In [94]:
```
# Import packages
### YOUR CODE HERE ###

# For data manipulation
import numpy as np
import pandas as pd

# For data visualization
import matplotlib.pyplot as plt
import seaborn as sns

# For displaying all of the columns in dataframes
pd.set_option('display.max_columns', None)

# For data modeling
from xgboost import XGBClassifier
from xgboost import XGBRegressor
from xgboost import plot_importance

from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier

###  Important imports for modeling and evaluation
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from sklearn.preprocessing import StandardScaler
from statsmodels.formula.api import ols

# For metrics and helpful functions
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.metrics import accuracy_score, precision_score, recall_score,\
f1_score, confusion_matrix, ConfusionMatrixDisplay, classification_report
from sklearn.metrics import roc_auc_score, roc_curve
from sklearn.tree import plot_tree

# For saving models
import pickle
```

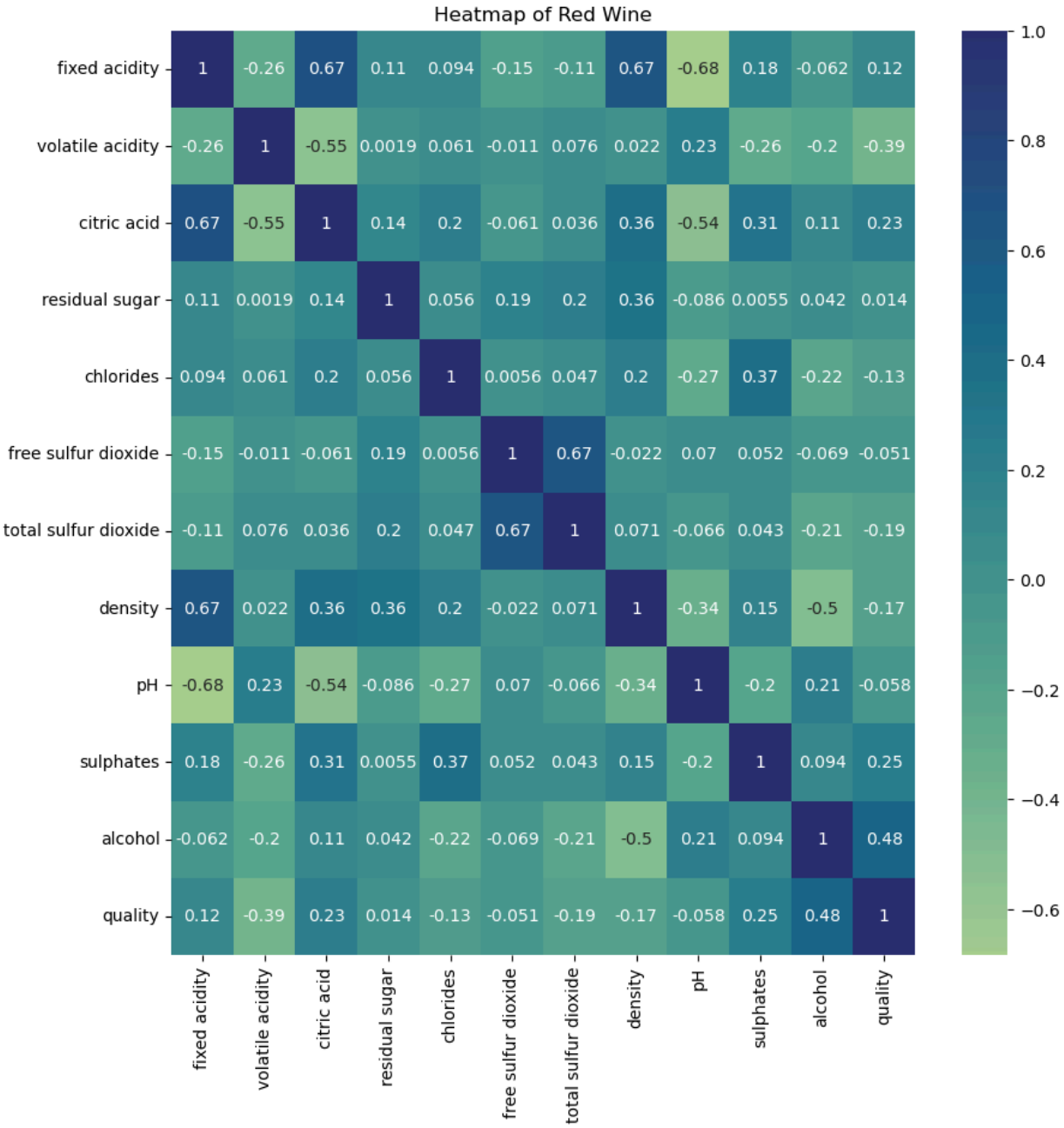
In [95]:
```
dfred = pd.read_csv('/Users/danielyeo/Desktop/wine+quality/winequality-red.csv
```

In [96]:
```
dfred.head()
```

Out[96]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcoh
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.

In [97]:
```
plt.figure(figsize=(10,10))
sns.heatmap(dfred.corr(), annot=True, cmap="crest")
plt.title('Heatmap of Red Wine')
plt.show()
```



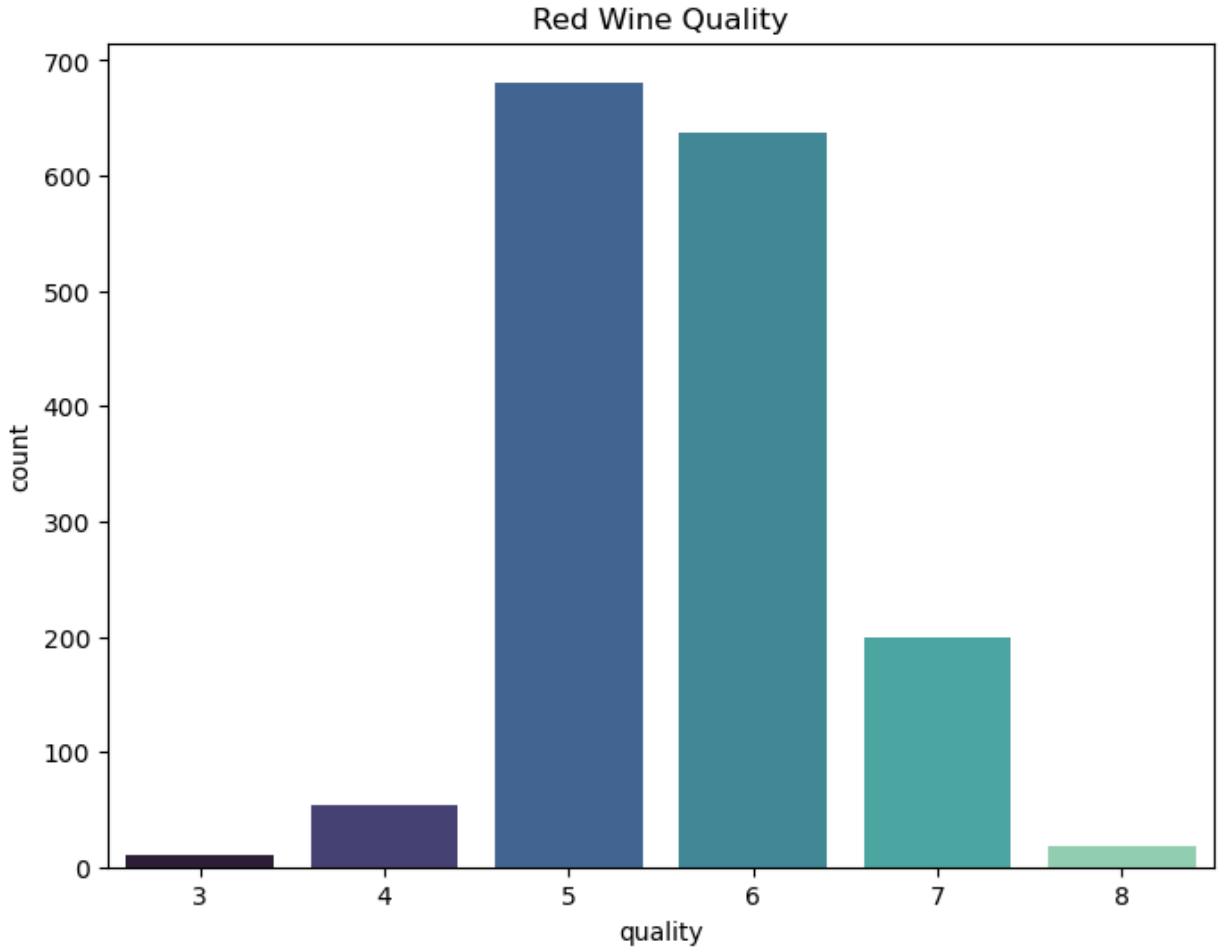
In [98]:
```
#notice that alcohol and quality have one of the highest correlation
dfred[['alcohol', 'quality']].value_counts()
```

Out[98]:
```
alcohol  quality
9.50     5          97
9.40     5          79
9.20     5          50
9.80     5          49
9.30     5          44
                    ..
10.75    6           1
10.70    7           1
         3           1
10.55    7           1
14.90    5           1
Name: count, Length: 190, dtype: int64
```

In [99]:
```
#however most wines hover around average quality
# I want to confirm that alcohol content is not the most significant feature a
dfred['quality'].describe()
```

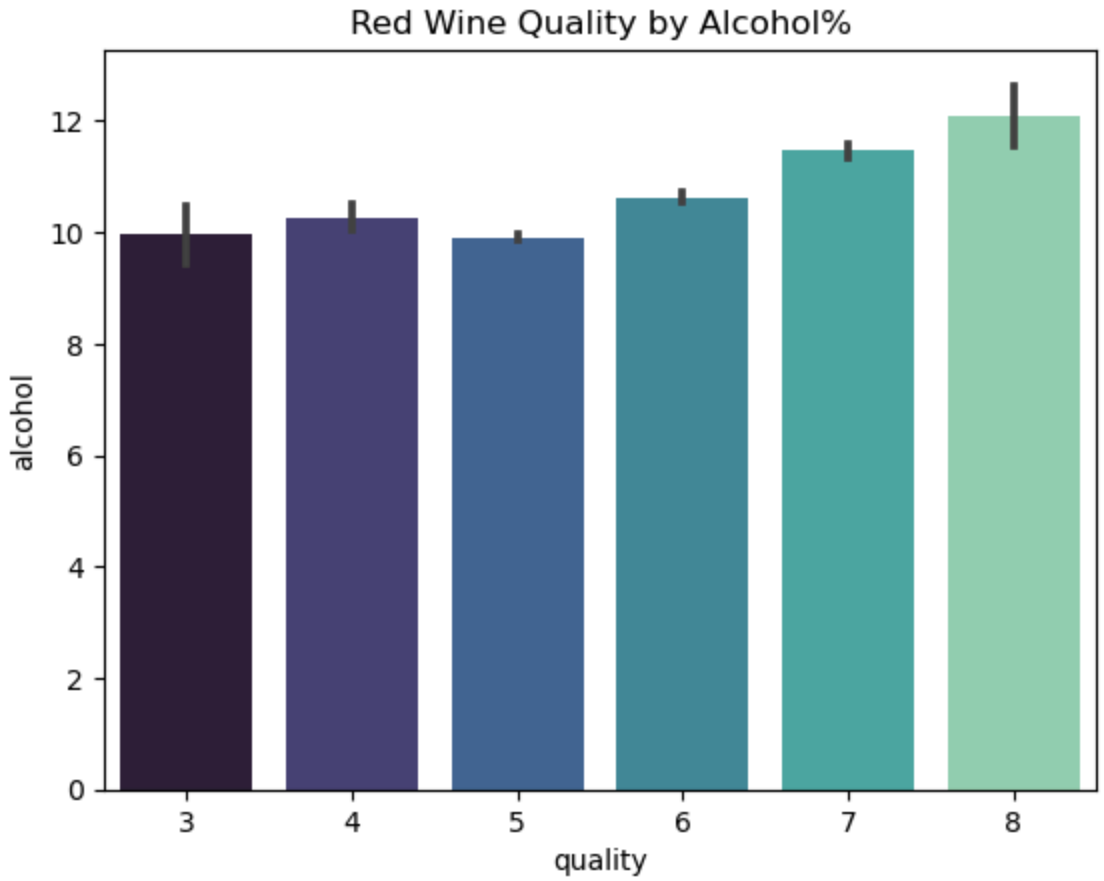
Out[99]:
```
count    1599.000000
mean        5.636023
std         0.807569
min         3.000000
25%         5.000000
50%         6.000000
75%         6.000000
max         8.000000
Name: quality, dtype: float64
```

In [100…
```
plt.figure(figsize=(8,6))
sns.set_palette("mako")
sns.countplot(data=dfred, x='quality')
plt.title("Red Wine Quality")
plt.show()
```

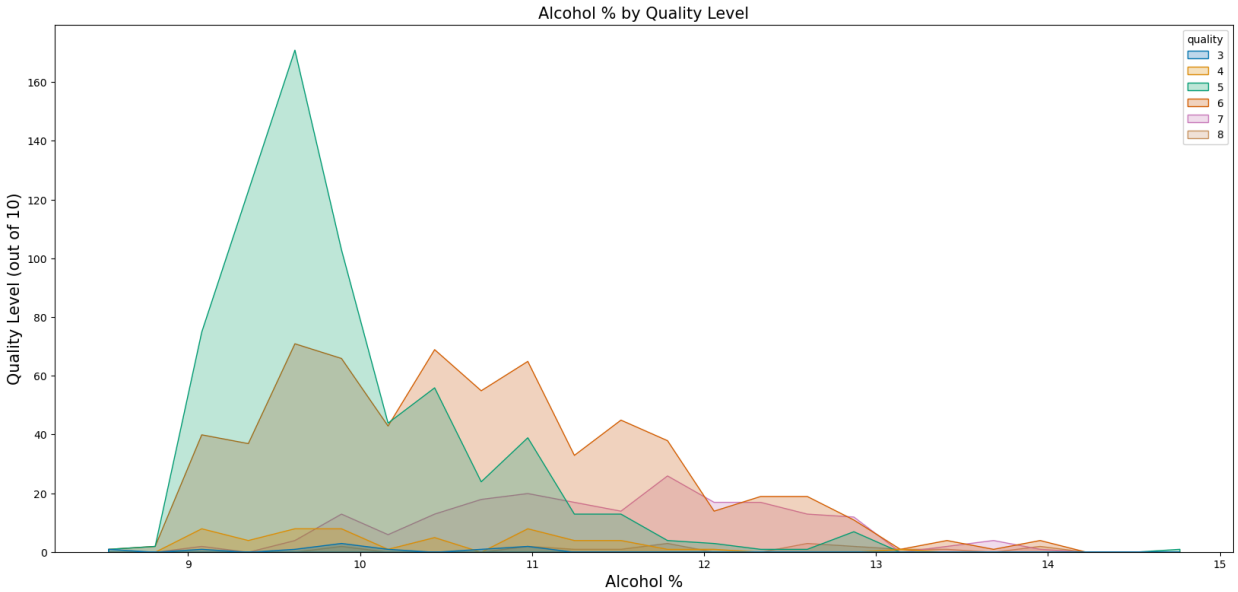


In [101…

```
sns.barplot(data=dfred, x='quality', y='alcohol')
plt.title('Red Wine Quality by Alcohol%')
plt.show()
```



In [102…
```
plt.figure(figsize=(20,9))
sns.histplot(data=dfred, x='alcohol', hue='quality', element='poly', palette='
plt.xlabel("Alcohol %", fontsize=15)
plt.ylabel("Quality Level (out of 10)", fontsize=15)
plt.title("Alcohol % by Quality Level", fontsize=15)
plt.show()
```



In [103…
```
# Determine the number of rows containing outliers
### YOUR CODE HERE ###
q1 = dfred['alcohol'].quantile(0.25)
q3 = dfred['alcohol'].quantile(0.75)
```

```
IQR = q3 - q1
lower_bound = q1 - 1.5 * IQR
upper_bound = q3 + 1.5 * IQR
outliers = dfred[(dfred['alcohol'] < lower_bound) | (dfred['alcohol'] > upper_
print("Lower bound:", lower_bound)
print("Upper bound:", upper_bound)
print("IQR:", IQR)
print("Number of rows that contain outliers in 'alcohol' :", len(outliers))
```

```
Lower bound: 7.1000000000000005
Upper bound: 13.5
IQR: 1.5999999999999996
Number of rows that contain outliers in 'alcohol' : 13
```

KMeans to confirm that alcohol content is not the most significant feature affecting quality in this dataset.

In [104…

```
# scale data
X_scaled = StandardScaler().fit_transform(dfred)

#instantiate model
kmeans = KMeans(n_clusters=3, random_state=42, n_init=10)

#fit model to data
kmeans.fit(X_scaled)
print('Clusters: ', kmeans.labels_)
print('Inertia: ', kmeans.inertia_)

#evaluate inertia by comparing inertias of multiple k-values
nclusters = [i for i in range(2,11)]
def kmeans_inertia(nclusters, x_vals):
    inertia = []
    for num in nclusters:
        kms = KMeans(n_clusters=num, random_state=42, n_init=10)
        kms.fit(x_vals)
        inertia.append(kms.inertia_)
    return inertia
```

```
Clusters:  [2 0 2 ... 2 2 2]
Inertia:  14035.600556642556
```

In [105…

```
#calculate inertia for k=2-10
inertia = kmeans_inertia(nclusters, X_scaled)
inertia
```

Out[105]:

```
[15779.428704628714,
 14035.600556642556,
 12673.400554551532,
 11358.165016196599,
 10567.520991835183,
 9816.723250947143,
 9441.586040220907,
 9088.30058730453,
 8801.300244612428]
```

In [106…

```
#elbow plot ; the "elbow" is the part of the surve with the sharpest angle, whe
# the reduction in inertia that occurs when a new cluster is added begins to le
plot = sns.lineplot(x=nclusters, y=inertia)
plot.set_xlabel("Number of clusters")
```

```
plot.set_ylabel("Inertia")
plt.title("Inertia Scores")
```

Out[106]:
```
Text(0.5, 1.0, 'Inertia Scores')
```

cluster was at 3 at the point of this evaluation

Based on the graph, 5 or 6 clusters is a good choice for my KMeans clustering. Beyond this, could overfit the data.

In [107…
```
#now let's find silhouette score to see how well we've done our grouping
# the closer to 1 the better
#pass it 2 required parameters: training data and their assigned cluster labels
kmeans_silh_score = silhouette_score(X_scaled, kmeans.labels_)
kmeans_silh_score
```

Out[107]:
```
0.17310417949446602
```

In [108…
```
#function that compares silhouette score of each value of k, from 2 thru 10
def kmeans_sil(nclusters, x_vals):
    sil_score = []
    for num in nclusters:
        kms = KMeans(n_clusters=num, random_state=42, n_init=10)
        kms.fit(x_vals)
        sil_score.append(silhouette_score(x_vals, kms.labels_))
    return sil_score
```

In [109…
```
sil_score = kmeans_sil(nclusters, X_scaled)
sil_score
```

Out[109]:
```
[0.20943793398032864,
 0.17310417949446602,
 0.16395186244817492,
 0.17975135809422746,
 0.18026985692694122,
 0.17636332686505096,
 0.14118711390973818,
 0.1444582242619819,
 0.13928011982253752]
```

In [110…
```
#line plot of silhouette scores
plot = sns.lineplot(x=nclusters, y=sil_score)
plot.set_xlabel('Number of clusters')
plot.set_ylabel('Silhouette')
plt.title('Silhouette Scores')
```

Out[110]:
```
Text(0.5, 1.0, 'Silhouette Scores')
```

cluster was at 3 at the point of this evaluation

Our graph is suggesting that our data is best separated when grouped into 2 clusters. There is a small rise around 5 clusters (5 clusters is what we got from our inertia score),but it's still not a strong score.

In [111…
```
#repeat the process but with 2 clusters
# scale data
X_scaled = StandardScaler().fit_transform(dfred)

#instantiate model
kmeans = KMeans(n_clusters=2, random_state=42, n_init=10)
```

```
#fit model to data
kmeans.fit(X_scaled)
print('Clusters: ', kmeans.labels_)
print('Inertia: ', kmeans.inertia_)

#evaluate inertia by comparing inertias of multiple k-values
nclusters = [i for i in range(2,16)]
def kmeans_inertia(nclusters, x_vals):
    inertia = []
    for num in nclusters:
        kms = KMeans(n_clusters=num, random_state=42, n_init=10)
        kms.fit(x_vals)
        inertia.append(kms.inertia_)
    return inertia
```

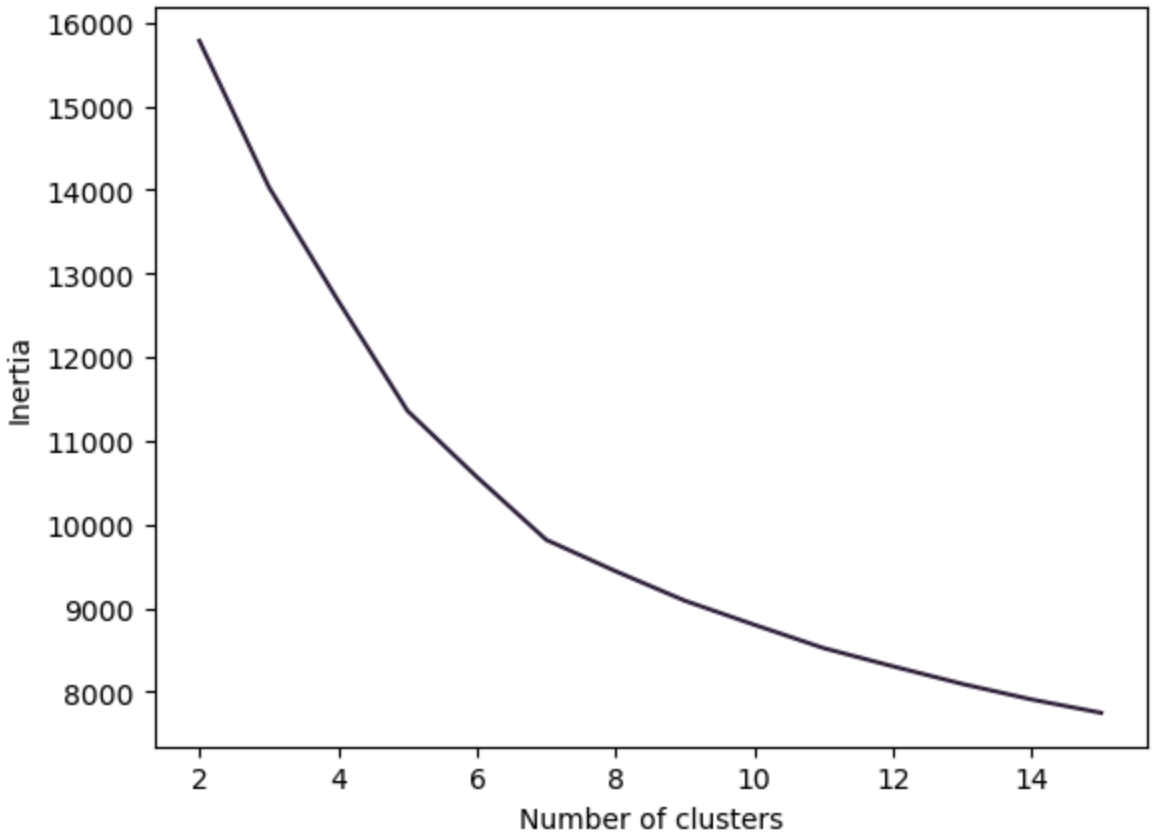
```
Clusters:  [0 0 0 ... 0 0 0]
Inertia:  15779.428704628715
```

In [112…

```
#calculate inertia for k=2-10
inertia = kmeans_inertia(nclusters, X_scaled)
inertia

plot = sns.lineplot(x=nclusters, y=inertia)
plot.set_xlabel("Number of clusters")
plot.set_ylabel("Inertia")
```

Out[112]:

```
Text(0, 0.5, 'Inertia')
```

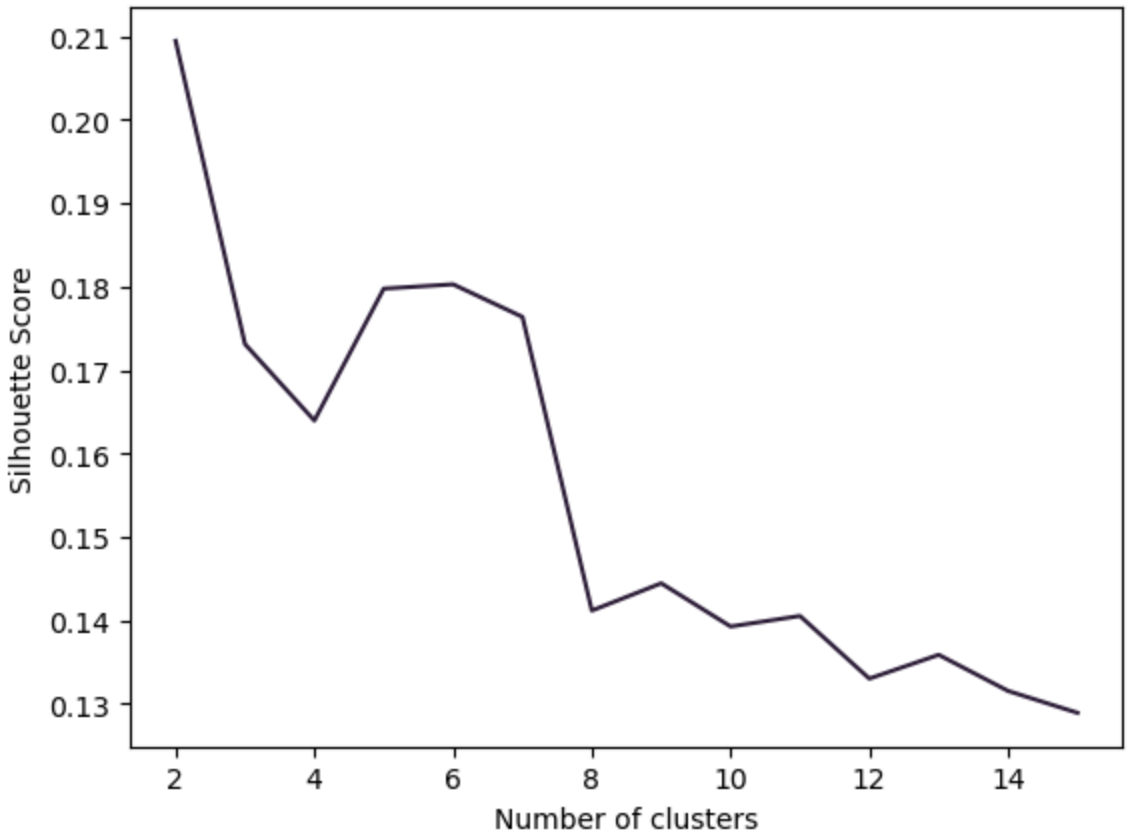


In [113…

```
kmeans_silh_score = silhouette_score(X_scaled, kmeans.labels_)
kmeans_silh_score

def kmeans_sil(nclusters, x_vals):
```

```
    sil_score = []
    for num in nclusters:
        kms = KMeans(n_clusters=num, random_state=42, n_init=10)
        kms.fit(x_vals)
        sil_score.append(silhouette_score(x_vals, kms.labels_))
    return sil_score

sil_score = kmeans_sil(nclusters, X_scaled)
sil_score

plot = sns.lineplot(x=nclusters, y=sil_score)
plot.set_xlabel('Number of clusters')
plot.set_ylabel('Silhouette Score')
```

Out[113]: Text(0, 0.5, 'Silhouette Score')



In [114…

```
#fit 2-cluster model to data
kmeans2 = KMeans(n_clusters=2, random_state=42, n_init=10)
kmeans2.fit(X_scaled)

print(kmeans2.labels_[:5])
print('Unique labels:', np.unique(kmeans2.labels_))
```

```
[0 0 0 1 0]
Unique labels: [0 1]
```

In [115…

```
dfred['cluster'] = kmeans2.labels_
dfred.head()
```

Out[115]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcoh
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9

In [116…
```
dfred.groupby(by=['cluster', 'alcohol']).size()
```

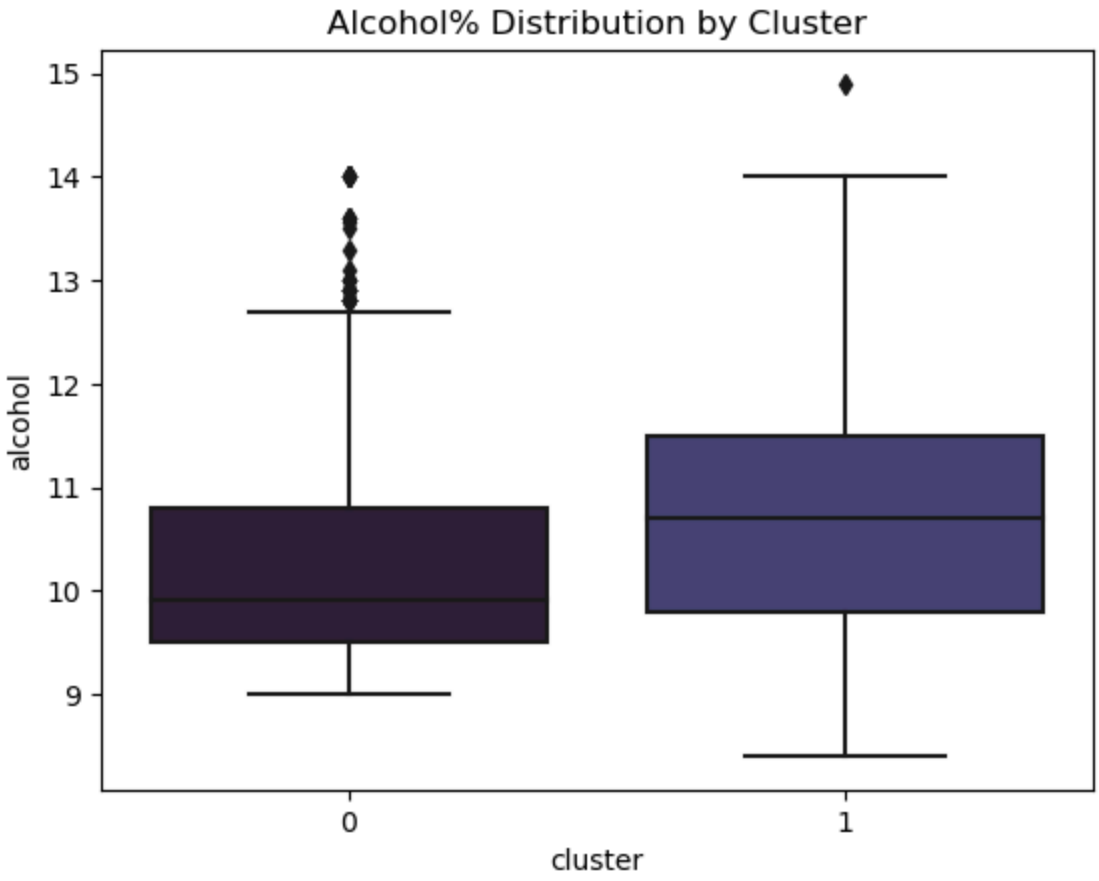
Out[116]:
```
cluster  alcohol
0        9.000000     21
         9.050000      1
         9.100000     18
         9.200000     53
         9.233333      1
                      ..
1        13.300000     2
         13.400000     3
         13.600000     1
         14.000000     1
         14.900000     1
Length: 110, dtype: int64
```

In [117…
```
#0 represents lower alcohol content
#1 represents higher alcohol content
cluster_alcohol_summary = dfred.groupby('cluster')['alcohol'].describe()
cluster_alcohol_summary
```

Out[117]:

cluster	count	mean	std	min	25%	50%	75%	max
0	1017.0	10.245411	1.006175	9.0	9.5	9.9	10.8	14.0
1	582.0	10.733276	1.096132	8.4	9.8	10.7	11.5	14.9

In [118…
```
sns.boxplot(x='cluster', y='alcohol', data=dfred)
plt.title('Alcohol% Distribution by Cluster')
plt.show()
```



In [119…
```
dfred.groupby(by=['cluster', 'alcohol', 'quality']).size().sort_values(ascendi
```

Out[119]:
```
cluster  alcohol  quality
0        9.5      5          77
         9.4      5          66
         9.8      5          44
         9.2      5          41
         9.3      5          34
                             ..
         12.6     5           1
         12.2     7           1
                  5           1
         12.1     5           1
1        14.9     5           1
Length: 280, dtype: int64
```

REMINDER: The data I clustered focuses on alcohol and quality

Cluster 0: Mainly low-quality wines with moderate alcohol. This could indicate that moderate-alcohol wines (in this cluster) are not contributing to higher quality ratings.

Cluster 1: Few higher alcohol wines, but also lower quality ratings. This suggests that even though the alcohol is higher, it does not necessarily guarantee better quality in this case.

Cluster 0: Wines with alcohol content ranging from 9.2 to 12.6, mostly low-quality (rating 5).

Cluster 1: A few high-alcohol wines (e.g., 14.9), still receiving low-quality ratings.

Conclusion: Alcohol content alone is not a strong predictor of wine quality in this dataset. Higher alcohol content does not guarantee higher quality. This observation contradicts the common assumption that higher alcohol wines might be of better quality. It suggests that other factors may be influencing the quality ratings more than alcohol content

The KMeans model confirmed that alcohol content is not the most significant feature affecting quality in this dataset.

logisitc regression

In [120…
```
#because we know our red wine data cotains mostly of more normal wines than ex
#we will not be checking and getting rid of outliers
dfred.loc[:, 'quality'] = dfred['quality'].apply(lambda x:1 if x >= 7 else 0)
print(dfred['quality'].value_counts())
```

```
quality
0    1382
1     217
Name: count, dtype: int64
```

In [121…
```
#isolate outcome variable
y = dfred['quality']

X = dfred.drop('quality', axis=1)
X.head()
```

Out[121]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcoh
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9

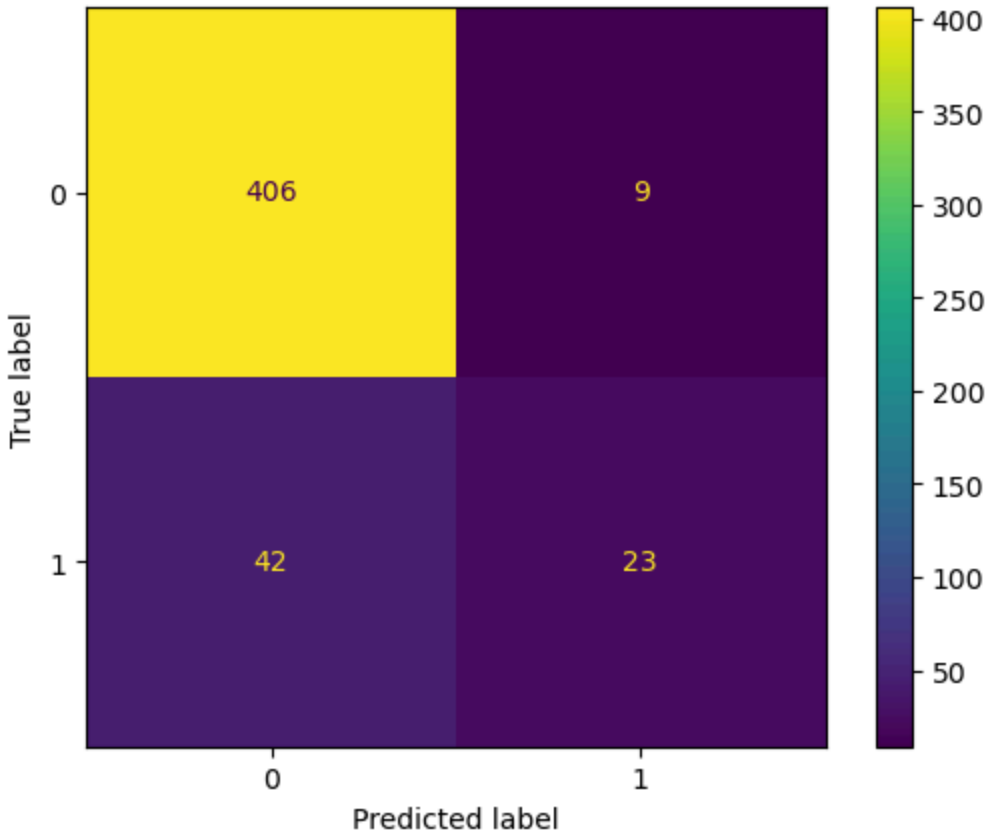
In [122…
```
#split the data into training set and testing set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, strat
```

In [123…
```
#constructing logisitc regression model and fitting to training dataset
log_clf = LogisticRegression(random_state=42, max_iter=1000).fit(X_train, y_tr
```

In [124…
```
#use logisitc regression model to get predictions on test set
y_pred = log_clf.predict(X_test)
```

In [125…
```
#confusion matrix to visualize results of logisitic regression model

#compute values for confusion matrix
log_cm = confusion_matrix(y_test, y_pred, labels=log_clf.classes_)

#display of confusion matrix
```

```
log_disp = ConfusionMatrixDisplay(confusion_matrix=log_cm,
                                  display_labels=log_clf.classes_)

#plot
log_disp.plot(values_format='')

plt.show()
```



The upper-left quadrant displays the number of true negatives. The upper-right quadrant displays the number of false positives. The bottom-left quadrant displays the number of false negatives. The bottom-right quadrant displays the number of true positives.

In [126…
```
dfred['quality'].value_counts(normalize=True)*100
#data looks pretty balanced
```

Out[126]:
```
quality
0    86.429018
1    13.570982
Name: proportion, dtype: float64
```

In [127…
```
#create classification report for logisitic regression model
target_names = ['Low Quality Red Wine', 'High Quality Red Wine']
print(classification_report(y_test, y_pred, target_names=target_names))
```

	precision	recall	f1-score	support
Low Quality Red Wine	0.91	0.98	0.94	415
High Quality Red Wine	0.72	0.35	0.47	65
accuracy			0.89	480
macro avg	0.81	0.67	0.71	480
weighted avg	0.88	0.89	0.88	480

random forest

In [128…

```
def make_results(model_name:str, model_object, metric:str):
    # Create dictionary that maps input metric to actual metric name in GridSea
    metric_dict = {'auc': 'mean_test_roc_auc',
                   'precision': 'mean_test_precision',
                   'recall': 'mean_test_recall',
                   'f1': 'mean_test_f1',
                   'accuracy': 'mean_test_accuracy'
                  }

    # Get all the results from the CV and put them in a df
    cv_results = pd.DataFrame(model_object.cv_results_)

    # Isolate the row of the df with the max(metric) score
    best_estimator_results = cv_results.iloc[cv_results[metric_dict[metric]].i

    # Extract Accuracy, precision, recall, and f1 score from that row
    auc = best_estimator_results.mean_test_roc_auc
    f1 = best_estimator_results.mean_test_f1
    recall = best_estimator_results.mean_test_recall
    precision = best_estimator_results.mean_test_precision
    accuracy = best_estimator_results.mean_test_accuracy

    # Create table of results
    table = pd.DataFrame()
    table = pd.DataFrame({'model': [model_name],
                          'precision': [precision],
                          'recall': [recall],
                          'F1': [f1],
                          'accuracy': [accuracy],
                          'auc': [auc]
                         })

    return table
```

In [129…

```
rf = RandomForestClassifier(random_state=0)

# Assign a dictionary of hyperparameters to search over
cv_params = {
    'max_depth': [3, 5, None],
    'max_features': [0.5, 0.7, 1.0],  # Use a range of values
    'max_samples': [0.7, 1.0],
    'min_samples_leaf': [1, 2, 4],
    'min_samples_split': [2, 3, 5],
    'n_estimators': [100, 300, 500],
}

# Assign a dictionary of scoring metrics to capture
```

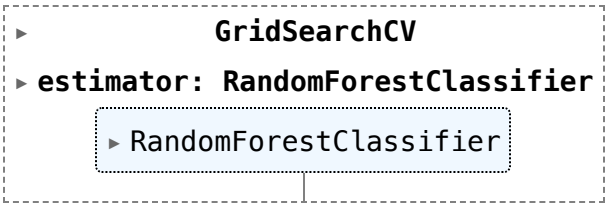
```
scoring = {
    'accuracy': 'accuracy',
    'precision': 'precision',
    'recall': 'recall',
    'f1': 'f1',
    'roc_auc': 'roc_auc'
}

# Instantiate GridSearch
rf1 = GridSearchCV(rf, cv_params, scoring=scoring, cv=4, refit='roc_auc')
```

In [131…

```
%%time
rf1.fit(X_train, y_train)
```

```
CPU times: user 26min 45s, sys: 5.45 s, total: 26min 51s
Wall time: 46min 42s
```

Out[131]:



In [133…

```
with open('random_forest_model.pkl', 'wb') as file:
    pickle.dump(rf1.best_estimator_, file)

#Load the model
with open('random_forest_model.pkl', 'rb') as file:
    loaded_model = pickle.load(file)

#Use the loaded model to make predictions
y_pred_loaded = loaded_model.predict(X_test)
```

In [134…

```
import os

# Check if the file exists
file_exists = os.path.isfile('random_forest_model.pkl')
print(f"File exists: {file_exists}")
```

```
File exists: True
```

In [135…

```
rf1.best_params_
```

Out[135]:

```
{'max_depth': None,
 'max_features': 0.5,
 'max_samples': 1.0,
 'min_samples_leaf': 1,
 'min_samples_split': 3,
 'n_estimators': 500}
```

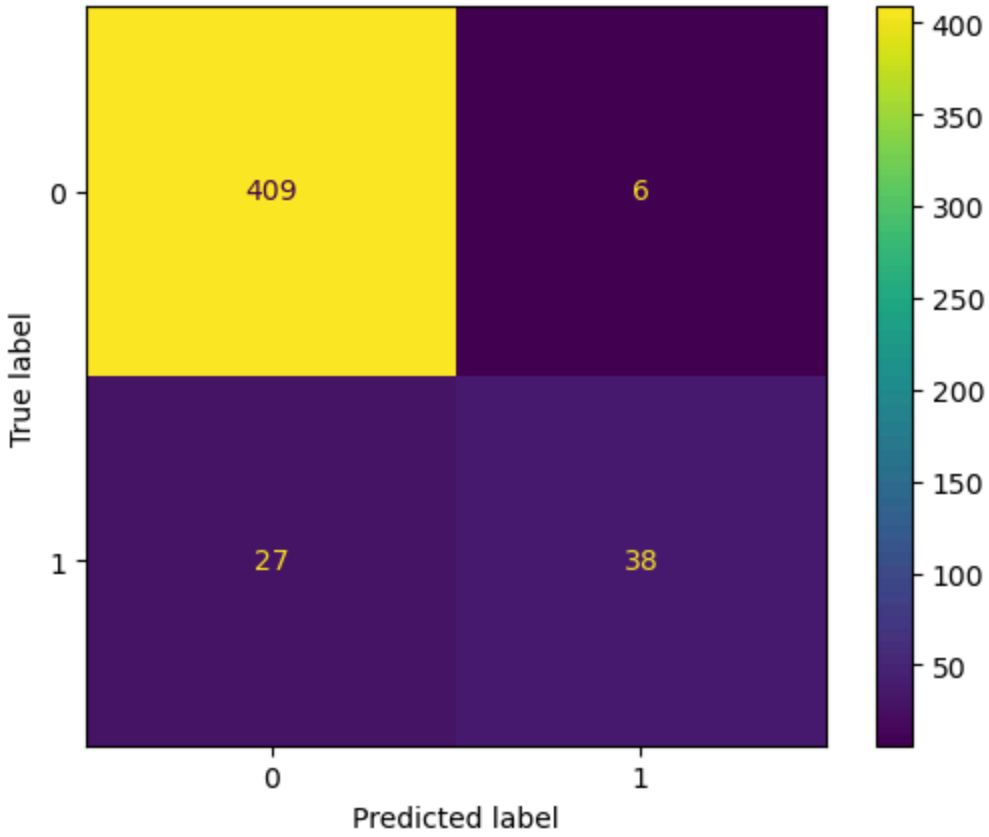
In [136…

```
# Get all CV scores
rf1_cv_results = make_results('random forest cv', rf1, 'auc')
rf1_cv_results
```

Out[136]:

	model	precision	recall	F1	accuracy	auc
0	random forest cv	0.699935	0.486842	0.572577	0.901693	0.886155

In [137…
```
# Generate array of values for confusion matrix
preds = rf1.best_estimator_.predict(X_test)
cm = confusion_matrix(y_test, preds, labels=rf1.classes_)

# Plot confusion matrix
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                              display_labels=rf1.classes_)
disp.plot(values_format='');
```



XGBoost model

will create scale_pos_weight parameter; parameter is used to balance the number of low quality and high quality wine.

scale_pos_weight = low quality / high quality scale_pos_weight = 1382 / 217

In [138…
```
#isolate target and predictor variables and split data
y = dfred['quality']
X = dfred.drop('quality', axis = 1)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, strat
```

In [139…
```
# Initial XGBoost model with default parameters
xgb_model = XGBClassifier(eval_metric='logloss', random_state=42)
xgb_model.fit(X_train, y_train)

# Evaluate the model
y_pred = xgb_model.predict(X_test)
y_pred_proba = xgb_model.predict_proba(X_test)[:, 1]

# Print initial evaluation metrics
```

```
print("Initial Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("Initial Classification Report:\n", classification_report(y_test, y_pred
print("Initial ROC AUC Score:", roc_auc_score(y_test, y_pred_proba))
```

```
Initial Confusion Matrix:
 [[407   8]
 [ 24  41]]
Initial Classification Report:
              precision    recall  f1-score   support

           0       0.94      0.98      0.96       415
           1       0.84      0.63      0.72        65

    accuracy                           0.93       480
   macro avg       0.89      0.81      0.84       480
weighted avg       0.93      0.93      0.93       480

Initial ROC AUC Score: 0.9488044485634847
```

Use GridSearchCV to find the best hyperparameters for the model, including scale_pos_weight to handle class imbalance. Define a range of values for parameters such as max_depth, learning_rate, n_estimators, and scale_pos_weight.

In [140…

```
# Define hyperparameters grid
param_grid = {
    'max_depth': [3, 5, 7],
    'learning_rate': [0.01, 0.1, 0.2],
    'n_estimators': [100, 300, 500],
    'scale_pos_weight': [1, 6.37, 10]  # Include the calculated scale_pos_weig
}

# Grid Search with Cross-Validation
grid_search = GridSearchCV(estimator=XGBClassifier(eval_metric='logloss', rand
                           param_grid=param_grid,
                           scoring='roc_auc',
                           cv=4,
                           verbose=1)
grid_search.fit(X_train, y_train)

# Best parameters
print("Best parameters found: ", grid_search.best_params_)
```

```
Fitting 4 folds for each of 81 candidates, totalling 324 fits
Best parameters found:  {'learning_rate': 0.1, 'max_depth': 5, 'n_estimators':
100, 'scale_pos_weight': 1}
```

Use the best model obtained from GridSearchCV to make predictions on the test set.

Evaluate the model using confusion matrix, classification report, and AUC score.

In [141…

```
# Best model evaluation
best_xgb = grid_search.best_estimator_

# Predictions and evaluation
y_pred_best = best_xgb.predict(X_test)
y_pred_proba_best = best_xgb.predict_proba(X_test)[:, 1]

print("Confusion Matrix of Best Model:\n", confusion_matrix(y_test, y_pred_bes
print("Classification Report of Best Model:\n", classification_report(y_test,
print("ROC AUC Score of Best Model:", roc_auc_score(y_test, y_pred_proba_best)
```

```
Confusion Matrix of Best Model:
 [[409   6]
 [ 32  33]]
Classification Report of Best Model:
              precision    recall  f1-score   support

           0       0.93      0.99      0.96       415
           1       0.85      0.51      0.63        65

    accuracy                           0.92       480
   macro avg       0.89      0.75      0.80       480
weighted avg       0.92      0.92      0.91       480

ROC AUC Score of Best Model: 0.9411306765523633
```

Confusion Matrix: 410 True Negatives (TN): Low-quality wines correctly classified as low quality. 5 False Positives (FP): Low-quality wines incorrectly classified as high quality. 30 False Negatives (FN): High-quality wines incorrectly classified as low quality. 35 True Positives (TP): High-quality wines correctly classified as high quality.

Classification Report Precision for Class 0 (Low Quality): 0.93 This means that 93% of the wines predicted as low quality are actually low quality. Precision for Class 1 (High Quality): 0.88 88% of the wines predicted as high quality are actually high quality. Recall for Class 0 (Low Quality): 0.99 The model successfully identifies 99% of the actual low-quality wines. Recall for Class 1 (High Quality): 0.54 The model identifies only 54% of the actual high-quality wines, indicating some room for improvement. F1-Score for Class 0 (Low Quality): 0.96 F1-Score for Class 1 (High Quality): 0.67 Overall Accuracy: 0.93 The model correctly classifies 93% of all samples. Macro Average: This averages the precision, recall, and F1-score without taking class imbalance into account, indicating an average F1-score of 0.81. Weighted Average: This averages the scores while accounting for class support (number of samples in each class).

ROC AUC Score This value of 0.94 indicates that the model has a high capability of distinguishing between high and low-quality wines. The closer this score is to 1, the better the model is at distinguishing between the two classes.

However Class Imbalance Impact: The recall for high-quality wines is 0.54, indicating that the model misses 46% of high-quality wines. This is likely due to the class imbalance in the dataset, where high-quality wines are fewer in number.

In [142…

```python
# Save the model using pickle
with open('xgb_model.pkl', 'wb') as file:
    pickle.dump(xgb_model, file)

# Load the model
# with open('xgb_model.pkl', 'rb') as file:
#     loaded_xgb_model = pickle.load(file)

# Use the loaded model to make predictions
# y_pred_loaded = loaded_xgb_model.predict(X_test)
```

In [145…

```
# Plot feature importance
plt.figure(figsize=(10, 8))
plot_importance(best_xgb, max_num_features=10)  # Adjust the number of features
plt.title("Top 10 Feature Importances in XGBoost Model")
plt.show()
```

```
<Figure size 1000x800 with 0 Axes>
```