# Data Mining

**Tamás Budavári** - budavari@jhu.edu
**Class 2**

- Probability Density Function (PDF)
- Cummulative Density Function (CDF)
- Moments

---

# Statistics

### Probability Density Function (PDF)

- Probabiliy of $x$ between $a$ and $b$ for any $(a, b)$ is

$$P_{ab} = \int_a^b p(x)\, dx$$

- Always

$$\int_{-\infty}^{\infty} p(x)\, dx = 1$$

- Example 1: uniform distribution on $(a, b)$

$U(x; a, b) = \frac{\mathbf{1}_{ab}(x)}{b-a}$, where $\mathbf{1}_{ab}(x)$ is 1 between $a$ and $b$, but 0 otherwise

- Example 2: Gaussian or normal distribution

$$G\left(x; \mu, \sigma^2\right) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right]$$

- Example 3: Log-normal distribution

$$LN\left(x; \mu, \sigma^2\right) = \frac{1}{x\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(\ln x-\mu)^2}{2\sigma^2}\right]$$

- Example 4: Exponential distribution $(\lambda > 0)$

$$EXP\,(x; \lambda) = \lambda e^{-\lambda x}, x \geq 0$$

- Gauss on Money!

- Even the formula

Note: All the image files should be stored in the "files" folder. In order to properly display the images, put the ".ipynb" file and the "files" folder under a same folder.

## Cummulative Distribution Function (CDF)

- Integral up to given $x$: prob of being less than $x$

$$\mathrm{CDF}(x) = \int_{-\infty}^{x} p(t)\, dt$$

In [1]:
```
%pylab inline
```

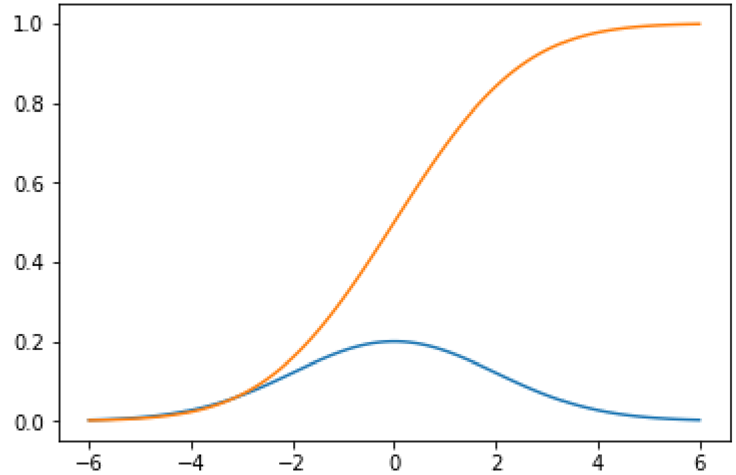
```
Populating the interactive namespace from numpy and matplotlib
```

## Statistical functions (scipy.stats)

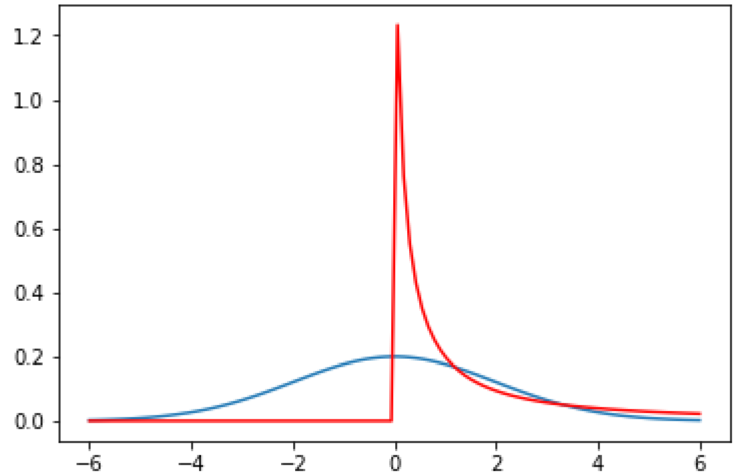
This module contains a large number of probability distributions as well as a growing library of statistical functions.

See https://docs.scipy.org/doc/scipy/reference/stats.html (https://docs.scipy.org/doc/scipy/reference/stats.html) for a complete guide.

In [2]:
```
# Plot the PDF and CDF of Gaussian distribution
from scipy.stats import norm as gaussian
x = np.linspace(-6,6,100) # get 100 evenly spaced numbers over [-6, 6].
mu, sig = 0, 2 # assign multiple values at the same time
plot(x, gaussian.pdf(x,mu,sig));
plot(x, gaussian.cdf(x,mu,sig));
```



In [3]:
```
# Plot the PDF of the Gaussian adn Log-normal distribution
from scipy.stats import lognorm
plot(x, gaussian.pdf(x,0,sig));
plot(x, lognorm.pdf(x,sig), color='r');
```



## Characterization of PDFs

- Expectation value of $X$

$$\mu = \mathbb{E}[X] = \int_{-\infty}^{\infty} x\, p(x)\, dx$$

- Expectation value of any $f(X)$

$$\mathbb{E}[f(X)] = \int_{-\infty}^{\infty} f(x)\, p(x)\, dx$$

- Moments

$$\mathbb{E}[X^k]$$

- Central moments

$$\mathbb{E}\left[(X-\mu)^k\right]$$

- Variance

$$\mathbb{Var}[X] = \mathbb{E}\left[(X-\mu)^2\right]$$

- Standard deviation

$$\sigma = \sqrt{\mathbb{Var}[X]}$$

- Normalized moments

$$\mathbb{E}\left[\left(\frac{X-\mu}{\sigma}\right)^k\right]$$

- Skewness

3rd normalized moment ($k$=3): $\mathbb{E}\left[\left(\frac{X-\mu}{\sigma}\right)^3\right]$

- Kurtosis

4th normalized moment ($k$=4): $\mathbb{E}\left[\left(\frac{X-\mu}{\sigma}\right)^4\right]$

- **Excess Kurtosis**

The excess kurtosis is defined as kurtosis minus 3. More commonly used since the **excess kurtosis of a standard normal distribution is 0**. In other words, the kurtosis of a standard normal distribution is 3.

Kurtosis is sometimes reported as “excess kurtosis".

In [4]:
```
# Mean, Variance, Skewness and Kurtosis of Gaussian
mean, var, skew, kurt = gaussian.stats(mu, sig, moments='mvsk');
mu, sig, mean, var, skew, kurt
```

Out[4]:
```
(0, 2, array(0.0), array(4.0), array(0.0), array(0.0))
```

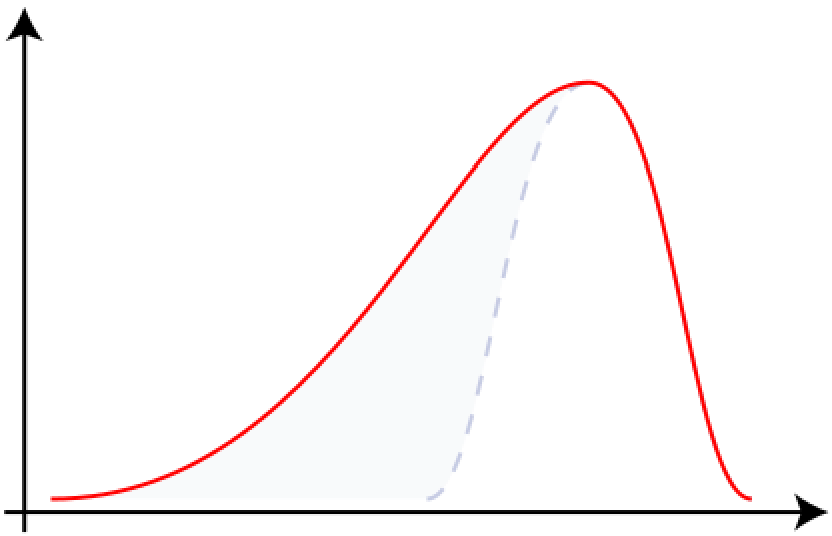
Composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) **excess kurtosis**. Default is 'mv'.

In [5]:
```
# Mean, Variance, Skewness and Kurtosis of Log-normal
mean, var, skew, kurt = lognorm.stats(sig, moments='mvsk');
sig, mean, var, skew, kurt
```

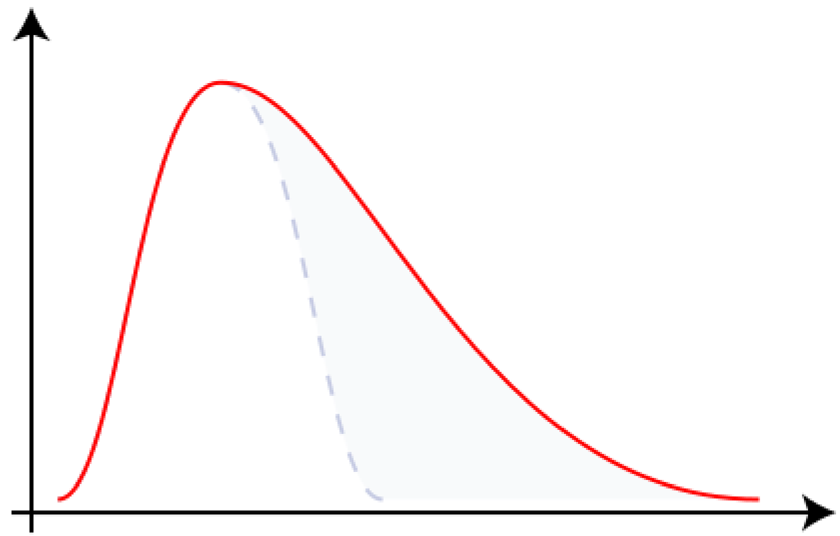
Out[5]:
```
(2,
 array(7.38905609893065),
 array(2926.359837008584),
 array(414.359343300147),
 array(9220556.977307005))
```

**Negative skew**: The left tail is longer; the mass of the distribution is concentrated on the right of the figure. The distribution is said to be left-skewed, left-tailed, or skewed to the left, despite the fact that the curve itself appears to be skewed or leaning to the right; left instead refers to the left tail being drawn out and, often, the mean being skewed to the left of a typical center of the data. A left-skewed distribution usually appears as a right-leaning curve.

**Positive skew**: The right tail is longer; the mass of the distribution is concentrated on the left of the figure. The distribution is said to be right-skewed, right-tailed, or skewed to the right, despite the fact that the curve itself appears to be skewed or leaning to the left; right instead refers to the right tail being drawn out and, often, the mean being skewed to the right of a typical center of the data. A right-skewed distribution usually appears as a left-leaning curve.



Negative Skew



Positive Skew

# Python by Examples

## Data Structure

In [6]:
```
# Tuple
t = (1,2)
t = 100, 0.1
N, mu = t
N
```

Out[6]: 100

In [7]:
```
# List
l = [1,2,3,4,5]
[l,l]
```

Out[7]: [[1, 2, 3, 4, 5], [1, 2, 3, 4, 5]]

## Function

In [8]:
```
def f(x,k=2):
    return x**k
```

In [9]:
```
f3 = f(3)
print (f3)
f(2), f(2,2), f(2,3), f(2,k=4), f3
```

```
9
```

Out[9]:
```
(4, 4, 8, 16, 9)
```

## Object-Oriented Programming

In [10]:
```
import math
```

In [11]:
```
class Robot(object):

    def __init__(self, x=0, y=0, angle=0):
        self.x, self.y, self.angle = x, y, angle
        self.path = [(x,y)]

    def move(self, l):
        self.x += l* math.cos(self.angle)
        self.y += l* math.sin(self.angle)
        self.path.append( (self.x, self.y) )

    def left(self, a):
        self.angle += a

    def right(self, a):
        self.left(-a)
```

In [12]:
```
r = Robot(100,0,np.pi/2)
r.move(10)
r.left(math.pi/4)
r.move(5)
r.path
```

Out[12]:
```
[(100, 0), (100.0, 10.0), (96.46446609406726, 13.535533905932738)]
```

## Others

In [13]:
```
import sys
```

In [14]:
```
sys.stdout.write('asdf')
sys.stdout.write('fdasfsad')
```

```
asdffdasfsad
```

In [15]:
```
out = open('test.txt', 'w')
# Loops
for i in range(10):
    out.write (str(i*i) + ' ')
print ('done')
```

```
done
```

In [16]:
```
# Lambda expressions
g = lambda x: x*x
g(2)
```

Out[16]:
```
4
```

In [17]:
```
# Using math functions and routines
math.pi, math.sin(1.57)
```

Out[17]:
```
(3.141592653589793, 0.9999996829318346)
```

In [18]:
```
# Same using numpy's methods
np.pi, np.sin(1.57), np.sin([0,np.pi,1.57])
```

Out[18]:
```
(3.141592653589793,
 0.99999968293183461,
 array([  0.00000000e+00,   1.22464680e-16,   9.99999683e-01]))
```

In [19]:
```
# numpy methods work also on arrays, e.g., elementwise
np.sin( [1.57, 3.14, np.pi] )
```

Out[19]:
```
array([  9.99999683e-01,   1.59265292e-03,   1.22464680e-16])
```

## Linear Algebra

In [20]:
```
import numpy as np
# Arrays: vectors and matrices
l = [1,2,3]
A = np.array([l,l], dtype=np.int32) # Generate a 2-by-3 matrix
print (A.shape)
print (A.T) # matrix transpose, same as A.transpose()

b = A.T.dot(A) # matrix product
print("b =", b)

c = A.T @ (A) # another matrix product
print("c =", c)

d = A * A # elementwise product (shape must match)
print("d =", d)
```

```
(2, 3)
[[1 1]
 [2 2]
 [3 3]]
b = [[ 2  4  6]
 [ 4  8 12]
 [ 6 12 18]]
c = [[ 2  4  6]
 [ 4  8 12]
 [ 6 12 18]]
d = [[1 4 9]
 [1 4 9]]
```

In [21]:
```
# Matrix inverse
B = np.array([[1.0, 2.0], [3.0, 4.0]])
np.linalg.inv(B)
```

Out[21]:
```
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

```
In [22]: # Eigenvectors and eigenvalues of a matrix
         np.linalg.eig(B)

Out[22]: (array([-0.37228132,  5.37228132]), array([[-0.82456484, -0.41597356],
                 [ 0.56576746, -0.90937671]]))
```

```
In [23]: # Trace of a matrix
         np.trace(B)

Out[23]: 5.0
```

```
In [24]: # Sove a linear system: Bx = y
         y = np.array([[5.], [7.]])
         np.linalg.solve(B, y)

Out[24]: array([[-3.],
                [ 4.]])
```

```
In [25]: # Identity matrix
         I = np.eye(2)
         I

Out[25]: array([[ 1.,  0.],
                [ 0.,  1.]])
```

```
In [26]: # Slicing arrays
         b[1:2,0:-1] # same as b[1,[0,1]] or b[1,0:2]

Out[26]: array([[4, 8]], dtype=int32)
```

## Map function

**Map** applies a function to all the items in an input_list. Here is the blueprint:

map(function_to_apply, list_of_inputs)

Most of the times we want to pass all the list elements to a function one-by-one and then collect the output. For instance:

In [27]:
```python
for element in l:
    print(math.sin(element))
```

```
0.8414709848078965
0.9092974268256817
0.1411200080598672
```

Map allows us to implement this in a much simpler and nicer way:

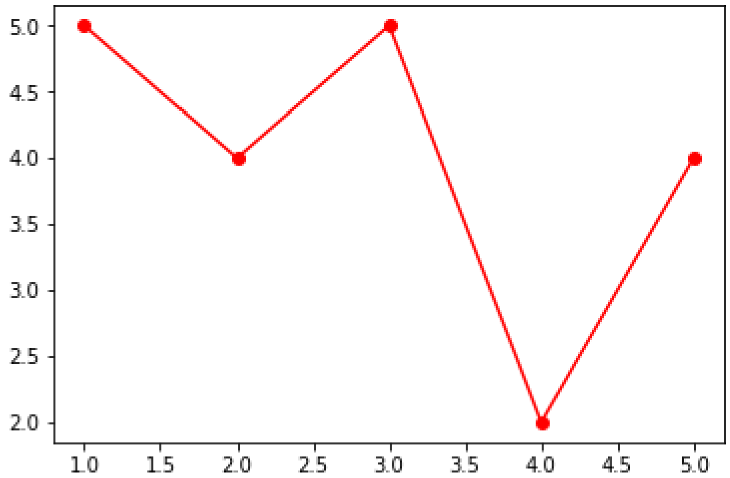
In [28]:
```python
# Componentwise operations
for s in map(math.sin, l):
    print(s)

# Or
print (np.sin(l))
```

```
0.8414709848078965
0.9092974268256817
0.1411200080598672
[ 0.84147098  0.90929743  0.14112001]
```

## Plot

In [29]:
```
plt.plot([1,2,3,4,5],[5,4,5,2,4], 'ro-');
plt.savefig('test.png')
# Change extension to .pdf to have it in that format
```




### A Little Bit More

- NumPy (https://docs.scipy.org/doc/numpy/user/quickstart.html): the fundamental package for scientific computing with Python.
- SciPy (https://docs.scipy.org/doc/scipy/reference/tutorial/index.html): a Python library used for scientific computing and technical computing.
- Pandas (http://pandas.pydata.org/pandas-docs/stable/tutorials.html): a Python library written for the Python programming language for data manipulation and analysis.