Linear Regression with Multiple Variables: A Machine Learning Approach

Renzo A. Viloche Morales

April 14, 2017

1 Basic Model and Notation

Linear regression models are the simplest prediction models for a single *target* (output) based on multiple input variables or *features*[1]. It is assumed that the output is a *continuous* and a linear function of all considered inputs. The general *hypothesis function* for a number of different n features has the form of:

$$\hat{y} = h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \ldots + \theta_n x_n \tag{1}$$

Equation (1) can also be expressed as an inner product between a *parameter vector*[2] and observations inputs (features values),

$$h_\theta(x) = \boldsymbol{\theta}^T \cdot \boldsymbol{x} \tag{2}$$

where, $\boldsymbol{\theta} = [\theta_0\ \theta_1\ \ldots\ \theta_n]^T$, and $\boldsymbol{x} = [x_0\ x_1\ \ldots\ x_n]^T$ are, respectively, the parameter vector and the observed feature values[3].

The previous equation showed how the model computes a predicted output from a single "feature vector". The following equation is a vectorized and compact way of expressing the input-ouput relation for m different training examples

$$\boldsymbol{h_\theta(X)} = \boldsymbol{X\theta} \tag{3}$$

Clearly, equation (3) is just a linear system of m equations in the form of the hypothesis function (1) for a set of observed inputs and targets. Observations inputs are stored at the $m \times (n+1)$ dimensional $\boldsymbol{X}$ matrix and all predicted outputs (targets) represented by the $\boldsymbol{\hat{y}}$ vector,

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x^{(1)}}\ ^T \\ \boldsymbol{x^{(2)}}\ ^T \\ \boldsymbol{x^{(3)}}\ ^T \\ \ldots \\ \boldsymbol{x^{(m)}}\ ^T \end{bmatrix}, \qquad \boldsymbol{\hat{y}} = \boldsymbol{h_\theta(X)} = \begin{bmatrix} \boldsymbol{y^{(1)}} \\ \boldsymbol{y^{(2)}} \\ \boldsymbol{y^{(3)}} \\ \ldots \\ \boldsymbol{y^{(m)}} \end{bmatrix} \tag{4}$$

The i-th row vector $\boldsymbol{x^{(i)}} = [x_0^{(i)}\ x_1^{(i)}\ x_2^{(i)}\ \ldots\ x_n^{(i)}]^T$ represents the i-th observation sample (the j-th column is represented by the $\boldsymbol{x_j}$ vector which represents all different observations for the j-th feature). As it will be shown, the representation of equations (3) and (4) will be particularly

[1]This procedure is also known in the literature as *Multivariate Linear Regression.*

[2]The parameter vector is also known under the name of *weight vector.*

[3]In this notation, the first component is $x_0 = 1$ in order to preserve equation (1).

convenient to express both, iterative and non-iterative, learning algorithms that are able to find an "optimal" set of parameters values (according to some specific criterion). In the next section, this criterion and the linear regression problem will be mathematically stated.

2 Minimizing the Cost Function

The linear regression problem can be generally stated as an optimization problem where a *cost function* or *objective function* must be minimized. The straightforward choice for the cost function is a quadratic sum of the residuals from observed outputs and predicted values[4]

$$J(\boldsymbol{\theta}) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(\boldsymbol{x^{(i)}}) - y^{(i)})^2 \tag{5}$$

where $y^{(i)}$ is the output target value with repect to the ith-example. The optimization problem to be solved is

$$\min_{\boldsymbol{\theta}\in\boldsymbol{\Re^{n+1}}} \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(\boldsymbol{x^{(i)}}) - y^{(i)})^2 \tag{6}$$

in which a global minimum solution is guaranteed to exist if the basic model (equation 1) is chosen. A more general condition to the convergence of algorithms to a minimum is that $J(\boldsymbol{\theta})$ must be a *convex function.* The next sections will describe two algorithms that can be used to find the parameter values $\theta_0,\ \theta_1,\ \theta_2,\ \ldots,\ \theta_n$ that minimizes $J(\boldsymbol{\theta})$.

2.1 Gradient Descent Algorithm

The Gradient Descent algorithm operates iteratively computing at each step a solution that is (in theory) closer to the global minimum. The key idea here is to update the previous solution with the corresponding cost function opposite highest rate

$$\boldsymbol{\theta}_{new} = \boldsymbol{\theta}_{old} - \alpha\boldsymbol{\nabla}[J(\boldsymbol{\theta}_{old})] \tag{7}$$

where the scalar α is known as the *learning rate.* The reason for adopting a scalar (usually different from zero) is that the learning rate is able to module the displacement or "jumps" allowing a faster (or slower) convergence to the global minimum. Assuming the basic Multivariate Linear Regression model, and the cost function previously defined, each component of the solution updating rule (equation 7) can be expressed as

$$\begin{array}{c}
\theta_0 := \theta_0 - \frac{\alpha}{m}\sum_{i=1}^{m}(h_\theta(\boldsymbol{x^{(i)}}) - y^{(i)}) \\
\theta_1 := \theta_1 - \frac{\alpha}{m}\sum_{i=1}^{m}(h_\theta(\boldsymbol{x^{(i)}}) - y^{(i)})x_1^{(i)} \\
\theta_2 := \theta_2 - \frac{\alpha}{m}\sum_{i=1}^{m}(h_\theta(\boldsymbol{x^{(i)}}) - y^{(i)})x_2^{(i)} \\
\vdots \\
\theta_n := \theta_n - \frac{\alpha}{m}\sum_{i=1}^{m}(h_\theta(\boldsymbol{x^{(i)}}) - y^{(i)})x_n^{(i)}
\end{array} \tag{8}$$

where, the ':=' symbol denotes the assignment operation commonly used for algorithms[5]. The following describes the Gradient Descent algorithm,

[4]The cost function is a *error loss* measure and is defined here as the half of the mean square error (MSE) which simplifies the analytical expression for the Gradient Descent algorithm.

[5]This means that the value on the right-side of this symbol will substitute any previous value present at the variable whose storing the θ_i information.

Gradient Descent Algorithm (Basic Multiple Linear Regression):

```
Set initial guess for the parameter values: $\theta_0\ ,\theta_1\ ,\theta_2\ ,\ \ldots\ ,\theta_n$
Set the learning rate value $\alpha$
Set the total number of update iterations $N_{\text{iter}}$
begin
  for $k := 1$ to $N_{\text{iter}}$ do
      Compute every component for the new solution $\theta_j$ , for $j = 0,\ 1,\ \ldots\ ,\ n$
      $\theta_j := \theta_j - \frac{\alpha}{m}\sum_{i=1}^{m}(h_\theta(\boldsymbol{x^{(i)}}) - y^{(i)})x_j^{(i)}$ , (components must be simultaneously updated!)
      Compute the error
      $J(k) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(\boldsymbol{x^{(i)}}) - y^{(i)})^2$
  od
end
```

Although not a required condition, a random set of values for the parameters can be used to initialize the search for the solution. The learning rate α must be properly chosen in the sense that the algorithm produces a cost function that decreases with the number of iterations (the error is minimized). A rule of thumb says that the smaller the value of α is, the greater the chance for the global minimum being reached (at the expense of a required higher number of iterations). In the other hand, a large learning rate can lead to convergence failure. In effect, large "jumps" for the solution update (equation 7) can produce a (i) divergent, or (ii) oscillating characteristic for the cost function evolution. Consequently, one must always check the $J_k(\boldsymbol{\theta}) \times k$ behavior in order to assure an acceptable $\boldsymbol{\theta}$ solution (this is why it is important to compute and store the error value at each iteration). In some cases, the number of iterations can be extended if a correct minimization procedure holds true, and a lower error value is desired.

2.2 Feature Scaling

A typical problem that difficults the convergence of the Gradient Descent is the difference on the range of values of each feature. In fact, most variables used in prediction models can display a diversity in terms of unities and typical values. The computational gradient can then become dominated by a majority of small component values (lower scale feateures) and the size of each new "update step" can be severely decreased. A popular solution here consists of applying a "normalization" operation known as *feature scaling.* In feature scaling each corresponding feature $\boldsymbol{x_j}$, for $j = 1, 2,\ ,\ldots\ , n$, is transformed into a new variable $\widetilde{\boldsymbol{x_j}}$ according to

$$\widetilde{\boldsymbol{x_j}} = \frac{\boldsymbol{x_j} - \mu_{x_j}}{\sigma_{x_j}} \qquad j = 1,\ 2,\ \ldots\ ,\ n \tag{9}$$

which is a rescaled version of the original feature with its mean value μ_{x_j} and standard deviation σ_{x_j}. Ideally, using equation (9) will reduce the range of values to the $[-1,\ 1]$ interval producing a zero mean feature[6]. Another common choice for the normalization factor is the largest deviation value $max\{\boldsymbol{x_j}\} - min\{\boldsymbol{x_j}\}$ in the place of the standard deviation value.

A final observation is that using the Gradient Descent (or any other regression method) the transformation of variables will obviously produce a different optimized solution $\widetilde{\theta_0},\ \widetilde{\theta_1},\ \widetilde{\theta_2},\ \ldots\ ,\widetilde{\theta_n}$,

[6]This is expected when $\boldsymbol{x_j}$ is normally distributed for example.

since features are now different from the originals. However, it is possible to use the following set of equations,

$$\begin{array}{ll} \theta_0 = \widetilde{\theta_0} - \sum_{j=1}^{n} \frac{\mu_{x_j}\widetilde{\theta_j}}{\sigma_{x_j}} & \\ \theta_k = \widetilde{\theta_k}/\sigma_{x_k} \qquad k = 2,\ ,3\ ,\ \ldots\ ,\ n \end{array} \tag{10}$$

which recovers the expected solution related to the original features $\boldsymbol{x_1},\ \boldsymbol{x_2},\ \ldots\ ,\boldsymbol{x_n}$.

2.3 Normal Equation Method

An analytical solution to the Multivariate Linear Regression problem is the *Normal Equation* which can be obtained explicitly by either, setting the cost function derivatives with respect to θ_j's to zero, or using Linear Algebra arguments. The following result provides the theoretical optimum value,

$$\boldsymbol{\theta_{min}} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y} \tag{11}$$

where, once again, $\boldsymbol{X} \in \Re^{m\times(n+1)}$, $\boldsymbol{y} \in \Re^m$ and $\boldsymbol{\theta_{min}} \in \Re^{n+1}$. Although the apparent ease of applying Normal Equation to find the parameter values, it is important to observe that this method can also be disadvantageous in some situations. The major issue here is when the number of features n is sufficiently large so computing the inverse matrix term $(\boldsymbol{X}^T\boldsymbol{X})^{-1}$ can be slow or even impractical. The table 1 describes the main differences among Gradient Descent and Normal Equation methods. A practical rule of thumb here is to consider the Gradient Descent method whenever the number of features n is something in the order of 10^5 or greater.

Table 1: Comparison between the Multivariate Linear Regression methods.

Gradient Descent	**Normal Equation**
Need to choose α	No need to choose α
Need many iterations	No need to iterate
Feature scaling is recommended	No need of feature scaling
O(kn^2)	O(n^3), need to compute $(\boldsymbol{X}^T\boldsymbol{X})^{-1}$
Works well when n is large	Slow if n is large

As a final remark, there are some specific situations which will cause $\boldsymbol{X}^T\boldsymbol{X}$ to be non-invertible. In effect, this will hold true if the resulting $\boldsymbol{X}^T\boldsymbol{X}$ matrix possess linear dependent features (columns), or if the number of training examples m is less or equal than the number of features n. In both scenarios, the resulting matrix is non-invertible and called *singular* or *degenerate*. The recommendation here is to discard some redundant features, or consider a regularization technique.

3 Polynomial Regression

A simple modification of the hypothesis function can give more flexibility when fitting a training dataset. The "trick" here is to include polynomial terms to the basic model using original data features information. For instance, suppose the case where $n = 2$ features and the corresponding linear regression model: $\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2$. We want to fit our training data to a general quadratic form such as,

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1 x_2 + \theta_4 x_1^2 + \theta_5 x_2^2$$

where the two models differ by just 3 nonlinear terms. However, this quadratic polynomial model can also be viewed as a 5-dimensional linear model. This is possible by just feeding the 2-dimensional linear model with the 3 new following features,

$$x_3 = x_1 x_2$$

$$x_4 = x_1^2$$

$$x_5 = x_2^2$$

Concretely, by "artificially" including these new features on the original dataset, we provide a sufficient condition to fit a training data to the hypothesis $\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4 + \theta_5 x_5$. Therefore, solving a *polynomial regression* problem is equivalent to solve a "virtual"[7] and higher dimensional linear regression problem. In this sense, all previous methods of finding the optimum parameters for the minimization problem are valid (Gradient Descent and the Normal Equation).

A few observations remain on the use of polynomial models:

1. Deciding the polynomial degree: There is a no well defined criterion for defining the polynomial order that best fit the training data. A common practice is to fit different (polynomial) models, and use a *cross-validation* set to select the model with lower cost function-error (or equivalently lower RMSE) in order to make predictions.

2. The parameter number "escalade": Suppose that, instead of using a parabolic equation, we have chosen a cubic polynomial model. The general form for this model would be,

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1 x_2 + \theta_4 x_1^2 + \theta_5 x_2^2 + \theta_6 x_1^2 x_2 + \theta_7 x_1 x_2^2 + \theta_8 x_1^3 + \theta_9 x_2^3$$

which indicates a substantial increase of the number of parameters. However, in many realistic applications the number of features would be far greater than 2. Because of the new terms will include combinations of all original features, the resulting total number of parameters will grow as $\sim n^d$, where d is the polynomial degree. Even if the required number of parameters is not exceptionally large, it may be greater than the number of training examples m. A reasonable solution then is to subjectively choose a few terms (the most "relevant" ones). Another option is to work with another regression method, such as *Neural Networks*, that can represent well complex behavior without requiring a large number of parameters.

3. Feature scaling: Since we are introducing nonlinear features to the training dataset, a *feature scaling* procedure is definitely recommended in order to assure an efficient convergence to the optimum parameter values when using the Gradient Descent method (this is not the case when using the Normal equation approach).

4. Overfitting: There is a natural tendency towards *overfitting* (when the model is too flexible and random errors are taken into account to the model) specially when using high order polynomials. This is more likely to happen when the number of parameters is greater than the number of training examples m. In this case, a *regularization* procedure can be helpful by "smoothing" the model response (lowering the variance).

[7]The word *virtual* here is used in the context of the features, treating them as if they all come from independent measurements, where in reality they are not!

<u>5. Other nonlinear terms</u>: In theory, some other nonlinear terms based on the following functions, $\sqrt{x},\ \sqrt[3]{x},\ 1/x^2,\ \exp(x),\ \log(-x)$ could be also included to the model. However, this is not a very common practice, for a large number of features m we mostly do not posses strong evidences that any of these terms will be effective adjusting data (except for the case $m = 2$ where we can visually check this).

4 Regularization

Regularization is meant to control and reduce the overfitting problem: the model have adjusted excessively well available data, but fails on generalizing and making predictions based on new data. The idea is to preserve the model complexity (for instance, a high polynomial degree) while "penalizing" the parameter values on the cost function J as follows,

$$J(\boldsymbol{\theta}) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(\boldsymbol{x^{(i)}}) - y^{(i)})^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right] \tag{12}$$

The λ is known as the *regularization parameter* and allows to modulate the "inflation effect" over the parameter values. Concretely, since overfitting is usually related to low training errors, minimizing (12) would drastically reduce the parameter values that has more effect on the cost. The optimization problem can be restated as,

$$\min_{\boldsymbol{\theta}\in\Re^{n+1}} \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(\boldsymbol{x^{(i)}}) - y^{(i)})^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right] \tag{13}$$

The next sections will add the regularization parameter to the Gradient Descent and the Normal Equation algorithms.

4.1 Gradient Descent with Regularization

In order to account for the regularization on the Gradient Descent update rule, an additional term (scaled by the learning rate α) should be added to the cost function gradient as follows,

$$\boldsymbol{\theta}_{new} = \boldsymbol{\theta}_{old} - \alpha\left[\boldsymbol{\nabla} J(\boldsymbol{\theta}_{old}) + \frac{\lambda}{m}\widetilde{\boldsymbol{\theta}}\right] \tag{14}$$

where, λ is the chosen regularization parameter, m is the number of training examples, and $\widetilde{\boldsymbol{\theta}}$ is the exactly same $\boldsymbol{\Re^{n+1}}$ vector $\boldsymbol{\theta_{old}}$, except from its first component which is equal to zero. The following set of updating rules for each component can be expressed as the following assignment algorithm operations,

$$\begin{array}{ll} \theta_0 := \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(\boldsymbol{x^{(i)}}) - y^{(i)}) & \\ \theta_j := \theta_j - \alpha\left[\frac{1}{m}\sum_{i=1}^{m}(h_\theta(\boldsymbol{x^{(i)}}) - y^{(i)})x_j^{(i)} + \frac{\lambda}{m}\theta_j\right] & j\in\{1,\ 2,\ \ldots,\ n\} \end{array} \tag{15}$$

Concretely, the second update rule can be also expressed (after some manipulation) as,

$$\theta_j := \theta_j\left(1 - \alpha\frac{\lambda}{m}\right) - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(\boldsymbol{x^{(i)}}) - y^{(i)})x_j^{(i)} \qquad j\in\{1,\ 2,\ \ldots,\ n\}$$

where the term $1 - \alpha\frac{\lambda}{m}$ should be less than 1 and the second term stays the same as in the original update rule from equation (7). The regularization parameter has a more pronounced decreasing effect on the parameter values when compared to the non regularized case ($\lambda = 0$).

The following chart presents the Gradient Descent algorithm with the regularization procedure taken into account.

```
Gradient Descent Algorithm with Regularization:
Set initial guess for the parameter values: $\theta_0 , \theta_1 , \theta_2 , \dots , \theta_n$
Set the learning rate value $\alpha$
Set the total number of update iterations $N_{\text{iter}}$
Set the regularization parameter $\lambda$
begin
  for $k := 1$ to $N_{\text{iter}}$ do
    Compute every component for the new solution $\theta_j$ , for
    $\theta_j := \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(\boldsymbol{x^{(i)}}) - y^{(i)})x_j^{(i)}$      $j = 0$
    $\theta_j := \theta_j - \alpha\left[\frac{1}{m}\sum_{i=1}^{m}(h_\theta(\boldsymbol{x^{(i)}}) - y^{(i)})x_j^{(i)} + \frac{\lambda}{m}\theta_j\right]$      $j = 1, 2, \dots , n$
    Compute the error
    $J(k) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(\boldsymbol{x^{(i)}}) - y^{(i)})^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right]$
  od
end
```

4.2 Normal Equation with Regularization

The non iterative Normal Equation procedure can add in the regularization parameter inside the parentheses of the original closed form,

$$\boldsymbol{\theta_{min}} = \left(\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{L}\right)^{-1}\boldsymbol{X}^T\boldsymbol{y} \tag{16}$$

where the matrix $\boldsymbol{L} \in \Re^{(n+1)\times(n+1)}$ is very much alike the *identity matrix*, except by the fact the first diagonal element is zero,

$$\boldsymbol{L} = \begin{bmatrix} 0 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \end{bmatrix}$$

Computing the expression (16) with a positive regularization parameter can help to overcome the problem: *if $m \le n$, then $\boldsymbol{X^T X}$ is not invertible.* In fact, even if that is the case, the term $\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{L}$ can become invertible with the right choice of λ.

5 Octave/Matlab Code Examples

This section presents samples of code in Octave/MATLAB$^{\text{TM}}$ used for the Multivariate Regression methods presented here.

5.1 Feature Scaling

The following function applies the feature scaling procedure that it is used for convenience before any of the regression methods (Gradient Descent or Normal Equation). The only input required to the function is the training data matrix. The outputs are respectively: `Xfeat` (normalized features), `mu_x` (features mean values), and `sigma_x` (features standard deviation values).

```
function [Xfeat,mu_x,sigma_x] = feat_scaling(X)
%% discard 'intercept terms'
extra_col=false; % 'true' when intercept terms exists
if sum(X(:,1)==1)==size(X,1)
    X = X(:,2:end); % discard 'intercept terms'
    extra_col = true;
end


%% applies feature scaling
mu_x = mean(X,1); % computes the mean of all features
sigma_x = std(X,0,1); % computes the std of all features
X = bsxfun(@minus,X,mu_x); % mean of data is reduced to zero
X = bsxfun(@rdivide,X,sigma_x); % data normalization


%% output data will be the same format as the input
if extra_col % recovers the 'intercept terms'
    Xfeat = [ones(size(X,1),1) X];
else
    Xfeat = X;
end
end
```

5.2 Gradient Descent with Regularization

The following script is a function that implements a Multivariate Regression using the Gradient Descent method with a regularization parameter. The input arguments are:

- `X` - Training examples (disposed in columns);
- `y` - Outputs of the model (column vector);
- `alpha` - Learning rate value ($\alpha > 0$);
- `numIter` - Total number of iterations for the Gradient Descent;
- `lambda` - Regularization parameter value ($\lambda \geq 0$).

The function should also return three variables (outputs): `theta` (parameter values of the prediction model), `J` (cost function values and its respective iteration step), and `rmse` (the training root mean square value).

```
function [theta,J,rmse] = linreg_grad2(X,y,alpha,numIter,lambda)
%% adds the 'intercept terms' to training data
if sum(X(:,1)==1)~=size(X,1)
    X = [ones(size(X,1),1) X];
end


%% applies feature scaling (mean and standard deviation)
[Xfeat,mu_x,sigma_x] = feat_scaling(X);
```

```
%% regression parameters
m = size(X,1); % number of training examples
theta = zeros(size(X,2),1); % initial 'guess' for the solution
J = nan(numIter,2); % array for cost function values (for each iteration)
J(:,1) = (1:numIter)'; % store iteration steps

%% Gradient Descent algorithm (with regularization) for a fixed number of iterations
for i=1:numIter
    J(i,2) = 1/(2*m)*sum((Xfeat*theta-y).^2);
    grad = (1/m*(Xfeat*theta-y)'*Xfeat)';
    theta = theta - alpha*(grad+lambda/m*[0;theta(2:end)]);
end

%% Returns the non-normalized parameter solution
theta(1) = theta(1)-sum(mu_x.*theta(2:end)'./sigma_x);
theta(2:end) = theta(2:end)./sigma_x';


%% compute the predicted RMSE (root mean square error)
rmse = sqrt(mean((X*theta-y).^2));
end
```

5.3 Normal Equation with Regularization

The present script implements a function computing the theoretical optimum solution by using the Normal Equation with a regularization parameter. The input arguments are:

- `X` - Training examples (disposed in columns);
- `y` - Outputs of the model (column vector);
- `lambda` - Regularization parameter value ($\lambda \geq 0$).

The function returns two outputs: `theta` (theoretical parameter values), and `rmse` (the training root mean square value).

```
function [theta,rmse] = normal_eq2(X,y,lambda)
%% adds the 'intercept' term to training data
if sum(X(:,1)==1)~=size(X,1)
    X = [ones(size(X,1),1) X];
end


%% applies feature scaling (mean and standard deviation)
[Xfeat,mu_x,sigma_x] = feat_scaling(X);


%% Regression paramaters
n = size(X,2) - 1; % number of features
A = Xfeat'*Xfeat+lambda*diag([0 ones(1,n)]); % Matrix to be inverted

%% computes the theoretical solution using the pseudoinverse
theta = pinv(A)*(Xfeat'*y);


%% Returns the non-normalized parameter solution
theta(1) = theta(1)-sum(mu_x.*theta(2:end)'./sigma_x);
theta(2:end) = theta(2:end)./sigma_x';
```

```
%% computes the the predicted RMSE (root mean square error)
rmse = sqrt(mean((X*theta-y).^2));
end
```

References

[1] A. Ng. *Machine Learning Course Notes.* Online Open Course available at Coursera. https://www.coursera.org/learn/machine-learning/. Last access in April 2017.