Using a recommendation engine to personalize your web application

Enhancing the user experience with Apache Mahout and WebSphere Application Server

Emilio F Zegarra
Tony Efremenko

September 21, 2011

Most businesses are interested in finding new ways to drive traffic and generate revenue from their online investments. However, increasing the amount of information and services that are available online makes it challenging for a business to make such information available to customers in a quick and personalized way. One way to address this challenge is to use a recommendation engine, which can drive visitors to your web site to explore further offerings. These engines apply a variety of patterns and analyze user habits to offer recommendations to users, and can be helpful in presenting offerings that a user might not otherwise know about. This article explains how to integrate Apache Mahout, an open source recommendation engine, with IBM® WebSphere® Application Server V8.0 and IBM Rational® Application Developer for WebSphere Software V8.0.3.

Introduction

To stay relevant in a fast paced, global industry, technical professionals must keep track of the big trends in IT and find ways to incorporate the important ones in their company’s technology portfolio. One such trend is the use of recommendation engines to drive users to explore further offerings from your web site or business. These engines provide recommendations to users based on a variety of patterns, and are helpful in guiding users to consider offerings that they might not otherwise be aware of, based on their specific user habits.

Some very popular web sites make extensive use of recommendation engines. Visitors to Amazon or Netflix, for example, often see personalized recommendations phrased something like, “If you liked that item, you might like also like this one...” These sites use recommendations to help drive users (and revenue) to other things they offer in an intelligent, meaningful way, tailored specifically to the user and the user’s preferences.

Even if your business doesn’t offer books or movies, there could be plenty of reasons for implementing something similar. You can recommend related products that your business...
provides, especially if you have a broad portfolio of offerings. You can offer recommendations for more abstract concepts, such as relevant pages you’d like the user to visit, a list of popular services, applicable educational opportunities, special offers, or access to helpful technical support documents.

Instead of trying to guess what your broad user base is interested in, personalization by means of recommendations enables you to identify the likes and dislikes of individual users unobtrusively and intelligently, and use this information to customize each user’s experience. The task of recommending a new choice based on past behavior is one that has broad uses across many applications and industries, and so this is the example that will be referred to throughout this article.

Apache Mahout is an open source recommendation engine that provides a good application of these concepts, and is small but powerful enough to implement in small to medium business cases. This article outlines how to integrate Apache Mahout V0.5 with IBM WebSphere Application Server V8 using IBM Rational Application Developer for WebSphere Software V8.0.3. It begins with an overview of recommendation engines, describes Apache Mahout at a high level, explains how you can integrate it with WebSphere Application Server and Rational Application Developer, and then offers some next steps for finding out more about this technology.

**Overview of a recommendation engine**

The main purpose of a recommendation engine is to make inferences on existing data to show relationships between objects. Objects can be many things, including users, items, products, and so on. Relationships provide a degree of likeness or belonging between objects. For example, relationships can represent ratings of how much a user likes an item (scalar), or indicate if a user bookmarked a particular page (binary).

To make a recommendation, recommendation engines perform several steps to mine the data. Initially, you begin with input data that represents the objects as well as their relationships. Input data consists of object identifiers and the relationships to other objects. Figure 1 shows this at a high level.
Consider the ratings users give to items. Using this input data, a recommendation engine computes a similarity between objects. Computing the similarity between objects can take a great deal of time depending on the size of the data or the particular algorithm. Distributed algorithms such as Apache Hadoop can be used to parallelize the computation of the similarities. There are different types of algorithms to compute similarities. Finally, using the similarity information, the recommendation engine can make recommendation requests based on the parameters requested.

It could be that you believed recommendation engines were helpful, but stayed away because you thought they were too complicated to try. The recommendation engine domain is in fact large and can be very complex. Fortunately, there are tools available that make it easy to show the necessary concepts within the time and space of a single article. Even better, once taught, those same tools can be used to apply the lesson to the real world. The designers of Apache Mahout made scalability and availability a central part of the project, so you can build out your solution as your needs expand.

There are a few important decisions to make when you decide to start personalizing your application and want to use recommendation engine:

- **Which algorithm to apply**
  The most important decision you'll need to make is what algorithm to apply to your data. The selection of the algorithm depends on what you want to identify and what type of relationship is specified in your data. Some of the common approaches used for recommendations include:
  - **Collaborative filtering**: This approach relies on the social interaction between users. The recommendations are based on rankings provided by other users.
- **Clustering:** With this approach, the recommendation engine tries to build recommendations based on the similarities between either the users or the items themselves.
- **Categorization:** This approach automatically groups items together into categories using common attributes. In categorization, the computer attempts to classify all the items.

This article will focus on collaborative filtering to help you learn about the social aspects of your users; this is also a good starting point for adding recommendations to Web applications. Collaborative filtering is an easy and popular technique. It's easy because your customers do the important work for you – they drive the criteria of what you want to highlight. Collaborative filtering analyzes ratings from other users or items to make recommendations. There are two approaches within collaborative filtering: the main difference between them lies in the ability of each to scale as the number of users in the system grows:

- **User-based recommendation**
  This type of recommendation builds similarities between users by looking at the commonalities of the items rated by each user. For example, if the items are courses, two users could be considered very similar if they both took the same courses. In the other extreme, their similarity would be low if they did not take any similar course. To make recommendations, the algorithms rely on the ratings that similar users gave to those courses not taken by the user. This recommendation is the most basic one; however, its main limitation is that in order to generate the similarities, it needs to compare each user to every other user. This is acceptable for an application with a low number of users, but if the number of users increases, the time to perform this evaluation increases exponentially.

- **Item-based recommendation**
  Item-based recommendation, on the other hand, begins by looking at the items that are associated with the user. For each item associated with the user, the algorithm computes how similar it is to the other items in the collection to build the list of recommendations. In order to determine how likely the user is to like a recommended item, the algorithm looks at the ratings that the user has given to the item and gives a weighted rating to each recommended item. The main issue with item-based recommendation is that it needs to build a similarity index for every available item. Changes in the items, however, are less frequent than changes in users and, therefore, it is feasible with this type of recommendation to pre-compute similarities offline and update them at specific periods.

- **How to scale the process out to your users**
  The actual process of calculating similarity between users and items is a process-intensive operation. Depending on the size of your data set, the operation could take a few milliseconds to several minutes. When working with Web-based applications, response time becomes an issue if users need to wait for an extended period of time to receive a recommendation. While it is possible to compute similarities between users and items in real time, you need to evaluate this carefully when you work with larger datasets. For both user- and item-based recommendation approaches, best practices recommend performing these calculations offline if the data sets are larger (for example, over 1 million ratings). Using offline calculations becomes feasible in cases where new items are added infrequently or users rate items only
occasionally, because similarities only need to be computed when a new item is added or a new rating is applied to an item. In such as scenario, the recommendation engine can work using similarities that would be eventually consistent.

One approach for pre-computing similarities offline is by leveraging the distributed computation capabilities of products such as Apache Hadoop, an open source implementation of the MapReduce technique. This is why you'll often hear about those systems mentioned along with recommendation engines. If you're trying to recommend based on a vast, scattered array of data, you'll need to boil the data down, and these systems enable you to do that. Fortunately, Apache Mahout provides jobs that can be submitted to Apache Hadoop to help you compute your similarities. Once this calculation is complete, you can load the results into your desired data source so that your Web application can make use of it.

**Where to store your data**

Finally, you need to determine where to store your data. This data can contain the raw input data or the data with the similarities already computed by an offline process, such as Apache Hadoop. If the source of your data is a vast archive of raw data, you might need to mine it to get something to feed the recommendation engine. You can store your data sets in a file system or in a distributed data source. In the case where your data sets are small, you can have your programs read the data from the file system and store it in working memory. However, if the data sets are large, you might want to consider using a database management system such as IBM DB2®, Apache Derby, and so on. If you select a distributed data source, you will want to ensure that proper query optimization settings (such as indexes) are properly configured.

But it doesn't have to be that complicated. To keep things simple, assume here that your pool of data is small enough to fit into a small database, such as Apache Derby, the Java™-based open source database management system shipped with WebSphere Application Server V8. The important thing is that when moving forward in your approach, you'll need to determine, based on your particular data, whether you will need to make use of a distributed file system or a traditional relational database management system.

**Sample scenario**

As described in the developerWorks article *Introducing Apache Mahout*, the goal of the Apache Mahout project is to build scalable machine learning libraries. Apache Mahout is implemented on top of Apache Hadoop but is not restricted to distributed file systems.

This brings us to the focus of this article, namely the machine-learning algorithms provided by Apache Mahout to process your data into a recommendation. For purpose of this article, we'll focus on the user-based filtering machine-learning task that Apache Mahout currently implements. Social references are used in this example because there are many ways to get this data and the data is simple to log into a database.

Assume that you want to create a Web application that enables users to get item recommendations based on ratings provided by other users. The data set available contains ratings made by users about items. For simplicity, the sample data included with this article is generic, as we will only work with the identifiers of the users and the items. In a real world
Using a recommendation engine to personalize your web application, you will want to replace the unique identifier of a recommended item by its display name before presenting to user.

**Figure 2. Sample topology**

![Diagram of topology]

So, as shown in Figure 2, users would request a servlet in the application. The application will make a call to the recommendation engine to come up with a set of recommendations for the user. The recommendation engine will retrieve the data from a data source and calculate the similarities in real time.

**Configure the development environment**

To perform the steps outlined in this article, you will need to set up your development environment as described here:

A. **Download Apache Mahout**
   1. Visit the [Apache Mahout web site](http://mahout.apache.org) and download the latest version of Apache Mahout. As the time of this writing, the latest version was 0.5 (mahout-distribution-0.5.zip).
   2. Extract the contents of the archive to a known location. These content files will be referenced later.

B. **Create the Java EE application project**
   1. Start Rational Application Developer for WebSphere Software V8.0.3.
   2. Switch to the Java EE perspective.
   3. Select **File > New > Enterprise Application Project**.
   4. For Project Name, enter **RecommenderApp**.
   5. Ensure the target runtime is set to **WebSphere Application Server v8.0** and click **Next**.
   6. On the next panel, click **New module...** and from the popup, select **Web module only** and set its name to **RecommenderWeb**.
   7. Click **OK** and then **Finish**.

C. **Create and populate database with sample data**

Apache Derby is a Java-based database that uses a file store for storage. Apache Derby is used in this example because it is included with Rational Application Developer.

   1. Select **Window > Show View > Data Source Explorer**.
   2. Right click on **Database Connections** and select **New ...**
   3. For JDBC driver, select **Derby 10.5 – Embedded JDBC Driver Default**.
   4. Since Derby stores databases in the file system, you need to specify where it will reside.
      For Database location, enter the path and name to use for the database. For this sample, **PREFERENCES** is used as the as the database name.
   5. Leave the username and password fields blank and click **Finish**.

   Next, you will define your data model using a script (Listing 1). The script first creates a schema called **PREFERENCES** and a table called **taste_preferences**. This table holds all the
ratings users make about each item. It contains four columns: user_id, item_id, preferences and timestamp. Each tuple in the table represents that user user_id has ranked item item_id and given it a rating of preference.

**Listing 1. Data source schema**

```sql
CREATE SCHEMA PREFERENCES;
CREATE TABLE PREFERENCES.taste_preferences
(
    user_id BIGINT NOT NULL,
    item_id BIGINT NOT NULL,
    preference FLOAT NOT NULL,
    "timestamp" BIGINT,
    PRIMARY KEY (user_id, item_id)
);
```

Because of the large number of accesses required to the database, it is very important to also define two indexes to speed up search time (Listing 2).

**Listing 2. Table indexes SQL**

```sql
CREATE INDEX PREFERENCES.user_id_idx ON PREFERENCES.taste_preferences ( user_id );
CREATE INDEX PREFERENCES.item_id_idx ON PREFERENCES.taste_preferences ( item_id );
```

Next, create the data model:

6. Download createtable.sql *(included with this article)* and save it within the RecommenderWeb project.
7. From the Java EE perspective, right click on the RecommenderWeb project and select Refresh. The createtable.sql script should now show up.
8. Right click on the createtable.sql script and select Execute SQL Files.
9. Verify server type is set to Derby _10.x* and the connection profile and Database names are correct and click OK.
10. You can verify that the script completed successfully by reviewing the SQL Results view (Figure 3).

**Figure 3. SQL Results view after table creation**

Now that you have your table created, you can load it with some data:
11. Download file **u.data (included with this article)**. This is a comma-separated list of about 10K user ratings from the MovieLens data set. As you did with the SQL script, save this file to the RecommenderWeb project.

12. From the Java EE perspective, right click on the **RecommenderWeb** project and select **Refresh**. The u.data file should now display.

13. From the Data Source Explorer view, expand **Database Connections > PREFERENCES > PREFERENCES > Schemas > PREFERENCES > Tables**.

14. Right click on table **TASTE_PREFERENCES** and select **Load...**

15. For Input File, click **Browse** and navigate to the **RecommenderWeb** folder and select **u.data** and click **OK**.

16. For Column delimiter, select **Tab**. Ensure the **Replace existing data** is checked and click **Finish**.

17. The load process should now start. It could take up to 60 seconds for the entire dataset to be loaded. You can verify the successful completion of the loading from the SQL Results view (Figure 4).

**Figure 4. SQL Results view after data loaded into database**

18. You can verify the data loaded by right clicking on the **TASTE_PREFERENCES** table and selecting **Data > Sample Contents**.

19. Disconnect from the database by selecting the **Data Source Explorer** view and expanding **Database Connections**.

20. Right click on **PREFERENCES** and select **Disconnect**.

You have now completed creating your data storage and your sample data.

**D. Configure the Apache Mahout libraries**

In order to develop the recommender code, you need to import the required Apache Mahout libraries into your enterprise application. (If you are going to share the Apache Mahout libraries among multiple enterprise projects in your environment, then the recommended method would be to configure a shared library.)

1. Expand **EAR Projects**, right click on **RecommenderApp**, and select **Import > Import** ...

2. Select **General > File System** and click **Next**.

3. In From directory, browse to the location where you extracted the Apache Mahout files and click **OK**.

4. Select the files listed in the table below from the import dialog:

<table>
<thead>
<tr>
<th>Library name</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>mahout-core-0.5.jar</td>
<td>mahout-distribution-0.5/</td>
</tr>
<tr>
<td>mahout-core-0.5-job.jar</td>
<td>mahout-distribution-0.5/</td>
</tr>
<tr>
<td>slf4j-jcl-1.6.0.jar</td>
<td>mahout-distribution-0.5/lib</td>
</tr>
</tbody>
</table>
5. Click **Finish**.
6. Now add these references to the Web application so you can define your class path for compilation. Right click on the **RecommenderWeb** project and select **Properties**.
7. Select **Java Build Path** and click **Add JARs ...**
8. From the popup dialog, expand **RecommenderApp**, select **mahout-core-0.5.jar** and **mahout-core-0.5-job.jar** and click **OK** (Figure 5).

**Figure 5. Java Build Path editor for the RecommenderWeb project**

9. From the RecommenderWeb project, expand **WebContent > META-INF** and double click on **MANIFEST.MF**.
10. Under **Dependencies > Jar or Module** place a checkmark next to all three libraries (Figure 6).

**Figure 6. Dependencies editor for the RecommenderWeb project**

11. Save and close the editor.
At this point, you have finished configuring the development environment and are ready to begin writing recommender code.

**Building the recommendation engine**

Next, you will create the servlet that will handle the recommendation engine code from Apache Mahout.

**A. Create a servlet class**

Begin by creating the servlet class that will represent your Web application.

1. Right click on the **RecommenderWeb** project and select **New > Servlet**.
2. For Java Package, enter `com.ibm.sample.recommender`.
3. For Class Name enter `TestServlet`.

![Create Servlet – Specify class file destination panel](image)

4. Click **Next**.
5. On the next panel, accept the defaults and click **Next**.
6. On the next panel, uncheck **doPost** and click **Finish** (Figure 8).
At the completion of this step, you will have an empty servlet class where you can add your code for the recommendation engine.

B. **Import the source code**

Sample source code for the servlet class is *included with this article*, contained in file snippet1.txt. Download this file and use it to replace the contents of the TestServlet.java source file.

This is a good time to review the important pieces of the code to help you understand what is being done.

1. **Create the data model**

   Previously, you created the database and loaded the data set. In this step, you define a data model object that provides the access to this data set by using JDBC calls. Apache Mahout provides JDBC implementation classes for MySQL and PostgreSQL data sources only. If you want to use a different data source you have three options:
   - Use the GenericJDBCDataModel class and provide all needed SQL queries in the constructor method.
   - Extend the AbstractJDBCDataModel to add an implementation for your data source.
   - Try one of the existing implementations to verify it is compatible.
For the case of Derby, the PostgreSQL implementation is compatible so it is used here (Listing 3). However, for a more robust implementation, you might want to opt for the second option of extending the AbstractJDBCDataModel implementation.

### Listing 3. Datamodel definition

```java
private @Resource (name="jdbc/taste") DataSource tasteDS;

dataModel = new PostgreSQLJDBCDataModel(tasteDS,
"PREFERENCES.TASTE_PREFERENCES",
"USER_ID", "ITEM_ID", "PREFERENCE",
"TIMESTAMP");
```

The org.apache.mahout.cf.taste.model.DataModel interface class represents the data model which is accessed by the Similarity and Recommender classes. There are different implementations for this interface (File, JDBC, and so on). In order to manage the data used after your application is initialized, you can use the methods shown in the table below.

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FastIDSet getItemIDsFromUser(userID)</td>
<td>Returns all item IDs of items user expresses a preference for</td>
</tr>
<tr>
<td>Float getPreferenceValue(userID, itemID)</td>
<td>preference value from the given user for the given item or null if none exists</td>
</tr>
<tr>
<td>int getNumUsers()</td>
<td>Total number of users known to the model.</td>
</tr>
<tr>
<td>void setPreference(userID, itemID, float value)</td>
<td>Sets a particular preference (item plus rating) for a user.</td>
</tr>
<tr>
<td>void removePreference(userID, itemID)</td>
<td>Removes a particular preference for a user.</td>
</tr>
</tbody>
</table>

2. **Generate the user similarity**

Once you have defined the data model, the next thing you need to do is compute the similarity of the data. Apache Mahout provides several algorithms to compute similarity between users. These include: City Block Similarity, Euclidean Distance Similarity, LogLikelihood Similarity, Pearson Correlation Similarity, SpearmanCorrelationSimilarity, TanimotoCoefficientSimilarity, UncenteredCosineSimilarity, and others. (See the the javadoc documentation included in the Mahout distribution for more information about these algorithms.)

The length of the process to compute the similarity between users increases as the number of items and ratings increases. For this reason, you should perform similarity calculations offline for large data sets; for example, by creating jobs for Apache Mahout. Similarity results from these offline tasks can then be included in the data model.

The example used in this article makes use of the PearsonCorrelationSimilarity algorithm, which performs the calculations in real time eliminating the need to perform offline computations.

So, let's perform the similarity calculation for your data model:

```java
UserSimilarity similarity = new PearsonCorrelationSimilarity(dataModel);
```
The similarity variable now contains the similarity information for all the users available in the data model.

3. **Define the user neighborhood**
   The user-based recommendation algorithm makes use of a user neighborhood to specify what users should be considered like a given user. This example uses the NearestNUserNeighborhood implementation, which enables you to specify a limit for how many users should be included in a neighborhood. In this sample, membership in the neighborhood consists of the five most similar users, thus the neighborhood is defined as follows:

   ```java
   UserNeighborhood neighborhood = new NearestNUserNeighborhood(5, similarity, dataModel);
   ```

   Members in a neighborhood are considered to be very similar to the user. Therefore, as the membership increases, the number of recommended items also increases as more members would have recommended more items.

4. **Make the recommendations**
   Now that you have calculated the similarity between users and specified the neighborhood of a user, you can proceed and make user recommendations. Apache Mahout provides the Recommender interface to access the recommendations for users:

   ```java
   Recommender recommender = new GenericUserBasedRecommender(dataModel, neighborhood, similarity);
   ```

   With the recommender defined, you can make recommendations by invoking the `recommender.recommend(long, int)` method and passing the user ID and the max number of recommended items you want to receive. From there, you use an Iterator to process each recommendation.

   Listing 4 shows how to extract recommendations for a user.

   **Listing 4. Iterating thru the recommendations**

   ```java
   List<RecommendedItem> list = recommender.recommend(USER_ID, 10);
   Iterator<RecommendedItem> iter = list.iterator();
   while ( iter.hasNext()) {
      RecommendedItem item = iter.next();
      out.println("<tr><td>" + item.getItemId() + "</td><td>" + item.getValue() + "</td></tr>);
   }
   ```

   So, in summary, the user-based recommendation first computed the similarity between users from the dataset, and then used a neighborhood item to specify what users should be considered similar to the current user. With this information, the recommender algorithm is able to make recommendations based on what items similar users have rated and use this information to estimate how much the current user might like those items.

5. **Test the engine in a Web application**
   Before testing the application in WebSphere Application Server V8, you need to configure the data source on the server and define the JNDI reference for the data source so servlet can access it.
To configure the data source:

1. From the Servers view, right click on WebSphere Application Server at localhost and select **Start**.
2. Once the server has started, right click on the server and select **Administration > Run Administrative Console**.
3. If authentication is enabled, enter your userID and password when the console opens. Click **Login**.
4. Expand **Resources > JDBC > JDBC Providers**.
5. For scope, select **Server** scope.
6. Click **New ...**
7. Enter values as shown in Figure 9.

**Figure 9. New JDBC provider panel**

8. Click **Next**.
9. Verify configuration and click **Finish**.
10. Back in the JDBC Providers list, click on **Derby JDBC Provider**.
11. Click **New ...**
12. For JNDI name enter **jdbc/taste** (Figure 10).

**Figure 10. Data source JDNI name definition panel**

14. Click **Next**.
15. For Database name enter the file path location to the Derby data source you created for loading the data model. Uncheck **Use this data source in contained managed persistence (CMP)** (Figure 11).

**Figure 11. Database name specification panel**

16. Click **Next**.
17. For Step 3, leave the security configuration blank and click **Next**.
18. For Step 4, verify the settings defined and click **Finish**.
19. Save the configuration by clicking on **Save**.

Those are all of the changes you need to make on WebSphere Application Server V8. The data source binding information is already defined in the servlet code using Servlet 2.5 annotations (refer to snippet1.txt), therefore it is not necessary to configure it in the Web module deployment descriptor. The last step is to install the application to the server:

20. From the Servers view, right click on the WebSphere Application Server V8 test server and select **Add or Remove** ...
21. Select **RecommenderApp** and click **Add**.
22. Click **Finish**.

That’s it. The application is now installed and you are finally ready to test.

To test the application:

23. Start the WebSphere Application Server V8 test server from the Servers view, if it is not already started.
24. Expand **Dynamic Web Projects > RecommenderWeb > RecommenderWeb > Servlets** (Figure 12).

**Figure 12. Dynamic Web Projects view**

25. Right click on **TestServlet** and select **Run As > Run on Server**.
26. Select **WebSphere Application Server V8 at localhost** and click **Finish**.
27. The internal browser should not launch and the results of the recommendation can be displayed (Figure 13).

**Figure 13. Recommendation results**

As you can see from the example, the user you want to retrieve recommendations for is user 400. This user has already rated 22 items. Based on the data model and the recommendation algorithm, Apache Mahout was able to recommend a set of items along with a predicted rating that this user might have given them.

**Where to go from here**

Now that you know something about recommendation engines, what next?

For starters, you can begin to see the extra value provided by products that already include a recommendation engine. For example, IBM WebSphere Portal and IBM WebSphere Commerce Suite include a recommendation engine as part of their base offerings. Through software products such as these, IBM has refined and improved support in these areas while managing concerns about online privacy, performance, and integration. (Keep these built-in features of these products in mind should you need to consider any “build or buy” decisions.)

This article did not address one big possible flow, namely the processing of huge volumes of data using map reduce jobs. It wasn’t addressed here, but it has been addressed by IBM. IBM’s Big Data initiatives can help to manage huge amounts of data required by certain applications. As mentioned earlier, the example solution presented here will not scale past a certain point due to the resource limits of greatly increasing the number of users. Big processing problems are
becoming more common and don’t need to stop your progress. Using Big Data initiatives, there’s no reason to be held back by any scale of processing.

You do not need a large data center to use Apache Mahout. Of course, with cloud computing, you don’t need a data center for any solution. Cloud computing is not directly related to the recommendation engine space, but it’s worth mentioning that IBM’s cloud initiatives can give you capacity to try some of these things in ways that might not otherwise be practical.

**Conclusion**

You can see that recommendation engines can add a powerful new dimension to your web applications by driving users to other products or web offerings, based on their individual characteristics or behavior. This article provided a brief introduction of techniques used by recommendation engines, and what you would need to do to make them more scalable. You also saw how Apache Mahout leverages these techniques and helps you integrate them into your Web applications. By integrating these concepts into IBM WebSphere Application Server, you learned that you can extend your existing Web applications and add effective personalization to them.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code sample</td>
<td>1109_recsample.zip</td>
<td>9.4 MB</td>
</tr>
</tbody>
</table>
Related topics

- Programming Collective Intelligence, Toby Segaran, O'Reilly Media, Inc., 2007.
- What is Apache Mahout?
- Introducing Apache Mahout
- Apache Hadoop
- What is big data?
- GroupLens Research - MovieLens Data Sets
- IBM Portal LikeMinds Recommendations
- IBM WebSphere Portal 6.1.5 Information Center
- IBM Cloud Computing
- “New York Times” Quietly Rolls Out Recommendations Service