# brief contents

## PART 1  BACKGROUND AND FUNDAMENTALS .........................................................1

1. Hadoop in a heartbeat 3
2. Introduction to YARN 22

## PART 2  DATA LOGISTICS ........................................................................59

3. Data serialization—working with text and beyond 61
4. Organizing and optimizing data in HDFS 139
5. Moving data into and out of Hadoop 174

## PART 3  BIG DATA PATTERNS ..................................................................253

6. Applying MapReduce patterns to big data 255
7. Utilizing data structures and algorithms at scale 302
8. Tuning, debugging, and testing 337

## PART 4  BEYOND MAPREDUCE ...............................................................385

9. SQL on Hadoop 387
10. Writing a YARN application 425
Predictive analytics with Mahout

This chapter covers

- Using recommenders to make product suggestions
- Spam email classification with naive Bayes
- Clustering to identify trends or patterns in data

Predictive analytics is the field of deriving information from current and historical data. It’s one of the main tools of the data scientist, whose job is to examine large datasets (often called big data) and derive meaningful insights from that data, optimally in the form of new products. Predictive analytics can be broken down into three broad categories:

- **Recommenders**—Recommender systems suggest items based on past behavior or interests. These items can be other users in a social network, or products and services in retail websites.
- **Classification**—Classification (otherwise known as supervised learning) infers or assigns a category to previously unseen data based on discoveries made from some prior observations about similar data. Examples of classification include email spam filtering and detection of fraudulent credit card transactions.
Clusterings—A clustering system (also known as unsupervised learning) groups data together into clusters. It does so without learning the characteristics about related data. Clustering is useful when you’re trying to discover hidden structures in your data, such as user habits.

Mahout is a machine learning library that includes implementations of these three classes of predictive analytics techniques. Many of its algorithms have MapReduce implementations, which is the focus of this chapter, and this is where Mahout comes into its own—in its ability to work with huge datasets that other predictive analytics tools can’t support. In fact, Mahout only starts to make sense if you’re working with datasets that number in the millions or more.

In this chapter we’ll look at the Mahout MapReduce implementations of recommenders, classifiers, and clusterers. You’ll use recommenders to recommend movies similar to movies that users have already rated; you’ll write a classifier that can filter out spam emails; and, finally, we’ll look at how you can use clustering to discover structure in your data.

**Mahout version** At the time of writing, Mahout 1.0, which is the first Mahout version to support Hadoop 2, hasn’t been released. The code and examples in this chapter were developed with a snapshot of the Mahout 1.0 release, and as a result aren’t guaranteed to work with a newer version of Mahout.

We’ll get things started with a look at recommenders.

### 12.1 Using recommenders to make product suggestions

Recommender systems, which are also known as collaborative filtering (CF) systems, are the computer equivalent of you asking your friends for a restaurant recommendation. The more recommendations you receive from your friends for a particular restaurant, the higher the probability that you’ll go there. In the online world you see recommender engines in play every day—most social and retail websites recommend new people for you to interact with, or a new product for you to buy.

There are two types of collaborative recommenders: user-based recommenders and item-based recommenders:

- **User-based recommenders** look at users similar to a target user, and use their collaborative ratings to make predictions to the target user.
- **Item-based recommenders** look at similar items, and use this information to recommend items that are related to items previously used by a target user.

You can see the results of recommenders in action in figure 12.1.

Both types of recommender systems need to be able to determine the degree of similarity between users or items, so we first need to look at how similarity metrics work.
Using recommenders to make product suggestions

An example of a recommender suggesting other users of interest in a social network

A movie recommender suggesting previously unseen movies to a user

Who to follow
Twitter accounts suggested for you based on who you follow and more.

Suggestions to Watch Instantly

Figure 12.1 Examples of recommenders that suggest other users of interest in a social network, and movies that a user may be interested in watching

12.1.1 Visualizing similarity metrics

In both user- and item-based recommenders, the system needs to find similar users or items. They do this by comparing users or items with each other to arrive at a similarity score. Popular measures that can calculate these scores include Euclidean distance and Pearson’s correlation. These algorithms operate on numerical data, where the data points are vector-like (points in space).

The Euclidean distance is the most commonly used distance measure. Figure 12.2 shows how the Euclidean distance calculation works.

If \( p = (p_1, p_2) \) and \( q = (q_1, q_2) \), then the Euclidean distance is

\[
d(p, q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2}
\]

Figure 12.2 A plot showing user preferences for two movies, and how the Euclidean distance is calculated
The Euclidean distance is in a family of related distance measures, which also includes the Manhattan distance (the distance between two points measured along axes at right angles). These measures are all similar in how they calculate distances, so switching from one to the other is not likely to significantly change results.

Correlation-based measures, however, are less concerned with the distance between points in a dataset and care more about similarity, which is the degree of linear relationship between two variables. Pearson’s correlation is widely used in science to measure the dependencies between two variables. Its advantage over the Euclidean distance is that it can be used to find correlations between users even if one user tends to give higher scores than another user, assuming that, in general, they like and dislike the same movies. Figure 12.3 illustrates this correlation.

Pearson’s correlation will result in a number between -1 and 1, which is an indicator of how much two series of numbers move proportionate to each other, and exhibits a linear relationship. A value of 1 indicates the highest correlation and -1 the lowest. Figure 12.3 shows an example of a highly correlated relationship.

The math behind Pearson’s correlation is more complex than the Euclidean distance and is outside of the scope of this chapter, but if you’re interested, you can read more details on Wikipedia (http://en.wikipedia.org/wiki/Pearson_product-moment_correlation_coefficient).

Mahout can support the Euclidean and Pearson’s similarity measures for both user-based and item-based recommenders. Mahout supports additional similarity measures for item-based recommenders such as the Tanimoto coefficient. All the user-based and item-based similarity implementations can be seen by searching for implementations of the UserSimilarity and ItemSimilarity interfaces.

Next we’ll look at the dataset you’ll use in this section on recommenders.

---

1 For info on the Tanimoto coefficient, see the Wikipedia article on the Jaccard index: http://en.wikipedia.org/wiki/Jaccard_index#Tanimoto_Similarity_and_Distance.
### 12.1.2 The GroupLens dataset

GroupLens is a research lab in the Department of Computer Science and Engineering at the University of Minnesota. They perform research into recommender systems, among other research areas, and provide a dataset of 1 million ratings from 6,000 users on 4,000 movies at [www.grouplens.org/node/12](http://www.grouplens.org/node/12). In this section you’ll download this data and prepare it in a format that can be used by the Mahout recommender engines.

**Mahout installation**  Follow the instructions in the appendix to install Mahout on your system before executing the instructions in this section.

First, download and unzip the data as follows:

```bash
$ export MAHOUT_HOME=<path-to-your-mahout-directory>
$ cd $MAHOUT_HOME
$ mkdir -p corpus/groupLens-1m
$ cd corpus/groupLens-1m
$ curl -O http://files.grouplens.org/datasets/movielens/ml-1m.zip
$ unzip ml-1m.zip
```

Archive:  ml-1m.zip
  creating: ml-1m/
  inflating: ml-1m/movies.dat
  inflating: ml-1m/ratings.dat
  inflating: ml-1m/README
  creating: __MACOSX/
  creating: __MACOSX/ml-1m/
  inflating: __MACOSX/ml-1m/._README
  inflating: ml-1m/users.dat

The ratings file contains data in the following format:

```
UserID::MovieID::Rating::Timestamp
```

- UserIDs range between 1 and 6040
- MovieIDs range between 1 and 3952
- Ratings are made on a 5-star scale (whole-star ratings only)
- Timestamp is represented in seconds since the epoch
- Each user has at least 20 ratings

You can see an example from the top of the file:

```bash
$ head -n 5 ml-1m/ratings.dat
1::1193::5::978300760
1::661::3::978302109
1::914::3::978301968
1::3408::4::978300275
1::2355::5::978824291
```

This isn’t quite in the format that you need for Mahout, which expects a CSV-delimited format as follows:

```
UserID,ItemID,Value
```
You can write a simple awk script to convert the GroupLens data into CSV form:

```bash
$ awk -F "::" '{print $1","$2","$3}' ml-1m/ratings.dat > ratings.csv
```

Once you run this script, your data will be in a form that you can use for the techniques in this section.

### 12.1.3 User-based recommenders

Mahout doesn’t have a way to run user-based recommenders in MapReduce because the user-based recommender is only designed to work within a single JVM. Because user-based recommenders aren’t relevant for this book, we’ll skip them and move on to look at item-based recommenders. If you’re curious about user-based recommenders, the book *Mahout in Action*, by Sean Owen, et al. (Manning, 2011, [www.manning.com/owen/](http://www.manning.com/owen/)), covers them in detail.

### 12.1.4 Item-based recommenders

Item-based recommenders calculate recommendations based on items, not users. The goal is the same as that of user-based recommenders—recommend items that the user will hopefully be interested in. But rather than look at similarities between users, item-based recommenders look at similarities between items.

Item similarities tend to be less volatile than user similarities, and as a result they lend themselves more to precomputation, which can speed up the recommender operation. The item-based recommender in Mahout supports a distributed execution model so that it can be computed using MapReduce. The distributed recommender will be the focus of this section.

Let’s use figure 12.4 to walk through a simple example of item recommendation.

![Figure 12.4](image.png)

*Figure 12.4 A simple example of item recommendation*
To recommend items for Anna, item recommendation looks at other items that co-occur with items Anna reviews (items 3 and 4), determines their similarity (based on reviewer ratings), and then ranks them by multiplying Anna’s rating with the similarity rating for other items. In this example, item 3 would have a higher predicted value than item 4, because John rated 1 and 3 as high, but Peter rated 1 as low.

Let’s now look at how you can get the distributed item-based recommender working.

---

**TECHNIQUE 108 Item-based recommenders using movie ratings**

How would you use the GroupLens movie rating dataset to recommend movies for three users in the dataset using MapReduce?

**Problem**

You have a large number of user item preferences, and you want to recommend additional items to users in a scalable fashion.

**Solution**

Use Mahout’s MapReduce item-based recommender in combination with the GroupLens data to make movie recommendations. Mahout’s item recommender uses ten MapReduce jobs to perform this recommendation, which perform functions such as data preparation, similarity co-occurrence calculations, and final recommendation calculations.

**Discussion**

The distributed item-based recommender requires two inputs: the recommendations file and a file of user IDs for which item recommendations will be generated. Let’s go ahead and create the file of user IDs (you’ll select the first three IDs from the ratings file), and then push both files into HDFS:

```
$ cat > user-ids.txt << EOF
1
2
3
EOF

$ hadoop fs -put user-ids.txt ratings.csv .
```

You’re now ready to run the item-based recommender:

```
$ export MAHOUT_HOME=<path-to-your-mahout-directory>
$ export HADOOP_HOME=<path-to-your-hadoop-directory>
$ $MAHOUT_HOME/bin/mahout recommenditembased \
   -Dmapred.reduce.tasks=10 \
   --similarityClassname SIMILARITY_PEARSON_CORRELATION \
   --input ratings.csv \
   --output item-rec-output \
   --tempDir item-rec-tmp \
   --usersFile user-ids.txt
```
You must export the `HADOOP_HOME` environment variable to refer to the location of your local install of Hadoop (/usr/lib/hadoop for CDH-packaged installations, and /usr/local/hadoop for installations that followed the tarball instructions in the appendix). Mahout uses this variable to discover your Hadoop cluster settings. If this step is omitted, Mahout will use the local filesystem for storage and run MapReduce jobs on your client host.

When the job completes, you can view the output in HDFS. The output format consists of the user ID, followed by a comma-separated list of item IDs and their related scores:

```
$ hadoop fs -cat item-rec-output/part*
1 [1946:5.0,3049:5.0,1016:5.0,1366:5.0,368:5.0,617:5.0,832:5.0,1937:5.0,2133:5.0,2639:5.0]
2 [449:5.0,2735:5.0,1958:5.0,3260:5.0,869:5.0,3507:5.0,474:5.0,3148:5.0,2745:5.0,1351:5.0]
3 [368:5.0,2372:5.0,2541:5.0,1937:5.0,2478:5.0,2212:5.0,832:5.0,1078:5.0,3421:5.0,2398:5.0]
```

**Summary**

Running the distributed item-based recommender resulted in ten MapReduce jobs being executed. The jobs and a quick description of them can be seen in figure 12.5.

The distributed implementation of an item-based recommender creates a co-occurrence matrix to associate similar items together. It does this by combining items with similar ratings from each user and then counting the number of times that each pair of items was rated by all the users. It then predicts the ratings for unknown items by multiplying the users’ ratings for an item with all the item’s co-occurrences, and then sorts all these item predictions and retains the top $K$ as recommendations.

So far we’ve focused on the item-based recommender where items are entities such as movies or products. You could also use this approach to recommend users. Existing user relationships could be modeled by replacing item IDs with user IDs. The rating value itself could then become a constant, or you could use it to model the level of friendship (for example, if two users commonly interact, their rating could be higher than those who don’t).

Mahout also comes with another distributed item-based recommender called the *slope-one recommender*, which is a simpler recommender that just requires two MapReduce jobs. It doesn’t use a similarity algorithm to measure the similarity of items, and instead performs a simple average of the difference between rating values. Take a look at the `SlopeOneAverageDiffsJob` class if you’re interested in more details.

In this section you learned how user-based and item-based recommenders work, and you looked in detail at how a MapReduce item-based recommender could be used to make movie recommendations. Let’s now move on to how you can use classification to make movie predictions.
1. **Data preparation**
   - **ItemIDIndex**: Maps item IDs from long to int.
   - **ToUserVectors**: Creates a vector of users and items they reviewed.
   - **ToItemVectors**: Inverts the data so that on each line an item contains a list of users and their rating for that item.

2. **Create a similarity co-occurrence matrix**
   - **VectorNorm**: Normalizes the ratings.
   - **Co-occurrence similarity**: Creates a co-occurrence matrix for each item and other items rated by the same user, only retaining item pairs that are similar.
   - **Unsymmetrify/TopK**: Retains only the top K most similar related items.

3. **Calculate the recommendations**
   - **SimilarityMatrixRow Wrapper**: Nullifies similarity numbers of co-occurrences containing an item the user has already reviewed.
   - **UserVectorSplitter**: Inner join between users to be recommended with the ratings data.
   - **ToVectorAndPref**: Inner join co-occurrence matrix items with items reviewed by user.
   - **Recommend**: Multiple user rating for each co-occurring item and rank to determine best predictions.

---

**Figure 12.5** MapReduce jobs executed for distributed item-based recommender

### 12.2 Classification
Classification, also known as *supervised learning*, is a fancy term for a system that makes predictions on data based on previously known data. As humans we do this all the time—when you see an email titled “REQUEST FOR URGENT BUSINESS RELATIONSHIP,” do you eagerly open it? No. Prior experience has told you that this combination of words, and the fact that they’re uppercase, means that this email is most likely spam. This is an example of human supervised learning, where your current behavior is a result of previous observations you made on similar data. You may not have seen an email subject with the exact same sequence of words, but
you’ve seen enough examples of similar email subjects that were spam to make you immediately suspicious.

Supervised learning works in exactly the same way. In the case of email spam detection, you train a system using data that has already been labeled (or marked) as being either spam or ham (legitimate email) to build a model, and then use that model to make predictions about emails that the system hasn’t seen before.

In this section we’ll look at one of the simpler supervised learning algorithms, naive Bayes, and see how you can use it in conjunction with Hadoop to build a scalable spam training and classification system.

There are multiple steps involved in building such a system, which are illustrated in figure 12.6.

Some of the terms you saw in figure 12.6 will be used throughout this section and are defined in figure 12.7.

Two additional terms also need to be defined:

- **Training**—The process by which categorized documents are used to build a model, which can be used by a classifier to categorized unseen documents.
- **Classifier**—A classifier uses a model extracted from the training data to make predictions about unseen documents or datasets.

Before we dive into the Hadoop side of supervised learning, we’ll walk through building a classifier using a simple training dataset to get an idea of the overall process.

![Figure 12.6 The steps involved in building a supervised learning model](image-url)
A category (also called a “label”) is a class of items that you want to classify your data into. The examples here show the categories in a spam-classification system.

A document is an item to train or classify. In this example, they are emails.

Training data

email 1  
email 2  
email 3  

Test data

email 1  
email 2  
email 3  

A feature is an attribute of your data that you want to include when training and classifying. They could be individual words, a series of words, or any other attributes.

Spam

Ham

Figure 12.7  Classification term definitions

12.2.1  Writing a homemade naive Bayesian classifier

We just identified the five steps it takes to build a classifier. In this section, you’ll apply these five steps to write your own naive Bayes spam classifier.

To classify emails, a naive Bayes classifier examines words that occur in emails and checks to see if they’re more likely to occur in spam or ham categories. If a word is used frequently in an email, and the spam category has also observed a high frequency of that word (and the ham category has seen the word less frequently), then the word is deemed to be more spam than ham. The Bayes theorem comes into play once each word has had a spam and ham probability calculated, and combines them together to form the overall email probability of ham or spam.

Your first task is to find some data to train your classifier.

FINDING SOME TRAINING DATA

The two hardest parts of building a classifier happen to be the first two steps. When it comes to training data, your classifier is only as good as the data used to train it. Luckily
for you, there are some high-quality datasets you can use when working with spam. For now we’ll conjure up a few example subject lines that you’ll use for training data, as shown in figure 12.8.

SELECTING FEATURES

After you have a set of training data that you’re happy with, you can start examining that data for attributes that you want to use for training. This is called feature selection, and there’s a whole science around this one topic. The goal of feature selection is to pick features that you believe will increase the ability of your classifier to separate data into the different categories.

When building an email classifier, there are several attributes within an email that can be used as features, such as the contents of the subject line, other email header data points, and the body of the email. As an example, the date the email was sent doesn’t lend itself to being a feature because there’s nothing about the date that can be used to separate ham from spam. But the email subject and body definitely contain text that can be used to help identify spam.

The text in the email subject and body are useful, but should all the words be used as features? The short answer is no. You don’t want words that commonly appear in the English language (referred to in text-mining circles as stopwords) to be used as features because they’ll appear in every single email. Similarly, you don’t want to use words that are rare and only appear in a handful of emails, because they won’t help with classification either.

For now, let’s assume that your data only consists of the email subject line.

PREPARING DATA

Now that you know what features you’re interested in, you need to prepare the data for training. You’re working with the naive Bayes classifier, which can work with data in text form, so you don’t need to manipulate or transform your data. Therefore, all you need to do is feed your training data to your classifier:\(^2\)

```java
Classifier c = new Classifier();
c.train("ham windyhill roofing estimate");
c.train("ham quick hadoop meetup");
c.train("spam cheap quick xanax");
c.train("spam quick easy money");
```

TRAINING A CLASSIFIER

For naive Bayes, training is a matter of reading in the documents for each category, and for each document, extracting the words and calculating for each word the probability of that word in the category.

---

To start with, you need to split up the words for the training document and extract the category and the associated words:\(^3\)

```java
public static class Classifier {
    Map<String, Category> categories = new HashMap<String, Category>();

    public void train(String document) {
        String[] parts = StringUtils.split(document);
        String category = parts[0];
        List<String> words = Arrays.asList(
            Arrays.copyOfRange(parts, 1, parts.length));
        Category cat = categories.get(category);
        if (cat == null) {
            cat = new Category(category);
            categories.put(category, cat);
        }
        cat.train(words);
        for (Category c : categories.values()) {
            c.updateProbability(numDocuments);
        }
    }
}
```

Next you need to model the notion of a category, and keep track of the words in the category, the number of times you saw the word in the category, and the number of documents that were used to train the category:\(^4\)

```java
public static class Category {
    String label;
    int numDocuments;
    double categoryProbability;
    Map<String, MutableInt> features =
        new HashMap<String, MutableInt>();

    void train(List<String> words) {
        numDocuments++;
        for (String word : words) {
            MutableInt i = features.get(word);
            if (i == null) {
                i = new MutableInt(0);
                features.put(word, i);
            }
            i.increment();
        }
    }

    void updateProbability(int totalDocuments) {
        categoryProbability = (double) numDocuments /
            (double) totalDocuments;
    }
}
```

---

\(^3\) GitHub source: https://github.com/alexholmes/hiped2/blob/master/src/main/java/hip/mahout/
HomegrownNBClassifier.java

\(^4\) GitHub source: https://github.com/alexholmes/hiped2/blob/master/src/main/java/hip/mahout/
HomegrownNBClassifier.java
CHAPTER 12  Predictive analytics with Mahout

Category = ham  Category = spam
Training documents = 2  Training documents = 2
Words in category = 6  Words in category = 5
Category probability = 0.5  Category probability = 0.5

Word frequencies:
- hadoop 1
- meetup 1
- quick 1
- roofing 1
- estimate 1
- windyhill 1

Word frequencies:
- easy 1
- money 1
- quick 2
- xanax 1
- cheap 1

The number of times the word “cheap” appeared in the training documents for the category “spam.”

The probability that a randomly selected document is in this category, calculated by dividing the number of documents for the category by the total number of documents.

All the words that appeared in the training documents for the category.

After you’ve trained your classifier with four documents, it contains all the data it needs for classification. Figure 12.9 shows the data that you have in your classifier after you’ve finished training.

RUNNING A CLASSIFIER

You’ve built a model of the training data, and you’re now ready to use it to classify some new emails. There are two parts to your naive Bayes classifier: First you need to extract the words from the email and calculate the probability that the words exist in each category. Then you multiply these probabilities to form the overall probability for each category.

When you see a new email subject, you need to calculate the probability of each word being in a category. The probability is expressed numerically between 0 and 1, where 1 indicates a probability of 100 percent. To calculate this number, you take the number of times a word appears in the category and divide it by the total number of documents for the category. The following code shows this probability being calculated for a word in a specific category.5

```java
public static class Category {
    int numDocuments;
    Map<String, MutableInt> features =
        new HashMap<String, MutableInt>();

    double weightedProbability(String word) {
        MutableInt i = features.get(word);
        return (i == null ? 0.1 : (i.doubleValue() / (i.doubleValue() + numDocuments)))
    }
}
```


---

If you take the example of the word *quick*, it appeared twice in the spam category, and there were two documents used to train the spam category, so the probability is \((2/2) = 1\). The same word appeared once in the ham category, so its probability in the ham category is \((1/2) = 0.5\). These values will be used as the value of \(P(W|C)\) in the Bayes formula, which you’ll see shortly.

You now have the probability that each individual word is in a category, but how do you calculate the probability of the overall document being in a category? Bayes comes to the rescue with a formula into which you can plug your word probabilities, shown in figure 12.10.

Figure 12.11 shows how you would use your naive Bayes classifier to calculate probabilities of membership in spam and ham categories.
You want to classify a previously unseen email.

**Email subject: quick money**

\[
P(D|C_i) = P(W_1|C_i) * P(W_2|C_i) * P(C_i)
\]

Probability that email is ham.

\[
P(D|C_{ham}) = \frac{1}{2} * (0.1) * 0.5 = 0.025
\]

Number of times "quick" appears in category "ham." Divided by the number of training documents in "ham."

"Money" doesn’t appear in the ham category, so you set its probability to 0.1.

Probability that email is spam.

\[
P(D|C_{spam}) = \frac{2}{2} * \frac{1}{2} * 0.5 = 0.25
\]

**Figure 12.11** Example application of Bayes theorem on ham and spam

The following code calculates the probability of the input document belonging to each of the categories using the naive Bayes formula you just saw:

```java
public void classify(String words) {
    String[] parts = StringUtils.split(words);
    for (Category c : categories.values()) {
        double p = 1.0;
        for (String word : parts) {
            p *= c.weightedProbability(word);
        }
        System.out.println("Probability of document '" + words + "' for category '" + c.label + "' is " + (p * c.categoryProbability));
    }
}
```

If you run this code, you’ll see the same values that you calculated in figure 12.11:

```
$ hip hip.mahout.HomegrownNBClassifier
Category = ham, numDocs = 2, categoryProbability = 0.5
  hadoop 1
  meetup 1
  quick 1
  roofing 1
  estimate 1
  windyhill 1
Category = spam, numDocs = 2, categoryProbability = 0.5
  easy 1
  money 1
  quick 2
```
Probability of document 'quick money' for category 'ham' is 0.025
Probability of document 'quick money' for category 'spam' is 0.25

In real life, your classifier wouldn’t work because of the small number of training documents. You’d ideally want a large and equal number of spam and ham documents for training purposes. Let’s move on to how you’d do that with Mahout.

12.2.2 A scalable spam-detection classification system

Classification in Mahout can be executed sequentially or via MapReduce. All of the Mahout supervised learning algorithms can be executed in MapReduce, but only a few of them have parallel execution models. Naive Bayes is one of them.6

Technique 109 Using Mahout to train and test a spam classifier

How would you train a spam classifier using MapReduce and see how well it performs on unseen emails?

- Problem
  You have a large spam corpus, and you want to build a spam classifier.

- Solution
  Use Mahout’s MapReduce naive Bayes classifier on the SpamAssassin corpus to train a model and then test its effectiveness on unseen emails.

- Discussion
  For this technique you’ll use the SpamAssassin corpus, which can be downloaded from http://spamassassin.apache.org/publiccorpus/. You’ll use this corpus both to train your classifier and to test it to see how well it performs at detecting spam. Your first step is to download and extract the spam and ham datasets:

```
$ cd $MAHOUT_HOME
$ mkdir -p corpus/spam-assassin
$ cd corpus/spam-assassin
$ curl -O http://spamassassin.apache.org/publiccorpus/20021010_spam.tar.bz2
$ curl -O http://spamassassin.apache.org/publiccorpus/20021010_easy_ham.tar.bz2
$ tar xjf 20021010_spam.tar.bz2
$ tar xjf 20021010_easy_ham.tar.bz2
$ ls -l spam/* | wc -l
501
$ ls -l easy_ham/* | wc -l
2551
```


---

Next you need to extract an even number of spam and ham emails for training and testing. Using the same number of spam and ham emails will help with the performance of your classifier:

```$ mkdir -p docs/easy_ham docs/spam
$ ls -l spam/* | head -n 400 | while read file; do cp $file docs/$file; done
$ ls -l easy_ham/* | head -n 400 | while read file; do cp $file docs/$file; done
```

Next you need to upload the corpus to HDFS and convert it into a form that the training code can work with. The data as it stands right now consists of a file for each email, and the format you want to convert it into contains an email (or document) per line, with the category name as the first token in the line. Mahout has a built-in tool that can perform that conversion for you, after which you’ll split the data into training and testing sets:

```$ hdfs dfs -put docs .
$ export HADOOP_HOME=/usr/lib/hadoop
$ $MAHOUT_HOME/bin/mahout seqdirectory
  -i docs
  -o seq

$ $MAHOUT_HOME/bin/mahout seq2sparse
  -i seq
  -o vectors
  -lnorm -nv
  -wt tfidf

$ $MAHOUT_HOME/bin/mahout split
  -i vectors/tfidf-vectors
  --trainingOutput train-vectors
  --testOutput test-vectors
  --randomSelectionPct 40 --overwrite --sequenceFiles -xm sequential
```

Now that your data is in a form that can be used by the training tool, you can run that tool to generate the model:

```$ export HADOOP_HOME=/usr/lib/hadoop
$ $MAHOUT_HOME/bin/mahout trainnb
  -i train-vectors -el
  -o model
  -li labelindex
  -ow
```

When the training has completed, it persists the model into several subdirectories under the model directory in HDFS. You can now run the classifier on your test data by specifying the location of your model:
$ export HADOOP_HOME=/usr/lib/hadoop
$ $MAHOUT_HOME/bin/mahout testnb \
   -i test-vectors \
   -m model \
   -l labelindex \
   -ow -o testing \
   ...

Summary

| Correctly Classified Instances | 296 | 98.9967% |
| Incorrectly Classified Instance | 3  | 1.0033% |
| Total Classified Instances     | 299 |

Confusion Matrix

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>Classified as</th>
</tr>
</thead>
<tbody>
<tr>
<td>153</td>
<td>0</td>
<td>easy_ham</td>
</tr>
<tr>
<td>3</td>
<td>143</td>
<td>spam</td>
</tr>
</tbody>
</table>

Statistics

| Kappa     | 0.961 |
| Accuracy  | 98.9967% |
| Relevance | 65.9817% |
| Reliability (standard deviation) | 0.5715 |
| Weighted precision | 0.9902 |
| Weighted recall | 0.99 |
| Weighted F1 score | 0.99 |

The output of the `testnb` command shows something called a confusion matrix, which is telling you that the classifier correctly identified 143 spam emails as being spam, and incorrectly identified 3 spam emails as ham. Your classifier was also 100-percent accurate at classifying ham emails.

Summary

The scalability that can be achieved with training in Mahout comes at a high cost, as multiple steps are required to train the model.

When you completed testing your classifier, you were presented with something called a confusion matrix. The confusion matrix is used to visualize the performance of your classifier and tells you how well your model works. Let’s look at the confusion matrix and try and make head or tail of the output (see figure 12.12).

The MapReduce naive Bayes training comes into its own when you’re working with large training sets (hundreds of thousands and more), which start hitting the memory limits of a single host.
Online and offline classification

There are two options available when using your classifier to classify documents—online and offline:

- In online mode you’re responding to a classification request in real time, such as a backend REST API. In this case you can use the Mahout BayesAlgorithm and InMemoryBayesDatastore to load the model from the filesystem and perform inline classification.
- Offline mode would be suited to cases where you want to classify millions of documents. In this situation you’d write a MapReduce job that performs a function similar to Mahout’s testing code. You should look at the BayesClassifierDriver class and use the code there to write your own MapReduce classifier.

12.2.3 Additional classification algorithms

Mahout contains other classification algorithms (in various stages of completion), which are outlined in table 12.1.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logistic regression</td>
<td>Logistic regression is a model used to predict the probability of the occurrence of an event. It makes use of several predictor variables that may be either numerical or categories.</td>
</tr>
<tr>
<td>Support Vector Machines</td>
<td>As with naive Bayes, Support Vector Machines (or SVMs) can be used to solve the task of assigning objects to classes. The way this task is solved is completely different from the settings in naive Bayes.</td>
</tr>
<tr>
<td>Neural network</td>
<td>Neural networks are a means of classifying multidimensional objects.</td>
</tr>
<tr>
<td>Hidden Markov models</td>
<td>Hidden Markov models are used in multiple areas of machine learning, such as speech recognition, handwritten letter recognition, and natural language processing.</td>
</tr>
</tbody>
</table>
For more details on these algorithms, take a peek at http://mahout.apache.org/).

The final predictive analytics technique we’ll look at is clustering. It involves a series of algorithms whose goal is to group data together and help you make new discoveries about your data.

12.3 Clustering with K-means

In the previous section we looked at supervised learning, where a classifier was built with some data that was previously categorized. Clustering is an example of an unsupervised learning technique where no such categorized data is used to build the model. Clustering is different from classification and recommenders—the goal of classification and recommenders is to make predictions about some data, such as whether or not a web page is in a sports category, or whether a user is likely to enjoy a book they haven’t read. The goal of clustering is to partition data into a number of clusters so that data in each cluster is more similar to each other than to data in other clusters.

Clustering can be used to group together related genes for medical research purposes, or to help readers find additional resources related to the current article.

To help you understand clustering, figure 12.13 shows an example of data that is tightly grouped into three distinct clusters. The goal of clustering is to identify these clusters so that you can make new discoveries about your data. In this figure it’s obvious that these clusters exist, but in reality, your data isn’t likely to be so well separated.

Clustering has many applications in fields such as marketing (finding groups of customers with similar behaviors), biology (classifying plants and animals), and more.

Mahout has a number of clustering algorithms, but we’ll look at K-means, one of the simpler algorithms.

12.3.1 A gentle introduction

K-means is the oldest and simplest clustering algorithm. With K-means, you tell it ahead of time how many clusters you’re looking for (the $K$ in K-means).

The K-means process starts with the initial placement of the $K$ cluster centroids. The initial K centroids can either be randomly located or specifically placed. Randomly
located centroids will likely give different results, so the recommendation is that cen­
troids be located as far away from each other as possible.

Once the initial centroids have been placed, K-means follows an iterative algo­rithm whereby each data point is associated to the nearest cluster centroid, and then the cluster centroids are repositioned relative to all the data points. This process repeats until the cluster centroids don’t move, at which time the clusters are consid­ered to have **converged**.

To determine the distances between data points and the cluster centroids, clustering supports most of the similarity metrics that you saw in the recommenders section, such as Euclidean distance, Tanimoto, and Manhattan.

A high-level algorithm for K-means is as follows:

1. Model the objects that you want to cluster into \( N \) dimensions.
2. Place the \( K \) centroids into the space represented by the objects.
3. Using a distance metric, assign each object to the centroid that is closest to it.
4. Recalculate the position of the \( K \) centroids.
5. Repeat steps 3 and 4 until either the \( K \) centroids don’t move/converge beyond a certain threshold, or the maximum number of iterations has been reached.

Figure 12.14 shows this algorithm in play in a two-dimensional space.

Now that you understand the basics of K-means, let’s look at how you can use it in Mahout.

### 12.3.2 Parallel K-means

As with many algorithms in Mahout, K-means has both sequential (in-memory) and parallel (MapReduce) implementations. In this section we’ll look at the parallel implementation of K-means.

#### TECHNIQUE 110  **K-means with a synthetic 2D dataset**

How would you execute parallel K-means on a synthetic two-dimensional (2D) dataset which has 3,000 data points containing 20 clusters?

- **Problem**
  You want to detect clusters using K-means.

- **Solution**
  Use Mahout’s MapReduce K-means algorithm to cluster data together and observe the importance of the placement of the initial centroids.

---

7 The dataset can be downloaded from the “Clustering datasets” web page from the University of Eastern Finland’s Speech and Image Processing Unit at [http://cs.joensuu.fi/sipu/datasets/](http://cs.joensuu.fi/sipu/datasets/).
TECHNIQUE 110  K-means with a synthetic 2D dataset

Model the data that you want to cluster.

Assign each object to the closest centroid.

Recalculate the positions of the K centroids.

Repeat steps 3 and 4 until the centroids don’t move.

Assign each object to the closest centroid.

Recalculate the positions of the K centroids.

Object 4 is assigned to centroid A because it is closer.

At this point you’re done because the centroids won’t move.

**Figure 12.14** A walk-through of the K-means algorithm on a simple dataset

**Discussion**

The synthetic data is a series of two-dimensional data points that represent twenty synthetic clusters:

```
$ head -n 5 test-data/mahout/synthetic.txt
54620 43523
52694 42750
53253 43024
54925 42624
54973 43980
```

If you produce a scatter plot of this data, you’ll see output similar to figure 12.15.
You need to convert the synthetic 2D data into the SequenceFile format required by Mahout for clustering:\(^8\)

```
public static void write(File inputFile, Path outputPath)
        throws IOException {
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(conf);

    The value is a VectorWritable, which is a Mahout type.
    SequenceFile.Writer writer = 
        SequenceFile.createWriter(fs, conf, outputPath,
            NullWritable.class,
            VectorWritable.class,
            SequenceFile.CompressionType.BLOCK,
            new DefaultCodec());

    try {
        for (String line : FileUtils.readLines(inputFile)) {
            String parts[] = StringUtils.split(line);

            writer.append(NullWritable.get(),
                new VectorWritable(new DenseVector(

```

\(^8\) GitHub source: https://github.com/alexholmes/hiped2/blob/master/src/main/java/hip/mahout/
Synthetic2DClusteringPrep.java.
TECHNIQUE 110  K-means with a synthetic 2D dataset

```java
new double[]{
    Double.valueOf(parts[0]),
    Double.valueOf(parts[1])
}

finally {
    writer.close();
}
```

Next, use your utility to create the input data in HDFS:

```
$ hip --nolib hip.mahout.Synthetic2DClusteringPrep \
test-data/mahout/synthetic.txt syn-seq

$ hadoop fs -mkdir syn-clusters
```

Create an empty directory in HDFS, where Mahout will generate the random K centroids.

Time to run the clustering:

```
$ export HADOOP_HOME=/usr/lib/hadoop

$ $MAHOUT_HOME/bin/mahout \
kmeans \
  -i syn-seq \
  -c syn-clusters \
  -o syn-kmeans \
  -dm \ 
  org.apache.mahout.common.distance.EuclideanDistanceMeasure \ 
  -x 100 \ 
  -k 20 \ 
  --clustering \ 
  -ow \ 
  -i syn-kmeans/clusters-16-final
```

This will start an iterative sequence of MapReduce jobs until the clusters converge or you hit the maximum number of iterations. When the clustering has completed, there should be a number of directories in HDFS that contain output for each of the MapReduce iterations.

Summary

When the clustering has completed, you can use the `clusterdump` Mahout utility to dump the cluster details of the last job:

```
$ $MAHOUT_HOME/bin/mahout clusterdump -i syn-kmeans/clusters-16-final
```

`clusterdump` writes out a line for each cluster. `VL` indicates that the cluster has converged, and `CL` means that the cluster hasn’t converged.

```
VL-2976{  
n=65
```

The number of data points connected to this cluster, which for this cluster is 65.
The cluster
centroid:
\[
c = \{8966.923, 51193.292\}
\]
\[
r = \{955.286, 1163.688\}
\]

\{VL-2997\{n=88 c=[11394.705, 50557.114] r=[920.032, 1179.291]\}
\{VL-2950\{n=464 c=[39502.394, 42808.983] r=[4022.406, 4273.647]\}
\{VL-2956\{n=900 c=[57117.122, 47795.646] r=[3623.267, 7669.076]\}
\{VL-2963\{n=307 c=[28842.176, 58910.573] r=[2532.197, 2463.770]\}
\{VL-2968\{n=24 c=[12087.458, 59659.125] r=[610.980, 587.461]\}
\{VL-2973\{n=21 c=[9767.762, 60524.619] r=[334.271, 680.851]\}
\{VL-2974\{n=149 c=[17056.611, 54574.094] r=[1424.306, 1499.089]\}
\{VL-2979\{n=15 c=[13094.200, 61833.467] r=[654.127, 769.270]\}
\{VL-2982\{n=152 c=[19948.691, 61123.151] r=[1272.827, 1526.470]\}
\{VL-2983\{n=282 c=[36355.816, 54751.617] r=[1492.798, 4144.325]\}
\{VL-2984\{n=13 c=[8319.385, 58726.923] r=[866.362, 745.068]\}
\{VL-2985\{n=13 c=[10221.231, 62308.692] r=[499.819, 494.718]\}
\{VL-2986\{n=6 c=[11515.167, 63338.667] r=[573.996, 795.132]\}
\{VL-2990\{n=6 c=[8507.333, 63171.167] r=[862.534, 506.489]\}
\{VL-2993\{n=295 c=[27153.434, 45502.441] r=[3910.728, 1400.395]\}
\{VL-2994\{n=19 c=[10641.105, 57988.000] r=[506.490, 1047.906]\}
\{VL-2995\{n=14 c=[8254.714, 60997.643] r=[552.130, 481.433]\}
\{VL-2998\{n=145 c=[4732.103, 54700.310] r=[1308.754, 1182.706]\}
\{VL-2999\{n=22 c=[10977.409, 61125.545] r=[513.230, 369.197]\}

**Final output directory for K-means** In the example here, the K-means algorithm iterated for 16 times before it converged; hence the 16 in the directory name. The number of iterations can vary across executions (because initial centroids are randomly calculated), so you should perform a listing in the syn-kmeans directory in HDFS to determine which directory to use with the mahout clusterdump command (it will be the directory containing the word final). If this directory doesn’t exist, it means that the algorithm wasn’t able to converge before the maximum number of iterations, as defined by the -x argument.

If you superimpose these clusters on top of the original scatterplot, you can see how well the algorithm worked. Note that the clusters will vary based on the initial centroids, so your results will differ from those presented here (see figure 12.16).

Clustering is sensitive to the initial locations of the centroids; if they had been more randomly distributed, the clustering algorithm would have identified more of the clusters.

In this simplified example, you only worked with data in two dimensions, which is easy to visualize and vectorize. Working in two dimensions means that you can only work with two features, because each dimension represents a single feature. If you want to work with \( n \) features, you need \( n \) dimensions, which the Mahout vector classes all support. An example would be working with text, where each unique word is considered a separate feature and therefore a dimension.
12.3.3 K-means and text

Because K-means works with vectorized data, if you want to use K-means with text data (such as clustering on the Reuters news collection), you need to vectorize that data prior to using it with Mahout. How you do this is outside of the scope of this book, but you can look at and execute the $MAHOUT_HOME/examples/bin/build-reuters.sh script (this script is bundled with the Mahout distribution), which takes the Reuters news collection, vectorizes it, and then runs K-means over that data. *Mahout in Action* by Owen et al. (Manning, 2011, www.manning.com/owen/) also contains a section that goes through this exercise in detail.

12.3.4 Other Mahout clustering algorithms

Mahout contains other clustering algorithms, some of which are listed in table 12.2. A complete list can be viewed at http://mahout.apache.org/.
Table 12.2  An overview of Mahout clustering algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hierarchical clustering/Top Down clustering</td>
<td>Hierarchical clustering is the process of finding bigger clusters, as well as the smaller clusters inside the bigger clusters. Top Down clustering is a type of hierarchical clustering. It tries to find bigger clusters first and then does fine-grained clustering on these clusters—hence the name Top Down.</td>
</tr>
<tr>
<td>Canopy clustering</td>
<td>Canopy clustering is a simple, fast, and surprisingly accurate method of grouping objects into clusters. Canopy clustering is often used as an initial step in more rigorous clustering techniques, such as K-means clustering. By starting with an initial clustering, the number of more expensive distance measurements can be significantly reduced by ignoring points outside of the initial canopies.</td>
</tr>
<tr>
<td>Fuzzy K-means</td>
<td>Fuzzy K-means (also called Fuzzy C-means) is an extension of K-means, the popular simple clustering technique. While K-means discovers hard clusters (a point belongs to only one cluster), Fuzzy K-means is a more statistically formalized method and discovers soft clusters where a particular point can belong to more than one cluster with a certain probability.</td>
</tr>
<tr>
<td>Latent Dirichlet Allocation (LDA)</td>
<td>Latent Dirichlet Allocation is a powerful learning algorithm for automatically and jointly clustering words into topics and documents into mixtures of topics.</td>
</tr>
</tbody>
</table>


12.4 Chapter summary

After reading this chapter, you should be able to understand and apply three predictive analytics algorithms: item-based recommenders, naive Bayes classification, and K-means clustering. All these algorithms exist to help you better understand and use data in your systems, whether to help recommend products or services to your users, to classify new data coming into your system (such as email spam detection), or even to find out new facts about your existing data, as in the case of clustering.
It’s always a good time to upgrade your Hadoop skills! Hadoop in Practice, Second Edition provides a collection of 104 tested, instantly useful techniques for analyzing real-time streams, moving data securely, machine learning, managing large-scale clusters, and taming big data using Hadoop.

This completely revised second edition covers changes and new features in Hadoop core, including MapReduce 2 and YARN. You’ll pick up hands-on best practices for integrating Spark, Kafka, and Impala with Hadoop, and get new and updated techniques for the latest versions of Flume, Sqoop, and Mahout. In short, this is the most practical, up-to-date coverage of Hadoop available.

What’s Inside
- Thoroughly updated for Hadoop 2
- How to write YARN applications
- Integrate real-time technologies like Storm, Impala, and Spark
- Predictive analytics using Mahout and RR

Readers need to know a programming language like Java and have basic familiarity with Hadoop.

Alex Holmes works on tough big data problems. He is a software engineer, author, speaker, and blogger specializing in large-scale Hadoop projects.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/HadoopinPracticeSecondEdition

Very insightful. A deep dive into the Hadoop world.
—Andrea Tarocchi, Red Hat, Inc.

The most complete material on Hadoop and its ecosystem known to mankind!
—Arthur Zubarev, Vital Insights

Clear and concise, full of insights and highly applicable information.
—Edward de Oliveira Ribeiro
DataStax, Inc.

Comprehensive up-to-date coverage of Hadoop 2.
—Muthusamy Manigandan
OzoneMedia

MANNING $49.99 / Can $52.99 [INCLUDING eBook]