REPORT AI LAB 6

Tạ Thị Phương Thảo
ITDSIU20082

1a)
This code generates a scatter plot of the Iris dataset of 150 samples (Setosa (label 0), Versicolor (label 1), and Virginica (label 2)) using the features at indices 0 and 1. Here's a breakdown of the code:

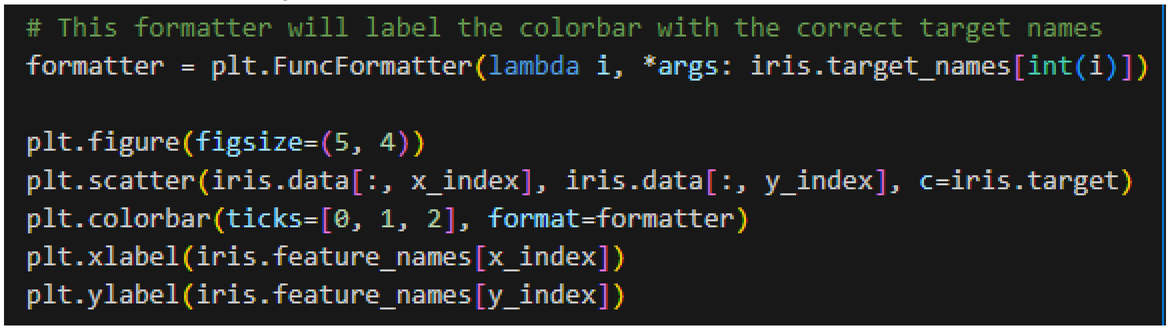
#Load dataset

- The code begins by importing the necessary libraries. The ‘load_iris’ function from ‘sklearn.datasets’ is imported to load the Iris dataset, and ‘matplotlib.pyplot’ is imported as ‘plt’ for data visualization.
- The Iris dataset is loaded using the ‘load_iris’ function, and the dataset is assigned to the variable ‘iris’.
  # The indices of the features that we are plotting
- The indices of the features to be plotted are set. In this case, ‘x_index’ is set to 0, and ‘y_index’ is set to 1. These indices correspond to the first and second features of the Iris dataset.

#Label the colorbar with the correct target names

- The formatter is a lambda function that takes an index ‘i’ and returns the corresponding target name from ‘iris.target_names’.
- A new figure is created with a size of 5 inches by 4 inches
- The scatter plot is created using ‘plt.scatter’. The x-coordinate of the plotted points is obtained by accessing ‘iris.data[:, x_index]’, which selects all rows and the column specified by ‘x_index’. The y-coordinate is obtained similarly using ‘iris.data[:, y_index]’. The color of each point is determined by ‘iris.target’, which represents the target labels of the dataset.
- A colorbar is added to the plot using ‘plt.colorbar’. The ‘ticks’ parameter specifies the values at which tick marks should be placed on the colorbar, and the ‘format’ parameter uses the previously defined formatter to label the tick marks with the corresponding target names.
- The x-axis label is set to ‘iris.feature_names[x_index]’, which retrieves the name of the feature at index ‘x_index’. Similarly, the y-axis label is set to

‘iris.feature_names[y_index]’:



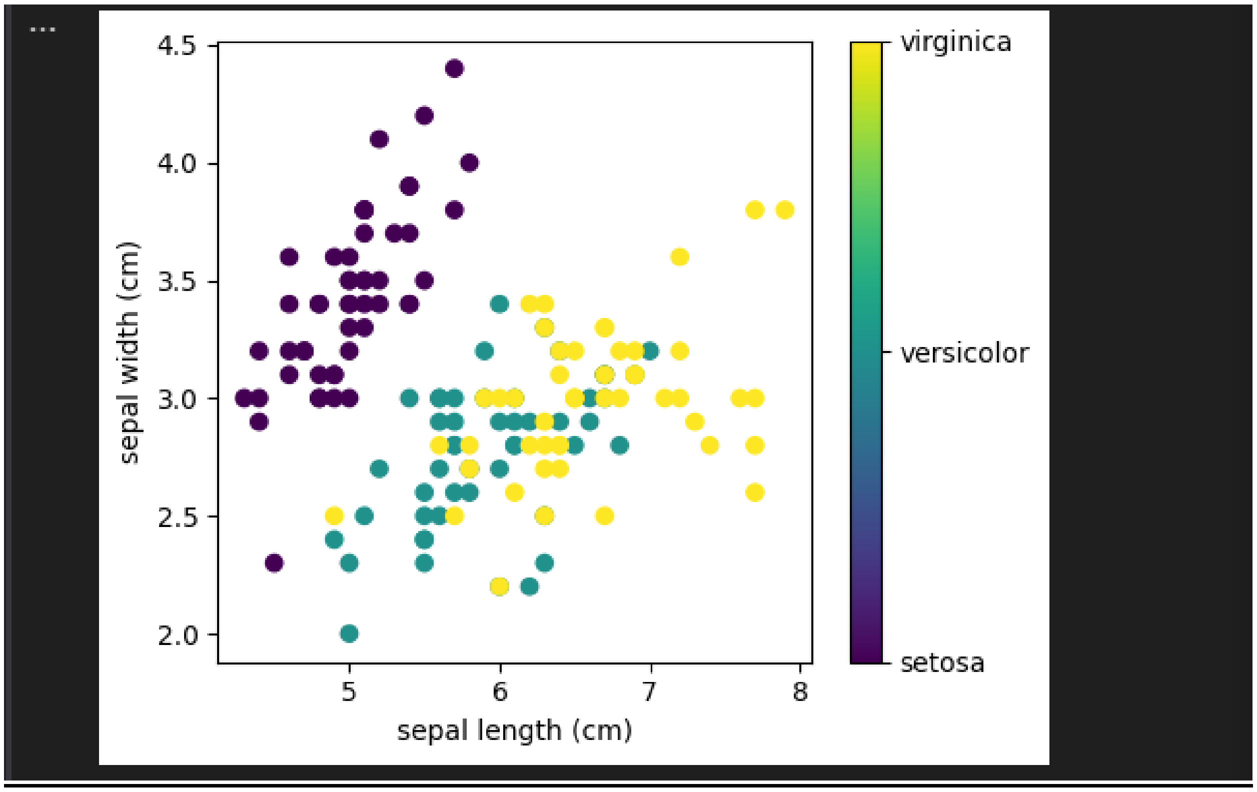
```python
# This formatter will label the colorbar with the correct target names
formatter = plt.FuncFormatter(lambda i, *args: iris.target_names[int(i)])

plt.figure(figsize=(5, 4))
plt.scatter(iris.data[:, x_index], iris.data[:, y_index], c=iris.target)
plt.colorbar(ticks=[0, 1, 2], format=formatter)
plt.xlabel(iris.feature_names[x_index])
plt.ylabel(iris.feature_names[y_index])
```

# Show the scatter plot

- The layout of the plot is adjusted to make sure all elements are properly displayed using ‘plt.tight_layout()’.
- Finally, the plot is displayed using ‘plt.show()’.

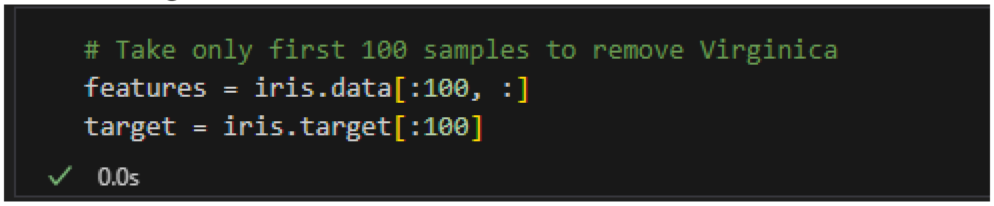
This code generates a scatter plot of the Iris dataset, with the x-axis representing the first feature, the y-axis representing the second feature, and different colors indicating different target classes. The colorbar provides a visual representation of the target classes, and the x-axis and y-axis labels provide additional context for the plotted data.

<u>Output:</u> Visualize the dataset of all 3 classes: Virginica, Versicolor and Setosa (150 samples):



1b) As same as 1a but slightly different, this code generates a scatter plot of the Iris dataset of 100 samples which contain 2 classes: Setosa and Versicolor:

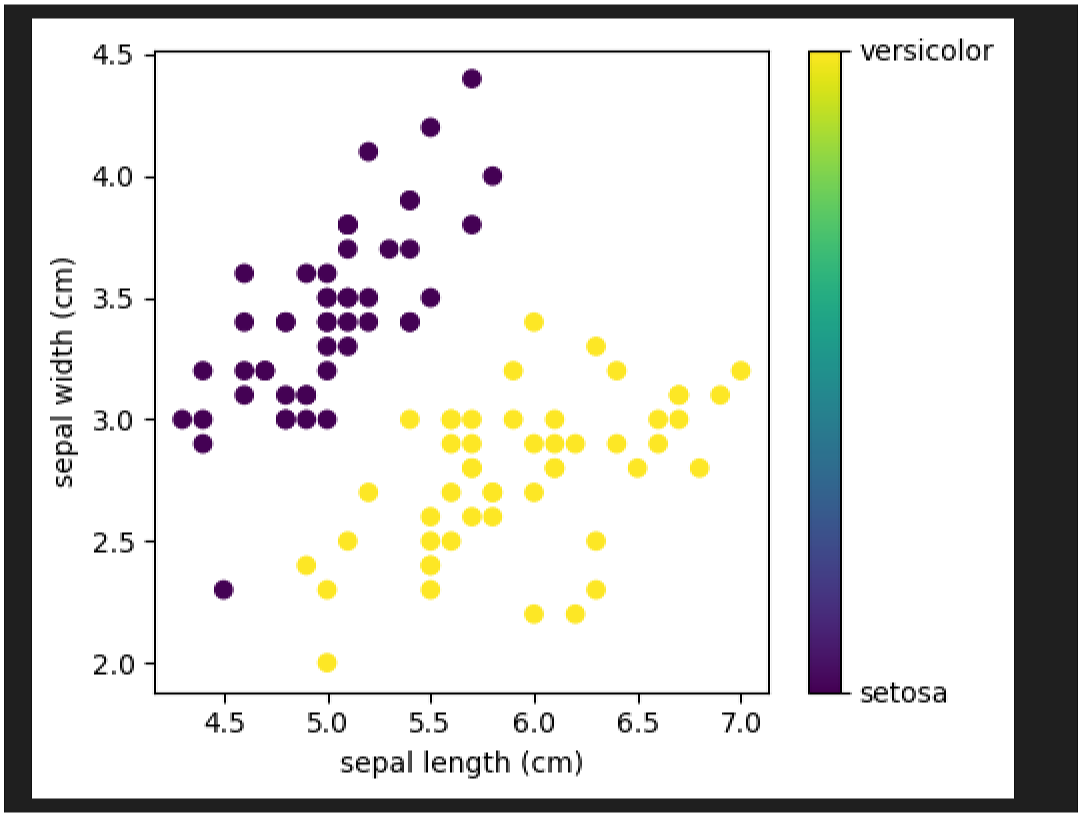
- First, we will need 100 first samples to remove Virginica by using 'iris.data' and 'iris.target':



```
# Take only first 100 samples to remove Virginica
features = iris.data[:100, :]
target = iris.target[:100]
```

- Due to its tiny different (100 sample instead of 150 samples), the code we use for **#Load dataset, #Label the colorbar with the correct target names** and **# Show the scatter plot** are the same to question 1a to analyze and visualize the dataset

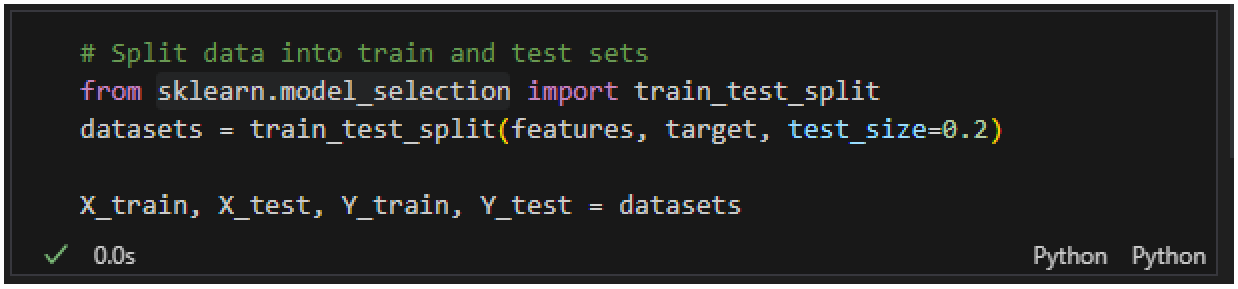
<u>Output:</u> Visualize the dataset of all 2 classes: Versicolor and Setosa (100 samples):



#Split the dataset to train and test sets:

- We use the function 'train_test_split' imported from 'sklearn.model_selection' module, which using to split the dataset into training and test sets.
- There are 3 arguments is called in this 'train_test_split' function:
    - 'features': input the features of the dataset
    - 'target': corresponding the target labels
    - 'test_size': this argument is set to 0.2, which means 20% of the data will be used for testing, while the remaining 80% will be used for training.
- After that, we assigned the variables of the 'datasets':
    - 'datasets' is a tuple that contains four elements: X_train, X_test, Y_train, and Y_test.
    - X_train: the training data features.
    - X_test: the test data features.
    - Y_train: the corresponding training data labels (outputs).
    - Y_test: the corresponding test data labels (outputs).

The training set is used to train a model, while the test set is used to evaluate the performance of the trained model.



```
# Split data into train and test sets
from sklearn.model_selection import train_test_split
datasets = train_test_split(features, target, test_size=0.2)

X_train, X_test, Y_train, Y_test = datasets
```

✓ 0.0s Python Python

#Make Perceptron class

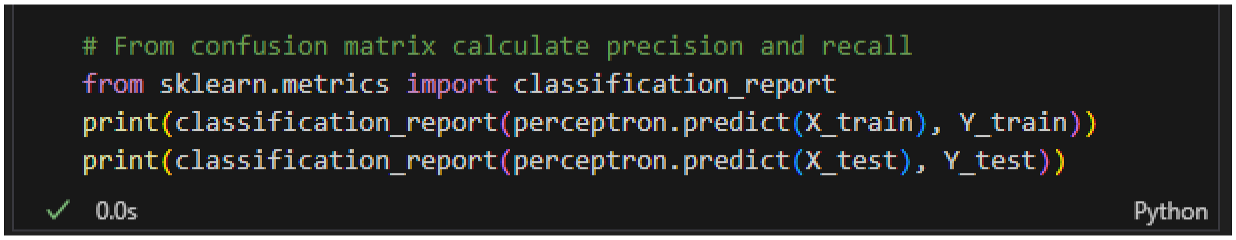
After splitting the data into training and test sets, we will create a class called Perceptron which applies the Perceptron Algorithm to train and make predictions for all epochs in the data, the functions work on this class are:

- __init__ (Constructor): weight and bias = none (indicates that the Perceptron model has not been trained yet. These variables will be updated with appropriate values during the training process)
- Model: calculates the dot product of the weights and the input vector, and applies a thresholding operation based on the bias to make a binary prediction (0 or 1)
- Predict: predictor to predict on the data based on weight
- Fit: the function returns the weight matrix as a NumPy array, which contains the weights for each epoch.

This class allows you to train a Perceptron model, make predictions, and retrieve the weight matrix and accuracy scores during training.

#Calculate precision and recall

- From the 'sklearn.metrics' module, we can use the 'classification_report' function from it to calculate the precision and recall scores based on the confusion matrix, which includes precision, recall, F1-score, and other metrics, based on the predicted and true labels:

  ```
  # From confusion matrix calculate precision and recall
  from sklearn.metrics import classification_report
  print(classification_report(perceptron.predict(X_train), Y_train))
  print(classification_report(perceptron.predict(X_test), Y_test))
  ```

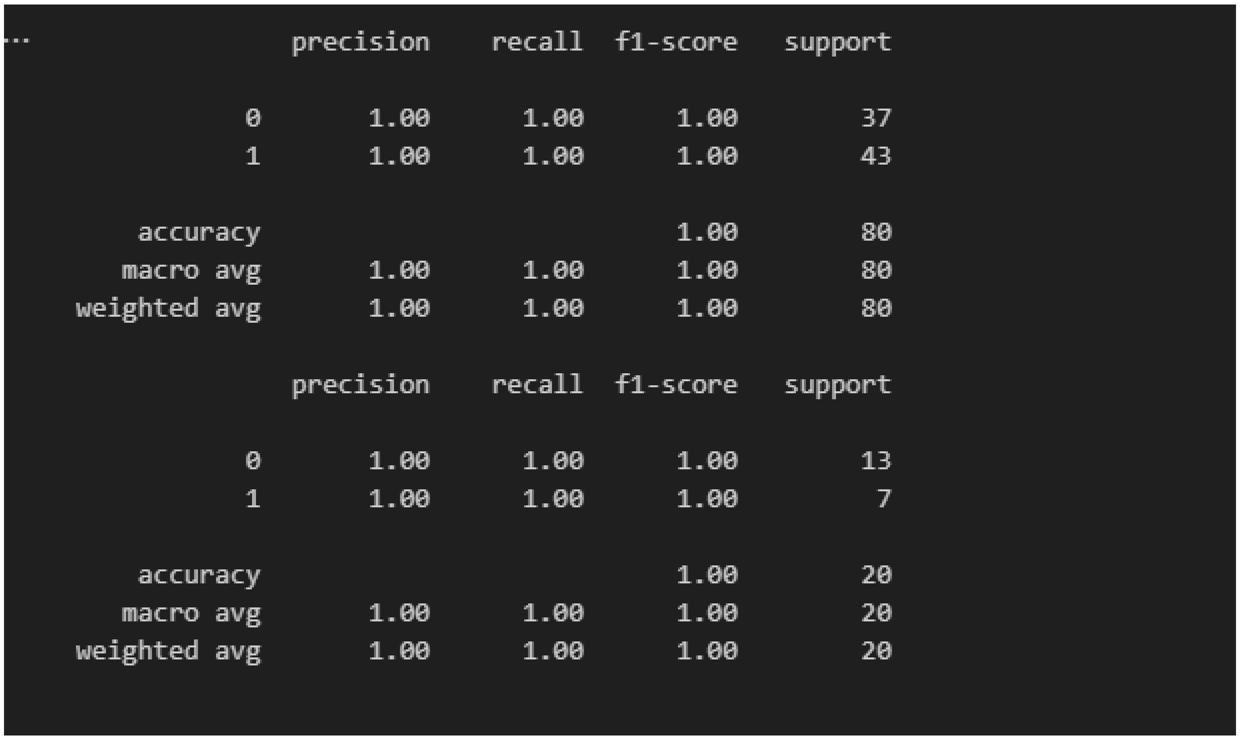
  ✓ 0.0s Python

- The first print statement calculates and displays the classification report for the predictions made by the perceptron model on the training data (X_train) and compares them with the true labels (Y_train).
  - perceptron.predict(X_train): predicted labels for the training data using the predict() method.

    - Y_train: the true labels of the training data.
- The second print statement calculates and displays the classification report for the predictions made by the perceptron model on the test data (X_test) and compares them with the true labels (Y_test).
    - perceptron.predict(X_test): predicted labels for the test data using the predict() method.
    - Y_test: the true labels of the test data.

This classification can help evaluate the performance of the perceptron model by providing insights into how well it predicts each class.

Output:



```
...               precision    recall  f1-score   support

           0       1.00      1.00      1.00        37
           1       1.00      1.00      1.00        43

    accuracy                           1.00        80
   macro avg       1.00      1.00      1.00        80
weighted avg       1.00      1.00      1.00        80

              precision    recall  f1-score   support

           0       1.00      1.00      1.00        13
           1       1.00      1.00      1.00         7

    accuracy                           1.00        20
   macro avg       1.00      1.00      1.00        20
weighted avg       1.00      1.00      1.00        20
```