EBook Gratis

APRENDIZAJE numpy

Free unaffiliated eBook created from
Stack Overflow contributors.

#numpy

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: numpy

It is an unofficial and free numpy ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official numpy.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con numpy

Observaciones

NumPy (pronunciado "numb pie" o, a veces, "numb pea") es una extensión del lenguaje de programación Python que agrega soporte para matrices grandes y multidimensionales, junto con una amplia biblioteca de funciones matemáticas de alto nivel para operar en estas matrices.

Versiones

Versión	Fecha de lanzamiento
1.3.0	2009-03-20
1.4.0	2010-07-21
1.5.0	2010-11-18
1.6.0	2011-05-15
1.6.1	2011-07-24
1.6.2	2012-05-20
1.7.0	2013-02-12
1.7.1	2013-04-07
1.7.2	2013-12-31
1.8.0	2013-11-10
1.8.1	2014-03-26
1.8.2	2014-08-09
1.9.0	2014-09-07
1.9.1	2014-11-02
1.9.2	2015-03-01
1.10.0	2015-10-07
1.10.1	2015-10-12
1.10.2	2015-12-14

Versión	Fecha de lanzamiento
1.10.4 *	2016-01-07
1.11.0	2016-05-29

Examples

Instalación en Mac

La forma más fácil de configurar NumPy en Mac es con pip

```
pip install numpy
```

Instalación mediante Conda .
Conda disponible para Windows, Mac y Linux.

1. Instale Conda. Hay dos formas de instalar Conda, ya sea con Anaconda (paquete completo, incluye numpy) o Miniconda (solo Conda, Python y los paquetes de los que dependen, sin ningún paquete adicional). Tanto Anaconda como Miniconda instalan la misma Conda.
2. Comando adicional para Miniconda, escriba el comando `conda install numpy`

Instalación en Windows

En general, la instalación a través de pypi (el índice de paquete predeterminado usado por pip) generalmente falla en las computadoras con Windows. La forma más fácil de instalar en Windows es mediante el uso de archivos binarios precompilados.

Una fuente de ruedas precompiladas de muchos paquetes es el sitio de Christopher Gohkle . Elija una versión de acuerdo con su versión y sistema de Python. Un ejemplo para Python 3.5 en un sistema de 64 bits:

1. Descargue `numpy-1.11.1+mkl-cp35-cp35m-win_amd64.whl` desde aquí
2. Abra un terminal de Windows (cmd o powershell)
3. Escriba el comando `pip install C:\path_to_download\numpy-1.11.1+mkl-cp35-cp35m-win_amd64.whl`

Si no quiere perder el tiempo con paquetes individuales, puede usar la distribución Winpython que agrupa a la mayoría de los paquetes y proporciona un entorno limitado para trabajar. De manera similar, la distrubución de Anaconda Python viene preinstalada con muchos y otros paquetes comunes.

Otra fuente popular es el `conda` paquetes `conda` , que también es compatible con entornos virtuales .

1. Descarga e instala `conda` .
2. Abra un terminal de Windows.
3. Escribe el comando `conda install numpy`

Instalación en Linux

NumPy está disponible en los repositorios predeterminados de las distribuciones de Linux más populares y se puede instalar de la misma manera que los paquetes en una distribución de Linux generalmente se instalan.

Algunas distribuciones de Linux tienen diferentes paquetes NumPy para Python 2.xy Python 3.x. En Ubuntu y Debian, instale `numpy` en el nivel del sistema usando el administrador de paquetes APT:

```
sudo apt-get install python-numpy
sudo apt-get install python3-numpy
```

Para otras distribuciones, use sus administradores de paquetes, como zypper (Suse), yum (Fedora), etc.

`numpy` también se puede instalar con el gestor de paquetes de Python `pip` para Python 2 y con `pip3` para Python 3:

```
pip install numpy  # install numpy for Python 2
pip3 install numpy  # install numpy for Python 3
```

`pip` está disponible en los repositorios predeterminados de las distribuciones de Linux más populares y se puede instalar para Python 2 y Python 3 usando:

```
sudo apt-get install python-pip  # pip for Python 2
sudo apt-get install python3-pip  # pip for Python 3
```

Después de la instalación, use `pip` para Python 2 y `pip3` para Python 3 para usar pip para instalar paquetes de Python. Pero tenga en cuenta que es posible que necesite instalar muchas dependencias, que son necesarias para compilar numpy desde la fuente (incluidos los paquetes de desarrollo, compiladores, fortran, etc.).

Además de instalar `numpy` en el nivel del sistema, también es común (quizás incluso muy recomendable) instalar `numpy` en entornos virtuales utilizando paquetes populares de Python como `virtualenv` . En Ubuntu, `virtualenv` se puede instalar usando:

```
sudo apt-get install virtualenv
```

Luego, crea y activa un virtualenv para Python 2 o Python 3 y luego usa `pip` para instalar `numpy` :

```
virtualenv venv  # create virtualenv named venv for Python 2
virtualenv venv -p python3  # create virtualenv named venv for Python 3
source venv/bin/activate  # activate virtualenv named venv
pip install numpy  # use pip for Python 2 and Python 3; do not use pip3 for Python3
```

Importación básica

Importe el módulo numpy para usar cualquier parte de él.

```
import numpy as np
```

La mayoría de los ejemplos usarán `np` como abreviatura para numpy. Suponga que "np" significa "numpy" en los ejemplos de código.

```
x = np.array([1,2,3,4])
```

Cuaderno temporal de Jupyter alojado por Rackspace

Los portátiles Jupyter son un entorno de desarrollo interactivo y basado en navegador. Fueron desarrollados originalmente para ejecutar la computación python y, como tal, juegan muy bien con numpy. Para probar el numpy en una computadora portátil Jupyter sin instalar completamente ninguno de los sistemas locales, Rackspace proporciona computadoras portátiles temporales gratuitas en tmpnb.org .

Nota: este no es un servicio propietario con ningún tipo de ventas adicionales. Jupyter es una tecnología totalmente de código abierto desarrollada por UC Berkeley y Cal Poly San Luis Obispo. Rackspace dona este servicio como parte del proceso de desarrollo.

Para probar `numpy` en tmpnb.org:

1. visita tmpnb.org
2. seleccione `Welcome to Python.ipynb` o
3. Nuevo >> Python 2 o
4. Nuevo >> Python 3

Lea Empezando con numpy en línea: https://riptutorial.com/es/numpy/topic/823/empezando-con-numpy

Capítulo 2: Ahorro y carga de matrices

Introducción

Las matrices numpy se pueden guardar y cargar de varias maneras.

Examples

Usando numpy.save y numpy.load

np.save y np.load proporcionan un marco fácil de usar para guardar y cargar arrays de números de tamaño arbitrario:

```
import numpy as np

a = np.random.randint(10,size=(3,3))
np.save('arr', a)

a2 = np.load('arr.npy')
print a2
```

Lea Ahorro y carga de matrices en línea: https://riptutorial.com/es/numpy/topic/10891/ahorro-y-carga-de-matrices

Capítulo 3: Álgebra lineal con np.linalg

Observaciones

A partir de la versión 1.8, varias de las rutinas en `np.linalg` pueden operar en una 'pila' de matrices. Es decir, la rutina puede calcular resultados para matrices múltiples si se apilan juntas. Por ejemplo, `A` aquí se interpreta como dos matrices apiladas de 3 por 3:

```
np.random.seed(123)
A = np.random.rand(2,3,3)
b = np.random.rand(2,3)
x = np.linalg.solve(A, b)

print np.dot(A[0,:,:], x[0,:])
# array([ 0.53155137,  0.53182759,  0.63440096])

print b[0,:]
# array([ 0.53155137,  0.53182759,  0.63440096])
```

Los documentos oficiales de `np` especifican esto a través de especificaciones de parámetros como `a : (..., M, M) array_like .`

Examples

Resolver sistemas lineales con np.solve.

Considera las siguientes tres ecuaciones:

```
x0 + 2 * x1 + x2 = 4
     x1 + x2 = 3
x0 +          x2 = 5
```

Podemos expresar este sistema como una ecuación matricial `A * x = b` con:

```
A = np.array([[1, 2, 1],
              [0, 1, 1],
              [1, 0, 1]])
b = np.array([4, 3, 5])
```

Luego, use `np.linalg.solve` para resolver para `x` :

```
x = np.linalg.solve(A, b)
# Out: x = array([ 1.5, -0.5,  3.5])
```

`A` debe ser una matriz cuadrada y de rango completo: todas sus filas deben ser linealmente independientes. `A` debe ser invertible / no singular (su determinante no es cero). Por ejemplo, si una fila de `A` es un múltiplo de otra, llamar a `linalg.solve` generará `LinAlgError: Singular matrix` :

```
A = np.array([[1, 2, 1],
              [2, 4, 2],   # Note that this row 2 * the first row
              [1, 0, 1]])
b = np.array([4,8,5])
```

Tales sistemas se pueden resolver con `np.linalg.lstsq` .

Encuentre la solución de mínimos cuadrados para un sistema lineal con np.linalg.lstsq

Los cuadrados mínimos es un enfoque estándar para problemas con más ecuaciones que incógnitas, también conocidas como sistemas sobredeterminados.

Considera las cuatro ecuaciones:

```
x0 + 2 * x1 + x2 = 4
x0 + x1 + 2 * x2 = 3
2 * x0 + x1 + x2 = 5
x0 + x1 + x2 = 4
```

Podemos expresar esto como una multiplicación matricial `A * x = b` :

```
A = np.array([[1, 2, 1],
              [1,1,2],
              [2,1,1],
              [1,1,1]])
b = np.array([4,3,5,4])
```

Luego resuelve con `np.linalg.lstsq` :

```
x, residuals, rank, s = np.linalg.lstsq(A,b)
```

`x` es la solución, `residuals` la suma, `rank` el rango de matriz de la entrada `A` y `s` los valores singulares de `A` Si `b` tiene más de una dimensión, `lstsq` resolverá el sistema correspondiente a cada columna de `b` :

```
A = np.array([[1, 2, 1],
              [1,1,2],
              [2,1,1],
              [1,1,1]])
b = np.array([[4,3,5,4],[1,2,3,4]]).T # transpose to align dimensions
x, residuals, rank, s = np.linalg.lstsq(A,b)
print x # columns of x are solutions corresponding to columns of b
#[[ 2.05263158  1.63157895]
# [ 1.05263158 -0.36842105]
# [ 0.05263158  0.63157895]]
print residuals # also one for each column in b
#[ 0.84210526  5.26315789]
```

`rank` y `s` solo dependen de `A` , y por lo tanto son los mismos que los anteriores.

Lea Álgebra lineal con np.linalg en línea: https://riptutorial.com/es/numpy/topic/3753/algebra-

lineal-con-np-linalg

Capítulo 4: Archivo IO con numpy

Examples

Guardando y cargando matrices numpy usando archivos binarios

```
x = np.random.random([100,100])
x.tofile('/path/to/dir/saved_binary.npy')
y = fromfile('/path/to/dir/saved_binary.npy')
z = y.reshape(100,100)
all(x==z)
# Output:
#     True
```

Cargando datos numéricos de archivos de texto con estructura consistente.

La función `np.loadtxt` se puede usar para leer archivos similares a csv:

```
# File:
#    # Col_1 Col_2
#    1, 1
#    2, 4
#    3, 9
np.loadtxt('/path/to/dir/csvlike.txt', delimiter=',', comments='#')
# Output:
# array([[ 1.,  1.],
#        [ 2.,  4.],
#        [ 3.,  9.]])
```

El mismo archivo podría leerse usando una expresión regular con `np.fromregex` :

```
np.fromregex('/path/to/dir/csvlike.txt', r'(\d+),\s(\d+)', np.int64)
# Output:
# array([[1, 1],
#        [2, 4],
#        [3, 9]])
```

Guardando datos como archivo ASCII estilo CSV

De forma `np.loadtxt` a `np.loadtxt` , `np.savetxt` se puede usar para guardar datos en un archivo ASCII

```
import numpy as np
x = np.random.random([100,100])
np.savetxt("filename.txt", x)
```

Para controlar el formato:

```
np.savetxt("filename.txt", x, delimiter=", " ,
```

```
    newline="\n", comments="$ ", fmt="%1.2f",
    header="commented example text")
```

Salida:

```
$ commented example text
0.30, 0.61, 0.34, 0.13, 0.52, 0.62, 0.35, 0.87, 0.48, [...]
```

Leyendo archivos CSV

Tres funciones principales disponibles (descripción de las páginas del manual):

> `fromfile` : una forma altamente eficiente de leer datos binarios con un tipo de datos conocido, así como analizar archivos de texto con formato simple. Los datos escritos usando el método tofile se pueden leer usando esta función.
>
> `genfromtxt` : carga datos de un archivo de texto, con los valores faltantes manejados como se especifica. Cada línea después de las primeras líneas skip_header se divide en el carácter delimitador y los caracteres que siguen al carácter de comentarios se descartan.
>
> `loadtxt` : carga los datos de un archivo de texto. Cada fila en el archivo de texto debe tener el mismo número de valores.

`genfromtxt` es una función de envoltorio para `loadtxt` . `genfromtxt` es el más sencillo de usar, ya que tiene muchos parámetros para manejar el archivo de entrada.

Número consistente de columnas, tipo de datos consistente (numérico o cadena):

Dado un archivo de entrada, `myfile.csv` con el contenido:

```
#descriptive text line to skip
1.0, 2, 3
4, 5.5, 6

import numpy as np
np.genfromtxt('path/to/myfile.csv',delimiter=',',skiprows=1)
```

da una matriz:

```
array([[ 1. ,  2. ,  3. ],
       [ 4. ,  5.5,  6. ]])
```

Número consistente de columnas, tipo de datos mixtos (a través de columnas):

```
1   2.0000  buckle_my_shoe
3   4.0000  margery_door

import numpy as np
np.genfromtxt('filename', dtype= None)
```

```
array([(1, 2.0, 'buckle_my_shoe'), (3, 4.0, 'margery_door')],
dtype=[('f0', '<i4'), ('f1', '<f8'), ('f2', '|S14')])
```

Tenga en cuenta que el uso de `dtype=None` da lugar a una nueva versión.

Cantidad inconsistente de columnas:

archivo: 1 2 3 4 5 6 7 8 9 10 11 22 13 14 15 16 17 18 19 20 21 22 23 24

En la matriz de una sola fila:

```
result=np.fromfile(path_to_file,dtype=float,sep="\t",count=-1)
```

Lea Archivo IO con numpy en línea: https://riptutorial.com/es/numpy/topic/4973/archivo-io-con-numpy

Capítulo 5: Arrays

Introducción

Las matrices de N-dimensional o `ndarrays` son el objeto central de numpy utilizado para almacenar elementos del mismo tipo de datos. Proporcionan una estructura de datos eficiente que es superior a las matrices comunes de Python.

Observaciones

Siempre que sea posible, exprese operaciones sobre datos en términos de matrices y operaciones vectoriales. Las operaciones vectoriales se ejecutan mucho más rápido que el equivalente para los bucles.

Examples

Crear una matriz

Matriz vacía

```
np.empty((2,3))
```

Tenga en cuenta que, en este caso, los valores de esta matriz no están establecidos. Esta forma de crear una matriz, por lo tanto, solo es útil si la matriz se llena más adelante en el código.

De una lista

```
np.array([0,1,2,3])
# Out: array([0, 1, 2, 3])
```

Crear un rango

```
np.arange(4)
# Out: array([0, 1, 2, 3])
```

Crear ceros

```
np.zeros((3,2))
# Out:
# array([[ 0.,  0.],
#        [ 0.,  0.],
#        [ 0.,  0.]])
```

Crear unos

```
np.ones((3,2))
# Out:
# array([[ 1.,  1.],
#        [ 1.,  1.],
#        [ 1.,  1.]])
```

Crear elementos de matriz espaciados lineales

```
np.linspace(0,1,21)
# Out:
# array([ 0.  ,  0.05,  0.1 ,  0.15,  0.2 ,  0.25,  0.3 ,  0.35,  0.4 ,
#         0.45,  0.5 ,  0.55,  0.6 ,  0.65,  0.7 ,  0.75,  0.8 ,  0.85,
#         0.9 ,  0.95,  1.  ])
```

Crear elementos de matriz espaciados por registro

```
np.logspace(-2,2,5)
# Out:
# array([  1.00000000e-02,   1.00000000e-01,   1.00000000e+00,
#          1.00000000e+01,   1.00000000e+02])
```

Crear una matriz a partir de una función dada

```
np.fromfunction(lambda i: i**2, (5,))
# Out:
# array([  0.,   1.,   4.,   9.,  16.])
np.fromfunction(lambda i,j: i**2, (3,3))
# Out:
# array([[ 0.,  0.,  0.],
#        [ 1.,  1.,  1.],
#        [ 4.,  4.,  4.]])
```

Operadores de matrices

```
x = np.arange(4)
x
#Out:array([0, 1, 2, 3])
```

la adición escalar es un elemento sabio

```
x+10
#Out: array([10, 11, 12, 13])
```

la multiplicación escalar es un elemento sabio

```
x*2
#Out: array([0, 2, 4, 6])
```

la suma de matrices es un elemento sabio

```
x+x
```

```
#Out: array([0, 2, 4, 6])
```

la multiplicación de matrices es un elemento sabio

```
x*x
#Out: array([0, 1, 4, 9])
```

El producto puntual (o más generalmente la multiplicación de matrices) se realiza con una función.

```
x.dot(x)
#Out: 14
```

En Python 3.5, el operador `@` se agregó como operador de infijo para la multiplicación de matrices

```
x = np.diag(np.arange(4))
print(x)
'''
   Out: array([[0, 0, 0, 0],
   [0, 1, 0, 0],
   [0, 0, 2, 0],
   [0, 0, 0, 3]])
'''
print(x@x)
print(x)
'''
   Out: array([[0, 0, 0, 0],
   [0, 1, 0, 0],
   [0, 0, 4, 0],
   [0, 0, 0, 9]])
'''
```

Anexar Devuelve copia con los valores adjuntos. Fuera de lugar.

```
#np.append(array, values_to_append, axis=None)
x = np.array([0,1,2,3,4])
np.append(x, [5,6,7,8,9])
# Out: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
x
# Out: array([0, 1, 2, 3, 4])
y = np.append(x, [5,6,7,8,9])
y
# Out: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

hstack Pila horizontal. (pila de columnas)
Vack Pila vertical. (pila de filas)

```
# np.hstack(tup), np.vstack(tup)
x = np.array([0,0,0])
y = np.array([1,1,1])
z = np.array([2,2,2])
np.hstack(x,y,z)
# Out: array([0, 0, 0, 1, 1, 1, 2, 2, 2])
np.vstack(x,y,z)
```

```
# Out: array([[0, 0, 0],
#             [1, 1, 1],
#             [2, 2, 2]])
```

Acceso a Array

La sintaxis de división es `i:j:k` donde `i` es el índice de inicio (incluido), `j` es el índice de detención (exclusivo) y `k` es el tamaño del paso. Al igual que otras estructuras de datos de Python, el primer elemento tiene un índice de 0:

```
x = np.arange(10)
x[0]
# Out: 0

x[0:4]
# Out: array([0, 1, 2, 3])

x[0:4:2]
# Out:array([0, 2])
```

Los valores negativos cuentan desde el final de la matriz. `-1` por lo tanto accede al último elemento en una matriz:

```
x[-1]
# Out: 9
x[-1:0:-1]
# Out: array([9, 8, 7, 6, 5, 4, 3, 2, 1])
```

Se puede acceder a las matrices multidimensionales especificando cada dimensión separada por comas. Se aplican todas las reglas anteriores.

```
x = np.arange(16).reshape((4,4))
x
# Out:
#     array([[ 0,  1,  2,  3],
#            [ 4,  5,  6,  7],
#            [ 8,  9, 10, 11],
#            [12, 13, 14, 15]])

x[1,1]
# Out: 5

x[0:3,0]
# Out: array([0, 4, 8])

x[0:3, 0:3]
# Out:
#     array([[ 0,  1,  2],
#            [ 4,  5,  6],
#            [ 8,  9, 10]])

x[0:3:2, 0:3:2]
# Out:
#     array([[ 0,  2],
#            [ 8, 10]])
```

Transponiendo una matriz

```
arr = np.arange(10).reshape(2, 5)
```

Utilizando el método `.transpose` :

```
arr.transpose()
# Out:
#      array([[0, 5],
#             [1, 6],
#             [2, 7],
#             [3, 8],
#             [4, 9]])
```

`.T` método:

```
arr.T
# Out:
#     array([[0, 5],
#             [1, 6],
#             [2, 7],
#             [3, 8],
#             [4, 9]])
```

O `np.transpose` :

```
np.transpose(arr)
# Out:
#     array([[0, 5],
#             [1, 6],
#             [2, 7],
#             [3, 8],
#             [4, 9]])
```

En el caso de una matriz bidimensional, esto es equivalente a una transposición de matriz estándar (como se muestra arriba). En el caso de n dimensiones, puede especificar una permutación de los ejes de matriz. Por defecto, esto invierte `array.shape` :

```
a = np.arange(12).reshape((3,2,2))
a.transpose() # equivalent to a.transpose(2,1,0)
# Out:
#   array([[[ 0,  4,  8],
#           [ 2,  6, 10]],
#
#          [[ 1,  5,  9],
#           [ 3,  7, 11]]])
```

Pero cualquier permutación de los índices de eje es posible:

```
a.transpose(2,0,1)
# Out:
#    array([[[ 0,  2],
#            [ 4,  6],
```

```
#            [ 8, 10]],
#
#           [[ 1,  3],
#            [ 5,  7],
#            [ 9, 11]]])

a = np.arange(24).reshape((2,3,4))  # shape (2,3,4)
a.transpose(2,0,1).shape
# Out:
#    (4, 2, 3)
```

Indexación booleana

```
arr = np.arange(7)
print(arr)
# Out: array([0, 1, 2, 3, 4, 5, 6])
```

La comparación con un escalar devuelve una matriz booleana:

```
arr > 4
# Out: array([False, False, False, False, False,  True,  True], dtype=bool)
```

Esta matriz se puede usar en la indexación para seleccionar solo los números mayores de 4:

```
arr[arr>4]
# Out: array([5, 6])
```

La indexación booleana se puede utilizar entre diferentes matrices (por ejemplo, matrices paralelas relacionadas):

```
# Two related arrays of same length, i.e. parallel arrays
idxs = np.arange(10)
sqrs = idxs**2

# Retrieve elements from one array using a condition on the other
my_sqrs = sqrs[idxs % 2 == 0]
print(my_sqrs)
# Out: array([0, 4, 16, 36, 64])
```

Remodelando una matriz

El `numpy.reshape` (igual que `numpy.ndarray.reshape`) devuelve una matriz del mismo tamaño total, pero en una nueva forma:

```
print(np.arange(10).reshape((2, 5)))
# [[0 1 2 3 4]
#  [5 6 7 8 9]]
```

Devuelve una nueva matriz y no funciona en su lugar:

```
a = np.arange(12)
```

```
a.reshape((3, 4))
print(a)
# [ 0  1  2  3  4  5  6  7  8  9 10 11]
```

Sin embargo, *es* posible sobrescribir el atributo de `shape` de un `ndarray` :

```
a = np.arange(12)
a.shape = (3, 4)
print(a)
# [[ 0  1  2  3]
#  [ 4  5  6  7]
#  [ 8  9 10 11]]
```

Este comportamiento puede ser sorprendente al principio, pero los `ndarray` s se almacenan en bloques contiguos de memoria, y su `shape` solo especifica cómo este flujo de datos debe interpretarse como un objeto multidimensional.

Hasta un eje en la tupla de `shape` puede tener un valor de `-1` . `numpy` entonces `numpy` la longitud de este eje para ti:

```
a = np.arange(12)
print(a.reshape((3, -1)))
# [[ 0  1  2  3]
#  [ 4  5  6  7]
#  [ 8  9 10 11]]
```

O:

```
a = np.arange(12)
print(a.reshape((3, 2, -1)))

# [[[ 0  1]
#   [ 2  3]]

#  [[ 4  5]
#   [ 6  7]]

#  [[ 8  9]
#   [10 11]]]
```

Múltiples dimensiones no especificadas, por ejemplo, `a.reshape((3, -1, -1))` no están permitidas y lanzarán un `ValueError` .

Operaciones de matriz de difusión

Las operaciones aritméticas se realizan elementwise en matrices de Numpy. Para matrices de forma idéntica, esto significa que la operación se ejecuta entre elementos en los índices correspondientes.

```
# Create two arrays of the same size
a = np.arange(6).reshape(2, 3)
b = np.ones(6).reshape(2, 3)
```

```
a
# array([0, 1, 2],
#       [3, 4, 5])
b
# array([1, 1, 1],
#       [1, 1, 1])

# a + b: a and b are added elementwise
a + b
# array([1, 2, 3],
#       [4, 5, 6])
```

Las operaciones aritméticas también pueden ejecutarse en matrices de diferentes formas por medio de la *difusión* Numpy. En general, una matriz se "difunde" sobre la otra para que las operaciones elementales se realicen en subarreglas de forma congruente.

```
# Create arrays of shapes (1, 5) and (13, 1) respectively
a = np.arange(5).reshape(1, 5)
a
# array([[0, 1, 2, 3, 4]])
b = np.arange(4).reshape(4, 1)
b
# array([0],
#       [1],
#       [2],
#       [3])

# When multiplying a * b, slices with the same dimensions are multiplied
# elementwise. In the case of a * b, the one and only row of a is multiplied
# with each scalar down the one and only column of b.
a*b
# array([[ 0,  0,  0,  0,  0],
#        [ 0,  1,  2,  3,  4],
#        [ 0,  2,  4,  6,  8],
#        [ 0,  3,  6,  9, 12]])
```

Para ilustrar esto aún más, considere la multiplicación de matrices 2D y 3D con subdimensiones congruentes.

```
# Create arrays of shapes (2, 2, 3) and (2, 3) respectively
a = np.arange(12).reshape(2, 2, 3)
a
# array([[[ 0  1  2]
#         [ 3  4  5]]
#
#        [[ 6  7  8]
#         [ 9 10 11]]])
b = np.arange(6).reshape(2, 3)
# array([[0, 1, 2],
#        [3, 4, 5]])

# Executing a*b broadcasts b to each (2, 3) slice of a,
# multiplying elementwise.
a*b
# array([[[ 0,  1,  4],
#         [ 9, 16, 25]],
#
```

```
#        [[ 0,  7, 16],
#         [27, 40, 55]]])

# Executing b*a gives the same result, i.e. the smaller
# array is broadcast over the other.
```

¿Cuándo se aplica la difusión de matriz?

La difusión tiene lugar cuando dos matrices tienen formas *compatibles* .

Las formas se comparan por componentes partiendo de las que se arrastran. Dos dimensiones son compatibles si son iguales o una de ellas es `1` . Si una forma tiene una dimensión mayor que la otra, no se comparan los componentes excedentes.

Algunos ejemplos de formas compatibles:

```
(7, 5, 3)    # compatible because dimensions are the same
(7, 5, 3)


(7, 5, 3)    # compatible because second dimension is 1
(7, 1, 3)

(7, 5, 3, 5) # compatible because exceeding dimensions are not compared
      (3, 5)

(3, 4, 5)    # incompatible
   (5, 5)

(3, 4, 5)    # compatible
   (1, 5)
```

Aquí está la documentación oficial sobre la difusión de matriz .

Rellena una matriz con el contenido de un archivo CSV

```
filePath = "file.csv"
data = np.genfromtxt(filePath)
```

Se admiten muchas opciones, consulte la documentación oficial para obtener una lista completa:

```
data = np.genfromtxt(filePath, dtype='float', delimiter=';', skip_header=1, usecols=(0,1,3) )
```

Numpy matriz n-dimensional: la ndarray

La estructura de datos principal en numpy es la `ndarray` (abreviatura de la matriz *n* -dimensional). `ndarray` s son

- homogéneos (es decir, contienen elementos del mismo tipo de datos)
- contienen elementos de tamaños fijos (dados por una *forma* , una tupla de *n* enteros

positivos que especifican los tamaños de cada dimensión)

Matriz unidimensional:

```
x = np.arange(15)
# array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])
x.shape
# (15,)
```

Matriz bidimensional:

```
x = np.asarray([[0, 1, 2, 3, 4], [5, 6, 7, 8, 9], [10, 11, 12, 13, 14]])
x
# array([[ 0, 1, 2, 3, 4],
# [ 5, 6, 7, 8, 9],
# [10, 11, 12, 13, 14]])
x.shape
# (3, 5)
```

Tridimensional:

```
np.arange(12).reshape([2,3,2])
```

Para inicializar una matriz sin especificar su contenido usa:

```
x = np.empty([2, 2])
# array([[ 0., 0.],
# [ 0., 0.]])
```

Adivinación de tipo de datos y casting automático.

El tipo de datos está configurado para flotar por defecto

```
x = np.empty([2, 2])
# array([[ 0., 0.],
# [ 0., 0.]])

x.dtype
# dtype('float64')
```

Si se proporcionan algunos datos, numpy adivina el tipo de datos:

```
x = np.asarray([[1, 2], [3, 4]])
x.dtype
# dtype('int32')
```

Tenga en cuenta que al realizar asignaciones numpy intentará convertir automáticamente los valores para que se ajusten al tipo de datos del `ndarray`

```
x[1, 1] = 1.5 # assign a float value
x[1, 1]
# 1
```

```
# value has been casted to int
x[1, 1] = 'z' # value cannot be casted, resulting in a ValueError
```

Difusión de matriz

Consulte también Operaciones de matriz de difusión .

```
x = np.asarray([[1, 2], [3, 4]])
# array([[1, 2],
#        [3, 4]])
y = np.asarray([[5, 6]])
# array([[5, 6]])
```

En terminología matricial, tendríamos una matriz 2x2 y un vector fila 1x2. Todavía somos capaces de hacer una suma.

```
# x + y
array([[ 6, 8],
       [ 8, 10]])
```

Esto se debe a que la matriz `y` está " *estirada* " para:

```
array([[5, 6],
       [5, 6]])
```

para adaptarse a la forma de `x` .

Recursos:

- Introducción a la ndarray a partir de la documentación oficial: La matriz N-dimensional (ndarray)
- Referencia de la clase: ndarray .

Lea Arrays en línea: https://riptutorial.com/es/numpy/topic/1296/arrays

Capítulo 6: Filtrando datos

Examples

Filtrado de datos con una matriz booleana

Cuando solo se proporciona un único argumento a numpy `where` funciona, devuelve los índices de la matriz de entrada (la `condition`) que se evalúan como verdaderas (el mismo comportamiento que `numpy.nonzero`). Esto se puede usar para extraer los índices de una matriz que satisfacen una condición dada.

```
import numpy as np

a = np.arange(20).reshape(2,10)
# a = array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
#            [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]])

# Generate boolean array indicating which values in a are both greater than 7 and less than 13
condition = np.bitwise_and(a>7, a<13)
# condition = array([[False, False, False, False, False, False, False, False,  True, True],
#                    [True,  True,  True, False, False, False, False, False, False, False]],
dtype=bool)

# Get the indices of a where the condition is True
ind = np.where(condition)
# ind = (array([0, 0, 1, 1, 1]), array([8, 9, 0, 1, 2]))

keep = a[ind]
# keep = [ 8  9 10 11 12]
```

Si no necesita los índices, esto se puede lograr en un solo paso usando `extract` , donde nuevamente especifique la `condition` como primer argumento, pero le da a la `array` que devuelva los valores desde donde la condición es verdadera como segundo argumento.

```
# np.extract(condition, array)
keep = np.extract(condition, a)
# keep = [ 8  9 10 11 12]
```

Se pueden proporcionar dos argumentos adicionales, `x` e `y` a `where` , en cuyo caso, la salida contendrá los valores de `x` donde la condición es `True` y los valores de `y` donde la condición es `False` .

```
# Set elements of a which are NOT greater than 7 and less than 13 to zero, np.where(condition,
x, y)
a = np.where(condition, a, a*0)
print(a)
# Out: array([[ 0,  0,  0,  0,  0,  0,  0,  0,  8,  9],
#             [10, 11, 12,  0,  0,  0,  0,  0,  0,  0]])
```

Índices de filtrado directo.

Para casos simples, puede filtrar datos directamente.

```
a = np.random.normal(size=10)
print(a)
#[-1.19423121  1.10481873  0.26332982 -0.53300387 -0.04809928  1.77107775
# 1.16741359  0.17699948 -0.06342169 -1.74213078]
b = a[a>0]
print(b)
#[ 1.10481873  0.26332982  1.77107775  1.16741359  0.17699948]
```

Lea Filtrando datos en línea: https://riptutorial.com/es/numpy/topic/6187/filtrando-datos

Capítulo 7: Generando datos aleatorios

Introducción

El módulo `random` de NumPy proporciona métodos convenientes para generar datos aleatorios que tienen la forma y distribución deseadas.

Aquí está la documentación oficial .

Examples

Creando una matriz aleatoria simple

```
# Generates 5 random numbers from a uniform distribution [0, 1)
np.random.rand(5)
# Out: array([ 0.4071833 ,  0.069167  ,  0.69742877,  0.45354268,  0.7220556 ])
```

Poniendo la semilla

Utilizando `random.seed` :

```
np.random.seed(0)
np.random.rand(5)
# Out: array([ 0.5488135 ,  0.71518937,  0.60276338,  0.54488318,  0.4236548 ])
```

Al crear un objeto generador de números aleatorios:

```
prng = np.random.RandomState(0)
prng.rand(5)
# Out: array([ 0.5488135 ,  0.71518937,  0.60276338,  0.54488318,  0.4236548 ])
```

Creando enteros aleatorios

```
# Creates a 5x5 random integer array ranging from 10 (inclusive) to 20 (inclusive)
np.random.randint(10, 20, (5, 5))

'''
Out: array([[12, 14, 17, 16, 18],
            [18, 11, 16, 17, 17],
            [18, 11, 15, 19, 18],
            [19, 14, 13, 10, 13],
            [15, 10, 12, 13, 18]])
'''
```

Seleccionando una muestra aleatoria de una matriz

```
letters = list('abcde')
```

Seleccione tres letras al azar (*con reemplazo* : el mismo elemento se puede elegir varias veces):

```
np.random.choice(letters, 3)
'''
Out: array(['e', 'e', 'd'],
      dtype='<U1')
'''
```

Muestreo sin reemplazo:

```
np.random.choice(letters, 3, replace=False)
'''
Out: array(['a', 'c', 'd'],
      dtype='<U1')
'''
```

Asignar probabilidad a cada letra:

```
# Choses 'a' with 40% chance, 'b' with 30% and the remaining ones with 10% each
np.random.choice(letters, size=10, p=[0.4, 0.3, 0.1, 0.1, 0.1])

'''
Out: array(['a', 'b', 'e', 'b', 'a', 'b', 'b', 'c', 'a', 'b'],
  dtype='<U1')
'''
```

Generando números aleatorios extraídos de distribuciones específicas.

Extraer muestras de una distribución normal (gaussiana).

```
# Generate 5 random numbers from a standard normal distribution
# (mean = 0, standard deviation = 1)
np.random.randn(5)
# Out: array([-0.84423086,  0.70564081, -0.39878617, -0.82719653, -0.4157447 ])

# This result can also be achieved with the more general np.random.normal
np.random.normal(0, 1, 5)
# Out: array([-0.84423086,  0.70564081, -0.39878617, -0.82719653, -0.4157447 ])

# Specify the distribution's parameters
# Generate 5 random numbers drawn from a normal distribution with mean=70, std=10
np.random.normal(70, 10, 5)
# Out: array([ 72.06498837,  65.43118674,  59.40024236,  76.14957316,  84.29660766])
```

Hay varias distribuciones adicionales disponibles en `numpy.random` , por ejemplo, `poisson` , `binomial` y `logistic`

```
np.random.poisson(2.5, 5)  # 5 numbers, lambda=5
# Out: array([0, 2, 4, 3, 5])

np.random.binomial(4, 0.3, 5)  # 5 numbers, n=4, p=0.3
# Out: array([1, 0, 2, 1, 0])

np.random.logistic(2.3, 1.2, 5)  # 5 numbers, location=2.3, scale=1.2
```

```
# Out: array([ 1.23471936,  2.28598718, -0.81045893,  2.2474899 ,  4.15836878])
```

Lea Generando datos aleatorios en línea: https://riptutorial.com/es/numpy/topic/2060/generando-datos-aleatorios

Capítulo 8: Indexación booleana

Examples

Creando un array booleano

Una matriz booleana se puede crear manualmente usando `dtype=bool` al crear la matriz. Los valores distintos de `0` , `None` , `False` o cadenas vacías se consideran verdaderos.

```
import numpy as np

bool_arr = np.array([1, 0.5, 0, None, 'a', '', True, False], dtype=bool)
print(bool_arr)
# output: [ True  True False False  True False  True False]
```

Alternativamente, numpy crea automáticamente una matriz booleana cuando se hacen comparaciones entre matrices y escalares o entre matrices de la misma forma.

```
arr_1 = np.random.randn(3, 3)
arr_2 = np.random.randn(3, 3)

bool_arr = arr_1 < 0.5
print(bool_arr.dtype)
# output: bool

bool_arr = arr_1 < arr_2
print(bool_arr.dtype)
# output: bool
```

Lea Indexación booleana en línea: https://riptutorial.com/es/numpy/topic/6072/indexacion-booleana

Capítulo 9: numpy.cross

Sintaxis

- `numpy.cross(a, b)` # producto cruzado de *a* y *b* (o vectores en *a* y *b*)
- `numpy.cross(a, b, axisa=-1)` # producto cruzado de vectores en *a* con *b* , los vectores st en a están dispuestos a lo largo del eje *axisa*
- `numpy.cross(a, b, axisa=-1, axisb=-1, axisc=-1)` # productos cruzados de vectores en *a* y *b* , vectores de salida dispuestos a lo largo del eje especificado por *axisc*
- `numpy.cross(a, b, axis=None)` # productos cruzados de vectores en *a* y *b* , vectores en *a* , *b* , y en salida distribuidos a lo largo del *eje*

Parámetros

Columna	Columna
a, b	En el uso más simple, `a` y `b` son dos vectores de 2 o 3 elementos. También pueden ser matrices de vectores (es decir, matrices bidimensionales). Si `a` es una matriz y 'b' es un vector, `cross(a,b)` devuelve una matriz cuyos elementos son los productos cruzados de cada vector en `a` con el vector `b` . La `b` es una matriz y `a` es un solo vector, `cross(a,b)` devuelve una matriz cuyos elementos son los productos cruzados de `a` con cada vector en `b` . `a` y `b` pueden ser ambos arrays si tienen la misma forma. En este caso, la `cross(a,b)` devuelve la `cross(a[0],b[0]), cross(a[1], b[1]), ...`
axisa / b	Si `a` es una matriz, puede tener vectores distribuidos en el eje de variación más rápida, el eje de variación más lenta o algo intermedio. `axisa` le dice a la `cross()` cómo los vectores están dispuestos en `a` . Por defecto, toma el valor del eje que varía más lentamente. `axisb` funciona igual con la entrada `b` . Si la salida de `cross()` va a ser una matriz, los vectores de salida pueden tener diferentes ejes de matriz; `axisc` le dice a la `cross` cómo colocar los vectores en su matriz de salida. Por defecto, `axisc` indica el eje de variación más lenta.
eje	Un parámetro de conveniencia que establece `axisa` , `axisb` y `axisc` en el mismo valor si se desea. Si el `axis` y cualquiera de los otros parámetros están presentes en la llamada, el valor del `axis` anulará los otros valores.

Examples

Producto cruzado de dos vectores

Numpy proporciona una función `cross` para la computación de productos cruzados vectoriales. El producto cruzado de los vectores `[1, 0, 0]` y `[0, 1, 0]` es `[0, 0, 1]` . Numpy nos dice:

```
>>> a = np.array([1, 0, 0])
>>> b = np.array([0, 1, 0])
>>> np.cross(a, b)
array([0, 0, 1])
```

como se esperaba.

Mientras que los productos cruzados se definen normalmente solo para vectores tridimensionales. Sin embargo, cualquiera de los argumentos de la función Numpy puede ser dos vectores de elementos. Si el vector `c` se da como `[c1, c2]` , Numpy asigna cero a la tercera dimensión: `[c1, c2, 0]` . Asi que,

```
>>> c = np.array([0, 2])
>>> np.cross(a, c)
array([0, 0, 2])
```

A diferencia del `dot` que existe tanto como una función Numpy como un método de `ndarray` , `cross` solo existe como una función independiente:

```
>>> a.cross(b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'numpy.ndarray' object has no attribute 'cross'
```

Productos cruzados múltiples con una llamada

Cualquiera de las entradas puede ser una matriz de vectores de 3 (o 2) elementos.

```
>>> a=np.array([[1,0,0],[0,1,0],[0,0,1]])
>>> b=np.array([1,0,0])
>>> np.cross(a,b)
array([[ 0,  0,  0],
       [ 0,  0, -1],
       [ 0,  1,  0]])
```

El resultado en este caso es array ([np.cross (a [0], b), np.cross (a [1], b), np.cross (a [2], b)])

`b` también puede ser una matriz de vectores de 3 (o 2) elementos, pero debe tener la misma forma que `a` . De lo contrario, el cálculo falla con un error de "desajuste de forma". Para que podamos tener

```
>>> b=np.array([[0,0,1],[1,0,0],[0,1,0]])
>>> np.cross(a,b)
array([[ 0, -1,  0],
       [ 0,  0, -1],
       [-1,  0,  0]])
```

y ahora el resultado es `array([np.cross(a[0],b[0]), np.cross(a[1],b[1]), np.cross(a[2],b[2])])`

Más flexibilidad con múltiples productos cruzados

En nuestros dos últimos ejemplos, numpy asumió que `a[0,:]` era el primer vector, `a[1,:]` el segundo, y `a[2,:]` el tercero. Numpy.cross tiene un argumento opcional axisa que nos permite especificar qué eje define los vectores. Asi que,

```
>>> a=np.array([[1,1,1],[0,1,0],[1,0,-1]])
>>> b=np.array([0,0,1])
>>> np.cross(a,b)
array([[ 1, -1,  0],
       [ 1,  0,  0],
       [ 0, -1,  0]])
>>> np.cross(a,b,axisa=0)
array([[ 0, -1,  0],
       [ 1, -1,  0],
       [ 0, -1,  0]])
>>> np.cross(a,b,axisa=1)
array([[ 1, -1,  0],
       [ 1,  0,  0],
       [ 0, -1,  0]])
```

El resultado de `axisa=1` y el resultado predeterminado son ambos `(np.cross([1,1,1],b), np.cross([0,1,0],b), np.cross([1,0,-1],b))` . De forma predeterminada, `axisa` siempre indica el último eje (el que varía más lentamente) de la matriz. El resultado de `axisa=0` es `(np.cross([1,0,1],b), np.cross([1,1,0],b), np.cross([1,0,-1],b))` .

Un parámetro opcional similar, `axisb` , realiza la misma función para la entrada `b` , si también es una matriz bidimensional.

Los parámetros axisa y axisb le dicen a los números cómo distribuir los datos de entrada. Un tercer parámetro, axisc le dice a Númpy cómo distribuir la salida si `a` o `b` es multidimensional. Usando las mismas entradas `a` y `b` que arriba, obtenemos

```
>>> np.cross(a,b,1)
array([[ 1, -1,  0],
       [ 1,  0,  0],
       [ 0, -1,  0]])
>>> np.cross(a,b,1,axisc=0)
array([[ 1,  1,  0],
       [-1,  0, -1],
       [ 0,  0,  0]])
>>> np.cross(a,b,1,axisc=1)
array([[ 1, -1,  0],
       [ 1,  0,  0],
       [ 0, -1,  0]])
```

Entonces, `axisc=1` y el `axisc` predeterminado dan el mismo resultado, es decir, los elementos de cada vector son contiguos en el índice de movimiento más rápido de la matriz de salida. Axisc es por defecto el último eje de la matriz. `axisc=0` distribuye los elementos de cada vector en la dimensión variable más lenta de la matriz.

Si desea que `axisa` , `axisb` y `axisc` tengan el mismo valor, no necesita configurar los tres parámetros. Puede configurar un cuarto parámetro, `axis` , al valor único necesario y los otros tres parámetros se establecerán automáticamente. axis reemplaza a axisa, axisb o axisc si alguno de ellos está presente en la llamada de función.

Lea numpy.cross en línea: https://riptutorial.com/es/numpy/topic/6166/numpy-cross

Capítulo 10: numpy.dot

Sintaxis

- numpy.dot (a, b, out = None)

Parámetros

Nombre	Detalles
una	una matriz numpy
segundo	una matriz numpy
afuera	una matriz numpy

Observaciones

numpy.dot

Devuelve el producto punto de a y b. Si a y b son ambos escalares o ambas matrices 1-D, se devuelve un escalar; de lo contrario se devuelve una matriz. Si se da fuera, entonces se devuelve.

Examples

Multiplicación de matrices

La multiplicación de matrices se puede hacer de dos maneras equivalentes con la función de punto. Una forma es usar la función miembro de punto de numpy.ndarray.

```
>>> import numpy as np
>>> A = np.ones((4,4))
>>> A
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
>>> B = np.ones((4,2))
>>> B
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]])
>>> A.dot(B)
array([[ 4.,  4.],
       [ 4.,  4.],
       [ 4.,  4.],
       [ 4.,  4.]])
```

La segunda forma de hacer la multiplicación de matrices es con la función de biblioteca numpy.

```
>>> np.dot(A,B)
array([[ 4.,  4.],
       [ 4.,  4.],
       [ 4.,  4.],
       [ 4.,  4.]])
```

Productos vectoriales punto

La función de puntos también se puede usar para calcular productos de puntos vectoriales entre dos matrices numpy unidimensionales.

```
>>> v = np.array([1,2])
>>> w = np.array([1,2])
>>> v.dot(w)
5
>>> np.dot(w,v)
5
>>> np.dot(v,w)
5
```

El parametro de salida

La función de punto numpy tiene un parámetro opcional out = None. Este parámetro le permite especificar una matriz para escribir el resultado. Esta matriz debe ser exactamente la misma forma y tipo que la matriz que se habría devuelto, o se lanzará una excepción.

```
>>> I = np.eye(2)
>>> I
array([[ 1.,  0.],
       [ 0.,  1.]])
>>> result = np.zeros((2,2))
>>> result
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> np.dot(I, I, out=result)
array([[ 1.,  0.],
       [ 0.,  1.]])
>>> result
array([[ 1.,  0.],
       [ 0.,  1.]])
```

Intentemos cambiar el dtype del resultado a int.

```
>>> np.dot(I, I, out=result)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: output array is not acceptable (must have the right type, nr dimensions, and be a
C-Array)
```

Y si intentamos usar un orden de memoria subyacente diferente, digamos estilo Fortran (por lo que las columnas son contiguas en lugar de filas), también se produce un error.

```
>>> result = np.zeros((2,2), order='F')
>>> np.dot(I, I, out=result)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: output array is not acceptable (must have the right type, nr dimensions, and be a
C-Array)
```

Operaciones matriciales sobre matrices de vectores.

numpy.dot se puede utilizar para multiplicar una lista de vectores por una matriz, pero la orientación de los vectores debe ser vertical, de modo que una lista de ocho vectores de dos componentes aparezca como dos vectores de ocho componentes:

```
>>> a
array([[ 1.,  2.],
       [ 3.,  1.]])
>>> b
array([[  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.],
       [  9.,  10.,  11.,  12.,  13.,  14.,  15.,  16.]])
>>> np.dot(a, b)
array([[ 19.,  22.,  25.,  28.,  31.,  34.,  37.,  40.],
       [ 12.,  16.,  20.,  24.,  28.,  32.,  36.,  40.]])
```

Si la lista de vectores se presenta con sus ejes al revés (lo que suele ser el caso), la matriz debe transponerse antes y después de la operación de puntos como esta:

```
>>> b
array([[  1.,   9.],
       [  2.,  10.],
       [  3.,  11.],
       [  4.,  12.],
       [  5.,  13.],
       [  6.,  14.],
       [  7.,  15.],
       [  8.,  16.]])
>>> np.dot(a, b.T).T
array([[ 19.,  12.],
       [ 22.,  16.],
       [ 25.,  20.],
       [ 28.,  24.],
       [ 31.,  28.],
       [ 34.,  32.],
       [ 37.,  36.],
       [ 40.,  40.]])
```

Aunque la función de punto es muy rápida, a veces es mejor usar einsum. El equivalente de lo anterior sería:

```
>>> np.einsum('...ij,...j', a, b)
```

Es un poco más lento pero permite multiplicar una lista de vértices por una lista correspondiente de matrices. Este sería un proceso muy complicado usando el punto:

```
>>> a
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]],
       [[24, 25],
        [26, 27]],
       [[28, 29],
        [30, 31]]])
>>> np.einsum('...ij,...j', a, b)
array([[   9.,   29.],
       [  58.,   82.],
       [ 123.,  151.],
       [ 204.,  236.],
       [ 301.,  337.],
       [ 414.,  454.],
       [ 543.,  587.],
       [ 688.,  736.]])
```

numpy.dot se puede usar para encontrar el producto de puntos de cada vector en una lista con un vector correspondiente en otra lista. Esto es bastante complicado y lento en comparación con la multiplicación y suma de elementos a lo largo del último eje. Algo como esto (que requiere que se calcule una matriz mucho más grande pero que se ignore en su mayoría)

```
>>> np.diag(np.dot(b,b.T))
array([  82.,  104.,  130.,  160.,  194.,  232.,  274.,  320.])
```

Producto de puntos utilizando la multiplicación y la suma de elementos.

```
>>> (b * b).sum(axis=-1)
array([  82.,  104.,  130.,  160.,  194.,  232.,  274.,  320.])
```

Usando einsum se podría lograr con

```
>>> np.einsum('...j,...j', b, b)
array([  82.,  104.,  130.,  160.,  194.,  232.,  274.,  320.])
```

Lea numpy.dot en línea: https://riptutorial.com/es/numpy/topic/3198/numpy-dot

Capítulo 11: Regresión lineal simple

Introducción

Ajustar una línea (u otra función) a un conjunto de puntos de datos.

Examples

Utilizando np.polyfit

Creamos un conjunto de datos que luego ajustamos con una línea recta $ f (x) = mx + c $.

```
npoints = 20
slope = 2
offset = 3
x = np.arange(npoints)
y = slope * x + offset + np.random.normal(size=npoints)
p = np.polyfit(x,y,1)           # Last argument is degree of polynomial
```

Para ver lo que hemos hecho:

```
import matplotlib.pyplot as plt
f = np.poly1d(p)                # So we can call f(x)
fig = plt.figure()
ax  = fig.add_subplot(111)
plt.plot(x, y, 'bo', label="Data")
plt.plot(x,f(x), 'b-',label="Polyfit")
plt.show()
```

Nota: este ejemplo sigue la documentación numpy en https://docs.scipy.org/doc/numpy/reference/generated/numpy.polyfit.html muy de cerca.

Usando np.linalg.lstsq

Usamos el mismo conjunto de datos que con polyfit:

```
npoints = 20
slope = 2
offset = 3
x = np.arange(npoints)
y = slope * x + offset + np.random.normal(size=npoints)
```

Ahora, intentamos encontrar una solución minimizando el sistema de ecuaciones lineales A b = c minimizando | cA b | ** 2

```
import matplotlib.pyplot as plt # So we can plot the resulting fit
A = np.vstack([x,np.ones(npoints)]).T
m, c = np.linalg.lstsq(A, y)[0] # Don't care about residuals right now
fig = plt.figure()
```

```
ax  = fig.add_subplot(111)
plt.plot(x, y, 'bo', label="Data")
plt.plot(x, m*x+c, 'r--',label="Least Squares")
plt.show()
```

Nota: este ejemplo sigue la documentación numpy en https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.lstsq.html muy de cerca.

Lea Regresión lineal simple en línea: https://riptutorial.com/es/numpy/topic/8808/regresion-lineal-simple

Capítulo 12: subclasificando ndarray

Sintaxis

- `def __array_prepare__(self, out_arr: ndarray, context: Tuple[ufunc, Tuple, int] = None) -> ndarray: # called on the way into a ufunc`
- `def __array_wrap__(self, out_arr: ndarray, context: Tuple[ufunc, Tuple, int] = None) -> ndarray: # called on the way out of a ufunc`
- `__array_priority__: int # used to determine which argument to invoke the above methods on when a ufunc is called`
- `def __array_finalize__(self, obj: ndarray): # called whenever a new instance of this class comes into existence, even if this happens by routes other than __new__`

Examples

Seguimiento de una propiedad extra en matrices

```
class MySubClass(np.ndarray):
    def __new__(cls, input_array, info=None):
        obj = np.asarray(input_array).view(cls)
        obj.info = info
        return obj

    def __array_finalize__(self, obj):
        # handles MySubClass(...)
        if obj is None:
            pass

        # handles my_subclass[...] or my_subclass.view(MySubClass) or ufunc output
        elif isinstance(obj, MySubClass):
            self.info = obj.info

        # handles my_arr.view(MySubClass)
        else:
            self.info = None

    def __array_prepare__(self, out_arr, context=None):
        # called before a ufunc runs
        if context is not None:
            func, args, which_return_val = context

        return super().__array_prepare__(out_arr, context)

    def __array_wrap__(self, out_arr, context=None):
        # called after a ufunc runs
        if context is not None:
            func, args, which_return_val = context

        return super().__array_wrap__(out_arr, context)
```

Para la tupla de `context` , `func` es un objeto ufunc como `np.add` , `args` es una `tuple` y `which_return_val` es un entero que especifica qué valor de retorno del ufunc se está procesando

Lea subclasificando ndarray en línea: https://riptutorial.com/es/numpy/topic/6431/subclasificando-ndarray

Creditos

S. No	Capítulos	Contributors
1	Empezando con numpy	Andras Deak, Chris Mueller, Code-Ninja, Community, Dux, edwinksl, grovduck, Hammer, hashcode55, Joshua Cook, karel, Kersten, Mpaull, prasastoadi, rlee827, sebix, user2314737, Yassie
2	Ahorro y carga de matrices	obachtos
3	Álgebra lineal con np.linalg	Daniel, DataSwede, Fermi paradox, Mahdi, Sean Easter
4	Archivo IO con numpy	Alex, atomh33ls, Sparkler
5	Arrays	Andras Deak, ayhan, B Samedi, B8vrede, Benjamin, DataSwede, Dux, Gwen, Hamlet, Hammer, KARANJ, Keith L, pixatlazaki, Ryan, Sean Easter, Sparkler, The Hagen, TPVasconcelos, user2314737
6	Filtrando datos	Alex, farleytpm
7	Generando datos aleatorios	amin, ayhan, B8vrede, Dux, Fermi paradox, user2314737
8	Indexación booleana	Chris Mueller
9	numpy.cross	bob.sacamento, Mad Physicist
10	numpy.dot	bpachev, paddyg, Shubham Dang
11	Regresión lineal simple	Alex
12	subclasificando ndarray	Eric