Department of Electrical and Computer Engineering

ELE 888 / EE 8209 - Intelligent Systems: Winter 2017 - Lab #1
Linear Regression and Regularization

Author: Raymond Phan
Edited By: Md. Moinuddin Bhuiyan

Objective

In this lab assignment, you will implement multivariable linear regression via Gradient Descent and see how it works with real data. You will then make predictions after you train your prediction model with Gradient Descent. Multivariable is an extension of univariate or single variable linear regression used for fitting the line of best fit with training data that uses only one feature. In addition, you will explore training your model by adding regularization to the multivariable formulation and comparing its performance with the original Gradient Descent formulation. Finally, you will explore the practical aspects of Gradient Descent such as feature normalization and determining the best learning rate to allow the algorithm to converge.

List of Files Included

In this lab assignment, you are given some files with code already completed to help you complete this lab assignment. These are primarily for setup so you can concentrate on actually implementing machine learning algorithms rather than being bogged down with minor minutiae. There are additionally other files that are skeletons where it is your task to complete these files or functions and submit them with your lab report.

Demonstration Scripts

These are scripts that help familiarize yourself with the lab and may help you in completing the required portions of this lab.

- `linear_regression`: Script that finds the parameters and the costs of a first-order line of best fit givenexample training data by linear regression through Gradient Descent.
- `plot_line_and_points`: Given a set of points and an expected range of values of these points along with the parameters from univariate linear regression, this script plots both the points and the line of best fit through these points in one figure. This will be useful to serve as a template for the second part of this lab when plotting data.
- `plot_costs`: Plots the linear regression cost function as a function of the total number of iterations. The input into this function is an array of costs returned from linear_regression.
- `create_plots`: This demonstrates how to plot surface and contour plots for the univariate linear regression problem.
- `normalize_features`: This takes in a matrix of examples and returns a new matrix that normalizes each feature of all examples where each feature has zero mean and unit variance. This also returns the mean and standard deviation of each feature of the input matrix of examples for use in normalizing other features. This is useful when training a prediction model by normalizing the features of the training examples first before

finding the model.

- `normalize_input`: This takes in a matrix of examples from the same domain as given in `normalize_features` as well as the mean and standard deviation of each of the features found from `normalize_features` and normalizes this matrix. This is especially useful when predicting the output of new instances to the trained model that used feature normalization as part of the pre-processing step.
- `poly_features`: This takes a data set of a single feature of size `m x 1` (a column vector) and returns a `m x p` matrix where column j of a particular example $x^{(i)}$ seen in row i of this matrix is a new feature such that $(x^{(i)})^j$. There are `p` columns where `p` is the highest order polynomial desired. This is to be used in the second part of the lab.

Dataset Files

These are MATLAB MAT files that contain datasets that will be used for this lab. These are:

- `lab1winedata.dat`: A dataset containing characteristics and the quality of red wine samples obtained north of Portugal from Paulo Cortez, University of Minho, Portgual.
- `lab1poly.dat`: A dataset describing the amount of water flowing out of a dam using the change of water level in a reservoir. This was taken from Andrew Ng's online Machine Learning course through `coursera.org`.

To complete

These are the scripts you need to complete for this lab. Each script will either be a skeleton that you need to complete in functionality or is a function that you need to write with a few lines of code to get you started.

- `multivar_regression`: Implement multivariable linear regression.
- `wine_predict`: This script is to be completed with the use of `multivar_regression` and other functions that are included in this lab to complete the first half of the lab assignment.
- `multivar_regression_reg`: Implement multivariable linear regression with regularization.
- `water_level`: This script is to be completed with the use of `multivar_regression_reg` and other functions that are included in this lab to complete the second half of this assignment.

Warmup

> *This section is strictly for your benefit. There is nothing to write up about in your report here. Please read to get the fullest and rewarding experience in your lab.*

In class, we implemented linear regression via Gradient Descent using only one feature, which is also called the univariate or one-variable version of Gradient Descent. Recalling from the lecture, the goal is to find the best parameters of our prediction model $h_\theta(x) = \theta_0 + \theta_1 x$ where θ_0 - the intercept, and θ_1 - the slope. These parameters will form a line of best fit for fitting the training data. These parameters are found by minimizing the following cost function $J(\theta_0, \theta_1)$.

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

m is the number of training examples in your training data set and each training example forms the pair $(x^{(i)}, y^{(i)})$ where $x^{(i)}$ are the features comprising of training example i and $y^{(i)}$ is the expected output given $x^{(i)}$ for

$i = 1,2, \dots . m$

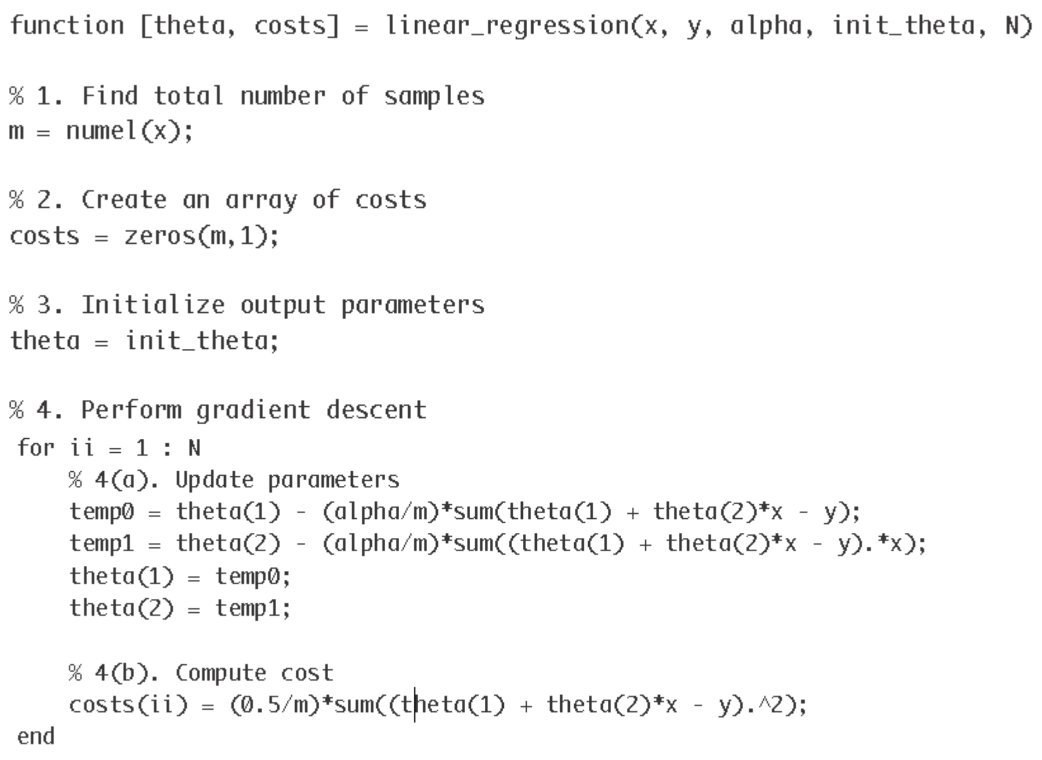
Continuing with the cost function, this was the procedure we developed in class using Gradient Descent for performing linear regression.

1. Start with an initial set of parameters (i.e. θ_0, θ_1)
2. Simultaneously update the parameters and

$$
\begin{aligned}
&\text{repeat until convergence } \{ \\
&\theta_0 \leftarrow \theta_0 - (\alpha/m) \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \\
&\theta_1 \leftarrow \theta_1 - (\alpha/m) \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x^{(i)} \\
&\}
\end{aligned}
$$

3. Prediction model is now of the form $h_\theta(x) = \theta_0 + \theta_1 x$. Use this to predict new values.

MATLAB code to perform the above is shown below in Figure 1. Take note that this is a reduced version of the code that only performs the parameter estimation.



```
function [theta, costs] = linear_regression(x, y, alpha, init_theta, N)

% 1. Find total number of samples
m = numel(x);

% 2. Create an array of costs
costs = zeros(m,1);

% 3. Initialize output parameters
theta = init_theta;

% 4. Perform gradient descent
 for ii = 1 : N
     % 4(a). Update parameters
     temp0 = theta(1) - (alpha/m)*sum(theta(1) + theta(2)*x - y);
     temp1 = theta(2) - (alpha/m)*sum((theta(1) + theta(2)*x - y).*x);
     theta(1) = temp0;
     theta(2) = temp1;

     % 4(b). Compute cost
     costs(ii) = (0.5/m)*sum((theta(1) + theta(2)*x - y).^2);
 end
```

Figure 1: Univariate Linear Regression via Gradient Descent

This code can be downloaded off of the ELE 888 / EE 8209 D2L course under the Lab 1 section and is called `linear_regression.m`*.* The functionality of the code can be summarized as follows:

1. The function `linear_regression` requires the following inputs:
   - `x`: The training examples $x^{(i)}, i = 1, 2, \ldots, m$ denoted by the use of a single feature. Expected to be a column vector.
   - `y`: The expected output $y^{(i)}$ for each $x^{(i)}, i = 1, 2, \ldots, m$. Expected to be a column vector.
   - `alpha`: The learning rate or step-size. Should be a positive number - $(\alpha > 0)$
   - `init_theta`: The initial parameters as a two-element column vector (i.e. `init_theta = [0; 0];`).
   - `N`: The total number of iterations you would like to perform. Should be a positive number - $(N > 0)$
2. The first step is to determine the total number of samples. `numel` is a function that counts how many elements are in an array or matrix. We also initialize an array of costs called `costs` where each element in `costs` tells you the cost of $J(\theta_0, \theta_1)$ at each iteration. Initially every element in this array is 0. Finally, we initialize the output parameters to be what the initial input parameters were to allow for easy updating.
3. We now perform Gradient Descent via linear regression. We simultaneously update θ_0, θ_1 for as many iterations declared by `N`. We also compute the cost at each iteration.
4. What is returned are the final learned parameters `theta` as a two element column vector and the array of `costs` created before.

Figure 2 below shows you how to run this code with our example seen at the end of the first class:

```
x = [1;2;3]; % Training data
y = [2;4;5];

initial_theta = [0; 0]; % Initial parameters
N = 50; % Number of iterations
alpha = 0.01; % Learning rate

[theta, costs] = linear_regression(x, y,  alpha, initial_theta, N);
```

Figure 2: Running regression example as seen in class

After running this code, we can plot the points as well as the line of best fit that fits through these points. This code is shown below wrapped in a function in Figure 3.

```
function plot_line_and_points(x, y, minX, maxX, theta)

    % Find a range of points from the smallest to largest x value
    xx = linspace(minX, maxX).';

    % Compute the predictions
    % This code is equivalent to doing yy = theta(1) + theta(2)*xx;
    % in the case of linear regression
    xp = [ones(numel(xx),1) xx];
    yy = xp*theta;

    % Plot the points and the line
    figure;
    plot(x, y, 'rx', xx, yy, 'b', 'MarkerSize', 16);
    title(sprintf('Plotted points and line of best fit: %f + %f x', theta(1), theta(2)));
```

Figure 3: Code to plot points and line of best fit

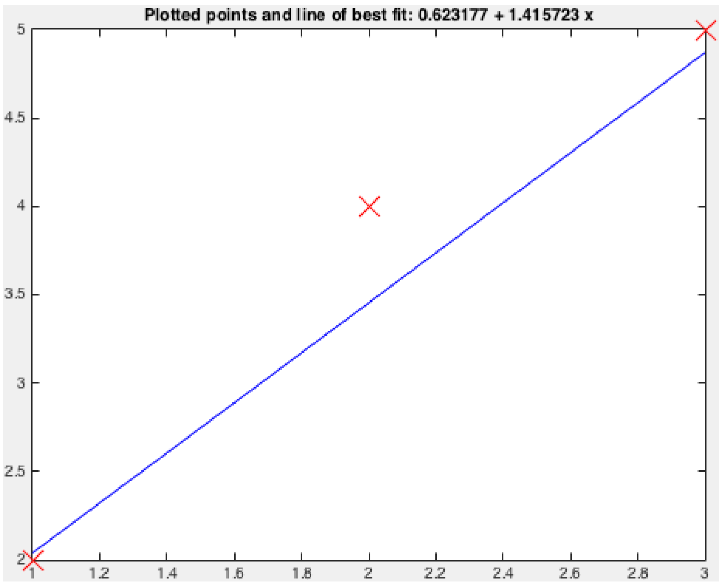
This code can be downloaded off of the ELE 888 / EE 8209 D2L course under the Lab 1 section and is called `plot_line_and_points.m`.

The code is self-explanatory. The first part is to generate a bunch of intermediate `x` points spanning from input value `minX` to the input value `maxX`. These intermediate points are used to give the perception of drawing a continuous line as MATLAB can only store things discretely. These usually are the smallest and largest `x` points in your one feature dataset. The function `linspace` does that and by default, it generates 100 points linearly spaced. Next, we compute the output given by the parameters for each value of `x` generated from `linspace`. Take note that the predictions were computed by a linear algebra approach via matrix multiplication as covered in class. Be aware that `linspace` generates a **row** vector of points and so transposing this will ensure that it's a column vector. This formulation will make it easier when you extend this to more features. The last part is to plot both the points as red crosses and the line of best fit through these points and provide a title that shows the parameters output from the linear regression code. The `'MarkerSize'` attribute was changed to size 16 to make the points larger for easier display.

Continuing after the code in Figure 2, by running the following command:

```
plot_line_and_points(x, y, min(x), max(x), theta);
```

We thus get:



After running this code, we can also plot what the cost function output looks like at each iteration. Figure 4 shows us that code wrapped in another function.

```
function plot_costs(costs)
    N = numel(costs);
    figure;
    plot(1:N, costs, 'b', 'MarkerSize', 16);
    xlabel('Iteration number');
    ylabel('Cost: J(\theta)');
    title('Cost function output vs. the # of iterations');
    grid;
```

Figure 4: Code to plot the cost function as a function of the iteration

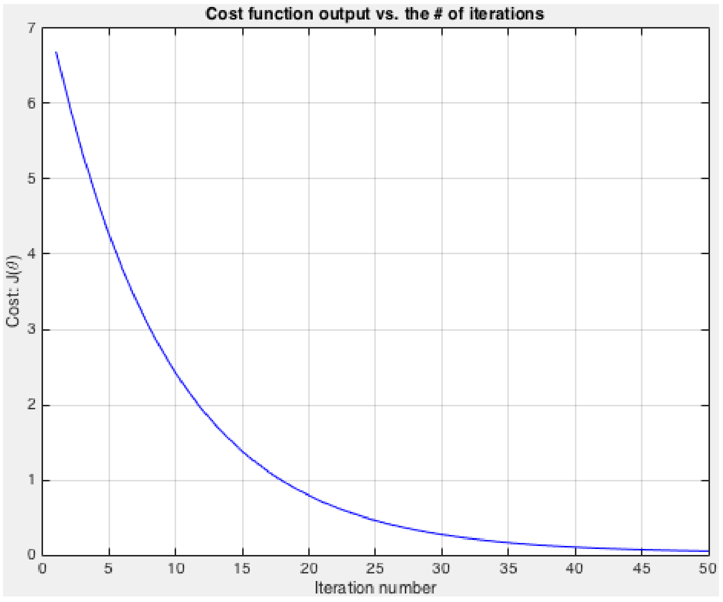
> *This code can be downloaded off of the ELE 888 / EE 8209 D2L course under the Lab 1 section and is called `plot_costs.m`.*

The code is also self-explanatory. The `x` axis of the plot is the iteration number while the `y` axis of the plot is the cost at each iteration. We label the axes and title the graph accordingly. We also give it a nice grid.

Continuing from Figure 2, after we compute the linear regression coefficients, we can run the function to plot the graph:

```
plot_costs(costs);
```

We get:



Notice how the cost function is decreasing at each iteration. This gives us an indication that we chose a correct learning rate to ensure convergence. An additional visualization we can show given our points defined earlier the **cost function surface** where the `x` and `y` axes denote the parameters θ_0 and θ_1 while the `z` axis computes the cost given the pair (θ_0, θ_1). We can also show a contour plot that achieves the same effect but on a 2D surface. Figure 5 shows us this code, assuming you ran the code given in Figure 2 above:

```
%%% Section to show cost function surface
m = numel(x);
[X,Y] = ndgrid(-2:0.01:2, -2:0.01:2);
Z = zeros(size(X));
for ii = 1 : size(X,1)
   for jj = 1 : size(X,2)
      theta0 = X(ii,jj);
      theta1 = Y(ii,jj);
      Z(ii,jj) = (0.5/m)*sum((theta0 + theta1*x - y).^2);
   end
end
```

```
surf(X,Y,Z);
xlabel('\theta_0'); ylabel('\theta_1'); zlabel('J(\theta_0, \theta_1)');
axis ij;
shading interp;

%%% Section to show contour plot
figure;
contour(X, Y, Z, logspace(-2,2,20));
xlabel('\theta_0'); ylabel('\theta_1');
colorbar;
```

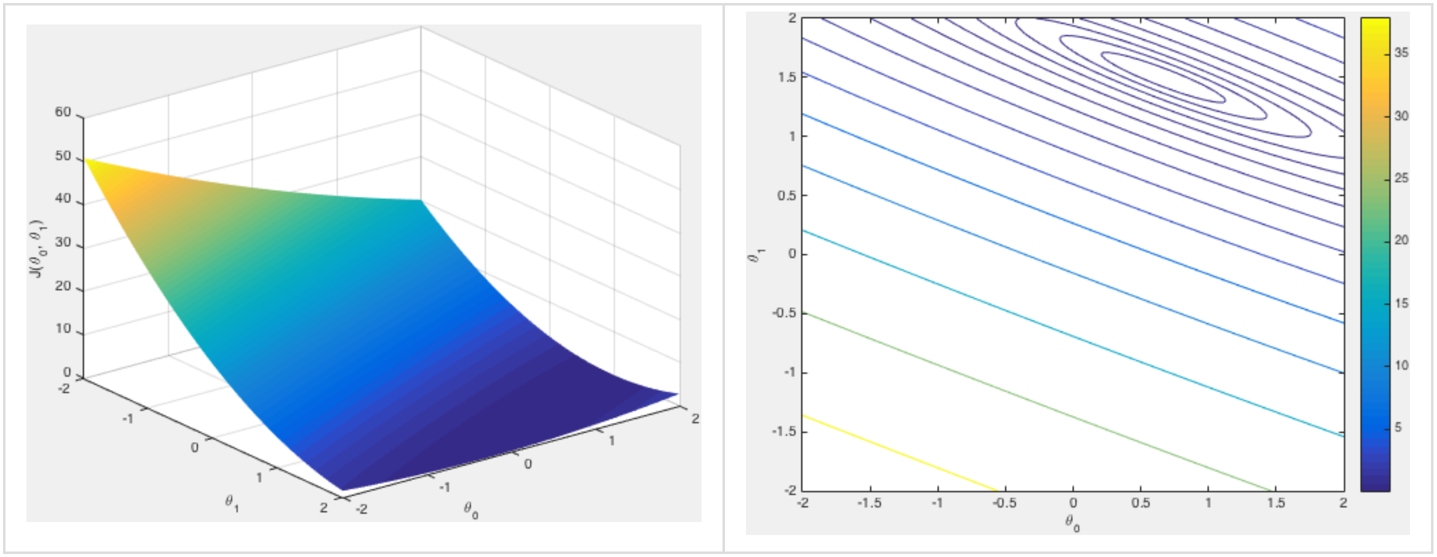
Figure 5: Showing the cost function surface and contour plot

> *The code running Figures 2 through 5 can be downloaded off of the ELE 888 / EE 8209 D2L course under the Lab 1 section and is called `create_plots.m`.*

The first section of code generates a grid of 2D coordinates with `ndgrid` which spans between -2 to 2 for both the θ_0 and θ_1 parameters in steps of 0.01. Next we go through each pair of (θ_0, θ_1) and compute the cost $J(\theta_0, \theta_1)$ given the input and output values to be placed in an output array where each spatial position gives you the cost for each pair (θ_0, θ_1). We then show this surface plot with the `surf` command and label the axes accordingly. We also flip the `y` coordinate of the plot (`axis ij`) and set the shading to interpolation mode to allow for better visualization (`shading interp`).

The next section plots a contour plot like we've seen in class for 20 logarithmically spaced cost levels between 0.01 and 100. A colour bar has also been added in so you can see which colour corresponds to what height of the surface plot.

These are the plots that you get:



Note how there is a clear minimum delineated by the contour plot which can be seen as the lowest point on the surface plot as well. Remember that this unique minimum is only possible because the cost function is **convex**. Minimizing convex optimization problems are important because there is guaranteed to be one unique minimum without settling in any areas with local minima.

1 - Multivariable Linear Regression

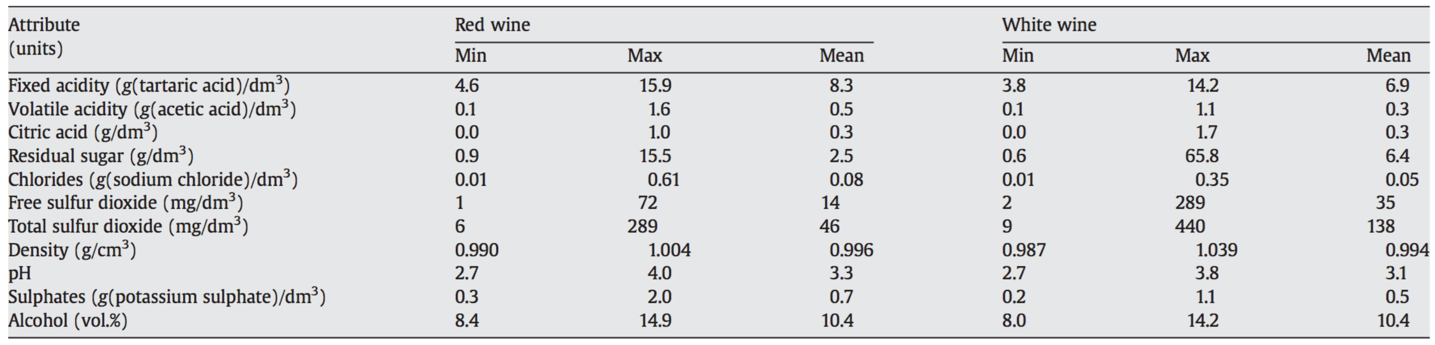
1.1 - Introduction

This section of the lab concerns implementing multivariable linear regression. The warmup section summarized how to do this for univariate or for one feature as well as showing the surface plots and contour plots associated with this problem. In general, regression problems seldomly use one feature and so one popular solution is to increase the number of features and thus increasing the number of model parameters. This increased number of parameters will optimistically increase accuracy and provide a better fit for the training and ultimately test data.

The data we will be using for this section comes from a real world problem. This data comes from measuring different quantities of various red wine samples that originate from the north of Portugal. This data comes from Paulo Cortez from the University of Minho, Portugal. This data is posted publicly on the UCI Machine Learning Database via http://archive.ics.uci.edu/ml/datasets/Wine+Quality.

> *If you're curious about the study, the scientific paper that details this research can be found in the Lab 1 section of the ELE 888 / EE 8209 D2L course page under the file name* `CortezWineDataMining.pdf`.

Your instructor has post-processed the data for ease of processing and model training and is posted on the Ryerson ELE 888 / EE 8209 D2L course website under the filename `lab1winedata.mat`. There are 1599 samples with 10 features in order to assess the quality of the wine. Download this file to use for completing this part of the lab. The goal of this data is to measure the quality of the red wine using various numerical features. The features are summarized in Table 1 below as well different statistics related to each feature collected.



Attribute (units)	Red wine			White wine		
	Min	Max	Mean	Min	Max	Mean
Fixed acidity (g(tartaric acid)/dm^3)	4.6	15.9	8.3	3.8	14.2	6.9
Volatile acidity (g(acetic acid)/dm^3)	0.1	1.6	0.5	0.1	1.1	0.3
Citric acid (g/dm^3)	0.0	1.0	0.3	0.0	1.7	0.3
Residual sugar (g/dm^3)	0.9	15.5	2.5	0.6	65.8	6.4
Chlorides (g(sodium chloride)/dm^3)	0.01	0.61	0.08	0.01	0.35	0.05
Free sulfur dioxide (mg/dm^3)	1	72	14	2	289	35
Total sulfur dioxide (mg/dm^3)	6	289	46	9	440	138
Density (g/cm^3)	0.990	1.004	0.996	0.987	1.039	0.994
pH	2.7	4.0	3.3	2.7	3.8	3.1
Sulphates (g(potassium sulphate)/dm^3)	0.3	2.0	0.7	0.2	1.1	0.5
Alcohol (vol.%)	8.4	14.9	10.4	8.0	14.2	10.4

Table 1: Features collected from red wine samples with relevant statistics

You will also notice that each feature has a different dynamic range and this is an aspect that we will explore in this lab. In MATLAB, ensure that the data is placed in your working directory and use `load lab1winedata` in the MATLAB Command Prompt or MATLAB's interactive import data tool to import this into MATLAB will load in this data as a `1599 x 10` matrix stored in `X`. Each column is a feature and how the matrix is populated is in order of appearance in the table (i.e. The first column is fixed acidity, the second column is volatile acidity, etc.). The output variable `y` stores the expected quality for each training example (row) of the matrix `X` given the 10 features.

1.2 - Implement Multivariable Linear Regression

Using `linear_regression.m` as a model, write a MATLAB function called `multivar_regression.m` that performs multivariable linear regression. You will be using this function to conduct your experiments and record your observations. There is also a skeleton file under the same file name on our course website where you can download it and fill in what is required.

If you recall from class, the cost function now incorporates multiple features and the prediction model is now a weighted sum of features:

$$J(\theta_0, \theta_1, \ldots, \theta_n) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)} - y^{(i)})^2$$

There are now n features as opposed to just one feature as seen with univariate linear regression. Recalling from class, the algorithm for implementing multivariable linear regression via Gradient Descent is the following:

1. Start with an initial set of parameters (i.e. $\theta_0 = 0, \theta_1 = 0, \ldots, \theta_n = 0$).
2. Simultaneously update the parameters and

$$\text{repeat until convergence } \{$$

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

$$\text{for } j = 0, 1, \ldots, n, \quad x_0^{(i)} = 1, \text{for } i = 1, 2, ..., m$$

$$\}$$

3. Prediction model is now of the form $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n$. Use this to predict new values.

$x_j^{(i)}$ is the jth feature for the ith training example and for the bias term, we enforce that $x_0^i = 1$ for $i = 1, 2, \ldots, m$. You can use `linear_regression.m` and modify the loop to incorporate multiple features or you can use the linear algebra formulation derived in class if you are more adventurous. **Be advised that if you pursue the linear algebra approach, you should prepend a column of ones to your data matrix before calling this method.** In fact, this is the recommended approach as MATLAB performs matrix algebra calculations very quickly and efficiently. The function should have the following input and output specifications:

Inputs

1. `X`: A data matrix where each column is a feature and each row is a training example. If you decide to pursue the linear algebra approach, you **must** ensure that the first column of this matrix will append a column of ones necessary for regression to work in the data matrix. This will be a `m x (n + 1)` matrix where `m` is the number of training examples and `n` is the number of features. You can leave this as `m x n` if you don't adopt the linear algebra approach. Assuming that you have already created the matrix `X`, this column of ones can be appended by doing:

   ```
   X = [ones(m,1) X];
   ```

   `ones(m,1)` creates a `m x 1` column vector of all ones.
2. `y`: A column vector of `m x 1` that denotes the expected output of each training example.
3. `init_theta`: The initial parameters. This is a `(n + 1) x 1` column vector. Remember there is an additional parameter θ_0 for the bias term.
4. `alpha`: The learning rate for gradient descent.
5. `N`: The total number of iterations.

Outputs

1. `theta`: The final learned parameters that fit the training data. This is a `(n x 1) x 1` column vector.
2. `costs`: The cost function evaluated at each iteration. This is a `N x 1` column vector.

> *Tip: Multivariable is simply the more general case of linear regression. Univariate is simply a special case. As such, to double check that your method is working correctly, make sure it **matches** the example output parameters given the example input seen in the warmup section. If it matches, then you know you have a correct implementation.*

1.3 - Train and Assess a Wine Quality Prediction Model

Now that you've implemented multivariable gradient descent, it's time to put it to use. You will use various features to try and obtain an accurate prediction model for predicting the quality of wine. From this point onwards, you are to complete the `wine_prediction` script to produce all results and you are to include this with your report.

1. Use the **fixed acidity, citric acid and density** features and create your prediction model using `multivar_regression.m` that you created in Section 1.2. First create a data matrix `X` where the first column has all ones, the second column is the fixed acuity, the third column is the citric acid and the last column is the density. This corresponds to the first, third and eighth columns respectively of the original data matrix `X`. You can select out these features by doing:

```
Xtrain = X(:,[1 3 8]);
```

Use `Xtrain` to create the final data matrix `X` for use in this section. Set the learning rate $\alpha = 0.01$ and use `N = 100` iterations. Report the learned parameters $\vec{\theta}$ in your report and also plot the cost function as a function of the number of iterations to be included in your report. The code in `plot_costs` may be of use here.

Because there are three variables of interest here, a surface plot is not possible for visualization so just stick with the cost function vs. number of iterations plot.

2. After creating the model, use training examples 1, 4, 55, 126, 213 and 275 from the data matrix `X` and predict what the actual quality would be. Table 2 below shows what the features are at these specific training examples.

Training Example #	Fixed Acidity	Citric Acid	Density	Quality
1	7.4	0	0.9978	5
4	11.2	0.56	0.998	6
55	7.6	0.15	0.9955	6
126	9	0.04	0.9984	5
213	11.6	0.64	0.998	6
275	7.5	0.18	0.9991	5

Assuming you created the data matrix `X` from the previous step, you can access this subset of data by doing:

```
Xtest = X([1 4 55 126 213 275], :);
```

3. Compare the predicted values computed in the previous step to the actual quality of the wine for each of the training examples tested. You can determine what each of the quality measures are by:

```
Ytest = y([1 4 55 126 213 275]);
```

Comment on any deviations between the predicted values and the true values. Why do you think there is a deviation from the true result? Could the dynamic range of the features be a factor in the deviation?

1.4 - Feature Normalization

1. With reference to the previous section, you may have noticed that the dynamic range of all of the features is quite different from one another. This may be a factor in the accuracy of the linear regression model that was created. Therefore, one thing we can try doing is **feature** normalize each of the features prior to training the model. Because of the normalization, convergence should be faster as all of the features will roughly be in the same dynamic range. We will use a learning rate of $\alpha = 0.1$ and will leave the total number of iterations to be `N = 100`. It is recommended you save a copy of the data matrix prior to doing this step.

You can achieve this normalization by the MATLAB function:

```
function [X, mu, sigma] = normalize_features(X)
    mu = mean(X, 1);
    sigma = std(X,[],1);
    X = bsxfun(@minus, X, mu);
    X = bsxfun(@rdivide, X, sigma);
```

> *This code is made available to you in the Lab 1 section of the ELE 888 / EE 8209 D2L course website and is called `normalize_features.m`.This is a very useful function that you should download for use later.*

We first find the mean of each feature which will be placed in a row vector. We also find the standard deviation of each feature. The last two lines of code in `normalize_features` subtracts the mean of each feature over all examples and divides each value by its respective standard deviation. What is returned are the normalized features of all of the training examples as well as the means and standard deviations of the features as row vectors.

To run this function, given that your training data and labels are stored in `X` and `y` respectively, you would do it like so:

```
[Xbar, mu, sigma] = normalize_features(X, y);
```

Once you feature normalize the data matrix, re-run multivariable linear regression using this new matrix maintaining the same learning rate and number of iterations as before. Report the learned parameters $\vec{\theta}$ in your report and also plot the cost function as a function of the number of iterations.

2. After creating the model, again use training examples 1, 4, 55, 126, 213 and 275 from the data matrix `X` and predict what the actual quality would be. **Remember that you must normalize the training examples so they have zero-mean and unit variance before finding the parameters**. Consult the code above in the previous question for insight. Is there a difference between the results using just three features with normalization and without normalization? Is the accuracy better or worse? What could you do to improve the accuracy?

1.5 - Selecting the Right Learning Rate

1. Carrying on from Section 1.5, in addition to normalizing the features, another thing we can try and do is to vary α and to find the model parameters with each value of α. You would then take a look at the cost associated with each α and choose the one that has the smallest cost. As such, vary α so that you train your model **using the normalized features** using the following values: $\alpha = \{0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30\}$ while keeping the number of iterations to be `N = 100`. For each α, take note of the final cost assigned at the final iteration. Choose the learning rate that produces the smallest cost and report this. Also plot the cost function for this selected learning rate as a function of the number of iterations.
2. Using the parameters chosen at the desired learning rate α, use the training examples 1, 4, 55, 126, 213 and 275 from the data matrix `X` and predict what the quality would be. **Remember that you must normalize the training examples so they have zero-mean and unit variance before finding the parameters.**

Compare these results to those without normalization, with normalization with a learning rate of $\alpha = 0.01$ and your selected learning rate from the previous question. Is the accuracy from this combined approach better or worse?

2 - Regularization

2.1 - Introduction

This section of the lab concerns incorporating regularization into the multivariable linear regression algorithm to prevent overfitting the line of best fit through the data. If you recall from the lectures, having as many features as possible is a good idea to get the best line of fit possible. This will certainly provide model parameters that produce a very good line of fit but this will **not** generalize well to new examples. This is what we call **overfitting** the model parameters to the data. One method to reduce overfitting is to reduce the amount features but it takes time and experimentation to figure out which features to eliminate. Another method is to simply keep all of the features and use **regularization** to compensate.

The goal of regularization is to keep all of the features but reduce the magnitude or the values of all of the θ_j s to prevent overfitting. This works quite well when there are a lot of features where each feature contributes a bit to predicting the output. Take note that the bias term θ_0 is not part of the regularization as the bias contributes the most to the accuracy of the algorithm, so we need to ensure that its magnitude isn't reduced.

To demonstrate how regularization is useful, we will use data provided by Andrew Ng's Machine Learning course on `coursera.org`. This data describes the amount of water flowing out of a dam using the change of water level in a reservoir. There are 12 training examples and output labels with this data. Our goal is to predict the change of water level given the amount of water flowing out of a dam. This data is highly non-linear, which makes this a suitable candidate for regularization. This data is available on the ELE 888 / EE 8209 course website and is called `lab1poly.mat`. Like the previous portion of this lab, ensure that this file is in your working directory and run the `load lab1poly.mat` command. There will only be two variables in your workspace:

1. `X`: The change in water level in a reservoir as a `12 x 1` column vector.
2. `y`: The water flowing out of a dam as a `12 x 1` column vector.

2.2 Implement Multivariable Linear Regression with Regularization

Using `multivar_regression.m` as a model, write a MATLAB function called `multivar_regression_reg.m` that performs multivariable linear regression with reguarlization. You will be using this function to conduct your experiments and record your observations. The inputs and outputs are exactly the same with the exception of the additonal input parameter `lambda` that controls the regularization (more on this later). Make this the second last parameter in your implementation.

Specifically, the function prototype should now look like:

```
function [theta,costs] = multivar_regression_reg(X, y, init_theta, alpha, lambda, N)
```

To be unambiguous, a skeleton of this function has been posted in the Lab #1 section of the ELE 888 / EE 8209 course website for you to download. You may use this to complete the function.

Recalling from lecture in regularization, the cost function is modified so that an additional penalty term is added with respect to the parameters. Recalling from the lecture, the cost function is now:

$$J(\theta_0, \theta_1, \ldots, \theta_n) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m} \sum_{j=1}^{n} \theta_j^2$$

Remember that we don't apply regularization to the bias term θ_0. λ is the regularization parameter which controls the overfitting issue with a large number of features. $\lambda = 0$ implies that no regularization is performed and normal Gradient Descent is to be sought. When choosing the regularization parameter, you need to choose something sensible. Making λ quite large would mean that each of the parameters contributes a very tiny amount in order to ensure the cost is minimized and so this would result in **underfitting** the model parameters to the data. We will explore different values of λ and see how this affects our model parameters.

Using Gradient Descent to minimize the modified cost function now looks like the following.

1. Start with an initial set of parameters (i.e. $\theta_0 = 0, \theta_1 = 0, \ldots, \theta_n = 0$).
2. $$\begin{gathered}\text{repeat until convergence } \{ \\ \theta_0 \leftarrow \theta_0 - \alpha \left[\frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^i) - y^{(i)}) \right] \\ \theta_j \leftarrow \theta_j - \alpha \left[\frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}) + \lambda \frac{\theta_j}{m} \right] \\ \text{for } j = 1, 2, \ldots, n \} \end{gathered}$$
3. Prediction model is now of the form $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n$. Use this to predict new values.

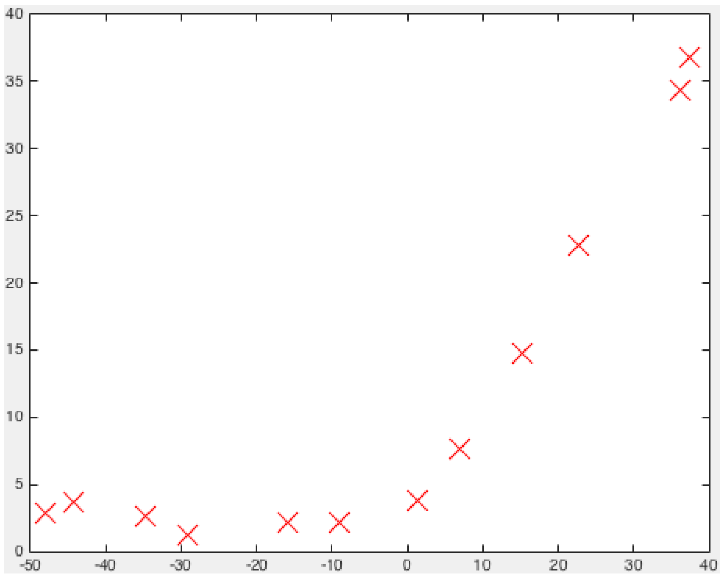
"

Tip: Recall that setting $\lambda = 0$ for the above formulation corresponds to normal multivariable linear regression. As such, to double check that your method is working correctly, make sure it ***matches*** *any of the output parameters you achieved in the prevous section with your regular multivariable linear regression code when seting `lambda = 0`. If it matches, then you know you have a correct implementation.*

2.3 - Train a Prediction Model to Predict Water Level Changes

Now that you've implemented multivariable gradient descent with regularization, it's time to put it to use. From this point onwards, you are to complete the `water_level` script to produce all results and you are to include this with your report. Here is what the dataset looks like for self-containment along with the code to generate the visualization.

```
load lab1poly.mat
plot(X, y, 'rx', 'MarkerSize', 16);
```



You can clearly see that the data is non-linear in nature. Therefore, using just the basic linear regression model with one feature will be unsuitable to model the behaviour of this dataset correctly. As such, it may be prudent to add more features to correctly model the non-linear behaviour. This can be done by adding polynomial features and representing the prediction model as a weighted sum of polynomials instead. Concretely, we can represent our prediction model as:

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

The above introduces `p` polynomial features for use in training. Or more formally:

$$h_\theta(x) = \sum_{i=0}^{p} \theta_i x^i$$

This is enforcing that $x_0 = 1$. However, with each polynomial term added of increasing order, the dynamic range will also get larger. For example, if $x = 40$, then $x^4 = 2560000$. When adding polynomial features, it is recommended that you normalize each feature prior to training your model to ensure that Gradient Descent converges when finding the parameters.

For the purpose of this part of the lab, we will introduce ***eight (8)*** polynomial features or `p = 8`. Therefore, given a vector of single features as given in this part of the lab, we wish to create 7 more polynomial features for a total of 8. You can use the `poly_features.m` file that transforms a column vector of single features of size `m x 1` to a matrix of size `m x p` where each column i is each training sample raised to the power of i. The code for this is shown below.

```
function xp = poly_features(x, p)
    xp = bsxfun(@power, x, 1:p);
```

“

This code is made available to you in the Lab 1 section of the ELE 888 / EE 8209 D2L course website and is called `poly_features.m`. This is a very useful function that you should download for use later.

To call the code, simply do:

```
p = 8;
xp = poly_features(x, p);
```

`p` is our desired number of polynomials. Be aware that this code **does not** append a column of ones to incorporate the bias term. Make sure you do this before proceeding with the following questions.

1. Let's start by setting the regularization parameter $\lambda = 0$. This should essentially be normal linear regression without regularization. Create the polynomial features with `poly_features` using the input data `X` then ensure that you normalize the features. Using `normalize_features` is prudent here. Now create your prediction model using `multivar_regression_reg` to find the model parameters. Set the learning rate $\alpha = 0.1$ and the total number of iterations to be `N = 500`. Include the model parameters trained with $\lambda = 0$ in your report. Once that's performed, use `plot_line_and_points` as a template for drawing the points `X` and the line of best fit with the trained parameters. Remember that MATLAB plots points in a discrete manner so if you want to make the plot look continuous, you need to generate intermediate points with a small enough step size in between the given data points. To get a good idea of how the plot looks like, it's recommended that you make the minimum and maximum values in the range you want to plot a bit wider so you can see how well the model fits the data. Something like this will do:

   ```
   xx = linspace(min(X) - 15, max(X) + 15).';
   ```

   `xx` would be a set of 100 linearly spaced points between the smallest and largest `X` value but there is some breathing room before the smallest point and after the largest point to see the plot better. You can use this in conjunction with `poly_features` and `normalize_input` with the mean and standard deviation of each feature produced by `normalize_features` to create new features to draw the final fitted line with the training data points. Include this plot in your report. Also plot the cost function as a function of the number of iterations (i.e. use `plot_costs`) Comment on the accuracy of the model parameters. Do you believe this would generalize well for new instances?

2. Repeat the previous question but set the regularization parameter $\lambda = 1$. Use the same $\alpha = 0.1$ and `N = 500` like before. Make sure you include the model parameters trained and the plot of the data points and the line of best fit with the trained parameters. Also plot the cost function as a function of the number of iterations (i.e. use `plot_costs`). Comment on the accuracy of the model parameters. Do you believe this would generalize well for new instances?

3. Finally, repeat the previous question but use the regularization $\lambda = 100$. Use the same $\alpha = 0.1$ and `N = 500` like before. Make sure you include the model parameters trained and the plot of the data points and the line of best fit with the trained parameters. Also plot the cost function as a function of the number of iterations (i.e. use `plot_costs`). Comment on the accuracy of the model parameters. Do you believe this would generalize well for new instances?

Discussion

These are open ended discussion questions designed to assess whether you have learned all you needed to learn in this lab. There are no right or wrong answers. Answer whatever comes off the top of your head and which makes sense!

1. From the first part of this lab, we dealt with only three features from the wine dataset. Do you beieve that accuracy could be improved by adding more features?
2. Continuing from (1), what steps would you take to ensure that Gradient Descent would converge and give you the best parameters possible for fitting?

3. In this lab, we made the number of iterations for Gradient Descent an input parameter. Is there a way to remove this functionality and allow Gradient Descent to stop iterating after a certain point? More specifically, what measures would you use to allow Gradient Descent to be performed without specifying the total number of iterations before hand?
4. From the wine data set, we see that the quality is actually a **discrete** output label rather than a continuous one. Regression seeks to find continuously valued output. Do you think more accuracy would be achieved if we changed the task to **classification** instead of regression? Explain.
5. As seen in the second part of this lab, we chose only three different values of the regularization parameter λ. For each parameter, we observed the fitted line as well as the cost function for each parameter. Is there a way to automatically assess or figure out the best regularization parameter to choose?
6. You'll have also noticed that for the second part, we required that we increased the total number of iterations for the algorithm to converge. Specifically, we needed 500 iterations for the second part in comparison to 100 for the first part. If you tried specifying 100 iterations for this second part instead of the 500, no matter which regularization parameter you choose, the fitted curve more or less looks the same and in fact does not converge after 100 iterations. Why do you believe that an increased number of iterations is needed to ensure the best parameters are obtained to get a good fit? Could it be something to do with the highly non-linear nature of the data?