# MATLAB Companion Script for *Machine Learning* ex5 (Optional)

## Introduction

Coursera's *Machine Learning* was designed to provide you with a greater understanding of machine learning algorithms- what they are, how they work, and where to apply them. You are also shown techniques to improve their performance and to address common issues. As is mentioned in the course, there are many tools available that allow you to use machine learning algorithms *without* having to implement them yourself. This Live Script was created by MathWorks to help *Machine Learning* students explore the data analysis and machine learning tools available in MATLAB.

## FAQ

### Who is this intended for?

- This script is intended for students using MATLAB Online who have completed ex5 and want to learn more about the corresponding machine learning tools in MATLAB.

### How do I use this script?

- In the sections that follow, read the information provided about the data analysis and machine learning tools in MATLAB, then run the code in each section and examine the results. You may also be presented with instructions for using a MATLAB machine learning app. This script should be located in the ex5 folder which should be set as your Current Folder in MATLAB Online.

### Can I use the tools in this companion script to complete the programming exercises?

- No. Most algorithm steps implemented in the programming exercises are handled automatically by MATLAB machine learning functions. Additionally, the final results will be similar, but not identical, to those in the programming exercises due to differences in implementation, parameter settings, and randomization.

### Where can I obtain help with this script or report issues?

- As this script is not part of the original course materials, please direct any questions, comments, or issues to the *MATLAB Help* discussion forum.

# Regularized Linear Regression and Cross-Validation

In this Live Script, we will use functions and apps from the Statistics and Machine Learning Toolbox to partition data into training and validation sets and determine optimal regularization strengths using cross-validation.

## Files needed for this script

- `ex5data1.mat` - Waterflow dataset

**Table of Contents**

# Data Partitioning, High Bias and High Variance Models

Many functions used to train models in the Statistics and Machine Learning Toolbox are equipped to partition a dataset into training and cross-validation sets automatically. However, creating a fixed partition is often helpful for reproducibility of results, or when comparing the performance of several different algorithms. These partitions can also be useful when training models using functions for building and training simpler models like `fitlm` and `fitglm` which do not partition data automatically or provide options for validation. In this section we will use the `cvpartition` function to partition the water flow data from ex5.

## Load the data

Recall that the data contained in `ex5data1.mat` is divided into three sets:

- A **training** set to train the model
- A **cross-validation** set to determine the regularization parameter
- A **test** set to evaluate performance.

2

In MATLAB, there is no need to store training and validation sets in separate variables. Run the code below to combine the training, test, and validation sets into a single `table` variable, `data`, and clear the original variables.

```
clear;
load ex5data1.mat;
LevelChange = [X;Xtest;Xval];
Outflow = [y;ytest;yval];
data = table(LevelChange,Outflow);
clearvars -except data;
```

## Partition the dataset

Next we will use the `cvpartition` function to partition the water flow data into a training set with 12 examples (for direct comparison with your results from ex5) and validation set with the remaining data. The result will be a `cvpartition` variable that contains the partition information and the indicies of corresponding to the two sets. This variable can then be passed to MATLAB machine learning functions along with the original dataset for model training and validation. Alternatively, we can also extract a binary vector corresponding to the training and validation examples from the partition variable using the `training` and `test` functions, as demonstrated below, for use with functions like `fitlm` and `fitglm`.

```
m = height(data); % height() returns the number of rows in a table
c = cvpartition(m,'HoldOut',m-12) % Choose 12 training examples
```

```
c =
Hold-out cross validation partition
   NumObservations: 54
       NumTestSets: 1
         TrainSize: 12
          TestSize: 42
```

```
tinds = training(c) % Training indices
```

```
tinds = 54×1 logical array
   0
   0
   0
   0
   0
   0
   0
   0
   0
   0
    :
    :
    :
```

**Note**: In the terminology of the Statistics and Machine Learning Toolbox, 'test' sets correspond to the 'validation' sets described in class. A 'test' set as described in the course - data not used in training OR validation - would still need to be stored in a separate `table` or matrix.

## Fit a high bias and high variance model to the training data

As in ex5, we will fit a linear model as an example of a high bias model, followed by an 8th degree polynomial model as a high variance model example. The code below uses `fitlm` to train the models using then plots the results. If you are interested, you can re-run both this section and the previous one to visualize the effect that the small training set size and random partition selections have on high bias/variance models.

```matlab
% Train the models
highBiasMdl = fitlm(data(tinds,[1,2]));
HighVarMdl = fitlm(data(tinds,[1 2]),'poly8');
% Plot the model and training data
xplot = linspace(min(data.LevelChange(tinds)),max(data.LevelChange(tinds)))';
y1 = predict(highBiasMdl,xplot);
y2 = predict(HighVarMdl,xplot);
figure; hold on;
plot(data.LevelChange(tinds),data.Outflow(tinds),'rx','MarkerSize',10,'LineWidth',2);
plot(xplot,y1); plot(xplot,y2); hold off;
xlabel('Change in water level (x)');
ylabel('Water flowing out of the dam (y)');
legend({'Training Data','Linear Model','Polynomial Model'})
```
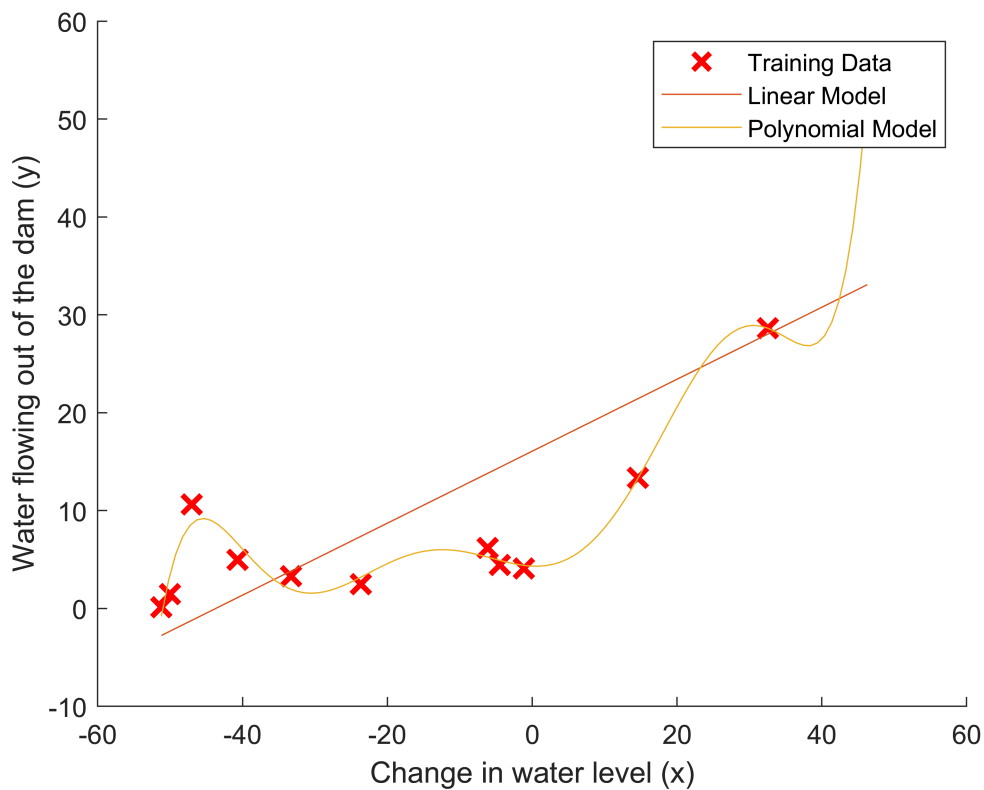


# Regularized Linear Regression using `fitrlinear`

As discussed in ex5, the linear model is too simple to accurately reflect the complexity of the water flow data, while polynomial model tends to overfit the data and changes dramatically depending on the partition. In ex5 you found that the performance of the polynomial model can be improved on new data by adding regularization during training. Cross validation was then used to determine the 'best' value of the regularization parameter,

$\lambda$. In this section, we use the `fitrlinear` function to fit a regularized polynomial model to the data. Like `fitclinear` (used in the companion script for ex2), `fitrlinear` is designed to train models with a large number of variables and it also includes options for adding regularization.

## Load and normalize the data

The code below will reload the data and create a (normalized) polynomial feature matrix X by combining the training and validation sets as before. As with `fitclinear`, `fitrlinear` requires feature and response data as numeric arrays (i.e. not in a `table`).

```
load ex5data1.mat;
LevelChange = [X;Xtest;Xval];
Outflow = [y;ytest;yval];
clearvars -except LevelChange Outflow;
mu = mean(LevelChange);
sd = std(LevelChange);
LevelChange = ((LevelChange-mu)/sd).^(1:8);
```

## Partition the data

In this section we will let `fitrlinear` partition the data automatically for us. We also provide code in this section and the next one to create the corresponding model using a `cvpartition` variable as comments.  Run the code below to select the proportion of data to hold out during training for validation purposes.

```
holdout = 0.35;
% c = cvpartiton(length(LevelChange),'HoldOut',holdout)
```

## Fit linear regression models using multiple $\lambda$ values

In the companion script for ex3, we trained a regularized model by passing a single regularization value `lambda` to `fitclinear` using the `'Lambda'` option. We can also use this option with `fitrlinear`. However, by passing *a vector* of $\lambda$ values to these functions, a separate model will automatically be trained for each $\lambda$ value. Run the code below to create a vector of $\lambda$ values and train the regression models for each $\lambda$. When using multiple regularization values with `fitrlinear`, the trained models will be returned in a `RegressionPartitionedLinear` model variable instead of a `RegressionLinear` variable which is discussed in the next section.

```
lambda = [0 0.00001 0.0001 0.001 0.01 0.03 0.1 0.3 1];
partMdl = fitrlinear(LevelChange,Outflow,'Learner','leastsquares','Regularization','Ridge','Lam
```

```
partMdl =
  classreg.learning.partition.RegressionPartitionedLinear
    CrossValidatedModel: 'Linear'
           ResponseName: 'Y'
        NumObservations: 54
                  KFold: 1
              Partition: [1×1 cvpartition]
      ResponseTransform: 'none'


  Properties, Methods
```

```
% partMdl = fitrlinear(LevelChange,Outflow,'Learner','leastsquares','Regularization','Ridge','L
```

## Working with partitioned model variables

The individual regression models are contained in the `Trained` property of a `RegressionPartitionedLinear` variable. In the case of hold-out cross validation, the `Trained` property is a cell array which contains a single `RegressionLinear` (or `ClassificationLinear` when using `fitclinear`) model variable. This variable contains the trained models for each regularization value $\lambda$. The model partition that was used during training and validation is found in the `Partition` property.

The model coefficients ($\theta$) are found in the $p \times l$ matrix `Beta` and the $1 \times l$ vector `bias`, where $p$ is the number of model coefficients (8 in the current example) and $l$ is the number of $\lambda$ values (9). The columns of `Beta` therefore correspond to coefficients for a particular model. The model coefficients and the corresponding lambda values are displayed below for comparison. Run the code below to extract the trained model variable, the partition variable, and the model coefficients. Do the increasing regularization values have the expected effect on the size of the coefficients and bias?

```
polyRegMdl = partMdl.Trained{1}
```

```
polyRegMdl =
  RegressionLinear
          ResponseName: 'Y'
     ResponseTransform: 'none'
                  Beta: [8×9 double]
                  Bias: [6.5733 6.5733 6.5733 6.5733 6.5733 6.7712 7.2735 7.7711 8.4469]
                Lambda: [0 1.0000e-05 1.0000e-04 1.0000e-03 0.0100 0.0300 0.1000 0.3000 1]
               Learner: 'leastsquares'


  Properties, Methods
```

```
c = partMdl.Partition
```

```
c =
Hold-out cross validation partition
   NumObservations: 54
       NumTestSets: 1
         TrainSize: 36
          TestSize: 18
```

```
[polyRegMdl.Bias; polyRegMdl.Beta]
```

```
ans = 9×9
    6.5733    6.5733    6.5733    6.5733    6.5733    6.7712    7.2735    7.7711 ···
   10.0842   10.0841   10.0841   10.0834   10.0818    6.4593    4.7198    3.2213
    0.4492    0.4492    0.4492    0.4491    0.4491    3.2745    2.0172    1.2232
   -2.9277   -2.9277   -2.9277   -2.9275   -2.9270    1.6966    2.1396    2.1304
   10.2803   10.2802   10.2802   10.2795   10.2780    2.0711    1.6120    1.2583
    1.4705    1.4705    1.4705    1.4704    1.4701    0.0516    0.9029    1.3817
   -6.1213   -6.1212   -6.1212   -6.1209   -6.1196   -0.4768    0.5482    0.8838
    0.1758    0.1758    0.1758    0.1758    0.1757    0.2250   -0.0688   -0.1477
    1.2817    1.2817    1.2817    1.2814    1.2820    0.1684   -0.0958   -0.1308
```

```
lambda
```

```
lambda = 1×9
```

```
     0     0.0000     0.0001     0.0010     0.0100     0.0300     0.1000     0.3000 ···
```

## Evaluate model performance and select models using `kfoldLoss` and `selectModels`

Instead of using learning curves to detect high bias or variance issues, followed by validation curves to determine the optimal regularization parameter value, we will use the `kfoldLoss`* function to compute the cost of all models in the partitioned model variable using the validation set. We then extract the individual model with the lowest cost using the `selectModels` function. The result will be a `RegressionLinear` model variable corresponding to the best model as found by cross-validation.

```
valCost = kfoldLoss(partMdl)
```

```
valCost = 1×9
   15.4675    15.4665    15.4693    15.4366    15.5159    11.2346    10.6399    13.8328 ···
```

```
[~,bestidx] = min(valCost)
```

```
bestidx = 7
```

```
fprintf('The best value of lambda found by cross-validation is: %g', lambda(bestidx))
```

```
The best value of lambda found by cross-validation is: 0.1
```

```
bestRegMdl = selectModels(polyRegMdl,bestidx)
```

```
bestRegMdl =
  RegressionLinear
        ResponseName: 'Y'
   ResponseTransform: 'none'
                Beta: [8×1 double]
                Bias: 7.2735
              Lambda: 0.1000
             Learner: 'leastsquares'


  Properties, Methods
```

*K-fold cross validation is discussed at the end of this Live Script. K-fold validation with 1 fold is equivalent to hold-out validation.

## Predict the response for all models and visualize the results

Run the code below to plot the training data, validation data, and the models for each $\lambda$. Note, when using the `predict` function along with a model variable containing multiple models (coefficient sets), `predict` will return a response *matrix*, where the $i$th column of the matrix corresponds to the prediction using the $i$th model (trained with the $i$th regularization value).

```
% Extract the training and validation data
xtrain = LevelChange(training(c),1);
ytrain = Outflow(training(c));
xval = LevelChange(test(c),1);
yval = Outflow(test(c));
% Plot the data and models
```

```matlab
xplot = linspace(min(LevelChange(:,1)),max(LevelChange(:,1)))';
xplot = xplot.^(1:8);
Y = predict(polyRegMdl,xplot);
figure; hold on;
plot(xtrain*sd+mu,ytrain,'go','MarkerSize',10);
plot(xval*sd+mu,yval,'LineStyle','none','Marker','square','MarkerSize',10);
plt = plot(xplot(:,1)*sd+mu,Y,'--'); hold off;
set(plt,{'Color'},num2cell(jet(length(lambda)),2));
set(plt(bestidx),'LineStyle','-','LineWidth',2);
ylim([min(Outflow) max(Outflow)]);
xlabel('Change in water level (x)');
ylabel('Water flowing out of the dam (y)');
legend(['Training','Validation',cellstr("lambda = "+string(lambda))],'Location','NorthWest')
title(sprintf('Best lambda = %g',lambda(bestidx)));
```
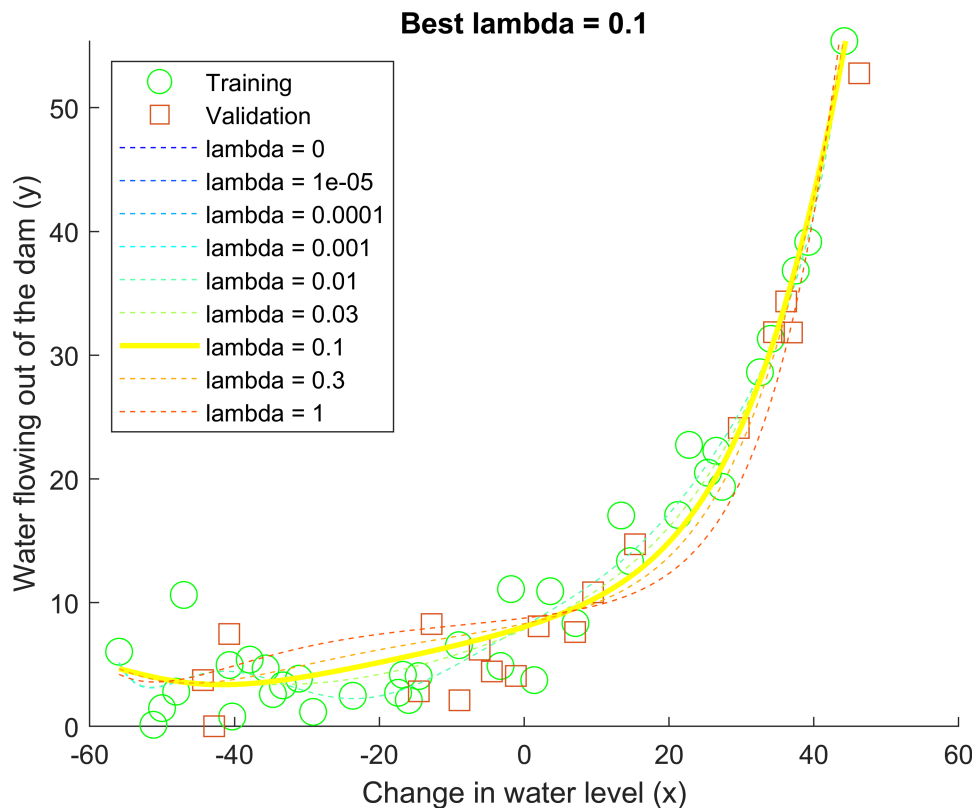


## Hyperparameter Optimization

Model parameters, such as the regularization strength or optimization solver settings, are commonly referred to as *hyperparameters*. It is often the case that we do not want to explore the effect of hyperparameters on a model; rather we are only interested in obtaining *the best* model and/or hyperparameter value as judged by cross-validation. In these cases we can utilize the *hyperparameter optimization* options in `fitrlinear/fitclinear`, which then automatically return the best model found after training over a set or range of hyperparameter settings. In this section we use hyperparameter optimization with `fitrlinear` to automatically obtain the best cross-validation strength and the corresponding model.

## Specify the cross-validation method

To use hyperparameter optimization, the fist step is to create a `structure` variable that contains information about the cross-validation method that will be used to judge model performance. We can then provide a list of parameters to optimize using the `'OptimizeHyperparameters'` option when training the model and providing a cell array of parameter names to be optimized. If desired, we can also adjust the permissible range or values of hyperparameters to be searched during optimization from their default values by obtaining the corresponding `optimizableVariable` structure for the given machine learning function, method, and dataset by using the `hyperparameters` function. Below we obtain the variable for `fitrlinear` with least-squares regression and the current dataset, then select only the first optimizable parameter, $\lambda$.

Run the code below to create an options structure for hold-out cross-validation using the same percentage as in the previous section, then create a hyperparameter options variable with a custom range of $\lambda$ values from `1e-6` to `1` to search.

```
opts = struct('HoldOut',holdout);
optimizableParams = hyperparameters('fitrlinear',LevelChange,Outflow)
```

optimizableParams = 3×1 optimizableVariable

|   | 1 |
|---|---|
| 1 | 1×1 optim... |
| 2 | 1×1 optim... |
| 3 | 1×1 optim... |

```
optimizableParams = optimizableParams(1)
```

```
optimizableParams =
  optimizableVariable with properties:

         Name: 'Lambda'
        Range: [1.8519e-07 1.8519e+03]
         Type: 'real'
    Transform: 'log'
     Optimize: 1
```

```
optimizableParams.Range = [1e-6, 1]
```

```
optimizableParams =
  optimizableVariable with properties:

         Name: 'Lambda'
        Range: [1.0000e-06 1]
         Type: 'real'
    Transform: 'log'
     Optimize: 1
```

## Automatically select $\lambda$ using hyperparameter optimization

By including the validation options structure and the hyperparameter options variable created in the previous section and naming the hyperparameter to be optimized in the call to `fitrlinear` or `fitclinear`, these functions will automatically search through the given parameter values and return a model variable

corresponding to the best $\lambda$ found as judged by cross validation. Since many models (30 by default) are trained during the search, this will take considerably longer to run than when training a single model. During training, the progress is displayed. If desired, the fit information for the final model, along with the hyperparameter optimization information, can also be obtained by providing additional output variables. See the documentation for details.

Run the code below to call `fitrlinear` with hyperparameter optimization for the regularization strength. The final model variable will be displayed, along with the best $\lambda$ value found (you may have to scroll past the progress plots in the output). The best model found in the previous section using a fixed set of parameters and manual model selection will be plotted alongside the optimized model computed below for comparison. Which model performs better?

```
optimRegMdl = fitrlinear(LevelChange,Outflow,...
                         'Learner','leastsquares',...
                         'Regularization','Ridge',...
                         'OptimizeHyperparameters',optimizableParams,...
                         'HyperparameterOptimizationOptions',opts)
```

| Iter | Eval result | Objective: log(1+loss) | Objective runtime | BestSoFar (observed) | BestSoFar (estim.) | Lambda |
|------|------|------|------|------|------|------|
| 1 | Best | 2.96 | 0.17836 | 2.96 | 2.96 | 0.00010287 |
| 2 | Best | 2.9574 | 0.16835 | 2.9574 | 2.9587 | 4.0796e-06 |
| 3 | Accept | 3.4899 | 0.047095 | 2.9574 | 2.9574 | 0.68052 |
| 4 | Best | 2.9468 | 0.030375 | 2.9468 | 2.9468 | 0.011317 |
| 5 | Best | 2.9086 | 0.038811 | 2.9086 | 2.9087 | 0.028297 |
| 6 | Accept | 2.9799 | 0.031324 | 2.9086 | 2.9087 | 0.0021431 |
| 7 | Accept | 2.9695 | 0.02633 | 2.9086 | 2.9087 | 0.00060402 |
| 8 | Accept | 2.9574 | 0.027336 | 2.9086 | 2.9087 | 1.0007e-06 |
| 9 | Accept | 2.9579 | 0.026663 | 2.9086 | 2.9088 | 2.0806e-05 |
| 10 | Best | 2.9019 | 0.029329 | 2.9019 | 2.9039 | 0.037109 |
| 11 | Best | 2.9014 | 0.030588 | 2.9014 | 2.9019 | 0.038557 |
| 12 | Best | 2.9009 | 0.031766 | 2.9009 | 2.9013 | 0.042661 |
| 13 | Accept | 2.9009 | 0.025885 | 2.9009 | 2.9013 | 0.042435 |
| 14 | Accept | 2.9009 | 0.026276 | 2.9009 | 2.901 | 0.042287 |
| 15 | Accept | 2.9009 | 0.026326 | 2.9009 | 2.901 | 0.042126 |
| 16 | Accept | 3.0123 | 0.024976 | 2.9009 | 2.9008 | 0.15941 |
| 17 | Accept | 2.9014 | 0.024706 | 2.9009 | 2.9009 | 0.047354 |
| 18 | Accept | 2.9011 | 0.024552 | 2.9009 | 2.9009 | 0.045727 |
| 19 | Accept | 2.9574 | 0.025743 | 2.9009 | 2.9009 | 1.552e-06 |
| 20 | Accept | 2.9642 | 0.026402 | 2.9009 | 2.9009 | 0.0002966 |

| Iter | Eval result | Objective: log(1+loss) | Objective runtime | BestSoFar (observed) | BestSoFar (estim.) | Lambda |
|------|------|------|------|------|------|------|
| 21 | Accept | 2.9576 | 0.026389 | 2.9009 | 2.9009 | 1.1255e-05 |
| 22 | Accept | 2.9742 | 0.037678 | 2.9009 | 2.9009 | 0.0049742 |
| 23 | Accept | 2.9226 | 0.025015 | 2.9009 | 2.901 | 0.07882 |
| 24 | Accept | 2.9584 | 0.027277 | 2.9009 | 2.901 | 4.2236e-05 |
| 25 | Accept | 2.9754 | 0.025773 | 2.9009 | 2.901 | 0.001128 |
| 26 | Accept | 2.9228 | 0.035166 | 2.9009 | 2.901 | 0.018717 |
| 27 | Accept | 2.9616 | 0.029506 | 2.9009 | 2.901 | 0.00017166 |
| 28 | Accept | 2.9047 | 0.024909 | 2.9009 | 2.901 | 0.05559 |
| 29 | Accept | 2.9575 | 0.025391 | 2.9009 | 2.901 | 6.6263e-06 |
| 30 | Accept | 3.6716 | 0.025026 | 2.9009 | 2.901 | 1 |

**Objective function model**
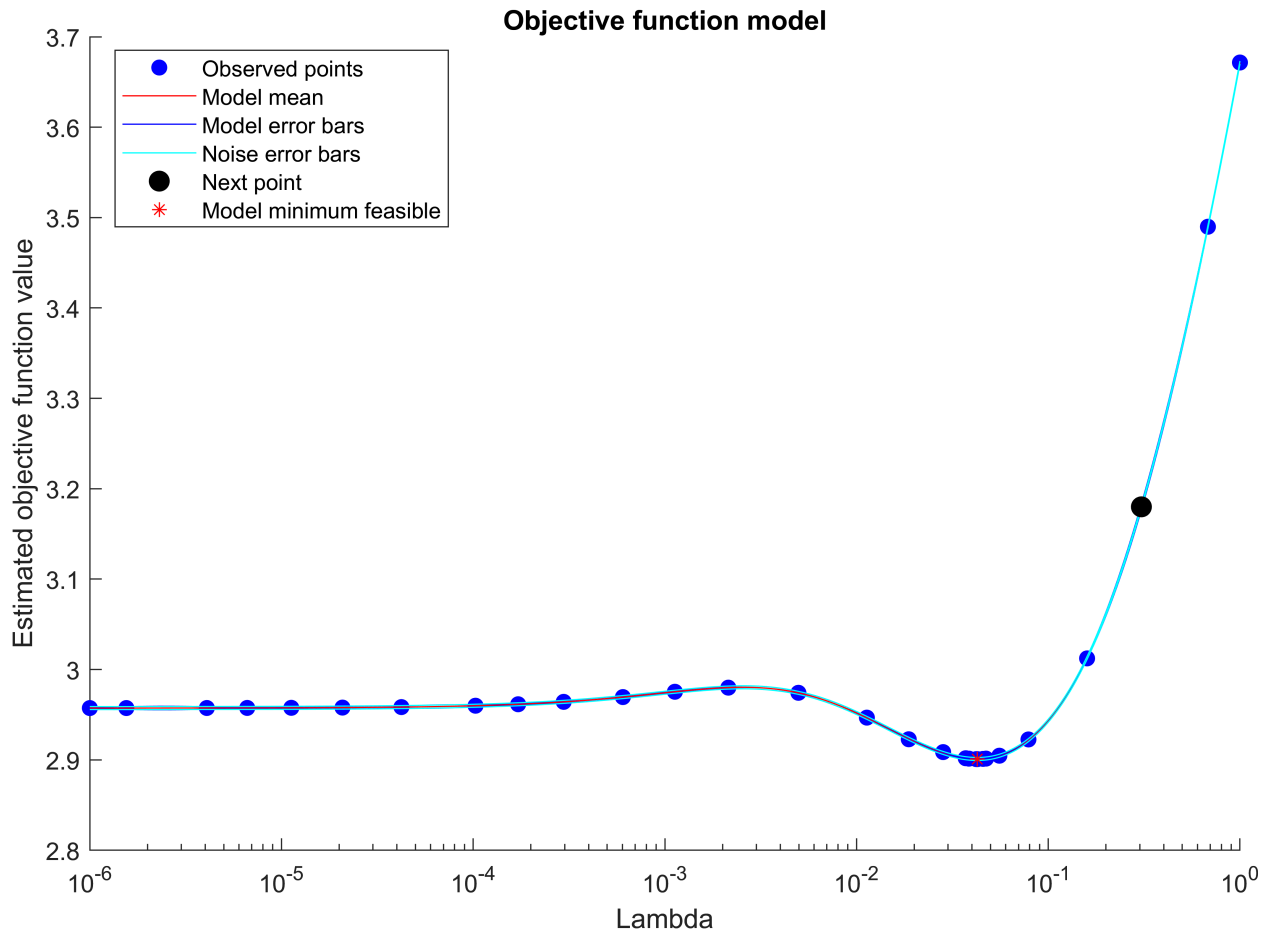
```
_____
Optimization completed.
MaxObjectiveEvaluations of 30 reached.
Total function evaluations: 30
Total elapsed time: 27.404 seconds.
Total objective function evaluation time: 1.1533

Best observed feasible point:
      Lambda
     _____

     0.042661

Observed objective function value = 2.9009
Estimated objective function value = 2.901
Function evaluation time = 0.031766

Best estimated feasible point (according to models):
      Lambda
     _____

     0.042661

Estimated objective function value = 2.901
Estimated function evaluation time = 0.032062
optimRegMdl =
  RegressionLinear
          ResponseName: 'Y'
     ResponseTransform: 'none'
                  Beta: [8×1 double]
```

```
        Bias: 6.5132
      Lambda: 0.0427
     Learner: 'leastsquares'

  Properties, Methods
```

```matlab
figure; hold on;
plot(xtrain*sd+mu,ytrain,'go','MarkerSize',10);
plot(xval*sd+mu,yval,'LineStyle','none','Marker','square','MarkerSize',10);
plt = plot(xplot(:,1)*sd+mu,Y(:,bestidx),'b','LineWidth',2);
plot(xplot(:,1)*sd+mu,predict(optimRegMdl,xplot),'r','LineWidth',2); hold off;
ylim([min(Outflow) max(Outflow)]);
xlabel('Change in water level (x)');
ylabel('Water flowing out of the dam (y)');
legend('Training','Validation',sprintf('lambda = %g',lambda(bestidx)),'Optimized lambda','Locat
title(sprintf('Best lambda estimated by hyperparameter\noptimization: %g',optimRegMdl.Lambda));
```
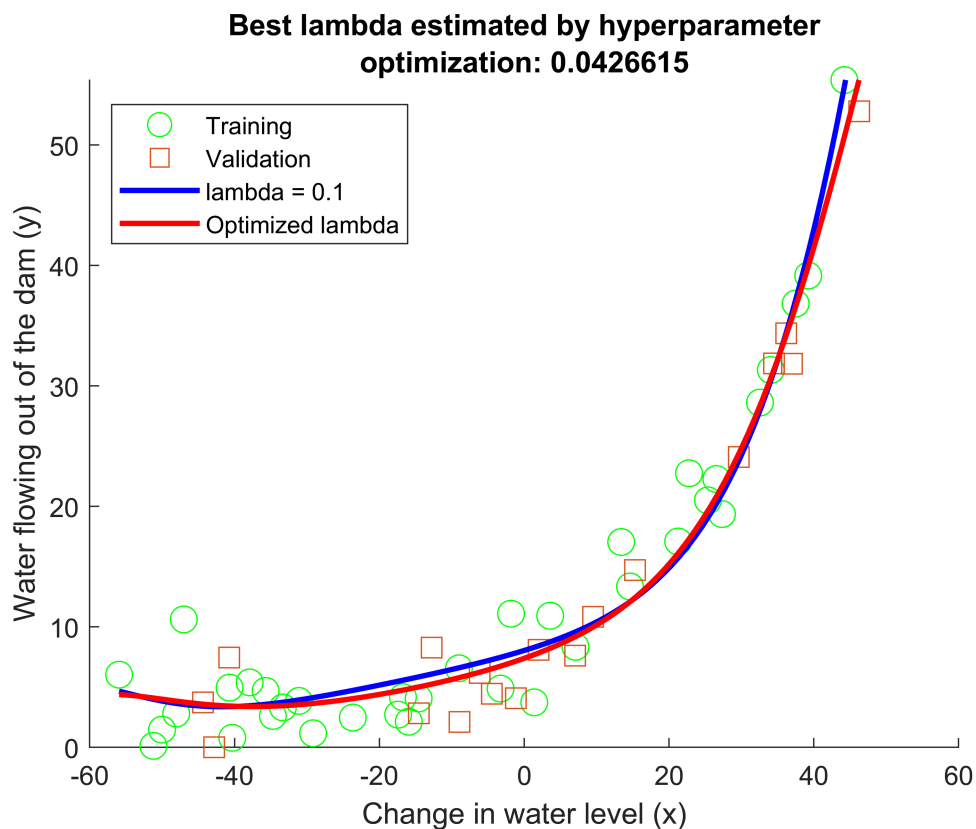


# K-Fold Cross-Validation

This section discusses the most common type of cross validation used with MATLAB machine learning functions - K-fold cross-validation - which is not used in the programming exercises.

## Introduction

Recall that the cross-validation technique used in the programming exercises is referred to as 'hold-out' validation, as the validation examples are 'held out' of the training process. This method is most often used when there is a large number of training examples relative to the number of parameters in the model. For

smaller datasets like the water flow example, the model coefficients and optimal parameter values can depend strongly on how the training and validation sets are distributed and the random nature with which they are selected. There are several alternative cross-validation methods available in MATLAB that are preferable in situations where there is not much data available. The most common is referred to as *K-fold* cross-validation,

The idea behind K-fold cross-validation is the following:

1. Start with a data set, *X* and choose a number of 'folds', *k*
2. Randomly select *k* equally sized subsets (folds) of *X*: $X_1, \dots, X_k$
3. Denote $X_1$ as the cross-validation set: $X_1^{cv}$ and combine the other subsets into a single training set, $X_1^t$
4. Train and cross-validate a model using $X_1^t$ and $X_1^{cv}$ to provide a set of model coefficients, $\theta_1$.
5. Repeat steps 3 and 4 with $X_i$ as the chosen validation set for $i = 2, \dots, k$, until there are *k* sets of trained coefficients, $\theta_1, \dots, \theta_k$
6. Average the *k* sets of coefficients to produce a final set of model coefficients, *θ*

## Using K-fold cross-validation instead of hold-out validation

In the following section we outline two ways to incorporate K-fold cross-validation instead of hold-out cross-validation when using MATLAB machine learning functions. Sample code is provided for each method, which can be used in the first part of the script if you are interested in trying out K-fold cross validation for yourself.

**Create a K-fold `cvpartition` variable for use with `fitlm` or `fitglm`:**

The code below will create a `cvpartition` variable for K-fold cross-validation:

```
cvpartition(m,'KFold',k);
```

- m is the number of examples in your data set
- k is the desired number of folds

**Use K-fold cross-validation with `fitrlinear` or `fitclinear`**

Use the `'KFold'` option with the desired number of folds:

```
fitrlinear(LevelChange,Outflow,'Learner','leastsquares','Regularization','Ridge','Lambda',lambda,'KFold',k)
```

**Use K-fold cross-validation and hyperparameter optimization with `fitrlinear` or `fitclinear`**

Create an options structure with the `'KFold'` option and number of folds, then pass it to a training function using the `'HyperparameterOptimizationOptions'` option:

```
opts = struct('KFold',k);
optimizableParams = hyperparameters('fitrlinear',LevelChange,Outflow);

optimRegMdl = fitrlinear(LevelChange,Outflow,'Learner','leastsquares','Regularization','Ridge',...
'OptimizeHyperparameters',{'Lambda'},HyperparameterOptimizationOptions',opts)
```

## Use K-fold cross-validation with the Regression or Classification Learner Apps

When training the 8th degree polynomial regression model without regularization and cross-validation, you most likely found that the resulting models looked very different after successive trainings which was a result of the

random nature of the partition selection and small dataset size. By using K-fold cross validation, we can achieve more consistent results- even before adding regularization! K-fold cross-validation can therefore be helpful when dealing with high variance due to small data set size. Run the code below to reload the data and create the 8th degree polynomial features, then follow the instructions to train a polynomial regression model using K-fold cross-validation using the Regression Learner App.

```
clear;
load ex5data1.mat;
LevelChange = [X;Xtest;Xval];
Outflow = [y;ytest;yval];
clearvars -except LevelChange Outflow;
mu = mean(LevelChange);
sd = std(LevelChange);
LevelChange = ((LevelChange-mu)/sd).^(1:8);
data = [LevelChange,Outflow];
```

**Note:** If you have difficulty reading the instructions below while the app is open in MATLAB Online, export this script to a pdf file which you can then use to display the instructions in a separate browser tab or window. To export this script, click on the 'Save' button in the 'Live Editor' tab above, then select 'Export to PDF'.

## Open the app and select the variables

1. In the MATLAB Apps tab, select the **Regression Learner** app from the Machine Learning section (you may need to expand the menu of available apps).
2. Select '**New Session -> From Workspace**' to start a new interactive session.
3. Under '**Data Set Variable'**, select '**data**'.
4. Under '**Validation**' select '**Cross-Validation**' (if not already selected).
5. Use the slider bar to select 10 folds.
6. Click the '**Start Session**' button.

## Select and train the model

1. In the Model Type list, the default model is '**Fine Tree**'. Expand the model list and select '**Linear**' from the '**Linear Regression Models**' list.
2. Select '**Train**' to train the model.

## Export and visualize the model

1. To plot the results vs. the original level change variable in the app, select '**column_1**' under '**X-axis** in the **'Response Plot'.**
2. Next, select '**Export Model -> Export Model**'.
3. Select the default output variable name ('**trainedModel**').
4. Extract the linear model variable and plot the model vs the dataset by running the code below

```
polyRegMdl = trainedModel.LinearModel
xplot = linspace(min(LevelChange(:,1)),max(LevelChange(:,1)))';
```

```
Y = predict(polyRegMdl,xplot.^(1:8));
figure; hold on;
plot(LevelChange(:,1)*sd+mu,Outflow,'bo','MarkerSize',10);
plot(xplot(:,1)*sd+mu,Y,'--','LineWidth',2); hold off;
ylim([min(Outflow) max(Outflow)]);
legend({'Data','Model'},'Location','northwest')
xlabel('Change in water level (x)');
ylabel('Water flowing out of the dam (y)');
```

Note that there is no distinction between training and validation data in the plot above- all data values are used in training and validation with K-fold cross-validation. Compare the plot above to the original unregularized 8th degree model from the first section:

- Which method performs better?
- Which is more consistent after multiple trainings?

Next, how does the current K-fold validated model compare to the 8th order, regularized model developed in the second section of this script?