Speedy Recs

A Unique Anime Recommender System

Mark Cunningham
markcham@ou.edu

Ethan Saloom
esaloom@ou.edu

Branson Stickney
bstick2@ou.edu

ABSTRACT

Recommender systems are implemented in nearly every website that offers consumers a product. In this paper, we describe the process behind creating a recommender system by computing ratings and recommendations using five different algorithms, including one hybrid algorithm. We evaluated different parameters involved in model creation for several algorithms, and assessed multiple evaluation schemes parameters, which are used to evaluate the models using a train and test partitioned dataset. We then evaluated their performance against each other using multiple error and accuracy measurements.

Keywords

UBCF; IBCF; SVD; ALS; anime; recommender; system; user-based; item-based; collaborative; filtering; singular; value; decomposition; alternating; least; squares; recommenderlab.

1. INTRODUCTION

The objective of Speedy Recs is to recommend new anime for users to watch based on their previously rated anime titles. In this paper, we refer to "anime", which is referring to all categories or formats that MyAnimeList separates Japanese animated content into: TV series, movies, original net animation (ONA), original video animation (OVA), and specials. Anime and rating data was scraped from MyAnimeList by Hernan Valdivieso and uploaded to Kaggle, which is where we source our datasets from. In the following sections, we discuss what data preprocessing activities were required in order to correctly work with the algorithms.

We used RStudio to manipulate our data, create models, and generate our recommendations. RStudio has many different packages we can use to further analyze the algorithms for errors, as well as see how the algorithms compare to each other. We wanted to dive into which algorithms work best with a recommender system. We used RStudio to preprocess our data to narrow our search to make for faster recommendations. The preprocessed data is then sent into the algorithms, most of the data is used to train and the rest is used to test. Once a recommendation has been generated, we can then calculate for error between the test and the generated value. Using RStudio, we can then plot the errors on graphs so that we can visualize which algorithms are generating better recommendations. We will explain more about each algorithm, how the algorithm works with our data, and the errors generated from each algorithm.

Also, the Speedy Recs application will be explained. The user interface has a simple design. The complete user interface will be explained and the steps on how to use the interface will be clarified. Our goal is to deliver an application to help anime watchers find new shows, but also to reveal the aspects of what goes into creating a recommender system and the different algorithms you can implement.

2. RELATED WORK

Many of the recommenders we found didn't provide as in-depth of a description behind parameter setting, difference in algorithms, and most importantly they did not have a functional application which showed the outputs of different algorithms from user input. In addition, almost none of the recommenders offered recommendations for anime, but instead offered general movie recommendations. MyAnimeList (MAL) is a popular site to find new anime, the problem is, they do not recommend anime based on your preferences. MAL normally recommends just popular anime, which can be nice, but we want recommendations based on our watching preferences.

3. Data Preprocessing & Preview

3.1 Initial Dataframe Filtering

The dataset being used for this report is the Anime Recommendation Database 2020 from Kaggle.com, which retrieved its data from MyAnimeList.com. The database contains a number of spreadsheets, but we used just the following two spreadsheets: anime.csv and rating_complete.csv. We chose anime.csv because it contains 35 columns of data consisting of interval, nominal, and ordinal data describing the anime themselves.

We begin by filtering out explicit content and anime that has less than 1000 users having completed the anime. The ratings dataframe is then updated to remove those corresponding anime. We also remove columns we know that won't be used going forward, which leaves us with 18 columns of data. The resulting data frames are named anime_df and ratings_df, respectively.

3.2 Previewing the Data

```
## Rows: 6
## Columns: 18
## $ MAL_ID       <int> 1, 5, 6, 7, 8, 16
## $ Name         <chr> "Cowboy Bebop", "Cowboy Bebop: Tengoku no
## $ Score        <chr> "8.78", "8.39", "8.24", "7.27", "6.98", "
## $ Genres       <chr> "Action, Adventure, Comedy, Drama, Sci-Fi
## $ English.name <chr> "Cowboy Bebop", "Cowboy Bebop:The Movie",
## $ Type         <chr> "TV", "Movie", "TV", "TV", "TV", "TV"
## $ Episodes     <chr> "26", "1", "26", "26", "52", "24"
## $ Aired        <chr> "Apr 3, 1998 to Apr 24, 1999", "Sep 1, 20
## $ Premiered    <chr> "Spring 1998", "Unknown", "Spring 1998",
## $ Studios      <chr> "Sunrise", "Bones", "Madhouse", "Sunrise"
## $ Source       <chr> "Original", "Original", "Manga", "Origina
## $ Duration     <chr> "24 min. per ep.", "1 hr. 55 min.", "24 m
## $ Rating       <chr> "R - 17+ (violence & profanity)", "R - 17
## $ Ranked       <chr> "28.0", "159.0", "266.0", "2481.0", "3710
## $ Popularity   <int> 39, 518, 201, 1467, 4369, 687
## $ Members      <int> 1251960, 273145, 558913, 94683, 13224, 21
## $ Completed    <int> 718161, 208333, 343492, 46165, 7314, 8114
## $ Dropped      <int> 26678, 770, 13925, 5378, 1108, 11026
```

Figure 1: anime_df data frame

```
## Rows: 6
## Columns: 3
## $ user_id  <int> 0, 0, 0, 0, 0, 0
## $ anime_id <int> 430, 1004, 3010, 570, 431, 578
## $ rating   <int> 9, 5, 7, 7, 8, 10
```

Figure 2: ratings_df data frame

The "types" of anime found in our dataset are shown in Figure 3. These include mostly "TV" series, which represent seasonal anime that generally has 12 episodes per season. Each season of an anime has its own unique rating within the anime_df dataframe.

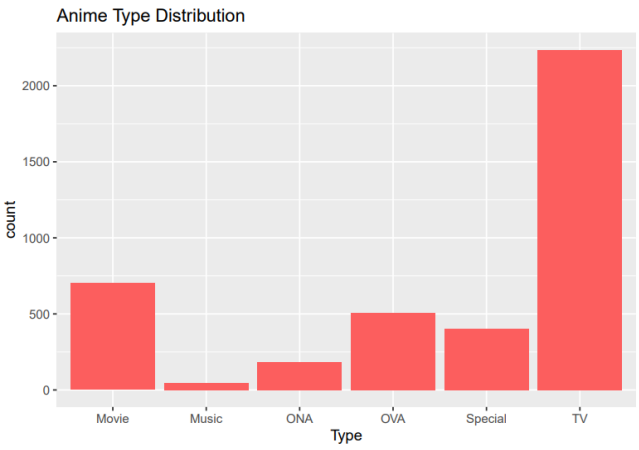

Figure 3: Anime Type Distribution

3.3 Sampling the Dataframes

In order for us to accomplish our goals of comparing algorithms on equal ground and providing recommendations as fast as possible, it was necessary for us to sample our ratings_df dataframe. We began by converting the ratings dataframe into a realRatingMatrix, which is a special kind of sparse matrix that the Recommenderlab package uses as inputs for almost all of its algorithms. A sparse matrix in this context is a user-item matrix which only stores user ratings for items that have a rating. This helps massively with efficiency since most of the values in our user-item matrix are NA. This matrix can be converted back to a data frame at any time. Since we're mainly concerned with anime that have a larger audience, we removed all anime that have less than 1000 ratings. We also removed users who have rated less than 100 anime in order to generate more specific similarity vectors while also reducing the size of our rating matrix. Users with a very low amount of ratings slow down the efficiency of algorithms that depend on user-item matrices because a user with a small amount of ratings doesn't contribute much to algorithm learning. However, with a row count of 151,233, a file size of 477 megabytes, and 38,937,705 ratings, the matrix was still too large to even be accepted into any recommender algorithms.

To remedy this, we used the sample function in R on the realRatingMatrix, which randomly samples a given number of users within the matrix. We chose a value of 1000 since it was able to generate relatively quick recommendations while still sampling all 3140 anime in our anime_df data frame, even after repeated sampling. The size of the newly sampled realRatingMatrix was 3.4 megabytes and included 260682 total ratings. Initially, we sampled 5000 users, but some algorithms proved to be too time consuming to continue using this value. These algorithms are discussed in Section 4.

3.4 Post-Sampling Data Preview

Next, we take a look at the distributions of ratings after applying filtering and sampling, which is shown in Figure 4. We can see that the most common rating is eight, followed closely by seven and nine, respectively. If we normalize the ratings, we can see that they do indeed follow a normal distribution, with the majority of ratings concentrated towards the center of the distribution. This is shown in Figure 5.

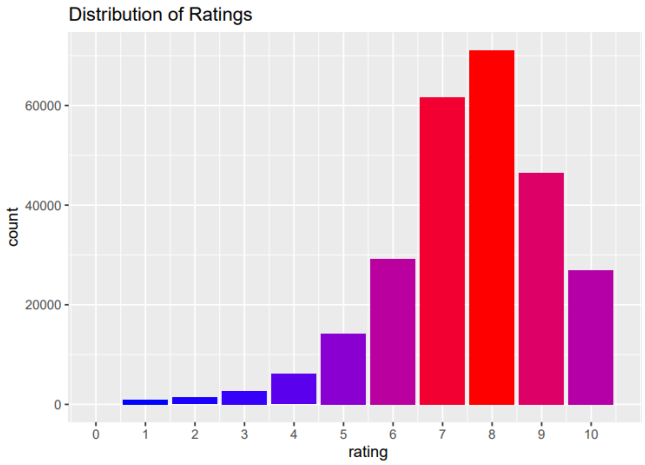

Figure 4: Distribution of Ratings

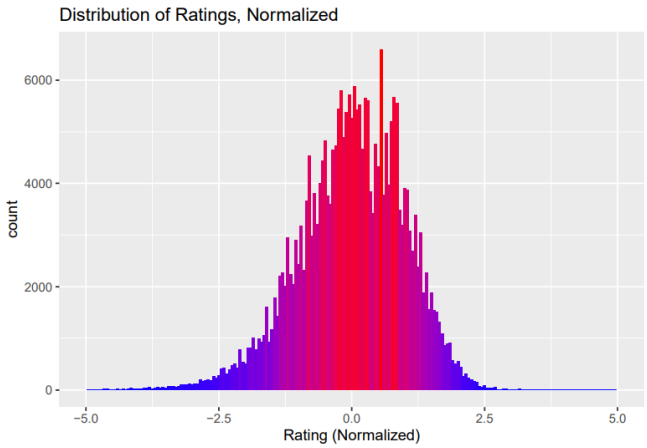

Figure 5: Distribution of Ratings, Normalized

After filtering by users with greater than 100 ratings, we can see that most users are concentrated towards the 100-300 rating range, with a steep dropoff afterwards. The distribution of ratings are shown in Figure 6.

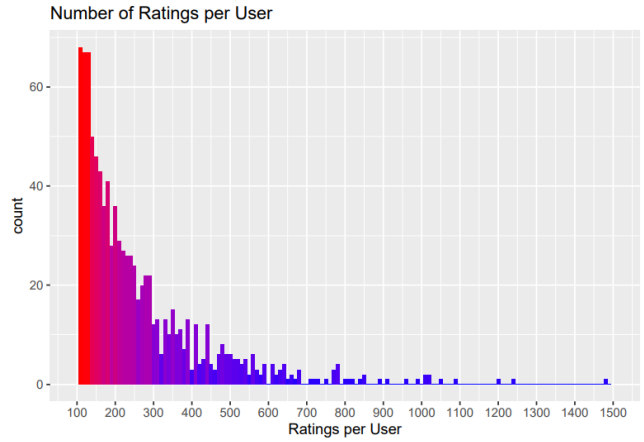

Figure 6: Distribution of Ratings, Normalized

3.5 Examining Similarity

Before applying any algorithms we can examine the similarity between users and items. The distance measure used here is Cosine similarity, and only shows the first 100 users and items (anime). These similarities are shown in Figure 7 and 8.

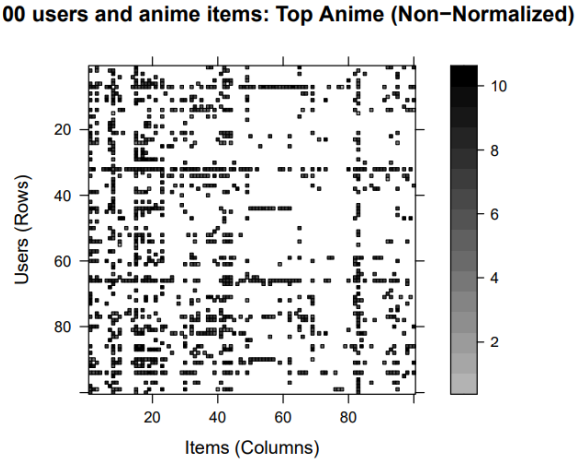

Figure 7: User-Item Similarity

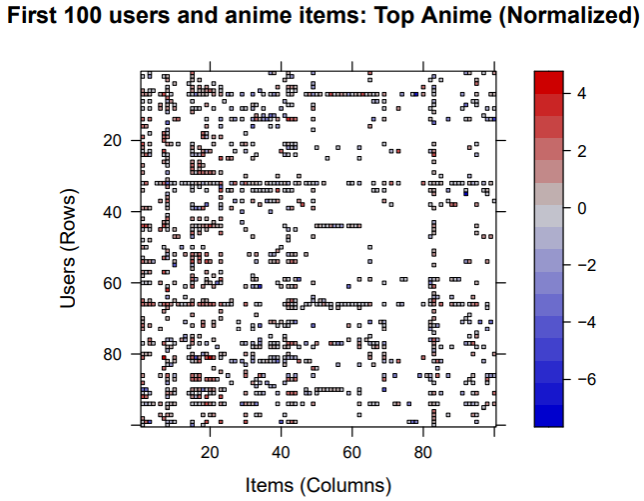

Figure 8: User-Item Similarity, Normalized

Next, we examine the similarity between individual anime and the similarity between individual users. From Figure 9, we see that anime similarity is a lighter color than user similarity, signaling that anime have less in common than users do. The anime similarity image has greater deviations of color than the user similarity image, which suggests that there are higher levels of differentiation between individual anime. However, this is only the first 100 items, so it might not paint the full picture.

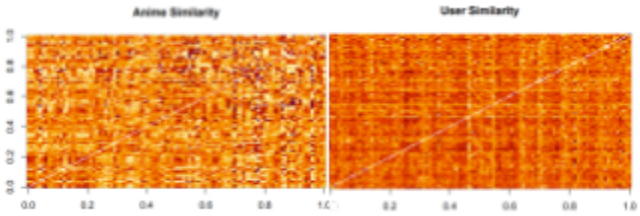

Figure 9: Anime Similarity and User Similarity

4. Creating the Algorithms

For our recommender system, we will be utilizing User-Based Collaborative Filtering (UBCF), Item-Based Collaborative Filtering (IBCF), and matrix factorization in combination with Alternating Least Squares (ALS). Collaborative filtering is a widely used approach to create recommender systems that uses the assumption that if two users have similar ratings for a particular anime, they are more likely to have similar ratings for another anime than a randomly selected person. Collaborative filtering relies on user ratings to generate a list of similar users, or to generate similar items (anime). Collaborative filtering consists of memory-based and model-based techniques. Latent-factor models are another popular choice in recommender systems that use matrix factorization to “uncover latent features that explain observed ratings” [Hu, 2008].

4.1 Evaluation Schemes

Recommenderlab offers the evaluationScheme function, which allows the user to specify how the train and test sets should be divided and how they should be evaluated. In the following paragraphs, we discuss the methods which we use.

An important additional parameter that we use throughout the paper is the "given" parameter, which specifies the number of items given to a single user for training. This parameter can also implement an all-but-x approach using a negative number, meaning that the evaluation scheme will use all ratings in the train set, but x number of ratings will be withheld in the test set in order to evaluate.

The split method splits the users into two groups based on the value of the train parameter and uses the first half of the split to train the model, and the second half to evaluate the model. If using an all-but-x given scheme, then the scheme uses only the second half of the split to exclude ratings from. We implemented the split method using an 80/20 split. The cross method uses k-fold cross validation and uses the user specified k-value as the number of folds.

Four evaluationScheme objects were created in our R program, including two split methods and two cross methods. For each of these two methods we also varied the "given" parameter values to the minimum number of ratings in the dataframe and an all-but-x approach using x=5.

4.2 Item-Based Collaborative Filtering

IBCF is different from UBCF in that it is a model-based technique, and it generates its recommendations based on similarities between items learned from an anime-anime matrix. We used the Cosine similarity method to measure distances between ratings. The ratings are normalized using a parameter argument within the function call. This algorithm can be broken down into three steps. First, all similarities between anime are stored in a matrix. This is the learned model. Second, a given user inputs the ratings for a number of anime. Lastly, the weighted sum of the user's ratings is calculated from the similarities and the ratings for anime that the user has seen. The result is a list of anime with predicted ratings. For our user interface, only the top-N anime will be shown.

4.2.1 Which Value of K to Choose?

The value of k in the IBCF model corresponds to the number of closest neighbors to store for each anime. We measured the error values with 10 different k-values using the predict function available in base R and the calcPredictionAccuracy function from RecommenderLab. The results are shown in Figure 10. The error curves appear roughly mound-shaped, indicating that the lowest and highest k-values have the lowest errors. However, we didn't want to use a low k-value since that wouldn't allow a large number of similar items, which might be necessary because our program only takes in 5 ratings from the user. The variance in RMSE was relatively insignificant and we noticed a decline in error at k=50, which is a healthy compromise between model build time and error rate.

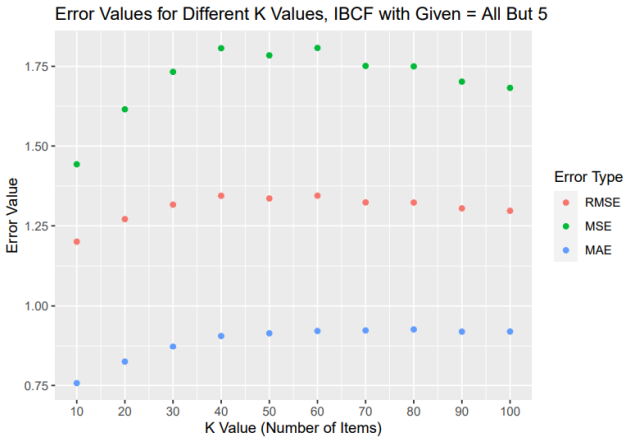

Figure 10: Error Values for Different K-Values

4.3 User-Based Collaborative Filtering

The memory-based algorithm we employed is UBCF. This technique finds unknown ratings, or predictions, for a user by finding a neighborhood consisting of k-most similar users or "all users within a given similarity threshold" [Hahsler, 2022]. This algorithm can be broken down into two steps for a given user seeking recommendations. First, the k-most similar users are found based on their user-anime similarity. The second step is to generate an average rating for each anime not yet rated by the given user, based on the averages of the users within the neighborhood consisting of k-most similar users. Memory-based techniques employ measures of similarity like the Pearson coefficient and Cosine similarity [Isinkaye, 2015]. We used the Cosine similarity measure as a parameter for our model creation. We then select the highest rated anime from the list of averages within the neighborhood. Below, Figure 11 visualizes this algorithm.

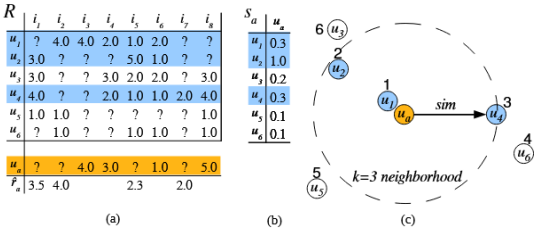

Figure 11: UBCF Algorithm Visual [Hahsler 2022]

4.3.1 Which Value of nn to Choose?

The nn-value is a parameter within the UBCF function that Recommenderlab offers. It represents the number of nearest neighbors the algorithm calculates for each user. We measured the error values with 10 different nn-values using the predict function available in base R and the calcPredictionAccuracy function from RecommenderLab. From Figure 12, we can estimate that error rate decreases linearly, although it starts to reach diminishing returns around nn=50. Once again, we choose 50 since it is an appropriate trade off between prediction time and error rate.

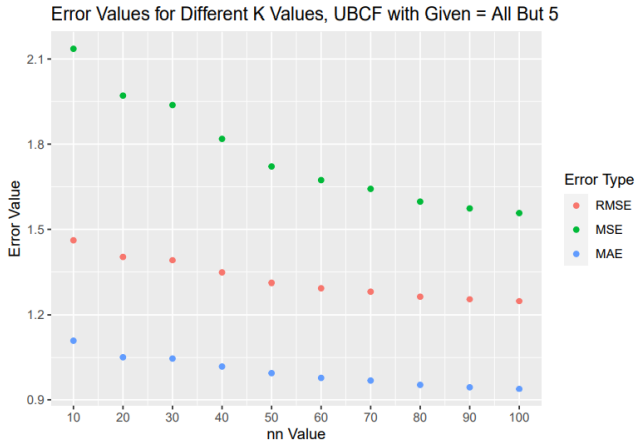

Figure 12: Error Values for Different nn-Values

4.4 Alternating Least Squares

Matrix factorization is a dimensionality reduction technique where a higher-level matrix is broken into lower order matrices. Singular Value Decomposition (SVD) is a specific kind of matrix factorization where it achieves dimensionality reduction by "transforming correlated variables into a set of uncorrelated ones" to better expose the original underlying relationships [Baker, 2005]. Applying this technique to recommender systems allows us to break our user-anime ratings matrix into two lower order matrices: a user matrix and an anime matrix. However, matrix factorization is only a data pre-processing step, because the matrix still must be solved for. To accomplish this, we will use an iterative algorithm called Alternating Least Squares (ALS) to minimize the error between the predicted and true values of the SVD matrix [Gosh, 2020]. This algorithm works by solving x_u and y_i, which are k-dimensional vectors that represent every user and every anime, respectively. Essentially, ALS holds one of these vectors constant, while using gradient descent to solve for the other, and then repeats the process holding the other vector constant, for the maximum iterations specified [Zhengzheng 2017].

Within Recommenderlab, two notable parameters are available to pass to the model creation function, which are n_factors and n_iterations. N_factors are the number of latent factors the model creates based on the lower-dimensional matrices projected from the user-anime matrix. We left this value as the default, which is ten. N_iterations is the number of iterations that the model undergoes when solving for the unknown ratings. When the model is solving these ratings, the prediction error lowers with more iterations, but ALS in Recommenderlab is extremely time consuming. In fact, we had to lower the number of iterations from the default of ten down to five in an effort to increase the speed of recommendations generated using ALS.

Algorithm : *Alternating Least Squares (ALS)*

Procedure ALS (x_u ,y_i)
Initialization $x_u \leftarrow 0$
Initialization matrix y_i *with random values*
Repeat
    Fix y_i *, solve* x_u *by minimizing's the objective function (the sum of squared errors)*
    Fix y_i *solve* y_i *by minimizing the objective function similarly*
Until reaching the maximum iteration
Return x_u ,y_i
End procedure

Figure 13: Alternating Least Squares Algorithm [Gosh 2020]

4.5 Singular Value Decomposition

Singular Value decomposition is similar to the alternating least squares method, but it only solves the SVD matrix, and does not iterate to make the error smaller. The SVD model that Recommenderlab uses solves for missing values using column-mean imputation, meaning that all NA values are replaced with column means and the resulting matrix is a unique solution obtained after convergence. We used the default parameters for this algorithm.

4.6 Hybrid Model

The hybrid model is a simple combination of Recommenders, where each model is given a user-defined weight. We chose our hybrid model to have IBCF, UBCF, SVD, and Popular algorithms, with a weighting of 0.4, 0.2, 0.2, 0.2, respectively.

5. Model Evaluation

Evaluating recommender systems can be performed using a number of different performance metrics, including rating error rate, recommendation accuracy, information retrieval measures, and probability of detection. Some of the most common rating accuracy methods to evaluate recommender systems are the Root mean square error (RMSE), which finds the standard deviation between the real vs predicted ratings, the Mean Squared Error (MSE), which finds the mean of squared difference between the real vs predicted ratings, and the Mean Absolute Error (MAE), which finds the mean of the absolute difference between real vs predicted ratings [Hahsler 2022]. For these methods, the data is required to be split into a training and test set. Models learn using the training set and are evaluated using the test set. The differences between the actual ratings in the test set and predicted ratings in the test set were recorded for all algorithms used in this project.

5.1 Predicted Ratings for Known Part of Test Data

This step of model evaluation involved creating prediction matrices of type realRatingMatrx, which used the "known" partition of the data split that was created using in section 4.1 to predict the "unknown" partition of the data split. For example, we used an evaluationScheme named anime_eval_split_allbut5. The result of computing the predicted ratings for this evaluation scheme is a realRatingMatrix with 200 rows, representing the

20% "unknown" section of the split, and 3140 columns, representing the 3140 unique anime titles. In a similar fashion, the predicted ratings for a 4-fold cross-evaluation consist of a realRatingMatrix with 250 rows.

5.2 Calculating Error Between Prediction and Unknown Part of Test Data

Recommenderlab provides a function called calcPrediction Accuracy, which measures the three different error value measurements described in the Section 5 introduction. The function takes two arguments: the predicted ratings matrices (described in Section 5.1) and the actual "unknown" ratings that were partitioned when we created the evaluation schemes. Figure 14 shows the total error values for all different algorithms and all different evaluation schemes used.

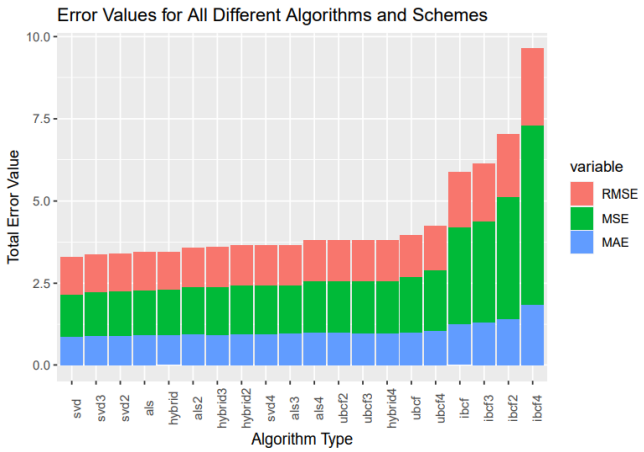

Figure 14: Error Values for All Algorithms and Schemes

The algorithms with the lowest total average error were Alternating Least Squares and Singular Value Decomposition. This doesn't necessarily mean they will produce the best recommendations, but it does mean they are the best at predicting ratings given learning data. The error values using the cross-validation method are shown in Figure 15, while the error values using the split-method are shown in Figure 16. Unsurprisingly, error values using the minimum number of ratings as the "value" parameter discussed earlier had higher error values than using the all-but-5 approach, since they had less data to train on. Our application will use the cross method since it is less time intensive than cross-validation, and use the all-but-5 approach as the "given" parameter.

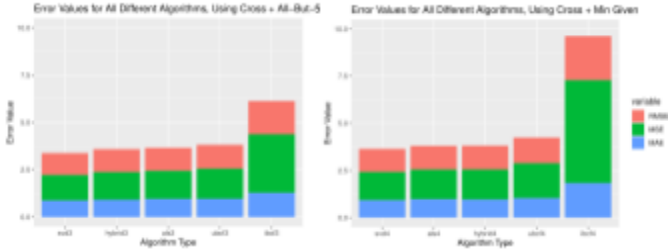

Figure 15: Error Values for All Algorithms, Cross Method

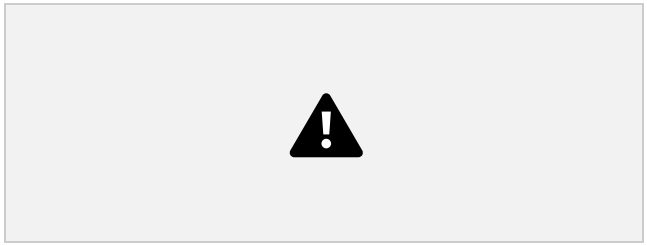

Figure 16: Error Values for All Algorithms, Split Method

5.3 Precision-Recall Curve

Precision is defined as the number of true positives divided by all positives (true positives and false positives), whereas recall (or sensitivity) is the number of true positives divided by all results that should have been positive (true positives and false negatives). Precision-recall curves are mainly used for classes which are heavily imbalanced, so it isn't a perfect match for our dataset, but to be comprehensive, we included it anyway.

To generate a precision-recall curve using Recommenderlab, we use the evaluate() function in R, which takes the following arguments: evaluationScheme, a list of algorithms, and the type of result to evaluate. In this case, we are looking to evaluate a "topNList", which is a list generated from the algorithms of their most recommended items. The numbers on the graph correspond to the size of the list.

A higher precision value corresponds to a low false positive rate, while a high recall value relates to a low false negative rate. When looking at Figure 17, we can see that IBCF initially generates recommendations with a low false positive rate, but as the size of the list of recommendations increases, it shifts to preferring a low false negative rate.

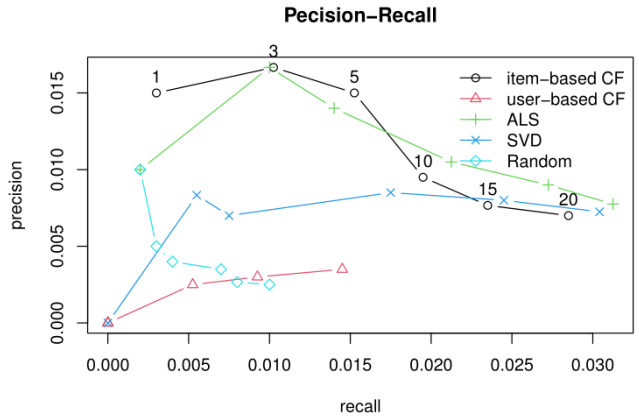

Figure 17: Precision-Recall Curve

5.4 ROC Curve

We can also choose to measure the generated lists of recommendations. For this performance metric, we must choose a threshold value for a recommendation to be considered a true positive. For instance, if we choose 5/10, the true positive condition would be an anime recommendation with a 5/10 rating, while the false positive condition would be an anime recommendation with less than a 5/10 rating. An ROC curve plots the true positive rate against the false positive rate, and is linear with a slope of one with a random classifier. The larger the area

under the curve, the better the classifier performance. The ROC curve is shown in Figure 18. All of our models outperformed the random classifier, although UBCF only starts to beat it once the list is greater than or equal to ten. More often than not, anime that are very popular are also highly rated, so it makes sense that the popular algorithm performed very well.

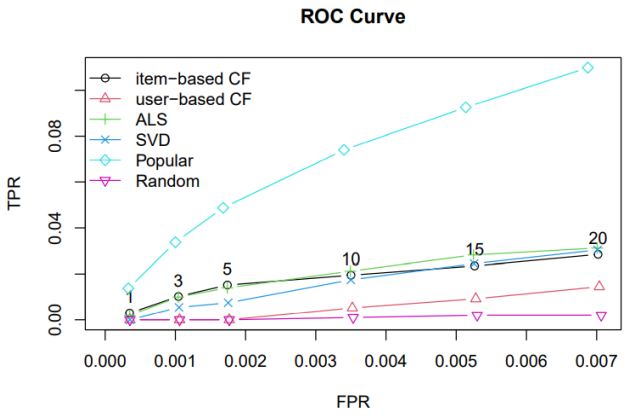

Figure 18: ROC Curve

6. User Interface

6.1 User Manual

To create a Shiny application, there needs to be a file named app.R, which consists of two main components: the UI and the server. The UI is the front-end of the application, and includes modules that render various components within the application. The server is a function which takes in the input object, output object, and session in order to communicate between objects. It is essentially the back end of the application, responsible for connecting the code to the user.

We made use of selectizeInput buttons to obtain data from the application user. Any input that these buttons receive is stored in the input object, and can be accessed by the server by using input$button1. The function updateSelectizeInput allows the dropdown lists showing the anime names to update after every letter is typed into the text box. This allows us to not be forced to display all 3140 anime at one time in every box, which speeds up the application. The updating feature is shown in Figure 19.

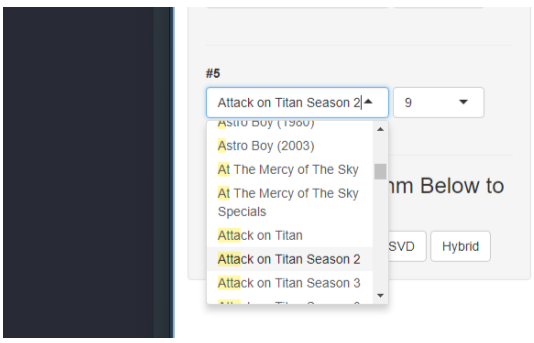

Figure 19: Auto-updating Anime Selection Box

The observeEvent function monitors for when the user clicks one of the five recommendation buttons, and displays a table to output that contains the return value for one of the five recommender functions. The recommender functions are stored in a separate file named userrecommender.R to keep the app.R file clean and organized, but could've also been put inside the app.R file as well. A source() function call containing userrecommender.R imports this R file into the app.R file. The userrecommender.R file contains six functions and imports eight .rda files containing pre-processed data frames, evaluation schemes, algorithm models, and prediction models. In order for the application to load quickly, it was necessary to save these files so they could be loaded instantly. Any recommendation model that requires the model be run again with a user input vector is done so by adding the user rating matrix to the existing algorithm models. All 5 recommendation functions are very similar, and consist of the following steps:

- find the MAL_ID associated with all five user inputted anime.
- create a dataframe containing the MAL_ID and the rating for that specific anime.
- Use rbind() function to combine the user ratings data frame with the original sampled dataframe of 1000 users.
- Transform the dataframe into a realRatingMatrix.
- If the method is UBCF or Hybrid, create a new model using Recommender() and the combined realRatingMatrix.
- Use predict() function with the first parameter being the model to evaluate with, and the second parameter being the first column of the combined realRatingMatrix. The first column is always the user ratings column.
- Pass the result of the above step to the getAnimeNames() function, which finds the English name associated with the MAL_ID of the prediction, and returns it as a dataframe.

6.2 Examples of Shiny Application

In order to demonstrate the functionality of our Shiny recommender system, we have included a screenshot of the application running with the code in the background, which is shown in Figure 20. The locally hosted address in the console of R matches the address of the application, meaning that it is running on my computer.

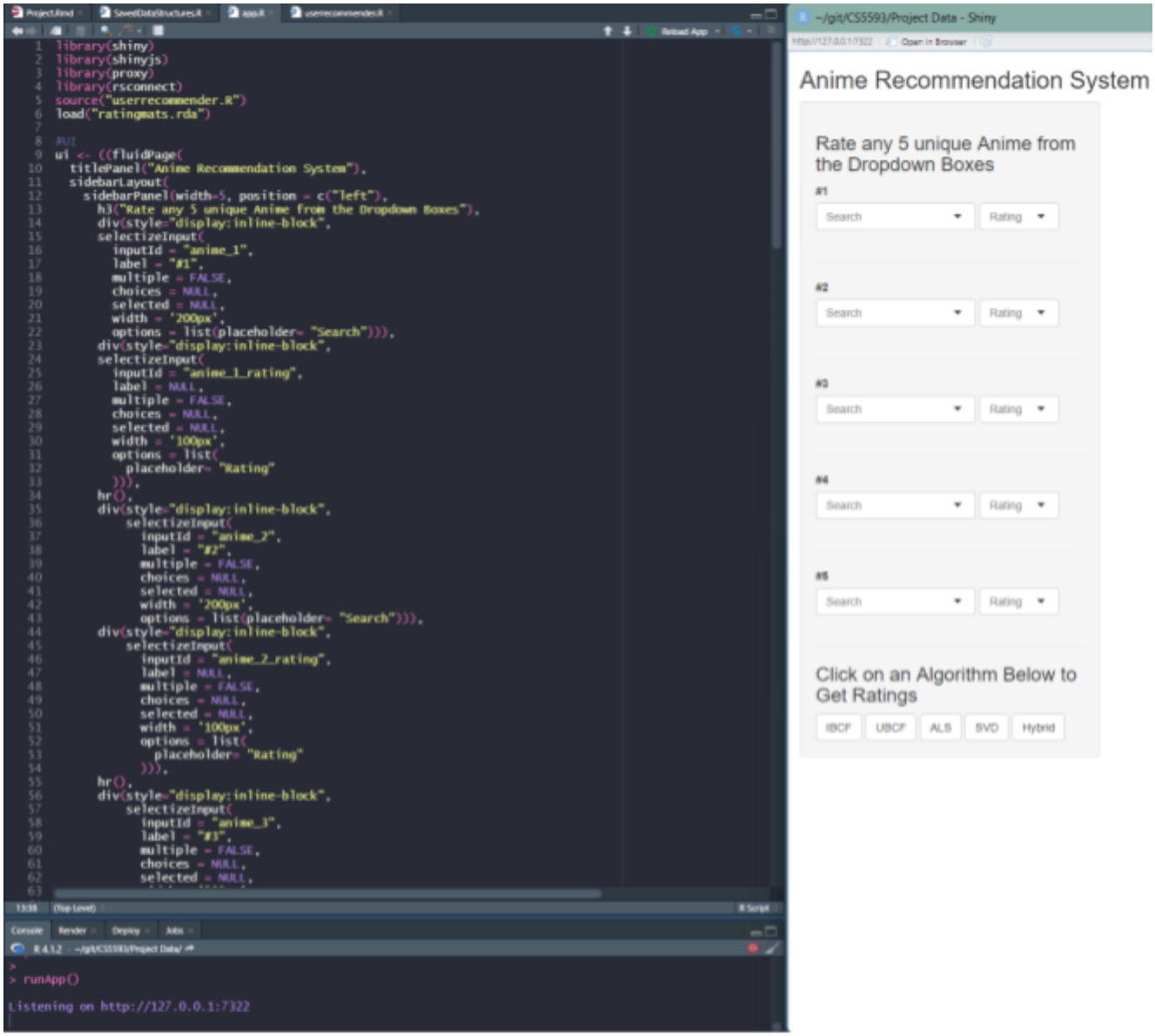

Figure 20: Shiny Application Running with Background Code

Once a user has selected 5 unique anime titles, and provided a rating for each anime, the user must click one of the five buttons below the input data in order to receive recommendations. The list of recommendations is updated even after changes are made to the list of reviewed anime. Figures 21 ,22, 23, 24, and 25 show that our application generates unique recommendations for all 5 algorithms. The slowest performing algorithm by far is the Alternating Least Squares algorithm, which we could not find a way to increase the speed of even after parameter adjusting.

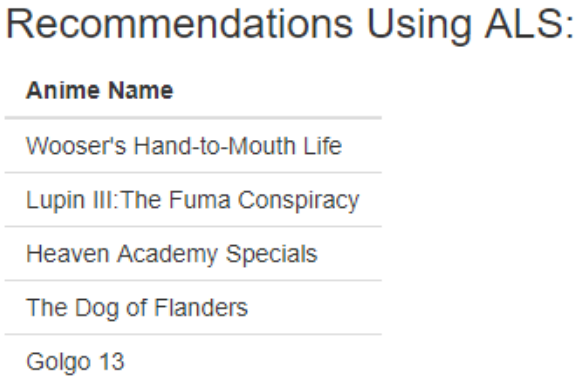

Figure 21: ALS Recommendations

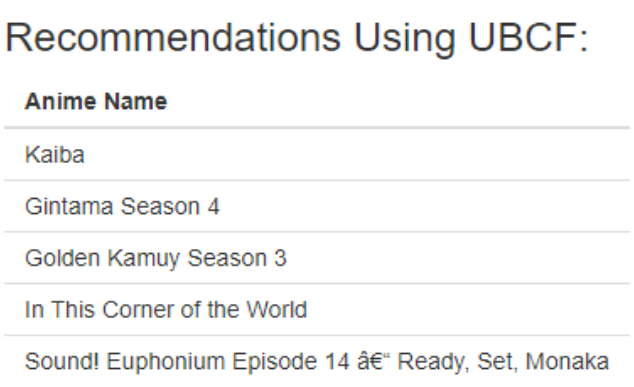

Figure 22: UBCF Recommendations

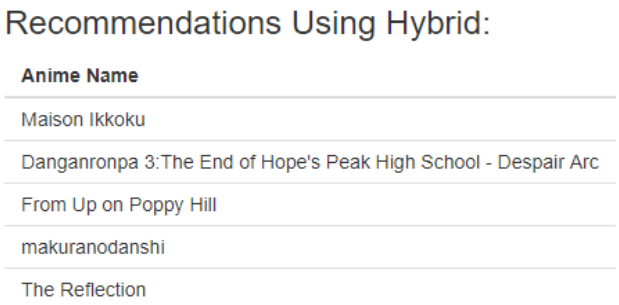

Figure 23: Hybrid Recommendations

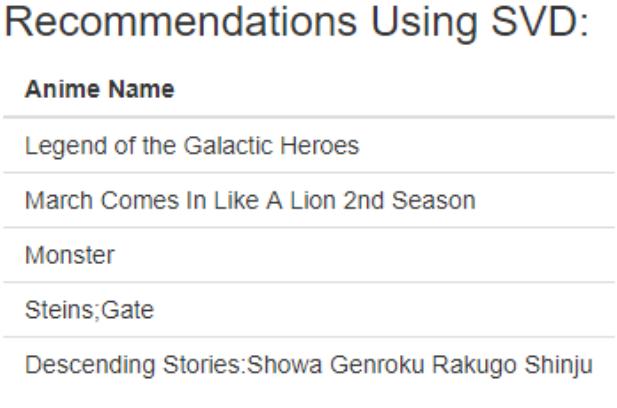

Figure 24: SVD Recommendations

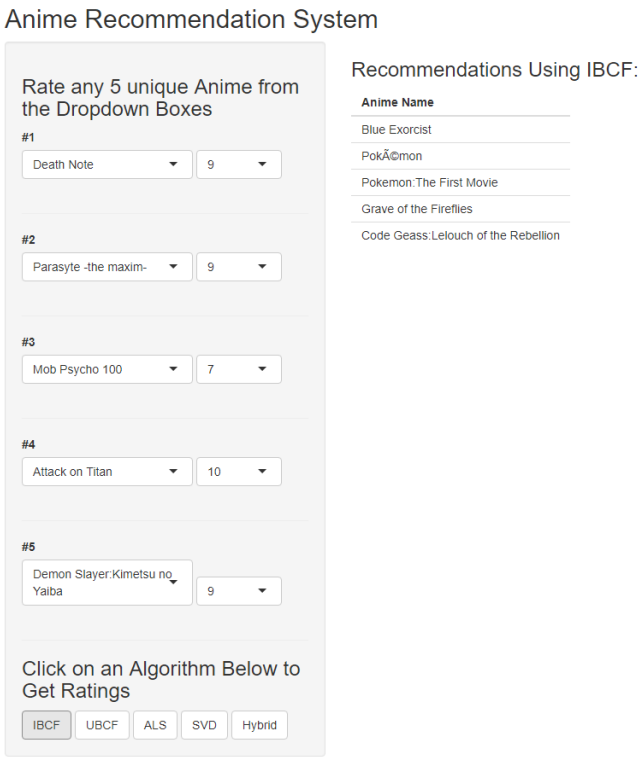

Figure 25: IBCF Recommendations

7. Conclusions and Future Work

Creating an anime recommender system consisted of a lot of moving parts that are dependent upon one another. When a strong recommendation is desired, a large dataset is required. However, we found that most datasets found on the internet include a large amount of data that wasn't relevant to our goals. Once we found our anime database file, we still had to spend a great deal of time performing proper data preprocessing techniques such as filtering and sampling. The Recommenderlab package was a joy to work with, but still took some time to thoroughly understand exactly what every function was used for. The algorithms and prediction models have their limits, but we found that most apart from ALS performed quickly and with little error. We were also happy to see that the recommendations that were provided to us when we used our application were relevant and appropriate to our tastes.

Further improvements could be made on our application, including implementing a popular option which only recommends anime based on a certain level of popularity. This could be done using the popularity ranks that are found in our dataframe or by using the popular algorithm within Recommenderlab. Scraping the data from MyAnimeList and updating the data to the current year would be another improvement that could be made.

8. REFERENCES

[1] Amatriain, Xavier & Jaimes, Alejandro & Oliver, Nuria & Pujol, Josep. (2011). Data Mining Methods for Recommender Systems. 10.1007/978-0-387-85820-3_2.

[2] Baker, K. (2005). Singular value decomposition tutorial. The Ohio State University.

[3] Gosh, S., Nahar, N., Wahab, M. A., Biswas, M., Hossain, M. S., & Andersson, K. (2020, December). Recommendation system for e-commerce using alternating least squares (ALS) on apache spark. In International Conference on Intelligent Computing & Optimization (pp. 880-893). Springer, Cham.

[4] Hahsler, Michael. (2022). recommenderlab: An R Framework for Developing and Testing Recommendation Algorithms. https://doi.org/10.48550/arxiv.2205.12371.

[5] Hu, Yifan & Koren, Yehuda & Volinsky, Chris. (2008). Collaborative Filtering for Implicit Feedback Datasets. Proceedings - IEEE International Conference on Data Mining, ICDM. 263-272. 10.1109/ICDM.2008.22.

[6] Isinkaye, F. O., Folajimi, Y. O., & Ojokoh, B. A. (2015). Recommendation systems: Principles, methods and evaluation. Egyptian Informatics Journal, 16(3), 261–273. https://doi.org/10.1016/j.eij.2015.06.005.

[7] Kirchner, K., Zec, J., & Delibašić, B. (2016). Facilitating data preprocessing by a generic framework: A proposal for clustering. Artificial Intelligence Review, 45(3), 271–297. https://doi.org/10.1007/s10462-015-9446-6.

[8] Liu, Z., & Zhang, A. (2020). A Survey on Sampling and Profiling over Big Data (Technical Report) (arXiv:2005.05079). arXiv. http://arxiv.org/abs/2005.05079

[9] Marcus, A. “Principles of Effective Visual Communication for Graphical User Interface Design.” 1995, pp. 425–441., https://doi.org/["10.1016/B978-0-08-051574-8.50044-3", "info:doi/10.1016/B978-0-08-051574-8.50044-3"]. https://www.diliaranasirova.com/assets/PSYC579/pdfs/05.2-Marcus.pdf.

[10] MyAnimeList.net - Anime and Manga Database and Community. *MyAnimeList.net*. Retrieved July 7, 2022 from https://myanimelist.net/

[11] Shovan Biswas. 2022. RPubs - Movie Recommender System with Large Dataset. Retrieved July 7, 2022 from https://rpubs.com/ShovanBiswas/708070

[12] Taras Hnot. 2016. Recommender Systems Comparison. Retrieved July 7, 2022 from https://rstudio-pubs-static.s3.amazonaws.com/178921_b24a5c82d6e145889c7de06190fc0baf.html

[13] Valdivieso, Hernan. (2021: Aug.) “Anime Recommendation Database 2020.” Version 1. Retrieved 27 May 2022 from https://www.kaggle.com/datasets/hernan4444/anime-recommendation-database-2020?resource=download.

[14] Zhengzheng Xian, Qiliang Li, Gai Li, Lei Li, "New Collaborative Filtering Algorithms Based on SVD++ and Differential Privacy", Mathematical Problems in Engineering, vol. 2017, Article ID 1975719, 14 pages, 2017. https://doi.org/10.1155/2017/1975719.