Report to Project-1

Yuanbo Han
October 8, 2017

Abstract

This report solves the two main problems, namely polynomial regression and ridge regression, in Project_1 of the course "Introduction to Statistical Learning and Machine Learning". The whole programming work is finished on MATLAB, which shows significant simplicity and efficiency while handling matrices.

While dealing with the first problem, I rewrite the regression function in matrix format from the very start, and thus the program need not modifying substantially as the model changes from linear to polynomial. It proves to be effective and efficient. At length, I draw the conclusion by comparing errors with different degrees that taking $\text{degree} = 3$ is the best fitting of polynomial regression.

When it comes to the second problem, I recall and make full use of the previous work in *Exerc_Chap2*[1], and design several subtle functions to solve the problem step by step. Finally, I choose $\delta^2 = 7.05$ for the ridge regression as it minimizes the test error in 5-fold cross-validation.

In addition, the programming codes with enough comments are all normalized, which provides great readability and clarity.

Key Words： least squares, polynomial regression, ridge regression, cross-validation

Contents

[1] Yanwei Fu. The Exercise of Chap 2. Introduction to Statistical Learning and Machine Learning, Sep. 24, 2017.

1 Sketch

Chapter **2** describes the solution of Problem 1. Section **2.1** adds a bias variable for linear regression. Section **2.2** modifies the model into polynomial basis and finds the best polynomial degree.

Chapter **3** reports the whole process of constructing a ridge regression model for Problem 2. Section **3.1** standardizes the original data. Section **3.2** computes theta's, training errors and test errors for a range of d2's. Section **3.3** improves the model by K-fold cross-validation and discusses the best d2.

Chapter **4** records the summaries and my own reflections during my work on the entire project.

Of course, Chapter **1** serves here as the sketch, briefly introducing the coming chapters and sections.

2 Linear Regression and Nonlinear Bases

2.1 Adding a Bias Variable

As the script *example_basis.m* shows, if we omitted the bias assuming that the y-intercept is zero, the training and test errors would respectively be 28122.82 and 28298.97, which seems not only poor but also ridiculous for a regression model. Such being the case, we first expect to improve the performance by adding a bias variable. Mathematically, we will take

$$\widehat{y_i} = w_2 x_i + w_1,$$

instead of

$$\widehat{y_i} = w x_i.$$

For convenience, let's rewrite the above equation in matrix form:

$$\hat{\mathrm{y}} = \mathrm{X}_1 \mathrm{W},$$

where $\mathrm{W} = [w_1\ w_2]$, and $\mathrm{X}_1 = [\mathbf{1}\ \mathrm{X}] = \begin{bmatrix} 1 & x_1 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix}$.

We first assume ${\mathrm{X}_1}^T\mathrm{X}_1$ is invertible. Then it can be proved through differentiation that when the *residual sum of squares (RSS)* defined by $\|\mathrm{y} - \hat{\mathrm{y}}\|_2^2$ meets its minimum,

$$\mathrm{W} = \left({\mathrm{X}_1}^T\mathrm{X}_1\right)^{-1}{\mathrm{X}_1}^T\mathrm{y}.$$

In fact, when ${\mathrm{X}_1}^T\mathrm{X}_1$ is singular, replace $\left({\mathrm{X}_1}^T\mathrm{X}_1\right)^{-1}$ with $\left({\mathrm{X}_1}^T\mathrm{X}_1\right)^{\dagger}$, the Moore-Penrose pseudoinverse, and the above equation still holds to minimize *RSS*.

Based on this theory, I write the function *leastSquaresBias*, which has the same input/model/predict format as the *leastSquares* function, but includes a bias variable w_1.

Having substituted *leastSquaresBias* for *leastSquares* in *example_basis.m*, the training and test errors become 3551.35 and 3393.87, and the new regression plot is shown in **Figure 1-2**.

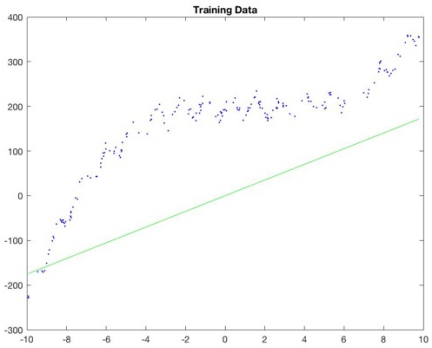

Figure 1-1 Plot for *leastSquares*

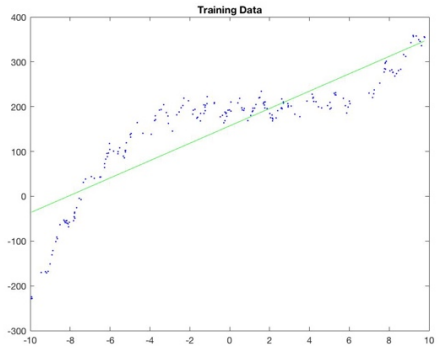

Figure 1-2 Plot for *leastSquaresBias*

Figure 1

2.2 Polynomial Basis

Although we have dramatically decreased the training and test errors by adding one bias variable, they are still too large for a reliable model. Actually, the scatter diagram does not look like a straight line as we presumed in **2.1**. Therefore, we try to apply polynomial regression in this section.

The function *leastSquaresBasis(X, y, deg)* written by me takes an input data vector X, the response data vector y and the polynomial order deg. The model/predict format of *leastSquaresBasis* is the same as *leastSquares* and *leastSquaresBias*. In detail, this function computes by the equation

$$\mathrm{W} = \left({\mathrm{X}_{poly}}^{T}\mathrm{X}_{poly}\right)^{-1}{\mathrm{X}_{poly}}^{T}\mathrm{y}$$

or

$$\mathrm{W} = \left({\mathrm{X}_{poly}}^{T}\mathrm{X}_{poly}\right)^{\dagger}{\mathrm{X}_{poly}}^{T}\mathrm{y},$$

where

$$\mathrm{X}_{poly} = \begin{bmatrix} 1 & x_1 & {x_1}^2 & \cdots & {x_1}^{deg} \\ 1 & x_2 & {x_2}^2 & \cdots & {x_2}^{deg} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & {x_n}^2 & \cdots & {x_n}^{deg} \end{bmatrix}.$$

A special case that *leastSquaresBasis(X, y, 1)* is identical to *leastSquaresBias(X, y)*.

Now, setting $\deg = 3$ as an example. Let

$$\text{model} = leastSquaresBasis(X, y, 3),$$

then

$$\text{model.w} = [198.43 \quad 0.09 \quad -1.29 \quad 0.31]^T.$$

That is to say, if we presume a polynomial regression of $\text{degree} = 3$, then the model is

$$\hat{y} = 198.43 + 0.09x - 1.29x^2 + 0.31x^3.$$

After figuring out these above, write a script *LeastSquaresFitting.m* to compute the training and test errors for $\deg = 0$ through $\deg = 10$. The outcomes are in **Table 2** below.

Error\Deg	0	1	2	3	4	5	6	7	8	9	10
Training	15480.52	3551.35	2167.99	252.05	251.46	251.14	248.58	247.01	241.31	235.76	235.07
Test	14390.76	3393.87	2480.73	242.80	242.13	239.54	246.01	242.89	245.97	249.30	256.30

Table 2 Training and test errors for deg=0 through deg=10

In conclusion, the regression model improves substantially when deg increases from 0 to 3. Afterwards, the training and test errors approximately keep steady (decrease rather slow precisely), which implies an overfitting. In this way, we'd like to choose $\deg = 3$ as the best model of polynomial regression.

3 Regularization

3.1 Load and Standardize the Data

The original data is stored in *prostate.data.txt*, so first use the MATLAB built-in function *importdata* to load it. Second, obtain a vector R of random indices with the same number of patients via the built-in function *randperm*. Randomly shuffle the orders of patients in the table by R, and then choose the first 50 patients as the training data. The remaining patients will be the test data.

In order to standardize the data, which is necessary to place all the 8 attributes on the same scale, we will set both variables (i.e. X_train and y_train) to have zero mean and standardize the input variables (i.e. X_train) to have unit variance.

To be concrete, for all i and j, let

$$\mathrm{X\_train(i,:)} = (\mathrm{X\_train(i,:)} - \overline{\mathrm{X\_train}}) \,./\, \mathrm{X\_train\_sigma}$$
$$\mathrm{Y\_train(i)} = \mathrm{Y\_train(i)} - \overline{\mathrm{Y\_train}}$$
$$\mathrm{X\_test(j,:)} = (\mathrm{X\_test(j,:)} - \overline{\mathrm{X\_train}}) \,./\, \mathrm{X\_train\_sigma}$$
$$\mathrm{Y\_test(j)} = \mathrm{Y\_test(j)} - \overline{\mathrm{Y\_train}}$$

where $\overline{\mathrm{X\_train}}$ is a vector containing mean values of X_train 's each column, and X_train_sigma is a vector containing standard deviations of X_train 's each column. $\overline{\mathrm{Y\_train}}$ the similar.

To accomplish the above steps, I write a script *StandardizeData.m*. All the obtained data is saved into *stdData.mat* for convenience and in case of data loss.

There's nothing complicated, but please note that X_test and Y_test have already been standardized.

3.2 Ridge Regression

3.2.1 Theta

With standardized data, we are going to construct a model using ridge regression. The ridge method is a regularized version of least squares, with objective function:

$$\min_{\theta \in \mathbb{R}^{\mathrm{d}}} \|y - \mathrm{X}\theta\|_2^2 + \delta^2 \|\theta\|_2^2,$$

where δ^2 is a scalar called regularizer.

In *Exerc_Chap2* of this course, we have already learned and studied this function. Differentiating the above cost function can yield the normal equation

$$(\mathrm{X}^T\mathrm{X} + \delta^2\mathrm{I}_d)\theta = \mathrm{X}^T y.$$

Thus,

$$\theta = (\mathrm{X}^T\mathrm{X} + \delta^2\mathrm{I}_d)^{-1}\mathrm{X}^T y,$$

or

$$\theta = (\mathrm{X}^T\mathrm{X} + \delta^2\mathrm{I}_d)^{\dagger}\mathrm{X}^T y,$$

again, the Moore-Penrose pseudoinverse.

And that is all about how we compute θ. The script *ComputeTheta.m* written by me do this process for a range of regularizers (δ^2) from 0.01 to 10000, and then plot the regularization path (values of each θ in the y-axis against δ^2 in the x-axis). The plot is displayed below (**Figure 3**).

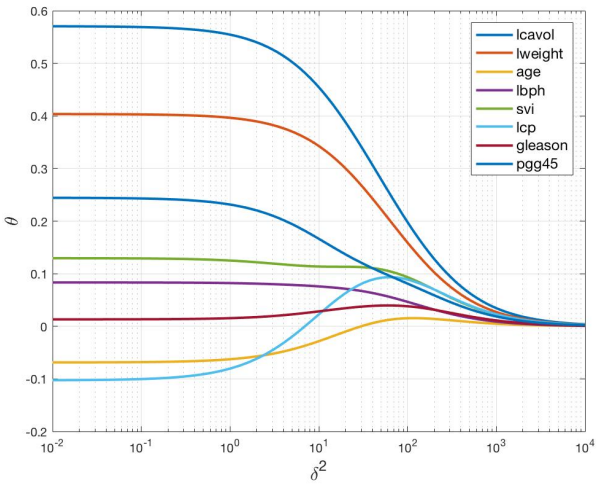

Figure 3 Regularization Path

You may well gain a diagram totally different from **Figure 3**. That is a normal phenomenon, because we have applied a random function *randperm* to the script *StandardizeData.m*. The same situation occurs to **Figure 4** and **Figure 5**, and I would not explain again.

3.2.2 Training and Test Errors

The error for this problem is evaluated by

$$\frac{\|y - \mathrm{X}\theta\|_2}{\|y\|_2}.$$

My script *ComputeErrors.m* computes the training and test errors for each θ obtained from *ComputeTheta.m*. Since I have chosen a large number of (100 precisely) regularizers (δ^2) to compute θ, it is unpractical to draw a table displaying them all. So I plot them in **Figure 4**, which is more explicit.

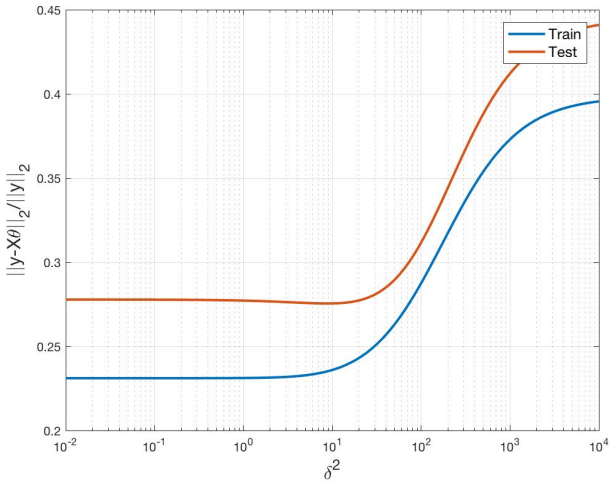

Figure 4 Training and Test Errors

3.3 K-fold Cross-validation

Furthermore, we will use K-fold cross-validation to polish our model. Let's make it in two steps.

Firstly, I write a function *cvInd(n,K)* to randomly divide $\{1, 2, 3, \cdots, n\}$ into K equal (or approximately equal when remainder exists) groups. Here, K is an integer variable, and defaults to 5 when omitted.

Then, my function *cvErrors(X,y,d2,Indices)* computes and returns the training and test errors of data (X, *y*) with regularizer *d2*, based on K-fold cross-validation of ridge regression. In detail, Indices is the matrix generated by *cvInd(n,K)*. Parameters input, function *cvErrors* could identify K automatically, then take one group out of K as the test set, the other (K-1) groups as the training set, and compute the training and test error. It carries out this process K times in total, and finally returns the mean values of K's training and test errors.

Since there are only 97 patients' data, K cannot be set too large. I set $K = 5$ in practice. My script *PlotCVErrors.m* uses the two functions discussed above, and plots the final outcomes in **Figure 5**.

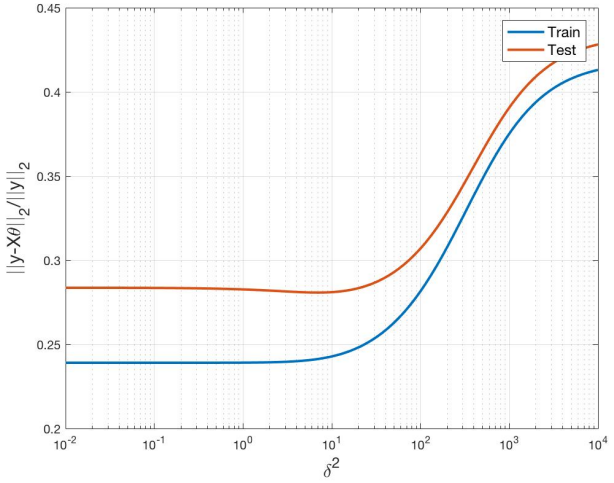

Figure 5 Training and Test Errors of 5-fold Cross-Validation

When the test error takes minimum, its corresponding δ^2 is exactly what we want. In my experiment, that is $\delta^2 = 7.05$. The process of finding it is also included in *PlotCVErrors.m*.

4 Summaries and Reflections

By doing this project, I have learned some points and techniques. Here I will take them down only as a record for myself.

1. When constructing a polynomial regression model, the best fitting degree could be found by computing and comparing the training and test errors for a range of degrees.
2. If the data has been standardized, and meanwhile the error is evaluated by relative value, then what has been subtracted must be added back to the data in denominator.
3. K-fold cross-validation fully uses all the data, which performs handsomely when the amount of data is not large enough.

Appendix

MATLAB Codes (Including Comments)

Contents

All the programming codes involved in this project are displayed below in order of practical use.

1 Codes for Problem 1

1.1 *leastSquaresBias.m*

```
function [model] = leastSquaresBias(X, y)
%LEASTSQUARESBIAS(X,y) Solves linear least-squares problem with a bias
% variable (assumes X'*X is invertible), where model.w(1) is the bias
% variable and model.w(2) the slope variable.
%
% Yuanbo Han, Oct. 7, 2017

X_1 = [ ones(size(X,1),1), X ];
w = (X_1'*X_1)\X_1'*y;

model.w = w;
model.predict = @predict;
end

function [yhat] = predict(model, Xhat)
w = model.w;
Xhat_1 = [ ones(size(Xhat,1),1), Xhat ];
yhat = Xhat_1*w;
end
```

1.2 *leastSquaresBasis.m*

```
function [model] = leastSquaresBasis(X,y,deg)
%LEASTSQUARESBASIS(X,y,deg) Solves least-squares problem with polynomial
% order deg (assumes X'*X is invertible), where model.w is the deg-by-1
% vector of coefficients.
%
% Yuanbo Han, Oct. 7, 2017

X_poly = ones(size(X,1), deg+1);
for i = 1:deg
   X_poly(:,i+1) = X.^i;
end

w = (X_poly'*X_poly)\X_poly'*y;

model.degree = deg;
model.w = w;
model.predict = @predict;
end
```

```
function [yhat] = predict(model,Xhat)
w = model.w;
deg = model.degree;

Xhat_poly = ones(size(Xhat,1), deg+1);
for i = 1:deg
    Xhat_poly(:,i+1) = Xhat.^i;
end

yhat = Xhat_poly*w;
end
```

1.3 *LeastSquaresFitting.m*

```
% Edited by Yuanbo Han, Oct. 7, 2017

% Clear variables and close figures
clear all;
close all;

% Load data
load basisData.mat; % Loads X and y
[n,d] = size(X);

% Fit least-squares model for deg = 0 through deg = 10
for deg = 0:10
    fprintf('deg = %d\n', deg);
    model = leastSquaresBasis(X,y,deg);

    % Compute and report the training error
    yhat = model.predict(model,X);
    trainError = sum((yhat - y).^2)/n;
    fprintf('Training error = %.2f\n',trainError);

    % Compute and report the test error
    t = size(Xtest,1);
    yhat = model.predict(model,Xtest);
    testError = sum((yhat - ytest).^2)/t;
    fprintf('Test error = %.2f\n',testError);

    fprintf('\n');
```

```
end
```

2 Codes for Problem 2

2.1 *StandardizeData.m*

```
% Standardize the original data and save them into 'stdData.mat'.
% Edited by Yuanbo Han, Oct. 7, 2017

% Clear variables and close figures
clear all;
close all;

% Get the original data from 'prostate.data.txt'
file = importdata('prostate.data.txt');
X = file.data(:,1:8);
y = file.data(:,9);
T = file.colheaders;
n = size(X,1);

% Shuffle to obtain train and test data
R = randperm(n);
X_train = X(R(1:50),:);
X_test = X(R(51:n),:);
y_train = y(R(1:50));
y_test = y(R(51:n));

clear file R;

% Standardize the training data
X_train_bar = mean(X_train);
X_train_sigma = std(X_train,1);
for i = 1:50
    X_train(i,:) = (X_train(i,:) - X_train_bar) ./ X_train_sigma;
end
y_train_bar = mean(y_train);
y_train = y_train - y_train_bar;

% Standardize the test data
for i = 1:(n-50)
    X_test(i,:) = (X_test(i,:) - X_train_bar) ./ X_train_sigma;
end
y_test = y_test - y_train_bar;
```

```
clear i;
save stdData;
```

2.2 *Ridge.m*

```
function [theta] = Ridge(X,y,d2)
%RIDGE(X,y,d2) returns a vector theta of coefficient estimates for a
% multilinear ridge regression of the responses in y on the predictors in
% X, where d2 is the regularization parameter. Here, X and y must have been
% standardized.
%
% Yuanbo Han, Oct. 7, 2017

theta = ( X'*X + d2.*eye(size(X,2)) )\X'*y ;
end
```

2.3 *ComputeTheta.m*

```
% Compute theta for a range of d2 and plot them.
% Edited by Yuanbo Han, Oct. 8, 2017

% Load the standardized data in 'stdData.mat'
load stdData.mat;

% Compute the ridge regression solutions (theta) for a range of
% regularizers (d2)
d2 = logspace(-2,4,100);
theta = zeros(8,length(d2));
for i = 1:length(d2)
    theta(:,i) = Ridge(X_train,y_train,d2(i));
end

% Plot the regularization path (the values of each theta in the y-axis
% against d2 in the x-axis)
figure;
for i = 1:8
    semilogx( d2, theta(i,:), 'LineWidth', 2 );
    hold on;
end
xlabel('\delta^2','FontSize',15);
ylabel('\theta','FontSize',15);
```

```
legend(T(1:8),'Location','NorthEast','FontSize',12);
grid on;

clear i;
save stdData.mat d2 theta -append;
```

2.4 *ComputeErrors.m*

```
% Compute and display the training and test errors for each computed value
% of theta.
% Edited by Yuanbo Han, Oct. 8, 2017

% Load the standardized data from 'stdData.mat'
load stdData;

% Compute the training and test errors for each computed value of theta
trainError = zeros(length(d2),1);
testError = zeros(length(d2),1);
for i = 1:length(d2)
   trainError(i) = sqrt( (y_train - X_train * theta(:,i))' * (y_train - X_train *
theta(:,i) ) / ((y_train+y_train_bar)'*(y_train+y_train_bar)) );
   testError(i) = sqrt( (y_test - X_test * theta(:,i))' * (y_test - X_test * theta(:,i) )
/ ((y_test+y_train_bar)'*(y_test+y_train_bar)) );
end

fprintf('trainError =\n\n');
disp(trainError);
fprintf('testError =\n\n');
disp(testError);

% Plot the training and test errors as a function of d2
figure;
semilogx(d2,trainError,d2,testError,'LineWidth',2);
xlabel('\delta^2','FontSize',15);
ylabel('\mid\midy-X\theta\mid\mid_2/\mid\midy\mid\mid_2','FontSize',15);
legend({'Train','Test'},'Location','NorthEast','FontSize',12);
grid on;

clear i;
```

2.5 *cvInd.m*

```
function [Indices] = cvInd(n,K)
```

```
%CVIND(K) returns a K-by-(n/K) matrix (take floor(n/K) instead when n/K is
% not an integer) of randomly generated indices for a K-fold
% cross-validation of n-time observations, where row vectors are K disjoint
% subsets of [1:n] with the same length. K defaults to 5 when omitted.
%
% Yuanbo Han, Oct. 8, 2017

if ~exist('K','var')
   K = 5;
end

step = floor(n/K);
R_ind = randperm(n);
Indices = zeros(K,step);

for i = 1:K
   strtindex = (i-1) * step + 1;
   stpindex = strtindex + step - 1;
   Indices(i,:) = R_ind(strtindex:stpindex);
end
end
```

2.6 *cvErrors.m*

```
function [trainError,testError] = cvErrors(X,y,d2,Indices)
%CVERRORS returns the trainError and the testError of a K-fold
% cross-validation on ridge regression, whose partition is given by Indices
% (an index matrix with K rows). X is the observation matrix, y the
% response vector and d2 the ridge parameter delta^2.
%
% Yuanbo Han, Oct. 8 ,2017

K = size(Indices,1);
trainErrorArray = zeros(K,1);
testErrorArray = zeros(K,1);

for i = 1:K
   X_test = X(Indices(i,:),:);
   y_test = y(Indices(i,:));
   temp = X;
   temp(Indices(i,:),:) = [];
   X_train = temp;
```

```
    temp = y;
    temp(Indices(i,:)) = [];
    y_train = temp;

    % Standardize the training data
    X_train_bar = mean(X_train);
    X_train_sigma = std(X_train,1);
    for j = 1:size(X_train,1)
        X_train(j,:) = (X_train(j,:) - X_train_bar) ./ X_train_sigma;
    end
    y_train_bar = mean(y_train);
    y_train = y_train - y_train_bar;

    % Standardize the test data
    for j = 1:size(X_test,1)
        X_test(j,:) = (X_test(j,:) - X_train_bar) ./ X_train_sigma;
    end
    y_test = y_test - y_train_bar;

    % Compute theta and errors
    theta = Ridge(X_train,y_train,d2);
    trainErrorArray(i) = sqrt( (y_train - X_train * theta)' * (y_train - X_train * theta)
/ ((y_train+y_train_bar)'*(y_train+y_train_bar)) );
    testErrorArray(i) = sqrt( (y_test - X_test * theta)' * (y_test - X_test * theta) /
((y_test+y_train_bar)'*(y_test+y_train_bar)) );
end

trainError = mean(trainErrorArray);
testError = mean(testErrorArray);
end
```

2.7 *PlotCVErrors.m*

```
% Compute the training and test errors for each d2 using cross-validation,
% and then plot the errors as a function of d2.
% Edited by Yuanbo Han, Oct. 8, 2017

% Load the standardized data from 'stdData.mat'
load stdData;

% Compute the training and test errors for each d2 using cross-validation
l = length(d2);
```

```
trainError = zeros(l,1);
testError = zeros(l,1);

% Below is a K-fold cross-validation.
K = 5;  % K can be changed to any integer in [2:n].

Indices = cvInd(n,K);
for i = 1:l
   [trainError(i),testError(i)] = cvErrors(X,y,d2(i),Indices);
end

% Plot the training and test errors as a function of d2
figure;
semilogx(d2,trainError,d2,testError,'LineWidth',2);
xlabel('\delta^2','FontSize',15);
ylabel('\mid\midy-X\theta\mid\mid_2/\mid\midy\mid\mid_2','FontSize',15);
legend({'Train','Test'},'Location','NorthEast','FontSize',12);
grid on;

% Find the d2 when testError takes minimum
[testError_min, index] = min(testError);
fprintf('When testError takes minimum, d2 = %.2f.\n', d2(index));

clear i l K;
```