Data testing with KNN

In [1]:

```
import pandas as pd
```

Pandas is a powerful data manipulation library for Python. It provides data structures like DataFrames and Series, which allow you to work with structured data in a more intuitive way. DataFrames are similar to tables in a database or Excel spreadsheet, making it easier to analyze and manipulate data.

In [2]:

```
import numpy as np
```

NumPy is a fundamental package for scientific computing with Python. It provides support for arrays and matrices, along with a wide range of mathematical functions to operate on these arrays. NumPy is often used for numerical and mathematical operations on large datasets.

In [54]:

```
data=pd.read_csv("D:\csv\diabetes.csv")
```

In [4]:

```
data.head(3)
```

Out[4]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunct
0	6	148	72	35	0	33.6	0.6
1	1	85	66	29	0	26.6	0.3
2	8	183	64	0	0	23.3	0.6

In [5]:

```
print(len(data))
```

```
768
```

In [8]:

```
# replace zeros
zero_not_accepted=['Glucose','BloodPressure','SkinThickness','Insulin','BMI']
```

In [10]:

```
for column in zero_not_accepted:
    data[column]=data[column].replace(0,np.NaN)
    mean=int(data[column].mean(skipna=True))
    data[column]=data[column].replace(np.NaN,mean)
```

for column in zeros_not_accepted:: This is a loop that iterates through the columns specified in the zeros_not_accepted list. It suggests that you want to apply this data cleaning operation to specific columns and not the entire DataFrame.

data[column] = data[column].replace(0, np.NaN): In this line, you are replacing all the zero values (0) in the selected column with NaN (Not a Number). NaN is often used to represent missing or undefined values in a dataset.

mean = int(data[column].mean(skipna=True)): Here, you are calculating the mean (average) of the non-missing values in the selected column. The skipna=True argument ensures that NaN values are excluded from the calculation. The result is converted to an integer using int().

data[column] = data[column].replace(np.NaN, mean): Finally, you are replacing all the NaN values in the selected column with the calculated mean value. This step essentially imputes (fills in) the missing or zero values in the column with the mean value of the non-missing values.

In summary, this code snippet is used to handle missing or zero values in specific columns of your DataFrame by replacing them with the mean of the non-missing values in the same column. This is a common strategy for data imputation, and it helps ensure that your data is suitable for machine learning algorithms that might not handle missing values well. The process involves identifying problematic values (zeros in this case), converting them to NaN, calculating the mean of the remaining values, and then replacing the NaN values with the mean.

In [11]:

```
# input features(independent variable)
x=data.iloc[:,0:8]
x.head()
```

Out[11]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunct
0	6	148.0	72.0	35.0	155.0	33.6	0.6
1	1	85.0	66.0	29.0	155.0	26.6	0.3
2	8	183.0	64.0	29.0	155.0	23.3	0.6
3	1	89.0	66.0	23.0	94.0	28.1	0.1
4	0	137.0	40.0	35.0	168.0	43.1	2.2

data: This is assumed to be a DataFrame containing your dataset, where rows represent observations or samples, and columns represent different features or variables.

.iloc[]: This is an indexing method in Pandas used for integer-location based indexing.

[:, 0:8]: This is a slice operation specifying the rows and columns to select:

: in the row position means you want to select all rows. 0:8 in the column position means you want to select columns from index 0 up to, but not including, index 8. This will select columns with indices 0, 1, 2, 3, 4, 5, 6, and 7. So, x now contains a subset of your original DataFrame data, with all rows and the first 8 columns. This is typically done to extract the input features (independent variables) for your machine learning model.

In [12]:

```
# target variable(depending variable)
y=data.iloc[:,8]
y.head()
```

Out[12]:

```
0    1
1    0
2    1
3    0
4    1
Name: Outcome, dtype: int64
```

data: This is the same DataFrame as before.

.iloc[]: Again, this is the integer-location based indexing method.

[:, 8]: This specifies that you want to select all rows (indicated by :) and the column with index 8. In Python indexing, the first column has an index of 0, the second has an index of 1, and so on.

So, y now contains a subset of your original DataFrame data, consisting of all rows but only the column with index 8. Typically, this is done to extract the target variable (dependent variable) you want to predict using a machine learning model.

In summary, x contains the input features for your model, and y contains the corresponding target variable. These are often used in supervised machine learning where you have labeled data and you want to train a model to make predictions based on these features.

In [13]:

```
data.head()
```

Out[13]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunct
0	6	148.0	72.0	35.0	155.0	33.6	0.6
1	1	85.0	66.0	29.0	155.0	26.6	0.3
2	8	183.0	64.0	29.0	155.0	23.3	0.6
3	1	89.0	66.0	23.0	94.0	28.1	0.1
4	0	137.0	40.0	35.0	168.0	43.1	2.2

sklearn

Scikit-learn is a machine learning library in Python that provides tools for various tasks like classification, regression, clustering, dimensionality reduction, and more. It's built on top of NumPy, SciPy, and Matplotlib.

train_test_split

train_test_split is a function from scikit-learn used to split a dataset into training and testing sets. This is essential to evaluate the performance of a machine learning model. It helps prevent overfitting by creating separate sets for training and testing.

In [14]:

```
from sklearn.model_selection import train_test_split
```

This function is used for splitting a dataset into two subsets: a training set and a testing (or validation) set. It's commonly used in machine learning to assess how well a model performs on data it has not seen during training. It takes your dataset and splits it into two parts, typically with a larger portion for training and a smaller portion for testing.

In [15]:

```
x_train,x_test,y_train,y_test=train_test_split(x,y,random_state=0,test_size=0.2)
```

In [16]:

```
x_train
```

Out[16]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFun
603	7	150.0	78.0	29.0	126.0	35.2	
118	4	97.0	60.0	23.0	155.0	28.2	
247	0	165.0	90.0	33.0	680.0	52.3	
157	1	109.0	56.0	21.0	135.0	25.2	
468	8	120.0	72.0	29.0	155.0	30.0	
...	...	...	...	...	...	...	
763	10	101.0	76.0	48.0	180.0	32.9	
192	7	159.0	66.0	29.0	155.0	30.4	
629	4	94.0	65.0	22.0	155.0	24.7	
559	11	85.0	74.0	29.0	155.0	30.1	
684	5	136.0	82.0	29.0	155.0	32.0	

614 rows × 8 columns

In [17]:

```
len(x_train)
```

Out[17]:

```
614
```

In [18]:

```
y_train
```

Out[18]:

```
603    1
118    0
247    0
157    0
468    1
      ..
763    0
192    1
629    0
559    0
684    0
Name: Outcome, Length: 614, dtype: int64
```

In [19]:

```
len(y_train)
```

Out[19]:

```
614
```

StandardScaler

StandardScaler is a preprocessing technique in scikit-learn that standardizes the features by removing the mean and scaling to unit variance. This is important for many machine learning algorithms as it helps ensure that all features have the same scale, which can improve the model's convergence and performance.

In [20]:

```
from sklearn.preprocessing import StandardScaler
```

StandardScaler is a preprocessing technique used to scale and center your data. It's important in many machine learning algorithms because it ensures that all features have the same mean (zero) and standard deviation (one). This can improve the performance of certain algorithms, particularly those that rely on distances between data points, like k-means clustering or support vector machines.

In [21]:

```
# feature scalling
sc_x=StandardScaler()
```

The code you provided is performing feature scaling on your data using the StandardScaler from scikit-learn (sc_x = StandardScaler()). Feature scaling is a preprocessing step in machine learning that standardizes or normalizes the features in your dataset to have similar scales. This is important because many machine learning algorithms are sensitive to the scale of the input features, and scaling them can improve the performance and convergence of these algorithms.

In [22]:

```
x_train=sc_x.fit_transform(x_train)
```

This line applies the standardization transformation to your training data (x_train). The fit_transform method first computes the mean and standard deviation for each feature in x_train and then scales the features based on these statistics. After this line is executed, x_train will contain the scaled features.

In [23]:

```
x_test=sc_x.transform(x_test)
```

This line applies the same transformation to your test data (x_test) that was computed based on the training data. It's important to use the statistics (mean and standard deviation) calculated from the training data to scale the test data to ensure consistency. The transform method only applies the scaling based on the statistics calculated during the fit_transform step; it doesn't recompute these statistics.

In [215]:

```
 # feature scalling
sc_x=StandardScaler()
x_train=sc_x.fit_transform(x_train)
x_test=sc_x.transform(x_test)
```

In [24]:

```
from sklearn.neighbors import KNeighborsClassifier
```

`KNeighborsClassifier` is a supervised machine learning algorithm for classification tasks provided by scikit-learn (sklearn). It's a simple yet effective method that falls under the category of instance-based or lazy learning. Here's an explanation of `KNeighborsClassifier` :

1. **Nearest Neighbors Algorithm**:

   - The `KNeighborsClassifier` is based on the nearest neighbors algorithm. It's used for both classification and regression tasks, but in this case, we're focusing on classification.

2. **K-Nearest Neighbors (K-NN)**:

   - The central idea of K-NN is to make predictions for a new data point by considering the class labels of its nearest neighbors in the training dataset.
   - The "K" in K-NN refers to the number of nearest neighbors to consider. For example, if K is set to 3, the algorithm will consider the class labels of the three nearest neighbors to make a prediction.

3. **How K-NN Works**:

   - Given a new data point, K-NN calculates the distances (often Euclidean distance) between that point and all the points in the training dataset.
   - It then selects the K nearest neighbors (the data points with the smallest distances) to the new data point.
   - For classification, it assigns the class label that is most common among those K neighbors to the new data point.

4. **Hyperparameter Tuning**:

   - The most critical hyperparameter in K-NN is the value of K. Choosing the right K value is crucial, as it can significantly impact the model's performance.

   - Other hyperparameters, such as the distance metric (Euclidean, Manhattan, etc.), can also be adjusted to suit the specific problem.
5. **Pros**:
   - K-NN is simple to understand and easy to implement.
   - It can work well for both multi-class and binary classification problems.
   - It's a non-parametric algorithm, meaning it doesn't make assumptions about the underlying data distribution.
6. **Cons**:
   - K-NN can be sensitive to the choice of K, and selecting the wrong K can lead to poor results.
   - It can be computationally expensive, especially for large datasets, as it requires calculating distances to all data points during prediction.
   - It doesn't perform well when there are irrelevant or noisy features in the data.
7. **Use Cases**:
   - K-NN is often used in recommendation systems, image recognition, and various other classification tasks.
   - It can be a good starting point for exploring a dataset or as a baseline model for classification problems.

To use `KNeighborsClassifier` in scikit-learn, you typically create an instance of the classifier, fit it to your training data, and then use it to make predictions for new, unseen data points. You can also fine-tune the

In [25]:

```
classifier=KNeighborsClassifier(n_neighbors=11,p=2,metric="euclidean")
```

It will consider 11 nearest neighbors when making predictions. It will use the Euclidean distance metric to measure the distance between data points. These hyperparameters can be adjusted based on your specific problem and dataset. The choice of n_neighbors and the distance metric can significantly impact the performance of the K-NN classifier, so it's important to experiment and choose values that work best for your particular use case.

In [26]:

```
classifier.fit(x_train,y_train)
```

Out[26]:

```
KNeighborsClassifier(metric='euclidean', n_neighbors=11)
```

In [27]:

```
# predict the test set results
y_pred=classifier.predict(x_test)
y_pred
```

```
C:\Users\ASUS\anaconda3\lib\site-packages\sklearn\neighbors\_classificatio
n.py:228: FutureWarning: Unlike other reduction functions (e.g. `skew`, `k
urtosis`), the default behavior of `mode` typically preserves the axis it
acts along. In SciPy 1.11.0, this behavior will change: the default value
of `keepdims` will become False, the `axis` over which the statistic is ta
ken will be eliminated, and the value None will no longer be accepted. Set
`keepdims` to True or False to avoid this warning.
  mode, _ = stats.mode(_y[neigh_ind, k], axis=1)
```

Out[27]:

```
array([1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0,
       0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1,
       1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1,
       1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1,
       0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
      dtype=int64)
```

y_pred: This is a variable where the predicted labels (or classes) for the test data points will be stored.

classifier.predict(x_test): This is the core prediction step using the trained K-NN classifier. Here's what happens:

x_test: This is the set of test data, which typically contains the same features as the training data but for unseen instances.

classifier: This is the trained K-NN classifier you previously created with specified hyperparameters.

predict(): This method of the KNeighborsClassifier class takes the test data x_test and uses the trained model to predict the class labels for each data point in x_test.

So, after executing this line of code, y_pred will contain an array or series of predicted class labels for each data point in the x_test dataset.

For example, if you are working on a classification problem where you are predicting whether an email is spam (1) or not spam (0), y_pred would be an array containing these predicted labels for the test emails based on the K-NN model's analysis of the test data. These predicted labels can then be compared to the actual labels (y_test) to evaluate the performance of the classifier using various metrics such as accuracy, precision, recall, F1-score, etc.

confusion matrix

confusion_matrixThe confusion matrix is a table used in classification to evaluate the performance of a classification model. It compares the predicted classes with the actual classes and shows the counts of true positive, true negative, false positive, and false negative predictions. It's useful to understand the quality of the model's predictions.

In [28]:

```
from sklearn.metrics import confusion_matrix
```

A confusion matrix is used to evaluate the performance of a classification model. It provides a summary of the actual versus predicted class labels for a classification problem. It is particularly useful for understanding the types of errors your model is making, such as false positives and false negatives.

In [29]:

```
cm=confusion_matrix(y_test,y_pred)
print(cm)
```

```
[[94 13]
 [15 32]]
```

This function calculates the confusion matrix for a classification model based on the actual target labels (y_test) and the predicted labels (y_pred). Here's how the confusion matrix is structured:

True Positives (TP): The number of data points that were actually positive (e.g., class 1) and were correctly predicted as positive by the model.

True Negatives (TN): The number of data points that were actually negative (e.g., class 0) and were correctly predicted as negative by the model.

False Positives (FP): The number of data points that were actually negative but were incorrectly predicted as positive by the model. These are also called Type I errors.

False Negatives (FN): The number of data points that were actually positive but were incorrectly predicted as negative by the model. These are also called Type II errors.

NOTE---- Printing the confusion matrix (print(cm)) will display these counts in the console, helping you assess the model's performance. You can use these values to calculate various evaluation metrics like accuracy, precision, recall, and F1-score to get a more comprehensive understanding of how well your classification model is performing.

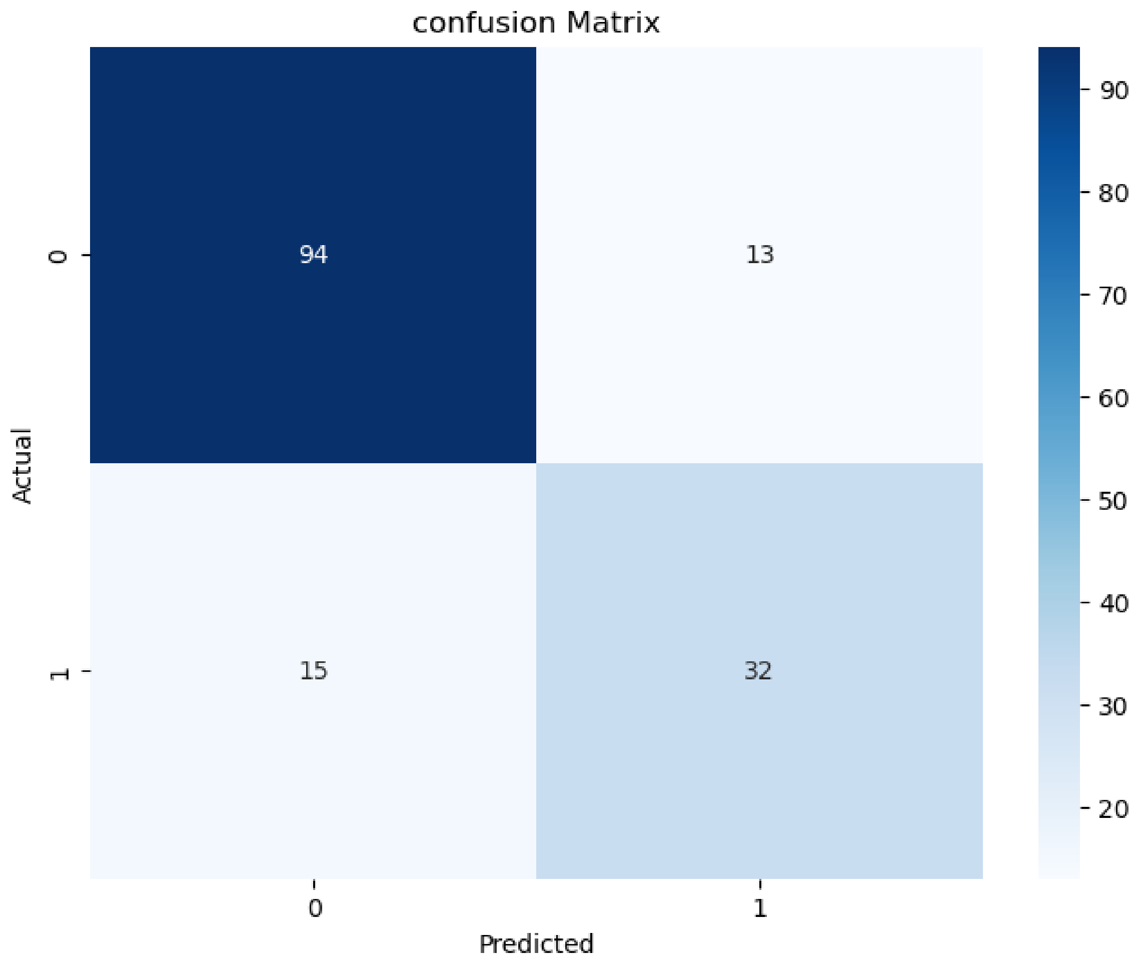
visualize the confusion matrix

In [44]:

```
import matplotlib.pyplot as plt
import seaborn as sns
```

In [49]:

```
plt.figure(figsize=(8,6))
sns.heatmap(cm,annot=True,fmt='d',cmap="Blues")
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('confusion Matrix')
plt.show()
```



f1_score

The F1 score is a metric used to evaluate the accuracy of a binary classification model, taking into account both precision and recall. It's the harmonic mean of precision and recall and is particularly useful when the class distribution is imbalanced.

In [31]:

```
from sklearn.metrics import f1_score
```

The F1 score is a metric used to evaluate the performance of a binary classification model. It takes into account both precision (the ability of the model to correctly identify positive cases) and recall (the ability of the model to capture all positive cases) to provide a single score that balances these two metrics. It's a good measure when you

In [32]:

```
f1_score(y_test,y_pred)
```

Out[32]:

```
0.6956521739130436
```

accuray_score

Accuracy score is a simple metric used to evaluate classification models. It calculates the ratio of correctly predicted instances to the total instances in the dataset. However, accuracy might not be the best metric when dealing with imbalanced classes.

In [33]:

```
from sklearn.metrics import accuracy_score
```

In [34]:

```
accuracy_score(y_test,y_pred)
```

Out[34]:

```
0.8181818181818182
```

BRAHMASTRA

In [53]:

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

data=pd.read_csv("D:\csv\diabetes.csv")
# replace zero
zero_not_accepted=['Glucose','BloodPressure','SkinThickness','Insulin','BMI']

# fill na value
for column in zero_not_accepted:
    data[column]=data[column].replace(0,np.NaN)
    mean=int(data[column].mean(skipna=True))
    data[column]=data[column].replace(np.NaN,mean)

#select the input data
x=data.iloc[:,0:8]

# select the output data
y=data.iloc[:,8]

# for predict the modeling
x_train,x_test,y_train,y_test=train_test_split(x,y,random_state=0,test_size=0.2)

#be similear we import this li..
sc_x=StandardScaler()
x_train=sc_x.fit_transform(x_train)
x_test=sc_x.transform(x_test)

# algorithum
classifier=KNeighborsClassifier(n_neighbors=11,p=2,metric="euclidean")
classifier.fit(x_train,y_train)
y_pred=classifier.predict(x_test)

# for cheching
cm=confusion_matrix(y_test,y_pred)
print(cm)
# for chaching
f1_score(y_test,y_pred)
print(f1_score)

# to get accuracy
accuracy_score(y_test,y_pred)
print(accuracy_score)
```

```
[[94 13]
 [15 32]]
<function f1_score at 0x0000019FA733BAF0>
<function accuracy_score at 0x0000019FA733B700>
```

```
C:\Users\ASUS\anaconda3\lib\site-packages\sklearn\neighbors\_classificatio
n.py:228: FutureWarning: Unlike other reduction functions (e.g. `skew`, `k
urtosis`), the default behavior of `mode` typically preserves the axis it
acts along. In SciPy 1.11.0, this behavior will change: the default value
of `keepdims` will become False, the `axis` over which the statistic is ta
ken will be eliminated, and the value None will no longer be accepted. Set
`keepdims` to True or False to avoid this warning.
  mode, _ = stats.mode(_y[neigh_ind, k], axis=1)
```

In [41]:

```
# Visualize the F1 Score (if needed)
print("F1 Score:", f1_score(y_test, y_pred))
```

```
F1 Score: 0.6956521739130436
```

In [42]:

```
# Visualize the Accuracy Score (if needed)
print("Accuracy Score:", accuracy_score(y_test, y_pred))
```

```
Accuracy Score: 0.8181818181818182
```

In [ ]:

In [ ]: