Programming Exercise 5:

Regularized Linear Regression and Bias vs Variance

Introduction

In this exercise, you will implement regularized linear regression and use it to study models with different bias-variance properties.

All the information you need for solving this assignment is in this notebook, and all the code you will be implementing will take place within this notebook.

Before we begin with the exercises, we need to import all libraries required for this programming exercise. Throughout the course, we will be using `numpy` for all arrays and matrix operations, `matplotlib` for plotting, and `scipy` for scientific and numerical computation functions and tools.

In [1]:
```
# used for manipulating directory paths
import os

# Scientific and vector computation for python
import numpy as np

# Plotting library
from matplotlib import pyplot

# Optimization module in scipy
from scipy import optimize

# will be used to load MATLAB mat datafile format
from scipy.io import loadmat

# library written for this exercise providing additional functions for assignment s
import utils

# tells matplotlib to embed plots within the notebook
%matplotlib inline
```

Outline

The following is a breakdown of each part of this exercise.

Section	Part	Submitted Function
1	Regularized Linear Regression Cost Function	`linearRegCostFunction`
2	Regularized Linear Regression Gradient	`linearRegCostFunction`
3	Learning Curve	`learningCurve`

Section	Part	Submitted Function
4	Polynomial Feature Mapping	`polyFeatures`
5	Cross Validation Curve	`validationCurve`

1 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.

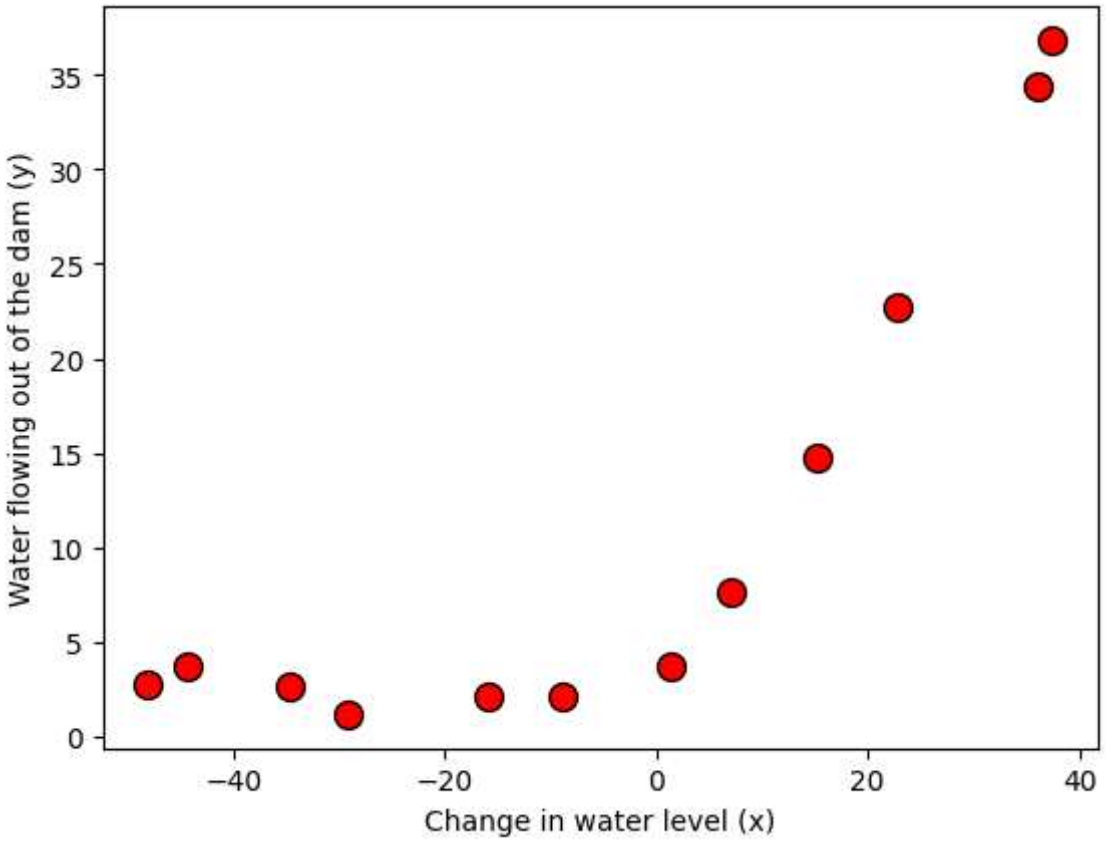
1.1 Visualizing the dataset

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. 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`

Run the next cell to plot the training data. 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.

In [2]:
```
# Load from ex5data1.mat, where all variables will be store in a dictionary
data = loadmat(os.path.join('Data', 'ex5data1.mat'))

# Extract train, test, validation data from dictionary
# and also convert y's form 2-D matrix (MATLAB format) to a numpy vector
X, y = data['X'], data['y'][:, 0]
Xtest, ytest = data['Xtest'], data['ytest'][:, 0]
Xval, yval = data['Xval'], data['yval'][:, 0]

# m = Number of examples
m = y.size

# Plot training data
pyplot.plot(X, y, 'ro', ms=10, mec='k', mew=1)
pyplot.xlabel('Change in water level (x)')
pyplot.ylabel('Water flowing out of the dam (y)');
```



1.2 Regularized linear regression cost function

Recall that regularized linear regression has the following cost function:

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

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.

You should now complete the code in the function `linearRegCostFunction` in the next cell. Your task is to calculate the regularized linear regression cost function. If possible, try to vectorize your code and avoid writing loops.

In [3]:
```
def linearRegCostFunction(X, y, theta, lambda_=0.0):
    """
    Compute cost and gradient for regularized linear regression
    with multiple variables. Computes the cost of using theta as
    the parameter for linear regression to fit the data points in X and y.

    Parameters
    ----------
    X : array_like
        The dataset. Matrix with shape (m x n + 1) where m is the
        total number of examples, and n is the number of features
        before adding the bias term.

    y : array_like
        The functions values at each datapoint. A vector of
```

```
        shape (m, ).

    theta : array_like
        The parameters for linear regression. A vector of shape (n+1,).

    lambda_ : float, optional
        The regularization parameter.

    Returns
    -------
    J : float
        The computed cost function.

    grad : array_like
        The value of the cost function gradient w.r.t theta.
        A vector of shape (n+1, ).

    Instructions
    ------------
    Compute the cost and gradient of regularized linear regression for
    a particular choice of theta.
    You should set J to the cost and grad to the gradient.
    """
    # Initialize some useful values
    m = y.size # number of training examples

    # You need to return the following variables correctly
    J = 0
    grad = np.zeros(theta.shape)

    # ====================== YOUR CODE HERE ======================
    h = X.dot(theta)
    J = (1 / (2 * m)) * np.sum(np.square(h - y)) + (lambda_ / (2 * m)) * np.sum(np.

    grad = (1 / m) * (h - y).dot(X)

    grad[1:] = grad[1:] + (lambda_ / m) * theta[1:]



    # ============================================================
    return J, grad
```

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

In [4]:
```
theta = np.array([1, 1])
J, _ = linearRegCostFunction(np.concatenate([np.ones((m, 1)), X], axis=1), y, theta

print('Cost at theta = [1, 1]:\t   %f ' % J)
print('This value should be about 303.993192)\n ' )
```

```
Cost at theta = [1, 1]:    303.993192
This value should be about 303.993192)
```

1.3 Regularized linear regression gradient

Correspondingly, the partial derivative of the cost function for regularized linear regression is defined as:

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

$$\frac{\partial J(\theta)}{\partial \theta_j} = \left(\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)x_j^{(i)}\right) + \frac{\lambda}{m}\theta_j \qquad \text{for } j \geq 1 \qquad (2)$$

In the function `linearRegCostFunction` above, add code to calculate the gradient, returning it in the variable `grad`. **Do not forget to re-execute the cell containing this function to update the function's definition.**

When you are finished, use the next cell to run your gradient function using theta initialized at `[1, 1]`. You should expect to see a gradient of `[-15.30, 598.250]`.

In [5]:
```
theta = np.array([1, 1])
J, grad = linearRegCostFunction(np.concatenate([np.ones((m, 1)), X], axis=1), y, th

print('Gradient at theta = [1, 1]:  [{:.6f}, {:.6f}] '.format(*grad))
print(' (this value should be about [-15.303016, 598.250744])\n')
```

```
Gradient at theta = [1, 1]:  [-15.303016, 598.250744]
 (this value should be about [-15.303016, 598.250744])
```

Fitting linear regression

Once your cost function and gradient are working correctly, the next cell will run the code in `trainLinearReg` (found in the module `utils.py`) to compute the optimal values of θ. This training function uses `scipy`'s optimization module to minimize the cost function.

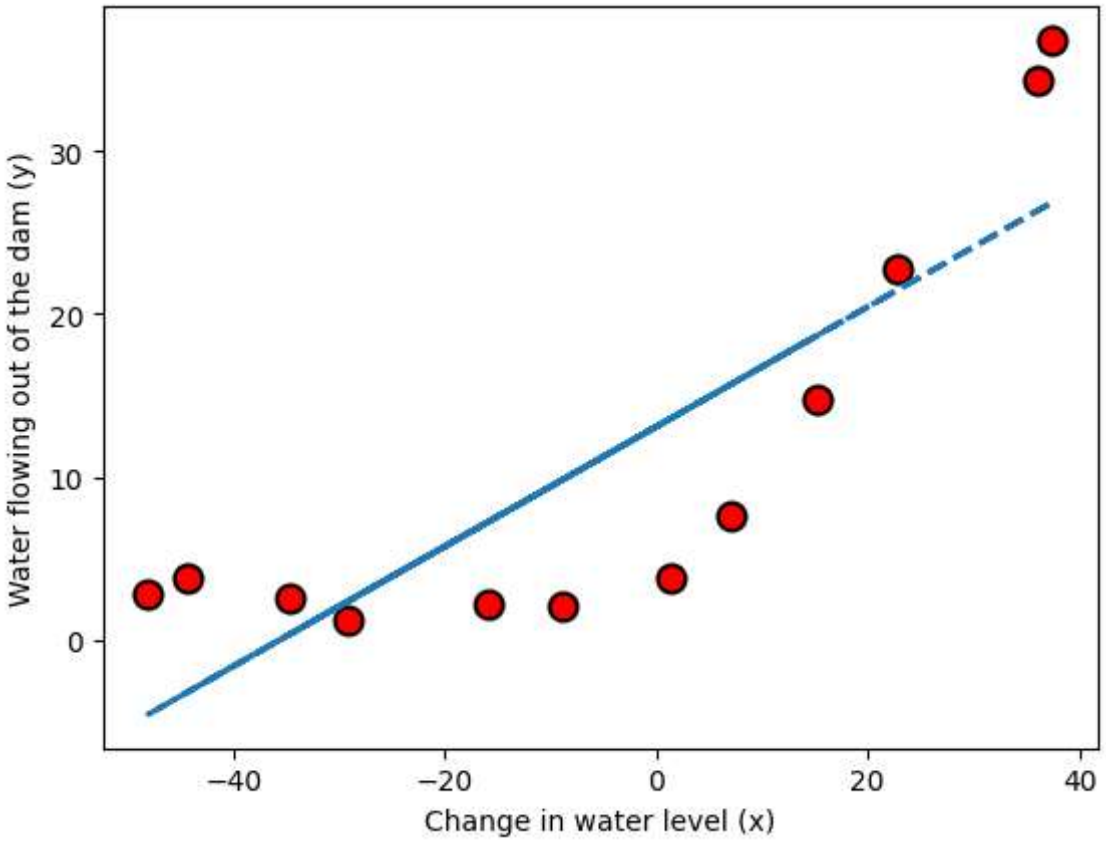
In this part, we set regularization parameter λ to zero. Because our current implementation of linear regression is trying to fit a 2-dimensional θ, regularization will not be incredibly helpful for a θ of such low dimension. In the later parts of the exercise, you will be using polynomial regression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

Finally, the code in the next cell should also plot the best fit line, which should look like the figure below.

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.

In [6]:
```
# add a columns of ones for the y-intercept
X_aug = np.concatenate([np.ones((m, 1)), X], axis=1)
theta = utils.trainLinearReg(linearRegCostFunction, X_aug, y, lambda_=0)

#  Plot fit over the data
pyplot.plot(X, y, 'ro', ms=10, mec='k', mew=1.5)
```

```
pyplot.xlabel('Change in water level (x)')
pyplot.ylabel('Water flowing out of the dam (y)')
pyplot.plot(X, np.dot(X_aug, theta), '--', lw=2);
```



2 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.

2.1 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 fill in the function `learningCurve` in the next cell, 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 (by calling `utils.trainLinearReg(...)`) 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

$$J_{\text{train}} = \frac{1}{2m}\left[\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2\right]$$

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 error train and error val.

In [7]:
```
def learningCurve(X, y, Xval, yval, lambda_=0):
    """
    Generates the train and cross validation set errors needed to plot a learning c
    returns the train and cross validation set errors for a learning curve.

    In this function, you will compute the train and test errors for
    dataset sizes from 1 up to m. In practice, when working with larger
    datasets, you might want to do this in larger intervals.

    Parameters
    ----------
    X : array_like
        The training dataset. Matrix with shape (m x n + 1) where m is the
        total number of examples, and n is the number of features
        before adding the bias term.

    y : array_like
        The functions values at each training datapoint. A vector of
        shape (m, ).

    Xval : array_like
        The validation dataset. Matrix with shape (m_val x n + 1) where m is the
        total number of examples, and n is the number of features
        before adding the bias term.

    yval : array_like
        The functions values at each validation datapoint. A vector of
        shape (m_val, ).

    lambda_ : float, optional
        The regularization parameter.

    Returns
    -------
    error_train : array_like
        A vector of shape m. error_train[i] contains the training error for
        i examples.
    error_val : array_like
        A vecotr of shape m. error_val[i] contains the validation error for
        i training examples.

    Instructions
```

```
    ------------
    Fill in this function to return training errors in error_train and the
    cross validation errors in error_val. i.e., error_train[i] and
    error_val[i] should give you the errors obtained after training on i examples.

    Notes
    -----
    - You should evaluate the training error on the first i training
      examples (i.e., X[:i, :] and y[:i]).

      For the cross-validation error, you should instead evaluate on
      the _entire_ cross validation set (Xval and yval).

    - If you are using your cost function (linearRegCostFunction) to compute
      the training and cross validation error, you should call the function with
      the lambda argument set to 0. Do note that you will still need to use
      lambda when running the training to obtain the theta parameters.

    Hint
    ----
    You can loop over the examples with the following:

           for i in range(1, m+1):
               # Compute train/cross validation errors using training examples
               # X[:i, :] and y[:i], storing the result in
               # error_train[i-1] and error_val[i-1]
               ....
    """
    # Number of training examples
    m = y.size

    # You need to return these values correctly
    error_train = np.zeros(m)
    error_val   = np.zeros(m)

    # ====================== YOUR CODE HERE ======================
    for i in range(1, m + 1):
        theta_t = utils.trainLinearReg(linearRegCostFunction, X[:i], y[:i], lambda_
        error_train[i - 1], _ = linearRegCostFunction(X[:i], y[:i], theta_t, lambda
        error_val[i - 1], _ = linearRegCostFunction(Xval, yval, theta_t, lambda_ =


    # =============================================================
    return error_train, error_val
```

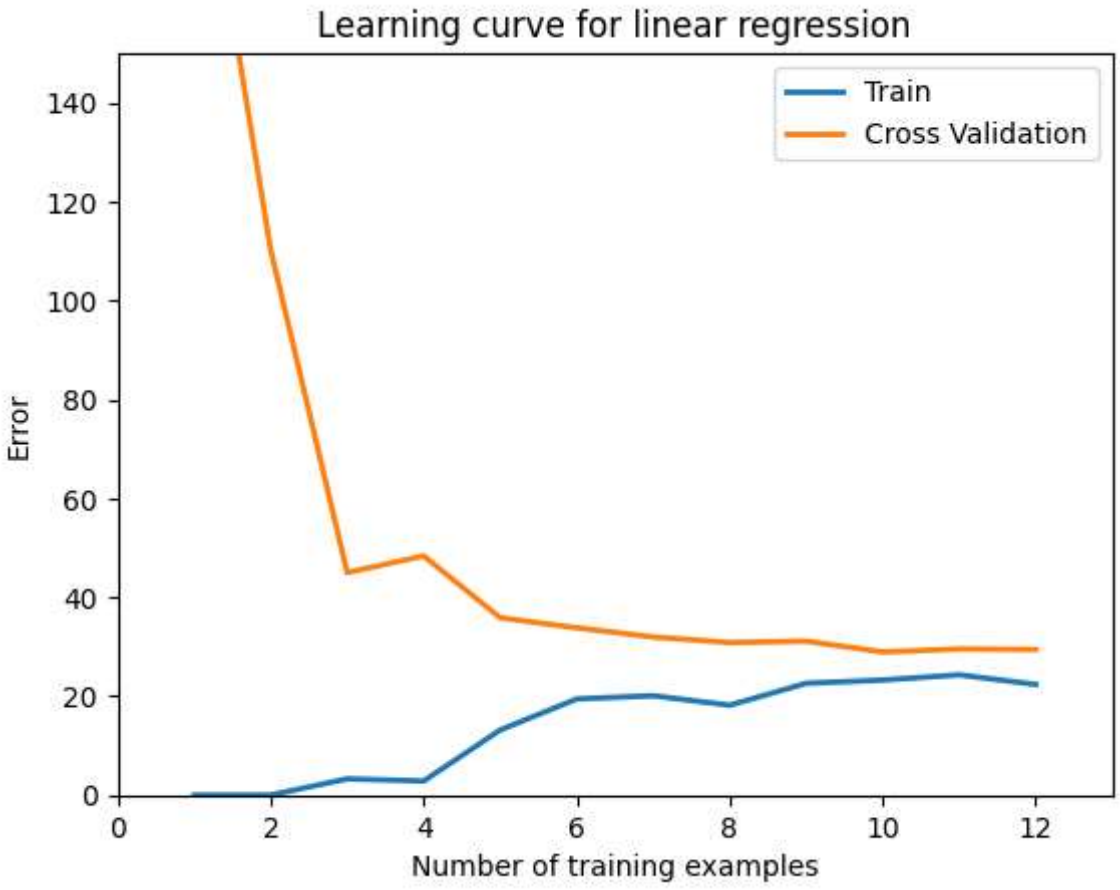
When you are finished implementing the function `learningCurve`, executing the next cell prints the learning curves and produce a plot similar to the figure below.

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

In [8]:
```
X_aug = np.concatenate([np.ones((m, 1)), X], axis=1)
Xval_aug = np.concatenate([np.ones((yval.size, 1)), Xval], axis=1)
error_train, error_val = learningCurve(X_aug, y, Xval_aug, yval, lambda_=0)
```

```
pyplot.plot(np.arange(1, m+1), error_train, np.arange(1, m+1), error_val, lw=2)
pyplot.title('Learning curve for linear regression')
pyplot.legend(['Train', 'Cross Validation'])
pyplot.xlabel('Number of training examples')
pyplot.ylabel('Error')
pyplot.axis([0, 13, 0, 150])

print('# Training Examples\tTrain Error\tCross Validation Error')
for i in range(m):
    print('  \t%d\t\t%f\t%f' % (i+1, error_train[i], error_val[i]))
```

```
# Training Examples     Train Error     Cross Validation Error
        1               0.000000        205.121096
        2               0.000000        110.302641
        3               3.286595        45.010231
        4               2.842678        48.368911
        5               13.154049       35.865165
        6               19.443963       33.829962
        7               20.098522       31.970986
        8               18.172859       30.862446
        9               22.609405       31.135998
        10              23.261462       28.936207
        11              24.317250       29.551432
        12              22.373906       29.433818
```



3 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 polynomial regression, our hypothesis has the form:

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

Notice that by defining $x_1 = (\text{waterLevel})$, $x_2 = (\text{waterLevel})^2$, $\cdots$, $x_p = (\text{waterLevel})^p$, 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 in the function `polyFeatures` in the next cell. The function should map 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.

In [9]:
```
def polyFeatures(X, p):
    """
    Maps X (1D vector) into the p-th power.

    Parameters
    ----------
    X : array_like
        A data vector of size m, where m is the number of examples.

    p : int
        The polynomial power to map the features.

    Returns
    -------
    X_poly : array_like
        A matrix of shape (m x p) where p is the polynomial
        power and m is the number of examples. That is:

        X_poly[i, :] = [X[i], X[i]**2, X[i]**3 ...  X[i]**p]

    Instructions
    ------------
    Given a vector X, return a matrix X_poly where the p-th column of
    X contains the values of X to the p-th power.
    """
    # You need to return the following variables correctly.
    X_poly = np.zeros((X.shape[0], p))

    # ====================== YOUR CODE HERE ======================


    for i in range(p):
        X_poly[:, i] = X[:, 0] ** (i + 1)

    # ============================================================
    return X_poly
```

Now you have a function that will map features to a higher dimension. The next cell will apply it to the training set, the test set, and the cross validation set.

In [10]:
```
p = 8

# Map X onto Polynomial Features and Normalize
X_poly = polyFeatures(X, p)
X_poly, mu, sigma = utils.featureNormalize(X_poly)
X_poly = np.concatenate([np.ones((m, 1)), X_poly], axis=1)

# Map X_poly_test and normalize (using mu and sigma)
X_poly_test = polyFeatures(Xtest, p)
X_poly_test -= mu
X_poly_test /= sigma
X_poly_test = np.concatenate([np.ones((ytest.size, 1)), X_poly_test], axis=1)

# Map X_poly_val and normalize (using mu and sigma)
X_poly_val = polyFeatures(Xval, p)
X_poly_val -= mu
X_poly_val /= sigma
X_poly_val = np.concatenate([np.ones((yval.size, 1)), X_poly_val], axis=1)

print('Normalized Training Example 1:')
X_poly[0, :]
```

```
Normalized Training Example 1:
```

Out[10]:
```
array([ 1.        , -0.36214078, -0.75508669,  0.18222588, -0.70618991,
        0.30661792, -0.59087767,  0.3445158 , -0.50848117])
```

3.1 Learning Polynomial Regression

After you have completed the function `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 first call `featureNormalize` and normalize the features of the training set, storing the mu, sigma parameters separately. We have already implemented this function for you (in `utils.py` module) and it is the same function from the first exercise.

After learning the parameters θ, you should see two plots generated for polynomial regression with $\lambda = 0$, which should be similar to the ones here:

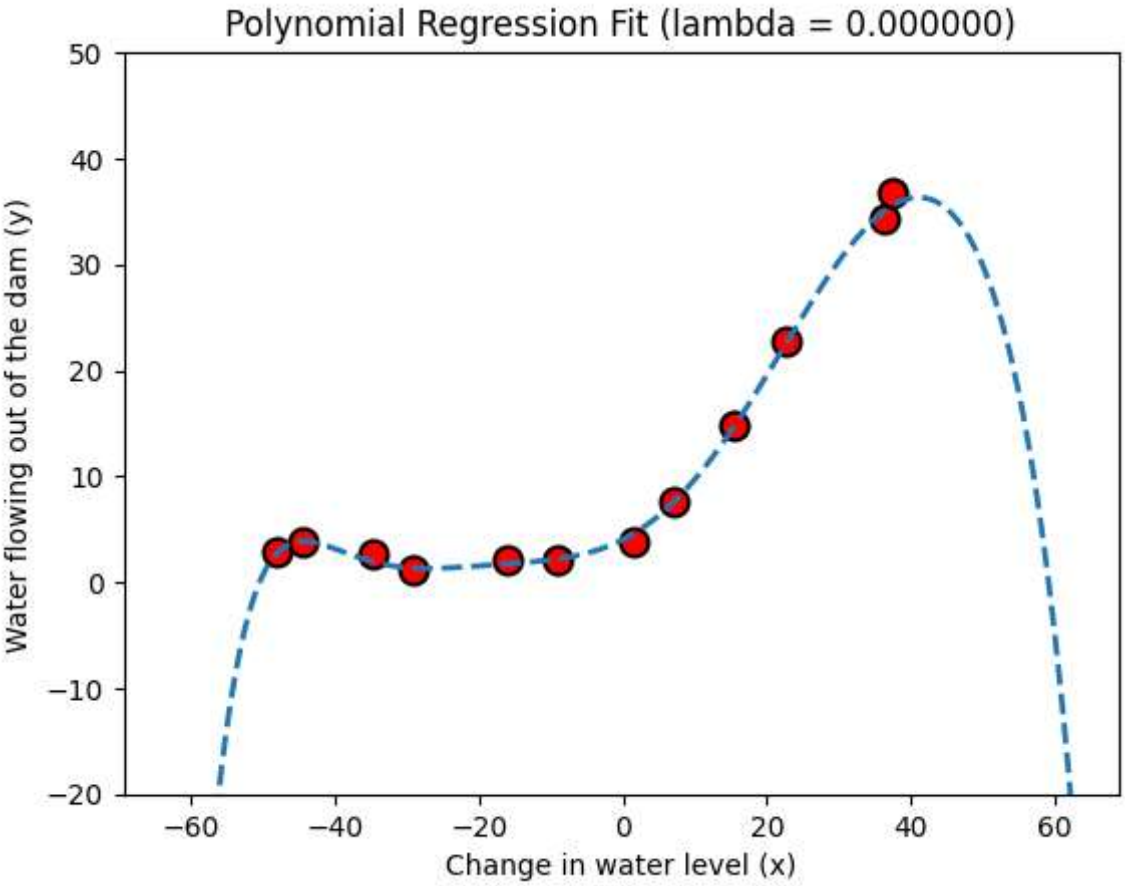
You should see that the polynomial fit is able to follow the datapoints very well, thus, obtaining a low training error. The figure on the right shows that the training error essentially stays zero for all numbers of training samples. However, the polynomial fit is very

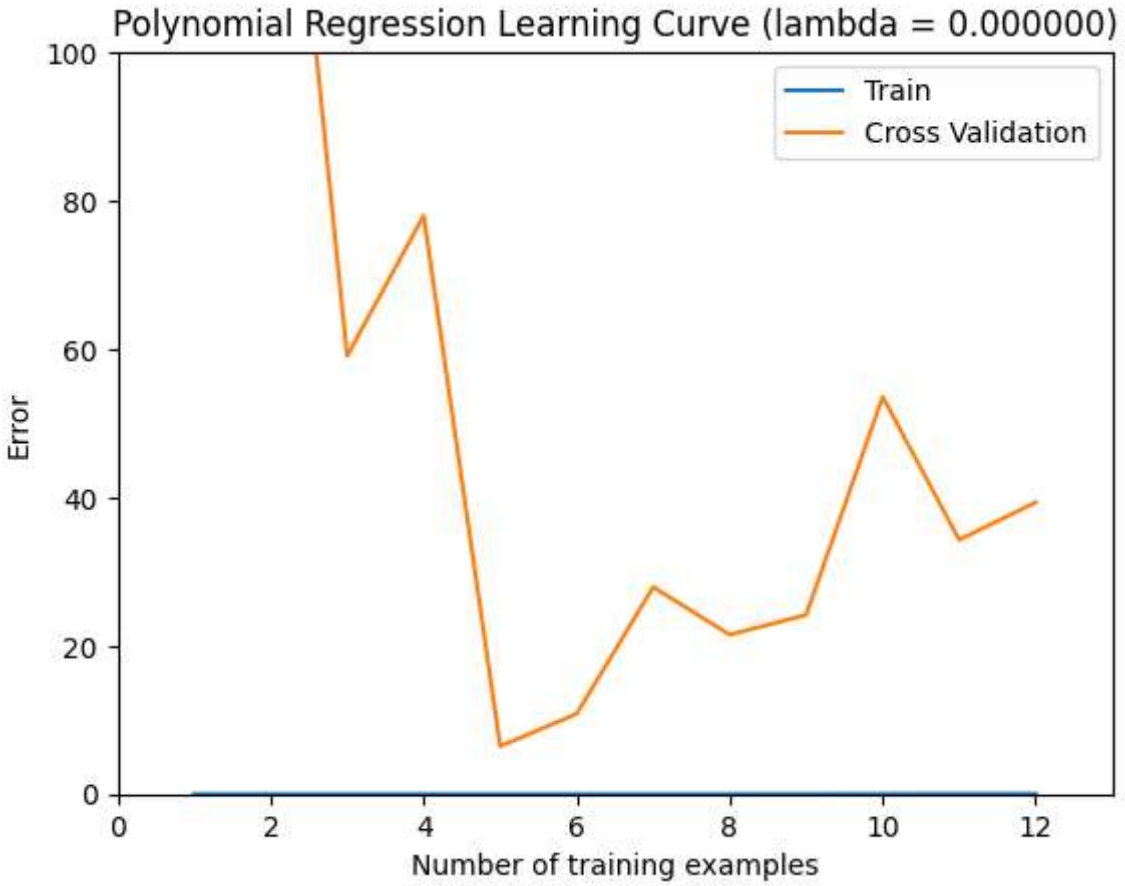
complex and even drops off at the extremes. This is an indicator that the polynomial regression model is overfitting the training data and will not generalize well.

To better understand the problems with the unregularized ($\lambda = 0$) model, you can see that the learning curve shows the same effect where the 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.

In [11]:
```
lambda_ = 0
theta = utils.trainLinearReg(linearRegCostFunction, X_poly, y,
                             lambda_=lambda_, maxiter=55)

# Plot training data and fit
pyplot.plot(X, y, 'ro', ms=10, mew=1.5, mec='k')

utils.plotFit(polyFeatures, np.min(X), np.max(X), mu, sigma, theta, p)

pyplot.xlabel('Change in water level (x)')
pyplot.ylabel('Water flowing out of the dam (y)')
pyplot.title('Polynomial Regression Fit (lambda = %f)' % lambda_)
pyplot.ylim([-20, 50])

pyplot.figure()
error_train, error_val = learningCurve(X_poly, y, X_poly_val, yval, lambda_)
pyplot.plot(np.arange(1, 1+m), error_train, np.arange(1, 1+m), error_val)

pyplot.title('Polynomial Regression Learning Curve (lambda = %f)' % lambda_)
pyplot.xlabel('Number of training examples')
pyplot.ylabel('Error')
pyplot.axis([0, 13, 0, 100])
pyplot.legend(['Train', 'Cross Validation'])

print('Polynomial Regression (lambda = %f)\n' % lambda_)
print('# Training Examples\tTrain Error\tCross Validation Error')
for i in range(m):
    print('  \t%d\t\t%f\t%f' % (i+1, error_train[i], error_val[i]))
```

```
Polynomial Regression (lambda = 0.000000)

# Training Examples     Train Error     Cross Validation Error
        1               0.000000        160.721900
        2               0.000000        160.121511
        3               0.000000        59.071635
        4               0.000000        77.998006
        5               0.000000        6.448210
        6               0.000000        10.827583
        7               0.000000        27.915831
        8               0.000000        21.489777
        9               0.008623        24.168614
        10              0.021190        53.552222
        11              0.034803        34.287211
        12              0.030577        39.338563
```





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.

3.2 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 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 $\lambda = 1$, the generated plots should look like the figure below. You should see a polynomial fit that follows the data trend well (left) and a learning curve (right) 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 high-bias 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 below) 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.

You do not need to submit any solutions for this exercise.

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

Your task is to complete the code in the function `validationCurve`. Specifically, you should should use the `utils.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}.

In [12]:
```
def validationCurve(X, y, Xval, yval):
    """
    Generate the train and validation errors needed to plot a validation
    curve that we can use to select lambda_.

    Parameters
    ----------
    X : array_like
        The training dataset. Matrix with shape (m x n) where m is the
        total number of training examples, and n is the number of features
        including any polynomial features.
```

```
    y : array_like
        The functions values at each training datapoint. A vector of
        shape (m, ).

    Xval : array_like
        The validation dataset. Matrix with shape (m_val x n) where m is the
        total number of validation examples, and n is the number of features
        including any polynomial features.

    yval : array_like
        The functions values at each validation datapoint. A vector of
        shape (m_val, ).

    Returns
    -------
    lambda_vec : list
        The values of the regularization parameters which were used in
        cross validation.

    error_train : list
        The training error computed at each value for the regularization
        parameter.

    error_val : list
        The validation error computed at each value for the regularization
        parameter.

    Instructions
    ------------
    Fill in this function to return training errors in `error_train` and
    the validation errors in `error_val`. The vector `lambda_vec` contains
    the different lambda parameters to use for each calculation of the
    errors, i.e, `error_train[i]`, and `error_val[i]` should give you the
    errors obtained after training with `lambda_ = lambda_vec[i]`.

    Note
    ----
    You can loop over lambda_vec with the following:

          for i in range(len(lambda_vec))
              lambda = lambda_vec[i]
              # Compute train / val errors when training linear
              # regression with regularization parameter lambda_
              # You should store the result in error_train[i]
              # and error_val[i]
              ....
    """
    # Selected values of lambda (you should not change this)
    lambda_vec = [0, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, 3, 10]

    # You need to return these variables correctly.
    error_train = np.zeros(len(lambda_vec))
    error_val = np.zeros(len(lambda_vec))

    # ====================== YOUR CODE HERE ======================

    for i in range(len(lambda_vec)):
        lambda_try = lambda_vec[i]
        theta_t = utils.trainLinearReg(linearRegCostFunction, X, y, lambda_ = lambd
        error_train[i], _ = linearRegCostFunction(X, y, theta_t, lambda_ = 0)
```

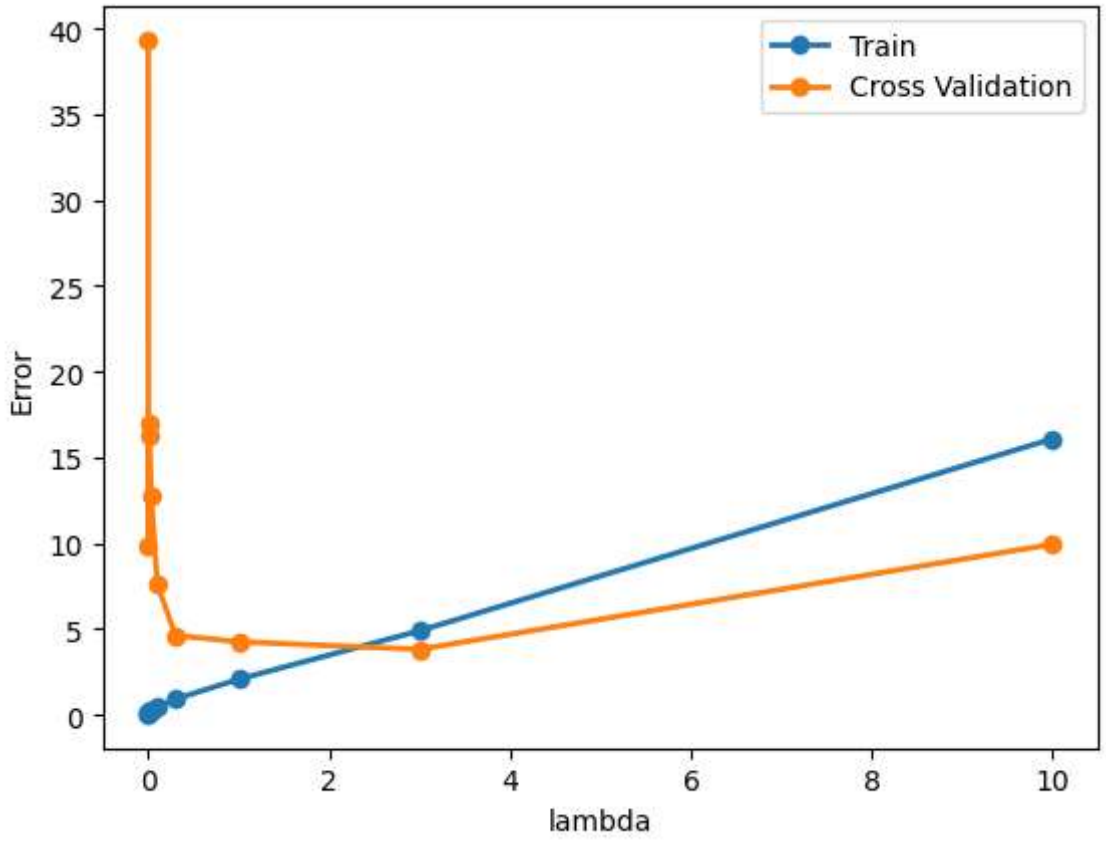
```
        error_val[i], _ = linearRegCostFunction(Xval, yval, theta_t, lambda_ = 0)

    # ============================================================
    return lambda_vec, error_train, error_val
```

After you have completed the code, the next cell will run your function and 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 the figure below.

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.

In [13]:
```
lambda_vec, error_train, error_val = validationCurve(X_poly, y, X_poly_val, yval)

pyplot.plot(lambda_vec, error_train, '-o', lambda_vec, error_val, '-o', lw=2)
pyplot.legend(['Train', 'Cross Validation'])
pyplot.xlabel('lambda')
pyplot.ylabel('Error')

print('lambda\t\tTrain Error\tValidation Error')
for i in range(len(lambda_vec)):
    print(' %f\t%f\t%f' % (lambda_vec[i], error_train[i], error_val[i]))
```

```
lambda          Train Error     Validation Error
 0.000000       0.030577        39.338563
 0.001000       0.112736        9.869358
 0.003000       0.170922        16.306332
 0.010000       0.221506        16.944348
 0.030000       0.281854        12.829266
 0.100000       0.459324        7.586737
 0.300000       0.921776        4.636765
 1.000000       2.076201        4.260599
 3.000000       4.901376        3.822907
 10.000000      16.092273       9.945554
```



3.4 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 $\lambda = 3$.

You do not need to submit any solutions for this optional (ungraded) exercise.

In [ ]:

3.5 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.

For this optional (ungraded) exercise, you should implement the above strategy for computing the learning curves. For reference, the figure below shows the learning curve we obtained for polynomial regression with $\lambda = 0.01$. Your figure may differ slightly due to the random selection of examples.

You do not need to submit any solutions for this optional (ungraded) exercise.

In [ ]: