Huawei HCIA-AI Series Training

HCIA-AI V1.0
AI Mathematics
Experiment Guide

Issue: 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: 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.

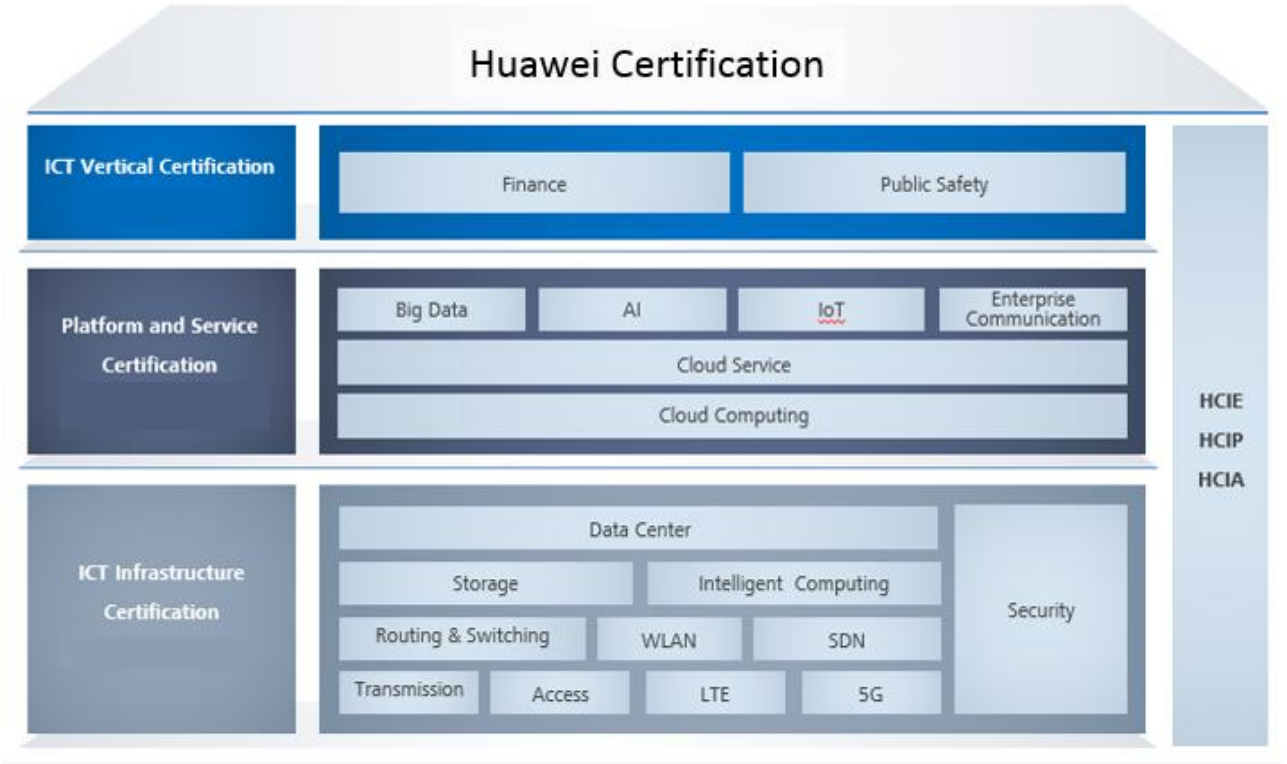
HCIA-AI V1.0 certification is intended to popularize AI and help understand deep learning and Huawei Cloud EI, and learn the basic capabilities of programming based on the TensorFlow framework, as a motive to promote talent training in the AI industry.

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!



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 theory, 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 and probability theory.
- Master the implementation of Python optimization.

1.2 Description

This document describes five experiments:

- Experiment 1: basic mathematics experiment
- Experiment 2: implementation of operators related to linear algebra
- Experiment 3: implementation of statistical distribution of the probability theory
- Experiment 4: implementation of the least squares method
- Experiment 5: implementation of gradient descent

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 theory, 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 the image is read, it can

be transformed from 1x100 to 10x10 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 indicate the addition, subtraction, division, and multiplication operations on elements with the same position in two matrices.

Input:

Matrix creation:

```
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

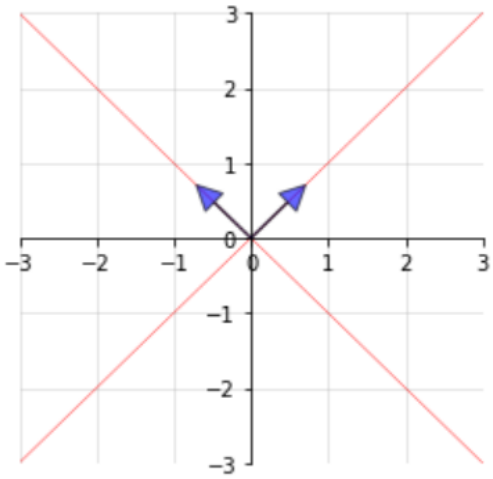


Illustration: The vectors with the blue arrow are eigenvectors, and the space formed by the two red lines is the eigenspace.

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 Singular Value Decomposition

Case introduction:

If we are a news provider, we need to push a large number of articles from different fields to users every day. However, the quality of these articles varies. To improve user experience, we need to classify a lot of articles and select better articles for

users. So the question is, what method can be used to automatically classify the articles? The following describes a simple approach. Generally, the title of an article consists of a few core concept words, which represent the subject of the article. Therefore, we can determine whether two articles belong to the same type based on the similarity between titles. The similarity between titles depends on the similarity between words. Then how do we check the similarity between words in the titles? We can use singular value decomposition to solve this problem. Assume that there are eight titles. The keywords of the titles are as follows:

```
title_1 = ["dad","dad","stock"]
title_2 = ["books","books","value","estate"]
title_3 = ["books","decomposition"]
title_4 = ["stock"]
title_5 = ["dad"]
title_6 = ["value","singular","decomposition"]
title_7 = ["dad","singular"]
title_8 = ["singular","estate","decomposition"]
```

Input:

#Import modules:

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

#Enter keywords:

```
words = ["books","dad","stock","value","singular","estate","decomposition"]
```

#Assume that there are eight titles and seven keywords. Record the number of times each keyword appears in each title to obtain matrix X. In matrix X, each row indicates a title, each column indicates a keyword, and each element in the matrix indicates the number of times a keyword appears in a title.

```
X=np.array([[0,2,1,0,0,0,0],[2,0,0,1,0,1,0],[1,0,0,0,0,0,1],[0,0,1,0,0,0,0],[0,1,0,0,0,0,0],[0,0,0,1,1,0,1],
[0,1,0,0,1,0,0],[0,0,0,0,1,1,1]])
```

#Singular value decomposition:

```
U,s,Vh=np.linalg.svd(X)
```

#Output the left singular matrix U and its shape:

```
print("U=",U)
print("U.shape",U.shape)
```

Output:

```
U= [[-1.87135757e-01 -7.93624528e-01  2.45011855e-01 -2.05404352e-01
  -3.88578059e-16  5.75779114e-16 -2.57394431e-01 -4.08248290e-01]
 [-6.92896814e-01  2.88368077e-01  5.67788037e-01  2.22142537e-01
   2.54000254e-01 -6.37019839e-16 -2.21623012e-02  2.05865892e-17]
 [-3.53233681e-01  1.22606651e-01  3.49203461e-02 -4.51735990e-01
  -7.62000762e-01  1.27403968e-15  2.72513448e-01  3.80488702e-17]
```



```
 [-2.61369658e-02 -1.33189110e-01   7.51079037e-02 -6.44727454e-01
    5.08000508e-01   1.77635684e-15   3.68146235e-01   4.08248290e-01]
 [-8.04993957e-02 -3.30217709e-01   8.49519758e-02   2.19661551e-01
   -2.54000254e-01 -4.81127681e-16 -3.12770333e-01   8.16496581e-01]
 [-3.95029694e-01   1.56123876e-02 -5.28290830e-01 -6.82340484e-02
    1.27000127e-01 -7.07106781e-01 -2.09360158e-01   1.55512464e-17]
 [-2.02089013e-01 -3.80395849e-01 -2.12899198e-01   4.80790894e-01
    8.04483689e-16 -1.60632798e-15   7.33466480e-01   1.76241226e-16]
 [-3.95029694e-01   1.56123876e-02 -5.28290830e-01 -6.82340484e-02
    1.27000127e-01   7.07106781e-01 -2.09360158e-01 -1.23226632e-16]]
U.shape (8, 8)
```

#Output the singular matrix and its shape:

```
print("s=",s)
print("s.shape",s.shape)
```

Output：

the results in descending order, where each singular value corresponds to a left singular vector and a right singular vector:

```
s= [2.85653844 2.63792139 2.06449303 1.14829917 1.         1.
 0.54848559]
s.shape (7,)
```

#Output the right singular matrix Vh and its shape:

```
print("Vh",Vh)
print("Vh.shape",Vh.shape)
```

Output:

```
Vh [[-6.08788345e-01 -2.29949618e-01 -7.46612474e-02 -3.80854846e-01


   -3.47325416e-01 -3.80854846e-01 -4.00237243e-01]
 [ 2.65111314e-01 -8.71088358e-01 -3.51342402e-01   1.15234846e-01
   -1.32365989e-01   1.15234846e-01   5.83153945e-02]
 [ 5.66965547e-01   1.75382762e-01   1.55059743e-01   1.91316736e-02
   -6.14911671e-01   1.91316736e-02 -4.94872736e-01]
 [-6.48865369e-03   2.52237176e-01 -7.40339999e-01   1.34031699e-01
    2.99854608e-01   1.34031699e-01 -5.12239408e-01]
 [-2.54000254e-01 -2.54000254e-01   5.08000508e-01   3.81000381e-01
    2.54000254e-01   3.81000381e-01 -5.08000508e-01]
 [ 0.00000000e+00 -7.68640544e-16   2.33583082e-15 -7.07106781e-01
   -1.21802199e-15   7.07106781e-01   1.91457709e-15]
 [ 4.16034348e-01 -1.71550021e-01   2.01922906e-01 -4.22112199e-01
    5.73845817e-01 -4.22112199e-01 -2.66564648e-01]]
```



```
Vh.shape (7, 7)

#Set the coordinate axis ranges.
plt.axis([-0.8,0.2,-0.8,0.8])

#Each keyword is represented by a vector of 1x8. Now, the vector is reduced to a vector of 1x2 for visualization.
for i in range(len(words)):
    plt.text(U[i,0],U[i,1],words[i])
plt.show()
```

Figure 3-2 Visualization result

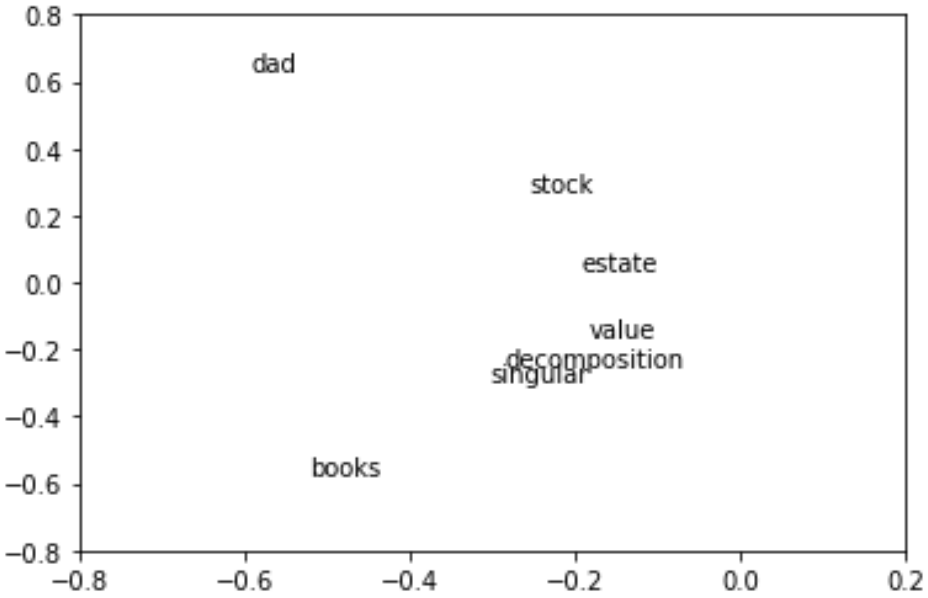


Illustration: After obtaining the visualized two-dimensional result, we can group keywords, such as **singular**, **value**, and **decomposition**, which are close to each other. The words **stock** and **estate** often appear at the same time.

After the word vector representation is obtained, we can calculate the similarity between words by selecting a vector distance (for example, Euclidean metric or Manhattan distance) calculation manner and obtain the similarity between article titles based on some policies (for example, a sum or a mean value of the word pair similarities). Then we can classify articles based on the similarity between article titles, which can approximately represent the similarity between article contents.

3.2.9 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

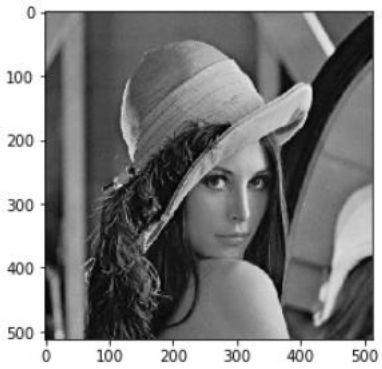
information, but the singular values usually attenuate quickly. Therefore, the first K singular values and corresponding singular vectors include most information in the image. As a result, an image formed by the first K singular values and their singular vectors can achieve basically the same definition 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:

Figure 3-3




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

Output:

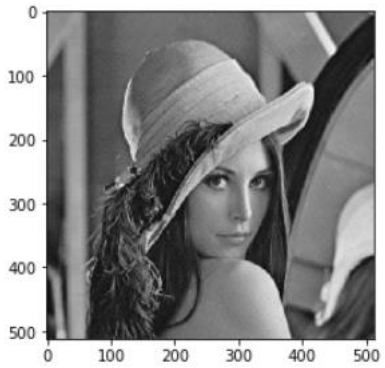
```
(512, 512)
```

```
# 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= 100
# 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()
```

Figure 3-4



3.2.10 Solving a System of Linear Equations

Solving a system of linear equations is simple because it requires only one function (scipy.linalg.solve).

Case introduction: There are 3 kinds of fruits with unknown prices: apples, bananas, grapes. Tom has spent CNY10 buying 10 kg apples, 2 kg bananas, and 5 kg grapes. Lily has spent CNY8 buying 4 kg apples, 4 kg bananas, and 2 kg grapes. Tom has spent CNY5 buying 2 kg apples, 2 kg bananas, and 2 kg grapes. How much does it cost to buy 1 kg apples, bananas, and grapes, respectively?

Based on the known conditions, the following equations can be constructed, where x_1, x_2, and x_3 are the prices of apples, bananas, and grapes, respectively. The purpose is to calculate the values of x_1, x_2, and x_3.

$10x_1 + 2x_2 + 5x_3 = 10$

$4x_1 + 4x_2 + 2x_3 = 8$

$2x_1 + 2x_2 + 2x_3 = 5$

Input:


```
import numpy as np
from scipy.linalg import solve
a = np.array([[10, 2, 5], [4, 4, 2], [2, 2, 2]])
b = np.array([10,8,5])
x = solve(a, b)
print(x)
```

Output of print(x):

```
array([0.25 1.25 1.   ])
```

4 Probability Theory Experiment

4.1 Introduction

4.1.1 Probability Theory

Probability theory 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.

The probability theory 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 Theory 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:

```
b=[1,3,5,6]
np.cov(b)
```

Output:

```
4.916666666666666
```

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 x parameter indicates the data distribution of each bin, corresponding to the x axis. The bins parameter indicates the number of bars in total. The normed parameter indicates whether to perform normalization. By default, the sum of the percentages of all bars is 1.
```



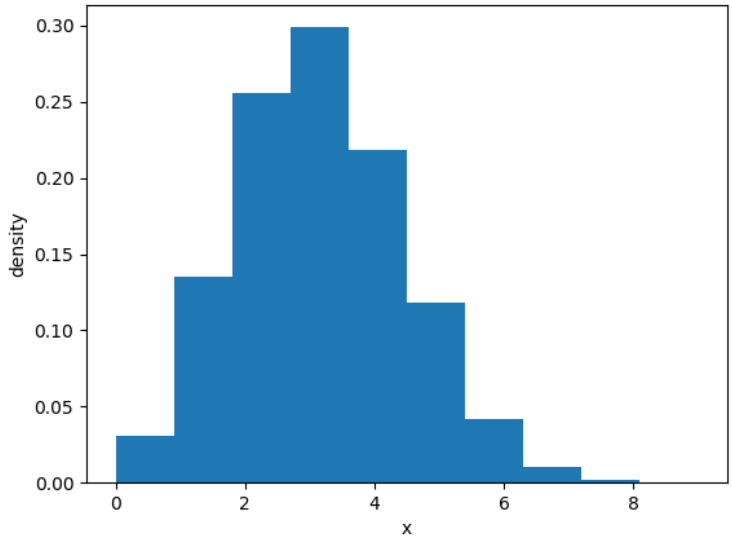
```
plt.hist(binom_sim, bins=10, normed=True)
plt.xlabel(('x'))
plt.ylabel('density')
plt.show()
```

Output:

```
Data: [2 4 3 ... 3 4 1]# 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. The **λ** parameter indicates the occurrence probability of an event. Both the mean value and variance of the random variable X are λ.

Input:

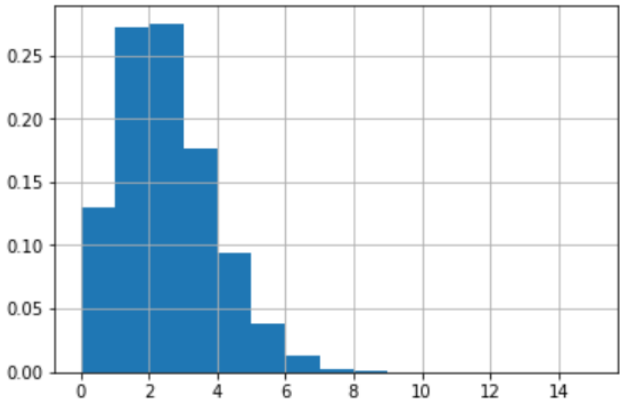
```
import numpy as np
import matplotlib.pyplot as plt

# 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, normed=True, range=[0, 15])
```



```
# Generate grids.
plt.grid()
plt.show()
```

The following figure shows the Poisson distribution.

Figure 4-2



4.2.8 Normal Distribution

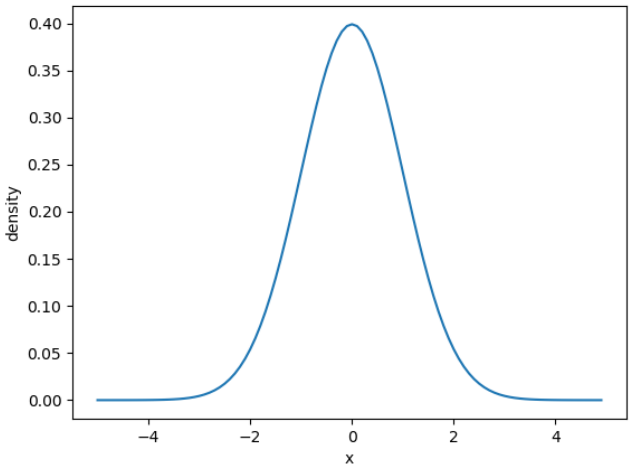
Normal distribution is a continuous probability distribution. Its function can obtain a value anywhere on the curve. Normal distribution is described by two parameters: **μ** and **σ**, which indicate the mean value and standard deviation, respectively.

Input:

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


mu = 0
sigma = 1
# Distribution sampling points.
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 Implementation of the Least Squares Method

5.1.1 Algorithm

The least squares method, as the basis of the classification regression algorithm, has a long history. It seeks the best function that matches data by minimizing the error sum of squares. The least squares method can easily obtain unknown parameters, and minimize the sum of the squared errors between the predicted data and the actual data.

5.1.2 Case Introduction

Assume that we need to predict the price of a stock based on historical data. According to past experience, the price of a stock fluctuates in a sinusoidal manner with time. Therefore, we want to use a polynomial $y = \theta_n x^n + \theta_{n-1} x^{n-1} + \ldots + \theta_1 x^0 + \theta_0 x^0$ to represent the relationship, where y indicates the stock price, x indicates the time, and the $\theta_i (i = 0, \ldots\ldots, n)$ parameter is unknown. Once the $\theta_i (i = 0, \ldots\ldots, n)$ parameter is determined, we can estimate the stock price of any time point according to the formula. How can we obtain a better $\theta_i (i = 0, \ldots\ldots, n)$ parameter based on the historical data to make the expression comply with the actual condition as much as possible? The most direct method is to use the time in historical data in the polynomial to predict the stock price $\hat{y}$. If predicted price $\hat{y}$ is close to the actual price y, the polynomial can almost reflect the actual condition. How do we express the difference between the stock price obtained by the polynomial and the actual stock price? One common way is to use the square of the difference between the two prices, that is, using the function $\frac{1}{2}(\hat{y} - y)^2$. However, we have a large amount of historical data. A good polynomial should be optimal in the overall data, not just at a specific data point. We can use $\frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}(\hat{y}_i - y_i)^2$ to represent the overall quality, where **m** indicates the number of historical data points. Then the problem is simplified into an optimization problem, that is, to obtain $\theta_i (i = 0, \ldots\ldots, n)$ that can minimize the value of $\frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}(\hat{y}_i - y_i)^2$. The following first creates sinusoidal data with noise, which is regarded as the historical data. Then the least squares method is used to obtain $\theta_i (□ = 0, \ldots\ldots, n)$ that minimizes the value of $\frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}(\hat{y}_i - y_i)^2$ based on the historical data.

5.1.3 Code Implementation

Input:

```
import numpy as np
import scipy as sp
import pylab as pl
from scipy.optimize import leastsq    # Introduce the least squares function.

n = 9    # Degree of the polynomial.
```

#Define a target function:

```
def real_func(x):
      # Target function: sin(2*pi*x)
    return np.sin(2 * np.pi * x)
```

#Define a polynomial function and use the polynomial to fit data:

```
def fit_func(p, x):
    f = np.poly1d(p)
    return f(x)
```

#Define a residual function. The value of the function is the difference between the polynomial fitting result and the actual value.

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

x = np.linspace(0, 1, 9)    # Randomly select nine points as x.
x_points = np.linspace(0, 1, 1000)    # Continuous points required for drawing a graph.
y0 = real_func(x)    # Target function.
y1 = [np.random.normal(0, 0.1) + y for y in y0]    # Function obtained after the noise that complies with normal distribution is added to the target function.
p_init = np.random.randn(n)    # Randomly initialize the polynomial parameters.
# Invoke the leastsq function of scipy.optimize to minimize the sum of the squared errors to find the optimal matching function.
# The func function is a residual function, and x0 is the initial parameter value. Package parameters except the initialization parameter in the residual function into args.
plsq = leastsq(func=residuals_func, x0=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()
```

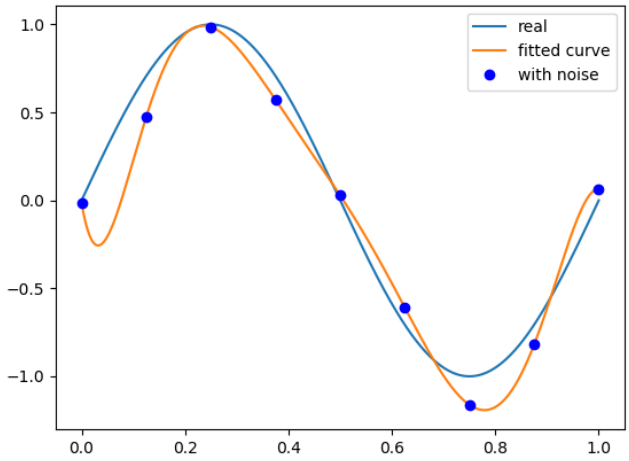


```
pl.show()
```

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]
```

Figure 5-1 Visualized chart



5.2 Gradient Descent Implementation

5.2.1 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.

5.2.2 Case Introduction

Currently, the real estate industry is popular. How do we predict the price of a house? According to experience, the house price y is determined by three key factors: house area $x0$, floor area ratio $x1$, and afforested area $x2$. The relationship between the house price and key factors is linear. That is, the relationship between the house price and key factors can be described through the following formula: $y = \theta_0 * x0 + \theta_1 * x1 + \theta_2 * x2$, where θ_0, θ_1, and θ_2 are unknown. Once the three values are

determined, we can estimate the price of any house according to the formula. In order to obtain θ_0, θ_1, and θ_2, we have found a copy of the city's house prices and data about the house areas, floor area ratios, and afforested areas. How do we use the data to help us obtain θ_0, θ_1, and θ_2? To evaluate the quality of θ_0, θ_1, and θ_2, we can compare the value of y_{pre} (obtained through the formula θ_0*s + θ_1*r + θ_2*d based on most of the actual data) with that of the real house price y. If the values are similar, the θ_0, θ_1, and θ_2 are high-quality data. Similar to the case in 5.1.2 Case Introduction where we have obtained a large amount of known data, a good polynomial should be optimal in the overall data rather than just at some data points. Therefore, $\frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}\left(y_i - y_{(pre,i)}\right)^2$ is used to show the overall quality, and **m** indicates the number of known data points. Then the problem is simplified into an optimization problem, that is, to obtain θ_0, θ_1, and θ_2 that can minimize the value of $\frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}\left(y_i - y_{(pre,i)}\right)^2$. The following is an example. We have five copies of the actual data.

House Price	House Area	Floor Area Ratio	Afforested Area
95.364	1	0	3
97.217205	1	1	3
75.195834	1	2	3
60.105519	1	3	2
49.342380	1	4	4

The following describes how to use the gradient descent algorithm to obtain the optimal θ_0, θ_1, and θ_2 to minimize the value of $\frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}\left(y_i - y_{(pre,i)}\right)^2$.

5.2.3 Code Implementation

Input:

#There are five samples in the training set (x,y), and each sample point has three components (x0,x1,x2)

```
x = [(1, 0., 3), (1, 1., 3), (1, 2., 3), (1, 3., 2), (1, 4., 4)]
y = [95.364, 97.217205, 75.195834, 60.105519, 49.342380]  # Output corresponding to the y[i] sample point.
epsilon = 0.0001  # Iteration threshold. (When the difference between the two iteration loss functions is less than the threshold, the iteration stops.)
alpha = 0.01  # Learning rate.
diff = [0, 0]
max_itor = 1000
error1 = 0
error0 = 0
cnt = 0
m = len(x)
# Initialize parameters.
```



```
theta0 = 0
theta1 = 0
theta2 = 0
while True:
    cnt += 1

    # Parameter iteration calculation.
    for i in range(m):
        # The fitting function is as follows: y = theta0 * x[0] + theta1 * x[1] +theta2 * x[2]
        # Calculate the residual value (Value of the fitting function – Actual value).
        diff[0] = (theta0* x[i][0] + theta1 * x[i][1] + theta2 * x[i][2]) - y[i]
        # Gradient = diff[0] * x[i][j]. Update parameters based on the value (Step x Gradient).
        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* x[lp][0] + theta1 * x[lp][1] + theta2 * x[lp][2]))**2/2
    # If the difference between two iteration loss functions is less than the threshold, the iteration stops and the loop ends.
    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    )
```

As parameters are updated continuously based on the gradient descent algorithm, the error becomes smaller and smaller until the loop ends. The result is as follows:

```
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

