Bradley Voytek, Ph.D.

UC San Diego

Department of Cognitive Science
Halıcıoğlu Data Science Institute
Neurosciences Graduate Program

bvoytek@ucsd.edu
@bradleyvoytek

Quick student survey

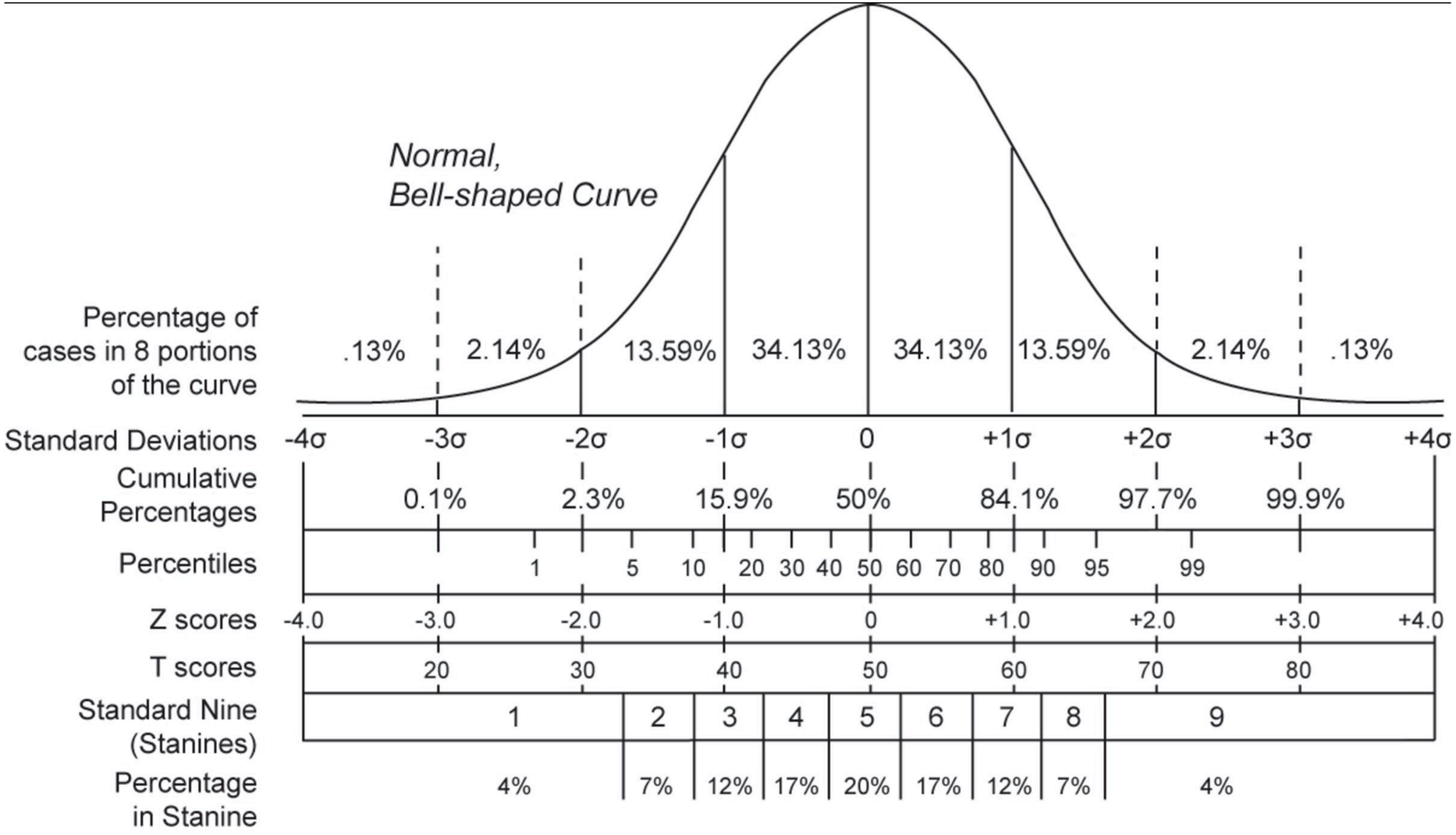
- http://bit.ly/108f19q

COGS 108
Data Science in Practice

Distributions

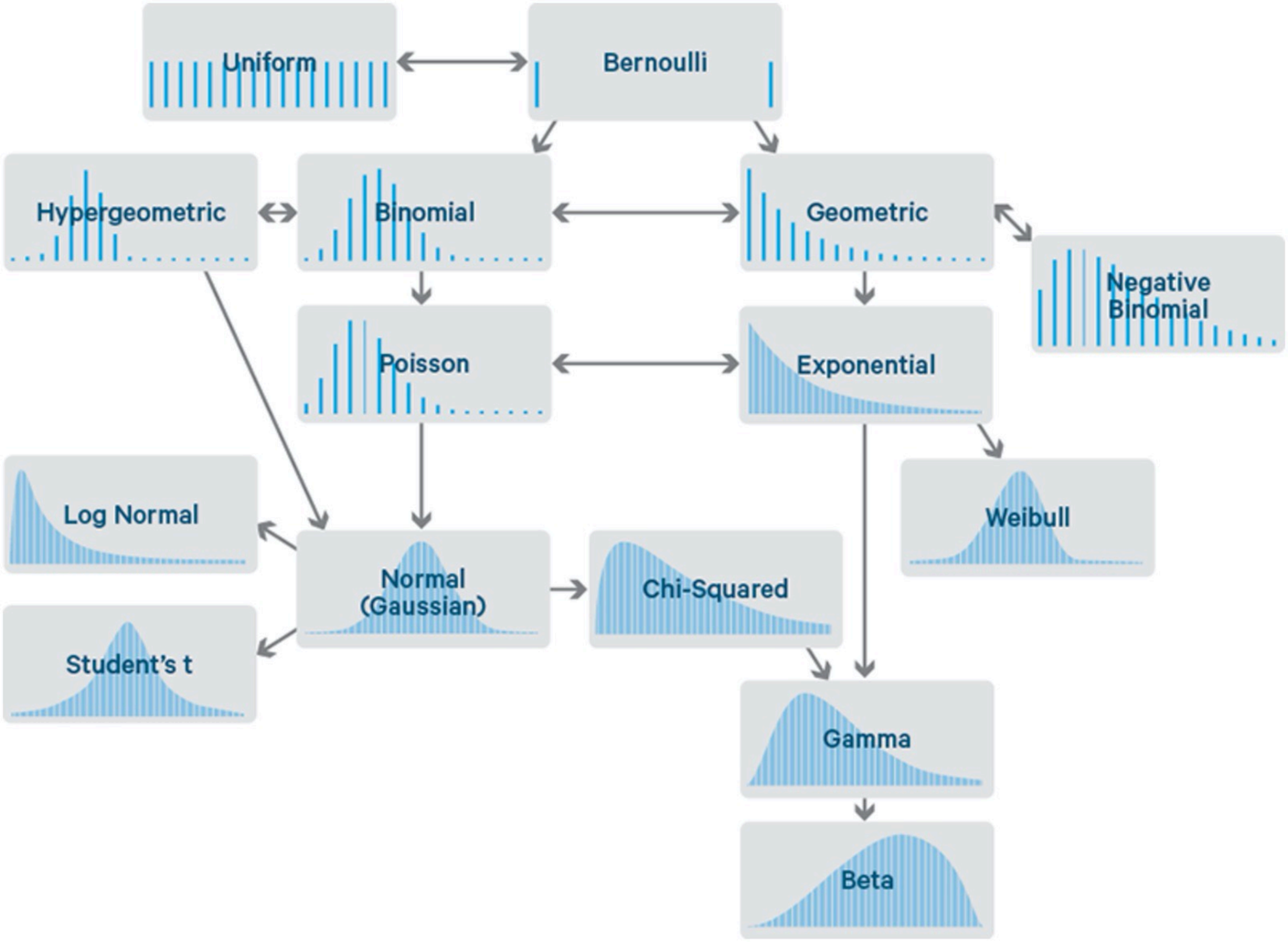
UC San Diego

Distributions



UC San Diego

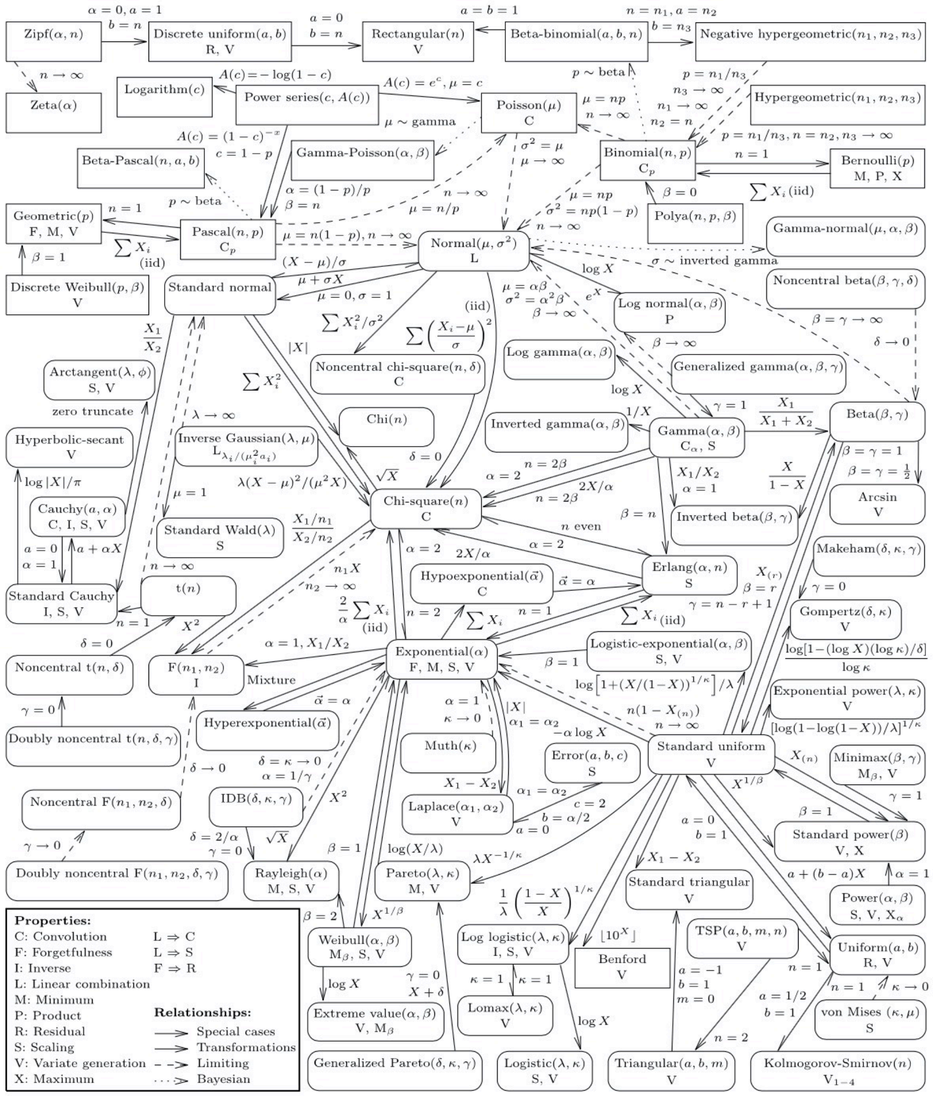
Distributions



Source: *Cloudera blog*: http://blog.cloudera.com/blog/2015/12/common-probability-distributions-the-data-scientists-crib-sheet/

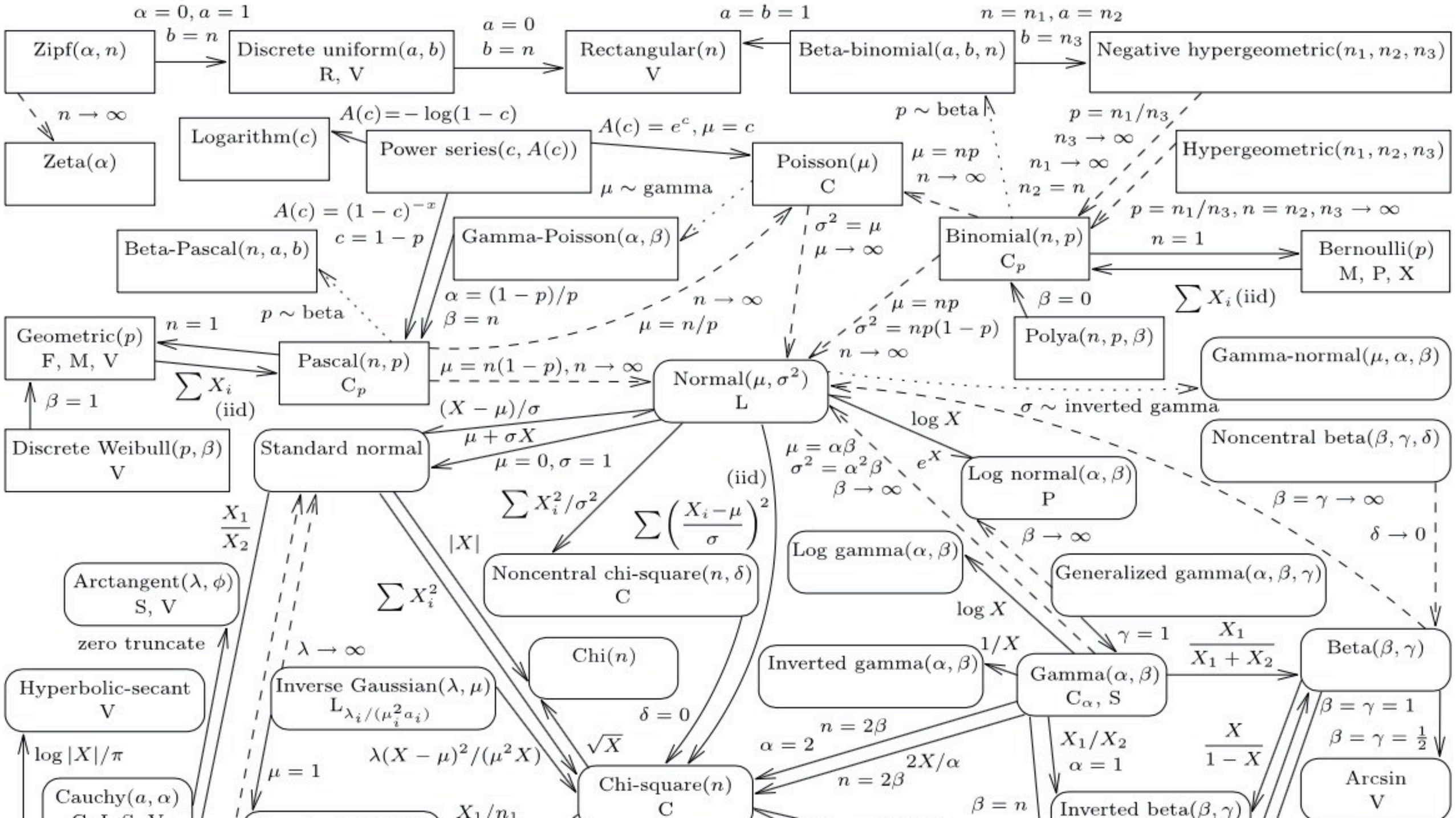
UC San Diego

Distributions

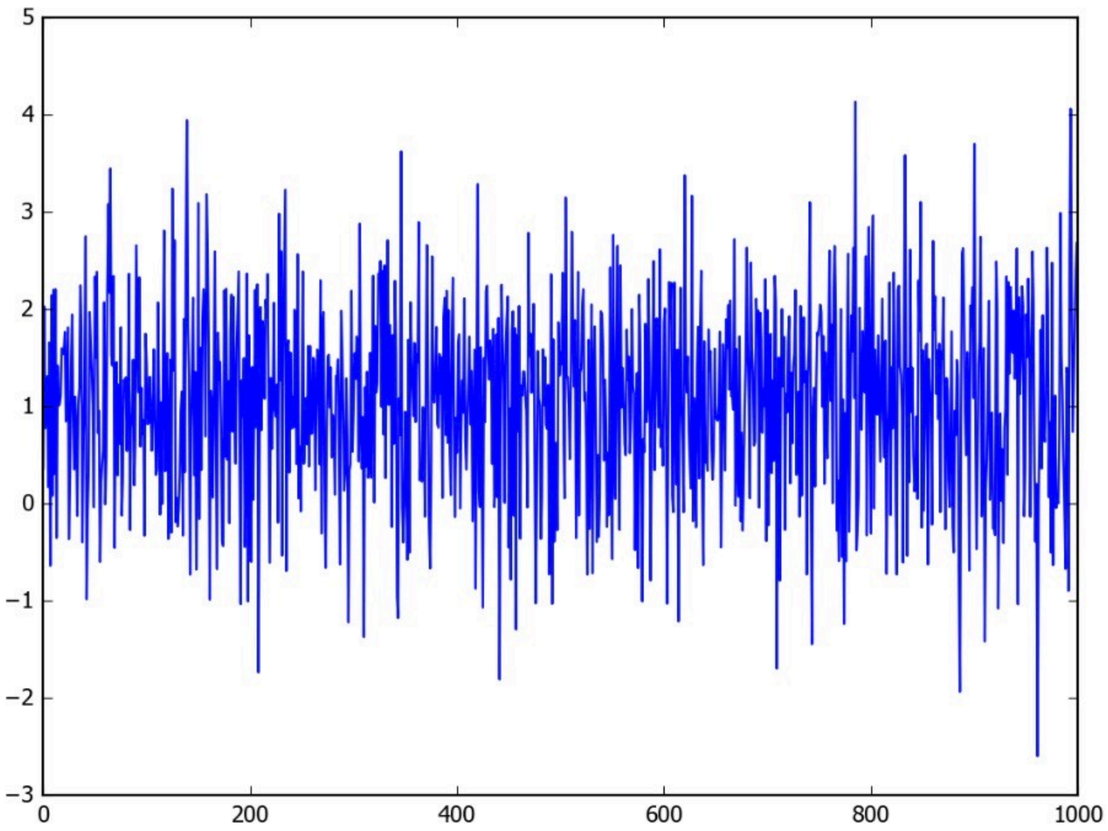


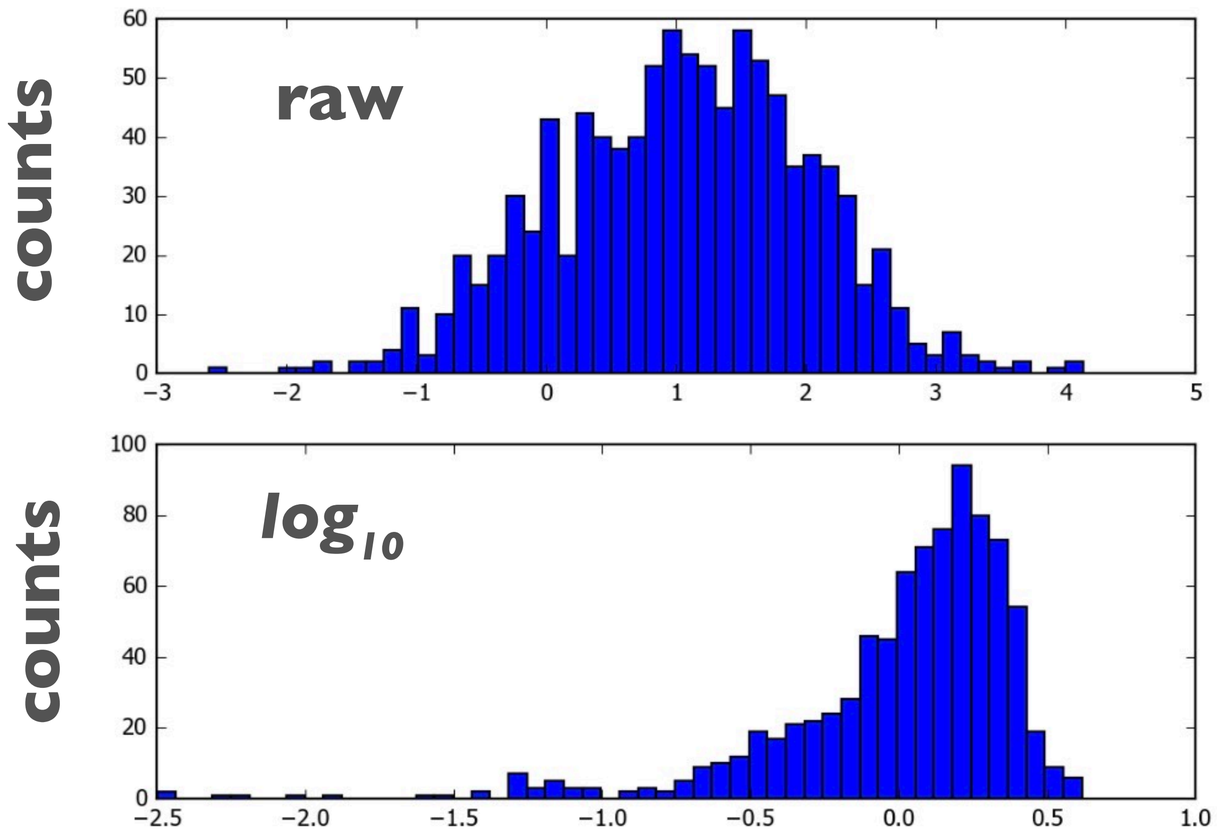
Source: Leemis & McQueston, *American Statistician* 2008

UC San Diego

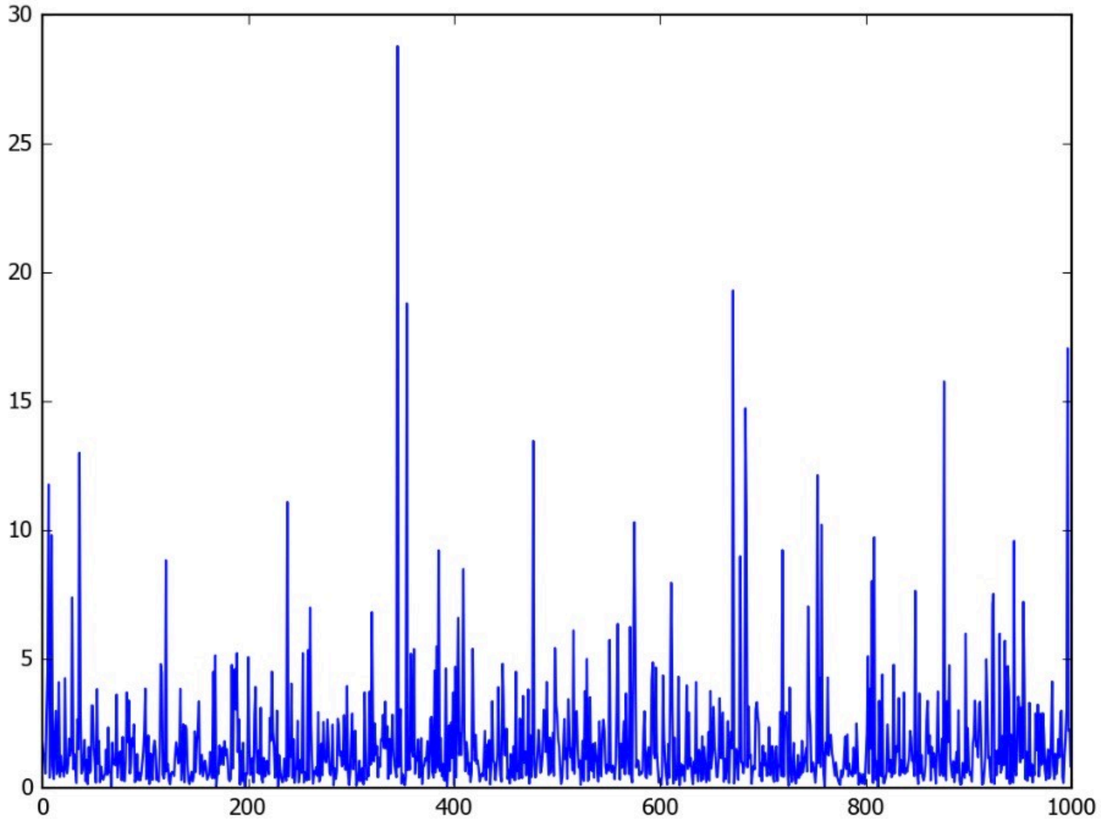

Zipf(α, n)
$\alpha = 0, a = 1$
$b = n$
Discrete uniform(a, b)
R, V
$a = 0$
$b = n$
Rectangular(n)
V
$a = b = 1$
Beta-binomial(a, b, n)
$n = n_1, a = n_2$
$b = n_3$
Negative hypergeometric(n_1, n_2, n_3)
$n \to \infty$
Zeta(α)
Logarithm(c)
$A(c) = -\log(1-c)$
Power series($c, A(c)$)
$A(c) = e^c, \mu = c$
Poisson(μ)
C
$p \sim$ beta
$p = n_1/n_3$
$n_3 \to \infty$
$n_1 \to \infty$
$n_2 = n$
Hypergeometric(n_1, n_2, n_3)
$\mu = np$
$n \to \infty$
$\mu \sim$ gamma
$A(c) = (1-c)^{-x}$
$c = 1 - p$
Beta-Pascal(n, a, b)
Gamma-Poisson(α, β)
$\sigma^2 = \mu$
$\mu \to \infty$
Binomial(n, p)
C_p
$p = n_1/n_3, n = n_2, n_3 \to \infty$
$n = 1$
Bernoulli(p)
M, P, X
$\alpha = (1-p)/p$
$\beta = n$
$n \to \infty$
$\mu = n/p$
$\mu = np$
$\sigma^2 = np(1-p)$
$n \to \infty$
$\beta = 0$
$\sum X_i$ (iid)
Polya(n, p, β)
Geometric(p)
F, M, V
$n = 1$
$p \sim$ beta
Pascal(n, p)
C_p
$\sum X_i$ (iid)
$\mu = n(1-p), n \to \infty$
Normal(μ, σ^2)
L
Gamma-normal(μ, α, β)
$\beta = 1$
$\sigma \sim$ inverted gamma
Discrete Weibull(p, β)
V
Standard normal
$(X - \mu)/\sigma$
$\mu + \sigma X$
$\mu = 0, \sigma = 1$
$\log X$
$\mu = \alpha\beta$
$\sigma^2 = \alpha^2\beta$
$\beta \to \infty$
e^X
Log normal(α, β)
P
Noncentral beta(β, γ, δ)
$\beta = \gamma \to \infty$
$\delta \to 0$
$\frac{X_1}{X_2}$
(iid)
$\sum X_i^2/\sigma^2$
$\sum \left(\frac{X_i - \mu}{\sigma}\right)^2$
$|X|$
$\sum X_i^2$
Noncentral chi-square(n, δ)
C
Log gamma(α, β)
$\beta \to \infty$
Generalized gamma(α, β, γ)
$\log X$
Arctangent(λ, ϕ)
S, V
zero truncate
$\lambda \to \infty$
Chi(n)
$1/X$
Inverted gamma(α, β)
Gamma(α, β)
C_α, S
$\gamma = 1$
$\frac{X_1}{X_1 + X_2}$
Beta(β, γ)
Hyperbolic-secant
V
Inverse Gaussian(λ, μ)
$L_{\lambda_i/(\mu_i^2 a_i)}$
$\delta = 0$
$\sqrt{X}$
$n = 2\beta$
$\alpha = 2$
$2X/\alpha$
$n = 2\beta$
X_1/X_2
$\alpha = 1$
$\frac{X}{1-X}$
$\beta = \gamma = 1$
$\beta = \gamma = \frac{1}{2}$
$\log|X|/\pi$
$\lambda(X-\mu)^2/(\mu^2 X)$
$\mu = 1$
Chi-square(n)
C
$\beta = n$
Inverted beta(β, γ)
Arcsin
V
Cauchy(a, α)
X_1/n_1

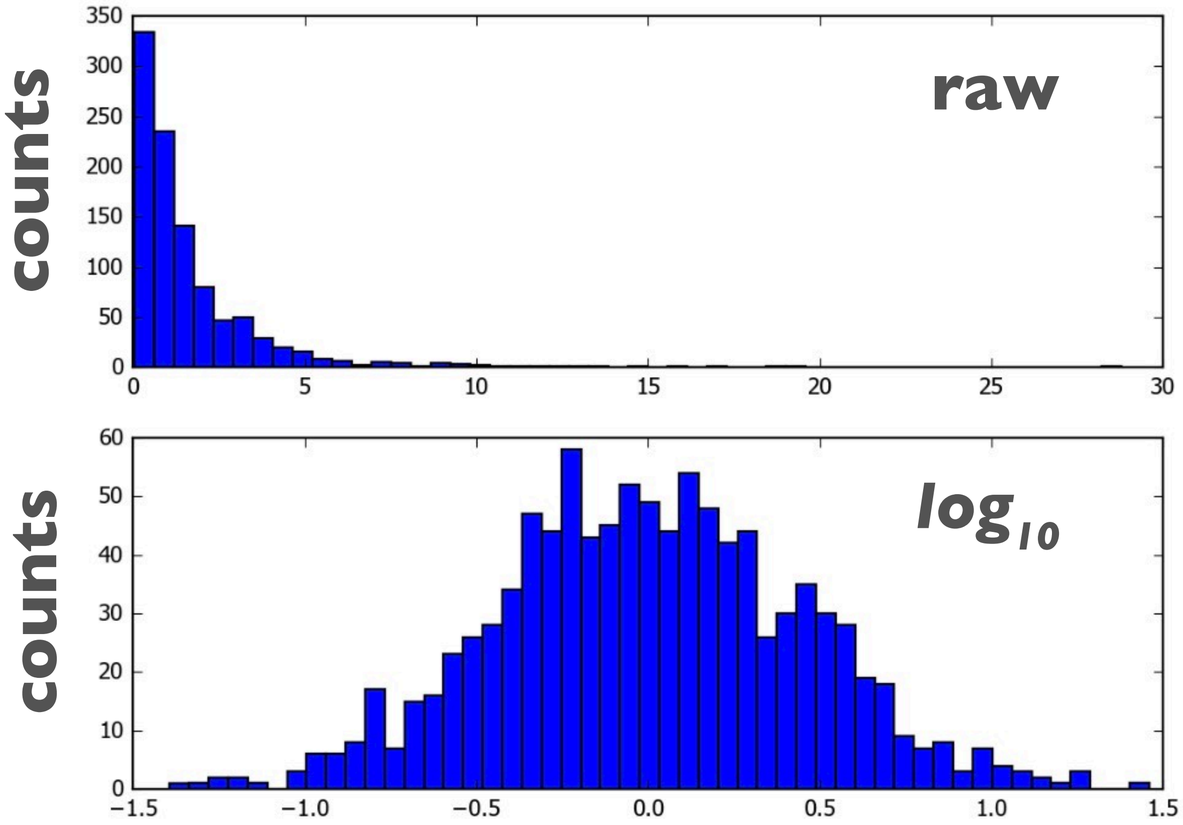
Normal


5
4
3
2
1
0
−1
−2
−3
0
200
400
600
800
1000

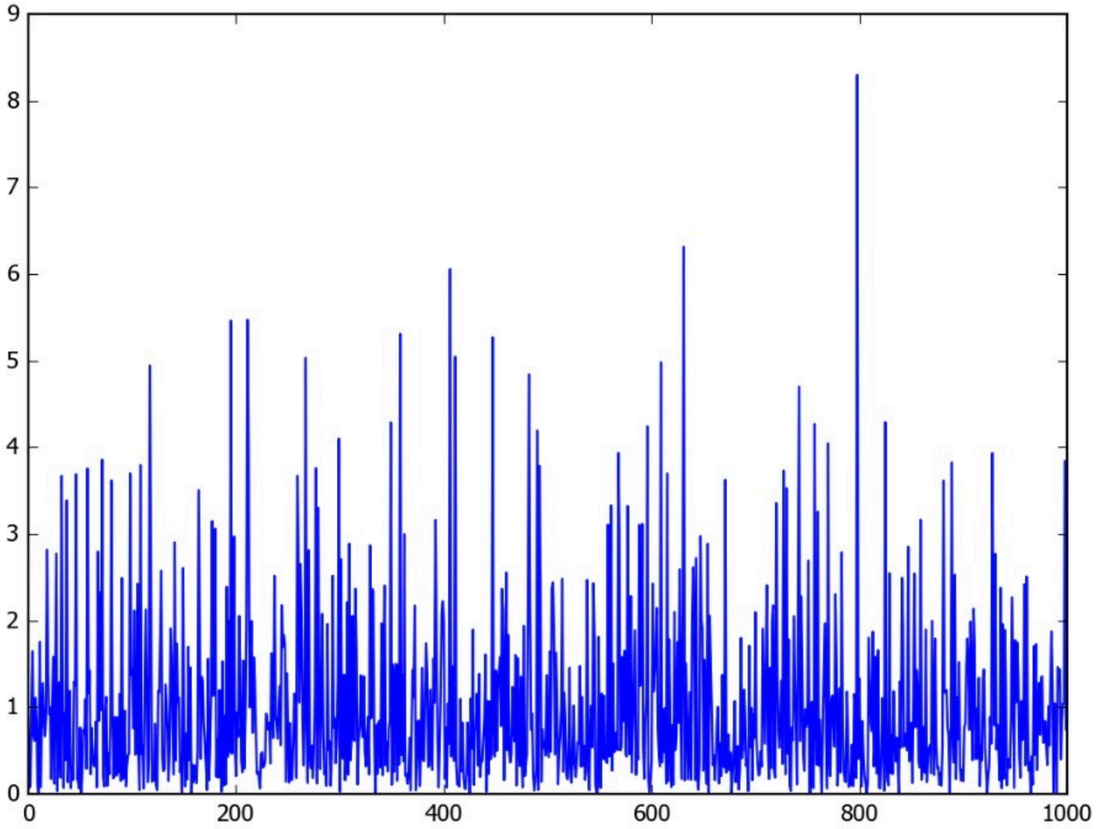

counts
raw
60
50
40
30
20
10
0
−3
−2
−1
0
1
2
3
4
5
counts
log10
100
80
60
40
20
0
−2.5
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0

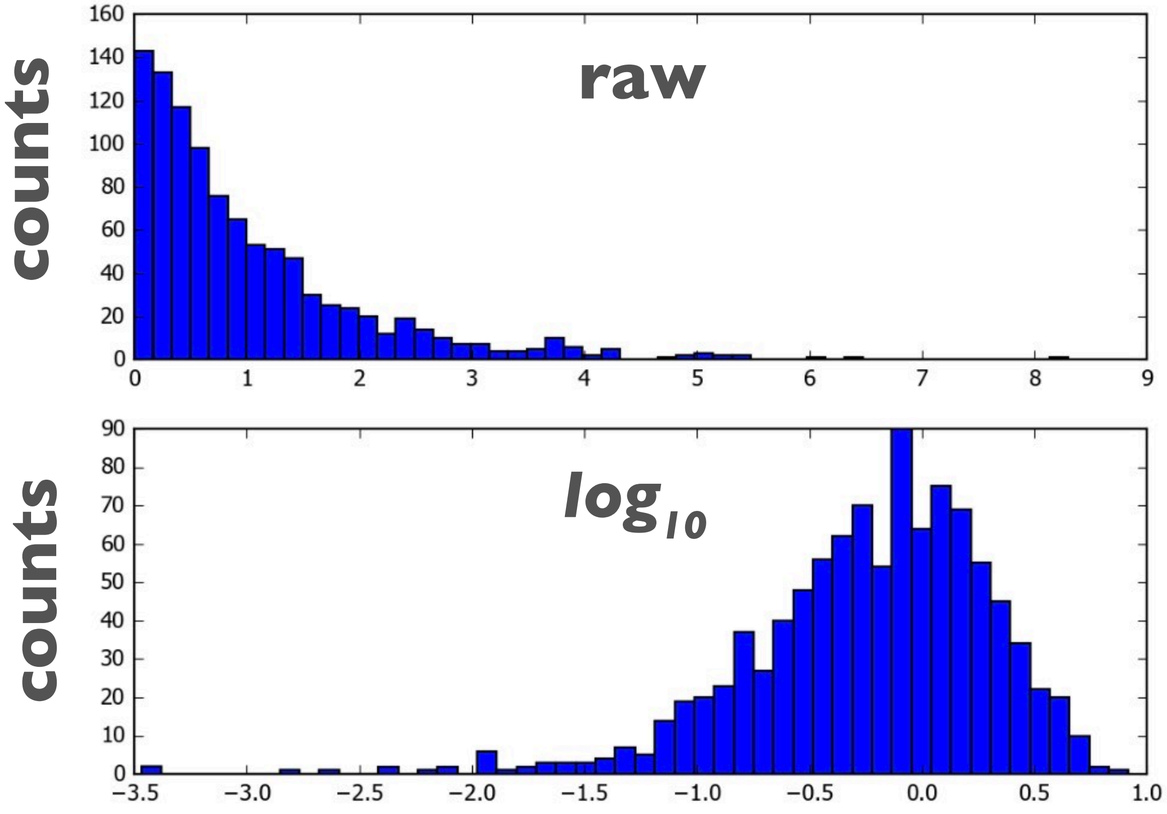
Lognormal


30
25
20
15
10
5
0
0
200
400
600
800
1000

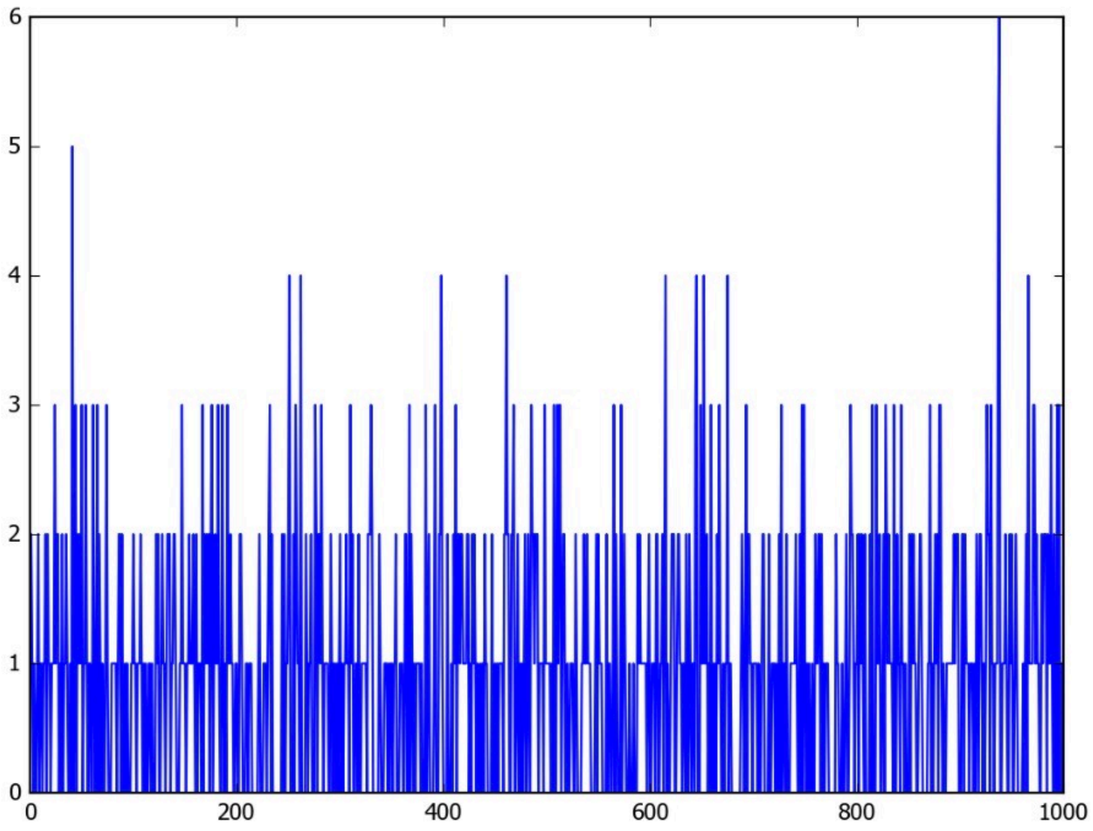

counts
raw
350
300
250
200
150
100
50
0
0
5
10
15
20
25
30
counts
log_{10}
60
50
40
30
20
10
0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

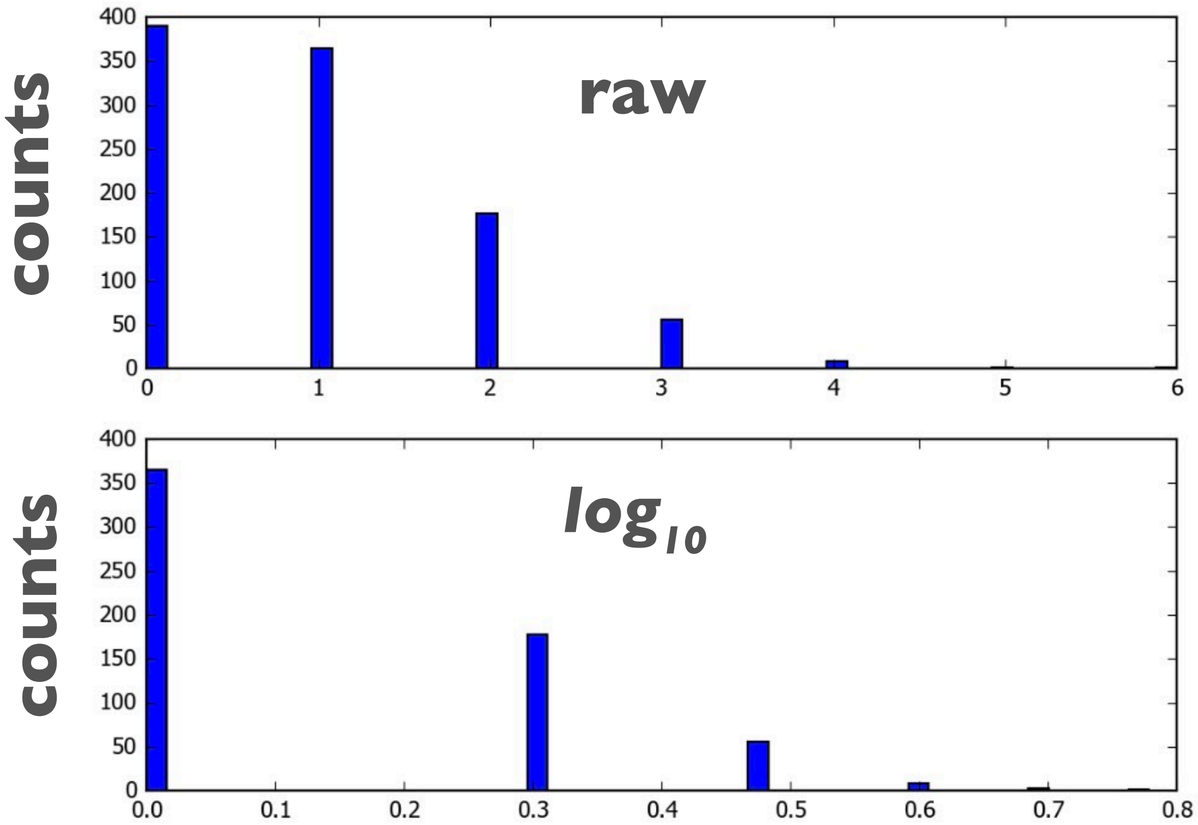
Exponential



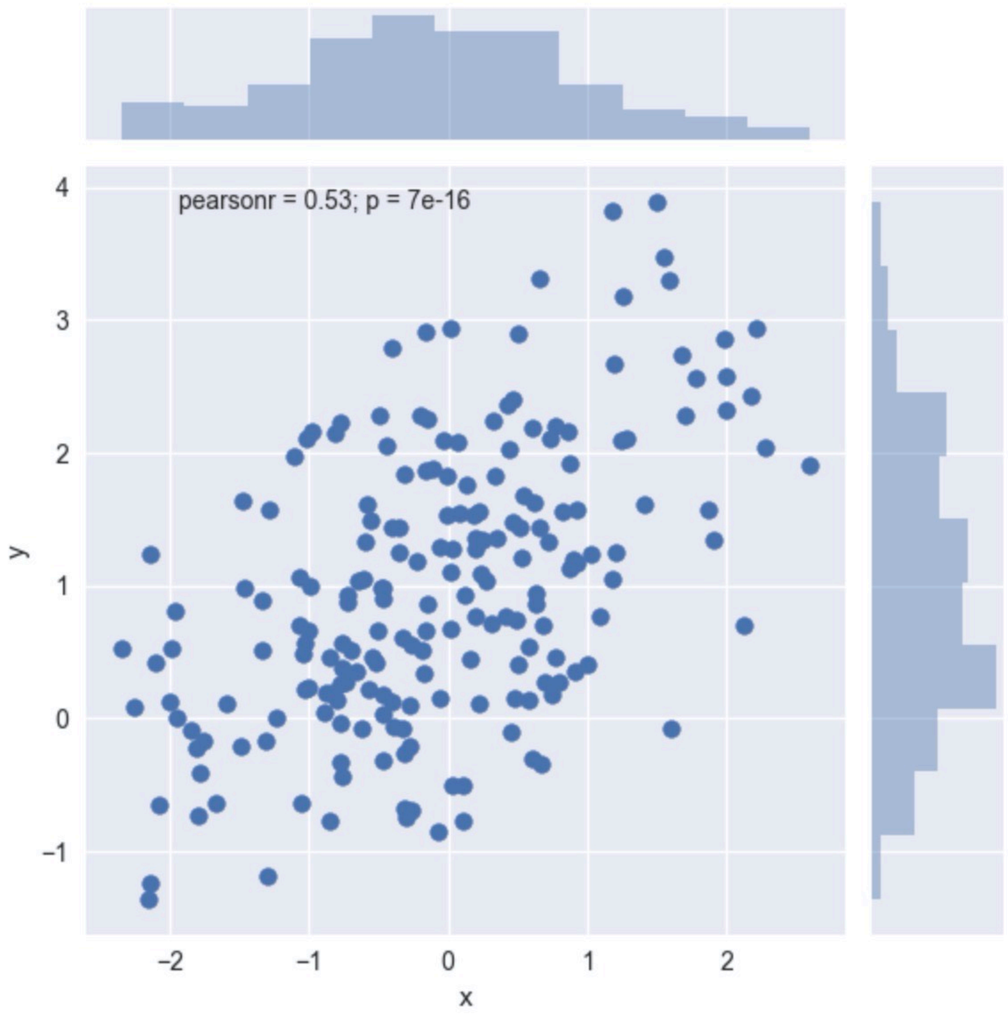


Poisson

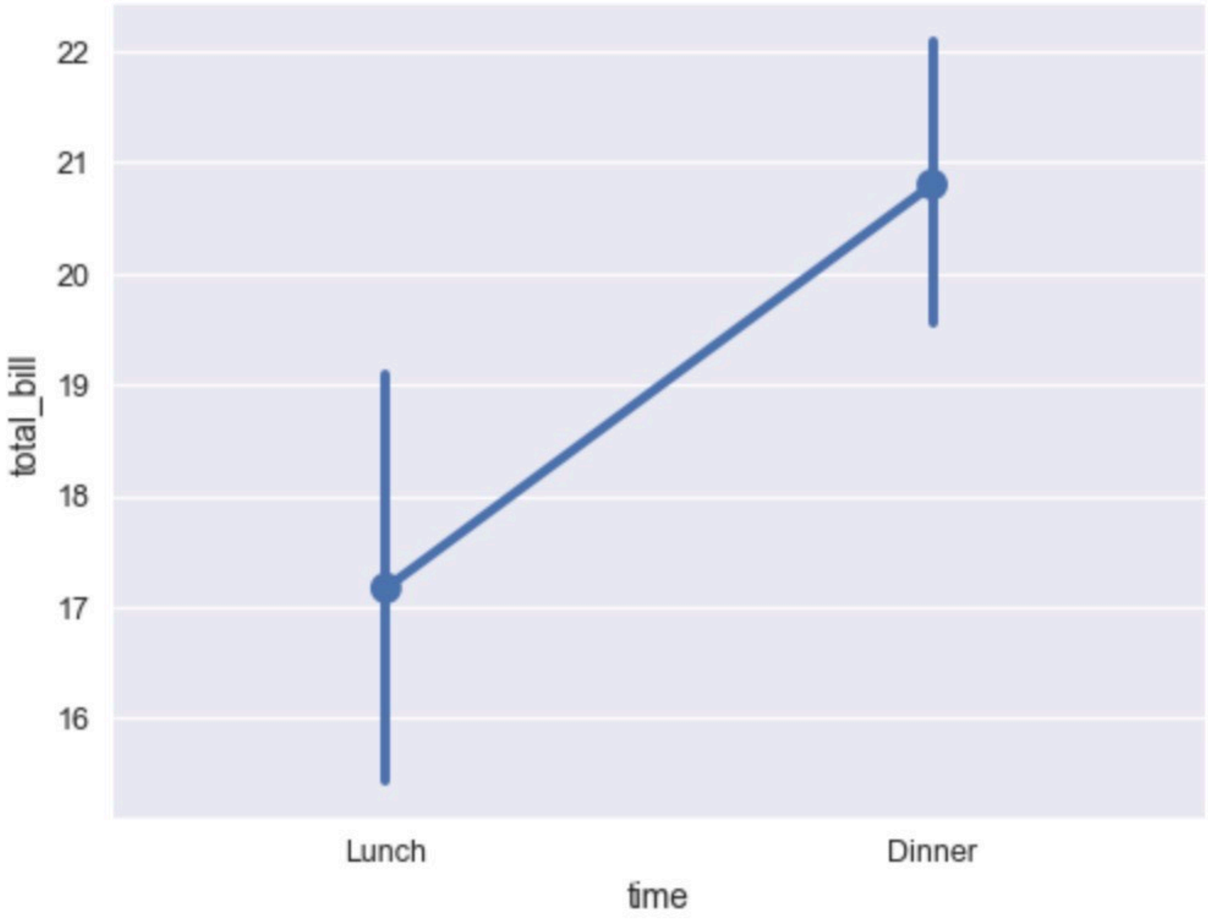

0
1
2
3
4
5
6
0
200
400
600
800
1000


counts
raw
0
50
100
150
200
250
300
350
400
0
1
2
3
4
5
6
counts
log10
0
50
100
150
200
250
300
350
400
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8

Data distributions



Data distributions



Central limit theorem

For a random sample from a distribution (any distribution!) with (finite) mean and (finite) variance, if n is sufficiently large then the sample mean follows an approximate normal distribution

```
%load_ext autoreload
%matplotlib inline
%config InlineBackend.figure_format = 'retina' # high res plotting

import random
import numpy as np
from scipy import stats

import seaborn as sns # not necessary but I like seaborn
sns.set_style('whitegrid')
sns.set_style("whitegrid", {'axes.grid' : False})

import matplotlib
import matplotlib.pyplot as plt

# plot settings
font = {'family' : 'Bitstream Vera Sans',
        'weight' : 'regular',
        'size'   : 13}
figure = {'figsize' : (10,4)}

matplotlib.rc('font', **font)
matplotlib.rc('figure', **figure)
```

```
%%time
# magic that tells you how long the cell takes to run

num_rolls = 100000 # how many simulated die rolls?

# define the function to simulate die rolls
def rollDie(number):
    roll = []
    for i in range(number):
        # note this appends the result at the end of the list every iteration
        roll = np.append(roll, random.randint(1,6))
    return roll

simulated_rolls = rollDie(num_rolls)
```

```
CPU times: user 2.05 s, sys: 99.5 ms, total: 2.15 s
Wall time: 2.2 s
```

~2.5 seconds to simulate 100,000 die rolls is a long time!

```
%%time

# note this version creates a numpy vector of zeros first, and rather than appending a number at the end of a
# such as was done above, it rewrites the zeros with the result as it moves through the loop.

# this is called initializing your data, and can speed things up
# the reason this is faster is because when you append, as we did above, numpy has to grow the array each loop

num_rolls = 100000

def rollDie(number):
    roll = np.zeros((number, 1)) # initialize your vector
    for i in range(number):
        roll[i] = random.randint(1,6)
    return roll

simulated_rolls = rollDie(num_rolls)
```

```
CPU times: user 129 ms, sys: 2.4 ms, total: 131 ms
Wall time: 132 ms
```

~0.15 seconds to simulate 100,000 die rolls; much better!

This is a 15-20x speedup!

```
%%time

## now let's get rid of the for loop, and take advantage of Python list comprehension

num_rolls = 100000

simulated_rolls = [random.randint(1,6) for i in range(num_rolls)]
```

```
CPU times: user 95.4 ms, sys: 3.14 ms, total: 98.5 ms
Wall time: 99 ms
```

~0.1 seconds to simulate 100,000 die rolls!
The moral is to always profile your code!

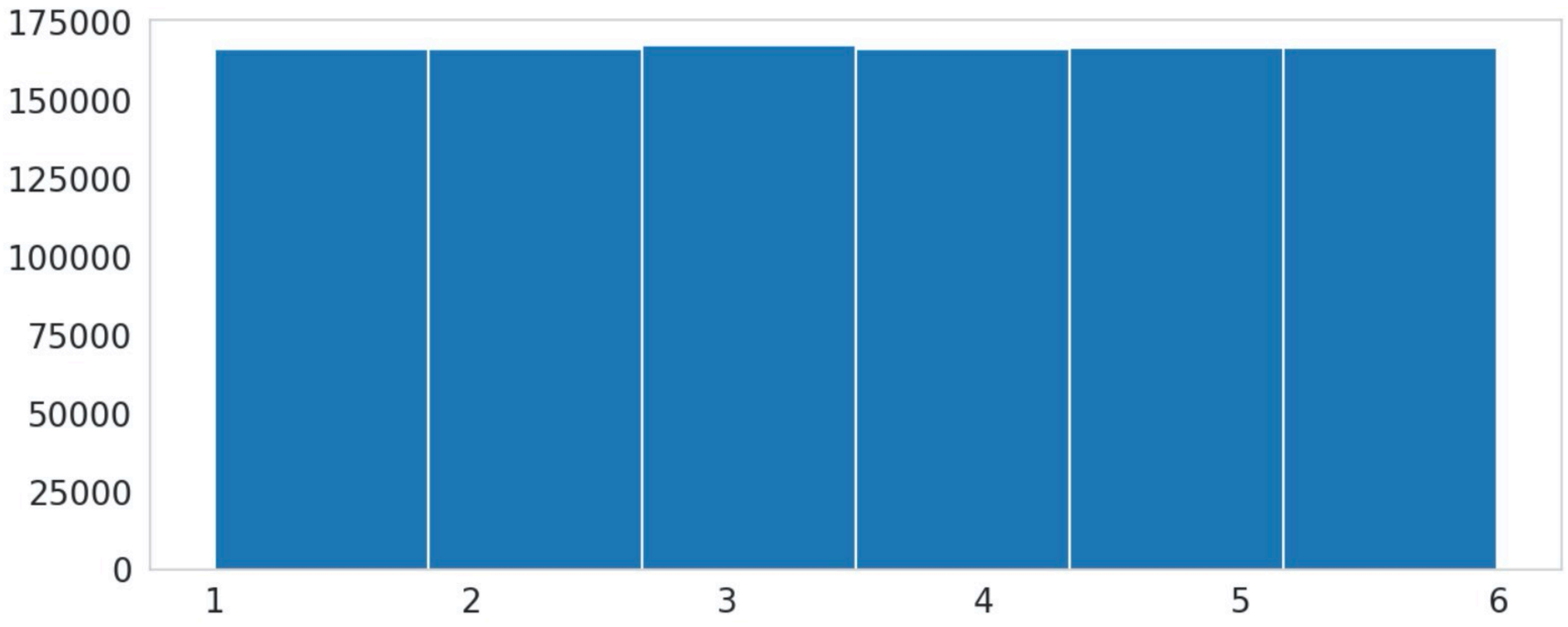
Now let's simulate 1,000,000 die rolls of a fair die using the faster function.
Also plot the histogram of the results

Aside: Code profiling

In software engineering, profiling is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. Most commonly, profiling information serves to aid program optimization.

Source: https://en.wikipedia.org/wiki/Profiling_(computer_programming)

```python
num_rolls = 1000000

simulated_rolls = [random.randint(1,6) for i in range(num_rolls)]

# np.unique finds the unique values are in our data (1, 2, 3, 4, 5, 6)
# and np.size asks how many there are (6)
# we then use that to define the number of bins we want in our histogram
number_of_bins = np.size(np.unique(simulated_rolls))

plt.hist(simulated_rolls, number_of_bins)
plt.show()
```



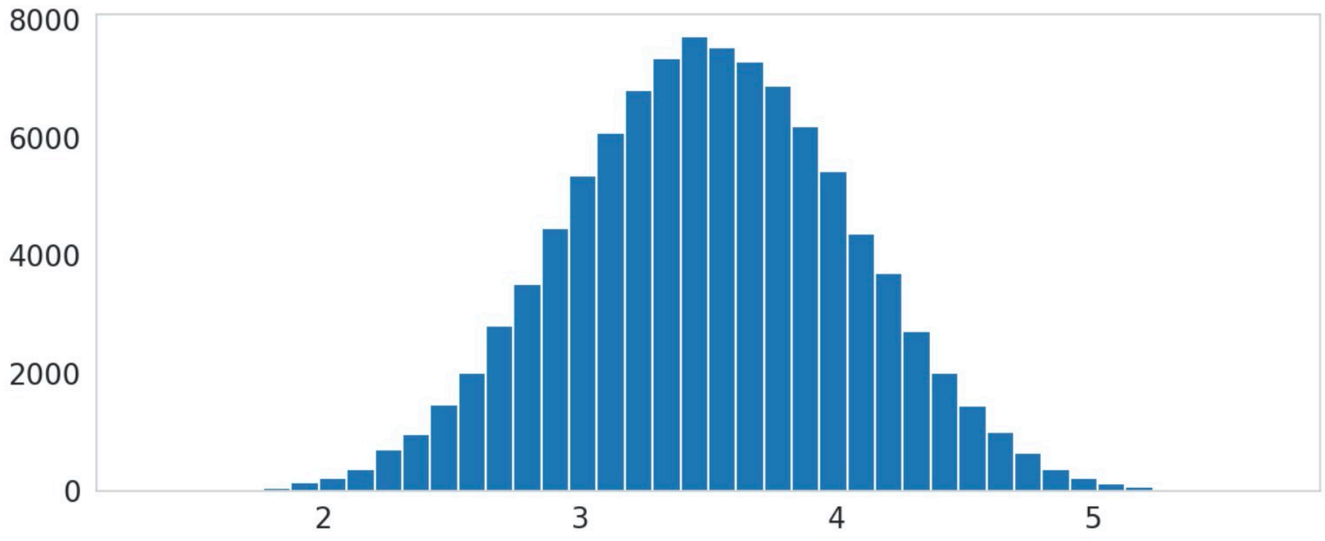
Looks decently uniform!

So let's look at the magic that is the Central Limit Theorem.

First, we break our data vector of die roll results, simulated_rolls, into chunks of 10 rolls each.

Then we average each of those 10 rolls and store the result.

```python
chuck_size = 10 # each chunk has this many rolls
samples = num_rolls//chuck_size # instead of <num_rolls> elements, we now have num_rolls/10
                                # so instead of 1,000,000 we have 100,000
roll_avg = np.zeros((samples, 1)) # initialize your vector

for i in range(samples):
    # for each loop we need to average from 0:9, 10:19, 20:29, and so on
    # this gives us a low range and a high range to average across
    # the low range (0,10,20,etc) is just i*10
    # the high range is that, plus 9
    low_range = i*10
    high_range = low_range+9
    roll_avg[i] = np.mean(simulated_rolls[low_range:high_range]) # average the chunks

# plot the histogram!
plt.hist(roll_avg, np.size(np.unique(roll_avg)))
plt.show()
```




```
np.mean((1,2,3,4,5,6)), np.mean(roll_avg)
```

```
(3.5, 3.5005277777777777)
```

Great! Just like the CLT says, averages of samples will result in normally distributed averages.

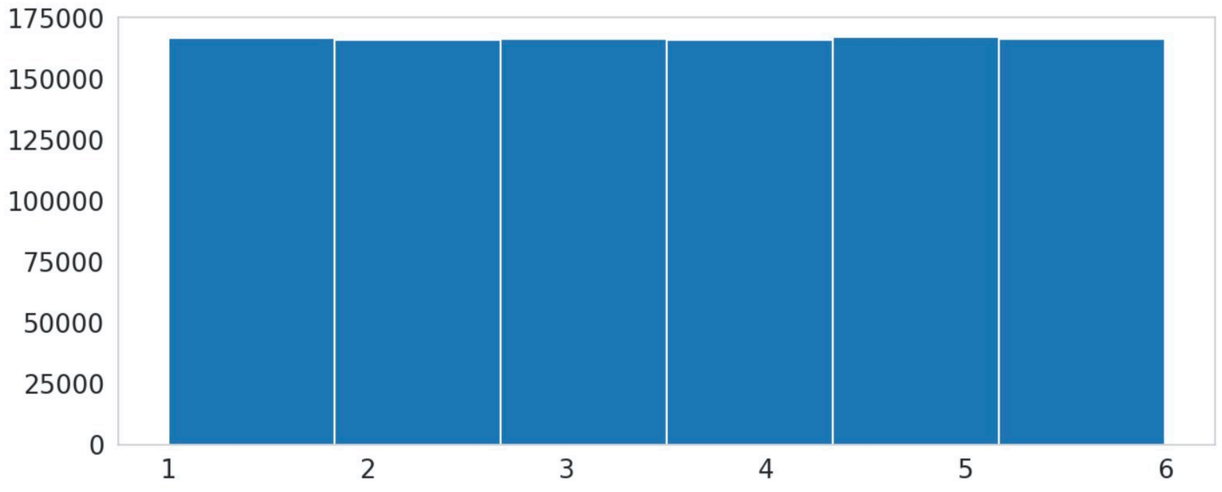
Unsurprusingly, the mean appears to be ~3.5, which is the mean of (1,2,3,4,5,6)

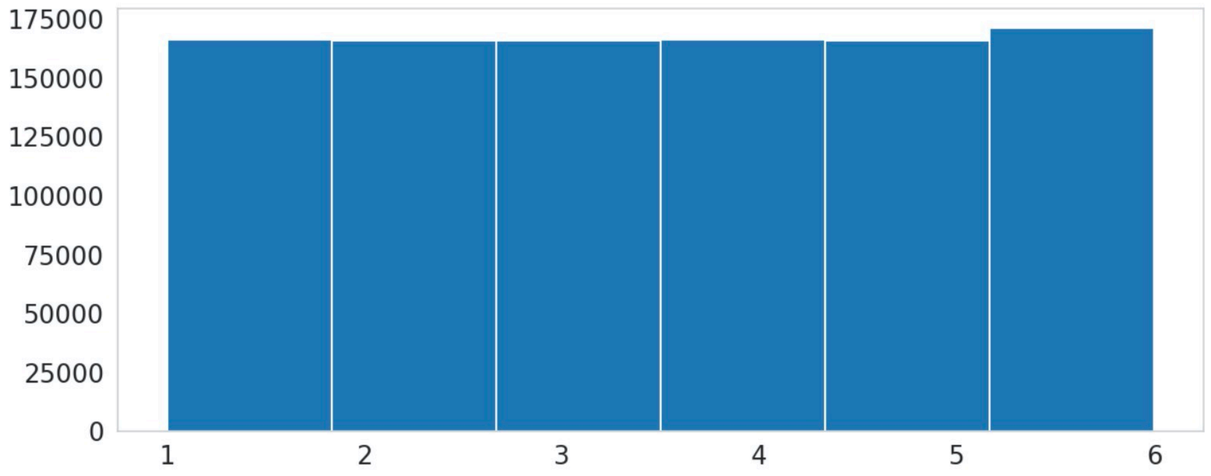
We're going to show how we can leverage the CLT to run statistical analyses that assume normality on data that are otherwise non-normal (such as die roll probabilities).

Note that although we're doing die rolls here, these could just as easily be star-ratings for Yelp, Uber, etc. or thumbs up/down for songs on Padora or Netflix shows.

```python
# now, let's create a second set of fake die rolls
simulated_biased_rolls = rollDie(num_rolls)

# we're going to replace approximately 0.5% of each non-six result with a 6 to simulated a biased die
percent_bias = 0.005
replace_size = np.around((num_rolls/number_of_bins)*percent_bias)
replace_size = replace_size.astype(int)

# loop from 1 to 5
for i in range((number_of_bins-1)):
    idx = np.where(simulated_biased_rolls==(i+1)) # find each instance of i+1 as the die result
    replace_idx = np.random.choice(idx[0], size=replace_size, replace=True)
    simulated_biased_rolls[replace_idx] = 6
```

```python
plt.hist(simulated_rolls, 6)
plt.show()

plt.hist(simulated_biased_rolls, 6)
plt.show()
```

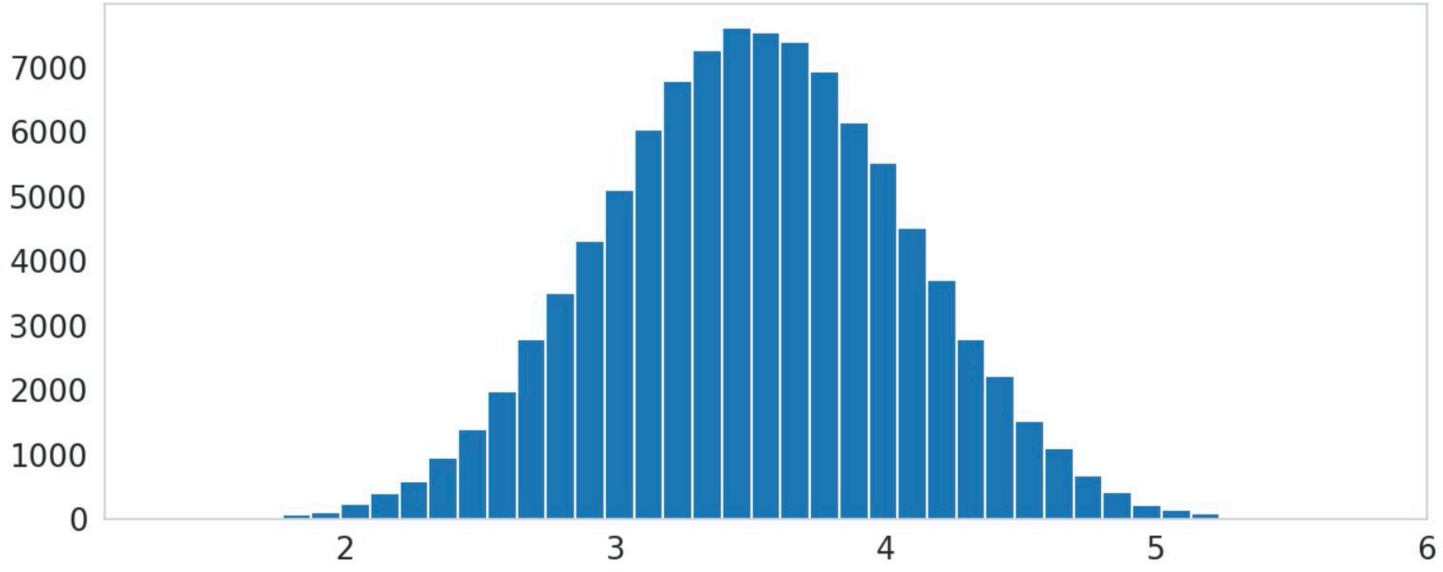




Okay it looks like there's a *little* bump at 6, but it's not hugely obvious.

Let's look at the histogram of the sampels from the biased rolls and see if we see anything obvious.

```python
chuck_size = 10
samples = num_rolls//chuck_size
biased_roll_avg = np.zeros((samples, 1))

for i in range(samples):
    low_range = i*10
    high_range = low_range+9
    biased_roll_avg[i] = np.mean(simulated_biased_rolls[low_range:high_range]) # average the chunks

# plot the histogram!
plt.hist(biased_roll_avg, np.size(np.unique(biased_roll_avg)))
plt.show()
```



Well, okay. Looks about the same as above...
What's the mean of this one?

```
np.mean(biased_roll_avg)
```

```
3.5132655555555554
```

Just *ever so slightly* greater than the 3.5 we'd expect.
Is this significant though? How can we check?

This is where the "art" of data science starts to come into play!

How can we determine (statistically) if our die is loaded?

We can start with the assumption that any two distributions of random samples of the means of fair die will be centered around 3.5.

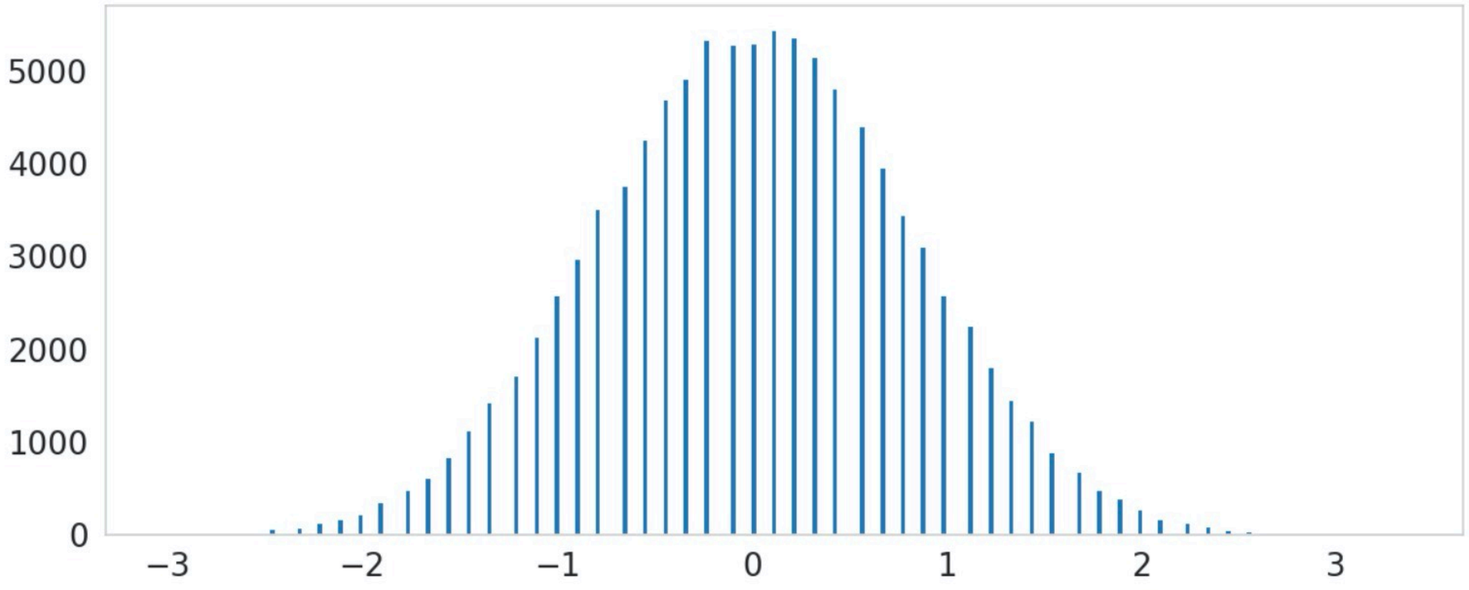
Additionally we can say that, given this, if we take a random sample of the mean of 10 die rolls from die_1, and a random sample of the mean of 10 die rolls from die_2, the difference of those means should itself be normally distributed (thanks CLT!) around 0.

That is, any differeces should cancel out, given enough data.

If, however, one of the die is biased—such as toward 6, in our case—that bias should force the mean of the samples to be slightly *greater* than the expected mean of 3.5, and should force the mean of the *differences* between the die to be slightly greater than 0.

Armed with this, we can now perform a very simple independent samples t-ttest of the distribution of the differences against the assumption that the mean of the differences should be 0.

```python
# get the distribution of the differences of means
chuck_size = 10
samples = num_rolls//chuck_size
roll_diff = np.zeros((samples, 1))

for i in range(samples):
    low_range = i*10
    high_range = low_range+9
    # get the mean of the differences between the die
    clean_mean = np.mean(simulated_rolls[low_range:high_range])
    biased_mean = np.mean(simulated_biased_rolls[low_range:high_range])
    roll_diff[i] = biased_mean-clean_mean

# plot the histogram!
plt.hist(roll_diff, np.size(np.unique(roll_diff)))
plt.show()
```




```
np.mean(roll_diff)
```

```
0.013033333333333333
```

The distribution of differences between means looks normal, too!

Also note that the mean of these differences is, indeed, slightly greater than 1.

Now we can use a standard t-test to look at how big this difference is, and whether it's "significant".

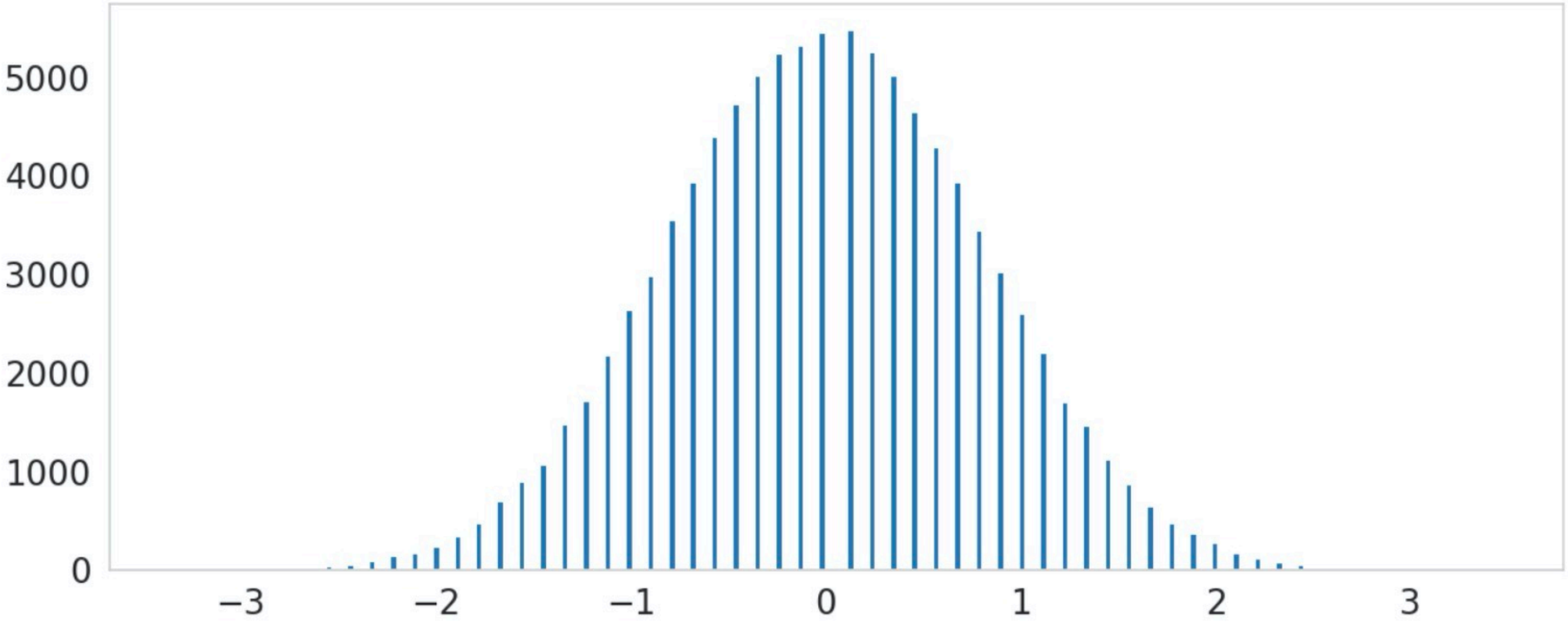
```python
t,p = stats.ttest_1samp(roll_diff, popmean=0) # compare our differences against the null, which is a mean of 0
print(['t-value: ', t[0], '; p-value: ', p[0]])
```

```
['t-value: ', 5.111394619326785, '; p-value: ', 3.203765113691647e-07]
```

Yes! It is significant.

And just to sanity check, let's do the same thing, but this time compare our un-biased die against a second simulation of an un-biased die.

```python
simulated_rolls_taketwo = rollDie(num_rolls) # simulate a second, unbiased set of rolls

chuck_size = 10
samples = num_rolls//chuck_size
unbiased_roll_diff = np.zeros((samples, 1))

for i in range(samples):
    low_range = i*10
    high_range = low_range+9
    # get the mean of the differences between the die
    clean_mean = np.mean(simulated_rolls[low_range:high_range])
    clean_mean_taketwo = np.mean(simulated_rolls_taketwo[low_range:high_range])
    unbiased_roll_diff[i] = clean_mean_taketwo-clean_mean

# plot the histogram!
plt.hist(unbiased_roll_diff, np.size(np.unique(unbiased_roll_diff)))
plt.show()
t,p = stats.ttest_1samp(unbiased_roll_diff, popmean=0) # compare our differences against the
                                                        # null, which is a mean of 0
print(['t-value: ', t[0], '; p-value: ', p[0]])
```




```
['t-value: ', -0.5704611270639239, '; p-value: ', 0.5683662621453525]
```

No difference between them!

Nice job, statistics.

COGS 108
Data Science in Practice

Linear modeling

UC San Diego

A line

slope

$$y = mx + b$$

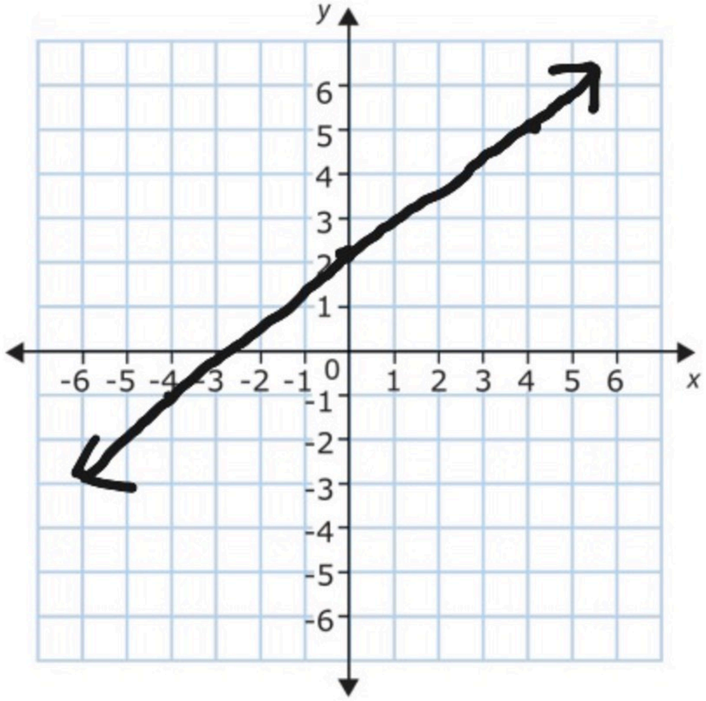
intercept

A line

slope

$$y = mx + b$$

intercept



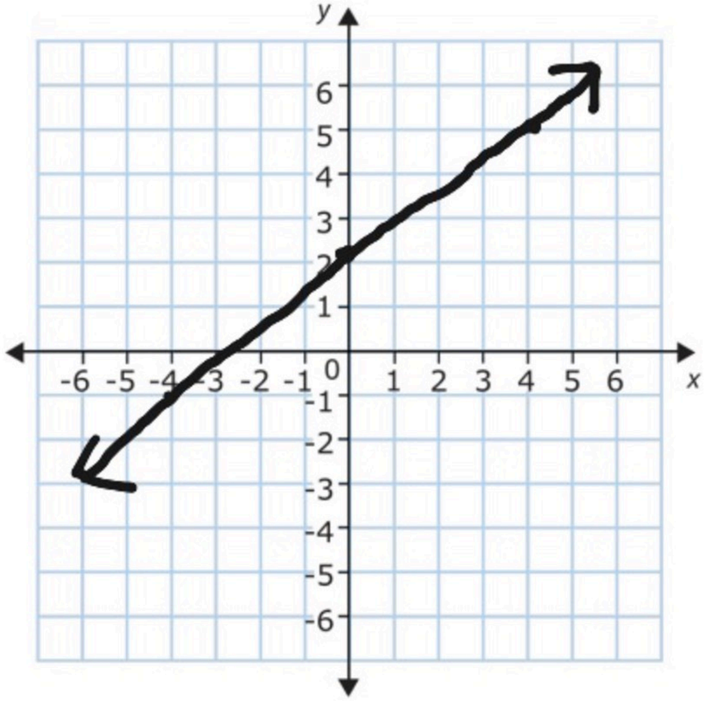
$$y = \frac{3}{4}x + 2$$

Ordinary Least Squares (OLS)

slope

$$y = mx + b$$

intercept



$$y = \frac{3}{4}x + 2$$

intercept

error

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$$

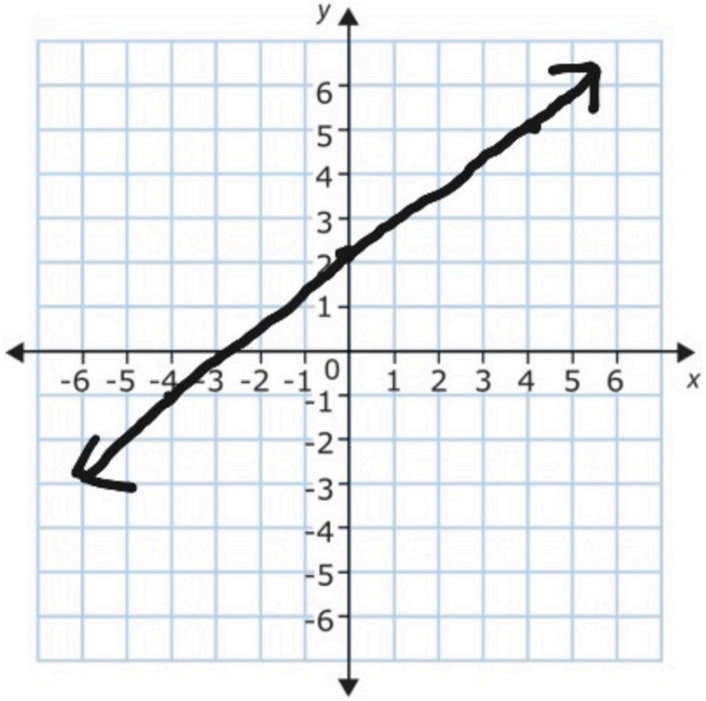
slope

Ordinary Least Squares (OLS)

slope

$$y = mx + b$$

intercept



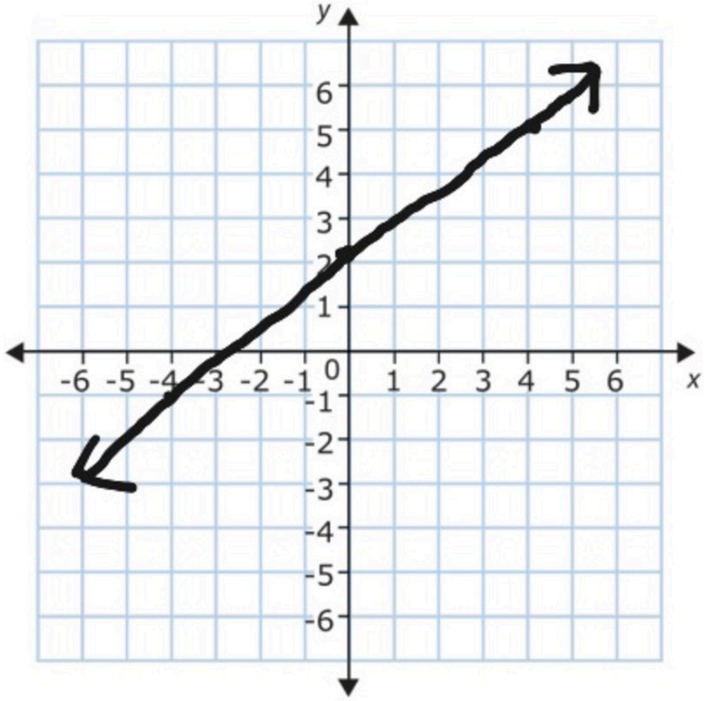
$$y = \frac{3}{4}x + 2$$

Ordinary Least Squares (OLS)

slope

$$y = mx + b$$

intercept



$$y = \frac{3}{4}x + 2$$

intercept

error

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$$

slope

Ordinary Least Squares (OLS)

$$y_i = \underbrace{\beta_0}_{\text{intercept}} + \underbrace{\beta_1}_{\text{slope}} x_i + \underbrace{\varepsilon_i}_{\text{error}}$$

Ordinary Least Squares (OLS)

parameter
intercept

residual
error

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$$

slope
parameter

Ordinary Least Squares (OLS)

parameter
intercept

residual
error

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$$

slope
parameter

Goal: find values for the parameters to provide a "best" fit for the data

Ordinary Least Squares (OLS)

parameter
intercept

residual
error

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$$

slope
parameter

Goal: find values for the parameters to provide a "best" fit for the data

how?

Ordinary Least Squares (OLS)

parameter
intercept

residual
error

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$$

slope
parameter

minimize this!

Ordinary Least Squares (OLS)

parameter
intercept

residual
error

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$$

slope
parameter

minimize this!

how?

Ordinary Least Squares (OLS)

parameter
intercept

residual
error

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$$

slope
parameter

$$\min \sum_{i=1}^{n} \varepsilon_i^2$$

Ordinary Least Squares (OLS)

parameter
intercept

residual
error

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$$

slope
parameter

$$\min \sum_{i=1}^{n} \varepsilon_i^2$$

why squared?

Ordinary Least Squares (OLS)

parameter
intercept

residual
error

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$$

slope
parameter

$$\min \sum_{i=1}^{n} \varepsilon_i^2$$

$$y_i - \beta_0 - \beta_1 x_i = \varepsilon_i$$

Ordinary Least Squares (OLS)

parameter
intercept

residual
error

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$$

slope
parameter

$$\min \sum_{i=1}^{n} \varepsilon_i^2$$

$$y_i - \beta_0 - \beta_1 x_i = \varepsilon_i$$

$$\min \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_i)^2$$

Ordinary Least Squares (OLS)

parameter
intercept

residual
error

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$$

slope
parameter

$$\beta_0 = \mu_y - \beta_1 \mu_x$$

$$\beta_1 = r_{xy} \frac{\sigma_y}{\sigma_x}$$

Ordinary Least Squares (OLS)

parameter
intercept

residual
error

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$$

slope
parameter

$$\beta_0 = \mu_y - \beta_1 \mu_x$$

WHAT DOES THIS MEAN?!?

$$\beta_1 = r_{xy} \frac{\sigma_y}{\sigma_x}$$

Ordinary Least Squares (OLS)

parameter
intercept

residual
error

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$$

slope
parameter

$$\beta_0 = \mu_y - \beta_1 \mu_x$$

Ordinary Least Squares (OLS)

parameter
intercept

residual
error

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$$

slope
parameter

$$\beta_0 = \mu_y - \beta_1 \mu_x$$

when we see the average value of x, we predict the average value of y

Ordinary Least Squares (OLS)

parameter
intercept

residual
error

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$$

slope
parameter

$$\beta_0 = \mu_y - \beta_1 \mu_x$$

when we see the average value of x, we predict the average value of y

$$\beta_1 = r_{xy} \frac{\sigma_y}{\sigma_x}$$

Ordinary Least Squares (OLS)

parameter
intercept

residual
error

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$$

slope
parameter

$$\beta_0 = \mu_y - \beta_1 \mu_x$$

when we see the average value of *x*, we predict the average value of *y*

$$\beta_1 = r_{xy} \frac{\sigma_y}{\sigma_x}$$

when the input increases by σ_x, the prediction increases by the *x*,*y* correlation

COGS 108
Data Science in Practice

Modeling and Model Validation

UC San Diego

Simple models in practice

Import libraries

We will be using the library statsmodels to run our linear regression

```
import numpy as np
import statsmodels.formula.api as smf
import pandas as pd
import scipy as sp

%matplotlib notebook
%config InlineBackend.figure_format = 'retina'
%matplotlib inline
import matplotlib.pyplot as plt
```

Simple models in practice

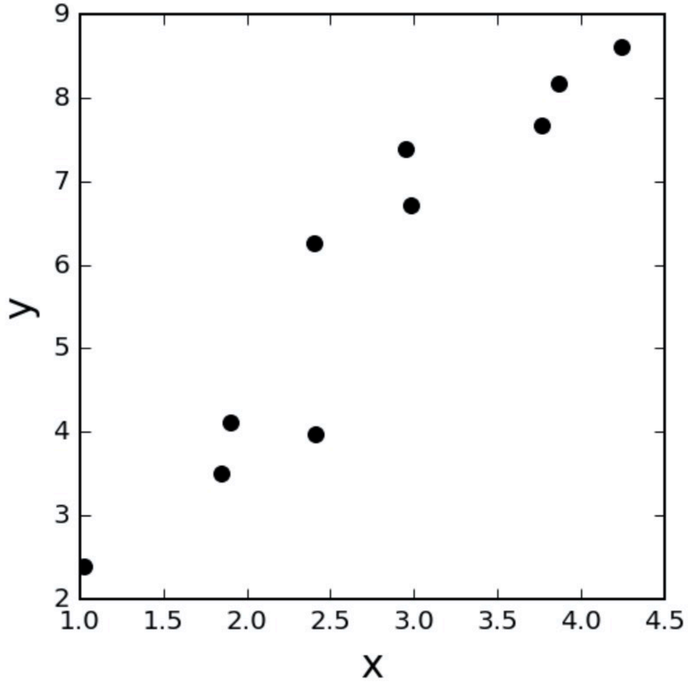
1a. Fitting a line

$$y_i = \beta_0 + \beta_1 x_i + \epsilon_i$$

```
# Define true statistics relating x and y
N_points = 10
true_beta0 = 0
true_beta1 = 2
noise_stdev = 1

# Set random seed
np.random.seed(0)

# Generate correlated data
x = np.random.randn(N_points) + 2
y = true_beta0 + true_beta1*x + np.random.randn(N_points)*noise_stdev
print('x=', x)
print('y=', y)
```

Simple models in practice

```
# Plot x and y
plt.figure(figsize=(4,4))
plt.plot(x, y, 'k.', ms=12)
plt.xlabel('x',size=15)
plt.ylabel('y',size=15)
plt.show()
```



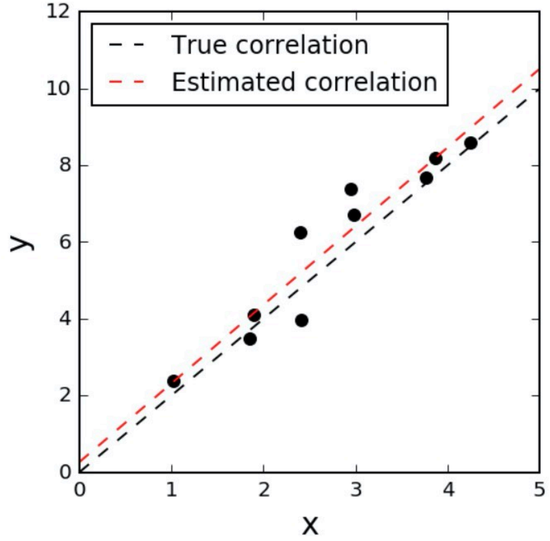
Simple models in practice

```
# Fit line to data
A = np.vstack([x, np.ones(len(x))]).T
m, b = np.linalg.lstsq(A, y)[0]
print('True statistics: y =', true_beta1, '*x +', true_beta0)
print('Estimated stats: y =', m, '*x +', b)
print('R squared (fraction of variance explained) =',
      np.round(sp.stats.pearsonr(x,y)[0],2))
```

```
True statistics: y = 2 *x + 0
Estimated stats: y = 2.04994401661 *x + 0.26389814044
R squared (fraction of variance explained) = 0.95
```

Simple models in practice

```
# Plot fitted line
plt.figure(figsize=(4,4))
plt.plot(x, y, 'k.', ms=12)
plt.plot([0,5], [true_beta1*x+true_beta0 for x in [0,5]], 'k--',label='True correlation')
plt.plot([0,5], [m*x+b for x in [0,5]], 'r--',label='Estimated correlation')
plt.xlabel('x',size=15)
plt.ylabel('y',size=15)
plt.xlim((0,5))
plt.legend(loc='best')
plt.show()
```



Simple models in practice

1b. Outliers and normality of errors

Minimizing the squared error of the line fit works well if our data is normally distributed.

In this section, we simulate some data in which the error is not normally distributed, and see the fitted line is not as accurate.

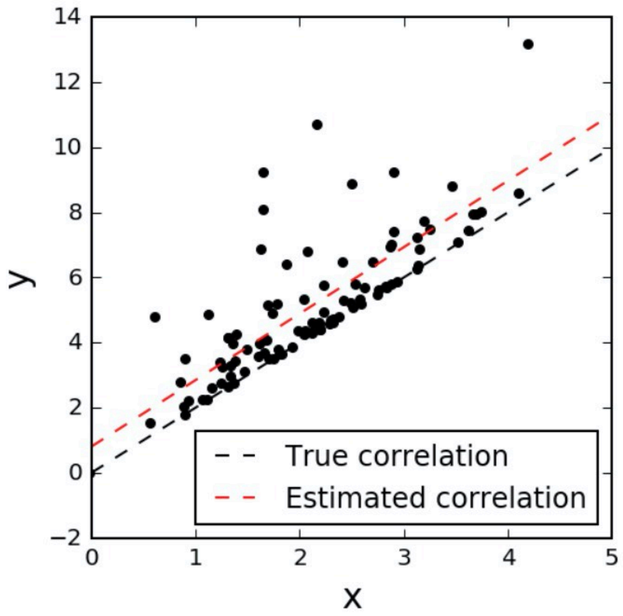
```
# Simulate data with non-normal distribution of error
np.random.seed(1)
N_points = 100
x = np.random.randn(N_points) + 2
y = true_beta0 + true_beta1*x + np.random.randn(N_points)**2
```

```
# Fit line to data
A = np.vstack([x, np.ones(len(x))]).T
m, b = np.linalg.lstsq(A, y)[0]
print('True statistics: y =', true_beta1, '*x +', true_beta0)
print('Estimated stats: y =', m, '*x +', b)
print('R squared (fraction of variance explained) =',np.round(sp.stats.pearsonr(x,y)[0],2))
```

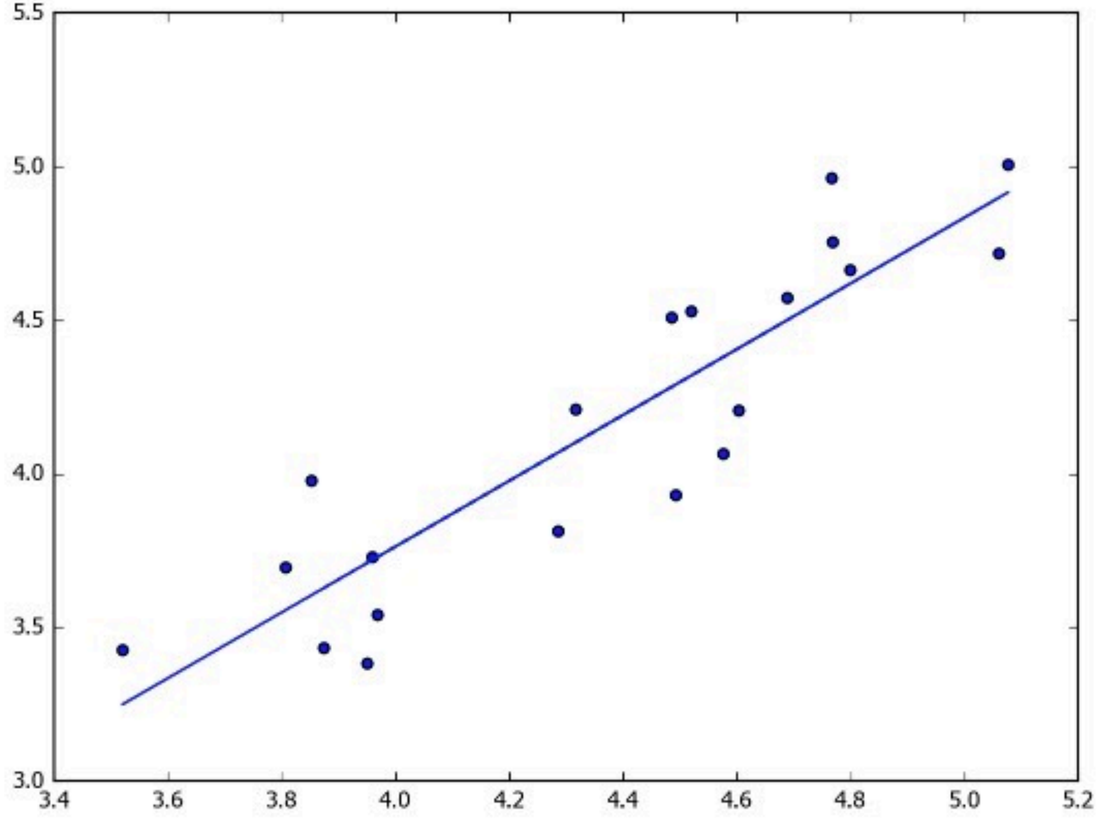
```
True statistics: y = 2 *x + 0
Estimated stats: y = 2.04202383003 *x + 0.805367000302
R squared (fraction of variance explained) = 0.82
```

Simple models in practice

```
# Plot fitted line
plt.figure(figsize=(4,4))
plt.plot(x, y, 'k.', ms=8)
plt.plot([0,5], [true_beta1*x+true_beta0 for x in [0,5]], 'k--',label='True correlation')
plt.plot([0,5], [m*x+b for x in [0,5]], 'r--', label='Estimated correlation')
plt.xlabel('x',size=15)
plt.ylabel('y',size=15)
plt.xlim((0,5))
plt.legend(loc='best')
plt.show()
```

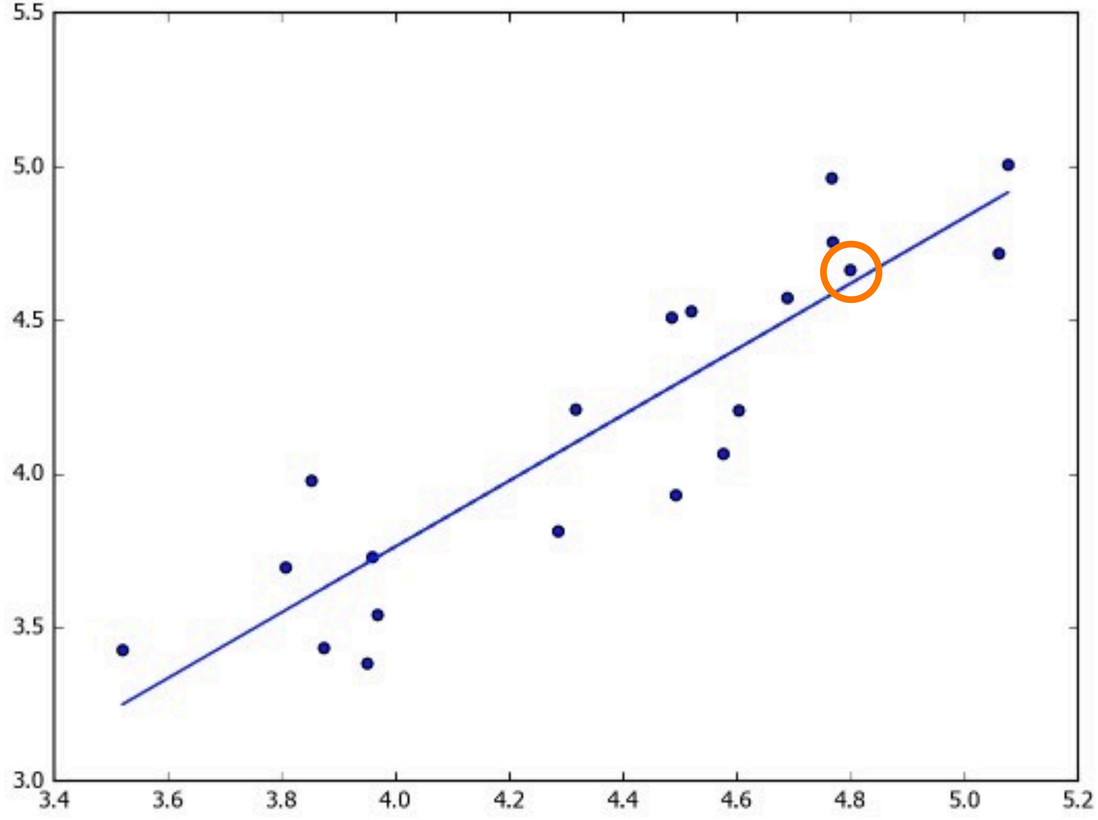


Outliers!



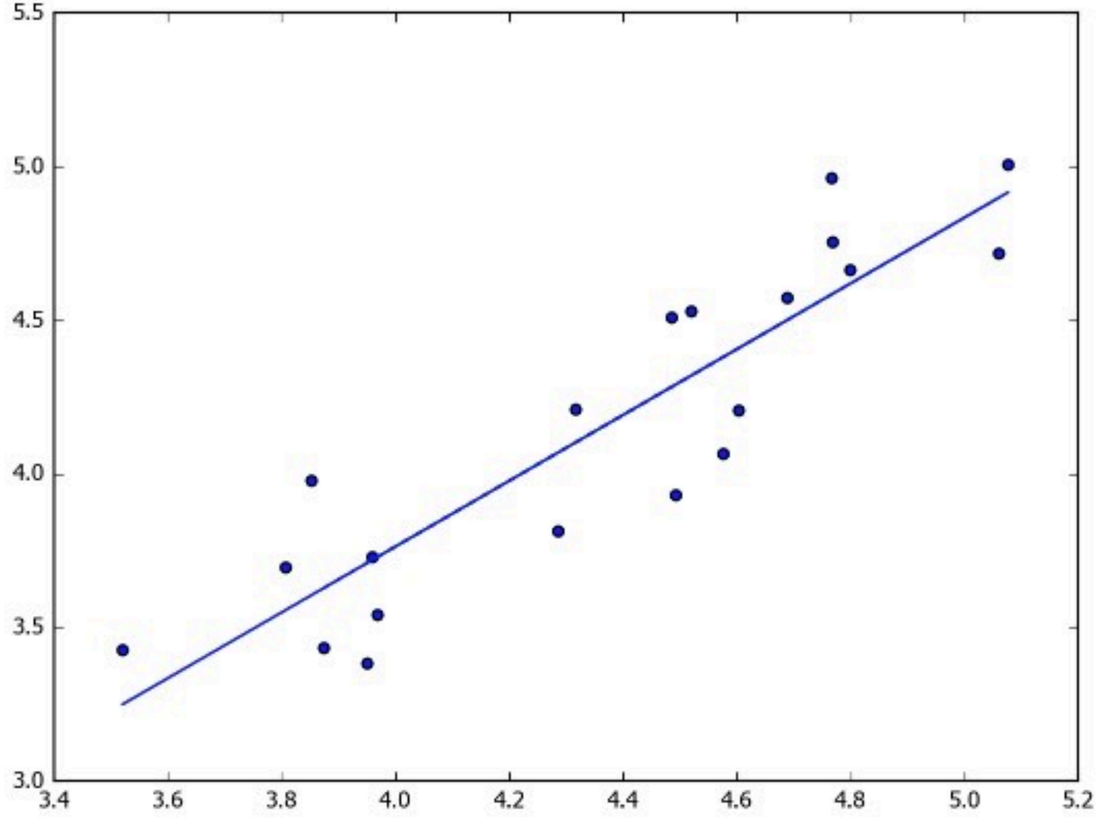

```
r = 0.899192316859 ; p = 7.04649901141e-08
```

Outliers!

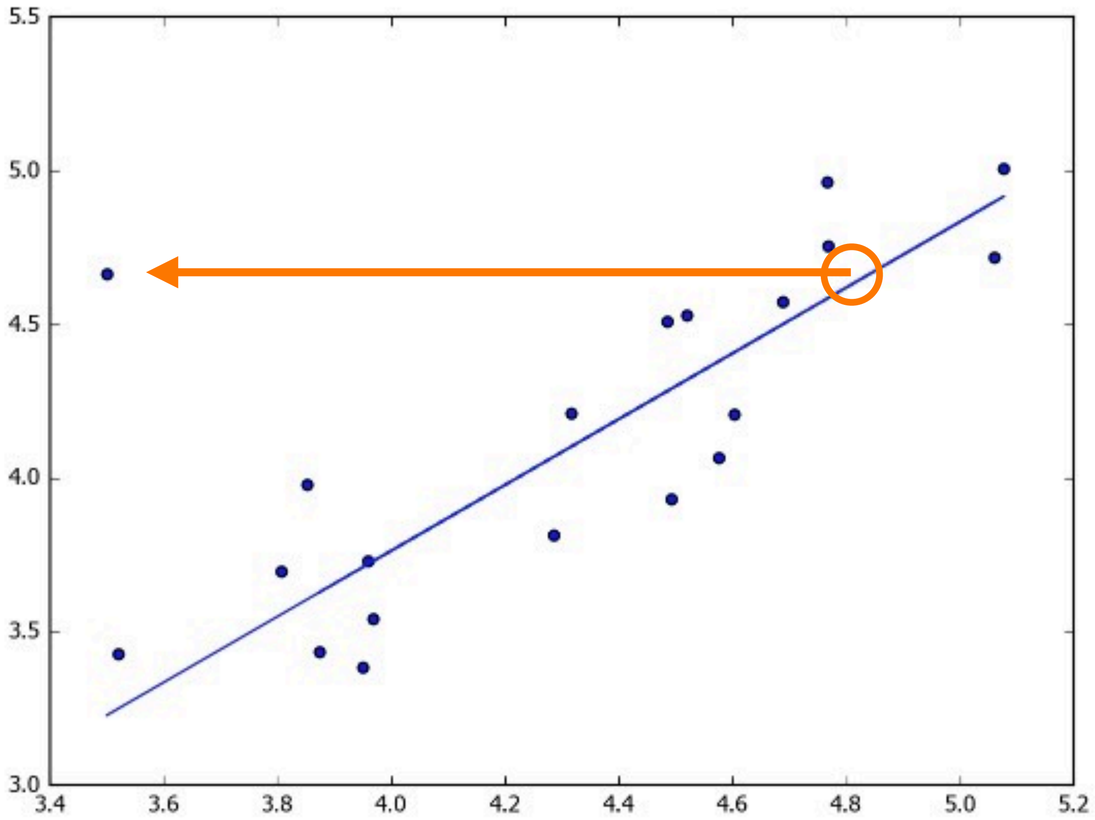


r = 0.899192316859 ; p = 7.04649901141e-08

Outliers!

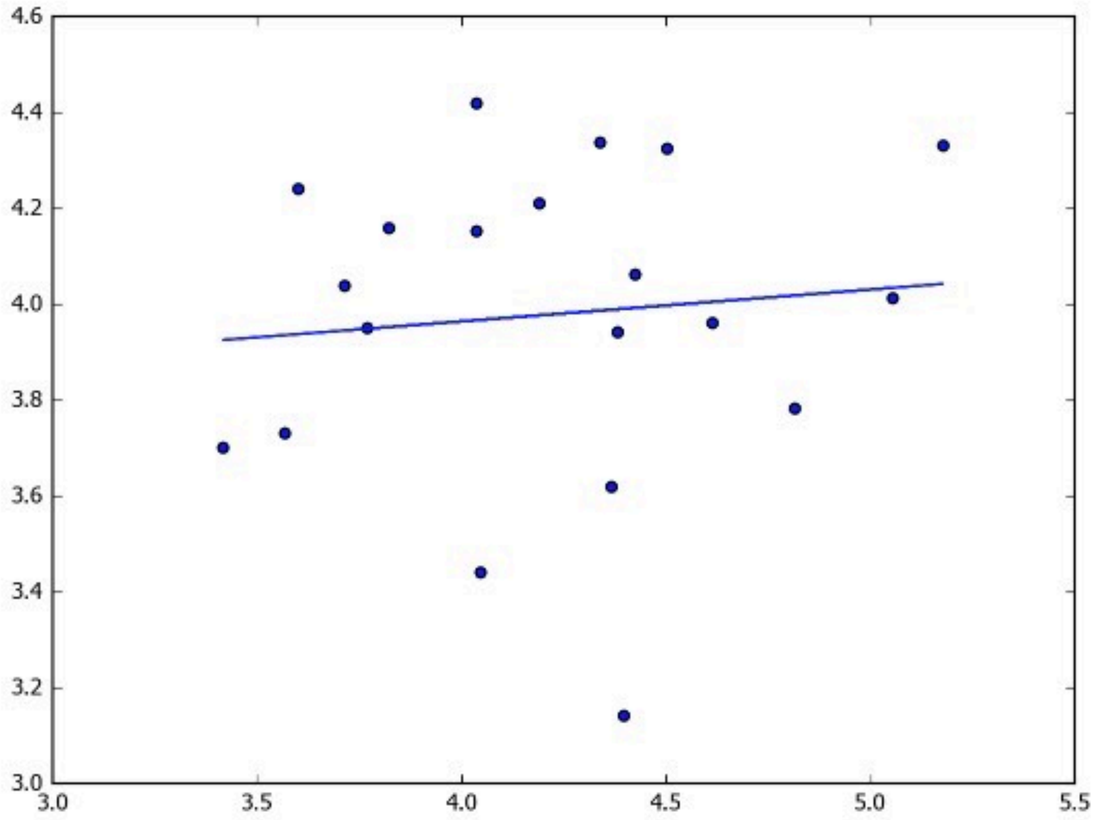


r = 0.899192316859 ; p = 7.04649901141e-08



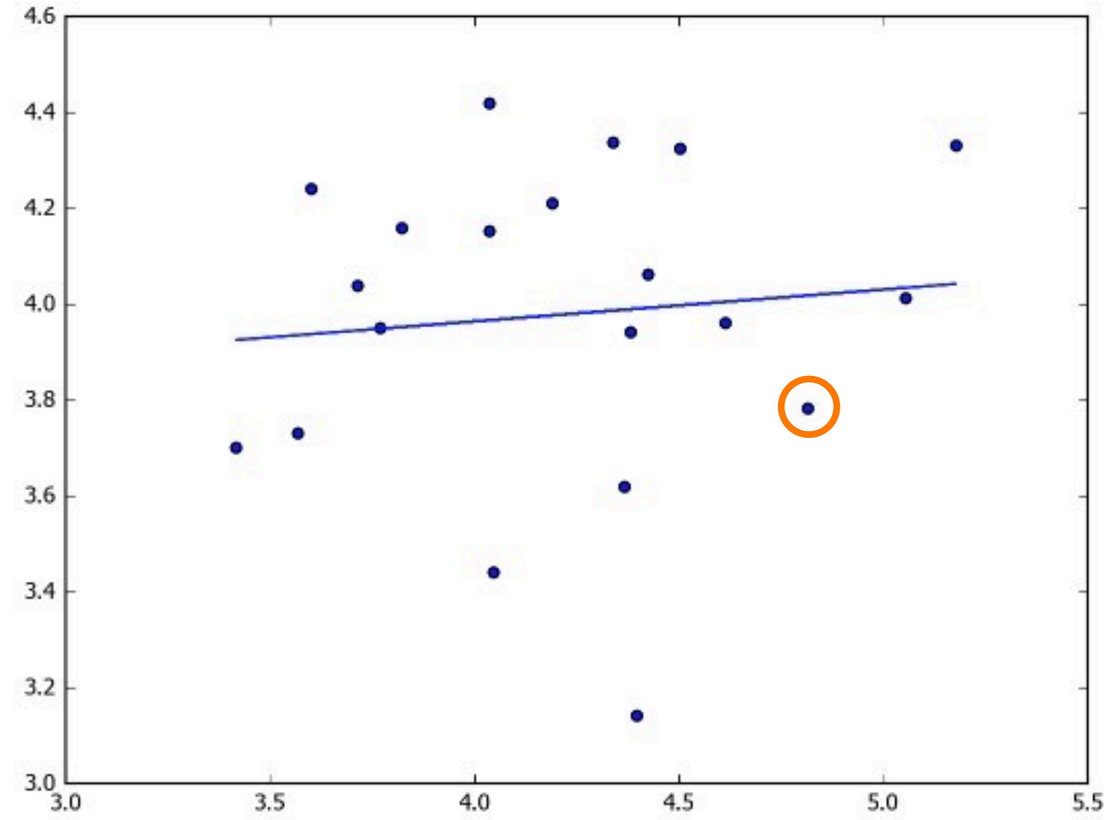
0.709892176011 0.000454396182138

Outliers!



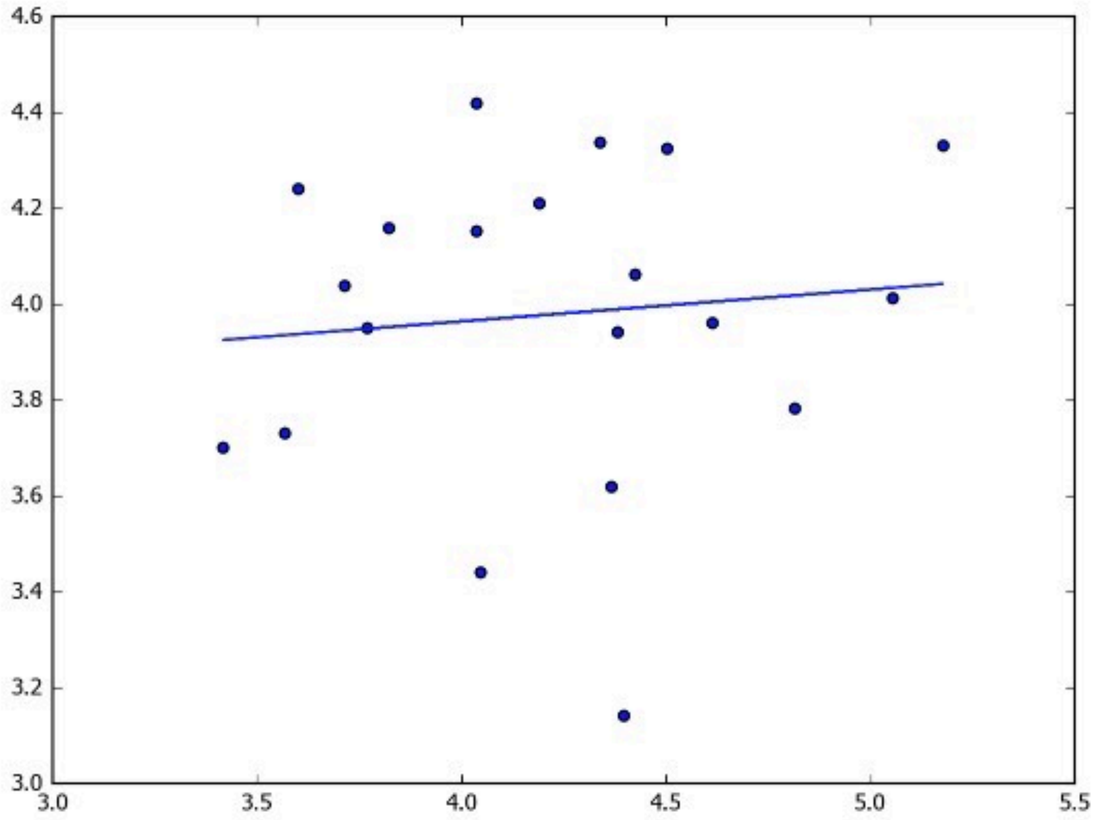

```
r = 0.0982029696446 ; p = 0.680415082529
```

Outliers!

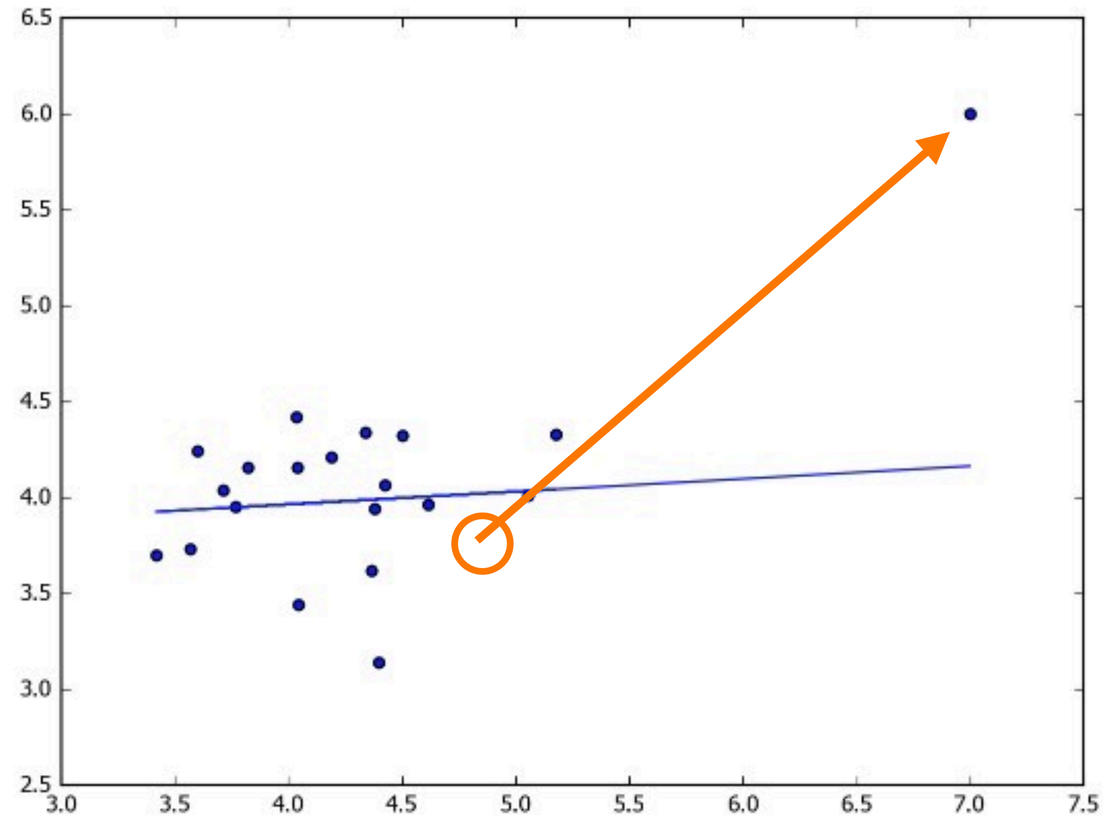



```
r = 0.0982029696446 ; p = 0.680415082529
```

Outliers!

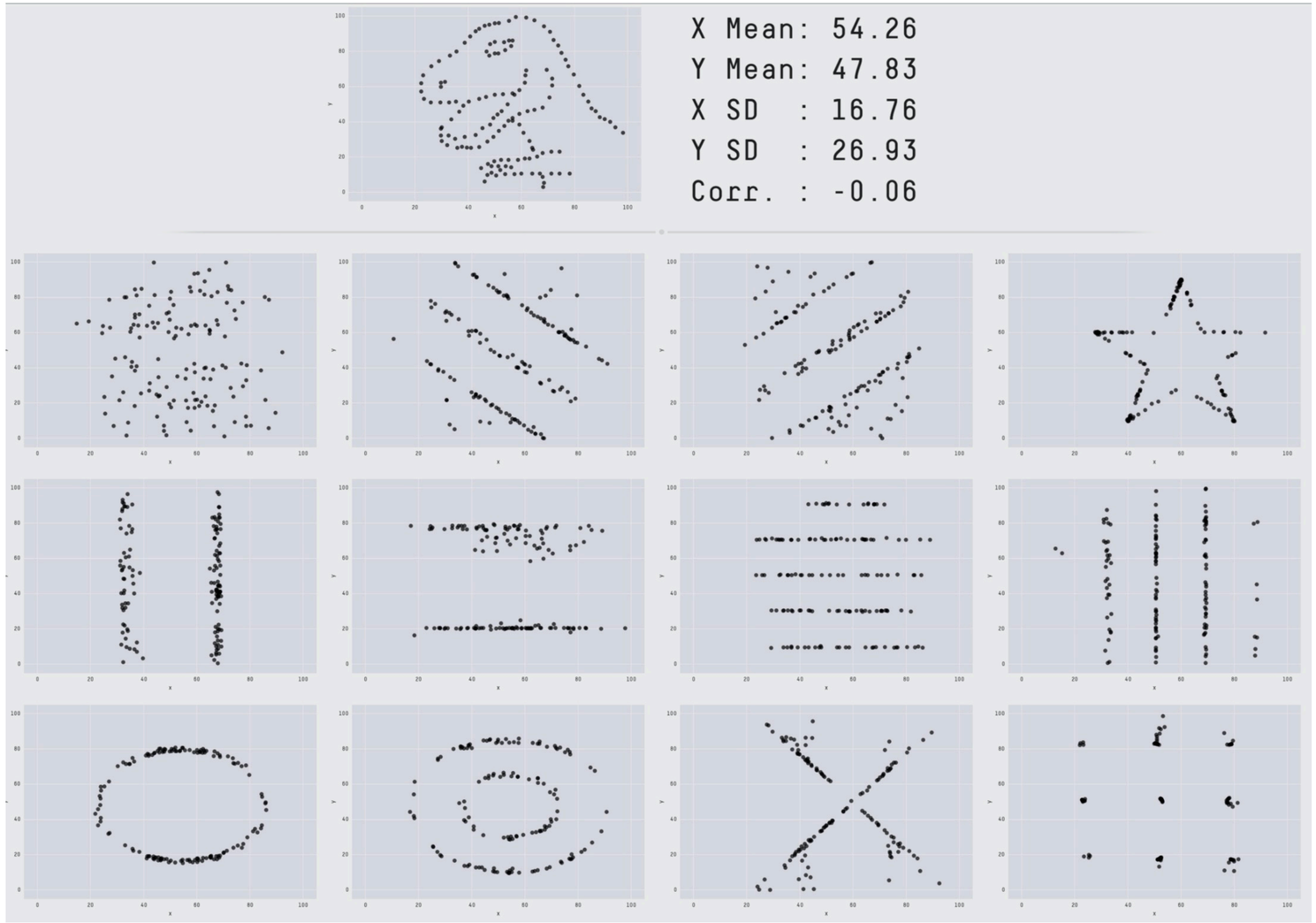



```
r = 0.0982029696446 ; p = 0.680415082529
```




```
r = 0.702033103659 ; p = 0.000559678419262
```

Visualize your data!



Source: Matejka & Fitzmaurice, *ACM SIGCHI* 2017

UC San Diego

Visualize your data!

Source: Matejka & Fitzmaurice, *ACM SIGCHI* 2017