AMERICAN INTERNATIONAL UNIVERSITY-
BANGLADESH

PROGRAMMING IN PYTHON

FINAL TERM REPORT

SPRING 22-23

Project Title: Classification based machine learning model development
Project Members: Md. Oli Ullah Rafi & Md. Apon Riaz Talukder
Project Members Id: 20-42934-1 & 20-42905-1
Course And Section: Introduction To Python [A]
Honorable Faculty: Abdus Salam

Project Overview:

A classification-based machine learning model is to be developed in Python as part of the project. The utilization of a genuine dataset, comprising a minimum of 200 instances, is mandated. The project encompasses tasks such as data preprocessing, Data-Cleaning, exploratory data analysis, and the construction of classification-based models using the naive Bayes algorithm. The accuracy of the model, along with metrics such as Recall, F-1 score, Precision, and the generation of a confusion matrix, is to be determined from the dataset. The evaluation and comparison of classifiers will be predicated on predictive accuracy. The final project report will adhere to a standardized template, encompassing components such as a cover page, introduction, dataset overview, data pre-processing and exploratory data analysis, model development, and a conclusive section.

Dataset Overview:

i. **Data Source with Valid URL for Diabetes Prediction:**
Diabetes Prediction Dataset with Valid URL: (https://www.kaggle.com/datasets/iammustafatz/diabetes-prediction-dataset)

ii. **Description of the Dataset**:
The goal of this project is to utilize Python for the creation of a Classification-based Machine Learning Model. The dataset contains 100,000 instances and 9 attributes. This project will leverage a large dataset of 100,000 individuals to explore the relationship between heart disease, BMI, HbA1c levels, blood glucose levels, and diabetes. The analysis aims to predict diabetes in patients based on their medical history and demographic information, providing valuable insights for healthcare professionals in patient identification.

Hypertension:
Including hypertension status is crucial for identifying high-risk individuals and developing personalized prevention and management strategies. Hypertension is marked as 0 and 1, indicating its absence or presence in this database.

Heart Disease:
Heart disease is a common complication in individuals with diabetes. It is categorized as 0 and 1, denoting its absence or presence.

Smoking History:
Smoking is a risk factor for the development of type 2 diabetes and related complications. Including smoking history can enhance the accuracy of diabetes risk prediction and enable personalized prevention and management strategies. The database categorizes diabetes based on smoking status, including 'Never,' 'No info,' 'Current,' 'Former,' 'Ever,' and 'Not Current.

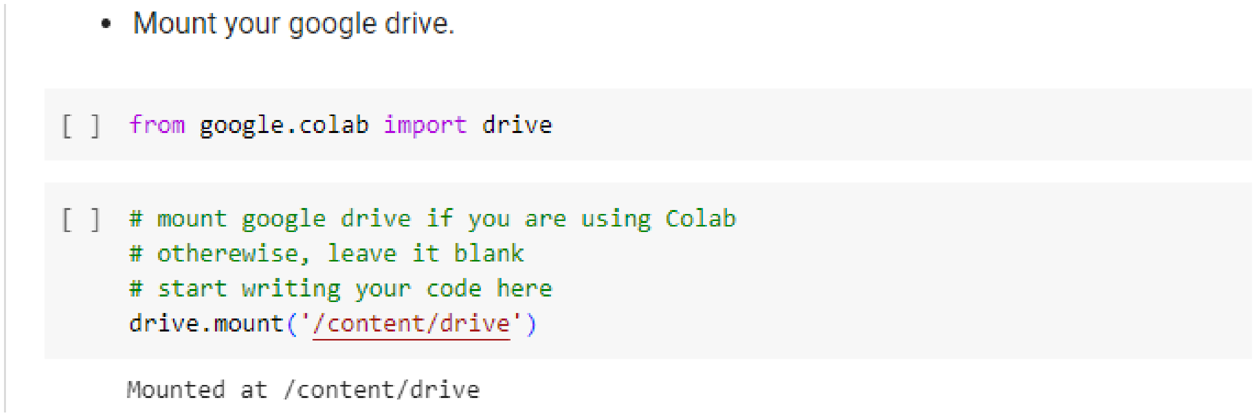
BMI:
Higher BMI levels are associated with a higher risk of insulin resistance and other diabetes-related factors. In this dataset, BMI ranges from 0.52 to 23.45.

HbA1c Level:
Including HbA1c levels can improve the accuracy of diabetes risk prediction and facilitate personalized prevention and management strategies. In this database, HbA1c levels vary from 3.5 to 9.0.

Blood Glucose Level:
Including blood glucose levels can help identify individuals at higher risk of developing diabetes and enable the development of targeted prevention and management strategies based on their risk profile. In this dataset, blood glucose levels vary from 80 to 300.

Mount google drive using Colab:

Mount the dataset from Google Drive in Google Colab, connect to the dataset, and then import the necessary libraries.



- Mount your google drive.

```
[ ]  from google.colab import drive
```

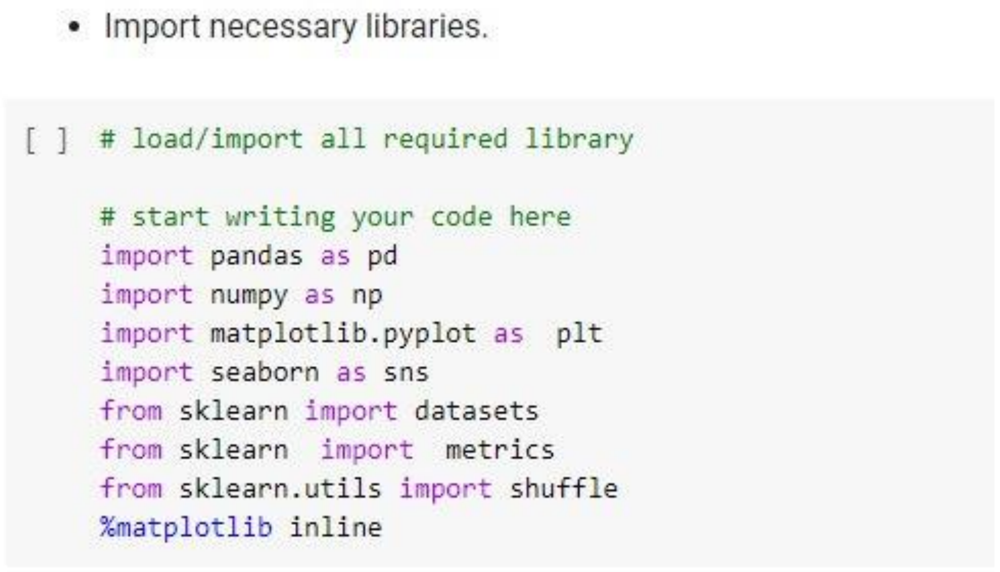
```
[ ]  # mount google drive if you are using Colab
     # otherewise, leave it blank
     # start writing your code here
     drive.mount('/content/drive')

     Mounted at /content/drive
```

Figure 01: Mounting Google Drive

Mounting your Google Drive in Google Colab integrates your cloud storage as a locally accessible directory, eliminating the need for file uploads and downloads. This simplifies your workflow, allowing immediate interaction with datasets. After establishing the connection, navigating to the dataset's location in your mounted Drive lets you easily import essential libraries like pandas or Numpy. This streamlined process enhances efficiency, maximizes productivity, and minimizes technological roadblocks during data analysis tasks.

Import all the necessary libraries:

Importing essential libraries such as Pandas for data manipulation, Matplotlib for visualization, and NumPy for numerical computations, establishes the foundational tools required to efficiently process and analyze. data within the Python environment.



- Import necessary libraries.

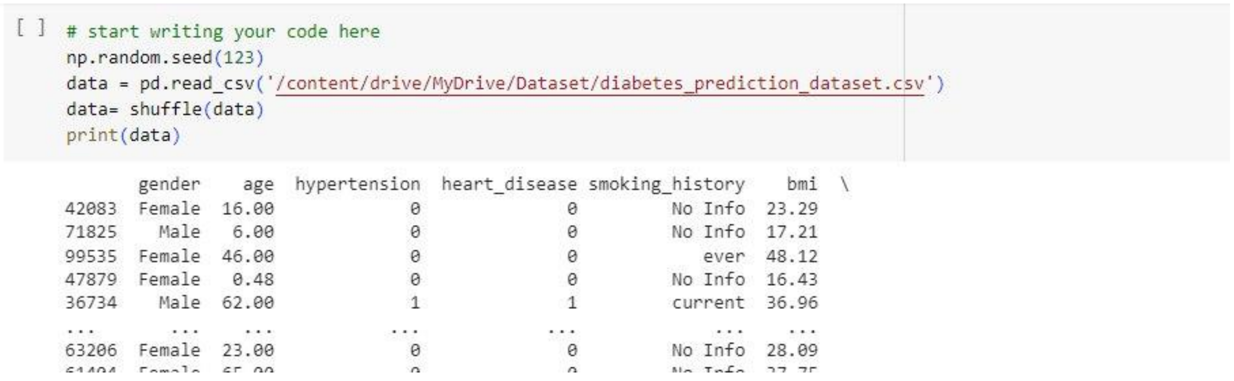
```
[ ]  # load/import all required library

     # start writing your code here
     import pandas as pd
     import numpy as np
     import matplotlib.pyplot as  plt
     import seaborn as sns
     from sklearn import datasets
     from sklearn  import  metrics
     from sklearn.utils import shuffle
     %matplotlib inline
```

Figure 02: Import all Necessary Libraries

The establishment of a robust foundation for efficient data processing and analysis in the Python environment involves importing essential libraries, each playing a vital role. Pandas empowers seamless exploration and transformation of datasets through high-performance, user-friendly data structures, and analysis tools. Matplotlib grants the ability to generate visually compelling data visualizations, facilitating comprehension and communication of insights. NumPy optimizes numerical computations with a fundamental array data structure and powerful array operations, enabling sophisticated data analysis techniques. The integration of these libraries forms a comprehensive toolkit, equipping analysts to navigate, understand, and extract meaningful insights from complex datasets.

Task-01: Read/Load the dataset file using Pandas library to complete this task.

Using a random seed of 123 so that after running the dataset, the attributes remain consistently the same each time the dataset is rerun, maintaining uniform data.



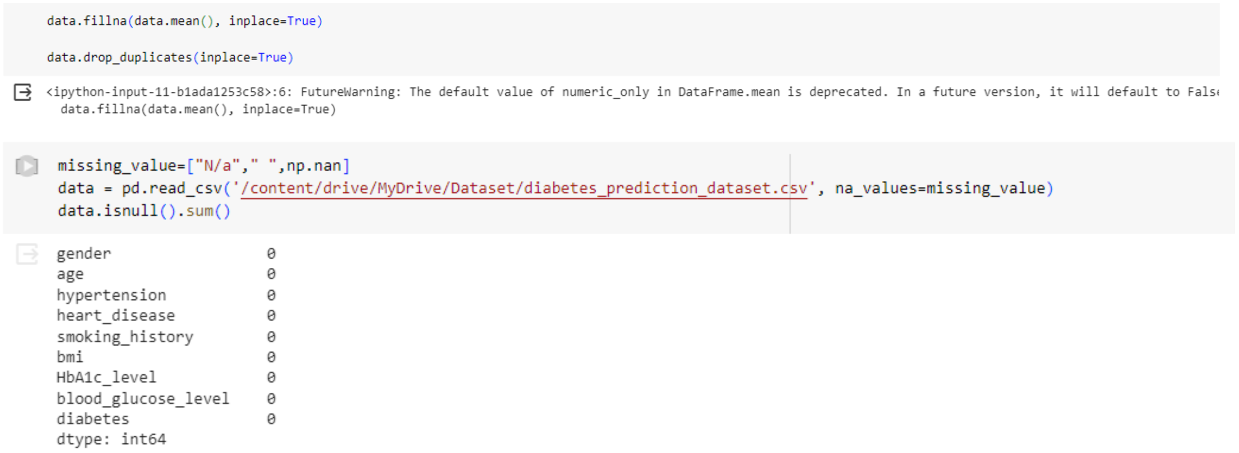
```
[ ] # start writing your code here
    np.random.seed(123)
    data = pd.read_csv('/content/drive/MyDrive/Dataset/diabetes_prediction_dataset.csv')
    data= shuffle(data)
    print(data)
```

```
       gender    age  hypertension  heart_disease smoking_history    bmi  \
42083  Female  16.00             0              0         No Info  23.29
71825    Male   6.00             0              0         No Info  17.21
99535  Female  46.00             0              0            ever  48.12
47879  Female   0.48             0              0         No Info  16.43
36734    Male  62.00             1              1         current  36.96
...       ...    ...           ...            ...             ...    ...
63206  Female  23.00             0              0         No Info  28.09
```

Figure 03: Read/Load the Dataset Using Pandas

The dataset is first loaded into the Python environment using the Pandas library, and a random seed of 123 is set to maintain attribute consistency across multiple runs. This ensures reproducibility and uniform data for accurate analysis.

Task-02. Applying appropriate data cleaning techniques to the dataset and replacing erroneous data using proper methods. Do not delete any record except for duplicate entries. Using the Pandas library to complete this task.



```
data.fillna(data.mean(), inplace=True)

data.drop_duplicates(inplace=True)
```

```
<ipython-input-11-b1ada1253c58>:6: FutureWarning: The default value of numeric_only in DataFrame.mean is deprecated. In a future version, it will default to Fals
  data.fillna(data.mean(), inplace=True)
```

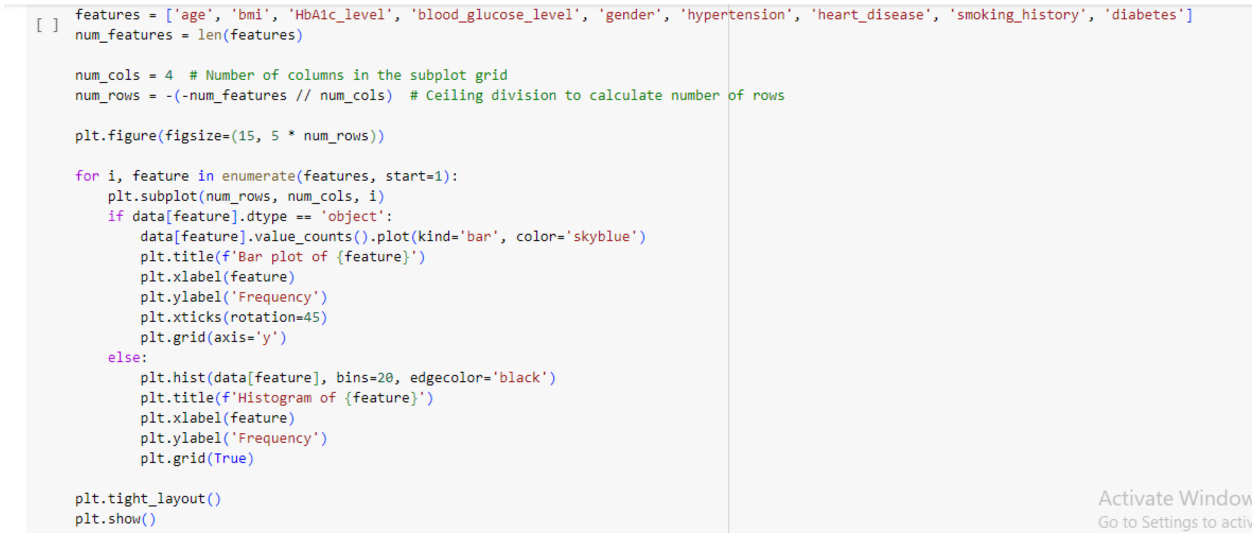
```
missing_value=["N/a"," ",np.nan]
data = pd.read_csv('/content/drive/MyDrive/Dataset/diabetes_prediction_dataset.csv', na_values=missing_value)
data.isnull().sum()
```

```
gender                 0
age                    0
hypertension           0
heart_disease          0
smoking_history        0
bmi                    0
HbA1c_level            0
blood_glucose_level    0
diabetes               0
dtype: int64
```

Figure 04: Applying Appropriate Data Cleaning Techniques.

The crucial stage of data cleaning. Utilizing the versatile Pandas library, this step meticulously tackles inaccuracies within the dataset. Erroneous data is identified and addressed using appropriate methods, while duplicate entries are eliminated to uphold data integrity. Crucially, no single record beyond duplicates is discarded, ensuring comprehensiveness remains paramount throughout the process. This phase paves the way for reliable and trustworthy analysis in later stages.

Task-03. Draw graphs to analyze the frequency distributions of the features. The Matplotlib library is being used to complete this task.



```
features = ['age', 'bmi', 'HbA1c_level', 'blood_glucose_level', 'gender', 'hypertension', 'heart_disease', 'smoking_history', 'diabetes']
num_features = len(features)

num_cols = 4  # Number of columns in the subplot grid
num_rows = -(-num_features // num_cols)  # Ceiling division to calculate number of rows

plt.figure(figsize=(15, 5 * num_rows))

for i, feature in enumerate(features, start=1):
    plt.subplot(num_rows, num_cols, i)
    if data[feature].dtype == 'object':
        data[feature].value_counts().plot(kind='bar', color='skyblue')
        plt.title(f'Bar plot of {feature}')
        plt.xlabel(feature)
        plt.ylabel('Frequency')
        plt.xticks(rotation=45)
        plt.grid(axis='y')
    else:
        plt.hist(data[feature], bins=20, edgecolor='black')
        plt.title(f'Histogram of {feature}')
        plt.xlabel(feature)
        plt.ylabel('Frequency')
        plt.grid(True)

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

Figure 05: Analyze the Frequency Distributions (i)

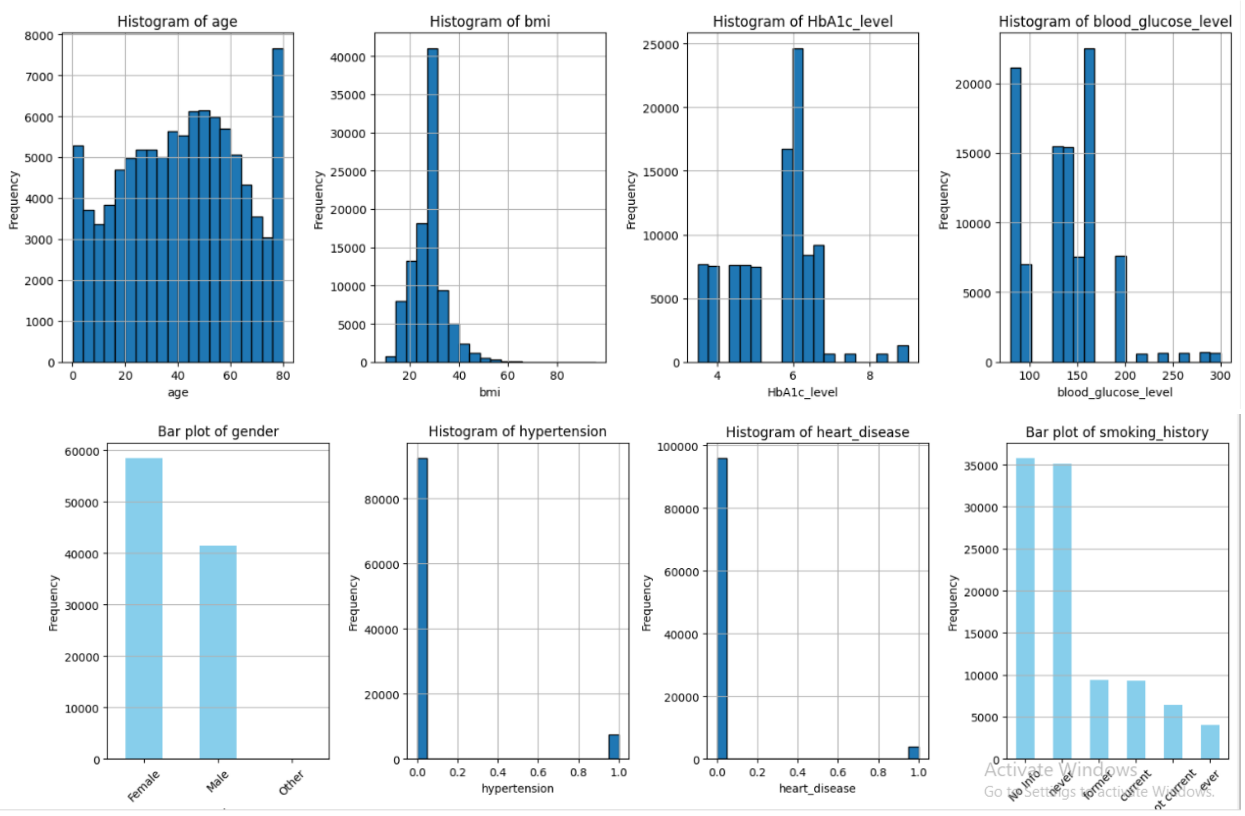


Figure 06: Analyze the Frequency Distributions (ii)

Matplotlib is used to visualize the frequency distributions of individual features, likely using histograms. These graphs reveal distribution patterns, central tendencies, spread, and potential outliers. To provide more specific insights, please describe the key visual elements you observe in the graphs, such as the features depicted, distribution shapes, notable trends, variability comparisons, and outlier identification.

Task-04: Draw the graphs to illustrate whether there is any relationship between the target column and any other columns of the dataset. Using the Matplotlib library to complete this task.

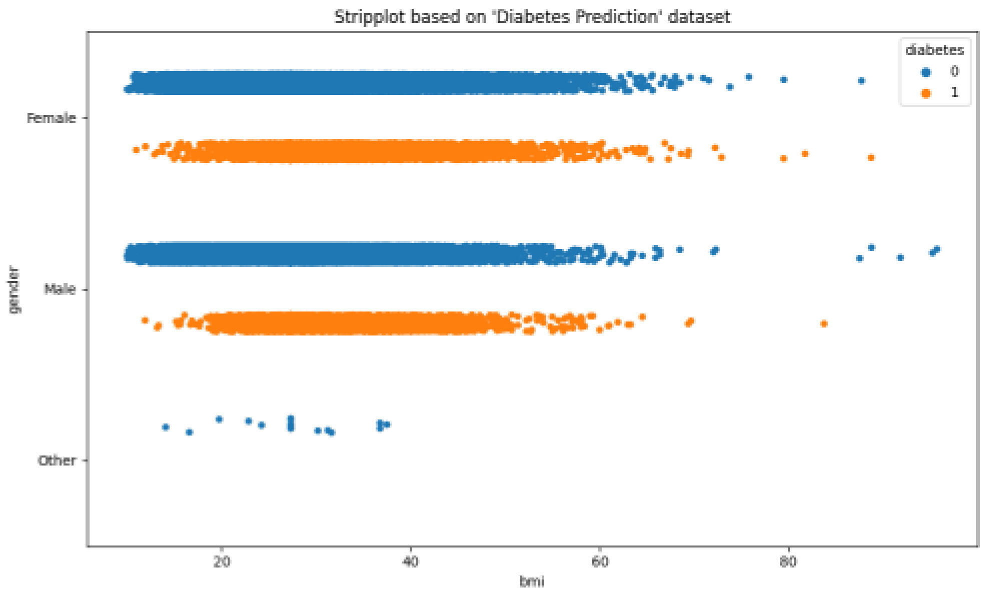


Figure 07: Visualization of BMI vs Gender on Stripplot

Stripplot is a type of plot used to display the distribution of a numerical variable across different categories of a categorical variable. It does so by plotting individual data points as dots along a horizontal or vertical axis corresponding to the categories. Here x axis and y axis represent bmi and gender. Blue and Yellow color indicates negative and positive diabetes result.

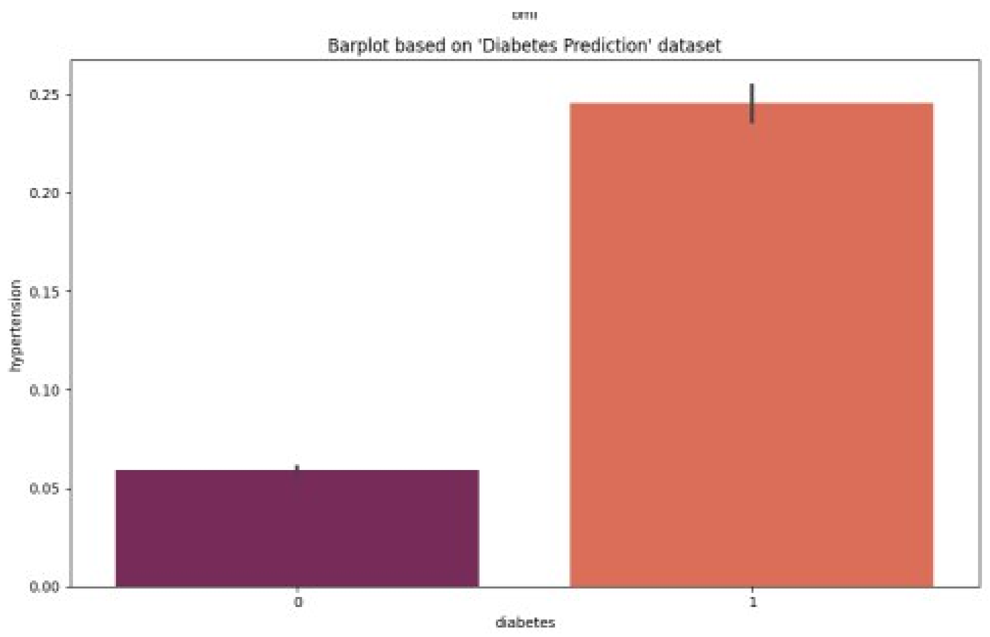


Figure 08: Visualization of Diabetes on Bar plot

Bar plot is a type of chart used to visualize categorical data by representing each category as a bar of a certain height or length. It is a way to aggregate and compare data across different categories. By default, the height of each bar represents the mean value of the data within that category. Here x-axis and y-axis represent diabetes and hypertension. Rocket fuel red and Peach color indicates negative and positive diabetes results. As hypertension increases the tendency to get diabetes also increases.

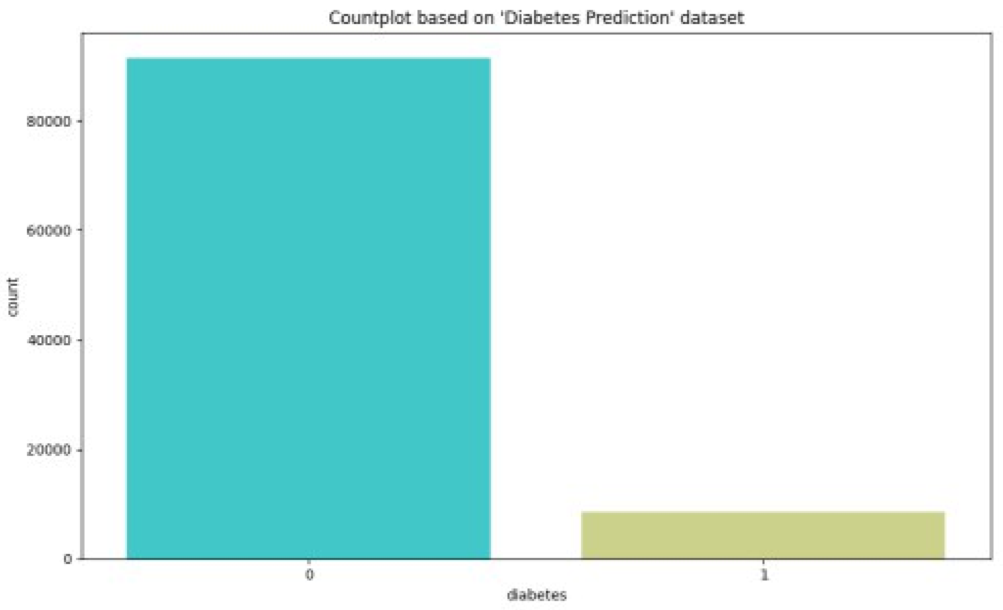
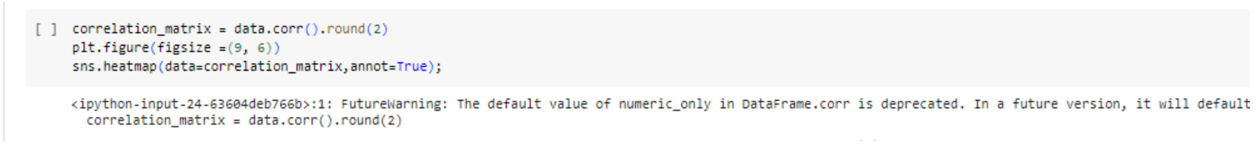


Figure 09: Visualization of Diabetes on Countplot

Countplot is a chart that shows the number of occurrences of each category in a categorical variable. This plot is provided by the seaborn library and is a variation of a bar plot, where the height of each bar represents the frequency of each category.

Correlation:



```
[ ]  correlation_matrix = data.corr().round(2)
     plt.figure(figsize =(9, 6))
     sns.heatmap(data=correlation_matrix,annot=True);

     <ipython-input-24-63604deb766b>:1: FutureWarning: The default value of numeric_only in DataFrame.corr is deprecated. In a future version, it will default
       correlation_matrix = data.corr().round(2)
```

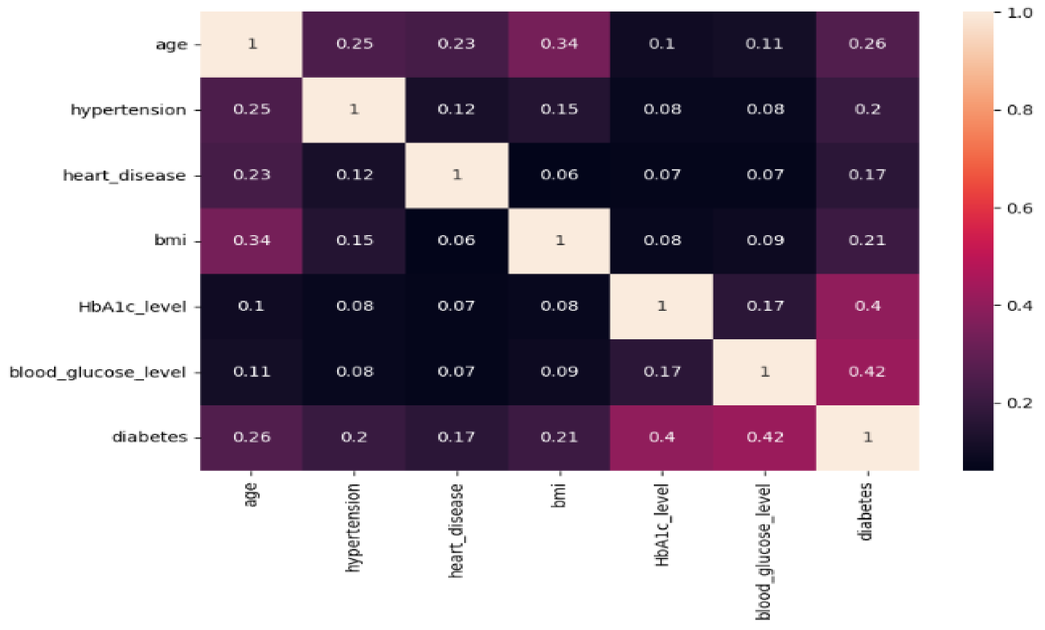
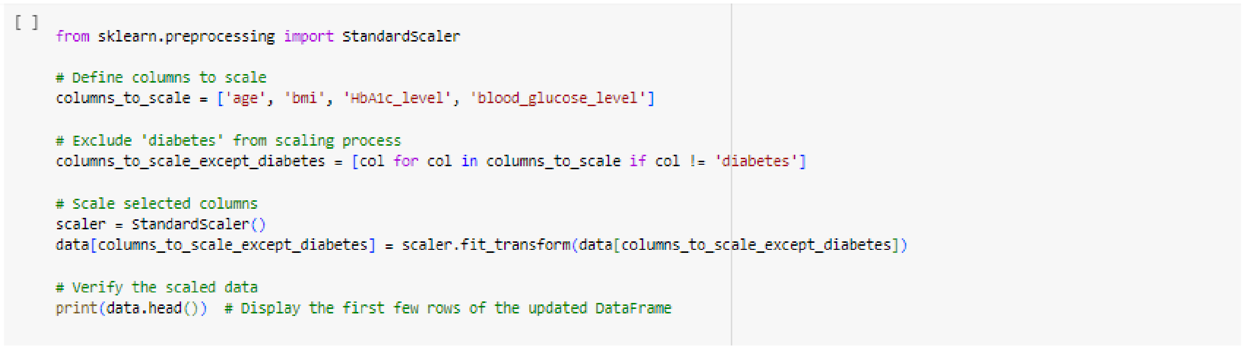


Figure 10: Visualization of Correlation Matrix

This code will output the correlation coefficients between the columns 'age,' 'bmi,' 'HbA1c_level,' 'blood_glucose_level,' 'gender,' 'hypertension,' 'heart_disease,' 'smoking_history,' and 'diabetes' in the DataFrame 'data.'

Task-05: Perform scaling on the features of the dataset, applying data conversion if needed before the scaling process.

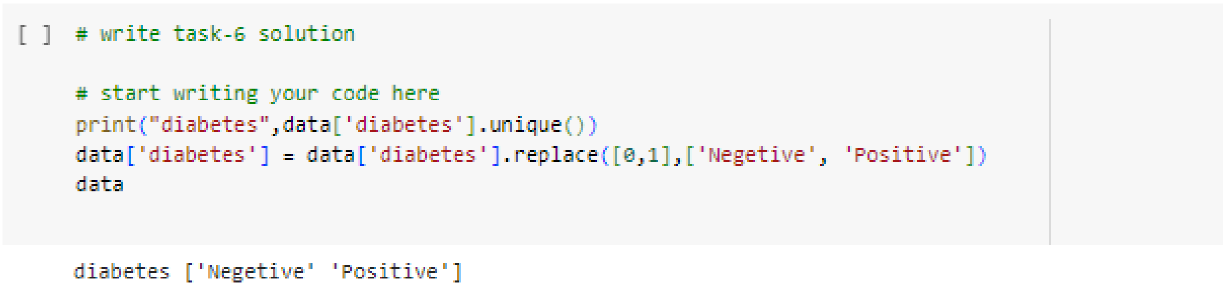


```
[ ]
from sklearn.preprocessing import StandardScaler

# Define columns to scale
columns_to_scale = ['age', 'bmi', 'HbA1c_level', 'blood_glucose_level']

# Exclude 'diabetes' from scaling process
columns_to_scale_except_diabetes = [col for col in columns_to_scale if col != 'diabetes']

# Scale selected columns
scaler = StandardScaler()
data[columns_to_scale_except_diabetes] = scaler.fit_transform(data[columns_to_scale_except_diabetes])

# Verify the scaled data
print(data.head())  # Display the first few rows of the updated DataFrame
```

Figure 11: Scaling on the Features and Applied Data Conversion

Perform scaling on the dataset features, applying data conversion as needed before the scaling process. This step ensures uniformity and optimal performance in numerical computations, enhancing the accuracy of subsequent analyses. Scaling is crucial for maintaining consistent data ranges, preventing certain features from dominating others. By standardizing the variables, the dataset becomes more conducive to modeling and analysis, providing a foundation for robust machine learning algorithms.

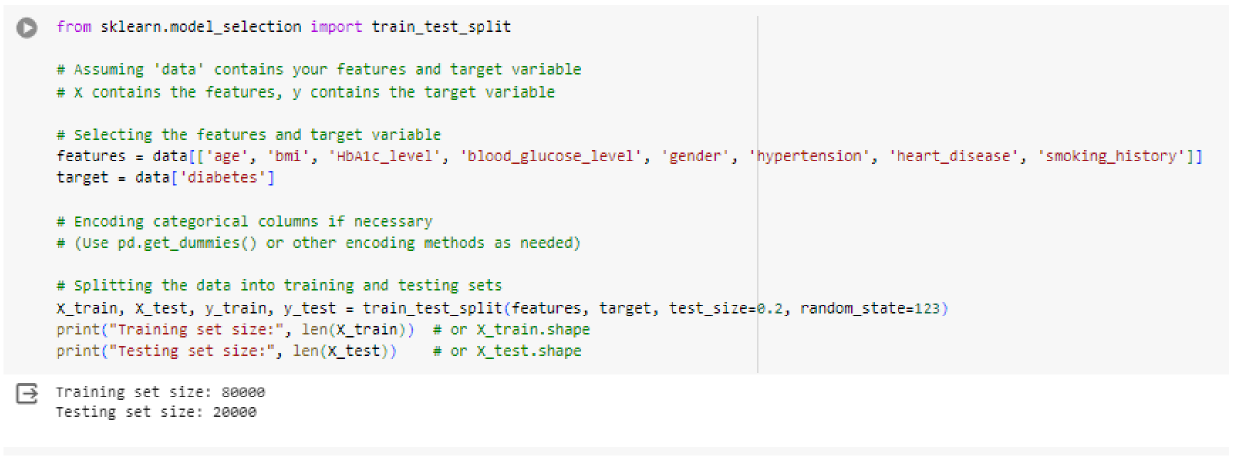
Task-07: Split the data into two parts: a training dataset and a testing dataset using the `train_test_split ()` function. Also, use the value 123 as the value for the `random_state` parameter of this function.



```
[ ] # write task-6 solution

    # start writing your code here
    print("diabetes",data['diabetes'].unique())
    data['diabetes'] = data['diabetes'].replace([0,1],['Negetive', 'Positive'])
    data

    diabetes ['Negetive' 'Positive']
```

Figure 12: Replaced 0,1 into Positive and Negative in the Dataset

Partition the data into two sets—training and testing—employing the `train_test_split()` function. Specify a `random_state` value of 123 to ensure consistent results in subsequent runs. This division facilitates model training on one subset while evaluating its performance on another, ensuring a reliable assessment of predictive capabilities. Utilizing a fixed random seed enhances reproducibility, crucial for consistent experimentation and model comparison. This systematic approach supports the development and evaluation of machine learning models, optimizing their effectiveness in real-world applications.

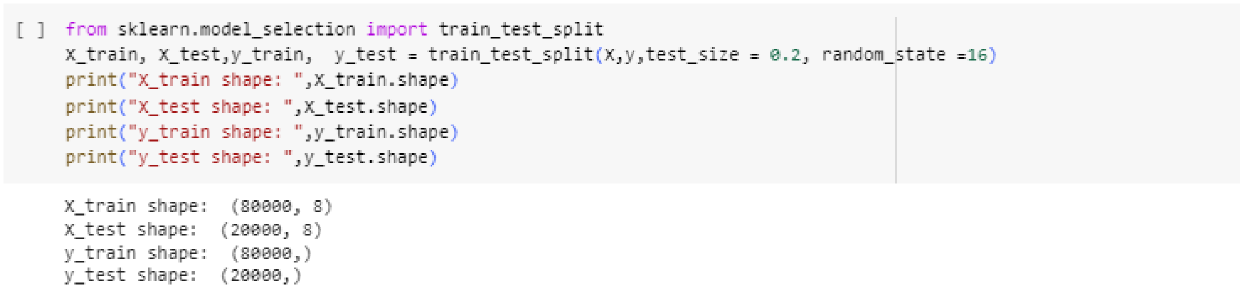


```
from sklearn.model_selection import train_test_split

# Assuming 'data' contains your features and target variable
# X contains the features, y contains the target variable

# Selecting the features and target variable
features = data[['age', 'bmi', 'HbA1c_level', 'blood_glucose_level', 'gender', 'hypertension', 'heart_disease', 'smoking_history']]
target = data['diabetes']

# Encoding categorical columns if necessary
# (Use pd.get_dummies() or other encoding methods as needed)

# Splitting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(features, target, test_size=0.2, random_state=123)
print("Training set size:", len(X_train))  # or X_train.shape
print("Testing set size:", len(X_test))    # or X_test.shape
```

```
Training set size: 80000
Testing set size: 20000
```

Figure 13: Split Data into Testing and Training

Separate the dataset into distinct training and testing subsets. This division is crucial for evaluating the performance of machine learning models. The training set is used to train and build the model, while the testing set assesses its predictive accuracy on unseen data. This strategic partitioning ensures a robust assessment of the model's generalization capabilities, enhancing its reliability in real-world scenarios.



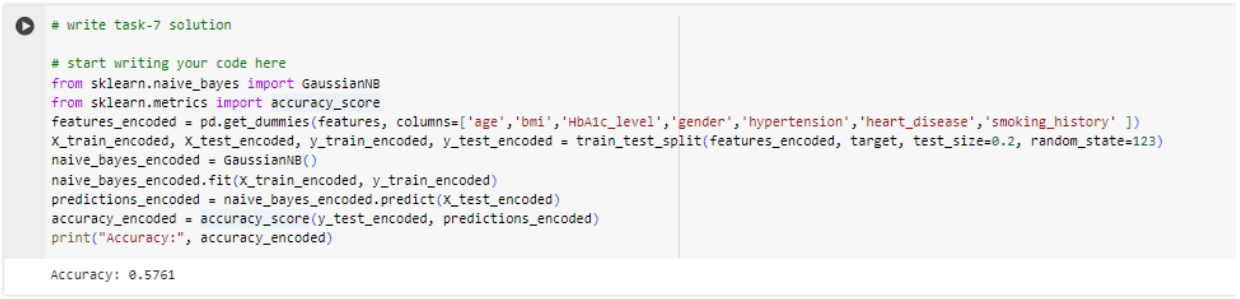
```
[ ]  from sklearn.model_selection import train_test_split
     X_train, X_test,y_train,  y_test = train_test_split(X,y,test_size = 0.2, random_state =16)
     print("X_train shape: ",X_train.shape)
     print("X_test shape: ",X_test.shape)
     print("y_train shape: ",y_train.shape)
     print("y_test shape: ",y_test.shape)
```

```
X_train shape:  (80000, 8)
X_test shape:  (20000, 8)
y_train shape:  (80000,)
y_test shape:  (20000,)
```

Figure 14: Train the Split Data

Employ the `train_test_split()` function to train the divided dataset. This method facilitates the creation of training and testing subsets, a fundamental step in machine learning model development. Setting the `random_state` parameter to 123 ensures reproducibility, allowing consistent results across multiple executions. Training involves feeding the model with the training subset, enabling it to learn patterns and relationships within the data. This process is pivotal for honing the model's predictive capabilities. The testing subset, held separately, serves to assess the model's performance on unseen data, gauging its effectiveness in real-world applications.

Task-07: Applying the Naïve Bayes Classifier to the dataset. Building your prediction model in this step.



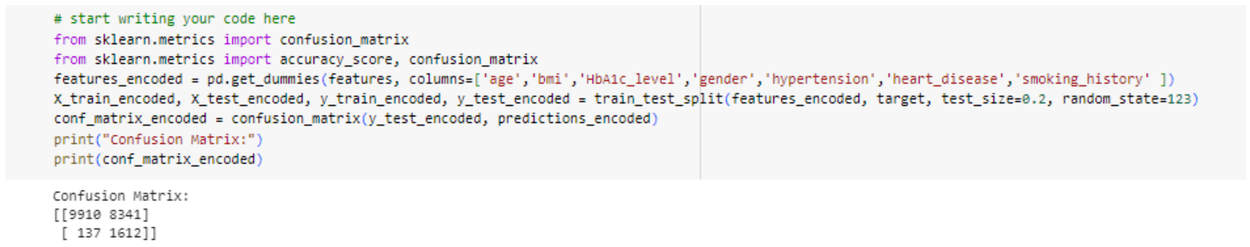
```
# write task-7 solution

# start writing your code here
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score
features_encoded = pd.get_dummies(features, columns=['age','bmi','HbA1c_level','gender','hypertension','heart_disease','smoking_history' ])
X_train_encoded, X_test_encoded, y_train_encoded, y_test_encoded = train_test_split(features_encoded, target, test_size=0.2, random_state=123)
naive_bayes_encoded = GaussianNB()
naive_bayes_encoded.fit(X_train_encoded, y_train_encoded)
predictions_encoded = naive_bayes_encoded.predict(X_test_encoded)
accuracy_encoded = accuracy_score(y_test_encoded, predictions_encoded)
print("Accuracy:", accuracy_encoded)
```

```
Accuracy: 0.5761
```

Figure 15: Naïve Bayes Classifier Prediction Model

Apply the Naïve Bayes Classifier to construct a prediction model from the dataset. This step involves leveraging the classifier's probabilistic approach, which assumes independence between features. By building the model, it learns patterns within the data to make predictions. The Naïve Bayes method is particularly effective for classification tasks, making it suitable for predicting outcomes based on input features. This process empowers the model to discern and categorize patterns in the dataset, enhancing its ability to make accurate predictions when presented with new, unseen data.

Task-08: Calculating the confusion matrix for your model and interpreting it in detail in the report.



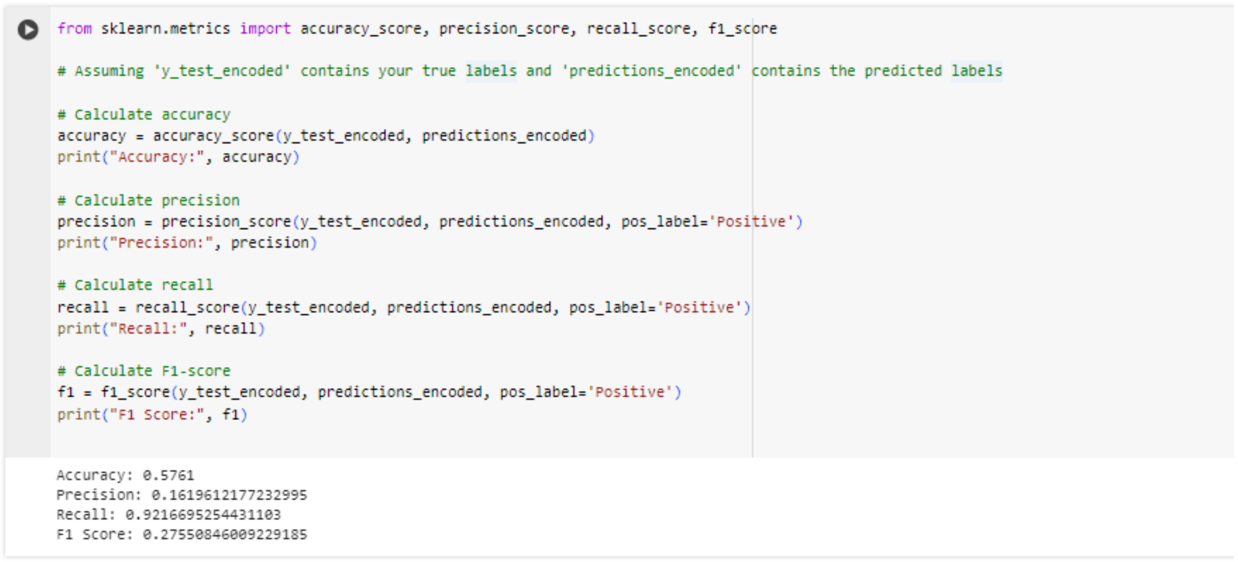
```
# start writing your code here
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score, confusion_matrix
features_encoded = pd.get_dummies(features, columns=['age','bmi','HbA1c_level','gender','hypertension','heart_disease','smoking_history' ])
X_train_encoded, X_test_encoded, y_train_encoded, y_test_encoded = train_test_split(features_encoded, target, test_size=0.2, random_state=123)
conf_matrix_encoded = confusion_matrix(y_test_encoded, predictions_encoded)
print("Confusion Matrix:")
print(conf_matrix_encoded)
```

```
Confusion Matrix:
[[9910 8341]
 [ 137 1612]]
```

Figure 16: Calculate the Confusion Matrix

Compute the confusion matrix for your model and provide a comprehensive interpretation in the report. This matrix is a crucial evaluation tool, offering insights into the performance of a classification model. It breaks down the true positive, true negative, false positive, and false negative predictions, facilitating a detailed analysis of model accuracy. A thorough interpretation in the report ensures a nuanced understanding of how well the model is distinguishing between classes, enabling stakeholders to make informed decisions based on the model's strengths and areas for improvement.

Task-09: Calculating the accuracy, precision, recall and f-1 score of this model.



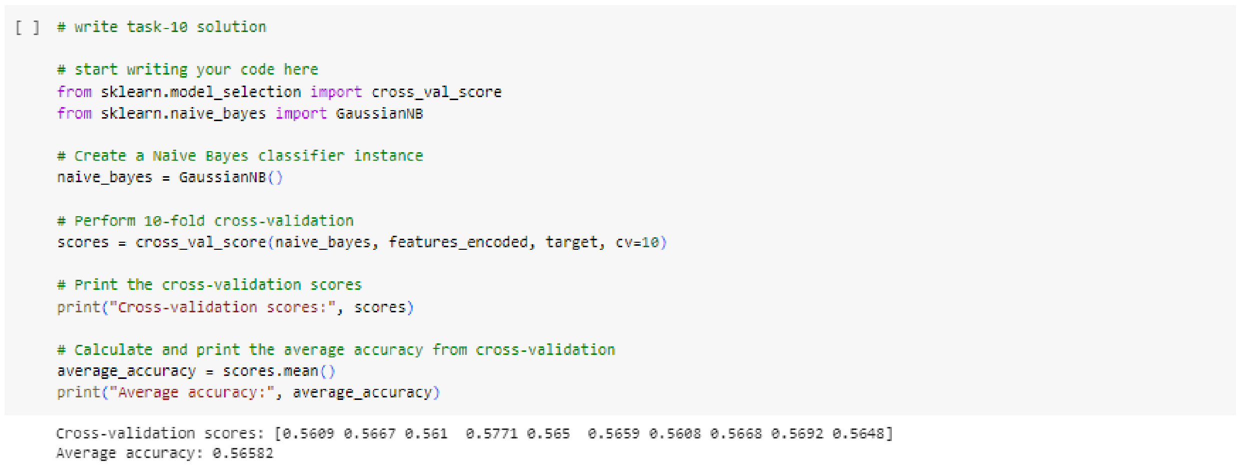
```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# Assuming 'y_test_encoded' contains your true labels and 'predictions_encoded' contains the predicted labels

# Calculate accuracy
accuracy = accuracy_score(y_test_encoded, predictions_encoded)
print("Accuracy:", accuracy)

# Calculate precision
precision = precision_score(y_test_encoded, predictions_encoded, pos_label='Positive')
print("Precision:", precision)

# Calculate recall
recall = recall_score(y_test_encoded, predictions_encoded, pos_label='Positive')
print("Recall:", recall)

# Calculate F1-score
f1 = f1_score(y_test_encoded, predictions_encoded, pos_label='Positive')
print("F1 Score:", f1)
```

```
Accuracy: 0.5761
Precision: 0.1619612177232995
Recall: 0.9216695254431103
F1 Score: 0.27550846009229185
```

Figure 17: Calculate the accuracy, precision, recall and f-1 score

Compute the accuracy, precision, recall, and F1 score metrics for the model. These measures collectively assess the performance of a classification model. Accuracy gauges the overall correctness, precision evaluates the true positive rate among predicted positives, recall measures the true positive rate among actual positives, and the F1 score balances precision and recall. Calculating and examining these metrics provides a comprehensive understanding of the model's effectiveness in correctly identifying and classifying instances, aiding in the assessment and refinement of its predictive capabilities.

Task-10: Showing how 10-fold cross validation can be used to build a naïve bayes classifier and report the accuracy of this model.



```
[ ]  # write task-10 solution

     # start writing your code here
     from sklearn.model_selection import cross_val_score
     from sklearn.naive_bayes import GaussianNB

     # Create a Naive Bayes classifier instance
     naive_bayes = GaussianNB()

     # Perform 10-fold cross-validation
     scores = cross_val_score(naive_bayes, features_encoded, target, cv=10)

     # Print the cross-validation scores
     print("Cross-validation scores:", scores)

     # Calculate and print the average accuracy from cross-validation
     average_accuracy = scores.mean()
     print("Average accuracy:", average_accuracy)

     Cross-validation scores: [0.5609 0.5667 0.561  0.5771 0.565  0.5659 0.5608 0.5668 0.5692 0.5648]
     Average accuracy: 0.56582
```

Figure 18: 10-fold cross validation using naïve bayes classifier.

Demonstrate the application of 10-fold cross-validation to construct a Naïve Bayes classifier and report the accuracy of the resulting model. This technique involves dividing the dataset into 10 folds, training the model on nine and validating on one iteratively. The process ensures a robust evaluation, as each instance serves as both training and validation data. By averaging the accuracies across the folds, a more reliable assessment of the classifier's performance is achieved. This approach enhances the model's generalization capabilities, providing a more accurate reflection of its effectiveness across diverse subsets of the data.

Conclusion: Following the application of Naive Bayes, the dataset achieved its highest accuracy at 0.5761. Subsequently, a training and test set was derived from the original dataset to create a machine learning model. Testing this model with a separate test dataset revealed the accuracy obtained from the algorithms applied to the dataset. Measuring accuracy to predict the outcome of a situation is a crucial concept in data science, aiding in enhancing generalization and minimizing overfitting.