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

In [2]:
```
quality = pd.read_csv("Wine_Quality.csv")
```

In [3]:
```
quality.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

```
So in here we have 11 independent features and 1 dependent feature(quality).
```

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

Out[6]:
```
(1599, 12)
```

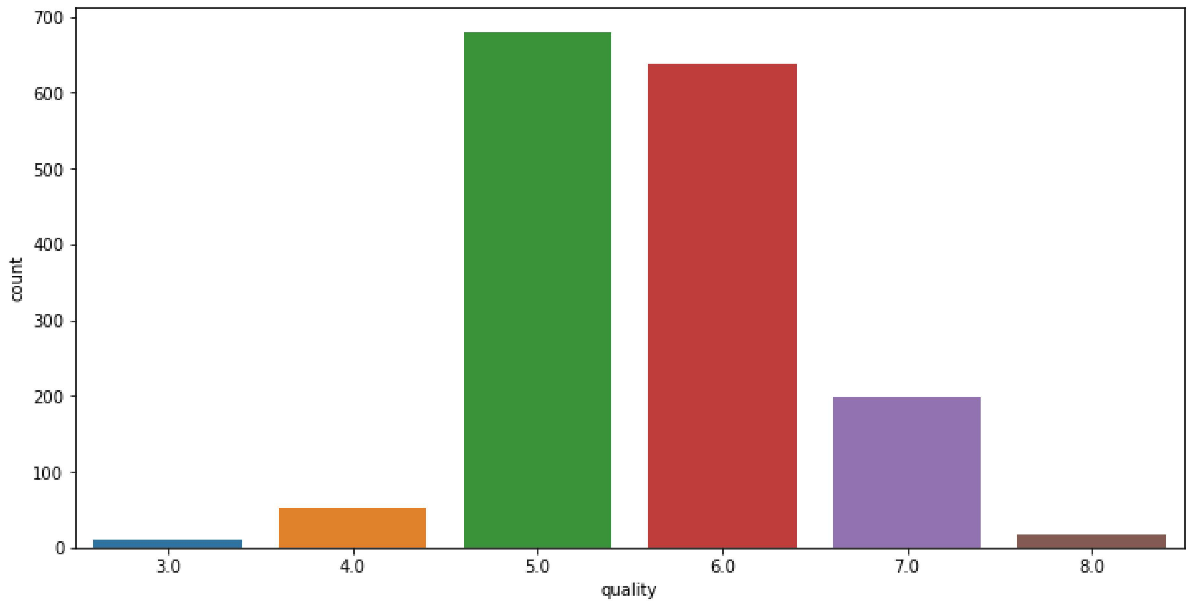
In [7]:
```
quality.dropna(inplace = True)
```

In [8]:
```
quality.describe()
```

Out[8]:

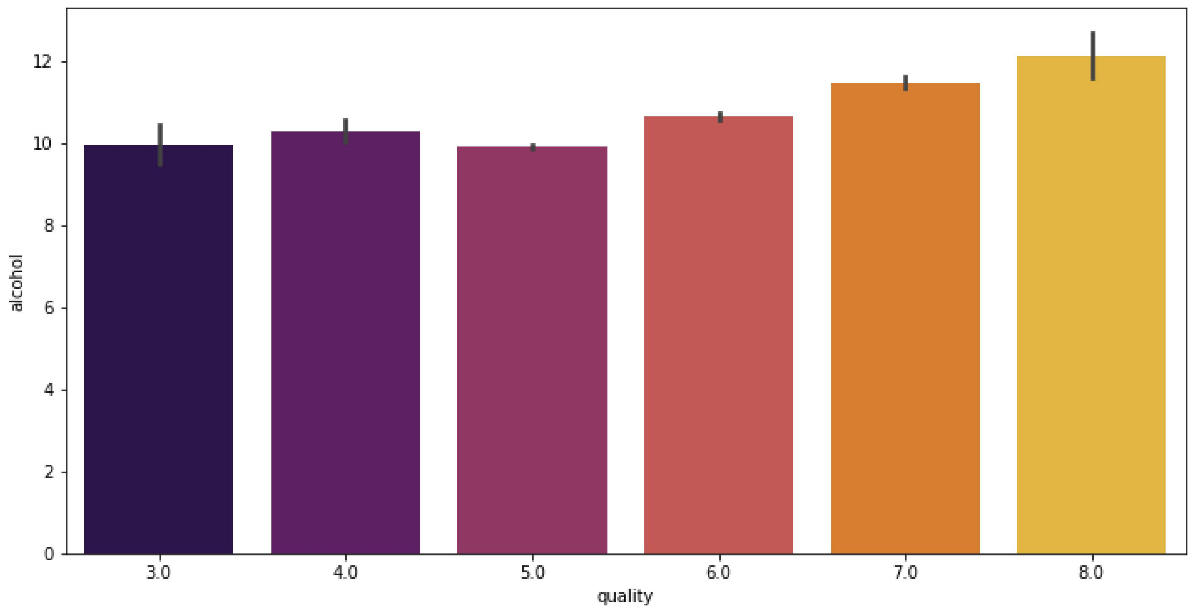
	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfu dioxid
count	1596.000000	1596.000000	1596.000000	1596.000000	1596.000000	1596.000000	1596.0000
mean	8.321366	0.527666	0.271128	2.536936	0.087487	15.882206	46.43107
std	1.742121	0.179154	0.194847	1.408341	0.047107	10.467380	32.89307
min	4.600000	0.120000	0.000000	0.900000	0.012000	1.000000	6.0000
25%	7.100000	0.390000	0.090000	1.900000	0.070000	7.000000	22.0000
50%	7.900000	0.520000	0.260000	2.200000	0.079000	14.000000	38.0000
75%	9.200000	0.640000	0.420000	2.600000	0.090000	21.000000	62.0000
max	15.900000	1.580000	1.000000	15.500000	0.611000	72.000000	289.0000

In [9]:
```
plt.figure(figsize = (12,6))
sns.countplot(x=quality['quality'])
plt.show()
```

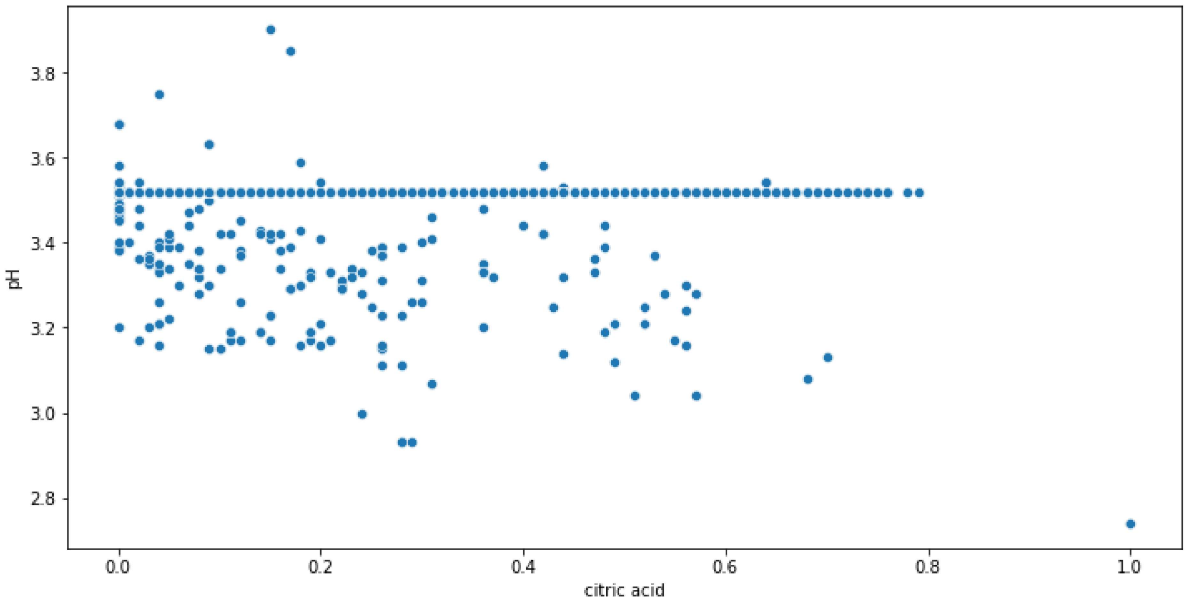


```
Overall, this code creates a countplot to visualize the distribution of
values in the 'quality' column
of the DataFrame. It helps in understanding the frequency or occurrence of
different quality levels
and provides insights into the data distribution.
```

In [10]:
```
plt.figure(figsize = (12,6))
sns.barplot(x='quality', y = 'alcohol', data = quality, palette = 'inferno')
plt.show()
```

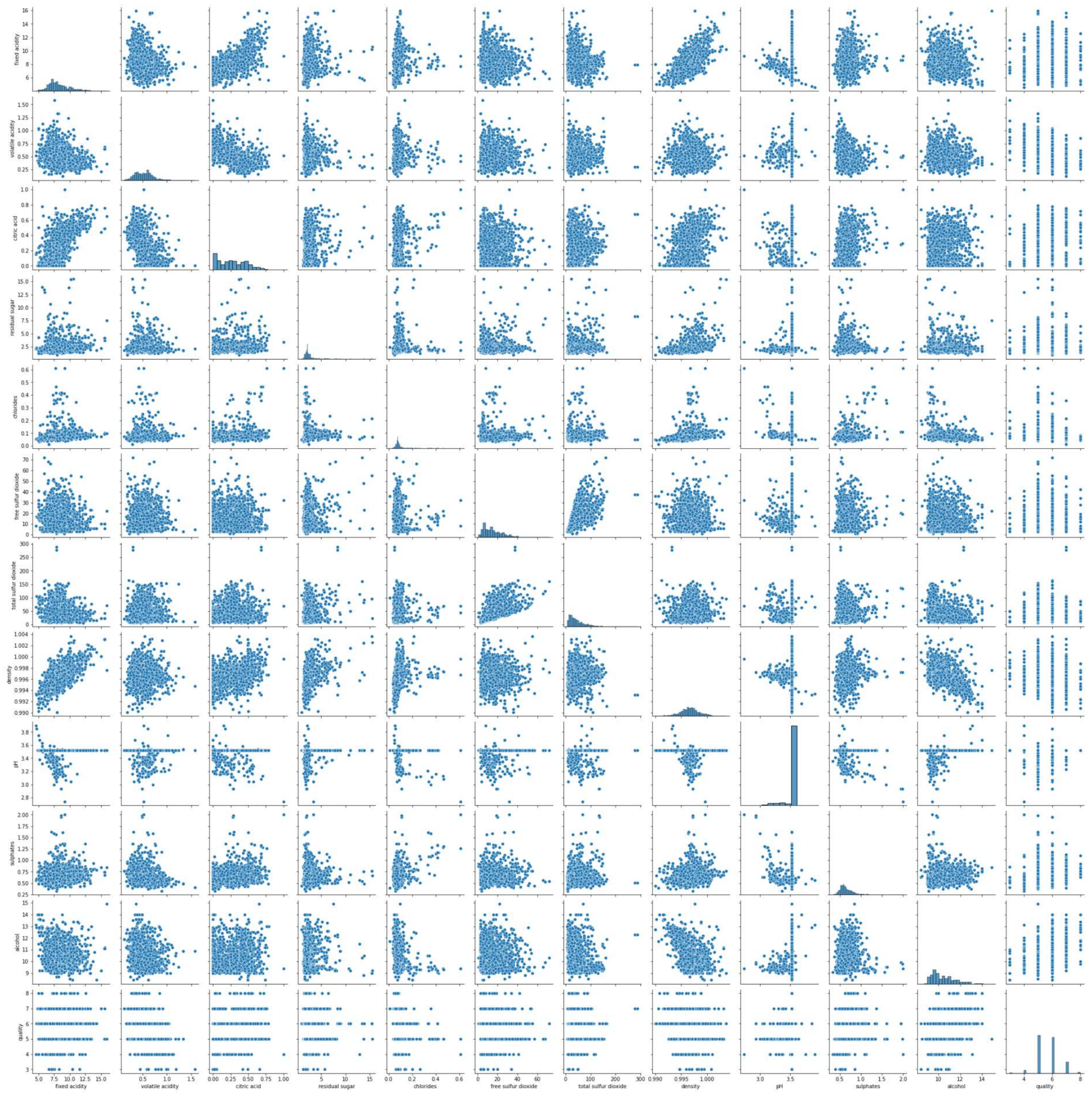


```
In [11]: plt.figure(figsize = (12,6))
         sns.scatterplot(x='citric acid', y = 'pH', data = quality)
         plt.show()
```

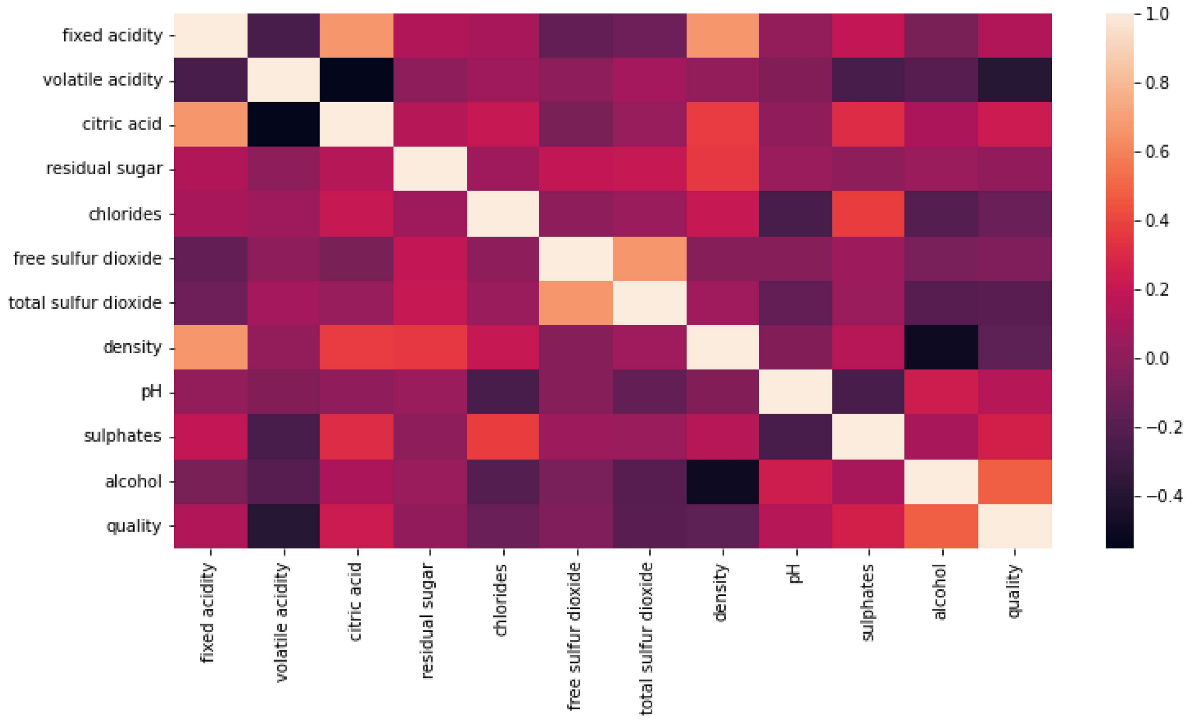


```
In [12]: plt.figure(figsize = (12,6))
         sns.pairplot(quality)
         plt.show()
```

```
<Figure size 864x432 with 0 Axes>
```



In [13]:
```
plt.figure(figsize = (12,6))
sns.heatmap(quality.corr())
plt.show()
```



In [14]:
```
x=quality.drop(['quality'], axis=1)
y=quality['quality']
```

In [15]:
```
from sklearn.ensemble import IsolationForest
```

In [16]:
```
# Outlier Detection using Isolation Forest
outlier_detector = IsolationForest(contamination=0.05)
outlier_labels = outlier_detector.fit_predict(x)
```

```
C:\Users\user\anaconda3\lib\site-packages\sklearn\base.py:439: UserWarning: X
does not have valid feature names, but IsolationForest was fitted with featur
e names
  warnings.warn(
```

In [17]:
```
# Remove outliers from the dataset
x = x[outlier_labels == 1]
y = y[outlier_labels == 1]
```

In [ ]:

```
In summary, the code separates the features (independent variables) into the
DataFrame x by excluding the 'quality' column, and assigns the target
variable (dependent variable) to the Series y by extracting only the
'quality' column.
```

```
This kind of separation is commonly done when preparing data for machine
learning models, where the features and the target variable need to be
handled separately.
```

```
Splitting the data into dependent and independent variables before
preprocessing ensures that
the preprocessing steps are applied consistently to both sets.

If you perform preprocessing for entire dataset before splitting, there is a
risk of data leakage,
where information from the test set or target variable leaks into the
training set,
leading to biased results.
```

```
Preprocessing steps such as scaling or normalization are typically applied to
the features
(independent variables) rather than the target variable (dependent variable).
```

Data Preprocessing

In [18]:

```
pip install imbalanced-learn
```

```
Requirement already satisfied: imbalanced-learn in c:\users\user\anaconda3\li
b\site-packages (0.11.0)Note: you may need to restart the kernel to use updat
ed packages.

Requirement already satisfied: numpy>=1.17.3 in c:\users\user\anaconda3\lib\s
ite-packages (from imbalanced-learn) (1.21.5)
Requirement already satisfied: scipy>=1.5.0 in c:\users\user\anaconda3\lib\si
te-packages (from imbalanced-learn) (1.7.3)
Requirement already satisfied: threadpoolctl>=2.0.0 in c:\users\user\anaconda
3\lib\site-packages (from imbalanced-learn) (2.2.0)
Requirement already satisfied: joblib>=1.1.1 in c:\users\user\anaconda3\lib\s
ite-packages (from imbalanced-learn) (1.2.0)
Requirement already satisfied: scikit-learn>=1.0.2 in c:\users\user\anaconda3
\lib\site-packages (from imbalanced-learn) (1.2.2)
```

In [19]:

```
## oversampling
from imblearn.over_sampling import SMOTE

os = SMOTE()
x_res, y_res = os.fit_resample(x, y)
```

```
The fit_sample() method performs oversampling by generating synthetic samples
from the minority class
(es) in the dataset, in order to balance the class distribution. It creates
new samples by
interpolating between existing samples of the minority class. The resulting
x_res and y_res
are the oversampled feature matrix and target variable, respectively.
```

```
Oversampling is typically used when dealing with imbalanced datasets, where
one or more classes
```

```
are underrepresented compared to other classes. In such cases, oversampling
helps to address the
class imbalance by creating synthetic samples of the minority class(es).
This can improve the performance of machine learning models by providing a
more balanced
representation of the classes during training.
```

In [20]:
```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x_res,y_res,test_size=0.2,
```

In [21]:
```
from sklearn.preprocessing import StandardScaler

stdscale = StandardScaler().fit(x_train)
x_train_std = stdscale.transform(x_train)
x_test_std = stdscale.transform(x_test)
```

In [22]:
```
from sklearn.metrics import accuracy_score
```

Logistic Regression

In [23]:
```
from sklearn.linear_model import LogisticRegression

# Train a logistic regression model
lr = LogisticRegression()
lr.fit(x_train_std, y_train)

# Make predictions on the test set
predictions = lr.predict(x_test_std)

accuracy_score(y_test, predictions)
```

Out[23]:
```
0.5613577023498695
```

In [28]:
```
from sklearn.metrics import classification_report
```

In [29]:
```
# Evaluate model performance
print(classification_report(y_test, predictions))
```

```
              precision    recall  f1-score   support

         3.0       0.76      0.97      0.85       117
         4.0       0.57      0.53      0.55       124
         5.0       0.47      0.44      0.46       123
         6.0       0.43      0.28      0.34       143
         7.0       0.49      0.50      0.50       137
         8.0       0.58      0.72      0.64       122

    accuracy                           0.56       766
   macro avg       0.55      0.57      0.56       766
weighted avg       0.54      0.56      0.55       766
```

Decision Tree Classifier

In [30]:
```
from sklearn.tree import DecisionTreeClassifier

dt = DecisionTreeClassifier()
dt.fit(x_train_std, y_train)

accuracy_score(y_test, dt.predict(x_test_std))
```

Out[30]:
```
0.7924281984334204
```

Random Forest Classifier

In [31]:
```
from sklearn.ensemble import RandomForestClassifier

rf = RandomForestClassifier(random_state = 42)
rf.fit(x_train_std, y_train)

accuracy_score(y_test, rf.predict(x_test_std))
```

Out[31]:
```
0.835509138381201
```

```
# Hence Random Forest Classifier is the best classification model to predict
the wine test quality as
it has the highest accuracy score.
```