Introduction

Frank Rosenblatt first presented the Perceptron, a linear binary classifier, in 1958. It is a single-layer neural network that gains the ability to categorize incoming data into two groups using a set of repeatedly updated weights and biases. A supervised learning method called the Perceptron algorithm is very effective at handling categorization issues including fraud detection, spam filtering, and picture recognition.

This program uses NumPy, a potent library for numerical computing, to implement the Perceptron algorithm in Python. Fit() and Predict() are two of the methods in the Perceptron class. The predict() method makes predictions based on the trained weights and biases, whereas the fit() method trains the Perceptron on a specified set of input data and labels.

Methodology

A set of inputs is given to the Perceptron algorithm, which multiplies them by appropriate weights before adding them all up. The Perceptron produces a positive output if the sum exceeds a predetermined threshold and a negative output otherwise. This procedure is repeated until the algorithm finds the proper weights and biases for correctly classifying the input data.

The following arguments are taken by the fit() method:

1. X: A numpy.ndarray containing the input data, where each column is a feature and each row is an observation.

2. y: a numpy.ndarray with values of -1 or 1 that represents the labels for the incoming data.

3. The perceptron algorithm's learning rate is represented by the float learning rate. It regulates how much the weights are changed throughout each training epoch.

4. The maximum number of epochs that can be used to train a perceptron is indicated by the integer num epochs. A single pass through the whole training set constitutes an epoch.

5. stop criterion: a float indicating the smallest weight difference between epochs necessary for the training to end. The training is said to have converged if the change in weights is less than this amount.

The weights are initialized randomly and updated iteratively using the fit() method until the algorithm finds a set of weights that accurately categorizes the input data. The weights are adjusted throughout the training phase by iteratively looping through each observation in the input data, computing the weighted sum of the inputs and the current set of weights, and comparing the predicted and actual labels' errors. Once

the maximum number of epochs is achieved or the change in weights reaches the stop threshold, the process keeps going.

The activation function used in the perceptron is the Sign function which gives us an output of 1 if the input of the function is positive and an output of -1 if the input of the function is non-positive.

$$sign(x) = \begin{cases} 1 \text{ if } x > 0 \\ -1 \text{ if } x < 0 \end{cases}$$

The formula to update the weights if the output of the sign function is not equal to the label for the given features after giving it the weighted sum of the features is:

$$w(t+1) = w(t) + learning\_rate * x(w) * error$$

$w(t+1)$ = updated weight
$w(t)$ = current weight
$x(w)$ = the feature which corresponds to the weight
$error$ = expected label - predicted label

The following argument is accepted by the predict() method:

1. X: A numpy.ndarray containing the input data, where each column is a feature and each row is an observation.

The predict() method returns a numpy.ndarray with values of -1 or 1 that represents the predicted labels for the input data. The weighted total of the inputs is calculated using the training biases and weights, and the technique then returns the sign of the sum as the predicted label.

Example 1

```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Generate a random binary classification dataset
X, y = make_classification(n_samples=1000, n_features=5, n_classes=2, random_state=42)
y[y==0] = -1
# Split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create a Perceptron object and train it on the training set
perceptron = Perceptron(bias_input=1)
perceptron.fit(X_train, y_train, learning_rate=0.1, num_epochs=1000, stop_criterion=0.01)

# Use the trained model to make predictions on the test set
y_pred = perceptron.predict(X_test)

# Evaluate the accuracy of the predictions
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

```
26]  ✓ 2.2s

Accuracy: 0.775
```

In this example, we first create a random binary classification dataset with 1000 samples, 5 features, and 2 classes using the Scikit-learn make classification function. The train test split method was then used to divide the dataset into training and test sets.

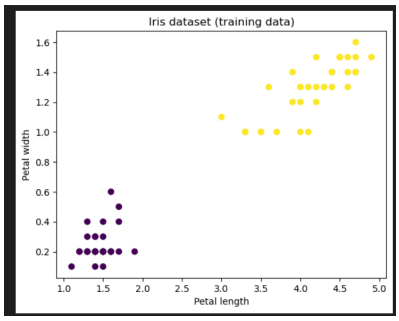
Then, with a learning rate of 0.1, a maximum of 1000 epochs, and a stop criteria of 0.01 on the training set, we create a Perceptron object with a bias input of 1, train it, and then test it. This indicates that if the difference in weights across epochs is smaller than 0.01 the training will cease.

When we see the Accuracy on the console it gave us 0.775 success.

Example 2:

We'll train a Perceptron in the example that follows to categorize iris flowers according to the size and length of their petals. We will only choose the petal length and width features when loading the iris dataset from scikit-learn.

When we plot the data set we got



It is a linearly separable dataset after applying training to our perceptron model we got 100% accuracy.

```
perceptron = Perceptron()
perceptron.fit(X_train, y_train, learning_rate=0.1, num_epochs=1000, stop_criterion=0.01

# Predict the labels of the testing data
y_pred = perceptron.predict(X_test)

# Calculate the accuracy of the Perceptron on the testing data
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
```

✓ 0.0s

```
Accuracy: 1.00
```

Conclusion

The Perceptron is a straightforward yet effective technique for binary classification applications, to sum up. It operates by locating the ideal decision border in the feature space that divides the two classes. The technique is ideal for large-scale datasets since it is quick and effective. The Perceptron does have certain limitations, though. It may not be able to handle datasets that call for more sophisticated choice boundaries because, for example, it can only learn linear decision boundaries. In addition, if the classes are not linearly separable, the algorithm might not converge.

Despite these drawbacks, the Perceptron method is still a cornerstone of machine learning. Because of its simplicity, it is simple to comprehend and use, and it may be used as a building block for more complicated algorithms like neural networks. The Perceptron is an effective tool for binary classification jobs and can be an excellent place to look into other machine learning techniques.