Wine Recommender System

Shane Nelson sn888@drexel.edu
Allie Schneider as5664@drexel.edu
Francis Villamater fv48@drexel.edu
Will Wu ww437@drexel.edu

College of Computing and Informatics, Drexel University, Philadelphia, PA 19104

***Abstract* – The goal of this project is to construct a recommender system which can be used by both novice consumers and experienced wine sommeliers alike to provide recommendations for new wines to try. The obtained dataset contains ratings for thousands of wines and the commonly used characteristics to categorize them including, but not limited to, variety, country/province/region, winery, price, designation, and even a user-inputted description. This paper provides an overview of the data cleaning and pre-processing steps taken to prepare the data for different machine learning models and an analysis of the results of our modeling for different types of recommender systems.**

***Keywords* – Wine, Wine Enthusiast, Recommender System, Python, Machine Learning**

GitHub Repository - https://github.com/snelson97/DSCI591_G8SAFW

1. Introduction

Wine is one of the most popularly consumed alcoholic beverages in the world, with multiple different varieties being produced by thousands of different wineries found in many countries around the globe. Although it may be easy for an experienced wine taster to identify and select a wine based on their individual knowledge, the average consumer would almost certainly not consider themself a wine connoisseur and may become overwhelmed with the myriad combinations of region, winery, variety, and description of a wine. Of the different types of recommender systems which exist, a content-based system and a knowledge-based system both stand out as being types of systems which would be particularly relevant. The goal for a content-based system is to utilize features in the data to characterize the qualities of a wine (the "content"), then take user reviews/ratings and identify a single, "target" user for which the system could predict their rating of un-reviewed wines in order to provide recommendations for new wines to try based on the highest predicted ratings. The deliverable of the content-based system is to take wine review data for a target user as input and provide a list of recommended, previously un-reviewed, wines as output. Alternatively, one type of knowledge-based recommender, constraint-based, takes input search criteria from a user, (for example description keyword(s), country/region, winery, price, etc.) and provides an output of related wines based on the search criteria and ranked by other user ratings. Similarly, a case-based knowledge recommender takes an input wine name/ID and provides a list of similar wines to try based on similarity calculations. The deliverable of the knowledge-based system is to take input search criteria or a specific example of wine and provide relevant, related wines based on similar characteristics. Another potential option is to use clustering to group wines based on shared characteristic combinations and provide recommendations based on wines within the same cluster ranked by average user rating.

The system is applicable to a wide range of users from novice wine consumers to well-versed wine sommeliers. A novice user can use this system to help them to find new wines that they enjoy. The system can also aid novice wine consumers in cultivating a better understanding of their palate for wine. Additionally, the recommender system can provide guidance and education that will help the novice wine consumer understand the complexities of wine by offering insights into different grape varieties, regions, and styles. This knowledge empowers novice wine consumers to make more informed decisions and enhances their overall appreciation of wine. Finally, the system can save a novice wine consumer valuable time and energy in their wine selection process. Instead of feeling overwhelmed by an extensive wine list or relying on guesswork, they can rely on the system's recommendations, which are tailored to their preferences and curated based on expert knowledge. A well-versed wine sommelier can use this system to expand the breadth of their experiences. The system can serve as a valuable tool for discovering new and obscure wines, expanding the wine sommelier's knowledge beyond the existing repertoire. The system may also bring to light different perspectives and suggestions by helping the wine sommelier to explore different wine styles and regions. The system may empower wine sommeliers to elevate their wine selection skills, provide personalized recommendations, and deliver exceptional experiences to their clientele.

2. The Dataset

Our Wine Recommender System uses a dataset that was acquired from Kaggle.com[1]. The dataset was scraped from WineEnthusiast in 2017 by a Kaggle user and includes nearly 120,000 unique wine reviews provided by wine sommeliers. Each of the 120,000 reviews included in this dataset have been reviewed by a wine enthusiast who recorded information about their review. The information recorded from each wine review includes the country the wine was made, a description of the wine, a designation for each wine, a points or score of how much the wine was enjoyed, the price of each bottle of wine, the province and region the wine was made, the reviewer's name and twitter handle, the title of the wine, the variety of the wine, and the winery that made the wine for a total of 13 attributes.

This dataset has been selected for the use of the Wine Recommender System because of the wide range of information that was recorded for each wine that was reviewed. The points or score of how much the wine was enjoyed by the wine enthusiast gives the system a numeric value for the quality and taste of the wine. The inclusion of a description of the wine straight from a wine enthusiast will help the system to understand how wines differ from each other. The variety of each wine is one of the foremost identifying features of a wine and is used to distinguish wines based on the types of grapes used in production. The country, province and region of each wine allows the system to compare wines that were made thousands and thousands of miles apart, as well as compare wines that were made within the same geographic area. Finally, the price of each bottle of wine will allow the system to determine how accessible each wine is to the average consumer.

WineEnthusiast (winemag.com) started as a monthly print magazine and has expanded into an acclaimed, multifaceted media brand offering of-the-moment content in the print and digital publishing space[2]. Wine Enthusiast has over 4 million readers and considers itself as the most influential voice in wine and drinks journalism today. WineEnthusiast offers perspectives, stories and insights on wine and drinks. WineEnthusiast has a global network of editors, writers, and tasters which allow for an accessible but expert view on the world of wine. WineEnthusiast offers 10 annual glossy magazine editions, a website (winemag.com), a biweekly podcast, a wine review buying guide, and virtual and in-person events.

3. Exploratory Data Analysis

Starting with the target attribute, 'points' is a discrete variable with a range from 80-100 increasing by 1, for a total of 21 levels. The most common value in 'points' is '88' at 15,141 (12.7%), followed by '87' at 15,126 (12.7%), and '90' in third at 13,787 (11.6%). When looking at the distribution on a histogram (Figure 1), within this range from 80-100, the distribution is fairly normal with, if anything, a very slight right skew since the median, 88, is marginally less than the mean, 88.44.

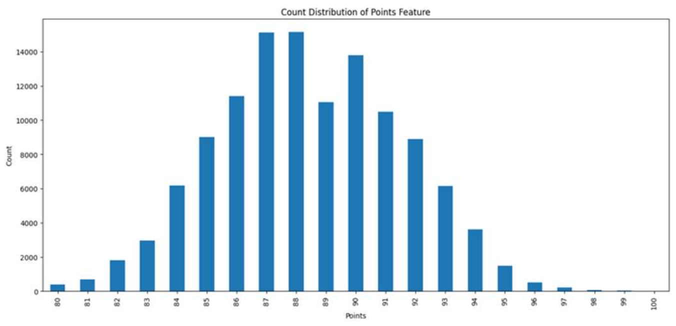


Figure 1: Distribution of Points

Analysis of the 'description' attribute is limited considering it was used as a free-text field, but it contains 118,938 unique values, indicating that there is a different description for nearly every wine in the dataset. The 'winery' attribute, a categorical free-text field, contains 16,757 unique values and provides information about the specific winery/company which produced the wine. The attribute 'designation' is a subcategory of winery and is also a free-text field with 37,979 unique values and information about the specific vineyard within a winery from where the grapes were obtained. Unfortunately, a quick scan of this attribute will reveal that not all values are written in English, nor has the field been standardized. The top five values are ‘Reserve’, ‘Estate’, ‘Reserva’, ‘Riserva’, ‘Estate Grown’, respectively; here we can see three variations of the word ‘Reserve’, making this attribute unfit for use in our model. The ‘variety’ attribute has 707 unique values and indicates the type of grapes used to produce the wine.

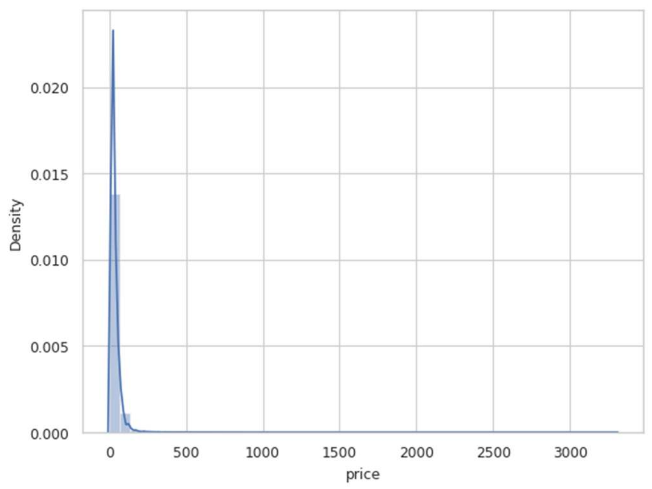


Figure 2: Distribution of Price

The ‘price’ attribute is visibly heavily skewed to the right as seen in the histogram (Figure 2), which is supported by the fact that the mean ($35.36) is more than 10 price points higher than the median ($25) and price ranging from $0-$3,300. Under our attribute ‘country’, data is heavily dominated by ‘US’ at 50,298, comprising just over 42% of the entire dataset. The second and third most common values are ‘France’ at 19,771 (17%) and ‘Italy’ at 17,830 (15%), respectively. There are only 42 unique values, but the top 3 values already make up almost 75% of the dataset, which indicates that a more specific location feature would be beneficial. The ‘province’ feature is much better, with 422 unique provinces providing a better level of detail than the country feature, however once again the data is still dominated by California wines (>30%). California is much larger geographically than any other province included in the dataset, and it would be unfair to consider Northern California and Southern California wines to be from the same location, therefore we must continue to explore the region features and their usability. ‘Region_2 ‘is only populated for provinces of California, New York, Oregon, and Washington and provides a great level of detail compared to just using the province feature, however there is a significant number of null values for all countries outside of the US and also for some excluded US states. ‘Region_1’ is an extremely specific feature with over 1,200 unique values, however there are still a large number of null values (>10%).

Finally, the ‘taster_name’ feature has 19 unique values and is not necessarily something that we would use as input for any machine learning model, but it would definitely be required for a content-based recommender so that a target user can be identified, and recommendations can be made based on wines which have not yet been reviewed by the user. The ‘title’ feature, similar to taster name, has 118,840 unique values and does not provide much usefulness for modeling purposes but would be valuable for the end-user when making recommendations. Without the wine title, it would not be possible to recommend specific wines and it would also not be possible to distinguish between two wines with the same features, so although this feature will be excluded from modeling, it must be kept in the final data frame for labeling purposes.

4. Data Cleaning and Pre-Processing

Missing Values

We encountered 8,265 missing values in the price column. Since this represents approximately 7% of the total 118,971 rows, we have decided to remove these rows with missing price values. Even after dropping them, we still have over 110,000 rows available for analysis. Similar to the approach taken with the price column, we have also chosen to drop the missing values in the country and variety columns. By removing the missing values in the country column, any associated missing values in the province column is automatically eliminated. Furthermore, the variety feature only contains a single missing value (which could be a result from data entry error) and we have dropped it as well.

Data Transformation

As previously noted in Figure 2, the price feature is heavily skewed with many extreme outliers. To address this potential issue, we have chosen to employ a logarithmic transformation on the column. This transformation proportionally reduces all values, eliminates outliers, adjusts skewness, equalizes variance, and lowers the standard deviation. Consequently, the emphasis shifts towards relative differences between values rather than absolute differences, which can prove beneficial in the development of our model.

Furthermore, we have also chosen to update the range of the 'points' feature for easier interpretability by end-users of our system. Currently, the scale of points being from 80-100 may mislead a user into thinking a rating of 82 is 'high' because of their limited knowledge of the range of ratings, even though an 82 rating is actually relatively low. Therefore we find it beneficial to re-scale this feature from 0-11, which is a more traditional range. This is a purely 'cosmetic' update and will not affect the modeling or feature correlation whatsoever. Figure 3 visualizes the correlation between the points and price features after transforming each, displaying a strong, positive relationship (Pearson Correlation = 0.62).

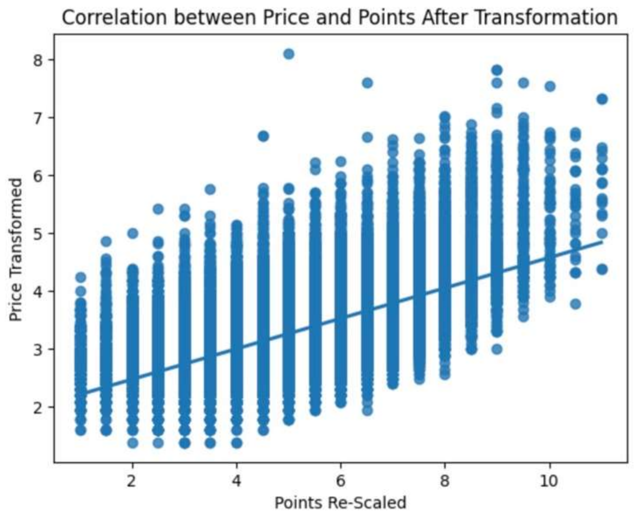


Figure 3: Correlation Between Price and Points After Transformation

We have also added another categorical column 'point_range' to be used for classification algorithms, which groups wines into 'high' or 'low' ratings based on the median rating of 88. Any wines below 88 are considered 'low' while anything above is considered 'high'.

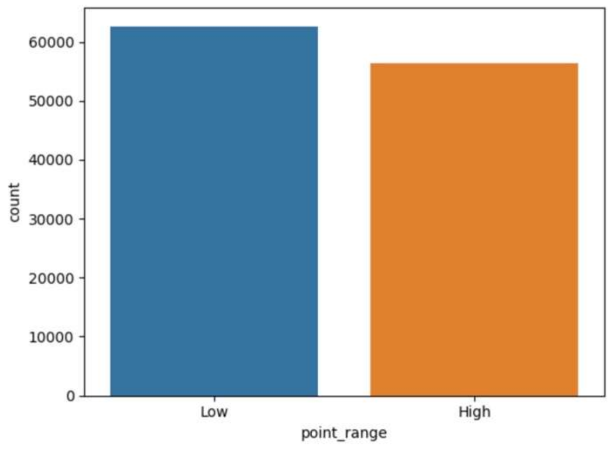


Figure 4: Distribution of High/Low Point Range

Figure 4 displays the distribution of the newly created 'point_range' feature which is relatively balanced and would not require any over/under-sampling for classification modeling.

Finally, the description feature is initially a string of text for each review, therefore we found it easiest to convert each description into a lowercase list of words with all punctuation removed. After removing stop words from each description, there are 46,638 unique keywords found in the dataset which could potentially be used for a knowledge-based recommender system.

Feature Creation

Due to the previously discussed difficulties with using just one of the country, province, or region features, we have created a new 'location' feature which is a copy of province with region_2 used in place of any of the 4 U.S. states for which this feature is populated.

Additionally, we have created a categorical price bucket feature to group prices in increments 10-, 20-, or 100-dollar ranges with a final '$200.00+' bucket. The creation of this categorical feature is required for cosine similarity calculations which we will be discussing later.

6. Predictive Modeling

Clustering

We used K-Means to build a clustering model on the transformed price and transformed points features. We decided to use only these two features as they are the only numeric features in our dataset. Including categorical features that have been transformed to numerical results in undistinguishable clusters, thus we have chosen to only use the original numeric features.

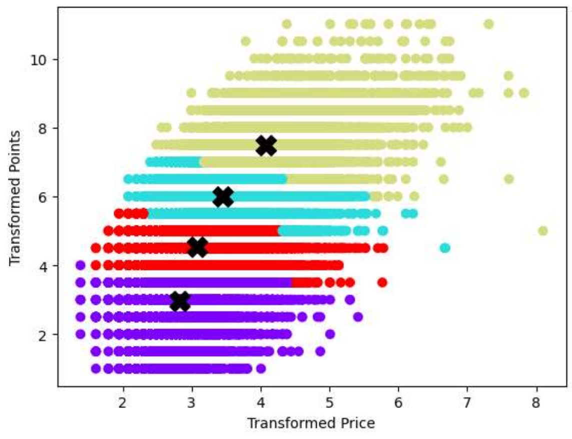


Figure 5: K-Means Clustering (4 Clusters)

We performed the elbow method to determine the optimal number of clusters. We found that four clusters are optimal. Figure 5 shows these four clusters on a graph.

Cosine Similarity

Cosine similarity is an incredibly effective way to measure and compare two wines and has applications in a content-based recommender system. Calculating the cosine similarity between a target wine's vector and other the vector of other wines in the database allows for the creation of a similarity score that can be used to rank related wines and derive a predicted rating based on the average rating of n related wines. For our cosine similarity calculations, we created a feature vector of the wine's price bucket, variety, country, and province.

To test the effectiveness of our cosine similarity calculations and their ability to predict the rating of a wine, we chose a target user from our dataset, Susan Kostrzewa, and randomly selected 50 wines from her set of reviews. For each of the 50 selected wines, we calculated the cosine similarity between the selected wine and all other wines in Susan's reviews and made a prediction of that wine's rating based on the average rating made by Susan for the 10 most similar wines.

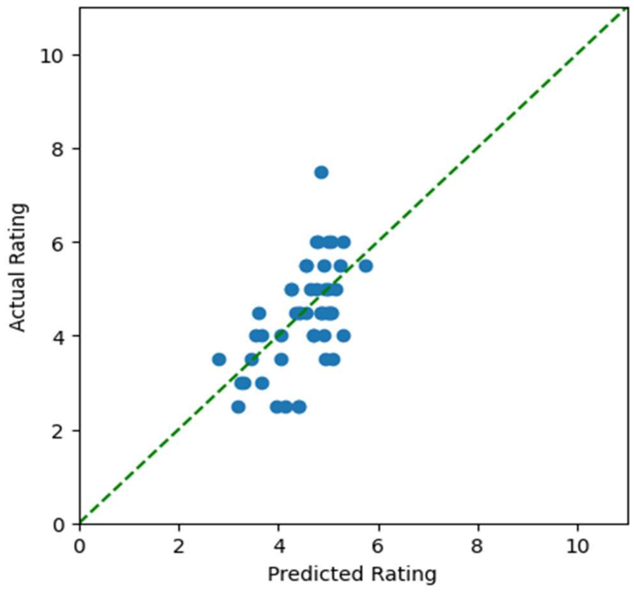


Figure 6: Cosine Similarity Predictions vs. Actual Rating

The resulting root mean-squared error (RMSE) of the cosine similarity predictions for Susan's 50 wines is 0.9, indicating that the predictions were generally within 1 point of the actual rating made by Susan. Figure 6 displays the 50 predictions compared to the 50 actual ratings with the green line indicating a perfect prediction, and it is clear that most of the predictions are very close to the line with only 1 prediction being notably incorrect. Table 1 illustrates how the predicted rating was made for one example wine, shown in Row 1, and highlighted in light grey.

Country	Province	Variety	Price	Rating
South Africa	Stellenbosch	Red Blend	10-19.99	5.0
South Africa	Stellenbosch	Merlot	10-19.99	5.0
South Africa	Paarl	Red Blend	10-19.99	5.5
South Africa	Stellenbosch	Pinotage	10-19.99	5.5
South Africa	Stellenbosch	Chenin Blanc	10-19.99	5.0
South Africa	Paarl	Red Blend	10-19.99	4.5
South Africa	Stellenbosch	Sauvignon Blanc	10-19.99	4.0
South Africa	Western Cape	Red Blend	10-19.99	6.0
South Africa	Stellenbosch	Chardonnay	10-19.99	5.0
South Africa	Stellenbosch	Red Blend	40-49.99	5.0
South Africa	Stellenbosch	Cabernet Sauvignon	10-19.99	4.5

Table 1: Neighboring Wines in Cosine Similarity Rating Prediction

The wines shown below the target wine are the 10 closest "neighbors" with the highest cosine similarity to the target wine for which we want to make a prediction. It can be seen that all of the neighbor wines have 3 attributes which match the target wine, thus making sense logically as to why they have the highest cosine similarity. The resulting prediction in this case based on the average rating of the 10 neighbor wines is a perfectly correct 5.0, which matches the actual rating made by Susan.

Classification Models

We performed four different classification models including: Logistic Regression, Decision Tree Classifier, Random Forest Classifier, and K-Nearest Neighbors. Additional pseudo code and documentation about each of these 4 classifiers can be found in the Appendix section. In each of these classification models, we encoded the points feature to 0 and 1, where 1 represents points greater than or equal to 88, and 0 represents points less than 88. Since the classification modeling has applications for a content-based recommender system, we filtered our dataset on our target user of 'Roger Voss', similar to the cosine similarity methodology. It is important to note that for classification modeling in a content-based system, the model must be run separately for each distinct user in the system. Each user has distinct tastes and preferences, therefore it would be incorrect to train a model with wine characteristics and ratings from different users. Our X variable, or model input, is variety and location, both of which have been one hot encoded, and price which has been log transformed. We explored using country instead of location, and price bucket instead of numeric price, but we found that neither of those features improved the modeling results compared to the location and numeric price input variables. Our Y variable, or our target, is outputted as a 1 or a 0, 1 representing 'Yes, recommend', and 0 representing 'No, do not recommend'. We performed an 80/20 train/test for Roger Voss' 18,068 reviews and used these train and test sets for our four different classification models to compare results.

Metrics used for evaluating our machine learning model classification performance are a confusion matrix, accuracy, precision, recall, F1-score, and Receiver Operating Characteristics (ROC) Curve and Area Under the Curve (AUC), although we prioritize accuracy, precision, and AUC for model comparison. A confusion matrix clearly defines true positives, false positives, false negatives, and true negatives. From those, we can obtain the remaining metrics which help quantify the model's correctness, ability to capture positive instances and exclude negative ones as well as the strengths and weaknesses of the model. ROC-AUC is a good visualization for a model's tradeoff between true positive rate and false positive rate.
Our ground truth would be checking to see if the wines that are marked as having a score of greater than 88 points were actually scored that way. In this context, we'd like to minimize false positives as much as possible. This is because we'd like to avoid recommending any "bad" wines as "good"; according to our definition of a "good" wine having a score of greater than 88 points.

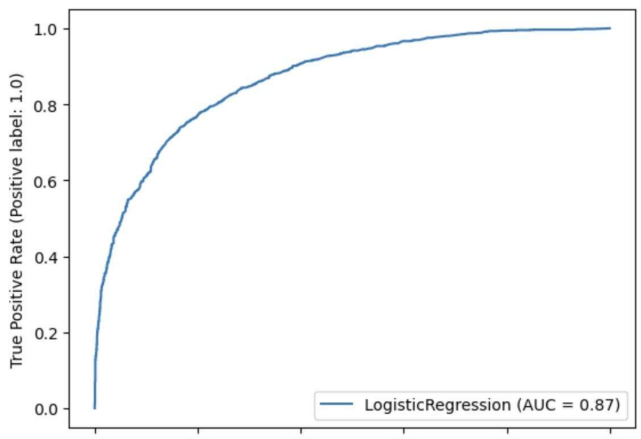


Figure 7: Logistic Regression ROC Curve

Starting with the logistic regression, we performed no hyperparameter tuning and our model resulted in an accuracy of 0.79, a precision of 0.79, and the AUC of this model is 0.87. The ROC Curve for the logistic regression model is shown in Figure 7. Additionally, the confusion matrix for the logistic regression model is shown below in Table 2.

	Predicted Positive	Predicted Negative
Actual Positive	1535	318
Actual Negative	413	1231

Table 2: Logistic Regression Confusion Matrix

Logistic Regression is a widely used classification algorithm that models the probability of a binary outcome. In our case, the binary outcome indicates whether a user would enjoy a wine or not. The logistic regression model learns the relationships between the input features (such as price, variety, and location) and the binary outcome (enjoyment prediction) by finding the best-fitting S-shaped logistic curve. The logistic regression model showed a consistent and balanced performance across various evaluation metrics. It accurately predicted whether a user would enjoy a wine with reasonable accuracy, precision, specificity, and AUC.

Moving on to the decision tree classifier, we did hyperparameter tuning using grid search cross-validation technique with 10 folds to get the optimal parameters for this algorithm. The resulting parameters are criterion of gini, a max_depth of 25, and min_samples_split of 150. Our model resulted in an accuracy of 0.78, a precision of 0.79, and an AUC of 0.86, with the ROC curve shown below in Figure 8 and the confusion matrix directly below in Table 3.

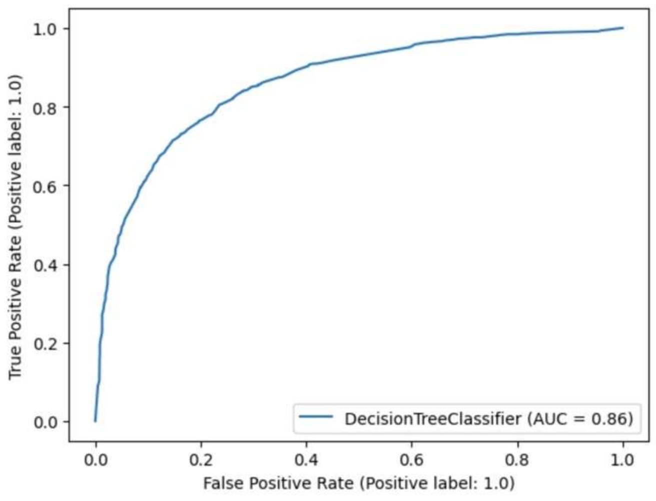


Figure 8: Decision Tree ROC Curve

	Predicted Positive	Predicted Negative
Actual Positive	1490	363
Actual Negative	394	1250

Table 3: Decision Tree Confusion Matrix

The Decision Tree Classifier is a non-linear algorithm that splits the data into subsets based on the values of input features. It creates a tree-like structure where each internal node represents a feature, each branch represents a decision based on that feature, and each leaf node represents a class label (in this case, wine enjoyment). The decision tree classifier achieved similar results to the logistic regression model. It effectively captured decision boundaries within the feature space, enabling it to provide reliable predictions. Overall, this model performed similar to the logistic regression but slightly worse by comparison.

Next, we implemented a random forest classifier using hyperparameter tuning with grid search cross-validation once again. Since the random forest classifier is a more complex model than decision tree, we only used 3 folds to make the cross validator less computationally demanding. The resulting parameters for the random forest model are criterion of entropy, max_depth of 25, min_samples_split of 75, and n_estimators of 200. Overall, the model performed extremely similar to the decision tree classifier, with an accuracy of 0.79, a precision of 0.79, and an AUC of 0.86. The ROC Curve is provided below in Figure 9 and the confusion matrix is visible in Table 4.

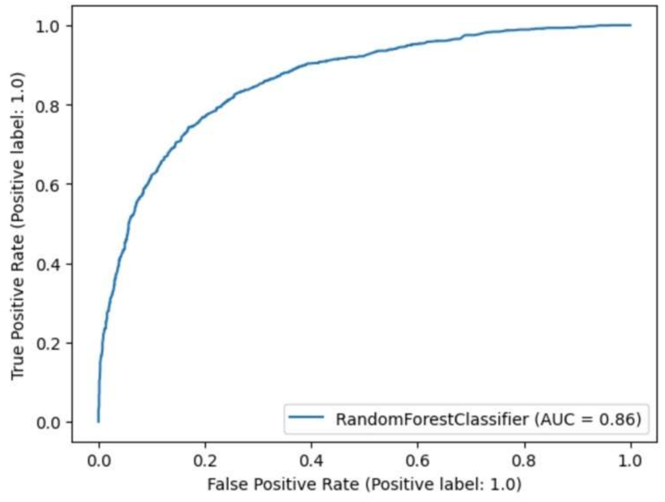


Figure 9: Random Forest ROC Curve

	Predicted Positive	Predicted Negative
Actual Positive	1502	351
Actual Negative	400	1244

Table 4: Random Forest Confusion Matrix

The Random Forest Classifier is an ensemble method that creates multiple decision trees and combines their predictions to improve accuracy and reduce overfitting. Each tree is built on a random subset of the data and a random subset of features. The random forest classifier delivered results consistent with the other models. Unfortunately, we were hoping that given the increased complexity of this model there would be improved results, but we found that model performance could not be significantly improved using the random forest model.

Finally, we implemented a K-Nearest Neighbors classification model and used grid search cross-validation with 10 folds to find the best number of neighbors, which turned out to be 27. Once again, the results of this model were extremely similar to all of the other classification models, with an accuracy of 0.78, precision of 0.80, and AUC of 0.86. The ROC Curve and confusion matrix are shown below in Figure 10 and Table 5.

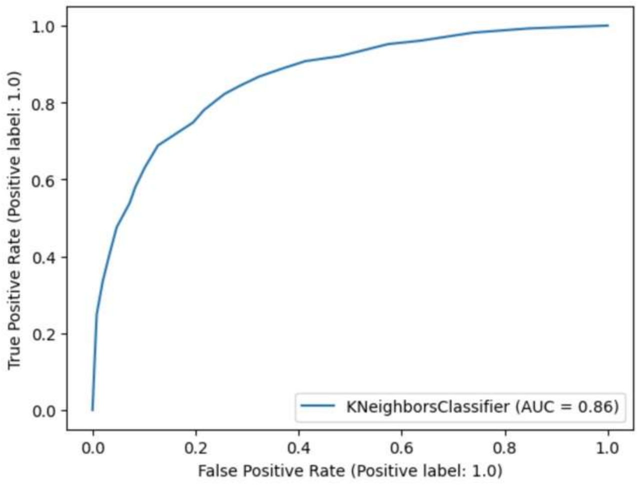


Figure 10: K-Nearest Neighbors ROC Curve

	Predicted Positive	Predicted Negative
Actual Positive	1452	401
Actual Negative	361	1283

Table 5: K-Nearest Neighbors Confusion Matrix

K-Nearest Neighbors (KNN) is a simple yet effective algorithm that classifies data points based on the class labels of their K nearest neighbors. The prediction is made by majority voting among the K neighbors. The K-Nearest Neighbors model displayed competitive performance in predicting wine enjoyment. It leveraged the similarities between data points to make predictions, and its accuracy, precision, specificity, and F1-score were on par with the other models.

Overall, despite extremely similar model performance, we declared Logistic Regression to be the best model due to the highest AUC being provided by this model. The accuracy of 0.79 is very good, and it would be quite effective at predicting whether or not a user would enjoy a wine. Additionally, the model had a similar false positive rate to the rest of the models while also providing the fewest false negative ratings, indicating that it did not 'miss' predicting enjoyable wines for the target user as often as the other models.

Regression Models

The regression models we have performed are similar to the classification models, except here, we are predicting a numeric rating that would be made for all wines instead of just a Yes/No recommendation. Although the target output variable is now numeric instead of categorical, the input variables remain the same as the classification models and are log-transformed price, variety, and location. It is worth noting once again that we explored using categorical price buckets and country instead of our created location attribute, however once again the regression modeling results were better with numeric price and the location attribute. Since the regression modeling would be used for a content-based recommender system, the steps taken are very similar to the classification models. We selected the same target user, Roger Voss, and split his 18,068 reviews in an 80/20 train-test split to evaluate how each regression model performs. The regression models we implemented are Linear Regression, Decision Tree Regressor, and Random Forest Regressor.

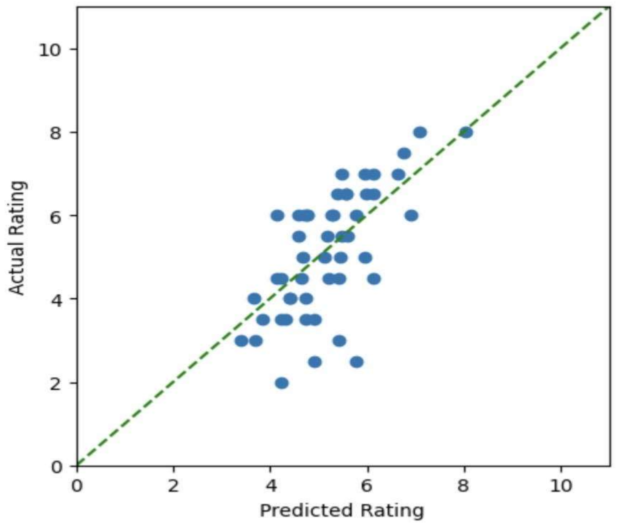


Figure 11: Linear Regression Predictions vs. Actual Rating

Starting with linear regression, we did not perform any hyperparameter tuning which resulted in an r-squared value of 0.58 and a root mean-squared error (RMSE) of 1.01. In Figure 11, we have shown a sample of 50 wine ratings and the model's predicted rating.

Overall, although this is the simplest regression model, the linear regression did not perform as well as we had hoped. The model provided insights into predicting wine ratings, but it had its limitations. An R-squared value of 0.58 indicates that approximately 58% of the variance in wine ratings was captured by the model. An RMSE of 1.01 indicates that this model was generally unable to predict a wine rating within a point of the actual rating. While it offers a basic understanding of how input features affect wine ratings, it might not capture the complexities of the data fully.

For our decision tree regressor, we used grid search cross-validation for hyperparameter tuning and found the best parameters to be max_depth of 28 and min_samples_split of 150. The decision tree essentially performed exactly the same as the linear regression, with an r-squared of 0.58 and an RMSE of 1.01. Figure 12 displays a sample of 50 predicted ratings compared to the actual ratings.

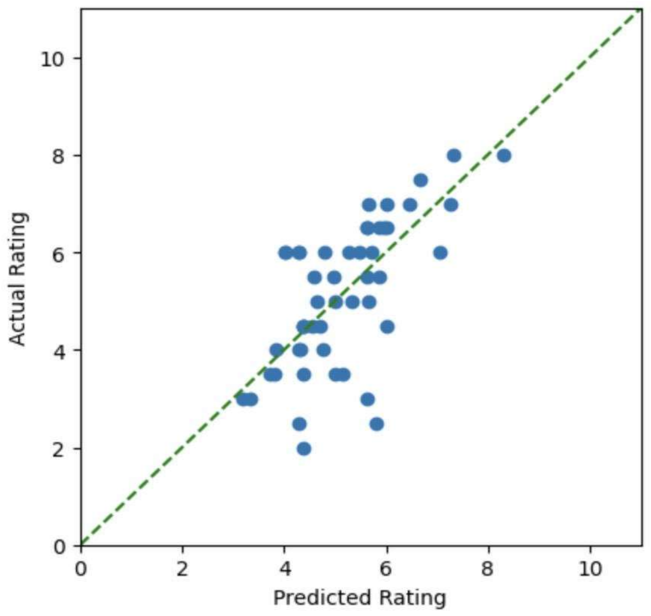


Figure 12: Decision Tree Predictions vs. Actual Rating

The decision tree regressor model performed similarly to the linear regression model. The decision tree regressor exhibited similar limitations to the linear regression model. An R-squared value of 0.58 and an RMSE of 1.01 indicate performance comparable to linear regression. While the decision tree model might capture non-linear relationships between features and ratings better than linear regression, it might still miss some nuances in the data.

Finally, we implemented a random forest regressor with grid search cross-validation for hyperparameter tuning. The resulting best parameters were max_depth of 25, min_samples_split of 150, and n_estimators of 200. We were very happy to see that this more complex model was able to boost performance compared to linear regression and decision tree regression, providing an r-squared of 0.60 and an RMSE of 0.98. The improvements made by the random forest regressor model indicate better capability of this model predicting the actual rating for any given wine, and in general it was able to predict a wine rating within 1 point of the actual rating. We have provided a scatter plot to visualize the predicted rating of 50 sample wines compared to the actual rating in Figure 13.

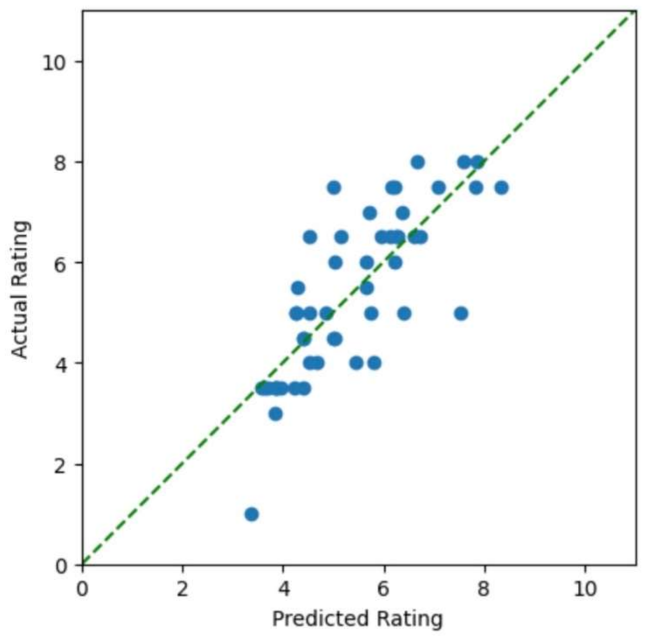


Figure 13: Random Forest Predictions vs. Actual Rating

Although it may not be extremely noticeable compared to the other two scatterplots for linear regression and decision tree regression, the predicted ratings for the random forest regressor shown in Figure 13 has many more predictions laying on or very close to the 'perfect prediction' line with only one outlier prediction that is very incorrect compared to the others. Given the random forest regressor provided the highest r-squared value and lowest RMSE compared to the other two regression models, it is easy to declare the random forest as the best performing regression model.

7. Applications in Recommender Systems

Content-Based System

Each of the aforementioned predictive modeling approaches can be used in a content-based recommender system to provide user-specific recommendations based on their wine review history. Content-based recommenders require historical user reviews in order to make recommendations, therefore this type of system would not be usable for brand new users who have not reviewed a wine before. The cosine similarity method, which take a wine's country, province, variety, and price, would be able to provide recommendations to a user based on the similarity calculation between a new, un-reviewed wine and all reviewed wines by the user. For example, if the system were to be provided with 10 wines that a user has not yet reviewed, a cosine similarity calculation could be made between each new wine and all reviewed wines, and the average rating from x number of neighbors with the highest cosine similarity could then be used to rank the un-reviewed wines in order from highest to lowest predicted rating (in our testing we found it best with 10 neighbors). The input for the cosine similarity system would be all of a user's wine reviews, and a database of many wines and their characteristics. The output of this system would be a list of recommended, un-reviewed wines based on the highest average rating of already reviewed neighbor wines.

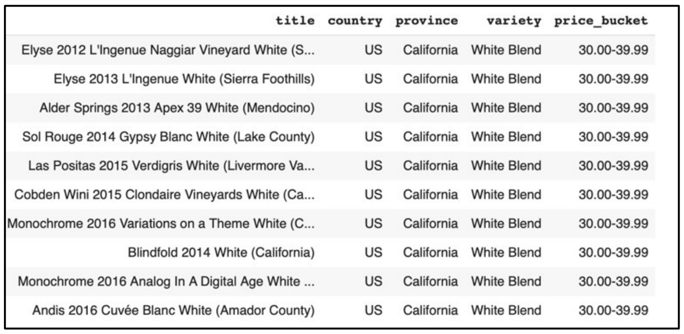
The classification content-based system works slightly differently from the cosine similarity system in that it takes fewer content input characteristics and only seeks to predict Yes or No for whether a user would like a wine. Unlike the cosine similarity system, we could only use the location attribute for classification modeling since the province attribute is a sub-category of country, therefore there would be perfect multicollinearity if each of these attributes were used as input (i.e., every wine with province of "California" would have country of "USA"). Just like the cosine similarity system, the classification technique would also need to filter the input dataset only for reviews from a single user, and in practice the model would be fit with all of a user's reviews. Then, the model could make predictions would be made for all un-reviewed wines in the database and rank the ratings based on highest average rating by other users who have wine reviews in the database. Although the overall compiled list of recommended wines would be 80% accurate, the ranking technique could potentially result in recommendations being provided in an order that would not match user preferences, therefore it would be best to employ this technique in a hybrid fashion with the knowledge-based recommender described in a later section.

Finally, the regression content-based system works quite similarly to the classification system, except the target variable is numeric instead of categorical. The input variables for a regression system are the exact same as the classification system, but the fact that the system would be predicting a numeric rating likely gives the regression system the upper hand to classification in terms of ranking of wine recommendations. Unlike the classification system which would filter out wines that the system predicts a '0' (aka 'Bad') and then provides recommendation based on average rating of *other* users, the regression system would not necessarily filter out any wines but instead present recommendations in order of highest predicted rating. The system would be fit with all of a user's reviews and make numeric predictions for un-reviewed wines. Similar to the classification system, it would also be beneficial to use the regression system in a hybrid fashion.

Knowledge-Based Recommender

As described in the previous section, although content-based systems are very helpful for users with extensive wine review history, for newer users with little to no wine review history, it is impossible for the system to make accurate recommendations, this is referred to as the cold-start problem. A knowledge-based system is the perfect way to combat this issue, and there are two different subtypes of knowledge-based systems which can be created: case-based and constraint-based.

A case-based system requests and input wine from a user, automatically extracts the wine characteristics, and calculates the cosine similarity between the wine and other wines in the database. For example, if a user were to input a wine they enjoyed with title 'Las Positas 2014 Verdigris White' from country USA, province California, variety White Blend, and price bucket of $30-$39.99, the case-based system could make recommendations shown in Figure 14 which have a perfect cosine similarity rating.



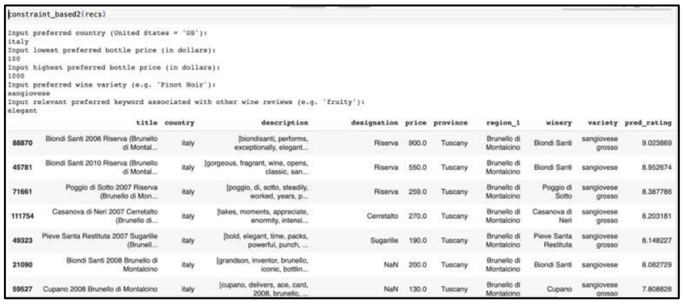
title	country	province	variety	price_bucket
Elyse 2012 L'Ingenue Naggiar Vineyard White (S...	US	California	White Blend	30.00-39.99
Elyse 2013 L'Ingenue White (Sierra Foothills)	US	California	White Blend	30.00-39.99
Alder Springs 2013 Apex 39 White (Mendocino)	US	California	White Blend	30.00-39.99
Sol Rouge 2014 Gypsy Blanc White (Lake County)	US	California	White Blend	30.00-39.99
Las Positas 2015 Verdigris White (Livermore Va...	US	California	White Blend	30.00-39.99
Cobden Wini 2015 Clondaire Vineyards White (Ca...	US	California	White Blend	30.00-39.99
Monochrome 2016 Variations on a Theme White (C...	US	California	White Blend	30.00-39.99
Blindfold 2014 White (California)	US	California	White Blend	30.00-39.99
Monochrome 2016 Analog In A Digital Age White ...	US	California	White Blend	30.00-39.99
Andis 2016 Cuvée Blanc White (Amador County)	US	California	White Blend	30.00-39.99

Figure 14: Case-Based Recommendations

A constraint-based system is another type of knowledge-based recommender which takes user input characteristics and provides output of wines which match the user specifications. We have created a simple knowledge-based recommender which takes country, variety, price range, and a keyword associated with the wine as input, and provides a list of relevant wines from the database which match the criteria.

Hybrid System

Due to some of the limitations described in the classification and regression sub-sections of the content-based recommender system, we found it very helpful to incorporate a knowledge-based recommender system to filter the large number of results provided by either machine learning technique. For the user Roger Voss, our regression system made numeric predictions for 92,552 un-reviewed wines in the database, which would be nearly impossible for the user to fully examine. By including a constraint-based recommender system along with the content-based regression technique, Roger could successfully filter the recommendations based on criteria of his choosing. Figure 15 displays how Roger could input additional criteria and receive recommendations of relevant wines ranked by predicted rating.



```
constraint_based2(recs)
Input preferred country (United States = 'US'):
italy
Input lowest preferred bottle price (in dollars):
100
Input highest preferred bottle price (in dollars):
1000
Input preferred wine variety (e.g. 'Pinot Noir'):
sangiovese
Input relevant preferred keyword associated with other wine reviews (e.g. 'fruity'):
elegant
```

	title	country	description	designation	price	province	region_1	winery	variety	pred_rating
88870	Biondi Santi 2006 Riserva (Brunello di Montal...	italy	[biondisanti, performs, exceptionally, elegant...	Riserva	900.0	Tuscany	Brunello di Montalcino	Biondi Santi	sangiovese grosso	9.023869
45781	Biondi Santi 2010 Riserva (Brunello di Montal...	italy	[gorgeous, fragrant, wine, opens, classic, san...	Riserva	550.0	Tuscany	Brunello di Montalcino	Biondi Santi	sangiovese	8.952674
71661	Poggio di Sotto 2007 Riserva (Brunello di Mon...	italy	[poggio, di, sotto, steadily, worked, years, p...	Riserva	259.0	Tuscany	Brunello di Montalcino	Poggio di Sotto	sangiovese grosso	8.387786
111754	Casanova di Neri 2007 Cerretalto (Brunello di...	italy	[takes, moments, appreciate, enormity, intensi...	Cerretalto	270.0	Tuscany	Brunello di Montalcino	Casanova di Neri	sangiovese grosso	8.203181
49323	Pieve Santa Restituta 2007 Sugarille (Brunell...	italy	[bold, elegant, time, packs, powerful, punch, ...	Sugarille	190.0	Tuscany	Brunello di Montalcino	Pieve Santa Restituta	sangiovese grosso	8.148227
21090	Biondi Santi 2008 Brunello di Montalcino	italy	[grandson, inventor, brunello, iconic, bottlin...	NaN	200.0	Tuscany	Brunello di Montalcino	Biondi Santi	sangiovese	8.082729
59527	Cupano 2008 Brunello di Montalcino	italy	[cupano, delivers, ace, card, 2008, brunello, ...	NaN	130.0	Tuscany	Brunello di Montalcino	Cupano	sangiovese grosso	7.808826

Figure 15: Hybrid Recommender System

The hybrid approach for content-based recommendations would allow for the benefits of the user-specific rating predictions from the content-based system along with the customizability of results provided in the constraint-based system.

The same idea could be applied to recommendations made by a classification system, however the ranking of results would still likely not be as accurate as the regression-based system since wines would be ranked by *other* user ratings, and it is important to note that taste preferences are different for every user.

8. Limitations

Although we have been able to successfully build multiple different types of recommender systems with a high level of accuracy in their predictions, there are some limitations caused by the dataset that we obtained. To begin, the dataset only contains reviews as recent as 2017 and is therefore relatively out of date. There are many wines produced between 2017 and now which would not be part of the database and therefore would be impossible to recommend. Additionally, there are only 19 unique users in our data, and to make matters worse there are very few wines reviewed by more than 1 user, making it impossible to implement a collaborative filtering system. A collaborative filtering system calculates similar users based on their review history, and for wines that a certain user has not yet reviewed, predicted ratings are made using the ratings of other similar users. It would have been preferable to have a dataset which had the same wine reviewed by many different users so that a collaborative filtering system could be created. It would also have been helpful for the data to contain more variables describing the content of the wine. Although there is a description feature, we discovered this was input by the user when making the review, and it therefore could not be considered an official characteristic of the wine. Additionally, we discovered that some other features such as designation and winery were unusable due to their high number of unique values and lack of standardization when being input. By having standardized features such as official description, red/white/pink wine classification, year of production, and other identifying attributes, we would have been able to provide our machine learning algorithms with more input variables to be used for training and model fitting.

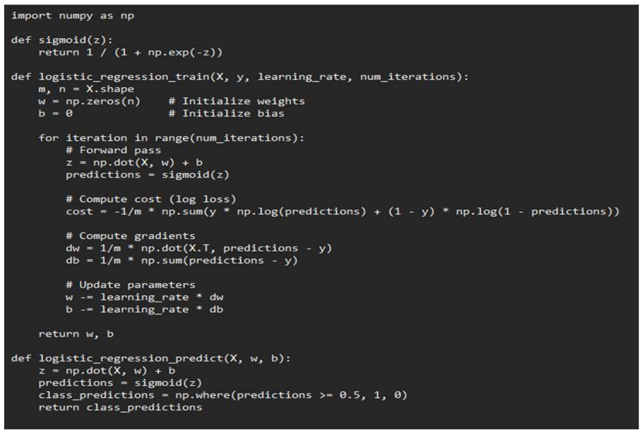
9. Conclusion

Overall, we found this project to be successful in providing wine recommendations for users based on the dataset. Although there were some limitations described in the previous section which prevented us from creating another type of recommender system (collaborative filtering) and some issues with the raw data missing certain information or containing flawed variables, we still think that the content-based and knowledge-based recommender systems provide effective wine recommendations which are applicable for both new and experienced wine consumers. The cosine similarity calculations provided the most accurate predicted ratings for our content-based system; however, this is the most computationally demanding method which would require a lot of time to make recommendations in the real world. A RMSE of less than 1 for both the cosine similarity and random forest regressor methods is quite impressive, and it would be generally agreed that predicting a rating within a point of the actual rating is an accurate and high-quality way of making recommendations. Each of the classification systems we created performed similarly well, but we were happiest with the logistic regression method and think that would be best in a real-world setting. Using a hybrid method to combine either of the classification or regression techniques with the constraint-based knowledge recommender technique would be the best approach, however we believe that the numeric predictions made by the regression system would likely be more valuable compared to the simple yes/no prediction made by the classification system. The creation of the two types of knowledge-based recommender systems makes our system usable for novice and experienced users, and overall we believe this system could accurately recommend a wine to any potential user depending on what they are seeking to accomplish.

10. Appendix

In this section, we provided pseudo code and in-depth explanations of the “behind the scenes” workings of the machine-learning methods described in this paper.

Logistic Regression is a widely used machine learning method for binary classification tasks, predicting the probability of an observation belonging to a specific category. It utilizes the logistic function to model the relationship between input features and the probability of the outcome.

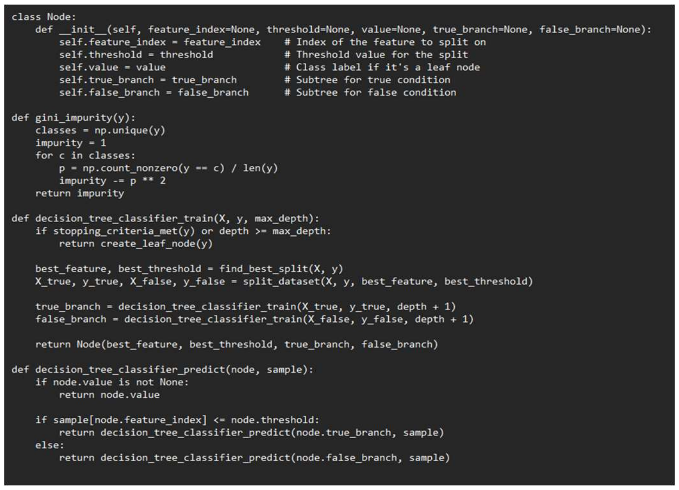


```
import numpy as np

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def logistic_regression_train(X, y, learning_rate, num_iterations):
    m, n = X.shape
    w = np.zeros(n)    # Initialize weights
    b = 0              # Initialize bias

    for iteration in range(num_iterations):
        # Forward pass
        z = np.dot(X, w) + b
        predictions = sigmoid(z)

        # Compute cost (log loss)
        cost = -1/m * np.sum(y * np.log(predictions) + (1 - y) * np.log(1 - predictions))

        # Compute gradients
        dw = 1/m * np.dot(X.T, predictions - y)
        db = 1/m * np.sum(predictions - y)

        # Update parameters
        w -= learning_rate * dw
        b -= learning_rate * db

    return w, b

def logistic_regression_predict(X, w, b):
    z = np.dot(X, w) + b
    predictions = sigmoid(z)
    class_predictions = np.where(predictions >= 0.5, 1, 0)
    return class_predictions
```

Figure 16: Logistic Regression Pseudo Code

By adjusting model parameters through maximum likelihood estimation, it establishes a decision boundary for class separation. Regularization techniques can be applied to prevent overfitting. Known for its interpretability and versatility, logistic regression has numerous applications in finance, healthcare, marketing, amongst others. Pseudo code for this type of modeling is shown in Figure 16.

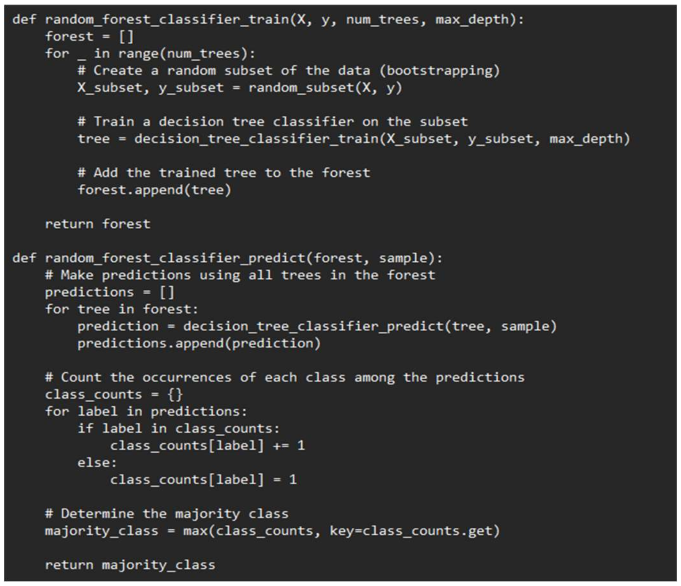
A Decision Tree Classifier is a fundamental machine learning algorithm used for both classification and regression tasks. It works by recursively partitioning the feature space into subsets based on feature values, creating a tree-like structure of decisions. Each internal node represents a feature, each branch represents a decision based on that feature, and each leaf node represents a class label or numerical value. The algorithm selects splits that maximize information gain or minimize impurity at each step. Decision trees are valued for their interpretability and ability to handle complex relationships in data. However, these are prone to overfitting, which can be mitigated using pruning or ensemble methods like Random Forest. Pseudo code for this type of modeling is shown in Figure 17.



```
class Node:
    def __init__(self, feature_index=None, threshold=None, value=None, true_branch=None, false_branch=None):
        self.feature_index = feature_index    # Index of the feature to split on
        self.threshold = threshold            # Threshold value for the split
        self.value = value                    # Class label if it's a leaf node
        self.true_branch = true_branch        # Subtree for true condition
        self.false_branch = false_branch      # Subtree for false condition

def gini_impurity(y):
    classes = np.unique(y)
    impurity = 1
    for c in classes:
        p = np.count_nonzero(y == c) / len(y)
        impurity -= p ** 2
    return impurity

def decision_tree_classifier_train(X, y, max_depth):
    if stopping_criteria_met(y) or depth >= max_depth:
        return create_leaf_node(y)

    best_feature, best_threshold = find_best_split(X, y)
    X_true, y_true, X_false, y_false = split_dataset(X, y, best_feature, best_threshold)

    true_branch = decision_tree_classifier_train(X_true, y_true, depth + 1)
    false_branch = decision_tree_classifier_train(X_false, y_false, depth + 1)

    return Node(best_feature, best_threshold, true_branch, false_branch)

def decision_tree_classifier_predict(node, sample):
    if node.value is not None:
        return node.value

    if sample[node.feature_index] <= node.threshold:
        return decision_tree_classifier_predict(node.true_branch, sample)
    else:
        return decision_tree_classifier_predict(node.false_branch, sample)
```

Figure 17: Decision Tree Classifier Pseudo Code

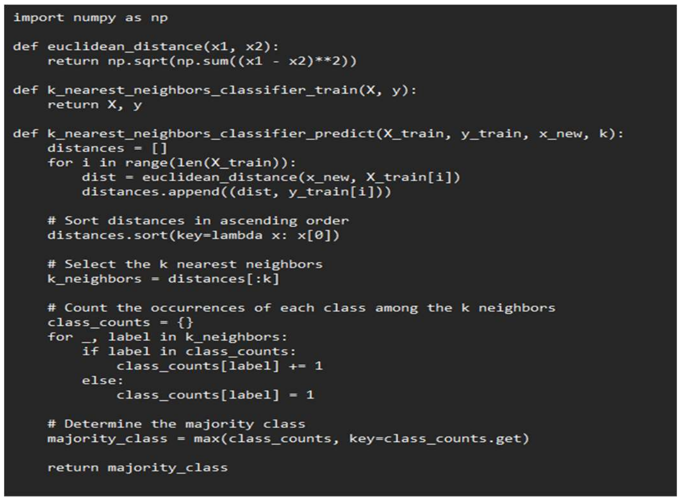
A Random Forest Classifier is a machine learning method that builds multiple decision trees during training and combines their predictions for robust classification. Each tree is constructed using a random subset of the training data and a subset of the features, introducing diversity, and reducing overfitting. The final prediction is determined through a majority vote or averaging of individual tree predictions. Random Forests excel at handling complex relationships, noisy data, and high-dimensional feature spaces. They provide improved accuracy, reduced overfitting, and valuable insights into feature importance. However, their ensemble nature can make them less interpretable than individual decision trees. Pseudo code for this type of modeling is shown in Figure 18.



```
def random_forest_classifier_train(X, y, num_trees, max_depth):
    forest = []
    for _ in range(num_trees):
        # Create a random subset of the data (bootstrapping)
        X_subset, y_subset = random_subset(X, y)

        # Train a decision tree classifier on the subset
        tree = decision_tree_classifier_train(X_subset, y_subset, max_depth)

        # Add the trained tree to the forest
        forest.append(tree)

    return forest

def random_forest_classifier_predict(forest, sample):
    # Make predictions using all trees in the forest
    predictions = []
    for tree in forest:
        prediction = decision_tree_classifier_predict(tree, sample)
        predictions.append(prediction)

    # Count the occurrences of each class among the predictions
    class_counts = {}
    for label in predictions:
        if label in class_counts:
            class_counts[label] += 1
        else:
            class_counts[label] = 1

    # Determine the majority class
    majority_class = max(class_counts, key=class_counts.get)

    return majority_class
```

Figure 18: Random Forest Classifier Pseudo Code

The k-Nearest Neighbors (k-NN) algorithm is a non-parametric and instance-based classification method used for pattern recognition and regression. In k-NN, an observation's class or value is determined by the majority class or the average of the k nearest data points in the feature space. The distance metric, such as Euclidean distance, plays a crucial role in measuring similarity.

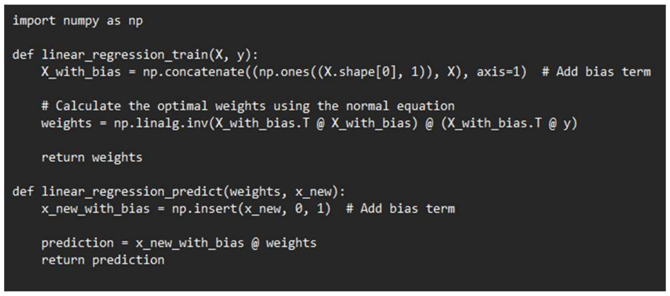


```
import numpy as np

def euclidean_distance(x1, x2):
    return np.sqrt(np.sum((x1 - x2)**2))

def k_nearest_neighbors_classifier_train(X, y):
    return X, y

def k_nearest_neighbors_classifier_predict(X_train, y_train, x_new, k):
    distances = []
    for i in range(len(X_train)):
        dist = euclidean_distance(x_new, X_train[i])
        distances.append((dist, y_train[i]))

    # Sort distances in ascending order
    distances.sort(key=lambda x: x[0])

    # Select the k nearest neighbors
    k_neighbors = distances[:k]

    # Count the occurrences of each class among the k neighbors
    class_counts = {}
    for _, label in k_neighbors:
        if label in class_counts:
            class_counts[label] += 1
        else:
            class_counts[label] = 1

    # Determine the majority class
    majority_class = max(class_counts, key=class_counts.get)

    return majority_class
```

Figure 19: k-NN Classifier Pseudo Code

k-NN is simple to understand and implement, suitable for both small and large datasets, and flexible in handling multi-class problems. However, it can be sensitive to the choice of k, may struggle with high-dimensional data, and requires careful preprocessing to normalize features for accurate distance calculation. Pseudo code for this type of modeling is shown in Figure 19.

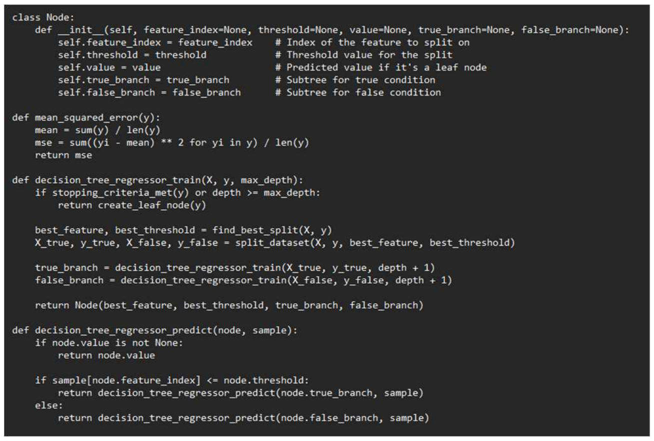
Linear Regression is a fundamental and widely used statistical method employed in machine learning for predicting a continuous numeric outcome. It establishes a linear relationship between input features and the target variable by finding the best-fitting line that minimizes the sum of squared differences between the predicted and actual values. The model's coefficients represent the slope and intercept of this line, providing insights into the feature influences on the target. Linear Regression is interpretable, computationally efficient, and suitable for capturing simple relationships in data. However, it assumes a linear relationship between variables and may not perform well when dealing with complex non-linear patterns. Pseudo code for this type of modeling is shown in Figure 20.



```
import numpy as np

def linear_regression_train(X, y):
    X_with_bias = np.concatenate((np.ones((X.shape[0], 1)), X), axis=1)  # Add bias term

    # Calculate the optimal weights using the normal equation
    weights = np.linalg.inv(X_with_bias.T @ X_with_bias) @ (X_with_bias.T @ y)

    return weights

def linear_regression_predict(weights, x_new):
    x_new_with_bias = np.insert(x_new, 0, 1)  # Add bias term

    prediction = x_new_with_bias @ weights
    return prediction
```

Figure 20: Linear Regression Pseudo Code

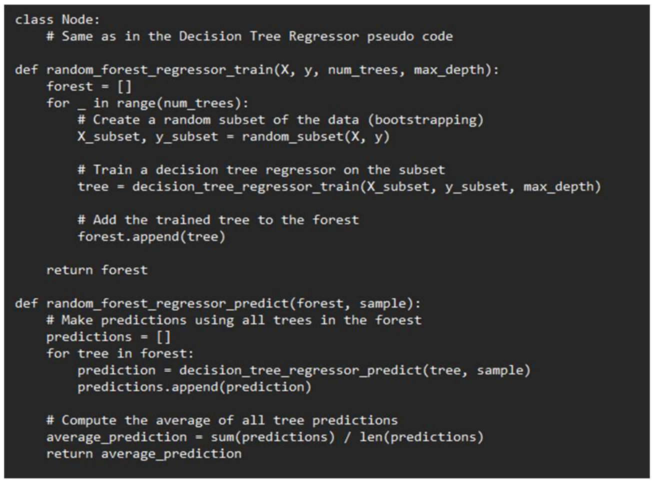
A Decision Tree Regressor is a machine learning algorithm used for regression tasks. It works similarly to the classification version but predicts continuous numeric values instead of categorical labels. The algorithm partitions the feature space into segments based on feature values, creating a tree-like structure of decisions. Each internal node represents a feature, each branch represents a decision based on that feature, and each leaf node represents a predicted numeric value. Decision trees in regression minimize variance by fitting different constant values to each leaf, often the mean of the target values in that region. Decision Tree Regressors are adept at capturing non-linear relationships and handling noisy data. However, they can be sensitive to outliers and overfitting, which can be mitigated using pruning or ensemble methods like Random Forest. Pseudo code for this type of modeling is shown in Figure 21.



```
class Node:
    def __init__(self, feature_index=None, threshold=None, value=None, true_branch=None, false_branch=None):
        self.feature_index = feature_index    # Index of the feature to split on
        self.threshold = threshold            # Threshold value for the split
        self.value = value                    # Predicted value if it's a leaf node
        self.true_branch = true_branch        # Subtree for true condition
        self.false_branch = false_branch      # Subtree for false condition

def mean_squared_error(y):
    mean = sum(y) / len(y)
    mse = sum((yi - mean) ** 2 for yi in y) / len(y)
    return mse

def decision_tree_regressor_train(X, y, max_depth):
    if stopping_criteria_met(y) or depth >= max_depth:
        return create_leaf_node(y)

    best_feature, best_threshold = find_best_split(X, y)
    X_true, y_true, X_false, y_false = split_dataset(X, y, best_feature, best_threshold)

    true_branch = decision_tree_regressor_train(X_true, y_true, depth + 1)
    false_branch = decision_tree_regressor_train(X_false, y_false, depth + 1)

    return Node(best_feature, best_threshold, true_branch, false_branch)

def decision_tree_regressor_predict(node, sample):
    if node.value is not None:
        return node.value

    if sample[node.feature_index] <= node.threshold:
        return decision_tree_regressor_predict(node.true_branch, sample)
    else:
        return decision_tree_regressor_predict(node.false_branch, sample)
```

Figure 21: Decision Tree Regressor Pseudo Code

A Random Forest Regressor is an ensemble learning algorithm designed for regression tasks. It builds multiple decision trees during training and combines their predictions to produce a more accurate and stable final prediction. Each tree is constructed using a random subset of the training data and a subset of the features, introducing diversity, and reducing overfitting. The final prediction from the random forest is typically the average of the individual tree predictions. Random Forest Regressors are effective at capturing complex relationships, handling noisy data, and providing insights into feature importance. They tend to have better generalization performance compared to a single decision tree. However, like any ensemble method, Random Forests might be less interpretable than individual decision trees. Pseudo code for this type of modeling is shown in Figure 22.



```
class Node:
    # Same as in the Decision Tree Regressor pseudo code

def random_forest_regressor_train(X, y, num_trees, max_depth):
    forest = []
    for _ in range(num_trees):
        # Create a random subset of the data (bootstrapping)
        X_subset, y_subset = random_subset(X, y)

        # Train a decision tree regressor on the subset
        tree = decision_tree_regressor_train(X_subset, y_subset, max_depth)

        # Add the trained tree to the forest
        forest.append(tree)

    return forest

def random_forest_regressor_predict(forest, sample):
    # Make predictions using all trees in the forest
    predictions = []
    for tree in forest:
        prediction = decision_tree_regressor_predict(tree, sample)
        predictions.append(prediction)

    # Compute the average of all tree predictions
    average_prediction = sum(predictions) / len(predictions)
    return average_prediction
```

Figure 22: Random Forest Regressor Pseudo Code

References:

1. Zackthoutt. (2017, November 27). Wine reviews. Kaggle. https://www.kaggle.com/datasets/zynicide/wine-reviews
2. About Us. Wine Enthusiast. (2021, May 5). https://www.winemag.com/about-us/
3. Hastie, T., Tibshirani, R., & Friedman, J. (2009). The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer Science & Business Media.
4. Bishop, C. M. (2006). Pattern Recognition and Machine Learning. Springer.
5. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Vanderplas, J. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12(Oct), 2825-2830.
6. Breiman, L., Friedman, J., Stone, C. J., & Olshen, R. A. (1984). Classification and regression trees. CRC press.
7. Scikit-learn: Machine learning in Python. (https://scikit-learn.org/stable/modules/tree.html)
8. Scikit-learn: Machine learning in Python. (https://scikit-learn.org/stable/modules/ensemble.html#random-forests)
9. Scikit-learn: Machine learning in Python. (https://scikit-learn.org/stable/modules/neighbors.html)
10. Montgomery, D. C., Peck, E. A., & Vining, G. G. (2012). Introduction to Linear Regression Analysis. John Wiley & Sons.
11. Scikit-learn: Machine learning in Python. (https://scikit-learn.org/stable/modules/linear_model.html#ordinary-least-squares)
12. Scikit-learn: Machine learning in Python. (https://scikit-learn.org/stable/modules/tree.html#regression)
13. Scikit-learn: Machine learning in Python. (https://scikit-learn.org/stable/modules/ensemble.html#random-forests)