Overview

This documentation concerns the non-distributed, non-Hadoop-based recommender engine / collaborative filtering code inside Mahout. It was formerly a separate project called "Taste" and has continued development inside Mahout alongside other Hadoop-based code. It may be viewed as a somewhat separate, more comprehensive and more mature aspect of this code, compared to current development efforts focusing on Hadoop-based distributed recommenders. This remains the best entry point into Mahout recommender engines of all kinds.

A Mahout-based collaborative filtering 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 Mahout to figure out, from past purchase data, which CDs a customer might be interested in listening to.

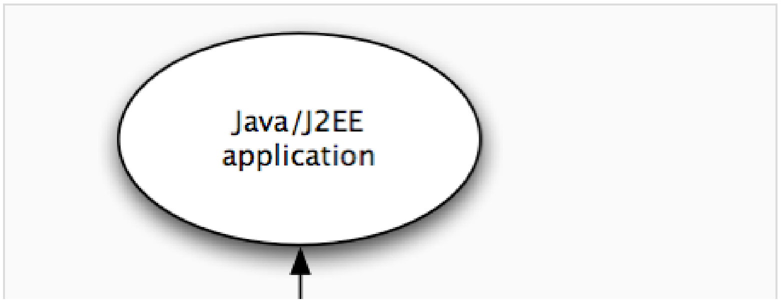
Mahout provides a rich set of components from which you can construct a customized recommender system from a selection of algorithms. Mahout is designed to be enterprise-ready; it's designed for performance, scalability and flexibility.

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

- **DataModel**
- **UserSimilarity**
- **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!

Architecture



Twitter

Tweets Follow

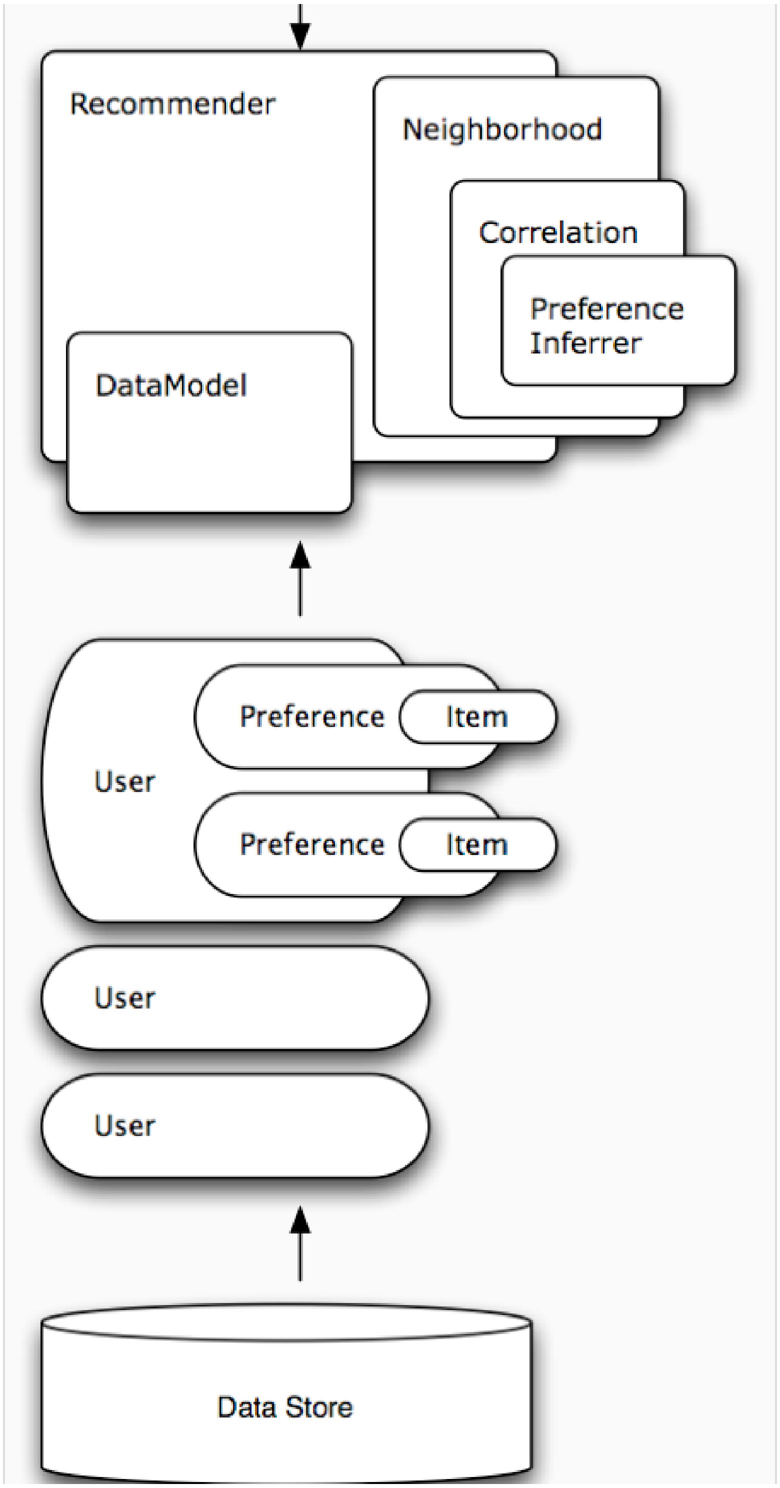
GOTO Berlin @GOTOber 10 Dec
Presentation "@ApacheMahout's new DSL for Distributed Machine Learning on #SPARK" by Sebastian Schelter [PDF] bit.ly/11sLRju
Retweeted by Apache Mahout
Expand

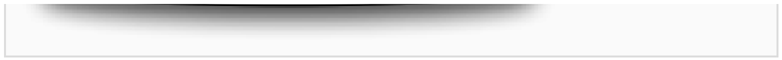
Indizen Insight @indizen_insight 2 Dec
@ApacheMahout to Get Self-Optimizing Matrix Algebra Interface for @ApacheSpark and #Flink infoq.com/news/2014/11/m... via @InfoQ @davidtalby
Retweeted by Apache Mahout
Expand

Suneel Marthi @suneelmarthi 25 Nov
So #Mahout recommendation engine on @elasticsearch github.com/codelibs/elast... cc:/ @MaineC
Retweeted by Apache Mahout
Show Summary

Suneel Marthi @suneelmarthi 25 Nov

Tweet to @ApacheMahout


Recommender
Neighborhood
Correlation
Preference Inferrer
DataModel
User
Preference
Item
Preference
Item
User
User
Data Store



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

Recommender

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

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. Be sure to wrap this with a **ReloadFromJDBCDataModel** to get good performance! Mahout provides **MySQLJDBCDataModel**, for example, to access preference data from a database via JDBC and MySQL. Another exists for PostgreSQL. Mahout also provides a **FileDataModel**, which is fine for small applications.

Users and items are identified solely by an ID value in the framework. Further, this ID value must be numeric; it is a Java long type through the APIs. A **Preference** object or **PreferenceArray** object encapsulates the relation between user and preferred items (or items and users preferring them).

Finally, Mahout supports, in various ways, a so-called "boolean" data model in which users do not express preferences of varying strengths for items, but simply express an association or none at all. For example, while users might express a preference from 1 to 5 in the context of a movie recommender site, there may be no notion of a preference value between users and pages in the context of recommending pages on a web site: there is only a notion of an association, or none, between a user and pages that have been visited.

UserSimilarity

A **UserSimilarity** defines a notion of similarity between two users. This is a crucial part of a recommendation engine. These are attached to a **Neighborhood** implementation. **ItemSimilarity** is analagous, but find similarity between items.

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.

Examples

User-based Recommender

User-based recommenders are the "original", conventional style of recommender systems. 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 ten 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. The file should be in

CSV format, with lines of the form "userID,itemID,prefValue" (e.g. "39505,290002,3.5"):

```
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);
```

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

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

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);
```

Item-based Recommender

We could have created an item-based recommender instead. Item-based recommenders 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-computed 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);
```

Integration with your application

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.

Performance

Runtime Performance

The more data you give, the better. Though Mahout is designed for performance, you will undoubtedly run into performance issues at some point. For best results, consider using the following command-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 tens millions of preferences. Mahout recommenders 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:NewRatio=9: Increase heap allocated to 'old' objects, which is most of them in this framework
- -XX:+UseParallelGC -XX:+UseParallelOldGC (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 Mahout 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 wrap it with the **ReloadFromJDBCDataModel** to load data into memory!.

Algorithm Performance: Which One Is Best?

There is no right answer; it depends on your data, your application, environment, and performance needs. Mahout 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, Mahout 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);
```

For "boolean" data model situations, where there are no notions of preference value, the above evaluation based on estimated preference does not make sense. In this case, try a *RecommenderIRStatsEvaluator*, which presents traditional information retrieval figures like precision and recall, which are more meaningful.

Useful Links

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.