Module IV - Basic Analysis

Drew Conway — Department of Politics

NEW YORK UNIVERSITY

June 29, 2010

Agenda for Module IV

Loading data from multiple sources

- Local network data files
- Connecting to a database
- Building directly from the Internet

Agenda for Module IV

Loading data from multiple sources

- Local network data files
- Connecting to a database
- Building directly from the Internet

Brief review of Python dictionaries

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

Agenda for Module IV

Loading data from multiple sources

- Local network data files
- Connecting to a database
- 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, Betweeness Eigenvector
- Calculating degree distribution
- Plotting statistics using `matplotlib`
- Calculating cliques, clustering and transitivity

Agenda for Module IV

Loading data from multiple sources

- Local network data files
- Connecting to a database
- 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, Betweeness 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

Agenda for Module IV

Loading data from multiple sources

- Local network data files
- Connecting to a database
- 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, Betweeness 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

Loading a network file

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

$$\underset{\text{Net variable}}{\underset{\uparrow}{>>> G}} \quad = \quad \underset{\text{NX function, file directory path}}{\underset{\uparrow}{\text{read\_format(“path/to/file.txt”,}}} \quad \underset{\text{Graph type, nodes type, etc.}}{\underset{\uparrow}{\textit{...options...}\text{)}}}$$

Loading a network file

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

$$\underset{\text{Net variable}}{\underset{\uparrow}{>>> G}} \quad = \quad \underset{\text{NX function, file directory path}}{\underset{\uparrow}{\text{read\_format(“path/to/file.txt”,}}} \qquad \underset{\text{Graph type, nodes type, etc.}}{\underset{\uparrow}{\ldots options\ldots}})$$

Loading a network file

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

$$\underset{\text{Net variable}}{\underset{\uparrow}{>>> G}} \quad = \quad \underset{\text{NX function, file directory path}}{\underset{\uparrow}{\text{read\_format(“path/to/file.txt”},}} \quad \underset{\text{Graph type, nodes type, etc.}}{\underset{\uparrow}{\textit{...options...})}}$$

Loading a network file

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

$$\underset{\text{Net variable}}{\underset{\uparrow}{>>> G}} \quad = \quad \underset{\text{NX function, file directory path}}{\underset{\uparrow}{\text{read\_format(``path/to/file.txt'',}}} \qquad \underset{\text{Graph type, nodes type, etc.}}{\underset{\uparrow}{\textit{...options...}}})$$

Loading a network file

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

$$\underset{\text{Net variable}}{\underset{\uparrow}{>>> G}} \quad = \quad \underset{\text{NX function, file directory path}}{\underset{\uparrow}{\text{read\_format(“path/to/file.txt”,}}} \quad \underset{\text{Graph type, nodes type, etc.}}{\underset{\uparrow}{\textit{...options...}\text{)}}}$$

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 `info()` to check that data has been loaded correctly

Loading a network file

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

$$\underset{\text{Net variable}}{\underset{\uparrow}{>>> G}} \quad = \quad \underset{\text{NX function, file directory path}}{\underset{\uparrow}{\text{read\_format(“path/to/file.txt”,}}} \qquad \underset{\text{Graph type, nodes type, etc.}}{\underset{\uparrow}{\textit{...options...})}}$$

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 `info()` to check that data has been loaded correctly

It's time to fire up your console and load Python!

Loading the Hartford drug users network

Starting NetworkX and loading data

```
>>> from networkx import *
>>> hartford=read_edgelist("../../data/hartford_drug.txt",create_using=DiGraph(),nodetype=int)
>>> info(hartford)
Name:
Type:                   DiGraph
Number of nodes:        212
Number of edges:        337
Average in degree:      1.5896
Average out degree:     1.5896
```

Loading the Hartford drug users network

Starting NetworkX and loading data

```
>>> from networkx import *
>>> hartford=read_edgelist("../../data/hartford_drug.txt",create_using=DiGraph(),nodetype=int)
>>> info(hartford)
Name:
Type:                   DiGraph
Number of nodes:        212
Number of edges:        337
Average in degree:      1.5896
Average out degree:     1.5896
```

What did we just do?

Loading the Hartford drug users network

Starting NetworkX and loading data

```
>>> from networkx import *
>>> hartford=read_edgelist("../../data/hartford_drug.txt",create_using=DiGraph(),nodetype=int)
>>> info(hartford)
Name:
Type:                   DiGraph
Number of nodes:        212
Number of edges:        337
Average in degree:      1.5896
Average out degree:     1.5896
```

What did we just do?

- Used the read_edgelist function to load EL file

Loading the Hartford drug users network

Starting NetworkX and loading data

```
>>> from networkx import *
>>> hartford=read_edgelist("../../data/hartford_drug.txt",create_using=DiGraph(),nodetype=int)
>>> info(hartford)
Name:
Type:                   DiGraph
Number of nodes:        212
Number of edges:        337
Average in degree:      1.5896
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

Loading the Hartford drug users network

Starting NetworkX and loading data

```
>>> from networkx import *
>>> hartford=read_edgelist("../../data/hartford_drug.txt",create_using=DiGraph(),nodetype=int)
>>> info(hartford)
Name:
Type:                   DiGraph
Number of nodes:        212
Number of edges:        337
Average in degree:      1.5896
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

Loading the Hartford drug users network

Starting NetworkX and loading data

```
>>> from networkx import *
>>> hartford=read_edgelist("../../data/hartford_drug.txt",create_using=DiGraph(),nodetype=int)
>>> info(hartford)
Name:
Type:                   DiGraph
Number of nodes:        212
Number of edges:        337
Average in degree:      1.5896
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

Loading the Hartford drug users network

Starting NetworkX and loading data

```
>>> from networkx import *
>>> hartford=read_edgelist("../../data/hartford_drug.txt",create_using=DiGraph(),nodetype=int)
>>> info(hartford)
Name:
Type:                   DiGraph
Number of nodes:        212
Number of edges:        337
Average in degree:      1.5896
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

Loading the Hartford drug users network

Starting NetworkX and loading data

```
>>> from networkx import *
>>> hartford=read_edgelist("../../data/hartford_drug.txt",create_using=DiGraph(),nodetype=int)
>>> info(hartford)
Name:
Type:                  DiGraph
Number of nodes:       212
Number of edges:       337
Average in degree:     1.5896
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

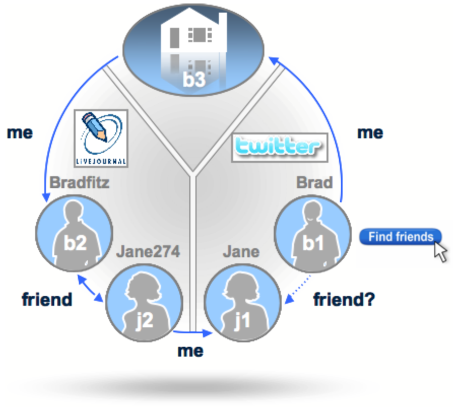
Some formats may have more or less options, **always check the documentations!**

Building a network from a database

As data sets become larger and persistently changing, it may make more sense to store them in a database rather than a single file

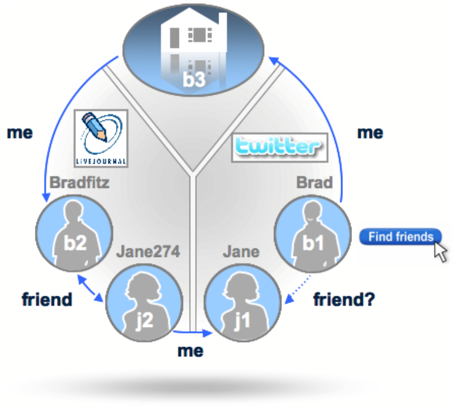
- As we have seen, Python provides binding to many modern database frameworks

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

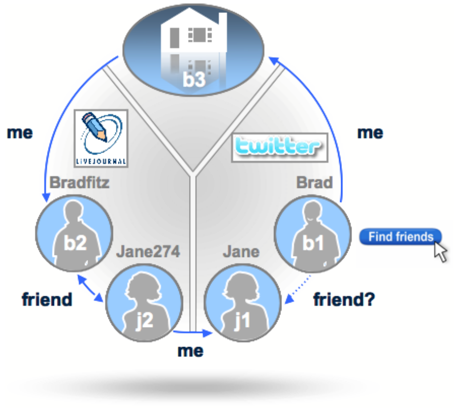
Building the social network among LiveJournal users



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

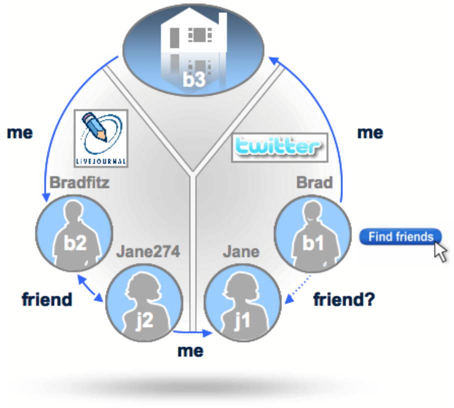
Building the social network among LiveJournal users



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

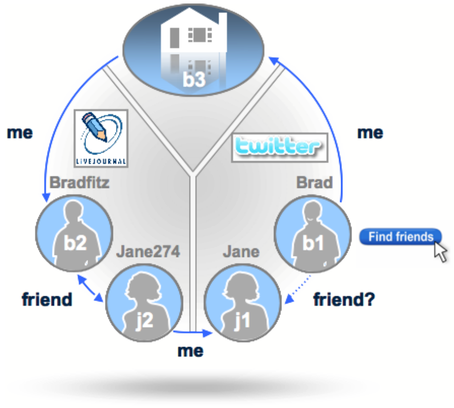
Building the social network among LiveJournal users



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" $seed \longrightarrow friend \longrightarrow \cdots \longrightarrow friend_k$

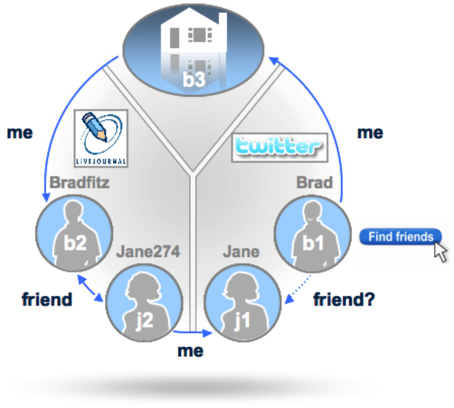
Building the social network among LiveJournal users



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" $seed \longrightarrow friend \longrightarrow \cdots \longrightarrow friend_k$
  - 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* $\longrightarrow$ *friend* $\longrightarrow \cdots \longrightarrow$ $friend_k$
  - Seed: imichaeldotorg.livejournal.com
  - $k = 3$
- Note the low value of k

The code, part 1

Loading the libraries and setting things up

```
from cjson import *
from urllib import *
from networkx import *
from time import *
from scipy import array,unique
...
if __name__ == "__main__":
    seed_url=‘‘http://imichaeldotorg.livejournal.com"
    sg=get_sg(seed_url)
    net,newnodes=create_egonet(sg)
    info(net)
```

Get the JSON from SocialGraph

```
def get_sg(seed_url):
  sgapi_url="http://socialgraph.apis.google.com/lookup?q="+seed_url+"&edo=1&edi=1&fme=1&pretty=0"
  try:
      furl=urlopen(sgapi_url)
      fr=furl.read()
      furl.close()
      return fr
  except IOError:
      print "Could not connect to website"
      print sgapi_url
      return
```

The code, part 1

Loading the libraries and setting things up

```
from cjson import *
from urllib import *
from networkx import *
from time import *
from scipy import array,unique
...
if __name__ == "__main__":
    seed_url=‘‘http://imichaeldotorg.livejournal.com"
    sg=get_sg(seed_url)
    net,newnodes=create_egonet(sg)
    info(net)
```

```
Name:                  [‘http://imichaeldotorg.livejournal.com/’]
Type:                  DiGraph
Number of nodes:       5
Number of edges:       5
Average in degree:     1.0
Average out degree:    1.0
```

Get the JSON from SocialGraph

```
def get_sg(seed_url):
  sgapi_url="http://socialgraph.apis.google.com/lookup?q="+seed_url+"&edo=1&edi=1&fme=1&pretty=0"
  try:
      furl=urlopen(sgapi_url)
      fr=furl.read()
      furl.close()
      return fr
  except IOError:
      print "Could not connect to website"
      print sgapi_url
      return
```

Build egonet and snowball

Creating the egonet

```
def create_egonet(s):
    try:
        raw=decode(s)
        G=DiGraph()
        pendants=[]
        n=raw['nodes']
        nk=n.keys()
        G.name=str(nk)
        pendants=[]
        for a in range(0,len(nk)):
            for b in range(0,len(nk)):
                if a!=b:
                    G.add_edge(nk[a],nk[b])
        for k in nk:
            ego=n[k]
            ego_out=ego['nodes_referenced']
            for o in ego_out:
                G.add_edge(k,o)
                pendants.append(o)
            ego_in=ego['nodes_referenced_by']
            for i in ego_in:
                G.add_edge(i,k)
                pendants.append(i)
        pendants=array(pendants,dtype=str)
        pendants.flatten()
        pendants=unique(pendants)
        return G,pendants
    except DecodeError:
    ...
    except KeyError:
```

Rolling the snowball

```
def snowball_round(G,seeds,myspace=False):
    t0=time()
    if myspace:
        seeds=get_myspace_url(seeds)
    sb_data=[]
    for s in range(0,len(seeds)):
        s_sg=get_sg(seeds[s])
        new_ego,pen=create_egonet(s_sg)
        for p in pen:
                sb_data.append(p)
        if s<1:
            sb_net=compose(G,new_ego)
        else:
            sb_net=compose(new_ego,sb_net)
        del new_ego
        if s==round(len(seeds)*0.2):
            sb_net.name='20% complete'
            sb_net.info()
            print 'AT: '+strftime('%m/%d/%Y, %H:%M:%S', gmtime())
            print ''
    ...
    # More time keeping, probably a MUCH better way to do this
    sb_data=array(sb_data)
    sb_data.flatten()
    sb_data=unique(sb_data)
    sb_net.info()
    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

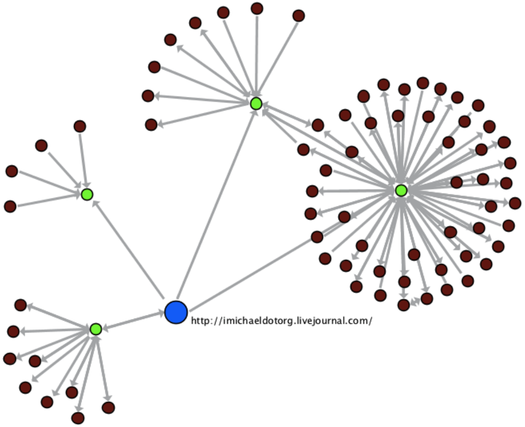
Step	Nodes	Edges	Mean Degree	Density
Seed	5	5	2.0	0.25
$k = 2$	75	115	3.0	0.02
$k = 3$	4,938	8,659	3.5	$3.6(10^{-4})$

- 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	$3.6(10^{-4})$

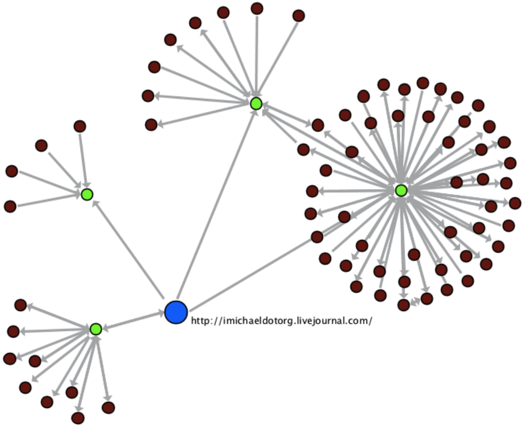
- Our seed is abnormally isolated, with only four neighbors
- Large jump after first snowball



Build the whole network

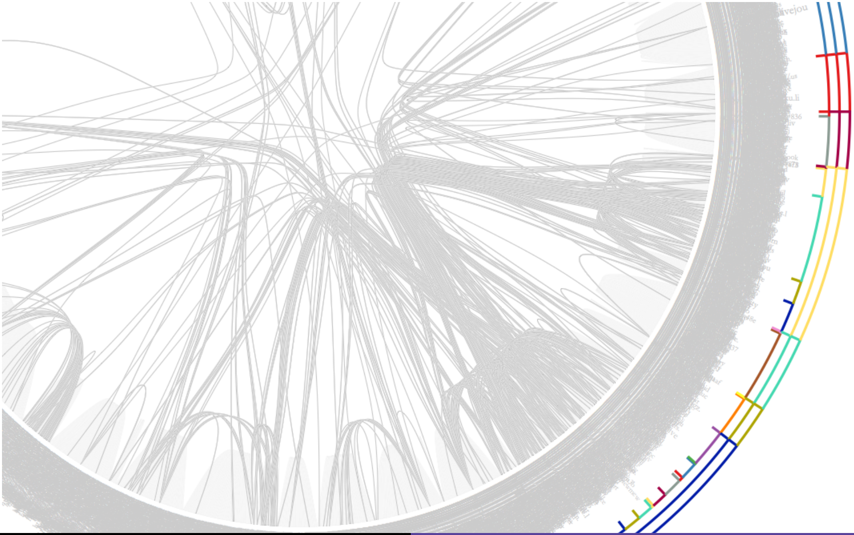
Step	Nodes	Edges	Mean Degree	Density
Seed	5	5	2.0	0.25
$k = 2$	75	115	3.0	0.02
$k = 3$	4,938	8,659	3.5	$3.6(10^{-4})$

- 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...



Python Dictionaries

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

Python Dictionaries

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

Working with the dict type

```
# Keys and values can be of any data type
>>> fruit_dict={"apple":1,"orange":[0.23,0.11],"banana":True }
```

Python Dictionaries

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

Working with the dict type

```
# Keys and values can be of any data type
>>> fruit_dict={"apple":1,"orange":[0.23,0.11],"banana":True }

# Can retrieve the keys and values as Python lists (vector)
>>> fruit_dict.keys()
["orange","apple","banana"]
```

Python Dictionaries

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

Working with the dict type

```
# Keys and values can be of any data type
>>> fruit_dict={"apple":1,"orange":[0.23,0.11],"banana":True }

# Can retrieve the keys and values as Python lists (vector)
>>> fruit_dict.keys()
["orange","apple","banana"]

# Or create a (key,value) tuple
>>> fruit_dict.items()
[("orange",[0.23,0.11]),("apple",1),("Banana",True)]
```

Python Dictionaries

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

Working with the dict type

```
# Keys and values can be of any data type
>>> fruit_dict={"apple":1,"orange":[0.23,0.11],"banana":True }

# Can retrieve the keys and values as Python lists (vector)
>>> fruit_dict.keys()
["orange","apple","banana"]

# Or create a (key,value) tuple
>>> fruit_dict.items()
[("orange",[0.23,0.11]),("apple",1),("Banana",True)]
# This becomes especially useful when you master Python ‘‘list comprehension’’
```

Python Dictionaries

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

Working with the dict type

```
# Keys and values can be of any data type
>>> fruit_dict={"apple":1,"orange":[0.23,0.11],"banana":True }

# Can retrieve the keys and values as Python lists (vector)
>>> fruit_dict.keys()
["orange","apple","banana"]

# Or create a (key,value) tuple
>>> fruit_dict.items()
[("orange",[0.23,0.11]),("apple",1),("Banana",True)]
# 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

Python Dictionaries

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

Working with the dict type

```
# Keys and values can be of any data type
>>> fruit_dict={"apple":1,"orange":[0.23,0.11],"banana":True }

# Can retrieve the keys and values as Python lists (vector)
>>> fruit_dict.keys()
["orange","apple","banana"]

# Or create a (key,value) tuple
>>> fruit_dict.items()
[("orange",[0.23,0.11]),("apple",1),("Banana",True)]
# This becomes especially useful when you master Python ''list comprehension''
```

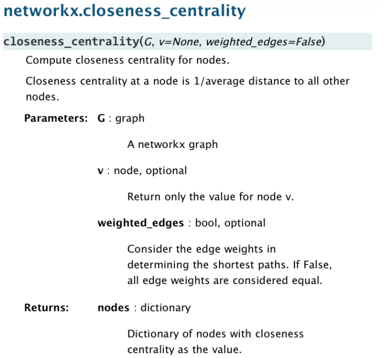
The Python dictionary is an extremely flexible and useful data structure, making it one of the primary advantages of Python over other languages

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

Now, try creating a dict of your own

Using dictionaries for network analysis

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:

In-degree centrality of Hartford data

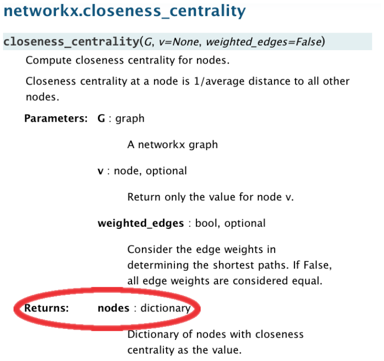
```
>>> in_cen=in_degree_centrality(hartford)
>>> in_cen
{1: 0.014218009478672987, 2: 0.018957345971563982,...
...
90: 0.0047393364928909956, 293: 0.0}
```

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

- But we can do so much more!

Using dictionaries for network analysis

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:

In-degree centrality of Hartford data

```
>>> in_cen=in_degree_centrality(hartford)
>>> in_cen
{1: 0.014218009478672987, 2: 0.018957345971563982,...
...
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

Find main component & symmetrize

```
# Many of the centrality metrics require undirected graphs, so we will symmetrize
>>> hartford_ud=hartford.to_undirected()
# The network also has many small components, but for
# this analysis we are interested in the largest
>>> hartford_mc=hartford_main=connected_component_subgraphs(hartford_ud)[0]
```

Next, we will calculate multiple measures

Computing multiple centralities

```
# Betweenness centrality
>>> bet_cen=betweenness_centrality(hartford_mc)
# Closeness centrality
>>> clo_cen=closeness_centrality(hartford_mc)
# Eigenvector centrality
>>> eig_cen=eigenvector_centrality(hartford_mc)
```

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

Find main component & symmetrize

```
# Many of the centrality metrics require undirected graphs, so we will symmetrize
>>> hartford_ud=hartford.to_undirected()
# The network also has many small components, but for
# this analysis we are interested in the largest
>>> hartford_mc=hartford_main=connected_component_subgraphs(hartford_ud)[0]
```

Next, we will calculate multiple measures

Computing multiple centralities

```
# Betweenness centrality
>>> bet_cen=betweenness_centrality(hartford_mc)
# Closeness centrality
>>> clo_cen=closeness_centrality(hartford_mc)
# Eigenvector centrality
>>> eig_cen=eigenvector_centrality(hartford_mc)
```

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

Find main component & symmetrize

```
# Many of the centrality metrics require undirected graphs, so we will symmetrize
>>> hartford_ud=hartford.to_undirected()
# The network also has many small components, but for
# this analysis we are interested in the largest
>>> hartford_mc=hartford_main=connected_component_subgraphs(hartford_ud)[0]
```

Next, we will calculate multiple measures

Computing multiple centralities

```
# Betweenness centrality
>>> bet_cen=betweenness_centrality(hartford_mc)
# Closeness centrality
>>> clo_cen=closeness_centrality(hartford_mc)
# Eigenvector centrality
>>> eig_cen=eigenvector_centrality(hartford_mc)
```

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

Find main component & symmetrize

```
# Many of the centrality metrics require undirected graphs, so we will symmetrize
>>> hartford_ud=hartford.to_undirected()
# The network also has many small components, but for
# this analysis we are interested in the largest
>>> hartford_mc=hartford_main=connected_component_subgraphs(hartford_ud)[0]
```

Next, we will calculate multiple measures

Computing multiple centralities

```
# Betweenness centrality
>>> bet_cen=betweenness_centrality(hartford_mc)
# Closeness centrality
>>> clo_cen=closeness_centrality(hartford_mc)
# Eigenvector centrality
>>> eig_cen=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

Function for finding most central actor

```
def highest_centrality(cent_dict):
    """Returns node key with largest value from
    NX centrality dict"""
    # Create ordered tuple of centrality data
    cent_items=cent_dict.items()
    # List comprehension!
    cent_items=[(b,a) for (a,b) in cent_items]
    # Sort in descending order
    cent_items.sort()
    cent_items.reverse()
    return cent_items[0][1]
```

Now, just ask for the answer

Finding Most central actors

```
>>>  print("Actor "+str(highest_centrality(bet_cen))+" has the highest Betweenness centrality")
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

Function for finding most central actor

```
def highest_centrality(cent_dict):
    """Returns node key with largest value from
    NX centrality dict"""
    # Create ordered tuple of centrality data
    cent_items=cent_dict.items()
    # List comprehension!
    cent_items=[(b,a) for (a,b) in cent_items]
    # Sort in descending order
    cent_items.sort()
    cent_items.reverse()
    return cent_items[0][1]
```

Now, just ask for the answer

Finding Most central actors

```
>>>  print("Actor "+str(highest_centrality(bet_cen))+" has the highest Betweenness centrality")
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

Function for finding most central actor

```
def highest_centrality(cent_dict):
    """Returns node key with largest value from
    NX centrality dict"""
    # Create ordered tuple of centrality data
    cent_items=cent_dict.items()
    # List comprehension!
    cent_items=[(b,a) for (a,b) in cent_items]
    # Sort in descending order
    cent_items.sort()
    cent_items.reverse()
    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

```
>>> print("Actor "+str(highest_centrality(bet_cen))+" has the highest Betweenness centrality")
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

Function for finding most central actor

```
def highest_centrality(cent_dict):
    """Returns node key with largest value from
    NX centrality dict"""
    # Create ordered tuple of centrality data
    cent_items=cent_dict.items()
    # List comprehension!
    cent_items=[(b,a) for (a,b) in cent_items]
    # Sort in descending order
    cent_items.sort()
    cent_items.reverse()
    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

```
>>>  print("Actor "+str(highest_centrality(bet_cen))+" has the highest Betweenness centrality")
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

Function for finding most central actor

```
def highest_centrality(cent_dict):
    """Returns node key with largest value from
    NX centrality dict"""
    # Create ordered tuple of centrality data
    cent_items=cent_dict.items()
    # List comprehension!
    cent_items=[(b,a) for (a,b) in cent_items]
    # Sort in descending order
    cent_items.sort()
    cent_items.reverse()
    return cent_items[0][1]
```

List comprehension

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

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

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

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

```
# Calculate clustering coefficients of each node (return as dict)
clus=clustering(hartford_mc,with_labels=True)
# Get counts of nodes membership for each clustering coefficient, and clean up
unique_clus=list(unique(clus.values()))
clus_counts=zip(map(lambda c: clus.values().count(c),unique_clus),unique_clus)
clus_counts.sort()
clus_counts.reverse()
# Create a subgraph from nodes with most frequent clustering coefficient
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

```
# Calculate clustering coefficients of each node (return as dict)
clus=clustering(hartford_mc,with_labels=True)
# Get counts of nodes membership for each clustering coefficient, and clean up
unique_clus=list(unique(clus.values()))
clus_counts=zip(map(lambda c: clus.values().count(c),unique_clus),unique_clus)
clus_counts.sort()
clus_counts.reverse()
# Create a subgraph from nodes with most frequent clustering coefficient
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

```
# Calculate clustering coefficients of each node (return as dict)
clus=clustering(hartford_mc,with_labels=True)
# Get counts of nodes membership for each clustering coefficient, and clean up
unique_clus=list(unique(clus.values()))
clus_counts=zip(map(lambda c: clus.values().count(c),unique_clus),unique_clus)
clus_counts.sort()
clus_counts.reverse()
# Create a subgraph from nodes with most frequent clustering coefficient
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`

Toy community detection example (not a good one)

Calculating clustering coefficients

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

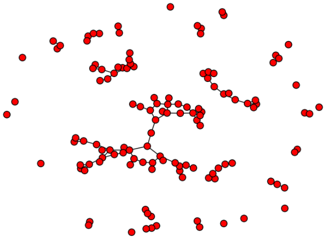
- Use the `with_labels` to return a `dict` keyed by node label
- The `zip` function takes two `lists` and returns a `tuple`
- More complex list comprehension with logic operator

Toy community detection example (not a good one)

Calculating clustering coefficients

```
# Calculate clustering coefficients of each node (return as dict)
clus=clustering(hartford_mc,with_labels=True)
# Get counts of nodes membership for each clustering coefficient, and clean up
unique_clus=list(unique(clus.values()))
clus_counts=zip(map(lambda c: clus.values().count(c),unique_clus),unique_clus)
clus_counts.sort()
clus_counts.reverse()
# Create a subgraph from nodes with most frequent clustering coefficient
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



Introduction to matplotlib

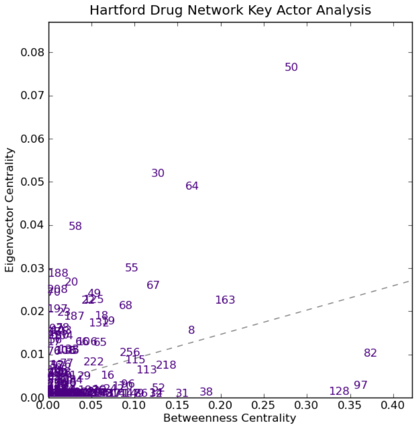
Recall Python's scientific computing trinity: NumPy, SciPy and matplotlib

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

Introduction to matplotlib

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

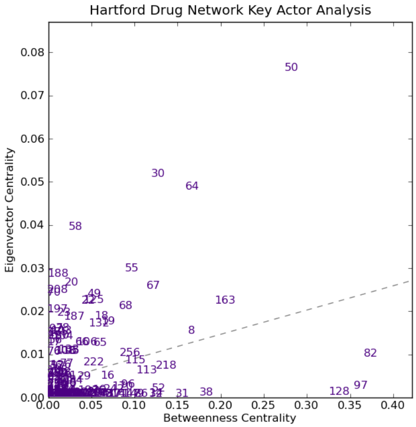
- While NumPy and SciPy do most of the behind the scenes work, you will interact with matplotlib frequently for when doing network analysis



Introduction to matplotlib

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

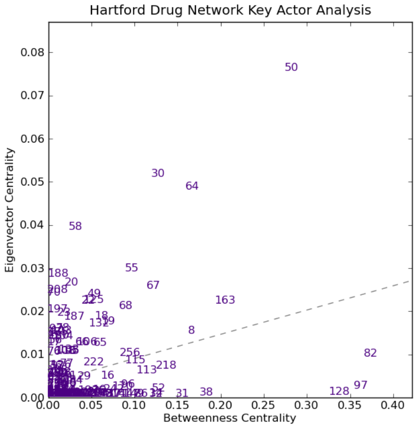


We will need to create a function that takes two centrality dict and generates this plot

Introduction to `matplotlib`

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



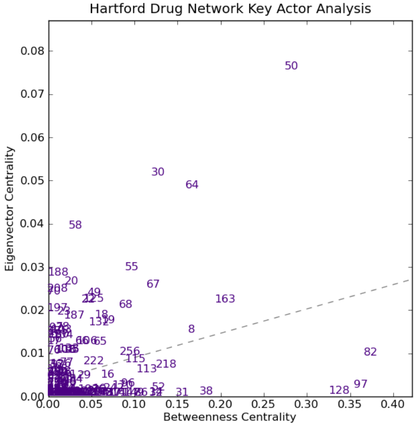
We will need to create a function that takes two centrality `dict` and generates this plot

1. Create a `matplotlib` figure

Introduction to matplotlib

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



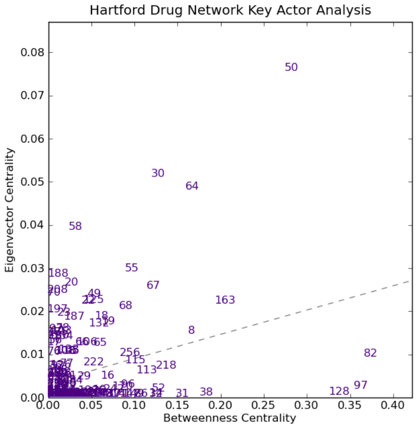
We will need to create a function that takes two centrality dict and generates this plot

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

Introduction to matplotlib

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



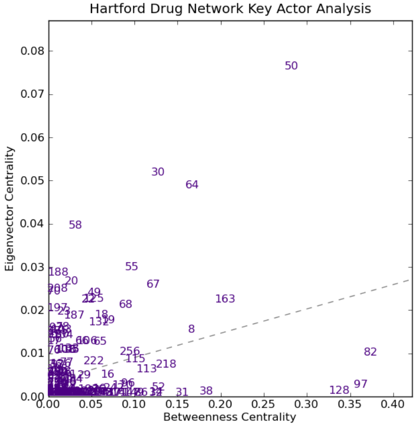
We will need to create a function that takes two centrality dict and generates this plot

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

Introduction to matplotlib

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



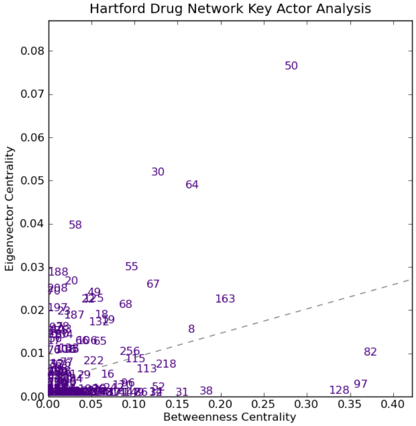
We will need to create a function that takes two centrality dict and generates this plot

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

Introduction to matplotlib

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

Creating a key actor plot in matplotlib

The centrality_scatter function, part one

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

Creating a key actor plot in matplotlib

The centrality_scatter function, part one

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

- Create a canvas to draw on

Creating a key actor plot in matplotlib

The centrality_scatter function, part one

```
def centrality_scatter(met_dict1,met_dict2,path="",ylab="",xlab="",title="",reg=False):
    # Create figure and drawing axis
    fig=P.figure(figsize=(7,7))
    ax1=fig.add_subplot(111)
    # Create items so actors can be sorted properly
    met_items1=met_dict1.items()
    met_items2=met_dict2.items()
    met_items1.sort()
    met_items2.sort()
    # Grab data
    xdata=[(b) for (a,b) in met_items1]
    ydata=[(b) for (a,b) in met_items2]
    # Add each actor to the plot by ID
    for p in xrange(len(met_items1)):
        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

Creating a key actor plot in matplotlib

The centrality_scatter function, part one

```
def centrality_scatter(met_dict1,met_dict2,path="",ylab="",xlab="",title="",reg=False):
    # Create figure and drawing axis
    fig=P.figure(figsize=(7,7))
    ax1=fig.add_subplot(111)
    # Create items so actors can be sorted properly
    met_items1=met_dict1.items()
    met_items2=met_dict2.items()
    met_items1.sort()
    met_items2.sort()
    # Grab data
    xdata=[(b) for (a,b) in met_items1]
    ydata=[(b) for (a,b) in met_items2]
    # Add each actor to the plot by ID
    for p in xrange(len(met_items1)):
        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

Creating a key actor plot in matplotlib

The centrality_scatter function, part one

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

Creating a key actor plot in matplotlib

The centrality_scatter function, part one

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

- Add a best fit line

Creating a key actor plot in matplotlib

The centrality_scatter function, part one

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

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

Creating a key actor plot in matplotlib

The centrality_scatter function, part one

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

Exporting network data and analytics

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

Exporting network data and analytics

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

Exporting network data and analytics

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

Exporting network data and analytics

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

Exporting network data and analytics

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

Exporting network data and analytics

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

Exporting network data and analytics

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

Exporting network data and analytics

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

Exporting network data and analytics

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

Saving network data in different formats

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

NX syntax for loading a file

$$\begin{array}{cccc} >>> & \text{write\_format(G,} & \text{“path/to/file.txt”,} & \textit{...options...}\text{)} \\ & \uparrow & \uparrow & \uparrow \\ & \text{NX function, net variable} & \text{File to be written} & \text{Nodes/edge data, etc.} \end{array}$$

Saving network data in different formats

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

NX syntax for loading a file

$$
\begin{array}{llll}
>>> & \text{write\_format(G,} & \text{``path/to/file.txt'',} & \text{...}\textit{options}\text{...)} \\
& \uparrow & \uparrow & \uparrow \\
& \text{NX function, net variable} & \text{File to be written} & \text{Nodes/edge data, etc.}
\end{array}
$$

Saving network data in different formats

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

NX syntax for loading a file

$$\begin{array}{cccc} >>> & \text{write\_format(G,} & \text{“path/to/file.txt”,} & \textit{...options...}\text{)} \\ & \uparrow & \uparrow & \uparrow \\ & \text{NX function, net variable} & \text{File to be written} & \text{Nodes/edge data, etc.} \end{array}$$

Saving network data in different formats

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

NX syntax for loading a file

$$\begin{array}{llll} >>> & \text{write\_format(G,} & \text{“path/to/file.txt”,} & \textit{...options...}\text{)} \\ & \uparrow & \uparrow & \uparrow \\ & \text{NX function, net variable} & \text{File to be written} & \text{Nodes/edge data, etc.} \end{array}$$

Saving network data in different formats

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

NX syntax for loading a file

$$\texttt{>>>}\quad \underset{\text{NX function, net variable}}{\underset{\uparrow}{\text{write\_format(G,}}}\quad \underset{\text{File to be written}}{\underset{\uparrow}{\text{“path/to/file.txt”,}}}\quad \underset{\text{Nodes/edge data, etc.}}{\underset{\uparrow}{\textit{...options...}\text{)}}}$$

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

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

```
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

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

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

```
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

```
def add_metric(G,met_dict):
    """Adds metric data to G from a dictionary keyed by node labels"""
    if(G.nodes().sort()==met_dict.keys().sort()):
        for i in met_dict.keys():
            G.add_node(i,metric=met_dict[i])
        return G
    else:
        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

```
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

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

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

Using the Python CSV library

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

Exporting centrality data to CSV

```
import csv
...
def csv_exporter(data_dict,path):
    """Takes a dict of centralities keyed by column headers and exports
    data as a CSV file"""
    # Create column header list
    col_headers=["Actor"]
    col_headers.extend(data_dict.keys())
    # Create CSV writer and write column headers
    writer=csv.DictWriter(open(path,"w"),fieldnames=col_headers)
    writer.writerow(dict((h,h) for h in col_headers))
    # Write each row of data
    for j in data_dict[col_headers[1]].keys():
        # Create a new dict for each row
        row=dict.fromkeys(col_headers)
        row["Actor"]=j
        for k in data_dict.keys():
            row[k]=data_dict[k][j]
        writer.writerow(row)
```

Using the Python CSV library

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`

Exporting centrality data to CSV

```
import csv
...
def csv_exporter(data_dict,path):
    """Takes a dict of centralities keyed by column headers and exports
    data as a CSV file"""
    # Create column header list
    col_headers=["Actor"]
    col_headers.extend(data_dict.keys())
    # Create CSV writer and write column headers
    writer=csv.DictWriter(open(path,"w"),fieldnames=col_headers)
    writer.writerow(dict((h,h) for h in col_headers))
    # Write each row of data
    for j in data_dict[col_headers[1]].keys():
        # Create a new dict for each row
        row=dict.fromkeys(col_headers)
        row["Actor"]=j
        for k in data_dict.keys():
            row[k]=data_dict[k][j]
        writer.writerow(row)
```

Using the Python CSV library

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

Exporting centrality data to CSV

```
import csv
...
def csv_exporter(data_dict,path):
    """Takes a dict of centralities keyed by column headers and exports
    data as a CSV file"""
    # Create column header list
    col_headers=["Actor"]
    col_headers.extend(data_dict.keys())
    # Create CSV writer and write column headers
    writer=csv.DictWriter(open(path,"w"),fieldnames=col_headers)
    writer.writerow(dict((h,h) for h in col_headers))
    # Write each row of data
    for j in data_dict[col_headers[1]].keys():
        # Create a new dict for each row
        row=dict.fromkeys(col_headers)
        row["Actor"]=j
        for k in data_dict.keys():
            row[k]=data_dict[k][j]
        writer.writerow(row)
```

Using the Python CSV library

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

Exporting centrality data to CSV

```
import csv
...
def csv_exporter(data_dict,path):
    """Takes a dict of centralities keyed by column headers and exports
    data as a CSV file"""
    # Create column header list
    col_headers=["Actor"]
    col_headers.extend(data_dict.keys())
    # Create CSV writer and write column headers
    writer=csv.DictWriter(open(path,"w"),fieldnames=col_headers)
    writer.writerow(dict((h,h) for h in col_headers))
    # Write each row of data
    for j in data_dict[col_headers[1]].keys():
        # Create a new dict for each row
        row=dict.fromkeys(col_headers)
        row["Actor"]=j
        for k in data_dict.keys():
            row[k]=data_dict[k][j]
        writer.writerow(row)
```

Using the Python CSV library

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

Exporting centrality data to CSV

```
import csv
...
def csv_exporter(data_dict,path):
    """Takes a dict of centralities keyed by column headers and exports
    data as a CSV file"""
    # Create column header list
    col_headers=["Actor"]
    col_headers.extend(data_dict.keys())
    # Create CSV writer and write column headers
    writer=csv.DictWriter(open(path,"w"),fieldnames=col_headers)
    writer.writerow(dict((h,h) for h in col_headers))
    # Write each row of data
    for j in data_dict[col_headers[1]].keys():
        # Create a new dict for each row
        row=dict.fromkeys(col_headers)
        row["Actor"]=j
        for k in data_dict.keys():
            row[k]=data_dict[k][j]
        writer.writerow(row)
```

Using the Python CSV library

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

Exporting centrality data to CSV

```
import csv
...
def csv_exporter(data_dict,path):
    """Takes a dict of centralities keyed by column headers and exports
    data as a CSV file"""
    # Create column header list
    col_headers=["Actor"]
    col_headers.extend(data_dict.keys())
    # Create CSV writer and write column headers
    writer=csv.DictWriter(open(path,"w"),fieldnames=col_headers)
    writer.writerow(dict((h,h) for h in col_headers))
    # Write each row of data
    for j in data_dict[col_headers[1]].keys():
        # Create a new dict for each row
        row=dict.fromkeys(col_headers)
        row["Actor"]=j
        for k in data_dict.keys():
            row[k]=data_dict[k][j]
        writer.writerow(row)
```

Using the Python CSV library

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

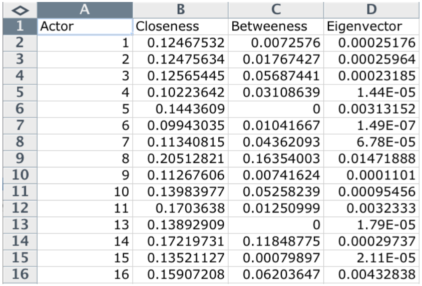
Exporting centrality data to CSV

```
import csv
...
def csv_exporter(data_dict,path):
    """Takes a dict of centralities keyed by column headers and exports
    data as a CSV file"""
    # Create column header list
    col_headers=["Actor"]
    col_headers.extend(data_dict.keys())
    # Create CSV writer and write column headers
    writer=csv.DictWriter(open(path,"w"),fieldnames=col_headers)
    writer.writerow(dict((h,h) for h in col_headers))
    # Write each row of data
    for j in data_dict[col_headers[1]].keys():
        # Create a new dict for each row
        row=dict.fromkeys(col_headers)
        row["Actor"]=j
        for k in data_dict.keys():
            row[k]=data_dict[k][j]
        writer.writerow(row)
```

The results of CSV export

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	Betweeness	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