Chapter 3

Linear Regression

3.1 Learning Objectives

Our study of machine learning will start with linear regression. Linear regression is not only one of machine learning algorithms, but also, more importantly, a basic concept than will be applied or generalized to many other machine learning algorithms, such as logistic regression, neural networks and deep learning. After finishing this chapter, a reader should be able to

- Understand supervised learning setting
- Describe linear regression model based on gradient decent algorithm
- Apply linear regression model to make predictions
- Understand practical issues such as feature scaling, learning rate
- Use Python to implement linear regression from the scratch based on gradient decent algorithm
- Rapidly implement linear regression in Python using sklearn library

3.2 Mathematical Description of Linear Regression

3.2.1 Linear regression setting

Linear regression is one of the most basic algorithms in machine learning. Many fancy and popular machine learning algorithms can be viewed as generalizations or extensions of linear regression. Thus, it serves as a good starting point of machine learning. In a supervised learning problem, we are given a training dataset of m examples or instances of input-output pairs $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}) \dots (x^{(m)}, y^{(m)})$, where the superscript indicates the index of examples. Each input $x^{(i)}$ is a vector including n ***features*** (or ***attributes***, or ***predictors***). The output, often referred as the ***target*** or ***label***, will be assumed to be univariate $y^{(i)} \in \mathbb{R}$ in this chapter. The later chapters show that $y^{(i)}$ could be a vector in general. We would like to ***learn*** or find a model of how the inputs affect the outputs. Given a new input value x, we can make a prediction on its output as $\hat{y}(x)$ based on the model learned from the training dataset. A regression task is also called curve fitting in some texts. If the output y can be approximated by a linear combination of features, the supervised learning problem can be simplified to a linear regression problem.

3.2.2 Linear regression on a single feature

We will begin the linear regression with a single feature. Then the algorithm will be generalized to handle the input with multiple features in general, in the next section.

The linear regression on a single feature is to predict a target y on a single feature variable x, assuming that there is approximately a linear relationship between x and y, as shown in Fig.1. Mathematically, the relationship can be represented as

$$y = \theta_0 + \theta_1 x + \epsilon \tag{3.1}$$

where θ_0, θ_1 are the intercept and the slope for the assumed linear relationship. ϵ is the error term for a consideration of following facts: the true relationship is probably not linear, there are may be other variables that cause variation in y, and there may be measurement errors. Thus, the goal of a linear regression algorithm is to find the optimal values of parameters θ_0, θ_1 to fit the training data examples with a minimal error.

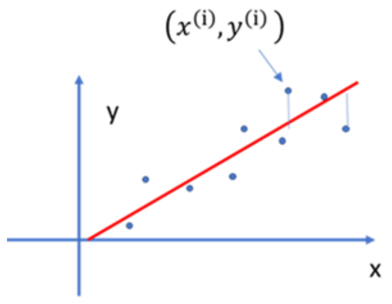


Fig.1 Linear regression

[Example 3.1] An example of linear regress is to predict the housing prices based on the area, given a training set shown in the following table.

Size in feet^2 (x) (predictors or features)	Prices in \$1000 (y) (target or label)
2104	430
1810	312
950	200
1500	280
…	…

To predict target y given a value of variable x, first we need to estimate the coefficients θ_0, θ_1 based on m observations$\left(x^{(1)}, y^{(1)}\right), \left(x^{(2)}, y^{(2)}\right)$ ... $(x^{(m)}, y^{(m)})$. Then y is predicted by the hypothesis

$$\hat{y} = h_\theta(x) = \theta_0 + \theta_1 x \tag{3.2}$$

To find the optimal coefficients, a **cost function** is defined as mean square error

$$J(\theta_0,\theta_1) = \frac{1}{2m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})^2 = \frac{1}{2m}\sum_{i=1}^{m}(\theta_0 + \theta_1 x^{(i)} - y^{(i)})^2 \tag{3.3}$$

The optimal coefficients θ_0, θ_1 should minimize the cost function. An analytical solution can be found by solving the system of linear equations

$$\begin{cases} \frac{\partial}{\partial\theta_0} J(\theta_0,\theta_1) = 0 \\ \frac{\partial}{\partial\theta_1} J(\theta_0,\theta_1) = 0 \end{cases} \tag{3.4}$$

One can show that the solution is

$$\begin{cases} \theta_1 = \frac{\sum_{i=1}^{m}(x^{(i)}-\bar{x})(y^{(i)}-\bar{y})}{\sum_{i=1}^{m}(x^{(i)}-\bar{x})^2} \\ \theta_0 = \bar{y} - \theta_1\bar{x} \end{cases} \tag{3.5}$$

where $\bar{x} = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$, $\bar{x} = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$ are the sample means. (3.5) is also known as formal equation.

3.2.3 Gradient descent

However, in machine learning, the solution to (3.4) is typically obtained by a gradient descent algorithm in an iterative manner, instead of using (3.5). The reason is that (3.5) requires the linearity of the model (3.2) that is not the case in many machine learning, and the computation of (3.5) for multiple features are prohibitively expensive, and that the gradient descent algorithm developed here will be generalized to non-linear model in general, as we will see later. Furthermore, as we will see, computing the formal equation for larger dataset multi-dimensional input are prohibitively expensive. The concept of gradient decent algorithm is illustrated in Fig.2. If function $J(\theta)$ is convex (it is the case for most of machine learning problems), the value of θ for the minimum of $J(\theta)$ can be iteratively computed by

$$\theta \coloneqq \theta - \alpha\frac{dJ(\theta)}{d\theta} \tag{3.6}$$

where α is a carefully selected constant.

Note: A ***convex function*** is a continuous function whose value at the midpoint of every interval in its domain does not exceed the arithmetic mean of its values at the ends of the interval.

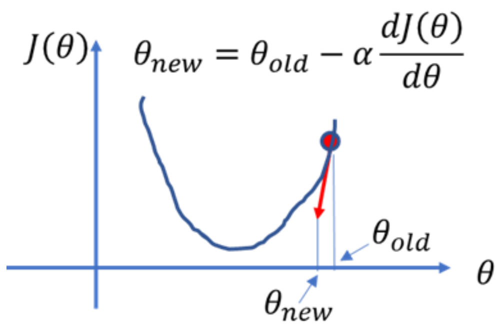

$J(\theta)$
$\theta_{new} = \theta_{old} - \alpha\frac{dJ(\theta)}{d\theta}$
θ_{old}
θ
θ_{new}

Fig.2. Search for a minimum point using gradient descent

If $J(\theta_0,\theta_1)$, e.g. (3.3), is a two-dimensional convex function, the location (θ_0,θ_1) for the minimum $J(\theta_0,\theta_1)$ can be obtained iteratively by

$$\theta_0 := \theta_0 - \alpha \frac{\partial J(\theta_0, \theta_1)}{\partial \theta_0} \quad (3.7\text{ a})$$

$$\theta_1 := \theta_1 - \alpha \frac{\partial J(\theta_0, \theta_1)}{\partial \theta_1} \quad (3.7\text{ b})$$

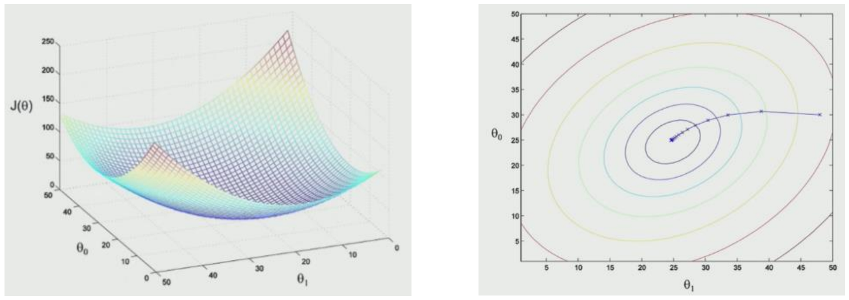


Fig.3 (from CS229 Andrew N.G)

Therefore, a gradient descent algorithm for linear regression can be summarized as the following three steps:

1) *Set initial values for θ_0, θ_1, select a learning rate α (a hyperparameter)*
2) *Repeat:*
   (i) *Update θ_0, θ_1: (simultaneously, not sequentially)*

$$temp0 := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) = \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (\theta_0 + \theta_1 x^{(i)} - y^{(i)}) \quad (3.8\text{ a})$$

$$temp1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) = \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (\theta_0 + \theta_1 x^{(i)} - y^{(i)}) x^{(i)} \quad (3.8\text{ b})$$

$$\theta_0 := temp0$$
$$\theta_1 := temp1$$

   (ii) *Update cost function $J(\theta_0, \theta_1)$*
   (iii) *Terminate the iteration: if the predefined maximal iterations have been completed, then exit to 3), otherwise go back to (i). Or Compare the current cost with the previous cost. If they are close enough, then exit to 3).*
3) *Return θ_0, θ_1*

Fig.3 shows an example of cost function and convergence path of gradient descent.

Remarks: we update θ_0, and θ_1 simultaneously: first, we calculate the temp0 and temp1 with old θ_0, and θ_1, and then we get new θ_0, and $\theta_1$from temp0 and temp1.

3.2.4 Linear regression for multiple features

In many applications, the target *y* is a function of multiple features. In other words, input *x* is a vector.

[Example 3.2] let's consider predicting the housing prices based on multiple features, such as size, number of bedrooms, number of floors, and age of house. The training set is shown in the following table.

features				label
Size in feet^2 **x1**	Number of bedrooms **x2**	Number of floors **x3**	Age of house **x4**	Prices in $1000 **y**
2104	4	2	15	430
1810	3	2	10	312
950	2	1	8	200
1500	3	1	12	280
…				…

An intuitive approach is to generalize the linear regression algorithm from a single feature to multiple features by vectorization. Equation (3.1) can be generalized as

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n + e \tag{3.9}$$

where $x_1, x_2, \ldots, x_n$ are *n* features. This model can be written in vector format

$$y = \boldsymbol{\theta}^T \mathbf{x} + e \tag{3.10}$$

where **θ** and **x** are the column vectors, defined as

$$\boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix} \in R^{(n+1)\times 1}, \qquad \mathbf{x} = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} \in R^{(n+1)\times 1}$$

Thus, given **x**, the corresponding prediction of y is

$$\hat{y} = h_\theta(\mathbf{x}) = \boldsymbol{\theta}^T \mathbf{x} \tag{3.11}$$

In this chapter, the *m* examples in the training set are represented by a matrix $X \in R^{m\times(n+1)}$

$$\mathbf{X} = \begin{bmatrix} --\mathbf{x}^{(1)^T}-- \\ --\mathbf{x}^{(2)^T}-- \\ \vdots \\ --\mathbf{x}^{(m)^T}-- \end{bmatrix} = \begin{bmatrix} 1 & x_1^{(1)} & \ldots & x_n^{(1)} \\ 1 & x_1^{(2)} & \ldots & x_n^{(2)} \\ \vdots & \vdots & & \vdots \\ 1 & x_1^{m} & \cdots & x_n^{(m)} \end{bmatrix} \tag{3.12}$$

where $\mathbf{x}^{(i)^T}$ is the ith example row vector, and $x_j^{(i)}$ is the value of the *jth* feature in the *ith* example. The elements of the first column of X are “1” to accommodate θ_0 in (3.9). In other words, we can have $x_0^{(i)} = 1$.

The labels of the training set are expressed in a column vector $Y \in R^{m\times 1}$

$$Y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

where $y^{(i)}$ is the target (or label) of example $x^{(i)}$.

Thus, the error vector $\boldsymbol{E} \in R^{m\times 1}$ can be defined as

$$\boldsymbol{E} = \begin{bmatrix} e^{(1)} \\ e^{(2)} \\ \vdots \\ e^{(m)} \end{bmatrix} = \mathbf{X} \cdot \theta - Y \tag{3.13}$$

The cost function can be represented by

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(e^{(i)}\right)^2 = \frac{1}{2m}\boldsymbol{E}^{\boldsymbol{T}}\boldsymbol{E} = \frac{1}{2m}(\mathbf{X}\cdot\theta - Y)^T(\mathbf{X}\cdot\theta - Y) \tag{3.14}$$

where $(\ )^{\mathrm{T}}$ is the operator of matrix transpose. By searching the minimal cost over θ, we can find optimal θ for the linear regression model. The derivative of the cost function with respect to θ_j is

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)x_j^{(i)} \qquad j=0,1,\ldots,n, \qquad x_0^{(i)} = 1 \tag{3.15}$$

(3.15) can be equivalently written in a vectorized format as a gradient vector

$$grad = \begin{bmatrix} \frac{\partial J(\theta)}{\partial \theta_0} \\ \frac{\partial J(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{bmatrix} = \frac{1}{m}X^T\cdot(X\cdot\theta - Y) \tag{3.16}$$

The theoretical solution to $\underset{\theta}{minimize}\, J(\theta)$ is the solution to $grad = \frac{1}{m}X^T\cdot(X\cdot\theta - Y) = 0$, which is called **normal equation** given by

$$\theta^* = (X^TX)^{-1}X^TY \tag{3.17}$$

The corresponding gradient decent algorithm can be represented in a vectorized format as follows.

1) *Set initial values for* $\theta \in R^{(n+1)\times 1}$ *, select* α
2) *Repeat:*
   - *(i)* *Compute gradient vector:* $grad \in R^{(n+1)\times 1}$
     $grad = \frac{1}{m}X^T\cdot(X\cdot\theta - Y)$
   - *(ii)* *Update* $\theta := \theta - \alpha\cdot grad$
   - *(iii)* *Update cost function* $J(\theta)$ *(for termination and behavior visualization)*
     $J(\theta) = \frac{1}{2m}(X\cdot\theta - Y)^T(X\cdot\theta - Y)$
   - *(iv)* *Terminate iteration: If the predefined maximal iterations have been completed, then exit to 3), otherwise go back to (i). Or Compare the current cost with the previous cost. If they are close enough, then exit to 3).*
3) *Return* θ

3.2.5 Maximum likelihood estimation—a probabilistic interpretation of linear regression

We have seen how a problem of linear regression can be solved in terms of error square minimization through gradient descent. Here we re-visit the linear regression from a probabilistic perspective, thereby gaining some insights into error functions which are helpful for us to understand more machine learning algorithms we will develop later.

In linear regression, it is assumed that the relationship between the target (or label) variable and feature variable is approximately linear, thus we can estimate the target value for a new feature value using the linear model. However, the exact relationship can be represented by (3.10)

$$y = \boldsymbol{\theta}^T \mathbf{x} + e = h_\theta(x) + e$$

where e is a random noise taking unmodelled effects into account. In the example of house price, the mood of buyers, purchase season, or marriage status of sellers may affect the prices, but not be included in features x. For each data example, there exists an error $e^{(i)}$

$$y^{(i)} = \theta^T x^{(i)} + e^{(i)} = h_\theta\left(x^{(i)}\right) + e^{(i)} \tag{3.18}$$

If we know the probability distribution of e, we can express the uncertainty of y using probability density function. For this purpose, we assume that the error e has a Gaussian distribution with zero mean and a variance of σ^2, i.e. $e \sim N(0, \sigma^2)$. Thus, the value of y has a Gaussian distribution with a mean $h_\theta(x)$ and a variance of σ^2,

$$p(y|x; \theta, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} exp\left(-\frac{\left(y - h_\theta(x)\right)^2}{2\sigma^2}\right) \tag{3.19}$$

We now use the training examples $\left\{x^{(i)}, y^{(i)}, i = 1,2, \dots, m\right\}$ to determine the value of the unknow parameters θ and σ^2 by maximum likelihood (3.20). If the date examples are assumed to be drawn independently from the distribution (3.19), then the likelihood function is given by

$$\mathcal{L}(\theta, \sigma^2) = p(\mathbf{y}|\mathbf{x}; \theta, \sigma^2) = \prod_{i=1}^{m} p(y^{(i)}|x^{(i)}; \theta, \sigma^2) = \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi}\sigma} exp\left(-\frac{\left(y^{(i)} - h_\theta\left(x^{(i)}\right)\right)^2}{2\sigma^2}\right) \tag{3.20}$$

It is convenient to maximize the logarithm of the likelihood function (3.20). By applying the log operation, we obtain the log likelihood function in the form

$$\ell(\theta, \sigma^2) = \ln\left(\mathcal{L}(\theta, \sigma^2)\right) = -\frac{m}{2}\ln(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{m}\left(y^{(i)} - h_\theta\left(x^{(i)}\right)\right)^2 \tag{3.21}$$

Instead of maximizing the likelihood function with respect to θ in (3.20), we can equivalently minimize the sum term of the right side of (3.21), $\frac{1}{2}\sum_{i=1}^{m}\left(y^{(i)} - h_\theta\left(x^{(i)}\right)\right)^2$. We therefore see that maximizing likelihood function is equivalent to minimizing the sum of the squares error function defined by (3.14). As a result, the optimal value of θ, θ_{ML}, can be calculated by

$$\theta_{ML} = argmin \frac{1}{2}\sum_{i=1}^{m}\left(y^{(i)} - h_\theta\left(x^{(i)}\right)\right)^2 \tag{3.22}$$

Therefore, mathematically, **maximum likelihood estimation** (MLE) (3.22) is equivalent to the error square minimization problem (3.14).

Then optimal value of σ^2, ${\sigma^2}_{ML}$, can be obtained by maximizing (3.21) with respect to σ^2, with $\theta = \theta_{ML}$, as

$${\sigma^2}_{ML} = \frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)} - h_{\theta_{ML}}\left(x^{(i)}\right)\right)^2 \tag{3.23}$$

In fact, the resulting ${\sigma^2}_{ML}$ is equal to the average of error squares over all examples, which is equivalent to the minimized cost function (except a scale of ½ in the cost function (3.14)).

Having determined the parameters θ and σ (3.22 and 3.23), we can make prediction for a new value of x, x0, using the probabilistic model defined by (3.19), which gives the probability distribution of y rather than simply a point estimate, shown by the blue curve in Fig.4. Of course, the estimation of y,

$$\hat{y} = h_{\theta_{ML}}(x_0) \tag{3.24}$$

has the maximum value of the probability density function, which justifies $\hat{y}$ as a good prediction in terms of maximum likelihood. In general, $h_\theta(x)$ does not have to be linear with x, as shown by the red curve in the figure.

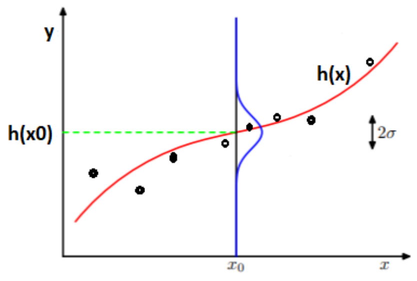


Fig.4 MLE at x₀

3.2.6 Linear basis function models

The linear regression model involves a linear combination of the input features

$$\hat{y} = h_\theta(\mathbf{x}) = \boldsymbol{\theta}^T\mathbf{x} \tag{3.25}$$

The key property of this model is that it is a linear combination of input variables and also a linear combination of parameters. To model a nonlinear relationship between input features and target output, we can generalize the linear model (3.25) to a linear combination of some nonlinear functions of the input features

$$\hat{y} = h_\theta(\mathbf{x}) = \boldsymbol{\theta}^T\boldsymbol{\phi}(x) = \sum_{j=0}^{n} \boldsymbol{\theta}_j\phi_j(\boldsymbol{x}) \tag{3.26}$$

where $\phi_j(\boldsymbol{x})$ are known as basis functions. For example, a set of quadratic basis functions can be

$$\phi(x) = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1x_2 \\ x_1^2 \\ x_2^2 \end{bmatrix} \tag{3.27}$$

Thus, the linear model (3.26) with the basis functions (3.27) can model quadratic surfaces. Another example of (3.26) is polynomial regression with basis function $\phi_j(x) = x^j, j = 0,1 \ldots, m$, where m is the order of polynomial model. In practice, to implement the model of (3.26), we usually apply a certain of pre-processing or feature extraction, to the original data variables x so that the extracted features can be expressed in terms of the basis functions $\{\phi_j(\boldsymbol{x})\}$.

3.3 Practice for Linear Regression

In this section we will use some examples to demonstrate how we can apply the theoretical models to solve the real problems in practice. The purpose is to provide the hands-on experiences involving addressing practical issues and programming.

3.3.1 Practical issues: feature scaling and learning rate

Before we dive into a particular problem, we discuss two practical issues which usually exist in machine learning. The first issue is *feature scaling*. Before we apply data to a machine learning algorithm, we should scale the data (if necessary) so that all features have comparable numerical values. The second issue is learning rate selection. It is essential to select an appropriate value of learning rate α for a successful converge.

Feature scaling

Before using the dataset, we need to make sure that all features are on a similar quantitative scale. In the housing dataset, the house size and number of bedrooms are apparently not on a similar scale. This will result in inefficient computation in gradient decent with a slow converge path, as shown in Fig.5(a). A balanced scale across features will likely have a relatively straight converge path for search the optimal parameters, as shown in Fig.5(b).

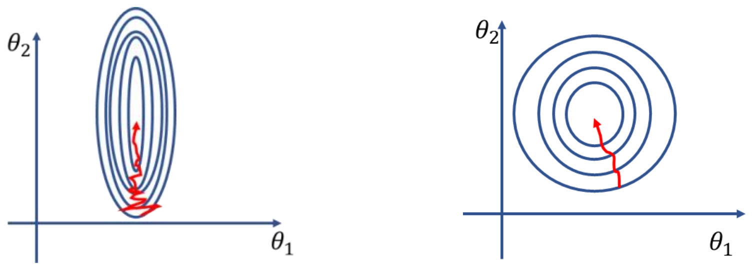


Fig.5 Contour plots and converge path (red arrowed lines): (a) unbalanced feature scale, (b) balanced feature scale.

There are two basic scaling methods:

a) Min-max scaler transforms features by scaling each feature individually to a given range, e.g. between zero and one. For instance, the values of feature x_i can be scaled to range between 0 and 1 by

$$z_i = \frac{x_i - \min_j\{x_i^{(j)}\}}{\max_j\{x_i^{(j)}\} - \min_j\{x_i^{(j)}\}} \tag{3.28}$$

b) Standard scaler standardizes features by removing the mean and scaling to unit variance. Centering and scaling happen independently on each feature by computing the relevant statistics on the samples in the training set. Mean and standard deviation are then stored to be used on later data using the transform method. The standardized feature of a feature x_i is calculated as:

$$z_i = \frac{x_i - u_i}{s_i} \tag{3.29}$$

where u_i is the mean of the feature x_i, and s_i is the standard deviation of the feature x_i.

Choice of learning rate

In the gradient descent, the parameters are updated as

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \tag{3.30}$$

where α is learning rate, which is a hyperparameter. We should choose a value for α so that the cost function $J(\theta)$ decreases after every iteration, as shown in Fig.6. Thus, we can use a plot of cost function versus number of iterations to judge whether the gradient descent is working correctly or not. If the plot is something like Fig.7(b) or (c), a smaller α should be chosen. In summary, if α is too small, convergence is slow. If α is too large, J(θ) may not decrease on every iteration or may not converge. In practice, to choose a good α value, we can try different values, such as …, 0.001, 0.01, 0.1, 1, …, and plot the cost function vs. # of iterations for each value, and then identify a good value for α based on the shape of the plots.

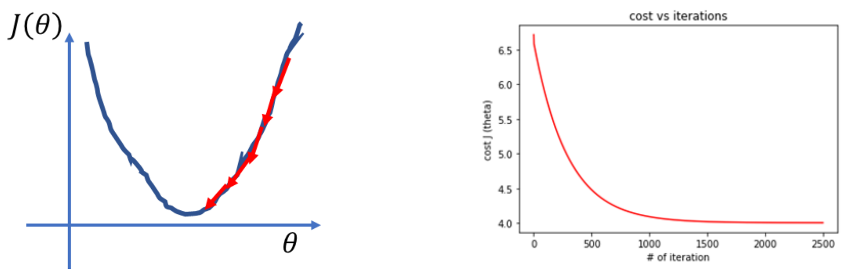


Fig.6 an appropriate value for α, (a) converge path, (b) cost function vs # of iterations

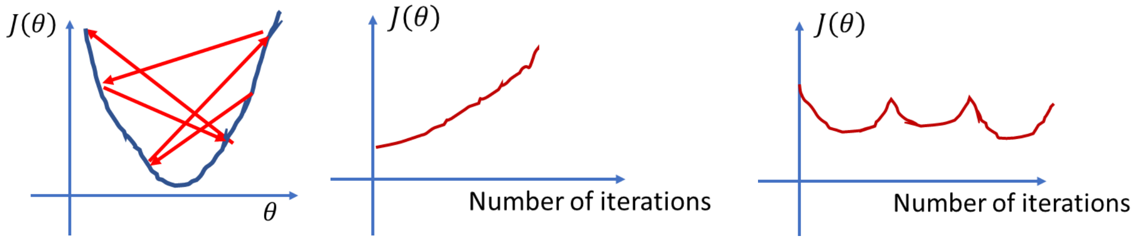


Fig.7 value of α is too large, (a) the path does not converge, (b) and (c) possible plots of cost function.

3.3.2 Implementation of Linear Regression in Python

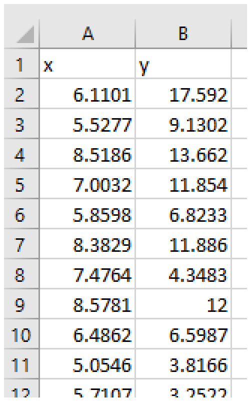
Linear regression on a single feature

The task of linear regression is to learn a linear model that can predict *y* based on a value of *x*. In the dataset, there are 97 examples of input-output pairs (*x*,*y*), stored in the file ex1data1.txt with the first column for x and the second column for y. The part of the data examples is shown in Fig.8(a) by excel. In order to test the performance of the learned model, the dataset is split into two parts: training set with 64 examples and test set with 33 examples. The complete Python program is attached as the file simple_linear_reg.py in the appendix of this chapter. Some explanations are given below.

1) ***Prepare the datasets***: load the data file, split the data into training set and test set, and store the values of feature and label into corresponding array variables.

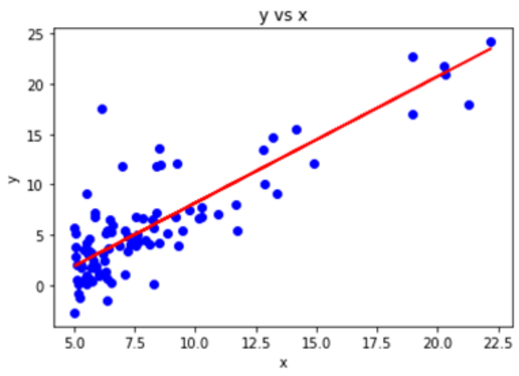
```
dataset = np.loadtxt("ex1data1.txt", delimiter=",")
X = dataset[:, :-1]
y = dataset[:, 1]
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 1/3, random_state = 42)
```

2) ***Implement linear regression from the scratch.*** A reader is encouraged to read the complete code in simple_linear_reg.py for details, and to understand how a gradient descent algorithm for linear regression is implemented from the scratch. Please run the python program by your own. The results of simple_linear_reg.py are shown in Fig.8(b) and Fig.9.



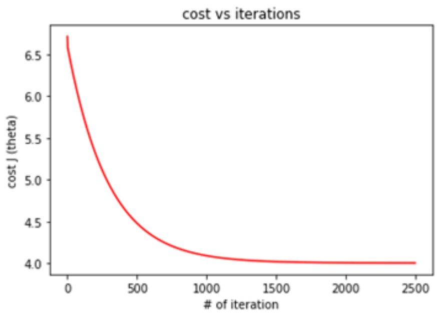
	A	B
1	x	y
2	6.1101	17.592
3	5.5277	9.1302
4	8.5186	13.662
5	7.0032	11.854
6	5.8598	6.8233
7	8.3829	11.886
8	7.4764	4.3483
9	8.5781	12
10	6.4862	6.5987
11	5.0546	3.8166
12	5.7107	3.2522

(a)

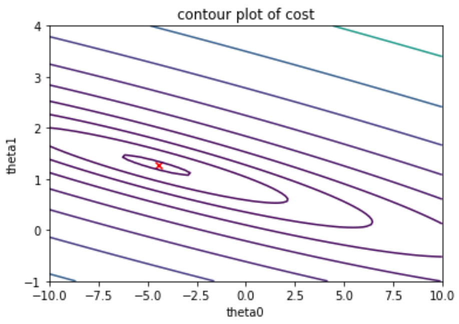


(b)

Fig.8 Linear regression on dataset ex1data1.txt. (a) some examples in dataset, (b) data examples (blue dots) and linear regression prediction line (red line): $y = \theta_0 + \theta_1 x$ with $\theta_0 = -4.46791403, \theta_1 = 1.26456023$



(a)



(b)

Fig.9 visualization of cost function $J(\theta_0, \theta_1)$. (a) cost function $J(\theta_0, \theta_1)$ is a decreasing function of iterations, (b) contour plot (the red x indicates the location of (θ_0, θ_1) for the minimal cost).

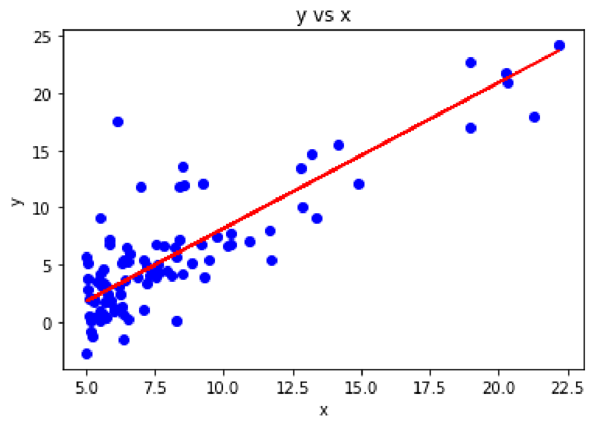
3) ***Implement the linear regression with sklearn library.*** The following piece of codes shows how to train or fit a LinearRegression model from sklearn library, and plot (print) the result.

```
from sklearn.linear_model import LinearRegression
reg = LinearRegression()
reg.fit(X_train, y_train)
y_pred = reg.predict(X_test)

plt.scatter(X, y, color = 'blue')
plt.plot(X_train, reg.predict(X_train), color = 'red')
plt.title('y vs x')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

```
print("theta0 is ", reg.intercept_)
print("theta1 is ", reg.coef_[0])
```

Output:

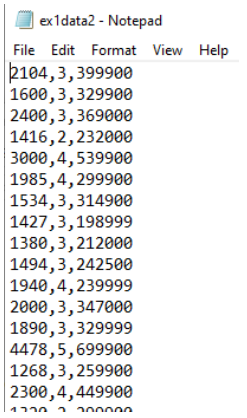


```
theta0 is  -4.546971116577826
theta1 is  1.272579439236726
```

Linear regression on multiple features

1) Data pre-processing: normalization

The previous python code (gradient descent algorithm) from scratch can be used for multiple linear regression without any modification, because the code, using vectorization, can accommodate the situation with any number of features. In the following example, we will implement linear regression with multiple features to predict the prices of houses. The file *ex1data2.txt* contains a training set of housing prices in Portland, Oregon. The first column is the size of the house (in square feet), the second column is the number of bedrooms, and the third column is the price of the house. A part of the dataset is shown at the right side. By looking at the values, note that house sizes are about 1000 times the number of bedrooms. When features differ by orders of magnitude, first performing feature scaling can make gradient descent converge much more quickly. For example, the follow piece of codes transforms all features into the range [0,1]. X is the original input, and X_norm is the transformed data in which all features are normalized to the range [0,1]:



```
2104,3,399900
1600,3,329900
2400,3,369000
1416,2,232000
3000,4,539900
1985,4,299900
1534,3,314900
1427,3,198999
1380,3,212000
1494,3,242500
1940,4,239999
2000,3,347000
1890,3,329999
4478,5,699900
1268,3,259900
2300,4,449900
```

$$x_{norm} = \frac{x - min}{max - min}$$

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
scaler.fit(X)
X_norm=scaler.transform(X)
feature_min=scaler.data_min_
feature_max=scaler.data_max_
feature_range=scaler.data_range_
```

After normalizing the features, it is important to store the parameters used for normalization, such as minimum, maximum, and range. The variables, feature_min, feature_max, feature_range, keep a record of the normalization parameters. Given a new x value (living room area and number of bedrooms), to predict the price of the house, we must first normalize x using the same parameters that we had previously used for the training set normalization.

```
X_new_norm=(X_new-feature_min)/feature_range
```

2) *Data pre-processing: Split into training set and test set.*

we split the dataset into two parts: training set and test set. We will use the trainning set to learn the model. The following code implements the splitting.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_norm, y, test_size = 1/3, random_state = 42)
```

3) *Python program and results.*

The complete code is attached as mult_linear_reg.py in Appendix. The results are visualized by Fig.10.

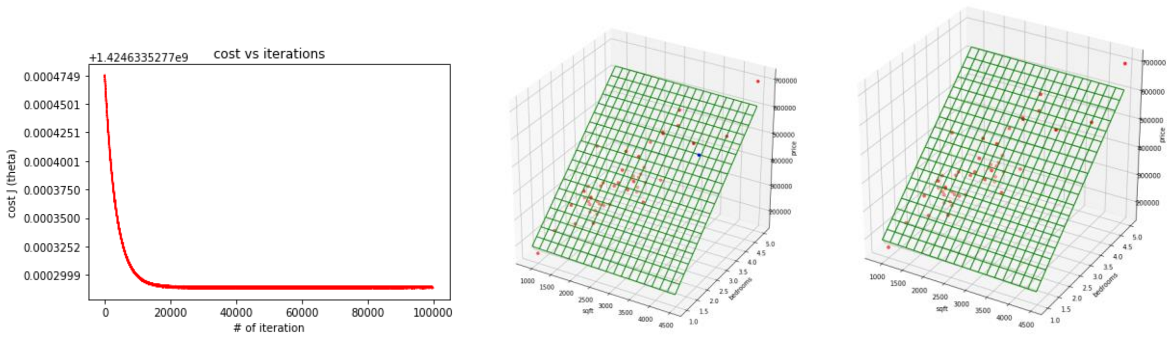


Fig.10 (a) cost versus the number of iterations (only plot after 100000 iteration). (b) result from the scratch: data examples (red dots) and linear regression plane (green frame). (c) result based sklearn linear regression model

In mult_linear_reg.py, a few lines are written to output the predicted prices for the house with area of 4500 and 2 bedrooms, using gradient decent algorithm and sklearn, respectively. The results are very close. The outputs are listed as follows.

```
predict for [4500,2]

python from scratch:
```

```
θ0,θ1,θ2: 190315.5123446676 432062.6922871012 -13377.2805744575
price: [621655.33491019]

sklearn:
θ0,θ1,θ2: 190315.51234468038 432062.692287245 -13377.280574555421
price: [621655.33491033]
```

Now we also use the test set to test the effectiveness of two implementations (from the scratch model and sklearn model) on the test set and the result is shown below.

```
Area,#of bds: 2526.0 3.0, real_price: 469000.00, my predic: 383095.42, sklearn predict: 383095.42
Area,#of bds: 2162.0 4.0, real_price: 287000.00, my predic: 336378.01, sklearn predict: 336378.01
Area,#of bds: 1458.0 3.0, real_price: 464500.00, my predic: 255835.92, sklearn predict: 255835.92
Area,#of bds: 1200.0 3.0, real_price: 299000.00, my predic: 225093.45, sklearn predict: 225093.45
Area,#of bds: 3890.0 3.0, real_price: 573900.00, my predic: 545625.34, sklearn predict: 545625.34
Area,#of bds: 1239.0 3.0, real_price: 229900.00, my predic: 229740.57, sklearn predict: 229740.57
Area,#of bds: 1890.0 3.0, real_price: 329999.00, my predic: 307311.67, sklearn predict: 307311.67
Area,#of bds: 3031.0 4.0, real_price: 599000.00, my predic: 439925.30, sklearn predict: 439925.30
Area,#of bds: 3000.0 4.0, real_price: 539900.00, my predic: 436231.44, sklearn predict: 436231.44
Area,#of bds: 1100.0 3.0, real_price: 249900.00, my predic: 213177.77, sklearn predict: 213177.77
Area,#of bds: 1380.0 3.0, real_price: 212000.00, my predic: 246541.68, sklearn predict: 246541.68
Area,#of bds: 1416.0 2.0, real_price: 232000.00, my predic: 254175.65, sklearn predict: 254175.65
Area,#of bds: 1534.0 3.0, real_price: 314900.00, my predic: 264891.84, sklearn predict: 264891.84
Area,#of bds: 1664.0 2.0, real_price: 368500.00, my predic: 283726.54, sklearn predict: 283726.54
Area,#of bds: 3137.0 3.0, real_price: 579900.00, my predic: 455900.25, sklearn predict: 455900.25
Area,#of bds: 4478.0 5.0, real_price: 699900.00, my predic: 609000.92, sklearn predict: 609000.92
```

Summary

In this chapter, we studied the gradient descent algorithm for a linear regression task. Specifically, a cost function based on mean square error is defined for a linear model with a given training set. The derivatives of the cost function with respect to the parameters of the linear model are calculated and then are used to update the linear parameters by a gradient descent method. The new derivatives of the cost function based on the updated model are used to further update the linear parameters. The iterative update process continues until a stop criterion is satisfied.

Python is used to implement linear regression, and the results are visualized by Python matplotlib package. Some important and frequently used Python functions are summarized through the following examples:

1) Load data file

```python
dataset = np.loadtxt("ex1data1.txt", delimiter=",")
X = dataset[:, :-1]
y = dataset[:, 1]
```

2) Splitting dataset into training set and test set

```python
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 1/3, rando
m_state = 42)
```

3) Normalization

```python
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
```

```python
scaler.fit(X)
X_norm=scaler.transform(X)
feature_min=scaler.data_min_
feature_max=scaler.data_max_
feature_range=scaler.data_range
```

4) Visualization

```python
plt.scatter(X, y, color = 'blue')
plt.plot(X_train, predict(X_train, theta), color = 'red')
```

5) sklearn for linear regression

```python
from sklearn.linear_model import LinearRegression
reg = LinearRegression()
reg.fit(X_train, y_train)
y_pred = reg.predict(X_test)
```

The reader is encouraged to practice the Python functions by running the attached python files. After finishing the chapter, you should be able to implement a linear regression in Python either from the scratch or using sklearn library.

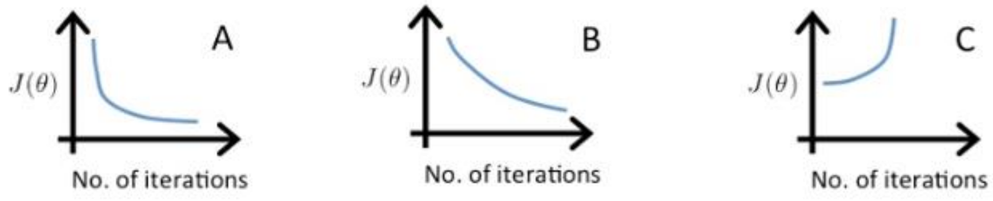
References

[1] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani, Chapter 3, Linear Regression, in *An introduction to Statistical Learning with Applications in R.* Springer.

[2] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

Exercises

1. You run gradient descent for 30 iterations with α=0.1 and compute $J(\theta)$ for each iteration. You find that the value of $J(\theta)$ decreases slowly and is still decreasing after 15 iterations. Based on this, which of the following conclusions seems most plausible?
   a) α=0.1 is an effective choice of learning rate.
   b) It would be more promising to try a larger value of α (say α=0.5)
   c) Rather than use the current value of α, it would be more promising to try a smaller value of α (say α=0.05)
2. Suppose you have m=100 training examples with n=5 features and would learn a linear regression model to directly fit the dataset. According to the notations in this chapter, what are the dimensions of θ, X, and Y?
3. A person ran a gradient descent three times with three different values of α, 0.01, 0.1 and 1, respectively, and got three plots shown in following figure. Which of α values is corresponding to each plot?



4. Suppose you train a linear regression to fit the dataset ex1data1.txt (can be downloaded from website) like Fig.8. In this exercise, **you are asked to plot the first five learned lines during the first five iterations, and the convergence path of the first 100 steps using contours plot.** The figure (left below) shows possible result for five learned lines and the figure (right below) shows possible convergence path. (hint: utilize or modify the file simple_linear_reg.py) Since the choice of initial value of θ leads to different results, you are asked to repeat the exercise using two different initial values of θ.

   Hint: to save the parameter theta for each iteration, you can modify gradient_descent as follows:

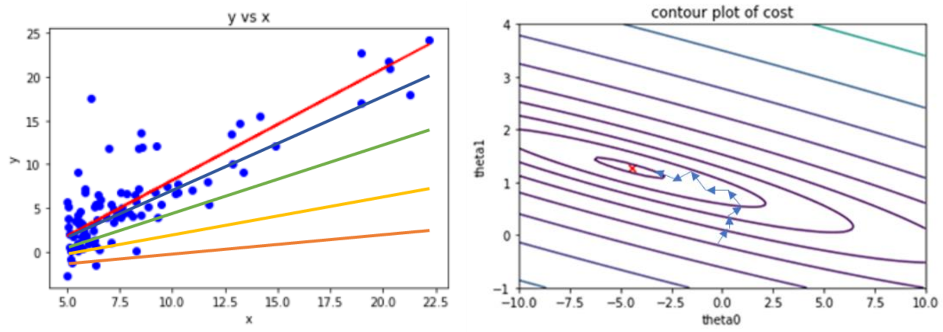
```
def gradient_descent(alpha, x, y, numIterations):
    cost=[]
    m = x.shape[0] # number of samples
    x0=np.ones(m).reshape((m,1))
    x=np.concatenate((x0,x), axis=1)
    theta= np.zeros([numIterations+1, x.shape[1]])
    x_transpose = x.transpose()

    for iter in range(0, numIterations):

        hypothesis = np.dot(x, theta[iter])
        loss = hypothesis - y
        J = np.sum(loss ** 2) / (2 * m)  # cost
        #print ("iter %s | J: %.3f" % (iter, J))
        gradient = np.dot(x_transpose, loss) / m
        theta[iter+1] = theta[iter] - alpha * gradient  # update
        cost.append(J)
    return theta,cost;
```

1) Use $\theta = \begin{bmatrix}1\\1\end{bmatrix}$ as the initial value for the iteration in gradient descent.

2) Use $\theta = \begin{bmatrix}0\\0\end{bmatrix}$ as the initial value for the iteration in gradient descent.



5. In this exercise, you are asked to generate an artificial dataset based on a linear model, and then learn the model using gradient descent. Specifically,
    1) Generate 100 data examples $(x^{(i)}, y^{(i)}), i = 1,2,\ldots,100$, which are **randomly** drawn from the model
    $$y = a + bx + N(0,1)$$
    where a and b are constants (say a=1, b=2), $0 \le x \le 2$, N(0,1) is a random number of Gaussian distribution with mean=0 and variance=1. Plot a 2D scatterplot for your generated data examples. Please note that sampling on x should be randomly rather than uniformly.
    2) Learn a linear regression model $y = \theta_0 + \theta_1 x$ to fit the generated examples. Compare the results (θ_0, θ_1) with the pre-defined model parameters (a, b). Plot the fitted line along with the data examples, and the cost function versus iteration index.

6. In this exercise, you are asked to generate an artificial dataset based on a nonlinear model, and then learn the model using gradient descent. Specifically,
    1) Generate 100 data examples $(x^{(i)}, y^{(i)}), i = 1,2,\ldots,100$, which are **randomly** drawn from the model
    $$y = a + bx + cx^2 + dx^3 + ex^4 + 0.5 \cdot N(0,1)$$
    where a,b,c,d, and e are constants (say a=1, b=2, c=1.5, d=3, e=1), $0 \le x \le 1$, N(0,1) is a random number of Gaussian distribution with mean=0 and variance=1. Plot a 2D scatterplot for your generated data examples. Please note that sampling on x should be randomly rather than uniformly.
    2) Learn a linear regression model $y = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$ to fit the generated examples. Compare the results $(\theta_0, \theta_1, \theta_2, \theta_3, \theta_4)$ with the pre-defined model parameters (a, b, c, d, e). Plot the fitted line along with the data examples, and the cost function versus iteration index.

Appendix

1) Simple_linear_reg.py

```python
import numpy as np
import matplotlib.pyplot as plt


def predict(x, theta):
    m = x.shape[0] # number of samples
    x0=np.ones(m).reshape((m,1))
    x=np.concatenate((x0,x), axis=1) # add the first column with "1"
    return np.dot(x,theta)



def gradient_descent(alpha, x, y, numIterations):
    cost=[]
    m = x.shape[0] # number of samples
    x0=np.ones(m).reshape((m,1))
    x=np.concatenate((x0,x), axis=1)
    theta = np.ones(x.shape[1])
    x_transpose = x.transpose()
    for iter in range(0, numIterations):
        hypothesis = np.dot(x, theta)
        loss = hypothesis - y
        J = np.sum(loss ** 2) / (2 * m)  # cost
        #print ("iter %s | J: %.3f" % (iter, J))
        gradient = np.dot(x_transpose, loss) / m
```

```
            theta = theta - alpha * gradient  # update
            cost.append(J)
        return theta,cost;


    dataset = np.loadtxt("ex1data1.txt", delimiter=",")
    X = dataset[:, :-1]
    y = dataset[:, 1]
    from sklearn.model_selection import train_test_split
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 1/3, random_s
    tate = 42)

    theta ,cost= gradient_descent(alpha=0.01, x=X_train, y=y_train, numIterations=2500)

    plt.scatter(X, y, color = 'blue')
    plt.plot(X_train, predict(X_train, theta), color = 'red')
    plt.title('y vs x')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.show()

    print("theta is ", theta)
    plt.plot(cost[1:], color='red')
    plt.title('cost vs iterations')
    plt.xlabel('# of iteration')
    plt.ylabel('cost J (theta)')
    plt.show()


    def computeCost(x,y,theta):
        m=x.shape[0]
        J=0.
        x0=np.ones(m).reshape(m,1)
        x=np.concatenate((x0,x), axis=1)
        loss=np.dot(x,theta)-y
        J=np.sum(loss**2)/(2*m)
        return J

    xlist = np.linspace(-10.0, 10.0, 100)
    ylist = np.linspace(-1.0, 4.0, 100)
    theta0_vals, theta1_vals = np.meshgrid(xlist, ylist)
    J_vals = np.zeros((theta0_vals.shape[0], theta1_vals.shape[0]))
    for i in range(0,theta0_vals.shape[0]):
        for j in range(0, theta1_vals.shape[0]):
            t=np.array([xlist[i], ylist[j]])
            J_vals[i,j]=computeCost(X_train,y_train,t)

    J_vals=J_vals.transpose()
    plt.figure()
    levels = np.logspace(-2, 3,20)
    contour = plt.contour(theta0_vals, theta1_vals, J_vals, levels)

    plt.scatter(theta[0], theta[1], marker='x', color='red')
    plt.title('contour plot of cost')
    plt.xlabel('theta0')
    plt.ylabel('theta1')
    plt.show()
    from mpl_toolkits.mplot3d import Axes3D

    fig = plt.figure(figsize=(14,6))

```

```python
    ax = fig.add_subplot(1, 2, 1, projection='3d')

    p = ax.plot_surface(theta0_vals, theta1_vals, J_vals, rstride=4, cstride=4, linewid
    th=0)
    ax.set_xlabel('theta0')
    ax.set_ylabel('theta1')
    ax.set_zlabel('J')
    plt.show()
    RSS=2*len(y_train)*cost[-1]
    from math import sqrt
    RSE=sqrt(RSS/(len(y_train)-2))

    from statistics import variance
    R2=1-RSS/(variance(y_train)*(len(y_train)-1))
    print("RSS= %5.2f, RSE= %5.2f, R2= %5.3f" % (RSS, RSE, R2))

    # using sklearn model
    from sklearn.linear_model import LinearRegression
    reg = LinearRegression()
    reg.fit(X_train, y_train)
    y_pred = reg.predict(X_test)

    plt.scatter(X, y, color = 'blue')
    plt.plot(X_train, reg.predict(X_train), color = 'red')
    plt.title('y vs x')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.show()
    print("R2= %5.3f"% reg.score(X_train,y_train))
    print("theta0 is ", reg.intercept_)
    print("theta1 is ", reg.coef_[0])
```

2) mult_linear_reg.py

```python
import numpy as np
import matplotlib.pyplot as plt


def predict(x, theta):
    m = x.shape[0] # number of samples
    x0=np.ones(m).reshape((m,1))
    x=np.concatenate((x0,x), axis=1) # add the first column with "1"
    return np.dot(x,theta)

def computeCost(x,y,theta):
    m=x.shape[0]
    J=0.
    x0=np.ones(m).reshape(m,1)
    x=np.concatenate((x0,x), axis=1)
    loss=np.dot(x,theta)-y
    J=np.sum(loss**2)/(2*m)
    return J

def gradient_descent(alpha, x, y, numIterations):
    cost=[]
    m = x.shape[0] # number of samples
    x0=np.ones(m).reshape((m,1))
    x=np.concatenate((x0,x), axis=1)
    theta = np.ones(x.shape[1])
    x_transpose = x.transpose()
    for iter in range(0, numIterations):
        hypothesis = np.dot(x, theta)
        loss = hypothesis - y
        J = np.sum(loss ** 2) / (2 * m)  # cost
        #print ("iter %s | J: %.3f" % (iter, J))
        gradient = np.dot(x_transpose, loss) / m
        theta = theta - alpha * gradient  # update
        cost.append(J)
    return theta,cost;


dataset = np.loadtxt("ex1data2.txt", delimiter=",")
X = dataset[:, :-1]
y = dataset[:, 2]
#from sklearn import preprocessing
#X = preprocessing.normalize(X, axis=0, norm='l2')
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
scaler.fit(X)
X_norm=scaler.transform(X)
feature_min=scaler.data_min_
feature_max=scaler.data_max_
feature_range=scaler.data_range_
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_norm, y, test_size = 1/3, random_state = 42)

theta ,cost= gradient_descent(alpha=0.01, x=X_train, y=y_train, numIterations=200000)


plt.plot(cost[100000:], color='red')
plt.title('cost vs iterations')
plt.xlabel('# of iteration')
plt.ylabel('cost J (theta)')
plt.show()

print("predict for [4500,2]")
print("python from scratch:")
print("\u03B8"+str(0)+","+"\u03B8"+str(1)+","+"\u03B8"+str(2)+":", theta[0], theta[1], theta[2])

#predict for a specific case

X_new=np.array([[4500, 2]])
X_new_norm=(X_new-feature_min)/feature_range
Y_new_pred=predict(X_new_norm,theta)

print("price:", Y_new_pred)

from sklearn.linear_model import LinearRegression
lin_reg=LinearRegression()
lin_reg.fit(X_train, y_train)

print("sklearn:")
print("\u03B8"+str(0)+","+"\u03B8"+str(1)+","+"\u03B8"+str(2)+":", lin_reg.intercept_, lin_reg.coef_[0], lin_reg.coef_[1])
```

```python
    print("price:", lin_reg.predict(X_new_norm))

    #visualization of the results
    from mpl_toolkits import mplot3d
    fig = plt.figure(figsize=(10,10))

    ax = plt.axes(projection='3d')
    xdata = X[:,0]
    ydata = X[:,1]
    ax.scatter3D(xdata, ydata, y, c='red', cmap='Greens')
    ax.scatter3D(X_new[0][0], X_new[0][1], Y_new_pred, c='blue')
    ax.set_xlabel('sqft')
    ax.set_ylabel('bedrooms')
    ax.set_zlabel('price');

    xlist = np.linspace(0., 1.0, 20)
    ylist = np.linspace(0., 1.0, 20)
    X1,X2= np.meshgrid(xlist, ylist)
    pred=np.zeros((X1.shape[0],X2.shape[0]))
    for i in range(0,X1.shape[0]):
        for j in range(0, X2.shape[0]):
            t=np.array([[xlist[i], ylist[j]]])
            pred[i,j]=predict(t,theta)
    xlist=xlist*feature_range[0]+feature_min[0]
    ylist=ylist*feature_range[1]+feature_min[1]
    X1,X2 = np.meshgrid(xlist, ylist)


    ax.plot_wireframe(X1, X2, pred, color='green')

    plt.show()


    fig = plt.figure(figsize=(10,10))
    ax = plt.axes(projection='3d')
    xdata = X[:,0]
    ydata = X[:,1]
    ax.scatter3D(xdata, ydata, y, c='red', cmap='Greens');
    ax.set_xlabel('sqft')
    ax.set_ylabel('bedrooms')
    ax.set_zlabel('price');

    xlist = np.linspace(0., 1.0, 20)
    ylist = np.linspace(0., 1.0, 20)
    X1,X2= np.meshgrid(xlist, ylist)
    pred=np.zeros((X1.shape[0],X2.shape[0]))
    for i in range(0,X1.shape[0]):
        for j in range(0, X2.shape[0]):
            t=np.array([[xlist[i], ylist[j]]])
            pred[i,j]=lin_reg.predict(t)
    xlist=xlist*feature_range[0]+feature_min[0]
    ylist=ylist*feature_range[1]+feature_min[1]
    X1,X2 = np.meshgrid(xlist, ylist)


    ax.plot_wireframe(X1, X2, pred, color='green')

    plt.show()


    # predict all test samples
```

```python
Y_test_pred=predict(X_test,theta)
Y_test_pred_model=lin_reg.predict(X_test)

for i in range (0,len(X_test)):
    T=X_test[i]*feature_range+feature_min
    print("Area,#of bds: %5.1f %2.1f, real_price: %8.2f, my predic: %8.2f, sklearn predict: %8.2f"
          % (T[0], T[1] , y_test[i], Y_test_pred[i], Y_test_pred_model[i]))
```