Lecture Notes 3: Sampling, Simulation, Low-Level Extensions

Sampling from Distributions

```
In [1]: import numpy
        import numpy.random
```

Sampling from a uniform distribution between 0 and 1

```
In [2]: numpy.random.uniform(0, 1, 10)
```

```
Out[2]: array([ 0.17606033,  0.00494842,  0.28319008,  0.34096885,  0.52294137,
                0.67754303,  0.22497136,  0.434496  ,  0.14027741,  0.42131596])
```

Sampling from normal distibution of mean 10 and standard deviation 0.01

```
In [3]: numpy.random.normal(10, 0.01, 10)
```

```
Out[3]: array([ 10.01266723,  10.00810526,   9.99967544,  10.01518245,
                10.00915431,  10.00015659,   9.98994313,  10.00769405,
                 9.97010107,  10.00185344])
```

Demo 1: Testing A Statistical Paradox (the James-Stein Paradox)

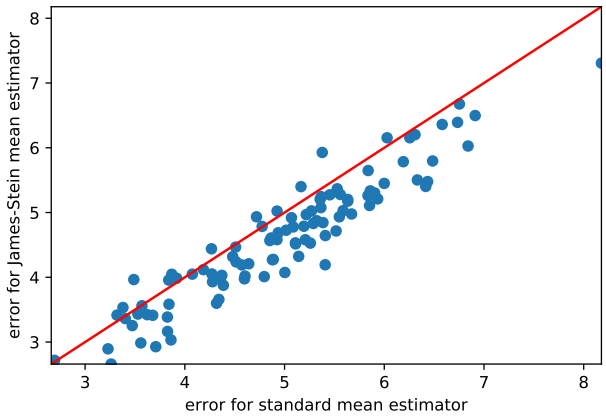
When estimating the mean of a data distribution, moving the estimator away from the empirical mean and closer to some arbitrary point (e.g. the origin) makes the estimator more accurate. Let's verify it:

```
In [4]: # Testing many times:
        errstd  = list()
        errstd2 = list()

        n = 10  # number of samples
        d = 50  # dimensionality of feature space

        for i in range(100):
            # sample from a distribution of mean 1 and standard deviation 1
            m = 1.0

            X = numpy.random.normal(m, 1, (n,d))

            # empirical mean
            m_emp = X.mean(axis=0)

            # some coefficient
            c = (1 - ((d - 2) / n) / ((m_emp) ** 2).sum())

            # james-stein estimator
            m_js = c * m_emp

            # the error between the true mean and the standard estimator
            errstd.append(((m - m_emp) ** 2).sum())

            # the error between the true mean and the said better estimator
            errstd2.append(((m - m_js) ** 2).sum())
```

In [5]:
```
import matplotlib
from matplotlib import pyplot as plt
%matplotlib inline

# Export options
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('pdf', 'png')
plt.rcParams['savefig.dpi'] = 90
```

In [6]:
```
plt.scatter(errstd, errstd2)

# Draw a line where they are equal
l = min(errstd + errstd2)
h = max(errstd + errstd2)
plt.plot([l, h], [l, h], color='red')

plt.xlabel('error for standard mean estimator')
plt.ylabel('error for James-Stein mean estimator')
plt.axis([l, h, l, h])
```

Out[6]: [2.6603059313097819, 8.177779952489022, 2.6603059313097819, 8.177779952489022]



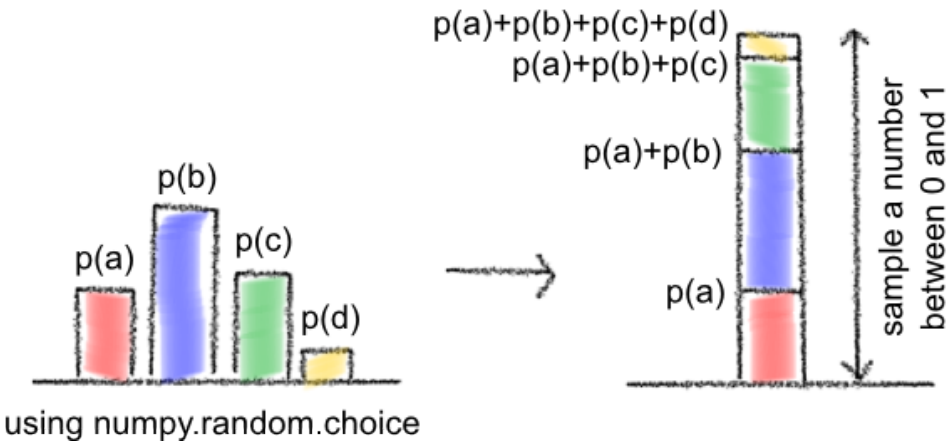
Observation: most points are below the curve (i.e. James-Stein estimator is better).

Making discrete choices

Let us suppose we have 7 fruits to choose from:

In [7]:
```
fruits = ['watermelon', 'apple', 'grape', 'grapefruit', 'lemon', 'banana', 'cherry']
```

We use the function random.choice to randomly choose from that list




```
In [8]: import random
        random.choice(fruits)

Out[8]: 'banana'

In [9]: print(random.choice(fruits))
        print(random.choice(fruits))
        print(random.choice(fruits))

banana
banana
lemon
```

The function choice of the module numpy.random provides some more functionalities such as choosing repeatedly

```
In [10]: numpy.random.choice(fruits, 10)

Out[10]: array(['lemon', 'grape', 'grapefruit', 'banana', 'banana', 'cherry',
                'grape', 'banana', 'grapefruit', 'lemon'],
               dtype='<U10')
```

or specify a certain distribution of fruits to choose from

```
In [11]: p = [0.05, 0.70, 0.05, 0.05, 0.05, 0.05, 0.05]
         numpy.random.choice(fruits, 20, p=p)

Out[11]: array(['apple', 'apple', 'apple', 'banana', 'apple', 'lemon', 'apple',
                'apple', 'lemon', 'apple', 'apple', 'apple', 'grape', 'apple',
                'apple', 'apple', 'apple', 'apple', 'apple', 'apple'],
               dtype='<U10')
```

Another way to make discrete choices

```
In [12]: # Define fruits probabilities
         p = [0.05, 0.70, 0.05, 0.05, 0.05, 0.05, 0.05]
```

```
        # Cumulate them
        l = numpy.cumsum([0] + p[:-1]) # lower-bounds
        h = numpy.cumsum(p)            # upper-bounds

        # Draw a number between 0 and 1
        u = numpy.random.uniform(0, 1)

        # Find which basket it belongs to
        s = numpy.logical_and(u > l, u < h)
        print(s)

        # retrieve the label
        fruits[numpy.argmax(s)]
```

```
[False False False False  True False False]
```

Out[12]: 'lemon'

The code can be parallelized to choose multiple fruits at the same time

In [13]:
```
# generate many numbers between 0 and 1
u = numpy.random.uniform(0, 1, 20)

# find the basket to which they belong
na = numpy.newaxis
s = numpy.logical_and(u[:, na] > l[na, :], u[:, na] < h[na, :])

# Broadcasting shapes
print(u[:, na].shape, l[na, :].shape, s.shape)

print([fruits[i] for i in numpy.argmax(s, axis=1)])
```

```
(20, 1) (1, 7) (20, 7)
['apple', 'apple', 'apple', 'apple', 'apple', 'cherry', 'apple', 'apple', 'apple', 'banana', 'grape', 'a
```

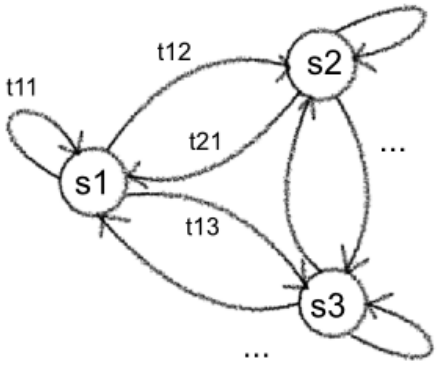
More importantly, the code can be parallelized to choose from multiple distributions at the same time. Let us create a matrix of probability distributions.

In [14]:
```
P = [
    [0.0,0.05,0.05,0.05,0.05,0.05,0.05,0.70],
    [0.0,0.05,0.05,0.05,0.05,0.05,0.70,0.05],
    [0.0,0.05,0.05,0.70,0.05,0.05,0.05,0.05],
]
# (note that we have added a zero-probability state
#  at the beginning for ease of implementation)
```

We stack them, and compute the bounds `L` and `H`

In [15]:
```
C = numpy.cumsum(P,axis=1) # lower-bounds

L, H = C[:, :-1], C[:, 1:]
```

Draw 8 fruits from each distribution, and test the bounds




```
In [16]: R = numpy.random.uniform(0, 1, (3, 8))

         S = numpy.logical_and((R[:, :, na] > L[:, na, :]), (R[:, :, na] < H[:, na, :]))

         # Broadcasting shapes
         print(R[:, :, na].shape, L[:, na, :].shape, S.shape)
```

```
(3, 8, 1) (3, 1, 7) (3, 8, 7)
```

```
In [17]: for s in S:
             print(numpy.argmax(s, axis=1))
             print([fruits[i] for i in numpy.argmax(s, axis=1)])
```

```
[6 6 4 6 0 6 6 6]
['cherry', 'cherry', 'lemon', 'cherry', 'watermelon', 'cherry', 'cherry', 'cherry']
[2 2 4 5 5 5 0 5]
['grape', 'grape', 'lemon', 'banana', 'banana', 'banana', 'watermelon', 'banana']
[5 4 1 2 1 2 3 2]
['banana', 'lemon', 'apple', 'grape', 'apple', 'grape', 'grapefruit', 'grape']
```

Demo 2: Monte Carlo Simulation of a Markov Chain

A Markov chain transits between a set of states, where the transition between pairs of states is associated with a fixed probability. The set of probabilities can be stored in a transition matrix.

```
In [18]: # Transition matrix
         T = numpy.array([
                 [0.9,0.1,0.0], # transiting from state 1 to state 1,2,3
                 [0.0,0.9,0.1], # transiting from state 2 to state 1,2,3
                 [1.0,0.0,0.0], # transiting from state 3 to state 1,2,3
             ])
```

Thanks to our parallel implementation, we can choose from multiple probability distributions and thus simulate multiple Markov chains in parallel, each of them starting in a different state.

```
In [19]: # Add empty state to transition matrix
         pad_shape = ((0, 0), (1, 0))  # ((before_1, after_1), (before_2, after_2))
         P = numpy.pad(T, pad_shape, mode='constant')
         print(P)
```

```
[[ 0.   0.9  0.1  0. ]
 [ 0.   0.   0.9  0.1]
 [ 0.   1.   0.   0. ]]
```

```
In [20]: # Implementing transition for each particle
         def mcstep(X, P):
             Xp = numpy.dot(X, P)

             Xc = numpy.cumsum(Xp, axis=1)

             L,H = Xc[:, :-1], Xc[:, 1:]

             R = numpy.random.uniform(0, 1, (len(Xp), 1))

             return numpy.logical_and((R > L), (R < H))
```

```
In [21]: # Initialize all particles to state 1
         A = numpy.outer(numpy.ones([30]),[1.0,0,0])

         print('initial distribution of particles states: [{:.3f} {:.3f} {:.3f}]'.format(*tuple(A.mean(a

         for i in range(20):
                 print('(iter {:2d})'.format(i,numpy.argmax(A,axis=1)))
                 A = mcstep(A, P)

         # Print distribution of sampled states
         print('final distribution of particles states: [{:.3f} {:.3f} {:.3f}]'.format(*tuple(A.mean(axi
```

```
initial distribution of particles states: [1.000 0.000 0.000]
(iter  0)
(iter  1)
(iter  2)
(iter  3)
(iter  4)
(iter  5)
(iter  6)
(iter  7)
(iter  8)
(iter  9)
(iter 10)
(iter 11)
(iter 12)
(iter 13)
(iter 14)
(iter 15)
(iter 16)
(iter 17)
(iter 18)
(iter 19)
final distribution of particles states: [0.467 0.467 0.067]
```