Book Recommendation

Kate Kitsakos, Emma Ratigan, Kaining Shen, Tion Wong, Timothy Yang

Abstract

The primary goal of this project was to create a recommender system that takes the name of a book the user enjoyed as its input and returns the title of a different book that the user will likely also enjoy. A recommender system is an information filtering program that is used to predict how a user would score an item or service they have not yet seen. In terms of this project, a predictive model was created that uses the book entered by the user to forecast whether a user would rate other books in the system highly (a rating of 4 or 5) or poorly (a rating of 1, 2, or 3). The model then returns the title with the highest predicted rating. When making predictions, the recommender system draws from a data set containing the ratings of 10,000 books by 53,424 distinct users. Challenges faced during system construction included determining the best data mining technique to use when building the model, time constraints due to a large data set and unoptimized code, and limitations when determining the accuracy. The highest accuracy achieved by the recommendation system was 77.67% using the optimized Link Rating Classifier.

Introduction

Recommender systems have their origins in the early 1990s. The first recommender system, Tapestry, was introduced in 1992.[1] Tapestry was designed to recommend content from various news organizations. It was built around the idea of using social collaboration to sort through large numbers of documents.[2] This was the first time that the term "collaborative filtering" was introduced as a way to describe using feedback from multiple users to make predictions.[2] Over the last two decades, recommender systems have become integral parts of many systems. They are used by major companies such as Netflix, Google, Twitter, Amazon, and LinkedIn to connect users with products, services, and other users that may be of interest to them.[1] Recommender systems are so prevalent because they provide personalized results based on the user's preferences or characteristics.[3] This targeted nature is highly valuable to companies that are looking to efficiently market their products, such as Amazon, or create personalized user experiences, such as Netflix. Recommender systems can be built based on several different kinds of algorithms. Algorithms that are content-based recommend products or services that are similar to ones that a user has previously liked.[4] Clustering algorithms recommend things that go well together regardless of the user's preferences. Products and services can also be recommended to a user based on the feedback from many other users through collaborative filtering algorithms.[4] Collaborative filtering algorithms can be broken down even further to item-centric or user-centric and model-based or memory-based. Item-centric collaborative filtering looks to find similarities between items in order to determine which products or services to recommend to a user. User-centric algorithms focus on finding similarities between users and then recommends items to the target user based on what those similar users enjoyed.[5] Another distinction in collaborative filtering algorithms can be made between algorithms that are based on memory and algorithms that are based on models. Memory-based collaborative filtering uses all of the data to make every prediction.[5] These algorithms are simple to implement, easy to update, and generally result in quality predictions, but because they require the entire data set to make predictions, they need significant memory and are very slow. Model-based algorithms, on the other hand, use the data to train a model and then use the model to make predictions.[5] These algorithms are generally smaller in size than the data set and can therefore be faster to implement than memory-based filtering. They do have their disadvantages, however, as it is more difficult to update the model to include new data after it has been trained and the quality of predictions can vary based on the kind of model that is selected.[5]

Data Overview

The recommender system was built using a Github sourced Goodreads data file that describes 10,000 books and users' ratings. The file includes two excel sheets, books.csv and ratings.csv. The books.csv sheet contains 10,000 rows of data with each row corresponding to a unique book. It has 23 attributes, including book id, book title, rating count and average rating. The average rating of the 10,000 books is 4.002 out of 5. There is an average of 54,001 ratings per book with a median of 21,155 ratings per book. Having a mean that is significantly larger than the median suggests that there are books at the top of the list that were rated significantly more times than other books in the data set.

The second excel sheet (ratings.csv) contains user ratings for the books in books.csv. The ratings sheet consists of 3 attributes and has 5,976,479 rows. Each row corresponds to a user id, a book id and a rating. The number of unique user ids recorded in this table is 53,424 and each user rated an average of over 100 books. The possible ratings values are 1, 2, 3, 4, and 5 with 1 being the lowest and 5 being the highest.

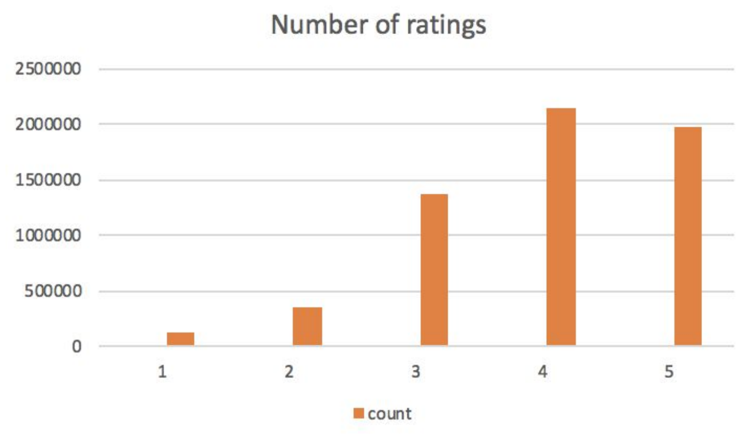


Figure 1: The number of instances for each rating value

As shown in Figure 1, almost 70% of the total number of ratings is a 4 or 5 rating. This is likely due to the fact that the books used in this data set are mostly commonly rated books on Goodreads, such as the *The Hunger Games* series and the *Harry Potter* series. From the ratings.csv, the average rating for all records is 3.92. This is roughly the same as the average rating from books.csv. The books.csv file sums to over 500 million ratings, so the ratings contained in the ratings.csv file is roughly 1% of that. Because the average ratings of the two files were so close, we were comfortable assessing that the ratings.csv file was representative of the larger data trends and using it to build our recommender system.

Methods

Item-centric, memory-based collaborative filtering was the primary method used to generate ratings for our book recommender system. Memory-based collaborative filtering is more applicable to the recommender system in this context because (1) newly inputted data is easier to incorporate and (2) the given data sets lack information required for model-based collaborative filtering. Training a model would have been difficult to implement in this system because it requires details such as book genres, themes, frequently used words, and other attributes that are not readily available in the data set. Model-based collaborative filtering also has a tendency to be overspecialized, meaning the recommender system would have solely look at books with similar attributes, resulting in a lack of variety in the outputs. It is not uncommon for users to express interest in a diverse collection of literature, which furthers memory-based collaborative modeling as the optimal method.

Similarly, item-centric collaborative filtering was favored over user-centric collaborative filtering due to the lack of available data attributes. User-based collaborative filtering requires information about the users, such as demographic and book genre preferences, that was not included in the dataset. This meant that the program had to rely on the book attributes to make predictions. The system works by prompting the user to enter a single book that they have previously read and enjoyed and then outputs a different book that they will likely enjoy reading. After the initial book is entered into the program, it is linked to other books that have been rated by other users who also enjoyed the input book. The program output is the most highly rated of these linked books.

This is item-centric filtering. The program looks for similarly rated books instead of users with similar demographics or interests. Although the use of item-based filtering was necessary due to the lack of data regarding the users, it also had the added benefit of allowing the user to select the book they wanted their recommendation to be based off of. This creates an additional layer of personalization within our system so that users can get recommendations for different kinds of books depending on what they feel like reading at the moment.

Classification, regression, and clustering failed to operate as functional predictive models for the book recommender. Regression was immediately ruled out because the given data cannot be evaluated to fit in a numerical or continuous model. Clustering is insufficient because there are no clear, established correlations that can be grouped together before evaluation. Classification most lined up with the given task, for the recommender is predicting whether or not a user would enjoy a given book, however, the ratings are not objective which makes classification difficult. Therefore, collaborative filtering was established as the best predictive model for the book recommender.

The original data set contained 10,000 books and 5,976,479 ratings. For the creation of the recommender system, the data set was limited to the 100 books with the most ratings. This was done because the original data set took too long to run code on due to its large size. This means that the recommender system is only concerned with calculating the link ratings between these 100 books and that the input and output values for the system has to be in this subset. The most popular books were selected instead of choosing random titles because we wanted to try to maximize the number of common users between the books. Common users were more likely to appear in books that had been rated frequently than between books that had only been rated occasionally.

Every pair of books within the training set was given a similarity value. The value, termed a “link rating”, was calculated based on user ratings within the dataset. The calculation started by identifying all of the users that rated the first book a 4 or a 5. Of that group of users, all of the people that also rated the second book were then identified. The average rating for the second book was calculated from this subset of users. This result is the link rating from the first book to the second book. It is a measure of how users who enjoyed the first book viewed the second book; in other words, it is an indirect measure of the similarity between the two books. It should be noted that the link rating values are directional. The value from the first book to the second book may not be the same as the value from the second book to the first book.

The link rating calculation required the creation of a threshold value. The threshold was included in order to prevent books that had high link ratings but few common users rating them from being recommended over books with lower link ratings but a large number of common users. The concern was that link ratings stemming from a smaller number of users would not be as accurate predictors of whether a book will be enjoyed as link ratings based on a large set of common users. For example, if Book A had an average rating of 5.0 based on 2 ratings and Book B had an average rating of 4.5 based on 50 ratings, the average rating of Book B is much more likely to be indicative of how a user will rate it since it is less susceptible to atypical opinions (outliers). The threshold helped address this concern by setting a lower limit to the number of common users a pair of books had to share in order to be considered for recommendation. The threshold value was set at the beginning of the system construction before the link values were calculated. After the link values had been computed, all of the links that contained fewer common users than the threshold were removed from consideration for recommendation. The link value calculation and threshold pruning was performed on the entire training set before any user inputs were allowed.

On very large data sets, tests are generally invulnerable to noise and outliers. Because of this, overfitting and overly optimistic results were unlikely. Regardless, precautions, such as using independent test sets and performing k-fold cross validations, were taken. Training and testing data were kept separate to ensure that the model had not previously encountered the data that the results were based upon. When the recommender system was built, the test set was separated from the training set prior to system construction. This restricted the results from becoming overly optimistic. The size of the test and training sets varied as we evaluated several different data splits including 50:50, 75:25, and 90:10 training set to test set ratios. K-fold cross validation with 2, 4, and 10 folds were used to protect against overfitting. By comparing the performance between the different values of k, it was evident that there were no significant differences between 2, 4, or 10 folds, indicating that the large size of the data set meant that there was a sufficient amount of data to train on, even when using 2-fold cross validation, to avoid overfitting.

The code itself was entirely done in Python, using the csv, time, and random libraries. The majority of the code consisted of 3 main tasks: preprocessing data, recommender system, and k-fold cross validation. To begin, we processed the files "ratings.txt" (from ratings.csv) and "nameToID.txt" (from books.csv), and output 3 arrays:

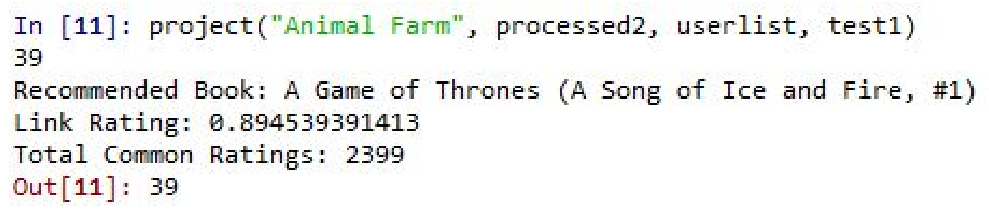
1. A list containing all the user_ids, in order ascending order
2. A list of lists, with each subarray containing all the user_ids that rated the book_id corresponding to the index >=4
3. A list of lists, with each subarray containing every book_id that a user rated and its corresponding rating

Using these 3 arrays, we were able to build the recommender system, which worked like this:

(keep in mind that booklist is an initially empty array containing, with entries appended to it for every new book in this format: booklist=[([book_id, ratings, high ratings]), ([...]) , …])

```
Convert input string to valid book_id                                   ##uses nameToID.txt
For every user_id that rated the input book >=4:                        ##uses list 2
    For every book_id that this user_id rated:                          ##uses list 1 and list 3
        If this book_id is in booklist:
            common ratings+=1 for this book_id
            If this user_id rated this book_id >=4:
                high ratings+=1 for this book_id
        Else:
            If this user_id rated this book_id >=4:
                booklist.append([book_id, 1, 1])
            Else:
                booklist.append([book_id, 1, 0])

Using booklist, return the book_id with the highest link rating score (high ratings/common
ratings) that has common ratings>threshold
```

The recommender system part of our code enabled us to input a string containing the name of any book, and return a valid book_id or book name representing the recommended book. We later came to realize that performing this code on our large dataset was very inefficient, especially as the size of the "training" data increased (thereby increasing the size of list 1 and list 3). There were also some special cases which we had to take into account, including the case where no book had a common rating score which exceeded the threshold or the case where there were no entries in booklist, meaning that there are no common ratings at all. To ensure that the performance of our k-fold cross validation would not be affected for both of these rare cases, we would return recommended **book_id** = 0, which is not a valid book_id.



```
In [11]: project("Animal Farm", processed2, userlist, test1)
39
Recommended Book: A Game of Thrones (A Song of Ice and Fire, #1)
Link Rating: 0.894539391413
Total Common Ratings: 2399
Out[11]: 39
```

Figure 2: A example of one recommended book

Finally, in order to evaluate the performance of our recommender system, we performed k-fold cross validation tests. First, we shuffled the data set to ensure that data was evenly distributed. For each fold, we created a new array mapping every book to its recommended book using the “training” portion of our data to create the 3 arrays. After creating the mapping, we went through every single user in the test set, and every single book that that user rated. Performance was evaluated using the following guidelines:

If this user rated this book >=4 and rated the recommended book >=4:
- True Positives +=1

If this user rated this book >=4 and rated the recommended book <4:
- False Positives +=1

If this user rated this book <4 and rated the recommended book >=4:
- False Negatives +=1

If this user rated this book <4 and rated the recommended book <4:
- True Negatives +=1

If the user did not rate the recommended book:
- Nothing happens

The accuracy for this fold is:

$$Accuracy = \frac{True\ Positives + True\ Negatives}{True\ Positives + False\ Positives + False\ Negatives + True\ Negatives}$$

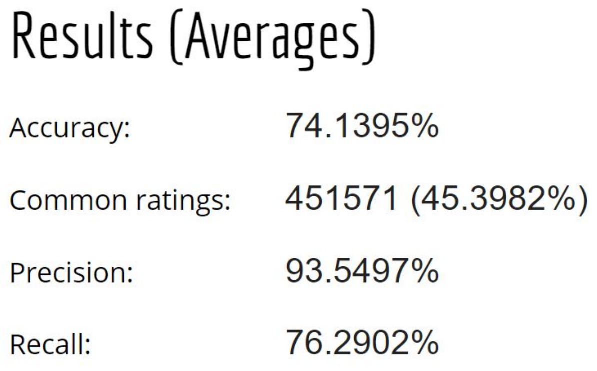
By repeating this process using different splits of the initial dataset (which creates different mappings), obtained an accuracy for each fold, and the final accuracy was the average among all folds.

We called the total number of recommended books that the user also rated the *Common Ratings*, (*True Positives+False Positives+False Negatives+True Negatives)*. Because we don't take into account that a given user did not rate the recommended book at all, we also included the *Common Ratings Percentage* metric, $\frac{Common\ Ratings}{Total\ Ratings}$, which indicated the percentage of recommended books that the user also rated. In general, a user is more likely to read and give a rating to a book that they liked rather than a book that they disliked, and therefore we thought that this would be appropriate as another evaluator of performance. We also kept track of *Precision*, $\frac{True\ Positives}{True\ Positives + False\ Positives}$, which is a measure of the proportion of books our model recommended that were actually highly rated, and *Recall*, $\frac{True\ Positives}{True\ Positives + False\ Negatives}$, which is a measure of the ability to identify all the users that actually enjoyed the input book if they enjoyed the recommended book.

Our recommendation system is based on positive reinforcement. This means that our system was coded with the assumption that the user enjoys the input book, and outputs a recommended book that we expect the user to also enjoy, as opposed to taking in a book that the user did not enjoy and outputting a book that the user should stay away from. As a result, we expect our number of true positives and false negatives to be high. We suspect that accuracy, precision*,* and the common ratings percentage would be the best metrics to evaluate the performance. Accuracy gives a comprehensive performance for our data, as it takes into account all four metrics: true positive, true negative, false positive, and false negative. Precision gives a better evaluation of the performance of our recommendation system, since we based it on positive reinforcement. The common ratings percentage tells us the likelihood that a user has also rated any given recommended book.

Results

As previously noted, there is a threshold for the minimum number of common ratings a book needs to be considered for recommendation. We tested 3 arbitrary values (50, 100, 150), to see if these thresholds had any significant impact on the accuracy of the recommender system. On each threshold value we ran 2-fold, 4-fold, and 10-fold cross validation tests. The overall results are summarized in Figure 3 below.



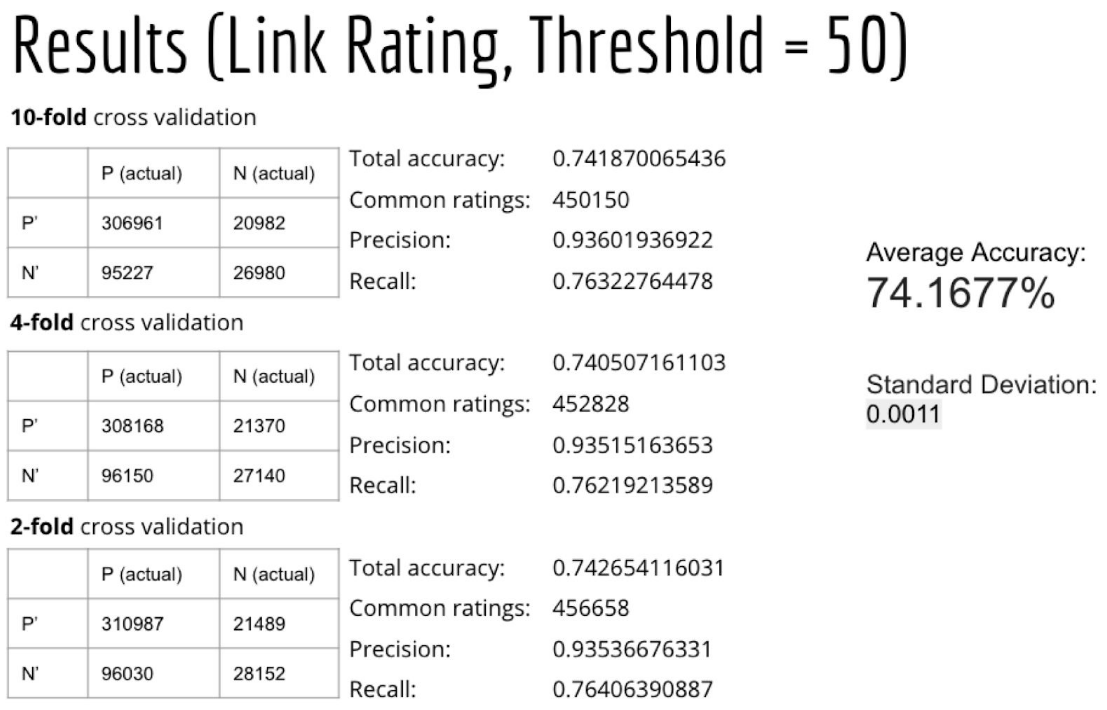
Results (Averages)

Accuracy:	74.1395%
Common ratings:	451571 (45.3982%)
Precision:	93.5497%
Recall:	76.2902%

Figure 3: The overall results for 2-fold, 4-fold, and 10-fold cross validation

Figure 3 displays the averages of the results for the three cross validation tests on the three threshold values. Although these averages incorporated different threshold values and cross validation tests, the results from each individual analysis were similar enough that the overall averages still provide a complete picture of the recommender system results. The standard deviation of the overall average accuracy was 0.02923 which illustrates the results' close grouping.

The breakdown of the cross validation results for a threshold of 50 are given below in Figure 4:



Results (Link Rating, Threshold = 50)

10-fold cross validation

	P (actual)	N (actual)
P'	306961	20982
N'	95227	26980

Total accuracy:	0.741870065436
Common ratings:	450150
Precision:	0.93601936922
Recall:	0.76322764478

4-fold cross validation

	P (actual)	N (actual)
P'	308168	21370
N'	96150	27140

Total accuracy:	0.740507161103
Common ratings:	452828
Precision:	0.93515163653
Recall:	0.76219213589

2-fold cross validation

	P (actual)	N (actual)
P'	310987	21489
N'	96030	28152

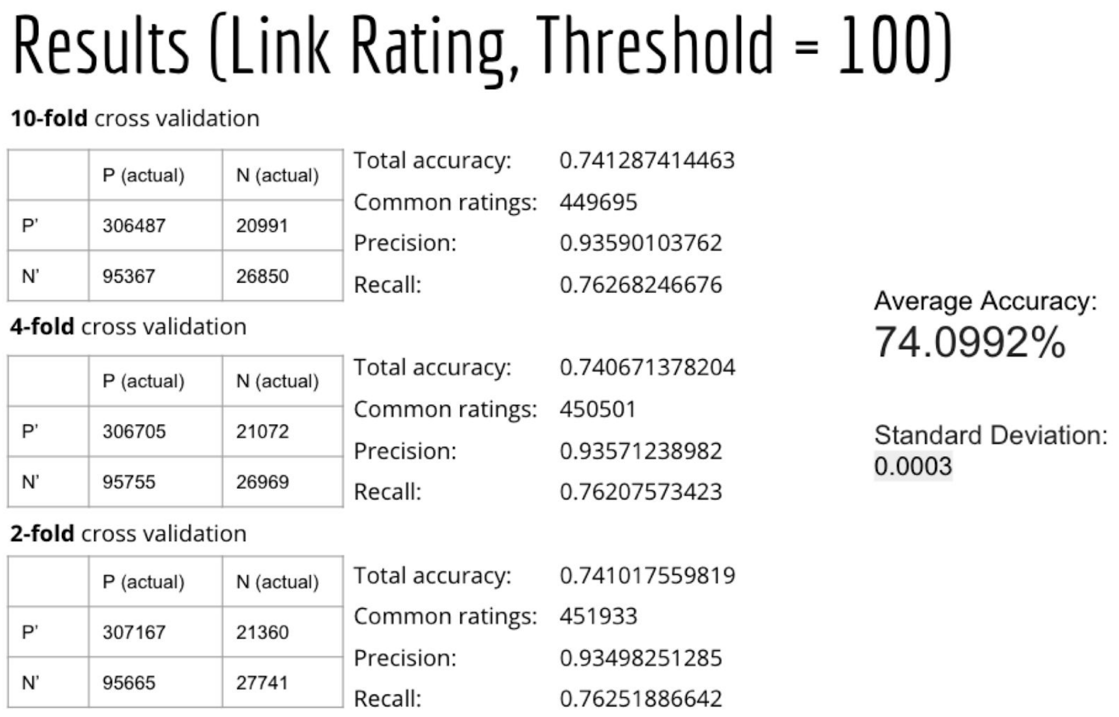
Total accuracy:	0.742654116031
Common ratings:	456658
Precision:	0.93536676331
Recall:	0.76406390887

Average Accuracy:
74.1677%

Standard Deviation:
0.0011

Figure 4: The results for each cross validation with threshold = 50

The average accuracy was about 0.7417. This percentage did not change drastically as the number of folds increased. The number of false negatives (users rating the recommended book highly, but rated the inputted book poorly) is 4 times the number of false positives. This could potentially mean that the recommendation system was not working, but given the high number of true positives (user rated both the input and output books highly), this does not seem likely. After noting a similar pattern in the results for the 100 and 150 thresholds (Figure 5 and Figure 6) it was hypothesized that the recommender system was favoring books with high average ratings.



Results (Link Rating, Threshold = 100)

10-fold cross validation

	P (actual)	N (actual)
P'	306487	20991
N'	95367	26850

Total accuracy:	0.741287414463
Common ratings:	449695
Precision:	0.93590103762
Recall:	0.76268246676

4-fold cross validation

	P (actual)	N (actual)
P'	306705	21072
N'	95755	26969

Total accuracy:	0.740671378204
Common ratings:	450501
Precision:	0.93571238982
Recall:	0.76207573423

2-fold cross validation

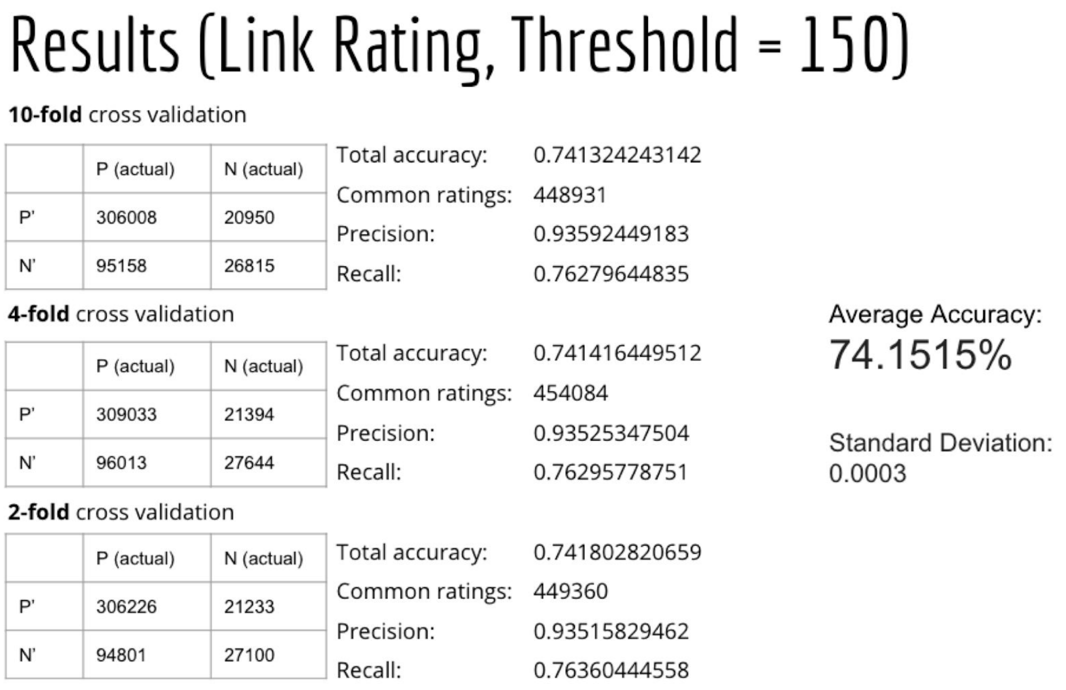
	P (actual)	N (actual)
P'	307167	21360
N'	95665	27741

Total accuracy:	0.741017559819
Common ratings:	451933
Precision:	0.93498251285
Recall:	0.76251886642

Average Accuracy:
74.0992%

Standard Deviation:
0.0003

Figure 5: The results for each cross validation with threshold = 100



Results (Link Rating, Threshold = 150)

10-fold cross validation

	P (actual)	N (actual)
P'	306008	20950
N'	95158	26815

Total accuracy:	0.741324243142
Common ratings:	448931
Precision:	0.93592449183
Recall:	0.76279644835

4-fold cross validation

	P (actual)	N (actual)
P'	309033	21394
N'	96013	27644

Total accuracy:	0.741416449512
Common ratings:	454084
Precision:	0.93525347504
Recall:	0.76295778751

2-fold cross validation

	P (actual)	N (actual)
P'	306226	21233
N'	94801	27100

Total accuracy:	0.741802820659
Common ratings:	449360
Precision:	0.93515829462
Recall:	0.76360444558

Average Accuracy:
74.1515%

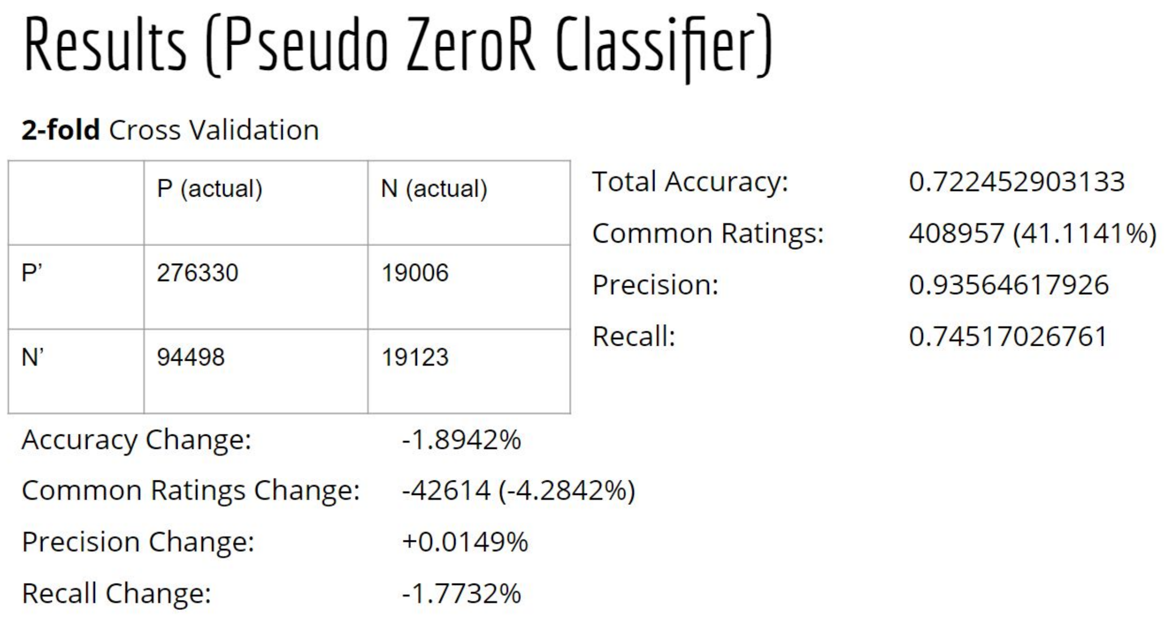
Standard Deviation:
0.0003

Figure 6: The results for each cross validation with threshold = 150

As with the threshold of 50, there was no significant change in either the accuracy or the number of common ratings across all 3 different cross validations for either of these threshold values (Appendices 1, 2, 3, 4). The precision was significantly higher than the accuracy and the recall, mainly because the number of true positives was high, as was predicted.

Since our system is based on commonly highly rated books, it naturally favors the books with the most ratings in general, since they typically also have a high average rating. This means that for any given book, the system will likely recommend one of the most highly-rated books, most notably *Harry Potter* and *the Deathly Hallows*, which was the highest rated book among the first 100, with an average rating of 4.61. This resulted in a high number of false negatives (user gave input book a low rating, but gave the recommended book a high rating), as indicated by the results, and thus a lower accuracy.

To test the hypothesis that our recommender system was favoring popular books with a high average rating, we compared the results of only recommending *Harry Potter and the Deathly Hallows*, the book with the highest average rating among the 100 most popular books, to those of our recommender system. The theory was tested using a modified version of a ZeroR classifier. Unsurprisingly, among the first 100 books, the most commonly recommended book was book_id = 25, *Harry Potter and the Deathly Hallows*. Rather than using the recommender system for testing, we would instead always recommend this book, regardless of what title the user input. The results of this provided some baseline statistics for which we could evaluate the performance of the various classifiers.



Results (Pseudo ZeroR Classifier)

2-fold Cross Validation

	P (actual)	N (actual)
P’	276330	19006
N’	94498	19123

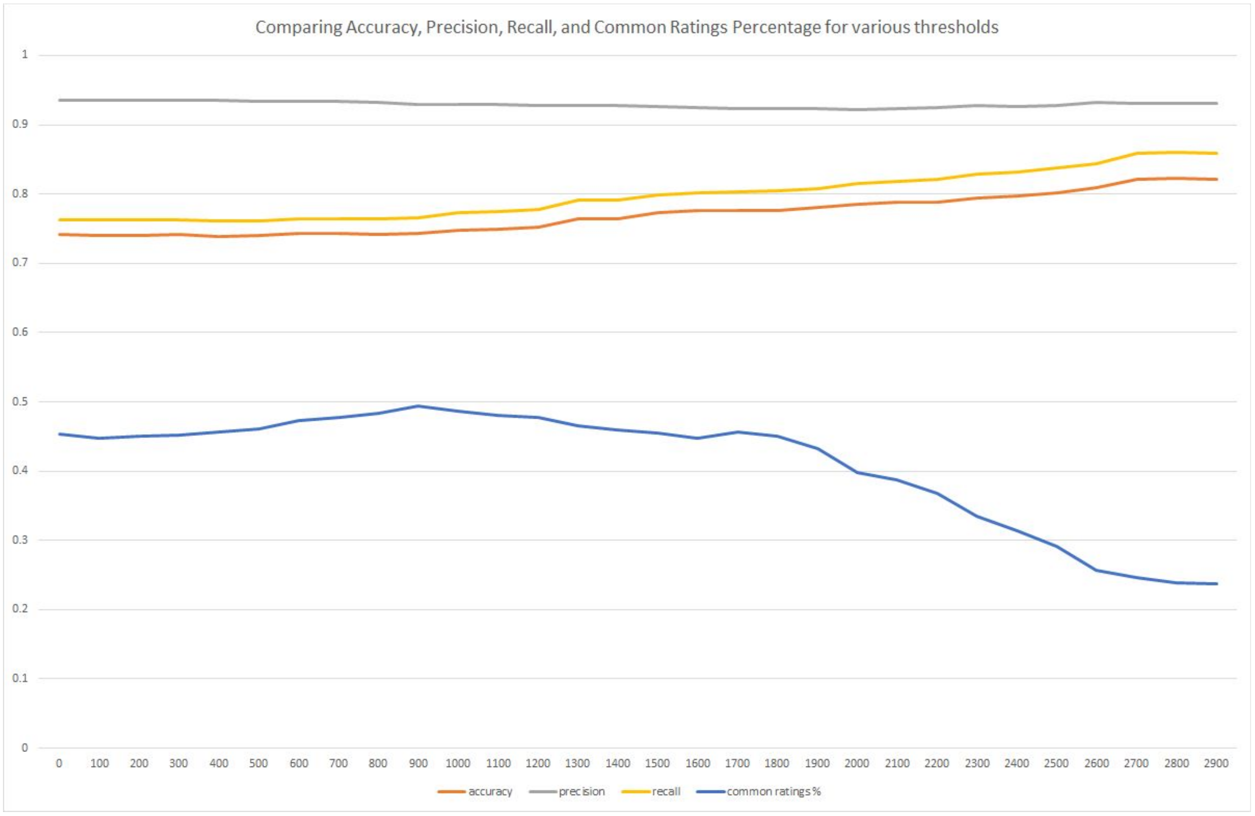
Total Accuracy:	0.722452903133
Common Ratings:	408957 (41.1141%)
Precision:	0.93564617926
Recall:	0.74517026761

Accuracy Change:	-1.8942%
Common Ratings Change:	-42614 (-4.2842%)
Precision Change:	+0.0149%
Recall Change:	-1.7732%

Figure 7: The results for Pseudo ZeroR Classifier

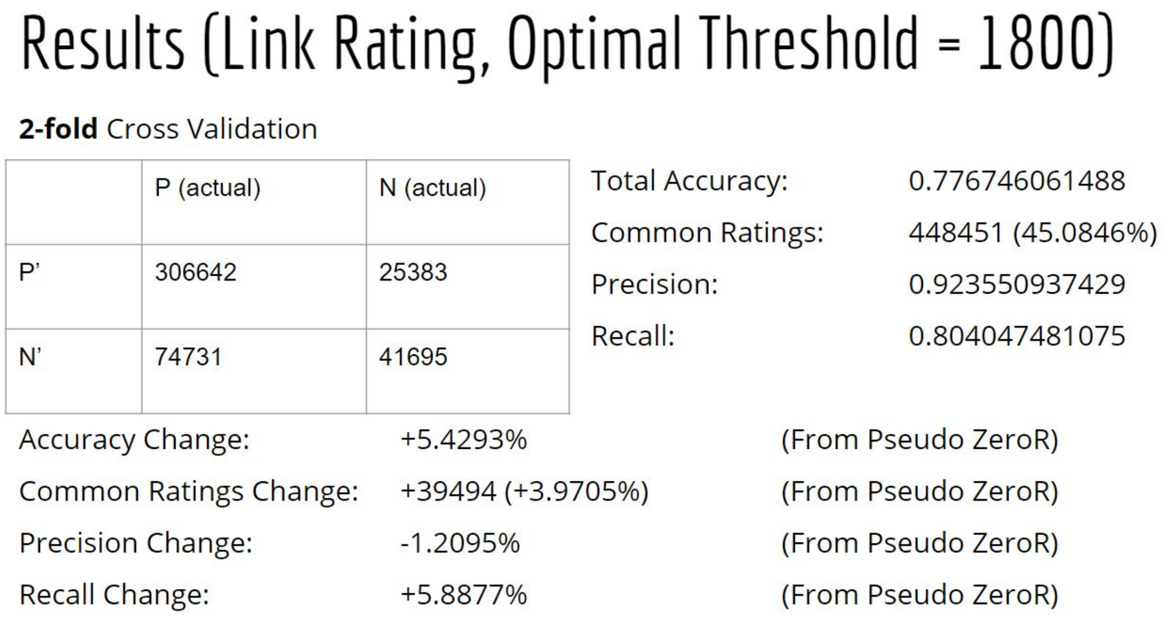
Interestingly, the performance of our Pseudo ZeroR Classifier was only slightly worse than our average, and our precision even went up very slightly. We also decided to only use 2-fold cross validation from now on, since we did not find any significant differences between evaluating our system at different folds, and the 2-fold cross validation had the lowest running time out of the three validation tests.

In the beginning, we chose the values of 50, 100, and 150 for the threshold arbitrarily. After using those values in the initial testing, we chose to optimize our system to find the best value(s) for this threshold. To do this, we performed a 2-fold cross validation on every threshold from 0 to 2900 in intervals of 100 and compared the accuracy, precision, recall, and common ratings percentage for all of them. We chose an upper bound of 2900 after running the program several times and noticing that very few books had over 2900 common ratings with another book. The results are given in Appendix 5 and charted in Figure 8.



<u>Figure 8</u>: The results for each Threshold from 0 to 2900

Interestingly, the results indicated that the higher thresholds resulted in higher accuracies, but also that the higher thresholds resulted in lower common ratings percentages. This was due to the way we coded our system: for the case that there were no books above the threshold, our system returned book_id = 0, which is not an id value associated with any book. When we evaluated our data, this meant that we would not take into account this recommendation at all, since it was impossible that a user rated a book which does not exist. As we increase our threshold, the chance that we recommend a give book with book_id = 0 increases, meaning that while we are giving highly accurate results, these results are from a very small pool, and for most of the 100 books, we are giving recommendations to a book that does not exist. This is not optimal. In order to find the "Optimal Threshold", we decided to find the threshold which gave the highest accuracy without sacrificing common ratings percentage, in other words, the highest accuracy given before common ratings percentage began decreasing below the threshold = 0 baseline. This value was determined to be 1800. The results for this threshold are summarized below.



Results (Link Rating, Optimal Threshold = 1800)

2-fold Cross Validation

	P (actual)	N (actual)
P’	306642	25383
N’	74731	41695

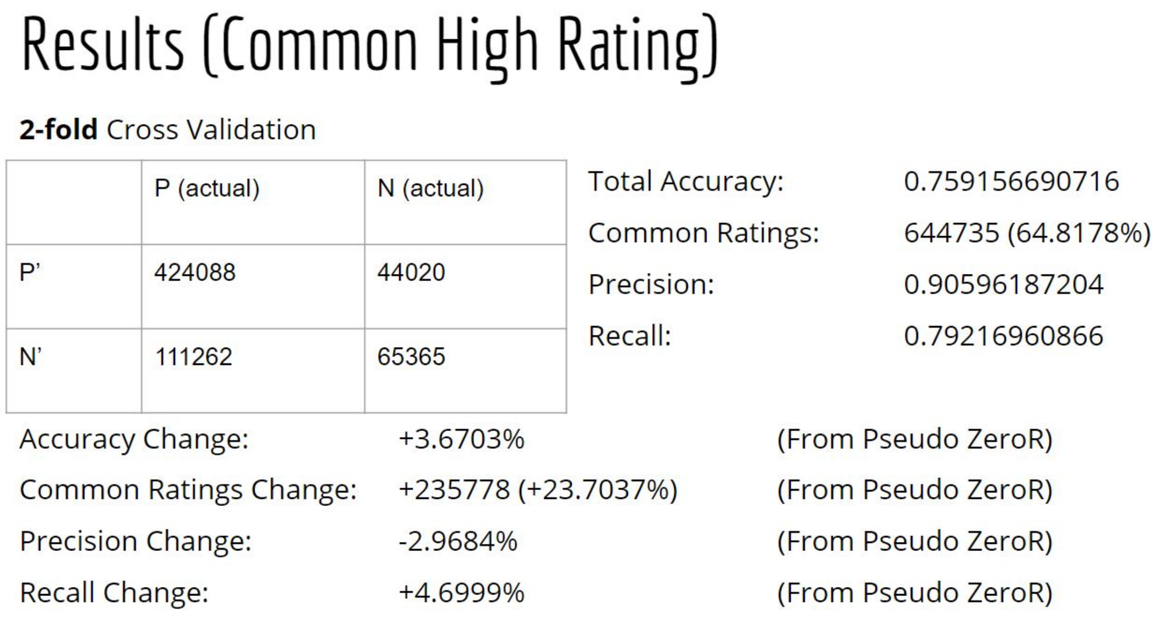
Total Accuracy:	0.776746061488
Common Ratings:	448451 (45.0846%)
Precision:	0.923550937429
Recall:	0.804047481075

Accuracy Change:	+5.4293%	(From Pseudo ZeroR)
Common Ratings Change:	+39494 (+3.9705%)	(From Pseudo ZeroR)
Precision Change:	-1.2095%	(From Pseudo ZeroR)
Recall Change:	+5.8877%	(From Pseudo ZeroR)

Figure 9: The results of the Link Rating Classifier with threshold = 1800

A threshold of 1800 gave slightly higher values in almost every single metric except precision when compared to the baseline Pseudo ZeroR. We suspect that this is because the threshold was so high that the system ignored some books with very high link ratings.

Because we noticed that higher thresholds resulted in higher accuracies, we decided to try a recommendation system that would return the book with the maximum number of high (>=4) common ratings, rather than the maximum link rating. The results of this system are given in Figure 10.



Results (Common High Rating)

2-fold Cross Validation

	P (actual)	N (actual)
P’	424088	44020
N’	111262	65365

Total Accuracy:	0.759156690716
Common Ratings:	644735 (64.8178%)
Precision:	0.90596187204
Recall:	0.79216960866

Accuracy Change:	+3.6703%	(From Pseudo ZeroR)
Common Ratings Change:	+235778 (+23.7037%)	(From Pseudo ZeroR)
Precision Change:	-2.9684%	(From Pseudo ZeroR)
Recall Change:	+4.6999%	(From Pseudo ZeroR)

Figure 10: The result of maximum number of high(>=4) common ratings

This change to the recommender system output surprisingly good results. Compared to the ZeroR, the output included a much better accuracy, common ratings percentage, and recall rate, while only giving a slightly worse precision rate. This is because we are maximizing the common high rating and therefore ignoring books with a high percentage of common high ratings but low numbers of ratings in general. Compared to the Optimal Link Rating, the output included slightly lower values of accuracy, recall rate, and precision rate, but a much higher common ratings percentage. This was also to be expected, as this classifier is highly biased towards the books with the highest number of ratings. This classifier shows the advantage of our Link Rating classifier, as it is less biased towards the most popular popular books, maximizing the link rating rather than the number of high common ratings.

Comparison of the performances of various classifiers across several metrics

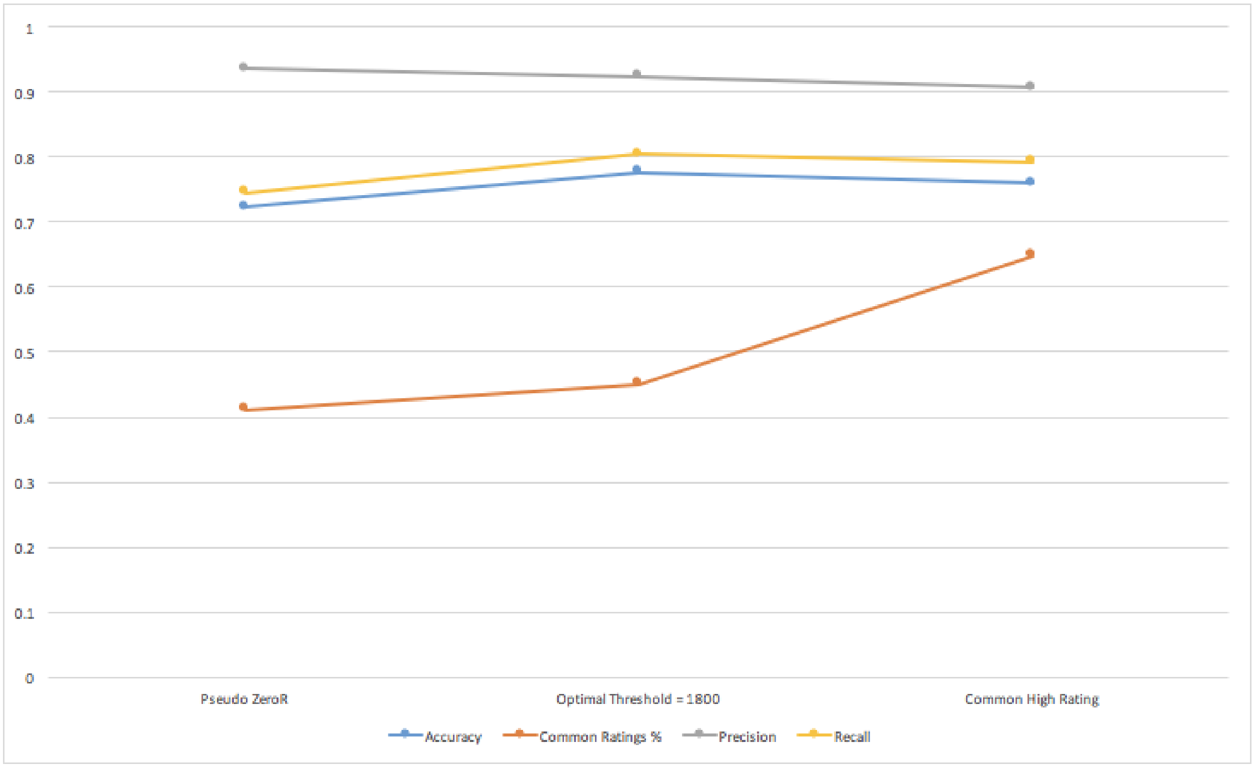


Figure 11: The comparison of the performances of different classifiers across several metrics

Examining the final results, we observe that although our baseline classifier, the Pseudo ZeroR, generally performed the worst, it still had the highest precision with about 93.5%. This was logical because our Pseudo ZeroR always recommended the highest rated book, meaning that it would give a large number of true positives and a small number of false positives. The best performing classifier in terms of accuracy was the Optimized Link Rating, with an accuracy of 77.6746%. However, it is also useful to note that the Common High Rating classifier performed much better than all others in terms of the common rating percentage, despite giving lower performance values in other metrics.

Conclusion

The highest accuracy obtained among the recommender systems was 77.67%, given by the optimal Link Rating Classifier at a threshold of 1800. We expected the recommender system to be more biased towards popular and highly rated books. We did not expect, however, that this bias would be by such a large margin that classifying every book using that most frequently recommended book would give only a slightly worse result.

There were limitations of both the dataset and the inefficiency of the recommender system. The data set was very skewed – the most popular books had significantly more ratings and higher averages than the other books. Using only the first 100 books included close to a million ratings, which took a long time for the program to preprocess the data and calculate the link ratings. This resulted in a shorter amount of time to be spent testing and improving our code.

We were able to optimize the code to some extent, improving the run time from several minutes to several seconds. However, the final version of the code we had still took around 90 minutes to do a 10-fold cross validation over the 1 million ratings. Due to this inefficiency, we could only run our training and testing on the first 100 books out of the entire 10,000 book data set, which we suspect impacted on our accuracy.

Although we were able to construct a successful recommender system that can identify a book a user will like with an accuracy of 77.67%, there were several areas of this project that could be expanded or improved upon in the future. The code could be further optimized to decrease the running time, which would allow for more data to be incorporated in the system and more tests to assess the results. The accuracy of the recommender system could likely be improved if the link rating values were normalized to account for the popularity of the book, which could decrease the number of false positives output as recommendations. Another possible improvement includes creating a link rating model that adjusts the threshold value based on the size data set, rather than testing on flat threshold values. This would allow the system to scale to different sized data sets without requiring a manual determination and input of the best threshold value.

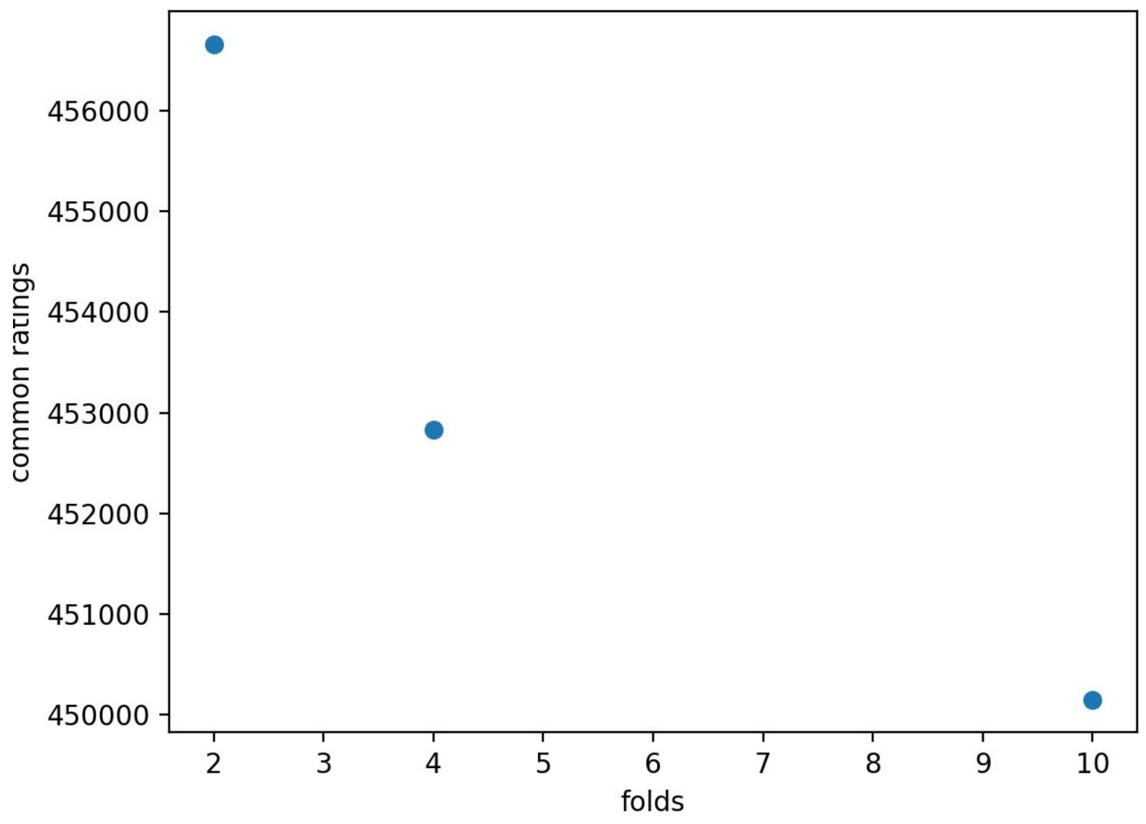
An area for expansion of this project would be to allow users to input ratings for books they have already read into the data set. This new data would then be used to update the recommender system. Another feature could be added to allow users to skip through recommendations or get multiple book suggestions at once. In terms of the system itself, one area of extension would be to incorporate user modeling as a way to improve the results. This was not feasible based on the current data set, but obtaining additional information about the users' demographics and preferences would allow for implementation of user-based collaborative filtering, which would further refine the system's predictions.

References

1. Alencar, P., Cowan, D., & Portugal, I. (n.d.). *The Use of Machine Learning Algorithms in Recommender Systems: A Systematic Review.* Waterloo, Canada: University of Waterloo.
2. Rodriguez, D. (2013). *Recommender Systems.* Windsor: University of Alcala.
3. Lu, J., Mao, M., Wang, W., Wu, D., Zhang, G. (n.d.). *Recommender System Application Developments: A Survey.* Sydney, Australia: University of Technology.
4. *How Do Recommendation Engines Work? And What are the Benefits?*. (n.d.). Retrieved December 3, 2018, from https://www.marutitech.com/recommendation-engine-benefits/
5. Carleton College. (n.d.). *Recommender Systems*. Retrieved December 3, 2018, from http://www.cs.carleton.edu/cs_comps/0607/recommend/recommender/collaborativefilteri ng.html

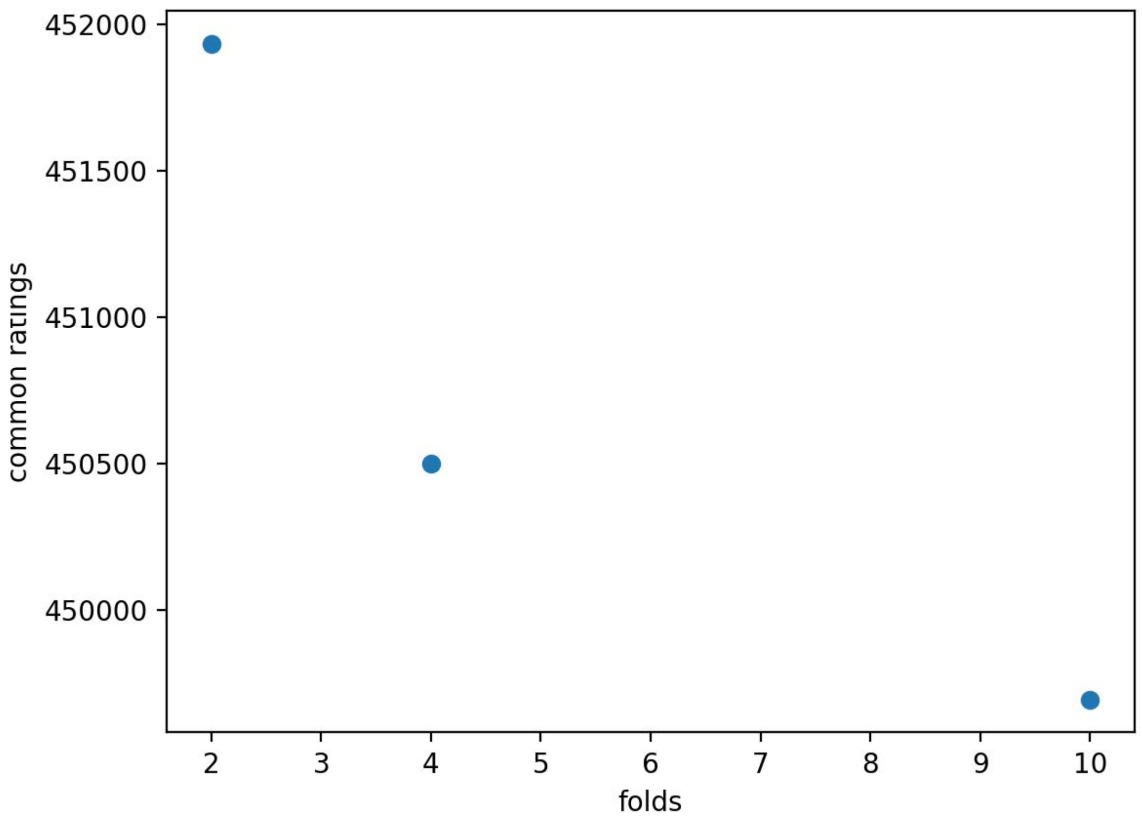
Appendices

1.



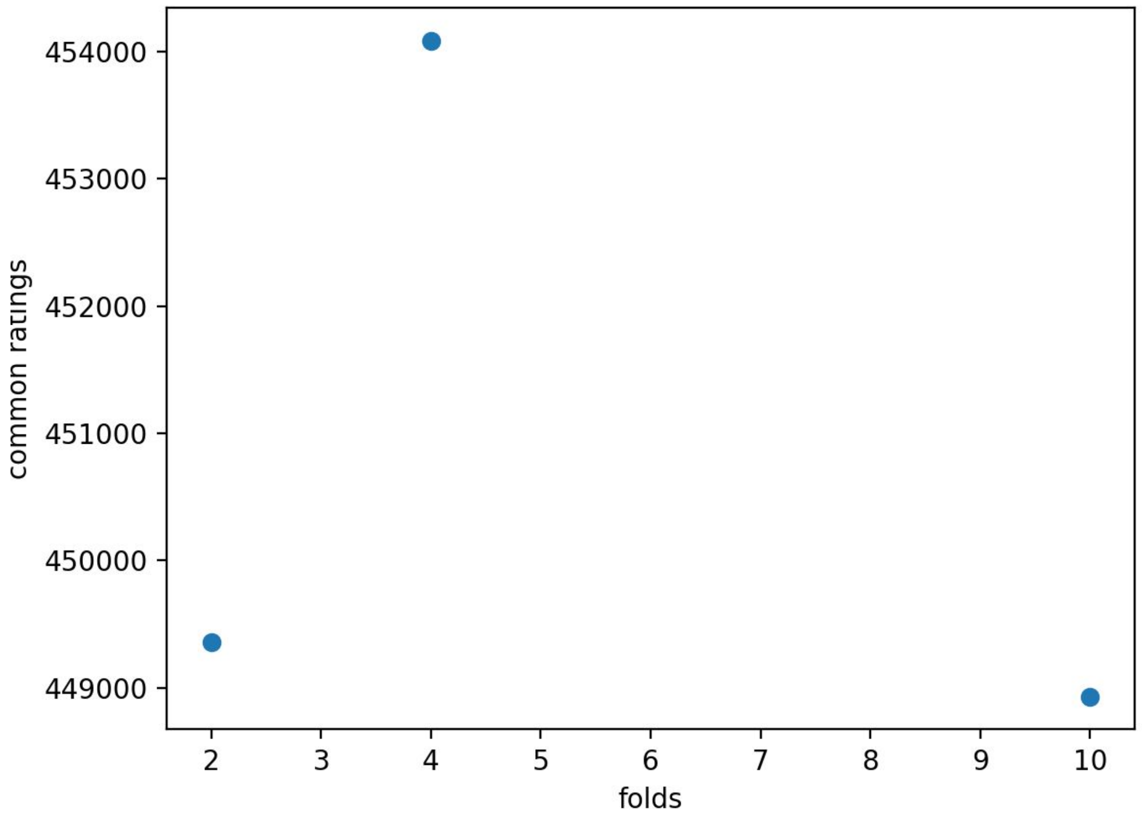
Appendix 1: Scatterplot of number of folds vs common ratings for threshold = 50

2.



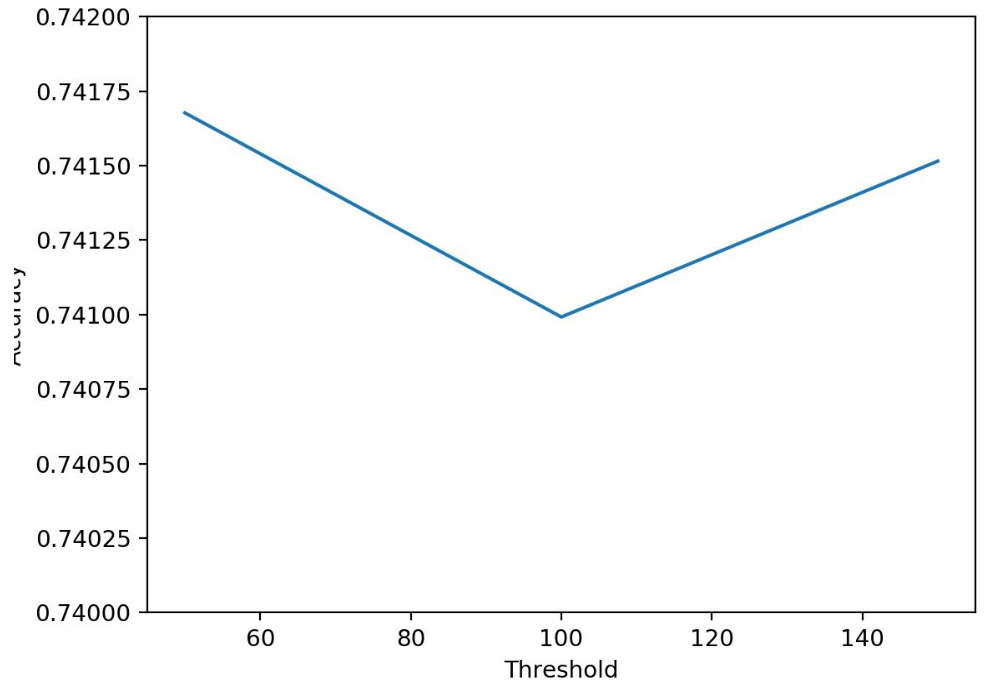
Appendix 2: Scatterplot of number of folds vs common ratings for threshold = 100

3.



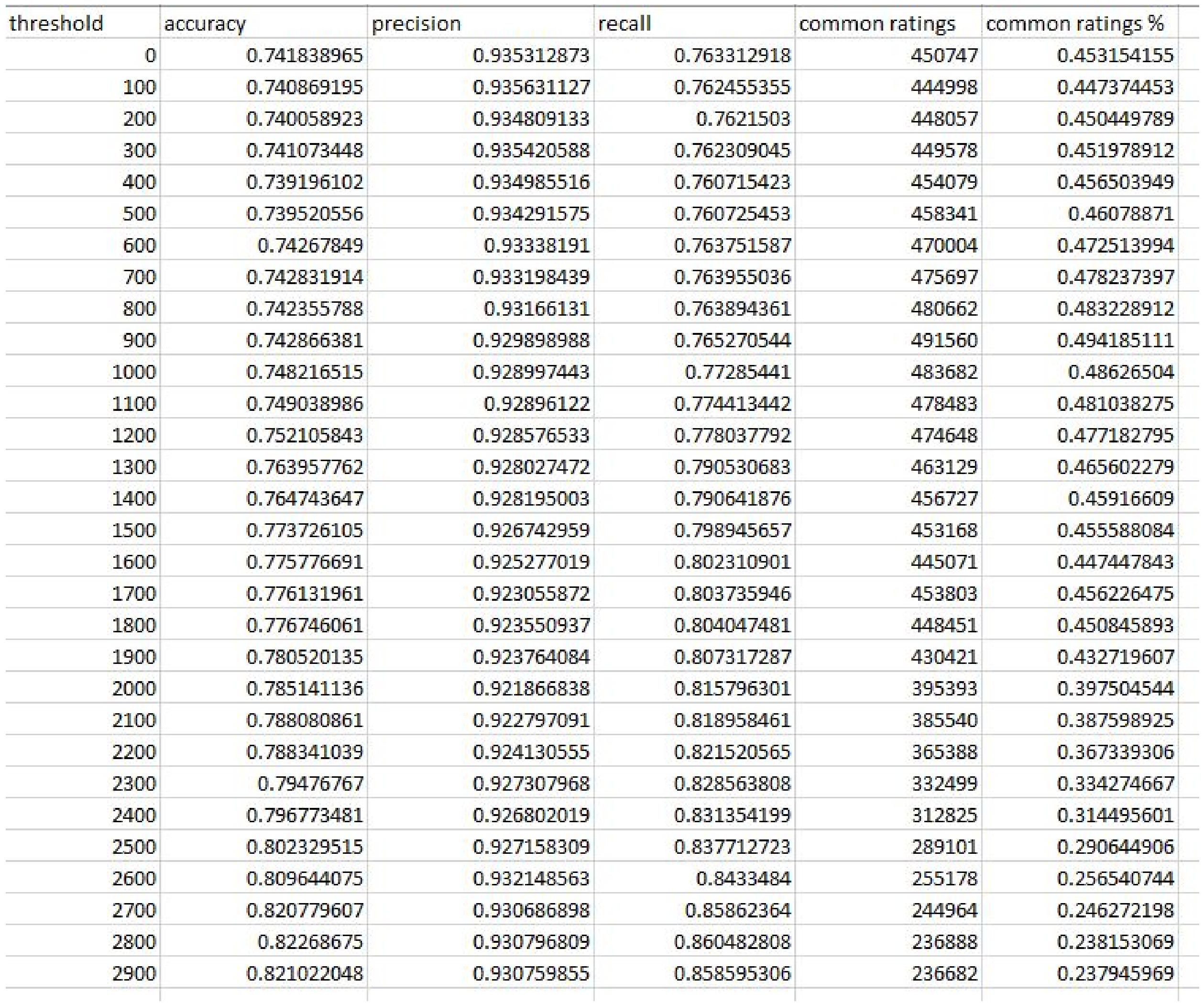
Appendix 3: Scatterplot of number of folds vs common ratings for threshold = 150

4.



Appendix 4: Graph of threshold vs accuracy

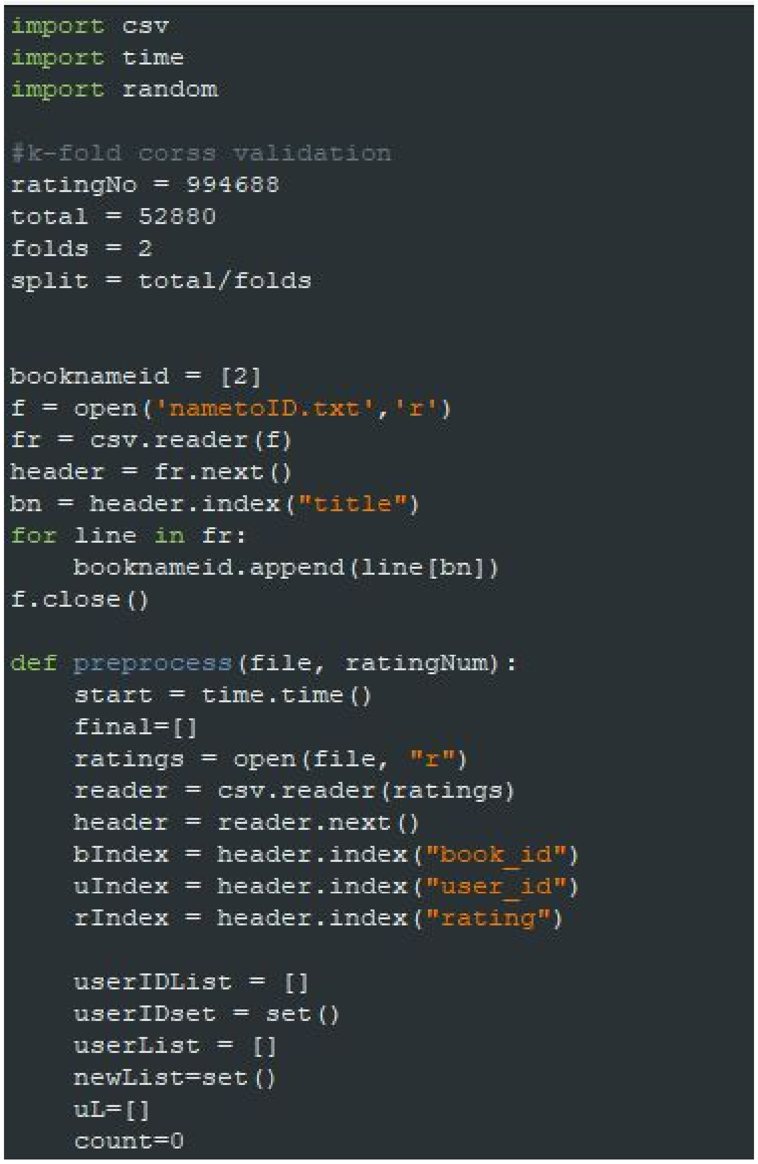
5.



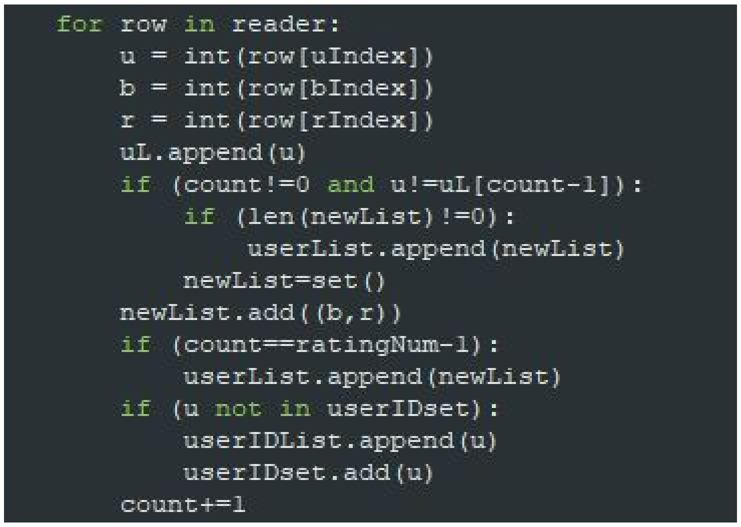
threshold	accuracy	precision	recall	common ratings	common ratings %
0	0.741838965	0.935312873	0.763312918	450747	0.453154155
100	0.740869195	0.935631127	0.762455355	444998	0.447374453
200	0.740058923	0.934809133	0.7621503	448057	0.450449789
300	0.741073448	0.935420588	0.762309045	449578	0.451978912
400	0.739196102	0.934985516	0.760715423	454079	0.456503949
500	0.739520556	0.934291575	0.760725453	458341	0.46078871
600	0.74267849	0.93338191	0.763751587	470004	0.472513994
700	0.742831914	0.933198439	0.763955036	475697	0.478237397
800	0.742355788	0.93166131	0.763894361	480662	0.483228912
900	0.742866381	0.929898988	0.765270544	491560	0.494185111
1000	0.748216515	0.928997443	0.77285441	483682	0.48626504
1100	0.749038986	0.92896122	0.774413442	478483	0.481038275
1200	0.752105843	0.928576533	0.778037792	474648	0.477182795
1300	0.763957762	0.928027472	0.790530683	463129	0.465602279
1400	0.764743647	0.928195003	0.790641876	456727	0.45916609
1500	0.773726105	0.926742959	0.798945657	453168	0.455588084
1600	0.775776691	0.925277019	0.802310901	445071	0.447447843
1700	0.776131961	0.923055872	0.803735946	453803	0.456226475
1800	0.776746061	0.923550937	0.804047481	448451	0.450845893
1900	0.780520135	0.923764084	0.807317287	430421	0.432719607
2000	0.785141136	0.921866838	0.815796301	395393	0.397504544
2100	0.788080861	0.922797091	0.818958461	385540	0.387598925
2200	0.788341039	0.924130555	0.821520565	365388	0.367339306
2300	0.79476767	0.927307968	0.828563808	332499	0.334274667
2400	0.796773481	0.926802019	0.831354199	312825	0.314495601
2500	0.802329515	0.927158309	0.837712723	289101	0.290644906
2600	0.809644075	0.932148563	0.8433484	255178	0.256540744
2700	0.820779607	0.930686898	0.85862364	244964	0.246272198
2800	0.82268675	0.930796809	0.860482808	236888	0.238153069
2900	0.821022048	0.930759855	0.858595306	236682	0.237945969

Appendix 5: Raw data for performance of Link Rating at each Threshold

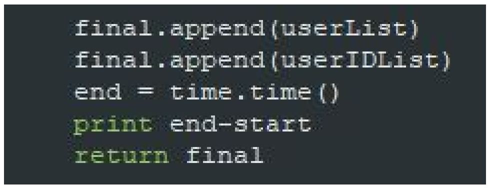
6. python code for entire project



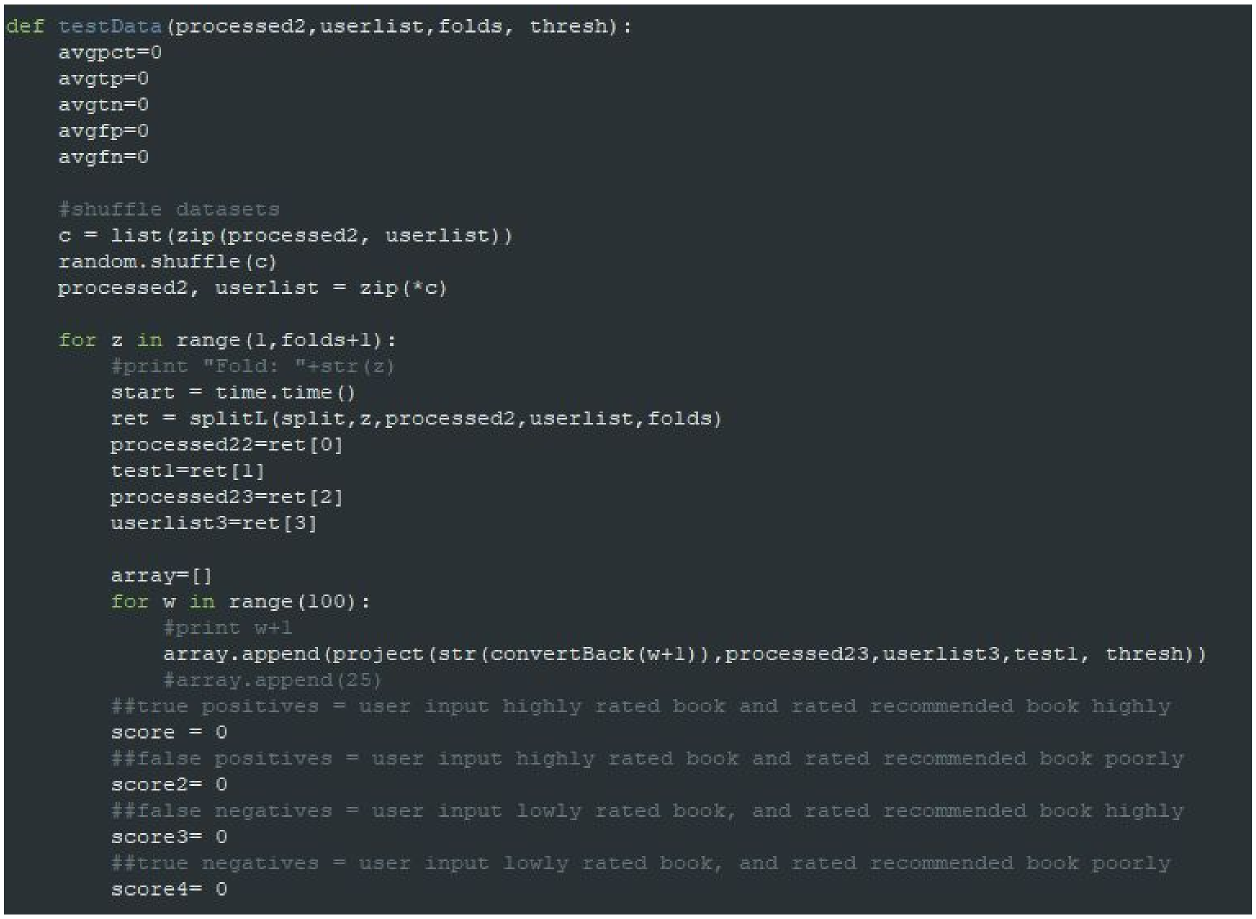
```python
import csv
import time
import random

#k-fold corss validation
ratingNo = 994688
total = 52880
folds = 2
split = total/folds


booknameid = [2]
f = open('nametoID.txt','r')
fr = csv.reader(f)
header = fr.next()
bn = header.index("title")
for line in fr:
    booknameid.append(line[bn])
f.close()

def preprocess(file, ratingNum):
    start = time.time()
    final=[]
    ratings = open(file, "r")
    reader = csv.reader(ratings)
    header = reader.next()
    bIndex = header.index("book_id")
    uIndex = header.index("user_id")
    rIndex = header.index("rating")

    userIDList = []
    userIDset = set()
    userList = []
    newList=set()
    uL=[]
    count=0
```



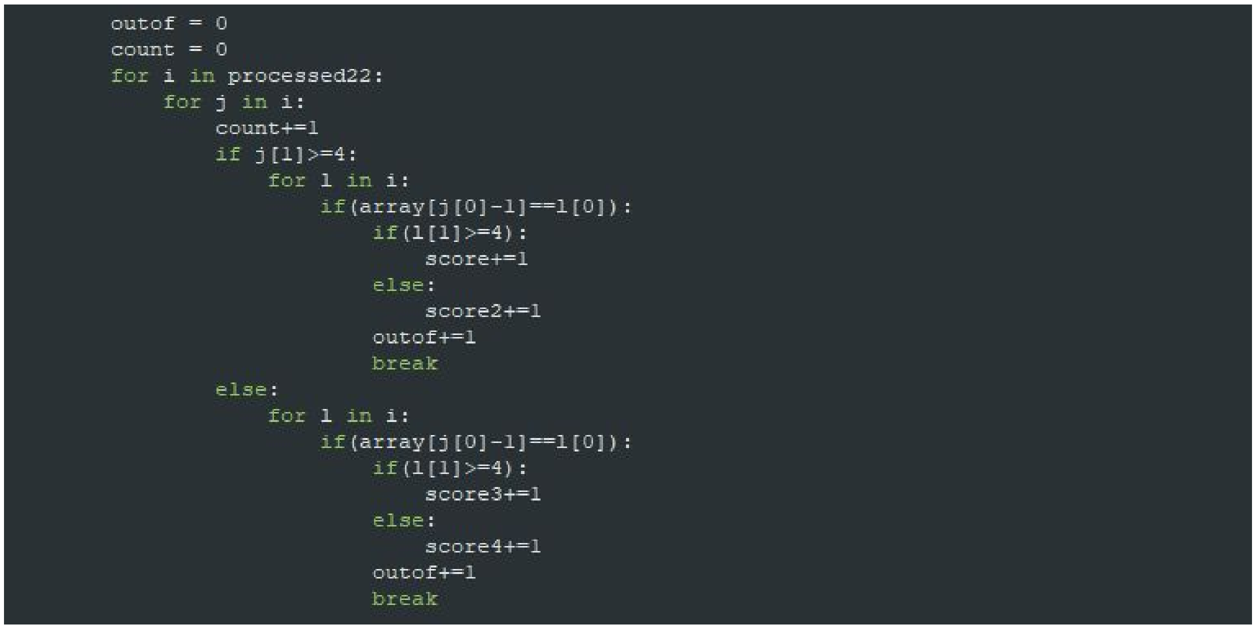
```python
    for row in reader:
        u = int(row[uIndex])
        b = int(row[bIndex])
        r = int(row[rIndex])
        uL.append(u)
        if (count!=0 and u!=uL[count-1]):
            if (len(newList)!=0):
                userList.append(newList)
            newList=set()
        newList.add((b,r))
        if (count==ratingNum-1):
            userList.append(newList)
        if (u not in userIDset):
            userIDList.append(u)
            userIDset.add(u)
        count+=1
```



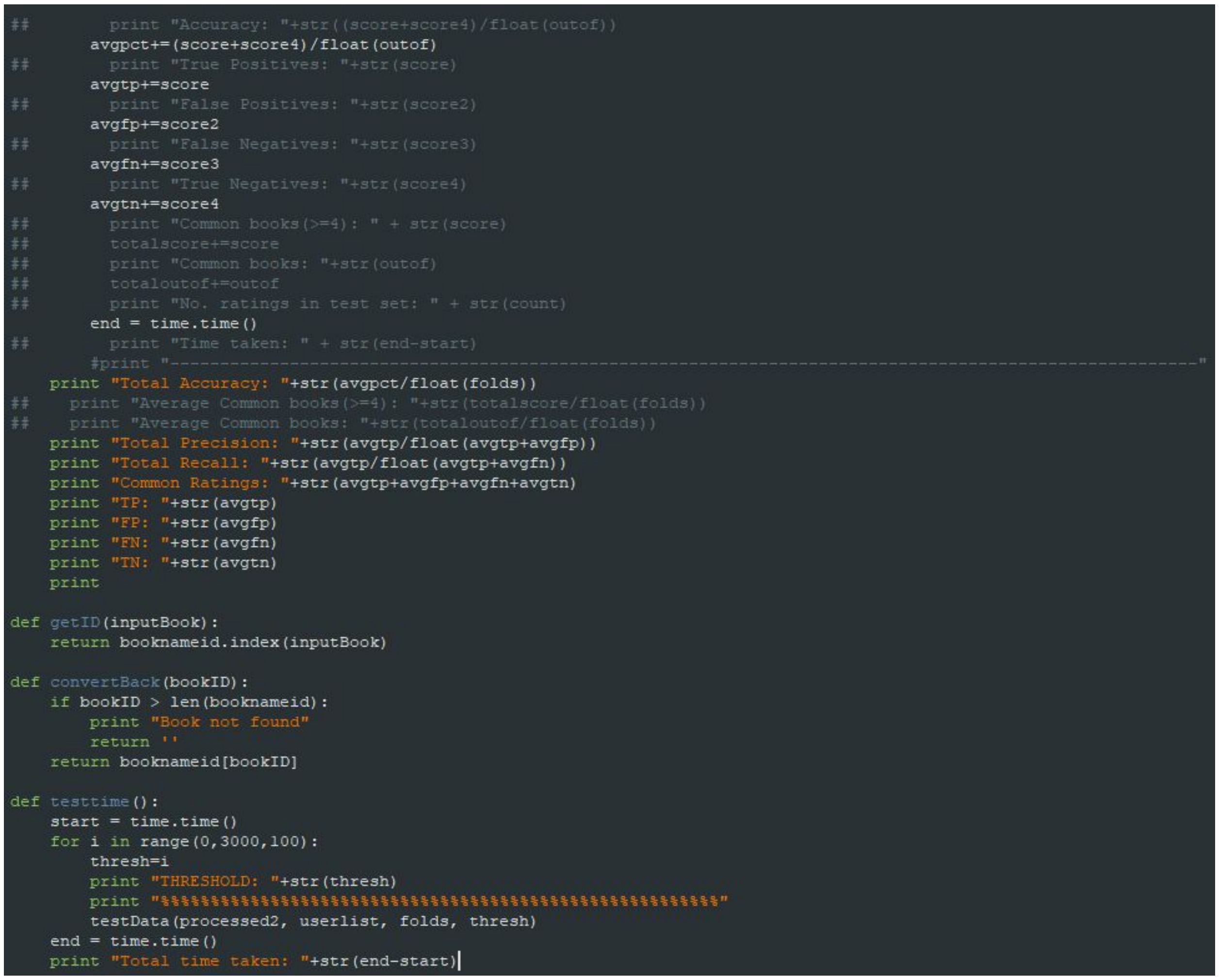
```
    final.append(userList)
    final.append(userIDList)
    end = time.time()
    print end-start
    return final
```



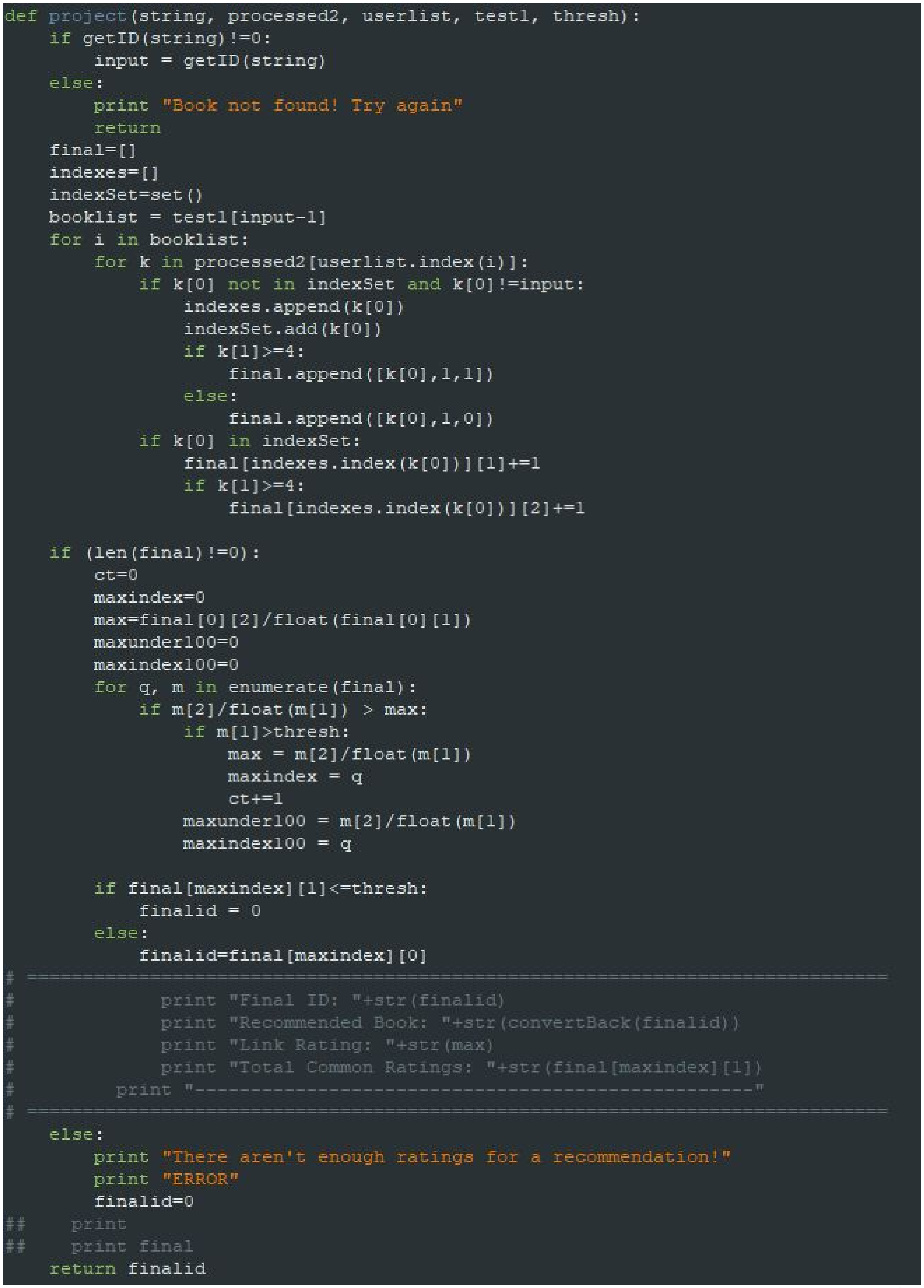
```
def testData(processed2,userlist,folds, thresh):
    avgpct=0
    avgtp=0
    avgtn=0
    avgfp=0
    avgfn=0

    #shuffle datasets
    c = list(zip(processed2, userlist))
    random.shuffle(c)
    processed2, userlist = zip(*c)

    for z in range(1,folds+1):
        #print "Fold: "+str(z)
        start = time.time()
        ret = splitL(split,z,processed2,userlist,folds)
        processed22=ret[0]
        test1=ret[1]
        processed23=ret[2]
        userlist3=ret[3]

        array=[]
        for w in range(100):
            #print w+1
            array.append(project(str(convertBack(w+1)),processed23,userlist3,test1, thresh))
            #array.append(25)
        ##true positives = user input highly rated book and rated recommended book highly
        score = 0
        ##false positives = user input highly rated book and rated recommended book poorly
        score2= 0
        ##false negatives = user input lowly rated book, and rated recommended book highly
        score3= 0
        ##true negatives = user input lowly rated book, and rated recommended book poorly
        score4= 0
```



```
        outof = 0
        count = 0
        for i in processed22:
            for j in i:
                count+=1
                if j[1]>=4:
                    for l in i:
                        if(array[j[0]-1]==l[0]):
                            if(l[1]>=4):
                                score+=1
                            else:
                                score2+=1
                            outof+=1
                            break
                else:
                    for l in i:
                        if(array[j[0]-1]==l[0]):
                            if(l[1]>=4):
                                score3+=1
                            else:
                                score4+=1
                            outof+=1
                            break
```



```
##        print "Accuracy: "+str((score+score4)/float(outof))
        avgpct+=(score+score4)/float(outof)
##        print "True Positives: "+str(score)
        avgtp+=score
##        print "False Positives: "+str(score2)
        avgfp+=score2
##        print "False Negatives: "+str(score3)
        avgfn+=score3
##        print "True Negatives: "+str(score4)
        avgtn+=score4
##        print "Common books(>=4): " + str(score)
##        totalscore+=score
##        print "Common books: "+str(outof)
##        totaloutof+=outof
##        print "No. ratings in test set: " + str(count)
        end = time.time()
##        print "Time taken: " + str(end-start)
        #print "----------------------------------------------------------------------------------------------------"
    print "Total Accuracy: "+str(avgpct/float(folds))
##    print "Average Common books(>=4): "+str(totalscore/float(folds))
##    print "Average Common books: "+str(totaloutof/float(folds))
    print "Total Precision: "+str(avgtp/float(avgtp+avgfp))
    print "Total Recall: "+str(avgtp/float(avgtp+avgfn))
    print "Common Ratings: "+str(avgtp+avgfp+avgfn+avgtn)
    print "TP: "+str(avgtp)
    print "FP: "+str(avgfp)
    print "FN: "+str(avgfn)
    print "TN: "+str(avgtn)
    print

def getID(inputBook):
    return booknameid.index(inputBook)

def convertBack(bookID):
    if bookID > len(booknameid):
        print "Book not found"
        return ''
    return booknameid[bookID]

def testtime():
    start = time.time()
    for i in range(0,3000,100):
        thresh=i
        print "THRESHOLD: "+str(thresh)
        print "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%"
        testData(processed2, userlist, folds, thresh)
    end = time.time()
    print "Total time taken: "+str(end-start)
```



```python
def project(string, processed2, userlist, test1, thresh):
    if getID(string)!=0:
        input = getID(string)
    else:
        print "Book not found! Try again"
        return
    final=[]
    indexes=[]
    indexSet=set()
    booklist = test1[input-1]
    for i in booklist:
        for k in processed2[userlist.index(i)]:
            if k[0] not in indexSet and k[0]!=input:
                indexes.append(k[0])
                indexSet.add(k[0])
                if k[1]>=4:
                    final.append([k[0],1,1])
                else:
                    final.append([k[0],1,0])
            if k[0] in indexSet:
                final[indexes.index(k[0])][1]+=1
                if k[1]>=4:
                    final[indexes.index(k[0])][2]+=1

    if (len(final)!=0):
        ct=0
        maxindex=0
        max=final[0][2]/float(final[0][1])
        maxunder100=0
        maxindex100=0
        for q, m in enumerate(final):
            if m[2]/float(m[1]) > max:
                if m[1]>thresh:
                    max = m[2]/float(m[1])
                    maxindex = q
                    ct+=1
                maxunder100 = m[2]/float(m[1])
                maxindex100 = q

        if final[maxindex][1]<=thresh:
            finalid = 0
        else:
            finalid=final[maxindex][0]
# =============================================================================
#             print "Final ID: "+str(finalid)
#             print "Recommended Book: "+str(convertBack(finalid))
#             print "Link Rating: "+str(max)
#             print "Total Common Ratings: "+str(final[maxindex][1])
#         print "--------------------------------------------------"
# =============================================================================
    else:
        print "There aren't enough ratings for a recommendation!"
        print "ERROR"
        finalid=0
##    print
##    print final
    return finalid
```