**Data Analytics-3**

import pandas as pd

import numpy as np

import seaborn as sns

import matplotlib.pyplot as plt

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

from sklearn.preprocessing import StandardScaler

from sklearn.naive\_bayes import GaussianNB

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

\_

df = pd.read\_csv("iris.csv")

df.head()

\_

# Set independent and dependent variables

X = df.drop('variety', axis=1) # Independent variable

y = df['variety'] # Dependent variable

# train test split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

\_

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

\_

# Train Naive Bayes model

model = GaussianNB()

model.fit(X\_train\_scaled, y\_train)

# Predict

y\_pred = model.predict(X\_test\_scaled)

\_

# Evaluate the model

cm = confusion\_matrix(y\_test, y\_pred, labels=model.classes\_)

cm\_df = pd.DataFrame(cm, index=model.classes\_, columns=model.classes\_)

# Plot Confusion Matrix

sns.heatmap(cm\_df, annot=True, cmap='Blues', fmt='d')

plt.title('Confusion Matrix')

plt.xlabel('Predicted')

plt.ylabel('Actual')

plt.show()

\_

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

precision = precision\_score(y\_test, y\_pred, average='macro')

recall = recall\_score(y\_test, y\_pred, average='macro')

error\_rate = 1 - accuracy

print(f"Accuracy: {accuracy:.2f}")

print(f"Error Rate: {error\_rate:.2f}")

print(f"Precision (Macro): {precision:.2f}")

print(f"Recall (Macro): {recall:.2f}")

Code Explanation:

1)import pandas as pd

import numpy as np

import seaborn as sns

import matplotlib.pyplot as plt

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

from sklearn.preprocessing import StandardScaler

from sklearn.naive\_bayes import GaussianNB

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

Let's break down the code step by step:

1. **Importing Libraries**:
   * import pandas as pd: This imports the pandas library and gives it an alias pd. pandas is used for data manipulation and analysis, especially with dataframes.
   * import numpy as np: This imports the numpy library as np. numpy is used for numerical operations, especially with arrays and matrices.
   * import seaborn as sns: This imports the seaborn library as sns. seaborn is a data visualization library built on top of matplotlib and is used for creating attractive and informative statistical graphics.
   * import matplotlib.pyplot as plt: This imports matplotlib as plt, which is a plotting library for creating visualizations such as graphs and charts.
   * from sklearn.model\_selection import train\_test\_split: This imports the train\_test\_split function from sklearn.model\_selection. This function is used to split the dataset into training and testing sets.
   * from sklearn.preprocessing import StandardScaler: This imports StandardScaler from sklearn.preprocessing. This tool is used for feature scaling, which standardizes the features of the dataset by removing the mean and scaling to unit variance.
   * from sklearn.naive\_bayes import GaussianNB: This imports GaussianNB, which is a specific implementation of the Naive Bayes classifier for Gaussian-distributed data.
   * from sklearn.metrics import confusion\_matrix, accuracy\_score, precision\_score, recall\_score: This imports various functions from sklearn.metrics that are used to evaluate the performance of the machine learning model. These metrics include:
     + confusion\_matrix: To compute the confusion matrix (TP, FP, TN, FN).
     + accuracy\_score: To calculate the accuracy of the model.
     + precision\_score: To calculate the precision (positive predictive value) of the model.
     + recall\_score: To calculate the recall (true positive rate) of the model.

Each of these libraries and functions is crucial for building, evaluating, and visualizing machine learning models in Python. Once you provide the code for implementing the Naive Bayes classification, I will continue with the explanation.

2). # Set independent and dependent variables

X = df.drop('variety', axis=1) # Independent variable

y = df['variety'] # Dependent variable

# train test split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

->

Let's break down this part of the code:

### 1. ****Set Independent and Dependent Variables****:

* **Independent Variable (X)**:
* X = df.drop('variety', axis=1)
  + df.drop('variety', axis=1): This line drops the 'variety' column from the DataFrame df. The axis=1 indicates that a column is being dropped (rows would be dropped if axis=0 was specified).
  + X represents the independent variables or features (i.e., all columns except the target column 'variety').
* **Dependent Variable (y)**:
* y = df['variety']
  + y is the dependent variable, which is the target variable we are trying to predict. In this case, it's the 'variety' column from the df DataFrame, which contains the species of Iris flowers.

### 2. ****Train-Test Split****:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

* train\_test\_split: This function from sklearn.model\_selection is used to split the dataset into training and testing subsets.
  + X: The independent variables (features).
  + y: The dependent variable (target).
  + test\_size=0.2: This means that 20% of the data will be used for testing, while the remaining 80% will be used for training.
  + random\_state=42: This ensures that the split is reproducible, meaning if you run the code multiple times, you will get the same training and test split. The number 42 is arbitrary and can be any integer.
* **Result**:
  + X\_train: Features for training.
  + X\_test: Features for testing.
  + y\_train: Labels for training (i.e., the correct 'variety' values corresponding to X\_train).
  + y\_test: Labels for testing (i.e., the correct 'variety' values corresponding to X\_test).

This step sets up the data for training and testing the model. You train the model using X\_train and y\_train, and then evaluate it on X\_test and y\_test.

3). scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

-> Let me explain this section of the code:

### 1. ****StandardScaler****:

scaler = StandardScaler()

* StandardScaler is a part of the sklearn.preprocessing module and is used to standardize the features of the dataset.
* The purpose of using StandardScaler is to normalize the dataset so that each feature has a mean of 0 and a standard deviation of 1. This is important when the features have different scales because some machine learning models, like Naïve Bayes and others, perform better when the data is scaled.

### 2. ****Scaling the Training Data****:

X\_train\_scaled = scaler.fit\_transform(X\_train)

* scaler.fit\_transform(X\_train):
  + The fit() part of the method calculates the mean and standard deviation for each feature in X\_train.
  + transform() applies the scaling to the data, adjusting each feature so that it has a mean of 0 and a standard deviation of 1.
  + fit\_transform() combines both steps: it first calculates the mean and standard deviation of the training data and then applies the transformation to it.
* X\_train\_scaled now contains the scaled version of the training dataset, which will be used to train the model.

### 3. ****Scaling the Test Data****:

X\_test\_scaled = scaler.transform(X\_test)

* scaler.transform(X\_test):
  + Here, we use the same scaler object that was fitted on the training data (scaler) to transform the test data.
  + The test data is scaled using the mean and standard deviation calculated from the training data, ensuring that both datasets are on the same scale.
  + X\_test\_scaled now contains the scaled version of the test dataset.

### Why Scaling is Important:

* Scaling is necessary because features with larger numerical ranges can dominate the model's performance. For instance, if one feature is in the range of 0-1000 and another is in the range of 0-1, the model may give more importance to the larger feature simply because of its magnitude.
* Scaling ensures that all features contribute equally to the model, allowing the model to learn more effectively.

To summarize:

* X\_train\_scaled: Scaled version of the training data.
* X\_test\_scaled: Scaled version of the testing data.

4) # Train Naive Bayes model

model = GaussianNB()

model.fit(X\_train\_scaled, y\_train)

# Predict

y\_pred = model.predict(X\_test\_scaled)

### -> Explanation of this part of the code:

### 1. ****Training the Naïve Bayes Model****:

model = GaussianNB()

model.fit(X\_train\_scaled, y\_train)

* **GaussianNB()**:
  + This is the class for **Gaussian Naïve Bayes**, a type of Naïve Bayes classifier. It assumes that the features follow a Gaussian (normal) distribution.
  + It's useful for classification problems, especially when the data is continuous and each feature follows a Gaussian distribution.
* **model.fit(X\_train\_scaled, y\_train)**:
  + .fit() is used to train the model.
  + X\_train\_scaled: The scaled features (independent variables) from the training dataset.
  + y\_train: The target variable (dependent variable) or the actual class labels for each row in the training data.
  + During the fitting process, the model learns the relationship between the features (X) and the target variable (y) using the training data. This includes calculating probabilities and the likelihoods necessary to apply the Naïve Bayes theorem.

### 2. ****Making Predictions****:

y\_pred = model.predict(X\_test\_scaled)

* **model.predict(X\_test\_scaled)**:
  + After training the model, we use the .predict() method to make predictions.
  + X\_test\_scaled: The scaled features of the test data, which the model uses to predict the class labels for the test samples.
  + The result, y\_pred, is an array of predicted class labels for each sample in the test set based on the features in X\_test\_scaled.

### Summary:

* **Training**: The model learns from the scaled training data (X\_train\_scaled) and corresponding labels (y\_train).
* **Prediction**: Once the model is trained, it predicts the class labels for the test data (X\_test\_scaled), which are stored in y\_pred.

At this point, the model has been trained and is ready to make predictions. Next, you would typically evaluate how well the model performs using metrics like accuracy, precision, recall, and confusion matrix (which is the next part of the code).

5) # Evaluate the model

cm = confusion\_matrix(y\_test, y\_pred, labels=model.classes\_)

cm\_df = pd.DataFrame(cm, index=model.classes\_, columns=model.classes\_)

# Plot Confusion Matrix

sns.heatmap(cm\_df, annot=True, cmap='Blues', fmt='d')

plt.title('Confusion Matrix')

plt.xlabel('Predicted')

plt.ylabel('Actual')

plt.show()

### -> Explanation of this part of the code:

### 1. ****Confusion Matrix Calculation****:

cm = confusion\_matrix(y\_test, y\_pred, labels=model.classes\_)

cm\_df = pd.DataFrame(cm, index=model.classes\_, columns=model.classes\_)

* **confusion\_matrix(y\_test, y\_pred, labels=model.classes\_)**:
  + confusion\_matrix is a function from sklearn.metrics that computes the confusion matrix to evaluate the performance of a classification model.
  + **y\_test**: The true labels (actual classes) from the test data.
  + **y\_pred**: The predicted labels from the model.
  + **labels=model.classes\_**: This ensures that the confusion matrix uses the class labels that the model was trained on. model.classes\_ gives the list of class labels the Naïve Bayes model was trained on (e.g., 'Setosa', 'Versicolor', 'Virginica' for the Iris dataset).
* **pd.DataFrame(cm, index=model.classes\_, columns=model.classes\_)**:
  + This converts the confusion matrix (cm), which is a 2D array, into a pandas DataFrame (cm\_df).
  + The index and columns of the DataFrame are set to the class labels (using model.classes\_), so it will clearly show which class corresponds to each row and column.

### 2. ****Plotting the Confusion Matrix****:

sns.heatmap(cm\_df, annot=True, cmap='Blues', fmt='d')

plt.title('Confusion Matrix')

plt.xlabel('Predicted')

plt.ylabel('Actual')

plt.show()

* **sns.heatmap(cm\_df, annot=True, cmap='Blues', fmt='d')**:
  + This line uses Seaborn's heatmap function to plot the confusion matrix as a heatmap.
  + **cm\_df**: The DataFrame containing the confusion matrix.
  + **annot=True**: This option annotates (labels) each cell in the heatmap with the numeric value (the number of predictions for that class).
  + **cmap='Blues'**: The color map used to represent the values. 'Blues' creates a blue color gradient, where higher values will be darker.
  + **fmt='d'**: This formats the annotations as integers (i.e., it will display whole numbers in the heatmap).
* **plt.title('Confusion Matrix')**:
  + This sets the title of the plot as "Confusion Matrix."
* **plt.xlabel('Predicted')**:
  + This labels the x-axis as "Predicted" to represent the predicted class labels.
* **plt.ylabel('Actual')**:
  + This labels the y-axis as "Actual" to represent the true class labels.
* **plt.show()**:
  + This displays the plot.

### Summary:

* The **confusion matrix** is calculated to evaluate how well the model performed in terms of predicting each class.
* The **heatmap** visualization of the confusion matrix provides an easy-to-read representation of the model's performance, highlighting how many predictions were correct (diagonal cells) and how many were incorrect (off-diagonal cells).

#### Key Points:

* The **diagonal** cells of the confusion matrix show the number of correct predictions for each class.
* The **off-diagonal** cells show the number of misclassifications between different classes.

This is an important step to assess model performance, especially for multi-class classification problems like the Iris dataset.

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

precision = precision\_score(y\_test, y\_pred, average='macro')

recall = recall\_score(y\_test, y\_pred, average='macro')

error\_rate = 1 - accuracy

print(f"Accuracy: {accuracy:.2f}")

print(f"Error Rate: {error\_rate:.2f}")

print(f"Precision (Macro): {precision:.2f}")

print(f"Recall (Macro): {recall:.2f}")

### -> Explanation of this part of the code:

### 1. ****Accuracy****:

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

* **accuracy\_score(y\_test, y\_pred)**: This function calculates the **accuracy** of the model, which is the ratio of correct predictions to the total number of predictions. It is a measure of how often the classifier is correct.
  + **y\_test**: True labels (actual values from the test set).
  + **y\_pred**: Predicted labels from the model.
* **Accuracy** is calculated as:

Accuracy=Number of Correct PredictionsTotal Number of Predictions\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}

* The result will be a value between 0 and 1, where 1 means 100% accuracy.

### 2. ****Precision****:

precision = precision\_score(y\_test, y\_pred, average='macro')

* **precision\_score(y\_test, y\_pred, average='macro')**: This function calculates **precision** for each class and then averages them. Precision is the ratio of correctly predicted positive observations to the total predicted positives. It is an important metric when you want to know how many of the positive predictions were actually correct.
  + **average='macro'**: This calculates the precision for each class independently and then averages them (macro-average).
  + The formula for precision for each class is:

Precision=TPTP+FP\text{Precision} = \frac{TP}{TP + FP}

where:

* + - **TP (True Positives)** is the number of correct positive predictions.
    - **FP (False Positives)** is the number of incorrect positive predictions.
* **Macro-averaging** means that each class's precision is treated equally (no weight based on class distribution).

### 3. ****Recall****:

recall = recall\_score(y\_test, y\_pred, average='macro')

* **recall\_score(y\_test, y\_pred, average='macro')**: This function calculates **recall** for each class and then averages them. Recall measures how many actual positive observations were correctly identified by the model.
  + **average='macro'**: This calculates the recall for each class independently and averages them (macro-average).
  + The formula for recall for each class is:

Recall=TPTP+FN\text{Recall} = \frac{TP}{TP + FN}

where:

* + - **TP (True Positives)** is the number of correct positive predictions.
    - **FN (False Negatives)** is the number of actual positive values that were incorrectly predicted as negative.
* **Macro-averaging** means that recall is calculated independently for each class and then averaged.

### 4. ****Error Rate****:

error\_rate = 1 - accuracy

* **error\_rate = 1 - accuracy**: The error rate is the proportion of incorrect predictions. It is simply the complement of the accuracy.
  + If accuracy is high (close to 1), the error rate will be low (close to 0).
  + The error rate is a measure of how often the model is incorrect.
  + This can be useful if you want to focus on reducing errors instead of maximizing accuracy.

VIVA questions:  
Here are some **basic questions** that can be asked in the viva for your **Naïve Bayes Classification** implementation on the **Iris dataset**, along with **detailed answers** in simple language:

### 1. ****What is the Naïve Bayes classification algorithm?****

* **Answer**: Naïve Bayes is a probabilistic machine learning algorithm used for classification tasks. It applies Bayes' Theorem with the "naïve" assumption that all features are independent of each other. In other words, it assumes that the presence of one feature doesn't affect the presence of other features. It's simple, fast, and effective for problems like spam classification or predicting species of plants like in the Iris dataset.

### 2. ****Why do we use**** StandardScaler ****in this code?****

* **Answer**: StandardScaler is used to scale the features so that they have a mean of 0 and a standard deviation of 1. This is important because many machine learning algorithms, including Naïve Bayes, perform better when the features are on the same scale. Without scaling, features with larger values could dominate the model, leading to inaccurate predictions.

### 3. ****What does the**** train\_test\_split ****function do?****

* **Answer**: The train\_test\_split function splits the dataset into two parts: training data and testing data. The model is trained on the training data and tested on the testing data. This ensures that we are evaluating the model on data it has never seen before, which gives a better idea of how it will perform on real-world data. In this code, 80% of the data is used for training and 20% for testing.

### 4. ****What does the Naïve Bayes model predict in this case?****

* **Answer**: The Naïve Bayes model in this case is predicting the species of the Iris plant based on its features (like sepal length, sepal width, petal length, and petal width). The model takes these features as input and classifies them into one of the three species: Setosa, Versicolor, or Virginica.

### 5. ****What is a confusion matrix, and what does it tell us?****

* **Answer**: A confusion matrix is a table used to evaluate the performance of a classification model. It compares the actual values (true labels) to the predicted values. The matrix has four parts:
  + **True Positives (TP)**: Correctly predicted positive cases.
  + **False Positives (FP)**: Incorrectly predicted positive cases.
  + **True Negatives (TN)**: Correctly predicted negative cases.
  + **False Negatives (FN)**: Incorrectly predicted negative cases. It helps to understand how well the model is doing in predicting each class.

### 6. ****What is the purpose of the**** sns.heatmap ****function?****

* **Answer**: The sns.heatmap function is used to visually display the confusion matrix in a color-coded format. The heatmap makes it easier to interpret the matrix by showing the number of correct and incorrect predictions for each class, with darker colors representing higher values.

### 7. ****What does**** accuracy\_score ****measure, and how is it calculated?****

* **Answer**: accuracy\_score measures how many predictions the model got right. It is calculated as the ratio of correct predictions to the total number of predictions:

Accuracy=TP+TNTP+TN+FP+FNAccuracy = \frac{TP + TN}{TP + TN + FP + FN}

It tells us the overall correctness of the model, i.e., how often the model predicted the correct class.

### 8. ****What is the error rate, and how do you calculate it?****

* **Answer**: The error rate is the percentage of incorrect predictions made by the model. It is calculated as:

Error Rate=1−AccuracyError\ Rate = 1 - Accuracy

A higher error rate means the model is making more incorrect predictions.

### 9. ****What is precision, and how is it different from recall?****

* **Answer**:
  + **Precision** is the percentage of true positive predictions among all positive predictions. It answers the question, "Of all the cases the model predicted as positive, how many were actually positive?" Precision is calculated as:

Precision=TPTP+FPPrecision = \frac{TP}{TP + FP}

* + **Recall** (also called sensitivity or true positive rate) is the percentage of actual positives that were correctly identified by the model. It answers the question, "Of all the actual positive cases, how many did the model correctly predict?" Recall is calculated as:

Recall=TPTP+FNRecall = \frac{TP}{TP + FN}

### 10. ****What does**** precision\_score ****and**** recall\_score ****with**** average='macro' ****mean?****

* **Answer**: When you set average='macro', the precision and recall scores are calculated for each class (in this case, for each Iris species) and then averaged. It treats all classes equally, regardless of how many instances belong to each class.

### 11. ****Why is Naïve Bayes called "naïve"?****

* **Answer**: Naïve Bayes is called "naïve" because it makes the assumption that all features are independent, which is often not true in real-world data. This assumption simplifies the computation of probabilities but may not always hold. Despite this, the algorithm performs surprisingly well in many situations.

### 12. ****What happens if Naïve Bayes assumptions do not hold true?****

* **Answer**: If the assumptions of Naïve Bayes (independence of features) do not hold, the model may not perform as well as expected. However, it is still a strong classifier, especially for text classification tasks like spam detection. In practice, it works well even when the independence assumption is violated to some extent.

### 13. ****What are some limitations of the Naïve Bayes classifier?****

* **Answer**: Some limitations of Naïve Bayes include:
  + It assumes that all features are independent, which is often not the case.
  + It may perform poorly if the features are highly correlated.
  + It is sensitive to irrelevant features, as the independence assumption doesn't hold if features are related.

### 14. ****How can you improve the performance of a Naïve Bayes model?****

* **Answer**: You can improve the performance of a Naïve Bayes model by:
  + Removing or transforming highly correlated features.
  + Using feature engineering to create new, more meaningful features.
  + Using more sophisticated variants of Naïve Bayes, like Multinomial or Bernoulli Naïve Bayes, depending on the type of data.

These are some of the **basic questions** that can be asked during the viva for the Naïve Bayes classification algorithm applied to the **Iris dataset**. These questions focus on the key aspects of the model, including its assumptions, performance metrics, and practical application.