Wine Quality Prediction

Introduction

Wine is a beverage made from fermented grape and other fruit juices with lower amount of alcohol content. Qulaity of wine is graded based on taste of wine and vintage. This process is time taking, costly and inefficient. A wine itself includes different parameters like fixed acidity, volatile acidity, citric acid, residual sugar, chlorides free sulfur dioxide, total sulfur dioxide, density, pH, sulphates, alcohol and quality.

Define the problem

In industry, understanding the demands of wine safety testing can be a complex task for the laboratory with numerous residues to monitor. But our application's prediction provide ideal solutions for the analysis of wine, which will make the whole process effecient and cheaper with less human interaction.

Objective

Our main objective is to predict the wine quality using machine learning Python programming language. A large dataset is considered and wine quality is modelled to analyse its quality through different parameters like fixed acidity, alcohol etc. All these parameters will be analysed through Machine learning algorithms which will help rate the wine on scale 1-10 or bad-good. It can support wine expert evaluations and ultimately improve the production.

Importing the necessary libraries

In [135…

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
import warnings
warnings.filterwarnings('ignore')

from statsmodels.stats.outliers_influence import variance_inflation_factor
from sklearn.model_selection import train_test_split , GridSearchCV, cross_val_scor
from sklearn.preprocessing import StandardScaler
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, f1_score, auc,confusion_matrix,classifi
from sklearn.metrics import roc_curve,plot_confusion_matrix
from sklearn.ensemble import RandomForestClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
```

In [2]:

```
#  Gather the data
```

```
df = pd.read_csv('QualityPrediction.csv')
```

In [3]:
```
df.head()
```

Out[3]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4

In [4]:
```
#  Our target variable is the quality and rest of the variables are predictor varib
# The target variable is multi-categorical in nature and falls under ordinal dataty

df.tail()
```

Out[4]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcoh
1594	6.2	0.600	0.08	2.0	0.090	32.0	44.0	0.99490	3.45	0.58	1(
1595	5.9	0.550	0.10	2.2	0.062	39.0	51.0	0.99512	3.52	0.76	1'
1596	6.3	0.510	0.13	2.3	0.076	29.0	40.0	0.99574	3.42	0.75	1'
1597	5.9	0.645	0.12	2.0	0.075	32.0	44.0	0.99547	3.57	0.71	1(
1598	6.0	0.310	0.47	3.6	0.067	18.0	42.0	0.99549	3.39	0.66	1'

In [5]:
```
df.shape
```

Out[5]:
```
(1599, 12)
```

In [124…
```
df.describe().T.style.background_gradient(cmap='Blues')
```

Out[124]:

	count	mean	std	min	25%	50%	75%	ma
fixed acidity	1599.000000	8.319637	1.741096	4.600000	7.100000	7.900000	9.200000	15.90000
volatile acidity	1599.000000	0.527821	0.179060	0.120000	0.390000	0.520000	0.640000	1.58000
citric acid	1599.000000	0.270976	0.194801	0.000000	0.090000	0.260000	0.420000	1.00000
residual sugar	1599.000000	2.538806	1.409928	0.900000	1.900000	2.200000	2.600000	15.50000
chlorides	1599.000000	0.087467	0.047065	0.012000	0.070000	0.079000	0.090000	0.61100
free sulfur dioxide	1599.000000	15.874922	10.460157	1.000000	7.000000	14.000000	21.000000	72.00000
total sulfur dioxide	1599.000000	46.467792	32.895324	6.000000	22.000000	38.000000	62.000000	289.00000
density	1599.000000	0.996747	0.001887	0.990070	0.995600	0.996750	0.997835	1.00369
pH	1599.000000	3.311113	0.154386	2.740000	3.210000	3.310000	3.400000	4.01000
sulphates	1599.000000	0.658149	0.169507	0.330000	0.550000	0.620000	0.730000	2.00000
alcohol	1599.000000	10.422983	1.065668	8.400000	9.500000	10.200000	11.100000	14.90000
quality	1599.000000	5.636023	0.807569	3.000000	5.000000	6.000000	6.000000	8.00000

In [7]:
```
df.info()
# There are no null values in the dataset.
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1599 entries, 0 to 1598
Data columns (total 12 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   fixed acidity         1599 non-null   float64
 1   volatile acidity      1599 non-null   float64
 2   citric acid           1599 non-null   float64
 3   residual sugar        1599 non-null   float64
 4   chlorides             1599 non-null   float64
 5   free sulfur dioxide   1599 non-null   float64
 6   total sulfur dioxide  1599 non-null   float64
 7   density               1599 non-null   float64
 8   pH                    1599 non-null   float64
 9   sulphates             1599 non-null   float64
 10  alcohol               1599 non-null   float64
 11  quality               1599 non-null   int64  
dtypes: float64(11), int64(1)
memory usage: 150.0 KB
```

In [8]:
```
df.isnull().sum()
```

Out[8]:
```
fixed acidity           0
volatile acidity        0
citric acid             0
residual sugar          0
chlorides               0
free sulfur dioxide     0
total sulfur dioxide    0
density                 0
pH                      0
sulphates               0
alcohol                 0
quality                 0
dtype: int64
```

In [9]:
```
df['quality'].nunique()
```

Out[9]:
```
6
```

In [10]:
```
# Let's check the distribution of quality in the dataset

df['quality'].value_counts()*100/len(df['quality'])
```

Out[10]:
```
5    42.589118
6    39.899937
7    12.445278
4     3.314572
8     1.125704
3     0.625391
Name: quality, dtype: float64
```

In [11]:
```
# Majority of the wines have quality is in the range of 5,6 and 7.
df.quality.value_counts()
```

Out[11]:
```
5    681
6    638
7    199
4     53
8     18
3     10
Name: quality, dtype: int64
```

In [12]:
```
df.dtypes
# There are 11 continuous features and 1 categorical variable which is our Target v
```

Out[12]:
```
fixed acidity           float64
volatile acidity        float64
citric acid             float64
residual sugar          float64
chlorides               float64
free sulfur dioxide     float64
total sulfur dioxide    float64
density                 float64
pH                      float64
sulphates               float64
alcohol                 float64
quality                   int64
dtype: object
```

Checking duplicate records:

In [13]:
```
df.duplicated().sum()

# There are about 240 duplicate rows in the dataframe.
```

```
# In this case we will have to investigate and discuss further with the client if t
# needs to be discarded.
```

Out[13]:
```
240
```

In [14]:
```
# Outlier detection using z-score-

def z_outliers(column_name,data):
    outlier=[]
    mean=np.mean(data)
    std = np.std(data)
    for i,value in enumerate(data):
        z = (value-mean)/std
        if z>3 or z<-3:
            outlier.append(value)
    print('No of outliers in feature {}: {}'.format(column_name, len(outlier)))
    print('Outlier values : {}'.format(outlier))
```

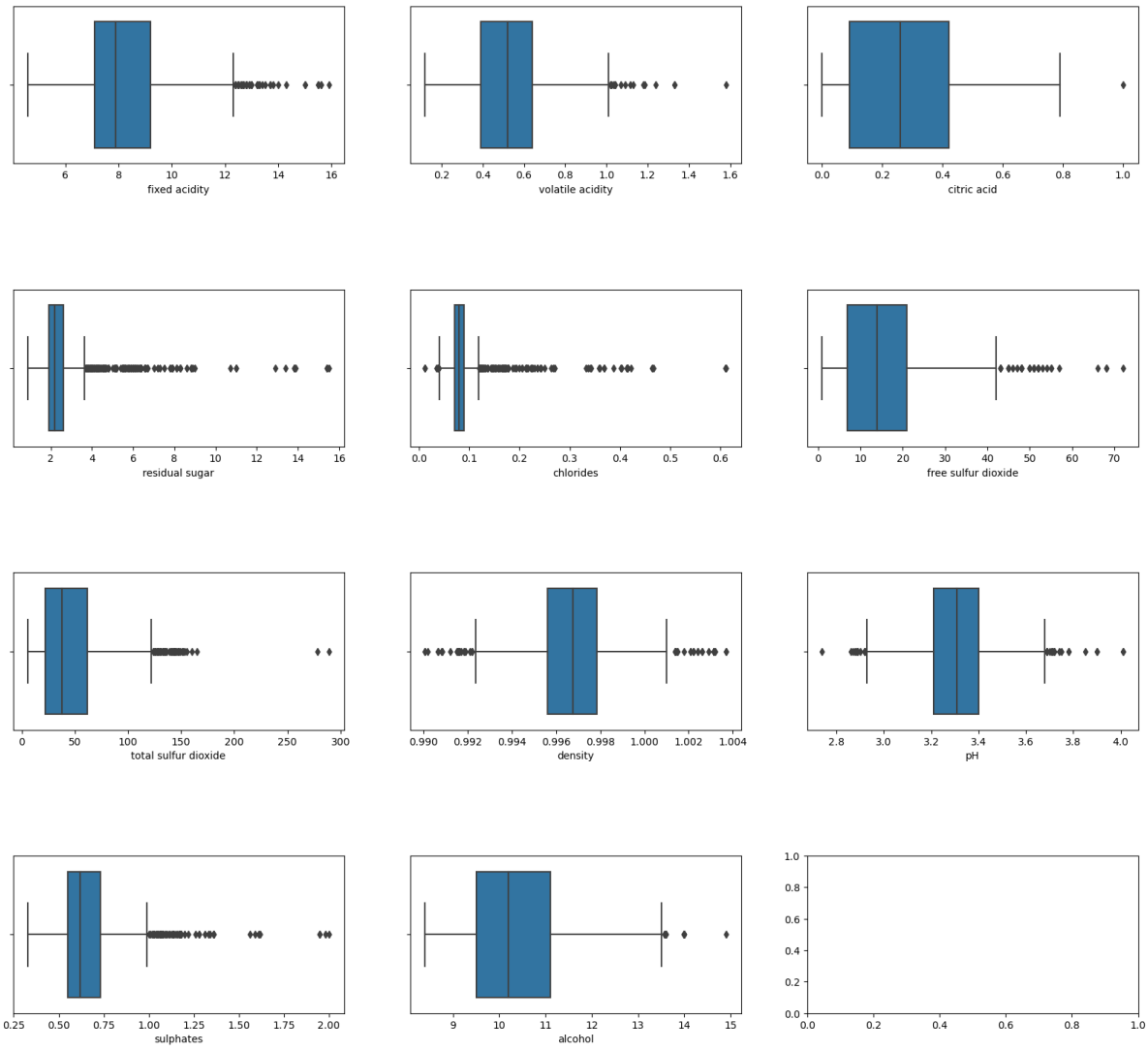
In [15]:
```
for i in df.columns:
    z_outliers(i, df[i])
    print('\n')
```

```
No of outliers in feature fixed acidity: 12
Outlier values : [15.0, 15.0, 13.8, 14.0, 13.7, 13.7, 15.6, 14.3, 15.5, 15.5, 15.
6, 15.9]


No of outliers in feature volatile acidity: 10
Outlier values : [1.13, 1.07, 1.33, 1.33, 1.09, 1.24, 1.185, 1.115, 1.58, 1.18]


No of outliers in feature citric acid: 1
Outlier values : [1.0]


No of outliers in feature residual sugar: 30
Outlier values : [10.7, 7.3, 7.2, 7.0, 11.0, 11.0, 7.9, 7.9, 15.5, 8.3, 7.9, 8.6,
7.5, 9.0, 8.8, 8.8, 8.9, 8.1, 8.1, 8.3, 8.3, 7.8, 12.9, 13.4, 15.4, 15.4, 13.8, 1
3.8, 13.9, 7.8]


No of outliers in feature chlorides: 31
Outlier values : [0.368, 0.341, 0.332, 0.464, 0.401, 0.467, 0.236, 0.61, 0.36, 0.2
7, 0.337, 0.263, 0.611, 0.358, 0.343, 0.413, 0.25, 0.422, 0.387, 0.415, 0.243, 0.2
41, 0.414, 0.369, 0.403, 0.414, 0.415, 0.415, 0.267, 0.235, 0.23]


No of outliers in feature free sulfur dioxide: 22
Outlier values : [52.0, 51.0, 50.0, 68.0, 68.0, 54.0, 53.0, 52.0, 51.0, 57.0, 50.
0, 48.0, 48.0, 72.0, 51.0, 51.0, 52.0, 55.0, 55.0, 48.0, 48.0, 66.0]


No of outliers in feature total sulfur dioxide: 15
Outlier values : [148.0, 153.0, 165.0, 151.0, 149.0, 147.0, 148.0, 155.0, 151.0, 1
52.0, 278.0, 289.0, 160.0, 147.0, 147.0]


No of outliers in feature density: 18
Outlier values : [1.0032, 1.0026, 1.00315, 1.00315, 1.00315, 1.0026, 0.99064, 0.99
064, 1.00289, 0.99007, 0.99007, 0.9902, 0.9908, 0.99084, 1.00369, 1.00369, 1.0024
2, 1.00242]


No of outliers in feature pH: 8
Outlier values : [3.9, 3.85, 2.74, 3.9, 3.78, 3.78, 4.01, 4.01]


No of outliers in feature sulphates: 27
Outlier values : [1.56, 1.28, 1.2, 1.28, 1.95, 1.22, 1.95, 1.98, 1.31, 2.0, 1.59,
1.61, 1.26, 1.36, 1.18, 1.36, 1.36, 1.17, 1.62, 1.18, 1.34, 1.17, 1.17, 1.33, 1.1
8, 1.17, 1.17]


No of outliers in feature alcohol: 8
Outlier values : [14.0, 14.0, 14.0, 14.0, 14.9, 14.0, 14.0, 14.0]


No of outliers in feature quality: 10
Outlier values : [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
```

In [16]:

```
# Boxplot visualization to study outliers in each variable:

fig, axes = plt.subplots(nrows=4,ncols=3,figsize=(20,18))
fig.subplots_adjust(hspace=0.8)
```

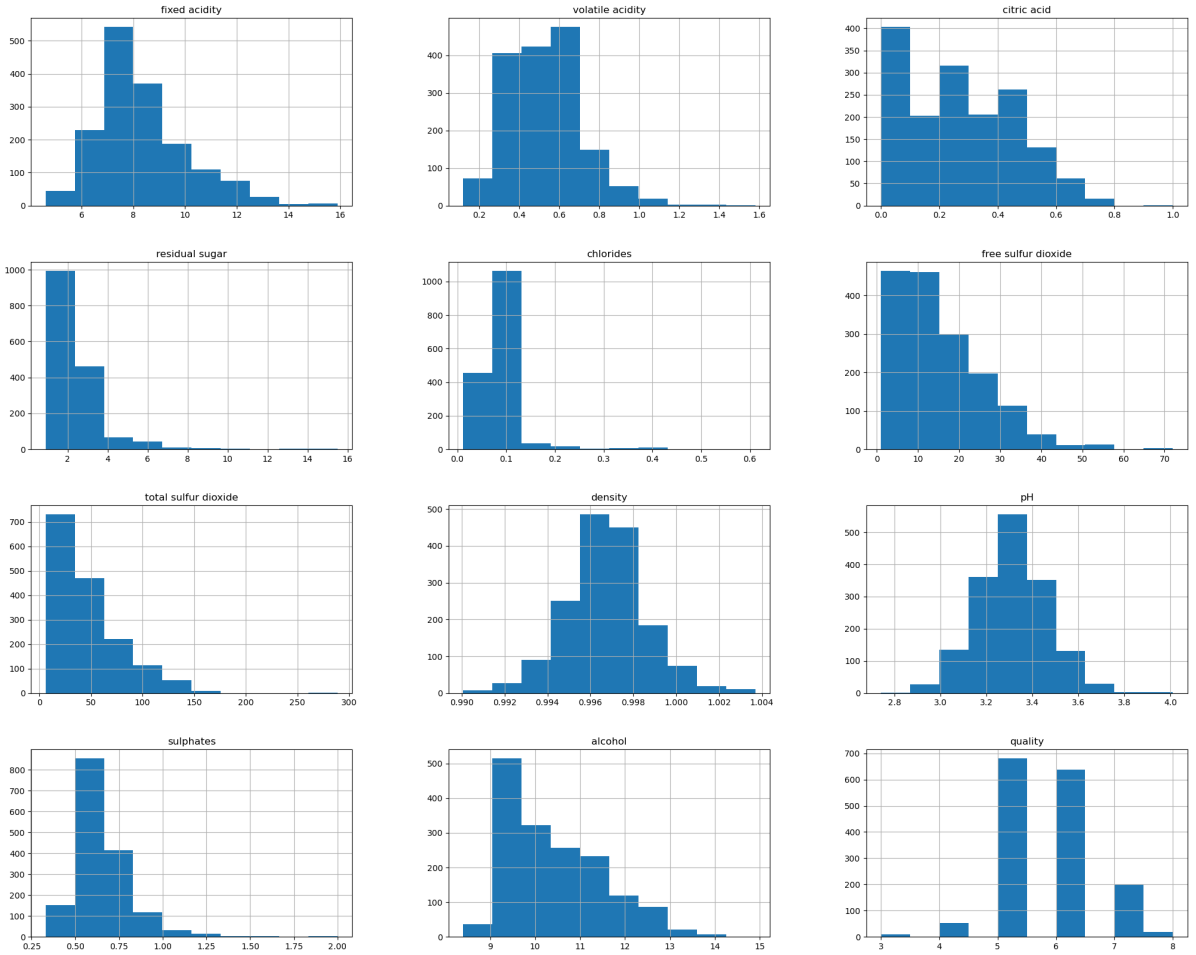
```
sns.boxplot(df['fixed acidity'], ax=axes[0,0])
sns.boxplot(df['volatile acidity'], ax=axes[0,1])
sns.boxplot(df['citric acid'], ax=axes[0,2])
sns.boxplot(df['residual sugar'], ax=axes[1,0])
sns.boxplot(df['chlorides'], ax=axes[1,1])
sns.boxplot(df['free sulfur dioxide'], ax=axes[1,2])
sns.boxplot(df['total sulfur dioxide'], ax=axes[2,0])
sns.boxplot(df['density'], ax=axes[2,1])
sns.boxplot(df['pH'], ax=axes[2,2])
sns.boxplot(df['sulphates'], ax=axes[3,0])
sns.boxplot(df['alcohol'], ax=axes[3,1])
plt.show()
```



Data Visualization

In [23]:
```
#  Let's try to understand the data better with the help of Data Visualization.
```

In [24]:
```
# Distribution of each variable

df.hist(figsize=(25,20))
plt.show()
```

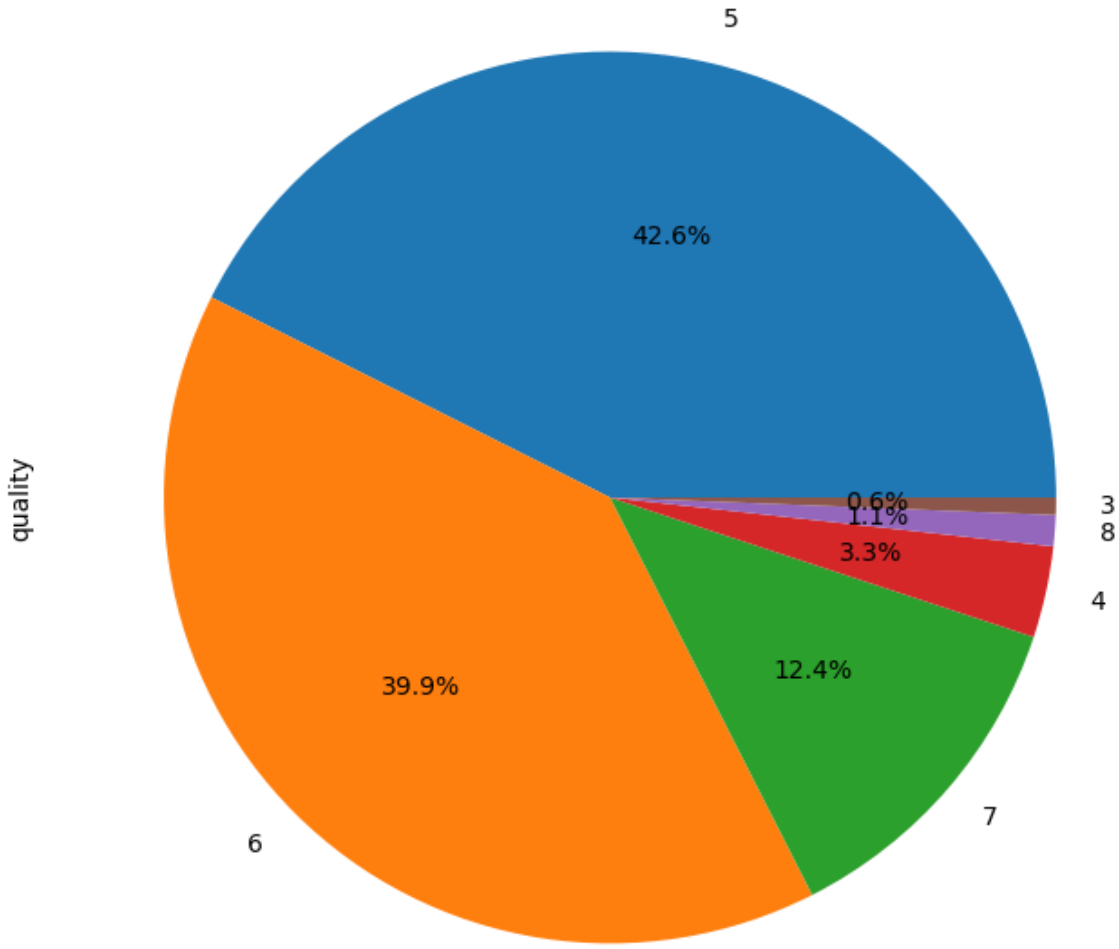


In [127…

```
# Distribtion of Quality

plt.figure(figsize=(10,8))
df['quality'].value_counts().plot(kind='pie', autopct='%0.1f%%')
```

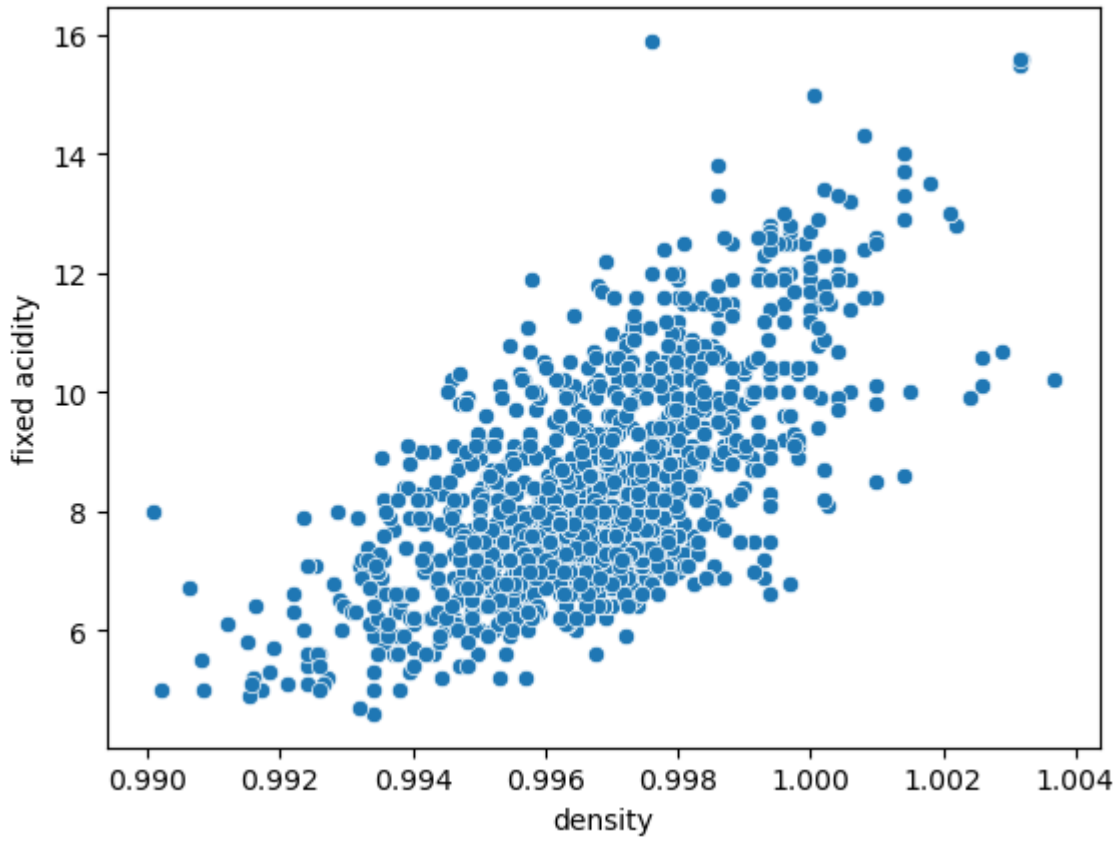
Out[127]:

```
<AxesSubplot:ylabel='quality'>
```



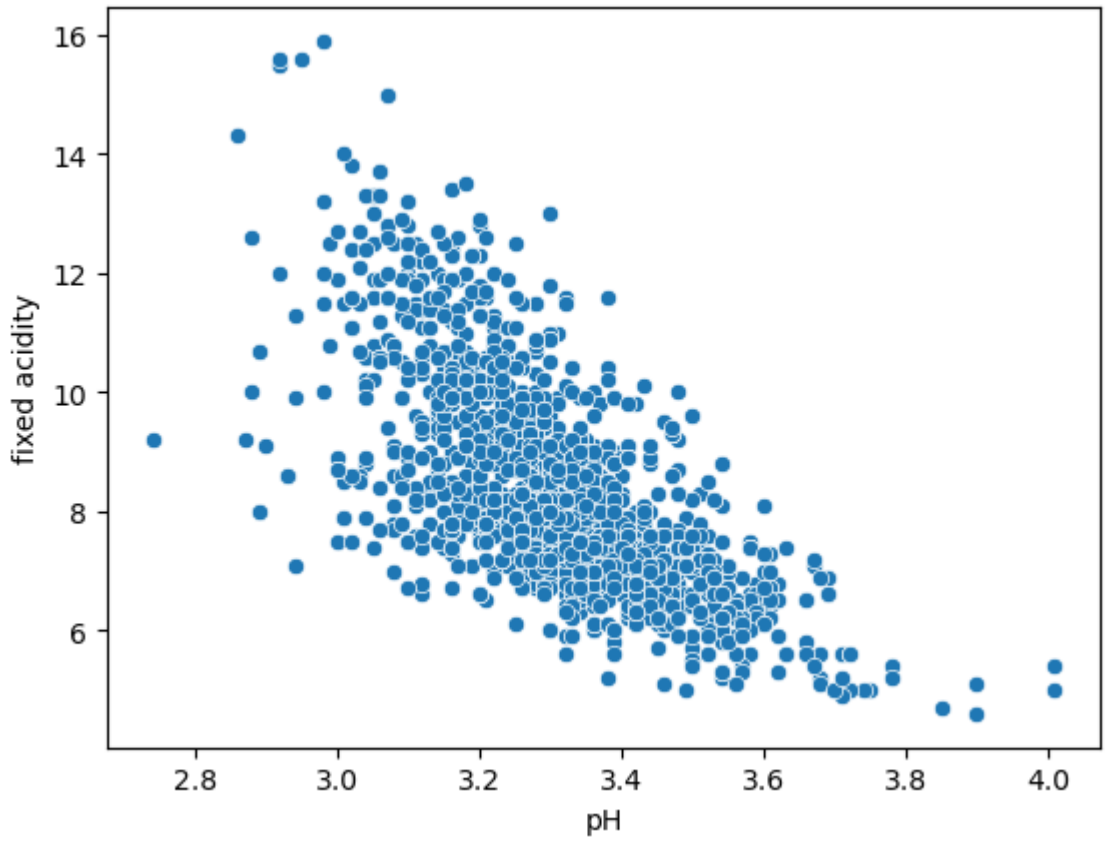
In [131…
```
sns.scatterplot(data=df, x=df['density'], y=df['fixed acidity'])

# As the density increases the fixed acidity also increases
```

Out[131]:
```
<AxesSubplot:xlabel='density', ylabel='fixed acidity'>
```



In [132…
```
sns.scatterplot(x=df['pH'],y=df['fixed acidity'])

# As th pH increases the fixed acidity descreases
```

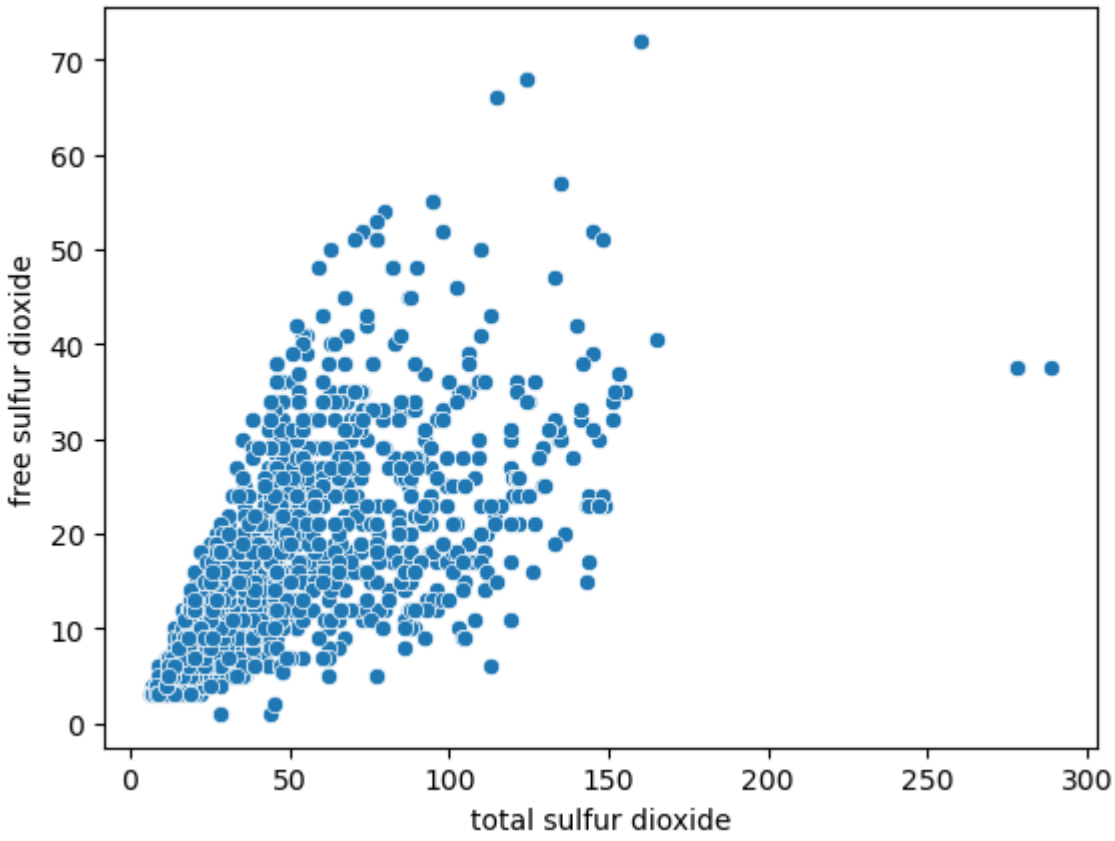
Out[132]:
```
<AxesSubplot:xlabel='pH', ylabel='fixed acidity'>
```



In [133…
```
sns.scatterplot(x=df['total sulfur dioxide'], y=df['free sulfur dioxide'])
```

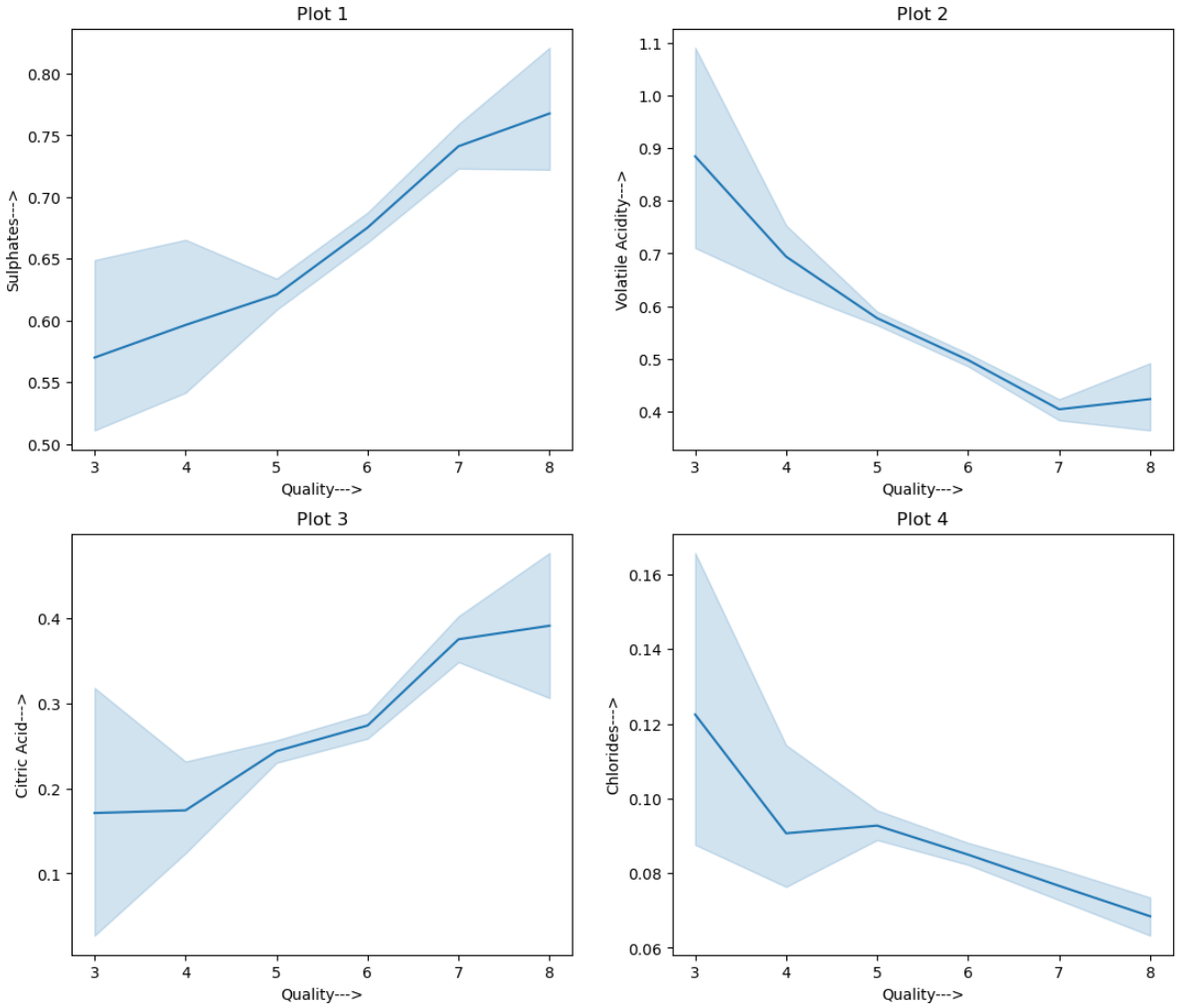
```
# As the total sulphr dioxide content increases the free sulphur dioxide content al
```

Out[133]:
```
<AxesSubplot:xlabel='total sulfur dioxide', ylabel='free sulfur dioxide'>
```



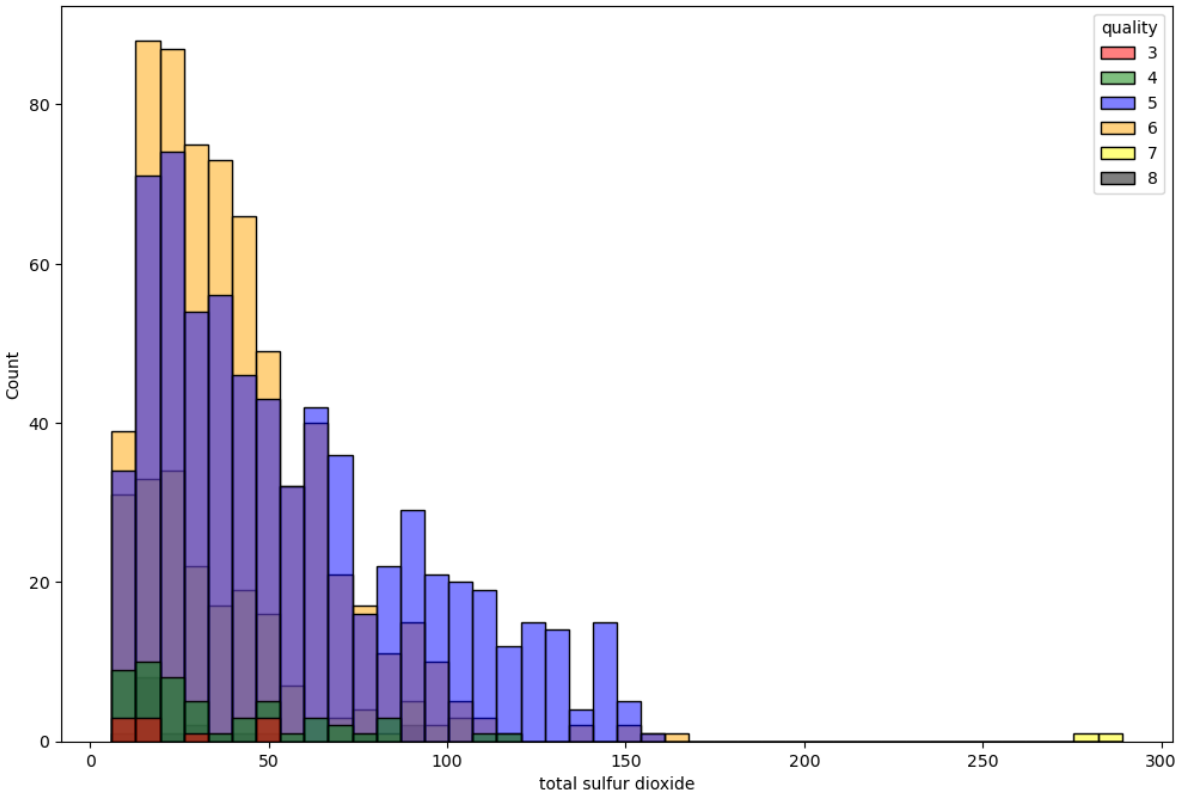
In [25]:
```
#  From the below lineplots we see how our Target variable 'quality' changes with r

fig , axes = plt.subplots(2,2, figsize = (13,11))
sns.lineplot(df.quality, df.sulphates, ax = axes[0,0])
axes[0,0].set_xlabel('Quality--->')
axes[0,0].set_ylabel('Sulphates--->')
axes[0,0].set_title('Plot 1')

sns.lineplot(df['quality'], df['volatile acidity'] , ax = axes[0,1])
axes[0,1].set_xlabel('Quality--->')
axes[0,1].set_ylabel('Volatile Acidity--->')
axes[0,1].set_title('Plot 2')

sns.lineplot(df['quality'], df['citric acid'], ax = axes[1,0])
axes[1,0].set_xlabel('Quality--->')
axes[1,0].set_ylabel('Citric Acid--->')
axes[1,0].set_title('Plot 3')

sns.lineplot(df.quality, df.chlorides, ax = axes[1,1])
axes[1,1].set_xlabel('Quality--->')
axes[1,1].set_ylabel('Chlorides--->')
axes[1,1].set_title('Plot 4')
```

Out[25]:
```
Text(0.5, 1.0, 'Plot 4')
```



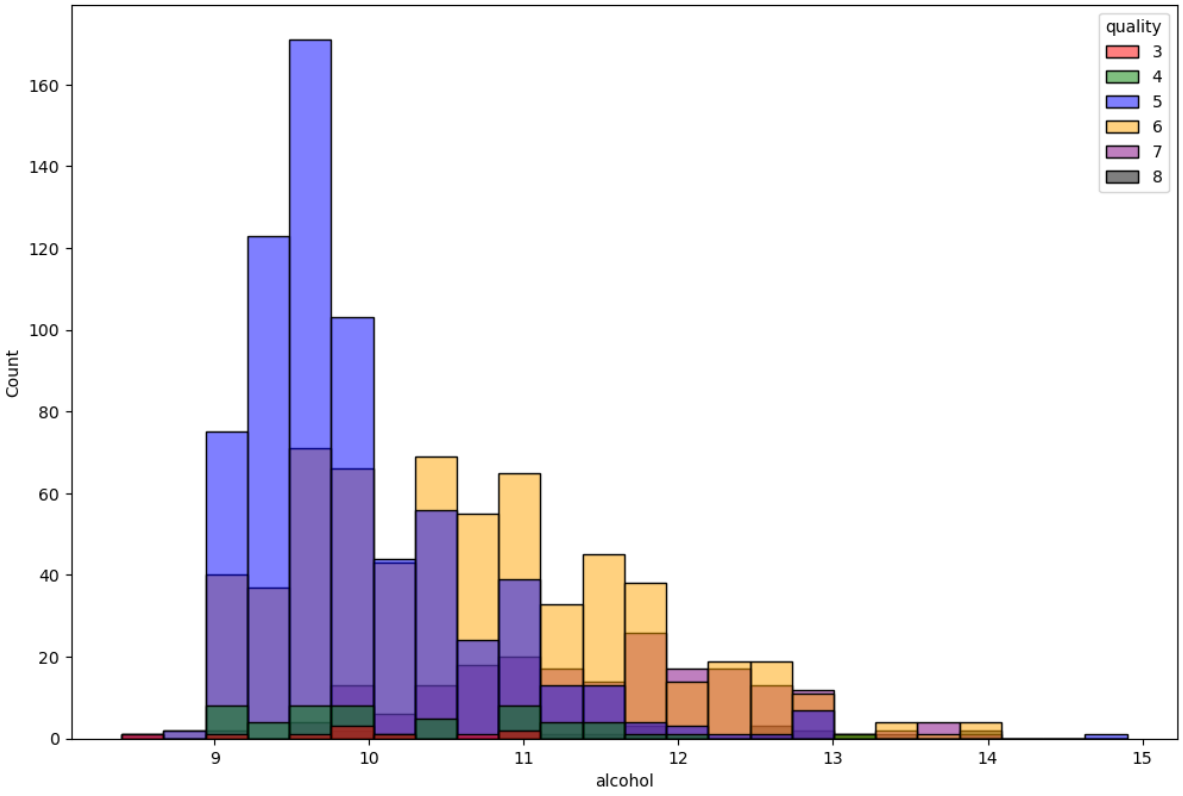
In [26]:
```
plt.figure(figsize=(12,8))
sns.histplot(data=df , x = 'total sulfur dioxide', hue = 'quality', palette=['red',
                                                                              'yello

# From this plot we see there are only 2 wines of quality 7 that have total sulfur
#  the normal range of total sulfur dioxide of ~ (0 to 170)
```

Out[26]:
```
<AxesSubplot:xlabel='total sulfur dioxide', ylabel='Count'>
```



In [27]:
```
plt.figure(figsize=(12,8))
sns.histplot(data=df , x = 'alcohol', hue = 'quality', palette=['red', 'green', 'bl
```

Out[27]:
```
<AxesSubplot:xlabel='alcohol', ylabel='Count'>
```



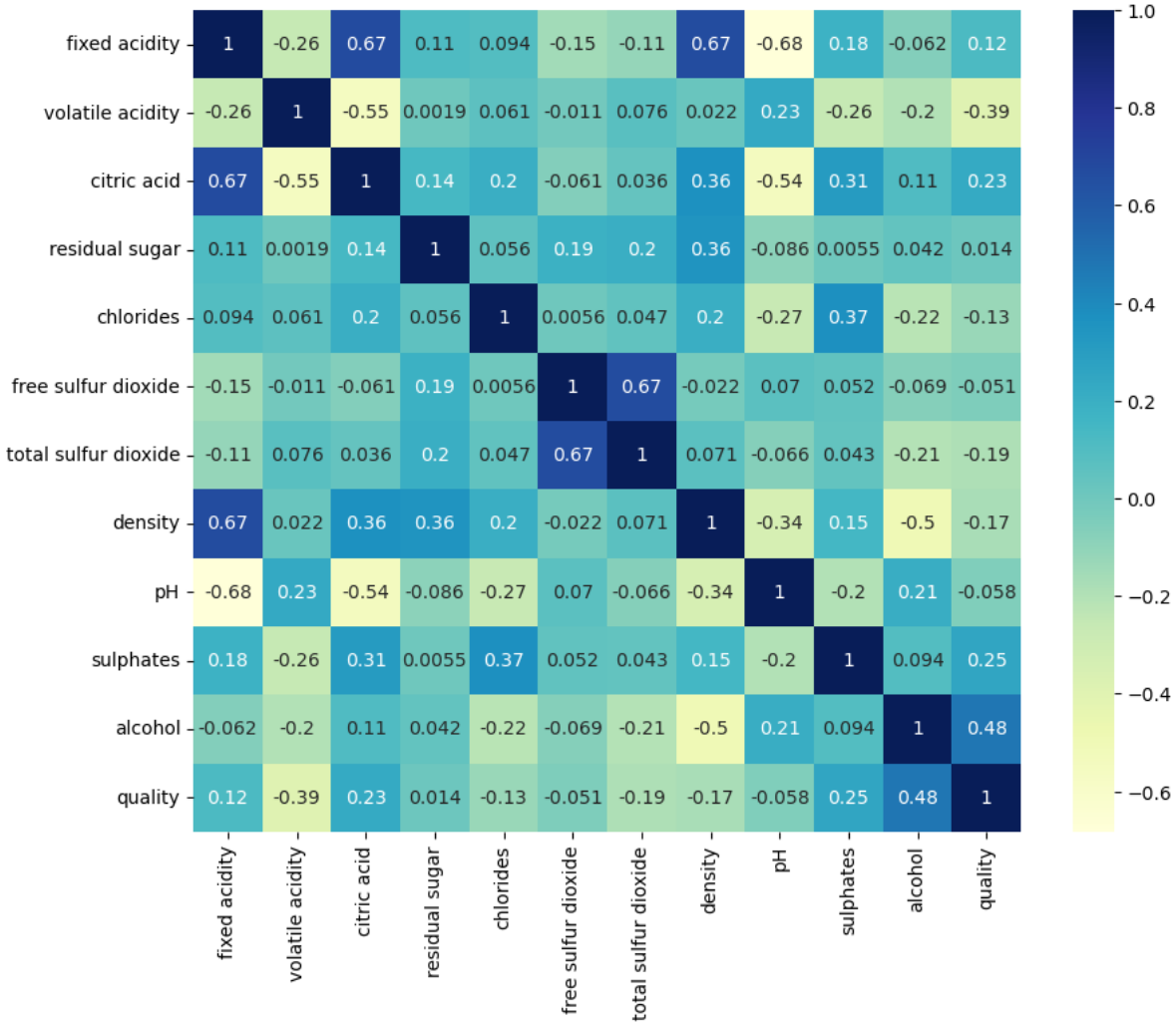
In [28]:
```
# To check how the independent variables are related to each other we will calculat
```

In [29]:
```
df.corr()
```

Out[29]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	
fixed acidity	1.000000	-0.256131	0.671703	0.114777	0.093705	-0.153794	-0.113181	0.668047	-(
volatile acidity	-0.256131	1.000000	-0.552496	0.001918	0.061298	-0.010504	0.076470	0.022026	(
citric acid	0.671703	-0.552496	1.000000	0.143577	0.203823	-0.060978	0.035533	0.364947	-(
residual sugar	0.114777	0.001918	0.143577	1.000000	0.055610	0.187049	0.203028	0.355283	-(
chlorides	0.093705	0.061298	0.203823	0.055610	1.000000	0.005562	0.047400	0.200632	-(
free sulfur dioxide	-0.153794	-0.010504	-0.060978	0.187049	0.005562	1.000000	0.667666	-0.021946	(
total sulfur dioxide	-0.113181	0.076470	0.035533	0.203028	0.047400	0.667666	1.000000	0.071269	-(
density	0.668047	0.022026	0.364947	0.355283	0.200632	-0.021946	0.071269	1.000000	-(
pH	-0.682978	0.234937	-0.541904	-0.085652	-0.265026	0.070377	-0.066495	-0.341699	1
sulphates	0.183006	-0.260987	0.312770	0.005527	0.371260	0.051658	0.042947	0.148506	-(
alcohol	-0.061668	-0.202288	0.109903	0.042075	-0.221141	-0.069408	-0.205654	-0.496180	(
quality	0.124052	-0.390558	0.226373	0.013732	-0.128907	-0.050656	-0.185100	-0.174919	-(

In [30]:
```
plt.figure(figsize=(10,8))
sns.heatmap(df.corr(), cmap = 'YlGnBu', annot = True)
plt.show()
```



In [31]:
```
# Let's check VIF (Vaiance Inflation Fcator) for all variables:

vif_data = pd.DataFrame()
```

VIF is Variance Inflation Factor which is used to quantify the multicollinearity between the predictor variables. When there is high correlation between two predictor variables, it becomes difficult to determine the individual effect of the predictor variables on the target variable.

In [32]:
```
vif_data['features'] = df.drop('quality',axis=1).columns
```

In [33]:
```
vif_data['features']
```

Out[33]:
```
0           fixed acidity
1        volatile acidity
2             citric acid
3          residual sugar
4               chlorides
5     free sulfur dioxide
6    total sulfur dioxide
7                 density
8                      pH
9               sulphates
10                alcohol
Name: features, dtype: object
```

In [34]:
```
vif_data['VIF'] = [variance_inflation_factor(df.drop('quality', axis=1),i) for i in
```

In [35]:
```
vif_data
```

Out[35]:

	features	VIF
0	fixed acidity	74.452265
1	volatile acidity	17.060026
2	citric acid	9.183495
3	residual sugar	4.662992
4	chlorides	6.554877
5	free sulfur dioxide	6.442682
6	total sulfur dioxide	6.519699
7	density	1479.287209
8	pH	1070.967685
9	sulphates	21.590621
10	alcohol	124.394866

In [36]:
```
# Let's also check the Multicollinearity with OLS method:
```

Predictor variables are highly corelated to each other. If the VIF is greater than 5 it means there is Multicollinearity between the predictor variables. Hence we will not leverage Linear Regression and Logistic Regression models as the independent variables are having high correlation with each other.

In [37]:
```
import statsmodels.api as sm
```

In [38]:
```
X = df[['fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar',
       'chlorides', 'free sulfur dioxide', 'total sulfur dioxide', 'density',
       'pH', 'sulphates', 'alcohol']]
Y =df['quality']
```

In [39]:
```
X = sm.add_constant(X)
```

In [40]:
```
model = sm.OLS(Y,X).fit()
```

In [41]:
```
model.summary()
```

Out[41]:

OLS Regression Results

Dep. Variable:	quality	**R-squared:**	0.361
Model:	OLS	**Adj. R-squared:**	0.356
Method:	Least Squares	**F-statistic:**	81.35
Date:	Tue, 23 Jan 2024	**Prob (F-statistic):**	1.79e-145
Time:	18:56:37	**Log-Likelihood:**	-1569.1
No. Observations:	1599	**AIC:**	3162.
Df Residuals:	1587	**BIC:**	3227.
Df Model:	11		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	21.9652	21.195	1.036	0.300	-19.607	63.538
fixed acidity	0.0250	0.026	0.963	0.336	-0.026	0.076
volatile acidity	-1.0836	0.121	-8.948	0.000	-1.321	-0.846
citric acid	-0.1826	0.147	-1.240	0.215	-0.471	0.106
residual sugar	0.0163	0.015	1.089	0.276	-0.013	0.046
chlorides	-1.8742	0.419	-4.470	0.000	-2.697	-1.052
free sulfur dioxide	0.0044	0.002	2.009	0.045	0.000	0.009
total sulfur dioxide	-0.0033	0.001	-4.480	0.000	-0.005	-0.002
density	-17.8812	21.633	-0.827	0.409	-60.314	24.551
pH	-0.4137	0.192	-2.159	0.031	-0.789	-0.038
sulphates	0.9163	0.114	8.014	0.000	0.692	1.141
alcohol	0.2762	0.026	10.429	0.000	0.224	0.328

Omnibus:	27.376	**Durbin-Watson:**	1.757
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	40.965
Skew:	-0.168	**Prob(JB):**	1.27e-09
Kurtosis:	3.708	**Cond. No.**	1.13e+05

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 1.13e+05. This might indicate that there are strong multicollinearity or other numerical problems.

In [42]:
```
# From OLS method, we can conclude that there is no need to drop any variables on b
```

In [152…
```
#  Assigning the target variable as 'y' and predictor variables as 'x'.
x = df.drop('quality', axis = 1).values
```

In [153…
```
y = df.quality.values.reshape(-1,1)
```

In [154…
```
x.shape
```

Out[154]:
```
(1599, 11)
```

In [155…
```
y.shape
```

Out[155]:
```
(1599, 1)
```

Splitting the data into train and test datasets

In [44]:
```
xtrain , xtest, ytrain, ytest =  train_test_split(x,y,test_size=0.2,random_state=5)
```

Scaling is always performed after splitting the data into train and test in order to avoid the test data getting exposed to the model during Model building stage which will cause Data leakage. We will perform scaling in distance based algorithms.

In [46]:
```
xtrain
```

Out[46]:
```
array([[7.70e+00, 7.15e-01, 1.00e-02, ..., 3.41e+00, 5.70e-01, 1.18e+01],
       [1.12e+01, 2.80e-01, 5.60e-01, ..., 3.16e+00, 5.80e-01, 9.80e+00],
       [4.60e+00, 5.20e-01, 1.50e-01, ..., 3.90e+00, 5.60e-01, 1.31e+01],
       ...,
       [8.90e+00, 8.40e-01, 3.40e-01, ..., 3.12e+00, 4.80e-01, 9.10e+00],
       [1.28e+01, 3.00e-01, 7.40e-01, ..., 3.20e+00, 7.70e-01, 1.08e+01],
       [6.90e+00, 5.10e-01, 2.30e-01, ..., 3.40e+00, 8.40e-01, 1.12e+01]])
```

In [47]:
```
xtrain.shape
```

Out[47]:
```
(1279, 11)
```

In [48]:
```
xtest
```

Out[48]:
```
array([[ 7.2  ,  0.63 ,  0.   , ...,  3.37 ,  0.58 ,  9.   ],
       [11.6  ,  0.47 ,  0.44 , ...,  3.38 ,  0.86 ,  9.9  ],
       [ 7.7  ,  0.96 ,  0.2  , ...,  3.36 ,  0.44 , 10.9  ],
       ...,
       [ 7.2  ,  0.5  ,  0.18 , ...,  3.52 ,  0.72 ,  9.6  ],
       [ 7.7  ,  0.75 ,  0.27 , ...,  3.24 ,  0.45 ,  9.3  ],
       [ 7.7  ,  0.705,  0.1  , ...,  3.39 ,  0.49 ,  9.7  ]])
```

In [49]:
```
xtest.shape
```

Out[49]:
```
(320, 11)
```

In [50]:
```
# Now that we have performed EDA on the data to understand it better, let us explor
# train and test the dataset to understand how each model performs:
```

Model Building

In [134…
```
Models = {}
```

Decision Tree

In [51]:
```
# We will build decision tree using both the criterion i.e. GINI and Entropy.
```

In [52]:
```
#  Decision Tree with depth 4 (GINI):
```

In [53]:
```
model_dt_4 = DecisionTreeClassifier(random_state=4, max_depth =4)
```

In [54]:
```
# Model creation:

model_dt_4.fit(xtrain,ytrain)
```

Out[54]:
```
DecisionTreeClassifier(max_depth=4, random_state=4)
```

In [55]:
```
y_pred_4 = model_dt_4.predict(xtest) # Testing the model on unseen data i.e. Test d
accuracy_score_4 = accuracy_score(ytest,y_pred_4)
print('Accuracy Score for model with depth 4 is: ',accuracy_score_4)
```

```
Accuracy Score for model with depth 4 is:  0.609375
```

In [56]:
```
#  Decision Tree with depth 6 (GINI):
```

In [57]:
```
model_dt_6 = DecisionTreeClassifier(random_state=6,max_depth=6)
```

In [58]:
```
model_dt_6.fit(xtrain,ytrain)
```

Out[58]:
```
DecisionTreeClassifier(max_depth=6, random_state=6)
```

In [59]:
```
y_pred_6 = model_dt_6.predict(xtest)
accuracy_score_6 = accuracy_score(ytest,y_pred_6)
print('Accuracy Score for model with depth 6 is: ',accuracy_score_6)
```

```
Accuracy Score for model with depth 6 is:  0.615625
```

In [60]:
```
#  Decision Tree with depth 8 (GINI):
```

In [61]:
```
model_dt_8 = DecisionTreeClassifier(random_state=8,max_depth=8)
```

In [62]:
```
model_dt_8.fit(xtrain,ytrain)
```

Out[62]:
```
DecisionTreeClassifier(max_depth=8, random_state=8)
```

In [63]:
```
y_pred_8 = model_dt_8.predict(xtest)
accuracy_score_8 = accuracy_score(ytest,y_pred_8)
print('Accuracy Score for model with depth 8 is: ',accuracy_score_8)
```

```
Accuracy Score for model with depth 8 is:  0.628125
```

In [64]:
```
# Decision Tree using Entropy
```

In [65]:
```
model_dt_ent = DecisionTreeClassifier(random_state=8,max_depth=8, criterion='entrop
```

In [66]:
```
model_dt_ent.fit(xtrain,ytrain)
```

Out[66]:
```
DecisionTreeClassifier(criterion='entropy', max_depth=8, random_state=8)
```

In [67]:
```
y_pred_ent = model_dt_ent.predict(xtest)
accuracy_score_ent = accuracy_score(ytest,y_pred_ent)
print('Accuracy Score for model with depth 8 using Entropy is: ',accuracy_score_ent
```

```
Accuracy Score for model with depth 8 using Entropy is:  0.6
```

In [68]:
```
classificationReport_dt = classification_report(ytest,y_pred_8)
```

In [69]:
```
print(classificationReport_dt)
```

```
              precision    recall  f1-score   support

           3       0.00      0.00      0.00         1
           4       0.00      0.00      0.00         6
           5       0.79      0.71      0.75       152
           6       0.53      0.68      0.60       115
           7       0.48      0.38      0.42        40
           8       0.00      0.00      0.00         6

    accuracy                           0.63       320
   macro avg       0.30      0.29      0.29       320
weighted avg       0.63      0.63      0.62       320
```

In [70]:
```
# Hyperparameter Tuning
```

In [71]:
```
from sklearn.model_selection import GridSearchCV
```

In [72]:
```
parameters = {'criterion' : ['gini', 'entropy'], 'max_depth': [4,6,10]}
```

In [73]:
```
dt_grid = DecisionTreeClassifier()
```

In [74]:
```
grid_search= GridSearchCV(estimator=dt_grid, param_grid=parameters, cv=10, scoring=
```

In [75]:
```
grid_search.fit(xtrain,ytrain)
```

Out[75]:
```
GridSearchCV(cv=10, estimator=DecisionTreeClassifier(),
             param_grid={'criterion': ['gini', 'entropy'],
                         'max_depth': [4, 6, 10]},
             scoring='accuracy')
```

In [76]:
```
grid_search.best_params_
```

Out[76]:
```
{'criterion': 'gini', 'max_depth': 10}
```

Evaluating Test data

In [77]:
```
y_grid_dt = grid_search.predict(xtest)
```

In [78]:
```
accuracy_score(ytest,y_grid_dt)
```

Out[78]:
```
0.65625
```

Evaluating Train data

In [79]:
```
y_pred_dt_train = grid_search.predict(xtrain)
```

In [80]:
```
accuracy_score(ytrain,y_pred_dt_train)
```

Out[80]:
```
0.8803752931978108
```

In [137…
```
Models['Decision Tree '] = [accuracy_score(ytest,y_grid_dt), f1_score(ytest,y_grid_
```

Decision Tree after hyperparameter tuning is overfitting as the train accuracy is around 91% and Test accuracy is around 63%.

Random Forest

In [81]:
```
model_rf = RandomForestClassifier()
```

In [82]:
```
plain_model = model_rf.fit(xtrain,ytrain)
```

In [83]:
```
y_plain_rf_test = plain_model.predict(xtest)
```

In [84]:
```
accuracy_score(ytest,y_plain_rf_test)
```

Out[84]:
```
0.74375
```

In [85]:
```
y_plain_rf_train = plain_model.predict(xtrain)
```

In [86]:
```
accuracy_score(ytrain, y_plain_rf_train)
```

Out[86]:
```
1.0
```

In [87]:
```
#  We will hypertune the parameters with the help of GridSearchCV which will provid
#  parameters that may improve the effeciency of the model.
```

In [88]:
```
param_dist = {'max_depth' : [2,4,6,8], 'criterion' : ['gini', 'entropy'], 'bootstra
              'max_features' : ['auto', 'sqrt', 'log2', None]}
```

In [123…
```
cv_rf = GridSearchCV(model_rf, cv =10, param_grid = param_dist, verbose = 1 ) # Run
#  possible PnCs of these parameters
```

In [90]:
```
cv_rf.fit(xtrain,ytrain)
```

Out[90]:
```
Fitting 10 folds for each of 64 candidates, totalling 640 fits
GridSearchCV(cv=10, estimator=RandomForestClassifier(), n_jobs=3,
             param_grid={'bootstrap': [True, False],
                         'criterion': ['gini', 'entropy'],
                         'max_depth': [2, 4, 6, 8],
                         'max_features': ['auto', 'sqrt', 'log2', None]},
             verbose=1)
```

In [91]:
```
print('Best parameters using GridSearchCV are: \n', cv_rf.best_params_)
```

```
Best parameters using GridSearchCV are:
 {'bootstrap': False, 'criterion': 'entropy', 'max_depth': 8, 'max_features': 'sqr
t'}
```

In [92]:
```
model_rf.set_params(criterion = 'entropy', max_depth = 8, max_features = 'auto', bo
```

Out[92]:
```
RandomForestClassifier(bootstrap=False, criterion='entropy', max_depth=8)
```

In [93]:
```
model_rf.fit(xtrain,ytrain)
y_pred_rf = model_rf.predict(xtest)
```

Evaluating Test data

In [94]:
```
accuracy_score(ytest,y_pred_rf)
```

Out[94]:
```
0.721875
```

In [95]:
```
classificationReport_rf =  classification_report(ytest, y_pred_rf)
print(classificationReport_rf)
```

```
              precision    recall  f1-score   support

           3       0.00      0.00      0.00         1
           4       0.00      0.00      0.00         6
           5       0.81      0.84      0.83       152
           6       0.62      0.75      0.68       115
           7       0.77      0.42      0.55        40
           8       0.00      0.00      0.00         6

    accuracy                           0.72       320
   macro avg       0.37      0.34      0.34       320
weighted avg       0.70      0.72      0.70       320
```

Evaluating Train data

In [96]:
```
y_pred_rf_train = model_rf.predict(xtrain)
```

In [97]:
```
accuracy_score(ytrain,y_pred_rf_train)
```

Out[97]:
```
0.9265050820953871
```

In [139…
```
Models['Random Forest']= [accuracy_score(ytest,y_pred_rf), f1_score(ytest,y_pred_rf
```

The model is overfitting as the Training score is around 91% and Test score is around 73% regardless of hyperparameter Tuning

Gaussian Naive Bayes

In [98]:
```
model_gnb = GaussianNB()

# Since the predictor variables are co-related to each other due to presence of Mul
# this algorithm that predictor variables are not related to each other may impact
```

In [99]:
```
model_gnb.fit(xtrain,ytrain)
```

Out[99]:
```
GaussianNB()
```

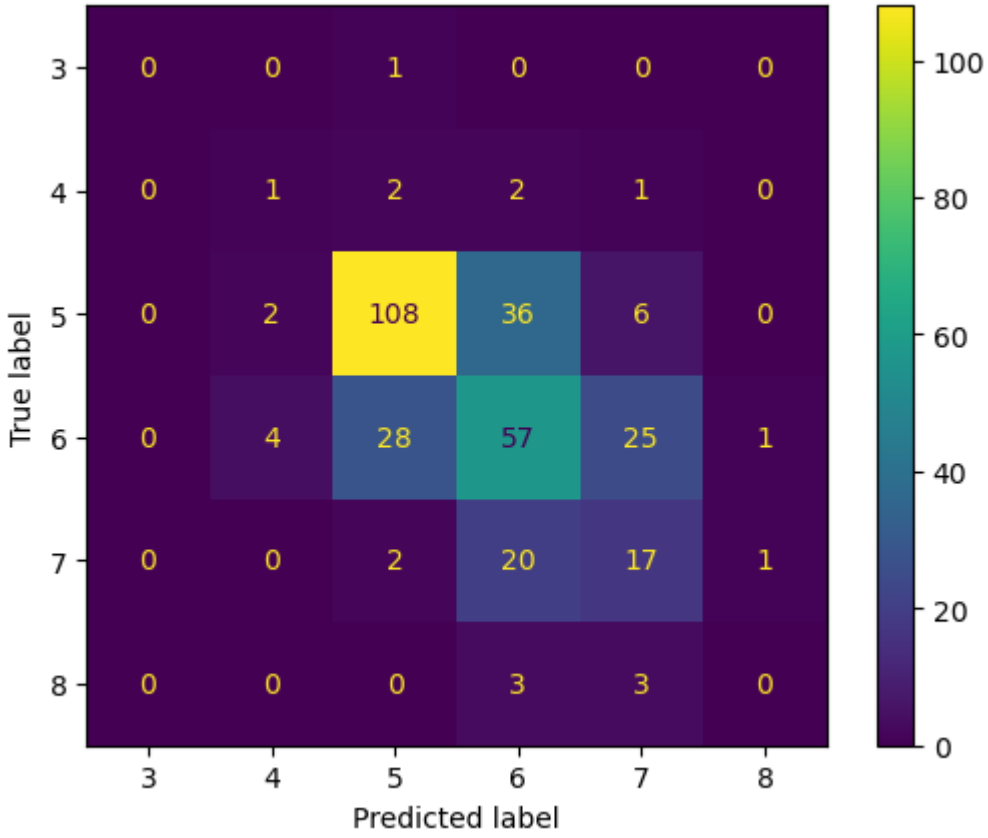
In [100…
```
y_pred_gnb = model_gnb.predict(xtest)
```

In [101…
```
# Accuracy obtained from this model is below average.

accuracy_score_gnb = accuracy_score(ytest,y_pred_gnb)
print(accuracy_score_gnb)
```

```
0.571875
```

In [102…
```
plot_confusion_matrix(model_gnb,xtest,ytest)
```

Out[102]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x1c7f6b89a60>
```



In [103…
```
classification_report_gnb = classification_report(ytest,y_pred_gnb)
print(classification_report_gnb)

# From the below classification report we see that accuracy is not very good. It is
# which contradicts the assumption of this algorithm.
```

```
              precision    recall  f1-score   support

           3       0.00      0.00      0.00         1
           4       0.14      0.17      0.15         6
           5       0.77      0.71      0.74       152
           6       0.48      0.50      0.49       115
           7       0.33      0.42      0.37        40
           8       0.00      0.00      0.00         6

    accuracy                           0.57       320
   macro avg       0.29      0.30      0.29       320
weighted avg       0.58      0.57      0.58       320
```

In [141…
```
Models['Gaussian Naive Bayes'] = [accuracy_score(ytest,y_pred_gnb), f1_score(ytest,
```

K Nearest Neighbours with Cross Validation

In [104…
```
# KNN is a distance based model hence we will first scale our features before train

ss = StandardScaler()
```

In [105…
```
xtrain_ss = ss.fit_transform(xtrain)
xtest_ss = ss.transform(xtest)
```

In [106…
```
from sklearn.neighbors import KNeighborsClassifier
```

In [107…
```
# Taking the K value as 3 at first and observing how the model performs.

knn3 = KNeighborsClassifier(n_neighbors = 3)
```

In [108…
```
knn3.fit(xtrain_ss,ytrain)
```

Out[108]:
```
KNeighborsClassifier(n_neighbors=3)
```

In [109…
```
y_pred_knn3 = knn3.predict(xtest_ss)
```

In [110…
```
print(classification_report(ytest,y_pred_knn3))
```

```
              precision    recall  f1-score   support

           3       0.00      0.00      0.00         1
           4       0.00      0.00      0.00         6
           5       0.74      0.63      0.68       152
           6       0.52      0.65      0.58       115
           7       0.48      0.40      0.44        40
           8       0.00      0.00      0.00         6

    accuracy                           0.58       320
   macro avg       0.29      0.28      0.28       320
weighted avg       0.60      0.58      0.59       320
```

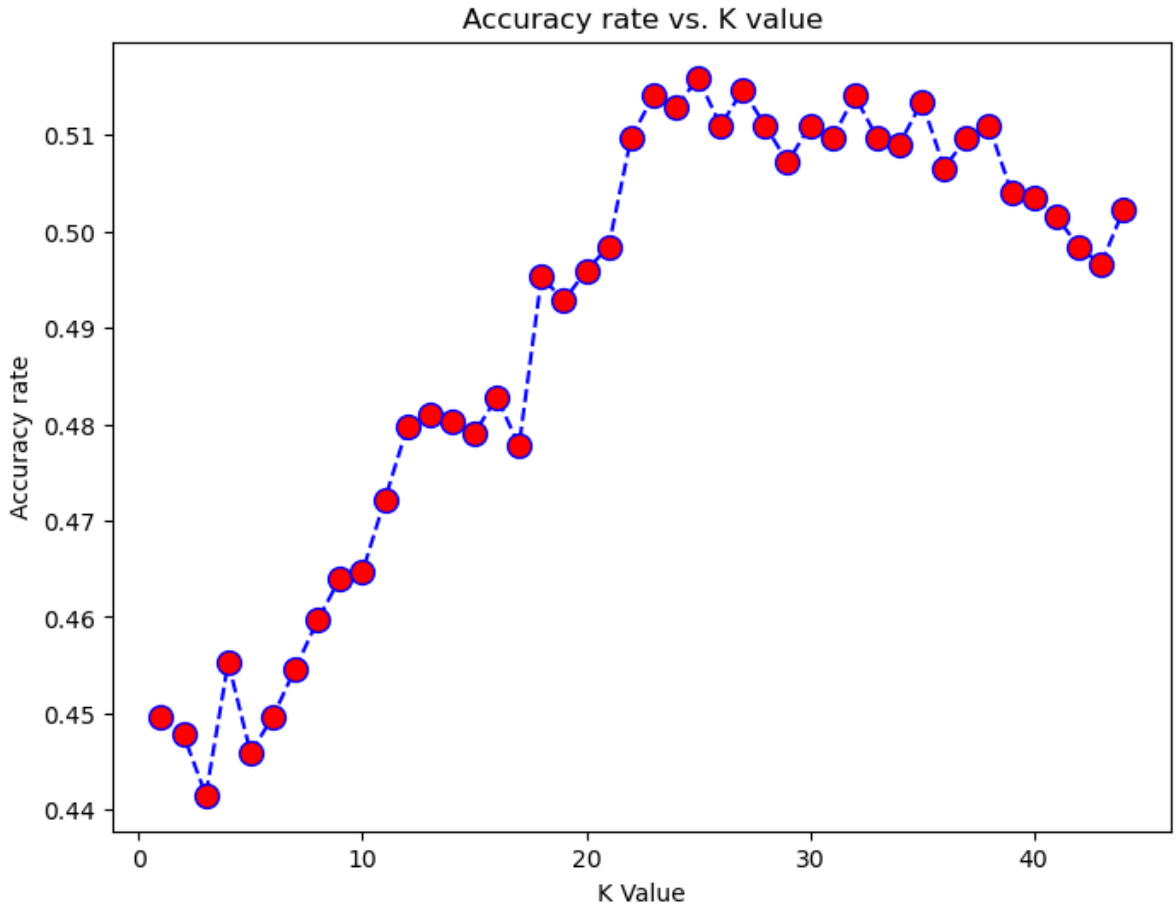
In [111…
```
#  Choosing K value:-
#  We will apply loop for values of K from 1 to 45 and decide which values of K wil
#  And also we need to make sure our model is stable which can be checked by plotti
#  and the respective K values.
```

In [112…
```
accuracy_rate = []
for i in range(1,45):
    knn = KNeighborsClassifier(n_neighbors = i)
    score= cross_val_score(knn,x,y,cv=10)
    accuracy_rate.append(score.mean())
```

In [113…
```
error_rate = []
for i in range(1,45):
    knn = KNeighborsClassifier(n_neighbors = i)
    score = cross_val_score(knn,x,y,cv=10)
    error_rate.append(1-score.mean())
```

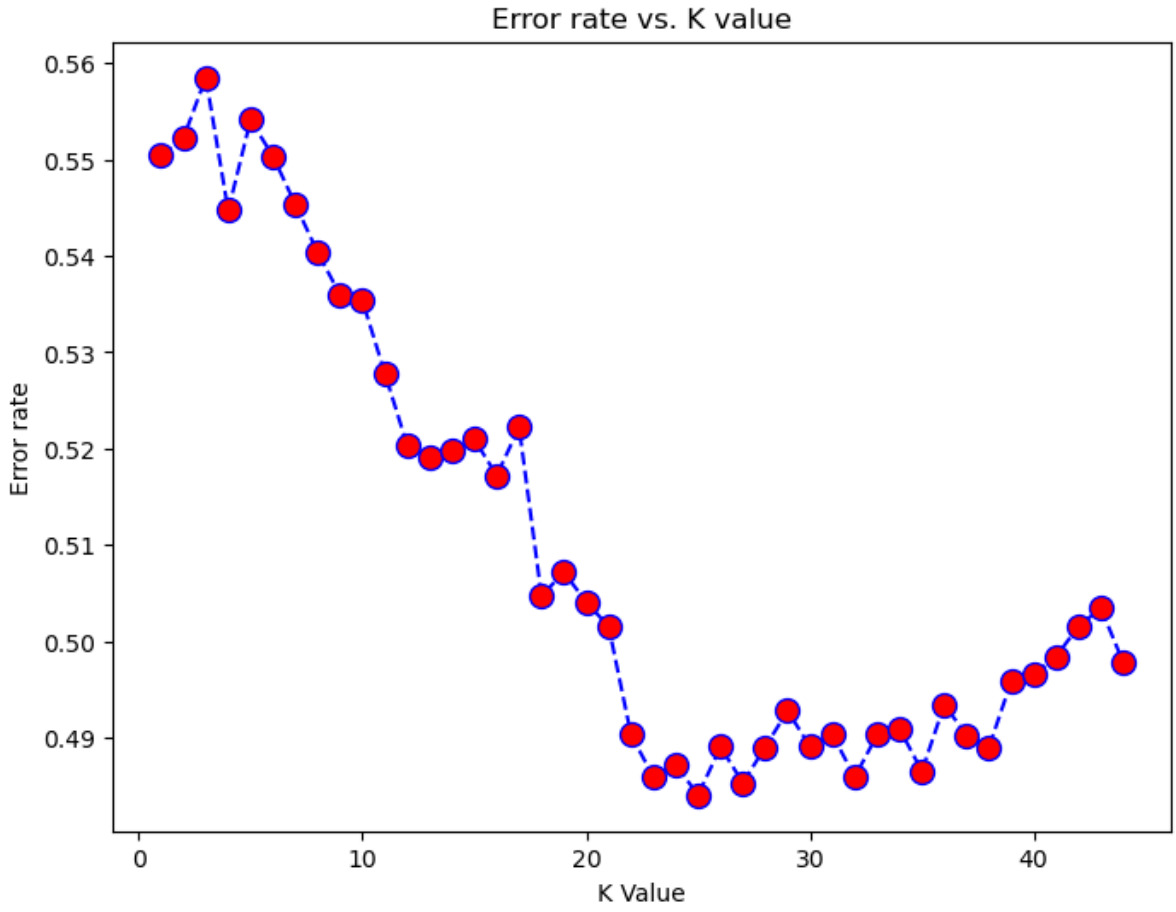
In [114…
```
plt.figure(figsize=(8,6))
plt.plot(range(1,45), accuracy_rate, color = 'blue', linestyle = 'dashed', marker='
plt.title('Accuracy rate vs. K value')
plt.xlabel('K Value')
plt.ylabel('Accuracy rate')
```

Out[114]:
```
Text(0, 0.5, 'Accuracy rate')
```



In [115…
```
plt.figure(figsize=(8,6))
plt.plot(range(1,45), error_rate , color = 'blue', linestyle = 'dashed', marker='o'
plt.title('Error rate vs. K value')
plt.xlabel('K Value')
plt.ylabel('Error rate')
```

Out[115]:
```
Text(0, 0.5, 'Error rate')
```



In [116…
```
#  At around K values of 12 to 15 we see some stability however beyond these points
#  accuracy rate. Lets see how the model performs at K = 14.
knn14 = KNeighborsClassifier(n_neighbors = 14 )
```

In [117…
```
knn14.fit(xtrain_ss,ytrain)
```

Out[117]:
```
KNeighborsClassifier(n_neighbors=14)
```

In [118…
```
y_pred_knn14 = knn14.predict(xtest_ss)
```

In [119…
```
print(classification_report(ytest,y_pred_knn14))
```

```
              precision    recall  f1-score   support

           3       0.00      0.00      0.00         1
           4       0.00      0.00      0.00         6
           5       0.71      0.72      0.71       152
           6       0.50      0.57      0.53       115
           7       0.47      0.38      0.42        40
           8       0.00      0.00      0.00         6

    accuracy                           0.59       320
   macro avg       0.28      0.28      0.28       320
weighted avg       0.57      0.59      0.58       320
```

In [120…
```
#  The graph is condensed for K values beyond 23. At K=23 the accuracy rate improve
#  due to the fact that the accuracy rate beyond this point is not much fluctuating

knn23 = KNeighborsClassifier(n_neighbors = 23 )
knn23.fit(xtrain_ss,ytrain)
y_pred_knn23 = knn23.predict(xtest_ss)
print(classification_report(ytest,y_pred_knn23))
```

	precision	recall	f1-score	support
3	0.00	0.00	0.00	1
4	0.00	0.00	0.00	6
5	0.73	0.74	0.73	152
6	0.52	0.63	0.57	115
7	0.50	0.35	0.41	40
8	0.00	0.00	0.00	6
accuracy			0.62	320
macro avg	0.29	0.29	0.29	320
weighted avg	0.60	0.62	0.60	320

In [121…
```
#  At K=24 again we see the accuracy rate rose to 63.

knn24 = KNeighborsClassifier(n_neighbors = 24 )
knn24.fit(xtrain_ss,ytrain)
y_pred_knn24 = knn24.predict(xtest_ss)
print(classification_report(ytest,y_pred_knn24))
```

	precision	recall	f1-score	support
3	0.00	0.00	0.00	1
4	0.00	0.00	0.00	6
5	0.73	0.76	0.74	152
6	0.53	0.63	0.58	115
7	0.54	0.38	0.44	40
8	0.00	0.00	0.00	6
accuracy			0.63	320
macro avg	0.30	0.29	0.29	320
weighted avg	0.61	0.63	0.62	320

In [142…
```
# Evaluating Train data

y_pred_knn24_train = knn24.predict(xtrain_ss)
accuracy_score(ytrain, y_pred_knn24_train)
```

Out[142]:
```
0.6239249413604379
```

In [143…
```
Models['KNN'] = [accuracy_score(ytest,y_pred_knn24), f1_score(ytest,y_pred_knn24, a
```

For KNN model, at k=24 the accuracy score rose to around 62% and the performance of the model for both train and test data is around 62%. We can conclude that this model is much stable than other models even though the model is average in terms of performance. However, it is important to note that this model is neither overfitting nor underfitting which brings more stability.

Conclusion

In [151…
```
Models_comparison = pd.DataFrame.from_dict(Models).T
Models_comparison.columns = ['Accuracy score', 'F1-score']
Models_comparison.style.background_gradient(cmap='Blues')
```

Out[151]:

	Accuracy score	F1-score
Decision Tree	0.656250	0.650751
Random Forest	0.721875	0.704163
Gaussian Naive Bayes	0.571875	0.575082
KNN	0.631250	0.615707

Insight on Overfitting and Underfitting :

The models are exhibiting different levels of bias and variances. Random Forest and Decision Tree models are overfitting. Naive Baye's model is underfitting. K-Nearest Neigbour model is neither Overfitting nor underfitting. The KNN model is stable as compared to other models despite the fact that the performance of the model is average.

In [ ]: