Assignment #4, Recommendation System

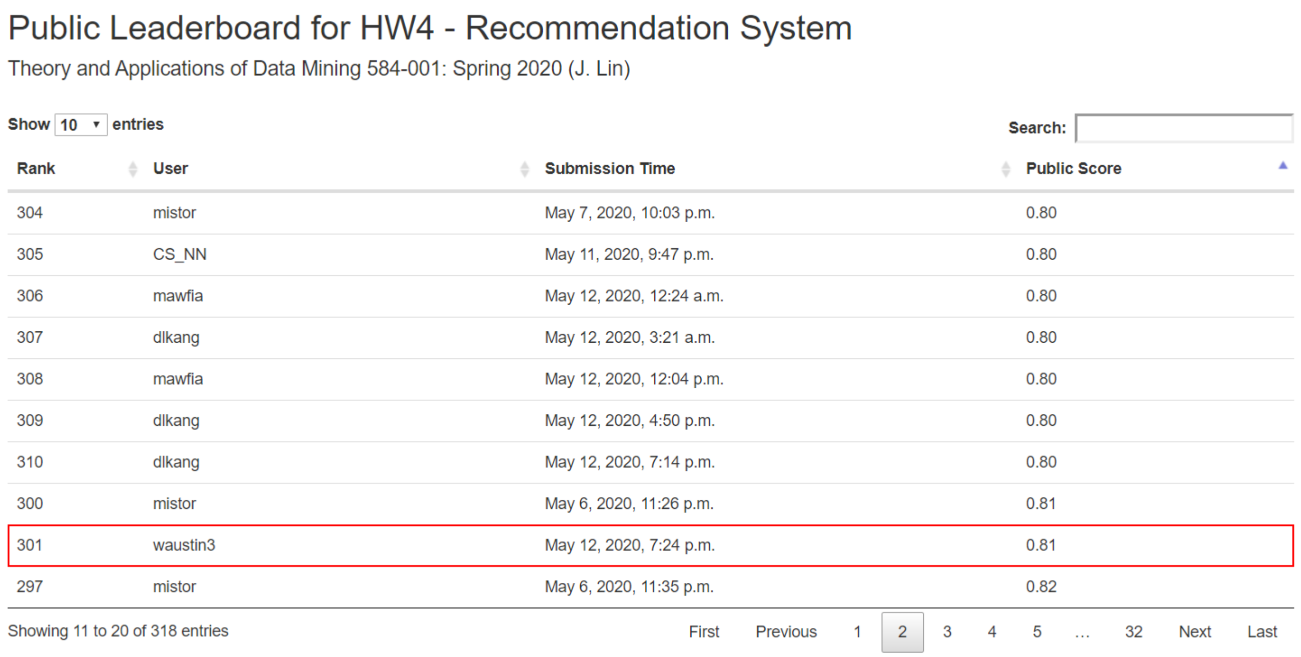
CS 584: Data Mining
Due May 12th, 2020
Submitted by William Austin

1. Miner Website Credentials

Registered Miner Username: **waustin3**

2. Rank & Accuracy of Submission

My best and final Miner submission for this assignment (submission #3, on May 12th) had an RMSE (root mean square error) values of **0.81**. At the time of this writing, this value is the 7th best value in the class.



Public Leaderboard for HW4 - Recommendation System

Theory and Applications of Data Mining 584-001: Spring 2020 (J. Lin)

Show 10 entries

Search:

Rank	User	Submission Time	Public Score
304	mistor	May 7, 2020, 10:03 p.m.	0.80
305	CS_NN	May 11, 2020, 9:47 p.m.	0.80
306	mawfia	May 12, 2020, 12:24 a.m.	0.80
307	dlkang	May 12, 2020, 3:21 a.m.	0.80
308	mawfia	May 12, 2020, 12:04 p.m.	0.80
309	dlkang	May 12, 2020, 4:50 p.m.	0.80
310	dlkang	May 12, 2020, 7:14 p.m.	0.80
300	mistor	May 6, 2020, 11:26 p.m.	0.81
301	waustin3	May 12, 2020, 7:24 p.m.	0.81
297	mistor	May 6, 2020, 11:35 p.m.	0.82

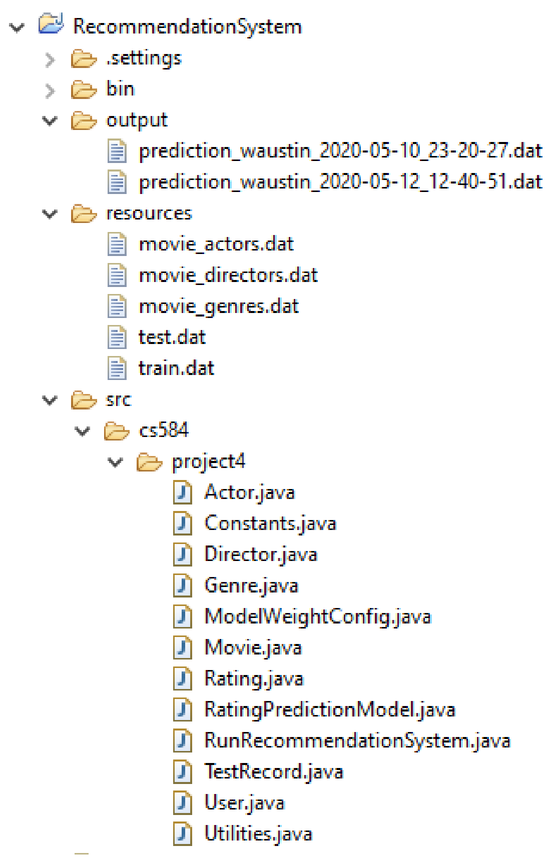
Showing 11 to 20 of 318 entries

My only other submission for the project were:

- Submission #1. May 10th, RMSE = 0.86
- Submission #2. May 12th, RMSE = 0.83

3. Instructions for Running the Code

I implemented my application in Java, and my JDK version is 1.8. I chose Java because it's the language I am most comfortable with, and I had never implemented a project like this before in Java. The project structure is typical for Java applications, with the **`src/`** directory being the class path directory for the code. All my code is in the **`cs584.project4`** package. The Eclipse structure is shown below:



```
RecommendationSystem
    .settings
    bin
    output
        prediction_waustin_2020-05-10_23-20-27.dat
        prediction_waustin_2020-05-12_12-40-51.dat
    resources
        movie_actors.dat
        movie_directors.dat
        movie_genres.dat
        test.dat
        train.dat
    src
        cs584
            project4
                Actor.java
                Constants.java
                Director.java
                Genre.java
                ModelWeightConfig.java
                Movie.java
                Rating.java
                RatingPredictionModel.java
                RunRecommendationSystem.java
                TestRecord.java
                User.java
                Utilities.java
```

To look at the code, start with **RunRecommendationSystem.java**. The **`main()`** method in this class contains comments for all of the different activities I tried over the course of the project and can be used to trace to the implentation as well. Most of the code for the project and the key algorithms for rating prediction is in **RatingPredictionModel.java**.

Note that in my submission, I am omitting the contents of my **`resources/`** directory. For my project, it contains the training file, test file, format file, and supplemental data files, which are large and have already been provided.

There are no external libraries or dependencies for the code other than the Java standard system libraries. My code for this project is also available on GitHub. The URL is:

https://github.com/william-r-austin/RecommendationSystem

4. High-Level Approach and Discussion

Most of the time that I spent on this project falls into 3 main categories:

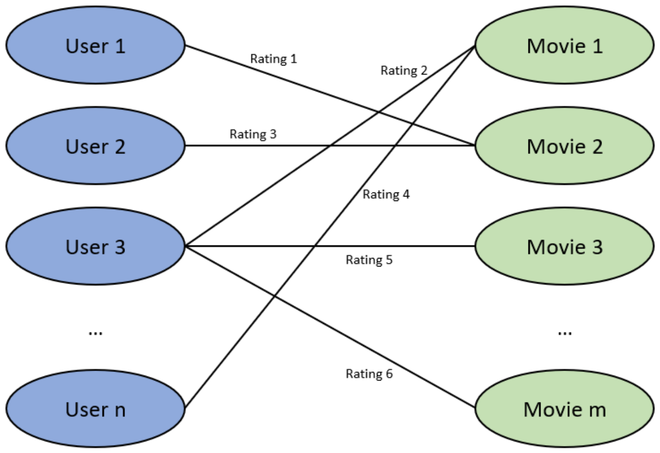
- Brain-storming new potential methods for estimating ratings.
- Creating the implementation for these ideas, and updating the code to support all required operations, including cross-validation support.
- Testing, parameter tuning, and experimenting with different methods to yield improved results.

In this section, I'll highlight the main activities that I performed over the course of completing this project. I'll discuss what did and did not work, and if applicable, where to find the implementation in my code. The sections are presented in roughly sequential order.

1. Data Representation

The first thing I did as I was starting this project was to parse the provided training and test files. However, it is not obvious what data structures to use for storing the data. I wanted to make sure that the full training and test datasets could fit into memory (16 GB for my laptop), along with any supplemental data. Reading the training and test data both take place in the **RatingPredictionModel.java** class, in the **`readTrainingData()`** and **`readTestData()`** methods.

After thinking about the problem, I realized that the training data could be represented as a bipartite map, with two distinct types of nodes, as shown below.



Given this data structure, I decided to model the problem two **HashMap** objects and a **List**. These are:

- **`Map<User, Set<Rating>> userRatingMap`**
- **`Map<Movie, Set<Rating>> movieRatingMap`**
- **`List<Rating> ratingList`**

In this case the **`User`** and **`Movie`** objects roughly model the nodes. A **`Rating`** models an edge in the graph, with pointers to its **`User`** and **`Movie`**. As we will see, this turned out to be a good decision because it allows us to access most information in constant time, and it supports arbitrary operations on the data.

2. Implement Cross-Validation and Basic Rating Method

A very basic approach to this problem is simply to create average ratings for each user and movie. This has the benefit of simplicity and is very efficient to compute. In my code, I maintain a map with the results of these and **RatingPredictionModel.java**.

For prediction purposes, I determined it would make more sense to compute a prediction directly based on the other ratings for the user, rather than for the movie. If we think of each user as having a distribution of ratings for all the movies that they've watched, these ratings may not be transferrable to other users. In particular, each user will have a different *mean* (average) for their ratings.

To get around this problem, we can look at other users that saw the movie rated it, *as compared to their average rating*. This approach then calculates an offset for the movie, and then applies it to the average for the user. It is loosely based on the Slope One algorithm (https://en.wikipedia.org/wiki/Slope_One). In high-level terms, this approach can be thought of as calculating the rough "likeability" for a movie and combining that offset factor with a user's base rating to make a prediction. I sometimes refer to this "likeability" method in my code.

To test this idea out, I implemented 7-fold cross-validation and calculated the RMSE scores on the training data. The result was an RMSE of **0.8615**, which is good for such a basic method. This is implemented in the **`predictBasic()`** method of my model class.

Next, I copied the logic and applied it to the test data to create an output file. This is implemented in **`createOutputBasic()`** method of my model class. This produced my first Miner submission, and yield an official RMSE value of **0.86**, as expected.

A quick analysis of this method yields a number of advantages:

- The method is based on averages, so it is very resistant to degradation, based on a bad split of the data during cross-validation. In all cases when I used this method across the project, the RMSE score was always between 0.852 and 0.863.
- It is fast! The steps are:
  - Reading the data: $O(n)$.
  - Computing the averages: $O(n)$, because we are only aggregating each rating twice, and hash maps have $O(1)$ access
  - Compute the predictions: $O(1)$ for each record because of constant time lookups, and $O(n)$ combined. This is much faster than an approach like KNN, which can suffer from slow prediction times.

  Therefore, this is approximately linear performance. On my computer, reading the data, cross validating the data (computing new averages for each fold), and making predictions for all 641699 records in the training set takes about 5-10 seconds.

3. Implement Multi-Estimate Approach using Supplemental Data

The next step that I took for the project was to implement more specific estimates for the user, based on the supplemental data and specific to the movie we need to estimate for. The additional data I pulled into the RatingPredictionModel was:

- Genres - **`readGenreInfo()`**
- Director - **`readDirectorInfo()`**
- Actors - **`readActorInfo()`**

The next step that I took for the project was to implement more specific estimates for the user, based on the supplemental data and specific to the movie we need to estimate for.

For these estimates, I looked at other movies the user watched with a similar genre, director, or cast. Then I used the ratings the user gave to these other movies to make a prediction for the current film. Note that each movie may have multiple genres and cast members, so each previous movie rating is weighted according to the cosine similarity in those cases. On the other had, movies only have a single director, so the similarity score will always be 0 or 1 in that case.

A table showing the RMSE scores for each of these initial approaches that I tried and tested independent of each other is shown in the table below. Note that the estimates based on supplemental movie metadata may not give a usable rating prediction, if for example, the user has never watched other movies with the given genre, director or cast. Therefore, I report how many records out of my validation set had a usable estimate, and the RMSE score is only based on that subset.

Description	Valid Count	RMSE
Average rating of all users (Compute average per user first, and then average these values).	10000	1.0098
Average rating given to all other movies watched by this user.	10000	0.9247
Average rating of all movies (Compute average per movie first, and then average these values).	10000	1.0279
Average rating given to this movie by all other users that watched it.	10000	0.8810
"Likeability" method described in the previous section.	10000	0.8533
Average rating given by this user to other movies. Ratings are weighted by how similar the movie genre was to the genre for the movie to be predicted.	9994	0.8966
Average rating given by this user to other movies they watched with the same director as the movie to be predicted.	4643	1.0022
Average rating given by this user to other movies. Ratings are weighted by how similar the movie cast was to the cast for the movie to be predicted.	8835	0.9462

As expected, the user-specific and movie specific estimates perform better than the average-of-averages approach. In addition, we see that the "likeability" approach had the best performance, the estimate based on movie genre is decent, and the estimates based on the director and cast are not as useful.

Note that I only pulled in the top 10 actors for the cast estimate. Grabbing more did not seem useful.

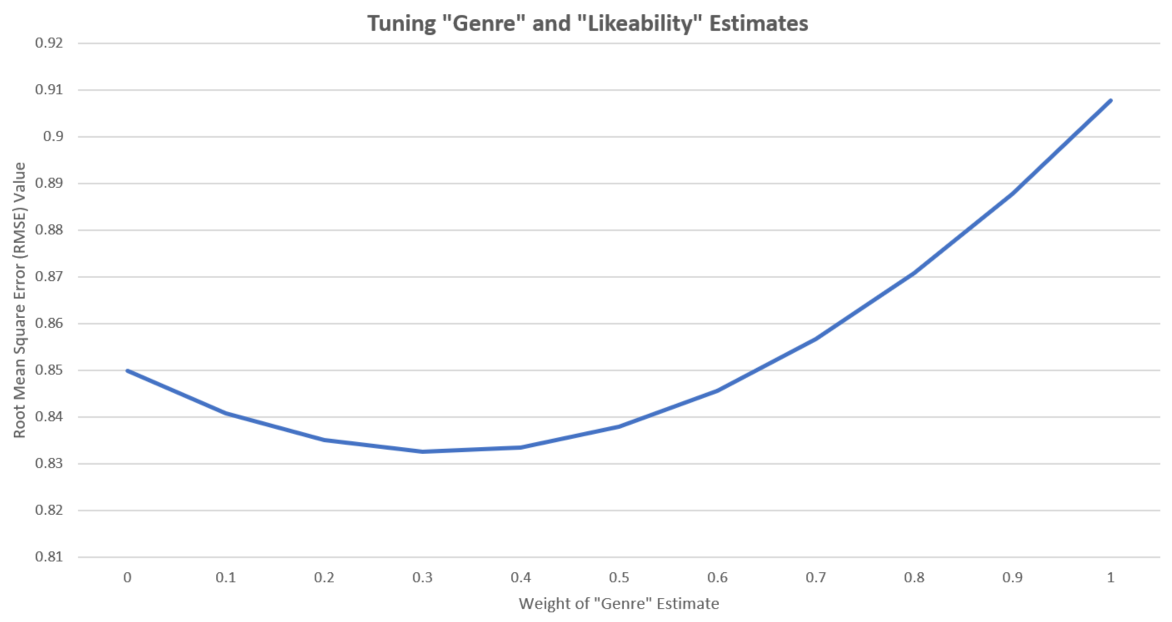
During this phase of the project, I also took the time to perform two key code updates:

- I combined my cross-validation and test output logic so that I could guarantee I was calling the same code in both the training and submission scenarios. The **`predictMultiEstimate()`** implements the high-level flow of control for both of the scenarios and accepts arguments to specify the desired behavior. In both of these scenarios, predictions are generated by the **`getMultiEstimatePredictions()`** method.
- With the enhancements to compute estimates based on supplemental movie data, the method to calculate predictions runs somewhat more slowly. However, once these estimates (like the ones shown in the table) are generated, we can easily combine them in any combination we like, with almost no penalty. Therefore, I added the ability to pass in a list of arbitrary weight configurations (modeled by the **ModelWeightConfig** class) and a list with the estimate for each will be computed. This has the additional benefit that we can think of each additional configuration as a "fall back", in the case that we would like to primarily use combination of more advanced metrics that may not be applicable for every test record. When generating submission files, I added these "back up" configurations by

including the list returned by the **`getBackupConfigs()`** method in the **RunRecommendationSystem.java** class.

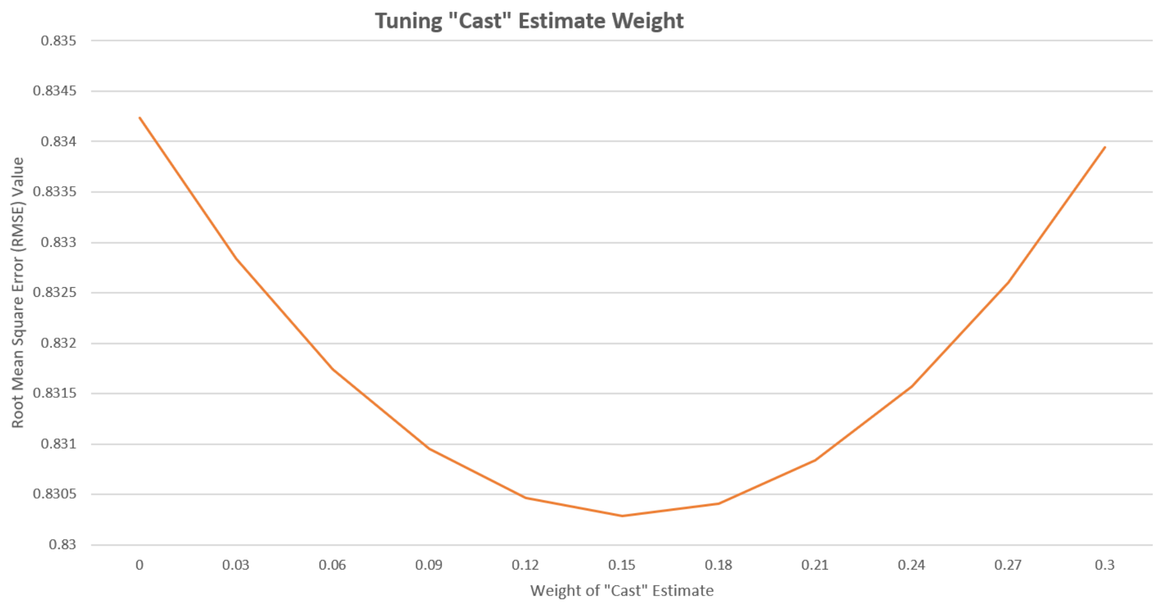
4. Tune Weighted Multi-Estimate Approach

Based on my results from the previous section, the “likeability” rating method has the best RMSE value. However, given that the “genre” approach is also fairly decent, I decided to try to combine these two estimators, with different weights to see if I could generate improved predictions. The graph of my experiment, showing the RMSE score, as compared to the weight of the “genre” estimate $\left(w_{genre}\right)$ is shown below. The weight of the “likeability” estimate $\left(w_{likeability}\right)$ can be found according to: $w_{likeability} = 1 - w_{genre}$



I was obviously pleased to find that a combination of these 2 estimators produced better results than either of them individually. The best combination is: $w_{likeability} = 0.7$, and $w_{genre} = 0.3$, which lowers our best RMSE value to **0.8326**. This makes sense, given that “likeability” has a better individual RMSE score, so it has a higher weight. Note that values on the end points of the above curve match the values for the individual estimators.

The next estimator that I decided to include was for the movie “cast” predictor. My approach was to keep the “genre” and “likeability” estimates in the same 0.3 : 0.7 ratio, but then add in the “cast” estimate, and given that it is worse estimator than either of the above estimators, then the weight, w_{cast} should be less than 0.3, so only test combinations in that range. My results are below:



Again, we get a nice curve, showing that the best combination is:

$$w_{likeability} = 0.595, w_{genre} = 0.255, w_{cast} = 0.15$$

And our RMSE score has again dropped to **0.8303**. I used this combination of weights to form the configuration for my second Miner submission, yielding an official score of **0.83** on the site.

5. Estimate Rating based on Weighted User Similarity and Common Movie Ratings

At this point in the project I felt that I had squeezed all of the possible performance out of my existing set of estimators. Therefore, I used a different approach to design my last estimator, which I refer to as the "common watch rating".

The idea is:

1. When we are given a user/movie pair, to predict, find the list of other users that has watched that movie. In other words, for each of these other users, we can find their rating for the movie in the training set.
2. Filter this list of users based on users that have at least one *other* movie in common with our user to predict a rating for.
3. For each of these common movies between our main user, and the other user, use their rating for the common movie from the training set and compute the offset from that user's average movie rating (for all movies they've watched).
4. Next, for each movie subtract the offset for our other user from the offset for our other users. Take the absolute value and sum these values for all movies that the 2 users have in common. Divide by the count to get the average difference in offset.
    a. This step uses the offset between the rating and the average rating as a proxy for how much a user liked the movie. If both users had the same reaction to the movie (better or not as good as the average), their offsets will be similar, and their average offsets will be similar.
5. For each other user, normalize their average offset from our main user, through:

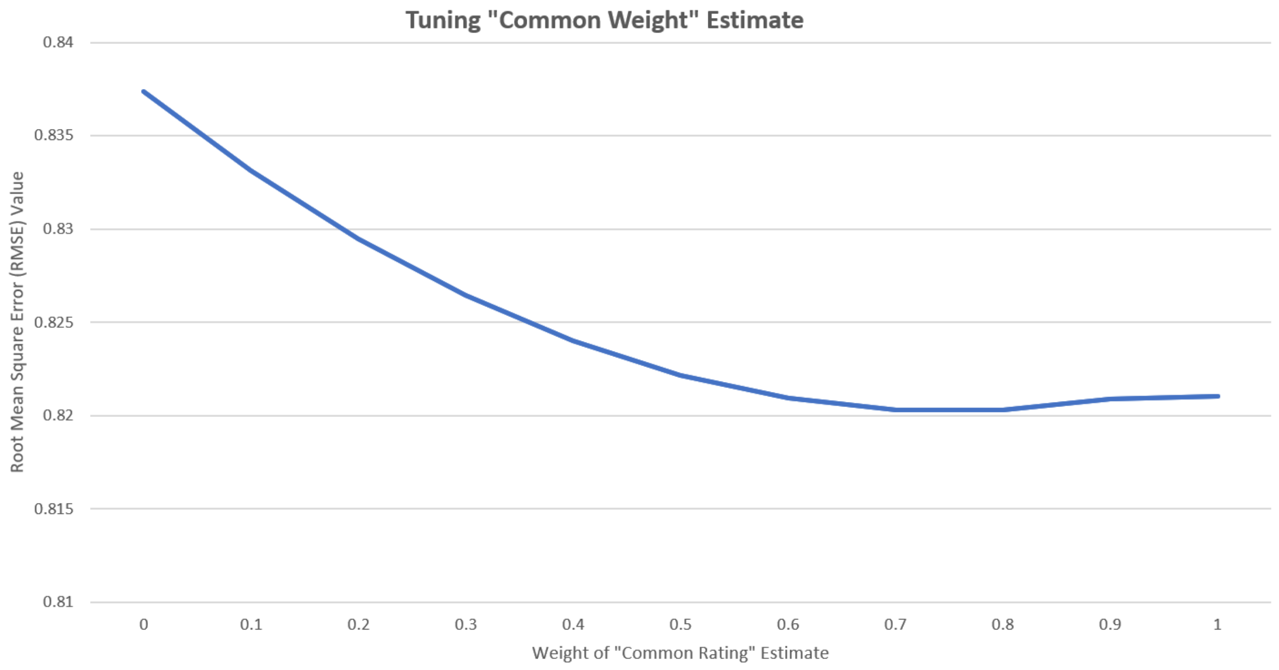
$$Normalized = \frac{offset_{user} - offset_{min}}{offset_{max} - offset_{min}}$$

This normalized value is in the $[0, 1]$ range. The users that rated common most similarly to our main user will have an offset that is low.

6. Compute the final weight for each common value using:
$$Weight = (1 - normalized)^2$$
7. This gives a weight preference to the similar users.
8. Compute the final prediction using the user ratings for the movie to be predicted, computing an offset compared to their average rating, and using this to generate an offset that is weighted by the result of the previous step and added to our main users average rating.

After implementing this algorithm and testing it via cross validation, it's RMSE was 0.8156, and it gave a valid rating for 997/1000 users in the sample.

Therefore, I added it to my config in the same manner as I had the other weights. My graph showing the fraction of weight for the "common rating" estimator is below.



As before, I kept the weight ratio of the other three estimators ("likeability", "genre", "cast") the same. This minimum value for this curve has an RMSE value of **0.82031**. The weights are:

$$w_{common-rating} = 0.8, w_{likeability} = 0.119, w_{genre} = 0.051, w_{cast} = 0.03$$

The final step for this project was to create a submission file based on this configuration. This became my third and final submission with an RMSE value of **0.81**.

6. Additional Discussion and Improvements

Some brief notes about the project:

- My cross-validation scores ended up being very close to my Miner scores, so I could do a lot of testing without submitting every time.
- Incorporating tags did not seemed useful because they are subjective (based on a user), not particularly dense (many lightly used tags), and I did not want to pull in NLP logic to get additional value from them.
- When using **HashMap** in Java, it is critical to implement hashCode() and equals() correctly, which I did, using a generated ID. This prevented me from running into any strange processing issues related to these collections.
- Some refactoring of the code would make sense if I had extra time (utility function, I/O classes, etc.)

5. Algorithm Efficiency and Runtime

Although we saw that the basic algorithm is quite efficient, the processing time for the last estimator is significant, and it took me about a half hour to generate the output file. If the average number of movies watched by a user is j, and the number of users watching each movie is k, to predict n ratings, the runtime will be $O(njk)$, and for a large sample size like our training and test sets, this will be very slow.