Python 数据分析实践（Data Analysis Action） ¶

Chap 3 Python 数据分析和挖掘工具库

内容：

- Python 数据分析与数据挖掘相关库

实践：

- 库的导入和添加
- NumPy
- Pandas
- SciPy
- Matplotlib
- StatsModels
- Scikit（即 Sklearn 或 Scikit-learn）
- Keras
- Gensim
- NLTK

0. 库的导入和添加

库的导入模式有如下几种：

```
import math  # 采用 math.sin() 方式引用
import math as m  # m.sin() 引用
from math import exp as e  # 如果不需要导入 math 库中的其他函数，只导入 math 库中的 exp 函数，e(1) 计算指数
from math import *  # 直接的导入，去掉 math， exp(1)，容易引起命名冲突
```

建议用法：

对于导入库起个常用的别名

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import math as m
```

在 Python 软件开发过程中，：

- 不建议直接引入类似 NumPy 这种大型库的全部内容（即，不建议 `from numpy import *`），如果大量引入第三方库，容易引起命名冲突。

当看到 np.arange 时，就应该想到它引用的是 NumPy 中的 arange 函数。

In [1]:

```
# 必要准备工作：导入库，配置环境等
# 导入python特征（为Python2）
# 2版本中3/2=1， 3版本中3/2=1.5， 3//2=1
from __future__ import division

# 导入库并为库起个别名
import numpy as np
import pandas as pd
np.set_printoptions(precision=4, suppress=True)

# 启动绘图
%matplotlib inline
import matplotlib.pyplot as plt
```

In [2]:

```
# 运行代码时会有很多warning输出，如提醒新版本之类的
import warnings
warnings.filterwarnings('ignore','ConvergenceWarning')
```

Python 数据分析和数据挖掘中常使用到的相关扩展库

扩展库	简介
Numpy	提供数组支持，以及相应的高效处理函数
Scipy	提供矩阵支持，以及矩阵相关的数值计算模块
Matplotlib	强大的数据可视化工具、作图库
Pandas	强大、灵活的数据分析和探索工具
StatsModels	统计建模和计量经济学，包括描述统计、统计模型估计和推断
Scikit-learn	支持回归、分类、聚类等的强大的机器学习库
NLTK	自然语言处理相关的语料和技术支持库
Gensim	文本主题模型LDA、文本相似度计算、Google的Word2Vec等的库

此外，Keras（深度学习库，建立神经网络模型以及深度学习模型），视频处理的 OpenSC 等等。

练习： 进行库的安装和更新

检查是否安装 StatsModels 库

注意：下面命令，如果库没有安装，则会进行下载安装，后台需要花时间

```
!pip install statsmodels
```

Python 中的 list 列表对象与 Numpy 的 ndarray 类型

Python 没有提供数组功能，虽然列表可以完成基本的数组功能，但数据量较大时，列表的速度就会慢得难以接受。

NumPy 提供了真正的数组功能，以及对数组进行快速处理的函数。而且NumPy还是很多高级的扩展库的依赖库，其他 SciPy、Matplotlib、Pandas 等库都依赖于它。NumPy 内置函数处理数据的速度都是C语言级别的，因此，尽量使用内置函数。

1. NumPy 的 ndarray: 一种多维数组对象 (a multidimensional array object)

NumPy，是 Numerical Python 的简称，是 Python 中高性能科学计算和数据分析的基础包。是数据分析中几乎所有高级工具的构建基础。

NumPy 最重要的一个特点就是其N维数组对象（即 ndarray），该对象是一个快速而灵活的大数据集容器。可以利用这种数组对整块数据执行一些数学运算，其语法跟标量元素之间的运算一样。 ndarray 是一个通用的同构数据多维容器，也就是说，其中的所有元素必须是相同类型的。每个数组都有一个 shape（一个表示各维度大小的元组）和一个 dtype（一个用于说明数组数据类型的对象）

对于数值型数据，NumPy 数组在存储和处理数据时要比内置的 Python 数据结构高效得多。其部分功能如下：

- ndarray，一个具有矢量算术运算和复杂广播能力的快速高效且节省空间的多维数组
- 用于对数组执行快速元素级计算的标准数学函数（无需编写循环）
- 用于读写硬盘上基于数组的数据的工具以及用于操作内存映射文件的工具
- 线性代数运算、傅立叶变换，以及随机数生成等功能
- 用于将 C、C++、Fortran 代码集成到 Python 的工具（由 C 和 Fortran 编写的库可以直接操作 NumPy 数组中的数据，无需进行数据复制工作）
- 作为在算法之间传递数据的容器

NumPy 本身并没有提供多么高级的数据分析功能，理解 NumPy 数组以及面向数组的计算将有助于更加高效的使用诸如 pandas 之类的工具。

对于大部分数据分析应用而言，我们最关注的功能主要集中在：

- 用于数据整理和清理、子集构造和过滤、转换等快速的矢量化数组运算
- 常用的数组算法，如排序、唯一化、集合运算等
- 高效的描述统计和数据聚合/摘要运算
- 用于异构数据集的合并/连接运算的数据对齐和关系型数据运算
- 将条件逻辑表述为数组表达式（而不是带有 `if-elif-else` 分支的循环）
- 数组的分组运算（聚合、转换、函数应用等）

主要介绍 NumPy 数组的基本用法。精通面向数组的编程和思维方式是成为 Python 数据分析达人的关键步骤。

注意： 按照标准的 NumPy 约定，我们总是使用 `import numpy as np` ， 尽量不要使用 `from numpy import *`

In [3]:

```
# 导入numpy库，给个别名，查看numpy库的版本号
import numpy as np
np.__version__
```

Out[3]:

'1.18.1'

1) 创建 ndarrays （Creating ndarrays）

创建数组最简单的办法一是使用 array 函数。它接受一切序列型的对象（包括其他数组），然后产生一个新的含有传入数据的 NumPy 数组。

- Python 列表数据被 `np.array` 转换为一个 NumPy 数组
- 嵌套序列（有一组等长列表组成的列表）将会被转换为一个多维数组

除法显示说明，np.array 会尝试给新建的这个数组推断出一个较为合适的数据类型，数据类型保存在一个特殊的 dtype 对象中。

除了 np.array 之外， 还有一些函数可以新创建数组。比如：

- `zeros` 和 `ones` 分别可以创建指定长度或形状的全 0 或全 1 数组。
- `empty` 可以创建一个没有任何具体值的数组; **注意：empty 并不是返回全 0 数组。通常返回的都是一些未初始化的垃圾值。**
- `arange` 是 Python 内置函数 range 的数组版。 **注意：** 要用这些方法创建多维数组，只需传入一个表示形状的元组即可。

In [4]:

```
#Python标准的列表
a = [6, 7, 8, 0, 1]
print(type(a))
a
```

```
<class 'list'>
```

Out[4]:

```
[6, 7, 8, 0, 1]
```

In [5]:

```
# NumPy的array函数将列表转换为NumPy数组
arr1 = np.array(a)
print(type(arr1))
arr1
```

```
<class 'numpy.ndarray'>
```

Out[5]:

```
array([6, 7, 8, 0, 1])
```

In [6]:

```
# 嵌套序列（等长列表组成的列表）将会被转换为一个多维数组
# 创建多维数组
b = [[1, 2, 3, 4], [5, 6, 7, 8]]
arr2 = np.array(b)
print(type(arr2))
arr2
```

```
<class 'numpy.ndarray'>
```

Out[6]:

```
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

In [7]:

```
# 多维数组的维度
arr2.ndim
```

Out[7]:

```
2
```

In [8]:

```
# 多维数组的形状shape
arr2.shape
```

Out[8]:

```
(2, 4)
```

In [9]:

```
# 数组类型
print(arr1.dtype)
print(arr2.dtype)
```

```
int32
int32
```

In [10]:

```
# 创建全0或全1的一维数组，需要输入shape
np.zeros(5)
```

Out[10]:

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

In [11]:

```
# 创建全0或1的多维数组
np.ones((3, 4))
```

Out[11]:

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

In [12]:

```
# empty创建一个没有任何具体值的数组;并不是返回全0数组。通常返回的都是一些未初始化的垃圾值。
np.empty((3,2))
```

Out[12]:

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

In [13]:

```
# arange是Python内置函数range的数组版
a = list(range(5))
print(type(a), a)
b = np.array(a)
print(type(b), b)
c = np.arange(5)
print(type(c), c)
```

```
<class 'list'> [0, 1, 2, 3, 4]
<class 'numpy.ndarray'> [0 1 2 3 4]
<class 'numpy.ndarray'> [0 1 2 3 4]
```

In [14]:

```
d = np.zeros(5)
print(type(d), d)
e= np.ones(5)
print(type(e), e)
```

```
<class 'numpy.ndarray'> [0. 0. 0. 0. 0.]
<class 'numpy.ndarray'> [1. 1. 1. 1. 1.]
```

In [15]:

```
# 将多个ndarray数组进行合并
d = np.array((a,b,c))
print(type(d))
d
```

```
<class 'numpy.ndarray'>
```

Out[15]:

```
array([[0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4]])
```

In [16]:

```
np.eye(3)
```

Out[16]:

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

下表列出了数组创建函数

由于 NumPy 关注的是数值计算，如果没有特别指定，数据类型基本都是 float64（浮点数）

函数	说明
array	将输入数据（列表，元组，数组或其他序列类型）转换为ndarray。要么推断出dtype，要么显示指定dtype。默认直接复制数据
asarray	将输入转换为ndarray，如果输入本身就是ndarray就不进行复制
arange	类似于内置的range，但返回的是一个ndarray而不是列表
ones、ones_like	根据指定的形状和dtype创建一个全1数据。ones_like以另一个数组为参数，并根据其形状和dtype创建一个全1数组
zeros、zeros_like	类似于ones和ones_like，只不过产生的是全0数组而已
empty、empty_line	创建新数组，只分配内存空间但不填充任何值
eye、identity	创建一个正方的 $N * N$ 单位矩阵（对角线为1， 其余为0）

2) ndarray 的数据类型（Data Types for ndarrays）

dtype（数据类型）是一个特殊的对象，它含有 ndarray 将一块内存解释为特定数据类型所需的信息。

dtype 是 NumPy 如此强大和灵活的原因之一。多数情况下，它们直接映射到相应的机器表示，使得“读写磁盘上的二进制数据流”和“集成低级语言代码（如 C、Fortran ）”等工作变得更加简单。

数值型 dtype 的命名方式相同：一个类型名（如 float 或 int ），后面跟一个用于表示各元素位长的数字。标准的双精度浮点值（即 Python 中的 float 对象）需要占用 8 字节（即 64 位）。因此，该类型在 NumPy 中就记作 float64。

可以通过 ndarray 的 astype 方法显示的转换 dtype。

注意： 调用 astype 如论如何都会创建出一个新的数组（原始数据的一份拷贝），即使新的 dtype 跟老的 dtype 相同也是如此。

注意：浮点数（比如 float64 和 float32）只能表示近似的分数值。在复杂计算中，由于可能会积累一些浮点错误，因此比较操作只能在一定小数位以内有效。

下面的表列出了 NumPy 所支持的全部数据类型。

类型	类型代码	说明
int8、uint8	i1, u1	有符号和无符号的8位（1个字节）整型
int16、uint16	i2, u2	有符号和无符号的16位（2个字节）整型
int32、uint32	i4, u4	有符号和无符号的32位（4个字节）整型
int64、uint64	i8, u8	有符号和无符号的64位（8个字节）整型
float16	f2	半精度浮点数
float32	f4或f	标准的单精度浮点数。与C的float兼容
float64	f8或d	标准的双精度浮点数。与C的double和Python的float对象兼容
float128	f16或g	扩展精度浮点数
complex64、complex128 、complex256	c8、c16、c32	分别用两个32位、64位或128位浮点数表示的复数
bool	?	存储True和False值的布尔类型
object	O	Python对象类型
string_	S	固定长度的字符串类型（每个字符1个字节）。例如，要创建一个长度为10的字符串，应使用S10
unicode_	U	固定长度的unicode类型（字节数由平台决定）。跟字符串的定义方式一样（如U10）

In [17]:

```
# 创建数组时，指定数组类型
arr1 = np.array([1, 2, 3], dtype=np.float64)
arr2 = np.array([1, 2, 3], dtype=np.int32)
print(arr1.dtype, arr2.dtype)
```

```
float64 int32
```

In [18]:

```
# 可以通过ndarray的astype方法显示的转换dtype，整数转成浮点数
arr = np.array([1, 2, 3, 4, 5])
print(arr.dtype, arr)

float_arr = arr.astype(np.float64)
print(float_arr.dtype,float_arr)
```

```
int32 [1 2 3 4 5]
float64 [1. 2. 3. 4. 5.]
```

In [19]:

```
# 浮点数转成整数，则小数部分直接被截断
arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
arr
arr.astype(np.int32)
```

Out[19]:

```
array([ 3, -1, -2,  0, 12, 10])
```

In [20]:

```
# 字符串类型如果都是数字，也可以转换为数值形式
numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)
numeric_strings.astype(float)
```

Out[20]:

```
array([ 1.25, -9.6 , 42.  ])
```

In [21]:

```
# 把一个数组的dtype转换为另外一个数组的数据类型 A.astype(B.dtype),将B的数据类型赋予A
int_array = np.arange(10)
print(int_array.dtype, int_array)
calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)
int_array.astype(calibers.dtype)
```

```
int32 [0 1 2 3 4 5 6 7 8 9]
```

Out[21]:

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

In [22]:

```
# 可以使用简洁的类型代码来表示dtype
empty_uint32 = np.empty(8, dtype='u4')
empty_uint32
```

Out[22]:

```
array([ 822093296,         670, 2407204976,       32760, 2206363486,
        814232219,  822093360,         670], dtype=uint32)
```

3) 数组和标量之间的运算

（Operations between arrays and scalars）

数组很重要，可以使我们不用编写循环即可对数据执行批量运算。这通常叫做**矢量化（vectorization）**。

- 大小相等的数组之间的任何算术运算都将运算应用到元素级，加减乘除和方乘
- 数组与标量的算术运算也会将那个标量值传播到各个元素。
- 不同大小的数组之间的运算叫做**广播（broadcasting）**

In [23]:

```
# 等大小数组的算术运算
arr = np.array([[1., 2., 3.], [4., 5., 6.]])
print('arr: ', arr)
print('sum: ', arr + arr)
print('multiplication: ', arr * arr)
print('difference: ', arr - arr)
print('inverse: ', 1 / arr)
print('squared root:', arr ** 0.5)
```

```
arr:  [[1. 2. 3.]
 [4. 5. 6.]]
sum:  [[ 2.  4.  6.]
 [ 8. 10. 12.]]
multiplication:  [[ 1.  4.  9.]
 [16. 25. 36.]]
difference:  [[0. 0. 0.]
 [0. 0. 0.]]
inverse:  [[1.     0.5    0.3333]
 [0.25   0.2    0.1667]]
squared root: [[1.     1.4142 1.7321]
 [2.     2.2361 2.4495]]
```

4） 基本的索引和切片（Basic indexing and slicing）

NumPy 数组的索引是一个内容丰富的主题，因为选取数据子集或单个元素的方式有很多。

一维数组很简单，从表面看，和 Python 列表的功能差不多。将一个标量值赋值给一个切片，该值会自动传播（即广播）到整个选区。但是跟列表最重要的区别是， 数组切片是原始数组的视图。这意味着数据不会被复制，视图上的任何修改都会直接反映到源数组上。这是因为 NumPy 的设计目的是处理大数据，可以想象一下，如果坚持将数据复制的话会产生何等的性能和内存问题。 **注意：** 如果要想得到的是 ndarray 切片的一份副本而不是视图，就需要显示地进行复制操作。 `arr[5:8].copy()`

对于高维数组，各索引位置上的元素不再是标量，而是数组。可以传入一个以逗号隔开的索引列表来选取单个元素，例如 `arr[0][2] = arr[0,2]` 这两个方式等价，都是取数组中第一个维度的第三个位置的元素。在多维数组中，如果省略了后面的索引，则返回对象是一个维度低一些的 ndarray（它含有高一级维度上的所有数据）。

In [24]:

```
arr = np.arange(10)
print(arr)
print(arr[5])
print(arr[5:8])

# 将一个标量值赋值给一个切片，该值会自动传播（即广播）到整个选区。
arr[5:8] = 12
print(arr)
```

```
[0 1 2 3 4 5 6 7 8 9]
5
[5 6 7]
[ 0  1  2  3  4 12 12 12  8  9]
```

In [25]:

```
# 数组切片操作直接修改源数组
arr_slice = arr[5:8]
arr_slice[1] = 12345
print(arr)

arr_slice[:] = 64
print(arr)
```

```
[    0     1     2     3     4    12 12345    12     8     9]
[ 0  1  2  3  4 64 64 64  8  9]
```

In [26]:

```
# 对于高维数组，各索引位置上的元素不再是标量，而是一个数组
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
arr2d[2]
```

Out[26]:

```
array([7, 8, 9])
```

In [27]:

```
# 对于高维数组，对索引递归可以对元素进行访问。下面两个方式是等价访问数组中的一个元素
print(arr2d[0][2])
print(arr2d[0, 2])
```

```
3
3
```

In [28]:

```
arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
arr3d
```

Out[28]:

```
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]]])
```

In [29]:

```
arr3d[0]
```

Out[29]:

```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [30]:

```
old_values = arr3d[0].copy()
# 标量值和数组都可以被赋值给ndarray
arr3d[0] = 42
print(arr3d)
print()

arr3d[0] = old_values
print(arr3d)
```

```
[[[42 42 42]
  [42 42 42]]

 [[ 7  8  9]
  [10 11 12]]]

[[[ 1  2  3]
  [ 4  5  6]]

 [[ 7  8  9]
  [10 11 12]]]
```

In [31]:

```
# 在多维数组中，如果省略了后面的索引，则返回对象是一个维度低一些的ndarray
#（它含有高一级维度上的所有数据）。
arr3d[1, 0]
```

Out[31]:

```
array([7, 8, 9])
```

5) 数学和统计方法（ Mathematical and statistical methods）

可以通过数组上的一组数学函数对整个数组或某个轴向的数据进行统计计算。

sum、mean 以及标准差 std 等聚合计算即可以当做数组的实例方法调用（`arr.mean()`），也可以当做顶级 NumPy 函数使用（`np.mean(arr)`）。

- `arr.mean()` 与 `np.mean(arr)` 相等，都是返回 arr 数组的均值 mean。
- `arr.sum()` 与 `np.sum(arr)` 相等，都是返回 arr 数组的加和 sum。
- `arr.std()` 与 `np.std(arr)` 相等，都是返回 arr 数组的标准差 std。

mean 和 sum 这类函数可以接受一个 axis 参数（用于计算该轴向上的统计值），最终返回结果是一个 *n-1* 维的数组（即少一维的数组）

下面的表列出全部的基本数组统计方法。

下表列出了基本数组统计方法

方法	说明
sum	对数组中全部或某轴向的元素求和。零长度的数组的sum为0
mean	算术平均数。零长度的数组的mean为NaN
std、var	分别为标准差和方差，自由度可调（默认为n）
min、max	最大值和最小值
argmin、argmax	分别为最大和最小元素的索引
cumsum	所有元素的累积和
cumprod	所有元素的累积乘积

In [32]:

```
# 生成正态分布的二维数组5*4
arr = np.random.randn(5, 4) # normally-distributed data
print(arr)
print(arr.mean())
print(np.mean(arr))
print(arr.sum(), np.sum(arr))
print(arr.std(), np.std(arr))
print(arr.argmin(),arr.argmax())
```

```
[[ 2.826    0.8176   0.1587   1.3734]
 [-0.1232 -0.3234   1.4672 -0.9664]
 [-0.4874 -0.0723   0.0897   1.0692]
 [-0.4506 -0.4653 -0.2514 -0.4003]
 [ 0.3753   1.5912 -0.5697   1.7626]]
0.37104328167618766
0.37104328167618766
7.4208656335237535 7.4208656335237535
0.9787796056354319 0.9787796056354319
7 0
```

In [33]:

```
# mean可以接受一个axis参数（用于计算该轴向上的统计值），返回结果是n-1维的数组
# 在axis=0的轴上求均值，即合并第一个维度上的值，原先是5*4的二维，返回是4的一维
print(arr[0])  # 第一行数据
print(arr[:,0])  # 第一列数据

# 合并第一个维度的加和
print(arr.sum(0))

# 求第一个维度的均值
print(arr.mean(axis=0))

# 在第二个维度上求加和
print(arr.sum(1))
```

```
[2.826  0.8176 0.1587 1.3734]
[ 2.826  -0.1232 -0.4874 -0.4506  0.3753]
[2.1401 1.5477 0.8945 2.8386]
[0.428  0.3095 0.1789 0.5677]
[ 5.1756  0.0542  0.5992 -1.5676  3.1594]
```

In [34]:

```
# 其他如cumsum和cumprod之类的方法则不聚合，而是产生一个由中间结果组成的数组。
arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
print(arr)

# 在第一个维度上的元素累积和，cumsum--所有元素的累积和
print(arr.cumsum(0))

# 在第二个维度上的元素累积乘积，cumprod--所有元素的累积乘积
print(arr.cumprod(1))
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
[[ 0  1  2]
 [ 3  5  7]
 [ 9 12 15]]
[[  0   0   0]
 [  3  12  60]
 [  6  42 336]]
```

2. SciPy

两个 NumPy 数组相乘时，只是对应元素相乘，并不是矩阵乘法。

SciPy 提供了真正的矩阵，以及大量基于矩阵运算的对象和函数。

In [35]:

```
# NumPy只是对应元素相乘
a = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
print(a)
print(a * a)
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
[[ 0  1  4]
 [ 9 16 25]
 [36 49 64]]
```

In [36]:

```
# 导入Scipy库
import scipy
from scipy import linalg
```

In [37]:

```
a * a
```

Out[37]:

```
array([[ 0,  1,  4],
       [ 9, 16, 25],
       [36, 49, 64]])
```

In [38]:

```
b = scipy.mat(a)
b * b
```

```
C:\Anaconda3\lib\site-packages\ipykernel_launcher.py:1: DeprecationWarning: scipy.ma
t is deprecated and will be removed in SciPy 2.0.0, use numpy.mat instead
  """Entry point for launching an IPython kernel.
```

Out[38]:

```
matrix([[ 15,  18,  21],
        [ 42,  54,  66],
        [ 69,  90, 111]])
```

In [39]:

```
# 真正的矩阵乘法
scipy.mat(a) * scipy.mat(a)
```

```
C:\Anaconda3\lib\site-packages\ipykernel_launcher.py:2: DeprecationWarning: scipy.ma
t is deprecated and will be removed in SciPy 2.0.0, use numpy.mat instead
```

Out[39]:

```
matrix([[ 15,  18,  21],
        [ 42,  54,  66],
        [ 69,  90, 111]])
```

In [40]:

```
# 将NumPy的ndarray类型转换为矩阵
ma=scipy.mat(a)
print(type(ma))
print(ma)
```

```
<class 'numpy.matrix'>
[[0 1 2]
 [3 4 5]
 [6 7 8]]

C:\Anaconda3\lib\site-packages\ipykernel_launcher.py:2: DeprecationWarning: scipy.ma
t is deprecated and will be removed in SciPy 2.0.0, use numpy.mat instead
```

In [41]:

```
# 1.逆矩阵的求解
a = np.array([[1, 2, 3], [4, 5, 6], [0., 0., 1]])
ma = scipy.mat(a) * scipy.mat(a)
mb = linalg.inv(ma)
print(mb)
```

```
[[ 3.6667 -1.3333 -2.    ]
 [-2.6667  1.      0.    ]
 [ 0.      0.      1.    ]]

C:\Anaconda3\lib\site-packages\ipykernel_launcher.py:3: DeprecationWarning: scipy.ma
t is deprecated and will be removed in SciPy 2.0.0, use numpy.mat instead
  This is separate from the ipykernel package so we can avoid doing imports until
```

In [42]:

```
ma * mb
```

Out[42]:

```
matrix([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]])
```

In [43]:

```
# 2.求方阵的行列式值
linalg.det(ma)
```

Out[43]:

```
9.000000000000016
```

In [44]:

```
# 3.求ma的模，矩阵的模是矩阵中每个元素的平方和再开方
linalg.norm(ma)
```

Out[44]:

67.22350779303324

In [45]:

```
# 4.求特征值及特征向量
r,v = linalg.eig(ma)
print("Root: ", r)
print("Vector: ", v)
```

```
Root:  [ 0.2154+0.j 41.7846+0.j  1.    +0.j]
Vector:  [[-0.8069 -0.3437  0.    ]
 [ 0.5907 -0.9391 -0.8321]
 [ 0.      0.      0.5547]]
```

In [46]:

```
# 5.LU矩阵分解
x,y,z=linalg.lu(ma)
print(x)
print('L矩阵： ', y)
print('U矩阵： ', z)
```

```
[[0. 1. 0.]
 [1. 0. 0.]
 [0. 0. 1.]]
L矩阵：  [[ 1.     0.     0.   ]
 [ 0.375  1.     0.   ]
 [ 0.    -0.     1.   ]]
U矩阵：  [[24.    33.    48.   ]
 [ 0.    -0.375  0.   ]
 [ 0.     0.     1.   ]]
```

3. Matplotlib

Python 中最著名的绘图库。主要用于二维绘图，用于数据的可视化

In [47]:

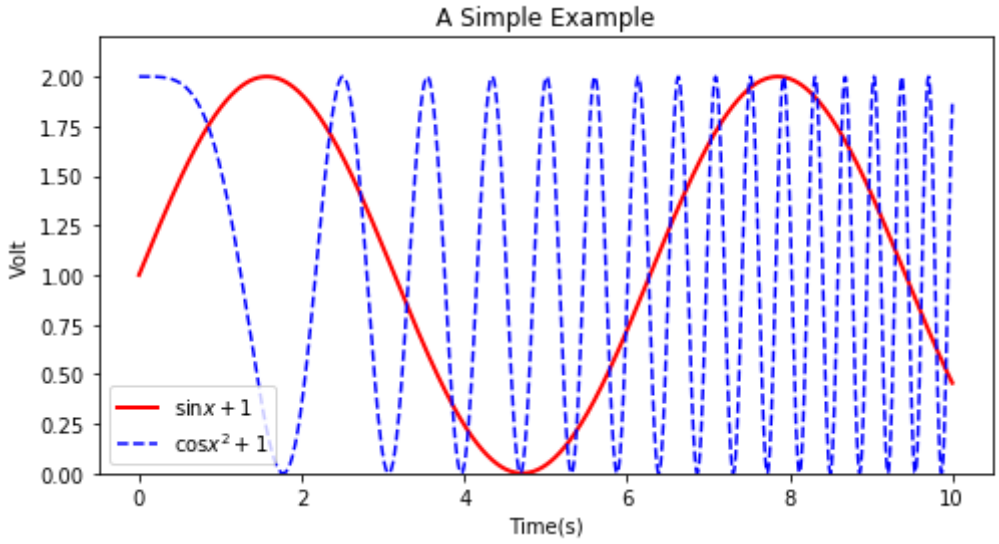
```
# 启动绘图
%matplotlib inline
import matplotlib.pyplot as plt
```

In [48]:

```
x = np.linspace(0, 10, 1000) # 自变量
y = np.sin(x) + 1  # 因变量
z = np.cos(x ** 2) + 1  # 因变量
```

In [49]:

```
plt.figure(figsize=(8,4))  # 设置图像大小
plt.plot(x,y,label='$\sin x+1$', color='red',linewidth=2) # 作图，设置标签，线条颜色大小
plt.plot(x,z,'b--', label='$\cos x^2+1$') # 作图，设置标签、线条类型
plt.xlabel('Time(s)')  # x轴名称
plt.ylabel('Volt')  # y轴名称
plt.title('A Simple Example')  # 标题
plt.ylim(0, 2.2)  # 显示的y轴范围
plt.legend()  # 显示图例
plt.show()  # 显示作图结果
```



4. Pandas

pandas 是后续内容的首选库，Python 中最强大的数据分析和探索工具。因为它含有使数据分析工作变得更快更简单的高级数据结构和操作工具。pandas 是基于NumPy构建的，让以NumPy为中心的应用变得更加简单。

2008 年时，pandas 还无法提供数据分析的工具。在以后的几年中，pandas 逐渐成长为一个非常大的库，能解决越来越多的数据分析问题，但是也逐渐背离了简洁性和易用性。

pandas 名字源于 panel data（面板数据，是计量经济学中关于多维结构化数据集的一个术语）以及 Python data analysis（Python 数据分析），很适合金融数据分析应用的工具

- 兼具NumPy高性能的数组计算功能以及电子表格和关系型数据库（如SQL）灵活的数据处理能力
- 提供了复杂精细的索引功能，以便更为便捷地完成重塑、切片和切块、聚合以及选取数据子集等操作
- 对于金融行业的用户，pandas 提供了大量适用于金融数据的高性能时间序列功能和工具
- 用的最多的 pandas 对象是 DataFrame，是一个面向列（column-oriented）的二维表结构，含有行标和列标

pandas 能够满足的需求：

- 具备按轴自动或显式数据对齐功能的数据结构。这可以防止许多由于数据未对齐以及来自不同数据源（索引方式不同）的数据而导致的常见错误

- 集成时间序列功能
- 既能处理时间序列数据也能处理非时间序列数据的数据结构
- 数学运算和约简（比如对某个轴求和）可以根据不同的元数据（轴编号）执行
- 灵活处理缺失数据
- 合并及其他出现在常见数据库（例如基于 SQL 的）中的关系型运算

Pandas 基本的数据结构是 Series（序列，一维数组）和 DataFrame（二维数据表格，每列是一个 Series）。 我们使用下面的 pandas 引入约定：

```
from pandas import Series, DataFrame
import pandas as pd
```

因为 Series 和 DataFrame 用的次数非常多，因此将其引入本地命名空间中会更方便。

In [50]:

```
from pandas import Series, DataFrame
import pandas as pd
```

In [51]:

```
# 创建一个Series序列
s = Series([1,2,3], index=['d', 'a', 'c'])
s
```

Out[51]:

```
d    1
a    2
c    3
dtype: int64
```

In [52]:

```
s[0]
```

Out[52]:

```
1
```

In [53]:

```
# 创建一个Series序列
s = Series([1,2,3], index=['a', 'b', 'c'])
s
```

Out[53]:

```
a    1
b    2
c    3
dtype: int64
```

In [54]:

```
# 创建一个表DataFrame
d = DataFrame([[1,2,3],[4,5,6]], columns=['a', 'b', 'c'])
d
```

Out[54]:

	a	b	c
0	1	2	3
1	4	5	6

In [55]:

```
# 使用已有的序列来创建表，series是表的一个列
d2 = DataFrame(s)
d2
```

Out[55]:

	0
a	1
b	2
c	3

In [56]:

```
d.describe()  # 数据的基本统计量
```

Out[56]:

	a	b	c
count	2.00000	2.00000	2.00000
mean	2.50000	3.50000	4.50000
std	2.12132	2.12132	2.12132
min	1.00000	2.00000	3.00000
25%	1.75000	2.75000	3.75000
50%	2.50000	3.50000	4.50000
75%	3.25000	4.25000	5.25000
max	4.00000	5.00000	6.00000

In [57]:

```
# 读取文件，注意文件的存储路径不能有中文，否则出错
df = pd.read_csv('data/WeatherResult.csv', header=None, encoding='GBK')#读取csv并创建DF
df.head() # 预览前5行数据
```

Out[57]:

	0	1	2	3	4
0	北京	18日（今天）	多云转小雪	2	-4
1	北京	19日（明天）	阴转晴	1	-8
2	北京	20日（后天）	晴转多云	0	-8
3	北京	21日（周六）	晴	1	-8
4	北京	22日（周日）	晴	-1	-9

In [58]:

```
df.tail()  # 预览最后5行数据
```

Out[58]:

	0	1	2	3	4
72	五大连池	23日（后天）	晴	-17	-31
73	五大连池	24日（周二）	晴	-14	-29
74	五大连池	25日（周三）	晴转多云	-11	-19
75	五大连池	26日（周四）	小雪	-8	-21
76	五大连池	27日（周五）	多云	-11	-22

In [59]:

```
df.describe() # 天气数据的基本统计量
```

Out[59]:

	3	4
count	77.000000	77.000000
mean	6.597403	-2.220779
std	9.143965	10.238759
min	-19.000000	-33.000000
25%	2.000000	-6.000000
50%	8.000000	-2.000000
75%	12.000000	3.000000
max	21.000000	16.000000

5. StatsModels

Pandas着重在数据的读取、处理和探索，而StatsModels则更注重数据的统计建模分析，支持与Pdandas进行数据交互。

安装statsmodels 库

In [60]:

```
# 检查是否安装 StatsModels 库
# 慎重：没有安装的，需要在后台花费时间下载安装，较慢！
# 如果下面平稳性检验出现错误，安装最新版本statsmodels(pip install -U)，然后 Restart
!conda install statsmodels
```

```
Collecting package metadata (current_repodata.json): ...working... done
Solving environment: ...working... done

# All requested packages already installed.
```

In [61]:

```
# 使用StatsModels来进行ADF平稳性检验
# ADF平稳性检验用来检验金融、经济时间序列数据的平稳性
from statsmodels.tsa.stattools import adfuller as ADF  # 导入ADF检验
import numpy as np
ADF(np.random.rand(100))  # 返回的结果有ADF值、p值等
```

Out[61]:

```
(-9.95337338333341,
 2.4807172329060215e-17,
 0,
 99,
 {'1%': -3.498198082189098,
  '5%': -2.891208211860468,
  '10%': -2.5825959973472097},
 39.943792646215826)
```

In [62]:

```
#?ADF # 查阅帮助
```

6. Scikit-Learn

Scikit-Learn 是 Python 下强大的机器学习工具包。 Scikit-Learn官网 (http://scikit-learn.org/stable/index.html)

包含数据预处理、分类、回归、聚类、预测和模型分析等。Scikit-Learn 依赖于 NumPy、SciPy 和 Matplotlib，因此，需要提前安装好这些库。

所有的模型提供的接口有：

- model.fit(): 训练模型，对于监督模型来说就是 fit(X,y)，对于非监督模型就是 fit(X)

监督模型提供的接口有：

- model.predict(X_new): 预测新样本
- model.predict_proba(X_new): 预测概率，仅对某些模型有用（比如 LR ）
- model.score(): 得分越高，fit越好

非监督模型提供的接口有：

- model.transform(): 从数据中学到的新的“基空间”
- model.fit_transform(): 从数据中学到新的基并将数据按照这组“基”进行转换

安装 Scikit-learn 库

```
conda install scikit-learn
```

加载 Scikit-learn 库

```
import sklearn
```

In [63]:

```
import sklearn
sklearn.__version__
```

Out[63]:

'0.22.1'

In [64]:

```
# 导入库
from sklearn.linear_model import LinearRegression  # 导入线性回归模型
model = LinearRegression() # 建立线性回归模型
print(model)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

In [65]:

```
# 导入sklearn本身提供的实例数据
from sklearn import datasets

iris = datasets.load_iris()  # 加载数据集
print(iris.data.shape)  # 查看数据集大小 150个样本，4个特征
```

```
(150, 4)
```

In [66]:

```
b = datasets.load_boston()
print((b.data.shape))
```

```
(506, 13)
```

In [67]:

```
iris.data[:5]  # 前5条数据
```

Out[67]:

```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       [4.6, 3.1, 1.5, 0.2],
       [5. , 3.6, 1.4, 0.2]])
```

In [68]:

```
iris.target[:5]  # 前5条数据的类标（class label）
```

Out[68]:

```
array([0, 0, 0, 0, 0])
```

In [69]:

```
from sklearn import svm  # 导入SVM模型

clf = svm.LinearSVC()  # 建立线性SVM分类器
clf.fit(iris.data, iris.target)  # 用于数据训练模型
clf.coef_  # 查看训练好的模型的参数,这里是支持向量
```

```
C:\Anaconda3\lib\site-packages\sklearn\svm\_base.py:947: ConvergenceWarning: Libline
ar failed to converge, increase the number of iterations.
  "the number of iterations.", ConvergenceWarning)
```

Out[69]:

```
array([[ 0.1842,  0.4512, -0.8079, -0.4507],
       [ 0.055 , -0.8967,  0.4087, -0.9607],
       [-0.8507, -0.9867,  1.381 ,  1.8654]])
```

In [70]:

```
# 训练好模型后，输入新的数据样本1进行类标预测
clf.predict([[5.0, 3.6, 1.3, 0.25]])
```

Out[70]:

```
array([0])
```

In [71]:

```
# 训练好模型后，输入新的数据样本2进行类标预测
clf.predict([[5.0, 1.6, 2.3, 1.95]])
```

Out[71]:

```
array([1])
```

In [72]:

```
# 训练好模型后，输入新的数据样本3进行类标预测
clf.predict([[5.0, 1.6, 4.3, 1.95]])
```

Out[72]:

```
array([2])
```

7. Keras

Scikit-Learn 足够强大，但是没有包含一个强大的模型--人工神经网络。近年来火热的“深度学习”算法，本质上就是一种神经网络。

Keras 库可以用来搭建神经网络，它本身并非简单的神经网络库，而是一个基于 Tensorflow/Theano/CNTK 后端的强大的深度学习库，还可以搭建各种深度学习模型，如自编码器、循环神经网络、递归神经网络、卷积神经网络等。Keras 支持 CPU 和 GPU 切换，速度很快。

Theano 本身也是 Python 的一个库，由深度学习专家 Yoshua Bengio 带领的实验室 MILA 开放，用来定义、优化和解决多维数组数据对应数学表达式的模拟估计问题，具有高效地实现符号分解、高度优化的速度和稳定性等特点，重要的是它还实现了 GPU 加速，使得密集型数据的处理速度是CPU的数十倍。MILA 于 2017 年 9 月 28 日宣布将停止对 Theano 的开发，同年 11 月 15 日发布最终版本 1.0.0。

Tensorflow 最初是由 Google 机器智能研究部门的 Google Brain 团队中的研究人员和工程师开发的，用于进行机器学习和深度神经网络研究， 但它是一个非常基础的系统，因此也可以应用于众多其他领域。

Keras大大简化了搭建各种神经网络模型的步骤。

```
conda install keras
```

注意： 目前keras中使用到TensorFlow，它进展迅速，更新很快，需要随时跟进说明手册，TensorFlow官网(https://www.tensorflow.org/)。

延伸阅读

Colaboratory 是一个 Google 研究项目，旨在帮助传播机器学习培训和研究成果。它是一个 Jupyter 笔记本环境（已安装 Tensorflow），并提供一块 GPU 使用。 工具链接：https://colab.research.google.com/(https://colab.research.google.com/)

In [73]:

```
%system python -V
```

Out[73]:

```
['Python 3.7.6']
```

In [74]:

```
%system conda list keras
```

Out[74]:

```
['# packages in environment at C:\\Anaconda3:',
 '#',
 '# Name                    Version                   Build  Channel',
 'keras                     2.3.1                     0      ',
 'keras-applications        1.0.8                     py_0   ',
 'keras-base                2.3.1                    py37_0  ',
 'keras-preprocessing       1.1.0                     py_1   ']
```

In [75]:

```
%system conda list tensorflow
```

Out[75]:

```
['# packages in environment at C:\\Anaconda3:',
 '#',
 '# Name                    Version                   Build  Channel',
 'tensorflow                2.1.0           eigen_py37hd727fc0_0  ',
 'tensorflow-base           2.1.0           eigen_py37h49b2757_0  ',
 'tensorflow-estimator      2.1.0              pyhd54b08b_0  ']
```

例子1

下面的例子是简单搭建一个LSTM,详见代码 L03/code/LSTM.py 。

在这个模型中，我们将 3 个 LSTM 层叠在一起，使模型能够学习更高层次的时间表示。前两个 LSTM 返回完整的输出序列，但最后一个只返回输出序列的最后一步，从而降低了时间维度（即将输入序列转换成单个向量）。

输出结果在 code/LSTM-Outputs.txt 输出结果每行为每个测试样本在10个类别中的预测概率分布。

In [76]:

```
from keras.models import Sequential
from keras.layers import LSTM, Dense
import numpy as np
import codecs,csv

data_dim = 16
timesteps = 8
nb_classes = 10
filename = "code/LSTM-Outputs.txt"
```

```
Using TensorFlow backend.
```

In [77]:

```
# 简单搭建一个LSTM
model = Sequential()  # 模型初始化
# expected input data shape: (batch_size, timesteps, data_dim)
model.add(LSTM(32, return_sequences=True, input_shape=(timesteps, data_dim)))  # returns a sequenc
model.add(LSTM(32, return_sequences=True))  # returns a sequence of vectors of dimension 32
model.add(LSTM(32))  # return a single vector of dimension 32
model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])
```

In [78]:

```
# 生成训练数据
x_train = np.random.random((1000, timesteps, data_dim))
y_train = np.random.random((1000, nb_classes))
# 生成验证数据
x_val = np.random.random((100, timesteps, data_dim))
y_val = np.random.random((100, nb_classes))
# 生成验证数据
x_test = np.random.random((100, timesteps, data_dim))
y_test = np.random.random((100, nb_classes))

# 训练模型
model.fit(x_train, y_train, batch_size=64, epochs=50,validation_data=(x_val, y_val))
```

```
Train on 1000 samples, validate on 100 samples
Epoch 1/50
1000/1000 [==============================] - 2s 2ms/step - loss: 12.0155 - accu
racy: 0.0980 - val_loss: 12.9912 - val_accuracy: 0.0600
Epoch 2/50
1000/1000 [==============================] - 0s 345us/step - loss: 13.3774 - ac
curacy: 0.0970 - val_loss: 13.6259 - val_accuracy: 0.0700
Epoch 3/50
1000/1000 [==============================] - 0s 390us/step - loss: 13.6863 - ac
curacy: 0.1010 - val_loss: 13.7819 - val_accuracy: 0.0700
Epoch 4/50
1000/1000 [==============================] - 0s 368us/step - loss: 13.7912 - ac
curacy: 0.1010 - val_loss: 13.8274 - val_accuracy: 0.0700
Epoch 5/50
1000/1000 [==============================] - 0s 334us/step - loss: 13.8253 - ac
curacy: 0.1010 - val_loss: 13.8484 - val_accuracy: 0.0700
Epoch 6/50
1000/1000 [==============================] - 0s 305us/step - loss: 13.8364 - ac
curacy: 0.1010 - val_loss: 13.8539 - val_accuracy: 0.0700
Epoch 7/50
1000/1000 [==============================] - 0s 310us/step - loss: 13.8382 - ac
curacy: 0.1010 - val_loss: 13.8587 - val_accuracy: 0.0700
Epoch 8/50
1000/1000 [==============================] - 0s 295us/step - loss: 13.8390 - ac
curacy: 0.1010 - val_loss: 13.8588 - val_accuracy: 0.0700
Epoch 9/50
1000/1000 [==============================] - 0s 317us/step - loss: 13.8356 - ac
curacy: 0.1010 - val_loss: 13.8583 - val_accuracy: 0.0700
Epoch 10/50
1000/1000 [==============================] - 0s 315us/step - loss: 13.8278 - ac
curacy: 0.1010 - val_loss: 13.8631 - val_accuracy: 0.0700
Epoch 11/50
1000/1000 [==============================] - 0s 294us/step - loss: 13.8274 - ac
curacy: 0.1010 - val_loss: 13.8492 - val_accuracy: 0.0700
Epoch 12/50
1000/1000 [==============================] - 0s 293us/step - loss: 13.8215 - ac
curacy: 0.1010 - val_loss: 13.8416 - val_accuracy: 0.0700
Epoch 13/50
1000/1000 [==============================] - 0s 292us/step - loss: 13.8134 - ac
curacy: 0.1010 - val_loss: 13.8331 - val_accuracy: 0.0700
Epoch 14/50
1000/1000 [==============================] - 0s 337us/step - loss: 13.8144 - ac
curacy: 0.1010 - val_loss: 13.8126 - val_accuracy: 0.0700
Epoch 15/50
```

```
1000/1000 [==============================] - 0s 297us/step - loss: 13.7999 - ac
curacy: 0.1010 - val_loss: 13.7985 - val_accuracy: 0.0700
Epoch 16/50
1000/1000 [==============================] - 0s 300us/step - loss: 13.7871 - ac
curacy: 0.1010 - val_loss: 13.7909 - val_accuracy: 0.0700
Epoch 17/50
1000/1000 [==============================] - 0s 303us/step - loss: 13.7685 - ac
curacy: 0.1010 - val_loss: 13.8029 - val_accuracy: 0.0700
Epoch 18/50
1000/1000 [==============================] - 0s 327us/step - loss: 13.7771 - ac
curacy: 0.1010 - val_loss: 13.7892 - val_accuracy: 0.0700
Epoch 19/50
1000/1000 [==============================] - 0s 291us/step - loss: 13.7633 - ac
curacy: 0.1010 - val_loss: 13.7870 - val_accuracy: 0.0700
Epoch 20/50
1000/1000 [==============================] - 0s 293us/step - loss: 13.7560 - ac
curacy: 0.1010 - val_loss: 13.7767 - val_accuracy: 0.0700
Epoch 21/50
1000/1000 [==============================] - 0s 331us/step - loss: 13.7479 - ac
curacy: 0.1010 - val_loss: 13.7796 - val_accuracy: 0.0700
Epoch 22/50
1000/1000 [==============================] - 0s 315us/step - loss: 13.7453 - ac
curacy: 0.1010 - val_loss: 13.7693 - val_accuracy: 0.0700
Epoch 23/50
1000/1000 [==============================] - 0s 299us/step - loss: 13.7359 - ac
curacy: 0.1010 - val_loss: 13.7610 - val_accuracy: 0.0700
Epoch 24/50
1000/1000 [==============================] - 0s 304us/step - loss: 13.7318 - ac
curacy: 0.1010 - val_loss: 13.7467 - val_accuracy: 0.0700
Epoch 25/50
1000/1000 [==============================] - 0s 340us/step - loss: 13.7235 - ac
curacy: 0.1010 - val_loss: 13.7312 - val_accuracy: 0.0700
Epoch 26/50
1000/1000 [==============================] - 0s 301us/step - loss: 13.7085 - ac
curacy: 0.1010 - val_loss: 13.7383 - val_accuracy: 0.0700
Epoch 27/50
1000/1000 [==============================] - 0s 314us/step - loss: 13.7138 - ac
curacy: 0.1010 - val_loss: 13.7211 - val_accuracy: 0.0700
Epoch 28/50
1000/1000 [==============================] - 0s 322us/step - loss: 13.7000 - ac
curacy: 0.1010 - val_loss: 13.7184 - val_accuracy: 0.0700
Epoch 29/50
1000/1000 [==============================] - 0s 308us/step - loss: 13.6935 - ac
curacy: 0.1010 - val_loss: 13.7109 - val_accuracy: 0.0700
Epoch 30/50
1000/1000 [==============================] - 0s 321us/step - loss: 13.6887 - ac
curacy: 0.1010 - val_loss: 13.7021 - val_accuracy: 0.0700
Epoch 31/50
1000/1000 [==============================] - 0s 306us/step - loss: 13.6800 - ac
curacy: 0.1010 - val_loss: 13.6943 - val_accuracy: 0.0700
Epoch 32/50
1000/1000 [==============================] - 0s 336us/step - loss: 13.6732 - ac
curacy: 0.1010 - val_loss: 13.6882 - val_accuracy: 0.0700
Epoch 33/50
1000/1000 [==============================] - 0s 306us/step - loss: 13.6671 - ac
curacy: 0.1010 - val_loss: 13.6771 - val_accuracy: 0.0700
Epoch 34/50
1000/1000 [==============================] - 0s 298us/step - loss: 13.6619 - ac
curacy: 0.1010 - val_loss: 13.6619 - val_accuracy: 0.0700
Epoch 35/50
1000/1000 [==============================] - 0s 342us/step - loss: 13.6507 - ac
```

```
curacy: 0.1010 - val_loss: 13.6552 - val_accuracy: 0.0700
Epoch 36/50
1000/1000 [==============================] - 0s 297us/step - loss: 13.6452 - ac
curacy: 0.1010 - val_loss: 13.6410 - val_accuracy: 0.0700
Epoch 37/50
1000/1000 [==============================] - 0s 303us/step - loss: 13.6274 - ac
curacy: 0.1010 - val_loss: 13.6509 - val_accuracy: 0.0700
Epoch 38/50
1000/1000 [==============================] - 0s 299us/step - loss: 13.6354 - ac
curacy: 0.1010 - val_loss: 13.6390 - val_accuracy: 0.0700
Epoch 39/50
1000/1000 [==============================] - 0s 342us/step - loss: 13.6240 - ac
curacy: 0.1010 - val_loss: 13.6303 - val_accuracy: 0.0700
Epoch 40/50
1000/1000 [==============================] - 0s 306us/step - loss: 13.6124 - ac
curacy: 0.1010 - val_loss: 13.6297 - val_accuracy: 0.0700
Epoch 41/50
1000/1000 [==============================] - 0s 297us/step - loss: 13.6069 - ac
curacy: 0.1010 - val_loss: 13.6303 - val_accuracy: 0.0700
Epoch 42/50
1000/1000 [==============================] - 0s 308us/step - loss: 13.6047 - ac
curacy: 0.1010 - val_loss: 13.6257 - val_accuracy: 0.0700
Epoch 43/50
1000/1000 [==============================] - 0s 335us/step - loss: 13.5994 - ac
curacy: 0.1010 - val_loss: 13.6131 - val_accuracy: 0.0700
Epoch 44/50
1000/1000 [==============================] - 0s 309us/step - loss: 13.5960 - ac
curacy: 0.1010 - val_loss: 13.5927 - val_accuracy: 0.0700
Epoch 45/50
1000/1000 [==============================] - 0s 295us/step - loss: 13.5751 - ac
curacy: 0.1010 - val_loss: 13.6059 - val_accuracy: 0.0700
Epoch 46/50
1000/1000 [==============================] - 0s 338us/step - loss: 13.5847 - ac
curacy: 0.1010 - val_loss: 13.5861 - val_accuracy: 0.0700
Epoch 47/50
1000/1000 [==============================] - 0s 296us/step - loss: 13.5693 - ac
curacy: 0.1010 - val_loss: 13.5835 - val_accuracy: 0.0700
Epoch 48/50
1000/1000 [==============================] - 0s 323us/step - loss: 13.5657 - ac
curacy: 0.1010 - val_loss: 13.5801 - val_accuracy: 0.0700
Epoch 49/50
1000/1000 [==============================] - 0s 304us/step - loss: 13.5574 - ac
curacy: 0.1010 - val_loss: 13.5775 - val_accuracy: 0.0700
Epoch 50/50
1000/1000 [==============================] - 0s 327us/step - loss: 13.5599 - ac
curacy: 0.1010 - val_loss: 13.5619 - val_accuracy: 0.0700
```

Out[78]:

```
<keras.callbacks.callbacks.History at 0x29e4160e848>
```

In [79]:

```
# 预测结果
results = model.predict(x_test, batch_size=32, verbose=0)
# 保存预测结果
with codecs.open(filename, 'w', encoding='utf-8') as f:
    f_csv = csv.writer(f)
    f_csv.writerows(results)

score = model.evaluate(x_test, y_test, batch_size=16) # 在test数据上评估模型
print("\nEvaluation Metrics: \n", model.metrics_names)
print(score)
```

```
100/100 [==============================] - 0s 351us/step

Evaluation Metrics:
 ['loss', 'accuracy']
[13.430894813537599, 0.07999999821186066]
```

例子2

下面的例子是简单搭建一个MLP(多层感知机),详见代码 L03/code/MLP.py 。

在这个模型中，我们使用了一个预测x平方的回归例子数据，并绘制结果的散布图。

In [80]:

```
# 导入库
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.optimizers import SGD

# 生成数据
import numpy as np
# x为（-0.5,0.5）内均匀生成，其中训练样本1000个，测试样本100个
x_train=np.linspace(-0.5,0.5,1000)
x_test=np.linspace(-0.5,0.5,100)
# y为x*x，并添加波动值作为噪音
y_train=np.square(x_train)+np.random.normal(0,0.02,x_train.shape)
y_test=np.square(x_test)+np.random.normal(0,0.02,x_test.shape)
```

In [81]:

```
# 简单搭建一个MLP（多层感知机）

# 模型初始化
model = Sequential()
# 输入层（1个节点）到第一隐含层（64节点）的连接，使用relu作为激活函数
# 在第一层必须指定输入的大小，这里是1个隐藏单元
model.add(Dense(64, activation='relu', input_dim=1))
model.add(Dropout(0.5))  # 使用Dropout防止过拟合
model.add(Dense(1, activation='relu'))  # 添加输出层（1节点）

model.summary() # 打印模型信息以及参数数量
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 64)                128
_________________________________________________________________
dropout_1 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 65
=================================================================
Total params: 193
Trainable params: 193
Non-trainable params: 0
_________________________________________________________________
```

In [82]:

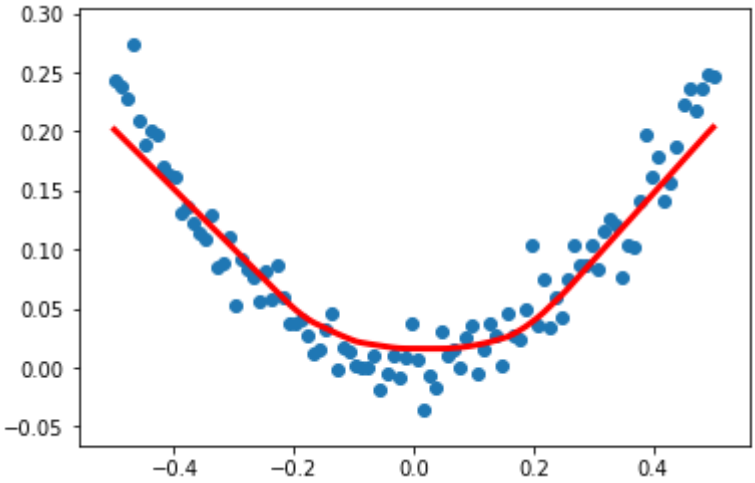
```
sgd = SGD(lr=0.3)  # 定义求解算法
model.compile(loss='mse',optimizer=sgd,metrics=['accuracy'])  # 编译生成模型

model.fit(x_train, y_train, epochs=10, batch_size=32)  # 训练模型
score = model.evaluate(x_test, y_test, batch_size=32)  # 测试模型
y_pred = model.predict(x_test)  #得到预测值
```

```
Epoch 1/10
1000/1000 [==============================] - 0s 161us/step - loss: 0.0056 - accurac
y: 0.0000e+00
Epoch 2/10
1000/1000 [==============================] - 0s 53us/step - loss: 0.0030 - accuracy:
0.0000e+00
Epoch 3/10
1000/1000 [==============================] - 0s 52us/step - loss: 0.0021 - accuracy:
0.0000e+00
Epoch 4/10
1000/1000 [==============================] - 0s 76us/step - loss: 0.0017 - accuracy:
0.0000e+00
Epoch 5/10
1000/1000 [==============================] - 0s 69us/step - loss: 0.0016 - accuracy:
0.0000e+00
Epoch 6/10
1000/1000 [==============================] - 0s 60us/step - loss: 0.0014 - accuracy:
0.0000e+00
Epoch 7/10
1000/1000 [==============================] - 0s 71us/step - loss: 0.0011 - accuracy:
0.0000e+00
Epoch 8/10
1000/1000 [==============================] - 0s 63us/step - loss: 0.0012 - accuracy:
0.0000e+00
Epoch 9/10
1000/1000 [==============================] - 0s 80us/step - loss: 0.0013 - accuracy:
0.0000e+00
Epoch 10/10
1000/1000 [==============================] - 0s 58us/step - loss: 0.0014 - accuracy:
0.0000e+00
100/100 [==============================] - 0s 530us/step
```

In [83]:

```
# 绘图
import matplotlib.pyplot as plt
# 将真实y值y_test绘制为散点图
plt.scatter(x_test,y_test)
# 将预测y值y_pred绘制为红色曲线
plt.plot(x_test,y_pred,'r-',lw=3)
plt.show()
```



8. Gensim

Gensim 处理语言方面的任务，例如，文本相似度、LDA 语言模型、Word2Vec（Google 公司在 2013 年开源的著名的词向量构造工具）等。

Gensim 对 Word2Vec 代码进行了优化，在 Linux 环境下运行更快。

In [84]:

```
import gensim, logging  # 安装logging
logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)s', level=logging.INFO)
# logging是用来输出训练日志
```

In [85]:

```
# 得到分词后的句子，每个句子以词列表的形式输入
sentences = [['first', 'sentence'],
             ['second', 'sentence']]
```

In [ ]:

```
# 用上面的句子训练词向量模型
import warnings
warnings.filterwarnings("ignore")
model = gensim.models.Word2Vec(sentences, size=10, min_count=1) # size是词向量维度
```

In [87]:

```
# 输出单词sentence的词向量
print(model['sentence'])
```

```
[-0.0281  0.0002 -0.0475  0.0328  0.0277  0.0052  0.0248  0.0488  0.0421
  0.0011]
```

9. NLTK

NLTK 的全称是 Natural Language ToolKit，是一套基于python的自然语言处理工具集。 NLTK 中包含了许多用于进行语言处理和分析的数据语料和模型工具等，通过在Python解释器中执行。

安装1：

conda install nltk (注意：下载安装时间比较久)

安装2：通过在Python解释器中运行 Python 后，在提示符后面输入：

```
import nltk
nltk.download()
```

In [88]:

```
import nltk
from nltk.corpus import wordnet
```

单词的同义词集

In [89]:

```
# 获得一个词的所有sense，包括词语的各种变形的sense：
wordnet.synsets('channel')
# 一个synset(同义词集：指意义相同的词条的集合)被一个三元组描述：（单词.词性.序号）
```

Out[89]:

```
[Synset('channel.n.01'),
 Synset('channel.n.02'),
 Synset('groove.n.01'),
 Synset('channel.n.04'),
 Synset('channel.n.05'),
 Synset('duct.n.01'),
 Synset('channel.n.07'),
 Synset('distribution_channel.n.01'),
 Synset('impart.v.03'),
 Synset('channel.v.02'),
 Synset('transmit.v.04')]
```

In [90]:

```
# 获得一个词的所有某个pos词性的sense
wordnet.synsets('channel',pos= wordnet.VERB)
```

Out[90]:

```
[Synset('impart.v.03'), Synset('channel.v.02'), Synset('transmit.v.04')]
```

In [91]:

```
# 一个同义词集synset的定义
print(wordnet.synset('channel.n.01').definition())
# 一个同义词集的所有词条
print(wordnet.synset('channel.n.01').lemmas())
#一个同义词集的所有词条的名字
print(wordnet.synset('channel.n.01').lemma_names())
#一个词条所属的同义词集
print(wordnet.lemma('channel.n.01.channel').synset())
#一个词条所属的同义词集的名字
print(wordnet.lemma('channel.n.01.channel').name())
```

```
a path over which electrical signals can pass
[Lemma('channel.n.01.channel'), Lemma('channel.n.01.transmission_channel')]
['channel', 'transmission_channel']
Synset('channel.n.01')
channel
```

In [92]:

```
#在wordnet中，名词和动词被组织成了完整的层次式分类体系，因此可以通过计算两个sense在分类树中的距离，.
x = wordnet.synsets('girl')[-1]
y = wordnet.synsets('woman')[-1]
print(x.shortest_path_distance(y))
```

```
11
```

In [93]:

```
# 形容词和副词没有被组织成分类体系，所以不能用path_distance。形容词和副词最有用的关系是similar to。
a = wordnet.synsets('glorious')[0]
a.similar_tos()
```

Out[93]:

```
[Synset('bright.s.06'),
 Synset('celebrated.s.02'),
 Synset('divine.s.06'),
 Synset('empyreal.s.02'),
 Synset('illustrious.s.02'),
 Synset('incandescent.s.02'),
 Synset('lustrous.s.02')]
```

取词干Stem

In [94]:

```
import nltk
sent1='The computers are computing the sum of numbers.'
#sent2='A dog was running across the kitchen.'
tokens_1=nltk.word_tokenize(sent1)
print (tokens_1)

stemmer = nltk.stem.PorterStemmer()
stem_1 = [stemmer.stem(t) for t in tokens_1]
print(stem_1)
```

```
['The', 'computers', 'are', 'computing', 'the', 'sum', 'of', 'numbers', '.']
['the', 'comput', 'are', 'comput', 'the', 'sum', 'of', 'number', '.']
```

词性标注

In [95]:

```
# 句子中每个词的词性标注
sent="A man struck my video camera with a hammer"
sent_tokenize_list = nltk.word_tokenize(sent)
nltk.pos_tag(sent_tokenize_list)
```

Out[95]:

```
[('A', 'DT'),
 ('man', 'NN'),
 ('struck', 'VBD'),
 ('my', 'PRP$'),
 ('video', 'NN'),
 ('camera', 'NN'),
 ('with', 'IN'),
 ('a', 'DT'),
 ('hammer', 'NN')]
```

10. jieba 分词 / 词性标注

安装

```
pip intall jieba
```

功能

1） 分词（tokenization/segmentation），有三种模式

- seg_words = jieba.cut(s, cut_all=True) # 全模式
- seg_words = jieba.cut(s, cut_all=False) # 默认是精确模式
- seg_words = jieba.cut_for_search(s) # 搜索引擎模式

In [96]:

```
import jieba
sent='中国媒体正在向美国政府对华挥舞关税大棒的行为发出警告'
seg_words = jieba.cut(sent, cut_all=False)  # 默认是精确模式
#seg_words = jieba.cut(sent, cut_all=True) # 全模式
#seg_words = jieba.cut_for_search(sent)  # 搜索引擎模式
print(' / '.join(seg_words))
```

```
Building prefix dict from the default dictionary ...
2020-02-26 20:28:21,722 : DEBUG : Building prefix dict from the default dictionary
...
Loading model from cache C:\Users\lanman\AppData\Local\Temp\jieba.cache
2020-02-26 20:28:21,726 : DEBUG : Loading model from cache C:\Users\lanman\AppData\L
ocal\Temp\jieba.cache
Loading model cost 1.555 seconds.
2020-02-26 20:28:23,280 : DEBUG : Loading model cost 1.555 seconds.
Prefix dict has been built successfully.
2020-02-26 20:28:23,289 : DEBUG : Prefix dict has been built successfully.

中国 / 媒体 / 正在 / 向 / 美国政府 / 对华 / 挥舞 / 关税 / 大棒 / 的 / 行为 / 发出 /
警告
```

In [97]:

```
sent="这样的人才能经受住考验"
seg_words = jieba.cut(sent, cut_all=True) # 全模式
print(' / '.join(seg_words))
```

```
这样 / 的 / 人才 / 才能 / 经受 / 住 / 考验
```

In [98]:

```
sent="中资已成为澳门最大的外来投资者"
seg_words = jieba.cut_for_search(sent)  # 搜索引擎模式
print(' / '.join(seg_words))
```

```
中资 / 已 / 成为 / 澳门 / 最大 / 的 / 外来 / 投资 / 投资者
```

2. 词性标注(Part-of-Speech, POS)

jieba.posseg.dt 为默认词性标注分词器。

标注句子分词后每个词的词性,采用和 ictclas 兼容的标记法。

In [99]:

```
import jieba.posseg as pseg
sent='美国本土企业特斯拉公司首席执行官马斯克早前宣布，将在中国上海建立首个海外工厂'
words = pseg.cut(sent)
for word, flag in words:
    print('%s/%s '%(word,flag),end='')
```

美国/ns 本土/n 企业/n 特斯拉/nrt 公司/n 首席/n 执行官/n 马斯克/nr 早前/t 宣布/v ，/x
将/d 在/p 中国/ns 上海/ns 建立/v 首个/m 海外/s 工厂/n

11. SnowNLP

pip安装

```
pip install snownlp
```

功能:

- 1. 转化为拼音
- 2. 繁体转为简体
- 3. 自动摘要, 抽取式摘要
- 4. 情感分析

In [100]:

```
# 1. 转化为拼音
from snownlp import SnowNLP
s = SnowNLP("杭州西湖风景很好，是旅游胜地！")
s.pinyin
```

Out[100]:

```
['hang',
 'zhou',
 'xi',
 'hu',
 'feng',
 'jing',
 'hen',
 'hao',
 ' ',
 'shi',
 'lv',
 'you',
 'sheng',
 'di',
 '！']
```

In [101]:

```
# 2. 繁体转为简体
s = SnowNLP(u'「繁體字」「繁體中文」的叫法在臺灣亦很常見。')
s.han
```

Out[101]:

'「繁体字」「繁体中文」的叫法在台湾亦很常见。'

In [102]:

```
# 3. 自动摘要，抽取式摘要
text = u'''
自然语言处理是计算机科学领域与人工智能领域中的一个重要方向。
它研究能实现人与计算机之间用自然语言进行有效通信的各种理论和方法。
自然语言处理是一门融语言学、计算机科学、数学于一体的科学。
因此，这一领域的研究将涉及自然语言，即人们日常使用的语言，所以它与语言学的研究有着密切的联系，但又有
自然语言处理并不是一般地研究自然语言，而在于研制能有效地实现自然语言通信的计算机系统，特别是其中的软
因而它是计算机科学的一部分。
'''
s = SnowNLP(text)
s.summary(3)
```

Out[102]:

['因而它是计算机科学的一部分',
 '自然语言处理是计算机科学领域与人工智能领域中的一个重要方向',
 '自然语言处理是一门融语言学、计算机科学、数学于一体的科学']

In [103]:

```
# 4. 情感分析
text = u'才用了一次就坏了'
s=SnowNLP(text)
s.sentiments
```

Out[103]:

0.03180889798546371

12. PKUSeg

北京大学开源的一个中文分词工具包

GitHub地址：https://github.com/lancopku/PKUSeg-python (https://github.com/lancopku/PKUSeg-python)

安装

```
pip install pkuseg
```

In [104]:

```
#代码示例1： 使用默认模型及默认词典分词
import pkuseg
seg = pkuseg.pkuseg()                                    # 以默认配置加载模型

text = seg.cut('我爱北京天安门')                          # 进行分词
print(text)
```

['我', '爱', '北京', '天安门']

In [105]:

```
#代码示例2：设置用户自定义词典
import pkuseg
lexicon = ['北京大学', '北京天安门']                      # 希望分词时用户词典中的词固定不分开
seg = pkuseg.pkuseg(user_dict=lexicon)                   # 加载模型，给定用户词典

text = seg.cut('我爱北京天安门')                          # 进行分词
print(text)
```

['我', '爱', '北京天安门']

In [106]:

```
# 代码示例3：对文件分词,时间比较长！
import pkuseg
pkuseg.test('data/input.txt', 'data/output.txt', nthread=20)      # 对input.txt的文件分词输出到outpu
                                                         # 使用默认模型和词典，开20个进程
```

```
total_time:     209.346
```

13. Stanford CoreNLP

Stanford CoreNLP是用处理自然语言的工具集合。它可以给出词语的基本形式：词性（它们是公司名、人名等，规范化得日期，时间，和数字），根据短语和语法依赖来标记句子的结构，发现实体之间的关系、情感以及人们所说的话等。选择Stanford CoreNLP的理由：

- 1、它是一个众多语法分析工具集成的工具包。
- 2、对任意的文本来说它具有快和鲁棒性强的特点，并且还广泛的用于生产中。
- 3、它还具有时用性，及时性。
- 4、支持数量众多的（主要）自然语言。
- 5、支持编程语言的接口丰富。
- 6、能够作为简单的web服务运行。

Stanford CoreNLP包含许多的斯坦福的NLP工具，包括：词性标注器、命名实体的识别器、解析器（句子与语法结构）、指代消解器（就是在篇章中确定代词指向哪个名词短语的问题）、情感分析器、引导模式学习器、开放信息提取器。 Stanford CoreNLP的目的是为了让一系列语言分析工具应用到一段文本上变得更容易。用两行代码就可以让一段原生的文本在一个工具管道（一系列处理文本的操作）中流过，而且一个解释管道还可以包含其他自定义或者第三方的解释器。Stanford CoreNLP 具有很强的灵活性和扩展性。通过一个选项的设置你就可以选择那些语言处理的工具可以应用到这段文本。

1）下载Stanford CoreNLP

- 1.最新版本为stanford-corenlp-3.9.2(Python3.7暂时不支持)，可以下载stanford-corenlp-3.9.1。

下载地址1: https://stanfordnlp.github.io/CoreNLP/#download (https://stanfordnlp.github.io/CoreNLP/#download)

下载地址2:https://stanfordnlp.github.io/CoreNLP/ (https://stanfordnlp.github.io/CoreNLP/)

- 2.解压缩到 D:\stanfordNLP\stanford-corenlp-full-2018-02-27

2) 安装CoreNLP

```
pip install stanfordcorenlp
```

3 ） 安装Java

Stanford CoreNLP需要Java 1.8+才可以运行。

In [107]:

```
%system java -version
```

Out[107]:

```
['java version "1.8.0_161"',
 'Java(TM) SE Runtime Environment (build 1.8.0_161-b12)',
 'Java HotSpot(TM) 64-Bit Server VM (build 25.161-b12, mixed mode)']
```

In [108]:

```
%system conda list stanfordcorenlp
```

Out[108]:

```
['# packages in environment at C:\\Anaconda3:',
 '#',
 '# Name                    Version                   Build  Channel',
 'stanfordcorenlp          3.9.1.1                  pypi_0    pypi']
```

In [109]:

```
# 1.词性标注POS：中文
from stanfordcorenlp import StanfordCoreNLP #导入StandfordCoreNLP模块
import logging

nlp = StanfordCoreNLP(path_or_host='D:\stanfordNLP\stanford-corenlp-full-2018-02-27', lang='zh',
                      quiet=False,logging_level=logging.DEBUG)
sentence='美国本土企业特斯拉公司首席执行官马斯克早前宣布，将在中国上海建立首个海外工厂'
print(nlp.pos_tag(sentence))
nlp.close() # Do not forget to close! The backend server will consume a lot memery.
```

```
2020-02-26 20:32:18,967 : INFO : Initializing native server...
2020-02-26 20:32:18,977 : INFO : java -Xmx4g -cp "D:\stanfordNLP\stanford-corenlp-fu
ll-2018-02-27\*" edu.stanford.nlp.pipeline.StanfordCoreNLPServer -port 9000
2020-02-26 20:32:18,998 : INFO : Server shell PID: 13960
2020-02-26 20:32:20,545 : INFO : The server is available.
2020-02-26 20:32:20,547 : INFO : {'properties': "{'annotators': 'pos', 'outputForma
t': 'json'}", 'pipelineLanguage': 'zh'}
2020-02-26 20:32:43,895 : INFO : Cleanup...
2020-02-26 20:32:43,921 : INFO : Killing pid: 10748, cmdline: ['java', '-Xmx4g', '-c
p', 'D:\\stanfordNLP\\stanford-corenlp-full-2018-02-27\\*', 'edu.stanford.nlp.pipeli
ne.StanfordCoreNLPServer', '-port', '9000']
2020-02-26 20:32:43,923 : INFO : Killing shell pid: 13960, cmdline: ['C:\\WINDOWS\\s
ystem32\\cmd.exe', '/c', 'java -Xmx4g -cp D:\\stanfordNLP\\stanford-corenlp-full-201
8-02-27\\* edu.stanford.nlp.pipeline.StanfordCoreNLPServer -port 9000']

[('美国', 'NR'), ('本土', 'NN'), ('企业', 'NN'), ('特斯拉', 'NR'), ('公司', 'NN'),
('首席', 'JJ'), ('执行官', 'NN'), ('马斯克早前', 'NR'), ('宣布', 'VV'), (', ', 'P
U'), ('将', 'AD'), ('在', 'P'), ('中国', 'NR'), ('上海', 'NR'), ('建立', 'VV'),
('首', 'OD'), ('个', 'M'), ('海外', 'NN'), ('工厂', 'NN')]
```

In [110]:

```
# 2.词性标注POS：英文
from stanfordcorenlp import StanfordCoreNLP #导入StandfordCoreNLP模块
import logging

nlp = StanfordCoreNLP('D:\stanfordNLP\stanford-corenlp-full-2018-02-27')
sentence="A man struck my video camera with a hammer"
print(nlp.pos_tag(sentence))
nlp.close() # Do not forget to close! The backend server will consume a lot memery.
```

```
2020-02-26 20:32:44,217 : INFO : Initializing native server...
2020-02-26 20:32:44,219 : INFO : java -Xmx4g -cp "D:\stanfordNLP\stanford-corenlp-fu
ll-2018-02-27\*" edu.stanford.nlp.pipeline.StanfordCoreNLPServer -port 9001
2020-02-26 20:32:44,232 : INFO : Server shell PID: 4612
2020-02-26 20:32:45,235 : INFO : The server is available.
2020-02-26 20:32:45,238 : INFO : {'properties': "{'annotators': 'pos', 'outputForma
t': 'json'}", 'pipelineLanguage': 'en'}
2020-02-26 20:32:47,773 : INFO : Cleanup...
2020-02-26 20:32:47,793 : INFO : Killing pid: 7172, cmdline: ['java', '-Xmx4g', '-c
p', 'D:\\stanfordNLP\\stanford-corenlp-full-2018-02-27\\*', 'edu.stanford.nlp.pipeli
ne.StanfordCoreNLPServer', '-port', '9001']
2020-02-26 20:32:47,793 : INFO : Killing shell pid: 4612, cmdline: ['C:\\WINDOWS\\sy
stem32\\cmd.exe', '/c', 'java -Xmx4g -cp D:\\stanfordNLP\\stanford-corenlp-full-2018
-02-27\\* edu.stanford.nlp.pipeline.StanfordCoreNLPServer -port 9001']

[('A', 'DT'), ('man', 'NN'), ('struck', 'VBD'), ('my', 'PRP$'), ('video', 'NN'), ('c
amera', 'NN'), ('with', 'IN'), ('a', 'DT'), ('hammer', 'NN')]
```

In [111]:

```
# 3. 句子解析Parser：中文
from stanfordcorenlp import StanfordCoreNLP #导入StandfordCoreNLP模块
import logging
nlp = StanfordCoreNLP('D:\stanfordNLP\stanford-corenlp-full-2018-02-27', lang='zh',logging_level=30
sentence='美国本土企业特斯拉公司首席执行官马斯克早前宣布，将在中国上海建立首个海外工厂'
print(nlp.parse(sentence))
nlp.close() # Do not forget to close! The backend server will consume a lot memery.
```

```
2020-02-26 20:32:48,039 : INFO : Initializing native server...
2020-02-26 20:32:48,040 : INFO : java -Xmx4g -cp "D:\stanfordNLP\stanford-corenlp-fu
ll-2018-02-27\*" edu.stanford.nlp.pipeline.StanfordCoreNLPServer -port 9001
2020-02-26 20:32:48,052 : INFO : Server shell PID: 11104
2020-02-26 20:32:49,063 : INFO : The server is available.
2020-02-26 20:32:49,063 : INFO : {'properties': "{'annotators': 'pos,parse', 'output
Format': 'json'}", 'pipelineLanguage': 'zh'}
2020-02-26 20:33:46,641 : INFO : Cleanup...
2020-02-26 20:33:46,661 : INFO : Killing pid: 5480, cmdline: ['java', '-Xmx4g', '-c
p', 'D:\\stanfordNLP\\stanford-corenlp-full-2018-02-27\\*', 'edu.stanford.nlp.pipeli
ne.StanfordCoreNLPServer', '-port', '9001']
2020-02-26 20:33:46,661 : INFO : Killing shell pid: 11104, cmdline: ['C:\\WINDOWS\\s
ystem32\\cmd.exe', '/c', 'java -Xmx4g -cp D:\\stanfordNLP\\stanford-corenlp-full-201
8-02-27\\* edu.stanford.nlp.pipeline.StanfordCoreNLPServer -port 9001']


(ROOT
  (IP
    (NP
      (NP
        (NP (NR 美国))
        (NP (NN 本土) (NN 企业)))
      (NP
        (NP (NR 特斯拉) (NN 公司))
        (ADJP (JJ 首席))
        (NP (NN 执行官)))
      (NP (NR 马斯克早前)))
    (VP (VV 宣布) (PU ，)
      (IP
        (VP
          (ADVP (AD 将))
          (PP (P 在)
            (NP (NR 中国) (NR 上海)))
          (VP (VV 建立)
            (NP
              (QP (OD 首)
                (CLP (M 个)))
              (NP (NN 海外) (NN 工厂)))))))))
```

In [112]:

```
# 4. 句子解析Parser：英文
from stanfordcorenlp import StanfordCoreNLP #导入StandfordCoreNLP模块
import logging
nlp = StanfordCoreNLP('D:\stanfordNLP\stanford-corenlp-full-2018-02-27')
sentence="A man saw a boy with a hammer."
print(nlp.parse(sentence))
nlp.close() # Do not forget to close! The backend server will consume a lot memery.
```

```
2020-02-26 20:33:46,979 : INFO : Initializing native server...
2020-02-26 20:33:46,979 : INFO : java -Xmx4g -cp "D:\stanfordNLP\stanford-corenlp-fu
ll-2018-02-27\*" edu.stanford.nlp.pipeline.StanfordCoreNLPServer -port 9002
2020-02-26 20:33:47,004 : INFO : Server shell PID: 6740
2020-02-26 20:33:48,010 : INFO : The server is available.
2020-02-26 20:33:48,010 : INFO : {'properties': "{'annotators': 'pos,parse', 'output
Format': 'json'}", 'pipelineLanguage': 'en'}
2020-02-26 20:33:53,195 : INFO : Cleanup...
2020-02-26 20:33:53,213 : INFO : Killing pid: 5720, cmdline: ['java', '-Xmx4g', '-c
p', 'D:\\stanfordNLP\\stanford-corenlp-full-2018-02-27\\*', 'edu.stanford.nlp.pipeli
ne.StanfordCoreNLPServer', '-port', '9002']
2020-02-26 20:33:53,215 : INFO : Killing shell pid: 6740, cmdline: ['C:\\WINDOWS\\sy
stem32\\cmd.exe', '/c', 'java -Xmx4g -cp D:\\stanfordNLP\\stanford-corenlp-full-2018
-02-27\\* edu.stanford.nlp.pipeline.StanfordCoreNLPServer -port 9002']


(ROOT
  (S
    (NP (DT A) (NN man))
    (VP (VBD saw)
      (NP (DT a) (NN boy))
      (PP (IN with)
        (NP (DT a) (NN hammer))))
    (. .)))
```

In [113]:

```
# 5. nlp进行分词、词性标注、NER、句子解析：中文
from stanfordcorenlp import StanfordCoreNLP

nlp = StanfordCoreNLP('D:\stanfordNLP\stanford-corenlp-full-2018-02-27',lang="zh")

sentence = '中国应对疫情措施体现了中国的制度优势'
#sentence = '世卫组织专家考察组组长表示，中国应对疫情措施体现了中国的制度优势'
print('Tokenize:', nlp.word_tokenize(sentence))
print('Part of Speech:', nlp.pos_tag(sentence))
print('Named Entities:', nlp.ner(sentence))
print('Constituency Parsing:', nlp.parse(sentence))
print('Dependency Parsing:', nlp.dependency_parse(sentence))

nlp.close() # Do not forget to close! The backend server will consume a lot memery.
```

```
2020-02-26 20:33:53,463 : INFO : Initializing native server...
2020-02-26 20:33:53,464 : INFO : java -Xmx4g -cp "D:\stanfordNLP\stanford-corenlp-fu
ll-2018-02-27\*" edu.stanford.nlp.pipeline.StanfordCoreNLPServer -port 9002
2020-02-26 20:33:53,478 : INFO : Server shell PID: 5112
2020-02-26 20:33:54,488 : INFO : The server is available.
2020-02-26 20:33:54,488 : INFO : {'properties': "{'annotators': 'ssplit,tokenize',
'outputFormat': 'json'}", 'pipelineLanguage': 'zh'}
2020-02-26 20:34:10,987 : INFO : {'properties': "{'annotators': 'pos', 'outputForma
t': 'json'}", 'pipelineLanguage': 'zh'}

Tokenize: ['中国', '应对', '疫情', '措施', '体现', '了', '中国', '的', '制度', '优
势']

2020-02-26 20:34:13,119 : INFO : {'properties': "{'annotators': 'ner', 'outputForma
t': 'json'}", 'pipelineLanguage': 'zh'}

Part of Speech: [('中国', 'NR'), ('应对', 'VV'), ('疫情', 'NN'), ('措施', 'NN'),
('体现', 'VV'), ('了', 'AS'), ('中国', 'NR'), ('的', 'DEG'), ('制度', 'NN'), ('优
势', 'NN')]

2020-02-26 20:34:22,807 : INFO : {'properties': "{'annotators': 'pos,parse', 'output
Format': 'json'}", 'pipelineLanguage': 'zh'}

Named Entities: [('中国', 'COUNTRY'), ('应对', 'O'), ('疫情', 'O'), ('措施', 'O'),
('体现', 'O'), ('了', 'O'), ('中国', 'COUNTRY'), ('的', 'O'), ('制度', 'O'), ('优
势', 'O')]

2020-02-26 20:34:51,901 : INFO : {'properties': "{'annotators': 'depparse', 'outputF
ormat': 'json'}", 'pipelineLanguage': 'zh'}

Constituency Parsing: (ROOT
  (IP
    (NP (NR 中国))
    (VP
      (VP (VV 应对)
        (NP
          (ADJP (NN 疫情))
          (NP (NN 措施))))
      (VP (VV 体现) (AS 了)
        (NP
```

```
(DNP
  (NP (NR 中国))
  (DEG 的))
(NP (NN 制度) (NN 优势)))))))
```

```
2020-02-26 20:35:14,213 : INFO : Cleanup...
2020-02-26 20:35:14,213 : INFO : Killing pid: 13248, cmdline: ['java', '-Xmx4g', '-cp', 'D:\\stanfordNLP\\stanford-corenlp-full-2018-02-27\\*', 'edu.stanford.nlp.pipeline.StanfordCoreNLPServer', '-port', '9002']
2020-02-26 20:35:14,213 : INFO : Killing shell pid: 5112, cmdline: ['C:\\WINDOWS\\system32\\cmd.exe', '/c', 'java -Xmx4g -cp D:\\stanfordNLP\\stanford-corenlp-full-2018-02-27\\* edu.stanford.nlp.pipeline.StanfordCoreNLPServer -port 9002']
```

```
Dependency Parsing: [('ROOT', 0, 2), ('nsubj', 2, 1), ('compound:nn', 4, 3), ('dobj', 2, 4), ('conj', 2, 5), ('aux:asp', 5, 6), ('nmod:assmod', 10, 7), ('case', 7, 8), ('compound:nn', 10, 9), ('dobj', 5, 10)]
```

In [114]:

```
# 6. nlp进行分词、词性标注、NER、句子解析：英文
from stanfordcorenlp import StanfordCoreNLP

nlp = StanfordCoreNLP('D:\stanfordNLP\stanford-corenlp-full-2018-02-27')

sentence = 'Stanford CoreNLP provides a set of human language technology tools.'
print('Tokenize:', nlp.word_tokenize(sentence))
print('Part of Speech:', nlp.pos_tag(sentence))
print('Named Entities:', nlp.ner(sentence))
print('Constituency Parsing:', nlp.parse(sentence))
print('Dependency Parsing:', nlp.dependency_parse(sentence))

nlp.close() # Do not forget to close! The backend server will consume a lot memery.
```

```
2020-02-26 20:35:14,474 : INFO : Initializing native server...
2020-02-26 20:35:14,476 : INFO : java -Xmx4g -cp "D:\stanfordNLP\stanford-corenlp-fu
ll-2018-02-27\*" edu.stanford.nlp.pipeline.StanfordCoreNLPServer -port 9000
2020-02-26 20:35:14,490 : INFO : Server shell PID: 2372
2020-02-26 20:35:15,496 : INFO : The server is available.
2020-02-26 20:35:15,498 : INFO : {'properties': "{'annotators': 'ssplit,tokenize',
'outputFormat': 'json'}", 'pipelineLanguage': 'en'}
2020-02-26 20:35:16,127 : INFO : {'properties': "{'annotators': 'pos', 'outputForma
t': 'json'}", 'pipelineLanguage': 'en'}

Tokenize: ['Stanford', 'CoreNLP', 'provides', 'a', 'set', 'of', 'human', 'language',
'technology', 'tools', '.']

2020-02-26 20:35:18,398 : INFO : {'properties': "{'annotators': 'ner', 'outputForma
t': 'json'}", 'pipelineLanguage': 'en'}

Part of Speech: [('Stanford', 'NNP'), ('CoreNLP', 'NNP'), ('provides', 'VBZ'), ('a',
'DT'), ('set', 'NN'), ('of', 'IN'), ('human', 'JJ'), ('language', 'NN'), ('technolog
y', 'NN'), ('tools', 'NNS'), ('.', '.')]

2020-02-26 20:35:53,938 : INFO : {'properties': "{'annotators': 'pos,parse', 'output
Format': 'json'}", 'pipelineLanguage': 'en'}

Named Entities: [('Stanford', 'ORGANIZATION'), ('CoreNLP', 'O'), ('provides', 'O'),
('a', 'O'), ('set', 'O'), ('of', 'O'), ('human', 'O'), ('language', 'O'), ('technolo
gy', 'O'), ('tools', 'O'), ('.', 'O')]

2020-02-26 20:35:55,087 : INFO : {'properties': "{'annotators': 'depparse', 'outputF
ormat': 'json'}", 'pipelineLanguage': 'en'}

Constituency Parsing: (ROOT
  (S
    (NP (NNP Stanford) (NNP CoreNLP))
    (VP (VBZ provides)
      (NP
        (NP (DT a) (NN set))
        (PP (IN of)
          (NP (JJ human) (NN language) (NN technology) (NNS tools)))))
    (. .)))
```

```
2020-02-26 20:36:22,488 : INFO : Cleanup...
2020-02-26 20:36:22,504 : INFO : Killing pid: 11244, cmdline: ['java', '-Xmx4
g', '-cp', 'D:\\stanfordNLP\\stanford-corenlp-full-2018-02-27\\*', 'edu.stanfor
d.nlp.pipeline.StanfordCoreNLPServer', '-port', '9000']
2020-02-26 20:36:22,506 : INFO : Killing shell pid: 2372, cmdline: ['C:\\WINDOW
S\\system32\\cmd.exe', '/c', 'java -Xmx4g -cp D:\\stanfordNLP\\stanford-corenlp
-full-2018-02-27\\* edu.stanford.nlp.pipeline.StanfordCoreNLPServer -port 900
0']

Dependency Parsing: [('ROOT', 0, 3), ('compound', 2, 1), ('nsubj', 3, 2), ('det', 5,
4), ('dobj', 3, 5), ('case', 10, 6), ('amod', 10, 7), ('compound', 10, 8), ('compoun
d', 10, 9), ('nmod', 5, 10), ('punct', 3, 11)]
```

结论

多做练习，多动手!