Apache Mahout - Taste Documentation

Sean Owen

Table of contents

1. Overview

Taste is a flexible, fast collaborative filtering engine for Java. The engine takes users' preferences for items ("tastes") and returns estimated preferences for other items. For example, a site that sells books or CDs could easily use Taste to figure out, from past purchase data, which CDs a customer might be interested in listening to.

Taste provides a rich set of components from which you can construct a customized recommender system from a selection of algorithms. Taste is designed to be enterprise-ready; it's designed for performance, scalability and flexibility. Taste is not just for Java; it can be run as an external server which exposes recommendation logic to your application via web services and HTTP.

Top-level packages define the Taste interfaces to these key abstractions:

- `DataModel`
- `UserSimilarity` and `ItemSimilarity`
- `UserNeighborhood`
- `Recommender`

Subpackages of `org.apache.mahout.cf.taste.impl` hold implementations of these interfaces. These are the pieces from which you will build your own recommendation engine. That's it! For the academically inclined, Taste supports both *memory-based* and *item-based* recommender systems, *slope one* recommenders, and a couple other experimental implementations. It does not currently support *model-based* recommenders.

2. Architecture

This diagram shows the relationship between various Taste components in a user-based recommender. An item-based recommender system is similar except that there are no PreferenceInferrers or Neighborhood algorithms involved.

2.1. Recommender

A `Recommender` is the core abstraction in Taste. Given a `DataModel`, it can produce recommendations. Applications will most likely use the `GenericUserBasedRecommender` implementation or `GenericItemBasedRecommender`, possibly decorated by `CachingRecommender`.

2.2. DataModel

A `DataModel` is the interface to information about user preferences. An implementation might draw this data from any source, but a database is the most likely source. Taste provides `MySQLJDBCDataModel` to access preference data from a database via JDBC, though many applications will want to write their own. Taste also provides a `FileDataModel`.

Along with `DataModel`, Taste uses the `User`, `Item` and `Preference` abstractions to represent the users, items, and preferences for those items in the recommendation engine. Custom `DataModel` implementations would return implementations of these interfaces that are appropriate to the application - maybe an `OnlineUser` implementation that represents an online store user, and a `BookItem` implementation representing a book.

2.3. UserSimilarity, ItemSimilarity

A `UserSimilarity` defines a notion of similarity between two `User`s. This is a crucial part of a recommendation engine. These are attached to a `Neighborhood` implementation. `ItemSimilarity`s are analagous, but find similarity between `Item`s.

2.4. UserNeighborhood

In a user-based recommender, recommendations are produced by finding a "neighborhood" of similar users near a given user. A `UserNeighborhood` defines a means of determining that neighborhood — for example, nearest 10 users. Implementations typically need a `UserSimilarity` to operate.

3. Requirements

3.1. Required

- Java / J2SE 5.0

3.2. Optional

- Apache Ant 1.5 or later, if you want to build from source or build examples.
- Taste web applications require a Servlet 2.3+ container, such as Jakarta Tomcat. It may in fact work with older containers with slight modification.
- `MySQLJDBCDataModel` implementation requires a MySQL 4.x (or later) database. Again, it may be made to work with earlier versions or other databases with slight changes.

4. Demo

To build and run the demo, follow the instructions below, which are written for Unix-like operating systems:

1. Download the "1 Million MovieLens Dataset" from http://www.grouplens.org/.
2. Unpack the archive and copy `movies.dat` and `ratings.dat` to `src/main/examples/org/apache/mahout/cf/taste/example/grouplens` under the Taste distribution directory.
3. Navigate to the directory where you unpacked the Mahout distribution, and navigate to `trunk/core`.
4. Build Mahout with `ant`.
5. Build the example web application with `ant -f taste-build.xml build-grouplens-example`.
6. Download and install Tomcat.
7. Copy `taste.war` to the `webapps` directory under the Tomcat installation directory.
8. Increase the heap space that is given to Tomcat by setting the `JAVA_OPTS` environment variable to "`-server -da -dsa -Xms1024m -Xmx1024m`", to allow 1024MB of heap space and enable performance optimizations. Using `bash`, one can do this with the command `export JAVA_OPTS="..."`
9. Start Tomcat. This is usually done by running `bin/startup.sh` from the Tomcat installation directory. You may get an error asking you to set `JAVA_HOME`; do so as above.
10. Get recommendations by accessing the web application in your browser: `http://localhost:8080/taste/RecommenderServlet?userID=1` This will produce a simple preference-item ID list which could be consumed by a client application. Get more useful human-readable output with the `debug` parameter: `http://localhost:8080/taste/RecommenderServlet?userID=1&debug=true`

Incidentally, Taste's web service interface may then be found at:
`http://localhost:8080/taste/RecommenderService.jws`
Its WSDL file will be here...
`http://localhost:8080/taste/RecommenderService.jws?wsdl`
... and you can even access it in your browser via a simple HTTP request:
`.../RecommenderService.jws?method=recommend&userID=1&howMany=10`

5. Examples

5.1. User-based Recommender

User-based recommenders are the "original", conventional style of recommender system. They can produce good recommendations when tweaked properly; they are not necessarily the fastest recommender systems and are thus suitable for small data sets (roughly, less than a million ratings). We'll start with an example of this.

First, create a `DataModel` of some kind. Here, we'll use a simple on based on data in a file:

```
DataModel model = new FileDataModel(new File("data.txt"));
```

We'll use the PearsonCorrelationSimilarity implementation of `UserSimilarity` as our user correlation algorithm, and add an optional preference inference algorithm:

```
UserSimilarity userSimilarity = new PearsonCorrelationSimilarity(model); // Optional:
userSimilarity.setPreferenceInferrer(new AveragingPreferenceInferrer());
```

Now we create a `UserNeighborhood` algorithm. Here we use nearest-3:

```
UserNeighborhood neighborhood = new NearestNUserNeighborhood(3,
userSimilarity, model);
```

Now we can create our `Recommender`, and add a caching decorator:

```
Recommender recommender = new GenericUserBasedRecommender(model,
neighborhood, userSimilarity); Recommender cachingRecommender = new
CachingRecommender(recommender);
```

Now we can get 10 recommendations for user ID "1234" — done!

```
List<RecommendedItem> recommendations =
cachingRecommender.recommend("1234", 10);
```

5.2. Item-based Recommender

We could have created an item-based recommender instead. Item-based recommender base recommendation not on user similarity, but on item similarity. In theory these are about the same approach to the problem, just from different angles. However the similarity of two items is relatively fixed, more so than the similarity of two users. So, item-based recommenders can use pre-computed similarity values in the computations, which make them much faster. For large data sets, item-based recommenders are more appropriate.

Let's start over, again with a `FileDataModel` to start:

```
DataModel model = new FileDataModel(new File("data.txt"));
```

We'll also need an `ItemSimilarity`. We could use `PearsonCorrelationSimilarity`, which computes item similarity in realtime, but, this is generally too slow to be useful. Instead, in a real application, you would feed a list of pre-computed correlations to a `GenericItemSimilarity`:

```
// Construct the list of pre-compted correlations
Collection<GenericItemSimilarity.ItemItemSimilarity> correlations = ...; ItemSimilarity
itemSimilarity = new GenericItemSimilarity(correlations);
```

Then we can finish as before to produce recommendations:

```
Recommender recommender = new GenericItemBasedRecommender(model,
itemSimilarity); Recommender cachingRecommender = new
CachingRecommender(recommender); ... List<RecommendedItem>
recommendations = cachingRecommender.recommend("1234", 10);
```

5.3. Slope-One Recommender

This is a simple yet effective `Recommender` and we present another example to round out the list:

```
DataModel model = new FileDataModel(new File("data.txt")); // Make a weighted
slope one recommender Recommender recommender = new
SlopeOneRecommender(model); Recommender cachingRecommender = new
CachingRecommender(recommender);
```

6. Integration with your application

6.1. Direct

You can create a `Recommender`, as shown above, wherever you like in your Java application, and use it. This includes simple Java applications or GUI applications, server applications, and J2EE web applications.

6.2. Standalone server

Taste can also be run as an external server, which may be the only option for non-Java applications. A Taste Recommender can be exposed as a web application via `org.apach.mahout.cf.taste.web.RecommenderServlet`, and your application can then access recommendations via simple HTTP requests and response, or as a full-fledged SOAP web service. See above, and see `the javadoc` for details.

To deploy your `Recommender` as an external server:

1. Create an implementation of `org.apache.mahout.cf.taste.recommender.Recommender`.
2. Compile it and create a JAR file containing your implementation.
3. Build a WAR file that will run your Recommender as a web application: `ant -Dmy-recommender.jar=yourJARfile.jar -Dmy-recommender-class=com.foo.YourRecommender build-server`
4. Follow from the "Install Tomcat" step above under Demo.

7. Performance

7.1. Runtime Performance

The more data you give Taste, the better. Though Taste is designed for performance, you will undoubtedly run into performance issues at some point. For best results, consider using the following commad-line flags to your JVM:

- `-server`: Enables the server VM, which is generally appropriate for long-running, computation-intensive applications.
- `-Xms1024m -Xmx1024m`: Make the heap as big as possible -- a gigabyte doesn't hurt when dealing with millions of preferences. Taste will generally use as much memory as you give it for caching, which helps performance. Set the initial and max size to the same value to avoid wasting time growing the heap, and to avoid having the JVM run minor collections to avoid growing the heap, which will clear cached values.
- `-da -dsa`: Disable all assertions.
- `-XX:+UseParallelGC` (multi-processor machines only): Use a GC algorithm designed to take advantage of multiple processors, and designed for throughput. This is a default in J2SE 5.0.
- `-XX:-DisableExplicitGC`: Disable calls to `System.gc()`. These calls can only hurt in the presence of modern GC algorithms; they may force Taste to remove cached data needlessly. This flag isn't needed if you're sure your code and third-party code you use doesn't call this method.

Also consider the following tips:

- Use `CachingRecommender` on top of your custom `Recommender` implementation.
- When using `JDBCDataModel`, make sure you've taken basic steps to optimize the table storing preference data. Create a primary key on the user ID and item ID columns, and an index on them. Set them to be non-null. And so on. Tune your database for lots of concurrent reads! When using JDBC, the database is almost always the bottleneck. Plenty of memory and caching are even more important.
- Also, pooling database connections is essential to performance. If using a J2EE container, it probably provides a way to configure connection pools. If you are creating your own `DataSource` directly, try wrapping it in `org.apache.mahout.cf.taste.impl.model.jdbc.ConnectionPoolDataSource`
- See MySQL-specific notes on performance in the javadoc for `MySQLJDBCDataModel`.

7.2. Algorithm Performance: Which One Is Best?

There is no right answer; it depends on your data, your application, environment, and performance needs. Taste provides the building blocks from which you can construct the best `Recommender` for your application. The links below provide research on this topic. You will probably need a bit of trial-and-error to find a setup that works best. The code sample above provides a good starting point.

Fortunately, Taste provides a way to evaluate the accuracy of your `Recommender` on your own data, in `org.apache.mahout.cf.taste.eval`:

DataModel myModel = ...; RecommenderBuilder builder = new RecommenderBuilder() { public Recommender buildRecommender(DataModel model) { // build and return the Recommender to evaluate here } }; RecommenderEvaluator evaluator = new AverageAbsoluteDifferenceRecommenderEvaluator(); double evaluation = evaluator.evaluate(builder, myModel, 0.9, 1.0);

8. Useful Links

You'll want to look at these packages too, which offer more algorithms and approaches that you may find useful:

- Cofi: A Java-Based Collaborative Filtering Library
- CoFE

Here's a handful of research papers that I've read and found particularly useful:

J.S. Breese, D. Heckerman and C. Kadie, "Empirical Analysis of Predictive Algorithms for Collaborative Filtering," in Proceedings of the Fourteenth Conference on Uncertainity in Artificial Intelligence (UAI 1998), 1998.

B. Sarwar, G. Karypis, J. Konstan and J. Riedl, "Item-based collaborative filtering recommendation algorithms," in Proceedings of the Tenth International Conference on the World Wide Web (WWW 10), pp. 285-295, 2001.

P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom and J. Riedl, "GroupLens: an open architecture for collaborative filtering of netnews," in Proceedings of the 1994 ACM conference on Computer Supported Cooperative Work (CSCW 1994), pp. 175-186, 1994.

J.L. Herlocker, J.A. Konstan, A. Borchers and J. Riedl, "An algorithmic framework for performing collaborative filtering," in Proceedings of the 22nd annual international ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 99), pp. 230-237, 1999.

Clifford Lyon, "Movie Recommender," CSCI E-280 final project, Harvard University, 2004.

Daniel Lemire, Anna Maclachlan, "Slope One Predictors for Online Rating-Based

Collaborative Filtering," Proceedings of SIAM Data Mining (SDM '05), 2005.

Michelle Anderson, Marcel Ball, Harold Boley, Stephen Greene, Nancy Howse, Daniel Lemire and Sean McGrath, "RACOFI: A Rule-Applying Collaborative Filtering System," Proceedings of COLA '03, 2003.

These links will take you to all the collaborative filtering reading you could ever want!

- Paul Perry's notes
- James Thornton's collaborative filtering resources
- Daniel Lemire's blog which frequently covers collaborative filtering topics