INF5830 2013: scipy.stats

Statistical software

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-values 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 were 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

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 connect to this 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 how many percentage which are 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. Ho 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)
```

Numbers that 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 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 for 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])
```

Observe that the first argument is 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 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).

The c^2-distribution (chi square)

This is also included. As with the T-distribution there is an obligatory argument for the degrees of freedom, here 1.

```
>>> stats.chi2.cdf(1.55, 1)
array(0.78686455912603148)
>>> stats.chi2.ppf(.95,1)
array(3.8414588206941236)
>>> stats.chi2.sf(3.84,1)
array(0.050043521248705189)
```

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

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 do the cumulative density function similarly to the continuous case.

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

(Don't ask my 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
```

This didn't work on the old IFI Redhat 5 installation, I haven't checked it on the RedHat 6 yet.

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)
```

Tests

T-tests

With the distributions available, it is not too difficult to perform statistical tests. For example, if we have a sample X we may explore how probable it is that it is drawn from a population with a known mean m.

```
>>> X = np.array([181, 176, 174, 179, 187, 182, 177, 183, 172, 177])
>>> m = 180
>>> N = len(X)
>>> xbar = X.mean()
>>> xbar
178.80000000000001
>>> s = X.std(ddof=1)
>>> s
4.5166359162544856
>>> t = (xbar - m)/(s/np.sqrt(N))
>>> stats.t.cdf(t,9)
0.21128581406710023
```

But scipy.stats also has included a function which does all this for us:

```
>>> stats.ttest_1samp(X,180)
(-0.84016805041679798, 0.42257162813420018)
```

Why aren't the numbers the same?

There is also a built in function for the two-sided t-test:

```
>>> Y = X+1
>>> stats.ttest_ind(X,Y)
(array(-0.49507377148833714), 0.62653710799051554)
```

And a function for the paired t-test. We return to this after obligatory assignment 2.

c^2-tests (chi square)

To carry out the cumbersome chi-square test for an m*n-table, in newer installations there is a function which does that for us.

```
>>> O = [[8,4667],[15820, 14287173]]
>>> stats.chi2_contingency(O)
Out[216]:
(1.0496327485633206,
 0.30559168331780329,
 1L,
 array([[  5.17176524e+00,   4.66982823e+03],
       [  1.58228282e+04,   1.42871702e+07]]))
```

Here the out-values are

- The score chi2 (1.0496327485633206)
- The p-value (0.30559168331780329)
- The degrees of freedom (1)
- The expectations (array([[5.17176524e+00, 4.66982823e+03],
  [1.58228282e+04, 1.42871702e+07]]))

Again, unfortunately, I didn't find the chi2_contingency under the Redhat 5 installations, and haven't checked the Redhat 6 installations.