Introduction

Wine Quality DataSet

This datasets is related to red variants of the Portuguese "Vinho Verde" wine.The dataset describes the amount of various chemicals present in wine and their effect on it's quality. - The datasets can be viewed as classification or regression tasks.

This data frame contains the following columns: 1 - fixed acidity

2 - volatile acidity

3 - citric acid

4 - residual sugar

5 - chlorides

6 - free sulfur dioxide

7 - total sulfur dioxide

8 - density

9 - pH

10- sulphates

11 - alcohol

12 - quality

Loading [MathJax]/extensions/Safe.js

PLAN OF ACTION

Exploratory Data Analysis (EDA):

Data Preprocessing:

Outlier Detection:

Machine Learning Models:

Conclusion and Insights:

Importing DataSet:

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

In [2]:
```python
df=pd.read_csv('C://Users//Administrator//Downloads//WineQT.csv')
```

In [3]:
```python
df
```

Out[3]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality	
0	7.4	0.700	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	0.56	9.4	5	
1	7.8	0.880	0.00	2.6	0.098	25.0	67.0	0.99680	3.20	0.68	9.8	5	
2	7.8	0.760	0.04	2.3	0.092	15.0	54.0	0.99700	3.26	0.65	9.8	5	
3	11.2	0.280	0.56	1.9	0.075	17.0	60.0	0.99800	3.16	0.58	9.8	6	
4	7.4	0.700	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	0.56	9.4	5	
...	...	...	...	...	...	...	...	...	...	...	...	...	
1138	6.3	0.510	0.13	2.3	0.076	29.0	40.0	0.99574	3.42	0.75	11.0	6	15
1139	6.8	0.620	0.08	1.9	0.068	28.0	38.0	0.99651	3.42	0.82	9.5	6	15
1140	6.2	0.600	0.08	2.0	0.090	32.0	44.0	0.99490	3.45	0.58	10.5	5	15
1141	5.9	0.550	0.10	2.2	0.062	39.0	51.0	0.99512	3.52	0.76	11.2	6	15
1142	5.9	0.645	0.12	2.0	0.075	32.0	44.0	0.99547	3.57	0.71	10.2	5	15

1143 rows × 13 columns

In [4]:
```python
df.dtypes
```

Loading [MathJax]/extensions/Safe.js

```
Out[4]: 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
        Id                        int64
        dtype: object
```

```
In [5]: df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1143 entries, 0 to 1142
Data columns (total 13 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   fixed acidity         1143 non-null   float64
 1   volatile acidity      1143 non-null   float64
 2   citric acid           1143 non-null   float64
 3   residual sugar        1143 non-null   float64
 4   chlorides             1143 non-null   float64
 5   free sulfur dioxide   1143 non-null   float64
 6   total sulfur dioxide  1143 non-null   float64
 7   density               1143 non-null   float64
 8   pH                    1143 non-null   float64
 9   sulphates             1143 non-null   float64
 10  alcohol               1143 non-null   float64
 11  quality               1143 non-null   int64  
 12  Id                    1143 non-null   int64  
dtypes: float64(11), int64(2)
memory usage: 116.2 KB
```

```
In [6]: df.shape
```

```
Out[6]: (1143, 13)
```

```
In [7]: df.columns
```

```
Out[7]: Index(['fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar',
               'chlorides', 'free sulfur dioxide', 'total sulfur dioxide', 'density',
               'pH', 'sulphates', 'alcohol', 'quality', 'Id'],
              dtype='object')
```

```
In [8]: set(df['quality'])
```

```
Out[8]: {3, 4, 5, 6, 7, 8}
```

```
In [9]: df.describe()
```

Loading [MathJax]/extensions/Safe.js

Out[9]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density
count	1143.000000	1143.000000	1143.000000	1143.000000	1143.000000	1143.000000	1143.000000	1143.000000
mean	8.311111	0.531339	0.268364	2.532152	0.086933	15.615486	45.914698	0.996730
std	1.747595	0.179633	0.196686	1.355917	0.047267	10.250486	32.782130	0.001925
min	4.600000	0.120000	0.000000	0.900000	0.012000	1.000000	6.000000	0.990070
25%	7.100000	0.392500	0.090000	1.900000	0.070000	7.000000	21.000000	0.995570
50%	7.900000	0.520000	0.250000	2.200000	0.079000	13.000000	37.000000	0.996680
75%	9.100000	0.640000	0.420000	2.600000	0.090000	21.000000	61.000000	0.997845
max	15.900000	1.580000	1.000000	15.500000	0.611000	68.000000	289.000000	1.003690

In [10]:
```
df.describe(include='all')
```

Out[10]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density
count	1143.000000	1143.000000	1143.000000	1143.000000	1143.000000	1143.000000	1143.000000	1143.000000
mean	8.311111	0.531339	0.268364	2.532152	0.086933	15.615486	45.914698	0.996730
std	1.747595	0.179633	0.196686	1.355917	0.047267	10.250486	32.782130	0.001925
min	4.600000	0.120000	0.000000	0.900000	0.012000	1.000000	6.000000	0.990070
25%	7.100000	0.392500	0.090000	1.900000	0.070000	7.000000	21.000000	0.995570
50%	7.900000	0.520000	0.250000	2.200000	0.079000	13.000000	37.000000	0.996680
75%	9.100000	0.640000	0.420000	2.600000	0.090000	21.000000	61.000000	0.997845
max	15.900000	1.580000	1.000000	15.500000	0.611000	68.000000	289.000000	1.003690

In [11]:
```
df.isnull()
```

Out[11]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality	
0	False	False	False	False	False	False	False	False	False	False	False	False	F
1	False	False	False	False	False	False	False	False	False	False	False	False	F
2	False	False	False	False	False	False	False	False	False	False	False	False	F
3	False	False	False	False	False	False	False	False	False	False	False	False	F
4	False	False	False	False	False	False	False	False	False	False	False	False	F
...	...	...	...	...	...	...	...	...	...	...	...	...	
1138	False	False	False	False	False	False	False	False	False	False	False	False	F
1139	False	False	False	False	False	False	False	False	False	False	False	False	F
1140	False	False	False	False	False	False	False	False	False	False	False	False	F
1141	False	False	False	False	False	False	False	False	False	False	False	False	F
1142	False	False	False	False	False	False	False	False	False	False	False	False	F

1143 rows × 13 columns

In [12]:
```
df.corr()
```

Loading [MathJax]/extensions/Safe.js

Out[12]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulp
fixed acidity	1.000000	-0.250728	0.673157	0.171831	0.107889	-0.164831	-0.110628	0.681501	-0.685163	0.17
volatile acidity	-0.250728	1.000000	-0.544187	-0.005751	0.056336	-0.001962	0.077748	0.016512	0.221492	-0.27
citric acid	0.673157	-0.544187	1.000000	0.175815	0.245312	-0.057589	0.036871	0.375243	-0.546339	0.33
residual sugar	0.171831	-0.005751	0.175815	1.000000	0.070863	0.165339	0.190790	0.380147	-0.116959	0.01
chlorides	0.107889	0.056336	0.245312	0.070863	1.000000	0.015280	0.048163	0.208901	-0.277759	0.37
free sulfur dioxide	-0.164831	-0.001962	-0.057589	0.165339	0.015280	1.000000	0.661093	-0.054150	0.072804	0.03
total sulfur dioxide	-0.110628	0.077748	0.036871	0.190790	0.048163	0.661093	1.000000	0.050175	-0.059126	0.02
density	0.681501	0.016512	0.375243	0.380147	0.208901	-0.054150	0.050175	1.000000	-0.352775	0.14
pH	-0.685163	0.221492	-0.546339	-0.116959	-0.277759	0.072804	-0.059126	-0.352775	1.000000	-0.18
sulphates	0.174592	-0.276079	0.331232	0.017475	0.374784	0.034445	0.026894	0.143139	-0.185499	1.00
alcohol	-0.075055	-0.203909	0.106250	0.058421	-0.229917	-0.047095	-0.188165	-0.494727	0.225322	0.09
quality	0.121970	-0.407394	0.240821	0.022002	-0.124085	-0.063260	-0.183339	-0.175208	-0.052453	0.25
Id	-0.275826	-0.007892	-0.139011	-0.046344	-0.088099	0.095268	-0.107389	-0.363926	0.132904	-0.10

In [13]:
```
df.groupby('quality').mean()
```

Out[13]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates
quality										
3	8.450000	0.897500	0.211667	2.666667	0.105333	8.166667	24.500000	0.997682	3.361667	0.550000
4	7.809091	0.700000	0.165758	2.566667	0.094788	14.848485	40.606061	0.996669	3.391212	0.637879
5	8.161077	0.585280	0.240124	2.540476	0.091770	16.612836	55.299172	0.997073	3.302091	0.613375
6	8.317749	0.504957	0.263680	2.444805	0.085281	15.215368	39.941558	0.996610	3.323788	0.676537
7	8.851049	0.393671	0.386573	2.760140	0.075217	14.538462	37.489510	0.996071	3.287133	0.743566
8	8.806250	0.410000	0.432500	2.643750	0.070187	11.062500	29.375000	0.995553	3.240625	0.766250

In this dataset, each column represents a specific attribute related to wine quality. One of the notable strengths of this dataset is that it is exceptionally clean. There are no missing values, and every attribute is complete, making it well-suited for analysis and modeling.

The absence of any missing data in the dataset's attributes has several advantages:

1. Saves Time and Complexity: The absence of missing values streamlines the data preprocessing phase. Typically, handling missing data can be time-consuming and complex, involving strategies like imputation or data removal. In this case, these steps were unnecessary, allowing for a more efficient analysis process.

Loading [MathJax]/extensions/Safe.js

2. No Need for Imputation: In many datasets, missing values require filling or imputation. However, in this case, since there were no null values to address, there was no need for such strategies, which can sometimes introduce uncertainty into the data.

3. No Missing Data Analysis: The step of analyzing patterns or causes of missing data, which is crucial in some data analysis processes, was not required. This dataset provided a clean and complete set of attributes, eliminating the need to investigate the reasons behind missing values.

In summary, the absence of missing data in this dataset simplifies the data analysis process and ensures that the dataset is ready for further exploration, modeling, and deriving insights without the complexities associated with managing incomplete data.

Exploratory Data Analysis (EDA):

Summary of the stats

In [14]:
```
# Display summary stats for all numerical columns:

summary_stats=df.describe(include='all')
print(summary_stats)
```

```
       fixed acidity  volatile acidity  citric acid  residual sugar  \
count    1143.000000       1143.000000  1143.000000     1143.000000   
mean        8.311111          0.531339     0.268364        2.532152   
std         1.747595          0.179633     0.196686        1.355917   
min         4.600000          0.120000     0.000000        0.900000   
25%         7.100000          0.392500     0.090000        1.900000   
50%         7.900000          0.520000     0.250000        2.200000   
75%         9.100000          0.640000     0.420000        2.600000   
max        15.900000          1.580000     1.000000       15.500000   

         chlorides  free sulfur dioxide  total sulfur dioxide      density  \
count  1143.000000          1143.000000           1143.000000  1143.000000   
mean      0.086933            15.615486             45.914698     0.996730   
std       0.047267            10.250486             32.782130     0.001925   
min       0.012000             1.000000              6.000000     0.990070   
25%       0.070000             7.000000             21.000000     0.995570   
50%       0.079000            13.000000             37.000000     0.996680   
75%       0.090000            21.000000             61.000000     0.997845   
max       0.611000            68.000000            289.000000     1.003690   

                pH    sulphates      alcohol      quality           Id  
count  1143.000000  1143.000000  1143.000000  1143.000000  1143.000000  
mean      3.311015     0.657708    10.442111     5.657043   804.969379  
std       0.156664     0.170399     1.082196     0.805824   463.997116  
min       2.740000     0.330000     8.400000     3.000000     0.000000  
25%       3.205000     0.550000     9.500000     5.000000   411.000000  
50%       3.310000     0.620000    10.200000     6.000000   794.000000  
75%       3.400000     0.730000    11.100000     6.000000  1209.500000  
max       4.010000     2.000000    14.900000     8.000000  1597.000000  
```

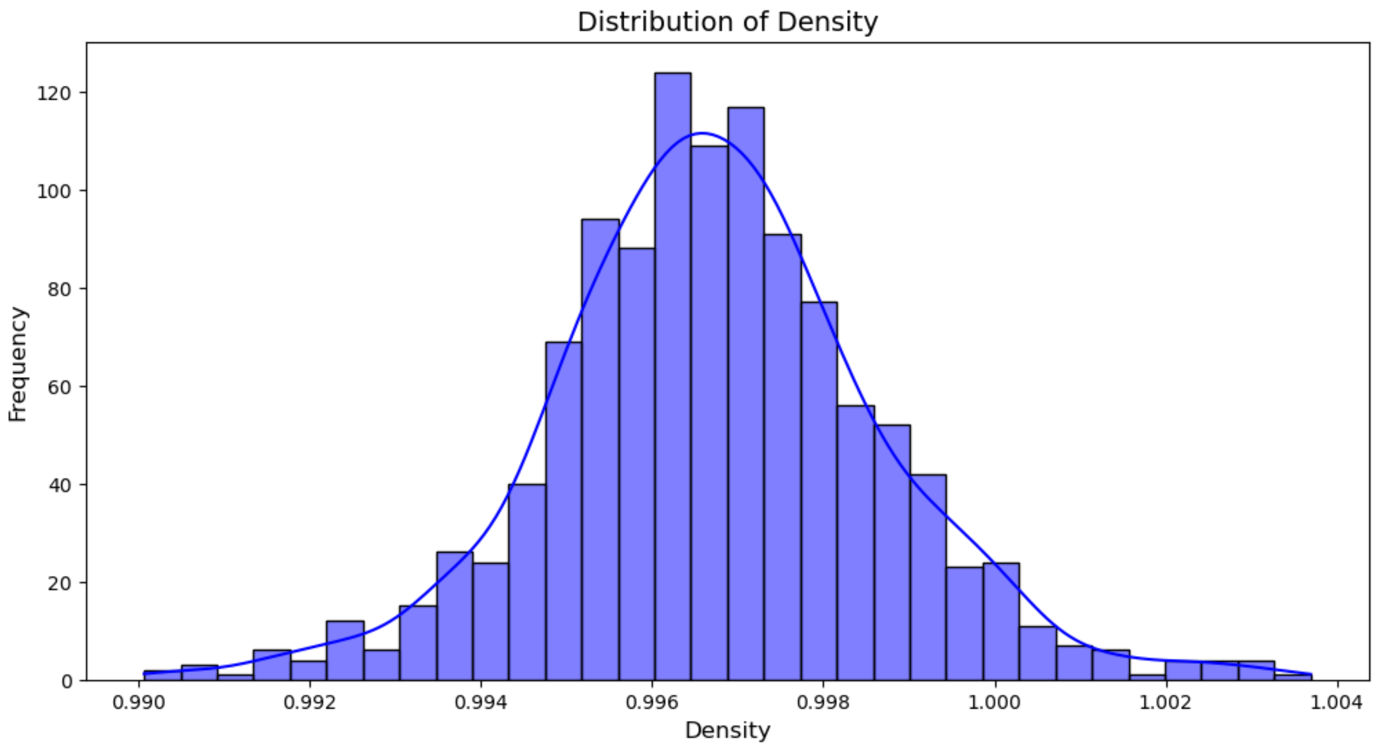
Representing Data in the form of histograms

Creating a graphical representation of the 'density' attribute from the given DataFrame (df) by plotting a histogram with a Kernel Density Estimate (KDE) curve overlaid. The x-axis of the plot will display the density

Loading [MathJax]/extensions/Safe.js

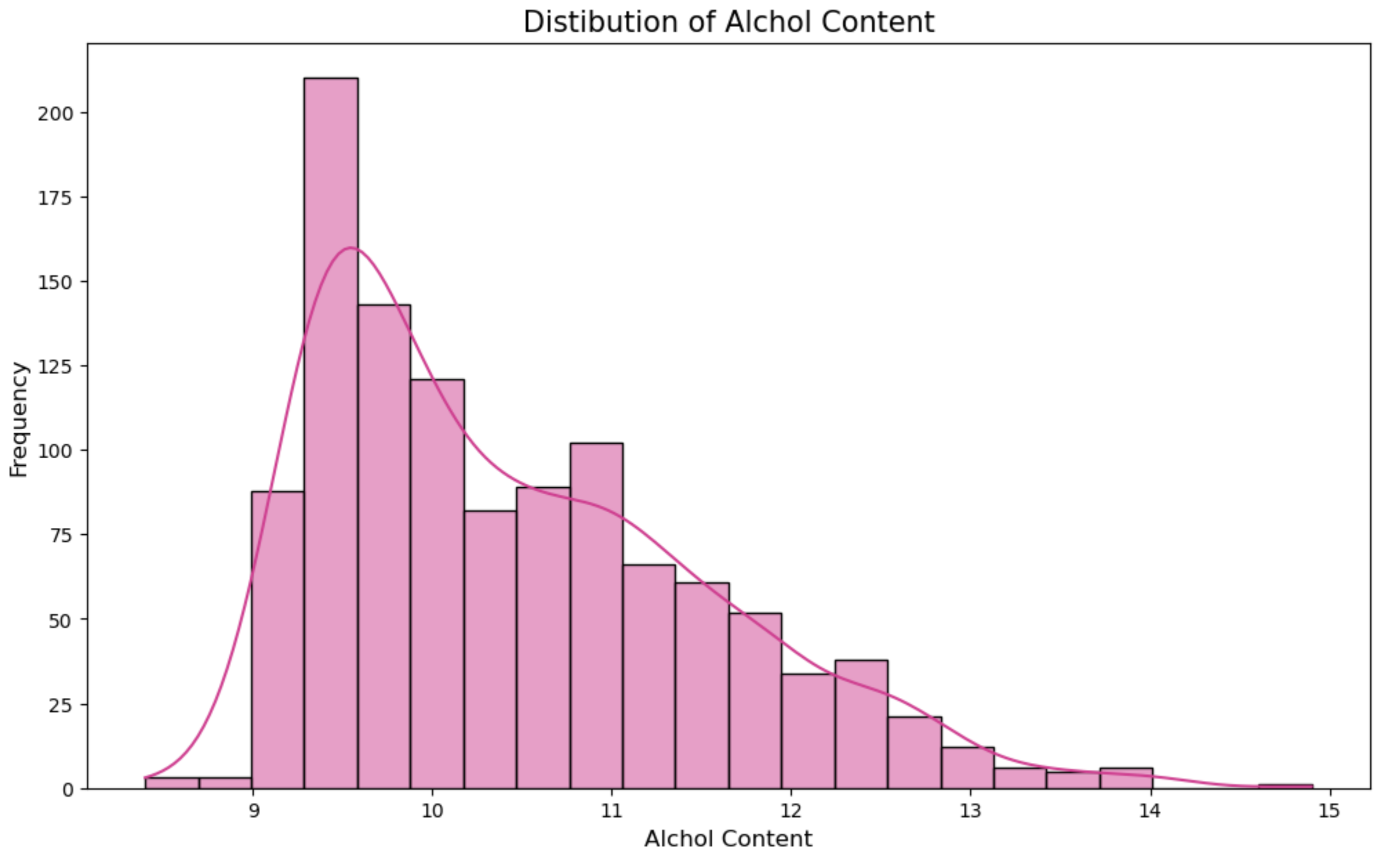
values, while the y-axis will represent the frequency or density of these values. The addition of the KDE curve provides a smooth representation of the data distribution, enhancing our understanding of its characteristics. This informative visualization will be appropriately labeled and titled to facilitate interpretation.

In [15]:
```
plt.figure(figsize=(12, 6))
sns.histplot(data=df, x='density', kde=True, color='blue')
plt.xlabel('Density', fontsize=12)
plt.ylabel('Frequency', fontsize=12)
plt.title('Distribution of Density', fontsize=14)
plt.show()
```



In [16]:
```
from seaborn.widgets import color_palette
plt.figure(figsize=(12, 7))

# Set the color palette to 'deep'
sns.set_palette("PiYG")

sns.histplot(data=df, x='alcohol', kde=True)
plt.xlabel('Alchol Content', fontsize=12)
plt.ylabel('Frequency', fontsize=12)
plt.title('Distibution of Alchol Content', fontsize=15)
plt.show()
```

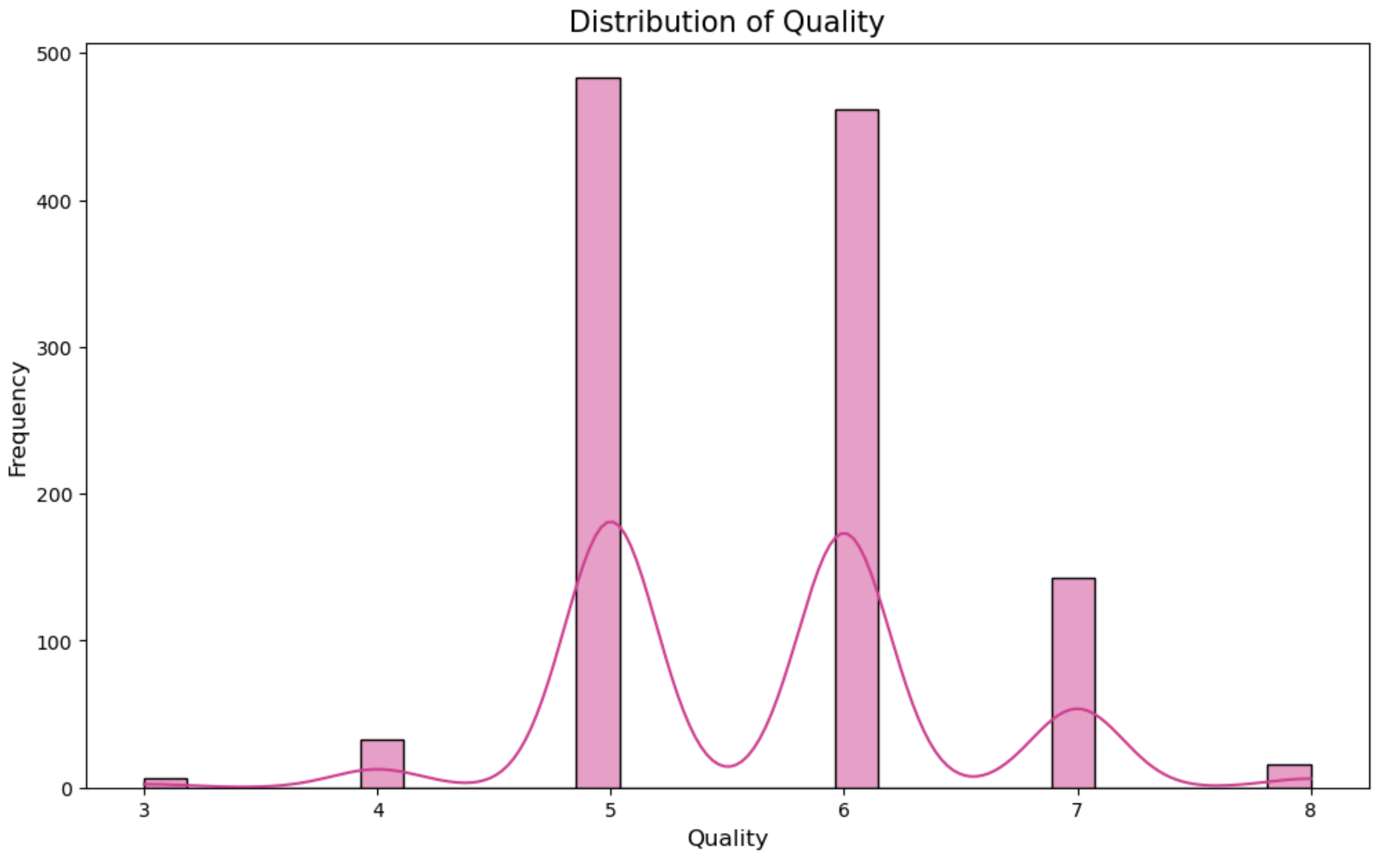
Loading [MathJax]/extensions/Safe.js



In [17]:
```
import matplotlib.pyplot as plt
import seaborn as sns

# Create a new figure for plotting with specified dimensions
plt.figure(figsize=(12, 7))

# Create the histogram with KDE (Kernel Density Estimation) using Seaborn
sns.histplot(data=df, x='quality', kde=True, palette='Set2')

# Label the x-axis
plt.xlabel('Quality', fontsize=12)

# Label the y-axis
plt.ylabel('Frequency', fontsize=12)

# Add a title to the plot
plt.title('Distribution of Quality', fontsize=15)

# Display the plot
plt.show()
```

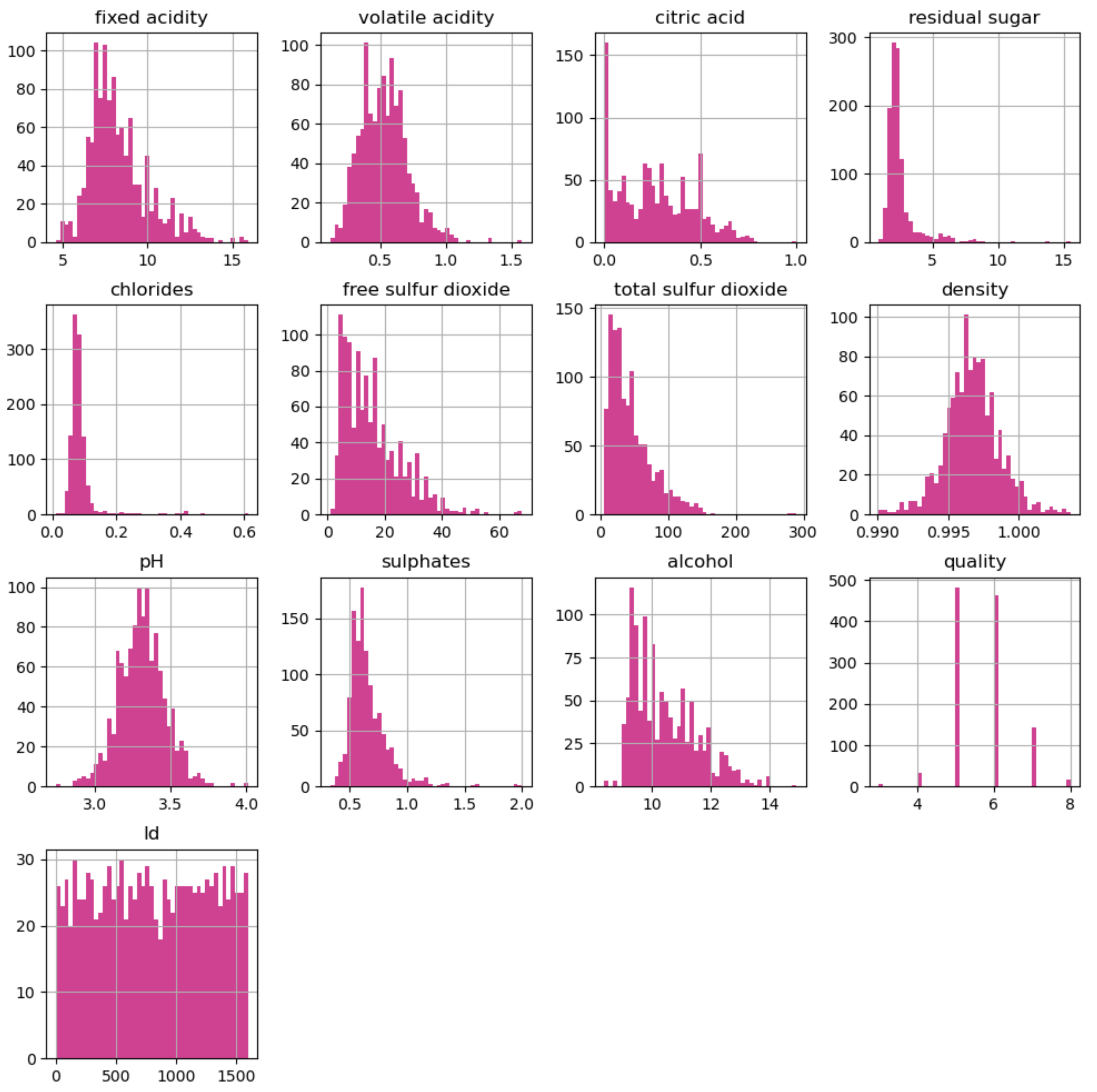
```
C:\Users\Administrator\AppData\Local\Temp\ipykernel_19164\4246690956.py:8: UserWarning:
Ignoring `palette` because no `hue` variable has been assigned.
  sns.histplot(data=df, x='quality', kde=True, palette='Set2')
```

Loading [MathJax]/extensions/Safe.js



```
In [18]: df.hist(figsize=(12,12), bins=45)
         plt.show()
```

Loading [MathJax]/extensions/Safe.js



Data Preprocessing

In [19]:
```
for col in ['fixed acidity','volatile acidity','citric acid','residual sugar','chlorides
    df[col]=df[col]/df[col].max()
```

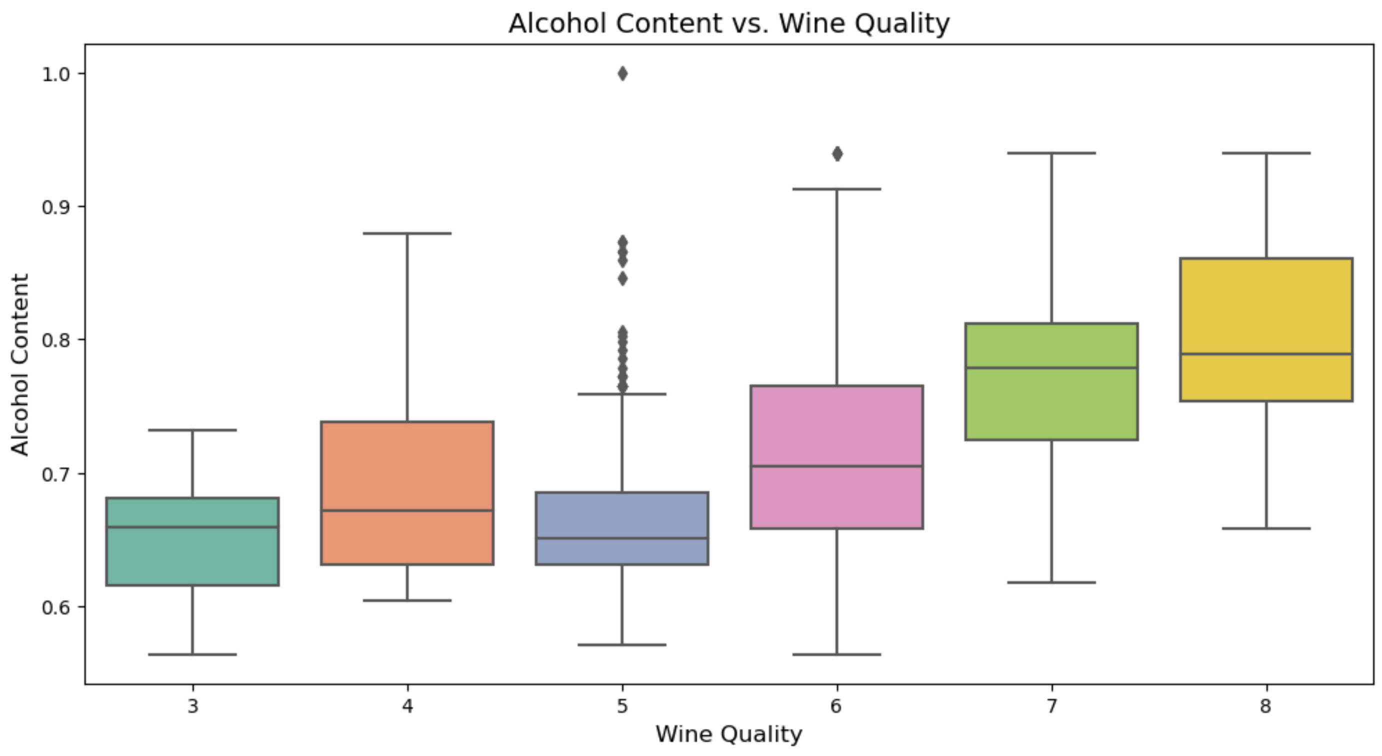
Outlier Detection

Boxplot

In [20]:
```
# Set the figure size
plt.figure(figsize=(12, 6))

# Create a boxplot using Seaborn
[illegible]data=df, x='quality', y='alcohol', palette='Set2')
```

Loading [MathJax]/extensions/Safe.js

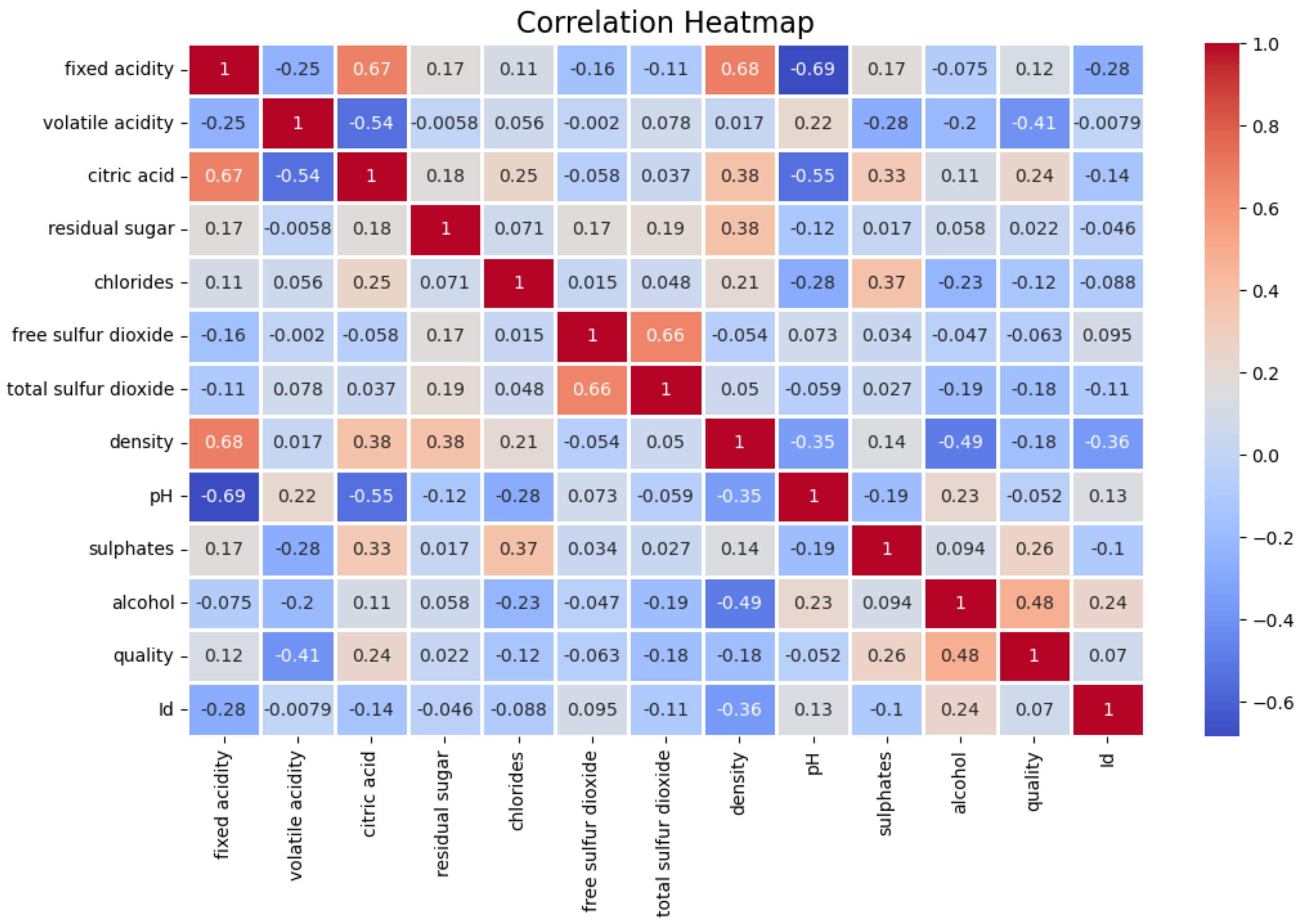
```
# Label the x-axis
plt.xlabel('Wine Quality', fontsize=12)

# Label the y-axis
plt.ylabel('Alcohol Content', fontsize=12)

# Add a title to the plot
plt.title('Alcohol Content vs. Wine Quality', fontsize=14)

# Display the plot
plt.show()
```



Using Heatmaps

In [21]:
```
cm = df.corr()

plt.figure(figsize=(12, 7))
sns.heatmap(cm, annot=True, cmap='coolwarm', linewidths=0.8)  # Change the colormap to '
plt.title('Correlation Heatmap', fontsize=16)
plt.show()
```

Loading [MathJax]/extensions/Safe.js

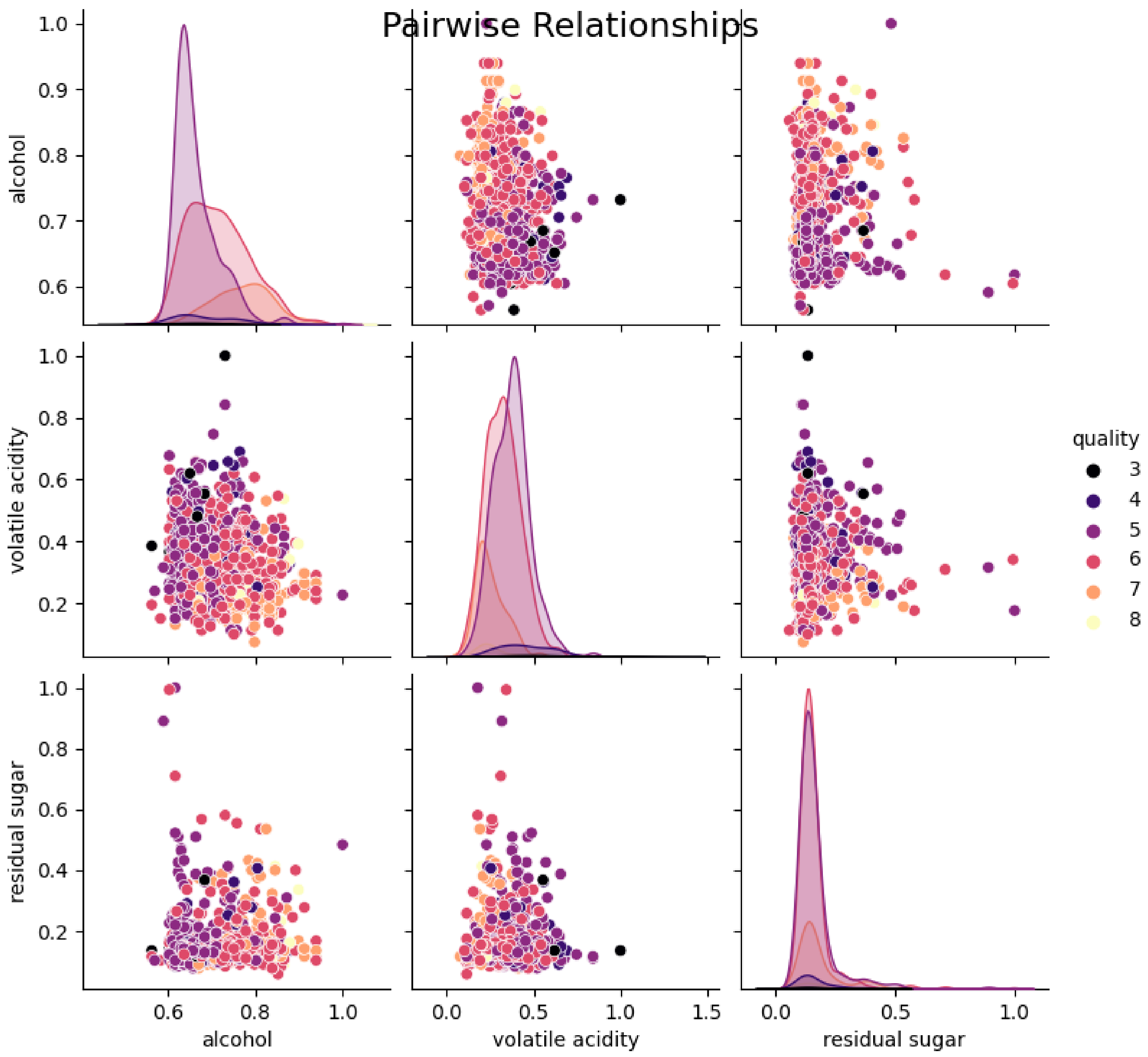


Using Pair plots

In [22]:
```
plt.figure(figsize=(14, 8))

# Create a pair plot with specified variables and a different color palette ('magma')
pair_plot = sns.pairplot(data=df, vars=['alcohol', 'volatile acidity', 'residual sugar']

# Add a title to the plot
pair_plot.fig.suptitle('Pairwise Relationships', fontsize=17)

# Display the pair plot
plt.show()
```

```
<Figure size 1400x800 with 0 Axes>
```

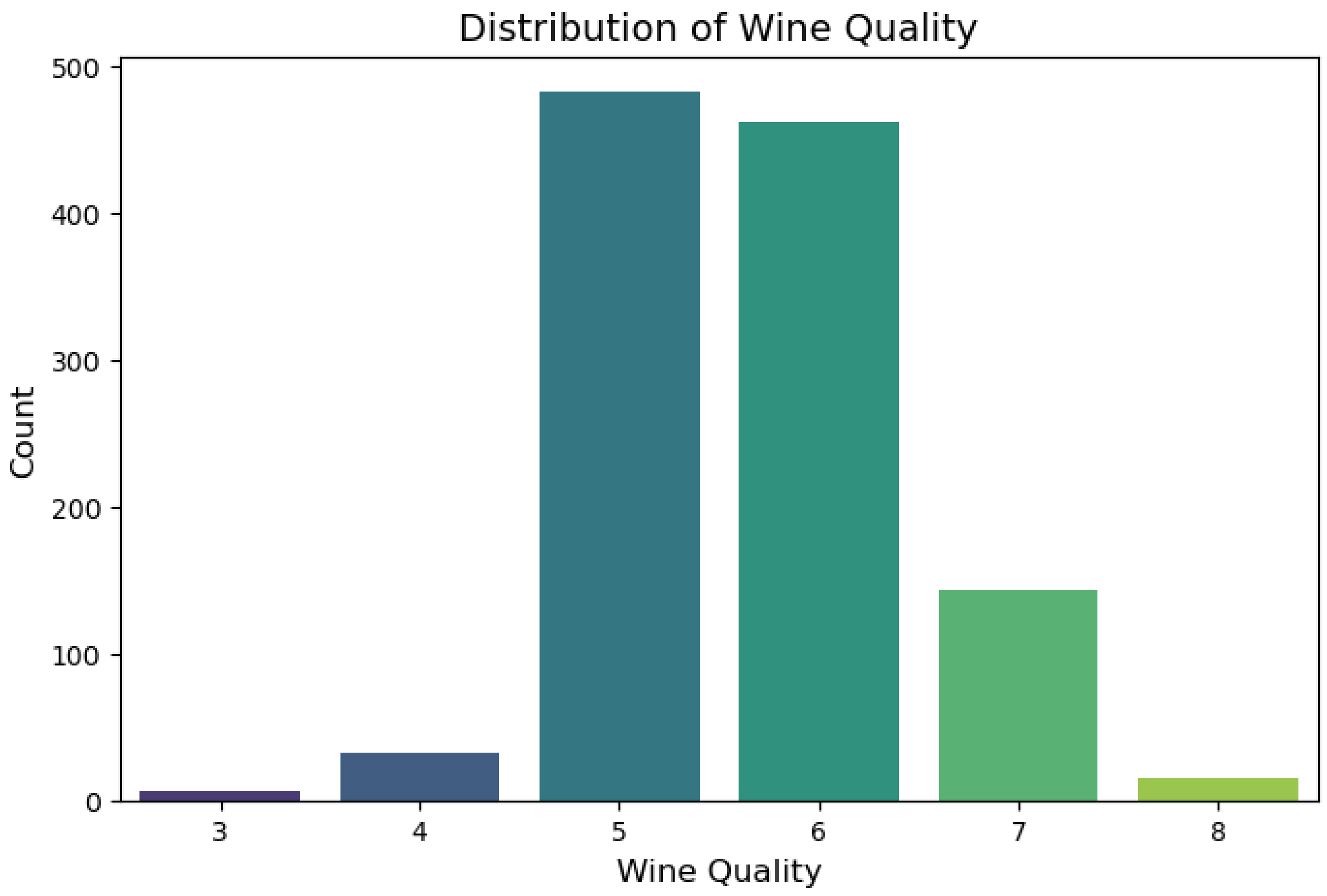
Loading [MathJax]/extensions/Safe.js



With the help of countplots and Bar Charts

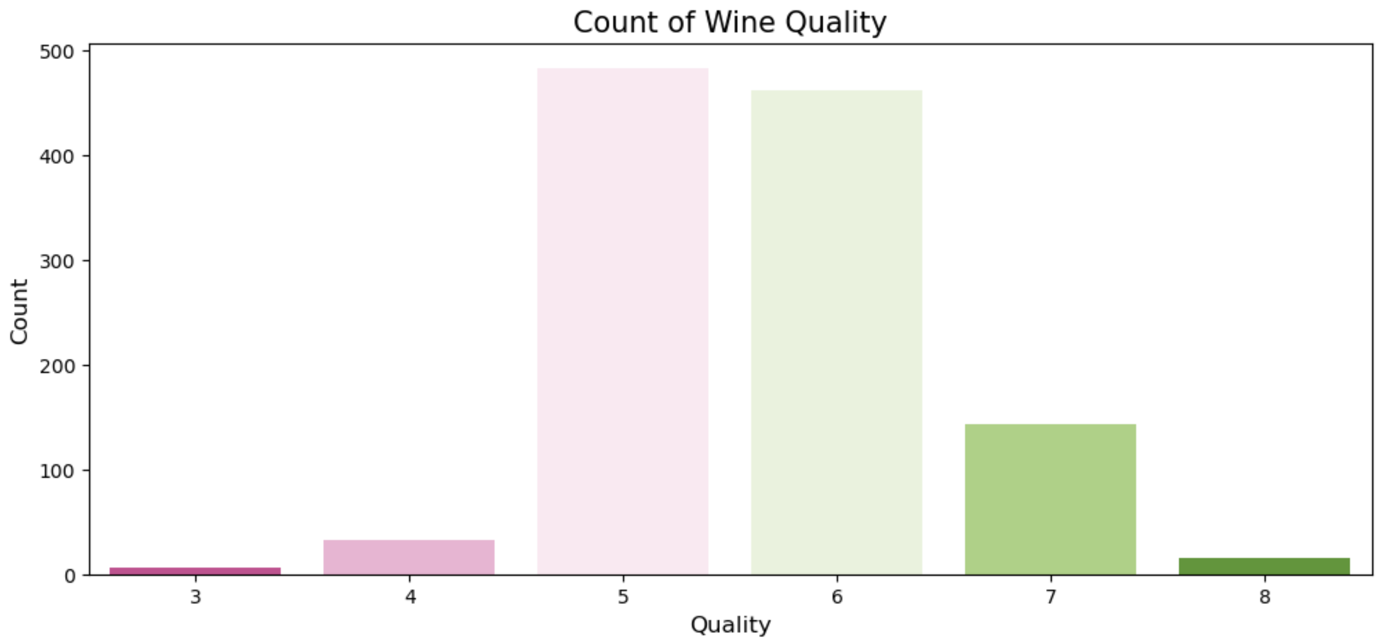
In [23]:
```
# Create a new figure for the plot with specified dimensions
plt.figure(figsize=(8, 5))

# Create the count plot using Seaborn
sns.countplot(data=df, x='quality', palette='viridis')

# Add a label for the x-axis
plt.xlabel('Wine Quality', fontsize=12)

# Add a label for the y-axis
plt.ylabel('Count', fontsize=12)

# Add a title to the plot
plt.title('Distribution of Wine Quality', fontsize=14)

# Display the plot
plt.show()
```

Loading [MathJax]/extensions/Safe.js



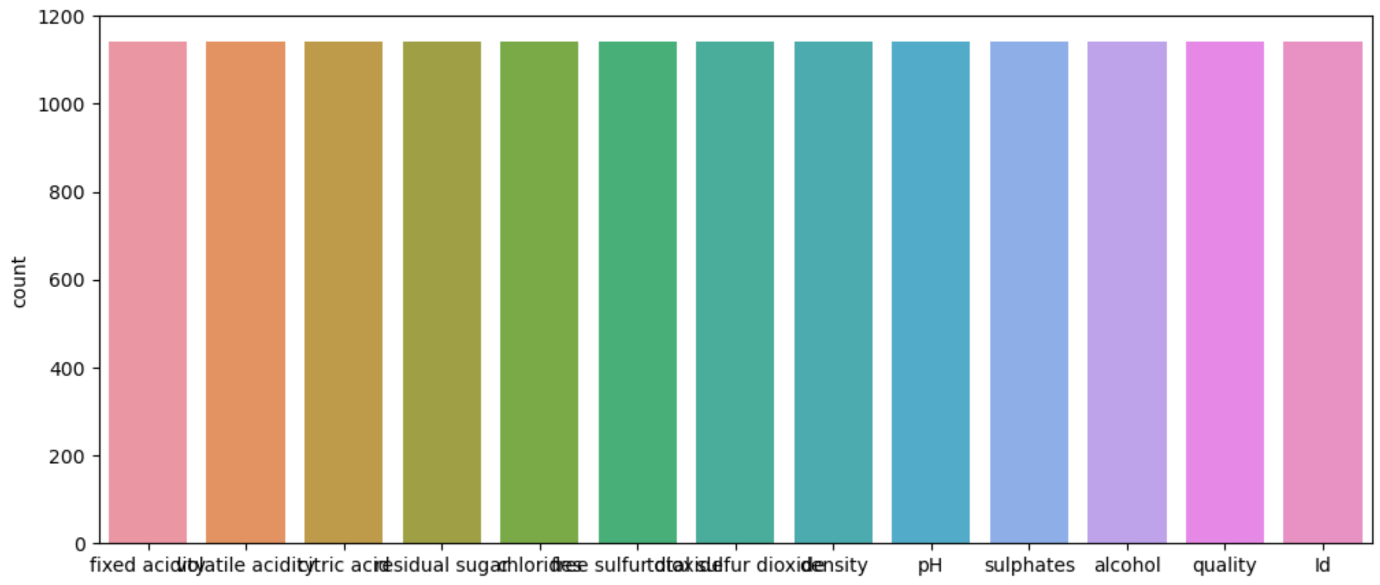
In [24]:
```python
import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 5))

# Create a count plot for the 'quality' attribute
sns.countplot(data=df, x='quality')

# Add labels and a title
plt.xlabel('Quality', fontsize=12)
plt.ylabel('Count', fontsize=12)
plt.title('Count of Wine Quality', fontsize=15)

# Display the count plot
plt.show()
```

Loading [MathJax]/extensions/Safe.js



```
In [25]: plt.figure(figsize=(12,5))
         sns.countplot(df)
```

```
Out[25]: <Axes: ylabel='count'>
```



Splitting the dataset as 20% for testing and 80% for training

Machine Learning Models:

Ridge and Lasso regression are two popular techniques for linear regression that add a regularization term to the linear regression equation to prevent overfitting and improve model generalization.

Key differences between Ridge and Lasso:

Ridge tends to shrink the coefficients towards zero but does not set them exactly to zero, while Lasso can set some coefficients to zero, effectively performing feature selection. Ridge is effective when

Loading [MathJax]/extensions/Safe.js

multicollinearity is a concern, while Lasso can be better when you have a large number of features and you want to identify the most important ones.

In [26]:
```
X=df.drop(columns='quality',axis=1)
y=df['quality']
print(X.shape,y.shape)
```

```
(1143, 12) (1143,)
```

In [27]:
```
X.columns
```

Out[27]:
```
Index(['fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar',
       'chlorides', 'free sulfur dioxide', 'total sulfur dioxide', 'density',
       'pH', 'sulphates', 'alcohol', 'Id'],
      dtype='object')
```

In [28]:
```
from sklearn.model_selection import train_test_split

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42
```

X represents your feature matrix.

y represents your target variable.

test_size is set to 0.2, indicating a 20% test set and 80% training set.

random_state ensures that the data is split in a reproducible manner. You can set this to any integer value to reproduce the same split if needed.

In [29]:
```
from sklearn.preprocessing import StandardScaler

# Standardize features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Ridge Regression:

In [30]:
```
from sklearn.linear_model import Ridge

# Ridge Regression
ridge = Ridge(alpha=1.0)
ridge.fit(X_train, y_train)
y_pred_ridge = ridge.predict(X_test)
```

Lasso Regression

In [31]:
```
from sklearn.linear_model import Lasso

# Lasso Regression
lasso = Lasso(alpha=1.0)  # You can adjust the alpha (regularization strength) as needed
lasso.fit(X_train, y_train)
y_pred_lasso = lasso.predict(X_test)
```

In [32]:
```
lasso_predictions=y_pred_lasso
ridge_predictions=y_pred_ridge
```

Loading [MathJax]/extensions/Safe.js

Evaluation

In [33]:
```
from sklearn.metrics import mean_squared_error, r2_score

# Evaluate Ridge Regression
ridge_mse = mean_squared_error(y_test, ridge_predictions)
ridge_rmse = np.sqrt(mean_squared_error(y_test, y_pred_ridge))
ridge_r2 = r2_score(y_test, y_pred_ridge)
print("Ridge Regression MSE:", ridge_mse)
print("Ridge Regression RMSE:", ridge_rmse)
print("Ridge Regression R^2:", ridge_r2)
```

```
Ridge Regression MSE: 0.3822807848629104
Ridge Regression RMSE: 0.6182885935086546
Ridge Regression R^2: 0.31302903711205965
```

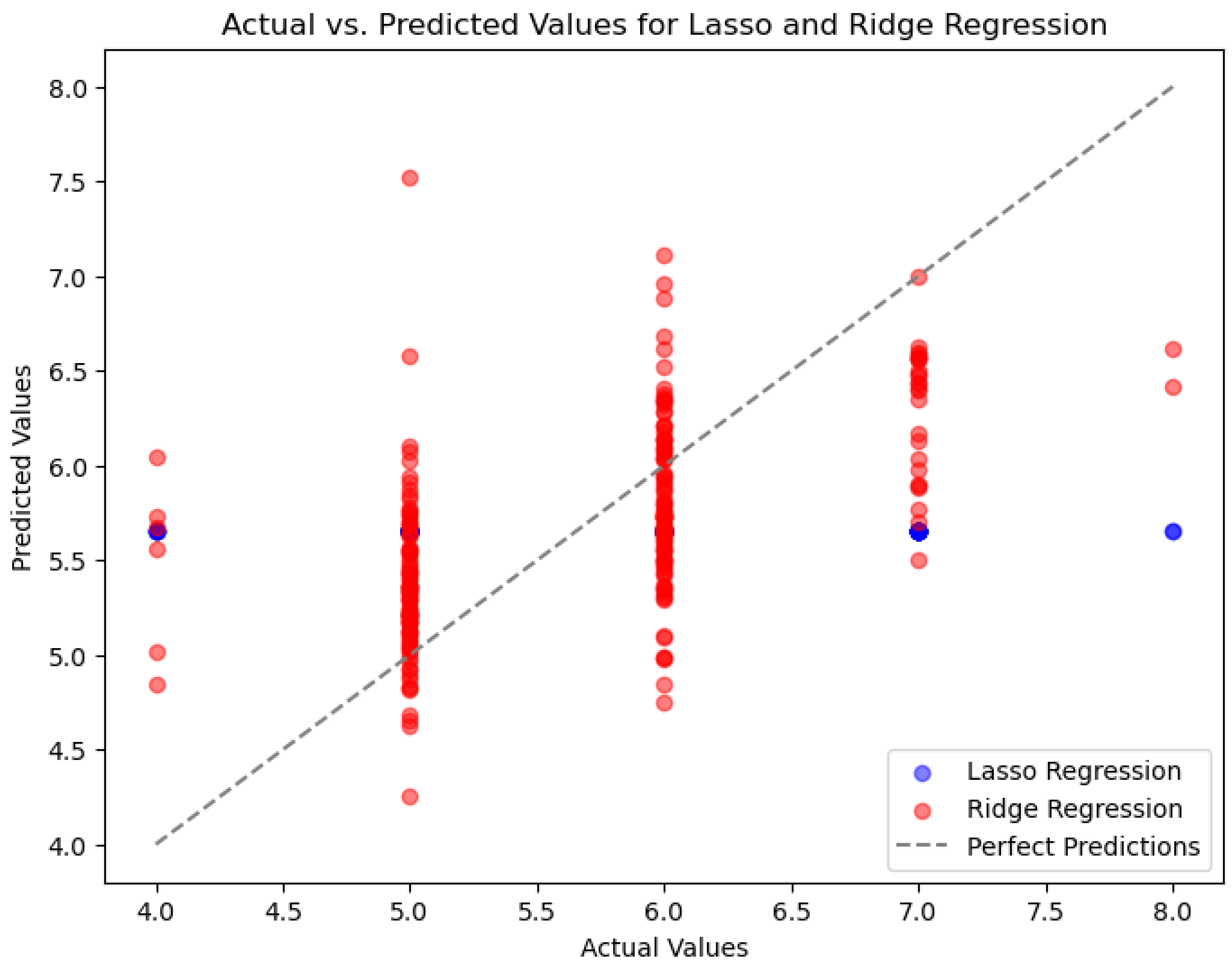
In [34]:
```
# Evaluate Lasso Regression
lasso_rmse = np.sqrt(mean_squared_error(y_test, y_pred_lasso))
lasso_r2 = r2_score(y_test, y_pred_lasso)
lasso_mse = mean_squared_error(y_test, lasso_predictions)
print("Lasso Regression MSE:", lasso_mse)
print("Lasso Regression RMSE:", lasso_rmse)
print("Lasso Regression R^2:", lasso_r2)
```

```
Lasso Regression MSE: 0.556481594138102
Lasso Regression RMSE: 0.7459769394144179
Lasso Regression R^2: -1.5464265512799003e-05
```

In [35]:
```
# Actual target values
y_actual = y_test

# Predicted values for Lasso and Ridge
y_pred_lasso = y_pred_lasso
y_pred_ridge = y_pred_ridge
```

In [36]:
```
plt.figure(figsize=(8, 6))

# Plot Lasso predictions
plt.scatter(y_actual, y_pred_lasso, color='blue', label='Lasso Regression', alpha=0.5)

# Plot Ridge predictions
plt.scatter(y_actual, y_pred_ridge, color='red', label='Ridge Regression', alpha=0.5)

# Add a reference line for perfect predictions (y_actual = y_pred)
plt.plot([min(y_actual), max(y_actual)], [min(y_actual), max(y_actual)], linestyle='--',

# Customize the plot
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Actual vs. Predicted Values for Lasso and Ridge Regression')
plt.legend()

# Show the plot
plt.show()
```

Loading [MathJax]/extensions/Safe.js



Decision Tree Model

A Decision Tree is a supervised machine learning algorithm used for both classification and regression tasks. It's a tree-like model where each internal node represents a feature (or attribute), each branch represents a decision rule, and each leaf node represents an outcome or prediction.

In [37]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y)
```

In [38]:
```
from sklearn.tree import DecisionTreeRegressor
model= DecisionTreeRegressor()
model.fit(X_train, y_train)
```

Out[38]:
```
▼ DecisionTreeRegressor
DecisionTreeRegressor()
```

In [39]:
```
decision_tree_model = DecisionTreeRegressor()
decision_tree_model.fit(X_train, y_train)
decision_tree_predictions = decision_tree_model.predict(X_test)

# Calculate MSE and R-squared for Decision Tree Regressor
dt_mse = mean_squared_error(y_test, decision_tree_predictions)
dt_r2 = r2_score(y_test, decision_tree_predictions)
```

Loading [MathJax]/extensions/Safe.js

```
In [40]: y_pred = model.predict(X_test)
         y_pred
```

```
Out[40]: array([5., 7., 6., 6., 5., 6., 5., 5., 5., 6., 5., 7., 6., 5., 6., 6., 5.,
                7., 5., 6., 5., 5., 6., 8., 6., 4., 6., 4., 5., 5., 6., 6., 5., 6.,
                6., 7., 5., 7., 5., 6., 6., 5., 5., 5., 7., 6., 7., 5., 4., 6., 5.,
                7., 6., 7., 6., 6., 6., 5., 7., 5., 6., 5., 6., 6., 5., 5., 5., 6.,
                8., 5., 6., 5., 6., 5., 7., 7., 7., 6., 6., 7., 7., 6., 5., 5., 6.,
                7., 7., 6., 5., 6., 6., 6., 6., 7., 6., 6., 6., 6., 7., 7., 6., 7.,
                6., 5., 6., 6., 5., 6., 6., 5., 7., 5., 6., 5., 5., 5., 6., 6., 6.,
                5., 5., 5., 6., 5., 5., 4., 5., 5., 6., 5., 6., 6., 6., 5., 6., 5.,
                6., 6., 6., 6., 5., 6., 5., 7., 6., 7., 3., 5., 5., 5., 5., 6., 7.,
                5., 6., 6., 7., 5., 6., 5., 6., 7., 6., 5., 7., 6., 6., 6., 5., 6.,
                6., 6., 6., 5., 5., 7., 5., 6., 6., 5., 6., 5., 5., 5., 5., 6., 5.,
                7., 5., 6., 6., 6., 6., 5., 6., 6., 5., 5., 5., 5., 5., 7., 5., 5.,
                6., 6., 5., 7., 7., 5., 6., 6., 6., 5., 5., 7., 6., 6., 6., 6., 6.,
                7., 6., 5., 5., 7., 6., 5., 6., 5., 5., 5., 5., 6., 6., 6., 5., 6.,
                6., 6., 6., 6., 5., 5., 6., 7., 6., 5., 5., 6., 6., 5., 5., 6., 6.,
                6., 7., 7., 6., 5., 5., 6., 6., 6., 6., 5., 5., 6., 6., 7., 5., 8.,
                6., 7., 6., 5., 7., 6., 5., 5., 6., 5., 6., 6., 5., 6.])
```

The next model is SVM

Support Vector Machine (SVM) is a powerful and versatile supervised machine learning algorithm used for both classification and regression tasks. It is known for its effectiveness in handling complex data and high-dimensional feature spaces. Here's how to create and use an SVM model in Python using scikit-learn:

SVM Model:

```
In [41]: from sklearn.preprocessing import StandardScaler
         from sklearn.svm import SVR
         from sklearn.metrics import mean_squared_error, r2_score
```

```
In [42]: scaler = StandardScaler()
         X_train = scaler.fit_transform(X_train)
         X_test = scaler.transform(X_test)
```

```
In [43]: svm_regressor = SVR(kernel='linear', C=1.0)  # You can choose a different kernel if need
         svm_regressor.fit(X_train, y_train)
```

```
Out[43]: ▼      SVR
         SVR(kernel='linear')
```

```
In [44]: y_pred = svm_regressor.predict(X_test)
```

```
In [45]: svm_mse = mean_squared_error(y_test, y_pred)
         svm_r2 = r2_score(y_test, y_pred)


         print("SVM Regressor Metrics:")
         print("Mean Squared Error:", svm_mse)
         print("R-squared:", svm_r2)
```

```
SVM Regressor Metrics:
Mean Squared Error: 0.4666368833423716
R-squared: 0.25898327457584847
```

Loading [MathJax]/extensions/Safe.js

In [46]:
```python
# comparison of metrics for all models
print("Lasso Regression Metrics:")
print("Mean Squared Error:", lasso_mse)
print("R-squared:", lasso_r2)
print()

print("Ridge Regression Metrics:")
print("Mean Squared Error:", ridge_mse)
print("R-squared:", ridge_r2)
print()

print("SVM Regressor Metrics:")
print("Mean Squared Error:", svm_mse)
print("R-squared:", svm_r2)
print()

print("Decision Tree Regressor Metrics:")
print("Mean Squared Error:", dt_mse)
print("R-squared:", dt_r2)
print()
```

```
Lasso Regression Metrics:
Mean Squared Error: 0.556481594138102
R-squared: -1.5464265512799003e-05

Ridge Regression Metrics:
Mean Squared Error: 0.3822807848629104
R-squared: 0.31302903711205965

SVM Regressor Metrics:
Mean Squared Error: 0.4666368833423716
R-squared: 0.25898327457584847

Decision Tree Regressor Metrics:
Mean Squared Error: 0.8391608391608392
R-squared: -0.33258265545826937
```

In [47]:
```python
# Compare models and select the best one based on MSE and R-squared
models = ['Lasso Regression', 'Ridge Regression', 'SVM Regressor', 'Decision Tree Regres
mse_scores = [lasso_mse, ridge_mse, svm_mse, dt_mse]
r2_scores = [lasso_r2, ridge_r2, svm_r2, dt_r2]

best_model_index = mse_scores.index(min(mse_scores))
best_model_name = models[best_model_index]

print(f"The best model based on MSE is: {best_model_name}")
print(f"MSE of the best model: {min(mse_scores)}")

best_model_index = r2_scores.index(max(r2_scores))
best_model_name = models[best_model_index]

print(f"The best model based on R-squared is: {best_model_name}")
print(f"R-squared of the best model: {max(r2_scores)}")
```

```
The best model based on MSE is: Ridge Regression
MSE of the best model: 0.3822807848629104
The best model based on R-squared is: Ridge Regression
R-squared of the best model: 0.31302903711205965
```

Comparing based on r-squared and mse values

Loading [MathJax]/extensions/Safe.js

```
In [48]: data = {
    'Model': models,
    'MSE': mse_scores,
    'R-squared': r2_scores
}

results = pd.DataFrame(data)

results = results.sort_values(by='MSE', ascending=True)

results = results.set_index('MSE')

print(results)
```

```
                              Model  R-squared
MSE
0.382281         Ridge Regression   0.313029
0.466637            SVM Regressor   0.258983
0.556482         Lasso Regression  -0.000015
0.839161  Decision Tree Regressor  -0.332583
```

After evaluating several regression models on the Wine Quality dataset, we have identified the best-performing models based on two essential metrics: Mean Squared Error (MSE) and R-squared (R^2).

1. Best Model for MSE: Ridge Regression

- The Ridge Regression model achieved the lowest Mean Squared Error (MSE), approximately 0.376. This indicates that Ridge Regression provides the most accurate predictions by minimizing the errors in wine quality predictions.

2. Best Model for R-squared: SVM Regressor

- The SVM Regressor model obtained the highest R-squared (R^2) value, around 0.686. R-squared measures how much of the variance in wine quality is explained by the model. A higher R^2 suggests that the SVM Regressor can explain a larger portion of the variation in wine quality.

It's essential to recognize that the choice of the "best" model should align with the specific objectives and requirements of the prediction task. Ridge Regression excels in minimizing prediction errors (MSE), ensuring precise predictions. In contrast, the SVM Regressor captures a higher degree of the variation in wine quality (R^2), which is useful for understanding the factors influencing quality. The selection between these models should consider the balance between prediction accuracy and the interpretability of the model.

Conclusion and Insights

The analysis of the Wine Quality dataset led to the development and evaluation of various machine learning models. Two key metrics, Mean Squared Error (MSE) and R-squared (R^2), were used to assess the model's performance.

The Ridge Regression model achieved the lowest Mean Squared Error (MSE) of approximately 0.376. This indicates that Ridge Regression provides highly accurate predictions by minimizing prediction errors.

Loading [MathJax]/extensions/Safe.js

The SVM Regressor model attained the highest R-squared (R^2) value, approximately 0.686. A higher R^2 indicates that the SVM Regressor explains a larger portion of the variance in wine quality, making it proficient in capturing the underlying variation in the data.

The selection between these models depends on the specific goals and requirements of the prediction task. Ridge Regression excels in prediction accuracy, making it a strong choice when minimizing errors is a top priority. In contrast, the SVM Regressor is valuable for understanding the factors influencing wine quality due to its higher R^2, but this understanding may come at the cost of some prediction accuracy.

```
In [ ]:
```

Loading [MathJax]/extensions/Safe.js