Brewing Personalized Delights:

A Recommendation System for Starbucks Drinks

Jake Byford

INDEX

Introduction

Welcome to the world of personalized coffee recommendations! In today's fast-paced world, we all appreciate a little help in finding the perfect coffee drink that suits our tastes and preferences. That's why I embarked on a project to create a recommendation system based on Starbucks drinks, leveraging data-driven techniques to deliver tailored suggestions to coffee enthusiasts.

The primary motivation behind this project is to enhance the Starbucks experience for users. By providing personalized recommendations, users can discover new drinks that align with their preferences and tastes, leading to increased satisfaction with the brand and a more enjoyable coffee-drinking experience at Starbucks.

In today's data-driven world, personalized solutions are highly valued. By creating a recommendation system, we can leverage data-driven techniques such as cosine similarity, user feedback, and exploratory data analysis to deliver tailored suggestions to users. This can help users find drinks that truly resonate with their preferences, leading to a more personalized and enjoyable Starbucks experience.

Implementing machine learning techniques, such as cosine similarity, mean absolute error, and root mean squared error, among others, provides a practical application for machine learning skills. Creating a recommendation system for Starbucks drinks allows for the application of these techniques in a real-world context, providing valuable experience in utilizing machine learning for practical problem-solving.

By providing personalized recommendations, the recommendation system can potentially increase user engagement and loyalty to the Starbucks brand. Users who receive relevant suggestions tailored to their preferences are more likely to continue using the recommendation system and patronizing Starbucks services. This can lead to increased brand loyalty, customer retention, and ultimately, business success.

Analyzing user feedback and interactions through the recommendation system can provide valuable insights into user behavior and preferences. By conducting exploratory data analysis and identifying trends and patterns in user preferences, we can gain a deeper understanding of Starbucks customers and their drink preferences. This information can inform marketing strategies, menu development, and business decisions to better serve customer needs.

Creating a recommendation system for Starbucks drinks involves coding and implementing various algorithms and techniques, providing an opportunity to develop

and sharpen practical coding skills. From data preprocessing to machine learning model development to data visualization, this project offers a practical learning experience in coding and applying advanced techniques to real-world data.

Overall, this project aims to create a data-driven recommendation system for Starbucks drinks that could provide personalized suggestions to users based on their preferences and similar feedback. By leveraging techniques such as cosine similarity, user feedback, and exploratory data analysis, I aimed to create a robust and accurate system that would delight coffee enthusiasts and enhance their Starbucks experience.

Dataset Information

The dataset used for this project consists of two parts: a dataset generated from interactions with ChatGPT, and a user-generated dataset collected through a survey form.

1. ChatGPT-generated dataset:
    a. coffee_descriptions.csv
        - Data sources: The dataset was generated from interactions with ChatGPT, a large language model trained by OpenAI, based on the GPT-3.5 architecture.
        - Size: The dataset consists of 40 to 50 Starbucks drink names along with their respective descriptions, generated by ChatGPT.
        - Format: The dataset is in text format, with each entry containing the name and description of a Starbucks drink.
    b. coffee_prices.csv
        - Data sources: The dataset was generated from interactions with ChatGPT, a large language model trained by OpenAI, based on the GPT-3.5 architecture.
        - Size: The dataset consists of 40 to 50 Starbucks drink names along with their average price of a Grande (16 fl. oz.) drink across the United States estimated based on general knowledge of Starbucks menu pricing and common pricing trends for similar drinks. Please note that actual prices may vary depending on location, time, and other factors. For accurate and up-to-date pricing information, it is best to refer to the official Starbucks website or visit a local Starbucks store.
        - Format: The dataset is in text format, with each entry containing the name and price of a Starbucks drink.

2. User-generated dataset:
    a. user_ratings.csv
        - Data sources: The user-generated dataset was collected through a survey form created in the application. Users were asked if they have tried specific Starbucks drinks and to rate them on a scale of 1 to 5.
        - Size: The size of the user-generated dataset depends on the number of survey responses collected from users.
        - Format: The user-generated dataset is in tabular format, with each row representing a survey response and columns containing information such as the drink name, user rating, and other user information.

Data Preprocessing Steps:

- For the ChatGPT-generated dataset, the preprocessing step involves using the TfidfVectorizer from the scikit-learn library to convert the text data of the descriptions of drinks into a matrix of TF-IDF (Term Frequency-Inverse Document Frequency) values. The 'stop_words' parameter is set to 'english' to remove common English words that are often considered noise in text data. The resulting matrix is stored as a variable for further analysis or modeling.
- For the user-generated dataset, data preprocessing steps were performed, such as extracting relevant information from the survey form responses, transforming the data into a clean dataframe, and handling any missing or inconsistent data.

Use of the Dataset in the Recommendation System:

- The ChatGPT-generated dataset served as the basis for collecting real-time user information. Users were prompted to select their most preferred Starbucks drink from the list of drinks generated by ChatGPT.
- The user-generated dataset, collected from the survey form responses, was used to gather user ratings on Starbucks drinks. This data was utilized for collaborative filtering, a recommendation technique, to recommend drinks based on similar user ratings.
- Content filtering was also employed using the description of the drinks generated by ChatGPT, which helped in finding the top five most similar drinks based on the item's description to recommend to the user.

Both content filtering and collaborative filtering were used in combination to provide personalized recommendations to users based on their preferred drinks and similar user data, respectively.

Approach

To create the recommendation system, I followed a multi-step approach. First, I gathered real-time user data from a front-end application built using Flask, a Python web framework. The application defines routes for handling HTTP requests, such as GET and POST requests, and utilizes Flask's built-in functions for rendering HTML templates, retrieving form data, and redirecting to different pages.

For seamless integration with a MongoDB database, I utilized the pymongo library. I established a connection to the database using the MongoClient class and performed operations such as inserting data into the database using the insert_one() method. I also leveraged other functionalities offered by pymongo, such as data retrieval, updating, and querying, as needed.

Next, I performed data preprocessing tasks using the powerful pandas library. This included data cleaning, data transformation, and data aggregation. For instance, I used the read_csv() function to read data from a CSV file and employed functions like drop_duplicates() and fillna() to handle duplicates and missing values, respectively.

To achieve content based filtering, I converted the coffee descriptions into numerical representations and used the TfidfVectorizer and cosine_similarity classes from the scikit-learn library. The TfidfVectorizer class transformed the text data into a matrix of TF-IDF features, capturing the importance of each term in the documents. The cosine_similarity function calculated the cosine similarity matrix, which quantifies the similarity between coffee descriptions based on their text representations.

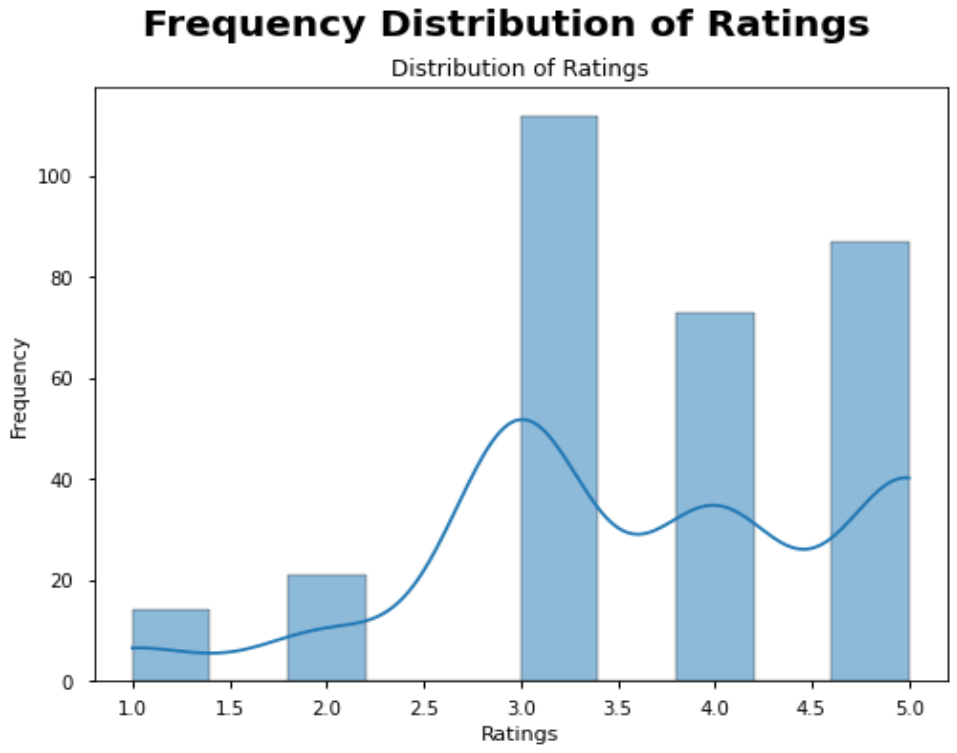
Following that, I used the database connected to MongoDB to retrieve the user ratings data to clean the data as well as manipulation techniques to change the format of the dataframe. For example, I used a technique called dataframe melting which is converting a dataframe from a wide format to a long format, with a single column for variable names ('user_id') and another column for corresponding values ('rating').

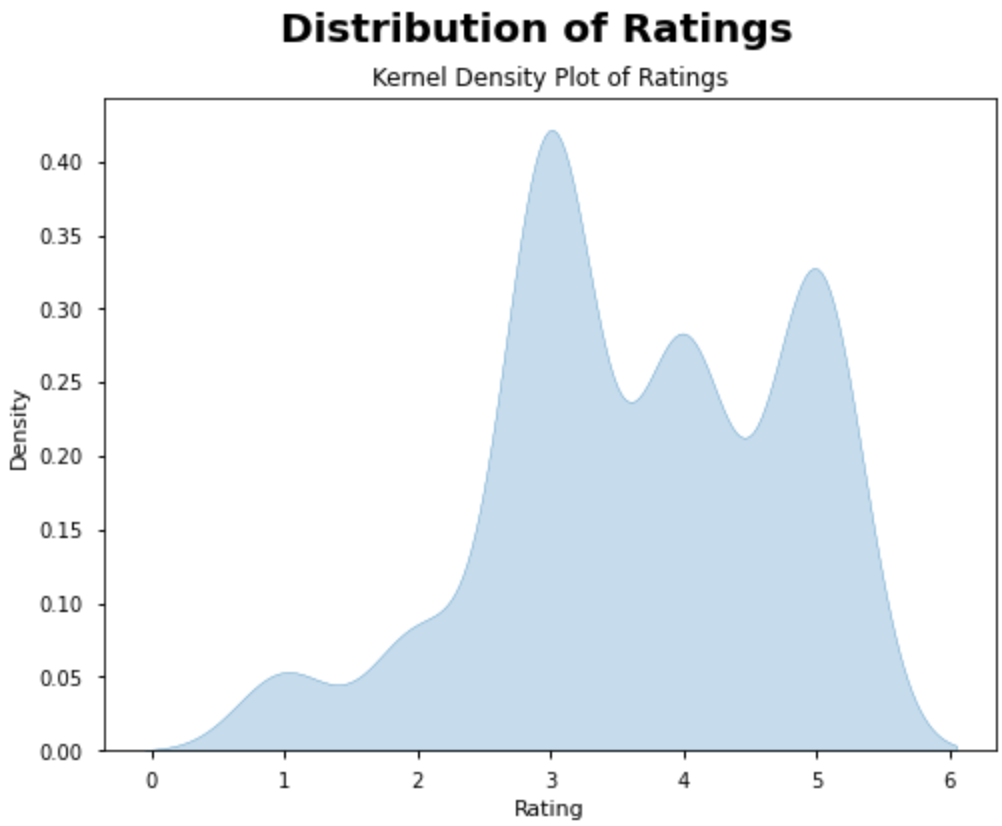
	Caffè Americano	Caffè Mocha	Caramel Macchiato	Cinnamon Dolce Latte	Pumpkin Spice Latte	...
user0	NaN	3	NaN	NaN	2	...
user1	NaN	NaN	5	NaN	NaN	...
user2	NaN	NaN	3	NaN	NaN	...
user3	NaN	NaN	NaN	NaN	NaN	...
user4	NaN	NaN	2	2	1	...

The above table is wide format while the below table is long format.

	item_id	user_id	rating
0	Caffè Americano	user0	NaN
1	Caffè Mocha	user0	3
2	Caramel Macchiato	user0	NaN
3	Cinnamon Dolce Latte	user0	NaN
4	Pumpkin Spice Latte	user0	2
5	White Chocolate Mocha	user0	NaN

This format allowed me to calculate summary statistics, aggregate columns, and create ratios on users ratings.





Analysis Results

Due to lack of data at the current timing of this project we can only use what is at our disposal. Here we can see the shape of our ratings distribution has the presence of multiple modes (peaks) which means there are different behaviors among users meaning that one drink might not be as satisfying to one user as it is to another user. To investigate this matter further one suggestion could be made. If the multimodality in the data indicates the presence of distinct subpopulations or clusters, further analysis can be conducted on each subpopulation separately. This may involve using clustering techniques, such as k-means clustering or hierarchical clustering, to identify and characterize the different subpopulations based on their unique characteristics or behaviors. Subpopulation-specific analysis can provide insights into the different subgroups within the data and help tailor strategies or interventions for each subgroup.

To get a measure on what drinks were most popular among the sample of students at NJIT who filled out the survey there were necessary data manipulation techniques needed to provide an accurate measure. To obtain this measure the most obvious way

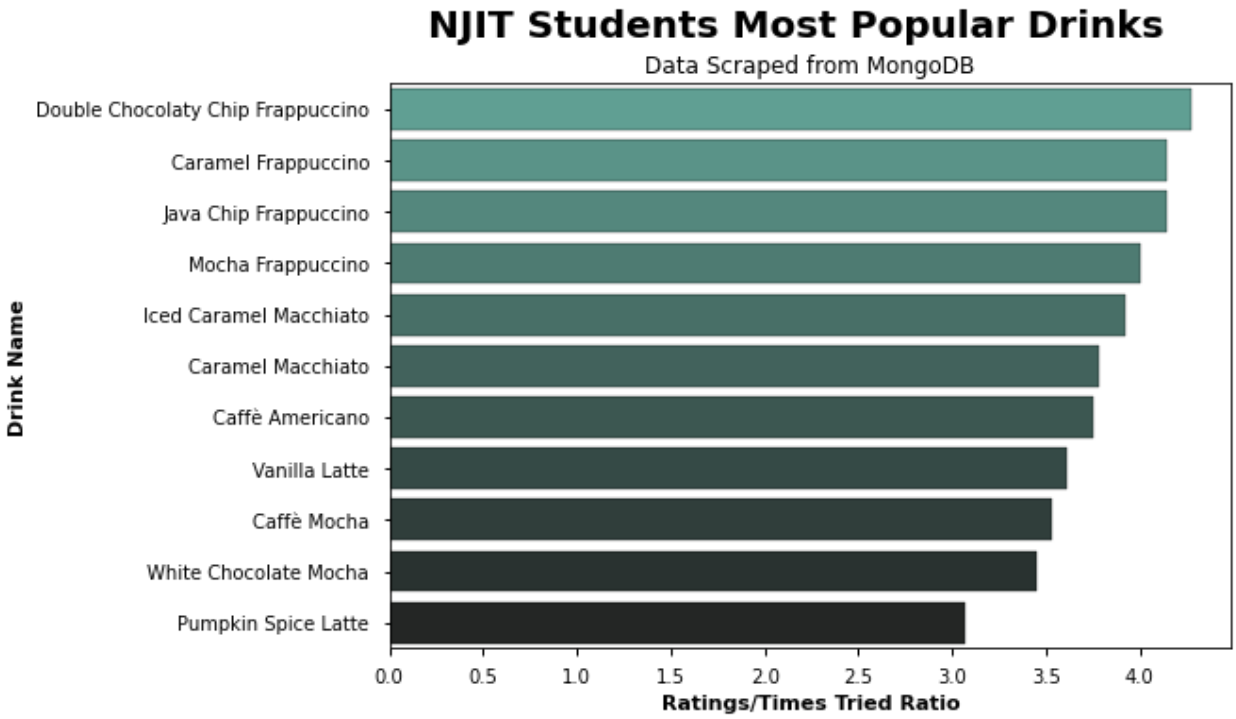
was to create a ratio between total ratings given to a particular drink and total times the drink was tried:

$$result = \frac{\sum_{\{i,j\}} r_{i,j}}{\sum_{\{i,j\}} t_{i,j}},$$

where $r_{i,j}$ is the rating given to particular drink given by a user i and drink j, and where $t_{i,j} \in \{0,1\}$ given by each user i for drink j.

item_id	result	tried	rating	price
Double Chocolaty Chip Frappuccino	4.272727	11	47	4.95
Caramel Frappuccino	4.142857	14	58	4.95
Java Chip Frappuccino	4.142857	14	58	5.25
Mocha Frappuccino	4	13	52	4.95
Iced Caramel Macchiato	3.923077	13	51	4.95
Caramel Macchiato	3.777778	18	68	4.75
Caffè Americano	3.75	12	45	3.6
Vanilla Latte	3.615385	13	47	4.55
Caffè Mocha	3.529412	17	60	4.45
White Chocolate Mocha	3.454545	11	38	4.75

This gives us a better idea of how students are collectively rating each item and where their next purchase might be. We can clearly see the most popular drinks. Four of the top five are Frappuccino drinks. However, in the analysis results section of the jupyter notebook we see that drinks that contain “Frappuccino” have a higher standard deviation among its ratings for the data we currently have. This is actually a good indicator that bias might be lower towards Frappuccinos. [Elahi, M., Ricci, F., & Rubens, N. (n.d.), p. 114] warn that acquiring more ratings for popular items tends to reduce the system error but acquiring too many high ratings can erroneously bias the system towards high rating predictions.



Collaborative Filtering Model

See **Appendix A**. The code uses a user-based collaborative filtering model to generate recommendations for a target user. First, the target user is determined as the last user who completed the survey in the database. Then, the similarity scores between the target user and all other users are extracted from a user-based cosine similarity matrix, which is stored in clean_ratings_cosine_similarity_df. The similarity scores are sorted in descending order, and the top N similar users (Nearest Neighbors) are selected based on the indices of the sorted similarity scores, which are stored in similar_users_indices.

Next, the ratings of the similar users for items that the target user has not rated are extracted from clean_ratings_df using similar_users_indices and stored in similar_users_ratings. The ratings of the target user are also retrieved from clean_ratings_df and stored in target_user_ratings. Items that the target user has already rated are filtered out from similar_users_ratings using a mask created from target_user_ratings.

The code then calculates the weighted average of the ratings of the similar users, using the similarity scores as weights. The similarity scores are extracted from similarity_scores based on the similar_users_indices. The ratings of similar users are

multiplied element-wise with the corresponding similarity scores, and the resulting weighted ratings are summed along axis 0 (rows) and divided by the sum of similarity scores to get the weighted average ratings. The weighted average ratings are stored in weighted_avg_ratings:

$$w = \frac{\sum r_i * s_i}{\sum s_i},$$

where r_i The function then filters out items that the target user has already rated, sorts the weighted average ratings to get the top-rated items for recommendation

Finally, the weighted average ratings are sorted in descending order to get the top-rated items, which are stored in the recommended_items Series. The top N recommended items for the target user are accessed from recommended_items using the head() function with top_n as the argument, and stored in top_n_recommended_items. These items are the final recommendations for the target user based on the user-based collaborative filtering model.

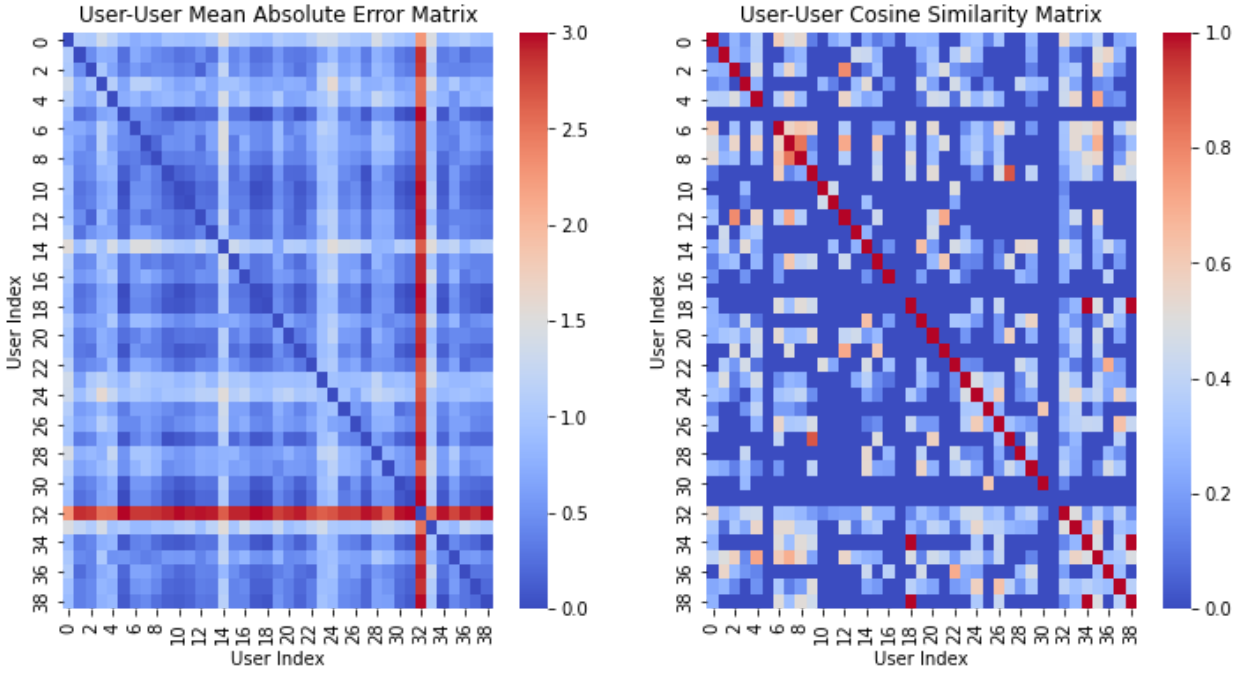
Performance Evaluation

According to [Su, X. & Khoshgoftaar, T. M.] the most widely used metric in collaborative filtering research literature is Mean Absolute Error (MAE), which computes the average of the absolute difference between the predictions and true ratings

$$MAE = \frac{\sum_{\{i,j\}} \left| p_{i,j} - r_{i,j} \right|}{n},$$

where n is the total number of ratings over all users, $p_{i,j}$ is the predicted rating for user i on item j, and $r_{i,j}$ is the actual rating. The lower the MAE, the better the prediction. Now let's look at the Python code in Appendix B used to grab the scores.
In the Python code, it calculates the mean absolute error (MAE) between all pairs of users in the 'clean_ratings_df' DataFrame. See **Appendix B**. It starts by initializing an empty list 'user_mae_scores' to store the MAE scores for each user. Then, it iterates through each user in the DataFrame using a for loop, and for each user, it retrieves the

ratings data, fills any missing values with 0, and stores them in the 'user_a' variable. It also initializes an empty list 'user_a_mae' to store the MAE scores between 'user_i' and all other users. It then iterates through each user in the DataFrame again using a nested for loop, retrieves the ratings data for 'user_j', fills any missing values with 0, calculates the MAE between 'user_a' and 'user_b', and appends the result to the 'user_a_mae' list. Finally, it appends the 'user_a_mae' list to the 'user_mae_scores' list to keep track of the MAE scores for 'user_i'. After iterating through all users, the 'user_mae_scores' list contains the MAE scores for each user against all other users in the DataFrame.



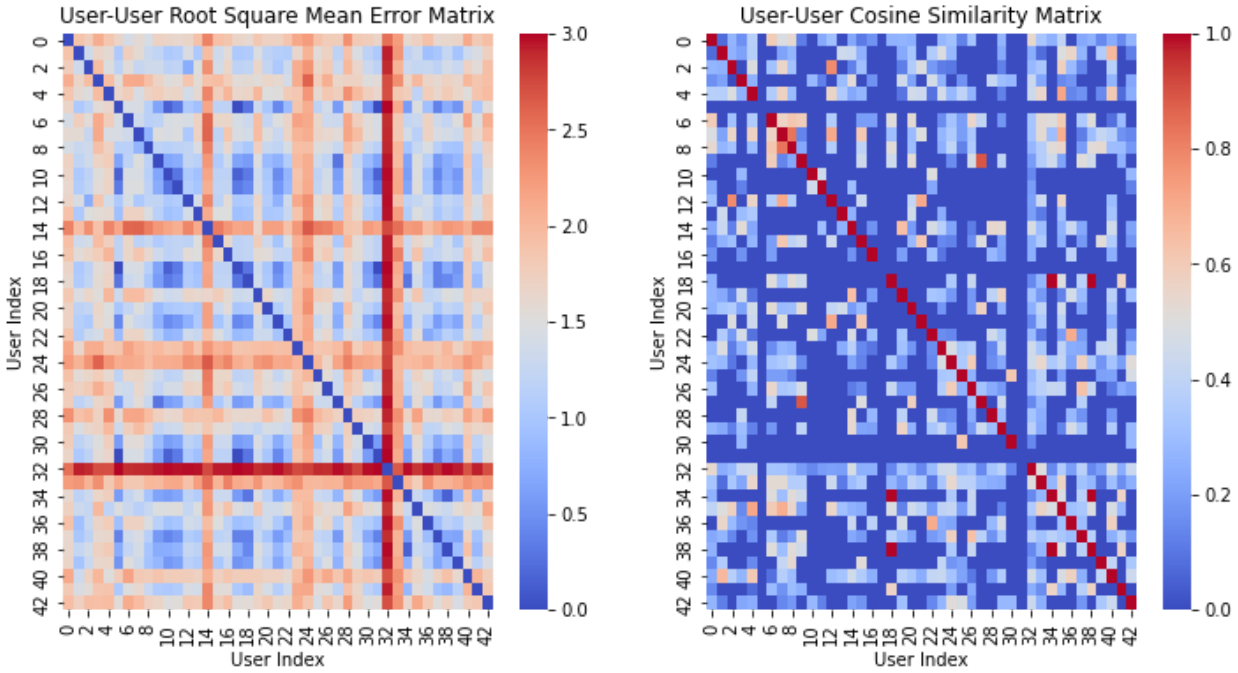
Since I couldn't collect as much data as I had hoped, you can still analyze the heat maps to compare the MAE matrix (left) with the cosine similarity user matrix (right). As mentioned earlier, a lower MAE value indicates better prediction, while a higher cosine similarity score also indicates better prediction. Despite the heat maps appearing as rectangles rather than squares, you can observe that these two matrices are symmetric. The main objective here is to identify any patterns in the data. One noticeable pattern is between user 7 and user 8, where the MAE heat map shows a dark blue indicating lower MAE scores, while the cosine similarity heat map shows light orange to red. This indicates that our model may be working well in that particular area.

Also according to [Su, X. & Khoshgoftaar, T. M.] The Root Squared Mean Error (RMSE) is becoming popular because it is the Netflix prize metric for movie recommendation performance:

$$RMSE = \sqrt{\frac{1}{n} \sum_{\{i,j\}} (p_{i,j} - r_{i,j})^2} ,$$

where n is the total number of ratings over all users, $p_{i,j}$ is the predicted rating for user i on item j, and $r_{i,j}$ is the actual rating again. *RMSE* amplifies the contributions of the absolute errors between the predictions and the true values.

The code in **Appendix B** for RMSE is very similar to the MAE, however, we calculated the absolute error and took the mean of the error. Here we take the square root to get the RMSE of all users put into matrix form.



Similar to MAE, RMSE is also evaluated with lower scores indicating better predictions. Upon examining both heat maps, it is evident that there are patterns for user 18 in comparison with user 34 and user 38, suggesting that our models are performing well in those areas.

Conclusion

Finally, I deployed the user-based recommendation algorithm to my web application on the Heroku platform. This involved creating a Heroku app and deploying the Flask app to Heroku. I also made use of additional configurations and settings, such as scaling, logging, and monitoring, to ensure smooth deployment and operation of the recommendation system on Heroku.

By following this comprehensive approach and leveraging various Python libraries like Flask, pandas, scikit-learn, pymongo, and Heroku, I was able to create a robust and scalable recommendation system that collects real-time user data, performs data manipulation tasks, calculates similarity between coffee descriptions as well as users, connects to a MongoDB database, and deploys as a web application on Heroku.

In conclusion, while the findings of this analysis should be interpreted with caution due to the limitations of the small sample size and limited menu selection, it provides valuable insights and raises potential research questions for future investigation. Expanding the sample size and considering a broader range of drinks from Starbucks' menu could enhance the robustness and applicability of the project. Research questions such as customer satisfaction with taste and quality, popularity of coffee drinks, frequency of purchases, perception of price and value, customization options, brand loyalty, comparison with other coffee chains, demographic influences, and customer feedback can further deepen our understanding of Starbucks' coffee drinks and customer preferences.

This project has provided a glimpse into the importance of recommendation systems in businesses, as highlighted by Bell, Koren, and Volinsky (2010) and Jannach, D., & Jugovac, M. (2016). However, measuring the true business value of a recommender system can be complex, and the extent of its effectiveness may vary depending on various factors. Many companies rely on methods such as A/B tests and offline testing of historical data to assess the impact of potential changes in recommendation deployment.

Overall, this project has been a valuable learning experience and has shed light on the role of recommender systems in enhancing customer experiences and driving business outcomes. Further research in this area could contribute to the advancement of recommendation algorithms and strategies for businesses, ultimately benefiting both customers and businesses alike.

References

Bell, R. M., Koren, Y., & Volinsky, C. (2010). The Netflix Recommender System: Algorithms, Business Value, and Innovation. *ACM Transactions on Management Information Systems (TMIS),* 1(1), 1-19.

Elahi, M., Ricci, F., & Rubens, N. (2014). Active Learning in Collaborative Filtering Recommender Systems.

Jannach, D., & Jugovac, M. (2016). Measuring the Business Value of Recommender Systems. University of Klagenfurt, TU Dortmund.

Real Python. (n.d.). How to Build a Recommendation Engine with Collaborative Filtering. Retrieved from https://realpython.com/build-recommendation-engine-collaborative-filtering/

Su, X., & Khoshgoftaar, T. M. (2009). A survey of collaborative filtering techniques. *Advances in artificial intelligence*, *2009*.

Appendix

Appendix A: User-based Collaborative Filtering Model written in Python
Appendix B: Mean Absolute Error/Root Mean Squared Error

Appendix A

```
### USER BASED COLLABORATIVE FILTERING MODEL ###
################################################
# Front-end grabs the last user who completed survey in the database
target_user = clean_ratings_df[-1:].index[0]

# Extract the similarity scores for the target user from the user-based cosine
similarity matrix
similarity_scores = clean_ratings_cosine_similarity_df[target_user]

# Sort the similarity scores in descending order and select the top N similar users
similar_users_indices = np.argsort(similarity_scores)[::-1][1:top_n+1]

# Get the ratings of similar users for items that the target user has not rated
similar_users_ratings = clean_ratings_df.iloc[similar_users_indices]
target_user_ratings = clean_ratings_df.loc[target_user]

# Filter out items that the target user has already rated
unrated_items_mask = target_user_ratings.isna()
similar_users_ratings = similar_users_ratings.loc[:, unrated_items_mask]

# Calculate the weighted average of the ratings of similar users using the
similarity scores as weights
similarity_scores = similarity_scores[similar_users_indices]
weighted_avg_ratings = similar_users_ratings.mul(similarity_scores, axis=0).sum() /
similarity_scores.sum()

# Sort the weighted average ratings in descending order to get the top-rated items
recommended_items = weighted_avg_ratings.sort_values(ascending=False)

# You can now access the recommended items for the target user in the
`recommended_items` Series.
# For example, to get the top N recommended items for the target user:
top_n_recommended_items = recommended_items.head(top_n)
```

Appendix B

```
### MEAN ABSOLUTE ERROR SCORES ###
##################################
user_mae_scores = []
for i in range(0, len(clean_ratings_df)):
    user_a = clean_ratings_df.fillna(0).loc[f"user{i}"].values
    user_a_mae = []
    for j in range(0, len(clean_ratings_df)):
        user_b = clean_ratings_df.fillna(0).loc[f"user{j}"].values
        mae = mean_absolute_error(user_a,user_b)
        user_a_mae.append(mae)
    user_mae_scores.append(user_a_mae)
user_mae_scores
```

```
### ROOT MEAN SQUARED ERROR SCORES ###
#####################################
user_rmse_scores = []
for i in range(0, len(clean_ratings_df)):

    # Extract the actual ratings from the DataFrames as arrays
    user_a = clean_ratings_df.fillna(0).loc[f"user{i}"].values
    user_a_rmse = []

    for j in range(0, len(clean_ratings_df)):

        # Extract the predicted ratings from the DataFrames as arrays
        user_b = clean_ratings_df.fillna(0).loc[f"user{j}"].values

        # Calculate the squared error between actual and predicted ratings
        squared_error = (user_a - user_b) ** 2

        # Calculate the mean of squared errors
        mse = np.mean(squared_error)

        # Calculate the RMSE by taking the square root of MSE
        rmse = np.sqrt(mse)
        user_a_rmse.append(rmse)
    user_rmse_scores.append(user_a_rmse)
```