CS 545 ASSIGNMENT 1

Paresh Bhambhani

September 17, 2015

Contents

1 The Perceptron Algorithm

For the first part of the assignment, four variants of Perceptron were implemented.

- Perceptron without bias
- Perceptron with bias
- Perceptron based on Pocket Algorithm
- Modified Perceptron

1.1 Implementation

1.1.1 Perceptron without bias

The idea behind perceptron is to iterate over the training examples, and update the weight vector w in a way that would make x_i more likely to be correctly classified.
The pseudo code for implementing Perceptron without a Bias is given below.

Algorithm 1: Perceptron without bias

```
input  : labeled data D in homogeneous coordinates
output: weight vector w
w = 0
converged = False
while not converged and number of iteration $< T$ do
    converged = True
    for i in 1,...,$|D|$: do
        if $x_i$ is misclassified: then
            update w and set converged = false
        end
    end
end
return w
```

For the complete code refer Appendix A.1.

1.1.2 Perceptron with bias

To introduce bias we add an extra dimension to x_0 and set it to 1 and then learn the weight vector of dimension $d + 1$. The first element of the weight vector is the bias. So when we do $\tilde{\mathbf{w}}.\tilde{\mathbf{x}}$ we get $w_0 + \mathbf{wx}$ where w_0 is the bias. Pseudo code is as below.

Algorithm 2: Perceptron with bias

```
input  : labeled data D in homogeneous coordinates
output: weight vector w

$\tilde{x} = (1, x_1, ..., x_d)$
$\tilde{w} = (w_0, w_1, ..., w_d)$
$\tilde{w} = 0$
converged = False
while not converged and number of iteration $< T$ do
    converged = True
    for i in 1,...,$|D|$: do
        if $\tilde{x}_i$ is misclassified: then
            update $\tilde{w}$ and set converged = false
        end
    end
end
return w
```

For the complete code refer Appendix A.2.

1.1.3 Perceptron based on Pocket Algorithm

In pocket algorithm the weight vector is updated only once per iteration as compared to *d* times (dimension of data sample) in regular perceptron. Pseudo code for pocket Algorithm is as presented below.

Algorithm 3: Perceptron based on Pocket Algorithm

```
input  : labeled data D in homogeneous coordinates
output: weight vector w

w = 0, $w_{pocket}$ = 0
converged = False
while not converged and number of iteration $< T$ do
    converged = True
    for i in 1,...,$|D|$: do
        if $x_i$ is misclassified: then
            update w and set converged = false
            if w leads to better $E_i n$ than $w_{pocket}$ then
                $w_{pocket} = w$
            end
        end
    end
end
return w
```

For the complete code refer Appendix A.3.

1.1.4 Modified Perceptron

The pseudo code for the modified Perceptron is given below.

Algorithm 4: Modified Perceptron

```
input  : labeled data D in homogeneous coordinates
output: weight vector w

w = 'small random values'
constant $c$ s.t. $0 < c < 1$
converged = False
while not converged and number of iteration $< T$ do
    converged = True
    for i in 1,...,$|D|$: do
        calculate $\lambda_i = y_i \mathbf{w}(t)^\intercal \mathbf{x}_i$
    end
    for i for which $\lambda_i < c||w||$ do
        choose example $j$ that maximizes $\lambda_i$
    end
    update the weight vector w based on example $j$
end
return w
```

In modified perceptron we need to choose small random values for w. I have normalized the weight vector in my implementation so that I am assured that the random values will be bounded and secondly while checking $\lambda_i < c||w||$ since $||w||$ is 1, I only have to check for $\lambda_i < c$.

For the complete code refer Appendix A.4.

1.2 Accuracy

To compare the accuracy of the four algorithms it is necessary that they be provided with the exact same data to work with. To achieve this, the four codes listed in appendix A were compiled in one master code and provided with the same data. It was run for ten iterations for heart data and five iterations for Gisette data and the Mean error was calculated. Mean In-sample and Mean Out -of-sample error for the four perceptron algorithms on Heart and Gisette data is shown in the Table 1. For Heart data 100 samples were used as test data and 170 samples were used as training data. For Gisette data 1500 samples were used as testing data and 4500 samples were used as training data. The sample error was calculated using the code:

```
def insample_error(self,X,y,w):
N = len(X)
mismatch = 0
for i in range(len(X)) :
if int(np.sign(np.dot(w,X[i]))) != int(np.sign(y[i])):
mismatch += 1
in_error = mismatch / float(N)
return in_error
```

An input of Train data and train labels will give the in sample error and an input of test data and test labels would give the out of sample error.

The graph shown in Figure 1 and Figure 2 show the in-sample and out-of-sample error plot for multiple runs for Heart and Gisette data respectively. Table 1 lists the mean error.

Table 1: Accuracy of different implementations of Perceptron

Data Set	Perceptron Flavor	Converges	Mean E_{in}	Mean E_{out}
Heart Data	Perceptron no bias	No	0.33	0.345
	Perceptron with bias	No	0..274	0.289
	Pocket Perceptron	No	0.152	0.189
	Modified Perceptron	No	0.499	0.501
Gisette Data	Perceptron no bias	Yes	0	0.034
	Perceptron with bias	Yes	0	0.034
	Pocket Perceptron	Yes	0	0.034
	Modified Perceptron	No	0.5	0.5

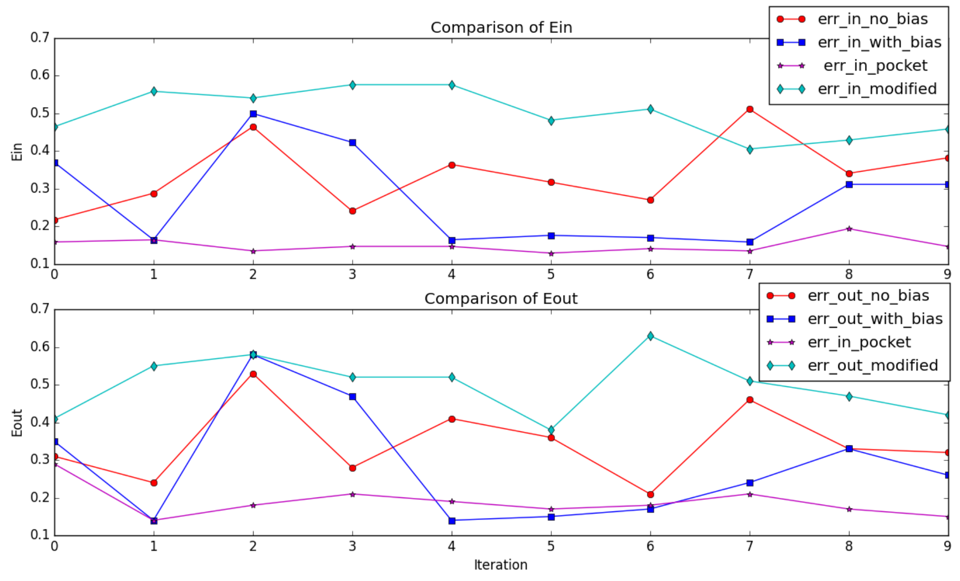

Figure 1: Accuracy of 4 variation of perceptron on Heart data

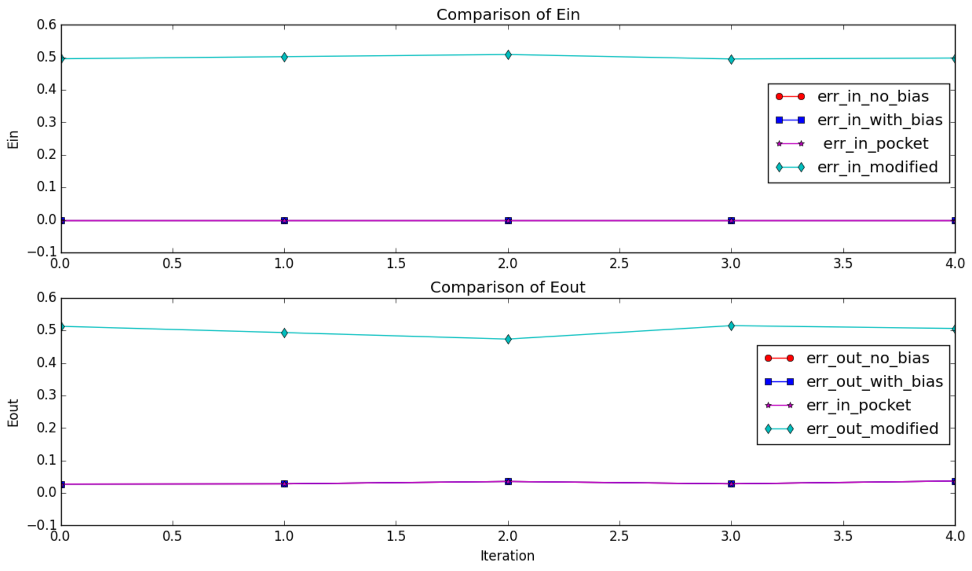

Figure 2: Accuracy of 4 variation of perceptron on Gisette data

From Table 1 we can infer that the Perceptron with pocket algorithm works best for us as it gives the lowest in-sample and out-of-sample error. This confirms the intuition that Pocket will work better since it updates the pocket weight only if that leads to better performance than the previous values else it retains the previous value that gave the lowest error. The modified perceptron unlike other variations, updates the weight vector once per iteration. It chooses the misclassified sample closest to it and then updates the weight vector according to that sample.
For Gisette data set it is observed that the Modified Perceptron does not converge in 100 iterations where other algorithms converge in less than 20. This might be because for Gisette dataset, the feature values have large variance. So, when the perceptron updates the weight vector according to the nearest misclassified value, it will take a lot many iterations for it to update the vector for the furthest value hence it does not converge soon (100 iterations here).

2 Learning Curves

2.1 Observations: Learning Curve plot

Results in the graph shown in Figure 3 show a plot of classifier accuracy (Out-of-sample error) as a function of number of training examples plotted for perceptron with bias that has been run on Gisette data. Random 3000 samples were selected as test data and remaining 3000 acted as the super set of training samples out of which an increasing number of training-samples were extracted and used for training of the perceptron. For the code for the learning curve Refer Appendix B.

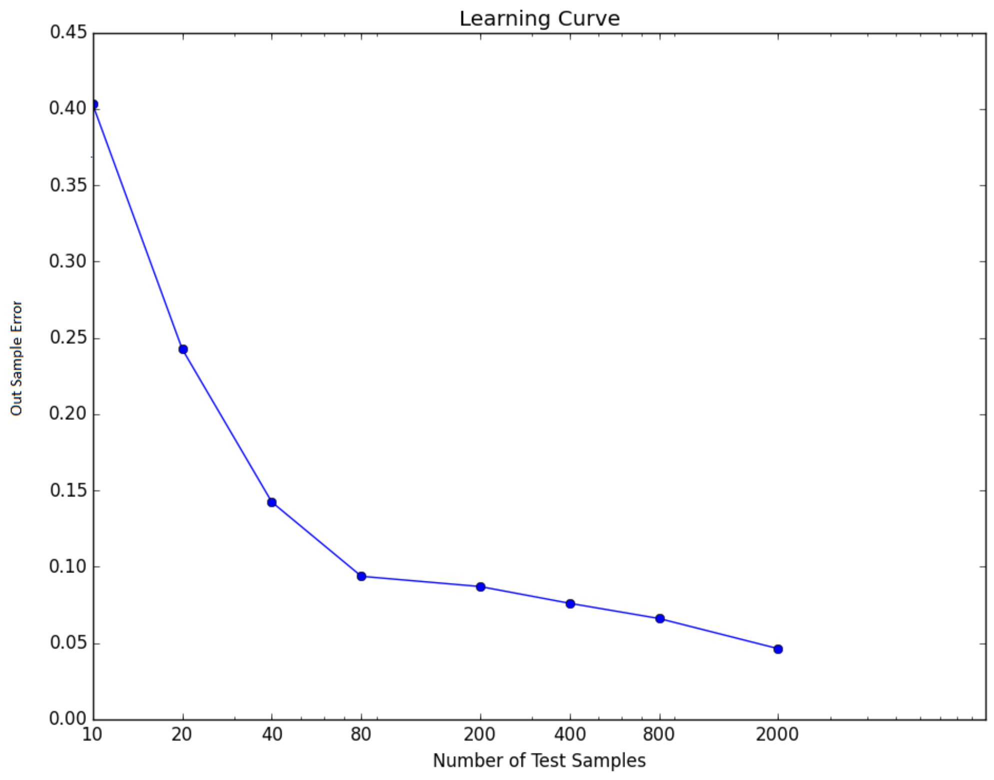

Figure 3: Learning Curve

2.2 Conclusion from Learning Curve Plot

It can be seen from the Learning curve plot that as the number of training samples increases the out-of-sample error decreases that is the perceptron Algorithm learns better and gets more accurate. This is so, because the weight vector gets more accurate with more training samples.

3 Data Normalization

3.1 How to scale data?

Mathematically the formula for scaling is:

$$\mathbf{x}_{scaled} = \frac{\mathbf{x} - \mathbf{x}_{min}}{\mathbf{x}_{max} - \mathbf{x}_{min}}$$

In Python one can use the MinMaxScaler() to scale the features to a given range.

3.2 Accuracy of Perceptron on Original vs Scaled data

Results in the graph shown in Figure 4 show a plot of in-sample error and out-of-sample error of perceptron algorithm with a bias for 10 runs for Original data as well as Scaled data. For python code for this implementation refer appendix C.

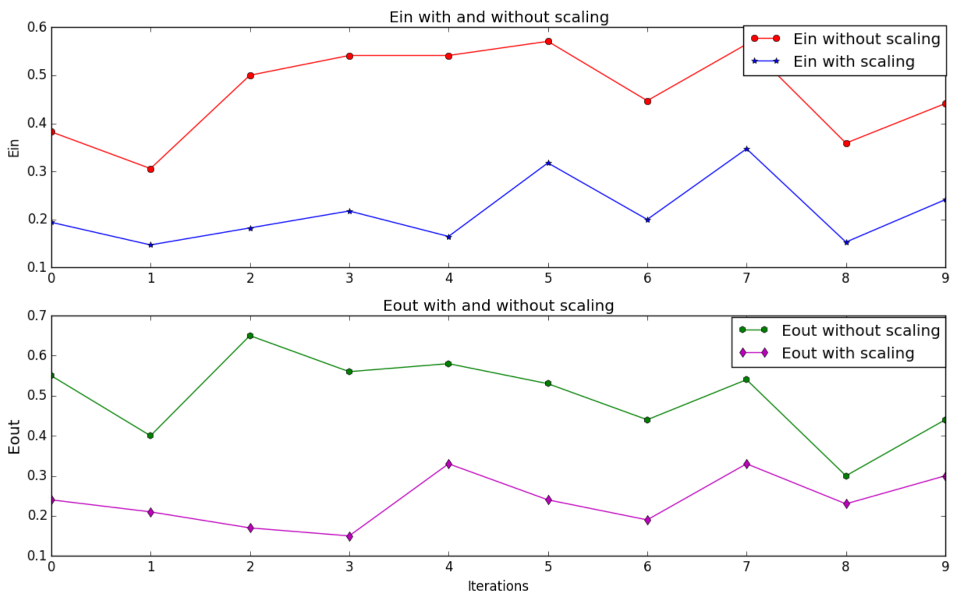

Figure 4: Accuracy of perceptron on Original and Scaled data

It can be seen that the algorithm has lower error on scaled data than on original data. One can conclude that scaled data leads to a better performance of the perceptron. We scale data to ensure that we have the same range of values for each of the feature inputs. Scaling reduces the standard deviation in the features of the sample data which suppresses the effect of outliers.

3.3 Standardization vs Normalization

Mathematically Standardization is done as:

$$\mathbf{X}_{new} = \frac{\mathbf{X} - \mu}{\sigma}$$

where μ is the mean data and σ is the standard deviation. This results in the data having zero mean and unit variance. Standardization is important when we are comparing measurements that have different units. To think of a condition where Standardization would be a better idea than scaling, consider the following hypothesis. We have a data where all the samples are 'good' that is the variance is under control but then we have one value which due to noise is extremely large (or small). If we scale the data then the extreme value will be assigned 1 (-1 if its extremely small) and it would result in erroneous results. But if you consider standardization then the extreme value be $>$ 1 or 1.5. Based on the data we are handling we can decide a threshold to eliminate such extreme samples which we are sure are a product of noise.
Consider an example: we have a data D =]10,40,500,700,9999999999] where the last entry is due to noise. If you run Scaling on this data, it will assign 1 to the last entry and -1 to the first and scale the rest accordingly. Instead if you standardize it, for the last value the standardized value will come out to be 1.788 while rest all the values are very small. Hence Standardization will give us an indication of an anomaly in the data which we can remove and get the correct results unlike scaling.

Appendices

Provided in this appendix are the python code for variation of Perceptron that have been run on Gisette data. The code when run on Heart data needs a few modification of extracting the training and testing data and labels from the heart.data file. It has been omitted in this report firstly to avoid repetition since it is almost same as the below code and secondly to keep the report length in check. If required for grading, all the codes can be sent as a separate attachment.

A Python Codes for different Perceptron Implementations

A.1 Perceptron No Bias

```
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
iterations += 1
self.converged = converged
if converged :
print 'Converged in %d iterations ' % iterations
else :
print 'Not converged'

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

if __name__=='__main__' :
X = np.genfromtxt("/***PATH***/gisette_train.data")
y = np.genfromtxt("***PATH***/gisette_train.labels")
p = Perceptron()
p.fit(X,y)
```

A.2 Perceptron with Bias

```
import numpy as np
```

```
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
iterations += 1
self.converged = converged
if converged :
print 'Converged in %d iterations ' % iterations
else :
print 'Not converged'

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

if __name__=='__main__' :
data = np.genfromtxt("****PATH****/gisette_train.data")
X = np.c_[np.ones(len(data[:,0])),data]
y = np.genfromtxt("****PATH****/gisette_train.labels")
p = Perceptron()
p.fit(X,y)
```

A.3 Perceptron based on Pocket Algorithm

```
import numpy as np

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]))
  self.w_pocket = 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]
  E_in_w = self.insample_error(X,y,self.w)
  E_in_w_pocket = self.insample_error(X,y,self.w_pocket)
  if E_in_w < E_in_w_pocket:
  self.w_pocket = self.w
  converged = False
  iterations += 1
  self.converged = converged
  if converged :
  print 'converged in %d iterations ' % iterations
  else:
  print 'Not converged'

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

def insample_error(self,X,y,w):
N = len(X)
mismatch = 0
for i in range(len(X)) :
if int(np.sign(np.dot(w,X[i]))) != int(np.sign(y[i])):
mismatch += 1
in_error = mismatch / float(N)
return in_error

if __name__=='__main__' :
data = np.genfromtxt("****PATH****/gisette_train.data")
X = np.c_[np.ones(len(data[:,0])),data]
y = np.genfromtxt("****PATH****/gisette_train.labels")
p = Perceptron()
p.fit(X,y)
```

A.4 Modified Perceptron

```
import numpy as np

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=[]
for i in range(len(X[0])):
self.w.append(np.random.uniform(-1,1))
c = np.random.rand()
converged = False
```

```
iterations = 0
while (not converged and iterations < self.max_iterations) :
self.w = preprocessing.normalize(self.w,norm='l2')
Lambda= np.zeros(len(X))
flag = 1
index=0
converged = True
max_lambda=-10000000000
for i in range(len(X)) :
Lambda[i] = y[i] * self.discriminant(X[i])
if Lambda[i] < c:
if Lambda[i] > max_lambda:
max_lambda = Lambda[i]
index=i
flag = 0
converged = False
if flag == 0:
self.w = self.w + y[index] * self.learning_rate * X[index]
iterations += 1
self.converged = converged
if converged :
print 'Converged in %d iterations ' % iterations
else :
print 'Not converged'

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

def insample_error(self,X,y,w):
N = len(X)
mismatch = 0
for i in range(len(X)) :
if int(np.sign(np.dot(w,X[i]))) != int(np.sign(y[i])):
mismatch += 1
in_error = mismatch / float(N)
return in_error

if __name__=='__main__' :
data = np.genfromtxt("****PATH****/gisette_train.data")
y = np.genfromtxt("****PATH****/gisette_train.labels")
p = Perceptron()
p.fit(X,y)
```

B Code for Learning Curve Implementation

```
import numpy as np
import matplotlib.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
iterations += 1
self.converged = converged
if converged :
print 'No. of training samples taken are %d' % len(X)
print 'The algorithm converged in %d iterations ' % iterations
else :
print 'not converged'

""" Define the product w(transpose) and x"""

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

"""In sample error which is calculated as the no. of mismatches divided by the total no of training examples"""

def insample_error(self,X,y,w):
N = len(X)
mismatch = 0
for i in range(len(X)) :
if int(np.sign(np.dot(w,X[i]))) != int(np.sign(y[i])):
mismatch += 1
in_error = mismatch / float(N)
return in_error

"""Out sample error is the in sample error calculation performed on the test data"""

def outsample_error(self,X,y,w):
err=[]
err.append(p.insample_error(X,y,w))
return np.mean(err)

"""To generate the logarithmic scale on x-axis depending on the no. of training samples"""
```

```
def generate_x_axis(data):
Num_ex = []
initial_tick = 10
log_base = 2
Num_ex.append(initial_tick)
while (initial_tick < len(train_data)):
for k in range (1,4):
if (initial_tick*(log_base**k)) > len(train_data):
break
Num_ex.append(initial_tick*(log_base**k))
initial_tick = initial_tick*10
return Num_ex

if __name__=='__main__' :
p = Perceptron()
data = np.genfromtxt("****PATH****/gisette_train.data")
labels = np.genfromtxt("****PATH****/gisette_train.labels")
I = np.arange(len(data))
np.random.shuffle(I)
data = data[I]
labels = labels[I]
train_data = data[0:3000,:]
test_data = np.c_[np.ones(len(data[3000:len(data),0])),data[3000:len(data),:]]
train_labels = labels[0:3000]
test_labels = labels[3000:len(labels)]
Num_ex = generate_x_axis(data)
Error = []
for i in Num_ex :
if i <= len(train_data):
X_train = np.c_[np.ones(len(train_data[0:i,0])),train_data[0:i,:]]
y_train = train_labels[0:i]
p.fit(X_train,y_train)
Error.append(p.outsample_error(test_data,test_labels,p.w))
fig = plt.figure(figsize=(5,5))
plt.plot(Num_ex, Error, marker='o')
plt.xscale('log')
plt.xticks(Num_ex,('10','20','40','80','200','400','800','2000'))
plt.xlabel('Number of Test Samples')
plt.ylabel('In Sample Error')
plt.title('Learning Curve')
plt.show()
plt.savefig("LearningCurve.jpg")
```

C Code for Perceptron Accuracy on original vs scaled data

```
import numpy as np
from sklearn.preprocessing import MinMaxScaler
import matplotlib.pyplot as plt

class Perceptron :

def __init__(self, max_iterations=1000, 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
iterations += 1
self.converged = converged
if converged :
print 'converged in %d iterations ' % iterations
else:
print 'not converged'

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

"""In sample error which is calculated as the no. of mismatches divided by the total no of training examples"""

def insample_error(self,X,y,w):
N = len(X)
mismatch = 0
for i in range(len(X)) :
if int(np.sign(np.dot(w,X[i]))) != int(np.sign(y[i])):
mismatch += 1
in_error = mismatch / float(N)
return in_error

"""Out sample error is the in sample error calculation performed on the test data"""

def outsample_error(self,X,y,w):
err=[]
err.append(p.insample_error(X,y,w))
return np.mean(err)

"""Extract the original and scaled data test and training sets from the heart data"""

def get_data (path):
```

```
data=np.genfromtxt(path, delimiter=",", comments="#")
I = np.arange(270)
np.random.shuffle(I)
data = data[I]
y_train = data[0:170,1]
y_test = data[170:270,1]
X_train_temp = data[0:170,2::]
X_test_temp = data[170:270,2::]
X_test = np.c_[np.ones(len(X_test_temp[:,0])),X_test_temp]
X_train = np.c_[np.ones(len(X_train_temp[:,0])),X_train_temp]
X_train_temp_tr = X_train_temp.T
X_test_temp_tr = X_test_temp.T
X_train_scale_tr = np.zeros(shape=(len(X_train_temp_tr),len(X_train_temp_tr.T)))
X_test_scale_tr = np.zeros(shape=(len(X_test_temp_tr),len(X_test_temp_tr.T)))
std_scale = MinMaxScaler((-1,1),copy=True)
for i in range(len(X_train_temp_tr)):
scale_arr_a = []
scale_arr_a = (std_scale.fit_transform(X_train_temp_tr[i]))
X_train_scale_tr[i] = scale_arr_a
for i in range(len(X_test_temp_tr)):
scale_arr_b = []
scale_arr_b = (std_scale.fit_transform(X_test_temp_tr[i]))
X_test_scale_tr[i] = scale_arr_b
X_train_scale = X_train_scale_tr.T
X_test_scale = X_test_scale_tr.T
X_train_scale = np.c_[np.ones(len(X_train_scale[:,0])),X_train_scale]
X_test_scale = np.c_[np.ones(len(X_test_scale[:,0])),X_test_scale]
return X_test, X_test_scale, X_train, X_train_scale, y_test, y_train

if __name__=='__main__' :
in_sample_err_without_scaling = []
out_sample_err_without_scaling = []
in_sample_err_with_scaling = []
out_sample_err_with_scaling = []
for i in range (1,11):
X_test, X_test_scale, X_train, X_train_scale, y_test, y_train = get_data ("****PATH****/heart.data")
p = Perceptron()
p.fit(X_train,y_train)
in_sample_err_without_scaling.append(p.insample_error(X_train,y_train,p.w))
out_sample_err_without_scaling.append(p.outsample_error(X_test,y_test,p.w))
p.fit(X_train_scale,y_train)
in_sample_err_with_scaling.append(p.insample_error(X_train_scale,y_train,p.w))
out_sample_err_with_scaling.append(p.outsample_error(X_test_scale,y_test,p.w))
fig = plt.figure(figsize=(5,5))
plt.subplot(2,1,1)
plt.title('Ein with and without scaling')
plt.plot(in_sample_err_without_scaling, marker='o', label='Ein without scaling',color='r')
plt.plot(in_sample_err_with_scaling, marker='*', label='Ein with scaling',color='b')
plt.ylabel('Ein')
plt.subplot(2,1,2)
plt.plot(out_sample_err_without_scaling, marker='h', label='Eout without scaling',color='g')
plt.plot(out_sample_err_with_scaling, marker='d', label='Eout with scaling',color='m')
plt.title('Eout with and without scaling')
plt.ylabel('Ein')
plt.xlabel('Iterations')
plt.show()
```