Implement top-notch machine learning algorithms for classification, clustering, and recommendations with Apache Mahout

Apache Mahout Essentials

Jayani Withanawasam

What you will learn from this book

- Get started with the fundamentals of Big Data, batch, and real-time data processing with an introduction to Mahout and its applications
- Understand the key machine learning concepts behind algorithms in Apache Mahout
- Apply machine learning algorithms provided by Apache Mahout in real-world practical scenarios
- Implement and evaluate widely-used clustering, classification, and recommendation algorithms using Apache Mahout
- Discover tips and tricks to improve the accuracy and performance of your results
- Set up Apache Mahout in a production environment with Apache Hadoop
- Glance at the Spark DSL advancements in Apache Mahout 1.0
- Provide dynamic and interactive data visualizations for Apache Mahout
- Build a recommendation engine for real-time use cases and use user-based and item-based recommendation algorithms

Who this book is written for

If you are a Java developer or data scientist, haven’t worked with Apache Mahout before, and want to get up to speed on implementing machine learning on big data, this is the perfect guide for you.


$24.99 US
£16.99 UK

Prices do not include local sales tax or VAT where applicable


Free Sample
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 4 'Recommendations'
- A synopsis of the book’s content
- More information on Apache Mahout Essentials
Jayani Withanawasam is R&D engineer and a senior software engineer at Zaizi Asia, where she focuses on applying machine learning techniques to provide smart content management solutions.

She is currently pursuing an MSc degree in artificial intelligence at the University of Moratuwa, Sri Lanka, and has completed her BE in software engineering (with first class honors) from the University of Westminster, UK.

She has more than 6 years of industry experience, and she has worked in areas such as machine learning, natural language processing, and semantic web technologies during her tenure.

She is passionate about working with semantic technologies and big data.
Apache Mahout is a scalable machine learning library that provides algorithms for classification, clustering, and recommendations.

This book helps you to use Apache Mahout to implement widely used machine learning algorithms in order to gain better insights about large and complex datasets in a scalable manner.

Starting from fundamental concepts in machine learning and Apache Mahout, real-world applications, a diverse range of popular algorithms and their implementations, code examples, evaluation strategies, and best practices are given for each machine learning technique. Further, this book contains a complete step-by-step guide to set up Apache Mahout in the production environment, using Apache Hadoop to unleash the scalable power of Apache Mahout in a distributed environment. Finally, you are guided toward the data visualization techniques for Apache Mahout, which make your data come alive!

What this book covers

Chapter 1, *Introducing Apache Mahout*, provides an introduction to machine learning and Apache Mahout.

Chapter 2, *Clustering*, provides an introduction to unsupervised learning and clustering techniques (K-Means clustering and other algorithms) in Apache Mahout along with performance optimization tips for clustering.

Chapter 3, *Regression and Classification*, provides an introduction to supervised learning and classification techniques (linear regression, logistic regression, Naïve Bayes, and HMMs) in Apache Mahout.
Chapter 4, *Recommendations*, provides a comparison between collaborative- and content-based filtering and recommenders in Apache Mahout (user-based, item-based, and matrix-factorization-based).

Chapter 5, *Apache Mahout in Production*, provides a guide to scaling Apache Mahout in the production environment with Apache Hadoop.

Chapter 6, *Visualization*, provides a guide to visualizing data using D3.js.
In this chapter, we will cover the recommendation techniques used in Apache Mahout. We will discuss the related MapReduce- and Spark-based implementations with respect to a real-world example, with Java code examples as well as command-line executions.

In this chapter, we will cover the following topics:

- Collaborative versus content-based filtering
- User-based recommenders
- Data models
- Similarity
- Neighborhoods
- Recommenders
- Item-based recommenders with Spark
- Matrix factorization-based recommenders
  - SVD recommenders
  - ALS-WS
- Evaluation techniques
- Recommendation tips and tricks

"A lot of times, people don't know what they want until you show it to them."

– Steve Jobs
Before we proceed with the chapter, let's think about the significance of the preceding quote for a moment.

- How many times have you come across relevant items to buy, which were suggested by Amazon recommendations?
- How many times have you found your friends when suggested by Facebook, which you did not notice earlier?
- How many videos have you watched when recommended by YouTube, which you ultimately found to be useful later?

The volume of available information is growing at an alarming rate. By 2014, Facebook had around 1.35 billion active users. The number of electronic items itself is around 24 million in Amazon. There are approximately 9,796 movies on Netflix.

Consequently, filtering out relevant information is essential to make the right decisions and realize new information.

**Collaborative versus content-based filtering**

There are two main approaches you can take when it comes to filtering information.

**Content-based filtering**

Content-based filtering is an unsupervised mechanism based on the attributes of the items and the preferences and model of the user.

For example, if a user views a movie with a certain set of attributes, such as genre, actors, and awards, the systems recommend items with similar attributes. The preferences of the user (for example, previous "likes" for movies) are mapped with the attributes or features of the recommended item.

User ratings are not required in this approach. However, this approach requires considerable effort when it comes to feature or attribute extraction, and it is also relatively less precise than collaborative filtering approaches, which we will discuss later.
Collaborative filtering

Collaborative filtering approaches consider the notion of similarity between items and users. The features of a product or the properties of users are not considered here, as in content-based filtering.

As shown in the following image, you must be familiar with statements such as "people who bought this item also bought..." and "people who viewed this item also viewed...", which represent the collaborative filtering approach in real-world applications.

In collaborative filtering, for each item or user, a neighborhood is formed with similar related items or users. Once you view an item, recommendations are drawn from that neighborhood.

The collaborative filtering approach uses historical data on user behavior, such as clicks, views, and purchases, to provide better recommendations.

Collaborative filtering can be achieved using the following techniques.

- Item-based recommendations
- User-based recommendations
- Matrix factorization-based recommendations

Apache Mahout has implemented machine learning algorithms to enable collaborative filtering approaches. So, in this chapter, we will mainly discuss this approach as opposed to content-based approaches.
Hybrid filtering
If both content-based and collaborative filtering approaches are used at the same time to provide recommendations, then the approach is known as hybrid filtering.

User-based recommenders
In user-based recommenders, similar users from a given neighborhood are identified and item recommendations are given based on what similar users already bought or viewed, which a particular user did not buy or view yet.

For example, as shown in the following figure, if Nimal likes the movies *Interstellar* (2014) and *Lucy* (2014) and Sunil also likes the same movies, and in addition, if Sunil likes *The Matrix* (1999) as well, then we can recommend *The Matrix* (1999) to Nimal, as the chances are that Nimal and Sunil are like-minded people.
A real-world example – movie recommendations

Let's explain this approach using a real-world example on a movie recommendation site, as shown in the following figure:
Recommendations

Users who watched the movies (items) rated them according to their preferences. The rating is a value between 1 (lowest) and 10 (highest).

The user, item, and preferences (ratings) information is given in the following table; you need to save this data as movies.csv in order to execute the example that follows:

<table>
<thead>
<tr>
<th>User</th>
<th>Item</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>
The Java code example for user-based recommendations is given as follows:

```java
DataModel model = new FileDataModel(new File("movie.csv"));

UserSimilarity similarity = new PearsonCorrelationSimilarity(model);

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

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

List<RecommendedItem> recommendations = recommender.recommend(3, 2);

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

In this example, we need to get the top two recommendations for **user 03 (Nimal)**. User 03 liked the following movies, with the given ratings in brackets:

```
User 3 > Item 1 (5), Item 4 (8), Item 5 (9), Item 7 (10)
```

Here, we can see that **user 04** and **user 05** share some similar interests (**Item 01 and Item 04**) with **user 03**. So, we assume that **user 03** might like the other items that **user 04** and **user 05** likes, which **user 03** has not tried out before. Accordingly, we recommend **Item 06** and **Item 03**.

```
User 4 > Item 1, Item 4, Item 6 (8), Item 3 (6)
User 5 > Item 1, Item 2, Item 3 (4), Item 4, Item 5, Item 6 (8)
```

This shows the result of the code example:

```
RecommendedItem[item:6, value:8.0]
RecommendedItem[item:3, value:5.181073]
```

The value of **item:6** is higher than that of **item:3** because both **User 4** and **User 5** have rated **item:6** higher.

Even though **user 01** and **user 02** have some common interests (**Item 01 and Item 04**) with **user 03**, they are not considered due to low ratings given to these co-occurring items.

When we walk-through this code, we learn a few significant abstractions that have been provided by Apache Mahout.
Recommendations

Data models
A data model represents how we read data from different data sources. In our code example, we used FileDataModel, which takes comma-separated values (CSV) as input.

In addition, Apache Mahout supports the following input methods:

- JDBCDataModel: This method reads from the JDBC driver
- GenericDataModel: This is populated through Java calls
- GenericBooleanPrefDataModel: This uses the given user data, which is suitable for small experiments

The following is the code example with GenericDataModel:

```java
// Preferences for all users
FastByIDMap<PreferenceArray> userData=new FastByIDMap<PreferenceArray>();

// Preferences for user 1
List<Preference> prefsUser1=new ArrayList<Preference>();
prefsUser1.add(new GenericPreference(1,1,10));
prefsUser1.add(new GenericPreference(1,4,9));

// Preferences for user 2
List<Preference> prefsUser2=new ArrayList<Preference>();
prefsUser2.add(new GenericPreference(2,1,10));

// Preferences for user 3
List<Preference> prefsUser3=new ArrayList<Preference>();
prefsUser3.add(new GenericPreference(3,1,10));
prefsUser3.add(new GenericPreference(3,2,8));
prefsUser3.add(new GenericPreference(3,3,5));

// Add preferences for all users
userData.put(1,new GenericUserPreferenceArray(prefsUser1));
userData.put(2,new GenericUserPreferenceArray(prefsUser2));
userData.put(3,new GenericUserPreferenceArray(prefsUser3));

DataModel model = new GenericDataModel(userData);
// Get possible unseen preferred items for user 2
CandidateItemsStrategy strategy=new SamplingCandidateItemsStrategy(1,1);
FastIDSet candidateItems=strategy.getCandidateItems(2,new GenericUserPreferenceArray(prefsUser2),model);
```
for (Long candidateRecommendation : candidateItems) {
    System.out.println("Candidate item: " +
        candidateRecommendation);
}

The `userData` variable contains preferences of all the users for the items available along with their ratings. `PreferenceArray` is a memory-efficient way of having a collection of preferences for each user. `Preference` contains a user's preference for an item with the preference value (rating). You can filter out the considered items and users for result candidate items using `CandidateItemStrategy`.

The result for the preceding code example is given as follows:

    Candidate item: 2
    Candidate item: 3
    Candidate item: 4

As you can see from the result, items 2, 3, and 4 are recommended for user 2, as these are the items that user 2 has not rated but users with similar preferences to user 2 (user 1 and user 2) have rated.

**The similarity measure**

Similarity represents the similarity between two users in user-based recommendations or the similarity between two items in item-based recommendations.

Two users can be considered to be similar if the distance or angle between them is small in the user–item space. An example of this is shown in the following figure:
Recommendations

In our example, we have used the PearsonCorrelationSimilarity measure to find similarity between two users. Some other available similarity measures are listed as follows:

- **EuclideanDistanceSimilarity**: This measures the Euclidean distance between two users or items as dimensions and preference values given will be values along those dimensions. EuclideanDistanceSimilarity will not work if you have not given preference values.

- **TanimotoCoefficientSimilarity**: This is applicable if preference values consist of binary responses (true and false). TanimotoCoefficientSimilarity is the number of similar items two users bought or the total number of items they bought.

- **LogLikelihoodSimilarity**: This is a measure based on likelihood ratios. The number of co-occurring events, in this context, the number of times either users or items that occurred together and the number of times either users or items that do not occur together, are considered for evaluating similarity.

- **SpearmanCorrelationSimilarity**: In SpearmanCorrelationSimilarity, the relative rankings of preference values are compared instead of preference values.

- **UncenteredCosineSimilarity**: This is an implementation of cosine similarity. The angle between two preference vectors is considered for calculation.

When defining similarity measures, you need to keep in mind that not all datasets will work with all similarity measures. You need to consider the nature of the dataset when selecting a similarity measure.

Also, to determine the optimal similarity measure for your scenario, you need to have a good understanding of the dataset.

Trying out different similarity measures with your training dataset is essential to find the optimal similarity measure.

The neighborhood

Using the selected neighborhood algorithm, we can compute the "neighborhood" of users for a given user. This neighborhood can be used to compute recommendations.
An example of a movie recommendation scenario is given in the following figure:

We have used the nearest neighbour algorithm in the preceding example.

- **Nearest neighbour algorithm**: This calculates a neighborhood comprising of the nearest $N$ users to a given user

- **ThresholdUserNeighborhood**: This calculates a neighborhood comprising of all the users whose similarity to the given user is the same as or outstrips a given threshold

**Recommenders**

Given the data model neighborhood and similarity, the selected Apache Mahout recommender estimates similar items for unseen or new items for the user.

Different recommenders such as `GenericUserBasedRecommender`, `GenericItemBasedRecommender`, and `SVDRecommender` are explained in this chapter in detail.

**Evaluation techniques**

A good recommender should be able to infer the items that users are likely to be interested in.
Recommendations

There are two different methods to evaluate the quality of a recommender for a given dataset, as follows:

- Prediction-based
- Information Retrieval (IR-based)

During the evaluation, the dataset is split into training and test datasets. The training dataset is used to create the model, and evaluation is done based on the test dataset. In the following example, 0.8 is given as the evaluation percentage.

**The IR-based method (precision/recall)**

The code example to evaluate the recommender with the IR-based method is given here.

In the recommendation context, **Precision** denotes the fraction of top recommendations that are relevant recommendations. **Recall** denotes the fraction of relevant recommendations that appear in the top recommendations:

```java
DataModel model = new FileDataModel (new File("movie.csv"));

RecommenderIRStatsEvaluator evaluator = new GenericRecommenderIRStatsEvaluator();
RecommenderBuilder builder = new RecommenderBuilder() {
    public Recommender buildRecommender(DataModel model) throws TasteException {
        UserSimilarity similarity = new LogLikelihoodSimilarity(model);
        UserNeighborhood neighborhood = new NearestNUserNeighborhood(5, similarity, model);
        return new GenericUserBasedRecommender(model, neighborhood, similarity);
    }
};

IRStatistics stats = evaluator.evaluate(builder, null, model, null, 2, GenericRecommenderIRStatsEvaluator.CHOOSE_THRESHOLD, 0.8);
System.out.println("Precision: " + stats.getPrecision()+" Recall:");
} + stats.getRecall());
```
Addressing the issues with inaccurate recommendation results

The accuracy of the recommendation results mainly relies on finding the right similarity measure and the right neighborhood algorithm that fits the dataset in hand well.

If the dataset is sparse (if there are large number of items and less number of user preferences), PearsonCorrelationSimilarity will provide a better solution than UncenteredCosineSimilarity.

The nearest neighbor graph and the distance similarity distribution can be used to examine the similarity measure further.

Distance distribution is based on measures such as average neighbor similarity, neighbor similarity ratio, and neighbor stability, whereas the nearest neighbor graph is based on average graph distance, clustering coefficient, graph density, graph diameter, and maximum graph distance.

Moreover, a threshold can be used to filter out the inaccurate results. If values are missing, we can introduce some default values or eliminate that data.

Item-based recommenders

An item-based recommender measures the similarities between different items and picks the top k closest (in similarity) items to a given item in order to arrive at a rating prediction or recommendation for a given user for a given item.

For the movie recommendation scenario, an item-based recommender works as given in the following figure:
Let’s say both Sunil and Roshan like the movies *Interstellar (2014)* and *Star Wars (1977)*. Then, we can infer that *Interstellar (2014)* and *Star Wars (1977)* could be similar items. So, when Nimal likes *Interstellar (2014)*, we recommend *Star Wars (1977)* to Nimal based on our previous observation.

The following is the Java code example for item-based recommenders:

```java
DataModel model = new FileDataModel (new File("movie.csv"));

ItemSimilarity itemSimilarity = new EuclideanDistanceSimilarity (model);

Recommender itemRecommender = new
GenericItemBasedRecommender(model,itemSimilarity);

List<RecommendedItem> itemRecommendations =
  itemRecommender.recommend(3, 2);

for (RecommendedItem itemRecommendation : itemRecommendations) {
    System.out.println("Item: " + itemRecommendation);
}
```

In the preceding code example, we recommend two items for user 3. The result is given as follows:

```
Item: RecommendedItem[item:2, value:7.7220707]
Item: RecommendedItem[item:3, value:7.5602336]
```
Item-based recommenders with Spark

An item-based recommender can also be executed on top of Spark. We have given the steps to set up the Spark server in Chapter 3, Regression and Classification in detail.

1. Start the Spark servers:
   ```bash
   [SPARK_HOME]/sbin/start-all
   ```
2. Prepare the input data (only the user ID and item ID, no preference values).
3. Copy the input data to HDFS.
4. Execute the following mahout command to generate recommendations for each item:
   ```bash
   mahout spark-itemsimilarity --input inputfile --output outputdirectory
   ```
   
   *movie.csv* can be used as *inputfile*, or else you can give your own data file as well.

   The generated indicator matrix is given in the following figure; this can be found in the *outputdirectory/indicator-matrix/* directory:

   ```
   5:1.184939225613802 6:1.184939225613802 7:1.184939225613802 8:0.5053430784314124 9:0.5053430784314124
   5:2.231495513142097 4:0.8563430784314124
   7:2.231495513142097 3:2.231495513142097 4:1.184939225613802 5:0.13844293988399518 6:0.13844293988399518 2:0.13844293988399518
   4:1.184939225613802 7:1.184939225613802 5:0.13844293988399518 6:0.13844293988399518
   4:1.184939225613802 7:1.184939225613802 5:0.13844293988399518 6:0.13844293988399518
   3:2.231495513142097 4:1.184939225613802 5:0.13844293988399518 6:0.13844293988399518
   2:2.231495513142097 7:2.231495513142097 6:1.184939225613802 4:0.5053430784314124
   ```

   Similar items for each other item are given in the indicator matrix, with a similarity value according to the following format:

   ```
   itemIDx itemIDy:valuey itemIDz:valuez
   ```

Matrix factorization-based recommenders

So far, we have discussed two main collaborative filtering approaches, namely user-based and item-based recommenders.

Even though they are capable of providing users with relevant recommendations, a major challenge that these approaches face is the sparsity of large datasets. Not all users will provide ratings on all the available items. Also, new items and new users tend to lack sufficient historical data to predict good recommendations. This is known as the **cold start problem**.
Recommendations

Further, the requirement for scalable recommendation algorithms remains the same along with the requirement to perform well in sparse datasets.

Also, some users tend to have a bias toward ratings, and the previous approaches have not made an attempt to correct this bias. Also, hidden patterns between the features of available items and the features of users that lead to certain ratings are not exploited.

Matrix factorization is another way of doing collaborative filtering, which is intended to solve the previously mentioned problems.

Ratings can be induced by certain imperceptible factors, which are not straightforward for us to estimate, so we need to use mathematical techniques to do that for us.

For example, if a particular user has rated The Matrix, the influence factor to do this can be the "amount of sci-fi involved in the movie." If the same user has rated Titanic, then the influence factor can be the "amount of romance involved in the movie." So, using matrix factorization methods, we can recommend Her movie to that particular user by inferring these factors (latent factors).

The following figure denotes an example feature space of the preceding scenario. However, factors are not necessarily intuitively understandable by humans. The distance between each movie can be used to relate those movies.
In Apache Mahout, there are a few methods on which matrix factorization can be done, as follows:

- Alternative Least Squares with Weighted-Lamda-Regularization (ALS-WS)
- SGD

SVD++ is a combination of matrix factorization and the latent factor model.

**Alternative least squares**

ALS-WS is one of the factorizers that can be used to generate recommendations, which is inherently parallel.

Mahout's ALS recommender is a matrix factorization algorithm that uses ALS-WR.

**Singular value decomposition**

Using *Singular Value Decomposition* (SVD), we can come up with a more generalized set of features to represent the user-item preferences for a large dataset using dimensionality reduction techniques. This approach helps to generalize users into lesser dimensions.

The following is the Java code example for SVD using ALS-WR as the factorizer; the number of target features should be given as input, which in this case (3, 0.065) is given as lambda (the regularization parameter), and the number of iterations is given as 1:

```java
DataModel svdmodel = new FileDataModel(new File("movie.csv"));

ALSWRFactorizer factorizer = new ALSWRFactorizer(svdmodel, 3, 0.065, 1);

Recommender svdrecommender = new SVDRecommender(svdmodel, factorizer);
for (RecommendedItem recommendation : svdrecommender.recommend(3, 1))
{
    System.out.println(recommendation);
}
```

The following is the output of the preceding code:

```java
RecommendedItem[item:3, value:7.2046385]
```
Recommendations

The following is the command-line execution of the ALS-WS algorithm; the input and output directories should be available in HDFS:

```
mahout parallelALS --input alsmovieinput --output alsmovieoutput --lambda 0.1 --implicitFeedback true --alpha 0.8 --numFeatures 2 --numIterations 5 --numThreadsPerSolver 1 --tempDir tmp
```

In this approach, the user-to-item matrix is factored into the user-to-feature matrix (U) and the item-to-feature matrix (M), as shown in the following figure:

![Matrix Factorization](image)

Then, we can get use the following command to get only the topmost recommendation for each item:

```
mahout recommendfactorized --input alsmovieinput --userFeatures alsmovieoutput/U/ --itemFeatures alsmovieoutput/M/ --numRecommendations 1 --output recommendations --maxRating 1
```

The following figure shows the outcome of the preceding command:

```
<table>
<thead>
<tr>
<th></th>
<th>rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6.07240428</td>
</tr>
<tr>
<td>2</td>
<td>6.0771763</td>
</tr>
<tr>
<td>3</td>
<td>6.03447142</td>
</tr>
<tr>
<td>4</td>
<td>2.8048554</td>
</tr>
<tr>
<td>5</td>
<td>7.016289039</td>
</tr>
</tbody>
</table>
```

The resulting information can be embedded in a search engine, such as Apache Solr, to provide better search recommendations.

Algorithm usage tips and tricks

The optimal recommendation algorithm depends on the nature of data and the scenario in hand.

However, if you have fewer users than items, then it is better to use user-based recommendations. In contrast, if you have fewer items than users, then it is better to use item-based recommendations to gain better performance.

In SVD algorithms, preprocessing can be slow. However, execution is faster than in other methods.
Summary
The Apache Mahout recommendations module helps you to recommend items to users which they have not seen before, based on their previous preferences. The collaborative filtering approach is implemented in Mahout. User-based recommendations, item-based recommendations, and matrix factorization are the key approaches that are geared toward collaborative filtering in Mahout.
Where to buy this book

You can buy Apache Mahout Essentials from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.