Applications

Of Data Science

Network Models

Applications of Data Science - Class 11

Giora Simchoni

`gsimchoni@gmail.com and add #dsapps in subject`

Stat. and OR Department, TAU

2021-12-31

APPLICATIONS

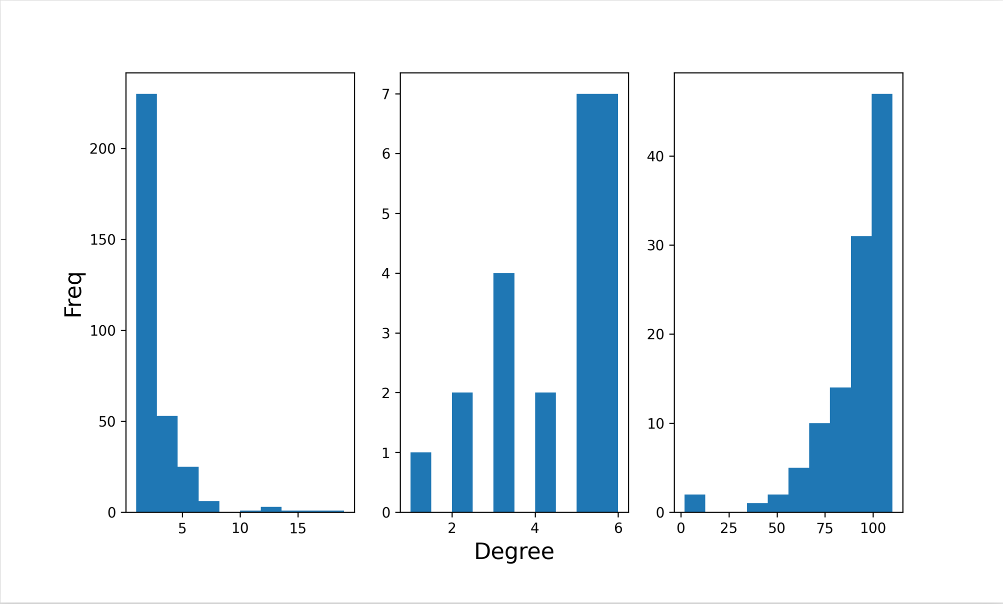
OF DATA SCIENCE

Motivation

APPLICATIONS
OF DATA SCIENCE

Degree histograms of 3 real networks:

- The cast of RuPaul's Drag Race Twitterverse
- The Israeli artists musical cooperations in the 21st century
- The Sci-Fi themed correlation network



Which is which?

Why model networks?

- Know your network better:
  - how it was formed
  - what class it belongs to (clustering)
  - how and why it deviates from model
- Predict behavior in network:
  - epidemic spread/resistance
  - search
  - link prediction
  - node disambiguation
- Generalization
- Simulations and the ability to estimate metrics on huge networks

The Erdős–Rényi model

(Random Graphs)

APPLICATIONS

OF DATA SCIENCE

You know the ER model

In a n nodes undirected network, suppose edges form identically and independently from each other.

Each possible edge is a Bernoulli trial with probability p.

Then the no. of edges M is a Binomial RV $\sim Binom(\binom{n}{2}, p)$

> 💡 So what is $P(M = m)$? What is $E(M)$? What is $Var(M)$?

The $G(n, p)$ collection of all simple networks formed this way is the Erdős–Rényi model.

Mean Degree

The mean number of edges is: $E(M) = \binom{n}{2}p$

We have defined the mean degree to be: $c = \frac{2m}{n}$, but now we treat m as a RV, so the mean degree would be:

$$c = E(D) = E(\frac{2M}{n}) = \frac{2E(M)}{n} = \frac{2}{n}\binom{n}{2}p = (n-1)p$$

> 💡 Say it in words, it makes sense.

But this implies we could have formulated the ER model slightly different:

Let D be the degree of a particular node in an undirected network with n nodes. Suppose each node "chooses" its neighbors with probability p, then: $D \sim Binom(n-1, p)$

Estimating n, p and c from our networks:

```
def estimate_p(m, n): return m / (n * (n - 1) / 2)
def estimate_c(n, p): return (n - 1) * p
def calculate_c(G, l=1): return np.mean(np.array(list(dict(G.degre

n_isr, n_sci, n_ru = I.number_of_nodes(), Sc.number_of_nodes(), Ru
m_isr, m_sci, m_ru = I.number_of_edges(), Sc.number_of_edges(), Ru
p_isr, p_sci, p_ru = estimate_p(m_isr, n_isr), estimate_p(m_sci, n
c_isr0, c_sci0, c_ru0 = estimate_c(n_isr, p_isr), estimate_c(n_sci
c_isr, c_sci, c_ru = calculate_c(I), calculate_c(Sc), calculate_c

pd.DataFrame({
  'network': ['Israeli Artists', 'Sci-Fi Books', 'RuPaul Verse'],
  'n': [n_isr, n_sci, n_ru],
  'm': [m_isr, m_sci, m_ru],
  'p': [p_isr, p_sci, p_ru],
  'c^': [c_isr0, c_sci0, c_ru0],
  'c': [c_isr, c_sci, c_ru]
})
```

```
##              network    n     m         p          c^          c
## 0  Israeli Artists  321   381  0.007418   2.373832   2.373832
## 1     Sci-Fi Books   23    51  0.201581   4.434783   4.434783
## 2     RuPaul Verse  112  5119  0.823520  91.410714  91.410714
```

Degree Distribution

And since we're dealing with large *sparse* networks, we might mention that as $n \to \infty$ and p is small, we expect the Binomial distribution to be approximated by the Poisson distribution, such that:

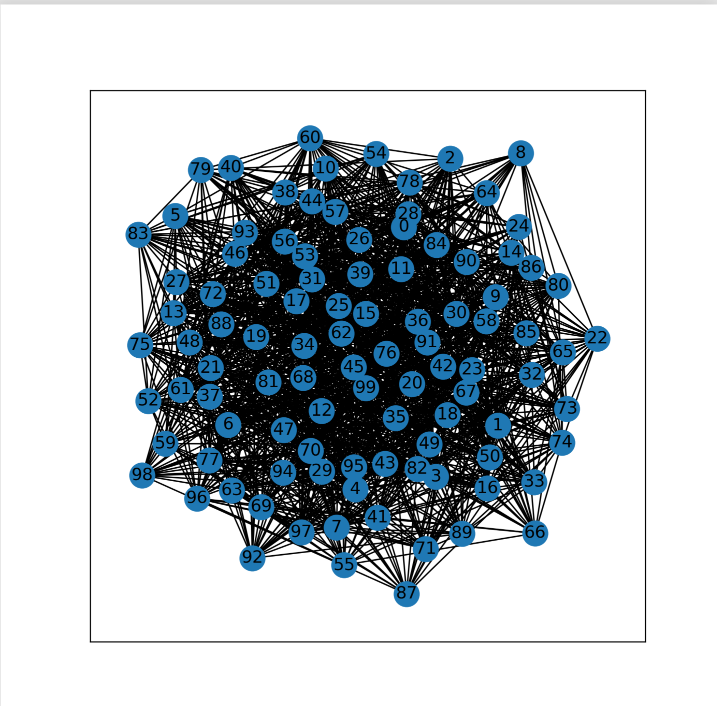
$D \sim Pois((n-1)p)$ or $D \sim Pois(c)$

Which is why the ER model is sometimes referred to as the Poisson random graph.

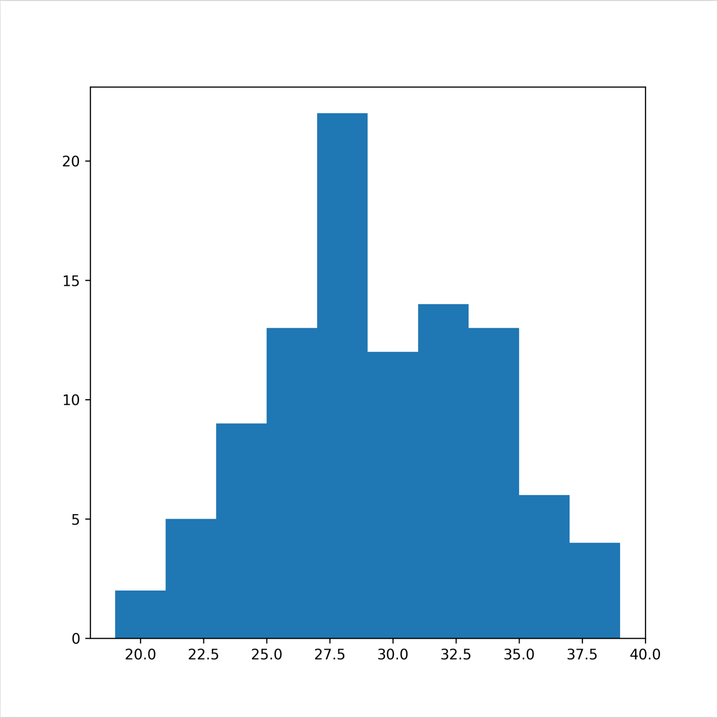
ER model In NetworkX

```
G = nx.erdos_renyi_graph(n = 100, p = 0.3)

plt.figure()
nx.draw_networkx(G)
plt.show()
```



```python
deg = [deg for node, deg in G.degree()]
h = plt.hist(deg)
plt.show()
```



💡 Does this degree distribution resemble any of the distributions we've seen?

Transitivity

$$Transitivity(G) = \frac{\#\text{closed triads}}{\#\text{triads}}$$

In other words it is the probability of two neighbors of the same node to also be connected.

> What is that probability in the ER model? Reasonable?

```
pd.DataFrame({'network': ['Israeli Artists', 'Sci-Fi Books', 'RuPa
  'n': [n_isr, n_sci, n_ru],
  'p': [p_isr, p_sci, p_ru],
  'transitivity': [nx.transitivity(I), nx.transitivity(Sc), nx.tra
```

```
##            network    n         p  transitivity
## 0  Israeli Artists  321  0.007418      0.019231
## 1     Sci-Fi Books   23  0.201581      0.522388
## 2    RuPaul Verse  112  0.823520      0.888871
```

Diameter

We have defined the diameter as the maximal shortest distance between any pair of nodes.

It turns out this has a nice expression in the ER model:

- Take a random node in a ER network, how many neighbors do you expect it to have?
- Take each one of these neighbors, and take each of *its* neighbors - what do you get?
- Do this l times, what do you get? What would you get "in the end"?

The average path length from a node to all other nodes is $\sim \frac{\ln(n)}{\ln(c)}$.

Similarly one can show the diameter is $\sim \frac{\ln(n)}{\ln(c)}$

```
d_isr = nx.diameter(I)
d_sci = nx.diameter(Sc)
d_ru = nx.diameter(Ru)

pd.DataFrame({
  'network': ['Israeli Artists', 'Sci-Fi Books', 'RuPaul Verse'],
  'n': [n_isr, n_sci, n_ru],
  'c': [c_isr, c_sci, c_ru],
  'ln(n)/ln(c)': [np.log(n_isr)/np.log(c_isr), np.log(n_sci)/np.lo
  'd': [d_isr, d_sci, d_ru]
})
```

```
##              network    n          c  ln(n)/ln(c)   d
## 0  Israeli Artists  321   2.373832     6.676003  17
## 1     Sci-Fi Books   23   4.434783     2.105095   5
## 2     RuPaul Verse  112  91.410714     1.044988   3
```

💡 What famous phenomenon this could help explain?

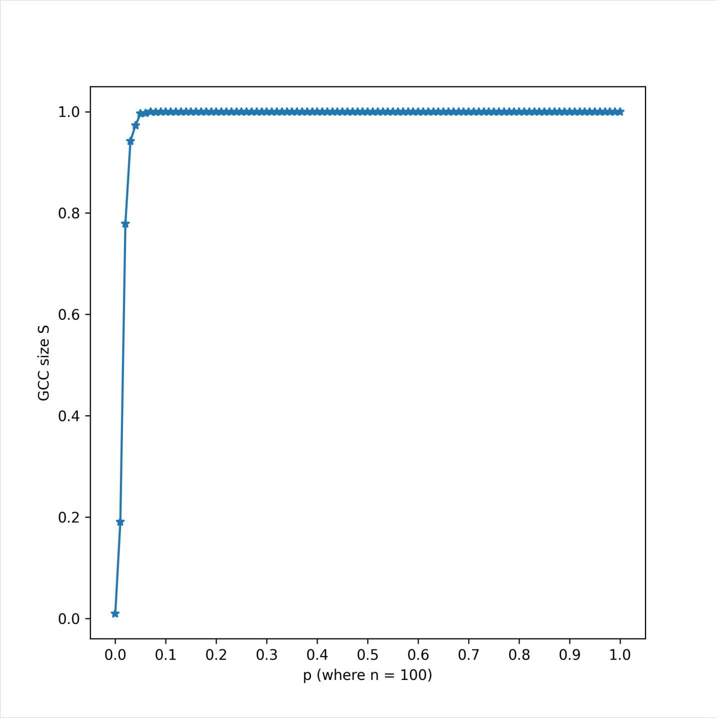
The Giant Component

What is the size of the largest component with $p = 0$?

What is the size of the largest component with $p = 1$?

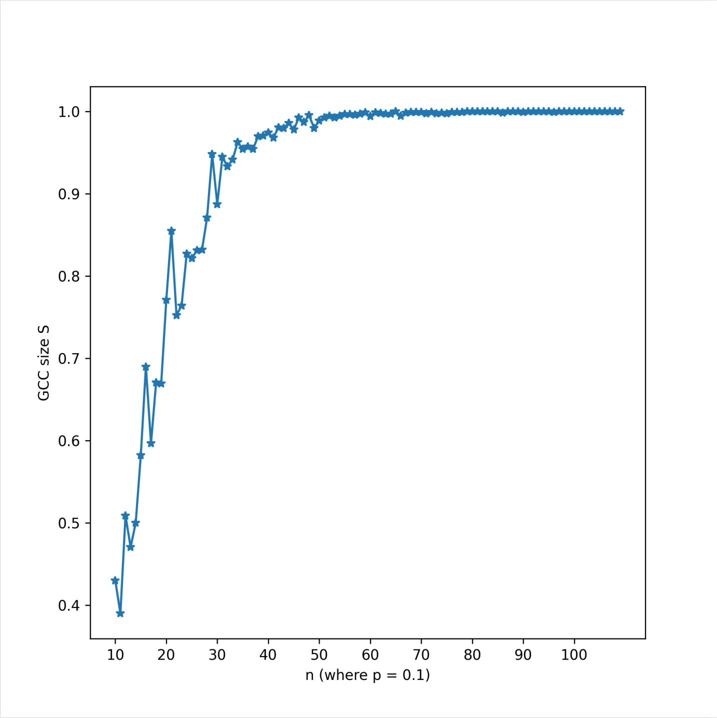
```
n = 100
s = []
for p in np.linspace(0, 1, 101):
  s_p = []
  for i in range(10):
    G = nx.erdos_renyi_graph(n = n, p = p)
    Gcc = sorted(nx.connected_components(G), key=len, reverse=True
    Gcc_n = G.subgraph(Gcc[0]).number_of_nodes()
    s_p.append(Gcc_n / n)
  s.append(np.mean(s_p))

plt.plot(s, marker='*')
locs, labels = plt.xticks(np.linspace(0, 100, 11), np.round(np.lin
plt.xlabel('p (where n = 100)')
plt.ylabel('GCC size S')
plt.show()
```



Surprisingly, for a given n, when p is varied from 0 to 1, the giant component size S suddenly "jumps"!

This is also true for a given p as you increase n: imagine sitting at the comfort of your home watching your ER network form, when suddenly...



Don't blink! Because there seems to be a *critical point*, a *transition point*, a *percolation point* where S increases from 0 and finally reaches to 1 making your network fully connected.

A slight wave of hands

S is the probability of a node being connected to the GCC.

$$S = P(\text{node i is connected to GCC}) =$$
$$= 1 - P(\text{node i is disconnected from GCC}) =$$
$$= 1 - P(\text{node i is disconnected from GCC via node j})^{n-1}$$

For a given i, j:

$$P(\text{node i is disconnected via node j}) =$$
$$= P(\text{node i is disconnected with j}) +$$
$$P(\text{node i is connected with j but j is disconnected from GCC}) =$$
$$= (1 - p) + p(1 - S) = 1 - pS$$

$S = 1 - (1 - pS)^{n-1}$

And you can immediately tell finding a closed solution for S would be hard.

Substituting $p = \frac{c}{n-1}$ where c is the mean degree:

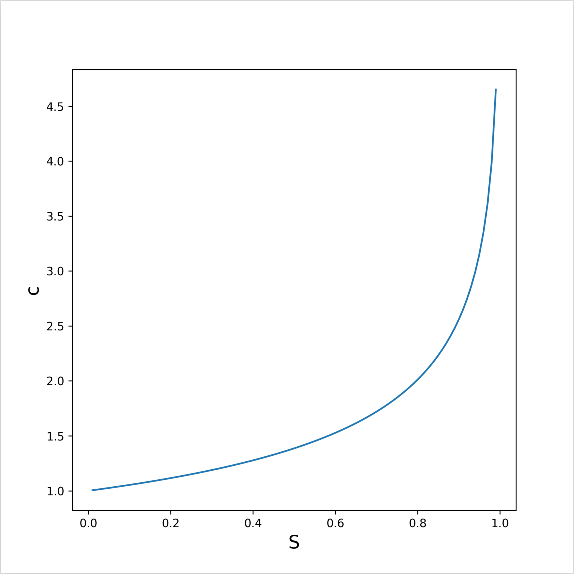
$S = 1 - (1 - \frac{c}{n-1} S)^{n-1}$

Recall that $e^x = \lim_{n \to \infty} (1 + \frac{x}{n})^n$ so in the limit of n:

$S = 1 - e^{-cS}$

Now there is no closed solution for S but we can isolate $c = (n-1)p$:

$c = -\frac{\ln(1-S)}{S}$



We can see that around $c = 1$ (or $p = 1/n$) the size of the GCC starts increasing (makes sense).

It can be shown that around $c = \ln(n)$ (or $p = \frac{\ln(n)}{n}$) we expect the network to be fully connected.

> 💡 So according to the ER model, for 1K nodes, a probability of 0.7% of an edge is enough to make the network connected. What do you think of that?

All of the 3 networks we've talked about are fully connected (though notice how the Israeli Artists and the Sci-Fi Books networks were created).

But the model gives us a way of describing at what *stage* they are and how *robust* they are to node failures:

```
pd.DataFrame({
  'network': ['Israeli Artists', 'Sci-Fi Books', 'RuPaul Verse'],
  'n': [n_isr, n_sci, n_ru],
  'c': [c_isr, c_sci, c_ru],
  'ln(n)': [np.log(n_isr), np.log(n_sci), np.log(n_ru)],
  'stage': ['supercritical', 'connected', 'connected']
})
```

```
##             network    n          c     ln(n)          stage
## 0   Israeli Artists  321   2.373832  5.771441  supercritical
## 1      Sci-Fi Books   23   4.434783  3.135494      connected
## 2      RuPaul Verse  112  91.410714  4.718499      connected
```

ER Model Summary

Models well:

- Diameter, average path length
- Giant Component, Percolation, Network Robustness

Does not model well:

- Degree distribution
- Transitivity (CC)
- Communities
- Homophily

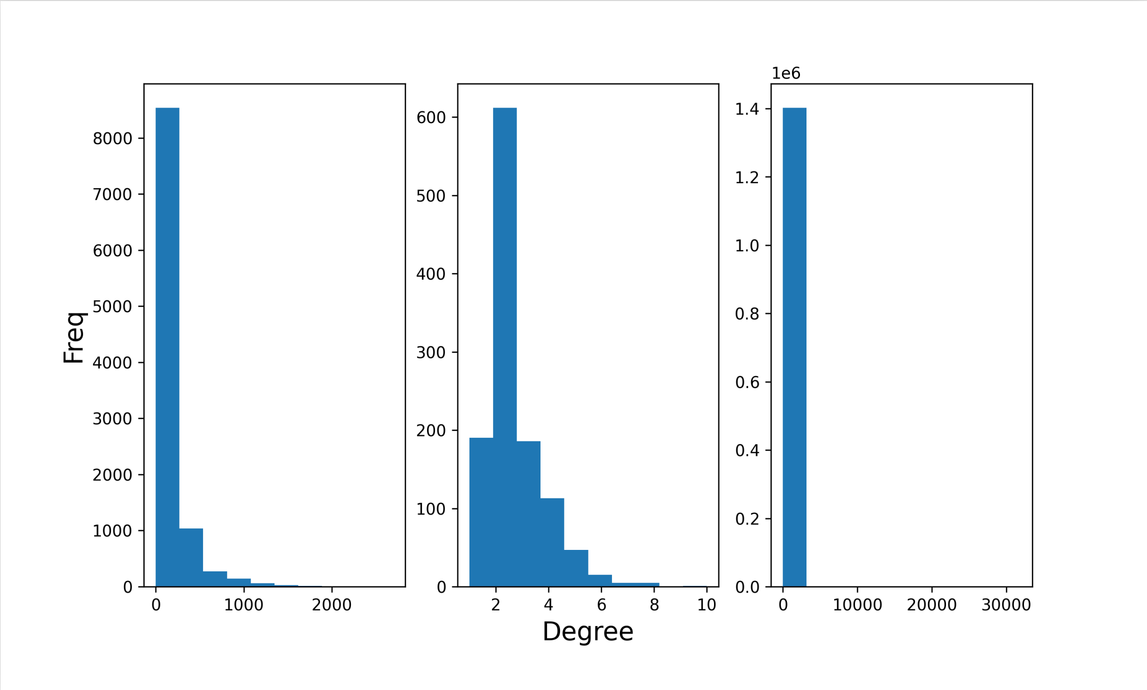
The Power-Law Distribution

Let's see some larger real-life networks:

- The clients of a high-end Brazilian escort service, connected by whether they hired the same escort
- European cities, connected by whether a E-Road connects them
- Hyves - A Dutch social network

```
##     network        n        m         p           c     S
## 0    Escort    10106   668183  0.013086  132.234910  0.96
## 1  EuroRoad     1174     1417  0.002058    2.413969  0.89
## 2     Hyves  1402673  2777419  0.000003    3.960180  1.00
```

Degree histograms



One of the main shortcomings of the ER model is that none of these look even remotely Poisson.

Power Law

If $X \sim PL(\alpha, x_{\min})$, then:

$f(x) = Cx^{-\alpha}$ for $x \geq x_{\min} > 0; \alpha > 1$

where $C = \frac{\alpha - 1}{x_{\min}^{1-\alpha}}$

and we can write: $f(x) = \frac{\alpha-1}{x_{\min}} (\frac{x}{x_{\min}})^{-\alpha}$

> What do you need to check in order to verify this is a proper density function?

The CDF of X:

$$F_X(x) = \int_{-\infty}^{x} f(x)dx = C\int_{x_{\min}}^{x} x^{-\alpha}dx = C \cdot \left[\frac{x^{1-\alpha}}{1-\alpha}\right]_{x_{\min}}^{x} = \frac{\alpha-1}{x_{\min}^{1-\alpha}}\left[\frac{x^{1-\alpha}}{1-\alpha} - \frac{x_{\min}^{1-\alpha}}{1-\alpha}\right] = 1 - \left(\frac{x}{x_{\min}}\right)^{1-\alpha}$$

The Expectation of X:

$$E(X) = \int_{-\infty}^{\infty} xf(x)dx = C\int_{x_{\min}}^{\infty} x^{1-\alpha}dx = \frac{\alpha-1}{x_{\min}^{1-\alpha}}\left[\frac{x^{2-\alpha}}{2-\alpha}\right]_{x_{\min}}^{\infty} = x_{\min}\left(\frac{\alpha-1}{\alpha-2}\right)$$

Where $E(X)$ converges only if $\alpha > 2$.

The MLE for α is: $\hat{\alpha} = 1 + \frac{n}{\sum_i \ln(\frac{x_i}{x_{\min}})}$

> 💡 The estimate for $x_{\min}$ is not simply $min(X)$! With empirical data it is best advised to first estimate $x_{\min}$, i.e. from *where* the distribution behaves like PL. See Clauset et. al. article and the powerlaw library.

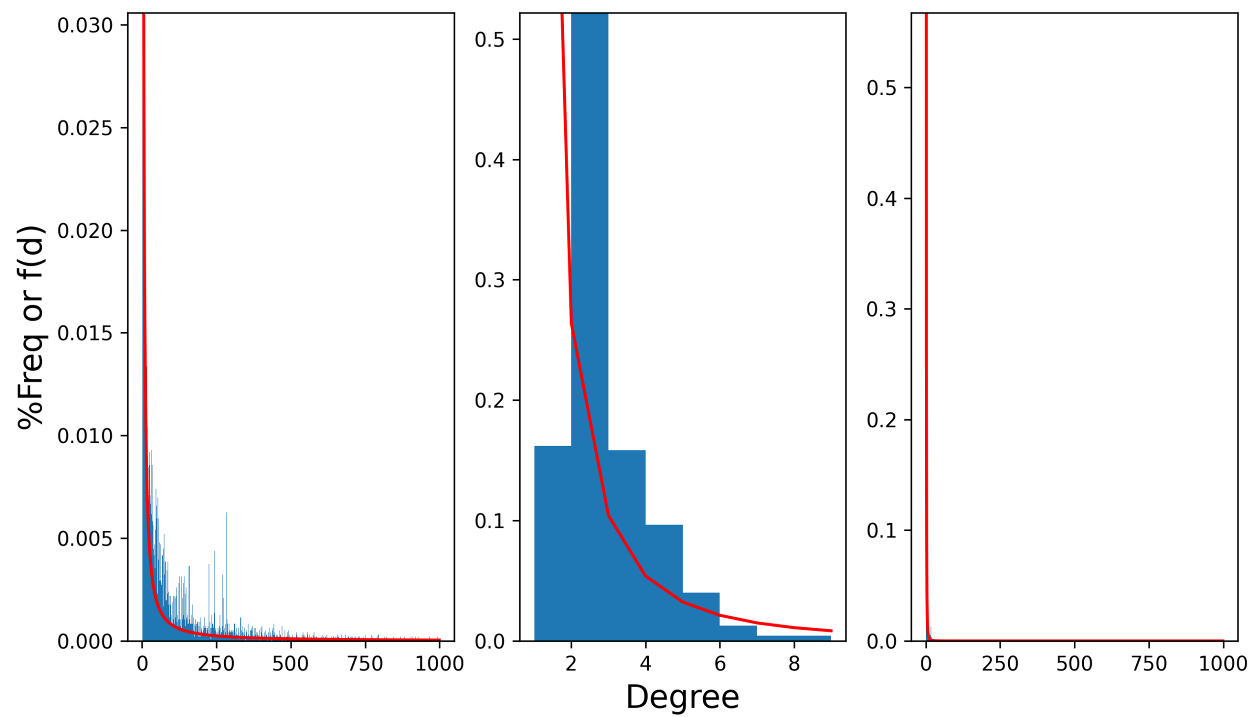
Fitting the PL to our networks

```
def fit_pl(G):
  degree_seq = np.array(list(dict(G.degree()).values()))
  n = G.number_of_nodes()
  # estimating d_min as min(d) is NOT right!
  d_min = np.min(degree_seq)
  if d_min == 0:
    d_min = 1
    degree_seq = degree_seq[degree_seq >= 1]
    n = len(degree_seq)
  alpha = 1 + n/np.sum(np.log(degree_seq/d_min))
  davg = np.mean(degree_seq)
  Ed = d_min * (alpha - 1)/(alpha - 2) if alpha > 2 else np.inf
  dvar = np.var(degree_seq)
  Ed2 = d_min**2 * (alpha - 1)/(alpha - 3) if alpha > 3 else np.ir
  Vd = Ed2 - (Ed)**2 if alpha > 3 else np.inf
  return d_min, alpha, davg, Ed, dvar, Vd
```

```python
dmin_e, alpha_e, davg_e, Ed_e, dvar_e, Vd_e = fit_pl(E)
dmin_eu, alpha_eu, davg_eu, Ed_eu, dvar_eu, Vd_eu = fit_pl(Euro)
dmin_h, alpha_h, davg_h, Ed_h, dvar_h, Vd_h = fit_pl(H)

pd.DataFrame({
  'network': ['Escort', 'EuroRoad', 'Hyves'],
  'dmin': [dmin_e, dmin_eu, dmin_h],
  'alpha': [alpha_e, alpha_eu, alpha_h],
  'Avg(d)': [davg_e, davg_eu, davg_h],
  'E(d)': [Ed_e, Ed_eu, Ed_h],
  'Var(d)': [dvar_e, dvar_eu, dvar_h],
  'V(d)': [Vd_e, Vd_eu, Vd_h],
})
```

```
##     network  dmin     alpha       Avg(d)      E(d)        Var(d)  V(d)
## 0    Escort     1  1.262319  137.457930       inf  49816.740516   inf
## 1  EuroRoad     1  2.289899    2.413969  4.449473      1.412956   inf
## 2     Hyves     1  2.658003    3.960180  2.519750   2051.663316   inf
```


%Freq or f(d)
0.030
0.025
0.020
0.015
0.010
0.005
0.000
0
250
500
750
1000
0.5
0.4
0.3
0.2
0.1
0.0
2
4
6
8
Degree

Why do we like the Power Law distribution so much?

- scale invariance
- heavy tail
- the 80/20 (Pareto) rule

Scale Invariance

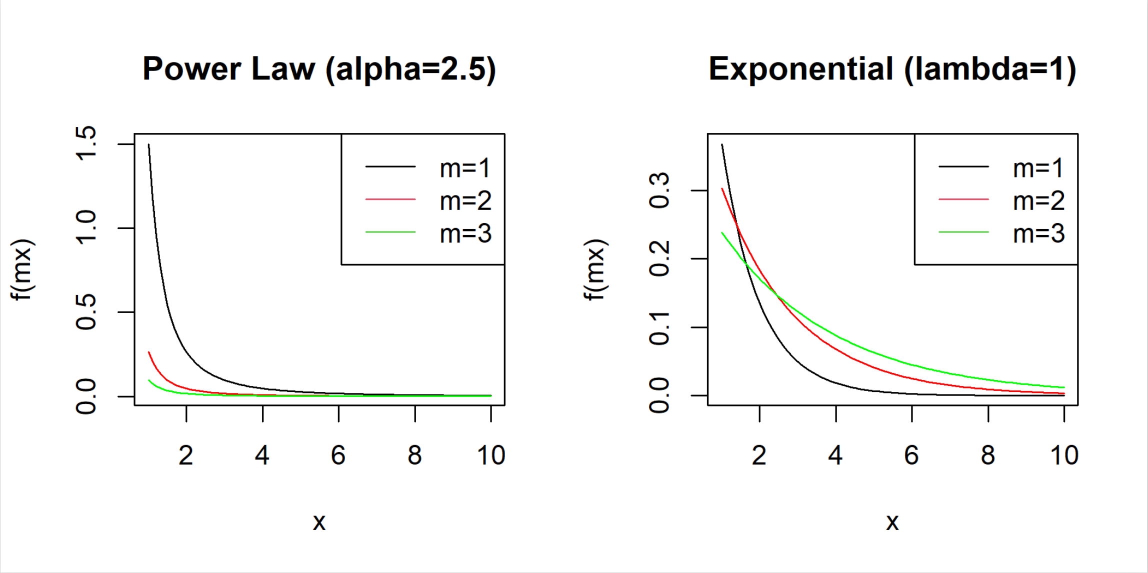
If we multiply X by constant m the density "scales":

$$f(mx) = C(mx)^{-\alpha} = Cm^{-\alpha}x^{-\alpha} = C'x^{-\alpha} \propto f(x)$$

Thus, mX will distribute Power-Law with the same α parameter and a different constant.

> 💡 How would mX distribute if X distributed Normal? Exponential?

Meaning, the relative likelihood between small and large events is the same, no matter what choice of "small" we make:

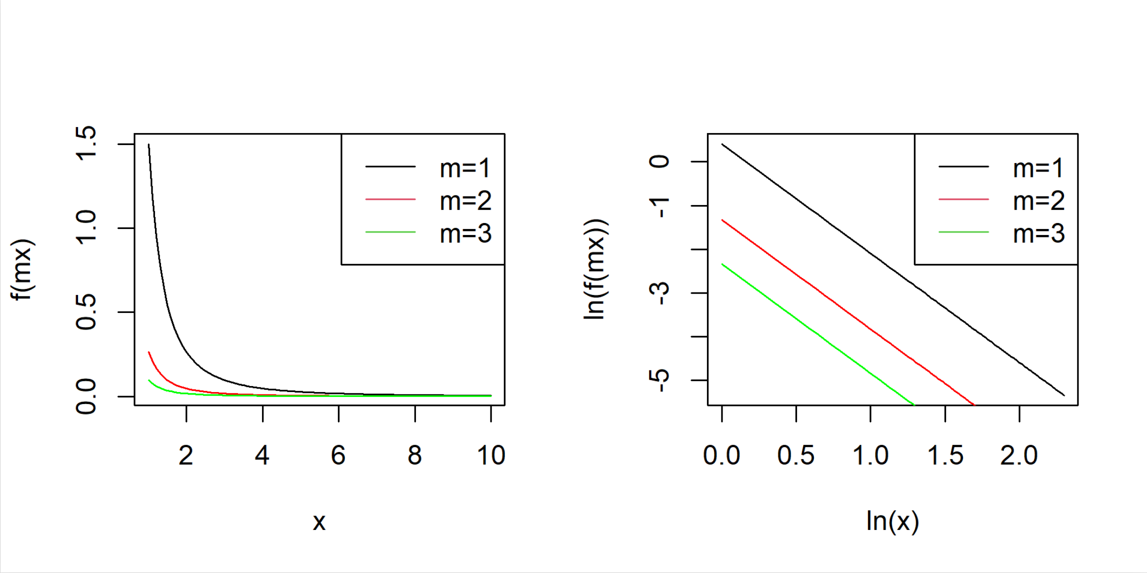


Another way of seeing this:

If we take the log transformation of the Power-Law distribution we get:

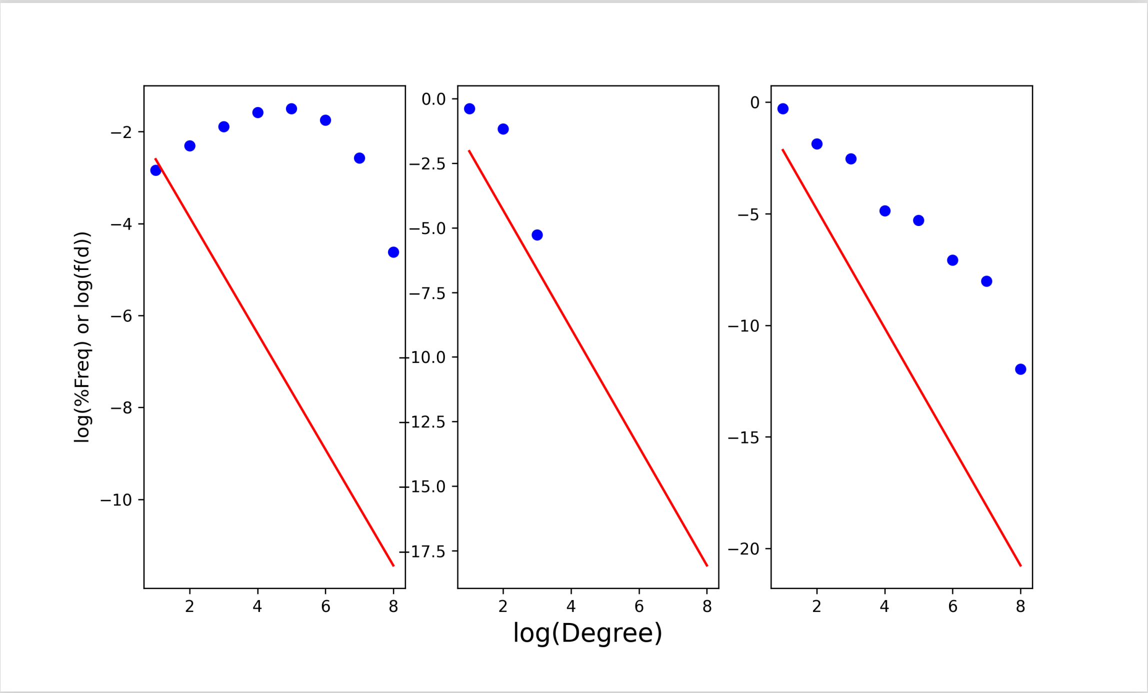
$$\ln(f(x)) = \ln(Cx^{-\alpha}) = \ln(C) - \alpha \ln(x)$$

Thus the log-log plot of Power-Law should show a straight line with slope $-\alpha$, and multiplying by m only changes the intercept.



Seeing the log-log transformation with our distributions should also result in a straight line with a slope roughly $-\alpha$:

```
## <string>:7: RuntimeWarning: divide by zero encountered in log
```



Do not estimate α from log-log plots

This does not look "good" for a few reasons:

- n, choice of log
- We did not estimate $x_{\min}$ correctly (and therefor α), using simply the minimum degree
- log-log plots, even when done right, are misleading, do not extrapolate α out of them
- Maybe the PL fit is useful but just not right

Heavy Tail

The k-th moment of the PL distribution converges only for $k < \lfloor \alpha - 1 \rfloor$:

$$E(X^k) = \int_{x_{\min}}^{\infty} x^k f(x) dx = \frac{\alpha-1}{x_{\min}^{1-\alpha}} \int_{x_{\min}}^{\infty} x^{k-\alpha} dx = x_{\min}^k \left(\frac{\alpha-1}{\alpha-1-k}\right)$$

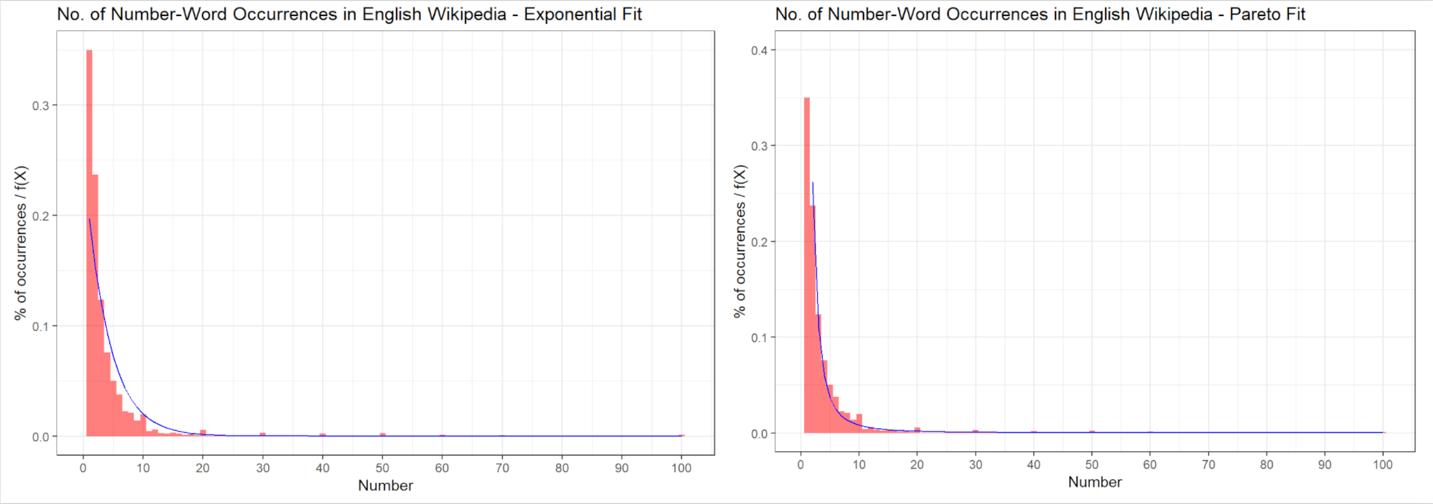
- For $1 < \alpha \leq 2$ the mean is infinite (and all higher moments)
- For $2 < \alpha \leq 3$ the mean is finite but the variance is infinite

As a result, the PL distribution is very good for modeling extreme distributions, where we *expect* extreme values, as opposed to e.g. the Normal distribution.

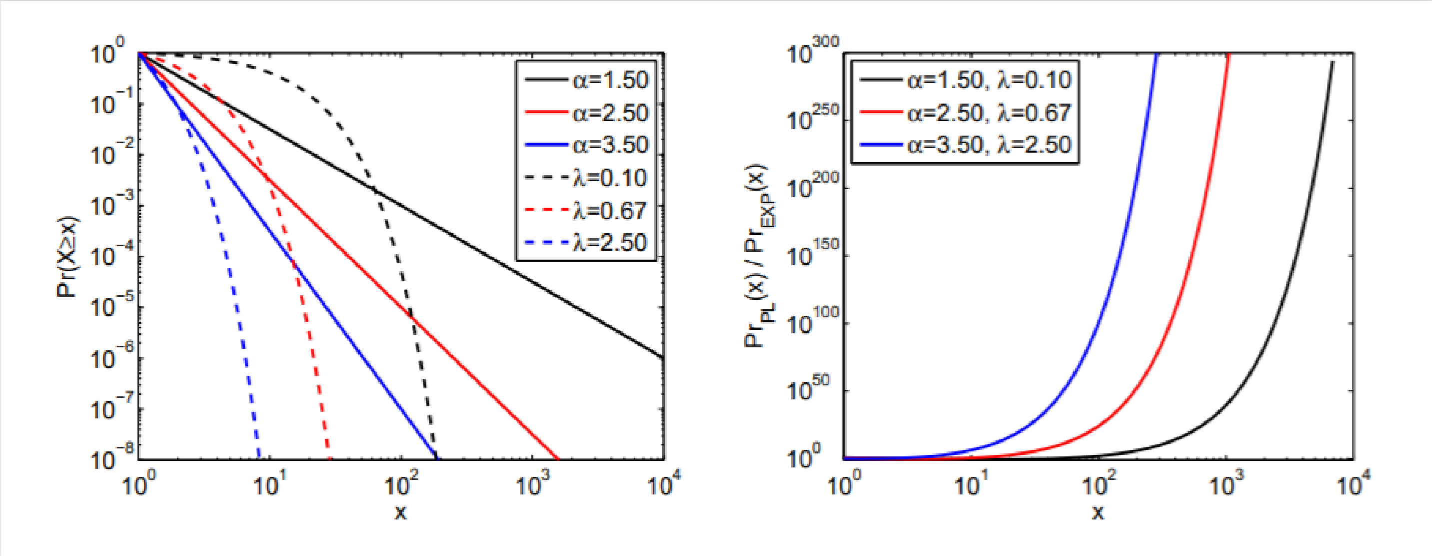
Also, if we sample from a typical PL distribution with $2 < \alpha < 3$ the value could be "anywhere" and isn't confined by a scale parameter

> When sampling from a Normal distribution, we can expect in 95% of cases...

For example, here is a post of mine, showing how the PL distribution is much more suitable for modeling how words are distributed in a large corpus of text such as Wikipedia, in comparison with the Exponential distribution, both fitted via MLE:



Another nice way of showing this is comparing $1 - F_X(x)$ under the PL distribution vs. $1 - F_X(x)$ under the Exponential distribution for suitable values of α and λ:



80/20 (Pareto) Rule

The Pareto rule says that "80% of the wealth is in the hands of the richest 20% of people".

But the Pareto distribution is just a different formulation of the Power Law distribution, and this phenomenon is indeed characteristic to the PL distribution.

If X is "wealth" and we want to know what is the probability of being "wealthy", i.e. the part *of population* which holds at least m wealth:

$$P(X) = P(X > m) = 1 - F_X(m) = (\frac{m}{x_{\min}})^{1-\alpha}$$

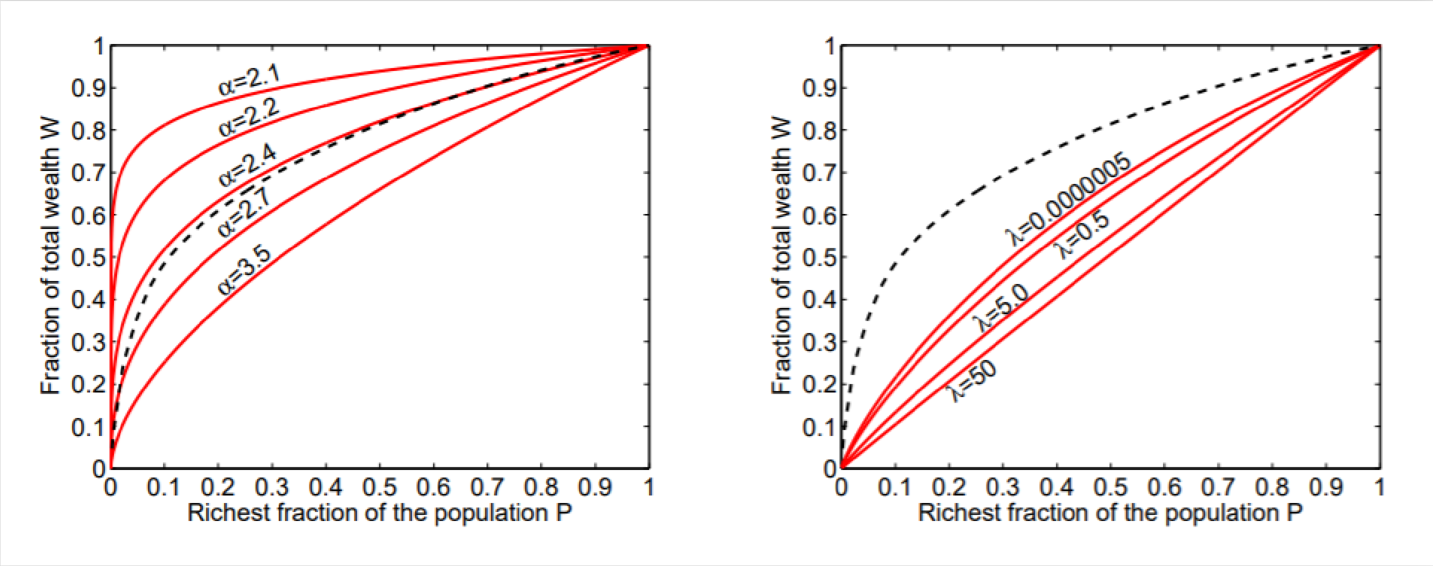
The part *of wealth* that those "wealthy" have is the division of expectations:

$$W(X) = \frac{E(X>m)}{E(X)} = \frac{\int_m^\infty xCx^{-\alpha}dx}{x_{\min}(\frac{\alpha-1}{\alpha-2})} = (\frac{m}{x_{\min}})^{2-\alpha}$$

So, being wealthy isn't dependent on m, only on α:

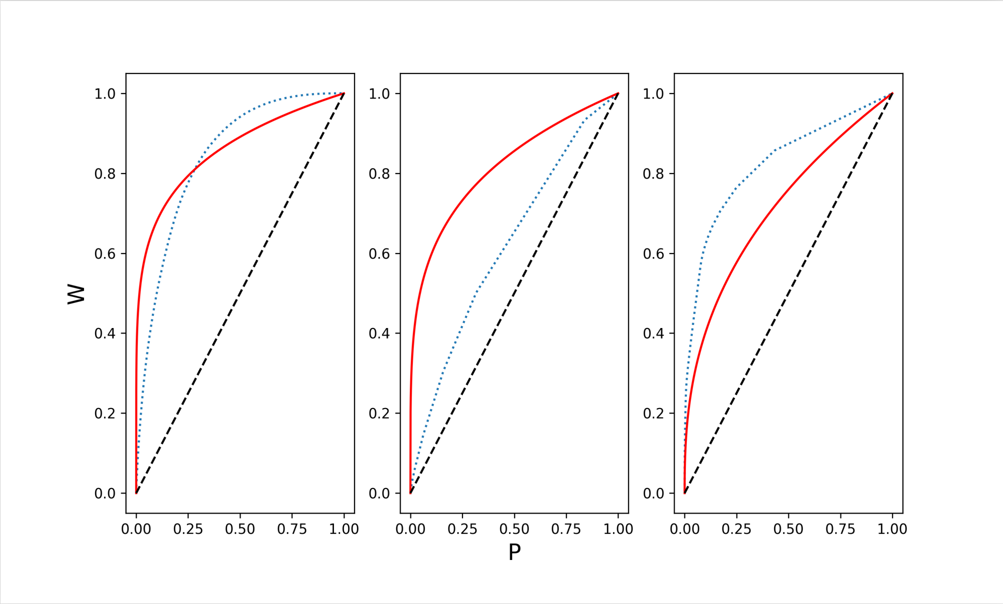
$$W(X) = P(X)^{\frac{\alpha-2}{\alpha-1}}$$

And this is how we get to the 80/20 rule without defining what wealthy is, the Lorenz curve:



💡 The dashed line shows the empirical Lorenz curve for the wealthiest individuals in the USA (data from the Forbes, 2003)!

```python
def lorenz_curve(G, alpha):
  deg = [deg for node, deg in G.degree()]
  deg_sorted = np.array(sorted(deg, reverse=True))
  W = deg_sorted.cumsum() / deg_sorted.sum()
  P = np.linspace(0.0, 1.0, W.size)
  alpha = alpha if alpha > 2 else 2.2
  WE = P**((alpha-2)/(alpha-1))
  return P, W, WE
P_e, W_e, WE_e = lorenz_curve(E, alpha_e)
P_eu, W_eu, WE_eu = lorenz_curve(Euro, alpha_eu)
P_h, W_h, WE_h = lorenz_curve(H, alpha_h)
```



Nice, but.

So we agree many large networks degree distributions behave Power-Law.

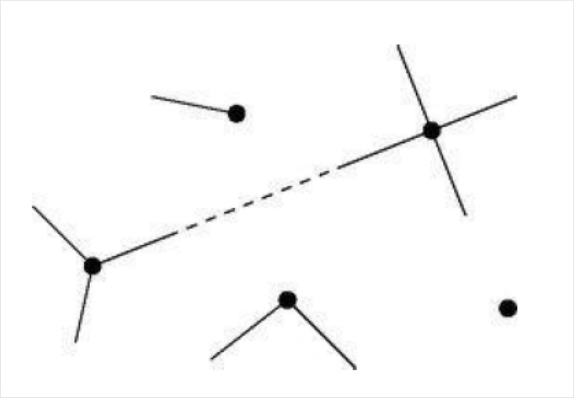
- That is not a model of a how a network was formed, what process causes "scale-free" networks?
- How do all *other* network metrics behave? Do we see improvement where the ER model did not make sense?

The Configuration Model

Applications
Of Data Science

Scenario 1

- We have a sequence of each node's degree: $k_1, \ldots, k_n$
- Each node is given a number of "stubs" of edges equal to its required degree.
- Choose a random pair of stubs, connect.
- From the stubs left, choose another random pair, connect.
- Repeat until all stubs are left.



Scenario 2:

- We have a degree *distribution*: $k_1 : p_{k_1}, \ldots, k_m : p_{k_m}$ where $\sum_{i=1}^{m} p_{k_i} = 1$
- We sample n degrees from this distribution and continue as usual.

> What if p_{k_i} is Poisson? What if it is Power-Law?

Finally notice that as with the ER model, we get a *distribution* of networks from every sequence or degree-distribution specified. And this distribution of networks is Uniform!

Two issues

> 💡 Try building a network with degree sequence $\{1, 2, 2\}$

1. If m is no. of edges $\sum_i k_i = 2m$ --> the sum of degrees must be even! Even those sampled from p_{k_i}

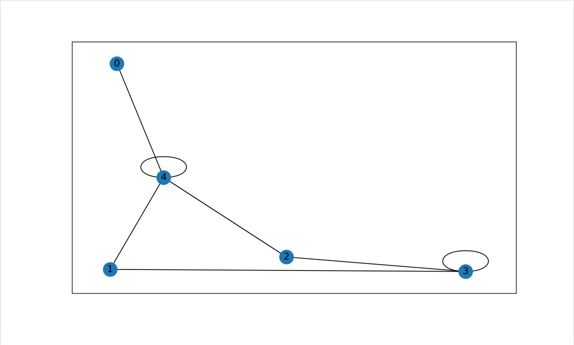
2. Nothing preventing connecting two of node i's stubs together
   Nothing preventing connecting nodes i and j more than once -->
   Self-edges and multi-edges are possible! (But with large n chances are small)

Configuration model In NetworkX

```
G = nx.configuration_model([1,2,3,4,6])
print(G.degree())
```

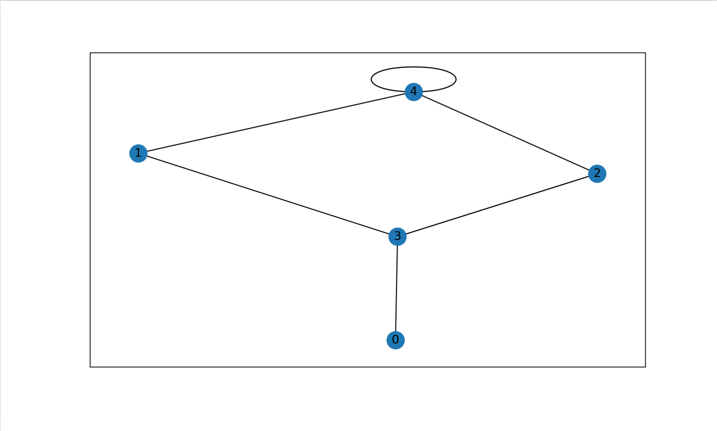
```
## [(0, 1), (1, 2), (2, 3), (3, 4), (4, 6)]
```

```
nx.draw_networkx(G)
plt.show()
```



Unfortunately matplotlib cannot draw self edges easily 🤬

```
G = nx.configuration_model([1,2,3,4,6])
nx.draw_networkx(G)
plt.show()
```



Each such graph is a sample from the same sequence/distribution of degrees.

Transitivity

It can be shown that the Clustering Coefficient can be defined via the degree distribution moments:

$$Transitivity(G_{Conf}) = \frac{1}{n}\frac{[E(D^2)-E(D)]^2}{E(D)^3}$$

Which can be estimated by: $\frac{1}{n}\frac{[\overline{D^2}-\overline{D}]^2}{\overline{D}^3}$

In the case $D_G \sim Pois(\lambda)$:

$$Transitivity(G_{Conf}) = \frac{1}{n}\frac{[\lambda+\lambda^2-\lambda]^2}{\lambda^3} = \frac{\lambda}{n}$$

But $\lambda \approx np$ under the ER model, so we get the original estimate: $Transitivity(G_{Conf}) = p$

In the case $D_G \sim PL(\alpha, d_{\min})$, notice this estimate will only be valid if $\alpha > 3$, otherwise the second moment does not converge:

$$Transitivity(G_{Conf}) = \frac{1}{nx_{\min}}[\frac{\alpha-2}{\alpha-1}](x_{\min}[\frac{\alpha-2}{\alpha-3}] - 1)^2$$

So we now have a way of seeing the actual Transitivity of our networks, vs. a theoretical result of two models:

```
def transitivity_conf(G):
  D = calculate_c(G, l=1)
  D2 = calculate_c(G, l=2)
  n = G.number_of_nodes()
  return (1/n)*((D2-D)**2/(D**3))

def transitivity_pl_theoretical(n, alpha, dmin):
  if dmin > 0 and alpha > 3:
    return (1/(alpha*dmin)) * ((alpha-2)/(alpha-1)) * (dmin*(alpha
  else:
    return np.nan
```

```python
tc_e = transitivity_conf(E); tpl_e = transitivity_pl_theoretical(r
tc_eu = transitivity_conf(Euro); tpl_eu = transitivity_pl_theoreti
tc_h = transitivity_conf(H); tpl_h = transitivity_pl_theoretical(r

pd.DataFrame({
  'network': ['Escort', 'EuroRoad', 'Hyves'],
  'Trans_actual': [0.377628, nx.transitivity(Euro), 0.001559],
  'Trans_ER': [p_e, p_eu, p_h],
  'Trans_conf': [tc_e, tc_eu, tc_h],
  'Trans_pl_theoretical': [tpl_e, tpl_eu, tpl_h]})
```

```
##     network  Trans_actual  Trans_ER  Trans_conf  Trans_pl_theoretical
## 0    Escort      0.377628  0.013086    0.186232                   NaN
## 1  EuroRoad      0.033886  0.002058    0.001410                   NaN
## 2     Hyves      0.001559  0.000003    0.048872                   NaN
```

- The Configuration model is (maybe) slightly better than ER in estimating Transitivity
- For large networks I input the actual Transitivity (too long! that's why we need an estimator)
- But it seems that with these types of Random Graphs models, estimating Transitivity is hard

Diameter

It can be shown that the maximal shortest distance between any pair of nodes in the Configuration model is:

$$Diameter(G_{Conf}) = \frac{\ln(n)}{\ln(\frac{E(D^2)-E(D)}{E(D)})}$$

Which can be estimated by: $\frac{\ln(n)}{\ln(\frac{\overline{D^2}-\overline{D}}{\overline{D}})}$

In the case $D_G \sim Pois(\lambda)$:

$$Diameter(G_{Conf}) = \frac{\ln(n)}{\ln(\frac{\lambda+\lambda^2-\lambda}{\lambda})} = \frac{\ln(n)}{\ln(\lambda)}$$

But $\lambda \approx np \approx c$ under the ER model, so we get the original number:

$$Diameter(G_{Conf}) = \frac{\ln(n)}{\ln(c)}$$

In the case $D_G \sim PL(\alpha, d_{\min})$, again this estimate is only valid if $\alpha > 3$, otherwise the second moment does not converge:

$$Diameter(G_{Conf}) = \frac{\ln(n)}{\ln(x_{\min}[\frac{\alpha-2}{\alpha-3}]-1)}$$

```
def diameter_conf(G):
  D = calculate_c(G, l=1)
  D2 = calculate_c(G, l=2)
  n = G.number_of_nodes()
  return np.log(n) / np.log((D2-D)/D)

def diameter_pl_theoretical(n, alpha, dmin):
  if dmin > 0 and alpha > 3:
    return np.log(n) / np.log(dmin * (alpha - 2)/(alpha - 3) - 1)
  else:
    return np.nan
```

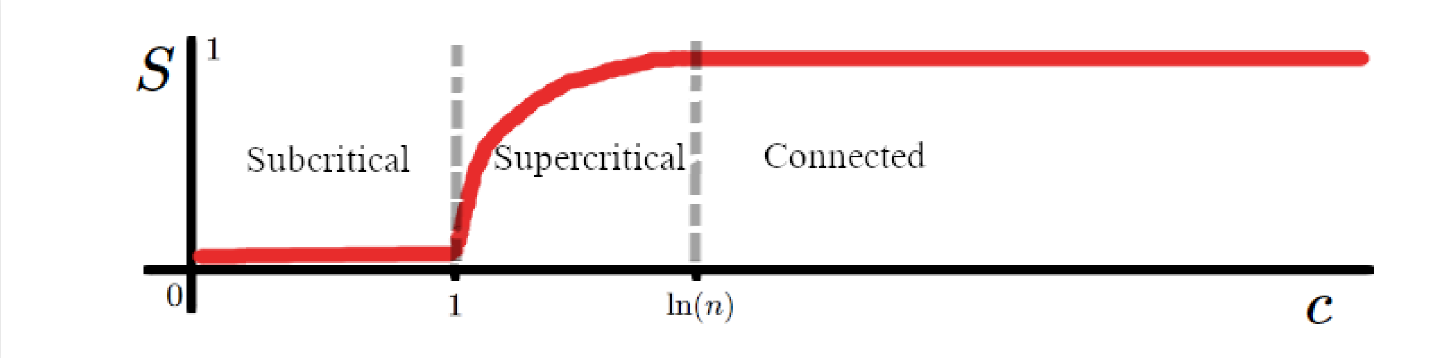
```
dc_e = diameter_conf(E); dpl_e = diameter_pl_theoretical(n_e, alph
dc_eu = diameter_conf(Euro); dpl_eu = diameter_pl_theoretical(n_eu
dc_h = diameter_conf(H); dpl_h = diameter_pl_theoretical(n_h, alph

pd.DataFrame({
  'network': ['Escort', 'EuroRoad', 'Hyves'],
  'Diameter_actual': ['8*', '62*', 10],
  'Diameter_ER': [np.log(n_e)/np.log(c_e), np.log(n_eu)/np.log(c_e
  'Diameter_conf': [dc_e, dc_eu, dc_h],
})
```

```
##      network Diameter_actual  Diameter_ER  Diameter_conf
## 0    Escort               8*     1.887754       1.484283
## 1  EuroRoad              62*     8.020417      10.202411
## 2     Hyves               10    10.284093       2.262518
```

The Giant Component

In the ER model we found:



💡 What is the maximum value c can reach?

In the Configuration model we find a similar pattern. It can be shown that once $E(D^2) > 2E(D)$ (estimated by $\overline{D^2} > 2\overline{D}$) the giant component starts forming, until the network is fully connected.

In the case $D_G \sim Pois(\lambda)$ this means $\lambda > 1$ but again we note $\lambda = c$ and this is the original ER critical point.

In the case $D_G \sim PL(\alpha, d_{\min})$ it can be shown the critical point is determined by α and $d_{\min}$. But for a pure PL distribution one can show:

- for $\alpha > 3.478$ we would expect a subcritical phase and no giant component
- once $\alpha < 3.478$ (critical point) the giant component starts to emerge
- for $2 < \alpha \leq 3$ the second moment is infinite, the first moment is finite and so the $E(D^2) > 2E(D)$ definitely holds an we are still in the supercritical phase
- for $\alpha < 2$ the network is fully connected

Configuration Model Summary

Models well:

- Diameter, average path length
- Giant Component, Percolation, Network Robustness
- Degree Distribution

Does not model well:

- Transitivity (CC)
- Communities
- Homophily

Why model networks?

- Know your network better:
  - how it was formed
  - what class it belongs to (clustering)
  - how and why it deviates from model
- Predict behavior in network:
  - epidemic spread/resistance
  - search
  - link prediction
  - node disambiguation
- Generalization
- Simulations and the ability to estimate metrics on huge networks