Huawei HCIA-AI Series Training

HCIA-AI V3.0
AI Mathematics
Experiment Guide

Issue: 1.0

HUAWEI TECHNOLOGIES CO., LTD.

Trademarks and Permissions

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: https://www.huawei.com/

Email: support@huawei.com

Introduction to Huawei Certification System

Based on cutting-edge technologies and professional training systems, Huawei certification meets the diverse AI technology demands of clients. Huawei is committed to providing practical and professional technical certification for our clients.

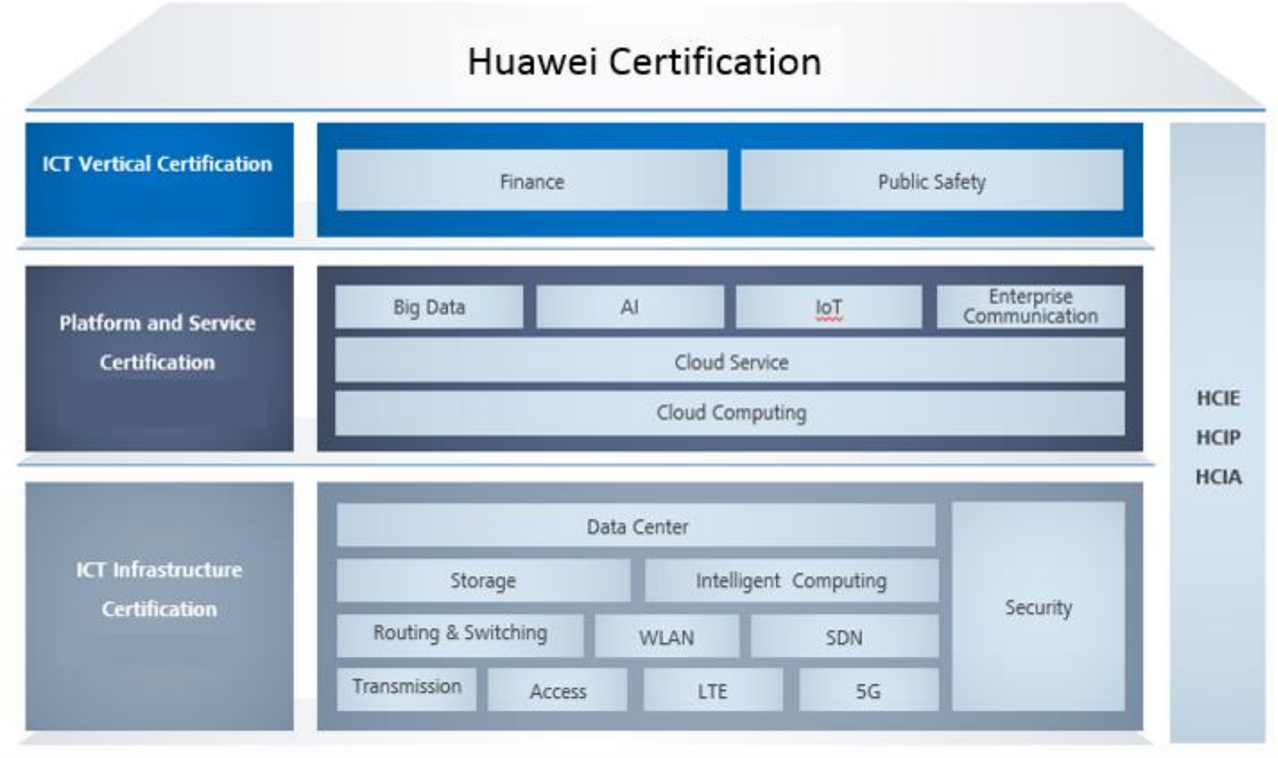
HCIA-AI V1.0 certification is intended to popularize AI and help understand deep learning and Huawei Cloud EI, and learn the basic capabilities of programming based on the TensorFlow framework, as a motive to promote talent training in the AI industry.

Content of HCIA-AI V1.0 includes but is not limited to: AI overview, Python programming and experiments, mathematics basics and experiments, TensorFlow introduction and experiments, deep learning pre-knowledge, deep learning overview, Huawei cloud EI overview, and application experiments for image recognition, voice recognition and man-machine dialogue.

HCIA-AI certification will prove that you systematically understand and grasp Python programming, essential mathematics knowledge in AI, basic programming methods of machine learning and deep learning platform TensorFlow, pre-knowledge and overview of deep learning, overview of Huawei cloud EI, basic programming for image recognition, voice recognition, and man-machine dialogue. With this certification, you have required knowledge and techniques for AI pre-sales basic support, AI after-sales technical support, AI products sales, AI project management, and are qualified for positions such as natural language processing (NLP) engineers, image processing engineers, voice processing engineers and machine learning algorithm engineers.

Enterprises with HCIA-AI-certified engineers have the basic understanding of AI technology, framework, and programming, and capable of leveraging AI, machine learning, and deep learning technologies, as well as the open-source TensorFlow framework to design and develop AI products and solutions like machine learning, image recognition, voice recognition, and man-machine dialogue.

Huawei certification will help you open the industry window and the door to changes, standing in the forefront of the AI world!


Huawei Certification
ICT Vertical Certification
Finance
Public Safety
Platform and Service Certification
Big Data
AI
IoT
Enterprise Communication
Cloud Service
Cloud Computing
ICT Infrastructure Certification
Data Center
Storage
Intelligent Computing
Routing & Switching
WLAN
SDN
Transmission
Access
LTE
5G
Security
HCIE
HCIP
HCIA

Contents

1 Experiment Overview

1.1 Experiment Introduction

This course will introduce the implementation of basic mathematics experiments based on Python, including basic operators, linear algebra, probability and statistics, and optimization. Upon completion of this course, you will be able to master the implementation of basic mathematical methods based on Python and apply them to actual projects to solve business problems.

After completing this experiment, you will be able to:

- Master how to use Python to implement basic operators.
- Master how to use Python to implement calculation related to linear algebra, probability and statistics.
- Master the implementation of Python optimization.

1.2 Description

This document describes five experiments:

- Experiment 1: Basic mathematics experiment
- Experiment 2: Linear algebra experiment
- Experiment 3: Probability and statistics experiment
- Experiment 4: Optimization experiment

1.3 Skill Requirements

This course is a basic mathematics course based on Python. Before starting this experiment, you are expected to master knowledge about the basic linear algebra, probability and statistics, and optimization.

1.4 Experiment Environment Overview

- Use a PC running the Windows 7/Windows 10 64-bit operating system. The PC must be able to access the Internet.
- Download and install Anaconda 3 4.4.0 or a later version based on the operating system version.

2 Basic Mathematics Experiment

2.1 Introduction

2.1.1 Content

The basic mathematics knowledge is widely used in data mining, especially in algorithm design and numerical processing. The main purpose of this section is to implement some basic mathematical algorithms based on the Python language and basic mathematics modules, laying a foundation for learning data mining.

2.1.2 Frameworks

This document mainly uses the math library, NumPy library, and SciPy library. The math library is a standard library of Python and provides some common mathematical functions. The NumPy library is an extended library of Python, used for numerical calculation. It can solve problems about linear algebra, random number generation, and Fourier transform. The SciPy library is used to handle problems related to statistics, optimization, interpolation, and integration.

2.2 Implementation

Import libraries:

```
import math
import numpy as np
```

2.2.1 ceil Implementation

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

Input:

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

Output:

5

Input:

```
math.ceil(4.99)
```

Output:

```
5
```

2.2.2 floor Implementation

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

Input:

```
math.floor(4.1)
```

Output:

```
4
```

Input:

```
math.floor(4.999)
```

Output:

```
4
```

2.2.3 degrees Implementation

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

Input:

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

Output:

```
45.0
```

Input:

```
math.degrees(math.pi)
```

Output:

```
180.0
```

2.2.4 exp Implementation

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

Input:

```
math.exp(1)
```

Output:

```
2.718281828459045
```

2.2.5 fabs Implementation

The fabs(x) function returns the absolute value of x.

Input:

```
math.fabs(-0.003)
```

Output:

```
0.003
```

2.2.6 factorial Implementation

The factorial(x) function returns the factorial of x.

Input:

```
math.factorial(3)
```

Output:

```
6
```

2.2.7 fsum Implementation

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

Input:

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

Output:

```
10
```

2.2.8 fmod Implementation

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

Input:

```
math.fmod(20,3)
```

Output:

```
2.0
```

2.2.9 log Implementation

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

Input:

```
math.log(10)
```

Output:

```
2.302585092994046
```

2.2.10 sqrt Implementation

The sqrt(x) function returns the square root of x.

Input:

```
math.sqrt(100)
```

Output:

```
10.0
```

2.2.11 pi Implementation

pi is a numerical constant, indicating the circular constant.

Input:

```
math.pi
```

Output:

```
3.141592653589793
```

2.2.12 pow Implementation

The pow(x, y) function returns the x to the power of y, that is, x^y

Input:

```
math.pow(3,4)
```

Output:

```
81.0
```

3 Linear Algebra Experiment

3.1 Introduction

3.1.1 Linear Algebra

Linear algebra is a discipline widely used in various engineering fields. The concepts and conclusions of linear algebra can greatly simplify the derivations and expressions of data mining formulas. Linear algebra can simplify complex problems so that we can perform efficient mathematical operations.

Linear algebra is a mathematical tool. It not only provides the technology for array operations, but also provides data structures such as vectors and matrices to store numbers and rules for addition, subtraction, multiplication, and division.

3.1.2 Code Implementation

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 NumPy. The mathematical science library SciPy is also used to illustrate equation solution in this section.

3.2 Linear Algebra Implementation

Import libraries:

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

3.2.1 Reshape Operation

There is no reshape operation in mathematics, but it is a very common operation in the NumPy operation library. 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 inputting the image, it can be transformed from 10x10 to 1x100 through the reshape operation. The following is an example:

Input:

Generate a vector that contains integers from 0 to 11.

```
x = np.arange(12)
print(x)
```

Output:

```
[ 0  1  2  3  4  5  6  7  8  9 10 11]
```

View the array size.

```
x.shape
```

Output:

```
(12,)
```

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

```
x = x.reshape(1,12)
print(x)
```

Output:

```
[[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11]]
```

View the array size.

```
x.shape
```

Output:

```
(1, 12)
```

Convert x to a 3x4 matrix.

```
x = x.reshape(3,4)
print(x)
```

Output:


```
[[ 0,  1,  2,  3],
 [ 4,  5,  6,  7],
 [ 8,  9, 10, 11]]
```

3.2.2 Transpose Implementation

The transpose of vectors and matrices is to switch the row and column indices. For the transpose of tensors of three dimensions and above, you need to specify the transpose dimension.

Input:

Generate a 3x4 matrix and transpose the matrix.

```
A = np.arange(12).reshape(3,4)
print(A)
```

Output:

```
[[ 0,  1,  2,  3],
 [ 4,  5,  6,  7],
 [ 8,  9, 10, 11]]
```

Input:

```
A.T
```

Output:

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

3.2.3 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.

Input:

```
A = np.arange(6).reshape(3,2)
B = np.arange(6).reshape(2,3)
print(A)
```

Output:

```
[[0 1]
 [2 3]
```

```
 [4 5]]
```

Input:

```
print(B)
```

Output:

```
[[0, 1, 2],
 [3, 4, 5]]
```

Matrix multiplication:

```
np.matmul(A,B)
```

Output:

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

3.2.4 Matrix Operations

Element operations are operations on matrices of the same shape. For example, element operations include the addition, subtraction, division, and multiplication operations on elements with the same position in two matrices.

Input:

Create matrix A:

```
A = np.arange(6).reshape(3,2)
```

Matrix multiplication:

```
print(A*A)
```

Output:

```
array([[ 0,  1],
       [ 4,  9],
       [16, 25]])
```

Matrix addition:

```
print(A + A)
```

Output:

```
array([[ 0,  2],
       [ 4,  6],
```

```
       [ 8, 10]])
```

3.2.5 Inverse Matrix Implementation

Inverse matrix implementation is applicable only to square matrices.

Input:

```
A = np.arange(4).reshape(2,2)
print(A)
```

Output:

```
array([[0, 1],
       [2, 3]])
```

Inverse matrix:

```
np.linalg.inv(A)
```

Output:

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

3.2.6 Eigenvalue and Eigenvector

This section describes how to obtain the matrix eigenvalues and eigenvectors and implement visualization.

Input:

#Import libraries:

```
from scipy.linalg import eig
import numpy as np
import matplotlib.pyplot as plt
```

#Obtain the eigenvalue and eigenvector:

```
A = [[1, 2],# Generate a 2x2 matrix.
     [2, 1]]
evals, evecs = eig(A) # Calculate the eigenvalue (evals) and eigenvector (evecs) of A.
evecs = evecs[:, 0], evecs[:, 1]
```

#The plt.subplots() function returns a figure instance named **fig** and an AxesSubplot instance named **ax**. The **fig** parameter indicates the entire figure, and **ax** indicates the coordinate axis. Plotting:

```
fig, ax = plt.subplots()
```

#Make the coordinate axis pass the origin.

```
for spine in ['left', 'bottom']:# Make the coordinate axis in the lower left corner pass the origin.
    ax.spines[spine].set_position('zero')
```

#Draw a grid:

```
ax.grid(alpha=0.4)
```

#Set the coordinate axis ranges.

```
xmin, xmax = -3, 3
ymin, ymax = -3, 3
ax.set(xlim=(xmin, xmax), ylim=(ymin, ymax))
```

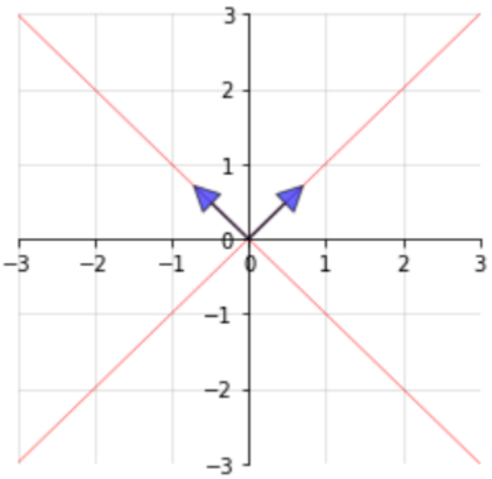
#Draw an eigenvector. Annotation is to use an arrow that points to the content to be explained and add a description. In the following code, **s** indicates the input, **xy** indicates the arrow direction, **xytext** indicates the text location, and **arrowprops** uses **arrowstyle** to indicate the arrow style or type.

```
for v in evecs:
    ax.annotate(s="", xy=v, xytext=(0, 0),
                arrowprops=dict(facecolor='blue',
                shrink=0,
                alpha=0.6,
                width=0.5))
```

#Draw the eigenspace:

```
x = np.linspace(xmin, xmax, 3)# Return evenly spaced numbers over a specified interval.
for v in evecs:
    a = v[1] / v[0] # Unit vector in the eigenvector direction.
    ax.plot(x, a * x, 'r-', lw=0.4)# The lw parameter indicates the line thickness.
plt.show()
```

Figure 3-1 Visualized chart



Interpretation: The vectors with the blue arrow are eigenvectors, and the space formed by the two red lines is the eigenspace.

3.2.7 Determinant

This section describes how to obtain the determinant of a matrix.

Input:

```
E = [[1, 2, 3],
     [4, 5, 6],
     [7, 8, 9]]
print(np.linalg.det(E))
```

Output:

```
0.0
```

3.2.8 Application Scenario of Singular Value Decomposition: Image Compression

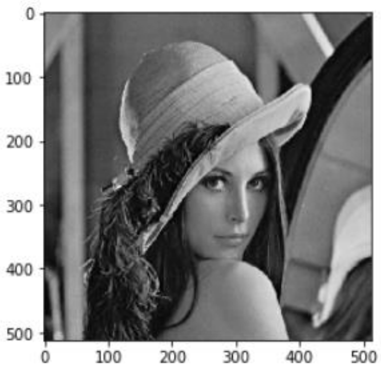
A grayscale image can be regarded as a matrix. If singular value decomposition is performed on such a matrix, singular values of the singular value matrix are arranged in descending order. A singular vector with a larger singular value can save more information, but the singular values usually attenuate quickly. Therefore, the first K singular values and corresponding singular vectors include most information in the image. As a result, an image formed by the first K singular values and their singular vectors can achieve basically the same definition as the original image, but the data amount is greatly reduced. In this way, image data compression can be implemented.

Input:

```
import numpy as np
from pylab import *
import matplotlib.pyplot as plt

# Read and save the grayscale image.
img = imread('lena.jpg')[:]
plt.savefig('./lena_gray')
plt.gray()
# Draw a grayscale image.
plt.figure(1)
plt.imshow(img)
```

Output:




```
# Read and print the image length and width.
m,n = img.shape
print(np.shape(img))
```

Output:

```
(512, 512)
```

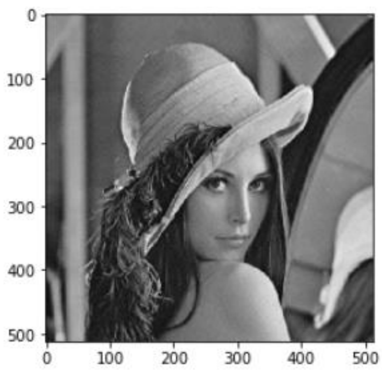
```
# Perform singular value decomposition on the image matrix.
U,sigma,V = np.linalg.svd(img)
# Print the singular value shape.
print(np.shape(sigma))
```

Output:

```
(512,)
```

```
# Arrange singular values into a diagonal matrix.
sigma = resize(sigma, [m,1])*eye(m,n)
# Use the first K singular values and their singular vectors for image compression.
k= 10
# Create an image with the first K singular values and their singular vectors.
img1 = np.dot(U[:,0:k],np.dot(sigma[0:k,0:k],V[0:k,:]))
plt.figure(2)
# Print the compressed image.
plt.imshow(img1)
plt.show()
```

Output:



4 Probability and statistics Experiment

4.1 Introduction

4.1.1 Probability and statistics

Probability and statistics is a branch of mathematics concerned with the quantitative regularity of random phenomena. A random phenomenon is a situation in which we know what outcomes could happen, but we do not know which particular outcome did or will happen, while a decisive phenomenon is a situation in which a result inevitably occurs under certain conditions.

Probability and statistics is a mathematical tool used to describe uncertainties. A large number of data mining algorithms build models based on the sample probabilistic information or through inference.

4.1.2 Experiment Overview

This section describes the knowledge of probability and statistics, and mainly uses the NumPy and SciPy frameworks.

4.2 Probability and statistics Implementation

Import libraries:

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

4.2.1 Mean Value Implementation

Input:

#Data preparation:

```
ll = [[1,2,3,4,5,6],[3,4,5,6,7,8]]
np.mean(ll)      # Calculate the mean value of all elements.
```

Output:

```
4.5
```

Input:

```
np.mean(ll,0) # Calculate the mean value by column. The value 0 indicates the column vector.
```

Output:

```
array([2., 3., 4., 5., 6., 7.])
```

Input:

```
np.mean(ll,1)  # Calculate the mean value by row. The value 1 indicates the row vector.
```

Output:

```
array([3.5, 5.5])
```

4.2.2 Variance Implementation

#Data preparation:

```
b=[1,3,5,6]
ll=[[1,2,3,4,5,6],[3,4,5,6,7,8]]
```

#Calculate the variance:

```
np.var(b)
```

Output:

```
3.6875
```

Input:

```
np.var(ll,1)      # The value of the second parameter is 1, indicating that the variance is calculated by row.
```

Output:

```
[2.91666667 2.91666667]
```

4.2.3 Standard Deviation Implementation

Input:

#Data preparation:

```
ll=[[1,2,3,4,5,6],[3,4,5,6,7,8]]
np.std(ll)
```

Output:

```
1.9790570145063195
```

4.2.4 Covariance Implementation

Input:

#Data preparation:

```
x = np.array([[1, 2], [3, 7]])
print(np.cov(x))
```

Output:

```
 [[0.5 2. ]
 [2.   8. ]]
```

4.2.5 Correlation Coefficient

Input:

#Data preparation:

```
vc=[1,2,39,0,8]
vb=[1,2,38,0,8]
```

#Function-based implementation:

```
np.corrcoef(vc,vb)
```

Output:

```
array([[1.        , 0.99998623],
       [0.99998623, 1.        ]])
```

4.2.6 Binomial Distribution Implementation

The random variable X, which complies with binomial distribution, indicates the number of successful times in n times of independent and identically distributed Bernoulli experiments. The success probability of each experiment is p.

Input:

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

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

# The n and p parameters indicate the success times and probability in the binomial formula, respectively, and size indicates the number of sampling times.
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))
# Generate a histogram. The bins parameter indicates the number of bars in total. By default, the sum of the percentages of all bars is 1.
plt.hist(binom_sim, bins=10)
plt.xlabel(('x'))
plt.ylabel('density')
plt.show()
```

Output:

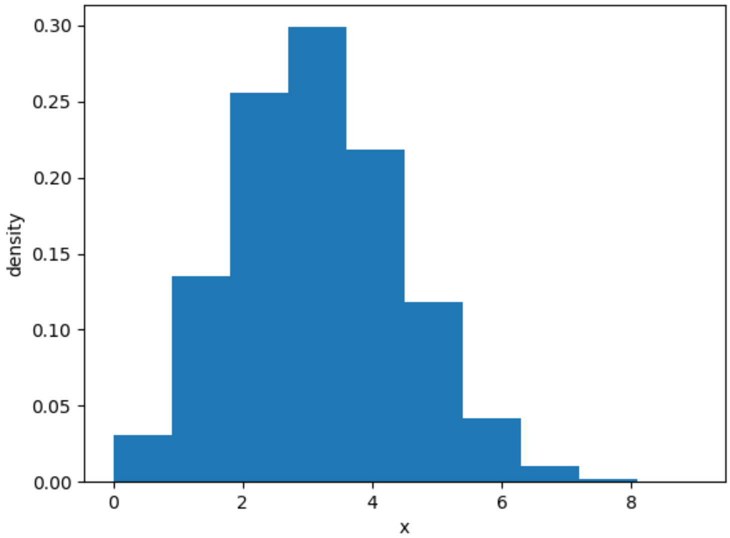
Data: [2 4 3 ... 3 4 1]# 10,000 numbers that comply with the binomial distribution.

Mean: 2.9821

SD: 1.43478

The following figure shows the binomial distribution.

Figure 4-1



4.2.7 Poisson Distribution Implementation

A random variable X that complies with the Poisson distribution indicates the number of occurrences of an event within a fixed time interval with the **λ** parameter. Both the mean value and variance of the random variable X are λ.

Input:

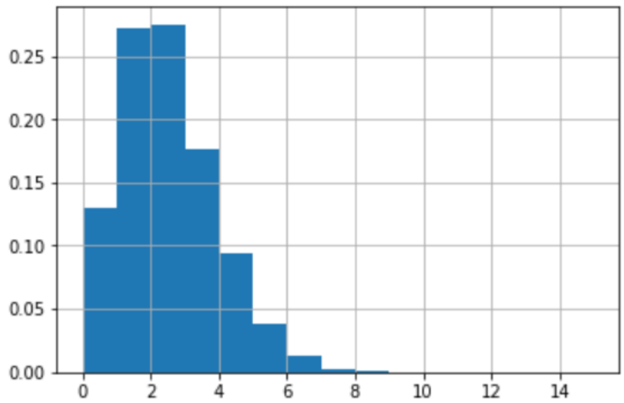
```
import numpy as np
import matplotlib.pyplot as plt

# Generate 10,000 numbers that comply with the Poisson distribution where the value of lambda is 2.
X= np.random.poisson(lam=2, size=10000)

a = plt.hist(X, bins=15, range=[0, 15])
# Generate grids.
plt.grid()
plt.show()
```

The following figure shows the Poisson distribution.

Figure 4-2



4.2.8 Normal Distribution

Normal distribution is a continuous probability distribution. Its function can obtain a value anywhere on the curve. Normal distribution is described by two parameters: **μ** and **σ**, which indicate the mean value and standard deviation, respectively.

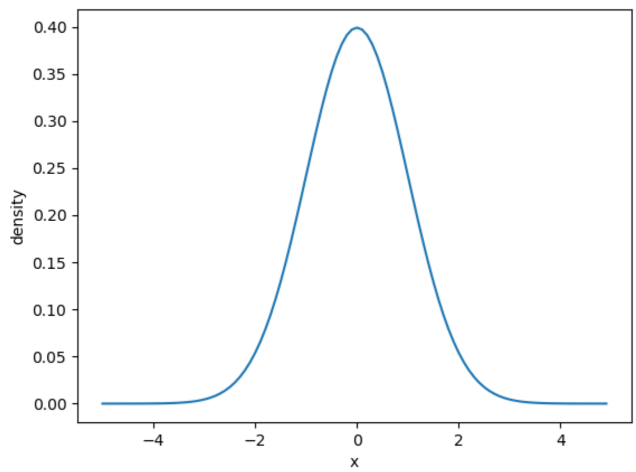
Input:

```
from scipy.stats import norm
import numpy as np
import matplotlib.pyplot as plt

mu = 0
sigma = 1
#Return evenly spaced values within a given interval (start, stop, step).
x = np.arange(-5, 5, 0.1)
# Generate normal distribution that complies with mu and sigma.
y = norm.pdf(x, mu, sigma)
plt.plot(x, y)
plt.xlabel('x')
plt.ylabel('density')
plt.show()
```

The following figure shows the distribution.

Figure 4-3



5 Optimization Experiment

5.1 Gradient Descent Implementation

5.1.1 Algorithm

Gradient descent is a first-order iterative optimization algorithm for finding the minimum of a function. The operation of each step is to solve the gradient vectors of the target function. The gradient direction negative to the current position is used as the search direction (as the target function descends the most quickly in this direction, the gradient descent method is also called the steepest descent method).

The gradient descent method has the following characteristics: If the function is closer to the target value, the step is smaller, and the descent speed is slower.

5.1.2 Case Introduction

Find the local minima of the function $\mathrm{y} = (\mathrm{x} - 6)^2$ starting from the point x=1. It is easy to find the answers by calculating $\mathrm{y} = (\mathrm{x} - 6)^2 = 0, \mathrm{x} = 6$. Thus $\mathrm{x} = 6$ is the local minima of the function.

The Pseudo code is as follows:

Initialize parameters: learning rate=0.01

$$\frac{\mathrm{dy}}{\mathrm{dx}} = \frac{\mathrm{d}(x-6)^2}{\mathrm{dx}} = 2 \times (x-6)$$

$$x_0 = 1$$

Iteration 1:

$$x_1 = x_0 - (learning\ rate) \times \frac{dy}{dx}$$

$$x_1 = 1 - 0.01 \times 2 \times (1 - 6) = 1.1$$

Iteration 2:

$$x_2 = x_1 - (learning\ rate) \times \frac{dy}{dx}$$

$$x_2 = 1.1 - 0.01 \times 2 \times (1.1 - 6) = 1.198$$

5.1.3 Code Implementation

Input:

```
cur_x = 1 # The algorithm starts at x=1
rate = 0.01 # Learning rate
precision = 0.000001 #This tells us when to stop the algorithm
previous_step_size = 1 #
max_iters = 10000 # maximum number of iterations
iters = 0 #iteration counter
df = lambda x: 2*(x-6) #Gradient of our function

while previous_step_size > precision and iters < max_iters:
    prev_x = cur_x #Store current x value in prev_x
    cur_x = cur_x - rate * df(prev_x) #Grad descent
    previous_step_size = abs(cur_x - prev_x) #Change in x
    iters = iters+1 #iteration count
    print("Iteration",iters,"\nX value is",cur_x) #Print iterations

print("The local minimum occurs at", cur_x)
```

Output:

```
Iteration 1
X value is 1.1
Iteration 2
X value is 1.1980000000000002
Iteration 3
X value is 1.29404
Iteration 4
X value is 1.3881592
Iteration 5
X value is 1.480396016
```

Iteration 6

X value is 1.57078809568

...

...

Iteration 570

X value is 5.99995013071414

Iteration 571

X value is 5.999951128099857

The local minimum occurs at 5.999951128099857