Imperial College London

Machine Learning Coursework 1

Youssef Rizk (00940962)

Examined by
Dr. András György

February 6, 2017

Contents

1 Problem 1

This problem inquires about the confidence level with which one can reject the hypothesis that the probability of choosing to stay in the EU is at least 50%, given that a public poll was published, where 2052 people were asked, and of whom 55% favored leaving the EU while 45% wanted to stay.

Using Hoeffding's Inequality, where for any $0 < \delta \leq 1$, with probability at least $1 - \delta$,

$$\sum_{t=1}^{n} X_t < \sum_{t=1}^{n} E[X_t] + \sqrt{\frac{1}{2}\ln(\frac{1}{\delta})\sum_{t=1}^{n}(b_t - a_t)^2}$$

Given the specific context, we can model each person who participated in the poll as a random variable following a Bernoulli distribution, i.e. $X_t \sim Ber(p)$, and noting that the random variables X_t are independent and identically distributed. In that case, $b_t = 1$ and $a_t = 0$. We now define $\hat{p}$ as the empirical risk so that $\hat{p} = 0.45$, and we define p as the true risk so that $p = 0.5$. Finally, we note that $\sum_{t=1}^{n} E[X_t] = np$ and $\sum_{t=1}^{n} X_t = n\hat{p}$. Then,

$$\begin{aligned}
\sum_{t=1}^{n} X_t &< \sum_{t=1}^{n} E[X_t] + \sqrt{\frac{1}{2}\ln(\frac{1}{\delta})\sum_{t=1}^{n}(b_t - a_t)^2} \\
n\hat{p} &< np + \sqrt{\frac{1}{2}\ln(\frac{1}{\delta})n} \\
n(\hat{p} - p) &< \sqrt{\frac{1}{2}\ln(\frac{1}{\delta})n} \\
n^2(\hat{p} - p)^2 &< \frac{1}{2}\ln(\frac{1}{\delta})n \\
2n(\hat{p} - p)^2 &< \ln(\frac{1}{\delta}) \\
\exp(2n(\hat{p} - p)^2) &< \frac{1}{\delta} \\
\delta &< \exp(-2n(\hat{p} - p)^2) \\
\delta &< 0.000035005687839
\end{aligned}$$

So, the confidence level with which I can reject the hypothesis that the probability of choosing to stay in the EU is at least 50% is $1 - \delta > 1 - 0.000035005687839 \approx 0.999965$.

2 Problem 2

2.1 Part A

This part asks us to show the $\rho = \min_{i\in\{1,..,n\}} y^{(i)} w_*^\top x^{(i)} > 0$. I proceed to show this by recalling that w_* is defined such that $\text{sign}(w_*^\intercal x^{(i)}) = y^{(i)}$ for all $i \in \{1,..,n\}$, i.e. that the parameter vector w_* correctly classifies all points in the training data set. This leads to the following:

$$\begin{aligned} w_*^\intercal x^{(i)} y^{(i)} &= \begin{cases} -|w_*^\intercal x^{(i)}| \times -1 = |w_*^\intercal x^{(i)}| & y^{(i)} = -1 \\ |w_*^\intercal x^{(i)}| \times 1 = |w_*^\intercal x^{(i)}| & y^{(i)} = 1 \end{cases} \\ &= |w_*^\intercal x^{(i)}| \end{aligned}$$

Since $|w_*^\intercal x^{(i)}| > 0$ for all $i \in \{1,...,n\}$, then $\rho = \min_{i\in\{1,...,n\}} y^{(i)} w_*^\intercal x^{(i)} > 0$.

2.2 Part B

This part of the question asks us to show that for any t before the algorithm stops, $w_*^\intercal w_t \geq t\rho$. To show this, it is necessary to first show that $w_*^\intercal w_t \geq w_*^\intercal w_{t-1} + \rho$. This follows from rewriting w_t as $w_t = w_{t-1} + y_{t-1}x_{t-1}$ and noting that $w_*^\intercal y_{t-1}x_{t-1} > \rho$ from part A. Then,

$$\begin{aligned} w_*^\intercal w_t &= w_*^\intercal (w_{t-1} + y_{t-1}x_{t-1}) \\ &= w_*^\intercal w_{t-1} + w_*^\intercal y_{t-1}x_{t-1} \\ &\geq w_*^\intercal w_{t-1} + \rho \end{aligned}$$

Since $w_*^\intercal w_t \geq w_*^\intercal w_{t-1} + \rho$, if I can prove that $w_*^\intercal w_{t-1} + \rho \geq t\rho$, then I will prove the claim. Thus I proceed with induction on $w_*^\intercal w_{t-1} + \rho \geq t\rho$

Let $t = 1$

$$\begin{aligned} w_*^\intercal w_0 + \rho &\geq \rho \\ \rho &\geq \rho && \text{(Since } w_0 = \mathbf{0}\text{)} \end{aligned}$$

We now assume

$$w_*^\intercal w_{t-1} + \rho \geq t\rho$$

Then,

$$w_*^\intercal w_{t-1} + w_*^\intercal y_{t-1}x_{t-1} + \rho \geq t\rho + w_*^\intercal y_{t-1}x_{t-1} \quad \text{(Adding } w_*^\intercal y_{t-1}x_{t-1} \text{ to both sides of the inequality)}$$

We know that $w_*^\intercal y_{t-1}x_{t-1} \geq \rho$ and so the above inequality can be simplified to the form we desire to prove that the claim is valid for $t+1$,

$$\begin{aligned} w_*^\intercal w_{t-1} + w_*^\intercal y_{t-1}x_{t-1} + \rho &\geq t\rho + \rho \\ w_*^\intercal (w_{t-1} + y_{t-1}x_{t-1}) &\geq (t+1)\rho \\ w_*^\intercal w_t + \rho &\geq (t+1)\rho && \text{Since } w_t = w_{t-1} + x_{t-1}y_{t-1} \\ &\therefore \end{aligned}$$

2.3 Part C

This part of the question asks us to show that $||w_t||^2 \leq tR^2$ where $R = \max_{i\in\{1,..,n\}} ||x^{(i)}||$. As in the previous part, I will proceed by first proving another property and using induction, namely that $||w_t||^2 \leq ||w_{t-1}||^2 + ||x_{t-1}||^2$. This follows from

$$\begin{aligned} ||w_t||^2 &= ||w_{t-1} + y_{t-1}x_{t-1}||^2 && \text{Since } w_t = w_{t-1} + x_{t-1}y_{t-1} \\ &= ||w_{t-1}||^2 + y_{t-1}^2||x_{t-1}||^2 + 2y_{t-1}w_{t-1}^\intercal x_{t-1} \\ &\leq ||w_{t-1}||^2 + ||x_{t-1}||^2 \end{aligned}$$

The last step comes about from the fact that $y_{t-1}^2 = 1$. Also, $y_{t-1}w_{t-1}^\intercal x_{t-1} \leq 0$ because we know that there has been a misclassification. Proving $||w_t||^2 \leq tR^2$ by induction, we obtain the following:

Prove the claim for the base case

$$\begin{aligned} ||w_1||^2 &\leq R^2 && \text{Let } t = 1 \\ ||w_0 + x_0y_0||^2 &\leq R^2 && \text{Since } w_1 = w_0 + x_0y_0 \\ ||x_0||^2 &\leq R^2 && \text{Since } w_0 = \mathbf{0} \ \& \ |y_0| = 1 \end{aligned}$$

The above inequality hold since by definition, $||x_0|| \leq R$. Now we assume that the statement is true for t, i.e. $||w_t||^2 \leq tR^2$. We want to show that it is also true for $t+1$. We start from the hint provided.

$$||w_{t+1}||^2 \leq ||w_t||^2 + ||x_t||^2 \qquad \text{Substituting } t+1 \text{ into hint}$$

From the previous assumption, we know that $||w_t||^2 \leq tR^2$, so,

$$||w_{t+1}||^2 \leq tR^2 + ||x_t||^2$$

By definition, we also know that $||x_t||^2 \leq R$, and so the whole inequality simplifies to

$$\begin{aligned} ||w_{t+1}||^2 &\leq tR^2 + R^2 \\ ||w_{t+1}||^2 &\leq (t+1)R^2 \\ &\therefore \end{aligned}$$

2.4 Part D

This part of the question asks us to show that $t \leq \frac{R^2||w_*||^2}{\rho^2}$. To show this, we employ the results of parts B and C.

From part B,

$$w_*^\intercal w_t \geq t\rho \Rightarrow t \leq \frac{w_*^\intercal w_t}{\rho}$$

and from part C,

$$||w_t||^2 \leq tR^2 \Rightarrow t \geq \frac{||w_t||^2}{R^2}$$

Using the Cauchy-Schwartz inequality, (namely that $a^\intercal b \leq ||a||||b||$), we can rewrite the result from part B as,

$$\begin{aligned} ||w_*||||w_t|| &\geq t\rho \\ ||w_*||^2||w_t||^2 &\geq t^2\rho^2 \\ ||w_t||^2 &\geq \frac{t^2\rho^2}{||w_*||^2} \end{aligned}$$

Substituting the above inequality into the result of part C, we obtain the following:

$$\frac{t^2\rho^2}{||w_*||^2} \leq tR^2$$

And finally rearranging, we obtain the wanted result:

$$t \leq \frac{R^2||w^*||^2}{\rho^2}$$

Since t represents the number of iterations of the Perceptron Learning Algorithm and is finite, this result provides an upper bound for t, hence showing that the algorithm stops after at most $\frac{R^2||w_*||^2}{\rho^2}$.

3 Problem 3

Problem 3 aims to explore a more practical side of the Perceptron Learning Algorithm. It walks us through the process of writing the algorithm and applying it to several applications. The programs in this section were written in MATLAB. Please ensure that all the files in the "Assignment 1" folder are in the same directory as the `assignment1_yr914.m` file, which is located in the folder called `Assignment 1`. The data used for part C is located in a folder called `data` which is in the same directory as `Assignment 1`.

3.1 Part A

This section introduced us to a practical side of the Perceptron Algorithm, as we had to write a program to implement it. Although several practical implementations limit the number of iterations of the algorithms to a predefined bound, I decided to implement the algorithm initially without including a limit of the number of iterations. I do this using a Boolean variable `correct`, which is `true` if there are no misclassifications of the training data, and `false` if there are. Naturally, the algorithm terminates when `correct = true`. I then repeatedly iterate through the training data, always choosing the next entry in my array, and check whether the data point has been correctly classified given the current weight vector. If the classification is correct, I proceed to the next data entry, but if not, I update the value of the weight vector according to the rules of the algorithm, and continue iterating through the data again, proceeding to the next element. To assign a value to `correct`, I employ a function called `errorprob`, which is shown in a later section and essentially measures the training error given the current weights. `correct = true` when the training error is equal to 0. The code for the algorithm is shown below.

```
function [w] = percep(X,Y)

w = zeros(1,size(X,2)); %Weights vector initialization
correct = false; % boolean variable will control loops

while ~correct
    for ii = 1 : size(X,1)

        if sign(X(ii,:)*w') ~= Y(ii) %Check if the classification is right
            w = w + X(ii,:) * Y(ii);
        end
        correct = errorprob(sign(w*X').*Y) == 0; %if not, we start again

    end
end

end
```

The function here takes in input a matrix consisting of the training data set `X` and the corresponding classifications vector `Y`. Please note that the offset of 1 has to be added to `X` before applying it as the input to the algorithm. To test the functionality of the program, I created a set of 2-element data points, which are uniformly distributed in the unit square, and are separated by the line $y = 0.5$, such that any point under this line is assigned a value of -1, and any point above this line is assigned a value of 1. The idea is to test this algorithm on a simple data set to ensure correct functionality. The below script shows a testing procedure for this program:

```
X = [ones(100,1) rand(100,2)];
Y = simpleClassify(X);

scatter(X(:,2),X(:,3))
hold on
X1 = 0.5*ones(100,1);
plot(X(:,2),X1);
hold on

w = percep(X,Y);
w = w/w(3)
Y1 = -w(2)*X(:,2) - w(1);
plot(X(:,2),Y1);
legend('Data Points','Actual Separator','Calculated Separator');
title('Testing the Perceptron Algorithm');
xlabel('x1')
ylabel('x2')
```

This employs 2, 4, 10, 100 randomly selected data points and their classifications (reference Appendix for the code of `simpleClassify` function) as input to the Perceptron program. I then proceed to plot the data points, along with the true classifier line and the calculated line, and the resulting plot is shown. The idea with the testing script is to employ the algorithm on a data set where the classification is already known, and compare the performance of the algorithm to.

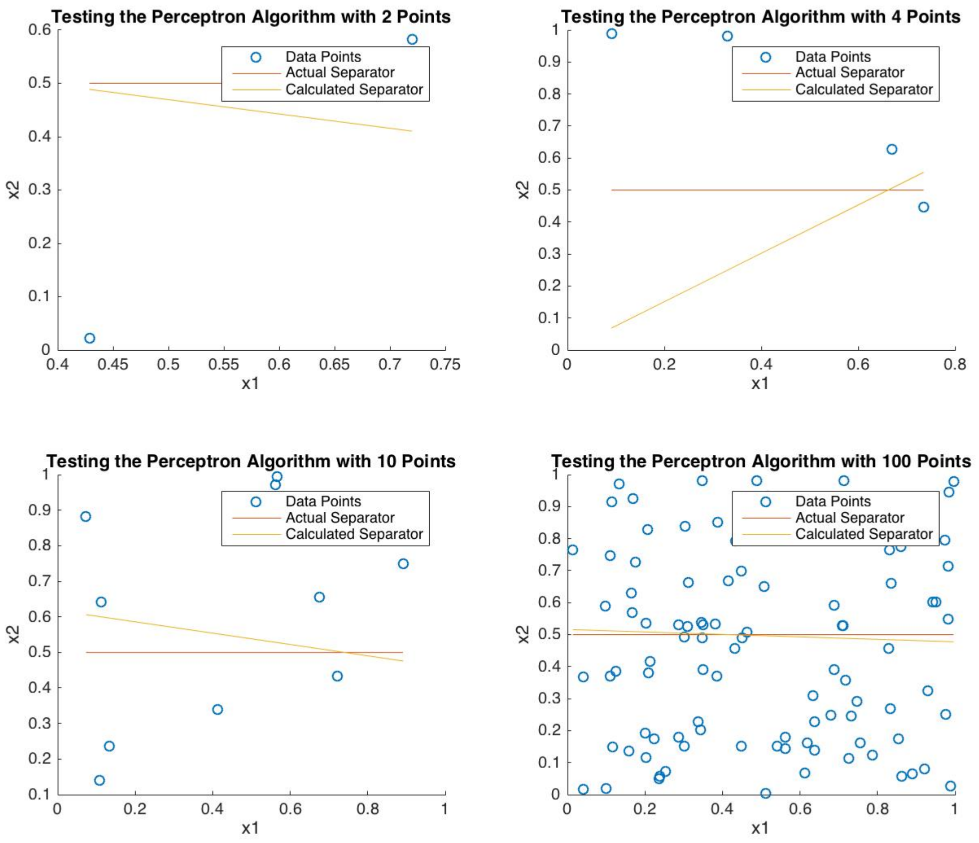


Figure 1: Testing Percpetron with different data sizes

3.2 Part B

This part of the third problem aims to demonstrate the procedure through which a Perceptron algorithm, and more generally, a machine learning algorithm is tested for correctness. The idea is to employ a simple example, where the performance of the algorithm can be compared to an already known result.

3.2.1 Closed-form Formula of a Linear Separator

Naturally, the Perceptron Learning Algorithm is not perfect, and thus may produce an error with certain test data points. In the context of this question, our separator line is linear and the data points are uniformly distributed in the unit square. There are two main ways to measure the performance of the algorithm in terms of its error, a theoretical way and a more algorithmic approximate way. The theoretical way is discussed first.
We can label the actual linear separator line as $x_2 = \hat{a}x_1 + \hat{b}$ (in the particular case, $\hat{a} = 1$ and $\hat{b} = 0.1$) and the separator line obtained through our algorithm as $x_2 = ax_1 + b$. When $\hat{a}$ and a are different and $\hat{b}$ or b are different, then the computed separator line is not equal to the actual line, and will then incur and error. Specifically, the algorithm will incur an error for any data point located in the area between the two lines, as shown in the below figure (obtained from the lecture notes).

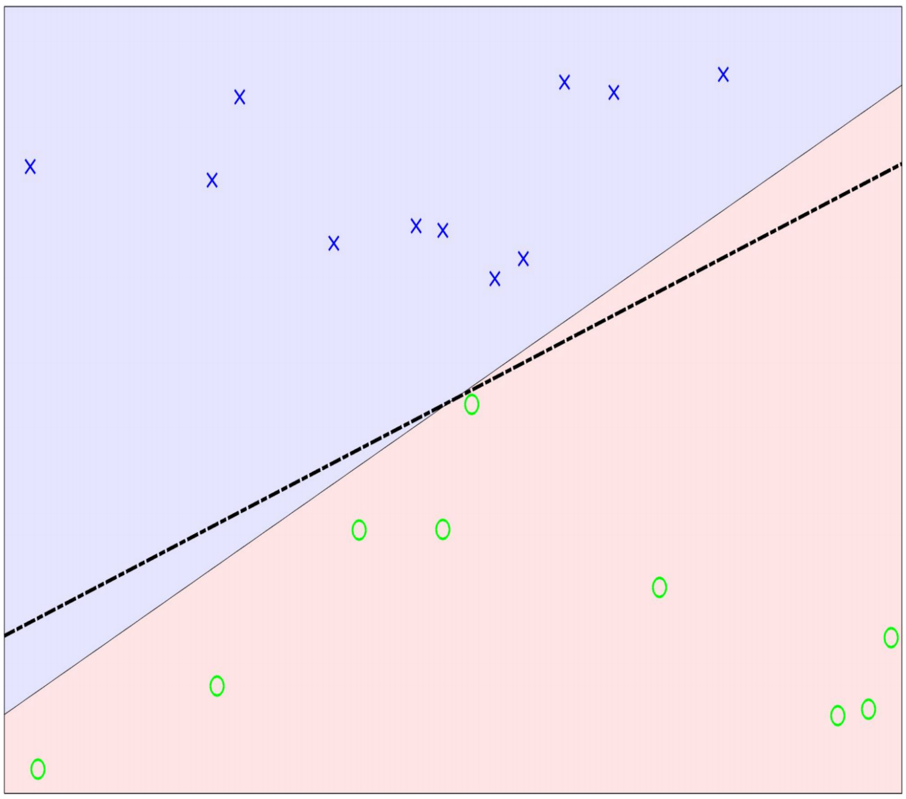


Figure 2: Visualization of the test error

Consequently, the probability of error is equivalent to the probability that a point falls in the area between the two lines, or equivalently, $p(e) = \frac{\text{Area between lines}}{\text{Total area}}$. We note here that this formula is only applicable because the points are uniformly distributed, and that Total area = 1 since this is a unit square. Thus, letting $\bar{x}$ denote the intersection point of the two lines, if $\bar{x} \notin (0,1)$, then

$$\begin{aligned} p(e) &= \left| \int_0^1 (a-\hat{a})x + (b-\hat{b})dx \right| \\ &= \left| \frac{(a-\hat{a})}{2} + (b-\hat{b}) \right| \end{aligned}$$

alternatively, if $\bar{x} \in (0,1)$, then

$$\begin{aligned} p(e) &= \left| \int_0^{\bar{x}} (a-\hat{a})x + (b-\hat{b})dx \right| + \left| \int_{\bar{x}}^1 (a-\hat{a})x + (b-\hat{b})dx \right| \\ &= \left| \frac{(a-\hat{a})}{2}\bar{x}^2 + (b-\hat{b})\bar{x} \right| + \left| (a-\hat{a}) - \frac{(a-\hat{a})}{2}\bar{x}^2 + (b-\hat{b}) - (b-\hat{b})\bar{x} \right| \end{aligned}$$

Although the above formulas provide a good foundation for the test error of the algorithm, they do not consider several cases, and thus would not always provide an accurate test error figure.
A more algorithmic way exists that provides a reasonable approximation for the test error. Instead of finding a closed-form formula, we can create a sufficiently large test data set, and calculate the empirical test error, i.e. measuring the number of misclassified test points. This would be done using the function `errorprob` again. Of course, there need to be certain restrictions on the number of data points in the test set that we will employ. These can be deduced

from Hoeffding's inequality. I assume that I want a 90% confidence interval that the deviation between the empirical test error and the true test error is less than 0.01. Thus we require

$$\begin{aligned} 2e^{-2(0.01)^2 n} &= 0.1 \\ n &= 14979 \end{aligned}$$

Thus, to guarantee a 90% confidence interval that the deviation between the two test errors is no greater than 0.1, we require a test data set consisting of approximately 15,000 data points. The `errorprob` program is shown below.

```
function p = errorprob(y)

    p = 0;
    for i = 1:size(y,2)
        if y(i) == -1
            p = p+1;
        end
    end
    p = p/(size(y,2));

end
```

3.2.2 Training the Algorithm

It is now time to employ the Perceptron algorithm to be able to classify points according to the relationship laid out in Part B. Again, I am generating uniformly distributed random points in the unit square and classifying them according to the given rule using a function, `classify` (which can be found in the appendix).
I begin the process of training the algorithm by training it on 100 to 500 data points in increments of 100. I then plot the test error for each using the `errorprob` function written, all of which is done using the following script. Please note that I am using 100,000 test data points rather than the calculated 15,000 for better results. I have also attached the resulting plot of test error vs. number of data points.

```
%Exercise 3.b.2

Test = [ones(100000,1) rand(100000,2)];
testClassify = classify(Test);
p = zeros(1,5);
figure
for i = 1:5
    X = zeros(i*100,3);
    X(:,1:3) = [ones(i*100,1) rand(i*100,2)];
    Y = classify(X);
    A = percep(X,Y);
    A = A/A(3);
    p(i) = errorprob(sign(A*Test').*testClassify);
end
plot(1:5,p)
```

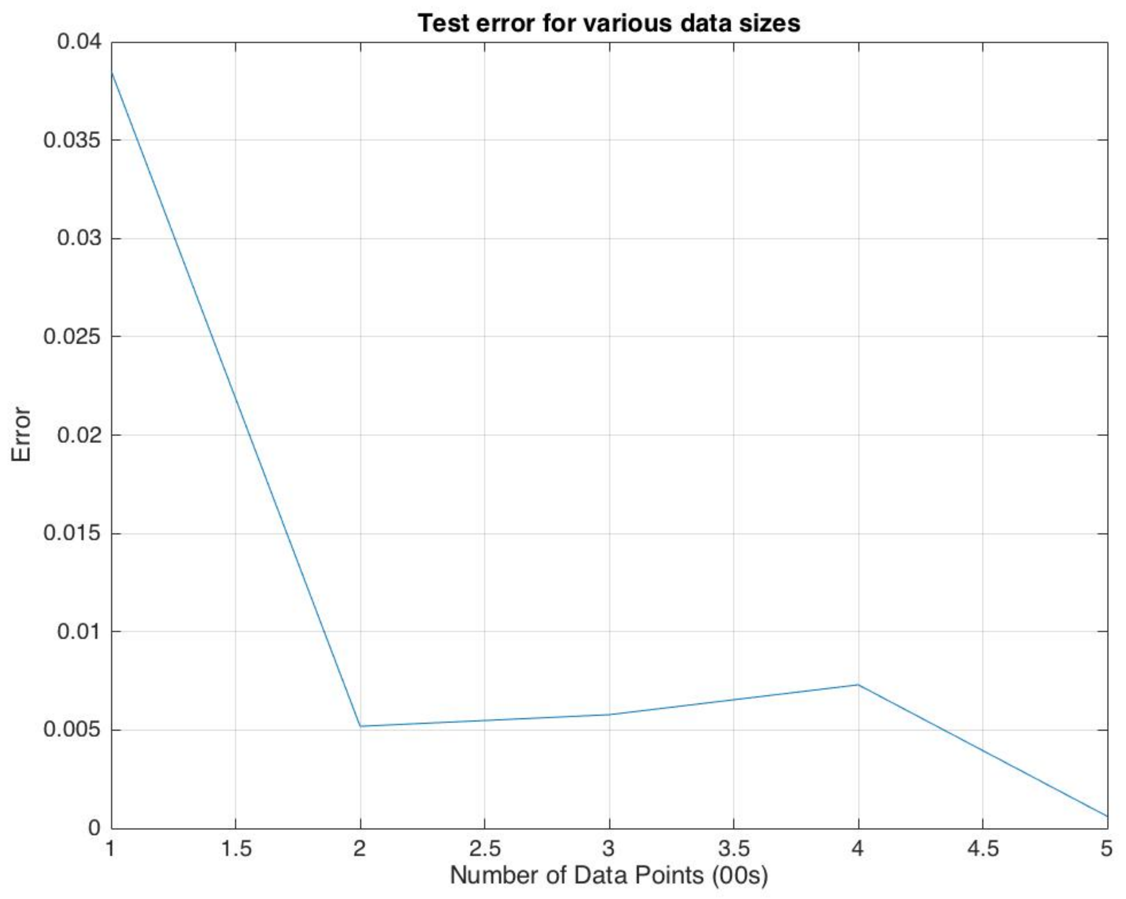


Figure 3: Test error as a function of the number of data points

3.2.3 Examining the Test Error Further

It may seem surprising that the test error does not decrease as the number of data points in the training set increases. However, we must note that the data points generated are actually random, and it is thus important to take an average of the test error for each given number of data points to get an accurate picture of the trend. This is done through the following script, where each test error is calculated 100 times and both the normal average and an average considering only the middle 90% of data points, i.e. removing the first and last 5%. As we can see from figure 4, the two lines are indeed quite close. To gain a deeper insight, I found it imperative to also plot the average and the 90% confidence interval where I consider the range between the 5th and 95th percentile. This is shown in figure 5. Noticeably, the trend we would expect, where the test error decreases with the number of data points indeed holds in a general statistical sense, as can be seen from the supporting figures.

```
%Exercise 3.b.3

Test = [ones(100000,1) rand(100000,2)];
testClassify = classify(Test);


figure
n = 100;
```

```
prob = zeros(n,5);
for ii = 1:n
p = zeros(1,5);
for i = 1:5
    X = zeros(i*100,3);
    X(:,1:3) = [ones(i*100,1) rand(i*100,2)];
    Y = classify(X);
    A = percep(X,Y);
    A = A/A(3);
    p(i) = errorprob(sign(A*Test').*testClassify);
    prob(ii,i) = p(i)
end
end
sort(prob);
plot(1:5,mean(prob(6:95,1:5)))
hold on;
plot(1:5,mean(prob))

title('Average test error for various data sizes')
xlabel('Number of data points (00s)')
ylabel('Error')
grid on
legend('90% Confidence','Average')
```

The generated plot is shown overleaf (Please note that a more detailed script to produce figure 5 is included in the assignment MATLAB file).

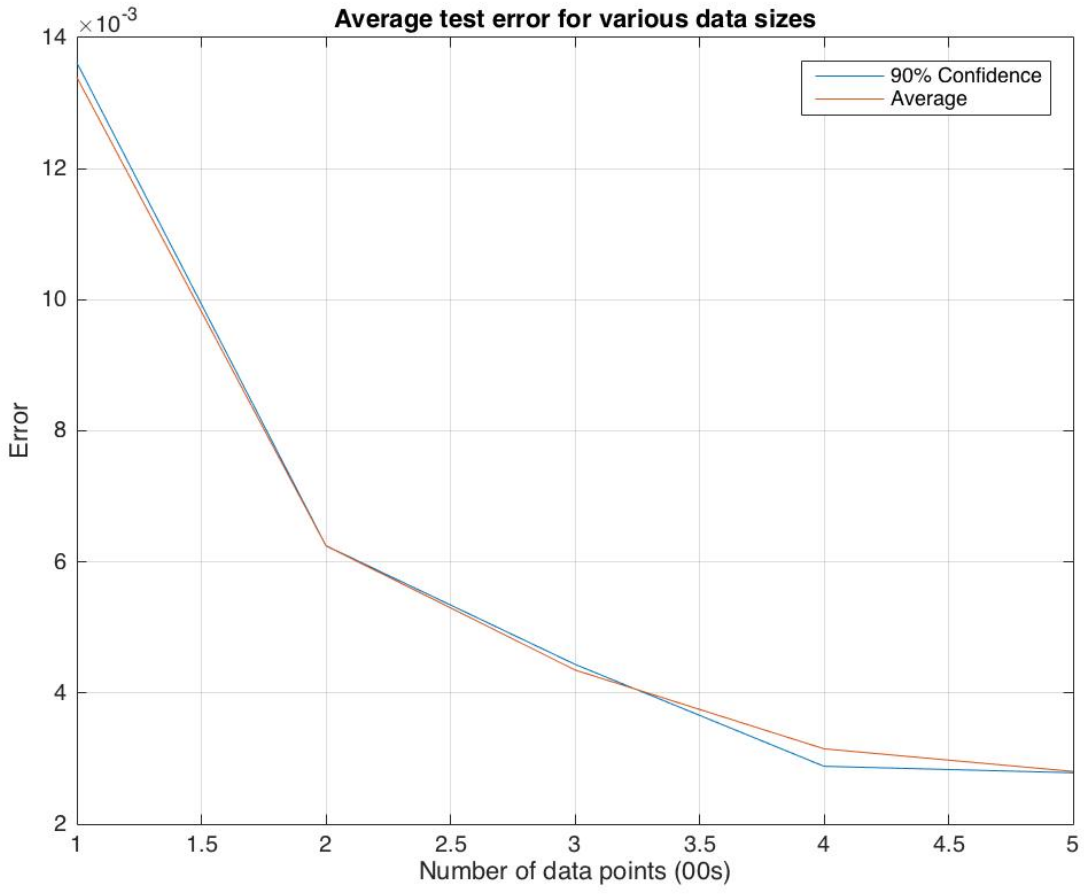


Figure 4: Average and 90% confidence test error for various data point sizes

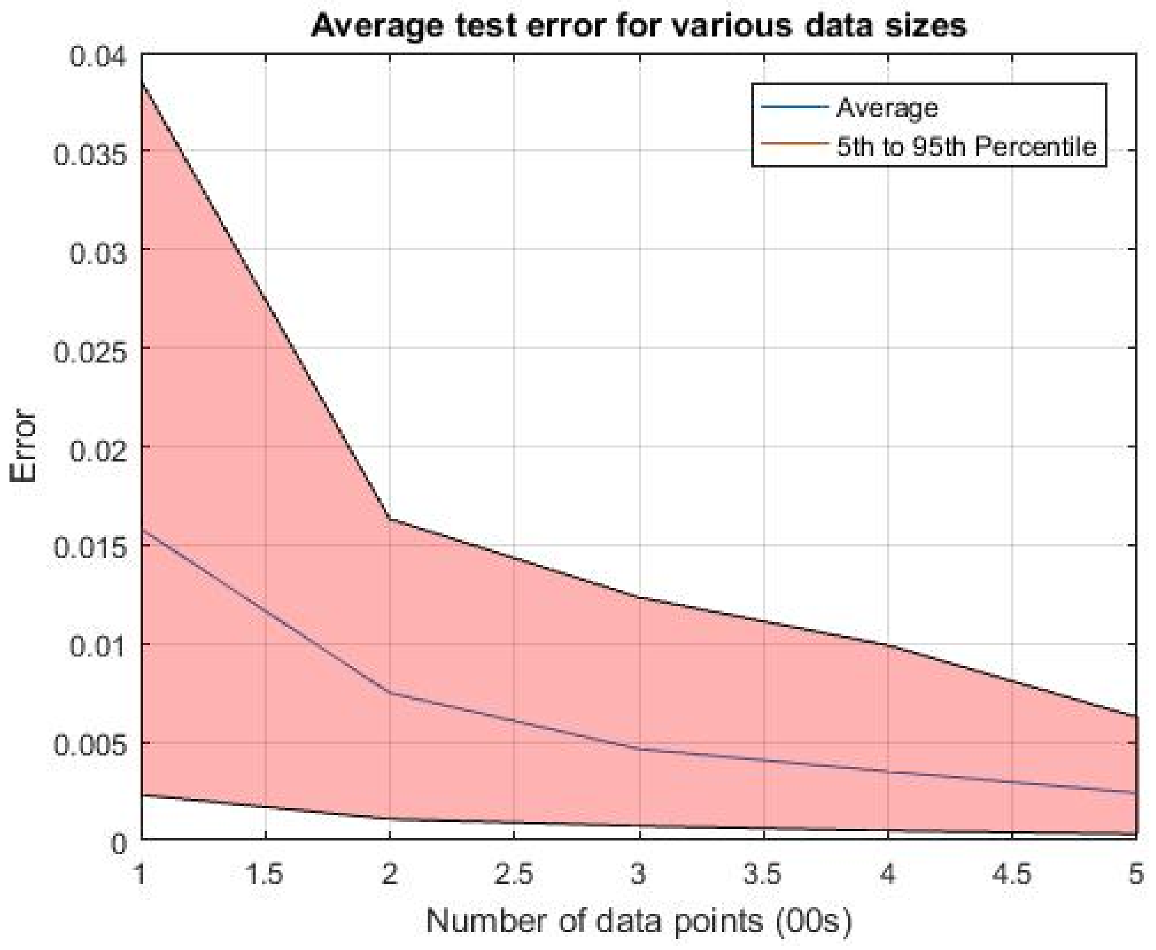


Figure 5: Average and 90% confidence interval

3.2.4 Considering cases where there is a margin between the two classes

There is often a possibility that the two classes are separated by some margin. To examine the effect of this on the Perceptron algorithm, I modify the data sets such that the two classes are separated by some margin. In this case, consider the data points such that they are still uniformly distributed over the region $K_\gamma = \{(x_1, x_2) \in [0,1]^2 : |x_2 - x_2 - 0.1| > \gamma\}$, for $\gamma = 0.3, 0.1, 0.01, 0.001$. I specifically want to examine the test error as a function of the number of data points for each of the values of the margin, and compare it to the original case where there was no margin. To accomplish this, I needed to define a function which modifies the `rand` function in MATLAB to ensure that the data points adhere to the above relationship. This was done through the following function, which produces an $n \times 3$ matrix containing n-data points with 3 elements each, where the data points abide the relationship given above:

```
function y = filterMargin(n,gamma)


    y = zeros(n,3);

    for i = 1:n
    found = false;
    while ~found
```

```
        a = [ones(1,1) rand(1,2)];
        if abs(a(3) - a(2) - 0.1) > gamma
            y(i,:) = a;
            found = true;
        end

    end
    end

end
```

I apply a similar script to that in the previous section to obtain the appropriate results.

```
%exercise 3b4
n = 100;

for margin = [0.3 0.1 0.01 0.001 0]
    Test = filterMargin(100000, margin);
    testClassify = classify(Test);

    prob = zeros(n,5);

    for ii = 1:n
        for i = 1:5
            X = filterMargin(i*100,margin);
            Y = classify(X);
            A = percep(X,Y);
            A = A/A(3);
            prob(ii,i) = errorprob(sign(A*Test').*testClassify)
        end
    end
    sort(prob);
    plot(1:5,mean(prob(6:95,1:5)))
    hold on;
end

grid on;
title('Test error vs. number of data points for various margin sizes');
xlabel('Number of data points');
ylabel('Error');
legend('Margin = 0.3','Margin = 0.1','Margin = 0.01','Margin = 0.001','Margin = 0');
```

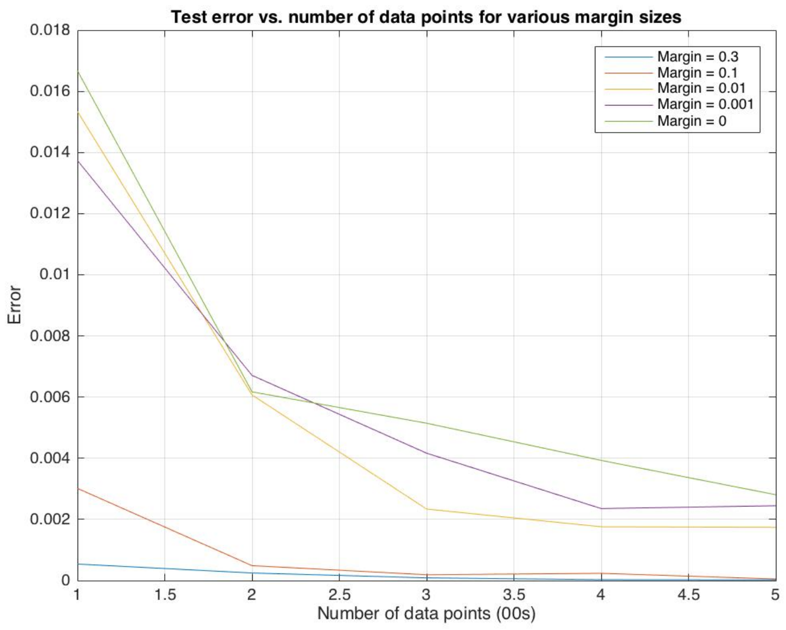


Figure 6: Comparison of the test error when data is distributed around different margins

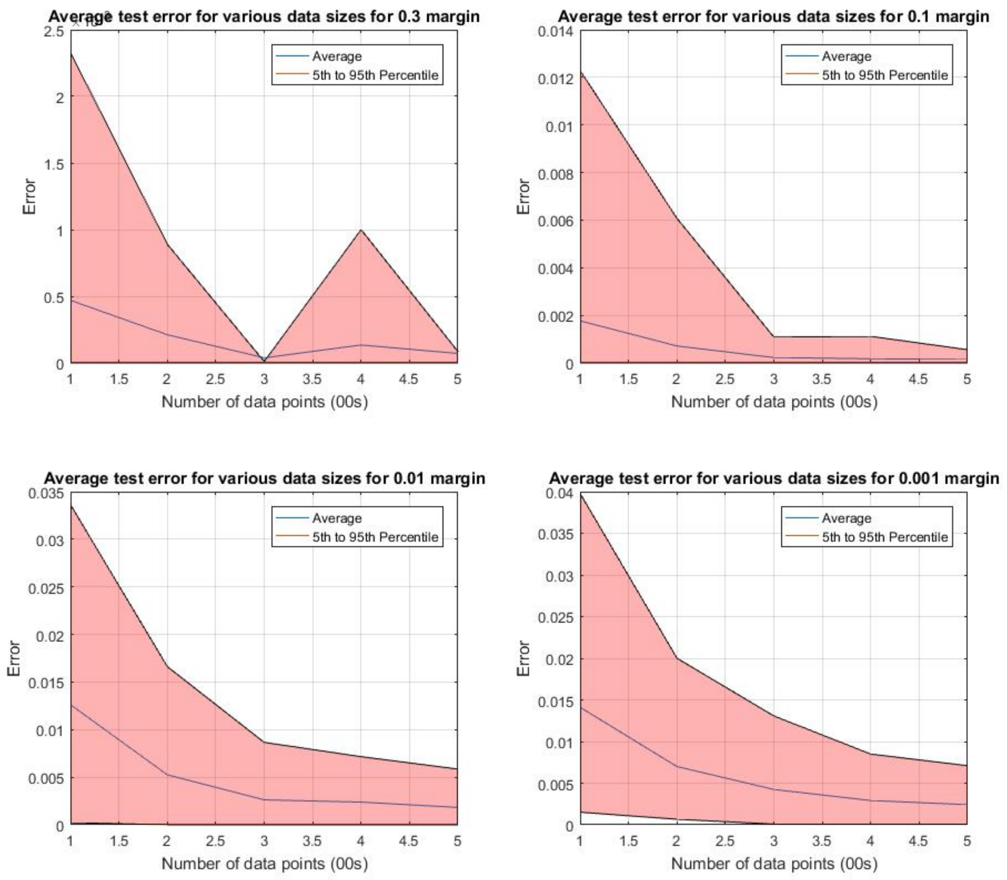


Figure 7: Comparison of the test error when data is distributed around different margins, showing average and 90% confidence interval

General trends can be inferred from the above figures. We observe again that the test error seems to decrease with the number of data points in an exponential-like fashion. More importantly, however, we find that, generally, the larger the margin of separation between the two classes, the smaller the test error for a given number of data points. Indeed we find that, in general, the margin of 0.3 manifests the smallest test error, followed by that of 0.1, and so on, with the largest test error belonging to a zero margin data set.

3.2.5 Comparing the actual number of iterations to the theoretical bound

In problem 2 of this assignment, we proved that $t \leq \frac{R^2||w^*||^2}{\rho^2}$, where t is the number of updates made to the weights vector. It is quite revealing to now compare the actual number of iterations and the theoretical bound. To explore this, I decided to compare the ratio between of actual number of iterations and the theoretical bound for data sets having different margins, specifically the set of margins introduced in the previous section. This was done by running the following script, which takes the 90% confidence interval for the 100 generated ratios:

```
%exercise 3b4 part 2
n = 100;
boundVector = [];

for margin = [0.3 0.1 0.01 0.001 0]
    Test = filterMargin(100000, margin);
    testClassify = classify(Test);

    bound = zeros(1,n);

    for ii = 1:n
        X = filterMargin(100,margin);
        Y = classify(X);
        [A,count] = theoreticalPercep(X,Y);
        for i = 1:n
            normvector(i) = norm(X(i,:));
            minvector(i) = A*X(i,:)'*Y(i);
        end

        maxNorm = max(normvector);
        minRho = min(minvector);

        upperBound = (norm(A)^2*maxNorm)/minRho^2;
        bound(ii) = count/upperBound;
    end
    sort(bound);
    boundVector = horzcat(boundVector,[mean(bound(6:95))]);
    hold on;
end

plot([0.3 0.1 0.01 0.001 0],boundVector)

grid on;
title('number of iterations to upper bound ratio for various margin sizes');
xlabel('Margin size');
ylabel('Ratio');
```

The resulting output plot is also shown overleaf and reveals a common trend. Firstly, we notice that the ratios are all significantly smaller than 1, which indeed confirms the the upper bound established by the second problem. Moreover, there seems to be a relationship between this ratio and the size of the margin that separates the two classes, for I find that the larger the margin size, the higher the ratio.

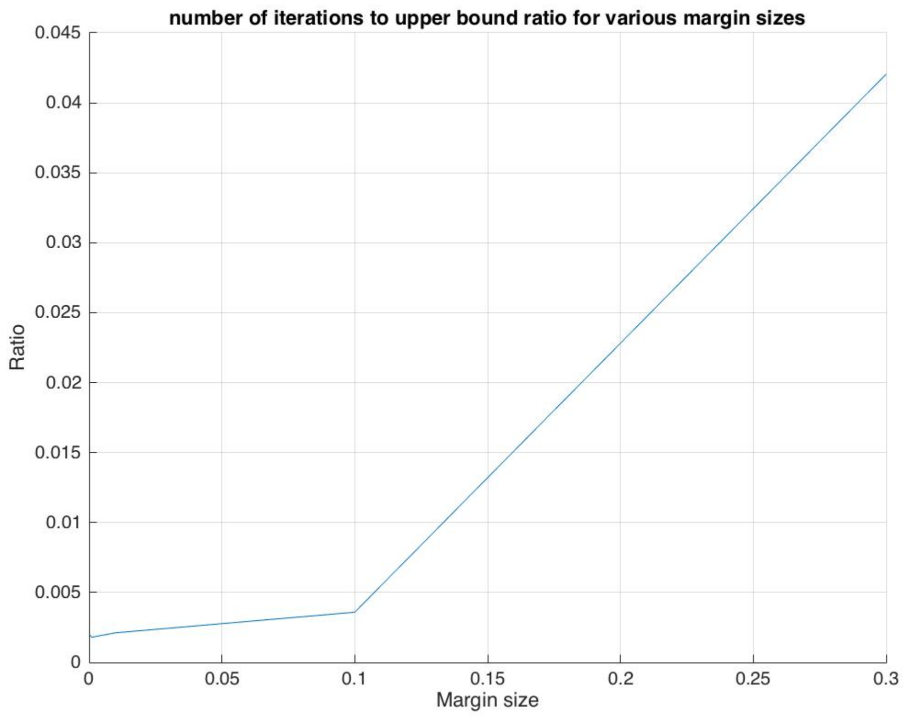


Figure 8: Ratio of actual number of iterations to theoretical bound for 5 different margin sizes

However, this investigation was conducted only with 5 margin sizes, and so is quite lacking. Thus, it was redone with far more margin sizes, specifically on an array containing margin sizes ranging from 0 to 0.3 in increments of 0.02. The resulting plot is shown blow.

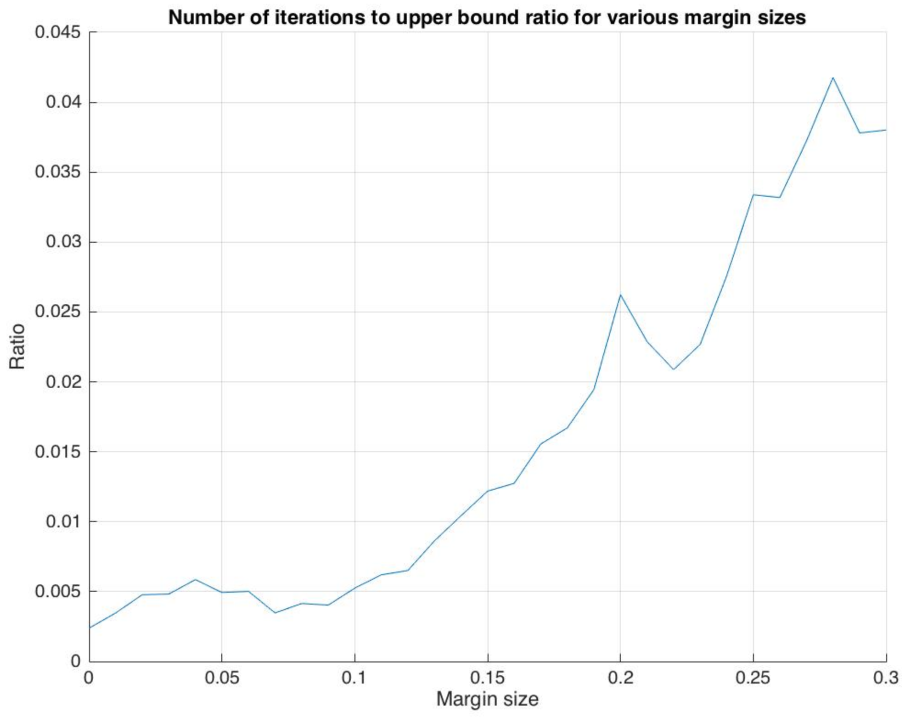


Figure 9: Ratio of actual number of iterations to theoretical bound for more margin sizes

3.2.6 Producing different weight vectors

On the same data set, the algorithm as it currently is will produce the same weights vector if run multiple times, despite the fact that there may exist infinitely many. A way to maneuver around this issue is to change the weight vector's initialization at the beginning of the program. When the weights are initialized to random numbers, this can produce varying weight vectors as a final result, all of which achieve a zero training error if the data set is linearly separable. Using a new function, `randPercep`, which initializes the weights randomly, two classification lines that were produced based on a randomly generated data set of 100 points were $x_2 = 0.9952x_1 + 0.1146$ (`w* = [-0.1146 -0.9952 1]`) & $x_2 = 0.9596x_1 + 0.1214$ (`w* = [-0.1214 -0.9596 1]`).

```
function [w,count] = randPercep(X,Y)

w = rand(1,size(X,2)); %Weights vector initialization
count = 0;
correct = false; % boolean variable will control loops

while ~correct
    for ii = 1 : size(X,1)

        if sign(X(ii,:)*w') ~= Y(ii) %Check if the classification is right
            w = w + X(ii,:) * Y(ii);
            count = count+1;
        end
        correct = errorprob(sign(w*X').*Y) == 0; %if not, we start again

    end
end
```

```
w= w/w(3);

end
```

3.3 Part C

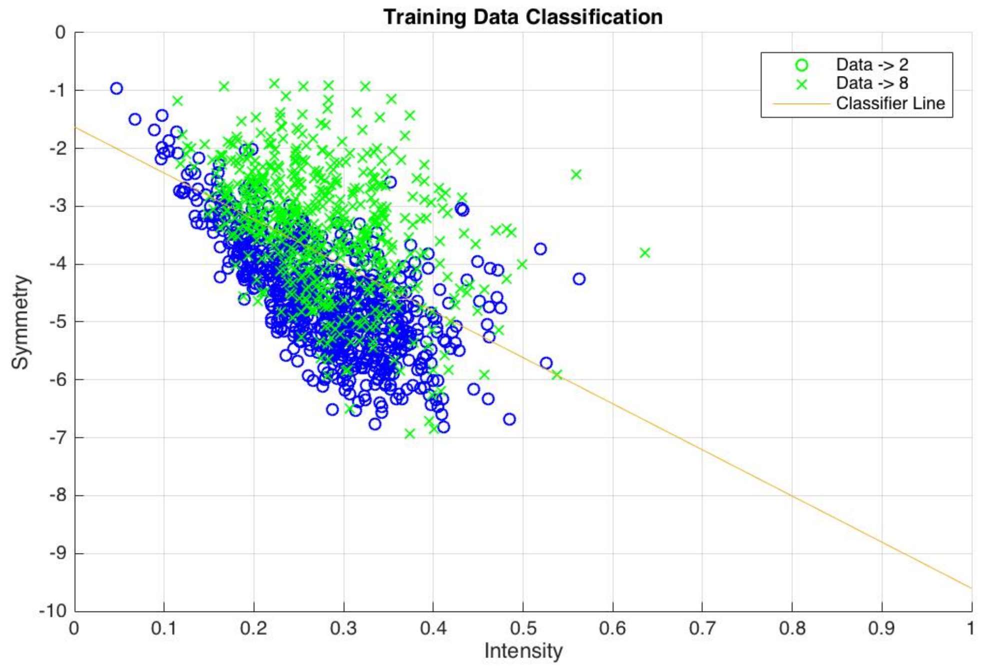
The final part of the third problem covers the aspect of employing the algorithm on a real data set to actually learn. This will be done based on data representing handwritten numbers, represented in two ways: as 256 grey-scale values and as 2-element data points consisting of intensity and symmetry measurements.

3.3.1 Applying Perceptron algorithm on the training data

Although the exercises in the previous part have highlighted the correct functionality of my algorithm, there is a major fault with it, in that it is built upon the assumption that the data is always linearly separable. In reality, this is unlikely to be the case and as a result the alogrithm as it is now would never terminate. It is thus necessary to make a modification to it, by setting a predefined maximum bound for the number of updates the algorithm is allowed to make to the weights vector. The new Perceptron algorithm, `modifiedPercep`, is define below, where `n` is the bound on the number of iterations:

```
function [w] = modifiedPercep(X,Y,n)

w = zeros(1,size(X,2)); %Weights vector initialization

for i = 1:n
    for ii = 1 : size(X,1)

        if sign(X(ii,:)*w') ~= Y(ii) %Check if the classification is right
            w = w + X(ii,:) * Y(ii);
        end

    end
end

end
```

Running this on the training data set, we obtain a weights vector that correctly classifies every point in the training set, indicating that the training data is actually linearly separable. Please note that I employ two functions, `classify2`, `filter1`, `filter2` & `filter8`, which can be found in the Appendix. The aforementioned result entails that our training error here is 0%, and the program returns a test error of 5.49%.
When applying a similar procedure to the 2-D sets, we obtain different results, predominantly because the training data here is not linearly separable. This can be seen in the overleaf figure, noting that the data points overlap and can thus not be separated. I set the limit on the number of iterations of the algorithm to 1000, as there is no substantial reduction in the test and error when the number of iterations is exceeded. In fact, we obtain a training error of 22.94% and the test error of 28.85%.



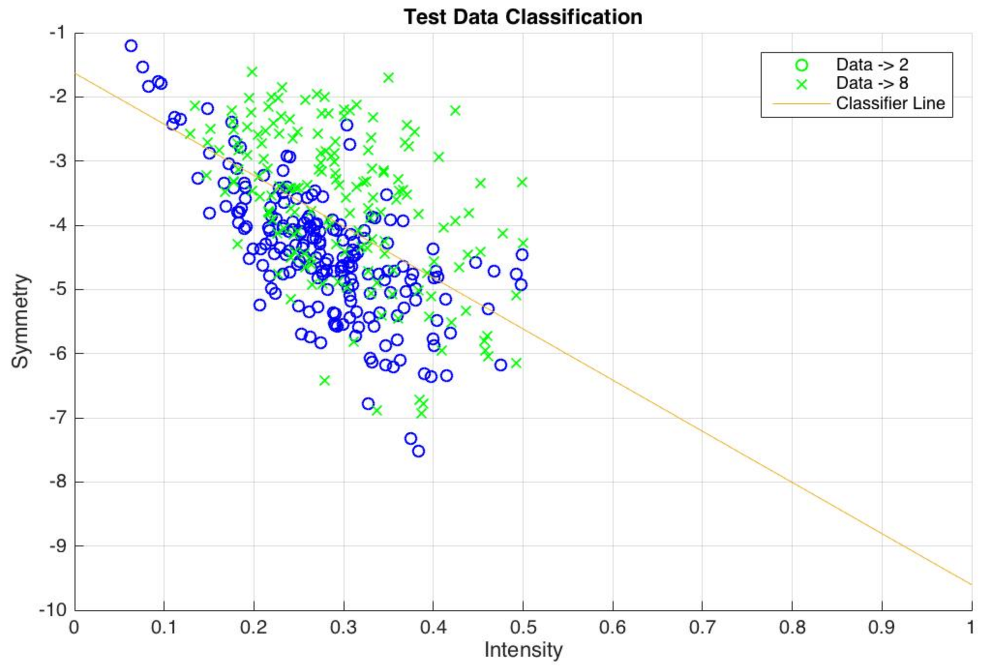


Figure 10: The two plots show the training and test data, along with the linear classifier computed by the algorithm

The script that generates the error probabilities and the above plots is included here:

```
%Exercise 3.c.1

unfilteredTraining256 = importdata('zip.train');
filteredTraining256 = filter1(unfilteredTraining256);
unfilteredTest256 = importdata('zip.test');
filteredTest256 = filter1(unfilteredTest256);
trainOut256 = classify2(filteredTraining256);
testOut256 = classify2(filteredTest256);
trainIn256 = [ones(1273,1) filteredTraining256(:,2:257)];
testIn256 = [ones(364,1) filteredTest256(:,2:257)];
weights256 = modpercep(trainIn256,trainOut256,1000);
errorprob(sign(weights256*trainIn256').*trainOut256)
errorprob(sign(weights256*testIn256').*testOut256)

unfilteredTraining2 = importdata('features.train');
filteredTraining2 = filter1(unfilteredTraining2);

train2 = filter2(filteredTraining2);
train8 = filter8(filteredTraining2);

subplot(2,1,1)
plot1 = scatter(train2(:,2),train2(:,3),'bo');
hold on;
scatter(train8(:,2),train8(:,3),'gx');
hold on;

unfilteredTest2 = importdata('features.test');
filteredTest2 = filter1(unfilteredTest2);
trainOut2 = classify2(filteredTraining2);
testOut2 = classify2(filteredTest2);
trainIn2 = [ones(1273,1) filteredTraining2(:,2:3)];
testIn2 = [ones(364,1) filteredTest2(:,2:3)];
weights2 = modpercep(trainIn2,trainOut2,1000);
weights2_norm = weights2/weights2(3);

plot(0:0.1:1,(-weights2_norm(2)*[0:0.1:1]' - weights2_norm(1)))
grid on;
title('Training Data Classification');
xlabel('Intensity');
ylabel('Symmetry');
legend('Data -> 2','Data -> 8','Classifier Line')

hold off;

test2 = filter2(filteredTest2);
test8 = filter8(filteredTest2);

subplot(2,1,2)
plot2 = scatter(test2(:,2),test2(:,3),'bo');
hold on;
scatter(test8(:,2),test8(:,3),'gx');
hold on;

plot(0:0.1:1,(-weights2_norm(2)*[0:0.1:1]' - weights2_norm(1)))
grid on;
title('Test Data Classification');
xlabel('Intensity');
ylabel('Symmetry');
```

```
legend('Data -> 2','Data -> 8','Classifier Line')

subplot()


errorprob(sign(weights2*trainIn2').*trainOut2)
errorprob(sign(weights2*testIn2').*testOut2)
```

3.3.2 Further exploring the classification error

It becomes quite useful to note the changes in the training & test errors of the two data representations as a function of the number of iterations of the algorithm.

I begin by examining this for the 256 data point set. As stated in the previous section, the algorithm attains a zero-valued training error for the given training data, while resulting in a test error of around 0.0549. It is thus imperative to note the variation of these quantities with the number of iterations of the algorithm. This can be done through the following script:

```
%Exercise 3.c.2

p_training256 = zeros(1,35);
p_test256 = zeros(1,35);

unfilteredTraining256 = importdata('zip.train');
filteredTraining256 = filter1(unfilteredTraining256);
unfilteredTest256 = importdata('zip.test');
filteredTest256 = filter1(unfilteredTest256);
trainOut256 = classify2(filteredTraining256);
testOut256 = classify2(filteredTest256);
trainIn256 = [ones(1273,1) filteredTraining256(:,2:257)];
testIn256 = [ones(364,1) filteredTest256(:,2:257)];

for i = 1:35
    weights256 = modpercep(trainIn256,trainOut256,i);
    p_training256(i) = errorprob(sign(weights256*trainIn256').*trainOut256)
    p_test256(i) = errorprob(sign(weights256*testIn256').*testOut256)
end

plot(1:35,p_training256)
hold on;
plot(1:35, p_test256)
title('Training and Test errors for 256-D data')
grid on;
xlabel('Number of iterations');
ylabel('Error');
legend('Training Error','Test Error');
```

Running this script, we obtain the following plot, which shows a trend. As may be expected, the training error gradually decreases as the number of iterations increases, and eventually becomes zero, indicated that the training data set is linearly separable. The test error, however, seems to oscillate around a fixed level, approximated to 0.0542 using MATLAB's `mean` function. The new algorithm, given a maximum bound for the number of iterations of 35, runs for 30 rounds before converging on a fixed training and test error. The original algorithm, although not given a maximum bound, also runs for 30 rounds before coming to a halt.

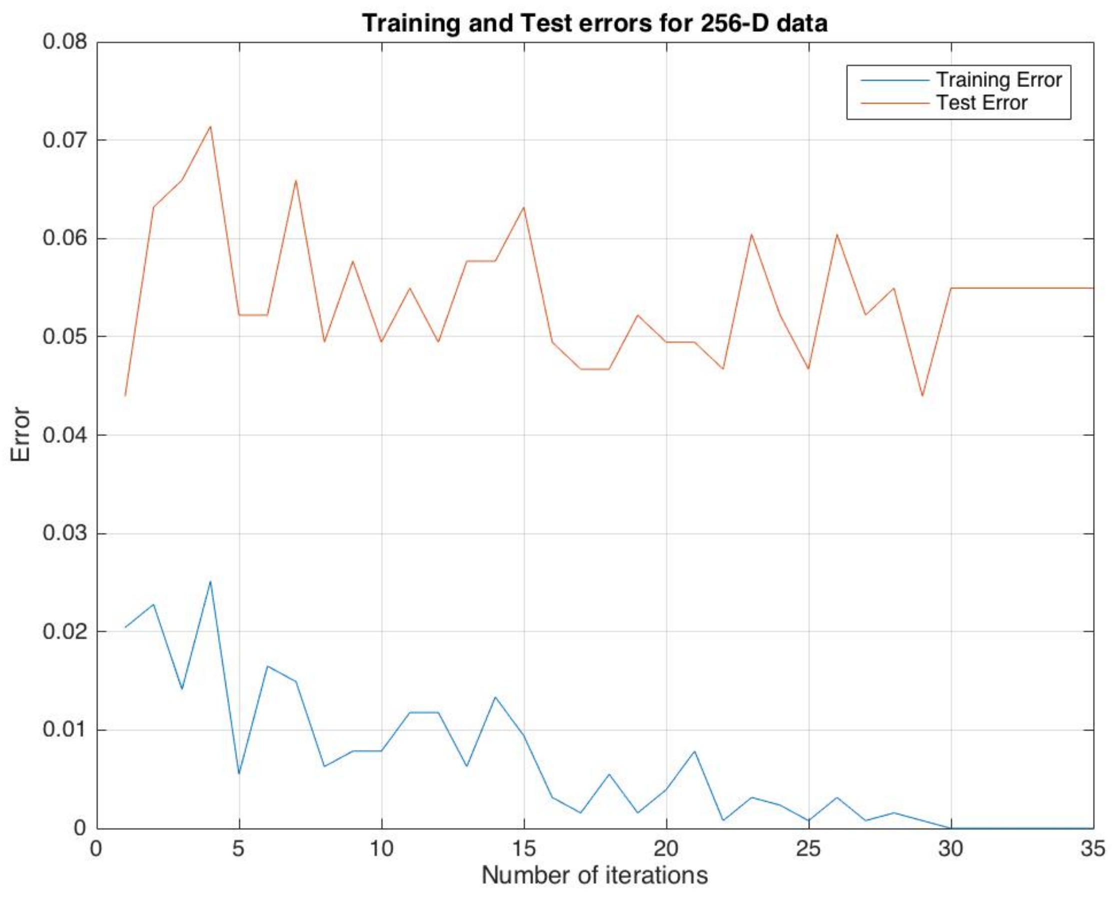


Figure 11: The variation between the training and test errors with the number of iterations of the algorithm for 256-D data set

It is beneficial to apply the same process to the 2-D data set. Again, this is done through a similar script:

```
n = size([10:10:1000],2);
p_training2 = zeros(1,n);
p_test2 = zeros(1,n);

unfilteredTraining2 = importdata('features.train');
filteredTraining2 = filter1(unfilteredTraining2);
unfilteredTest2 = importdata('features.test');
filteredTest2 = filter1(unfilteredTest2);
trainOut2 = classify2(filteredTraining2);
testOut2 = classify2(filteredTest2);
trainIn2 = [ones(1273,1) filteredTraining2(:,2:3)];
testIn2 = [ones(364,1) filteredTest2(:,2:3)];

for i = 10:10:1000
    weights2 = modpercep(trainIn2,trainOut2,i);
    p_training2(i/10) = errorprob(sign(weights2*trainIn2').*trainOut2)
    p_test2(i/10) = errorprob(sign(weights2*testIn2').*testOut2)
end

plot(10:10:1000,p_training2)
hold on;
```

```
plot(10:10:1000, p_test2)
hold on;
title('Training and Test errors for 2-D data')
grid on;
xlabel('Number of iterations');
ylabel('Error');
legend('Training Error','Test Error');
```

There generated plot is shown below. Interestingly here, we notice that neither the test nor training errors converge to a constant value but instead oscillate about some average value, approximated to 0.2391 for the training error and 0.3030. The minimum obtained values for training and test errors were 0.2262 & 0.2857, respectively. Please note here that the number of iterations was incremented by 10 each time, thus, the values for some intermediate number of iterations are omitted. Importantly, and in contrast to the results obtained for the 256-D data set, the original algorithm cannot be used on this data set, because the two classes are not linearly separable, and consequently the algorithm would never terminate.

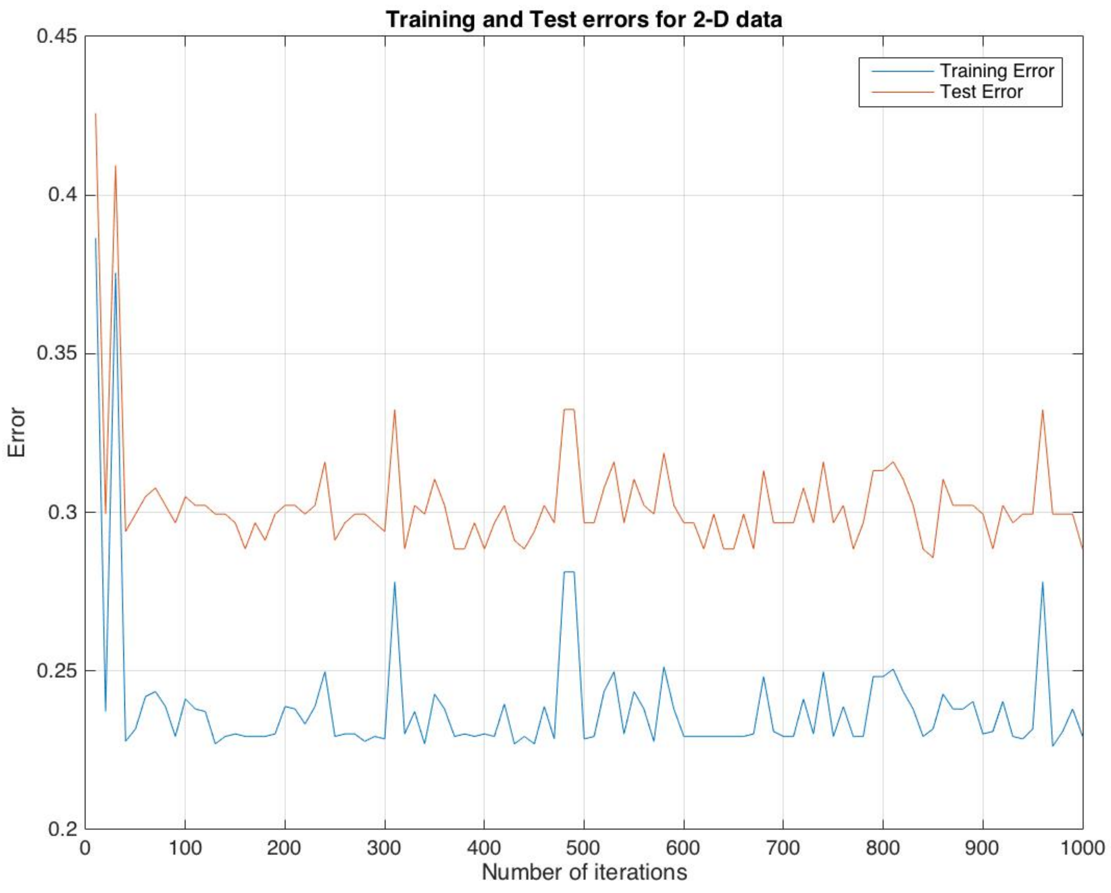


Figure 12: The variation between the training and test errors with the number of iterations of the algorithm for 2-D data set

3.3.3 Employing regression to initialize the weights vector

We employ linear regression here to initialize the weights vector in the Perceptron algorithm and we proceed from there as before. The optimal linear regression weights can be computed through the pseudoinverse, X^+, of the input data, where $X^+ = (X^\top X)^{-1}X^\top$. The optimal linear regression weights are then $w = X^+Y$. A modified Perceptron algorithm, `regresPercep`, can be found in the Appendix. To test the performance of the modified Perceptron algorithm which uses regression, I decided to plot the linear classifier of the previously modified algorithm and the new one applied to the 2-D data. The following plot is obtained. We can note that the two generate classifiers are quite similar from a qualitative perspective.

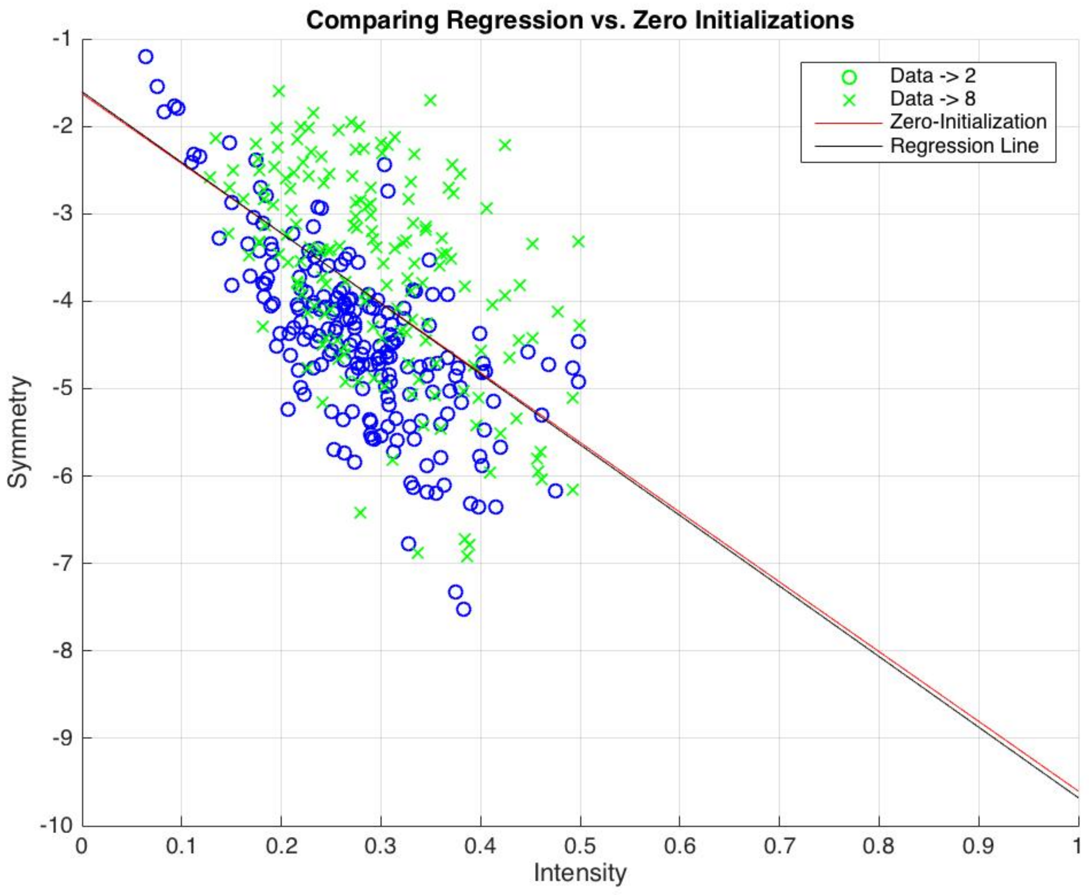


Figure 13: Comparison of the two linear classifiers applied to the 2-D data set

It is perhaps more quantitatively rigorous to examine the training and test error of the new algorithm as a function of the number of iterations of the algorithm as we did in the previous section. The following script was used to conduct the comparison and generate the overleaf plot.

```
%exercise3c3 part 2

n = size([10:10:1000],2);
p_training2 = zeros(1,n);
p_test2 = zeros(1,n);
p_training2r = zeros(1,n);
p_test2r = zeros(1,n);

unfilteredTraining2 = importdata('features.train');
filteredTraining2 = filter1(unfilteredTraining2);
unfilteredTest2 = importdata('features.test');
```

```
filteredTest2 = filter1(unfilteredTest2);
trainOut2 = classify2(filteredTraining2);
testOut2 = classify2(filteredTest2);
trainIn2 = [ones(1273,1) filteredTraining2(:,2:3)];
testIn2 = [ones(364,1) filteredTest2(:,2:3)];

for i = 10:10:1000
    weights2 = modpercep(trainIn2,trainOut2,i);
    weights2r = regresPercep(trainIn2,trainOut2,i);
    p_training2(i/10) = errorprob(sign(weights2*trainIn2').*trainOut2)
    p_test2(i/10) = errorprob(sign(weights2*testIn2').*testOut2)
    p_training2r(i/10) = errorprob(sign(weights2r'*trainIn2').*trainOut2)
    p_test2r(i/10) = errorprob(sign(weights2r'*testIn2').*testOut2)
end

plot(10:10:1000,p_training2, 'r--')
hold on;
plot(10:10:1000, p_test2,'r')
hold on;
plot(10:10:1000,p_training2r,'b--')
hold on;
plot(10:10:1000, p_test2r, 'b')
title('Training and Test errors for 2-D data')
grid on;
xlabel('Number of iterations');
ylabel('Error');
legend('Training Error (Zero)','Test Error (Zero)','Training Error (Regression)','Test Err
```

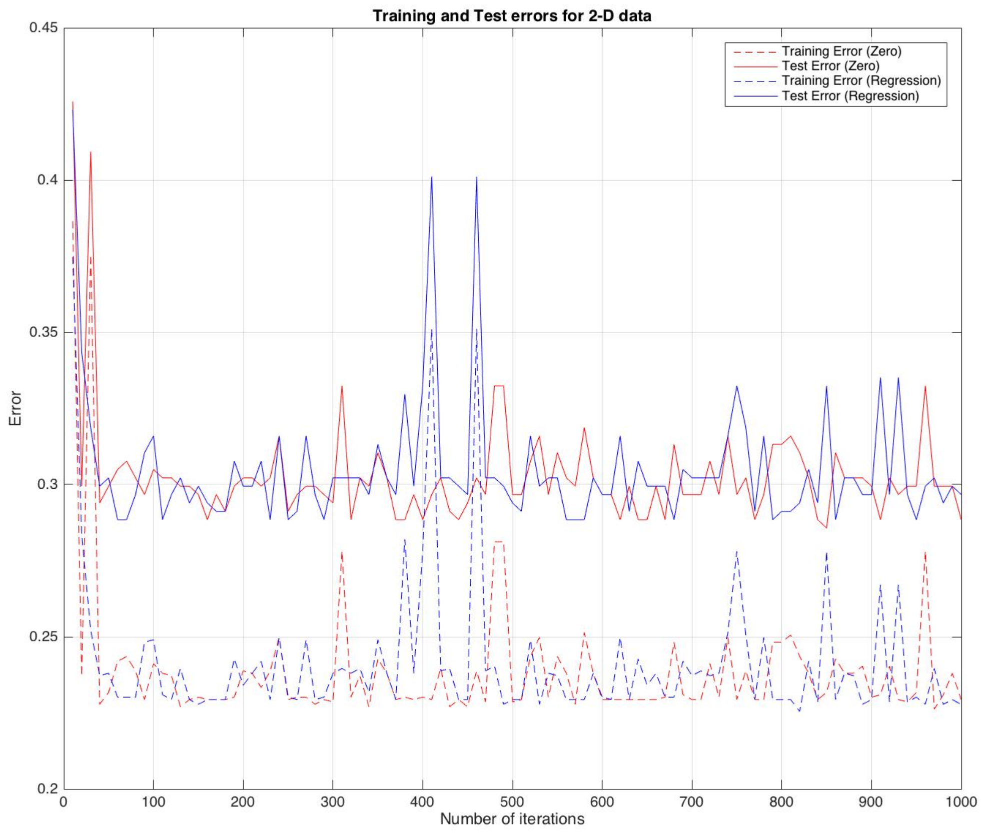


Figure 14: Comparison of the error of the two linear classifiers applied to the 2-D data set

The dashed lines indicate training error and the solid lines indicates test error. The red lines represent the zero-initialized Perceptron, while the blue lines highlight the regression-initialized Perceptron. We can see that there is not a clear verdict over which method is better, as it depends on the number of iterations of the algorithm. Specifically, if we examine these results with respect to the lowest error, then we find out that the zero-initialized algorithm provides the lowest test error of 0.2857, while the regression-initialized version produces the lower training error of 0.2255.

3.3.4 Upper bound on the difference between the empirical and ideal test errors

Over parts B & C, we repeatedly use test data sets to assess the test performance of the algorithm. In reality, this is just an approximation for the true test error, since we do not actually know the true distribution of data points. It is thus imperative to assess how close the empirical test error is to the true figure. This can be achieved by appling Hoeffding's inequality as follows.

$$P(|\hat{R_n}(h) - R(h)| > \epsilon) \leq 2e^{-2\epsilon^2 n} \qquad \text{Hoeffding's Inequality}$$

We would like a high-probability upper bound for the difference between the actual and ideal test errors. In my interpretation, I will take high-probability to mean a 90% confidence interval. Also, filtering the test data set for the 2 & 8 classes, we note that there are 364 test data points, i.e. $n = 364$. The inequality can now be employed where $\hat{R_n}(h)$ is the empirical test error and $R(h)$ is the true test error.

$$P(|\hat{R_n}(h) - R(h)| > \epsilon) \leq 2e^{-2\epsilon^2 n} = 0.1 \qquad \text{Assuming confidence interval of 90\%}$$

Having a value for n, we can now solve the following for ϵ.

$$\begin{aligned} 2e^{-2\epsilon^2 n} &= 0.1 \\ &\downarrow \\ 0.05 &= e^{-2\epsilon^2 n} \\ \epsilon &= \sqrt{\frac{\ln(0.05)}{-2 \times 364}} \\ \epsilon &\approx 0.064 \end{aligned}$$

4 Appendix

`simpleClassify` function

```
function [y] = simpleClassify(x)

    for i = 1:size(x,1)

        if x(i,3) > 0.5
            y(i) = 1;
        else
            y(i) = -1;
        end

    end

end
```

`classify` function

```
function [y] = classify(x)

    for i = 1:size(x,1)
        if x(i,3) >= x(i,2) + 0.1
            y(i) = 1;
        else
            y(i) = -1;

        end
    end
end
```

`classify2` function

```
function [y] = classify2(x)

    for i = 1:size(x,1)
        if x(i,1) == 2
            y(i) = 1;
        else
            y(i) = -1;

        end
    end
end
```

`filter1` function

```
%This is a function that takes the whole unfiltered trainind data set and
%return only those elements that characterize either 2 or 8

function [y] = filter1(x)

    y = [];
    for i = 1:size(x,1)

        a = x(i,:);
        if (a(1) == 2) || (a(1) == 8)
            y = vertcat(y,a);
        end

    end

end
```

`filter2` function

```
%This is a function that takes the whole unfiltered trainind data set and
%return only those elements that characterize 2

function [y] = filter2(x)

    y = [];

    for i = 1:size(x,1)

        a = x(i,:);
        if (a(1) == 2)
            y = vertcat(y,a);
        end

    end

end
```

`filter8` function

```
%This is a function that takes the whole unfiltered trainind data set and
%return only those elements that characterize 8

function [y] = filter8(x)

    y = [];

    for i = 1:size(x,1)

        a = x(i,:);
        if (a(1) == 8)
            y = vertcat(y,a);
        end

    end

end
```

`regresPercep` function

```
function [w] = regresPercep(X,Y,n)
```

```
w = pinv(X)*Y'; %Weights vector initialization

for i = 1:n %setting a defined number of iterations

    for ii = 1 : size(X,1)

        if sign(X(ii,:)*w) ~= Y(ii) %Check if the classification is right
            w = w + X(ii,:)' * Y(ii);   %then add (or subtract) this point to w
        end

end

end
```