Capstone 2 Final Report

Github Repositories

Data Cleaning:
https://github.com/elizabeamedalla/Capstone-2-Recommendation-System/blob/master/Data%20Cleaning.ipynb

Data Exploratory Analysis:
https://github.com/elizabeamedalla/Capstone-2-Recommendation-System/blob/master/Data%20Analysis.ipynb

Statistical Inference:
https://github.com/elizabeamedalla/Capstone-2-Recommendation-System/blob/master/Statistical%20Inference.ipynb

Recommendation Engine using Surprise Package from Sci-kit: (*in Google Colab - will have to upload to Github*)
https://colab.research.google.com/drive/1gruJ9ttR35GuHdQX5LOuSSIIQHjpqIMC?usp=sharing

Google Slides Presentation (*will have to upload as PDF*):
https://docs.google.com/presentation/d/1FsR5o3fbvrn_BjwERy9YPy71Z9AIJ841XK91JGEiV_g/edit?usp=sharing

Problem Statement

In today's digital world, recommendation engines are some of the most powerful tools for marketing. A recommender system is an information filtering system comprising a bunch of machine learning algorithms that can predict 'ratings' or 'preferences' a user would give an item. A recommendation engine helps to address the challenge of information overload in the e-commerce space.

A **recommendation engine** can significantly boost revenues, Click-Through Rates (CTRs), conversions, and other essential metrics. It can have positive effects on the user experience, thus translating to higher customer satisfaction and retention.

Amazon is one of the leading e-commerce in the United States and has a heavy focus on data-driven marketing. If you see them testing something and then rolling it out, you can bet it's getting them great results. **Amazon** currently uses item-to-item

collaborative filtering, which scales to massive data sets and produces high-quality **recommendations** in real-time. This type of filtering matches each of the user's purchased and rated items to similar items, then combines those similar items into a **recommendation** list for the user.

This project will be utilizing Amazon data to answer the following questions:

- How will a given user most likely rate a specific product they have not purchased or reviewed before?
- What “new” products might a system recommend to them?

This project will be focusing on different types of collaborative filtering, where we try to group similar users together and use information about the group to make recommendations to the user. This project will present different types of collaborative filtering using the Surprise package from Sci-kit Learn.

Data Cleaning

Data was acquired from http://jmcauley.ucsd.edu/data/amazon/. Since there are multiple categories, we will use the Amazon Beauty Products Reviews Dataset. It contains 371,345 ratings and 32,992 beauty products. Dates of reviews were from 2000 to 2018.

```
MISSING ROWS per COLUMN
title: 0, 0.00%
image: 0, 0.00%
brand: 0, 0.00%
rank: 0, 0.00%
main_cat: 0, 0.00%
asin: 0, 0.00%
description: 0, 0.00%
also_view: 0, 0.00%
also_buy: 0, 0.00%
price: 0, 0.00%
similar_item: 0, 0.00%
details: 0, 0.00%
feature: 0, 0.00%
tech1: 0, 0.00%
date: 0, 0.00%
```

First, the data was extracted from the zip file. From the zip file, there are two dataframes: one dataframe with all the reviews and one dataframe for just the product information. These were merged together into one big dataframe. This one big dataframe consists of 371345 rows and 26 columns. Second, we filled the empty rows in each column with NaN values. Next, we drop columns that we will never use in this project. Lastly, we changed the datatype of some columns. The date column should be in DateTime format and numbers should be in integer like the rating column.

This is what the clean dataframe looks like. We would have to change the data types later on to minimize the memory of the dataframe.

	rating	verified	review_time	reviewer_id	product_id	reviewer_name	reviewer_text	title	brand
0	5	False	2000-06-03	A2XMFX1BR0IJFJ	0061073717	Jonathan Reed (jonathan.reed2@virgin.net)	This calender is brilliant and has plenty of g...	Workout Headphones by Arena Essentials	HarperCollins
1	5	False	2000-05-06	ATKPYXA8XFKGJ	0061073717	Gwen Bates	This calender really is great. In addition to...	Workout Headphones by Arena Essentials	HarperCollins
2	1	True	2015-02-19	A1V6B6TNIC10QE	0143026860	theodore j bigham	great	Black Diamond	Swedish Beauty

Data Exploration

For data visualization, we will be using Bokeh. Bokeh is a Python library for interactive visualization that targets web browsers for representation. This section will show the basics of Bokeh and how to work with it. We have a couple of graphs that are too big that if we use Bokeh to plot it, it will exceed the RAM that Jupyter can support. For that reason, we will be using the Matplotlib for the visualization of larger memory graphs.

The initial approach for data exploration is to check for unique values. Between reviewer_id and reviewer_name, the reviewer_id column is more unique than the reviewer_name column. We can also inspect the unique values of the entire dataframe.

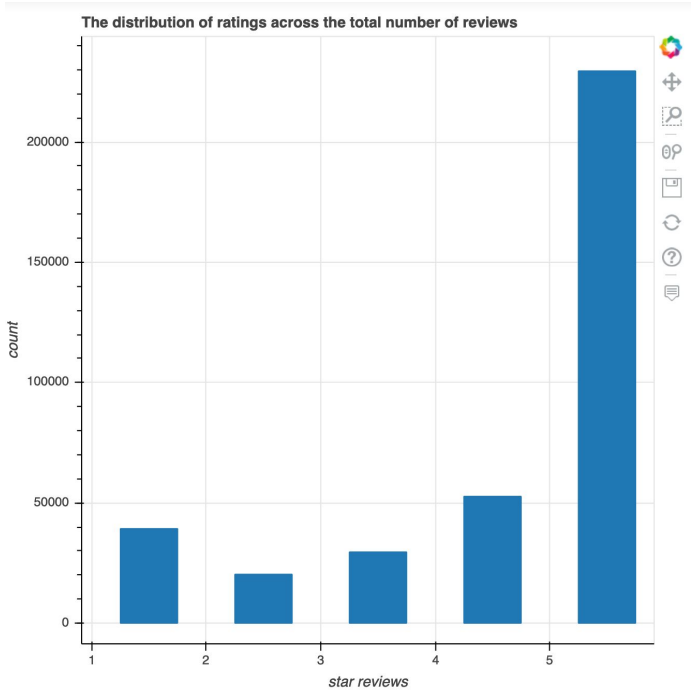
```
Reviews are taken from: 2000-06-03 00:00:00 to 2018-10-02 00:00:00

Number of reviews: 371,345
Number of customers: 324,038
Number of unique products: 32,586
Number of reviewer name: 221,127
Number of unique titles of products: 32,392
```

Customers were identified from the reviewer_id column. The number of reviews and customers are not the same due to some reviewers having multiple review entries. Customers and reviewer names are not the same because some reviewers prefer to be anonymous. Anonymous reviewers were collectively named as Amazon Customers and Kindle Customers, and these two names made up the majority of the names in the dataset.

```
df.reviewer_name.value_counts()
```

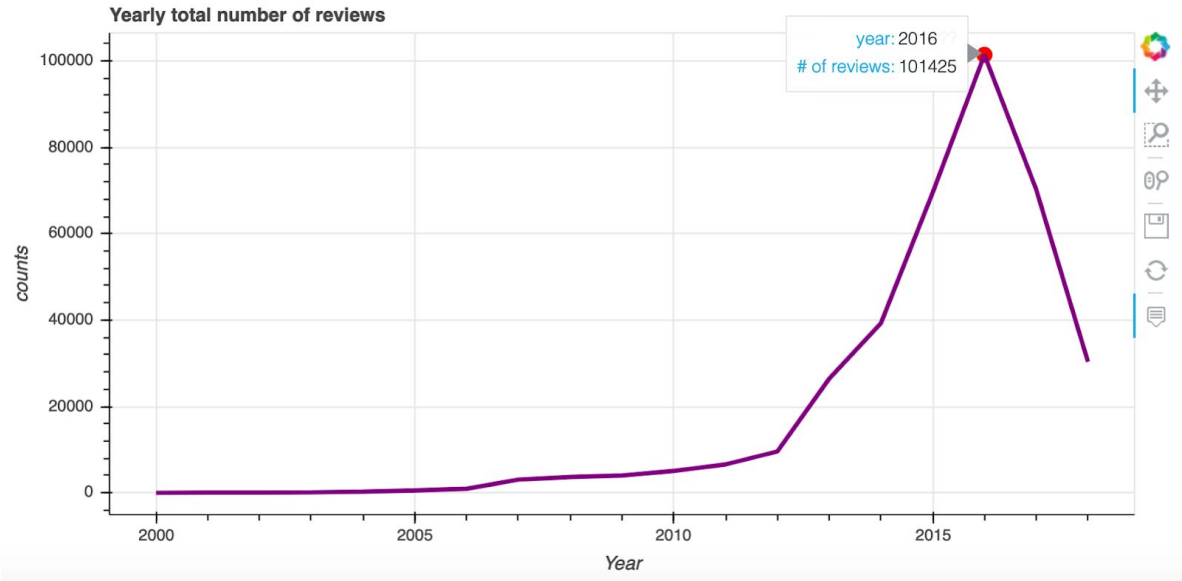
```
Amazon Customer      33222
Kindle Customer       1764
Jennifer               337
Sarah                  332
Jessica                303
                      ...
Usagi                    1
lissyj                   1
Anntodd                  1
kilowatthalfred          1
Yasmine Moustafa         1
Name: reviewer_name, Length: 221126, dtype: int64
```

Looking at the graph, it looks like the distribution of reviews is skewed: the number of frequency of the 5 ratings is way higher than the rest of the ratings combined. The highest rating is 5 with 229,549 reviews. The lowest is 2 with 20,293 reviews.

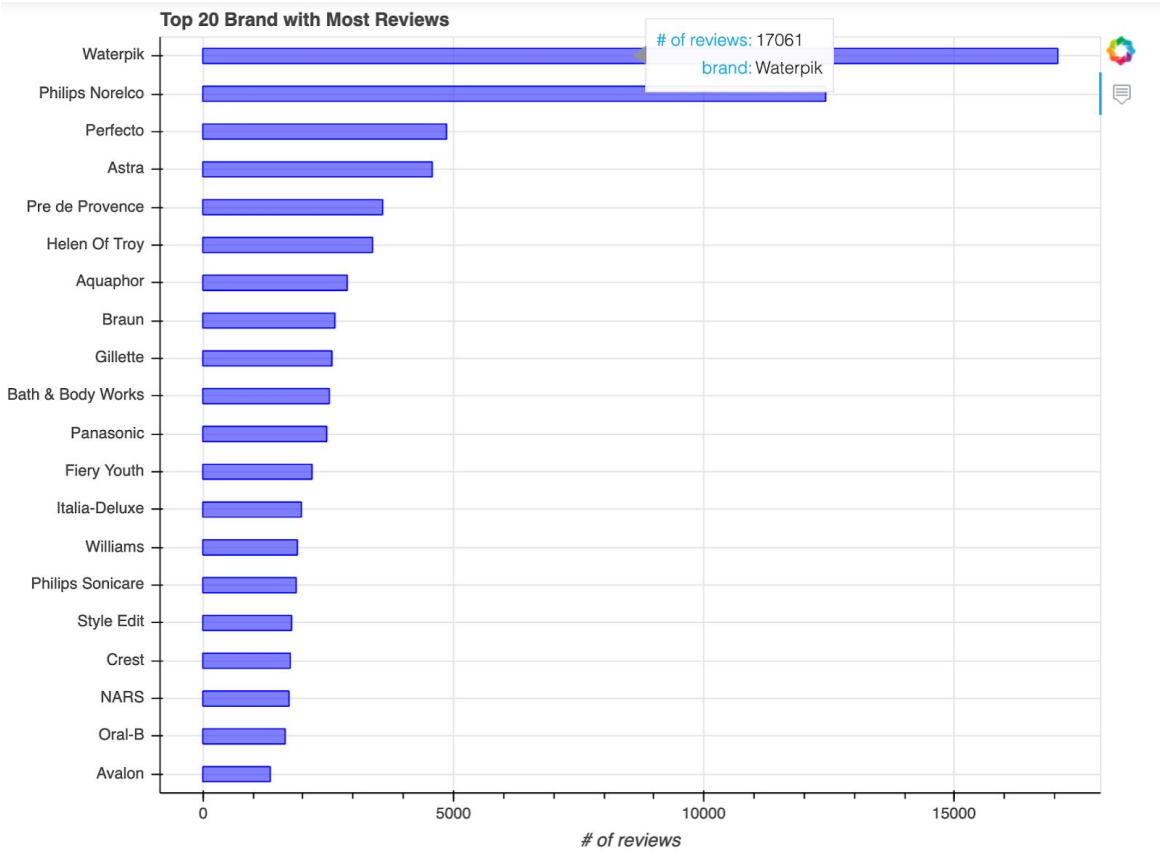


Second, let's check how many reviews are there per year. The graph shows how many total reviews are per year. The dates of reviews are from 2000 to 2018. 2016 has

the highest reviews with 101,425 reviews and 2000 has the lowest with 13 reviews.

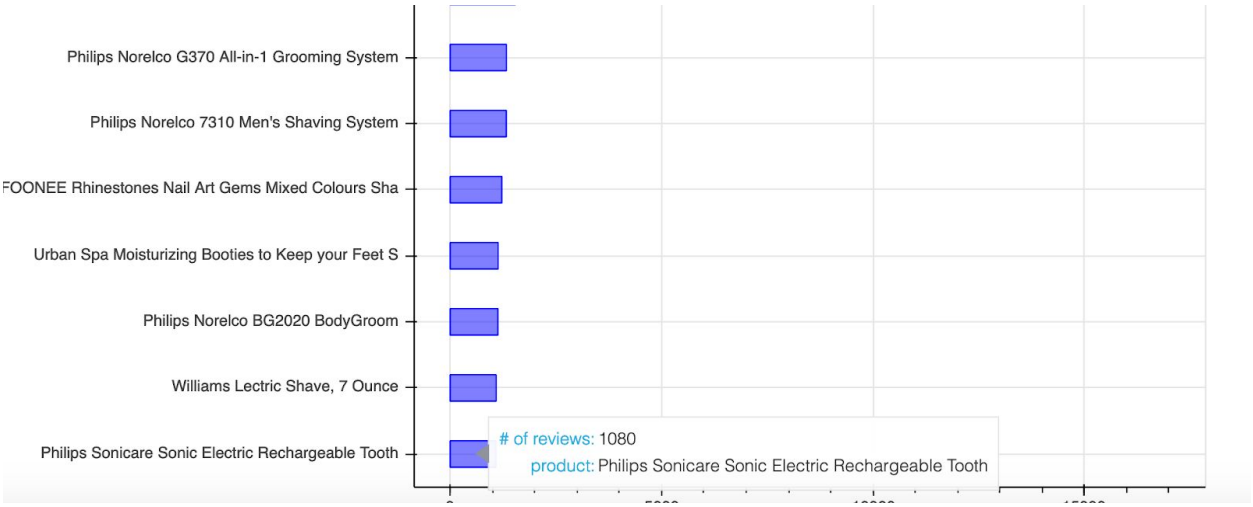


Next, we want to see the top reviewed brands for the entire 19 years. Waterpik brand had the highest reviews with 17,061 reviews. The least on the top 20 is the Avalon which has 1,344 reviews.



It is not sufficient enough to plot the brands. Some brands may have multiple products. The next thing is to plot the top reviewed products. The graph is too big that it did not fully fit in the screenshot. Waterpik Ultra Water Flosser is the top reviewed product with 17,013 reviews.

The least of the top 20 is Philips Sonicare Sonic Electric Rechargeable Toothbrush with 1,080 reviews.



Statistical Inference

Now let's go to the statistical inference. I found that the Wilson confidence interval. Wilson CI (in particular, it's lower bound) is the ideal formulation of the confidence interval to use for ranking items based on their review scores. It compensates for the "small number of elements" problem, for example, the fact that it's quite likely for a bad product with few reviews to have only good reviews simply by statistical chance.

I found that most recommender systems rank the reviews based on the highest-rated product at the top and lowest-rated at the bottom. There's some sort of "score" to sort by but this score is computed by **Score** = Average rating = (Positive ratings) / (Total ratings). Average rating works fine if you always have a ton of ratings, but suppose item 1 has 2 positive ratings and 0 negative ratings, while item 2 has 100 positive ratings and 3 negative ratings. This

algorithm puts item two (tons of positive ratings) below item one (very few positive ratings but better score). Source: https://www.evanmiller.org/how-not-to-sort-by-average-rating.html

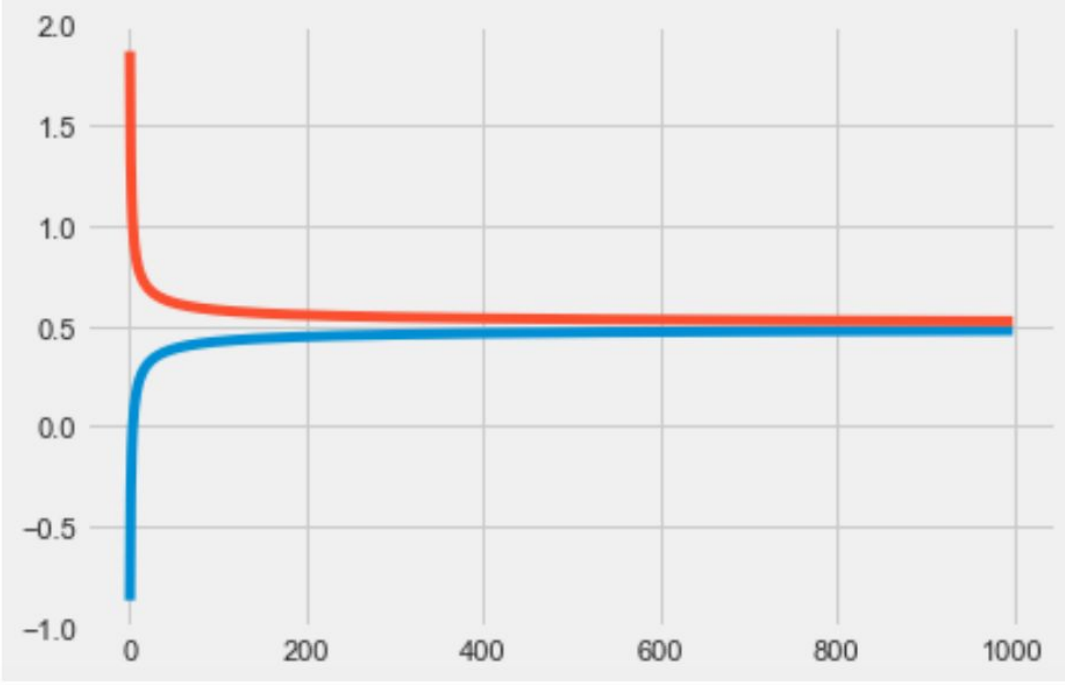
Fortunately, I found out that we can score the reviews by balancing the proportion of positive ratings with the uncertainty of a small number of observations. The math for this was worked out in 1927 by Edwin B. Wilson. It is a Bayesian approach of using the lower bound of a confidence interval around the mean.

$$\left(\hat{p} + \frac{z^2_{\alpha/2}}{2n} \pm z_{\alpha/2}\sqrt{[\hat{p}(1-\hat{p}) + z^2_{\alpha/2}/4n]/n}\right)/(1 + z^2_{\alpha/2}/n).$$

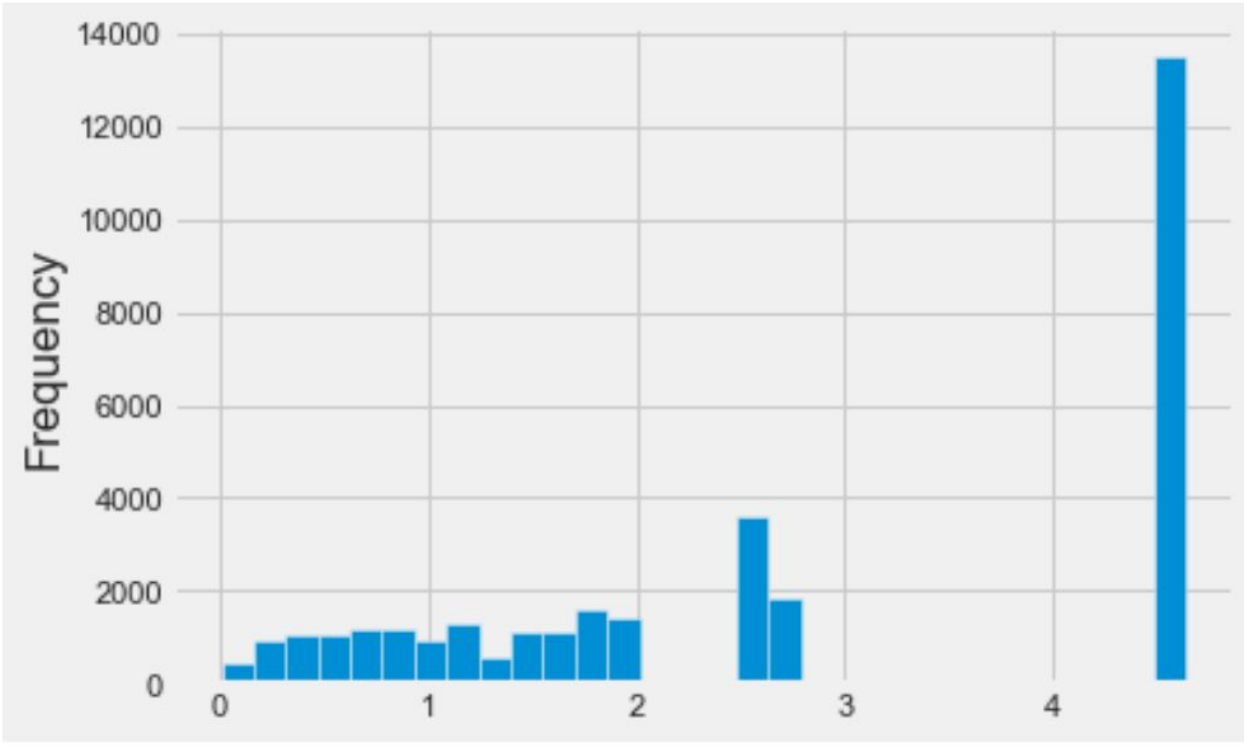
Here $\hat{p}$ is the *observed* fraction of positive ratings, $z_{\alpha/2}$ is the (1-α/2) quantile of the standard normal distribution, and *n* is the total number of ratings.

To begin with, we need to categorize what “positive” and “negative” reviews are. We created a function which would give us the boolean result of the reviews. True=positive, False =negative, categorizing the reviews that if the rating is 4 and above is considered positive reviews while 3 and below is considered negative.

When applied to a product with an underlying "recommendation ratio" of 0.5, the Wilson Confidence Interval converges to 0.5. This isn't so surprising, as it would be a lousy estimator otherwise.



Now lets try applying it in our data. We want to plot the frequency of scores (the boolean function we created for positive and negative reviews).



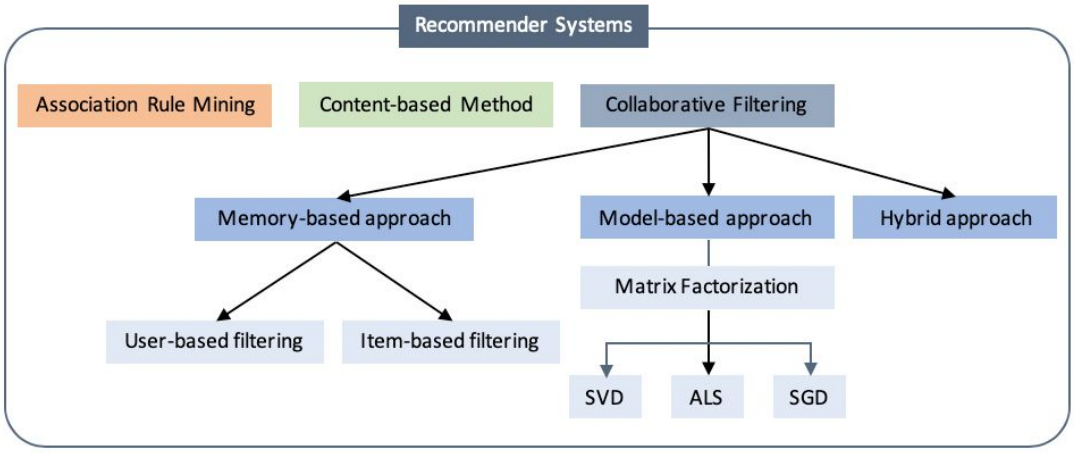
Since the data is skewed to begin with, the differences between the upper and lower bound is generally huge. Actually, it looks like this isn't such a good test set after all, as I had neglected to notice that the reviews are only a sample of all of the ones the product received. Nevertheless the point stands that getting just the average scores will underestimate the lower bounds of the data. We need to consider the confidence interval of the lower bound scores collective score of the smaller number of observations. **The main problem addressed here is what to do with items that have a small number of ratings, since an item with a single perfect rating probably should not sit atop a list of top-rated items.**

Recommendation Engine

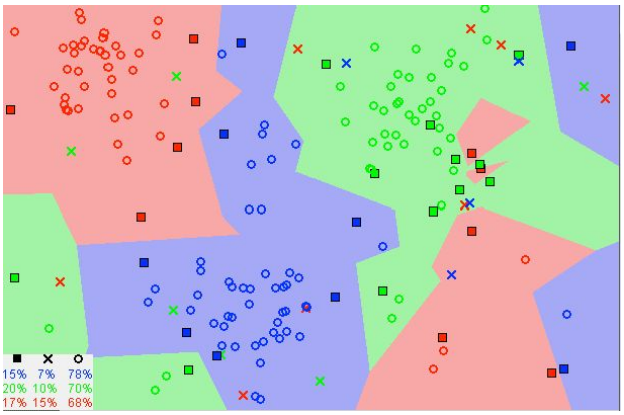
There are quite a few libraries and toolkits in Python that provide implementations of various algorithms that you can use to build a recommender. But the one that this section will present is Surprise. Surprise is a Python SciKit that comes with various recommender algorithms and similarity metrics to make it easy to build and analyze recommenders.

First we need to prepare the data. Surprise uses a Reader class to parse the final dataset . The default format in which it accepts data is that each rating is stored in a separate line in the order user item rating. (Source: https://surprise.readthedocs.io/en/stable/reader.html) The Dataset module is used to load the data in Pandas dataframe using Dataset.load_from_df.

Collaborative filtering is a technique that can filter out items that a user might like on the basis of reactions by similar users. There are two types of collaborative filtering; memory-based and model-based. Memory-based techniques rely heavily on simple similarity measures to match similar people or items together. Model-based techniques on the other hand try to further fill out this matrix. If we have a huge matrix with users on one dimension and items on the other, with the cells containing votes or likes, then memory-based techniques use similarity measures on two vectors (rows or columns) of such a matrix to generate a number representing similarity. They tackle the task of "guessing" how much a user will like an item that they did not encounter before.



For memory-based approaches discussed above, the algorithm that would fit the bill is KNN because the algorithm is very close to the centered cosine similarity. For memory-based collaborative filtering, this project will present *KNNBasic(), KNNWithMeans()* and *KNNWithZScore().* The KNN algorithm assumes that similar things exist in close proximity. In other words, similar things are near to each other.



Notice in the image on the right that most of the time, similar data points are close to each other. The KNN algorithm hinges on this assumption being true enough for the algorithm to be useful. KNN captures the idea of similarity (sometimes called distance, proximity, or closeness) with some

mathematics we might have learned in our childhood— calculating the distance between points on a graph. Source: https://towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d01761

For model-based algorithms, we are going to use SVD(), SVDpp() and NMF. Singular Value Decomposition (SVD) is a matrix decomposition method for reducing a matrix to its constituent parts in order to make certain subsequent matrix calculations simpler. The famous SVD algorithm, as popularized by Simon Funk during the Netflix Prize. The prediction $\hat{r}_{ui}$ is set as:

$$\hat{r}_{ui} = \mu + bu + bi + qTipu$$

If the user u is unknown, then the bias bu and the factors pu are assumed to be zero. The same applies for item i with bi and qi.The SVDpp algorithm is an extension of SVD that takes into account implicit ratings. Below is a diagram of a model-based collaborative filtering approach.

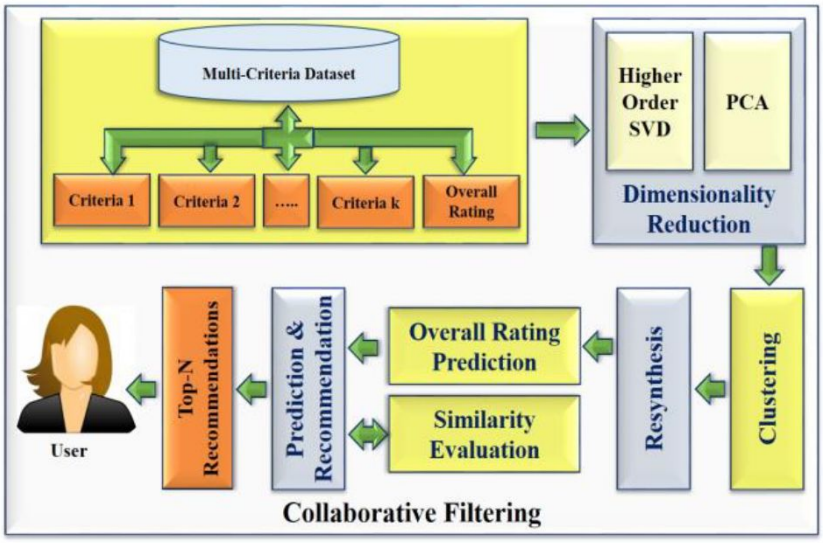
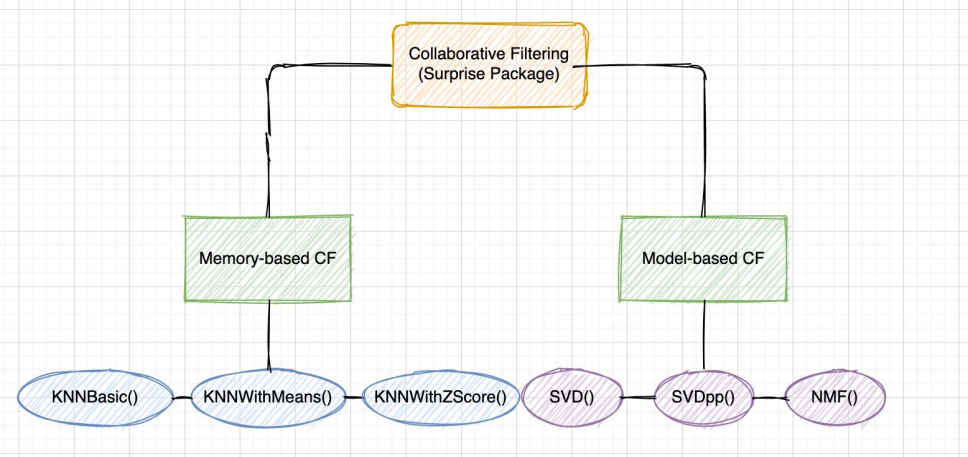


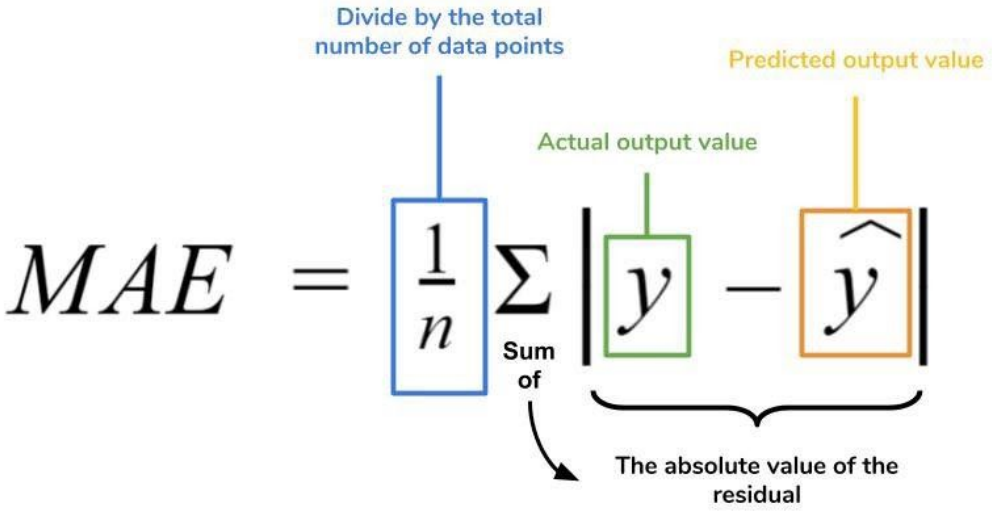
Figure 4. Proposed Collaborative Filtering Architecture.

Source: Cornell University



After knowing the algorithms that are going to be used, we'll have to run those first without hypertuning. The score that is going to be used to evaluate which algorithm performs best will be Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE).

MAE is the difference between the actual value(rating) and the predicted value. All you have to know is the lower the MAE value is, the better will be our algorithm.



Source: DataQuest

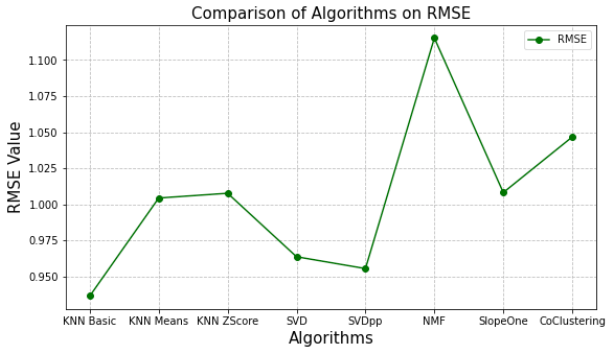
RMSE is similar to MAE but the only difference is that the absolute value of the residual(see above image) is squared and the square root of the whole term is taken for comparison. The advantage of using RMSE over MAE is that it penalizes the term more when the error is high. (Note that RMSE is always greater than MAE)

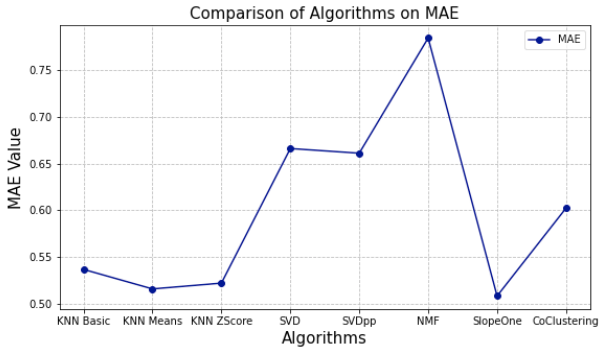
$$RMSE = \sqrt{\frac{\sum_{i=1}^{N} (Predicted_i - Actual_i)^2}{N}}$$

Source: includehelp.com

Here is the result of the evaluation of each model. We will pay more attention with RMSE scores. Looks like KNN Basic has the best RMSE score of 0.9365 for the memory-based CF while SVDpp topped in the model-based CF with the score of 0.955.

Algorithm	RMSE	MAE
KNN Basic	0.9365	0.5364
KNN Means	1.0044	0.5155
KNN ZScore	1.0077	0.5217
SVD	0.9637	0.666
SVDpp	0.9555	0.661
NMF	1.1153	0.7845
SlopeOne	1.0082	0.5079
CoClustering	1.0467	0.6021





We will get the top three vanilla models to use in our recommender engine. We are only going to consider the top RMSE scores which are the top lowest scores or errors.

1. Winner: KNN Basic (0.9372)
2. 2nd place: SVDpp (0.9576)
3. 3rd place: SVD (0.9698)

These top 3 models performed the best with no parameters. Next, we will use the GridSearchCV for cross validation and hyperparameter tuning on these models to test which ultimately end up performing the best.

Hypertuning

```
# Parameter space
svd_param_grid = {'n_epochs': [20, 25],
                  'lr_all': [0.007, 0.009, 0.01],
                  'reg_all': [0.4, 0.6]}

svdpp_gs = GridSearchCV(SVDpp, svd_param_grid, measures=['rmse', 'mae'], cv=5, n_jobs=5)
svdpp_gs.fit(surprise_data)

svd_gs = GridSearchCV(SVD, svd_param_grid, measures=['rmse', 'mae'], cv=5, n_jobs=5)
svd_gs.fit(surprise_data)
```

The parameters used for SVD are:

- **n_epochs** – The number of iterations of the SGD procedure.
- **lr_all** – The learning rate for all parameters.
- **reg_all** – The regularization term for all parameters.

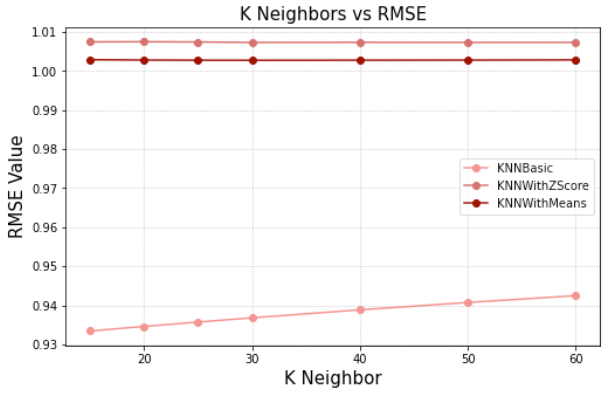
To hypertune the KNN algorithms, we used GridSearch across a single parameter, ‘k’, and tested with a range between 10 and 60.

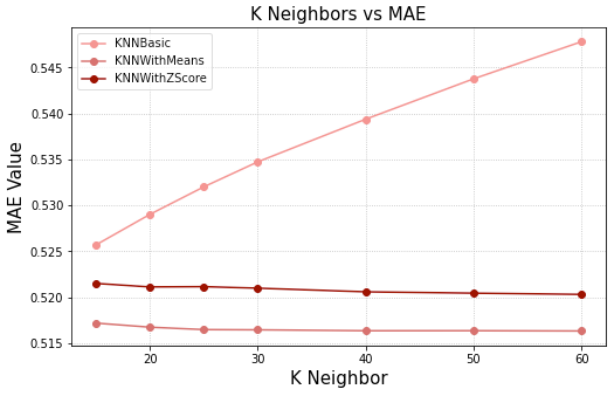
```
param_grid = {'k': [15, 20, 25, 30, 40, 50, 60]}

knnbasic_gs = GridSearchCV(KNNBasic, param_grid, measures=['rmse', 'mae'], cv=5, n_jobs=1)
knnbasic_gs.fit(surprise_data)
```

```
knnmeans_gs = GridSearchCV(KNNWithMeans, param_grid, measures=['rmse', 'mae'], cv=5, n_jobs=1)
knnmeans_gs.fit(surprise_data)
```

```
knnz_gs = GridSearchCV(KNNWithZScore, param_grid, measures=['rmse', 'mae'], cv=5, n_jobs=1)
knnz_gs.fit(surprise_data)
```

Now let’s plot the RMSE and MAE scores of each KNN algorithm to see which one has the lowest errors.





Evaluating the best hypertuned models

The best performing model was KNN Basic with an RMSE of 0.9339 and parameter k=15.

```
KNN Basic = RMSE: 0.9334 ; MAE: 0.5257
KNN with Means = RMSE: 1.0027 ; MAE: 0.5163
KNN Z = RMSE: 1.0073 ; MAE: 0.5203
SVDpp = RMSE: 1.0028 ; MAE: 0.7269
SVD = RMSE: 1.0002 ; MAE: 0.7246
```

Apply the Best Model

After hypertuning all our models, we found out that KNNBasic with k=15 performed the best on the data with an RMSE of 0.93.

Next, we are going to apply the train_test_split() of Surprise to create the train set and test set. Our train set has 19395 users and 7309 items.

After fitting the train set to the KNNBasic() algorithm, we wrote a function called ‘custom_get_predictions’ to predict items for a user. This function takes a user the number N of predictions to make, and returns the top N predictions for that user. First, the function creates an “anti testset”, which is a testset that omits items that the user has already reviewed. Then the function tests the algorithm using that testset to generate new predictions for this user, then returns the top N.

We used this function to get the item predictions for a random user in the testset.

```
reviewer = testset[0][0]
print("Getting predictions for reviewer:", reviewer)
custom_get_predictions(reviewer, 5)
```

```
Getting predictions for reviewer: A1ZWQROX1DE4ZZ
#0 (5.0) recommendation for user A2GJX2KCUSR0EI is B00006L9LC: Citre Shine Moisture Burst Shampoo - 16 fl oz
#1 (5.0) recommendation for user A2GJX2KCUSR0EI is B0014BB6WA: Summer's Eve Simply Sensitive Cleansing Wash for Sensitive Skin 9 oz
#2 (5.0) recommendation for user A2GJX2KCUSR0EI is B0151MF970: Lifetrons Essential Oil Booster Micro-Vibration Eye &amp;amp; Face Massager
#3 (5.0) recommendation for user A2GJX2KCUSR0EI is B000NN7FT8: Ambi Skincare Even &amp;amp; Clear Foaming Cleanser, 6 Ounce
#4 (5.0) recommendation for user A2GJX2KCUSR0EI is B00AG4OUOM: Organic Sweet Orange 100% Pure Therapeutic Grade Essential Oil- 10 ml
```

Conclusion

Recommendation engines are one of the useful tools for e-commerce to reach out to customers or users and advertise the product based on user ratings. Surprise is one of the useful and simple recommendation engines from Scikit. From surprise, there are various algorithms that we could use, such as k-Nearest Neighbor and SVD. To determine which algorithm performs best for our model, we used error scores Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE) and chose the model with lowest error. KNNBasic() works best for the Amazon Beauty Products Review Dataset and we can use this model to predict what Amazon users would like. Though this project uses the Amazon Beauty Products, this can be applied in any categories of Amazon Reviews Dataset.