Here’s a detailed line-by-line explanation of the provided code:

```python

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.preprocessing import StandardScaler

from sklearn.neighbors import KNeighborsClassifier

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import confusion\_matrix, f1\_score, recall\_score, precision\_score, accuracy\_score

```

1. \*\*Import Libraries\*\*:

- `numpy`: A library for numerical computations, particularly with arrays and matrices.

- `pandas`: A library for data manipulation and analysis, especially with tabular data (dataframes).

- `matplotlib.pyplot`: A plotting library for creating static, animated, and interactive visualizations in Python.

- `seaborn`: A statistical data visualization library based on Matplotlib that provides a high-level interface for drawing attractive graphics.

- `StandardScaler`: A class from `sklearn.preprocessing` for scaling features to have a mean of 0 and a variance of 1.

- `KNeighborsClassifier`: A machine learning algorithm for classification based on the k-nearest neighbors approach.

- `train\_test\_split`: A utility from `sklearn.model\_selection` for splitting datasets into training and testing sets.

- Various metrics (confusion matrix, f1\_score, recall\_score, precision\_score, accuracy\_score) from `sklearn.metrics` for evaluating the performance of classification models.

---

```python

df=pd.read\_csv("C:\\Users\\Asus\\Downloads\\diabetes.csv")

```

2. \*\*Load Data\*\*: Reads a CSV file named `diabetes.csv` into a DataFrame called `df`. This file is expected to contain diabetes-related data.

---

```python

df.head()

```

3. \*\*Preview Data\*\*: Displays the first five rows of the DataFrame `df` to get a quick view of the dataset's structure and the type of data it contains.

---

```

Pregnancies Glucose BloodPressure SkinThickness Insulin BMI \

0 6 148 72 35 0 33.6

1 1 85 66 29 0 26.6

2 8 183 64 0 0 23.3

3 1 89 66 23 94 28.1

4 0 137 40 35 168 43.1

Pedigree Age Outcome

0 0.627 50 1

1 0.351 31 0

2 0.672 32 1

3 0.167 21 0

4 2.288 33 1

```

4. \*\*Data Structure\*\*: This block shows the actual output of `df.head()`, indicating the columns:

- `Pregnancies`: Number of pregnancies.

- `Glucose`: Glucose level.

- `BloodPressure`: Blood pressure value.

- `SkinThickness`: Thickness of skin fold.

- `Insulin`: Insulin level.

- `BMI`: Body Mass Index.

- `Pedigree`: Diabetes pedigree function.

- `Age`: Age of the patient.

- `Outcome`: Target variable indicating diabetes status (1 for diabetic, 0 for non-diabetic).

---

```python

df.shape

```

5. \*\*Check Data Shape\*\*: Returns the shape of the DataFrame `df`, which is a tuple showing the number of rows and columns (768 rows, 9 columns in this case).

---

```python

df.describe()

```

6. \*\*Descriptive Statistics\*\*: Generates summary statistics for each numeric column in the DataFrame, including count, mean, standard deviation, minimum, maximum, and quartiles. This helps in understanding the distribution and spread of the data.

---

```

Pregnancies Glucose BloodPressure SkinThickness Insulin \

count 768.000000 768.000000 768.000000 768.000000 768.000000

mean 3.845052 120.894531 69.105469 20.536458 79.799479

std 3.369578 31.972618 19.355807 15.952218 115.244002

min 0.000000 0.000000 0.000000 0.000000 0.000000

25% 1.000000 99.000000 62.000000 0.000000 0.000000

50% 3.000000 117.000000 72.000000 23.000000 30.500000

75% 6.000000 140.250000 80.000000 32.000000 127.250000

max 17.000000 199.000000 122.000000 99.000000 846.000000

BMI Pedigree Age Outcome

count 768.000000 768.000000 768.000000 768.000000

mean 31.992578 0.471876 33.240885 0.348958

std 7.884160 0.331329 11.760232 0.476951

min 0.000000 0.078000 21.000000 0.000000

25% 27.300000 0.243750 24.000000 0.000000

50% 32.000000 0.372500 29.000000 0.000000

75% 36.600000 0.626250 41.000000 1.000000

max 67.100000 2.420000 81.000000 1.000000

```

7. \*\*Summary Statistics Output\*\*: This block shows the actual output of `df.describe()`, which provides insights into each column's distribution.

---

```python

# Replace zeros

zero\_not\_accepted=["Glucose","BloodPressure","SkinThickness","BMI","Insulin"]

for column in zero\_not\_accepted:

df[column]=df[column].replace(0,np.NaN)

mean=int(df[column].mean(skipna=True))

df[column]=df[column].replace(np.NaN,mean)

```

8. \*\*Handle Zeros\*\*: This block identifies columns that should not contain zero values (e.g., Glucose, Blood Pressure) and replaces zeros with `NaN`. Then it calculates the mean of each column (ignoring `NaN` values) and replaces `NaN` with this mean.

- `zero\_not\_accepted`: List of columns to check for zero values.

- `replace(0, np.NaN)`: Converts zeros to `NaN`.

- `mean=int(df[column].mean(skipna=True))`: Calculates the mean while ignoring `NaN` values.

- `replace(np.NaN, mean)`: Fills `NaN` values with the calculated mean.

---

```python

df["Glucose"]

```

9. \*\*Access Glucose Column\*\*: This line extracts the `Glucose` column from the DataFrame. The output below shows the values in the `Glucose` column.

```

0 148.0

1 85.0

2 183.0

3 89.0

4 137.0

...

763 101.0

764 122.0

765 121.0

766 126.0

767 93.0

Name: Glucose, Length: 768, dtype: float64

```

---

```python

# Split dataset

X=df.iloc[:,0:8]

y=df.iloc[:,8]

```

10. \*\*Split Features and Target\*\*: Separates the DataFrame into feature variables `X` (the first 8 columns) and the target variable `y` (the 9th column, `Outcome`).

---

```python

X\_train,X\_test,y\_train,y\_test=train\_test\_split(X,y,random\_state=0,test\_size=0.2)

```

11. \*\*Train-Test Split\*\*: Uses `train\_test\_split` to randomly divide the data into training and testing sets. Here, 20% of the data is reserved for testing.

- `random\_state=0`: Ensures reproducibility of the split.

---

```python

# Feature Scaling

sc\_X=StandardScaler()

X\_train=sc\_X.fit\_transform(X\_train)

X\_test=sc\_X.transform(X\_test)

```

12. \*\*Feature Scaling\*\*: Standardizes the feature variables to have a mean of 0 and a standard deviation of 1 using `StandardScaler`. This is important for many machine learning algorithms that are sensitive to the scale of the data.

- `fit\_transform(X\_train)`: Fits the scaler to the training data and transforms it.

- `transform(X\_test)`: Transforms the test data using the parameters learned from the training data.

---

```python

knn=KNeighborsClassifier(n\_neighbors=11)

```

13. \*\*Instantiate K-Neighbors Classifier\*\*: Creates a

KNeighborsClassifier object with `n\_neighbors=11`, meaning it will consider the 11 nearest neighbors to make predictions.

---

```python

knn.fit(X\_train,y\_train)

```

14. \*\*Train the Model\*\*: Fits the KNeighborsClassifier to the training data (`X\_train` and `y\_train`), allowing the model to learn from the data.

---

```python

y\_pred=knn.predict(X\_test)

```

15. \*\*Make Predictions\*\*: Uses the trained KNeighborsClassifier model to predict the labels for the test set (`X\_test`), storing the predicted outcomes in `y\_pred`.

---

```python

# Evaluate The Model

cf\_matrix=confusion\_matrix(y\_test,y\_pred)

```

16. \*\*Evaluate Model Performance\*\*: Computes the confusion matrix comparing the actual outcomes (`y\_test`) and the predicted outcomes (`y\_pred`).

---

```python

ax = sns.heatmap(cf\_matrix, annot=True, cmap='Blues')

ax.set\_title('Seaborn Confusion Matrix with labels\n\n')

ax.set\_xlabel('\nPredicted Values')

ax.set\_ylabel('Actual Values ');

```

17. \*\*Visualize Confusion Matrix\*\*: Uses Seaborn to create a heatmap of the confusion matrix.

- `annot=True`: Annotates the cells with the numeric values.

- `cmap='Blues'`: Sets the color map to blue.

---

```python

## Display the visualization of the Confusion Matrix.

plt.show()

```

18. \*\*Display Plot\*\*: Renders the heatmap on the screen.

---

```python

tn, fp, fn, tp = confusion\_matrix(y\_test, y\_pred).ravel()

```

19. \*\*Extract Confusion Matrix Values\*\*: Unpacks the values from the confusion matrix into four variables:

- `tn`: True Negatives

- `fp`: False Positives

- `fn`: False Negatives

- `tp`: True Positives

---

```python

tn, fp, fn, tp

```

20. \*\*Display Confusion Matrix Values\*\*: Outputs the extracted values, which are:

- True Negatives: 94

- False Positives: 13

- False Negatives: 15

- True Positives: 32

---

```python

# The accuracy rate is equal to (tn+tp)/(tn+tp+fn+fp)

accuracy\_score(y\_test,y\_pred)

```

21. \*\*Calculate Accuracy\*\*: Computes the accuracy of the model using `accuracy\_score`, which is the ratio of correctly predicted instances (true positives + true negatives) to the total instances. This will display the accuracy.

---

```python

# The precision is the ratio of tp/(tp + fp)

precision\_score(y\_test,y\_pred)

```

22. \*\*Calculate Precision\*\*: Computes the precision of the model using `precision\_score`, which measures the proportion of true positive predictions among all positive predictions (true positives / (true positives + false positives)).

---

```python

## The recall is the ratio of tp/(tp + fn)

recall\_score(y\_test,y\_pred)

```

23. \*\*Calculate Recall\*\*: Computes the recall of the model using `recall\_score`, which measures the ability of the classifier to find all positive instances (true positives / (true positives + false negatives)).

---

```python

#error rate=1-accuracy which is lies between 0 and 1

error\_rate=1 - accuracy\_score(y\_test,y\_pred)

```

24. \*\*Calculate Error Rate\*\*: Computes the error rate as `1 - accuracy`, which represents the proportion of incorrect predictions. The error rate will also be between 0 and 1.

---

```python

error\_rate

```

25. \*\*Display Error Rate\*\*: Outputs the calculated error rate based on the accuracy score.

---

This comprehensive breakdown covers each line of the provided code, explaining the functionality and significance of every part in the context of loading, processing, and modeling a diabetes dataset using machine learning techniques.