Data Mining Group Project on Wine Dataset

By, Shanmukh Behara (ug8788), Snigdha Enumulapally (xc2373), Damoder Reddy Pannala (LZ7285), Sri Samhitha Pallelamudi (ti6266),

Objective

1) Predicting Quality of wine 2) Classification based on Wine Type

Importing Libraries

In [4]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from sklearn import preprocessing
from sklearn.model_selection import train_test_split
import seaborn as sns
from sklearn.decomposition import PCA

%matplotlib inline

from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from sklearn.impute import KNNImputer
from sklearn.preprocessing import StandardScaler,MinMaxScaler,RobustScaler
from pandas.plotting import scatter_matrix
from sklearn import neighbors
from sklearn.base import BaseEstimator, TransformerMixin, clone, ClassifierMixin
from sklearn.metrics import mean_squared_error   #RMSE
from sklearn.metrics import mean_absolute_error
```

Reading Data

In [5]:
```
df=pd.read_csv('wine.csv')
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6497 entries, 0 to 6496
Data columns (total 13 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   fixed acidity         6497 non-null   float64
 1   volatile acidity      6497 non-null   float64
 2   citric acid           6497 non-null   float64
 3   residual sugar        6497 non-null   float64
 4   chlorides             6497 non-null   float64
 5   free sulfur dioxide   6491 non-null   float64
 6   total sulfur dioxide  6497 non-null   float64
 7   density               6494 non-null   float64
 8   pH                    6497 non-null   float64
 9   sulphates             6482 non-null   float64
 10  alcohol               6487 non-null   float64
 11  Type                  6497 non-null   object 
 12  quality               6497 non-null   int64  
dtypes: float64(11), int64(1), object(1)
memory usage: 660.0+ KB
```

Data Exploration

In [6]:
```
#Data attributes
df.columns
```

Out[6]:
```
Index(['fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar',
       'chlorides', 'free sulfur dioxide', 'total sulfur dioxide', 'density',
       'pH', 'sulphates', 'alcohol', 'Type', 'quality'],
      dtype='object')
```

In [7]:
```
#Replacing space with '_' in column names
df.columns = [s.strip().replace(' ', '_') for s in df.columns]
df.columns
```

Out[7]:
```
Index(['fixed_acidity', 'volatile_acidity', 'citric_acid', 'residual_sugar',
       'chlorides', 'free_sulfur_dioxide', 'total_sulfur_dioxide', 'density',
       'pH', 'sulphates', 'alcohol', 'Type', 'quality'],
      dtype='object')
```

In [8]:
```
# Printing the number of rows and columns
print("Rows, columns: " + str(df.shape))

# First five rows of the dataset
df.head()
```

```
Rows, columns: (6497, 13)
```

Out[8]:

	fixed_acidity	volatile_acidity	citric_acid	residual_sugar	chlorides	free_sulfur_dioxide	total_sulf
0	7.39926	0.964904	0.000000	2.200110	0.087996	16.000800	
1	12.29877	0.389961	0.630032	2.300115	0.090995	6.000300	
2	6.39936	0.269973	0.490025	7.300365	0.045998	53.002650	
3	7.39926	0.189981	0.300015	12.800640	0.052997	48.502425	
4	6.19938	0.339966	0.300015	11.100555	0.046998	28.001400	

In [9]:
```
#Describing the data
df.describe()

#There is no column with SD '0'. So no need to drop a column.
```

Out[9]:

	fixed_acidity	volatile_acidity	citric_acid	residual_sugar	chlorides	free_sulfur_dioxide
count	6497.000000	6497.000000	6497.000000	6497.000000	6497.000000	6491.000000
mean	7.214586	0.339632	0.318649	5.443508	0.056031	30.535805
std	1.296304	0.164620	0.145325	4.758042	0.035032	17.756039
min	3.799620	0.079992	0.000000	0.600030	0.009000	1.000050
25%	6.399360	0.229977	0.250013	1.800090	0.037998	17.000850
50%	6.999300	0.289971	0.310016	3.000150	0.046998	29.001450
75%	7.699230	0.399960	0.390020	8.100405	0.064997	41.002050
max	15.898410	1.579842	1.660083	65.803290	0.610969	289.014450

In [10]:
```
#Describing dataset for both types of wine.
pd.set_option('display.max_columns', 500)
df.groupby("Type").describe()
```

Out[10]:

	fixed_acidity								volatile_acidity	
	count	mean	std	min	25%	50%	75%	max	count	mean
Type										
Red	4898.0	6.854102	0.843784	3.79962	6.29937	6.79932	7.29927	14.19858	4898.0	0.278213
White	1599.0	8.318805	1.740922	4.59954	7.09929	7.89921	9.19908	15.89841	1599.0	0.527768

Inference:

- We find that mean quality score of Red wine is 5.87 and that of White wine 5.63
- There are some significant differences between Red Wine and White wine as mentioned below: 1) White wine has higher average fixed acidity, volatile acidity, chlorides, pH and sulphates than Red wine.

In [11]:
```
#Removing Duplicates
#winedf is the dataframe after removing duplicates

winedf = df.drop_duplicates(keep='first').copy()
winedf.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 5324 entries, 0 to 6495
Data columns (total 13 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   fixed_acidity         5324 non-null   float64
 1   volatile_acidity      5324 non-null   float64
 2   citric_acid           5324 non-null   float64
 3   residual_sugar        5324 non-null   float64
 4   chlorides             5324 non-null   float64
 5   free_sulfur_dioxide   5319 non-null   float64
 6   total_sulfur_dioxide  5324 non-null   float64
 7   density               5321 non-null   float64
 8   pH                    5324 non-null   float64
 9   sulphates             5313 non-null   float64
 10  alcohol               5314 non-null   float64
 11  Type                  5324 non-null   object 
 12  quality               5324 non-null   int64  
dtypes: float64(11), int64(1), object(1)
memory usage: 582.3+ KB
```

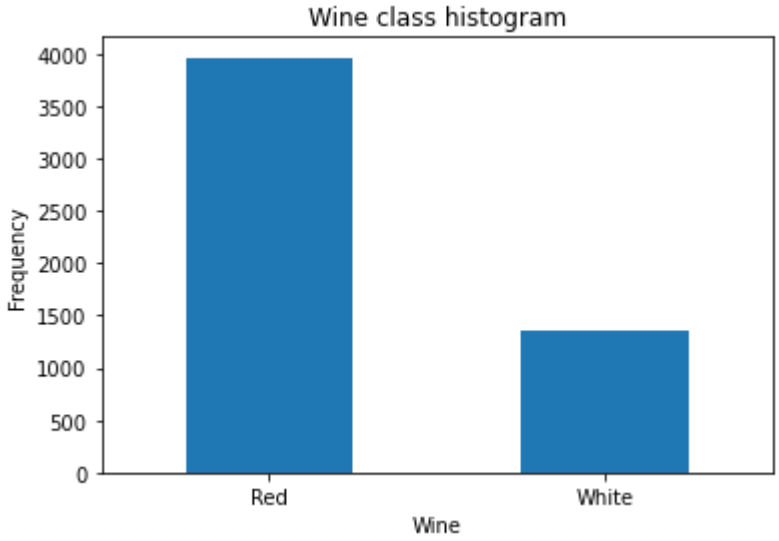
In [12]:
```
#Rating quality based on range

conditions= [(winedf['quality'] >= 7),
             (winedf['quality'] <= 6) & (winedf['quality'] >= 4),
             (winedf['quality'] <= 3)]

rating = [0,1,2]
winedf['rating'] = np.select(conditions, rating, default= 2)

winedf.rating.value_counts().sort_index()

#0 - Outstanding
#1 - Good
#2 - Not recommended
```

Out[12]:
```
0    1009
1    4285
2      30
Name: rating, dtype: int64
```

- We have more number of wines of Good quality

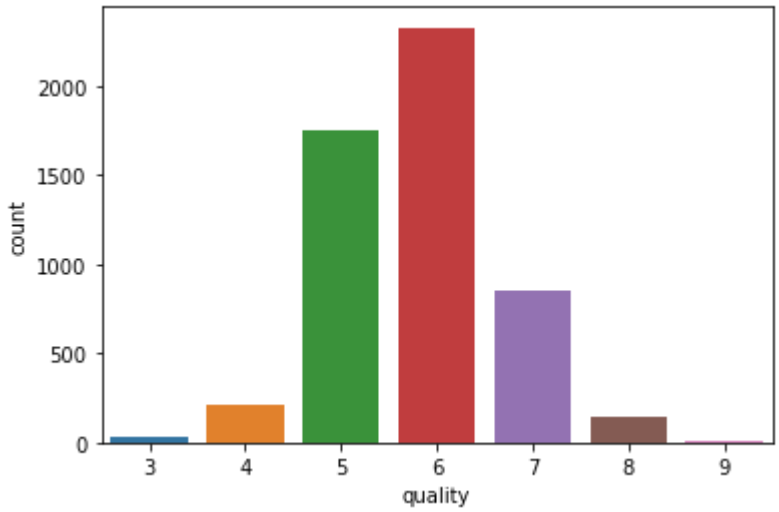
Bar Plots, Histograms and Inferences:

In [13]:
```
#Checking the frequency of Red and White wines

pd.value_counts(winedf['Type']).plot.bar()
plt.title('Wine class histogram')
plt.xlabel('Wine')
plt.ylabel('Frequency')
plt.xticks(rotation = 0)
plt.show()
```



In [11]:
```
#Range of values in 'Quality'
#Plotting No.of values for each attribute
sns.countplot(x='quality', data=winedf)
```

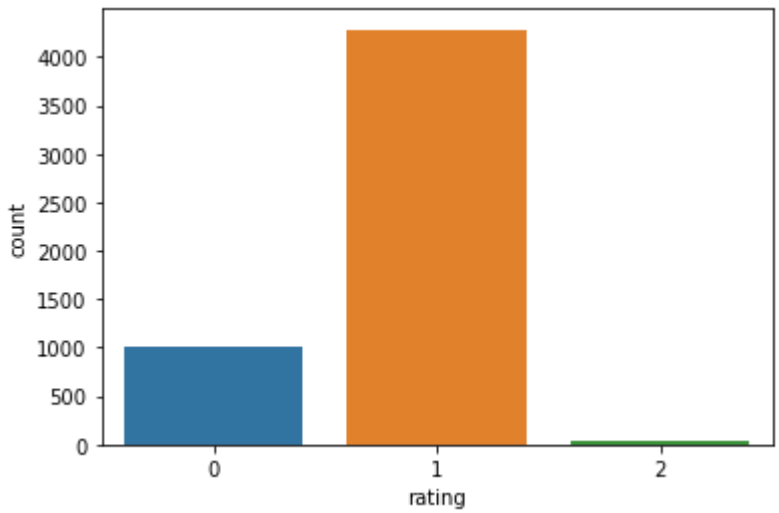
Out[11]: <AxesSubplot:xlabel='quality', ylabel='count'>



- Range is 3-9
- More number of Wines are of quality 6 which is a Good type of quality

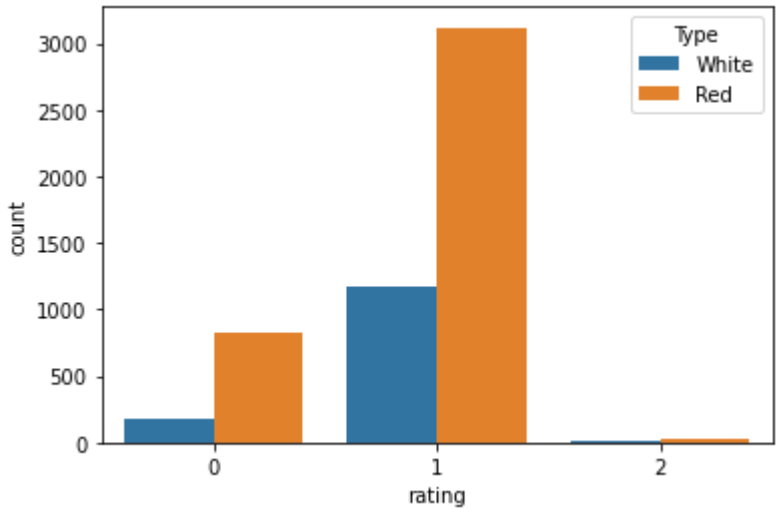
In [12]:
```
#Range of values in 'Rating'
#Plotting No.of values for each attribute
sns.countplot(x='rating', data=winedf)
```

Out[12]: <AxesSubplot:xlabel='rating', ylabel='count'>



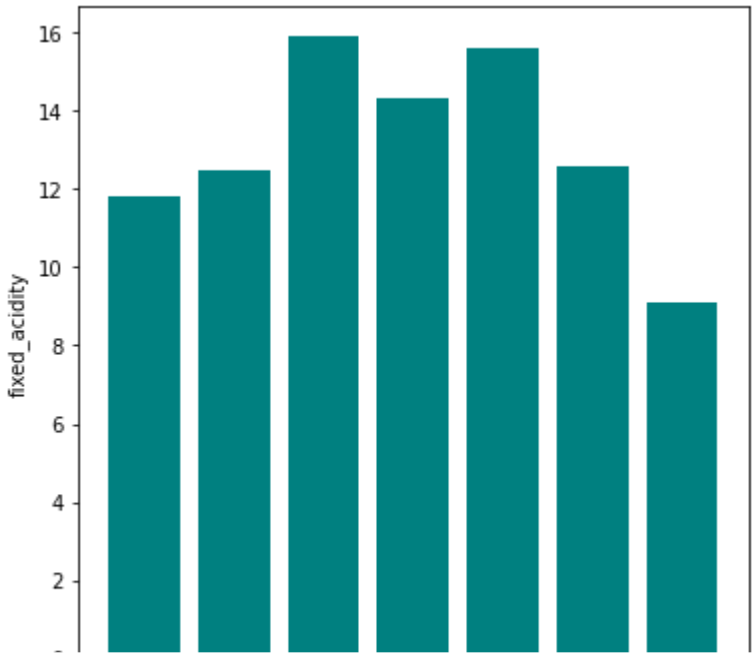
In [13]:
```
#Plotting rating for different types of wine

sns.countplot(x="rating", hue="Type", data=winedf)
```

Out[13]: <AxesSubplot:xlabel='rating', ylabel='count'>

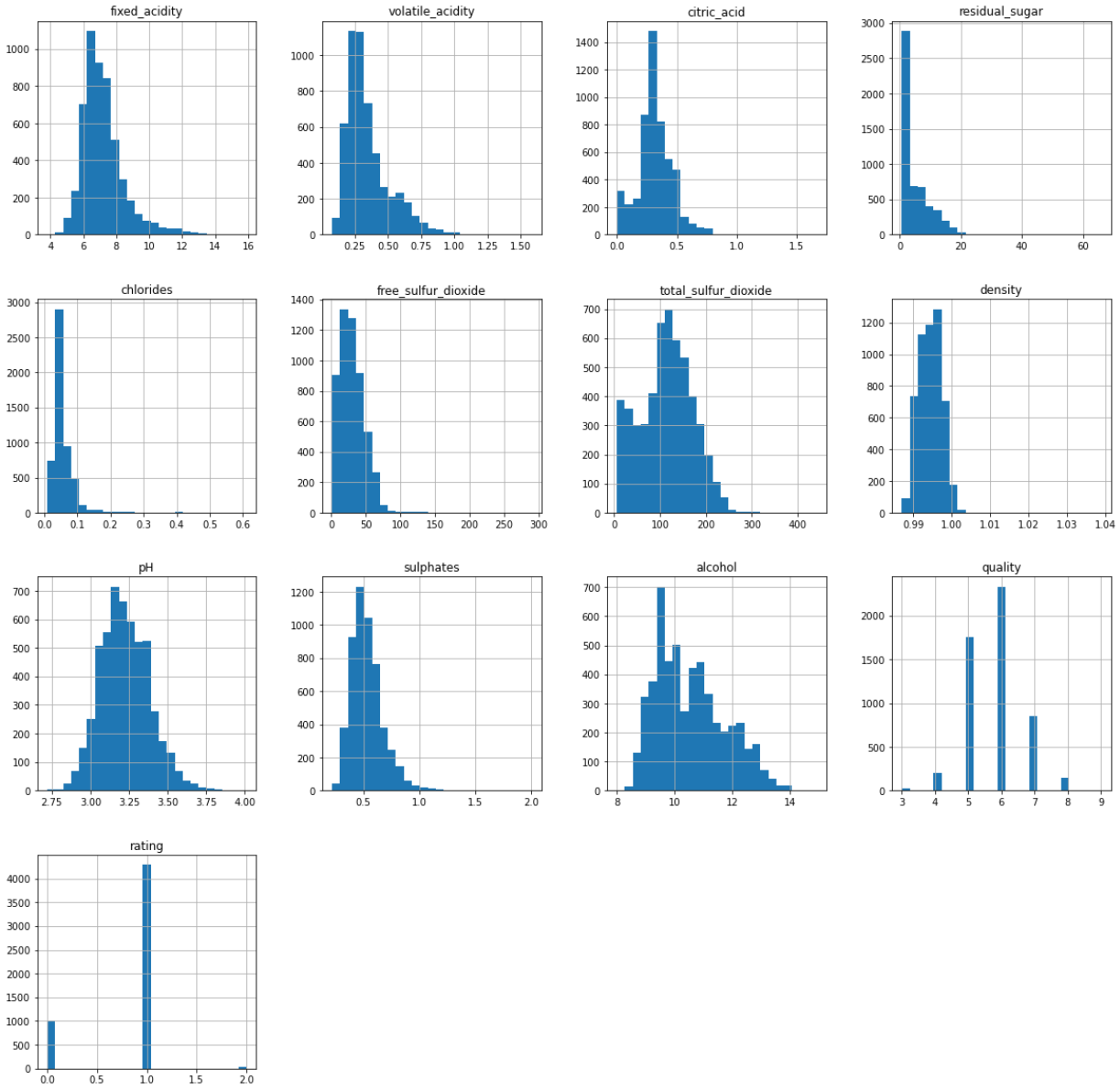


- More number of Outstanding wines are in Red wine
- More number of Good wines are in White wine
- Equal no.of wines are of bad wines in both types of wines

In [14]:
```
#Bar-graphs for each attribute against Quality to check what values can make char

for i in winedf.columns[:-2]:
    plt.figure(figsize=[6,6])
    # plot bar graph
    plt.bar(winedf['quality'],winedf[i],color='teal')
    # label x-axis
    plt.xlabel('quality')
    #label y-axis
    plt.ylabel(i)
```



In [15]:
```
#Histogram
winedf.hist(bins=25,figsize=(20,20))
plt.show()

#Histogram shows how the data is distributed on features
```

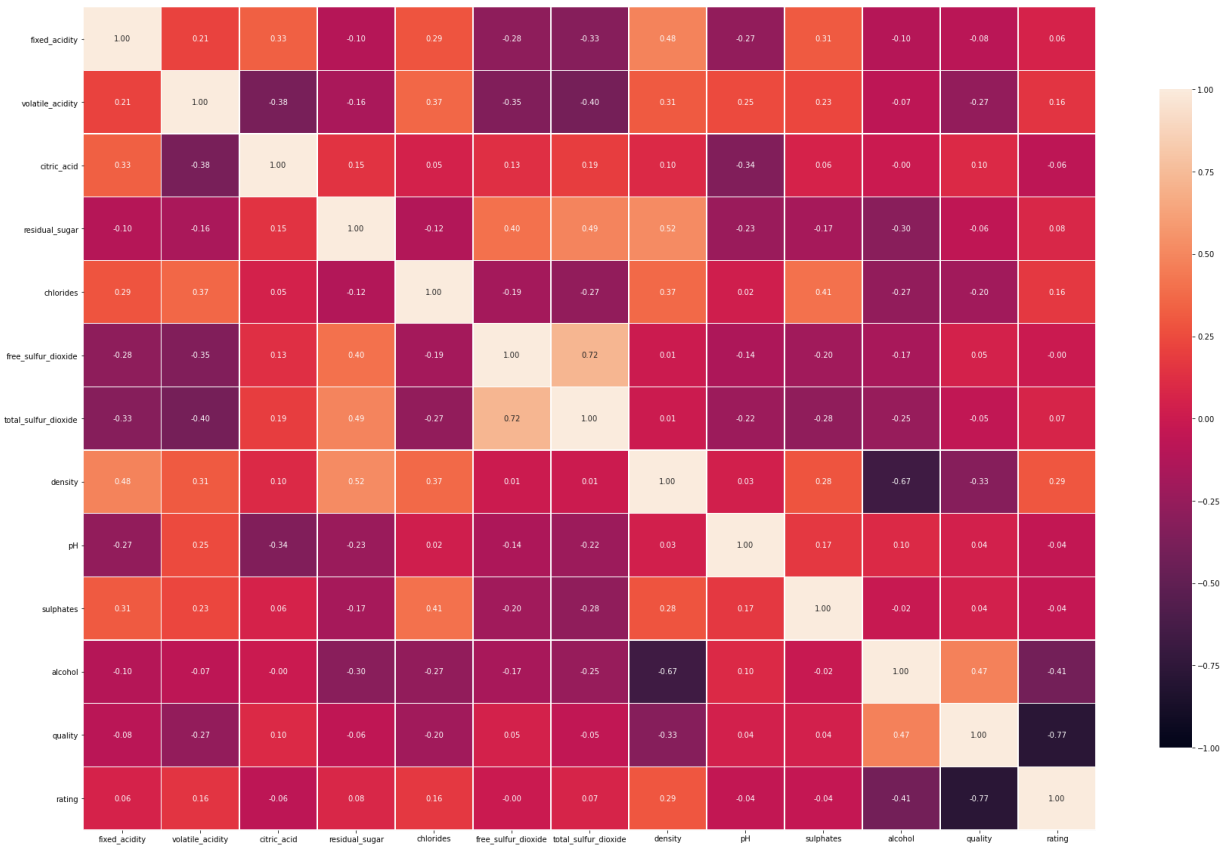


- Most of the attributes are normally distributed but right skewed.

Heatmap, Scatter Plots and Inferences

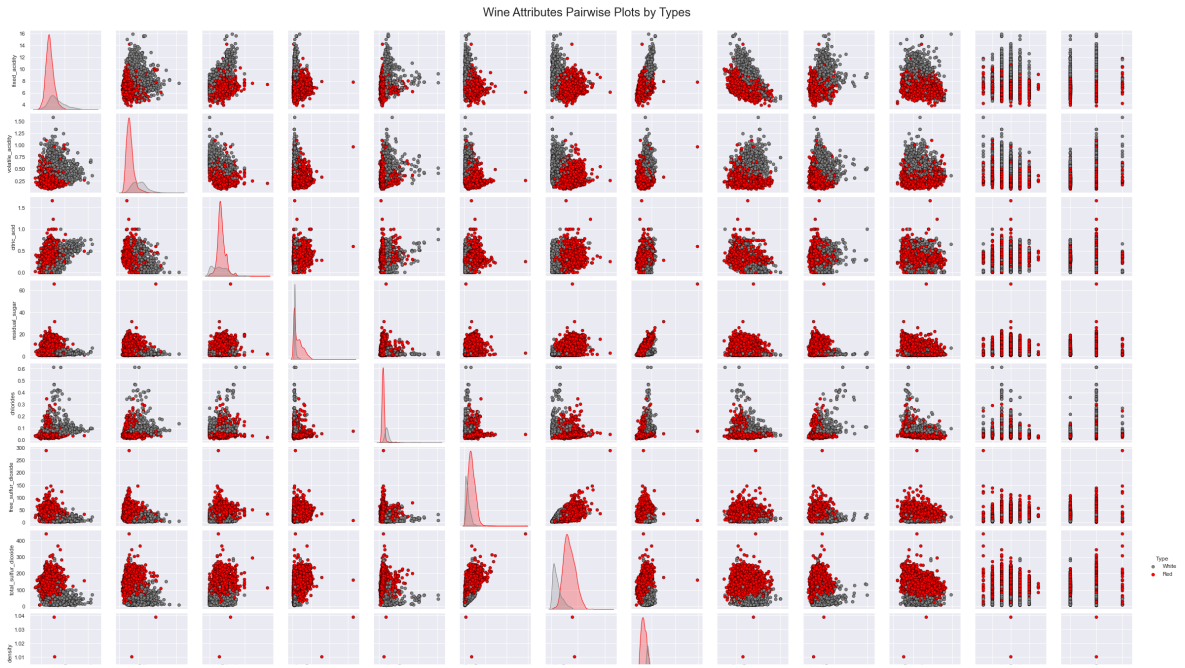
In [16]:
```
#Heatmap with Correlation Values

corr = winedf.corr()
fig, ax = plt.subplots()
fig.set_size_inches(30, 20)
sns.heatmap(corr, annot=True, fmt=".2f",linewidth=0.3,
            vmin=-1, vmax=1, cbar_kws={"shrink": .8})
```

Out[16]: <AxesSubplot:>



- Maximum Positive correlation is in between Total and Free Sulfur dioxide i.e., 0.72 (One column can be dropped)
- Maximum negative correlation is in between Alcohol-Density i.e., 0.67.

In [17]:
```
#Scatter plot for both Red and White wines

sns.set(font_scale=1.0)
g = sns.pairplot(data = winedf, hue='Type', palette={'Red': 'Red', 'White': 'Grey
fig = g.fig
fig.subplots_adjust(top=0.96, wspace=0.2)
t = fig.suptitle('Wine Attributes Pairwise Plots by Types', fontsize=24)
```



From the above Correlation plot and Scatter plot

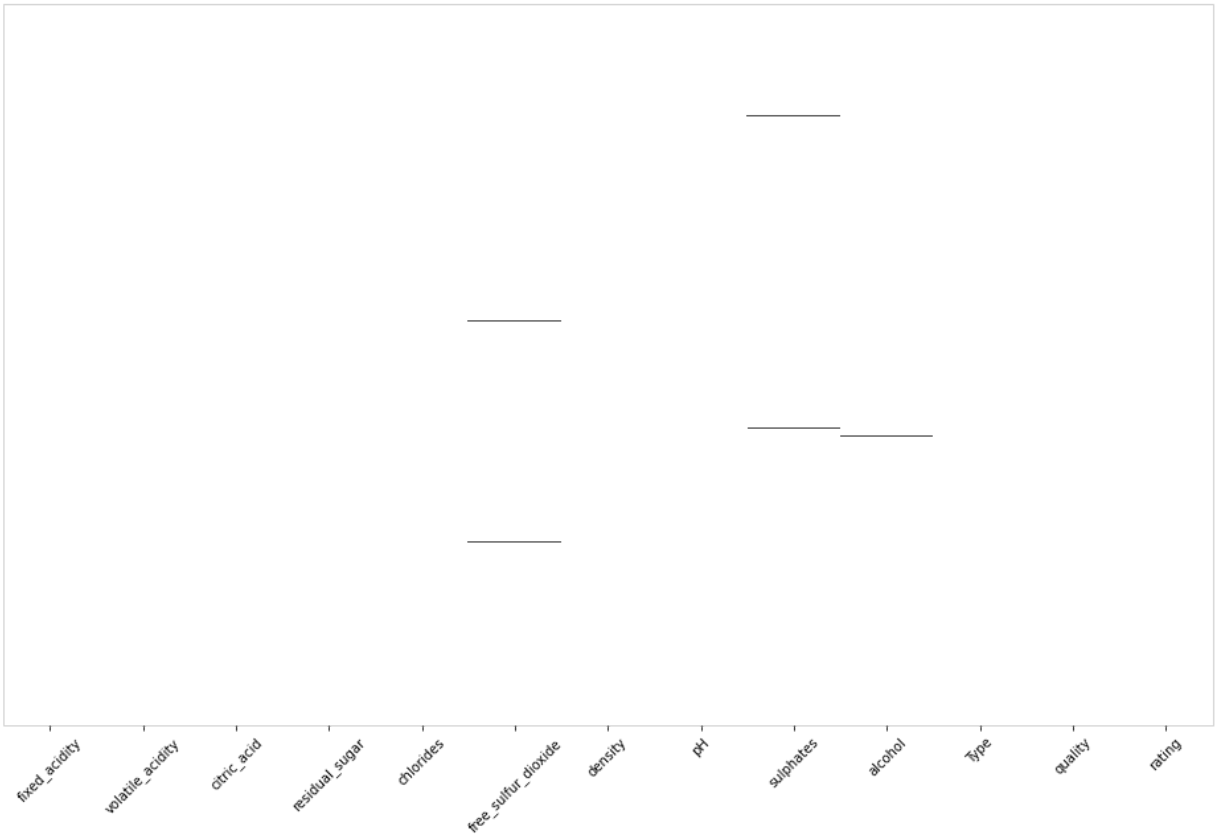
- We have inverse relationships between pH and fixed acidity and it is the same in case of citric acid and volatile acidity.
- We have a strong positive relationship between free sulfur dioxide and total sulfur dioxide.
- Sugar content in the wine may represent the wine density in both the wines.
- Low density contents are high on alcohol content.

In [14]:
```
#Dropping Total Suplhuric acid as corr is greater than 0.7
winedf = winedf.drop(columns=['total_sulfur_dioxide'])
winedf.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 5324 entries, 0 to 6495
Data columns (total 13 columns):
 #   Column               Non-Null Count  Dtype  
---  ------               --------------  -----  
 0   fixed_acidity        5324 non-null   float64
 1   volatile_acidity     5324 non-null   float64
 2   citric_acid          5324 non-null   float64
 3   residual_sugar       5324 non-null   float64
 4   chlorides            5324 non-null   float64
 5   free_sulfur_dioxide  5319 non-null   float64
 6   density              5321 non-null   float64
 7   pH                   5324 non-null   float64
 8   sulphates            5313 non-null   float64
 9   alcohol              5314 non-null   float64
 10  Type                 5324 non-null   object 
 11  quality              5324 non-null   int64  
 12  rating               5324 non-null   int32  
dtypes: float64(10), int32(1), int64(1), object(1)
memory usage: 561.5+ KB
```

Detecting Missing Values

In [15]:
```
#Checking if there are any missing values with the help of heatmap.

naInfo = np.zeros(winedf.shape)
naInfo[winedf.isna().values] = 1
naInfo = pd.DataFrame(naInfo, columns=winedf.columns)

fig, ax = plt.subplots()
fig.set_size_inches(13, 9)
ax = sns.heatmap(naInfo, vmin=0, vmax=1, cmap=["white", "#444444"], cbar=False, a
ax.set_yticks([])

rect = plt.Rectangle((0, 0), naInfo.shape[1], naInfo.shape[0], linewidth=1,
                     edgecolor='lightgrey',facecolor='none')
rect = ax.add_patch(rect)
rect.set_clip_on(False)

plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



In [16]:
```
# Calculating No.of Missing Values

print(winedf.isna().sum())
```

```
fixed_acidity           0
volatile_acidity        0
citric_acid             0
residual_sugar          0
chlorides               0
free_sulfur_dioxide     5
density                 3
pH                      0
sulphates              11
alcohol                10
Type                    0
quality                 0
rating                  0
dtype: int64
```

We have :

- 5 missed values in 'free sulfur dioxide' attribute
- 3 missed values in 'density' attribute
- 11 missed values in 'sulphates' attribute
- 10 missed values in 'alcohol' attribute

Data preparation

Imputing Missing Values

In [17]:
```
#Replacing the missing values with mean values.

class DataFrameImputer(TransformerMixin):
    def fit(self, X, y=None):

        self.fill = pd.Series([X[c].value_counts().index[0]
            if X[c].dtype == np.dtype('O') else X[c].mean() for c in X],
            index=X.columns)

        return self

    def transform(self, X, y=None):
        return X.fillna(self.fill)

winedf = DataFrameImputer().fit_transform(winedf)
```

In [18]:
```
#Checking if the values are imputed
winedf.info()

#Checking if there are any null values
winedf.isnull().values.any()
```

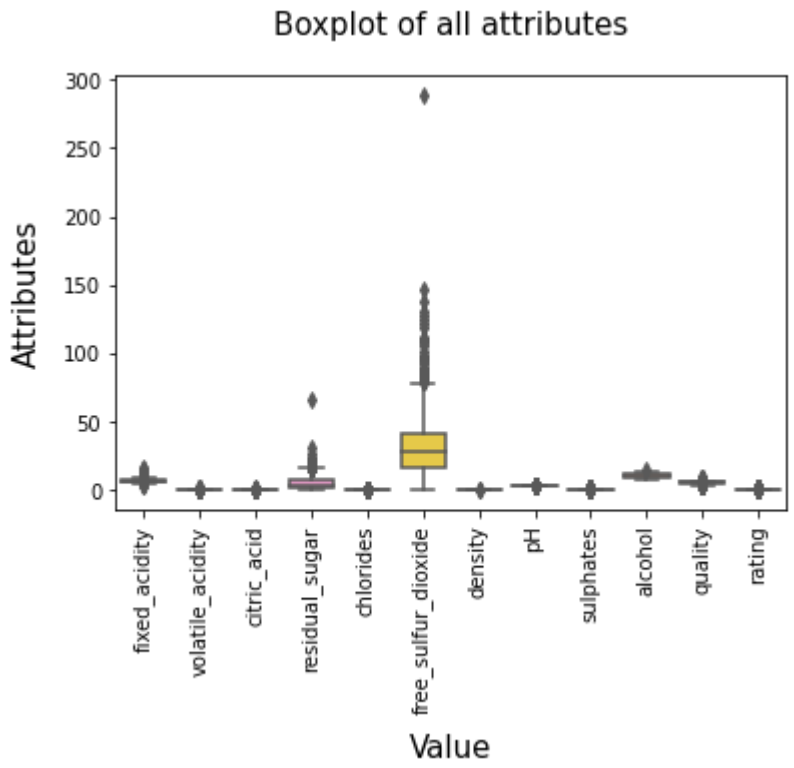
```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 5324 entries, 0 to 6495
Data columns (total 13 columns):
 #   Column               Non-Null Count  Dtype  
---  ------               --------------  -----  
 0   fixed_acidity        5324 non-null   float64
 1   volatile_acidity     5324 non-null   float64
 2   citric_acid          5324 non-null   float64
 3   residual_sugar       5324 non-null   float64
 4   chlorides            5324 non-null   float64
 5   free_sulfur_dioxide  5324 non-null   float64
 6   density              5324 non-null   float64
 7   pH                   5324 non-null   float64
 8   sulphates            5324 non-null   float64
 9   alcohol              5324 non-null   float64
 10  Type                 5324 non-null   object 
 11  quality              5324 non-null   int64  
 12  rating               5324 non-null   int32  
dtypes: float64(10), int32(1), int64(1), object(1)
memory usage: 561.5+ KB
```

Out[18]:
```
False
```

- Null values are imputed

Detecting Outliers and Imputing

In [19]:
```
#Box Plot to detect outliers
import seaborn as sns
ax = sns.boxplot(data = winedf, orient = "v", palette = "Set2", color='red')
#add labels
ax.set_xlabel("Value", fontsize = 15, labelpad = 10)
ax.set_ylabel("Attributes", fontsize = 15, labelpad = 10)
ax.set_title("Boxplot of all attributes", fontsize = 15,
             pad = 20)
plt.xticks(rotation=90)
#edit ticks
ax.tick_params(which = "both", labelsize = 10)
```



In [20]:
```
#Checking outlier values for Residual_Sugar and Free_Sulphur_Dioxide

cols=['residual_sugar','free_sulfur_dioxide']
for x in cols:
    print(winedf.groupby([x]).size())
```

```
residual_sugar
0.600030      1
0.700035      7
0.800040     25
0.900045     36
0.950048      3
             ..
22.601130     1
23.501175     1
26.051303     1
31.601580     1
65.803290     1
Length: 316, dtype: int64
free_sulfur_dioxide
1.000050        2
2.000100        2
3.000150       50
4.000200       43
5.000250      111
             ...
128.006400      1
131.006550      1
138.506925      1
146.507325      1
289.014450      1
Length: 134, dtype: int64
```

In [21]:
```
 values will null values

unds(df):
le(0.25)
le(0.75)

+1.5*IQR)
1.5*IQR)
,lower)

 Outliers '''

owerBounds(winedf['residual_sugar'])
_sugar'] = np.where((winedf['residual_sugar'] > tupleKM[0]) | (winedf['residual_su

xide
owerBounds(winedf['free_sulfur_dioxide'])
fur_dioxide'] = np.where((winedf['free_sulfur_dioxide'] > tupleMY[0]) | (winedf['
```

In [22]:
```
#Imputing Missing Values for residual_sugar/ free_sulfur_dioxide
from sklearn.impute import KNNImputer
imputer = KNNImputer(n_neighbors=2)
winedf[['residual_sugar','free_sulfur_dioxide']]=imputer.fit_transform(winedf[['r
print(winedf.info())
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 5324 entries, 0 to 6495
Data columns (total 13 columns):
 #   Column               Non-Null Count  Dtype  
---  ------               --------------  -----  
 0   fixed_acidity        5324 non-null   float64
 1   volatile_acidity     5324 non-null   float64
 2   citric_acid          5324 non-null   float64
 3   residual_sugar       5324 non-null   float64
 4   chlorides            5324 non-null   float64
 5   free_sulfur_dioxide  5324 non-null   float64
 6   density              5324 non-null   float64
 7   pH                   5324 non-null   float64
 8   sulphates            5324 non-null   float64
 9   alcohol              5324 non-null   float64
 10  Type                 5324 non-null   object 
 11  quality              5324 non-null   int64  
 12  rating               5324 non-null   int32  
dtypes: float64(10), int32(1), int64(1), object(1)
memory usage: 561.5+ KB
None
```

In [30]:
```
# Creating dummy variables for 'Object' type attributes

winedf['Type_New']=[1 if p == 'Red' else 0 for p in winedf.Type]
winedf=winedf.drop(['Type'],axis=1)
winedf.head()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-30-518a571889dd> in <module>
      1 # Creating dummy variables for 'Object' type attributes
      2
----> 3 winedf['Type_New']=[1 if p == 'Red' else 0 for p in winedf.Type]
      4 winedf=winedf.drop(['Type'],axis=1)
      5 winedf.head()

~\anaconda3\lib\site-packages\pandas\core\generic.py in __getattr__(self, name)
   5463             if self._info_axis._can_hold_identifiers_and_holds_name(nam
e):
   5464                 return self[name]
-> 5465             return object.__getattribute__(self, name)
   5466
   5467     def __setattr__(self, name: str, value) -> None:

AttributeError: 'DataFrame' object has no attribute 'Type'
```

PCA (Principal Component Analysis)

- PCA is used to explore large datasets
- Features that are highly correlated, reduce to fewer features without losing too much information.
- Final dataset contains most of the information with smaller number of numerical variables

In [31]:
```
#sklearn.preprocessing.MinMaxScaler

def scaler(df,scaler,cols):
    df = pd.DataFrame(scaler.fit_transform(df),columns=cols) #scaled object
    return df

scaled_df=scaler(df=winedf.iloc[:,1:],scaler = preprocessing.MinMaxScaler(),cols=
scaled_df.head(10)
```

Out[31]:

	volatile_acidity	citric_acid	residual_sugar	chlorides	free_sulfur_dioxide	density	pH	su
0	0.590000	0.000000	0.103896	0.131229	0.194805	0.201465	0.666666	0
1	0.206667	0.379518	0.110390	0.136211	0.064935	0.256223	0.341085	0
2	0.126667	0.295181	0.435065	0.061462	0.675325	0.163675	0.403101	0
3	0.073333	0.180723	0.792208	0.073088	0.616883	0.221517	0.325581	0
4	0.173333	0.180723	0.681818	0.063123	0.350649	0.211877	0.356589	0
5	0.353333	0.072289	0.045455	0.000000	0.831169	0.005591	0.418604	0
6	0.220000	0.174699	0.077922	0.038205	0.324675	0.034320	0.186046	0
7	0.100000	0.108434	0.512987	0.058139	0.753247	0.163289	0.341085	0
8	0.186667	0.096386	0.370130	0.044849	0.454545	0.136701	0.348837	0
9	0.086667	0.174699	0.308442	0.033223	0.545455	0.077316	0.162791	0

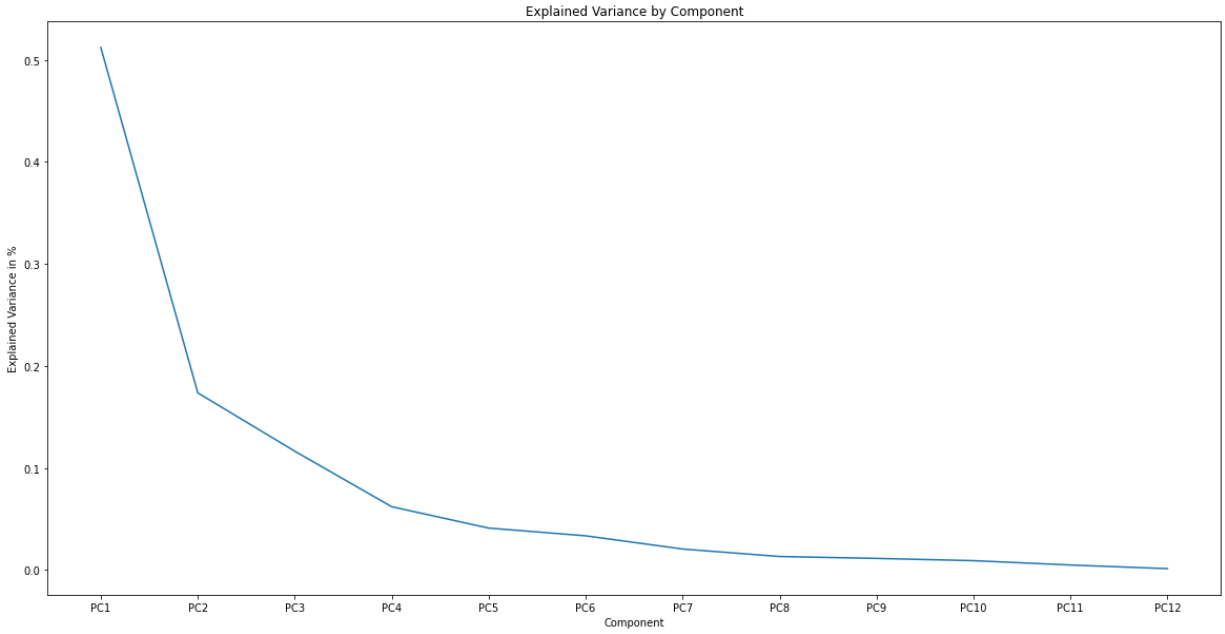
In [32]:
```
#sklearn.preprocessing.MinMaxScaler

def scale_sum(df,scaler,index):
    df = pd.DataFrame(scaler.fit_transform(df)) #scaled object
    pcs = PCA()  #pca technique
    pcs.fit(df) #fit data to pca
    pcsSummary_df = pd.DataFrame({'Explained Variance': np.sqrt(pcs.explained_var
                            'Explained Variance Ratio': pcs.explained_variance_rat
                            'Cumulative Proportion': np.cumsum(pcs.explained_varia
    pcsSummary_df = pcsSummary_df.transpose()
    pcsSummary_df.columns = ['PC{}'.format(i) for i in range(1, len(pcsSummary_df
    pcsSummary_df.round(4)

    pcsComponents_df = pd.DataFrame(pcs.components_.transpose(),columns=pcsSummar

    scores=pd.DataFrame(pcs.fit_transform(df),columns=pcsSummary_df.columns)

    return pcsSummary_df, pcsComponents_df, scores  # return #1 summary of PCA, #

pcsSummary_df=scale_sum(df=winedf.iloc[:,1:],scaler = preprocessing.MinMaxScaler(
pcsSummary_df
```

Out[32]:

	PC1	PC2	PC3	PC4	PC5	PC6	PC7	PC8	P
Explained Variance	0.474694	0.276431	0.226402	0.165231	0.134511	0.121509	0.095205	0.076420	0.0708
Explained Variance Ratio	0.512227	0.173704	0.116519	0.062061	0.041130	0.033562	0.020604	0.013276	0.0114
Cumulative Proportion	0.512227	0.685931	0.802450	0.864511	0.905640	0.939203	0.959807	0.973083	0.9845

In [33]:
```
#Scree Plot
plt.figure(figsize=(20,10))
plt.plot(pcsSummary_df[1:2].transpose())
plt.xlabel('Component')
plt.ylabel('Explained Variance in %')
plt.title('Explained Variance by Component')
```

Out[33]:
```
Text(0.5, 1.0, 'Explained Variance by Component')
```



- According to Elbow rule, Infection point is at 4th component

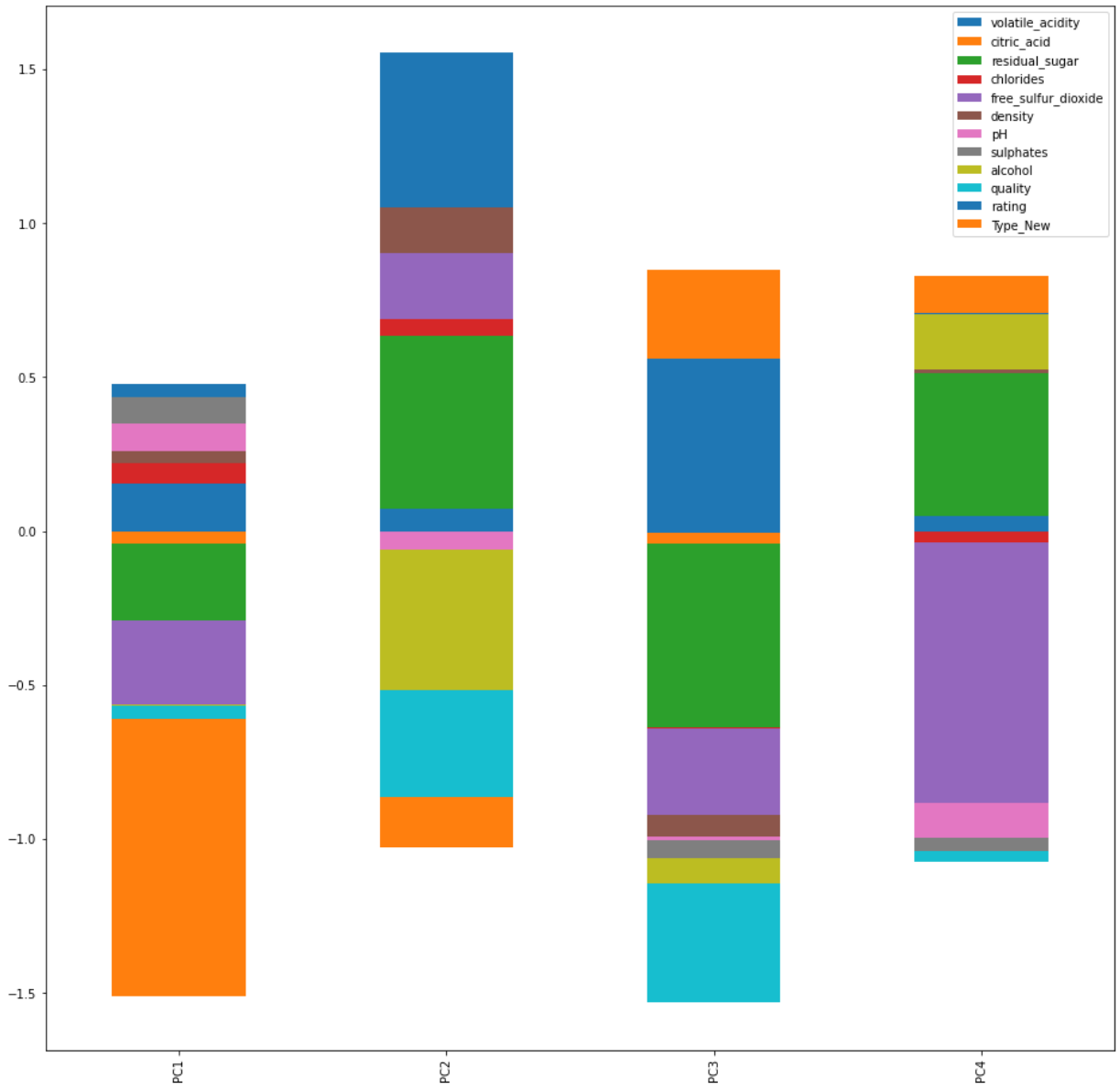
In [34]:
```
pcsComponents_df=scale_sum(df=winedf.iloc[:,1:],scaler = preprocessing.MinMaxScal
pcsComponents_df

#how much does each feature contribute to each component
```

Out[34]:

	PC1	PC2	PC3	PC4	PC5	PC6	PC7	
volatile_acidity	0.156171	0.072370	-0.004004	0.050165	0.136109	-0.175580	0.622798	0.2
citric_acid	-0.040624	0.001762	-0.036174	-0.002703	-0.155647	0.404488	-0.445897	0.4
residual_sugar	-0.250725	0.561145	-0.598636	0.462314	0.183396	-0.029967	-0.045846	0.0
chlorides	0.063243	0.052229	-0.001199	-0.031946	-0.094059	0.062145	-0.016361	0.1
free_sulfur_dioxide	-0.272357	0.215724	-0.281731	-0.848891	0.232393	0.116242	0.100533	0.0
density	0.039957	0.149823	-0.069192	0.014128	-0.116301	-0.031242	-0.073412	0.0
pH	0.089976	-0.058579	-0.014371	-0.112426	0.297793	-0.798992	-0.429882	0.1
sulphates	0.086360	-0.001228	-0.055298	-0.041426	-0.056414	0.065677	-0.287107	0.3
alcohol	-0.003651	-0.454826	-0.085226	0.177977	0.775285	0.309041	0.012567	0.1
quality	-0.043045	-0.347634	-0.384314	-0.034923	-0.088681	0.021517	-0.247070	-0.6
rating	0.041247	0.500395	0.561669	0.003083	0.377991	0.166962	-0.256394	-0.3
Type_New	-0.901222	-0.163155	0.287144	0.121018	-0.059688	-0.140589	0.035577	0.0

In [35]:
```
pcsComponents_df[['PC1','PC2','PC3','PC4',]].transpose().plot(kind='bar', stacked
```

Out[35]: <AxesSubplot:>



Analysing the features represented by each component

For Component 1

- Type_New
- Free Sulphur Diaoxide
- Residual Sugar

For Component 2

- Volatile acidity
- Residual Sugar
- Alcohol

For Component 3

- Volatile acidity
- Residual Sugar
- Quality

For Component 4

- Free Sulphur Dioxide
- Residual Sugar

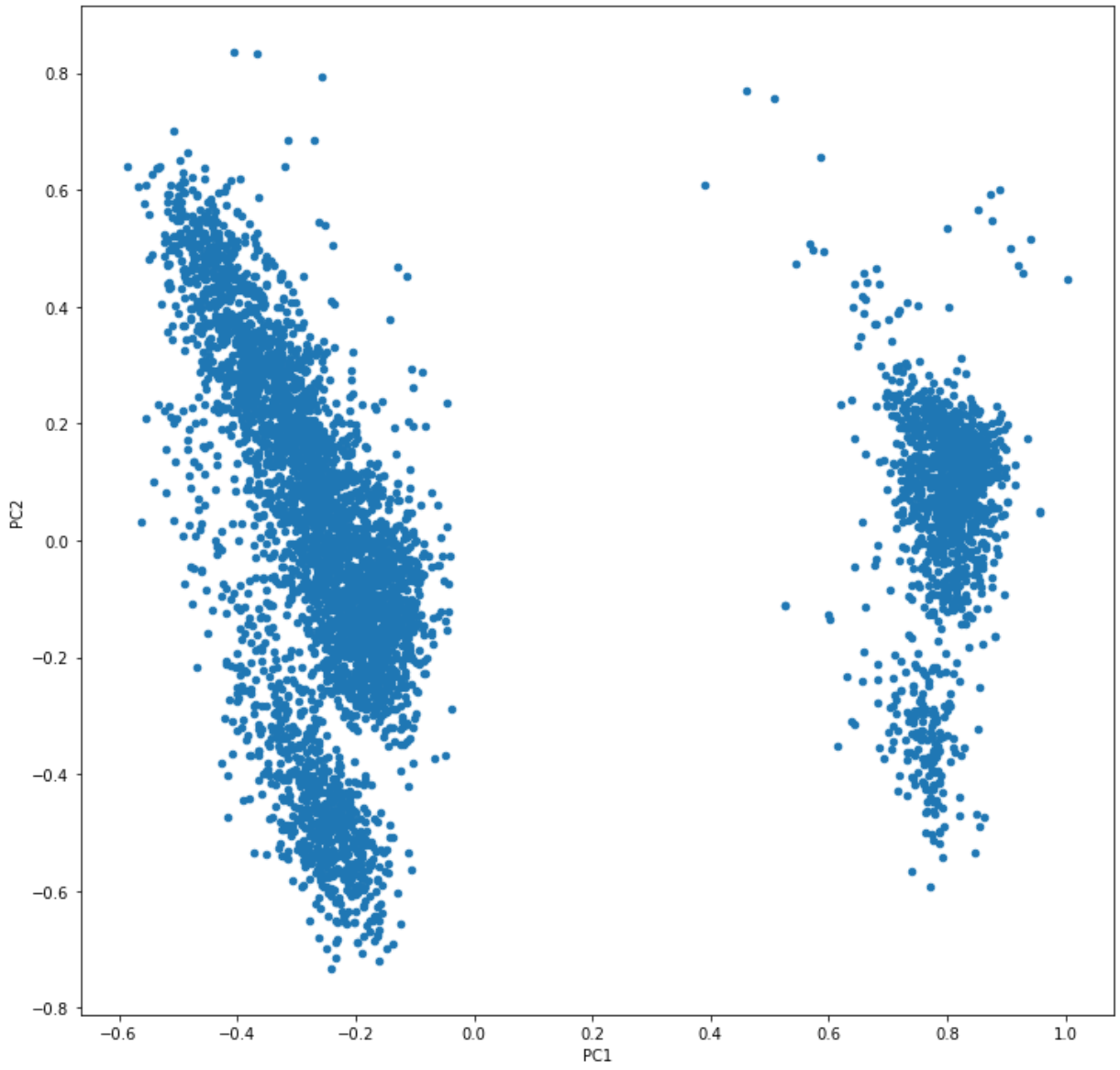
In [36]:
```
scores=scale_sum(df=winedf.iloc[:,1:],scaler = preprocessing.MinMaxScaler(),index
scores.head(10)
```

Out[36]:

	PC1	PC2	PC3	PC4	PC5	PC6	PC7	PC8	PC9
0	0.878996	0.131833	0.033554	-0.034612	0.133040	-0.277994	0.169709	0.064470	0.100815
1	0.802302	0.154191	0.069184	0.081731	-0.183936	0.148274	-0.087992	0.040670	-0.156529
2	-0.368408	0.239437	-0.059510	-0.187142	-0.050316	-0.014528	-0.068843	-0.035740	-0.016927
3	-0.474477	0.135406	-0.600907	0.023037	-0.235662	-0.090264	0.032775	-0.021409	-0.026407
4	-0.321714	0.411926	-0.046105	0.202584	-0.119262	-0.102161	0.014322	0.019017	-0.004728
5	-0.275535	-0.182825	0.160568	-0.371469	0.508606	0.060831	0.269874	0.079490	-0.008463
6	-0.230924	-0.654457	-0.184230	0.060430	-0.002304	0.157250	0.192978	-0.106364	0.033234
7	-0.401625	0.339238	-0.061843	-0.198863	0.035997	-0.017240	0.060869	-0.005255	-0.025941
8	-0.265486	0.175891	0.101919	-0.003073	-0.009709	-0.055727	0.075152	0.028523	0.063937
9	-0.358515	-0.322821	-0.306434	-0.058455	-0.094621	0.148527	0.154030	-0.034327	-0.045020

In [37]:
```
#Plotting first 2 components

scores.plot.scatter(x='PC1', y='PC2',figsize=(12, 12))
```

Out[37]: <AxesSubplot:xlabel='PC1', ylabel='PC2'>



In [63]:
```
cols=['PC1', 'PC2', 'PC3'] #list # of components I want to use

silhouette=[]
inertia=[]
clusters=[]
for i in range(2,40):  #2=2 clusters, use enough to where we see declining silhou
    kmeans = KMeans(n_clusters=i, random_state=0).fit(scores[cols])
    silhouette.append(silhouette_score(scores[cols],kmeans.labels_)) #silhoustte
    inertia.append(kmeans.inertia_)
    clusters.append(i)
```

In [ ]:
```
kmeans_eval=pd.DataFrame({'silhouette_score': silhouette,'inertia': inertia,},inc
kmeans_eval  #index # clusters, silhouette_score is the score for that many clust
```

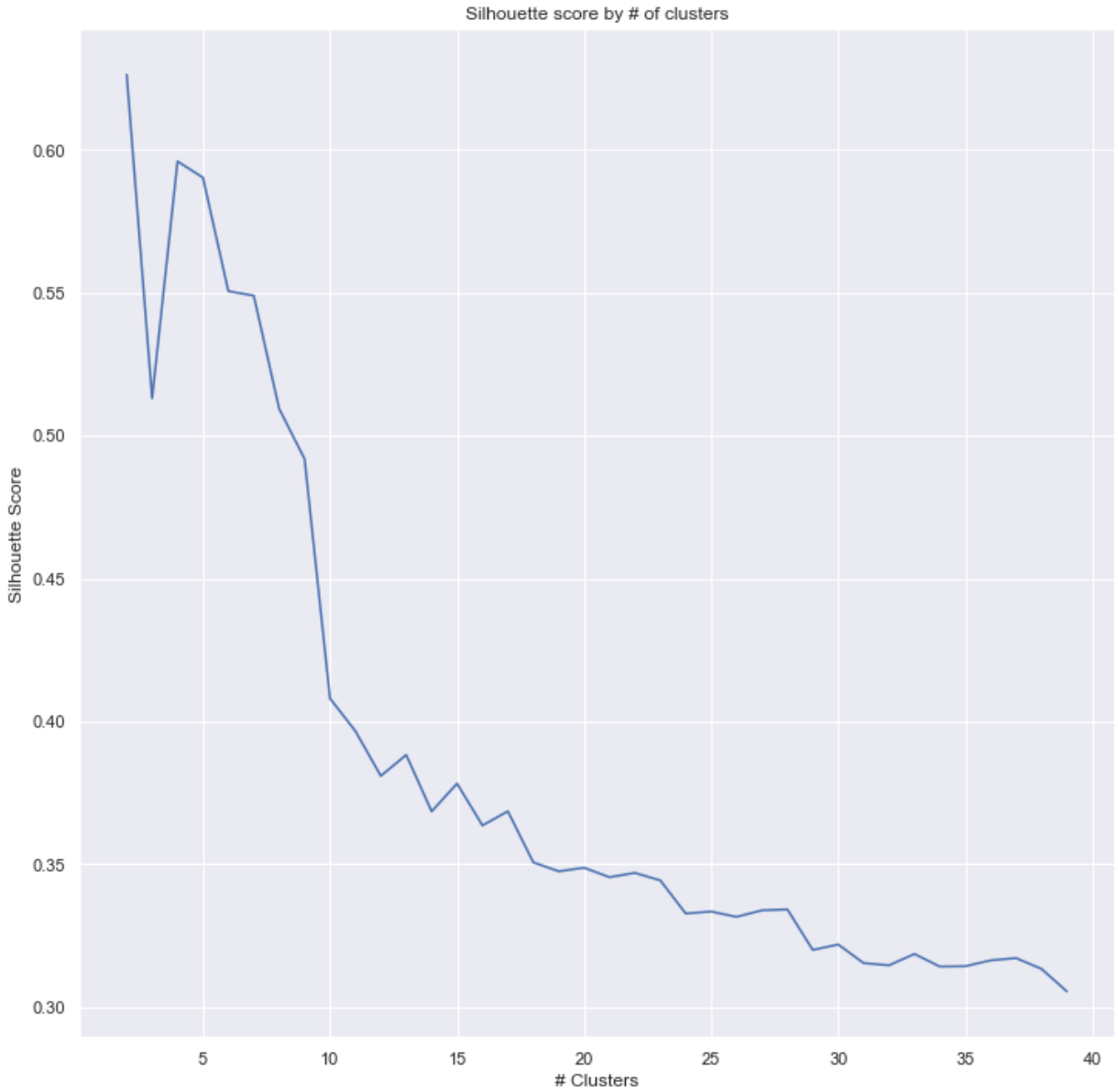
Silhouette Score

- A method that calculates the average distance of each point from all other points in a cluster, and then compares that value with the average distance to every point in every other cluster.
- ranges from −1 to +1, where a high value indicates that the object is well matched to its own cluster and poorly matched to neighboring clusters

In [ ]:
```
kmeans_eval['inertia'].plot(figsize=(12, 12))#use elbow method of finding # of cc
plt.xlabel('# Clusters')
plt.ylabel('Inertia')
plt.title('Scree Plot Inertia')
```

In [38]:
```
kmeans_eval['silhouette_score'].plot(figsize=(12, 12)) #plot silhouete_score, vis
plt.xlabel('# Clusters')
plt.ylabel('Silhouette Score')
plt.title('Silhouette score by # of clusters')
#print(kmeans_eval['silhouette_score'])
```

Out[38]: Text(0.5, 1.0, 'Silhouette score by # of clusters')

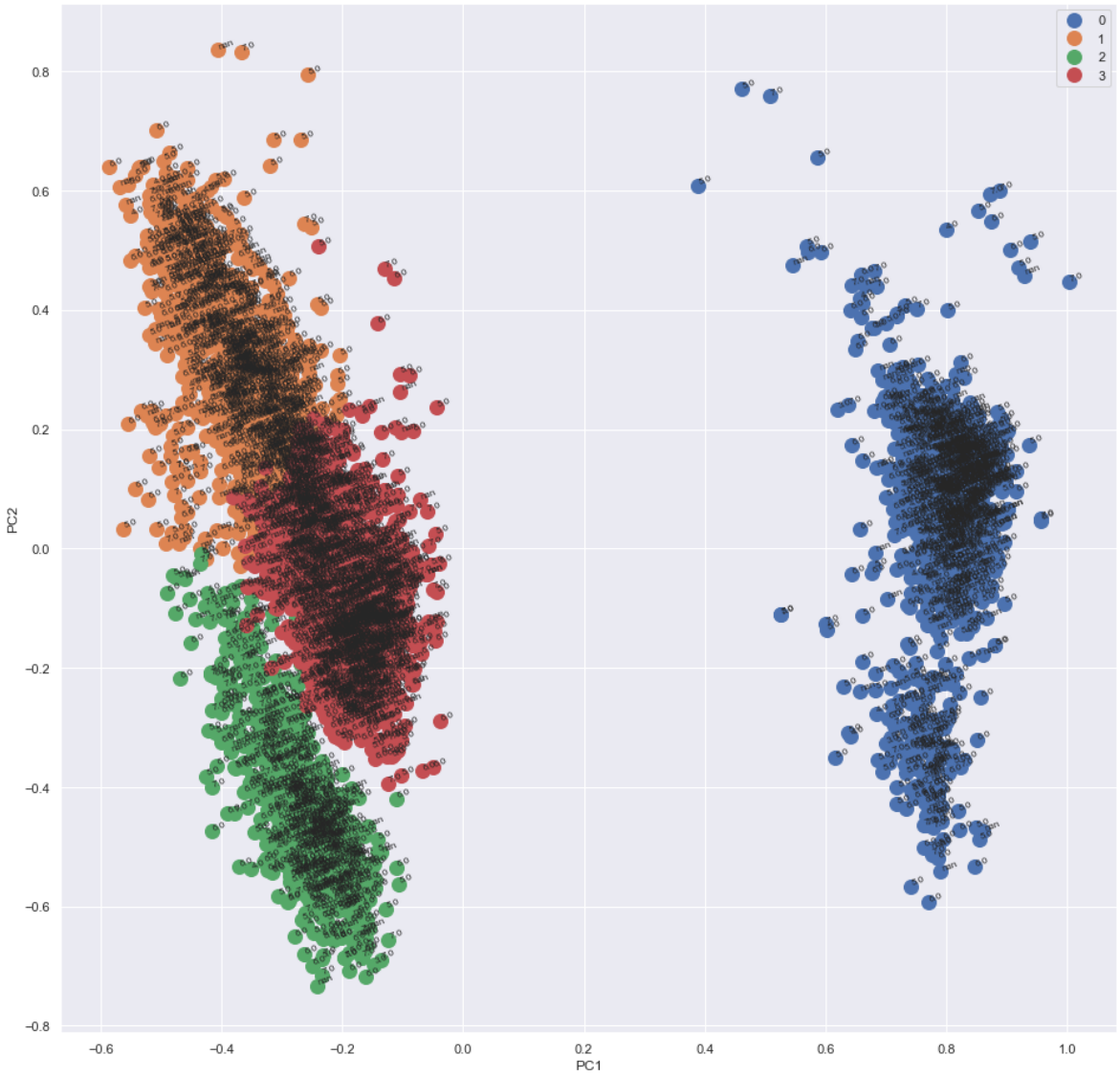


In [ ]:
```
kmeans= KMeans(n_clusters=5).fit(scores[cols]) #kmeans number of clusters is subj
scores['cluster']=kmeans.labels_   #kmeans
scores['quality']=winedf['quality']


groups = scores.groupby('cluster') #grouping subplots
winedf['cluster']=scores['cluster']  #add the cluster to primary df
ax = scores.plot.scatter(x='PC1', y='PC2', figsize=(16, 16))


for name, group in groups:  #grouping subplots
    ax.plot(group.PC1, group.PC2, marker='o', linestyle='', ms=12, label=name) #g
ax.legend() #grouping subplots
```

In [40]:
```
kmeans= KMeans(n_clusters=4).fit(scores[cols]) #kmeans number of clusters is subj
scores['cluster']=kmeans.labels_   #kmeans


groups = scores.groupby('cluster') #grouping subplots

winedf['cluster']=scores['cluster']  #add the cluster to primary df

ax = scores.plot.scatter(x='PC1', y='PC2', figsize=(16, 16))
points = scores[['PC1','PC2','quality']]

_ = points.apply(lambda x:   ax.text(*x, rotation=20, horizontalalignment='left',
                                     verticalalignment='bottom', fontsize=8), axi


for name, group in groups:  #grouping subplots
    ax.plot(group.PC1, group.PC2, marker='o', linestyle='', ms=12, label=name) #g
ax.legend() #grouping subplots
```

```
*c* argument looks like a single numeric RGB or RGBA sequence, which should be
avoided as value-mapping will have precedence in case its length matches with *
x* & *y*.  Please use the *color* keyword-argument or provide a 2-D array with
a single row if you intend to specify the same RGB or RGBA value for all point
s.
```

Out[40]:
```
<matplotlib.legend.Legend at 0x7f78e5724490>
```



K-Means Clustering

- Group similar data points together and discover underlying patterns.
- K-means allocates every data point to the nearest cluster
- The 'means' in the K-means refers to averaging of the data; that is, finding the centroid.

In [48]:
```
rating = winedf["rating"]
```

In [49]:
```
# DATA PREPARATION: Prepare target variable and independent variable
X =  np.array(winedf.iloc[:,winedf.columns != 'Type'])
y = rating
print("Shape of X(PREDICTORS): {}".format(X.shape))
print("Shape of y(TARGET): {}".format(y.shape))
```

```
Shape of X(PREDICTORS): (5324, 13)
Shape of y(TARGET): (5324,)
```

DATA PREPARATION

In [50]:
```
# DATA PREPARATION: Create training/testing sets
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.3, random_st


print("Number of X_train dataset: ", X_train.shape)
print("Number of y_train dataset: ", y_train.shape)
print("\nNumber of X_test dataset: ", X_test.shape)
print("Number of y_test dataset: ", y_test.shape)
```

```
Number of X_train dataset:  (3726, 13)
Number of y_train dataset:  (3726,)

Number of X_test dataset:  (1598, 13)
Number of y_test dataset:  (1598,)
```

MODEL - PIPELINE CREATION

In [53]:
```
# MODEL - PIPELINE: Libraries needed
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LinearRegression, Lasso, Ridge,RidgeCV, LassoCV,
```

In [54]:
```
# 4. MODEL - PIPELINE CREATION- Create pipelines with or without PCA

##  First create three different pipelines without PCA
# Linear Regression
pipeline_linearreg = Pipeline([('scalar1', MinMaxScaler()),
                               ('lr_model', LinearRegression())])
# Ridge Regression
pipeline_Ridgereg = Pipeline([('scalar1', StandardScaler()),
                               ('ridge_reg_model', Ridge(normalize=True,alpha=1))])
# Lasso Regression
pipeline_LassoReg = Pipeline([('scalar1', StandardScaler()),
                               ('Lasso_reg_model', Lasso(normalize=False,alpha=1))])

## Now create pipelines with PCA

# Pipeline for logistic regression
pipeline_pca_linearreg = Pipeline([('scalar1', MinMaxScaler()),
                           ('pca1', PCA(n_components=2)),
                           ('lr_model', LinearRegression())])

# Pipeline for Ridge regression
pipeline_pca_Ridgereg = Pipeline([('scalar1', StandardScaler()),
                               ('ridge_reg_model', Ridge(normalize=True,alpha=1))])
# Pipeline for Lasso regression
pipeline_pca_LassoReg = Pipeline([('scalar1', StandardScaler()),
                               ('Lasso_reg_model', Lasso(normalize=False,alpha=1))])

# Create the list of pipelines for classifier
pipelines = [pipeline_linearreg, pipeline_Ridgereg,pipeline_LassoReg,pipeline_pca
pipelines
```

Out[54]:
```
[Pipeline(steps=[('scalar1', MinMaxScaler()), ('lr_model', LinearRegression
())]),
 Pipeline(steps=[('scalar1', StandardScaler()),
                 ('ridge_reg_model', Ridge(alpha=1, normalize=True))]),
 Pipeline(steps=[('scalar1', StandardScaler()),
                 ('Lasso_reg_model', Lasso(alpha=1))]),
 Pipeline(steps=[('scalar1', MinMaxScaler()), ('pca1', PCA(n_components=2)),
                 ('lr_model', LinearRegression())]),
 Pipeline(steps=[('scalar1', StandardScaler()),
                 ('ridge_reg_model', Ridge(alpha=1, normalize=True))]),
 Pipeline(steps=[('scalar1', StandardScaler()),
                 ('Lasso_reg_model', Lasso(alpha=1))])]
```

In [55]:
```
# 4. MODEL - PIPELINE CREATION: Fit the pipelines in training dataset
for pipe in pipelines:
    pipe.fit(X_train, y_train)
    #predicted_values = pipe.predict(X_test)
```

In [58]:
```
# 5. MODEL EVALUATION: Dictionary of pipelines and classifier type for ease of re
pipe_dict = {0: 'Linear Regression',1:'Ridge Regression',2:'Lasso Regression',3:
             4:'Ridge Regression with PCA',5:'Lasso Regression with PCA'}

# Model evaluation in training dataset
for i, model in enumerate(pipelines):
    print("{} MAE: {}".format(pipe_dict[i], mean_absolute_error(y_train, model.pr
    print("{} MSE: {}".format(pipe_dict[i],  mean_squared_error(y_train,  model.p
    print("{} RMSE: {}".format(pipe_dict[i], np.sqrt(mean_squared_error(y_train,
    print("{} R squared: {}".format(pipe_dict[i],  model.score(X_train,y_train)))
    print("\n")
```

```
Linear Regression MAE: 5.259703491895763e-16
Linear Regression MSE: 4.297552047675707e-31
Linear Regression RMSE: 6.55557171242578e-16
Linear Regression R squared: 1.0


Ridge Regression MAE: 0.13456841850073692
Ridge Regression MSE: 0.031126295445949435
Ridge Regression RMSE: 0.17642645903024137
Ridge Regression R squared: 0.8099437014777006


Lasso Regression MAE: 0.3122800739547922
Lasso Regression MSE: 0.1637740800381702
Lasso Regression RMSE: 0.4046901037067378
Lasso Regression R squared: 0.0


Linear Regression with PCA MAE: 0.2543405661950434
Linear Regression with PCA MSE: 0.10847972525121836
Linear Regression with PCA RMSE: 0.3293626045124406
Linear Regression with PCA R squared: 0.3376257999682525


Ridge Regression with PCA MAE: 0.13456841850073692
Ridge Regression with PCA MSE: 0.031126295445949435
Ridge Regression with PCA RMSE: 0.17642645903024137
Ridge Regression with PCA R squared: 0.8099437014777006


Lasso Regression with PCA MAE: 0.3122800739547922
Lasso Regression with PCA MSE: 0.1637740800381702
Lasso Regression with PCA RMSE: 0.4046901037067378
Lasso Regression with PCA R squared: 0.0
```

In [61]:
```
# 5. MODEL EVALUATION : Model evaluation in testing dataset
for i, model in enumerate(pipelines):
    print("{} Training Accuracy: {}".format(pipe_dict[i], model.score(X_train, y_
```

```
Linear Regression Training Accuracy: 1.0
Ridge Regression Training Accuracy: 0.8099
Lasso Regression Training Accuracy: 0.0
Linear Regression with PCA Training Accuracy: 0.3376
Ridge Regression with PCA Training Accuracy: 0.8099
Lasso Regression with PCA Training Accuracy: 0.0
```

In [60]:
```
# 5. MODEL EVALUATION : Model evaluation in testing dataset
for i, model in enumerate(pipelines):
    print("{} Test Accuracy: {}".format(pipe_dict[i], model.score(X_test, y_test)
```

```
Linear Regression Test Accuracy: 1.0
Ridge Regression Test Accuracy: 0.8077
Lasso Regression Test Accuracy: -0.0001
Linear Regression with PCA Test Accuracy: 0.3199
Ridge Regression with PCA Test Accuracy: 0.8077
Lasso Regression with PCA Test Accuracy: -0.0001
```

In [49]:
```
# Initializer
best_accuracy = 0.0
best_regression_model = 0
best_pipeline = ""

for i, model in enumerate(pipelines):
    if model.score(X_test, y_test) > best_accuracy:
        best_accuracy = model.score(X_test, y_test)
        best_pipeline = model
        best_regression_model = i
print("Regression Model with best accuracy in test dataset: {}".format(pipe_dict[
```

```
Regression Model with best accuracy in test dataset: Linear Regression
```

Classification

In [50]:
```
import warnings
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
import seaborn as sns
warnings.filterwarnings('ignore')
%matplotlib inline
```

In [51]:
```
# Load the dataframe
raw_data = pd.read_csv('wine(1).csv')
```

In [52]:
```
# DATA PREPROCESSING: Remove the duplicated rows
without_duplicate = raw_data.drop_duplicates(keep='first').copy()
without_duplicate.head(10)
```

Out[52]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	s
0	7.39926	0.964904	0.000000	2.200110	0.087996	16.000800	31.99840	0.997460	3.580179	
1	12.29877	0.389961	0.630032	2.300115	0.090995	6.000300	17.99910	1.000300	3.160158	
2	6.39936	0.269973	0.490025	7.300365	0.045998	53.002650	205.98970	0.995500	3.240162	
3	7.39926	0.189981	0.300015	12.800640	0.052997	48.502425	228.98855	0.998500	3.140157	
4	6.19938	0.339966	0.300015	11.100555	0.046998	28.001400	236.98815	0.998000	3.180159	
5	4.99950	0.609939	0.120006	1.300065	0.009000	65.003250	99.99500	0.987301	3.260163	
6	7.29927	0.409959	0.290015	1.800090	0.031998	26.001300	73.99630	0.988791	2.960148	
7	6.59934	0.229977	0.180009	8.500425	0.043998	59.002950	187.99060	0.995480	3.160158	
8	5.99940	0.359964	0.160008	6.300315	0.035998	36.001800	190.99045	0.994101	3.170159	
9	6.59934	0.209979	0.290015	5.350268	0.028999	43.002150	105.99470	0.991021	2.930147	

In [53]:
```
# DATA PREPROCESSING: Remove all NAs
# Drop the whole row that contains the empty cell
# Always assign the new dataframe
without_dup_na = without_duplicate.dropna(how= "any").copy()
```

3. DATA PREPARATION

In [54]:
```
# DATA PREPARATION: Prepare target variable and independent variable
X = np.array(without_dup_na.iloc[:,without_dup_na.columns != 'Type'])
y = np.array(without_dup_na.iloc[:,without_dup_na.columns == 'Type'])

print("Shape of X(PREDICTORS): {}".format(X.shape))
print("Shape of y(TARGET): {}".format(y.shape))
```

```
Shape of X(PREDICTORS): (5295, 12)
Shape of y(TARGET): (5295, 1)
```

In [55]:
```
# DATA PREPARATION: Create training/testing sets
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.3, random_st

print("Number of X_train dataset: ", X_train.shape)
print("Number of y_train dataset: ", y_train.shape)
print("\nNumber of X_test dataset: ", X_test.shape)
print("Number of y_test dataset: ", y_test.shape)
```

```
Number of X_train dataset:  (3706, 12)
Number of y_train dataset:  (3706, 1)

Number of X_test dataset:  (1589, 12)
Number of y_test dataset:  (1589, 1)
```

In [56]:
```
# DATA PREPARATION: Handling Unbalanced dataset
# Perform SMOTE algorithm to handle the unbalanced dataset
# https://www.kaggle.com/qianchao/smote-with-imbalance-data

print("Before OverSampling of training dataset, counts of label 'white': {}".form
print("Before OverSampling of training dataset, counts of label 'red': {} \n".for
```

```
Before OverSampling of training dataset, counts of label 'white': [0]
Before OverSampling of training dataset, counts of label 'red': [0]
```

In [57]:
```
# DATA PREPARATION: Handling Unbalanced dataset by oversampling label red wine

from imblearn.over_sampling import SMOTE
sm = SMOTE(random_state=10)

X_train_res, y_train_res = sm.fit_resample(X_train, y_train.ravel())

print('After OverSampling, the shape of train_X_res: {}'.format(X_train_res.shape
print('After OverSampling, the shape of train_y_res: {} \n'.format(y_train_res.sh

print("After OverSampling, counts of label 'white': {}".format(sum(y_train_res=='
print("After OverSampling, counts of label 'red': {} \n".format(sum(y_train_res==
```

```
After OverSampling, the shape of train_X_res: (5560, 12)
After OverSampling, the shape of train_y_res: (5560,)

After OverSampling, counts of label 'white': 0
After OverSampling, counts of label 'red': 0
```

4. MODEL - PIPELINE CREATION

In [58]:
```
# MODEL - PIPELINE: Libraries needed
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
```

In [59]:
```
# 4. MODEL - PIPELINE CREATION- Create pipelines with or without PCA

##  First create three different pipelines without PCA
# Logistic Regression
pipeline_lr = Pipeline([('scalar1', StandardScaler()),
                        ('lr_classifier', LogisticRegression(random_state=0))])

# Pipeline for decision tree
pipeline_dt = Pipeline([('dt_classifier', DecisionTreeClassifier())])

# Pipeline for random forest classification
pipeline_rf = Pipeline([('rf_classifier', RandomForestClassifier())])

## Now create pipelines with PCA

# Pipeline for logistic regression
pipeline_pca_lr = Pipeline([('scalar1', StandardScaler()),
                        ('pca1', PCA(n_components=2)),
                        ('lr_classifier', LogisticRegression(random_state=0))])

# Pipeline for decision tree
pipeline_pca_dt = Pipeline([('scalar2', StandardScaler()),
                        ('pca2', PCA(n_components=2)),
                        ('dt_classifier', DecisionTreeClassifier())])
# Pipeline for random forest classification
pipeline_pca_rf = Pipeline([('scalar3', StandardScaler()),
                        ('pca3', PCA(n_components=2)),
                        ('rf_classifier', RandomForestClassifier())])

# Create the list of pipelines for classifier
pipelines = [pipeline_lr, pipeline_dt, pipeline_rf, pipeline_pca_lr, pipeline_pca
```

In [60]:
```
# 4. MODEL - PIPELINE CREATION: Fit the pipelines in training dataset
for pipe in pipelines:
    pipe.fit(X_train_res, y_train_res)
```

In [61]:
```
# 5. MODEL EVALUATION: Dictionary of pipelines and classifier type for ease of re
pipe_dict = {0: 'Logistic Regression only', 1: 'Decision Tree classifier only', 2

# Model evaluation in training dataset
for i, model in enumerate(pipelines):
    print("{} Training Accuracy: {}".format(pipe_dict[i], model.score(X_train_res
```

```
Logistic Regression only Training Accuracy: 0.9948
Decision Tree classifier only Training Accuracy: 0.9996
Random Forest Classifier only Training Accuracy: 0.9996
First PCA and Logistic Regression Training Accuracy: 0.9813
First PCA and Decision Tree classifier Training Accuracy: 0.9996
First PCA and Random Forest Classifier Training Accuracy: 0.9996
```

In [62]:
```
# 5. MODEL EVALUATION : Model evaluation in testing dataset
for i, model in enumerate(pipelines):
    print("{} Test Accuracy: {}".format(pipe_dict[i], model.score(X_test, y_test)
```

```
Logistic Regression only Test Accuracy: 0.9906
Decision Tree classifier only Test Accuracy: 0.9729
Random Forest Classifier only Test Accuracy: 0.9924
First PCA and Logistic Regression Test Accuracy: 0.9767
First PCA and Decision Tree classifier Test Accuracy: 0.9685
First PCA and Random Forest Classifier Test Accuracy: 0.9799
```

5. MODEL EVALUATION :Best Accuracy in testing dataset

In [63]:
```


# Initializer
best_accuracy = 0.0
best_classifier = 0
best_pipeline = ""

for i, model in enumerate(pipelines):
    if model.score(X_test, y_test) > best_accuracy:
        best_accuracy = model.score(X_test, y_test)
        best_pipeline = model
        best_classifier = i
print("Classifier with best accuracy in test dataset: {}".format(pipe_dict[best_c
```

```
Classifier with best accuracy in test dataset: Random Forest Classifier only
```

6. PERFORMANCE METRICS OF BEST ALGORITHM

Since Logistic Regression as well as Random forest works really well. Hence we will do detailed performance metrics analysis of these algorithms.

Similarly, performing PCA before applying LR or Random forests algorithms did not enhance the performance of the model

enhance the performance of the model.

In [64]:
```
# 6. PERFORMANCE METRICS OF BEST ALGORITHM: Pipeline for logistic regression as w

# Preprocessing of training data, fit model for Logistic Regression only (Note:NO
pipeline_lr.fit(X_train_res, y_train_res)

y_pred_LR = pipeline_lr.predict(X_test)

# Preprocessing of training data, fit model for Random forest classification only
pipeline_rf.fit(X_train_res, y_train_res)

y_pred_RF = pipeline_rf.predict(X_test)
```

In [65]:
```
# 6. DETAILED PERFORMANCE METRICS OF BEST ALGORITHM: LOGISTIC REGRESSION WITHOUT

from sklearn.metrics import confusion_matrix
confusion = confusion_matrix(y_test, y_pred_LR)
print('Confusion Matrix\n')
print(confusion)

#importing accuracy_score, precision_score, recall_score, f1_score
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_sco
print('\nAccuracy score: {:.2f}\n'.format(accuracy_score(y_test, y_pred_LR).round

print('Micro Precision: {:.2f}'.format(precision_score(y_test, y_pred_LR, average
print('Micro Recall: {:.2f}'.format(recall_score(y_test, y_pred_LR, average='micr
print('Micro F1-score: {:.2f}\n'.format(f1_score(y_test, y_pred_LR, average='micr

print('Macro Precision: {:.2f}'.format(precision_score(y_test, y_pred_LR, average
print('Macro Recall: {:.2f}'.format(recall_score(y_test, y_pred_LR, average='macr
print('Macro F1-score: {:.2f}\n'.format(f1_score(y_test, y_pred_LR, average='macr

print('Weighted Precision: {:.2f}'.format(precision_score(y_test, y_pred_LR, aver
print('Weighted Recall: {:.2f}'.format(recall_score(y_test, y_pred_LR, average='w
print('Weighted F1-score: {:.2f}'.format(f1_score(y_test, y_pred_LR, average='wei

from sklearn.metrics import classification_report
print('\nClassification Report\n')
print(classification_report(y_test, y_pred_LR, target_names=['Red wine', 'White w
```

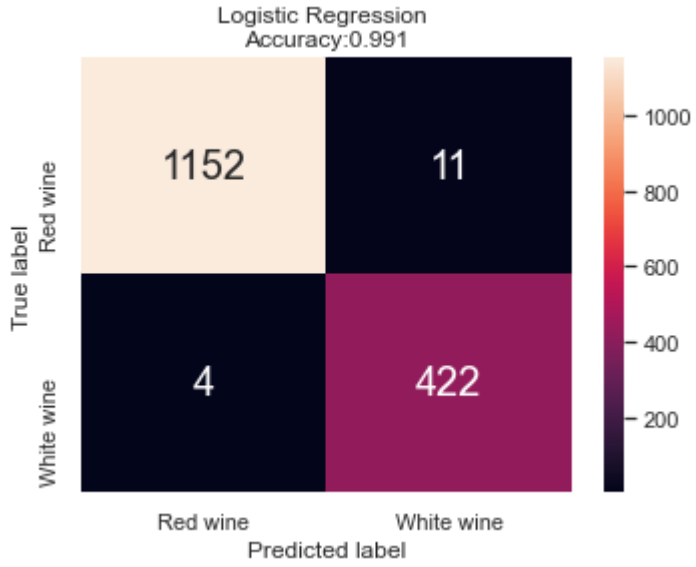
```
Confusion Matrix

[[1152   11]
 [   4  422]]

Accuracy score: 0.99

Micro Precision: 0.99
Micro Recall: 0.99
Micro F1-score: 0.99

Macro Precision: 0.99
Macro Recall: 0.99
Macro F1-score: 0.99

Weighted Precision: 0.99
Weighted Recall: 0.99
Weighted F1-score: 0.99

Classification Report

              precision    recall  f1-score   support

    Red wine       1.00      0.99      0.99      1163
  White wine       0.97      0.99      0.98       426

    accuracy                           0.99      1589
   macro avg       0.99      0.99      0.99      1589
weighted avg       0.99      0.99      0.99      1589
```

In [66]:
```
# 6. DETAILED PERFORMANCE METRICS OF BEST ALGORITHM: RANDOM FOREST CLASSIFICATION
confusion = confusion_matrix(y_test, y_pred_RF)
print('Confusion Matrix\n')
print(confusion)

#importing accuracy_score, precision_score, recall_score, f1_score
print('\nAccuracy score: {:.2f}\n'.format(accuracy_score(y_test, y_pred_RF).round

print('Micro Precision: {:.2f}'.format(precision_score(y_test, y_pred_RF, average
print('Micro Recall: {:.2f}'.format(recall_score(y_test, y_pred_RF, average='micr
print('Micro F1-score: {:.2f}\n'.format(f1_score(y_test, y_pred_RF, average='micr

print('Macro Precision: {:.2f}'.format(precision_score(y_test, y_pred_RF, average
print('Macro Recall: {:.2f}'.format(recall_score(y_test, y_pred_RF, average='macr
print('Macro F1-score: {:.2f}\n'.format(f1_score(y_test, y_pred_RF, average='macr

print('Weighted Precision: {:.2f}'.format(precision_score(y_test, y_pred_RF, aver
print('Weighted Recall: {:.2f}'.format(recall_score(y_test, y_pred_RF, average='w
print('Weighted F1-score: {:.2f}'.format(f1_score(y_test, y_pred_RF, average='wei

from sklearn.metrics import classification_report
print('\nClassification Report\n')
print(classification_report(y_test, y_pred_RF, target_names=['Red wine', 'White w
```

```
Confusion Matrix

[[1160    3]
 [   5  421]]

Accuracy score: 0.99

Micro Precision: 0.99
Micro Recall: 0.99
Micro F1-score: 0.99

Macro Precision: 0.99
Macro Recall: 0.99
Macro F1-score: 0.99

Weighted Precision: 0.99
Weighted Recall: 0.99
Weighted F1-score: 0.99

Classification Report

              precision    recall  f1-score   support

    Red wine       1.00      1.00      1.00      1163
  White wine       0.99      0.99      0.99       426

    accuracy                           0.99      1589
   macro avg       0.99      0.99      0.99      1589
weighted avg       0.99      0.99      0.99      1589
```

7. FINAL EVALUATION

In [67]:
```
# 7. FINAL EVALUATION: Both Logistic regression and Random forest works equally w

#  Plot the confusion matrix

# Creates a confusion matrix
cm = confusion_matrix(y_test, y_pred_LR)

# Transform to df for easier plotting
cm_df = pd.DataFrame(cm,
                     index = ['Red wine','White wine'],
                     columns = ['Red wine','White wine'])

plt.figure(figsize=(5.5,4))
sns.heatmap(cm_df, annot=True, fmt='d',annot_kws={"size": 20})
plt.title('Logistic Regression \nAccuracy:{0:.3f}'.format(accuracy_score(y_test,
plt.ylabel('True label')
plt.xlabel('Predicted label')
plt.show()
```



In [68]:
```
# Final conclusion:
# Since the data looks very ideal hence, the Logistic Regression as well as Randc
```