This file also contains the EDA and Preprocessing, but since this came after doing the other attempts, and it has overall lower correlation with data, we decided that only one locations should suffice.

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

import plotly.express as px
import plotly.graph_objects as go
from plotly.subplots import make_subplots

from collections import Counter

from sklearn.model_selection import train_test_split,RandomizedSearchCV
from sklearn.metrics import f1_score,accuracy_score,recall_score,precision_score
from xgboost import XGBClassifier
#from hyperopt import STATUS_OK,Trials, fmin, hp, tpe
```

```
The history saving thread hit an unexpected error (OperationalError('attempt to writ
e a readonly database')).History will not be written to the database.
```

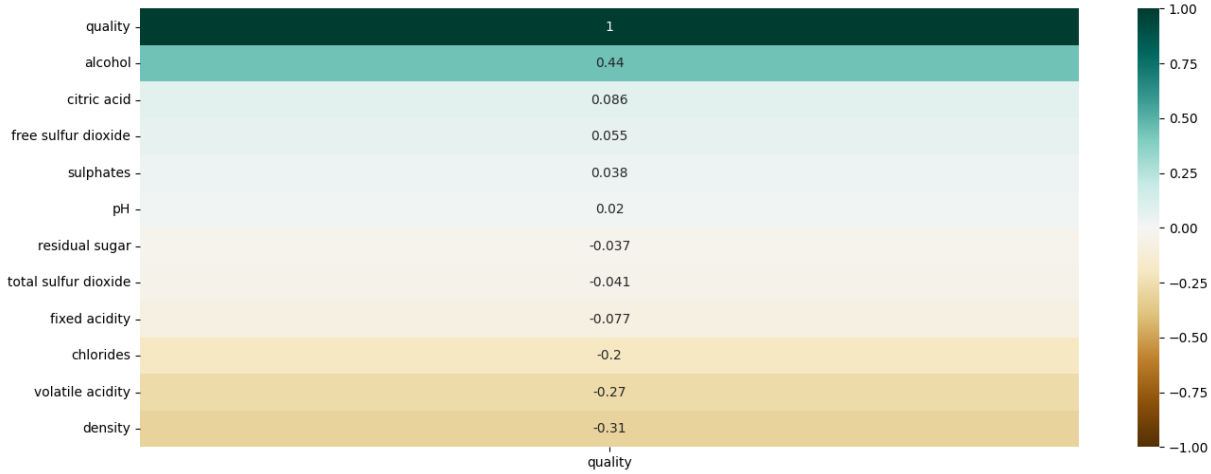
In [2]:
```
#this was used to randomize the data as it was made by just adding one of wine file
"""
total_wine = pd.read_csv("data/winequality-total.csv", delimiter=";")
total = total_wine.sample(frac=1)
total_wine.to_csv("data/winequality-total.csv")
"""
```

Out[2]:
```
'\ntotal_wine = pd.read_csv("data/winequality-total.csv", delimiter=";")\ntotal =
total_wine.sample(frac=1)\ntotal_wine.to_csv("data/winequality-total.csv")\n'
```

In [3]:
```
total_wine = pd.read_csv("data/winequality-total.csv",index_col=0)
```

In [4]:
```
plt.figure(figsize=(16,6))
sns.heatmap(total_wine.corr()[["quality"]].sort_values(by='quality',ascending=False
```

Out[4]:
```
<Axes: >
```



The only valuable ones that seem to be present: alcohol, volatile aciditiy, chlorides, and density. However, to maintain consistence, and to also test the effectiveness of correlation analysis, we will drop the same ones we did for the xgboost implmentation on only red wine. Granted, the dynamics do change with the entence of white wine, but if the accuracy is at least comparable, we can investigate later.

In [5]:
```
total_wine = total_wine.drop(["residual sugar","pH", "free sulfur dioxide"], axis=1
```

Alcohol

In [6]:
```
fig = make_subplots(rows=2,cols=2)

#histogram for red
fig.append_trace(go.Histogram(
    x=total_wine["alcohol"],
    name="red hist"), row=1,col=1)

#boxplot for red
fig.append_trace(go.Box(
    x=total_wine["alcohol"],
    name="red box"
),row=1,col=2)

fig.append_trace(go.Box(
    x=total_wine["quality"],
    y=total_wine["alcohol"],
    name="quality/alcohol"
),row=2, col=1)

#histogram for white
fig.update_layout(height=600, width=1200, title_text="Alochol data")
fig.show()
```

Allmost the same as what was seen ealier in red_wine. Therefore, going to treat it about the same

---

Volatile acidity

In [7]:
```
fig = make_subplots(rows=2,cols=2)

#histogram for red
fig.append_trace(go.Histogram(
    x=total_wine["volatile acidity"],
    name="normal hist"), row=1,col=1)

#boxplot for red
fig.append_trace(go.Box(
    x=total_wine["volatile acidity"],
    name="normal box"
),row=1,col=2)
```

```
fig.append_trace(go.Box(
    x=total_wine["quality"],
    y=total_wine["volatile acidity"],
    name="quality/alcohol"
),row=2, col=1)

#histogram for white
fig.update_layout(height=600, width=1200, title_text="VA data")
fig.show()
```

Extremely right skewed, can be adjusted later if necessary, but it seems that the lower the value goes the more likely it is high quality, but not a strong indicator

---

Density

In [8]:
```
fig = make_subplots(rows=2,cols=2)

#histogram for red
fig.append_trace(go.Histogram(
    x=total_wine["density"],
    name="normal hist"), row=1,col=1)

#boxplot for red
fig.append_trace(go.Box(
    x=total_wine["density"],
    name="normal box"
),row=1,col=2)

fig.append_trace(go.Box(
    x=total_wine["quality"],
    y=total_wine["density"],
    name="quality/density"
),row=2, col=1)

#histogram for white
fig.update_layout(height=600, width=1200, title_text="density data")
fig.show()
```

Should drop that outlier but follow a similar trend to VA, where the values tend to decrease. Additionally, it apppears to still be a shaky, but still gaussian distribution.

Chlorides

In [9]:
```
fig = make_subplots(rows=2,cols=2)

#histogram for red
fig.append_trace(go.Histogram(
    x=total_wine["chlorides"],
    name="normal hist"), row=1,col=1)

#boxplot for red
fig.append_trace(go.Box(
    x=total_wine["chlorides"],
```

```
    name="normal box"
),row=1,col=2)

fig.append_trace(go.Box(
    x=total_wine["quality"],
    y=total_wine["chlorides"],
    name="quality/chlorides"
),row=2, col=1)

#histogram for white
fig.update_layout(height=600, width=1200, title_text="chlorides data")
fig.show()
```

Very far skew, but it seems relevent. The

Preprocessing

In [10]:
```
def remove_outliers(df: pd.DataFrame, n :float, columns):
    #this is the Tukey ruel which gets the values that exists outside of the outer
    #This is valuable if outliers effect the data alot (typically regression), in c
    #From EDA it seems that we should not remove outliers as they can help point to
    total_outliers = []

    for col in columns:

        #generating the quantile ranges that will be used to determine outliers
        q1 = df[col].quantile(.25)
        q3 = df[col].quantile(.75)

        iqr = q3 - q1
        outer_fence = iqr * 1.5

        outliers = df[(df[col] < q1 - outer_fence) | (df[col] > q3 + outer_fence)].

        total_outliers.extend(outliers)

    #select the indexes (tuples) that have more than n attributes that are outliers
    #creates an object that has keys (index), with values (amount of apperences, wh
    outliers = Counter(total_outliers)

    #iterates over all items and reutrns the
    items_greater =[]
    for i in outliers.items():
        if(i[1] >= n):
            items_greater.append(i[0])

    return items_greater
```

In [11]:
```
len(total_wine)
```

Out[11]: 6497

In [12]:
```
#finding all outliers that have significant outliers
outliers = remove_outliers(total_wine,2,total_wine.columns[:-1])
len(total_wine.drop(outliers, axis=0).reset_index(drop=True))
```

Out[12]: 6109

Since it is so low and Dt are natually resistant, am going to keep outliers in (might hugley effect our imbalanced)

Training Model (with same features as red wine)

In [13]:
```
total_train, total_test = train_test_split(total_wine, test_size=0.2, stratify=tota
```

In [14]:
```
X_train = total_train.drop("quality" , axis=1)
y_train = total_train["quality"]
```

```
X_test = total_test.drop("quality" , axis=1).copy()
y_test = total_test["quality"]
```

In [15]:
```
#need to map the data for XGBoost to work, expects data in this format
y_train = y_train.map({3: 0, 4: 1, 5:2, 6:3, 7:4, 8:5, 9:6})
y_test = y_test.map({3: 0, 4: 1, 5:2, 6:3, 7:4, 8:5, 9:6})
```

Hyper parameters:

```
    Booster: choose the type of booster to use (we will use tree in
    this case)
        -- 3 options
        tree( gbtree,dart)
        linear(gblinear)

    Booster Parameters: only the tree booster ones, only listing the
    ones usuful to mulitlable imbalanced data (trees for the win)

        -- eta: the learning rat for Gradient boosting, and its range
    typically is 0.01 - 0.2
        -- gamma: how the node is split in a tree, the larger the more
    conservative a tree is, range(0 --> infinity)
        -- max_depth: maximum depth of a tree typical values are (3-
    10), should use cv
        -- min_child_weight: tune using cv but range is 0-->infinite
        -- subsample: fraction of observations to be samples for tree,
    lower values more conservative, typical values (0,1)
        -- colsample_bytree: ratio of columns when construction each
    tree
        -- colsample_bylevel: ratio of columns at each level of the
    tree
        -- tree method: constuction algorithm used in model (multiple
    choices)
        -- max_leaves: is maximum number of nodes to be added

        Others
        -- alpha : used for lasso regression, increasing makes the
    model more conservative
        -- lambda : used for ridge regression, increasing makes the
    model more conservative

    Learning Task: parameters used to define the optimization objective
    for learning

        --objective: should use multi:softprob or multi: softmax
        --eval metric: should use auc, or merror
```

In [16]:
```
space = {'max_depth': [3, 6, 10, 15, 20],
         'learning_rate': [0.01, 0.1, 0.2, 0.3, 0.4],
         'subsample': np.arange(0.5, 1.0, 0.1),
         'gamma' : np.arange(1,9,0.1),
         'colsample_bytree': np.arange(0.5, 1.0, 0.1),
```

```
         'colsample_bylevel': np.arange(0.5, 1.0, 0.1),
         'min_child_weight' : np.arange(1, 10, 1),
         'n_estimators': [100,150, 250, 500, 750],
        }
```

In [17]:
```
#making the base model
#multi prob is a vector, containing all the classes a
model = XGBClassifier(objective="multi:softprob",eval_metric="auc",)
clf = RandomizedSearchCV(estimator=model,
                         param_distributions=space, #assigning space
                         scoring="f1_weighted", #eval metric for the hyperparms, we
                         n_iter=25, #amount of iterations per cv (random combinatio
                         n_jobs=4, #amount of parralel processes to run
                         random_state=1)

clf.fit(X_train,y_train)
```

```
/home/cole/anaconda3/envs/datasci/lib/python3.10/site-packages/sklearn/model_selecti
on/_split.py:700: UserWarning:

The least populated class in y has only 4 members, which is less than n_splits=5.
```

Out[17]:
```
▸ RandomizedSearchCV
▸ estimator: XGBClassifier
    ▸ XGBClassifier
```

In [18]:
```
#assigning be params
best_hyperparams = clf.best_params_
best_hyperparams
```

Out[18]:
```
{'subsample': 0.7999999999999999,
 'n_estimators': 250,
 'min_child_weight': 4,
 'max_depth': 15,
 'learning_rate': 0.1,
 'gamma': 1.7000000000000006,
 'colsample_bytree': 0.6,
 'colsample_bylevel': 0.5}
```

In [19]:
```
accuracy_f1 = []
recall = []
precision = []
accuracy = []
for i in range(0,5):
    clf=XGBClassifier(
        n_estimators = best_hyperparams['n_estimators'],
        max_depth = int(best_hyperparams['max_depth']),
        learning_rate = best_hyperparams['learning_rate'],
        gamma = best_hyperparams['gamma'],
        min_child_weight=int(best_hyperparams['min_child_weight']),
        colsample_bytree=int(best_hyperparams['colsample_bytree']),
        colsample_bylevel=best_hyperparams['colsample_bylevel'],
```

```
        objective="multi:softprob", #type of objective function that is used, you h
        eval_metric="auc", #the measure to determine within the gradient boosting t
        seed=i)

    clf.fit(X_train,y_train)

    y_pred = clf.predict(X_test)

    #appending the scores from the particular rune to the list
    accuracy_f1.append(f1_score(y_test,y_pred,average="weighted",zero_division=1))
    recall.append(recall_score(y_test,y_pred,average="weighted",zero_division=1))
    precision.append(precision_score(y_test, y_pred,average="weighted",zero_divisio
    accuracy.append(accuracy_score(y_test,y_pred))
```

In [20]:
```
average_f1 = np.mean(accuracy_f1)
average_recall = np.mean(recall)
average_precision = np.mean(precision)
average_accuracy = np.mean(accuracy)
print("Average F1: " + str(average_f1))
print("Average Recall: " + str(average_recall))
print("Average Precision: " + str(average_precision))
print("Average Accuracy: " + str(average_accuracy))
```

```
Average F1: 0.5273647702420745
Average Recall: 0.558
Average Precision: 0.5370645680886247
Average Accuracy: 0.558
```

Testing Model (only using high direct corelation)

In [21]:
```
plt.figure(figsize=(16,6))
sns.heatmap(total_wine.corr()[["quality"]].sort_values(by='quality',ascending=False
```

Out[21]: <Axes: >

In [22]:
```
#dropping all values that are below 0.1 and between -0.1
total_wine_dropped = total_wine.drop(["citric acid", "sulphates", "total sulfur dio
```

In [23]:
```
X_train = total_wine_dropped.drop("quality" , axis=1)
y_train = total_wine_dropped["quality"]
X_test = total_wine_dropped.drop("quality" , axis=1).copy()
y_test = total_wine_dropped["quality"]
```

In [24]:
```
#mapping values to different type for algorithm
y_train = y_train.map({3: 0, 4: 1, 5:2, 6:3, 7:4, 8:5, 9:6})
y_test = y_test.map({3: 0, 4: 1, 5:2, 6:3, 7:4, 8:5, 9:6})
```

In [26]:
```
#making the base model
model = XGBClassifier(objective="multi:softprob",eval_metric="auc",)
clf = RandomizedSearchCV(estimator=model,
                         param_distributions=space,
                         scoring="f1_weighted",
                         n_iter=25,
                         n_jobs=4,
                         random_state=1)

clf.fit(X_train,y_train)
```

Out[26]:
```
▸ RandomizedSearchCV
▸ estimator: XGBClassifier
  ▸ XGBClassifier
```

In [27]:
```
best_hyperparams = clf.best_params_
best_hyperparams
```

Out[27]:
```
{'subsample': 0.6,
 'n_estimators': 150,
 'min_child_weight': 1,
 'max_depth': 15,
 'learning_rate': 0.2,
 'gamma': 6.200000000000005,
 'colsample_bytree': 0.6,
 'colsample_bylevel': 0.5}
```

In [28]:
```
accuracy_f1 = []
recall = []
precision = []
accuracy = []
for i in range(0,5):
    clf=XGBClassifier(
        n_estimators = best_hyperparams['n_estimators'],
        max_depth = int(best_hyperparams['max_depth']),
        learning_rate = best_hyperparams['learning_rate'],
        gamma = best_hyperparams['gamma'],
        min_child_weight=int(best_hyperparams['min_child_weight']),
        colsample_bytree=int(best_hyperparams['colsample_bytree']),
        colsample_bylevel=best_hyperparams['colsample_bylevel'],
        objective="multi:softprob", #type of objective function that is used, you h
        eval_metric="auc", #the measure to determine within the gradient boosting t
        seed=i)
```

```
    clf.fit(X_train,y_train)

    y_pred = clf.predict(X_test)

    #appending the scores from the particular rune to the list
    accuracy_f1.append(f1_score(y_test,y_pred,average="weighted",zero_division=1))
    recall.append(recall_score(y_test,y_pred,average="weighted",zero_division=1))
    precision.append(precision_score(y_test, y_pred,average="weighted",zero_divisio
    accuracy.append(accuracy_score(y_test,y_pred))
```

In [29]:

```
average_f1 = np.mean(accuracy_f1)
average_recall = np.mean(recall)
average_precision = np.mean(precision)
average_accuracy = np.mean(accuracy)
print("Average F1: " + str(average_f1))
print("Average Recall: " + str(average_recall))
print("Average Precision: " + str(average_precision))
print("Average Accuracy: " + str(average_accuracy))
```

```
Average F1: 0.5142225228156577
Average Recall: 0.5545636447591196
Average Precision: 0.5883452301824906
Average Accuracy: 0.5545636447591196
```