

# Binning Data with Python

October 22, 2015

## 1 Tutorial on binning, PDFs, CDFs, 1-CDFs and more

### 1.1 Introduction

In this course, we will need to plot various empirical probability distributions. As we deal with data, whose sparsity, and order of magnitudes may vary a lot, we have provided this tutorial to help you in producing appropriate visualizations of the data. But first, some mathematical background:

### 1.2 Discrete variables: Probability *mass* functions (PMF)

Let us assume we have some random variable  $V$  that can have only discrete values. Then the function describing the probabilities for different outcomes is the *probability mass function* (PMF)  $P(v)$ . If we assume, for simplicity, that our random variable  $V$  can only take integer values, the probabilities for obtaining different values of  $V$  can be written as:

- Probability of  $V$  being  $v$  ( $P(V = v)$ ) is simply written as  $P(v)$
- Probability of  $V$  being some value between  $x$  and  $y$  (with  $x$  and  $y$  included) equals

$$\sum_{v=x}^y P(v)$$

The probability mass function is also normalized to one:

$$\sum_{i=-\infty}^{\infty} P(i) = 1$$

### 1.3 Continuous variables: Probability *density* function (PDF)

The counterpart of a PMF for a *continuous* random variable  $v$  is its probability *density* function (PDF), denoted also typically by  $P(v)$ . For a probability density function  $P(v)$  it holds:

- Probability of observing a value between  $x$  and  $y$  ( $y > x$ ) equals

$$\int_x^y P(v)dv$$

- The distribution is normalized to one:

$$\int_{-\infty}^{\infty} P(v)dv = 1$$

### Example of a PDF denoted by $f(x)$ :

(Figure from: <http://physics.mercer.edu/hpage/CSP/pdf-cpf.gif>)

( $F(x)$  denotes the cumulative density function, more of that later)

Note that PDFs can have values greater than 1!

**From now on in this tutorial, we mostly assume that we work with continuous random variables**

In other words, we assume that the elements in our data are real numbers that arise from a distribution that is continuously distributed, and described by a probability density function (PDF). In practice, the same methods can (usually) be used also when we are dealing with discretely distributed data, such as node degrees in a network.

## 1.4 Computing and plotting empirical PDFs:

Let us start by presenting data that are either (i) “narrowly” distributed, or (ii) have a fat-tail using standard matplotlib plotting settings:

```
In [1]: import matplotlib.pyplot as plt
```

```
# this is only for this ipython notebook:
%matplotlib inline

import numpy as np

# "narrowly" distributed data, uniform distribution
rands = np.random.rand(10000)
# fat-tailed distribution
one_per_rands = 1./rands

fig = plt.figure(figsize=(12,6))
ax1 = fig.add_subplot(121)
ax2 = fig.add_subplot(122)

# basic histograms (see hist function below)
# work well with narrowly distributed data
#
# option: normed = True
#     divides the counts by the total number of observations and by
#     bin widths to compute the probability_density_function
#
#####
# NOTE (!) #
# With numpy.histogram, the option normed=True #
# does not always work properly! (use option density=True there!) #
# However, when using matplotlib (ax.hist) this bug is corrected! #
#####

pdf, bins, _ = ax1.hist(rands, normed=True, label="pdf, uniform")
ax1.legend()
print "If the histogram yields a probability distribution," + \
      "the following values should equal 1:"
print np.sum(pdf*(bins[1:]-bins[:-1]))

pdf, bins, _ = ax2.hist(one_per_rands, normed=True,
```

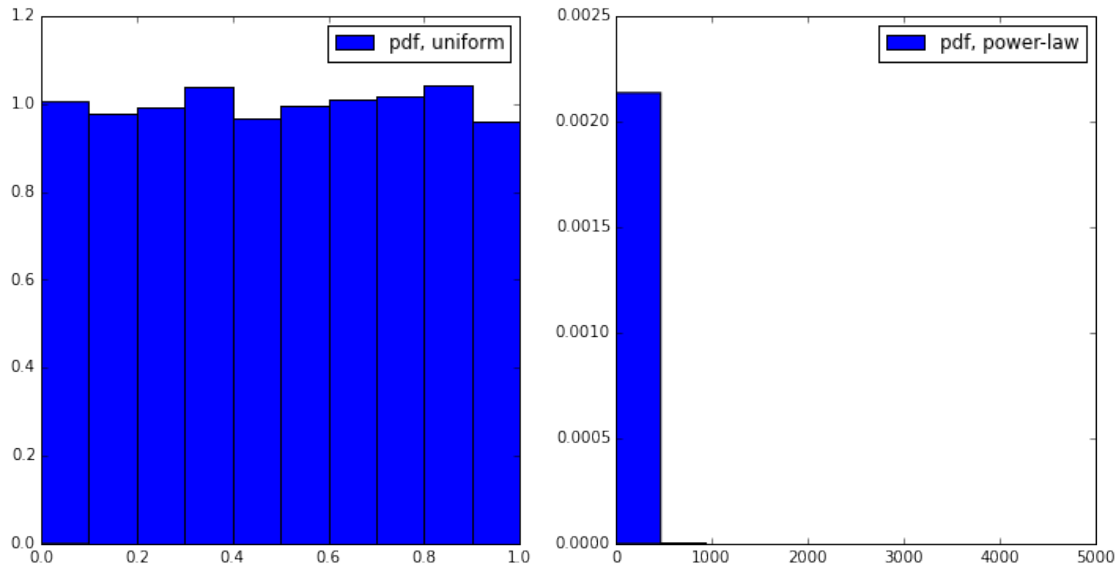
```

label='pdf, power-law')
ax2.legend()
print np.sum(pdf*(bins[1:]-bins[:-1])) # should equal 1"

print "Do they? Why should they?"

```

If the histogram yields a probability distribution, the following values should equal 1:  
 1.0  
 1.0  
 Do they? Why should they?



- Further tricks needed for seeing the shape of the (fat-tailed) power-law distribution!

## 1.5 What about increasing the number of bins?

```

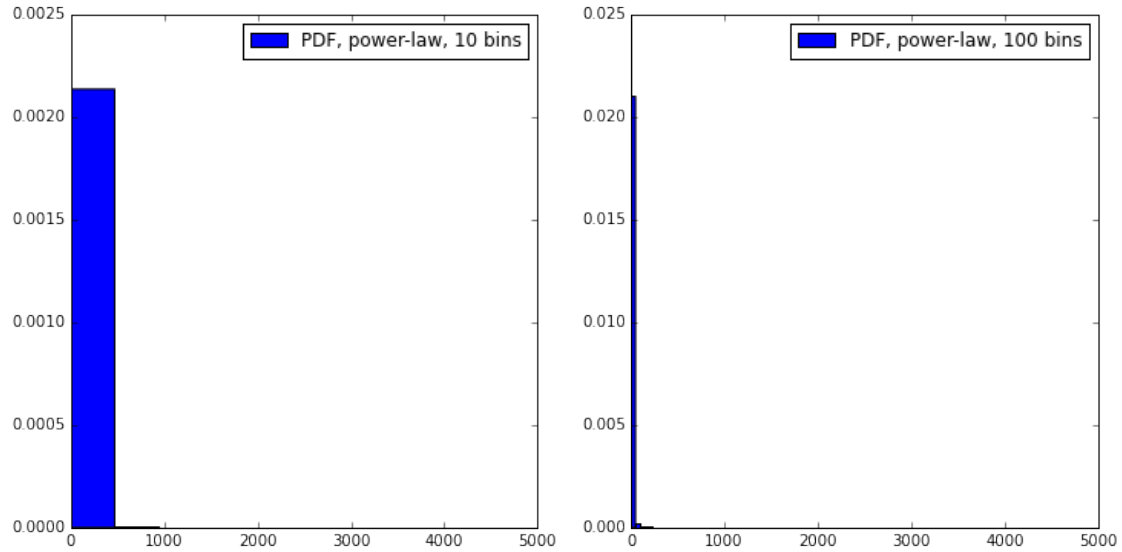
In [2]: fig = plt.figure(figsize=(12,6))
ax1 = fig.add_subplot(121)
ax2 = fig.add_subplot(122)

_ = ax1.hist(one_per_rands, 10, normed=True,
             label='PDF, power-law, 10 bins')
_ = ax2.hist(one_per_rands, 100, normed=True,
             label='PDF, power-law, 100 bins')
ax1.legend()
ax2.legend()

print "Increasing the number of bins does not really help out:"

```

Increasing the number of bins does not really help out:



## 1.6 Solution: logarithmic bins:

Use logarithmic bins instead of linear!

```
In [3]: fig = plt.figure(figsize=(12,6))
ax1 = fig.add_subplot(121)
ax2 = fig.add_subplot(122)

_ = ax1.hist(one_per_rands, bins=100, normed=True)
ax1.set_title('PDF, lin-lin, power-law, 100 bins')
ax1.set_xlabel('x')
ax1.set_ylabel('PDF')

# creating logarithmically spaced bins

print "Min and max:"
print np.min(one_per_rands), np.max(one_per_rands)

# create log bins: (by specifying the multiplier)
bins = [np.min(one_per_rands)]
cur_value = bins[0]
multiplier = 2.5
while cur_value < np.max(one_per_rands):
    cur_value = cur_value * multiplier
    bins.append(cur_value)
bins = np.array(bins)
# an alterante way, if one wants to just specify the
# number of bins to be used:
# bins = np.logspace(np.log10(np.min(one_per_rands)),
#                    np.log10(np.max(one_per_rands)),
#                    num=10)
```

```

print "The created bins:"
print bins

bin_widths = bins[1:]-bins[0:-1]
print "Bin widths are increasing:"
print bin_widths
print "The bin width is multiplied always" + \
      "multiplied by a constant factor:"
print "bin_width[1]/bin_width[0]=", bin_widths[1]/bin_widths[0]
print "bin_width[2]/bin_width[1]=", bin_widths[2]/bin_widths[1]

_ = ax2.hist(one_per_rands, bins=bins, normed=True)
ax2.set_title('PDF, log-log, power-law, ' + str(len(bins)) + " bins")
ax2.set_xscale('log')
ax2.set_yscale('log')
ax2.set_xlabel('x')
ax2.set_ylabel('PDF')

```

Min and max:

1.00010308326 4674.56479613

The created bins:

```

[ 1.00010308e+00  2.50025771e+00  6.25064427e+00  1.56266107e+01
 3.90665267e+01  9.76663167e+01  2.44165792e+02  6.10414480e+02
 1.52603620e+03  3.81509050e+03  9.53772624e+03]

```

Bin widths are increasing:

```

[ 1.50015462e+00  3.75038656e+00  9.37596641e+00  2.34399160e+01
 5.85997900e+01  1.46499475e+02  3.66248688e+02  9.15621719e+02
 2.28905430e+03  5.72263575e+03]

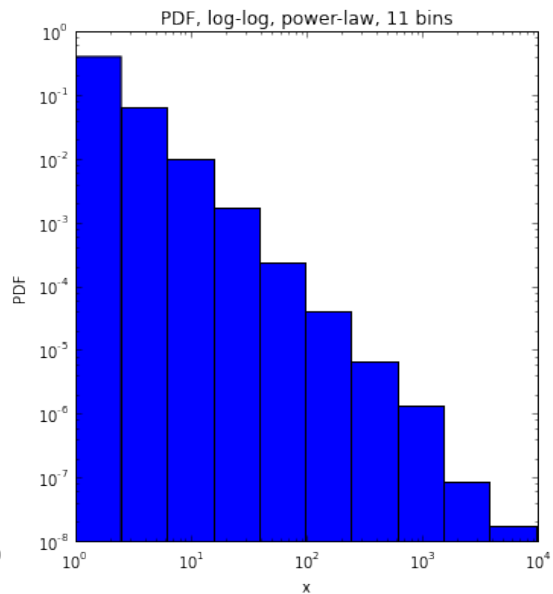
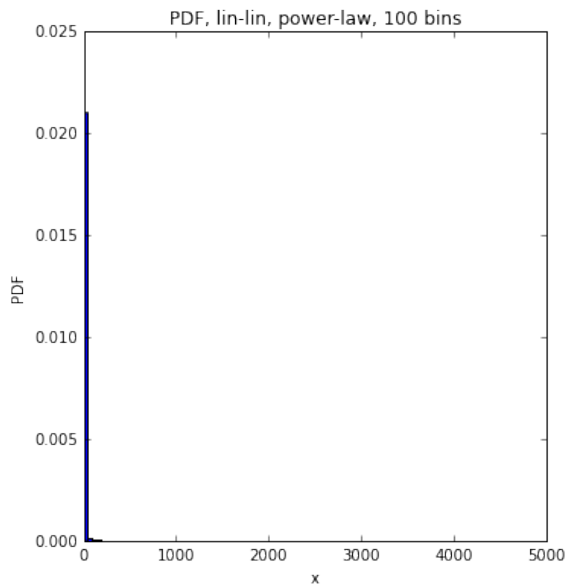
```

The bin width is multiplied alwaysmultiplied by a constant factor:

bin\_width[1]/bin\_width[0]= 2.5

bin\_width[2]/bin\_width[1]= 2.5

Out[3]: <matplotlib.text.Text at 0x7f9af78bbc10>



Now we can see something in the tail as well! So, how does the PDF of  $1/x$ , with  $x \text{ Uniform}(0,1)$  look like in the log-log scale? (Can you show this using pen and paper?)

For interpreting loglog pdf plots, one needs to know how different ‘basic’ distributions look like on log-log scale...

## 1.7 Let’s see how different distributions look like with different x- and y-scales:

```
In [4]: # import from scipy some probability distributions to use:
        from scipy.stats import lognorm, expon

        # Trying out different broad distributions with
        # linear and logarithmic PDFs:

        n_points = 10000

        # power law:
        # slope = -2! (why?)
        one_over_rands = 1/np.random.rand(n_points)
        # http://en.wikipedia.org/wiki/Power_law

        # exponential distribution
        exs = expon.rvs(size=n_points)
        # http://en.wikipedia.org/wiki/Exponential_distribution

        # lognormal (looks like a normal distribution in a log-log scale!)
        lognorms = lognorm.rvs(1.0, size=n_points)
        # http://en.wikipedia.org/wiki/Log-normal_distribution

        fig = plt.figure(figsize=(15,15))
        n_bins = 25

        dist_data = [one_over_rands, exs, lognorms]
        dist_names = ["power law", "exponential", "lognormal"]
        for i, (rands, name) in enumerate(zip(dist_data, dist_names)):
            # linear-linear scale
            ax = fig.add_subplot(4, 3, i+1)
            ax.hist(rands, n_bins, normed=True)
            ax.text(0.5,0.9, "PDF, lin-lin: " + name,
                   transform=ax.transAxes)

            # log-log scale
            ax = fig.add_subplot(4, 3, i+4)
            bins = np.logspace(np.log10(np.min(rands)),
                              np.log10(np.max(rands)),
                              num=n_bins)
            ax.hist(rands, normed=True, bins=bins)
            ax.set_xscale('log')
            ax.set_yscale('log')
            ax.text(0.5,0.9, "PDF, log-log: " + name,
                   transform=ax.transAxes)

            # lin-log
            ax = fig.add_subplot(4, 3, i+7)
            ax.hist(rands, normed=True, bins=n_bins)
```

```

ax.text(0.5,0.9, "PDF, lin-log: " + name,
        transform=ax.transAxes)
ax.set_yscale('log')
# log-lin
ax = fig.add_subplot(4, 3, i+10)
bins = np.logspace(np.log10(np.min(rands)),
                  np.log10(np.max(rands)),
                  num=n_bins)
ax.hist(rands, normed=True, bins=bins)
ax.text(0.5,0.9, "PDF, log-lin: " + name,
        transform=ax.transAxes)
ax.set_xscale('log')

for ax in fig.axes:
    ax.set_xlabel('x')
    ax.set_ylabel('PDF(x)')

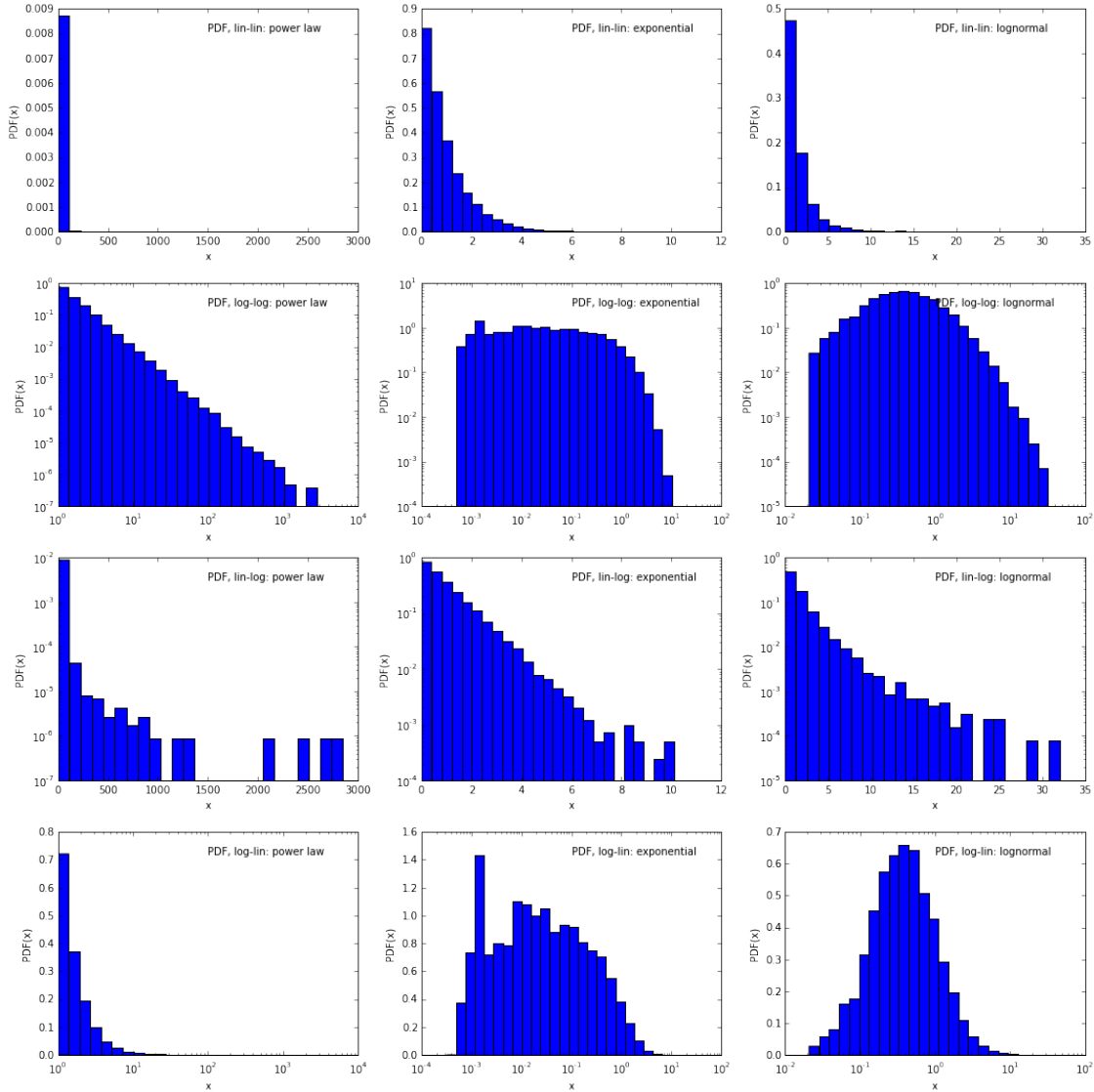
plt.tight_layout()

print "Distributions can look a lot different depending on the binning and axes scales!\n"
print "Note that PDFs can have a value over 1!"
print "(it is the bin_width*pdf which counts for the normalization)"

```

Distributions can look a lot different depending on the binning and axes scales!

Note that PDFs can have a value over 1!  
(it is the bin\_width\*pdf which counts for the normalization)



## 1.8 Summary so far:

- Choose axes scales and bins according to the data! (no all-round solution exists)
- Note: choosing appropriate bins can be difficult (especially with real, sparse data)!
- One way for getting around binning is to plot the cumulative density functions!

## 1.9 Don't want to bin? Use CDF(x) and 1-CDF(x)!

Representing PDFs using binning strategies can be difficult due to limited samples of data etc.. To neglect the need for binning, one can use the cumulative distribution function (CDF) and the complementary cumulative distribution function (1-CDF).



## 1.10 Cumulative density function (CDF(x)):

- The **standard** mathematical definitions:
- Discrete variable:  $CDF(x) = \sum_{i \leq x} P(x)$
- Continuous variable:  $CDF(x) = \int_{-\infty}^x P(x') dx$

## 1.11 Complementary cumulative density function (cCDF(x), 1-CDF(x))

- In this course, the **complementary cumulative distributions** are often more practical.
- 1-CDF is especially useful when dealing with fat-tailed distribution as they enable one to zoom in to the *tail* of the distribution.
- *In this course* we will use the following definitions for “1-CDF(x)”:
- Discrete variable:  $1 - CDF(x) = \sum_{i \geq x} P(x)$
- Continuous variable:  $1 - CDF(x) = \int_x^{\infty} P(x') dx$
- **Note:**

The standard definition of the discrete “1-CDF(x)” does not take into account the values that equal  $x$ . Thus our above “1-CDF(x)” + “CDF(x)” does not exactly equal one. The reason why we take up our nonstandard 1-CDF(x) definition is that it enables us to work more practically with real data. Especially, were we to plot the 1-CDF(x) on a logarithmic y-axis, the largest observed data point that we have would not be visible with the standard 1-CDF(x) definition. (Why?)

- So given some data vector  $d$  data, we present the empirical 1-CDFs as:

$$1 - CDF(x) = \frac{\text{number of elements in } d \text{ that are } \geq x}{\text{number of elements in } d}$$

In [5]: `def plot_ccdf(data, ax):`

```
    """
    Plot the complementary cumulative distribution function
    (1-CDF(x)) based on the data on the axes object.

    Note that this way of computing and plotting the ccdf is not
    the best approach for a discrete variable, where many
    observations can have exactly same value!
    """
    # Note that, here we use the convention for presenting an
    # empirical 1-CDF (ccdf) as discussed
    # a quick way of computing a ccdf (valid for continuous data):
    sorted_vals = np.sort(np.unique(data))
    ccdf = np.zeros(len(sorted_vals))
    n = float(len(data))
    for i, val in enumerate(sorted_vals):
        ccdf[i] = np.sum(data >= val)/n
    ax.plot(sorted_vals, ccdf, "-")
    # faster (approximative) way:
    # sorted_vals = np.sort(data)
    # ccdf = np.linspace(1, 1./len(data), len(data))
    # ax.plot(sorted_vals, ccdf)
```

```

def plot_cdf(data, ax):
    """
    Plot CDF(x) on the axes object

    Note that this way of computing and plotting the CDF is not
    the best approach for a discrete variable, where many
    observations can have exactly same value!
    """

    # Note that, here we use the convention for presenting an
    # empirical 1-CDF (ccdf) as discussed
    # a quick way of computing a cdf (valid for continuous data):
    sorted_vals = np.sort(np.unique(data))
    cdf = np.zeros(len(sorted_vals))
    n = float(len(data))
    for i, val in enumerate(sorted_vals):
        cdf[i] = np.sum(data <= val)/n
    ax.plot(sorted_vals, cdf, "-")

    # faster (approximative) way:
    # sorted_vals = np.sort(data)
    # now probs run from "0 to 1"
    # probs = np.linspace(1./len(data),1, len(data))
    # ax.plot(sorted_vals, probs, "-")

fig = plt.figure(figsize=(15,15))
fig.suptitle('Different broad distribution CDFs in' + \
            'lin-lin, log-log, and lin-log axes')
# loop over different empirical data distributions
# enumerate, enumerates the list elements (gives out i in addition to the data)
# zip([1,2],[a,b]) returns [[1,"a"], [2,"b"]]

dist_data = [one_over_rands, exps, lognorms]
dist_names = ["power law", "exponential", "lognormal"]
for i, (rands, name) in enumerate(zip(dist_data, dist_names)):
    # linear-linear scale
    ax = fig.add_subplot(4,3,i+1)
    plot_cdf(rands, ax)
    ax.grid()
    ax.text(0.6,0.9, "lin-lin, CDF: " + name,
            transform=ax.transAxes)

    # log-log scale
    ax = fig.add_subplot(4,3,i+4)
    plot_cdf(rands, ax)
    ax.set_xscale('log')
    ax.set_yscale('log')
    ax.grid()
    ax.text(0.6,0.9, "log-log, CDF: " + name,
            transform=ax.transAxes)

    # lin-lin 1-CDF
    ax = fig.add_subplot(4,3,i+7)
    plot_ccdf(rands, ax)

```

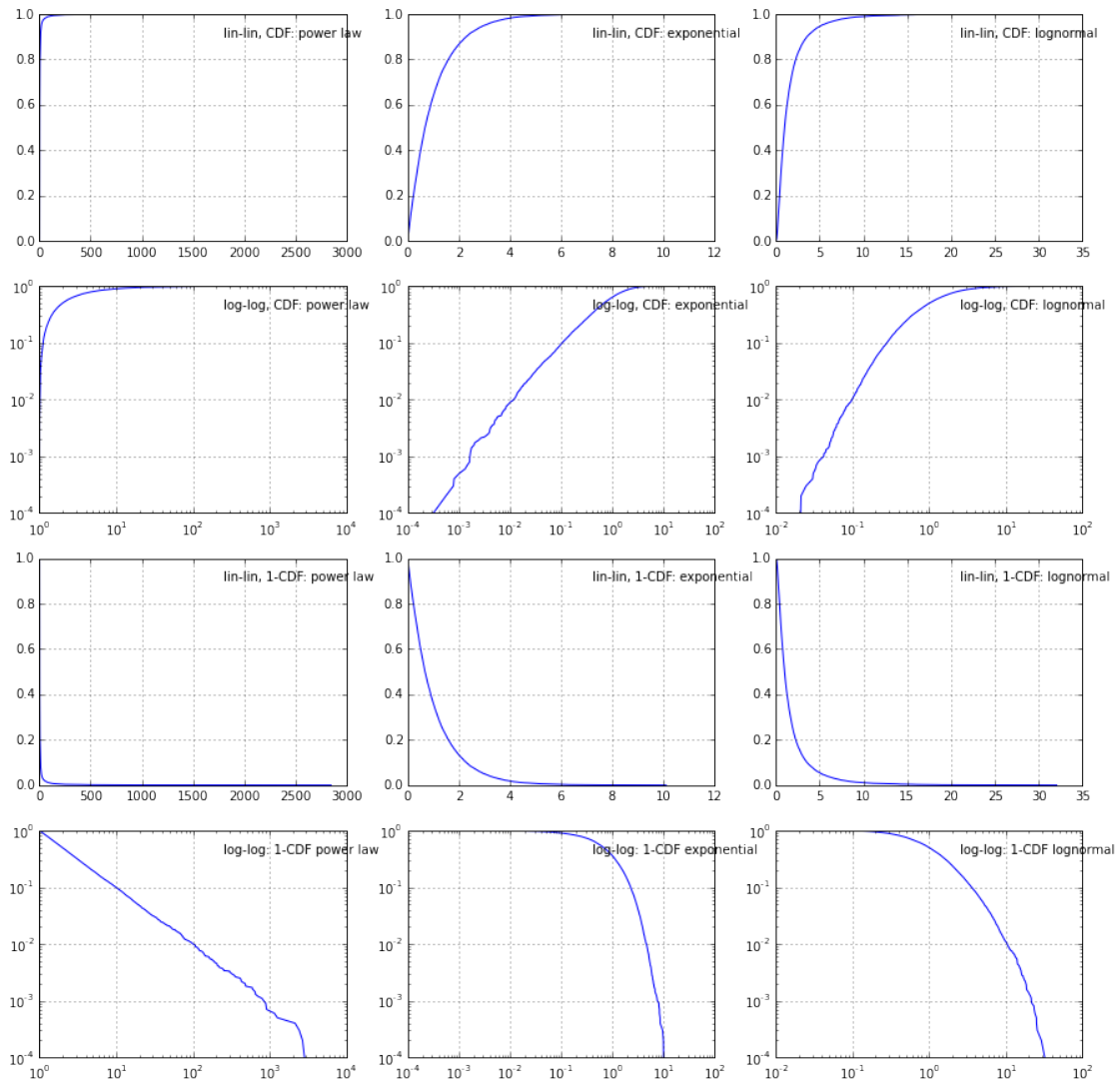
```

ax.text(0.6,0.9, "lin-lin, 1-CDF: " + name,
        transform=ax.transAxes)
ax.grid()

# log-log 1-CDF
ax = fig.add_subplot(4,3,i+10)
plot_ccdf(rands, ax)
ax.text(0.6,0.9, "log-log: 1-CDF " + name,
        transform=ax.transAxes)
ax.set_yscale('log')
ax.set_xscale('log')
ax.grid()

```

Different broad distribution CDFs in lin-lin, log-log, and lin-log axes



## 1.12 Notes

- **\*\* Only with 1-CDF one can zoom in to the tail \*\***
- **\*\* Sometimes lin-scale good, sometimes log scale \*\***
- **\*\* Note that for power laws, PDF, and 1-CDF are both straight lines in log-log coordinates \*\***

## 1.13 Plotting 2D-distributions

What is the dependence between values of vectors X and Y?

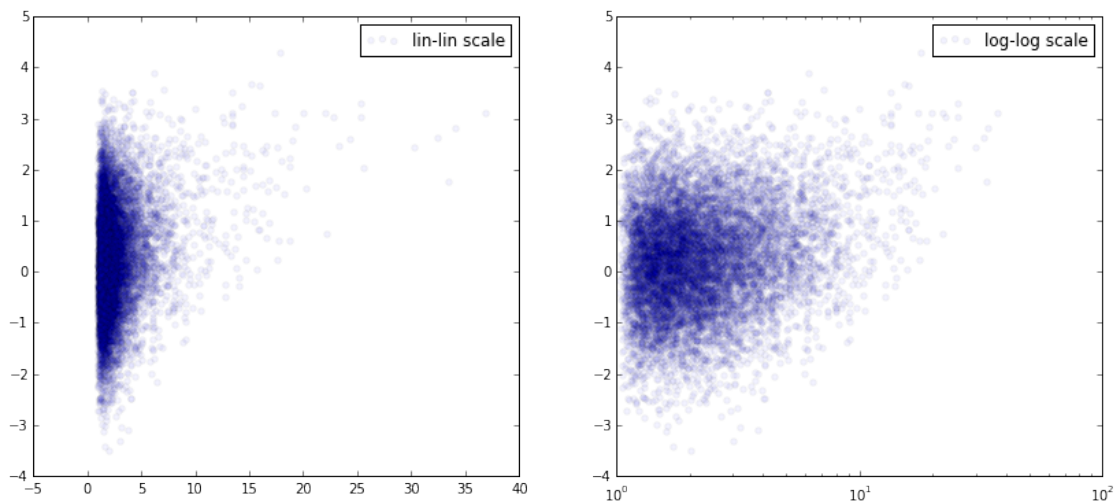
First approach: simple scatter plot :

```
In [6]: fig = plt.figure(figsize=(14,6))
```

```
n_points = 10000
x = np.exp(-np.random.randn(n_points)) + 1
# insert slight linear dependence!
y = x*0.1 + np.random.randn(n_points)

ax1 = fig.add_subplot(121)
# alpha controls transparency
ax1.scatter(x,y, alpha=0.05, label="lin-lin scale")
ax1.legend()

ax2 = fig.add_subplot(122)
# alpha controls transparency
ax2.scatter(x,y, alpha=0.05, label="log-log scale")
ax2.legend()
ax2.set_xscale('log')
```



## 1.14 Can we spot the slight trend?

Compute bin-averages! (with `binned_statistic`):

```

In [7]: from scipy.stats import binned_statistic

# Note that the
# binned_statistic function above may not be available at Aalto computers
# due to an outdated scipy version.
# To use binned_statistic (or binned_statistic_2d) functions
# copy the file bs.py to your coding directory from the course web-page
# and take it into use use it similarly with:
#
# from bs import binned_statistic

fig = plt.figure(figsize=(14,6))
# linear bins for the lin-scale:

n_bins = 20
bin_centers, _, _ = binned_statistic(x, x,
                                     statistic='mean',
                                     bins=n_bins)
bin_averages, _, _ = binned_statistic(x, y,
                                     statistic='mean',
                                     bins=n_bins)
bin_stdevs, _, _ = binned_statistic(x, y,
                                     statistic='std',
                                     bins=n_bins)

ax1 = fig.add_subplot(121)

# Note: alpha controls the level of transparency
ax1.scatter(x,y, alpha=0.05, label="lin-lin")
ax1.errorbar(bin_centers, bin_averages, bin_stdevs,
            fmt="o-", color="r", label="avg with stdev")
ax1.legend(loc='best')
ax1.set_xlabel("x")
ax1.set_ylabel("y")
print "Note the missing points with larger values of x!"

# generate logarithmic x-bins
log_bins_for_x = np.logspace(np.log10(np.min(x)),
                             np.log10(np.max(x)),
                             num=n_bins)

# get bin centers and averages:
bin_centers, _, _ = binned_statistic(x, x,
                                     statistic='mean',
                                     bins=log_bins_for_x)

# Note: instead of just taking the center of each bin,
# it can be sometimes very important to set the x-value of a bin
# to the actual mean of the x-values in that bin!
# (This is the case e.g. in the exercise where we implement
# the Barabasi-Albert network!)

bin_averages, _, _ = binned_statistic(x, y,
                                     statistic='mean',
                                     bins=log_bins_for_x)

```

```

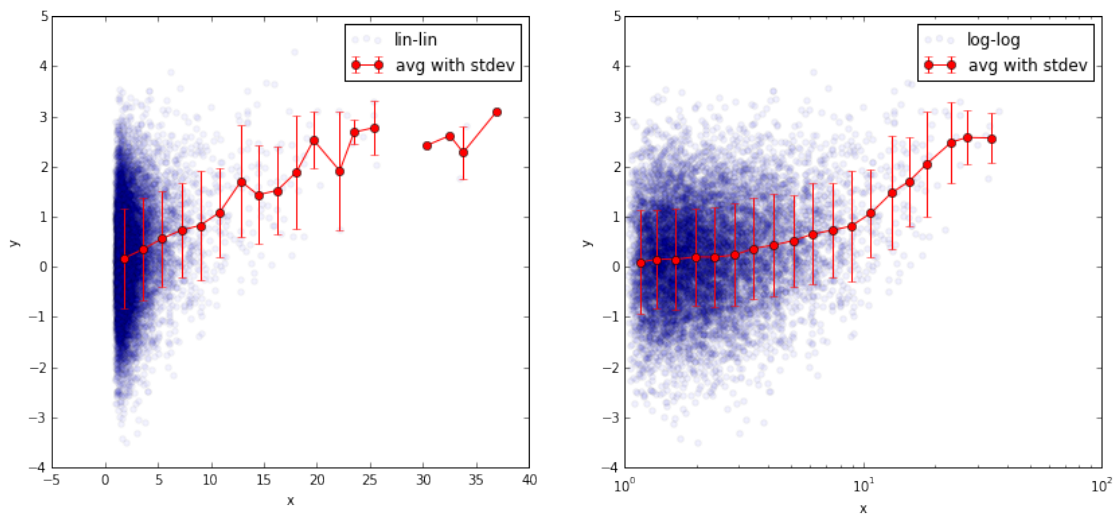
bin_stdevs, _, _ = binned_statistic(x, y,
                                   statistic='std',
                                   bins=log_bins_for_x)

ax2 = fig.add_subplot(122)
ax2.scatter(x,y, alpha=0.06, label="log-log")
ax2.errorbar(bin_centers, bin_averages, bin_stdevs,
             fmt="o-", color="r", label="avg with stdev")

ax2.legend(loc='best')
ax2.set_xlabel("x")
ax2.set_ylabel("y")
ax2.set_xscale('log')

```

Note the missing points with larger values of x!



More sophisticated approaches for presenting 2D-distributions include e.g. heatmaps, which can be produced using using `binned_statistic2d` and `pcolor`. More of those later in the course!

### 1.15 Summary:

- Know your data.
- Use proper axes that fit the purpose!
- With log-log axes it is possible to ‘zoom’ into the tail.
- Binning can be tricky
  - PDFs especially
  - 1-CDFs helpful in investigating the tail + no need for binning