Chapter 3

Linear Regression

Regression refers to modeling the relationship between one or more independent variables and a dependent variable. These variables take continuous (or numerical) values. Linear regression is the simplest and most popular among the standard tools for regression. In linear regression, we assume that the relationship between the dependent variable and the independent variables is *approximately* linear, i.e., that the dependent variable can be predicted as a linear combination of input variables. Thus, in a linear regression model, the prediction is not only a linear function of the parameters, but also a linear function of input variables.

Through regression in this chapter, we introduce the basic concepts of machine learning, such as models, datasets, loss function, training, inference, gradient descent algorithm, and so on. Furthermore, linear regression can be generalized to many other more powerful machine learning paradigms, such as logistic regression, neural networks, and deep neural networks. Thus, this chapter provides a foundation for the rest of the book.

In this chapter you will learn about

- Linear regression model and its analytic solution.
- Gradient descent algorithm for linear regression.
- Linear models using basis functions: polynomial curve fitting.
- Analysis of bias and variance of a model.
- Basic Python programming for data-preprocessing and gradient descent algorithm for linear regression models.

3.1 Linear Regression with Single Feature

3.1.1 Linear regression model

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 the setting of a supervised learning task, we are given a training dataset that consists of *m* examples or samples defined as input-output pairs $\left(\mathbf{x}^{(1)}, y^{(1)}\right), \left(\mathbf{x}^{(2)}, y^{(2)}\right) \dots (\mathbf{x}^{(m)}, y^{(m)})$, where the superscript indicates the index of examples. Each input $\mathbf{x}^{(i)}$ is a vector including n ***features*** (or ***attributes***, or ***predictors***). The output, often referred as the ***target*** or ***label***, is assumed to be univariate, i.e., $y^{(i)} \in \mathbb{R}$, in this chapter. The later chapters show that $y^{(i)}$ could be a vector in general. The goal is to ***learn*** or ***train*** a model that describes the relationship between the input and the output, based on the training dataset. After we have completed training, given a new input value *x*, we can use the trained model to ***predict*** its output, denoted as $\hat{y}(x)$. 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 reduced to a linear regression problem.

We will begin to address the linear regression with a single feature. 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.3.1. Mathematically, the relationship can be represented as

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

where θ_0^t, θ_1^t are the intercept and the slope for the assumed linear relationship, respectively. ϵ is the error term for a consideration of the following facts: the true relationship is *not exactly* linear, there are other variables that cause variation in y, and there exist measurement errors. In a linear regression task, we are given a set of data examples drawn from (3.1), and learn a linear model (or hypothesis) defined by

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

In Fig.3.1, the blue dots represent the data example points, and the red line is the linear regression line defined by (3.2).

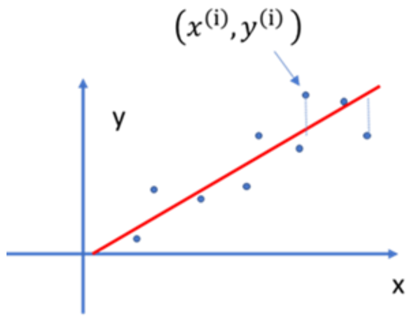


Fig.3.1 Linear regression

3.1.2 Cost function

In general, a linear model (e.g. the red line in Fig.3.1) does not fit the training dataset exactly (e.g. the blue dots in Fig.3.1). To find the optimal values for the parameters in (3.2), we need to define a *cost (or loss) function* that quantifies the mismatch between the true values and predicted values of the target. There are various loss functions available. However, a common choice of loss function in regression problems is the mean squared error, given by

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

3.1.3 Analytic solution

The optimal coefficients θ_0, θ_1 minimize the cost function. Since the cost function of linear regression is a quadratic function of coefficients, there is just one critical point on the cost surface, and it corresponds to the minimum of the cost. 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

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

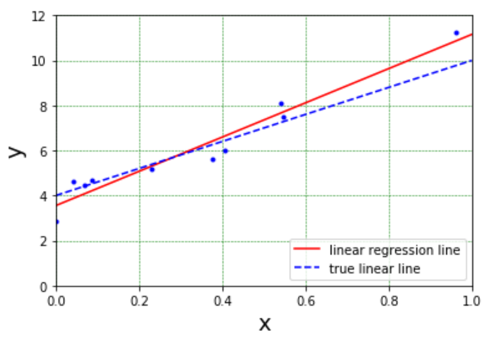
or equivalently,

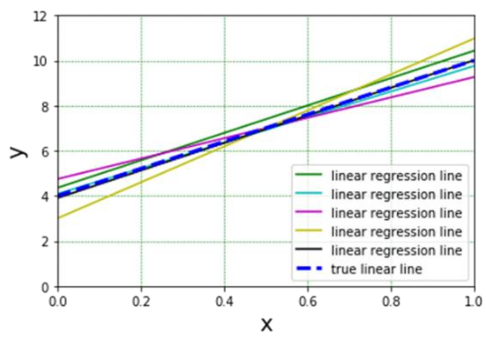
$$\begin{cases} \theta_1 = \frac{\sum_{i=1}^{m}\left(x^{(i)}-\bar{x}\right)\left(y^{(i)}-\bar{y}\right)}{\sum_{i=1}^{m}\left(x^{(i)}-\bar{x}\right)^2} \\ \theta_0 = \bar{y} - \theta_1\bar{x} \end{cases} \tag{3.5b}$$

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

Let's consider a dataset consisting of 10 examples drawn from the relationship $y = 4 + 6x + \epsilon$, where ϵ is a standard Gaussian random variable (mean=0, variance=1). The following Python program implements this linear regression on the dataset by the normal equation (3.5). The result is plotted in Fig.3.2(a), where the blue dots represent the data examples, the dashed blue line is the target linear line $y = 4 + 6x$, and the red line is the linear regression line $y = \theta_0 + \theta_1 x$ for prediction. In Fig.3.2(b), different linear regression lines are plotted, each of which is computed on a separate dataset randomly drawn from the same relationship $y = 4 + 6x + \epsilon$. The average of this linear regression lines is quite close to the true linear line.

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

# Fixing random state for reproducibility
np.random.seed(1968)
x=np.random.rand(10)  # uniform [0,1)
y=4+6*x+np.random.randn(10) # randn: Normal with mean=0, var=1

# normal equation
x_bar=statistics.mean(x)
y_bar=statistics.mean(y)
theta1=sum(np.multiply((x-x_bar),(y-y_bar)))/sum(np.multiply((x-x_bar),(x-x_bar
)))
theta0=y_bar-theta1*x_bar

# visualize results
x_new=np.array([0,1])
y_predict=theta1*x_new+theta0
y_true=6*x_new+4
plt.plot(x_new,y_predict,"r-", label='linear regression line')
plt.plot(x_new,y_true,"b--", label='true linear line')
plt.plot(x,y,"b.")
plt.xlabel('x', fontsize=18)
plt.ylabel('y', fontsize=18)
plt.grid(color = 'green', linestyle = '--', linewidth = 0.5)
plt.legend(loc='lower right')
plt.xlim([0,1])
plt.ylim([0,12])
plt.show()
```





(a) linear regression on one dataset (b) linear regressions on different sets of observations

Fig.3.2 Linear regression model for prediction

Although the analytic solution allows for nice mathematical analysis, it is not available for general neural network and deep learning models because the relationship between the output and inputs are very complicated (at least not linear). Fortunately, we will see that we can train or learn the model in an iterative manner efficiently in practice.

3.1.4 Gradient descent algorithm

In practice, the solution to (3.4) is usually obtained by a gradient descent algorithm in an iterative manner, instead of the analytic solution (3.5). The analytic solution requires linearity of the model which does not hold in many applications, and the gradient descent algorithm developed here will be generalized to non-linear models, as we will see later. Furthermore, computing the analytic solution for large dataset with multi-dimensional input is prohibitively expensive. The concept of gradient decent algorithm is illustrated in Fig.3.3. If function $J(\theta)$ is convex (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), the value of θ for the minimum of $J(\theta)$ can be iteratively computed by

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

where α is a carefully selected constant, called ***learning rate***, which controls the amount of updating.

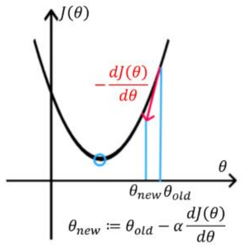

$J(\theta)$
$-\frac{dJ(\theta)}{d\theta}$
θ
$\theta_{new}\theta_{old}$
$\theta_{new} := \theta_{old} - \alpha\frac{dJ(\theta)}{d\theta}$

Fig.3.3 Search for a minimum point using gradient descent: the blue circle represents the minimum point and the red arrow represents the direction of the negative gradient.

Since the cost function $J(\theta_0, \theta_1)$, is a two-dimensional convex function, the optimal (θ_0, θ_1) for the minimum $J(\theta_0, \theta_1)$ can be obtained iteratively by

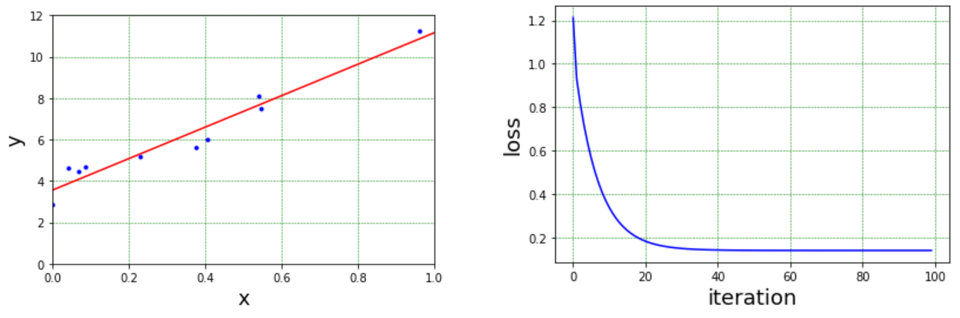
$$\theta_0 := \theta_0 - \alpha \frac{\partial J(\theta_0,\theta_1)}{\partial \theta_0} \quad \text{and} \quad \theta_1 := \theta_1 - \alpha \frac{\partial J(\theta_0,\theta_1)}{\partial \theta_1} \tag{3.7}$$

Therefore, the gradient descent algorithm for linear regression can be summarized as the follows.

```
1) Set initial values for $\theta_0, \theta_1$, select a learning rate $\alpha$ (a hyperparameter)
2) Repeat:
    (i) Update $\theta_0, \theta_1$: (simultaneously, not sequentially)
            $temp0 := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) = \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (\theta_0 + \theta_1 x^{(i)} - y^{(i)})$
            $temp1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) = \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (\theta_0 + \theta_1 x^{(i)} - y^{(i)}) x^{(i)}$
            $\theta_0 \coloneqq temp0$
            $\theta_1 \coloneqq temp1$
    (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 $\theta_0, \theta_1$
```

The following python program shows an example of linear regression using gradient descent algorithm. The results are plotted in Fig.3.3. It is usually helpful to plot the loss versus iteration step for monitoring the training process, as in Fig.3.3(b).

```python
# gradient descent algorithm for a single feature input

# Fixing random state for reproducibility
np.random.seed(1968)
x=np.random.rand(10)
y=4+6*x+np.random.randn(10)

# gradient descent algorithm
theta0=1
theta1=1
alpha=1
cost=[]
m=x.shape[0]

for iter in range(0, 100):
    error=theta0+theta1*x-y
    temp0=theta0-alpha*statistics.mean(error)
    temp1=theta1-alpha*statistics.mean(error*x)
    theta0=temp0
    theta1=temp1
    J=sum((theta0+theta1*x-y)**2)/(2*m)
    cost.append(J)

# plot the linear model and traning examples
x_new=np.array([0,1])
y_predict=theta1*x_new+theta0
plt.plot(x_new,y_predict,"r-")
plt.plot(x,y,"b.")
plt.xlabel('x', fontsize=18)
plt.ylabel('y', fontsize=18)
plt.grid(color = 'green', linestyle = '--', linewidth = 0.5)
plt.xlim([0,1])
plt.ylim([0,12])
plt.show()
# plot the loss function vs iteration steps
plt.plot(cost, 'b-')
plt.xlabel('iteration', fontsize="18")
plt.ylabel('loss', fontsize="18")
plt.grid(color = 'green', linestyle = '--', linewidth = 0.5)
plt.show()
```



(a) linear model (b) loss function

Fig.3.3 Linear regression using gradient descent algorithm

3.2 Linear Regression with Multiple Features

In this section, we will address the linear regression models with multiple features. In many applications, the target y is a function of multiple features. In other words, input **x** is a vector. 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

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n + e \qquad (3.8)$$

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

$$y = \boldsymbol{\theta}^T \mathbf{x} + e \qquad (3.9)$$

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

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

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

$$\hat{y} = h_\theta(\mathbf{x}) = \boldsymbol{\theta}^T \mathbf{x} \qquad (3.10)$$

To represent the cost function in a form of vectors and matrices, we arrange the dataset in a matrix: each row associated with one example. Specifically, the m examples in the training set are represented by a matrix $X \in \mathbb{R}^{m\times(n+1)}$

$$\mathbf{X} = \begin{bmatrix} --\mathbf{x}^{(1)^\top}-- \\ --\mathbf{x}^{(2)^\top}-- \\ \vdots \\ --\mathbf{x}^{(m)^\top}-- \end{bmatrix} = \begin{bmatrix} 1 & x_1^{(1)} & \cdots & x_n^{(1)} \\ 1 & x_1^{(2)} & \cdots & x_n^{(2)} \\ \vdots & \vdots & & \vdots \\ 1 & x_1^{m} & \cdots & x_n^{(m)} \end{bmatrix} \qquad (3.11)$$

where $\mathbf{x}^{(i)^{\mathsf{T}}}$ 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 (3.8). In other words, we can have an additional dummy feature $x_0^{(i)} = 1$.

The targets or labels of the training set are expressed as a column vector $\mathbf{y} \in \mathbb{R}^{m\times 1}$

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

where $y^{(i)}$ is the target (or label) of example $\mathbf{x}^{(i)}$. Thus, the error vector $\boldsymbol{E} \in \mathbb{R}^{m\times 1}$ can be defined as

$$\boldsymbol{E} = \begin{bmatrix} e^{(1)} \\ e^{(2)} \\ \vdots \\ e^{(m)} \end{bmatrix} = \mathbf{X}\cdot\ \boldsymbol{\theta} - \mathbf{y} \tag{3.12}$$

The cost function can be represented by

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(e^{(i)}\right)^2 = \frac{1}{2m}\boldsymbol{E}^{\mathsf{T}}\boldsymbol{E} = \frac{1}{2m}(\mathbf{X}\cdot\boldsymbol{\theta} - \mathbf{y})^{\mathsf{T}}(\mathbf{X}\cdot\boldsymbol{\theta} - \mathbf{y}) \tag{3.13}$$

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

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_j} = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)x_j^{(i)} = \frac{1}{m}\left(X_{column\_j}\right)^{\mathsf{T}}(\mathbf{X}\cdot\boldsymbol{\theta} - \mathbf{y})$$
$$j{=}0,1,\ldots,n, \qquad x_0^{(i)} = 1 \tag{3.14}$$

which can be equivalently written in a compact vectorized format as a gradient vector

$$grad = \begin{bmatrix} \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_0} \\ \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_n} \end{bmatrix} = \frac{1}{m}X^{\mathsf{T}}\cdot(X\cdot\boldsymbol{\theta} - \mathbf{y}) \tag{3.15}$$

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

$$\boldsymbol{\theta}^{*} = (X^{\mathsf{T}}X)^{-1}X^{\mathsf{T}}\mathbf{y} \tag{3.16}$$

This normal equation is a theoretical solution to the linear regression problem. However, it can be computationally costly for large data sets. In practice, it may be convenient to use the gradient decent algorithm to search for the optimal solution.

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

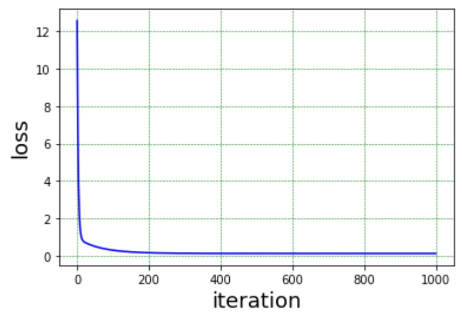
```
1) Set initial values for $\boldsymbol{\theta} \in \mathbb{R}^{(n+1)\times 1}$ , select learning rate $\alpha$
2) Repeat:
    (i)    Compute gradient vector:  $grad \in \mathbb{R}^{(n+1)\times 1}$
           $grad = \frac{1}{m} X^T \cdot (X \cdot \boldsymbol{\theta} - \mathbf{y})$
    (ii)   Update  $\boldsymbol{\theta} \coloneqq \boldsymbol{\theta} - \alpha \cdot grad$
    (iii)  Update cost function $J(\boldsymbol{\theta})$ (not required for gradient descent, but for behavior
           monitoring and visualization)
           $J(\theta) = \frac{1}{2m} (X \cdot \boldsymbol{\theta} - \mathbf{y})^\intercal (X \cdot \boldsymbol{\theta} - \mathbf{y})$
    (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 $\boldsymbol{\theta}$
```

To make the gradient descent algorithm work efficiently, we may need to pay attention to the following practical considerations: 1) balancing the scales of all features; 2) selecting an appropriate learning rate; and 3) dividing the large data set into batches and updating the parameters based on one batch at a time. We will address these issues in more details later.

The following Python program is to fit a linear regression model on a dataset consisting of 10 data examples randomly generated from $y = 4 + 6x_1 - x_2 + e$. With the settings in the program, the parameters of the resulting regression model are `[ 4.42621106   5.57510254  -0.80772246]`.

```
#linear regression with multiple features using gradient descent

# generate training examples, x(m,3), y(m,1)
np.random.seed(1969) # for reproducibility
m=10
x0=np.ones(m).reshape((m,1))
x=np.random.rand(m,2)
x=np.concatenate((x0,x),axis=1) # add the first column with all 1
w=np.array([4,6,-1]) # true model
y=np.dot(x,w)+0.5*np.random.randn(10)

#gradient descent algorithm for multiple features
cost=[]
n=x.shape[1] # n=3, the number of features

# initial values for parameters, theta=np.array([1,1,1])
theta=np.ones(n)

alpha=0.1 #learning rate
for iter in range(0, 1000):
    hypothesis=np.dot(x, theta)
    loss=hypothesis-y
    J=np.sum(loss**2)/(2*m)
    gradient=np.dot(np.transpose(x), loss)/m
    theta=theta-alpha*gradient
    cost.append(J)

# plot the loss function vs iteration steps
```

```
plt.plot(cost, 'b-')
plt.xlabel('iteration', fontsize="18")
plt.ylabel('loss', fontsize="18")
plt.grid(color = 'green', linestyle = '--', linewidth = 0.5)
plt.show()
print('true theta:',w)
print('predicted theta:',theta)
```



```
true theta: [ 4  6 -1]
predicted theta: [ 4.42621106  5.57510254 -0.80772246]
```

3.3 Linear Models for Regression

Since the output of linear regression is a linear function of the input variables, linear regression has significant limitations in many applications. In this section, we will extend the linear regression model to general linear models by considering the non-linear relationship between the output and the input variables.

3.3.1 Polynomial curve fitting

Suppose that we are given a training set comprising m observations of x and the corresponding values of the target y, denoted as $\{(x^{(i)}, y^{(i)}, i = 1,2, \dots, m)\}$. Our goal is to predict the value of y for a new value of x. In general, y is not a linear function of x, even approximately. One choice is to fit a polynomial function to the dataset. An n-order polynomial function can be written as

$$\hat{y}(x, \boldsymbol{\theta}) = \theta_0 + \theta_1 x + \theta_2 x^2 + \cdots + \theta_n x^n \tag{3.17}$$

where n is the order of the polynomial function, $\boldsymbol{\theta}$ is the vector comprising the polynomial coefficients $\theta_0, \theta_1 \dots \theta_n$.

Thus, the task of polynomial curve fitting is equivalent to a linear regression with n input features, $x, x^2, \cdots, x^n$. With a modification of matrix X, all conclusions in Section 3.2 can be applied to polynomial curve fitting. For example, the normal equation (3.16) can be used to compute the optimal value of $\boldsymbol{\theta}$ in an analytic form

$$\boldsymbol{\theta}^* = (X^\intercal X)^{-1} X^\intercal \mathbf{y} \tag{3.18}$$

where

$$\mathbf{X} = \begin{bmatrix} 1 & x^{(1)} & \left(x^{(1)}\right)^2 & \left(x^{(1)}\right)^3 \dots & \left(x^{(1)}\right)^n \\ 1 & x^{(2)} & \left(x^{(2)}\right)^2 & \left(x^{(2)}\right)^3 \dots & \left(x^{(2)}\right)^n \\ \vdots & \vdots & & & \vdots \\ 1 & x^{(m)} & \left(x^{(m)}\right)^2 & \left(x^{(m)}\right)^3 \dots & \left(x^{(m)}\right)^n \end{bmatrix} \tag{3.18a}$$

and

$$\mathbf{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix} \tag{3.18b}$$

Let us consider an example in which we try to fit polynomial functions (with different orders) to a dataset comprised of 10 sample points. The training set was generated by sampling the following function uniformly in range [0,1]

$$y = 10(x - 0.5)^2 - x + 0.5 \times \mathcal{N}(0,1) \tag{3.19}$$

where $\mathcal{N}(0,1)$ is a random number drawn from the standard Gaussian distribution. The sample points are plotted as blue dots in Fig.3.4, and the underlying true function $10(x - 0.5)^2 - x$ is plotted as the dash blue curve.

By the normal equation (3.18), we obtain the optimal polynomial coefficients $\boldsymbol{\theta}^*$. The resulting polynomial functions with different orders are plotted as the red solid lines in Fig.3.4.

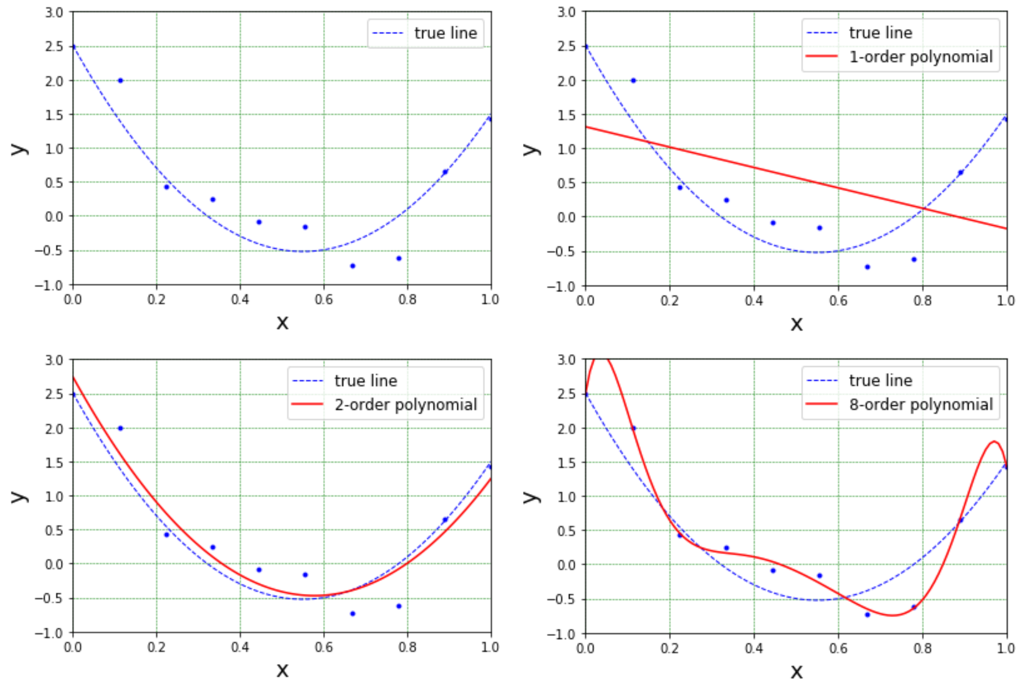


Fig.3.4 Plots of polynomials with different orders (shown as red curves) fitted to the data set (shown as blue dots)

The gradient descent algorithm in Section 3.2 can be exactly applied for polynomial curve fitting with the input data matrix X specified as in (3.18a). However, when the order of polynomial is high (say $n = 9$), it may be difficult for the algorithm to converge because the high-order features (say x^9) is much smaller than the low-order feature (e.g. x) for $0 \le x \le 1$, which results in a very slow convergence in the high-order feature dimensions.

From Fig.3.4, we can see that the polynomial with 1-order (i.e., linear) is not accurate enough to capture the feature of the data pattern, and the model is said to *underfit* the data set. We also find out that, on the other hand, if the order is too high (e.g., 8), the curve is *overfitting* the data set. When the overfitting occurs, the model minimizes the loss function just for a particular data set in order to perfectly fit all samples in this data set, and thus cannot generalize well for other data sets drawn from the same distribution.

3.3.2 Linear models with basis functions

The polynomial curve fitting, discussed in the previous section, is one of methods to model a nonlinear relationship. We can generalize the linear model (3.17) to a class of linear models, which are linear combinations of basis functions $\{\phi_j(\mathbf{x}), j = 0, 1, \dots, n\}$

$$\hat{y} = h_\theta(\mathbf{x}) = \boldsymbol{\theta}^T \boldsymbol{\phi}(\mathbf{x}) = \sum_{j=0}^{n} \theta_j \phi_j(\mathbf{x}) \tag{3.20}$$

where $\phi_j(\mathbf{x})$ are known as basis functions.

Thus, polynomial regression is an instance of (3.20) with basis function $\phi_j(x) = x^j, j = 0,1 \dots, m$, where m is the order of polynomial model. A set of quadratic basis functions for two-feature input can be

$$\boldsymbol{\phi}(\mathbf{x}) = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1 x_2 \\ x_1^2 \\ x_2^2 \end{bmatrix} \tag{3.21}$$

Thus, the linear model (3.20) with the basis functions (3.21) can model quadratic surfaces.

Other possible basis functions include Gaussian functions, sinc functions, sigmoid functions, and sinusoidal functions, defined respectively as

$$\phi_j(x) = e^{\frac{-(x-\mu_j)^2}{2s^2}} \tag{3.22}$$

$$\phi_j(x) = \frac{\sin\left(2\pi\omega(x-\mu_j)\right)}{2\pi\omega(x-\mu_j)} \tag{3.23}$$

$$\phi_j(x) = \sigma\left(\frac{x-\mu_j}{s}\right), \text{ where } \sigma(z) = \frac{1}{1+e^{-z}} \tag{3.24}$$

$$\phi_{cj}(x) = cos\left(2\pi\omega_j x\right), \phi_{sj}(x) = sin\left(2\pi\omega_j x\right) \tag{3.25}$$

where s, ω, ω_j control the shape of the functions, and μ_j specifies the location of shapes.

The loss function (3.13) can be generalized as

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(e^{(i)}\right)^2 = \frac{1}{2m}\boldsymbol{E}^\intercal\boldsymbol{E} = \frac{1}{2m}(\boldsymbol{\Phi}\cdot\boldsymbol{\theta} - \mathbf{y})^\intercal(\boldsymbol{\Phi}\cdot\boldsymbol{\theta} - \mathbf{y}) \tag{3.26}$$

where

$$\boldsymbol{\Phi} = \begin{bmatrix} \phi_0\left(x^{(1)}\right) & \phi_1\left(x^{(1)}\right) & \phi_2\left(x^{(1)}\right) & \phi_3\left(x^{(1)}\right)\dots & \phi_n\left(x^{(1)}\right) \\ \phi_0\left(x^{(2)}\right) & \phi_1\left(x^{(2)}\right) & \phi_2\left(x^{(2)}\right) & \phi_3\left(x^{(2)}\right)\dots & \phi_n\left(x^{(2)}\right) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \phi_0\left(x^{(m)}\right) & \phi_1\left(x^{(m)}\right) & \phi_2\left(x^{(m)}\right) & \phi_3\left(x^{(m)}\right)\cdots & \phi_n\left(x^{(m)}\right) \end{bmatrix}$$

Thus, the gradient of loss function with respect to the parameters is

$$grad = \frac{\partial J(\boldsymbol{\theta})}{\partial\boldsymbol{\theta}} = \begin{bmatrix} \frac{\partial J(\boldsymbol{\theta})}{\partial\theta_0} \\ \frac{\partial J(\boldsymbol{\theta})}{\partial\theta_1} \\ \vdots \\ \frac{\partial J(\boldsymbol{\theta})}{\partial\theta_n} \end{bmatrix} = \frac{1}{m}\boldsymbol{\Phi}^\intercal\cdot(\boldsymbol{\Phi}\cdot\boldsymbol{\theta} - \mathbf{y}) \tag{3.27}$$

The normal equation for optimal $\boldsymbol{\theta}$ is the solution to $\frac{\partial J(\boldsymbol{\theta})}{\partial\boldsymbol{\theta}} = \mathbf{0}$**,** given by

$$\boldsymbol{\theta}^* = (\boldsymbol{\Phi}^\intercal\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^\intercal\mathbf{y} \tag{3.28}$$

In the corresponding gradient descent algorithm, the parameters are updated by

$$\boldsymbol{\theta} \coloneqq \boldsymbol{\theta} - \alpha\cdot\frac{\partial J(\boldsymbol{\theta})}{\partial\boldsymbol{\theta}} \tag{3.29}$$

3.4 A Probabilistic Interpretation of Linear Regression

3.4.1 Equivalence of least square error and maximum likelihood estimation

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 y and feature variable $\mathbf{x} \in \mathbb{R}^{n+1}$ is approximately linear, thus we can estimate the target value for a new feature value using the linear model. The target and the input can be modeled as

$$y = \boldsymbol{\theta}^T\mathbf{x} + e = h_\theta(\mathbf{x}) + e \tag{3.30}$$

where e is a random noise taking unmodeled 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 a particular data example, there exists an error $e^{(i)}$

$$y^{(i)} = \boldsymbol{\theta}^T\mathbf{x}^{(i)} + e^{(i)} = h_\theta\left(\mathbf{x}^{(i)}\right) + e^{(i)} \tag{3.31}$$

If we know the probability distribution of *e*, we can express the uncertainty of *y* using the 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 $h_\theta(\mathbf{x})$ and a variance of σ^2,

$$p(y|\mathbf{x}; \boldsymbol{\theta}, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma}} exp\left(-\frac{\left(y - h_\theta(\mathbf{x})\right)^2}{2\sigma^2}\right) \tag{3.32}$$

We now use the training examples $\left\{\mathbf{x}^{(i)}, y^{(i)}, i = 1,2, \dots, m\right\}$ to determine the optimal value of the unknown parameters θ and σ^2 by maximum likelihood estimation. If the date examples are assumed to be drawn independently from the distribution (3.32), then the likelihood function is given by

$$\mathcal{L}(\theta, \sigma^2) = p(\mathbf{y}|\mathbf{x}; \theta, \sigma^2) = \prod_{i=1}^{m} p(y^{(i)}|\mathbf{x}^{(i)}; \theta, \sigma^2) = \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma}} exp\left(-\frac{\left(y^{(i)} - h_\theta\left(\mathbf{x}^{(i)}\right)\right)^2}{2\sigma^2}\right) \tag{3.33}$$

It is convenient to maximize the logarithm of the likelihood function (3.33). By applying the log operation, we obtain the log likelihood function in the form

$$\ell(\theta, \sigma^2) = \ln\left(\mathcal{L}(\theta, \sigma^2)\right) = -\frac{m}{2}\ln(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{m}\left(y^{(i)} - h_\theta\left(\mathbf{x}^{(i)}\right)\right)^2 \tag{3.34}$$

Instead of maximizing the likelihood function with respect to θ in (3.33), we can equivalently minimize the sum term of the right side of (3.34), $\frac{1}{2}\sum_{i=1}^{m}\left(y^{(i)} - h_\theta\left(\mathbf{x}^{(i)}\right)\right)^2$. We therefore see that maximizing the likelihood function is equivalent to minimizing the sum of the squares error function defined by (3.13). As a result, the optimal value of θ, θ_{ML}, can be calculated by

$$\theta_{ML} = argmin\frac{1}{2}\sum_{i=1}^{m}\left(y^{(i)} - h_\theta\left(\mathbf{x}^{(i)}\right)\right)^2 \tag{3.35}$$

Therefore, mathematically, *maximum likelihood estimation* (MLE) (3.35) is equivalent to the error square minimization problem (3.13), under the assumption of Gaussian distribution for the error.

Then, the optimal value of σ^2 can be obtained by maximizing (3.34) with respect to σ^2, with $\theta = \theta_{ML}$, as

$$\sigma^2{}_{ML} = \frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)} - h_{\theta_{ML}}\left(\mathbf{x}^{(i)}\right)\right)^2 \tag{3.36}$$

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 (3.13)).

Having determined the parameters θ and σ (3.35) and (3.36), we can make a prediction $\hat{y}$ for a new value of **x**, using the probabilistic model defined by (3.32), which gives the probability distribution of *y* rather than simply a point estimate. Of course, the estimation of y,

$$\hat{y} = h_{\theta_{ML}}(\mathbf{x}) \tag{3.37}$$

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

3.4.2 Bias and Variance

From Fig.3.4, we can see that the model *underfits* the data set if the model is too simple (e.g. low-order polynomials). The prediction curve has a large distance from the data examples (or the target curve). This implies that the model is not able to capture the basic important pattern of the data. On the other hand, the model *overfits* the data set if the model is too flexible (e.g., very high-order polynomials). In the case of overfitting, although the prediction curve fits the data set closely, it is very sensitive to the selection of data set. In other words, different data sets drawn from the same underlying distribution may result in very significant prediction curves.

A natural question is: which model is the best? The phenomenon of underfitting or overfitting is highly related to the bias and variance of the model. The bias of a model measures how far the model, in average over data sets, is away from the optimal model. The variance of a model indicates how sensitive the learned parameters of the model are to a particular data set. In this section, we will calculate the average loss of a model in terms of bias, variance, and intrinsic noise, and then describe the relationship between underfitting/overfitting and bias/variance. The analysis of bias and variance will shed lights on the model selection when we deal with complex models such as neural networks.

Suppose we have a data set whose examples $(\mathbf{x}, y)$ are independently drawn from

$$y = h(\mathbf{x}) + \epsilon \tag{3.38}$$

where $h(\mathbf{x})$ is the underlying target curve, ϵ is a Gaussian noise with a mean of zero. In general, the joint probability density of $(\mathbf{x}, y)$ is denoted as $p(\mathbf{x}, y)$. Note that the optimal model is the function $\mathbb{E}[y|\mathbf{x}] = h(\mathbf{x})$. However, we don't know the exact function $\mathbb{E}[y|\mathbf{x}]$ unless an unlimited amount of data examples is available.

We use a regression model, denoted as $\hat{y}(\mathbf{x}; \boldsymbol{\theta})$, to estimate the underlying optimal model $h(\mathbf{x})$, based on a dataset comprising of limited data examples. A common choice of loss function is the square loss

$$L(y, \hat{y}(\mathbf{x}; \boldsymbol{\theta})\,) = \left(y - \hat{y}(\mathbf{x}; \boldsymbol{\theta})\right)^2 \tag{3.39}$$

Thus, the expected loss, with respect to $p(\mathbf{x}, y)$, is

$$\mathbb{E}[L] = \mathbb{E}\left[\left(y - \hat{y}(\mathbf{x}; \boldsymbol{\theta})\right)^2\right] = \mathbb{E}\left[\left(y - h(\mathbf{x}) + h(\mathbf{x}) - \hat{y}(\mathbf{x}; \boldsymbol{\theta})\right)^2\right]$$

$$= \mathbb{E}\left[\left(y - h(\mathbf{x})\right)^2\right] + \mathbb{E}\left[\left(h(\mathbf{x}) - \hat{y}(\mathbf{x}; \boldsymbol{\theta})\right)^2\right] + 2\mathbb{E}\left[\left(y - h(\mathbf{x})\right)\left(h(\mathbf{x}) - \hat{y}(\mathbf{x}; \boldsymbol{\theta})\right)\right]$$

$$= \mathbb{E}\left[\left(y - h(\mathbf{x})\right)^2\right] + \mathbb{E}\left[\left(h(\mathbf{x}) - \hat{y}(\mathbf{x}; \boldsymbol{\theta})\right)^2\right]$$

$$= \iint \left(y - h(\mathbf{x})\right)^2 p(\mathbf{x}, y)\,dy\,d\mathbf{x} + \iint \left(h(\mathbf{x}) - \hat{y}(\mathbf{x}; \boldsymbol{\theta})\right)^2 p(\mathbf{x}, y)\,dy\,d\mathbf{x}$$

$$= \iint \left(y - h(\mathbf{x})\right)^2 p(\mathbf{y}|\mathbf{x})\,dy \cdot p(\mathbf{x})\,d\mathbf{x} + \int \left(h(\mathbf{x}) - \hat{y}(\mathbf{x}; \boldsymbol{\theta})\right)^2 p(\mathbf{x})\,d\mathbf{x}$$

$$= \int Var(y|\mathbf{x})\, p(\mathbf{x})\,d\mathbf{x} + \int \left(h(\mathbf{x}) - \hat{y}(\mathbf{x}; \boldsymbol{\theta})\right)^2 p(\mathbf{x})\,d\mathbf{x}$$

$$= \int Var(\epsilon|\mathbf{x})\, p(\mathbf{x})d\mathbf{x} + \int \left(h(\mathbf{x}) - \hat{y}(\mathbf{x};\boldsymbol{\theta})\right)^2 p(\mathbf{x})d\mathbf{x} \tag{3.40}$$

where $p(\mathbf{x})$ is the probability density function of $\mathbf{x}$, calculated by

$$p(\mathbf{x}) = \int p(\mathbf{x}, y)dy \tag{3.40a}$$

The first term in (3.40) is the variance of the noise, averaged over $\mathbf{x}$. It is usually reasonable to assume that $\mathbf{x}$ is independent of the noise ϵ. In this case, the first term is reduced to the variance of the noise. It represents the minimum loss that is achieved only when the second term in (3.40) is equal to zero, i.e., the optimal model is found $\hat{y}(\mathbf{x};\boldsymbol{\theta}) = h(\mathbf{x})$. Thus, it is independent of the learned model, and is irreducible.

The second term in (3.40) is the loss component that corresponds to the misfit between the learned model $\hat{y}(\mathbf{x};\boldsymbol{\theta})$and the optimal model $h(\mathbf{x})$. If we had an unlimited amount of training data examples $(\mathbf{x}, y)$, we would be able to obtain the optimal model by $h(\mathbf{x}) = \mathbb{E}[y|\mathbf{x}]$. In fact, we use a parametric function $\hat{y}(\mathbf{x};\boldsymbol{\theta})$ to model $h(\mathbf{x})$ based on a particular data set with a size of m. Different data sets independently drawn from $p(\mathbf{x}, y)$ will result in different learned parameters $\boldsymbol{\theta}$ and consequently different values of the loss. Now we will decompose the second term in (3.40) into two components: one associated with the bias, and another associated with the variance.

For a given data set $\mathcal{D}$, the resulting model is denoted as $\hat{y}(\mathbf{x};\boldsymbol{\theta}_{\mathcal{D}})$. The average of the models over all possible data sets is denoted as $\mathbb{E}_{\mathcal{D}}[\hat{y}(\mathbf{x};\boldsymbol{\theta}_{\mathcal{D}})]$. For a data set $\mathcal{D}$, the square term in (3.40), $\left(h(\mathbf{x}) - \hat{y}(\mathbf{x};\boldsymbol{\theta})\right)^2$, can be represented as

$$\left(h(\mathbf{x}) - \hat{y}(\mathbf{x};\boldsymbol{\theta}_{\mathcal{D}})\right)^2 = \left(h(\mathbf{x}) - \mathbb{E}_{\mathcal{D}}[\hat{y}(\mathbf{x};\boldsymbol{\theta}_{\mathcal{D}})] + \mathbb{E}_{\mathcal{D}}[\hat{y}(\mathbf{x};\boldsymbol{\theta}_{\mathcal{D}})] - \hat{y}(\mathbf{x};\boldsymbol{\theta}_{\mathcal{D}})\right)^2$$

$$= (h(\mathbf{x}) - \mathbb{E}_{\mathcal{D}}[\hat{y}(\mathbf{x};\boldsymbol{\theta}_{\mathcal{D}})])^2 + \left(\mathbb{E}_{\mathcal{D}}[\hat{y}(\mathbf{x};\boldsymbol{\theta}_{\mathcal{D}})] - \hat{y}(\mathbf{x};\boldsymbol{\theta}_{\mathcal{D}})\right)^2$$

$$+2(h(\mathbf{x}) - \mathbb{E}_{\mathcal{D}}[\hat{y}(\mathbf{x};\boldsymbol{\theta}_{\mathcal{D}})])\left(\mathbb{E}_{\mathcal{D}}[\hat{y}(\mathbf{x};\boldsymbol{\theta}_{\mathcal{D}})] - \hat{y}(\mathbf{x};\boldsymbol{\theta}_{\mathcal{D}})\right) \tag{3.41}$$

Taking the expectation on both sides of (3.41) with respect to data sets, we have

$$\mathbb{E}_{\mathcal{D}}\left[\left(h(\mathbf{x}) - \hat{y}(\mathbf{x};\boldsymbol{\theta}_{\mathcal{D}})\right)^2\right] = \mathbb{E}_{\mathcal{D}}[(h(\mathbf{x}) - \mathbb{E}_{\mathcal{D}}[\hat{y}(\mathbf{x};\boldsymbol{\theta}_{\mathcal{D}})])^2] + \mathbb{E}_{\mathcal{D}}\left[\left(\mathbb{E}_{\mathcal{D}}[\hat{y}(\mathbf{x};\boldsymbol{\theta}_{\mathcal{D}})] - \hat{y}(\mathbf{x};\boldsymbol{\theta}_{\mathcal{D}})\right)^2\right]$$

$$= (h(\mathbf{x}) - \mathbb{E}_{\mathcal{D}}[\hat{y}(\mathbf{x};\boldsymbol{\theta}_{\mathcal{D}})])^2 + Variance[\hat{y}(\mathbf{x};\boldsymbol{\theta}_{\mathcal{D}})]$$

$$= (prediction\ bias)^2 + prediction\ variance \tag{3.42}$$

Thus, combining (3.40) and (3.42), we obtain the total expected loss for a single point $\mathbf{x}$**,** with respect to data sets,

$$\mathbb{E}_{\mathcal{D}}[L(\mathbf{x})] = noise\ variance + (prediction\ bias)^2 + prediction\ variance \tag{3.43}$$

where

$$noise\ variance = Variance(\epsilon|\mathbf{x}) \tag{3.43a}$$

$$(prediction\ bias)^2 = (h(\mathbf{x}) - \mathbb{E}_{\mathcal{D}}[\hat{y}(\mathbf{x};\boldsymbol{\theta}_{\mathcal{D}})])^2 \tag{3.43b}$$

$$prediction\ variance = \mathbb{E}_{\mathcal{D}}\left[\left(\mathbb{E}_{\mathcal{D}}[\hat{y}(\mathbf{x};\boldsymbol{\theta}_{\mathcal{D}})] - \hat{y}(\mathbf{x};\boldsymbol{\theta}_{\mathcal{D}})\right)^2\right] \tag{3.43c}$$

The total expectation of the lose

$$\mathbb{E}[L] = \int \mathbb{E}_{\mathcal{D}}[L(\mathbf{x})]\, p(\mathbf{x}) d\mathbf{x} \tag{3.44}$$

Equation (3.43) reveals important insights in the performance of a model. On average, the loss of a model can be decomposed into three parts: 1) intrinsic noise in the data set; 2) the squared bias from the true underlying function; and 3) the variance of its predictions. The bias and the variance depend on the selection of a model and the size of training data set.

In general, a simple model is likely to have a large bias but a small variance, and thus has a problem of underfitting. In contrast, a highly flexible model is sensitive to the selection of training set due to the limited size of the training set, and thus has a high variance but a low bias, which leads to an overfitting. This can be demonstrated by the following experiment. First, we independently generate 10 training sets with 10 sample points per data set, from the same distribution defined by (3.19), copied here

$$y = 10(x - 0.5)^2 - x + 0.5 \times \mathcal{N}(0,1)$$

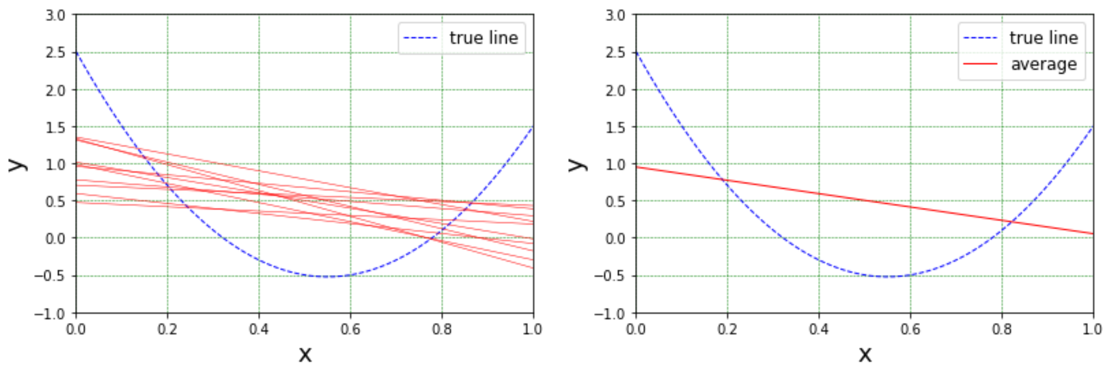
x is sampled fixedly and uniformly (i.e., equal distance) in the range of $[0,1)$, and y is generated by the above equation. Thus, the underlying true relationship is

$$h(x) = 10(x - 0.5)^2 - x \tag{3.45}$$

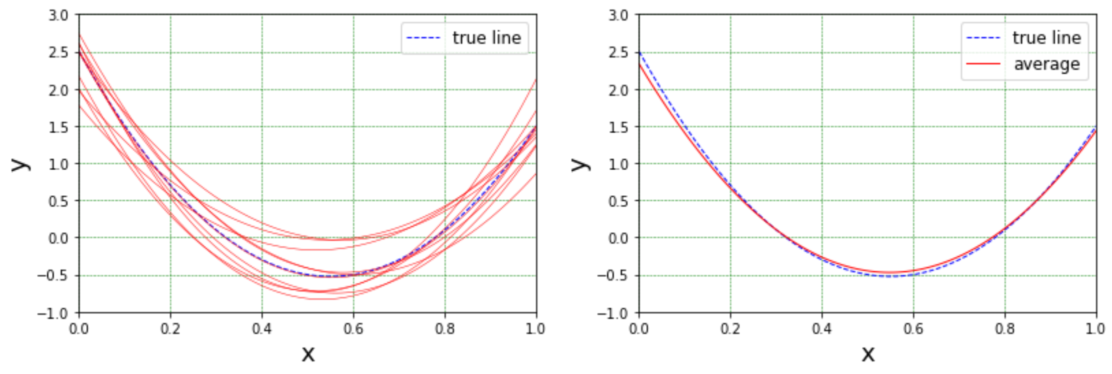
which is a second order polynomial and plotted as the blue dashed lines in Fig.3.5.

Then, we select a certain-order polynomial model, and independently fit the model to each data set. As a result, we obtain 10 prediction curves corresponding to 10 data sets, shown in the left figures in Fig.3.5. The difference between the true line and the average of 10 prediction curves indicates the prediction *bias*, shown in the right figures in Fig.3.5. The variation of the 10 curves shows the prediction *variance*.

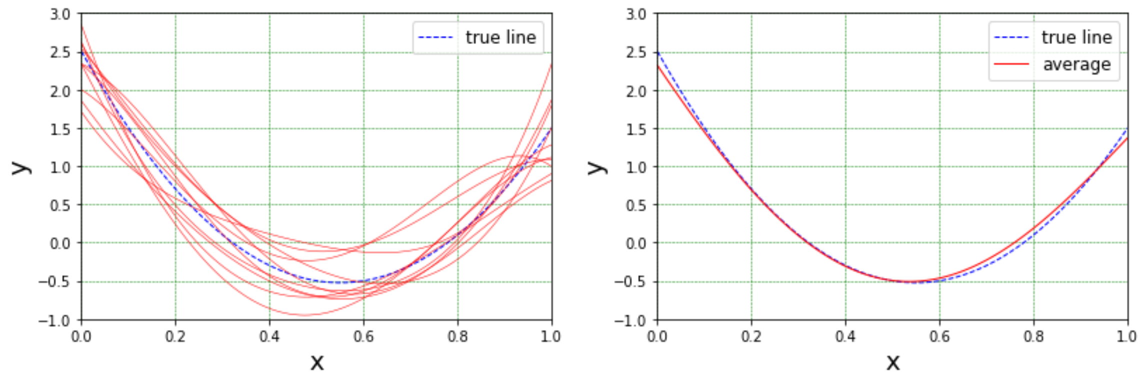
Fig.3.5 shows the results of the experiment. In Fig.3.5(a), the linear model is obviously not sufficient to fit a second order polynomial relationship, thus has a large bias. As we expect, the second order polynomial model is a good fit for the data sets, shown in Fig.3.5(b). The higher order polynomials exhibit some degrees of high variance or overfitting, as shown in Fig.3.5 (c) and (d), but the average over data sets fits the underlying true relationship very well if the number of data sets is large.



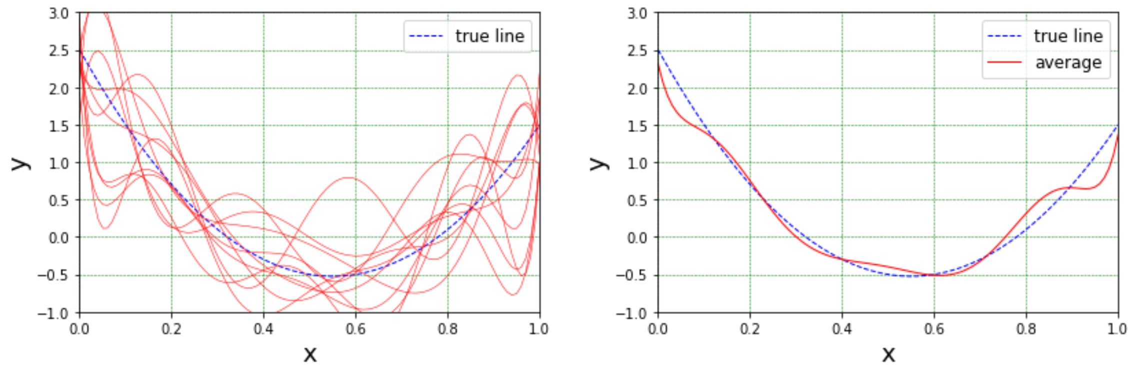
(a) First-order polynomial (linear): high bias, low variance



(b) second-order polynomial: a good fit



(c) fourth-order polynomial: low bias, high variance



(d) eighth-order polynomial: low bias, high variance

Fig.3.5 curve fitting with different order polynomials using different data sets

Thus, to avoid underfitting and overfitting, it is crucial to select a model with an appropriate complexity. There are two strategies to alleviate overfitting: 1) reducing the complexity of the model or applying regularization (discussed in chapter 4), and 2) increasing the size of the training data set. Note that the analysis of the bias and the variance does not require $h(\mathbf{x})$ to be linear. The concepts and implications in this section can be applied to non-linear models, such as neural networks.

3.5 An Example: House Price Prediction

In this section we will use an example to demonstrate how we can apply the linear regression model to solve the real problems. Through this example, we address some common issues in machine learning: train/test data set partition, feature scaling, learning rate selection, etc.

3.5.1 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 convergence.

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.3.6(a). A balanced scale across features will likely have a relatively straight converge path for search the optimal parameters, as shown in Fig.3.6(b).

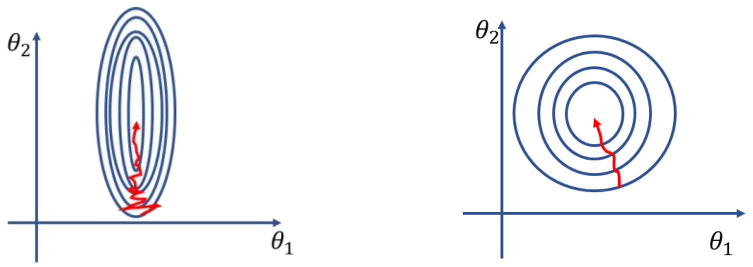


Fig.3.6 Contour plots and converge path (red arrowed lines):
(a) unbalanced feature scale, (b) balanced feature scale.

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

$$z_i = \frac{x_i - \min_j\{x_i^{(j)}\}}{\max_j\{x_i^{(j)}\} - \min_j\{x_i^{(j)}\}} \tag{3.46}$$

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. The standardized feature of a feature x_i is calculated as:

$$z_i = \frac{x_i - u_i}{s_i} \tag{3.47}$$

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

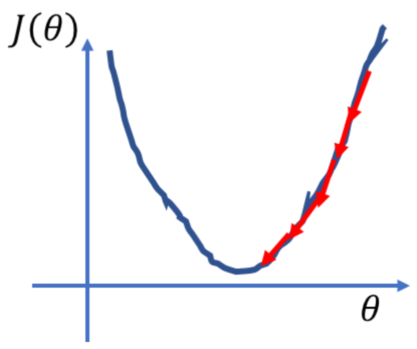
Choice of learning rate

In the gradient descent, the parameters are updated as

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \tag{3.48}$$

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.3.7. Thus, we can use a plot of cost function versus number of iterations to monitor whether the gradient descent is working correctly or not. The plots in Fig.3.8(b) and (c) indicate that the value of α is too large and consequently results in overshooting during the update of θ. The cost will oscillate or even increase during the training process.

In general, 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.



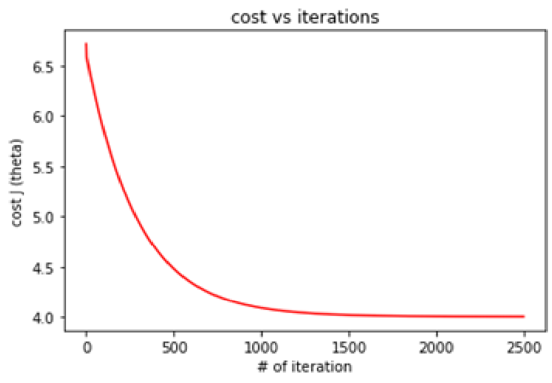


Fig.3.7 an appropriate value for α, (a) converge path, (b) cost function vs # of iterations

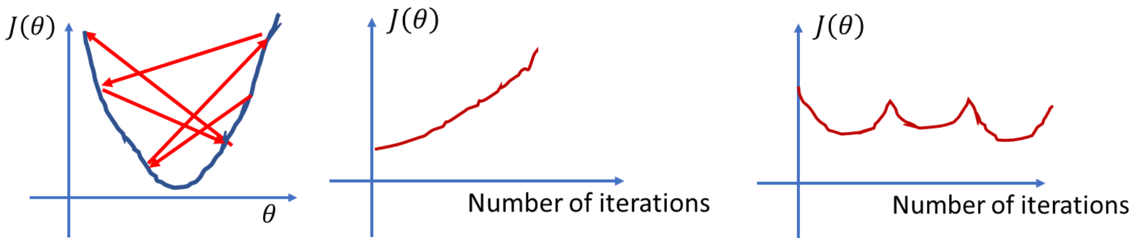


Fig.3.8 value of α is too large, (a) the path does not converge, (b) and (c) possible plots of cost function.

3.5.2 Linear regression for house price prediction in Python

Problem and dataset

Suppose that the price of a house (in a particular region) depends on some features, such as square footage, the number of bedrooms, the number of bathrooms, the number of stories, parking space, whether on a main road, whether furnished, etc., and that we have a dataset comprised of samples on the prices of such houses. We will build a linear regression model to predict the price of a house given its features.

The dataset can be downloaded at kaggle.com, as *Housing.csv*. This dataset includes 545 house price samples, with 12 features per sample. The details of the dataset will be described when we proceed with the steps of the project in the following text.

Steps of the project

1) Understanding the dataset

First, import the required packages and read the dataset file using pandas DataFrame.

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

# read the data file
housing = pd.DataFrame(pd.read_csv("Housing.csv"))
print('housing shape:', housing.shape)
housing.head()
```

```
housing shape: (545, 13)
```

	price	area	bedrooms	bathrooms	stories	mainroad	guestroom	basement	hotwaterheating	airconditioning	parking	prefarea	furnishingstatus
0	13300000	7420	4	2	3	yes	no	no	no	yes	2	yes	furnished
1	12250000	8960	4	4	4	yes	no	no	no	yes	3	no	furnished
2	12250000	9960	3	2	2	yes	no	yes	no	no	2	yes	semi-furnished
3	12215000	7500	4	2	2	yes	no	yes	no	yes	3	yes	furnished
4	11410000	7420	4	1	2	yes	yes	yes	no	yes	2	no	furnished

The dataset is loaded to a DataFrame, named *housing*, which has 545 rows and 13 columns. Each row represents one sample. The first column lists the prices, and other columns specify the relevant 12 features. Some features are specified by numerical numbers while others by words (e.g., yes, no, furnished, semi-furnished, unfurnished).

2) Data pre-processing

Convert descriptive features to numerical features

We substitute the descriptive feature values to numerical values according to the mapping: “no”→0, “yes”→1, “unfurnished”→0, “semi-furnished”→1, “furnished”→2.

```python
# List of variables to map

varlist =  ['mainroad', 'guestroom', 'basement', 'hotwaterheating', 'airconditi
oning', 'prefarea']

# Defining the map function
def binary_map(x):
    return x.map({'yes': 1, "no": 0})

# Applying the function to the housing list
housing[varlist] = housing[varlist].apply(binary_map)

housing['furnishingstatus'] = housing['furnishingstatus'].replace(['unfurnished
'], 0)
housing['furnishingstatus'] = housing['furnishingstatus'].replace(['semi-furnis
hed'], 1)
housing['furnishingstatus'] = housing['furnishingstatus'].replace(['furnished']
, 2)
```

```
housing.head()
```

	price	area	bedrooms	bathrooms	stories	mainroad	guestroom	basement	hotwaterheating	airconditioning	parking	prefarea	furnishingstatus
0	13300000	7420	4	2	3	1	0	0	0	1	2	1	2
1	12250000	8960	4	4	4	1	0	0	0	1	3	0	2
2	12250000	9960	3	2	2	1	0	1	0	0	2	1	1
3	12215000	7500	4	2	2	1	0	1	0	1	3	1	2
4	11410000	7420	4	1	2	1	1	1	0	1	2	0	2

Split dataset into a training set and a test set

```
from sklearn.model_selection import train_test_split

np.random.seed(0) # to repeat to generate the same result.
df_train, df_test = train_test_split(housing, train_size = 0.7, test_size = 0.3
, random_state = 100)
```

Load the data to numpy arrays separately in terms of train, test, features, and labels, and scale the price down by 1000 for a numerical convenience.

```
train=df_train.to_numpy()
test=df_test.to_numpy()
X_train=train[:,1:]
y_train=train[:,0]/1000. # scaled down by 1000
X_test=test[:,1:]
y_test=test[:,0]/1000.   # scaled down by 1000
```

Scale features

On the training data, we normalize each feature into the range [0,1] by a function *MinMaxScaler()*. 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. To predict the price of the house with a feature vector x, we must first normalize x using the same value of parameters (such as *feature_min*, *feature_max*, *feature*) that we had previously used for the training set normalization.

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

X_train_scaled = scaler.fit_transform(X_train) #train is numpy array (381,12)
feature_min=scaler.data_min_    # shape (12.)
feature_max=scaler.data_max_    # shape (12,)
feature_range=scaler.data_range_   # shape (12,)

X_test_scaled=(X_test-feature_min)/feature_range # scale test features
```

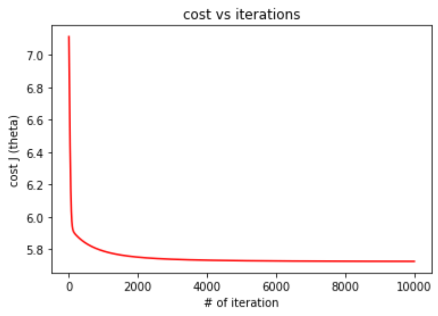
3) Define and run gradient_descent function for linear regression.

```
# x(m,n): m is the number of samples, n is the number of features

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) # now, x(m, n+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)
        #theta_list.append(theta)
    return theta,cost
```

```
theta ,cost= gradient_descent(alpha=0.01, x=X_train_scaled, y=y_train, numItera
tions=10000)

theta
```

```
array([1662.57265997, 2466.86385012,  693.75803667, 2208.83766241,
 1212.95881281,  614.7736432 ,  362.00404946,  247.70616561,
 1002.98979197,  775.37672733,  759.95528449,  705.08284328,
  381.26621482])
```

```
plt.plot(np.log10(cost), color='red')
plt.title('cost vs iterations')
plt.xlabel('# of iteration')
plt.ylabel('cost J (theta)')
plt.show()
```



4) *Validate by test dataset.*

Now, we use the trained model to predict the prices of the houses in the test dataset and show both the true prices and predicted prices.

```
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)
```

```
Y_test_pred=predict(X_test_scaled,theta)
```

```
print("Area        #of bds    real_price    predict from Linear reg ")
for i in range (0,len(X_test_scaled)):
    T=X_test[i]
    print("{0:8.2f},{1:8.2f},{2:12.2f},{3:20.2f}".format(T[0],T[1],y_test[i],Y_
test_pred[i]))
```

```
Area        #of bds    real_price    predict from Linear reg
 2880.00,     3.00,      4403.00,              4081.51
 6000.00,     3.00,      7350.00,              6548.83
10269.00,     3.00,      5250.00,              5534.49
 5320.00,     3.00,      4550.00,              5220.89
 4950.00,     4.00,      4382.00,              4739.91
 4320.00,     3.00,      4690.00,              3989.78
  ...
```

Summary and Further Reading

In this chapter, we introduced a type of models, called linear regressions. We described two ways to learn or fit the models: the normal equation (analytic solution) and the gradient descent algorithm. Both ways are based on minimizing a cost function that specifies the error between the predictions and the true values.

The normal equation provides theoretical optimal solutions to the parameters of a model by solving a system of linear equations. However, its implementation is not computationally efficient in practice. The gradient descent algorithm iteratively searches for the optimal solutions along the negative direction of the gradient of the cost function with respect to parameters. The gradient descent algorithm and its variants are widely applied in training complex models (e.g., deep neural networks). There are some practical issues relevant to convergence, such as the choice of learning rate and feature scaling.

The concept of simple linear regression can be extended to a linear model with multiple features corresponding to a set of basis functions. One such example is a polynomial curve fitting, which has been treated in detail.

It is very helpful to view the linear regression model from a probabilistic perspective. The linear regression with least square errors is equivalent to maximum likelihood estimation. We analyze the expectation of loss in terms of bias and variance of a model, given the size of the training dataset. Overfitting or underfitting can be diagnosed based on the measurement of bias and variance. From the insightful analysis, one can see how the training dataset size and the complexity of the model affect the bias and variance. As we will see in later chapters, the conclusions on the trade-off between bias and variance can be applied to general machine learning models.

After finishing the chapter, one should understand the theoretical aspects of linear regression and be able to implement a linear regression in Python from scratch.

Files: C:\Users\weido\ch1_linear_reg\ch3_linear_reg.ipynb, Housing.csv.

Further reading

[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] Bishop, C. M. Bishop, Chapter 3 Linear models for regression, in Pattern recognition and machine *learning*. Springer, 2006.

Exercises

1. Derive the normal equation (3.5a) and (3.5b) for linear regression with one feature.

2. Derive the normal equation (3.16) for linear regression with multiple features.

3. 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)

4. 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?

5. Suppose you train a linear regression to fit the dataset (x,y) in Fig.3.3(a). The dataset can be generated by the following statements:

```
# Fixing random state for reproducibility
np.random.seed(1968)
x=np.random.rand(10)
y=4+6*x+np.random.randn(10)
```

   In this exercise, you are asked to plot the first five learned lines during the first five iterations, along with the data samples. Since different initial values of θ lead to different results, you are asked to repeat the exercise using two different initial values of θ.

   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.

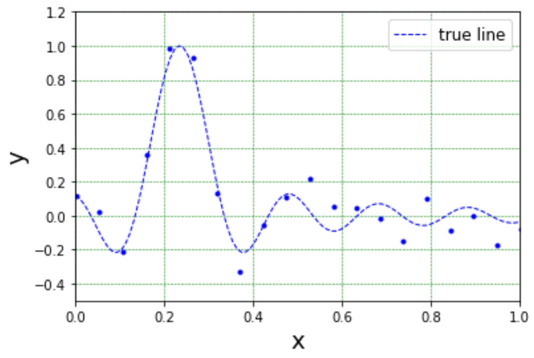
6. In this exercise, you are asked to generate an artificial dataset based on a linear model, and then learn the model using gradient descent. Specifically,
   1) Generate 100 data examples $(x^{(i)}, y^{(i)}), i = 1,2, \ldots, 100$, which are **randomly** drawn from the model
   $$y = a + bx + N(0,1)$$
   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 random rather than uniform.

   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.

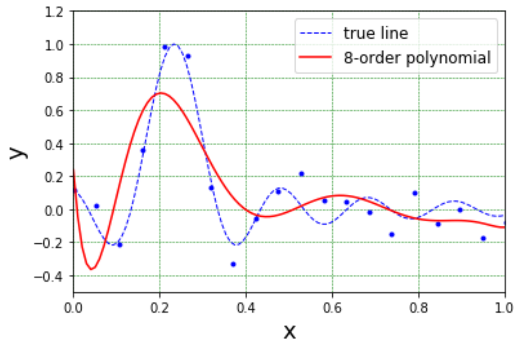
7. In this exercise, you are asked to generate an artificial dataset based on a nonlinear model, and then fit it to a polynomial model. Specifically,
   1) Generate 20 data examples $(x^{(i)}, y^{(i)}), i = 0,1,2, \ldots, 19$, which are drawn from the model

$$y = \frac{\sin\big(2\pi \times 5 \times (x - 0.234)\big)}{2\pi \times 5 \times (x - 0.234)} + 0.1 \times \mathcal{N}(0,1)$$

where $\mathcal{N}(0,1)$ is a random number that follows Gaussian distributions with mean=0 and variance=1. $x^{(i)}$ are sampled uniformly (i.e., with equal distance) from 0 and 1. Plot your generated data examples and the underlying sinc function curve in one figure, something like the figure below.



2) Using the normal equation (3.16), learn a polynomial model $y = \theta_0 + \theta_1 x + \theta_2 x^2 + \cdots + \theta_n x^n$ to fit the generated examples, for $n = 8$. Plot the examples, underlying sinc function and the learned polynomial function. The plots should be something like below.



3) Repeat 2) for $n = 1, 2, 3, 10, 20$.

4) Using the gradient descent algorithm (instead of the normal equation), learn a polynomial model $y = \theta_0 + \theta_1 x + \theta_2 x^2 + \cdots + \theta_n x^n$ to fit the generated examples for $n = 10$. Compare the result with the result obtained in 3) by the normal equation. Is there any problem with this result? Why? If there is a problem, discuss the ways to fix it.

8. In your own words, describe the bias and variance of a model. What is the trade-off between bias and variance? In general, how do the bias and variance change when the complexity of a model increases? How do the bias and variance change when the size of training dataset increases?