1 Probability and Statistics Functions

1.1 Random Number Generation with NumPy

NumPy random number generators are all stored in the module `numpy.random`. These can be imported with using `import numpy as np` and then calling `np.random.rand`, for example, or by importing `import numpy.random as rnd` and `using rnd.rand.1`.

1.1.1 `rand`, `random_sample`

`rand` and `random_sample` are uniform randomnumber generators whichare identicalexceptthat rand takes a variable number of integer inputs – one for each dimension – while `random_sample` takes a n-element tuple.

`random_sample` is the preferred NumPy function, and `rand` is a convenience function primarily for *MATLAB* users.

```
>>> x = rand(3,4,5)
>>> y = random_sample((3,4,5))
```

1.1.2 `randn, standard_normal`

randn and standard_normal are standard normal random number generators. randn, like rand, takes a variable number of integer inputs, and standard_normal takes an n-element tuple. Both can be called with no arguments to generate a single standard normal (e.g. randn()). `standard_normal` is the preferred NumPy function, and randn is a convenience function primarily for MATLAB users .

```
>>> x = randn(3,4,5)
>>> y = standard_normal((3,4,5))
```

1.1.3 `randint, random_integers`

`randint` and `random_integers` are uniform integer random number generators which take 3 inputs, low, high and size. Low is the lower bound of the integers generated, high is the upper and size is a n-element tuple. `randint` and `random_integers` differ in that `randint` generates integers exclusive of the value in high (as do most Python functions), while `random_integers` includes the value in high in its range.

```
>>> x = randint(0,10,(100))
>>> x.max() # Is 9 since range is [0,10)
9
>>> y = random_integers(0,10,(100))
>>> y.max() # Is 10 since range is [0,10]
10
```

1.1.4 `shuffle`

shuffle randomly reorders the elements of an array in place.

```
>>> x = arange(10)
>>> shuffle(x)
>>> x
array([4, 6, 3, 7, 9, 0, 2, 1, 8, 5])
```

1.1.5 `permutation`

permutation returns randomly reordered elements of an array as a copy while not directly changing the input.

```
>>> x = arange(10)
>>> permutation(x)
array([2, 5, 3, 0, 6, 1, 9, 8, 4, 7])
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

NumPy provides a large selection of random number generators for specific distribution. All take between 0 and 2 required inputs which are parameters of the distribution, plus a tuple containing the size of the output. All random number generators are in the module numpy.random.

1.2 Simulation and Random Number Generation

Computer simulated random numbers are usually constructed from very complex but ultimately deterministic functions. Because they are not actually random, simulated random numbers are generally described to as **pseudo-random**.

All pseudo-random numbers inNumPy use one core random number generator based on the ***Mersenne Twister***, a generator which can produce a very long series of pseudo-random data before repeating (up to $2^1 9937 - 1$ non-repeating values) .

1.2.1 `RandomState`

`RandomState` is the class used to control the random number generators. Multiple generators can be initialized by RandomState.

```
>>> gen1 = np.random.RandomState()
>>> gen2 = np.random.RandomState()
>>> gen1.uniform() # Generate a uniform
0.6767614077579269
>>> state1 = gen1.get_state()
>>> gen1.uniform()
0.6046087317893271
>>> gen2.uniform() # Different, since gen2 has different seed
0.04519705909244154
>>> gen2.set_state(state1)
>>> gen2.uniform() # Same uniform as gen1 after assigning state
0.6046087317893271
```

1.2.2 `State`

Pseudo-random number generators track a set of values known as the *state*. The state is usually a vector which has the property that if two instances of the same pseudo-random number generator have the same state, the sequence of pseudo-random numbers generated will be identical. The state of the default random number generator can be read using `numpy.random.get_state` and can be restored using `numpy.random.set_state`.

```
>>> st = get_state()
>>> randn(4)
array([ 0.37283499, 0.63661908, 1.51588209,
1.36540624])
>>> set_state(st)
>>> randn(4)
array([ 0.37283499, 0.63661908, 1.51588209,
1.36540624])
```

The two sequences are identical since they the state is the same when `randn` is called. The state is a 5- element tuple where the second element is a 625 by 1 vector of unsigned 32-bit integers. In practice the state should only be stored using get_state and restored using `set_state`.

1.2.3 `get_state`

`get_state()` gets the current state of the random number generator, which is a 5-element tuple. It can be called as a function, in which case it gets the state of the default random number generator, or as a method on a particular instance of RandomState.

`set_state`

`set_state(state)` sets the state of the random number generator. It can be called as a function, in which case it sets the state of the default random number generator, or as a method on a particular instance of RandomState. `set_state` should generally only be called using a state tuple returned by get_state.

`seed`

`numpy.random.seed` is a more useful function for initializing the random number generator, and can be used in one of two ways. `seed()` will initialize (or reinitialize) the random number generator using some actual random data provided by the operating system.

`seed( s )` takes a vector of values (can be scalar) to initialize the random number generator at particular state. seed(s) is particularly useful for producing simulation studies which are reproducible. In the following example, calls to `seed()` produce different random numbers, since these reinitialize using random data from the computer, while calls to `seed(0)` produce the same (sequence) of random numbers.

```
>>> seed()
>>> randn()
array([ 0.62968838])
>>> seed()
>>> randn()
array([ 2.230155])
>>> seed(0)
>>> randn()
array([ 1.76405235])
>>> seed(0)
>>> randn()
array([ 1.76405235])
```

NumPy always calls `seed()` when the first randomnumberis generated. As a result. calling `standard_normal()` across two “fresh” sessions will not produce the same random number.

1.3 Statistics Functions

`mean`

`mean` computes the average of an array. An optional second argument provides the axis to use (default is to use entire array). `mean` can be used either as a function or as a method on an array.

```
>>> x = arange(10.0)
>>> x.mean()
4.5
>>> mean(x)
4.5
>>> x= reshape(arange(20.0),(4,5))
>>> mean(x,0)
231
array([ 7.5, 8.5, 9.5, 10.5, 11.5])
>>> x.mean(1)
array([ 2., 7., 12., 17.])
```

`median`

`median` computed the median value in an array. An optional second argument provides the axis to use (default is to use entire array).

```
>>> x= randn(4,5)
>>> x
array([[0.74448693,
0.63673031,
0.40608815,
0.40529852, 0.93803737],
[ 0.77746525, 0.33487689, 0.78147524, 0.5050722
, 0.58048329],
[0.51451403,
0.79600763,
0.92590814, 0.53996231,
0.24834136],
[0.83610656,
0.29678017, 0.66112691,
0.10792584, 1.23180865]])
```

```
>>> median(x)
0.45558017286810903
>>> median(x, 0)
array([0.62950048,
0.16997507,
0.18769355, 0.19857318,
0.59318936])
```

Note that when an array or axis dimension contains an even number of elements (n), median returns the average of the 2 inner elements.

`std`

`std` computes the standard deviation of an array. An optional second argument provides the axis to use (default is to use entire array). std can be used either as a function or as a method on an array.

`var`

`var` computes the variance of an array. An optional second argument provides the axis to

`corrcoef`

`corrcoef(x)` computes the correlation between the rows of a 2-dimensional array x . `corrcoef(x, y)` computes the correlation between two 1- dimensional vectors. An optional keyword argument rowvar can be used to compute the correlation between the columns of the input – this is corrcoef(x, rowvar=False) and `corrcoef(x.T)` are identical

```
>>> x= randn(3,4)
>>> corrcoef(x)
array([[ 1. , 0.36780596, 0.08159501],
[ 0.36780596, 1. , 0.66841624],
[ 0.08159501, 0.66841624, 1. ]])
>>> corrcoef(x[0],x[1])
array([[ 1. , 0.36780596],
[ 0.36780596, 1. ]])
>>> corrcoef(x, rowvar=False)
array([[ 1. , 0.98221501,
0.19209871,
0.81622298],
[0.98221501,
1. , 0.37294497, 0.91018215],
[0.19209871,
0.37294497, 1. , 0.72377239],
[0.81622298,
0.91018215, 0.72377239, 1. ]])
>>> corrcoef(x.T)
array([[ 1. , 0.98221501,
0.19209871,
0.81622298],
[0.98221501,
1. , 0.37294497, 0.91018215],
[0.19209871,
0.37294497, 1. , 0.72377239],
[0.81622298,
0.91018215, 0.72377239, 1. ]])
```

`cov`

`cov(x)` computes the covariance of an array x . `cov(x,y)` computes the covariance between two 1-dimensional vectors. An optional keyword argument rowvar can be used to compute the covariance between the columns of the input – this is `cov(x, rowvar=False)` and `cov(x.T)` are identical.

`histogram`

`histogram` can be used to compute the histogram (empirical frequency, using k bins) of a set of data. An optional second argument provides the number of bins. If omitted, `k =10` bins are used. histogram returns two outputs, the first with a k-element vector containing the number of observations in each bin, and the second with the k + 1 endpoints of the k bins.

```
>>> x = randn(1000)
>>> count, binends = histogram(x)
>>> count
array([ 7, 27, 68, 158, 237, 218, 163, 79, 36, 7])
>>> binends
array([3.06828057,
2.46725067,
1.86622077,
1.26519086,
0.66416096,
0.06313105,
0.53789885, 1.13892875, 1.73995866, 2.34098856,
2.94201846])
>>> count, binends = histogram(x, 25)
```

`histogram2d`

`histogram2d(x,y)` computes a 2-dimensional histogram for 1-dimensional vectors. An optional keyword argument bins provides the number of bins to use. bins can contain either a single scalar integer or a 2-element list or array containing the number of bins to use in each dimension.

1.4 Distributions

1.4.1 `normal`

`normal()` generates draws from a standard Normal (Gaussian). `normal(mu, sigma)` generates draws from a Normal with mean μ and standard deviation σ. `normal(mu, sigma, (10,10))` generates a 10 by 10 array of draws from a Normal with mean μ and standard deviation σ.

`normal(mu, sigma)` is equivalent to `mu + sigma * standard_normal()`.

1.4.2 `poisson`

`poisson()` generates a draw from a Poisson distribution with $\lambda = 1$. `poisson(lambda)` generates a draw from a Poisson distribution with expectation λ. poisson(lambda, (10,10)) generates a 10 by 10 array of draws from a Poisson distribution with expectation λ.

1.4.3 `standard_t`

`standard_t(nu)` generates a draw from a Student's t with shape parameter ν. `standard_t(nu, (10,10))` generates a 10 by 10 array of draws from a Student's t with shape parameter ν.

1.4.4 `uniform`

`uniform()` generates a uniform random variable on (0, 1). uniform(low, high) generates a uniform on (l , h). `uniform(low, high, (10,10))` generates a 10 by 10 array of uniforms on (l , h).

1.5 Continuous Random Variables

SciPy contains a large number of functions for working with continuous random variables. Each function resides in its own class (e.g. norm for Normal or gamma for Gamma), and classes expose methods for random number generation, computing the PDF, CDF and inverse CDF, fitting parameters using MLE, and computing various moments. The methods are listed below, where dist is a generic placeholder for the distribution name in SciPy.

- `dist.rvs`
  Pseudo-randomnumbergeneration. Generically, rvs is called using dist.rvs(*args, loc=0, scale=1, size=size) where size is an n-element tuple containing the size of the array to be generated.

- `dist.pdf`
  Probability density function evaluation for an array of data (element-by-element). Generically, pdf is called using `dist.pdf(x, *args, loc=0, scale=1)` where x is an array that contains the values to use when evaluating PDF.

- `dist.logpdf`
  Log probability density function evaluation for an array of data (element-by-element). Generically, logpdf is called using dist.logpdf(x, *args, loc=0, scale=1) where x is an array that contains the values to use when evaluating log PDF.

- `dist.cdf`
  Cumulative distribution function evaluation for an array of data (element-by-element). Generically, cdf is called using `dist.cdf(x, *args, loc=0, scale=1)` where x is an array that contains the values to use when evaluating CDF.

- `dist.ppf`
  Inverse CDF evaluation (also known as percent point function) for an array of values between 0 and 1. Generically, ppf is called using `dist.ppf(p, *args, loc=0, scale=1)` where p is an array with all elements between 0 and 1 that contains the values to use when evaluating inverse CDF.

- `dist.fit`
  Estimate shape, location, and scale parameters from data by maximum likelihood using an array of data. Generically, fit is called using `dist.fit(data, *args, floc=0, fscale=1)` where data is a data array used to estimate the parameters. floc forces the location to a particular value (e.g. floc=0). `fscale` similarly forces the scale to a particular value (e.g. `fscale=1`) . It is necessary to use floc and/or fscale when computing MLEs if the distribution does not have a location and/or scale. For example, the gamma distribution is defined using 2 parameters, often referred to as shape and scale. In order to useMLto estimate parameters from a gamma, floc=0 must be used.

- `dist.median`
  Returns the median of the distribution. Generically, median is called using dist.median(*args, loc=0, scale=1).
- `dist.mean`
  Returns the mean of the distribution. Generically, mean is called using dist.mean(*args, loc=0, scale=1).
- `dist.moment`
  nth non-centralmomentevaluation of the distribution. Generically, moment is called using dist.moment(r, *args, loc=0, scale=1) where r is the order of the moment to compute.
- `dist.varr`
  Returns the variance of the distribution. Generically, var is called using dist.var(*args, loc=0, scale=1).
- `dist.std`
  Returns the standard deviation of the distribution. Generically, std is called using dist.std(*args, loc=0, scale=1).

1.5.1 Example

The gamma distribution is used as an example. The gamma distribution takes 1 shape parameter a (a is the only element of *args), which is set to 2 in all examples.

```
>>> import scipy.stats as stats
>>> gamma = stats.gamma
>>> gamma.mean(2), gamma.median(2), gamma.std(2), gamma.var(2)
(2.0, 1.6783469900166608, 1.4142135623730951, 2.0)
>>> gamma.moment(2,2) gamma.
moment(1,2)**2 # Variance
>>> gamma.cdf(5, 2), gamma.pdf(5, 2)
(0.95957231800548726, 0.033689734995427337)
>>> gamma.ppf(.95957231800548726, 2)
5.0000000000000018
>>> log(gamma.pdf(5, 2)) gamma.
logpdf(5, 2)
0.0
>>> gamma.rvs(2, size=(2,2))
array([[ 1.83072394, 2.61422551],
[ 1.31966169, 2.34600179]])
>>> gamma.fit(gamma.rvs(2, size=(1000)), floc = 0) # a, 0, shape
```

```
(2.209958533078413, 0, 0.89187262845460313)
```

1.6 More Statistical Functions

`mode`

mode computes the mode of an array. An optional second argument provides the axis to use (default is to use entire array). Returns two outputs: the first contains the values of the mode, the second contains the number of occurrences.

```
>>> x=randint(1,11,1000)
>>> stats.mode(x)
(array([ 4.]), array([ 112.]))
```

`moment`

moment computed the rth central moment for an array. An optional second argument provides the axis to use (default is to use entire array).

```
>>> x = randn(1000)
>>> moment = stats.moment
>>> moment(x,2) moment(
x,1)**2
0.94668836546169166
>>> var(x)
0.94668836546169166
>>> x = randn(1000,2)
>>> moment(x,2,0) # axis 0
array([ 0.97029259, 1.03384203])
```

`skew`

skew computes the skewness of an array. An optional second argument provides the axis to use (default is to use entire array).

```
>>> x = randn(1000)
>>> skew = stats.skew
>>> skew(x)
0.027187705042705772
>>> x = randn(1000,2)
>>> skew(x,0)
array([ 0.05790773, 0.00482564])
```

`kurtosis`

kurtosis computes the excess kurtosis (actual kurtosis minus 3) of an array. An optional second argument provides the axis to use (default is to use entire array). Setting the keyword argument fisher=False will compute the actual kurtosis.

```
>>> x = randn(1000)
>>> kurtosis = stats.kurtosis
>>> kurtosis(x)
0.2112381820194531
>>> kurtosis(x, fisher=False)
2.788761817980547
>>> kurtosis(x, fisher=False) kurtosis(
x) # Must be 3
3.0
>>> x = randn(1000,2)
>>> kurtosis(x,0)
array([0.13813704,
0.08395426])
```

`pearsonr`

pearsonr computes the Pearson correlation between two 1-dimensional vectors. It also returns the 2- tailed p-value for the null hypothesis that the correlation is 0.

```
>>> x = randn(10)
>>> y = x + randn(10)
>>> pearsonr = stats.pearsonr
>>> corr, pval = pearsonr(x, y)
>>> corr
0.40806165708698366
>>> pval
0.24174029858660467
```

`spearmanr`

spearmanr computes the Spearmancorrelation (rank correlation). It can be used with a single 2-dimensional array input, or 2 1-dimensional arrays. Takes an optional keyword argument axis indicating whether to treat columns (0) or rows (1) as variables. If the input array has more than 2 variables, returns the correlation matrix. If the input array as 2 variables, returns only the correlation between the variables.

```
>>> x = randn(10,3)
>>> spearmanr = stats.spearmanr
>>> rho, pval = spearmanr(x)
>>> rho
array([[ 1. , 0.02087009,
0.05867387],
[0.02087009,
1. , 0.21258926],
[0.05867387,
0.21258926, 1. ]])
>>> pval
array([[ 0. , 0.83671325, 0.56200781],
[ 0.83671325, 0. , 0.03371181],
[ 0.56200781, 0.03371181, 0. ]])
>>> rho, pval = spearmanr(x[:,1],x[:,2])
>>> corr
0.020870087008700869
>>> pval
0.83671325461864643
```

`linregress`

linregress estimates a linear regression between 2 1-dimensional arrays. It takes two inputs, the independent variables (regressors) and the dependent variable (regressand). Models always include a constant.

```
>>> x = randn(10)
>>> y = x + randn(10)
>>> linregress = stats.linregress
>>> slope, intercept, rvalue, pvalue, stderr = linregress(x,y)
>>> slope
1.6976690163576993
>>> rsquare = rvalue**2
>>> rsquare
0.59144988449163494
>>> x.shape = 10,1
>>> y.shape = 10,1
>>> z = hstack((x,y))
>>> linregress(z) # Alternative form, [x y]
(1.6976690163576993,
0.79983724584931648,
0.76905779008578734,
0.0093169560056056751,
0.4988520051409559)
```

1.7 Select Statistical Tests

`normaltest`

`normaltest` tests for normality in an array of data. An optional second argument provides the axis to use (default is to use entire array). Returns the test statistic and the p-value of the test. This test is a small sample modified version of the Jarque-Bera test statistic.

`kstest`

`kstest` implements the Kolmogorov-Smirnov test. Requires two inputs, the data to use in the test and the distribution, which can be a string or a frozen random variable object. If the distribution is provided as a string, then any required shape parameters are passed in the third argument using a tuple containing these parameters, in order.

```
>>> x = randn(100)
>>> kstest = stats.kstest
>>> stat, pval = kstest(x, 'norm')
>>> stat
0.11526423481470172
>>> pval
```

```
0.12963296757465059
>>> ncdf = stats.norm().cdf # No () on cdf to get the function
>>> kstest(x, ncdf)
(0.11526423481470172, 0.12963296757465059)
>>> x = gamma.rvs(2, size = 100)
>>> kstest(x, 'gamma', (2,)) # (2,) contains the shape parameter
(0.079237623453142447, 0.54096739528138205)
>>> gcdf = gamma(2).cdf
>>> kstest(x, gcdf)
(0.079237623453142447, 0.54096739528138205)
```

1.7.1 `ks_2samp`

`ks_2samp` implements a 2-sample version of the Kolmogorov-Smirnov test. It is called `ks_2samp(x,y)` where both inputs are 1-dimensonal arrays, and returns the test statistic and p-value for the null that the distribution of x is the same as that of y .

1.7.2 `shapiro`

`shapiro` implements the Shapiro-Wilk test for normality on a 1-dimensional array of data. It returns the test statistic and p-value for the null of normality.

2 Statsmodels

`Statsmodels` is a Python module that allows users to explore data, estimate statistical models, and perform statistical tests. An extensive list of descriptive statistics, statistical tests, plotting functions, and result statistics are available for different types of data and each estimator. Researchers across fields may find that statsmodels fully meets their needs for statistical computing and data analysis in Python.

Features include:

- Linear regression models
- Generalized linear models
- Discrete choice models
- Robust linear models
- Many models and functions for time series analysis
- Nonparametric estimators
- A collection of datasets for examples
- A wide range of statistical tests
- Input-output tools for producing tables in a number of formats (Text, LaTex, HTML) and for reading Stata files into NumPy and Pandas.
- Plotting functions
- Extensive unit tests to ensure correctness of results
- Many more models and extensions in development