

Module IV - Basic Analysis

Drew Conway and Aric Hagberg

June 29, 2010

Loading data from multiple sources

- ▶ Local network data files
- ▶ Building directly from the Internet

Agenda for Module IV

Loading data from multiple sources

- ▶ Local network data files
- ▶ Building directly from the Internet

Brief review of Python dictionaries

- ▶ Why is the dict so useful?
- ▶ How NetworkX utilizes it?

Loading data from multiple sources

- ▶ Local network data files
- ▶ Building directly from the Internet

Brief review of Python dictionaries

- ▶ Why is the `dict` so useful?
- ▶ How NetworkX utilizes it?

Running basic centralities

- ▶ Degree, Closeness, Betweenness Eigenvector
- ▶ Calculating degree distribution
- ▶ Plotting statistics using `matplotlib`
- ▶ Calculating cliques, clustering and transitivity

Loading data from multiple sources

- ▶ Local network data files
- ▶ Building directly from the Internet

Brief review of Python dictionaries

- ▶ Why is the dict so useful?
- ▶ How NetworkX utilizes it?

Running basic centralities

- ▶ Degree, Closeness, Betweenness Eigenvector
- ▶ Calculating degree distribution
- ▶ Plotting statistics using `matplotlib`
- ▶ Calculating cliques, clustering and transitivity

Outputting data into multiple formats

- ▶ Writing network data
- ▶ Saving network analysis statistics

Loading data from multiple sources

- ▶ Local network data files
- ▶ Building directly from the Internet

Brief review of Python dictionaries

- ▶ Why is the dict so useful?
- ▶ How NetworkX utilizes it?

Running basic centralities

- ▶ Degree, Closeness, Betweenness Eigenvector
- ▶ Calculating degree distribution
- ▶ Plotting statistics using `matplotlib`
- ▶ Calculating cliques, clustering and transitivity

Outputting data into multiple formats

- ▶ Writing network data
- ▶ Saving network analysis statistics

Basic visualization

- ▶ Review of NetworkX's plotting algorithms
- ▶ Adding analysis to visualization

As we have seen, one of the main advantages of working with NetworkX is that it can read many different network formats

- ▶ For those that are unfamiliar with working at the **command-line**, however, the process can be confusing

NX syntax for loading a file

```
>>> G = nx.read_format("path/to/file.txt", ...options...)
```


Net variable

↑

NX function, file directory path

Graph type, nodes type, etc.

As we have seen, one of the main advantages of working with NetworkX is that it can read many different network formats

- ▶ For those that are unfamiliar with working at the **command-line**, however, the process can be confusing

NX syntax for loading a file

```
>>> G = nx.read_format("path/to/file.txt", ...options...)
```


Net variable

NX function, file directory path



Graph type, nodes type, etc.

As we have seen, one of the main advantages of working with NetworkX is that it can read many different network formats

- ▶ For those that are unfamiliar with working at the **command-line**, however, the process can be confusing

NX syntax for loading a file

```
>>> G = nx.read_format("path/to/file.txt", ...options...)
```

↑
Net variable

↑
NX function, file directory path

↑
Graph type, nodes type, etc.

As we have seen, one of the main advantages of working with NetworkX is that it can read many different network formats

- ▶ For those that are unfamiliar with working at the **command-line**, however, the process can be confusing

NX syntax for loading a file

```
>>> G = nx.read_format("path/to/file.txt", ...options...)
```


Net variable

↑

NX function, file directory path



Graph type, nodes type, etc.

Let's try!

- ▶ We will load the edge list of Hartford drug users network
- ▶ Specify that the network be a directed graph, and the nodes be integers
- ▶ Use `nx.info()` to check that data has been loaded correctly

Loading the Hartford drug users network

Starting NetworkX and loading data

```
1 >>> hartford=nx.read_edgelist("../data/hartford_drug.txt",create_using=nx.DiGraph(),nodetype=int)
2 >>> nx.info(hartford)
3 Name:
4 Type:          DiGraph
5 Number of nodes: 212
6 Number of edges: 337
7 Average in degree: 1.5896
8 Average out degree: 1.5896}
```

Starting NetworkX and loading data

```
1 >>> hartford=nx.read_edgelist("../data/hartford_drug.txt",create_using=nx.DiGraph(),nodetype=int)
2 >>> nx.info(hartford)
3 Name:
4 Type:          DiGraph
5 Number of nodes: 212
6 Number of edges: 337
7 Average in degree: 1.5896
8 Average out degree: 1.5896}
```

What did we just do?

Starting NetworkX and loading data

```
1 >>> hartford=nx.read_edgelist("../data/hartford_drug.txt",create_using=nx.DiGraph(),nodetype=int)
2 >>> nx.info(hartford)
3 Name:
4 Type:                DiGraph
5 Number of nodes:     212
6 Number of edges:     337
7 Average in degree:   1.5896
8 Average out degree:  1.5896}
```

What did we just do?

- ▶ Used the `read_edgelist` function to load EL file

Starting NetworkX and loading data

```
1 >>> hartford=nx.read_edgelist("../data/hartford_drug.txt",create_using=nx.DiGraph(),nodetype=int)
2 >>> nx.info(hartford)
3 Name:
4 Type:                DiGraph
5 Number of nodes:     212
6 Number of edges:     337
7 Average in degree:   1.5896
8 Average out degree:  1.5896}
```

What did we just do?

- ▶ Used the `read_edgelist` function to load EL file
- ▶ Specified path to Hartford drug users file

Starting NetworkX and loading data

```
1 >>> hartford=nx.read_edgelist("../data/hartford_drug.txt",create_using=nx.DiGraph(),nodetype=int)
2 >>> nx.info(hartford)
3 Name:
4 Type:                DiGraph
5 Number of nodes:      212
6 Number of edges:      337
7 Average in degree:    1.5896
8 Average out degree:   1.5896}
```

What did we just do?

- ▶ Used the `read_edgelist` function to load EL file
- ▶ Specified path to Hartford drug users file
- ▶ Used the `create_using` option to force NX to create as a directed graph

Starting NetworkX and loading data

```
1 >>> hartford=nx.read_edgelist("../data/hartford_drug.txt",create_using=nx.DiGraph(),nodetype=int)
2 >>> nx.info(hartford)
3 Name:
4 Type:                DiGraph
5 Number of nodes:     212
6 Number of edges:     337
7 Average in degree:   1.5896
8 Average out degree:  1.5896}
```

What did we just do?

- ▶ Used the `read_edgelist` function to load EL file
- ▶ Specified path to Hartford drug users file
- ▶ Used the `create_using` option to force NX to create as a directed graph
- ▶ Used the `nodetype` option to force NX to store nodes as integers

Starting NetworkX and loading data

```
1 >>> hartford=nx.read_edgelist("../data/hartford_drug.txt",create_using=nx.DiGraph(),nodetype=int)
2 >>> nx.info(hartford)
3 Name:
4 Type:                DiGraph
5 Number of nodes:     212
6 Number of edges:     337
7 Average in degree:   1.5896
8 Average out degree:  1.5896}
```

What did we just do?

- ▶ Used the `read_edgelist` function to load EL file
- ▶ Specified path to Hartford drug users file
- ▶ Used the `create_using` option to force NX to create as a directed graph
- ▶ Used the `nodetype` option to force NX to store nodes as integers
- ▶ Used the `info` function to check that it all worked

Starting NetworkX and loading data

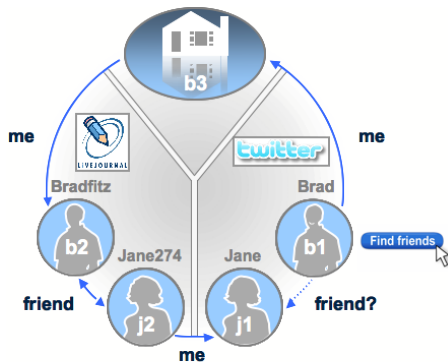
```
1 >>> hartford=nx.read_edgelist("../data/hartford_drug.txt",create_using=nx.DiGraph(),nodetype=int)
2 >>> nx.info(hartford)
3 Name:
4 Type:                DiGraph
5 Number of nodes:      212
6 Number of edges:      337
7 Average in degree:    1.5896
8 Average out degree:   1.5896}
```

What did we just do?

- ▶ Used the `read_edgelist` function to load EL file
- ▶ Specified path to Hartford drug users file
- ▶ Used the `create_using` option to force NX to create as a directed graph
- ▶ Used the `nodetype` option to force NX to store nodes as integers
- ▶ Used the `info` function to check that it all worked

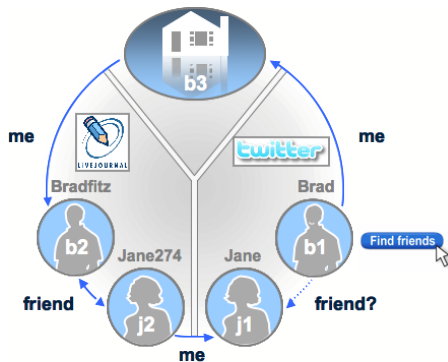
Some formats may have more or less options, **always check the documentations!**

Building the social network among LiveJournal users



Perhaps the most powerful aspect of NetworkX is its ability to work in Python to generate networks from live-streaming data

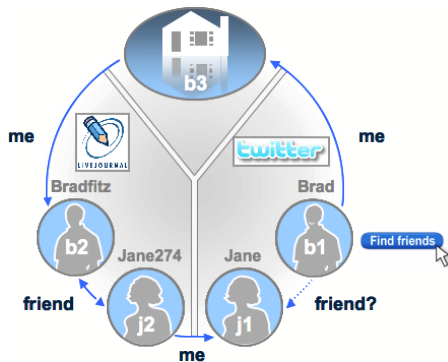
Building the social network among LiveJournal users



Perhaps the most powerful aspect of NetworkX is its ability to work in Python to generate networks from live-streaming data

- In Python, use NetworkX, cjson and a other standard scientific libraries to parse Google's SocialGraph data

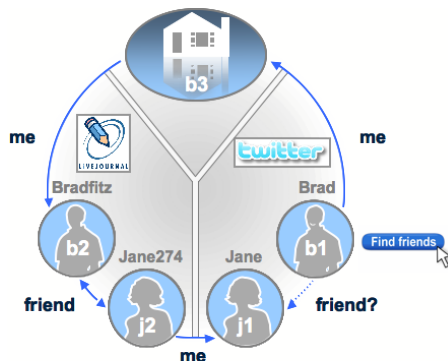
Building the social network among LiveJournal users



Perhaps the most powerful aspect of NetworkX is its ability to work in Python to generate networks from live-streaming data

- ▶ In Python, use NetworkX, cjson and a other standard scientific libraries to parse Google's SocialGraph data
- ▶ Using a "seed" user, we will build out a network

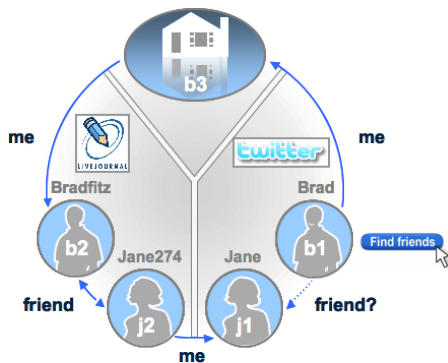
Building the social network among LiveJournal users



Perhaps the most powerful aspect of NetworkX is its ability to work in Python to generate networks from live-streaming data

- ▶ In Python, use NetworkX, cjson and a other standard scientific libraries to parse Google's SocialGraph data
- ▶ Using a "seed" user, we will build out a network
- ▶ Through a process called "k-snowball searching"
 $seed \rightarrow friend \rightarrow \dots \rightarrow friend_k$

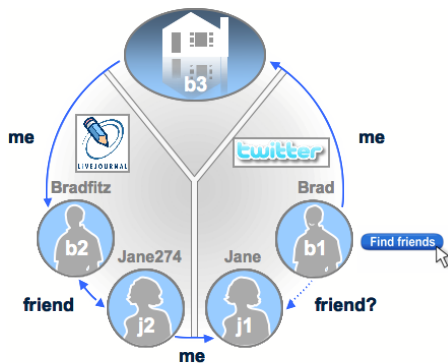
Building the social network among LiveJournal users



Perhaps the most powerful aspect of NetworkX is its ability to work in Python to generate networks from live-streaming data

- ▶ In Python, use NetworkX, cjson and a other standard scientific libraries to parse Google's SocialGraph data
- ▶ Using a "seed" user, we will build out a network
- ▶ Through a process called "k-snowball searching"
 $seed \rightarrow friend \rightarrow \dots \rightarrow friend_k$
 - ▶ Seed: imichaeldotorg.livejournal.com
 - ▶ $k = 3$

Building the social network among LiveJournal users



Perhaps the most powerful aspect of NetworkX is its ability to work in Python to generate networks from live-streaming data

- ▶ In Python, use NetworkX, cjson and a other standard scientific libraries to parse Google's SocialGraph data
- ▶ Using a "seed" user, we will build out a network
- ▶ Through a process called "k-snowball searching"
 $seed \rightarrow friend \rightarrow \dots \rightarrow friend_k$
 - ▶ Seed: imichaeldotorg.livejournal.com
 - ▶ $k = 3$
- ▶ Note the low value of k

The code, part 1

Loading the libraries

```
1 from json import *
2 from urllib import *
3 from time import *
4 from scipy import array, unique
5 ...
6 if __name__ == "__main__":
7     seed="imichaeldotorg"
8     seed_url="http://"+seed+".livejournal.com"
9     # Scrape, parse and build seed's ego net
10    sg=get_sg(seed_url)
11    net,newnodes=create_egonet(sg)
12    nx.write_pajek(net,"../data/"+seed+"_ego.net")
13    nx.info(net)
```

Scraping egonet relationships from seed

```
1 def get_sg(seed_url):
2     sgapi_url="http://socialgraph.apis.google.com/lookup?q="+seed_url+"&edo=1&edi=1&fme=1&pretty=0"
3     try:
4         furl=urlopen(sgapi_url)
5         fr=furl.read()
6         furl.close()
7         return fr
8     except IOError:
9         print "Could not connect to website"
10        print sgapi_url
11        return {}
```

Build egonet and snowball

Creating the egonet

```
1 def create_egonet(s):
2     try:
3         raw=decode(s)
4         G=nx.DiGraph()
5         pendants=[]
6         n=raw['nodes']
7         nk=n.keys()
8         G.name=str(nk)
9         pendants=[]
10        for a in range(0,len(nk)):
11            for b in range(0,len(nk)):
12                if a!=b:
13                    G.add_edge(nk[a],nk[b])
14        for k in nk:
15            ego=n[k]
16            ego_out=ego['nodes_referenced']
17            for o in ego_out:
18                G.add_edge(k,o)
19                pendants.append(o)
20            ego_in=ego['nodes_referenced_by']
21            for i in ego_in:
22                G.add_edge(i,k)
23                pendants.append(i)
24        pendants=array(pendants, dtype=str)
25        pendants.flatten()
26        pendants=unique(pendants)
27        return G, pendants
28    except DecodeError:
29        ...
30    except KeyError:
```

Rolling the snowball

```
1 def snowball_round(G, seeds, myspace=False):
2     t0=time()
3     if myspace:
4         seeds=get_myspace_url(seeds)
5     sb_data=[]
6     for s in range(0,len(seeds)):
7         s_sg=get_sg(seeds[s])
8         new_ego, pen=create_egonet(s_sg)
9         for p in pen:
10             sb_data.append(p)
11         if s<1:
12             sb_net=nx.compose(G, new_ego)
13         else:
14             sb_net=nx.compose(new_ego, sb_net)
15     del new_ego
16     if s==round(len(seeds)*0.2):
17         sb_net.name='20% complete'
18         nx.info(sb_net)
19         print 'AT: '+strftime('%m/%d/%Y, %H:%M:%S', gmtime())
20         print ''
21     ...
22     # More time keeping, probably a MUCH better way to do this
23     sb_data=array(sb_data)
24     sb_data.flatten()
25     sb_data=unique(sb_data)
26     nx.info(sb_net)
27     return sb_net, sb_data
```

Build the whole network

Step	Nodes	Edges	Mean Degree	Density
Seed	5	5	2.0	0.25
$k = 2$	75	115	3.0	0.02
$k = 3$	4,938	8,659	3.5	$3.6(10^{-4})$

Build the whole network

Step	Nodes	Edges	Mean Degree	Density
Seed	5	5	2.0	0.25
$k = 2$	75	115	3.0	0.02
$k = 3$	4,938	8,659	3.5	$3.6(10^{-4})$

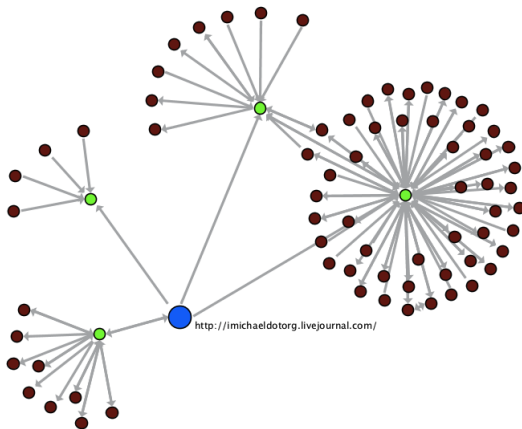
- Our seed is abnormally isolated, with only four neighbors

Build the whole network

Step	Nodes	Edges	Mean Degree	Density
Seed	5	5	2.0	0.25
$k = 2$	75	115	3.0	0.02
$k = 3$	4,938	8,659	3.5	$3.6(10^{-4})$

► Our seed is abnormally isolated, with only four neighbors

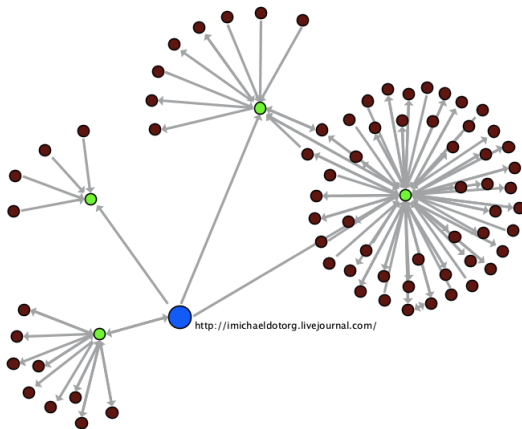
► Large jump after first snowball



Build the whole network

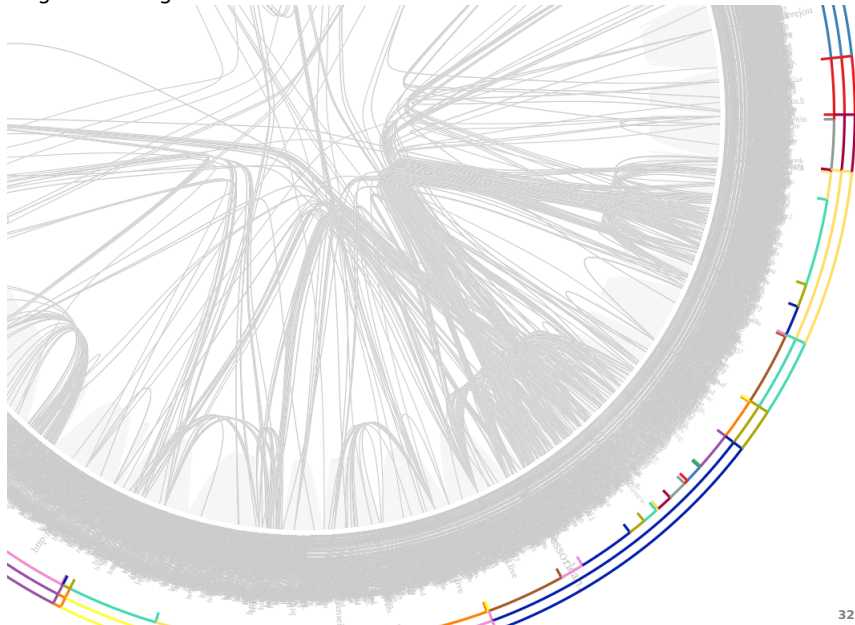
Step	Nodes	Edges	Mean Degree	Density
Seed	5	5	2.0	0.25
$k = 2$	75	115	3.0	0.02
$k = 3$	4,938	8,659	3.5	$3.6(10^{-4})$

- ▶ Our seed is abnormally isolated, with only four neighbors
- ▶ Large jump after first snowball
- ▶ Massive structural leap at $k = 3$



The full network

To get a feeling for the size of the full network...



The dict type is a data structure that represents a key→value mapping

Working with the dict type

```
1 # Keys and values can be of any data type
2 >>> fruit_dict={"apple":1,"orange":[0.23,0.11],"banana":True}
3 # Can retrieve the keys and values as Python lists (vector)
4 >>> fruit_dict.keys()
5 ["orange","apple","banana"]
6 # Or create a (key,value) tuple
7 >>> fruit_dict.items()
8 [{"orange",[0.23,0.11]},"apple",1],["Banana",True]]
9 # This becomes especially useful when you master Python "list comprehension"
```

The dict type is a data structure that represents a key→value mapping

Working with the dict type

```
1 # Keys and values can be of any data type
2 >>> fruit_dict={"apple":1,"orange":[0.23,0.11],"banana":True}
3 # Can retrieve the keys and values as Python lists (vector)
4 >>> fruit_dict.keys()
5 ["orange","apple","banana"]
6 # Or create a (key,value) tuple
7 >>> fruit_dict.items()
8 [{"orange":[0.23,0.11]},"apple",1],["Banana",True]]
9 # This becomes especially useful when you master Python "list comprehension"
```

The Python dictionary is an extremely flexible and useful data structure, making it one of the primary advantages of Python over other languages

- ▶ This is particularly useful when performing analysis on networks, where node labels are natural keys

The dict type is a data structure that represents a key→value mapping

Working with the dict type

```
1 # Keys and values can be of any data type
2 >>> fruit_dict={"apple":1,"orange":[0.23,0.11],"banana":True}
3 # Can retrieve the keys and values as Python lists (vector)
4 >>> fruit_dict.keys()
5 ["orange","apple","banana"]
6 # Or create a (key,value) tuple
7 >>> fruit_dict.items()
8 [{"orange":[0.23,0.11]},"apple",1},{"Banana",True}]
9 # This becomes especially useful when you master Python "list comprehension"
```

The Python dictionary is an extremely flexible and useful data structure, making it one of the primary advantages of Python over other languages

- ▶ This is particularly useful when performing analysis on networks, where node labels are natural keys

Now, try creating a dict of your own

From the documentation...

networkx.closeness centrality

```
closeness centrality(G, v=None, weighted_edges=False)
```

Compute closeness centrality for nodes.

Closeness centrality at a node is 1/average distance to all other nodes.

Parameters: **G** : graph

A networkx graph

v : node, optional

Return only the value for node v.

weighted_edges : bool, optional

Consider the edge weights in determining the shortest paths. If False, all edge weights are considered equal.

Returns: **nodes** : dictionary

Dictionary of nodes with closeness centrality as the value.

NetworkX's metric's make extensive use of the dict type

- In this case the key→value mapping is of the form: {node_label: metric}

Let's look at an example:

```
1 >>> in_cen=nx.in_degree_centrality(hartford)
2 >>> in_cen
3 \{1: 0.014218009478672987, 2: 0.018957345971563982,...
4 ...
5 90: 0.0047393364928909956, 293: 0.0\}
```

We can see that node #90 has in-degree centrality 0.0047

- But we can do so much more!

Running multiple measures

For our first analysis in NetworkX, we will do some basic network manipulation, then run multiple measures to find highest centrality nodes

- ▶ First, we will need to convert to an undirected network, and extract the main component

```
1 # We will symmetrize for simplicity
2 >>> hartford_ud=hartford.to_undirected()
3 # The network also has many small components, but for
4 # this analysis we are interested in the largest
5 >>> hartford_mc=hartford_main=nx.connected_component_subgraphs(hartford_ud)[0]
```

Next, we will calculate multiple measures

```
1 # Betweenness centrality
2 >>> bet_cen=nx.betweenness_centrality(hartford_mc)
3 # Closeness centrality
4 >>> clo_cen=nx.closeness_centrality(hartford_mc)
5 # Eigenvector centrality
6 >>> eig_cen=nx.eigenvector_centrality(hartford_mc)
```

Finding most central actors

To find the most central actors we will use Python's list comprehension technique to do basic data manipulation on our centrality dictionaries

```
1 def highest centrality(cent_dict):
2     """Returns node key with largest value from
3     NX centrality dict"""
4     # Create ordered tuple of centrality data
5     cent_items=cent_dict.items()
6     # List comprehension!
7     cent_items=[(b,a) for (a,b) in cent_items]
8     # Sort in descending order
9     cent_items.sort()
10    cent_items.reverse()
11    return cent_items[0][1]
```

Now, just ask for the answer

Finding Most central actors

```
1 >>> print("Actor "+str(highest centrality(bet_cen))+ " has the highest Betweenness centrality")
2 Actor 82 has the highest Betweenness centrality
```

Finding most central actors

To find the most central actors we will use Python's list comprehension technique to do basic data manipulation on our centrality dictionaries

```
1 def highest centrality(cent_dict):
2     """Returns node key with largest value from
3     NX centrality dict"""
4     # Create ordered tuple of centrality data
5     cent_items=cent_dict.items()
6     # List comprehension!
7     cent_items=[(b,a) for (a,b) in cent_items]
8     # Sort in descending order
9     cent_items.sort()
10    cent_items.reverse()
11    return cent_items[0][1]
```

List comprehension

- ▶ Given a dict: `d={1: 0.15, 2: 0.67}`
- ▶ `d.items()` → `[(1,0.15),(2,0.67)]`
- ▶ `d=[(b,a) for (a,b) in d]` → `[(0.15,1),(0.67,2)]`

Now, just ask for the answer

Finding Most central actors

```
1 >>> print("Actor "+str(highest centrality(bet_cen))+ " has the highest Betweenness centrality")
2 Actor 82 has the highest Betweenness centrality
```

Finding most central actors

To find the most central actors we will use Python's list comprehension technique to do basic data manipulation on our centrality dictionaries

```
1 def highest centrality(cent_dict):
2     """Returns node key with largest value from
3     NX centrality dict"""
4     # Create ordered tuple of centrality data
5     cent_items=cent_dict.items()
6     # List comprehension!
7     cent_items=[(b,a) for (a,b) in cent_items]
8     # Sort in descending order
9     cent_items.sort()
10    cent_items.reverse()
11    return cent_items[0][1]
```

List comprehension

- ▶ Given a dict: `d={1: 0.15, 2: 0.67}`
- ▶ `d.items()` → `[(1,0.15),(2,0.67)]`
- ▶ `d=[(b,a) for (a,b) in d]` → `[(0.15,1),(0.67,2)]`

Here, we use list comprehension in order to use Python's built-in sort and reverse list functions

Now, just ask for the answer

Finding Most central actors

```
1 >>> print("Actor "+str(highest centrality(bet_cen))+ " has the highest Betweenness centrality")
2 Actor 82 has the highest Betweenness centrality
```


One of the most popular network level statistical description of a network is its degree distribution

- In NetworkX this is a simply one-line operation

Get list of degree rank frequency

```
1 # Create a Barabasi-Albert network
2 >>> ba_net=barabasi_albert_graph(1000,2)
3 # Built-in function for degree distribution
4 >>> dh=degree_histogram(ba_net)
```

Calculating degree distribution

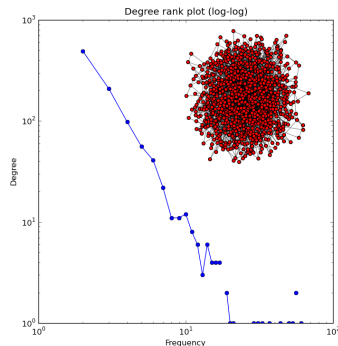
One of the most popular network level statistical description of a network is its degree distribution

- In NetworkX this is a simply one-line operation

Get list of degree rank frequency

```
1 # Create a Barabasi-Albert network
2 >>> ba_net=barabasi_albert_graph(1000,2)
3 # Built-in function for degree distribution
4 >>> dh=degree_histogram(ba_net)
```

- As we will see next, we can use `matplotlib` to take this data and create publication ready plots
- Ex. from http://networkx.lanl.gov/examples/drawing/degree_histogram.html



Calculating basic community structure

Often in network analysis we are interested in estimating the cohesiveness of a network, or the communities that exists within the structure

Often in network analysis we are interested in estimating the cohesiveness of a network, or the communities that exists within the structure

Cliques

- ▶ Maximal cliques are the largest complete subgraph containing a given point. There are several algorithms for finding cliques, including Bron & Kerbosch (1973), Tomita, Tanaka and Takahashi (2006), Cazals and Karande (2008)

Often in network analysis we are interested in estimating the cohesiveness of a network, or the communities that exists within the structure

Cliques

- ▶ Maximal cliques are the largest complete subgraph containing a given point. There are several algorithms for finding cliques, including Bron & Kerbosch (1973), Tomita, Tanaka and Takahashi (2006), Cazals and Karande (2008)

Clustering

- ▶ For each node find the fraction of possible triangles that exist, $c_v = \frac{2T(v)}{\deg(v)(\deg(v)-1)}$, where $T(v)$ is the number of triangles through node v .

Often in network analysis we are interested in estimating the cohesiveness of a network, or the communities that exists within the structure

Cliques

- ▶ Maximal cliques are the largest complete subgraph containing a given point. There are several algorithms for finding cliques, including Bron & Kerbosch (1973), Tomita, Tanaka and Takahashi (2006), Cazals and Karande (2008)

Clustering

- ▶ For each node find the fraction of possible triangles that exist, $c_v = \frac{2T(v)}{\deg(v)(\deg(v)-1)}$, where $T(v)$ is the number of triangles through node v .

Transitivity

- ▶ The fraction of all possible triangles which are in fact triangles. Or, $Trans = 3 \left(\frac{T}{t} \right)$, where $T = \#$ of possible triangles and $t = \#$ of actual triads

Often in network analysis we are interested in estimating the cohesiveness of a network, or the communities that exists within the structure

Cliques

- ▶ Maximal cliques are the largest complete subgraph containing a given point. There are several algorithms for finding cliques, including Bron & Kerbosch (1973), Tomita, Tanaka and Takahashi (2006), Cazals and Karande (2008)

Clustering

- ▶ For each node find the fraction of possible triangles that exist, $c_v = \frac{2T(v)}{\deg(v)(\deg(v)-1)}$, where $T(v)$ is the number of triangles through node v .

Transitivity

- ▶ The fraction of all possible triangles which are in fact triangles. Or, $Trans = 3 \left(\frac{T}{t} \right)$, where $T = \#$ of possible triangles and $t = \#$ of actual triads

We will use clustering coefficients to identify community structure in the Hartford drug network

Toy community detection example (not a good one)

Calculating clustering coefficients

```
1 # Calculate clustering coefficients of each node (return as dict)
2 clus=clustering(hartford_mc,with_labels=True)
3 # Get counts of nodes membership for each clustering coefficient, and clean up
4 unique_clus=list(unique(clus.values()))
5 clus_counts=zip(map(lambda c: clus.values().count(c),unique_clus),unique_clus)
6 clus_counts.sort()
7 clus_counts.reverse()
8 # Create a subgraph from nodes with most frequent clustering coefficient
9 mode_clus_sg=subgraph(hartford_mc,[(a) for (a,b) in clus.items() if b==clus_counts[0][1]])
```


Toy community detection example (not a good one)

Calculating clustering coefficients

```
1 # Calculate clustering coefficients of each node (return as dict)
2 clus=clustering(hartford_mc,with_labels=True)
3 # Get counts of nodes membership for each clustering coefficient, and clean up
4 unique_clus=list(unique(clus.values()))
5 clus_counts=zip(map(lambda c: clus.values().count(c),unique_clus),unique_clus)
6 clus_counts.sort()
7 clus_counts.reverse()
8 # Create a subgraph from nodes with most frequent clustering coefficient
9 mode_clus_sg=subgraph(hartford_mc,[(a) for (a,b) in clus.items() if b==clus_counts[0][1]])
```

- Use the `with_labels` to return a dict keyed by node label

Toy community detection example (not a good one)

Calculating clustering coefficients

```
1 # Calculate clustering coefficients of each node (return as dict)
2 clus=clustering(hartford_mc,with_labels=True)
3 # Get counts of nodes membership for each clustering coefficient, and clean up
4 unique_clus=list(unique(clus.values()))
5 clus_counts=zip(map(lambda c: clus.values().count(c),unique_clus),unique_clus)
6 clus_counts.sort()
7 clus_counts.reverse()
8 # Create a subgraph from nodes with most frequent clustering coefficient
9 mode_clus_sg=subgraph(hartford_mc,[(a) for (a,b) in clus.items() if b==clus_counts[0][1]])
```

- ▶ Use the `with_labels` to return a dict keyed by node label
- ▶ The `zip` function takes two lists and returns a tuple

Calculating clustering coefficients

```
1 # Calculate clustering coefficients of each node (return as dict)
2 clus=clustering(hartford_mc,with_labels=True)
3 # Get counts of nodes membership for each clustering coefficient, and clean up
4 unique_clus=list(unique(clus.values()))
5 clus_counts=zip(map(lambda c: clus.values().count(c),unique_clus),unique_clus)
6 clus_counts.sort()
7 clus_counts.reverse()
8 # Create a subgraph from nodes with most frequent clustering coefficient
9 mode_clus_sg=subgraph(hartford_mc,[(a) for (a,b) in clus.items() if b==clus_counts[0][1]])
```

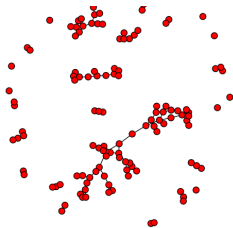
- ▶ Use the `with_labels` to return a dict keyed by node label
- ▶ The `zip` function takes two lists and returns a tuple
- ▶ More complex list comprehension with logic operator

Toy community detection example (not a good one)

Calculating clustering coefficients

```
1 # Calculate clustering coefficients of each node (return as dict)
2 clus=clustering(hartford_mc,with_labels=True)
3 # Get counts of nodes membership for each clustering coefficient, and clean up
4 unique_clus=list(unique(clus.values()))
5 clus_counts=zip(map(lambda c: clus.values().count(c),unique_clus),unique_clus)
6 clus_counts.sort()
7 clus_counts.reverse()
8 # Create a subgraph from nodes with most frequent clustering coefficient
9 mode_clus_sg=subgraph(hartford_mc,[(a) for (a,b) in clus.items() if b==clus_counts[0][1]])
```

- ▶ Use the with_labels to return a dict keyed by node label
- ▶ The zip function takes two lists and returns a tuple
- ▶ More complex list comprehension with logic operator

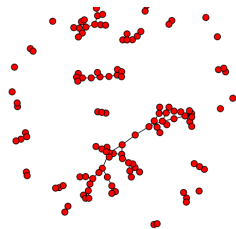


Toy community detection example (not a good one)

Calculating clustering coefficients

```
1 # Calculate clustering coefficients of each node (return as dict)
2 clus=clustering(hartford_mc,with_labels=True)
3 # Get counts of nodes membership for each clustering coefficient, and clean up
4 unique_clus=list(unique(clus.values()))
5 clus_counts=zip(map(lambda c: clus.values().count(c),unique_clus),unique_clus)
6 clus_counts.sort()
7 clus_counts.reverse()
8 # Create a subgraph from nodes with most frequent clustering coefficient
9 mode_clus_sg=subgraph(hartford_mc,[(a) for (a,b) in clus.items() if b==clus_counts[0][1]])
```

- ▶ Use the with_labels to return a dict keyed by node label
- ▶ The zip function takes two lists and returns a tuple
- ▶ More complex list comprehension with logic operator



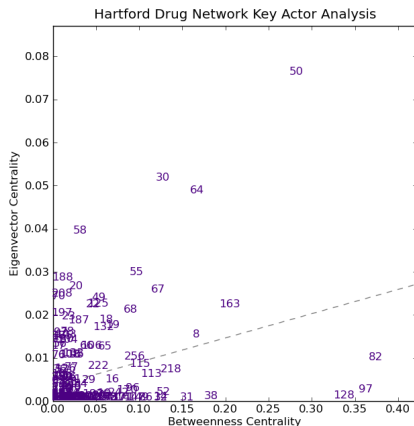
Later, we'll learn how to create a network visualization like the one above

Recall Python's scientific computing trinity: NumPy, SciPy and matplotlib

- ▶ While NumPy and SciPy do most of the behind the scenes work, you will interact with matplotlib frequently for when doing network analysis

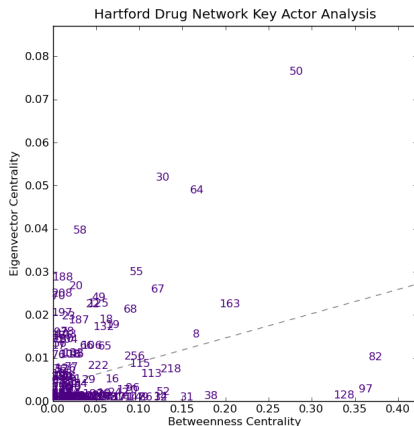
Recall Python's scientific computing trinity: NumPy, SciPy and matplotlib

- ▶ While NumPy and SciPy do most of the behind the scenes work, you will interact with matplotlib frequently for when doing network analysis



Recall Python's scientific computing trinity: NumPy, SciPy and matplotlib

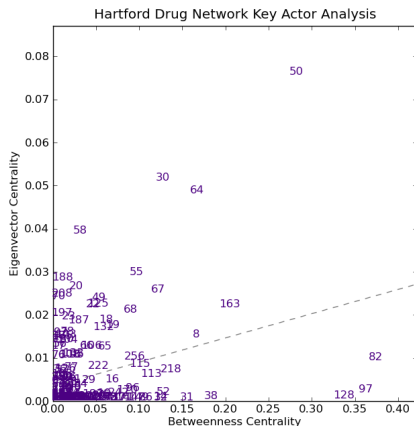
- ▶ While NumPy and SciPy do most of the behind the scenes work, you will interact with matplotlib frequently for when doing network analysis



We will need to create a function that takes two centrality dict and generates this plot

Recall Python's scientific computing trinity: NumPy, SciPy and matplotlib

- ▶ While NumPy and SciPy do most of the behind the scenes work, you will interact with matplotlib frequently for when doing network analysis

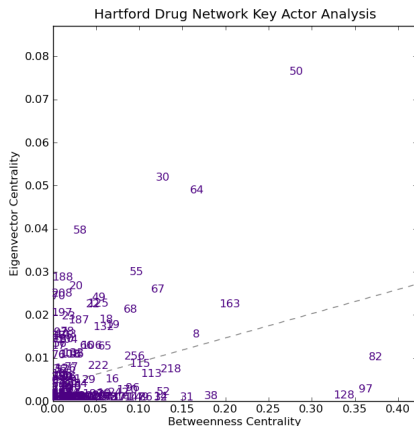


We will need to create a function that takes two centrality dict and generates this plot

1. Create a matplotlib figure

Recall Python's scientific computing trinity: NumPy, SciPy and matplotlib

- ▶ While NumPy and SciPy do most of the behind the scenes work, you will interact with matplotlib frequently for when doing network analysis

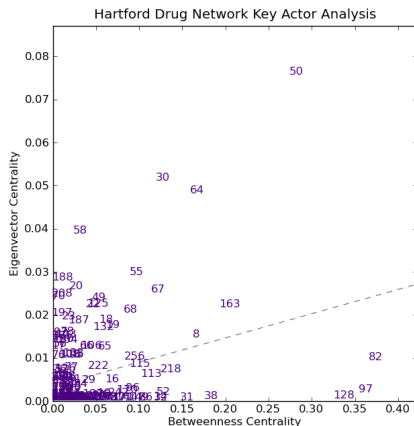


We will need to create a function that takes two centrality dict and generates this plot

1. Create a matplotlib figure
2. Plot each node label as a point

Recall Python's scientific computing trinity: NumPy, SciPy and matplotlib

- ▶ While NumPy and SciPy do most of the behind the scenes work, you will interact with matplotlib frequently for when doing network analysis

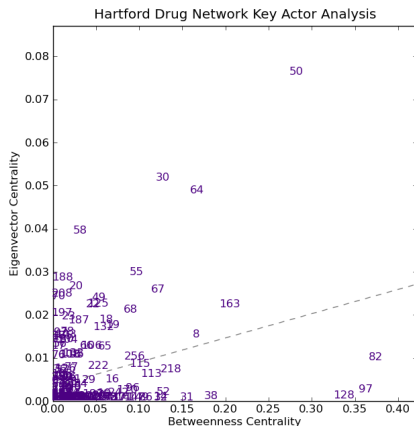


We will need to create a function that takes two centrality dict and generates this plot

1. Create a matplotlib figure
2. Plot each node label as a point
3. Add a "best fit" line

Recall Python's scientific computing trinity: NumPy, SciPy and matplotlib

- ▶ While NumPy and SciPy do most of the behind the scenes work, you will interact with matplotlib frequently for when doing network analysis

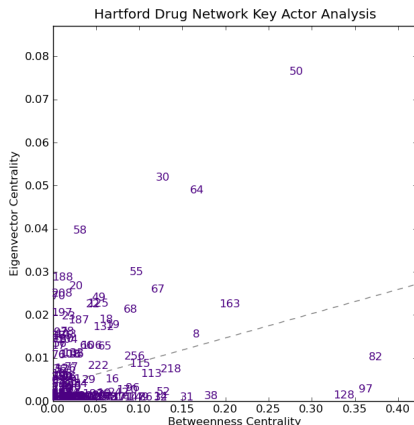


We will need to create a function that takes two centrality dict and generates this plot

1. Create a matplotlib figure
2. Plot each node label as a point
3. Add a "best fit" line
4. Add axis and title labels

Recall Python's scientific computing trinity: NumPy, SciPy and matplotlib

- ▶ While NumPy and SciPy do most of the behind the scenes work, you will interact with matplotlib frequently for when doing network analysis



We will need to create a function that takes two centrality dict and generates this plot

1. Create a matplotlib figure
2. Plot each node label as a point
3. Add a "best fit" line
4. Add axis and title labels
5. Save figure as a PNG file

The centrality_scatter function, part one

```
1 def centrality_scatter(met_dict1, met_dict2, path="", ylab="", xlab="", title="", reg=False):
2     # Create figure and drawing axis
3     fig=P.figure(figsize=(7,7))
4     ax1=fig.add_subplot(111)
5     # Create items so actors can be sorted properly
6     met_items1=met_dict1.items()
7     met_items2=met_dict2.items()
8     met_items1.sort()
9     met_items2.sort()
10    # Grab data
11    xdata=[(b) for (a,b) in met_items1]
12    ydata=[(b) for (a,b) in met_items2]
13    # Add each actor to the plot by ID
14    for p in xrange(len(met_items1)):
15        ax1.text(x=xdata[p], y=ydata[p], s=str(met_items1[p][0]), color="indigo")
```

The centrality_scatter function, part one

```
1 def centrality_scatter(met_dict1, met_dict2, path="", ylab="", xlab="", title="", reg=False):
2     # Create figure and drawing axis
3     fig=P.figure(figsize=(7,7))
4     ax1=fig.add_subplot(111)
5     # Create items so actors can be sorted properly
6     met_items1=met_dict1.items()
7     met_items2=met_dict2.items()
8     met_items1.sort()
9     met_items2.sort()
10    # Grab data
11    xdata=[(b) for (a,b) in met_items1]
12    ydata=[(b) for (a,b) in met_items2]
13    # Add each actor to the plot by ID
14    for p in xrange(len(met_items1)):
15        ax1.text(x=xdata[p], y=ydata[p], s=str(met_items1[p][0]), color="indigo")
```

- Create a canvas to draw on

The centrality_scatter function, part one

```
1 def centrality_scatter(met_dict1, met_dict2, path="", ylab="", xlab="", title="", reg=False):
2     # Create figure and drawing axis
3     fig=P.figure(figsize=(7,7))
4     ax1=fig.add_subplot(111)
5     # Create items so actors can be sorted properly
6     met_items1=met_dict1.items()
7     met_items2=met_dict2.items()
8     met_items1.sort()
9     met_items2.sort()
10    # Grab data
11    xdata=[(b) for (a,b) in met_items1]
12    ydata=[(b) for (a,b) in met_items2]
13    # Add each actor to the plot by ID
14    for p in xrange(len(met_items1)):
15        ax1.text(x=xdata[p], y=ydata[p], s=str(met_items1[p][0]), color="indigo")
```

- ▶ Create a canvas to draw on
- ▶ manipulate and store centrality data

The centrality_scatter function, part one

```
1 def centrality_scatter(met_dict1, met_dict2, path="", ylab="", xlab="", title="", reg=False):
2     # Create figure and drawing axis
3     fig=P.figure(figsize=(7,7))
4     ax1=fig.add_subplot(111)
5     # Create items so actors can be sorted properly
6     met_items1=met_dict1.items()
7     met_items2=met_dict2.items()
8     met_items1.sort()
9     met_items2.sort()
10    # Grab data
11    xdata=[(b) for (a,b) in met_items1]
12    ydata=[(b) for (a,b) in met_items2]
13    # Add each actor to the plot by ID
14    for p in xrange(len(met_items1)):
15        ax1.text(x=xdata[p], y=ydata[p], s=str(met_items1[p][0]), color="indigo")
```

- ▶ Create a canvas to draw on
- ▶ manipulate and store centrality data
- ▶ Add points to plot as node labels

The centrality_scatter function, part two

```
1 def centrality_scatter(met_dict1, met_dict2, path="", ylab="", xlab="", title="", reg=False):
2     ...
3     # If adding a best fit line, we will use NumPy to calculate the points.
4     if reg:
5         # Function returns y-intercept and slope. So, we create a function to
6         # draw LOBF from this data
7         slope, yint = polyfit(xdata, ydata, 1)
8         xline = P.xticks()[0]
9         yline = map(lambda x: slope*x + yint, xline)
10        # Add line
11        ax1.plot(xline, yline, ls='—', color='grey')
12        # Set new x- and y-axis limits to data
13        P.xlim((0.0, max(xdata) + (.15 * max(xdata)))) # Give a little buffer
14        P.ylim((0.0, max(ydata) + (.15 * max(ydata))))
15        # Add labels
16        ax1.set_title(title)
17        ax1.set_xlabel(xlab)
18        ax1.set_ylabel(ylab)
19        # Save figure
20        P.savefig(path, dpi=100)
```

The centrality_scatter function, part two

```
1 def centrality_scatter(met_dict1, met_dict2, path="", ylab="", xlab="", title="", reg=False):
2     ...
3     # If adding a best fit line, we will use NumPy to calculate the points.
4     if reg:
5         # Function returns y-intercept and slope. So, we create a function to
6         # draw LOBF from this data
7         slope, yint = polyfit(xdata, ydata, 1)
8         xline = P.xticks()[0]
9         yline = map(lambda x: slope*x + yint, xline)
10        # Add line
11        ax1.plot(xline, yline, ls='—', color='grey')
12        # Set new x- and y-axis limits to data
13        P.xlim((0.0, max(xdata) + (.15 * max(xdata)))) # Give a little buffer
14        P.ylim((0.0, max(ydata) + (.15 * max(ydata))))
15        # Add labels
16        ax1.set_title(title)
17        ax1.set_xlabel(xlab)
18        ax1.set_ylabel(ylab)
19        # Save figure
20        P.savefig(path, dpi=100)
```

- Add a best fit line

The centrality_scatter function, part two

```
1 def centrality_scatter(met_dict1, met_dict2, path="", ylab="", xlab="", title="", reg=False):
2     ...
3     # If adding a best fit line, we will use NumPy to calculate the points.
4     if reg:
5         # Function returns y-intercept and slope. So, we create a function to
6         # draw LOBF from this data
7         slope, yint = polyfit(xdata, ydata, 1)
8         xline = P.xticks()[0]
9         yline = map(lambda x: slope*x + yint, xline)
10        # Add line
11        ax1.plot(xline, yline, ls='—', color='grey')
12        # Set new x- and y-axis limits to data
13        P.xlim((0.0, max(xdata) + (.15 * max(xdata)))) # Give a little buffer
14        P.ylim((0.0, max(ydata) + (.15 * max(ydata))))
15        # Add labels
16        ax1.set_title(title)
17        ax1.set_xlabel(xlab)
18        ax1.set_ylabel(ylab)
19        # Save figure
20        P.savefig(path, dpi=100)
```

- ▶ Add a best fit line
- ▶ Resize figure to fit data

The centrality_scatter function, part two

```
1 def centrality_scatter(met_dict1, met_dict2, path="", ylab="", xlab="", title="", reg=False):
2     ...
3     # If adding a best fit line, we will use NumPy to calculate the points.
4     if reg:
5         # Function returns y-intercept and slope. So, we create a function to
6         # draw LOBF from this data
7         slope, yint = polyfit(xdata, ydata, 1)
8         xline = P.xticks()[0]
9         yline = map(lambda x: slope*x + yint, xline)
10        # Add line
11        ax1.plot(xline, yline, ls='—', color='grey')
12        # Set new x- and y-axis limits to data
13        P.xlim((0.0, max(xdata) + (.15 * max(xdata)))) # Give a little buffer
14        P.ylim((0.0, max(ydata) + (.15 * max(ydata))))
15        # Add labels
16        ax1.set_title(title)
17        ax1.set_xlabel(xlab)
18        ax1.set_ylabel(ylab)
19        # Save figure
20        P.savefig(path, dpi=100)
```

- ▶ Add a best fit line
- ▶ Resize figure to fit data
- ▶ Add labels, and save the figure as a PNG file

As powerful as NetworkX and the complementing scientific computing packages in Python are, it may often be useful or necessary to output your data for additional analysis

As powerful as NetworkX and the complementing scientific computing packages in Python are, it may often be useful or necessary to output your data for additional analysis

- ▶ Suite of tools lacks your specific need

As powerful as NetworkX and the complementing scientific computing packages in Python are, it may often be useful or necessary to output your data for additional analysis

- ▶ Suite of tools lacks your specific need
- ▶ Require alternate visualization

As powerful as NetworkX and the complementing scientific computing packages in Python are, it may often be useful or necessary to output your data for additional analysis

- ▶ Suite of tools lacks your specific need
- ▶ Require alternate visualization
- ▶ Storage for later analysis

As powerful as NetworkX and the complementing scientific computing packages in Python are, it may often be useful or necessary to output your data for additional analysis

- ▶ Suite of tools lacks your specific need
- ▶ Require alternate visualization
- ▶ Storage for later analysis

In most cases this will entail either exporting the raw network data, or metrics from some network analysis

As powerful as NetworkX and the complementing scientific computing packages in Python are, it may often be useful or necessary to output your data for additional analysis

- ▶ Suite of tools lacks your specific need
- ▶ Require alternate visualization
- ▶ Storage for later analysis

In most cases this will entail either exporting the raw network data, or metrics from some network analysis

1. NetworkX can write out network data in as many formats as it can read them, and the process is equally straightforward

As powerful as NetworkX and the complementing scientific computing packages in Python are, it may often be useful or necessary to output your data for additional analysis

- ▶ Suite of tools lacks your specific need
- ▶ Require alternate visualization
- ▶ Storage for later analysis

In most cases this will entail either exporting the raw network data, or metrics from some network analysis

1. NetworkX can write out network data in as many formats as it can read them, and the process is equally straightforward
2. When you want to export metrics we can use Python's built-in XML and CSV libraries

As powerful as NetworkX and the complementing scientific computing packages in Python are, it may often be useful or necessary to output your data for additional analysis

- ▶ Suite of tools lacks your specific need
- ▶ Require alternate visualization
- ▶ Storage for later analysis

In most cases this will entail either exporting the raw network data, or metrics from some network analysis

1. NetworkX can write out network data in as many formats as it can read them, and the process is equally straightforward
2. When you want to export metrics we can use Python's built-in XML and CSV libraries
3. Depending on your needs you may prefer one, the other or both

As powerful as NetworkX and the complementing scientific computing packages in Python are, it may often be useful or necessary to output your data for additional analysis

- ▶ Suite of tools lacks your specific need
- ▶ Require alternate visualization
- ▶ Storage for later analysis

In most cases this will entail either exporting the raw network data, or metrics from some network analysis

1. NetworkX can write out network data in as many formats as it can read them, and the process is equally straightforward
2. When you want to export metrics we can use Python's built-in XML and CSV libraries
3. Depending on your needs you may prefer one, the other or both

Next, we will review how to save data in different formats and export metrics to a CSV file using the Hartford drug net data

The syntax for exporting network data follows exactly the syntax for loading it

NX syntax for writing a network file

```
>>> nx.write_format(G, "path/to/file.txt", ...options...)
      ↑           ↑           ↑
      NX function, net variable  File to be written  Nodes/edge data, etc.
```

The syntax for exporting network data follows exactly the syntax for loading it

NX syntax for writing a network file

```
>>> nx.write_format(G, "path/to/file.txt", ...options...)
      ↑                ↑                ↑
      NX function, net variable  File to be written  Nodes/edge data, etc.
```


The syntax for exporting network data follows exactly the syntax for loading it

NX syntax for writing a network file

```
>>> nx.write_format(G, "path/to/file.txt", ...options...)
      ↑           ↑           ↑
      NX function, net variable  File to be written  Nodes/edge data, etc.
```

The syntax for exporting network data follows exactly the syntax for loading it

NX syntax for writing a network file

```
>>> nx.write_format(G, "path/to/file.txt", ...options...)
      ↑           ↑           ↑
      NX function, net variable  File to be written  Nodes/edge data, etc.
```

The syntax for exporting network data follows exactly the syntax for loading it

NX syntax for writing a network file

```
>>> nx.write_format(G, "path/to/file.txt", ...options...)
      ↑           ↑           ↑
      NX function, net variable  File to be written  Nodes/edge data, etc.
```

Let's try!

- ▶ Output the Hartford drug net data as an adjacency list
- ▶ Add metric data to each node of the network
- ▶ Output new network in Pajek format with node attributes

As shown, this is a simple one line operation

Output Hartford drug net data as an adjacency list

```
1 nx.write_adjlist(hartford_mc, "../../data/hartford_mc_adj.txt")
```

Next, we will add the Eigenvector centrality of each node to the graph object

Adding node attributes

```
1 def add_metric(G, met_dict):
2     """Adds metric data to G from a dictionary keyed by node labels"""
3     if G.nodes().sort()==met_dict.keys().sort():
4         for i in met_dict.keys():
5             G.add_node(i, metric=met_dict[i])
6         return G
7     else:
8         raise ValueError("Node labels do not match")
```

As shown, this is a simple one line operation

Output Hartford drug net data as an adjacency list

```
1 nx.write_adjlist(hartford_mc, "../../data/hartford_mc_adj.txt")
```

Next, we will add the Eigenvector centrality of each node to the graph object

Adding node attributes

```
1 def add_metric(G, met_dict):
2     """Adds metric data to G from a dictionary keyed by node labels"""
3     if G.nodes().sort()==met_dict.keys().sort():
4         for i in met_dict.keys():
5             G.add_node(i, metric=met_dict[i])
6         return G
7     else:
8         raise ValueError("Node labels do not match")
```

- Quick error checking

Saving network data and adding node attributes

As shown, this is a simple one line operation

Output Hartford drug net data as an adjacency list

```
1 nx.write_adjlist(hartford_mc, "../../data/hartford_mc_adj.txt")
```

Next, we will add the Eigenvector centrality of each node to the graph object

Adding node attributes

```
1 def add_metric(G, met_dict):
2     """Adds metric data to G from a dictionary keyed by node labels"""
3     if (G.nodes().sort()==met_dict.keys().sort()):
4         for i in met_dict.keys():
5             G.add_node(i, metric=met_dict[i])
6         return G
7     else:
8         raise ValueError("Node labels do not match")
```

- ▶ Quick error checking
- ▶ Add node attribute as “metric”

Python has powerful built-in tools for reading and writing standard data formats

- ▶ One of the most useful, and frequently used, is the CSV library and the DictWriter

```
1 import csv
2 ...
3 def csv_exporter(data_dict,path):
4     """Takes a dict of centralities keyed by column headers and exports
5     data as a CSV file"""
6     # Create column header list
7     col_headers=["Actor"]
8     col_headers.extend(data_dict.keys())
9     # Create CSV writer and write column headers
10    writer=csv.DictWriter(open(path,"w"),fieldnames=col_headers)
11    writer.writerow(dict((h,h) for h in col_headers))
12    # Write each row of data
13    for j in data_dict[col_headers[1]].keys():
14        # Create a new dict for each row
15        row=dict.fromkeys(col_headers)
16        row["Actor"]=j
17        for k in data_dict.keys():
18            row[k]=data_dict[k][j]
19        writer.writerow(row)
```

We can now open the CSV file in our favorite spreadsheet program

- ▶ Perform traditional data exploration
- ▶ Load into other analytics platforms for additional analysis (e.g., R)
- ▶ Store for latter use

	A	B	C	D
1	Actor	Closeness	Betweenness	Eigenvector
2	1	0.12467532	0.0072576	0.00025176
3	2	0.12475634	0.01767427	0.00025964
4	3	0.12565445	0.05687441	0.00023185
5	4	0.10223642	0.03108639	1.44E-05
6	5	0.1443609	0	0.00313152
7	6	0.09943035	0.01041667	1.49E-07
8	7	0.11340815	0.04362093	6.78E-05
9	8	0.20512821	0.16354003	0.01471888
10	9	0.11267606	0.00741624	0.0001101
11	10	0.13983977	0.05258239	0.00095456
12	11	0.1703638	0.01250999	0.0032333
13	13	0.13892909	0	1.79E-05
14	14	0.17219731	0.11848775	0.00029737
15	15	0.13521127	0.00079897	2.11E-05
16	16	0.15907208	0.06203647	0.00432838

What makes a good network visualization technique

Development of visualization techniques and algorithms has become somewhat of a cottage industry

What makes a good network visualization technique

Development of visualization techniques and algorithms has become somewhat of a cottage industry

- ▶ Maximize “visibility” of network
- ▶ Scale up to very large graphs
- ▶ Display nodal- (centrality) of network-level (community structure) information

What makes a good network visualization technique

Development of visualization techniques and algorithms has become somewhat of a cottage industry

- ▶ Maximize “visibility” of network
- ▶ Scale up to very large graphs
- ▶ Display nodal- (centrality) of network-level (community structure) information

NetworkX was designed as a data manipulation and analysis tool, and therefore is not meant as a visualization platform

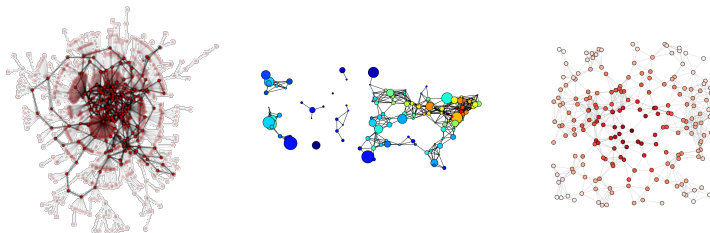
What makes a good network visualization technique

Development of visualization techniques and algorithms has become somewhat of a cottage industry

- ▶ Maximize “visibility” of network
- ▶ Scale up to very large graphs
- ▶ Display nodal- (centrality) of network-level (community structure) information

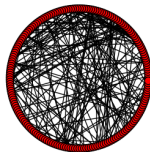
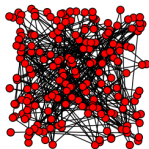
NetworkX was designed as a data manipulation and analysis tool, and therefore is not meant as a visualization platform

- ▶ It is, however, still capable of making very nice visualization



The most basic visualization techniques are the random and circular layouts

- ▶ The random layout places nodes in...random positions
- ▶ The circular layout places nodes in...a circle

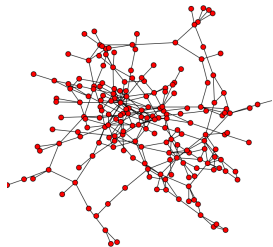


```
1 # Use subplots to draw random and circular layouts
2 # of drug net side-by-side
3 fig1=P.figure(figsize=(9,4))
4 fig1.add_subplot(121)
5 nx.draw_random(hartford_mc, with_labels=False, node_size=60)
6 fig1.add_subplot(122)
7 nx.draw_circular(hartford_mc, with_labels=False, node_size=60)
8 P.savefig("../images/networks/rand_circ.png")
```

More commonly used visualization techniques include the spring and spectral layouts

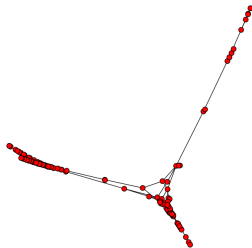
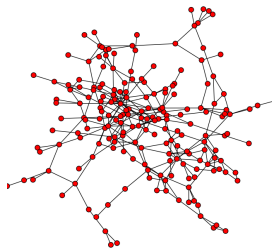
More commonly used visualization techniques include the spring and spectral layouts

- ▶ The spring layout is a version of the Fruchterman-Reingold force-directed algorithm, which attempts to minimize overlapping edges



More commonly used visualization techniques include the spring and spectral layouts

- ▶ The spring layout is a version of the Fruchterman-Reingold force-directed algorithm, which attempts to minimize overlapping edges
- ▶ The spectral layout finds node position using the eigenvectors of the graph Laplacian, which is useful for quickly visualizing structural clustering



The shell layout draws nodes as concentric circles

- ▶ Two dimensional extension of the circle layout
- ▶ We may have some reason to isolate certain nodes

25th percentile Eigenvector centrality actors

```
1 P.figure(figsize=(8,8))
2 # Find actors in 25th percentile
3 max_eig=max([(b) for (a,b) in eig_cen.items()])
4 s1=[(a) for (a,b) in eig_cen.items() if b>=.25*max_eig]
5 s2=hartford_mc.nodes()
6 # setdiff1d is a very useful NumPy function!
7 s2=list(setdiff1d(s2,s1))
8 shells=[s1,s2]
9 # Calculate position and draw
10 shell_pos=shell_layout(hartford_mc, shells)
11 draw_networkx(hartford_mc, shell_pos, with_labels=False, node_size=60)
12 P.savefig("../images/networks/shell.png")
```

The shell layout draws nodes as concentric circles

- ▶ Two dimensional extension of the circle layout
- ▶ We may have some reason to isolate certain nodes

25th percentile Eigenvector centrality actors

```
1 P.figure(figsize=(8,8))
2 # Find actors in 25th percentile
3 max_eig=max([(b) for (a,b) in eig_cen.items()])
4 s1=[(a) for (a,b) in eig_cen.items() if b>=.25*max_eig]
5 s2=hartford_mc.nodes()
6 # setdiff1d is a very useful NumPy function!
7 s2=list(setdiff1d(s2,s1))
8 shells=[s1,s2]
9 # Calculate position and draw
10 shell_pos=shell_layout(hartford_mc, shells)
11 draw_networkx(hartford_mc, shell_pos, with_labels=False, node_size=60)
12 P.savefig("../images/networks/shell.png")
```

The shell layout draws nodes as concentric circles

- ▶ Two dimensional extension of the circle layout
- ▶ We may have some reason to isolate certain nodes

25th percentile Eigenvector centrality actors

```
1 P.figure(figsize=(8,8))
2 # Find actors in 25th percentile
3 max_eig=max([(b) for (a,b) in eig_cen.items()])
4 s1=[(a) for (a,b) in eig_cen.items() if b>=.25*max_eig]
5 s2=hartford_mc.nodes()
6 # setdiff1d is a very useful NumPy function!
7 s2=list(setdiff1d(s2,s1))
8 shells=[s1,s2]
9 # Calculate position and draw
10 shell_pos=shell_layout(hartford_mc,shells)
11 draw_networkx(hartford_mc,shell_pos,with_labels=False,node_size=60)
12 P.savefig("../images/networks/shell.png")
```

The shell layout draws nodes as concentric circles

- ▶ Two dimensional extension of the circle layout
- ▶ We may have some reason to isolate certain nodes

25th percentile Eigenvector centrality actors

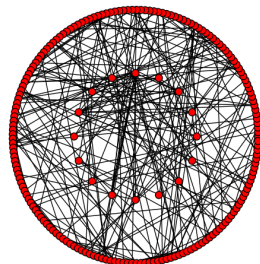
```
1 P.figure(figsize=(8,8))
2 # Find actors in 25th percentile
3 max_eig=max([(b) for (a,b) in eig_cen.items()])
4 s1=[(a) for (a,b) in eig_cen.items() if b>=.25*max_eig]
5 s2=hartford_mc.nodes()
6 # setdiff1d is a very useful NumPy function!
7 s2=list(setdiff1d(s2,s1))
8 shells=[s1,s2]
9 # Calculate position and draw
10 shell_pos=shell_layout(hartford_mc,shells)
11 draw_networkx(hartford_mc,shell_pos,with_labels=False,node_size=60)
12 P.savefig("../images/networks/shell.png")
```

The shell layout draws nodes as concentric circles

- ▶ Two dimensional extension of the circle layout
- ▶ We may have some reason to isolate certain nodes

25th percentile Eigenvector centrality actors

```
1 P.figure(figsize=(8,8))
2 # Find actors in 25th percentile
3 max_eig=max([(b) for (a,b) in eig_cen.items()])
4 s1=[(a) for (a,b) in eig_cen.items() if b>=.25*max_eig]
5 s2=hartford_mc.nodes()
6 # setdiff1d is a very useful NumPy function!
7 s2=list(setdiff1d(s2,s1))
8 shells=[s1,s2]
9 # Calculate position and draw
10 shell_pos=shell_layout(hartford_mc, shells)
11 draw_networkx(hartford_mc, shell_pos, with_labels=False, node_size=60)
12 P.savefig("../images/networks/shell.png")
```

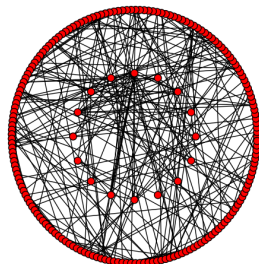


The shell layout draws nodes as concentric circles

- ▶ Two dimensional extension of the circle layout
- ▶ We may have some reason to isolate certain nodes

25th percentile Eigenvector centrality actors

```
1 P.figure(figsize=(8,8))
2 # Find actors in 25th percentile
3 max_eig=max([(b) for (a,b) in eig_cen.items()])
4 s1=[(a) for (a,b) in eig_cen.items() if b>=.25*max_eig]
5 s2=hartford_mc.nodes()
6 # setdiff1d is a very useful NumPy function!
7 s2=list(setdiff1d(s2,s1))
8 shells=[s1,s2]
9 # Calculate position and draw
10 shell_pos=shell_layout(hartford_mc, shells)
11 draw_networkx(hartford_mc, shell_pos, with_labels=False, node_size=60)
12 P.savefig("../images/networks/shell.png")
```



Beyond layout, we may also want to add analytical data to our visualization

NetworkX allows you to alter the size, color and shape of the nodes and edges in any visualization

NetworkX allows you to alter the size, color and shape of the nodes and edges in any visualization

- ▶ This can be particularly useful if we want to make some actors more prominent than others

NetworkX allows you to alter the size, color and shape of the nodes and edges in any visualization

- ▶ This can be particularly useful if we want to make some actors more prominent than others

In our final exercise, we will add the following analysis to the Hartford drug network

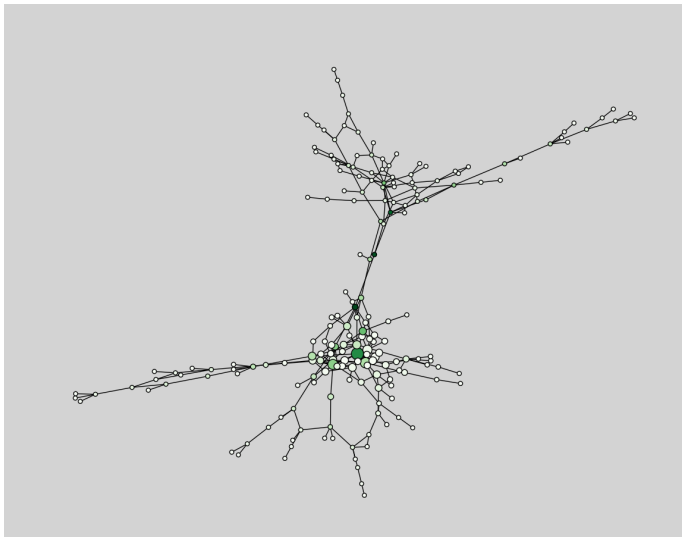
- ▶ Node size by Eigenvector centrality
- ▶ Intensity of node color by betweenness centrality
- ▶ Edge thickness by edge betweenness

The code to add analysis to visualization

More list comprehension and matplotlib colormaps

```
1 # Adding analysis to visualization
2 P.figure(figsize=(15,15))
3 P.subplot(111,axisbg="lightgrey")
4 spring_pos=nx.spring_layout(hartford_mc,iterations=1000)
5 # Use betweenness centrality for node color intensity
6 bet_color=bet_cen.items()
7 bet_color.sort()
8 bet_color=[(b) for (a,b) in bet_color]
9 # Use Eigenvector centrality to set node size
10 eig_size=eig_cen.items()
11 eig_size.sort()
12 eig_size=[((b)*2000)+20 for (a,b) in eig_size]
13 # Use matplotlib's colormap for node intensity
14 draw_networkx(hartford_mc,spring_pos,node_color=bet_color,...
15               ...cmap=P.cm.Greens,node_size=eig_size,with_labels=False)
16 P.savefig("../images/networks/analysis.png")
```

Final visualization



Basic Analysis

Basic Analysis

- ▶ How to load local data, and an example of building networks from data streamed directly from the Internet

Basic Analysis

- ▶ How to load local data, and an example of building networks from data streamed directly from the Internet
- ▶ A brief review of the Python dict data type

Basic Analysis

- ▶ How to load local data, and an example of building networks from data streamed directly from the Internet
- ▶ A brief review of the Python `dict` data type
- ▶ Calculating basic metrics, how they are stored in NetworkX and how to manipulate them (list comps!)

Basic Analysis

- ▶ How to load local data, and an example of building networks from data streamed directly from the Internet
- ▶ A brief review of the Python dict data type
- ▶ Calculating basic metrics, how they are stored in NetworkX and how to manipulate them (list comps!)
- ▶ How to use matplotlib to visualize our analysis

Basic Analysis

- ▶ How to load local data, and an example of building networks from data streamed directly from the Internet
- ▶ A brief review of the Python dict data type
- ▶ Calculating basic metrics, how they are stored in NetworkX and how to manipulate them (list comps!)
- ▶ How to use matplotlib to visualize our analysis
- ▶ Getting data out of NetworkX both as raw network data or analytics using the CSV library

Basic Analysis

- ▶ How to load local data, and an example of building networks from data streamed directly from the Internet
- ▶ A brief review of the Python dict data type
- ▶ Calculating basic metrics, how they are stored in NetworkX and how to manipulate them (list comps!)
- ▶ How to use matplotlib to visualize our analysis
- ▶ Getting data out of NetworkX both as raw network data or analytics using the CSV library
- ▶ Network visualization techniques in NetworkX and how to add network analysis to a visualization

Basic Analysis

- ▶ How to load local data, and an example of building networks from data streamed directly from the Internet
- ▶ A brief review of the Python dict data type
- ▶ Calculating basic metrics, how they are stored in NetworkX and how to manipulate them (list comps!)
- ▶ How to use matplotlib to visualize our analysis
- ▶ Getting data out of NetworkX both as raw network data or analytics using the CSV library
- ▶ Network visualization techniques in NetworkX and how to add network analysis to a visualization

Questions?