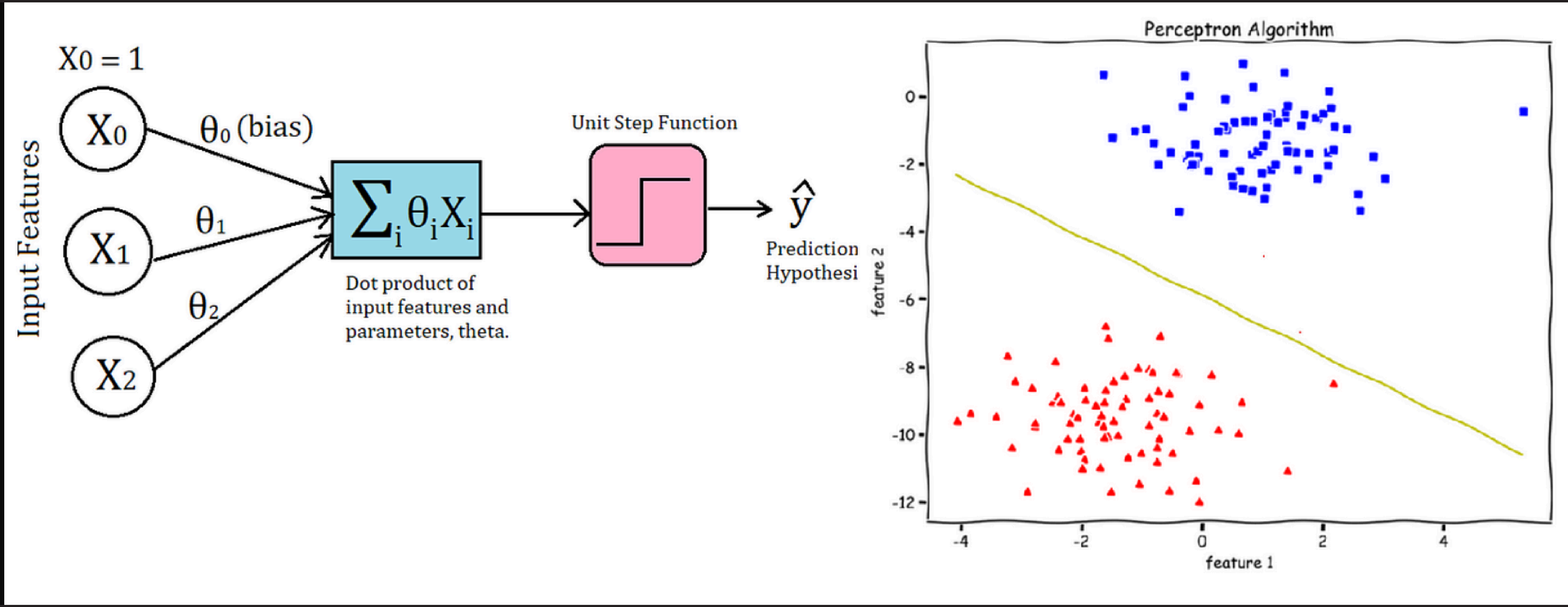
PERCEPTRON

200101005
YUSUF ÖMER TURSUN



In this homework, I have coded Perceptron algorithm and in my report, I will be explaining all the functions I have written for this algorithm.

CONSRUCTOR

```
class Perceptron:
    Tabnine | Edit | Test | Explain | Document | Ask
    def __init__(self, filename, learning_rate=0.01, num_iterations=500):

        self.filename = filename
        self.learning_rate = learning_rate
        self.num_iterations = num_iterations
        self.teta = None  # Initial teta values will be 0 in the calculated_values function.
```

CONSTRUCTOR

The first function, __init__ is a building block for the class used for initializing the Perceptron class. The work of this function is initialization of the Perceptron model and configuration for some basic parameters.

Parameters

filename: This is a parameter that shows the name of the file from which the model reads the training data.

learning_rate: It indicates the learning rate of the model. The greater the value of the parameter, the more the theta values are altered in every iteration. A low value of the parameter can be used for slow but more stable learning, while a high value would give fast learning at the risk of instability during training. Default value is 0.01.

num_iterations: Number of iterations in which the values of theta are updated. It tells how many iterations model is trained for. Default is 500 and can iterate that much of times. Increasing this will allow the model to train on more data. Increasing the value too high will take unnecessary time.

self.theta: This is an attribute that holds the values of theta. By default, theta does not have any value, but once the function calculated_values() is executed, initial theta values are created as zero and then updated during training.

FUNCTION FOR READING TRAIN DATAS

```python
def read_train_datas(self):
    # I am reading the training data file
    train_data = np.array(pd.read_excel(self.filename, sheet_name="TRAINData"), dtype=np.int64)
    # I do not include the column SubjectID
    train_data = train_data[:, 1:]
    # Now bias terms should be added
    ones_column = np.ones((train_data.shape[0], 1))
    train_data = np.hstack((ones_column, train_data))
    return train_data
```

read_train_datas()

1 - train_data = np.array(pd.read_excel(self.filename, sheet_name="TRAINData"), dtype=np.int64)

1 - This line reads the file containing the training data with the specified filename parameter (self.filename). Here we assume that we are reading the “TRAINData” sheet in the Excel file. With the Pandas library, the Excel file is read and then converted to a NumPy array (np.array) and all data is converted to the np.int64 data type.

2 - train_data = train_data[:, 1:]

2 - In this row, I have removed the first column of the read data, SubjectID, because this column is not included in the attributes needed for the model to learn. Here I used NumPy slicing to get all the data except the first column.

3 -ones_column = np.ones((train_data.shape[0], 1))
 train_data = np.hstack((ones_column, train_data))

3- It is important to add a bias term in the Perceptron algorithm. This is necessary for the model to be able to perform linear classification.

FUNCTION FOR READING TEST DATAS

```python
def read_test_datas(self):
    # I am reading the test data file
    test_data = np.array(pd.read_excel(self.filename, sheet_name="TESTData"), dtype=np.int64)
    # I do not include the last column in test data. Because all the values are NaN values. Also the first column is excluded
    test_data = test_data[:, 1:-1]
    # Now bias terms should be added
    ones_column = np.ones((test_data.shape[0], 1))
    test_data = np.hstack((ones_column, test_data))
    return test_data
```

read_test_datas()

```
1 - test_data = np.array(pd.read_excel(self.filename, sheet_name="TESTData"), dtype=np.int64)
```

1 - This line reads the file (filename) containing the test data. Here the “TESTData” sheet from the Excel file is retrieved.
Pandas reads the Excel file and then converts it into a NumPy array (np.array). This data is completely converted to the np.int64 data type, i.e. the test data must be in numeric format.

```
2 - test_data = test_data[:, 1:-1]
```

2 - The last column in the test dataset is removed. This column contains NaN (empty) values, which are redundant in the learning phase of the model. The first column is also removed, because it contains SubjectID and is redundant for classification.

```
3 - ones_column = np.ones((test_data.shape[0], 1))
test_data = np.hstack((ones_column, test_data))
```

3 - In the Perceptron algorithm it is necessary to add a bias term. Bias allows the model to perform linear classification.

FUNCTION FOR CALCULATING THE VECTOREL PRODUCT OF DATAS AND THETAS

```python
def calculated_values(self):
    # I am calculating the dot product of training examples and weights
    training_examples = self.read_train_datas()
    self.teta = np.zeros(training_examples.shape[1] - 1)  # Initializing theta weights

    for _ in range(self.num_iterations):
        for i in range(training_examples.shape[0]):
            value = 0
            for j in range(training_examples.shape[1] - 1):  # I don't include the class values
                value += training_examples[i][j] * self.teta[j]

            class_value = training_examples[i][-1]  # class value of dataset
            if class_value != self.control_statement(value):
                delta = class_value - self.control_statement(value)
                self.change_theta_values(training_examples[i], delta)

    print("Training completed. Learned theta values:", self.teta)
```

calculated_values()

calculated_values method of the Perceptron class is responsible for training the perceptron model using the provided training data. The algorithm used here is a simple perceptron algorithm with a binary classification problem.

1. The method initializes the teta weights to zeros, which represent the parameters of the linear function used for classification. The number of teta weights is equal to the number of features in the training data minus one (since we don't include the class values).
2. The outer loop runs for a specified number of iterations (num_iterations). This is to ensure that the model converges to a stable solution.
3. The inner loop iterates over each training example. For each example, the algorithm calculates the dot product of the example's features and the current teta weights. This value represents the predicted class label based on the current model.
4. The predicted class label is compared with the actual class label of the training example. If they are not equal, it means that the model made an error in its prediction.
5. If an error was made, the algorithm calculates the delta value, which is the difference between the actual class label and the predicted class label. This delta value is used to update the teta weights.
6. The change_theta_values method is called to update the teta weights based on the learning rate and the delta value. The update is performed by adding the product of the learning rate, the feature value, and the delta value to the corresponding teta weight.
7. After all training examples have been processed for a given iteration, the algorithm prints a message indicating that training has completed, along with the learned teta values.

This process is repeated for a specified number of iterations, allowing the model to learn from the training data and improve its accuracy over time

FUNCTION FOR CONTROLLING THE THETA VALUES' UPDATE

```python
def control_statement(self, value):
    # I am using a simple thresholding function
    # If value is greater than or equal to 0, then class label is 4
    if value >= 0:
        return 4
    # If value is less than 0, then class label is 2
    else:
        return 2
```

control_statement()

This method, control_statement, is a crucial part of the Perceptron algorithm. It acts as the activation function or decision boundary for the perceptron.

1.
The method takes a single parameter value, which is typically the dot product of the input features and the perceptron's teta values(including the bias term).
2.
It implements a simple thresholding function:
If the input value is greater than or equal to 0, it returns the class label 4.
If the input value is less than 0, it returns the class label 2.
3.
This function essentially divides the input space into two regions:
The region where the weighted sum of inputs is non-negative (≥ 0) is classified as class 4.
The region where the weighted sum of inputs is negative (< 0) is classified as class 2.

This simple thresholding mechanism is what allows the perceptron to make binary classifications based on its learned weights. The learning process (implemented elsewhere in the class) adjusts the teta values to position this decision boundary optimally for the training data.

FUNCTION FOR UPDATING THE THETA VALUES

```
def update_theta_values(self, training_example, delta):
    # I am updating the weights of theta values based on learning rate and delta valu
    for i in range(len(self.teta)):
        self.teta[i] += self.learning_rate * training_example[i] * delta
```

update_theta_values()

This function updates the theta (weight) values of the model in the Perceptron algorithm. The weight update is done as follows:

For each weight, the weights are updated using the learning rate (learning_rate), the training sample (training_example) and the error (delta).
The delta represents the error made by the model, which will correct the weights to reach the correct class.
For each weight:
theta[i] is updated as follows, theta[i] += learning_rate * training_example[i] * delta.
=X, or in other words, this function ensures that the weights shift in the right direction after each incorrect prediction in the model's learning process.

FUNCTION FOR PREDICTING THE CLASS VALUES

```python
def predict(self, test_data):
    # I am predicting the class labels for the test data without using np.dot
    predictions = []
    for example in test_data:
        value = 0
        # I am calculating the dot product of test example and theta
        for i in range(len(self.teta)):
            value += self.teta[i] * example[i]
        predictions.append(self.control_statement(value))
    return predictions
```

predict()

This function predicts the test data and returns the class label for each test instance.

It sums over the test data for every instance, multiplying each feature by theta weights. In fact, it is the calculation of the dot product, but here I did it manually.
This cumulative value is then fed to the controlstatement function, which does class prediction. If this value is above zero, it assigns class 4; otherwise, if it is below zero, it assigns class 2.
The predicted class is appended to the predictions list and this list is returned.
Simply put, this function predicts which class every test sample should be classified to based on the learned theta values.

FUNCTION FOR WRITING CLASS VALUES OF THE TEST DATAS

```python
def write_class_values_of_test_data(self, predicted_class_values):
    # I load the original test data from the specified Excel file
    test_data = pd.read_excel(self.filename, sheet_name='TESTData')

    # I add the predicted values to a new column or replace the existing 'Class' column
    test_data['Class'] = predicted_class_values

    # I write the updated data back to the same Excel file
    with pd.ExcelWriter(self.filename, engine='openpyxl', mode='a', if_sheet_exists='replace') as writer:
        test_data.to_excel(writer, sheet_name='TESTData', index=False)
```

write_class_values_of_test_data()

This function will write predicted class values of test data into the same Excel file.

1. First, this method reads the original test data from the given Excel file by using the pd.read_excel function of pandas. We use the sheet_name='TESTData' parameter to read the corresponding sheet.
2. The forecasted class values are added as a new column to the loaded test data, replacing the existing column 'Class'. This is done by assigning the predicted_class_values list to the column 'Class' of the test_data DataFrame.
3. Now write the updated data into that same Excel file using the pd.ExcelWriter function. Change the engine parameter to 'openpyxl' so that the writing can be done into an already existing Excel file. Change the mode parameter to 'a' to allow appending of the new data into the existing file. Change if_sheet_exists parameter to 'replace' to overwrite the existing 'TESTData' sheet.
4. Finally, the updated DataFrame is written into the Excel file through the to_excel method of the ExcelWriter object. The sheet_name parameter has been set as 'TESTData' for naming the sheet, while the index parameter is False to exclude the index column in the output.

This way, the predicted class values will be kept in the same Excel file for further analysis or usage.

MAIN FUNCTION FOR TESTING THE MODEL

```python
def main():

    filename = "DataForPerceptron.xlsx"

    # I am creating an instance of the class Perceptron
    perceptron = Perceptron(filename)

    # Here I am producting the teta values and training datas.
    perceptron.calculated_values()

    # I have done my model. Now I am testing my model with test datas
    test_data = perceptron.read_test_datas()
    predictions = perceptron.predict(test_data)

    # I write the predicted class values to the Excel file
    perceptron.write_class_values_of_test_data(predictions)

    for i in range(len(predictions)):
        print("SubjectID ",550+i+1,": ", predictions[i])


if __name__ == "__main__":
    main()
```

main()

1. The function starts by defining the filename of the Excel file containing the data: "DataForPerceptron.xlsx".

2. An instance of the Perceptron class is created with the given filename.

3. perceptron.calculated_values() is called. This method trains the perceptron by calculating and updating the theta (weight) values using the training data.

4. After training, the code proceeds to test the model:
perceptron.read_test_datas() reads the test data from the Excel file.
perceptron.predict(test_data) uses the trained model to make predictions on the test data.

5. The predicted class values are then written back to the Excel file using perceptron.write_class_values_of_test_data(predictions).

6. Finally, there's a loop that prints out the predictions for each subject:
It iterates through the predictions list.
For each prediction, it prints the SubjectID (starting from 551 and incrementing) along with the predicted class value.

This main function essentially orchestrates the entire process of training the perceptron, making predictions on test data, saving those predictions, and displaying the results. It demonstrates the typical workflow of a machine learning task: load data, train model, make predictions, and output results.

OUTPUTS OF THE CODE - CLASS VALUES FOR TEST DATAS - SCREENSHOT

```
SubjectID 551 : 4
SubjectID 552 : 2
SubjectID 553 : 2
SubjectID 554 : 4
SubjectID 555 : 4
SubjectID 556 : 4
SubjectID 557 : 4
SubjectID 558 : 2
SubjectID 559 : 2
SubjectID 560 : 4
SubjectID 561 : 2
SubjectID 562 : 2
SubjectID 563 : 2
SubjectID 564 : 2
SubjectID 565 : 2
SubjectID 566 : 2
SubjectID 567 : 4
SubjectID 568 : 4
SubjectID 569 : 2
SubjectID 570 : 2
SubjectID 571 : 2
SubjectID 572 : 4
SubjectID 573 : 2
SubjectID 574 : 4
SubjectID 575 : 2
SubjectID 576 : 4
SubjectID 577 : 4
SubjectID 578 : 4
SubjectID 579 : 2
SubjectID 580 : 4
SubjectID 581 : 2
SubjectID 582 : 2
SubjectID 583 : 2
SubjectID 584 : 2
SubjectID 585 : 2
SubjectID 586 : 2
SubjectID 587 : 2
SubjectID 588 : 2
SubjectID 589 : 2
SubjectID 590 : 4
SubjectID 591 : 4
SubjectID 592 : 2
SubjectID 593 : 2
SubjectID 594 : 4
SubjectID 595 : 2
SubjectID 596 : 4
SubjectID 597 : 4
SubjectID 598 : 4
SubjectID 599 : 2
SubjectID 600 : 2
SubjectID 601 : 2
SubjectID 602 : 2
SubjectID 603 : 2
SubjectID 604 : 2
SubjectID 605 : 2
SubjectID 606 : 2
SubjectID 607 : 2
SubjectID 608 : 2
SubjectID 609 : 2
SubjectID 610 : 2
SubjectID 611 : 4
SubjectID 612 : 2
SubjectID 613 : 2
SubjectID 614 : 2
SubjectID 615 : 2
SubjectID 616 : 2
SubjectID 617 : 2
SubjectID 618 : 4
SubjectID 619 : 2
SubjectID 620 : 2
SubjectID 621 : 4
SubjectID 622 : 2
SubjectID 623 : 2
SubjectID 624 : 2
SubjectID 625 : 2
SubjectID 626 : 2
SubjectID 627 : 2
SubjectID 628 : 2
SubjectID 629 : 2
SubjectID 630 : 2
SubjectID 631 : 2
SubjectID 632 : 2
SubjectID 633 : 4
SubjectID 634 : 2
SubjectID 635 : 2
SubjectID 636 : 2
SubjectID 637 : 2
SubjectID 638 : 2
SubjectID 639 : 2
SubjectID 640 : 2
SubjectID 641 : 2
SubjectID 642 : 2
SubjectID 643 : 4
SubjectID 644 : 2
SubjectID 645 : 2
SubjectID 646 : 2
SubjectID 647 : 2
SubjectID 648 : 2
SubjectID 649 : 2
SubjectID 650 : 2
SubjectID 651 : 2
SubjectID 652 : 2
SubjectID 653 : 4
SubjectID 654 : 4
SubjectID 655 : 4
SubjectID 656 : 2
SubjectID 657 : 2
SubjectID 658 : 2
SubjectID 659 : 2
SubjectID 660 : 2
SubjectID 661 : 2
SubjectID 662 : 2
SubjectID 663 : 2
SubjectID 664 : 2
SubjectID 665 : 4
SubjectID 666 : 4
SubjectID 667 : 2
SubjectID 668 : 2
SubjectID 669 : 2
SubjectID 670 : 2
SubjectID 671 : 2
SubjectID 672 : 2
SubjectID 673 : 2
SubjectID 674 : 2
SubjectID 675 : 2
SubjectID 676 : 4
SubjectID 677 : 2
SubjectID 678 : 2
SubjectID 679 : 2
SubjectID 680 : 2
SubjectID 681 : 4
SubjectID 682 : 4
SubjectID 683 : 4
```

CONCLUSION

In my report, I shared the predictions of my model as a screenshot.
I will include the python file with my code in the assignment file, so you will be able to see the output by running my code.
In addition, I filled the Class column of the test data in the dataset with the model predictions, so at first the Class column of the TESTData page will be empty, but when you check the TESTData page after running the code, you will see the Class column full. I did this in the write_class_values_of_test_data() function.
So you will be able to see the Class column values in 3 places
1 - In my report
2 - Consol Screen
3 - in xlsx file