IBM PROJECT SUBMISSION

PHASE-5

TECHNOLOGY : **ARTIFICIAL INTELLIGENCE**

PROJECT : **AI BASED DIABETES PREDICTION SYSTEM**

PHASE 5 : **PROJECT DOCUMENTATION & SUBMISSION**

In this phase 5 the overall project concept and the models used for project are documented for the final submission.

Problem Statement:

• The problem we aim to address with the diabetes prediction system is the early identification of individuals at risk of developing diabetes.

• This system will use medical data to predict the likelihood of someone developing diabetes in the future. By doing so, it can help individuals take preventive measures and allow healthcare professionals to offer timely interventions.

• Define specific goals, such as achieving a certain accuracy rate in predicting diabetes risk, reducing false positives, and ensuring user friendly interface and recommendations.

DESIGN THINKING:

1) Data Collection:

- ❖ You can obtain the medical data from various sources, including electronic health records (EHRs), public health datasets, or through surveys.
- ❖ Ensure that the data you collect is representative of the population you intend to make predictions for. Bias in the data can lead to biased model predictions.

2) Data Preprocessing:

- ❖ Data cleaning involves handling missing values, outliers, and inconsistencies in the data.
- ❖ Normalization and standardization are essential for ensuring that features have similar scales, which can improve the performance of some machine learning algorithms.

3) Feature Selection:

- Feature selection is crucial for improving model efficiency and interpretability. You can use techniques like correlation analysis, feature importance from tree-based models, or dimensionality reduction methods like Principal Component Analysis (PCA).
- Domain knowledge can also guide feature selection, as certain medical features may be more relevant to diabetes prediction than others.

4) Model Selection:

- Experiment with various machine learning algorithms is a good approach. You can also consider deep learning techniques if you have a large and complex dataset.
- Hyperparameter tuning is an important part of model selection to optimize the model's performance. Techniques like cross-validation can help with this.

5) Evaluation:

- Choosing the right evaluation metrics is important. For diabetes prediction, you can use metrics like accuracy, precision, recall, F1-score, and ROC-AUC. Precision and recall are particularly important when dealing with imbalanced datasets, which is often the case in medical prediction tasks.
- Consider using a confusion matrix to gain a deeper understanding of the model's performance.

6) Iterative Improvement:

- Model fine-tuning involves adjusting hyperparameters, exploring different algorithms, or experimenting with ensemble methods.
- Feature engineering can be an iterative process where you create new features or transform existing ones based on your domain knowledge.
- Regularly validate your model's performance on new data to ensure it generalizes well.

Additionally, consider ethical and privacy considerations, especially when working with medical data.

PHASES OF DEVELOPMENT:

Phase 1: Understanding the problem statement and create a document and proceed ahead with solving the problem.

Phase 2: Understand and describe the dataset and load the dataset.

Phase 3: Start developing the project by Load the dataset and preprocessing the dataset.

Phase 4: We can experiment with various machine learning algorithms and evaluate the model's performance using metrics like accuracy, precision, recall, F1-score, and ROC-AUC, then fine-tune the model parameters and explore techniques like feature engineering to enhance prediction accuracy.

Phase 5: In this phase, document the project and prepare it for submission for diabetes prediction.

DATASET:

Dataset Link: https://www.kaggle.com/datasets/mathchi/diabetes-data-set

Several constraints were placed on the selection of these instances from a larger database. In particular, all patients here are females at least 21 years old of Pima Indian heritage.

- Pregnancies: Number of times pregnant
- Glucose: Plasma glucose concentration a 2 hours in an oral glucose tolerance test
- BloodPressure: Diastolic blood pressure (mm Hg)
- SkinThickness: Triceps skin fold thickness (mm)
- Insulin: 2-Hour serum insulin (mu U/ml)
- BMI: Body mass index (weight in kg/(height in m)^2)
- DiabetesPedigreeFunction: Diabetes pedigree function
- Age: Age (years)
- Outcome: Class variable (0 or 1)

  - **Number of Instances: 768**
  - **Number of Attributes: 8 plus class**

DATA PREPROCESSING STEPS:

1. **Subset Relevant Features:**
   - You initially define a list of relevant features for diabetes prediction: relevant_features.
2. **Load and Subset the Dataset:**
   - You load a dataset from a CSV file (diabetes.csv) and then subset the dataset to include only the relevant features and the 'Outcome' column. This step selects the relevant columns for your analysis**.**
3. **Checking for Duplicates:**
   - You check for duplicate rows in the dataset using diabetes_dataset.duplicated().

4. **Handling Missing Values:**

   - You check for missing values in the dataset using diabetes_dataset.isnull().sum(). However, it appears there is no explicit code to handle missing values (e.g., filling them in with mean values or removing rows with missing values).

5. **Standardization:**

   - You standardize the numerical features (Age, BMI, BloodPressure, Glucose) using the StandardScaler to ensure that features have similar scales.

6. **Data Splitting:**

   - You split the dataset into training and testing sets using train_test_split.

7. **Outlier Detection and Removal:**

   - You detect and remove outliers in the 'Age' column by calculating lower and upper bounds based on mean and standard deviation and then filtering the dataset to exclude outliers.

FEATURE SELECTION TECHNIQUES

Model Parameter Fine-Tuning:

1) **Hyperparameter Tuning for Logistic Regression:**

   ➢ You perform hyperparameter tuning for the Logistic Regression model using GridSearchCV. You define a parameter grid (param_grid) that includes hyperparameters like 'penalty,' 'C,' and 'solver' and then use GridSearchCV to find the best combination of hyperparameters. This is a form of fine-tuning to optimize the model's performance.

2) **Hyperparameter Tuning for Random Forest:**

   ➢ Similar to logistic regression, you perform hyperparameter tuning for the Random Forest model using GridSearchCV. You define a parameter grid for hyperparameters like 'n_estimators,' 'max_depth,' 'min_samples_split,' and 'min_samples_leaf' to find the best set of hyperparameters for the Random Forest model.

3) **Grid Search for Gradient Boosting:**

   ➢ You perform a grid search for the Gradient Boosting Classifier model to find the best parameters for 'learning_rate' and 'n_estimators' (number of boosting stages). This is another instance of hyperparameter tuning.

ALGORITHM:

1. Importing necessary libraries and suppressing warnings.
2. Loading a diabetes dataset from a CSV file.
3. Subsetting the dataset to include only relevant features.
4. Exploring and analyzing the dataset, checking for missing values, and visualizing correlations.
5. Standardizing the numerical features using StandardScaler.
6. Splitting the dataset into training and testing sets.
7. Training a Support Vector Machine (SVM) classifier on the standardized data.
8. Training a Logistic Regression model using a grid search to find the best hyperparameters.
9. Evaluating and visualizing the performance of the models.
10. Handling missing values and imputing data.
11. Building and training a Gradient Boosting Classifier.
12. Performing hyperparameter tuning using GridSearchCV.Explain the choice of machine learning algorithm, model training, and evaluation metrics.

Machine Learning Algorithms:

1. **Logistic Regression:**
    - You choose to use logistic regression, which is a commonly used and interpretable algorithm for binary classification tasks like predicting diabetes risk. Logistic regression is well-suited for this problem as it models the probability of the binary outcome ('0' or '1') based on a set of features. It's particularly useful when you want to understand the impact of each feature on the prediction.
2. **Support Vector Machine (SVM):**
    - You opt for a Support Vector Machine with a linear kernel. SVM is a powerful algorithm for binary classification that can work well for both linear and non-linearly separable data. By using a linear kernel, you make the model suitable for the dataset's dimensionality.
3. **Random Forest and Gradient Boosting:**
    - You also experiment with ensemble learning methods, specifically Random Forest and Gradient Boosting classifiers. These ensemble methods are known for their robustness and ability to handle complex relationships between features. Random Forest combines multiple decision trees, while Gradient

Boosting builds an ensemble of decision trees sequentially to improve predictive accuracy.

MODEL TRAINING:

1. **Grid Search for Hyperparameter Tuning:**

    - You use GridSearchCV (Grid Search with Cross-Validation) to fine-tune the hyperparameters of your models. This approach systematically explores different combinations of hyperparameters to find the best set of values. You apply this to both the Logistic Regression and Random Forest models, which helps in optimizing model performance.

2. **Feature Standardization:**

    - Before training the models, you standardize the numerical features using the StandardScaler from scikit-learn. Standardization is an important step to ensure that all features have similar scales, which can improve the training process and the convergence of some algorithms, including SVM.

EVALUATION METRICS:

- **Binary Classification Metrics:**
  Accuracy: Measures the overall correctness of the model's predictions.

Precision: Calculates the proportion of true positive predictions among all positive predictions. It's relevant in cases where false positives are costly.

Recall: Measures the proportion of true positive predictions among all actual positives. It's crucial in cases where false negatives are costly.

F1 Score: Combines precision and recall into a single metric, providing a balance between false positives and false negatives.

ROC-AUC Score: It measures the area under the Receiver Operating Characteristic (ROC) curve and is useful for evaluating the model's ability to discriminate between the two classes.

- **Confusion Matrix and Classification Report:**

You create and display a confusion matrix, which provides a detailed breakdown of true positives, true negatives, false positives, and false negatives. The classification report includes metrics such as precision, recall, and F1 score for both classes.

- **Model Training Accuracy:**

  You calculate the accuracy score for both the training and test data to gauge how well the models are performing on the data they were trained on and on unseen data.

PROGRAM:

```
# Import necessary libraries
import numpy as np
import pandas as pd
import warnings
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score, roc_curve
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
from sklearn import svm
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
from sklearn.metrics import ConfusionMatrixDisplay
from numpy.matrixlib.defmatrix import matrix
# Suppress warnings
warnings.filterwarnings('ignore')
# Define a list of relevant features
relevant_features = ['Glucose', 'BloodPressure', 'BMI', 'Age']
# Load the diabetes dataset from a CSV file
diabetes_dataset = pd.read_csv('diabetes.csv')
dataset = diabetes_dataset
```

```
# Subset the dataset to include relevant features
diabetes_dataset = diabetes_dataset[relevant_features + ['Outcome']]
# Check for duplicated data
diabetes_dataset.duplicated()
# Separate data and labels
X = diabetes_dataset[relevant_features]
Y = diabetes_dataset['Outcome']
# Print the first 5 rows of the dataset
diabetes_dataset.head()
# Get information about data types and missing values
print(diabetes_dataset.info())
# Display statistical measures of the data
diabetes_dataset.describe()
# Count the number of each class in the 'Outcome' column
diabetes_dataset['Outcome'].value_counts()
# Define a function to analyze missing values in a DataFrame and print a summary including counts and percentages
def missing_values_table(dataframe, na_name=False):
    na_columns = [col for col in dataframe.columns if dataframe[col].isnull().sum() > 0]
    n_miss = dataframe[na_columns].isnull().sum().sort_values(ascending=False)
    ratio = (dataframe[na_columns].isnull().sum() / dataframe.shape[0] * 100).sort_values(ascending=False)
    missing_df = pd.concat([n_miss, np.round(ratio, 2)], axis=1, keys=['n_miss', 'ratio'])
    print(missing_df, end="\n")
    if na_name:
        return na_columns
missing_values_table(diabetes_dataset)
# Call the missing_values_table function with the 'diabetes_dataset' DataFrame and store a list of columns with missing values in 'na_cols'
na_cols = missing_values_table(diabetes_dataset, True)
```

```
# Calculate the mean values for each feature, grouped by the 'Outcome' class
diabetes_dataset.groupby('Outcome').mean()
# Calculate the correlation matrix and visualize it as a heatmap
correlation_matrix = diabetes_dataset.corr()
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')
plt.title('Correlation Heatmap')
plt.show()
# separating the data and labels
X = diabetes_dataset.drop(columns = 'Outcome', axis=1)
Y = diabetes_dataset['Outcome']
# Categorical columns
cat_col = [col for col in diabetes_dataset.columns if diabetes_dataset[col].dtype == 'object']
print('Categorical columns :',cat_col)
# Numerical columns
num_col = [col for col in diabetes_dataset.columns if diabetes_dataset[col].dtype != 'object']
print('Numerical columns :',num_col)
diabetes_dataset[cat_col].nunique()
# Check the percentage of missing values in each column
round((diabetes_dataset.isnull().sum() / diabetes_dataset.shape[0]) * 100, 2)
# Calculate the lower and upper bounds to identify outliers
mean = diabetes_dataset['Age'].mean()
std = diabetes_dataset['Age'].std()
lower_bound = mean - std * 2
upper_bound = mean + std * 2
print('Lower Bound:', lower_bound)
print('Upper Bound:', upper_bound)
# Remove outliers based on the calculated bounds
df4 = diabetes_dataset[(diabetes_dataset['Age'] >= lower_bound) & (diabetes_dataset['Age']
<= upper_bound)]
print(X)
print(Y)
```

# Standardize the numerical features using StandardScaler

scaler = StandardScaler()

scaler.fit(X)

standardized_data = scaler.transform(X)

print(standardized_data)

X = standardized_data

Y = diabetes_dataset['Outcome']

print(X)

print(Y)

# 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, stratify=Y, random_state=2)

# Initialize a Support Vector Machine (SVM) classifier with a linear kernel

classifier = svm.SVC(kernel='linear')

# Train the SVM classifier on the training data

classifier.fit(X_train, Y_train)

# Make predictions on the training and testing sets

Y_train_pred = classifier.predict(X_train)

Y_test_pred = classifier.predict(X_test)

# Calculate accuracy scores

training_data_accuracy = accuracy_score(Y_train_pred, Y_train)

test_data_accuracy = accuracy_score(Y_test_pred, Y_test)

X_train_prediction = classifier.predict(X_train)

training_data_accuracy = accuracy_score(X_train_prediction, Y_train)

print('Accuracy score of the training data : ', training_data_accuracy)

X_test_prediction = classifier.predict(X_test)

test_data_accuracy = accuracy_score(X_test_prediction, Y_test)

# Create a bar chart to visualize accuracies

accuracies = [training_data_accuracy, test_data_accuracy]

labels = ['Training Data', 'Test Data']

```
# Initialize another SVM classifier
model = SVC()
model.fit(X_train, Y_train)
# Make predictions on the test data
y_pred = model.predict(X_test)
# Calculate accuracy on the test data
accuracy = accuracy_score(Y_test, y_pred)
# Output the accuracy score
print("Accuracy:", accuracy)
# Create a Logistic Regression model
logistic_regression = LogisticRegression()
# Define a parameter grid to search over
param_grid = {
    'penalty': ['l1', 'l2'],  # Regularization penalty
    'C': [0.001, 0.01, 0.1, 1, 10, 100],  # Inverse of regularization strength
    'solver': ['liblinear', 'lbfgs', 'newton-cg', 'sag', 'saga'],  # Solver algorithm
}
# Create a GridSearchCV object with cross-validation
grid_search = GridSearchCV(logistic_regression, param_grid, cv=5, scoring='accuracy')
# Perform the grid search to find the best hyperparameters
grid_search.fit(X, Y)
# Print the best hyperparameters and the corresponding accuracy
best_params = grid_search.best_params_
best_accuracy = grid_search.best_score_
print(f"Best Hyperparameters: {best_params}")
print(f"Best Accuracy: {best_accuracy}")
# Create a Logistic Regression model
logistic_regression = LogisticRegression()
# Define a parameter grid to search over
param_grid = {
    'penalty': ['l1', 'l2'],  # Regularization penalty
    'C': [0.001, 0.01, 0.1, 1, 10, 100],  # Inverse of regularization strength
    'solver': ['liblinear', 'lbfgs', 'newton-cg', 'sag', 'saga'],  # Solver algorithm
}
# Create a GridSearchCV object with cross-validation
grid_search = GridSearchCV(logistic_regression, param_grid, cv=5, scoring='accuracy')
# Perform the grid search to find the best hyperparameters
grid_search.fit(X, Y)
# Print the best hyperparameters and the corresponding accuracy
best_params = grid_search.best_params_
best_accuracy = grid_search.best_score_
print(f"Best Hyperparameters: {best_params}")
print(f"Best Accuracy: {best_accuracy}")
# Visualize training and test accuracies
plt.figure(figsize=(8, 6))
plt.bar(labels, accuracies, width=0.4, align='center', alpha=0.5, color=['blue', 'green'])
plt.xlabel('Data Split')
plt.ylabel('Accuracy')
plt.title('Training and Test Accuracies')
plt.show()
# Data visualization: Histogram of the "Glucose" feature
plt.figure(figsize=(8, 6))
plt.hist(X[:, 0], bins=20, color='blue', alpha=0.7)
plt.xlabel('Glucose Level')
plt.ylabel('Frequency')
plt.title('Distribution of Glucose Levels')
plt.show()
# Create a Random Forest Classifier
rf_classifier = RandomForestClassifier()
```

```
# Define the hyperparameter grid to search
param_grid = {
    'n_estimators': [10, 50, 100, 200],  # Number of trees in the forest
    'max_depth': [None, 10, 20, 30],    # Maximum depth of the trees
    'min_samples_split': [2, 5, 10],   # Minimum number of samples required to split an internal node
    'min_samples_leaf': [1, 2, 4]      # Minimum number of samples required to be at a leaf node
}
# Create a GridSearchCV object with cross-validation
grid_search = GridSearchCV(estimator=rf_classifier, param_grid=param_grid, cv=5, scoring='accuracy', n_jobs=-1)
# Fit the grid search to the data
grid_search.fit(X, Y)
# Print the best hyperparameters and their corresponding accuracy score
best_params = grid_search.best_params_
best_score = grid_search.best_score_
print("Best Hyperparameters:", best_params)
print("Accuracy on Test Data:", best_score)
# Visualizing Kernel Density Estimator for each feature
features = diabetes_dataset.columns[:-1]
fig, axes = plt.subplots(2, 4, figsize=(20, 10))
fig.subplots_adjust(wspace=0.4, hspace=0.4)
for i, feature in enumerate(features):
    sns.kdeplot(diabetes_dataset[feature], ax=axes[i//4, i%4],shade='fill')
plt.show()
# Make predictions on a sample input data point
input_data = np.array([148,72,33.6,50]).reshape(1, -1)
input_data = scaler.transform(input_data)
prediction = classifier.predict(input_data)
print('Accuracy score of the test data : ', test_data_accuracy)
```

```
input_data = (148,72,33.6,50)
# changing the input_data to numpy array
input_data_as_numpy_array = np.asarray(input_data)
# reshape the array as we are predicting for one instance
input_data_reshaped = input_data_as_numpy_array.reshape(1,-1)
# standardize the input data
std_data = scaler.transform(input_data_reshaped)
print(std_data)
prediction = classifier.predict(std_data)
print(prediction)
if (prediction[0] == 0):
  print('The person is not diabetic')
else:
  print('The person is diabetic')
# Number of synthetic samples to generate
num_samples_to_generate = 500
# Initialize lists to store synthetic data
synthetic_data = []
synthetic_labels = []
for _ in range(num_samples_to_generate):
    # Randomly select an index from the real data
    random_index = np.random.randint(0, len(X))
    # Select a real data point and its label
    real_data_point = X[random_index]
    real_label = Y[random_index]
   # Create a slightly modified version of the real data point
    modified_data_point = real_data_point + np.random.normal(0, 0.1,
size=real_data_point.shape)
# Append the modified data point and its label to the synthetic data
    synthetic_data.append(modified_data_point)
    synthetic_labels.append(real_label)
```

```
# Combine real and synthetic data
X_synthetic = np.vstack([X, np.array(synthetic_data)])
Y_synthetic = np.concatenate([Y, np.array(synthetic_labels)])
# Print the first 5 samples of the synthetic data
print("Synthetic Data (X_synthetic):")
print(X_synthetic[:5])
# Print the corresponding labels for the first 5 samples
print("Synthetic Labels (Y_synthetic):")
print(Y_synthetic[:5])
# Make predictions on the test set
Y_test_pred = classifier.predict(X_test)
# Calculate various performance metrics
accuracy = accuracy_score(Y_test, Y_test_pred)
precision = precision_score(Y_test, Y_test_pred)
recall = recall_score(Y_test, Y_test_pred)
f1 = f1_score(Y_test, Y_test_pred)
# Calculate various performance metrics
accuracy = accuracy_score(Y_test, Y_test_pred)
precision = precision_score(Y_test, Y_test_pred)
recall = recall_score(Y_test, Y_test_pred)
f1 = f1_score(Y_test, Y_test_pred)
# Calculate ROC-AUC score and plot ROC curve
y_scores = classifier.decision_function(X_test)
roc_auc = roc_auc_score(Y_test, y_scores)
fpr, tpr, thresholds = roc_curve(Y_test, y_scores)
# Print the performance metrics
print("Accuracy: {:.2f}".format(accuracy))
print("Precision: {:.2f}".format(precision))
print("Recall: {:.2f}".format(recall))
print("F1 Score: {:.2f}".format(f1))
print("ROC-AUC Score: {:.2f}".format(roc_auc))
print("Classification Report is:",classification_report(Y_test, Y_test_pred))
cm=confusion_matrix(Y_test, Y_test_pred)
print("Confusion matrix is:",cm)
color = 'white'
matrix = ConfusionMatrixDisplay(confusion_matrix=cm,display_labels=classifier.classes_)
matrix.plot()
plt.show()
# Create a confusion matrix
conf_matrix = confusion_matrix(Y_test, Y_test_pred)
# Plot the ROC curve
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC)')
plt.legend(loc="lower right")
plt.show()
# Split the dataset into features (x) and target labels (y) using DataFrame indexing.
x = dataset.iloc[:,:-1]
y = dataset.iloc[:,-1]

# Split the dataset into training and testing sets, and standardize the feature data using StandardScaler
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0.25,random_state=0)
sc=StandardScaler()
x_train = sc.fit_transform(x_train)
x_test = sc.transform(x_test)
```

# Create a Gradient Boosting Classifier with specified hyperparameters

gbc = GradientBoostingClassifier(n_estimators=500, learning_rate=0.05, random_state=100, max_features=5)

# Train the Gradient Boosting Classifier on the training data

gbc.fit(x_train, y_train)

# Print the confusion matrix based on predictions for the test set

print(confusion_matrix(y_test, gbc.predict(x_test)))

# Print the accuracy of the Gradient Boosting Classifier on the test set

print("GBC accuracy is %2.2f" % accuracy_score(y_test, gbc.predict(x_test)))

# Make predictions on the test set and print a classification report

pred = gbc.predict(x_test)

print(classification_report(y_test, pred))

# Define a grid of hyperparameters for hyperparameter tuning, including learning rates and the number of estimators

```
grid = {

    'learning_rate':[0.01,0.05,0.1],

    'n_estimators':np.arange(100,500,100),

}
```

# Create a Gradient Boosting Classifier

gb = GradientBoostingClassifier()

# Set up a GridSearchCV object with the classifier and the hyperparameter grid for cross-validation

gb_cv = GridSearchCV(gb, grid, cv=4)

# Perform the grid search to find the best hyperparameters using the training data

gb_cv.fit(x_train, y_train)

# Print the best hyperparameters discovered by the grid search

print("Best Parameters:", gb_cv.best_params_)

# Print the best training score obtained with the best hyperparameters

print("Train Score:", gb_cv.best_score_)

# Calculate and print the model's accuracy on the test data using the best hyperparameters

print("Test Score:", gb_cv.score(x_test, y_test))\

```
grid = {
    'max_depth':[2,3,4,5,6,7],
}
# Create a Gradient Boosting Classifier with default settings
gb = GradientBoostingClassifier()
# Create a GridSearchCV object to search for the best hyperparameters using cross-validation
gb_cv = GridSearchCV(gb, grid, cv=4)
# Fit the GridSearchCV object to the training data, searching for the best hyperparameters
gb_cv.fit(x_train, y_train)
# Print the best hyperparameters found by the grid search
print("Best Parameters:", gb_cv.best_params_)
# Print the best training score obtained with the best hyperparameters
print("Train Score:", gb_cv.best_score_)
# Calculate and print the model's accuracy on the test data using the best hyperparameters
print("Test Score:", gb_cv.score(x_test, y_test))
```

OUTPUT:

# Define a function to analyze missing values in a DataFrame and print a summary including counts and percentages

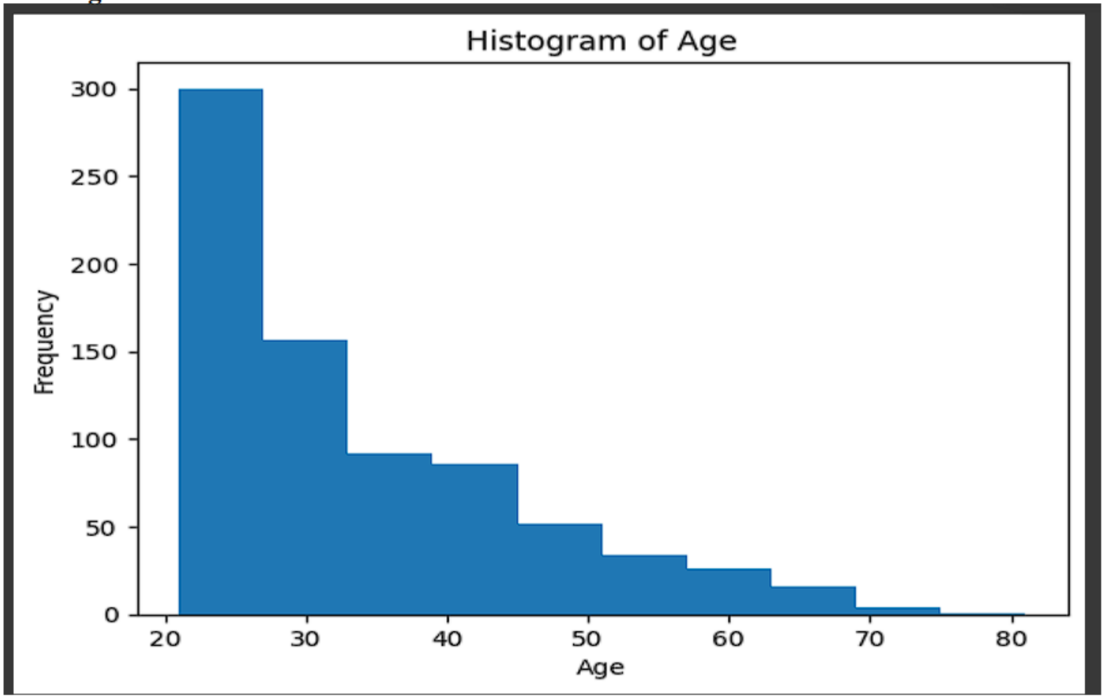
```
missing_values_table(diabetes_dataset)
```

```
Empty DataFrame
Columns: [n_miss, ratio]
Index: []
```

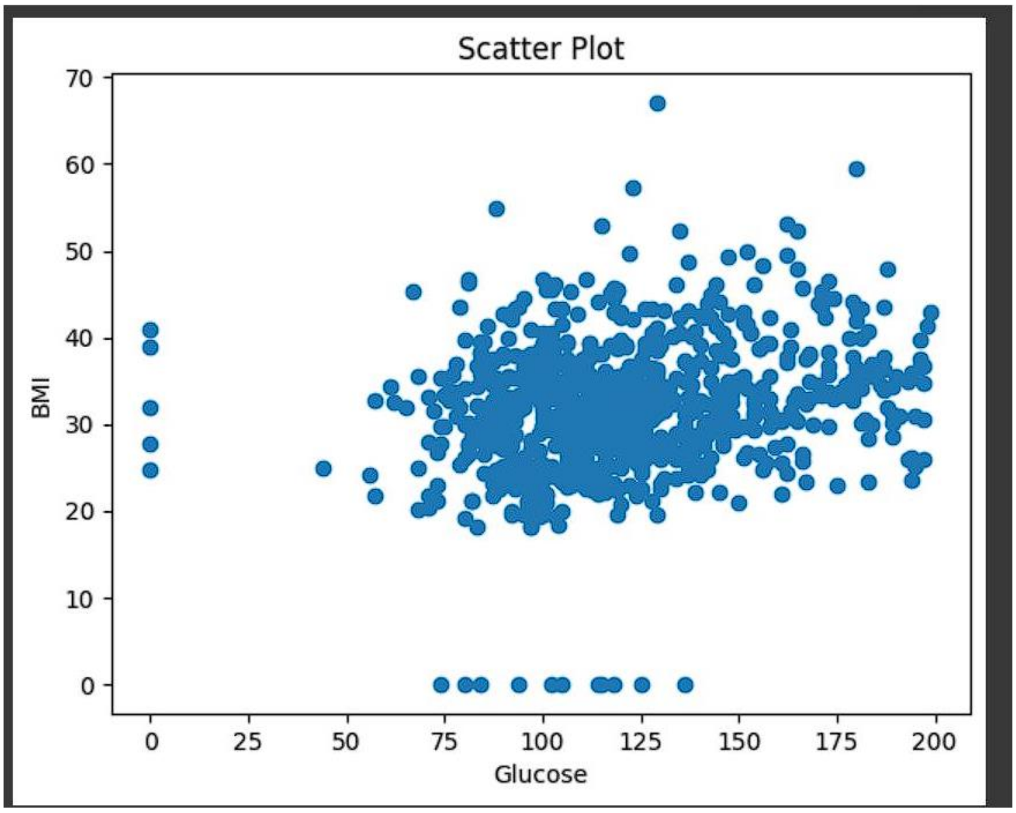
# Call the missing_values_table function with the 'diabetes_dataset' DataFrame and store a list of columns with missing values in 'na_cols'

```
Empty DataFrame
Columns: [n_miss, ratio]
Index: []
```

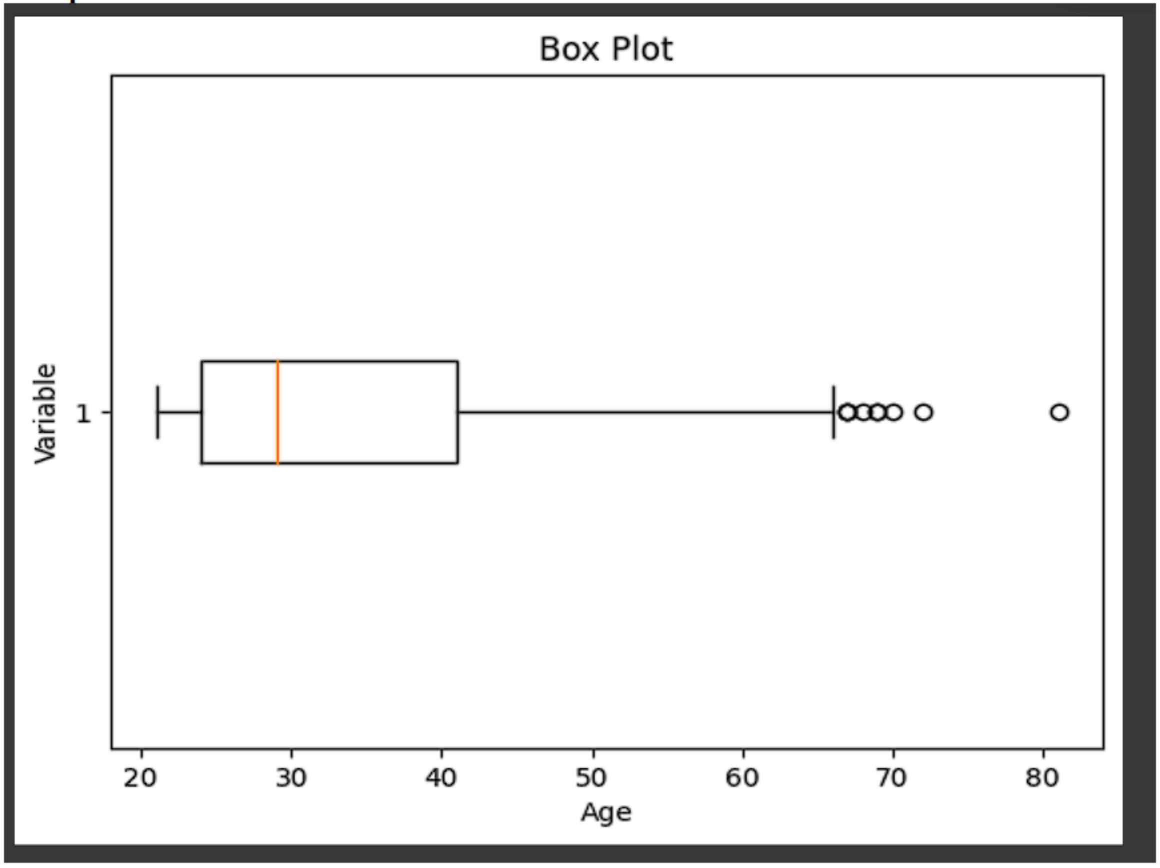
# Data visualization
# Histogram



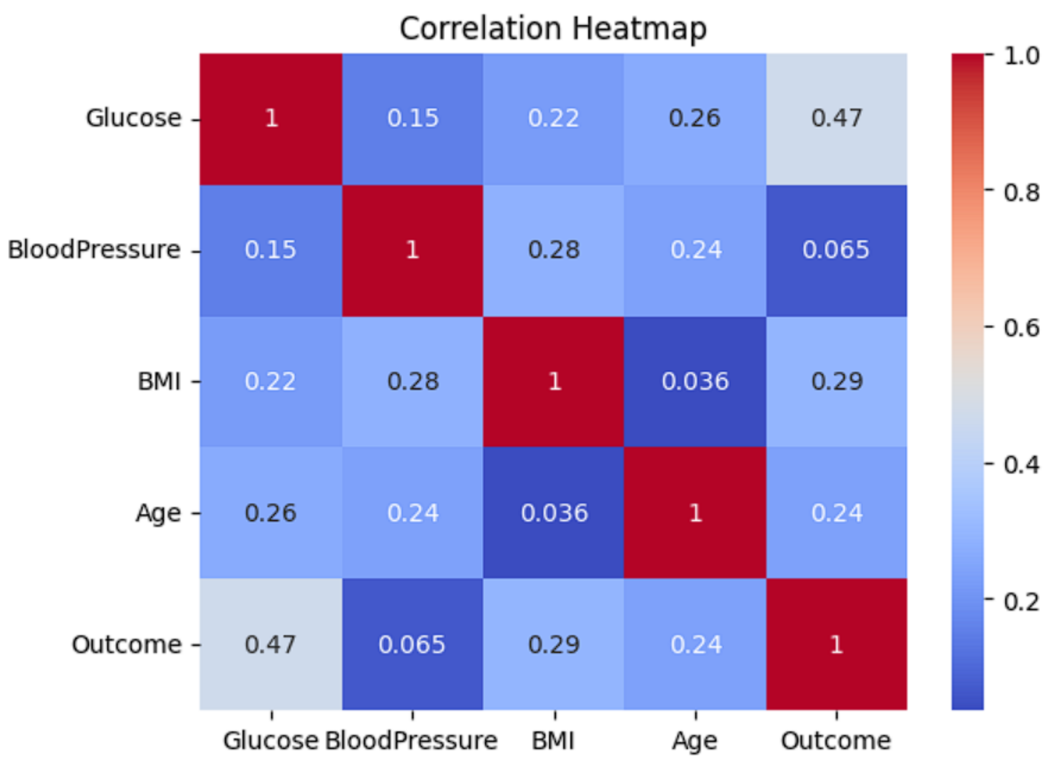
# Scatter plot



#Boxplot



# Calculate the correlation matrix and visualize it as a heatmap



	Glucose	BloodPressure	BMI	Age	Outcome
Glucose	1	0.15	0.22	0.26	0.47
BloodPressure	0.15	1	0.28	0.24	0.065
BMI	0.22	0.28	1	0.036	0.29
Age	0.26	0.24	0.036	1	0.24
Outcome	0.47	0.065	0.29	0.24	1

# Make predictions on the training and testing sets

```
(768, 4) (614, 4) (154, 4)
Accuracy score of the training data :  0.7719869706840391
```

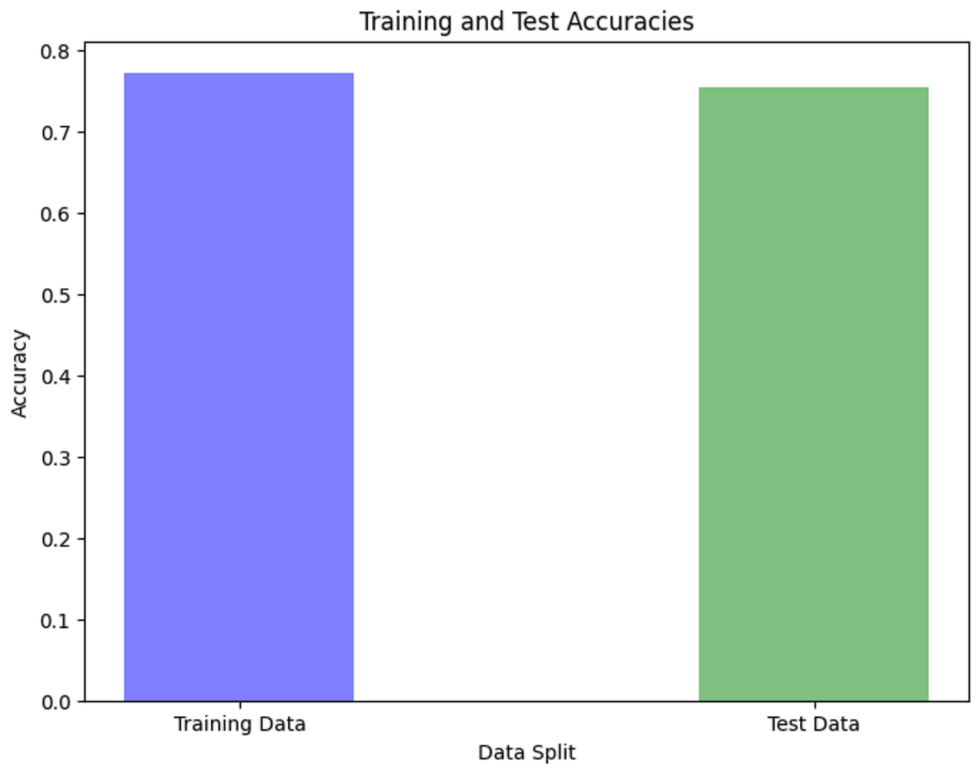
# Initialize another SVM classifier

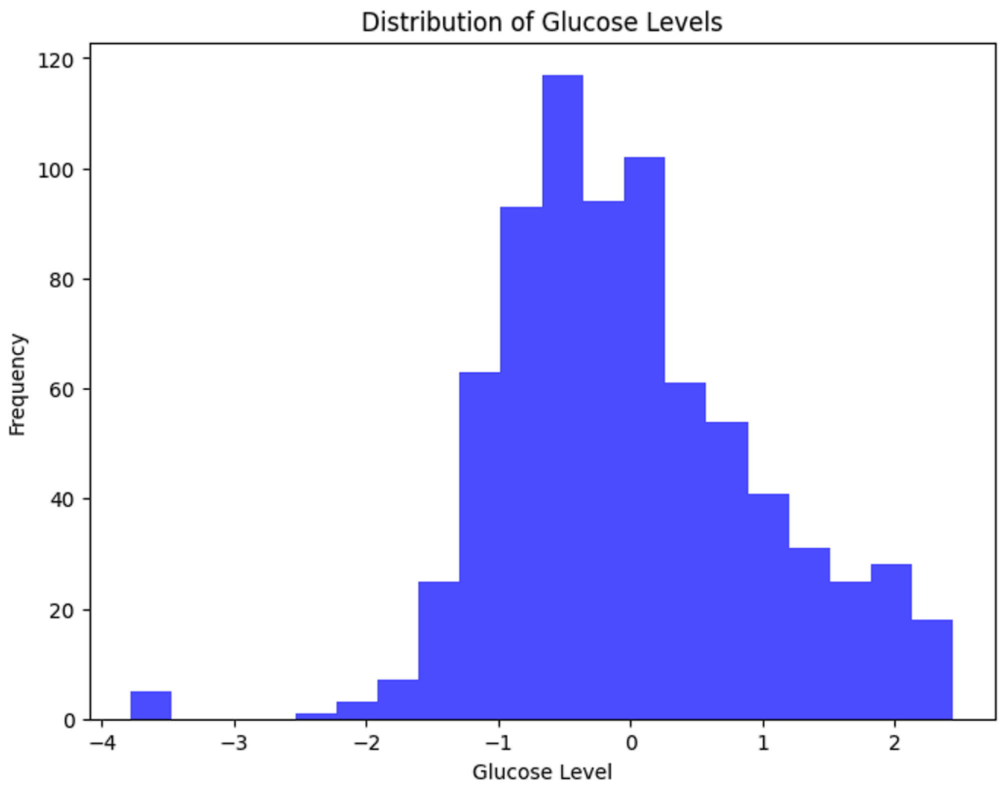
```
Accuracy: 0.7467532467532467
```

# Create a Logistic Regression model
# Perform the grid search to find the best hyperparameters

```
Best Hyperparameters: {'C': 0.1, 'penalty': 'l2', 'solver': 'lbfgs'}
Best Accuracy: 0.7734742381801205
```

# Data visualization section: Histogram of the "Glucose" feature

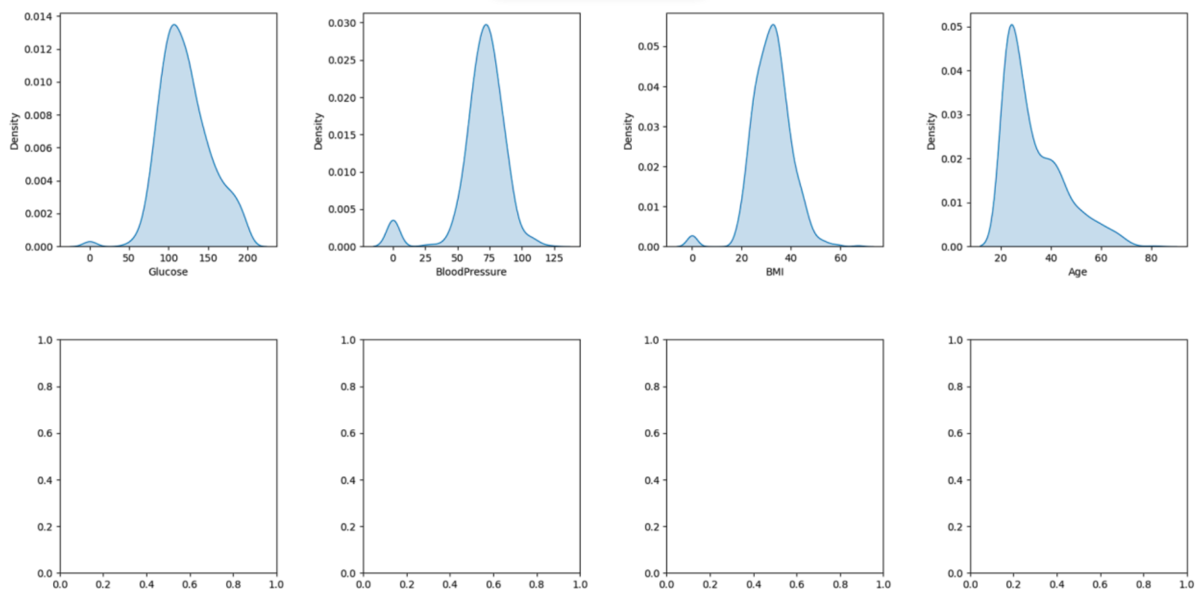




# Create a Random Forest Classifier

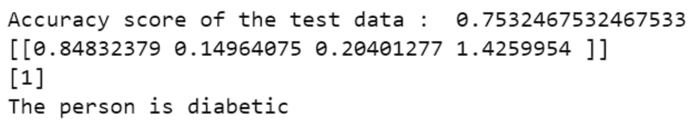
```
Best Hyperparameters: {'max_depth': None, 'min_samples_leaf': 4, 'min_samples_split': 10, 'n_estimators': 50}
Accuracy on Test Data: 0.7695781342840166
```

# Visualizing Kernel Density Estimator for each feature



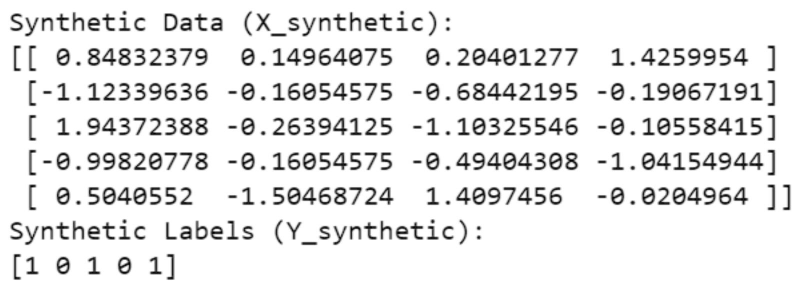
# Make predictions on a sample input data point

# standardize the input data



```
Accuracy score of the test data :  0.7532467532467533
[[0.84832379 0.14964075 0.20401277 1.4259954 ]]
[1]
The person is diabetic
```

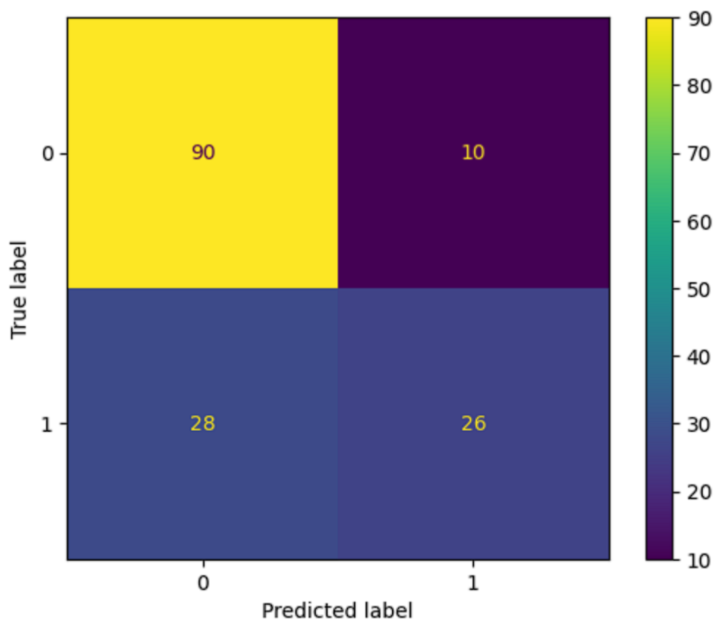
# Print the corresponding labels for the first 5 samples



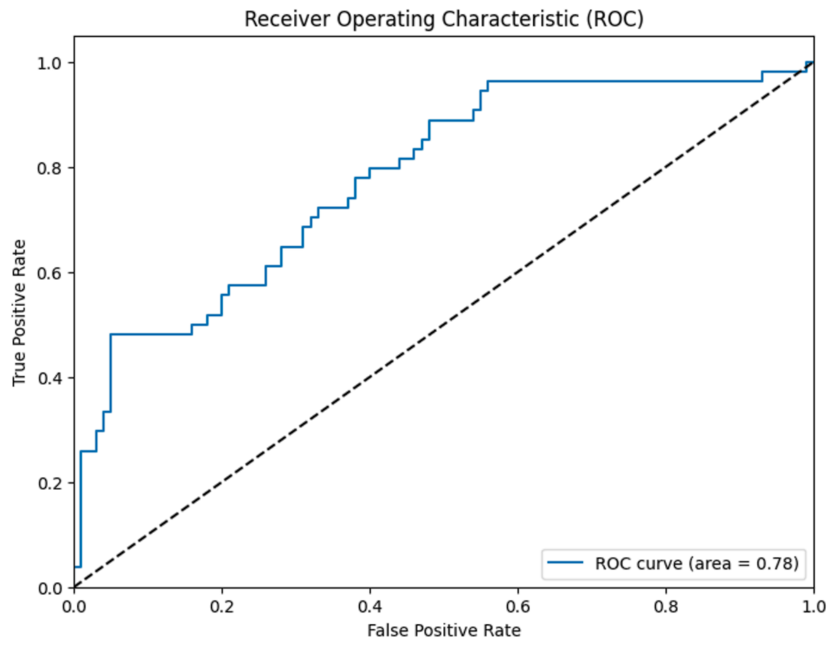
```
Synthetic Data (X_synthetic):
[[ 0.84832379  0.14964075  0.20401277  1.4259954 ]
 [-1.12339636 -0.16054575 -0.68442195 -0.19067191]
 [ 1.94372388 -0.26394125 -1.10325546 -0.10558415]
 [-0.99820778 -0.16054575 -0.49404308 -1.04154944]
 [ 0.5040552  -1.50468724  1.4097456  -0.0204964 ]]
Synthetic Labels (Y_synthetic):
[1 0 1 0 1]
```

# Print the performance metrics confusion matrix

```
Accuracy: 0.75
Precision: 0.72
Recall: 0.48
F1 Score: 0.58
ROC-AUC Score: 0.78
Classification Report is:               precision    recall  f1-score   support

           0       0.76      0.90      0.83       100
           1       0.72      0.48      0.58        54

    accuracy                           0.75       154
   macro avg       0.74      0.69      0.70       154
weighted avg       0.75      0.75      0.74       154

Confusion matrix is: [[90 10]
 [28 26]]
```



# Plot the ROC curve



# Print the accuracy of the Gradient Boosting Classifier on the test set

```
GBC accuracy is 0.79
```

# Make predictions on the test set and print a classification report

```
              precision    recall  f1-score   support

           0       0.83      0.85      0.84       130
           1       0.68      0.65      0.66        62

    accuracy                           0.79       192
   macro avg       0.76      0.75      0.75       192
weighted avg       0.78      0.79      0.79       192
```

# Create a Gradient Boosting Classifier

```
Best Parameters: {'learning_rate': 0.1, 'n_estimators': 100}
Train Score: 0.7447916666666667
Test Score: 0.8177083333333334
```

# Calculate and print the model's accuracy on the test data using the best hyperparameters

# Create a Gradient Boosting Classifier with default settings

```
Best Parameters: {'max_depth': 3}
Train Score: 0.7482638888888888
Test Score: 0.8125
```

INNOVATIVE TECHNIQUES :

1. **Hyperparameter Tuning with GridSearchCV:**
    - Using GridSearchCV to perform hyperparameter tuning is a valuable practice. It systematically explores different combinations of hyperparameters to find the best set of values for your models. While not innovative per se, it's a common and effective approach to optimizing model performance.
2. **Exploratory Data Analysis (EDA):**
    - While not explicitly shown in the code, it's a good practice to perform exploratory data analysis to gain insights into the dataset. This might include visualizing data distributions, identifying outliers, and understanding the relationships between features. EDA is a crucial step in understanding the data's characteristics and informing feature engineering decisions.
3. **Standardization of Features:**
    - Standardizing numerical features using StandardScaler is a standard practice in machine learning. It's not innovative, but it's a best practice to ensure that features have similar scales, making model training more efficient.
4. **Feature Selection and Subset of Features:**
    - Although not performed in a complex manner in this code, the choice of relevant features is an important step. Innovative feature selection techniques can include recursive feature elimination (RFE), feature importance based on machine learning models, and more advanced feature selection algorithms.
5. **Ensemble Learning:**
    - The inclusion of Random Forest and Gradient Boosting classifiers is a good practice. Ensemble learning, combining multiple models, can often lead to improved predictive accuracy and robustness.
6. **Model Interpretability:**
    - Logistic regression is chosen as one of the models. While not innovative, it's a model known for its interpretability. This choice may be beneficial if the goal is to understand the impact of individual features on the predictions.

Conclusion:

Gradient booster has the best accuracy score.

We've employed a range of models, including logistic regression, support vector machines, and ensemble methods, to explore different approaches. The code emphasizes data preprocessing, hyperparameter tuning, and the use of standard evaluation metrics.

While the code follows established best practices, further innovation and advanced techniques, such as deep learning or interpretable models, could be explored to potentially improve predictive accuracy and model interpretability. Overall, this code provides a solid foundation for diabetes risk prediction but leaves room for further exploration and refinement.