# tanaya_siddiqui_201865094_project

December 4, 2023

## 0.1 Importing the python libraries

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

Pandas: A data manipulation and analysis library, Pandas is essential for handling and analyzing structured data. It excels in tasks like reading data from various sources, data cleaning, and complex data operations using DataFrame and Series structures.

Numpy: NumPy is the fundamental package for scientific computing in Python, offering powerful array objects and a wide range of mathematical functions. It's key for numerical operations, especially on large, multi-dimensional arrays and matrices.

Matplotlib: Matplotlib is a versatile plotting library in Python, used for creating a wide variety of static, animated, and interactive visualizations. It's highly customizable and works well for plotting complex graphs and charts.

Seaborn: Seaborn builds on Matplotlib by providing a high-level interface for drawing attractive and informative statistical graphics. It simplifies the creation of complex visualizations and integrates well with Pandas data structures.

## 0.2 Loading the Wine Dataset using pandas

[2]:
```
df = pd.read_csv('wine-data-set .csv')
```

## 0.3 To get a quick look at the first few rows of the dataset

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

[3]:

| | fixed acidity | volatile acidity | citric acid | residual sugar | chlorides | \ |
|---|---|---|---|---|---|---|
| 0 | 7.0 | 0.27 | 0.36 | 20.7 | 0.045 | |
| 1 | 6.3 | 0.30 | 0.34 | 1.6 | 0.049 | |
| 2 | 8.1 | 0.28 | 0.40 | 6.9 | 0.050 | |
| 3 | 7.2 | 0.23 | 0.32 | 8.5 | 0.058 | |
| 4 | 7.2 | 0.23 | 0.32 | 8.5 | 0.058 | |

| | free sulfur dioxide | total sulfur dioxide | density | pH | sulphates | \ |
|---|---|---|---|---|---|---|
| 0 | 45.0 | 170.0 | 1.0010 | 3.00 | 0.45 | |

```
1                14.0                 132.0   0.9940  3.30       0.49
2                30.0                  97.0   0.9951  3.26       0.44
3                47.0                 186.0   0.9956  3.19       0.40
4                47.0                 186.0   0.9956  3.19       0.40

   alcohol  quality
0      8.8        6
1      9.5        6
2     10.1        6
3      9.9        6
4      9.9        6
```

[4]:
```
df.tail()
```

[4]:
```
      fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
6458            6.8             0.620         0.08             1.9      0.068
6459            6.2             0.600         0.08             2.0      0.090
6460            6.3             0.510         0.13             2.3      0.076
6461            5.9             0.645         0.12             2.0      0.075
6462            6.0             0.310         0.47             3.6      0.067

      free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  \
6458                 28.0                  38.0  0.99651  3.42       0.82
6459                 32.0                  44.0  0.99490  3.45       0.58
6460                 29.0                  40.0  0.99574  3.42       0.75
6461                 32.0                  44.0  0.99547  3.57       0.71
6462                 18.0                  42.0  0.99549  3.39       0.66

      alcohol  quality
6458      9.5        6
6459     10.5        5
6460     11.0        6
6461     10.2        5
6462     11.0        6
```

### 0.4 To see a summary of the dataset, including the number of non-null entries and data types for each column

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

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6463 entries, 0 to 6462
Data columns (total 12 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   fixed acidity         6463 non-null   float64
 1   volatile acidity      6463 non-null   float64
 2   citric acid           6463 non-null   float64
```

```
 3   residual sugar        6463 non-null   float64
 4   chlorides             6463 non-null   float64
 5   free sulfur dioxide   6463 non-null   float64
 6   total sulfur dioxide  6463 non-null   float64
 7   density               6463 non-null   float64
 8   pH                    6463 non-null   float64
 9   sulphates             6463 non-null   float64
 10  alcohol               6463 non-null   float64
 11  quality               6463 non-null   int64
dtypes: float64(11), int64(1)
memory usage: 606.0 KB
```

### 0.5 To get statistical summaries of the numeric columns

[6]:
```
df.describe()
```

[6]:

| | fixed acidity | volatile acidity | citric acid | residual sugar | \ |
|---|---|---|---|---|---|
| count | 6463.000000 | 6463.000000 | 6463.000000 | 6463.000000 | |
| mean | 7.217755 | 0.339589 | 0.318758 | 5.443958 | |
| std | 1.297913 | 0.164639 | 0.145252 | 4.756852 | |
| min | 3.800000 | 0.080000 | 0.000000 | 0.600000 | |
| 25% | 6.400000 | 0.230000 | 0.250000 | 1.800000 | |
| 50% | 7.000000 | 0.290000 | 0.310000 | 3.000000 | |
| 75% | 7.700000 | 0.400000 | 0.390000 | 8.100000 | |
| max | 15.900000 | 1.580000 | 1.660000 | 65.800000 | |

| | chlorides | free sulfur dioxide | total sulfur dioxide | density | \ |
|---|---|---|---|---|---|
| count | 6463.000000 | 6463.000000 | 6463.000000 | 6463.000000 | |
| mean | 0.056056 | 30.516865 | 115.694492 | 0.994698 | |
| std | 0.035076 | 17.758815 | 56.526736 | 0.003001 | |
| min | 0.009000 | 1.000000 | 6.000000 | 0.987110 | |
| 25% | 0.038000 | 17.000000 | 77.000000 | 0.992330 | |
| 50% | 0.047000 | 29.000000 | 118.000000 | 0.994890 | |
| 75% | 0.065000 | 41.000000 | 156.000000 | 0.997000 | |
| max | 0.611000 | 289.000000 | 440.000000 | 1.038980 | |

| | pH | sulphates | alcohol | quality |
|---|---|---|---|---|
| count | 6463.000000 | 6463.000000 | 6463.000000 | 6463.000000 |
| mean | 3.218332 | 0.531150 | 10.492825 | 5.818505 |
| std | 0.160650 | 0.148913 | 1.193128 | 0.873286 |
| min | 2.720000 | 0.220000 | 8.000000 | 3.000000 |
| 25% | 3.110000 | 0.430000 | 9.500000 | 5.000000 |
| 50% | 3.210000 | 0.510000 | 10.300000 | 6.000000 |
| 75% | 3.320000 | 0.600000 | 11.300000 | 6.000000 |
| max | 4.010000 | 2.000000 | 14.900000 | 9.000000 |

## Filling the missing values

[7]:
```
for col, value in df.items():
    if col != 'type':
        df[col] = df[col].fillna(df[col].mean()) # Removing rows from the␣
↪DataFrame that contain any missing (null) values
```

### 0.6 Identifying if there are any missing values in the dataset

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

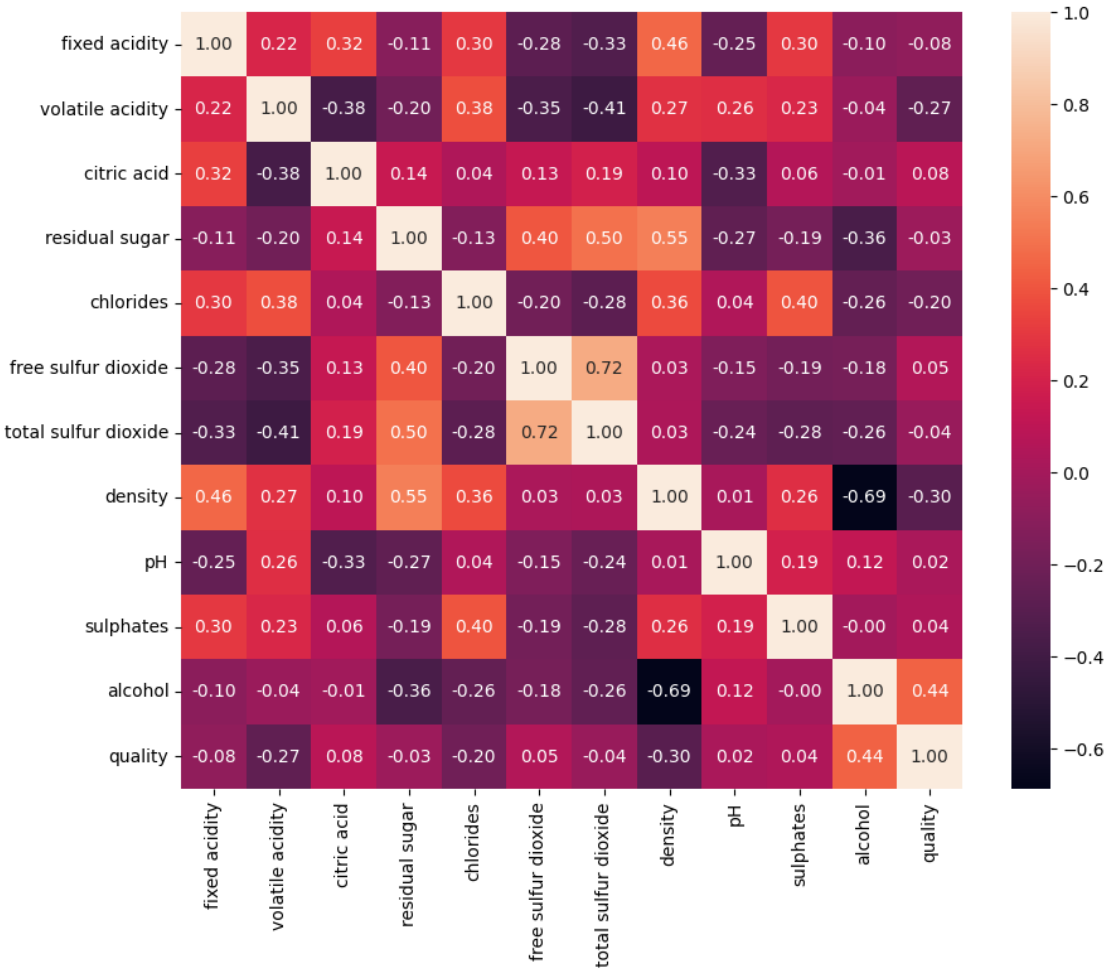
[8]:
```
fixed acidity           0
volatile acidity        0
citric acid             0
residual sugar          0
chlorides               0
free sulfur dioxide     0
total sulfur dioxide    0
density                 0
pH                      0
sulphates               0
alcohol                 0
quality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

### 0.7 Checking the skewness to apply required transformations

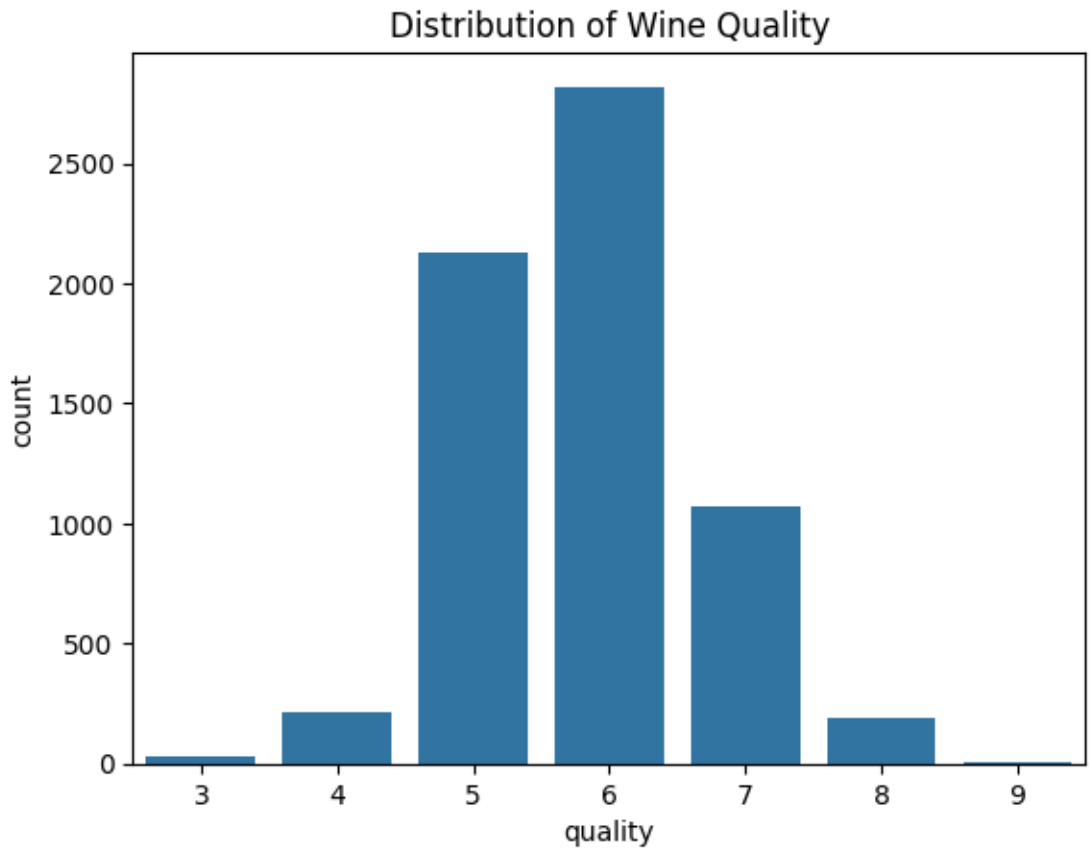
[9]:
```
df.skew()
```

[9]:
```
fixed acidity           1.721648
volatile acidity        1.500040
citric acid             0.474907
residual sugar          1.437126
chlorides               5.403432
free sulfur dioxide     1.223427
total sulfur dioxide   -0.000425
density                 0.504204
pH                      0.391094
sulphates               1.802941
alcohol                 0.565435
quality                 0.189878
dtype: float64
```

[10]:
```
plt.figure(figsize=(10, 8))
sns.heatmap(df.corr(), annot=True, fmt=".2f")
plt.show()
```



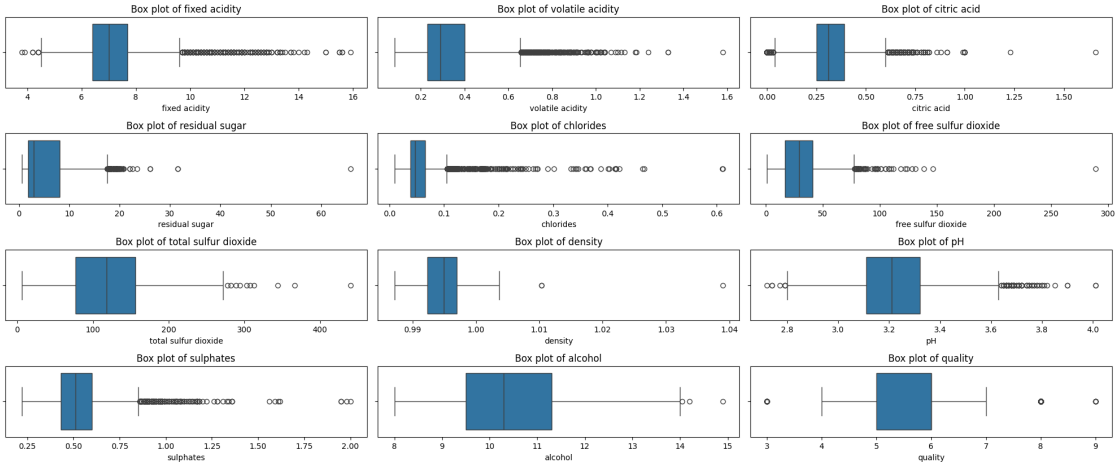

[11]:
```
sns.countplot(x='quality', data=df) # To create a count plot using Seaborn, where x-axis is 'quality'
plt.title('Distribution of Wine Quality') # Title for the plot
plt.show() # Visualize the distribution of the target variable 'quality'
```




[12]:
```
# Set the size of the overall figure
plt.figure(figsize=(20, 10))

# Loop over each column to create a subplot for each feature's box plot
for i, column in enumerate(df.columns):
    plt.subplot(len(df.columns) // 3 + 1, 3, i + 1)  # Adjust the layout as␣
 ↪necessary
    sns.boxplot(x=df[column])
    plt.title(f'Box plot of {column}')
    plt.tight_layout()

plt.show()
```

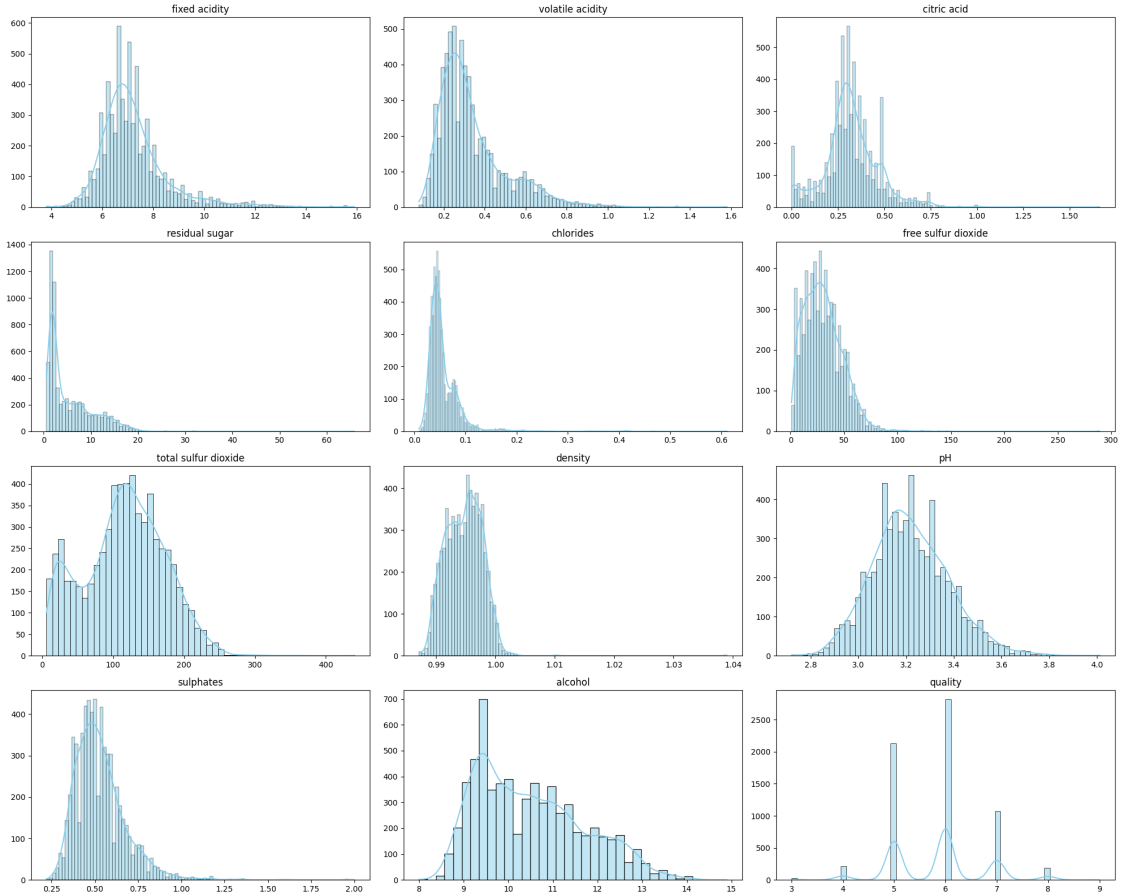



[13]:
```
# Select only the numeric columns for distribution plots
numeric_columns = df.select_dtypes(include=['float', 'int']).columns
num_columns = len(numeric_columns)

plt.figure(figsize=(20, 4 * 4))  # Adjust the figure size as needed

for i, column in enumerate(numeric_columns):
    plt.subplot(4, 3, i + 1)
    sns.histplot(df[column], kde=True, color='skyblue')  # 'kde=True' adds the␣
 ↪Kernel Density Estimate plot
    plt.title(column)
    plt.xlabel('')
    plt.ylabel('')

plt.tight_layout()
plt.show()
```




The "train_test_split" is a function in Python's Scikit-Learn library. Its used in machine learning to divide the dataset into two parts: one for training the machine learning model and the other for testing its performance. In our wine quality prediction project, splitting the data using train_test_split will allow us to train the model on one part of the data and test it on another, ensuring that the model can generalize well to new, unseen data.

[14]:
```
from sklearn.model_selection import train_test_split

X = df.drop('quality', axis=1) # Features (X): The data attributes used to make␣
  ↪predictions.
y = df['quality'] # Target (y): The outcome you're trying to predict.

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
  ↪random_state=42)
```

StandardScaler is a tool in Python's scikit-learn library used to standardize features in a dataset. It adjusts each feature so that it has a mean of 0 and a standard deviation of 1. StandardScaler computes the mean and standard deviation for each feature from the training data, then applies these values to scale both the training and testing data, ensuring consistency across the model.

[15]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler() # To create a StandardScaler Instance
X_train_scaled = scaler.fit_transform(X_train) # To calculate the mean and␣
 ↪standard deviation for each feature in X_train then use the statistics to␣
 ↪scale the training data
X_test_scaled = scaler.transform(X_test) # To scale the testing data
```

[16]:
```
pd.DataFrame(X_train_scaled, columns=X.columns).to_csv('X_train_scaled.csv',␣
 ↪index=False)
pd.DataFrame(y_train).to_csv('y_train.csv', index=False)
pd.DataFrame(X_test_scaled, columns=X.columns).to_csv('X_test_scaled.csv',␣
 ↪index=False)
pd.DataFrame(y_test).to_csv('y_test.csv', index=False)
```

Logistic Regression is a statistical method used for binary classification problems, where the goal is to predict a binary outcome (like yes/no, win/lose, pass/fail). It works by modeling the probability that a given input belongs to a particular category. In this wine quality project, Logistic Regression can be used to predict whether a wine is of high quality or not based on various predictors like acidity, alcohol content, etc.

[17]:
```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression() # To create an instance of the LogisticRegression␣
 ↪class
model = LogisticRegression(max_iter=1000)
model.fit(X_train_scaled, y_train) # To adjust the weights of the model using␣
 ↪the training data, so it can make accurate predictions
```

[17]:
```
LogisticRegression(max_iter=1000)
```

The accuracy_score function in Python is used to evaluate the performance of classification models by comparing the predicted labels against the true labels. It calculates the proportion of correct predictions made by the model, providing a straightforward metric to assess its accuracy. The function returns a value between 0 and 1, where higher values indicate better performance.

[18]:
```
from sklearn.metrics import accuracy_score

y_pred = model.predict(X_test_scaled) # Using the 'predict' method
accuracy = accuracy_score(y_test, y_pred) # To compare predicted values with␣
 ↪the actual values
print(f"Adjusted Model Accuracy: {accuracy*100:.2f}%") # Printing the model␣
 ↪accuracy
```

```
Adjusted Model Accuracy: 53.75%
```

[19]:
```
import joblib
joblib.dump(model, 'regression_model.joblib')
```

[19]: ['regression_model.joblib']

### 0.8 Applying a logarithmic transformation

Logarithmic transformation is a technique used in data preprocessing to stabilize variance, normalize data, and make the data more suitable for analysis, especially when dealing with skewed distributions. By applying a logarithm to each data point, large values are scaled down, and small values are scaled up, leading to a more uniform distribution.

Fixed Acidity have a skew to the right (positive skew). A logarithmic transformation can be applied to reduce the skewness.

[20]:
```
X_train_transformed = X_train.copy()
X_test_transformed = X_test.copy()

# Adding a small constant to avoid log(0) error
X_train_transformed['fixed acidity'] = np.log(X_train_transformed['fixed␣
 ↪acidity'] + 1)
X_test_transformed['fixed acidity'] = np.log(X_test_transformed['fixed␣
 ↪acidity'] + 1)

scaler = StandardScaler()
X_train_transformed_scaled = scaler.fit_transform(X_train_transformed)
X_test_transformed_scaled = scaler.transform(X_test_transformed)

model_transformed = LogisticRegression(max_iter=1000)
model_transformed.fit(X_train_transformed_scaled, y_train)

y_pred_transformed = model_transformed.predict(X_test_transformed_scaled)
accuracy_transformed = accuracy_score(y_test, y_pred_transformed)
print(f"Model Accuracy with Transformed Feature: {accuracy_transformed*100:.
 ↪2f}%")
```

```
Model Accuracy with Transformed Feature: 53.98%
```

### 0.9 Applying a square root transformation

The square root transformation is applied by taking the square root of each value in a feature. This transformation is often used to reduce right skewness in data. It's less intense than a logarithmic transformation and can be applied to zero values, unlike the log transformation

[21]:
```
X_train_transformed = X_train.copy()
X_test_transformed = X_test.copy()

# Adding a small constant to avoid log(0) error
X_train_transformed['residual sugar'] = np.log(X_train_transformed['residual␣
 ↪sugar'] + 1)
X_test_transformed['residual sugar'] = np.log(X_test_transformed['residual␣
 ↪sugar'] + 1)
```

```
scaler = StandardScaler()
X_train_transformed_scaled = scaler.fit_transform(X_train_transformed)
X_test_transformed_scaled = scaler.transform(X_test_transformed)

model_transformed = LogisticRegression(max_iter=1000)
model_transformed.fit(X_train_transformed_scaled, y_train)

y_pred_transformed = model_transformed.predict(X_test_transformed_scaled)
accuracy_transformed = accuracy_score(y_test, y_pred_transformed)
print(f"Model Accuracy with Transformed Feature: {accuracy_transformed*100:.
 ↪2f}%")
```

```
Model Accuracy with Transformed Feature: 53.67%
```

### 0.10 Applying an exponential transformation

This transformation involves taking the exponential (e^x, where e is Euler's number) of each value in a feature. It's the inverse of a logarithmic transformation and is useful for dealing with data that has a rapid increase or exponential growth.

[22]:
```
X_train_exp_transformed = X_train.copy()
X_test_exp_transformed = X_test.copy()

X_train_exp_transformed['chlorides'] = np.
 ↪exp(X_train_exp_transformed['chlorides'])
X_test_exp_transformed['chlorides'] = np.
 ↪exp(X_test_exp_transformed['chlorides'])

# Scaling the features
scaler = StandardScaler()
X_train_exp_scaled = scaler.fit_transform(X_train_exp_transformed)
X_test_exp_scaled = scaler.transform(X_test_exp_transformed)

# Train the logistic regression model
model_exp = LogisticRegression(max_iter=1000)
model_exp.fit(X_train_exp_scaled, y_train)

# Predict and evaluate the model
y_pred_exp = model_exp.predict(X_test_exp_scaled)
accuracy_exp = accuracy_score(y_test, y_pred_exp)
print(f"Model Accuracy with Exponential Transformation: {accuracy_exp*100:.
 ↪2f}%")
```

```
Model Accuracy with Exponential Transformation: 53.67%
```

## 0.11 Applying a power transformation

Power transformations involve raising data to a power. Different powers can be applied, and the choice depends on the distribution of the data.

[23]:
```
power = 3  # You can adjust this value as needed
X_train_power = X_train.copy()
X_test_power = X_test.copy()
X_train_power['volatile acidity'] = np.power(X_train_power['volatile acidity'],␣
 ↪power)
X_test_power['volatile acidity'] = np.power(X_test_power['volatile acidity'],␣
 ↪power)

# Scale the transformed data
scaler = StandardScaler()
X_train_power_scaled = scaler.fit_transform(X_train_power)
X_test_power_scaled = scaler.transform(X_test_power)

# Fit the logistic regression model
model_power = LogisticRegression(max_iter=1000)
model_power.fit(X_train_power_scaled, y_train)

# Predict and evaluate the model
y_pred_power = model_power.predict(X_test_power_scaled)
accuracy_power = accuracy_score(y_test, y_pred_power)
print(f"Model Accuracy with Power Transformation: {accuracy_power*100:.2f}%")
```

```
Model Accuracy with Power Transformation: 52.51%
```

## 0.12 Applying a box-cox transformation

The Box-Cox transformation is a statistical technique used to stabilize variance and make data more normally distributed. It's particularly useful for transforming non-normal dependent variables into a normal shape, which is a common requirement for many linear models.

[24]:
```
from scipy import stats

X_train_boxcox_transformed = X_train.copy()
X_test_boxcox_transformed = X_test.copy()

# Apply Box-Cox transformation (adding a small constant to avoid zero values)
X_train_boxcox_transformed['sulphates'], fitted_lambda = stats.
 ↪boxcox(X_train_boxcox_transformed['sulphates'] + 0.01)
X_test_boxcox_transformed['sulphates'] = stats.
 ↪boxcox(X_test_boxcox_transformed['sulphates'] + 0.01, lmbda=fitted_lambda)

# Scaling the features
X_train_boxcox_scaled = scaler.fit_transform(X_train_boxcox_transformed)
```

```
X_test_boxcox_scaled = scaler.transform(X_test_boxcox_transformed)

# Train the logistic regression model
model_boxcox = LogisticRegression(max_iter=1000)
model_boxcox.fit(X_train_boxcox_scaled, y_train)

# Predict and evaluate the model
y_pred_boxcox = model_boxcox.predict(X_test_boxcox_scaled)
accuracy_boxcox = accuracy_score(y_test, y_pred_boxcox)
print(f"Model Accuracy with Box-Cox Transformation: {accuracy_boxcox*100:.2f}%")
```

```
Model Accuracy with Box-Cox Transformation: 53.91%
```

### 0.13 Checking updated skewness

[25]:
```
# Apply the transformations directly to the DataFrame so I can check the␣
 ↪skewness again
df['fixed acidity'] = np.log(df['fixed acidity'] + 1)  # Logarithmic␣
 ↪transformation
df['residual sugar'] = np.sqrt(df['residual sugar'])   # Square root␣
 ↪transformation
df['chlorides'] = np.exp(df['chlorides'])              # Exponential␣
 ↪transformation
df['volatile acidity'] = np.power(df['volatile acidity'], 3)  # Power␣
 ↪transformation

# Now check the skewness again
updated_skewness = df.skew()
print(updated_skewness)
```

```
fixed acidity           0.975280
volatile acidity        7.438754
citric acid             0.474907
residual sugar          0.715961
chlorides               6.880743
free sulfur dioxide     1.223427
total sulfur dioxide   -0.000425
density                 0.504204
pH                      0.391094
sulphates               1.802941
alcohol                 0.565435
quality                 0.189878
dtype: float64
```

### 0.14 Applying box-cox transformation to 'chloride', 'volatile acidity' and 'sulphate' as they are highly skewed

[26]:
```
# The Box-Cox transformation requires that all values be positive, so you must␣
 ↪ensure there are no zero or negative values
df['chlorides'] += np.abs(df['chlorides'].min()) + 1e-5  # Shift data to␣
 ↪positive if necessary

# Apply the Box-Cox transformation
df['chlorides_transformed'], fitted_lambda = stats.boxcox(df['chlorides'])

print(f"Fitted lambda for Box-Cox transformation: {fitted_lambda}")

print(f"Skewness after Box-Cox transformation: {df['chlorides_transformed'].
 ↪skew()}")
```

```
Fitted lambda for Box-Cox transformation: -43.456133185464644
Skewness after Box-Cox transformation: 0.0
```

[27]:
```
# Check for any non-positive values in 'volatile acidity'
volatile_acidity_min = df['volatile acidity'].min()

if volatile_acidity_min <= 0:
    df['volatile acidity'] += (np.abs(volatile_acidity_min) + 1e-5)

# Apply the Box-Cox transformation
df['volatile_acidity_transformed'], volatile_acidity_lambda = stats.
 ↪boxcox(df['volatile acidity'])

print(f"Fitted lambda for volatile acidity Box-Cox transformation:␣
 ↪{volatile_acidity_lambda}")

print(f"Skewness after Box-Cox transformation for volatile acidity:␣
 ↪{df['volatile_acidity_transformed'].skew()}")
```

```
Fitted lambda for volatile acidity Box-Cox transformation: -0.0943181995058845
Skewness after Box-Cox transformation for volatile acidity: 0.010792704295494726
```

[28]:
```
# Check for any non-positive values in 'sulphates'
sulphates_min = df['sulphates'].min()

if sulphates_min <= 0:
    df['sulphates'] += (np.abs(sulphates_min) + 1e-5)

# Apply the Box-Cox transformation
df['sulphates_transformed'], sulphates_lambda = stats.boxcox(df['sulphates'])

print(f"Fitted lambda for sulphates Box-Cox transformation: {sulphates_lambda}")
```

```
print(f"Skewness after Box-Cox transformation for sulphates:␣
 ↪{df['sulphates_transformed'].skew()}")
```

```
Fitted lambda for sulphates Box-Cox transformation: -0.4634833382096388
Skewness after Box-Cox transformation for sulphates: -0.007557494011492347
```

[29]:
```
df['volatile acidity'] = df['volatile_acidity_transformed']
df['chlorides'] = df['chlorides_transformed']
df['sulphates'] = df['sulphates_transformed']


df.drop(['volatile_acidity_transformed', 'chlorides_transformed',␣
 ↪'sulphates_transformed'], axis=1, inplace=True)

df.skew()
```

[29]:
```
fixed acidity           0.975280
volatile acidity        0.010793
citric acid             0.474907
residual sugar          0.715961
chlorides               0.000000
free sulfur dioxide     1.223427
total sulfur dioxide   -0.000425
density                 0.504204
pH                      0.391094
sulphates              -0.007557
alcohol                 0.565435
quality                 0.189878
dtype: float64
```

[30]:
```
# Dropping predictors from the subset
reduced_df = df.drop(['alcohol', 'pH'], axis=1)

# Splitting into features and target
X_reduced = reduced_df.drop('quality', axis=1)
y_reduced = reduced_df['quality']

# Splitting into training and testing sets
X_train_reduced, X_test_reduced, y_train_reduced, y_test_reduced =␣
 ↪train_test_split(X_reduced, y_reduced, test_size=0.2, random_state=42)

# Scaling
scaler = StandardScaler()
X_train_reduced_scaled = scaler.fit_transform(X_train_reduced)
X_test_reduced_scaled = scaler.transform(X_test_reduced)
```

```
model_reduced = LogisticRegression(max_iter=1000, solver='saga')
model_reduced.fit(X_train_reduced_scaled, y_train_reduced)

y_pred_reduced = model_reduced.predict(X_test_reduced_scaled)
accuracy_reduced = accuracy_score(y_test_reduced, y_pred_reduced)
print(f"Model Accuracy with Reduced Features: {accuracy_reduced*100:.2f}%")
```

```
Model Accuracy with Reduced Features: 50.73%
```

[31]:
```
# Dropping predictors from the subset
reduced_df = df.drop(['chlorides', 'pH'], axis=1)

# Splitting into features and target
X_reduced = reduced_df.drop('quality', axis=1)
y_reduced = reduced_df['quality']

# Splitting into training and testing sets
X_train_reduced, X_test_reduced, y_train_reduced, y_test_reduced =␣
 ↪train_test_split(X_reduced, y_reduced, test_size=0.2, random_state=42)

# Scaling
scaler = StandardScaler()
X_train_reduced_scaled = scaler.fit_transform(X_train_reduced)
X_test_reduced_scaled = scaler.transform(X_test_reduced)

model_reduced = LogisticRegression(max_iter=1000, solver='saga')
model_reduced.fit(X_train_reduced_scaled, y_train_reduced)

y_pred_reduced = model_reduced.predict(X_test_reduced_scaled)
accuracy_reduced = accuracy_score(y_test_reduced, y_pred_reduced)
print(f"Model Accuracy with Reduced Features: {accuracy_reduced*100:.2f}%")
```

```
Model Accuracy with Reduced Features: 53.21%
```

[32]:
```
# Dropping predictors from the subset
reduced_df = df.drop(['residual sugar', 'density'], axis=1)

# Splitting into features and target
X_reduced = reduced_df.drop('quality', axis=1)
y_reduced = reduced_df['quality']

# Splitting into training and testing sets
X_train_reduced, X_test_reduced, y_train_reduced, y_test_reduced =␣
 ↪train_test_split(X_reduced, y_reduced, test_size=0.2, random_state=42)

# Scaling
scaler = StandardScaler()
X_train_reduced_scaled = scaler.fit_transform(X_train_reduced)
```

```
X_test_reduced_scaled = scaler.transform(X_test_reduced)

model_reduced = LogisticRegression(max_iter=1000, solver='saga')
model_reduced.fit(X_train_reduced_scaled, y_train_reduced)

y_pred_reduced = model_reduced.predict(X_test_reduced_scaled)
accuracy_reduced = accuracy_score(y_test_reduced, y_pred_reduced)
print(f"Model Accuracy with Reduced Features: {accuracy_reduced*100:.2f}%")
```

```
Model Accuracy with Reduced Features: 52.98%
```

[33]:
```
# Dropping predictors from the subset
reduced_df = df.drop(['volatile acidity', 'fixed acidity'], axis=1)

# Splitting into features and target
X_reduced = reduced_df.drop('quality', axis=1)
y_reduced = reduced_df['quality']

# Splitting into training and testing sets
X_train_reduced, X_test_reduced, y_train_reduced, y_test_reduced =␣
 ↪train_test_split(X_reduced, y_reduced, test_size=0.2, random_state=42)

# Scaling
scaler = StandardScaler()
X_train_reduced_scaled = scaler.fit_transform(X_train_reduced)
X_test_reduced_scaled = scaler.transform(X_test_reduced)

model_reduced = LogisticRegression(max_iter=1000, solver='saga')
model_reduced.fit(X_train_reduced_scaled, y_train_reduced)

y_pred_reduced = model_reduced.predict(X_test_reduced_scaled)
accuracy_reduced = accuracy_score(y_test_reduced, y_pred_reduced)
print(f"Model Accuracy with Reduced Features: {accuracy_reduced*100:.2f}%")
```

```
Model Accuracy with Reduced Features: 51.04%
```

[34]:
```
# Dropping predictors from the subset
reduced_df = df.drop(['citric acid', 'pH'], axis=1)

# Splitting into features and target
X_reduced = reduced_df.drop('quality', axis=1)
y_reduced = reduced_df['quality']

# Splitting into training and testing sets
X_train_reduced, X_test_reduced, y_train_reduced, y_test_reduced =␣
 ↪train_test_split(X_reduced, y_reduced, test_size=0.2, random_state=42)

# Scaling
```

```
scaler = StandardScaler()
X_train_reduced_scaled = scaler.fit_transform(X_train_reduced)
X_test_reduced_scaled = scaler.transform(X_test_reduced)

model_reduced = LogisticRegression(max_iter=1000, solver='saga')
model_reduced.fit(X_train_reduced_scaled, y_train_reduced)

y_pred_reduced = model_reduced.predict(X_test_reduced_scaled)
accuracy_reduced = accuracy_score(y_test_reduced, y_pred_reduced)
print(f"Model Accuracy with Reduced Features: {accuracy_reduced*100:.2f}%")
```

```
Model Accuracy with Reduced Features: 53.36%
```

After analyzing the various accuracy measurements post-data transformation and feature reduction, it's evident that the transformations have had a modest impact on our logistic regression model's performance. The logarithmic transformation yielded a slight improvement with an accuracy of 53.98%, while the power transformation slightly decreased the model's accuracy to 52.51%. The Box-Cox transformation also showed a positive effect with an accuracy close to the logarithmic at 53.91%. These transformations aimed to normalize the data, yet the minimal changes in accuracy suggest that our model's predictive capabilities are potentially influenced by other factors beyond skewness correction.

Feature reduction results revealed more about the importance of specific attributes. Dropping alcohol and pH led to the most substantial accuracy drop to 50.73%, indicating their significance in predicting wine quality. Other reduced feature models fluctuated around this accuracy mark, with the removal of chlorides and pH resulting in a 53.21% accuracy, which still represents a reasonable prediction rate. This suggests that while some features are pivotal, others can be excluded with a lesser impact on the model's performance.

Overall, these findings illustrate the delicate balance between feature selection and data transformation in machine learning. The slight variations in accuracy post-transformation and feature reduction underscore the importance of careful preprocessing decisions. The bottom-line accuracy, our starting point, served as a benchmark against which we measured the success of our preprocessing steps, guiding us to a better understanding of our model's behavior and the dataset's characteristics.

[ ]: