5. Predicting User Ratings (Building a Recommender)

The third challenge problem requires you to predict ratings for a select set of users and content items. In the data, you've seen that there are ratings and review events that indicate specific user's preferences for specific content items. In addition, the data also contains viewing and queueing events that provide implicit data about user's preferences. Using this information, you can construct a model that can be passed to an off-the-shelf recommendation engine to predict the requested ratings. You could build a recommendation engine yourself, as the mathematics isn't terribly difficult, but you'll save time and have more freedom to experiment with algorithms and similarity metrics using an off-the-shelf package. This exercise uses the Taste package included in Mahout for this challenge problem.

There is a wealth of information about building recommendation systems online and in print. The following resources are a good place to start:

- Mahout in Action
- Mining Massive Data Sets, chapter 9
- Big, Practical Recommendations with Alternating Least Squares

Before you get too far down the path of algorithms, you should take a closer look at the data. The 'canonical' input data for recommenders is ratings data. In total in the data set there are 926 ratings (or reviews with ratings) from 751 users for 757 items. Additionally there are about a half million events that might be considered a "play", such as playing a content item or adding it to the queue for later playback. If you add those events to the mix, there are 2193 users and 6504 items. Since the explicit rating data is so sparse, and since there is roughly 500 times more implicit data than explicit data, whatever solution you adopt should account for implicit events.

The data set contains a variety of other events that this exercise ignores. Some of them, like search behavior, reviewing recommendations, or account actions, carry no real signal for content preferences. Others, like the results of the problem one classifier or the viewing behavior, have some potential for use but are challenging to structure in a way that can be input into a standard recommender. For this problem, you will work with only the explicit and implicit ratings information.

Step 1. Transform your cleaned data

Before you can begin on this problem, you must determine whether you need to transform your cleaned data. The Mahout recommendation engine expects all input data to be in CSV format. You need two CSV files: one for the explicit ratings, and one for the implicit ratings. For the implicit ratings, you should combine all play-like events for a single user and item into a single event because that's what Mahout will do anyway.

You also need to determine how you want to segregate your data into a training data set and a test data set so that you can measure and select the algorithms that give the best RMSE. There are two typical approaches to selecting the training and test sets with time-stamped data, like what you have in this data set. One is to designate all data up to a given time stamp as training and the remainder as test. The other is to simply sample test data records randomly. The advantage of the time-ordered approach is that it is more realistic. It can, however, introduce bias if the time span is long enough for user preferences to shift or the sampling window is too short, in which case random sampling works better. Because the time window is short in the data set, our approach uses the time-ordered approach. The data set ranges from midnight on May 5th through midnight on May 12th. A solid approach is to treat all data prior to midnight on the 10th as training data and the remainder as test data.

To create the CSV files, you have several options: you could use the JSON data loader in Pig to load the data and easily extract the correct fields; you could define a Hive table using the JSON SerDe and then select the desired fields from the table; or you could write a map-only Hadoop streaming job. Because we can parse the JSON in Python without giving a schema and the operation is trivial, the streaming job is probably the easiest option.

1. Write a mapper for the explicit data

For the explicit data mapper, your script should iterate through all rating and review data and emit the user ID, item ID, and rating triplets and emit them if they meet given date criteria. Create a new file called `explicit.py` with the following contents:

```
#!/usr/bin/python

import datetime
import dateutil.parser
import json
```

```
import sys

def main():
   before = sys.argv[1] == 'before'
   cutoff = dateutil.parser.parse(sys.argv[2])
   epoch = datetime.datetime(1970,1,1,tzinfo=cutoff.tzinfo)

   # Read all lines from stdin
   for line in sys.stdin:
      data = json.loads(line)
      start = cutoff - datetime.timedelta(seconds=long(data['start']))

      if (before and start > epoch) or (not before and start <= epoch):
         for item, rating in data['rated'].iteritems():
            print "%s,%s,%s" % (data['user'], item, rating)

         for item, dict in data['reviewed'].iteritems():
            print "%s,%s,%s" % (data['user'], item, dict['rating'])

if __name__ == '__main__':
   main()
```

The odd time math with the cutoff and epoch is to avoid the parts of python that are highly version dependent and/or overly complicated.

2. Write a mapper for the implicit data

For the implicit data mapper, you just want your script to iterate over the rating and review data as well as the played movies, queued movies, and browsed movies. You could include the recently played items, but the risk is that the same items can show up in multiple sessions, making them seem more important. The number of recent items is very small, so it's probably better to leave them out. You don't need to worry about training versus test data because the implicit data has no ratings. Create a new file called `implicit.py` with the following contents:

```
#!/usr/bin/python

import json
import sys

def main():
   # Read all lines from stdin
   for line in sys.stdin:
      data = json.loads(line)
      for item in data['rated'].keys() + data['reviewed'].keys() + data['played'].keys() +
data['browsed'] + data['queued']:
         print "%s,%s" % (data['user'], item)

if __name__ == '__main__':
   main()
```

3. Run the jobs

You can now run these jobs to get the explicit and implicit data sets. Here again, you have to be careful with the treatment of the separators. Because these columns are going to be interpreted as numeric values, you don't want to have an extraneous tab character at the end of each line. The way to avoid it is to tell Hadoop that the key and value are comma delimited and also to use commas to delimit the final output.

Enter the following commands:

```
$ hadoop jar $STREAMING -D mapred.reduce.tasks=0 -D mapred.textoutputformat.separator=, -D
stream.map.output.field.separator=, -mapper 'explicit.py before "2013-05-11 00:00:00-08:00"' -file
explicit.py -input clean -output explicit_train
...
$ hadoop jar $STREAMING -D mapred.reduce.tasks=0 -D mapred.textoutputformat.separator=, -D
stream.map.output.field.separator=, -mapper 'explicit.py after "2013-05-11 00:00:00-08:00"' -file
explicit.py -input clean -output explicit_test
...
```

```
$ hadoop jar $STREAMING -D mapred.reduce.tasks=0 -D mapred.textoutputformat.separator=, -D
stream.map.output.field.separator=, -mapper implicit.py -file implicit.py -input clean -output
implicit
…
$ hadoop fs -cat explicit_train/part\* | head
10142325,9614,5
10760746,27597,4
10796192,30975e17,3
10905688,36598,3
10905688,15337,5
1091145,14303e57,1
11657116,9146e38,4
11749679,27017e20,4
11751432,38193e81,4
11764279,21482e41,3
cat: Unable to write to output stream.
$ hadoop fs -cat implicit/part\* | head
10108881,16316
10108881,10286
10108881,13338
10108881,9107
10108881,39122
10142325,9614
10142325,9614
10142325,38579
10151338,34645
10151338,34645
cat: Unable to write to output stream.
```

The outputs tells us that the data files are good. You should note also that the item IDs aren't all completely numeric (which you already discovered during the data exploration). Mahout requires that all IDs, user and item, be numeric, so you have to fix that.

4. Translate the IDs

First, you need to determine how ubiquitous the problem is:

```
$ hadoop fs -cat implicit/part\* | cut -d, -f1 | tr -d '1234567890' | sort | uniq

$ hadoop fs -cat implicit/part\* | cut -d, -f2 | tr -d '1234567890' | sort | uniq

e
```

In these commands, `cut` is used to extract only the first column from the comma-delimited data files, `tr` is used to remove all numbers from that data, and `sort` and `uniq` are used to output the unique items.

The result shows that 'e' is the only non-numeric character in the IDs, and it only shows up in the item IDs. We know from the information we were given initially that the items with an 'e' in the ID are TV shows, and the number after the 'e' is the episode number.

Before you translate these IDs, determine their range:

```
$ hadoop fs -cat implicit/part\* | cut -d, -f2 | cut -de -f1 | sort -n | head -1
1094
$ hadoop fs -cat implicit/part\* | cut -d, -f2 | cut -de -f1 | sort -n | tail -1
39984
```

In these commands, the first `cut` command extracts the first column from the comma-delimited data, and the second `cut` command extracts the number that comes before the 'e' in that data. The `sort -n` command sorts the data in numerical order. The `head` and `tail` commands output only the first and last lines, respectively.

Take a look at the episode numbers with the following commands:

```
$ hadoop fs -cat implicit/part\* | cut -d, -f2 | grep e | cut -de -f2 | sort -n | uniq | head
```

```
1
2
3
4
5
6
7
8
9
10
$ hadoop fs -cat implicit/part\* | cut -d, -f2 | grep e | cut -de -f2 | sort -n | tail -1
217
```

Given that the IDs range roughly from 1,000 to 40,000 and the episode numbers are not zero-padded, you can simply replace the ‘e’ with ‘00’ without affecting uniqueness, and translate back to the original IDs later. The same caveats as above apply about separators. Translate using `sed` and Hadoop streaming with the following commands:

```
$ hadoop jar $STREAMING -D mapred.reduce.tasks=0 -D mapred.textoutputformat.separator=, -D
stream.map.output.field.separator=, -mapper 'sed "s/e/00/"' -input explicit_train -output
explicit_train_clean
...
$ hadoop jar $STREAMING -D mapred.reduce.tasks=0 -D mapred.textoutputformat.separator=, -D
stream.map.output.field.separator=, -mapper 'sed "s/e/00/"' -input explicit_test -output
explicit_test_clean
...
$ hadoop jar $STREAMING -D mapred.reduce.tasks=0 -D mapred.textoutputformat.separator=, -D
stream.map.output.field.separator=, -mapper 'sed "s/e/00/"' -input implicit -output implicit_clean
...
```

You now have your training and test data sets.

Step 2. Establish baseline averages

In this step, you want establish baselines against which you can evaluate your recommenders. The most simplistic metric you can try is guessing the global average for all of our test data. Since you were careful about the separators before, you can do that pretty simply with Hive by creating external tables for your explicit data and query the results:

1. Create Hive tables

Enter the following commands to create external tables for our explicit data in Hive and query the results:

```
$ hive
hive> create external table explicit_train (user int, item bigint, rating int) row format delimited
fields terminated by ',' location '/user/cloudera/explicit_train_clean';
OK
Time taken: 0.215 seconds
hive> create external table explicit_test (user int, item bigint, rating int) row format delimited
fields terminated by ',' location '/user/cloudera/explicit_test_clean';
OK
Time taken: 0.191 seconds
```

During table creation, you must specify where the data files are and indicate that the data files are comma-delimited.

2. Calculate the global average rating and the RMSE

Once you’ve created the tables, you can use the math operations provided by Hive to calculate the average rating and the RMSE. Enter the following commands:

```
hive> select avg(rating) from explicit_train;
...
OK
3.597826086956522
Time taken: 6.132 seconds
hive> select sqrt(sum(pow(rating - 3.597826086956522, 2))/count(*)) from explicit_test;
...
```

```
OK
1.2733209628271343
Time taken: 4.222 seconds
```

The average rating is about 3.6, and the RMSE of just guessing that for every instance in our test data set is roughly 1.27. Not terrible, but we can go about this in a smarter fashion..

3. Compare global average to user ratings

While you're setting baselines, test to see what you get if you use each user's average rating instead of the global average rating. In cases where there are no previous ratings for a user, use the global average rating. Enter the following commands in Hive:

```
hive> create table baseline (user int, item bigint, rating float) row format delimited fields
terminated by ',';
OK
Time taken: 1.099 seconds
hive> insert into table baseline select explicit_test.user, explicit_test.item, if (avg.avg_rating >
0, avg.avg_rating, 3.597826086956522) from (select user, avg(rating) as avg_rating from explicit_train
group by user) avg full outer join explicit_test on explicit_test.user == avg.user;
...
OK
Time taken: 7.462 seconds
hive> select sqrt(sum(pow(e.rating - b.rating, 2))/count(*)) from baseline b join explicit_test e on
b.user == e.user and b.item == e.item;
OK
1.362417948479424
Time taken: 4.768 seconds
```

Interestingly, the user averages are actually worse than the global average, which is most likely because the data is so sparse that you can't make any reasonable guesses for individual users. You can confirm by looking at the data files in the `/user/hive/warehouse/baseline` directory in HDFS. What you would see is that most predictions are the global average rating, and of those that aren't, the majority have a round rating, meaning that the rating is likely an average over a single value. Since you can't rely on the user averages, you should work with the global average rating as your baseline.

Step 3. Working with Mahout's recommenders

There are two main classes of recommenders in Mahout: similarity-based and matrix factorization. The most effective way to determine which algorithms perform best is to try them, experimenting with a variety of algorithms and a variety of configuration options.

With similarity-based algorithms, the algorithm determines similarity between users and items and then estimates unknown ratings based on those similarities and the known ratings. The usual approach is to use the known ratings that are in common to evaluate the similarity of users or items, but using the known ratings is not required. You can use the much larger implicit rating data set to determine similarity. When working with implicit data, log-likelihood is the standard choice of similarity metric, though Tanimoto (Jaccard) is also used. You'll use both below.

Mahout offers a way to run a recommender from the command line, but in most cases, it's more effective (and sometimes necessary) to use Mahout's API instead, which means writing code in Java. Mahout's recommenders are mostly non-distributed, though there is one similarity-based variant that is built on MapReduce. Below, you'll first explore the predictability of the ratings using the non-distributed algorithms, and when you know how well each works, you can decide whether it's a good idea to switch to the distributed option.

1. Download your data

Because you'll be working with your data locally, you'll need to download it from HDFS. Enter the following commands:

```
$ hadoop fs -getmerge implicit_clean implicit.csv
$ hadoop fs -cat explicit_train_clean/part\* explicit_test_clean/part\* > explicit.csv
```

The commands merge the training and test data back together. The tool you'll be using in Mahout to evaluate the various algorithms has its own built-in training data sampling process.

2. Set up a Maven project

To get started, you must first setup a Maven project. You'll find a succinct tutorial on this process here: https://code.google.com/p/unresyst/wiki/CreateMahoutRecommender. In summary, the steps to create a Mahout Maven project are:

1. `$ mvn archetype:create -DarchetypeGroupId=org.apache.maven.archetypes -DgroupId=ccp.challenge1.recommend -DartifactId=recommend`
   This command will create an empty project in the `recommend` directory.
2. Open up the `recommend/pom.xml` file in an editor and add the following lines to the `<dependencies>...</dependencies>` section:

```
<dependency>
  <groupId>org.apache.mahout</groupId>
  <artifactId>mahout-core</artifactId>
  <version>0.7</version>
</dependency>
<dependency>
  <groupId>org.apache.mahout</groupId>
  <artifactId>mahout-math</artifactId>
  <version>0.7</version>
</dependency>
<dependency>
  <groupId>org.apache.mahout</groupId>
  <artifactId>mahout-math</artifactId>
  <version>0.7</version>
  <type>test-jar</type>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.apache.mahout</groupId>
  <artifactId>mahout-utils</artifactId>
  <version>0.5</version>
</dependency>
```

Rather than creating a new Maven project, you can copy the provided project from the `/home/cloudera/scripts/recommendations` directory to your current working directory with the following command:

```
$ cp -R /home/cloudera/scripts/recommendations/recommend .
```

Once you've created the project, you'll write an application to create one instance of every relevant recommender and evaluate them using the `RMSRecommenderEvaluator`. You'll run each evaluation 10 times and average the results to account for the randomness in the training data sampling process.

Create a new Java file in the Maven project, `ccp/challenge1/recommender/Recommend.java`, with the following contents:

```
package ccp.challenge1.recommender;

import java.io.File;
import java.util.HashMap;
import java.util.Map;
import java.util.TreeMap;
import org.apache.mahout.cf.taste.common.TasteException;
import org.apache.mahout.cf.taste.eval.RecommenderBuilder;
import org.apache.mahout.cf.taste.eval.RecommenderEvaluator;
import org.apache.mahout.cf.taste.impl.eval.AverageAbsoluteDifferenceRecommenderEvaluator;
import org.apache.mahout.cf.taste.impl.eval.RMSRecommenderEvaluator;
import org.apache.mahout.cf.taste.impl.model.file.FileDataModel;
import org.apache.mahout.cf.taste.impl.neighborhood.NearestNUserNeighborhood;
import org.apache.mahout.cf.taste.impl.neighborhood.ThresholdUserNeighborhood;
import org.apache.mahout.cf.taste.impl.recommender.GenericItemBasedRecommender;
import org.apache.mahout.cf.taste.impl.recommender.GenericUserBasedRecommender;
import org.apache.mahout.cf.taste.impl.recommender.slopeone.SlopeOneRecommender;
import org.apache.mahout.cf.taste.impl.recommender.svd.ALSWRFactorizer;
import org.apache.mahout.cf.taste.impl.recommender.svd.ExpectationMaximizationSVDFactorizer;
import org.apache.mahout.cf.taste.impl.recommender.svd.ImplicitLinearRegressionFactorizer;
import org.apache.mahout.cf.taste.impl.recommender.svd.SVDRecommender;
```

```
import org.apache.mahout.cf.taste.impl.similarity.LogLikelihoodSimilarity;
import org.apache.mahout.cf.taste.impl.similarity.TanimotoCoefficientSimilarity;
import org.apache.mahout.cf.taste.model.DataModel;
import org.apache.mahout.cf.taste.neighborhood.UserNeighborhood;
import org.apache.mahout.cf.taste.recommender.Recommender;

public class Recommend {
   private static final int ITERATIONS = 10;

   public static void main(String[] args) throws Exception {
      final DataModel ratingDataModel = new FileDataModel(new File("../explicit.csv"));
      final DataModel allDataModel = new FileDataModel(new File("../implicit.csv"));
      final LogLikelihoodSimilarity ll = new LogLikelihoodSimilarity(allDataModel);
      final TanimotoCoefficientSimilarity tanimoto = new TanimotoCoefficientSimilarity(allDataModel);
      RecommenderEvaluator rmse = new RMSRecommenderEvaluator();
      RecommenderEvaluator abs = new AverageAbsoluteDifferenceRecommenderEvaluator();
      Map<String, RecommenderBuilder> recs = new HashMap<String, RecommenderBuilder>();
      recs.put("User-user n-nearest neighbor with Tanimoto",
              new RecommenderBuilder() {
                 public Recommender buildRecommender(DataModel dm) throws TasteException {
                    UserNeighborhood nearest = new NearestNUserNeighborhood(10, tanimoto,
allDataModel);
                    return new GenericUserBasedRecommender(dm, nearest, tanimoto);
                 }
              });

      recs.put("User-user n-nearest neighbor with log-likelihood",
              new RecommenderBuilder() {
         public Recommender buildRecommender(DataModel dm) throws TasteException {
            UserNeighborhood nearest = new NearestNUserNeighborhood(10, ll, allDataModel);
            return new GenericUserBasedRecommender(dm, nearest, ll);
         }
      });

      recs.put("User-user threshold with Tanimoto",
              new RecommenderBuilder() {
         public Recommender buildRecommender(DataModel dm) throws TasteException {
            UserNeighborhood nearest = new ThresholdUserNeighborhood(0.7, tanimoto, allDataModel);
            return new GenericUserBasedRecommender(dm, nearest, tanimoto);
         }
      });

      recs.put("User-user threshold with log-likelihood",
              new RecommenderBuilder() {
         public Recommender buildRecommender(DataModel dm) throws TasteException {
            UserNeighborhood nearest = new ThresholdUserNeighborhood(0.7, ll, allDataModel);
            return new GenericUserBasedRecommender(dm, nearest, ll);
         }
      });

      recs.put("Item-item with log-likelihood",
              new RecommenderBuilder() {
         public Recommender buildRecommender(DataModel dm) throws TasteException {
            return new GenericItemBasedRecommender(dm, ll);
         }
      });

      recs.put("Item-item with Tanimoto",
              new RecommenderBuilder() {
         public Recommender buildRecommender(DataModel dm) throws TasteException {
            return new GenericItemBasedRecommender(dm, tanimoto);
         }
      });

      recs.put("Slope One",
              new RecommenderBuilder() {
```

```
         public Recommender buildRecommender(DataModel dm) throws TasteException {
            return new SlopeOneRecommender(dm);
         }
      });

      recs.put("SVD with ALS",
              new RecommenderBuilder() {
         public Recommender buildRecommender(DataModel dm) throws TasteException {
            return new SVDRecommender(dm, new ALSWRFactorizer(dm, 20, 1, 20));
         }
      });

      recs.put("SVD with EM",
              new RecommenderBuilder() {
         public Recommender buildRecommender(DataModel dm) throws TasteException {
            return new SVDRecommender(dm, new ExpectationMaximizationSVDFactorizer(dm, 20, 20));
         }
      });

      recs.put("SVD with implicit linear regression",
              new RecommenderBuilder() {
         public Recommender buildRecommender(DataModel dm) throws TasteException {
            return new SVDRecommender(dm, new ImplicitLinearRegressionFactorizer(dm));
         }
      });

      Map<String, Double> results = new TreeMap<String, Double>();

      // Evaluate the various recommenders
      for (String rec: recs.keySet()) {
         double result = 0.0;

         for (int i = 0; i < ITERATIONS; i++) {
            result += rmse.evaluate(recs.get(rec), null, ratingDataModel, 0.9, 1.0);
         }

         results.put(rec, result / ITERATIONS);
      }

      // Print the results iafter Mahout is done spewing log messages
      for (String rec: results.keySet()) {
         System.out.printf("%s: %.4f\n", rec, results.get(rec));
      }
   }
}
```

3. Build and run the Maven project

To build and run your Maven project, enter the following commands. Please note: depending on exactly where you put your Maven project, the path to the data files may differ.

```
$ mvn compile
…
BUILD SUCCESS
...
$ mvn exec:java -Dexec.mainClass="ccp.challenge1.recommender.Recommend"
…
Item-item with Tanimoto: 1.2142
Item-item with log-likelihood: 1.2151
SVD with ALS: 1.9169
SVD with EM: 1.2284
SVD with implicit linear regression: NaN
Slope One: 0.7175
User-user n-nearest neighbor with Tanimoto: NaN
User-user n-nearest neighbor with log-likelihood: NaN
```

```
User-user threshold with Tanimoto: NaN
User-user threshold with log-likelihood: NaN
```

These results reveal several things. First, the user-user algorithm doesn't have enough data to produce results and thus return "NaN", not a number. You could fix the issue by patching in the global average rating in the cases when the data is insufficient to make a prediction, but given how sparse the data is, that's likely to be the majority of the cases, leaving the results little different from just guessing the global average rating for all cases. Second, the item-item and SVD with EM algorithms do better than the baseline, but not by much. Third, the Slope One algorithm does very well and looks like it's the clear winner. Finally, none of the other algorithms do as well as the baseline, so you can ignore them.

Now that you have some contenders, you should iterate on those algorithms to see if you can improve the results any. For the item-item recommender, there are some item selection algorithm tweaks you can make, but given the failure of the user-user algorithms, it doesn't look like any of the other options will improve your results. For the Slope One algorithm, you have a couple of different weighting options. For SVD with EM you have the option to adjust the number of latent features used and the number of iterations.

4. Refine your recommender

Create a copy of the `Recommend.java` file called `Recommend2.java` with the following contents:

```
package com.cloudera.ccp.recommender;

import java.io.File;
import java.util.HashMap;
import java.util.Map;
import java.util.TreeMap;
import org.apache.mahout.cf.taste.common.TasteException;
import org.apache.mahout.cf.taste.common.Weighting;
import org.apache.mahout.cf.taste.eval.RecommenderBuilder;
import org.apache.mahout.cf.taste.eval.RecommenderEvaluator;
import org.apache.mahout.cf.taste.impl.eval.RMSRecommenderEvaluator;
import org.apache.mahout.cf.taste.impl.model.file.FileDataModel;
import org.apache.mahout.cf.taste.impl.recommender.GenericItemBasedRecommender;
import org.apache.mahout.cf.taste.impl.recommender.slopeone.MemoryDiffStorage;
import org.apache.mahout.cf.taste.impl.recommender.slopeone.SlopeOneRecommender;
import org.apache.mahout.cf.taste.impl.recommender.svd.ExpectationMaximizationSVDFactorizer;
import org.apache.mahout.cf.taste.impl.recommender.svd.SVDRecommender;
import org.apache.mahout.cf.taste.impl.similarity.LogLikelihoodSimilarity;
import org.apache.mahout.cf.taste.impl.similarity.TanimotoCoefficientSimilarity;
import org.apache.mahout.cf.taste.model.DataModel;
import org.apache.mahout.cf.taste.recommender.Recommender;

public class Recommend2 {
   private static final int ITERATIONS = 10;

   public static void main(String[] args) throws Exception {
      final DataModel ratingDataModel = new FileDataModel(new File("../explicit.csv"));
      final DataModel allDataModel = new FileDataModel(new File("../implicit.csv"));
      final LogLikelihoodSimilarity ll = new LogLikelihoodSimilarity(allDataModel);
      final TanimotoCoefficientSimilarity tanimoto = new TanimotoCoefficientSimilarity(allDataModel);
      RecommenderEvaluator rmse = new RMSRecommenderEvaluator();
      Map<String, RecommenderBuilder> recs = new HashMap<String, RecommenderBuilder>();

      recs.put("Item-item with log-likelihood",
              new RecommenderBuilder() {
         public Recommender buildRecommender(DataModel dm) throws TasteException {
            return new GenericItemBasedRecommender(dm, ll);
         }
      });
      recs.put("Item-item with Tanimoto",
              new RecommenderBuilder() {
         public Recommender buildRecommender(DataModel dm) throws TasteException {
            return new GenericItemBasedRecommender(dm, tanimoto);
         }
      });
      recs.put("Slope One unweighted",
```

```
            new RecommenderBuilder() {
         public Recommender buildRecommender(DataModel dm) throws TasteException {
            return new SlopeOneRecommender(dm, Weighting.UNWEIGHTED, Weighting.UNWEIGHTED, new
MemoryDiffStorage(dm, Weighting.UNWEIGHTED, Long.MAX_VALUE));
         }
      });
      recs.put("Slope One weighted",
            new RecommenderBuilder() {
         public Recommender buildRecommender(DataModel dm) throws TasteException {
            return new SlopeOneRecommender(dm, Weighting.WEIGHTED, Weighting.UNWEIGHTED, new
MemoryDiffStorage(dm, Weighting.UNWEIGHTED, Long.MAX_VALUE));
         }
      });
      recs.put("Slope One weighted with stddev",
            new RecommenderBuilder() {
         public Recommender buildRecommender(DataModel dm) throws TasteException {
            return new SlopeOneRecommender(dm, Weighting.WEIGHTED, Weighting.WEIGHTED, new
MemoryDiffStorage(dm, Weighting.WEIGHTED, Long.MAX_VALUE));
         }
      });
      recs.put("SVD with EM with 20 features and 10 iterations",
            new RecommenderBuilder() {
         public Recommender buildRecommender(DataModel dm) throws TasteException {
            return new SVDRecommender(dm, new ExpectationMaximizationSVDFactorizer(dm, 20, 10));
         }
      });
      recs.put("SVD with EM with 20 features and 20 iterations",
            new RecommenderBuilder() {
         public Recommender buildRecommender(DataModel dm) throws TasteException {
            return new SVDRecommender(dm, new ExpectationMaximizationSVDFactorizer(dm, 20, 20));
         }
      });
      recs.put("SVD with EM with 20 features and 40 iterations",
            new RecommenderBuilder() {
         public Recommender buildRecommender(DataModel dm) throws TasteException {
            return new SVDRecommender(dm, new ExpectationMaximizationSVDFactorizer(dm, 20, 40));
         }
      });
      recs.put("SVD with EM with 10 features and 20 iterations",
            new RecommenderBuilder() {
         public Recommender buildRecommender(DataModel dm) throws TasteException {
            return new SVDRecommender(dm, new ExpectationMaximizationSVDFactorizer(dm, 10, 20));
         }
      });
      recs.put("SVD with EM with 40 features and 20 iterations",
            new RecommenderBuilder() {
         public Recommender buildRecommender(DataModel dm) throws TasteException {
            return new SVDRecommender(dm, new ExpectationMaximizationSVDFactorizer(dm, 40, 20));
         }
      });

      Map<String, Double> results = new TreeMap<String, Double>();

      // Evaluate the various recommenders
      for (String rec: recs.keySet()) {
         double result = 0.0;

         for (int i = 0; i < ITERATIONS; i++) {
            result += rmse.evaluate(recs.get(rec), null, ratingDataModel, 0.9, 1.0);
         }
         results.put(rec, result / ITERATIONS);
      }

      // Print the results iafter Mahout is done spewing log messages
      for (String rec: results.keySet()) {
         System.out.printf("%s: %.4f\n", rec, results.get(rec));
```

```
      }
   }
}
```

5. Build and run your second recommender

To build and run your refined recommender, enter the following commands:

```
$ mvn compile
…
BUILD SUCCESS
...
$ mvn exec:java -Dexec.mainClass="ccp.challenge1.recommender.Recommend2"
...
Item-item with Tanimoto: 0.8728
Item-item with log-likelihood: 1.2331
SVD with EM with 10 features and 20 iterations: 1.0779
SVD with EM with 20 features and 10 iterations: 1.1918
SVD with EM with 20 features and 20 iterations: 1.2059
SVD with EM with 20 features and 40 iterations: 1.1135
SVD with EM with 40 features and 20 iterations: 1.1342
Slope One unweighted: 0.8043
Slope One weighted: 0.8275
Slope One weighted with stddev: 1.0222
```

Notice that even though the item-item parameters didn't change, the results are very different. Given the sparseness of the data, this likely signals that the randomness in the training data sampling has a disproportionately large influence, even averaged over 10 iterations. As such, you should interpret the close item-item result as a fluke and discard it. What you can see from these results is that unweighted Slope One is the winner, with weighted Slope One close behind.

Keep in mind, those RMSE values ignore the items for which no rating could be estimated. If you go back and dig through the Mahout output, you'll see that no algorithm was able to generate estimates for more than a handful of user/item pairs. This is typical for a sparse data set.

Step 4. Generate ratings

Now that we have an algorithm, we need to generate ratings. To generate the specific ratings in the `rateme.csv` file, you need to read in that file and have the recommender evaluate the user/item pair on each line then write out every estimate to an output file. When you were evaluating algorithms, the evaluator merrily ignored user/item pairs for which no estimate could be generated. Here you no longer have that option, so you need a fallback plan. You know from your baseline work that the global average works out better than the user averages. When you encounter a user/item pair that cannot be rated, you should therefore fall back to the global average. Of course, the global average you computed when determing the baselines was for only about 70% of the data. You need to go back and recompute for the full data set.

1. Recompute the global average for the full data set

```
$ hive
hive> SELECT avg(IF(a.rating IS NULL, b.rating, a.rating)) AS avg FROM explicit_train a FULL OUTER
JOIN explicit_test b ON (a.user == b.user and a.item == b.item);
...
OK
3.683585313174946
Time taken: 7.881 seconds
```

The results are different, but not by much.

2. Create your recommender

Now, you're ready to create your recommender using the Slope One algorithm. Since Slope One is so much better than the item-item options, it's probably worth the performance penalty to use a local recommender rather than a distributed item-item recommender.

Create a new file in the Maven project called `FinalRecommend.java` with the following contents:

```
package com.cloudera.ccp.recommender;
```

```
import java.io.File;
import java.io.PrintWriter;
import org.apache.mahout.cf.taste.common.NoSuchItemException;
import org.apache.mahout.cf.taste.common.NoSuchUserException;
import org.apache.mahout.cf.taste.impl.model.file.FileDataModel;
import org.apache.mahout.cf.taste.impl.recommender.slopeone.SlopeOneRecommender;
import org.apache.mahout.cf.taste.model.DataModel;
import org.apache.mahout.cf.taste.recommender.Recommender;
import org.apache.mahout.common.iterator.FileLineIterable;

public class FinalRecommend {
   private static final Float GLOBAL_AVERAGE = 3.6836f;

   public static void main(String[] args) throws Exception {
      DataModel explicitDataModel = new FileDataModel(new File("../explicit.csv"));
      Recommender recommender = new SlopeOneRecommender(explicitDataModel);
      PrintWriter out = new PrintWriter("../Task3Solution.csv");
      File in = new File("../rateme.csv");

      for (String testDatum : new FileLineIterable(in)) {
         String[] tokens = testDatum.split(",");
         String itemIdString = tokens[1];
         long userId = Long.parseLong(tokens[0]);
         long itemId = Long.parseLong(itemIdString);
         float estimate;

         try {
            estimate = recommender.estimatePreference(userId, itemId);
         } catch(NoSuchUserException e) {
            estimate = GLOBAL_AVERAGE;
         } catch(NoSuchItemException e) {
            estimate = GLOBAL_AVERAGE;
         }

         if (Float.isNaN(estimate)) {
            estimate = GLOBAL_AVERAGE;
         }

         if (itemId > 50000) {
            int i = itemIdString.lastIndexOf("00");

            itemIdString = itemIdString.substring(0, i) + 'e' + itemIdString.substring(i+2);
         }

         out.printf("%d,%s,%.4f\n", userId, itemIdString, estimate);
      }

      out.close();
   }
}
```

3. Build your project and run the file

Build the Maven project and run the file with the following commands:

```
$ cd <projectdir>
$ mvn compile
…
BUILD SUCCESS
...
$ mvn exec:java -Dexec.mainClass="ccp.challenge1.recommender.FinalRecommend"
...
```

4. Analyze the results

Analyze the results with the following commands:

```
$ wc -l ../Task3Solution.csv
    1000 ../Task3Solution.csv
$ head ../Task3Solution.csv
91059173,6155,3.6836
34025317,39419,3.6836
15309904,11248,3.6836
60633959,18963,3.6836
31917316,36814,3.6836
53736706,26212,3.6836
33961447,22606,3.6836
93036062,39815,3.6836
93165942,32989,3.6836
91328973,26806,3.6836
```

The first ten results all use the global average. Let's see how many use it in total:

```
$ grep 3.6836 ../Task3Solution.csv | wc -l
     852
```

Looks like about 85% of the results are just the global average. Given how sparse the data is and the results we saw while evaluating the algorithms, that's not too bad. Nonetheless, let's see how the other top algorithms fare.

5. Compare algorithms: Tanimoto (Jacard)

Open up `FinalRecommend.java` and paste in the item-item recommender with Tanimoto similarity metric from `Recommend.java` (changes in bold red):

```
package com.cloudera.ccp.recommender;

import java.io.File;
import java.io.PrintWriter;
import org.apache.mahout.cf.taste.common.NoSuchItemException;
import org.apache.mahout.cf.taste.common.NoSuchUserException;
import org.apache.mahout.cf.taste.impl.model.file.FileDataModel;
import org.apache.mahout.cf.taste.impl.recommender.GenericItemBasedRecommender;
import org.apache.mahout.cf.taste.impl.similarity.TanimotoCoefficientSimilarity;
import org.apache.mahout.cf.taste.model.DataModel;
import org.apache.mahout.cf.taste.recommender.Recommender;
import org.apache.mahout.common.iterator.FileLineIterable;

public class FinalRecommend {
   private static final Float GLOBAL_AVERAGE = 3.6836f;
   public static void main(String[] args) throws Exception {
      DataModel explicitDataModel = new FileDataModel(new File("../explicit.csv"));
      DataModel implicitDataModel = new FileDataModel(new File("../implicit.csv"));
      TanimotoCoefficientSimilarity tanimoto = new TanimotoCoefficientSimilarity(implicitDataModel);
      Recommender recommender = new GenericItemBasedRecommender(explicitDataModel, tanimoto);
      PrintWriter out = new PrintWriter("../Task3Solution.csv");
      File in = new File("../rateme.csv");

      for (String testDatum : new FileLineIterable(in)) {
         String[] tokens = testDatum.split(",");
         String itemIdString = tokens[1];
         long userId = Long.parseLong(tokens[0]);
         long itemId = Long.parseLong(itemIdString);
         float estimate;

         try {
            estimate = recommender.estimatePreference(userId, itemId);
         } catch(NoSuchUserException e) {
            estimate = GLOBAL_AVERAGE;
```

```
         } catch(NoSuchItemException e) {
            estimate = GLOBAL_AVERAGE;
         }

         if (Float.isNaN(estimate)) {
            estimate = GLOBAL_AVERAGE;
         }

         if (itemId > 50000) {
            int i = itemIdString.lastIndexOf("00");

            itemIdString = itemIdString.substring(0, i) + 'e' + itemIdString.substring(i+2);
         }

         out.printf("%d,%s,%.4f\n", userId, itemIdString, estimate);
      }

      out.close();
   }
}
```

6. Build the project and run

Build the Maven project and run the file with the following commands:

```
$ mvn compile
…
BUILD SUCCESS
...
$ mvn exec:java -Dexec.mainClass="ccp.challenge1.recommender.FinalRecommend"
…
```

7. Analyze the results

Analyze the results with the following commands:

```
$ wc -l ../Task3Solution.csv
    1000 ../Task3Solution.csv
$ grep 3.6836 ../Task3Solution.csv | wc -l
     996
```

Definitely not an improvement. Try another.

8. Compare algorithms: SVD with EM

Open up `FinalRecommend.java` and paste in the item-item recommender with SVD `Recommend.java` (changes in bold red):

```
package com.cloudera.ccp.recommender;

import java.io.File;
import java.io.PrintWriter;
import org.apache.mahout.cf.taste.common.NoSuchItemException;
import org.apache.mahout.cf.taste.common.NoSuchUserException;
import org.apache.mahout.cf.taste.impl.model.file.FileDataModel;
import org.apache.mahout.cf.taste.impl.recommender.svd.ExpectationMaximizationSVDFactorizer;
import org.apache.mahout.cf.taste.impl.recommender.svd.SVDRecommender;
import org.apache.mahout.cf.taste.model.DataModel;
import org.apache.mahout.cf.taste.recommender.Recommender;
import org.apache.mahout.common.iterator.FileLineIterable;

public class FinalRecommend {
   private static final Float GLOBAL_AVERAGE = 3.6836f;
   public static void main(String[] args) throws Exception {
```

```
      DataModel explicitDataModel = new FileDataModel(new File("../explicit.csv"));
      Recommender recommender = new SVDRecommender(explicitDataModel, new
ExpectationMaximizationSVDFactorizer(explicitDataModel, 20, 40));
      PrintWriter out = new PrintWriter("../Task3Solution.csv");
      File in = new File("../rateme.csv");

      for (String testDatum : new FileLineIterable(in)) {
         String[] tokens = testDatum.split(",");
         String itemIdString = tokens[1];
         long userId = Long.parseLong(tokens[0]);
         long itemId = Long.parseLong(itemIdString);
         float estimate;

         try {
            estimate = recommender.estimatePreference(userId, itemId);
         } catch(NoSuchUserException e) {
            estimate = GLOBAL_AVERAGE;
         } catch(NoSuchItemException e) {
            estimate = GLOBAL_AVERAGE;
         }

         if (Float.isNaN(estimate)) {
            estimate = GLOBAL_AVERAGE;
         }

         if (itemId > 50000) {
            int i = itemIdString.lastIndexOf("00");

            itemIdString = itemIdString.substring(0, i) + 'e' + itemIdString.substring(i+2);
         }

         out.printf("%d,%s,%.4f\n", userId, itemIdString, estimate);
      }

      out.close();
   }
}
```

9. Build the project and run

Build the Maven project and run the file with the following commands:

```
$ mvn compile
…
BUILD SUCCESS
...
$ mvn exec:java -Dexec.mainClass="ccp.challenge1.recommender.FinalRecommend"
…
```

10. Analyze the results

Analyze the results with the following commands:

```
$ wc -l ../Task3Solution.csv
    1000 ../Task3Solution.csv
$ grep 3.6836 ../Task3Solution.csv | wc -l
     567
```

These results are quite a bit better, but the Slope One recommender is probably still the better bet, because the results from the evaluation were so much better. Back out your edits and go back to the FinalRecommend implementation that used the SlopeOne recommender. Recompile and rerun the project to get the final results.

Step 5: Revisit the ‘e’

One last thing to check is that our translation for item names with ‘e’ in them is working correctly:

```
$ cd ..
$ grep e Task3Solution.csv
$ grep -E ',\d{6,7},' Task3Solution.csv
```

That's odd. Not only are there no item IDs with 'e' in them, there are also none with item IDs more than 5 digits, i.e. with the '00' translation still present. Look the `rateme.csv` file:

```
$ grep e rateme.csv
$ grep -E ',\d{6,7},' rateme.csv
```

No results. Let's check one more thing – let's see if any of the item IDs in the `rateme.csv` file corresponds to an item ID with an 'e' in it:

```
$ for id in `cat rateme.csv | cut -d, -f2`; do grep ${id}00 implicit.csv; done | head
65810548,18963004
12182640,273140045
52659285,27314001
52659285,27314003
52659285,27314002
86567478,273140015
24807730,344790080
36683624,34479003
36683624,34479002
36683624,34479001
```

This command is a small bash script that iterates over all of the item IDs in the ratings file and uses the `grep` command to find lines containing those IDs in the implicit data file.

As you can see from this output, there are obviously TV shows in the rateme.csv file, but they appear to be missing the episode portion of the ID. We now have to decide what to do with them. If we leave them as is, they'll end up with the global average rating as their estimates. Another option would be to try to build an amalgam of the predicted ratings for each of the episodes in the show. While that plan is easily said, there are lots of complicating details that make it much less easily done. First, we don't know how many episodes are in a show. We would have to estimate with the highest episode version we've encountered in the data. Second, do we count the whole series as one item, or do we break it up into seasons? The later is much more likely to be accurate, but since we also lack information on season length, it would be challenging to implement. (We'd have to infer seasons from viewing behavior, for example.) Finally, let's not forget with the sparsity of the data that most of our predictions are just going to be the global average anyway. Given the level of difficulty and the uncertain results, it's probably not worth tackling. Let's leave the TV shows as they are. Since we're not making any changes to the handling of TV shows, we're done, and the predicted ratings in the rateme.csv are our final results.

Comparing the results with the known ratings for the user/item pairs in the `rateme.csv` file, all of the top recommenders from our analysis perform similarly, in the RMSE 1.20–1.30 range.

Navigation
Table of Contents
Solution Kit Introduction
Project Introduction
Exploring the Data
Cleaning the Data
Classifying Users
Clustering Sessions
Predicting User Ratings (Building a Recommender)
Conclusion **(next)**

Cloudera, Inc.
1001 Page Mill Road Bldg 2
Palo Alto, CA 94304

www.cloudera.com
US: 1-888-789-1488
Intl: 1-650-362-0488