PHASE-4

ARTIFICIAL INTELLIGENCE

Project Title: **AI-BASED DIABETES PREDICTION SYSTEM**

INTRODUCTION:

A crucial part of the machine learning workflow involves selecting an appropriate machine learning algorithm, training the model, and evaluating its performance.

- **Selecting a machine learning algorithm**
- **Training the model**
- **Evaluating its performance**

Evaluating its performance: We will select relevant features that can impact diabetes risk prediction. We can experiment with various machine learning algorithms like Logistic Regression, Random Forest, and Gradient Boosting. We will evaluate the model's performance using metrics like accuracy, precision, recall, F1-score, and ROC-AUC. We will fine-tune the model parameters and explore techniques like feature engineering to enhance prediction accuracy.

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

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.

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

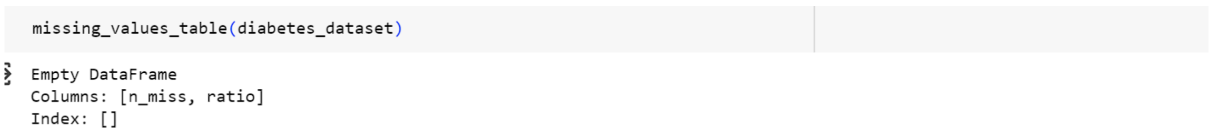
print("Train Score:", gb_cv.best_score_)

# 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))

<u>**Output:**</u>

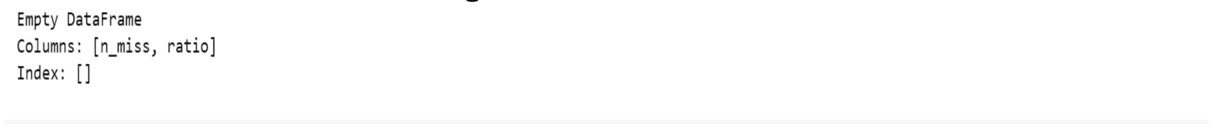
# Define a function to analyze missing values in a DataFrame and print a summary including counts and percentages



```
missing_values_table(diabetes_dataset)
```

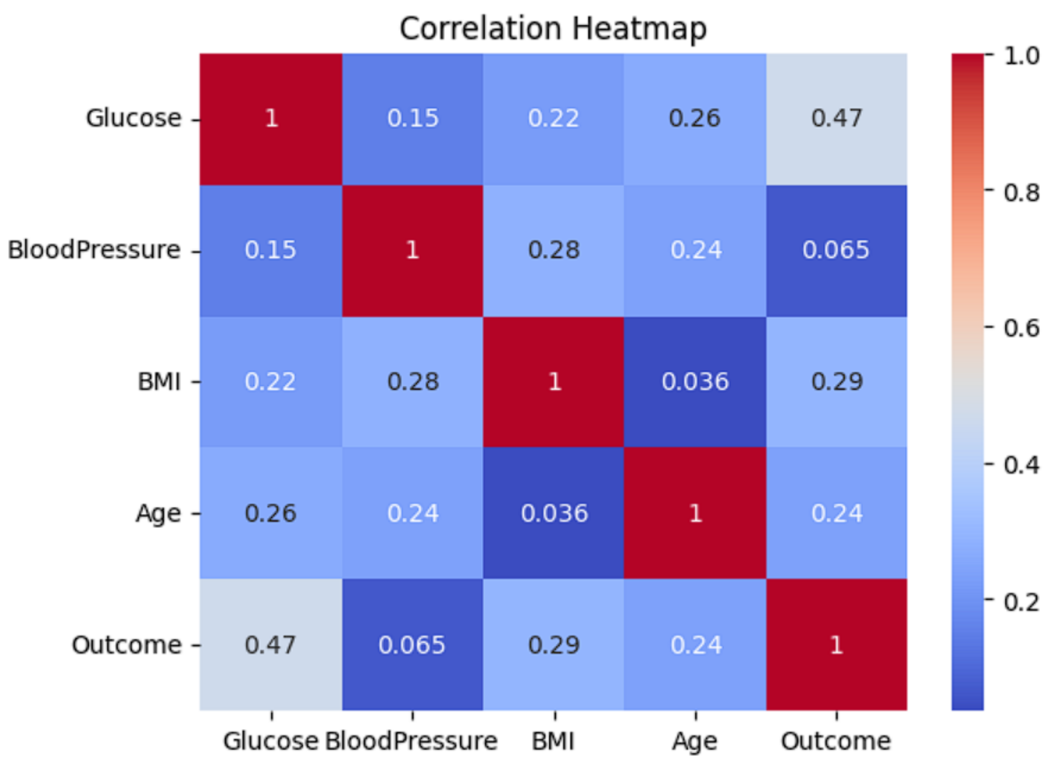
```
Empty DataFrame
Columns: [n_miss, ratio]
Index: []
```

# 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: []
```

# 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
```

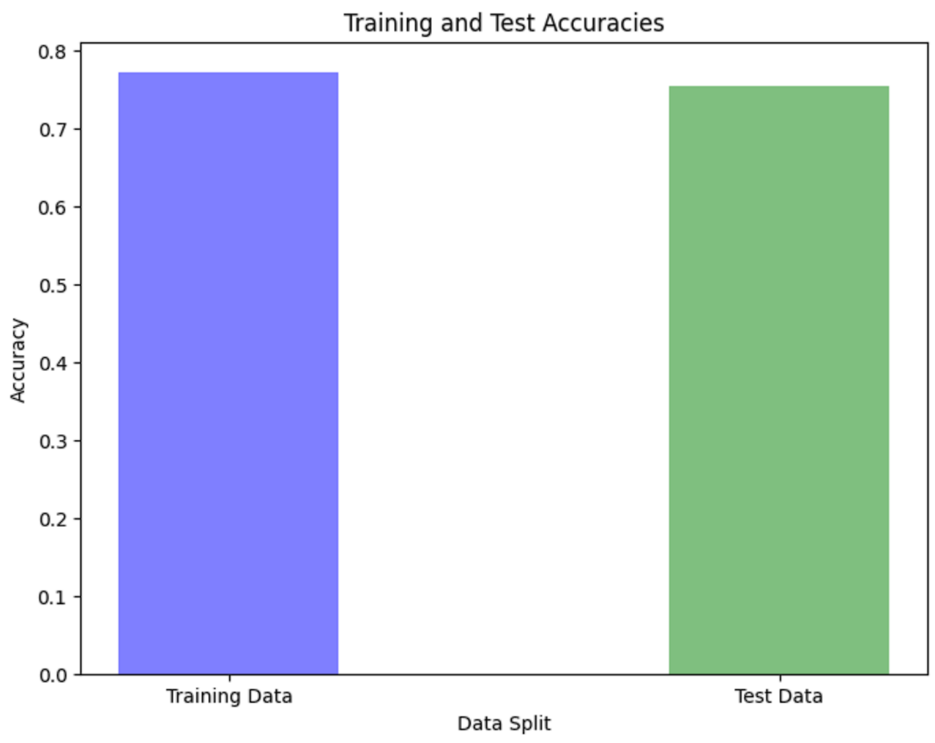
# Initialize another SVM classifier

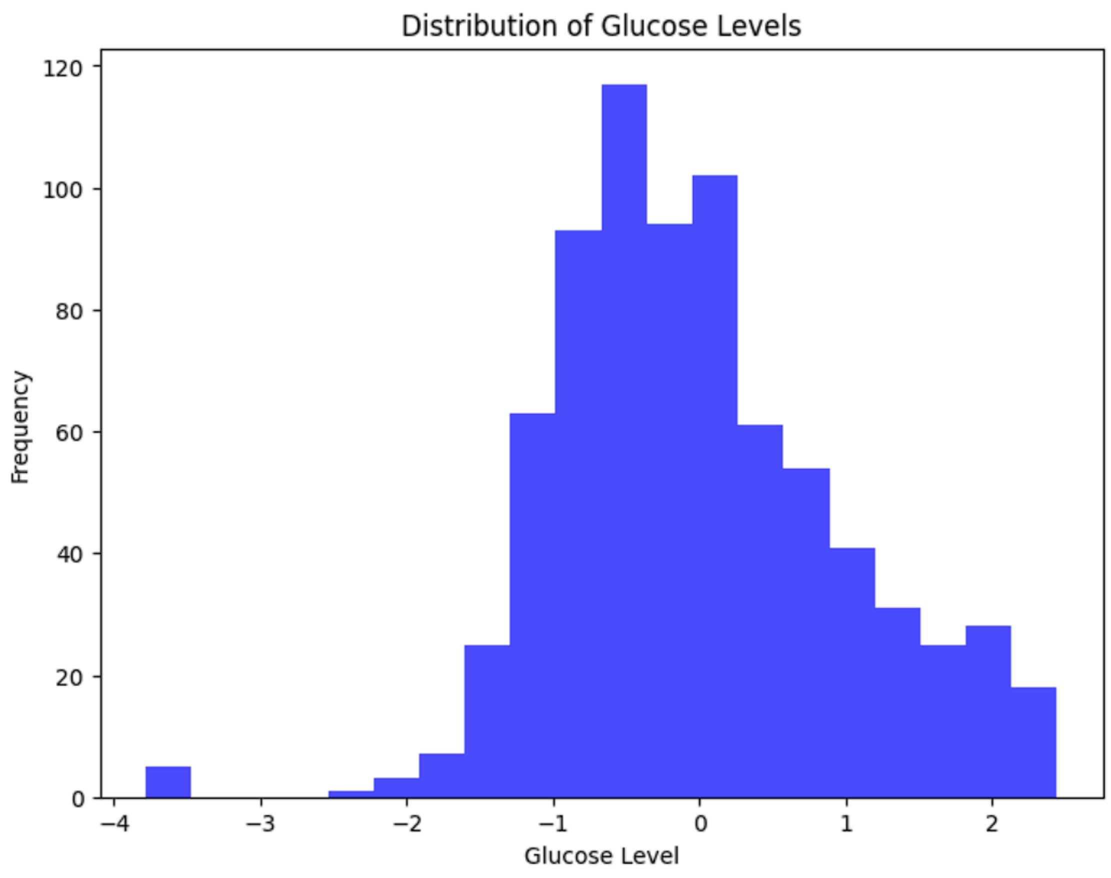
```
Accuracy: 0.7467532467532467
```

# 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

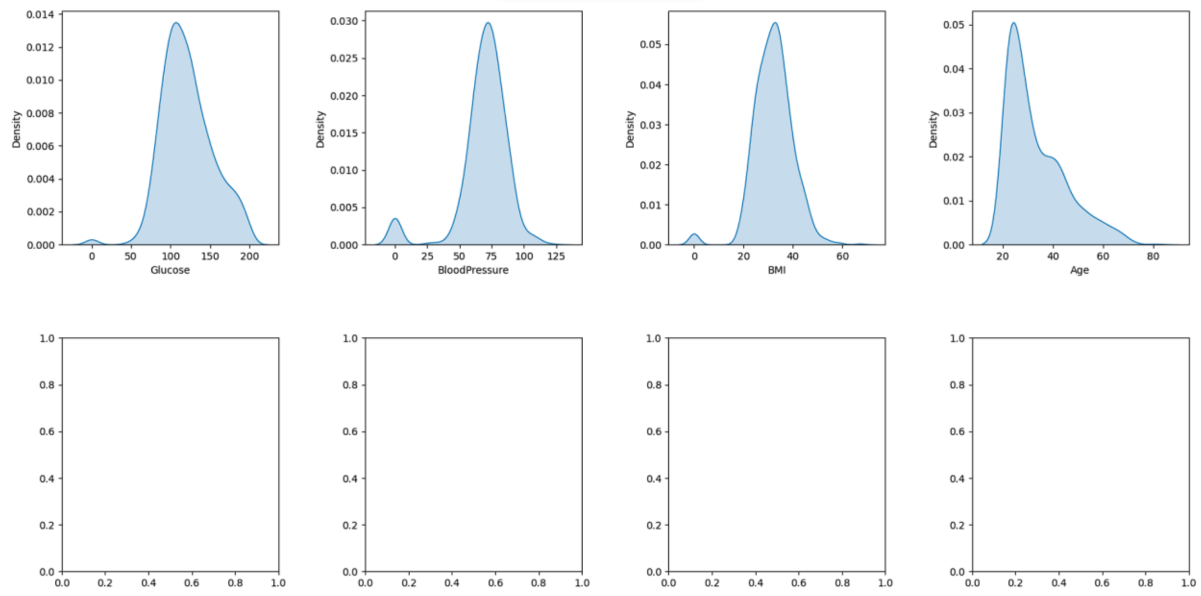




# Create a Random Forest Classifier

```
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

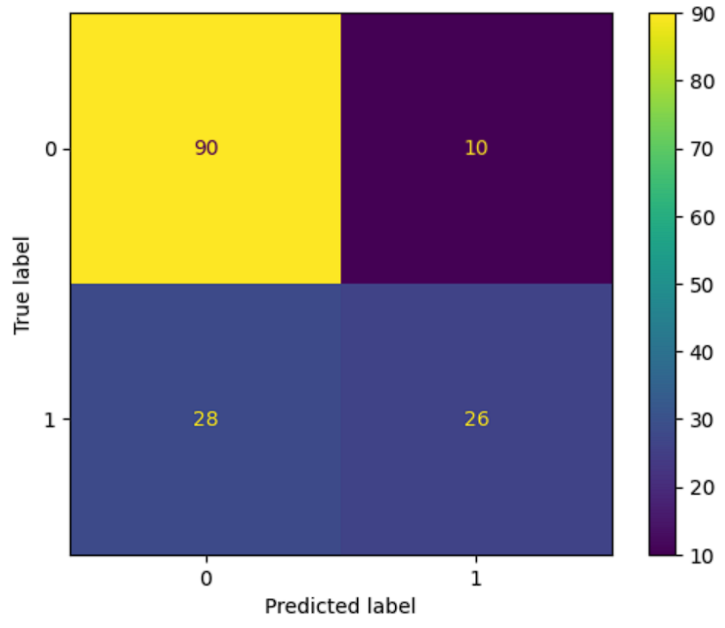
```
Accuracy score of the test data :  0.7532467532467533
[[0.84832379 0.14964075 0.20401277 1.4259954 ]]
[1]
The person is diabetic
```

# Print the corresponding labels for the first 5 samples

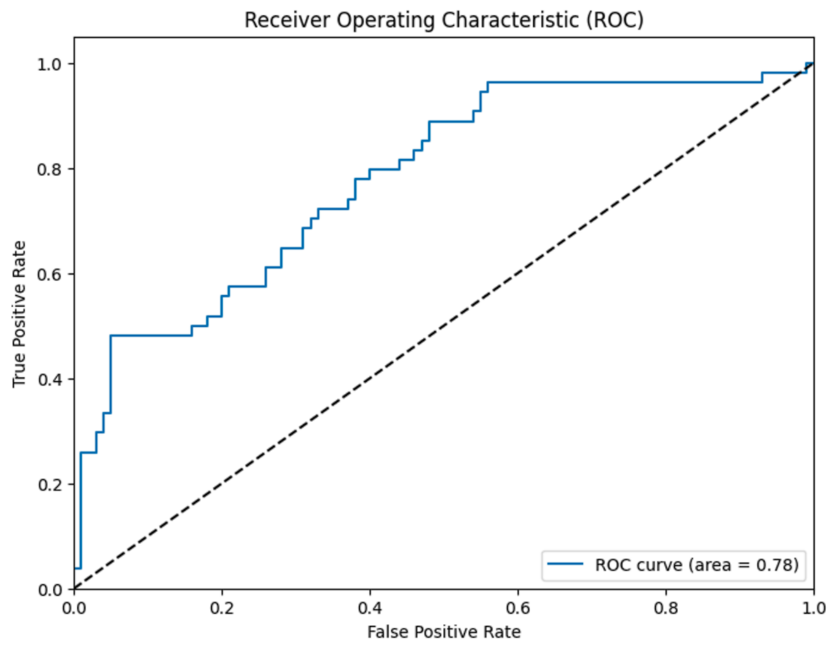
```
Synthetic Data (X_synthetic):
[[ 0.84832379  0.14964075  0.20401277  1.4259954 ]
 [-1.12339636 -0.16054575 -0.68442195 -0.19067191]
 [ 1.94372388 -0.26394125 -1.10325546 -0.10558415]
 [-0.99820778 -0.16054575 -0.49404308 -1.04154944]
 [ 0.5040552  -1.50468724  1.4097456  -0.0204964 ]]
Synthetic Labels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

# Print the performance metrics confusion matrix

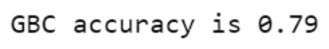
```
Accuracy: 0.75
Precision: 0.72
Recall: 0.48
F1 Score: 0.58
ROC-AUC Score: 0.78
Classification Report is:               precision    recall  f1-score   support

           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

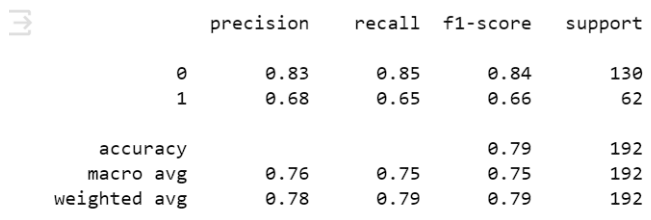


# Print the accuracy of the Gradient Boosting Classifier on the test set



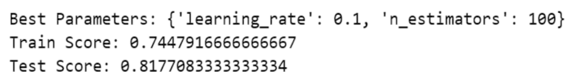
```
GBC accuracy is 0.79
```

# 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
```

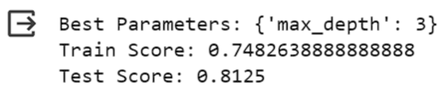
# Create a Gradient Boosting Classifier



```
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
```

Conclusion:

Gradient booster has the best accuracy score.

A gradient boosting classifier is used when the target column is binary. All the steps explained in the Gradient boosting regressor are used here, the only difference is we change the loss function. Earlier we used Mean squared error when the target column was continuous but this time, we will use log-likelihood as our loss function.