Machine Learning Coursework 1

I, Weichuan Yin, pledge that this assignment is completely my own work, and that I did not take, borrow or steal work from any other person, and that I did not allow any other person to use, have, borrow or steal portions of my work. I understand that if I violate this honesty pledge, I am subject to disciplinary action pursuant to the appropriate sections of Imperial College London.

Problem 1: Confidence level to reject the hypothesis that the probability of choosing to stay in the EU is at least 50%.

Answer:

The given conditions in this problem and their corresponding learning components are the following:

1. 55% of people supported leaving and 45% of people supported staying in the public poll: These corresponds to the in-sample data;
2. They ask 2052 people: Therefore N, the size of training sample is 2052;
3. Public poll is more or less the equivalent of training set, and the official Brexit is the testing set.

Choosing to stay in the EU means that the percentage of people will vote leaving in the official Brexit is less than 50% (or equally, the percentage of people will vote staying in the official Brexit is more than 50%). Therefore we can use Hoeffding's Inequality to make the prediction of the result of the official Brexit with some confidence levels. The hint given in the coursework sheet is the general form of the Hoeffding's Inequality, since we are only dealing with the binary result (leaving/stay) in this problem, we would not need this general form.

The Hoeffding's Inequality given in the lecture is $P\ (|Rn(h) - R(h)| > \varepsilon) \leq 2\exp(-2\varepsilon^2 * n)$, where $Rn(h)$the training/ empirical information we obtain from the training set, which refers to the public poll in this case. $R(h)$ Is the test information and it is out-of-sample, therefore it is the people's choice in the official Brexit.

Firstly, I notice this inequality is two-sided, it has a modulus on the left-hand-side. Thus a one-sided version can be obtained by getting rid of the modulus on the left-hand-side and divide the right-hand-side by 2. The one-sided Hoeffding's Inequality is$P\ (Rn(h) - R(h) > \varepsilon) \leq \exp(-2\varepsilon^2 * n)$. With a simple rearranging, it then becomes$P\ (R(h) < Rn(h) - \varepsilon) \leq \exp(-2\varepsilon^2 * n)$. As just mentioned, $R(h)$ is the out-of-sample information, therefore $R(h)$ is set to be the percentage of people will vote leave in the official Brexit (because it is followed by a less or equal sign). And $Rn(h)$ becomes the percentage of people voting leave in the public poll, which is 0.55(55%).

For predicting the probability staying in the EU, we need $R(h)$ to be less than 0.5(50%), so we set ε to be 0.05 (5%) and now the inequality is complete:

$$P\ (R(h) < Rn(h) - \varepsilon) \leq \exp(-2\varepsilon^2 * n)$$

$Rn(h) = 0.55$, percentage of people voted leaving in the public poll is 55%;

$\varepsilon = 0.05$, the bias is 0.05;

$n = 2052$; The sample data size (public poll population) is 2052.

And we get:

$$P(R(h) < 0.55 - 0.05) \leq \exp(-2 * 0.05^2 * 2052)$$

$$P(R(h) < 0.5) \leq 3.5 * 10^{-5}$$

On the left-hand-side, it is the confidence level of agreeing with the hypothesis that majority of population will vote staying in the official Brexit. And this confidence level is less than$3.5 * 10^{-5}$, which is almost none.

Therefore the conclusion can be made. The confidence level to reject the hypothesis that majority of population will vote staying in the official Brexit is greater than$(1 - 3.5 * 10^{-5})$, which is almost 100%. This conclusion actually can be guessed from the given conditions, since in the public vote with the sample size of 2052, there are already 55% people choose to leave. So we expect the official result from the Brexit to be leaving.

Problem 2: Convergence of the Perceptron Learning Algorithm. (See scanned PDF)

Problem 2: Convergence of the Perceptron Learning Algorithm.

- Given n data points $\mathbb{D} = \{(x^{(1)}, y^{(1)}), \cdots, (x^{(n)}, y^{(n)})\}$. where $x^{(i)} \in \mathbb{R}^{d+1}$

- $y^{(i)} \in \{1, -1\},\ i = 1, 2, \cdots, n$

- There exists $w^* \in \mathbb{R}^{d+1}$, $\text{sign}(W_*^T x^{(i)}) = y^{(i)}$ for all $i \in \{1, \cdots, n\}$.

(a) Define $\rho = \min_{i \in \{1, \cdots, n\}} y^{(i)} w_*^T x^{(i)}$. show $\rho > 0$.

Since there exists w^* S.T $y^{(i)} = \text{sign}(w_*^T \cdot x^{(i)})$.

This means $\begin{cases} \text{if } W_*^T x^{(i)} \geq 0, & y^{(i)} = 1 \\ \text{if } W_*^T x^{(i)} < 0, & y^{(i)} = -1. \end{cases}$

$\therefore y^{(i)} / W_*^T x^{(i)}$ are always of the same sign.

$\therefore y^{(i)} W_*^T x^{(i)} > 0$

$\therefore \min_{i \in \{1, \cdots, n\}} y^{(i)} W_*^T x^{(i)} > 0$

$\therefore \rho > 0$

(b) Show that for any t before the algorithm stops, $W_*^T W_t \geq t\rho$

Since $W_t = W_{t-1} + y_{t-1} X_{t-1}$.

$\therefore W_*^T W_t = W_*^T W_{t-1} + y_{t-1} W_*^T X_{t-1}$ (dot product W_*^T on the left)

And since $y_{t-1} W_*^T X_{t-1} \geq \rho$ ($\because \rho = \min_{i \in [1, \cdots, n]} y^{(i)} W_*^T X^{(i)}$).

$\therefore W_*^T W_t \geq W_*^T W_{t-1} + \rho$

Induction: assume the expression is true for $t-1$.

$\therefore$ assume $W_*^T W_{t-1} \geq (t-1)\rho$

$\because$ just proven: $W_*^T W_t \geq W_*^T W_{t-1} + \rho$

$\therefore W_*^T W_t \geq (t-1)\rho + \rho$

$\therefore W_*^T W_t \geq t\rho$.

Then try $t=1$.

$W_1 = W_0 + y_0 X_0$, $W_0 = 0$.

$= y_0 X_0$.

$\therefore W_*^T W_1 = y_0 W_*^T X_0 \geqslant \rho$

$\therefore W_*^T W_1 \geqslant 1 \times \rho$ (v) $\rightarrow$ expression is true for $t=1$.

$\therefore \because$ expression is true if $\begin{cases} \text{expression is true for } t-1. \\ \text{expression is true for } t=1. \end{cases}$

$\therefore W_*^T W_t \geq t\rho$ holds for $\in t$.

(c) Show that for any t before the algorithm stops, $\|W_t\|^2 \leq tR^2$, where $R = \max_{i\in\{1,\dots,n\}} \|x^{(i)}\|$.

$\because \begin{cases} \|W_t\|^2 = W_t^T W_t. & \text{-by definition} \\ W_t = W_{t-1} + y_{t-1}X_{t-1} & \text{- given in the question.} \end{cases}$

$$\therefore \|W_t\|^2 = (W_{t-1} + y_{t-1}X_{t-1})^T (W_{t-1} + y_{t-1}X_{t-1})$$

$$= (W_{t-1}^T + y_{t-1}X_{t-1}^T)(W_{t-1} + y_{t-1}X_{t-1})$$

$$= W_{t-1}^T W_{t-1} + y_{t-1} W_{t-1}^T X_{t-1} + y_{t-1} X_{t-1}^T W_{t-1} + X_{t-1}^T X_{t-1}$$

$$= \|W_{t-1}\|^2 + \|X_{t-1}\|^2 + y_{t-1} W_{t-1}^T X_{t-1} + y_{t-1} X_{t-1}^T W_{t-1}.$$

$\because X_{t-1}$, W_{t-1} are both column vectors.

$\therefore W_{t-1}^T X_{t-1} = X_{t-1}^T W_{t-1}$.

$\because \text{Sign}(W_{t-1}^T x_{t-1}) \neq y_{t-1}$

$\therefore \underset{<0}{y_{t-1} W_{t-1}^T x_{t-1}} = \underset{<0.}{y_{t-1} x_{t-1}^T W_{t-1}}$

$\therefore \|W_t\|^2 \leq \|W_{t-1}\|^2 + \|x_{t-1}\|^2$

Induction: assume the expression is true for $t-1$.

$\therefore \|W_{t-1}\|^2 \leq (t-1)R^2$

Try W_t

$\because$ just proven: $\|W_t\|^2 \leq \|W_{t-1}\|^2 + \|x_{t-1}\|^2$

$\therefore \|W_t\|^2 \leq (t-1)R^2 + \|x_{t-1}\|^2$

$\because \|x_{t-1}\|^2 \leq R^2$

$\therefore \|W_t\|^2 \leq (t-1)R^2 + R^2$

$\therefore \|W_t\|^2 \leq tR^2.$

Try $\|W_1\|^2 \leq R^2$ (V). — try for $t=1$.

$\therefore \|W_t\|^2 \leq tR^2$ is true for $\in t$.

(d) Show for any t before the algorithm stops, $t \leq \frac{R^2 \|W_*\|^2}{\rho^2}$, and conclude that the algorithm stops after at most $\frac{R^2 \|W_*\|^2}{\rho^2}$ updates.

$\because$ Cauchy-Schwarz Inequality: $W_*^T W_t \leq \|W_*\| \cdot \|W_t\|$

$\therefore$ proven in (b). $t\rho \leq W_*^T W_t$

$\therefore t\rho \leq \|W_*\| \|W_t\|$

$\therefore t^2\rho^2 \leq \|W_*\|^2 \cdot \|W_t\|^2$ (square both sides).

$\because$ Proven in (c). $\|W_t\|^2 \leq tR^2$

$\therefore t^2\rho^2 \leq \|W_*\|^2 \cdot tR^2$

$\because t$ is always positive

$\therefore t\rho^2 \leq \|W_*\|^2 R^2$ $\therefore t \leq \frac{R^2 \cdot \|W_*\|^2}{\rho}$

$\therefore t_{max} = \frac{R^2 \|W_*\|^2}{\rho}$

$\therefore$ Algorithm stops after t_{max} updates.

Problem 3(a): Implement the PLA and test it on some small synthetic datasets (size 2, 4, 10, 100)

3(a) (i): __What method you use in the learning algorithm to select the next point to update?__

```
for n = 1 : size(X,2)             %go through the training data one by on

    if sign(w'*X(:,n)) ~= Y(n)  %spot a misclassified data
      w = w + X(:,n)  * Y(n);     %update the weight w.r.t this misclassify.
      t=t+1;
    end
end
```

The above is the inner loop in my perceptron learning algorithm. Firstly, the PLA is going through all the data points in X, where X is the matrix storing all the training data points with its first row been all ones, the second to the last rows are to store the components of the data.

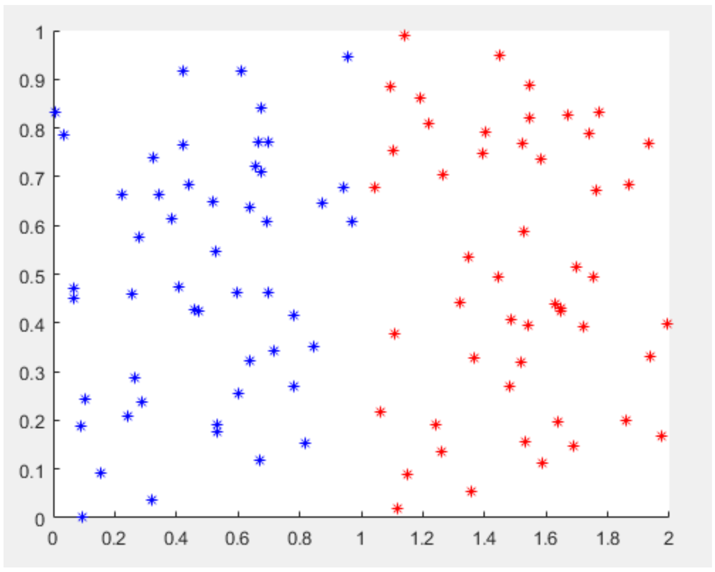
Inside the' for loop', it is the 'if' statement to detect whether there is a misclassified data. Then the weighting vector 'w' gets updated if a misclassification occurs. 't' is just an index variable to record the number of updates.

The next point to update is just the next data point is found to be classified. Notice that n does not go back to 1 when finding the next update point. It just carries on with the increment of the index variable n. I used rand() function to generate the tasked synthetic datasets in 2-dimension. Therefore in this sense, the next update point is semi-random graphically when plotting these synthetic data points out. But in the data matrix, the next update point follows the index n.

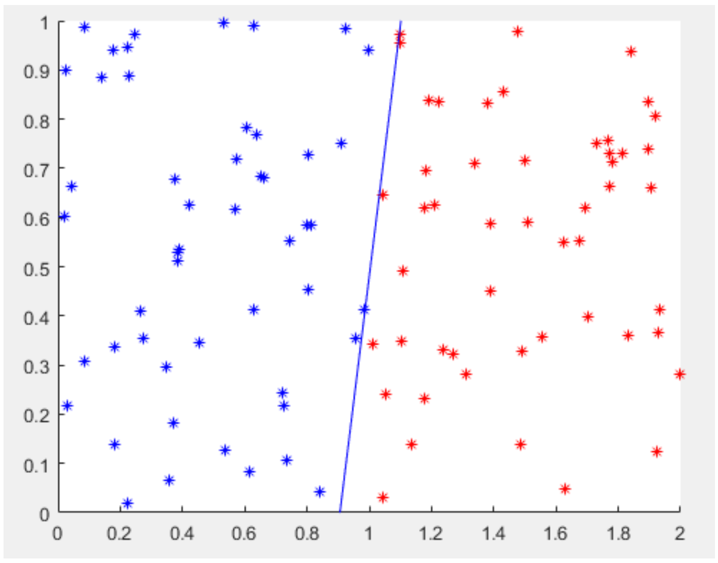
3(a) (ii): __How you generated your data and why it is useful in testing the correctness of your implementation?__

```
% training data set input class
X1=[ones(1,50);rand(1,50);rand(1,50)];    % class '-1'
X2=[ones(1,50);rand(1,50)+1;rand(1,50)]; % class '+1'
X=[X1,X2];
% output class [-1,+1];
Y=[-ones(1,50),ones(1,50)];
```

I deliberately generate some linearly separable data in 2D so I can observe them graphically. The code above is the example of generating an input data set with size 100 (50 to +1, 50 to -1). And graphically, it is represented by the point with [0, 1] in the horizontal axis been classified -1 and [1, 2] in the horizontal axis been classified +1. The example graph of generated data points on a 2D-axis is shown below.



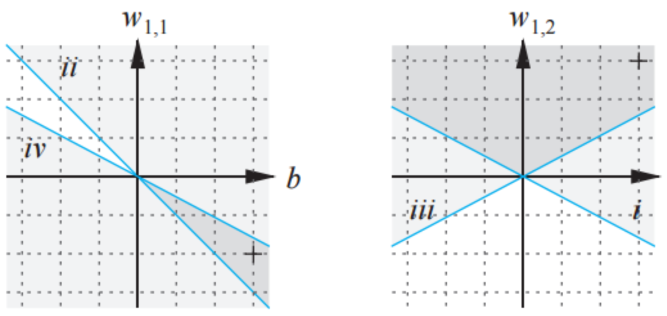
The second reason I choose to generate linear separable dataset in 2D is that I can check the correctness graphically after done the perceptron learning algorithm. Since the training dataset has dimension d = 2, the training dataset matrix X is therefore of 3 (d+1) rows and n columns, with n been the size of training data set. In this case, the final weight vector x is of dimension 3 as well. And with the help of the perceptron plot function 'plotpc()', the perceptron separation line in 2D coordinates can be easily plotted in the same graph above. And I can just check the correctness of the perceptron learning result with my eyes. One of the result is shown below (n=100).



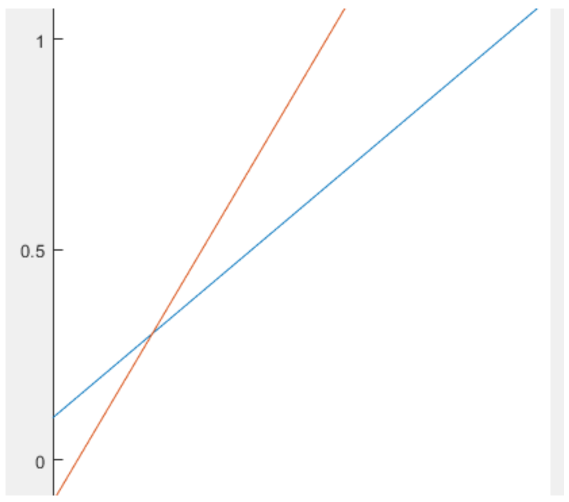
It is useful in testing the correctness of the perceptron implementation because the test error is the most important factor determining whether a perceptron implementation result is good or bad. In the real-world scenario, unlike this simple synthetic case, we never know the target function (f). Therefore the best we can do is trying to predict the target function instead of constructing it. Our best final result can only be g ≈ f. In this case, the training data is linearly separable and the target function is horizontal = 1. But the learning algorithm does not know it, so the separation line it produces is a tiny bit 'inclined'. Therefore if we put some more test data on it to test the correctness of this perceptron result, we will still get some error (misclassification occurs).

Problem 3(b): Consider the problem the 2D dataset is uniformly distributed in the positive unit square. And for any data points, the label is 1 if $x_2 - x_1 - 0.1 \geq 0$ and -1 otherwise.

3(b) (1): <u>Give a closed form formula on the test error of any separator line $ax + b$ (any other characterization of a linear separator is acceptable).</u>



In this case, the training dataset is still linearly separable. And the separation hyperplane (2D) is$X_2 = X_1 + 0.1$. For simplicity, I will just use x for $X_1$and y forX_2. So the separation hyperplane is $y = x + 0.1$. $y = ax + b$ is the trained separation line. Therefore the ideal theoretical error will just be the intersection area of the line $y = x + 0.1$ and$y = ax + b$. The graph on the right shows an example of one of the most usual case. The blue line is $y = x + 0.1$ and the red line is$y = ax + b$. Therefore the theoretical test error in this case is the two triangle area between these lines. This theoretical test error is the expectation of all the possible error can be made in a theoretically infinite size of test data set.

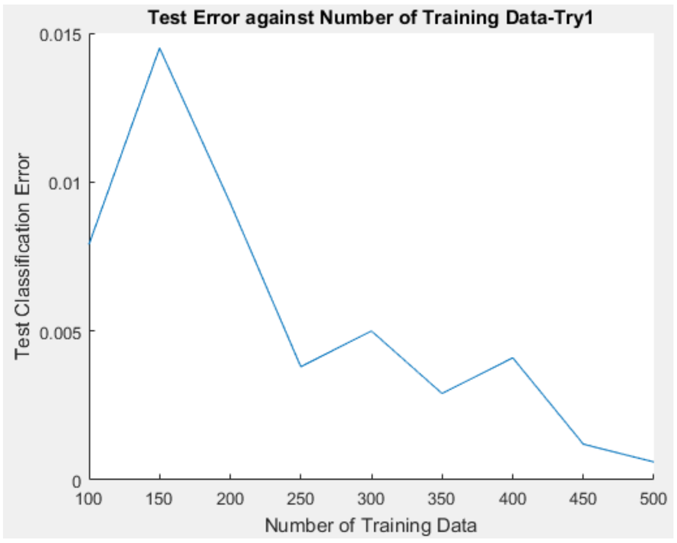


The open form solution to work out the test error is just for each specific separator line, use integrations and trigonometry to work out the area of the intersection between the target function line and the chosen separator line.

The close form solution can be obtained by taking all the considerations of different situations of the line $y = ax + b$ and work out the intersection area to the line $y = x + 0.1$. Then taking the average of these areas will give us the close form solution.

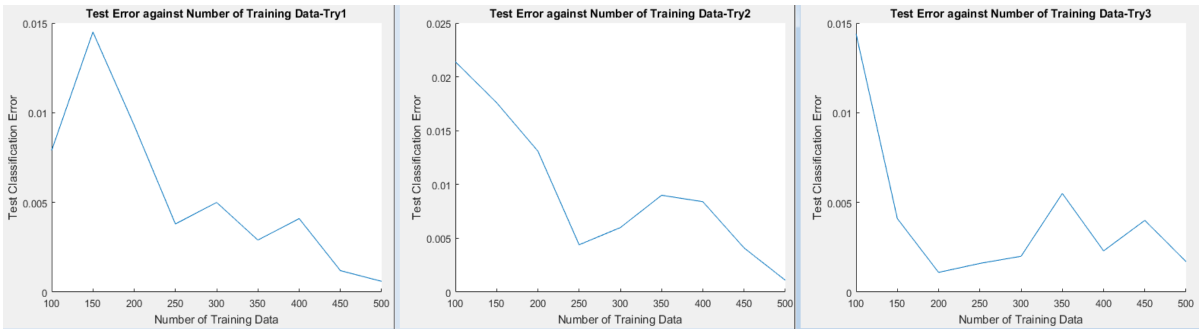
3(b) (2): ***Train your algorithm on 100,200,300,400,500 points and plot the resulting test error.***

The diagram on the right shows the test error gained from perceptron learning of different training data sizes. The x-axis is the size of training data. I used from 100 training data to 500 with the increment of 50. Intuitively, the perceptron result should perform better if it learns from a bigger training data set. However, the graph plotted does not totally agree with the intuition. Even the whole graph goes down, but there are many ripples in the middle. This is because the randomness of the training data set as well as the testing data set.

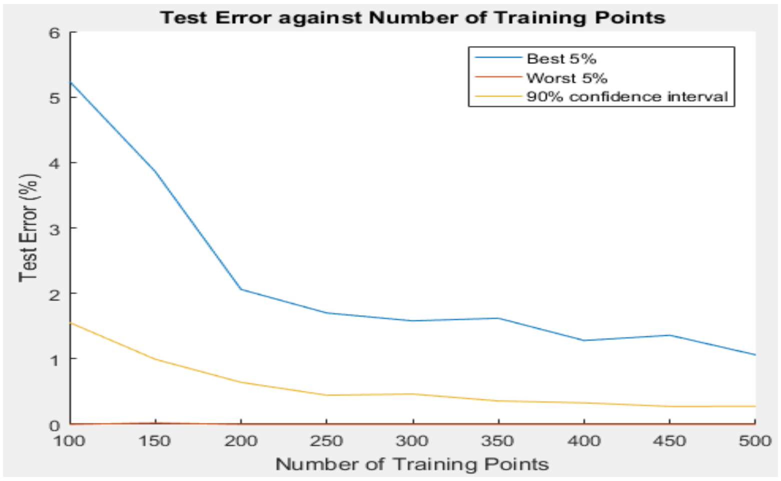


3(b) (3): ***Are the above test error values random? Repeat it 100 times, plot the average and the 90% confidence intervals (leave out the best and worst 5% of the runs).***

The above test error is shown random. The randomness is indicated by two features in the plot.

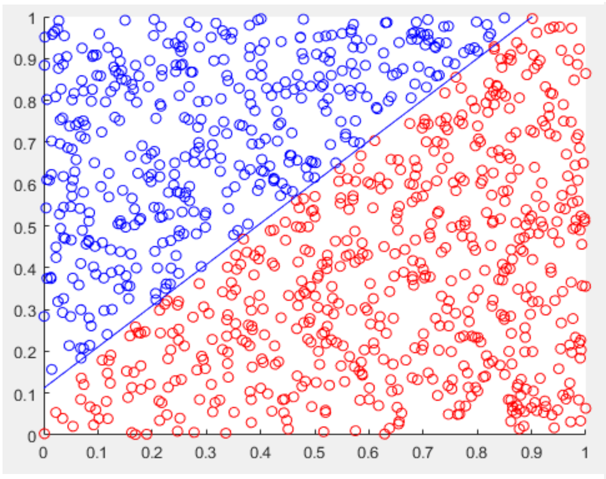


Firstly, there are ripples going on and down in the plot, which means for some bigger training dataset, the perceptron result performs even worse than the smaller training set. This is against the intuition definitively. And secondly, the ripples appear in difference positions on the plot for different tries. I have ran the code for 3 times and the results for each times are plotted above. This is because the randomness of the training data set as well as the testing data set.

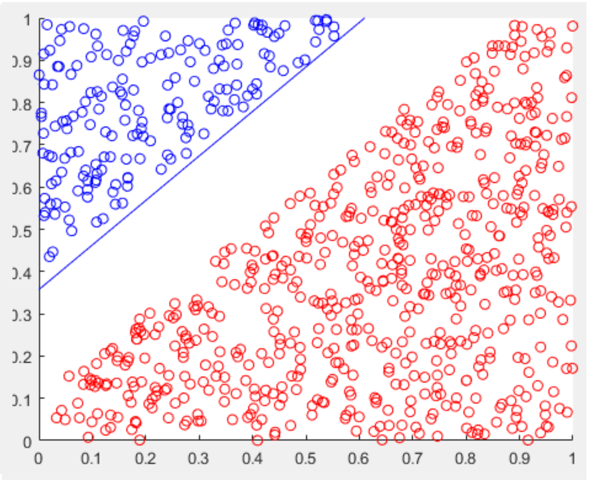


Then I ran the code for 100 times. For each size of training data, every time I give the perceptron learning algorithm a randomly generated different training set and testing set. And I record 100 hundred test errors. I put those 100 test errors in a vector and sort it. And take the biggest 5 errors, the lowest 5 errors and 90 errors in the middle respectively. Finally I take the average of the three of them and plot the diagram showing on the right hand side. In this averaged results, the intuition is reflected on the plot. Even though there are still several tiny ripples, and this is due to the randomness of the training and test data set still. I believe the ripple can be eliminated further either with taking more averages or using bigger sizes of test data.
With even bigger size of training data, the separator will become infinitively close to the target function separator $y = x + 0.1$ but never identical to.

3(b) (4)(i): ***<u>Repeat problems (b1) – (b3) when the classes are separated by some margin: using the same separator line as above $y = x + 0.1$ and the margin of 0.3, 0.1, 0.01, and 0.001 respectively. Comment on how the test error depends on the separations.</u>***



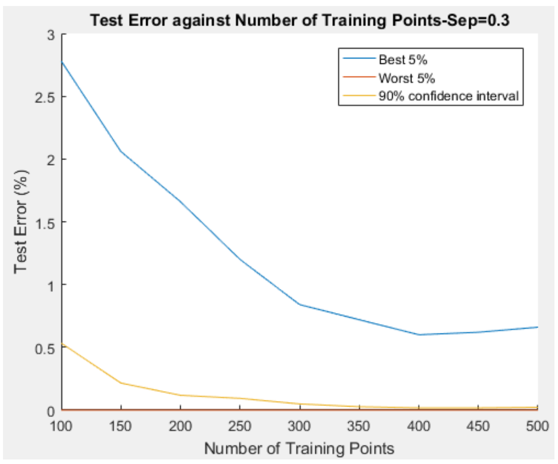
Data generated with separation = 0.3

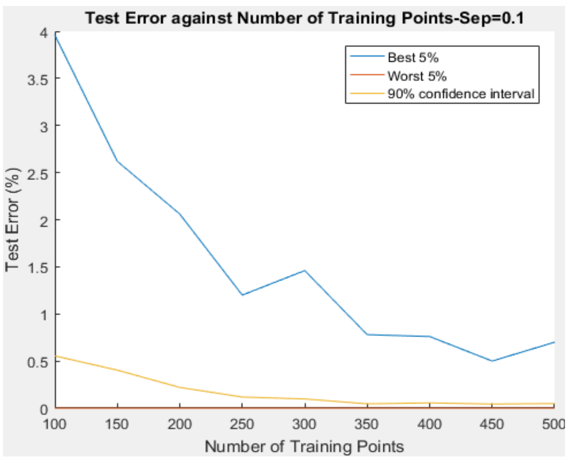


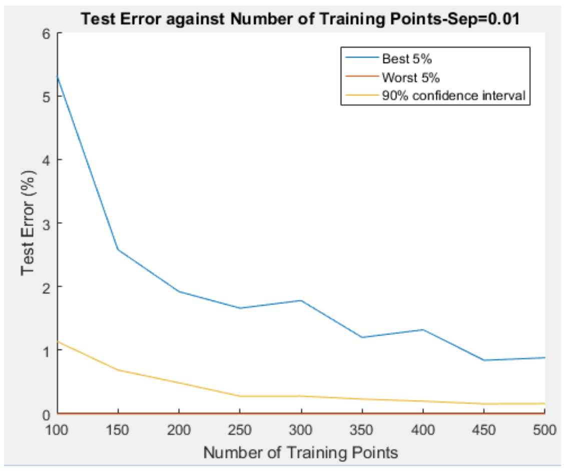
Data generated with separation = 0.01

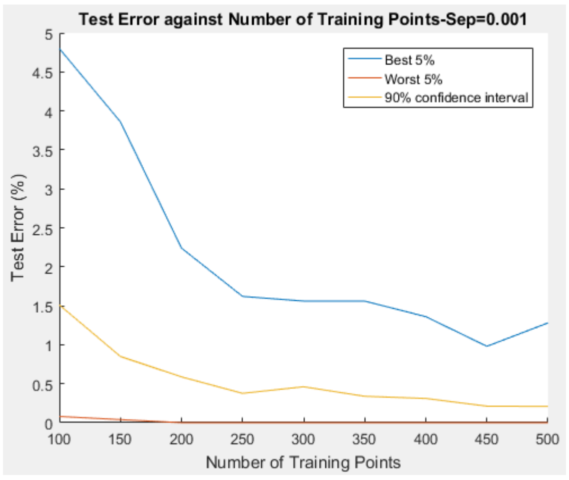
On the left, the plot is an example of points uniformly and randomly generate in the positive unit square with the separation of 0.3, and the right plot is with the separation of 0.01. Based on the graph, we have an intuition of that with the separation goes lower and lower, the points generated distributing more like the result without any separations above. And therefore we would expect the perceptron learning results as well as test error performance to be more similar to the case with no separation when the separation is getting smaller and smaller.

The average test error against the size of training data set plot is showing below, and I still used the middle 90 values to produce the 90% confidence intervals mean, as well as the top 5 and worst 5 plotted on the same graph (With the order 0.3,0.1,0.01,0.001). The 90% confidence intervals mean is represented by the yellow line, the top 5% is the blue line, and the bottom 5% is the red line.









The result totally follows my initial intuitions. The 90% confidence interval average error of the data with separation = 0.3 is the smallest (0.5%). For separations equal to 0.1 and 0.01, the average test error are around 0.6% and1.2%. And the average test error is the biggest (1.5%) with the separation of 0.001.

To be honest, the result should be more significant. This is because I only used test dataset with size 1000 to examine all the test performances in order to reduce the simulation time. But the result presenting here is well enough to conclude that there is a negative relation between the test error and the separation γ, with the separation γ getting bigger, the average test error will be smaller.

3(b) (4)(ii): ***Compare the number of updates to the theoretical bound of Problem 2. Find at least two viable choices for w_* in the definition of ρ.***

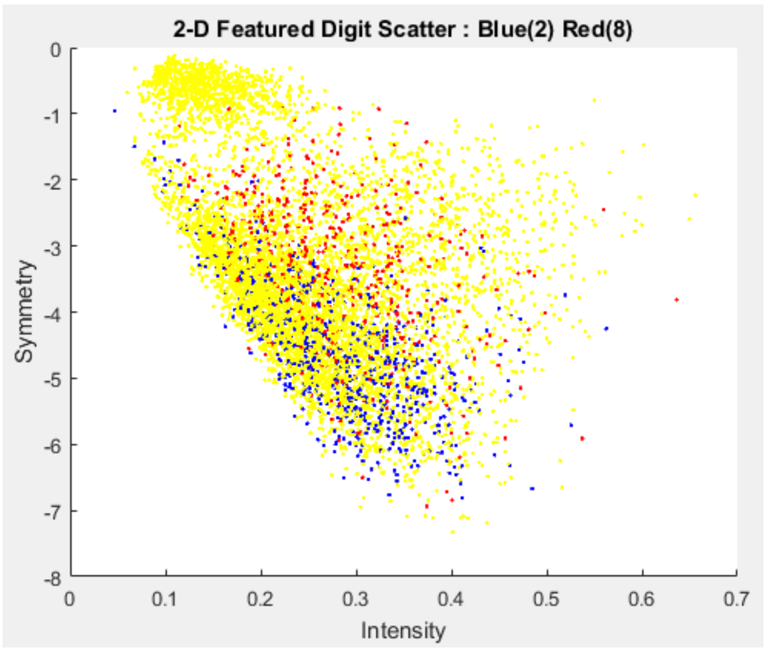
Problem 3(c): Handwritten Digit Classification

3(c) (1): Apply some modification of the perceptron algorithm to learn a linear classifier between class 2 and class 8, using first the raw dataset and then the one with 2-D features. What modification did you apply?

The modification I used is to apply 'Pocket Algorithm' in perceptron, which the weight vector only gets updated when the training error is lower. Therefore in this case, the perceptron will always the lowest training error weight vector in its pocket.

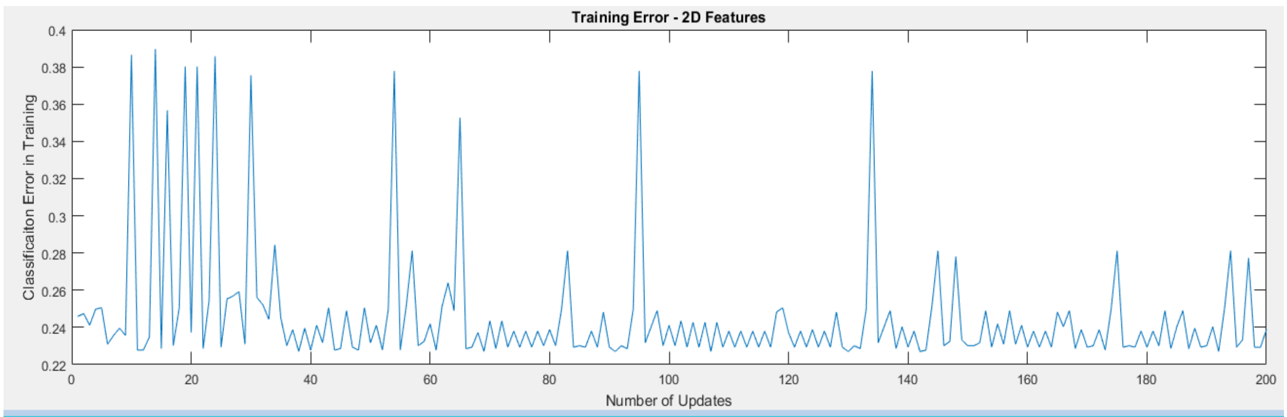
```
function[w_pocket,errorvector] = perceptron_modified2(X,Y,w_hand,update)
%Pocket Algorithm with output of the Weight in pocket and the error vector
%containing all the error against No.Update
pocketerror = Inf;                %set the initial pocket error to infinity,
errorvector = zeros (1,update); %reserve space for error vector
for n = 1:update
    handerror = 0;
    for i = 1:size(X,2)
       if sign(w_hand'*X(:,i))~=Y(i)
            w_hand = w_hand + X(:,i)*Y(i);
            handerror = handerror + 1;
        end
    end
    if  pocketerror > handerror
                pocketerror = handerror;
                w_pocket = w_hand;
    end
    errorvector (n) = pocketerror;
end
    errorvector = (errorvector/size(X,2))*100;
end
```

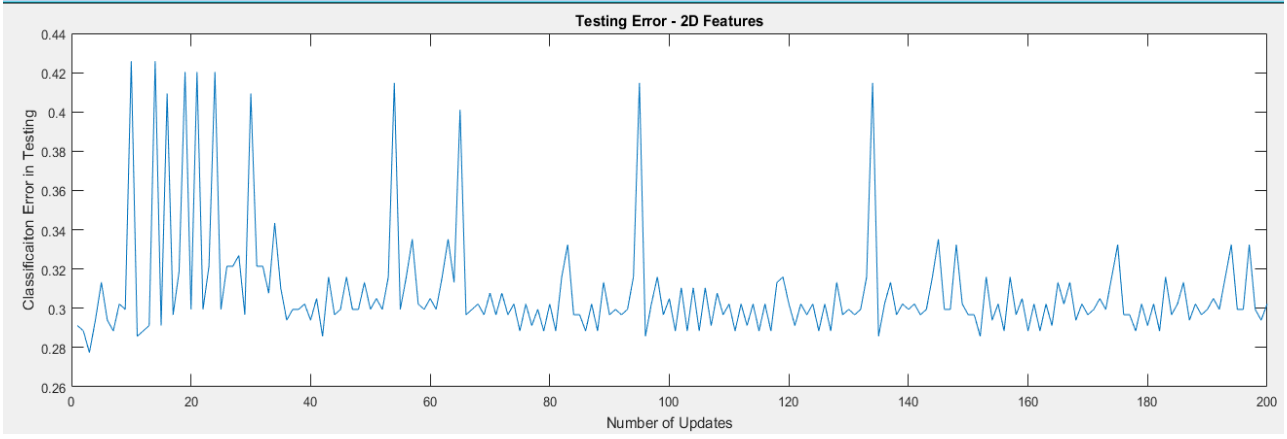
As you can see in the code above, the pocket algorithm firstly finds the misclassified point like the original perceptron learning algorithm. Then it compares the training error if the weight would be updated and the present/holding training error. The weight factor only gets updated (put into the pocket) when the updated error is lower than the present.



The 2-D featured training data can be plotted on a graph and is showing on the right here, with the red dots being class 8 and blue being class 2. And the data are NOT linearly separable. The raw training data cannot be plotted on the graph. Therefore I will use the plot of training error and test error to demonstrate the performance of PLA on these two datasets.

3(c) (2): ***<u>Compare the classification error on the training and test data of the two cases. Also, plot the change of the training and test error as a function of the number of updates of your algorithm and the original perceptron learning algorithm.</u>***

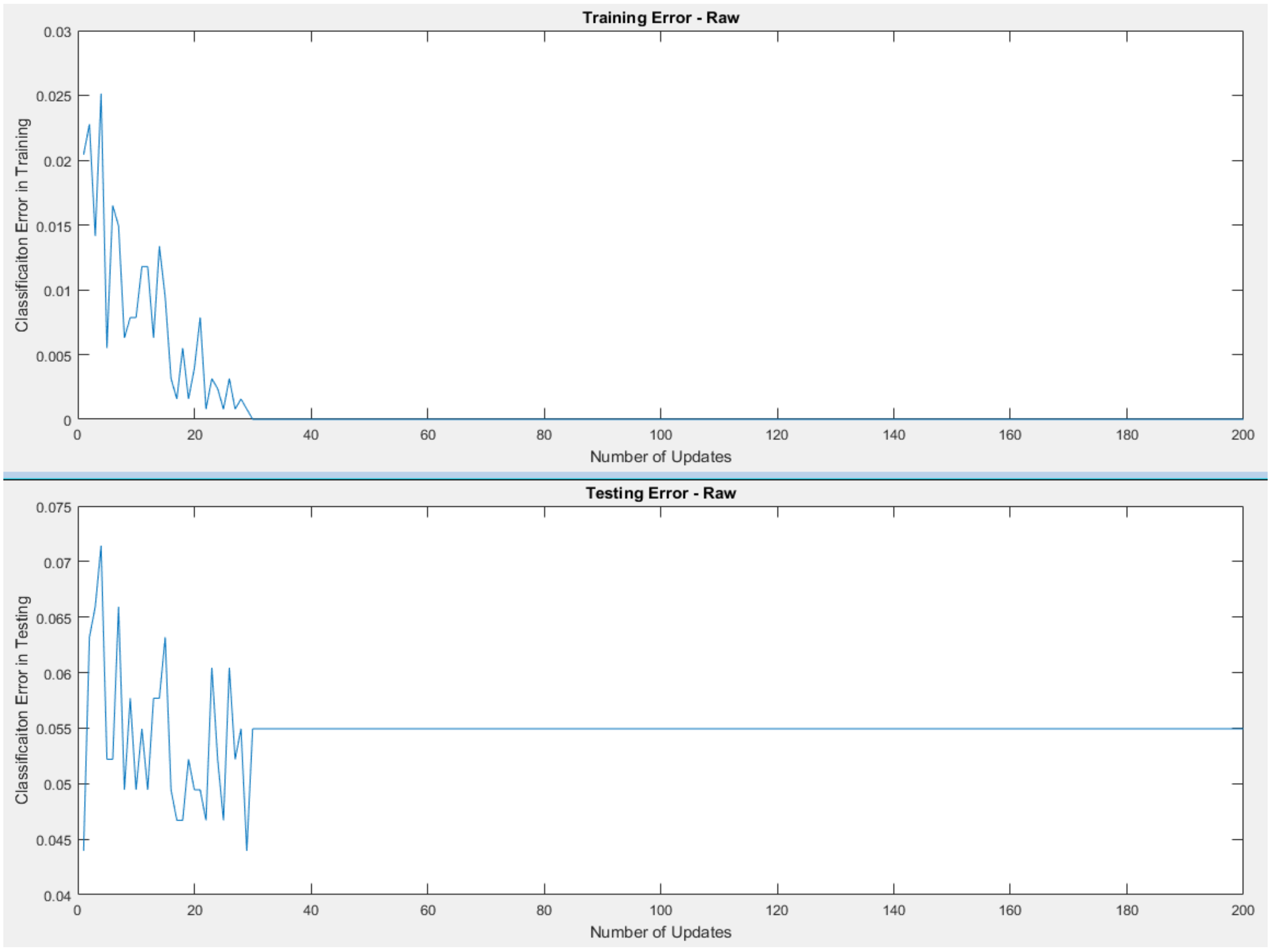




The plots above are the classification error on the training data and test data of the 2-D featured dataset. They are both fluctuating and this is because I used the original perceptron algorithm here. And the original means that as long as a misclassified data point is spotted, the algorithm will update the weight vector only based on this misclassification. In this case, the update only caring about on misclassification may disrupt many other points and this is the main reason why there are such big fluctuations in the classification error against the number of update.

Furthermore, the average misclassification rate in training data is around 0.23, and in test data is around 0.3. Therefore the misclassification rate in test data is higher.

Now let's look at the same plot on the raw dataset.

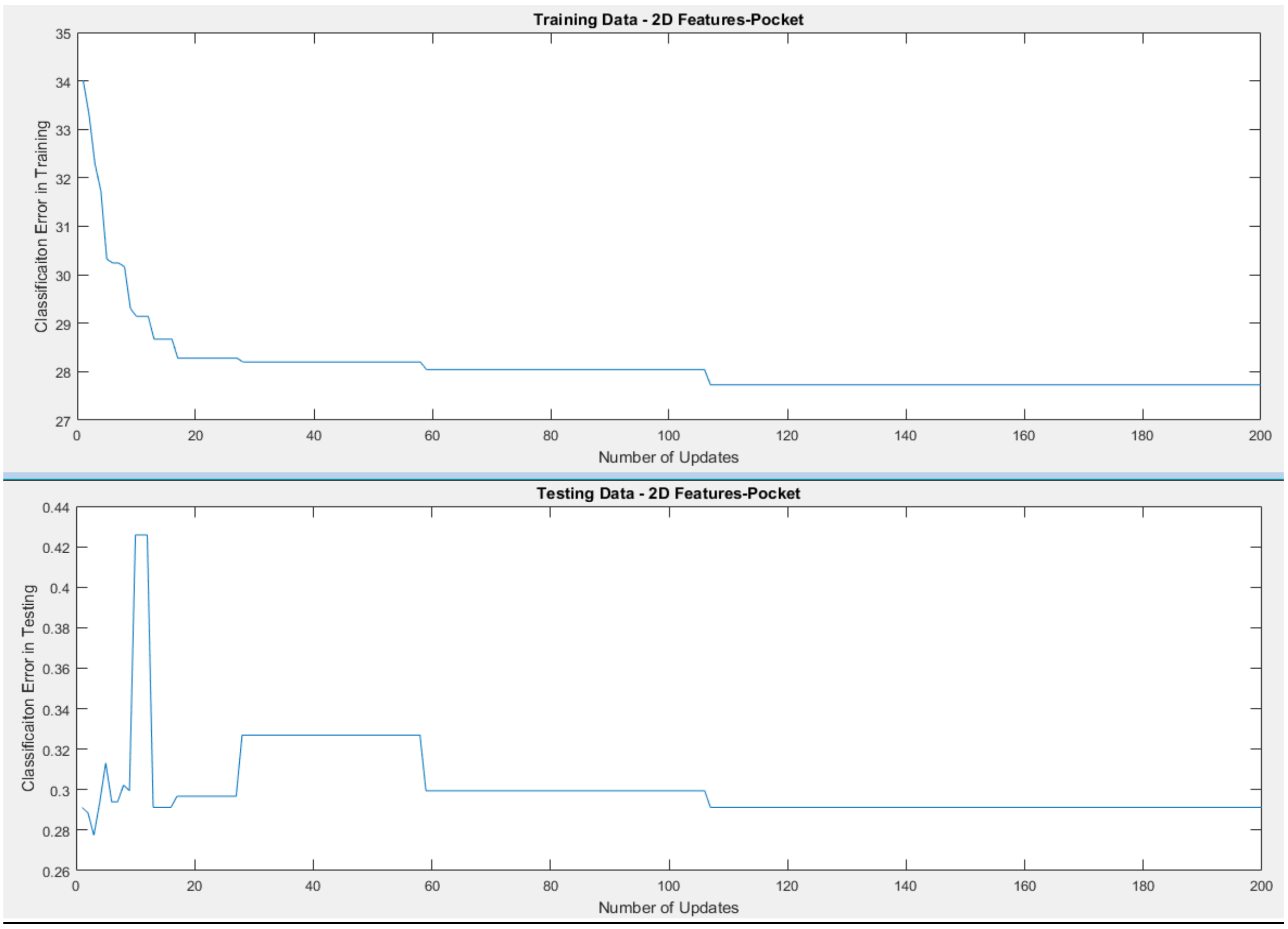


The plots of the error of raw dataset are very different from the previous 2-D dataset. Firstly, the misclassification number in the training raw dataset goes to zero. This indicates that the raw training data is linearly separable. This is reasonable since there are 256 elements in one raw data whereas only 2 elements in a 2-D featured data.

However, the misclassification in testing raw dataset is converging to 0.055 instead of zero. This is because it is never possible to find the exact same hypothesis as the unknown target function. There is also one thing to notice at the start of the testing error plot. The initial testing error with zero number of updates turns out to be the minimum. I think this is just a coincidence with the initialised weighting vector to be all zero.

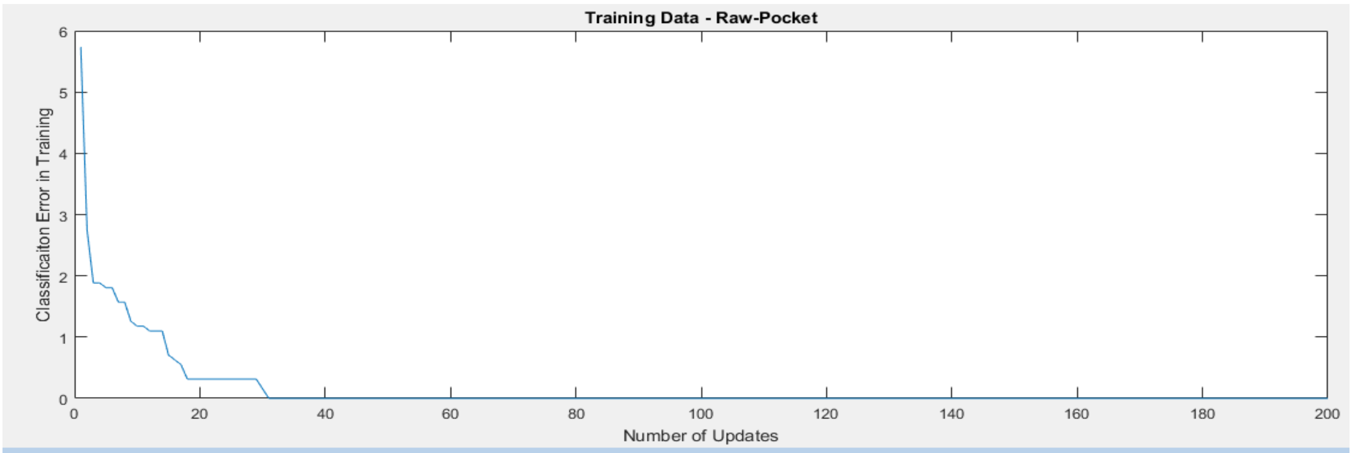
Beyond these 4 graphs, I also used the modified (pocket) perceptron algorithm on both datasets. The 4 results are showing below.

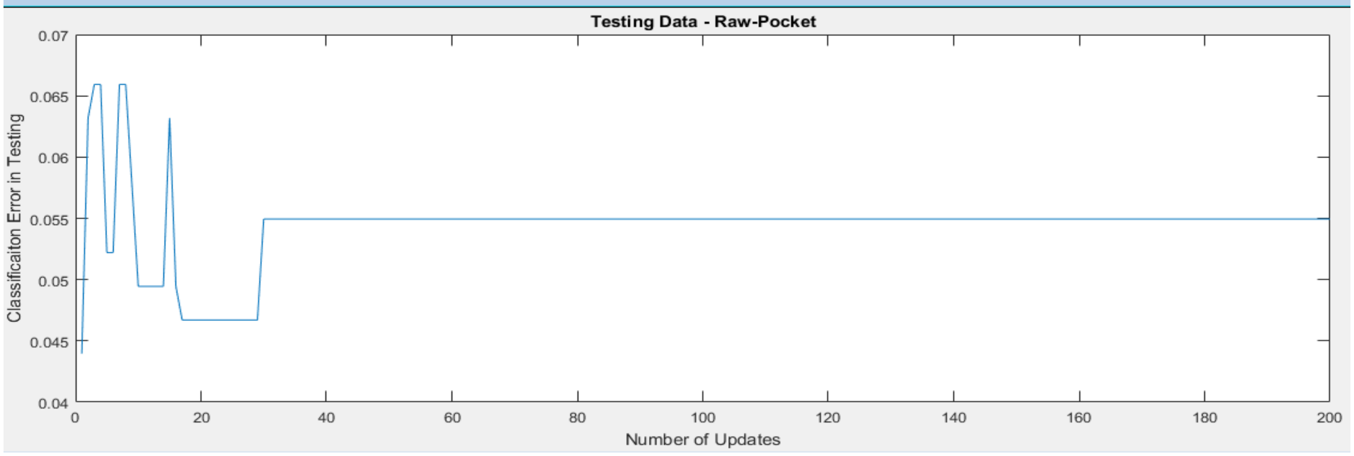
For 2-D featured dataset:



Since the main concept of pocket algorithm is to keep the minimum classification error in training. Therefore we the training error against the number of update can only be downward sloping. This is indicated on the training misclassification percentage against the number of updates with the minimum being 28%. And the testing misclassification rate is also fluctuating less than before, with its converging to around 28% as well in the end.

For raw dataset with pocket algorithm:





This is the error result of pocket algorithm learning on the raw dataset. Again, because of the linearly separable raw dataset, the training error is again zero after around 30 updates. And the testing misclassification rate is still 0.055. The biggest difference is still that there are much less fluctuations in error performance when using the modified pocket algorithm.

3(c) (3): <u>*Compute the optimal linear regression weights (minimizing the squared error) for the training set using the 2-D features, and repeat (c2) with your learning algorithm initialized with the optimal linear regression weights. Compare with the previous results.*</u>

On the right, it is the derivation of the minimising squared error in linear regression and the optimal linear regression weights. The optimal linear regression weights is produced in Matlab code by:

```
W_init = X_pi*Y1_train';
```

The optimal linear regression weight should provide a very good error performance when it is used as the initial weight. However, the original perceptron learning algorithm will still try to fix every misclassified data point it sees, therefore the error performance will still fluctuate and sometimes go worse even with the initial weight been the optimal

Linear Regression

Minimizing $\widehat{R}_n(w)$:

$$\widehat{R}_n(w) = \frac{1}{n}\|Xw - y\|^2$$

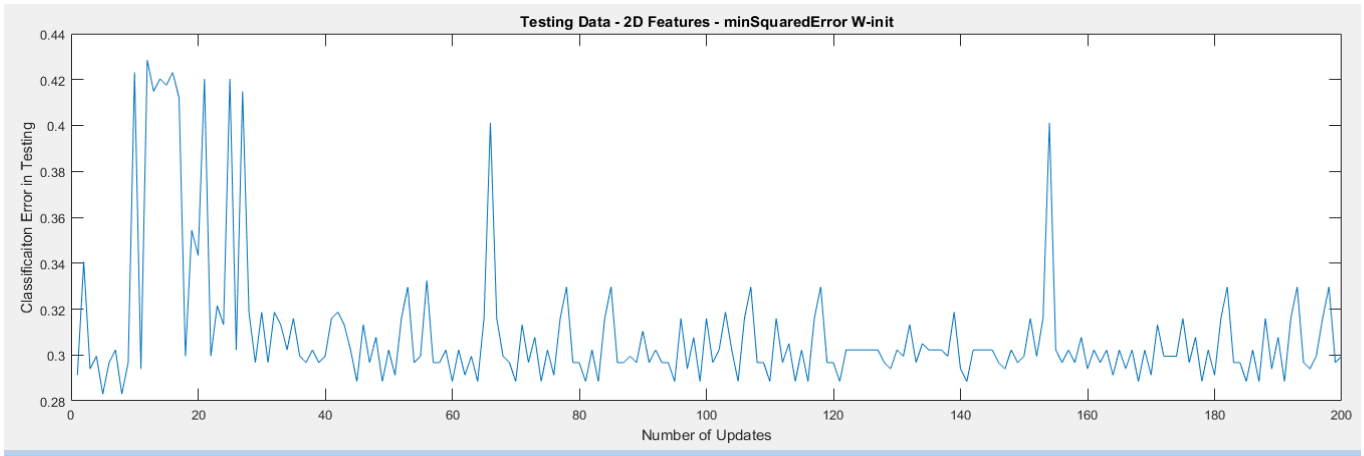
$$\nabla \widehat{R}_n(w) = \frac{2}{n}X^\top(Xw - y) = 0$$

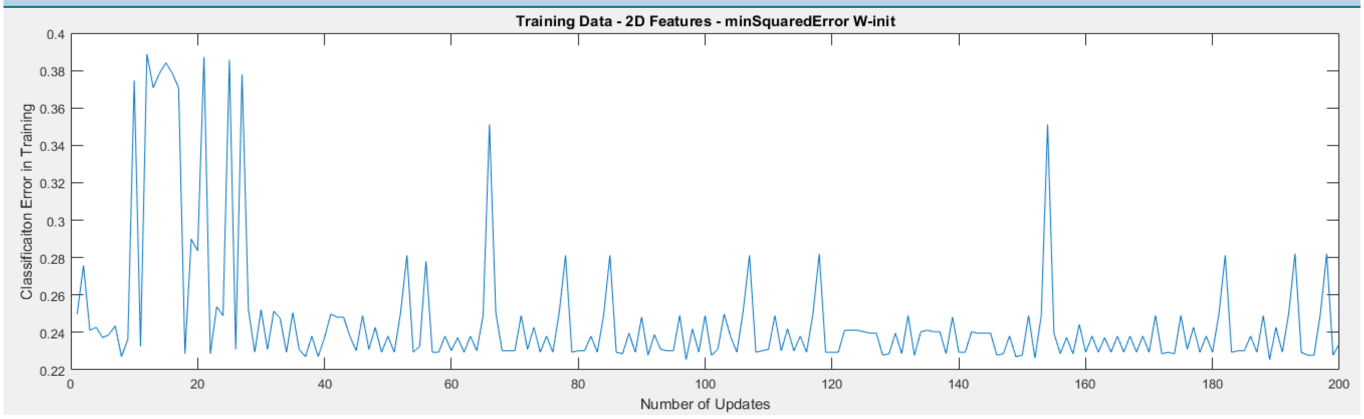
$$X^\top X w = X^\top y$$

$$w = X^+ y \text{ where } X^+ = (X^\top X)^{-1}X^\top$$

X^+ is the Moore-Penrose pseudoinverse.

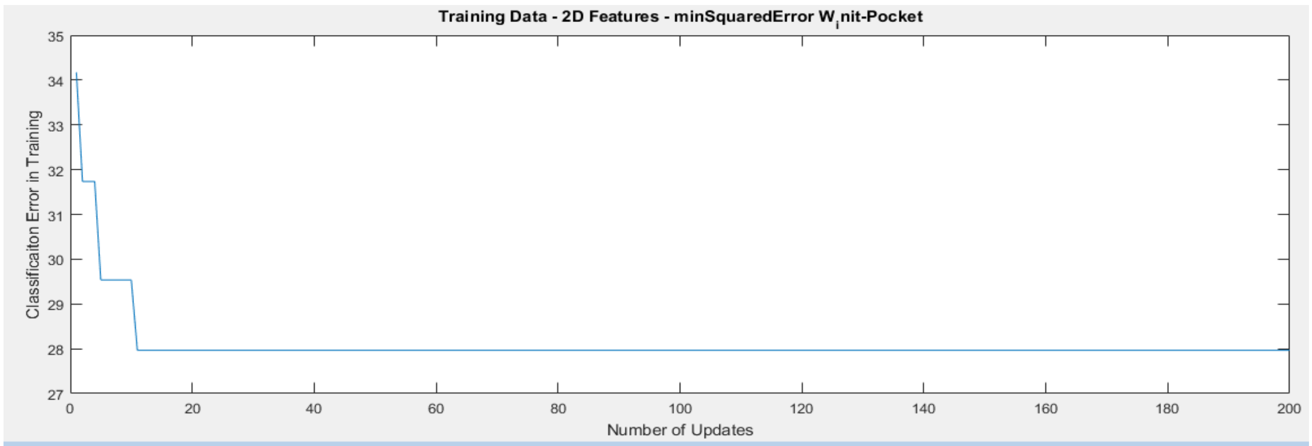
linear regression weight. And let's see the error performance plot with initial weight being optimal linear regression weights and by the original perceptron learning algorithm.

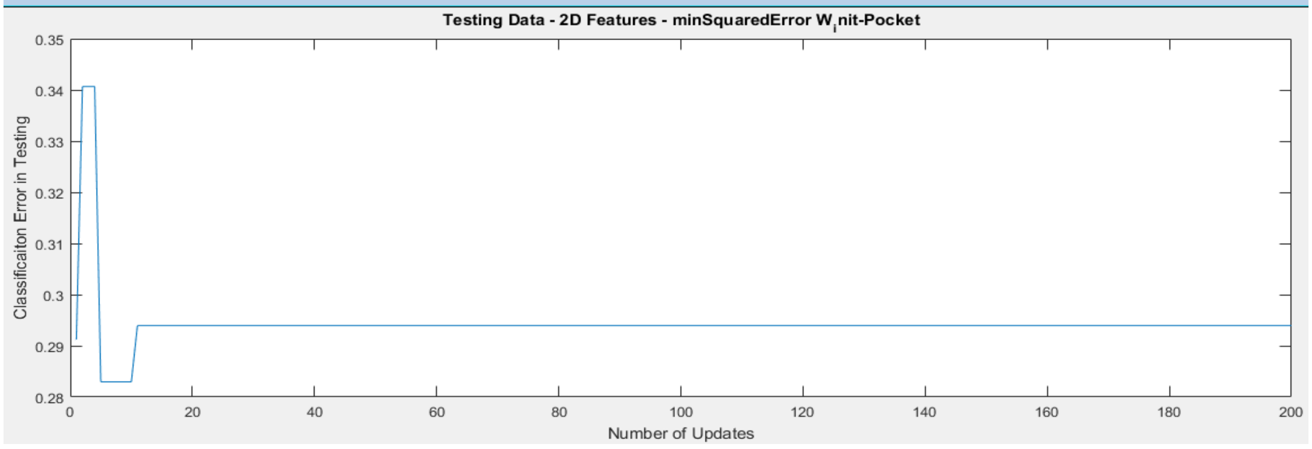




The mainly different in the error performance is at the start, the classification error in training and testing is very low at the start because weight vector is initialised with the optimal linear regression weight. This weight can produce a very good error performance (maybe not the best though). And after, because the original perceptron learning algorithm cannot keep the best error performing weight, the error performance still fluctuates as usual.

And now, I used the modified pocket algorithm with the initialised weight being optimal linear regression weight. The error performance plots is shown below.





When using the modified pocket algorithm along with the optimal linear regression weight. The improvement on test performance is very significant. It takes much less update to find the minimum classification error in training. When not using the optimal linear regression weight, it takes around 120 update for the pocket algorithm to find the minimum classification error weight. But with the optimal linear regression weight, it only takes 12 updates, this improvement on efficiency is huge.

In conclusion, using the modified pocket algorithm along with the optimal linear regression weight is the most efficient, smartest algorithm so far.

3(c) (4): ***<u>Assuming all data points are generated in an i.i.d. fashion, give a high-probability upper bound on the difference of the empirical test error and the ideal test error (defined as the expectation over the generating distribution).</u>***

Here we need to use the Hoeffding's Inequality $P\ (|Rn(h) - R(h)| \geq \varepsilon) \leq 2\exp(-2\varepsilon^2 * n)$.

In the Hoeffding's Inequality:

1. $Rn(h)$ is the empirical test error, which is the classification error we obtained from the test dataset;
2. $R(h)$ is the ideal test error, which is the close form error or the expectation of the classification error in the i.i.d fashioned test data;
3. n is the size of training dataset,n=7291.

In this problem, we would like the high-probability to be 90% because I think 90% is a reasonable confidence level to be thought as high-probability. The Hoeffding's Inequality is to work out the lower bound of the difference between the empirical test error and the ideal test error, therefore if we want the upper bound, we would like to do a small modification:

$$P(|Rn(h) - R(h)| \leq \varepsilon) \geq 1 - 2\exp(-2\varepsilon^2 * n)$$

The left hand side is the probable upper bound and the right hand side is the probability. We want the probability to be high. So 90% is reasonable, and therefore $2\exp(-2\varepsilon^2 * n)$ will be equal to $(1 - 90\%)$, which is 10%. Therefore we have:

$$2\exp(-2\varepsilon^2 * n) = 10\%$$

$$\exp(-2\varepsilon^2 * 7291) = 5\%$$

$$\varepsilon \quad = 0.01433 = 1.433\%$$

In conclusion, the 90% high-probability upper bond bound on the difference of the empirical test error and the ideal test error is 1.433%. This means there is an at most 10% chance that the empirical test error and the ideal test error would differ by more than 1.433%.