In [83]:

```
#make sure to download all of these packages
#python version that I used was 3.10, but most version of 3 should work
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
import plotly.express as px
import plotly.graph_objects as go
import seaborn as sns
%matplotlib inline

from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
```

In [84]:

```
#gets the values that are outside of the IQR for a specific set features (classes =
def outside_of_iqr(df : pd.DataFrame, investigate: str, classes: str) -> dict:

    #get all unique values of the class
    values = df[classes].unique()
    dictionary = dict()

    for i in values:
        specific_quality = df.loc[df[classes] == i]

        #get the IQR of the feature that is a certain class
        q1 = specific_quality[investigate].quantile(0.25)
        q3 = specific_quality[investigate].quantile(0.75)

        #get temporary values of total ones that are outside of the range of a part
        temp = specific_quality[investigate].loc[(specific_quality[investigate] > q
        dictionary.update({i: len(temp)})

    return dictionary
```

In [85]:

```
red_wine_data = pd.read_csv("data/winequality-red.csv",delimiter=";")
white_wine_data = pd.read_csv("data/winequality-white.csv",delimiter=";")
```

In [86]:

```
red_wine_data
```

Out[86]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphate
0	7.4	0.700	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	0.5
1	7.8	0.880	0.00	2.6	0.098	25.0	67.0	0.99680	3.20	0.6
2	7.8	0.760	0.04	2.3	0.092	15.0	54.0	0.99700	3.26	0.6
3	11.2	0.280	0.56	1.9	0.075	17.0	60.0	0.99800	3.16	0.5
4	7.4	0.700	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	0.5
...	...	...	...	...	...	...	...	...	...	.
1594	6.2	0.600	0.08	2.0	0.090	32.0	44.0	0.99490	3.45	0.5
1595	5.9	0.550	0.10	2.2	0.062	39.0	51.0	0.99512	3.52	0.7
1596	6.3	0.510	0.13	2.3	0.076	29.0	40.0	0.99574	3.42	0.7
1597	5.9	0.645	0.12	2.0	0.075	32.0	44.0	0.99547	3.57	0.7
1598	6.0	0.310	0.47	3.6	0.067	18.0	42.0	0.99549	3.39	0.6

1599 rows × 12 columns

In [87]:

```
red_wine_data.describe()
```

Out[87]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	tota
count	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599
mean	8.319637	0.527821	0.270976	2.538806	0.087467	15.874922	46
std	1.741096	0.179060	0.194801	1.409928	0.047065	10.460157	32
min	4.600000	0.120000	0.000000	0.900000	0.012000	1.000000	6
25%	7.100000	0.390000	0.090000	1.900000	0.070000	7.000000	22
50%	7.900000	0.520000	0.260000	2.200000	0.079000	14.000000	38
75%	9.200000	0.640000	0.420000	2.600000	0.090000	21.000000	62
max	15.900000	1.580000	1.000000	15.500000	0.611000	72.000000	289

Seems to be no missing data within all of the features at least for red

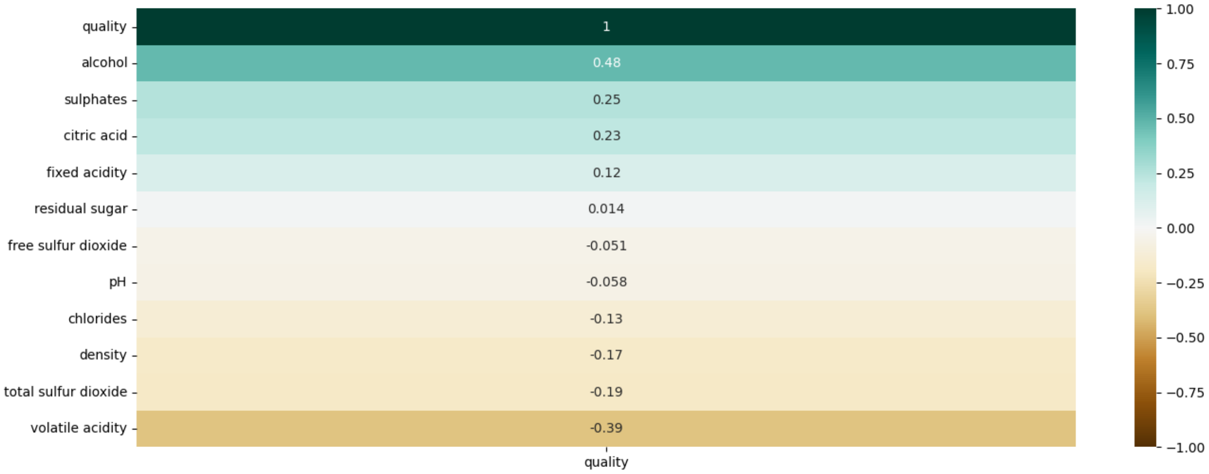
In [88]:

```
fig = go.Figure()
#getting the individual amounts of times the quality(lablels of the wine) appears i
quality_amounts_red = red_wine_data["quality"].groupby(red_wine_data["quality"]).co
quality_amounts_white = white_wine_data["quality"].groupby(white_wine_data["quality
#adding to a graph
fig.add_trace(go.Bar(x=quality_amounts_red.index,y=quality_amounts_red.values,name=
fig.add_trace(go.Bar(x=quality_amounts_white.index,y=quality_amounts_white.values,n

fig.update_layout(title="Quantity of Quality")
fig.show()
```

Overall the data set is imbalanced, so a accuracy measure will have to handle imbalanced. There is much more data in white, but at least it appears that they follow similar distribution

```
In [89]: """
vmin,vmax - the range of values for colormap(min-max)
cmap - sets the specific colormap to use
cetner - takes a float to centera color map
annot - if True sets the correlation values to appear
cbar - if False, the colorbar disapears

"""
plt.figure(figsize=(16,6))
sns.heatmap(red_wine_data.corr()[["quality"]].sort_values(by='quality',ascending=Fa
```

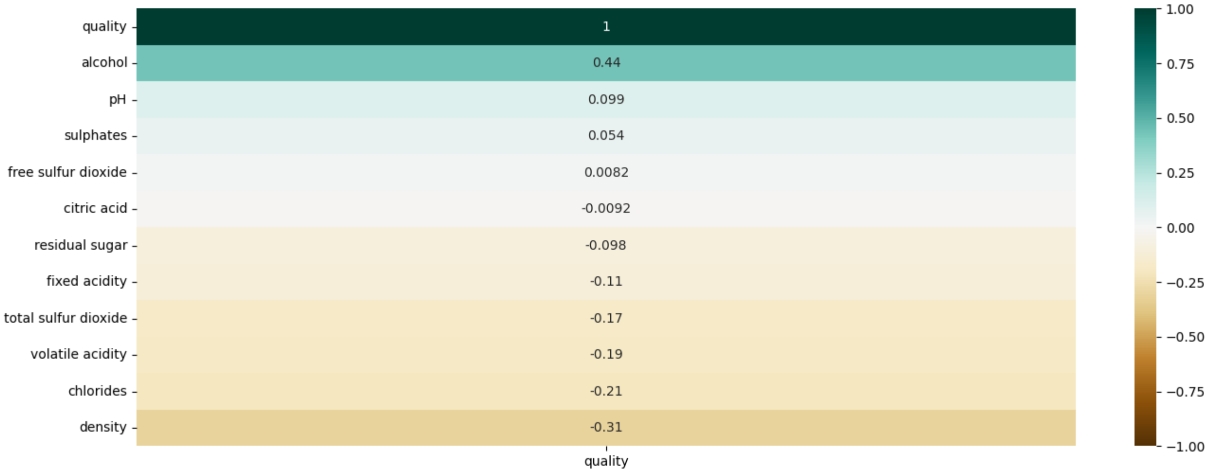
```
Out[89]: <AxesSubplot: >
```



So, this diagram shows the relationship (correlation) between a feature and our label (quality). If the value is high(1) or low(-1) that entails that there is a strong correlation between the label and the quality. Unfortunantly we don't have many high ones of the bat, high ones being (volatile acididty and alcohol). However, this is raw data that we can try nead the data a bit more (binning and such).

In [90]:
```
plt.figure(figsize=(16,6))
sns.heatmap(white_wine_data.corr()[["quality"]].sort_values(by='quality',ascending=
```

Out[90]:
```
<AxesSubplot: >
```



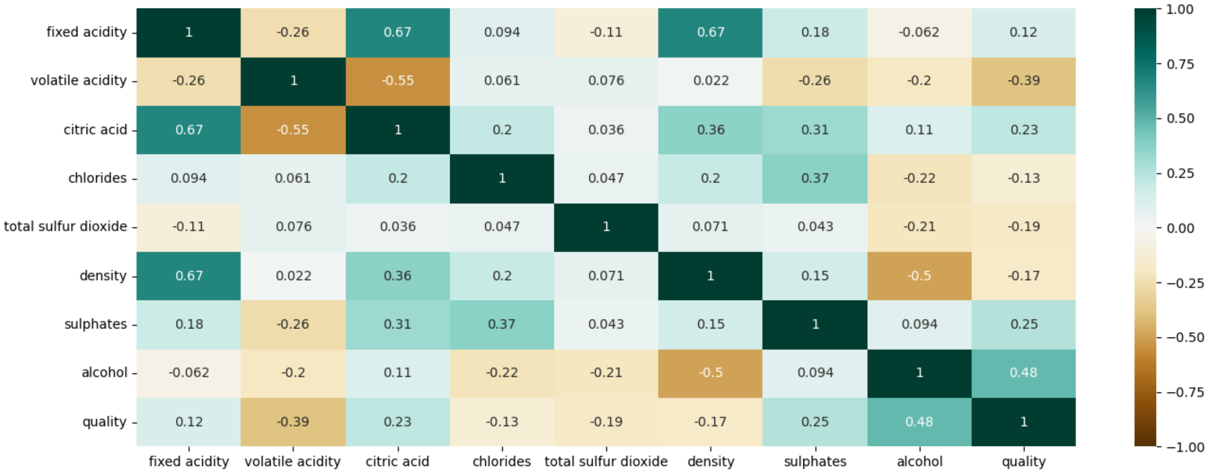
In white wines it appears that the lowest correlations (between -0.1 and 0.1) are the same (free sulfur dioxide,residual sugar,ph), so it seems fair to drop these from consideration; however it might be worthwhile to check later if there are some values that are throwing off others. For inital EDA these will be dropped.

Citric acid and suplates appeared to show strong correlation in red but not white. Will need to combine data sets and do EDA on all at the same time

Overall, it seems that red wine has much stonger correlation between specific attributes; therefore we are going to use the red wine for our models. This comes with an unfortunate tradeoff, as the white wine dataset has more data overall, which might pose issues with our neural network attempt.

In [91]:
```
red_wine_data = red_wine_data.drop(["free sulfur dioxide","residual sugar" ,"pH"],a
```

In [92]:
```
"""
    Overall shows the direct or indirect correlation between features of a dataset.
    of two features. Measuring anything beyond the relationship of two is not possi
"""
plt.figure(figsize=(16, 6))
heatmap = sns.heatmap(red_wine_data.corr(), vmin=-1, vmax=1, annot=True, cmap='BrBG
```



Attribute: Alcohol

In [93]:
```
#Alcohol stuff
from plotly.subplots import make_subplots

fig = make_subplots(rows=2,cols=2)

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

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

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

Appears to be a bit skewed but overal, not a huge amount of problems.

In [94]:

```
px.box(red_wine_data,x="quality",y="alcohol",title="Alchol vs Quality")
```

Can see a bit of a trend here, as the value of alchol increases so does the actual value of alchol, and it appears that there is a considerable jump from 5 to 6

In [95]:
```
mean_red = red_wine_data.loc[(red_wine_data["quality"] >= 7)]["alcohol"].mean()
greater_than_mean = red_wine_data.loc[red_wine_data["quality"] < 7]["alcohol"].valu

less_than_mean = red_wine_data.loc[(red_wine_data["quality"] >= 7)]["alcohol"].valu
sum(less_than_mean), sum(greater_than_mean),len(red_wine_data)
```

Out[95]:
```
(107, 140, 1599)
```

It seems that we might want to split the classes into just two binary variables, overall it seems that our bet will to have (1-5) and (6-10)

In [96]:
```
red_wine_data["alcohol_higher"] = 0
red_wine_data.loc[red_wine_data["alcohol"] >= mean_red, "alcohol_higher"] =1
```

Attribute: density

In [97]:

```
#density seems to be very very small differences, I doubt this will be any help but
red_wine_data["density"].describe()
```

Out[97]:

```
count    1599.000000
mean        0.996747
std         0.001887
min         0.990070
25%         0.995600
50%         0.996750
75%         0.997835
max         1.003690
Name: density, dtype: float64
```

In [98]:

```
px.histogram(red_wine_data,x="density")
```

In [99]:

```
px.box(red_wine_data,x="quality",y="density",title="Red Wine")
```

almost a perfect distribution for normal

Attribute: vaolatile acidity

In [100…

```
red_wine_data["volatile acidity"].describe()
```

Out[100]:

```
count    1599.000000
mean        0.527821
std         0.179060
min         0.120000
25%         0.390000
50%         0.520000
75%         0.640000
max         1.580000
Name: volatile acidity, dtype: float64
```

There is pretty good standard deviation, so plausible that we might be able to use this data pretty well for banding.

In [101…

```
fig = make_subplots(rows=1,cols=2)
```

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

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


fig.update_layout(height=600, width=1200, title_text="Volatile Acidity")
fig.show()
```

Bit skewed but more normal if anything

In [102…

```
px.box(red_wine_data,x="quality",y="volatile acidity",title="Red Wine")
```

In [103…

```
means_of_red = red_wine_data.groupby(red_wine_data["quality"]).mean()
medians_of_red = red_wine_data.groupby(red_wine_data["quality"]).median()
```

In [104…

```
means_of_red["volatile acidity"],medians_of_red["volatile acidity"]
```

Out[104]:

```
(quality
 3    0.884500
 4    0.693962
 5    0.577041
 6    0.497484
 7    0.403920
 8    0.423333
 Name: volatile acidity, dtype: float64,
 quality
 3    0.845
 4    0.670
 5    0.580
 6    0.490
 7    0.370
 8    0.370
 Name: volatile acidity, dtype: float64)
```

In [105…

```
#geting tuples that have a quality that is greater than five
greater_than_five = red_wine_data.loc[red_wine_data["quality"] > 5]
```

```
greater_than_five["volatile acidity"].describe()
```

Out[105]:

```
count    855.000000
mean       0.474146
std        0.161999
min        0.120000
25%        0.350000
50%        0.460000
75%        0.580000
max        1.040000
Name: volatile acidity, dtype: float64
```

In [106…

```
#getting the avlues that are greater than five, and have less than 0.6 volatile aci
high_quality_less = greater_than_five.loc[greater_than_five["volatile acidity"] < 0
#getting the values that are greater than five, and have more than 0.6 volatile aci
high_quality_more = greater_than_five.loc[greater_than_five["volatile acidity"] >=
print("Amount that fit the band " + str(len(high_quality_less)))
print("Amount that do not fit the band " + str(len(high_quality_more)))
```

```
Amount that fit the band 664
Amount that do not fit the band 191
```

In [107…

```
less_than_five = red_wine_data[red_wine_data["quality"] < 5]
less_than_five["volatile acidity"].describe()
```

Out[107]:

```
count    63.000000
mean      0.724206
std       0.247970
min       0.230000
25%       0.565000
50%       0.680000
75%       0.882500
max       1.580000
Name: volatile acidity, dtype: float64
```

In [108…

```
high_quality_less = less_than_five.loc[less_than_five["volatile acidity"] <= 0.6]
high_quality_more = less_than_five.loc[less_than_five["volatile acidity"] > 0.6]
print("Amount that fit the band " + str(len(high_quality_more)))
print("Amount that do not fit the band " + str(len(high_quality_less)))
```

```
Amount that fit the band 42
Amount that do not fit the band 21
```

In [109…

```
#creating a binary feature
red_wine_data["va_high"] = 0
red_wine_data.loc[red_wine_data["volatile acidity"] >= 0.6, "va_high"] = 1
red_wine_data
```

Out[109]:

	fixed acidity	volatile acidity	citric acid	chlorides	total sulfur dioxide	density	sulphates	alcohol	quality	alco
0	7.4	0.700	0.00	0.076	34.0	0.99780	0.56	9.4	5	
1	7.8	0.880	0.00	0.098	67.0	0.99680	0.68	9.8	5	
2	7.8	0.760	0.04	0.092	54.0	0.99700	0.65	9.8	5	
3	11.2	0.280	0.56	0.075	60.0	0.99800	0.58	9.8	6	
4	7.4	0.700	0.00	0.076	34.0	0.99780	0.56	9.4	5	
...	...	...	...	...	...	...	...	...	...	
1594	6.2	0.600	0.08	0.090	44.0	0.99490	0.58	10.5	5	
1595	5.9	0.550	0.10	0.062	51.0	0.99512	0.76	11.2	6	
1596	6.3	0.510	0.13	0.076	40.0	0.99574	0.75	11.0	6	
1597	5.9	0.645	0.12	0.075	44.0	0.99547	0.71	10.2	5	
1598	6.0	0.310	0.47	0.067	42.0	0.99549	0.66	11.0	6	

1599 rows × 11 columns

This also might cause overfitting due to the correlation with volatile acididty in general. However, it should at least be helpful in determining red wines better. Check confusion matrix at the end and run models with and without the values

---

Attribute : Total sulfur dioxide

In [110…

```
red_wine_data["total sulfur dioxide"].describe(), white_wine_data["total sulfur dio
```

```
Out[110]: (count    1599.000000
           mean       46.467792
           std        32.895324
           min         6.000000
           25%        22.000000
           50%        38.000000
           75%        62.000000
           max       289.000000
           Name: total sulfur dioxide, dtype: float64,
           count    4898.000000
           mean      138.360657
           std        42.498065
           min         9.000000
           25%       108.000000
           50%       134.000000
           75%       167.000000
           max       440.000000
           Name: total sulfur dioxide, dtype: float64)
```

```
In [111…  px.histogram(red_wine_data,x="total sulfur dioxide")
```

```
In [112…  px.box(red_wine_data,x="quality",y="total sulfur dioxide",title="Red Wine TSD")
```

This data is highly skewed and it might be worthwhile to try and to a transformation to smooth it out. Either smooth or turn into a standard scaler.

Attribute: citiric acid

In [113…
```
red_wine_data["citric acid"].describe()
```

Out[113]:
```
count    1599.000000
mean        0.270976
std         0.194801
min         0.000000
25%         0.090000
50%         0.260000
75%         0.420000
max         1.000000
Name: citric acid, dtype: float64
```

In [114…
```
px.histogram(red_wine_data,x="citric acid")
```

In [115…

```
px.box(red_wine_data,x="quality",y="citric acid",title="Red Wine Citric acid")
```

In [116…

```
outside_of_iqr(red_wine_data, "citric acid", "quality")
```

Out[116]:

```
{5: 327, 6: 309, 7: 96, 4: 25, 8: 9, 3: 6}
```

Even though it might look like there could be not real outliers, there is still a great deal of variance within the upper and lower, fence out outside of the IQR

Main winners: ones in parenthesis are simplified attributes. Might be helpful in certain algorithms, we can try to use both main attribute and other attribute, but should be warry of overfitting

Positive - Alcohol (alcohol_higher), sulphates, citric acid

Negative - Volatile acidity (va_high), Total sulfur dioxide, density

Possible drops -- fixed acidity, and chlorides

In [117…

```
red_wine_data["graphing qualities"] = ""
red_wine_data.loc[red_wine_data["quality"] > 6, "graphing qualities"] = "7-8"
red_wine_data.loc[red_wine_data["quality"] < 5, "graphing qualities"] = "3-4"
red_wine_data.loc[(red_wine_data["quality"] < 7) & (red_wine_data["quality"] > 4),
```

Cholrides/Sulphates : sulphates have a good correlation, but cholorides do not. However, both have a decent correlation together. It might be worth the time to remove chlorides to reduce confusion in the model, or integrate it somehow but lose unimportant information.

In [118…

```
#Alcohol stuff
fig = make_subplots(rows=2,cols=3)

#histogram for red
fig.append_trace(go.Histogram(
    x=red_wine_data["chlorides"],
    nbinsx=10,
    name="chloride hist"
), row=1,col=1)

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

fig.append_trace(go.Box(
    x=red_wine_data["quality"],
    y=red_wine_data["chlorides"],
    name="quality chlorides"
), row=1,col=3)

fig.append_trace(go.Histogram(
    x=red_wine_data["sulphates"],
    nbinsx=10,
    name="sulphate hist"
), row=2,col=1)

#boxplot for red
fig.append_trace(go.Box(
    x=red_wine_data["sulphates"],
    name="sulphate box"
),row=2,col=2)

fig.append_trace(go.Box(
    x=red_wine_data["quality"],
    y=red_wine_data["sulphates"],
    name="quality sulphates"
), row=2,col=3)

fig.update_layout(height=600, width=1250, title_text="Sulphates/Chlorides")
fig.show()
```

In [119…

```
fig = px.scatter(red_wine_data, x="chlorides", y="sulphates", color="graphing quali
fig.show()
```

Both have very similar distributions, and the correlation just appears to arise that they cluster around a similar area. There is no way to actually discern where certain values would be in their combination, so I would wager we could just drop chlorides.

---

fixed acidity/density/citric acid

In [120…

```
#Alcohol stuff
fig = make_subplots(rows=3,cols=3)

fig.append_trace(go.Histogram(
    x=red_wine_data["fixed acidity"],
    name="fixed acidity hist"
), row=1,col=1)

fig.append_trace(go.Box(
    x=red_wine_data["fixed acidity"],
    name="fixed acidity box"
),row=1,col=2)

fig.append_trace(go.Box(
    x=red_wine_data["quality"],
    y=red_wine_data["fixed acidity"],
    name="quality fixed acidity"
), row=1,col=3)

fig.append_trace(go.Histogram(
    x=red_wine_data["density"],
    name="density hist"
), row=2,col=1)

fig.append_trace(go.Box(
    x=red_wine_data["density"],
    name="density box"
),row=2,col=2)

fig.append_trace(go.Box(
    x=red_wine_data["quality"],
    y=red_wine_data["density"],
    name="quality density"
), row=2,col=3)

fig.append_trace(go.Histogram(
    x=red_wine_data["citric acid"],
    name="citric acid hist"
), row=3,col=1)

fig.append_trace(go.Box(
    x=red_wine_data["citric acid"],
    name="citric acid box"
),row=3,col=2)

fig.append_trace(go.Box(
    x=red_wine_data["quality"],
    y=red_wine_data["citric acid"],
    name="quality citric acid"
), row=3,col=3)


fig.update_layout(height=600, width=1250, title_text="Fixed Acidity/Density/Citric
fig.show()
```

Fixed acidity, and density have a similar distribution if you do not consider the scaling of the values. On the other hand, citric acid seems to be much different overall.

In [121…

```
red_wine_data = red_wine_data.drop(["graphing qualities"],axis=1)
red_train, red_test = train_test_split(red_wine_data,test_size=0.2,stratify=red_win
```

When it comes to the analysis of individual features, there does not seem anything more we can do except scale and remove outliers when needed.For KNN we should scale, DT won't need much, and NN will need scaling of some kind. However, there is a chance that there are interactions between features in more complex way. If we had more time we could try and figure out, but I have no experience beyond direct correlation. Therefore, we can leave that to another day.

In [122…

```
red_train.to_csv("data/red_wine_train.csv")
red_test.to_csv("data/red_wine_test.csv")
```

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
import keras
import sklearn
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
```

In [2]:
```
train = pd.read_csv('data/red_wine_train.csv',index_col=0)

#dropping columns based on heatmap correlations in EDA
train = train.drop(['fixed acidity', 'chlorides', 'density', 'total sulfur dioxide'
train.head()
```

Out[2]:

	volatile acidity	citric acid	sulphates	alcohol	quality
1348	0.655	0.03	0.39	9.5	5
117	0.560	0.12	0.50	9.4	6
1150	0.330	0.32	0.76	12.8	7
235	0.630	0.00	0.58	9.0	6
91	0.490	0.28	1.95	9.9	6

In [3]:
```
test = pd.read_csv('data/red_wine_test.csv', index_col=0)
test = test.drop(['fixed acidity', 'chlorides', 'density', 'total sulfur dioxide',
```

In [4]:
```
###begin raw
```

In [5]:
```
x_train = train.drop('quality',axis=1) #training df without class column

x_test = test.drop('quality',axis=1) #testing df without class column

y_train = train['quality'] #training df only class column

y_test = test['quality'] #testing df only class column

y_train = y_train.map({3:0, 4:1, 5:2, 6:3, 7:4, 8:5})
y_test = y_test.map({3:0, 4:1, 5:2, 6:3, 7:4, 8:5})
```

In [6]:
```
n_inputs = [x_train.shape[1]] #n cols => n inputs in model
n_units = 12
n_batch = 100
n_epochs = 10
```

In [7]:
```
tf.keras.backend.clear_session() #resets parameters, necessary before each new mode
keras.utils.set_random_seed(0) #setting random seed for the entire program

modelRaw = tf.keras.Sequential([tf.keras.layers.Dense(units=n_units, activation='re
                                tf.keras.layers.Dense(units=6, activation='softmax'
modelRaw.summary()

modelRaw.compile(optimizer='adam',loss='mae', metrics=['accuracy'])
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 dense (Dense)               (None, 12)                60        
                                                                 
 dense_1 (Dense)             (None, 6)                 78        
                                                                 
=================================================================
Total params: 138
Trainable params: 138
Non-trainable params: 0
_________________________________________________________________
```

In [8]:
```
modelRaw.fit(x_train, y_train, batch_size=n_batch, epochs=n_epochs)
```

```
Epoch 1/10
13/13 [==============================] - 0s 2ms/step - loss: 2.4726 - accuracy: 0.39
87
Epoch 2/10
13/13 [==============================] - 0s 2ms/step - loss: 2.4726 - accuracy: 0.39
87
Epoch 3/10
13/13 [==============================] - 0s 2ms/step - loss: 2.4726 - accuracy: 0.39
87
Epoch 4/10
13/13 [==============================] - 0s 2ms/step - loss: 2.4726 - accuracy: 0.39
87
Epoch 5/10
13/13 [==============================] - 0s 2ms/step - loss: 2.4726 - accuracy: 0.39
87
Epoch 6/10
13/13 [==============================] - 0s 2ms/step - loss: 2.4726 - accuracy: 0.39
87
Epoch 7/10
13/13 [==============================] - 0s 2ms/step - loss: 2.4726 - accuracy: 0.39
87
Epoch 8/10
13/13 [==============================] - 0s 2ms/step - loss: 2.4726 - accuracy: 0.39
87
Epoch 9/10
13/13 [==============================] - 0s 1ms/step - loss: 2.4726 - accuracy: 0.39
87
Epoch 10/10
13/13 [==============================] - 0s 2ms/step - loss: 2.4726 - accuracy: 0.39
87
```

Out[8]:
```
<keras.callbacks.History at 0x26ca4736f40>
```

In [9]:

```
###end raw

###begin scaled
```

In [10]:

```
#recombining train & test to get overall max and min values so test and train are s
whole_set = pd.concat([train,test])
whole_set.describe() #summary to show all columns have varying scales
```

Out[10]:

	volatile acidity	citric acid	sulphates	alcohol	quality
count	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000
mean	0.527821	0.270976	0.658149	10.422983	5.636023
std	0.179060	0.194801	0.169507	1.065668	0.807569
min	0.120000	0.000000	0.330000	8.400000	3.000000
25%	0.390000	0.090000	0.550000	9.500000	5.000000
50%	0.520000	0.260000	0.620000	10.200000	6.000000
75%	0.640000	0.420000	0.730000	11.100000	6.000000
max	1.580000	1.000000	2.000000	14.900000	8.000000

In [11]:

```
#drop quality class label column before scaling
whole_set.drop('quality',axis=1, inplace=True)

#build scaler
scaler = MinMaxScaler() #build scaler
scaler.fit(whole_set) #fit scaler to entire df w/o quality col
```

Out[11]:

```
MinMaxScaler()
```

In [12]:

```
x_trainScaled=scaler.transform(x_train)
x_testScaled=scaler.transform(x_test)

#make transformed data in a dataframe (.transform returns arrays, we want df) using
x_trainScaled = pd.DataFrame(x_trainScaled, columns=x_train.columns)
x_testScaled = pd.DataFrame(x_testScaled, columns=x_test.columns)
```

In [13]:

```
x_trainScaled.describe() #now each attr column has min 0 and max 1
```

Out[13]:

	volatile acidity	citric acid	sulphates	alcohol
count	1279.000000	1279.000000	1279.000000	1279.000000
mean	0.281668	0.266059	0.195751	0.310976
std	0.120700	0.193606	0.099773	0.166797
min	0.000000	0.000000	0.000000	0.000000
25%	0.191781	0.090000	0.131737	0.169231
50%	0.273973	0.250000	0.173653	0.276923
75%	0.356164	0.420000	0.239521	0.415385
max	1.000000	1.000000	1.000000	1.000000

In [14]:

```
tf.keras.backend.clear_session() #resets parameters, necessary before each new mode

modelScaled = tf.keras.Sequential([tf.keras.layers.Dense(units=n_units, activation=
                                    tf.keras.layers.Dense(units=6, activation='soft

modelScaled.summary()

modelScaled.compile(optimizer='adam', loss='mae', metrics=['accuracy'])

modelScaled.fit(x_trainScaled, y_train, batch_size=n_batch, epochs=n_epochs)
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 12)                60

 dense_1 (Dense)             (None, 6)                 78

=================================================================
Total params: 138
Trainable params: 138
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
13/13 [==============================] - 0s 2ms/step - loss: 2.4726 - accuracy: 0.10
16
Epoch 2/10
13/13 [==============================] - 0s 2ms/step - loss: 2.4726 - accuracy: 0.10
16
Epoch 3/10
13/13 [==============================] - 0s 2ms/step - loss: 2.4726 - accuracy: 0.10
16
Epoch 4/10
13/13 [==============================] - 0s 2ms/step - loss: 2.4726 - accuracy: 0.10
16
Epoch 5/10
13/13 [==============================] - 0s 2ms/step - loss: 2.4726 - accuracy: 0.10
16
Epoch 6/10
13/13 [==============================] - 0s 2ms/step - loss: 2.4726 - accuracy: 0.10
16
Epoch 7/10
13/13 [==============================] - 0s 2ms/step - loss: 2.4726 - accuracy: 0.10
16
Epoch 8/10
13/13 [==============================] - 0s 1ms/step - loss: 2.4726 - accuracy: 0.10
16
Epoch 9/10
13/13 [==============================] - 0s 2ms/step - loss: 2.4726 - accuracy: 0.10
16
Epoch 10/10
13/13 [==============================] - 0s 2ms/step - loss: 2.4726 - accuracy: 0.10
16
```

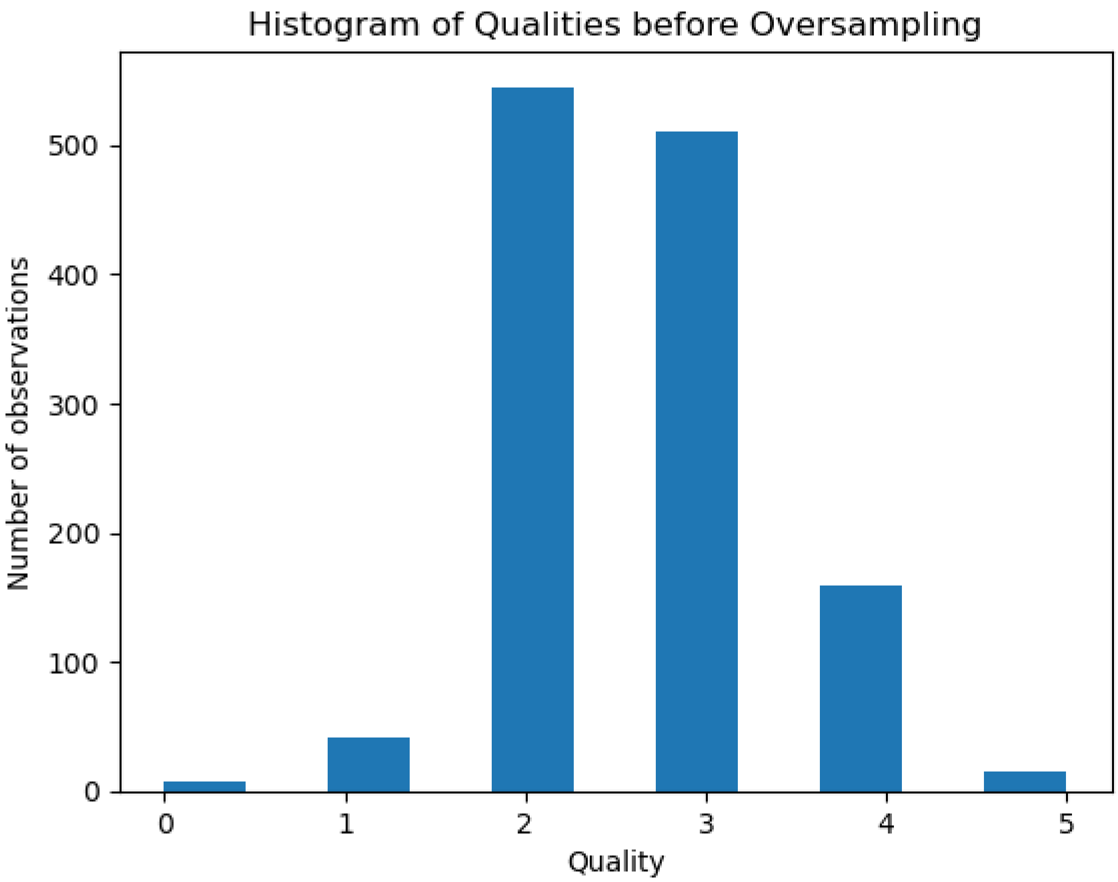
```
Out[14]: <keras.callbacks.History at 0x26ca5aa44f0>
```

In [15]:

```
###end scaled

###begin oversampled
```

In [16]:

```
#plot to show that oversampling balanced the data
plt.hist(y_train, bins=11)
plt.title("Histogram of Qualities before Oversampling")
plt.ylabel('Number of observations')
plt.xlabel('Quality')
plt.show()
```



In [17]:

```
#dividing train into separate dfs for each class value
qual3 = (train[train["quality"]==3])
qual4 = (train[train["quality"]==4])
qual5 = (train[train["quality"]==5])
qual6 = (train[train["quality"]==6])
qual7 = (train[train["quality"]==7])
qual8 = (train[train["quality"]==8])

#number of samples per class label to determine inbalance
n_qual3 = len(qual3) #8
n_qual4 = len(qual4) #42
n_qual5 = len(qual5) #545
n_qual6 = len(qual6) #510
n_qual7 = len(qual7) #159
n_qual8 = len(qual8) #15

n_max = max(n_qual3, n_qual4, n_qual5, n_qual6, n_qual7, n_qual8) #545
```

In [18]:

```
#oversample so that each class has the same number of observations,
#equal to the number of observations of quality 5
qual3Oversampled = qual3.sample(n_max, replace=True)
qual4Oversampled = qual4.sample(n_max, replace=True)
qual6Oversampled = qual6.sample(n_max, replace=True)
qual7Oversampled = qual7.sample(n_max, replace=True)
qual8Oversampled = qual8.sample(n_max, replace=True)
```

In [19]:

```
#concat back into one df. this is unscaled
```

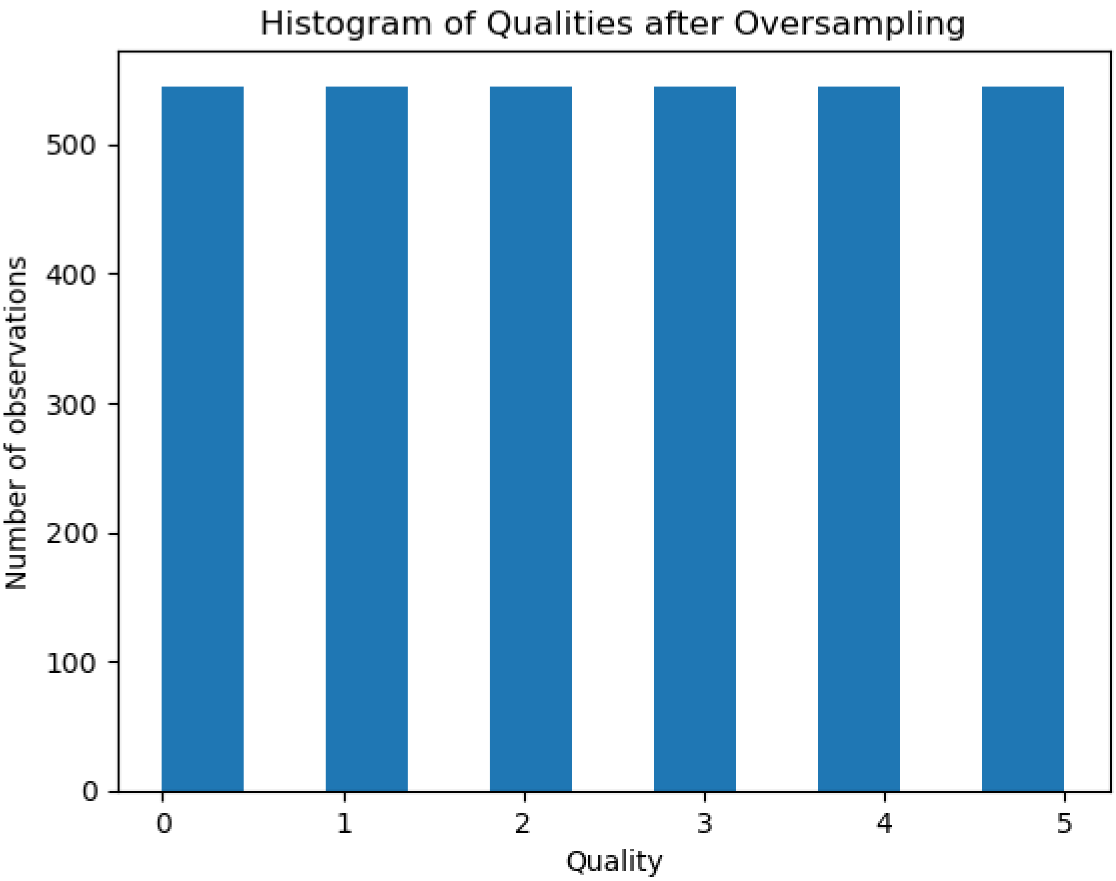
```
trainOversampled = pd.concat([qual3Oversampled, qual4Oversampled, qual5, qual6Overs
```

In [20]:

```
#scaling oversampled
x_trainOversampled = trainOversampled.drop('quality',axis=1) #training df without c

y_trainOversampled = trainOversampled['quality'] #training df only class column

#map values of train from [3,8] to [0,5]
y_trainOversampled = y_trainOversampled.map({3:0, 4:1, 5:2, 6:3, 7:4, 8:5})
```

In [21]:

```
#plot to show that oversampling balanced the data
plt.hist(y_trainOversampled, bins=11)
plt.title("Histogram of Qualities after Oversampling")
plt.ylabel('Number of observations')
plt.xlabel('Quality')
plt.show()
```



In [22]:

```
#scale trainOversampled w scaler from before
#the same scaler will work bcus our oversampled data will have same range as raw da
x_trainOversampledScaled=scaler.transform(x_trainOversampled)

#make transformed data in a dataframe (.transform returns arrays, we want df) using
x_trainScaledOversampled = pd.DataFrame(x_trainOversampledScaled, columns=x_train.c
```

In [23]:

```
tf.keras.backend.clear_session() #resets parameters, necessary before each new mode

modelOversampledScaled = tf.keras.Sequential([tf.keras.layers.Dense(units=n_units, 
                                              tf.keras.layers.Dense(units=6, acti
```

```python
modelOversampledScaled.summary()

modelOversampledScaled.compile(optimizer='adam', loss='mae', metrics=['accuracy'])

modelOversampledScaled.fit(x_trainOversampledScaled, y_trainOversampled, batch_size
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 dense (Dense)               (None, 12)                60        
                                                                 
 dense_1 (Dense)             (None, 6)                 78        
                                                                 
=================================================================
Total params: 138
Trainable params: 138
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
33/33 [==============================] - 1s 2ms/step - loss: 2.3889 - accuracy: 0.13
06
Epoch 2/10
33/33 [==============================] - 0s 2ms/step - loss: 2.3889 - accuracy: 0.13
06
Epoch 3/10
33/33 [==============================] - 0s 1ms/step - loss: 2.3889 - accuracy: 0.13
06
Epoch 4/10
33/33 [==============================] - 0s 2ms/step - loss: 2.3889 - accuracy: 0.13
06
Epoch 5/10
33/33 [==============================] - 0s 2ms/step - loss: 2.3889 - accuracy: 0.13
06
Epoch 6/10
33/33 [==============================] - 0s 1ms/step - loss: 2.3889 - accuracy: 0.13
06
Epoch 7/10
33/33 [==============================] - 0s 1ms/step - loss: 2.3889 - accuracy: 0.13
06
Epoch 8/10
33/33 [==============================] - 0s 2ms/step - loss: 2.3889 - accuracy: 0.13
06
Epoch 9/10
33/33 [==============================] - 0s 2ms/step - loss: 2.3889 - accuracy: 0.13
06
Epoch 10/10
33/33 [==============================] - 0s 2ms/step - loss: 2.3889 - accuracy: 0.13
06
```

```
Out[23]: <keras.callbacks.History at 0x26ca5d13640>
```

```python
In [24]: ###end oversampled
```

```
###begin weighted
#returning to scaled data, then calculating weights and scaling
```

In [25]:

```
#make each column a numpy array for class weights computation
qual3_numpy = qual3['quality'].to_numpy()
qual4_numpy = qual4['quality'].to_numpy()
qual5_numpy = qual5['quality'].to_numpy()
qual6_numpy = qual6['quality'].to_numpy()
qual7_numpy = qual7['quality'].to_numpy()
qual8_numpy = qual8['quality'].to_numpy()

#combine numpy arrays into whole numpy and set variable for class values
whole_numpy = np.concatenate((qual3_numpy, qual4_numpy, qual5_numpy, qual6_numpy, q
unique_classes = np.unique(whole_numpy)

#compute weights with sklearn method
weights = sklearn.utils.class_weight.compute_class_weight(class_weight='balanced',
weightsDict = {i:w for i,w in enumerate(weights)}
```

In [26]:

```
tf.keras.backend.clear_session() #resets parameters, necessary before each new mode

modelWeighted = tf.keras.Sequential([tf.keras.layers.Dense(units=n_units, activatio
                                     tf.keras.layers.Dense(units=6, activation='sof

modelWeighted.summary()

modelWeighted.compile(optimizer='adam', loss='mae', metrics=['accuracy'])

modelWeighted.fit(x_trainScaled, y_train, batch_size=n_batch, epochs=n_epochs, clas
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 dense (Dense)               (None, 12)                60        
                                                                 
 dense_1 (Dense)             (None, 6)                 78        
                                                                 
=================================================================
Total params: 138
Trainable params: 138
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
13/13 [==============================] - 1s 3ms/step - loss: 2.3889 - accuracy: 0.42
69
Epoch 2/10
13/13 [==============================] - 0s 2ms/step - loss: 2.3889 - accuracy: 0.42
69
Epoch 3/10
13/13 [==============================] - 0s 2ms/step - loss: 2.3889 - accuracy: 0.42
69
Epoch 4/10
13/13 [==============================] - 0s 2ms/step - loss: 2.3889 - accuracy: 0.42
69
Epoch 5/10
13/13 [==============================] - 0s 2ms/step - loss: 2.3889 - accuracy: 0.42
69
Epoch 6/10
13/13 [==============================] - 0s 2ms/step - loss: 2.3889 - accuracy: 0.42
69
Epoch 7/10
13/13 [==============================] - 0s 2ms/step - loss: 2.3889 - accuracy: 0.42
69
Epoch 8/10
13/13 [==============================] - 0s 2ms/step - loss: 2.3889 - accuracy: 0.42
69
Epoch 9/10
13/13 [==============================] - 0s 2ms/step - loss: 2.3889 - accuracy: 0.42
69
Epoch 10/10
13/13 [==============================] - 0s 2ms/step - loss: 2.3889 - accuracy: 0.42
69
```

Out[26]:

```
<keras.callbacks.History at 0x26ca7f117f0>
```

In [27]:

```python
#function to output accuracy, precision, recall, and F1 for each model
#input model name, model object, and appropriate test set x_test (scaled or unscale
#output header and metric scores as percents

def outputMetrics(modelname, model, x_test):
    y_pred = model.predict(x_test)

    y_pred_df = (pd.DataFrame(y_pred))

    y_pred_label = y_pred.argmax(axis=1)
    f1 = f1_score(y_test,y_pred_label,average="weighted",zero_division=1)
    recall = recall_score(y_test,y_pred_label,average="weighted",zero_division=1)
    precision = precision_score(y_test, y_pred_label,average="weighted",zero_divisi
    accuracy = accuracy_score(y_test,y_pred_label)
    print(modelname, "Metrics:")
    print("Accuracy: {:.2f}%".format(accuracy*100))
    print("Precision: {:.2f}%".format(precision*100))
    print("Recall: {:.2f}%".format(recall*100))
    print("F1: {:.2f}%".format(f1*100))

outputMetrics("Raw Model", modelRaw, x_test)
outputMetrics("Scaled Model", modelScaled, x_testScaled)
outputMetrics("Scaled & Oversampled Model", modelOversampledScaled, x_testScaled)
outputMetrics("Weighted Model", modelWeighted, x_testScaled)
```

```
10/10 [==============================] - 0s 2ms/step
Raw Model Metrics:
Accuracy: 40.00%
Precision: 76.00%
Recall: 40.00%
F1: 22.86%
10/10 [==============================] - 0s 1ms/step
Scaled Model Metrics:
Accuracy: 5.62%
Precision: 50.71%
Recall: 5.62%
F1: 5.63%
10/10 [==============================] - 0s 1ms/step
Scaled & Oversampled Model Metrics:
Accuracy: 3.12%
Precision: 56.69%
Recall: 3.12%
F1: 1.37%
10/10 [==============================] - 0s 2ms/step
Weighted Model Metrics:
Accuracy: 42.19%
Precision: 74.86%
Recall: 42.19%
F1: 25.22%
```

```python
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.model_selection import cross_val_score
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler

# Load the wine dataset from a CSV file
red_train = pd.read_csv('red_wine_train.csv', index_col=0)
red_test = pd.read_csv('red_wine_test.csv', index_col=0)

total = pd.concat([red_train, red_test])
total = total.drop(["quality"], axis=1)
scaler = StandardScaler()
scaler.fit(total)


# Split the data into training and test sets
X_train = red_train.drop("quality", axis=1)
X_test = red_train.drop("quality", axis=1)
y_train = red_train["quality"]
y_test = red_train["quality"]

X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

X_train = pd.DataFrame(X_train_scaled, columns=X_train.columns)
X_test = pd.DataFrame(X_test_scaled, columns=X_test.columns)

X = pd.concat([X_train, X_test])
y = pd.concat([y_test, y_train])

# Train a KNN classifier with k=2 on the training data
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)

# Make predictions on the test data
y_pred = knn.predict(X_test)

# Compute the accuracy of the predictions
prediction_accuracy = accuracy_score(y_test, y_pred)
print("Prediction Accuracy: {:.2f}%".format(prediction_accuracy*100))

# Compute cross-validation scores for different values of k
k_values = [i for i in range(2, 31)]
scores = []


for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    score = cross_val_score(knn, X_train, y_train, cv=5)
    scores.append(np.mean(score))

# Plot the cross-validation scores vs. k
plt.plot(k_values, scores, marker='o')
plt.xlabel("K values")
plt.ylabel("Accuracy score")

# Train a KNN classifier with the best value of k on the training data
knn = KNeighborsClassifier(n_neighbors=22)
knn.fit(X_train, y_train)

# Make predictions on the test data
y_pred = knn.predict(X_test)

# Compute scores
```

```python
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average='weighted', zero_division=1)
recall = recall_score(y_test, y_pred, average='weighted', zero_division=1)
f1 = f1_score(y_test, y_pred, average='weighted', zero_division=1)

# Print
print("Accuracy: {:.2f}%".format(accuracy*100))
print("Precision: {:.2f}%".format(precision*100))
print("Recall: {:.2f}%".format(recall*100))
print("F1 score: {:.2f}%".format(f1*100))

plt.show()
```

In [18]:
```
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.metrics import f1_score,accuracy_score,precision_score,recall_score
from sklearn.model_selection import RandomizedSearchCV
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 [19]:
```
red_train = pd.read_csv("data/red_wine_train.csv",index_col=0)
red_test = pd.read_csv("data/red_wine_test.csv",index_col=0)
```

In [20]:
```
red_train.describe()
```

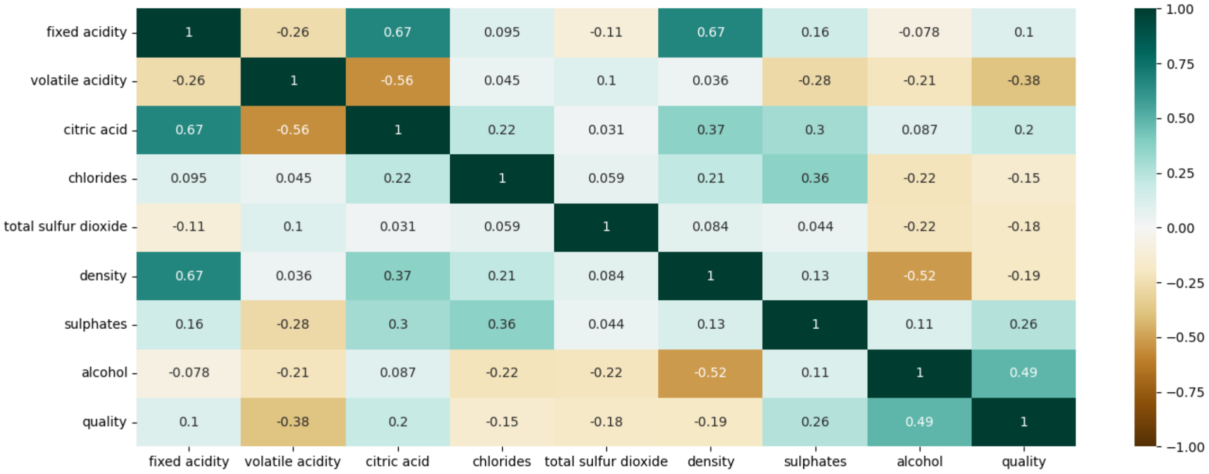
Out[20]:

	fixed acidity	volatile acidity	citric acid	chlorides	total sulfur dioxide	density	su
count	1279.000000	1279.000000	1279.000000	1279.000000	1279.000000	1279.000000	1279
mean	8.282877	0.531235	0.266059	0.087127	46.464816	0.996715	0
std	1.717760	0.176222	0.193606	0.047654	32.173470	0.001916	0
min	4.900000	0.120000	0.000000	0.012000	6.000000	0.990070	0
25%	7.100000	0.400000	0.090000	0.070000	23.000000	0.995545	0
50%	7.900000	0.520000	0.250000	0.079000	38.000000	0.996700	0
75%	9.100000	0.640000	0.420000	0.090000	62.000000	0.997800	0
max	15.900000	1.580000	1.000000	0.611000	278.000000	1.003690	2

First Attempt: outlier removal, and dropping fixed acidity and chlordies, and the two binary variables (no binning)

In [21]:
```
#dropping binary values that were created just
red_train = red_train.drop(["alcohol_higher", "va_high"], axis=1)
red_test= red_test.drop(["alcohol_higher", "va_high"], axis=1)
```

In [22]:
```
plt.figure(figsize=(16, 6))
heatmap = sns.heatmap(red_train.corr(), vmin=-1, vmax=1, annot=True, cmap='BrBG')
```



Preprocessing

For decision trees, there typically no reason to do much when it comes to altering the data. Decision trees do not have much problems with scaled, normalized, and other types continous data is not a big deal (especially with XGB). If we had categorical data, this would be different, espcially if we had to data cleanup, but other than outliers there is not much that can more be done that what is given.

In [23]:
```
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 [24]:
```
#chose two as during Eda it seemed that individually there was not much outliers an
print("old len: " + str(len(red_train)))
outliers = remove_outliers(red_train, 2 , red_train.columns[:-1])
print("new len: " + str(len(red_train.drop(outliers, axis = 0).reset_index(drop=Tru
```

```
old len: 1279
new len: 1213
```

overall outliers are neglible

Model Testing

In [25]:
```
X_train = red_train.drop("quality" , axis=1)
y_train = red_train["quality"]
X_test = red_test.drop("quality" , axis=1).copy()
y_test = red_test["quality"]
```

In [26]:
```
#xgb expects only what is present, not the actual scale of the data relative to qua
#therefore, we have to map to values 0-5 for the scale
y_train = y_train.map({3: 0, 4: 1, 5:2, 6:3, 7:4, 8:5})
y_test = y_test.map({3: 0, 4: 1, 5:2, 6:3, 7:4, 8:5})
```

Hyperopt

```
-- uses bayesian optimization to find the best parameter for
machine learning algorithm, by using probablistic search of the
hyperparamters supplied
   Compared to exahustive search it is much faster and its
performance is only a bit lower

How to implement:
    1. intitalise the domain space (same as a grid search)
    2. define the objective function  that we want to minimze
(error rate) of the model that we are testing (XGBoost decision
trees in this case)
    3. Optimize alogirhtm choice (the method used to construct the
surrogate objective function)
    4. Results, the score or the value pairs that the algorithm
uses to build the model
```

Below the hyperopt stuff that I personally was messing around wiht, but I do not think that I am going to include it in final report. Just skip...

In [27]:
```
# space = {"max_depth":hp.quniform("max_depth",3,18,1), #the max dept with integer
#          "gamma" : hp.uniform('gamma',1,9), #the gamma values with values ranging
#         'colsample_bytree' : hp.uniform('colsample_bytree', 0.5,1), #ratio of col
#         'min_child_weight' : hp.quniform('min_child_weight', 0, 10, 1), #tuning t
#         'n_estimators': 180,
#    'seed': 0
#  }


# def ojective(space:dict):
#    #creating a classifer with the opametrs pulled from the space that has be
#    #Most explanation is above
#    clf=XGBClassifier(
#        n_estimators =space['n_estimators'],
#        max_depth = int(space['max_depth']),
#        gamma = space['gamma'],
#        min_child_weight=int(space['min_child_weight']),
#        colsample_bytree=int(space['colsample_bytree']),
#        objective="multi:softprob", #type of objective function that is u
#        early_stopping_rounds=10, #sets the early stopping rounds if the
#        eval_metric="auc") #the measure to determine within the gradient

#    #evalutation train set and test set for doing a fit, efficivly measuring
#    evaluation = [( X_train, y_train), ( X_test, y_test)]

#    clf.fit(X_train,
#        y_train,
#        eval_set=evaluation, #passed in for the set
#        verbose=False)

#    # make a prediction
#    y_pred = clf.predict(X_test)

#    accuracy = f1_score(y_test,y_pred,average="weighted",zero_division=1)
#    print("SCORE:" + str(accuracy))
#    return {'loss' : -accuracy, "status" : STATUS_OK}

#trials = Trials()


# best_hyperparams = fmin(fn = ojective,
#            space = space,
#            algo = tpe.suggest,
#            max_evals = 50,
#            trials = trials)
```

Available optimizaiton algorithms: (oudated)

```
-- hp.choice(label,options) : returns a choice of one of the
options

-- hp.randint(label,upper) : returns a random integer better range
of 0 --> upper

-- hp.uniform(label,low, high) : returns a value uniformly between
the low and high

-- hp.uniform(label,low,high,q) : returns a value round to
(uniforn(low,high)/q) , and returns an integer
```

```
-- hp.normal(label,mean,std) : returns a real value that is
normally distributed with mean and standard deviation
```

Trials:

-- an object that contains or stores all the relevent information such as a hyperparameter. The loss functions for each type of parameter is stored here. Whenever doing iterations of training with current hyperparameters.

-- fmin is an optimization function that minimizes the loss function for each paramter inside of space.

-- algo: The type of the algorithm for finding best hyperparamter. Tpe is a type of decision tree, so effectively we are using a decision tree to do the hyperparamter choosing.

-- max evals: the amount of iterations that we choose to run through

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 [28]:

```
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(1,9,1),
         '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': [100,180, 250, 500, 750],
        }
```

In [29]:

```
#making the base model
#multi prob is a vector, containing all the classes
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)
```

Out[29]:

```
▸ RandomizedSearchCV
▸ estimator: XGBClassifier
    ▸ XGBClassifier
```

In [30]:

```
best_hyperparams = clf.best_params_
best_hyperparams
```

Out[30]:

```
{'subsample': 0.8999999999999999,
 'n_estimators': 750,
 'min_child_weight': 9,
 'max_depth': 15,
 'learning_rate': 0.1,
 'gamma': 2,
 'colsample_bytree': 0.8999999999999999,
 'colsample_bylevel': 0.6}
```

In [31]:

```
accuracy_f1 = []
recall = []
precision = []
accuracy = []
for i in range(0,5):
    #creating model with best hyperparameters
    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 [32]:

```
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: 0.5943817969672713
Average Recall: 0.62375
Average Precision: 0.653822962237242
Average Accuracy: 0.62375
```

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]:

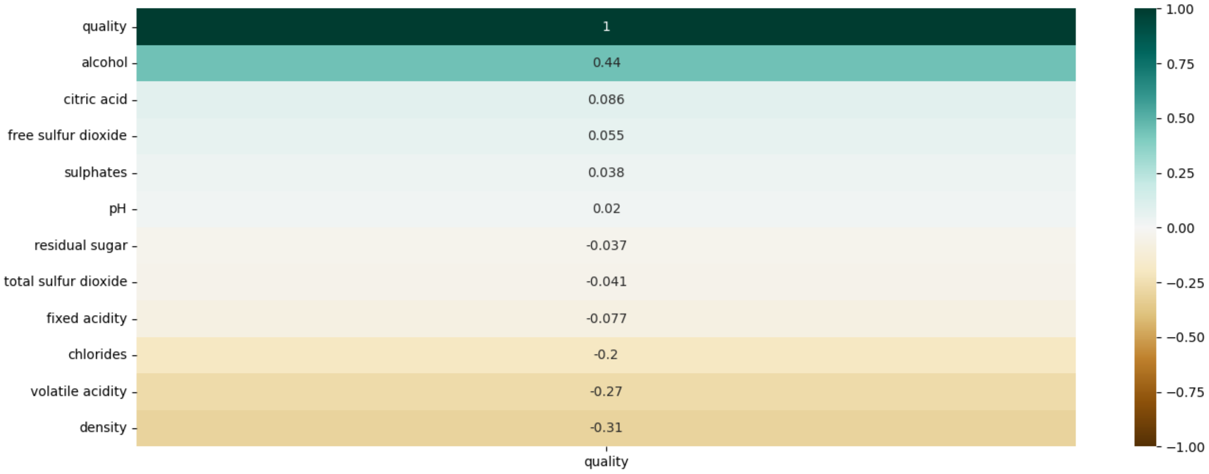
```
total_wine = pd.read_csv("data/winequality-total.csv",index_col=0)
```

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))
```

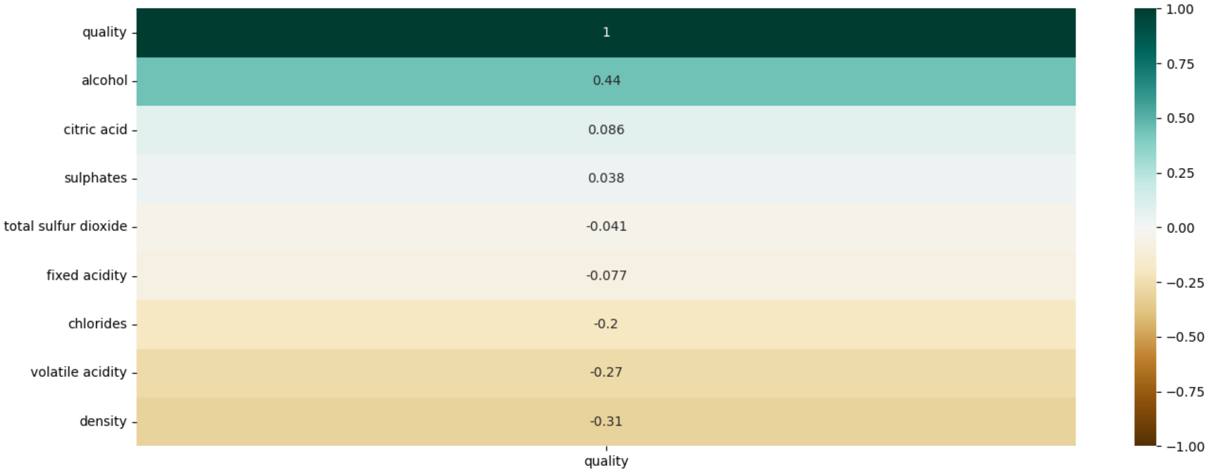
In [20]:
```
average_f1 = np.mean(accuracy_f1)
average_recall = np.mean(recall)
average_precision = np.mean(precision)
average_accuracy = np.mean(accuracy)
print("Average F1: " + str(average_f1))
print("Average Recall: " + str(average_recall))
print("Average Precision: " + str(average_precision))
print("Average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



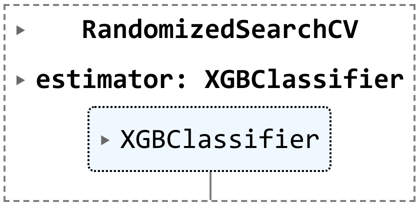
In [22]:
```
#dropping all values that are below 0.1 and between -0.1
total_wine_dropped = total_wine.drop(["citric acid", "sulphates", "total sulfur dio
```

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
```