Aslı Aykan
2016400222

CMPE 462 | Assignment 1

PART 1

The first part of the assignment is about perceptron and has three steps that contain different numbers of points. According to the target separating function, the points should be generated, and perceptron learning algorithm should be applied.

Code Summary: After taking the arguments, the program chooses the correct number of points and creates them in a randomized way, but in 2 groups which have the same number of points. Then, it finds a misclassified point and goes into a loop in which the coefficients of the decision boundary is recalculated (*first*, *second* and *third* are the coefficient names in the code) by calling *getMisclassifiedPoint()* at each iteration. If no point returns from the method, it means that the algorithm is done with dividing the points in a correct manner, so the program exits from the loop and plots the function.

STEP 1

In the first step, 50 points should be generated, so 25 of them are below the line (target separating function), and the others are above the line. Since the points are randomly generated, total number of iterations changes at every run. To compare the steps, I ran all these steps 5 times and calculated the average of them.

The command to run step 1: ***python assignment1.py part1 step1***

First Run: 15 iterations

Second Run: 4 iterations

Third Run: 7 iterations **Average Run: 13,4 iterations**

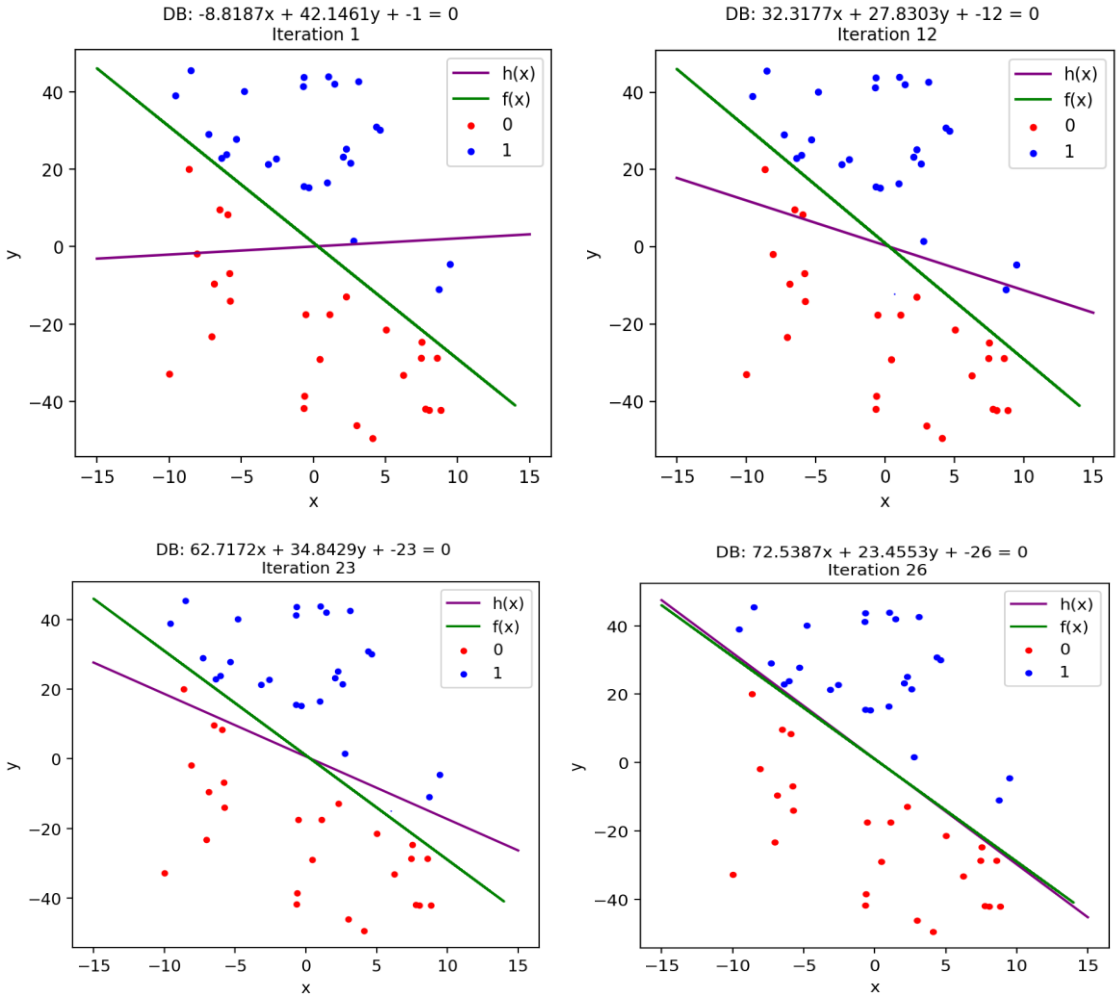
Fourth Run: 12 iterations

Fifth Run: 29 iterations

Iteration Example for Step 1

I ran step 1 again, which took 26 iterations. Some of the plots of the iterations are as follows:

Aslı Aykan
2016400222



As can be seen from the iteration example, the decision boundary approaches the targeting function as the number of iterations increases, and at the end, the decision boundary correctly divides the points as 0 and 1. (0s are below, 1s are above the line.)

STEP 2

In the second step, 100 points should be generated, so 50 of them are below the line (target separating function), and the others are above the line. Since the points are randomly generated, total number of iterations changes at every run. To compare the steps, I ran all these steps 5 times and took the average of them.

The command to run step 2: ***python assignment1.py part1 step2***

First Run: 21 iterations

Second Run: 27 iterations

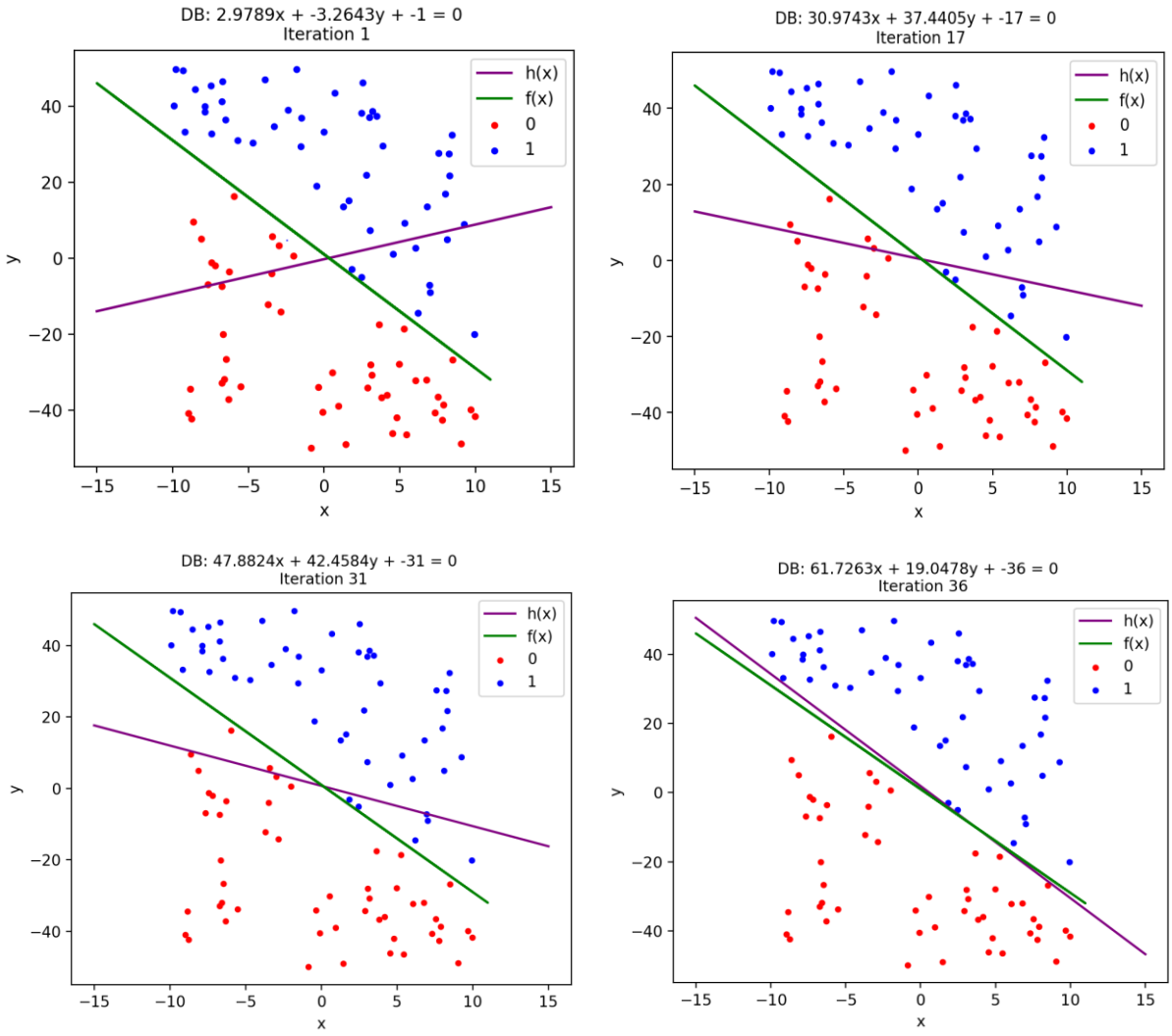
Third Run: 18 iterations **Average Run: 38,4 iterations**

Fourth Run: 84 iterations

Fifth Run: 42 iterations

Iteration Example for Step 2

I ran step 2 again, which took 36 iterations. Some of the plots of the iterations are as follows:



As can be seen from the iteration example, the decision boundary approaches the targeting function as the number of iterations increases, and at the end, the decision boundary correctly divides the points as 0 and 1. (0s are below, 1s are above the line.)

STEP 3

In the third step, 5000 points should be generated, so 2500 of them are below the line (target separating function), and the others are above the line. Since the points are randomly

generated, total number of iterations changes at every run. To compare the steps, I ran all these steps 5 times and took the average of them.

The command to run step 3: ***python assignment1.py part1 step3***

First Run: 82 iterations

Second Run: 1051 iterations

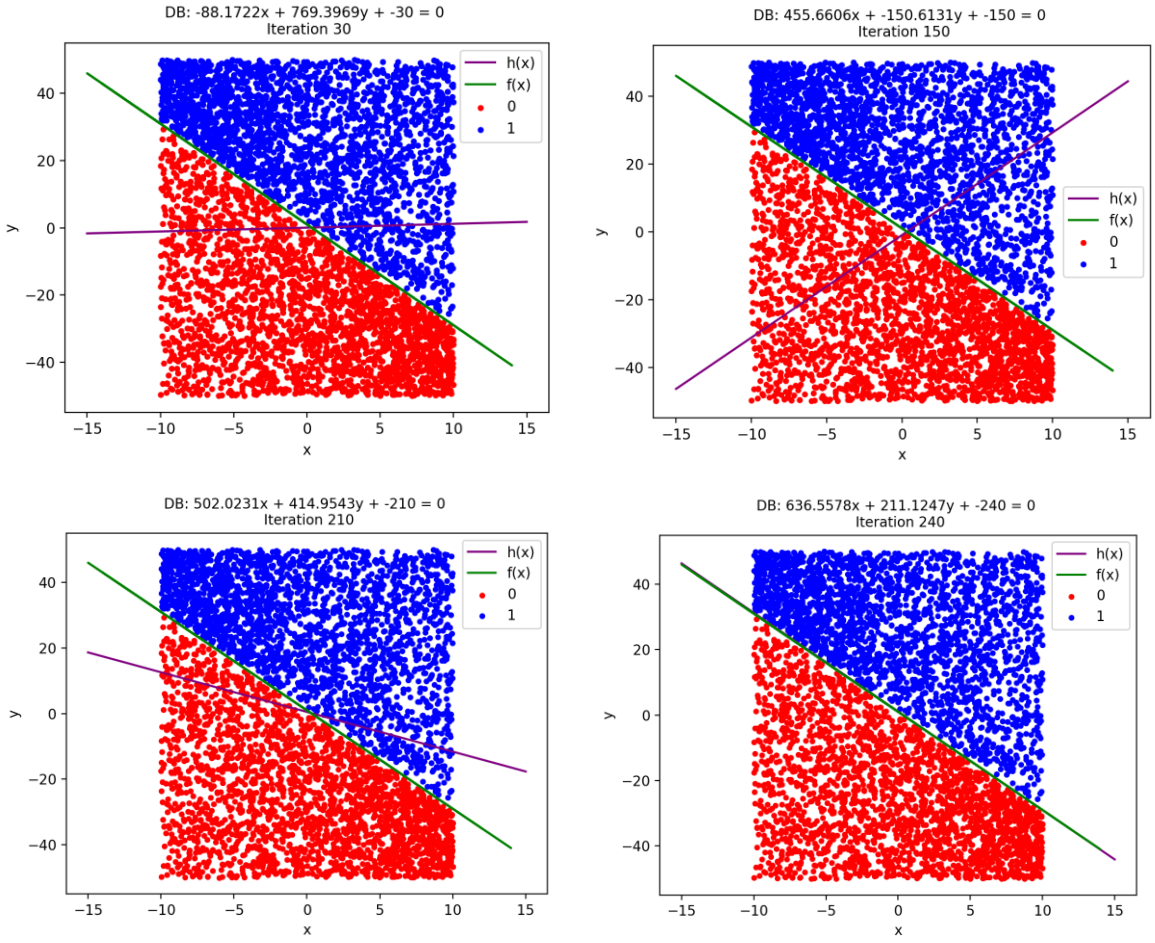
Third Run: 266 iterations **Average Run: 327,2 iterations**

Fourth Run: 170 iterations

Fifth Run: 67 iterations

Iteration Example for Step 3

I ran step 3 again, which took 240 iterations. Some of the plots of the iterations are as follows:



As can be seen from the iteration example, the decision boundary *usually* approaches the targeting function as the number of iterations increases, and at the end, the decision boundary correctly divides the points as 0 and 1. (0s are below, 1s are above the line.)

Aslı Aykan
2016400222

General Summary: To sum up part1, we can say that we generally need more iterations when we have more points while using perceptron learning algorithm, since the algorithm works by finding a misclassified point and recreating the decision boundary. However, it is not always the case. For example, the iteration example of step 2 had 26 iterations, but the fifth run of step 1 had 29 iterations. Since there is no order in points (or data), and the algorithm chooses a misclassified point in a randomized way.
Another thing that should be mentioned is that we cannot say that the algorithm always finds a closer solution to the target function as the number of points increases. If we compare the iteration examples of step 1 and step 2, we see that the resulting decision boundary in step 1 is closer to the targeting function than the decision boundary in step 2, although step 2 has more points (or data). Since, the algorithm does not try to find the best solution. As soon as it finds a solution that correctly divides the points into 2 groups, it stops running.
It is also worth to note that the algorithm does not always make the decision boundary approach to the targeting function at each iteration. Since there is no order in data (or points) and the algorithm works by finding any misclassified point at each iteration, it is possible that sometimes it may find a misclassified point and recreate the decision boundary which results in having a farther decision boundary to the targeting function.

Note: I wrote the code for plotting all the iterations, but because of the performance issues, the related code is in comments. So, when the code runs, it plots the final graph only.

PART 2

The second part of the assignment is about multiple linear regression and has three steps. In first two steps, a solution should be found for two different input data sets, and in the third step, a solution for the second data set should be found by using regularization.

STEP 1

The command to run step 2: ***python assignment1.py part2 step1***

For step 1, I implemented multiple linear regression for the input data set which is in *ds1.csv* file. I used the closed form solution to find the **w** value. The formula for w is $w= (X^TX)^{-1}t$. I generated the X matrix by dropping the last column of the .csv file, since the last column is the dependent variable, in other words, it corresponds to **t** in our formula. I also added the bias to the first column of X.

Time to complete step 1 is 58 msec.

Aslı Aykan
2016400222

STEP 2

The command to run step 2: ***python assignment1.py part2 step2***

For step 2, I implemented multiple linear regression for the input data set which is in *ds2.csv* file. I used the closed form solution to find the **w** value. The formula for w is $w= (X^TX)^{-1}t$. I generated the X matrix by dropping the last column of the .csv file, since the last column is the dependent variable, in other words, it corresponds to **t** in our formula. I also added the bias to the first column of X.

Time to complete step 2 is 232 msec.

Since the data is larger in step 2, the time passed for calculating the closed-form solution is longer.

The **w** value (closed-form solution) of step1 and step2 is printed by the code after each run, total time passed is also printed by the code.

STEP 3

The command to run step 3: ***python assignment1.py part2 step3***

For step 3, I used the cross-validation technique to choose a regularization parameter. I divided the input data set (*ds2.csv*) into five equal parts and ran the algorithm five times to be able to use each of them as a test data. I used five different candidate values for the regularization parameter. So, the algorithm runs 5*5=25 times. At the end of the algorithm, I used the calculated **w** value to test data and compare with real results. So, I calculated the distance between real values and calculated results and chose the set in which there exist smaller error values.

In the code, I created a list called errors and it contains five error lists for five different regularization parameter values, and each of them has five elements in it since the data set is divided into 5 equal parts.

I selected 5 candidate values for regularization parameter, compared the errors of them, and printed the index of the regularization parameter with lowest error. (For example, for [300, 320, 340, 360, 1000], the value that has lowest error is 360.) The **w** value calculated with regularization is also printed by the code. (It is calculated by this formula: $w= (X^TX+LI)^{-1}t$, in which L is the regularization parameter.)

Time passed for the cross-validation and **w** value calculation is: 1045 msec