Assignment 1

Gandhar Vaidya

September 15th , 2016

Contents

Abstract

This document briefly explains four variants of the Perceptron Algorithm . The document also presents their implementations in python. The implementations are demonstrated using two simple data sets.The test data sets are obtained from the UCI Machine Learning Repository.After implementation of Perceptron Algorithm on these 2 simple data sets, testing samples are taken to measure accuracy in terms out sample Error(Eout).Also, the variants of the algorithm are run on a scaled version of one of the data sets. At the end the results of these experiments are discussed with the help of different plots and learning curves.

1 Introduction

The Perceptron algorithm was invented in 1956 by Frank Rosenblatt. The algorithm works on the principle of linear classification of elements in two classes. The Perceptron algorithm works on a feed-back principle. In the initial section of the document creation of simple data along with extraction of input is explained. Code for extracting data and separating the labels and samples is present in the following section. Perceptron algorithm and its variants are explained in the sections ahead.All the variants are supplemented with their respective python codes used for weight updates. Codes are tested with the data sets and the accuracy of different variants is demonstrated with help of different plots. Learning Curve section presents the accuracy a particular Perceptron variant on the Heart Data set on a logarithmic scale. The next section 'Normalization of Data' includes implementation of classifiers on a scaled data set. The same section presents the results ,in terms of accuracy, of running a classifier on the scaled data set.

2 Perceptron Algorithm

This section shows implementation of the code along with its demonstration on simple data.

2.1 Perceptron model without using Bias(b)

This particular model of Perceptron does not make use of a bias in the process of weight updation. The bias term adds more margin to the linearly separated data,which we will see as the next variant of the algorithm.

2.1.1 Implementation

Python Code of the algorithm :

```
# Creating Simple Data
import numpy as np
from matplotlib import pyplot as plt

class Perceptron :

    def __init__(self, max_iterations=100, learning_rate=0.2) :

        self.max_iterations = max_iterations
        self.learning_rate = learning_rate

    def fit(self, X, y) :
        self.w = np.zeros(len(X[0]))
        converged = False
        iterations = 0
        while (not converged and iterations < self.max_iterations) :
            converged = True
            for i in range(len(X)) :
                if y[i] * self.discriminant(X[i]) <= 0 :
                    self.w = self.w + y[i] * self.learning_rate * X[i]
                    converged = False
                    plot_data(X, y, self.w)
            iterations += 1
        self.converged = converged
        if converged :
            print 'converged in %d iterations ' % iterations

    def discriminant(self, x) :
        return np.dot(self.w, x)

    def predict(self, X) :

        scores = np.dot(self.w, X)
        return np.sign(scores)

def generate_separable_data() :
```

```
    xA,yA,xB,yB = [np.random.uniform(-1, 1) for i in range(4)]
    w = np.random.uniform(-1, 1, 2)
    print w,w.shape
    X = np.random.uniform(-1, 1, [N, 2])
    print X,X.shape
    y = np.sign(np.dot(X, w))
    return X,y,w

if __name__=='__main__' :
    X,y,w = generate_separable_data()
    p = Perceptron()
    p.fit(X,y)
```

This is the basic code extracted from the assignment page. It has 4 major functions
1.Fit function is basically used for the weight update phase of the algorithm.
2.The discriminant function returns the dot product of the weight vector with the samples.
3.The predict function returns the sign of the dot product thereby facilitating the comparison with labels i.e. y which is either +1 or -1
4.The generate_separable_data function is used for data extraction and segregation into testing and training samples.

2.1.2 What is the Perceptron algorithm?

The Perceptron algorithm works on the principle of feedback.The algorithm iterates over the training samples and updates the weight vector w by which the sample xi is more likely to be correctly classified. Whenever a certain element gets misclassified the weights get updated to correct the error. The algorithm does not guarantee convergence and hence should be implemented for limited iterations. The following equation is the basic weight update equation for a Perceptron algorithm.

$$w' = w + \eta yixi \tag{1}$$

η is called as the learning rate and lies between 0 and 1.

2.2 Perceptron with bias

A Perceptron algorithm with bias produces better results. Bias term is valued as it allows us to shift the activation function to the left or right. It gives a better margin of separation between two classes. In simple words, it acts like the co-efficient b of a function

$$y = ax + b \tag{2}$$

It allows you to fit the prediction with the data better. Without bias , your line will always go through the origin (0,0) which may result in poor classification.

2.2.1 Implementation with bias

```
# Weight updates using bias
while (not converged and iterations < self.max_iterations) :
            converged = True
    for i in range(len(X)) :
        if y[i] * self.discriminant(X[i])+self.b <= 0 :
            self.w = self.w + y[i] * self.learning_rate * X[i] + self.b
            self.b = self.b*self.learning_rate + y[i]
```

The above code is just a snippet from the fit function describing the Perceptron with bias algorithm. The bias term gets updated after every misclassification to elevate the performance of the algorithm. This bias term is then used in the predict function as mentioned below

```
#adding bias to the scores
 def predict(self, X) :

        scores = np.dot(self.w, X)
        return np.sign(scores) + self.b
```

2.3 Pocket Algorithm with bias

In simple words , pocket algorithm can be explained as - I currently have the best value of weight vector in my pocket. By 'best' I mean the value of the weight vector that classifies the sample xi in the best possible way up until now . If xi gets misclassified and the updated weight vector is better than the weight in my pocket we swap the weight vectors. This judgement occurs on the basis of **In-Sample Error (Ein)**.

2.3.1 Implementation

```
#We start by initializing wpocket = 0 and Ein2 =1
Ein2= 1
w.pocket=0
```

I have used a similar weight update policy for the pocket algorithm as I have used for the bias Perceptron above. I have only made the following additions to check for the best weight in terms of Ein.

```
for i in range(len(X)) :
    if y[i] * self.discriminant(X[i]) <= 0 :
        self.w = self.w + y[i] * self.learning_rate * X[i] + self.b
        self.b = self.b + y[i]*self.learning_rate
        converged = False
        count = count + 1
    #determining wpocket on basis of lesser Ein
    Ein = count/len(X)
    if (Ein<Ein2):
        w_pocket = self.w
    Ein2 = Ein
```

The idea behind initializing Ein2 = 1 was to compare Ein for the 1st iteration with a higher value so that the Ein for the current weight vector is selected and the current weight vector gets selected as the wpocket.This is the basic principle of pocket algorithm wherein this algorithm takes 1 iteration more than the perceptron or any other algorithm to decide on wpocket. This results in the pocket algorithm running relatively slower than perceptron but with better results.

2.4 Modified Perceptron algorithm

The modified Perceptron algorithm makes use of λ to find out the maximum of dot product between weight vector and the training sample xi. The dot product of any vector **a.b** is defined as **a.b**$cos\theta$.
This product will be maximum when $cos\theta$ is 1. i.e θ is 0.
This means the weight vector looks for samples xi which are close to it and tries to classify them instead of looking for the ones which are far off. If a nearby sample is correctly classified it updates the weight and then all the elements in the vicinity of the new weight vector will get classified.

2.4.1 Implementation

```
#initializing a random but uniform w and c to be between 0 and 1.
self.w = np.random.uniform(-1,1,len(X[0]))
c = 0.03
```

```
#Weight update
 while (not converged and iterations < self.max_iterations) :
    converged = True
    for i in range(len(X)) :
        lamda = (y[i]*self.discriminant(X[i]))

        if (lamda < (c * np.linalg.norm(self.w))):
            l[i] = lamda
```

```
            converged = False
            count = count + 1
    Ein = count/len(X)


    j = np.argmax(l)
    self.w = self.w + self.learning_rate*y[j]*X[j]
```

Here as per the requirement of the algorithm, I am calculating lambda as λi=yi*w(t)T*xi, which evaluates λi for every training example. For comparison of lambda with mod of w , I made use of the **np.linalg.norm()** function as shown above. The condition, if true, appends the current value of lambda to the list l[i]. Outside the for when 1 iteration is done, the **np.argmax()** function is used to get the value of j which gives us maximum λ. After this step,the weights are updated using the formula

$$w(t+1) = w(t) + \eta yjxj \tag{3}$$

2.5 Results and Discussions

The above algorithms were implemented on the following Data sets.
1. Heart Data - 270 samples
2. Gisette handwritten digit recognition dataset- 6000 samples
The results on both the data varies, in terms of accuracy depending on a number of factors like number of iterations.

2.5.1 Implementing the test function

```
#test function is used to calculate the out of sample error using test data

def test(self,X,y):
        count=0.0
        Eout=0.0
        for j in range(len(X)):
            if self.predict(X[j]) != y[j]:
                count = count + 1
            Eout = count/len(X)
        print 'Eout',Eout
```

Here, X is an array of examples and y is an array of corresponding labels. In the test function, I am making a call to the predict function to check if it is a correct classification. In case of a misclassification I am incrementing the 'count'. After exhausting the number of samples, I am taking the Eout i.e. the out sample error which is the total number of misclassifications after training divided by the number of samples.

2.6 Results

Below I am going to discuss and reproduce results which I derived after implementing the codes for the above mentioned variants of perceptron over the data sets.

2.6.1 In-sample analysis erorr

In the first part of Results , I am going to speak about the In sample Error i.e. Ein which is acquired by implementing the above 4 variants of Perceptron on the training samples for the heart data.
The Ein for Gisette data set return 0 value since it converges after certain iterations.

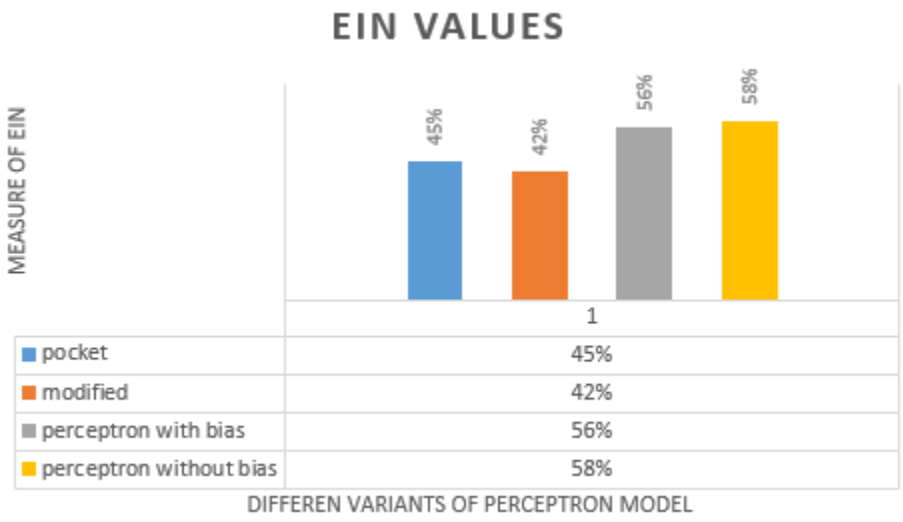

Figure 1: Comparison of In Sample Error (Ein)

As you can see from the figure above, Ein is maximum for perceptron without bias and minimum for modified perceptron. One can easily say by looking at the bar graph above that modified perceptron trains best and tries to classify the data set better than other variants.

2.6.2 Time to run

Pocket algorithm takes 55 seconds to run even for 1000 iterations. Perceptron took less than 20 seconds even for 5000 iterations Modified Perceptron takes more iterations to converge and also takes the most amount of time (easily over a minute) to reproduce results for 1000 iterations

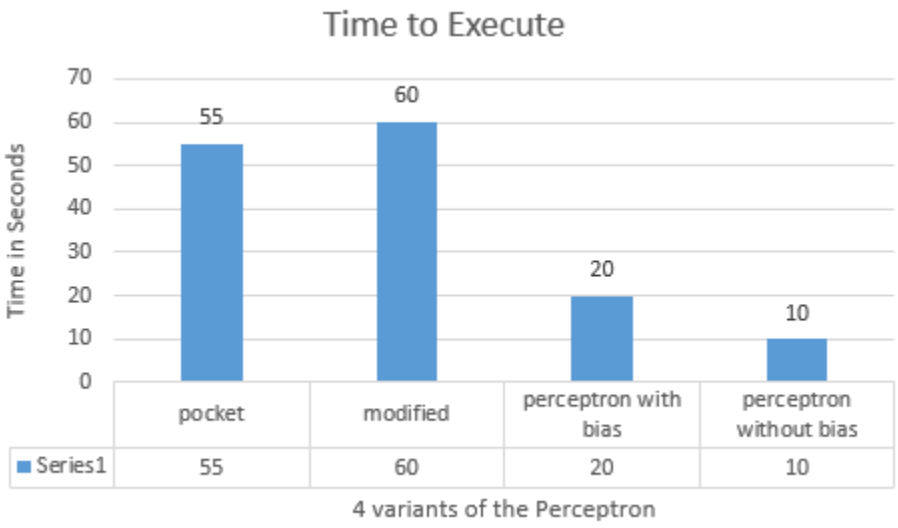

Figure 2: Comparison of Time

2.6.3 Out Sample Analysis (Eout) or Accuracy

Eout or Out of Sample Error is the error calculated using the testing data which the algorithm runs on after being trained on the trained data. This step is basically to gauge how well your data has been trained

after running it on the training data. The test function mentioned above was used in performing the below analysis and obtain the relevant graphs. The Accuracy varies over the number of iterations and hence I have used the number of iterations as a parameter to judge the accuracy of the algorithms.

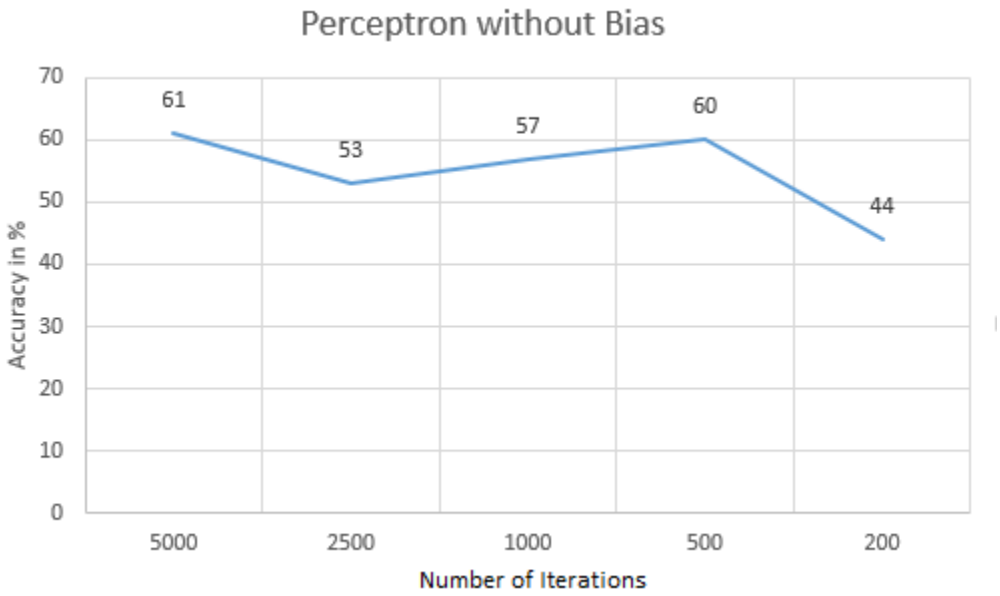

Figure 3: Comparison of Accuracy

As we can see from the above figure the Accuracy goes on increasing from 200 to 500 iterations and then there is a gradual descent till 2500 iterations and then the Accuracy hits peak at 5000 iterations. This model of Perceptron as expected will give a very low accuracy.(61 % in my case)

In the figure below, the results clearly show that the perceptron model with bias term , has more accuracy(63%) at 1000 iterations than its previous variant. The bias term adds more offset to the classifier and hence results in better performance.

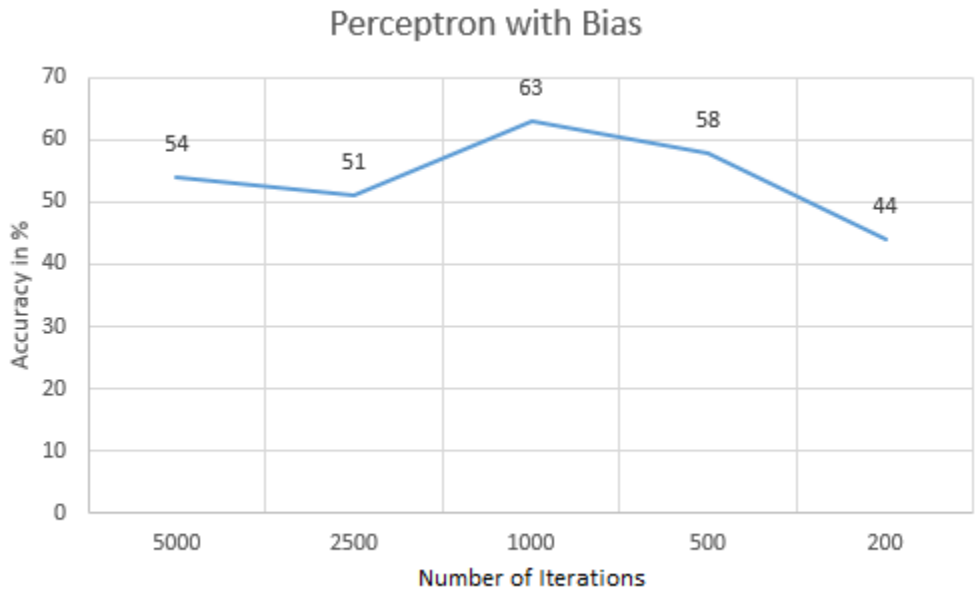

Figure 4: Comparison of Accuracy

In the figure below , we can clearly see a linear curve rising from 200 up until 5000 iterations. Also Pocket algorithm gives a better accuracy of 82.7% at 5000 iterations, which is superior to the previous models of the Perceptron.

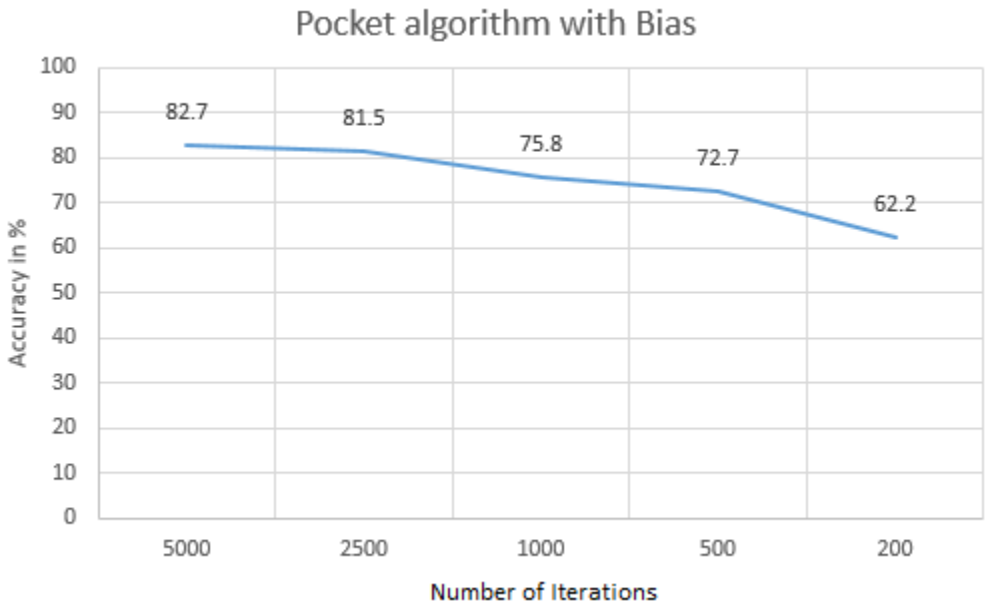

Figure 5: Comparison of Accuracy

For the Modified Perceptron model, the highest accuracy is acheieved at 500 iterations. This model surpasses the first two models of Perceptron but fails to compete with pocket algorithm.

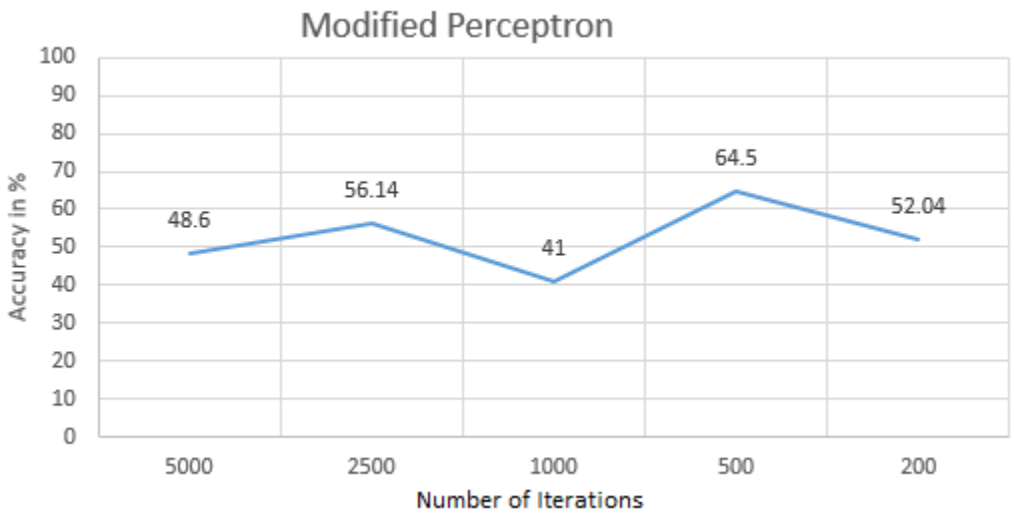

Figure 6: Comparison of Accuracy

2.7 Discussions for Gisette Data

When the above algorithms were run on the gisette data , the data converged for all ,with more efficiency than for the heart data set. The pocket and modified perceptron took longer times to converge as they are expected. The modified perceptron algorithm converged when I ran it with shuffled data at 5000 iterations and taking a higher value of c = 0.7.

2.7.1 Eout

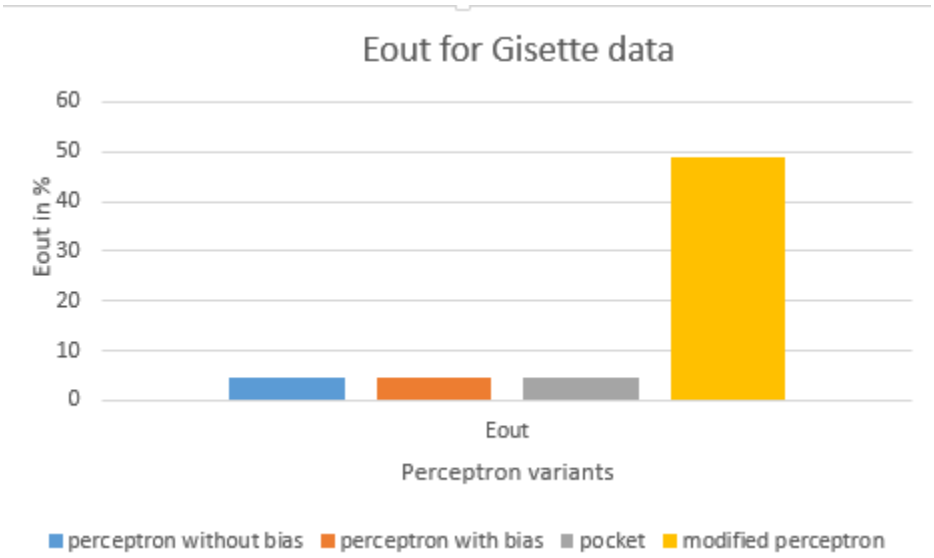

Figure 7: Comparison of Eout

2.7.2 Accuracy

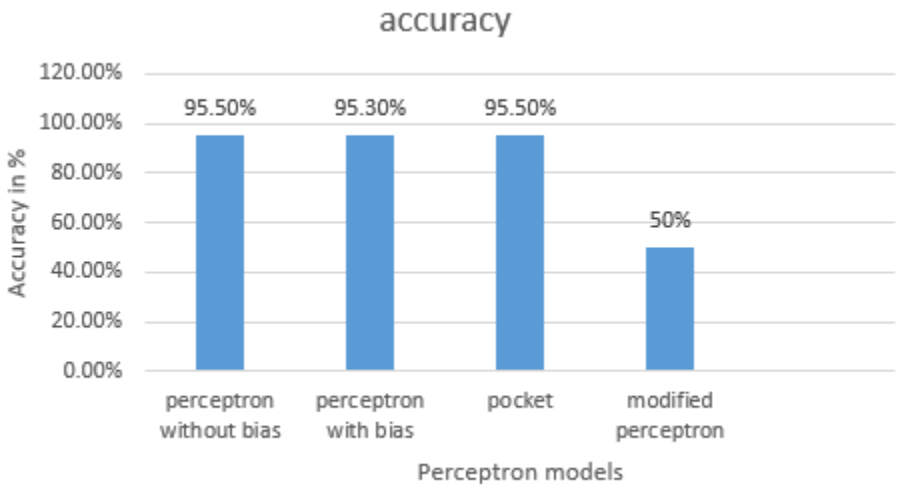

Figure 8: Comparison of Accuracy

2.7.3 Iterations

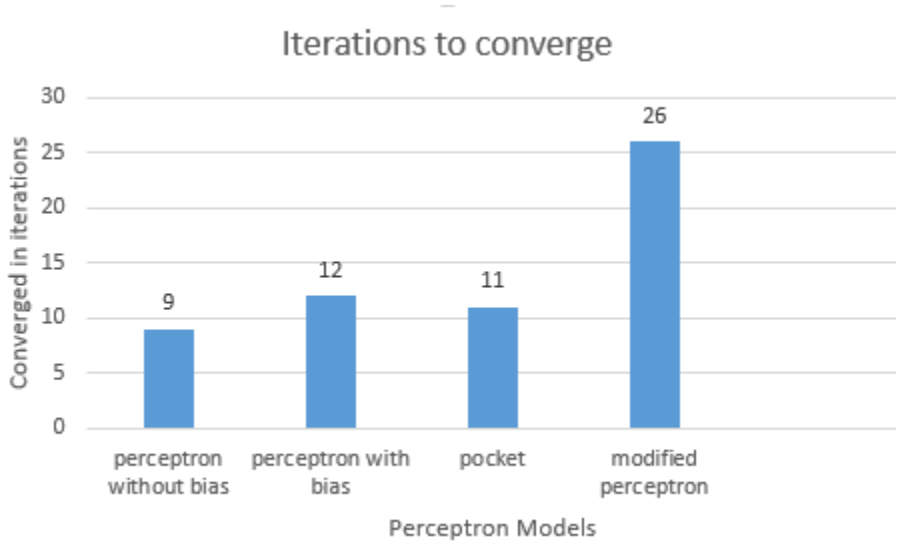

Figure 9: Iterations took to converge

From all the above analysis and demonstrations,it is pretty clear that for the data I got by shuffling, the pocket algorithm got trained the best and could produce results better and quicker than any of the other variants.The pocket algorithm used bias and hence the separation between the two classes should also be really definite.

3 Learning Curve

Learning Curve is a great way to learn the results of your implementations.I have constructed the learning curve by taking the x-axis scale as logbase=4. I obtained an almost linearly growing graph which can be

seen from the image below.It usually refers to a plot of the prediction accuracy/error vs. the training set size i.e.how better does the model get at predicting the target as we the increase number of instances used to train it.

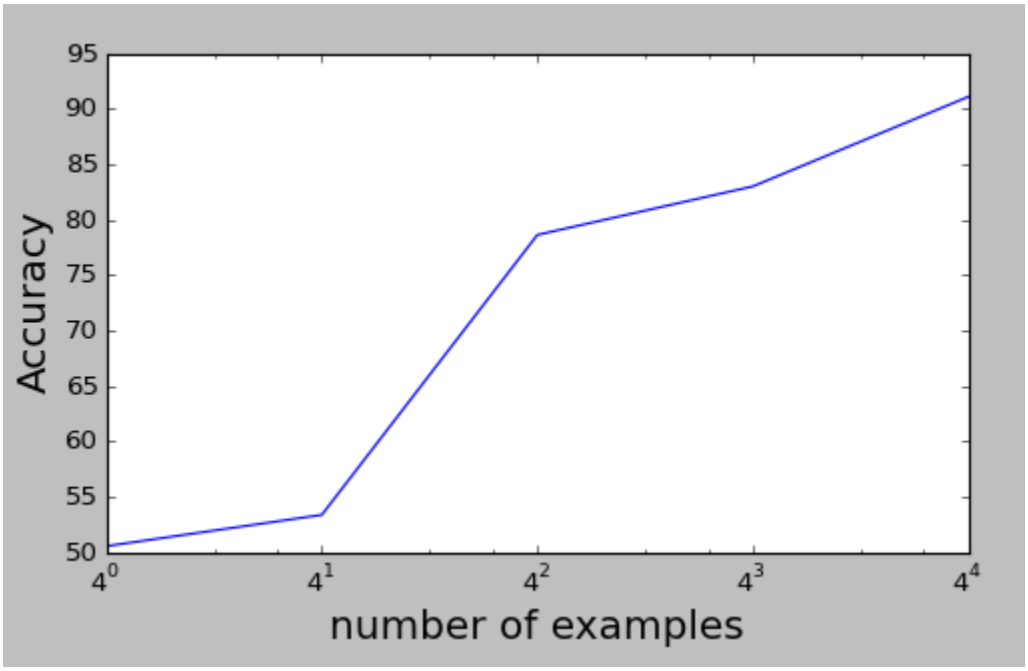

Figure 10: Learning curve

4 Data Scaling and Normalization

The data sets at hand are arbitrarily distributed and are very random in nature.This makes it harder for the algorithm to classify the examples.An efficient way to confine the data in a desired range so that it gets classified better is Scaling. Ensuring standardised feature values, implicitly weights all features equally in their representation.
The following formula is used to scale the data efficiently.

$$x' = (x - min(x))/(max(x) - min(x)) \tag{4}$$

4.1 Implementation of Scaling formula

```
#function for scaling
new_array = np.zeros(X.shape)
    a = -1
    b = 1
    minimum = np.amin(X)
    maximum = np.amax(X)
    for i in range(len(X)):
        for j in range(len(X[0])):
            new_array[i] = (((b-a)*(X[i] - minimum))/maximum-minimum) + a

    min2= np.amin(new_array)
    max2 = np.amax(new_array)
```

As you can see , I have taken 2 variables a and b to denote the current minimum(a) and current maximum(b). The loop iterates over the entire array of samples and then scales it down to fit the new_array to

be in the required range of [-1,1]

```
new_array is [[-0.75177305 -0.9964539  -0.9858156  ..., -0.99
  -0.9893617 ]
 [-0.76241135 -1.         -0.9893617  ..., -0.9929078  -1.
 [-0.79787234 -0.9964539  -0.9929078  ..., -0.9964539  -1.
 ...,
 [-0.80141844 -1.         -0.9929078  ..., -0.9929078  -1.
 [-0.79787234 -0.9964539  -0.9858156  ..., -0.9929078  -1.
 [-0.76241135 -0.9964539  -0.9858156  ..., -0.9929078  -0.989
  -0.9893617 ]]
min2= -1.0 max2= 1.0
```

Figure 11: Scaled Output

4.1.1 Normalized vs Non-normalized data

For the normalized data , Ein = 0.29 and Eout is 0.346
For the non-normalized data , Ein = 0.34 and Eout is 0.39
which in turns results in improvement in Accuracy in terms of Eout by 2%

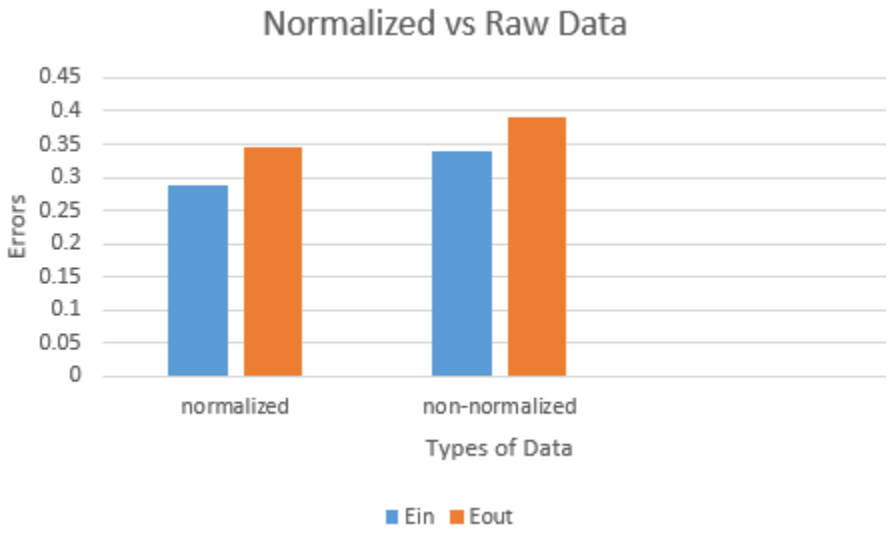

Figure 12: Normalization

Q.So why does this happen ?
The scaled version of the data contains samples in a desired range , here, we have scaled the data range to be in [-1,1] which massively cuts down the chances of misclassifications as the initial data has a wide range and may cause computational overhead.
The main advantage of scaling is to avoid attributes in greater numeric ranges dominating those in smaller numeric ranges. Another advantage is to avoid numerical difficulties during calculation. Because kernel values usually depend on the inner products of feature vectors, e.g. the linear kernel and the polynomial kernel.

4.1.2 Standardization

A weight magnitude is determined not only by the linear relationship between the corresponding input variable with the target, but also by the variable's range. The weight of each input variable has significant effect on target variable. If we could remove the effect of the ranges on the weight magnitudes, we could use the weight magnitudes to judge the relative importance of each input variable. We can do this with a common approach, this approach is to standardize the input variables by adjusting the values to have zero mean and unit variance. To do this correctly when partitioning data into training and testing sets, we must always calculate means and standard deviations using only the training set, and use the same means and standard deviations when standardizing the testing set. We must not use any information about the testing set when building a model. If we do, our test error will be lower than it will be when we truly see new data. We can do this by storing the means and standard deviations obtained from the training data, and just use them when standardizing the testing data. Python Code for the standardization function :

```
# Function for standardization
def standardize(origX):
    means = origX.mean(axis=0)
    stds = origX.std(axis=0)
    return (origX - means) / stds
```

Here first I included a function to get mean and then another function to get standard deviation. While I return a difference between original data and their mean divided by the standard deviation. I obtained a range of -3.6026 to 6.5643 for the given data set.

4.1.3 Why Standardization over scaling?

Standardized data does not have any bounds
In applications of boundary detections , standardization works better as it will have a bigger range than a scaled data.
Typical Standardization procedures equalize the range and/or data variability.

5 References

1. http://www.dataminingblog.com/standardization-vs-normalization/
2. http://stats.stackexchange.com/questions/41704/how-and-why-do-normalization-and-feature-scaling-work
3. https://www.biomedware.com/files/documentation/Preparing_data/Why_standardize_variables.htm
4. Lecture slides from CS 545
5. Mostafa, Y.,& Ismail, M. (2012). Learning from data: A short course.
6. Perceptron code obtained from Course website -
http://www.cs.colostate.edu/ cs545/fall15/doku.php?id=code:perceptron