# Chapter #2:
# Regression, Regularization and Model Selection

## Part1: Regression

### Supervised Learning

A small dataset (see Figure 1) of houses on sale, in the area code 70122, was collected from trulia.com. We want to learn from the dataset, for what sort of desired features, what will be the predicted price of a house and for that, we want to apply the machine learning models. This is *supervised learning* because, for the input features in the dataset the corresponding output, i.e., the prices (of houses) are given.

The dataset has three inputs (**X**) features: *Bed* ($X_1$, number of bedrooms), *Bath* ($X_2$, number of bathrooms), and *Area* ($X_3$, size of the living area (in sqft), and one output feature: Price ($Y$), as shown in Figure 1.

| Bed | Bath | Area | Price |
|---|---|---|---|
| 5 | 7.5 | 5823 | 695000 |
| 4 | 4 | 4314 | 749000 |
| 4 | 4 | 3800 | 320000 |
| 5 | 4 | 3500 | 38900 |
| 3 | 3 | 3193 | 500000 |
| 4 | 4 | 2962 | 300000 |
| 4 | 3 | 2860 | 439000 |
| 3 | 2 | 2576 | 225000 |
| 4 | 3 | 2480 | 225000 |
| 5 | 1 | 2440 | 204999 |
| 3 | 2 | 2400 | 75000 |
| 5 | 3 | 2367 | 139000 |
| … | … | … | … |

**Figure 1**: Snapshot: housing data-set (sorted by area in descending order).

**Notation**: We will typically denote an input variable [2] by the symbol $X$. If $X$ is a vector, its components can be accessed by subscripts $X_j$. Quantitative outputs will be denoted by $Y$, and qualitative outputs by $G$ (for the group). We use uppercase letters such as $X$, $Y$ or $G$ when referring to the generic aspects of a variable. Observed values are written in lowercase; hence the $i^{\text{th}}$ observed value of $X$ is written as $x_i$ (where $x_i$ is again a scalar or vector). Matrices are represented by bold uppercase letters; for example, a set of $N$ input p-vectors $x_i$, i = 1, …, $N$ would be represented by the $N \times p$ matrix **X**. In general, vectors will not be bold, except when they have $N$ components; this convention

distinguishes a p-vector of inputs $x_i$ for the $i^{th}$ observation from the N-vector $\mathbf{x}_j$ consisting of all the observations on variable $X_j$. Since all vectors are assumed to be column vectors, the $i^{th}$ row of $\mathbf{X}$ is $x_i^T$, the vector transpose of $x_i$.

So, in summary, in an $N \times p$ matrix $\mathbf{X}$, we will have N number of housing data (training data or observations) and p numbers of features. So, a column will be regarded as a (specific feature-based) vector. Based on Figure 1, we can say, column vector $X_3$ is having the areas for all the training samples.

Since all vectors are assumed to be column vectors, the $i^{th}$ row of $\mathbf{X}$ is $x_i^T$, the vector transpose of $x_i$ as mentioned before, it implies that a row in $\mathbf{X}$ having the feature(s) of a particular house is donated by $x_i^T$. For example, based on Figure 1, $x_4^T = [5 \quad 4 \quad 3500]$, when the 'vector' is, $x_4 = \begin{bmatrix} 5 \\ 4 \\ 3500 \end{bmatrix}$.

<u>For the moment, we can loosely state the learning task as follows:</u>
Given the value of an input vector $X$, make a good prediction of the output $Y$, denoted by $\hat{Y}$ (pronounced "y-hat"). If $Y$ takes values in $\Re$ then so should $\hat{Y}$; likewise for categorical outputs, $\hat{G}$ should take values in the same set $\mathcal{G}$ associated with $G$.

For a two-class $G$, one approach is to denote the binary coded target as $Y$ and then treat it as a quantitative output. The predictions $\hat{Y}$ will typically lie in [0, 1], and we can assign to $\hat{G}$ the class label according to whether $\hat{y} > 0.5$. This approach generalizes to $K$-level qualitative outputs as well.

We need data to construct prediction rules, often a lot of it. We thus suppose we have available a set of measurements $(x_i, y_i)$ or $(x_i, g_i)$, i = 1, …, $N$, known as the *training data*, with which to construct our prediction rule.

---

**Note**:
A system of linear equations can be expressed in Matrix form. Say the given equations are:

$$x + 2y = 4$$
$$2x - 7y = -3$$

we can present them compactly as: $\boldsymbol{Ax} = \boldsymbol{b}$

where, $\boldsymbol{A} = \begin{bmatrix} 1 & 2 \\ 2 & -7 \end{bmatrix}$, $\boldsymbol{x} = \begin{bmatrix} x \\ y \end{bmatrix}$ *and* $\boldsymbol{b} = \begin{bmatrix} 4 \\ -3 \end{bmatrix}$

or, we may also write, $\begin{bmatrix} 1 & 2 \\ 2 & -7 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 4 \\ -3 \end{bmatrix}$

Also, the linear system can be expressed as a single vector equation:

$$x\begin{bmatrix} 1 \\ 2 \end{bmatrix} + y\begin{bmatrix} 2 \\ -7 \end{bmatrix} = \begin{bmatrix} 4 \\ -3 \end{bmatrix}$$

---

Based on output types, prediction tasks can be classified as [2]:

Regression: when we predict quantitative outputs.
Classifications: when we predict qualitative outputs.

However, they are more in common rather than different. Both are doing 'function approximations'.

Inputs can also have quantitative and qualitative types. Qualitative variables are typically represented numerically by codes, such as [0, 1] for yes/no or, true/false or, success/failure. We can sometimes label the output [-1, 1] instead of [0, 1].

A K-level qualitative variable is represented by a vector of K-binary variables or K-bits. For, example, in a disordered protein predictor using the neural network we want to output 3 different detections: Ordered, Semi-disordered and Disordered and their label can be code as [001, 010, 100] – this coding is called, *one-hot encoding,* which is currently the most popular.

**Linear Models and Least Squares:** *Linear Regression*
Given a vector of inputs $X^T = (X_1, X_2, \ldots, X_p)$, we predict the output $Y$ via the model:

$$\hat{Y} = \hat{\beta}_0 + X_1\hat{\beta}_1 + X_2\hat{\beta}_2 + X_3\hat{\beta}_3 + \cdots + X_p\hat{\beta}_p \qquad (1)$$

In the context of our housing data, the $X_i$ is the $i^{th}$ feature of the available observed or training data. Here, $\hat{\beta}_i$ is the parameter or weight or coefficient of the linear equation. Our job is to determine the best value of $\hat{\beta}_i$.

Equation (1) can be summarized as,

$$\hat{\boldsymbol{Y}} = \hat{\boldsymbol{\beta}}_{\boldsymbol{0}} + \sum_{\boldsymbol{j=1}}^{\boldsymbol{p}} \boldsymbol{X_j}\hat{\boldsymbol{\beta}}_{\boldsymbol{j}} \quad (2)$$

Note: Compare the equation #2 with, $y = mx + c$

The term $\hat{\beta}_0$ is the intercept, also known as the *bias* in machine learning. Often it is convenient to include the constant variable 1 in $X$, include $\hat{\beta}_0$ in the vector of coefficients $\hat{\beta}$, and then write the linear model in vector form as an inner product. To clarify, we can see Equation # 1 as:

$$\hat{Y} = X_0\hat{\beta}_0 + X_1\hat{\beta}_1 + X_2\hat{\beta}_2 + X_3\hat{\beta}_3 + \cdots + X_p\hat{\beta}_p \quad (3)$$

where $X_0$=1 is assumed.

Finally,

$$\hat{\boldsymbol{Y}} = \boldsymbol{X^T}\hat{\boldsymbol{\beta}} \quad (4)$$

where, $\hat{\beta} \in \Re^{p+1}$ now. In the (p+1)-dimensional input-output space, ($X$, $Y$ ) represents a hyperplane. A hyperplane of a (p+1)-dimensional space is a flat subset with dimension p.

Note: think about why $X^T$?

Here, $X^T$ denotes vector or matrix transpose ($X$ being a column vector).

How do we fit the linear model to a set of training data? There are many different methods, but by far the most popular is the method of *least squares*. In this approach, we pick the coefficients $\beta$ to minimize the residual sum of squares (RSS), also known as a cost function in another context.

$$RSS(\beta) = \sum_{i=1}^{N} \left(y_i - x_i^T \beta\right)^2 \quad (5)$$

Our target for equation # 5 is to obtain a suitable value of $\beta$ so that the *RSS* ($\beta$) is minimized, i.e.:

$$\min_{\beta} RSS(\beta)$$

Question: Does the minimum exist? What method(s) can we use for minimization? Let us answer the 2nd question first (See Note next).

---

**Note**:

We can use Newton's (or, Newton-Raphson) methods, we can use Gradient descent, we can use Genetic Algorithms, etc.

For the aforementioned methods, we start them with some initial guess(es) (value(s)) and iteratively try to find the minima.

Let us discuss how the methods work here.

(a) **<u>Newton's method</u>**:

Say, we have an equation $f(x)=0$.

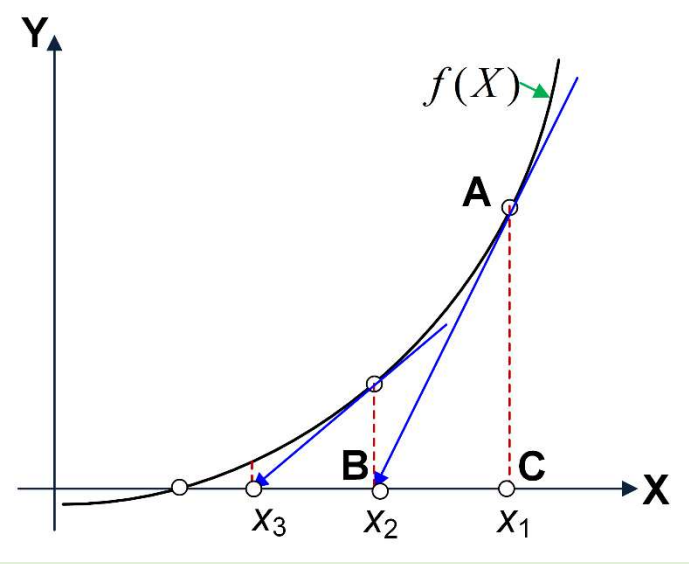



Figure (a): How Newton's method works in finding the solution.

For the solution (i.e., to find for what value of $x$, $f(x) = 0$), assume our initial point is $x_1$ and, X = $x_1$ intersects $x$-axis at C and $f(x)$ at A (see Figure (a)). Also, assume that the tangent at A intersects the $x$-axis at B, where the value of $x$ is $x_2$. From ΔABC and the definition of the slope of an equation, we can write:

$$f'(x_1) = \frac{f(x_1)-0}{x_1-x_2} \quad \text{(i)}$$

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} \quad \text{(ii)}$$

In general, we can write,

$$x_{t+1} = x_t - \frac{f(x_t)}{f'(x_t)} \quad \text{(iii)}$$

Equation (iii) can be iteratively applied to get the solution of the equation.

Are we after the solution of an equation or the minimization?

Minimization: if minimum exists then, we need to find, the value of $x$ for which $f'(x) = 0$.

We can think of going for the solution of the equation, $f'(x) = 0$ using (iii).

Therefore, we can similarly write,

$$x_{t+1} = x_t - \frac{f'(x_t)}{f''(x_t)} \quad \text{(iv)}$$

Now, we can use equation (iv) to find the minimum (or, maximum) of equation $f(x) = 0$.

**Caution**: For a function of more than one variables (in such case the derivatives are called partial derivatives), say $f(x_1, x_2, \dots) = 0$, in each iteration, the updating of each of the individual variables have to be done simultaneously, i.e., the assignment of $x_i(t+1)$ from $x_i$ for $\forall i$ must be simultaneous.

(b) **Gradient Descent**

Gradient descent is a simple equation to get the nearest local minima. It starts at a point $x_0$ and moves from $x_t$ (for the 1st point t=0) to $x_{t+1}$ by minimizing along the line extending from $x_t$ in the direction of negative of the gradient at $x_t$, i.e., $-\nabla f(x_t)$. The equation is written as:

$$x_{t+1} = x_t - \alpha \nabla f(x_t) \quad \text{(v)}$$

where, $\alpha$ is used to control the step size (also called the learning rate), and $\alpha > 0$. When it reaches a local minimum, the term $\nabla f(x_t)$ becomes 0, therefore equation (v) becomes $x_{t+1} = x_t$.

These methods return the local minimum unless there is only one global minimum.

(c) **Genetic Algorithm**

Genetic Algorithm (GA) is a population-based optimization algorithm. The formation was inspired by natural evolution. A pseudo-code for GA is given below:

1. Form the initial population (usually random)
2. Compute the fitness to evaluate each chromosomes (member of them population)
3. Select pairs to mate from best-ranked individuals and replenish population
   a. - Apply crossover operator
   b. - Apply mutation operator
4. Check for termination criteria, else go to step #2

Figure (b): Pseudocode for Genetic Algorithm

GA can get global minima, however it is not always assured. [**Note Ends**]

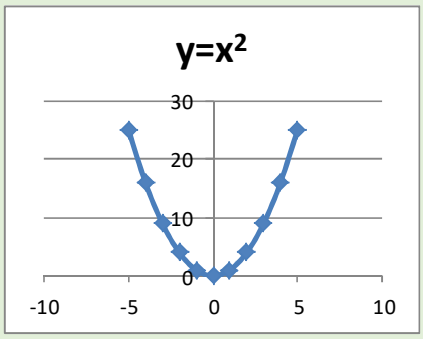
---

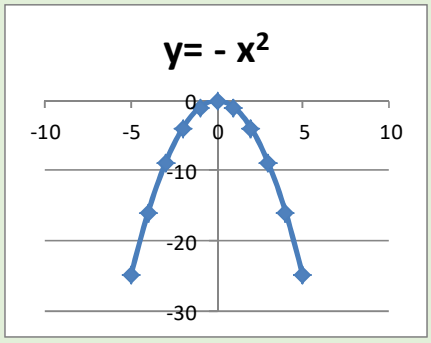
Now, let us answer the first question (Does the minimum exist for Equation (5)?) Since Equation (5) is a quadratic function of parameters, therefore its minimum always exists. How?

---

**Note**: Answering How?

Say, $f(x) => ax^2 + bx + c = 0$ is a quadratic equation and $a \neq 0$,
Now if $a > 0$ then the equation has a minimum point [compare $y = x^2 + \ldots$]
and if $a < 0$ then the equation has a maximum point [compare $y = -x^2 + \ldots$]




Finally, the RHS of equation (5) being in the whole square ensures that the coefficient of the (quadratic) variable is always positive and hence the minimum exists.

To find, $\min_{\beta} RSS(\beta)$, we can use Newton's methods as:

$$\beta_j(t+1) = \beta_j(t) - \frac{\frac{\partial}{\partial \beta_j} RSS(\beta)}{\frac{\partial}{\partial \beta_j}\left(\frac{\partial}{\partial \beta_j} RSS(\beta)\right)} \qquad (6)$$

For gradient descent the corresponding equation should be:

$$\beta_j(t+1) = \beta_j(t) \; - \; \alpha \; \frac{\partial}{\partial \beta_j} RSS(\beta) \qquad (7)$$

Here, to simplify further we can do the following:

We had: $$RSS(\beta) = \sum_{i=1}^{N} \left(y_i - x_i^T \beta\right)^2$$

To accommodate and to differentiate index $i$ from index $j$, where $j = \{0, 1, 2, \ldots, p\}$ to present each of the components of $\beta$, we rewrite the above equation as:

$$RSS(\beta) = \sum_{i=1}^{N} \left(y(i) - x^T(i)\, \beta\right)^2$$

$$\frac{\partial}{\partial \beta_j} RSS(\beta) = \frac{\partial}{\partial \beta_j} \sum_{i=1}^{N} \left(y(i) - x^T(i)\, \beta\right)^2$$

$$= 2\sum_{i=i}^{N} \left(y(i) - x^T(i)\,\beta\right).\frac{\partial}{\partial\beta_j}\left(y(i) - x^T(i)\,\beta\right)$$

$$= 2\sum_{i=1}^{N} \left(y(i) - x^T(i)\,\beta\right).\frac{\partial}{\partial\beta_j}\left(y(i) \; - x(i)_0\beta_0 - x(i)_1\beta_1 - \ldots - x(i)_j\beta_j - \ldots\right)$$

$$= 2\sum_{i=1}^{N} \left(x^T(i)\,\beta - y(i)\right).x(i)_j$$

Thus, $\frac{\partial}{\partial\beta_j}\left(\frac{\partial}{\partial\beta_j}RSS(\beta)\right) = \frac{\partial}{\partial\beta_j}\left(2\sum_{i=1}^{N} \left(x^T(i)\,\beta - y(i)\right).x(i)_j\right)$

$$= 2\frac{\partial}{\partial\beta_j}\sum_{i=1}^{N} \left(\left(x(i)_0\beta_0 + x(i)_1\beta_1 + \ldots + x(i)_j\beta_j + \ldots - y(i)\right).x(i)_j\right)$$

$$= 2\sum_{i=1}^{N} \left(x(i)_j\,.\,x(i)_j\right)$$

Using this we can simplify Equation (6) derived from Newton's method as:

$$\beta_j(t+1) = \beta_j(t) - \frac{\sum_{i=1}^{N} \left(x^T(i)\,\beta - y(i)\right).x(i)_j}{\sum_{N} \left(x(i)_j\,.\,x(i)_j\right)} \qquad (8)$$

Similarly from Equation (7) we can write for gradient descent:

$$\beta_j(t+1) = \beta_j(t) + \frac{2\alpha}{N}\sum_{i=1}^{N} \left(y(i) \; - x^T(i)\,\beta\right).x(i)_j \qquad (9)$$

Note: in Equation (8) and (9), $\frac{1}{N}$ is used to take the average error, but in (8)'s case the numerator and denumerator will have $\frac{1}{N}$ and will cancel each other.

Alternatively, we can use GA where, we pick random value(s) for (vector) $\beta$ and iterate for the values that minimize $RSS(\beta)$.

For all of the three iterative methods, the value of $j = \{0, 1, 2, \ldots, p\}$ and $i = \{1, 2, \ldots, N\}$. Each of the individual values of $\beta$ will require to be updated simultaneously (at least to be a correct approach theoretically).

**Exact Algorithm / Non-Iterative Algorithm:**
Continuing from Equation (5), we can write

$$\text{RSS}(\beta) = (\mathbf{y} - \mathbf{X}\beta)^T (\mathbf{y} - \mathbf{X}\beta) \tag{10}$$

where **X** is an $N \times p$ matrix with each row an input vector, and **y** is an $N$-vector of the outputs in the training set. Differentiating w.r.t. β we get the ***normal equations***: (How?)

$$\mathbf{X}^T (\mathbf{y} - \mathbf{X}\beta) = 0 \tag{11}$$

---

**Note**: Here is 'how' we get (11) from (10):

$$\begin{aligned} RSS(\beta) &= (\mathbf{y} - \mathbf{X}\beta)^T (\mathbf{y} - \mathbf{X}\beta) \\ &= [\mathbf{y}^T - (\mathbf{X}\beta)^T](\mathbf{y} - \mathbf{X}\beta) \qquad [\because (A \pm B)^T = A^T \pm B^T,\ (AB)^T = B^T A^T] \\ &= (\mathbf{y}^T - \beta^T \mathbf{X}^T)(\mathbf{y} - \mathbf{X}\beta) \\ &= \mathbf{y}^T\mathbf{y} - \beta^T\mathbf{X}^T\mathbf{y} - \mathbf{y}^T\mathbf{X}\beta + \beta^T\mathbf{X}^T\mathbf{X}\beta \qquad \begin{bmatrix} \because a^T b = b^T a \\ \therefore \mathbf{y}^T\mathbf{X}\beta = (\mathbf{X}\beta)^T\mathbf{y} = \beta^T\mathbf{X}^T\mathbf{y} \end{bmatrix} \\ &= \mathbf{y}^T\mathbf{y} - \beta^T\mathbf{X}^T\mathbf{y} - \beta^T\mathbf{X}^T\mathbf{y} + \beta^T\mathbf{X}^T\mathbf{X}\beta \\ &= \mathbf{y}^T\mathbf{y} - 2\beta^T\mathbf{X}^T\mathbf{y} + \beta^T\mathbf{X}^T\mathbf{X}\beta \end{aligned}$$

Therefore, we have

$$RSS(\beta) = \mathbf{y}^T\mathbf{y} - 2\beta^T\mathbf{X}^T\mathbf{y} + \beta^T\mathbf{X}^T\mathbf{X}\beta$$

Now, differentiating w.r.t. β and equating it to zero, we get:

$$\frac{\partial}{\partial\beta}[\mathbf{y}^T\mathbf{y} - 2\beta^T\mathbf{X}^T\mathbf{y} + \beta^T\mathbf{X}^T\mathbf{X}\beta] = 0$$

$$\Rightarrow \frac{\partial}{\partial\beta}(\mathbf{y}^T\mathbf{y}) - 2\frac{\partial}{\partial\beta}(\beta^T)\mathbf{X}^T\mathbf{y} + \frac{\partial}{\partial\beta}(\beta^T\mathbf{X}^T\mathbf{X}\beta) = 0$$

$$\Rightarrow 0 - 2\mathbf{X}^T\mathbf{y} + \frac{\partial}{\partial\beta}(\beta^T)\mathbf{X}^T\mathbf{X}\beta + \beta^T\mathbf{X}^T\mathbf{X}\frac{\partial}{\partial\beta}(\beta) = 0$$

$$\Rightarrow 0 - 2\mathbf{X}^T\mathbf{y} + \mathbf{X}^T\mathbf{X}\beta + \beta^T\mathbf{X}^T\mathbf{X} = 0 \qquad \begin{bmatrix} \because \beta^T\mathbf{X}^T\mathbf{X} = \beta^T(\mathbf{X}^T\mathbf{X}) \\ = (\mathbf{X}^T\mathbf{X})^T\beta = (\mathbf{X}^T)(\mathbf{X}^T)^T\beta \\ = \mathbf{X}^T\mathbf{X}\beta \end{bmatrix}$$

$$\Rightarrow -2\mathbf{X}^T\mathbf{y} + 2\mathbf{X}^T\mathbf{X}\beta = 0$$

$$\Rightarrow -2\mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta) = 0$$

$\therefore \mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta) = 0$

Suggestion: If needed have a quick review of the 'matrix cookbook'.

---

If $\mathbf{X}^T\mathbf{X}$ is **nonsingular** (or, invertible or, det $(\mathbf{X}^T\mathbf{X}) \neq 0$) then the unique solution is given by

$$\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\,\mathbf{X}^T\,\mathbf{y} \qquad (12)$$

This normal equation (Equation (12)) will be very useful.

---

**Note**: How to compute the inverse of a Matrix A?

$$A^{-1} = \frac{1}{\det(A)}(\text{Cofactor matrix of } A)^T$$

Example: $A = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 1 & 0 & 6 \end{bmatrix}$, then $A_{11} = \begin{vmatrix} 4 & 5 \\ 0 & 6 \end{vmatrix} = 24$, $A_{12} = -\begin{vmatrix} 0 & 5 \\ 1 & 6 \end{vmatrix} = 5$,

$A_{13} = \begin{vmatrix} 0 & 4 \\ 1 & 0 \end{vmatrix} = -4$, …

So, the cofactor-matrix becomes $\begin{vmatrix} 24 & 5 & -4 \\ -12 & 3 & 2 \\ -2 & -5 & 4 \end{vmatrix}$ and its transpose is =

$\begin{vmatrix} 24 & -12 & -2 \\ 5 & 3 & -5 \\ -4 & 2 & 4 \end{vmatrix}$. Now, the determinant of A, or, det(A) = 22.

Therefore, $A^{-1} = \frac{1}{22}\begin{vmatrix} 24 & -12 & -2 \\ 5 & 3 & -5 \\ -4 & 2 & 4 \end{vmatrix}$ = … (can multiply each element by 1/22).

---

Let us now see a practical example where we can use what we have learned (especially equation #12 (the normal equation)).

**Programming/ Programming Exercise**:

For the housing dataset, see "Moodle/Chapter #/Addition_Information/HousingData/". Inside the folder, there are two files: X.txt and Y.txt.

The columns in the X.txt are the input feature vectors. Note, that the starting column is the $X_0$ vector, which is from the theory and the value is all 1s.

Y.txt is the price or the output column. We are to compute β, which is the parameter of the linear equation, i.e.,

$$\hat{Y} = \beta_0 + x_1\beta_1 + x_2\beta_2 + x_3\beta_3$$

Once we know the values of β, we can put our desired feature values for $x_1$ (# of bedroom), $x_2$ (# of bathroom), $x_3$ (sqft area) to check the estimated price from the above equation.

## Steps:

1. You can use octave (or, Matlab):
2. I made a separate directory c:\octave_work\
3. Place the file X.txt and Y.txt in c:\octave_work\
4. Inside Octave, run the codes:

```
>cd 'c:\octave_work\'  % change the current directory to this
> load X.txt          % this will load the data from X.txt into X
> X                   % just to check the content of X
> load Y.txt          % this will load the data from Y.txt into Y
> Y                   % checking the content of X
> B=inv(X'*X)*X'*Y % this is the normal Eqn (#12)
 B =                          % Note: X' is X-transpose.
                              % inv is the Matrix inverse.
  3.3868e+004
 -3.6762e+004
  1.0501e+004
  1.3291e+002
```

These four values are the answers to our parameter $\beta$ for the equation:

$\hat{Y} = \beta_0 + x_1\beta_1 + x_2\beta_2 + x_3\beta_3$.

**Prediction**: For a Query, Q=[1 5 3 2500] for example, we can predict the house price by:

Q*B;  [which should return 2.1384e+005 or, 213,840]

**Note**: **Q** is in row form and **B** is in column form. Thus, we do Q*B instead of Q' *B.

**Important Note**: If $\mathbf{X}^T\mathbf{X}$ is **singular**, then function ***inv()*** does not work. In such a case, we can use ***pinv()*** function, which is a **pseudoinverse** function. That is, we write B=pinv(X'*X)*X'*Y instead of B=inv(X'*X)*X'*Y.

# Part 2

# Regularization and Model Selection

Given the dataset for regression, we can use models from linear to higher-order polynomial. However, here, we will focus on:

- How to determine the best model for the given problem, and
- How to take care of the overfitting problem to receive the benefit of the higher-order polynomials.

Let us start with a simple case:

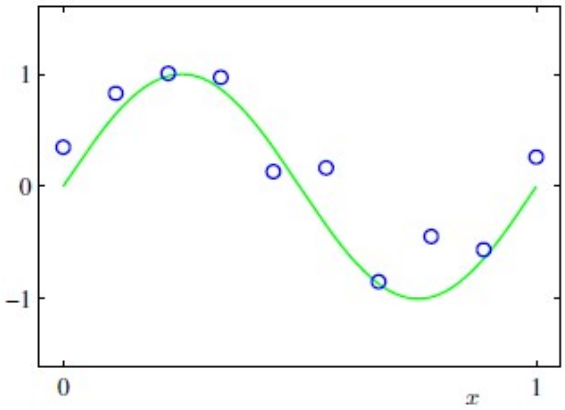



**Figure 1**: Plot of a training data set of $N = 10$ points, shown as blue circles, each comprising an observation of the input variable $x$ along with the corresponding target variable $y$. The green curve shows the function $sin\ (2\pi x)$ used to generate the data. Our goal is to predict the value of $y$, for some new value of $x$, without knowledge of the green curve [1].

Assume we have 10 training points as in Figure 1, and we are trying to fit various order of polynomial as in Figure 2.

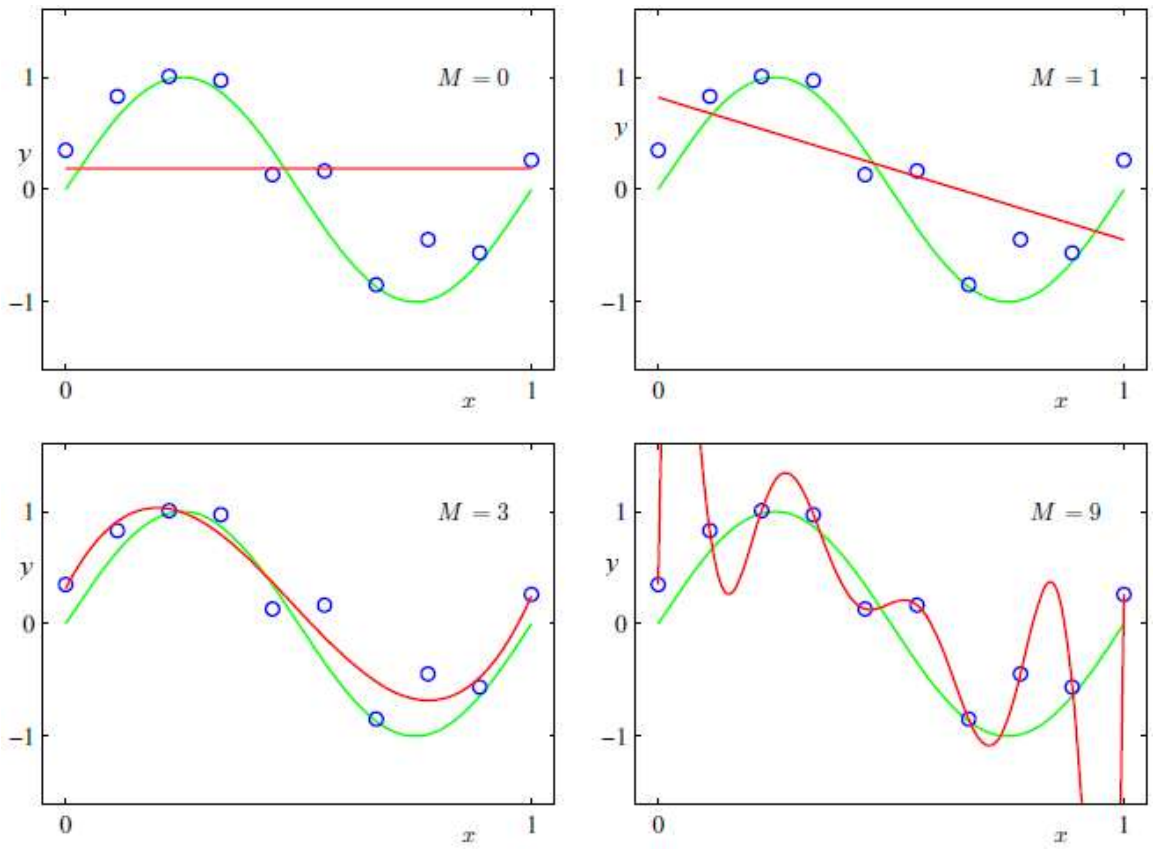



**Figure 2**: Plots of polynomials having various orders M, shown as red curves, fitting to the data set shown here.

There remains the problem of choosing the order *M* of the polynomial, and as we shall see, this will turn out to be an example of an important concept called *model selection*. In Figure 2, we show four examples of the results of fitting polynomials having orders *M* = 0, 1, 3 and 9 to the data set shown in Figure 1. We give the general Equation as:

$$\hat{y}(x,\beta) = \beta_0 + \beta_1 x + \beta_2 x^2 + \ldots + \beta_M x^M = \sum_{i=0}^{M} \beta_i x^i \qquad (1)$$

We notice that the constant (*M* = 0) and first-order (*M* = 1) polynomials give rather poor fits to the data and consequently rather poor presentations of the function $sin(2\pi x)$. The third-order (*M* = 3) polynomial seems to provide the best fit to the function $sin(2\pi x)$ of the examples shown in Figure 2. When we got to a much higher order polynomial (*M* = 9), we obtain an excellent fit to the training data. In fact,

the polynomial passes precisely through each data points and the *residual sum of squares* (RSS) must be 0. However, the fitted curve oscillates wildly and gives a very poor representation of the function $sin(2\pi x)$. This latter behavior is known as *overfitting*.

Let us define the root-mean-square (RMS) error defined by

$$E_{RMS} = \sqrt{RSS(\beta)/N} \qquad (2)$$

in which the division by $N$ allows us to compare different sizes of data sets on an equal footing, and the square root ensures that $E_{RMS}$ is measured on the same scale (and in the same units) as the target variable $\hat{y}$. Graphs of the training and test set RMS errors are shown, for various values of M, in Figure 3.

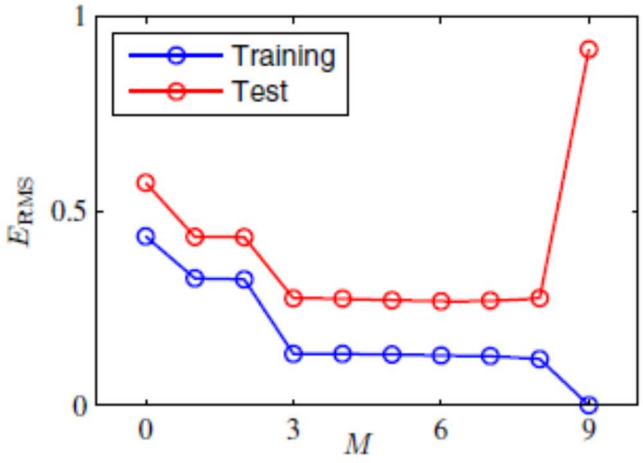



**Figure 3**: Graphs of the root-mean-square error, defined by (Eq$^{n}$ 2), evaluated on the training set and on an independent test set for various values of M.

The test set error is a measure of how well we are doing in predicting the values of $\hat{y}$ for new data observations of $x$. We note from Figure 3 that small values of $M$ give relatively large values of test set error, and this can be attributed to the fact that the corresponding polynomials are rather inflexible and are incapable of capturing the oscillations in the function $sin(2\pi x)$. Values of $M$ in the range $3 \le M \le 8$ give small values for the test set error, and these also offer reasonable representations of the generating function $sin(2\pi x)$, as can be seen, for the case of $M = 3$, from Figure 2.

For M = 9, the training set error goes to zero, as we might expect because this polynomial contains 10 degrees of freedom corresponding

to the 10 coefficients $\beta_0, \beta_1, \beta_2 ..., \beta_9$ and so can be tuned precisely to the 10 data points in the training set. However, the test set error has become very large and, as we saw in Figure 2, the corresponding function (Equation 1 with $M = 9$) exhibits wild oscillations.

This may seem paradoxical because a polynomial of given order contains all lower-order polynomials as special cases. The $M = 9$ polynomial is therefore capable of generating results at least as good as the $M = 3$ polynomial. Furthermore, we might suppose that the best predictor of new data would be the function $sin(2\pi x)$ from which the data was generated (and this is indeed the case). We know that a power series expansion of the function $sin(2\pi x)$ contains terms of all orders, so we might expect that results should improve monotonically as we increase $M$.

**Table 1**: Table of the coefficients $\beta$ for polynomials of various order. Observe how the typical magnitude of the coefficients increases dramatically as the order of the polynomial

| | $M = 0$ | $M = 1$ | $M = 3$ | $M = 9$ |
|---|---|---|---|---|
| $\beta_0$ | 0.19 | 0.82 | 0.31 | 0.35 |
| $\beta_1$ | | -1.27 | 7.99 | 232.37 |
| $\beta_2$ | | | -25.43 | -5321.83 |
| $\beta_3$ | | | 17.37 | 48568.31 |
| $\beta_4$ | | | | -231639.30 |
| $\beta_5$ | | | | 640042.26 |
| $\beta_6$ | | | | -1061800.52 |
| $\beta_7$ | | | | 1042400.18 |
| $\beta_8$ | | | | -557682.99 |
| $\beta_9$ | | | | 125201.43 |

We can gain some insight into the problem by examining the values of the coefficients $\beta$ obtained from polynomials of various order, as shown in *Table 1*. We see that, as $M$ increases, the magnitude of the coefficients typically gets larger. In particular, for the $M = 9$ polynomial, the coefficients have become finely tuned to the data by developing large positive and negative values so that the corresponding polynomial function matches each of the data points exactly, but between data points (particularly near the ends of the range) the function exhibits the large oscillations observed in Figure 2. Intuitively, what is happening is that the more flexible polynomials with larger values of $M$ are becoming increasingly tuned to the random noise on the target values.

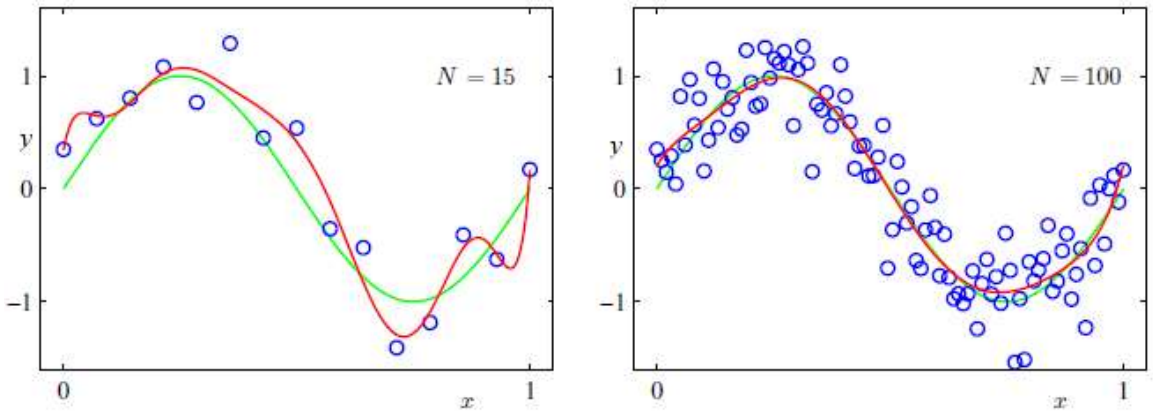



**Figure** 4: Plots of the solutions obtained by minimizing the sum-of-squares error function using the $M$ =9 polynomial for $N$ = 15 data points (left plot) and $N$ = 100 data points (right plot). We see that increasing the size of the data set reduces the overfitting problem.

It is also interesting to examine the behavior of a given model as the size of the data set is varied, as shown in Figure 4. We see that, for given model complexity, the overfitting problem becomes less severe as the size of the data set increases. Another way to say this is that the larger the data set, the more complex (in other words, more flexible) the model that we can afford to fit the data. One rough heuristic that is sometimes advocated is that the number of data points should be no less than some multiple (say 5 or 10) of the number of adaptive parameters in the model.

We continue here, assuming that we have data sets of limited size and we wish to use relatively complex and flexible models. One technique that is often used to control the overfitting phenomenon in such cases is that of ***regularization***, which involves adding a penalty term to the error function (see Equation 3) in order to discourage the coefficients from reaching large values. The simplest such penalty term takes the form of a sum of squares of all of the coefficients, leading to a modified error function of the form

$$RSS(\beta) = \sum_{i=1}^{N}\{(\hat{y}(x_i,\beta) - y_i)^2\} + \lambda \sum_{j=1}^{p} \beta_j^2 \qquad (3)$$

the coefficient $\lambda$ governs the relative importance of the regularization term compared with the sum-of-squares error term. Note that often the coefficient $\beta_0$ is omitted from the regularizer because its inclusion causes the results to depend on the choice of origin for the target variable, or it

may be included but with its own regularization coefficient. Again, the error function in Equation 3 can be minimized precisely in closed form. Techniques such as this are known in the statistics literature as ***shrinkage*** methods because they reduce the value of the coefficients. The particular case of a quadratic regularizer is called ***ridge regression***. In the context of neural networks, this approach is known as ***weight decay*** [2].

Figure 5 shows the results of fitting the polynomial of order $M = 9$ to the same data set as before but now using the regularized error function given by Eq$^{n}$ 3.

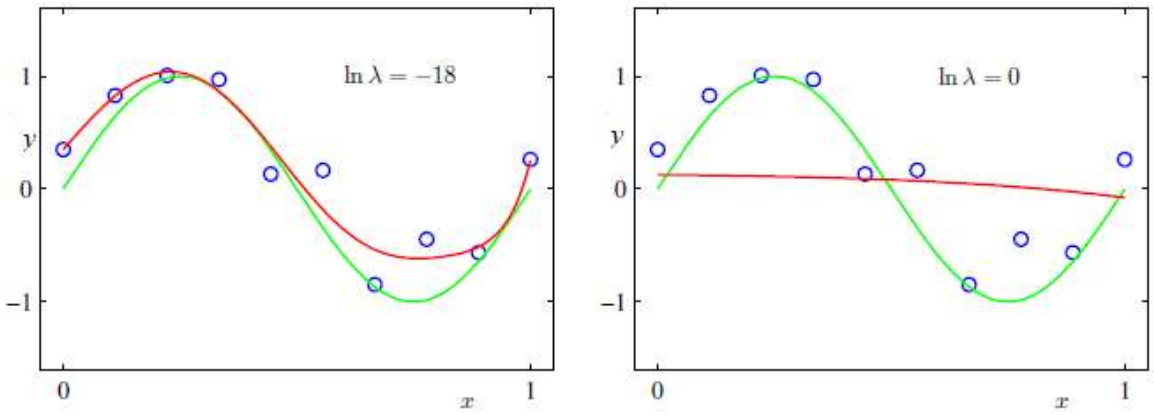



**Figure 5**: Plots of $M = 9$ polynomials fitted to the data set shown in Figure 1 using the regularized error function (Eq$^{n}$ 3) for two values of the regularization parameter $\lambda$ corresponding to $\ln\lambda$ = -18 and $\ln\lambda$ =0. The case of no regularizer, i.e. $\lambda$=0, corresponding to $\ln\lambda = -\infty$, is shown at the bottom right of Figure 2.

We see that, for a value of $\ln\lambda$ = -18, the overfitting has been suppressed and we now obtain a much closer representation of the underlying function $sin(2\pi x)$. If, however, we use too large a value for $\lambda$ then we again obtain a poor fit, as shown in Figure 5 for $\ln\lambda$=0. The corresponding coefficients from the fitted polynomials are given in Table 2, showing that regularization has the desired effect of reducing the magnitude of the coefficients.

**Table 2**: Table of the coefficients $\beta$ for M = 9 polynomials with various values for the regularization parameter $\lambda$. Note that $\ln \lambda = -\infty$ corresponds to a model with no regularization, i.e., to the graph at the bottom right in Figure 2. We see that, as the value of $\lambda$ increases, the typical

| | $\ln \lambda = -\infty$ | $\ln \lambda = -18$ | $\ln \lambda = 0$ |
|---|---|---|---|
| $\beta_0$ | 0.35 | 0.35 | 0.13 |
| $\beta_1$ | 232.37 | 4.74 | -0.05 |
| $\beta_2$ | -5321.83 | -0.77 | -0.06 |
| $\beta_3$ | 48568.31 | -31.97 | -0.05 |
| $\beta_4$ | -231639.30 | -3.89 | -0.03 |
| $\beta_5$ | 640042.26 | 55.28 | -0.02 |
| $\beta_6$ | -1061800.52 | 41.32 | -0.01 |
| $\beta_7$ | 1042400.18 | -45.95 | -0.00 |
| $\beta_8$ | -557682.99 | -91.53 | 0.00 |
| $\beta_9$ | 125201.43 | 72.68 | 0.01 |

The impact of the regularization term on the generalization error can be seen by plotting the value of the RMS error (Eq$^n$ 2) for both training and test sets against $\ln \lambda$, as shown in Figure 6. We see that in effect $\lambda$ now controls the effective complexity of the model and hence determines the degree of overfitting.

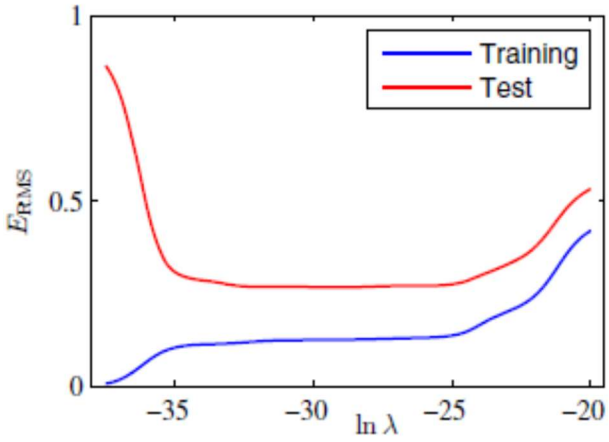



**Figure 6**: Graph of the root-mean-square error (Eq$^n$ 2) versus $\ln \lambda$ for the $M = 9$ polynomial.

The issue of model complexity is an important one, and here we simply note that, if we were trying to solve a practical application using this approach of minimizing an error function, we would have to find a way to determine a suitable value for the model complexity. The results above suggest a simple way of achieving this, namely by taking the available data and partitioning it into a training set, used to determine

the coefficients $\beta$, and a separate validation set also called a ***hold-out set***, used to optimize the model complexity (either $M$ or $\lambda$).

Now, let us discuss Equation 3 to get an idea about the $\lambda$. As usual, the target of Equation 3 is to minimize the error, i.e. we can rewrite the equation as:

$$\underset{min}{RSS_\lambda(\beta)} = \sum_{i=1}^{N}\{(\hat{y}(x_i,\beta) - y_i)^2\} + \lambda\sum_{j=1}^{p}\beta_j^2 \qquad (4)$$

Obviously, for the target to have a minimum error, the $\lambda\sum_{j=1}^{p}\beta_j^2$ term has to adjust such a way that if we set the value $\lambda$ too large then the values of $\beta$ will be too low and other way around. For a lower value of $\beta$ the predictor would be simpler and with a higher value of $\beta$ the predictor would be more complex and flexible.

To apply, we need to see the effect of adding the regularization term in the minimization equations. For Newton's methods we derived, we had:

$$\beta_j(t+1) = \beta_j(t) + \frac{\sum_{i=1}^{N}\left(y(i) - x^T(i)\,\beta\right)}{\sum_{i=1}^{N} x(i)_j . x(i)_j}$$

Similarly <u>involving regularization terms from Equation (4), we can have Newton's method</u> as:

$$\beta_j(t+1) = \beta_j(t) + \left[\frac{\sum_{i=1}^{N}\left(y(i) - x^T(i)\,\beta\right).x(i)_j - \lambda\,\beta(t)_j}{\sum_{i=1}^{N}\left[x(i)_j . x(i)_j\right] + \lambda}\right] \qquad (5)$$

Also, we had the *Gradient Descent* equation as:

$$\beta_j(t+1) = \beta_j(t) + \frac{2\alpha}{N}\sum_{i=1}^{N}\left(y(i) - x^T(i)\,\beta\right).x(i)_j$$

Similarly, for regularization, we can rewrite the gradient descent equation as:

$$\beta_j(t+1) = \beta_j(t) + \frac{2\alpha}{N}\left[\sum_{i=1}^{N} \left(y(i) - x^T(i)\,\beta\right).x(i)_j - \lambda\beta(t)_j\right] \quad (6)$$

For regularization, we usually do not treat $\beta_0$ in the same go (also mentioned before), therefore remember here that $j = \{1, 2, ..., p\}$.

Also, note that from Equation 6, we can write:

$$\beta_j(t+1) = \beta_j(t)(1 - \frac{2\alpha\lambda}{N}) + \frac{2\alpha}{N}\left[\sum_{i=1}^{N} \left(y(i) - x^T(i)\,\beta\right).x(i)_j\right]$$

Comparing the previous gradient descent (without regularization), we see the $\beta_j(t)$ is actually shrinking due to the term $(1 - \frac{2\alpha\lambda}{N})$ which is < 1 as $\alpha, \lambda$ and $N$ are positive quantities.

---

**Note:**

As we derived the term for differentiation in the previous part (Chapter #1) for the minimization equations such as newton's method and gradient descent approach. Here similarly, we can explain the derivation when regularization term is added.

$$\frac{\partial}{\partial\beta_j} RSS_\lambda(\beta) = \frac{\partial}{\partial\beta_j}\sum_{i=1}^{N}[\{y(i) - x(i)^T\beta\}^2] + \lambda\sum_{j=1}^{p}\beta_j^2$$

$$= \sum_{i=i}^{N}\left[2\left(y(i) - x^T(i)\,\beta\right).\frac{\partial}{\partial\beta_j}\left(y(i) - x^T(i)\,\beta\right) + \lambda\frac{\partial}{\partial\beta_j}(\beta_1^2 + \beta_2^2 + ... + \beta_j^2 + ...)\right]$$

$$= \sum_{i=1}^{N}\left[2\left(y(i) - x^T(i)\,\beta\right).\frac{\partial}{\partial\beta_j}\left(y(i) - x(i)_0\beta_0 - x(i)_1\beta_1 - ... - x(i)_j\beta_j - ...\right) + 2\lambda\beta_j\right]$$

$$= 2\sum_{i=1}^{N}\left[\left(x^T(i)\,\beta - y(i)\right).x(i)_j + \lambda\beta_j\right]$$

And, $\frac{\partial}{\partial \beta_j}\left(\frac{\partial}{\partial \beta_j} RSS_\lambda(\beta)\right) = 2\frac{\partial}{\partial \beta_j}\left(\sum_{i=1}^{N}\left[\left(x^T(i)\,\beta - y(i)\right).x(i)_j + \lambda\beta_j\right]\right)$

$$= 2\frac{\partial}{\partial \beta_j}\sum_{i=1}^{N}\left[\left(\left(x(i)_0\beta_0 + x(i)_1\beta_1 + \ldots + x(i)_j\beta_j + \ldots - y(i)\right).x(i)_j\right) + \lambda\beta_j\right]$$

$$= 2\sum_{i=1}^{N}\left[\left(x(i)_j\,.\,x(i)_j\right) + \lambda\right]$$

So, for <u>Newton's method,</u>

$$\beta_j(t+1) = \beta_j(t) - \frac{\frac{\partial}{\partial \beta_j} RSS_\lambda(\beta)}{\frac{\partial}{\partial \beta_j}\left(\frac{\partial}{\partial \beta_j} RSS_\lambda(\beta)\right)}$$

$$= \beta_j(t) - \frac{2\sum_{i=1}^{N}\left[\left(x^T(i)\,\beta - y(i)\right).x(i)_j + \lambda\beta_j\right]}{2\sum_{i=1}^{N}\left[\left(x(i)_j\,.\,x(i)_j\right) + \lambda\right]}$$

$$= \beta_j(t) - \frac{\sum_{i=1}^{N}\left(x^T(i)\,\beta - y(i)\right).x(i)_j + \lambda\beta_j}{\sum_{i=1}^{N}\left(x(i)_j\,.\,x(i)_j\right) + \lambda}$$

Finally, $\beta_j(t+1) = \beta_j(t) + \left[\frac{\sum_{i=1}^{N}\left(y(i) - x^T(i)\,\beta\right).x(i)_j - \lambda\,\beta(t)_j}{\sum_{i=1}^{N}\left[x(i)_j.x(i)_j\right] + \lambda}\right]$ => Eq$^n$(5)

Note: I have changed the minus sign inside and outside of the square bracket to look better.

Similarly, for the gradient descent method

$$\beta_j(t+1) = \beta_j(t) - \alpha\frac{\partial}{\partial \beta_j} RSS_\lambda(\beta)$$

$$= \beta_j(t) - 2\alpha\sum_{i=1}^{N}\left[\left(x^T(i)\,\beta - y(i)\right).x(i)_j + \lambda\beta_j\right]$$

Finally, $\beta_j(t+1) = \beta_j(t) + \frac{2\alpha}{N}\left[\sum_{i=1}^{N}\left(y(i) - x^T(i)\,\beta\right).x(i)_j - \lambda\beta(t)_j\right]$ => Eq$^n$(6)

**_____Note Ends___**

**Regularized Normal Equation**:

To recall, here is our normal equation derived before: $\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\,\mathbf{X}^T\,\mathbf{y}$.

To involve regularization, we rewrite the equation (derivation is long and thus avoided here):

$$\hat{\beta} = (\mathbf{X}^T\mathbf{X} + \lambda M_\lambda)^{-1}\,\mathbf{X}^T\,\mathbf{y} \qquad (7)$$

Here $M_\lambda$ is a ($p$+1) by ($p$+1) matrix with all the elements assigned to '0', except the diagonal elements. The diagonal elements are assigned to '1'. The only exception is that the first diagonal element, i.e., $M_\lambda(1,1) = 0$. Remember, here $p$ is the number of features in our data or the number of parameters in our equation (not including $\beta_0$).

The good news about equation (7) is: the regularization term takes care of the non-inevitability issue (if it exists).

---

**Question**: How would you use the exact or normal equation for a higher order polynomial?

**Ans**: Recall our housing example and data. Let us assume that we want to involve features: (1) Number of Beds ($x_1$) and (2) Living Areas ($x_2$) and we want to fit a second-order polynomial. Our equation should look like:

$$\hat{Y} = \beta_0 x_0 + \beta_1 x_1 + \beta_2 x_1^2 + \beta_3 x_2 + \beta_4 x_2^2$$

And, our matrix $\mathbf{X}$ of the equation $\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\,\mathbf{X}^T\,\mathbf{y}$ or, $\hat{\beta} = (\mathbf{X}^T\mathbf{X} + \lambda M_\lambda)^{-1}\,\mathbf{X}^T\,\mathbf{y}$ will look like, for example:

$$X = \begin{matrix} x_0 = 1 & x_1 & x_1^2 & x_2 & x_2^2 \\ \end{matrix} \begin{pmatrix} 1 & 3 & 9 & 2000 & 4000000 \\ 1 & 2 & 4 & 1500 & 2250000 \\ 1 & 1 & 1 & 1000 & 1000000 \\ 1 & 4 & 16 & 1200 & 1440000 \\ \dots & \dots & \dots & \dots & \dots \end{pmatrix}$$

$N \times (p+1)$

Here, in X, p = 4 (not 2).

We can use the developed linear model to fit nonlinear data. A simple way is to add powers of each feature as new features, then train a linear model on this extended set of features. This technique is called *Polynomial Regression* [3]. Thus, we can build the column $x_1^2$, and $x_2^2$ from $x_1$ and $x_2$ in excel or, once we load the matrix, we can manipulate the values in Octave/MATLAB.

We can use Scikit-Learn's PolynomialFeatures class to transform our training data, adding the square of each feature in the training set as a new feature.

For example, if you have $x_1$ loaded (e.g., by x1=X(:,2) MATLAB), you can generate $x_1^2$ by x1.^2. For the above example assume you have loaded x1 and x2 variables as column vectors, then

X=[ones (N,1), x1, x1.^2, x2, x2.^2], will form the **X** matrix.

*Model Selection*:
Assume we have a finite set of models, **M** = {$M_1$, $M_2$, …, $M_n$}. For example, we can have a set of the polynomial equation for an approximation of various orders, i.e., having a set of simple to complex equations. For simplicity, we consider $M_i$ is the $i^{th}$ order polynomial regression model. We need to discuss cross-validation next in order to go for model selection.

*Cross-Validation*:
To select a better model, we can think of using our dataset for the entire different models and then to check the average RSS to rank which

one to be picked, right? No, it is not that straightforward. For example, we saw that the higher-order polynomial can fit the data very well but on the other hand, that also generates higher error based on the test dataset. This is because the higher-order polynomial has low bias however they have very high variance. So, if we follow this path we will end up picking higher-order polynomial all the time and it will often be a poor choice.

Instead of the above approach, we can use, hold-out cross-validation or, simple cross-validation. We do the following simple steps:

**Algorithm: hold-out cross-validation**

1. Randomly split data (D): $D_{train}$ and $D_{holdout}$ (Say, 25-30% of the data).

2. For $\forall i$, train $M_i$ using $D_{train}$ and get the corresponding $P_i$ ($i^{th}$ predictor).

3. For $\forall i$, compute the error $E_i$ of $P_i$ using $D_{holdout}$.

4. Pick the predictor where the error is the lowest.

The disadvantage of the hold-out method is the wastage of the data, as we typically hold out 30% data to get the error reliably. Also, in many experiments, the data is not abundant. Further, we have seen that for higher-order polynomial, it is better to feed more data to get a reliable predictor. To avoid these issues, we follow k-fold cross-validation where we hold out fewer data. The steps are:

**Algorithm: k-fold cross-validation**

1. Randomly split data (D) into *k* disjoint subsets as: $D_1, D_2, ..., D_k$.

2. For $\forall i$, $M_i$ is evaluated as:
   For *j* =1 to k:
   Train the $M_i$ using data: ($D - D_j$) and Get predictor $P_{ij}$
   Test $P_{ij}$ using $D_j$ and get the error $E_{ij}$.
   EndFor *j*

$E_i$ = average of $E_{ij}$ for $\forall j$, which is the generalized error of $M_i$.

EndFor *i*

3. Pick the best model $M_i$ having the lowest generalized error $E_i$.
4. Re-train $M_{i=\text{best}}$ using full dataset D.

A typical choice of the value of k in such k-fold cross-validation is 10. We prefer to choose k to make the $1/k$ portion of the data to be a smaller fraction. This method is also called *leave-one-out cross*-validation.

**Best value selection for $\lambda$:**

How to obtain the best value of $\lambda$ (in regularization)?

1. Take a range of values with regular intervals: {0, …, R} of real or integer numbers for $\lambda$.
2. Evaluate the performance for each of the value of $\lambda$ using cross-validation approach for example.
3. Pick the best value of $\lambda$ for which the error was lowest.

**Note**: How did we obtain, $\hat{\beta} = (\mathbf{X}^T\mathbf{X} + \lambda M_\lambda)^{-1}\ \mathbf{X}^T\ \mathbf{y}$ ?

On page 11 we mentioned the "Regularized Normal Equation". Here we like to explain the derivation.

Review, how the derivation was done for the normal equation and we obtained $\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\ \mathbf{X}^T\ \mathbf{y}$.

To involve regularization we rewrite the equation:

$$\hat{\beta} = (\mathbf{X}^T\mathbf{X} + \lambda M_\lambda)^{-1}\ \mathbf{X}^T\ \mathbf{y} \qquad (7)$$

Here, $M_\lambda$ is a ($p$+1) by ($p$+1) matrix with all the elements assigned to '0', except the diagonal elements. The diagonal elements are assigned to '1'. The only exception is that the first diagonal element, i.e., $M_\lambda(1,1) = 0$. Remember, here $p$ is the number of features in our data or the number of parameters in our equation (not including $\beta_0$).

Here, we want to explain how we can get Equation (7) from (4). In vector form we can rewrite Equation (4) as follows:

$$\text{RSS}(\beta) = (y - X\beta)^T (y - X\beta) + \lambda\beta^T\beta$$

Now, we expand the equation as:

$\text{RSS}(\beta) = (y - X\beta)^T (y - X\beta) + \lambda\beta^T\beta$

$= (y^T - \beta^T X^T)(y - X\beta) + \lambda\beta^T\beta$ $[\because (\text{A} \pm \text{B})^\text{T} = \text{A}^\text{T} \pm \text{B}^\text{T}, (\text{AB})^\text{T} = \text{B}^\text{T}\text{A}^\text{T}]$

$= y^T y - \beta^T X^T y - y^T X\beta + \beta^T X^T X\beta + \lambda\beta^T\beta$ $[\because a^T b = b^T a, \therefore y^T X\beta = (X\beta)^T y = \beta^T X^T y]$

$= y^T y - \beta^T X^T y - \beta^T X^T y + \beta^T X^T X\beta + \lambda\beta^T\beta$

$= y^T y - 2\beta^T X^T y + \beta^T X^T X\beta + \lambda\beta^T\beta$

Therefore, we get the expanded form:

$$RSS(\beta) = y^T y - 2\beta^T X^T y + \beta^T X^T X\beta + \lambda\beta^T\beta$$

Now, differentiating $\text{RSS}(\beta) = y^T y - 2\beta^T X^T y + \beta^T X^T X\beta + \lambda\beta^T\beta$ with respect to $\beta$ and equating it to zero, we get:

$$\frac{\partial}{\partial\beta}(y^T y - 2\beta^T X^T y + \beta^T X^T X\beta + \lambda\beta^T\beta) = 0$$

$=> 0 - 2X^T y + \frac{\partial}{\partial\beta}(\beta^T)X^T X\beta + \beta^T X^T X\frac{\partial}{\partial\beta}(\beta) + \frac{\partial}{\partial\beta}(\lambda\beta^T I\beta) = 0$

[Here, $\text{M}_\lambda$ is the indetify martix, and we write $\lambda\beta^T\beta => \lambda\beta^T\text{M}_\lambda\beta$]

$=> 0 - 2X^T y + X^T X\beta + \beta^T X^T X + \lambda\text{M}_\lambda\beta + \lambda\beta^T\text{M}_\lambda = 0$

$[\because \beta^T X^T X = \beta^T(X^T X) = (X^T X)^T\beta = X^T X\beta)]$

$=> -2X^T y + 2X^T X\beta + 2\lambda\text{M}_\lambda\beta = 0$

$[\because \beta^T\text{M}_\lambda = \text{M}_\lambda{}^\text{T}\beta = \text{M}_\lambda\beta]$

$=> -2X^T y + 2\beta(X^T X + \lambda\text{M}_\lambda) = 0$

$=> X^T y = \beta(X^T X + \lambda\,\text{M}_\lambda)$

$=> \beta = (X^T X + \lambda\,\text{M}_\lambda)^T X^T y$ **[*Proved*]**

Here, further in the identity matrix $\text{M}_\lambda$, we assign $\text{M}_\lambda(1,1) = 0$ to keep $\beta_\text{o}$ out of the regularization effect.

**References**


[1] C. M. Bishop, *Pattern Recognition and Machine Learning*: Springer, 2009.

[2] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*: Springer, 2009.

[3] A. Géron, *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*, 2nd ed.: O'Reilly 2019.


-- x --