ECE 657A - Assignment 2

Submission Date: 10 March 2022

Libraries

We start by loading the necessary libraries

In [ ]:
```
# List of libraries
import math
import numpy as np
import pandas as pd
import random
import seaborn as sns
from sklearn import neighbors
from scipy import stats
from sklearn.decomposition import PCA
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import cross_validate
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.manifold import TSNE
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import ComplementNB
from sklearn.naive_bayes import GaussianNB
from sklearn.naive_bayes import MultinomialNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler
from sklearn.tree import DecisionTreeClassifier
from sklearn.tree import plot_tree
import matplotlib.pyplot as plt
```

Loading the Datasets

Before starting with the questions, we will first load the dataset that are required for this assignment. We will load the Abalone Dataset and the Wine dataset as shown below:

Abalone Dataset

For loading the abalone data, please see the code below:

In [ ]:
```
# Columns/ Features for the data
heading = ["Sex", "Length (mm)", "Diameter (mm)", "Height (mm)", "Whole Weight (g)", "Shucke

#Loading Data set by using pandas
abalone = pd.read_csv("abalone.csv", sep=",", names = heading)
```

For the purposes of this assignment, we will drop the sex feature because it is a categorical feature and does not work well with feature reduction.

In [ ]:
```
# Making a dataset by removing the categorical data and the label
abalone_remaining_features = abalone.drop(columns = ["Sex","Rings"])

# The target variable is the following
```

```
# The Minimum-Maximum Normalized dataset is the following
abalone_raw= ((abalone_remaining_features-abalone_remaining_features.min())/(abalone_remaini

# The train test split of the Minimum-Maximum Normalized Abalone data
abalone_raw_train , abalone_raw_test, abalone_rings_raw_train, abalone_rings_raw_test = trai
```

For this assignment, abalone_raw will represent the Minimum-Maximum Normalized Abalone features

Wine Dataset

Now we will load the wine dataset which is shown below:

In [ ]:
```
# loading the two wine datasets
wine_r = pd.read_csv("winequality-red.csv", sep=";")
wine_w = pd.read_csv("winequality-white.csv", sep=";")

#Columns/Features
D = ['fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar', 'chlorides', 'fre
L = 'quality'
C = 'color'
DL = D + [L]
DC = D + [C]
DLC = DL + [C]

# concatenation of the two wine datasets
wine_w= wine_w.copy()
wine_w[C]= np.zeros(wine_w.shape[0])
wine_r[C]= np.ones(wine_r.shape[0])
wine = pd.concat([wine_w,wine_r], ignore_index=True)
```

For the purposes of this assignment, we will drop the color feature because it is a categorical feature and does not work well with feature reduction.

In [ ]:
```
# Making a dataset of just the features
wine_remaining_features = wine.drop(columns = ["color","quality"])

# The target variable is the following
wine_quality = wine['quality']

# The Minimum-Maximum Normalized dataset is the following
wine_raw= ((wine_remaining_features-wine_remaining_features.min())/(wine_remaining_features.

# The train test split of the Minimum-Maximum Normalized Wine data
wine_raw_train , wine_raw_test, wine_quality_raw_train, wine_quality_raw_test = train_test_s
```

For this assignment, wine_raw will represent the Minimum-Maximum Normalized Wine features

Question 1: Representation Learning

1.1 PCA, LDA & t-SNE

1.1.1 Abalone Dataset

PCA

For generating a PCA plot of the abalone raw dataset, we will execute the following commands:

In [ ]:
```
# construct the PCA classifier for the abalone dataset
abalone_pca_all = PCA(n_components = (len(abalone_raw.columns)))

# make an array to name the principal components of the Abalone Dataset
abalone_col = ["PC-1","PC-2","PC-3","PC-4","PC-5","PC-6","PC-7"]
```

```
# executing PCA on the Min-Max Normalized Abalone Dataset using the training dataset
abalone_pca_all_train = pd.DataFrame(abalone_pca_all.fit_transform(abalone_raw_train),columns

# executing PCA on the Min-Max Normalized Abalone Dataset using the testing dataset
abalone_pca_all_test = pd.DataFrame(abalone_pca_all.transform(abalone_raw_test),columns = ab
```

Now we will add the labels back to the dataset as shown below:

In [ ]:
```
# we will add a feature coloumn for the rings in the training set
abalone_pca_all_train["Rings"] = abalone_rings_raw_train.reset_index().drop(columns=["index"

# we will add a feature coloumn for the rings in the testing set
abalone_pca_all_test["Rings"] = abalone_rings_raw_test.reset_index().drop(columns=["index"])
```

The final PCA dataset is the concatenation of the following datasets:

In [ ]:
```
abalone_pca_all_final = pd.concat([abalone_pca_all_train,abalone_pca_all_test])
```

Now we will plot the graph of the PCA as shown below:

In [ ]:
```
# initialize the plot
sns.set(rc = {'figure.figsize':(15,8)})

# provide the information regarding the scatter plot
sns.scatterplot(data = abalone_pca_all_final, x = "PC-1", y = "PC-2", hue = "Rings")

# assign a title to the plot
plt.title("Abalone Minimum-Maximum Normalized PCA Dataset")
```

Out[ ]:
```
Text(0.5, 1.0, 'Abalone Minimum-Maximum Normalized PCA Dataset')
```

LDA

In [ ]:
```
# initialize the LDA Classifier with all the components
abalone_lda_all = LinearDiscriminantAnalysis(n_components = len(abalone_raw.columns))

# ascertaining the mean and the standard deviation of the training data and then training the
abalone_lda_all_train = pd.DataFrame(abalone_lda_all.fit_transform(abalone_raw_train, abalon

                                                    transform function
abalone_lda_all_test = pd.DataFrame(abalone_lda_all.transform(abalone_raw_test))
```

The final PCA dataset is the concatenation of the following datasets:

In [ ]:
```
# we will add a feature coloumn for the rings in the training set
abalone_lda_all_train["Rings"] = abalone_rings_raw_train.reset_index().drop(columns=["index"

# we will add a feature coloumn for the rings in the testing set
abalone_lda_all_test["Rings"] = abalone_rings_raw_test.reset_index().drop(columns=["index"])
```

Now we will concatenate the two datasets as shown below:

In [ ]:
```
abalone_lda_all_final = pd.concat([abalone_lda_all_train,abalone_lda_all_test])
```

Now we will plot the graph of the LDA as shown below:

In [ ]:
```
# initialize the plot
sns.set(rc = {'figure.figsize':(15,8)})

# provide the information regarding the scatter plot
sns.scatterplot(data = abalone_lda_all_final, x = 0, y = 1, hue = "Rings")

# assign the title, x label & y label to the plot
plt.title("Abalone Minimum-Maximum Normalized PCA Dataset")
plt.xlabel("Component - 1")
plt.ylabel("Component - 2")
```

Out[ ]:
```
Text(0, 0.5, 'Component - 2')
```

t-SNE

We will first generate the t-SNE as shown below:

In [ ]:
```
# initialize the t-SNE Classifier with 2 components
abalone_tsne = TSNE(n_components=2, verbose=1, random_state=27)

# fit and train the classifier using the training dataset
abalone_tsne_final = pd.DataFrame(abalone_tsne.fit_transform(abalone_raw, abalone_rings),col
```

```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\manifold\_t_
sne.py:780: FutureWarning: The default initialization in TSNE will change from 'random' to 'p
ca' in 1.2.
  warnings.warn(
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\manifold\_t_
sne.py:790: FutureWarning: The default learning rate in TSNE will change from 200.0 to 'auto'
in 1.2.
  warnings.warn(
[t-SNE] Computing 91 nearest neighbors...
[t-SNE] Indexed 4177 samples in 0.018s...
[t-SNE] Computed neighbors for 4177 samples in 0.350s...
[t-SNE] Computed conditional probabilities for sample 1000 / 4177
[t-SNE] Computed conditional probabilities for sample 2000 / 4177
[t-SNE] Computed conditional probabilities for sample 3000 / 4177
[t-SNE] Computed conditional probabilities for sample 4000 / 4177
[t-SNE] Computed conditional probabilities for sample 4177 / 4177
[t-SNE] Mean sigma: 0.021783
[t-SNE] KL divergence after 250 iterations with early exaggeration: 67.945961
[t-SNE] KL divergence after 1000 iterations: 1.141203
```

Now we will add back the label in the dataset as shown below:

In [ ]:
```
abalone_tsne_final["Rings"] = abalone_rings
```

Now we will plot the scatter plot of the t-SNE data as shown below:

In [ ]:
```
# initialize the plot
sns.set(rc = {'figure.figsize':(15,8)})

# provide the paramters of the plot
sns.scatterplot(data=abalone_tsne_final, x="Comp-1", y="Comp-2", hue="Rings")

# assign the title of the plot
plt.title("t-SNE graph of Min-Max Normalized Abalone Dataset")
```

Out[ ]:
```
Text(0.5, 1.0, 't-SNE graph of Min-Max Normalized Abalone Dataset')
```

1.1.2 Wine Dataset

PCA

For generating a PCA plot of the wine raw dataset, we will train the PCA algorithm and then generate the Principal Components as shown below:

In [ ]:
```
# Initializing the PCA algorithm
wine_pca_all = PCA(n_components = len(wine_raw.columns))

# Array for naming the columns of the PCA dataset
wine_col = ["PC-1","PC-2","PC-3","PC-4","PC-5","PC-6","PC-7","PC-8","PC-9","PC-10","PC-11"]

# executing PCA on the Min-Max Normalized Wine Dataset using the training dataset
wine_pca_all_train = pd.DataFrame(wine_pca_all.fit_transform(wine_raw_train, wine_quality_ra

# executing PCA on the Min-Max Normalized wine Dataset using the testing dataset
wine_pca_all_test = pd.DataFrame(wine_pca_all.transform(wine_raw_test),columns = wine_col)
```

Now we combine the two dataset and add the Quality feature as shown below:

In [ ]:
```
# we will add a feature coloumn for the quality in the training set
wine_pca_all_train["Quality"] = wine_quality_raw_train.reset_index().drop(columns=["index"])

# we will add a feature coloumn for the quality in the testing set
wine_pca_all_test["Quality"] = wine_quality_raw_test.reset_index().drop(columns=["index"])
```

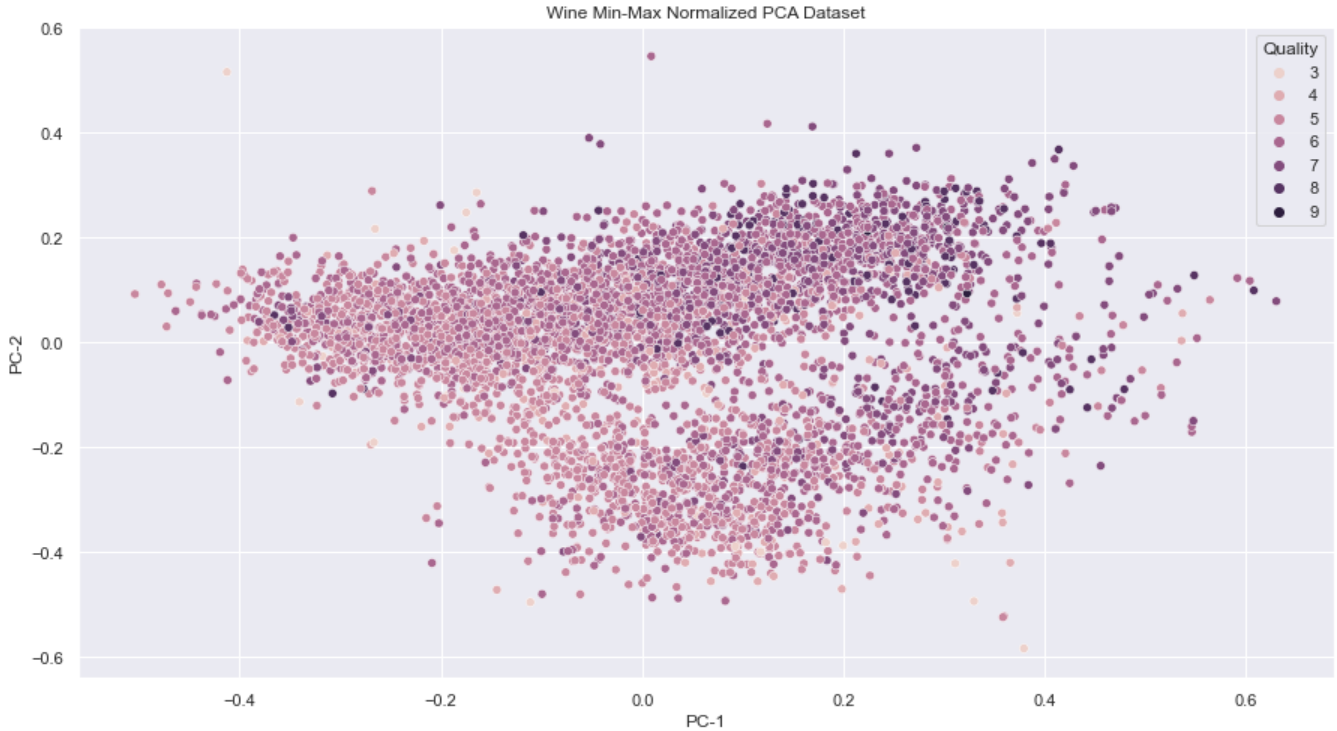
Now we will concatenate the two dataset as shown below:

In [ ]:
```
wine_pca_all_final = pd.concat([wine_pca_all_train,wine_pca_all_test])
```

Now we will plot the scatter plot using the principal compoenents as shown below:

In [ ]:
```
# initialize the plot
sns.set(rc = {'figure.figsize':(15,8)})

# provide the information regarding the scatter plot
sns.scatterplot(data=wine_pca_all_final, x="PC-1", y="PC-2", hue="Quality")

# assign a title to the plot
plt.title("Wine Min-Max Normalized PCA Dataset")
```

Out[ ]:
```
Text(0.5, 1.0, 'Wine Min-Max Normalized PCA Dataset')
```



LDA

vn below:

In [ ]:
```
# initialize the LDA Classifier with 6 components
wine_lda_all = LinearDiscriminantAnalysis(n_components = 6)

# ascertaining the mean and the standard deviation of the training data and then training the
wine_lda_all_train = pd.DataFrame(wine_lda_all.fit_transform(wine_raw_train, wine_quality_ra

# transform the test data using the transform function
wine_lda_all_test = pd.DataFrame(wine_lda_all.transform(wine_raw_test))
```

Now we will add the label data back to the two datasets as shown below:

In [ ]:
```
# we will add a feature coloumn for the quality in the training set
wine_lda_all_train["Quality"] = wine_quality_raw_train.reset_index().drop(columns=["index"])

# we will add a feature coloumn for the quality in the testing set
wine_lda_all_test["Quality"] = wine_quality_raw_test.reset_index().drop(columns=["index"])
```

We will concatenate the two datasets as shown below:

In [ ]:
```
wine_lda_all_final = pd.concat([wine_lda_all_train,wine_lda_all_test])
```

t-SNE

First we will generate the t-SNE dataset as shown below:

In [ ]:
```
# initialize the t-SNE Classifier with 2 components
wine_tsne = TSNE(n_components=2, verbose=1, random_state=27)

# fit and train the classifier using the training dataset
wine_tsne_final = pd.DataFrame(wine_tsne.fit_transform(wine_raw, wine_quality),columns=["Com

# Concatenate the quality feature to the dataset
wine_tsne_final["Quality"] =  wine_quality
```

```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\manifold\_t_
sne.py:780: FutureWarning: The default initialization in TSNE will change from 'random' to 'p
ca' in 1.2.
  warnings.warn(
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\manifold\_t_
sne.py:790: FutureWarning: The default learning rate in TSNE will change from 200.0 to 'auto'
in 1.2.
  warnings.warn(
[t-SNE] Computing 91 nearest neighbors...
[t-SNE] Indexed 6497 samples in 0.045s...
[t-SNE] Computed neighbors for 6497 samples in 1.737s...
[t-SNE] Computed conditional probabilities for sample 1000 / 6497
[t-SNE] Computed conditional probabilities for sample 2000 / 6497
[t-SNE] Computed conditional probabilities for sample 3000 / 6497
[t-SNE] Computed conditional probabilities for sample 4000 / 6497
[t-SNE] Computed conditional probabilities for sample 5000 / 6497
[t-SNE] Computed conditional probabilities for sample 6000 / 6497
[t-SNE] Computed conditional probabilities for sample 6497 / 6497
[t-SNE] Mean sigma: 0.069755
[t-SNE] KL divergence after 250 iterations with early exaggeration: 81.371368
[t-SNE] KL divergence after 1000 iterations: 1.374188
```

Now we will plot the graph as shown below:

In [ ]:
```
# initialize the t-SNE plot
sns.set(rc = {'figure.figsize':(15,8)})

# assign the datapoints for the axes
sns.scatterplot(data= wine_tsne_final, x="Comp-1", y="Comp-2", hue="Quality")
```

```
# assign the graph title
plt.title("t-SNE graph of Wine Dataset")
```

Out[ ]:
```
Text(0.5, 1.0, 't-SNE graph of Wine Dataset')
```

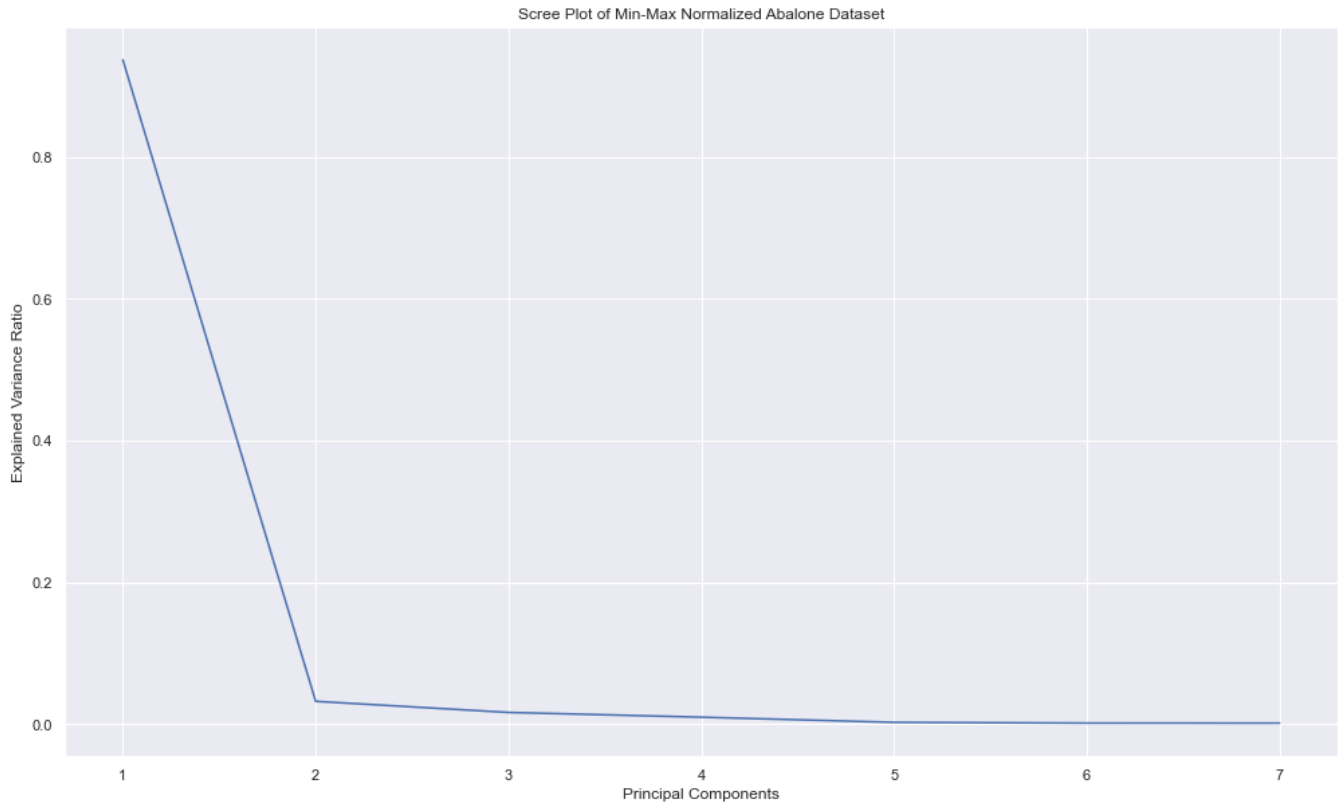
1.2 PCA Scree-Plot

1.2.1 Abalone Dataset

We will plot the scree plot by using the explained variance ratio of the sklearn.pca library

In [ ]:
```
# make a list of array of all of the Principal Component
abalone_PC_values = np.arange(abalone_pca_all.n_components_) + 1

# initialize a plot using seaborn
sns.set(rc = {'figure.figsize':(17,10)})

# assign the datapoints for the axes
sns.lineplot( x=abalone_PC_values, y=abalone_pca_all.explained_variance_ratio_)

# assign the X-Axis label
plt.xlabel("Principal Components")

# assign the Y-Axis label
plt.ylabel("Explained Variance Ratio")

# assign the Graph title
plt.title("Scree Plot of Min-Max Normalized Abalone Dataset")
```

Out[ ]:
```
Text(0.5, 1.0, 'Scree Plot of Min-Max Normalized Abalone Dataset')
```



1.2.2 Wine Dataset

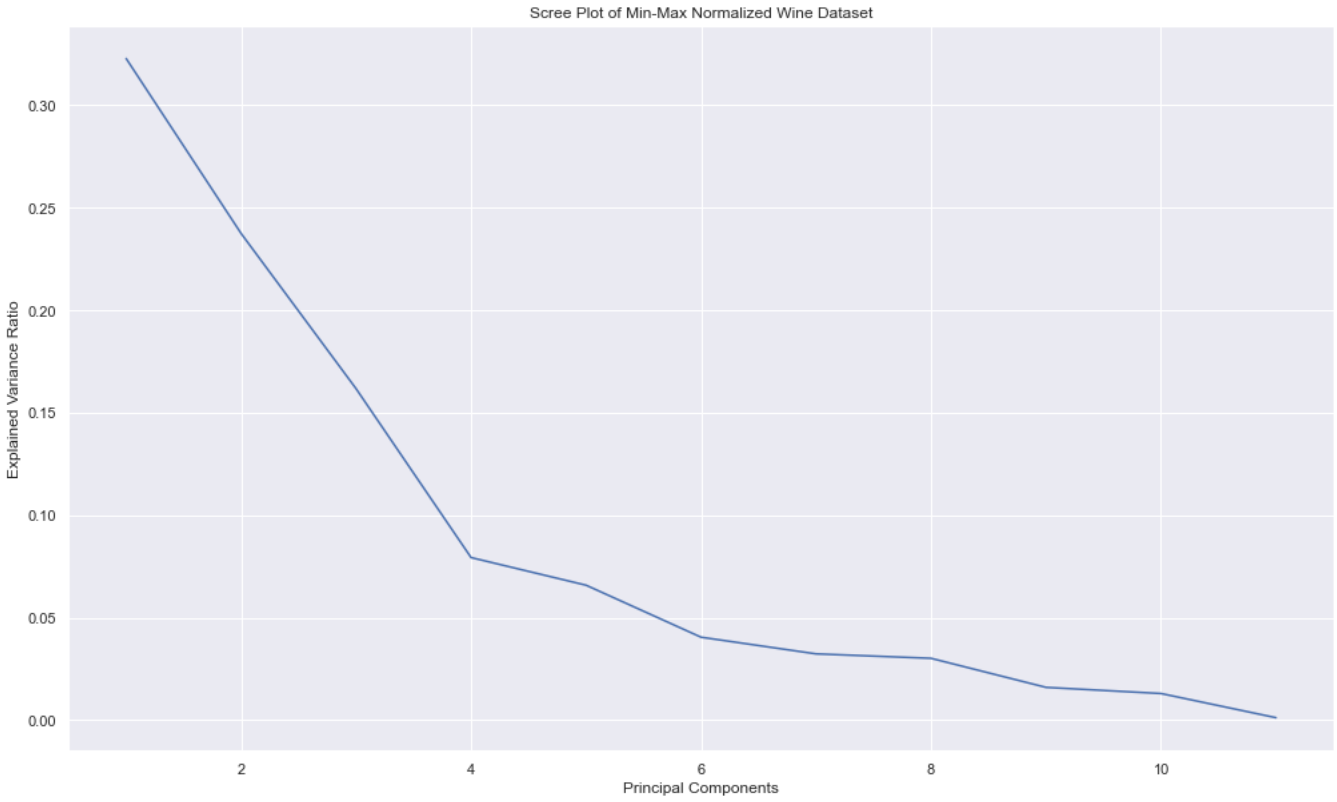
We will plot the scree plot by using the explained variance ratio of the sklearn.pca library

In [ ]:

```
# make a list of array of all of the Principal Component
wine_PC_values = np.arange(wine_pca_all.n_components_) + 1

# initialize a plot using seaborn
sns.set(rc = {'figure.figsize':(17,10)})

# assign the datapoints for the axes
sns.lineplot( x=wine_PC_values, y=wine_pca_all.explained_variance_ratio_)

# assign the X-Axis label
plt.xlabel("Principal Components")

# assign the Y-Axis label
plt.ylabel("Explained Variance Ratio")

# assign the Graph title
plt.title("Scree Plot of Min-Max Normalized Wine Dataset")
```

Out[ ]:

```
Text(0.5, 1.0, 'Scree Plot of Min-Max Normalized Wine Dataset')
```



1.3 kNN using PCA

1.3.1 Abalone Dataset

From the previous assignment, among the normalized datasets, the best accuracy was achieved at k = 115 and Euclidean distance. Therefore, we will use these setting for the determining the best number of Principal Components

To find the best number of Principal Components, we will create a for loop in which we generate a kNN classifier and dataset for a specific number of principal component. This will be in the range of 2 to 7. Then we are going to train these classifiers as shown below:

In [ ]:
```
# initialize for loop to generate the kNN classifier and its respective dataset for  "i"  pr
for i in range(2,abalone_pca_all_train.shape[1]):

    # generate kNN classifier for each set of principal component ranging from 2 to 7
    globals()[f"abalone_pca_all_knn_{i}"] = KNeighborsClassifier(n_neighbors=115, weights="d

    # train each classifier with PCA training data with the respective number of principal c
    globals()[f"abalone_pca_all_knn_{i}"].fit( abalone_pca_all_train.iloc[:,0:i], abalone_pc
```

Now, we will use the test data to see the performance of each of the classifier by using a for loop as shown below:

In [ ]:
```
# initialize the array for the test scores of the different classifiers
abalone_pca_knn_scores = []

# initialize the for loop to check the performance of the classifier on the PCA test data
for i in range(2,abalone_pca_all_train.shape[1]):

    # fit the PCA test data with the respective features subset on the classifier and store t
    abalone_pca_knn_scores.append(globals()[f"abalone_pca_all_knn_{i}"].score(abalone_pca_al
```

We saved all the scores in the abalone_pca_knn_scores array

The principal components required to achieve the maximum accuracy is shown below:

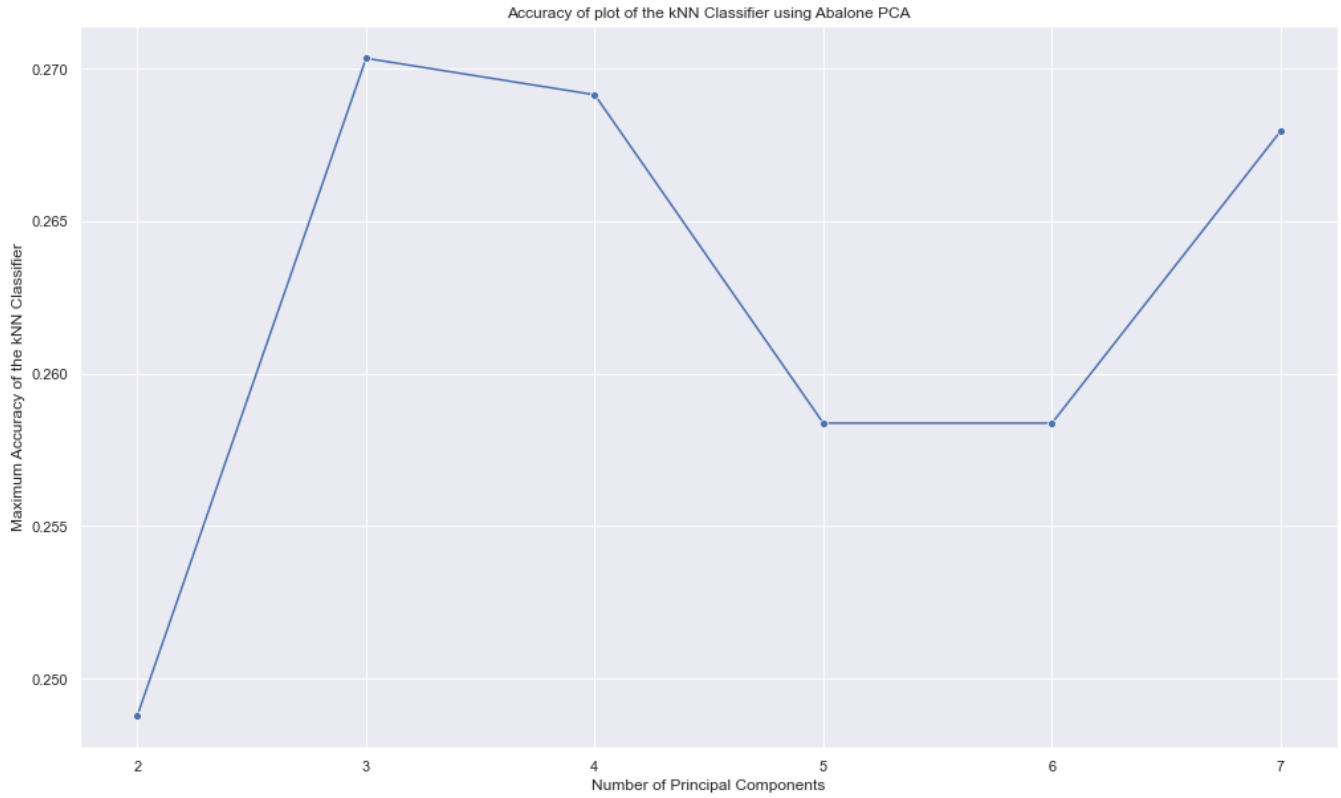
```
print("Number of Principal Components required for maximum accuracy: " + str(abalone_pca_knn_
```

```
Number of Principal Components required for maximum accuracy: 3
```

Now we will plot the graph of the accuracy on the y-axis and the number of principal components on the x-axis

In [ ]:
```
sns.set(rc = {'figure.figsize':(17,10)})
sns.lineplot( x=(range(2, (len(abalone_pca_knn_scores)+2))), y=abalone_pca_knn_scores,marker=
plt.xlabel("Number of Principal Components")
plt.ylabel("Maximum Accuracy of the kNN Classifier")
plt.title("Accuracy of plot of the kNN Classifier using Abalone PCA")
```

Out[ ]:
```
Text(0.5, 1.0, 'Accuracy of plot of the kNN Classifier using Abalone PCA')
```



For the PCA data, 3 principal components generate the best accuracy with the test data. Moving forward, we are only going to be using 3 principal components in the remaining questions. For that, we will generate the datset which is shown below:

In [ ]:
```
# generate the training set
abalone_pca_train = abalone_pca_all_train.iloc[:,0:3]
abalone_pca_train["Rings"] = abalone_pca_all_train["Rings"]

# generate the test set
abalone_pca_test = abalone_pca_all_test.iloc[:,0:3]
abalone_pca_test["Rings"] = abalone_pca_all_test["Rings"]
```

```
C:\Users\ibteh\AppData\Local\Temp/ipykernel_10572/3674897567.py:7: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  abalone_pca_test["Rings"] = abalone_pca_all_test["Rings"]
```

1.3.2 Wine Dataset

From the previous assignment, among the normalized datasets, the best accuracy was achieved at k = 124 and Manhattan distance. Therefore, we will use these setting for the determining the best number of Principal Components

onents, we will create a for loop in which we generate a kNN

classifier and dataset for a specific number of principal component. This will be in the range of 2 to 11. Then we are going to train these classifiers as shown below:

In [ ]:
```
# initialize for loop to generate the kNN classifier and its respective dataset for  "i"  pr
for i in range(2,wine_pca_all_train.shape[1]):

    # # generate kNN classifier for each set of principal component ranging from 2 to 11
    globals()[f"wine_pca_all_knn_{i}"] = KNeighborsClassifier(n_neighbors=124, weights="dist
    # train each classifier with PCA training data with the respective number of principal c
    globals()[f"wine_pca_all_knn_{i}"].fit( wine_pca_all_train.iloc[:,0:i], wine_pca_all_tra
```

Now, we will use the test data to see the performance of each of the classifier by using a for loop as shown below:

In [ ]:
```
# initialize the array for the test scores of the different classifiers
wine_pca_knn_scores = []

# initialize the for loop to check the performance of the classifier on the PCA test data
for i in range(2,wine_pca_all_train.shape[1]):

    # fit the PCA test data with the respective features subset on the classifier and store
    wine_pca_knn_scores.append(globals()[f"wine_pca_all_knn_{i}"].score(wine_pca_all_test.il
```

We saved all the scores in the abalone_pca_knn_scores array

The principal components required to achieve the maximum accuracy is shown below:

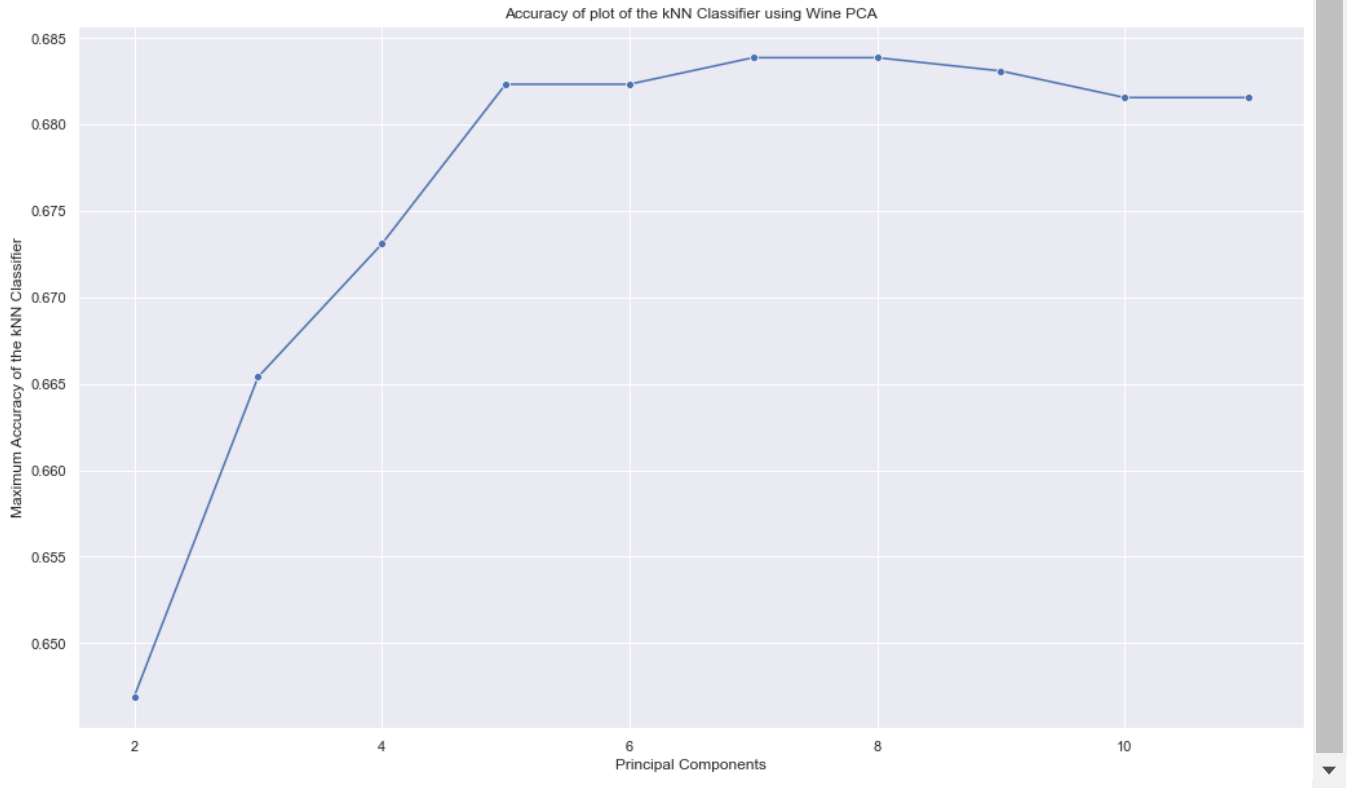
In [ ]:
```
print("Number of Principal Components required for maximum accuracy: " + str(wine_pca_knn_sc
```

```
Number of Principal Components required for maximum accuracy: 7
```

Now we will plot the graph of the accuracy on the y-axis and the number of principal components on the x-axis

In [ ]:
```
sns.set(rc = {'figure.figsize':(17,10)})
sns.lineplot( x=(range(2, (len(wine_pca_knn_scores)+2))), y=wine_pca_knn_scores,marker="o")
plt.xlabel("Principal Components")
plt.ylabel("Maximum Accuracy of the kNN Classifier")
plt.title("Accuracy of plot of the kNN Classifier using Wine PCA")
```

Out[ ]:
```
Text(0.5, 1.0, 'Accuracy of plot of the kNN Classifier using Wine PCA')
```



For the PCA data, 7 principal components generate the best accuracy with the test data. Moving forward, we are only going to be using 7 principal components in the remaining question. For that, we will generate the datset which is shown below:

In [ ]:
```
# generate the training set
wine_pca_train = wine_pca_all_train.iloc[:,0:7]
wine_pca_train["Quality"] = wine_pca_all_train["Quality"]

# generate the test set
wine_pca_test = wine_pca_all_test.iloc[:,0:7]
wine_pca_test["Quality"] = wine_pca_all_test["Quality"]
```

```
C:\Users\ibteh\AppData\Local\Temp/ipykernel_10572/330867122.py:7: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guid
e/indexing.html#returning-a-view-versus-a-copy
  wine_pca_test["Quality"] = wine_pca_all_test["Quality"]
```

1.4 kNN using LDA

1.4.1 Abalone Dataset

Now we will use the kNN classifier to see what is the performance with different number of principal components

In [ ]:
```
# initialize for loop to generate the kNN classifier and its respective dataset for  "i"  pr
for i in range(2,abalone_lda_all_train.shape[1]):

    # generate kNN classifier for each set of component ranging from 2 to 7
    globals()[f"abalone_lda_all_knn_{i}"] = KNeighborsClassifier(n_neighbors=115, weights="d

    # train each classifier with LDA training data with the respective number of principal c
    globals()[f"abalone_lda_all_knn_{i}"].fit( abalone_lda_all_train.iloc[:,0:i], abalone_lda
```

Now we will use test data to see the accuracy of the classifiers and save the data in an array as shown below:

In [ ]:

```
# initialize the array for the test scores of the different classifiers
abalone_lda_knn_scores = []

# initialize the for loop to check the performance of the classifier on the LDA test data
for i in range(2,abalone_lda_all_train.shape[1]):

    # fit the LDA test data with the respective features subset on the classifier and store t
    abalone_lda_knn_scores.append(globals()[f"abalone_lda_all_knn_{i}"].score(abalone_lda_al
```

The number of components required for the best accuracy is shown below:

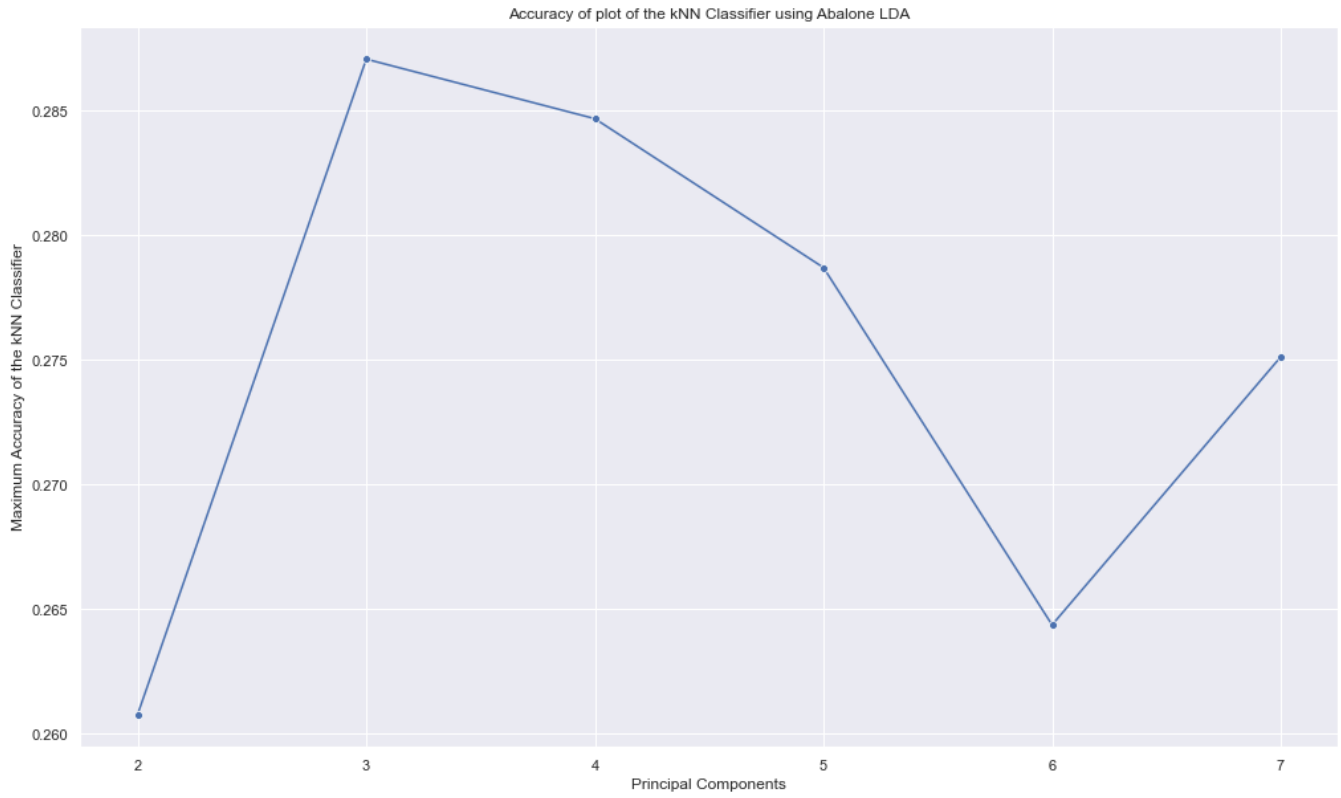
In [ ]:
```
print("Number of Components required for maximum accuracy: " + str(abalone_lda_knn_scores.in
```

```
Number of Components required for maximum accuracy: 3
```

Now we will plot the graph of the accuracy on the y-axis and the number of principal components on the x-axis

In [ ]:
```
# initialize the plot
sns.set(rc = {'figure.figsize':(17,10)})

# input the necessary parameters for the plot
sns.lineplot( x=(range(2, (len(abalone_lda_knn_scores)+2))), y=abalone_lda_knn_scores,marker=

# assign the x label of the graph
plt.xlabel("Principal Components")

# assign the y label of the graph
plt.ylabel("Maximum Accuracy of the kNN Classifier")

# assign the title of the graph
plt.title("Accuracy of plot of the kNN Classifier using Abalone LDA")
```

Out[ ]:
```
Text(0.5, 1.0, 'Accuracy of plot of the kNN Classifier using Abalone LDA')
```



In [ ]:
```
# generate the training set
abalone_lda_train = abalone_lda_all_train.iloc[:,0:4]
abalone_lda_train["Rings"] = abalone_lda_all_train["Rings"]
```

```
abalone_lda_test = abalone_lda_all_test.iloc[:,0:4]
abalone_lda_test["Rings"] = abalone_lda_all_test["Rings"]
```

```
C:\Users\ibteh\AppData\Local\Temp/ipykernel_10572/1847642409.py:3: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guid
e/indexing.html#returning-a-view-versus-a-copy
  abalone_lda_train["Rings"] = abalone_lda_all_train["Rings"]
C:\Users\ibteh\AppData\Local\Temp/ipykernel_10572/1847642409.py:7: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guid
e/indexing.html#returning-a-view-versus-a-copy
  abalone_lda_test["Rings"] = abalone_lda_all_test["Rings"]
```

1.3.2 Wine Dataset

Now we will use the kNN classifier to see what is the performance with different number of principal components

In [ ]:
```
# initialize for loop to generate the kNN classifier and its respective dataset for  "i"  pr
for i in range(2,wine_lda_all_train.shape[1]):

    # generate kNN classifier for each set of principal component ranging from 2 to 11
    globals()[f"wine_lda_knn_{i}"] = KNeighborsClassifier(n_neighbors=124, weights="distance

    # train each classifier with LDA training data with the respective number of principal c
    globals()[f"wine_lda_knn_{i}"].fit( wine_lda_all_train.iloc[:,0:i], wine_lda_all_train["
    #print(globals()[f"wine_lda_knn_{i}"])
```

In [ ]:
```
# initialize the array to store the test scores of the different classifiers
wine_lda_knn_scores = []

# initialize the for loop to check the performance of the classifier on the LDA test data
for i in range(2,wine_lda_all_train.shape[1]):

    # fit the LDA test data with the respective features subset on the classifier and store
    wine_lda_knn_scores.append(globals()[f"wine_lda_knn_{i}"].score(wine_lda_all_test.iloc[:
```

The number of components required to achieve the maximum accuracy is shown below:

In [ ]:
```
print("Number of Components required for maximum accuracy: " + str(wine_lda_knn_scores.index
```

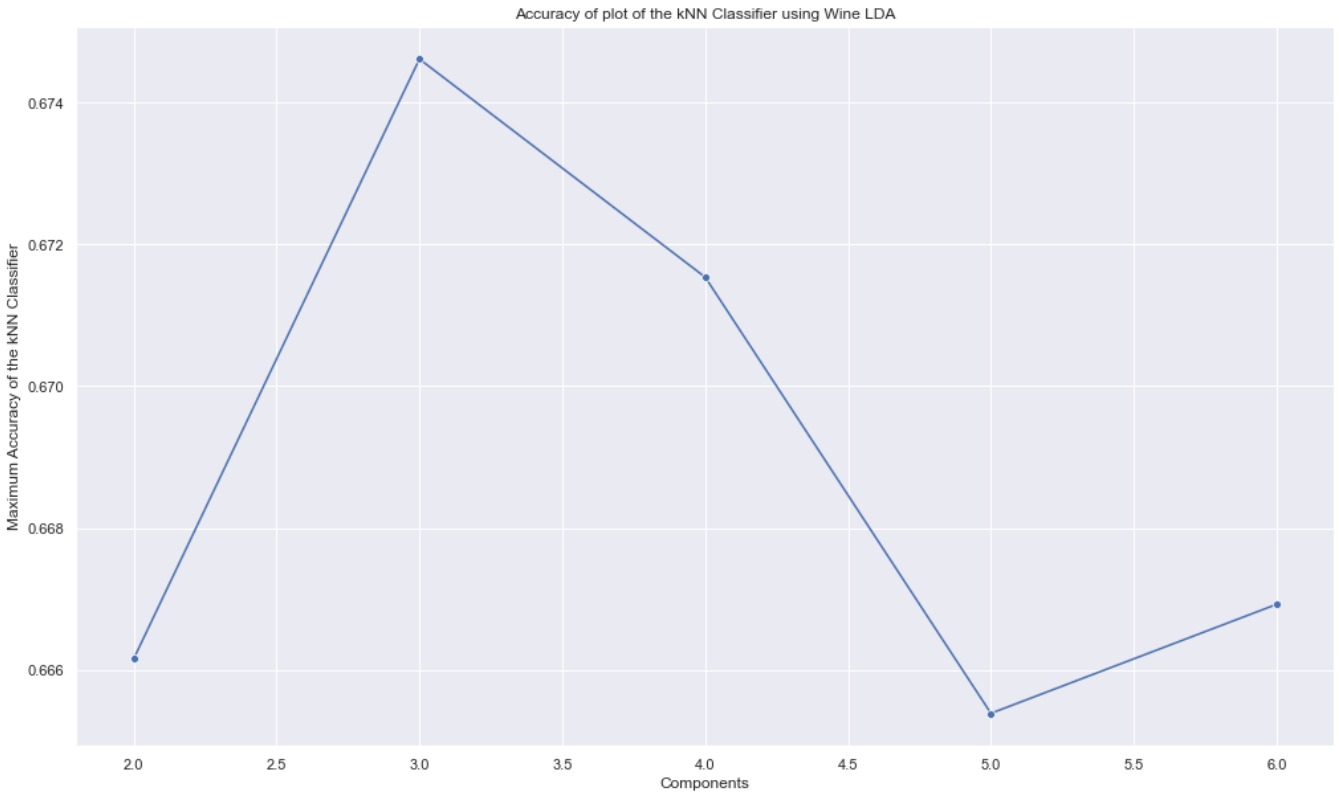
```
Number of Components required for maximum accuracy: 3
```

Now we will plot the graph of the accuracy on the y-axis and the number of principal components on the x-axis

In [ ]:
```
# initialize the plot
sns.set(rc = {'figure.figsize':(17,10)})

# provide the necessary parameters for the plot
sns.lineplot( x=(range(2, (len(wine_lda_knn_scores)+2))), y=wine_lda_knn_scores,marker="o")

# assign the x label for the graph
plt.xlabel("Components")

# assign the y label for the graph
plt.ylabel("Maximum Accuracy of the kNN Classifier")
```

```
# assign the title for the graph
plt.title("Accuracy of plot of the kNN Classifier using Wine LDA")
```

Out[ ]:
```
Text(0.5, 1.0, 'Accuracy of plot of the kNN Classifier using Wine LDA')
```



Now we will generate the training and testing dataset which will be used for the remaining questions

In [ ]:
```
# generate the training set
wine_lda_train = wine_lda_all_train.iloc[:,0:3]
wine_lda_train["Quality"] = wine_lda_all_train["Quality"]

# generate the testing set
wine_lda_test = wine_lda_all_test.iloc[:,0:3]
wine_lda_test["Quality"] = wine_lda_all_test["Quality"]
```

```
C:\Users\ibteh\AppData\Local\Temp/ipykernel_10572/48904243.py:3: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guid
e/indexing.html#returning-a-view-versus-a-copy
  wine_lda_train["Quality"] = wine_lda_all_train["Quality"]
C:\Users\ibteh\AppData\Local\Temp/ipykernel_10572/48904243.py:7: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guid
e/indexing.html#returning-a-view-versus-a-copy
  wine_lda_test["Quality"] = wine_lda_all_test["Quality"]
```

Question 2: Naive Bayes Classifier

2.1

2.1.1 Abalone Dataset

Multinomial Naive Bayes using Normalized Data

We will first conduct the cross validation using the Multinomial Naive Bayes using the training set and [illegible]hown below:

In [ ]:
```
# cross validation of the Multinomial Naive Bayes classifier using raw abalone training datas
abalone_raw_mnb_train_cv = cross_validate(MultinomialNB(), abalone_raw_train, abalone_rings_r

# mean validation accuracy
abalone_raw_mnb_train_cv_score = np.mean(abalone_raw_mnb_train_cv["test_score"])
abalone_raw_mnb_train_cv_score
```

```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\model_select
ion\_split.py:676: UserWarning: The least populated class in y has only 1 members, which is l
ess than n_splits=5.
  warnings.warn(
```

Out[ ]:
```
0.16701619183158348
```

Now we will initialize a multinomial Naive Bayes classifier and train it with training set as shown below:

In [ ]:
```
# initialize the Multinomial Naive Bayes Classifier for the raw abalone dataset
abalone_raw_mnb = MultinomialNB()

# fit the raw abalone training data into the classifier
abalone_raw_mnb.fit(abalone_raw_train, abalone_rings_raw_train)
```

Out[ ]:
```
MultinomialNB()
```

Now we will use the testing dataset to see the performance as shown below:

In [ ]:
```
# check the performance of the Multinomial Naive Bayes Classifier using the raw testing datas
abalone_raw_mnb_cscore_testing = abalone_raw_mnb.score(abalone_raw_test, abalone_rings_raw_te
abalone_raw_mnb_cscore_testing
```

Out[ ]:
```
0.15789473684210525
```

Complement Naive Bayes using Normalized Data

We will first conduct the cross validation using the Complement Naive Bayes using the training set and generate the mean validation accuracy as shown below:

In [ ]:
```
# cross validation of the Complement Naive Bayes classifier using raw abalone training datase
abalone_raw_cnb_train_cv = cross_validate(ComplementNB(), abalone_raw_train, abalone_rings_ra

# mean validation accuracy
abalone_raw_cnb_train_cv_score = np.mean(abalone_raw_cnb_train_cv["test_score"])
abalone_raw_cnb_train_cv_score
```

```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\model_select
ion\_split.py:676: UserWarning: The least populated class in y has only 1 members, which is l
ess than n_splits=5.
  warnings.warn(
```

Out[ ]:
```
0.1735972002183973
```

Now we will initialize a Complement Naive Bayes classifier and train it with training set as shown below:

In [ ]:
```
# initialize the Complement Naive Bayes Classifier for the raw abalone dataset
abalone_raw_cnb = ComplementNB()

# fit the raw abalone training data into the classifier
abalone_raw_cnb.fit(abalone_raw_train, abalone_rings_raw_train)
```

Out[ ]:
```
ComplementNB()
```

Now we will use the testing dataset to see the performance as shown below:

In [ ]:
```
# check the performance of the Complement Naive Bayes Classifier using the raw testing datase
```

```
abalone_raw_cnb_cscore_testing = abalone_raw_cnb.score(abalone_raw_test, abalone_rings_raw_t
abalone_raw_cnb_cscore_testing
```

Out[ ]:
```
0.1674641148325359
```

Multinomial Naive Bayes using PCA

Since the PCA dataset contains negative values, we will apply Min-Max Normalization to the PCA components again as shown below:

In [ ]:
```
# drop the ring feature from the PCA train dataset
abalone_pca_train_x = abalone_pca_train.drop(columns=["Rings"])

# drop the ring feature from the PCA test dataset
abalone_pca_test_x = abalone_pca_test.drop(columns=["Rings"])

# Conduct the Minimum - Maximum Normalization on the PCA dataset
abalone_PCAminmax_train = ((abalone_pca_train_x-abalone_pca_train_x.min())/(abalone_pca_trai
abalone_rings_PCAminmax_train = abalone_pca_train["Rings"]


abalone_PCAminmax_test = ((abalone_pca_test_x-abalone_pca_test_x.min())/(abalone_pca_test_x.
abalone_rings_PCAminmax_test = abalone_pca_test["Rings"]
```

We will first conduct the cross validation using the Multinomial Naive Bayes using the training set and generate the mean validation accuracy as shown below:

In [ ]:
```
# cross validation of the Multinomial Naive Bayes classifier using raw abalone training datas
abalone_pca_mnb_train_cv = cross_validate(MultinomialNB(),abalone_PCAminmax_train , abalone_

# mean validation accuracy
abalone_pca_mnb_train_cv_score = np.mean(abalone_pca_mnb_train_cv["test_score"])
abalone_pca_mnb_train_cv_score
```

Out[ ]:
```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\model_select
ion\_split.py:676: UserWarning: The least populated class in y has only 1 members, which is l
ess than n_splits=5.
  warnings.warn(
0.16671679063397868
```

Now we will initialize a multinomial Naive Bayes classifier and train it with training set as shown below:

In [ ]:
```
# initialize the Multinomial Naive Bayes Classifier for the pca abalone dataset
abalone_pca_mnb = MultinomialNB()

# fit the pca abalone training data into the classifier
abalone_pca_mnb.fit(abalone_PCAminmax_train , abalone_rings_PCAminmax_train)
```

Out[ ]:
```
MultinomialNB()
```

Now we will use the testing dataset to see the performance as shown below:

In [ ]:
```
# check the performance of the Multinomial Naive Bayes Classifier using the PCA testing datas
abalone_pca_mnb_cscore_testing = abalone_pca_mnb.score(abalone_PCAminmax_test , abalone_rings
abalone_pca_mnb_cscore_testing
```

Out[ ]:
```
0.15789473684210525
```

Complement Naive Bayes using PCA

In [ ]:
```
t Naive Bayes classifier using pca abalone training datase
abalone_pca_cnb_train_cv = cross_validate(ComplementNB(),abalone_PCAminmax_train , abalone_r
```

```
# mean validation accuracy
abalone_pca_cnb_train_cv_score = np.mean(abalone_pca_cnb_train_cv["test_score"])
abalone_pca_cnb_train_cv_score
```

```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\model_select
ion\_split.py:676: UserWarning: The least populated class in y has only 1 members, which is l
ess than n_splits=5.
  warnings.warn(
```

Out[ ]:
```
0.1786829927588769
```

Now we will initialize a Complement Naive Bayes classifier and train it with training set as shown below:

In [ ]:
```
# initialize the Complement Naive Bayes Classifier for the raw abalone dataset
abalone_pca_cnb = ComplementNB()

# fit the raw abalone training data into the classifier
abalone_pca_cnb.fit(abalone_PCAminmax_train , abalone_rings_PCAminmax_train)
```

Out[ ]:
```
ComplementNB()
```

Now we will use the testing dataset to see the performance as shown below:

In [ ]:
```
# check the performance of the Complement Naive Bayes Classifier using the PCA testing datase
abalone_pca_cnb_cscore_testing = abalone_pca_cnb.score(abalone_PCAminmax_test , abalone_rings
abalone_pca_cnb_cscore_testing
```

Out[ ]:
```
0.16148325358851676
```

Multinomial Naive Bayes using LDA

Since some of the LDA values are negative, we will use Minimum Maximum Normalization to remove negative values from the data as shown below:

In [ ]:
```
# drop the ring feature from the PCA train dataset
abalone_lda_train_x = abalone_lda_train.drop(columns=["Rings"])

# drop the ring feature from the lda test dataset
abalone_lda_test_x = abalone_lda_test.drop(columns=["Rings"])

# Conduct the Minimum - Maximum Normalization on the PCA dataset
abalone_LDAminmax_train = ((abalone_lda_train_x-abalone_lda_train_x.min())/(abalone_lda_trai
abalone_rings_LDAminmax_train = abalone_lda_train["Rings"]


abalone_LDAminmax_test = ((abalone_lda_test_x-abalone_lda_test_x.min())/(abalone_lda_test_x.
abalone_rings_LDAminmax_test = abalone_lda_test["Rings"]
```

We will first conduct the cross validation using the Multinomial Naive Bayes using the training set and generate the mean validation accuracy as shown below:

In [ ]:
```
# cross validation of the Multinomial Naive Bayes classifier using raw abalone training datas
abalone_lda_mnb_train_cv = cross_validate(MultinomialNB(),abalone_LDAminmax_train , abalone_
# mean validation accuracy
abalone_lda_mnb_train_cv_score = np.mean(abalone_lda_mnb_train_cv["test_score"])
abalone_lda_mnb_train_cv_score
```

```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\model_select
ion\_split.py:676: UserWarning: The least populated class in y has only 1 members, which is l
ess than n_splits=5.
```

Out[ ]:
```
0.16671679063397868
```

Now we will initialize a multinomial Naive Bayes classifier and train it with training set as shown below:

In [ ]:
```
# initialize the Multinomial Naive Bayes Classifier for the pca abalone dataset
abalone_lda_mnb = MultinomialNB()

# fit the lda abalone training data into the classifier
abalone_lda_mnb.fit(abalone_LDAminmax_train , abalone_rings_LDAminmax_train)
```

Out[ ]:
```
MultinomialNB()
```

Now we will use the testing dataset to see the performance as shown below:

In [ ]:
```
# check the performance of the Multinomial Naive Bayes Classifier using the LDA testing datas
abalone_lda_mnb_cscore_testing = abalone_lda_mnb.score(abalone_LDAminmax_test , abalone_rings
abalone_lda_mnb_cscore_testing
```

Out[ ]:
```
0.15789473684210525
```

Complement Naive Bayes using LDA

In [ ]:
```
# cross validation of the Complement Naive Bayes classifier using LDA abalone training datase
abalone_lda_cnb_train_cv = cross_validate(ComplementNB(),abalone_LDAminmax_train , abalone_r

# mean validation accuracy
abalone_lda_cnb_train_cv_score = np.mean(abalone_lda_cnb_train_cv["test_score"])
abalone_lda_cnb_train_cv_score
```

Out[ ]:
```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\model_select
ion\_split.py:676: UserWarning: The least populated class in y has only 1 members, which is l
ess than n_splits=5.
  warnings.warn(
0.243335302489192
```

Now we will initialize a Complement Naive Bayes classifier and train it with training set as shown below:

In [ ]:
```
# initialize the Complement Naive Bayes Classifier for the LDA abalone dataset
abalone_lda_cnb = ComplementNB()

# fit the raw abalone training data into the classifier
abalone_lda_cnb.fit(abalone_LDAminmax_train , abalone_rings_LDAminmax_train)
```

Out[ ]:
```
ComplementNB()
```

Now we will use the testing dataset to see the performance as shown below:

In [ ]:
```
# check the performance of the Complement Naive Bayes Classifier using the LDA testing datase
abalone_lda_cnb_cscore_testing = abalone_lda_cnb.score(abalone_LDAminmax_test , abalone_rings
abalone_lda_cnb_cscore_testing
```

Out[ ]:
```
0.07177033492822966
```

Summary of the Results

We are going to generate a table that is going to display the accuracy of the algorithms on the test data as shown below:

In [ ]:
```
# initialize dataframe
                                                                    e()
```

```
# make the index column
abalone_summary_table["Method"]= ["Multinomial NB", "Complement NB", "kNN"]
abalone_summary_table.set_index("Method",inplace=True)

# add the relevant data
abalone_summary_table["Raw (Mean Validation Accuracy)"] = [abalone_raw_mnb_train_cv_score,aba
abalone_summary_table["Raw (Test Accuracy)"] = [abalone_raw_mnb_cscore_testing,abalone_raw_cr
abalone_summary_table["PCA (Mean Validation Accuracy)"] = [abalone_pca_mnb_train_cv_score,aba
abalone_summary_table["PCA (Test Accuracy)"] = [abalone_pca_mnb_cscore_testing,abalone_pca_cr
abalone_summary_table["LDA (Mean Validation Accuracy)"] = [abalone_lda_mnb_train_cv_score,aba
abalone_summary_table["LDA (Test Accuracy)"] = [abalone_lda_mnb_cscore_testing,abalone_lda_cr
```

In [ ]:
```
abalone_summary_table
```

Out[ ]:

Method	Raw (Mean Validation Accuracy)	Raw (Test Accuracy)	PCA (Mean Validation Accuracy)	PCA (Test Accuracy)	LDA (Mean Validation Accuracy)	LDA (Test Accuracy)
Multinomial NB	0.167016	0.157895	0.166717	0.157895	0.166717	0.157895
Complement NB	0.173597	0.167464	0.178683	0.161483	0.243335	0.071770
kNN	None	0.276300	None	0.270335	None	0.287081

As we did not have to run cross validation for the mean validation accuracy, that is shown by "None" in the table above.

2.1.2 Wine Dataset

Multinomial Naive Bayes using Normalized Data

We will first conduct the cross validation using the Multinomial Naive Bayes using the training set and generate the mean validation accuracy as shown below:

In [ ]:
```
# cross validation of the Multinomial Naive Bayes classifier using raw wine training dataset
wine_raw_mnb_train_cv = cross_validate(MultinomialNB(), wine_raw_train, wine_quality_raw_tra

# mean validation accuracy
wine_raw_mnb_train_cv_score = np.mean(wine_raw_mnb_train_cv["test_score"])
wine_raw_mnb_train_cv_score
```

```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\model_select
ion\_split.py:676: UserWarning: The least populated class in y has only 4 members, which is l
ess than n_splits=5.
  warnings.warn(
```

Out[ ]:
```
0.43602076700969866
```

Now we will initialize a multinomial Naive Bayes classifier and train it with training set as shown below:

In [ ]:
```
# initialize the Multinomial Naive Bayes Classifier for the raw wine dataset
wine_raw_mnb = MultinomialNB()

# fit the raw wine training data into the classifier
wine_raw_mnb.fit(wine_raw_train, wine_quality_raw_train)
```

Out[ ]:
```
MultinomialNB()
```

Now we will use the testing dataset to see the performance as shown below:

In [ ]:

```
# check the performance of the Multinomial Naive Bayes Classifier using the raw testing datas
wine_raw_mnb_cscore_testing = wine_raw_mnb.score(wine_raw_test, wine_quality_raw_test)
wine_raw_mnb_cscore_testing
```

Out[ ]:
```
0.43846153846153846
```

Complement Naive Bayes using Normalized Data

We will first conduct the cross validation using the Complement Naive Bayes using the training set and generate the mean validation accuracy as shown below:

In [ ]:
```
# cross validation of the Complement Naive Bayes classifier using raw wine training dataset
wine_raw_cnb_train_cv = cross_validate(ComplementNB(), wine_raw_train, wine_quality_raw_trai

# mean validation accuracy
wine_raw_cnb_train_cv_score = np.mean(wine_raw_cnb_train_cv["test_score"])
wine_raw_cnb_train_cv_score
```

Out[ ]:
```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\model_select
ion\_split.py:676: UserWarning: The least populated class in y has only 4 members, which is l
ess than n_splits=5.
  warnings.warn(
0.43390112534241504
```

Now we will initialize a Complement Naive Bayes classifier and train it with training set as shown below:

In [ ]:
```
# initialize the Complement Naive Bayes Classifier for the raw wine dataset
wine_raw_cnb = ComplementNB()

# fit the raw wine training data into the classifier
wine_raw_cnb.fit(wine_raw_train, wine_quality_raw_train)
```

Out[ ]:
```
ComplementNB()
```

Now we will use the testing dataset to see the performance as shown below:

In [ ]:
```
# check the performance of the Complement Naive Bayes Classifier using the raw testing datase
wine_raw_cnb_cscore_testing = wine_raw_cnb.score(wine_raw_test, wine_quality_raw_test)
wine_raw_cnb_cscore_testing
```

Out[ ]:
```
0.4276923076923077
```

Multinomial Naive Bayes using PCA

Since the PCA dataset contains negative values, we will apply Min-Max Normalization to the PCA components again as shown below:

In [ ]:
```
# drop the quality feature from the PCA train dataset
wine_pca_train_x = wine_pca_train.drop(columns=["Quality"])

# drop the quality feature from the PCA test dataset
wine_pca_test_x = wine_pca_test.drop(columns=["Quality"])

# Conduct the Minimum - Maximum Normalization on the PCA dataset
wine_PCAminmax_train = ((wine_pca_train_x-wine_pca_train_x.min())/(wine_pca_train_x.max()-wi
wine_quality_PCAminmax_train = wine_pca_train["Quality"]


wine_PCAminmax_test = ((wine_pca_test_x-wine_pca_test_x.min())/(wine_pca_test_x.max()-wine_p
wine_quality_PCAminmax_test = wine_pca_test["Quality"]
```


sing the Multinomial Naive Bayes using the training set and

generate the mean validation accuracy as shown below:

In [ ]:
```
# cross validation of the Multinomial Naive Bayes classifier using raw wine training dataset
wine_pca_mnb_train_cv = cross_validate(MultinomialNB(),wine_PCAminmax_train , wine_quality_PC

# mean validation accuracy
wine_pca_mnb_train_cv_score = np.mean(wine_pca_mnb_train_cv["test_score"])
wine_pca_mnb_train_cv_score
```

```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\model_select
ion\_split.py:676: UserWarning: The least populated class in y has only 4 members, which is l
ess than n_splits=5.
  warnings.warn(
```

Out[ ]:
```
0.43602076700969866
```

Now we will initialize a multinomial Naive Bayes classifier and train it with training set as shown below:

In [ ]:
```
# initialize the Multinomial Naive Bayes Classifier for the pca abalone dataset
wine_pca_mnb = MultinomialNB()

# fit the pca wine training data into the classifier
wine_pca_mnb.fit(wine_PCAminmax_train , wine_quality_PCAminmax_train)
```

Out[ ]:
```
MultinomialNB()
```

Now we will use the testing dataset to see the performance as shown below:

In [ ]:
```
# check the performance of the Multinomial Naive Bayes Classifier using the PCA testing datas
wine_pca_mnb_cscore_testing = wine_pca_mnb.score(wine_PCAminmax_test , wine_quality_PCAminmax
wine_pca_mnb_cscore_testing
```

Out[ ]:
```
0.43846153846153846
```

Complement Naive Bayes using PCA

In [ ]:
```
# cross validation of the Complement Naive Bayes classifier using pca wine training dataset
wine_pca_cnb_train_cv = cross_validate(ComplementNB(),wine_PCAminmax_train , wine_quality_PCA

# mean validation accuracy
wine_pca_cnb_train_cv_score = np.mean(wine_pca_cnb_train_cv["test_score"])
wine_pca_cnb_train_cv_score
```

```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\model_select
ion\_split.py:676: UserWarning: The least populated class in y has only 4 members, which is l
ess than n_splits=5.
  warnings.warn(
```

Out[ ]:
```
0.41138853927593094
```

Now we will initialize a Complement Naive Bayes classifier and train it with training set as shown below:

In [ ]:
```
# initialize the Complement Naive Bayes Classifier for the quality wine dataset
wine_pca_cnb = ComplementNB()

# fit the raw wine training data into the classifier
wine_pca_cnb.fit(wine_PCAminmax_train , wine_quality_PCAminmax_train)
```

Out[ ]:
```
ComplementNB()
```

Now we will use the testing dataset to see the performance as shown below:

In [ ]:
```
# check the performance of the Complement Naive Bayes Classifier using the PCA testing datase
```

```
wine_pca_cnb_cscore_testing = wine_pca_cnb.score(wine_PCAminmax_test , wine_quality_PCAminma
wine_pca_cnb_cscore_testing
```

Out[ ]:
```
0.36615384615384616
```

Multinomial Naive Bayes using LDA

Since some of the LDA values are negative, we will use Minimum Maximum Normalization to remove negative values from the data as shown below:

In [ ]:
```
# drop the ring feature from the LDA train dataset
wine_lda_train_x = wine_lda_train.drop(columns=["Quality"])

# drop the ring feature from the lda test dataset
wine_lda_test_x = wine_lda_test.drop(columns=["Quality"])

# Conduct the Minimum - Maximum Normalization on the LDA training dataset
wine_LDAminmax_train = ((wine_lda_train_x-wine_lda_train_x.min())/(wine_lda_train_x.max()-wi
wine_quality_LDAminmax_train = wine_lda_train["Quality"]

# Conduct the Minimum - Maximum Normalization on the LDA testing dataset
wine_LDAminmax_test = ((wine_lda_test_x-wine_lda_test_x.min())/(wine_lda_test_x.max()-wine_l
wine_quality_LDAminmax_test = wine_lda_test["Quality"]
```

We will first conduct the cross validation using the Multinomial Naive Bayes using the training set and generate the mean validation accuracy as shown below:

In [ ]:
```
# cross validation of the Multinomial Naive Bayes classifier using raw wine training dataset
wine_lda_mnb_train_cv = cross_validate(MultinomialNB(),wine_LDAminmax_train , wine_quality_LD

# mean validation accuracy
wine_lda_mnb_train_cv_score = np.mean(wine_lda_mnb_train_cv["test_score"])
wine_lda_mnb_train_cv_score
```

Out[ ]:
```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\model_select
ion\_split.py:676: UserWarning: The least populated class in y has only 4 members, which is l
ess than n_splits=5.
  warnings.warn(
0.43602076700969866
```

Now we will initialize a multinomial Naive Bayes classifier and train it with training set as shown below:

In [ ]:
```
# initialize the Multinomial Naive Bayes Classifier for the LDA wine dataset
wine_lda_mnb = MultinomialNB()

# fit the lda wine training data into the classifier
wine_lda_mnb.fit(wine_LDAminmax_train , wine_quality_LDAminmax_train)
```

Out[ ]:
```
MultinomialNB()
```

Now we will use the testing dataset to see the performance as shown below:

In [ ]:
```
# check the performance of the Multinomial Naive Bayes Classifier using the LDA testing datas
wine_lda_mnb_cscore_testing = wine_lda_mnb.score(wine_LDAminmax_test , wine_quality_LDAminma
wine_lda_mnb_cscore_testing
```

Out[ ]:
```
0.43846153846153846
```

Complement Naive Bayes using LDA

In [ ]:
```
t Naive Bayes classifier using LDA wine training dataset
wine_lda_cnb_train_cv = cross_validate(ComplementNB(),wine_LDAminmax_train , wine_quality_LD
```

```
# mean validation accuracy
wine_lda_cnb_train_cv_score = np.mean(wine_lda_cnb_train_cv["test_score"])
wine_lda_cnb_train_cv_score
```

```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\model_selection\_split.py:676: UserWarning: The least populated class in y has only 4 members, which is less than n_splits=5.
  warnings.warn(
```

Out[ ]:
```
0.4262062264011254
```

Now we will initialize a Complement Naive Bayes classifier and train it with training set as shown below:

In [ ]:
```
# initialize the Complement Naive Bayes Classifier for the LDA wine dataset
wine_lda_cnb = ComplementNB()

# fit the raw wine training data into the classifier
wine_lda_cnb.fit(wine_LDAminmax_train , wine_quality_LDAminmax_train)
```

Out[ ]:
```
ComplementNB()
```

Now we will use the testing dataset to see the performance as shown below:

In [ ]:
```
# check the performance of the Complement Naive Bayes Classifier using the LDA testing datase
wine_lda_cnb_cscore_testing = wine_lda_cnb.score(wine_LDAminmax_test , wine_quality_LDAminma
wine_lda_cnb_cscore_testing
```

Out[ ]:
```
0.3861538461538462
```

Summary of the Results

We are going to generate a table that is going to display the accuracy of the algorithms on the test data as shown below:

In [ ]:
```
# initialize dataframe
wine_summary_table = pd.DataFrame()

# make the index column
wine_summary_table["Method"]= ["Multinomial NB", "Complement NB", "kNN"]
wine_summary_table.set_index("Method",inplace=True)

# add the relevant data
# There are no mean validation accuracy for kNN as cross_validate was not executed for it
wine_summary_table["Raw (Mean Validation Accuracy)"] = [wine_raw_mnb_train_cv_score,wine_raw
wine_summary_table["Raw (Test Accuracy)"] = [wine_raw_mnb_cscore_testing,wine_raw_cnb_cscore
wine_summary_table["PCA (Mean Validation Accuracy)"] = [wine_pca_mnb_train_cv_score,wine_pca
wine_summary_table["PCA (Test Accuracy)"] = [wine_pca_mnb_cscore_testing,wine_pca_cnb_cscore
wine_summary_table["LDA (Mean Validation Accuracy)"] = [wine_lda_mnb_train_cv_score,wine_lda
wine_summary_table["LDA (Test Accuracy)"] = [wine_lda_mnb_cscore_testing,wine_lda_cnb_cscore
```

In [ ]:
```
wine_summary_table
```

Out[ ]:

Method	Raw (Mean Validation Accuracy)	Raw (Test Accuracy)	PCA (Mean Validation Accuracy)	PCA (Test Accuracy)	LDA (Mean Validation Accuracy)	LDA (Test Accuracy)
Multinomial NB	0.436021	0.438462	0.436021	0.438462	0.436021	0.438462
Complement NB	0.433901	0.427692	0.411389	0.366154	0.426206	0.386154
kNN	None	0.682300	None	0.683846	None	0.674615

As we did not have to run cross validation for the mean validation accuracy, that is shown by "None" in the table above.

2.2

In complement Naive Bayes, instead of calculating the probability of an item belonging to a certain class, we calculate the probability of the item belonging to all the other classes. This results in getting more information to determine the classification of the perticular instance.

For the **Abalone** dataset, for all the three datasets(raw, PCA & LDA), complement naive bayes performed better than multinomial naive bayes.

For the **Wine** dataset, this was not the case. Multinomial Naive Bayes performed better in all the cases.

Question 3: Decision Tree Classifier

3.1 Decision Tree using Cross Validation

3.1.1 Abalone Dataset

First we will determine the maximum depth that is possible for the abalone dataset using the decision tree classifier which is shown below:

In [ ]:

```
abalone_dt_depth_check = DecisionTreeClassifier(random_state=27)
abalone_dt_depth_check.fit(abalone_PCAminmax_train , abalone_rings_PCAminmax_train)
abalone_dt_depth_check.tree_.max_depth
```

Out[ ]:

```
27
```

The limit of the maximum depth for the decision tree will be **27**

3.1.1.1 Normalized Data

First we will apply the normalized dataset as shown below:

In [ ]:

```
# array for the maximum depth of the tree from 1 to 27
decision_tree_depth = list(range(1,28))

# declare the paremeters for the decision tree classifier
parameters = {"max_depth":decision_tree_depth}

# initialize the GridSearchCV
abalone_raw_tree_cv = GridSearchCV(DecisionTreeClassifier(random_state=27), parameters, cv=5

# fit the normalized data
abalone_raw_tree_cv.fit(abalone_raw_train, abalone_rings_raw_train)

# view the result of the cross validation on the decision tree algorithm
```

```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\model_selection\_split.py:676: UserWarning: The least populated class in y has only 1 members, which is less than n_splits=5.
  warnings.warn(
```

Out[ ]:

```
{'mean_fit_time': array([0.00859566, 0.0117938 , 0.01279812, 0.0351974 , 0.01499724,
       0.01899676, 0.01980157, 0.02279634, 0.02639532, 0.02619643,
       0.03019972, 0.0309948 , 0.03379798, 0.03479619, 0.03319936,
       0.04659152, 0.03619828, 0.03599339, 0.03759494, 0.03739657,
       0.05159636, 0.04099698, 0.04019485, 0.03759551, 0.03939877,
       0.04359407, 0.04159474]),
 'std_fit_time': array([0.00135654, 0.00160286, 0.00172156, 0.012938  , 0.000634  ,
       0.00260797, 0.00264532, 0.00172081, 0.00392953, 0.0014703 ,
       0.00231426, 0.00167408, 0.00203906, 0.00299516, 0.00160107,
       0.0124956 , 0.00248558, 0.00357864, 0.00546063, 0.00500786,
       0.01523281, 0.00543883, 0.00371046, 0.00286982, 0.00445963,
       0.00716632, 0.00101858]),
 'mean_score_time': array([0.00420618, 0.0038034 , 0.00300331, 0.00920434, 0.00380363,
       0.00380363, 0.00279856, 0.00300217, 0.00320687, 0.0028029 ,
       0.00280166, 0.00580382, 0.00300212, 0.00320554, 0.00459957,
       0.00300384, 0.00380425, 0.00420518, 0.00360579, 0.00340171,
       0.00540419, 0.00300279, 0.00420671, 0.00440583, 0.00300136,
       0.00380316, 0.00320463]),
 'std_score_time': array([1.16836426e-03, 7.49004367e-04, 6.30600824e-04, 7.30699365e-03,
       2.63913598e-03, 7.50791100e-04, 3.97629500e-04, 6.33240123e-04,
       4.02053008e-04, 4.00687867e-04, 4.01021269e-04, 6.11388236e-03,
       3.67937884e-06, 9.84290122e-04, 1.85510925e-03, 6.34371761e-04,
       1.16672090e-03, 2.40197237e-03, 1.74665528e-03, 1.02056403e-03,
       2.05920080e-03, 3.18661393e-06, 1.46886704e-03, 1.74418377e-03,
       6.32039687e-04, 1.16575527e-03, 4.01241438e-04]),
 'param_max_depth': masked_array(data=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
                    17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False],
       fill_value='?',
            dtype=object),
 'params': [{'max_depth': 1},
  {'max_depth': 2},
  {'max_depth': 3},
  {'max_depth': 4},
  {'max_depth': 5},
  {'max_depth': 6},
  {'max_depth': 7},
  {'max_depth': 8},
  {'max_depth': 9},
  {'max_depth': 10},
  {'max_depth': 11},
  {'max_depth': 12},
  {'max_depth': 13},
  {'max_depth': 14},
  {'max_depth': 15},
  {'max_depth': 16},
  {'max_depth': 17},
  {'max_depth': 18},
  {'max_depth': 19},
  {'max_depth': 20},
  {'max_depth': 21},
  {'max_depth': 22},
  {'max_depth': 23},
  {'max_depth': 24},
  {'max_depth': 25},
  {'max_depth': 26},
  {'max_depth': 27}],
 'split0_test_score': array([0.20627803, 0.23916293, 0.26606876, 0.28550075, 0.29147982,
       0.27204783, 0.24962631, 0.25411061, 0.23766816, 0.23168909,
       0.24663677, 0.23019432, 0.24215247, 0.23318386, 0.21524664,
       0.20926756, 0.20328849, 0.19431988, 0.20179372, 0.18834081,
       0.18983558, 0.1838565 , 0.19282511, 0.20029895, 0.20029895,
       0.20029895, 0.20029895]),
 'split1_test_score': array([0.21107784, 0.23203593, 0.26047904, 0.27245509, 0.25149701,
       0.24251497, 0.23053892, 0.24700599, 0.24700599,
       0.21856287, 0.21107784, 0.21556886, 0.23802395, 0.21257485,
```

```
        0.19610778, 0.17365269, 0.18562874, 0.19011976, 0.20658683,
        0.2005988 , 0.20808383, 0.21107784, 0.20508982, 0.21257485,
        0.20658683, 0.20658683]),
 'split2_test_score': array([0.20359281, 0.23353293, 0.23502994, 0.25748503, 0.26646707,
        0.26646707, 0.24850299, 0.23952096, 0.24101796, 0.23203593,
        0.21856287, 0.22005988, 0.22754491, 0.20508982, 0.2005988 ,
        0.20658683, 0.21107784, 0.21107784, 0.20209581, 0.20958084,
        0.20209581, 0.1991018 , 0.20508982, 0.21107784, 0.21107784,
        0.21107784, 0.21107784]),
 'split3_test_score': array([0.21107784, 0.24700599, 0.26047904, 0.27694611, 0.26646707,
        0.27245509, 0.25      , 0.24401198, 0.23203593, 0.24101796,
        0.21257485, 0.20808383, 0.22005988, 0.21706587, 0.18712575,
        0.20658683, 0.2005988 , 0.20209581, 0.21556886, 0.20808383,
        0.19760479, 0.19461078, 0.19011976, 0.19011976, 0.19011976,
        0.19461078, 0.19461078]),
 'split4_test_score': array([0.20209581, 0.25449102, 0.25748503, 0.23353293, 0.26047904,
        0.25149701, 0.2260479 , 0.20958084, 0.21257485, 0.23203593,
        0.20808383, 0.21556886, 0.21107784, 0.21257485, 0.20359281,
        0.19311377, 0.19461078, 0.1991018 , 0.20658683, 0.19610778,
        0.19610778, 0.19311377, 0.20508982, 0.20359281, 0.1991018 ,
        0.2005988 , 0.20508982]),
 'mean_test_score': array([0.20682447, 0.24124576, 0.25590836, 0.26518398, 0.267278  ,
        0.26608621, 0.24513484, 0.23555266, 0.23406058, 0.23675698,
        0.22088424, 0.21699695, 0.22328079, 0.22118767, 0.20382777,
        0.20233255, 0.19664572, 0.19844481, 0.203233  , 0.20174002,
        0.19724855, 0.19575334, 0.20084047, 0.20203584, 0.20263464,
        0.20263464, 0.20353284]),
 'std_test_score': array([0.00372249, 0.00845298, 0.01080195, 0.01824856, 0.01328389,
        0.0076501 , 0.00959141, 0.01504246, 0.01178761, 0.00622026,
        0.01346749, 0.00774688, 0.01089112, 0.0124714 , 0.00996334,
        0.0064502 , 0.01265591, 0.00842166, 0.00823096, 0.00820335,
        0.0042685 , 0.00791278, 0.00800122, 0.00690679, 0.00830246,
        0.00567256, 0.00563452]),
 'rank_test_score': array([14,  6,  4,  3,  1,  2,  5,  8,  9,  7, 12, 13, 10, 11, 15, 20, 2
6,
        24, 17, 22, 25, 27, 23, 21, 18, 18, 16])}
```

The best hyperparameters for the decision tree classifier are:

In [ ]:
```
abalone_raw_tree_cv.best_params_
```

Out[ ]:
```
{'max_depth': 5}
```

The score achieved by the best hyperparameters is:

In [ ]:
```
abalone_raw_tree_cv.best_score_
```

Out[ ]:
```
0.2672780000537043
```

Now we will initialize a new decision tree and train it with the best hyperparameters that were discovered by GridSearchCV

In [ ]:
```
# initialize the decision tree classifier with the best hyperparemeter
abalone_raw_tree = DecisionTreeClassifier(random_state=27, max_depth= 5)

# train the classifier using the training dataset
abalone_raw_tree.fit(abalone_raw_train, abalone_rings_raw_train)

# view the accuracy of the classifier using the testing dataset
abalone_raw_tree_test_accuracy = abalone_raw_tree.score(abalone_raw_test, abalone_rings_raw_
abalone_raw_tree_test_accuracy
```

Out[ ]:
```
0.24282296650717702
```


3.1.1.2 PCA Dataset

Now we will conduct the same procedure on PCA Abalone dataset

In [ ]:

```
# array for the maximum depth of the tree from 1 to 27
decision_tree_depth = list(range(1,28))

# declare the paremeters for the decision tree classifier
parameters = {"max_depth":decision_tree_depth}

# initialize the GridSearchCV
abalone_pca_tree_cv = GridSearchCV(DecisionTreeClassifier(random_state=27), parameters, cv=5

# fit the pca data
abalone_pca_tree_cv.fit(abalone_PCAminmax_train , abalone_rings_PCAminmax_train)

# view the result of the cross validation on the decision tree algorithm
abalone_pca_tree_cv.cv_results_
```

```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\model_select
ion\_split.py:676: UserWarning: The least populated class in y has only 1 members, which is l
ess than n_splits=5.
  warnings.warn(
```

```
Out[ ]: {'mean_fit_time': array([0.01219778, 0.02139974, 0.01759748, 0.02619753, 0.02539907,
        0.04160171, 0.03760071, 0.0470046 , 0.05239878, 0.0826005 ,
        0.0630003 , 0.04480028, 0.04059744, 0.04519849, 0.04079905,
        0.04839544, 0.05379815, 0.05920286, 0.06220407, 0.05999489,
        0.06679845, 0.06199999, 0.07040014, 0.06339674, 0.04999843,
        0.04780006, 0.05639629]),
 'std_fit_time': array([0.00354252, 0.00567747, 0.00344356, 0.00348715, 0.00467336,
        0.02727254, 0.0033823 , 0.01161495, 0.01207327, 0.0214125 ,
        0.01193171, 0.00910885, 0.00279882, 0.00271287, 0.00614475,
        0.0220732 , 0.00943195, 0.02486192, 0.00479084, 0.01205353,
        0.01133607, 0.01019522, 0.0143628 , 0.01204232, 0.00632271,
        0.00515385, 0.01172368]),
 'mean_score_time': array([0.00760384, 0.00700188, 0.00860162, 0.00960188, 0.00720196,
        0.00639853, 0.00800109, 0.01399474, 0.01180472, 0.00839939,
        0.01420059, 0.0068006 , 0.00760221, 0.00740118, 0.00600114,
        0.01060429, 0.00620317, 0.00959687, 0.01079187, 0.00720248,
        0.01120734, 0.00860219, 0.00860138, 0.00760641, 0.00800314,
        0.00660291, 0.00860157]),
 'std_score_time': array([0.0016243 , 0.00141223, 0.00079844, 0.00557176, 0.00116392,
        0.00135652, 0.00303216, 0.00880868, 0.00515276, 0.00272359,
        0.01690513, 0.00097933, 0.0024178 , 0.00241566, 0.00089794,
        0.00842829, 0.00160021, 0.00665002, 0.00248255, 0.00193914,
        0.00462281, 0.00294309, 0.00162665, 0.00350367, 0.00126663,
        0.00185324, 0.00326309]),
 'param_max_depth': masked_array(data=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
                    17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False],
        fill_value='?',
             dtype=object),
 'params': [{'max_depth': 1},
  {'max_depth': 2},
  {'max_depth': 3},
  {'max_depth': 4},
  {'max_depth': 5},
  {'max_depth': 6},
  {'max_depth': 7},
  {'max_depth': 8},
  {'max_depth': 9},
  {'max_depth': 10},
  {'max_depth': 11},
  {'max_depth': 12},
  {'max_depth': 13},
  {'max_depth': 14},
  {'max_depth': 15},
  {'max_depth': 16},
  {'max_depth': 17},
  {'max_depth': 18},
  {'max_depth': 19},
  {'max_depth': 20},
  {'max_depth': 21},
  {'max_depth': 22},
  {'max_depth': 23},
  {'max_depth': 24},
  {'max_depth': 25},
  {'max_depth': 26},
  {'max_depth': 27}],
 'split0_test_score': array([0.20478326, 0.22421525, 0.24663677, 0.25411061, 0.24813154,
        0.23617339, 0.23617339, 0.22122571, 0.2122571 , 0.2122571 ,
        0.20926756, 0.20627803, 0.19880419, 0.20029895, 0.20328849,
        0.19581465, 0.19581465, 0.20478326, 0.19581465, 0.20328849,
        0.20179372, 0.18535127, 0.1793722 , 0.18535127, 0.18834081,
        0.1838565 , 0.1838565 ]),
 'split1_test_score': array([0.20808383, 0.23802395, 0.25299401, 0.24550898, 0.2754491 ,
        0.25898204, 0.23353293, 0.21556886, 0.20359281, 0.21706587,
Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js 964072, 0.16167665, 0.16616766,
        0.16916168, 0.16467066, 0.16467066, 0.15419162, 0.15868263,
```

```
        0.16766467, 0.15568862, 0.1511976 , 0.16017964, 0.16167665,
        0.16467066, 0.16766467]),
 'split2_test_score': array([0.19461078, 0.21706587, 0.23802395, 0.2739521 , 0.25      ,
        0.24251497, 0.24251497, 0.22305389, 0.23652695, 0.24550898,
        0.22155689, 0.20508982, 0.21257485, 0.20808383, 0.21706587,
        0.2005988 , 0.20209581, 0.19610778, 0.19461078, 0.1991018 ,
        0.17964072, 0.19461078, 0.17365269, 0.17664671, 0.17365269,
        0.1751497 , 0.17964072]),
 'split3_test_score': array([0.20658683, 0.21407186, 0.24101796, 0.27694611, 0.25      ,
        0.23353293, 0.23952096, 0.25      , 0.25449102, 0.23353293,
        0.22904192, 0.21856287, 0.21257485, 0.21257485, 0.22904192,
        0.21706587, 0.20658683, 0.1991018 , 0.21407186, 0.1991018 ,
        0.20808383, 0.21107784, 0.20508982, 0.20209581, 0.20508982,
        0.20359281, 0.20658683]),
 'split4_test_score': array([0.20209581, 0.21856287, 0.25      , 0.25898204, 0.24550898,
        0.24550898, 0.21856287, 0.23952096, 0.22904192, 0.23203593,
        0.23203593, 0.21257485, 0.21107784, 0.21706587, 0.21556886,
        0.19760479, 0.22155689, 0.20359281, 0.20958084, 0.20958084,
        0.20658683, 0.20359281, 0.2005988 , 0.18562874, 0.19311377,
        0.19011976, 0.18562874]),
 'mean_test_score': array([0.2032321 , 0.22238796, 0.24573454, 0.26189997, 0.25381792,
        0.24334246, 0.23406103, 0.22987388, 0.22718196, 0.22808016,
        0.21999723, 0.20622567, 0.20293449, 0.19994003, 0.20622656,
        0.19604916, 0.19814497, 0.19365126, 0.19365395, 0.19395111,
        0.19275395, 0.19006427, 0.18198222, 0.18198043, 0.18437475,
        0.18347789, 0.18467549]),
 'std_test_score': array([0.00475011, 0.00848454, 0.00553909, 0.01191212, 0.01094015,
        0.00891737, 0.00832122, 0.01283794, 0.0179765 , 0.0120069 ,
        0.00986128, 0.01003668, 0.01274242, 0.01991718, 0.0216271 ,
        0.01541307, 0.01876674, 0.01482143, 0.02113488, 0.0180466 ,
        0.01618332, 0.01923102, 0.01951627, 0.01366034, 0.01517485,
        0.01320559, 0.01262064]),
 'rank_test_score': array([13,  9,  3,  1,  2,  4,  5,  6,  8,  7, 10, 12, 14, 15, 11, 17, 1
6,
        20, 19, 18, 21, 22, 26, 27, 24, 25, 23])}
```

The best hyperparameters for the decision tree classifier are:

In [ ]:
```
abalone_pca_tree_cv.best_params_
```

Out[ ]:
```
{'max_depth': 4}
```

The score achieved by the best hyperparameters is:

In [ ]:
```
abalone_pca_tree_cv.best_score_
```

Out[ ]:
```
0.2618999668823787
```

Now we will initialize a new decision tree and train it with the best hyperparameters that were discovered by GridSearchCV

In [ ]:
```
# initialize the decision tree classifier with the best hyperparemeter
abalone_pca_tree = DecisionTreeClassifier(random_state=27, max_depth= 4)

# train the classifier using the training dataset
abalone_pca_tree.fit(abalone_PCAminmax_train , abalone_rings_PCAminmax_train)

# view the accuracy of the classifier using the testing dataset
abalone_pca_tree_test_accuracy = abalone_pca_tree.score(abalone_PCAminmax_test , abalone_rin
abalone_pca_tree_test_accuracy
```

Out[ ]:
```
0.26435406698564595
```

Now we will conduct the same procedure on PCA Abalone dataset

In [ ]:

```
# array for the maximum depth of the tree from 1 to 27
decision_tree_depth = list(range(1,28))

# declare the paremeters for the decision tree classifier
parameters = {"max_depth":decision_tree_depth}

# initialize the GridSearchCV
abalone_lda_tree_cv = GridSearchCV(DecisionTreeClassifier(random_state=27), parameters, cv=5

# fit the lda data
abalone_lda_tree_cv.fit(abalone_LDAminmax_train , abalone_rings_LDAminmax_train)

# view the result of the cross validation on the decision tree algorithm
abalone_lda_tree_cv.cv_results_
```

```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\model_select
ion\_split.py:676: UserWarning: The least populated class in y has only 1 members, which is l
ess than n_splits=5.
  warnings.warn(
```

Out[ ]:

```
{'mean_fit_time': array([0.00799961, 0.01079969, 0.01219535, 0.02199841, 0.02039647,
       0.02079668, 0.02219996, 0.02819328, 0.07119799, 0.0531949 ,
       0.07900081, 0.06300015, 0.05239754, 0.05280147, 0.05999942,
       0.04019866, 0.0360002 , 0.03559632, 0.03339763, 0.03079543,
       0.03219767, 0.03879738, 0.03599463, 0.03619819, 0.05499907,
       0.03859901, 0.03339653]),
 'std_fit_time': array([0.00126716, 0.00116562, 0.0023977 , 0.00555076, 0.00241695,
       0.00147181, 0.00204026, 0.00146795, 0.04378197, 0.01776399,
       0.0375975 , 0.01239323, 0.00233211, 0.00874913, 0.03839694,
       0.00906533, 0.00753649, 0.00371958, 0.00436483, 0.00039797,
       0.00299383, 0.00679289, 0.00141563, 0.00271389, 0.02161461,
       0.00500185, 0.00101939]),
 'mean_score_time': array([0.00380211, 0.00340161, 0.0040062 , 0.00460172, 0.00460253,
       0.00700526, 0.00280094, 0.00280738, 0.01360512, 0.00500045,
       0.00600004, 0.00620446, 0.00660086, 0.00600185, 0.00860333,
       0.00400195, 0.00320158, 0.00300221, 0.00260329, 0.00280442,
       0.00280156, 0.00300236, 0.00320263, 0.00420103, 0.00540051,
       0.00340405, 0.00320282]),
 'std_score_time': array([0.00074847, 0.00080153, 0.00167366, 0.00080037, 0.00135518,
       0.00260463, 0.00074694, 0.0004033 , 0.01146362, 0.0020951 ,
       0.00414891, 0.00160226, 0.00162596, 0.00328734, 0.00608931,
       0.00126822, 0.00040002, 0.0006343 , 0.00049018, 0.00040205,
       0.00040063, 0.00062941, 0.00074962, 0.00132658, 0.00205707,
       0.00049554, 0.00098023]),
 'param_max_depth': masked_array(data=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
                    17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False],
       fill_value='?',
            dtype=object),
 'params': [{'max_depth': 1},
  {'max_depth': 2},
  {'max_depth': 3},
  {'max_depth': 4},
  {'max_depth': 5},
  {'max_depth': 6},
  {'max_depth': 7},
  {'max_depth': 8},
  {'max_depth': 9},
  {'max_depth': 10},
  {'max_depth': 11},
  {'max_depth': 12},
  {'max_depth': 13},
  {'max_depth': 14},
  {'max_depth': 15},
  {'max_depth': 16},
  {'max_depth': 17},
  {'max_depth': 18},
  {'max_depth': 19},
  {'max_depth': 20},
  {'max_depth': 21},
  {'max_depth': 22},
  {'max_depth': 23},
  {'max_depth': 24},
  {'max_depth': 25},
  {'max_depth': 26},
  {'max_depth': 27}],
 'split0_test_score': array([0.2122571 , 0.23916293, 0.23916293, 0.25411061, 0.23916293,
       0.23318386, 0.23617339, 0.23617339, 0.23318386, 0.22421525,
       0.2077728 , 0.22122571, 0.20179372, 0.20029895, 0.20478326,
       0.20328849, 0.20478326, 0.20179372, 0.20328849, 0.19282511,
       0.19282511, 0.18684604, 0.19431988, 0.19730942, 0.18535127,
       0.18535127, 0.18535127]),
 'split1_test_score': array([0.21257485, 0.23652695, 0.26796407, 0.25898204, 0.25748503,
       0.23952096, 0.24101796, 0.25449102, 0.24251497, 0.24700599,
       0.21407186, 0.21407186, 0.21107784,
       0.19461078, 0.19610778, 0.20209581, 0.2005988 , 0.20209581,
```

```
        0.20508982, 0.19461078, 0.1991018 , 0.20359281, 0.2005988 ,
        0.20209581, 0.2005988 ]),
 'split2_test_score': array([0.20359281, 0.24700599, 0.25149701, 0.25299401, 0.24850299,
        0.27245509, 0.2754491 , 0.27694611, 0.2754491 , 0.25449102,
        0.24550898, 0.25149701, 0.24401198, 0.22305389, 0.20958084,
        0.21257485, 0.2005988 , 0.20359281, 0.2005988 , 0.1991018 ,
        0.1991018 , 0.19461078, 0.19311377, 0.19610778, 0.19610778,
        0.19610778, 0.19610778]),
 'split3_test_score': array([0.1991018 , 0.24251497, 0.24251497, 0.24401198, 0.24401198,
        0.2245509 , 0.22305389, 0.20958084, 0.21706587, 0.23502994,
        0.22005988, 0.21706587, 0.22005988, 0.20359281, 0.18562874,
        0.18562874, 0.18263473, 0.17664671, 0.17664671, 0.18263473,
        0.18562874, 0.1751497 , 0.18413174, 0.17664671, 0.17664671,
        0.17664671, 0.17664671]),
 'split4_test_score': array([0.20958084, 0.26646707, 0.26197605, 0.26347305, 0.26347305,
        0.24700599, 0.2739521 , 0.25149701, 0.25748503, 0.24550898,
        0.22904192, 0.22305389, 0.2005988 , 0.18413174, 0.1991018 ,
        0.19161677, 0.18113772, 0.18562874, 0.17365269, 0.18113772,
        0.17814371, 0.18263473, 0.19011976, 0.18562874, 0.18562874,
        0.18562874, 0.18562874]),
 'mean_test_score': array([0.20742148, 0.24633558, 0.25262301, 0.25471434, 0.2505272 ,
        0.24334336, 0.24992929, 0.24573767, 0.24513977, 0.24125023,
        0.22299168, 0.22658047, 0.21610725, 0.20502985, 0.2020345 ,
        0.19754393, 0.19305246, 0.19395156, 0.1909571 , 0.19155903,
        0.19215784, 0.18677041, 0.19215739, 0.19185709, 0.18886666,
        0.18916606, 0.18886666]),
 'std_test_score': array([0.00526268, 0.01065814, 0.01102217, 0.00652369, 0.00885097,
        0.01632109, 0.02106789, 0.0222825 , 0.02004514, 0.01054008,
        0.0133593 , 0.01260944, 0.01577231, 0.01317423, 0.00920907,
        0.00943011, 0.00953293, 0.01085826, 0.01297858, 0.00845892,
        0.0095386 , 0.00741746, 0.0049473 , 0.00954643, 0.00851171,
        0.00893494, 0.00851171]),
 'rank_test_score': array([13,  5,  2,  1,  3,  8,  4,  6,  7,  9, 11, 10, 12, 14, 15, 16, 1
8,
        17, 23, 22, 19, 27, 20, 21, 25, 24, 25])}
```

The best hyperparameters for the decision tree classifier are:

In [ ]:
```
abalone_lda_tree_cv.best_params_
```

Out[ ]:
```
{'max_depth': 4}
```

The score achieved by the best hyperparameters is:

In [ ]:
```
abalone_lda_tree_cv.best_score_
```

Out[ ]:
```
0.2547143381398638
```

Now we will initialize a new decision tree and train it with the best hyperparameters that were discovered by GridSearchCV

In [ ]:
```
# initialize the decision tree classifier with the best hyperparemeter
abalone_lda_tree = DecisionTreeClassifier(random_state=27, max_depth= 4)

# train the classifier using the training dataset
abalone_lda_tree.fit(abalone_LDAminmax_train , abalone_rings_LDAminmax_train)

# view the accuracy of the classifier using the testing dataset
abalone_lda_tree_test_accuracy = abalone_lda_tree.score(abalone_LDAminmax_test , abalone_ring
abalone_lda_tree_test_accuracy
```

Out[ ]:
```
0.02033492822966507
```

First we will determine the maximum depth that is possible for the wine dataset using the decision tree classifier which is shown below:

In [ ]:
```
wine_dt_depth_check = DecisionTreeClassifier(random_state=27)
wine_dt_depth_check.fit(wine_raw_train, wine_quality_raw_train)
wine_dt_depth_check.tree_.max_depth
```

Out[ ]:
```
28
```

The limit of the maximum depth for the decision tree will be **28**

3.1.2.1 Raw Data

First we will apply the normalized dataset as shown below:

In [ ]:
```
# array for the maximum depth of the tree from 1 to 28
decision_tree_depth = list(range(1,29))

# declare the paremeters for the decision tree classifier
parameters = {"max_depth":decision_tree_depth}

# initialize the GridSearchCV
wine_raw_tree_cv = GridSearchCV(DecisionTreeClassifier(random_state=27), parameters, cv=5)

# fit the normalized data
wine_raw_tree_cv.fit(wine_raw_train, wine_quality_raw_train)

# view the result of the cross validation on the decision tree algorithm
wine_raw_tree_cv.cv_results_
```

```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\model_selection\_split.py:676: UserWarning: The least populated class in y has only 4 members, which is less than n_splits=5.
  warnings.warn(
```

Out[ ]:

```
{'mean_fit_time': array([0.02879782, 0.0173914 , 0.02179585, 0.02479758, 0.0283958 ,
        0.02699714, 0.02839417, 0.03579893, 0.04859819, 0.09313154,
        0.05679884, 0.05520029, 0.05950041, 0.05220881, 0.05420308,
        0.06760244, 0.06899743, 0.06780348, 0.10839653, 0.06779547,
        0.0710011 , 0.0935987 , 0.07039728, 0.12120056, 0.07900133,
        0.07319722, 0.07519684, 0.05979772]),
 'std_fit_time': array([0.03089039, 0.00483786, 0.00915278, 0.00231316, 0.01052053,
        0.00260731, 0.00101962, 0.0042622 , 0.00920144, 0.04117031,
        0.00425811, 0.00563779, 0.0092735 , 0.00313022, 0.00371584,
        0.00990382, 0.01554594, 0.01530183, 0.01680449, 0.0157238 ,
        0.01426431, 0.02797247, 0.01009212, 0.04155767, 0.02413349,
        0.01685754, 0.02522863, 0.00820527]),
 'mean_score_time': array([0.00980444, 0.00420551, 0.00500154, 0.00380197, 0.00380321,
        0.00340114, 0.00340719, 0.0036016 , 0.00380435, 0.00544381,
        0.00420022, 0.0064023 , 0.00380449, 0.00460277, 0.00300355,
        0.00340176, 0.0030045 , 0.00420122, 0.00880723, 0.00480237,
        0.00593729, 0.00640059, 0.00440202, 0.01480126, 0.00359993,
        0.00460544, 0.00540485, 0.00340123]),
 'std_score_time': array([6.34046989e-03, 7.44178740e-04, 1.67271562e-03, 1.72152095e-03,
        1.16686696e-03, 4.90780568e-04, 4.89931034e-04, 1.01840242e-03,
        1.16784362e-03, 1.22220882e-03, 1.93755240e-03, 4.84048654e-03,
        3.99983372e-04, 2.33298876e-03, 2.70329242e-06, 8.02881027e-04,
        3.20369282e-06, 1.16699129e-03, 3.97472440e-03, 1.32646197e-03,
        1.77141128e-03, 3.13785654e-03, 1.85498591e-03, 2.26048339e-02,
        1.19898484e-03, 1.35764981e-03, 4.36467666e-03, 4.93139894e-04]),
 'param_max_depth': masked_array(data=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
                    17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False],
        fill_value='?',
             dtype=object),
 'params': [{'max_depth': 1},
  {'max_depth': 2},
  {'max_depth': 3},
  {'max_depth': 4},
  {'max_depth': 5},
  {'max_depth': 6},
  {'max_depth': 7},
  {'max_depth': 8},
  {'max_depth': 9},
  {'max_depth': 10},
  {'max_depth': 11},
  {'max_depth': 12},
  {'max_depth': 13},
  {'max_depth': 14},
  {'max_depth': 15},
  {'max_depth': 16},
  {'max_depth': 17},
  {'max_depth': 18},
  {'max_depth': 19},
  {'max_depth': 20},
  {'max_depth': 21},
  {'max_depth': 22},
  {'max_depth': 23},
  {'max_depth': 24},
  {'max_depth': 25},
  {'max_depth': 26},
  {'max_depth': 27},
  {'max_depth': 28}],
 'split0_test_score': array([0.48942308, 0.52884615, 0.53557692, 0.54038462, 0.5375    ,
        0.55384615, 0.54230769, 0.55      , 0.54903846, 0.54326923,
        0.56634615, 0.57596154, 0.55480769, 0.56730769, 0.55961538,
        0.56538462, 0.56538462, 0.57115385, 0.57307692, 0.57403846,
        0.56730769, 0.58557692, 0.57884615, 0.57788462, 0.57788462,
        0.57788462, 0.57788462, 0.57788462]),
Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js 538, 0.53269231, 0.55      , 0.54134615, 0.55480769,
        0.56442308, 0.54519231, 0.56057692, 0.55961538, 0.56346154,
```

```
       0.58173077, 0.57307692, 0.56730769, 0.56538462, 0.58557692,
       0.575     , 0.58461538, 0.57019231, 0.58461538, 0.59134615,
       0.5875    , 0.59038462, 0.58846154, 0.59615385, 0.59903846,
       0.59807692, 0.59807692, 0.59807692]),
 'split2_test_score': array([0.4985563 , 0.51010587, 0.51780558, 0.51876805, 0.53801732,
       0.53416747, 0.53512993, 0.5360924 , 0.54379211, 0.55822907,
       0.5707411 , 0.55052936, 0.56592878, 0.56977863, 0.57651588,
       0.5707411 , 0.56304139, 0.56207892, 0.56207892, 0.56400385,
       0.56111646, 0.57170356, 0.56785371, 0.56785371, 0.56592878,
       0.5707411 , 0.5707411 , 0.5707411 ]),
 'split3_test_score': array([0.48604427, 0.51588065, 0.49951877, 0.51299326, 0.52743022,
       0.53224254, 0.52550529, 0.533205  , 0.5466795 , 0.56207892,
       0.56689124, 0.57651588, 0.57940327, 0.58806545, 0.58036574,
       0.58902791, 0.57266603, 0.58614052, 0.57170356, 0.57844081,
       0.56689124, 0.57459095, 0.57940327, 0.58325313, 0.58325313,
       0.58325313, 0.58325313, 0.58325313]),
 'split4_test_score': array([0.50818094, 0.53031761, 0.53128008, 0.52358037, 0.51876805,
       0.52743022, 0.5360924 , 0.53994225, 0.52358037, 0.52165544,
       0.53897979, 0.52935515, 0.54571704, 0.5466795 , 0.54475457,
       0.54860443, 0.54956689, 0.55245428, 0.54764196, 0.54571704,
       0.55052936, 0.53512993, 0.53512993, 0.53512993, 0.53512993,
       0.53512993, 0.53512993, 0.53512993]),
 'mean_test_score': array([0.498364  , 0.52356852, 0.52683627, 0.52741449, 0.53530466,
       0.54242189, 0.53684552, 0.54396332, 0.54454116, 0.54973884,
       0.56493781, 0.56108777, 0.56263289, 0.56744318, 0.5693657 ,
       0.56975161, 0.56705486, 0.56840398, 0.56782335, 0.57070926,
       0.56666895, 0.5714772 , 0.56993892, 0.57205505, 0.57224698,
       0.57301714, 0.57301714, 0.57301714]),
 'std_test_score': array([0.00953644, 0.00891059, 0.01709096, 0.01148694, 0.01206659,
       0.0142202 , 0.00680611, 0.01006295, 0.0117639 , 0.01576837,
       0.01410886, 0.01855497, 0.0115008 , 0.01316145, 0.0150712 ,
       0.01316339, 0.01152895, 0.01113323, 0.01236863, 0.01527295,
       0.01204746, 0.01942763, 0.01859039, 0.02059732, 0.02139297,
       0.02096226, 0.02096226, 0.02096226]),
 'rank_test_score': array([28, 27, 26, 25, 24, 22, 23, 21, 20, 19, 16, 18, 17, 13, 10,  9, 1
4,
       11, 12,  7, 15,  6,  8,  5,  4,  1,  1,  1])}
```

The best hyperparameters for the decision tree classifier are:

In [ ]:
```
wine_raw_tree_cv.best_params_
```

Out[ ]:
```
{'max_depth': 26}
```

The score achieved by the best hyperparameters is:

In [ ]:
```
wine_raw_tree_cv.best_score_
```

Out[ ]:
```
0.5730171392611239
```

Now we will initialize a new decision tree and train it with the best hyperparameters that were discovered by GridSearchCV

In [ ]:
```
# initialize the decision tree classifier with the best hyperparemeter
wine_raw_tree = DecisionTreeClassifier(random_state=27, max_depth= 26)

# train the classifier using the training dataset
wine_raw_tree.fit(wine_raw_train, wine_quality_raw_train)

# view the accuracy of the classifier using the testing dataset
wine_raw_tree_test_accuracy = wine_raw_tree.score(wine_raw_test, wine_quality_raw_test)
wine_raw_tree_test_accuracy
```

```
0.61
```

3.1.2.2 PCA Dataset

Now we will conduct the same procedure on PCA Wine dataset

In [ ]:

```
# array for the maximum depth of the tree from 1 to 28
decision_tree_depth = list(range(1,29))

# declare the paremeters for the decision tree classifier
parameters = {"max_depth":decision_tree_depth}

# initialize the GridSearchCV
wine_pca_tree_cv = GridSearchCV(DecisionTreeClassifier(random_state=27), parameters, cv=5)

# fit the pca data
wine_pca_tree_cv.fit(wine_PCAminmax_train , wine_quality_PCAminmax_train)

# view the result of the cross validation on the decision tree algorithm
wine_pca_tree_cv.cv_results_
```

```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\model_selection\_split.py:676: UserWarning: The least populated class in y has only 4 members, which is less than n_splits=5.
  warnings.warn(
```

Out[ ]:
```
{'mean_fit_time': array([0.0191999 , 0.02239881, 0.03179903, 0.02759871, 0.03019738,
       0.15259652, 0.07499499, 0.08940215, 0.07119312, 0.06799517,
       0.06619682, 0.06839395, 0.07639642, 0.07339063, 0.08419313,
       0.07779694, 0.09600091, 0.08280044, 0.08219795, 0.07939668,
       0.0826025 , 0.10419989, 0.08320112, 0.08139205, 0.083394  ,
       0.08839631, 0.08039904, 0.08019261]),
 'std_fit_time': array([0.00312913, 0.00300041, 0.00810525, 0.00338243, 0.00172084,
       0.06027365, 0.02267106, 0.01163952, 0.01156608, 0.00575834,
       0.00171999, 0.00215582, 0.0043686 , 0.00279985, 0.00842689,
       0.00343168, 0.01800986, 0.00655533, 0.00519246, 0.00646444,
       0.0081179 , 0.02955135, 0.01014411, 0.00935204, 0.00882899,
       0.00913701, 0.00527777, 0.00643246]),
 'mean_score_time': array([0.00879989, 0.006005  , 0.00459962, 0.00420175, 0.00320382,
       0.01760468, 0.01020317, 0.00659814, 0.00580688, 0.0058012 ,
       0.00380383, 0.00520167, 0.00560579, 0.00520344, 0.00580444,
       0.00480719, 0.00540085, 0.00420094, 0.00560379, 0.00420117,
       0.00760021, 0.00600224, 0.00560284, 0.00680995, 0.00480328,
       0.00520191, 0.00480108, 0.00600867]),
 'std_score_time': array([0.00376157, 0.00200663, 0.00101897, 0.00159841, 0.00116636,
       0.01333471, 0.00714175, 0.00101797, 0.00074769, 0.00240751,
       0.00098134, 0.00147492, 0.00216015, 0.00194483, 0.0024835 ,
       0.00147539, 0.0022498 , 0.00116864, 0.00162878, 0.00160539,
       0.00621848, 0.00179299, 0.00102157, 0.00183211, 0.00194366,
       0.0014685 , 0.00147047, 0.00253056]),
 'param_max_depth': masked_array(data=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
                    17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False],
        fill_value='?',
             dtype=object),
 'params': [{'max_depth': 1},
  {'max_depth': 2},
  {'max_depth': 3},
  {'max_depth': 4},
  {'max_depth': 5},
  {'max_depth': 6},
  {'max_depth': 7},
  {'max_depth': 8},
  {'max_depth': 9},
  {'max_depth': 10},
  {'max_depth': 11},
  {'max_depth': 12},
  {'max_depth': 13},
  {'max_depth': 14},
  {'max_depth': 15},
  {'max_depth': 16},
  {'max_depth': 17},
  {'max_depth': 18},
  {'max_depth': 19},
  {'max_depth': 20},
  {'max_depth': 21},
  {'max_depth': 22},
  {'max_depth': 23},
  {'max_depth': 24},
  {'max_depth': 25},
  {'max_depth': 26},
  {'max_depth': 27},
  {'max_depth': 28}],
 'split0_test_score': array([0.42692308, 0.49326923, 0.49326923, 0.51538462, 0.525     ,
       0.52980769, 0.53173077, 0.55192308, 0.54519231, 0.54230769,
       0.55288462, 0.56057692, 0.56153846, 0.55384615, 0.56826923,
       0.57596154, 0.58173077, 0.57115385, 0.57884615, 0.59038462,
       0.57788462, 0.58557692, 0.57980769, 0.58942308, 0.58942308,
       0.58942308, 0.58942308, 0.58942308]),
 'split1_test_score': array([0.42980769, 0.48942308, 0.49807692, 0.50384615, 0.51730769,
       0.52788075    , 0.55288462, 0.55      ,
       0.55865385, 0.56057692, 0.58653846, 0.59326923, 0.57596154,
```

```
       0.58653846, 0.59423077, 0.58942308, 0.59423077, 0.58942308,
       0.5875    , 0.58846154, 0.58846154, 0.58557692, 0.58653846,
       0.58653846, 0.58653846, 0.58653846]),
 'split2_test_score': array([0.44465833, 0.49278152, 0.49566891, 0.50433109, 0.52646776,
       0.54090472, 0.53994225, 0.5572666 , 0.57266603, 0.54186718,
       0.56400385, 0.56977863, 0.56881617, 0.57747834, 0.56785371,
       0.56304139, 0.56400385, 0.56689124, 0.57362849, 0.57459095,
       0.57362849, 0.57651588, 0.58325313, 0.57844081, 0.57844081,
       0.57651588, 0.57266603, 0.57747834]),
 'split3_test_score': array([0.44562079, 0.49759384, 0.50336862, 0.51106833, 0.51299326,
       0.51876805, 0.52069297, 0.52550529, 0.52358037, 0.55149182,
       0.55534167, 0.5466795 , 0.56111646, 0.56400385, 0.57266603,
       0.56881617, 0.57266603, 0.57651588, 0.56977863, 0.57459095,
       0.56592878, 0.5919153 , 0.5919153 , 0.58999038, 0.5919153 ,
       0.5919153 , 0.5919153 , 0.5919153 ]),
 'split4_test_score': array([0.43984601, 0.48219442, 0.49278152, 0.51780558, 0.52646776,
       0.53128008, 0.55052936, 0.54764196, 0.54282964, 0.54090472,
       0.54956689, 0.54282964, 0.54475457, 0.54186718, 0.53897979,
       0.53512993, 0.55149182, 0.5466795 , 0.55149182, 0.55919153,
       0.55630414, 0.55822907, 0.54956689, 0.54956689, 0.55822907,
       0.55822907, 0.55822907, 0.55822907]),
 'mean_test_score': array([0.43737118, 0.49105242, 0.49663304, 0.51048715, 0.52164729,
       0.53069057, 0.53550215, 0.54396739, 0.54743059, 0.54531428,
       0.55609018, 0.55608832, 0.56455282, 0.56609295, 0.56474606,
       0.5658975 , 0.57282465, 0.57013271, 0.57359517, 0.57763623,
       0.5722492 , 0.58013974, 0.57860091, 0.57859962, 0.58090934,
       0.58052436, 0.57975439, 0.58071685]),
 'std_test_score': array([0.00766358, 0.00513446, 0.00386088, 0.00565477, 0.00550343,
       0.00709379, 0.00979646, 0.01128132, 0.01588449, 0.00448295,
       0.00495152, 0.00992006, 0.01352339, 0.01794109, 0.01322485,
       0.01726646, 0.0146319 , 0.01396042, 0.01383131, 0.01149091,
       0.01058716, 0.01208875, 0.0151035 , 0.01508976, 0.01221303,
       0.01231469, 0.01265685, 0.01225792]),
 'rank_test_score': array([28, 27, 26, 25, 24, 23, 22, 21, 19, 20, 17, 18, 16, 13, 15, 14, 1
0,
       12,  9,  8, 11,  4,  6,  7,  1,  3,  5,  2])}
```

The best hyperparameters for the decision tree classifier are:

In [ ]:
```
wine_pca_tree_cv.best_params_
```

Out[ ]:
```
{'max_depth': 25}
```

The score achieved by the best hyperparameters is:

In [ ]:
```
wine_pca_tree_cv.best_score_
```

Out[ ]:
```
0.5809093433034723
```

Now we will initialize a new decision tree and train it with the best hyperparameters that were discovered by GridSearchCV

In [ ]:
```
# initialize the decision tree classifier with the best hyperparemeter
wine_pca_tree = DecisionTreeClassifier(random_state=27, max_depth= 25)

# train the classifier using the training dataset
wine_pca_tree.fit(wine_PCAminmax_train , wine_quality_PCAminmax_train)

# view the accuracy of the classifier using the testing dataset
wine_pca_tree_test_accuracy = wine_pca_tree.score(wine_PCAminmax_test , wine_quality_PCAminma
wine_pca_tree_test_accuracy
```

Out[ ]:
```
0.39153846153846156
```


3.1.2.3 LDA Dataset

Now we will conduct the same procedure on PCA Abalone dataset

In [ ]:

```
# array for the maximum depth of the tree from 1 to 28
decision_tree_depth = list(range(1,29))

# declare the paremeters for the decision tree classifier
parameters = {"max_depth":decision_tree_depth}

# initialize the GridSearchCV
wine_lda_tree_cv = GridSearchCV(DecisionTreeClassifier(random_state=27), parameters, cv=5)

# fit the lda data
wine_lda_tree_cv.fit(wine_LDAminmax_train , wine_quality_LDAminmax_train)

# view the result of the cross validation on the decision tree algorithm
wine_lda_tree_cv.cv_results_
```

```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\model_selection\_split.py:676: UserWarning: The least populated class in y has only 4 members, which is less than n_splits=5.
  warnings.warn(
```

Out[ ]:
```
{'mean_fit_time': array([0.01380196, 0.01459594, 0.01920028, 0.01659899, 0.01939726,
       0.01759748, 0.01919785, 0.02539892, 0.02279921, 0.02419429,
       0.02599583, 0.02800221, 0.02779322, 0.02980356, 0.03079782,
       0.03360243, 0.03159671, 0.03239856, 0.03279662, 0.03379483,
       0.03379722, 0.03399534, 0.0341979 , 0.03419623, 0.03719726,
       0.04079542, 0.03479748, 0.03419456]),
 'std_fit_time': array([0.00204328, 0.00195867, 0.00193859, 0.0024175 , 0.00241524,
       0.00080433, 0.00040241, 0.00241481, 0.00075247, 0.00040377,
       0.00063159, 0.0008901 , 0.00116783, 0.00159799, 0.0007453 ,
       0.00232893, 0.00080081, 0.00049   , 0.00074948, 0.00074666,
       0.00074533, 0.0006307 , 0.00146918, 0.00171645, 0.0034863 ,
       0.0064338 , 0.00172295, 0.00116928]),
 'mean_score_time': array([0.00419898, 0.00519948, 0.00400019, 0.004004  , 0.00340219,
       0.00320239, 0.00280237, 0.00420103, 0.00320215, 0.00260305,
       0.00320568, 0.00319948, 0.00440807, 0.00340371, 0.00280075,
       0.00399952, 0.00320363, 0.00300293, 0.00380287, 0.00340581,
       0.00280528, 0.00340695, 0.00300264, 0.00360746, 0.00300298,
       0.00380301, 0.00320487, 0.00380344]),
 'std_score_time': array([7.48517085e-04, 2.04365936e-03, 8.95379925e-04, 8.97200494e-04,
       1.01901313e-03, 4.01173487e-04, 1.16540316e-03, 1.71891154e-03,
       4.02001225e-04, 4.88819700e-04, 4.03459260e-04, 7.48804805e-04,
       4.94520697e-04, 4.93918471e-04, 4.04147029e-04, 8.89831357e-04,
       3.99359361e-04, 2.77386466e-06, 7.48682045e-04, 8.05023021e-04,
       7.52424640e-04, 4.94013224e-04, 6.34379037e-04, 8.04541542e-04,
       1.72322378e-06, 1.16511747e-03, 3.99117954e-04, 9.84481967e-04]),
 'param_max_depth': masked_array(data=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
                    17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False],
       fill_value='?',
            dtype=object),
 'params': [{'max_depth': 1},
  {'max_depth': 2},
  {'max_depth': 3},
  {'max_depth': 4},
  {'max_depth': 5},
  {'max_depth': 6},
  {'max_depth': 7},
  {'max_depth': 8},
  {'max_depth': 9},
  {'max_depth': 10},
  {'max_depth': 11},
  {'max_depth': 12},
  {'max_depth': 13},
  {'max_depth': 14},
  {'max_depth': 15},
  {'max_depth': 16},
  {'max_depth': 17},
  {'max_depth': 18},
  {'max_depth': 19},
  {'max_depth': 20},
  {'max_depth': 21},
  {'max_depth': 22},
  {'max_depth': 23},
  {'max_depth': 24},
  {'max_depth': 25},
  {'max_depth': 26},
  {'max_depth': 27},
  {'max_depth': 28}],
 'split0_test_score': array([0.54134615, 0.54134615, 0.55480769, 0.55480769, 0.55      ,
       0.53557692, 0.54326923, 0.5375    , 0.54326923, 0.53653846,
       0.53942308, 0.52115385, 0.54615385, 0.54326923, 0.53942308,
       0.55096154, 0.55673077, 0.55384615, 0.55096154, 0.55673077,
       0.56153846, 0.56442308, 0.56057692, 0.56057692, 0.55961538,
       0.55961538, 0.55961538, 0.55961538]),
 'split1_test_score': array([0.52884615, 0.52884615, 0.55      , 0.55673077, 0.56346154,
       0.5625    , 0.56442308, 0.54615385, 0.55480769, 0.56634615,
```

```
       0.55288462, 0.56730769, 0.58653846, 0.5625    , 0.575     ,
       0.57884615, 0.57596154, 0.56730769, 0.56826923, 0.57019231,
       0.55673077, 0.57115385, 0.55      , 0.55576923, 0.55288462,
       0.55961538, 0.56442308, 0.55192308]),
 'split2_test_score': array([0.53897979, 0.53897979, 0.55630414, 0.55822907, 0.5572666 ,
       0.56304139, 0.56400385, 0.56015399, 0.55341675, 0.5572666 ,
       0.54764196, 0.54475457, 0.54764196, 0.55630414, 0.56977863,
       0.56304139, 0.5707411 , 0.56496631, 0.55437921, 0.56496631,
       0.57170356, 0.56400385, 0.56785371, 0.57170356, 0.57170356,
       0.57555342, 0.57459095, 0.5813282 ]),
 'split3_test_score': array([0.52839269, 0.52839269, 0.52935515, 0.53897979, 0.533205  ,
       0.53705486, 0.5360924 , 0.53031761, 0.53128008, 0.54571704,
       0.53801732, 0.54956689, 0.56111646, 0.56207892, 0.55437921,
       0.56304139, 0.57170356, 0.56881617, 0.56400385, 0.5707411 ,
       0.57362849, 0.57844081, 0.5707411 , 0.58229066, 0.5919153 ,
       0.57459095, 0.57844081, 0.57459095]),
 'split4_test_score': array([0.53512993, 0.53512993, 0.5466795 , 0.54379211, 0.55052936,
       0.55630414, 0.55822907, 0.54860443, 0.54186718, 0.5466795 ,
       0.53994225, 0.54282964, 0.52935515, 0.53031761, 0.53128008,
       0.5360924 , 0.5226179 , 0.533205  , 0.54379211, 0.54282964,
       0.54186718, 0.53128008, 0.53994225, 0.55149182, 0.54860443,
       0.55052936, 0.54764196, 0.54956689]),
 'mean_test_score': array([0.53453894, 0.53453894, 0.5474293 , 0.55050788, 0.5508925 ,
       0.55089546, 0.55320352, 0.54454598, 0.54492819, 0.55050955,
       0.54358185, 0.54512253, 0.55416118, 0.55089398, 0.5539722 ,
       0.55839657, 0.55955097, 0.55762827, 0.55628119, 0.56109203,
       0.56109369, 0.56186033, 0.5578228 , 0.56436644, 0.56494466,
       0.5639809 , 0.56494244, 0.5634049 ]),
 'std_test_score': array([0.00522672, 0.00522672, 0.00966488, 0.00767891, 0.0101231 ,
       0.0121463 , 0.0114827 , 0.01014651, 0.00857966, 0.01028868,
       0.00573641, 0.01477717, 0.01907155, 0.01241448, 0.0168451 ,
       0.01424142, 0.01956638, 0.01329291, 0.00884586, 0.01042559,
       0.01147678, 0.01617155, 0.01145423, 0.0112173 , 0.0155817 ,
       0.00964941, 0.01098411, 0.01252243]),
 'rank_test_score': array([27, 27, 22, 21, 19, 17, 16, 25, 24, 20, 26, 23, 14, 18, 15, 10,
9,
       12, 13,  8,  7,  6, 11,  3,  1,  4,  2,  5])}
```

The best hyperparameters for the decision tree classifier are:

In [ ]:
```
wine_lda_tree_cv.best_params_
```

Out[ ]:
```
{'max_depth': 25}
```

The score achieved by the best hyperparameters is:

In [ ]:
```
wine_lda_tree_cv.best_score_
```

Out[ ]:
```
0.5649446583253128
```

Now we will initialize a new decision tree and train it with the best hyperparameters that were discovered by GridSearchCV

In [ ]:
```
# initialize the decision tree classifier with the best hyperparemeter
abalone_lda_tree = DecisionTreeClassifier(random_state=27, max_depth= 5)

# train the classifier using the training dataset
abalone_lda_tree.fit(wine_LDAminmax_train , wine_quality_LDAminmax_train)

# view the accuracy of the classifier using the testing dataset
wine_lda_tree_test_accuracy = abalone_lda_tree.score(wine_LDAminmax_test , wine_quality_LDAm
wine_lda_tree_test_accuracy
```

```
0.4469230769230769
```

3.2 Plot of mean accuracy

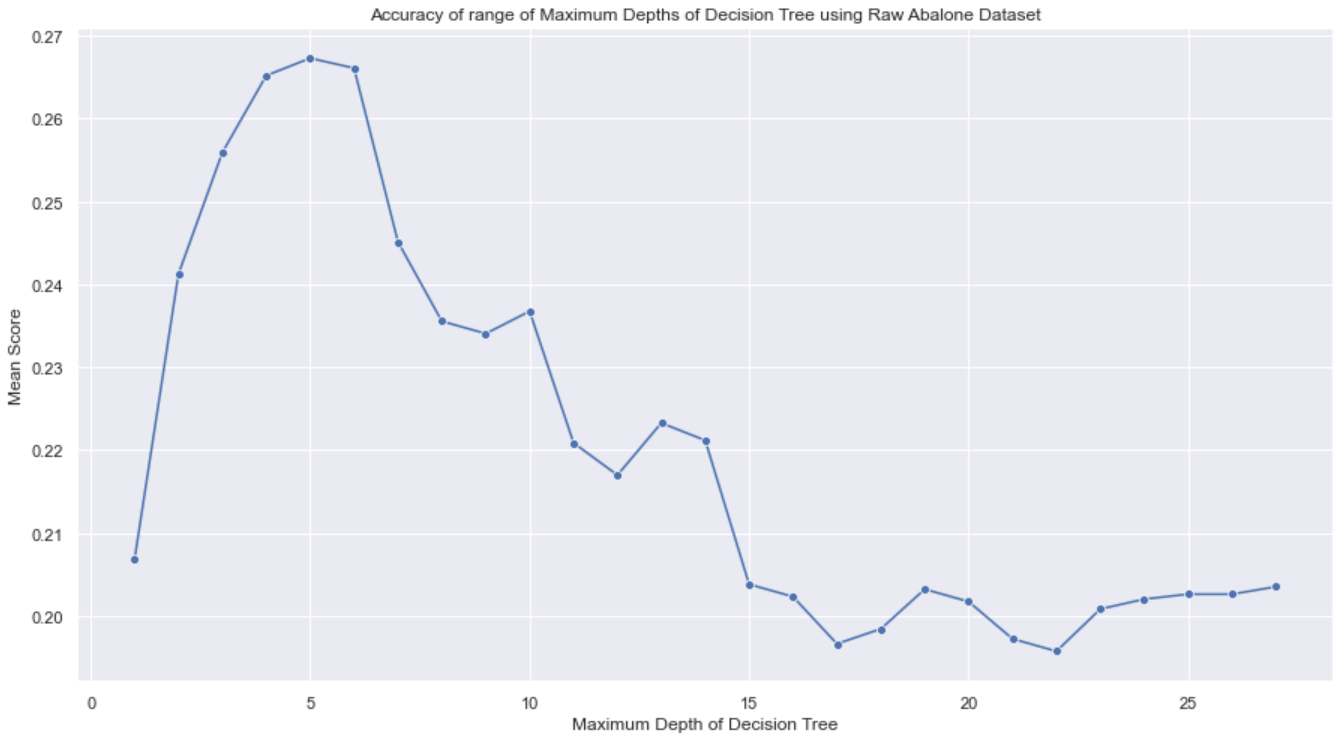
3.2.1 Abalone Dataset

Raw Dataset

In [ ]:
```
# generate a dataframe of the maximum depth and the respective score
abalone_raw_tree_cv_results = pd.DataFrame([abalone_raw_tree_cv.cv_results_['param_max_depth

# transpose the dataframe
abalone_raw_tree_cv_results = abalone_raw_tree_cv_results.T

# name the columns of the dataframe
abalone_raw_tree_cv_results.columns = ["Maximum Depth of Decision Tree","Mean Score"]
```

In [ ]:
```
# initialize the plot
sns.set(rc = {'figure.figsize':(15,8)})

# assign the datapoints for the axes
sns.lineplot(data= abalone_raw_tree_cv_results, x="Maximum Depth of Decision Tree", y="Mean S

# assign the graph title
plt.title("Accuracy of range of Maximum Depths of Decision Tree using Raw Abalone Dataset")
```

Out[ ]:
```
Text(0.5, 1.0, 'Accuracy of range of Maximum Depths of Decision Tree using Raw Abalone Datase
t')
```



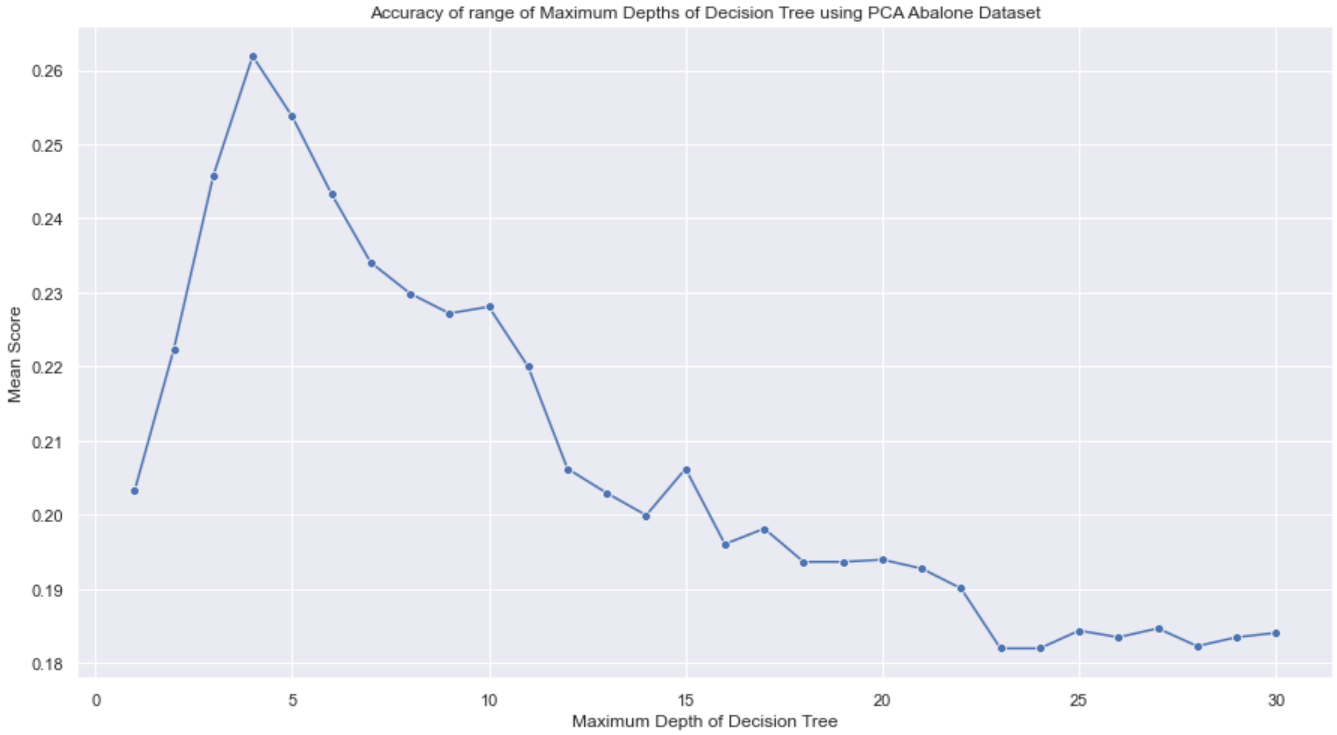
PCA Dataset

In [ ]:
```
# generate a dataframe of the maximum depth and the respective score
abalone_pca_tree_cv_results = pd.DataFrame([abalone_pca_tree_cv.cv_results_['param_max_depth

# transpose the dataframe
abalone_pca_tree_cv_results = abalone_pca_tree_cv_results.T

# name the columns of the dataframe
abalone_pca_tree_cv_results.columns = ["Maximum Depth of Decision Tree","Mean Score"]
```

In [ ]:
```
# initialize the t-SNE plot
sns.set(rc = {'figure.figsize':(15,8)})
```

```
sns.lineplot(data= abalone_pca_tree_cv_results, x="Maximum Depth of Decision Tree", y="Mean S

# assign the graph title
plt.title("Accuracy of range of Maximum Depths of Decision Tree using PCA Abalone Dataset")
```

Out[ ]:
```
Text(0.5, 1.0, 'Accuracy of range of Maximum Depths of Decision Tree using PCA Abalone Datase
t')
```

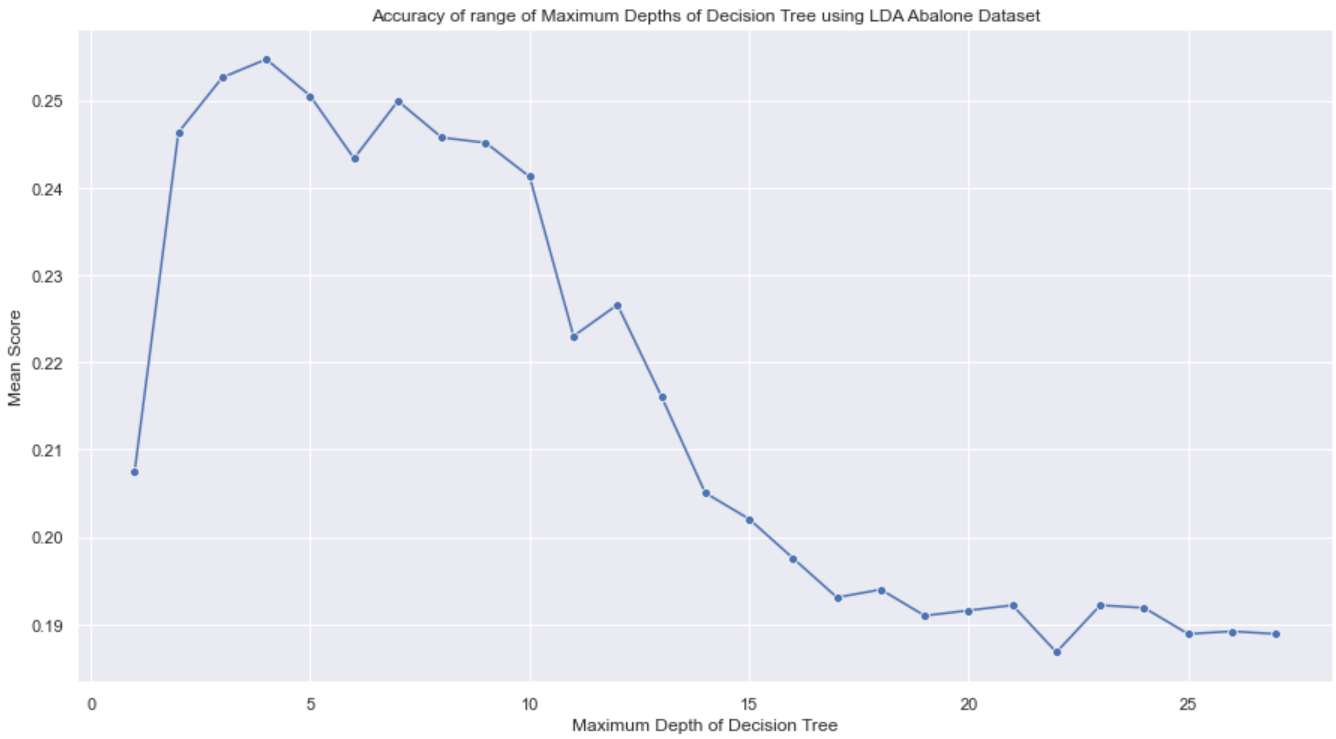


LDA Dataset

In [ ]:
```
# generate a dataframe of the maximum depth and the respective score
abalone_lda_tree_cv_results = pd.DataFrame([abalone_lda_tree_cv.cv_results_['param_max_depth

# transpose the dataframe
abalone_lda_tree_cv_results = abalone_lda_tree_cv_results.T

# name the columns of the dataframe
abalone_lda_tree_cv_results.columns = ["Maximum Depth of Decision Tree","Mean Score"]
```

In [ ]:
```
# initialize the t-SNE plot
sns.set(rc = {'figure.figsize':(15,8)})

# assign the datapoints for the axes
sns.lineplot(data= abalone_lda_tree_cv_results, x="Maximum Depth of Decision Tree", y="Mean S

# assign the graph title
plt.title("Accuracy of range of Maximum Depths of Decision Tree using LDA Abalone Dataset")
```

Out[ ]:
```
Text(0.5, 1.0, 'Accuracy of range of Maximum Depths of Decision Tree using LDA Abalone Datase
t')
```



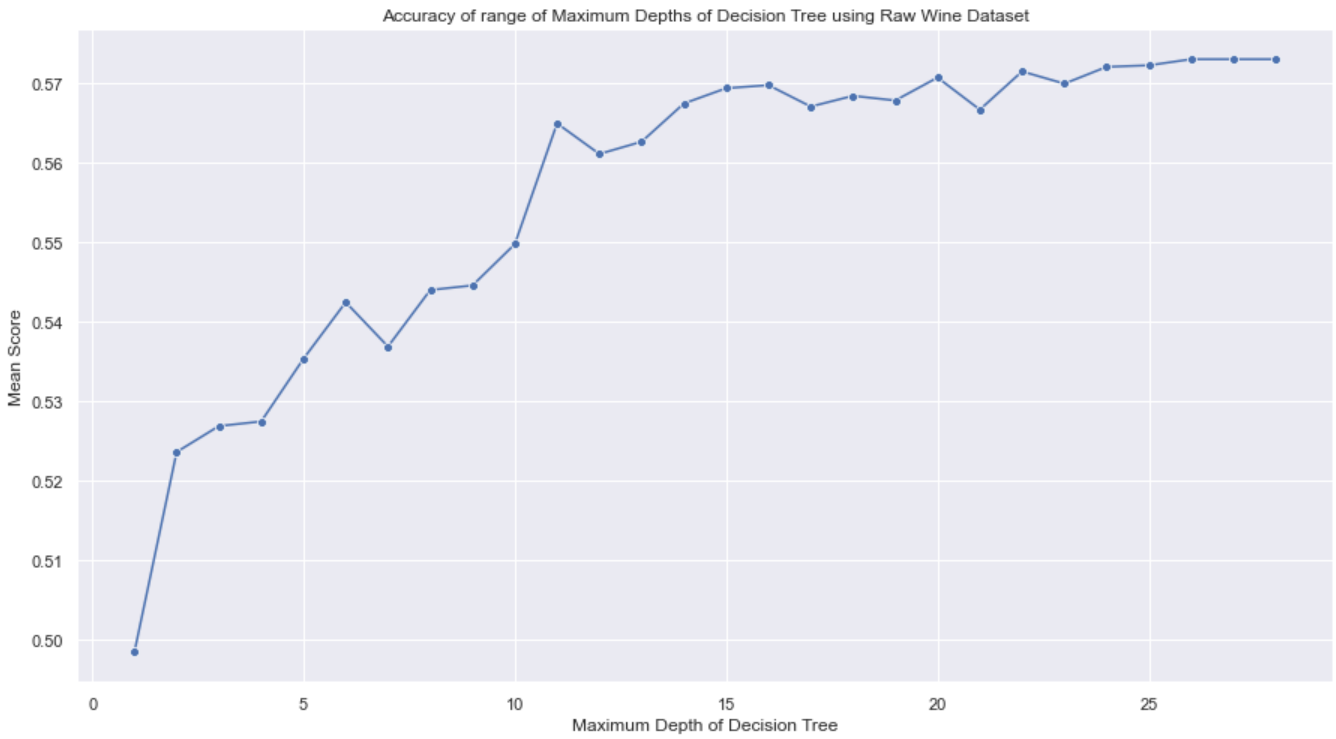
3.2.2 Wine Dataset

Raw Dataset

In [ ]:
```
# generate a dataframe of the maximum depth and the respective score
wine_raw_tree_cv_results = pd.DataFrame([wine_raw_tree_cv.cv_results_['param_max_depth'],win

# transpose the dataframe
wine_raw_tree_cv_results = wine_raw_tree_cv_results.T

# name the columns of the dataframe
wine_raw_tree_cv_results.columns = ["Maximum Depth of Decision Tree","Mean Score"]
```

In [ ]:
```
# initialize the plot
sns.set(rc = {'figure.figsize':(15,8)})

# assign the datapoints for the axes
sns.lineplot(data= wine_raw_tree_cv_results, x="Maximum Depth of Decision Tree", y="Mean Sco

# assign the graph title
plt.title("Accuracy of range of Maximum Depths of Decision Tree using Raw Wine Dataset")
```

Out[ ]:
```
Text(0.5, 1.0, 'Accuracy of range of Maximum Depths of Decision Tree using Raw Wine Dataset')
```

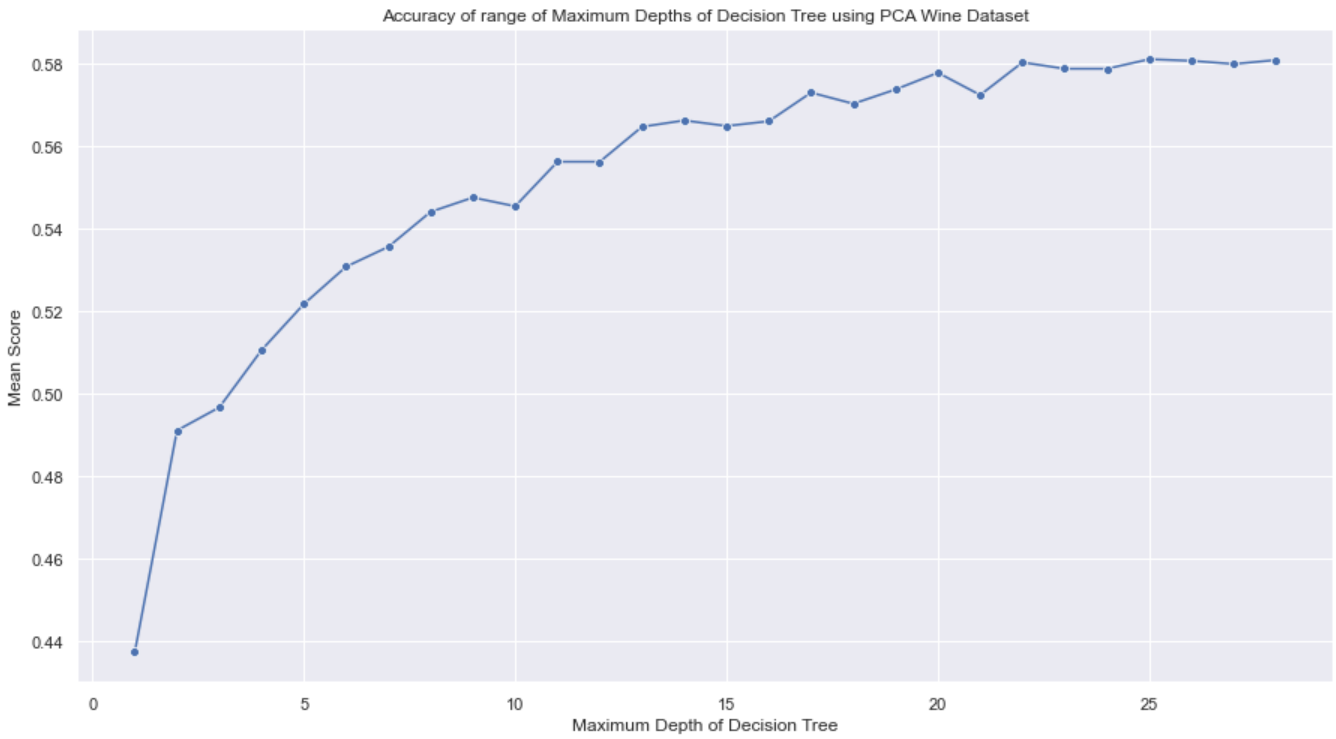


PCA Dataset

In [ ]:
```
# generate a dataframe of the maximum depth and the respective score
wine_pca_tree_cv_results = pd.DataFrame([wine_pca_tree_cv.cv_results_['param_max_depth'],win

# transpose the dataframe
wine_pca_tree_cv_results = wine_pca_tree_cv_results.T

# name the columns of the dataframe
wine_pca_tree_cv_results.columns = ["Maximum Depth of Decision Tree","Mean Score"]
```

In [ ]:
```
# initialize the t-SNE plot
sns.set(rc = {'figure.figsize':(15,8)})

# assign the datapoints for the axes
sns.lineplot(data= wine_pca_tree_cv_results, x="Maximum Depth of Decision Tree", y="Mean Sco

# assign the graph title
plt.title("Accuracy of range of Maximum Depths of Decision Tree using PCA Wine Dataset")
```

Out[ ]:
```
Text(0.5, 1.0, 'Accuracy of range of Maximum Depths of Decision Tree using PCA Wine Dataset')
```

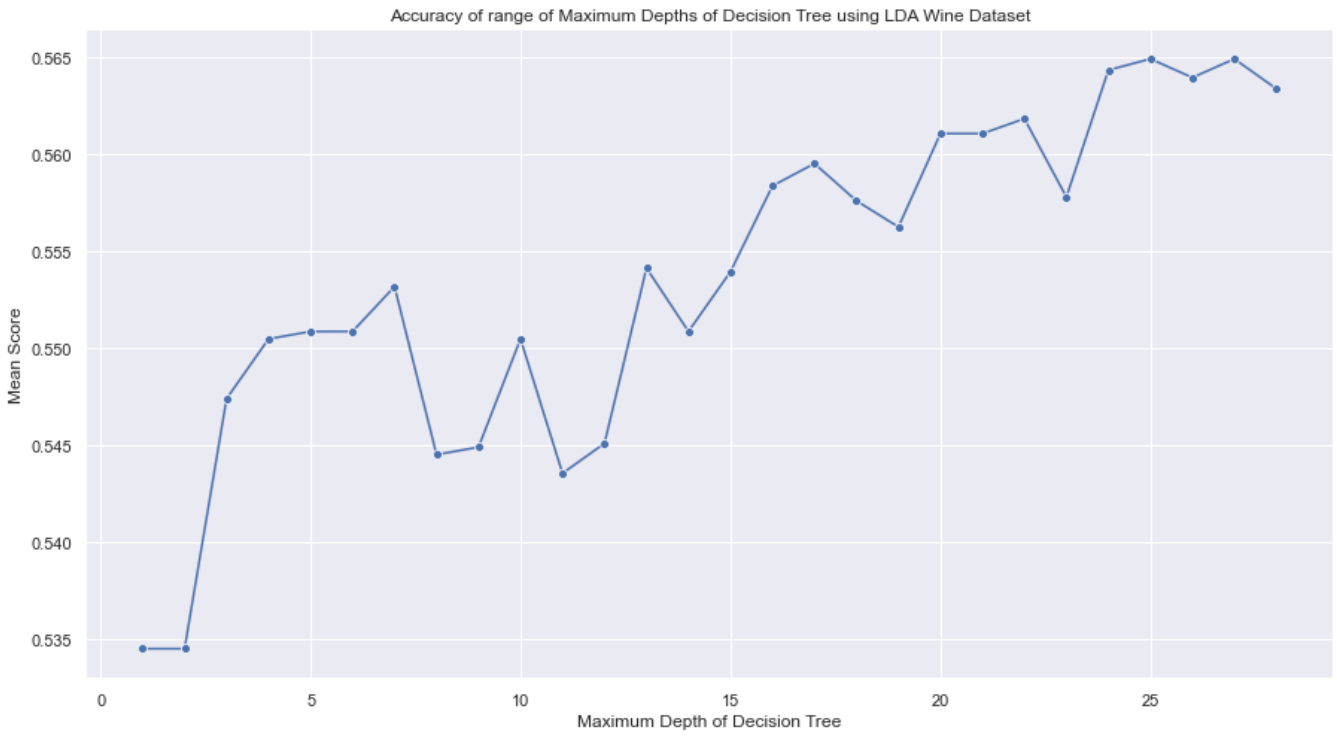


LDA Dataset

In [ ]:
```
# generate a dataframe of the maximum depth and the respective score
wine_lda_tree_cv_results = pd.DataFrame([wine_lda_tree_cv.cv_results_['param_max_depth'],wine

# transpose the dataframe
wine_lda_tree_cv_results = wine_lda_tree_cv_results.T

# name the columns of the dataframe
wine_lda_tree_cv_results.columns = ["Maximum Depth of Decision Tree","Mean Score"]
```

In [ ]:
```
# initialize the t-SNE plot
sns.set(rc = {'figure.figsize':(15,8)})

# assign the datapoints for the axes
sns.lineplot(data= wine_lda_tree_cv_results, x="Maximum Depth of Decision Tree", y="Mean Sco

# assign the graph title
plt.title("Accuracy of range of Maximum Depths of Decision Tree using LDA Wine Dataset")
```

Out[ ]:
```
Text(0.5, 1.0, 'Accuracy of range of Maximum Depths of Decision Tree using LDA Wine Dataset')
```



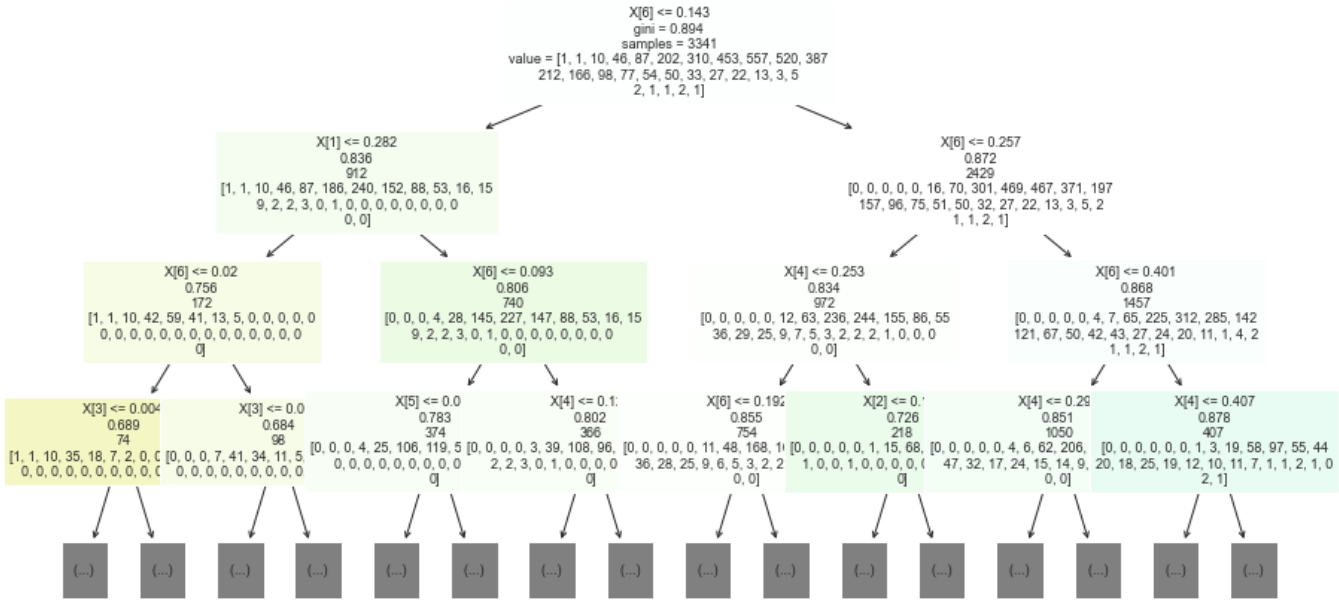
3.3 Interpretability

3.3.1 Abalone Dataset

We will plot the first 3 layers of the decision tree as shown below:

In [ ]:
```
plot_tree(abalone_raw_tree, max_depth = 3, filled = True, fontsize=9, label="root")
```

Out[ ]:
```
[Text(0.5, 0.9, 'X[6] <= 0.143\ngini = 0.894\nsamples = 3341\nvalue = [1, 1, 10, 46, 87, 202, 310, 453, 557, 520, 387\n212, 166, 98, 77, 54, 50, 33, 27, 22, 13, 3, 5\n2, 1, 1, 2, 1]'),
 Text(0.25, 0.7, 'X[1] <= 0.282\n0.836\n912\n[1, 1, 10, 46, 87, 186, 240, 152, 88, 53, 16, 15\n9, 2, 2, 3, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0\n0, 0]'),
 Text(0.125, 0.5, 'X[6] <= 0.02\n0.756\n172\n[1, 1, 10, 42, 59, 41, 13, 5, 0, 0, 0, 0, 0\n0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\n0]'),
 Text(0.0625, 0.3, 'X[3] <= 0.004\n0.689\n74\n[1, 1, 10, 35, 18, 7, 2, 0, 0, 0, 0, 0, 0, 0\n0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]'),
 Text(0.03125, 0.1, '\n  (...)  \n'),
 Text(0.09375, 0.1, '\n  (...)  \n'),
 Text(0.1875, 0.3, 'X[3] <= 0.038\n0.684\n98\n[0, 0, 0, 7, 41, 34, 11, 5, 0, 0, 0, 0, 0, 0\n0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]'),
 Text(0.15625, 0.1, '\n  (...)  \n'),
 Text(0.21875, 0.1, '\n  (...)  \n'),
 Text(0.375, 0.5, 'X[6] <= 0.093\n0.806\n740\n[0, 0, 0, 4, 28, 145, 227, 147, 88, 53, 16, 15\n9, 2, 2, 3, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0\n0, 0]'),
 Text(0.3125, 0.3, 'X[5] <= 0.044\n0.783\n374\n[0, 0, 0, 4, 25, 106, 119, 51, 33, 25, 4, 4, 3\n0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\n0]'),
 Text(0.28125, 0.1, '\n  (...)  \n'),
 Text(0.34375, 0.1, '\n  (...)  \n'),
 Text(0.4375, 0.3, 'X[4] <= 0.129\n0.802\n366\n[0, 0, 0, 0, 3, 39, 108, 96, 55, 28, 12, 11, 6\n2, 2, 3, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0\n0]'),
 Text(0.40625, 0.1, '\n  (...)  \n'),
 Text(0.46875, 0.1, '\n  (...)  \n'),
 Text(0.75, 0.7, 'X[6] <= 0.257\n0.872\n2429\n[0, 0, 0, 0, 0, 16, 70, 301, 469, 467, 371, 197\n157, 96, 75, 51, 50, 32, 27, 22, 13, 3, 5, 2\n1, 1, 2, 1]'),
 Text(0.625, 0.5, 'X[4] <= 0.253\n0.834\n972\n[0, 0, 0, 0, 0, 12, 63, 236, 244, 155, 86, 55\n36, 29, 25, 9, 7, 5, 3, 2, 2, 2, 1, 0, 0, 0\n0, 0]'),
 Text(0.5625, 0.3, 'X[6] <= 0.192\n0.855\n754\n[0, 0, 0, 0, 0, 11, 48, 168, 161, 121, 75, 51\n36, 28, 25, 9, 6, 5, 3, 2, 2, 2, 1, 0, 0, 0\n0, 0]'),
 Text(0.53125, 0.1, '\n  (...)  \n'),
 Text(0.59375, 0.1, '\n  (...)  \n'),
 Text(0.6875, 0.3, 'X[2] <= 0.135\n0.726\n218\n[0, 0, 0, 0, 0, 1, 15, 68, 83, 34, 11, 4, 0\n1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\n0]'),
 Text(0.65625, 0.1, '\n  (...)  \n'),
 Text(0.71875, 0.1, '\n  (...)  \n'),
 Text(0.875, 0.5, 'X[6] <= 0.401\n0.868\n1457\n[0, 0, 0, 0, 0, 4, 7, 65, 225, 312, 285, 142\n121, 67, 50, 42, 43, 27, 24, 20, 11, 1, 4, 2\n1, 1, 2, 1]'),
 Text(0.8125, 0.3, 'X[4] <= 0.298\n0.851\n1050\n[0, 0, 0, 0, 0, 4, 6, 62, 206, 254, 188, 87,\n77\n47, 32, 17, 24, 15, 14, 9, 4, 0, 3, 0, 0, 1\n0, 0]'),
 Text(0.78125, 0.1, '\n  (...)  \n'),
 Text(0.84375, 0.1, '\n  (...)  \n'),
 Text(0.9375, 0.3, 'X[4] <= 0.407\n0.878\n407\n[0, 0, 0, 0, 0, 0, 1, 3, 19, 58, 97, 55, 44\n20, 18, 25, 19, 12, 10, 11, 7, 1, 1, 2, 1, 0\n2, 1]'),
 Text(0.90625, 0.1, '\n  (...)  \n'),
 Text(0.96875, 0.1, '\n  (...)  \n')]
```



Based on the plot of the decision tree, the first node is "Shell Weight (g)" and the threshold is 0.143. The

original samples are 3341 and after the first split, samples that are less than 0.143, there are 912 samples and for samples that are more than the threshold, there are 2429 samples.

On the right hand side of the tree, after the root node, "Shell Weight (g)" is used consecutively for the nodes.

To determine which feature was used for nodes in the tree, we will execute the following code:

In [ ]:
```
# initialize an array to store the feature name
abalone_raw_node_features = []

# initialize a for loop that will run through the node feature of the tree
for i in range(0,len(abalone_raw_tree.tree_.feature)):

    # condition to remove all the leaves
    if (abalone_raw_tree.tree_.feature[i] >= 0):

        # add the feature to the array
        abalone_raw_node_features.append(abalone_raw_train.columns[abalone_raw_tree.tree_.fe

# remove duplicate values from the array by using "set"
set(abalone_raw_node_features)
```

Out[ ]:
```
{'Diameter (mm)',
 'Height (mm)',
 'Length (mm)',
 'Shell Weight (g)',
 'Shucked Weight (g)',
 'Viscera Weight (g)',
 'Whole Weight (g)'}
```

All the features were used for the decision tree

Now we will determine, how many times was each feature was used for a node.

In [ ]:
```
abalone_node_f = pd.DataFrame(np.unique(abalone_raw_node_features, return_counts=True))
abalone_node_f = abalone_node_f.T
abalone_node_f.columns = ["Features", "Frequency"]
abalone_node_f
```

Out[ ]:

	Features	Frequency
0	Diameter (mm)	1
1	Height (mm)	1
2	Length (mm)	2
3	Shell Weight (g)	11
4	Shucked Weight (g)	9
5	Viscera Weight (g)	5
6	Whole Weight (g)	2

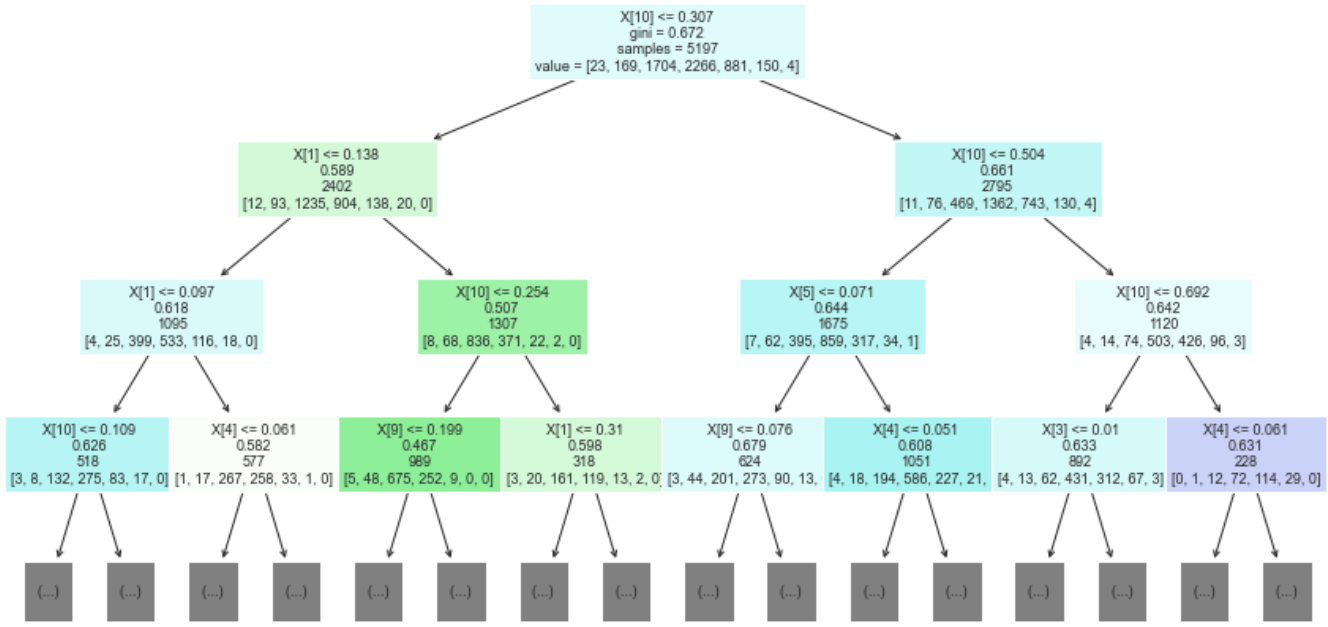
Shell weight was used the most, followed by Shucked Weight (g). Based on observations made on the dataset, the features with the highest variance and with the least impurity was used the most. In this case that is Shell Weight.

3.3.2 Wine Dataset

We will plot the first 3 layers of the decision tree as shown below:

In [ ]:
```
= 3, filled = True, fontsize=9, label="root")
```

Out[ ]:
```
[Text(0.5, 0.9, 'X[10] <= 0.307\ngini = 0.672\nsamples = 5197\nvalue = [23, 169, 1704, 2266,
881, 150, 4]'),
 Text(0.25, 0.7, 'X[1] <= 0.138\n0.589\n2402\n[12, 93, 1235, 904, 138, 20, 0]'),
 Text(0.125, 0.5, 'X[1] <= 0.097\n0.618\n1095\n[4, 25, 399, 533, 116, 18, 0]'),
 Text(0.0625, 0.3, 'X[10] <= 0.109\n0.626\n518\n[3, 8, 132, 275, 83, 17, 0]'),
 Text(0.03125, 0.1, '\n  (...)  \n'),
 Text(0.09375, 0.1, '\n  (...)  \n'),
 Text(0.1875, 0.3, 'X[4] <= 0.061\n0.582\n577\n[1, 17, 267, 258, 33, 1, 0]'),
 Text(0.15625, 0.1, '\n  (...)  \n'),
 Text(0.21875, 0.1, '\n  (...)  \n'),
 Text(0.375, 0.5, 'X[10] <= 0.254\n0.507\n1307\n[8, 68, 836, 371, 22, 2, 0]'),
 Text(0.3125, 0.3, 'X[9] <= 0.199\n0.467\n989\n[5, 48, 675, 252, 9, 0, 0]'),
 Text(0.28125, 0.1, '\n  (...)  \n'),
 Text(0.34375, 0.1, '\n  (...)  \n'),
 Text(0.4375, 0.3, 'X[1] <= 0.31\n0.598\n318\n[3, 20, 161, 119, 13, 2, 0]'),
 Text(0.40625, 0.1, '\n  (...)  \n'),
 Text(0.46875, 0.1, '\n  (...)  \n'),
 Text(0.75, 0.7, 'X[10] <= 0.504\n0.661\n2795\n[11, 76, 469, 1362, 743, 130, 4]'),
 Text(0.625, 0.5, 'X[5] <= 0.071\n0.644\n1675\n[7, 62, 395, 859, 317, 34, 1]'),
 Text(0.5625, 0.3, 'X[9] <= 0.076\n0.679\n624\n[3, 44, 201, 273, 90, 13, 0]'),
 Text(0.53125, 0.1, '\n  (...)  \n'),
 Text(0.59375, 0.1, '\n  (...)  \n'),
 Text(0.6875, 0.3, 'X[4] <= 0.051\n0.608\n1051\n[4, 18, 194, 586, 227, 21, 1]'),
 Text(0.65625, 0.1, '\n  (...)  \n'),
 Text(0.71875, 0.1, '\n  (...)  \n'),
 Text(0.875, 0.5, 'X[10] <= 0.692\n0.642\n1120\n[4, 14, 74, 503, 426, 96, 3]'),
 Text(0.8125, 0.3, 'X[3] <= 0.01\n0.633\n892\n[4, 13, 62, 431, 312, 67, 3]'),
 Text(0.78125, 0.1, '\n  (...)  \n'),
 Text(0.84375, 0.1, '\n  (...)  \n'),
 Text(0.9375, 0.3, 'X[4] <= 0.061\n0.631\n228\n[0, 1, 12, 72, 114, 29, 0]'),
 Text(0.90625, 0.1, '\n  (...)  \n'),
 Text(0.96875, 0.1, '\n  (...)  \n')]
```



To determine which feature was used for nodes in the tree, we will execute the following code:

In [ ]:
```
# initialize an array to store the feature name
wine_raw_node_features = []

# initialize a for loop that will run through the node feature of the tree
for i in range(0,len(wine_raw_tree.tree_.feature)):

    # condition to remove all the leaves
    if (wine_raw_tree.tree_.feature[i] >= 0):

        # add the feature to the array
        wine_raw_node_features.append(wine_raw_train.columns[wine_raw_tree.tree_.feature[i]]
```

```
# remove duplicate values from the array by using "set"
set(wine_raw_node_features)
```

Out[ ]:
```
{'alcohol',
 'chlorides',
 'citric acid',
 'density',
 'fixed acidity',
 'free sulfur dioxide',
 'pH',
 'residual sugar',
 'sulphates',
 'total sulfur dioxide',
 'volatile acidity'}
```

All the features were used for the decision tree

Now we will determine, how many times was each feature was used for a node.

In [ ]:
```
wine_node_f = pd.DataFrame(np.unique(wine_raw_node_features, return_counts=True))
wine_node_f = wine_node_f.T
wine_node_f.columns = ["Features", "Frequency"]
wine_node_f
```

Out[ ]:

	Features	Frequency
0	alcohol	103
1	chlorides	126
2	citric acid	130
3	density	132
4	fixed acidity	123
5	free sulfur dioxide	131
6	pH	139
7	residual sugar	111
8	sulphates	139
9	total sulfur dioxide	145
10	volatile acidity	122

Total sulfur dioxide has been used the most followed by sulphates. In the case of wine dataset, almost all the feaures have been used almost the same for the best decision tree.

Question 4: Random Forest Classifier

4.1 Random Forest using Cross Validation

4.1.1 Abalone Dataset

For the maximum depth of the tree in the random forest classifier, we will use the same that was used in the decision tree algorithm, which for the case of abalone is 30

Normalized Data

In [ ]:
```
# create an array for the maximum depths for the classifier parameters
abalone_rf_tree_depth = list(range(2,29,5))

# create an array for the number of estimators for the classifier parameters
abalone_rf_no_estimators = list(range(3,304,50))
```

```
# generate a dictionary for the parameters of random forest classifier
parameters = {"n_estimators": abalone_rf_no_estimators ,"max_depth":abalone_rf_tree_depth}

# initialize the GridSearchCV function
abalone_raw_rforest_cv = GridSearchCV(RandomForestClassifier(random_state=27), parameters, cv

# fit the training data using the GridSearchCV function
abalone_raw_rforest_cv.fit(abalone_raw_train , abalone_rings_raw_train)

# display the results of the tuning of the hyperprameters
abalone_raw_rforest_cv.cv_results_
```

```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\model_select
ion\_split.py:676: UserWarning: The least populated class in y has only 1 members, which is l
ess than n_splits=5.
  warnings.warn(
```

Out[ ]:
```
{'mean_fit_time': array([0.02920032, 0.32559977, 0.45640097, 1.00611014, 1.1062005 ,
       1.11459751, 1.30387034, 0.02599974, 0.35440016, 0.68439889,
       1.34880009, 1.39919815, 1.73865833, 2.06006455, 0.03339863,
       0.55169487, 1.0028904 , 1.57759819, 2.15739679, 2.56020422,
       2.95519962, 0.03819833, 0.60119739, 1.17199512, 1.75299568,
       2.29639993, 2.87439766, 3.43379989, 0.041399  , 0.63759656,
       1.24139895, 1.90270929, 2.85660133, 2.97440267, 3.59946661,
       0.03679919, 0.56121721, 1.51134896, 2.6738812 , 2.45005093,
       3.05146766, 3.65519962]),
 'std_fit_time': array([0.00823396, 0.06132626, 0.11103798, 0.28454887, 0.1749676 ,
       0.06264533, 0.08328069, 0.00369056, 0.00313549, 0.01578433,
       0.10570407, 0.07257194, 0.08069443, 0.04430974, 0.00300662,
       0.06787973, 0.01460739, 0.0966318 , 0.23926468, 0.1335484 ,
       0.05096738, 0.00299156, 0.01041068, 0.02673519, 0.05450596,
       0.04909119, 0.05852327, 0.06387856, 0.00516217, 0.02299929,
       0.0729063 , 0.17401503, 0.47571227, 0.07537884, 0.08994497,
       0.00203747, 0.01781956, 0.44738931, 0.57457842, 0.06250854,
       0.09344869, 0.09540878]),
 'mean_score_time': array([0.00880194, 0.03080044, 0.06459794, 0.11760097, 0.1180028 ,
       0.13880105, 0.14040155, 0.00640073, 0.03060017, 0.05459747,
       0.10439868, 0.10380263, 0.13040824, 0.15460114, 0.00700274,
       0.03320308, 0.06379933, 0.10179696, 0.14340386, 0.16099577,
       0.18319731, 0.00620165, 0.03700113, 0.0660018 , 0.10140471,
       0.13000002, 0.16259952, 0.18980484, 0.00640001, 0.03580313,
       0.06960239, 0.14759631, 0.15600348, 0.15939808, 0.1964046 ,
       0.00520158, 0.03100243, 0.07499995, 0.14407425, 0.15040722,
       0.1683907 , 0.22280049]),
 'std_score_time': array([0.0016005 , 0.00330946, 0.03537597, 0.04651514, 0.01952471,
       0.01328974, 0.00463049, 0.00249832, 0.00233275, 0.00215123,
       0.02459438, 0.00204255, 0.00531078, 0.00624872, 0.00351744,
       0.00132685, 0.00318904, 0.01404993, 0.02760918, 0.01890149,
       0.00649411, 0.00213404, 0.00316278, 0.00063468, 0.00833372,
       0.00414881, 0.00995112, 0.00271237, 0.00135616, 0.0007488 ,
       0.00571404, 0.09639675, 0.03947215, 0.00241275, 0.01671427,
       0.00097986, 0.00141354, 0.02301474, 0.03273742, 0.03784163,
       0.01285294, 0.05145738]),
 'param_max_depth': masked_array(data=[2, 2, 2, 2, 2, 2, 2, 7, 7, 7, 7, 7, 7, 7, 12, 12, 12,
                    12, 12, 12, 12, 17, 17, 17, 17, 17, 17, 17, 22, 22, 22,
                    22, 22, 22, 22, 27, 27, 27, 27, 27, 27, 27],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False],
       fill_value='?',
            dtype=object),
 'param_n_estimators': masked_array(data=[3, 53, 103, 153, 203, 253, 303, 3, 53, 103, 153, 203,
                    253, 303, 3, 53, 103, 153, 203, 253, 303, 3, 53, 103,
                    153, 203, 253, 303, 3, 53, 103, 153, 203, 253, 303, 3,
                    53, 103, 153, 203, 253, 303],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False],
       fill_value='?',
            dtype=object),
 'params': [{'max_depth': 2, 'n_estimators': 3},
  {'max_depth': 2, 'n_estimators': 53},
  {'max_depth': 2, 'n_estimators': 103},
  {'max_depth': 2, 'n_estimators': 153},
  {'max_depth': 2, 'n_estimators': 203},
  {'max_depth': 2, 'n_estimators': 253},
  {'max_depth': 2, 'n_estimators': 303},
  {'max_depth': 7, 'n_estimators': 3},
  {'max_depth': 7, 'n_estimators': 53},
```

```
 {'max_depth': 7, 'n_estimators': 103},
 {'max_depth': 7, 'n_estimators': 153},
 {'max_depth': 7, 'n_estimators': 203},
 {'max_depth': 7, 'n_estimators': 253},
 {'max_depth': 7, 'n_estimators': 303},
 {'max_depth': 12, 'n_estimators': 3},
 {'max_depth': 12, 'n_estimators': 53},
 {'max_depth': 12, 'n_estimators': 103},
 {'max_depth': 12, 'n_estimators': 153},
 {'max_depth': 12, 'n_estimators': 203},
 {'max_depth': 12, 'n_estimators': 253},
 {'max_depth': 12, 'n_estimators': 303},
 {'max_depth': 17, 'n_estimators': 3},
 {'max_depth': 17, 'n_estimators': 53},
 {'max_depth': 17, 'n_estimators': 103},
 {'max_depth': 17, 'n_estimators': 153},
 {'max_depth': 17, 'n_estimators': 203},
 {'max_depth': 17, 'n_estimators': 253},
 {'max_depth': 17, 'n_estimators': 303},
 {'max_depth': 22, 'n_estimators': 3},
 {'max_depth': 22, 'n_estimators': 53},
 {'max_depth': 22, 'n_estimators': 103},
 {'max_depth': 22, 'n_estimators': 153},
 {'max_depth': 22, 'n_estimators': 203},
 {'max_depth': 22, 'n_estimators': 253},
 {'max_depth': 22, 'n_estimators': 303},
 {'max_depth': 27, 'n_estimators': 3},
 {'max_depth': 27, 'n_estimators': 53},
 {'max_depth': 27, 'n_estimators': 103},
 {'max_depth': 27, 'n_estimators': 153},
 {'max_depth': 27, 'n_estimators': 203},
 {'max_depth': 27, 'n_estimators': 253},
 {'max_depth': 27, 'n_estimators': 303}],
'split0_test_score': array([0.26307922, 0.27653214, 0.27653214, 0.27802691, 0.27204783,
       0.27055306, 0.2735426 , 0.22122571, 0.26307922, 0.2735426 ,
       0.2690583 , 0.26307922, 0.26606876, 0.26756353, 0.23019432,
       0.25261584, 0.26606876, 0.26158445, 0.26307922, 0.26457399,
       0.26756353, 0.21524664, 0.26756353, 0.25112108, 0.24364723,
       0.24813154, 0.26008969, 0.26606876, 0.21524664, 0.24364723,
       0.23916293, 0.25261584, 0.245142  , 0.24663677, 0.25112108,
       0.21973094, 0.24962631, 0.24962631, 0.23766816, 0.245142  ,
       0.24663677, 0.245142  ]),
'split1_test_score': array([0.23952096, 0.24251497, 0.24251497, 0.24401198, 0.24700599,
       0.25      , 0.24550898, 0.23952096, 0.24850299, 0.24700599,
       0.26047904, 0.25598802, 0.26347305, 0.26047904, 0.22155689,
       0.26047904, 0.26197605, 0.25598802, 0.26796407, 0.26646707,
       0.27095808, 0.19760479, 0.23952096, 0.25898204, 0.25449102,
       0.25898204, 0.26047904, 0.25898204, 0.2005988 , 0.24700599,
       0.26646707, 0.26497006, 0.26197605, 0.26497006, 0.26197605,
       0.17664671, 0.24850299, 0.26197605, 0.26646707, 0.26197605,
       0.26047904, 0.26197605]),
'split2_test_score': array([0.27095808, 0.25449102, 0.25      , 0.25299401, 0.25299401,
       0.25449102, 0.25748503, 0.25149701, 0.25      , 0.26047904,
       0.26197605, 0.26197605, 0.25898204, 0.26646707, 0.21107784,
       0.23203593, 0.23952096, 0.25      , 0.24700599, 0.25      ,
       0.24401198, 0.2005988 , 0.22155689, 0.22754491, 0.23502994,
       0.23203593, 0.23502994, 0.23053892, 0.19610778, 0.23353293,
       0.22904192, 0.22005988, 0.22904192, 0.22904192, 0.23353293,
       0.21407186, 0.22305389, 0.23802395, 0.24251497, 0.24401198,
       0.23203593, 0.23802395]),
'split3_test_score': array([0.26047904, 0.2739521 , 0.2754491 , 0.26646707, 0.26946108,
       0.26946108, 0.26646707, 0.22904192, 0.27245509, 0.2754491 ,
       0.2739521 , 0.2739521 , 0.2739521 , 0.28143713, 0.21556886,
       0.26197605, 0.26197605, 0.2739521 , 0.26796407, 0.26047904,
       0.26646707, 0.20658683, 0.25898204, 0.26796407, 0.26197605,
       0.26497006, 0.25748503, 0.26497006, 0.21107784, 0.23353293,
       0.23802395, 0.24101796, 0.23952096, 0.24251497, 0.24850299,
       0.24251497, 0.24700599, 0.24401198, 0.23952096, 0.23203593,
       0.24251497, 0.24700599]),
```

```
 'split4_test_score': array([0.25449102, 0.23952096, 0.24401198, 0.24700599, 0.24850299,
        0.24700599, 0.24251497, 0.24550898, 0.24850299, 0.25149701,
        0.25299401, 0.25149701, 0.25898204, 0.25598802, 0.21407186,
        0.26197605, 0.25299401, 0.25299401, 0.25598802, 0.25598802,
        0.25598802, 0.19461078, 0.2245509 , 0.2245509 , 0.23053892,
        0.23952096, 0.23952096, 0.24101796, 0.21856287, 0.25449102,
        0.25      , 0.25149701, 0.24101796, 0.24401198, 0.25      ,
        0.20658683, 0.23652695, 0.24101796, 0.24550898, 0.24251497,
        0.24850299, 0.25449102]),
 'mean_test_score': array([0.25770566, 0.25740224, 0.25770164, 0.25770119, 0.25800238,
        0.25830223, 0.25710373, 0.23735891, 0.25650806, 0.26159475,
        0.2636919 , 0.26129848, 0.2642916 , 0.26638696, 0.21849395,
        0.25381658, 0.25650717, 0.25890372, 0.26040028, 0.25950162,
        0.26099774, 0.20292957, 0.24243486, 0.2460326 , 0.24513663,
        0.2487281 , 0.25052093, 0.25231555, 0.20831879, 0.24244202,
        0.24453917, 0.24603215, 0.24333978, 0.24543514, 0.24902661,
        0.20442523, 0.24004502, 0.24693125, 0.24633603, 0.24513619,
        0.24603394, 0.2493278 ]),
 'std_test_score': array([0.01051866, 0.01542528, 0.01514541, 0.01275864, 0.01062844,
        0.00985556, 0.01187707, 0.01095843, 0.00967476, 0.0114082 ,
        0.00723576, 0.00758965, 0.00554142, 0.00861324, 0.00677333,
        0.01143137, 0.0095091 , 0.00844206, 0.00800592, 0.00596369,
        0.00985743, 0.00732221, 0.0182728 , 0.01719202, 0.0117348 ,
        0.01210537, 0.01095622, 0.01411089, 0.00859367, 0.00807718,
        0.01282318, 0.01504364, 0.01072345, 0.01150954, 0.00909231,
        0.0148599 , 0.00970066, 0.0084432 , 0.01041269, 0.00962634,
        0.0092027 , 0.00821372]),
 'rank_test_score': array([12, 15, 13, 14, 11, 10, 16, 38, 17,  4,  3,  5,  2,  1, 39, 19, 1
8,
         9,  7,  8,  6, 42, 36, 28, 31, 24, 21, 20, 40, 35, 33, 29, 34, 30,
        23, 41, 37, 25, 26, 32, 27, 22])}
```

The best hyperparameters for the random forest classifier are:

In [ ]:
```
abalone_raw_rforest_cv.best_params_
```

Out[ ]:
```
{'max_depth': 7, 'n_estimators': 303}
```

The score achieved by the best hyperparameters is:

In [ ]:
```
abalone_raw_rforest_cv.best_score_
```

Out[ ]:
```
0.26638695702764875
```

Now we will initialize a new random forest and train it with the best hyperparameters that were discovered by GridSearchCV

In [ ]:
```
# initialize the decision tree classifier with the best hyperparemeter
abalone_raw_rforest = RandomForestClassifier(random_state=27, max_depth = 7, n_estimators = 

# train the classifier using the training dataset
abalone_raw_rforest.fit(abalone_raw_train, abalone_rings_raw_train)

# view the accuracy of the classifier using the testing dataset
abalone_raw_rforest_test_accuracy = abalone_raw_rforest.score(abalone_raw_test, abalone_ring
abalone_raw_rforest_test_accuracy
```

Out[ ]:
```
0.27751196172248804
```

PCA Data

In [ ]:
```
# create an array for the maximum depths for the classifier parameters
2,29,5))
```

```
# create an array for the number of estimators for the classifier parameters
abalone_rf_no_estimators = list(range(3,304,50))

# generate a dictionary for the parameters of random forest classifier
parameters = {"n_estimators": abalone_rf_no_estimators ,"max_depth":abalone_rf_tree_depth}

# initialize the GridSearchCV function
abalone_pca_rforest_cv = GridSearchCV(RandomForestClassifier(random_state=27), parameters, cv

# fit the training data using the GridSearchCV function
abalone_pca_rforest_cv.fit(abalone_PCAminmax_train , abalone_rings_PCAminmax_train)

# display the results of the tuning of the hyperprameters
abalone_pca_rforest_cv.cv_results_
```

```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\model_select
ion\_split.py:676: UserWarning: The least populated class in y has only 1 members, which is l
ess than n_splits=5.
  warnings.warn(
```

Out[ ]:
```
{'mean_fit_time': array([0.02039742, 0.23239756, 0.467801  , 0.66840153, 1.29219942,
       1.22199888, 1.64916921, 0.02280002, 0.37404709, 0.83038921,
       0.91931243, 1.33200035, 1.92230968, 2.52960382, 0.02619896,
       0.43100452, 0.77979913, 1.14399853, 1.67059984, 1.89279566,
       2.25239944, 0.02939568, 0.4542007 , 0.90419927, 1.32359543,
       1.77619953, 2.20180178, 2.62940164, 0.03219938, 0.48139844,
       0.92380095, 1.36240048, 2.0432024 , 2.37559967, 3.04439936,
       0.02859497, 0.47840042, 0.9718008 , 1.48659911, 2.01679931,
       3.22200289, 4.29879708]),
 'std_fit_time': array([1.85505814e-03, 1.47171994e-02, 5.56705374e-02, 2.12360665e-02,
       2.58152255e-01, 1.69389309e-01, 1.22132620e-01, 3.86508194e-04,
       3.55053614e-02, 1.57143054e-01, 1.57956042e-02, 2.10014070e-01,
       3.67352948e-01, 2.87370936e-01, 7.49144875e-04, 5.17222118e-02,
       2.36752914e-02, 1.89992012e-02, 1.98751006e-01, 5.46786812e-02,
       3.69688955e-02, 1.20132590e-03, 1.51370653e-02, 4.94042127e-02,
       2.95968101e-02, 5.92214568e-02, 4.26927270e-02, 4.52326360e-02,
       3.54442296e-03, 2.36804987e-02, 2.21148351e-02, 1.22742933e-02,
       2.02826342e-01, 7.58623300e-02, 2.29764580e-01, 4.90677414e-04,
       1.53943683e-02, 3.07256418e-02, 6.53322284e-02, 1.45175615e-01,
       6.54255507e-01, 7.87527745e-01]),
 'mean_score_time': array([0.00660334, 0.02900105, 0.05019794, 0.09960055, 0.11399865,
       0.14880314, 0.15160847, 0.00580716, 0.03220067, 0.09279895,
       0.08000879, 0.10379982, 0.20760212, 0.1991992 , 0.00420117,
       0.02939763, 0.05480108, 0.07679996, 0.13460217, 0.12440386,
       0.15120268, 0.00500216, 0.03140001, 0.05560412, 0.08160291,
       0.10860167, 0.13459883, 0.16300149, 0.00480146, 0.03600259,
       0.05740242, 0.0806016 , 0.14419656, 0.13700233, 0.18400478,
       0.00520668, 0.03039727, 0.06120028, 0.09139876, 0.11740379,
       0.23940001, 0.39800248]),
 'std_score_time': array([0.00196059, 0.0010945 , 0.00172038, 0.03355223, 0.01559353,
       0.05757858, 0.01182606, 0.00147267, 0.00652257, 0.03212021,
       0.00782221, 0.01119736, 0.06655766, 0.04747779, 0.00040054,
       0.00102072, 0.00194093, 0.0011691 , 0.04159812, 0.00249971,
       0.00270952, 0.00063399, 0.00135665, 0.00049324, 0.00257896,
       0.00272996, 0.00174448, 0.00766791, 0.00040016, 0.01052723,
       0.00287262, 0.00049225, 0.04469411, 0.00316107, 0.0297209 ,
       0.00040528, 0.00048533, 0.00518653, 0.00741969, 0.01399718,
       0.09289173, 0.2215547 ]),
 'param_max_depth': masked_array(data=[2, 2, 2, 2, 2, 2, 2, 7, 7, 7, 7, 7, 7, 7, 12, 12, 12,
                    12, 12, 12, 12, 17, 17, 17, 17, 17, 17, 17, 22, 22, 22,
                    22, 22, 22, 22, 27, 27, 27, 27, 27, 27, 27],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False],
        fill_value='?',
             dtype=object),
 'param_n_estimators': masked_array(data=[3, 53, 103, 153, 203, 253, 303, 3, 53, 103, 153, 20
3,
                    253, 303, 3, 53, 103, 153, 203, 253, 303, 3, 53, 103,
                    153, 203, 253, 303, 3, 53, 103, 153, 203, 253, 303, 3,
                    53, 103, 153, 203, 253, 303],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False],
        fill_value='?',
             dtype=object),
 'params': [{'max_depth': 2, 'n_estimators': 3},
  {'max_depth': 2, 'n_estimators': 53},
  {'max_depth': 2, 'n_estimators': 103},
  {'max_depth': 2, 'n_estimators': 153},
  {'max_depth': 2, 'n_estimators': 203},
  {'max_depth': 2, 'n_estimators': 253},
  {'max_depth': 2, 'n_estimators': 303},
```

```
 {'max_depth': 7, 'n_estimators': 3},
 {'max_depth': 7, 'n_estimators': 53},
 {'max_depth': 7, 'n_estimators': 103},
 {'max_depth': 7, 'n_estimators': 153},
 {'max_depth': 7, 'n_estimators': 203},
 {'max_depth': 7, 'n_estimators': 253},
 {'max_depth': 7, 'n_estimators': 303},
 {'max_depth': 12, 'n_estimators': 3},
 {'max_depth': 12, 'n_estimators': 53},
 {'max_depth': 12, 'n_estimators': 103},
 {'max_depth': 12, 'n_estimators': 153},
 {'max_depth': 12, 'n_estimators': 203},
 {'max_depth': 12, 'n_estimators': 253},
 {'max_depth': 12, 'n_estimators': 303},
 {'max_depth': 17, 'n_estimators': 3},
 {'max_depth': 17, 'n_estimators': 53},
 {'max_depth': 17, 'n_estimators': 103},
 {'max_depth': 17, 'n_estimators': 153},
 {'max_depth': 17, 'n_estimators': 203},
 {'max_depth': 17, 'n_estimators': 253},
 {'max_depth': 17, 'n_estimators': 303},
 {'max_depth': 22, 'n_estimators': 3},
 {'max_depth': 22, 'n_estimators': 53},
 {'max_depth': 22, 'n_estimators': 103},
 {'max_depth': 22, 'n_estimators': 153},
 {'max_depth': 22, 'n_estimators': 203},
 {'max_depth': 22, 'n_estimators': 253},
 {'max_depth': 22, 'n_estimators': 303},
 {'max_depth': 27, 'n_estimators': 3},
 {'max_depth': 27, 'n_estimators': 53},
 {'max_depth': 27, 'n_estimators': 103},
 {'max_depth': 27, 'n_estimators': 153},
 {'max_depth': 27, 'n_estimators': 203},
 {'max_depth': 27, 'n_estimators': 253},
 {'max_depth': 27, 'n_estimators': 303}],
'split0_test_score': array([0.22421525, 0.21674141, 0.22272048, 0.21973094, 0.21823617,
       0.21973094, 0.21973094, 0.23617339, 0.24663677, 0.25261584,
       0.26307922, 0.26307922, 0.26158445, 0.26606876, 0.2122571 ,
       0.25859492, 0.26307922, 0.25112108, 0.25560538, 0.24663677,
       0.24364723, 0.1838565 , 0.24962631, 0.24364723, 0.245142  ,
       0.24962631, 0.25411061, 0.25112108, 0.19880419, 0.23318386,
       0.24813154, 0.25261584, 0.25112108, 0.25112108, 0.24813154,
       0.20179372, 0.24364723, 0.25411061, 0.25411061, 0.24364723,
       0.245142  , 0.24663677]),
'split1_test_score': array([0.25748503, 0.25598802, 0.26047904, 0.25299401, 0.25598802,
       0.25149701, 0.25149701, 0.25449102, 0.2754491 , 0.26047904,
       0.27994012, 0.27694611, 0.27844311, 0.28143713, 0.21706587,
       0.22904192, 0.23353293, 0.23203593, 0.23652695, 0.23802395,
       0.23802395, 0.19610778, 0.2260479 , 0.22754491, 0.2260479 ,
       0.2245509 , 0.21856287, 0.21856287, 0.2245509 , 0.22305389,
       0.22155689, 0.23203593, 0.2260479 , 0.22305389, 0.22904192,
       0.20658683, 0.22904192, 0.2260479 , 0.23353293, 0.22754491,
       0.22155689, 0.22754491]),
'split2_test_score': array([0.2260479 , 0.25      , 0.24251497, 0.24101796, 0.24251497,
       0.24401198, 0.23952096, 0.23203593, 0.23502994, 0.25149701,
       0.25748503, 0.25598802, 0.25898204, 0.26047904, 0.24850299,
       0.24850299, 0.24850299, 0.24700599, 0.24850299, 0.24251497,
       0.24251497, 0.22754491, 0.22754491, 0.23802395, 0.23952096,
       0.23502994, 0.22904192, 0.2260479 , 0.21257485, 0.23353293,
       0.23802395, 0.22904192, 0.23502994, 0.23053892, 0.22754491,
       0.19311377, 0.24401198, 0.24251497, 0.24401198, 0.23802395,
       0.23053892, 0.23203593]),
'split3_test_score': array([0.23353293, 0.24401198, 0.2260479 , 0.24101796, 0.23652695,
       0.24101796, 0.24401198, 0.25149701, 0.26347305, 0.26197605,
       0.27245509, 0.27095808, 0.26796407, 0.26347305, 0.20508982,
       0.23502994, 0.23952096, 0.23952096, 0.24401198, 0.25449102,
       0.25149701, 0.18712575, 0.21407186, 0.23353293, 0.23203593,
       0.2260479 , 0.23353293, 0.22904192, 0.18862275, 0.2260479 ,
       0.2260479 , 0.23053892, 0.2260479 , 0.23353293, 0.22305389,
```

```
        0.21107784, 0.23952096, 0.23502994, 0.22754491, 0.2260479 ,
        0.2260479 , 0.2260479 ]),
 'split4_test_score': array([0.23652695, 0.26197605, 0.25149701, 0.24850299, 0.25      ,
        0.24850299, 0.25      , 0.21706587, 0.24401198, 0.25      ,
        0.25149701, 0.25449102, 0.25299401, 0.25299401, 0.20359281,
        0.23353293, 0.22904192, 0.24101796, 0.23502994, 0.22754491,
        0.22904192, 0.19161677, 0.21107784, 0.22155689, 0.2245509 ,
        0.22305389, 0.21706587, 0.21706587, 0.21556886, 0.22904192,
        0.22754491, 0.23053892, 0.23502994, 0.23353293, 0.23802395,
        0.19610778, 0.2260479 , 0.21856287, 0.22305389, 0.2260479 ,
        0.23053892, 0.23053892]),
 'mean_test_score': array([0.23556161, 0.24574349, 0.24065188, 0.24065278, 0.24065322,
        0.24095218, 0.24095218, 0.23825264, 0.25292017, 0.25531359,
        0.26489129, 0.26429249, 0.26399354, 0.2648904 , 0.21730172,
        0.24094054, 0.24273561, 0.24214038, 0.24393545, 0.24184232,
        0.24094502, 0.19725034, 0.22567376, 0.23286118, 0.23345954,
        0.2337576 , 0.22956464, 0.22836793, 0.20802431, 0.2289721 ,
        0.23226104, 0.23495431, 0.23465535, 0.23435595, 0.23315924,
        0.20173599, 0.236454  , 0.23525326, 0.23645087, 0.23226238,
        0.23076493, 0.23256089]),
 'std_test_score': array([0.01187379, 0.01568874, 0.01448458, 0.0114186 , 0.01300411,
        0.01120639, 0.01144385, 0.01364513, 0.01454519, 0.00492237,
        0.01021001, 0.00861451, 0.00867714, 0.00936136, 0.01634679,
        0.01095676, 0.01208028, 0.00655054, 0.00763216, 0.00897134,
        0.00736651, 0.01570163, 0.01360155, 0.00773943, 0.00786821,
        0.00959622, 0.01326883, 0.01222635, 0.01274509, 0.00405268,
        0.00959696, 0.00888138, 0.00916054, 0.00921662, 0.00892666,
        0.00658519, 0.0075033 , 0.01242328, 0.01128137, 0.00724299,
        0.00792264, 0.00734947]),
 'rank_test_score': array([22,  7, 18, 17, 16, 12, 12, 19,  6,  5,  1,  3,  4,  2, 39, 15,
9,
        10,  8, 11, 14, 42, 38, 30, 28, 27, 35, 37, 40, 36, 33, 24, 25, 26,
        29, 41, 20, 23, 21, 32, 34, 31])}
```

The best hyperparameters for the random forest classifier are:

In [ ]:
```
abalone_pca_rforest_cv.best_params_
```

Out[ ]:
```
{'max_depth': 7, 'n_estimators': 153}
```

The score achieved by the best hyperparameters is:

In [ ]:
```
abalone_pca_rforest_cv.best_score_
```

Out[ ]:
```
0.2648912936458921
```

Now we will initialize a new random forest and train it with the best hyperparameters that were discovered by GridSearchCV

In [ ]:
```
# initialize the decision tree classifier with the best hyperparemeter
abalone_pca_rforest = RandomForestClassifier(random_state = 27, max_depth = 7, n_estimators =

# train the classifier using the training dataset
abalone_pca_rforest.fit(abalone_PCAminmax_train , abalone_rings_PCAminmax_train)

# view the accuracy of the classifier using the testing dataset
abalone_pca_rforest_test_accuracy = abalone_pca_rforest.score(abalone_PCAminmax_test , abalo
abalone_pca_rforest_test_accuracy
```

Out[ ]:
```
0.2619617224880383
```

The performance of the best classifier on the test data is shown below:

In [ ]:
```
_PCAminmax_test , abalone_rings_PCAminmax_test)
```

Out[ ]:
```
0.2619617224880383
```

LDA Data

In [ ]:
```
# create an array for the maximum depths for the classifier parameters
abalone_rf_tree_depth = list(range(2,29,5))

# create an array for the number of estimators for the classifier parameters
abalone_rf_no_estimators = list(range(3,304,50))

# generate a dictionary for the parameters of random forest classifier
parameters = {"n_estimators": abalone_rf_no_estimators ,"max_depth":abalone_rf_tree_depth}

# initialize the GridSearchCV function
abalone_lda_rforest_cv = GridSearchCV(RandomForestClassifier(random_state=27), parameters, cv

# fit the training data using the GridSearchCV function
abalone_lda_rforest_cv.fit(abalone_LDAminmax_train , abalone_rings_LDAminmax_train)

# display the results of the tuning of the hyperprameters
abalone_lda_rforest_cv.cv_results_
```

```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\model_select
ion\_split.py:676: UserWarning: The least populated class in y has only 1 members, which is l
ess than n_splits=5.
  warnings.warn(
```

Out[ ]:
```
{'mean_fit_time': array([0.04600086, 0.43015275, 0.61660147, 1.06480513, 1.44069471,
       1.50584741, 1.28799701, 0.02659788, 0.40119548, 0.79299679,
       1.18599977, 1.52416244, 2.54640102, 2.54899688, 0.04059763,
       0.60559535, 1.10579767, 1.62319927, 2.40600142, 4.26320066,
       3.23499875, 0.03759627, 0.70961456, 1.21199837, 2.02299919,
       2.44159851, 2.99220014, 3.61460094, 0.03939624, 0.70500059,
       1.66780062, 2.56339669, 3.0579988 , 3.76700025, 4.47980156,
       0.0463974 , 0.77199936, 1.52059784, 2.24739985, 3.01720123,
       3.84799843, 4.50860052]),
 'std_fit_time': array([4.19734331e-03, 1.28176668e-01, 6.19270880e-02, 1.82155996e-01,
       5.33139589e-01, 2.59379570e-01, 7.16775261e-02, 8.01516920e-04,
       1.02801721e-02, 2.88046502e-02, 5.75231966e-02, 2.32663435e-02,
       2.69752856e-01, 5.92348596e-02, 2.57767052e-03, 2.64200160e-02,
       3.30048731e-02, 2.15716885e-02, 2.10744921e-01, 1.02706893e+00,
       4.34162485e-02, 4.89920847e-04, 1.09048091e-01, 9.27658683e-03,
       1.29650436e-01, 5.49073154e-02, 2.05569477e-02, 7.28932818e-02,
       4.91191769e-04, 1.16604788e-01, 1.09142566e-01, 2.90374543e-01,
       1.14456705e-01, 1.44903606e-01, 8.09454354e-02, 5.42622350e-03,
       1.94637460e-02, 3.44952255e-02, 6.64673502e-02, 7.75382261e-02,
       8.98950583e-02, 1.25023640e-01]),
 'mean_score_time': array([0.00799856, 0.03980203, 0.05839767, 0.12091241, 0.15660152,
       0.15293207, 0.12480168, 0.00460248, 0.02500238, 0.06711578,
       0.06699877, 0.08559961, 0.14539981, 0.15259991, 0.00600433,
       0.03120503, 0.05080314, 0.07960076, 0.10560083, 0.18500109,
       0.15839987, 0.00500441, 0.0424047 , 0.05420227, 0.09520383,
       0.10420108, 0.12999902, 0.15560112, 0.00460296, 0.03499999,
       0.07480068, 0.15580196, 0.14200048, 0.15139837, 0.19660034,
       0.00660367, 0.03940139, 0.08200164, 0.09760108, 0.14579811,
       0.17380185, 0.20759897]),
 'std_score_time': array([2.76279047e-03, 1.63016081e-02, 5.04822688e-03, 2.46350314e-02,
       1.09516630e-01, 4.14114657e-02, 1.86996316e-02, 4.90507934e-04,
       1.54972871e-03, 3.91632582e-02, 3.28868115e-03, 4.91019308e-04,
       3.12655355e-02, 1.61779383e-02, 8.89360334e-04, 3.18612868e-03,
       7.50675644e-04, 7.17247200e-03, 9.78992070e-03, 5.12909903e-02,
       2.23579594e-02, 1.03375833e-06, 2.63022811e-02, 4.02240602e-04,
       9.04106260e-03, 9.77092570e-04, 2.60773198e-03, 3.00858665e-03,
       4.89824235e-04, 9.52721773e-03, 1.51486195e-02, 3.93804876e-02,
       1.56624456e-02, 1.27549438e-02, 2.45035811e-02, 2.41658805e-03,
       8.40376865e-03, 2.44044270e-02, 1.77476110e-02, 2.85040787e-02,
       2.16661829e-02, 9.11087821e-03]),
 'param_max_depth': masked_array(data=[2, 2, 2, 2, 2, 2, 2, 7, 7, 7, 7, 7, 7, 7, 12, 12, 12,
                    12, 12, 12, 12, 17, 17, 17, 17, 17, 17, 17, 22, 22, 22,
                    22, 22, 22, 22, 27, 27, 27, 27, 27, 27, 27],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False],
       fill_value='?',
            dtype=object),
 'param_n_estimators': masked_array(data=[3, 53, 103, 153, 203, 253, 303, 3, 53, 103, 153, 20
3,
                    253, 303, 3, 53, 103, 153, 203, 253, 303, 3, 53, 103,
                    153, 203, 253, 303, 3, 53, 103, 153, 203, 253, 303, 3,
                    53, 103, 153, 203, 253, 303],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False],
       fill_value='?',
            dtype=object),
 'params': [{'max_depth': 2, 'n_estimators': 3},
  {'max_depth': 2, 'n_estimators': 53},
  {'max_depth': 2, 'n_estimators': 103},
  {'max_depth': 2, 'n_estimators': 153},
  {'max_depth': 2, 'n_estimators': 203},
```

```
 {'max_depth': 2, 'n_estimators': 253},
 {'max_depth': 2, 'n_estimators': 303},
 {'max_depth': 7, 'n_estimators': 3},
 {'max_depth': 7, 'n_estimators': 53},
 {'max_depth': 7, 'n_estimators': 103},
 {'max_depth': 7, 'n_estimators': 153},
 {'max_depth': 7, 'n_estimators': 203},
 {'max_depth': 7, 'n_estimators': 253},
 {'max_depth': 7, 'n_estimators': 303},
 {'max_depth': 12, 'n_estimators': 3},
 {'max_depth': 12, 'n_estimators': 53},
 {'max_depth': 12, 'n_estimators': 103},
 {'max_depth': 12, 'n_estimators': 153},
 {'max_depth': 12, 'n_estimators': 203},
 {'max_depth': 12, 'n_estimators': 253},
 {'max_depth': 12, 'n_estimators': 303},
 {'max_depth': 17, 'n_estimators': 3},
 {'max_depth': 17, 'n_estimators': 53},
 {'max_depth': 17, 'n_estimators': 103},
 {'max_depth': 17, 'n_estimators': 153},
 {'max_depth': 17, 'n_estimators': 203},
 {'max_depth': 17, 'n_estimators': 253},
 {'max_depth': 17, 'n_estimators': 303},
 {'max_depth': 22, 'n_estimators': 3},
 {'max_depth': 22, 'n_estimators': 53},
 {'max_depth': 22, 'n_estimators': 103},
 {'max_depth': 22, 'n_estimators': 153},
 {'max_depth': 22, 'n_estimators': 203},
 {'max_depth': 22, 'n_estimators': 253},
 {'max_depth': 22, 'n_estimators': 303},
 {'max_depth': 27, 'n_estimators': 3},
 {'max_depth': 27, 'n_estimators': 53},
 {'max_depth': 27, 'n_estimators': 103},
 {'max_depth': 27, 'n_estimators': 153},
 {'max_depth': 27, 'n_estimators': 203},
 {'max_depth': 27, 'n_estimators': 253},
 {'max_depth': 27, 'n_estimators': 303}],
'split0_test_score': array([0.23318386, 0.23318386, 0.23168909, 0.23168909, 0.23168909,
       0.23318386, 0.23467862, 0.22869955, 0.27055306, 0.24962631,
       0.25560538, 0.25710015, 0.26606876, 0.25411061, 0.20179372,
       0.26307922, 0.23467862, 0.23467862, 0.23168909, 0.24215247,
       0.23617339, 0.2077728 , 0.24364723, 0.23766816, 0.23916293,
       0.245142  , 0.245142  , 0.24813154, 0.21973094, 0.25710015,
       0.23019432, 0.23916293, 0.23617339, 0.23467862, 0.24215247,
       0.21674141, 0.24364723, 0.23467862, 0.23766816, 0.24215247,
       0.24215247, 0.23617339]),
'split1_test_score': array([0.25299401, 0.26197605, 0.26047904, 0.26646707, 0.26946108,
       0.26347305, 0.26497006, 0.25149701, 0.25449102, 0.25299401,
       0.25299401, 0.25449102, 0.23952096, 0.25149701, 0.21856287,
       0.22904192, 0.22155689, 0.22155689, 0.2245509 , 0.22904192,
       0.23053892, 0.2005988 , 0.23053892, 0.23353293, 0.21407186,
       0.21257485, 0.21556886, 0.22155689, 0.20808383, 0.2260479 ,
       0.2245509 , 0.22754491, 0.21407186, 0.21706587, 0.22155689,
       0.20359281, 0.23502994, 0.2260479 , 0.23802395, 0.22904192,
       0.22305389, 0.22754491]),
'split2_test_score': array([0.23353293, 0.25598802, 0.25748503, 0.25299401, 0.25299401,
       0.25149701, 0.25299401, 0.25299401, 0.26347305, 0.2739521 ,
       0.2754491 , 0.27694611, 0.27694611, 0.26946108, 0.20209581,
       0.23652695, 0.25      , 0.25      , 0.24850299, 0.25149701,
       0.25598802, 0.23203593, 0.22904192, 0.23652695, 0.23952096,
       0.24251497, 0.25898204, 0.25      , 0.21257485, 0.23353293,
       0.24401198, 0.23652695, 0.23203593, 0.23353293, 0.24101796,
       0.21257485, 0.2245509 , 0.23952096, 0.23802395, 0.23053892,
       0.23502994, 0.24101796]),
'split3_test_score': array([0.25      , 0.25299401, 0.25149701, 0.25149701, 0.25      ,
       0.25149701, 0.25299401, 0.24550898, 0.26347305, 0.26946108,
       0.26796407, 0.2739521 , 0.2754491 , 0.2739521 , 0.21556886,
Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js 497006, 0.25898204, 0.26047904,
       0.25898204, 0.22904192, 0.24850299, 0.24700599, 0.23802395,
```

```
       0.23952096, 0.23502994, 0.23652695, 0.19161677, 0.22904192,
       0.24251497, 0.24101796, 0.23652695, 0.23203593, 0.22904192,
       0.19161677, 0.23952096, 0.23203593, 0.24401198, 0.24101796,
       0.23353293, 0.23652695]),
 'split4_test_score': array([0.25598802, 0.27095808, 0.26047904, 0.25748503, 0.25748503,
       0.25898204, 0.25898204, 0.25898204, 0.24251497, 0.24850299,
       0.25449102, 0.25898204, 0.25748503, 0.26047904, 0.21706587,
       0.25      , 0.24700599, 0.24850299, 0.24101796, 0.23802395,
       0.24550898, 0.20658683, 0.22754491, 0.22904192, 0.24401198,
       0.24101796, 0.23802395, 0.23802395, 0.21257485, 0.25299401,
       0.24700599, 0.24700599, 0.24700599, 0.23802395, 0.23353293,
       0.21856287, 0.24101796, 0.24850299, 0.24251497, 0.24850299,
       0.23952096, 0.24401198]),
 'mean_test_score': array([0.24513977, 0.25502   , 0.25232584, 0.25202644, 0.25232584,
       0.25172659, 0.25292375, 0.24753632, 0.25890103, 0.2589073 ,
       0.26130072, 0.26429428, 0.26309399, 0.26189997, 0.21101743,
       0.24692722, 0.24244471, 0.24394171, 0.2409486 , 0.24423888,
       0.24543827, 0.21520725, 0.2358552 , 0.23675519, 0.23495833,
       0.23615415, 0.23854936, 0.23884786, 0.20891625, 0.23974338,
       0.23765563, 0.23825175, 0.23316282, 0.23106746, 0.23346043,
       0.20861774, 0.2367534 , 0.23615728, 0.2400486 , 0.23825085,
       0.23465804, 0.23705504]),
 'std_test_score': array([0.00980467, 0.01252412, 0.01082709, 0.01142937, 0.01226348,
       0.01033982, 0.01014604, 0.01034806, 0.00964912, 0.01064971,
       0.00885875, 0.00926738, 0.01371516, 0.00864105, 0.00746867,
       0.01249382, 0.01302293, 0.01474167, 0.01214003, 0.010866  ,
       0.01097784, 0.01278743, 0.0085373 , 0.0059303 , 0.01064056,
       0.01193497, 0.01415431, 0.0101548 , 0.00942012, 0.01278666,
       0.0087046 , 0.00636758, 0.01074941, 0.00727316, 0.00766669,
       0.00994828, 0.00671181, 0.00755119, 0.00267044, 0.00737922,
       0.00657077, 0.00557857]),
 'rank_test_score': array([16,  7,  9, 11,  9, 12,  8, 13,  6,  5,  4,  1,  2,  3, 40, 14, 1
9,
       18, 20, 17, 15, 39, 33, 29, 34, 32, 24, 23, 41, 22, 27, 25, 37, 38,
       36, 42, 30, 31, 21, 26, 35, 28])}
```

The best hyperparameters for the random forest classifier are:

In [ ]:
```
abalone_lda_rforest_cv.best_params_
```

Out[ ]:
```
{'max_depth': 7, 'n_estimators': 203}
```

The score achieved by the best hyperparameters is:

In [ ]:
```
abalone_lda_rforest_cv.best_score_
```

Out[ ]:
```
0.2642942813923722
```

Now we will initialize a new random forest and train it with the best hyperparameters that were discovered by GridSearchCV

In [ ]:
```
# initialize the decision tree classifier with the best hyperparemeter
abalone_lda_rforest = RandomForestClassifier(random_state=27, max_depth= 7, n_estimators =20

# train the classifier using the training dataset
abalone_lda_rforest.fit(abalone_LDAminmax_train , abalone_rings_LDAminmax_train)

# view the accuracy of the classifier using the testing dataset
abalone_lda_rforest_test_accuracy = abalone_lda_rforest.score(abalone_LDAminmax_test , abalo
abalone_lda_rforest_test_accuracy
```

Out[ ]:
```
0.034688995215311005
```


he test data is shown below:

In [ ]:
```
abalone_lda_rforest_cv.score(abalone_LDAminmax_test , abalone_rings_LDAminmax_test)
```

Out[ ]:
```
0.034688995215311005
```

4.1.2 Wine Dataset

For the maximum depth of the tree in the random forest classifier, we will use the same that was used in the decision tree algorithm, which for the case of abalone is 29

Normalized Data

In [ ]:
```
# create an array for the maximum depths for the classifier parameters
wine_rf_tree_depth = list(range(2,29,5))

# create an array for the number of estimators for the classifier parameters
wine_rf_no_estimators = list(range(3,304,50))

# generate a dictionary for the parameters of random forest classifier
parameters = {"n_estimators": wine_rf_no_estimators ,"max_depth":wine_rf_tree_depth}

# initialize the GridSearchCV function
wine_raw_rforest_cv = GridSearchCV(RandomForestClassifier(random_state=27), parameters, cv =

# fit the training data using the GridSearchCV function
wine_raw_rforest_cv.fit(wine_raw_train, wine_quality_raw_train)

# display the results of the tuning of the hyperprameters
wine_raw_rforest_cv.cv_results_
```

```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\model_sel
ection\_split.py:676: UserWarning: The least populated class in y has only 4 members, whic
h is less than n_splits=5.
  warnings.warn(
```

Out[ ]:
```
{'mean_fit_time': array([0.0269968 , 0.36440148, 0.53979983, 0.8727993 , 1.04219689,
       1.2815969 , 1.95507488, 0.02679663, 0.63739748, 0.9133996 ,
       1.3613946 , 1.74179649, 2.3437993 , 5.21472845, 0.08600268,
       1.20556479, 1.70350971, 2.93145046, 3.75150547, 3.16403623,
       4.09785953, 0.04059701, 0.73152261, 1.42319961, 2.0618145 ,
       3.2621139 , 3.7813024 , 4.25960264, 0.0445962 , 0.70379715,
       1.37039862, 2.08679957, 2.81000051, 3.63203974, 4.2397994 ,
       0.05259709, 0.73739629, 1.42519302, 2.14519687, 2.79499717,
       3.57080026, 4.20831242]),
 'std_fit_time': array([7.01610879e-03, 3.54831113e-02, 6.09958496e-02, 2.02213524e-01,
       9.31178110e-02, 5.26354305e-02, 4.07759278e-01, 4.00982156e-04,
       1.56175185e-01, 2.23330995e-02, 4.71362233e-02, 6.27556047e-02,
       1.17624808e-01, 1.01416218e+00, 3.82661696e-02, 3.50318898e-01,
       4.28410846e-01, 7.28126852e-01, 6.99654145e-01, 2.19343482e-01,
       5.82148727e-01, 4.90902439e-04, 3.27641481e-02, 9.46239299e-02,
       1.22973686e-01, 5.04850740e-01, 3.46365246e-01, 2.70117680e-01,
       2.33211473e-03, 5.05393073e-02, 3.13973864e-02, 8.34040357e-02,
       1.29703394e-01, 3.03411476e-01, 1.68545892e-01, 6.21637609e-03,
       2.57572059e-02, 4.86530111e-02, 4.54846185e-02, 6.27143222e-02,
       5.35385284e-02, 1.39822593e-01]),
 'mean_score_time': array([0.00740061, 0.03699813, 0.05960069, 0.0759995 , 0.09340425,
       0.1263998 , 0.15860453, 0.00480213, 0.03260107, 0.05720148,
       0.06800475, 0.1114037 , 0.13740106, 0.2783144 , 0.00999818,
       0.06268044, 0.07987995, 0.13003602, 0.14319844, 0.1792171 ,
       0.17560253, 0.00520372, 0.03180017, 0.07000093, 0.09120054,
       0.14759798, 0.19740248, 0.1799983 , 0.00600319, 0.03000317,
       0.06360221, 0.08979917, 0.11540217, 0.16080265, 0.1838007 ,
       0.00600204, 0.0332046 , 0.06260686, 0.10260344, 0.12000251,
       0.15820017, 0.18123364]),
 'std_score_time': array([0.00224465, 0.01046932, 0.01537148, 0.02389234, 0.03218962,
       0.02792522, 0.04049071, 0.00074677, 0.0105775 , 0.01259223,
       0.00509843, 0.02284908, 0.03218819, 0.12026307, 0.002965  ,
       0.02798326, 0.01815383, 0.03353552, 0.01628747, 0.05109716,
       0.01358454, 0.00040054, 0.00263617, 0.01951455, 0.00495335,
       0.01689639, 0.06236091, 0.00953253, 0.00109441, 0.000634  ,
       0.00873101, 0.0063338 , 0.00755153, 0.0423828 , 0.01090844,
       0.00154902, 0.00402053, 0.00776206, 0.01430509, 0.01146875,
       0.00673526, 0.01743261]),
 'param_max_depth': masked_array(data=[2, 2, 2, 2, 2, 2, 2, 7, 7, 7, 7, 7, 7, 7, 12, 12, 1
2,
                    12, 12, 12, 12, 17, 17, 17, 17, 17, 17, 17, 22, 22, 22,
                    22, 22, 22, 22, 27, 27, 27, 27, 27, 27, 27],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False],
        fill_value='?',
             dtype=object),
 'param_n_estimators': masked_array(data=[3, 53, 103, 153, 203, 253, 303, 3, 53, 103, 153,
203,
                    253, 303, 3, 53, 103, 153, 203, 253, 303, 3, 53, 103,
                    153, 203, 253, 303, 3, 53, 103, 153, 203, 253, 303, 3,
                    53, 103, 153, 203, 253, 303],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False],
        fill_value='?',
             dtype=object),
 'params': [{'max_depth': 2, 'n_estimators': 3},
  {'max_depth': 2, 'n_estimators': 53},
  {'max_depth': 2, 'n_estimators': 103},
  {'max_depth': 2, 'n_estimators': 153},
  {'max_depth': 2, 'n_estimators': 203},
  {'max_depth': 2, 'n_estimators': 253},
```

```
 {'max_depth': 2, 'n_estimators': 303},
 {'max_depth': 7, 'n_estimators': 3},
 {'max_depth': 7, 'n_estimators': 53},
 {'max_depth': 7, 'n_estimators': 103},
 {'max_depth': 7, 'n_estimators': 153},
 {'max_depth': 7, 'n_estimators': 203},
 {'max_depth': 7, 'n_estimators': 253},
 {'max_depth': 7, 'n_estimators': 303},
 {'max_depth': 12, 'n_estimators': 3},
 {'max_depth': 12, 'n_estimators': 53},
 {'max_depth': 12, 'n_estimators': 103},
 {'max_depth': 12, 'n_estimators': 153},
 {'max_depth': 12, 'n_estimators': 203},
 {'max_depth': 12, 'n_estimators': 253},
 {'max_depth': 12, 'n_estimators': 303},
 {'max_depth': 17, 'n_estimators': 3},
 {'max_depth': 17, 'n_estimators': 53},
 {'max_depth': 17, 'n_estimators': 103},
 {'max_depth': 17, 'n_estimators': 153},
 {'max_depth': 17, 'n_estimators': 203},
 {'max_depth': 17, 'n_estimators': 253},
 {'max_depth': 17, 'n_estimators': 303},
 {'max_depth': 22, 'n_estimators': 3},
 {'max_depth': 22, 'n_estimators': 53},
 {'max_depth': 22, 'n_estimators': 103},
 {'max_depth': 22, 'n_estimators': 153},
 {'max_depth': 22, 'n_estimators': 203},
 {'max_depth': 22, 'n_estimators': 253},
 {'max_depth': 22, 'n_estimators': 303},
 {'max_depth': 27, 'n_estimators': 3},
 {'max_depth': 27, 'n_estimators': 53},
 {'max_depth': 27, 'n_estimators': 103},
 {'max_depth': 27, 'n_estimators': 153},
 {'max_depth': 27, 'n_estimators': 203},
 {'max_depth': 27, 'n_estimators': 253},
 {'max_depth': 27, 'n_estimators': 303}],
'split0_test_score': array([0.48557692, 0.52980769, 0.53653846, 0.53365385, 0.5375    ,
       0.53557692, 0.53653846, 0.55576923, 0.6       , 0.60288462,
       0.59807692, 0.59711538, 0.59423077, 0.59326923, 0.56538462,
       0.65384615, 0.66923077, 0.6625    , 0.66538462, 0.66826923,
       0.66826923, 0.55769231, 0.67307692, 0.67115385, 0.67692308,
       0.67596154, 0.68269231, 0.67884615, 0.57692308, 0.65096154,
       0.67403846, 0.66730769, 0.66730769, 0.67307692, 0.67596154,
       0.56730769, 0.66346154, 0.675     , 0.66442308, 0.67211538,
       0.67403846, 0.67115385]),
'split1_test_score': array([0.52307692, 0.55288462, 0.54615385, 0.5375    , 0.5375    ,
       0.53461538, 0.53461538, 0.57307692, 0.60769231, 0.59711538,
       0.60288462, 0.61153846, 0.60384615, 0.60288462, 0.57788462,
       0.65769231, 0.6625    , 0.66442308, 0.6625    , 0.66153846,
       0.66538462, 0.58365385, 0.67692308, 0.68269231, 0.6875    ,
       0.68846154, 0.68461538, 0.67692308, 0.57115385, 0.68173077,
       0.68653846, 0.68365385, 0.68653846, 0.68557692, 0.68557692,
       0.56826923, 0.67307692, 0.6875    , 0.68557692, 0.68269231,
       0.68365385, 0.6875    ]),
'split2_test_score': array([0.49470645, 0.51973051, 0.51684312, 0.51684312, 0.5226179 ,
       0.52743022, 0.52646776, 0.55052936, 0.56977863, 0.56689124,
       0.57651588, 0.57170356, 0.57459095, 0.57555342, 0.56592878,
       0.66698749, 0.6612127 , 0.65928778, 0.65736285, 0.65928778,
       0.6612127 , 0.55822907, 0.6641001 , 0.67564966, 0.6746872 ,
       0.6746872 , 0.67372474, 0.6746872 , 0.58517806, 0.65640038,
       0.66602502, 0.66794995, 0.6746872 , 0.67949952, 0.68046198,
       0.56689124, 0.67179981, 0.67853705, 0.6746872 , 0.68238691,
       0.67949952, 0.68238691]),
'split3_test_score': array([0.48508181, 0.51876805, 0.51588065, 0.5226179 , 0.52069297,
       0.52454283, 0.5226179 , 0.55149182, 0.55630414, 0.56881617,
       0.56400385, 0.56977863, 0.57266603, 0.56977863, 0.5572666 ,
       0.63426372, 0.64292589, 0.64388835, 0.6506256 , 0.64870067,
       966314, 0.65255053, 0.65158807,
       0.6506256 , 0.64581328, 0.64773821, 0.54475457, 0.64388835,
```

```
       0.65640038, 0.65447546, 0.65832531, 0.65736285, 0.65351299,
       0.55052936, 0.64581328, 0.65158807, 0.64677575, 0.65255053,
       0.65832531, 0.65543792]),
 'split4_test_score': array([0.4773821 , 0.5360924 , 0.5360924 , 0.53994225, 0.53994225,
       0.54090472, 0.54186718, 0.55149182, 0.57555342, 0.58517806,
       0.57940327, 0.57747834, 0.57747834, 0.57651588, 0.56496631,
       0.64100096, 0.64870067, 0.64581328, 0.64388835, 0.64292589,
       0.64388835, 0.5572666 , 0.65543792, 0.6612127 , 0.6612127 ,
       0.65255053, 0.66698749, 0.66987488, 0.56207892, 0.64677575,
       0.64870067, 0.65351299, 0.66987488, 0.66794995, 0.66217517,
       0.55630414, 0.64966314, 0.65158807, 0.6506256 , 0.65640038,
       0.6612127 , 0.66698749]),
 'mean_test_score': array([0.49316484, 0.53145665, 0.5303017 , 0.53011142, 0.53165063,
       0.53261401, 0.53242134, 0.55647183, 0.5818657 , 0.58417709,
       0.58417691, 0.58552288, 0.58456245, 0.58360036, 0.56628618,
       0.65075813, 0.65691401, 0.6551825 , 0.65595228, 0.65614441,
       0.65729862, 0.56166673, 0.66384023, 0.66865181, 0.67038221,
       0.66845728, 0.67076664, 0.6696139 , 0.56801769, 0.65595136,
       0.6663406 , 0.66537999, 0.67134671, 0.67269323, 0.67153772,
       0.56186033, 0.66076294, 0.66884264, 0.66441771, 0.6692291 ,
       0.67134597, 0.67269323]),
 'std_test_score': array([0.01593183, 0.01250496, 0.01194002, 0.00889928, 0.00823215,
       0.00589128, 0.00696478, 0.00849867, 0.0191583 , 0.01451297,
       0.01436398, 0.01621583, 0.01230023, 0.01242057, 0.00661149,
       0.01173268, 0.00963858, 0.00861542, 0.00784529, 0.00912279,
       0.009719  , 0.01125999, 0.01027749, 0.01064854, 0.0125828 ,
       0.01460335, 0.01399794, 0.01134029, 0.01385685, 0.01356013,
       0.01324602, 0.01099081, 0.00927346, 0.00969517, 0.0119068 ,
       0.00714591, 0.01120203, 0.01466583, 0.01452333, 0.01269226,
       0.00997431, 0.01136555]),
 'rank_test_score': array([42, 39, 40, 41, 38, 36, 37, 35, 30, 27, 28, 25, 26, 29, 32, 24,
19,
       23, 21, 20, 18, 34, 16, 11,  7, 12,  6,  8, 31, 22, 13, 14,  4,  2,
        3, 33, 17, 10, 15,  9,  5,  1])}
```

The best hyperparameters for the random forest classifier are:

In [ ]:
```
wine_raw_rforest_cv.best_params_
```

Out[ ]:
```
{'max_depth': 27, 'n_estimators': 303}
```

The score achieved by the best hyperparameters is:

In [ ]:
```
wine_raw_rforest_cv.best_score_
```

Out[ ]:
```
0.6726932331383727
```

Now we will initialize a new random forest and train it with the best hyperparameters that were discovered by GridSearchCV

In [ ]:
```
# initialize the decision tree classifier with the best hyperparemeter
wine_raw_rforest = RandomForestClassifier(random_state=27, max_depth= 27, n_estimators =303)

# train the classifier using the training dataset
wine_raw_rforest.fit(wine_raw_train, wine_quality_raw_train)

# view the accuracy of the classifier using the testing dataset
wine_raw_rforest_test_accuracy = wine_raw_rforest.score(wine_raw_test, wine_quality_raw_test
wine_raw_rforest_test_accuracy
```

Out[ ]:
```
0.693076923076923
```

PCA Data

In [ ]:

```
# create an array for the maximum depths for the classifier parameters
wine_rf_tree_depth = list(range(2,29,5))

# create an array for the number of estimators for the classifier parameters
wine_rf_no_estimators = list(range(3,304,50))

# generate a dictionary for the parameters of random forest classifier
parameters = {"n_estimators": wine_rf_no_estimators ,"max_depth":wine_rf_tree_depth}

# initialize the GridSearchCV function
wine_pca_rforest_cv = GridSearchCV(RandomForestClassifier(random_state=27), parameters, cv =

# fit the training data using the GridSearchCV function
wine_pca_rforest_cv.fit(wine_PCAminmax_train , wine_quality_PCAminmax_train)

# display the results of the tuning of the hyperprameters
wine_pca_rforest_cv.cv_results_
```

```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\model_select
ion\_split.py:676: UserWarning: The least populated class in y has only 4 members, which is l
ess than n_splits=5.
  warnings.warn(
```

Out[ ]:
```
{'mean_fit_time': array([0.0293386 , 0.33899903, 0.85815787, 1.19396515, 2.07988586,
       1.70418487, 1.447997  , 0.03099771, 0.49419742, 0.96979809,
       1.3900013 , 1.83339744, 2.35402308, 2.79080224, 0.04039707,
       0.6595973 , 1.2640007 , 1.9623807 , 2.53460021, 3.45101647,
       4.30859838, 0.05139632, 0.78760109, 1.64979658, 2.24699645,
       3.01580181, 4.26739635, 5.48439426, 0.05020041, 0.83520155,
       1.65920296, 2.45740004, 3.15911326, 5.12519898, 5.74455075,
       0.05400057, 1.06720161, 1.55759401, 2.81300211, 3.09140277,
       3.83979974, 4.44140396]),
 'std_fit_time': array([3.87339488e-03, 5.34231810e-02, 9.79205041e-02, 3.97553895e-01,
       4.64708905e-01, 1.19673936e-01, 1.84714053e-02, 4.29524156e-06,
       2.42455311e-02, 6.39912929e-02, 1.57876900e-02, 4.55686186e-02,
       7.37385935e-02, 7.36562393e-02, 4.90353182e-04, 8.38240824e-03,
       1.53746094e-02, 7.78739030e-02, 4.15233229e-02, 5.37343117e-01,
       3.68112641e-01, 1.04956473e-02, 5.48168721e-02, 2.31445380e-01,
       3.55499711e-02, 1.53057481e-01, 2.98364427e-01, 6.41035694e-01,
       1.16631713e-03, 1.47965230e-02, 3.17931734e-02, 1.44363852e-01,
       4.25931910e-01, 9.20547760e-01, 1.00236911e+00, 5.50902573e-03,
       2.35002224e-01, 1.35685544e-01, 5.40706215e-01, 1.59241099e-01,
       7.36681715e-02, 1.65150538e-01]),
 'mean_score_time': array([0.00725884, 0.0256012 , 0.05430512, 0.1163754 , 0.15653915,
       0.12651696, 0.10259886, 0.00420303, 0.03120031, 0.04380145,
       0.06459866, 0.08380442, 0.10860076, 0.14879947, 0.00480418,
       0.02840352, 0.05200076, 0.08780141, 0.10339966, 0.165802  ,
       0.17059898, 0.00580335, 0.03400059, 0.06060357, 0.09039931,
       0.11800294, 0.17180228, 0.2494031 , 0.00540209, 0.03459616,
       0.06580057, 0.10600033, 0.11040092, 0.15820322, 0.23300133,
       0.00679965, 0.03380232, 0.06100521, 0.1068017 , 0.11459937,
       0.15340409, 0.1746007 ]),
 'std_score_time': array([0.00111598, 0.00820758, 0.01185197, 0.07692464, 0.02849284,
       0.03516778, 0.00048926, 0.00040057, 0.00901833, 0.00074774,
       0.00272529, 0.00223005, 0.00744694, 0.05461168, 0.00040236,
       0.00049077, 0.0006334 , 0.0176799 , 0.00174612, 0.03695022,
       0.01491954, 0.00116643, 0.00569108, 0.00233281, 0.00788387,
       0.00901207, 0.02538567, 0.04424796, 0.00135511, 0.0004917 ,
       0.00511026, 0.01735131, 0.00734142, 0.01368678, 0.0786315 ,
       0.00172704, 0.00342879, 0.00260772, 0.02609756, 0.00535376,
       0.02845709, 0.01976949]),
 'param_max_depth': masked_array(data=[2, 2, 2, 2, 2, 2, 2, 7, 7, 7, 7, 7, 7, 7, 12, 12, 12,
                    12, 12, 12, 12, 17, 17, 17, 17, 17, 17, 17, 22, 22, 22,
                    22, 22, 22, 22, 27, 27, 27, 27, 27, 27, 27],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False],
        fill_value='?',
             dtype=object),
 'param_n_estimators': masked_array(data=[3, 53, 103, 153, 203, 253, 303, 3, 53, 103, 153, 20
3,
                    253, 303, 3, 53, 103, 153, 203, 253, 303, 3, 53, 103,
                    153, 203, 253, 303, 3, 53, 103, 153, 203, 253, 303, 3,
                    53, 103, 153, 203, 253, 303],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False],
        fill_value='?',
             dtype=object),
 'params': [{'max_depth': 2, 'n_estimators': 3},
  {'max_depth': 2, 'n_estimators': 53},
  {'max_depth': 2, 'n_estimators': 103},
  {'max_depth': 2, 'n_estimators': 153},
  {'max_depth': 2, 'n_estimators': 203},
  {'max_depth': 2, 'n_estimators': 253},
  {'max_depth': 2, 'n_estimators': 303},
```

```
 {'max_depth': 7, 'n_estimators': 3},
 {'max_depth': 7, 'n_estimators': 53},
 {'max_depth': 7, 'n_estimators': 103},
 {'max_depth': 7, 'n_estimators': 153},
 {'max_depth': 7, 'n_estimators': 203},
 {'max_depth': 7, 'n_estimators': 253},
 {'max_depth': 7, 'n_estimators': 303},
 {'max_depth': 12, 'n_estimators': 3},
 {'max_depth': 12, 'n_estimators': 53},
 {'max_depth': 12, 'n_estimators': 103},
 {'max_depth': 12, 'n_estimators': 153},
 {'max_depth': 12, 'n_estimators': 203},
 {'max_depth': 12, 'n_estimators': 253},
 {'max_depth': 12, 'n_estimators': 303},
 {'max_depth': 17, 'n_estimators': 3},
 {'max_depth': 17, 'n_estimators': 53},
 {'max_depth': 17, 'n_estimators': 103},
 {'max_depth': 17, 'n_estimators': 153},
 {'max_depth': 17, 'n_estimators': 203},
 {'max_depth': 17, 'n_estimators': 253},
 {'max_depth': 17, 'n_estimators': 303},
 {'max_depth': 22, 'n_estimators': 3},
 {'max_depth': 22, 'n_estimators': 53},
 {'max_depth': 22, 'n_estimators': 103},
 {'max_depth': 22, 'n_estimators': 153},
 {'max_depth': 22, 'n_estimators': 203},
 {'max_depth': 22, 'n_estimators': 253},
 {'max_depth': 22, 'n_estimators': 303},
 {'max_depth': 27, 'n_estimators': 3},
 {'max_depth': 27, 'n_estimators': 53},
 {'max_depth': 27, 'n_estimators': 103},
 {'max_depth': 27, 'n_estimators': 153},
 {'max_depth': 27, 'n_estimators': 203},
 {'max_depth': 27, 'n_estimators': 253},
 {'max_depth': 27, 'n_estimators': 303}],
'split0_test_score': array([0.45576923, 0.5       , 0.48846154, 0.49423077, 0.49230769,
       0.49230769, 0.49230769, 0.52307692, 0.57115385, 0.57788462,
       0.57788462, 0.57692308, 0.57788462, 0.58173077, 0.55576923,
       0.63076923, 0.64134615, 0.64423077, 0.63846154, 0.64519231,
       0.64230769, 0.54903846, 0.65673077, 0.66153846, 0.66730769,
       0.66538462, 0.66442308, 0.65961538, 0.54230769, 0.65961538,
       0.65      , 0.64903846, 0.64807692, 0.65      , 0.65288462,
       0.56057692, 0.65288462, 0.64519231, 0.65192308, 0.65      ,
       0.65384615, 0.65384615]),
'split1_test_score': array([0.50384615, 0.51346154, 0.51634615, 0.51153846, 0.51153846,
       0.51153846, 0.51346154, 0.54519231, 0.58365385, 0.59711538,
       0.59519231, 0.59423077, 0.59230769, 0.59423077, 0.54903846,
       0.64615385, 0.65673077, 0.6625    , 0.65288462, 0.65673077,
       0.65576923, 0.55769231, 0.64711538, 0.66057692, 0.66826923,
       0.66634615, 0.66634615, 0.66346154, 0.55673077, 0.66153846,
       0.66923077, 0.67307692, 0.67403846, 0.66923077, 0.67019231,
       0.55192308, 0.67019231, 0.66826923, 0.66730769, 0.67307692,
       0.67115385, 0.66442308]),
'split2_test_score': array([0.49374398, 0.51395573, 0.50818094, 0.50240616, 0.50625602,
       0.50529355, 0.50625602, 0.533205  , 0.5813282 , 0.58517806,
       0.58421559, 0.58902791, 0.58421559, 0.58614052, 0.56111646,
       0.64677575, 0.64485082, 0.64196343, 0.64966314, 0.65158807,
       0.6506256 , 0.57844081, 0.65543792, 0.6506256 , 0.65543792,
       0.65543792, 0.66602502, 0.6641001 , 0.55630414, 0.65928778,
       0.66217517, 0.66987488, 0.6612127 , 0.6612127 , 0.66794995,
       0.56881617, 0.65928778, 0.66025024, 0.67661213, 0.67372474,
       0.66987488, 0.66987488]),
'split3_test_score': array([0.49278152, 0.50336862, 0.50818094, 0.50048123, 0.4985563 ,
       0.4985563 , 0.49759384, 0.5360924 , 0.57459095, 0.57844081,
       0.57362849, 0.57747834, 0.57555342, 0.57362849, 0.56207892,
       0.61405197, 0.61886429, 0.63137632, 0.63522618, 0.63426372,
       0.63522618, 0.51780558, 0.626564  , 0.6400385 , 0.64196343,
       0.63426372, 0.53224254, 0.63618864,
       0.63811357, 0.63715111, 0.63907603, 0.64196343, 0.6400385 ,
```

```
       0.55437921, 0.62848893, 0.63137632, 0.63426372, 0.64100096,
       0.63522618, 0.63330125]),
 'split4_test_score': array([0.47545717, 0.51010587, 0.49759384, 0.5014437 , 0.50625602,
       0.50433109, 0.50048123, 0.53801732, 0.56592878, 0.56977863,
       0.57844081, 0.57266603, 0.56977863, 0.57170356, 0.55822907,
       0.60923965, 0.61886429, 0.61982676, 0.62175168, 0.61886429,
       0.62271415, 0.55822907, 0.62560154, 0.63330125, 0.63522618,
       0.63041386, 0.63522618, 0.63618864, 0.52839269, 0.62463908,
       0.63426372, 0.63233879, 0.63041386, 0.62848893, 0.63426372,
       0.5226179 , 0.62463908, 0.62752647, 0.62271415, 0.62848893,
       0.62560154, 0.62560154]),
 'mean_test_score': array([0.48431961, 0.50817835, 0.50375268, 0.50202006, 0.5029829 ,
       0.50240542, 0.50202006, 0.53511679, 0.57533112, 0.5816795 ,
       0.58187236, 0.58206523, 0.57994799, 0.58148682, 0.55724643,
       0.62939809, 0.63613127, 0.63997946, 0.63959743, 0.64132783,
       0.64132857, 0.55224125, 0.64228992, 0.64921615, 0.65364089,
       0.65133172, 0.6540268 , 0.65152588, 0.54319557, 0.64825387,
       0.65075664, 0.65229603, 0.6505636 , 0.65017917, 0.65306582,
       0.55166266, 0.64709854, 0.64652291, 0.65056415, 0.65325831,
       0.65114052, 0.64940938]),
 'std_test_score': array([0.01694329, 0.0055682 , 0.00969087, 0.00555198, 0.00675543,
       0.00651524, 0.00728204, 0.00720409, 0.00650591, 0.00913266,
       0.00746392, 0.00815237, 0.00771959, 0.00826547, 0.0046658 ,
       0.01566196, 0.01499241, 0.01420795, 0.01110618, 0.01350533,
       0.01165901, 0.01973858, 0.01364184, 0.01113918, 0.01326246,
       0.01433553, 0.01421619, 0.01341062, 0.01178914, 0.01503689,
       0.01344703, 0.01660803, 0.01554755, 0.01430174, 0.01440914,
       0.01565089, 0.01769815, 0.01583285, 0.02001008, 0.0178103 ,
       0.01824465, 0.01726394]),
 'rank_test_score': array([42, 36, 37, 40, 38, 39, 40, 35, 30, 27, 26, 25, 29, 28, 31, 24, 2
3,
       21, 22, 20, 19, 32, 18, 14,  2,  7,  1,  6, 34, 15,  9,  5, 11, 12,
        4, 33, 16, 17, 10,  3,  8, 13])}
```

The best hyperparameters for the random forest classifier are:

In [ ]:
```
wine_pca_rforest_cv.best_params_
```

Out[ ]:
```
{'max_depth': 17, 'n_estimators': 253}
```

The score achieved by the best hyperparameters is:

In [ ]:
```
wine_pca_rforest_cv.best_score_
```

Out[ ]:
```
0.6540268009180424
```

Now we will initialize a new random forest and train it with the best hyperparameters that were discovered by GridSearchCV

In [ ]:
```
# initialize the decision tree classifier with the best hyperparemeter
wine_pca_rforest = RandomForestClassifier(random_state=27, max_depth= 17, n_estimators= 253)

# train the classifier using the training dataset
wine_pca_rforest.fit(wine_PCAminmax_train , wine_quality_PCAminmax_train)

# view the accuracy of the classifier using the testing dataset
wine_pca_rforest_test_accuracy = wine_pca_rforest.score(wine_PCAminmax_test , wine_quality_PC
wine_pca_rforest_test_accuracy
```

Out[ ]:
```
0.5
```

The performance of the best classifier on the test data is shown below:

In [ ]:
```
wine_pca_rforest_cv.score(wine_PCAminmax_test , wine_quality_PCAminmax_test)
```

Out[ ]:
```
0.5
```

LDA Data

In [ ]:
```
# create an array for the maximum depths for the classifier parameters
wine_rf_tree_depth = list(range(2,29,5))

# create an array for the number of estimators for the classifier parameters
wine_rf_no_estimators = list(range(3,304,50))

# generate a dictionary for the parameters of random forest classifier
parameters = {"n_estimators": wine_rf_no_estimators ,"max_depth":wine_rf_tree_depth}

# initialize the GridSearchCV function
wine_lda_rforest_cv = GridSearchCV(RandomForestClassifier(random_state=27), parameters, cv =

# fit the training data using the GridSearchCV function
wine_lda_rforest_cv.fit(wine_LDAminmax_train , wine_quality_LDAminmax_train)

# display the results of the tuning of the hyperprameters
wine_lda_rforest_cv.cv_results_
```

```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\model_sel
ection\_split.py:676: UserWarning: The least populated class in y has only 4 members, whic
h is less than n_splits=5.
  warnings.warn(
```

Out[ ]:
```
{'mean_fit_time': array([0.03079834, 0.23559966, 0.55119777, 0.6977983 , 0.8651978 ,
       1.12619972, 1.28719468, 0.02359829, 0.33839946, 0.67139678,
       1.1885983 , 1.60179653, 2.56990414, 2.1139998 , 0.0335988 ,
       0.44520082, 0.86780028, 1.3507967 , 1.84000306, 2.13780241,
       2.61779895, 0.0333971 , 0.56859941, 1.07020245, 1.55939856,
       2.02959924, 2.58819723, 3.13519559, 0.05260262, 0.73100076,
       1.23540158, 1.92443185, 2.5263772 , 4.1754921 , 3.84675875,
       0.03819814, 0.66760149, 1.36620069, 1.71219964, 2.44239836,
       3.23840113, 3.87420149]),
 'std_fit_time': array([2.71158272e-03, 1.75881153e-02, 9.83863465e-02, 6.60308888e-02,
       5.83329588e-02, 1.11598580e-01, 9.19455819e-02, 1.02110074e-03,
       2.57676450e-03, 3.43729160e-02, 3.71334792e-01, 2.38280220e-01,
       5.39820607e-01, 2.47839432e-01, 7.91259904e-03, 9.49420497e-03,
       7.73060271e-03, 1.12023795e-01, 3.43102350e-01, 6.66578321e-02,
       5.91271942e-02, 1.02108683e-03, 2.39854034e-02, 4.86021343e-02,
       8.63972491e-02, 6.62606207e-02, 1.03702140e-01, 2.21741250e-01,
       1.02126934e-02, 1.62400411e-01, 7.70072100e-02, 1.71717311e-01,
       1.67639723e-01, 1.15691805e+00, 2.84638989e-01, 1.59950354e-03,
       7.85682461e-02, 1.18353291e-01, 3.17163087e-02, 2.65585568e-01,
       5.63405516e-01, 5.04458458e-01]),
 'mean_score_time': array([0.00740371, 0.02260156, 0.04180241, 0.0622005 , 0.08640223,
       0.09340158, 0.11540146, 0.00440273, 0.02419806, 0.04420223,
       0.08200331, 0.12400374, 0.15051403, 0.12720037, 0.0044035 ,
       0.0283998 , 0.0550014 , 0.08920283, 0.11199927, 0.1527987 ,
       0.21100125, 0.00460253, 0.03640366, 0.05779996, 0.08760166,
       0.11140108, 0.14000344, 0.17200737, 0.00679579, 0.12020049,
       0.06339993, 0.10800662, 0.13199897, 0.22271171, 0.19656162,
       0.00480094, 0.04960008, 0.06279898, 0.10400071, 0.12340655,
       0.16819978, 0.25739913]),
 'std_score_time': array([0.00079854, 0.00162102, 0.00370901, 0.01051324, 0.01536871,
       0.00431555, 0.02232948, 0.00048893, 0.00075018, 0.0007481 ,
       0.02761567, 0.06165816, 0.01600882, 0.01101883, 0.00049113,
       0.00048728, 0.00252542, 0.00746636, 0.00282853, 0.01358527,
       0.03488932, 0.00048986, 0.00523616, 0.00462244, 0.00360954,
       0.00048726, 0.00316297, 0.00400217, 0.00193511, 0.16501431,
       0.00531481, 0.02074599, 0.03226841, 0.0612529 , 0.02336067,
       0.00039979, 0.03028743, 0.00621445, 0.03157928, 0.01107116,
       0.04116944, 0.13328888]),
 'param_max_depth': masked_array(data=[2, 2, 2, 2, 2, 2, 2, 7, 7, 7, 7, 7, 7, 7, 12, 12, 1
2,
                    12, 12, 12, 12, 17, 17, 17, 17, 17, 17, 17, 22, 22, 22,
                    22, 22, 22, 22, 27, 27, 27, 27, 27, 27, 27],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False],
        fill_value='?',
             dtype=object),
 'param_n_estimators': masked_array(data=[3, 53, 103, 153, 203, 253, 303, 3, 53, 103, 153,
203,
                    253, 303, 3, 53, 103, 153, 203, 253, 303, 3, 53, 103,
                    153, 203, 253, 303, 3, 53, 103, 153, 203, 253, 303, 3,
                    53, 103, 153, 203, 253, 303],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False],
        fill_value='?',
             dtype=object),
 'params': [{'max_depth': 2, 'n_estimators': 3},
  {'max_depth': 2, 'n_estimators': 53},
  {'max_depth': 2, 'n_estimators': 103},
  {'max_depth': 2, 'n_estimators': 153},
  {'max_depth': 2, 'n_estimators': 203},
  {'max_depth': 2, 'n_estimators': 253},
```

```
 {'max_depth': 2, 'n_estimators': 303},
 {'max_depth': 7, 'n_estimators': 3},
 {'max_depth': 7, 'n_estimators': 53},
 {'max_depth': 7, 'n_estimators': 103},
 {'max_depth': 7, 'n_estimators': 153},
 {'max_depth': 7, 'n_estimators': 203},
 {'max_depth': 7, 'n_estimators': 253},
 {'max_depth': 7, 'n_estimators': 303},
 {'max_depth': 12, 'n_estimators': 3},
 {'max_depth': 12, 'n_estimators': 53},
 {'max_depth': 12, 'n_estimators': 103},
 {'max_depth': 12, 'n_estimators': 153},
 {'max_depth': 12, 'n_estimators': 203},
 {'max_depth': 12, 'n_estimators': 253},
 {'max_depth': 12, 'n_estimators': 303},
 {'max_depth': 17, 'n_estimators': 3},
 {'max_depth': 17, 'n_estimators': 53},
 {'max_depth': 17, 'n_estimators': 103},
 {'max_depth': 17, 'n_estimators': 153},
 {'max_depth': 17, 'n_estimators': 203},
 {'max_depth': 17, 'n_estimators': 253},
 {'max_depth': 17, 'n_estimators': 303},
 {'max_depth': 22, 'n_estimators': 3},
 {'max_depth': 22, 'n_estimators': 53},
 {'max_depth': 22, 'n_estimators': 103},
 {'max_depth': 22, 'n_estimators': 153},
 {'max_depth': 22, 'n_estimators': 203},
 {'max_depth': 22, 'n_estimators': 253},
 {'max_depth': 22, 'n_estimators': 303},
 {'max_depth': 27, 'n_estimators': 3},
 {'max_depth': 27, 'n_estimators': 53},
 {'max_depth': 27, 'n_estimators': 103},
 {'max_depth': 27, 'n_estimators': 153},
 {'max_depth': 27, 'n_estimators': 203},
 {'max_depth': 27, 'n_estimators': 253},
 {'max_depth': 27, 'n_estimators': 303}],
'split0_test_score': array([0.49230769, 0.54423077, 0.54230769, 0.54230769, 0.54230769,
       0.54230769, 0.54134615, 0.55673077, 0.56730769, 0.57211538,
       0.57403846, 0.57019231, 0.57211538, 0.57211538, 0.54519231,
       0.60288462, 0.6       , 0.60576923, 0.60384615, 0.6       ,
       0.60096154, 0.54326923, 0.63076923, 0.63076923, 0.63653846,
       0.63846154, 0.63942308, 0.64423077, 0.53557692, 0.62211538,
       0.63653846, 0.63365385, 0.62403846, 0.62692308, 0.63173077,
       0.54326923, 0.61634615, 0.62019231, 0.625     , 0.62307692,
       0.625     , 0.62884615]),
'split1_test_score': array([0.45192308, 0.53173077, 0.53076923, 0.53173077, 0.53269231,
       0.53173077, 0.53173077, 0.54038462, 0.57115385, 0.56538462,
       0.5625    , 0.56346154, 0.57019231, 0.56923077, 0.52980769,
       0.59615385, 0.6       , 0.59711538, 0.60096154, 0.60288462,
       0.60576923, 0.56826923, 0.63076923, 0.63557692, 0.63173077,
       0.63076923, 0.63076923, 0.63557692, 0.53365385, 0.62980769,
       0.63365385, 0.63557692, 0.63269231, 0.62596154, 0.62307692,
       0.54615385, 0.625     , 0.62307692, 0.62307692, 0.62115385,
       0.61923077, 0.61730769]),
'split2_test_score': array([0.43792108, 0.53801732, 0.53801732, 0.53801732, 0.53994225,
       0.53897979, 0.53994225, 0.51973051, 0.56785371, 0.5707411 ,
       0.5707411 , 0.56977863, 0.56689124, 0.56400385, 0.54379211,
       0.6053898 , 0.60250241, 0.60346487, 0.60250241, 0.60442733,
       0.60635226, 0.56111646, 0.63137632, 0.62560154, 0.63233879,
       0.63907603, 0.64100096, 0.63715111, 0.52839269, 0.6294514 ,
       0.63618864, 0.63811357, 0.63330125, 0.63041386, 0.63041386,
       0.55052936, 0.63233879, 0.63137632, 0.63330125, 0.63137632,
       0.63618864, 0.6294514 ]),
'split3_test_score': array([0.48893167, 0.533205  , 0.533205  , 0.53224254, 0.53224254,
       0.53224254, 0.53224254, 0.55822907, 0.56496631, 0.55919153,
       0.56304139, 0.56400385, 0.56400385, 0.56496631, 0.55245428,
       0.59769009, 0.5919153 , 0.59095284, 0.59384023, 0.59095284,
                                     848893, 0.626564  , 0.626564  ,
       0.63137632, 0.63233879, 0.63233879, 0.53705486, 0.62175168,
```

```
        0.61982676, 0.6294514 , 0.6294514 , 0.63041386, 0.62752647,
        0.53801732, 0.62848893, 0.63426372, 0.63426372, 0.6294514 ,
        0.62271415, 0.62175168]),
 'split4_test_score': array([0.466795  , 0.53705486, 0.53705486, 0.53705486, 0.53705486,
        0.5360924 , 0.53705486, 0.54475457, 0.56785371, 0.57170356,
        0.57170356, 0.56977863, 0.56977863, 0.56881617, 0.54956689,
        0.59095284, 0.59961501, 0.60153994, 0.59961501, 0.60250241,
        0.60346487, 0.56015399, 0.61982676, 0.61982676, 0.61886429,
        0.62367661, 0.62175168, 0.62367661, 0.55437921, 0.6159769 ,
        0.61982676, 0.61982676, 0.6159769 , 0.61790183, 0.61790183,
        0.53801732, 0.61116458, 0.60635226, 0.62175168, 0.61116458,
        0.61501444, 0.60827719]),
 'mean_test_score': array([0.4675757 , 0.53684775, 0.53627082, 0.53627064, 0.53684793,
        0.53627064, 0.53646332, 0.54396591, 0.56782705, 0.56782724,
        0.5684049 , 0.56744299, 0.56859628, 0.5678265 , 0.54416266,
        0.59861424, 0.59880654, 0.59976845, 0.60015307, 0.60015344,
        0.60246261, 0.5580151 , 0.62824609, 0.62766769, 0.62920726,
        0.63267195, 0.63305675, 0.63459484, 0.53781151, 0.62382061,
        0.62920689, 0.6313245 , 0.62709206, 0.62632283, 0.62612997,
        0.54319742, 0.62266769, 0.62305231, 0.62747871, 0.62324461,
        0.6236296 , 0.62112682]),
 'std_test_score': array([0.02094163, 0.00436744, 0.00399438, 0.00392296, 0.00394729,
        0.0040163 , 0.0039117 , 0.01390814, 0.00197598, 0.00494867,
        0.00472676, 0.00303805, 0.00283996, 0.00297117, 0.00781306,
        0.00509409, 0.00359586, 0.00524528, 0.00346275, 0.00481493,
        0.00385167, 0.00821391, 0.00432377, 0.00527528, 0.0060644 ,
        0.00567004, 0.00688757, 0.00670295, 0.00878698, 0.0052204 ,
        0.00772329, 0.00640792, 0.0064553 , 0.00457989, 0.00507335,
        0.00482016, 0.00781204, 0.00981893, 0.00525846, 0.00714108,
        0.00712764, 0.00786099]),
 'rank_test_score': array([42, 37, 39, 40, 36, 40, 38, 33, 28, 27, 26, 30, 25, 29, 32, 24,
23,
        22, 21, 20, 19, 31,  7,  8,  5,  3,  2,  1, 35, 13,  6,  4, 10, 11,
        12, 34, 17, 16,  9, 15, 14, 18])}
```

The best hyperparameters for the random forest classifier are:

In [ ]:
```
wine_lda_rforest_cv.best_params_
```

Out[ ]:
```
{'max_depth': 17, 'n_estimators': 303}
```

The score achieved by the best hyperparameters is:

In [ ]:
```
wine_lda_rforest_cv.best_score_
```

Out[ ]:
```
0.6345948397127417
```

Now we will initialize a new random forest and train it with the best hyperparameters that were discovered by GridSearchCV

In [ ]:
```
# initialize the decision tree classifier with the best hyperparemeter
wine_lda_rforest = RandomForestClassifier(random_state=27, max_depth= 17, n_estimators = 303

# train the classifier using the training dataset
wine_lda_rforest.fit(wine_LDAminmax_train , wine_quality_LDAminmax_train)

# view the accuracy of the classifier using the testing dataset
wine_lda_rforest_test_accuracy = wine_lda_rforest.score(wine_LDAminmax_test , wine_quality_LI
wine_lda_rforest_test_accuracy
```

Out[ ]:
```
0.4530769230769231
```

The performance of the best classifier on the test data is shown below:

In [ ]:

```
wine_lda_rforest_cv.score(wine_LDAminmax_test , wine_quality_LDAminmax_test)
```

Out[ ]:
```
0.4530769230769231
```

4.2 Heatmap Plot of Mean Score

4.2.1 Abalone Dataset

4.2.1.1 Raw Dataset

We will generate the heatmap using the mean test score which was generated by GridSearchCV using the training data as shown below

In [ ]:
```
abalone_raw_rforest_meanscore = abalone_raw_rforest_cv.cv_results_["mean_test_score"]
```

the total number of elements in the array is

In [ ]:
```
len(abalone_raw_rforest_meanscore)
```

Out[ ]:
```
42
```

The mean test score of the random forest is arranged with respect to depth first. For our scenerio, 6 maximum depths were used and 7 number of estimators were tried. The total mean test scores produced are 42. We will reshape the array in 7 by 6 to generate the heatmap as shown below:

In [ ]:
```
# save the array using a pandas DataFrame and provide the column names
abalone_raw_heatmap_array = pd.DataFrame(abalone_raw_rforest_meanscore.reshape(7,6), columns

# add the index column which is the Number of Estimators
abalone_raw_heatmap_array["No. of Estimator"]= [3, 53, 103, 153, 203, 253, 303]

# set the Number of Estimators as the index column
abalone_raw_heatmap_array.set_index("No. of Estimator", inplace=True)
```

Now we will preview the data as shown below:

In [ ]:
```
abalone_raw_heatmap_array
```

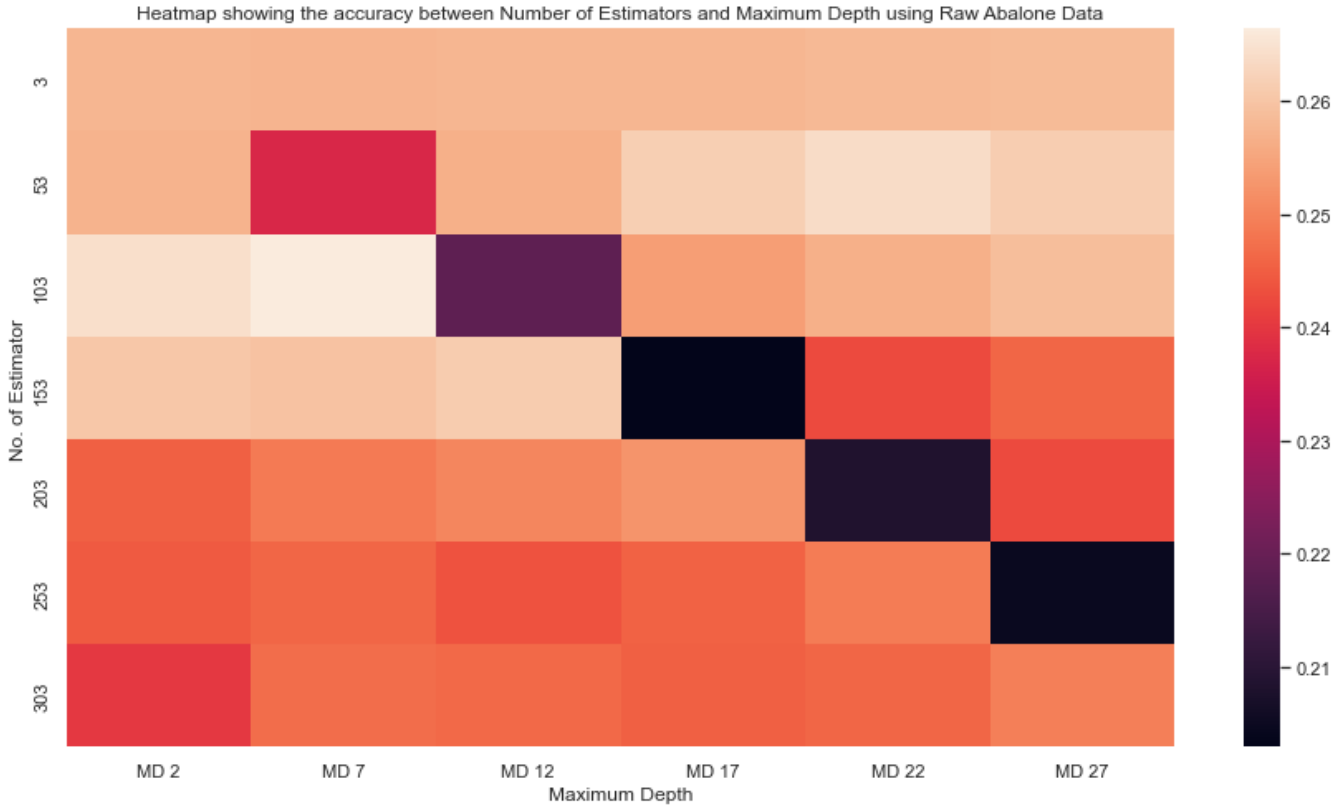
Out[ ]:

No. of Estimator	MD 2	MD 7	MD 12	MD 17	MD 22	MD 27
3	0.257706	0.257402	0.257702	0.257701	0.258002	0.258302
53	0.257104	0.237359	0.256508	0.261595	0.263692	0.261298
103	0.264292	0.266387	0.218494	0.253817	0.256507	0.258904
153	0.260400	0.259502	0.260998	0.202930	0.242435	0.246033
203	0.245137	0.248728	0.250521	0.252316	0.208319	0.242442
253	0.244539	0.246032	0.243340	0.245435	0.249027	0.204425
303	0.240045	0.246931	0.246336	0.245136	0.246034	0.249328

Now we will generate the heatmap using seaborn as shown below:

In [ ]:
```
# initialize the heat map using the heatmap array dataset
sns.heatmap(abalone_raw_heatmap_array)

plt.xlabel("Maximum Depth")
plt.title("Heatmap showing the accuracy between Number of Estimators and Maximum Depth using
```

Out[ ]:
```
Text(0.5, 1.0, 'Heatmap showing the accuracy between Number of Estimators and Maximum Depth u
sing Raw Abalone Data')
```



4.2.1.2 PCA Dataset

We will generate the heatmap using the mean test score which was generated by GridSearchCV using the training data as shown below

In [ ]:
```
abalone_pca_rforest_meanscore = abalone_pca_rforest_cv.cv_results_["mean_test_score"]
```

the total number of elements in the array is

In [ ]:
```
len(abalone_pca_rforest_meanscore)
```

Out[ ]:
```
42
```

The mean test score of the random forest is arranged with respect to depth first. For our scenerio, 6 maximum depths were used and 7 number of estimators were tried. The total mean test scores produced are 42. We will reshape the array in 7 by 6 to generate the heatmap as shown below:

In [ ]:
```
# save the array using a pandas DataFrame and provide the column names
abalone_pca_heatmap_array = pd.DataFrame(abalone_pca_rforest_meanscore.reshape(7,6), columns

# add the index column which is the Number of Estimators
abalone_pca_heatmap_array["No. of Estimator"]= [3, 53, 103, 153, 203, 253, 303]

# set the Number of Estimators as the index column
abalone_pca_heatmap_array.set_index("No. of Estimator", inplace=True)
```

Now we will preview the data as shown below:

In [ ]:
```
abalone_pca_heatmap_array
```

Out[ ]:

No. of Estimator	MD 2	MD 7	MD 12	MD 17	MD 22	MD 27
3	0.235562	0.245743	0.240652	0.240653	0.240653	0.240952
53	0.240952	0.238253	0.252920	0.255314	0.264891	0.264292
103	0.263994	0.264890	0.217302	0.240941	0.242736	0.242140
153	0.243935	0.241842	0.240945	0.197250	0.225674	0.232861
203	0.233460	0.233758	0.229565	0.228368	0.208024	0.228972
253	0.232261	0.234954	0.234655	0.234356	0.233159	0.201736
303	0.236454	0.235253	0.236451	0.232262	0.230765	0.232561

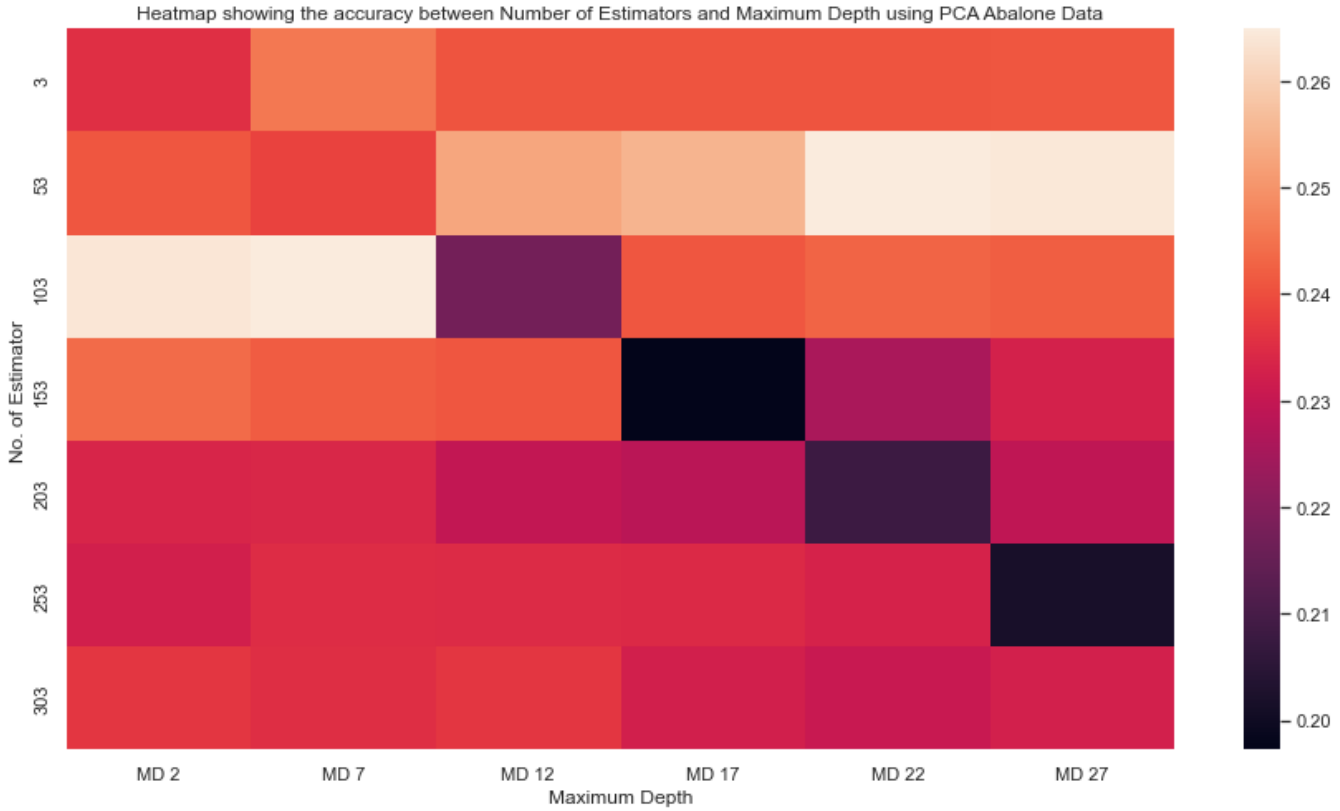
Now we will generate the heatmap using seaborn as shown below:

In [ ]:

```
# initialize the heat map using the heatmap array dataset
sns.heatmap(abalone_pca_heatmap_array)

plt.xlabel("Maximum Depth")

plt.title("Heatmap showing the accuracy between Number of Estimators and Maximum Depth using
```

Out[ ]:

```
Text(0.5, 1.0, 'Heatmap showing the accuracy between Number of Estimators and Maximum Depth u
sing PCA Abalone Data')
```



4.2.1.3 LDA Dataset

We will generate the heatmap using the mean test score which was generated by GridSearchCV using the training data as shown below

In [ ]:

```
abalone_lda_rforest_meanscore = abalone_lda_rforest_cv.cv_results_["mean_test_score"]
```

the total number of elements in the array is

In [ ]:

```
len(abalone_lda_rforest_meanscore)
```

Out[ ]: 42

The mean test score of the random forest is arranged with respect to depth first. For our scenerio, 6 maximum depths were used and 7 number of estimators were tried. The total mean test scores produced are 42. We will reshape the array in 7 by 6 to generate the heatmap as shown below:

In [ ]:
```
# save the array using a pandas DataFrame and provide the column names
abalone_lda_heatmap_array = pd.DataFrame(abalone_lda_rforest_meanscore.reshape(7,6), columns

# add the index column which is the Number of Estimators
abalone_lda_heatmap_array["No. of Estimator"]= [3, 53, 103, 153, 203, 253, 303]

# set the Number of Estimators as the index column
abalone_lda_heatmap_array.set_index("No. of Estimator", inplace=True)
```

Now we will preview the data as shown below:

In [ ]:
```
abalone_lda_heatmap_array
```

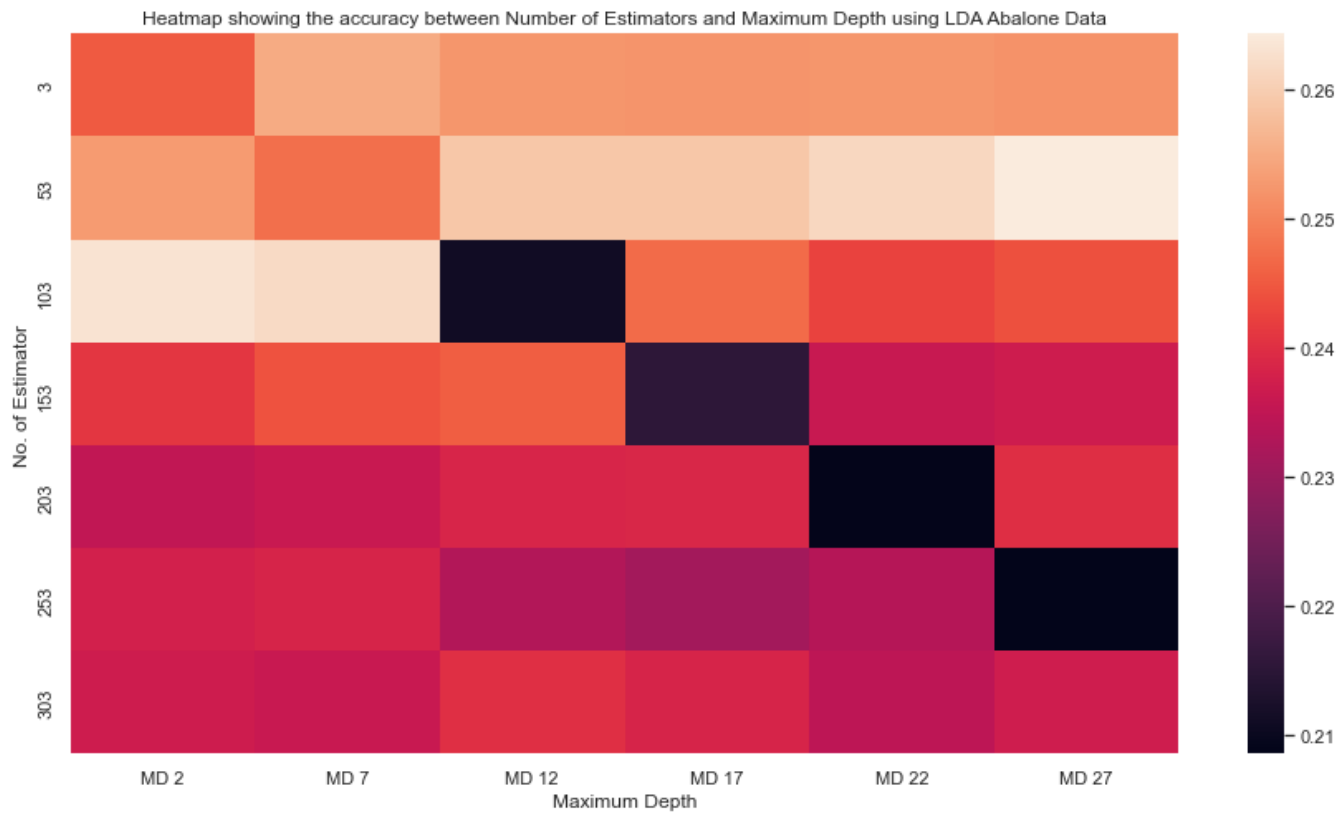
Out[ ]:

No. of Estimator	MD 2	MD 7	MD 12	MD 17	MD 22	MD 27
3	0.245140	0.255020	0.252326	0.252026	0.252326	0.251727
53	0.252924	0.247536	0.258901	0.258907	0.261301	0.264294
103	0.263094	0.261900	0.211017	0.246927	0.242445	0.243942
153	0.240949	0.244239	0.245438	0.215207	0.235855	0.236755
203	0.234958	0.236154	0.238549	0.238848	0.208916	0.239743
253	0.237656	0.238252	0.233163	0.231067	0.233460	0.208618
303	0.236753	0.236157	0.240049	0.238251	0.234658	0.237055

Now we will generate the heatmap using seaborn as shown below:

In [ ]:
```
# initialize the heat map using the heatmap array dataset
sns.heatmap(abalone_lda_heatmap_array)

plt.xlabel("Maximum Depth")

plt.title("Heatmap showing the accuracy between Number of Estimators and Maximum Depth using
```

Out[ ]:
```
Text(0.5, 1.0, 'Heatmap showing the accuracy between Number of Estimators and Maximum Depth u
sing LDA Abalone Data')
```



4.2.2 Wine Dataset

4.2.2.1 Raw Dataset

We will generate the heatmap using the mean test score which was generated by GridSearchCV using the training data as shown below

In [ ]:
```
wine_raw_rforest_cv.cv_results_
```

Out[ ]:
```
{'mean_fit_time': array([0.0269968 , 0.36440148, 0.53979983, 0.8727993 , 1.04219689,
       1.2815969 , 1.95507488, 0.02679663, 0.63739748, 0.9133996 ,
       1.3613946 , 1.74179649, 2.3437993 , 5.21472845, 0.08600268,
       1.20556479, 1.70350971, 2.93145046, 3.75150547, 3.16403623,
       4.09785953, 0.04059701, 0.73152261, 1.42319961, 2.0618145 ,
       3.2621139 , 3.7813024 , 4.25960264, 0.0445962 , 0.70379715,
       1.37039862, 2.08679957, 2.81000051, 3.63203974, 4.2397994 ,
       0.05259709, 0.73739629, 1.42519302, 2.14519687, 2.79499717,
       3.57080026, 4.20831242]),
 'std_fit_time': array([7.01610879e-03, 3.54831113e-02, 6.09958496e-02, 2.02213524e-01,
       9.31178110e-02, 5.26354305e-02, 4.07759278e-01, 4.00982156e-04,
       1.56175185e-01, 2.23330995e-02, 4.71362233e-02, 6.27556047e-02,
       1.17624808e-01, 1.01416218e+00, 3.82661696e-02, 3.50318898e-01,
       4.28410846e-01, 7.28126852e-01, 6.99654145e-01, 2.19343482e-01,
       5.82148727e-01, 4.90902439e-04, 3.27641481e-02, 9.46239299e-02,
       1.22973686e-01, 5.04850740e-01, 3.46365246e-01, 2.70117680e-01,
       2.33211473e-03, 5.05393073e-02, 3.13973864e-02, 8.34040357e-02,
       1.29703394e-01, 3.03411476e-01, 1.68545892e-01, 6.21637609e-03,
       2.57572059e-02, 4.86530111e-02, 4.54846185e-02, 6.27143222e-02,
       5.35385284e-02, 1.39822593e-01]),
 'mean_score_time': array([0.00740061, 0.03699813, 0.05960069, 0.0759995 , 0.09340425,
       0.1263998 , 0.15860453, 0.00480213, 0.03260107, 0.05720148,
       0.06800475, 0.1114037 , 0.13740106, 0.2783144 , 0.00999818,
       0.06268044, 0.07987995, 0.13003602, 0.14319844, 0.1792171 ,
       0.17560253, 0.00520372, 0.03180017, 0.07000093, 0.09120054,
       0.14759798, 0.19740248, 0.1799983 , 0.00600319, 0.03000317,
       0.06360221, 0.08979917, 0.11540217, 0.16080265, 0.1838007 ,
       0.00600204, 0.0332046 , 0.06260686, 0.10260344, 0.12000251,
       0.15820017, 0.18123364]),
 'std_score_time': array([0.00224465, 0.01046932, 0.01537148, 0.02389234, 0.03218962,
       0.02792522, 0.04049071, 0.00074677, 0.0105775 , 0.01259223,
       0.00509843, 0.02284908, 0.03218819, 0.12026307, 0.002965  ,
       0.02798326, 0.01815383, 0.03353552, 0.01628747, 0.05109716,
       0.01358454, 0.00040054, 0.00263617, 0.01951455, 0.00495335,
       0.01689639, 0.06236091, 0.00953253, 0.00109441, 0.000634  ,
       0.00873101, 0.0063338 , 0.00755153, 0.0423828 , 0.01090844,
       0.00154902, 0.00402053, 0.00776206, 0.01430509, 0.01146875,
       0.00673526, 0.01743261]),
 'param_max_depth': masked_array(data=[2, 2, 2, 2, 2, 2, 2, 7, 7, 7, 7, 7, 7, 7, 12, 12, 12,
                    12, 12, 12, 12, 17, 17, 17, 17, 17, 17, 17, 22, 22, 22,
                    22, 22, 22, 22, 27, 27, 27, 27, 27, 27, 27],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False],
        fill_value='?',
             dtype=object),
 'param_n_estimators': masked_array(data=[3, 53, 103, 153, 203, 253, 303, 3, 53, 103, 153, 20
3,
                    253, 303, 3, 53, 103, 153, 203, 253, 303, 3, 53, 103,
                    153, 203, 253, 303, 3, 53, 103, 153, 203, 253, 303, 3,
                    53, 103, 153, 203, 253, 303],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False],
        fill_value='?',
             dtype=object),
 'params': [{'max_depth': 2, 'n_estimators': 3},
  {'max_depth': 2, 'n_estimators': 53},
  {'max_depth': 2, 'n_estimators': 103},
  {'max_depth': 2, 'n_estimators': 153},
  {'max_depth': 2, 'n_estimators': 203},
  {'max_depth': 2, 'n_estimators': 253},
  {'max_depth': 2, 'n_estimators': 303},
```

```
 {'max_depth': 7, 'n_estimators': 3},
 {'max_depth': 7, 'n_estimators': 53},
 {'max_depth': 7, 'n_estimators': 103},
 {'max_depth': 7, 'n_estimators': 153},
 {'max_depth': 7, 'n_estimators': 203},
 {'max_depth': 7, 'n_estimators': 253},
 {'max_depth': 7, 'n_estimators': 303},
 {'max_depth': 12, 'n_estimators': 3},
 {'max_depth': 12, 'n_estimators': 53},
 {'max_depth': 12, 'n_estimators': 103},
 {'max_depth': 12, 'n_estimators': 153},
 {'max_depth': 12, 'n_estimators': 203},
 {'max_depth': 12, 'n_estimators': 253},
 {'max_depth': 12, 'n_estimators': 303},
 {'max_depth': 17, 'n_estimators': 3},
 {'max_depth': 17, 'n_estimators': 53},
 {'max_depth': 17, 'n_estimators': 103},
 {'max_depth': 17, 'n_estimators': 153},
 {'max_depth': 17, 'n_estimators': 203},
 {'max_depth': 17, 'n_estimators': 253},
 {'max_depth': 17, 'n_estimators': 303},
 {'max_depth': 22, 'n_estimators': 3},
 {'max_depth': 22, 'n_estimators': 53},
 {'max_depth': 22, 'n_estimators': 103},
 {'max_depth': 22, 'n_estimators': 153},
 {'max_depth': 22, 'n_estimators': 203},
 {'max_depth': 22, 'n_estimators': 253},
 {'max_depth': 22, 'n_estimators': 303},
 {'max_depth': 27, 'n_estimators': 3},
 {'max_depth': 27, 'n_estimators': 53},
 {'max_depth': 27, 'n_estimators': 103},
 {'max_depth': 27, 'n_estimators': 153},
 {'max_depth': 27, 'n_estimators': 203},
 {'max_depth': 27, 'n_estimators': 253},
 {'max_depth': 27, 'n_estimators': 303}],
'split0_test_score': array([0.48557692, 0.52980769, 0.53653846, 0.53365385, 0.5375    ,
       0.53557692, 0.53653846, 0.55576923, 0.6       , 0.60288462,
       0.59807692, 0.59711538, 0.59423077, 0.59326923, 0.56538462,
       0.65384615, 0.66923077, 0.6625    , 0.66538462, 0.66826923,
       0.66826923, 0.55769231, 0.67307692, 0.67115385, 0.67692308,
       0.67596154, 0.68269231, 0.67884615, 0.57692308, 0.65096154,
       0.67403846, 0.66730769, 0.66730769, 0.67307692, 0.67596154,
       0.56730769, 0.66346154, 0.675     , 0.66442308, 0.67211538,
       0.67403846, 0.67115385]),
'split1_test_score': array([0.52307692, 0.55288462, 0.54615385, 0.5375    , 0.5375    ,
       0.53461538, 0.53461538, 0.57307692, 0.60769231, 0.59711538,
       0.60288462, 0.61153846, 0.60384615, 0.60288462, 0.57788462,
       0.65769231, 0.6625    , 0.66442308, 0.6625    , 0.66153846,
       0.66538462, 0.58365385, 0.67692308, 0.68269231, 0.6875    ,
       0.68846154, 0.68461538, 0.67692308, 0.57115385, 0.68173077,
       0.68653846, 0.68365385, 0.68653846, 0.68557692, 0.68557692,
       0.56826923, 0.67307692, 0.6875    , 0.68557692, 0.68269231,
       0.68365385, 0.6875    ]),
'split2_test_score': array([0.49470645, 0.51973051, 0.51684312, 0.51684312, 0.5226179 ,
       0.52743022, 0.52646776, 0.55052936, 0.56977863, 0.56689124,
       0.57651588, 0.57170356, 0.57459095, 0.57555342, 0.56592878,
       0.66698749, 0.6612127 , 0.65928778, 0.65736285, 0.65928778,
       0.6612127 , 0.55822907, 0.6641001 , 0.67564966, 0.6746872 ,
       0.6746872 , 0.67372474, 0.6746872 , 0.58517806, 0.65640038,
       0.66602502, 0.66794995, 0.6746872 , 0.67949952, 0.68046198,
       0.56689124, 0.67179981, 0.67853705, 0.6746872 , 0.68238691,
       0.67949952, 0.68238691]),
'split3_test_score': array([0.48508181, 0.51876805, 0.51588065, 0.5226179 , 0.52069297,
       0.52454283, 0.5226179 , 0.55149182, 0.55630414, 0.56881617,
       0.56400385, 0.56977863, 0.57266603, 0.56977863, 0.5572666 ,
       0.63426372, 0.64292589, 0.64388835, 0.6506256 , 0.64870067,
       0.64773821, 0.55149182, 0.64966314, 0.65255053, 0.65158807,
       0.64773821, 0.54475457, 0.64388835,
       0.65640038, 0.65447546, 0.65832531, 0.65736285, 0.65351299,
```

```
       0.55052936, 0.64581328, 0.65158807, 0.64677575, 0.65255053,
       0.65832531, 0.65543792]),
 'split4_test_score': array([0.4773821 , 0.5360924 , 0.5360924 , 0.53994225, 0.53994225,
       0.54090472, 0.54186718, 0.55149182, 0.57555342, 0.58517806,
       0.57940327, 0.57747834, 0.57747834, 0.57651588, 0.56496631,
       0.64100096, 0.64870067, 0.64581328, 0.64388835, 0.64292589,
       0.64388835, 0.5572666 , 0.65543792, 0.6612127 , 0.6612127 ,
       0.65255053, 0.66698749, 0.66987488, 0.56207892, 0.64677575,
       0.64870067, 0.65351299, 0.66987488, 0.66794995, 0.66217517,
       0.55630414, 0.64966314, 0.65158807, 0.6506256 , 0.65640038,
       0.6612127 , 0.66698749]),
 'mean_test_score': array([0.49316484, 0.53145665, 0.5303017 , 0.53011142, 0.53165063,
       0.53261401, 0.53242134, 0.55647183, 0.5818657 , 0.58417709,
       0.58417691, 0.58552288, 0.58456245, 0.58360036, 0.56628618,
       0.65075813, 0.65691401, 0.6551825 , 0.65595228, 0.65614441,
       0.65729862, 0.56166673, 0.66384023, 0.66865181, 0.67038221,
       0.66845728, 0.67076664, 0.6696139 , 0.56801769, 0.65595136,
       0.6663406 , 0.66537999, 0.67134671, 0.67269323, 0.67153772,
       0.56186033, 0.66076294, 0.66884264, 0.66441771, 0.6692291 ,
       0.67134597, 0.67269323]),
 'std_test_score': array([0.01593183, 0.01250496, 0.01194002, 0.00889928, 0.00823215,
       0.00589128, 0.00696478, 0.00849867, 0.0191583 , 0.01451297,
       0.01436398, 0.01621583, 0.01230023, 0.01242057, 0.00661149,
       0.01173268, 0.00963858, 0.00861542, 0.00784529, 0.00912279,
       0.009719  , 0.01125999, 0.01027749, 0.01064854, 0.0125828 ,
       0.01460335, 0.01399794, 0.01134029, 0.01385685, 0.01356013,
       0.01324602, 0.01099081, 0.00927346, 0.00969517, 0.0119068 ,
       0.00714591, 0.01120203, 0.01466583, 0.01452333, 0.01269226,
       0.00997431, 0.01136555]),
 'rank_test_score': array([42, 39, 40, 41, 38, 36, 37, 35, 30, 27, 28, 25, 26, 29, 32, 24, 1
9,
       23, 21, 20, 18, 34, 16, 11,  7, 12,  6,  8, 31, 22, 13, 14,  4,  2,
        3, 33, 17, 10, 15,  9,  5,  1])}
```

In [ ]:
```
wine_raw_rforest_meanscore = wine_raw_rforest_cv.cv_results_["mean_test_score"]
```

the total number of elements in the array is

In [ ]:
```
len(wine_raw_rforest_meanscore)
```

Out[ ]:
```
42
```

The mean test score of the random forest is arranged with respect to depth first. For our scenerio, 6 maximum depths were used and 7 number of estimators were tried. The total mean test scores produced are 42. We will reshape the array in 7 by 6 to generate the heatmap as shown below:

In [ ]:
```
# save the array using a pandas DataFrame and provide the column names
wine_raw_heatmap_array = pd.DataFrame(wine_raw_rforest_meanscore.reshape(7,6), columns = ["M

# add the index column which is the Number of Estimators
wine_raw_heatmap_array["No. of Estimator"]= [3, 53, 103, 153, 203, 253, 303]

# set the Number of Estimators as the index column
wine_raw_heatmap_array.set_index("No. of Estimator", inplace=True)
```

Now we will preview the data as shown below:

In [ ]:
```
wine_raw_heatmap_array
```

Out[ ]:

No. of Estimator	MD 2	MD 7	MD 12	MD 17	MD 22	MD 27
3	0.493165	0.531457	0.530302	0.530111	0.531651	0.532614
53	0.532421	0.556472	0.581866	0.584177	0.584177	0.585523
103	0.584562	0.583600	0.566286	0.650758	0.656914	0.655182
153	0.655952	0.656144	0.657299	0.561667	0.663840	0.668652
203	0.670382	0.668457	0.670767	0.669614	0.568018	0.655951
253	0.666341	0.665380	0.671347	0.672693	0.671538	0.561860
303	0.660763	0.668843	0.664418	0.669229	0.671346	0.672693

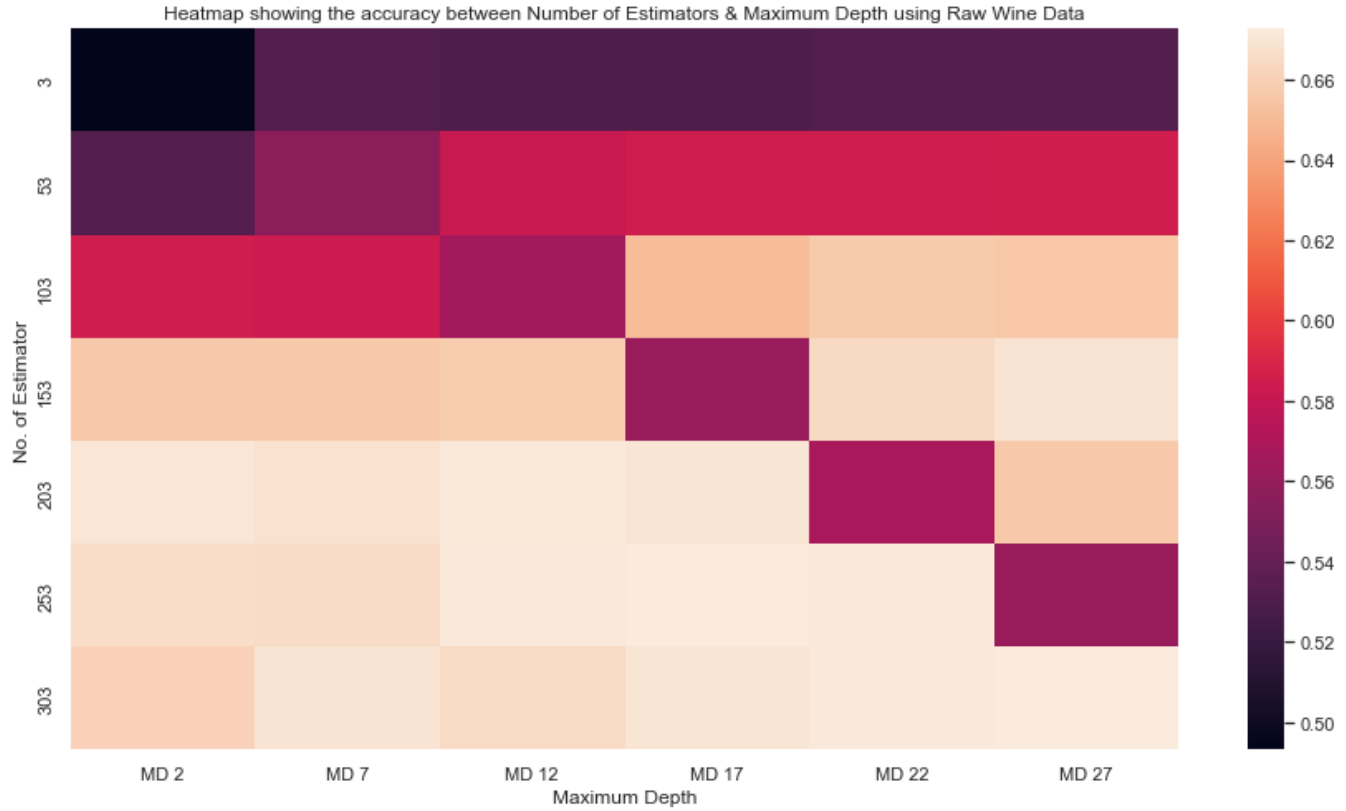
Now we will generate the heatmap using seaborn as shown below:

In [ ]:

```
# initialize the heat map using the heatmap array dataset
sns.heatmap(wine_raw_heatmap_array)

plt.xlabel("Maximum Depth")

plt.title("Heatmap showing the accuracy between Number of Estimators & Maximum Depth using Ra
```

Out[ ]:

```
Text(0.5, 1.0, 'Heatmap showing the accuracy between Number of Estimators & Maximum Depth usi
ng Raw Wine Data')
```



4.2.2.2 PCA Dataset

We will generate the heatmap using the mean test score which was generated by GridSearchCV using the training data as shown below

In [ ]:

```
wine_pca_rforest_cv.cv_results_
```

Out[ ]:

```
{'mean_fit_time': array([0.0293386 , 0.33899903, 0.85815787, 1.19396515, 2.07988586,
       1.70418487, 1.447997  , 0.03099771, 0.49419742, 0.96979809,
       1.3900013 , 1.83339744, 2.35402308, 2.79080224, 0.04039707,
       0.6595973 , 1.2640007 , 1.9623807 , 2.53460021, 3.45101647,
       4.30859838, 0.05139632, 0.78760109, 1.64979658, 2.24699645,
       3.01580181, 4.26739635, 5.48439426, 0.05020041, 0.83520155,
       1.65920296, 2.45740004, 3.15911326, 5.12519898, 5.74455075,
       0.05400057, 1.06720161, 1.55759401, 2.81300211, 3.09140277,
       3.83979974, 4.44140396]),
 'std_fit_time': array([3.87339488e-03, 5.34231810e-02, 9.79205041e-02, 3.97553895e-01,
       4.64708905e-01, 1.19673936e-01, 1.84714053e-02, 4.29524156e-06,
       2.42455311e-02, 6.39912929e-02, 1.57876900e-02, 4.55686186e-02,
       7.37385935e-02, 7.36562393e-02, 4.90353182e-04, 8.38240824e-03,
       1.53746094e-02, 7.78739030e-02, 4.15233229e-02, 5.37343117e-01,
       3.68112641e-01, 1.04956473e-02, 5.48168721e-02, 2.31445380e-01,
       3.55499711e-02, 1.53057481e-01, 2.98364427e-01, 6.41035694e-01,
       1.16631713e-03, 1.47965230e-02, 3.17931734e-02, 1.44363852e-01,
       4.25931910e-01, 9.20547760e-01, 1.00236911e+00, 5.50902573e-03,
       2.35002224e-01, 1.35685544e-01, 5.40706215e-01, 1.59241099e-01,
       7.36681715e-02, 1.65150538e-01]),
 'mean_score_time': array([0.00725884, 0.0256012 , 0.05430512, 0.1163754 , 0.15653915,
       0.12651696, 0.10259886, 0.00420303, 0.03120031, 0.04380145,
       0.06459866, 0.08380442, 0.10860076, 0.14879947, 0.00480418,
       0.02840352, 0.05200076, 0.08780141, 0.10339966, 0.165802  ,
       0.17059898, 0.00580335, 0.03400059, 0.06060357, 0.09039931,
       0.11800294, 0.17180228, 0.2494031 , 0.00540209, 0.03459616,
       0.06580057, 0.10600033, 0.11040092, 0.15820322, 0.23300133,
       0.00679965, 0.03380232, 0.06100521, 0.1068017 , 0.11459937,
       0.15340409, 0.1746007 ]),
 'std_score_time': array([0.00111598, 0.00820758, 0.01185197, 0.07692464, 0.02849284,
       0.03516778, 0.00048926, 0.00040057, 0.00901833, 0.00074774,
       0.00272529, 0.00223005, 0.00744694, 0.05461168, 0.00040236,
       0.00049077, 0.0006334 , 0.0176799 , 0.00174612, 0.03695022,
       0.01491954, 0.00116643, 0.00569108, 0.00233281, 0.00788387,
       0.00901207, 0.02538567, 0.04424796, 0.00135511, 0.0004917 ,
       0.00511026, 0.01735131, 0.00734142, 0.01368678, 0.0786315 ,
       0.00172704, 0.00342879, 0.00260772, 0.02609756, 0.00535376,
       0.02845709, 0.01976949]),
 'param_max_depth': masked_array(data=[2, 2, 2, 2, 2, 2, 2, 7, 7, 7, 7, 7, 7, 7, 12, 12, 12,
                    12, 12, 12, 12, 17, 17, 17, 17, 17, 17, 17, 22, 22, 22,
                    22, 22, 22, 22, 27, 27, 27, 27, 27, 27, 27],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False],
        fill_value='?',
             dtype=object),
 'param_n_estimators': masked_array(data=[3, 53, 103, 153, 203, 253, 303, 3, 53, 103, 153, 20
3,
                    253, 303, 3, 53, 103, 153, 203, 253, 303, 3, 53, 103,
                    153, 203, 253, 303, 3, 53, 103, 153, 203, 253, 303, 3,
                    53, 103, 153, 203, 253, 303],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False],
        fill_value='?',
             dtype=object),
 'params': [{'max_depth': 2, 'n_estimators': 3},
  {'max_depth': 2, 'n_estimators': 53},
  {'max_depth': 2, 'n_estimators': 103},
  {'max_depth': 2, 'n_estimators': 153},
  {'max_depth': 2, 'n_estimators': 203},
  {'max_depth': 2, 'n_estimators': 253},
  {'max_depth': 2, 'n_estimators': 303},
```

```
 {'max_depth': 7, 'n_estimators': 3},
 {'max_depth': 7, 'n_estimators': 53},
 {'max_depth': 7, 'n_estimators': 103},
 {'max_depth': 7, 'n_estimators': 153},
 {'max_depth': 7, 'n_estimators': 203},
 {'max_depth': 7, 'n_estimators': 253},
 {'max_depth': 7, 'n_estimators': 303},
 {'max_depth': 12, 'n_estimators': 3},
 {'max_depth': 12, 'n_estimators': 53},
 {'max_depth': 12, 'n_estimators': 103},
 {'max_depth': 12, 'n_estimators': 153},
 {'max_depth': 12, 'n_estimators': 203},
 {'max_depth': 12, 'n_estimators': 253},
 {'max_depth': 12, 'n_estimators': 303},
 {'max_depth': 17, 'n_estimators': 3},
 {'max_depth': 17, 'n_estimators': 53},
 {'max_depth': 17, 'n_estimators': 103},
 {'max_depth': 17, 'n_estimators': 153},
 {'max_depth': 17, 'n_estimators': 203},
 {'max_depth': 17, 'n_estimators': 253},
 {'max_depth': 17, 'n_estimators': 303},
 {'max_depth': 22, 'n_estimators': 3},
 {'max_depth': 22, 'n_estimators': 53},
 {'max_depth': 22, 'n_estimators': 103},
 {'max_depth': 22, 'n_estimators': 153},
 {'max_depth': 22, 'n_estimators': 203},
 {'max_depth': 22, 'n_estimators': 253},
 {'max_depth': 22, 'n_estimators': 303},
 {'max_depth': 27, 'n_estimators': 3},
 {'max_depth': 27, 'n_estimators': 53},
 {'max_depth': 27, 'n_estimators': 103},
 {'max_depth': 27, 'n_estimators': 153},
 {'max_depth': 27, 'n_estimators': 203},
 {'max_depth': 27, 'n_estimators': 253},
 {'max_depth': 27, 'n_estimators': 303}],
'split0_test_score': array([0.45576923, 0.5       , 0.48846154, 0.49423077, 0.49230769,
       0.49230769, 0.49230769, 0.52307692, 0.57115385, 0.57788462,
       0.57788462, 0.57692308, 0.57788462, 0.58173077, 0.55576923,
       0.63076923, 0.64134615, 0.64423077, 0.63846154, 0.64519231,
       0.64230769, 0.54903846, 0.65673077, 0.66153846, 0.66730769,
       0.66538462, 0.66442308, 0.65961538, 0.54230769, 0.65961538,
       0.65      , 0.64903846, 0.64807692, 0.65      , 0.65288462,
       0.56057692, 0.65288462, 0.64519231, 0.65192308, 0.65      ,
       0.65384615, 0.65384615]),
'split1_test_score': array([0.50384615, 0.51346154, 0.51634615, 0.51153846, 0.51153846,
       0.51153846, 0.51346154, 0.54519231, 0.58365385, 0.59711538,
       0.59519231, 0.59423077, 0.59230769, 0.59423077, 0.54903846,
       0.64615385, 0.65673077, 0.6625    , 0.65288462, 0.65673077,
       0.65576923, 0.55769231, 0.64711538, 0.66057692, 0.66826923,
       0.66634615, 0.66634615, 0.66346154, 0.55673077, 0.66153846,
       0.66923077, 0.67307692, 0.67403846, 0.66923077, 0.67019231,
       0.55192308, 0.67019231, 0.66826923, 0.66730769, 0.67307692,
       0.67115385, 0.66442308]),
'split2_test_score': array([0.49374398, 0.51395573, 0.50818094, 0.50240616, 0.50625602,
       0.50529355, 0.50625602, 0.533205  , 0.5813282 , 0.58517806,
       0.58421559, 0.58902791, 0.58421559, 0.58614052, 0.56111646,
       0.64677575, 0.64485082, 0.64196343, 0.64966314, 0.65158807,
       0.6506256 , 0.57844081, 0.65543792, 0.6506256 , 0.65543792,
       0.65543792, 0.66602502, 0.6641001 , 0.55630414, 0.65928778,
       0.66217517, 0.66987488, 0.6612127 , 0.6612127 , 0.66794995,
       0.56881617, 0.65928778, 0.66025024, 0.67661213, 0.67372474,
       0.66987488, 0.66987488]),
'split3_test_score': array([0.49278152, 0.50336862, 0.50818094, 0.50048123, 0.4985563 ,
       0.4985563 , 0.49759384, 0.5360924 , 0.57459095, 0.57844081,
       0.57362849, 0.57747834, 0.57555342, 0.57362849, 0.56207892,
       0.61405197, 0.61886429, 0.63137632, 0.63522618, 0.63426372,
       0.63522618, 0.51780558, 0.626564  , 0.6400385 , 0.64196343,
       0.63426372, 0.53224254, 0.63618864,
       0.63811357, 0.63715111, 0.63907603, 0.64196343, 0.6400385 ,
```

```
       0.55437921, 0.62848893, 0.63137632, 0.63426372, 0.64100096,
       0.63522618, 0.63330125]),
 'split4_test_score': array([0.47545717, 0.51010587, 0.49759384, 0.5014437 , 0.50625602,
       0.50433109, 0.50048123, 0.53801732, 0.56592878, 0.56977863,
       0.57844081, 0.57266603, 0.56977863, 0.57170356, 0.55822907,
       0.60923965, 0.61886429, 0.61982676, 0.62175168, 0.61886429,
       0.62271415, 0.55822907, 0.62560154, 0.63330125, 0.63522618,
       0.63041386, 0.63522618, 0.63618864, 0.52839269, 0.62463908,
       0.63426372, 0.63233879, 0.63041386, 0.62848893, 0.63426372,
       0.5226179 , 0.62463908, 0.62752647, 0.62271415, 0.62848893,
       0.62560154, 0.62560154]),
 'mean_test_score': array([0.48431961, 0.50817835, 0.50375268, 0.50202006, 0.5029829 ,
       0.50240542, 0.50202006, 0.53511679, 0.57533112, 0.5816795 ,
       0.58187236, 0.58206523, 0.57994799, 0.58148682, 0.55724643,
       0.62939809, 0.63613127, 0.63997946, 0.63959743, 0.64132783,
       0.64132857, 0.55224125, 0.64228992, 0.64921615, 0.65364089,
       0.65133172, 0.6540268 , 0.65152588, 0.54319557, 0.64825387,
       0.65075664, 0.65229603, 0.6505636 , 0.65017917, 0.65306582,
       0.55166266, 0.64709854, 0.64652291, 0.65056415, 0.65325831,
       0.65114052, 0.64940938]),
 'std_test_score': array([0.01694329, 0.0055682 , 0.00969087, 0.00555198, 0.00675543,
       0.00651524, 0.00728204, 0.00720409, 0.00650591, 0.00913266,
       0.00746392, 0.00815237, 0.00771959, 0.00826547, 0.0046658 ,
       0.01566196, 0.01499241, 0.01420795, 0.01110618, 0.01350533,
       0.01165901, 0.01973858, 0.01364184, 0.01113918, 0.01326246,
       0.01433553, 0.01421619, 0.01341062, 0.01178914, 0.01503689,
       0.01344703, 0.01660803, 0.01554755, 0.01430174, 0.01440914,
       0.01565089, 0.01769815, 0.01583285, 0.02001008, 0.0178103 ,
       0.01824465, 0.01726394]),
 'rank_test_score': array([42, 36, 37, 40, 38, 39, 40, 35, 30, 27, 26, 25, 29, 28, 31, 24, 2
3,
       21, 22, 20, 19, 32, 18, 14,  2,  7,  1,  6, 34, 15,  9,  5, 11, 12,
        4, 33, 16, 17, 10,  3,  8, 13])}
```

In [ ]:
```
wine_pca_rforest_meanscore = wine_pca_rforest_cv.cv_results_["mean_test_score"]
```

the total number of elements in the array is

In [ ]:
```
len(wine_pca_rforest_meanscore)
```

Out[ ]:
```
42
```

The mean test score of the random forest is arranged with respect to depth first. For our scenerio, 6 maximum depths were used and 7 number of estimators were tried. The total mean test scores produced are 42. We will reshape the array in 7 by 6 to generate the heatmap as shown below:

In [ ]:
```
# save the array using a pandas DataFrame and provide the column names
wine_pca_heatmap_array = pd.DataFrame(wine_pca_rforest_meanscore.reshape(7,6), columns = ["MI

# add the index column which is the Number of Estimators
wine_pca_heatmap_array["No. of Estimator"]= [3, 53, 103, 153, 203, 253, 303]

# set the Number of Estimators as the index column
wine_pca_heatmap_array.set_index("No. of Estimator", inplace=True)
```

Now we will preview the data as shown below:

In [ ]:
```
wine_pca_heatmap_array
```

Out[ ]:

No. of Estimator	MD 2	MD 7	MD 12	MD 17	MD 22	MD 27
3	0.484320	0.508178	0.503753	0.502020	0.502983	0.502405
53	0.502020	0.535117	0.575331	0.581679	0.581872	0.582065
103	0.579948	0.581487	0.557246	0.629398	0.636131	0.639979
153	0.639597	0.641328	0.641329	0.552241	0.642290	0.649216
203	0.653641	0.651332	0.654027	0.651526	0.543196	0.648254
253	0.650757	0.652296	0.650564	0.650179	0.653066	0.551663
303	0.647099	0.646523	0.650564	0.653258	0.651141	0.649409

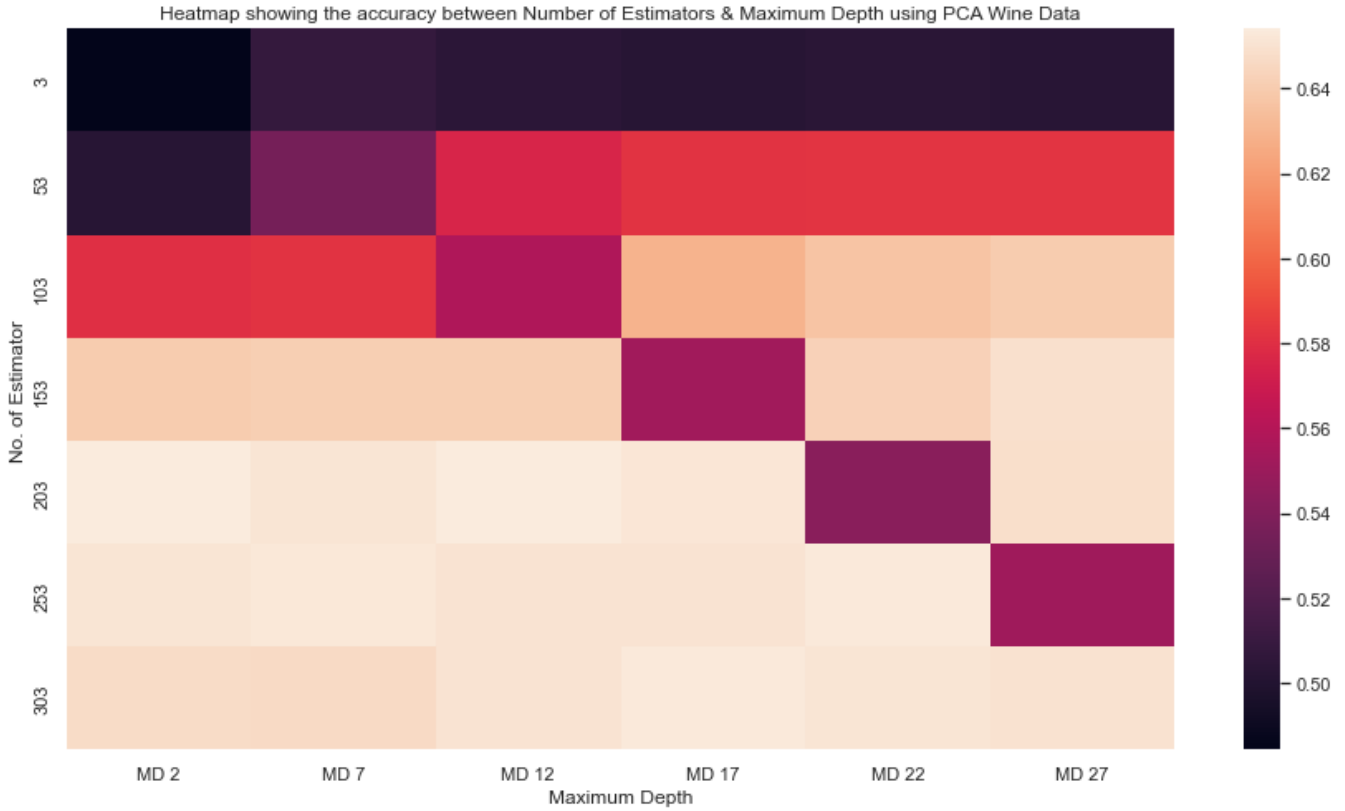
Now we will generate the heatmap using seaborn as shown below:

In [ ]:

```
# initialize the heat map using the heatmap array dataset
sns.heatmap(wine_pca_heatmap_array)

plt.xlabel("Maximum Depth")

plt.title("Heatmap showing the accuracy between Number of Estimators & Maximum Depth using PC
```

Out[ ]:

```
Text(0.5, 1.0, 'Heatmap showing the accuracy between Number of Estimators & Maximum Depth usi
ng PCA Wine Data')
```



4.2.2.3 LDA Dataset

We will generate the heatmap using the mean test score which was generated by GridSearchCV using the training data as shown below

In [ ]:

```
wine_lda_rforest_cv.cv_results_
```

Out[ ]:
```
{'mean_fit_time': array([0.03079834, 0.23559966, 0.55119777, 0.6977983 , 0.8651978 ,
       1.12619972, 1.28719468, 0.02359829, 0.33839946, 0.67139678,
       1.1885983 , 1.60179653, 2.56990414, 2.1139998 , 0.0335988 ,
       0.44520082, 0.86780028, 1.3507967 , 1.84000306, 2.13780241,
       2.61779895, 0.0333971 , 0.56859941, 1.07020245, 1.55939856,
       2.02959924, 2.58819723, 3.13519559, 0.05260262, 0.73100076,
       1.23540158, 1.92443185, 2.5263772 , 4.1754921 , 3.84675875,
       0.03819814, 0.66760149, 1.36620069, 1.71219964, 2.44239836,
       3.23840113, 3.87420149]),
 'std_fit_time': array([2.71158272e-03, 1.75881153e-02, 9.83863465e-02, 6.60308888e-02,
       5.83329588e-02, 1.11598580e-01, 9.19455819e-02, 1.02110074e-03,
       2.57676450e-03, 3.43729160e-02, 3.71334792e-01, 2.38280220e-01,
       5.39820607e-01, 2.47839432e-01, 7.91259904e-03, 9.49420497e-03,
       7.73060271e-03, 1.12023795e-01, 3.43102350e-01, 6.66578321e-02,
       5.91271942e-02, 1.02108683e-03, 2.39854034e-02, 4.86021343e-02,
       8.63972491e-02, 6.62606207e-02, 1.03702140e-01, 2.21741250e-01,
       1.02126934e-02, 1.62400411e-01, 7.70072100e-02, 1.71717311e-01,
       1.67639723e-01, 1.15691805e+00, 2.84638989e-01, 1.59950354e-03,
       7.85682461e-02, 1.18353291e-01, 3.17163087e-02, 2.65585568e-01,
       5.63405516e-01, 5.04458458e-01]),
 'mean_score_time': array([0.00740371, 0.02260156, 0.04180241, 0.0622005 , 0.08640223,
       0.09340158, 0.11540146, 0.00440273, 0.02419806, 0.04420223,
       0.08200331, 0.12400374, 0.15051403, 0.12720037, 0.0044035 ,
       0.0283998 , 0.0550014 , 0.08920283, 0.11199927, 0.1527987 ,
       0.21100125, 0.00460253, 0.03640366, 0.05779996, 0.08760166,
       0.11140108, 0.14000344, 0.17200737, 0.00679579, 0.12020049,
       0.06339993, 0.10800662, 0.13199897, 0.22271171, 0.19656162,
       0.00480094, 0.04960008, 0.06279898, 0.10400071, 0.12340655,
       0.16819978, 0.25739913]),
 'std_score_time': array([0.00079854, 0.00162102, 0.00370901, 0.01051324, 0.01536871,
       0.00431555, 0.02232948, 0.00048893, 0.00075018, 0.0007481 ,
       0.02761567, 0.06165816, 0.01600882, 0.01101883, 0.00049113,
       0.00048728, 0.00252542, 0.00746636, 0.00282853, 0.01358527,
       0.03488932, 0.00048986, 0.00523616, 0.00462244, 0.00360954,
       0.00048726, 0.00316297, 0.00400217, 0.00193511, 0.16501431,
       0.00531481, 0.02074599, 0.03226841, 0.0612529 , 0.02336067,
       0.00039979, 0.03028743, 0.00621445, 0.03157928, 0.01107116,
       0.04116944, 0.13328888]),
 'param_max_depth': masked_array(data=[2, 2, 2, 2, 2, 2, 2, 7, 7, 7, 7, 7, 7, 7, 12, 12, 12,
                    12, 12, 12, 12, 17, 17, 17, 17, 17, 17, 17, 22, 22, 22,
                    22, 22, 22, 22, 27, 27, 27, 27, 27, 27, 27],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False],
        fill_value='?',
             dtype=object),
 'param_n_estimators': masked_array(data=[3, 53, 103, 153, 203, 253, 303, 3, 53, 103, 153, 20
3,
                    253, 303, 3, 53, 103, 153, 203, 253, 303, 3, 53, 103,
                    153, 203, 253, 303, 3, 53, 103, 153, 203, 253, 303, 3,
                    53, 103, 153, 203, 253, 303],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False],
        fill_value='?',
             dtype=object),
 'params': [{'max_depth': 2, 'n_estimators': 3},
  {'max_depth': 2, 'n_estimators': 53},
  {'max_depth': 2, 'n_estimators': 103},
  {'max_depth': 2, 'n_estimators': 153},
  {'max_depth': 2, 'n_estimators': 203},
  {'max_depth': 2, 'n_estimators': 253},
  {'max_depth': 2, 'n_estimators': 303},
```

```
 {'max_depth': 7, 'n_estimators': 3},
 {'max_depth': 7, 'n_estimators': 53},
 {'max_depth': 7, 'n_estimators': 103},
 {'max_depth': 7, 'n_estimators': 153},
 {'max_depth': 7, 'n_estimators': 203},
 {'max_depth': 7, 'n_estimators': 253},
 {'max_depth': 7, 'n_estimators': 303},
 {'max_depth': 12, 'n_estimators': 3},
 {'max_depth': 12, 'n_estimators': 53},
 {'max_depth': 12, 'n_estimators': 103},
 {'max_depth': 12, 'n_estimators': 153},
 {'max_depth': 12, 'n_estimators': 203},
 {'max_depth': 12, 'n_estimators': 253},
 {'max_depth': 12, 'n_estimators': 303},
 {'max_depth': 17, 'n_estimators': 3},
 {'max_depth': 17, 'n_estimators': 53},
 {'max_depth': 17, 'n_estimators': 103},
 {'max_depth': 17, 'n_estimators': 153},
 {'max_depth': 17, 'n_estimators': 203},
 {'max_depth': 17, 'n_estimators': 253},
 {'max_depth': 17, 'n_estimators': 303},
 {'max_depth': 22, 'n_estimators': 3},
 {'max_depth': 22, 'n_estimators': 53},
 {'max_depth': 22, 'n_estimators': 103},
 {'max_depth': 22, 'n_estimators': 153},
 {'max_depth': 22, 'n_estimators': 203},
 {'max_depth': 22, 'n_estimators': 253},
 {'max_depth': 22, 'n_estimators': 303},
 {'max_depth': 27, 'n_estimators': 3},
 {'max_depth': 27, 'n_estimators': 53},
 {'max_depth': 27, 'n_estimators': 103},
 {'max_depth': 27, 'n_estimators': 153},
 {'max_depth': 27, 'n_estimators': 203},
 {'max_depth': 27, 'n_estimators': 253},
 {'max_depth': 27, 'n_estimators': 303}],
'split0_test_score': array([0.49230769, 0.54423077, 0.54230769, 0.54230769, 0.54230769,
       0.54230769, 0.54134615, 0.55673077, 0.56730769, 0.57211538,
       0.57403846, 0.57019231, 0.57211538, 0.57211538, 0.54519231,
       0.60288462, 0.6       , 0.60576923, 0.60384615, 0.6       ,
       0.60096154, 0.54326923, 0.63076923, 0.63076923, 0.63653846,
       0.63846154, 0.63942308, 0.64423077, 0.53557692, 0.62211538,
       0.63653846, 0.63365385, 0.62403846, 0.62692308, 0.63173077,
       0.54326923, 0.61634615, 0.62019231, 0.625     , 0.62307692,
       0.625     , 0.62884615]),
'split1_test_score': array([0.45192308, 0.53173077, 0.53076923, 0.53173077, 0.53269231,
       0.53173077, 0.53173077, 0.54038462, 0.57115385, 0.56538462,
       0.5625    , 0.56346154, 0.57019231, 0.56923077, 0.52980769,
       0.59615385, 0.6       , 0.59711538, 0.60096154, 0.60288462,
       0.60576923, 0.56826923, 0.63076923, 0.63557692, 0.63173077,
       0.63076923, 0.63076923, 0.63557692, 0.53365385, 0.62980769,
       0.63365385, 0.63557692, 0.63269231, 0.62596154, 0.62307692,
       0.54615385, 0.625     , 0.62307692, 0.62307692, 0.62115385,
       0.61923077, 0.61730769]),
'split2_test_score': array([0.43792108, 0.53801732, 0.53801732, 0.53801732, 0.53994225,
       0.53897979, 0.53994225, 0.51973051, 0.56785371, 0.5707411 ,
       0.5707411 , 0.56977863, 0.56689124, 0.56400385, 0.54379211,
       0.6053898 , 0.60250241, 0.60346487, 0.60250241, 0.60442733,
       0.60635226, 0.56111646, 0.63137632, 0.62560154, 0.63233879,
       0.63907603, 0.64100096, 0.63715111, 0.52839269, 0.6294514 ,
       0.63618864, 0.63811357, 0.63330125, 0.63041386, 0.63041386,
       0.55052936, 0.63233879, 0.63137632, 0.63330125, 0.63137632,
       0.63618864, 0.6294514 ]),
'split3_test_score': array([0.48893167, 0.533205  , 0.533205  , 0.53224254, 0.53224254,
       0.53224254, 0.53224254, 0.55822907, 0.56496631, 0.55919153,
       0.56304139, 0.56400385, 0.56400385, 0.56496631, 0.55245428,
       0.59769009, 0.5919153 , 0.59095284, 0.59384023, 0.59095284,
       0.59576516, 0.5572666 , 0.62848893, 0.626564  , 0.626564  ,
       0.53705486, 0.62175168,
       0.61982676, 0.6294514 , 0.6294514 , 0.63041386, 0.62752647,
```

```
        0.53801732, 0.62848893, 0.63426372, 0.63426372, 0.6294514 ,
        0.62271415, 0.62175168]),
 'split4_test_score': array([0.466795  , 0.53705486, 0.53705486, 0.53705486, 0.53705486,
        0.5360924 , 0.53705486, 0.54475457, 0.56785371, 0.57170356,
        0.57170356, 0.56977863, 0.56977863, 0.56881617, 0.54956689,
        0.59095284, 0.59961501, 0.60153994, 0.59961501, 0.60250241,
        0.60346487, 0.56015399, 0.61982676, 0.61982676, 0.61886429,
        0.62367661, 0.62175168, 0.62367661, 0.55437921, 0.6159769 ,
        0.61982676, 0.61982676, 0.6159769 , 0.61790183, 0.61790183,
        0.53801732, 0.61116458, 0.60635226, 0.62175168, 0.61116458,
        0.61501444, 0.60827719]),
 'mean_test_score': array([0.4675757 , 0.53684775, 0.53627082, 0.53627064, 0.53684793,
        0.53627064, 0.53646332, 0.54396591, 0.56782705, 0.56782724,
        0.5684049 , 0.56744299, 0.56859628, 0.5678265 , 0.54416266,
        0.59861424, 0.59880654, 0.59976845, 0.60015307, 0.60015344,
        0.60246261, 0.5580151 , 0.62824609, 0.62766769, 0.62920726,
        0.63267195, 0.63305675, 0.63459484, 0.53781151, 0.62382061,
        0.62920689, 0.6313245 , 0.62709206, 0.62632283, 0.62612997,
        0.54319742, 0.62266769, 0.62305231, 0.62747871, 0.62324461,
        0.6236296 , 0.62112682]),
 'std_test_score': array([0.02094163, 0.00436744, 0.00399438, 0.00392296, 0.00394729,
        0.0040163 , 0.0039117 , 0.01390814, 0.00197598, 0.00494867,
        0.00472676, 0.00303805, 0.00283996, 0.00297117, 0.00781306,
        0.00509409, 0.00359586, 0.00524528, 0.00346275, 0.00481493,
        0.00385167, 0.00821391, 0.00432377, 0.00527528, 0.0060644 ,
        0.00567004, 0.00688757, 0.00670295, 0.00878698, 0.0052204 ,
        0.00772329, 0.00640792, 0.0064553 , 0.00457989, 0.00507335,
        0.00482016, 0.00781204, 0.00981893, 0.00525846, 0.00714108,
        0.00712764, 0.00786099]),
 'rank_test_score': array([42, 37, 39, 40, 36, 40, 38, 33, 28, 27, 26, 30, 25, 29, 32, 24, 2
3,
        22, 21, 20, 19, 31,  7,  8,  5,  3,  2,  1, 35, 13,  6,  4, 10, 11,
        12, 34, 17, 16,  9, 15, 14, 18])}
```

In [ ]:
```
wine_lda_rforest_meanscore = wine_lda_rforest_cv.cv_results_["mean_test_score"]
```

the total number of elements in the array is

In [ ]:
```
len(wine_lda_rforest_meanscore)
```

Out[ ]:
```
42
```

The mean test score of the random forest is arranged with respect to depth first. For our scenerio, 6 maximum depths were used and 7 number of estimators were tried. The total mean test scores produced are 42. We will reshape the array in 7 by 6 to generate the heatmap as shown below:

In [ ]:
```
# save the array using a pandas DataFrame and provide the column names
wine_lda_heatmap_array = pd.DataFrame(wine_lda_rforest_meanscore.reshape(7,6), columns = ["MI

# add the index column which is the Number of Estimators
wine_lda_heatmap_array["No. of Estimator"]= [3, 53, 103, 153, 203, 253, 303]

# set the Number of Estimators as the index column
wine_lda_heatmap_array.set_index("No. of Estimator", inplace=True)
```

Now we will preview the data as shown below:

In [ ]:
```
wine_lda_heatmap_array
```

Out[ ]:

No. of Estimator	MD 2	MD 7	MD 12	MD 17	MD 22	MD 27
3	0.467576	0.536848	0.536271	0.536271	0.536848	0.536271
53	0.536463	0.543966	0.567827	0.567827	0.568405	0.567443
103	0.568596	0.567826	0.544163	0.598614	0.598807	0.599768
153	0.600153	0.600153	0.602463	0.558015	0.628246	0.627668
203	0.629207	0.632672	0.633057	0.634595	0.537812	0.623821
253	0.629207	0.631324	0.627092	0.626323	0.626130	0.543197
303	0.622668	0.623052	0.627479	0.623245	0.623630	0.621127

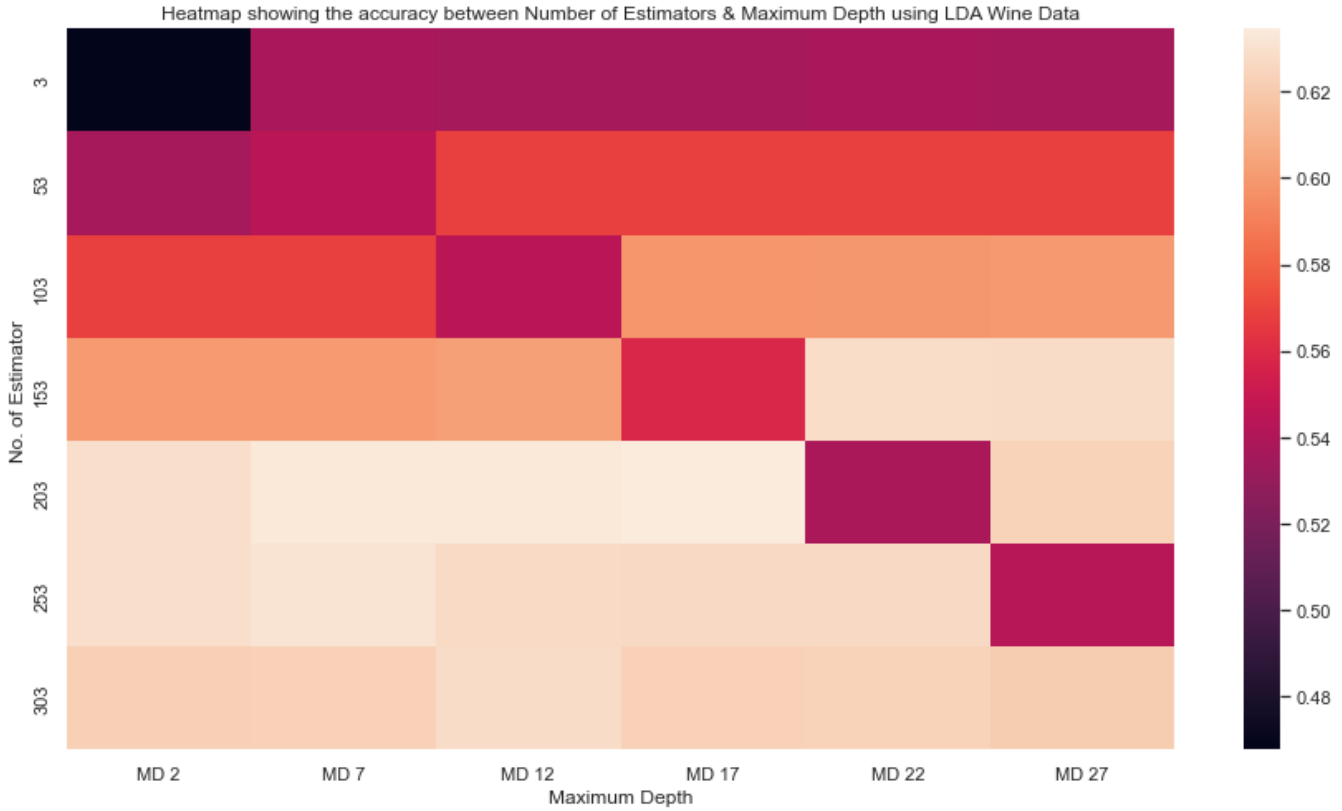
Now we will generate the heatmap using seaborn as shown below:

In [ ]:

```
# initialize the heat map using the heatmap array dataset
sns.heatmap(wine_lda_heatmap_array)

plt.xlabel("Maximum Depth")

plt.title("Heatmap showing the accuracy between Number of Estimators & Maximum Depth using LD
```

Out[ ]:

```
Text(0.5, 1.0, 'Heatmap showing the accuracy between Number of Estimators & Maximum Depth usi
ng LDA Wine Data')
```



Question 5: Gradient Tree Boosting

5.1 Gradient Tree using Cross Validation

5.1.1 Abalone Dataset

5.1.1.1 Raw Data

For the gradient tree boosting algorithm, we will only be varying the number of estimators only and try to ascertain which is the best hyperparameter that yields the best accuracy. We start by initializing the GridSearchCV function as shown below:

In [ ]:
```
# create an array for the number of estimators for the classifier parameters
abalone_gtboost_no_estimators = list(range(3,204,50))

# generate a dictionary for the parameters of random forest classifier
parameters = {"n_estimators": abalone_gtboost_no_estimators}

# initialize the GridSearchCV function
abalone_raw_gtboosting_cv = GridSearchCV(GradientBoostingClassifier(random_state=27), parame

# fit the training data using the GridSearchCV function
abalone_raw_gtboosting_cv.fit(abalone_raw_train, abalone_rings_raw_train)

# display the results of the tuning of the hyperparameters
abalone_raw_gtboosting_cv.cv_results_
```

```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\model_select
ion\_split.py:676: UserWarning: The least populated class in y has only 1 members, which is l
ess than n_splits=5.
  warnings.warn(
```

Out[ ]:
```
{'mean_fit_time': array([ 0.90939808, 13.73690434, 27.56994538, 41.66027298, 51.00899935]),
 'std_fit_time': array([0.22213655, 1.15131814, 2.4545933 , 6.38507814, 4.45273781]),
 'mean_score_time': array([0.00900154, 0.02519946, 0.05039997, 0.0617959 , 0.0734014 ]),
 'std_score_time': array([0.00178937, 0.00075056, 0.01357713, 0.00808334, 0.00873236]),
 'param_n_estimators': masked_array(data=[3, 53, 103, 153, 203],
              mask=[False, False, False, False, False],
        fill_value='?',
             dtype=object),
 'params': [{'n_estimators': 3},
  {'n_estimators': 53},
  {'n_estimators': 103},
  {'n_estimators': 153},
  {'n_estimators': 203}],
 'split0_test_score': array([0.2406577 , 0.24364723, 0.23467862, 0.22869955, 0.23617339]),
 'split1_test_score': array([0.25598802, 0.23353293, 0.24850299, 0.23952096, 0.23952096]),
 'split2_test_score': array([0.25      , 0.23353293, 0.24251497, 0.2260479 , 0.22155689]),
 'split3_test_score': array([0.25299401, 0.26497006, 0.24251497, 0.21556886, 0.22155689]),
 'split4_test_score': array([0.24401198, 0.25598802, 0.24550898, 0.24700599, 0.24251497]),
 'mean_test_score': array([0.24873034, 0.24633424, 0.24274411, 0.23136865, 0.23226462]),
 'std_test_score': array([0.00565501, 0.01245356, 0.00460361, 0.01091837, 0.0089701 ]),
 'rank_test_score': array([1, 2, 3, 5, 4])}
```

The best hyperparameters for the gradient boost tree are:

In [ ]:
```
abalone_raw_gtboosting_cv.best_params_
```

Out[ ]:
```
{'n_estimators': 3}
```

The score achieved by the best hyperparameters is:

In [ ]:
```
abalone_raw_gtboosting_cv.best_score_
```

Out[ ]:
```
0.24873034200656985
```

Now we will initialize a new gradient tree and train it with the best hyperparameters that were discovered by GridSearchCV

In [ ]:
```
# initialize the decision tree classifier with the best hyperparemeter
abalone_raw_gtboosting = GradientBoostingClassifier(random_state=27, n_estimators= 3)

# train the classifier using the training dataset
abalone_raw_gtboosting.fit(abalone_raw_train, abalone_rings_raw_train)

# view the accuracy of the classifier using the testing dataset
y = abalone_raw_gtboosting.score(abalone_raw_test, abalon
abalone_raw_gtboosting_test_accuracy
```

Out[ ]:
```
0.2535885167464115
```

5.1.1.2 PCA Data

In [ ]:
```
# create an array for the number of estimators for the classifier parameters
abalone_gtboost_no_estimators = list(range(3,204,50))

# generate a dictionary for the parameters of random forest classifier
parameters = {"n_estimators": abalone_gtboost_no_estimators}

# initialize the GridSearchCV function
abalone_pca_gtboosting_cv = GridSearchCV(GradientBoostingClassifier(random_state=27), parame

# fit the training data using the GridSearchCV function
abalone_pca_gtboosting_cv.fit(abalone_PCAminmax_train , abalone_rings_PCAminmax_train)

# display the results of the tuning of the hyperprameters
abalone_pca_gtboosting_cv.cv_results_
```

Out[ ]:
```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\model_select
ion\_split.py:676: UserWarning: The least populated class in y has only 1 members, which is l
ess than n_splits=5.
  warnings.warn(
{'mean_fit_time': array([ 0.72759986, 12.46819978, 27.77779679, 37.76960011, 44.38211308]),
 'std_fit_time': array([0.05698927, 0.34519657, 3.17447328, 1.81388216, 2.79316763]),
 'mean_score_time': array([0.00859857, 0.02679834, 0.04320207, 0.06379757, 0.08599825]),
 'std_score_time': array([0.0018539 , 0.00204422, 0.00171553, 0.00624448, 0.0387219 ]),
 'param_n_estimators': masked_array(data=[3, 53, 103, 153, 203],
              mask=[False, False, False, False, False],
        fill_value='?',
             dtype=object),
 'params': [{'n_estimators': 3},
  {'n_estimators': 53},
  {'n_estimators': 103},
  {'n_estimators': 153},
  {'n_estimators': 203}],
 'split0_test_score': array([0.22122571, 0.24364723, 0.24962631, 0.24663677, 0.23467862]),
 'split1_test_score': array([0.23353293, 0.24550898, 0.2245509 , 0.2260479 , 0.21706587]),
 'split2_test_score': array([0.26197605, 0.23203593, 0.21856287, 0.23502994, 0.23353293]),
 'split3_test_score': array([0.24700599, 0.24401198, 0.24101796, 0.23203593, 0.24101796]),
 'split4_test_score': array([0.24101796, 0.25898204, 0.22005988, 0.21556886, 0.20658683]),
 'mean_test_score': array([0.24095173, 0.24483723, 0.23076358, 0.23106388, 0.22657644]),
 'std_test_score': array([0.01358554, 0.00856151, 0.01235297, 0.01024153, 0.01274858]),
 'rank_test_score': array([2, 1, 4, 3, 5])}
```

The best hyperparameters for the gradient boost tree are:

In [ ]:
```
abalone_pca_gtboosting_cv.best_params_
```

Out[ ]:
```
{'n_estimators': 53}
```

The score achieved by the best hyperparameters is:

In [ ]:
```
abalone_pca_gtboosting_cv.best_score_
```

Out[ ]:
```
0.2448372313668627
```

Now we will initialize a new gradient tree and train it with the best hyperparameters that were discovered by GridSearchCV

In [ ]:
```
# initialize the decision tree classifier with the best hyperparemeter
abalone_pca_gtboosting = GradientBoostingClassifier(random_state=27, n_estimators= 53)
# train the classifier using the training dataset
```

```
abalone_pca_gtboosting.fit(abalone_PCAminmax_train , abalone_rings_PCAminmax_train)

# view the accuracy of the classifier using the testing dataset
abalone_pca_gtboosting_test_accuracy = abalone_pca_gtboosting.score(abalone_PCAminmax_test ,
abalone_pca_gtboosting_test_accuracy
```

Out[ ]:
```
0.23923444976076555
```

5.1.1.3 LDA Data

In [ ]:
```
# create an array for the number of estimators for the classifier parameters
abalone_gtboost_no_estimators = list(range(3,204,50))

# generate a dictionary for the parameters of random forest classifier
parameters = {"n_estimators": abalone_gtboost_no_estimators}

# initialize the GridSearchCV function
abalone_lda_gtboosting_cv = GridSearchCV(GradientBoostingClassifier(random_state=27), parame

# fit the training data using the GridSearchCV function
abalone_lda_gtboosting_cv.fit(abalone_LDAminmax_train , abalone_rings_LDAminmax_train)

# display the results of the tuning of the hyperprameters
abalone_lda_gtboosting_cv.cv_results_
```

Out[ ]:
```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\model_select
ion\_split.py:676: UserWarning: The least populated class in y has only 1 members, which is l
ess than n_splits=5.
  warnings.warn(
{'mean_fit_time': array([ 0.72340341, 12.76328931, 27.44009423, 37.41540041, 48.25794959]),
 'std_fit_time': array([0.06366081, 0.61075682, 2.99484953, 2.36584238, 3.82889513]),
 'mean_score_time': array([0.00839829, 0.02419519, 0.04160371, 0.05639615, 0.06599679]),
 'std_score_time': array([0.00185254, 0.00116341, 0.00135428, 0.00531159, 0.00532947]),
 'param_n_estimators': masked_array(data=[3, 53, 103, 153, 203],
              mask=[False, False, False, False, False],
        fill_value='?',
             dtype=object),
 'params': [{'n_estimators': 3},
  {'n_estimators': 53},
  {'n_estimators': 103},
  {'n_estimators': 153},
  {'n_estimators': 203}],
 'split0_test_score': array([0.22720478, 0.24364723, 0.24962631, 0.25261584, 0.2406577 ]),
 'split1_test_score': array([0.23802395, 0.24850299, 0.2260479 , 0.23203593, 0.23353293]),
 'split2_test_score': array([0.24251497, 0.25      , 0.24101796, 0.22904192, 0.21107784]),
 'split3_test_score': array([0.20508982, 0.24251497, 0.24101796, 0.2245509 , 0.20958084]),
 'split4_test_score': array([0.23652695, 0.24401198, 0.2260479 , 0.2245509 , 0.2245509 ]),
 'mean_test_score': array([0.22987209, 0.24573543, 0.23675161, 0.2325591 , 0.22388004]),
 'std_test_score': array([0.01335558, 0.00295121, 0.00928763, 0.01042286, 0.01219411]),
 'rank_test_score': array([4, 1, 2, 3, 5])}
```

The best hyperparameters for the gradient boost tree are:

In [ ]:
```
abalone_lda_gtboosting_cv.best_params_
```

Out[ ]:
```
{'n_estimators': 53}
```

The score achieved by the best hyperparameters is:

In [ ]:
```
abalone_lda_gtboosting_cv.best_score_
```

Out[ ]:
```
0.24573543495967706
```

Now we will initialize a new gradient tree and train it with the best hyperparameters that were discovered by GridSearchCV

In [ ]:
```
# initialize the decision tree classifier with the best hyperparemeter
abalone_lda_gtboosting = GradientBoostingClassifier(random_state=27, n_estimators= 53)

# train the classifier using the training dataset
abalone_lda_gtboosting.fit(abalone_LDAminmax_train , abalone_rings_LDAminmax_train)

# view the accuracy of the classifier using the testing dataset
abalone_lda_gtboosting_test_accuracy = abalone_lda_gtboosting.score(abalone_LDAminmax_test ,
abalone_lda_gtboosting_test_accuracy
```

Out[ ]:
```
0.037081339712918660
```

5.1.2 Wine Dataset

5.1.2.1 Raw Data

In [ ]:
```
# create an array for the number of estimators for the classifier parameters
wine_gtboost_no_estimators = list(range(3,204,50))

# generate a dictionary for the parameters of random forest classifier
parameters = {"n_estimators": wine_gtboost_no_estimators}

# initialize the GridSearchCV function
wine_raw_gtboosting_cv = GridSearchCV(GradientBoostingClassifier(random_state=27), parameters

# fit the training data using the GridSearchCV function
wine_raw_gtboosting_cv.fit(wine_raw_train, wine_quality_raw_train)

# display the results of the tuning of the hyperprameters
wine_raw_gtboosting_cv.cv_results_
```

Out[ ]:
```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\model_select
ion\_split.py:676: UserWarning: The least populated class in y has only 4 members, which is l
ess than n_splits=5.
  warnings.warn(
{'mean_fit_time': array([ 0.26699967,  4.67379928,  9.05959897, 18.33676858, 18.09918437]),
 'std_fit_time': array([3.89882926e-03, 8.36532404e-02, 1.47475649e-01, 4.14813475e+00,
        9.32012904e-01]),
 'mean_score_time': array([0.00459986, 0.01260052, 0.01860251, 0.0360003 , 0.03239989]),
 'std_score_time': array([0.00048876, 0.00049017, 0.00048905, 0.01269624, 0.00102194]),
 'param_n_estimators': masked_array(data=[3, 53, 103, 153, 203],
              mask=[False, False, False, False, False],
        fill_value='?',
             dtype=object),
 'params': [{'n_estimators': 3},
  {'n_estimators': 53},
  {'n_estimators': 103},
  {'n_estimators': 153},
  {'n_estimators': 203}],
 'split0_test_score': array([0.53653846, 0.5875    , 0.58846154, 0.59903846, 0.60961538]),
 'split1_test_score': array([0.52403846, 0.58076923, 0.59326923, 0.59711538, 0.60384615]),
 'split2_test_score': array([0.51491819, 0.57844081, 0.59287777, 0.61116458, 0.61790183]),
 'split3_test_score': array([0.52165544, 0.56015399, 0.56785371, 0.57651588, 0.57747834]),
 'split4_test_score': array([0.52454283, 0.56111646, 0.56977863, 0.58325313, 0.58325313]),
 'mean_test_score': array([0.52433868, 0.5735961 , 0.58244818, 0.59341749, 0.59841897]),
 'std_test_score': array([0.00699874, 0.01099705, 0.01127429, 0.01224098, 0.0155107 ]),
 'rank_test_score': array([5, 4, 3, 2, 1])}
```

The best hyperparameters for the gradient boost tree are:

In [ ]:
```
wine_raw_gtboosting_cv.best_params_
```

Out[ ]:
```
{'n_estimators': 203}
```


The score achieved by the best hyperparameters is:

In [ ]:
```
wine_raw_gtboosting_cv.best_score_
```

Out[ ]:
```
0.5984189679425482
```

Now we will initialize a new gradient tree and train it with the best hyperparameters that were discovered by GridSearchCV

In [ ]:
```
# initialize the decision tree classifier with the best hyperparemeter
wine_raw_gtboosting = GradientBoostingClassifier(random_state=27, n_estimators= 203)

# train the classifier using the training dataset
wine_raw_gtboosting.fit(wine_raw_train, wine_quality_raw_train)

# view the accuracy of the classifier using the testing dataset
wine_raw_gtboosting_test_accuracy = wine_raw_gtboosting.score(wine_raw_test, wine_quality_ra
wine_raw_gtboosting_test_accuracy
```

Out[ ]:
```
0.6076923076923076
```

5.1.1.2 PCA Data

In [ ]:
```
# create an array for the number of estimators for the classifier parameters
wine_gtboost_no_estimators = list(range(3,204,50))

# generate a dictionary for the parameters of random forest classifier
parameters = {"n_estimators": wine_gtboost_no_estimators}

# initialize the GridSearchCV function
wine_pca_gtboosting_cv = GridSearchCV(GradientBoostingClassifier(random_state=27), parameter

# fit the training data using the GridSearchCV function
wine_pca_gtboosting_cv.fit(wine_PCAminmax_train , wine_quality_PCAminmax_train)

# display the results of the tuning of the hyperprameters
wine_pca_gtboosting_cv.cv_results_
```

Out[ ]:
```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\model_select
ion\_split.py:676: UserWarning: The least populated class in y has only 4 members, which is l
ess than n_splits=5.
  warnings.warn(
{'mean_fit_time': array([ 0.33059883,  6.25639644, 16.01319847, 16.93199792, 23.06800275]),
 'std_fit_time': array([0.00392735, 0.86480327, 2.73605996, 0.55077099, 0.98214995]),
 'mean_score_time': array([0.00480127, 0.01400294, 0.02740345, 0.02380185, 0.02960081]),
 'std_score_time': array([0.00040078, 0.00502122, 0.00715275, 0.00040273, 0.00080047]),
 'param_n_estimators': masked_array(data=[3, 53, 103, 153, 203],
              mask=[False, False, False, False, False],
        fill_value='?',
             dtype=object),
 'params': [{'n_estimators': 3},
  {'n_estimators': 53},
  {'n_estimators': 103},
  {'n_estimators': 153},
  {'n_estimators': 203}],
 'split0_test_score': array([0.50096154, 0.56057692, 0.56442308, 0.57692308, 0.58076923]),
 'split1_test_score': array([0.47692308, 0.58557692, 0.59326923, 0.59807692, 0.60576923]),
 'split2_test_score': array([0.48893167, 0.56111646, 0.59287777, 0.60635226, 0.6053898 ]),
 'split3_test_score': array([0.49278152, 0.54379211, 0.55822907, 0.56977863, 0.56881617]),
 'split4_test_score': array([0.48026949, 0.55630414, 0.55919153, 0.56496631, 0.57170356]),
 'mean_test_score': array([0.48797346, 0.56147331, 0.57359813, 0.58321944, 0.5864896 ]),
 'std_test_score': array([0.00865087, 0.01357395, 0.01604117, 0.01618239, 0.01607869]),
 'rank_test_score': array([5, 4, 3, 2, 1])}
```

The best hyperparameters for the gradient boost tree are:

In [ ]:

```
wine_pca_gtboosting_cv.best_params_
```

Out[ ]:
```
{'n_estimators': 203}
```

The score achieved by the best hyperparameters is:

In [ ]:
```
wine_pca_gtboosting_cv.best_score_
```

Out[ ]:
```
0.5864895979862295
```

Now we will initialize a new gradient tree and train it with the best hyperparameters that were discovered by GridSearchCV

In [ ]:
```
# initialize the decision tree classifier with the best hyperparemeter
wine_pca_gtboosting = GradientBoostingClassifier(random_state=27, n_estimators= 203)

# train the classifier using the training dataset
wine_pca_gtboosting.fit(wine_PCAminmax_train , wine_quality_PCAminmax_train)

# view the accuracy of the classifier using the testing dataset
wine_pca_gtboosting_test_accuracy = wine_pca_gtboosting.score(wine_PCAminmax_test , wine_qua
wine_pca_gtboosting_test_accuracy
```

Out[ ]:
```
0.4330769230769231
```

5.1.1.3 LDA Data

In [ ]:
```
# create an array for the number of estimators for the classifier parameters
wine_gtboost_no_estimators = list(range(3,204,50))

# generate a dictionary for the parameters of random forest classifier
parameters = {"n_estimators": wine_gtboost_no_estimators}

# initialize the GridSearchCV function
wine_lda_gtboosting_cv = GridSearchCV(GradientBoostingClassifier(random_state=27), parameter

# fit the training data using the GridSearchCV function
wine_lda_gtboosting_cv.fit(wine_LDAminmax_train , wine_quality_LDAminmax_train)

# display the results of the tuning of the hyperprameters
wine_lda_gtboosting_cv.cv_results_
```

```
C:\Users\ibteh\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\model_select
ion\_split.py:676: UserWarning: The least populated class in y has only 4 members, which is l
ess than n_splits=5.
  warnings.warn(
```

Out[ ]:
```
{'mean_fit_time': array([ 0.2161974 ,  3.61499987,  9.46694598, 11.7002337 , 15.74487972]),
 'std_fit_time': array([0.02179355, 0.14403067, 1.8320693 , 1.10282268, 1.38919657]),
 'mean_score_time': array([0.00500169, 0.01100082, 0.02271791, 0.02440009, 0.03719668]),
 'std_score_time': array([6.33544062e-04, 7.29420592e-07, 6.51653992e-03, 1.74637205e-03,
        9.92647047e-03]),
 'param_n_estimators': masked_array(data=[3, 53, 103, 153, 203],
              mask=[False, False, False, False, False],
        fill_value='?',
             dtype=object),
 'params': [{'n_estimators': 3},
  {'n_estimators': 53},
  {'n_estimators': 103},
  {'n_estimators': 153},
  {'n_estimators': 203}],
 'split0_test_score': array([0.52596154, 0.54423077, 0.54903846, 0.56538462, 0.57788462]),
 'split1_test_score': array([0.52307692, 0.55865385, 0.56730769, 0.56153846, 0.56442308]),
 'split2_test_score': array([0.5120308 , 0.54956689, 0.55919153, 0.56496631, 0.57651588]),
 'split3_test_score': array([0.51973051, 0.55149182, 0.56304139, 0.57940327, 0.58421559]),
 'split4_test_score': array([0.52454283, 0.54475457, 0.55437921, 0.56496631, 0.56592878]),
 'mean_test_score': array([0.52106852, 0.54973958, 0.55859166, 0.5672518 , 0.57379359]),
 'std_test_score': array([0.00497025, 0.00524903, 0.00640527, 0.00623272, 0.00751568]),
 'rank_test_score': array([5, 4, 3, 2, 1])}
```

The best hyperparameters for the gradient boost tree are:

In [ ]:
```
wine_lda_gtboosting_cv.best_params_
```

Out[ ]:
```
{'n_estimators': 203}
```

The score achieved by the best hyperparameters is:

In [ ]:
```
wine_lda_gtboosting_cv.best_score_
```

Out[ ]:
```
0.5737935885096617
```

Now we will initialize a new gradient tree and train it with the best hyperparameters that were discovered by GridSearchCV

In [ ]:
```
# initialize the decision tree classifier with the best hyperparemeter
wine_lda_gtboosting = GradientBoostingClassifier(random_state=27, n_estimators= 203)

# train the classifier using the training dataset
wine_lda_gtboosting.fit(wine_LDAminmax_train , wine_quality_LDAminmax_train)

# view the accuracy of the classifier using the testing dataset
wine_lda_gtboosting_test_accuracy = wine_lda_gtboosting.score(wine_LDAminmax_test , wine_qua
wine_lda_gtboosting_test_accuracy
```

Out[ ]:
```
0.4023076923076923
```

5.2 Plot of Mean Accuracy

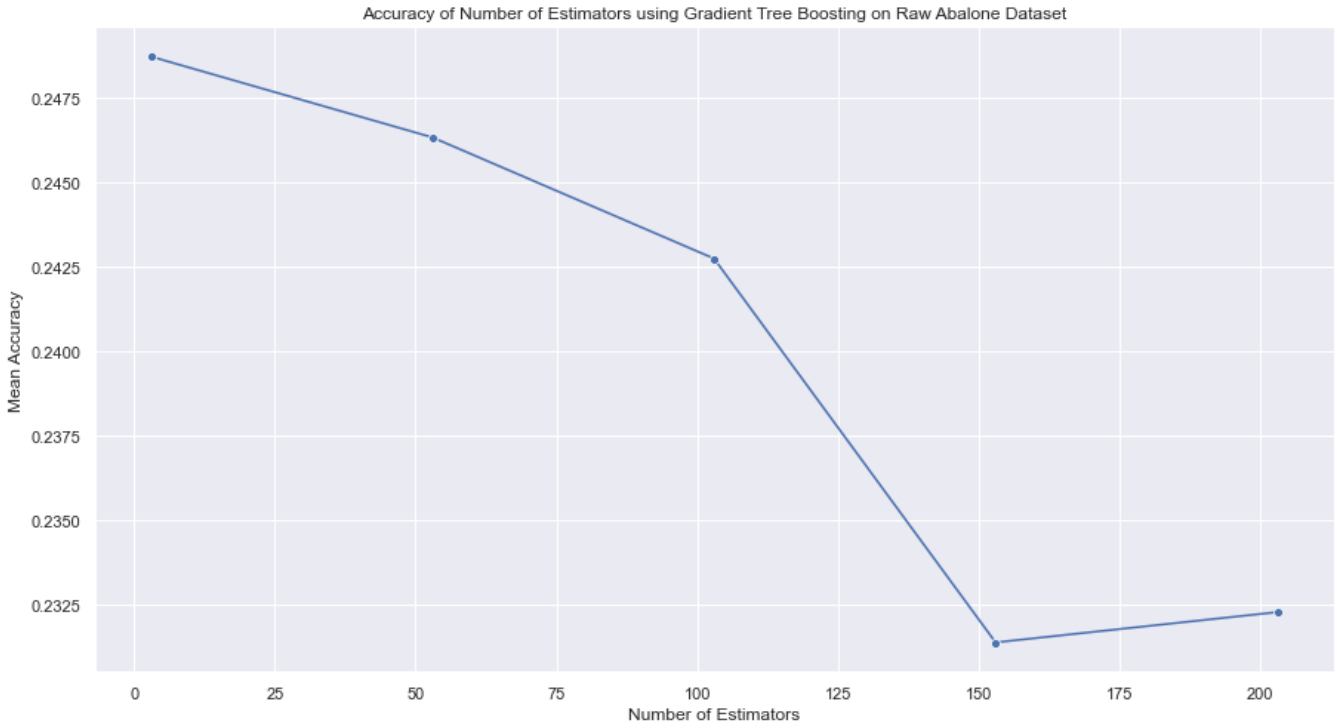
5.2.1 Abalone Dataset

Raw Dataset

In [ ]:
```
# generate a dataframe of the maximum depth and the respective score
abalone_raw_gtboosting_cv_results = pd.DataFrame([[3, 53, 103, 153, 203],abalone_raw_gtboost

# transpose the dataframe
abalone_raw_gtboosting_cv_results = abalone_raw_gtboosting_cv_results.T

# name the columns of the dataframe
olumns = ["Number of Estimators","Mean Accuracy"]
```

In [ ]:
```
# initialize the plot
sns.set(rc = {'figure.figsize':(15,8)})

# assign the datapoints for the axes
sns.lineplot(data= abalone_raw_gtboosting_cv_results, x="Number of Estimators", y="Mean Accu

# assign the graph title
plt.title("Accuracy of Number of Estimators using Gradient Tree Boosting on Raw Abalone Datas
```

Out[ ]:
```
Text(0.5, 1.0, 'Accuracy of Number of Estimators using Gradient Tree Boosting on Raw Abalone
Dataset')
```

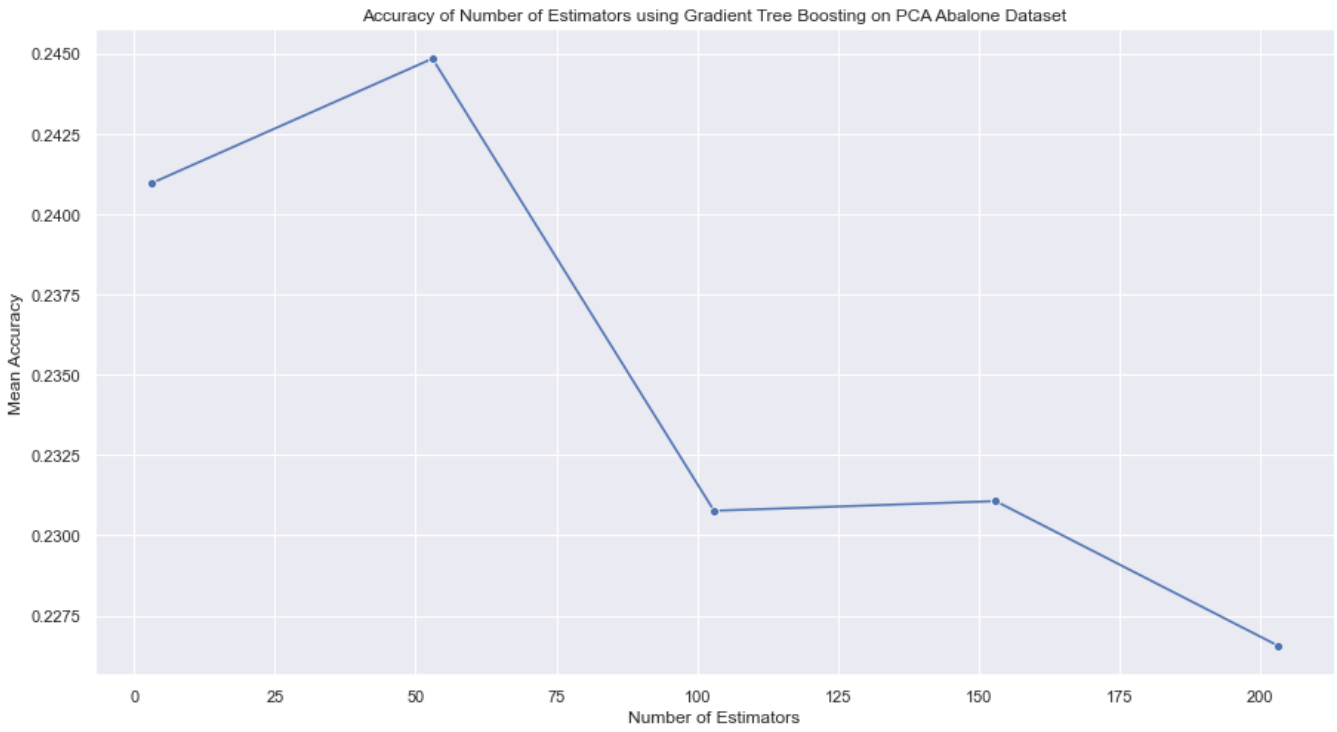


PCA Dataset

In [ ]:
```
# generate a dataframe of the maximum depth and the respective score
abalone_pca_gtboosting_cv_results = pd.DataFrame([[3, 53, 103, 153, 203],abalone_pca_gtboost

# transpose the dataframe
abalone_pca_gtboosting_cv_results = abalone_pca_gtboosting_cv_results.T

# name the columns of the dataframe
abalone_pca_gtboosting_cv_results.columns = ["Number of Estimators","Mean Accuracy"]
```

In [ ]:
```
# initialize the plot
sns.set(rc = {'figure.figsize':(15,8)})

# assign the datapoints for the axes
sns.lineplot(data= abalone_pca_gtboosting_cv_results, x="Number of Estimators", y="Mean Accu

# assign the graph title
plt.title("Accuracy of Number of Estimators using Gradient Tree Boosting on PCA Abalone Datas
```

Out[ ]:
```
Text(0.5, 1.0, 'Accuracy of Number of Estimators using Gradient Tree Boosting on PCA Abalone
Dataset')
```

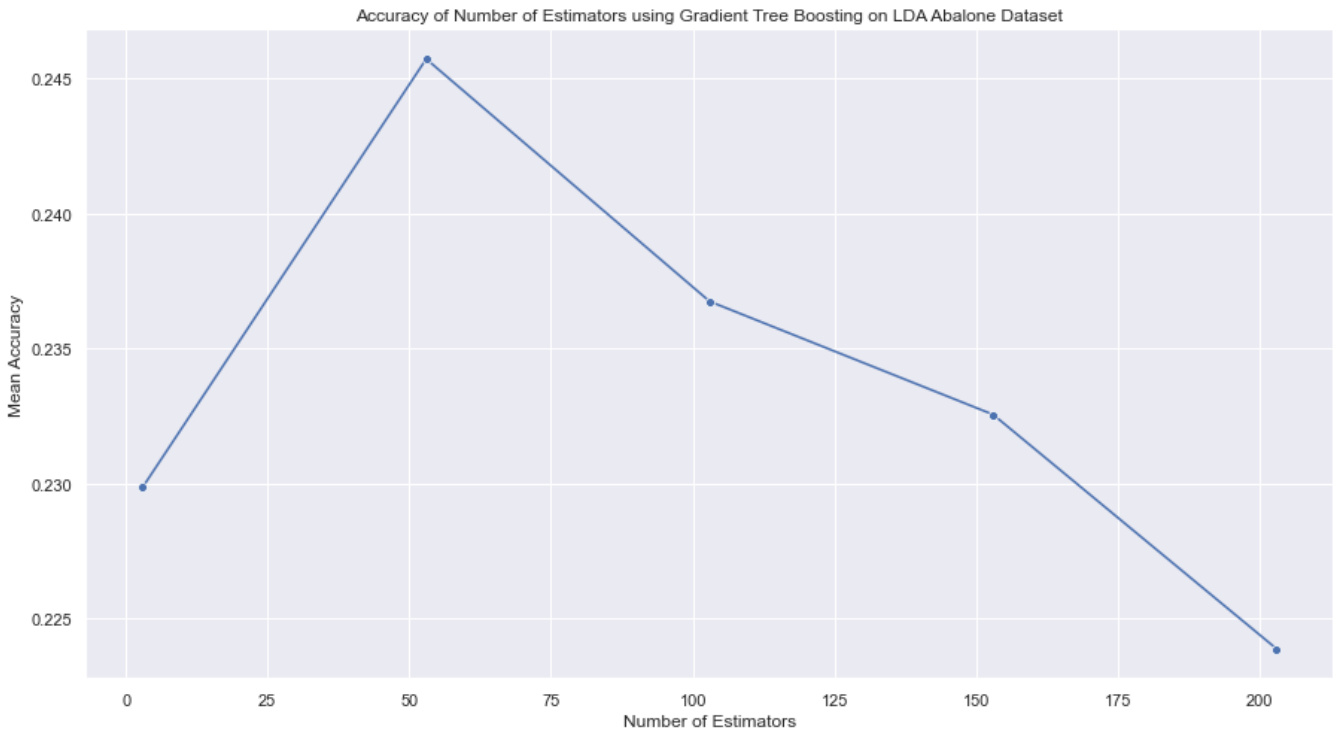


LDA Dataset

In [ ]:
```
# generate a dataframe of the maximum depth and the respective score
abalone_lda_gtboosting_cv_results = pd.DataFrame([[3, 53, 103, 153, 203],abalone_lda_gtboost

# transpose the dataframe
abalone_lda_gtboosting_cv_results = abalone_lda_gtboosting_cv_results.T

# name the columns of the dataframe
abalone_lda_gtboosting_cv_results.columns = ["Number of Estimators","Mean Accuracy"]
```

In [ ]:
```
# initialize the plot
sns.set(rc = {'figure.figsize':(15,8)})

# assign the datapoints for the axes
sns.lineplot(data= abalone_lda_gtboosting_cv_results, x="Number of Estimators", y="Mean Accu

# assign the graph title
plt.title("Accuracy of Number of Estimators using Gradient Tree Boosting on LDA Abalone Datas
```

Out[ ]:
```
Text(0.5, 1.0, 'Accuracy of Number of Estimators using Gradient Tree Boosting on LDA Abalone
Dataset')
```



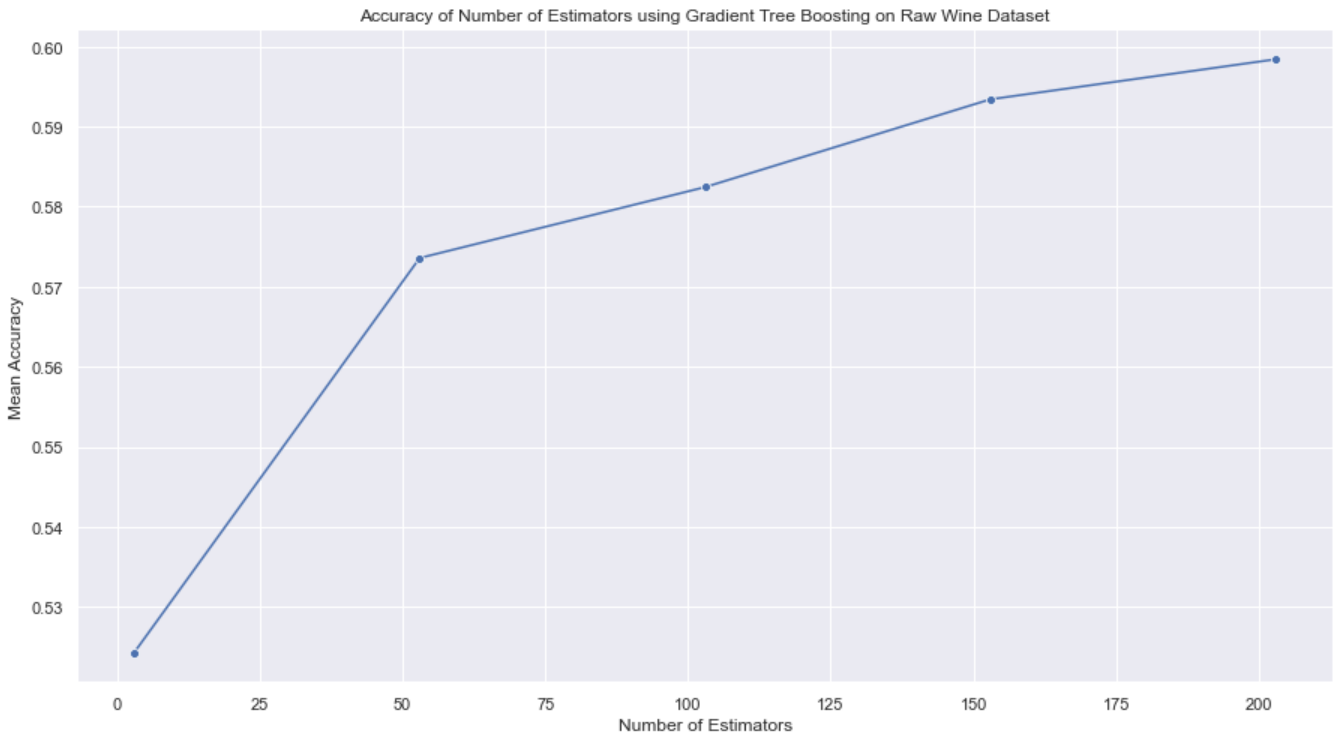
5.2.2 Wine Dataset

Raw Dataset

In [ ]:
```
# generate a dataframe of the maximum depth and the respective score
wine_raw_gtboosting_cv_results = pd.DataFrame([[3, 53, 103, 153, 203],wine_raw_gtboosting_cv

# transpose the dataframe
wine_raw_gtboosting_cv_results = wine_raw_gtboosting_cv_results.T

# name the columns of the dataframe
wine_raw_gtboosting_cv_results.columns = ["Number of Estimators","Mean Accuracy"]
```

In [ ]:
```
# initialize the plot
sns.set(rc = {'figure.figsize':(15,8)})

# assign the datapoints for the axes
sns.lineplot(data= wine_raw_gtboosting_cv_results, x="Number of Estimators", y="Mean Accuracy

# assign the graph title
plt.title("Accuracy of Number of Estimators using Gradient Tree Boosting on Raw Wine Dataset
```

Out[ ]:
```
Text(0.5, 1.0, 'Accuracy of Number of Estimators using Gradient Tree Boosting on Raw Wine Dat
aset')
```

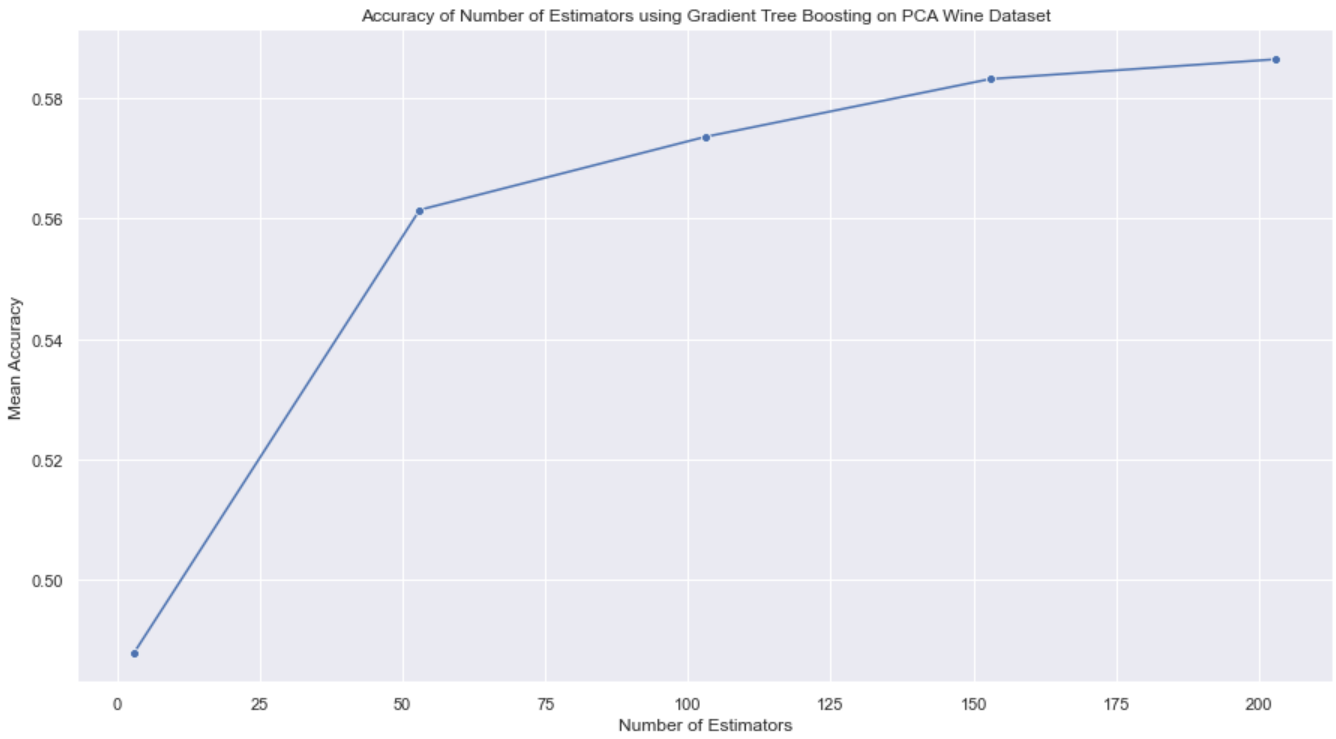


PCA Dataset

In [ ]:
```
# generate a dataframe of the maximum depth and the respective score
wine_pca_gtboosting_cv_results = pd.DataFrame([[3, 53, 103, 153, 203],wine_pca_gtboosting_cv

# transpose the dataframe
wine_pca_gtboosting_cv_results = wine_pca_gtboosting_cv_results.T

# name the columns of the dataframe
wine_pca_gtboosting_cv_results.columns = ["Number of Estimators","Mean Accuracy"]
```

In [ ]:
```
# initialize the plot
sns.set(rc = {'figure.figsize':(15,8)})

# assign the datapoints for the axes
sns.lineplot(data= wine_pca_gtboosting_cv_results, x="Number of Estimators", y="Mean Accuracy

# assign the graph title
plt.title("Accuracy of Number of Estimators using Gradient Tree Boosting on PCA Wine Dataset
```

Out[ ]:
```
Text(0.5, 1.0, 'Accuracy of Number of Estimators using Gradient Tree Boosting on PCA Wine Dat
aset')
```

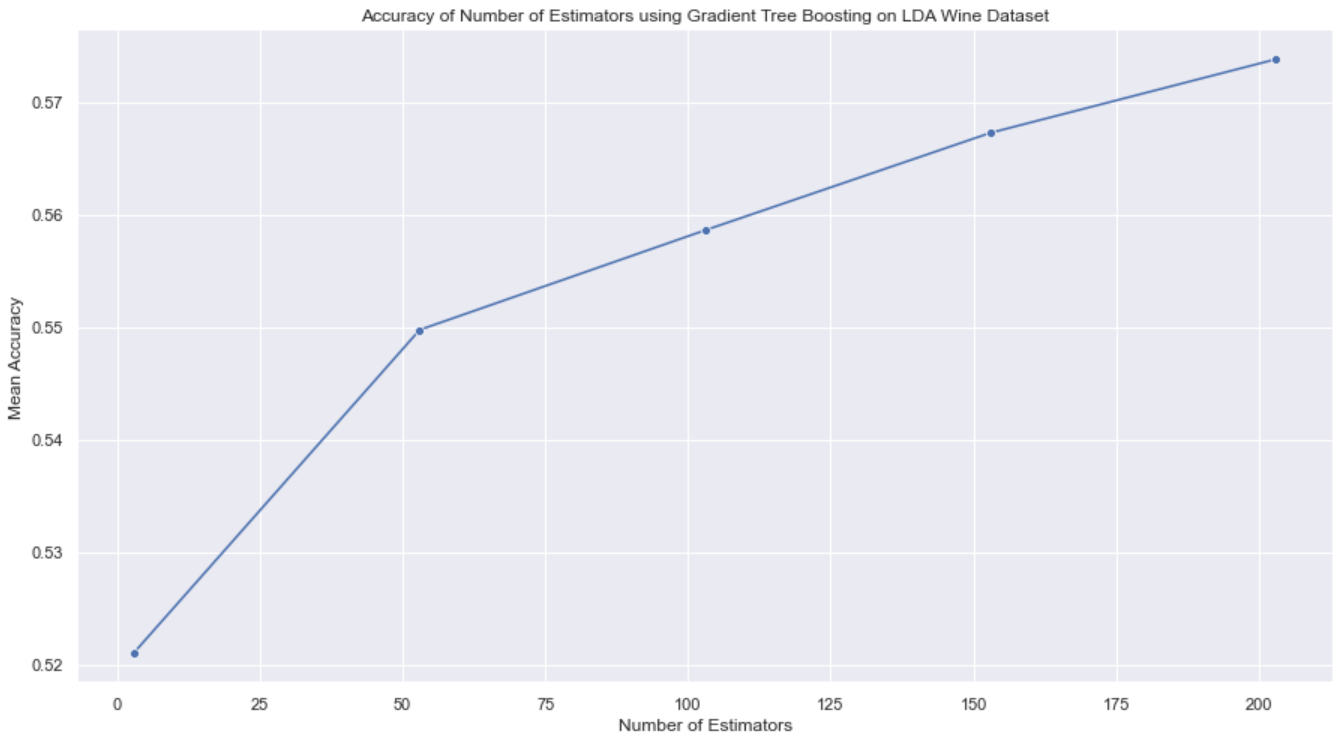


LDA Dataset

In [ ]:
```
# generate a dataframe of the maximum depth and the respective score
wine_lda_gtboosting_cv_results = pd.DataFrame([[3, 53, 103, 153, 203],wine_lda_gtboosting_cv

# transpose the dataframe
wine_lda_gtboosting_cv_results = wine_lda_gtboosting_cv_results.T

# name the columns of the dataframe
wine_lda_gtboosting_cv_results.columns = ["Number of Estimators","Mean Accuracy"]
```

In [ ]:
```
# initialize the plot
sns.set(rc = {'figure.figsize':(15,8)})

# assign the datapoints for the axes
sns.lineplot(data= wine_lda_gtboosting_cv_results, x="Number of Estimators", y="Mean Accuracy

# assign the graph title
plt.title("Accuracy of Number of Estimators using Gradient Tree Boosting on LDA Wine Dataset
```

Out[ ]:
```
Text(0.5, 1.0, 'Accuracy of Number of Estimators using Gradient Tree Boosting on LDA Wine Dat
aset')
```



Question 6: Finial Results

6.1 Comments on Pipe line

6.1.1 Abalone Dataset

The best pipeline for the abalone dataset is by using the raw dataset with the random forest classifier based on the testing accuracy

6.1.2 Wine Dataset

The best pipeline for the wine dataset is by using the raw dataset with the random forest classifier based on the testing accuracy.

6.2 Effect of Dimensionality Reduction

6.2.1 Abalone Dataset

kNN Classifier

For the kNN classifier, LDA has the best performance followed by raw dataset and PCA dataset. The difference among the three dataset is very small. As we are using all the components in PCA and LDA, there is no information loss.

Naive Bayes

For both the Naive Bayes calssifiers, the performance of raw dataset is better than that of PCA dataset and LDA datasets. The PCA dataset's performance is slightly less than that of raw dataset. The reason could be that the feature with the most discreminating power may have a small variance. Therefore, the effect of that feature in the classification is reduced.

The LDA dataset accuracy decreased significantly as the abalone dataset is not normally distributed. It may also be possible that LDA could not seperate the labels with use of the mean as the means may be very close.

Decision Tree

By applying PCA, the accuracy on the testing data did improve when compared to the raw dataset.

By applying LDA, the accuracy decreased significantly. This may be because the input features of the abalone dataset are not normally distributed and generalization error for the abalone dataset is high. It may also be possible that LDA could not seperate the labels with use of the mean as the means may be very close.

Random Forest

By applying PCA on the abalone dataset, the accuracy decreased when compared to the accuracy of using the raw data with random forest classifier. Although we selected the number of principal components with the highest accuracy for the dataset, when compared to raw data, there is enough data loss that is occurring to causes the raw dataset to outperform the PCA dataset.

Another reason for the raw dataset in outperforming the PCA dataset is that due to the computational requirements of the random forest algorithm, we didn't find the best hyperparameter that could have performed better than raw dataset.

By applying LDA, the accuracy decreased significantly. This may be because the input features of the abalone dataset are not normally distributed and generalization error for the abalone dataset is high.

Gradient Tree Boosting

By applying PCA and LDA on the abalone dataset, the accuracy decreased when compared to using raw data with gradient tree boosting classifier.

The reason for the raw dataset in outperforming the PCA dataset is that due to the computational requirements of the random forest algorithm, we didn't find the best hyperparameter that could have performed better than raw dataset.This is supported by the fact that the gradient of the graph in 6.2.2 is positive as number of estimators is increasing.

6.2.2 Wine Dataset

kNN Classifier

The performance of PCA dataset is slightly higher than that of raw dataset and LDA dataset. As all the components were being used for both PCA and LDA, information loss was non-existant. Therefore, performance is similar.

Naive Bayes

With Multinomial Naive Bayes, all the testing accuracies were the same.

With Complement Naive Bayes, the raw dataset resulted in to the best accuracy followed by PCA. The testing accuracy of the LDA dataset decreased significantly as the dataset is not normally distributed. It may also be possible that LDA could not seperate the labels with use of the mean as the means may be very close.

Decision Tree

By applying PCA or LDA, the accuracy decreases when compared to that of the raw dataset.

Random Forest

By applying PCA and LDA on the abalone dataset, the accuracy decreased when compared to using raw data with random forest classifier. PCA had a higher accuracy than LDA.

Additionally, this does not mean PCA & LDA are bad for random forest for these datasets. As limited combination of hyperparameters were tried to find the optimum values, it is entirely possible that the best hyperparameter was not discovered.

Gradient Tree Boosting

By applying PCA and LDA on the abalone dataset, the accuracy decreased when compared to using raw data with gradient tree boosting classifier.

This however does not mean that PCA & LDA produce bad results. As limited combination of hyperparameters were tried to find the optimum values, it is entirely possible that the best hyperparameter was not discovered.

6.3 Additional Observations

In the case of both abalone and wine datasets, the accuracy of random forest is the best for the raw dataset.

For the case of PCA and LDA, kNN produced better test accuracy. The reason for this does not necessarily mean that kNN is better than random forest or gradient boosting for these datasets. The problem is that random forest and gradient tree boosting are computationally expensive. This resulted in trying less hyperparameters for both the classifier. This may have resulted in the optimum hyperparameter to not have been discovered. Therefore, more testing is required to conclude this issue.

This fact is confirmed by the accuracy graph that was generated in 5.2. The gradient of the slope is positive.

6.4 Result Tables

Since raw, PCA & LDA datasets were using different settings for the same classifiers, the tables have been seperated for ease of understanding

6.4.1 Abalone Dataset

In [ ]:

```
# initialize dataframe
final_abalone_table = pd.DataFrame()

# make the index column
final_abalone_table ["Method"]= ["kNN", "Multinomial NB", "Complement NB", "Decision Tree",
final_abalone_table.set_index("Method", inplace = True)

# add the relevant data
final_abalone_table ["Settings for Raw Dataset"]= ["k = 115 & Euclidean", "No hyperparameter
final_abalone_table ["Settings for PCA Dataset"]= ["k = 115, Euclidean, Comp = 3", "No hyper
final_abalone_table ["Settings for LDA Dataset"]= ["k = 115, Euclidean, Comp = 3", "No hyper
# abalone_summary_table["Raw (Mean Validation Accuracy)"] = [abalone_raw_mnb_train_cv_score,

# generate the column for the test accuracies of the respective classifier used
# kNN value for the raw dataset is copied from the previous assignment
final_abalone_table ["Test Accuracy of Abalone_raw"] = [0.2763,abalone_raw_mnb_cscore_testing
final_abalone_table ["Test Accuracy of Abalone_pca"] = [max(abalone_pca_knn_scores), abalone_
final_abalone_table ["Test Accuracy of Abalone_lda"] = [max(abalone_lda_knn_scores), abalone_

# print the table
final_abalone_table
```

Out[ ]:

Method	Settings for Raw Dataset	Settings for PCA Dataset	Settings for LDA Dataset	Test Accuracy of Abalone_raw	Test Accuracy of Abalone_pca	Test Accuracy of Abalone_lda
kNN	k = 115 & Euclidean	k = 115, Euclidean, Comp = 3	k = 115, Euclidean, Comp = 3	0.276300	0.270335	0.287081
Multinomial NB	No hyperparameter	No hyperparameter	No hyperparameter	0.157895	0.157895	0.157895
Complement NB	No hyperparameter	No hyperparameter	No hyperparameter	0.167464	0.161483	0.071770
Decision Tree	{'max_depth': 5}	{'max_depth': 4}	{'max_depth': 4}	0.242823	0.264354	0.020335
Random Forest	{'max_depth': 7, 'n_estimators': 303}	{'max_depth': 7, 'n_estimators': 153}	{'max_depth': 7, 'n_estimators': 203}	0.277512	0.261962	0.034689
Gradient Boosting Tree	{'n_estimators': 3}	{'n_estimators': 53}	{'n_estimators': 53}	0.253589	0.239234	0.037081

6.4.2 Wine Dataset

In [ ]:

```
# initialize dataframe
final_wine_table = pd.DataFrame()

# make the index column
final_wine_table ["Method"]= ["kNN", "Multinomial NB", "Complement NB", "Decision Tree", "Ra
final_wine_table.set_index("Method", inplace = True)

# add the relevant data
final_wine_table ["Settings for Raw Dataset"]= ["k = 124 & Manhattan", "No hyperparameter",
final_wine_table ["Settings for PCA Dataset"]= ["k = 124, Manhattan, Comp = 3", "No hyperpara
final_wine_table ["Settings for LDA Dataset"]= ["k = 124, Manhattan, Comp = 3", "No hyperpara
# wine_summary_table["Raw (Mean Validation Accuracy)"] = [wine_raw_mnb_train_cv_score,wine_r

# generate the column for the test accuracies of the respective dataset for each classifier
# kNN value for the raw dataset is copied from the previous assignment
final_wine_table ["Test Accuracy of wine_raw"] = [0.68230,wine_raw_mnb_cscore_testing,wine_r
final_wine_table ["Test Accuracy of wine_pca"] = [max(wine_pca_knn_scores), wine_pca_mnb_csc
final_wine_table ["Test Accuracy of wine_lda"] = [max(wine_lda_knn_scores), wine_lda_mnb_csc

# print the table
final_wine_table
```

Out[ ]:

Method	Settings for Raw Dataset	Settings for PCA Dataset	Settings for LDA Dataset	Test Accuracy of wine_raw	Test Accuracy of wine_pca	Test Accuracy of wine_lda
kNN	k = 124 & Manhattan	k = 124, Manhattan, Comp = 3	k = 124, Manhattan, Comp = 3	0.682300	0.683846	0.674615
Multinomial NB	No hyperparameter	No hyperparameter	No hyperparameter	0.438462	0.438462	0.438462
Complement NB	No hyperparameter	No hyperparameter	No hyperparameter	0.427692	0.366154	0.386154
Decision Tree	{'max_depth': 26}	{'max_depth': 25}	{'max_depth': 25}	0.610000	0.391538	0.446923
Random Forest	{'max_depth': 27, 'n_estimators': 303}	{'max_depth': 17, 'n_estimators': 253}	{'max_depth': 17, 'n_estimators': 303}	0.693077	0.500000	0.453077
Gradient Boosting Tree	{'n_estimators': 203}	{'n_estimators': 203}	{'n_estimators': 203}	0.607692	0.433077	0.402308

References

1. **PCA using Python (scikit-learn)**, by Michael Galarnyk, *Towards Data Science*, 2017. Click **here**
2. **Seaborn: statistical data visualization**, by Michael L. Waskom, *Journal of Open Source Software*, 2021. Click **here**
3. **PCA Background**, by Gellert Toth, *DataSklr*, 2021. Click **here**
4. **t-SNE Python Example**, by Cory Maklin, *Towards Data Science*, 2019. Click **here**
5. **Find Maximum Tree Depth**, *Stack Overflow*, 2019. Click **here**
6. **PCA Is Not Feature Selection**, by Brandon Walker, *Towards Data Science*, 2019. Click **here**
7. **Scikit-learn: Machine Learning in Python**, by Pedregosa et al., *JMLR* 12, pp. 2825-2830, 2011. Click **here**
8. **Decision Tree Classifier in Python Sklearn with Example**, by Afham Fardeen, *Machine Learning Knowledge*, 2021. Click **here**
9. **Gradient Boosting with Scikit-Learn, XGBoost, LightGBM, and CatBoost**, by Jason Brownlee, *Machine Learning Mastery*, 2020. Click **here**
10. **GridSearchCV for Beginners**, by Scott Okamura, *Towards Data Science*, 2020. Click **here**
11. **Complete guide to Python's cross-validation with examples**, by Vaclav Dekanovsky, *Towards Data Science*, 2020. Click **here**
12. **Linear Discriminant Analysis With Python**, by Jason Brownlee, *Machine Learning Mastery*, 2020. Click **here**
13. **Decision Trees in Python with Scikit-Learn**, by Scott Robinson, *Stack Abuse*. Click **here**

heatsheet, by Inge Halilovic, *Medium*, 2017. Click **here**

15. **Complement Naive Bayes (CNB) Algorithm**, by Alakesh Bora, *Geeks for Geeks*, 2020. Click **here**