Lecture 4: Distributions of degree, triangles, and clustering.

Eduardo López

Continuing with our study of large scale structural properties of network, we focus on degree, triangle, and clustering distributions of networks. These provide concrete settings in which to learn how network properties work together or separately, highlighting differences between intrinsic and derivative properties networks. We expand our toolkit for using data.

1 Degree distribution of a network: a more detailed look

When looking at a real-world network, its degree distribution is one of the first properties that one should study. This is because the distribution is typically relevant to other aspects of the network's overall features. One particular quantity, although certainly not the only one, that is consistently associated with a network's degree distribution is the average shortest path length of the network, establishing a direct relation between the degree distribution and the "world-size" of the network (large, small, ultra-small).

Let us first discuss a very simple primer on what one can usually see when studying degree distributions, specially in the social context.

1.1 Dealing with data and making distributions

In order to make concrete the statements about the relations between $\Pr(k)$ and other network properties such as $\langle s \rangle$, we need to present some example networks and their respective degree distributions. The distributions are constructed as explained previously, first by determining $H(k)$ for all the

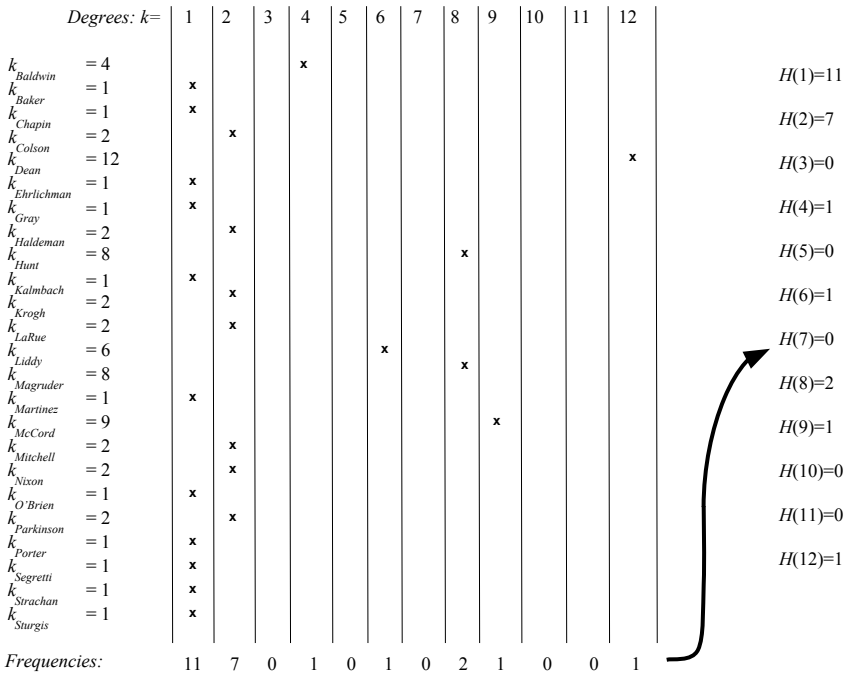

Figure 1: A schematic of how a degree histogram can be constructed manually. This example is for the Watergate example.

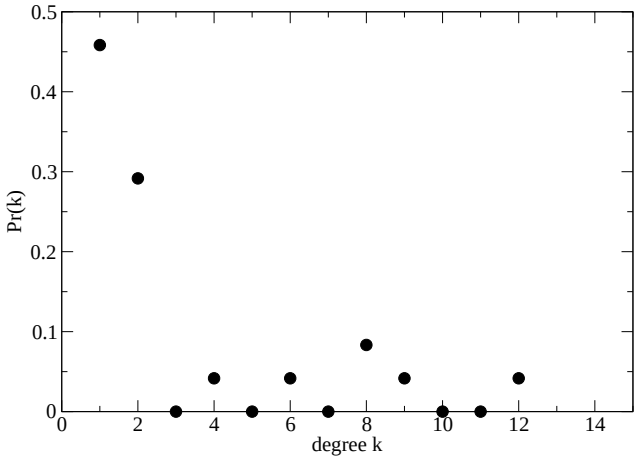

Figure 2: $\Pr(k)$ versus k for the Watergate network.

nodes and subsequently dividing these by n, which means

$$\Pr(k) = \frac{H(k)}{n} = \frac{H(k)}{\sum_{k=0}^{n-1} H(k)} \tag{1}$$

A simple diagram of the process of collecting $H(k)$ applied to the undirected version of the Watergate testimony network can be found in Fig. 1. The result of this exercise is the distribution shown in Fig. 2. The process produces the following values:

$$\begin{array}{llll} \Pr(k=1) = \frac{11}{24} & \Pr(k=2) = \frac{7}{24} & \Pr(k=4) = \frac{1}{24} & \Pr(k=6) = \frac{1}{24} \\ \Pr(k=8) = \frac{2}{24} & \Pr(k=9) = \frac{1}{24} & \Pr(k=12) = \frac{1}{24} & \Pr(k) = 0 \text{ for other } k. \end{array}$$

The analysis of a network of this size tends to be informative at a basic level. This means that in the distribution, we find some information that we are generally interested in. For instance, let us ask the following questions:

1. What is the smallest degree of the network $k_{\min}$?
2. What is the largest degree of the network $k_{\max}$?
3. What is the most probable degree value k^*?

These questions can be readily answered by inspecting the plot of the distribution (or alternatively of the histogram). In this case,

$$k_{\min} = 1, \quad k_{\max} = 12, \quad k^* = 1 \quad \text{[Watergate]}.$$

What happens when a network is large? In this case, the same procedure (conceptually speaking) leads to distributions that show interesting large scale patterns. For instance, consider the network for the email communication of the company Enron. This network is characterized by $n = 36,692$ and $m = 183,831$. In order to determine its degree distribution, we need to use code because manually studying it is prohibitively time-consuming. This can be done using the customary facilities of `python` and `networkx`, for instance.

1.2 Reading data and making distributions

To start the process of understanding how the data is read and then analyzed and converted to code, we start with this bit of code.

```
>>> import networkx as nx
>>> infile=open('email-Enron.txt','r') # Import Enron data
>>> lines=[] # Since we ignore data format, store it first.
>>> for line in infile:
...     lines.append(line)
...
>>> infile.close()
>>> lines[:5]
['# Directed graph (each unordered pair of nodes is saved once)
   : Email-Enron.txt \r\n',
 '# Enron email network (edge indicated that email was
   exchanged, undirected edges)\r\n',
 '# Nodes: 36692 Edges: 367662\r\n',
 '# FromNodeId\tToNodeId\r\n',
 '0\t1\r\n']
```

A bit of experience in knowing what we are reading will help a great deal as we progress.

The first thing to notice is the way to open a file in `python`. We make use of the command `open` which then accesses a file in read mode (`'r'`) called `'email-Enron.txt'`. Following this, since we cannot tell the format of the file before opening it, we create a `python` list object called *lines* in which we store all the lines of the file. This way, we can visually inspect the content and format of the lines of the file and make a determination about what we

need to do to construct a network. To make `python` sequentially read the content of `'email-Enron.txt'` under the assumption that it is made up of lines of information, the code uses some powerful facilities. In particular, `python` knows that if a file contains line breaks, it should assume there is an interesting element of information between line breaks. Thus, the line of code '`for line in infile:`' takes a line of the file at a time and *appends* it to the list `lines`.

To see the first five lines of the `python` list `lines`, we use the command `lines[:5]`. We find that the first 4 lines contain comments, as made clear by their written content as well as the fact that they begin with a character `#`. The fifth line seems to have interesting information. It reads `'0\t1\r\n'`. This line in fact contains two nodes `0` and `1` that are separated by a `'\t'` which represents a *tab* spacer. The line finishes with the sequence of two special instructions `'\r\n'` which represent, respectively, a carriage return (`'\r'`) and a newline (`'\n'`); for those with a long memory or trivia inclination, the carriage refers to old typewriter technology. This suggests that the information relevant to the creation of the network is contained in the list `lines` begining from the fifth line of the list. We can snoop around a bit more to confirm this. We do this with the following simple lines

```
>>> lines[5:10] # Elements lines[5] through lines[9]
['1\t0\r\n', '1\t2\r\n', '1\t3\r\n', '1\t4\r\n', '1\t5\r\n']
>>> lines[-1] # Last element of lines.
'36691\t8203\r\n'
```

Note that between the 5th and 10th lines of `lines`, the format of the content is the same. This also applies to the last line of `lines`, evoked by `lines[-1]` (this `python` convention on lists helps us read its elements as if we were locating list element going backwards from the end of the list, and hence the index `-1`).

In conclusion, we can feel pretty confident that from `lines[5]` onwards, all lines contain two nodes separated by `'\t'`.

To read this information and simultaneouosly create a network, we execute the following code

```
>>> E=nx.Graph() # Create the network variable E
>>> for line in lines[5:]: # Read from 5th line onwards
...     sl=line.split() # Split the string by blank space (tab)
...     i=sl[0] # First element of list sl
...     j=sl[1] # Second element of list sl
...     E.add_edge(i,j) # Link i,j in E
...
```

The code begins on the fifth line of the file and continues to the end of the file. Each line is split using the `python` string method `split` which takes a line such as `'0\t1\r\n'` and converts it into a new list with the form `['0', '1']`. The first element of this list, which we have called `sl`, is `sl[0]`, equal to `'0'` in our case; the second element if `sl[1]` and it is equal to `'1'`. We assign these two elements to `i` and `j`, respectively, and then add them to the links of the empty network `E` using the `networkx` method `E.add_edge(i,j)`. Going through all the remaining lines of `lines`, we produce the network `E` that we want to analyze.

Some interesting methods that can be used to analyze network `E` were mentioned before such as `.size()` (for the number of links) and `.order()` (for the number of nodes). For instance

```
>>> E.size() # Number of links in E
183831
>>> E.order() # Number of nodes in E
36692
```

It's interesting that the information printed in the header of the file that we saw above indicated the number of nodes and links for this network to be, respectively, 36692 and 367662. The nodes agree with `E.order()`, but not the links. Why? Because this network is a directed network and yet we have read it using a `Graph()` object from `networkx` which is *undirected*! Should we fix this? Not now. In this chapter, we are concentrating on generating $H(k)$ and Pr, and thus we can spare ourselves the pain for now.

To generate the histogram of degrees and subsequently its distribution, we need to go node by node in `E` and determine the degree. As we go, we must somehow store that the degree value in a way that counts how many times a given degree has appeared. Since we know that the degree satisfies

$$0 \leq k \leq n-1$$

for any node in a network, we can first create a `python` object (we use dictionaries) that will eventually hold the degree histogram. We can use the following instructions

```
>>> H={} # Define a dictionary that will be histogram
>>> n=E.order() # Make n equal to number of nodes in E
>>> for k in range(n): # Loop from k=0 to k=n-1
...     H[k]=0 # Initialize the histogram to 0 for all valid k
...
```

Now, to populate it

```
>>> for i in E.nodes(): # Loop over all nodes of E
...     k=E.degree(i) # For node i determine its degree
...     H[k]=H[k]+1 # H[k]-> H[k]+1 with node of degree k
...
```

Now `H` contains all frequencies of all the degrees. For instance, if we wanted to know how many nodes have degree 1, or 2, or 10, all we have to do is

```
>>> H[1]
11211
>>> H[2]
3800
>>> H[10]
588
```

There is an inconvenient characteristic to `H` as we have defined it. Although in principle a network can have a degree as large as $n-1$, in most real cases, this does not happen, specially when n is truly large. This means, for instance, that although our `H` is initialized for any possible k, it has a lot of *keys* `k` that are to some extent irrelevant. A more practical approach is to write the two pieces of code above in one go, simultaneously defining and populating the keys for `H` we care about (where there is a non-zero degree frequency). This is as follows (and replaces the two blocks of code just above

```
>>> H={}
>>> for i in E.nodes(): # Loop over all nodes of E
...     k=E.degree(i) # For node i determine its degree
...     H[k]=H.get(k,0)+1 # H[k]-> H[k]+1 with node of degree k
...
```

The result of this code is almost the same as before, except that none of the keys `k` is such that `H[k]` is equal to 0. We use this method now. The method `get` applied to a dictionary has the effect of either invoking the value of the dictionary with the key specified in its first argument and if the key has not defined, then it returns its other argument, in this case 0. Thus, `H.get(k,0)` returns `H[k]` if `H` already has the key `k`, and if not, it returns 0.

To answer questions such as those posed for the Watergate network, we can use some `python` facilities

```
>>> klist=H.keys() # Keys of H correspond to all degrees
>>> klist.sort() # Sort in increasing order elements in klist
>>> klist[0] # Smallest element of klist
1
```

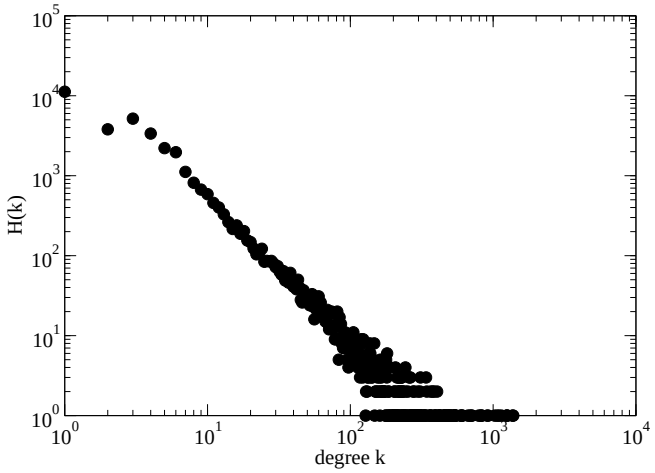

Figure 3: $H(k)$ versus k in double logarithmic scale for the social network emerging from email communications at Enron.

```
>>> klist[-1] # Largest element of klist
1383
```

The smallest degree in `E` is 1, the largest is $1,383$. What about the most probable?

```
>>> maxH=0
>>> for k in H.keys():
...     if H[k]>maxH:
...         maxH=H[k]
...         maxk=k
...
>>> maxH,maxk
(11211, 1)
```

which tells us that the largest frequency in $H(k)$ (or `H`) is $11,211$ which corresponds to degree 1.

Before we embark on actually discussing how to plot this histogram using `python`, I show it in Fig. 3 in double logarithmic scale. We can think about the three features we just extracted from the data moments ago: the minimum degree 1, visible along the horizontal axis, which has the first data point with coordinate 1 on this axis. We also said that, in fact, this degree

of $k = 1$ was the most frequent, with $H(k = 1) = 11,211$, also visible in the plot (the first point in the plot has a location along the vertical axis just above 10^4, consistent with $H(k = 1) = 11,211$). The largest degree is $1,383$, which means that this is the right-most location of any point along the horizontal axis (we see the right-most point of the plot with a horizontal coordinate of a little over 10^3, again consistent with $1,383$).

The next feature to pay attention to is the *shape of the histogram*. Note how $H(k)$ seems to decay almost as a line (in our double logarithmic scale, so be careful how this is interpreted). This kind of shape has generated much interest in the literature in the past 20 years of networks research (the so-called power-law degree distribution). If one is indeed able to confirm that the decay is linear, then the shape of this histogram and related distribution is known as a *power-law*. There are other shapes we find in real data, most notably Gaussian-like shapes that suggest a different kind of network in comparison to the one we are displaying now. We come back to this discussion soon.

To make a proper distribution out of this data, we must normalize the histogram. To do this, we first have to either add all $H(k)$, i.e., calculate $\sum_k H(k)$, or exercise the fact that we already know what the result is: n. For the sake of learning how to code a bit more, let us perform the sum and use it to create $\Pr(k)$

```
>>> sum=0. # Create variable into which sum of H goes
>>> for k in H.keys(): # Go through all degrees in H
...     sum=sum+H[k] # Add H[k] degree by degree
...
>>> sum # Print result of sum, which should be n
36692.0
>>> Pr={} # Now create new dictionary for probabilities
>>> for k in H.keys(): # Go through all degrees in H
...     Pr[k]=H[k]/sum # Normalize H to create Pr
...
```

Note that `sum` indeed produces the number of nodes as we expected. We can now plot $\Pr(k)$ to complete our analysis of the Enron dataset (see Fig. 4). For those that already know about logarithmic scales, you will recognize that normalizing $H(k)$ to create $\Pr(k)$ is reflected in the vertical scale of the plot simply as a rigid transport of $H(k)$ downwards. Mathematically, this is given by

$$y' = y/c \quad \Rightarrow \quad \log y' = \log y - \log c$$

where c is a constant. The equation shows that in logarithmic scale, dividing becomes equivalent to moving downwards (negative sign) by a constant

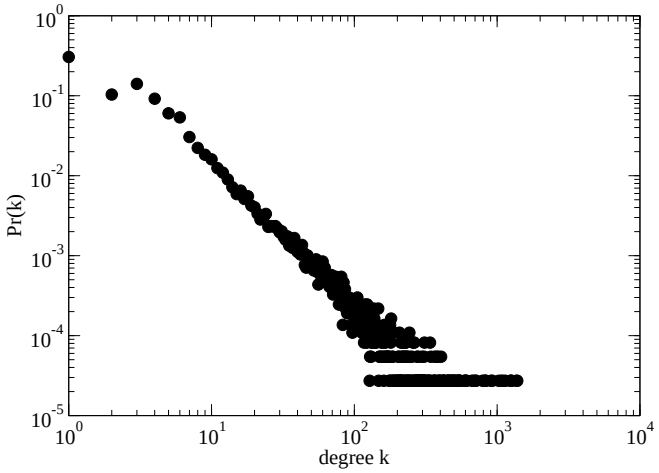

Figure 4: $Pr(k)$ versus k in double logarithmic scale for the social network emerging from email communications at Enron.

amount.

1.3 Some archetypes for Pr(k)

Having looked in detail at the steps required to create a histogram and distribution of degree, we now present a second example dataset to illustrate another typical situation one encounters when looking at large social data.

The dataset `'ca-CondMat-noself.txt'` is constituted by a set of links that emerge from individuals that coauthor research articles in a sub-field of physics. The rule that creates the links is that if, for instance, individuals i, j, h, g coauthor a paper, then links are generated between all pairs of the authors. Thus, an article with a coauthors generates $\binom{a}{2}$ links, one for each pair of a. Repeated coauthorship does not affect existing links. The probability distribution of k for this network, once again in double logarithmic plot, takes the shape shown in Fig. 5. Note that the plot curves inwards as opposed to remaining approximately linear as in Fig. 4. Making the plot with linear scale along the horizontal axis provides another clue about the overall behavior of the dataset (see Fig. 6). In this case, we see that Pr(k) bends outwards as k increases, signalling that its tail is slower than exponential (that is, that the probability of larger and larger k does not diminish

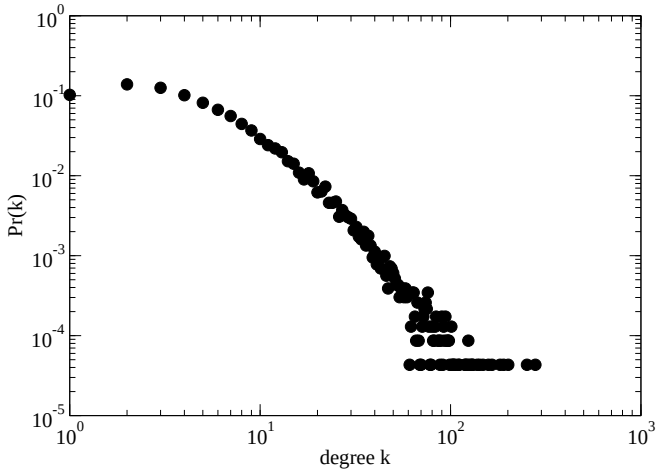

Figure 5: $Pr(k)$ versus k in double logarithmic scale for the social network emerging from coauthorship in the condensed matter physics preprint archive.

as fast as exponentially). Combining the observations of the behavior of $\Pr(k)$ in double logarithmic versus the linear logarithmic strongly suggests that the distribution falls under the category of a *heavy tail* distribution.

The distributions $\Pr(k)$ for the Enron email network and the coauthorship network represent two typical examples of probability distributions of degree found in large social networks.

There is yet another type of large network archetype that we often encounter. In this type of network, $\Pr(k)$ displays an exponential (or faster than exponential decay). Although there are various datasets that disply some of this behavior, it is useful to look at pure model networks for the simplicity of the distributions.

We introduce a so-called Erdős-Rényi model network, one in which the $\binom{n}{2}$ link indicators among n nodes are drawn randomly. Thus, any a_{ij} is present with probability p independent of the specific i and j. A network of this type can be generated via `networkx` with the routine `erdos_renyi_graph`

```
>>> ER=nx.erdos_renyi_graph(10000,0.0002) # n=10000 and p=2e-4.
>>> ER.order()
```

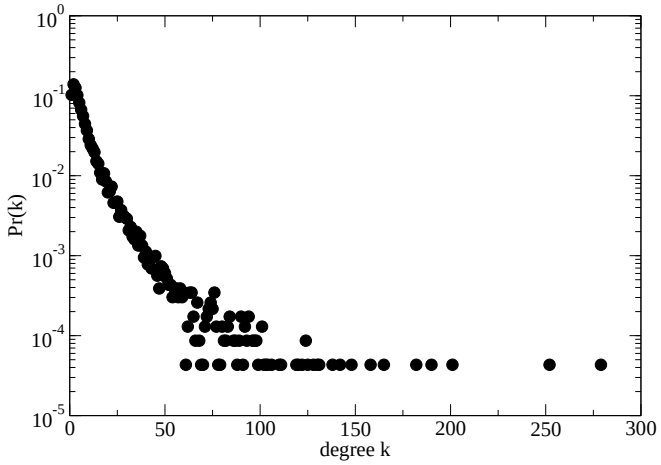

Figure 6: $Pr(k)$ versus k in linear-log scale for the social network emerging from coauthorship in the condensed matter physics preprint archive.

```
10000
>>> n=ER.order() # make n be the number of nodes
>>> ER.size() # m is random so each person has own m
9,824
>>> H={}
>>> for i in ER.nodes():
...     k=ER.degree(i)
...     H[k]=H.get(k,0)+1
...
>>> Pr={}
>>> for k in H.keys():
...     Pr[k]=float(H[k])/n
...
```

The distribution generated is shown in the linear (horizontal) logarithmic plot in Fig. 7. It has a behavior that suggests that it decays *exponentially* or faster, as the tail of the distribution does not show any outward-bending trend as k increases.

A summary to the comments above would be this. There are various kinds of degree distributions, and they tend to fall under some general categories:

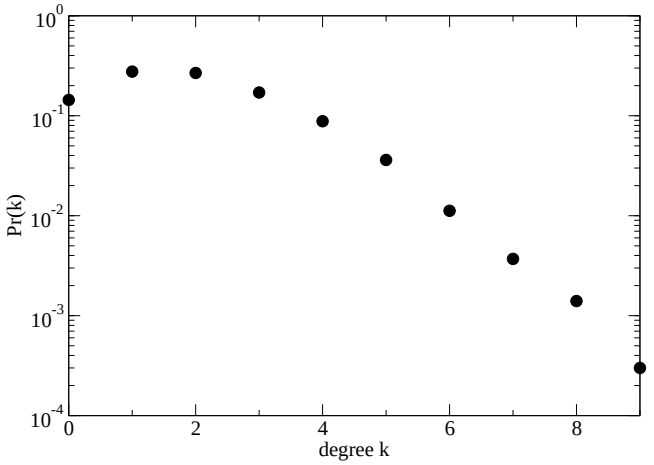

Figure 7: $\Pr(k)$ versus k of an Erdős-Rényi random network. Notice that in linear-logarithmic scale, the tail does not show any trend to bend outward as k increases. This set this distribution as a potential exponentially decaying one.

1. Heavy tail distributions, of which there are several kinds. The most common are power-law (as the Enron network) and subexponential (as the coauthorship network).

2. Narrow tail distributions that decay at least as fast as an exponential (or faster). Erdős-Rényi networks fall into this category, with the ddegree distribution decaying faster than exponentially (as a binomial, specifically).

The study of these different kinds of distributions on its own would seem like an academic exercise, but in fact, the main reason why we care about these distributions is because they usually carry information about other properties of the network. Two such properties that feel the effects of the degree distribution are the distributions of shortest path lengths and the distributions of triangles. And due to its dependence on triangles, local clustering properties are also affected by $\Pr(k)$

Let us study some of these relationships.

2 Distributions of local triangle counts

To understand the relevance of the degree distribution, let us revisit the datasets for Enron and condensed matter coauthorship. For both, we determine $\Pr(t)$, the probability that a randomly chosen node is visited by t triangles. To have a reference of the relation of this distribution with $\Pr(k)$, we plot the two together in Fig. 8. The plots show the parallel between the distributions.

Although there is indeed parallel, the distributions are not exactly right over one another. This should not be a surprise for several reasons. First, a number of triangles depends on their being a certain value of degree, but every node that has a given degree does not always show the same number of triangles. Second, triangles and degree have different "units" if you like. For a node of degree $k = 2$, only one triangle is possible, or $t \leq 1$. In general, the maximum number of triangles given a degree k is $\binom{k}{2}$, a result that should not be surprise given the definition of local clustering. These observations suggest that it may be possible to massage these distributions in order for them to match each other better. This, however, is a subtle task so I do not pursue this direction further.

Instead, I want us to look at the relation between the value of each node's degree and number of triangles. In Fig. 9 we find a scatter plot (k_i, t_i) for both datasets. Note 1) how closely k_i is related to t_i, as exhibited by the

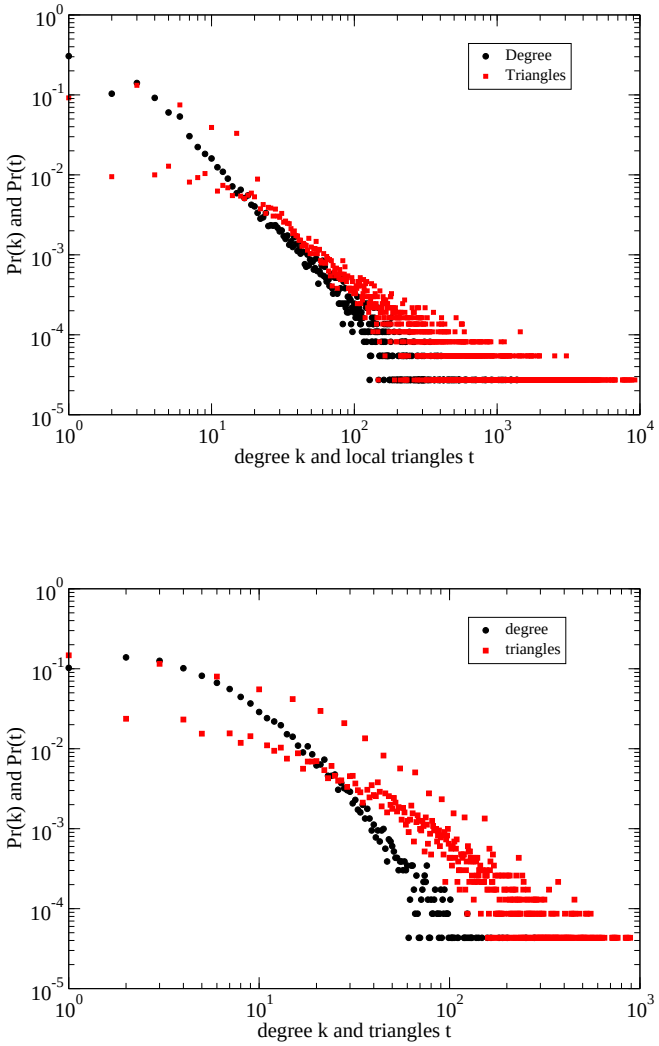

Figure 8: The distributions of degree $\Pr(k)$ and number of local triangles $\Pr(t)$ for the Enron email dataset (above) and the condensed matter coauthorship (below).

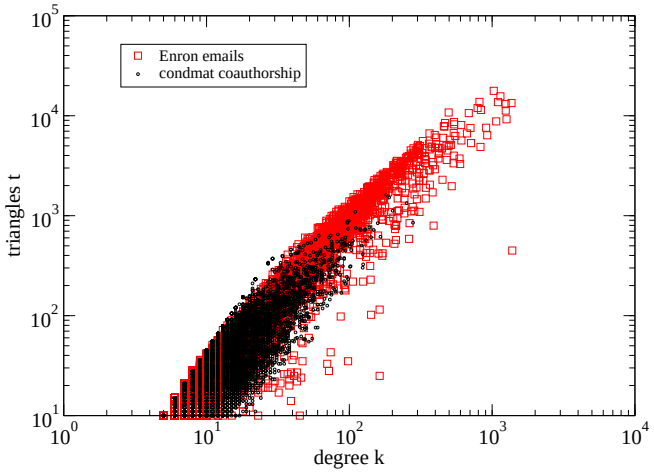

Figure 9: Scatter plots of k_i versus t_i for the network of emails in Enron (red) and the network of coauthorship in condensed matter physics (black).

very concentrated cloud of points making a line in the double logarithmic scale, and 2) how both datasets surprisingly are located in the same area of the plot! The first result, that k_i and t_i have a strong correlation with one another, is not altogether surprising in terms of our prior discussion. Also this result connects the triangles and degrees of each dataset with itself. The second result, however, seems very surprising: that data from two different settings overlap as they do in this case would not generally happen and therefore is hardly expected.

The first of these two results has some clear consequences. In probability, one can prove that if two variables are related, then their distributions are also related. This should come as no surprise. We often use this notion in our lives; for example, the probability of a longer than usual commute to work is correlated with the probability of bad weather. In our case, this means that $\Pr(t)$ could be explained by $\Pr(k)$ and the relationship captured by Fig. 9 (there are various ways to quantify this relation). Just to provide a sense of the strength of the relation, the correlation coefficient for the coauthorship data is 0.881 and for the Enron data is 0.919.

Now, the "amount" of $\Pr(t)$ that is explained by $\Pr(k)$ is not always the same. In some cases, $\Pr(t)$ may be mostly explained by $\Pr(k)$, while in

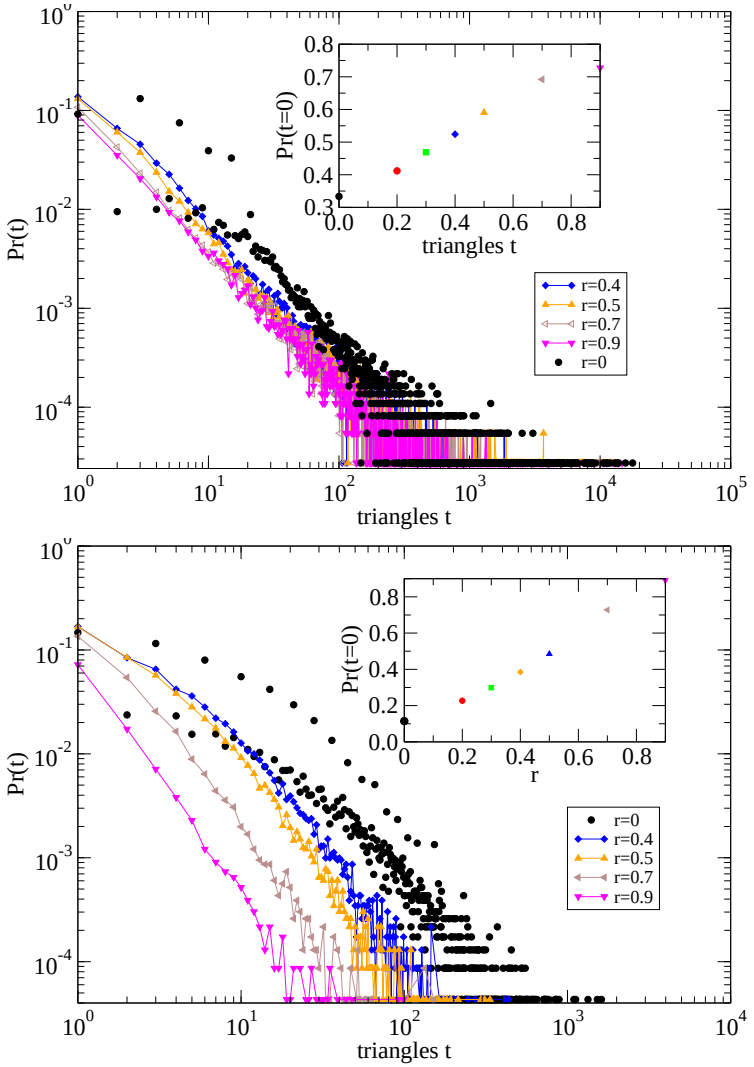

Figure 10: $Pr(t)$ for the Enron network (top) and the condensed matter coauthorship (bottom), along with various percentages r of link rewirings. The main plots show the distributions for $t \geq 1$ and the insets show $\Pr(t = 0)$. Each network is affected differently. Note there is a qualitative difference between the two datasets in that in the Enron network the shape of the distribution does not change as much as in the coauthorship network when r increases.

others, $\Pr(k)$ may only partially explain $\Pr(t)$. One of the main jobs of the social network analyst is to determine the level of this relation.

How do these levels of dependence look? Figure 10 shows $\Pr(t)$ for the Enron email network (top) and the condensed matter physics coauthorship networks measured from data ($r = 0$ where r is the fraction of randomly rewired links as in Lec. 3), as well as after random rewiring of the network at $r = 0.4, 0.5, 0.7$, and 0.9. An key factor to consider is that the algorithm generally *fixes the degree distribution*, i.e., while $\Pr(t)$ changes with r, $\Pr(k)$ is constrained to remain unchanged! This means that r does not affect k_i for each node *but does change* t_i. However, a closer inspection of Figure 10 shows subtlety: while the condensed matter coauthorship exhibits considerable changes in $\Pr(t)$, the changes in the Enron network are smaller. We can observe this simply from the amount of change in the plots for $\Pr(t)$ as a function of r. This suggests that while in the Enron email network local triangle counts t are more closely dependent on the values of k, the same does not hold for the coauthorship network.

To bring this point home, we also show Fig. 11, which contains the scatter plots of (k_i, t_i) for both networks and associated random rewirings of them. It is clear that while rewiring has a large effect on the coauthorship network, its effect on the Enron network is more limited. This leads us to the conclusion that t_i are largely explained by k_i is the Enron network, but in the coauthorship network, k_i is only one of the factors that can help us determine t_i for a node. Furthermore, to characterize any change in correlation between k_i and t_i, we present Fig. 12, which supports our interpretation. In fact, note that for the Enron network, rewiring doesn't seem to affect much the correlation, and in fact a little bit of rewiring *increases* correlation! Is the Enron network not that special?

Let us contrast this to what happens with v-shapes. We know from our earlier lectures that

$$\#\text{v-shapes visiting } i = v_i = \binom{k_i}{2}. \tag{2}$$

This means that once we know the degree of a node, we know the number of v-shapes that visit it. What this should lead to is a precise line connecting k_i and v_i. This relation is tested in Fig. 13, where we can see exactly what we predict. A consequence of this strict relation between k_i and v_i is that random rewirings of the network that leave k_i intact for each node also lead to the same v_i and thus, for instance, $\Pr(v_i)$ is *entirely* explained by $\Pr(k_i)$. This is visible in Fig. 14

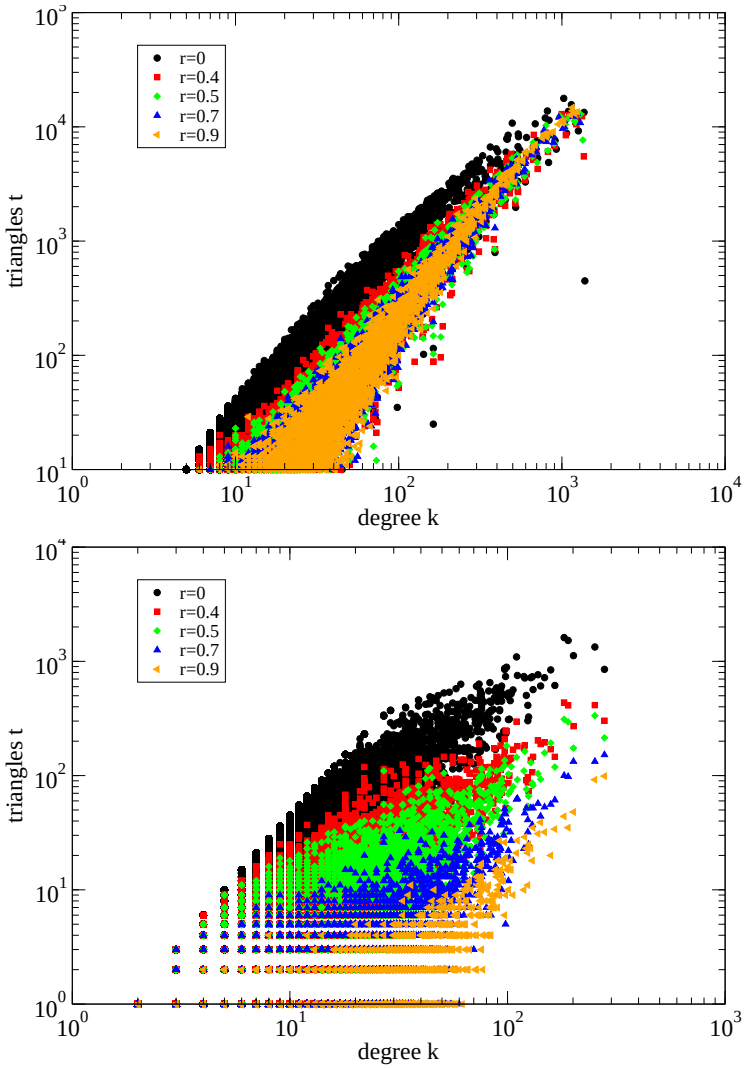

Figure 11: Scatter plots for (k_i, t_i) for the Enron (top) and condensed matter coauthorship networks as measured ($r = 0$) and for various fractions of rewiring ($r = 0.4, 0.5, 0.7, 0.9$). In the Enron network, the effects of rewiring a much more moderate than in the coauthorship network, indicating that k_i has a greater power to determine t_i in the Enron email network than in the coauthorship network.

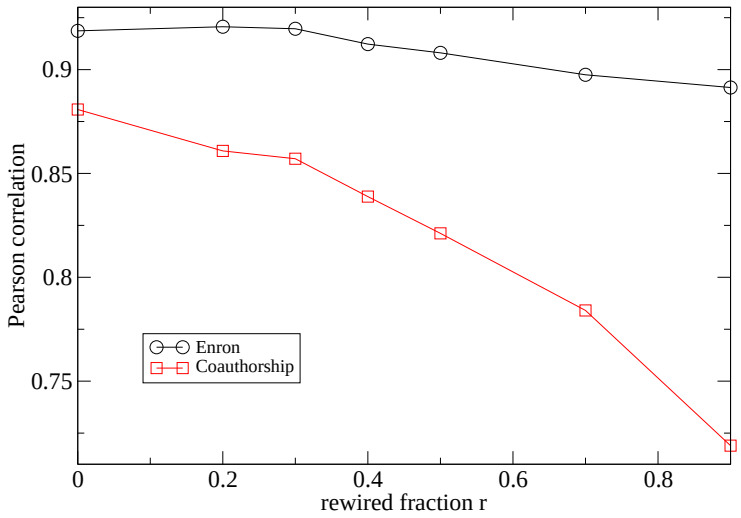

Figure 12: Pearson correlation coefficient for the scatter data (k_i, t_i) for both the Enron and the coauthorship networks under various levels of disorder r.

To wrap up our analysis, let us now turn to local clustering coefficients again and think about how their values are influenced by what we have learned. Recalling the definition of local clustering

$$c_i = \frac{t_i}{v_i} = \frac{t_i}{\binom{k_i}{2}}, \tag{3}$$

we can see that the denominator is completely specified from the value of k_i. The numerator, on the other hand, is partially specified, because a given value of k_i, as we learned above (and visualized in Fig. 9) can lead to various values of t_i. We do know, of course, that there is a high correlation between k_i and t_i and this means that knowing k_i does lead to a great deal of information about c_i. This problem could be treated with some mathematical precision. Alternatively, we can also approach it computationally. Just as we can create scatter plots of k_i and t_i (or v_i) we can also make such plots between k_i and c_i, and subsequently

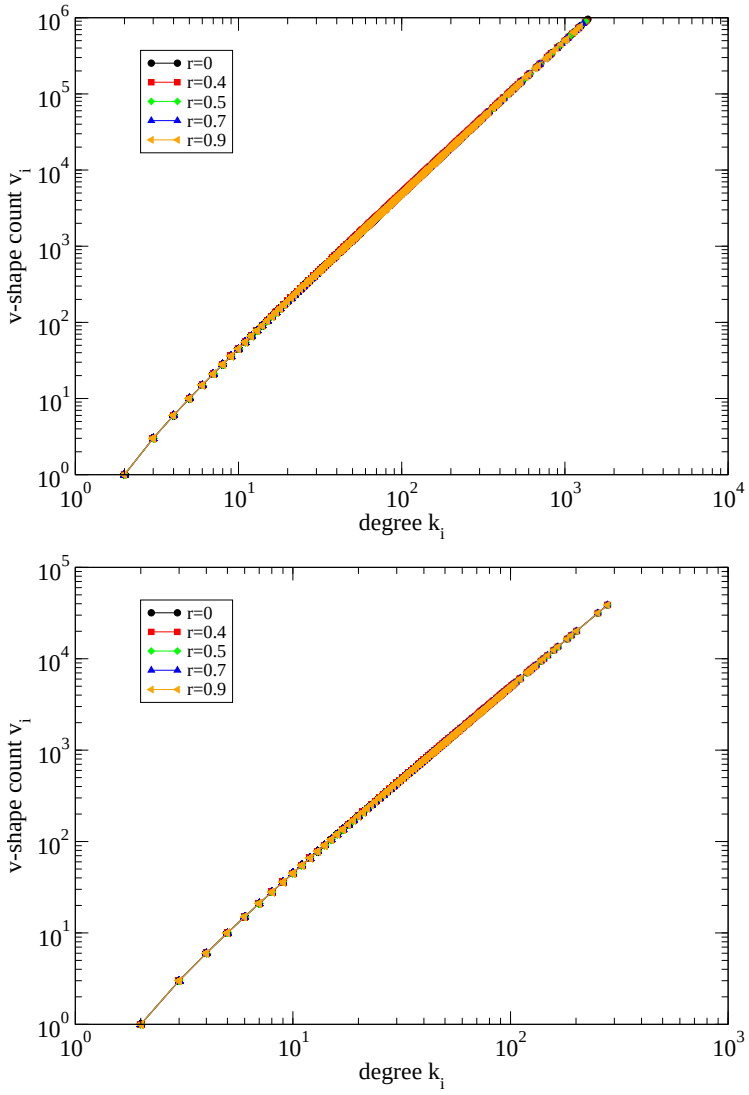

Figure 13: Scatter plots for (k_i, v_i) for the Enron (top) and condensed matter coauthorship networks as measured ($r = 0$) and for various fractions of rewiring ($r = 0.4, 0.5, 0.7, 0.9$). In these plots, the relation between k_i and v_i is precise, which means that k_i entirely predicts v_i.

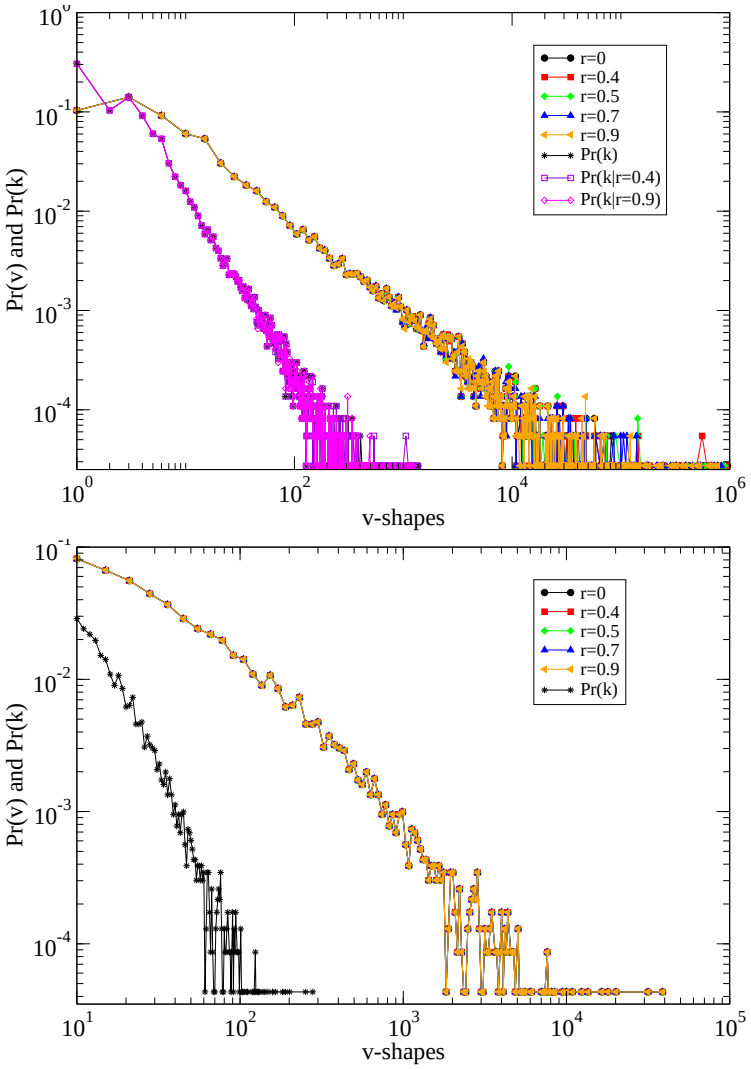

Figure 14: $\Pr(v_i)$ for Enron (top) and condensed matter physics coauthorship, and corresponding distributions with increasing rates of rewiring of links $r = 0.4, 0.5, 0.7, 0.9$. In addition, we plot $\Pr(k)$ for each dataset. Note that, in each plot, all the values of r produce overlapping lines, indicating the irrelevance of r. If you look closely, we can trace even small fluctuations on $\Pr(v_i)$ to $\Pr(k)$.