Predicting Wine Ratings

Introduction to Problem & Data

Problem Statement:

Wine is a widely enjoyed alcoholic beverage produced in many countries around the world. For my final project, I plan to build a predictive model that can accurately estimate the Wine Rating of various red wines, which differ across factors like price, country or region of origin, and vintage year. I will explore the trends and patterns in these ratings to offer meaningful insights into what contributes to a higher wine rating. These ratings, ranging from 0 to 5, are assigned by users on Vivino.com and serve as a useful indicator of a wine's taste and overall quality.

Such a model can be particularly valuable for individuals looking to purchase wine but unsure where to begin, or those seeking the best value for a given price point. The predictions will help guide more informed financial decisions, ensuring buyers select wines that are likely to be enjoyable and worth the cost. In addition, wine retailers and marketers can leverage these insights to better position their products, highlight high-performing wines, and tailor marketing strategies to align with consumer preferences. Ultimately, this model aims to give wine buyers actionable insights and greater confidence in their choices.

Dataset Description:

The data for this project comes from Kaggle in CSV format and offers detailed information on a range of wines from Vivino.com. The dataset will need some cleaning and wrangling due to potentially irrelevant columns and the presence of null values. Building an accurate regression model may pose some challenges because of the data's inherent volatility—wine ratings are purely based on user opinions, which are subjective by nature. Still, I expect that certain variables will have predictive power and can help estimate wine ratings with reasonable accuracy.

This dataset provides detailed information on various wines and their ratings, including attributes like name, country of origin, region, winery, price, and vintage year. It consists of 8,666 rows and 8 columns, which I can use as features in predicting wine ratings.

Data Pre-Processing & Preliminary Examination:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import requests
```

```
#load dataset
wine_data = 'Red.csv'
df = pd.read_csv(wine_data)
df.head()
```

	Name	Country	Region	Winery	Rating	NumberOfRatings	Price	Year
0	Pomerol 2011	France	Pomerol	Château La Providence	4.2	100	95.00	2011
1	Lirac 2017	France	Lirac	Château Mont-Redon	4.3	100	15.50	2017
2	Erta e China Rosso di Toscana 2015	Italy	Toscana	Renzo Masi	3.9	100	7.45	2015
3	Bardolino 2019	Italy	Bardolino	Cavalchina	3.5	100	8.72	2019
4	Ried Scheibner Pinot Noir 2016	Austria	Carnuntum	Markowitsch	3.9	100	29.15	2016

Next steps: Generate code with df View recommended plots New interactive sheet

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8666 entries, 0 to 8665
Data columns (total 8 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   Name             8666 non-null   object
```

```
 1   Country          8666 non-null   object
 2   Region           8666 non-null   object
 3   Winery           8666 non-null   object
 4   Rating           8666 non-null   float64
 5   NumberOfRatings  8666 non-null   int64
 6   Price            8666 non-null   float64
 7   Year             8666 non-null   object
dtypes: float64(2), int64(1), object(5)
memory usage: 541.8+ KB
```

```
#drop rows with null/0 values
df = df.dropna()
df = df[(df != 0).all(axis=1)]
df = df[df['Year'] != 'N.V.']
```

```
#final dataset:
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 8658 entries, 0 to 8665
Data columns (total 8 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   Name             8658 non-null   object
 1   Country          8658 non-null   object
 2   Region           8658 non-null   object
 3   Winery           8658 non-null   object
 4   Rating           8658 non-null   float64
 5   NumberOfRatings  8658 non-null   int64
 6   Price            8658 non-null   float64
 7   Year             8658 non-null   object
dtypes: float64(2), int64(1), object(5)
memory usage: 608.8+ KB
```

```
df['Rating'].min()
```

```
2.5
```

```
df['Rating'].max()
```

```
4.8
```

```
df['Price'].min()
```

```
3.55
```

```
df['Price'].max()
```

```
3410.79
```

```
df['Year'].min()
```

```
'1988'
```

```
df['Year'].max()
```

```
'2019'
```

```
df['NumberOfRatings'].max()
```

```
20293
```

```
df['NumberOfRatings'].min()
```

```
25
```

The dataset I will be working with contains detailed information on 8,666 red wines from around the world, with vintages ranging across multiple year from 1988 to 2019. The wines vary in price from as low as $3.55 to over$3,400, and have received between 25 and more than 20,000 ratings between 2.5 and 4.8 on Vivino.

Exploratory Data Analysis

```
df.head()
```

	Name	Country	Region	Winery	Rating	NumberOfRatings	Price	Year
0	Pomerol 2011	France	Pomerol	Château La Providence	4.2	100	95.00	2011
1	Lirac 2017	France	Lirac	Château Mont-Redon	4.3	100	15.50	2017
2	Erta e China Rosso di Toscana 2015	Italy	Toscana	Renzo Masi	3.9	100	7.45	2015
3	Bardolino 2019	Italy	Bardolino	Cavalchina	3.5	100	8.72	2019
4	Ried Scheibner Pinot Noir 2016	Austria	Carnuntum	Markowitsch	3.9	100	29.15	2016

Next steps: Generate code with df View recommended plots New interactive sheet

Descriptive Statistics

```
#mean rating
mean_value = np.mean(df['Rating'])
print(mean_value)
```

```
3.8901478401478404
```

```
#median Rating
df['Rating'].median()
```

```
3.9
```

```
#min Rating
df['Rating'].min()
```

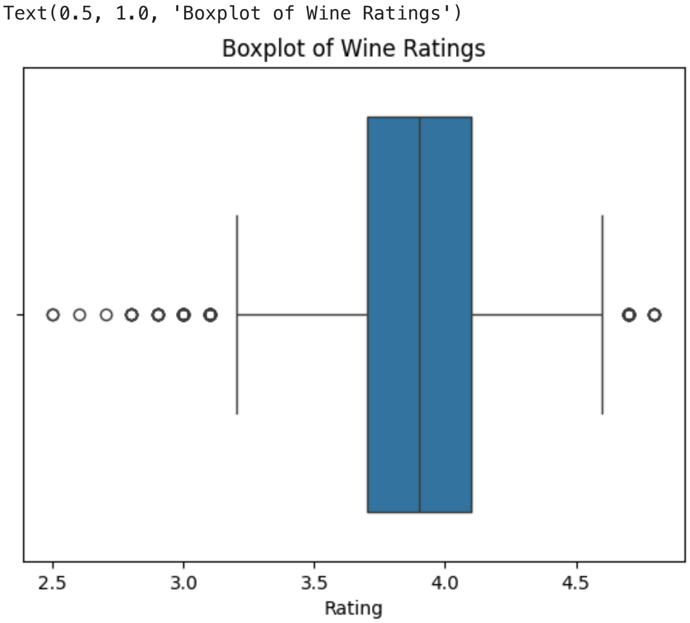
```
2.5
```

```
#max Rating
df['Rating'].max()
```

```
4.8
```

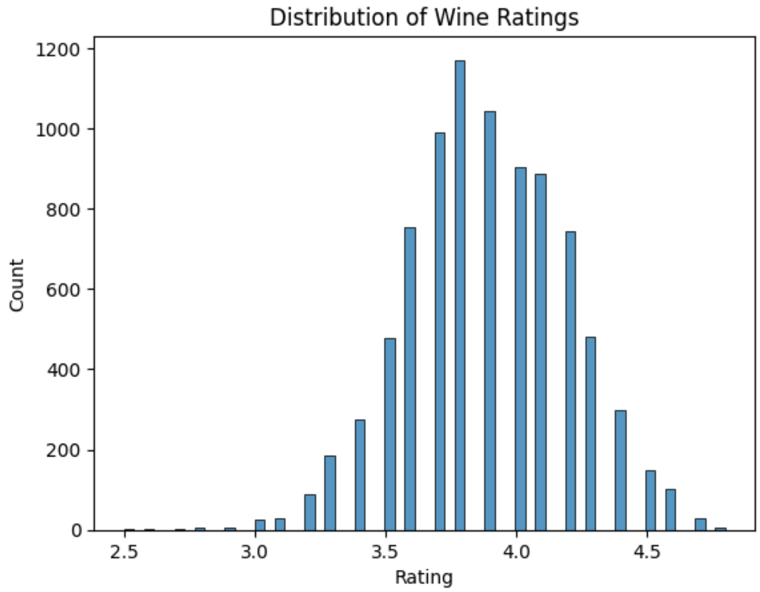
```
#box & whisker plot of Ratings
sns.boxplot(data = df, x = 'Rating')
plt.title('Boxplot of Wine Ratings')
```

```
Text(0.5, 1.0, 'Boxplot of Wine Ratings')
```



```
#histogram of Ratings
sns.histplot(data = df, x = 'Rating')
plt.title('Distribution of Wine Ratings')
```

```
Text(0.5, 1.0, 'Distribution of Wine Ratings')
```

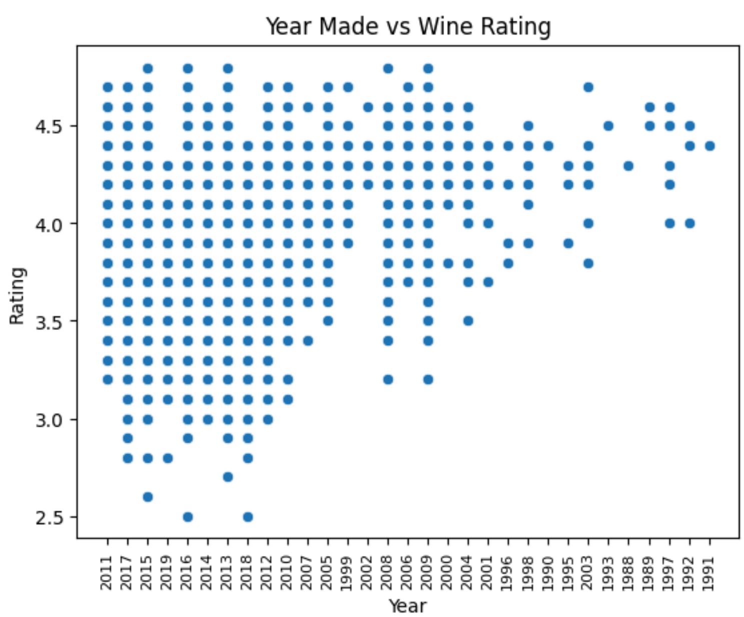



```
# This is formatted as code
```

Based on the distribution shown above, wine ratings appear to follow a roughly normal distribution centered around 3.9 to 4.0. While most wines cluster within this range, ratings span from a minimum of approximately 2.5 to a maximum close to 4.8 out of 5.

Initial Visualizations

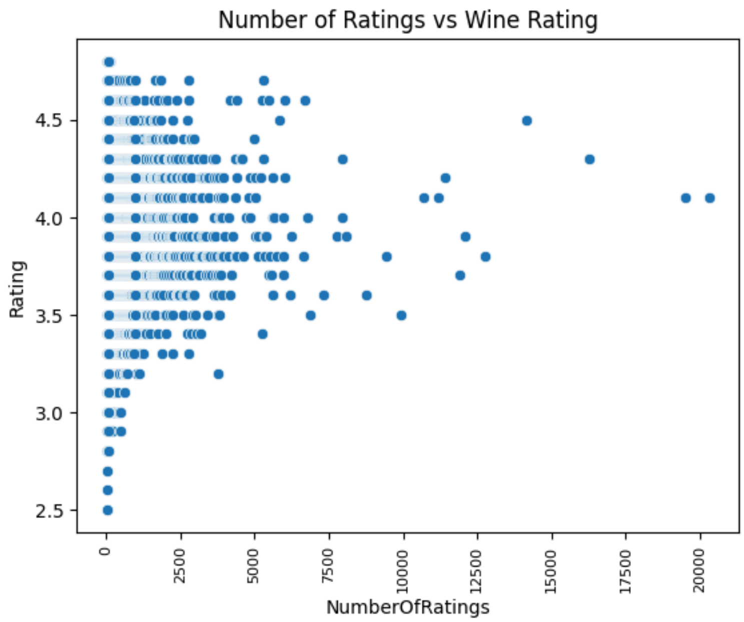
```
#plot year made against rating
sns.scatterplot(data = df, x = 'Year', y = 'Rating')
plt.title('Year Made vs Wine Rating')
plt.xticks(rotation=90, fontsize=8)
plt.figure(figsize=(15, 6))
plt.tight_layout();
```




```
<Figure size 1500x600 with 0 Axes>
```

This scatterplot illustrates the relationship between the year a wine was produced and its corresponding rating. It shows that the majority of wines in the dataset were made after 2005, with a notable concentration in the 2010s. While high ratings appear across all years, older vintages tend to be more sparsely represented but still show consistently strong ratings. This may suggest that more recent wines dominate the market in quantity, while older wines, though fewer, still hold high perceived quality.

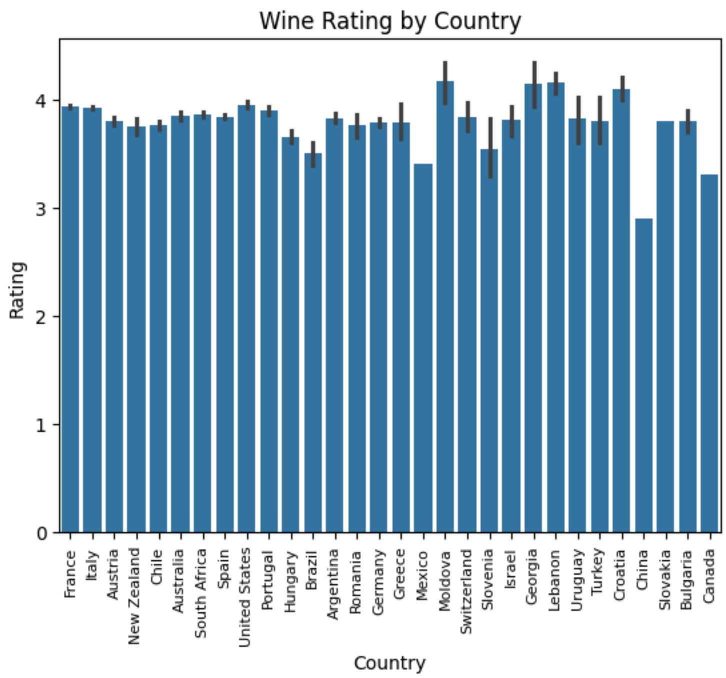
```
#plot Number of Ratings made against rating
sns.scatterplot(data = df, x = 'NumberOfRatings', y = 'Rating')
plt.title('Number of Ratings vs Wine Rating')
plt.xticks(rotation=90, fontsize=8)
plt.figure(figsize=(15, 6))
plt.tight_layout();
```




```
<Figure size 1500x600 with 0 Axes>
```

This scatterplot displays the relationship between the number of user ratings a wine has received and its overall rating. Most wines in the dataset have relatively few ratings, with a sharp drop-off beyond 2,000 reviews. While highly-rated wines appear across all ranges of review counts, there is no strong pattern suggesting that wines with more ratings are rated significantly higher or lower. This may indicate that popularity (as measured by number of ratings) does not necessarily correlate with perceived quality.

```
#plot age country made against wine rating
sns.barplot(data = df, x = 'Country', y = 'Rating')
plt.title('Wine Rating by Country')
plt.xticks(rotation=90, fontsize=8)
plt.figure(figsize=(15, 6))
plt.tight_layout();
```

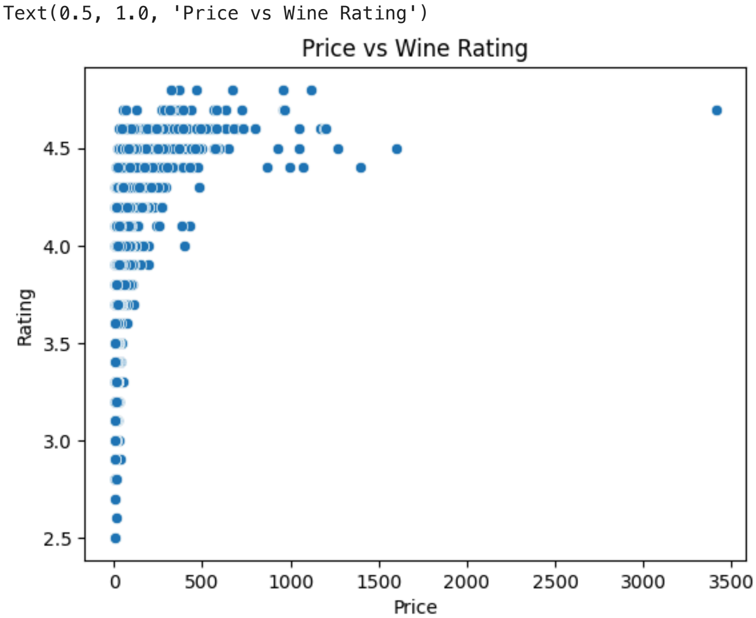



```
<Figure size 1500x600 with 0 Axes>
```

This bar chart displays the average wine rating by country. While most countries have average ratings clustered around 3.8 to 4.1, a few stand out with notably higher or lower scores. Countries like Switzerland, Slovenia, and Georgia show especially high average ratings, whereas countries such as China, Bulgaria, and Canada have lower average ratings in comparison. This suggests that while quality is fairly consistent across many regions, there are some clear differences in perceived wine quality based on country of origin.

```
#plot price against rating
sns.scatterplot(data = df, x = 'Price', y = 'Rating')
plt.title('Price vs Wine Rating')
```

```
Text(0.5, 1.0, 'Price vs Wine Rating')
```



This scatterplot shows the relationship between wine price and rating. While most wines are priced below 500, $afewoutliersreachover$ 3,000. Interestingly, high ratings are observed across nearly the entire price range, suggesting that spending more does not always guarantee a better-rated wine. In fact, many lower-priced wines receive ratings comparable to those of high-end bottles, indicating that good quality can be found at a variety of price points. However, none of the wines priced at 500 or higher have ratings below 5 stars, showing that the more expensive wines do tend to have higher rather than lower ratings.

Modeling & Interpretations

To predict wine ratings, I implemented several different regression models to determine which one performs best in capturing score variability and explaining the patterns in the data. For each model, I applied an 80-20 train-test split—training the model on 80% of the dataset and evaluating its performance on the remaining 20%.

```
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.tree import plot_tree
from sklearn.ensemble import RandomForestRegressor
from sklearn.compose import make_column_transformer
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
from sklearn.pipeline import Pipeline
from sklearn.inspection import permutation_importance
from sklearn.model_selection import GridSearchCV
```

Baseline Model

I assessed the performance of each model by comparing key metrics, such as mean squared error, against a baseline model. This baseline was established by predicting the average wine rating across the entire dataset.

```
#set up baseline model using mean wine rating, calculate baseline mse
y = df['Rating']
baseline_preds = np.ones(len(y))*y.mean()
mean_squared_error(y, baseline_preds)
```

```
0.0951177651610777
```

Multiple Regression Model

I chose to build a multiple regression model to use several independent variables to predict the dependent variable, wine rating, as I believed these factors might collectively impact the score. Multiple linear regression enabled me to analyze both the individual relationships and the combined influence of these predictors on the rating.

```
#create X & y, split into training and testing data
X = df[['Year', 'Country', 'Price']]
y = df['Rating']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 20)


#encode categorical column
cat_col = ['Country']
transformer = make_column_transformer(
    (OneHotEncoder(drop='first', sparse_output=False, handle_unknown='ignore'), cat_col),
    remainder='passthrough'
)


#create pipeline for multiple regression model
pipe = Pipeline([('encode', transformer), ('model', LinearRegression())])


#fit pipeline
pipe.fit(X_train, y_train)
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/compose/_column_transformer.py:1667: FutureWarning:
The format of the columns of the 'remainder' transformer in ColumnTransformer.transformers_ will change in version 1.7 to ma
At the moment the remainder columns are stored as indices (of type int). With the same ColumnTransformer configuration, in t
To use the new behavior now and suppress this warning, use ColumnTransformer(force_int_remainder_cols=False).

  warnings.warn(
```

Pipeline

encode: ColumnTransformer

onehotencoder — OneHotEncoder

remainder — passthrough

LinearRegression

```
#find coefficients
lr = pipe.named_steps['model']
coefficients = lr.coef_
names = transformer.get_feature_names_out()
pd.DataFrame(coefficients, names)
```

	0
onehotencoder__Country_Australia	-0.015372
onehotencoder__Country_Austria	-0.017916
onehotencoder__Country_Brazil	-0.355002
onehotencoder__Country_Bulgaria	0.070278
onehotencoder__Country_Canada	-0.513273
onehotencoder__Country_Chile	-0.052307
onehotencoder__Country_China	-0.938680
onehotencoder__Country_France	0.008103
onehotencoder__Country_Georgia	0.371068
onehotencoder__Country_Germany	-0.044411
onehotencoder__Country_Greece	-0.134829
onehotencoder__Country_Hungary	-0.183963
onehotencoder__Country_Israel	0.054948
onehotencoder__Country_Italy	0.066122
onehotencoder__Country_Lebanon	0.098942
onehotencoder__Country_Moldova	0.482218
onehotencoder__Country_New Zealand	-0.088278
onehotencoder__Country_Portugal	0.060311
onehotencoder__Country_Romania	-0.042239
onehotencoder__Country_Slovakia	-0.040455
onehotencoder__Country_Slovenia	-0.386003
onehotencoder__Country_South Africa	0.034726
onehotencoder__Country_Spain	-0.016708
onehotencoder__Country_Switzerland	0.046835
onehotencoder__Country_Turkey	-0.067641
onehotencoder__Country_United States	0.089419
onehotencoder__Country_Uruguay	0.034663
remainder__Year	-0.020909
remainder__Price	0.001267

```
#find y-int
lr.intercept_
```

```
np.float64(45.946953679979735)
```

```
#calculate mse for training data
y_train_preds = pipe.predict(X_train)
mean_squared_error(y_train, y_train_preds)
```

```
0.07049224450264435
```

```
#calculate mse for testing data
y_test_preds = pipe.predict(X_test)
mean_squared_error(y_test, y_test_preds)
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/preprocessing/_encoders.py:246: UserWarning: Found unknown categories in col
  warnings.warn(
0.06577345386007658
```

```
#determine feature importance
import warnings
from sklearn.inspection import permutation_importance

with warnings.catch_warnings():
    warnings.simplefilter("ignore", category=UserWarning)
    r = permutation_importance(pipe, X_test, y_test, n_repeats=10)

# Now display the importances
importances = pd.DataFrame(
    r['importances_mean'],
    index=X_test.columns,  # only ['Year', 'Country', 'Price']
    columns=['Importance']
)
print(importances)
```

```
         Importance
Year       0.105516
Country    0.051403
Price      0.260250
```

Overall, my multiple regression model outperformed the baseline model. Both the training and testing mean squared errors (0.070 and 0.066, respectively) were lower than the baseline MSE of approximately 0.095, with the training set showing slightly better performance. This suggests that the model was able to capture meaningful patterns in the data and account for variation in wine ratings using the available predictors, rather than simply predicting the average score.

In terms of feature importance, the most influential variable in predicting wine ratings was Price, followed by Year and Country. This indicates that higher wine prices tend to carry more predictive power in estimating ratings, while the year of production and country of origin contribute to a lesser, but still noticeable, extent.

K-Nearest Neighbors Regression Model

I decided to experiment with k-nearest neighbors (KNN) regression next, as this method predicts outcomes based on the similarity between data points in the feature space. Since some of my initial visualizations suggested the presence of local patterns or clusters of similar wines with shared characteristics, KNN seemed well-suited to capture these localized relationships in the data.

```
#create X & y, split into training and testing data
X = df[['Year', 'Country', 'Price']]
y = df['Rating']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 20)
```

```
#encode categorical column & scale data
cat_col = ['Country']
transformer = make_column_transformer(
    (OneHotEncoder(drop='first', handle_unknown='ignore'), cat_col),
    remainder=StandardScaler()
)
```

```
#create pipeline for knn regression model
pipe = Pipeline([
    ('encode', transformer),
    ('model', KNeighborsRegressor())
])


#define grid of hyperparameters for number of neighbors
param_grid = {'model__n_neighbors': [5, 10, 15, 20, 25, 30, 50]}


#perform grid-search w/ cross validation
import warnings
from sklearn.model_selection import GridSearchCV

with warnings.catch_warnings():
    warnings.simplefilter("ignore", category=UserWarning)
    grid_search = GridSearchCV(pipe, param_grid, cv=5, scoring='neg_mean_squared_error')
    grid_search.fit(X_train, y_train)


#determine best parameter
grid_search.best_params_
```

```
{'model__n_neighbors': 15}
```

```
#use 25 neighbors in model
knn = grid_search.best_estimator_


#calculate mse for training data
y_train_preds = knn.predict(X_train)
mean_squared_error(y_train, y_train_preds)
```

```
0.03475616517470401
```

```
#calculate mse for testing data
y_test_preds = knn.predict(X_test)
mean_squared_error(y_test, y_test_preds)
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/preprocessing/_encoders.py:246: UserWarning: Found unknown categories in col
  warnings.warn(
0.038344059532974074
```

```
#determine feature importance
with warnings.catch_warnings():
    warnings.simplefilter("ignore", category=UserWarning)
    r = permutation_importance(knn, X_test, y_test, n_repeats=10)

importances = pd.DataFrame(
    r['importances_mean'],
    index=X_test.columns,
    columns=['Importance']
)
print(importances)
```

```
           Importance
Year         0.069042
Country      0.093953
Price        1.129934
```

My KNN regression model outperformed both the baseline and the multiple linear regression model. While the model performed slightly better on the training set (MSE ≈ 0.035) than on the testing set (MSE ≈ 0.038), the test performance still represented a notable improvement over previous approaches. I believe this is because KNN is capable of capturing non-linear patterns and local relationships in the data—something that may be especially relevant for wine ratings.

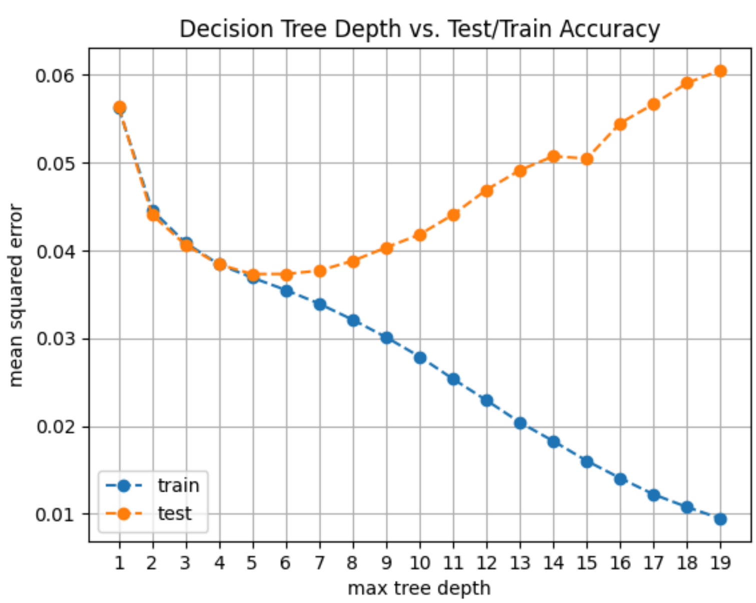
The strong performance may also be attributed to hyperparameter tuning through grid search, which allowed me to identify the optimal number of neighbors for the model, ultimately improving its accuracy.

In this model, Price emerged as by far the most important predictor, followed by Country and Year. This suggests that wine price continues to be the dominant factor in predicting ratings, while regional and vintage information contribute to a lesser, yet meaningful, extent.

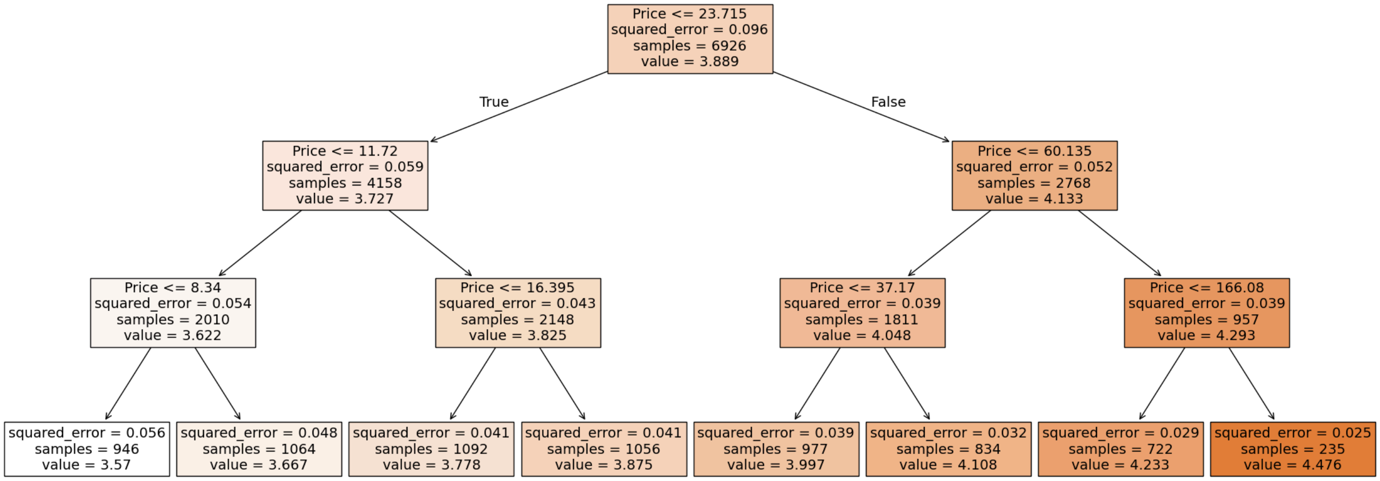
Decision Tree Regression Model

I also chose to build a decision tree regression model because, like k-nearest neighbors, it can capture non-linear relationships within the wine rating data. In addition, decision trees offer a clear and interpretable structure, making it easy to understand how predictions are made based on different feature values. This provides valuable insight into the factors that most strongly influence wine ratings.

```
#create X & y, split into training and testing data
X = df[['Year', 'Country', 'Price']]
y = df['Rating']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 20)


#encode categorical column
cat_col = ['Country']
ohe = OneHotEncoder(sparse_output=False, drop=None, handle_unknown='ignore')

encoder = make_column_transformer(
    (ohe, cat_col),
    remainder='passthrough',
    verbose_feature_names_out=False
)
X_train_encoded = encoder.fit_transform(X_train)
X_test_encoded = encoder.transform(X_test)


#find the optimal max depth while avoiding overfitting by plotting the test accuracies & finding the minimum one
train_scores = []
test_scores = []
for d in range(1, 20):
    dtree = DecisionTreeRegressor(max_depth = d).fit(X_train_encoded, y_train)
    y_train_preds = dtree.predict(X_train_encoded)
    y_test_preds = dtree.predict(X_test_encoded)
    train_scores.append(mean_squared_error(y_train, y_train_preds))
    test_scores.append(mean_squared_error(y_test, y_test_preds))

plt.plot(range(1, 20), train_scores, '--o', label = 'train')
plt.plot(range(1, 20), test_scores, '--o', label = 'test')
plt.grid()
plt.legend()
plt.xticks(range(1, 20))
plt.xlabel('max tree depth')
plt.ylabel('mean squared error')
plt.title('Decision Tree Depth vs. Test/Train Accuracy');
```




```
#fit a decision tree model with a max depth = 3 (lowest mse test score on graph)
dtree = DecisionTreeRegressor(max_depth = 3).fit(X_train_encoded, y_train)


#plot the tree
plt.figure(figsize=(25, 10))
plot_tree(dtree, filled=True, feature_names=encoder.get_feature_names_out().tolist(), fontsize=14);
```




```
#calculate mse for training data
y_train_preds = dtree.predict(X_train_encoded)
mean_squared_error(y_train, y_train_preds)
```

```
0.04085352642724851
```

```
#calculate mse for testing data
y_test_preds = dtree.predict(X_test_encoded)
mean_squared_error(y_test, y_test_preds)
```

```
0.040625448920175285
```

```
#determine feature importance
r = permutation_importance(dtree, X_test_encoded, y_test, n_repeats = 10)
pd.DataFrame(r['importances_mean'], index = encoder.get_feature_names_out().tolist())
```

	0
Country_Argentina	0.000000
Country_Australia	0.000000
Country_Austria	0.000000
Country_Brazil	0.000000
Country_Bulgaria	0.000000
Country_Canada	0.000000
Country_Chile	0.000000
Country_China	0.000000
Country_France	0.000000
Country_Georgia	0.000000
Country_Germany	0.000000
Country_Greece	0.000000
Country_Hungary	0.000000
Country_Israel	0.000000
Country_Italy	0.000000
Country_Lebanon	0.000000
Country_Moldova	0.000000
Country_New Zealand	0.000000
Country_Portugal	0.000000
Country_Romania	0.000000
Country_Slovakia	0.000000
Country_Slovenia	0.000000
Country_South Africa	0.000000
Country_Spain	0.000000
Country_Switzerland	0.000000
Country_Turkey	0.000000
Country_United States	0.000000
Country_Uruguay	0.000000
Year	0.000000
Price	1.127491

While the decision tree regression model performed better than both the baseline and the multiple linear regression model, its performance was slightly weaker than the KNN model. I suspect this is due to the low maximum depth selected for the tree (depth = 3), which may have limited its ability to capture more complex patterns in the data. With such a shallow tree, the model relied almost entirely on Price for its decisions, while completely ignoring Year and Country, which may have affected its predictive accuracy.

In this model, Price was once again the most influential predictor of wine ratings—by a wide margin. Unlike in previous models, however, Year and Country were assigned zero importance, indicating that the tree did not consider them relevant in its final structure. This could be a result of the tree's depth constraint, which prevented it from exploring more nuanced interactions among features.

Random Forest Regression Model

For my final model, I chose to expand on the decision tree approach by building a random forest regression model. Given that the single decision tree showed promising results, I wanted to explore ensemble methods like random forests, which aggregate the predictions of multiple trees to enhance overall predictive accuracy.

```
#create X & y, split into training and testing data
X = df[['Year', 'Country', 'Price']]
y = df['Rating']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 20)
```

```
#encode categorical column
cat_col = ['Country']
encoder = make_column_transformer(
    (OneHotEncoder(drop='first', sparse_output=False, handle_unknown='ignore'), cat_col),
    remainder='passthrough',
    verbose_feature_names_out=False
)
encoder.fit(X_train)
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/compose/_column_transformer.py:1667: FutureWarning:
The format of the columns of the 'remainder' transformer in ColumnTransformer.transformers_ will change in version 1.7 to ma
At the moment the remainder columns are stored as indices (of type int). With the same ColumnTransformer configuration, in t
To use the new behavior now and suppress this warning, use ColumnTransformer(force_int_remainder_cols=False).

  warnings.warn(
```

ColumnTransformer

onehotencoder | **remainder**

OneHotEncoder | passthrough

```
#create pipeline for multiple regression model
forest = Pipeline([
    ('encode', transformer),
    ('model', RandomForestRegressor())
])
```

```
X_test_encoded = encoder.transform(X_test)
forest = pipe.named_steps['model']
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/preprocessing/_encoders.py:246: UserWarning: Found unknown categories in col
  warnings.warn(
```

```
#define grid of hyperparameters for number of estimators and max depth
param_grid = {'model__n_estimators': [50, 100, 150, 200],'model__max_depth': [3, 4, 5, 6, 10]}
```

```
#perform grid-search w/ cross validation
with warnings.catch_warnings():
    warnings.simplefilter("ignore", category=UserWarning)
    grid_search = GridSearchCV(pipe, param_grid, cv=5, scoring='neg_mean_squared_error')
    grid_search.fit(X_train, y_train)
```

```
#determine best parameters
grid_search.best_params_
```

```
{'model__max_depth': 10, 'model__n_estimators': 150}
```

```
#use max depth of 6 & 200 estimators in model
forest = grid_search.best_estimator_
```

```
#calculate mse for training data
y_train_preds = forest.predict(X_train)
mean_squared_error(y_train, y_train_preds)
```

```
0.02570731664358873
```

```
#calculate mse for testing data
y_test_preds = forest.predict(X_test)
mean_squared_error(y_test, y_test_preds)
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/preprocessing/_encoders.py:246: UserWarning: Found unknown categories in col
  warnings.warn(
0.03571500464045164
```

```
pipe.fit(X_train, y_train)
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/compose/_column_transformer.py:1667: FutureWarning:
The format of the columns of the 'remainder' transformer in ColumnTransformer.transformers_ will change in version 1.7 to ma
At the moment the remainder columns are stored as indices (of type int). With the same ColumnTransformer configuration, in t
To use the new behavior now and suppress this warning, use ColumnTransformer(force_int_remainder_cols=False).

  warnings.warn(
```

```
Pipeline
  encode: ColumnTransformer
    onehotencoder        remainder
    OneHotEncoder        StandardScaler
  RandomForestRegressor
```

```
#determine feature importance
with warnings.catch_warnings():
    warnings.simplefilter("ignore", category=UserWarning)
    r = permutation_importance(model, X_test_encoded, y_test, n_repeats=10)
feature_names = encoder.get_feature_names_out()
importances_df = pd.DataFrame(r['importances_mean'], index=feature_names, columns=["Importance"])
print(importances_df.sort_values("Importance", ascending=False))
```

```
                       Importance
Price                    0.086110
Country_Spain            0.027969
Country_Australia        0.003933
Country_Austria          0.000000
Country_Brazil           0.000000
Country_Chile            0.000000
Country_China            0.000000
Country_Georgia          0.000000
Country_Germany          0.000000
Country_Hungary          0.000000
Country_Greece           0.000000
Country_Bulgaria         0.000000
Country_Canada           0.000000
Year                     0.000000
Country_Israel           0.000000
Country_Lebanon          0.000000
Country_Moldova          0.000000
Country_Slovakia         0.000000
Country_Romania          0.000000
Country_New Zealand      0.000000
Country_Portugal         0.000000
Country_Uruguay          0.000000
Country_Slovenia         0.000000
Country_Turkey           0.000000
Country_Switzerland      0.000000
Country_France          -0.002814
Country_United States   -0.005877
Country_Italy           -0.010794
Country_South Africa    -0.012401
```

Overall, my random forest model delivered the strongest performance out of all the models I tested. Although there was a slightly larger gap between the training and testing mean squared errors compared to some previous models, the random forest achieved the lowest MSE on the test set (≈ 0.036), indicating it was the most effective at predicting wine ratings.

In terms of feature importance, Price remained the most significant predictor by a large margin, followed by Country (specifically Spain and Australia). Interestingly, many other countries—including traditionally prominent wine producers—were assigned zero or even negative importance, as was Year, which suggests that the model found little value in these variables for improving prediction accuracy.

Next Steps & Discussion

Could not connect to the reCAPTCHA service. Please check your internet connection and reload to get a reCAPTCHA challenge.