Specifications

This project contains a Web Application and a Java library. The library needed to be developed so the performance of a few Item-Item Collaborative Filtering Recommender Systems can be assessed. The Web Application implements an Item-Item Collaborative Filtering Recommender System.

Algorithm Analysis Library Specifications

The Algorithm Analysis Library must contain classes that would make the performance analysis of collaborative filtering recommender algorithms as easy as possible. Using the library and a .csv file in the form of “user_id, item_id, rating”, applications that test the performance of collaborative filtering algorithms should be easy to develop. The “user_id” represents a unique string identifier of a user that rated an item, the “item_id” represents a unique string identifier of an item that was rated by a user, and the “rating” is the 1 to 5 rating given by the user to the item.

A user should be able to create a data-sample out of the .csv file. Split the data-sample into a training set, and a test set. Use the training set to compute a similarity matrix. Use a collaborative filtering algorithm to compute predicted ratings for the test set. Use the predicted ratings to calculate the metrics that measure the algorithm's performance. Create a .csv file in the form “user_id, item_id, received_rating, predicted_rating”. In addition each algorithm should have the option to be tested using cross-validation.

Algorithm Library Analysis and Design.

Per requirements the library should allow a user to analyze the performance of collaborative filtering algorithms using a .csv file in the form of “user_id, item_id, rating”, where the “user_id” represents a unique string identifier of a user that rated an item, the “item_id” represents a unique string identifier of an item that was rated by a user, and the “rating” is the 1 to 5 rating given by the user to the item.

I designed the library’s classes around the following use-case:

1. Create a set of ratings out of the .csv file.
2. Split the set of ratings into a training set and a test set in such a way that the train set contains 80% of the ratings. The train and test sets, contain each rating category, in approximately the same proportion as the entire set of ratings.
3. Check the performance of a collaborative filtering algorithm on the split data set.
4. Check the performance of a collaborative filtering algorithm using cross-validation.
5. Write out a .csv file in the following form: “user_id, item_id, received_rating, predicted_rating”.

1. Create a set of ratings out of the .csv file.

A quick analysis of the .csv file's data, shows that a class which represents a row, needed to be designed before any other classes. The instances of this class are the set of ratings elements. As shown in the figure Design9, the "Rating" class has three attributes which correspond to an input file row. The "userId" attribute maps to the "user_id" value of an input file row, the "businessId" attribute maps to the "item_id" value of the row, and the "stars" attribute maps to the "rating" value of the input file row.

The set of "Rating" class instances, is wrapped in another class named "DataSample". The ratings set is wrapped in a class so the set can be manipulated and analyzed easily. For example the use-case's second bullet point requires the ratings set to be split into two sets, a training set and a test set. This operation can be done within an instance of the "DataSample" class by using a split method. The method splits the ratings set into two sets based on a double value between 0 and 1, passed in as a parameter by the user. The double value represents the split ratio. For example if the user passes a value of 0.2 as a parameter the list of ratings attribute is split into two lists a test set list, and a train set list. The test set list contains 20% of ratings set's elements, the rest 80% are contained in the training set.

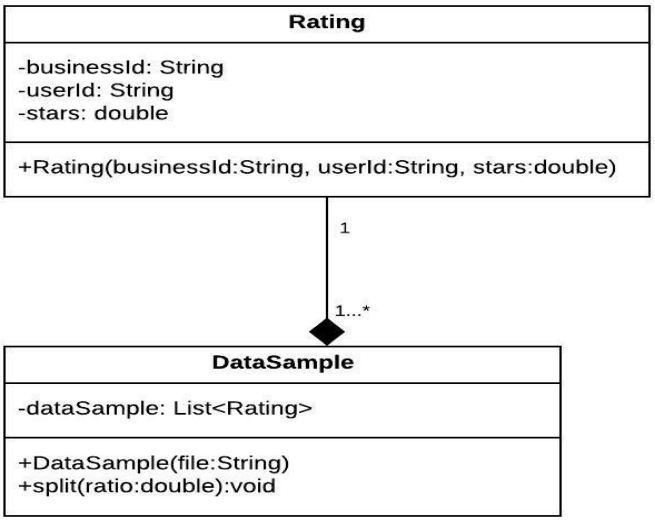

Figure Design9: Rating, and DataSample classes

2. Split the set of ratings into a training set and a test set in such a way that the train set contains 80% of the ratings. The train and test sets, contain each rating category, in approximately the same proportion as the entire set of ratings.

This use-case step is self explanatory. After the split into the training and test sets, each rating category should be found in each set in approximately same proportion as in the set of ratings.

3. Check the performance of a collaborative filtering algorithm on the split data set.

At this step an item-item collaborative filtering algorithm class needed to be designed.

The main function of the algorithm is to predict ratings. The algorithm would use an instance of the "DataSample" class, which is passed trough its constructor to compute a set of predicted ratings. More precisely, the algorithm predicts the rating values of the elements, in the DataSample object's test set attribute. The predicted ratings set contains elements that are instances of a class named "PredictedRating". This class has four attributes: a string that represents the user id, a string that represents the item id, a double that represents the received rating, and a double that represents the predicted rating. A list of predicted ratings that has as its elements instances of the "PredictedRatings" class is an attribute of the algorithm class. In the use-case the algorithm calculates a predicted rating using the formula:

$$pr_{ui} = \frac{\sum similarity(i,j) * rating_{uj}}{\sum similarity(i,j)}$$

where ***j*** is an item that is part of the set of item's ***i*** nearest neighbors. For this example the nearest neighbors set contains up to ten elements that were also rated by the user ***u***.

A quick analysis of the formula reveals that the most expensive computation the algorithm makes, is finding the item's ***i*** nearest neighbors set. This computation can be done faster with the help of a pre-computed item-item similarity matrix. The similarity matrix is a map data structure of the following form: Map<String, Map<String, Double>>. The outer map's key is a string that represents an item's id. The outer map's value is an embedded map that contains the items which have a positive similarity value with the outer map's key item. The value of the embedded map is the similarity value between the outer map's key and the embedded map's key. A class named SimilarityMatrix was designed. The class contains the methods which compute and make available for use the similarity matrix in the data structure form described above.

In order to be included in the predicted rating computation, the items that have a positive similarity value with item ***i*** must have also been rated by the user ***u***. To isolate these items faster a class named ItemUtilityMatrix was designed. The class uses a list of ratings objects to create a map data

structure in the following form: Map<String, Map<String, Double>>. The outer map's key is a string that represents an item's id. The outer map's value is an embedded map that contains as keys the users that rated the outer map's key item. The value of the embedded map is the rating given by the user to the item. For this use-case the instance of the ItemUtilityMatrix class creates a map such as described above using the train set created by the DataSample object's *"split"* method.

The algorithm's performance is measured by using the **Mean Absolute Error(MAE)**, and the **Root Mean Square Error(RMSE)**. The formulas are applied to the set of PredictedRating objects computed by the algorithm. The two metrics are calculated using two static methods that take as parameter a list of PredictedRatind objects. The methods are encapsulated in a class named Metrics.

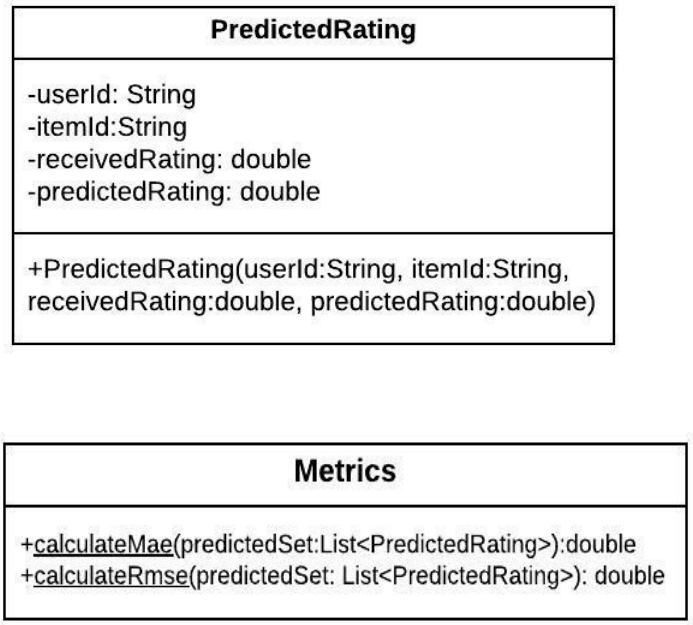

Figure Design10: The PredictedRating and Metrics classes

4. Check the performance of a collaborative filtering algorithm using cross-validation.

The algorithm class contains a method that takes one integer parameter, and performs stratified cross-validation on the data sample ratings set. Stratified cross-validation is used so each fold contains each rating category, in approximately same proportion as the entire data set. The number of folds is passed in by the user trough the integer parameter.

5. Write out a .csv file in the following form: “user_id, item_id, received_rating, predicted_rating”.

A class named WriteOnFile contains a static method named *‘write’*. The method takes in as parameter a list of PredictedRating objects and writes its elements to a .csv file in the form described above.

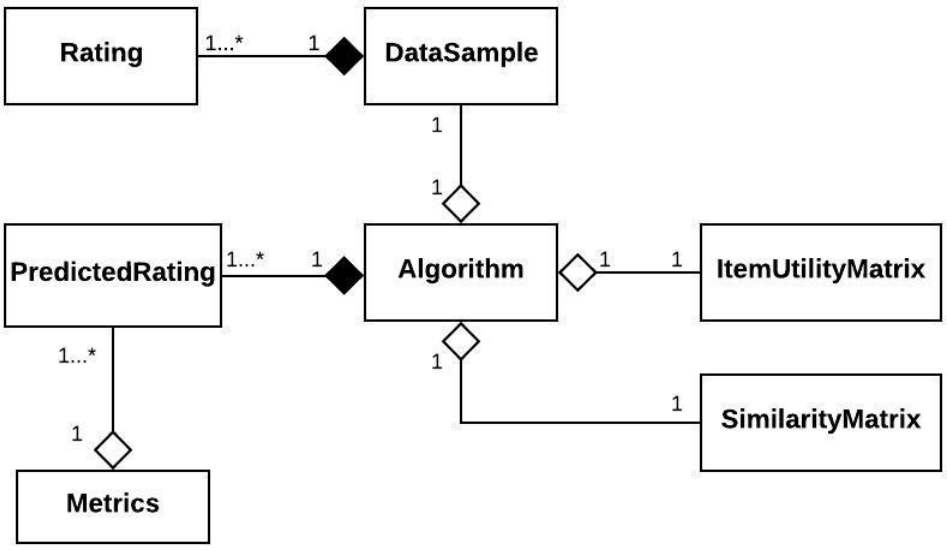

Figure Design11: The classes described in the use-case

Implementation

In the implementation of this project the following tools were used: Java 1.8, JavaScript 6, CSS3, HTML5, MongoDb 4.0, Maven, Apache Tomcat 9.0, and Eclipse IDE.

Algorithm Library

I developed the Java library of collaborative filtering algorithms before the web application so I could test the performance of a few algorithms. The most performant algorithm was integrated in the

web application. The development process of the library, and the testing of a few algorithms on a data set is described in this section.

Data Set

The data set used was extracted from the web application's database. A .csv file in the form of "user_id, item_id, rating" was created. The file contains all the ratings of locations which received more than 500 ratings.

DataSample Class

Per design requirements the DataSample class must create and encapsulate a set of Rating objects by reading the .csv file passed trough its constructor. The class must also contain a method that allows the user to split the data set into a train set and a test set.

The constructor first creates a list of ratings. A row in the file is an instance of a Rating class which is added to the list attribute named rawData. After all the rows are added to the rawData attribute, the constructor uses this attribute to add elements to the dataSample attribute. In order to be used by a collaborative filtering algorithm the dataSample list must not contain any duplicate elements. Two or more elements are considered duplicates if they have the same userId and the same itemId attribute values. In other words if the sample data contains two or more ratings given by the same user to the same item only one of the ratings will be included to the dataSample list.

After properly setting the dataSample attribute, the constructor uses the attribute to set the attributes that contain a list of ratings for each rating category. In addition, it sets the itemIds, userIds, and sampleAverage attributes. itemIds is a list of strings that represents all the item ids sorted in the natural order, userIds is a list of strings that represents all the user ids sorted in the natural order. The sampleAverage is the data sample's average rating value.

The split method sets the trainSet and testSet attributes. It uses the double parameter passed in by the user to add to the testSet list the desired number of elements. The remaining elements are added to the trainSet. The method ensures the testSet list contains each rating category in approximately the same ratio as the entire data sample.

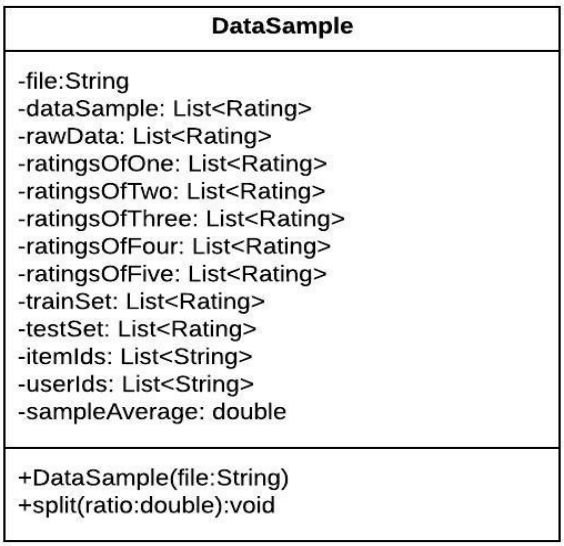

Figure Implementation1: DataSample class

ItemUtilityMatrix Class

The main attribute of this class is the map in the form of Map<String, Map<String, Double>>, where the map's key is a string that represents an item's id, and map's value is an embedded map. The embedded map contains as keys the user ids that rated the outer map's key. The value of the embedded map is the rating given by the user to the item.

This class has a constructor that takes as parameters a list of item ids strings, and a list of Rating instances (the train set). The constructor uses the parameters to build a Map as described above, and two more attributes. The two attributes are a double value which represents the average of the train set's ratings, and another Map in the form of Map<String, Double>, named averagesMap. The Map's keys represents the item ids, while its values represents the average rating received by each item.

UserUtilityMatrix Class

This class was added to the library after further analysis of the algorithms implemented. The class is structurally similar to the ItemUtilityMatrix class. The difference between the two classes is that instead of grouping the list of Rating instances by item ids, the class contains the ratings grouped by user ids.

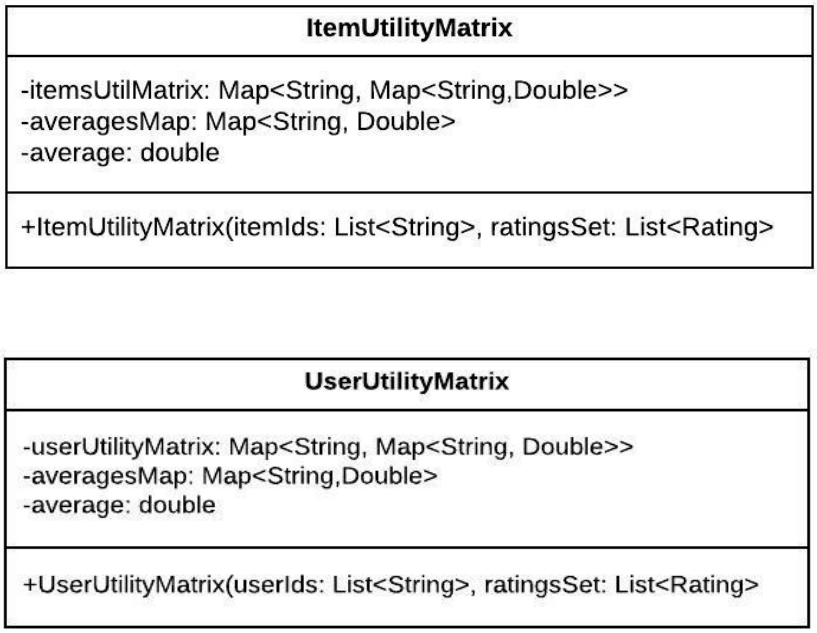

Figure Implementation2: ItemUtilityMatrix and UserUtilityMatrix clases

Similarity Matrix Classes

The library contains two item-item similarity matrix classes. The difference between the classes is that one class uses as a similarity measure between items Cosine Similarity while the other uses Adjusted Cosine Similarity. Both classes contain a Map attribute in the form: Map<String, Map<String, Double>>. The outer map's key is a string that represents an item's id. The outer map's value is an embedded map that contains the items which have a similarity value with the outer map's key item. The value of the embedded map is the similarity value between the outer map's key and the embedded map's key.

The Map attribute is set by the constructor using the *setMatrix()* method. The method uses an instance of ItemUtilityMatrix and an instance of UserUtilityMatrix to build the attribute Map. The class that uses a simple Cosine Similarity as the similarity measure, can use only an instance of the ItemUtilityMatrix class. The class that uses Adjusted Cosine Similarity as the similarity measure needs an instance of each ItemUtilityMatrix and UserUtilityMatrix.

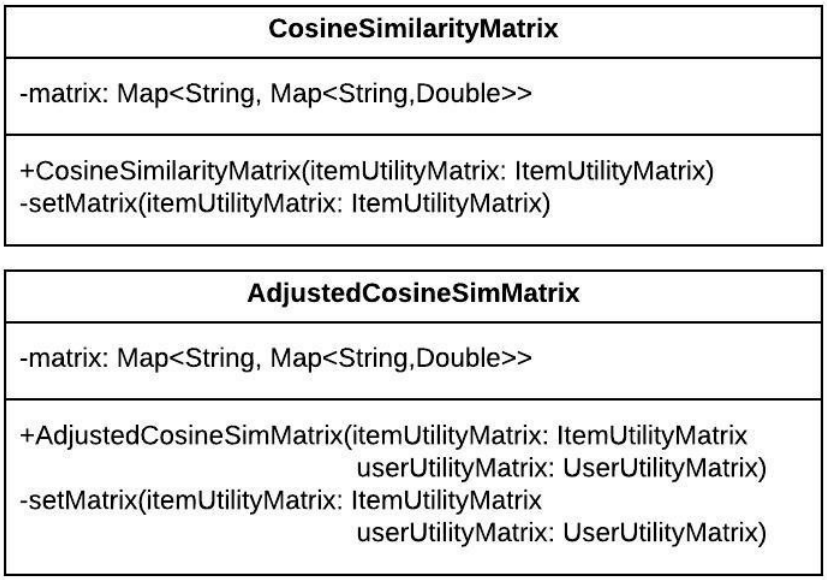

Figure Implementation3: CosineSimilarityMatrix and AdjustedCosineSimMatrix clases

Algorithm Classes

The library contains five algorithm classes. Two of the classes implement item-item collaborative filtering algorithms (one class uses cosine similarity and the other uses adjusted cosine similarity). One class implements the global baseline estimate algorithm, which is an algorithm that predicts ratings based on the formula $pr_{ui} = \mu + bias_u + bias_i$,where "μ" is the data sample's mean rating, $bias_u$ is the average rating of user "u" minus the data sample's mean rating, and $bias_i$ is the average rating of user "i" minus the data sample's mean rating. The remaining two classes implement a

combination of the item-item collaborative filtering algorithms and the global baseline estimate algorithm.

Per the user-case analyzed during the design process, an algorithm class has to use an instance of a DataSample class, to perform two functions. The first function is to compute a collection of PredictedRating objects using the DataSample object's trainSet, and testSet attributes. The second function is to perform stratified n-folds cross-validation. The number of folds is passed in by the user as a parameter.

The most important method in the algorithm class, is a method that predicts a rating for a given userId/itemId pair. The method implements the predicted rating formula, that represents the algorithm.

Global Baseline Estimate Algorithm Class

$$pr_{ui} = \mu + bias_u + bias_i$$

The global baseline estimate algorithm class was applied primarily to be used as a performance measuring starting point. The algorithm is less sophisticated than a collaborative filtering algorithm. It can be used to gauge by how much the predictions accuracy improves when a more sophisticated algorithm is used.

A quick algorithm analysis shows that a method which computes a predicted rating for a given user id and item id, needs information from both ItemUtilityMatrix, and UserUtilityMatrix instances. More precisely, the method needs, a ratings average value, and the averagesMap attribute from each ItemUtilityMatrix, and UserUtilityMatrix objects.

The two functions the algorithm has to perform are implemented by the class as follows:

- The lists of Rating objects which represent the training set, and the test set are extracted from the DataSample instance.
- An ItemUtilityMatrix, and UserUtilityMatrix objects are created using the training set.
- Compute a predicted rating for each element in the test set.
- Perform stratified k-folds cross validation using all the ratings in the DataSample instance. The number of folds is passed in by the user.

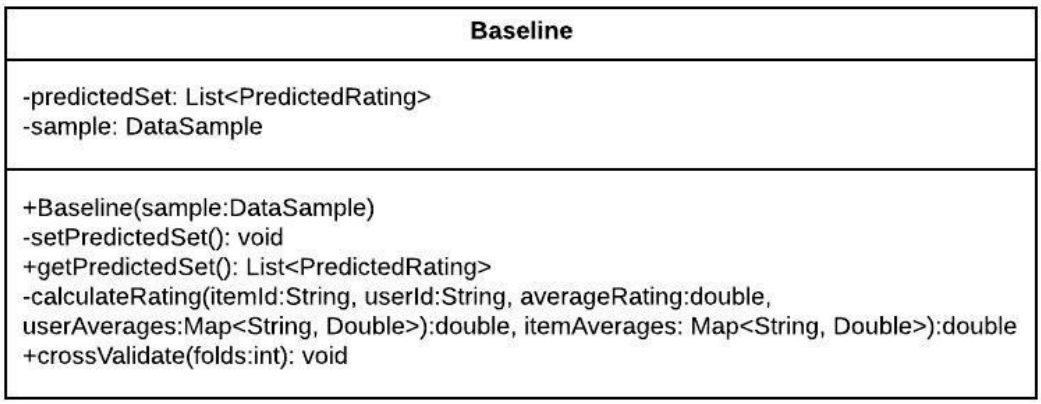

Figure Implementation4: Baseline algorithm class.

```
12 public class Main {
13
14     public static void main(String[] args) {
15
16         DataSample sample = new DataSample("C:\\Users\\costp\\Desktop\\data500.csv");
17         sample.split(0.2);
18
19         Baseline alg1 = new Baseline(sample);
20
21         List<PredictedRating> predictedSet1 = alg1.getPredictedSet();
22
23         System.out.println("************************");
24
25         System.out.println(Metrics.calculateRmse(predictedSet1));
26         System.out.println(Metrics.calculateMae(predictedSet1));
27
28         System.out.println("************************");
29
30         alg1.crossValidate(5);
31
32     |
```

Figure Implementation5: The main method of a Java application that uses the Baseline class.

Figure Implementation5 shows the main method of an application that checks the performance of the global baseline estimate algorithm on the .csv file data sample described in the Data Sample section above.

On line 16 an instance of the DataSample class is created out of the .csv file. On line 17 the ratings set contained by the DataSample object is split into a train set and a test set. The test set contains approximately 20% of the ratings. Line 19 creates an instance of the Baseline class using the

DataSample object. Line 21 extracts the list of PredictedRating objects out of the Baseline instance. Lines 25 and 26 print out to the console the results of the RMSE, and MAE calculations performed by the Metrics class' methods on the list of PredictedRating objects. Line 30 performs 5-fold cross-validation on the data sample using the global baseline estimate algorithm. The output of the program is showed in the Figure Implementation6 bellow.

```
<terminated> Main [Java Application] C:\Program Files\Java\jre1.8.0_261\bin\javaw.exe (Sep 12, 2020, 6:09:23 AM)
************************
1.2602448205496903
0.9524580278481422
************************
 Fold number 1 results are MAE value is 0.9524580278481422 RMSE value is 1.2602448205496903
 Fold number 2 results are MAE value is 0.9556099553588779 RMSE value is 1.2673059461022385
 Fold number 3 results are MAE value is 0.9520250821676832 RMSE value is 1.2615375518234466
 Fold number 4 results are MAE value is 0.9542243063136513 RMSE value is 1.2669827646890892
 Fold number 5 results are MAE value is 0.9518639881323159 RMSE value is 1.2608173645906324
```

Figure Implementation6: Application output.

Item-Item Collaborative Filtering Algorithm Classes

$$pr_{ui} = \frac{\sum similarity(i,j) * rating_{uj}}{\sum similarity(i,j)}$$

Two item-item collaborative filtering algorithm classes were developed, ItemsCosSim class, and ItemsAdjSim class.

The ItemsCosSim class' method that computes a predicted rating for a user id/item id pair needs information from an ItemUtilityMatrix object, and a CosineSimilarityMatrix object. The method that performs the same operation for the ItemsAdjSim class, needs information from an ItemUtilityMatrix object and an AdjustedCosineSimMatrix object.

Since both classes implement the same algorithm, the two functions an algorithm class must implement is described bellow for both classes:

- The lists of Rating objects which represent the training set, and the test set are extracted from the DataSample instance.
- An ItemUtilityMatrix , and UserUtilityMatrix objects are created using the training set.
- A CosineSimilarityMatrix object, or an AdjustedCosineSimMatrix object is created using the ItemUtilityMatrix and the UserUtilityMatrix.
- Compute a predicted rating for each element in the test set.

- Perform stratified k-folds cross validation using all the ratings in the DataSample instance. The number of folds is passed in by the user.

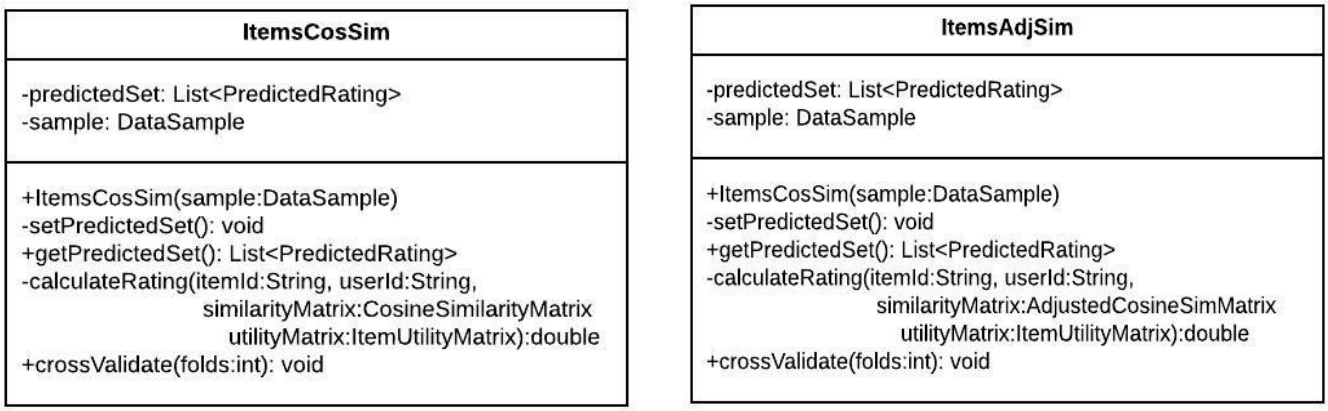

Figure Implementation7: ItemsCosSim, and ItemsAdjSim classes.

Figure Implementation8 shows the main method of an application that checks the performance of the item-item collaborative filtering algorithms on the data sample.

Lines 19 and 29 create instances of ItemsCosSim and ItemsAdjSim classes. Lines 21, 22, 31 and 32 extract the list of PredictedRating objects out of the algorithm instances, and print out to the console the results of the RMSE, and MAE calculations performed by the Metrics class. Lines 24 and 34 perform 5-fold cross-validation on the data sample for each algorithm class. The output of the program is showed in the Figure Implementation9.

Before I analyze the results, a quick note on how the algorithm classes predict ratings in marginal cases. Considering that the algorithms predict a rating for an item/user pair, the following marginal cases were taken into account. If the item does not have any positive similarities with any other item(the item does not have any neighbors), then the predicted rating is the average of the item's ratings. If none of the item's neighbors were rated by the user, then the predicted rating is also the average of the item's ratings.

```java
public class Main {

    public static void main(String[] args) {

        DataSample sample = new DataSample("C:\\Users\\costp\\Desktop\\data500.csv");
        sample.split(0.2);

        ItemsCosSim alg1 = new ItemsCosSim(sample);
        System.out.println("************************");
        System.out.println(Metrics.calculateRmse(alg1.getPredictedSet()));
        System.out.println(Metrics.calculateMae(alg1.getPredictedSet()));
        System.out.println("************************");
        alg1.crossValidate(5);
        System.out.println("************************");
        System.out.println("************************");
        System.out.println();

        ItemsAdjSim alg2 = new ItemsAdjSim(sample);
        System.out.println("************************");
        System.out.println(Metrics.calculateRmse(alg2.getPredictedSet()));
        System.out.println(Metrics.calculateMae(alg2.getPredictedSet()));
        System.out.println("************************");
        alg2.crossValidate(5);
        System.out.println("************************");
        System.out.println("************************");
        System.out.println();
```

Figure Implementation8: The main method of a Java application that uses the ItemsCosSim and ItemsAdjSim classes.

```
************************
1.2960668157197204
0.9593924943048406
************************
 Fold number 1 results are MAE value is 0.9593924943048406 RMSE value is 1.2960668157197204
 Fold number 2 results are MAE value is 0.950944885539522 RMSE value is 1.286348022799335
 Fold number 3 results are MAE value is 0.9483922069497898 RMSE value is 1.2839854471146144
 Fold number 4 results are MAE value is 0.9492873126116138 RMSE value is 1.285576625115182
 Fold number 5 results are MAE value is 0.9515082054719557 RMSE value is 1.291445622035357

************************
************************

************************
1.2842858041783933
0.9632811762474993
************************
 Fold number 1 results are MAE value is 0.9632811762474993 RMSE value is 1.2842858041783933
 Fold number 2 results are MAE value is 0.9569986142790888 RMSE value is 1.278056693147742
 Fold number 3 results are MAE value is 0.9554247927303658 RMSE value is 1.2753064470172086
 Fold number 4 results are MAE value is 0.9512400324096846 RMSE value is 1.2701614007020892
 Fold number 5 results are MAE value is 0.9595608935449134 RMSE value is 1.279040087878804

************************
************************
```

Figure Implementation9: Application output.

Figure Implementation9 shows that the collaborative filtering algorithms performed worse than the global baseline estimate algorithm. I tried to find an explanation for these counter-intuitive results by analyzing the data sample. At this point the only information known about the data sample was that

each item had at least 500 ratings. However, this information does not reveals much about the utility matrix the data sample forms.

An instance of a DataSample class can return a list of Rating objects. More information can be extracted from this list. Figure Implementation10 shows the main method of an application that

```
13 public class Main {
14
15     public static void main(String[] args) {
16
17         DataSample sample = new DataSample("C:\\Users\\costp\\Desktop\\data500.csv");
18
19         List<Rating> data = sample.getSampleData();
20
21         Map<String, List<Rating>> groupedItems = data.parallelStream()
22                                                  .collect(Collectors.groupingBy(Rating::getBusinessId));
23         Map<String, List<Rating>> groupedUsers = data.parallelStream()
24                                                      .collect(Collectors.groupingBy(Rating::getUserId));
25
26         System.out.println("Total number of ratings: " + data.size());
27         System.out.println("Number of items: " + groupedItems.keySet().size());
28         System.out.println("Number of users: " + groupedUsers.keySet().size());
29
```

Problems Javadoc Declaration Console

```
<terminated> Main [Java Application] C:\Program Files\Java\jre1.8.0_261\bin\javaw.exe (Oct 6, 2020, 4:39:17 PM)
Total number of ratings: 283332
Number of items: 356
Number of users: 133407
```

Figure Implementation10: Analysis of the Data Sample.

prints to the console the total number of ratings, the number of items and the number of users, contained by the data sample. There are 283,332 ratings and 133,407 users, which means there are users that do not have many ratings. Since the similarity between two items is calculated by the ratings received from the same users, the positive similarity values are to low, and don't make a big difference in the prediction of a rating. Furthermore, there must be many users that have only one rating, so many user/item pairs in the test set received a predicted rating value equal to the average rating of the item. The items have at least 500 ratings each, so the average rating of an item weights the most when the ratings are predicted. It became clear that a less sparse utility matrix is needed, so I decided to remove all the ratings made by users that made less than 5 ratings from the data sample. The improved data sample has 10226 users, 356 items, and a total of 107,780 ratings. A new .csv file containing the improved data sample was created so it can be used to test the algorithms' performance.

```
******* Baseline Results ****************
1.0430213901064962
0.7918023635240792
************************
 Fold number 1 results are MAE value is 0.7918023635240792 RMSE value is 1.0430213901064962
 Fold number 2 results are MAE value is 0.7893205410690608 RMSE value is 1.0387745760639286
 Fold number 3 results are MAE value is 0.790157187039708 RMSE value is 1.0390849237622395
 Fold number 4 results are MAE value is 0.788261448554104 RMSE value is 1.0331759946362233
 Fold number 5 results are MAE value is 0.7930196157081599 RMSE value is 1.0404942559012416

********** ItemsCosSim Results **************
1.0887071014822258
0.8194856061390021
************************
 Fold number 1 results are MAE value is 0.8194856061390021 RMSE value is 1.0887071014822258
 Fold number 2 results are MAE value is 0.8213415424327131 RMSE value is 1.0893287436194115
 Fold number 3 results are MAE value is 0.8211144519021432 RMSE value is 1.083294380440575
 Fold number 4 results are MAE value is 0.8170450460729933 RMSE value is 1.0811555791103957
 Fold number 5 results are MAE value is 0.8248868940656703 RMSE value is 1.0942219727810598

******* ItemsAdjSim Results *****************
1.1847588890915466
0.8614435144721456
************************
 Fold number 1 results are MAE value is 0.8614435144721454 RMSE value is 1.1847588890915466
 Fold number 2 results are MAE value is 0.8663816330998814 RMSE value is 1.1907714124356106
 Fold number 3 results are MAE value is 0.8603546070266217 RMSE value is 1.1841104963765312
 Fold number 4 results are MAE value is 0.8580377912817881 RMSE value is 1.1793540415167325
 Fold number 5 results are MAE value is 0.8630001138129527 RMSE value is 1.1885611854259677
```

Figure Implementation11: Results of Improved Data Sample.

As Figure Implementation11 shows, even though the results improved, the global baseline estimate algorithm still had the best performance.

Combined Global Baseline Estimate and Item-Item Collaborative Filtering Algorithm Classes

$$pr_{ui} = baseline_{ui} + \frac{\sum (similarity(i,j) * (rating_{uj} - baseline_{uj}))}{\sum similarity(i,j)}$$

Two classes that implement a combined global baseline estimate algorithm and an item-item collaborative filtering algorithm were developed based on the formula above. The algorithm first calculates the global baseline estimate for a given userId/itemId pair then it calculates the item-item collaborative filtering algorithm rating for the same userId/itemId pair. The advantage of calculating a rating by combining these two algorithms is that the pair receives a predicted rating equal to the value

calculated by the global baseline estimate in the case there are no neighbors of the itemId that were rated by the userId.

As was the case with the item-item collaborative filtering algorithm classes, the two classes that implement the combined algorithms perform the two functions the same way, so the implementation described bellow applies to both classes. The only difference is that one class uses an instance of an CosineSimilarityMatrix, and the other uses an AdjustedCosineSimMatrix instance. The classes implement the functions as such:

- The lists of Rating objects which represent the training set, and the test set are extracted from the DataSample instance.
- An ItemUtilityMatrix , and UserUtilityMatrix objects are created using the training set.
- A CosineSimilarityMatrix object, or an AdjustedCosineSimMatrix object is created using the ItemUtilityMatrix and the UserUtilityMatrix.
- Compute a predicted rating for each element in the test set.
  Perform stratified k-folds cross validation using all the ratings in the DataSample instance. The number of folds is passed in by the user.

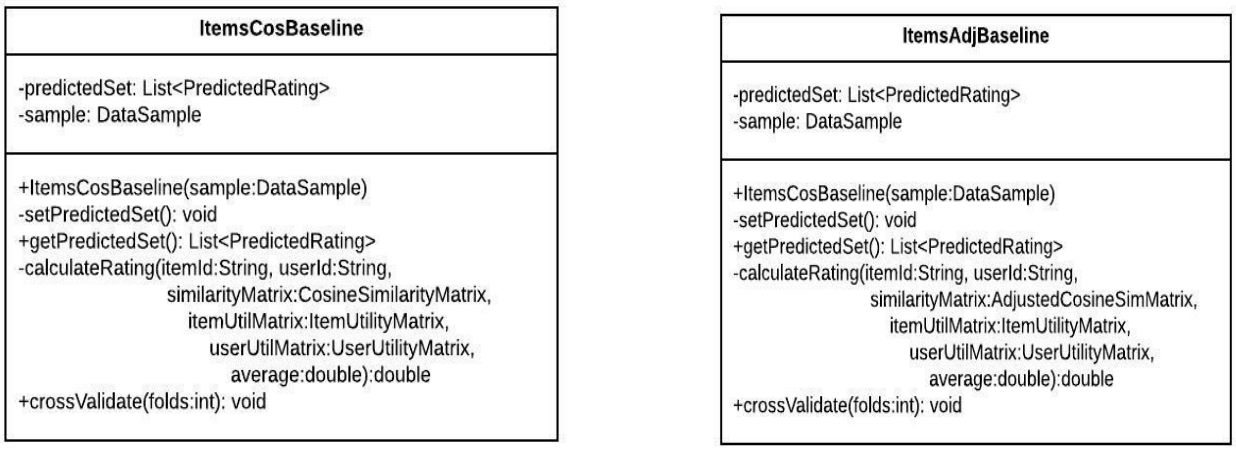

Figure Implementation12: The Combined Algorithms Classes.

Figure Implementation13 shows the main method of a Java application. It uses the improved data sample to test the performance of the algorithms implemented by the ItemsCosBaseline, and ItemsAdjBaseline classes. Figure Implementation14 shows the results improved slightly compared with the results obtained when the ItemsCosSim, and ItemsAdjSim classes were used.

```java
public class Main {

    public static void main(String[] args) {

        DataSample sample = new DataSample("C:\\Users\\costp\\Desktop\\improved500.csv");

        sample.split(0.2);

        ItemsCosBaseline alg1 = new ItemsCosBaseline(sample);
        System.out.println("******* ItemsCosBaseline Results *****************");
        System.out.println(Metrics.calculateRmse(alg1.getPredictedSet()));
        System.out.println(Metrics.calculateMae(alg1.getPredictedSet()));
        System.out.println("************************");
        alg1.crossValidate(5);


        ItemsAdjBaseline alg2 = new ItemsAdjBaseline(sample);
        System.out.println("********** ItemsAdjBaseline Results **************");
        System.out.println(Metrics.calculateRmse(alg2.getPredictedSet()));
        System.out.println(Metrics.calculateMae(alg2.getPredictedSet()));
        System.out.println("************************");
        alg2.crossValidate(5);

    }
```

Figure Implementation13: The Main Method of an Application That Implements the Combined Algorithms Classes.

```
******* ItemsCosBaseline Results *****************
1.0272586663654977
0.776684236392054
************************
 Fold number 1 results are MAE value is 0.776684236392054 RMSE value is 1.0272586663654977
 Fold number 2 results are MAE value is 0.7765647119097792 RMSE value is 1.0308094563094936
 Fold number 3 results are MAE value is 0.7822140010516347 RMSE value is 1.038039331023561
 Fold number 4 results are MAE value is 0.7824218539688084 RMSE value is 1.0373120173098944
 Fold number 5 results are MAE value is 0.7818403665875239 RMSE value is 1.0376755597727547

********** ItemsAdjBaseline Results **************
1.1421566811521968
0.8386486768951312
************************
 Fold number 1 results are MAE value is 0.8386486768951312 RMSE value is 1.1421566811521968
 Fold number 2 results are MAE value is 0.8396018301566717 RMSE value is 1.1426753855517227
 Fold number 3 results are MAE value is 0.8454775314743415 RMSE value is 1.1493903709025153
 Fold number 4 results are MAE value is 0.8417771418982192 RMSE value is 1.1424881196026546
 Fold number 5 results are MAE value is 0.8460432104047514 RMSE value is 1.151666218329758
```

Figure Implementation14: Application Output.