new Random(seed);
    seeds = new int[enc.length];
    for (int i = 0; i < enc.length; i++) {
      seeds[i] = r.nextInt();                       #3
    }
    this.encoders = enc;
  }
  public void addToVector(int[] categories, double weight, Vector data) {
    int[] hashes = new int[categories.length];
    for (int i = 0; i < probes; i++) {                    #A
      for (int j = 0; j < categories.length; j++) {
        hashes[i] += seeds[j] *
              encoders[j].hashForProbe(categories[j], i);   #4
      }
    }
    int n = data.size();                        #B
    for (int h : hashes) {
      h = h % n;
      if (h < 0) {
        h += n;
      }
      data.setQuick(h, data.getQuick(h) + weight);
    }
  }
  ...
}
```

#1 seeds combine hashes
#2 encoders encode constituent variables
#3 initialize seeds deterministically
#A combine hashes
#4 combine hashes of each feature
#B use combined hashes to set values

The seeds and encoders are kept around in the interaction encoder. The seeds are initialized in the constructor, which is where array of encoders is passed in. In the `addToVector()` method, hashes from the encoders for each variable in the interaction are combined with integer multipliers that were randomly chosen when the interaction encoder was constructed based on the seed supplied at that time. This multiplicative combination ensures that none of the feature locations chosen for the interaction encoder are likely to collide with any of the locations chosen for the constituent variables.

There are other possible approaches to interaction encoding. For instance, two sums of the locations of the first and second hashed locations for all constituent encoders could be kept and then combined using the standard double-hashing algorithm. The following code shows how this alternative encoding approach could work:

```
public class CategoryInteractionDoubleHashingEncoder {
  private CategoryFeatureEncoder[] encoders;    #1
  private int probes = 2;
  CategoryInteractionEncoder(CategoryFeatureEncoder... enc) {
    this.encoders = enc;
  }
  public void addToVector(int[] categories, double weight, Vector data) {
    int h0 = 0, h1 = 0;
    for (int j = 0; j < categories.length; j++) {
      h0 += encoders[j].hashForProbe(categories[j], 0);   #2
      h1 += encoders[j].hashForProbe(categories[j], 1);   #2
    }
    int n = data.size();
```

```
      for (int i = 0; i < probes; i++) {
        h = h0 + i * h1;                              #3
        if (h < 0) {
          h += n;
        }
        data.setQuick(h, data.getQuick(h) + weight);
      }
    }
    ...
  }
```

#1 Encoders encode constituent variables
#2 Computes exactly two base hashes
#3 Uses combined hashes to set values

Here there are encoders for each constituent variable just as before, but only the sums of the first and second probes for each constituent variable are used. When computing the locations for the interaction, combine these two sums using double hashing.

Neither of these approaches has been rigorously evaluated, but both seem to have worked well in practice. Ironically, to use a classifier for recommendations, interaction variables between user and item are required, but having these interactions increases how much work it is to encode variables and compute classification results. Since the Shop It To Me classification throughput requirement is so high, that is a big deal and has to be addressed by speedups in the classification system. These speedups avoid both the encoding cost and the compute cost of the classification computation itself.

17.4 Speeding up classification

As mentioned before, Shop It To Me had a firm requirement for a prodigious rate for performing classifications in their final system. In order to perform the required tens of millions of classifications per second on the available few machines, several improvements were needed over and above what base Mahout provides. Since the internal feature vector used has a length of over 100,000 elements, meeting this classification rate in a naive way would require a raw numerical throughput of several trillion floating point operations per second even without counting feature encoding. With only a few dozen CPU cores available for the task, this is not plausible as stated.

To meet these requirements, Shop It To Me engineers made improvements that included

- caching of partially-encoded feature vectors,
- caching of hash locations within encoders,
- splitting the computation of the model into pieces that can be separately cached.

These improvements ultimately reduced the run-time model computation to three lookups and two additions. The necessary pre-computation involved one encoding and model computation per user and a few hundred such computations per item. The overall savings due to these changes are about 3-4 orders of magnitude, which makes the overall model computation quite feasible. The improvements listed here may look relatively simple, but they provide a powerful extension of the way Mahout can be used for classification, and you may want to consider trying innovations such as these in your projects if your required throughput is as high as Shop It To Me's is.

17.4.1 Linear combination of feature vectors

The feature extraction system used in Mahout works by adding the vectors corresponding to separate features in order to get an overall feature vector. In the case of Shop It To Me, the feature vector can be divided into three parts related to the user, the item part, and a part that involves both the user and item.

$$x = x_{user} + x_{(cluster\ +\ ageGroup)} \times itemBrand + x_{item}$$

This division into parts can clearly help with feature encoding costs because we can cache these components instead of computing them on the fly each time we want to evaluate the model.

Caching the entire feature vector does not work well because of the very large number of values that would have to be cached. Caching the pieces works because there the number of unique sub-parts is much smaller. Essentially, the larger cache has a required size that is nearly the product of the sizes of the smaller caches.

Figure 17.6 shows how the three parts of the feature extraction are used in the computation of the model score.

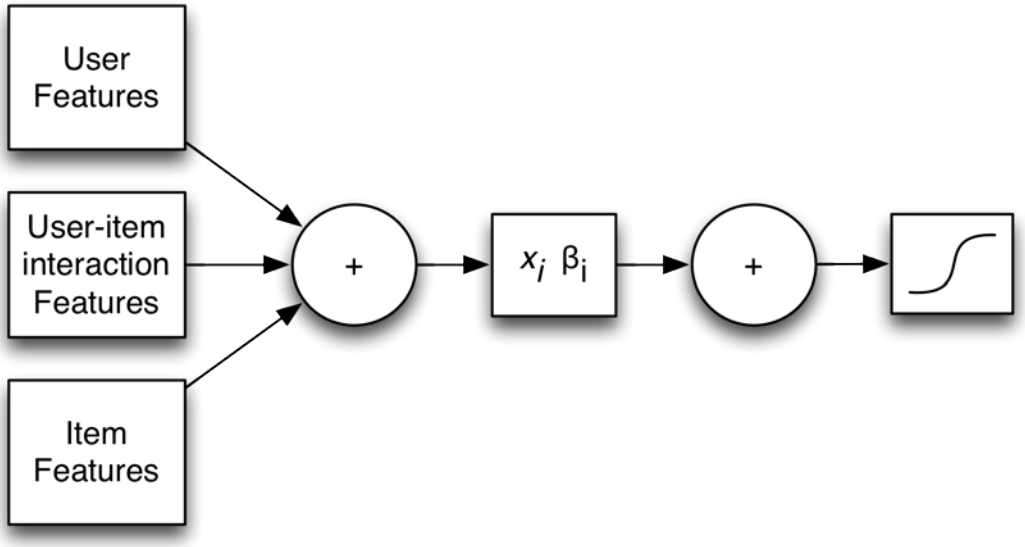


Figure 17.6 How model scoring works for Mahout SGD logistic regression model

Using caching for these three components helped with speed somewhat, but, by itself, this caching did not improve speed enough. It did lay the groundwork for additional improvements.

17.4.2 Linear expansion of model score

Splitting the feature vector into three pieces is useful for speeding up feature encoding, but the model computation that uses the feature vector is composed of linear operations all the way to the last step. This allows even more aggressive caching to be done. Moreover, the items cached at the later point in the computation are much smaller.

The way that logistic regression models such as the SGD models in Mahout work is that they produce a weighted sum of the features and then transform that weighted sum so that it is always in the range from 0 to 1. When ranking values, you can omit that last transformation without disturbing the ordering of the final results. In Mahout, the `classifyScalarNoLink(Vector)` method in `AbstractVectorClassifier` omits this final transformation exactly as desired.

Once this final transformation is removed and the model is simplified to a weighted sum, you can use the linear combination property of feature vectors. Using dot product notation, the model output z can be written this way:

$$z = \beta \cdot x$$

Here β represents the weights the model applies to each feature and x represents the features that are given to the model. Both quantities are vector-valued. The multiplication here is a dot product, which indicates the sum of the products of corresponding elements of β and x. The dot product is where all of the intensive computation happens. Note that this formula can be broken into three parts just as the feature vector can be split:

$$z = \beta \cdot x_{user} + \beta \cdot x_{(cluster + ageGroup) \times itemBrand} + \beta \cdot x_{item}$$

This split can be depicted by changing the previous diagram, which showed how the model computation works in Mahout's SGD models, such that more computations are done separately before being finally combined.

Figure 17.6 shows user, item, and user-item features being applied separately to the weight vector β before the three components of the score are combined. The clever thing about this is that these three parts can now be efficiently cached.

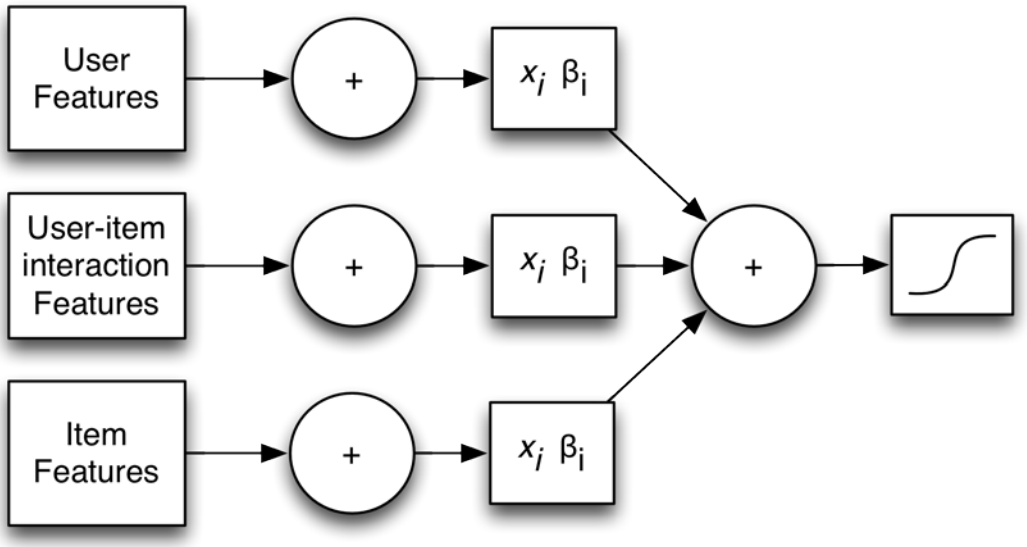


Figure 17.6 How model scoring can be rearranged to allow pre-computation and caching of intermediate results.

Importantly, the value of each part is a single floating-point number rather than a vector. In the case of a full cache hit, the cost of a single model computation for a particular user and item computation drops from several thousand floating point operations to just two table lookups and two additions. There are only two lookups because the user component of the score is computed outside the inner loop. Caching smaller objects also decreases memory consumption, allowing more items to be cached.

How this works in code is shown in listing 17.2.

Listing 17.2 Using Caching and Partial Model Evaluation to Speed Up Item Selection

```
public class ModelEvaluator {
  private OnlineLogisticRegression model;                          #1
  private List<Item> items = Lists.newArrayList();                 #1
  private Map<Item, Double> itemCache = Maps.newHashMap();
  private Map<Long, Double> interactionCache = Maps.newHashMap();
  private FeatureEncoder encoder = new FeatureEncoder();
  public List<ScoredItem> topItems(User u, int limit) {
    Vector userVector =
      new RandomAccessSparseVector(model.numFeatures());
    encoder.addUserFeatures(u, userVector);                                #2
    double userScore = model.classifyScalarNoLink(userVector);
    PriorityQueue<ScoredItem> r = new PriorityQueue<ScoredItem>();     #3
    for (Item item : items) {
      Double itemScore = itemCache.get(item);                          #D
      if (itemScore == null) {                                         #4
        Vector v = new RandomAccessSparseVector(model.numFeatures());
        encoder.addItemFeatures(item, v);
        itemScore = model.classifyScalarNoLink(v);
        itemCache.put(item, itemScore);
      }
      long code = encoder.interactionHash(u, item);                    #5
      Double interactionScore = interactionCache.get(code);
      if (interactionScore == null) {
        Vector v = new RandomAccessSparseVector(model.numFeatures());
        encoder.addInteractions(u, item, v);
        interactionScore = model.classifyScalarNoLink(v);
        interactionCache.put(code, interactionScore);
      }
      double score = userScore + itemScore + interactionScore;         #6
```

```
        r.add(new ScoredItem(score, item));
        while (r.size() > limit) {
          r.poll();
        }
      }
      return Lists.newArrayList(r);
    }
  }
```

#1 Must be injected by other code
#2 Scores user only once
#3 Accumulates best priority queue
#D Looks for cached item score
#4 If not found, computes it
#5 Looks for cached interaction effect
#6 Combines the components

This class shows simplified code that does the model evaluation using caches for item and interaction effects. Assume for now in this code that the model and the list of items will be populated by some other means.

What this class does when given a user is to first compute the score component for that user. Then it scans the list of all items and computes the model score for each item. The items with the top score are retained. The score is computed in three parts. The part due to the user variables is computed outside the item loop and retained. Then Inside the inner loop the item and user-item interaction parts are each computed if they are not found in the appropriate cache data structure. The `FeatureEncoder` class is a convenience class that is used to wrap around the feature encoders for the various user and item features and their interactions. Finally the components of the score are combined #6 to get the final score.

Clearly it is not as simple to split up the model computation in this way, but in some cases it is worthwhile due to the massive speedups available.

17.5 Summary

The online marketing case study presented in this chapter describes the innovative approaches used by Shop It To Me engineers in developing a Mahout classifier for predicting clicks for on-sale items presented to customers. The scale and the speed that Shop It To Me were able to achieve using Mahout classifiers both in training and when running a deployed model against live data illustrate how Mahout can be used in very large-scale problems.

The lessons learned from this real world classifier include seeing that classification can be a useful way to achieve results similar to recommendation in systems that preclude traditional recommendation approaches. Other key ideas are the innovations in the Shop It To Me system are the use of ranking objectives for learning models and the use of grouped AUC for meta-learning. Learning ranking and regression together appears to be a very powerful technique for nearly all applications of stochastic gradient descent learning. Grouped AUC as a target for the `AdaptiveLogisticRegression` to optimize learning meta-parameters looks like a very effective strategy whenever classifiers are used as recommenders.

In terms of throughput, the big lesson was the way that caching strategies were used to speed up the Mahout classifier. These methods included caching of partially encoded feature vectors, caching of hash locations within encoders, and splitting the computation of the model into pieces that can be separately cached. Even with the innovations in the Shop It To Me approach to using Mahout for classification, this is not the end of the line by any means. There are considerable degrees of improvement in training speed still available, and it is unlikely that the last ounce of speed has been milked from the deployed models.

Just how far Mahout can go is a subject for the next chapter. And that is the chapter you get to write.

A JVM Tuning

This appendix covers

- Tuning the JVM for recommender engines

A.1 Tuning for recommender engines

When using a data set of substantial size -- perhaps 10 million preferences and up -- it will be well worth tuning the JVM settings for performance. It is mostly the heap-related (memory-related) settings that are important. Optimal settings depend on the operating system, resources available, architecture, and JVM, but the following guidelines establish a good starting point. These flags are valid on all known Java 6 JVMs, which Mahout requires, with the exception of `-XX:NewRatio`, explained below.

Table A.1 Key JVM tuning parameters for recommender engines

`-Xmx`	This selects the maximum size that Java will allow the heap to grow to. For example, `-Xmx512m` indicates that the heap may grow to 512MB.
`-server`	Modern JVMs have two important flags, `-client` and `-server`, which select JVM settings broadly appropriate for short-running, small-footprint client-side applications, and long-running, resource-intensive server applications. The default depends on the JVM and environment. Obviously, `-server` is appropriate here.
`-d32` and `-d64`	These request 32-bit and 64-bit mode, respectively. On a 64-bit machine, both are possible. While it's typically best to let the JVM decide, selecting 32-bit mode will decrease memory requirements. Of course, 32-bit mode makes it impossible to use more than 2-3 gigabytes of heap (depends on the JVM), but, when pushing this boundary, a small memory savings may be worthwhile. Selecting 32-bit mode on a 64-bit machine may incur a small performance penalty.
`-XX:+NewRatio=`	Some heap space is always reserved for short-lived, temporary objects, and can never be used by more permanent data structures. Mahout runs lean: it creates few short-lived objects, but consumes a lot of heap space with long-lived objects. The default proportion of the heap reserved for short-lived objects is generally too large and ends up being wasted needlessly. A setting of 12, for example, requests that only 1/12th of the heap be reserved

	for short-lived objects. Note that this setting appears to be specific to Sun JVMs only.
`-XX:+UseParallelGC` and `-XX:+UseParallelOldGC`	This instructs the JVM to make better use of multiple processors or cores (common on even desktop computers today) by running garbage collection in parallel.

To demonstrate the effect of these JVM settings, we will experiment with the code found in listing 4.2. Let's start with a 32-bit client JVM defaults: `-client -d32 -Xmx512m`. These are JVM flags, not program flags, so they go before the program class name. On one modern 64-bit computer we used for testing, the load evaluation result indicates that it is producing recommendation in about 425ms, and consuming 248MB of heap in steady state. (Your timing results may vary, as may your exact heap size.)

What if we substitute `-server` for `-client`? If anything, it should increase performance. Memory requirements stay the same, but recommendation time drops to 192ms. This shows how much more appropriate the server JVM tuning is for an application like this.

Now, change `-d32` to `-d64`. You will encounter an `OutOfMemoryError` again. In fact, you'll probably need to allocate 768MB with `-Xmx768m` to get it to run again. Recommendation time drops again to 142ms. Not surprisingly, 64-bit mode is more efficient on a 64-bit machine. Steady-state memory requirements are about the same though: 256MB. 64-bit mode increases the memory consumed by objects and references, but by design, relatively few long-lived objects are created by Mahout's recommender engine to begin with.

You may have noticed a discrepancy: why, if steady-state memory requirements are only about 256MB, did the JVM fail due to lack of heap space when it had 512MB available? Memory requirements peak at a higher level during construction of the in-memory data representation in the `DataModel`.

You should find that with the `-XX:+NewRatio=12` flag allows you to reduce heap to about 640MB.

Finally, try adding `-XX:+UseParallelGC -XX:+UseParallelOldGC`. Assuming more than one processor core is available, this will allow garbage collection work to proceed in parallel with the main computation. On our test machine, recommendation time drops to 126ms. Compare that to the initial result of 425ms.

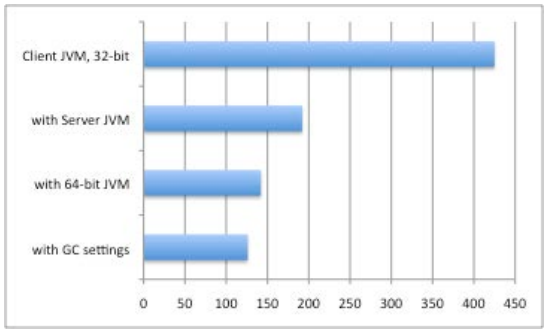


Figure A.1 Progressively better per-recommendation running times (in milliseconds) with various JVM tuning settings

Figure A.1 shows how, on a modern machine, improper JVM configuration can slow down performance by about 350%! Any serious deployment of a recommender engine must be properly tuned.

B
Mahout math

This appendix covers

- Vectors in Mahout
- Matrices in Mahout

Many algorithms implemented by Mahout rely heavily on vector and matrix math. Mahout has its own self-contained vector and matrix math library in the "math" module, which can be reused separately from Mahout itself. It is actually an adapted form of CERN's Colt library (http://dsd.lbl.gov/~hoschek/colt/). This appendix provides a practical overview of key parts of this math module that users encounter frequently while using Mahout: `Vector` and `Matrix` instances, and their associated operations. It is not exhaustive documentation; the project javadoc and source code can provide more detail for the interested.

B.1 Vectors

"Vector" means slightly different things in different contexts, though all of them are similar. Most people encounter a vector first in physics, where it denotes a direction, and is typically illustrated by an arrow. For our purposes, we will sometimes use vectors to represent *points* instead. This is not such a different idea; a point merely corresponds to a vector from the origin (zero point) to that point.

Most often in machine learning and Mahout, we will use vectors in a more abstract sense, one that doesn't necessarily map to a geometric interpretation. A vector can be merely a tuple, an ordered list of numbers. It has a size (or cardinality), and at each index (position) from 0 to size-1 the vector has some numeric value. Vectors are usually written like so: (2.3, 1.55, 0.0). This is a vector of size 3, whose value at index 0 is 2.3. In machine learning, we will sometimes deal with vectors with sizes in the millions.

B.1.1 Vector implementation

`Vector` in `org.apache.mahout.math` is the interface that all implementations implement. There are several. Representing vectors in a memory-efficient way, and accessing their values efficiently, are quite important to Mahout, and the best way to do these things varies according to the nature of the vector and how it will be used.

The "sparseness" or "denseness" of a vector's data is the most important consideration. In many cases, a vector has a (non-zero) value at only few indices relative to its size. The value at all other indices is implicitly zero. If we had a vector of size 1000 and only 2 indices had a non-zero value, it wouldn't make much sense to store 998 zeroes in memory along with those 2 non-zero values. However, it wouldn't seem so wasteful if only 100 values were zero.

A vector with relatively many non-zero values in relation to its size is called dense. Such a vector can be represented with an implementation that simply stores values in an array of `doubles`. Vector indices correspond directly to array indices. Its advantage is speed: being array-backed, it's fast to access and update any of its values. `DenseVector` provides such an implementation.

Sparse vectors call for a different implementation. In order to not consume memory storing a value for zero entries, they instead only store values and indices with a non-zero value. A hash table could be used to implement a simple index-to-value mapping. Accessing an index value is slower than with direct array access, but not by much. `RandomAccessSparseVector` implements this idea.

The problem with a hash-backed implementation is that it becomes relatively slow to iterate through all values in order by index. This is frequently required. An ordered mapping based on a tree structure or similar can address this problem, since it maintains keys in order. The price of this feature is longer access time. `SequentialAccessSparseVector` is the third and final implementation, providing a `Vector` with these characteristics.

Figure B.1 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 does not allocate the space needed for zero valued cells, so it keeps an integer to indicate the index of every non-zero double value.

B.1.2 Vector operations

Now we examine frequently-used methods of `Vector`.

`get(int)` and `set(int, double)` provide the basic operations to access and change vector values at an index. `getQuick(int)` and `setQuick(int, double)` do the same but without bounds checking, which becomes a useful performance optimization internally. Other callers should likely stick to the first two methods. `size()`, as expected, returns the vector's size.

It is often useful to associate a label to each index in a Vector; in various algorithms it is used to track what that index represents or means. Variations on the getter and setter methods above exist which operate on labels, allowing access by label rather than index; see `get(String)` and `set(String, double)` for example.

Iterating over all elements is accomplished through `iterateAll()` and `iterateNonZero()`, which iterate over all indices and values, or only those with a non-zero value. The `Iterator` returned from these methods provides `Element` objects, encapsulating an index and value.

The standard `clone()` method is implemented here to copy a `Vector`; implementations provide copy constructors as well. The method `like()` returns an empty Vector of the same class. This is a form of factory method.

At last, we can talk about where the mathematics comes into this math module. `plus(Vector)` and `minus(Vector)` provide vector standard addition and subtraction. One can multiply or divide all values by a scalar with `multiply(double)` or `divide(double)`. Note methods like `dot(Vector)`, which compute a dot product (inner product), and `cross(Vector)`, which creates a matrix from the pair-wise value products of two vectors.

B.1.3 Advanced Vector methods

So far, all of this should be straightforward and as expected. `assign(Vector, BinaryFunction)` is the first non-trivial method in `Vector`. It mutates the `Vector` and sets its value equal to the result of some function of the values in the `Vector` and another `Vector`. `BinaryFunction` encapsulates a function of two numeric values that results in another numeric value. This method doesn't let us do things we couldn't otherwise accomplish with basic operations above, but may let us do them more efficiently.

For example, given vectors A and B, consider computing $C = mA - B$, where m is a scalar value. This could be implemented as in `myOperation` below.

Listing B.1 Computing Vector operations efficiently

```
Vector myOperation(Vector A, Vector B, double m) {
  return A.times(m).minus(B);
}

Vector myFasterOperation(Vector A, Vector B, final double m) {
  A.assign(new BinaryFunction() {
    public double apply(double a, double b) {
      return a * m + b;
    }
  });
}
```

Both accomplish the task. The first is simple, but slower than it appears. Each of the two mathematical operations allocates an entirely new `Vector`. The `Vectors` are traversed twice, also. The second, however, does not allocate a new `Vector` -- it destructively changes A. This may be entirely fine and desirable. It also needs only one pass over the vectors to compute the result.

`aggregate(BinaryFunction, UnaryFunction)` can simplify computing a function of all values in a vector, especially when used in conjunction with ready-made functions in `Functions`. For instance to compute the sum of squares of values in a `Vector` we could make use of this method.

Listing B.2 Computing aggregations efficiently

```
double mySumOfSquares(Vector A) {
  double sum = 0.0;
  for (Element e : A.iterateNonZero()) {
    sum += e.get() * e.get();
  }
  return sum;
}

double myOtherSumOfSquares(Vector A) {
  return A.aggregate(Functions.plus, Functions.square);
}
```

The second is not, in this case, more efficient, but is more compact.

B.2 Matrices

Matrices on the other hand are unambiguous and likely familiar -- many readers will have at least glimpsed matrix operations in a mathematics course. A matrix is, simply, a table of numbers. This simple construct has proved tremendously useful for representing concepts in physics and linear algebra -- and by extension machine learning, Operations on matrices are a fundamental part of algorithms in these fields, and hence Mahout uses matrices in many places. Often, in Mahout, it will be useful to think of the rows or columns of a matrix as a vector in its own right.

A matrix is represented by implementation of the `Matrix` interface in Mahout. Its implementation follows the same pattern as `Vector`: several different classes implement `Matrix` in order to provide performance characteristics optimized for different situations and usage patterns. The same “sparseness” consideration applies to matrices. `SparseMatrix` and `DenseMatrix` represent matrices in a manner entirely analogous to how the corresponding Vector implementations optimize themselves for matrices with few (non-zero) values, and many values relative to its size, respectively.

`SparseRowMatrix` and `SparseColumnMatrix` are interesting variants on `SparseMatrix` that are intended for use where rows (or columns) of a matrix will frequently be accessed as a unit, as a vector. `SparseRowMatrix` supports use cases where a matrix has relatively few (non-zero) rows but rows must be accessed as row vectors. `SparseColumnMatrix` behaves similarly for columns.

B.2.1 Matrix operations

Many of the same methods exist, in a slightly different form to accommodate the matrix's two-dimensional nature. `get(int, int)` returns the value at a given row and column, for example. `size()` exists and returns both row and column size of the matrix. `like()` and `clone()` are implemented as well. Label bindings for rows and columns are supported, as are convenience operations like `assign(double)`.

`Matrix` implementations provide mathematical operations like `plus(Matrix)` and `times(Matrix)` to implement matrix addition and multiplication. Other common matrix-specific operations like `transpose()` to generate the transposed matrix, and `determinant()` to compute the determinant, exist as well.

Matrix operations are largely unsurprising, but make it easy to implement otherwise complex operations in a compact way. For example, listing B.3 illustrates multiplying an m x n matrix against a vector of size n (which can be thought of as the lone column vector of an n x 1 matrix).

Listing B.3 Multiplying a matrix and vector

```
Vector vector = new SequentialAccessSparseVector(n);
Matrix matrix = new SparseMatrix(new int[] {m, n});
Matrix vectorAsMatrix = new SparseColumnMatrix(new int[] {n, 1});
Matrix productMatrix = matrix.times(vectorAsMatrix);
Vector product = productMatrix.getColumn(0);
```

This sort of operation occurs frequently in linear algebra and its application to machine learning, and this is how relatively simple it can be to implement it with Mahout's math module.

B.3 Mahout Math and Hadoop

`Matrix` and `Vector` implementations are frequently used in Hadoop-related Mapper and Reducer jobs. This means they must necessarily be serializable -- convertible into a sequence of bytes that can be stored, transmitted, and reconstituted. Hadoop does not employ Java's standard serialization mechanism via `java.io.Serializable` to accomplish this. Instead, it defines a very similar interface called `Writable` for this purpose. Classes implement this in order to make themselves usable as keys and values in Hadoop.

`Vector` and `Matrix` do not themselves extend `Writable`, since this would make the math module depend on Hadoop, and conceptually it functions independently of something like Hadoop. Within the core Mahout module, you will find instead `VectorWritable` and `MatrixWritable`. These implement both `Writable` and `Vector` / `Matrix`, respectively. They encapsulate knowledge of how to serialize a vector and matrix within Hadoop. In `Mapper` and `Reducer` jobs which read or write vectors or matrices, "raw" implementations will be wrapped in such classes before passing to Hadoop, and will be returned in this form.

C Resources

The internet can provide more, and more recent, resources related to machine learning and Mahout than we can here. Search for “collaborative filtering”, “clustering”, “classification” and more at your favorite search engine, or in particular, research-specific search engines like Google Scholar (http://scholar.google.com).

However, here the authors present classic papers and references that will be of interest to anyone looking to better understand the underpinnings of the content of Mahout and this book.

C.1 Collaborative Filtering and Recommenders

- J.S. Breese, D. Heckerman and C. Kadie, "Empirical Analysis of Predictive Algorithms for Collaborative Filtering," in Proceedings of the Fourteenth Conference on Uncertainity in Artificial Intelligence (UAI 1998), 1998. http://research.microsoft.com/research/pubs/view.aspx?tr_id=166
- B. Sarwar, G. Karypis, J. Konstan and J. Riedl, "Item-based collaborative filtering recommendation algorithms" in Proceedings of the Tenth International Conference on the World Wide Web (WWW 10), pp. 285-295, 2001. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.29.340&rep=rep1&type=pdf
- P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom and J. Riedl, "GroupLens: an open architecture for collaborative filtering of netnews" in Proceedings of the 1994 ACM conference on Computer Supported Cooperative Work (CSCW 1994), pp. 175-186, 1994. http://www.si.umich.edu/~presnick/papers/cscw94/GroupLens.htm
- J.L. Herlocker, J.A. Konstan, A. Borchers and J. Riedl, "An algorithmic framework for performing collaborative filtering" in Proceedings of the 22nd annual international ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 99), pp. 230-237, 1999. http://www.grouplens.org/papers/pdf/algs.pdf
- Clifford Lyon, "Movie Recommender" CSCI E-280 final project, Harvard University, 2004. http://materialobjects.com/cf/MovieRecommender.pdf
- Daniel Lemire, Anna Maclachlan, "Slope One Predictors for Online Rating-Based Collaborative Filtering," Proceedings of SIAM Data Mining (SDM '05), 2005. http://www.daniel-lemire.com/fr/documents/publications/lemiremaclachlan_sdm05.pdf
- Michelle Anderson, Marcel Ball, Harold Boley, Stephen Greene, Nancy Howse, Daniel Lemire and Sean McGrath, "RACOFI: A Rule-Applying Collaborative Filtering System," Proceedings of COLA '03, 2003. http://www.daniel-lemire.com/fr/documents/publications/racofi_nrc.pdf
- Anand Rajaraman and Jeffrey D. Ullman, “Mining of Massive Data Sets,” 2010. http://infolab.stanford.edu/~ullman/mmds.html