HW1

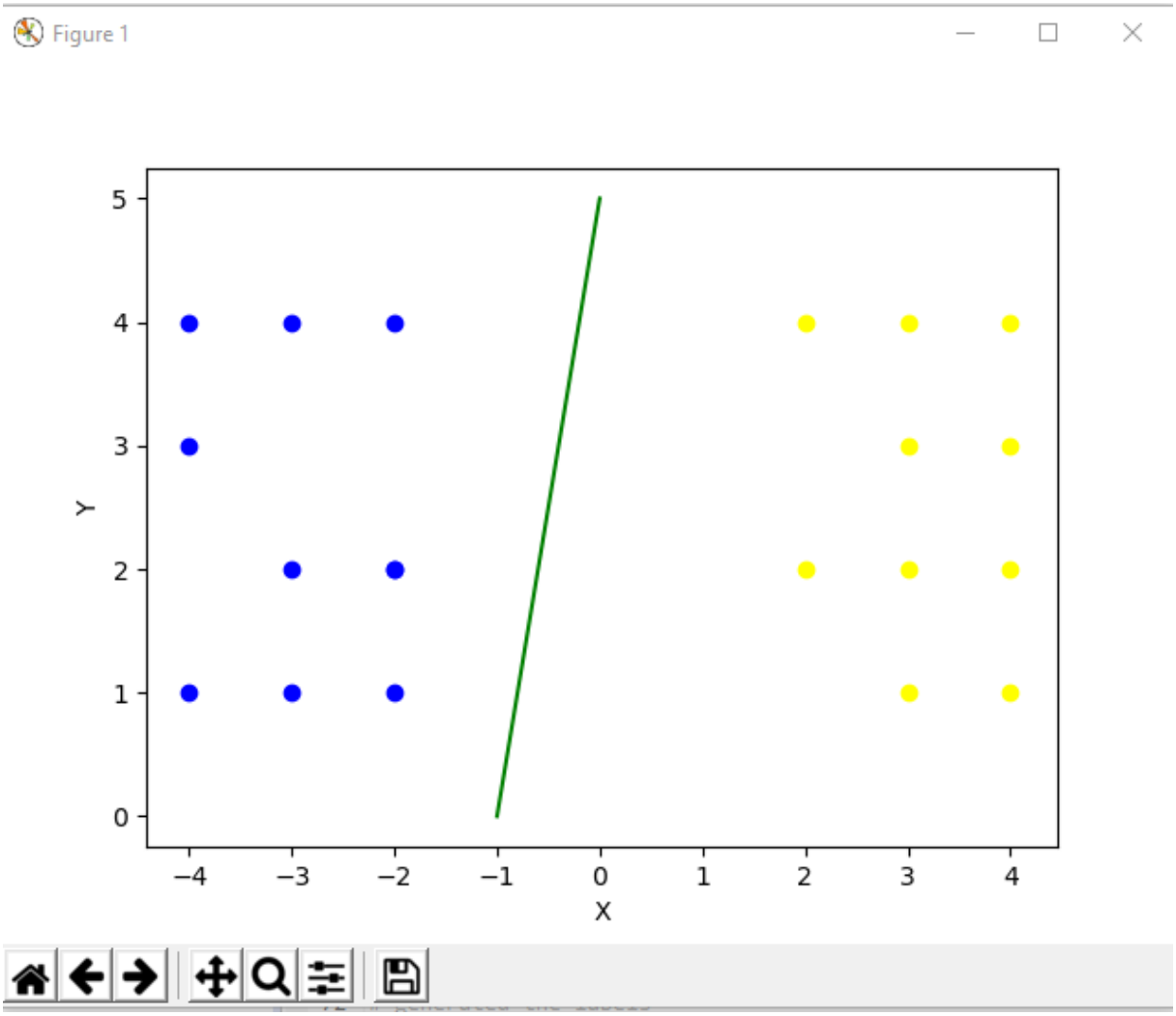
ELİF HANGÜL

Master In Intelligent Interactive Systems – Machine Learning

(a) Below is the code to generate a data set size of 20 and the target function f (in green)

```
# Question part a
# Generate a dataset of size 20. Plot the examples[(xn,yn)] as well as the target
function f on a plane.

# target array for 20 input. Mapping -1 or 1. For simplicity half the points map
to -1 and other half maps to 1
targets = np.array([-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1,1,1,1,1,1,1,1,1,1])
# generate sample data. The points that maps to -1 are denoted by neg as in
negative while the points map to 1 denoted by pos as in positive
neg = np.array([
    [-2,4],
    [-4,3],
    [-3,2],
    [-2,1],
    [-4,4],
    [-2,2],
    [-3,4],
    [-3,1],
    [-4,1],
    [-2,2],])
pos = np.array([

    [2,4],
    [3,4],
    [4,3],
    [4,4],
    [3,2],
    [2,2],
    [3,3],
    [4,2],
    [4, 1],
    [3, 1]])

# Prepare input data by combining negative and positive points
X = np.concatenate((neg,pos))
# assign targets array to y value
y=targets
# x_length is the length of X array in this case 20
x_length= len(X)

# Plotting the points. As by design there are 10 negative and 10 positive points.
for s, sample in enumerate(X):
# Plot the negative samples with color blue

    if s < (x_length/2):
        plt.scatter(sample[0], sample[1], c="blue")

    # Plot the positive samples with color yellow
    else:
        plt.scatter(sample[0], sample[1], c="yellow")
```

```
plt.xlabel('X')
plt.ylabel('Y')
# The target function f
plt.plot([-1,0],[0,5], "g")
# Part a is finished.
       plt.show()
```



(b) Below, we plot the training data. The target function f (in green) and hypothesis g is generated by PLA.

```
trained_model = p.train(X, y)

# use mlxtend library plot_decision_regions feature to plot the trained_model,
assign the label and show them
plot_decision_regions(X, y.astype(np.integer), clf=trained_model)
plt.title('Perceptron')
plt.xlabel('X1')
plt.ylabel('X2')
# A line that separates the points is found and market with a green color
plt.plot([-1,0],[0,5], "g")
plt.show()
# Print the number of updates it takes to converge
```

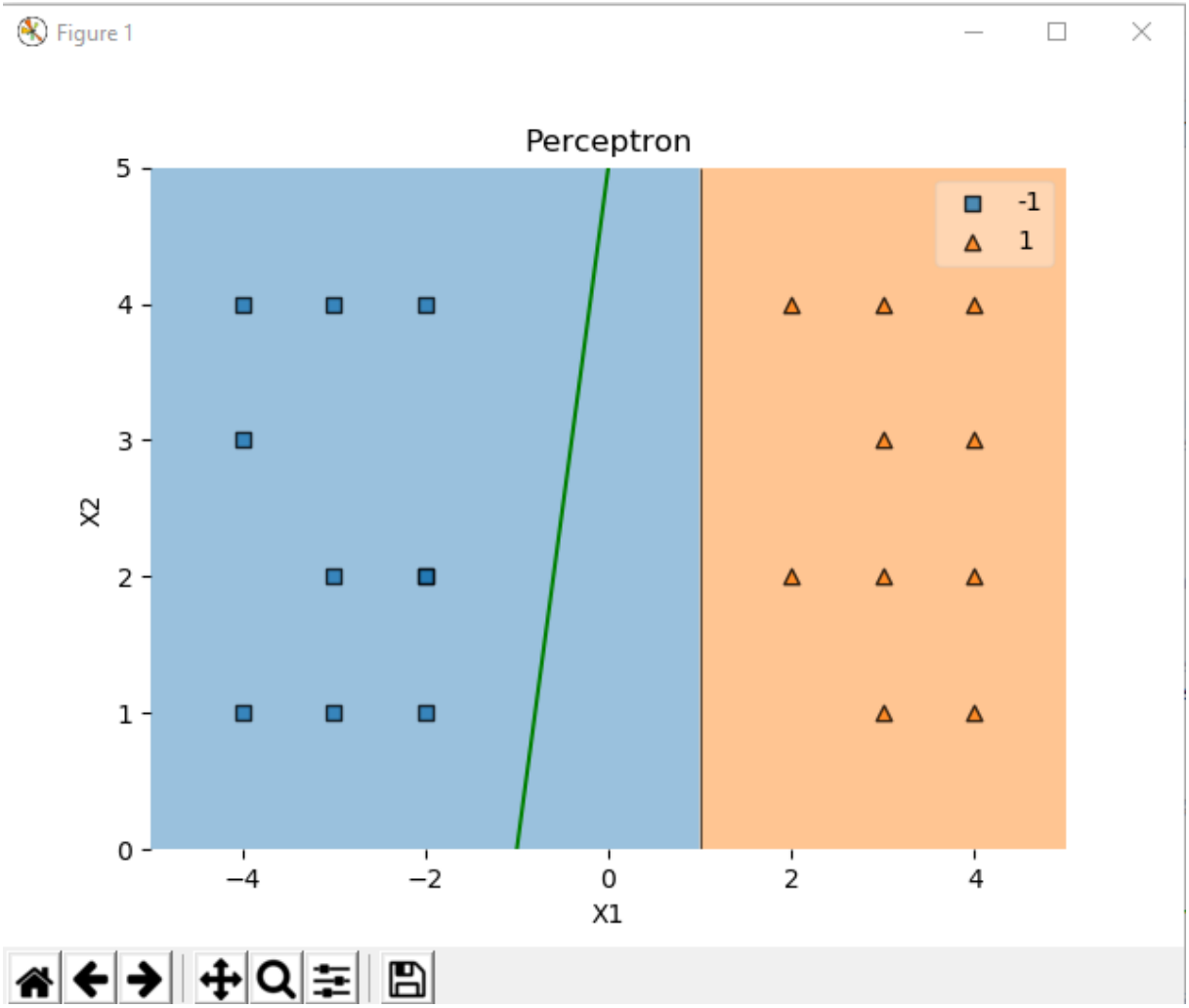
```
print("Number of updates till convergence (size: 20 data: generated by
hand)", p.counter)
```

⇨ The train methos that is called:

```
def train(self, X, y):
    # get the number of features of the dataset. X.shape returns the size of the
matrix as [mxn]. We are looking for the number n
    num_features = X.shape[1]
    # define a weight array depending on an object. This array length should be
the number of features of the dataset+1. +1 comes from the bias weight.
    # fill the first w array with zeros.
    self.w = np.zeros(num_features + 1)
    # hold a counter to count the number of times that the algorithm takes to
converge. Initialize with zero.
    self.counter=0
    # create a val variable. Assign self.w[0] value to val variable.
    # the point of doing this: In the for loop below, self.w[0] value is updated
by weight update value. Weight update is only change if the current
    # separator line of the perceptron is not working correctly and needs to be
updated. When weight update value changes so does self.w[0] change.
    # So in the each epoch I compare the val with self.w[0] to see if there is a
change. If there is a difference then counter is incremented by 1
    val=self.w[0]
    # Perform the epochs
    for i in range(self.epochs):
        # For every combination of X and y
        # check if self.w[0] is changed. If so increment the counter by 1. This
keep tracks of the number of iterations it takes to converge.
        if val!=self.w[0]:
            self.counter=self.counter+1
        # Update val
        val=self.w[0]
        # look into the samples and outcomes of combined X and y
        for sample, outcome in zip(X, y):
        # Turning -1 labels to 0 like a binary classification.
            if (outcome==-1):
                outcome=0
            # Use the predict method to generate a prediction and compare it with
the outcome
            prediction    = self.predict(sample)
            # difference between prediction and actual outcome
            diff    = (outcome - prediction)
            # Weight update = w_update variable is created. The value of the
variable calculated as in PLA rule. Learning rate*difference
            w_update = self.eta * diff
            # update the parameters of weight vector accordingly. If there is no
difference then the weight vector should not change.
            self.w[1:]    += w_update * sample
            self.w[0]     += w_update


    # returns the object
    return self

__# Generates a prediction based on the given sample
__def predict(self, sample):
```

```
    # np.dot calculates the dot product of given arrays
    # outcome is the added value of dot prodcut and weight update
    outcome = np.dot(sample, self.w[1:]) + self.w[0]
    # a prediction value is returned depending on the value of outcome. If it is
positive prediction is 1, if it is negative prediction is zero
        return np.where(outcome > 0, 1, 0)
```

⇨ Plotted plane:

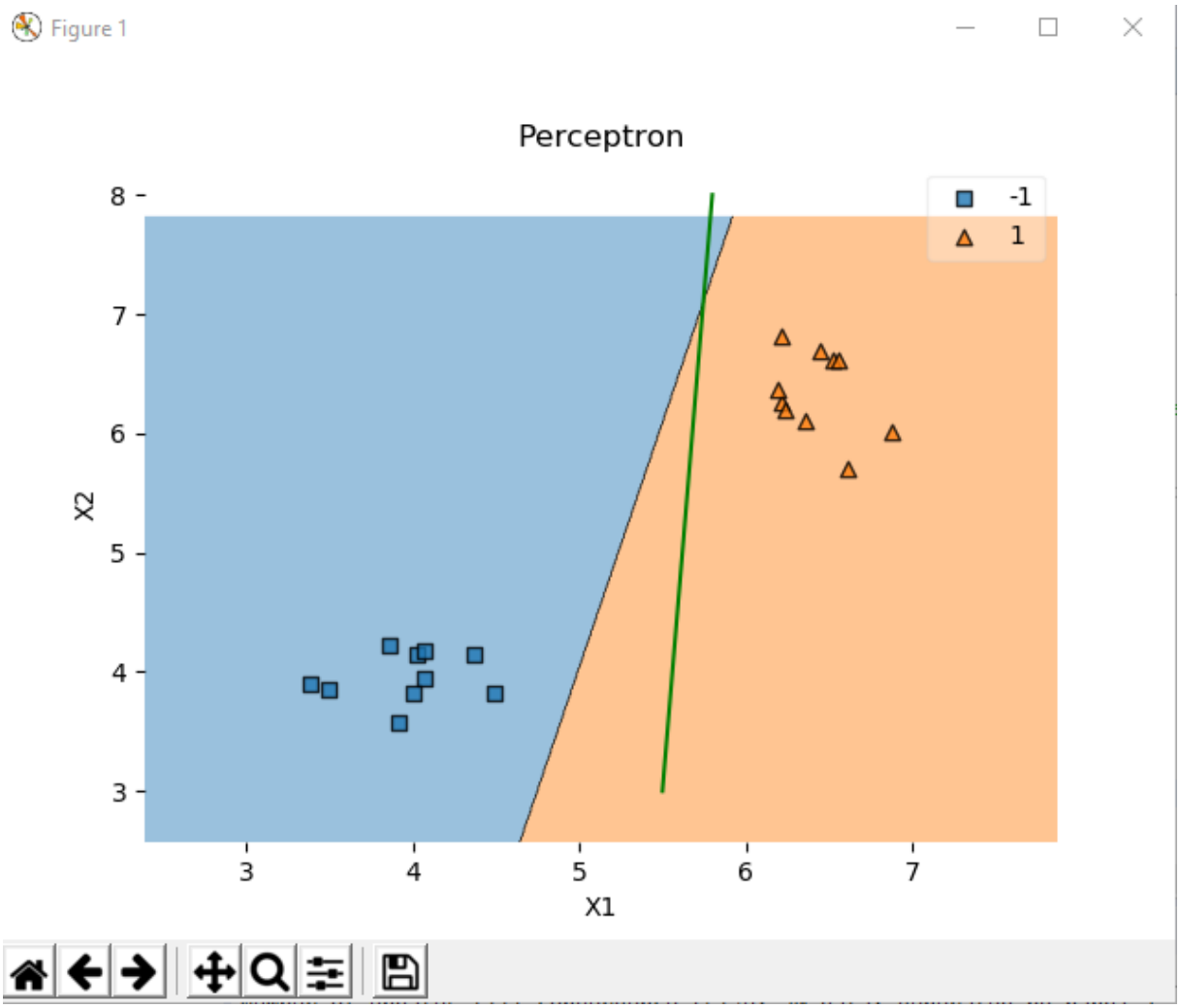


⇨ The number of updates to converge: 2

(c) Repeat (b) with randomly generated 20 inputs.

```
# Question part c
# Repeat everything in b) with another randomly generated dataset of size 20, and
compare the result to b)

# starting with creating the targets array. 10 negative ones and 10 positive ones
are created and combined.
negones = np.zeros(10)-1
ones = negones+2
targets = np.concatenate((negones, ones))
```

```
# Generate data randomly.
# Data is selected by normal distribution with spread of 0.25
# 10 data is mapped to -1 and 10 mapped to 1
# Data is 2 dimensional
neg = np.random.normal(4, 0.25, (10,2))
pos = np.random.normal(6.5, 0.25, (10,2))# Prepare input data
X = np.concatenate((neg,pos))
y=targets
trained_model = p2.train(X, y)

plot_decision_regions(X, y.astype(np.integer), clf=trained_model)
plt.title('Perceptron')
plt.xlabel('X1')
plt.ylabel('X2')
# A line that separates the points is found and market with a green color
plt.plot([5.5,5.8],[3,8], "g")
plt.show()
# Print the number of updates it takes to converge
print("Number of updates till convergence (size: 20 data: generated randomly)",
p2.counter)
```

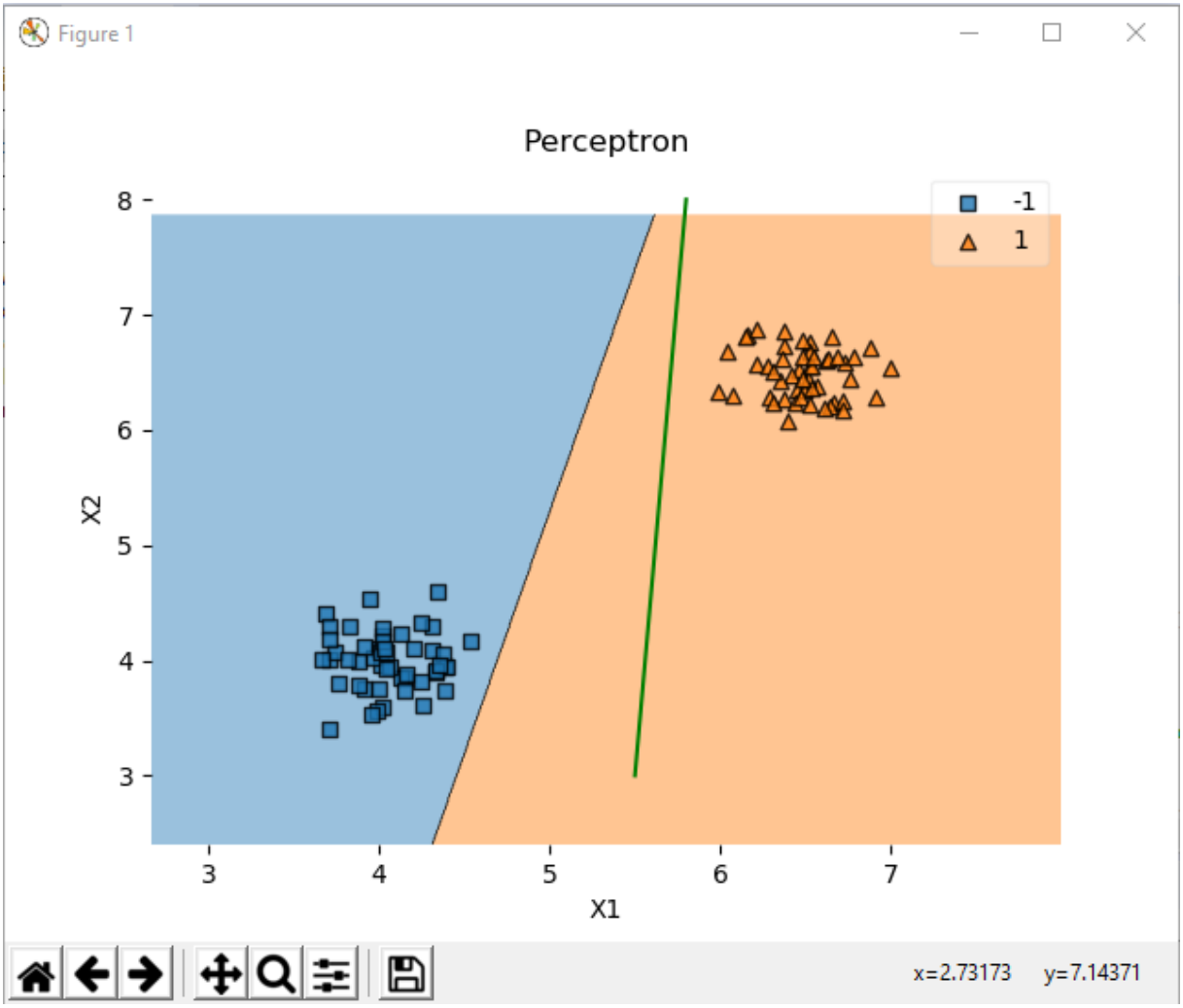


⇨ The number of updates to converge: 11

Observation: The number larger than (b)

(d) Repeat (b) with randomly generated 100 inputs.

```
# Question part d
# Repeat everything in b) with another randomly generated dataset of size 100, and
compare the results to b)

#input size 100
#Generate targets
negones = np.zeros(50)-1
ones = negones + 2
targets = np.concatenate((negones, ones))


# Generate data
neg = np.random.normal(4, 0.25, (50,2))
pos = np.random.normal(6.5, 0.25, (50,2))

# Prepare input data
X = np.concatenate((neg,pos))
y=targets
trained_model = p3.train(X, y)

plot_decision_regions(X, y.astype(np.integer), clf=trained_model)
plt.title('Perceptron')
plt.xlabel('X1')
plt.ylabel('X2')
# A line that separates the points is found and market with a green color
plt.plot([5.5,5.8],[3,8], "g")
plt.show()
# Print the number of updates it takes to converge
print("Number of updates till convergence (size: 100 data: generated randomly)",
p3.counter)
```



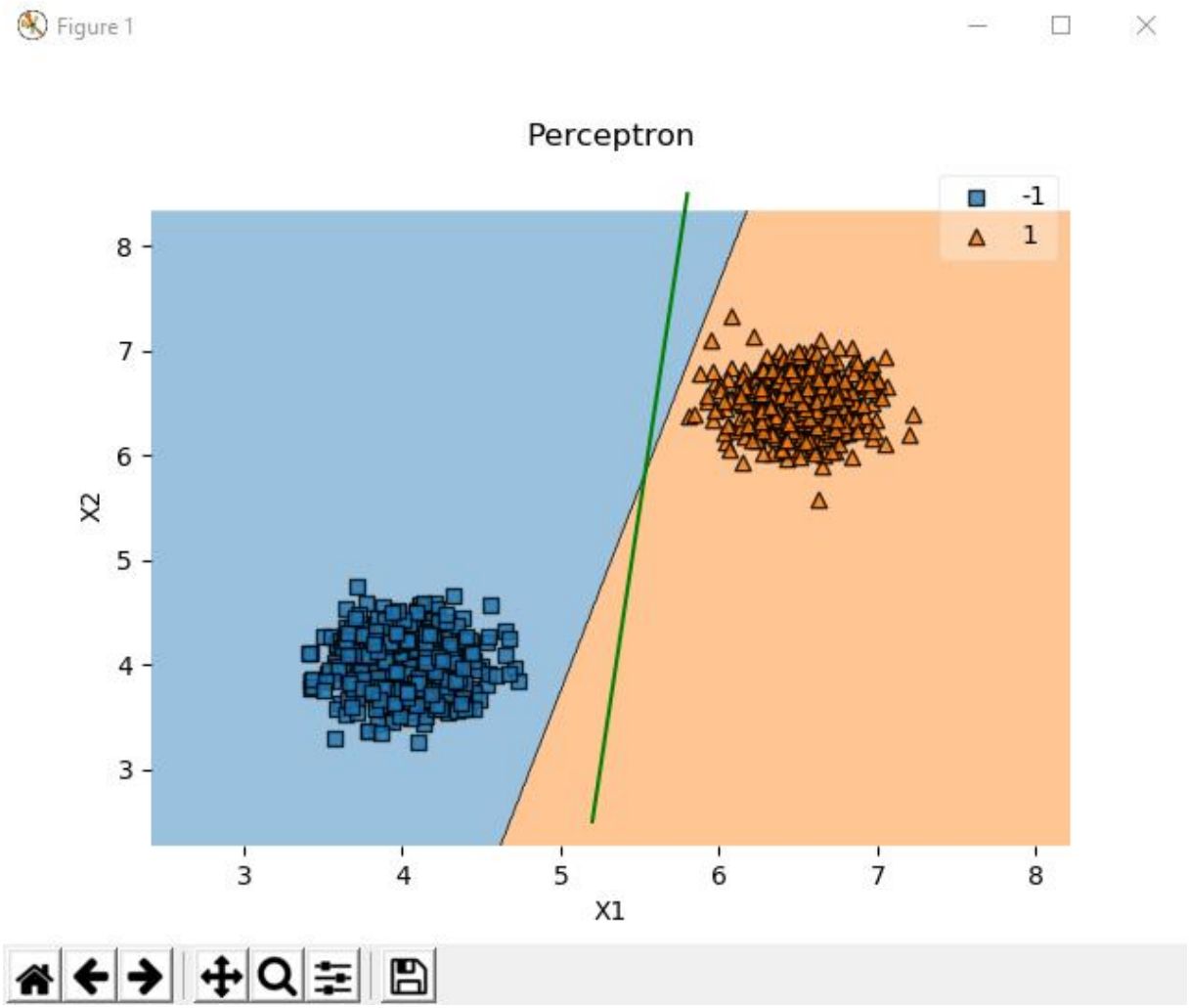
⇨ The number of updates to converge: 22

Observation: The number is higher than in (b).

(e) Repeat (b) with randomly generated 1000 inputs.

```
# Question part e
# Repeat everything in b) with another randomly generated dataset of size 1000.,
and compare the results to b)

#input size 1000
# Generate targets
negones = np.zeros(500)-1
ones = negones + 2
targets = np.concatenate((negones, ones))


# Generate data
neg = np.random.normal(4, 0.25, (500,2))
pos = np.random.normal(6.5, 0.25, (500,2))

# Prepare input data
```

```
X = np.concatenate((neg,pos))
y=targets
trained_model = p4.train(X, y)

plot_decision_regions(X, y.astype(np.integer), clf=trained_model)
plt.title('Perceptron')
plt.xlabel('X1')
plt.ylabel('X2')
# A line that separates the points is found and market with a green color
plt.plot([5.2,5.8],[2.5,8.5], "g")
plt.show()
# Print the number of updates it takes to converge
print("Number of updates till convergence (size: 1000 data: generated randomly)",
p4.counter)
```



⇨ The number of updates to converge: 98

Observation: The number is higher than in (b).

(f) Randomly generate data size of 1000 with 10 dimensions.

```
# Generate data with 10 dimensions
neg = np.random.normal(4, 0.25, (500,10))
pos = np.random.normal(6.5, 0.25, (500,10))

# Prepare input data
X = np.concatenate((neg,pos))
y=targets
# Essentially we use the same algorithm. In the train methos weight array is
calculated by the number of features X has.
# Therefore 10 dimension only makes the weight vector size of 11.(1 more is added
for bias)
trained_model = p5.train(X, y)
# Then we print the number of updates till convergence normally
print("Number of updates till convergence (size: 1000 data: generated randomly)",
p5.counter)
```

⇨ The number of updates to converge: 587

Observation: The number is higher than in (b).

(g) Summarize the conclusions

⇨ When generating the points I chose random distribution with a small scale in around a specific location. I believe this made the perceptron algorithm run less to converge as I saw much higher iteration numbers on my research.

⇨ While there can be drawn several different lines to separate my data (as they are generally clustered together) the idea is as the number of data becomes higher the algorithm runs more and the line produced by the algorithm becomes much closer to the target function.

⇨ The number of updates to converge increments by the input size N and dimension size d. Similarly, the greater the d and N become, the greater the running time gets also.

⇨ The more data points (N) leads to more accurate g. It becomes closer to f. Accuracy improves with data point count.

❖ Below is the whole code.

Perceptron Class:

```
import numpy as np
import matplotlib.pyplot as plt

class Perceptron:
    # Constructor method is defined to assign number of epochs denoted by epoch
and learning rate denoted by eta
    def __init__(self, epochs = 100, eta = 1):
        self.epochs = epochs
```

```
    self.eta = eta

    # Train perceptron. Parameters are the object, points and matching labels.
__def train(self, X, y):
    # get the number of features of the dataset. X.shape returns the size of the
matrix as [mxn]. We are looking for the number n
    num_features = X.shape[1]
    # define a weight array depending on an object. This array length should be
the number of features of the dataset+1. +1 comes from the bias weight.
    # fill the first w array with zeros.
    self.w = np.zeros(num_features + 1)
    # hold a counter to count the number of times that the algorithm takes to
converge. Initialize with zero.
    self.counter=0
    # create a val variable. Assign self.w[0] value to val variable.
    # the point of doing this: In the for loop below, self.w[0] value is updated
by weight update value. Weight update is only change if the current
    # separator line of the perceptron is not working correctly and needs to be
updated. When weight update value changes so does self.w[0] change.
    # So in the each epoch I compare the val with self.w[0] to see if there is a
change. If there is a difference then counter is incremented by 1
    val=self.w[0]
    # Perform the epochs
    for i in range(self.epochs):
        # For every combination of X and y
        # check if self.w[0] is changed. If so increment the counter by 1. This
keep tracks of the number of iterations it takes to converge.
        if val!=self.w[0]:
            self.counter=self.counter+1
        # Update val
        val=self.w[0]
        # look into the samples and outcomes of combined X and y
        for sample, outcome in zip(X, y):
        # Turning -1 labels to 0 like a binary classification.
            if (outcome==-1):
                outcome=0
            # Use the predict method to generate a prediction and compare it with
the outcome
            prediction    = self.predict(sample)
            # difference between prediction and actual outcome
            diff    = (outcome - prediction)
            # Weight update = w_update variable is created. The value of the
variable calculated as in PLA rule. Learning rate*difference
            w_update = self.eta * diff
            # update the parameters of weight vector accordingly. If there is no
difference then the weight vector should not change.
            self.w[1:]    += w_update * sample
            self.w[0]     += w_update

    # returns the object
    return self

__# Generates a prediction based on the given sample
__def predict(self, sample):
    # np.dot calculates the dot product of given arrays
    # outcome is the added value of dot prodcut and weight update
    outcome = np.dot(sample, self.w[1:]) + self.w[0]
    # a prediction value is returned depending on the value of outcome. If it is
positive prediction is 1, if it is negative prediction is zero
```

```
    return np.where(outcome > 0, 1, 0)
```

Dataset:

```
# Import libraries
import numpy as np
import matplotlib.pyplot as plt
from Perceptron import Perceptron
from mlxtend.plotting import plot_decision_regions

# create Perceptron class objects to use.
p = Perceptron(100, 1) # used for sample size 20 generated by hand
p2 = Perceptron(100, 1) # used for sample size 20 generated randomly
p3 = Perceptron(200, 1) # used for sample size 100 generated randomly
p4 = Perceptron(300, 1) # used for sample size 1000 generated randomly
p5 = Perceptron(1000, 1) # used for sample size 1000 with 10 dimensional data
generated randomly

# Question part a
# Generate a dataset of size 20. Plot the examples[(xn,yn)] as well as the target
function f on a plane.

# target array for 20 input. Mapping -1 or 1. For simplicity half the points map
to -1 and other half maps to 1
targets = np.array([-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1,1,1,1,1,1,1,1,1,1])
# generate sample data. The points that maps to -1 are denoted by neg as in
negative while the points map to 1 denoted by pos as in positive
neg = np.array([
    [-2,4],
    [-4,3],
    [-3,2],
    [-2,1],
    [-4,4],
    [-2,2],
    [-3,4],
    [-3,1],
    [-4,1],
    [-2,2],])
pos = np.array([

    [2,4],
    [3,4],
    [4,3],
    [4,4],
    [3,2],
    [2,2],
    [3,3],
    [4,2],
    [4, 1],
    [3, 1]])

# Prepare input data by combining negative and positive points
X = np.concatenate((neg,pos))
# assign targets array to y value
y=targets
# x_length is the length of X array in this case 20
x_length= len(X)
```

```
# Plotting the points. As by design there are 10 negative and 10 positive points.
for s, sample in enumerate(X):
# Plot the negative samples with color blue

    if s < (x_length/2):
        plt.scatter(sample[0], sample[1], c="blue")

    # Plot the positive samples with color yellow
    else:
        plt.scatter(sample[0], sample[1], c="yellow")

plt.xlabel('X')
plt.ylabel('Y')
# The target function f
plt.plot([-1,0],[0,5], "g")
# Part a is finished.
plt.show()

# Question part b
# Run the perceptron algorithm on the dataset. Report the number of updates that
the algorithm takes before converging. Plot the examples
# [(xn,yn)], the target function f, and the final hypothesis g in the same figure.
# generated the labels

# Now we run perceptron on the dataset.
# Call the train function from Perceptron class with X and y parameters to run the
perceptron algorithm on the dataset and return the trained model
trained_model = p.train(X, y)

# use mlxtend library plot_decision_regions feature to plot the trained_model,
assign the label and show them
plot_decision_regions(X, y.astype(np.integer), clf=trained_model)
plt.title('Perceptron')
plt.xlabel('X1')
plt.ylabel('X2')
# A line that separates the points is found and market with a green color
plt.plot([-1,0],[0,5], "g")
plt.show()
# Print the number of updates it takes to converge
print("Number of updates till convergence (size: 20 data: generated by hand)",
p.counter)

# Question part c
# Repeat everything in b) with another randomly generated dataset of size 20, and
compare the result to b)

# starting with creating the targets array. 10 negative ones and 10 positive ones
are created and combined.
negones = np.zeros(10)-1
ones = negones+2
targets = np.concatenate((negones, ones))


# Generate data randomly.
# Data is selected by normal distribution with spread of 0.25
# 10 data is mapped to -1 and 10 mapped to 1
# Data is 2 dimensional
neg = np.random.normal(4, 0.25, (10,2))
pos = np.random.normal(6.5, 0.25, (10,2))
```

```
# Prepare input data
X = np.concatenate((neg,pos))
y=targets
trained_model = p2.train(X, y)

plot_decision_regions(X, y.astype(np.integer), clf=trained_model)
plt.title('Perceptron')
plt.xlabel('X1')
plt.ylabel('X2')
# A line that separates the points is found and market with a green color
plt.plot([5.5,5.8],[3,8], "g")
plt.show()
# Print the number of updates it takes to converge
print("Number of updates till convergence (size: 20 data: generated randomly)",
p2.counter)

# Question part d
# Repeat everything in b) with another randomly generated dataset of size 100, and
compare the results to b)

#input size 100
#Generate targets
negones = np.zeros(50)-1
ones = negones + 2
targets = np.concatenate((negones, ones))


# Generate data
neg = np.random.normal(4, 0.25, (50,2))
pos = np.random.normal(6.5, 0.25, (50,2))

# Prepare input data
X = np.concatenate((neg,pos))
y=targets
trained_model = p3.train(X, y)

plot_decision_regions(X, y.astype(np.integer), clf=trained_model)
plt.title('Perceptron')
plt.xlabel('X1')
plt.ylabel('X2')
# A line that separates the points is found and market with a green color
plt.plot([5.5,5.8],[3,8], "g")
plt.show()
# Print the number of updates it takes to converge
print("Number of updates till convergence (size: 100 data: generated randomly)",
p3.counter)

# Question part e
# Repeat everything in b) with another randomly generated dataset of size 1000.,
and compare the results to b)

#input size 1000
# Generate targets
negones = np.zeros(500)-1
ones = negones + 2
targets = np.concatenate((negones, ones))
```

```
# Generate data
neg = np.random.normal(4, 0.25, (500,2))
pos = np.random.normal(6.5, 0.25, (500,2))

# Prepare input data
X = np.concatenate((neg,pos))
y=targets
trained_model = p4.train(X, y)

plot_decision_regions(X, y.astype(np.integer), clf=trained_model)
plt.title('Perceptron')
plt.xlabel('X1')
plt.ylabel('X2')
# A line that separates the points is found and market with a green color
plt.plot([5.2,5.8],[2.5,8], "g")
plt.show()
# Print the number of updates it takes to converge
print("Number of updates till convergence (size: 1000 data: generated randomly)",
p4.counter)


# Question part f
# Modify the experiment such that dataset has 10 dimensions. Run the algorithm on
a randomly generated dataset of size 1000. How many updates
# does the algorithm take to converge.

# Generate targets
negones = np.zeros(500)-1
ones = negones+2
targets = np.concatenate((negones, ones))


# Generate data with 10 dimensions
neg = np.random.normal(4, 0.25, (500,10))
pos = np.random.normal(6.5, 0.25, (500,10))

# Prepare input data
X = np.concatenate((neg,pos))
y=targets
# Essentially we use the same algorithm. In the train methos weight array is
calculated by the number of features X has.
# Therefore 10 dimension only makes the weight vector size of 11.(1 more is added
for bias)
trained_model = p5.train(X, y)
# Then we print the number of updates till convergence normally
print("Number of updates till convergence (size: 1000 data: generated randomly)",
p5.counter)
```

Information about Coding:

- ⇨ While doing this homework Python3 and Eclipse PyDev are used.

References:

- ⇨ Mlxtend by Sebastian Raschka
- ⇨ Learning From Data by Malik Magdon-Ismail, Yaser Abu-Mostafa, Hsuan-Tien Lin