Introduction to Machine Learning - Homework 1

Savinay Shukla

Q1) Below are the snapshots of the polyreg function observed with different values of **d :**

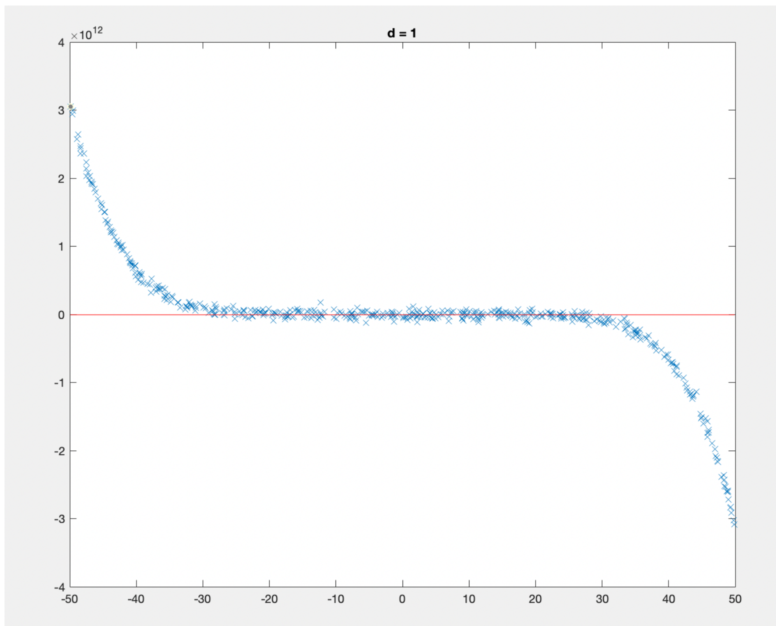

Figure 1.1 - polyreg plot with d = 1.

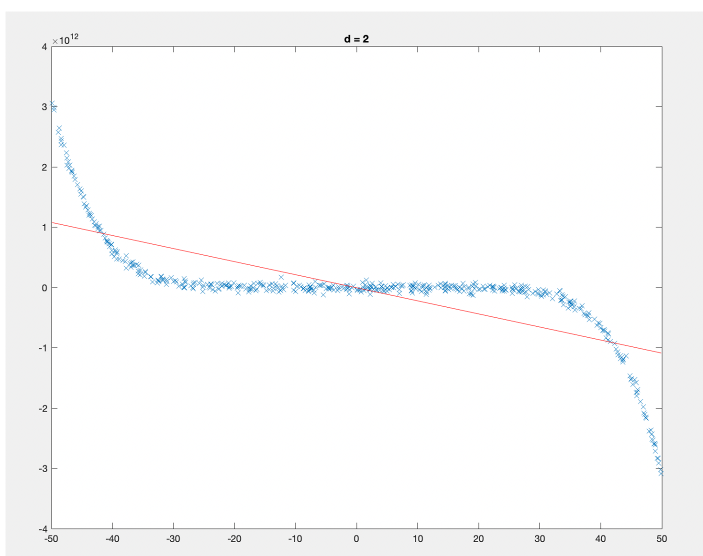

Figure 1.2 - polyreg plot with d = 2.

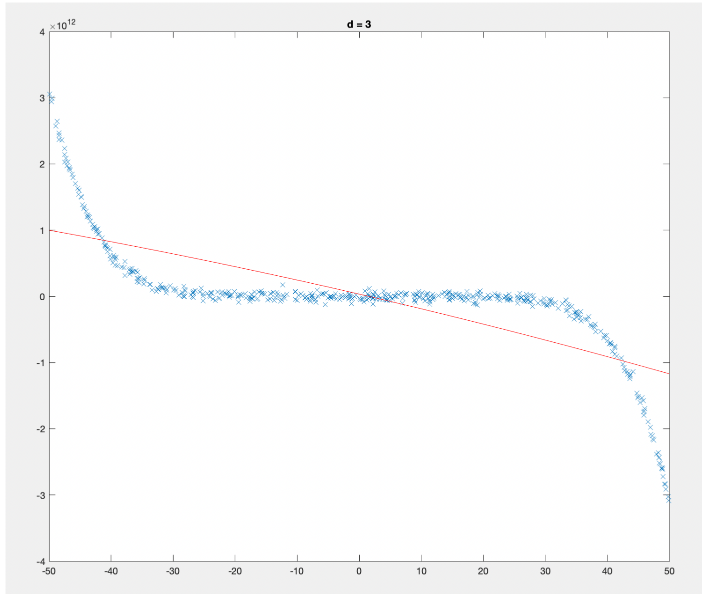

Figure 1.3 Polyreg graph with d = 3

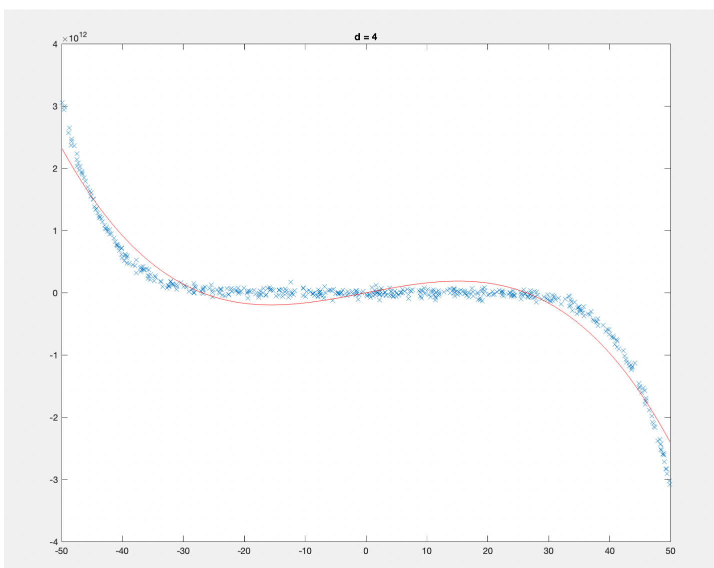

Figure 1.4 polyreg graph with d = 4.

We can observe that the polynomial function begins to closely approximate the data distribution as we increase the value of "**d**".

Below is the use of a cross-validation plot with various values of "**d**" to find the best order of polynomial that would fit this data.

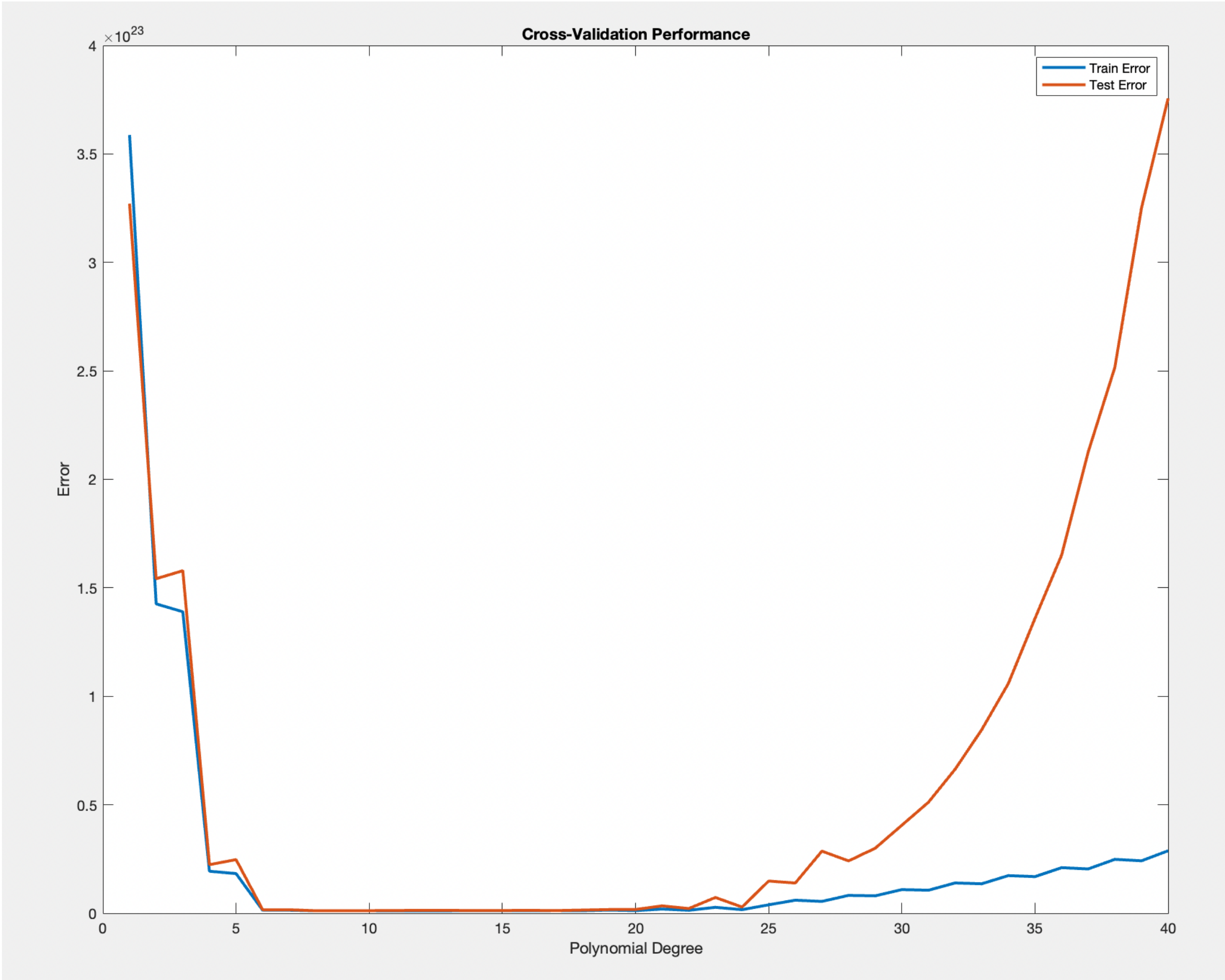

Figure 1.5 Cross-validation

As we can see from the plot above, the minimum error for the testing data is coming at the **degree = 6.**

For degrees > 6, we find that the test error starts to increase rapidly, and begins to overfit the model. So according to the data given, **we can say that the best order of the polynomial = 6, i.e. *d* = 6**. We can confirm this by plotting the curve against the data distribution. The curve will fit almost perfectly with the data at d = 6.

Below is the graph plot of with d = 6.

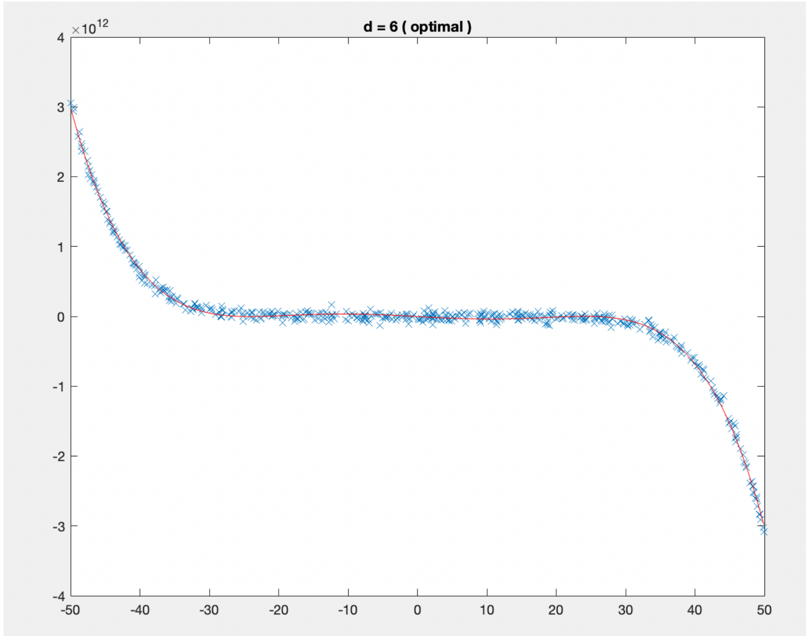

Figure 1.6 The polyreg function plot with optimal value of d = 6.

<u>Source Code (MATLAB) :</u>

```
clear all;
load("problem1.mat");
indxs = crossvalind("Kfold",length(x), 2);
xtrain = x(indxs == 1);
xtest = x(indxs == 2);
ytrain = y(indxs == 1);
ytest = y(indxs == 2);
trainError = [];
testError = [];
iterations = [];
for i=1:40
   [err,model,errT] = polyreg(xtrain,ytrain,i,xtest,ytest);
   trainError = [trainError, err];
   testError = [testError, errT];
   iterations = [iterations,i];
end
figure
plot(iterations,trainError,iterations,testError,'LineWidth',2.0)
xlabel("Polynomial Degree");
ylabel("Error");
legend("Train Error", "Test Error");
title("Cross-Validation Performance");
```

Polyreg.m

```
function [err,model,errT] = polyreg(x,y,D,xT,yT)
%
% Finds a D-1 order polynomial fit to the data
%
%    function [err,model,errT] = polyreg(x,y,D,xT,yT)
%
% x = vector of input scalars for training
% y = vector of output scalars for training
% D = the order plus one of the polynomial being fit
% xT = vector of input scalars for testing
% yT = vector of output scalars for testing
% err = average squared loss on training
% model = vector of polynomial parameter coefficients
% errT = average squared loss on testing
%
% Example Usage:
%
% x = 3*(rand(50,1)-0.5);
% y = x.*x.*x-x+rand(size(x));
% [err,model] = polyreg(x,y,4);
%
xx = zeros(length(x),D);
for i=1:D
 xx(:,i) = x.^(D-i);
end
model = pinv(xx)*y;
err   = (1/(2*length(x)))*sum((y-xx*model).^2);
if (nargin==5)
 xxT = zeros(length(xT),D);
 for i=1:D
   xxT(:,i) = xT.^(D-i);
 end
 errT  = (1/(2*length(xT)))*sum((yT-xxT*model).^2);
end
q  = (min(x):(max(x)/300):max(x))';
qq = zeros(length(q),D);
for i=1:D
 qq(:,i) = q.^(D-i);
end
clf
   plot(x,y,'X');
hold on
if (nargin==5)
   plot(xT,yT,'cO');
end
   plot(q,qq*model,'r')
```

Q2)

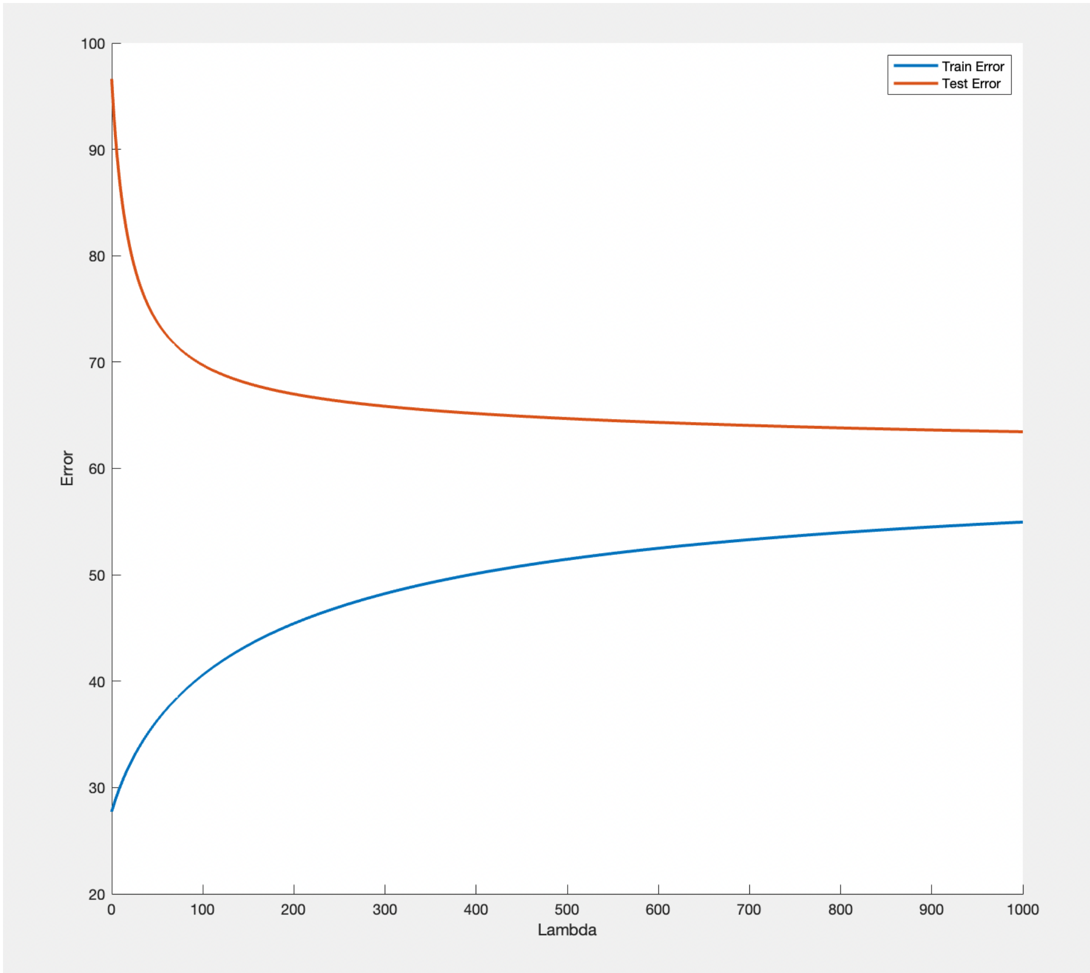

Figure 2.1 Error vs Penalty Parameter.

We can see that as we increase the penalty parameter – ***lambda***, we observe a decrease in testing error. We also observe an approximate converging trend of error curves (both test error curve and train error curve) with the above graph plot.

Additionally, **we see that after an approximate value of lambda = 750, there is not much improvement in test error**. However, this behavior is believed to be dependent on how the test-train data was initially split.

Below is an example of the test-train error curve observed after running the source code multiple times.

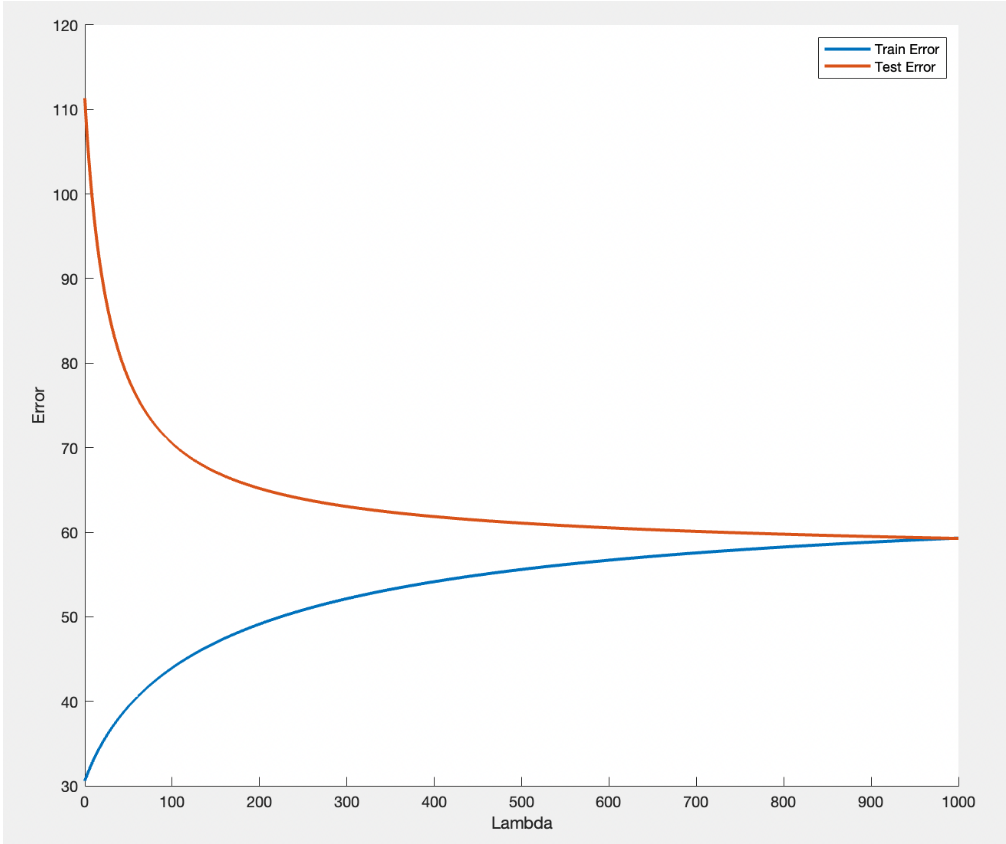

Figure 2.2 Converging Error vs Penalty Parameter

In the above example, we observe a convergence of test errors and train errors **at lambda = 1000.**

Source Code(MATLAB)

Driver code:

```
clear all;
load("problem2.mat");
clc;
indxs = crossvalind("Kfold",length(x), 2);
x_train = x(indxs == 1,:);
x_test = x(indxs == 2,:);
y_train = y(indxs == 1);
y_test = y(indxs == 2);
trainErrors = [];
testErrors = [];
iterations = [];
%Creating a lambda Set from 0 - 1000 lambda values.
lambdaSet = 0:0.2:1000;
for i=lambdaSet
   [err,theta,errT] = regularizedReg(x_train,y_train,i,x_test,y_test);
   trainErrors = [trainErrors, err];
   testErrors = [testErrors, errT];
   iterations = [iterations,i];
end
figure
hold on;
plot(iterations,trainErrors,iterations,testErrors,'LineWidth',2.0);
xlabel('Lambda');
ylabel('Error');
legend('Train Error', 'Test Error');
hold off;
```

---

regularizedReg.m

```
function [trainErr,theta,testErr] = regularizedReg(x,y,lambda,xT,yT)
% This method returns "trainErr" - regularized error during training.
%                     "theta" - vector of mulivariate coefficients.
%                     "testErr" - regularized error during testing.
% This method takes the following parameters:
%                     "x" - training feature vector;
%                     "y" - training target vector;
%                     "xT" - testing feature vector;
%                     "yT" - testing target vector;
%                     lambda - penalty parameter;
%
%
theta = (x' * x + lambda * eye(size(x'* x))) \ x' * y;
trainErr   = (1 / (2 * length(x))) * sum((y - x*theta).^2);
trainErr = trainErr + (lambda/(2*(length(x)))) * (theta' * theta);
```

```
if(nargin == 5)
   testErr   = (1 / (2 * length(xT))) * sum((yT - xT*theta).^2);
   testErr = testErr + (lambda/(2*(length(xT)))) * (theta' * theta);
end
end
```

Q3)

PROOF FOR : $g(-z) = 1 - g(z)$

$$g(z) = \frac{1}{1+e^{-z}}$$

$$= \frac{1}{1+\frac{1}{e^{z}}} = \frac{e^{z}}{1+e^{z}}$$

$$1 - g(z) = 1 - \frac{e^{z}}{1+e^{z}}$$

$$= \frac{1+e^{z}-e^{z}}{1+e^{z}} = \frac{1}{1+e^{z}} \rightarrow (1)$$

$$g(-z) = \frac{1}{1+e^{-(-z)}}$$

$$= \frac{1}{1+e^{z}} \quad — (2)$$

Since (1) = (2)

Hence: $g(-z) = 1 - g(z)$.

PROOF FOR: $g^{-1}(y) = \ln\left(\frac{y}{1-y}\right)$

Given that $g(z) = y$, we need to prove that $g^{-1}(y) = z$

$$= \ln\left(\frac{y}{1-y}\right) \quad \text{--- } 1$$

<u>Solving for $\ln\left(\frac{y}{1-y}\right)$:-</u>

$$\ln\left(\frac{y}{1-y}\right) = \ln(y) - \ln(1-y)$$

$$= \ln\left(\frac{1}{1+e^{-z}}\right) - \ln\left(1 - \frac{1}{1+e^{-z}}\right)$$

↳ Because $y = \frac{1}{1+e^{-z}}$

$$= \ln\left(\frac{1}{1+e^{-z}}\right) - \ln\left(\frac{e^{-z}}{1+e^{-z}}\right)$$

$$= \ln(1) - \ln(1+e^{-z}) - \ln(e^{-z}) + \ln(1+e^{-z})$$

$$= \ln(1) - \ln(e^{-z})$$

$$= -(-z)$$

$$= z$$

$$\ln\left(\frac{y}{1-y}\right) = z = g^{-1}(y)$$

Hence Proved!

Q4)

The gradient descent function is as follows :

$$\theta^1 = \theta^0 - \eta \nabla_\theta R_{emp}\Big|_{\theta^0}, \quad t = 1$$

So before we optimize our theta vector with the gradient descent algorithm, we need to find the derivative of our empirical risk function.

Given risk function is :

$$R_{emp}(\boldsymbol{\theta}) \quad = \quad \frac{1}{N}\sum_{i=1}^{N}(y_i - 1)\log(1 - f(\mathbf{x}_i;\boldsymbol{\theta})) - y_i \log(f(\mathbf{x}_i;\boldsymbol{\theta})).$$

Additionally the given classification function is:

$$f(\mathbf{x};\boldsymbol{\theta}) \quad = \quad \left(1 + \exp(-\boldsymbol{\theta}^\top \mathbf{x})\right)^{-1}$$

Below is described the way to obtain the derivative (gradients) for the given risk function:

Let $\theta^T x = t$

So after putting the value of $\theta^T x$ in $f(x;\theta)$ we get:-

$$f(t) = \frac{1}{1+e^{-t}}$$

Let $f(t) = s$

We can derive $R_{emp}(\theta)$ as follows:

$$\frac{d R_{emp}(\theta)}{d\theta} = \frac{d R_{emp}(\theta)}{ds} \times \frac{ds}{dt} \times \frac{dt}{d\theta}$$ (Product of partial derivatives)

Solving each derivative seperately :-

$$\frac{d R_{emp}(\theta)}{ds} = \frac{d}{ds}\left((y-1)\log(1-s) - y\log(s)\right)$$

$$= \frac{1-y}{1-s} - \frac{y}{s}$$

$$= \frac{s - ys - y + ys}{s(1-s)}$$

$$= \frac{s-y}{s(1-s)} \quad \text{--- (1)}$$

$$\frac{ds}{dt} = \frac{d}{dt}\left(\frac{1}{1+e^{-t}}\right)$$

$$= \frac{d}{dt}\left(\frac{e^t}{1+e^t}\right)$$

$$= \frac{e^t(1+e^t) - e^{2t}}{(1+e^t)^2}$$

$$= \frac{e^t + e^{2t} - e^{2t}}{(1+e^t)^2}$$

$$= \frac{e^t}{(1+e^t)^2} = \frac{e^t}{(1+e^t)} \times \frac{1}{(1+e^t)}$$

Since $\frac{e^t}{1+e^t} = s$

$\hookrightarrow = s \times (1-s)$

$$\therefore \frac{ds}{dt} = s(1-s) \quad \text{--- } (2)$$

$$\frac{dt}{d\theta} = \frac{d}{d\theta}(\theta^T x)$$

$$= x \quad \text{--- } (3)$$

$$\frac{d\,R_{emp}(\theta)}{d\theta} = (1) \times (2) \times (3)$$

$$= \left(\frac{s-y}{s(1-s)}\right) \times \left(s(1-s)\right) \times (x)$$

$$= (s-y)x$$

Substituting back the values of s and t,

$$\frac{d\,R_{emp}(\theta)}{d\theta} = \left(f(x;\theta) - y_i\right) x_i$$

$$\therefore \nabla R_{emp}(\theta) = \frac{1}{N} \sum \left(f(x;\theta - y_i\right) x_i$$

$\hookrightarrow$ To be used to find gradients.

Here are the graph plots for different values of **n** and **e**:

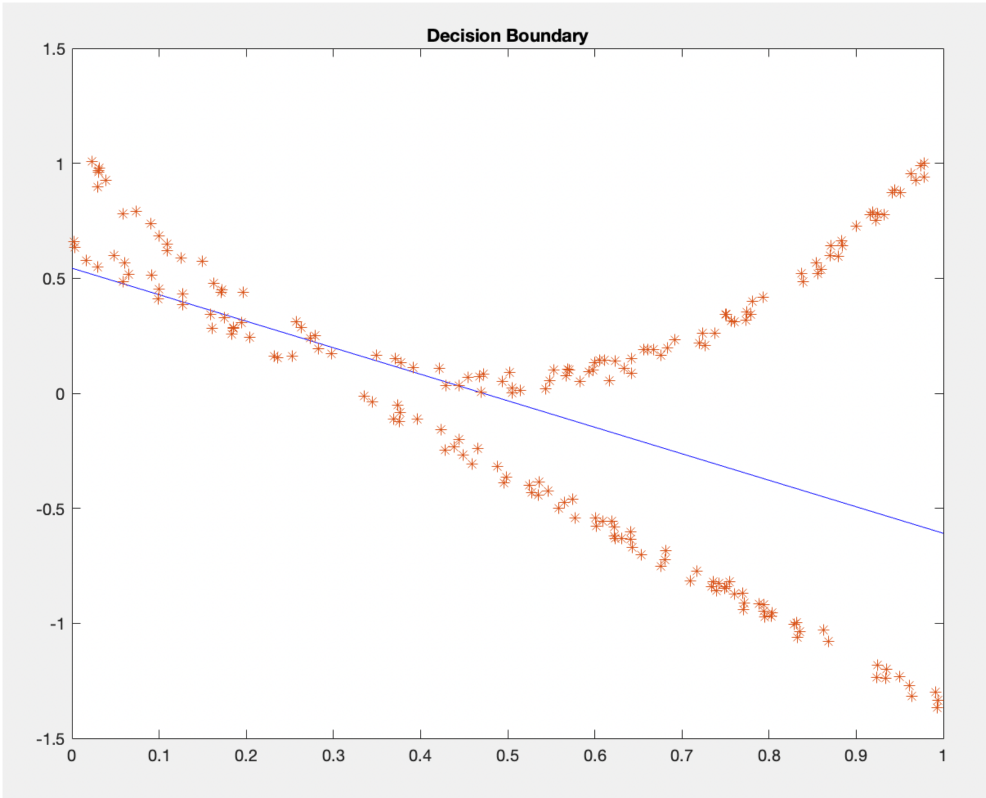

Figure 4.1 Decision boundary with n = 0.1 and epsilon = 0.001

The above graph is the decision boundary obtained when the gradient descent algorithm is applied with **n = 0.1** and **epsilon = 0.001**. The decision boundary is not accurately classifying the data as per these values of n and e.

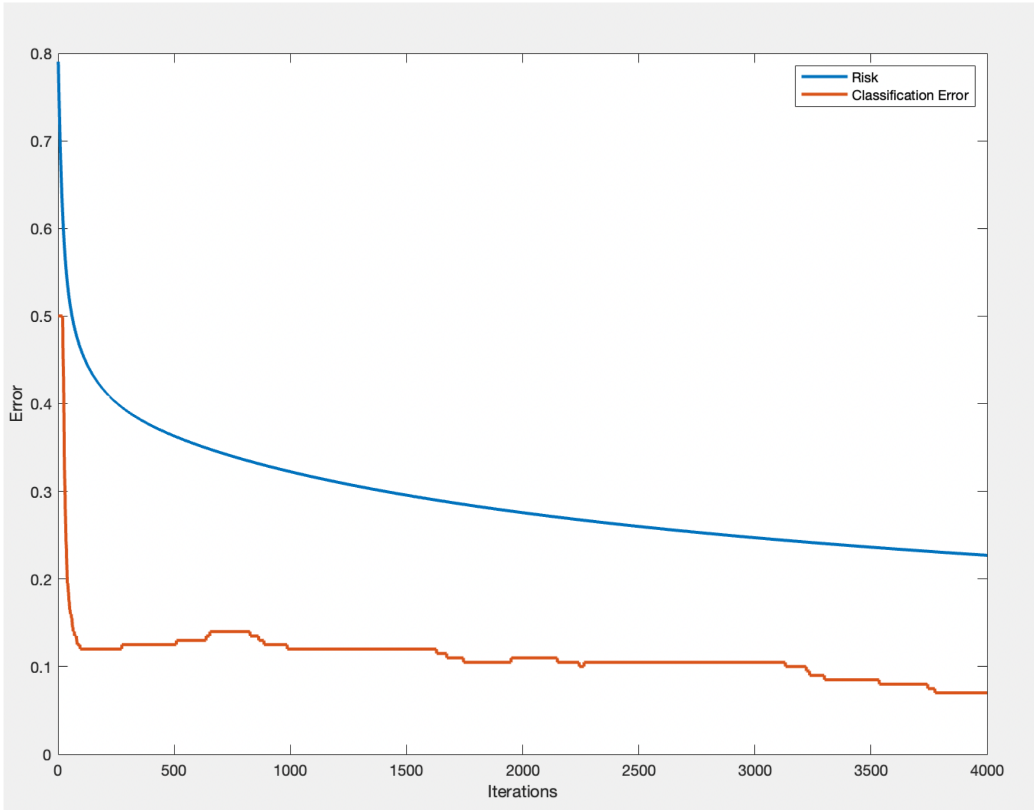

Figure 4.2 Risk and Classification Error versus Iterations.

The above is a risk plot and binary classification error plot for the same value of **n=0.01 and e=0.001**. Number of iterations recorded = approx 4000.

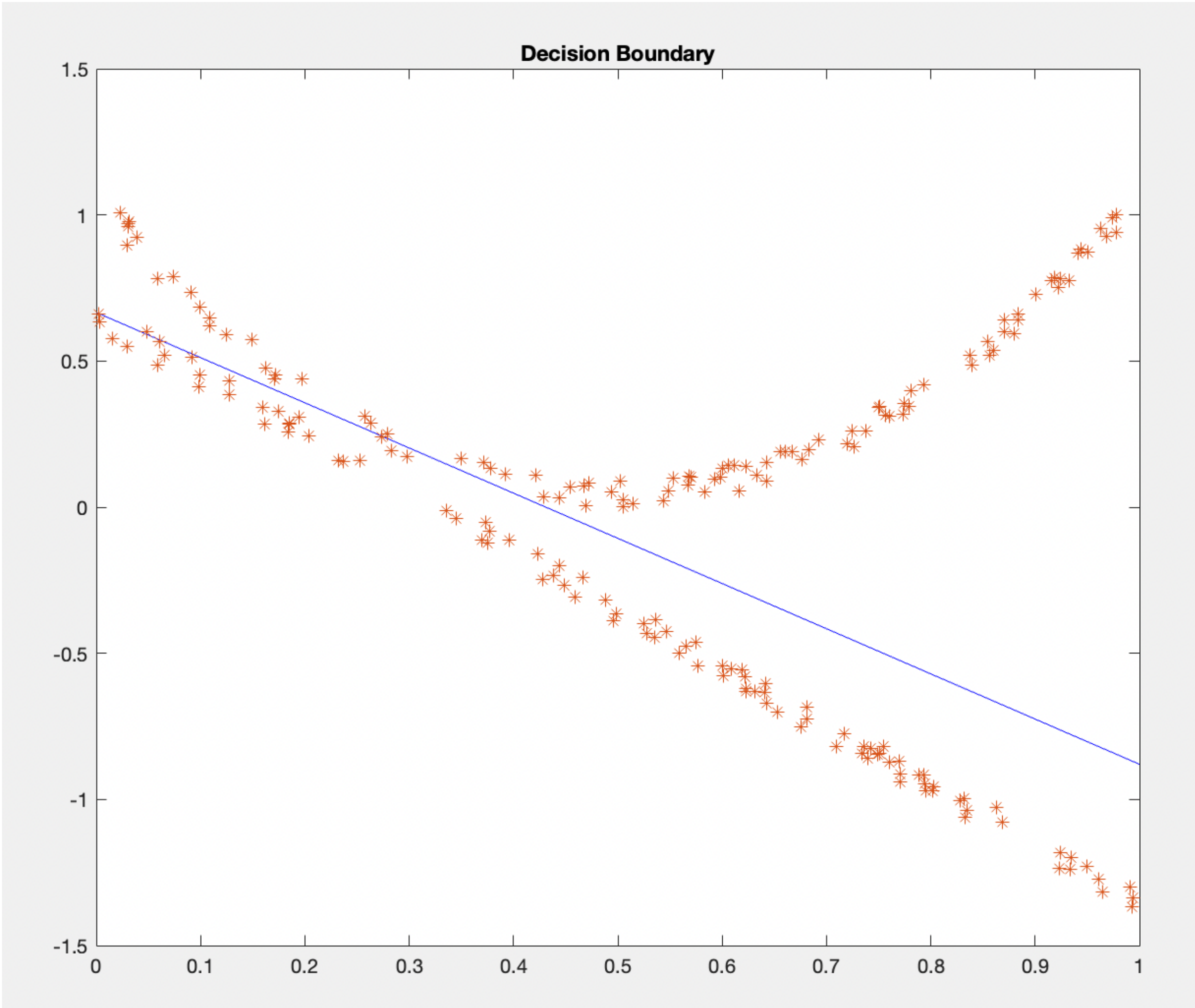

Figure 4.3 Decision Boundary Plot for n = 0.5 and epsilon = 0.001

The above is the decision boundary plot for **n = 0.5 and epsilon = 0.001**. We observe better accuracy with these values of n and epsilon than before. Recorded iterations were around 4000.

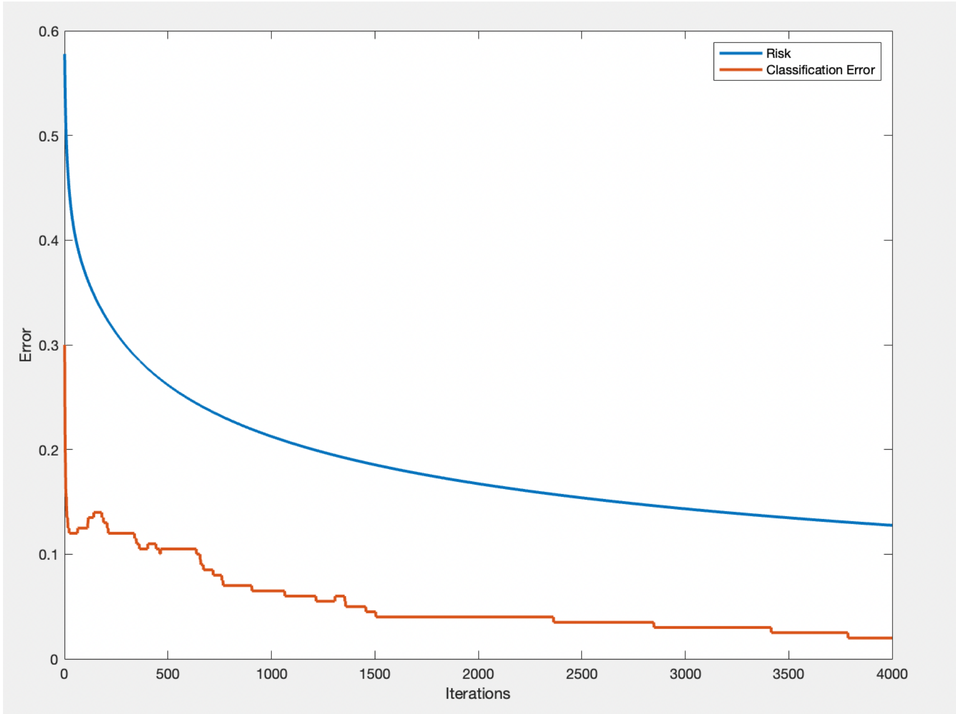

Figure 4.4 Risk and Classification Error versus Iterations for n = 0.5 and epsilon = 0.001

Below is the decision boundary plot for n=1 and epsilon = 0.001:

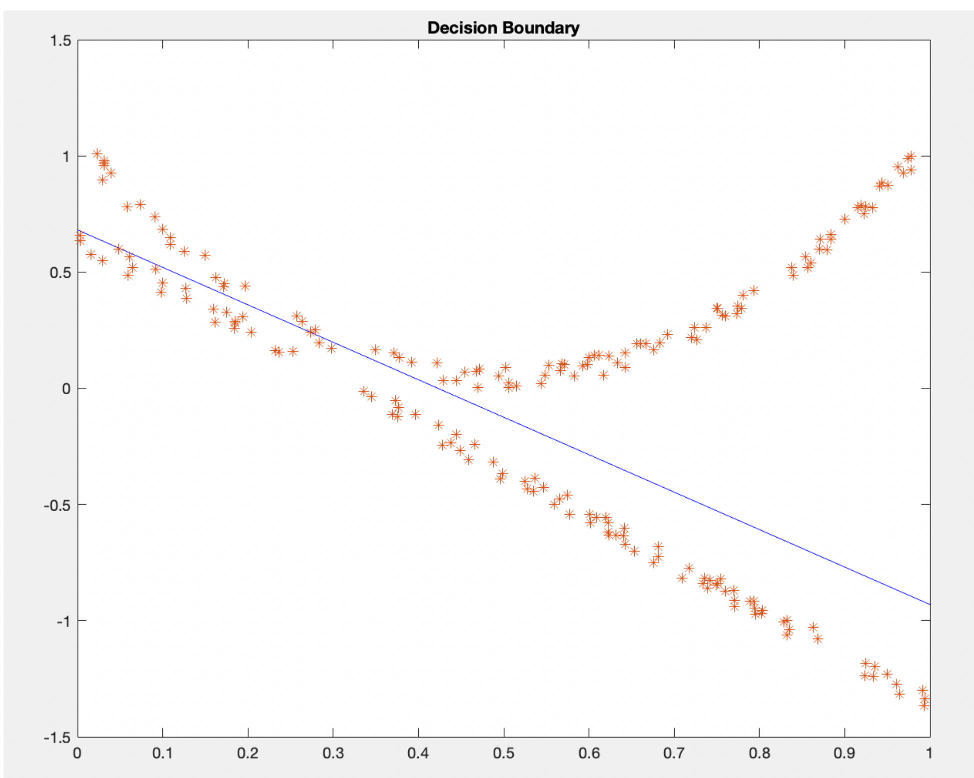

Figure 4.5 Decision Boundary Plot for n = 1 and epsilon = 0.001

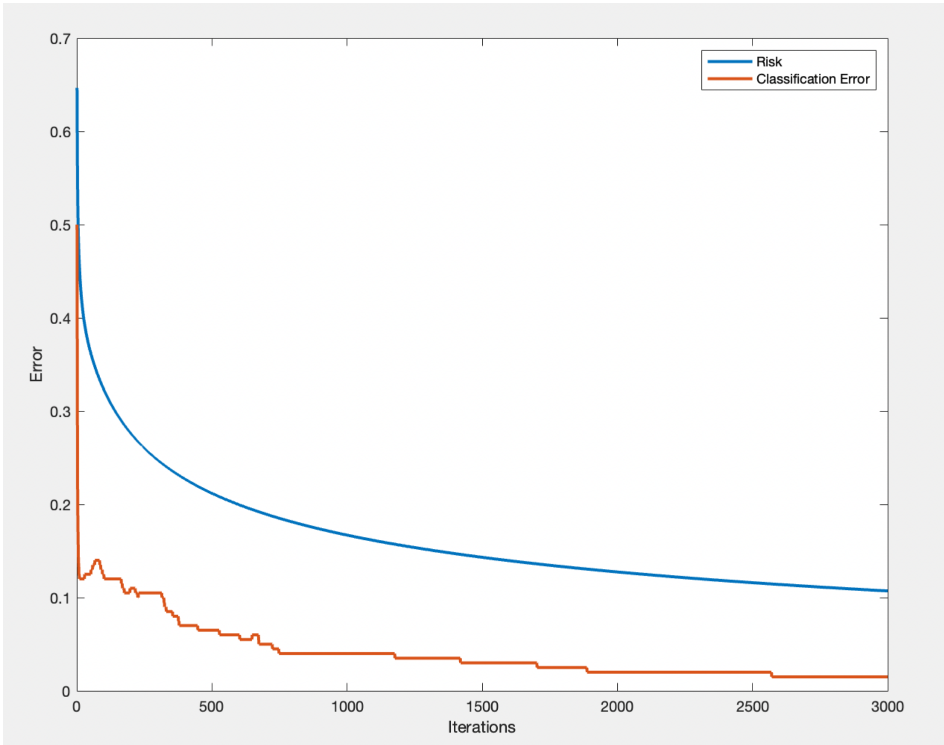

Figure 4.6 Risk and Classification Error vs Iterations for n = 1 and epsilon = 0.001

We have observed that with **n = 1 and epsilon = 0.001**, we get the **minimum classification error of 1%** and lowest risk at almost 3000 iterations. In addition, the decision boundary seems to classify the data points with most accuracy using these values of **n** and **e**.

Source :

Driver code:

```
%% Load DataSet
clear all;
load("dataset4.mat");
%% Parameters
stepSize =1;
epsilon = 0.001;
upperBoundLimit = 8000; %Set the maximum number of iterations
%% Training
[row, col] = size(X);
initialTheta = rand(size(X, 2),1); %random initialization of theta;
currTheta = initialTheta + epsilon;  %creating a currentTheta vector which has greater
values than initialTheta;
iterations = [];
costPerIteration = [];
errors=[];
i = 1;
while norm(currTheta - initialTheta) >= epsilon

   %an upperBound limit on number of iterations
   if i > upperBoundLimit
       break;
   end
   [risk, gradients] = computeCost(X, Y, initialTheta);
   prediction = 1 ./ ( 1 + exp (-X * initialTheta));
   prediction( prediction >= 0.5)  = 1;
   prediction( prediction < 0.5) = 0;
   %calculate the error = (sum of all predictions != Y)/length(Y)
   err = sum(prediction~=Y)/length(Y);
   errors = [errors, err];
   costPerIteration = [costPerIteration, risk];
   %Append the iteration count for plotting.
   iterations = [iterations, i];
   i = i + 1;
   %Update the theta vector;
   currTheta = initialTheta;
   initialTheta = initialTheta -  stepSize * gradients;
end
%For Plotting Risk and Classification Error graphs.
figure
plot(iterations, costPerIteration,iterations, errors,'LineWidth',2.0);
legend('Risk',"Classification Error");
ylabel('Error');
xlabel('Iterations');
figure
%For Plotting Decision Boundary
```

```
x = 0:0.001:1;
y = (-initialTheta(3) - initialTheta(1).*x)/initialTheta(2);
plot(x, y, "b"); hold on;
plot(X(:, 1), X(:, 2),"*");
title("Decision Boundary")
```

computeCost.m

```
%% Function for computing risk and gradients.
function [cost, gradients] = computeCost(x, y, theta)
h = sigmoid( x * theta);
cost = - (1 / length(x)) * sum ( y .* log(h) + (1-y).*log(1-h));
gradients = zeros(length(theta), 1);
for i = 1:size(gradients)
   gradients(i) = (1 / length(x)) * sum((h - y)' * x(:,i));
end
end
```

sigmoid.m

```
%% Function for Logistic Squashing functions.
function [sig] = sigmoid(x)
sig = 1 ./ (1 + exp(-x));
end
```