INF5830 2015: NumPy and SciPy

Python with extension packages have become one of the preferred tools for data science and machine learning. The packages NumPy and SciPy are backbones of this approach. We have so far mostly made our own implementations and used NLTK. We will now see how some of the same can be done by NumPy and SciPy.

NumPy

is a tool for scientific computing with Python. It adds both functionality and speed. The basic additional brick is the N-dimensional array data type. We will mainly consider one-dimensional arrays. A one-dimensional array is similar to a list but:

- All elements must be of the same type
- It has a fixed length in the sense that we do not append or remove elements from it
- It has additional methods and functionality

We may make a new array in many ways, e.g.

```
>>> import numpy as np
>>> a = range(10)
>>> b = np.array(a)
>>> c = np.arange(10)
>>> b
>>> c
>>> type(a)
>>> type(b)
>>> type(c)
>>> z=np.zeros(200)
>>> z
```

Let us inspect some of the new functionality

```
>>> d = b+c
>>> e = b*c
>>> d
>>> e
>>> b+3
>>> b*3
>>> b**3
>>> g=b/3
>>> g
>>> f = b*1.0
>>> h= f/3
>>> h
>>> h==g
>>> a+b
```

```
>>> k = range(800,900)
>>> a+k
```

Observe how numpy does type cohesion in a+b and transforms a to an array. See how this differs from the list operation in a+k.

Name spaces

In the example we imported numpy as np. It is tempting instead to import everything from numpy, as in

```
>>> import numpy
>>> from numpy import *
>>> a = range(10)
>>> b = array(a)
>>> c = arange(10)
```

This is convenient because it saves us from typing. But there is a danger. If another module uses the same names for classes or functions, we get a name conflict and we cannot access both functions using the same name. For example, numpy has its own random module. This is different from the Python module *random*, and uses some of the same function names with a different interpretation. Thus if we import both *random* and everything from numpy, we may experience a conflict.

It is important to

- Know which name spaces you are using
- Consult the documentation for the functions before you use them.

If we import numpy as np we should be safe.

Some tools for statistics in NumPy

The NumPy array has some built-in methods useful for statistics, e.g. consider the following. If *b* is the np.array from above, try

```
>>> b.mean()
>>> b.var()
>>> b.std()
```

One should beware how the variance and standard deviation is calculated, though. This is the variance and standard deviation from a population, i.e. we divide by *n*. If we are looking for the variance and standard deviation of a sample which we will use for estimating values for a population, i.e., divide by (n– 1), we have to include (ddof=1). Consider the observation {1,2,3}. Calculate its mean, sample variance and sample standard deviation by hand. Then see what the software yields:

```
>>> c = np.array([1,2,3])
>>> c.mean()
>>> c.var()
>>> c.std()
>>> c.var(ddof=1)
```

scipy.stats

When doing statistics one needs access to tables or statistical software. Tables were the old way. They were originally calculated by humans over long time using pen, papers and numerical methods. Today, we have software which gives us immediate access to statistical connections, e.g. between z-scores and p-values for the normal distribution. There are many different types of statistical software and it is integrated into different applications and devices including: calculators, spreadsheets, various statistical packages and dedicated programs including e.g. minitab and R. we have chosen to use scipy.stats for various reasons:

- It is integrated into python which means that we have access to it in the same environment where we work with texts using e.g. NLTK
- It uses the same syntax as python. We do not have to learn a new language, which is the case with e.g. R
- Though it is not as developed as some of the more specialized programs, it (more than) suffices for our purposes
- It is open source.

Here are just some hints. For more documentation, consult
http://docs.scipy.org/doc/scipy/reference/stats.html

SciPy is built on top of NumPy and extends NumPy with various modules. That NumPy is a part of SciPy includes name space, all NumPy functions are available in SciPy under the same name.

Distributions

The normal distributions and some common methods

The most important part (the part we don't want to make ourselves) is the statistical distributions. We start with the prototypical normal distribution

```
>>> import scipy
>>> from scipy import stats
>>> stats.norm
<scipy.stats.distributions.norm_gen object at 0x1ce23910>
```

This does not say too much. But there are many methods included in the distribution which we may explore by using autocompletion from `stats.norm`. (in idle, ipython or canopy). The most important is the cumulative density function.

```
>>> stats.norm.cdf(-1)
array(0.15865525393145707)
>>> stats.norm.cdf(2)
array(0.97724986805182079)
```

The examples show the percentage below standard deviation -1 and below standard deviation 2. To find out how large percentage of the population which is between standard deviation -1 and 1, we use:

```
>>> stats.norm.cdf(1)-stats.norm.cdf(-1)
0.68268949213708585
```

The number should be familiar by now. To check the percentage above standard deviation 2, we may use

```
>>> 1-stats.norm.cdf(2)
0.022750131948179209
```

There is even a separate function for this, the survival function:

```
>>> stats.norm.sf(2)
array(0.022750131948179209)
```

These examples take us from z to p. How do we go in the other direction? We use the percentage function. Say we want to find out the z-value such that 5 percent are above that value; or the z-value such that 2.5% are above the value. The following will do:

```
>>> stats.norm.ppf(.95)
array(1.6448536269514722)
>>> stats.norm.ppf(.975)
array(1.959963984540054)
```

These numbers should be familiar by now – at least the last one which we normally round off to 1.96. There are also other methods, like the probability density function

```
>>> stats.norm.pdf(0)
array(0.3989422804014327)
```

So far, the standard normal distribution. But the scipy.stats.norm distribution is in fact parametric, it takes 3 arguments. The following 2 commands are equivalent

```
>>> stats.norm.cdf(2)
array(0.97724986805182079)
>>> stats.norm.cdf(2, loc=0, scale=1)
array(0.97724986805182079)
```

loc=0 and scale=1 are default values. For the stats.norm distribution in scipy.stats, loc is mean, and scale is standard deviations. For other distributions, the same names, loc and scale, are used for other parameters. We may then use the distribution of young men, N(180,6), cf. the examples from the lectures. The probability of being above 190 cm and 200 cm are then, respectively

```
>>> stats.norm.sf(190, loc=180, scale=6)
array(0.047790352272814696)
>>> stats.norm.sf(200, loc=180, scale=6)
array(0.0004290603331967846)
```

The percentage method works equally well in the general case as for the standard distribution.

```
>>> stats.norm.ppf([0.25,0.5,0.75])
array([-0.67448975,  0.        ,  0.67448975])
>>> stats.norm.ppf([0.1*i for i in range(1,10)], 180,6)
array([ 172.31069061,  174.9502726 ,  176.85359692,  178.47991738,
        180.        ,  181.52008262,  183.14640308,  185.0497274 ,
        187.68930939])
```

The first argument may be a list. The method calculates several values. In the first example we get the quartiles, in the second example, the 10-percentiles. The expected type of the first argument is an array. If it is a list, it is transformed to an array. If it is a number, it is transformed to a one element array.

The binomial distribution

To specify the binomial distribution B(n,p), we need two parameters, n and p. There is no default here, we have to specify them. The probability mass function, b(k; n,p) can then be expressed as follows (where k=4, n=9, p=0.5)

```
>>> stats.binom.pmf(4,9,0.5)
array(0.24609375000000011)
```

(You may check that the number is correct

```
>>> 9*8*7*6/(1*2*3*4)
126.0
>>> 126/2**9
0.24609375)
```

We can calculate the cumulative density function similarly to the continuous case.

```
>>> stats.binom.cdf(4,9,0.5)
array(0.50000000000000011)
```

(Don’t ask me why the result isn’t 0.5.)

In newer versions of python+scipy. , there is an inverse percentage function

```
>>> stats.binom.ppf(.75,9,0.5)
Out[193]: 6.0
```

The binomial distribution also includes methods for variance and standard deviation. Try

```
>>> stats.binom.var(20, 0.5)
>>> stats.binom.std(20,0.5)
```

The geometric distribution

To calculate how many throws we need to throw a dice before we get a six, we use the geometric distribution. It needs one parameter for the probability (i.e. 1/6 for a dice and ½ for a fair coin).

```
stats.geom.pmf(1, 1.0/6)
Out[70]: 0.16666666666666666

In [71]: stats.geom.pmf(0, 1.0/6)
Out[71]: 0.0

In [72]: stats.geom.pmf(4, 1.0/6)
Out[72]: 0.096450617283950629
```

Other discrete distributions

There are also several other discrete distributions, including the uniform distribution `randint(x, min, max)` which ascribes a uniform probability to the numbers {min, min+1, min+2, …, max-1}

```
>>> stats.randint.pmf(17,10,20)
array(0.10000000000000001)
>>> stats.randint.cdf(17,10,20)
array(0.80000000000000004)
```

The T-distribution

The T-distribution is implemented similarly to the normal distribution, but it takes an extra argument for the degrees of freedom which has to be included before the loc, and scale arguments

```
>>> stats.norm.sf(190, loc=180, scale=6)
array(0.047790352272814696)
>>> stats.t.sf(190,100, loc=180, scale=6)
array(0.049354985476065094)
>>> stats.t.sf(190,10, loc=180, scale=6)
```

```
array(0.063273664459106316)
>>> stats.t.sf(190,6, loc=180, scale=6)
array(0.073314785210016997)
>>> stats.t.cdf(-2,10)
array(0.036694017385370196)
```

(In the last example, 10 is the degrees of freedom).

There are more than 50 different continuous distributions included in scipy.stats.

The end (for now.)

We will look more on the scipy.stats module later.