Statistical Machine Learning	**Assignment 1**
comp540 Spring 2019	**due 19 January 2019 at 8 pm**

This homework is due January 19 at 8 pm on Canvas. The code base `hw1.zip` *for the assignment is an attachment to* Assignment 1 *on Canvas. You will add your code at the indicated spots in the files there. Place your answers to the written portions of Problems 0, 1 and 2 (typeset) in a file called* `writeup.pdf`*; the code in* `sampler.py`*. Code for Problem 3 needs to be entered at the marked points in IPython notebooks in the folders* `part1` *and* `part2`*. You will solve this problem set in a group of two – only one submission per group, please. Upload the entire zip archive back to Canvas before the due date and time.*

We take the Rice Honor Code very seriously in this class. Please read the academic integrity section of the course policies at `https://canvas.rice.edu/courses/20448/pages/comp-540-statistical-machine-learning-course-policies`*, and enter the statement that you have followed the honor code described there in the submission box for your homework. Homeworks without the honor pledge will not be graded.*

Problem 0: Background refresher (25 points)

The purpose of this problem is give you a quick refresher of the mathematical topics we will need for the term. It also gives you a chance to write some Python code.

- (8 points) Write functions in Python to produce samples from four distributions: categorical, univariate Gaussian, multivariate Gaussian, and general mixture distributions. While there are sampling functions implemented in `scipy.stats` for these four distributions, you will write your own implementations using samples from the uniform distribution over the unit interval (`numpy.random.uniform()`). You may find the following Wikipedia pages helpful in designing your sampling algorithms.

  - `https://en.wikipedia.org/wiki/Categorical_distribution`
  - `https://en.wikipedia.org/wiki/Normal_distribution`
  - `https://en.wikipedia.org/wiki/Multivariate_normal_distribution`
  - `https://en.wikipedia.org/wiki/Mixture_distribution`

  Write your implementations in the file `sampler.py`. You may add arguments to the function signatures as long as your implementation supports the interface as specified in the documentation of the functions. Save and submit the completed `sampler.py`. Include the graphs below in `writeup.pdf` - use `matplotlib` for all these plots.

  - Plot the histogram of samples generated by a categorical distribution with probabilities [0.1,0.5,0.2,0.2]
  - Plot the univariate normal distribution with mean of 5 and standard deviation of 2.
  - Produce a scatter plot of the samples for a 2-D Gaussian with mean at [2,3] and a covariance matrix [[1,0.5],[0.2,1]].

  - Test your mixture sampling code by writing a function that implements an equal-weighted mixture of four Gaussians in 2 dimensions, centered at $(\pm 1, \pm 1)$ and having covariance I. Estimate the probability that a sample from this distribution lies within the unit circle centered at $(0.1, 0.2)$ and include that number in your writeup.

- (2 points) Prove that the sum of two independent Poisson random variables is also a Poisson random variable.

- (2 points) Let X_0 and X_1 be continuous random variables. Show that if

$$\begin{aligned} p(X_0 = x_0) &= \alpha_0 e^{-\frac{(x_0-\mu_0)^2}{2{\sigma_0}^2}} \\ P(X_1 = x_1 | X_0 = x_0) &= \alpha e^{-\frac{(x_1-x_0)^2}{2\sigma^2}} \end{aligned}$$

  there exists α_1, μ_1 and σ_1 such that

$$p(X_1 = x_1) = \alpha_1 e^{-\frac{(x_1-\mu_1)^2}{2{\sigma_1}^2}}$$

  Write down expressions for these quantities in terms of α_0, α, μ_0, σ_0 and σ.

- (2 points) Find the eigenvalues and eigenvectors of the following 2×2 matrix A.

$$A = \begin{pmatrix} 0 & 1 \\ -2 & -3 \end{pmatrix}$$

- (2 points) Provide one example for each of the following cases, where A and B are 2×2 matrices.

  - $(A + B)^2 \neq A^2 + 2AB + B^2$
  - $AB = 0$, $A \neq 0$, $B \neq 0$

- (2 points) Let u denote a real vector normalized to unit length. That is, $u^T u = 1$. Show that

$$A = I - 2uu^T$$

  is orthogonal, i.e., $A^T A = 1$.

- (4 points) A function f is convex on a given set S if and only if for $\lambda \in [0, 1]$ and for all $x, y \in S$, the following holds.

$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda) f(y)$$

  Moreover, a univariate functoon $f(x)$ is convex on a set S if and only if its second derivative $f''(x)$ is non-negative everywhere in the set. prove the following assertions.

  - $f(x) = e^x$ is convex for $x \in \Re$.
  - $f(x_1, x_2) = max(x_1, x_2)$ is convex on $\Re^2$.
  - If univariate functions f and g are convex on S, then $max(f, g)$ is convex on S.

  - If univariate functions f and g are convex and non-negative on S, and have their minimum within S at the same point, then fg is convex on S.

- (3 points) The entropy of a categorical distribution on K values is defined as

$$H(p) = -\sum_{i=1}^{K} p_i log(p_i)$$

  Using the method of Lagrange multipliers, find the categorical distribution that has the highest entropy.

Problem 1: Locally weighted linear regression (20 points)

Consider a linear regression problem in which we want to weight different training examples differently. Specifically, suppose we want to minimize

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{m} w^{(i)}\left(\theta^T x^{(i)} - y^{(i)}\right)^2$$

In class, we considered a special case where all the examples were equally weighted. Here you will extend those calculations to the weighted setting.

- (5 points) Show that $J(\theta)$ can be written in the form

$$J(\theta) = (X\theta - y)^T W (X\theta - y)$$

  for an appropriate diagonal matrix W, where X is the $m \times d$ input matrix and y is a $m \times 1$ vector denoting the associated outputs. State clearly what W is.

- (10 points) If all the $w^{(i)}$'s are equal to 1, the normal equation to solve for the parameter θ is:

$$X^T X\theta = X^T y$$

  and the value of θ that minimizes $J(\theta)$ is $(X^T X)^{-1} X^T y$. By computing the derivative of the weighted $J(\theta)$ and setting it equal to zero, generalize the normal equation to the weighted setting and solve for θ in closed form in terms of W, X and y.

- (5 points) To predict the target value for an input vector x, one choice for the weighting function $w^{(i)}$ is:

$$w^{(i)} = exp\left(-\frac{(x - x^{(i)})^T (x - x^{(i)})}{2\tau^2}\right)$$

  Points near x are weighted more heavily than points far away from x. The parameter τ is a bandwidth defining the sphere of influence around x. Note how the weights are defined by the input x. Write down an algorithm for calculating θ by batch gradient descent for locally weighted linear regression. Is locally weighted linear regression a parametric or a non-parametric method?

Problem 2: Properties of the linear regression estimator (10 points)

An estimator of an unknown parameter is called *unbiased* if its expected value equals the true value of the parameter. Here, you will prove that the least-squares estimate given by the normal equation for linear regression is an unbiased estimate of the true parameter θ^*. We first assume that the data $\mathcal{D} = \{(x^{(i)}, y^{(i)} | 1 \leq i \leq m; x^{(i)} \in \Re^d; y^{(i)} \in \Re\}$ comes from a linear model:

$$y^{(i)} = \theta^T x^{(i)} + \epsilon^{(i)}$$

where each $\epsilon^{(i)}$ is an independent random variable drawn from a normal distribution with zero mean and variance σ^2. When considering the bias of an estimator, we treat the input $x^{(i)}$'s as fixed but arbitrary, and the true parameter vector θ^* as fixed but unknown. Expectations are taken over possible realizations of the output values $y^{(i)}$'s.

- (5 points) Show that $E[\theta] = \theta^*$ for the least squares estimator.
- (5 points) Show that the variance of the least squares estimator is $Var(\theta) = (X^T X)^{-1} \sigma^2$.

Problem 3: Implementing linear regression and regularized linear regression (80 points)

Introduction

In this two-part problem, you will first implement linear regression and get experience working with it on a classical data set for predicting median home values at the census tract level in the Boston suburbs. Initially you will explore linear regression on one variable, and then you will extend your work to cover multiple variables. In the second part, you will implement regularized linear regression and use it to study models with different bias-variance properties. Unzip the code archive `hw1.zip`, and you will see two folders `part1` and `part2`. The material for the first part of the assignment is in `part1` and the files in `part1` are shown in Table 1. The material for the second part of the assignment is in `part2` and the files in this folder are shown in Table 2. Complete the problems in part 1 before moving on to part 2.

Setting up Python

I highly recommend the use of Anaconda Navigator. Anaconda is free for academics and contains all the Python modules we need for all the assignments in this class. For Anaconda, go to

`https://www.continuum.io/downloads`

and download the Python 3 version of Anaconda appropriate for your OS. Anaconda comes with most of the required packages for the class, and package management is made easy in the framework.

Name	Edit?	Read?	Description
linear_regressor.py	Yes	Yes	Fill in the loss function, prediction function, gradient descent algorithm (linear regression with one variable).
linear_regressor_multi.py	Yes	Yes	Fill in the loss function, prediction function, gradient descent algorithm and normal equation (linear regression with multiple variables)
utils.py	Yes	Yes	Fill in function to normalize features
ex1.ipynb	Yes	Yes	Python notebook that will run your functions for linear regression with a single variable (edit only where indicated)
ex1_multi.ipynb	Yes	Yes	Python notebook that will run your functions for linear regression with multiple variables (edit only where indicated)
housing.data.txt	No	Yes	The Boston housing data set
housing.data.names	No	Yes	Description of the variables in the Boston housing data set
plot_utils.py	No	Yes	Functions for data and model visualization

Table 1: Files in folder `part1` for the first part of the assignment.

Problem 3.1: Implementing linear regression (45 points)

Problem 3.1.A: Linear regression with one variable (15 points)

You will implement linear regression with one variable to predict the median value of a home in a census tract in the Boston suburbs from the percentage of the population in the census tract that is of lower economic status. The file `housing.data.txt` contains the data for our linear regression problem. We will build a model that predicts the median home value in a census tract (in $10000s) from the percentage of the population of lower economic status in a tract. The `ex1.ipynb` notebook has already been set up to load this data for you using the Python package `pandas`.

Plotting the data

Before starting on any task, it is often useful to understand the data by visualizing it. For this dataset, you can use a scatter plot to visualize the data, since it has only two features to plot (percentage of population of lower economic status and median home value). Many other problems that you will encounter in real life are multi-dimensional and cannot be plotted on a 2-d plot. In `ex1.ipynb`, the dataset is loaded from the data file into the variables `X` and `y`. You will see the plot in Figure 1 generated by `plot_utils.py` displayed using `plt.show()` in the script. You can save the plot in `fig1.pdf` in the `part1` folder using the `savefig` method of the plot function.

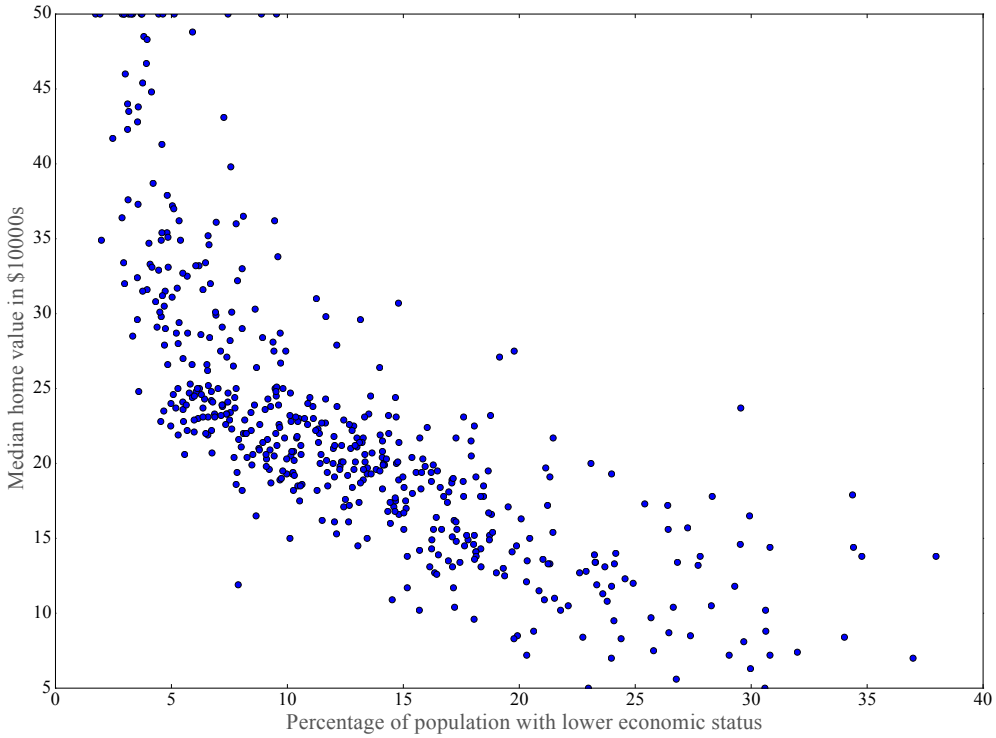

Figure 1: Scatter plot of training data

Gradient descent

You will implement gradient descent to fit the parameter θ to the housing data. Recall that the objective of linear regression is to minimize the cost function

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

where the hypothesis $h_\theta(x)$ is given by the linear model

$$h_\theta(x) = \theta^T x = \theta_0 + \theta_1 x$$

The gradient descent algorithm computes the value of θ to minimize the cost function $J(\theta)$. We will use batch gradient descent which performs the following update on each iteration.

$$\theta_j \leftarrow \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j{}^{(i)}$$

where all the θ_j's are updated simultaneously. The parameter α is the learning rate. With each step of gradient descent, the parameters θ_j come closer to the optimal values that achieve the

lowest cost $J(\theta)$. In `ex1.ipynb`, we have already set the learning rate (0.005), number of iterations (10000) as well as the initial values for the parameter θ (all zeros).

We store each example as a row in a `numpy` array `X`. To take into account the intercept term (θ_0), we add an additional first column to `X` and set it to all ones. This allows us to treat θ_0 as simply another 'feature'.

Problem 3.1.A1: Computing the cost function $J(\theta)$ (5 points)

As you perform gradient descent to minimize the cost function $J(\theta)$, it is helpful to monitor the convergence by computing the cost. You will implement a function to calculate $J(\theta)$ so you can check the convergence of your gradient descent implementation. Your should complete the code for the `loss` method for the `LinearReg_SquaredLoss` class in the file `linear_regressor.py`. Remember that the variables `X` and `y` are not scalar values, but matrices whose rows represent the examples from the training set.

Problem 3.1.A2: Implementing gradient descent (5 points)

Next, you will implement gradient descent in the method `train` for the `LinearRegressor` class in the file `linear_regressor.py`. The loop structure has been written for you, and you only need to supply the updates to θ within each iteration. Recall that we minimize the value of $J(\theta)$ by changing the values of the vector θ, not by changing `X` or `y`. A good way to verify that gradient descent is working correctly is to look at the value of $J(\theta)$ and check that it is decreasing with each step. The `train` method calls the `loss` method on every iteration and prints the cost. Assuming you have implemented gradient descent and and the loss function correctly, your value of $J(\theta)$ should never increase, and should converge to a steady value by the end of the algorithm. You should expect to see a cost of approximately 296.07 at the first iteration. After you are finished, `ex1.ipynb` will use your final parameters to plot the linear fit. The result should look something like Figure 2. The plot of the $J(\theta)$ values during gradient descent is in Figure 3.

Visualizing $J(\theta)$

To understand the cost function $J(\theta)$ better, we plot the cost over a 2-dimensional grid of θ_0 and θ_1 values. You will not need to code anything new for this part, but you should understand how the code you have written already is creating these images. In `ex1.ipynb`, we calculate $J(\theta)$ over a grid of (θ_0, θ_1) values using the `loss` method that you wrote. The 2-D array of $J(\theta)$ values is plotted using the surf and contour commands of `matplotlib`. The plots should look something like Figure 4.

The purpose of these plots is to show you how $J(\theta)$ varies with changes in θ_0 and θ_1. The cost function is bowl-shaped and has a global minimum. This is easier to see in the contour plot than in the 3D surface plot. This minimum is the optimal point for θ_0 and θ_1, and each step of gradient descent moves closer to this point.

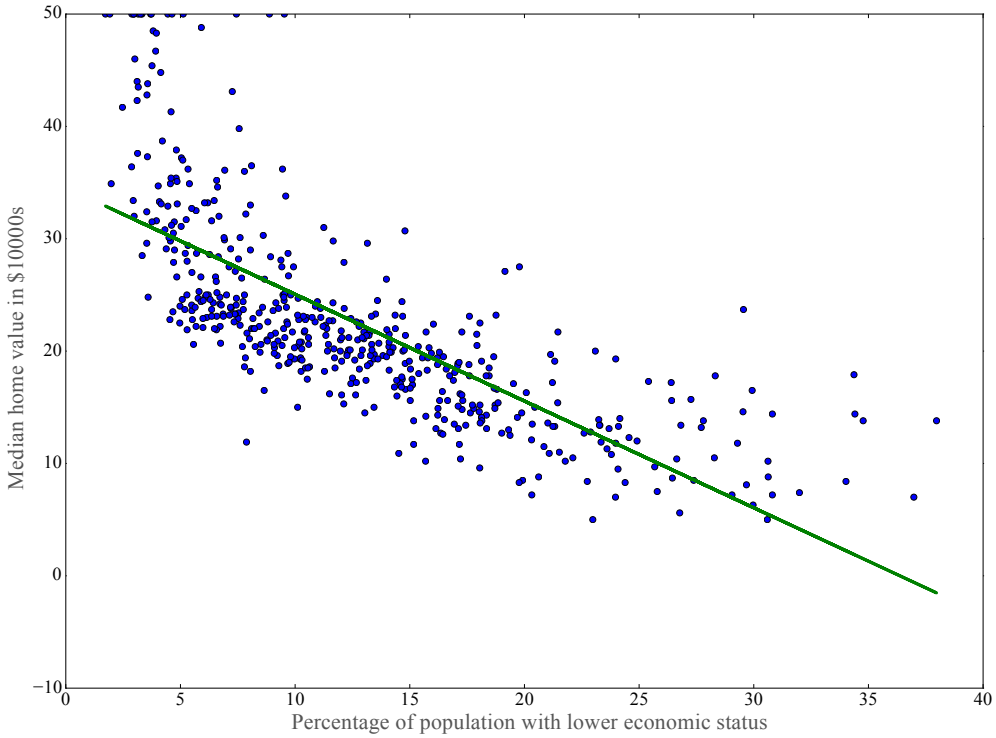

Figure 2: Fitting a linear model to the data in Figure 1

Problem 3.1.A3: Predicting on unseen data (5 points)

Your final values for θ will also be used to make predictions on median home values for census tracts where the percentage of the population of lower economic status is 5% and 50%. First, complete the `predict` method in the `LinearRegressor` class in `linear_regression.py`. Then fill in code for prediction using the computed θ at the indicated point in `ex1.ipynb`. Report the predictions of your model in your lab writeup.

Assessing model quality

Up to now, we have used all the given data to estimate the model. Then, we check its performance on two unseen points. To assess model quality in a more systematic fashion, we will explore train/test splits and crossvalidation approaches. We will use the mean squared error and the R^2 value as measures of model quality. The lower the mean squared error, the better the model is. The closer R^2 is to 1, the better the model is. The `ex1.ipynb` notebook shows you how to split `X` into training and test sets and use them for model estimation and model evaluation respectively. The notebook also shows you how to set up k-fold crossvalidation assessment of a linear model using built-in functions in the `sklearn` package.

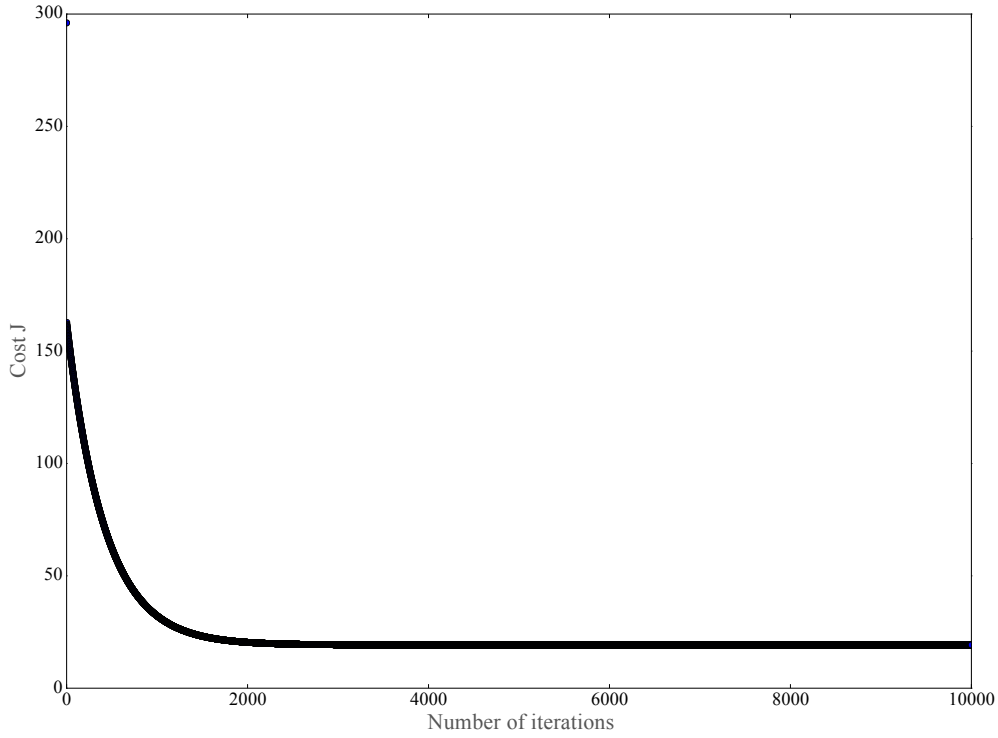

Figure 3: Convergence of gradient descent to fit the linear model in Figure 2

Problem 3.1.B: Linear regression with multiple variables (30 points)

Problem 3.1.B1: Feature normalization (5 points)

For this problem we will work with `ex1_multi.ipyb`, `linear_regressor_multi.py` and `utils.py`. The `ex1_multi.ipyb` notebook will start by loading and displaying some values from the full Boston housing dataset with thirteen features of census tracts that are believed to be predictive of the median home price in the tract (see `housing.names.txt` for a full description of these features). By looking at the values, you will note that the values of some of the features are about 1000 times the values of others. When features differ by orders of magnitude, feature scaling becomes important to make gradient descent converge quickly. Your task here is to complete the code in `feature_normalize.py` in `utils.py`. First, subtract the mean value of each feature from the dataset. Second, divide the feature values by their respective standard deviations. The standard deviation is a way of measuring how much variation there is in the range of values of a particular feature (most data points will lie within two standard deviations of the mean). You will do this for all the features and your code should work with datasets of all sizes (any number of features / examples). Note that each column of the matrix `X` corresponds to one feature. When normalizing the features, it is important to store the values used for normalization - the mean value and the standard deviation used for the computations. After learning the parameter θ, we want to predict the median home prices for new census tracts. Given the thirteen characteristics `x` of a new census

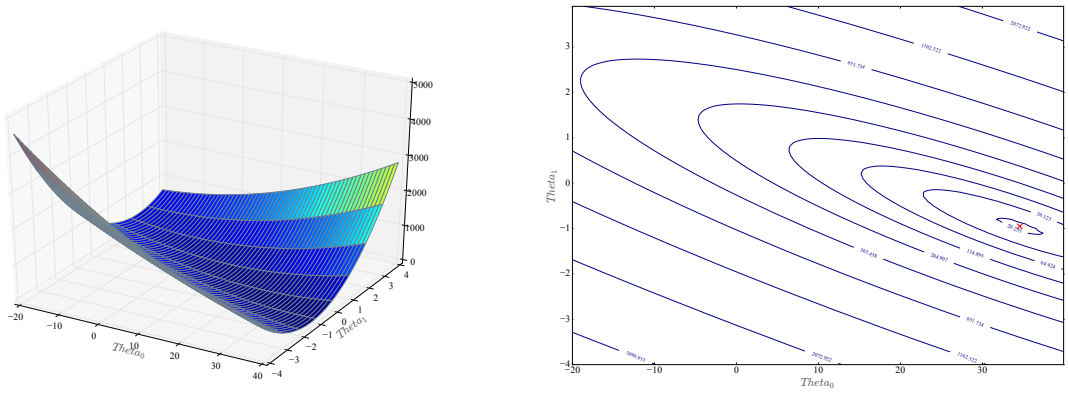

Figure 4: Surface and contour plot of cost function J (linear regression with single variable)

tract, we must first normalize `x` using the mean and standard deviations that we had previously computed from the training set. Then, we take the dot product of `x` (with a 1 prepended (to account for the intercept term) with the parameter vector θ to make a prediction.

Note that in `ex1_multi.ipynb`, when `feature_normalize` is called, the extra column of 1's has not yet been added to `X`.

Problem 3.1.B2: Loss function and gradient descent (10 points)

Previously, you implemented gradient descent on a univariate regression problem. The only difference now is that there are more features in the matrix `X`. The hypothesis function and the batch gradient descent update rule remain unchanged. You should complete the code for the `train` method and the `loss` method at the indicated points in `linear_regressor_multi.py` to implement the cost function and gradient descent for linear regression with multiple variables. Make sure your code supports any number of features and that it is **vectorized**. I recommend the use of `numpy`'s code vectorization facilities.

You should see the cost $J(\theta)$ converge as shown in Figure 5.

Problem 3.1.B3: Making predictions on unseen data (5 points)

Your final parameter values for θ will now be used to make predictions on median home values for an average census tract, characterized by average values for all the thirteen features. Complete the calculation in `ex1_multi.ipynb` in the indicated lines in that file. Now run the corresponding cell in `ex1_multi.ipynb` to see what the prediction of median home value for an *average* tract is. *Remember to scale the features correctly for this prediction.*

Problem 3.1.B4: Normal equations (5 points)

The closed form solution for θ is

$$\theta = (X^T X)^{-1} X^T y$$

Using this formula does not require any feature scaling, and you will get an exact solution in one calculation: there is no loop until convergence as in gradient descent. Complete the code in the method `normal_eqn` in `linear_regressor_multi.py` to use the formula above to calculate θ. Now make a prediction for the average census tract (same example as in the previous problem). Do the predictions match up? Remember that while you do not need to scale your features, you still need to add a 1 to the example to have an intercept term (θ_0).

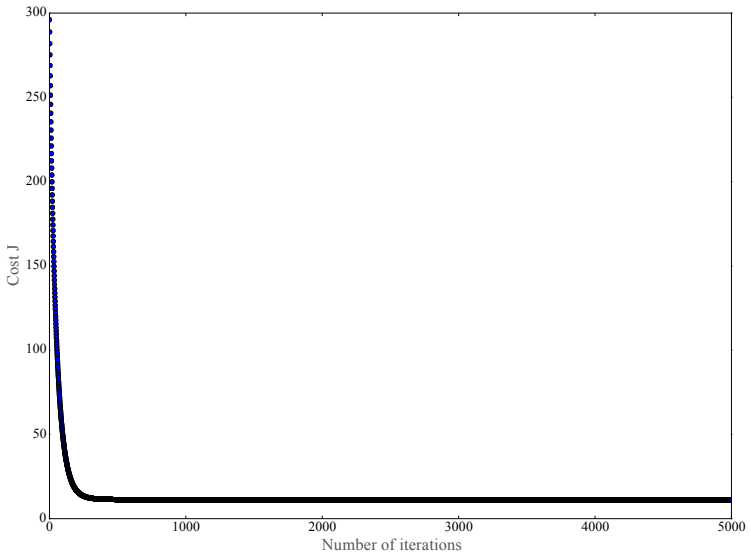

Figure 5: Convergence of gradient descent for linear regression with multiple variables (Boston housing data set)

Problem 3.1.B5: Exploring convergence of gradient descent (5 points)

In this part of the exercise, you will get to try out different learning rates for the dataset and find a learning rate that converges quickly. You can change the learning rate and the number of iterations by modifying the call to the `LinearReg` constructor in `ex1_multi.ipynb`. The next phase in `ex1_multi.ipynb` will call your `train` function and run gradient descent at the chosen learning rate for the chosen number of iterations. The function should also return the history of $J(\theta)$ values in a vector `J`. After the last iteration, the `ex1_multi.ipynb` script plots the `J` values against the number of the iterations. If you picked a learning rate within a good range, your plot should look similar to Figure 5. If your graph looks very different, especially if your value of $J(\theta)$ increases or even blows up, adjust your learning rate and try again. We recommend trying values of the learning rate α on a log-scale, at multiplicative steps of about 3 times the previous value (i.e., 0.3, 0.1, 0.03, 0.01 and so on). You may also want to adjust the number of iterations you are running if that will help you see the overall trend in the curve. Present plots of J as a function of the number of iterations for different learning rates. What are good learning rates and number of iterations for this problem? Include plots and a brief writeup in your lab report to justify your choices.

Problem 3.2: Implementing regularized linear regression (35 points)

In this part, you will implement regularized linear regression and use it to study models with different bias-variance properties. To get started, look at the code in the folder `part2`, which has the files shown in Table 2.

Name	Edit?	Read?	Description
reg_linear_regressor_multi.py	Yes	Yes	Fill in the regularized loss function, prediction function, gradient descent algorithm (regularized linear regression with multiple variables)
utils.py	Yes	Yes	Fill in function to generate a learning curve, an averaged learning curve, mapping data into polynomial features, and function to generate a cross validation curve
ex2.ipynb	No	Yes	Python notebook for running your functions for regularized linear regression (edit only where indicated)
plot_utils.py	No	Yes	functions to plot a polynomial fit and other plots
ex2data1.mat	No	No	Dataset for this assignment

Table 2: The files in folder `part2` for the second part of the assignment.

Regularized linear regression

In this problem, 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.

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 y, flowing out of the dam. This dataset is divided into three parts:

- A training set that you will use to learn the model: `X`, `y`.
- A validation set for determining the regularization parameter: `Xval`, `yval`.
- A test set for evaluating the performance of your model: `Xtest, ytest`. These are unseen examples that were not used during the training of the model.

Run the notebook `ex2.ipynb` and it will plot the training data as shown in Figure 6.

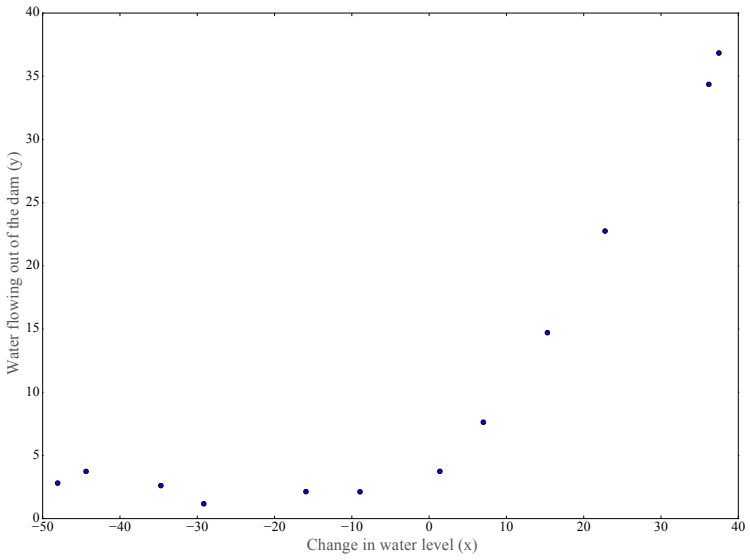

Figure 6: The training data for regularized linear regression

Next you will implement regularized linear regression and use it 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.

Problem 3.2.A1: Regularized linear regression cost function (5 points)

Regularized linear regression has the following cost function:

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

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 $J(\theta)$. 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 for the method `loss` in the class `Reg_LinearRegression_SquaredLoss` in the file `reg_linear_regressor_multi.py` to calculate $J(\theta)$. Vectorize your code and avoid writing for loops.

Problem 3.2.A2: Gradient of the Regularized linear regression cost function (5 points)

Correspondingly, the partial derivative of the regularized linear regression cost function with respect to θ_j is defined as:

$$\frac{\partial J(\theta)}{\partial \theta_0} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}){x_j}^{(i)}$$

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

You should now complete the code for the method `grad_loss` in the class `Reg_LinearRegression_SquaredLoss` in the file `reg_linear_regressor_multi.py` to calculate the gradient, returning it in the variable `grad`.

Learning linear regression model

Once your cost function and gradient are working correctly, the script `ex2.ipynb` will run the `train` method in `reg_linear_regressor_multi.py` to compute the optimal values of θ. This training function uses `scipy`'s `fmin_bfgs` to optimize the cost function. Here we have set the regularization parameter λ to zero. Because we are trying to fit a line on to data that is clearly non-linear, regularization will not be incredibly helpful. In the next problem, you will use polynomial regression and see the impact of regularization.

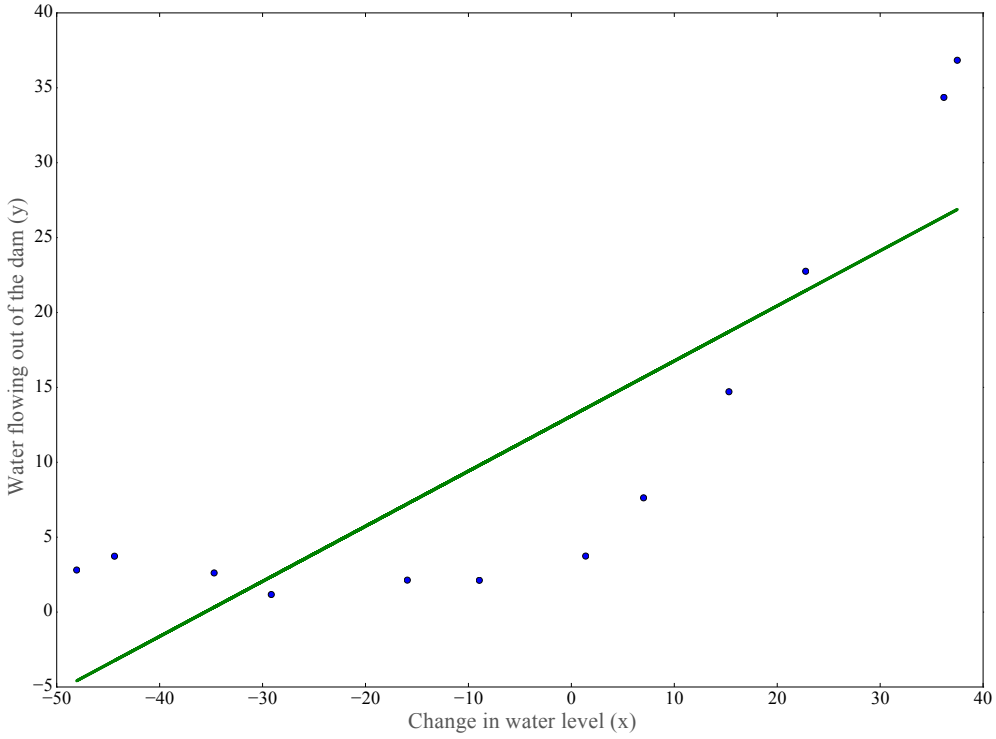

Figure 7: The best fit line for the training data

Finally, the `ex2.ipynb` script plots the best fit line, resulting in a plot like the one shown in Figure 7. The best fit line tells us that the model is not a good fit to the data because the data is non-linear. 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 problem, 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.

Bias and 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 the training data. In this problem, you will plot training and test errors on a learning curve to diagnose bias-variance problems.

Problem 3.2.A3: Learning curves (5 points)

A learning curve plots training and cross validation error as a function of training set size. You will complete the function `learning_curve` in `utils.py` so that it returns a vector of errors for the training set and validation set. To obtain different training set sizes, use different subsets of the original training set `X`. Specifically, for a training set size of i, you should use the first i examples.

You can use the `train` function to find the parameter θ. Note that the regularization λ `reg` is passed as a parameter to the `learning_curve` function. After learning the θ parameter, you should compute the error on the training and validation sets. Recall that the training error for a dataset is defined as:

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

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 `reg` to 0 only when using it to compute the training error and validation error. When you are computing the training set error, make sure you compute it on the training subset instead of the entire training set. However, for the validation error, you should compute it over the entire validation set. You should store the computed errors in the vectors `error_train` and `error_val`. When you are finished, `ex2.ipynb` will print the learning curves and produce a plot similar to Figure 8.

In Figure 8, 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. Next, you will implement polynomial regression to fit a better model for this dataset.

Polynomial regression: expanding the basis functions

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

$$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \ldots + \theta_p x^p$$

This is still a linear model from the point of view of the parameter space. We have augmented the features with powers of x. The `ex2.py` script builds these features using `sklearn`'s preprocessing

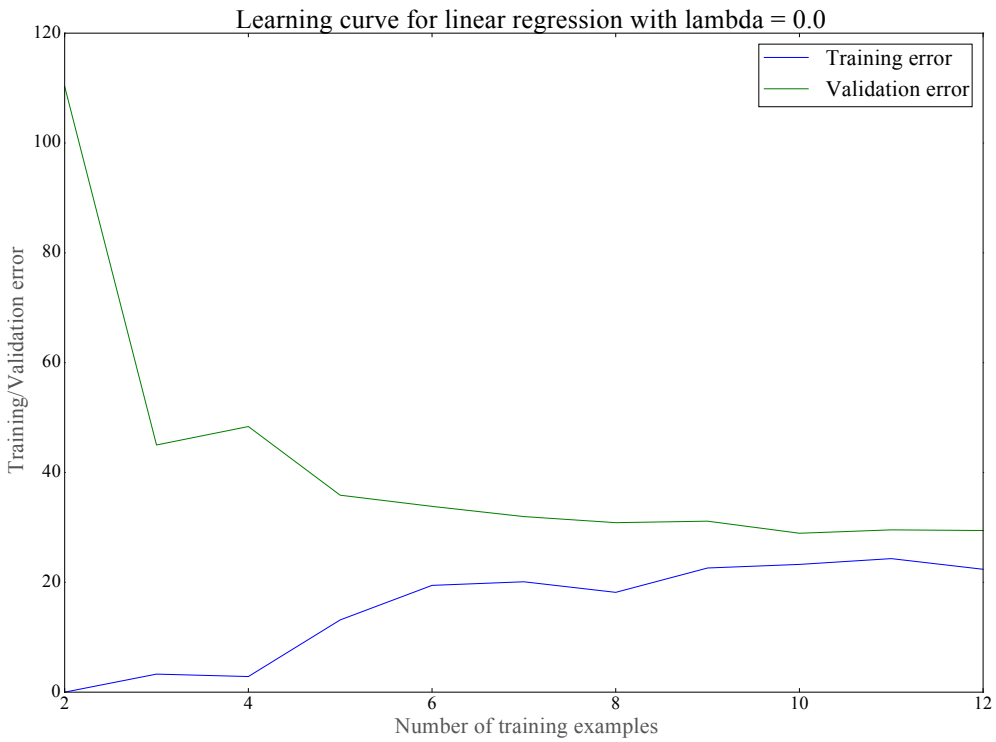

Figure 8: Learning curves

module.

Learning polynomial regression models

The `ex2.ipynb` script will proceed to train polynomial regression using your regularized 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.

You will use a polynomial of degree 6. It turns out that if we run the training directly on the projected data, it will not work well as the features would be badly scaled (e.g., an example with $x = 40$ will now have a feature $x^6 = 40^6 = 4.1 \times 10^9$). Therefore, you will need to use feature normalization. Before learning the parameter θ for the polynomial regression, `ex2.ipynb` will first call the `feature_normalize` function you wrote earlier. It will normalize the features of the training set, storing the `mu`, `sigma` parameters separately. After learning the parameter θ, you should see two plots (Figures 9 and 10) generated for polynomial regression with $\lambda = 0$.

From Figure 9, you should see that the polynomial fit is able to follow the data points 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.

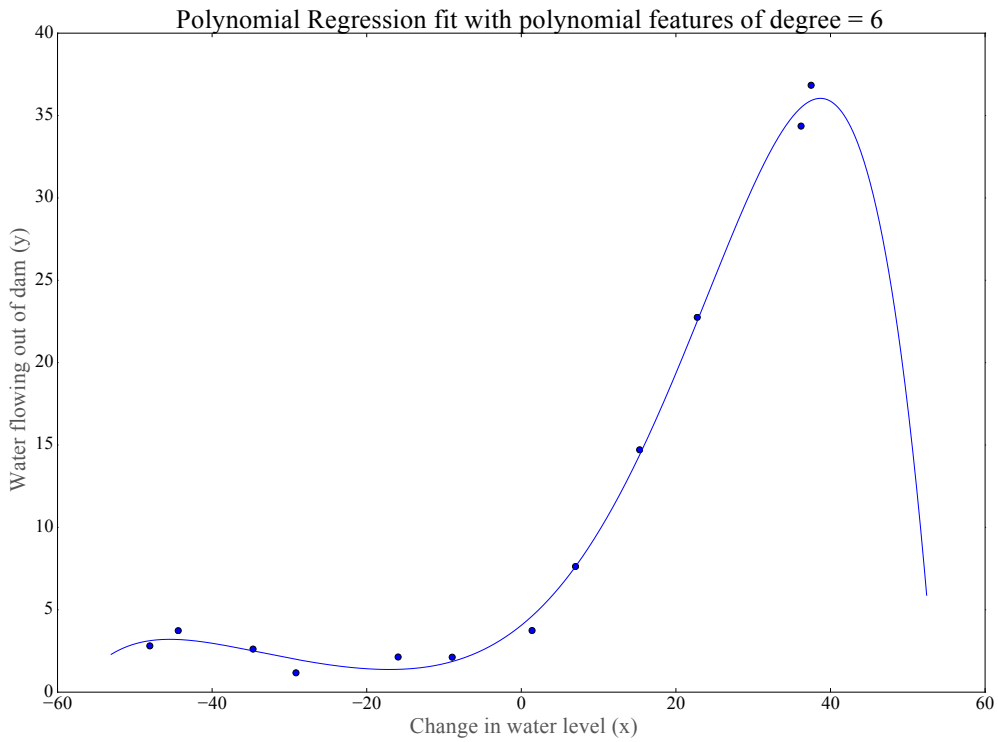

Figure 9: Polynomial fit for `lambda` $= 0$ with a p=6 order model.

To better understand the problems with the unregularized ($\lambda = 0$) model, you can see that the learning curve (Figure 10) shows the same effect where the training error is low, but the error on the validation set is high. There is a gap between the training and validation errors, indicating a high variance problem. One way to combat the overfitting (high-variance) problem is to add regularization to the model. Next, you will get to try different λ values to see how regularization can lead to a better model.

Problem 3.2.A4: Adjusting the regularization parameter (5 points)

You will now explore how the regularization parameter affects the bias-variance of regularized polynomial regression. You should now modify the `lambda` parameter in the script `ex2.ipynb` and try $\lambda = 1, 10, 100$. For each of these values, the script will generate a polynomial fit to the data and also a learning curve. Submit two plots for each value of `lambda`: the fit as well as the learning curve. Comment on the impact of the choice of `lambda` on the quality of the learned model in your lab writeup.

Problem 3.2.A5: Selecting λ using a validation set (5 points)

You have now observed that the value of λ can significantly affect the results of regularized polynomial regression on the training and validation set. In particular, a model without regularization

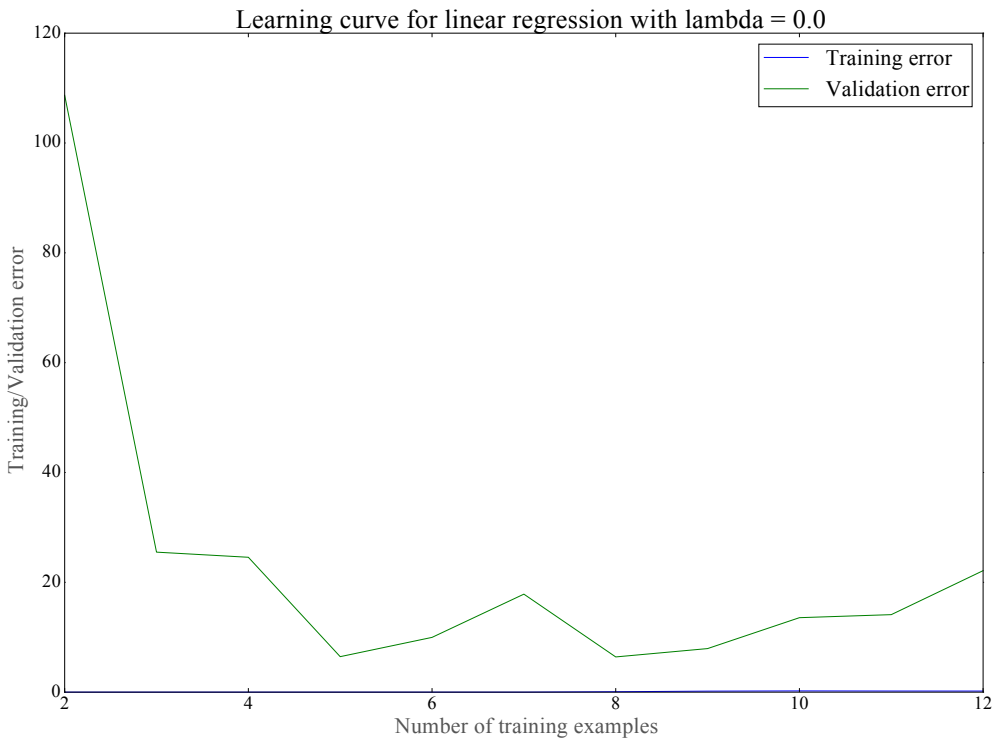

Figure 10: Learning curve for `lambda` $= 0$.

fits the training set well, but does not generalize. Conversely, a model with too much regularization does not fit the training set and testing set well. A good choice of λ can provide a good fit to the data.

You will implement an automated method to select the λ parameter. Concretely, you will use a validation set to evaluate how good each λ value is. After selecting the best λ value using the validation set, we can then evaluate the model on the test set to estimate how well the model will perform on actual unseen data. Complete the function `validation_curve.m` in `utils.py`. Specifically, you should should use the `train` method on an instance of the class `Reg_Linear_Regressor` to train the model using different values of λ and to compute the training error and 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}.

After you have completed the code, run the appropriate cell in `ex2.ipynb` to plot a validation curve of λ versus the error. This plot allows you select which λ value to use. Due to randomness in the training and validation splits of the dataset, the cross validation error can sometimes be lower than the training error. Submit a pdf version of this plot in your report. Comment on the best choice of λ for this problem.

Problem 3.2.A6: Computing test set error (5 points)

To get a better indication of a 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 regularization parameter, nor to learn the model parameters). Calculate the error of the best model that you found with the previous analysis and report it. You can add code at the noted spot in `ex2.ipynb` to print out this value.

Problem 3.2.A7: Plotting learning curves with randomly selected examples (5 points)

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 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 validation set. You will then learn the model parameters using the randomly chosen training set and evaluate the parameters on the randomly chosen training set and 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. Implement the above strategy for computing the learning curves. For reference, Figure 11 shows the learning curve we obtained for polynomial regression with $\lambda = 1$. Your figure may differ slightly due to the random selection of examples. Complete the function `learning_curve_averaged` in `utils.py` to generate compute and generate this plot. Call this function at the end of `ex2.ipynb` and plot the averaged learning curve.

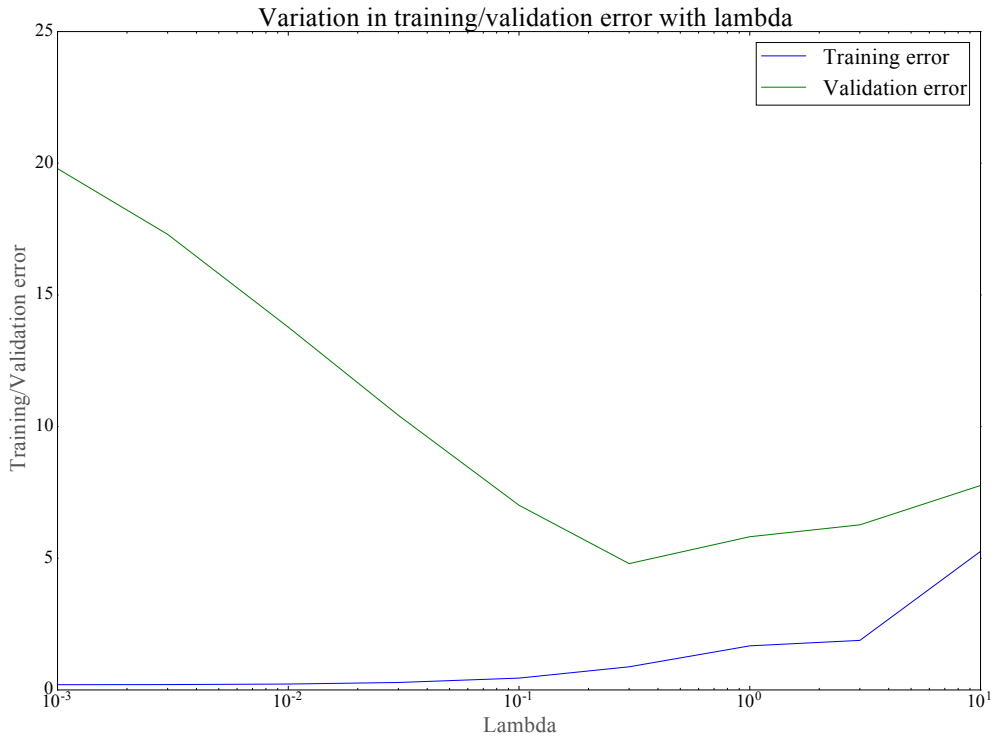

Figure 11: Averaged Learning curve for `lambda` = 1.

What to turn in

Please zip up all the files in the archive (including files that you did not modify) and submit it as `hw1_netid1_netid2.zip` on Canvas before the deadline, where `netid1` and `netid2` are your netid and your partner's netid. Include a PDF file `writeup.pdf` in the archive that presents your plots and discussion of results from the programming component of the assignment. Also include typeset solutions to the written problems 0, 1 and 2 of this assignment in your `writeup.pdf`. Only one submission per group of two, please. Do not forget to type in the honor pledge in the submission box when you upload your archive.

Acknowledgment

Problem 3 is adapted from Andrew Ng's exercise on linear regression.