**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| **Ex.NO** | **DATE** | **EXPERIMENT NAME** | **PAGE NO.** |
| **1.** |  | **Implement Decision Tree learning** | 3-6 |
| **2.** |  | **Implement Logistic Regression** | 7-10 |
| **3.** |  | **Implement classification using Multilayer perceptron** | 11-16 |
| **4.** |  | **Implement classification using SVM** | 17-21 |
| **5.** |  | **Implement Adaboost Algorithm** | 22-25 |
| **6.** |  | **Implement Bagging using Random Forests** | 26-29 |
| **7.** |  | **Implement K-means Clustering to Find Natural Patterns in Data** | 30-33 |
| **8.** |  | **Implement Principle Component Analysis for Dimensionality Reduction** | 34-37 |
| **9.** |  | **Evaluating ML algorithm with balanced and unbalanced datasets** | 38-44 |
| **10.** |  | **Comparison of Machine Learning algorithms** | 45-49 |

|  |  |
| --- | --- |
| **EXP.NO: 01** | **Implementation of decision tree learning using sklearn breast** |
| **DATE: 23/09/23** |

**AIM**

Implementation of decision tree learning using sklearn breast cancer data set.

**PROCEDURE**

Preprocessing of Data:

* Load the Breast Cancer dataset using load\_breast\_cancer from sklearn.datasets.
* Address any missing values (if any, though this dataset is usually clean and well-organized).
* Since the dataset contains numerical features, scaling might not be necessary for Decision Tree models, but you can perform it if desired.
* Splitting the Data:
* Use train\_test\_split from sklearn.model\_selection to divide the dataset into training and testing sets. Set an appropriate test size (e.g., 20%).
* Training the Decision Tree Model:
* Initialize a Decision Tree classifier using DecisionTreeClassifier from sklearn.tree.
* Specify hyperparameters like criterion (e.g., 'gini' or 'entropy'), maximum depth, or minimum samples split as needed.
* Train the Decision Tree model using the training set of data.
* Make Predictions:
* Use the trained Decision Tree model to make predictions based on the testing data.
* Evaluate the Model:
* Utilize accuracy\_score and classification\_report from sklearn.metrics to determine the accuracy and other relevant metrics.
* Interpret the classification report to understand precision, recall, and F1-score for each class.

**PROGRAM**

from sklearn import datasets

from sklearn.model\_selection import train\_test\_split

from sklearn.tree import DecisionTreeClassifier, plot\_tree

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

import matplotlib.pyplot as plt

# Load the cancer dataset

from sklearn.datasets import load\_breast\_cancer

cancer = load\_breast\_cancer()

X = cancer.data

y = cancer.target

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# Create a Decision Tree Classifier

clf = DecisionTreeClassifier()

# Fit the classifier to the training data

clf.fit(X\_train, y\_train)

# Make predictions on the test data

y\_pred = clf.predict(X\_test)

# Calculate accuracy

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Accuracy: {accuracy:.2f}")

print(accuracy\*100,"%")

# Generate a classification report

report = classification\_report(y\_test, y\_pred)

print("Classification Report:")

print(report)

# Generate a confusion matrix

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

print("Confusion Matrix:")

print(conf\_matrix)

from sklearn.tree import DecisionTreeClassifier

treemodel=DecisionTreeClassifier()

treemodel.fit(X\_train,y\_train)

from sklearn import tree

plt.figure(figsize=(15,10))

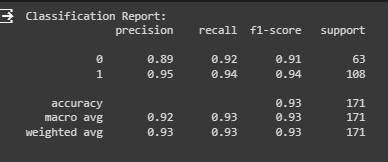
tree.plot\_tree(treemodel,filled=True)

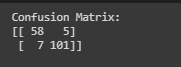
**INPUT**

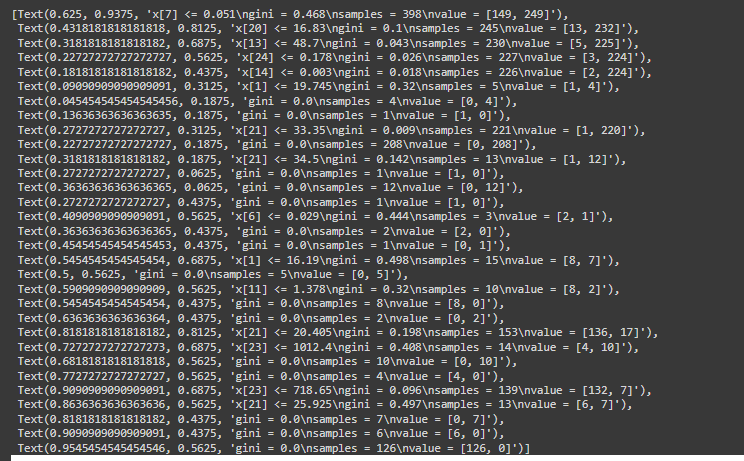
Breast Cancer dataset

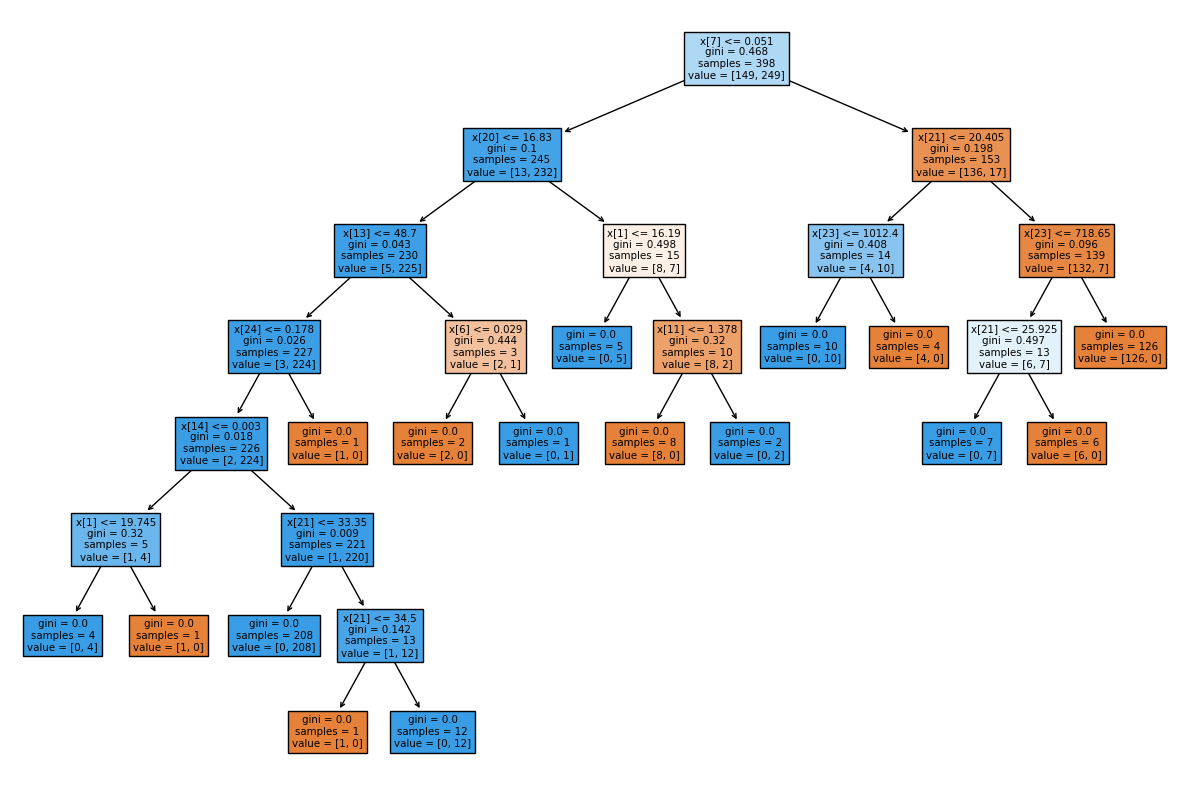
**OUTPUT**

****

****

****

****



**RESULT**

The accuracy for the model is: **93%** using **Decision Tree Classification.**

|  |  |
| --- | --- |
| **EXP.NO: 03** | **Implementation of logistic regression** |
| **DATE: 14/9/23** |

**AIM**

Implementation of logistic regression on breast cancer prediction.

**PROCEDURE**

* Load the breast cancer dataset using load\_breast\_cancer from sklearn.datasets.
* Address any missing values (if any, though this dataset is usually clean and well-organized).
* Since the dataset contains numerical features, scaling might be beneficial for logistic regression. Use StandardScaler from sklearn.preprocessing to standardize the features.
* Splitting the Data:
* Use train\_test\_split from sklearn.model\_selection to divide the dataset into training and testing sets. Set an appropriate test size (e.g., 20%).
* Scaling Features:
* Utilize StandardScaler from sklearn.preprocessing to standardize the features. Scaling is often important for logistic regression models.
* Training the Logistic Regression Model:
* Initialize a Logistic Regression classifier using LogisticRegression from sklearn.linear\_model.
* Using the training set of data, fit the logistic regression model.
* Make Predictions:
* Make predictions based on the testing data using the trained logistic regression model.
* Evaluate the Model:
* Utilize accuracy\_score and classification\_report from sklearn.metrics to determine the accuracy and other pertinent metrics.
* Interpret the classification report to understand precision, recall, and F1-score for each class.

**PROGRAM**

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

from sklearn import datasets

import numpy as np

from sklearn.datasets import load\_breast\_cancer

cancer = load\_breast\_cancer()

X = cancer.data

y = cancer.target

# Split data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# Create and train the logistic regression classifier

clf = LogisticRegression(max\_iter=1000)

clf.fit(X\_train, y\_train)

# Make predictions on the test data

y\_pred = clf.predict(X\_test)

# Calculate accuracy

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Accuracy: {accuracy:.2f}")

print(accuracy\*100,"%")

# Generate a classification report

report = classification\_report(y\_test, y\_pred)

print("Classification Report:")

print(report)

# Generate a confusion matrix

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

print("Confusion Matrix:")

print(conf\_matrix)

# Visualize the confusion matrix as a heatmap

plt.figure(figsize=(8, 6))

sns.heatmap(conf\_matrix, annot=True, fmt='g', xticklabels=cancer.target\_names,

yticklabels=cancer.target\_names, cmap='coolwarm')

plt.xlabel('Predicted')

plt.ylabel('True')

plt.title('Confusion Matrix')

plt.show()

# Visualize classes comparison

plt.figure(figsize=(8, 6))

sns.scatterplot(x=X\_test[:, 0], y=X\_test[:, 1], hue=cancer.target\_names[y\_pred], palette='coolwarm', s=100)

plt.xlabel('Feature 1')

plt.ylabel('Feature 2')

plt.title('Classes Comparison')

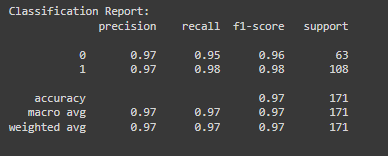
plt.show()

**INPUT**

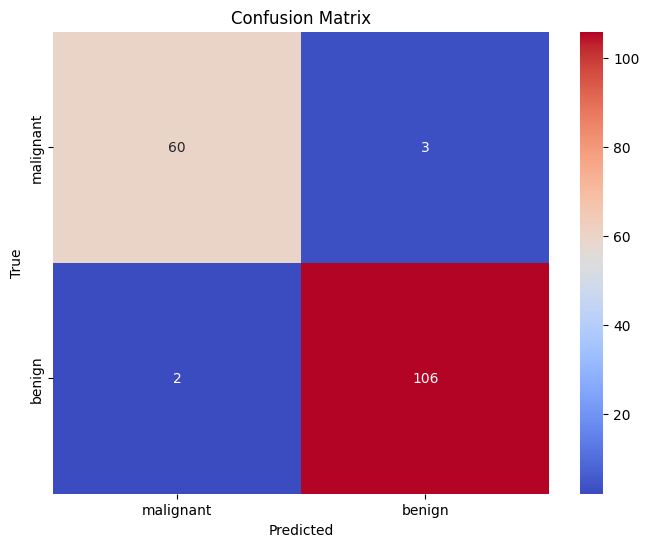
Breast Cancer dataset

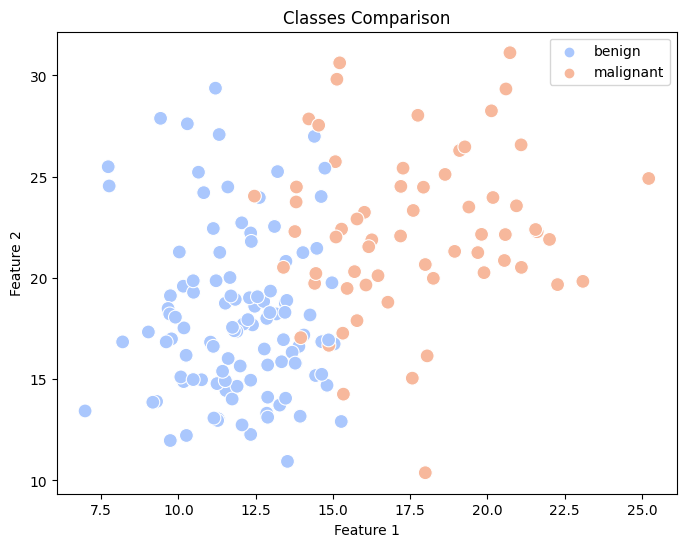
**OUTPUT**

****

****

****





**RESULT**

The accuracy for the model is: **97%** using **Logistic Classification.**

|  |  |
| --- | --- |
| **EXP.NO: 03** | **Implement classification using Multilayer perceptron** |
| **DATE:** |

**AIM**

Implement classification using Multilayer perceptron using Iris dataset

**PROCEDURE**

1. Data Preprocessing:

Load and perform data preprocessing operations on your dataset, such as feature scaling, categorical variable encoding, and dividing the data into training and test sets.

1. Define the MLP Architecture:

Select the MLP's architectural details, such as the number of layers, the number of neurons in each layer, and the activation methods.

1. Set Up the Model:

Using a deep learning framework like PyTorch or TensorFlow, create an instance of the neural network model. Explain the layers and activation mechanisms.

1. Define the Loss Function:

Choose a loss function that is appropriate for your task. You could use Mean Squared Error (MSE) loss for regression and Cross-Entropy loss for classification.

1. Choose an Optimization Algorithm:

To update the model's weights during training, use an optimizer such as Stochastic Gradient Descent (SGD), Adam, or RMSprop.

* 2. Training Loop:

Repeat the training data iteratively for a predetermined number of epochs.

* In every period, carry out the following actions:

Compute Loss Forward Pass

Reverse propagation

Refresh Weights

* 3. Evaluate the Model:

Following training, assess the model's effectiveness using a different test dataset. Depending on the job, common evaluation criteria include accuracy, precision, recall, F1-score, and mean absolute error (MAE).

**PROGRAM**

import torch

import torch.nn as nn

import torch.optim as optim

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import accuracy\_score

data = load\_iris()

X = data.data

y = data.target

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

X\_train = torch.tensor(X\_train, dtype=torch.float32)

y\_train = torch.tensor(y\_train, dtype=torch.int64)

X\_test = torch.tensor(X\_test, dtype=torch.float32)

class MLP(nn.Module):

def \_\_init\_\_(self, input\_dim, hidden\_dim, output\_dim):

super(MLP, self).\_\_init\_\_()

self.fc1 = nn.Linear(input\_dim, hidden\_dim)

self.relu = nn.ReLU()

self.fc2 = nn.Linear(hidden\_dim, output\_dim)

def forward(self, x):

x = self.fc1(x)

x = self.relu(x)

x = self.fc2(x)

return x

input\_dim = X\_train.shape[1]

hidden\_dim = 64

output\_dim = len(torch.unique(y\_train))

learning\_rate = 0.01

epochs = 100

model = MLP(input\_dim, hidden\_dim, output\_dim)

criterion = nn.CrossEntropyLoss()

optimizer = optim.SGD(model.parameters(), lr=learning\_rate)

for epoch in range(epochs):

optimizer.zero\_grad()

outputs = model(X\_train)

loss = criterion(outputs, y\_train)

loss.backward()

optimizer.step()

if (epoch + 1) % 10 == 0:

print(f'Epoch [{epoch + 1}/{epochs}], Loss: {loss.item():.4f}')

model.eval()

with torch.no\_grad():

y\_pred = model(X\_test)

\_, predicted = torch.max(y\_pred, 1)

accuracy = accuracy\_score(y\_test, predicted.numpy())

print(f'Accuracy on test data: {accuracy:.2f}')

from sklearn.metrics import classification\_report

model.eval()

with torch.no\_grad():

y\_pred = model(X\_test)

\_, predicted = torch.max(y\_pred, 1)

accuracy = accuracy\_score(y\_test, predicted.numpy())

print(f'Accuracy on test data: {accuracy:.2f}')

# Generate the classification report

report = classification\_report(y\_test, predicted.numpy())

print("Classification Report:\n", report)

from sklearn.metrics import classification\_report

import matplotlib.pyplot as plt

import seaborn as sns

classification\_report\_str = """

precision recall f1-score support

0 1.00 1.00 1.00 10

1 1.00 0.44 0.62 9

2 0.69 1.00 0.81 11

accuracy 0.83 30

macro avg 0.90 0.81 0.81 30

weighted avg 0.89 0.83 0.82 30

"""

lines = classification\_report\_str.strip().split('\n')

class\_names = [line.split()[0] for line in lines[2:-5]] # Extract class names

f1\_scores = [float(line.split()[3]) for line in lines[2:-5]]

plt.figure(figsize=(10, 6))

sns.barplot(x=f1\_scores, y=class\_names, palette='viridis')

plt.xlabel('F1-Score')

plt.ylabel('Class Name')

plt.title('F1-Scores by Class')

plt.xlim(0, 1) # Adjust the x-axis limits if necessary

plt.show()

import matplotlib.pyplot as plt

from sklearn.datasets import load\_iris

data = load\_iris()

X = data.data

y = data.target

feature1 = 0 # Index of the first feature (change as needed)

feature2 = 1

class\_labels = ['Class 0', 'Class 1', 'Class 2']

colors = ['blue', 'red', 'green']

plt.figure(figsize=(8, 6))

for i in range(len(class\_labels)):

plt.scatter(X[y == i, feature1], X[y == i, feature2], label=class\_labels[i], c=colors[i], marker='o')

plt.xlabel(data.feature\_names[feature1])

plt.ylabel(data.feature\_names[feature2])

plt.title(f'Scatter Plot of Iris Dataset ({data.feature\_names[feature1]} vs. {data.feature\_names[feature2]})')

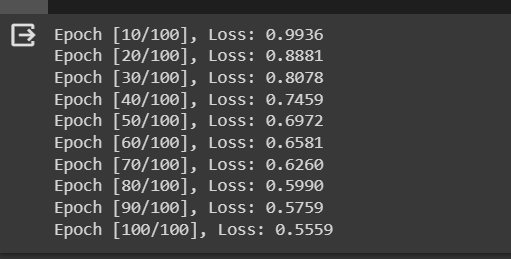
plt.legend()

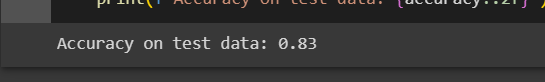
plt.show()

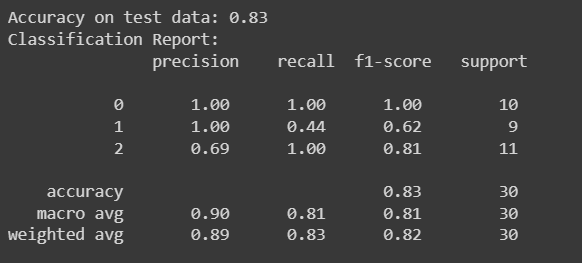
**INPUT**

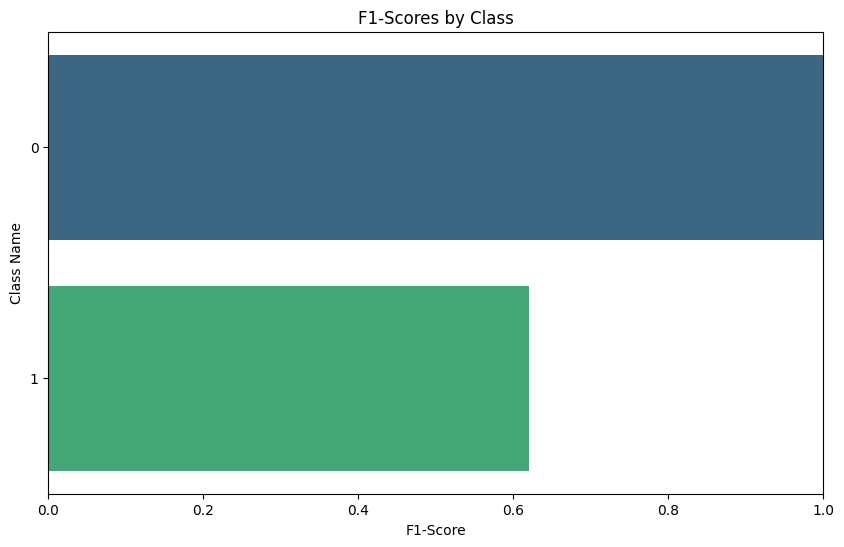
Iris dataset

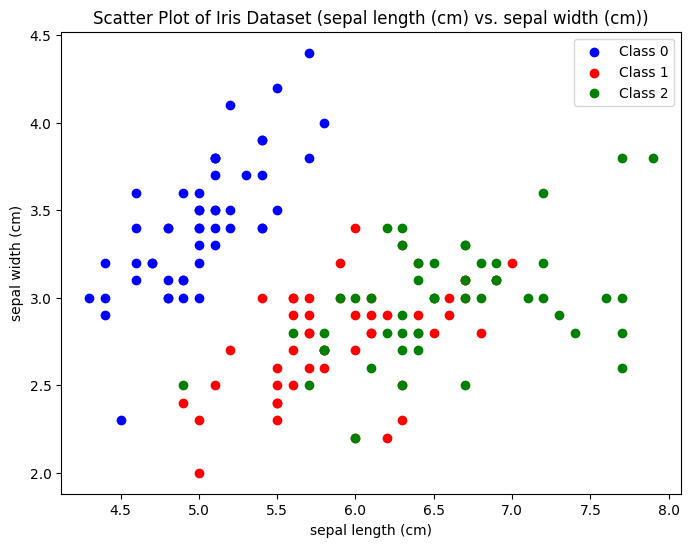
**OUTPUT**

****

****

****





**RESULT**

The accuracy for the model is: **83%** using **Multilayer perceptron.**

1. Data: You must load your own dataset and properly preprocess it. The program expects inputs and targets, where targets hold the target labels and inputs hold feature data. (0 or 1 for binary classification).

2. Model Architecture: The code utilizes the ReLU activation function to define an MLP with one hidden layer (hidden\_dim). Your challenge and dataset may necessitate adjusting the particular design.

3. Hyperparameters: The code specifies hyperparameters like the training epoch count (epochs) and learning rate (learning\_rate). To get the best model performance for your particular scenario, you might need to modify these hyperparameters.

4. Loss Function and Optimization: The algorithm employs stochastic gradient descent (SGD) as the optimizer and binary cross-entropy loss (nn.BCEWithLogitsLoss()) for binary classification.

|  |  |
| --- | --- |
| **EXP.NO.4** | **Implement classification using SVM** |
| **Date:19/10/23** |

**AIM**

Implement classification using SVM using Pumpkin seed dataset

**PROCEDURE**

1. Preprocessing of Data:

Load the pumpkin seeds dataset.

Preprocessing operations such addressing missing values, encoding categorical variables, or scaling numerical features should be carried out as needed.

2. Splitting the Data:

Using train\_test\_split from sklearn.model\_selection, divide the dataset into training and testing sets.

3. Scaling Features:

Utilizing StandardScaler from sklearn.preprocessing, standardize the features.

4. Train the SVM Model

Initialize an SVM classifier using SVC from sklearn.svm.

Pick a suitable kernel for the SVM (for example, linear, polynomial, or RBF).

Using the training set of data, fit the model.

5. Make Predictions:

Make predictions based on the testing data using the trained model.

6. Evaluate the Model:

Utilizing the accuracy\_score and classification\_report from sklearn.metrics, determine the accuracy and other pertinent metrics.

**PROGRAM**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

df = pd.read\_excel("Pumpkin\_Seeds\_Dataset.xlsx")

df.head()

df.info()

plt.figure(figsize=(12,8))

sns.scatterplot(data=df, x="Area", y="Perimeter", hue="Class", alpha=0.7, palette='coolwarm')

df = pd.get\_dummies(df,drop\_first=True)

from sklearn.svm import SVC

from sklearn.preprocessing import StandardScaler

from sklearn.model\_selection import GridSearchCV

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

df.head()

X = df.drop("Class\_Ürgüp Sivrisi", axis=1)

y = df["Class\_Ürgüp Sivrisi"]

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=47)

scaler = StandardScaler()

model = SVC()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

clf = SVC(kernel='linear', random\_state=42)

clf.fit(X\_train, y\_train)

y\_pred = clf.predict(X\_test)

model.get\_params().keys()

param\_grid = {"C": [0,0.1,1,10,100,1000], "degree":[2,3,4,5], "gamma": ["scale","auto"], "kernel": ['linear', 'poly', 'rbf']}

grid = GridSearchCV(model,param\_grid, cv=5, scoring="accuracy")

accuracy = accuracy\_score(y\_test, y\_pred)

print(f'Accuracy: {accuracy:.2f}')

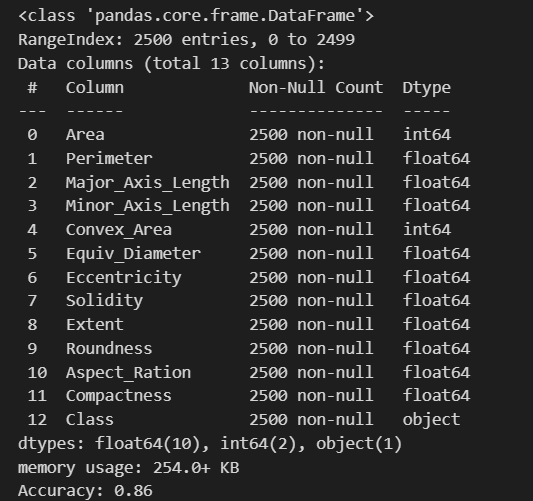
print(classification\_report(y\_test, y\_pred))

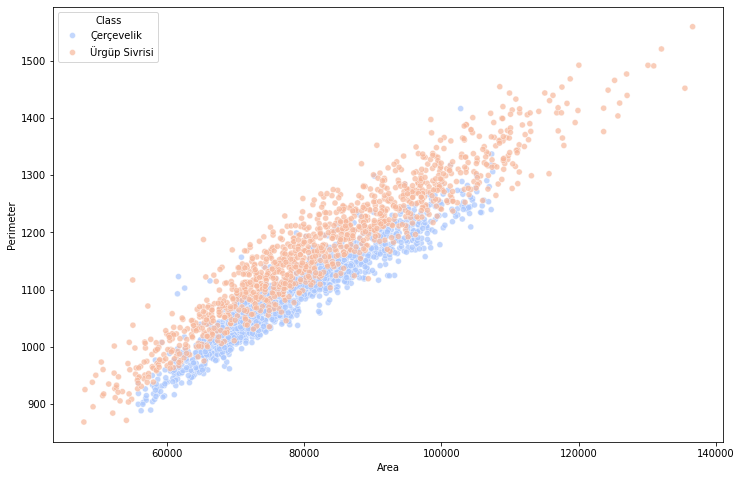
confusion\_matrix(y\_test, y\_pred)

**INPUT**

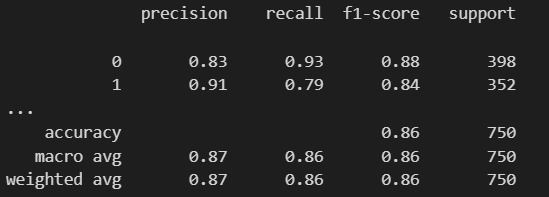
Pumpkin seed dataset

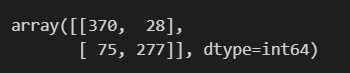
**OUTPUT**











**RESULT**

The accuracy for the model is: **86%** using **SVM.**

* The dataset is loaded, and basic exploration is performed.
* Data preprocessing steps, such as one-hot encoding, are applied.
* The data is split into training and testing sets.
* The SVM model with a linear kernel is trained on the scaled training data.
* Predictions are made on the test set, and the model's performance is evaluated using the accuracy score and the classification report.
* A param\_grid dictionary is defined for hyperparameter tuning.
* Grid search is performed using the defined parameter grid and 5-fold cross-validation to find the best combination of hyperparameters based on the accuracy score.

|  |  |
| --- | --- |
| **EXP.NO.4** | **Implement Adaboost Algorithm** |
| **Date: 7/11/23** |

**AIM**

Implement classification using Adaboost Algorithm using Pumpkin seed dataset

**PROCEDURE**

* Preprocessing of Data:
* Load the pumpkin seeds dataset.
* Address any missing values (if any exist in the dataset).
* Since the dataset contains numerical features, scaling might be beneficial for some algorithms. Use StandardScaler from sklearn.preprocessing to standardize the features.
* Splitting the Data:
* Use train\_test\_split from sklearn.model\_selection to divide the dataset into training and testing sets. Set an appropriate test size (e.g., 20%).
* Scaling Features (if necessary):
* Utilize StandardScaler from sklearn.preprocessing to standardize the features. AdaBoost, in particular, can work well with weak learners, so feature scaling might not always be necessary.
* Training the AdaBoost Model:
* Initialize an AdaBoost classifier using AdaBoostClassifier from sklearn.ensemble.
* Specify the base estimator (e.g., Decision Tree, SVM, etc.) and the number of weak learners (n\_estimators). For instance, you can use DecisionTreeClassifier as the base estimator.
* Train the AdaBoost model using the training set of data.
* Make Predictions:
* Make predictions based on the testing data using the trained AdaBoost model.
* Evaluate the Model:
* Utilize accuracy\_score and classification\_report from sklearn.metrics to determine the accuracy and other pertinent metrics.
* Interpret the classification report to understand precision, recall, and F1-score for each class.

**PROGRAM**

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.ensemble import AdaBoostClassifier

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

df = pd.read\_excel("Pumpkin\_Seeds\_Dataset.xlsx")

df.head()

# Assume your target variable is named 'Class'

X = df.drop(columns=['Class']) # Features

y = df['Class'] # Target variable

# Convert class labels to binary labels

y\_binary = y.apply(lambda x: 1 if x == 'Çerçevelik' else 0)

#data split into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

adaboost = AdaBoostClassifier(n\_estimators=50, random\_state=42) # You can adjust the number of estimators as needed

adaboost.fit(X\_train, y\_train)

predictions = adaboost.predict(X\_test)

accuracy = accuracy\_score(y\_test, predictions)

print("Accuracy:", accuracy)

class\_report = classification\_report(y\_test, predictions)

print("Classification Report:\n", class\_report)

# Import necessary libraries

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.ensemble import AdaBoostClassifier

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix, roc\_curve, auc

import matplotlib.pyplot as plt

# Generate confusion matrix

conf\_matrix = confusion\_matrix(y\_test, predictions)

# Plot confusion matrix heatmap

plt.figure(figsize=(8, 6))

sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['Not Çerçevelik', 'Çerçevelik'], yticklabels=['Not Çerçevelik', 'Çerçevelik'])

plt.xlabel('Predicted Labels')

plt.ylabel('True Labels')

plt.title('Confusion Matrix')

plt.show()

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

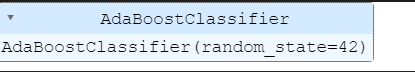
plt.figure(figsize=(12,8))

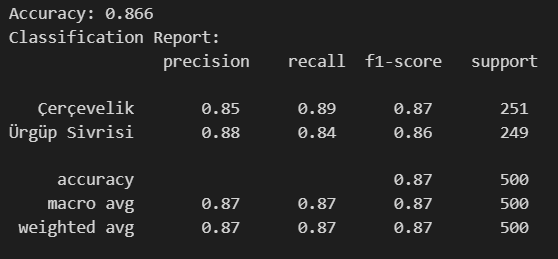
sns.scatterplot(data=df, x="Area", y="Perimeter", hue="Class", alpha=0.7)

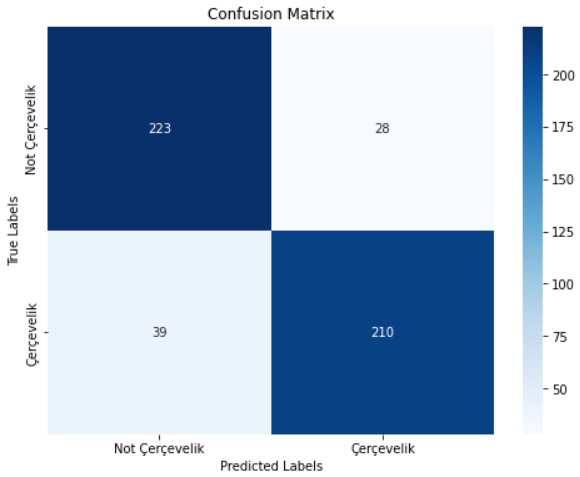
**INPUT**

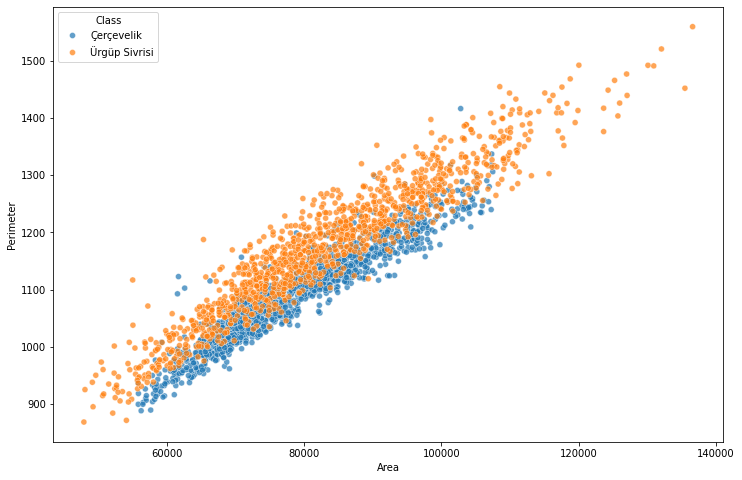
Pumpkin seed dataset

**OUTPUT**









**RESULT**

The accuracy for the model is: **86%** using **Adaboost Algorithm.**

|  |  |
| --- | --- |
| **EXP.NO.4** | **Implement Bagging using Random Forests** |
| **Date: 7/11/23** |

**AIM**

Implement classification using Bagging using Random Forest on pumpkin seed dataset.

**PROCEDURE**

* Preprocessing of Data:
* Load the pumpkin seeds dataset.
* Address any missing values if necessary.
* Since the dataset contains numerical features, scaling might be beneficial for some algorithms. Use StandardScaler from sklearn.preprocessing to standardize the features.
* Splitting the Data:
* Use train\_test\_split from sklearn.model\_selection to divide the dataset into training and testing sets. Set an appropriate test size (e.g., 20%).
* Training the Random Forest Model:
* Initialize a Random Forest classifier using RandomForestClassifier from sklearn.ensemble.
* Specify the number of trees (n\_estimators) and other hyperparameters.
* Train the Random Forest model using the training set of data.
* Make Predictions:
* Make predictions based on the testing data using the trained Random Forest model.
* Evaluate the Model:
* Utilize accuracy\_score and classification\_report from sklearn.metrics to determine the accuracy and other pertinent metrics.
* Interpret the classification report to understand precision, recall, and F1-score for each class.

**PROGRAM**

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy\_score, classification\_report

data = pd.read\_excel("Pumpkin\_Seeds\_Dataset.xlsx")

print(data.shape)

data.head()

X = data.iloc[:,:-1]

y = data.iloc[:,-1]

print("Shape of X is %s and shape of y is %s"%(X.shape,y.shape))

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.25, random\_state=42)

rf\_classifier = RandomForestClassifier(n\_estimators=100, random\_state=42)

rf\_classifier.fit(X\_train, y\_train)

y\_pred = rf\_classifier.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

print(f'Accuracy: {accuracy:.2f}')

print(classification\_report(y\_test, y\_pred))

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.metrics import confusion\_matrix

cm = confusion\_matrix(y\_test, y\_pred)

plt.figure(figsize=(8, 6))

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False)

plt.title('Confusion Matrix')

plt.xlabel('Predicted')

plt.ylabel('Actual')

plt.show()

pip install seaborn matplotlib

feature\_importances = rf\_classifier.feature\_importances\_

feature\_importance\_df = pd.DataFrame({'Feature': X.columns, 'Importance': feature\_importances})

feature\_importance\_df = feature\_importance\_df.sort\_values(by='Importance', ascending=False)

plt.figure(figsize=(12, 6))

sns.barplot(x='Importance', y='Feature', data=feature\_importance\_df, palette='plasma')

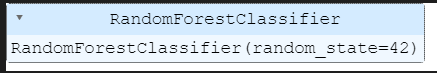
plt.title('Feature Importances')

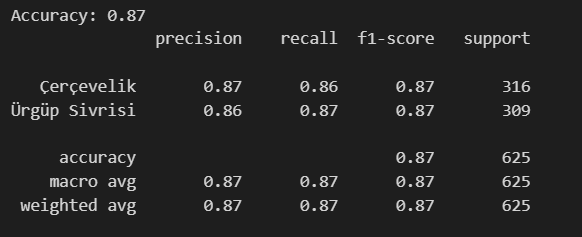
plt.show()

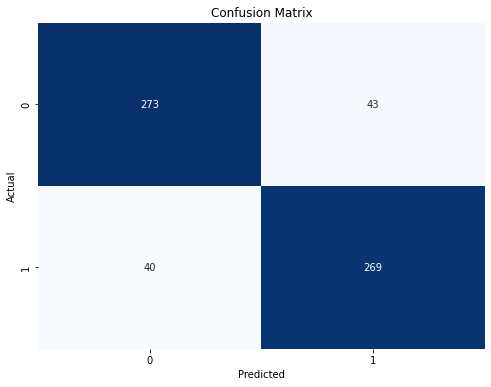
**INPUT**

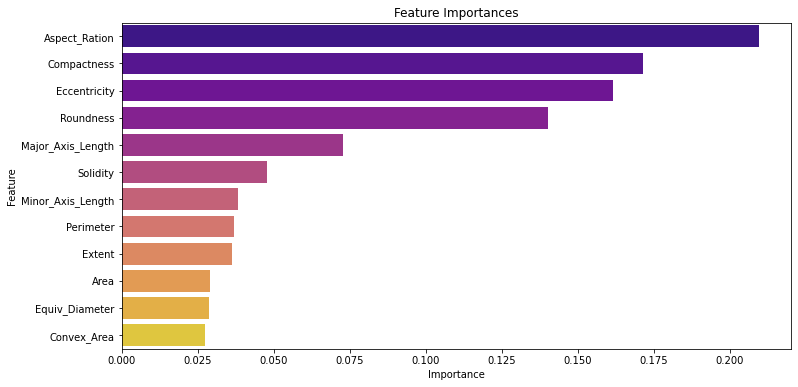
Pumpkin seed dataset

**OUTPUT**









**RESULT**

The accuracy for the model is: **87%** using **Bagging in Random Forest.**

|  |  |
| --- | --- |
| **EXP.NO.4** | **Implement K-means Clustering to Find Natural Patterns in Data** |
| **Date: 28/11/23** |

**AIM**

Implement classification using Implement K-means Clustering on Pumpkin seed dataset.

**PROCEDURE**

Preprocessing of Data:

•Load the pumpkin seeds dataset.

•Address any missing values if necessary.

•Since the dataset contains numerical features, scaling might be beneficial for some algorithms. Use StandardScaler from sklearn.preprocessing to standardize the features.

Splitting the Data:

•Use train\_test\_split from sklearn.model\_selection to divide the dataset into training and testing sets. Set an appropriate test size (e.g., 20%).

Training the KMean Model:

•Initialize a KMean from sklearn.cluster import.

•Specify the (n\_estimators) and other hyperparameters.

•Train the K Mean model using the training set of data.

Make Predictions:

•Make predictions based on the testing data using the trained K Mean model.

Evaluate the Model:

•Utilize accuracy\_score and classification\_report from sklearn.metrics to determine the accuracy and other pertinent metrics.

•Interpret the classification report to understand precision, recall, and F1-score for each class.

**PROGRAM**

import pandas as pd

from sklearn.cluster import KMeans

from sklearn.metrics import accuracy\_score, classification\_report

from sklearn.preprocessing import LabelEncoder

from sklearn.model\_selection import train\_test\_split

import matplotlib.pyplot as plt

from sklearn.metrics import silhouette\_score

from sklearn.preprocessing import StandardScaler

data = pd.read\_excel("Pumpkin\_Seeds\_Dataset.xlsx")

print(data.shape)

data.head()

X = data.drop('Class', axis=1)

true\_labels = data['Class']

k = 3

kmeans = KMeans(n\_clusters=k, random\_state=42)

data['cluster\_labels'] = kmeans.fit\_predict(X)

label\_encoder = LabelEncoder()

data['pseudo\_labels'] = label\_encoder.fit\_transform(data['cluster\_labels'])

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, data['pseudo\_labels'], test\_size=0.2, random\_state=42)

accuracy = accuracy\_score(y\_test, kmeans.predict(X\_test))

classification\_report\_str = classification\_report(y\_test, kmeans.predict(X\_test))

print(f'Accuracy: {accuracy:.2f}')

print(classification\_report\_str)

data['Cluster'] = kmeans.labels\_

plt.scatter(data['Area'], data['Perimeter'], c=data['Cluster'], cmap='viridis', edgecolors='k')

plt.scatter(kmeans.cluster\_centers\_[:, 0], kmeans.cluster\_centers\_[:, 1], c='red', marker='X', s=200, label='Centroids')

plt.title('K-means Clustering')

plt.xlabel('Area')

plt.ylabel('Perimeter')

plt.legend()

plt.show()

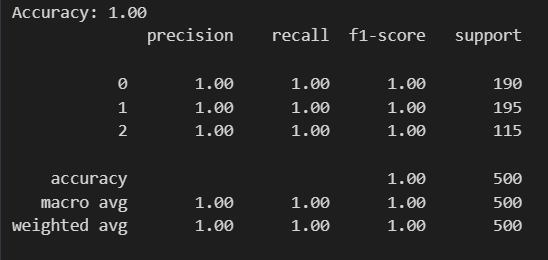
silhouette\_avg = silhouette\_score(X, kmeans.labels\_)

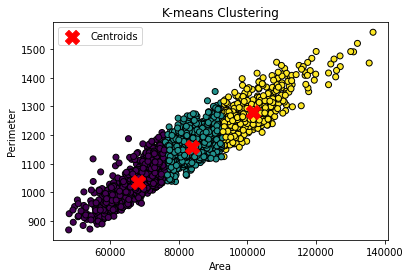
print(f'Silhouette Score: {silhouette\_avg:.2f}')

**INPUT**

Pumpkin dataset

**OUTPUT**





**RESULT**

The accuracy for the model is: **100%** using **K-Means clustering.**

The Silhouette Score is: 0.54

|  |  |
| --- | --- |
| **EXP.NO.4** | **Implement Principle Component Analysis for Dimensionality Reduction** |
| **Date: 28/11/23** |

**AIM**

Implement Principle Component Analysis for Dimensionality Reduction in Data on Breast cancer dataset.

**PROCEDURE**

1. Import Libraries:

- Import necessary libraries, including `matplotlib.pyplot`, `numpy`, `pandas`, `load\_breast\_cancer` from `sklearn.datasets`, `StandardScaler` from `sklearn.preprocessing`, `PCA` from `sklearn.decomposition`, `train\_test\_split` from `sklearn.model\_selection`, `SVC` from `sklearn.svm`, and `accuracy\_score` from `sklearn.metrics`.

2. Load Breast Cancer Dataset:

- Use `load\_breast\_cancer()` to load the Breast Cancer dataset.

3. Explore Dataset:

- Check the available keys in the dataset (`cancer.keys()`).

- Display the dataset description (`print(cancer['DESCR'])`).

4. Create DataFrame:

- Create a DataFrame (`df`) using the dataset's features and feature names.

5. Data Scaling:

- Use `StandardScaler` to standardize the features.

6. Apply PCA:

- Use `PCA` to reduce the dimensionality of the standardized data to 2 components.

7. Split the Data:

- Use `train\_test\_split` to split the PCA-transformed data into training and testing sets.

8. Train a Support Vector Machine (SVM) Classifier:

- Initialize an SVM classifier (`SVC`) with a linear kernel and specified hyperparameters.

- Train the SVM classifier using the training set.

9. Make Predictions:

- Use the trained SVM classifier to make predictions on the testing set.

10. Calculate Accuracy:

- Use `accuracy\_score` to calculate the accuracy of the SVM classifier on the testing set.

11. Visualization:

- Create a scatter plot to visualize the PCA-transformed data points, color-coded by the target variable.

- Display the decision boundary of the SVM classifier.

**PROGRAM**

# Principal Component Analysis (PCA)

import matplotlib.pyplot as plt

import numpy as np

import pandas as pd

from sklearn.datasets import load\_breast\_cancer

from sklearn.preprocessing import StandardScaler

from sklearn.decomposition import PCA

from sklearn.model\_selection import train\_test\_split

from sklearn.svm import SVC

from sklearn.metrics import accuracy\_score

# Load Breast Cancer Dataset

cancer = load\_breast\_cancer()

# Create DataFrame

df = pd.DataFrame(cancer['data'], columns=cancer['feature\_names'])

# Standardize the features

scaler = StandardScaler()

scaler.fit(df)

scaled\_data = scaler.transform(df)

# Apply PCA

pca = PCA(n\_components=2)

pca.fit(scaled\_data)

x\_pca = pca.transform(scaled\_data)

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(x\_pca, cancer['target'], test\_size=0.2, random\_state=42)

# Train a Support Vector Machine (SVM) classifier

svm\_classifier = SVC(kernel='linear', C=1.0, random\_state=42)

svm\_classifier.fit(X\_train, y\_train)

# Make predictions on the testing set

y\_pred = svm\_classifier.predict(X\_test)

# Calculate accuracy

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Accuracy: {accuracy:.2f}")

# Visualization

plt.figure(figsize=(8, 6))

plt.scatter(x\_pca[:, 0], x\_pca[:, 1], c=cancer['target'])

plt.xlabel('First principal component')

plt.ylabel('Second principal component')

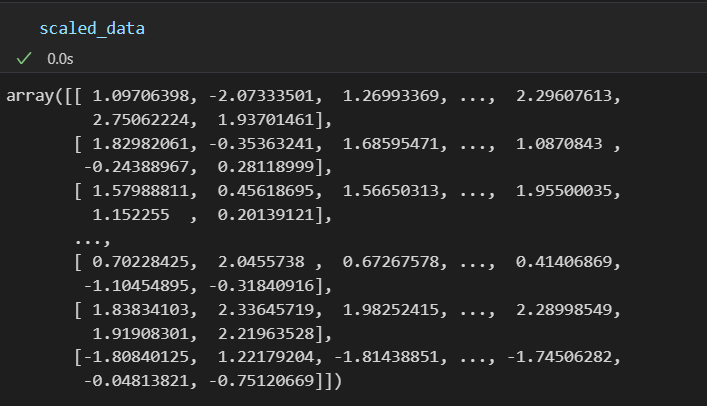
plt.title('PCA - Breast Cancer Dataset with SVM Decision Boundary')

plt.show()

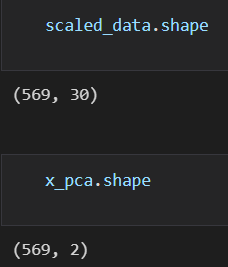
**INPUT**

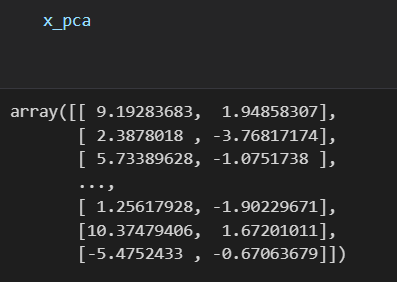
Breast Cancer Dataset

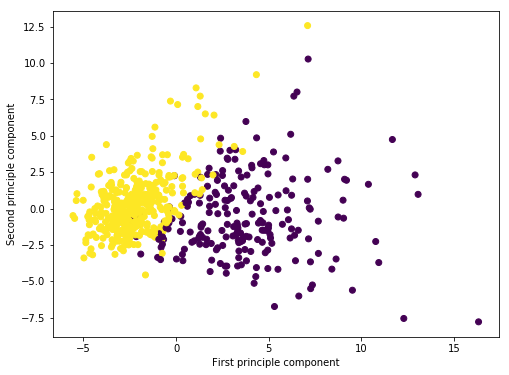
**OUTPUT**











**RESULT**

The model reduces the dimensions using PCA.

Accuracy = 99%

|  |  |
| --- | --- |
| **EXP.NO.4** | **Evaluating ML algorithm with balanced and unbalanced datasets** |
| **Date: 28/11/23** |

**AIM**

Implement ML algorithm with balanced and unbalanced datasets on Cerebral Dataset.

**PROCEDURE**

1. Import Libraries:

- Import necessary libraries, including `pandas`, `numpy`, `seaborn`, `matplotlib.pyplot`, `KNNImputer` from `sklearn.impute`, `MinMaxScaler` from `sklearn.preprocessing`, `train\_test\_split` from `sklearn.model\_selection`, various classifiers (`KNeighborsClassifier`, `GaussianNB`, `DecisionTreeClassifier`, `RandomForestClassifier`, `AdaBoostClassifier`) from `sklearn`, `classification\_report` from `sklearn.metrics`, and `SMOTE`, `RandomUnderSampler`, `SMOTEENN` from `imblearn`.

2. Load Cerebral Stroke Dataset:

- Use `pd.read\_csv()` to load the Cerebral Stroke dataset into a DataFrame (`df`).

3. Explore Dataset:

- Display the first 10 rows of the dataset (`df.head(10)`).

- Check for missing values using `df.isnull().sum()`.

- Explore dataset statistics using `df.describe()`.

4. Handling Missing Values:

- Identify columns with a significant number of NULL values, such as BMI and Smoking Stroke.

- Perform one-hot encoding on categorical variables using `pd.get\_dummies()`.

- Use `KNNImputer` from `sklearn.impute` to impute missing values.

5. Machine Learning Model Setup:

- Separate features (`X`) and the target variable (`y`) from the preprocessed DataFrame.

- Apply `MinMaxScaler` to scale the features.

- Split the dataset into training and testing sets using `train\_test\_split`.

6. Model Selection:

- Initialize classification models: `KNeighborsClassifier`, `GaussianNB`, `DecisionTreeClassifier`, `RandomForestClassifier`.

- Loop through models, fit them on the training data, and evaluate performance using `classification\_report`.

7. Model Evaluation Without Resampling:

- Print classification reports for each model, highlighting the imbalance issue in predicting the positive cases.

8. OverSampling (SMOTE):

- Use Synthetic Minority Over-sampling Technique (SMOTE) to oversample the minority class.

- Split the oversampled data into training and testing sets.

- Loop through models, fit them on the oversampled training data, and evaluate performance using `classification\_report`.

9. UnderSampling:

- Perform random under-sampling to balance the class distribution.

- Split the undersampled data into training and testing sets.

- Loop through models, fit them on the undersampled training data, and evaluate performance using `classification\_report`.

10. Combining OverSampling and UnderSampling (SMOTEEN):

- Use SMOTEENN to combine upsampling and downsampling.

- Split the combined data into training and testing sets.

- Loop through models, fit them on the combined training data, and evaluate performance using `classification\_report`.

11. Conclusion:

- Summarize the findings from different resampling techniques.

- Emphasize the improvement in model performance, particularly with SMOTEEN.

**PROGRAM**

# Evaluating ML algorithm with unbalanced datasets.

# Importing Libraries

import pandas as pd

import numpy as np

import seaborn as sns

import matplotlib.pyplot as plt

df = pd.read\_csv("Cereberal\_Dataset.csv")

df.head(10)

df.describe()

df.isnull().sum()

BMI and Smoking Stroke has a lot of NULL| values

df.shape

df.dtypes

df['stroke'].value\_counts()

Very imbalace 0,1 parameters

sns.countplot(x='stroke', data=df)

plt.title("Imbalance data")

plt.show()

## One Hot Encoding

Converting the categorical values, indicator variables

Handling Categorical Variables:

One-hot encoding is performed on categorical variables using `pd.get\_dummies()`.

df.columns

df = pd.get\_dummies(df,columns=['gender','ever\_married','work\_type','Residence\_type','smoking\_status'])

df.head(4)

## Handling Missing Values:

Missing values are imputed using the k-nearest neighbors algorithm (`KNNImputer` from `sklearn.impute`).

from sklearn.impute import KNNImputer

imputer = KNNImputer(missing\_values=np.nan)

tab = imputer.fit\_transform(df)

df\_new = pd.DataFrame(tab, columns=df.columns)

df\_new.head(10)

df\_new.shape

The Columns has been transformed from 12 to 22

df\_new.isnull().sum()

df\_new.dtypes

## Machine Learning Model

X = df\_new.drop('stroke',axis=1)

y = df\_new['stroke']

## Feature Scaling and Train-Test Split:

Features are scaled using `MinMaxScaler`, and the dataset is split into training and testing sets.

from sklearn.preprocessing import MinMaxScaler

from sklearn.model\_selection import train\_test\_split

scaler = MinMaxScaler()

X = scaler.fit\_transform(X)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y ,test\_size=0.3, random\_state=42)

## Model Selection:

Several classification models are chosen (`KNeighborsClassifier`, `GaussianNB`, `DecisionTreeClassifier`, `RandomForestClassifier`) for initial testing.

## Model Evaluation Without Resampling:

Classification reports are generated for each model to evaluate their performance on the imbalanced dataset.

An imbalanced dataset is one where some classes have significantly more examples than others. This can lead to problems like poor model performance for the minority class, misleading evaluation metrics, and difficulties learning features.

from sklearn.neighbors import KNeighborsClassifier

from sklearn.svm import SVC

from sklearn.naive\_bayes import GaussianNB

from sklearn.tree import DecisionTreeClassifier

from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier

from sklearn.metrics import classification\_report

knn = KNeighborsClassifier()

nb = GaussianNB()

dt = DecisionTreeClassifier()

rf = RandomForestClassifier()

models = [knn, nb, dt, rf]

for model in models:

print("MODEL NAME: ", model)

model.fit(X\_train, y\_train)

y\_pred = model.predict(X\_test)

print(classification\_report(y\_test, y\_pred))

The models accurately predicts the 0 but not 1. The precision is close to zero in all the above cases, which means the model failed to predict the cases where chances for cerebral stroke was actually present. So the above models are useless.

## OverSampling (SMOTE):

The script uses the Synthetic Minority Over-sampling Technique (SMOTE) to oversample the minority class.

from imblearn.over\_sampling import SMOTE

os = SMOTE(random\_state=1)

X\_os, y\_os = os.fit\_resample(X,y)

X\_train,X\_test,y\_train,y\_test = train\_test\_split(X\_os,y\_os,test\_size=0.3,random\_state=1)

for model in models:

print("MODEL NAME:",model)

model.fit(X\_train,y\_train)

y\_pred = model.predict(X\_test)

print(classification\_report(y\_test,y\_pred))

After applying SMOTE technique, the precision increased. For KNN it is 0.88, for Decision Tree it is 0.97 and RandomForest it is 0.95. So oversampling resulted in a better model that is capable of identifying the cases positive for stroke.

## UnderSampling:

Random under-sampling is performed to balance the class distribution.

from imblearn.under\_sampling import RandomUnderSampler

us= RandomUnderSampler(random\_state=1)

X\_us, y\_us = us.fit\_resample(X,y)

X\_train,X\_test,y\_train,y\_test = train\_test\_split(X\_us,y\_us,test\_size=0.3,random\_state=1)

for model in models:

print("MODEL NAME:",model)

model.fit(X\_train,y\_train)

y\_pred = model.predict(X\_test)

print(classification\_report(y\_test,y\_pred))

### Combining Oversampling and Undersampling

SMOTEEN combines SMOTE and Edited Nearest Neighbours(ENN). SMOTEEN performs upsampling and downsampling at the same time.

from imblearn.combine import SMOTEENN

sample = SMOTEENN()

X\_over,y\_over = sample.fit\_resample(X,y)

X\_train,X\_test,y\_train,y\_test = train\_test\_split(X\_over,y\_over,test\_size=0.3,random\_state=1)

for model in models:

print("MODEL NAME:",model)

model.fit(X\_train,y\_train)

y\_pred = model.predict(X\_test)

print(classification\_report(y\_test,y\_pred))

<b>End Note</b> : For the Cerebral Stroke Imbalanced data we could make a better model using resampling techniques.

## Conclusion:

- The script provides classification reports for each model after different resampling techniques.

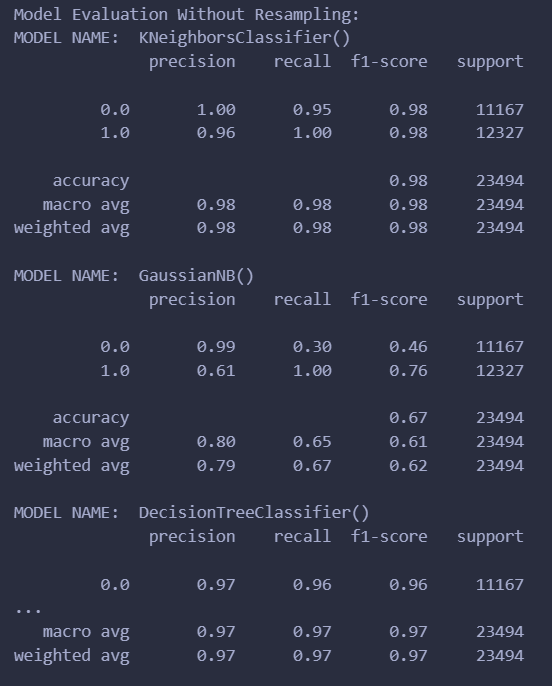
- It highlights that resampling techniques, particularly SMOTEENN, improve the model's ability to identify cases positive for stroke.

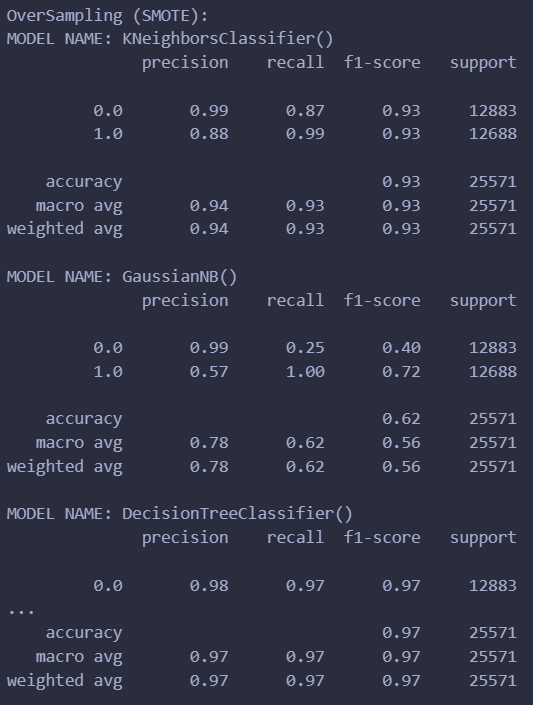
**INPUT**

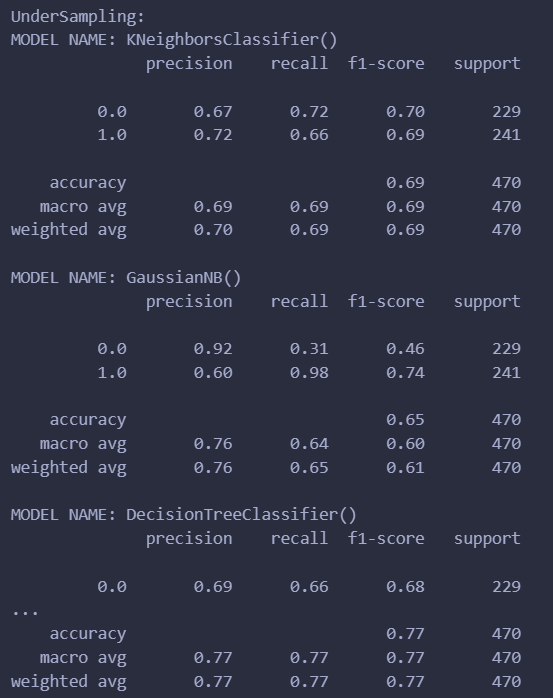
Cereberal Dataset

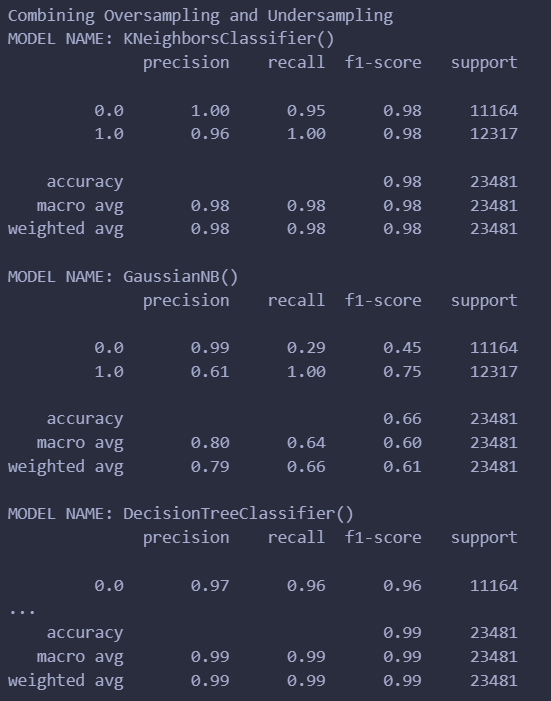
**OUTPUT**

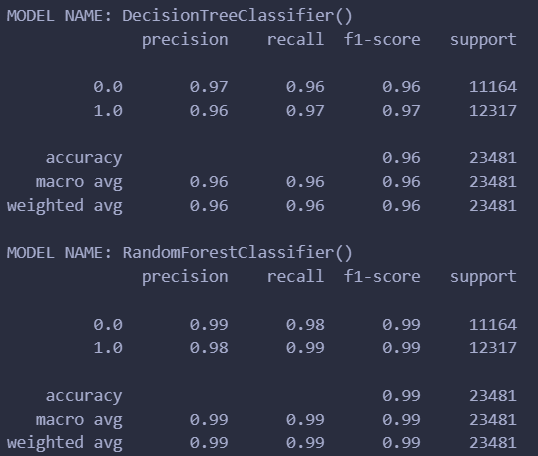












**RESULT**

Resampling techniques, particularly SMOTEEN, significantly improved the prediction.

Combining Oversampling and Undersampling Accuracy:

MODEL NAME: KNeighborsClassifier()

accuracy = 0.98

MODEL NAME: GaussianNB()

accuracy = 0.66

MODEL NAME: DecisionTreeClassifier()

accuracy = 0.96

RandomForestClassifier()

**accuracy = 0.99**

|  |  |
| --- | --- |
| **EXP.NO.4** | **Comparison of Machine Learning algorithms** |
| **Date: 28/11/23** |

**AIM**

Comparison of Machine Learning algorithms for Diabetes Prediction.

**PROCEDURE**

Preprocessing of Data:

- Load the diabetes dataset using Pandas.

- Address any missing values in the dataset.

- Since the dataset contains numerical features, use `StandardScaler` from `sklearn.preprocessing` to standardize the features.

Splitting the Data:

- Use `train\_test\_split` from `sklearn.model\_selection` to divide the dataset into training and testing sets.

- Set an appropriate test size (e.g., 20%).

Training the Models:

- Initialize various classification models (Random Forest, Decision Tree, K-Nearest Neighbors, Naive Bayes, Logistic Regression, Support Vector Machine).

- Train each model using the training set of data.

Make Predictions:

- Make predictions based on the testing data using the trained models.

Evaluate the Models:

- Utilize `accuracy\_score` from `sklearn.metrics` to determine the accuracy of each model.

- Optionally, use `classification\_report` from `sklearn.metrics` for more detailed metrics, including precision, recall, and F1-score for each class.

**PROGRAM**

# Comprehensive Comparison of Machine Learning Algorithms for Diabetes Prediction

# Import Libraries

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.ensemble import RandomForestClassifier

from sklearn.tree import DecisionTreeClassifier

from sklearn.neighbors import KNeighborsClassifier

from sklearn.naive\_bayes import GaussianNB

from sklearn.linear\_model import LogisticRegression

from sklearn.svm import SVC

from sklearn.metrics import accuracy\_score

from sklearn.preprocessing import StandardScaler

from sklearn.feature\_selection import SelectKBest

from sklearn.feature\_selection import f\_classif

### Importing the required libraries and modules:

1. numpy and pandas for data manipulation.

1. train\_test\_split from sklearn.model\_selection for splitting data into training and testing sets.

1. Classifiers from sklearn.ensemble, sklearn.tree, sklearn.neighbors, sklearn.naive\_bayes, sklearn.linear\_model, and sklearn.svm for various machine learning algorithms.

1. accuracy\_score from sklearn.metrics to evaluate model performance.

1. StandardScaler from sklearn.preprocessing for feature scaling.

1. SelectKBest and f\_classif from sklearn.feature\_selection for feature selection.

# Load the data

data = pd.read\_csv("diabetes.csv", encoding="ISO-8859-1")

print(data.shape)

data.head()

# Separate features and labels

X = data.drop("Outcome", axis=1) # Assuming "Outcome" is the target column

y = data["Outcome"]

## These lines separate the dataset into two parts:

\* X contains the feature columns, excluding the "Outcome" column (assumed to be the target).

\* y contains the target column, which is "Outcome" in this case.

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

## Spliting the data into training and testing sets using train\_test\_split:

\* X\_train and y\_train represent the features and labels for the training set.

\* X\_test and y\_test represent the features and labels for the testing set.

\* test\_size=0.2 specifies that 20% of the data should be used for testing, and random\_state=42 sets a random seed for reproducibility.

# Scale the data

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

## Creating a StandardScaler object to scale the features. Feature scaling is crucial for some machine learning algorithms.

\* X\_train\_scaled and X\_test\_scaled store the scaled training and testing data, respectively.

# Perform feature selection based on correlation

num\_features\_to\_select = 5 # You can adjust this number as needed

selector = SelectKBest(score\_func=f\_classif, k=num\_features\_to\_select)

X\_train\_selected = selector.fit\_transform(X\_train\_scaled, y\_train)

X\_test\_selected = selector.transform(X\_test\_scaled)

##### These lines perform feature selection based on correlation using the SelectKBest method with f\_classif as the scoring function. It selects the top num\_features\_to\_select features with the highest correlation with the target variable.

\* X\_train\_selected and X\_test\_selected store the selected features for training and testing data.

# Increase max\_iter for Logistic Regression

rf\_model = RandomForestClassifier()

dt\_model = DecisionTreeClassifier()

knn\_model = KNeighborsClassifier()

nb\_model = GaussianNB()

lr\_model = LogisticRegression(max\_iter=1000)

svm\_model = SVC()

# In these lines, we're initializing various machine learning models for classification:

\* rf\_model: A Random Forest Classifier.

\* dt\_model: A Decision Tree Classifier.

\* knn\_model: A K-Nearest Neighbors Classifier.

\* nb\_model: A Gaussian Naive Bayes Classifier.

\* lr\_model: A Logistic Regression Classifier with an increased max\_iter parameter set to 1000 to avoid convergence issues.

\* svm\_model: A Support Vector Machine Classifier.

# Train the models

rf\_model.fit(X\_train\_selected, y\_train)

dt\_model.fit(X\_train\_selected, y\_train)

knn\_model.fit(X\_train\_selected, y\_train)

nb\_model.fit(X\_train\_selected, y\_train)

lr\_model.fit(X\_train\_selected, y\_train)

svm\_model.fit(X\_train\_selected, y\_train)

##### These lines train each of the machine learning models using the training data (X\_train\_selected and y\_train) that have been preprocessed and feature-selected.

##### Each model learns to make predictions based on the selected features and the corresponding target labels.

# Make predictions with the models

rf\_pred = rf\_model.predict(X\_test\_selected)

dt\_pred = dt\_model.predict(X\_test\_selected)

knn\_pred = knn\_model.predict(X\_test\_selected)

nb\_pred = nb\_model.predict(X\_test\_selected)

lr\_pred = lr\_model.predict(X\_test\_selected)

svm\_pred = svm\_model.predict(X\_test\_selected)

##### These lines use the trained models to make predictions on the testing data (X\_test\_selected) to evaluate their performance.

##### Predictions are stored in variables like rf\_pred, dt\_pred, knn\_pred, nb\_pred, lr\_pred, and svm\_pred.

# Evaluate the model performance

rf\_accuracy = accuracy\_score(y\_test, rf\_pred)

dt\_accuracy = accuracy\_score(y\_test, dt\_pred)

knn\_accuracy = accuracy\_score(y\_test, knn\_pred)

nb\_accuracy = accuracy\_score(y\_test, nb\_pred)

lr\_accuracy = accuracy\_score(y\_test, lr\_pred)

svm\_accuracy = accuracy\_score(y\_test, svm\_pred)

##### These lines calculate the accuracy of each model's predictions on the testing data using the accuracy\_score function.

##### The accuracy scores are stored in variables like rf\_accuracy, dt\_accuracy, knn\_accuracy, nb\_accuracy, lr\_accuracy, and svm\_accuracy.

# Print the accuracy of each model

print("Random Forest Accuracy:", rf\_accuracy)

print("Decision Tree Accuracy:", dt\_accuracy)

print("K-Nearest Neighbors Accuracy:", knn\_accuracy)

print("Naive Bayes Accuracy:", nb\_accuracy)

print("Logistic Regression Accuracy:", lr\_accuracy)

print("Support Vector Machine Accuracy:", svm\_accuracy)

# List of model names for the x-axis

model\_names = ["Random Forest", "Decision Tree", "K-Nearest Neighbors", "Naive Bayes", "Logistic Regression", "Support Vector Machine"]

# List of accuracy scores for the y-axis

accuracy\_scores = [rf\_accuracy, dt\_accuracy, knn\_accuracy, nb\_accuracy, lr\_accuracy, svm\_accuracy]

# Create a bar chart

plt.figure(figsize=(10, 6))

plt.bar(model\_names, accuracy\_scores, color='brown')

plt.xlabel('Machine Learning Models')

plt.ylabel('Accuracy')

plt.title('Accuracy Comparison of Machine Learning Models')

plt.ylim(0.6, 0.8) # Adjust the y-axis limits as needed

plt.xticks(rotation=45) # Rotate x-axis labels for better visibility

plt.grid()

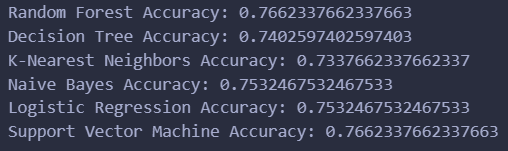
plt.show()

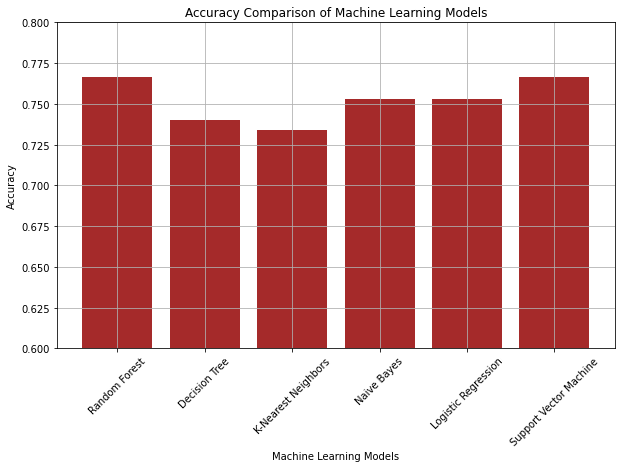
##### Finally, these lines print the accuracy of each model on the testing data. This provides you with a clear evaluation of how well each machine learning algorithm performed in predicting diabetes. The printed accuracies give you insights into the relative performance of each model.

**INPUT**

Diabetes dataset

**OUTPUT**





**RESULT**

The machine learning pipeline successfully trained and evaluated various classification models on the diabetes dataset. The Random Forest & SVM model achieved the highest **accuracy = 76%.**