Analysis of Beer Reviews and Recommendation of Favorites

Project Final Report

ALEXANDER CROLL

Data Mining, IT721A

Data Science, University of Skövde

Alan Said, Niclas Ståhl

October 8, 2017

Table of Contents

Table of Figures

1. Introduction

The following section will briefly describe the project assignment and then go into some detail about the chosen project.

1.1 Assignment Description

As part of the Data Mining course, students are set to carry out an individual data mining project as the final assignment in the course. Students are encouraged to make their own decisions on what to focus as long as the project will focus on some sort of recommender system.

Within this area, potential topics could include applying a known algorithm on a dataset to make recommendations, developing a new recommendation algorithm, using an existing framework to implement a new algorithm or some other possibility related to the general area.

The assignment itself consists of a practical programming part, a presentation of results as well as this written report.

The assignment in my case will focus on using an existing recommendation library to apply the item-based collaborative filtering (IBCF) method on an existing data set in order to retrieve recommendations. Further details of the project are described in the following section.

1.2 Project Details

The project's overall goal is to develop a beer recommender application which can take user inputs via some graphical user interface and give out recommendations to the user based on their input.

First, the project aims to briefly analyze beer ratings which are provided by an extensive beer review dataset and then continue to provide recommendations in terms of which beers might also be favorable for a user based on other beers that they enjoy.

The primary focus for the recommendation task will be on implementing collaborative filtering, more specifically item-based collaborative filtering, where based on a given beer (or several beers) a number of other beers that may be suitable for a user should be suggested.

The project will be implemented in R programming environment, utilizing the existing recommendation framework *recommenderlab*[1] that supports ready-to-be-used collaborative filtering algorithms to predict missing ratings and provide top-N recommendation lists.

[1] Hahsler, M. (2017)

The recommendation application is developed as an R Shiny application which provides the tools for a graphical user interface as well as the possibility to host the application online or respectively an easy way for other users to run the application on their machines. Additional details about the implementation, including the data used, the recommendation library and the R shiny application are described in the following chapter.

This paper is complementary to the R script files that make up the Shiny application as well as some pre-computed RData objects, the data preparation and testing code.

- server.R
- ui.R
- prep.R
- testing.R
- RData objects
- www folder including a bootstrap.css file
- data folder including original review data set

2. Implementation

In this chapter, the implementation will be described in more detail in regards to what data has been used, what functionality the library *recommenderlab* offers, how the application was implemented in R Shiny and how the recommender system was tested. Any choices included in this process will be motivated.

2.1 Data

The data used in this project consists of beer reviews from beeradvocate.com which is available at Data.World.[2]

The dataset includes user ratings of various beers over a period of 10 years which accounts to about 1.5 million records. The reviews contain information about the user, brewing company, beer type, name of the beer and the subjective user ratings as overall rating and some sub-categories as shown below:

- brewery_id: unique identifier for breweries
- brewery_name: name of the breweries
- review_time: time at which review occurred in UNIX timestamp format
- review_overall: overall review score on scale 1-5
- review_aroma: review score for category aroma on scale 1-5
- review_appearance: review score for category appearance on scale 1-5
- review_profilename: name of user who submitted review
- beer_style: name of the beer style
- review_palate: review score for category palate on scale 1-5
- review_taste: review score for category taste on scale 1-5
- beer_name: name of the beer
- beer_abv: alcohol percentage of beer
- beer_beerid: unique identifier for beers

By briefly looking at the available data, some key findings can be identified:

- there are more than 1.5 million beer ratings
- ratings include 5,743 breweries
- breweries account for 104 different beer styles
- 56,857 different beers were rated
- Reviews in sub-categories of aroma, appearance, palate and taste
- Some beers were rated multiple times by the same user
- There are ratings that do not belong to any specific user

[2] Data.World (2016)

The below graphic displays the distribution of ratings across the scale of 1-5. As expected, many of the ratings wrap around the mean while the extremes of 1 and 5 were used less frequently as ratings.

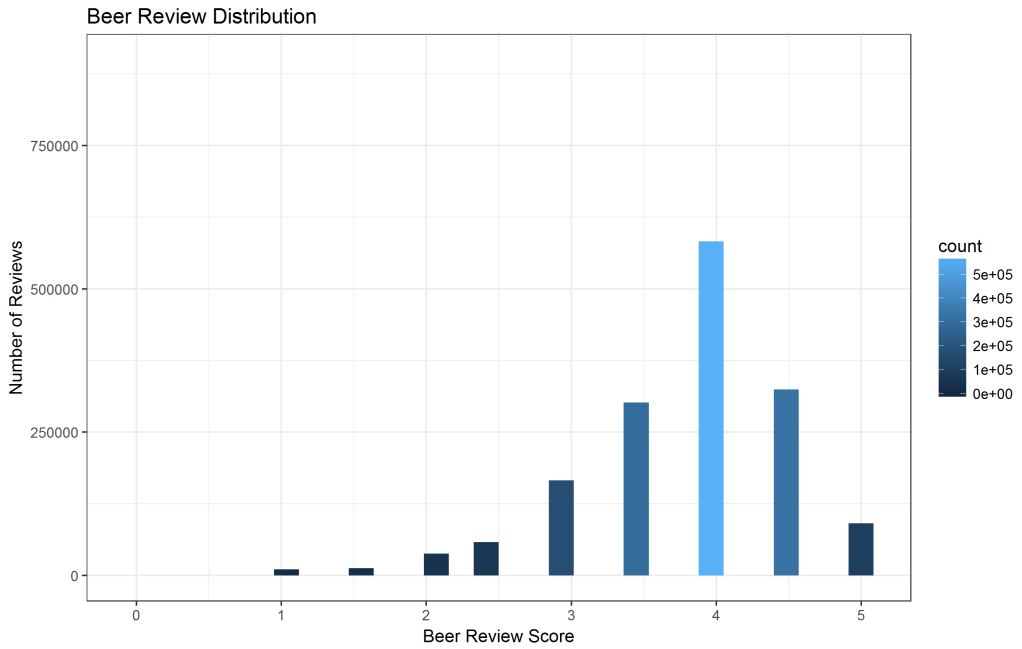

Figure 1: Distribution of beer ratings across scale 1-5

In order to facilitate retrieving recommendations from the R Shiny application, the raw data was processed beforehand. By pruning some of the data and saving data objects to RData objects, the application will load and produce results faster. Also, preprocessing can eliminate some complexities from the data. The following steps were carried out:

- Pruning the raw data to remove any reviews without reviewer name and also remove duplicate ratings (for ease of recommendations, no time series included)
- From all reviews, create lists for beers, breweries and beer styles which occurred at least 100 times (respectively) in the review data. These lists are provided as input lists for the drop-down lists in the R Shiny application.
- Creating a beer rating matrix where only beers with at least 100 reviews are included. This step eliminates “uncertain” reviews where only few users had an opinion about a beer.
- Creating a recommender object based on the previous beer rating matrix with the method item-based collaborative filtering.

All the above steps reduce the time for data processing at runtime. Particularly the steps that include using the *recommenderlab* library can take a significant amount of time given the amount of data we are dealing with.

From the steps above, RData objects were saved for the pruned reviews, the lists for beers, breweries and beer styles as well as a recommender object. These objects are loaded at runtime. The data preprocessing can be found in the file *prep.R*.

2.2 Recommenderlab Library

The *recommenderlab* library provides infrastructure for developing and testing recommender algorithms and was developed by Michael Hasler. It particularly focuses on collaborative filtering algorithms (e.g. Item-based, User-based) and includes functionality (among others) to

- prepare existing rating data and bring it into shape of a rating matrix
- create recommender objects of different types (IBCF, UBCF, Popular, etc.)
- predict ratings for unrated items by taking into account what has been rated
- retrieve top N lists with the best possible predictions
- evaluate a recommender object using different evaluation measures

In the following section, the approach to implement the beer recommender system will be described.

In order to get from pre-processed data to a recommender object, a few steps have to be implemented.

First, the data needs to brought into the correct format. For this purpose the data frame with all ratings needs to be converted from data frame to matrix format. The transformation is illustrated in below figures.

User	Beer	Rating
User1	Beer1	4
User1	Beer2	3
User2	Beer1	1
User2	Beer3	5
User3	Beer2	4
User3	Beer3	3
User4	Beer1	3
User4	Beer2	5
User4	Beer3	4

Figure 2: Beer Ratings in original data frame format

	Beer1	Beer2	Beer3
User1	4	3	NaN
User2	1	NaN	5
User3	NaN	4	3
User4	3	5	4

Figure 3: Beer Ratings in matrix format

Next, this matrix needs to be converted into an object of the recommenderlab class *realRatingMatrix*. When creating a recommender object, the library will only work with this specific matrix class.

After this, we are ready to train the recommender object using the item-based collaborative filtering method. As mentioned before, the library offers several different methods for recommender systems. Two popular methods of collaborative filtering algorithms are item- and user-based (UBCF). As the names suggest, IBCF gives recommendations based on similar items to the ones that the user rated and liked while UBCF gives recommendations based on similar users (meaning users that previously provided ratings are similar to the current user trying to get recommendations).

While the idea of recommendations between these two methods is similar, they differ in implementation. IBCF provides recommendations based on a similarity matrix which can be

pre-computed. UBCF on the other hand focuses on finding similar users to the current user (that wants recommendations), also called k nearest neighbors, which have to be found at runtime. This is an important factor to think about because these approaches will influence the application behavior. This application is supposed to provide recommendations to users on the fly and as fast as possible. A web application should not take too much time and have users wait in front of the screen.

Resources show that computation increase with the number of users and items in the rating matrix, for instance a 5,000 x 1,000 user/item rating matrix takes 75 seconds for IBCF training as compared to only 1 second for UBCF training. However, when providing recommendations ICBF (1 sec.) is much faster than UBCF (40 sec.) with almost no delay.[3] In my scenario with an approximately 30,000 x 3,000 user/item rating matrix, training the recommender system takes >20 minutes, but will be rewarded at runtime.

Since IBCF is pre-calculated, it obviously takes more time than UBCF in training, but this can be an acceptable downfall compared to the saved time in the R Shiny application. The high data volume of this beer recommender application makes it hard to go with UBCF which is why IBCF was the preferred choice for practical reasons.

The *recommenderlab* library now creates a recommender object which consists of a similarity matrix. By looking back at the previous example, we can simulate what is going on behind the scenes. By default, *recommenderlab* uses cosine similarity for IBCF. Cosine similarity is defined as follows[4]:

$$Similarity = \cos\theta = \frac{A * B}{\|A\| * \|B\|} = \frac{\sum_{i=1}^{n} A_i \; x \; B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^{\,2}} * \sqrt{\sum_{i=1}^{n} (B_i)^{\,2}}}$$

Hence, the cosine similarity between the three vectors v1 (beer1), v2 (beer2) and v3 (beer3) can be calculated as shown in the following section:

Cosine similarity between beers 1 and 2, represented by vectors v1 and v2:

v1 = 4 User1 + 3 User4

v2 = 3 User1 + 5 User4

cos(v1, v2) = (4*3 + 3*5) / sqrt((16+9) * (9+25)) = 27 / sqrt(850) = 0.926

[3] Big Data Doctor (2014)

[4] Savev, S. (2015)

Cosine similarity between beers 1 and 3, represented by vectors v1 and v3:

v1 = 1 User2 + 3 User4

v3 = 5 User2 + 4 User4

cos(v1, v3) = (1*5 + 3*4) / sqrt((1+9) * (25+16)) = 17 / sqrt(410) = 0.84

Cosine similarity between beers 2 and 3, represented by vectors v2 and v3:

v2 = 4 User3 + 5 User4

v3 = 3 User3 + 4 User4

cos(v2, v3) = (4*3 + 5*4) / sqrt((16+25) * (9+16)) = 32 / sqrt(1025) = 1

From this, the similarity matrix can be constructed:

	Beer1	**Beer2**	**Beer3**
Beer1	1	0.926	0.84
Beer2	0.926	1	1
Beer3	0.84	1	1

Figure 4: Similarity Matrix

With this information, missing ratings for existing users could be predicted. For this application scenario however, we would like to retrieve beer recommendations for users that visit the web application. Given the above example, let's pretend there is a new user who has the following ratings:

	Beer1	**Beer2**	**Beer3**
UserX	2	4.5	NaN

Figure 5: Ratings from web application user

Given the similarity matrix and the ratings from the new user X, missing ratings can now be predicted to retrieve beer recommendations.

Rating_Beer3 = (2*0.84 + 4.5*1) / (0.84 + 1) = 4,34

Obviously, this example lacks some depth, because there are only 3 beers included. But image there were 10 beers, in which case from only 2 or 3 existing ratings the remaining beers could be rated as well in the same fashion and from those remaining beers the ones with the best predicted ratings would be the top N recommendations.

2.3 R Shiny Application

Shiny is an open-source library in R which provides users with a framework to develop interactive web applications in R. The Shiny package makes it particularly easy to do so, because users do not need to have any specific web development skills because Shiny provides pre-built objects for the user interface. Users can simply use their R scripts and turn them into reactive applications.

Another great feature of R Shiny is deployment: Users can either provide their R code to others who can then simply run this code on their machines and create a local web application or the entire Shiny application can be hosted on the web to provide access to everyone. The easiest way is using the free web service *shinyapps.io*.[5]

The Shiny application that was created for this project is available at https://wasencroll.shinyapps.io/beer-recommender/.

The approach to my application was using the different ratings to provide as much information to the users as possible. Therefore, I decided to not only include beers in my application, but also provide reviews/ratings from breweries and beer styles. However, the focus lies on beer recommendations.

Each of these three categories is divided into the possibility to retrieve general top recommendations or categorical top recommendations. The general top recommendations will simply give the top N beers/breweries/beer styles while the categorical top recommendations also take into account a categorical parameter, e.g. top N beers from a particular beer style.

The following table gives an overview:

Beer Recommendations	**Brewery Recommendations**	**Beer Style Recommendations**
Top N beers	Top N breweries	Top N beer styles
Top N beers from a specific brewery	Top N breweries from a specific beer style	Top N beer styles from a specific brewery
Top N beers from a specific beer style		

[5] Shiny (2017)

Figure 6 illustrates the general outline of the web application and the just described overall functionality with three tabs for beers/breweries/beer styles and the options of general or categorical top N results.

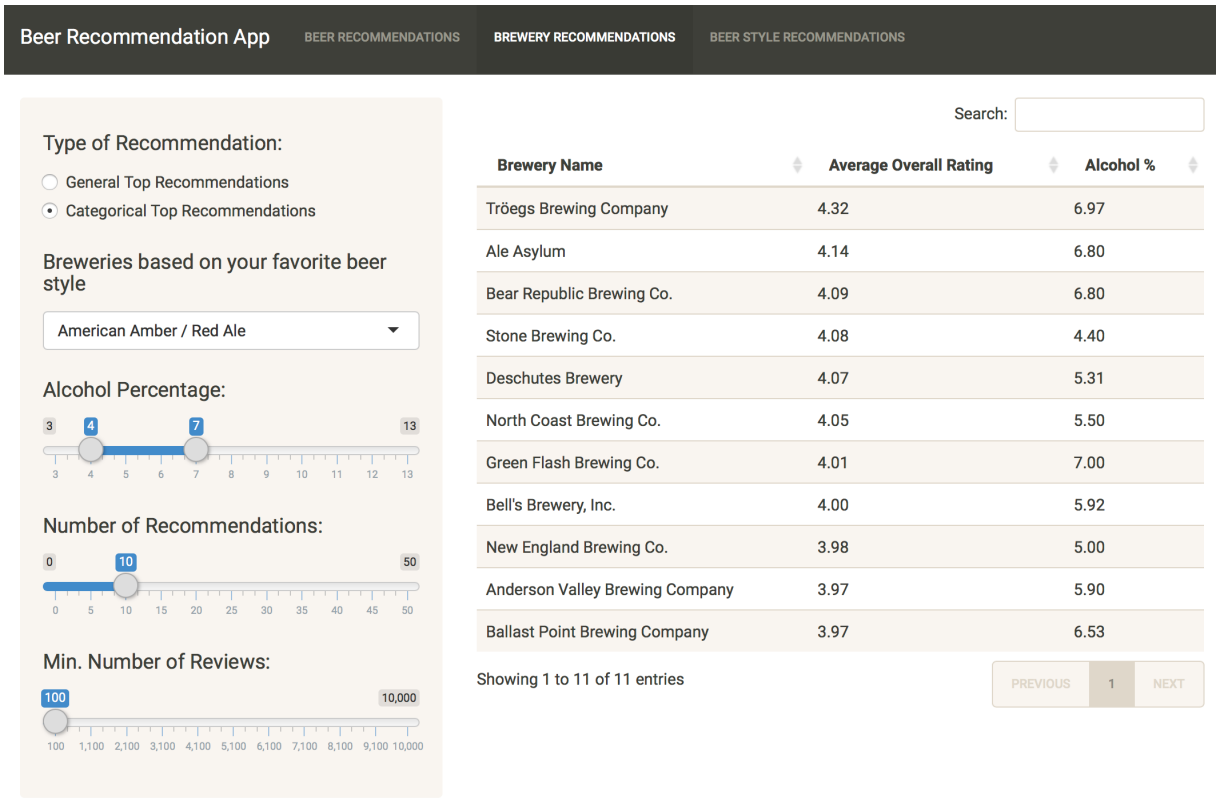

Figure 6: Beer Recommendation App (Overview)

As you can also see, the users can tweak their results by changing additional input parameters such as the alcohol percentage in the beer (abv), the number of recommendations they would like to receive as well as the minimum number of reviews a beer/brewery/beer style should have received in order to be considered in the results. This allows users to produce results that are in synch with their individual needs.

Here is an overview of the application's functionality:

- Different navigational tabs for recommendations of beers, breweries or beer styles
- A choice of general or categorical recommendations
- Drop-down inputs are fully searchable
- Any input is reactive, meaning that the results change upon changing user inputs
- Results are sorted by rating from high to low, this can be changed however
- In case extensive result lists are produced, users can search these as well

Even though popular recommendations can also be produced using the *recommenderlab* library and are therefore considered some kind of recommendation, simply providing popular recommendations with your own filtering implementation is not enough for a recommender system. Thus, the application includes the most important feature: individual recommendations based on a user's beer ratings. These individual recommendations are implemented using the *recommenderlab* library as described in the previous chapter. Below is a screen grab from the application which illustrates how it works in practice.

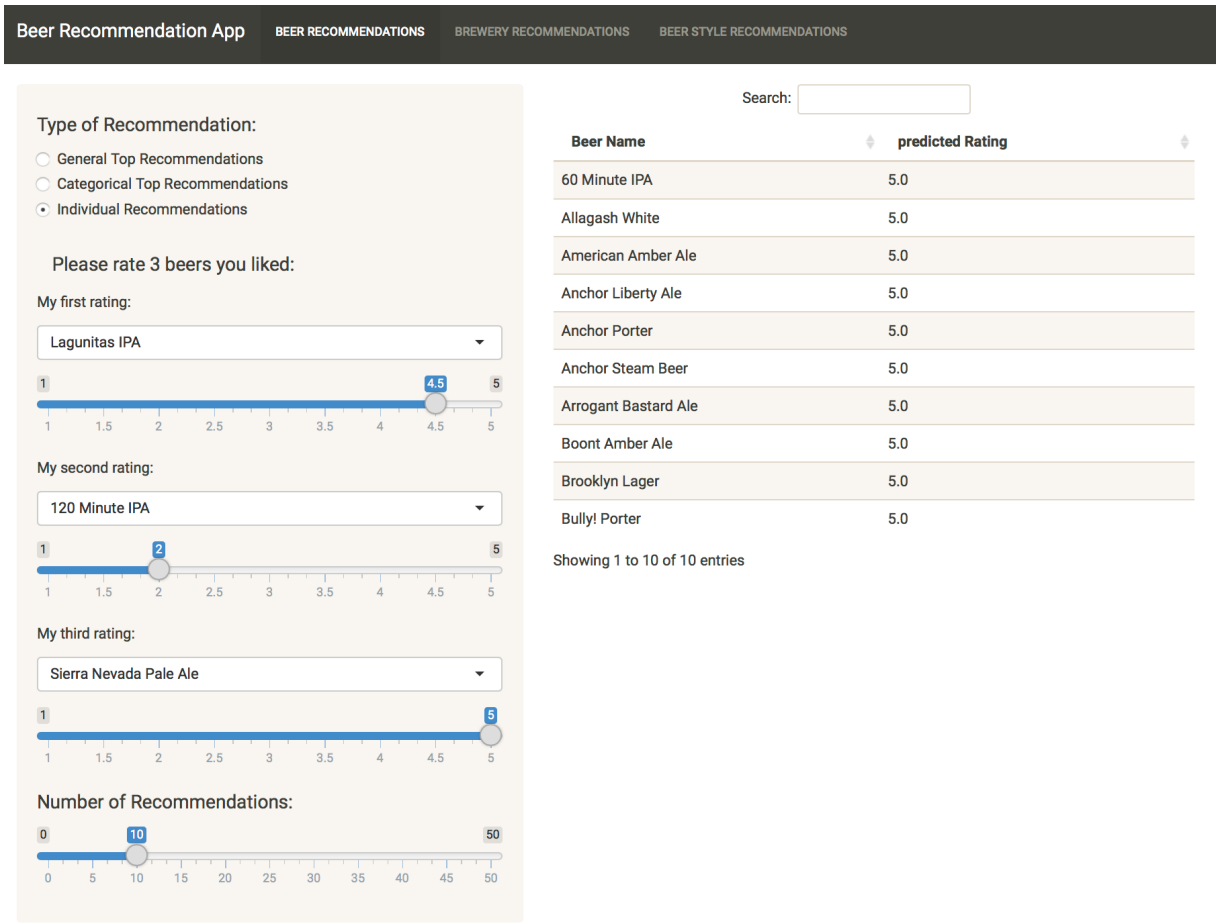

Figure 7: Beer recommendations based on IBCF

When using the functionality for individual beer recommendations, the user inputs change as compared to the other application functionality. The user now has to rate three beers that they have previously tasted and developed some opinion about. For this purpose, three separate drop-down lists are available from which users can pick their beers. Again, these are searchable. Once beers are selected, users can choose their ratings of each beer with the usage of input sliders on a scale from 1-5 (with 0.5 steps). Further, they can change the number of recommendations that will be made available to them with the input slider at the bottom.

Once these inputs are available to the application, it will create a temporary and anonymous user session that creates a user profile with their ratings in the same format as all other

available ratings. This enables the application to utilize *recommenderlab*'s functions by using the existing recommender object and project top N lists as recommendations.

The input objects with which a user provides their beer preferences are reactive, meaning that users can change their beers and corresponding ratings in case they would like to see new recommendations based on different beers or simply tweak around with the inputs to see if this changes their recommendations. However, recommendations are always provided based on the three currently rated beers. Former inputs are not stored and included because usability of the application should be kept easy: saving older ratings in the background is not transparent and could confuse interpretation of recommendation results. Given the high number of beers (>3000) included in the rating matrix, displaying the previously provided ratings somewhere would be excessive.

2.4 Testing

In order to test the recommender systems, there are a few measures that can help determine performance:

- Receiver-Operator-Curve (ROC), which illustrates the ration between true positives (sensitivity) and false positives (specificity)
- Precision/recall trade-off, where high precision illustrates a low false positive rate and high recall stands for a low false negative rate
- Root Mean Square Error (RMSE), which is an indicator for prediction accuracy

The ROC graph displays the true positive rating (y-axis) against the false positive rating (x-axis) of different instances of a learned recommender system. Generally, the curve should follow as closely to the left and upper boarder of the ROC space, forming a curve that moves up and right, because a good recommender system has good true positive ratings, but as few as false positive ratings as possible.[6]

In Figure 8, ROC lines for an IBCF recommender model using the beer review data are illustrated. These were fitted with cross-validation and varying parameters for k-nearest items as well as different values for the top N list number of recommendations. These variations are implemented in order to see which instance represents the best model to use within the Shiny application. The lines represent individual models with different values for k (1,5,10,30,50,75,100,125,150), while the different points on each line show true/false positive

[6] Tape, T.G. (w.y.)

ratings for different numbers of top N list recommendations. This combination provides insight into which model shows the best ROC measurements and also how many recommendations can be retrieved until retrieving weaker results.

In this case, the red curve for k = 5 obviously looks the best because it is the one that moves along left and upper boarder the most. However, when retrieving top N recommendations up to n = 15 or even higher, the green curve, i.e. the model with k = 10 can be a valid choice as well since the ROC curves converge here.

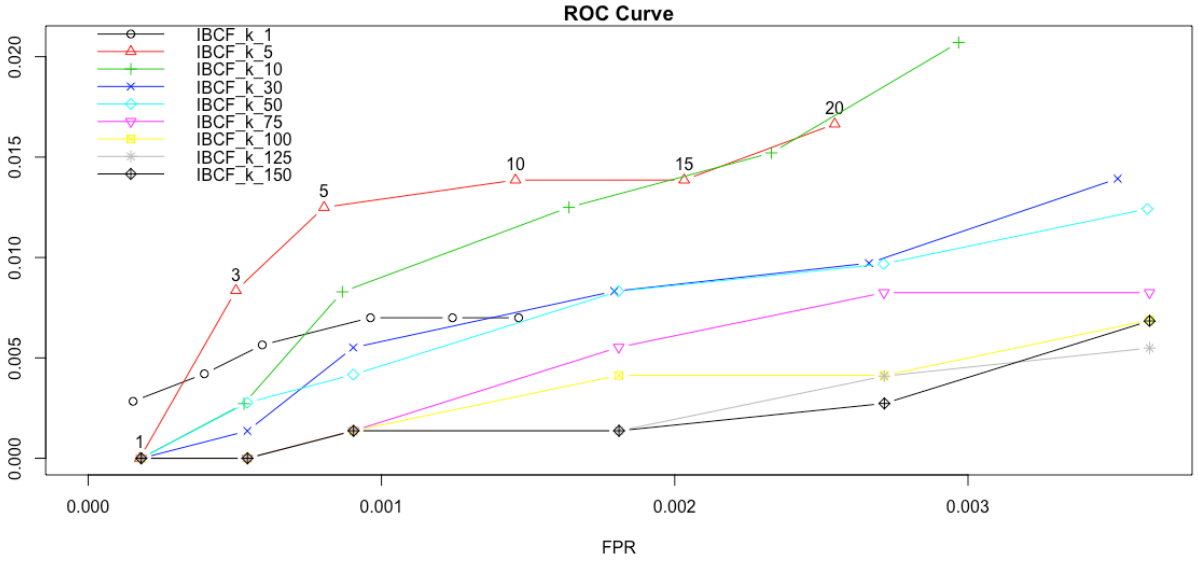

Figure 8: ROC measurements for different IBCF models (k nearest items vary)

As we move to the next measure, the trade-off between precision and recall, we are now looking for models that have high results for both precision (y-axis) and recall (x-axis) as this indicates that the recommender system delivers accurate results. Achieving high results in both measures means that there is both a low false positive/negative rate, which is ultimately what is desired: few false predictions.[7]

In Figure 9, the same setup as before was chosen using cross-validation and varying parameters for k-nearest items as well as top N list number of recommendations.

Comparing the results to the results and indications in Figure 8, once again we see that a model based on k = 5 shows good results and it is only once n for top N list recommendations increases past n = 10 that the other model of k = 10 can also provide similar results. Also, the curve of the IBCF model with k = 30 is not far from these results when it comes to top N predictions with higher n.

[7] scikit learn (2017)

The other models can be ignored because they either return many results with very few correct results or just the opposite, few results which are mostly correct. In the latter case, the model with k = 1 would actually be interesting, but only if users are looking to retrieve a single recommendation in which event that recommender model would actually present a higher precision as the others.

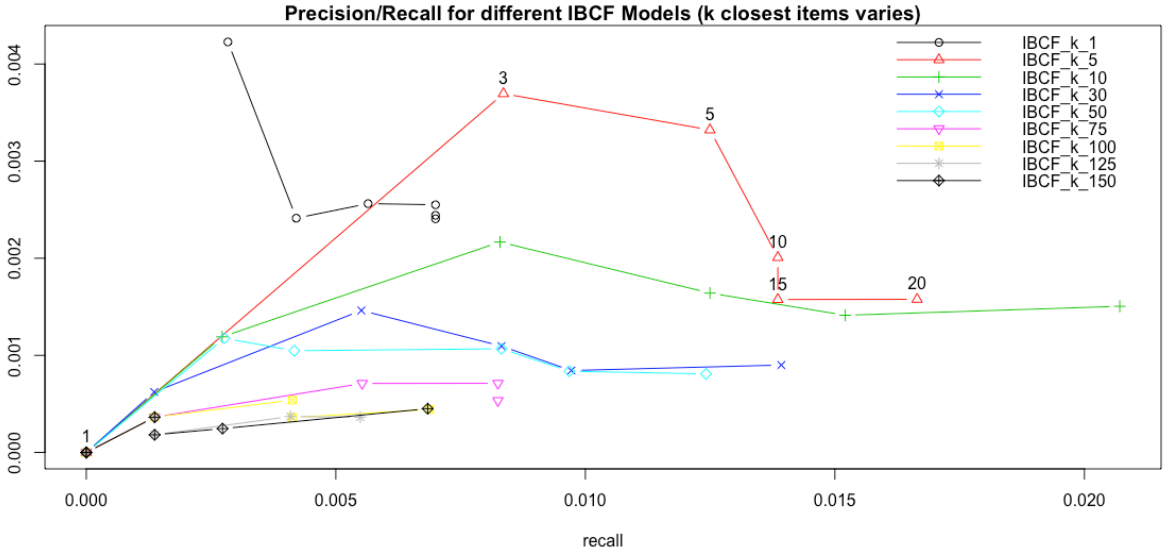

Figure 9: Precision/Recall for different IBCF models (k nearest items vary)

The third measure, Root Mean Square Error (RMSE) provides indication on prediction accuracy. The lower the value of RMSE, the lower the mean deviation of predicted from actual ratings. Again, the same setup was used for this test as can be seen in Figure 10:

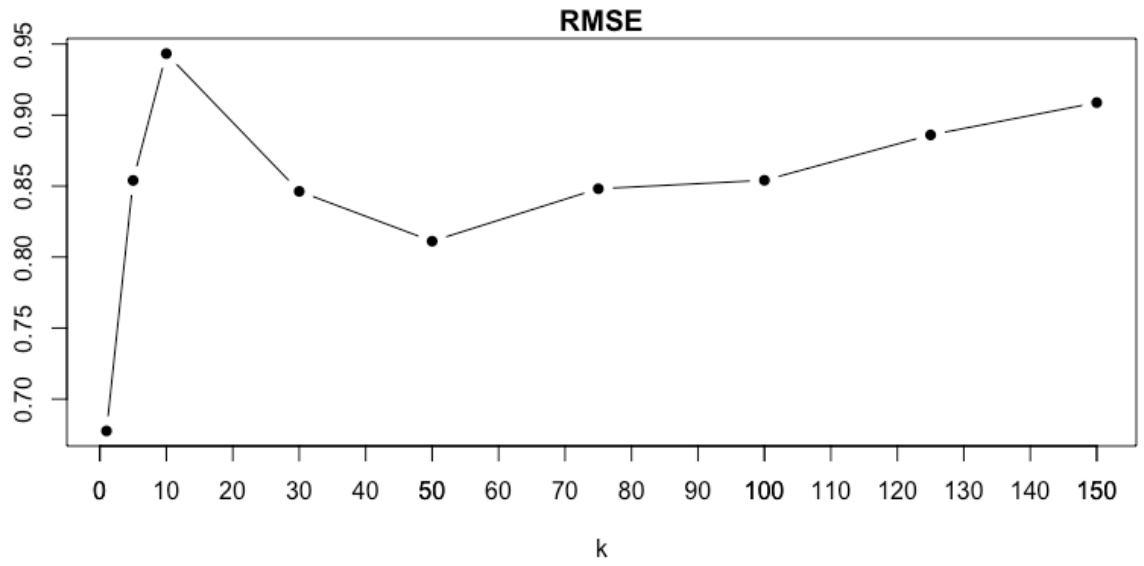

Figure 10: RMSE as k nearest items vary for IBCF models

The best RMSE values exist for models with k = 5 and with k = 30, 50 or 70. The high RMSE for k = 10 is an outlier and does not support findings from ROC and precision/recall. Generally, a model with a k nearest items value of k = 5 promises the best results according to testing given that only a few recommendations are required. Once 15 or even more recommendations are required, other models can perform in a similar fashion.

The second possibility to test the recommender system is by personally engaging in the application. Figure 7 shows how the application can be used to retrieve beer recommendations and the ratings used there are based on the author's preferences and experiences.

When looking at the provided recommendations and comparing these to additional beer drinking experiences, the recommender system seems to work well because all except two beers in the provided top 10 list are highly favorable by the author. In fact, the *60 Minute IPA* from *Dogfish Head Brewery* in Delaware is actually one of the all-time author's favorites. Beers such as *Anchor Liberty Ale* or *Brooklyn Lager* are also favorable from experience

The remaining beers that are unknown to the author by taste were researched on the breweries' websites and appear to lie within the general taste of beers favored by the author as well.

For this personal testing, recommender systems with different values for k were used to see if and how they differ in performance. It became apparent that k = 5 is too small of a value to learn a recommender system with because subjectively, beer recommendations were not as good as for higher values and furthermore, when requesting more than 10 or 15 recommendations, the next best recommendations were recommended with a predicted rating of 2. Hence, the final implementation of the recommendation application has a k-nearest items value of k = 20. This seemed acceptable given the measures from testing and under the assumption that users will not just retrieve one single recommendation, the application will perform well with 10-15 or even more recommendations.

3. Results

Obviously, recommendation results are highly dependent on the individual users who use the web application, but here are some general findings and results:

- The best rated beers from the entire pool of beers include beers such as *Citra DIPA* or *Heady Topper* and generally, the list seems to include less known beers
- Once using the minimum number of review input to require more reviews of a beer, more and more of these 'exotic' beers disappear and more well-known beers appear in the output. These are still rated above 4 on the scale. (includes *Sierra Nevada* beers, *Samuel Adams*)

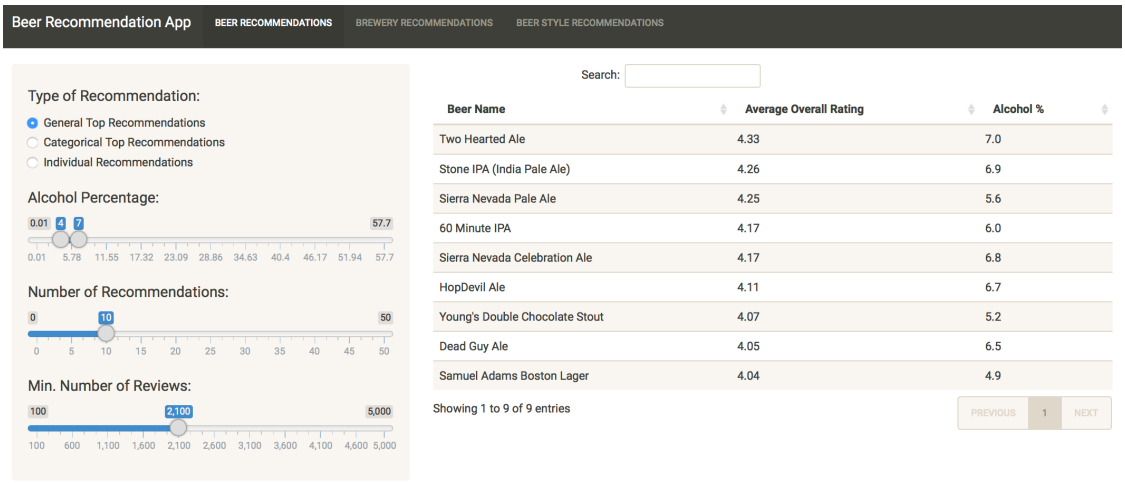

Figure 11: Top rated beers with 2000+ reviews

- The ratings include a lot of craft beers (e.g. from the US) that are usually well-rated
- As author from Germany, there are no German beers ranked really high, neither are many breweries (exception *Weihenstephan*)
- Particular breweries that sell mass product in stores where I come from are ranked at below 4

Overall, the application performs very well and provides fast results – no matter which functionality is used. The user interface makes it easy for users to navigate through the application.

Recommendation results are a good fit to personal taste and can give some ideas of which beers to taste next.

4. Conclusion

Recommender systems are a great way to engage users and provide them with useful (or less useful but still fun) information about things that they might like. Creating basic, but interactive versions of them is not too complex when looking out for a few things that are important. In my case, here are some of the issues I had to deal with in order to complete this project:

- Acquaintance of new frameworks: In my case, there were two new frameworks at a time – R Shiny and recommenderlab. It took me a while to get used to the concept of R Shiny, which ones understood is a great framework to create beautiful web applications. Since it is widely used, there are plenty of resources to get started. For recommenderlab it was a little different, because while there is some general documentation available, I obviously had to look out for my particular application scenario. For instance, there were some issues with the rating matrix and retrieving recommendations which did not work at first. Also, the reactive nature of the Shiny application makes things a little trickier as opposed to running R script lines manually.
- Dealing with large amounts of data: the beer review data was quite extensive which lead me to pruning it a bit before actually using it for the recommender system. This was definitely helpful because creating matrices and training recommender systems takes a long time with these amounts of data which can be particularly annoying during the development process. For test purposes, using e.g. the first 100 lines of a matrix will always do, but you never know how good your results really are when not including more data. As mentioned before, this was also the reason for using IBCF instead of UBCF, because while it takes time to create the application, it runs much smoother once it is deployed.
- Improving the model over time: in many scenarios, it is important to improve the recommender system over time as reviews change or new beers are added to the database. This is something that was not considered in this project. Since offline data from a file was used, this does not influence the results much because users from the review data cannot change their reviews anyway.

Concluding, a well-working recommender system was created and published for the public to use. The project itself gave me the opportunity to look deeper into the functionality of recommender systems – collaborative filtering in general – and understand what is going on behind the scenes. The application is certainly not perfected yet, but ready to be used in practice to start getting some new beer ideas.

5. References

Big Data Doctor (2014): Measuring the execution time of recommender systems in R, http://bigdata-doctor.com/performance-recommender-systems-in-r/, Date of Retrieval: Oct. 10, 2017

Data.World (2016): Beer Reviews from the Beeradvocate, https://data.world/socialmediadata/beeradvocate, Date of Retrieval: Feb. 25, 2017

Hahsler, M. (2017): Package 'recommenderlab', https://cran.r-project.org/web/packages/recommenderlab/, Date of Retrieval: Oct. 8, 2017

Savev, S. (2015): Cosine Similarity Part 1: The Basics, http://stefansavev.com/blog/cosine-similarity/, Date of Retrieval: Oct. 9, 2017

Scikit Learn (2017): Precision-Recall, http://scikit-learn.org/stable/auto_examples/model_selection/plot_precision_recall.html, Date of Retrieval: Oct. 9, 2017

Shiny (2017): Shiny from R Studio, http://shiny.rstudio.com, Date of Retrieval: Mar 2, 2017

Tape, T.G. (w.y.): Plotting and Interpretating an ROC Curve, http://gim.unmc.edu/dxtests/roc2.htm, Date of Retrieval: Oct. 9, 2017