HCIA-AI Huawei Certification Course

HCIA-AI
Math Basics
Experimental Guide

Version: 1.0

Huawei Technologies Co., Ltd.

Trademarks and Permissions

 and other Huawei trademarks are trademarks of Huawei Technologies Co., Ltd.

All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between Huawei and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

Huawei Technologies Co., Ltd.

Address: Huawei Industrial Base
Bantian, Longgang
Shenzhen 518129
People's Republic of China

Website: http://huawei.com

Contents

1 Basic Math Experiment

1.1 Introduction

1.1.1 Contents

Basic mathematics knowledge, including algorithm design and numerical processing knowledge, is widely applied in the Artificial Intelligence (AI) domain, especially in the field of traditional machine learning and deep learning. The main purpose of this section is to implement some common mathematical knowledge based on the Python language and basic mathematics modules to provide basic support for AI learning.

1.1.2 Framework

This document mainly uses the **math** library, **numpy** library, and **scipy** library. The **math** library, a standard library of Python, provides some common mathematical functions; the **numpy** library, a numerical calculation and expansion library of Python, is mainly used to handle issues such as linear algebra, random number generation, and Fourier Transform; the **scipy** library is used to deal with issues such as statistics, optimization, interpolation, and bonus points.

1.2 Implementation

Import corresponding modules in advance.

```
>>> import math
```

```
>>> import numpy as np
```

1.2.1 ceil Implementation

The value of **ceil(*x*)** is the minimum integer greater than or equal to *x*. If *x* is an integer, the returned value is *x*.

Code:

```
>>> math.ceil(4.01)
```

Output:

```
>>> 5

# Code:

>>> math.ceil(4.99)

# Output:

>>> 5
```

1.2.2 floor Implementation

The value of **floor(*x*)** is the maximum integer less than or equal to *x*. If *x* is an integer, the returned value is *x*.

```
# Code:

>>> math.floor(4.1)

# Output:

>>> 4

# Code:

>>> math.floor(4.999)

# Output:

>>> 4
```

1.2.3 cos Implementation

The **cos(*x*)** parameter is the cosine of ***x***, where ***x*** must be a radian (**math.pi/4** is a radian, indicating an angle of 45 degrees).

```
# Code:

>>> math.cos(math.pi/4)

# Output:

>>> 0.7071067811865476

# Code:

>>> math.cos(math.pi/3)

# Output:

>>> 0.5000000000000001
```

1.2.4 tan Implementation

The **tan(*x*)** parameter returns the tangent value of *x* (radian).

```
# Code:

>>> tan(pi/6)

# Output:

>>> 0.5773502691896257
```

1.2.5 degrees Implementation

The **degrees(*x*)** parameter converts *x* from a radian to an angle.

Code:

>>> math.degrees(math.pi/4)

Output:

>>> 45.0

Code:

>>> math.degrees(math.pi)

Output:

>>> 180.0

1.2.6 exp Implementation

The **exp(*x*)** parameter returns **math.e**, that is, the *x* power of **2.71828**.

Code:

>>> math.exp(1)

Output:

>>> 2.718281828459045

1.2.7 fabs Implementation

The **fabs(*x*)** parameter returns the absolute value of *x*.

Code:

>>> math.fabs(-0.003)

Output:

>>> 0.003

1.2.8 factorial Implementation

The **factorial(*x*)** parameter is the factorial of *x*.

Code:

>>> math.factorial(3)

Output:

>>> 6

1.2.9 fsum Implementation

The **fsum(iterable)** summarizes each element in the iterator.

Code:

```
>>> math.fsum([1,2,3,4])

# Output:

>>>10
```

1.2.10 fmod Implementation

The **fmod(x, y)** parameter obtains the remainder of *x/y*. The value is a floating-point number.

```
# Code:

>>> math.fmod(20,3)

# Output:

>>>2.0
```

1.2.11 log Implementation

The **log([*x*, base])** parameter returns the natural logarithm of *x*. By default, **e** is the base number. If the **base** parameter is fixed, the logarithm of *x* is returned based on the given **base**. The calculation formula is **log(x)/log(base)**.

```
# Code:

>>> math.log(10)

# Output:

>>> 2.302585092994046
```

1.2.12 sqrt Implementation

The **sqrt(*x*)** parameter indicates the square root of *x*.

```
# Code:

>>> math.sqrt(100)

# Output:

>>>10.0
```

1.2.13 pi Implementation

The **pi** parameter is a numeric constant, indicating the circular constant.

```
# Code:

>>> math.pi

# Output:

>>> 3.141592653589793
```

1.2.14 pow Implementation

The **pow(*x*, *y*)** parameter returns the *x* to the *y*th power, that is, *x****y*.

```
# Code:
```

```
>>> math.pow(3,4)
# Output:
>>> 81.0
```

1.2.15 trunc Implementation

The **trunc(*x*:Real)** parameter returns the integer part of *x*.

```
# Code:
>>> math.trunc(6.789)
# Output:
>>> 6
```

2 Linear Algebra Experiment

2.1 Introduction

2.1.1 Linear Algebra

Linear algebra is a mathematical branch widely used in various engineering technical disciplines. Its concepts and conclusions can greatly simplify the derivation and expression of AI formulas. Linear algebra can simplify complex problems so that we can perform efficient mathematical operations.

In the context of deep learning, linear algebra is a mathematical tool that provides a technique that helps us to operate arrays at the same time. Data structures like vectors and matrices can store numbers and rules for operations such as addition, subtraction, multiplication, and division.

2.1.2 Code Implementation

The numpy is a numerical processing module based on Python. It has powerful functions and advantages in processing matrix data. As linear algebra mainly processes matrices, this section is mainly based on the numpy. This section also uses the mathematical science library scipy to illustrate equation solution.

2.2 Linear Algebra Implementation

Import corresponding modules in advance.

```
>>> import numpy as np
```

```
>>> import scipy as sp
```

2.2.1 Tensor Implementation

Generate a two-dimensional tensor whose elements are **0** and two dimensions are **3** and **4**.

Code:

```
>>> np.zeros((3,4))
```

Output:

```
>>> np.array([[ 0.,    0.,    0.,    0.],
              [ 0.,    0.,    0.,    0.],
              [ 0.,    0.,    0.,    0.]])
```

Generate a three-dimensional random tensor whose three dimensions are **2**, **3**, and **4** respectively.

Code:

```
>>> np.random.rand(2,3,4)
```

Output:

```
>>> array([[[ 0.93187582,    0.4942617 ,    0.23241437,    0.82237576],
            [ 0.90066163,    0.30151126,    0.89734992,    0.56656615],
            [ 0.54487942,    0.80242768,    0.477167   ,    0.6101814 ]],

           [[ 0.61176321,    0.11454075,    0.58316117,    0.36850871],
            [ 0.18480808,    0.12397686,    0.22586973,    0.35246394],
            [ 0.01192416,    0.5990322 ,    0.34527612,    0.424322    ]]])
```

2.2.2 Matrix Implementation

In mathematics, a matrix is a set of complex numbers or real numbers arranged by a rectangular array. A matrix is derived from a square array consisted of coefficients and constants of an equation set.

Create a 3x3 zero matrix. The parameter of the **zeros** function in the matrix is of the tuple type **(3, 3)**.

Code:

```
>>> np.mat(np.zeros((3,3)));
```

Output:

```
>>> matrix([[0., 0., 0.],
            [0., 0., 0.],
            [0., 0., 0.]])
```

2.2.3 Identity Matrix Implementation

Identity matrix is a square array whose diagonal elements are all **1** and other elements are **0**.

Code:

```
>>> np.eye(4)
```

Output:

```
>>> array([[ 1.,    0.,    0.,    0.],
           [ 0.,    1.,    0.,    0.],
```

[0., 0., 1., 0.],

[0., 0., 0., 1.]])

2.2.4 reshape Operation

There is no 'reshape' operation in mathematics, but it is a very common operation in the operation libraries such as the numpy and TensorFlow. The reshape operation is used to change the dimension number of a tensor and size of each dimension. For example, a 10x10 image is directly saved as a sequence containing 100 elements. After the system reads the image, you can transform the image from 1x100 to 10x10 through the reshape operation. The example is as follows:

Code:

Generate a vector that contains integers from **0** to **11**.

>>> x = np.arange(12)

>>> x

Output:

>>> array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])

View the array size.

>>> x.shape

Output:

>>>(12,)

Convert the matrix x into a two-dimensional matrix, where the first dimension of the matrix is **1**.

>>> x = x.reshape(1,12)

>>> x

Output:

>>> array([[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]])

>>> x.shape

Output:

>>>(1, 12)

Convert *x* to a 3x4 matrix.

>>> x = x.reshape(3,4)

>>> x

Output:

>>> array([[0, 1, 2, 3],

[4, 5, 6, 7],

[8, 9, 10, 11]])

2.2.5 Transposition Implementation

The transposition of vectors and matrices is exchanging the row and column. For the transposition of the tensors in three dimensions and above, you need to specify the transposition dimension.

Code:

Generate a vector ***x*** **c**ontaining five elements and transposes the vector.

```
>>> x = np.arange(5).reshape(1,-1)
>>> x
array([[0, 1, 2, 3, 4]])
>>> x.T
array([[0],
       [1],
       [2],
       [3],
       [4]])
```

Generate a 3x4 matrix and transpose the matrix.

```
>>> A = np.arange(12).reshape(3,4)
>>> A
# Output:
>>> array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> A.T
# Output:
>>> array([[ 0,  4,  8],
       [ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11]])
```

Generate a 2x3x4 tensor.

```
>>> B = np.arange(24).reshape(2,3,4)
>>> B
# Output:
>>> array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
```

```
       [ 8,    9, 10, 11]],


      [[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]]])
```

Transpose the **0** and **1** dimensions of B.

```
>>> B.transpose(1,0,2)
# Output:
>>> array([[[ 0,    1,    2,    3],
        [12, 13, 14, 15]],

       [[ 4,    5,    6,    7],
        [16, 17, 18, 19]],

       [[ 8,    9, 10, 11],
        [20, 21, 22, 23]]])
```

2.2.6 Matrix Multiplication Implementation

To multiply the matrix A and matrix B, the column quantity of A must be equal to the row quantity of B.

Code:

```
>>> A = np.arange(6).reshape(3,2)
>>> B = np.arange(6).reshape(2,3)
>>> A
# Output:
>>> array([[0, 1],
        [2, 3],
        [4, 5]])
>>> B
# Output:
>>> array([[0, 1, 2],
        [3, 4, 5]])
# Matrix multiplication
>>> np.matmul(A,B)
```

```
# Output:
>>> array([[ 3,  4,  5],
           [ 9, 14, 19],
           [15, 24, 33]])
```

2.2.7 Matrix Corresponding Operation

Matrix corresponding operations are operations for tensors of the same shape and size. For example, adding, subtracting, multiplying, and dividing the elements with the same position in two tensors.

```
# Code:
# Matrix creation
>>> A = np.arange(6).reshape(3,2)
# Matrix multiplication
>>> A*A
# Output:
>>> array([[ 0,  1],
           [ 4,  9],
           [16, 25]])
# Matrix addition
>>> A + A
# Output:
>>> array([[ 0,  2],
           [ 4,  6],
           [ 8, 10]])
```

2.2.8 Inverse Matrix Implementation

Inverse matrix implementation is applicable only to square matrices.

```
# Code:
>>> A = np.arange(4).reshape(2,2)
>>> A
# Output:
>>> array([[0, 1],
           [2, 3]])
>>> np.linalg.inv(A)
# Output:
```

```
>>> array([[-1.5,    0.5],
           [ 1. ,    0. ]])
```

2.2.9 Eigenvalue and Eigenvector

Obtain the eigenvalue and eigenvector of a matrix.

Code:

```
>>> import numpy as np # Introduce the numpy module.
>>> x= np.diag(1, 2, 3) # Write the diagonal matrix x.
>>> x                                  #Output the diagonal matrix x.
```

Output:

```
>>> array([[1,0,0],
[0,2,0],
[0,0,3]])
>>> a,b= np.linalg.eig(x)      # Assign the eigenvalue to a, and corresponding eigenvector to b.
>>> a
```

Feature values: 1, 2, 3

Output:

```
>>> array([1.,2.,3.])
>>> b
```

Eigenvector

Output:

```
>>> array([1.,0.,0.],
[0.,1.,0.],
[0.,0.,1.])
```

2.2.10 Determinant

Obtain the determinant of a matrix.

Code:

```
>>> E = array([[1, 2, 3],
           [4, 5, 6],
           [7, 8, 9]])
>>> F = array([[-1,    0,    1],
           [ 2,    3,    4],
           [ 5,    6,    7]])
```

```
# Output:
>>> np.linalg.det(E)
>>> 6.6613381477509402e-16
# Output:
>>> np.linalg.det(F)
>>> 2.664535259100367e-15
```

2.2.11 Singular Value Decomposition

Create a matrix and perform singular value decomposition on the matrix.

```
# Code:
dataMat = [[1,1,1,0,0],
           [2,2,2,0,0],
           [1,1,1,0,0],
           [5,5,5,0,0],
           [1,1,0,2,2]]
>>> dataMat = mat(dataMat)
>>> U,Simga,VT = linalg.svd(dataMat)
>>> U
# Output:
>>> matrix([[ -1.77939726e-01,  -1.64228493e-02,   1.80501685e-02,
            9.53086885e-01,  -3.38915095e-02,   2.14510824e-01,
            1.10470800e-01],
          [ -3.55879451e-01,  -3.28456986e-02,   3.61003369e-02,
           -5.61842993e-02,  -6.73073067e-01,  -4.12278297e-01,
            4.94783103e-01],
          [ -1.77939726e-01,  -1.64228493e-02,   1.80501685e-02,
           -2.74354465e-01,  -5.05587078e-02,   8.25142037e-01,
            4.57226420e-01],
          [ -8.89698628e-01,  -8.21142464e-02,   9.02508423e-02,
           -1.13272764e-01,   2.86119270e-01,  -4.30192532e-02,
           -3.11452685e-01]])
>>> Simga
# Output:
>>>array([   9.72140007e+00,   5.29397912e+00,   6.84226362e-01,
```

```
                1.52344501e-15,      2.17780259e-16])
>>> VT
# Output:
>>> matrix([[ -5.81200877e-01,    -5.81200877e-01,    -5.67421508e-01,
                -3.49564973e-02,    -3.49564973e-02],
            [    4.61260083e-03,      4.61260083e-03,    -9.61674228e-02,
                 7.03814349e-01,      7.03814349e-01],
            [ -4.02721076e-01,    -4.02721076e-01,      8.17792552e-01,
                 5.85098794e-02,      5.85098794e-02],
            [ -7.06575299e-01,      7.06575299e-01,    -2.22044605e-16,
                 2.74107087e-02,    -2.74107087e-02],
            [    2.74107087e-02,    -2.74107087e-02,      2.18575158e-16,
                 7.06575299e-01,    -7.06575299e-01]])
```

2.2.12 Linear Equation Solving

Solving a linear equation is simple because it requires only one function (**scipy.linalg.solve**).

An example is finding the solution of the following system of non-homogeneous linear equations:

3x_1 + x_2 - 2x_3 = 5

x_1 - x_2 + 4x_3 = -2

2x_1 + 3x_3 = 2.5

Code:

```
>>> from scipy.linalg import solve
>>> a = np.array([[3, 1, -2], [1, -1, 4], [2, 0, 3]])
>>> b = np.array([5, -2, 2.5])
>>> x = solve(a, b)
>>> x
# Output:
>>> [0.5 4.5 0.5]
```

3 Probability Theory Experiment

3.1 Introduction

3.1.1 Probability Theory

Probability theory is a mathematical branch of studying the quantitative regularity of random phenomena. A random phenomenon is different from a decisive phenomenon in that a decisive phenomenon inevitably occurs under certain conditions.

The probability theory is a mathematical tool used to describe uncertainty. A large number of AI algorithms build models using the probabilistic or inference information of samples.

3.1.2 Experiment

This section describes the knowledge points of probability and statistics, and mainly uses the **numpy** and **scipy** frameworks.

3.2 Probability Theory Implementation

Import the corresponding modules in advance.

```
>>> import numpy as np
>>> import scipy as sp
```

3.2.1 Average Value Implementation

```
# Prepare the data.
>>> b = [1,3,5,6]
>>> ll = [[1,2,3,4,5,6],[3,4,5,6,7,8]]
# Code:
>>> np.mean(b)
# Output:
>>> 3.75
```

```
>>> np.mean(ll)     # Calculate the average value of all elements.
# Output:
>>> 4.5
>>> np.mean(ll,0)  # Calculate the average value by column.
# Output:
>>> [2. 3. 4. 5. 6. 7.]
>>> np.mean(ll,1)  # Calculate the average value by row.
# Output:
>>> [3.5 5.5]
```

3.2.2 Variance and Standard Deviation Implementation

```
# Prepare the data.
>>> b=[1,3,5,6]
>>> ll=[[1,2,3,4,5,6],[3,4,5,6,7,8]]
# Find the variance.
# Code:
>>> np.var(b)
# Output:
>>> 3.6875
# Code:
>>> np.var(ll[0])
# Output:
>>> 3.6874999999999996
# Code:
>>> np.var(ll,1))     # The second parameter is 1, indicating that the variance is calculated by row.
# Output:
>>> [2.91666667 2.91666667]
```

3.2.3 Standard Deviation Implementation

```
# Prepare the data.
>>> b=[1,3,5,6]
>>> ll=[[1,2,3,4,5,6],[3,4,5,6,7,8]]
# Code:
>>> np.std(b)
```

Output:

>>> 1.920286436967152

Code:

>>> np.std(ll)

Output:

>>> 1.9790570145063195

3.2.4 Covariance Implementation

Prepare the data.

>>> b=[1,3,5,6]

Code:

>>> np.cov(b)

Output:

>>> 4.916666666666666

3.2.5 Correlation Coefficient

Prepare the data.

>>> vc=[1,2,39,0,8]

>>> vb=[1,2,38,0,8]

Complete implementation using a function.

Code:

>>> np.corrcoef(vc,vb)

Output:

>>> 4.916666666666667

Complete implementation through customization.

Code:

>>> np.mean(np.multiply((vc-np.mean(vc)),(vb-np.mean(vb))))/(np.std(vb)*np.std(vc))

Output:

>>> 4.916666666666666

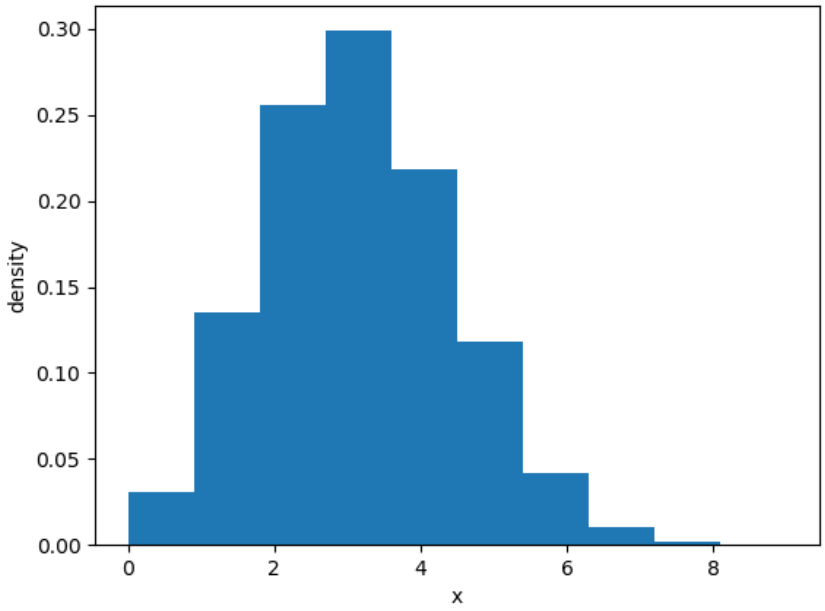
3.2.6 Binomial Distribution Implementation

The random variable *X*, which complies with binomial distribution, indicates the number of successful times (*n*-th and independent with the same distribution in the Bernoulli trial). The success rate of each test is *p*.

Code:

from scipy.stats import binom, norm, beta, expon

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

binom_sim = binom.rvs(n=10, p=0.3, size=10000)
print('Data:',binom_sim)
print('Mean: %g' % np.mean(binom_sim))
print('SD: %g' % np.std(binom_sim, ddof=1))
plt.hist(binom_sim, bins=10, normed=True)
plt.xlabel(('x'))
plt.ylabel('density')
plt.show()
# Output:
Data: [2 4 3 ... 3 4 1]
Mean: 2.9821
SD: 1.43478
# The distribution figure is as follows.
```



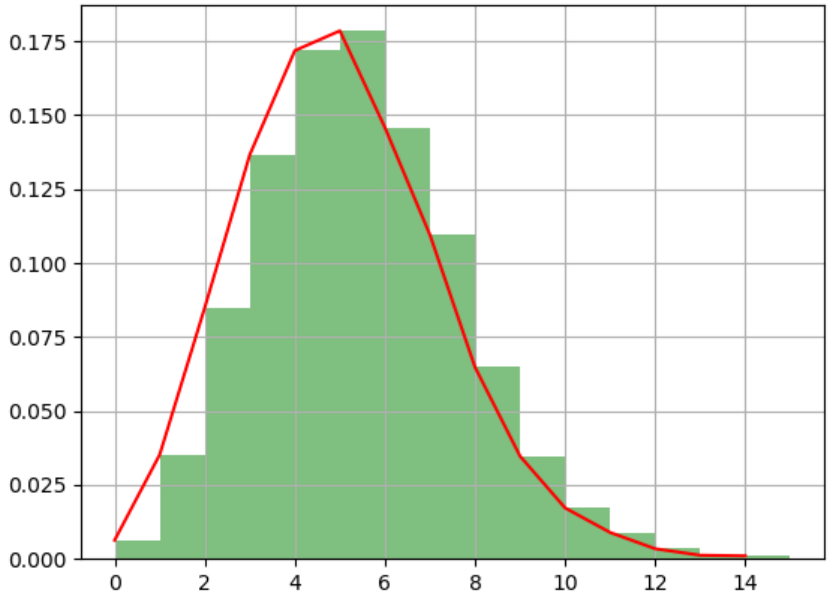
3.2.7 Poisson Distribution Implementation

A random variable *X* that obeys the Poisson distribution indicates the number of occurrences of an event within a fixed time interval that meets the **λ** parameter. The parameter **λ** indicates the rate of an event. Both the average value and variance of the random variable X are **λ**.

Code:

```
import numpy as np

import matplotlib.pyplot as plt


x = np.random.poisson(lam=5, size=10000)

pillar = 15


a = plt.hist(x, bins=pillar, normed=True, range=[0, pillar], color='g', alpha=0.5)

plt.plot(a[1][0:pillar], a[0], 'r')

plt.grid()

plt.show()
```

The distribution figure is as follows.



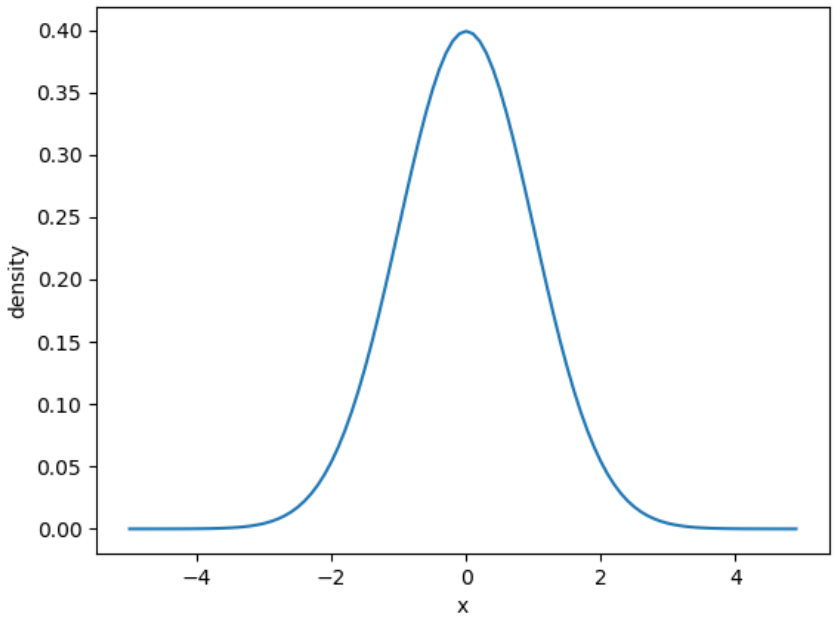
NOTE

The preceding figure can be displayed only in a visual system. Trainees can perform the test in the local environment.

3.2.8 Normal Distribution

Normal distribution is continuous. Its function can obtain a value anywhere on the real line. Normal distribution is described by two parameters: average value **μ** and standard deviation σ of distribution.

```
# Code:

from scipy.stats import norm

import numpy as np

import matplotlib.pyplot as plt


mu = 0

sigma = 1

x = np.arange(-5, 5, 0.1)

y = norm.pdf(x, mu, sigma)

plt.plot(x, y)

plt.xlabel('x')

plt.ylabel('density')

plt.show()

# The distribution figure is as follows.
```



NOTE

The preceding figure can be displayed only in a visual system. Trainees can perform the test in the local environment.

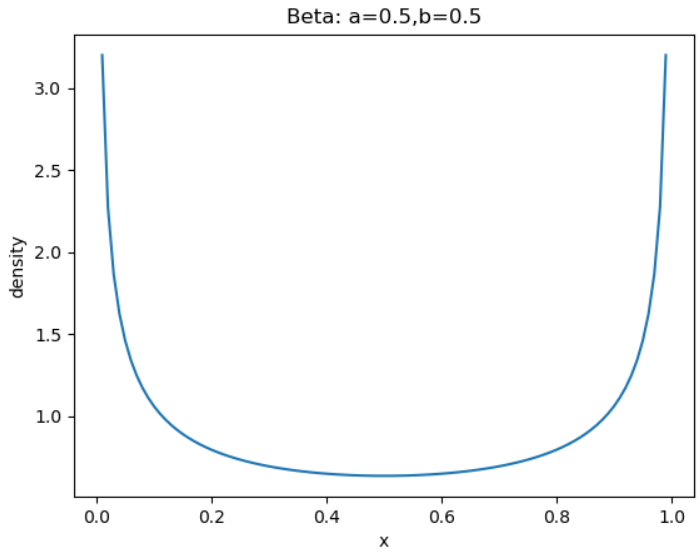
3.2.9 beta Distribution

The beta distribution, continuous distribution with values between **0** and **1**, is described by the values of two morphological parameters **α** and **β**.

Code:

```
from scipy.stats import beta

import numpy as np

import matplotlib.pyplot as plt


a = 0.5

b = 0.5

x = np.arange(0.01, 1, 0.01)

y = beta.pdf(x, a, b)

plt.plot(x, y)

plt.title('Beta: a=%.1f,b=%.1f' % (a, b))

plt.xlabel('x')

plt.ylabel('density')

plt.show()
```

The distribution figure is as follows.



NOTE

The preceding figure can be displayed only in a visual system. Trainees can perform the test in the local environment.

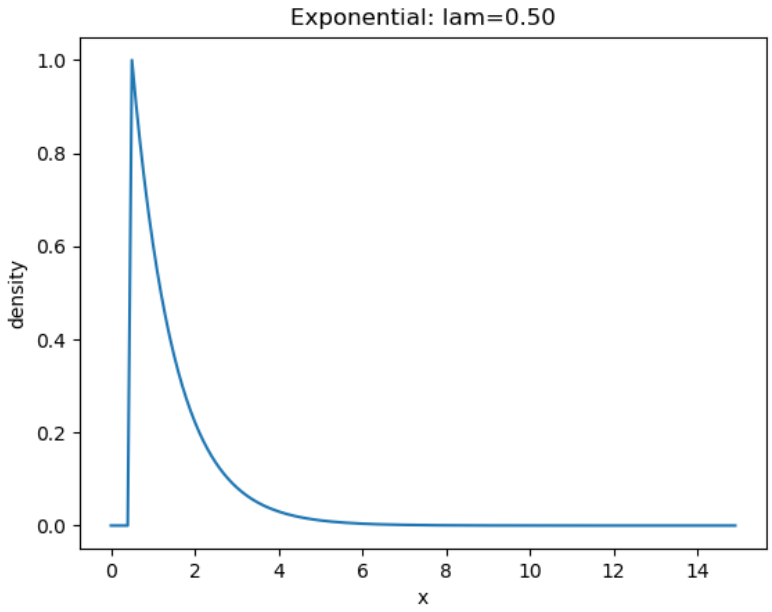
3.2.10 Exponential Distribution

Exponential distribution is continuous probability distribution, indicating the time interval of independent random events (for example, time interval for entering the airport by passengers or for calling the customer service center).

Code:

```
from scipy.stats import expon

import numpy as np

import matplotlib.pyplot as plt


lam = 0.5

x = np.arange(0, 15, 0.1)

y = expon.pdf(x, lam)

plt.plot(x, y)

plt.title('Exponential: lam=%.2f' % lam)

plt.xlabel('x')

plt.ylabel('density')

plt.show()
```

The distribution figure is as follows.



NOTE

The preceding figure can be displayed only in a visual system. Trainees can perform the test in the local environment.

4 Other Experiments

4.1 Least Square Implementation

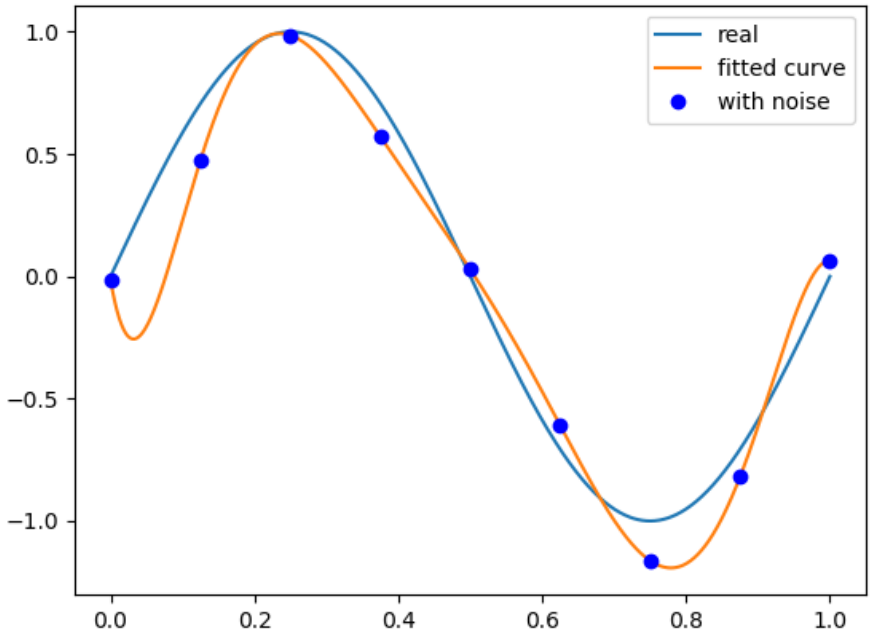
4.1.1 Algorithm

The least square method, as the basis of the classification regression algorithm, has a long history. It searches for the best function of the data through the least sum of square error. The least square method can help easily obtain unknown parameters, and ensure that the square sum of the error between the predicted data and the actual data is the smallest.

4.1.2 Code Implementation

```
# Code:

import numpy as np # Introduce the numpy.

import scipy as sp

import pylab as pl

from scipy.optimize import leastsq # Introduce the least square function.


n = 9 # Degree of polynomial


# Target function

def real_func(x):

    return np.sin(2 * np.pi * x)


# Polynomial function

def fit_func(p, x):

    f = np.poly1d(p)

    return f(x)
```

```
# Residual function
def residuals_func(p, y, x):
    ret = fit_func(p, x) - y
    return ret


x = np.linspace(0, 1, 9) # Randomly selected nine points as x.
x_points = np.linspace(0, 1, 1000) # Continuous points required for drawing the graph


y0 = real_func(x) # Target function
y1 = [np.random.normal(0, 0.1) + y for y in y0] # Function after the positive distribution noise is added
p_init = np.random.randn(n) # Randomly initialize polynomial parameters.
plsq = leastsq(residuals_func, p_init, args=(y1, x))


print('Fitting Parameters: ', plsq[0]) # Output fitting parameters.


pl.plot(x_points, real_func(x_points), label='real')
pl.plot(x_points, fit_func(plsq[0], x_points), label='fitted curve')
pl.plot(x, y1, 'bo', label='with noise')
pl.legend()
pl.show()
# Output:
Fitting Parameters:    [-1.22007936e+03    5.79215138e+03 -1.10709926e+04
1.08840736e+04
  -5.81549888e+03    1.65346694e+03 -2.42724147e+02    1.96199338e+01
  -2.14013567e-02]
# Visualized chart
```



NOTE

The preceding figure can be displayed only in a visual system. Trainees can perform the test in the local environment.

4.2 Gradient Descent Implementation

4.2.1 Algorithm

Gradient descent (steepest descent) is the most common method for solving unconstrained optimization problems. It is an iterative method. Each step mainly finds the gradient vector of the target function and uses the negative gradient direction of the current position as the search direction (the target function decreases the fastest in this direction, which is the reason why it is called steepest descent method).
The gradient descent method has the following characteristics: The closer the function to the target value, the smaller is the step and the slower is the decrease rate.

4.2.2 Code Implementation

Code:

Training set

Each sample point has three components (*x0*, *x1*, and *x2*).

x = [(1, 0., 3), (1, 1., 3), (1, 2., 3), (1, 3., 2), (1, 4., 4)]

y[i] (Output of the sample point)

```
y = [95.364, 97.217205, 75.195834, 60.105519, 49.342380]

# Iteration threshold (When the difference between the two iteration loss functions is less than the threshold, the iteration stops.)

epsilon = 0.0001

# Learning rate

alpha = 0.01

diff = [0, 0]

max_itor = 1000

error1 = 0

error0 = 0

cnt = 0

m = len(x)


# Parameter initialization

theta0 = 0

theta1 = 0

theta2 = 0


while True:

    cnt += 1


    # Parameter iteration calculation

    for i in range(m):

        # The fitting function is y = theta0 * x[0] + theta1 * x[1] +theta2 * x[2].

        # Residual error calculation

        diff[0] = (theta0 + theta1 * x[i][1] + theta2 * x[i][2]) - y[i]


        # Gradient = diff[0] * x[i][j]

        theta0 -= alpha * diff[0] * x[i][0]

        theta1 -= alpha * diff[0] * x[i][1]

        theta2 -= alpha * diff[0] * x[i][2]


    # Loss function calculation

    error1 = 0
```

```
        for lp in range(len(x)):
            error1 += (y[lp]-(theta0 + theta1 * x[lp][1] + theta2 * x[lp][2]))**2/2

        if abs(error1-error0) < epsilon:
            break
        else:
            error0 = error1

        print(' theta0: %f, theta1: %f, theta2: %f, error1: %f' % (theta0, theta1, theta2, error1) )

print('Done: theta0 : %f, theta1 : %f, theta2 : %f' % (theta0, theta1, theta2)    )
print ('Number of iterations: %d' % cnt    )
# Output:
theta0 : 2.782632, theta1 : 3.207850, theta2 : 7.998823, error1 : 5997.941160
theta0 : 4.254302, theta1 : 3.809652, theta2 : 11.972218, error1 : 3688.116951
theta0 : 5.154766, theta1 : 3.351648, theta2 : 14.188535, error1 : 2889.123934
theta0 : 5.800348, theta1 : 2.489862, theta2 : 15.617995, error1 : 2490.307286
theta0 : 6.326710, theta1 : 1.500854, theta2 : 16.676947, error1 : 2228.380594
theta0 : 6.792409, theta1 : 0.499552, theta2 : 17.545335, error1 : 2028.776801
… …
theta0 : 97.717864, theta1 : -13.224347, theta2 : 1.342491, error1 : 58.732358
theta0 : 97.718558, theta1 : -13.224339, theta2 : 1.342271, error1 : 58.732258
theta0 : 97.719251, theta1 : -13.224330, theta2 : 1.342051, error1 : 58.732157
Done: theta0 : 97.719942, theta1 : -13.224322, theta2 : 1.341832
Number of iterations: 2608
```