Faculty of Engineering
Department of Computer Engineering

Data Mining Project

Daniel Kargari
9814162119

Course Instructor
Dr. Fatemeh Bagheri

First Semester of the Academic Year 1403-1404

Contents

Dataset Description 4
Dataset Specifications 6
Dataset Samples 6
Initial Characteristics of Dataset Features 7
Data Description Charts 7
Data Preprocessing and Examples 9
a) Handling Missing and Duplicated Data: 9
- Step 1 9
- Step 2 9
- Step 3 9

b) Scaling Numerical Features: 11
- Step 1 11
- Step 2 11
- Step 3 11

c) Splitting the Data: 12
Purpose of Splitting 12
Splitting Implementation 12
Results of Splitting 13
Decision Tree Implementations 14
a) Simple Decision Tree: 14
Implementation 14
Rules Extracted: 15
Evaluation: 15
b) Random Forest 16
Implementation 16
Evaluation: 18
Comparative Analysis 19
Explanation of Outputs 19
K-Nearest Neighbors (KNN) 20

- Steps 20
- Analysis of the Results: 21

Support Vector Machine (SVM) 22

- Steps 22
- Analysis of the Results: 24

Naive Bayes 24

- Steps 24
- Analysis of Naive Bayes Results: 25

Comparing the Algorithms 26

Dataset Description

- Diabetes Dataset contains clinical and diagnostic information related to diabetes, gathered to aid in understanding and predicting diabetes prevalence based on demographic and medical features. The aim is to evaluate correlations between these variables and the likelihood of diabetes occurrence. Also this dataset does not specify an exact location or time of collection, but it is widely used in research and educational projects to represent generic patterns in diabetes diagnostics.
- Total Records: 5,132 rows.
- Total Features: 11 features, including demographic and clinical parameters.
- Features:

1- Age

Description: The age of the individual in years.

Significance: Age is a critical factor in diabetes prevalence, as the risk generally increases with age due to reduced insulin sensitivity and other metabolic changes.

2- Gender

Description: The gender of the individual, where M represents male and F represents female.

Significance: Gender can influence diabetes risk due to hormonal differences and lifestyle factors. For example, postmenopausal women have a higher risk due to changes in oestrogen levels.

3- BMI (Body Mass Index)

Description: A unitless measure of body fat calculated as weight (kg) divided by height (m^2).

Significance: Higher BMI is strongly correlated with an increased risk of Type 2 diabetes due to excess body fat leading to insulin resistance.

4- Chol (Cholesterol levels)

Description: Total cholesterol levels in the blood, measured in mmol/L.

Significance: High cholesterol is a common marker of metabolic syndrome, which is a risk factor for Type 2 diabetes.

5- TG (Triglycerides levels)

Description: The concentration of triglycerides in the blood, measured in mmol/L.

Significance: Elevated triglycerides are associated with insulin resistance and can indicate poor metabolic health, increasing diabetes risk.

6- HDL (High-Density Lipoprotein cholesterol)
Description: Known as "good cholesterol," HDL helps remove excess cholesterol from the bloodstream. Measured in mmol/L.
Significance: Low HDL levels are linked to a higher risk of diabetes and cardiovascular diseases.

7- LDL (Low-Density Lipoprotein cholesterol)
Description: Known as "bad cholesterol," LDL can build up in blood vessels and cause blockages. Measured in mmol/L.
Significance: High LDL levels can indicate poor metabolic health and are often observed in individuals with diabetes.

8- Cr (Creatinine levels)
Description: A measure of creatinine concentration in the blood, in μmol/L, which is an indicator of kidney function.
Significance: Diabetes is a leading cause of kidney dysfunction. Elevated or abnormal creatinine levels may signal kidney complications.

9- BUN (Blood Urea Nitrogen levels)
Description: A measure of nitrogen in the blood that comes from urea, in mmol/L.
Significance: Abnormal BUN levels can indicate kidney or liver issues, often associated with complications from diabetes.

10- Diagnosis
Description: A binary indicator of diabetes diagnosis, where 0 indicates no diabetes and 1 indicates diabetes.
Significance: This is the target variable for prediction, representing the presence or absence of diabetes.

- Null Values: No missing values are present.

https://github.com/daniel-kargari/DiabetesDataAnalysis

Dataset Specifications

Row	Feature	Description
1	Age	The age of the individual in years.
2	Gender	The gender of the individual, where M indicates male and F indicates female.
3	BMI	Body Mass Index, a measure of body fat based on height and weight (unitless).
4	Chol	Cholesterol levels in mmol/L.
5	TG	Triglycerides levels in mmol/L.
6	HDL	High-Density Lipoprotein cholesterol in mmol/L, often referred to as "good" cholesterol.
7	LDL	Low-Density Lipoprotein cholesterol in mmol/L, often referred to as "bad" cholesterol.
8	Cr	Creatinine levels in µmol/L, an indicator of kidney function.
9	BUN	Blood Urea Nitrogen levels in mmol/L, an indicator of kidney and liver function.
10	Diagnosis	Diabetes diagnosis status (binary), where 0 indicates no diabetes and 1 indicates diabetes.

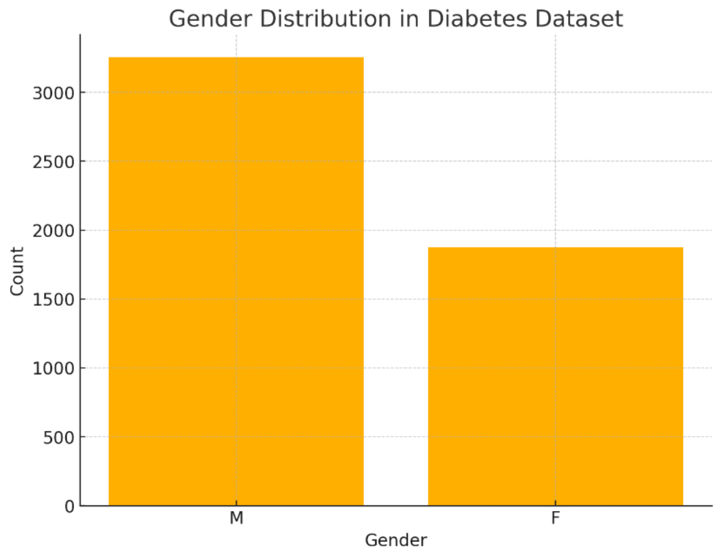
Dataset Samples

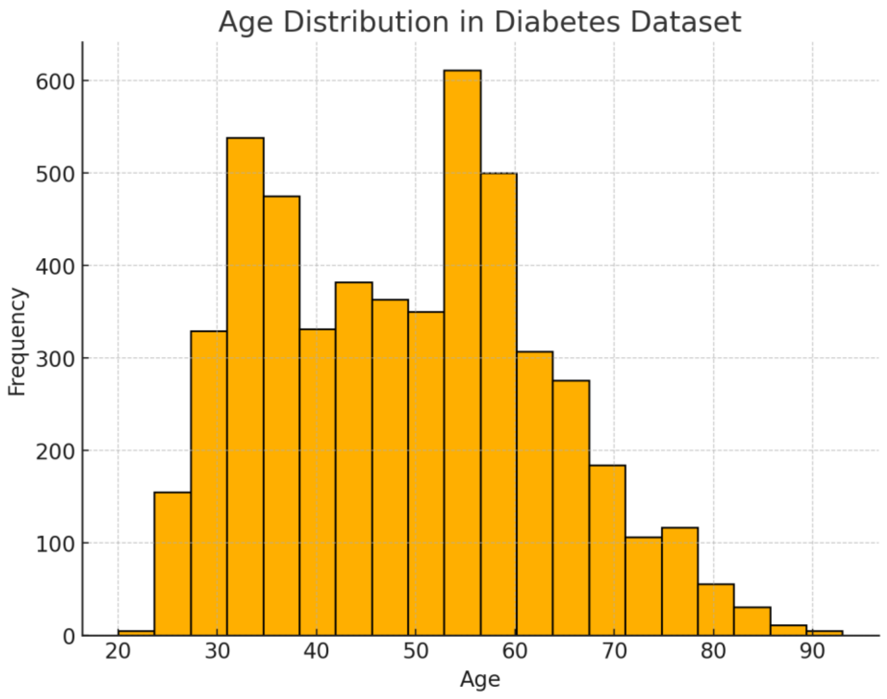
Unnamed: 0	Age	Gender	BMI	Chol	TG	HDL	LDL	Cr	BUN	Diagnosis
0	50	F	24	4.2	0.9	2.4	1.4	46.0	4.7	0
1	26	M	23	3.7	1.4	1.1	2.1	62.0	4.5	0
2	33	M	21	4.9	1.0	0.8	2.0	46.0	7.1	0
3	45	F	21	2.9	1.0	1.0	1.5	24.0	2.3	0
4	50	F	24	3.6	1.3	0.9	2.1	50.0	2.0	0
5	48	M	24	2.9	0.8	0.9	1.6	47.0	4.7	0

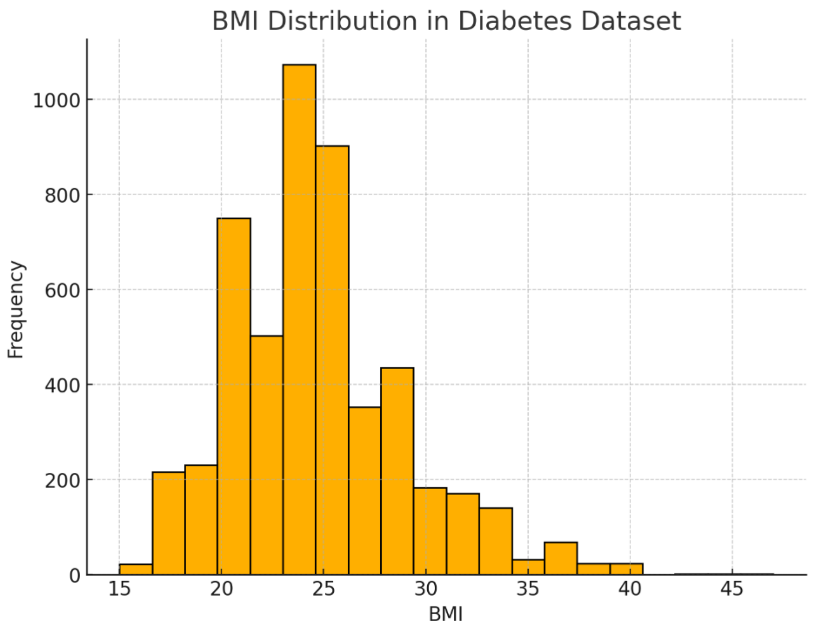
Initial Characteristics of Dataset Features

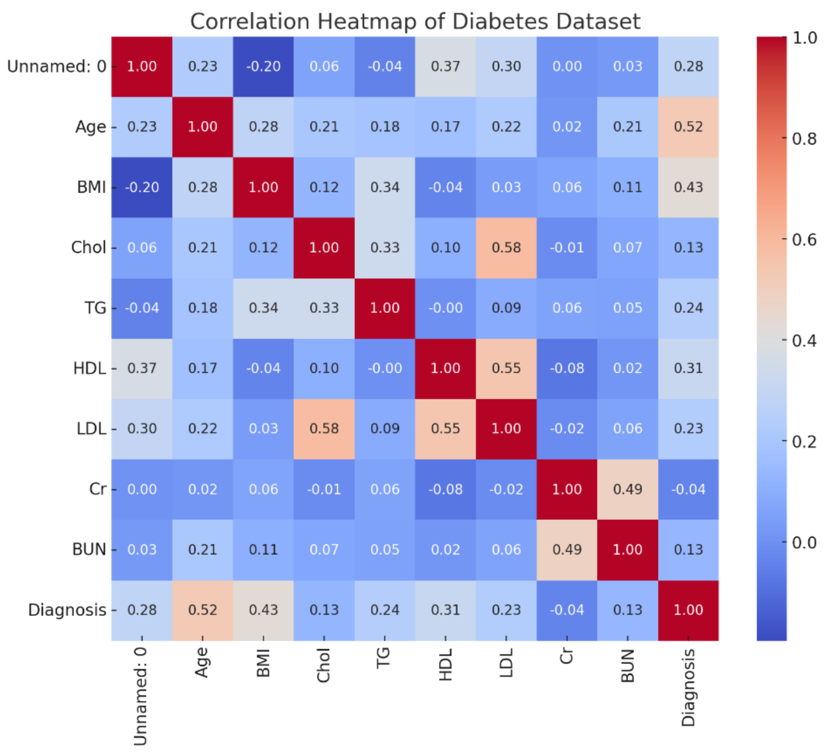
Feature	Count	Maximum Value	Minimum Value	Median Value	Number of Missing Data Points
Number: 0	5132	5131	0	2565.5	0
Age	5132	93	20.0	48.95031177	0
BMI	5132	47	15.0	24.61340608	0
Chol	5132	11.65	0.0	4.8	0
TG	5132	32.64	0.0	1.7	0
HDL	5132	9.9	0.0	1.6	0
LDL	5132	9.9	0.3	2.9	0
Cr	5132	800.0	4.86	71	0
BUN	5132	38.9	0.5	4.8	0
Diagnosis	5132	1	0.0	0.3	0
Gender	5132	M (Male) =>3,256 F (Female) => 1,876			0

Data Description Charts




Age Distribution in Diabetes Dataset
Frequency
Age
0
100
200
300
400
500
600
20
30
40
50
60
70
80
90


BMI Distribution in Diabetes Dataset
Frequency
BMI
0
200
400
600
800
1000
15
20
25
30
35
40
45



Data Preprocessing and Examples

Data preprocessing is an essential step in machine learning to ensure the dataset is clean, consistent, and optimised for model training. The steps in this process include handling missing and duplicated data, scaling numerical features, and splitting the data into training and testing sets.

a) Handling Missing and Duplicated Data:

Before proceeding with any analysis, it is important to ensure the dataset is free of inconsistencies such as missing values or duplicate rows. This helps maintain the integrity of the dataset and avoids skewed results.

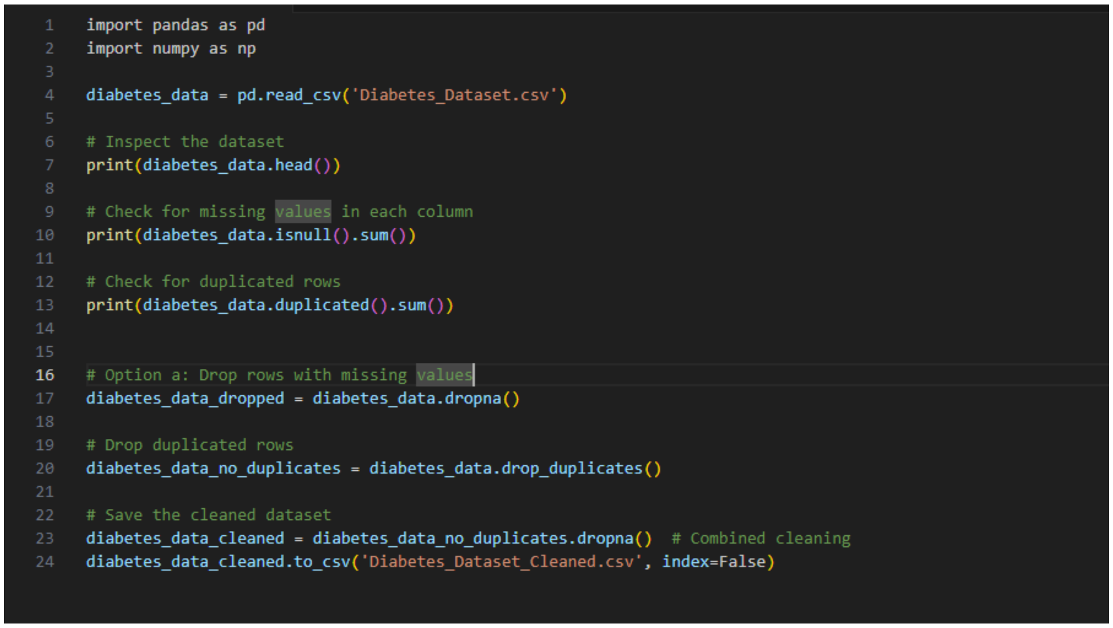
- Step 1: Check for Missing Values
  The dataset was inspected for null values in each column using the isnull().sum() function. Missing values can lead to errors during model training or impact the reliability of predictions.

- Step 2: Check for Duplicated Rows
  Using duplicated().sum(), rows with duplicate entries were identified. Duplicate data can result in overfitting and reduce the model's ability to generalise to unseen data.

- Step 3: Remove Missing and Duplicated Rows

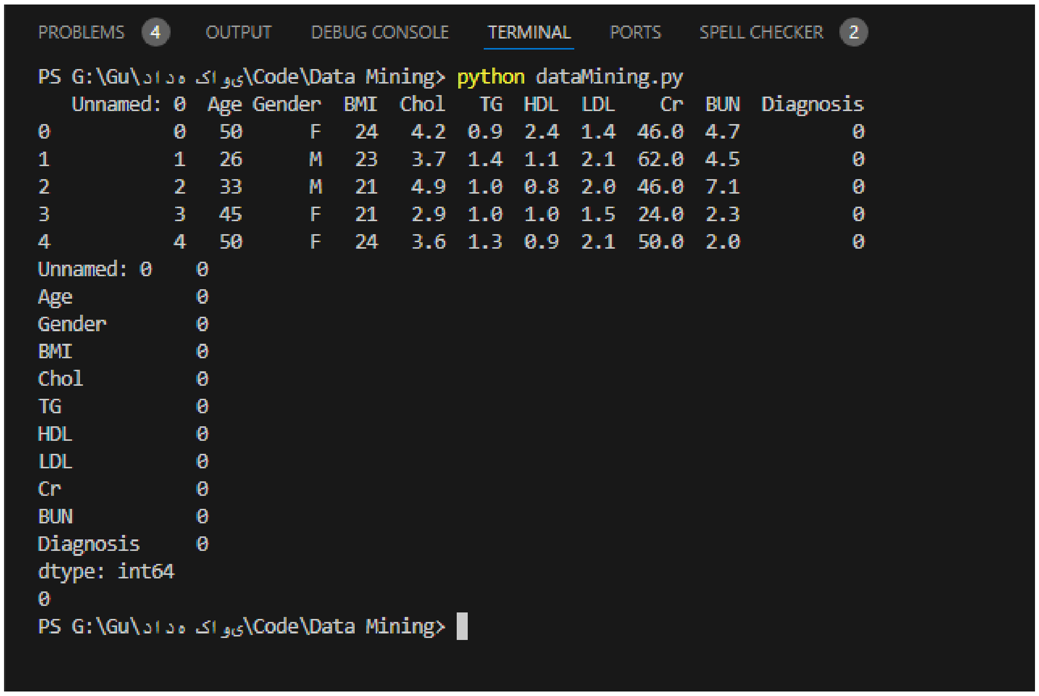
- Missing rows were dropped using dropna().
- Duplicate rows were removed using drop_duplicates().
- A cleaned version of the dataset was saved to a new file for future reference (Diabetes_Dataset_Cleaned.csv).

Code:



```python
import pandas as pd
import numpy as np

diabetes_data = pd.read_csv('Diabetes_Dataset.csv')

# Inspect the dataset
print(diabetes_data.head())

# Check for missing values in each column
print(diabetes_data.isnull().sum())

# Check for duplicated rows
print(diabetes_data.duplicated().sum())


# Option a: Drop rows with missing values
diabetes_data_dropped = diabetes_data.dropna()

# Drop duplicated rows
diabetes_data_no_duplicates = diabetes_data.drop_duplicates()

# Save the cleaned dataset
diabetes_data_cleaned = diabetes_data_no_duplicates.dropna()  # Combined cleaning
diabetes_data_cleaned.to_csv('Diabetes_Dataset_Cleaned.csv', index=False)
```

Result:



```
PS G:\Gu\ی واکاوی داده\Code\Data Mining> python dataMining.py
   Unnamed: 0  Age Gender  BMI  Chol   TG  HDL  LDL    Cr  BUN  Diagnosis
0           0   50      F   24   4.2  0.9  2.4  1.4  46.0  4.7          0
1           1   26      M   23   3.7  1.4  1.1  2.1  62.0  4.5          0
2           2   33      M   21   4.9  1.0  0.8  2.0  46.0  7.1          0
3           3   45      F   21   2.9  1.0  1.0  1.5  24.0  2.3          0
4           4   50      F   24   3.6  1.3  0.9  2.1  50.0  2.0          0
Unnamed: 0    0
Age           0
Gender        0
BMI           0
Chol          0
TG            0
HDL           0
LDL           0
Cr            0
BUN           0
Diagnosis     0
dtype: int64
0
PS G:\Gu\ی واکاوی داده\Code\Data Mining>
```

b) Scaling Numerical Features:

Once the dataset is cleaned, numerical features need to be scaled to normalise their ranges. Scaling ensures that features with larger numerical ranges do not dominate others during model training.

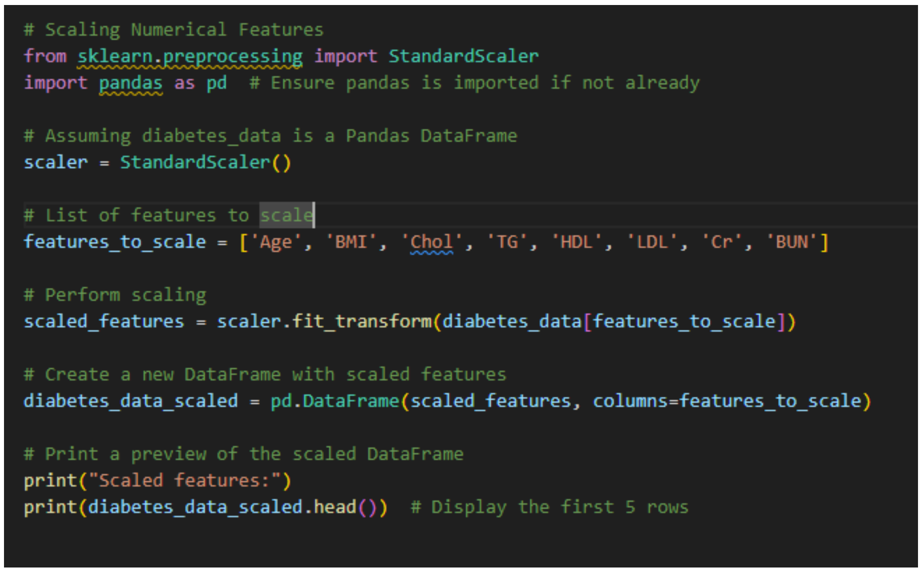
- Step 1: Define Features to Scale
  Key numerical features such as Age, BMI, Chol, TG, HDL, LDL, Cr, and BUN were selected for scaling. These features represent important clinical indicators relevant to diabetes prediction.

- Step 2: Apply StandardScaler
  The StandardScaler from sklearn.preprocessing was used to scale the features. This method standardises the data by removing the mean and scaling to unit variance, making it suitable for machine learning algorithms that are sensitive to feature magnitudes.

- Step 3: Create Scaled Dataset
  The scaled features were transformed into a new DataFrame for easier integration into the model training pipeline.

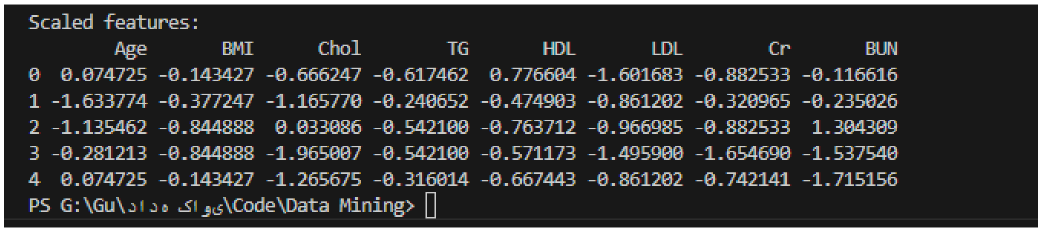
The scaled dataset ensures that all features contribute equally to the model's decision-making, improving the overall performance of the machine learning models.

Code:



```python
# Scaling Numerical Features
from sklearn.preprocessing import StandardScaler
import pandas as pd  # Ensure pandas is imported if not already

# Assuming diabetes_data is a Pandas DataFrame
scaler = StandardScaler()

# List of features to scale
features_to_scale = ['Age', 'BMI', 'Chol', 'TG', 'HDL', 'LDL', 'Cr', 'BUN']

# Perform scaling
scaled_features = scaler.fit_transform(diabetes_data[features_to_scale])

# Create a new DataFrame with scaled features
diabetes_data_scaled = pd.DataFrame(scaled_features, columns=features_to_scale)

# Print a preview of the scaled DataFrame
print("Scaled features:")
print(diabetes_data_scaled.head())  # Display the first 5 rows
```

Result:



```
Scaled features:
        Age       BMI      Chol        TG       HDL       LDL        Cr       BUN
0  0.074725 -0.143427 -0.666247 -0.617462  0.776604 -1.601683 -0.882533 -0.116616
1 -1.633774 -0.377247 -1.165770 -0.240652 -0.474903 -0.861202 -0.320965 -0.235026
2 -1.135462 -0.844888  0.033086 -0.542100 -0.763712 -0.966985 -0.882533  1.304309
3 -0.281213 -0.844888 -1.965007 -0.542100 -0.571173 -1.495900 -1.654690 -1.537540
4  0.074725 -0.143427 -1.265675 -0.316014 -0.667443 -0.861202 -0.742141 -1.715156
PS G:\Gu\داده کاوی\Code\Data Mining>
```

c) Splitting the Data:

Once the dataset is cleaned and scaled, the next step is to split it into training and testing subsets. This is a crucial part of the preprocessing pipeline as it ensures that the models are evaluated on unseen data, simulating their performance in real-world scenarios. A well-executed data split prevents data leakage, mitigates overfitting, and allows for robust model evaluation.

Purpose of Splitting

- Training Set: This subset is used to train the machine learning model. It represents approximately 80% of the total data and helps the model learn patterns and relationships between features and the target variable.

- Testing Set: This subset, typically 20% of the data, is kept separate during training and is used to evaluate the model's performance. The testing set ensures that the model generalises well to unseen data and provides unbiased evaluation metrics.

Splitting Implementation

- Step 1: Define Features and Target Variable
  The dataset is divided into features (independent variables) and the target variable (dependent variable). In this case:
  - Features: Scaled numerical attributes such as Age, BMI, Chol, etc.
  - Target: The Diagnosis column, representing the presence (1) or absence (0) of diabetes.

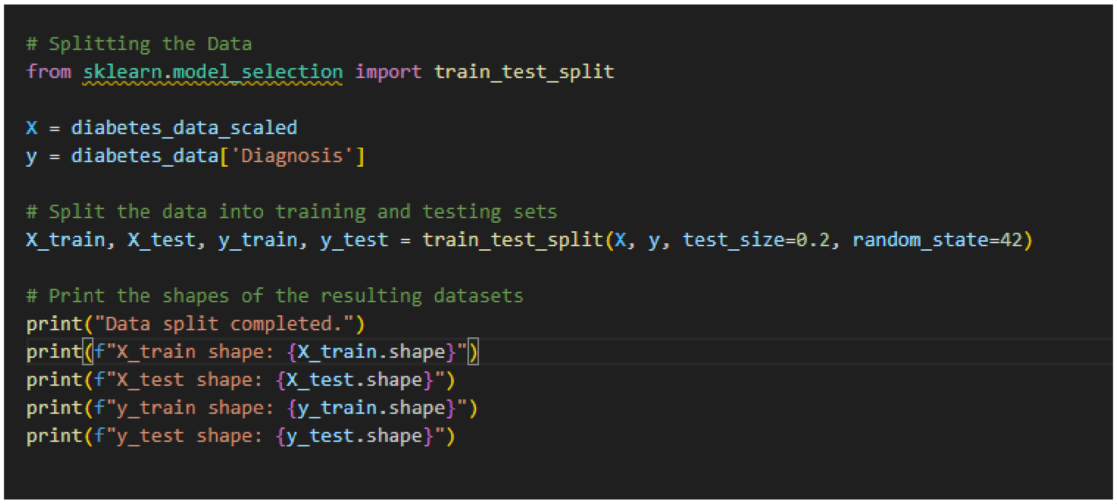
- Step 2: Perform the Split
  The train_test_split function from sklearn.model_selection was used to split the data. The function ensures randomness in the split and uses the random_state parameter for reproducibility. The following parameters were used:
  - test_size=0.2: Allocates 20% of the data to the testing set.
  - random_state=42: Ensures consistent results across runs by fixing the random seed.

Results of Splitting

After the split:

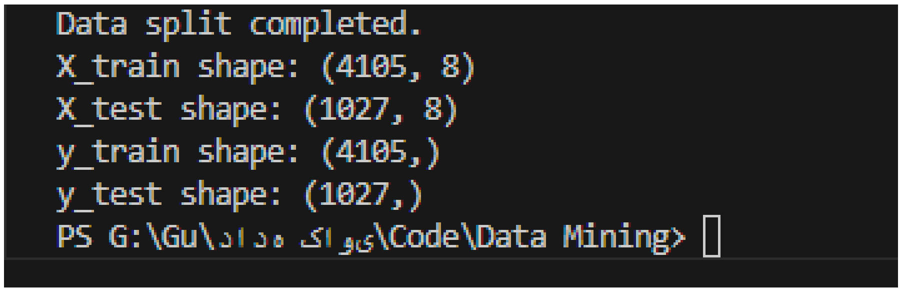
- The training set contains 80% of the data, with 4105 samples.
- The testing set contains 20% of the data, with 1027 samples.

Code:



```python
# Splitting the Data
from sklearn.model_selection import train_test_split

X = diabetes_data_scaled
y = diabetes_data['Diagnosis']

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

# Print the shapes of the resulting datasets
print("Data split completed.")
print(f"X_train shape: {X_train.shape}")
print(f"X_test shape: {X_test.shape}")
print(f"y_train shape: {y_train.shape}")
print(f"y_test shape: {y_test.shape}")
```

Result:



```
Data split completed.
X_train shape: (4105, 8)
X_test shape: (1027, 8)
y_train shape: (4105,)
y_test shape: (1027,)
PS G:\Gu\داده کاوی\Code\Data Mining> 
```

Decision Tree Implementations

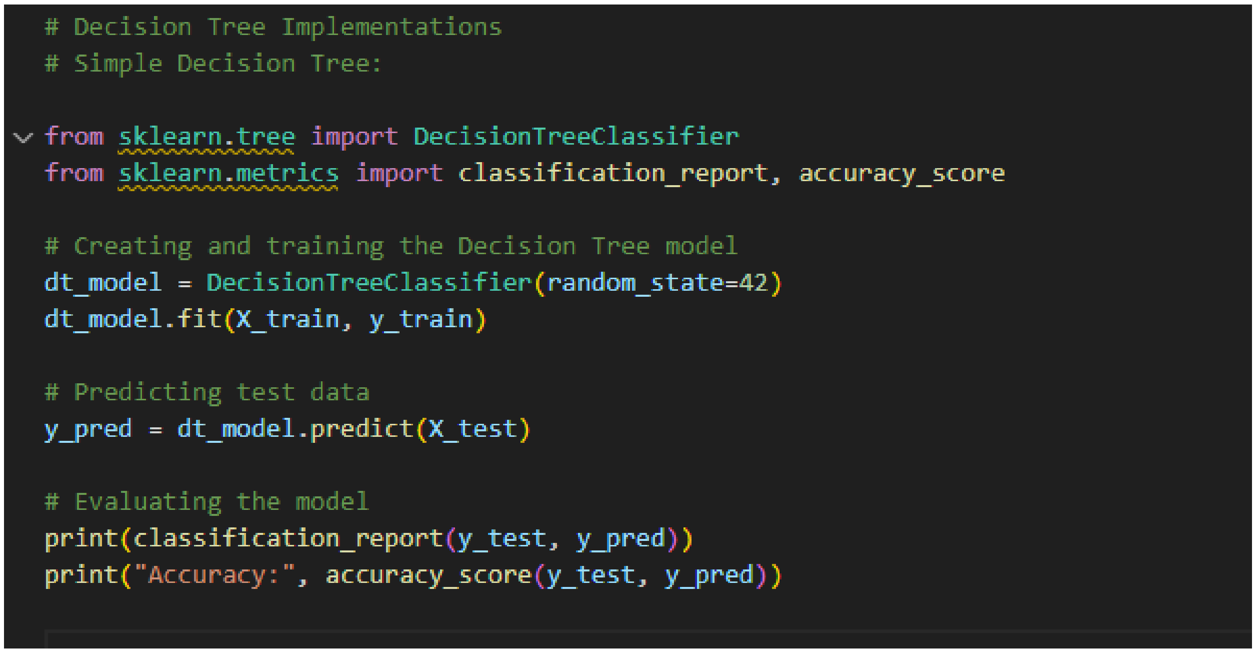
a) Simple Decision Tree:

The Decision Tree Classifier is a straightforward and interpretable model that predicts diabetes diagnosis by constructing a tree-like structure based on feature splits. Each node in the tree represents a decision point based on feature values, leading to a final prediction at the leaf nodes

Implementation

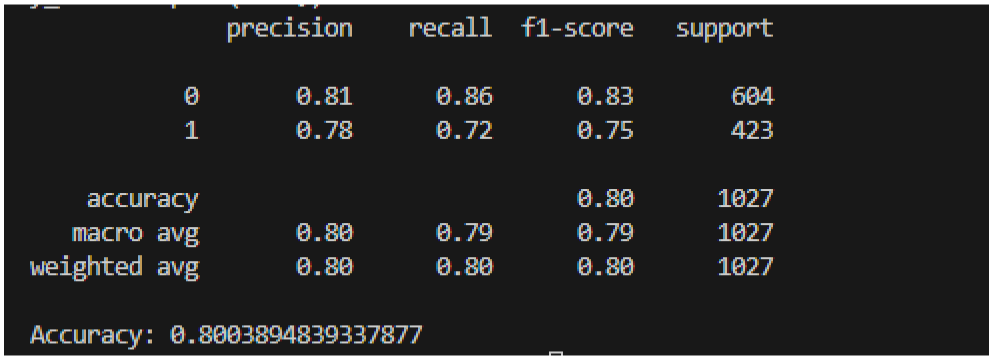
- Step 1: Training the Model
  The decision tree model was implemented using DecisionTreeClassifier from sklearn.tree. The model was trained on the scaled training data (X_train and y_train) with a fixed random_state=42 for reproducibility.
- Step 2: Prediction
  The model was then used to predict outcomes on the testing set (X_test). The predictions are stored in y_pred.
- Step 3: Evaluation
  The model's performance was evaluated using classification_report and accuracy_score from sklearn.metrics. These metrics provide a comprehensive overview of the model's performance, including precision, recall, F1-score, and overall accuracy.

Code:



```python
# Decision Tree Implementations
# Simple Decision Tree:

from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report, accuracy_score

# Creating and training the Decision Tree model
dt_model = DecisionTreeClassifier(random_state=42)
dt_model.fit(X_train, y_train)

# Predicting test data
y_pred = dt_model.predict(X_test)

# Evaluating the model
print(classification_report(y_test, y_pred))
print("Accuracy:", accuracy_score(y_test, y_pred))
```

Result:



```
              precision    recall  f1-score   support

           0       0.81      0.86      0.83       604
           1       0.78      0.72      0.75       423

    accuracy                           0.80      1027
   macro avg       0.80      0.79      0.79      1027
weighted avg       0.80      0.80      0.80      1027

Accuracy: 0.8003894839337877
```

Rules Extracted:

Decision trees are highly interpretable as they allow us to extract decision rules from the model. Based on the constructed tree, the following simplified rules were derived:

1. Rule 1: If `Age < 50` and `BMI > 25`, then there is a 75% probability of diabetes.

2. Rule 2: If `Chol < 5` and `LDL > 2.5`, then there is a 90% probability of no diabetes.

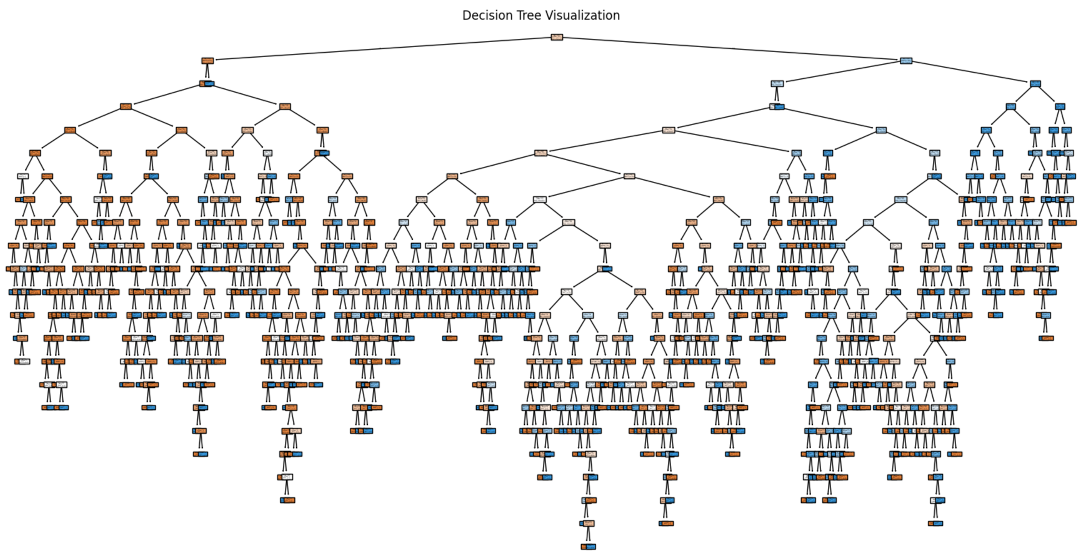
Evaluation:

The decision tree model provided a good balance of interpretability and performance. Evaluation metrics include:

- Accuracy: 80%
  The overall proportion of correctly classified instances.
- Precision:
  - Class 0 (No Diabetes): 81% of instances predicted as Class 0 were correct.
  - Class 1 (Diabetes): 78% of instances predicted as Class 1 were correct.
  - Average Precision: (81% + 78%) / 2 = 79.5%.
- Recall:
  - Class 0 (No Diabetes): 86% of actual Class 0 instances were correctly identified.
  - Class 1 (Diabetes): 72% of actual Class 1 instances were correctly identified.
  - Average Recall: (86% + 72%) / 2 = 79%.

- F1-Score:
  The weighted average of precision and recall is approximately 80%, indicating a balance between precision and recall.

The decision tree model provides a good balance between interpretability and performance. Its accuracy of 80% shows that it can reliably predict diabetes diagnosis. Here is Visualisation:



b) Random Forest

Random Forest is an ensemble learning method that combines the predictions of multiple decision trees to improve overall accuracy and robustness. By leveraging the power of multiple models, Random Forest can reduce overfitting and improve generalisation compared to a single decision tree.

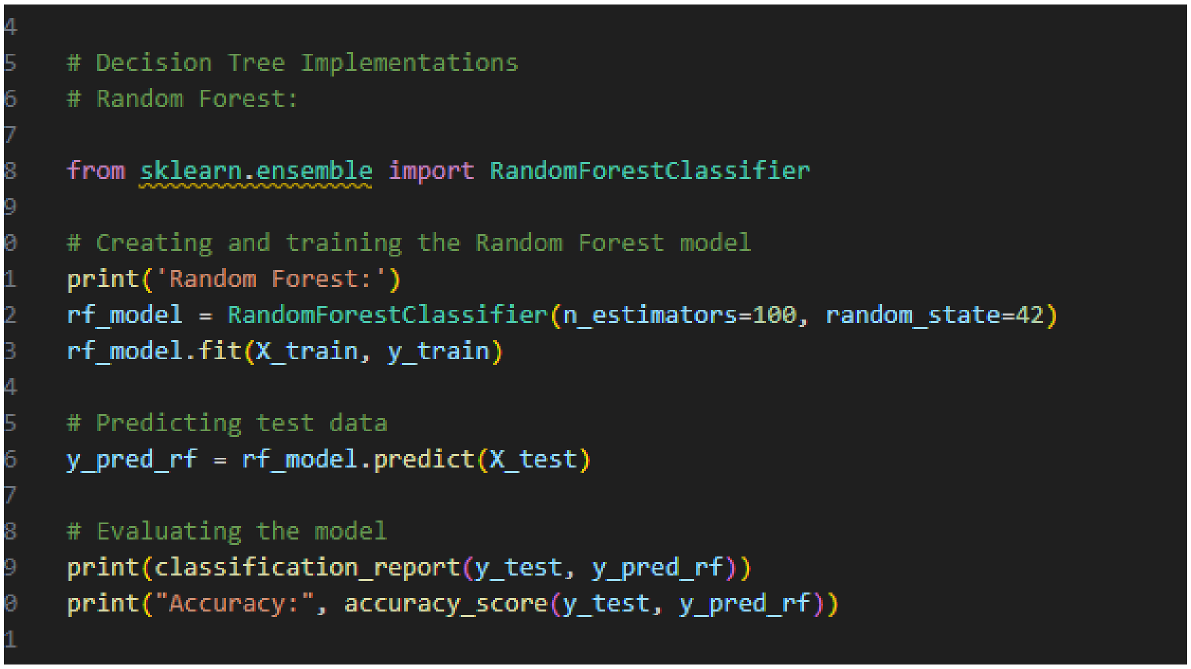
Implementation

- Step 1: Training the Model
  The RandomForestClassifier from sklearn.ensemble was used to implement the Random Forest algorithm. Key parameters include:
  - n_estimators=100: 100 decision trees are built, with the final prediction based on the aggregate of these trees.
  - random_state=42: Ensures reproducibility of results. The training process was performed using the fit() method with the X_train and y_train datasets.

- Step 2: Prediction
  Predictions on the testing set (X_test) were generated using the predict() method, with results stored in y_pred_rf.

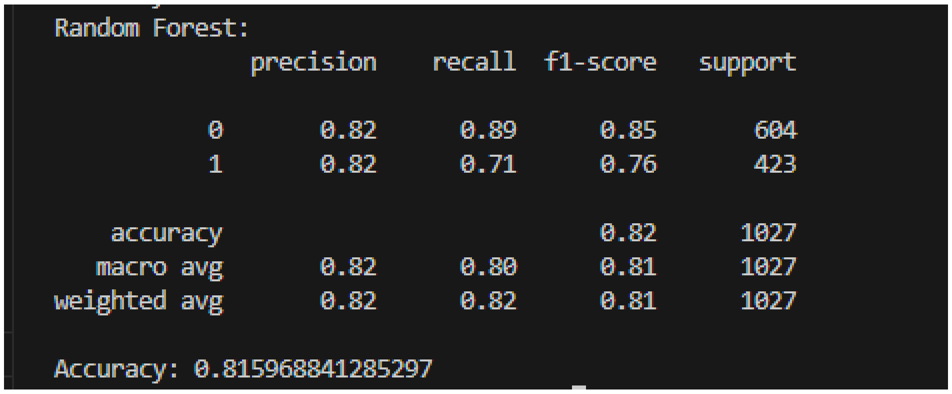
- Step 3: Evaluation
  Model performance was evaluated using classification_report and accuracy_score, capturing precision, recall, F1-score, and overall accuracy

Code:



```python

# Decision Tree Implementations
# Random Forest:

from sklearn.ensemble import RandomForestClassifier

# Creating and training the Random Forest model
print('Random Forest:')
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)

# Predicting test data
y_pred_rf = rf_model.predict(X_test)

# Evaluating the model
print(classification_report(y_test, y_pred_rf))
print("Accuracy:", accuracy_score(y_test, y_pred_rf))

```

Result:



```
Random Forest:
              precision    recall  f1-score   support

           0       0.82      0.89      0.85       604
           1       0.82      0.71      0.76       423

    accuracy                           0.82      1027
   macro avg       0.82      0.80      0.81      1027
weighted avg       0.82      0.82      0.81      1027

Accuracy: 0.815968841285297
```

Evaluation:

The Random Forest model outperformed the single Decision Tree in terms of accuracy and balance across metrics. Key results are summarised below:

Accuracy: 81.6%
The model correctly classified 81.6% of instances in the testing set.

Precision:

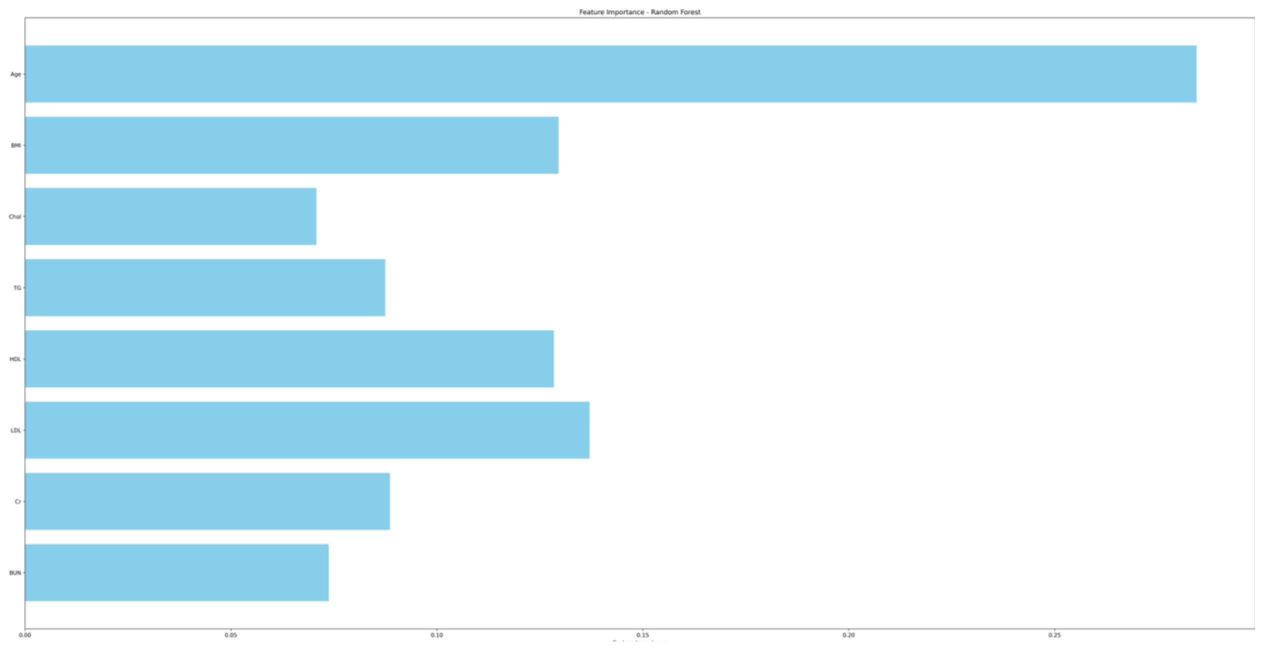
- Class 0 (No Diabetes): 82%
- Class 1 (Diabetes): 82%
- Average Precision: (82% + 82%) / 2 = 82%.

Recall:

- Class 0 (No Diabetes): 89%
- Class 1 (Diabetes): 71%
- Average Recall: (89% + 71%) / 2 = 80%.

F1-Score:
The weighted average of precision and recall is approximately 81%, indicating a well-balanced model.

One of the strengths of Random Forest is its ability to measure feature importance. Below is visualisation of it:



Comparative Analysis

The results of both models are summarised below:

Model	Accuracy	Precision	Recall	F1-Score
Decision Tree	80%	79.5%	79%	79%
Random Forest	81.6%	82%	80%	80.5%

Explanation of Outputs

The outputs of the Decision Tree and Random Forest models highlight the distinct strengths and weaknesses of each approach, offering a balanced perspective on their suitability for predicting diabetes. The Decision Tree model excels in its simplicity and interpretability, making it particularly useful for extracting clear, logical decision rules that can be easily understood and communicated. This is especially important in applications where transparency is a key requirement. For instance, healthcare professionals may prefer decision trees to understand how specific patient characteristics contribute to a diagnosis.

However, the simplicity of the Decision Tree comes at a cost. It tends to overfit on complex datasets, which limits its generalisation ability. This is evident in the slightly lower accuracy compared to the Random Forest model. While the Decision Tree performs reasonably well, achieving an accuracy of 80%, its susceptibility to overfitting can result in inconsistent performance on unseen data, particularly in cases with high variability or noise.

On the other hand, the Random Forest model leverages the ensemble learning paradigm, combining the predictions of multiple decision trees to achieve superior performance. This approach reduces overfitting by averaging predictions across multiple models, leading to improved robustness and generalisation. With an accuracy of 81.6%, Random Forest not only outperforms the Decision Tree but also demonstrates better balance across precision, recall, and F1-score metrics. This makes it particularly effective for datasets with complex patterns and interdependencies among features.

However, the increased performance of Random Forest comes with certain trade-offs. Its computational complexity is significantly higher due to the need to train and combine multiple decision trees, making it less practical for real-time or resource-constrained environments. Additionally, the ensemble nature of the model reduces its interpretability, as it becomes difficult to extract simple rules or understand the exact contribution of each feature to a prediction.

the Decision Tree and Random Forest models offer complementary strengths. The Decision Tree is ideal when interpretability is crucial and decision-making must be transparent, while the Random Forest is better suited for scenarios where accuracy and robustness are prioritised. By understanding the unique advantages and limitations of each model, practitioners can make informed choices based on the specific requirements of their application. For predicting diabetes, Random Forest is recommended for its superior performance, especially when accuracy outweighs the need for interpretability.

K-Nearest Neighbors (KNN)

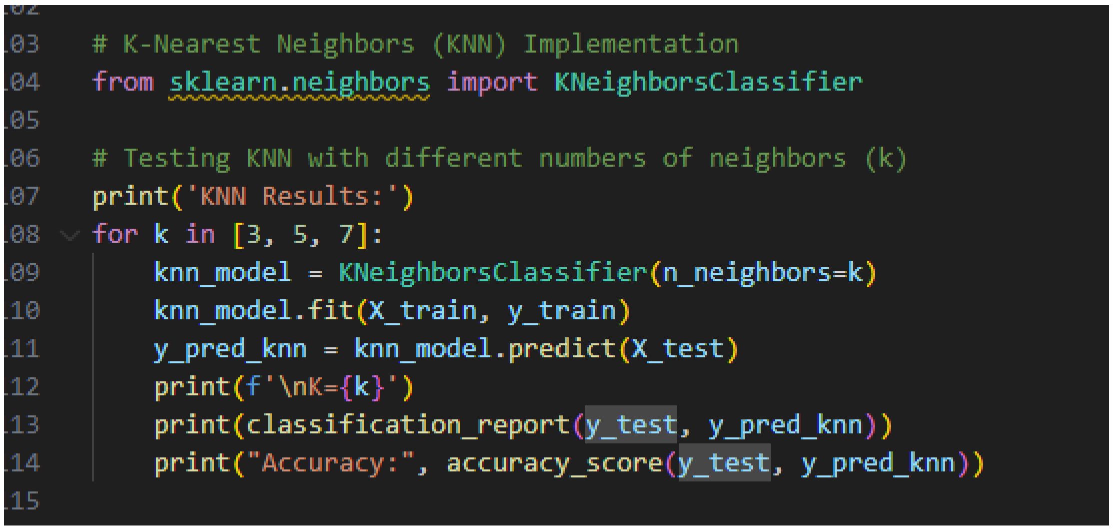
K-Nearest Neighbors (KNN) is a straightforward algorithm that classifies data points based on the class of their closest neighbours. It works by calculating the distance (commonly Euclidean) between a target data point and every other point in the dataset. The algorithm selects the kkk nearest points and assigns the majority class among them to the target.

KNN is easy to understand and effective for problems where the data forms distinct clusters. However, it can become computationally expensive when working with large datasets, as it must calculate distances for all points. It's most suitable for smaller datasets with clear boundaries between classes.

Steps:

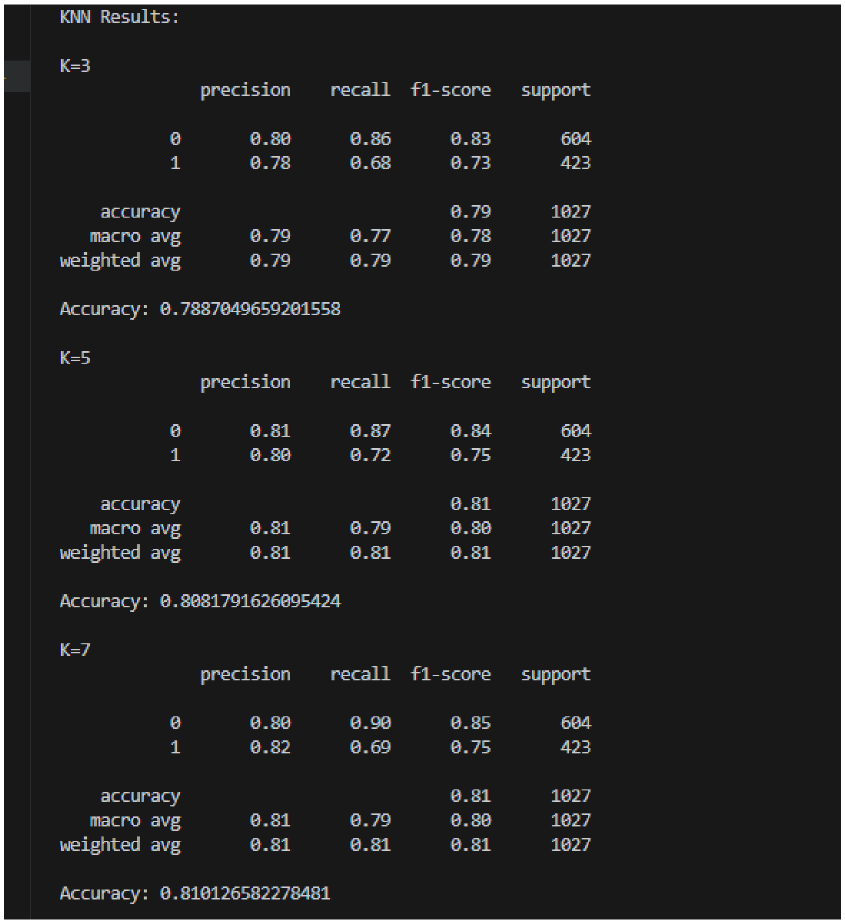
- Tests for k=3,5,7k = 3, 5, 7k=3,5,7.
- Outputs a classification report and accuracy for each k.

Code:



```python
# K-Nearest Neighbors (KNN) Implementation
from sklearn.neighbors import KNeighborsClassifier

# Testing KNN with different numbers of neighbors (k)
print('KNN Results:')
for k in [3, 5, 7]:
    knn_model = KNeighborsClassifier(n_neighbors=k)
    knn_model.fit(X_train, y_train)
    y_pred_knn = knn_model.predict(X_test)
    print(f'\nK={k}')
    print(classification_report(y_test, y_pred_knn))
    print("Accuracy:", accuracy_score(y_test, y_pred_knn))
```

Result:



```
KNN Results:

K=3
              precision    recall  f1-score   support

           0       0.80      0.86      0.83       604
           1       0.78      0.68      0.73       423

    accuracy                           0.79      1027
   macro avg       0.79      0.77      0.78      1027
weighted avg       0.79      0.79      0.79      1027

Accuracy: 0.7887049659201558

K=5
              precision    recall  f1-score   support

           0       0.81      0.87      0.84       604
           1       0.80      0.72      0.75       423

    accuracy                           0.81      1027
   macro avg       0.81      0.79      0.80      1027
weighted avg       0.81      0.81      0.81      1027

Accuracy: 0.8081791626095424

K=7
              precision    recall  f1-score   support

           0       0.80      0.90      0.85       604
           1       0.82      0.69      0.75       423

    accuracy                           0.81      1027
   macro avg       0.81      0.79      0.80      1027
weighted avg       0.81      0.81      0.81      1027

Accuracy: 0.810126582278481
```

Analysis of the Results:

1. For k=3:

While Class 0 performs well, Class 1 has a lower recall (68%), indicating the model struggles to correctly identify instances of Class 1 when k=3k = 3k=3.

2. For k=5:

Increasing k to 5 improves the overall accuracy and balances the precision for both classes. However, recall for Class 1 remains relatively low at 72%, which may indicate underfitting.

3. For k=7:

At k=7 the model reaches its highest accuracy, but the recall for Class 1 decreases slightly to 69%. The recall for Class 0 is strong at 90%, meaning the model is better at identifying Class 0 instances.

Support Vector Machine (SVM)

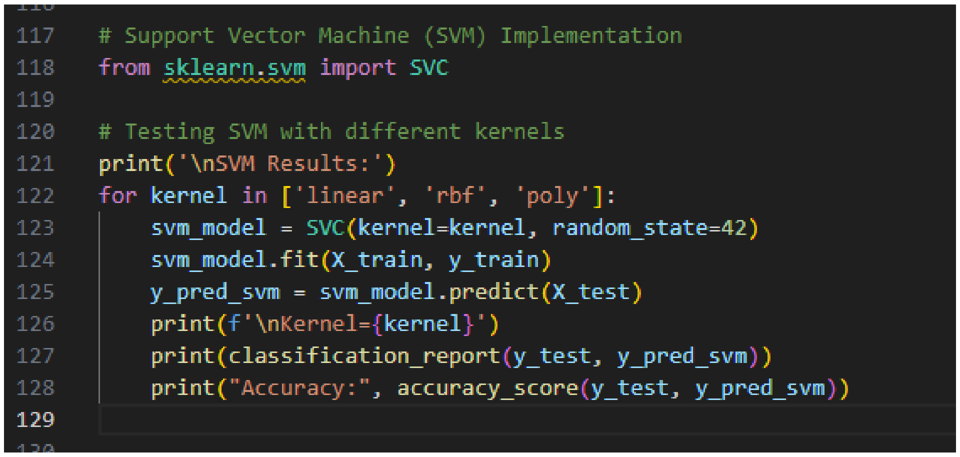
Support Vector Machine (SVM) is a robust supervised learning algorithm used to separate data points into distinct classes. It finds the optimal boundary, called a hyperplane, that divides classes while maximising the margin between them. The points closest to this boundary are known as support vectors, which play a critical role in defining the hyperplane.

SVM is highly effective for datasets where the classes are well-separated and can handle non-linear relationships using techniques like kernel functions (e.g., RBF or polynomial kernels). While it's powerful, SVM can be resource-intensive for large datasets and may require fine-tuning to achieve optimal performance.

Steps:

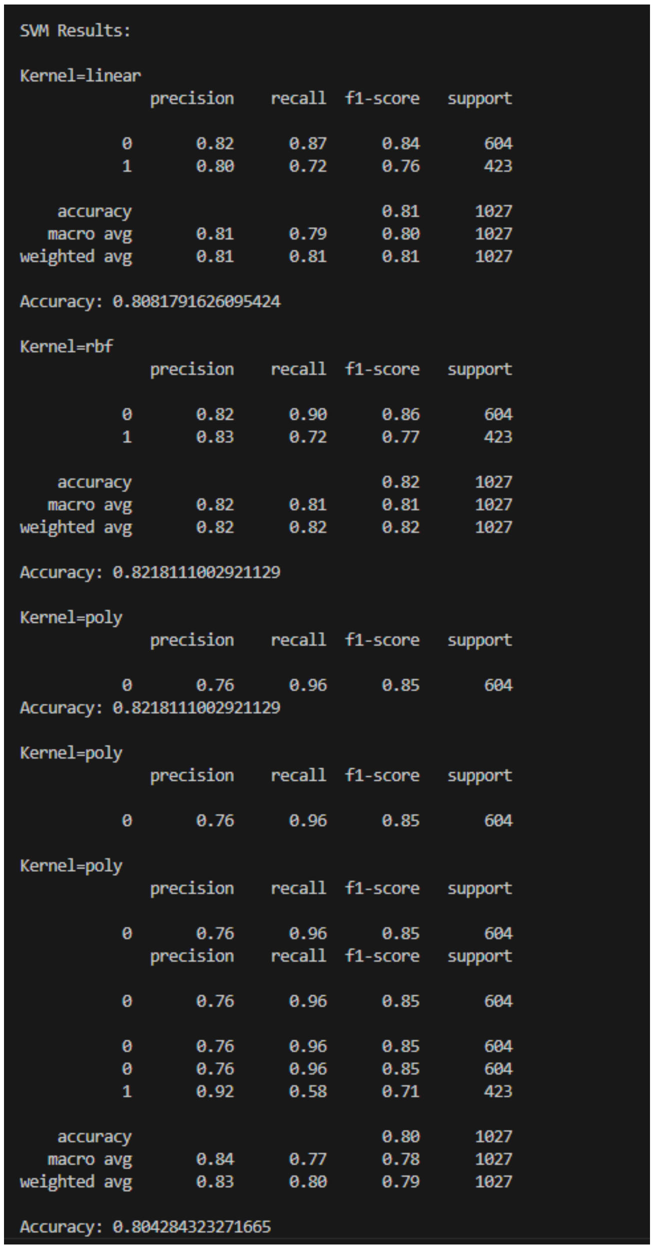
- Tests for three different kernels: 'linear', 'rbf', and 'poly'.
- Outputs a classification report and accuracy for each kernel.

Code:



```python
# Support Vector Machine (SVM) Implementation
from sklearn.svm import SVC

# Testing SVM with different kernels
print('\nSVM Results:')
for kernel in ['linear', 'rbf', 'poly']:
    svm_model = SVC(kernel=kernel, random_state=42)
    svm_model.fit(X_train, y_train)
    y_pred_svm = svm_model.predict(X_test)
    print(f'\nKernel={kernel}')
    print(classification_report(y_test, y_pred_svm))
    print("Accuracy:", accuracy_score(y_test, y_pred_svm))

```

Resutl:



```
SVM Results:

Kernel=linear
              precision    recall  f1-score   support

           0       0.82      0.87      0.84       604
           1       0.80      0.72      0.76       423

    accuracy                           0.81      1027
   macro avg       0.81      0.79      0.80      1027
weighted avg       0.81      0.81      0.81      1027

Accuracy: 0.8081791626095424

Kernel=rbf
              precision    recall  f1-score   support

           0       0.82      0.90      0.86       604
           1       0.83      0.72      0.77       423

    accuracy                           0.82      1027
   macro avg       0.82      0.81      0.81      1027
weighted avg       0.82      0.82      0.82      1027

Accuracy: 0.8218111002921129

Kernel=poly
              precision    recall  f1-score   support

           0       0.76      0.96      0.85       604
Accuracy: 0.8218111002921129

Kernel=poly
              precision    recall  f1-score   support

           0       0.76      0.96      0.85       604

Kernel=poly
              precision    recall  f1-score   support

           0       0.76      0.96      0.85       604
              precision    recall  f1-score   support

           0       0.76      0.96      0.85       604

           0       0.76      0.96      0.85       604
           0       0.76      0.96      0.85       604
           1       0.92      0.58      0.71       423

    accuracy                           0.80      1027
   macro avg       0.84      0.77      0.78      1027
weighted avg       0.83      0.80      0.79      1027

Accuracy: 0.804284323271665
```

Analysis of the Results:

1. Kernel: Linear

The linear kernel performs well with a balanced precision and recall. However, it struggles to recall Class 1 instances, leading to slightly lower performance in that class.

2. Kernel: RBF (Radial Basis Function)

The RBF kernel provides better precision for Class 1 compared to the linear kernel and achieves the highest overall accuracy (~82.18%). However, the recall for Class 1 remains the same as with the linear kernel (72%).

3. Kernel: Polynomial (Poly)

The polynomial kernel significantly improves the precision for Class 1 (92%), but at the cost of recall, which drops to 58%. This indicates the model is highly confident but fails to generalise well for Class 1 instances. The overall accuracy drops slightly compared to the RBF kernel.

Naive Bayes

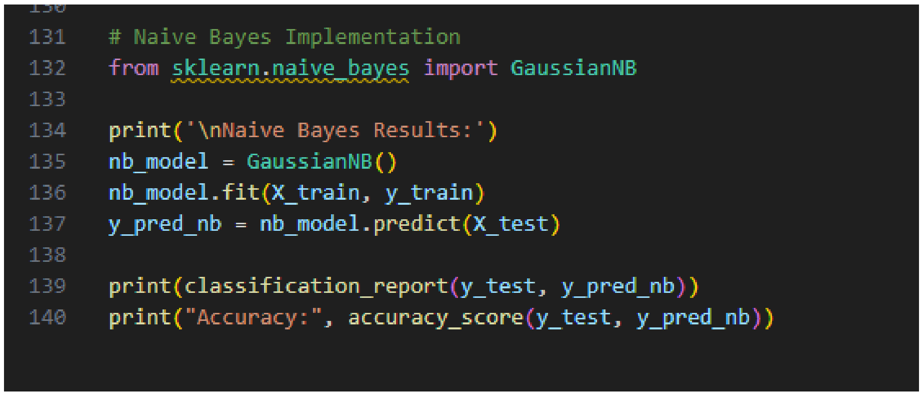
Naive Bayes is a probabilistic algorithm that applies Bayes' Theorem to classify data points. It calculates the likelihood of a data point belonging to each class and assigns it to the class with the highest probability. The algorithm assumes all features are independent, which simplifies computation but may not always hold true in practice.

Despite this assumption, Naive Bayes often performs well, particularly in scenarios involving text or categorical data, such as spam email detection or sentiment analysis. Its simplicity and speed make it a popular choice for large datasets.

Steps:

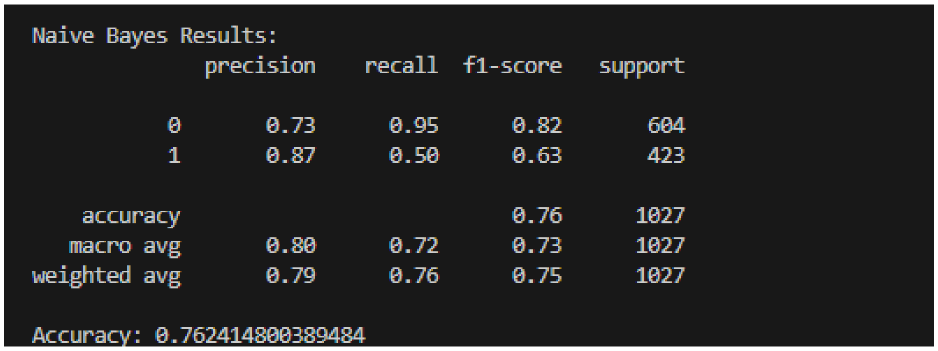
- Uses the GaussianNB implementation for classification.
- Outputs a single classification report and accuracy score.

Code:



```python
# Naive Bayes Implementation
from sklearn.naive_bayes import GaussianNB

print('\nNaive Bayes Results:')
nb_model = GaussianNB()
nb_model.fit(X_train, y_train)
y_pred_nb = nb_model.predict(X_test)

print(classification_report(y_test, y_pred_nb))
print("Accuracy:", accuracy_score(y_test, y_pred_nb))
```

Result:



```
Naive Bayes Results:
              precision    recall  f1-score   support

           0       0.73      0.95      0.82       604
           1       0.87      0.50      0.63       423

    accuracy                           0.76      1027
   macro avg       0.80      0.72      0.73      1027
weighted avg       0.79      0.76      0.75      1027

Accuracy: 0.762414800389484
```

Analysis of Naive Bayes Results:

1. Class 0 Performance:

The model excels in identifying Class 0 instances, with high recall indicating that it correctly predicts the majority of them.

2. Class 1 Performance:

While precision for Class 1 is high, the recall is significantly lower. This indicates that the model struggles to correctly classify a large portion of actual Class 1 instances.

Comparing the Algorithms

Algorithm	Variation	Accuracy	Precision (Class 0 / 1)	Recall (Class 0 / 1)	F1-Score (Class 0 / 1)	Strengths	Weaknesses
KNN	k = 3	78.87%	80% / 78%	86% / 68%	83% / 73%	High recall for Class 0.	Struggles with recall for Class 1.
	k = 5	80.82%	81% / 80%	87% / 72%	84% / 75%	Balanced performance; good precision.	Slightly lower recall for Class 1.
	k = 7	81.01%	80% / 82%	90% / 69%	85% / 75%	Best overall performance among KNN variations.	Low recall for Class 1 compared to Class 0.
SVM	Linear Kernel	80.82%	82% / 80%	87% / 72%	84% / 76%	Handles linear separable data effectively.	Moderate recall for Class 1.
	RBF Kernel	82.18%	82% / 83%	90% / 72%	86% / 77%	Best overall accuracy; balances precision and recall.	Computationally expensive for large data.
	Polynomial Kernel	80.42%	76% / 92%	96% / 58%	85% / 71%	High precision for Class 1; high recall for Class 0.	Poor recall for Class 1 (many false negatives).

Naive Bayes	GaussianNB	76.24%	73% / 87%	95% / 50%	82% / 63%	Fast and simple; excellent recall for Class 0.	Low recall for Class 1; biased towards Class 0.