Exercise 5 - Regularized Linear Regression and Bias v.s. Variance

Azka NA

December 29, 2020

1 Introduction

This is the guide for Andrew Ng's Machine Learning course programming assignment done in Python, adapted from the original guide written for Octave or MATLAB.

In this exercise, you will implement regularized linear regression and use it to study models with different bias-variance properties. Before starting on this programming exercise, we strongly recommend watching the video lectures and completing the review questions for the associated topics

For Programming Exercise 5: Regularized Linear Regression and Bias v.s. Variance, you will need to download the following files:

`exercise5.ipynb` - Jupyter notebook containing the script

`ex5data1.mat` - Dataset

---

2 Regularized Linear Regression

In the first half of the exercise, you will implement regularized linear regression to predict the amount of water flowing out of a dam using the change of water level in a reservoir. In the next half, you will go through some diagnostics of debugging learning algorithms and examine the effects of bias v.s. variance.

2.1 Visualizing the dataset

We will begin by visualizing the dataset containing historical records on the change in the water level, x, and the amount of water flowing out of the dam, y, as shown in Figure 1

This dataset is divided into three parts:

- A **training** set that your model will learn on: `X, y`

- A **cross validation** set for determining the regularization parameter: `Xval, yval`
- A **test** set for evaluating performance. These are "unseen" examples which your model did not see during training: `Xtest, ytest`

In the following parts, you will implement linear regression and use that to fit a straight line to the data and plot learning curves. Following that, you will implement polynomial regression to find a better fit to the data.

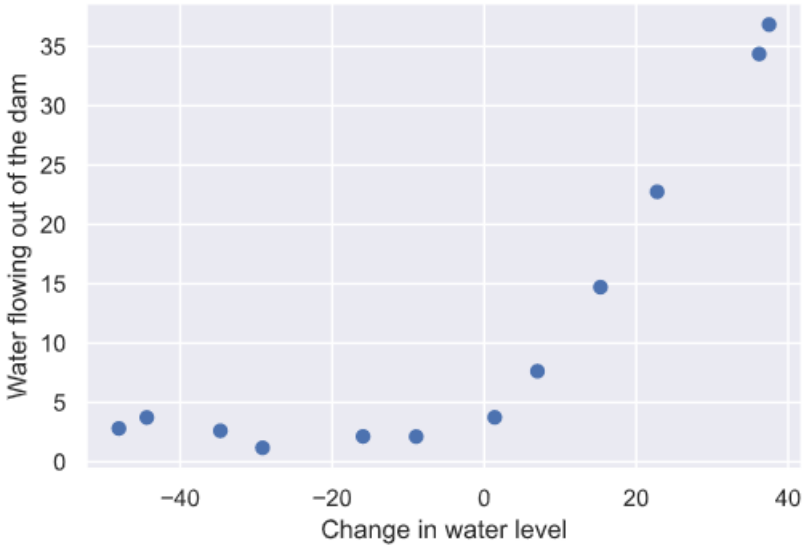

Figure 1: Data

2.2 Regularized linear regression cost function

Recall that regularized linear regression has the following cost function:

$$J(\theta_1) = \frac{1}{2m}\Big(\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2\Big) + \frac{\lambda}{2m}\Big(\sum_{j=1}^{n}{\theta_j}^2\Big), \tag{1}$$

where λ is a regularization parameter which controls the degree of regularization (thus, help preventing overfitting). The regularization term puts a penalty on the overall cost J. As the magnitudes of the model parameters θ_j increase, the penalty increases as well. Note that you should not regularize the θ_0 term.

Your task is to write a function to calculate the regularized linear regression cost function. If possible, try to vectorize your code and avoid writing loops.

```
def linearRegCostFunction(X, y, theta, Lambda):
  m = len(y)
  y = y.reshape(m,1)
  grad = np.zeros(theta.shape)
  n_1 = theta.shape[0]
  theta = theta.reshape(n_1,1)
  cost = 0
  h = np.dot(X, theta)
  cost = 1/(2*m) * np.sum((h-y)**2)
  cost_reg = cost + Lambda/(2*m)*np.sum(theta[1:]**2)

  grad[0] = (1/m)*(X[:,0:1].reshape(m,1)*(h - y)).sum(axis=0)
  grad[1:] = (1/m)*( (X[:,1:]*(h - y)).sum(axis=0)  + Lambda*theta
 [1:].reshape(n_1-1))

  return cost_reg, grad
```

When you are finished, run your cost function using `theta` initialized at `[1,1]`. You should expect to see an output of 303.993.

2.3 Regularized linear regression gradient

Correspondingly, the partial derivative of regularized linear regression's cost for θ_j is defined as

$$\frac{\partial J(\theta)}{\partial \theta_0} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \qquad \text{for} j = 0 \tag{2}$$

$$\frac{\partial J(\theta)}{\partial \theta_j} = \left(\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}\right) + \frac{\lambda}{m}\theta_j \qquad \text{for} j \geq 1 \tag{3}$$

Run your cost function using theta initialized at [1,1]. You should expect to see a gradient of `[-15.30, 598.250]`.

2.4 Fitting linear regression

Once your cost function and gradient are working correctly, compute the optimal values of θ using the `minimize` function from Scipy's `Optimize` module.

In this part, we set regularization parameter λ to zero. Because our current implementation of linear regression is trying to fit a 2-dimensional θ, regularization will not be incredibly helpful for a θ of such low dimension. In the later parts of the exercise, you will be using polynomial regression with regularization.

```
def trainLinearReg(linearRegCostFunction, X, y, Lambda=0, maxiter
 =200):

```

```
    initial_theta = np.zeros(X.shape[1])
    costFunction = lambda t: linearRegCostFunction(X, y, t, Lambda)
    options = {'maxiter': maxiter}
    minimizef = optimize.minimize(costFunction, initial_theta, jac=
True, method='TNC', options=options)
    return minimizef.x
```

Finally, we should also plot the best fit line, resulting in an image similar to Figure 2. The best fit line tells us that the model is not a good fit to the data because the data has a non-linear pattern. While visualizing the best fit as shown is one possible way to debug your learning algorithm, it is not always easy to visualize the data and model. In the next section, you will implement a function to generate learning curves that can help you debug your learning algorithm even if it is not easy to visualize the data.

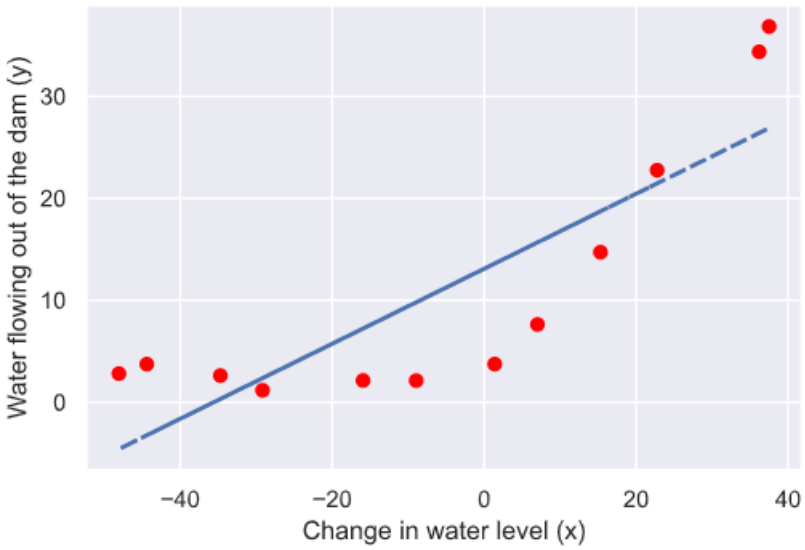

Figure 2: Linear fit

3 Bias-variance

An important concept in machine learning is the bias-variance tradeoff. Models with high bias are not complex enough for the data and tend to underfit, while models with high variance overfit to the training data.

In this part of the exercise, you will plot training and test errors on a learning curve to diagnose bias-variance problems.

3.1 Learning curves

You will now implement code to generate the learning curves that will be useful in debugging learning algorithms. Recall that a learning curve plots training and cross-validation error as a function of training set size. Your job is to write a function so that it returns a vector of errors for the training set and cross-validation set.

To plot the learning curve, we need a training and cross-validation set error for different training set sizes. To obtain different training set sizes, you should use different subsets of the original training set `X`. Specifically, for a training set size of i, you should use the first i examples (i.e., `X[:i,:]` and `y[:i]`).

You can use the `trainLinearReg` function to find the θ parameters. Note that the lambda is passed as a parameter to the `learningCurve` function. After learning the θ parameters, you should compute the **error** on the training and cross-validation sets. Recall that the training error for a dataset is defined as

$$J(\theta_1) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2\right]. \tag{4}$$

In particular, note that the training error does not include the regularization term. One way to compute the training error is to use your existing cost function and set λ to 0 only when using it to compute the training error and cross-validation error. When you are computing the training set error, make sure you compute it on the training subset (i.e., `X[:n]` and `y[:n]`) (instead of the entire training set). However, for the cross-validation error, you should compute it over the entire cross-validation set. You should store the computed errors in the vectors `training_error` and `validation_error`.

```
1  def learningCurve(X, y, Xval, yval, Lambda=0):
2    m = len(y)
3    training_error = np.zeros(m)
4    validation_error = np.zeros(m)
5
6    for i in range(m):
7        theta_optimized1 = trainLinearReg(linearRegCostFunction, X[:i
   +1], y[:i+1], Lambda)
8        training_error[i], _ = linearRegCostFunction(X[:i+1], y[:i+1],
    theta_optimized1, Lambda)
9        validation_error[i], _ = linearRegCostFunction(Xval, yval,
   theta_optimized1, Lambda)
10
11   return training_error, validation_error
```

When you are finished, print the learning curves and make a plot similar to Figure 3.

In Figure 3, you can observe that both the train error and cross-validation error are high when the number of training examples is increased. This reflects a **high bias** problem in the model — the linear regression model is too simple and is unable to fit

Figure 3: Linear regression learning curve

our dataset well. In the next section, you will implement polynomial regression to fit a better model for this dataset.

4 Polynomial regression

The problem with our linear model was that it was too simple for the data and resulted in underfitting (high bias). In this part of the exercise, you will address this problem by adding more features.

For use polynomial regression, our hypothesis has the form:

$$h_\theta(x) = \theta_0 + \theta_1 * (\text{waterLevel}) + \theta_2 * (\text{waterLevel})^2 + \cdots + \theta_p * (\text{waterLevel})^p \quad (5)$$
$$= \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_p x_p \quad (6)$$

Notice that by defining $x_1 = (\text{waterLevel})$, $x_2 = (\text{waterLevel})^2$, $\ldots$, $x_p = (\text{waterLevel})^p$, we obtain a linear regression model where the features are the various powers of the original value (waterLevel).

Now, you will add more features using the higher powers of the existing feature x in the dataset. Your task in this part is to complete the code for `polyFeatures` so that the function maps the original training set `X` of size $m \times 1$ into its higher powers. Specifically, when a training set `X` of size $m \times 1$ is passed into the function, the function should return a $m \times p$ matrix `X_poly`, where column 1 holds the original values of `X`,

column 2 holds the values of `X.^2`, column 3 holds the values of `X.^3`, and so on. Note that you don't have to account for the zero-eth power in this function.

```
def polyFeatures(X, p):
  m = X.shape[0]
  X_out = np.zeros((m,p))
  for i in range(p):
      X_out[:,i] = X.flatten()**(i+1)
  return X_out
```

Now you have a function that will map features to a higher dimension, then apply it to the training set, the test set, and the cross-validation set (which you haven't used yet).

4.1 Learning Polynomial Regression

After you have completed `polyFeatures`, we will proceed to train polynomial regression using your linear regression cost function.

Keep in mind that even though we have polynomial terms in our feature vector, we are still solving a linear regression optimization problem. The polynomial terms have simply turned into features that we can use for linear regression. We are using the same cost function and gradient that you wrote for the earlier part of this exercise.

For this part of the exercise, you will be using a polynomial of degree 8. It turns out that if we run the training directly on the projected data, will not work well as the features would be badly scaled (e.g., an example with $x = 40$ will now have a feature $x_8 = 40^8 = 6.5 \times 10^{12}$). Therefore, you will need to use feature normalization.

Before learning the parameters θ for the polynomial regression, we will first call `featureNormalize` and normalize the features of the training set, storing the mu, sigma parameters separately.

After learning the parameters θ, you should see two plots (Figure 4,5) generated for polynomial regression with $\lambda = 0$.

From Figure 4, you should see that the polynomial fit is able to follow the datapoints very well - thus, obtaining a low training error. However, the polynomial fit is very complex and even drops off at the extremes. This is an indicator that the polynomial regression model is overfitting the training data and will not generalize well.

To better understand the problems with the unregularized ($\lambda = 0$) model, you can see that the learning curve (Figure 5) shows the same effect where the low training error is low, but the cross validation error is high. There is a gap between the training and cross validation errors, indicating a high variance problem.

One way to combat the overfitting (high-variance) problem is to add regularization to the model. In the next section, you will get to try different λ parameters to see how regularization can lead to a better model.

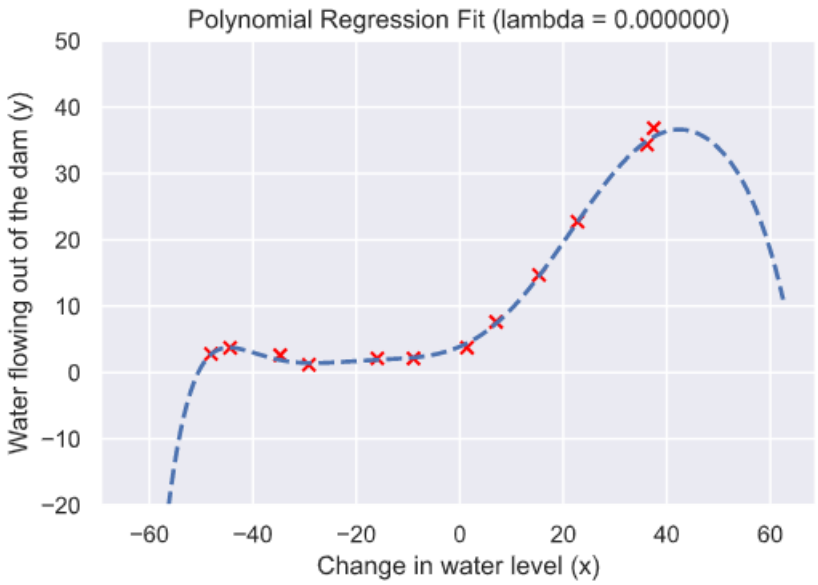

Figure 4: Polynomial fit, $\lambda = 0$

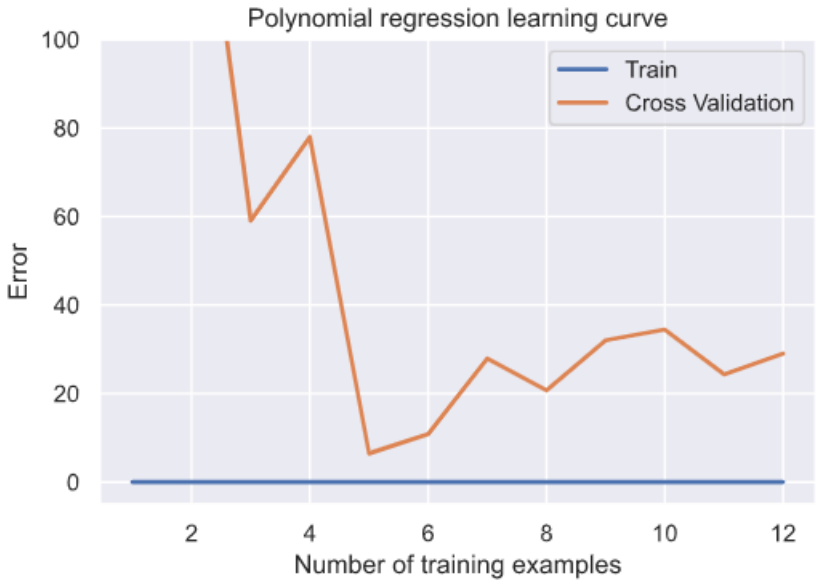

Figure 5: Polynomial learning curve, $\lambda = 0$

4.2 Optional (ungraded) exercise: Adjusting the regularization parameter

In this section, you will get to observe how the regularization parameter affects the bias-variance of regularized polynomial regression. You should now modify the the `Lambda`

parameter and try $\lambda = 1, 100$. For each of these values, the script should generate a polynomial fit to the data and also a learning curve.

For $\lambda = 1$, you should see a polynomial fit that follows the data trend well (Figure 6) and a learning curve (Figure 7) showing that both the cross-validation and training error converge to a relatively low value. This shows the $\lambda = 1$ regularized polynomial regression model does not have the highbias or high-variance problems. In effect, it achieves a good trade-off between bias and variance.

For $\lambda = 100$, you should see a polynomial fit (Figure 8) that does not follow the data well. In this case, there is too much regularization and the model is unable to fit the training data.

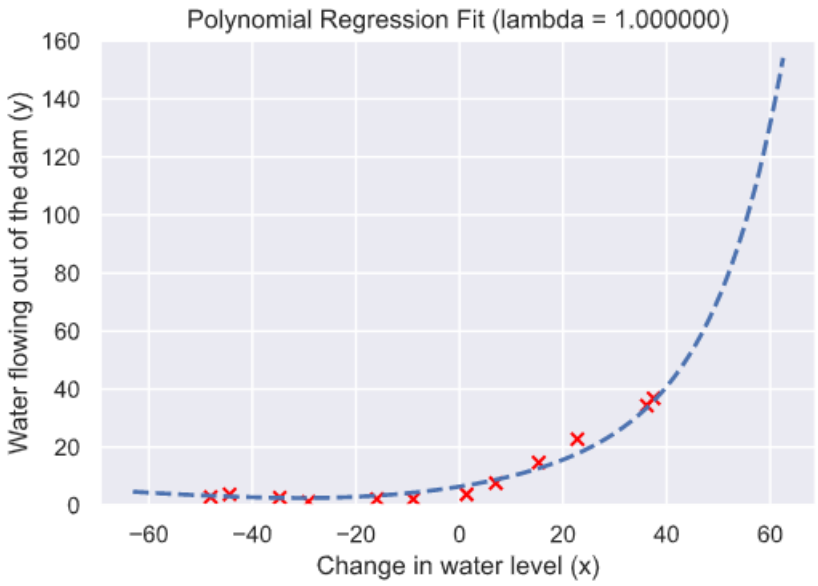

Figure 6: Polynomial fit, $\lambda = 1$

4.3 Selecting λ using a cross validation set

From the previous parts of the exercise, you observed that the value of λ can significantly affect the results of regularized polynomial regression on the training and cross-validation set. In particular, a model without regularization ($\lambda = 0$) fits the training set well, but does not generalize. Conversely, a model with too much regularization ($\lambda = 100$) does not fit the training set and testing set well. A good choice of λ (e.g., $\lambda = 1$) can provide a good fit to the data.

In this section, you will implement an automated method to select the λ parameter. Concretely, you will use a cross-validation set to evaluate how good each λ value is. After selecting the best λ value using the cross-validation set, we can then evaluate the

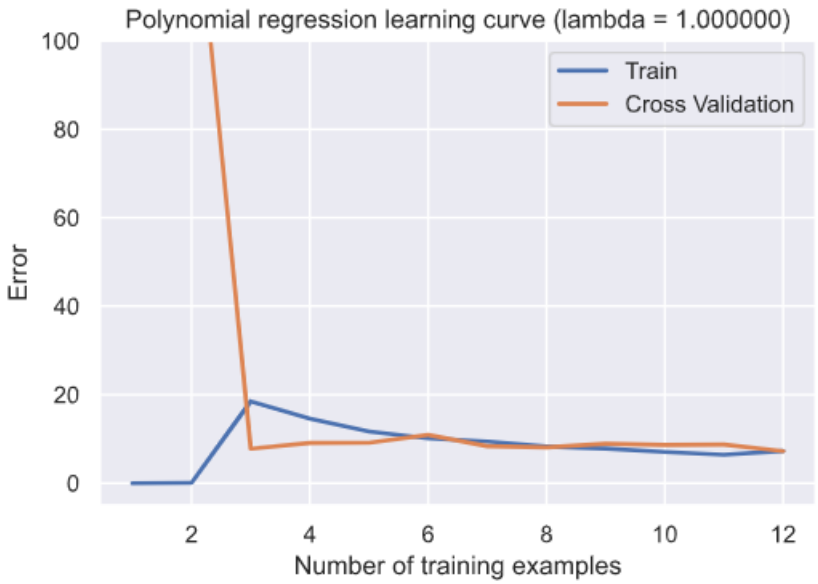

Figure 7: Polynomial learning curve, $\lambda = 1$

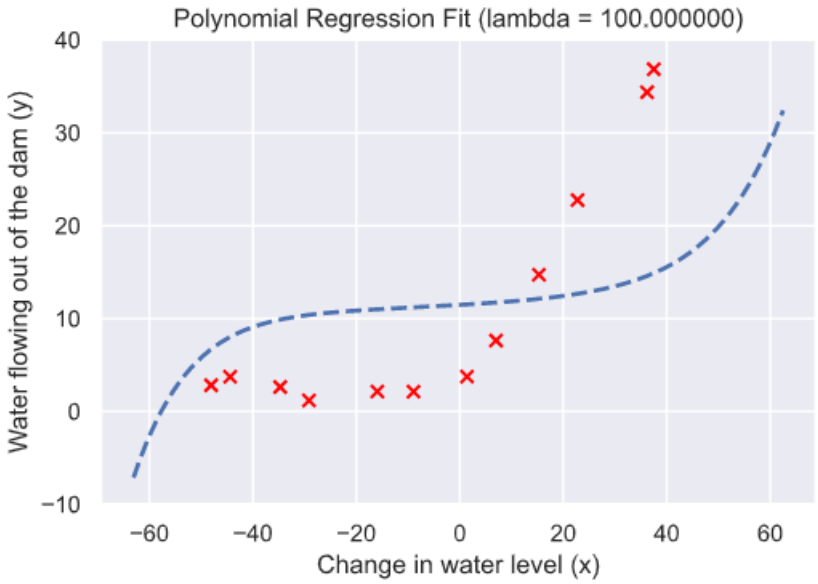

Figure 8: Polynomial fit, $\lambda = 100$

model on the test set to estimate how well the model will perform on actual unseen data.

Your task is to complete the code for `validationCurve`. Specifically, you should should use the `trainLinearReg` function to train the model using different values of λ and compute the training error and cross-validation error. You should try λ in the

following range: {0, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, 3, 10}.

```
def validationCurve(X, y, Xval, yval):

  lambda_vec = [0, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, 3, 10]

  training_error = np.zeros(len(lambda_vec))
  validation_error = np.zeros(len(lambda_vec))

  for i in range(len(lambda_vec)):
    lambda_try = lambda_vec[i]
    theta_t = trainLinearReg(linearRegCostFunction, X, y, Lambda =
 lambda_try)
    training_error[i], _ = linearRegCostFunction(X, y, theta_t,
 Lambda = 0)
    validation_error[i], _ = linearRegCostFunction(Xval, yval,
 theta_t, Lambda = 0)

  return lambda_vec, training_error, validation_error
```

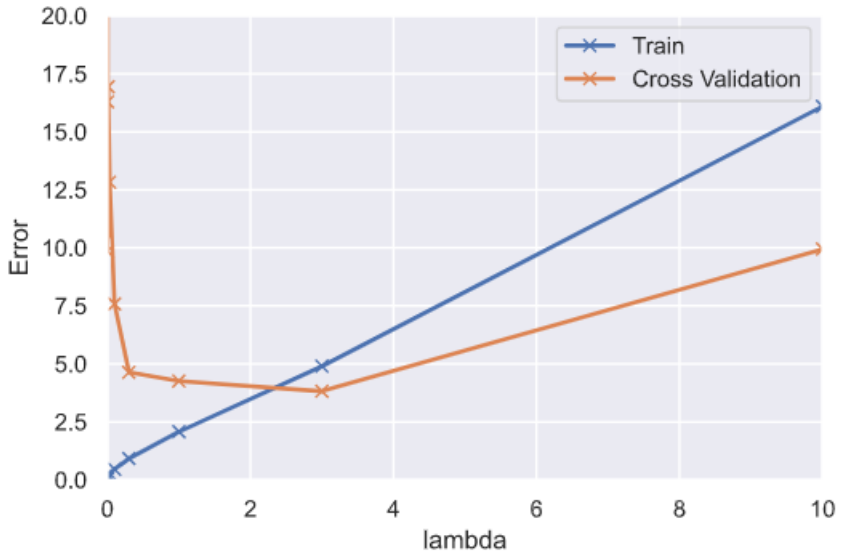

Figure 9: Selecting λ using a cross-validation set

After you have completed the code, run your function to plot a cross-validation curve of error v.s. λ that allows you select which λ parameter to use. You should see a plot similar to Figure 9. In this figure, we can see that the best value of λ is around 3. Due to randomness in the training and validation splits of the dataset, the cross-validation error can sometimes be lower than the training error.

4.4 Optional (ungraded) exercise: Computing test set error

In the previous part of the exercise, you implemented code to compute the cross-validation error for various values of the regularization parameter λ. However, to get a better indication of the model's performance in the real world, it is important to evaluate the "final" model on a test set that was not used in any part of training (that is, it was neither used to select the λ parameters, nor to learn the model parameters θ).

For this optional (ungraded) exercise, you should compute the test error using the best value of λ you found. In our cross validation, we obtained a test error of 3.8599 for $\lambda = 3$.

4.5 Optional (ungraded) exercise: Plotting learning curves with randomly selected examples

In practice, especially for small training sets, when you plot learning curves to debug your algorithms, it is often helpful to average across multiple sets of randomly selected examples to determine the training error and cross validation error.

Concretely, to determine the training error and cross validation error for i examples, you should first randomly select i examples from the training set and i examples from the cross validation set. You will then learn the parameters θ using the randomly chosen training set and evaluate the parameters θ on the randomly chosen training set and cross validation set. The above steps should then be repeated multiple times (say 50) and the averaged error should be used to determine the training error and cross validation error for i examples.