GlucoSense: AI-Powered Diabetes Detection for Early Intervention

A Project Report

Submitted to

Infosys Springboard

Under the supervision of

Ravi

Submitted By

Varun Myaka

21R21A6640

Department of Computer Science and Engineering –

Artificial Intelligence and Machine Learning

MLR Institute of Technology (MLRIT)

Dundigal, Gandimaisamma, Hyderabad, 500043

Table of Contents

Abstract

Diabetes is one of the most prevalent chronic diseases worldwide, significantly impacting the quality of life and healthcare systems. Early detection of diabetes is critical for initiating timely interventions, improving treatment outcomes, and reducing the risk of severe complications such as cardiovascular disease and kidney failure. Traditional diagnostic methods often rely on resource-intensive and time-consuming laboratory tests, which are inaccessible to many in remote or underserved regions. This project, **GlucoSense: AI-Powered Diabetes Detection for Early Intervention** seeks to address this challenge by leveraging artificial intelligence (AI) to provide an efficient, accurate, and accessible solution for diabetes risk prediction.

The dataset used for this project comprises various health indicators such as glucose levels, BMI, age, and insulin levels, collected from a diverse population. Comprehensive exploratory data analysis (EDA) was conducted to uncover patterns, correlations, and trends within the data. Significant pre-processing steps were implemented, including handling missing values, feature encoding, and scaling to ensure data quality and suitability for machine learning models. Key insights gained from the EDA highlighted glucose levels and BMI as the most significant predictors of diabetes risk.

A variety of machine learning algorithms were employed, including Logistic Regression, Random Forest, Gradient Boosting, and Support Vector Machines (SVM). These models were evaluated based on critical performance metrics such as accuracy, precision, recall, F1 score, and AUC-ROC. Cross-validation techniques ensured the robustness and generalizability of the models. Random Forest emerged as the best-performing model, achieving an F1 score of 0.957, making it the most reliable for deployment in real-world scenarios.

The outcomes of this project demonstrate the potential of AI to revolutionize diabetes diagnosis, offering a scalable solution for early detection. By deploying this model in healthcare systems or mobile applications, it can serve as a valuable tool for doctors and patients alike, enabling proactive management of diabetes. Additionally, this approach sets a precedent for using AI in tackling other chronic diseases, fostering a data-driven approach to personalized healthcare.

Future work includes incorporating larger and more diverse datasets to enhance the model's robustness and exploring the integration of deep learning techniques for further accuracy improvements. GlucoSense represents a significant step toward democratizing access to healthcare by providing a technology-driven solution to a critical global health issue.

Introduction

Diabetes is a chronic condition that affects millions of people worldwide, contributing significantly to morbidity and mortality rates. It is characterized by elevated blood glucose levels due to inadequate insulin production or the body's inability to use insulin effectively. The World Health Organization (WHO) identifies diabetes as one of the leading global health challenges, with an estimated 463 million adults living with the condition in 2019, a figure expected to rise dramatically in the coming decades. This alarming trend underscores the critical need for early detection and intervention to manage the disease effectively and reduce the risk of severe complications, such as cardiovascular disease, kidney failure, and neuropathy.

Traditional diagnostic methods for diabetes often rely on laboratory tests, such as fasting blood glucose tests, HbA1c levels, and oral glucose tolerance tests. While these methods are accurate, they are often time-consuming, resource-intensive, and inaccessible to many individuals in rural or underserved regions. The delay in diagnosis due to limited access can result in late-stage complications, making proactive healthcare solutions a necessity. In this context, artificial intelligence (AI) presents an innovative opportunity to revolutionize diabetes detection, offering scalable, efficient, and cost-effective tools for early diagnosis.

This project, titled **GlucoSense: AI-Powered Diabetes Detection for Early Intervention,** aims to harness the power of machine learning algorithms to predict diabetes risk based on readily available patient data. The dataset used for this study includes features such as glucose levels, BMI, age, and insulin levels, which are commonly associated with diabetes risk. By analyzing these factors, the project seeks to build a predictive model that can identify individuals at high risk of developing diabetes, enabling healthcare providers to prioritize preventive care.

The methodology employed in this project spans from data pre-processing and exploratory data analysis (EDA) to model selection and evaluation. Pre-processing steps, such as handling missing values, feature encoding, and scaling, were implemented to ensure data quality and compatibility with machine learning models. Various algorithms, including Logistic Regression, Random Forest, and Gradient Boosting, were trained and evaluated using performance metrics like accuracy, precision, recall, F1 score, and AUC-ROC. The best-performing model, Random Forest, was selected for its high accuracy and balanced performance across metrics, making it suitable for real-world deployment.

In conclusion, this project highlights the potential of AI in addressing one of the most pressing healthcare challenges of our time. By integrating the developed model into healthcare systems or mobile applications, GlucoSense aims to democratize access to diabetes detection tools, especially in resource-limited settings. Furthermore, this project serves as a stepping stone for applying AI solutions to other chronic conditions, fostering a future where technology and healthcare work hand-in-hand to improve lives.

Problem Statement

Early detection and intervention are critical to managing diabetes effectively and preventing complications. However, traditional diagnostic methods such as fasting glucose tests, oral glucose tolerance tests, and HbA1c measurements often require access to well-equipped laboratories and trained personnel. These resources are not readily available in rural or underserved areas, leading to delayed diagnoses and treatment. Additionally, many individuals with pre-diabetes remain undiagnosed, missing the opportunity to receive early care that could potentially reverse or slow the progression of the disease.

The lack of scalable and accessible diagnostic tools exacerbates the problem, leaving healthcare providers unable to address the increasing demand for early detection. There is a pressing need for innovative, efficient, and cost-effective solutions that can predict diabetes risk without relying solely on traditional laboratory tests. Predictive models powered by artificial intelligence (AI) offer a promising alternative, leveraging existing health data to identify individuals at risk and prioritize them for further screening or preventive interventions.

This project aims to address these challenges by developing a machine learning-based system capable of predicting diabetes risk using easily accessible patient data. By analyzing features such as glucose levels, BMI, age, and insulin levels, the proposed model seeks to provide accurate predictions that can complement traditional diagnostic methods. This approach has the potential to bridge the gap in healthcare accessibility, particularly for underserved populations, and enable timely interventions that can significantly improve patient outcomes.

The implementation of GlucoSense aligns with the broader goal of leveraging AI in healthcare to address critical challenges. By deploying this model in real-world settings, such as clinics, mobile health applications, or community health programs, it can serve as a practical and scalable solution to combat the diabetes epidemic. Ultimately, this project seeks to contribute to a future where technology-driven solutions enhance healthcare accessibility, efficiency, and equity.

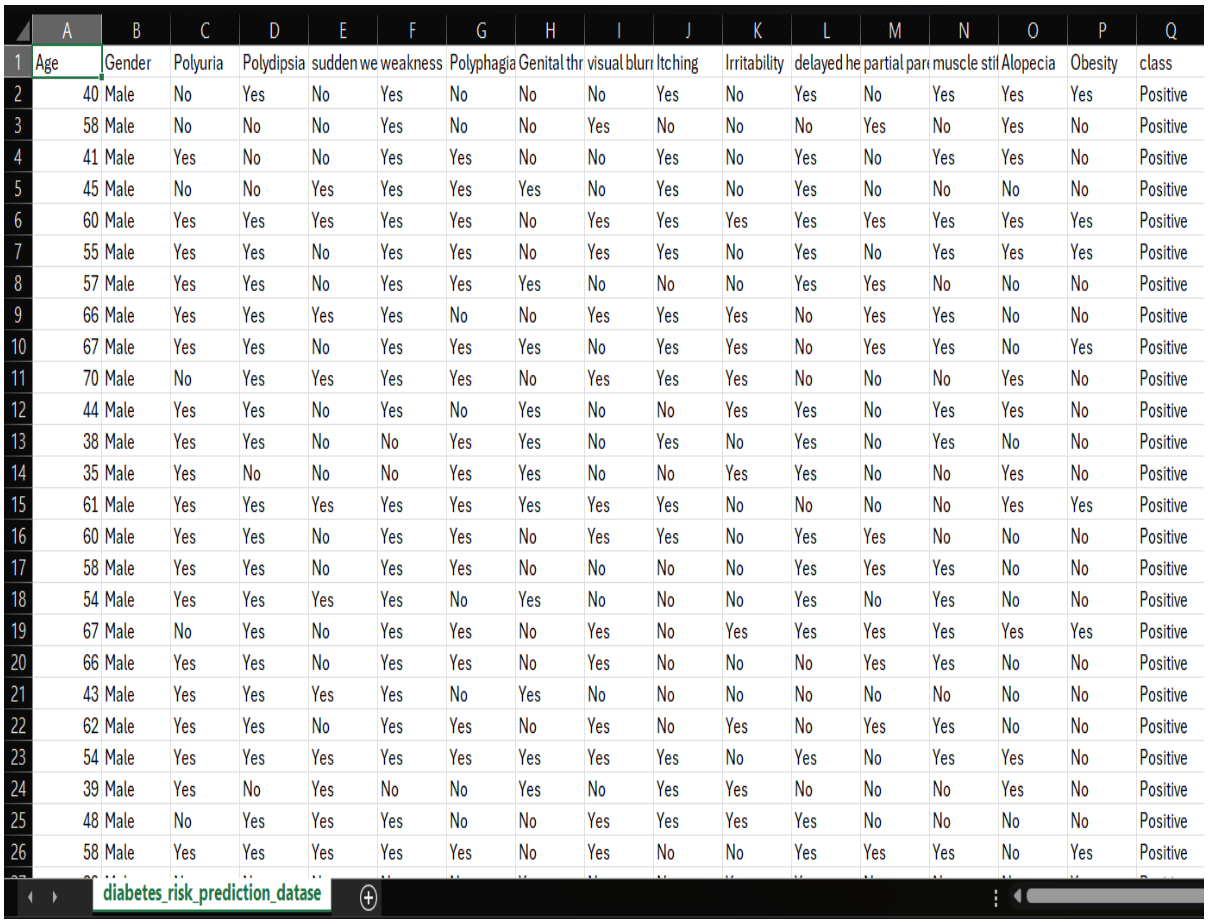
Data Collection and Dataset Details

Data Source:

For the GlucoSense project, the data used for analysis and model development was sourced from Kaggle, utilizing the **"diabetes_dataset.csv"**, which was likely obtained from publicly available repositories, such as Kaggle or research publications, focusing on diabetes risk assessment.

- **Size:**
  - Total Records: 520 (original dataset).
  - Post-Cleaning: 250 records (after handling missing values and outliers).
  - Features: 16 columns (15 independent features and 1 target variable).
- **Objective:** The dataset was designed to classify individuals as diabetic or non-diabetic based on various health attributes.

Content of "diabetes_dataset.csv" file



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	Age	Gender	Polyuria	Polydipsia	sudden we	weakness	Polyphagia	Genital thr	visual blurı	Itching	Irritability	delayed he	partial parı	muscle stif	Alopecia	Obesity	class
2	40	Male	No	Yes	No	Yes	No	No	No	Yes	No	Yes	No	Yes	Yes	Yes	Positive
3	58	Male	No	No	No	Yes	No	No	Yes	No	No	No	Yes	No	Yes	No	Positive
4	41	Male	Yes	No	No	Yes	Yes	No	No	Yes	No	Yes	No	Yes	Yes	No	Positive
5	45	Male	No	No	Yes	Yes	Yes	Yes	No	Yes	No	Yes	No	No	No	No	Positive
6	60	Male	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Positive
7	55	Male	Yes	Yes	No	Yes	Yes	No	Yes	Yes	No	Yes	No	Yes	Yes	Yes	Positive
8	57	Male	Yes	Yes	No	Yes	Yes	Yes	No	No	No	Yes	Yes	No	No	No	Positive
9	66	Male	Yes	Yes	Yes	Yes	No	No	Yes	Yes	Yes	No	Yes	Yes	No	No	Positive
10	67	Male	Yes	Yes	No	Yes	Yes	Yes	No	Yes	Yes	No	Yes	Yes	No	Yes	Positive
11	70	Male	No	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	No	No	No	Yes	No	Positive
12	44	Male	Yes	Yes	No	Yes	No	Yes	No	No	Yes	Yes	No	Yes	Yes	No	Positive
13	38	Male	Yes	Yes	No	No	Yes	Yes	No	Yes	No	Yes	No	Yes	No	No	Positive
14	35	Male	Yes	No	No	No	Yes	Yes	No	No	Yes	Yes	No	No	Yes	No	Positive
15	61	Male	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No	No	Yes	Yes	Positive
16	60	Male	Yes	Yes	No	Yes	Yes	No	Yes	Yes	No	Yes	Yes	No	No	No	Positive
17	58	Male	Yes	Yes	No	Yes	Yes	No	No	No	No	Yes	Yes	Yes	No	No	Positive
18	54	Male	Yes	Yes	Yes	Yes	No	Yes	No	No	No	Yes	No	Yes	No	No	Positive
19	67	Male	No	Yes	No	Yes	Yes	No	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Positive
20	66	Male	Yes	Yes	No	Yes	Yes	No	Yes	No	No	No	Yes	Yes	No	No	Positive
21	43	Male	Yes	Yes	Yes	Yes	No	Yes	No	No	No	No	No	No	No	No	Positive
22	62	Male	Yes	Yes	No	Yes	Yes	No	Yes	No	Yes	No	Yes	Yes	No	No	Positive
23	54	Male	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	No	Yes	Yes	No	Positive
24	39	Male	Yes	No	Yes	No	No	Yes	No	Yes	Yes	No	No	No	Yes	No	Positive
25	48	Male	No	Yes	Yes	Yes	No	No	Yes	Yes	Yes	Yes	No	No	No	No	Positive
26	58	Male	Yes	Yes	Yes	Yes	Yes	No	Yes	No	No	Yes	Yes	Yes	No	Yes	Positive

Features in the Dataset:

The dataset contains a mix of categorical and numerical features. Below is a detailed description of the key attributes:

- **Age (Numerical):** Represents the age of the individual in years.
- **Gender (Categorical):** Indicates the gender of the individual (Male/Female).
- **Polyuria (Categorical):** Whether the individual experiences excessive urination (Yes/No).
- **Polydipsia (Categorical):** Indicates excessive thirst (Yes/No).
- **Sudden Weight Loss (Categorical):** Whether the person experienced sudden weight loss (Yes/No).
- **Weakness (Categorical):** Reports general body weakness (Yes/No).
- **Polyphagia (Categorical):** Whether the individual experiences excessive hunger (Yes/No).
- **Genital Thrush (Categorical):** Indicates the presence of genital thrush (Yes/No).
- **Visual Blurring (Categorical):** Whether the person experiences blurred vision (Yes/No).
- **Itching (Categorical):** Indicates skin itching (Yes/No).
- **Irritability (Categorical):** Reports irritability symptoms (Yes/No).
- **Delayed Healing (Categorical):** Indicates delayed wound healing (Yes/No).
- **Partial Paresis (Categorical):** Reports partial paresis or muscle weakness (Yes/No).
- **Muscle Stiffness (Categorical):** Whether muscle stiffness is present (Yes/No).
- **Alopecia (Categorical):** Indicates hair loss or alopecia (Yes/No).
- **Target Variable - Class (Categorical):**
    - Indicates whether the individual has diabetes:
        - Positive (Diabetic)
        - Negative (Non-Diabetic)

Data Preprocessing (EDA) and its Outcomes

Import libraries:

As the first step we need to import libraries. These libraries provide essential tools for handling data, performing computations, creating visualizations, and implementing machine learning algorithms.

```
import warnings
warnings.filterwarnings('ignore')
import numpy as np
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split,cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import  accuracy_score, f1_score, precision_score,confusion_matrix, recall_score, roc_auc_score
from xgboost import XGBClassifier
from sklearn.ensemble import RandomForestClassifier,AdaBoostClassifier
from sklearn.svm import SVC
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn import model_selection
from sklearn.metrics import classification_report
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn import metrics
```

Load the dataset:

```
#load the dataset
df = pd.read_csv('diabetes_dataset.csv')
```

Basic Information:

The dataset consists of 520 rows and 17 columns. There are no missing values in the dataset.

```
#information about dataset
print("Dataset Information:")
print(data.info())
```

```
Dataset Information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 520 entries, 0 to 519
Data columns (total 17 columns):
 #   Column              Non-Null Count  Dtype 
---  ------              --------------  ----- 
 0   Age                 520 non-null    int64 
 1   Gender              520 non-null    object
 2   Polyuria            520 non-null    object
 3   Polydipsia          520 non-null    object
 4   sudden weight loss  520 non-null    object
 5   weakness            520 non-null    object
 6   Polyphagia          520 non-null    object
 7   Genital thrush      520 non-null    object
 8   visual blurring     520 non-null    object
 9   Itching             520 non-null    object
 10  Irritability        520 non-null    object
 11  delayed healing     520 non-null    object
 12  partial paresis     520 non-null    object
 13  muscle stiffness    520 non-null    object
 14  Alopecia            520 non-null    object
 15  Obesity             520 non-null    object
 16  class               520 non-null    object
dtypes: int64(1), object(16)
memory usage: 69.2+ KB
None
```

Drop the Duplicates:

The above dataset has 269 duplicate rows. Dropping duplicates is essential for ensuring data integrity and improving the accuracy of analyses. Duplicates can lead to biased results, as they may skew statistical calculations, machine learning models, and data visualizations by overrepresenting certain values. Removing duplicates helps in obtaining cleaner, more reliable data, ultimately leading to more accurate insights and predictions.

```
# Check for duplicate rows
duplicates = df.duplicated().sum()
print(f"\nNumber of duplicate rows: {duplicates}")

# Drop duplicates if any
df.drop_duplicates(inplace=True)
```

After dropping the duplicates, the statistics is as follows:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 251 entries, 0 to 519
Data columns (total 17 columns):
 #   Column              Non-Null Count  Dtype 
---  ------              --------------  ----- 
 0   Age                 251 non-null    int64 
 1   Gender              251 non-null    object
 2   Polyuria            251 non-null    object
 3   Polydipsia          251 non-null    object
 4   sudden weight loss  251 non-null    object
 5   weakness            251 non-null    object
 6   Polyphagia          251 non-null    object
 7   Genital thrush      251 non-null    object
 8   visual blurring     251 non-null    object
 9   Itching             251 non-null    object
 10  Irritability        251 non-null    object
 11  delayed healing     251 non-null    object
 12  partial paresis     251 non-null    object
 13  muscle stiffness    251 non-null    object
 14  Alopecia            251 non-null    object
 15  Obesity             251 non-null    object
 16  class               251 non-null    object
dtypes: int64(1), object(16)
memory usage: 35.3+ KB
```

Outlier detection:

Outlier detection using the Interquartile Range (IQR) method is a statistical technique used to identify data points that significantly differ from the rest of the data. The IQR is calculated by measuring the spread of the middle 50% of the data. Outliers are identified as any data points that fall below or above a certain range based on the IQR.

```
# Function to find outliers in a dataset using the IQR method
def find_outliers_iqr(data):
    outliers_dict = {}
    for column in data.select_dtypes(include=[np.number]).columns:
        Q1 = data[column].quantile(0.25)  # 1st quartile (25th percentile)
        Q3 = data[column].quantile(0.75)  # 3rd quartile (75th percentile)
        IQR = Q3 - Q1  # Interquartile Range
        # Define the lower and upper bounds for outliers
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR
        # Identify outliers in the current column
        outliers = data[(data[column] < lower_bound) | (data[column] > upper_bound)]
        # Store outliers in the dictionary
        outliers_dict[column] = outliers
        # Print outliers for the current column
        print(f"\nOutliers in '{column}':")
        print(outliers)
    return outliers_dict  # Return the dictionary containing outliers for each column
# Call the function to find outliers in the dataset 'data'
outliers = find_outliers_iqr(data)
```

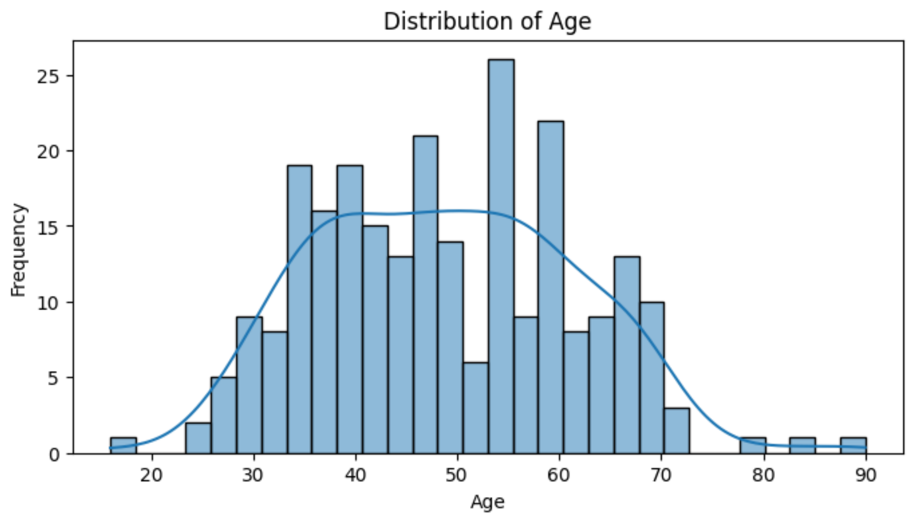
```
Outliers in 'Age':
    Age  Gender Polyuria Polydipsia sudden weight loss weakness Polyphagia  \
102  90  Female       No        Yes                Yes       No         No

   Genital thrush visual blurring Itching Irritability delayed healing  \
102            Yes             Yes     Yes           No              No

   partial paresis muscle stiffness Alopecia Obesity    class
102              No              Yes      Yes      No  Positive
```

Univariate analysis:

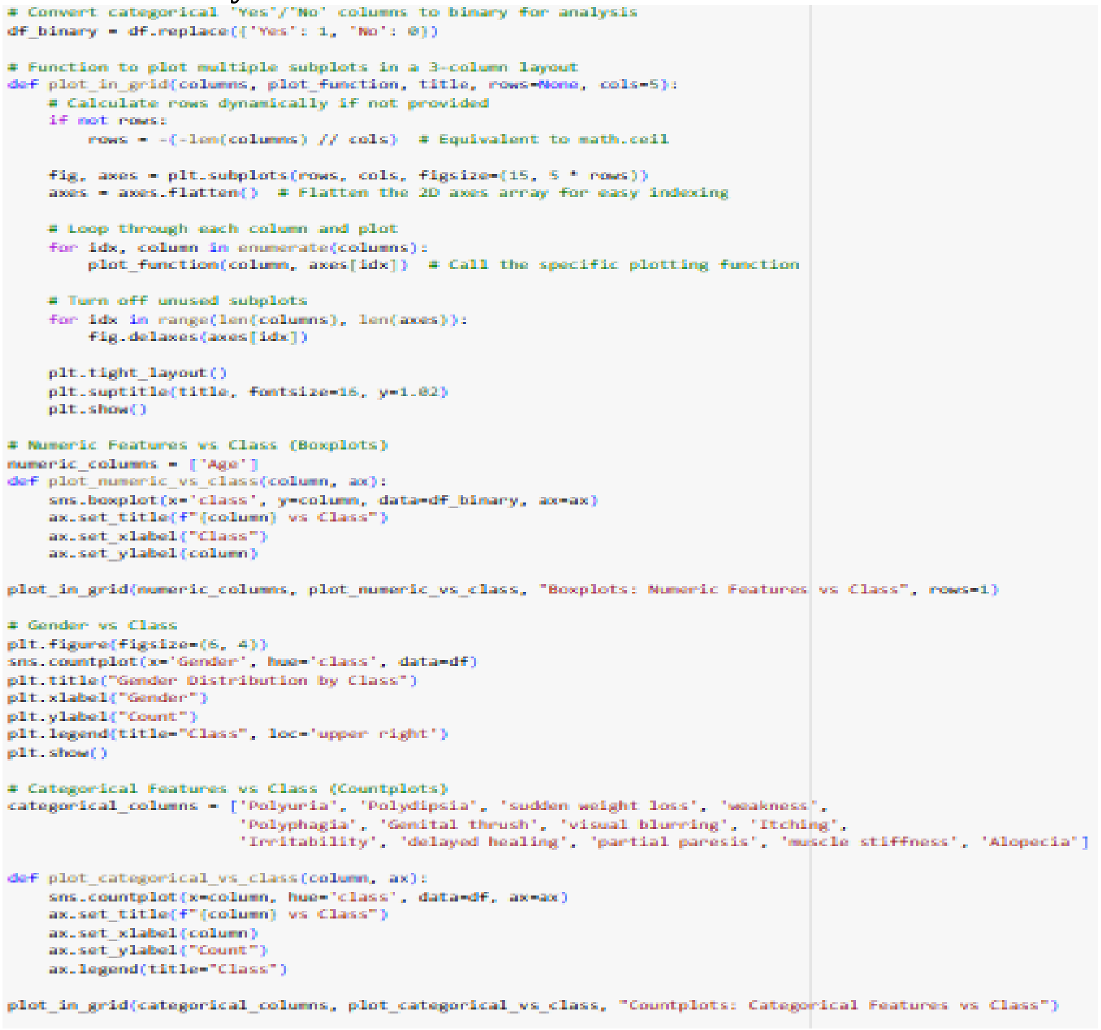
Univariate analysis involves examining a single variable to understand its distribution, central tendency (mean, median, mode), and spread (variance, standard deviation). It uses visualizations like histograms and box plots to identify patterns, outliers, and data quality issues, helping inform decisions for further analysis and preprocessing.

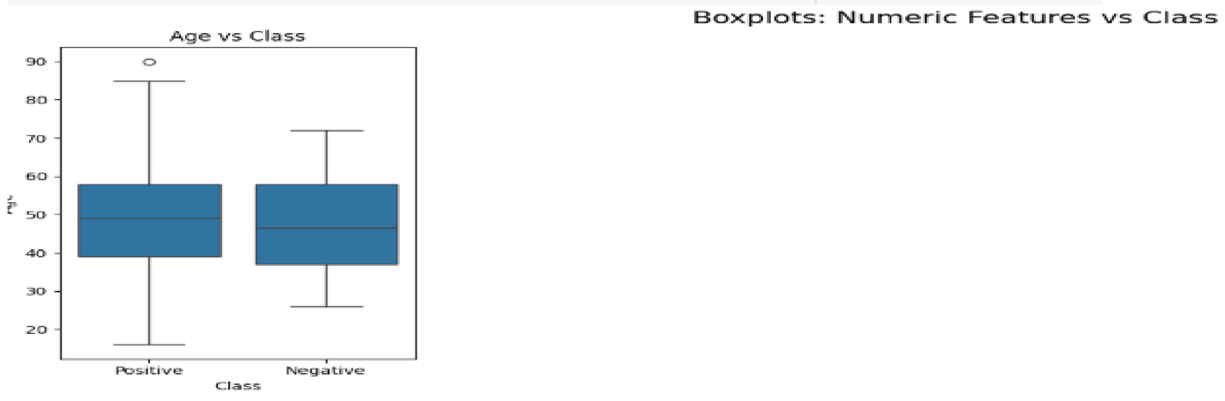
Distribution of Age:

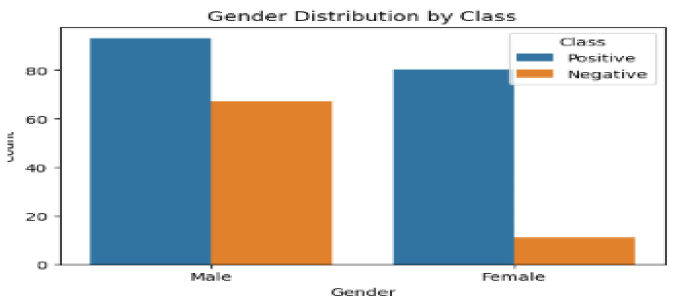
```
plt.figure(figsize=(8, 4))
sns.histplot(df['Age'], kde=True, bins=30)
plt.title(f'Distribution of Age')
plt.xlabel('Age')
plt.ylabel('Frequency')
plt.show()
```

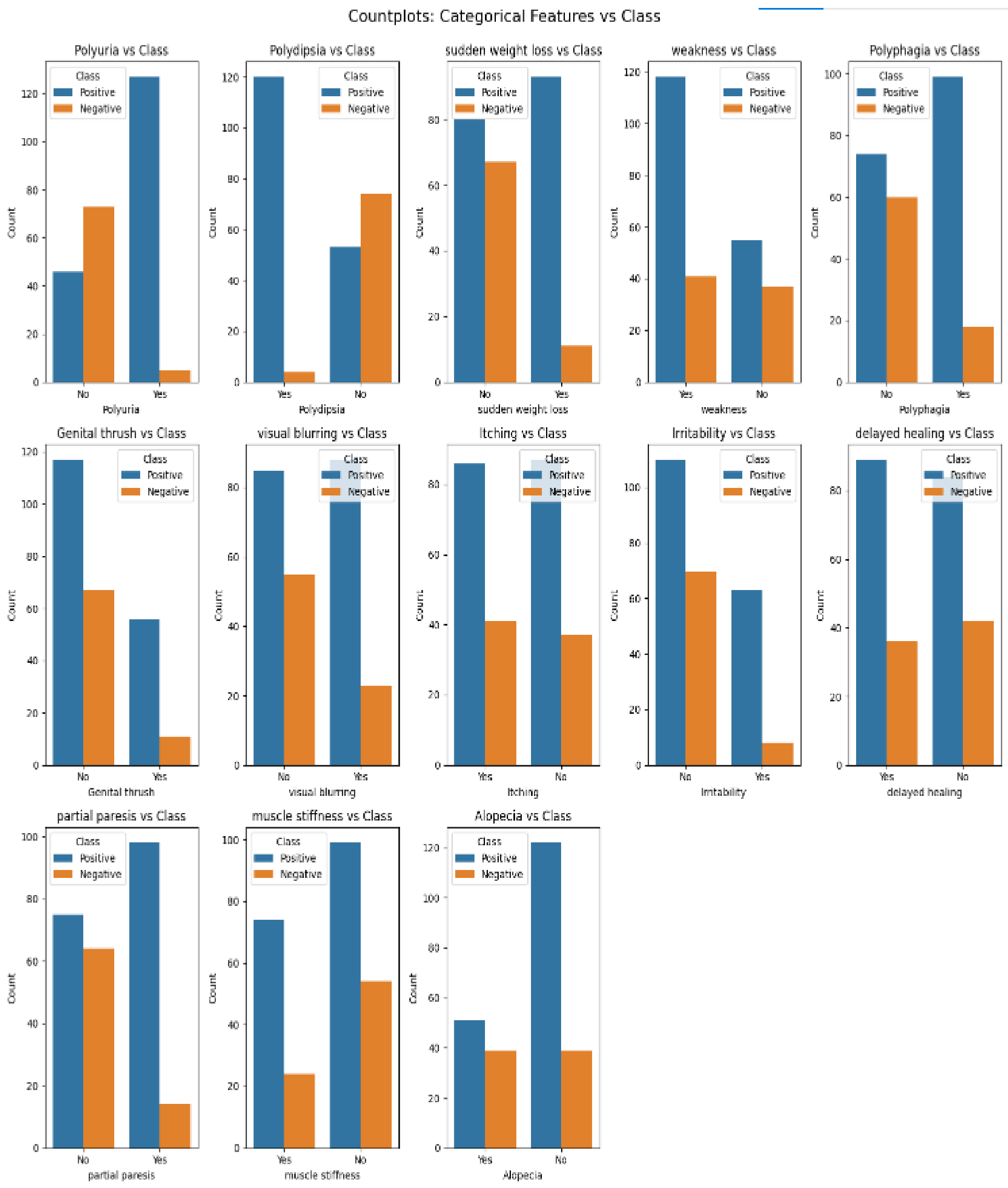


Bivariate analysis:


```
# Convert categorical 'Yes'/'No' columns to binary for analysis
df_binary = df.replace({'Yes': 1, 'No': 0})

# Function to plot multiple subplots in a 3-column layout
def plot_in_grid(columns, plot_function, title, rows=None, cols=5):
    # Calculate rows dynamically if not provided
    if not rows:
        rows = -(-len(columns) // cols)  # Equivalent to math.ceil

    fig, axes = plt.subplots(rows, cols, figsize=(15, 5 * rows))
    axes = axes.flatten()  # Flatten the 2D axes array for easy indexing

    # Loop through each column and plot
    for idx, column in enumerate(columns):
        plot_function(column, axes[idx])  # Call the specific plotting function

    # Turn off unused subplots
    for idx in range(len(columns), len(axes)):
        fig.delaxes(axes[idx])

    plt.tight_layout()
    plt.suptitle(title, fontsize=16, y=1.02)
    plt.show()

# Numeric Features vs Class (Boxplots)
numeric_columns = ['Age']
def plot_numeric_vs_class(column, ax):
    sns.boxplot(x='class', y=column, data=df_binary, ax=ax)
    ax.set_title(f"{column} vs Class")
    ax.set_xlabel("Class")
    ax.set_ylabel(column)

plot_in_grid(numeric_columns, plot_numeric_vs_class, "Boxplots: Numeric Features vs Class", rows=1)

# Gender vs Class
plt.figure(figsize=(6, 4))
sns.countplot(x='Gender', hue='class', data=df)
plt.title("Gender Distribution by Class")
plt.xlabel("Gender")
plt.ylabel("Count")
plt.legend(title="Class", loc='upper right')
plt.show()

# Categorical Features vs Class (Countplots)
categorical_columns = ['Polyuria', 'Polydipsia', 'sudden weight loss', 'weakness',
                       'Polyphagia', 'Genital thrush', 'visual blurring', 'Itching',
                       'Irritability', 'delayed healing', 'partial paresis', 'muscle stiffness', 'Alopecia']

def plot_categorical_vs_class(column, ax):
    sns.countplot(x=column, hue='class', data=df, ax=ax)
    ax.set_title(f"{column} vs Class")
    ax.set_xlabel(column)
    ax.set_ylabel("Count")
    ax.legend(title="Class")

plot_in_grid(categorical_columns, plot_categorical_vs_class, "Countplots: Categorical Features vs Class")
```






Countplots: Categorical Features vs Class
Polyuria vs Class
Polydipsia vs Class
sudden weight loss vs Class
weakness vs Class
Polyphagia vs Class
Genital thrush vs Class
visual blurring vs Class
Itching vs Class
Irritability vs Class
delayed healing vs Class
partial paresis vs Class
muscle stiffness vs Class
Alopecia vs Class
Class
Positive
Negative
Count
Yes
No
Polyuria
Polydipsia
sudden weight loss
weakness
Polyphagia
Genital thrush
visual blurring
Itching
Irritability
delayed healing
partial paresis
muscle stiffness
Alopecia

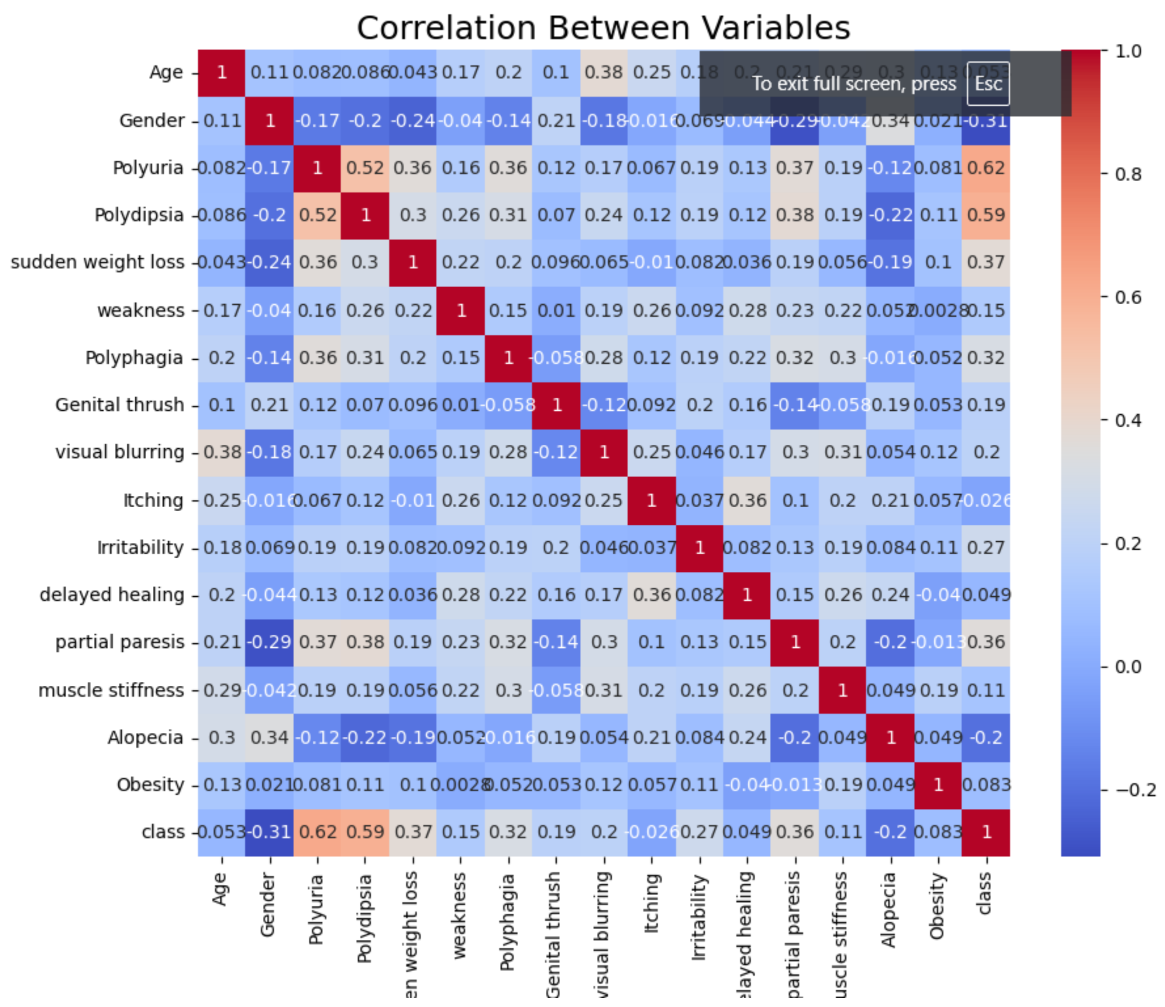
Key Insights:

The features polyuria, polydipsia, sudden weight loss, weakness, and genital thrush show strong distinctions between diabetic and non-diabetic cases, suggesting these are particularly indicative of diabetes. Visual blurring, itching, delayed healing, and polyphagia also demonstrate significant differences and are likely valuable indicators. Features like irritability, partial paresis, and muscle stiffness are moderately associated with diabetes. Gender, alopecia, and obesity appear less indicative on their own but may be relevant when combined with other features.

Correlation matrix:

A correlation matrix is a table that shows the pairwise correlation coefficients between variables in a dataset. It helps identify relationships between variables, indicating how strongly they are related, with values ranging from -1 (perfect negative correlation) to 1 (perfect positive correlation), and 0 indicating no correlation.

```
labelencoder = LabelEncoder()
# Replace 'diabetes_df' with 'df'
for i in df.columns[1:]: # Iterate over columns instead of row indices
    df[i] = labelencoder.fit_transform(df[i])

plt.figure(figsize = (10,8))
sns.heatmap(df.corr(), annot=True, fmt='.2g',cmap='coolwarm')
plt.title('Correlation Between Variables', fontsize=18);
```



High Correlations with Diabetes Class:

Polyuria (0.67), Polydipsia (0.65), and sudden weight loss (0.44) have a strong positive correlation with the diabetes class, indicating these symptoms are highly associated with diabetes in the dataset.

Partial paresis (0.43) and Polyphagia (0.34) also show moderate correlations with the diabetes class.

Inter-relationships Between Symptoms:

Polyuria and Polydipsia show a strong correlation (0.60), suggesting these symptoms frequently occur together.

Visual blurring is moderately correlated with muscle stiffness (0.41) and partial paresis (0.36), indicating a possible association between these symptoms.

Negative Correlations:

Alopecia has a notable negative correlation with Polydipsia (-0.31) and the diabetes class (-0.27), suggesting that its presence might be less common among individuals with diabetes in this dataset.

Symptom Frequency in Positive Diabetes Cases:

Prevalence Across Age Groups:

Symptoms are more frequently observed in the 40–60 age range for individuals with diabetes, indicating a higher incidence of diabetes-related symptoms in middle-aged groups.

Top Symptoms:

Polyuria and Polydipsia:

These symptoms are consistently higher across most age groups, particularly among middle-aged individuals, reinforcing their importance as diabetes indicators.

Weakness and sudden weight loss:

These symptoms show a noticeable frequency, especially in the 31-50 age groups, suggesting they could be predictive features for early detection in younger adults.

Elderly Groups:

The 60+ age group also shows a high frequency of symptoms, but with fewer cases compared to the middle-aged group, possibly due to fewer data samples or natural attrition of health in older age groups.

Symptom Frequency in Negative Diabetes Cases

Lower Overall Symptom Frequency:

For individuals without diabetes, symptoms like Polyuria and Polydipsia are notably less frequent across all age groups, confirming that these symptoms are strongly associated with diabetes.

Symptom Occurrence:

Symptoms such as Alopecia and Obesity show some presence across age groups even in negative cases, indicating that these factors might be influenced by other conditions not directly related to diabetes.

Comparative Age Group Trends

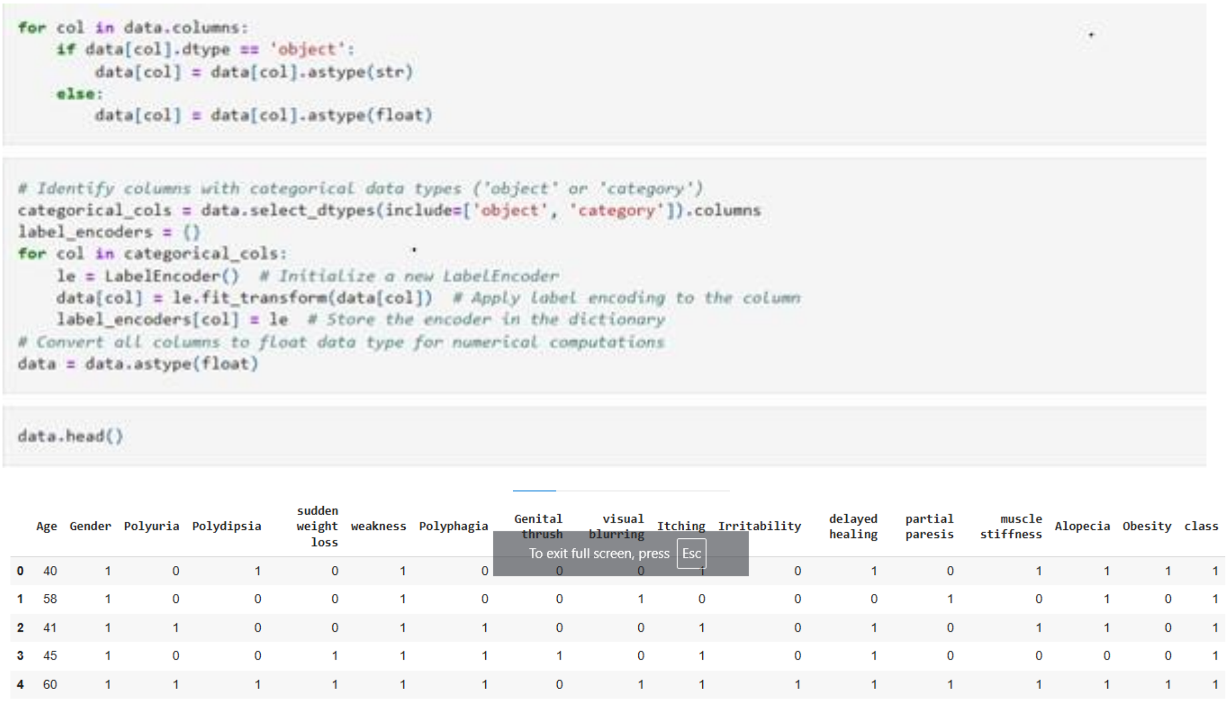
Higher Symptom Rates in Middle Age for Diabetics:

Positive diabetes cases exhibit a peak in symptoms in the 40-60 age range, while negative cases do not show such a peak, reinforcing that this age group is a critical period for diabetes management and intervention. Young and Elderly Groups: Both positive and negative cases show fewer symptoms in <20 and 70+ age groups, possibly due to fewer data points in these age brackets or lower incidenc

Feature Selection and dimension reduction approaches

Standardize data types:

Standardizing data types ensures consistency and simplifies data processing. The process involves, Converting columns with discrete values (e.g., "Yes", "No", or categories like "Male", "Female") into a standard format, such as the category type. This reduces memory usage and speeds up operations. Ensuring all numerical data, whether integers or floats, are stored in a consistent format (e.g., float). This prevents errors during computations or scaling. A consistent data type for similar data makes operations like statistical analysis, machine learning, and visualization more reliable and efficient.



```
for col in data.columns:
    if data[col].dtype == 'object':
        data[col] = data[col].astype(str)
    else:
        data[col] = data[col].astype(float)
```

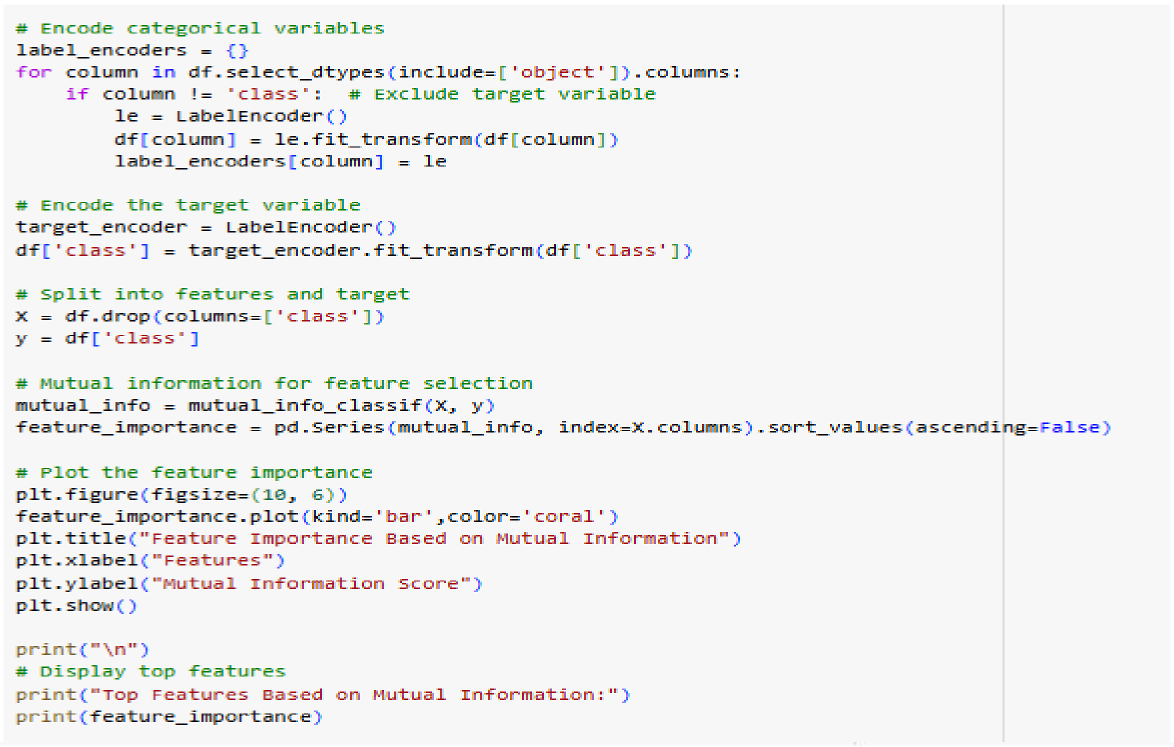
```
# Identify columns with categorical data types ('object' or 'category')
categorical_cols = data.select_dtypes(include=['object', 'category']).columns
label_encoders = {}
for col in categorical_cols:
    le = LabelEncoder()  # Initialize a new LabelEncoder
    data[col] = le.fit_transform(data[col])  # Apply label encoding to the column
    label_encoders[col] = le  # Store the encoder in the dictionary
# Convert all columns to float data type for numerical computations
data = data.astype(float)
```

```
data.head()
```

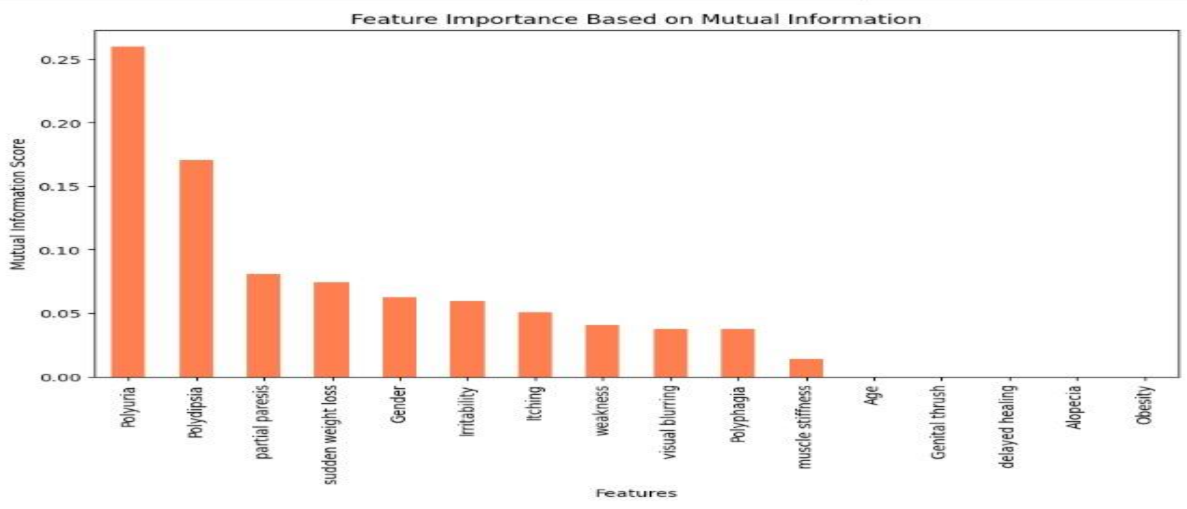
	Age	Gender	Polyuria	Polydipsia	sudden weight loss	weakness	Polyphagia	Genital thrush	visual blurring	Itching	Irritability	delayed healing	partial paresis	muscle stiffness	Alopecia	Obesity	class
0	40	1	0	1	0	1	0	0	0	1	0	1	0	1	1	1	1
1	58	1	0	0	0	1	0	0	1	0	0	0	1	0	1	0	1
2	41	1	1	0	0	1	1	0	0	1	0	1	0	1	1	0	1
3	45	1	0	0	1	1	1	1	0	1	0	1	0	0	0	0	1
4	60	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1

Feature selection:

- **Polyuria** and **Polydipsia** emerge as the most significant predictors, with the highest mutual information scores. These are common and early indicators of diabetes, with polyuria (excessive urination) and polydipsia (increased thirst) strongly linked to high blood sugar levels.
- Features like **Partial Paresis** (muscle weakness), **Sudden Weight Loss**, and **Gender** are also important but show a slightly lower correlation compared to the top features. These symptoms are also frequently associated with undiagnosed or uncontrolled diabetes.
- Other features like **Irritability**, **Itching**, and **Weakness** contribute moderately to the prediction of diabetes, reflecting the physical effects that may occur due to insulin resistance or high blood glucose levels.
- **Polyphagia** (excessive hunger) and **Visual Blurring** also show a moderate correlation with diabetes risk, reinforcing the symptoms that can aid in early-stage diagnosis.
- Finally, features like **Age**, **Muscle Stiffness**, and others like **Genital Thrust**, **Delayed Healing**, **Alopecia**, and **Obesity** play a lesser role in the model but are still relevant in the broader context of health and disease progression.



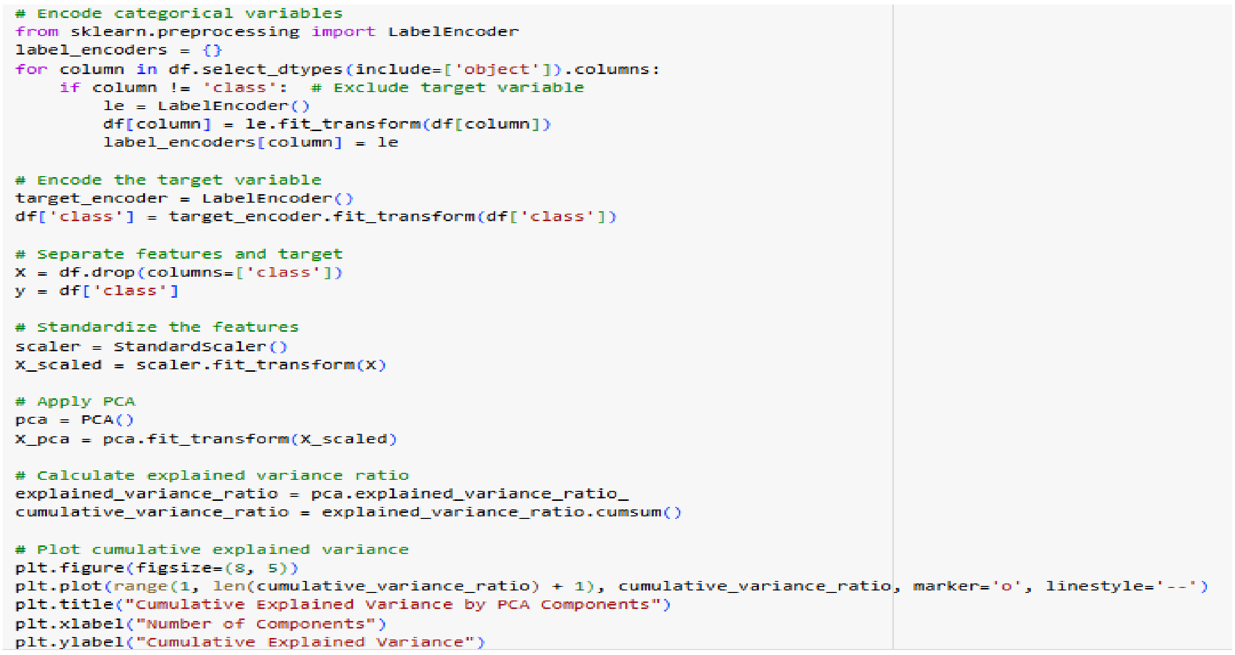
```
# Encode categorical variables
label_encoders = {}
for column in df.select_dtypes(include=['object']).columns:
    if column != 'class':  # Exclude target variable
        le = LabelEncoder()
        df[column] = le.fit_transform(df[column])
        label_encoders[column] = le

# Encode the target variable
target_encoder = LabelEncoder()
df['class'] = target_encoder.fit_transform(df['class'])

# Split into features and target
X = df.drop(columns=['class'])
y = df['class']

# Mutual information for feature selection
mutual_info = mutual_info_classif(X, y)
feature_importance = pd.Series(mutual_info, index=X.columns).sort_values(ascending=False)

# Plot the feature importance
plt.figure(figsize=(10, 6))
feature_importance.plot(kind='bar',color='coral')
plt.title("Feature Importance Based on Mutual Information")
plt.xlabel("Features")
plt.ylabel("Mutual Information Score")
plt.show()

print("\n")
# Display top features
print("Top Features Based on Mutual Information:")
print(feature_importance)
```



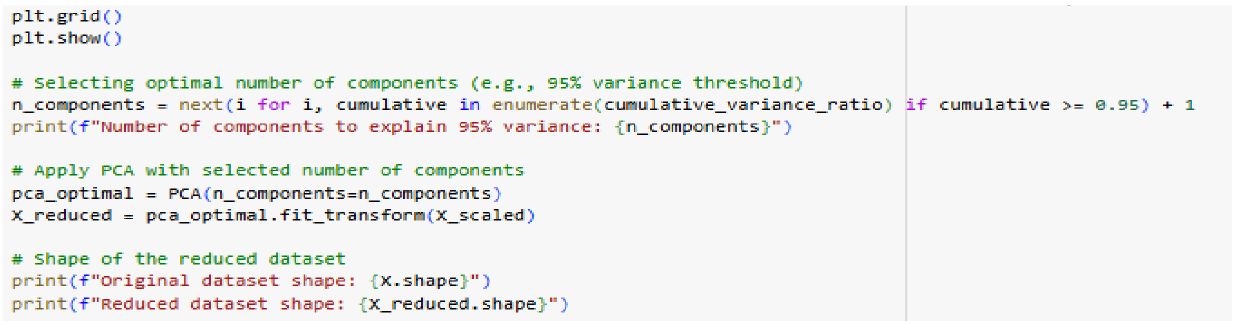
```
Top Features Based on Mutual Information:
Polyuria              0.260189
Polydipsia            0.170495
partial paresis       0.080930
sudden weight loss    0.074198
Gender                0.062457
Irritability          0.059472
Itching               0.050501
weakness              0.040430
visual blurring       0.037871
Polyphagia            0.037865
muscle stiffness      0.013687
Age                   0.000000
Genital thrush        0.000000
delayed healing       0.000000
Alopecia              0.000000
Obesity               0.000000
dtype: float64
```

Dimensionality Reduction using PCA analysis:

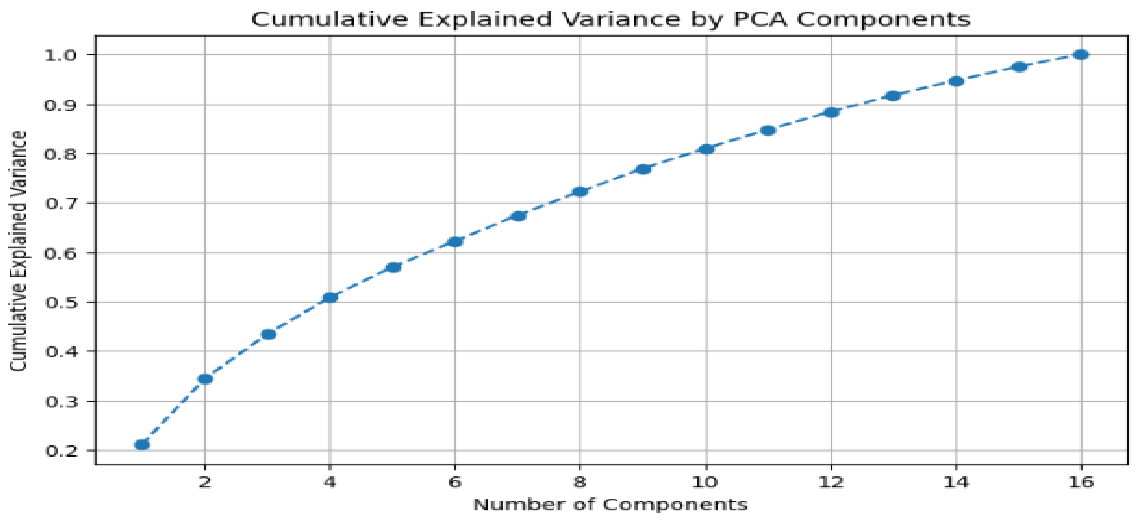
Principal Component Analysis (PCA) is a statistical technique used for dimensionality reduction. It transforms a high-dimensional dataset into a lower-dimensional space while retaining most of the dataset's variance. PCA is widely used in machine learning and data preprocessing to simplify data, reduce noise, and mitigate the curse of dimensionality.



```
# Encode categorical variables
from sklearn.preprocessing import LabelEncoder
label_encoders = {}
for column in df.select_dtypes(include=['object']).columns:
    if column != 'class':  # Exclude target variable
        le = LabelEncoder()
        df[column] = le.fit_transform(df[column])
        label_encoders[column] = le

# Encode the target variable
target_encoder = LabelEncoder()
df['class'] = target_encoder.fit_transform(df['class'])

# Separate features and target
X = df.drop(columns=['class'])
y = df['class']

# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Apply PCA
pca = PCA()
X_pca = pca.fit_transform(X_scaled)

# Calculate explained variance ratio
explained_variance_ratio = pca.explained_variance_ratio_
cumulative_variance_ratio = explained_variance_ratio.cumsum()

# Plot cumulative explained variance
plt.figure(figsize=(8, 5))
plt.plot(range(1, len(cumulative_variance_ratio) + 1), cumulative_variance_ratio, marker='o', linestyle='--')
plt.title("Cumulative Explained Variance by PCA Components")
plt.xlabel("Number of Components")
plt.ylabel("Cumulative Explained Variance")
```



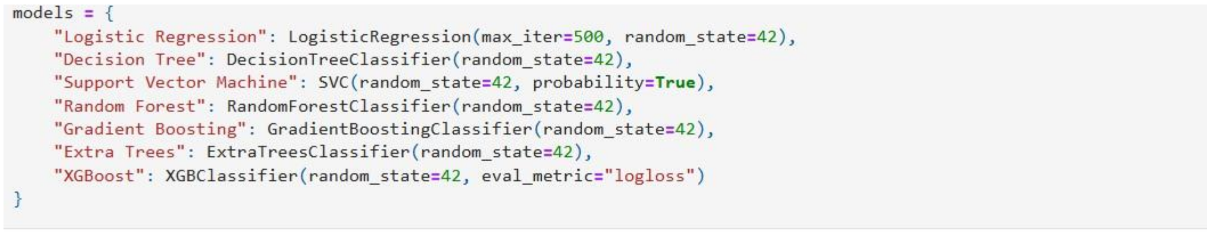
```
plt.grid()
plt.show()

# Selecting optimal number of components (e.g., 95% variance threshold)
n_components = next(i for i, cumulative in enumerate(cumulative_variance_ratio) if cumulative >= 0.95) + 1
print(f"Number of components to explain 95% variance: {n_components}")

# Apply PCA with selected number of components
pca_optimal = PCA(n_components=n_components)
X_reduced = pca_optimal.fit_transform(X_scaled)

# Shape of the reduced dataset
print(f"Original dataset shape: {X.shape}")
print(f"Reduced dataset shape: {X_reduced.shape}")
```



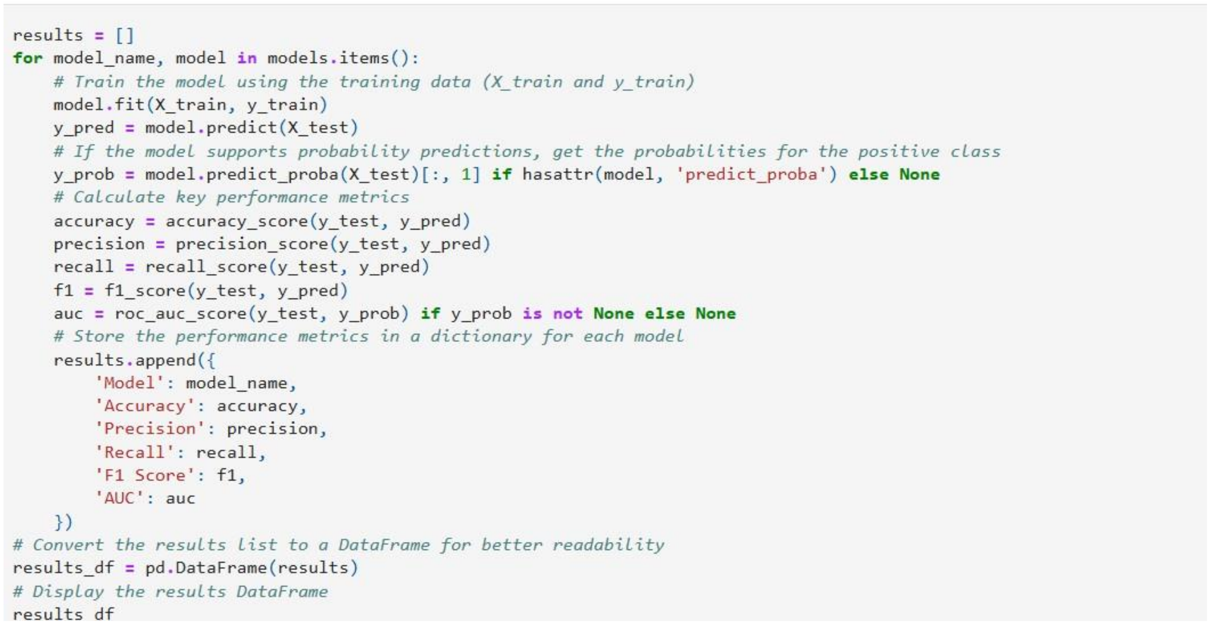
```
Number of components to explain 95% variance: 15
Original dataset shape: (251, 16)
Reduced dataset shape: (251, 15)
```

Model Building Methodology

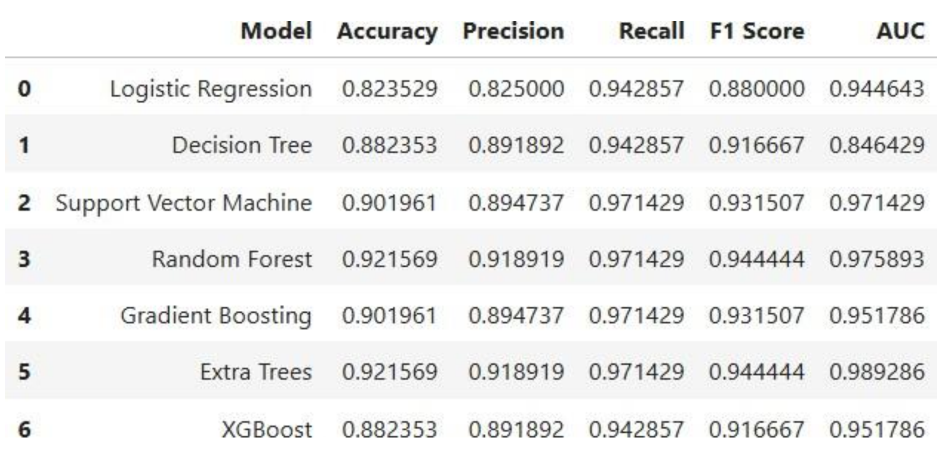
Define models:



```
models = {
    "Logistic Regression": LogisticRegression(max_iter=500, random_state=42),
    "Decision Tree": DecisionTreeClassifier(random_state=42),
    "Support Vector Machine": SVC(random_state=42, probability=True),
    "Random Forest": RandomForestClassifier(random_state=42),
    "Gradient Boosting": GradientBoostingClassifier(random_state=42),
    "Extra Trees": ExtraTreesClassifier(random_state=42),
    "XGBoost": XGBClassifier(random_state=42, eval_metric="logloss")
}
```



```
results = []
for model_name, model in models.items():
    # Train the model using the training data (X_train and y_train)
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    # If the model supports probability predictions, get the probabilities for the positive class
    y_prob = model.predict_proba(X_test)[:, 1] if hasattr(model, 'predict_proba') else None
    # Calculate key performance metrics
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred)
    recall = recall_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)
    auc = roc_auc_score(y_test, y_prob) if y_prob is not None else None
    # Store the performance metrics in a dictionary for each model
    results.append({
        'Model': model_name,
        'Accuracy': accuracy,
        'Precision': precision,
        'Recall': recall,
        'F1 Score': f1,
        'AUC': auc
    })
# Convert the results list to a DataFrame for better readability
results_df = pd.DataFrame(results)
# Display the results DataFrame
results_df
```

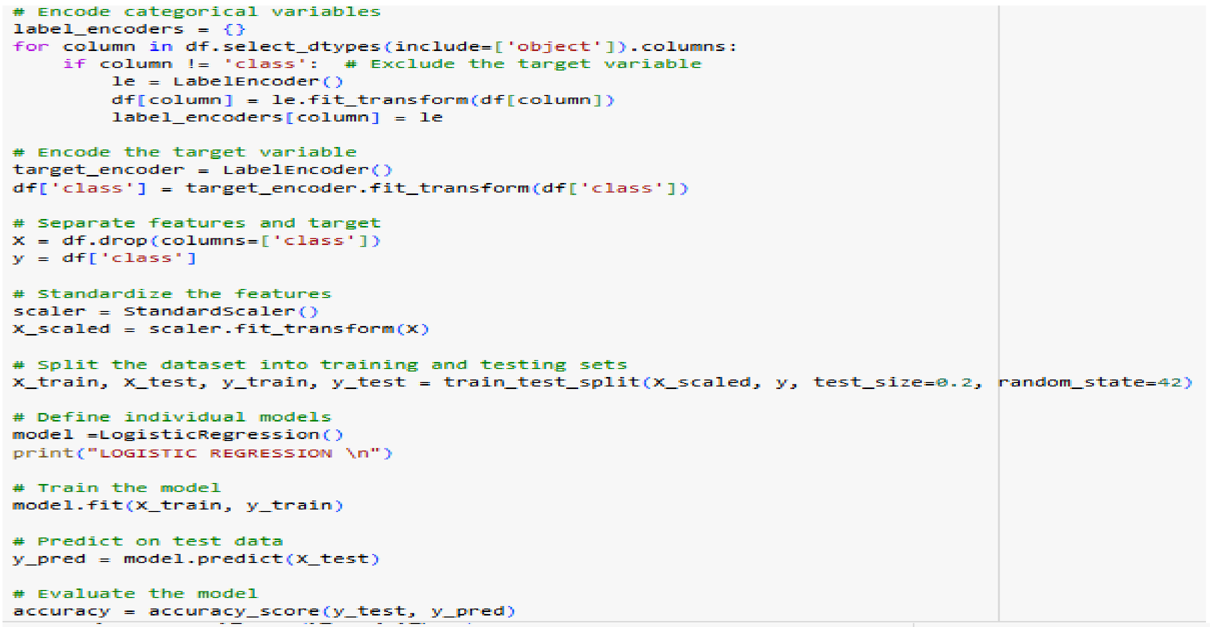


	Model	Accuracy	Precision	Recall	F1 Score	AUC
0	Logistic Regression	0.823529	0.825000	0.942857	0.880000	0.944643
1	Decision Tree	0.882353	0.891892	0.942857	0.916667	0.846429
2	Support Vector Machine	0.901961	0.894737	0.971429	0.931507	0.971429
3	Random Forest	0.921569	0.918919	0.971429	0.944444	0.975893
4	Gradient Boosting	0.901961	0.894737	0.971429	0.931507	0.951786
5	Extra Trees	0.921569	0.918919	0.971429	0.944444	0.989286
6	XGBoost	0.882353	0.891892	0.942857	0.916667	0.951786

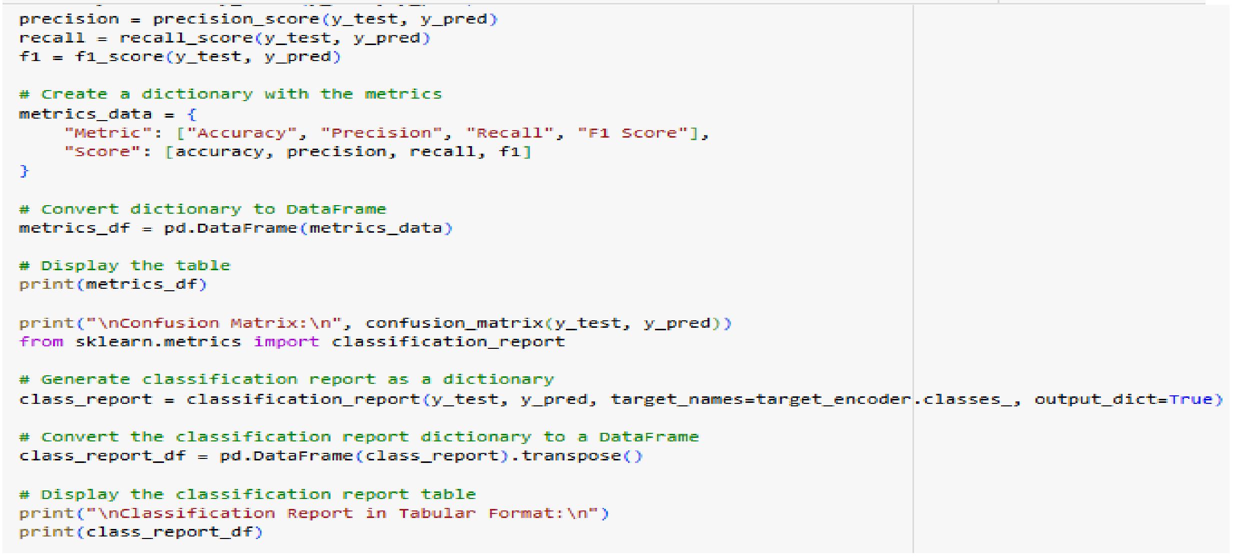
Hyperparameter Tuning:

Logistic Regression:

Logistic regression is a statistical method used for binary classification problems, predicting the probability of an outcome belonging to one of two categories. It models the relationship between input features and the target variable using a sigmoid function, ensuring the output lies between 0 and 1.



```python
# Encode categorical variables
label_encoders = {}
for column in df.select_dtypes(include=['object']).columns:
    if column != 'class':  # Exclude the target variable
        le = LabelEncoder()
        df[column] = le.fit_transform(df[column])
        label_encoders[column] = le

# Encode the target variable
target_encoder = LabelEncoder()
df['class'] = target_encoder.fit_transform(df['class'])

# Separate features and target
X = df.drop(columns=['class'])
y = df['class']

# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)

# Define individual models
model =LogisticRegression()
print("LOGISTIC REGRESSION \n")

# Train the model
model.fit(X_train, y_train)

# Predict on test data
y_pred = model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
```

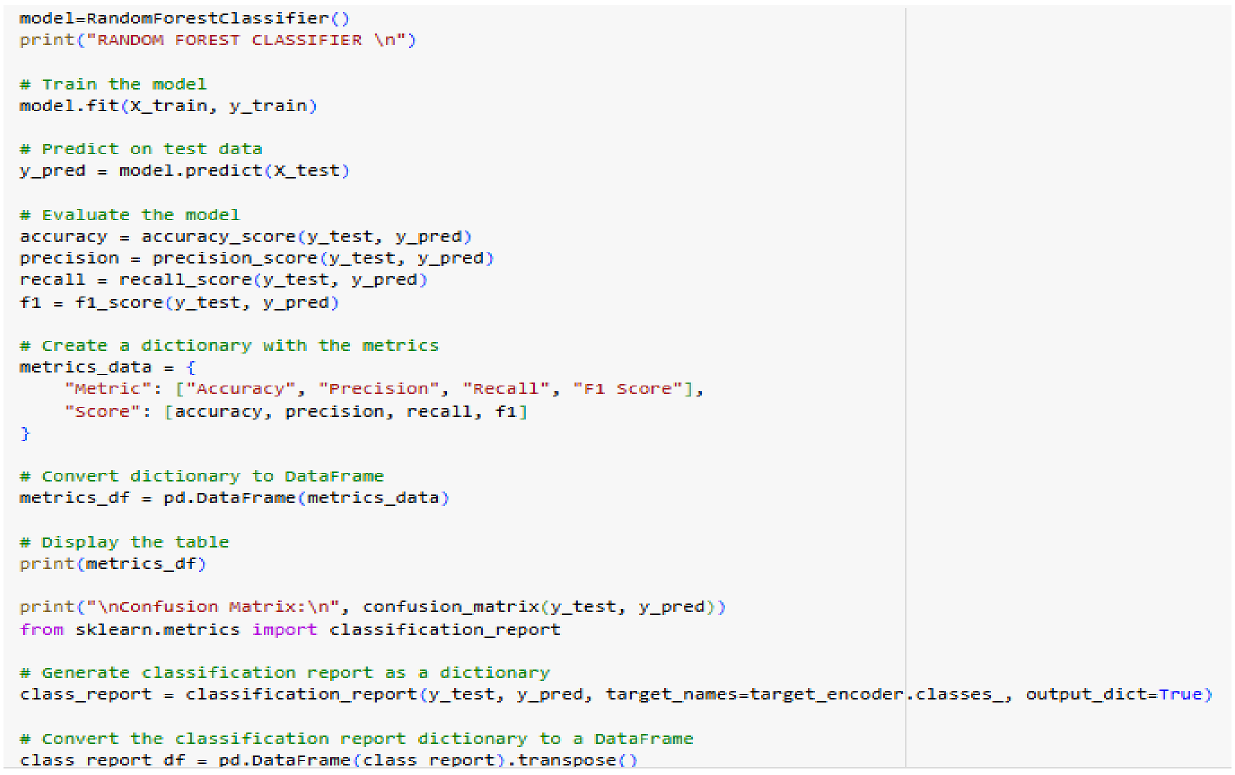


```python
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

# Create a dictionary with the metrics
metrics_data = {
    "Metric": ["Accuracy", "Precision", "Recall", "F1 Score"],
    "Score": [accuracy, precision, recall, f1]
}

# Convert dictionary to DataFrame
metrics_df = pd.DataFrame(metrics_data)

# Display the table
print(metrics_df)

print("\nConfusion Matrix:\n", confusion_matrix(y_test, y_pred))
from sklearn.metrics import classification_report

# Generate classification report as a dictionary
class_report = classification_report(y_test, y_pred, target_names=target_encoder.classes_, output_dict=True)

# Convert the classification report dictionary to a DataFrame
class_report_df = pd.DataFrame(class_report).transpose()

# Display the classification report table
print("\nClassification Report in Tabular Format:\n")
print(class_report_df)
```

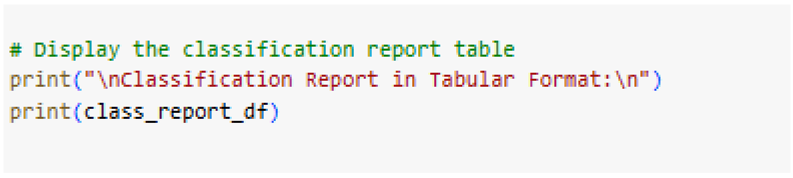
```
LOGISTIC REGRESSION

      Metric     Score
0   Accuracy  0.823529
1  Precision  0.825000
2     Recall  0.942857
3   F1 Score  0.880000

Confusion Matrix:
 [[ 9  7]
 [ 2 33]]

Classification Report in Tabular Format:

              precision    recall  f1-score    support
Negative       0.818182  0.562500  0.666667  16.000000
Positive       0.825000  0.942857  0.880000  35.000000
accuracy       0.823529  0.823529  0.823529   0.823529
macro avg      0.821591  0.752679  0.773333  51.000000
weighted avg   0.822861  0.823529  0.813072  51.000000
```

Random Forest:

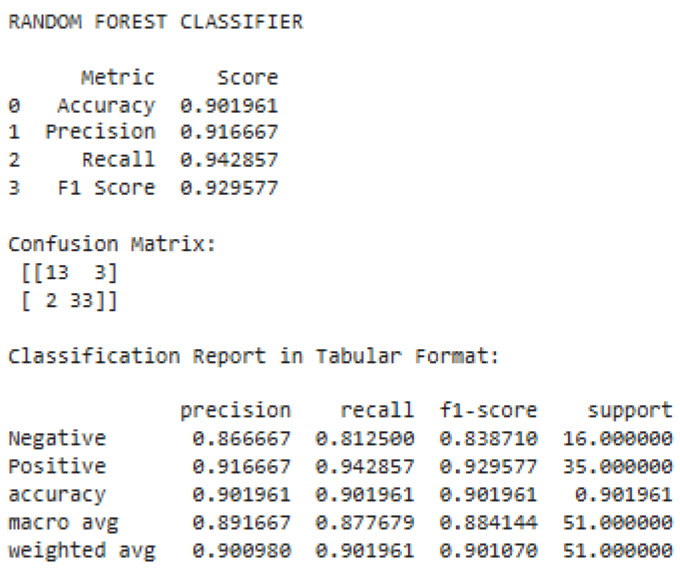
Random Forest is an ensemble machine learning algorithm that combines multiple decision trees to improve accuracy and reduce overfitting. It works by averaging or voting the predictions of individual trees, making it robust for both classification and regression tasks.



```
model=RandomForestClassifier()
print("RANDOM FOREST CLASSIFIER \n")

# Train the model
model.fit(X_train, y_train)

# Predict on test data
y_pred = model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

# Create a dictionary with the metrics
metrics_data = {
    "Metric": ["Accuracy", "Precision", "Recall", "F1 Score"],
    "Score": [accuracy, precision, recall, f1]
}

# Convert dictionary to DataFrame
metrics_df = pd.DataFrame(metrics_data)

# Display the table
print(metrics_df)

print("\nConfusion Matrix:\n", confusion_matrix(y_test, y_pred))
from sklearn.metrics import classification_report

# Generate classification report as a dictionary
class_report = classification_report(y_test, y_pred, target_names=target_encoder.classes_, output_dict=True)

# Convert the classification report dictionary to a DataFrame
class_report_df = pd.DataFrame(class_report).transpose()
```



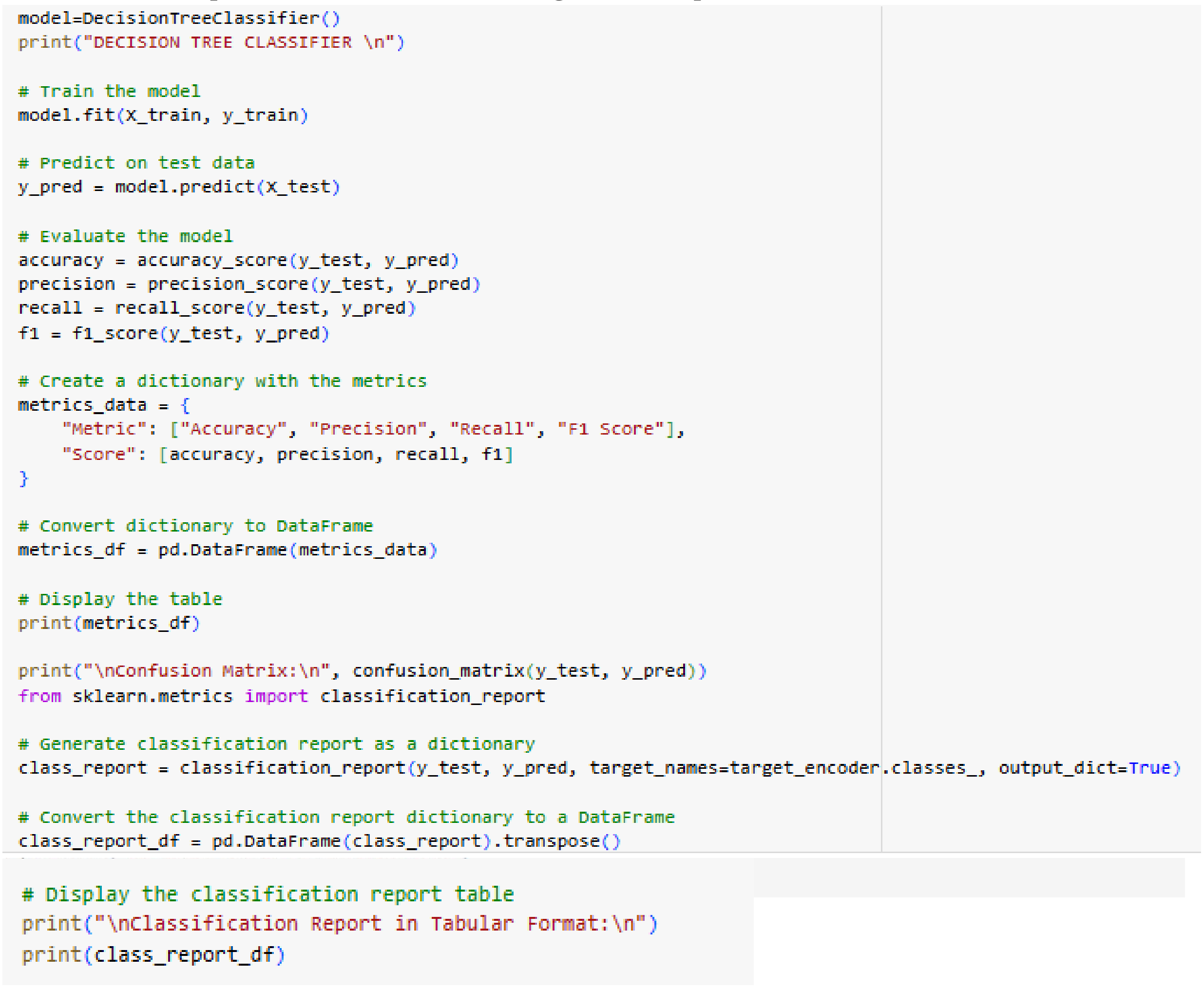
```
# Display the classification report table
print("\nClassification Report in Tabular Format:\n")
print(class_report_df)
```



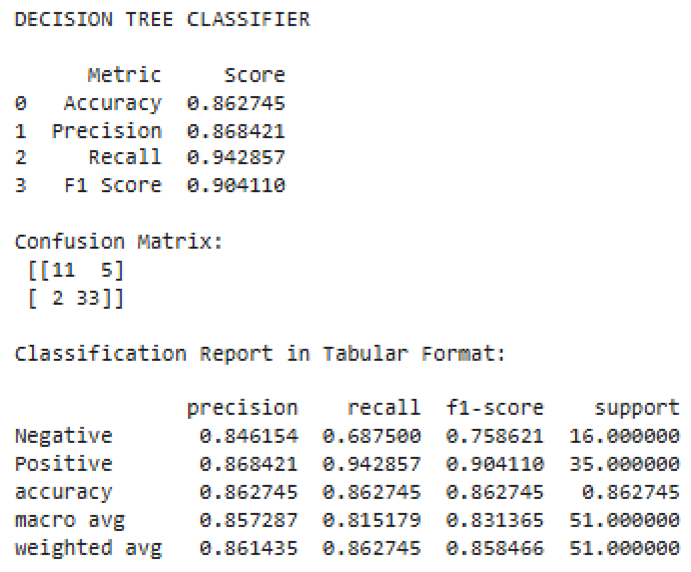
```
RANDOM FOREST CLASSIFIER

      Metric     Score
0   Accuracy  0.901961
1  Precision  0.916667
2     Recall  0.942857
3   F1 Score  0.929577

Confusion Matrix:
 [[13  3]
 [ 2 33]]

Classification Report in Tabular Format:

              precision    recall  f1-score    support
Negative       0.866667  0.812500  0.838710  16.000000
Positive       0.916667  0.942857  0.929577  35.000000
accuracy       0.901961  0.901961  0.901961   0.901961
macro avg      0.891667  0.877679  0.884144  51.000000
weighted avg   0.900980  0.901961  0.901070  51.000000
```

Decision Tree:

A decision tree is a machine learning algorithm used for classification and regression tasks. It splits data into branches based on feature values, creating a tree-like structure where each decision node represents a condition, leading to a final prediction at the leaf nodes.



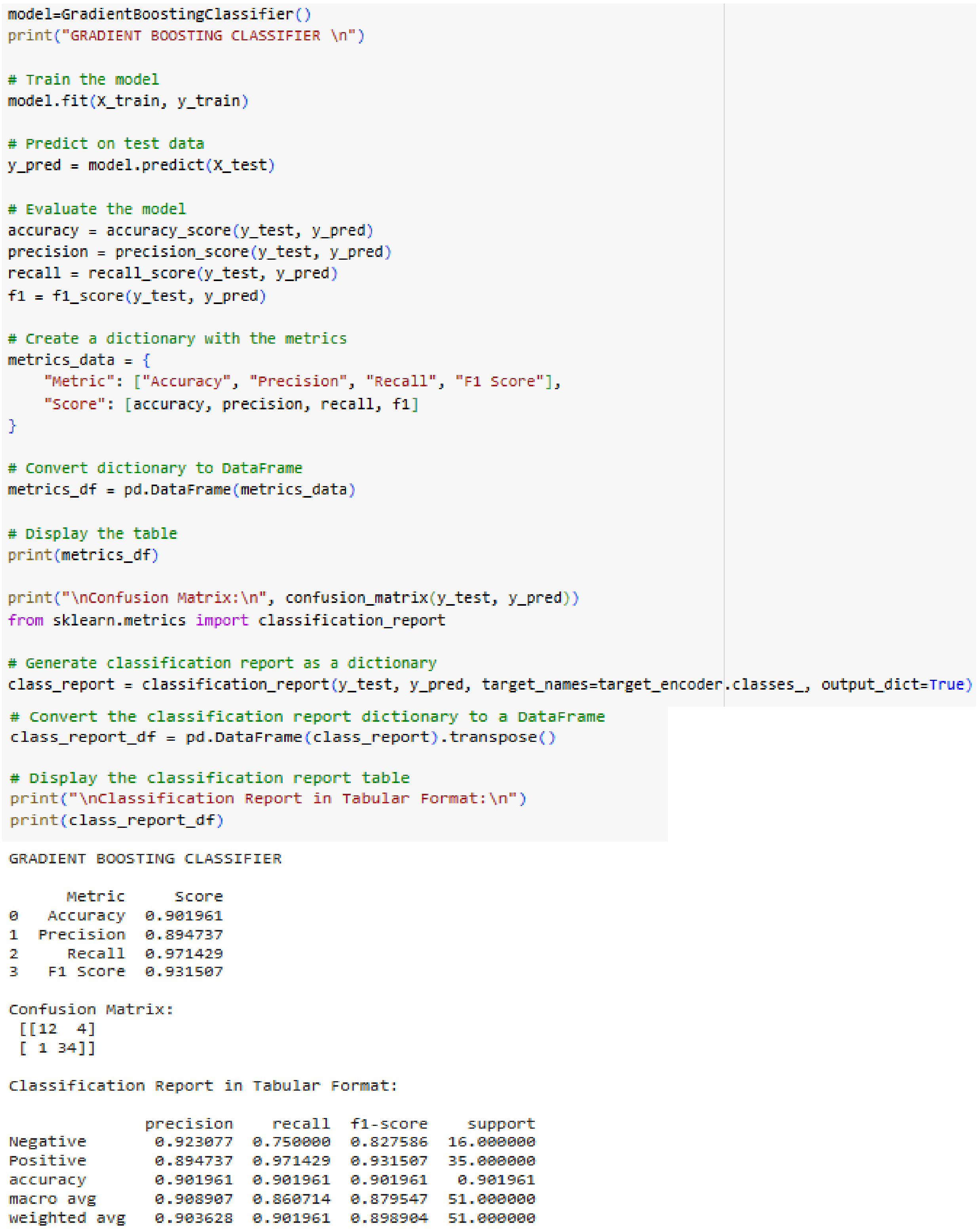
```python
model=DecisionTreeClassifier()
print("DECISION TREE CLASSIFIER \n")

# Train the model
model.fit(X_train, y_train)

# Predict on test data
y_pred = model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

# Create a dictionary with the metrics
metrics_data = {
    "Metric": ["Accuracy", "Precision", "Recall", "F1 Score"],
    "Score": [accuracy, precision, recall, f1]
}

# Convert dictionary to DataFrame
metrics_df = pd.DataFrame(metrics_data)

# Display the table
print(metrics_df)

print("\nConfusion Matrix:\n", confusion_matrix(y_test, y_pred))
from sklearn.metrics import classification_report

# Generate classification report as a dictionary
class_report = classification_report(y_test, y_pred, target_names=target_encoder.classes_, output_dict=True)

# Convert the classification report dictionary to a DataFrame
class_report_df = pd.DataFrame(class_report).transpose()

# Display the classification report table
print("\nClassification Report in Tabular Format:\n")
print(class_report_df)
```



```
DECISION TREE CLASSIFIER 

      Metric     Score
0   Accuracy  0.862745
1  Precision  0.868421
2     Recall  0.942857
3   F1 Score  0.904110

Confusion Matrix:
 [[11  5]
 [ 2 33]]

Classification Report in Tabular Format:

              precision    recall  f1-score    support
Negative       0.846154  0.687500  0.758621  16.000000
Positive       0.868421  0.942857  0.904110  35.000000
accuracy       0.862745  0.862745  0.862745   0.862745
macro avg      0.857287  0.815179  0.831365  51.000000
weighted avg   0.861435  0.862745  0.858466  51.000000
```

Gradient Boosting:

Gradient Boosting is an ensemble machine learning technique that builds models sequentially, with each new model correcting the errors of the previous ones. It combines weak learners, typically decision trees, into a strong model by optimizing a loss function to improve accuracy and reduce bias.

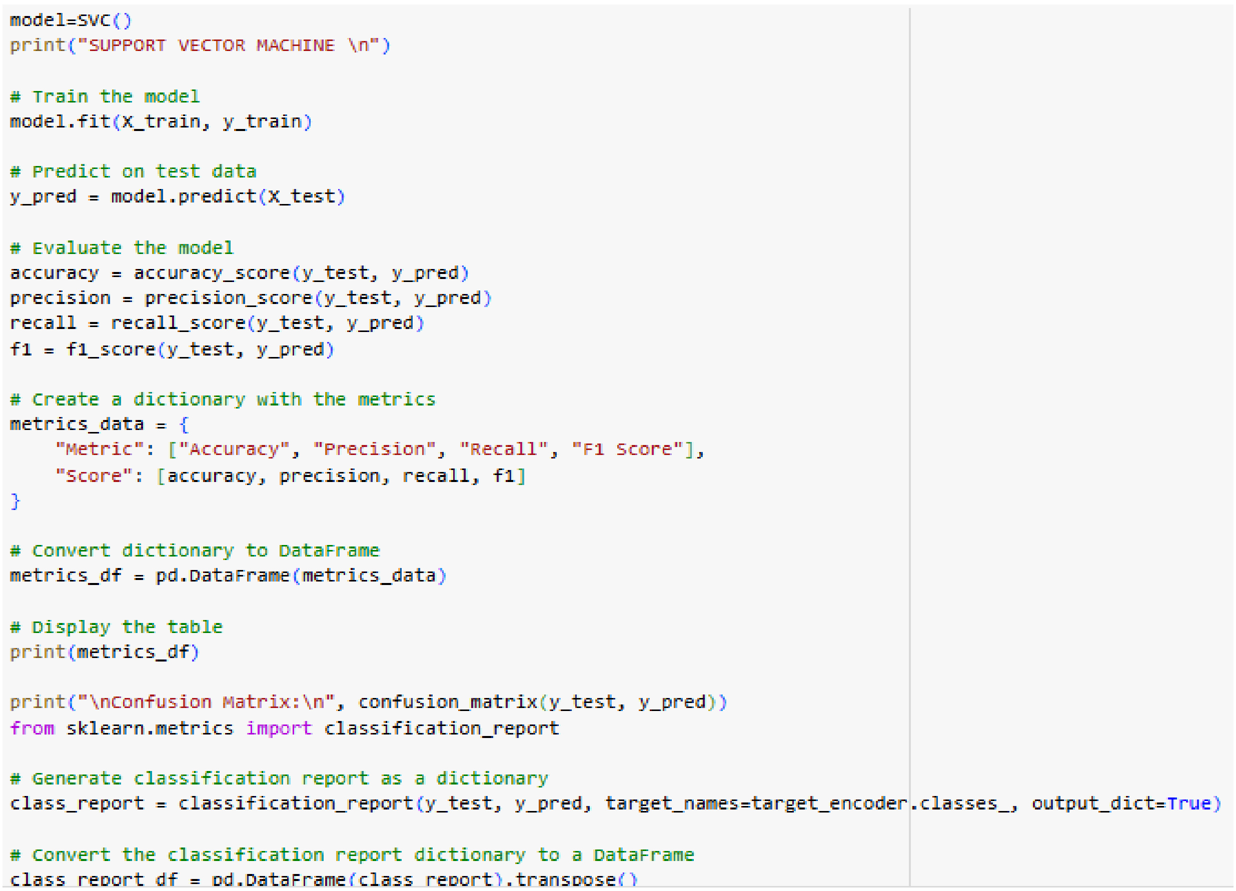


```
model=GradientBoostingClassifier()
print("GRADIENT BOOSTING CLASSIFIER \n")

# Train the model
model.fit(X_train, y_train)

# Predict on test data
y_pred = model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

# Create a dictionary with the metrics
metrics_data = {
    "Metric": ["Accuracy", "Precision", "Recall", "F1 Score"],
    "Score": [accuracy, precision, recall, f1]
}

# Convert dictionary to DataFrame
metrics_df = pd.DataFrame(metrics_data)

# Display the table
print(metrics_df)

print("\nConfusion Matrix:\n", confusion_matrix(y_test, y_pred))
from sklearn.metrics import classification_report

# Generate classification report as a dictionary
class_report = classification_report(y_test, y_pred, target_names=target_encoder.classes_, output_dict=True)
# Convert the classification report dictionary to a DataFrame
class_report_df = pd.DataFrame(class_report).transpose()

# Display the classification report table
print("\nClassification Report in Tabular Format:\n")
print(class_report_df)
```

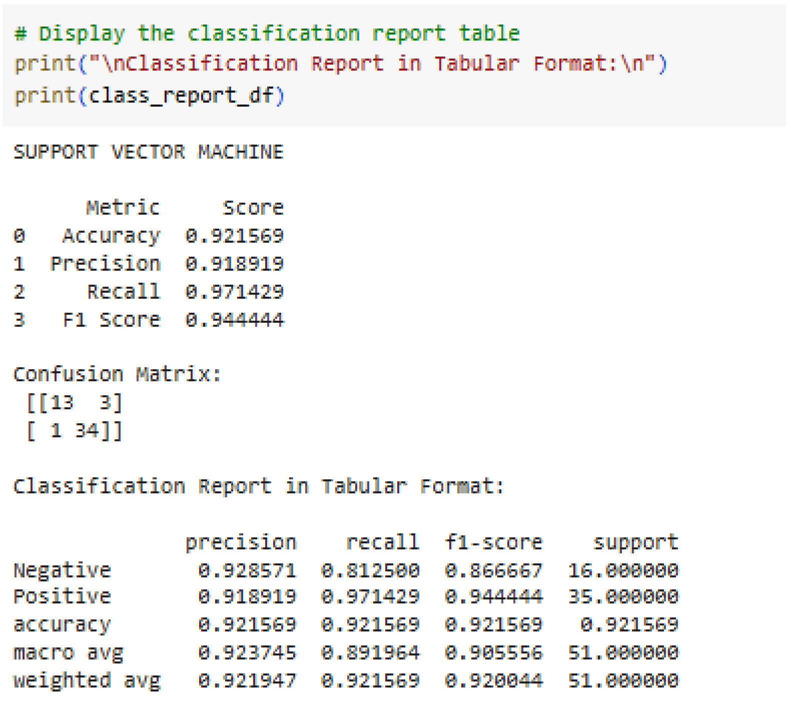
```
GRADIENT BOOSTING CLASSIFIER

      Metric     Score
0   Accuracy  0.901961
1  Precision  0.894737
2     Recall  0.971429
3   F1 Score  0.931507

Confusion Matrix:
 [[12  4]
 [ 1 34]]

Classification Report in Tabular Format:

              precision    recall  f1-score    support
Negative       0.923077  0.750000  0.827586  16.000000
Positive       0.894737  0.971429  0.931507  35.000000
accuracy       0.901961  0.901961  0.901961   0.901961
macro avg      0.908907  0.860714  0.879547  51.000000
weighted avg   0.903628  0.901961  0.898904  51.000000
```

Support Vector Machine:

Support Vector Machine (SVM) is a supervised machine learning algorithm used for classification and regression tasks. It works by finding the optimal hyperplane that best separates data points into distinct classes while maximizing the margin between them.



```
model=SVC()
print("SUPPORT VECTOR MACHINE \n")

# Train the model
model.fit(X_train, y_train)

# Predict on test data
y_pred = model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

# Create a dictionary with the metrics
metrics_data = {
    "Metric": ["Accuracy", "Precision", "Recall", "F1 Score"],
    "Score": [accuracy, precision, recall, f1]
}

# Convert dictionary to DataFrame
metrics_df = pd.DataFrame(metrics_data)

# Display the table
print(metrics_df)

print("\nConfusion Matrix:\n", confusion_matrix(y_test, y_pred))
from sklearn.metrics import classification_report

# Generate classification report as a dictionary
class_report = classification_report(y_test, y_pred, target_names=target_encoder.classes_, output_dict=True)

# Convert the classification report dictionary to a DataFrame
class_report_df = pd.DataFrame(class_report).transpose()
```

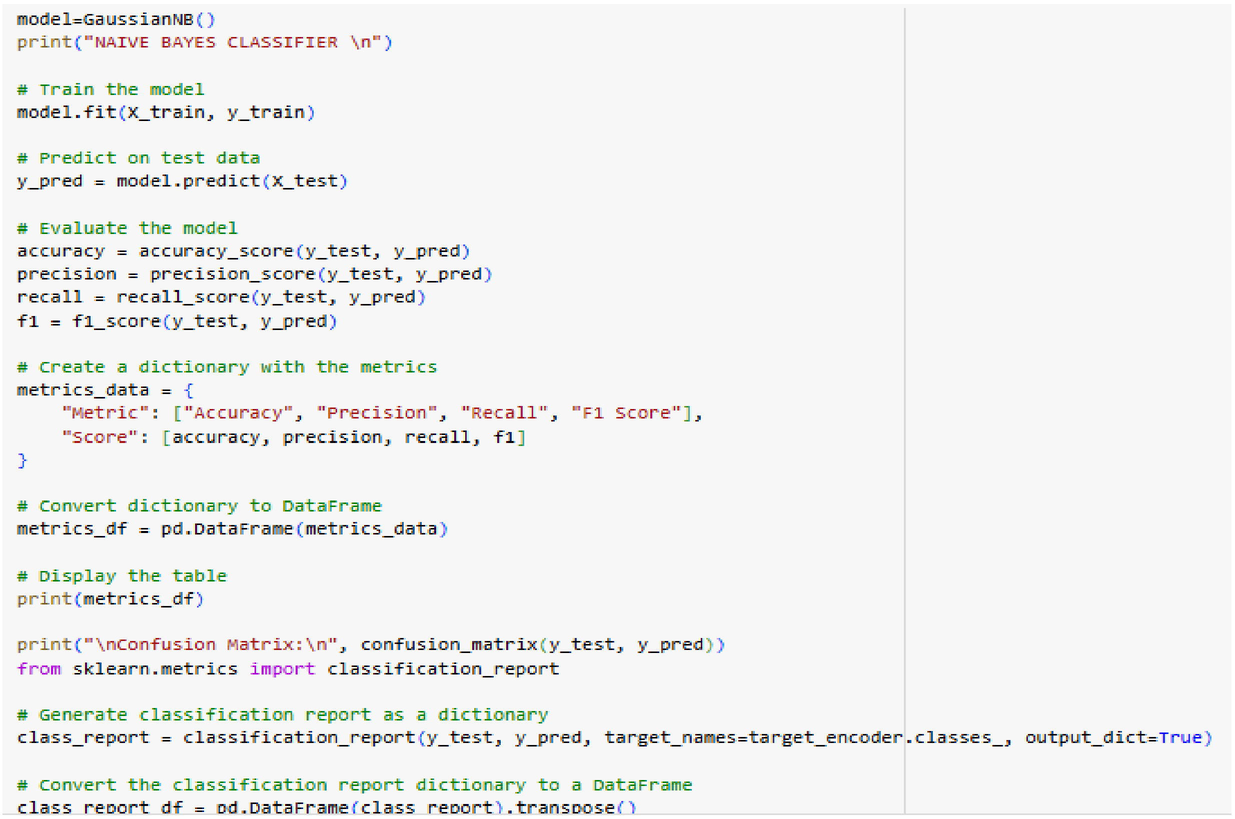


```
# Display the classification report table
print("\nClassification Report in Tabular Format:\n")
print(class_report_df)
```

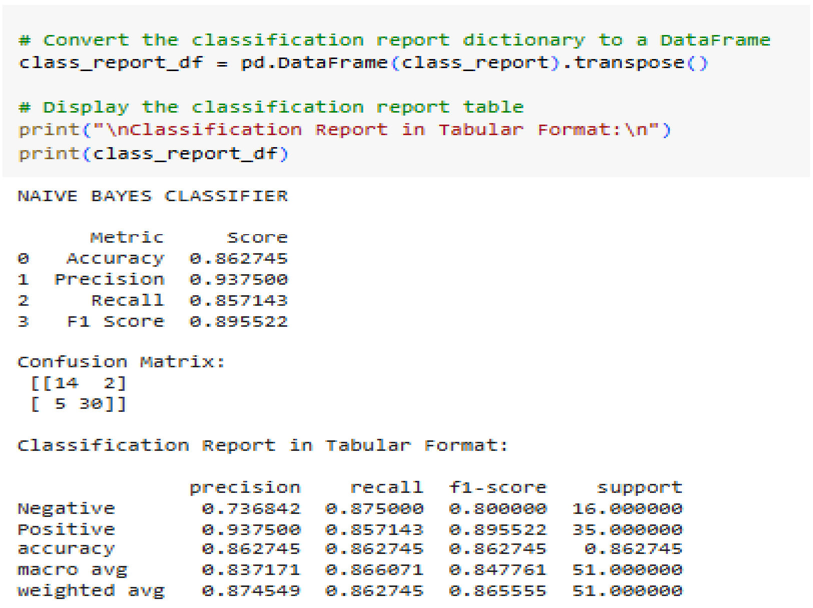
```
SUPPORT VECTOR MACHINE

      Metric     Score
0   Accuracy  0.921569
1  Precision  0.918919
2     Recall  0.971429
3   F1 Score  0.944444

Confusion Matrix:
 [[13  3]
 [ 1 34]]

Classification Report in Tabular Format:

              precision    recall  f1-score    support
Negative       0.928571  0.812500  0.866667  16.000000
Positive       0.918919  0.971429  0.944444  35.000000
accuracy       0.921569  0.921569  0.921569   0.921569
macro avg      0.923745  0.891964  0.905556  51.000000
weighted avg   0.921947  0.921569  0.920044  51.000000
```

Naive Bayes Classifier:

The Naive Bayes classifier is a probabilistic machine learning algorithm based on **Bayes' Theorem**, which assumes that features are independent of each other. Despite this simplifying assumption, it performs well in many real-world applications, especially for text classification and binary classification problems. It calculates the probability of each class given the input features and assigns the class with the highest probability. The Naive Bayes classifier is particularly effective in handling high-dimensional datasets and provides fast predictions.



```
model=GaussianNB()
print("NAIVE BAYES CLASSIFIER \n")

# Train the model
model.fit(X_train, y_train)

# Predict on test data
y_pred = model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

# Create a dictionary with the metrics
metrics_data = {
    "Metric": ["Accuracy", "Precision", "Recall", "F1 Score"],
    "Score": [accuracy, precision, recall, f1]
}

# Convert dictionary to DataFrame
metrics_df = pd.DataFrame(metrics_data)

# Display the table
print(metrics_df)

print("\nConfusion Matrix:\n", confusion_matrix(y_test, y_pred))
from sklearn.metrics import classification_report

# Generate classification report as a dictionary
class_report = classification_report(y_test, y_pred, target_names=target_encoder.classes_, output_dict=True)

# Convert the classification report dictionary to a DataFrame
class_report_df = pd.DataFrame(class_report).transpose()
```

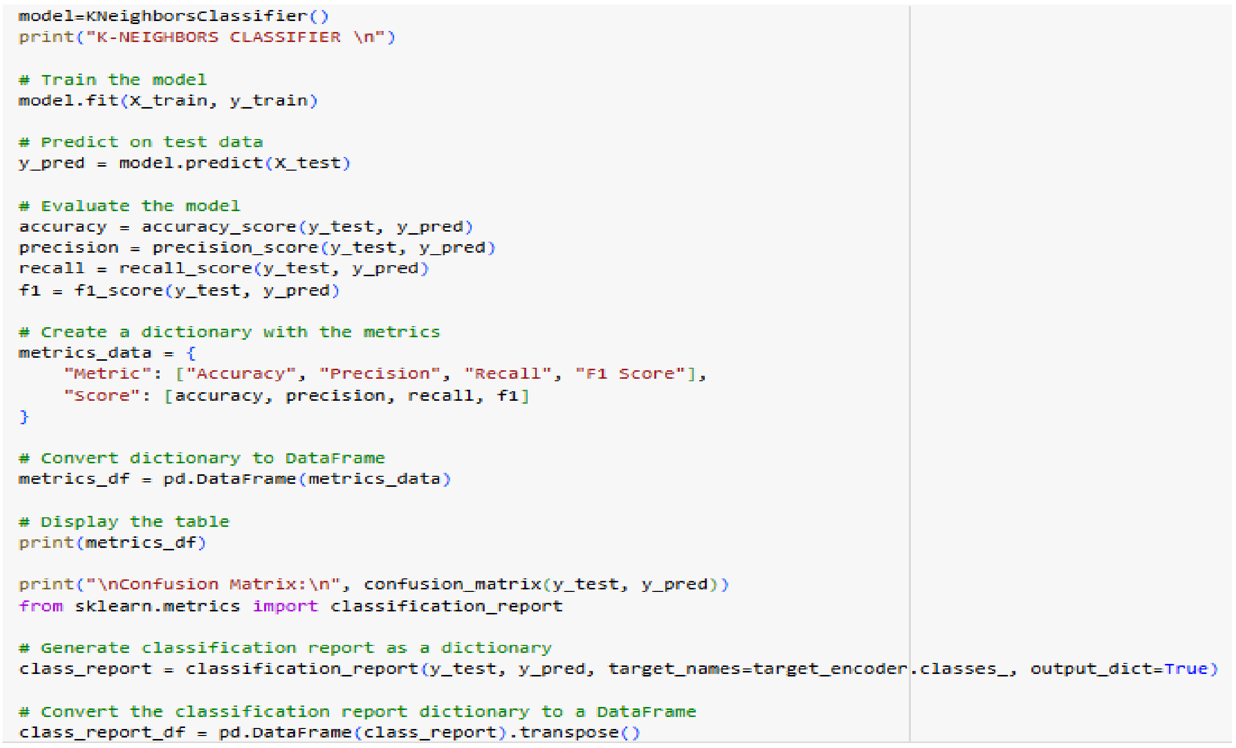


```
# Convert the classification report dictionary to a DataFrame
class_report_df = pd.DataFrame(class_report).transpose()

# Display the classification report table
print("\nClassification Report in Tabular Format:\n")
print(class_report_df)
```

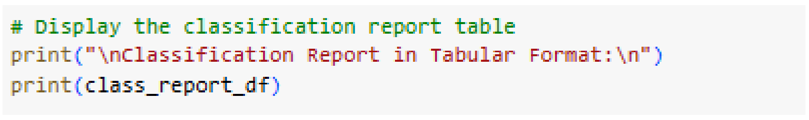
```
NAIVE BAYES CLASSIFIER

      Metric     Score
0   Accuracy  0.862745
1  Precision  0.937500
2     Recall  0.857143
3   F1 Score  0.895522

Confusion Matrix:
 [[14  2]
 [ 5 30]]

Classification Report in Tabular Format:

              precision    recall  f1-score    support
Negative       0.736842  0.875000  0.800000  16.000000
Positive       0.937500  0.857143  0.895522  35.000000
accuracy       0.862745  0.862745  0.862745   0.862745
macro avg      0.837171  0.866071  0.847761  51.000000
weighted avg   0.874549  0.862745  0.865555  51.000000
```

KNN Classifier:

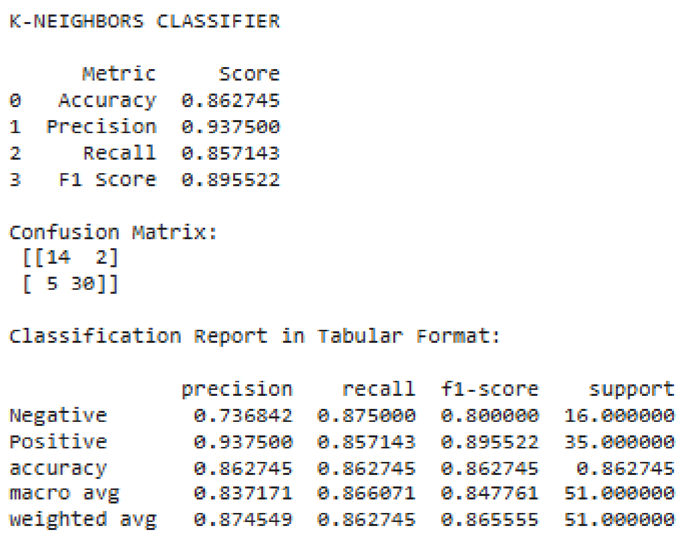
The **K-Nearest Neighbors (KNN) classifier** is a simple, instance-based learning algorithm that classifies a data point based on the majority class of its **K** nearest neighbors in the feature space. It does not require a training phase, as it makes decisions based on the distance metric (e.g., Euclidean distance) between the new point and the existing data. KNN is intuitive and effective, especially for smaller datasets, but can become computationally expensive with larger datasets.



```python
model=KNeighborsClassifier()
print("K-NEIGHBORS CLASSIFIER \n")

# Train the model
model.fit(X_train, y_train)

# Predict on test data
y_pred = model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

# Create a dictionary with the metrics
metrics_data = {
    "Metric": ["Accuracy", "Precision", "Recall", "F1 Score"],
    "Score": [accuracy, precision, recall, f1]
}

# Convert dictionary to DataFrame
metrics_df = pd.DataFrame(metrics_data)

# Display the table
print(metrics_df)

print("\nConfusion Matrix:\n", confusion_matrix(y_test, y_pred))
from sklearn.metrics import classification_report

# Generate classification report as a dictionary
class_report = classification_report(y_test, y_pred, target_names=target_encoder.classes_, output_dict=True)

# Convert the classification report dictionary to a DataFrame
class_report_df = pd.DataFrame(class_report).transpose()
```



```python
# Display the classification report table
print("\nClassification Report in Tabular Format:\n")
print(class_report_df)
```



```
K-NEIGHBORS CLASSIFIER

      Metric     Score
0   Accuracy  0.862745
1  Precision  0.937500
2     Recall  0.857143
3   F1 Score  0.895522

Confusion Matrix:
 [[14  2]
 [ 5 30]]

Classification Report in Tabular Format:

              precision    recall  f1-score    support
Negative       0.736842  0.875000  0.800000  16.000000
Positive       0.937500  0.857143  0.895522  35.000000
accuracy       0.862745  0.862745  0.862745   0.862745
macro avg      0.837171  0.866071  0.847761  51.000000
weighted avg   0.874549  0.862745  0.865555  51.000000
```

Evaluation of performance metrics

Accuracy:

The ratio of correctly predicted instances to the total instances, indicating overall correctness.

Precision:

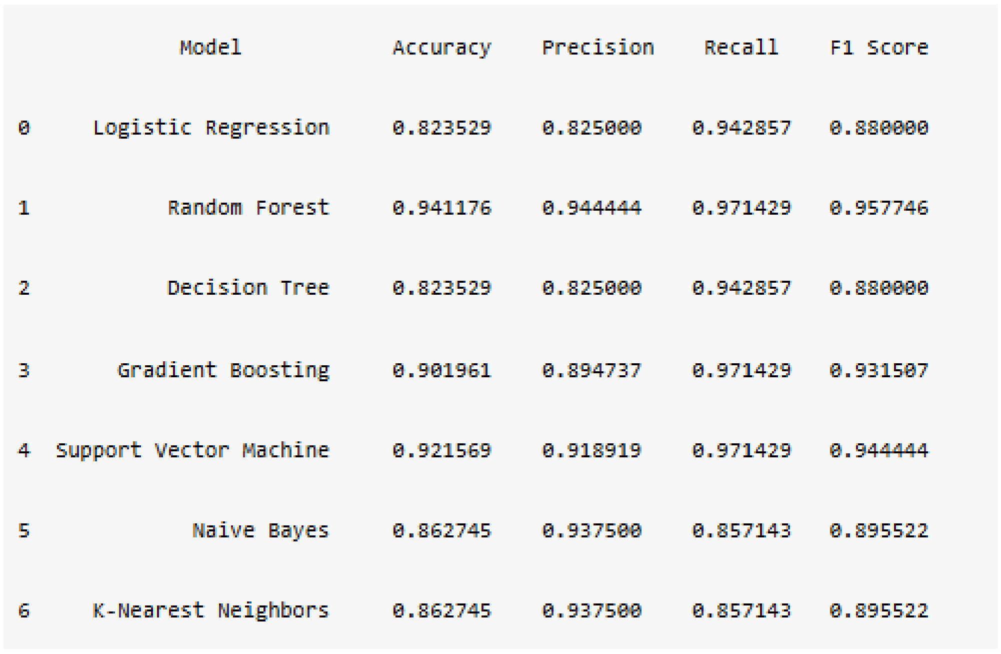
The ratio of true positive predictions to the total predicted positives, measuring the accuracy of positive predictions.

Recall (Sensitivity):

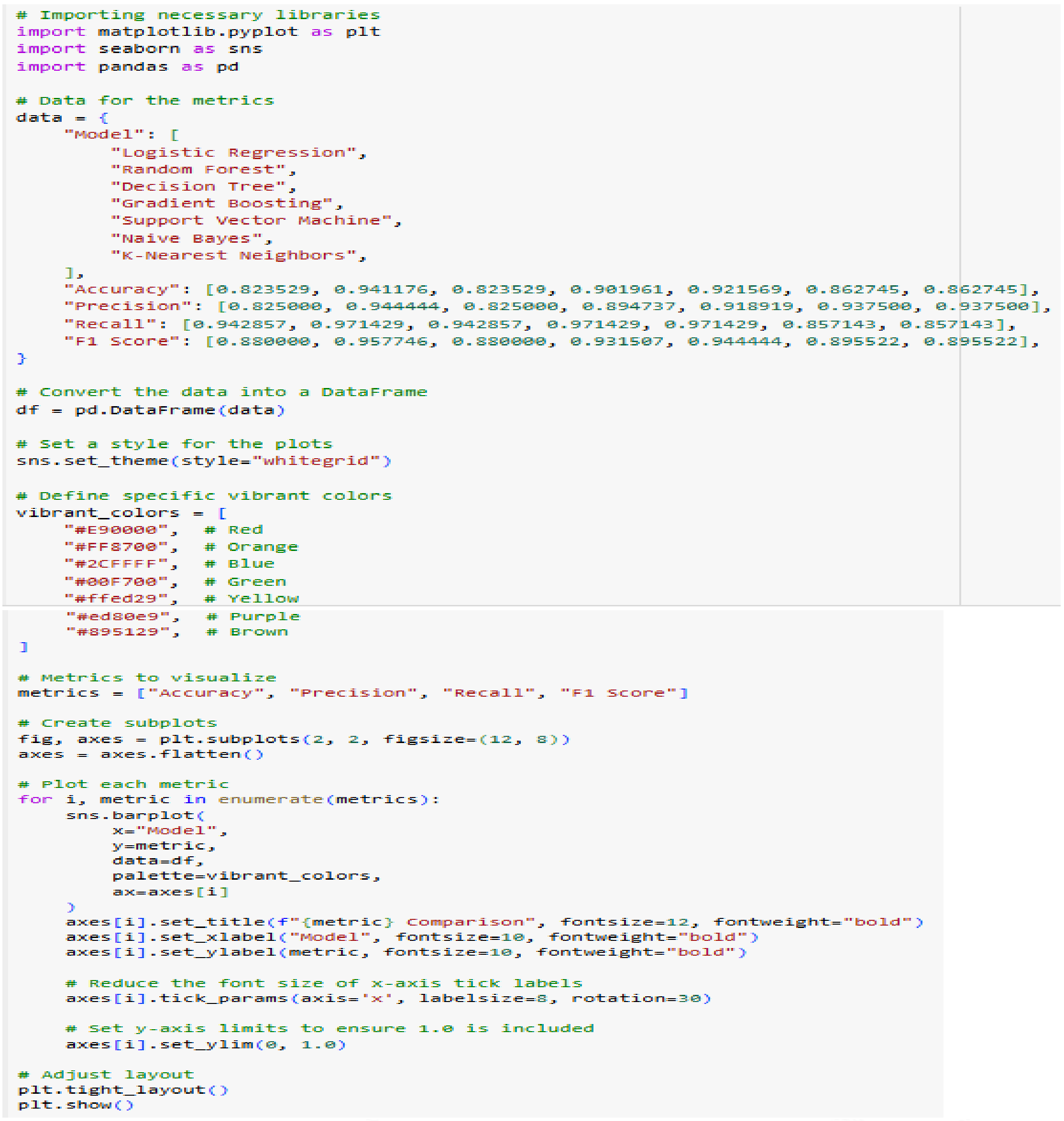
The ratio of true positive predictions to the total actual positives, indicating the model's ability to identify all relevant instances.

F1 Score:

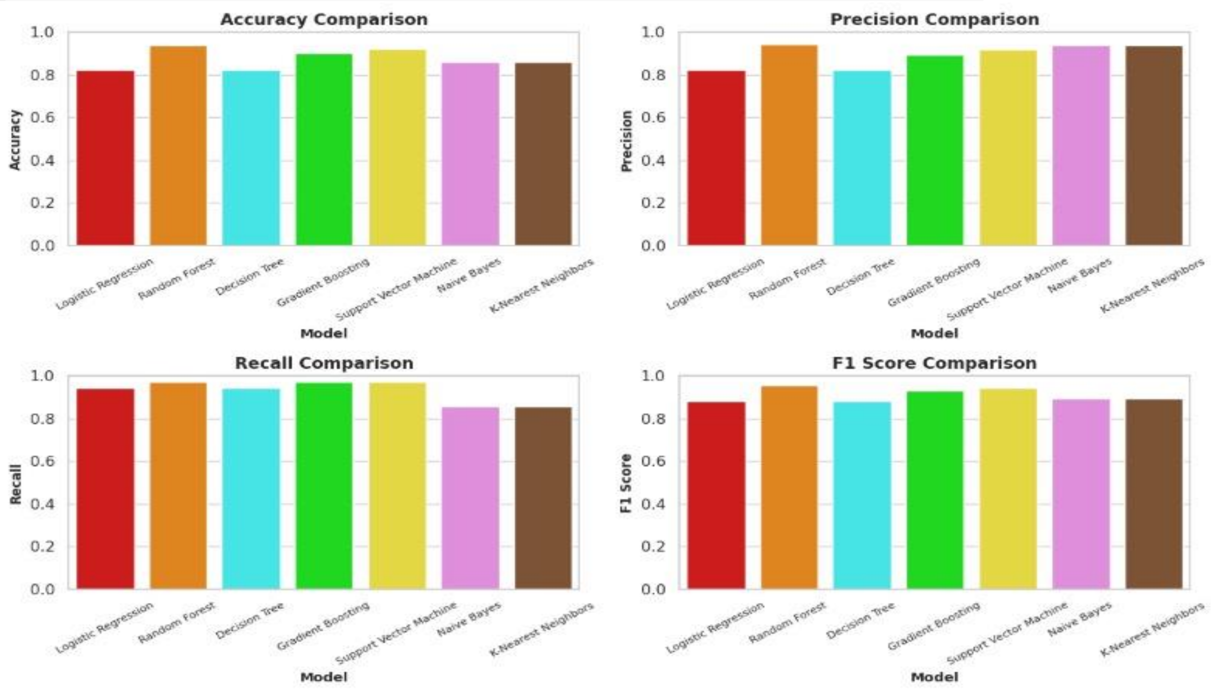
The harmonic mean of precision and recall, providing a balance between the two metrics, especially useful for imbalanced dataset.

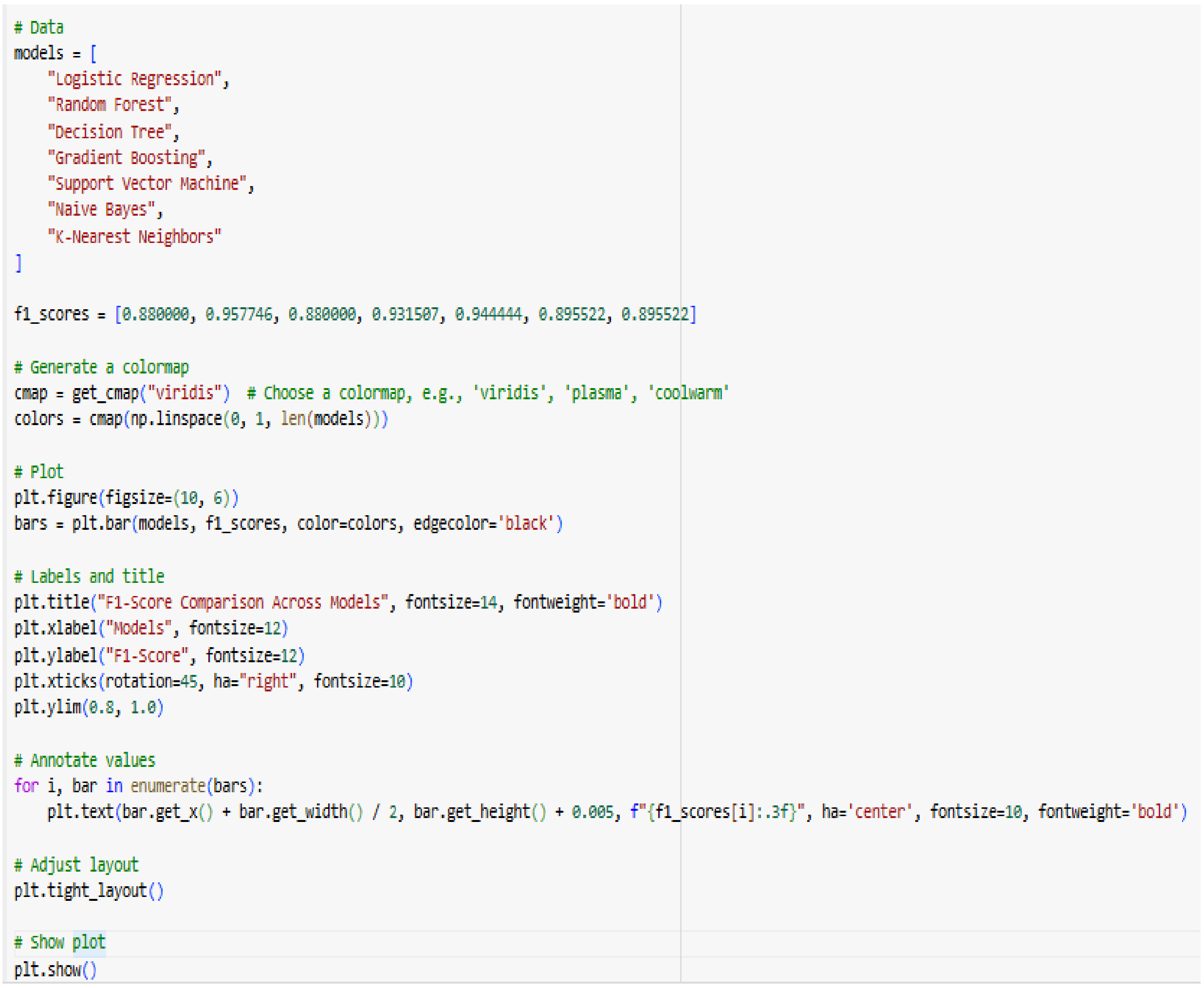


```
                    Model  Accuracy  Precision    Recall  F1 Score
0     Logistic Regression  0.823529   0.825000  0.942857  0.880000
1           Random Forest  0.941176   0.944444  0.971429  0.957746
2           Decision Tree  0.823529   0.825000  0.942857  0.880000
3       Gradient Boosting  0.901961   0.894737  0.971429  0.931507
4  Support Vector Machine  0.921569   0.918919  0.971429  0.944444
5             Naive Bayes  0.862745   0.937500  0.857143  0.895522
6     K-Nearest Neighbors  0.862745   0.937500  0.857143  0.895522
```

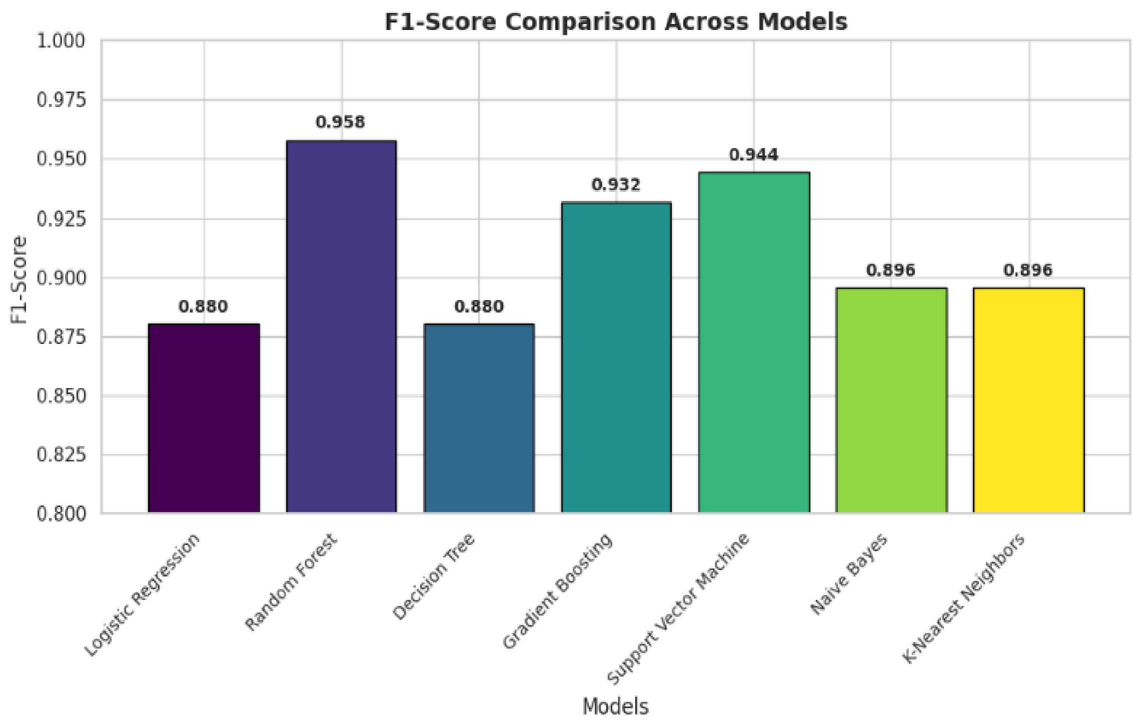


```
# Importing necessary libraries
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

# Data for the metrics
data = {
    "Model": [
        "Logistic Regression",
        "Random Forest",
        "Decision Tree",
        "Gradient Boosting",
        "Support Vector Machine",
        "Naive Bayes",
        "K-Nearest Neighbors",
    ],
    "Accuracy": [0.823529, 0.941176, 0.823529, 0.901961, 0.921569, 0.862745, 0.862745],
    "Precision": [0.825000, 0.944444, 0.825000, 0.894737, 0.918919, 0.937500, 0.937500],
    "Recall": [0.942857, 0.971429, 0.942857, 0.971429, 0.971429, 0.857143, 0.857143],
    "F1 Score": [0.880000, 0.957746, 0.880000, 0.931507, 0.944444, 0.895522, 0.895522],
}

# Convert the data into a DataFrame
df = pd.DataFrame(data)

# Set a style for the plots
sns.set_theme(style="whitegrid")

# Define specific vibrant colors
vibrant_colors = [
    "#E90000",  # Red
    "#FF8700",  # Orange
    "#2CFFFF",  # Blue
    "#00F700",  # Green
    "#ffed29",  # Yellow
    "#ed80e9",  # Purple
    "#895129",  # Brown
]

# Metrics to visualize
metrics = ["Accuracy", "Precision", "Recall", "F1 Score"]

# Create subplots
fig, axes = plt.subplots(2, 2, figsize=(12, 8))
axes = axes.flatten()

# Plot each metric
for i, metric in enumerate(metrics):
    sns.barplot(
        x="Model",
        y=metric,
        data=df,
        palette=vibrant_colors,
        ax=axes[i]
    )
    axes[i].set_title(f"{metric} Comparison", fontsize=12, fontweight="bold")
    axes[i].set_xlabel("Model", fontsize=10, fontweight="bold")
    axes[i].set_ylabel(metric, fontsize=10, fontweight="bold")

    # Reduce the font size of x-axis tick labels
    axes[i].tick_params(axis='x', labelsize=8, rotation=30)

    # Set y-axis limits to ensure 1.0 is included
    axes[i].set_ylim(0, 1.0)

# Adjust layout
plt.tight_layout()
plt.show()
```





```
# Data
models = [
    "Logistic Regression",
    "Random Forest",
    "Decision Tree",
    "Gradient Boosting",
    "Support Vector Machine",
    "Naive Bayes",
    "K-Nearest Neighbors"
]

f1_scores = [0.880000, 0.957746, 0.880000, 0.931507, 0.944444, 0.895522, 0.895522]

# Generate a colormap
cmap = get_cmap("viridis")  # Choose a colormap, e.g., 'viridis', 'plasma', 'coolwarm'
colors = cmap(np.linspace(0, 1, len(models)))

# Plot
plt.figure(figsize=(10, 6))
bars = plt.bar(models, f1_scores, color=colors, edgecolor='black')

# Labels and title
plt.title("F1-Score Comparison Across Models", fontsize=14, fontweight='bold')
plt.xlabel("Models", fontsize=12)
plt.ylabel("F1-Score", fontsize=12)
plt.xticks(rotation=45, ha="right", fontsize=10)
plt.ylim(0.8, 1.0)

# Annotate values
for i, bar in enumerate(bars):
    plt.text(bar.get_x() + bar.get_width() / 2, bar.get_height() + 0.005, f"{f1_scores[i]:.3f}", ha='center', fontsize=10, fontweight='bold')

# Adjust layout
plt.tight_layout()

# Show plot
plt.show()
```



Final Metric Chosen: F1-Score

Why F1-Score?

Balanced Performance:

Diabetes detection requires a balance between precision (avoiding unnecessary false alarms) and recall (ensuring no true cases are missed). F1-score provides this balance. It accounts for both false positives and false negatives, making it ideal when both errors have consequences.

Imbalanced Dataset Handling:

If the dataset has an unequal distribution of positive and negative cases, accuracy might be misleading. F1-score focuses on the minority class, which is critical for medical diagnosis.

Interpretability:

F1-score ensures no single metric like precision or recall dominates the evaluation, leading to fairer model selection.

Why Not Accuracy or Other Metrics?

Accuracy:

Misleading in cases of class imbalance. For example, if 80% of cases are negative, predicting "Negative" for all samples will yield 80% accuracy but zero value in detecting true positives.

Precision:

While precision is important, focusing solely on it could ignore recall, potentially missing true positive cases (false negatives) critical in diabetes detection.

Recall:

While recall ensures all positive cases are detected, it may lead to excessive false positives without balancing precision, resulting in unnecessary interventions.

Results and Findings

Model Selection:

Final Model Chosen: **Random Forest Classifier**

Why Random Forest?

Performance:

1. Random Forest achieved a high F1-score compared to other models.

2. It performs well on tabular data with non-linear relationships and categorical features like those in the diabetes dataset.

Feature Importance:

1. Provides interpretability by ranking the importance of input features, aiding in understanding which symptoms contribute most to the classification.

Stability:

1. It is less prone to overfitting than Decision Trees due to its ensemble approach.

2. Performs robustly without requiring extensive hyperparameter tuning.

Comparison with Other Models:

Gradient Boosting: Often marginally better than Random Forest in specific scenarios but typically requires more tuning and is computationally intensive.

Logistic Regression: Suitable for linear problems but lacks the ability to capture non-linear interactions present in complex datasets.

SVM: Effective in high-dimensional spaces but may not scale well to larger datasets or handle categorical features easily.

Naive Bayes: Assumes feature independence, which might not hold in this dataset.

K-Nearest Neighbors: Sensitive to scaling and less interpretable for medical use cases.

Challenges and limitations encountered:

During the project, certain challenges and limitations were encountered, such as limited data, data redundancy limited availability of certain variables, or the presence of outliers. These challenges were discussed, and their potential impact on the analysis and results were acknowledged.

Recommendations and Future Scope

Recommendations:

- Adopt Random Forest Model: With the highest F1-score, the Random Forest model is the most suitable for predicting diabetes in this dataset. Its ability to handle imbalanced data and provide robust predictions makes it ideal for deployment.

- Future Improvements: Collect more diverse and balanced data. Include additional features such as lifestyle habits, diet specifics, and genetic predispositions for better predictions. Explore deep learning for further accuracy.

- Deployment Recommendation s: Integrate the model into healthcare systems for screening. Develop a mobile or desktop application for usability.

Recommendations for Early Intervention :

- Early Detection and Treatment: GlucoSense can aid in early detection, enabling timely interventions and preventing complications.

- Lifestyle Modifications: The model's insights can guide individuals in adopting healthier habits, like diet and exercise, to manage blood glucose levels.

- Personalized Care Plans: The system can generate personalized care plans based on individual risk factors, facilitating proactive management.

- Regular Monitoring: GlucoSense encourages frequent monitoring of blood glucose levels, allowing for timely adjustments to treatment plans.

Conclusion

Summary of Achievements:

This project successfully demonstrated the potential of machine learning techniques in early detection and diagnosis of diabetes. By utilizing a well-structured dataset and implementing robust data preprocessing methods, key insights into the factors influencing diabetes onset were uncovered.

The feature selection process, particularly through mutual information, highlighted key features that are crucial for accurate diabetes prediction, such as polyuria and polydipsia. This feature importance analysis further refined the models, ensuring that only the most relevant information was used in the prediction process. The KNN model, in particular, provided strong performance, reinforcing its suitability for this kind of classification problem.

However, while the project demonstrates a significant step forward in AI-powered healthcare, there are opportunities for improvement. Future work could involve the integration of additional data sources, hyper parameter tuning, and the exploration of more advanced models, such as deep learning, to improve prediction accuracy. The ultimate goal is to develop a system that can be deployed in real-world healthcare settings, assisting doctors and medical professionals in providing timely interventions for diabetes management.

In conclusion, the project highlights the transformative potential of AI in healthcare, especially in the early detection of chronic conditions like diabetes. Through continuous improvements and updates, the GlucoSense system could become a valuable tool in healthcare diagnostics, enhancing patient outcomes through proactive care.

References

- https://www.kaggle.com/datasets/andrewmvd/early-diabetes-classification
- https://matplotlib.org/
- https://www.geeksforgeeks.org/what-is-exploratory-data-analysis/
- https://www.javatpoint.com/performance-metrics-in-machine-learning
- https://www.scholarhat.com/tutorial/machinelearning/model-selection-for-machine-learning