BST 267: Introduction to Social and Biological Networks

Lab 5

JP Onnela

Department of Biostatistics
Harvard T.H. Chan School of Public Health
Harvard University

December 1 & 2, 2022

Lab 5: Modeling Epidemics on Networks

Modeling Epidemics on Networks

- The goal of this lab is to practice modeling spreading processes on networks
- **Deliverable:** Return these through Canvas: 1) Jupyter Notebook (.ipynb file) and 2) HTML version of the notebook (.html file)

Quick Reference

BST 267: Introduction to Networks

Python 3 & NetworkX Reference

Some imports:

```
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
import random, math
```

Objects:

```
text = "Python"
type(text)
dir(text)
help(text)
help(text.lower)
```

Sequences:

```
S = "Python"
S = [1,2,3,4,5,6]
len(S)
S[0]
S[-1]
S[0:6]
```

Strings (immutable):

```
S = "Python Python"
S.find('y')
S.replace('y', 'Y')
S.split(" ")
```

Lists (mutable):

```
mylist = []
mylist = list()
numbers = [1,2,3,4]
mylist[0]
mylist[0] = 50
mylist.append(5)
mylist.pop()
mylist.sort()
mylist.reverse()
```

Ranges (immutable):

```
indices = range(10)
indices = range(1,11)
indices = range(1,11,2)
```

Dictionaries (mutable):

```
mydict = {}
mydict = dict()
mydict = {'Tim': 29, 'Jim': 31}
mydict['Tim']
mydict['Amy'] = 32
mydict.keys()
mydict.values()
```

Control flow and loops:

```
if a > b:
    print("a is greater")
elif b > a:
    print("b is greater")
else:
    print("a and be are equal")

for number in [1,2,3,4,5]:
    print(number, number**2)

mydict = {'Tim': 29, 'Jim': 31}
for name in mydict:
    print(name, mydict[name])
```

Adding and removing graph elements:

```
G = nx.Graph()
G.add_node(1)
G.add_nodes_from([1,2])
G.remove_node(1)
G.remove_nodes_from([1,2])
G.add_edge(1,2)
G.add_edges_from([(1,2), (1,3)])
G.remove_edge(1,2)
G.remove_edges_from([(1,2), (1,3)])
G.clear()
```

Listing and checking graph elements:

```
G.nodes()
G.edges()
G.has_node(1)
G.has_edge(1,2)
G.number_of_nodes()
G.number_of_edges()
```

Basic graph properties:

```
G.degree()
G.degree(1)
G.neighbors(1)
nx.clustering(G)
nx.clustering(G,1)
nx.connected_components(G)
```

Sampling:

```
S = [1,2,3,4,5]
random.choice(S)
random.sample(S,2)
```

Plotting:

```
plt.figure()
nx.draw(G)
degrees = G.degree().values()
clustering = nx.clustering(G).values()
plt.hist(degrees, 15)
plt.plot(degrees, clustering, "bo")
plt.xlabel("Degree")
plt.ylabel("Clustering")
```

Simulating SIR processes on networks

- In this lab we'll simulate SIR processes on networks
- Our goal is to write three functions:
  - Function `spread` will generate transitions from state S to state I
  - Function `recover` will generate transitions from state I to state R
  - Function `simulate` to track nodes in each state and to call `spread` and `recover`
- We will also write a function for making some plots
- Keep this in mind: Let's say that you keep track of the state of each node in a mutable object, such as a list. If you provide that object to a function as an input and modify the object, the function will modify the object itself (rather than a local copy of the object).

Getting started

- We'll need our standard imports

```
1 import networkx as nx
2 import random
3 import matplotlib.pyplot as plt
4 import numpy as np
5 %matplotlib notebook
```

Question 1: Modeling recovery

- Although logically the S to I transition has to happen before the I to R transition, the latter is easier to handle, so we'll start with that
- Write a function that carries out the I to R transition over a single time step
- The function should take as its input two lists `i_nodes` and `r_nodes` that list the IDs of the nodes in these states
- The function should also take a third input argument `p` that is the probability of recovery for an I node per time step
- The function should return a list called `new_recoveries` that contains the IDs of the nodes that recovered during this round

```
1 # Carry out the I -> R recovery process for one time step.
2 def recover(i_nodes, r_nodes, p):
3     # YOU WILL NEED TO WRITE THIS CODE
```

Question 2: Modeling transmission

- Let's then write the function to deal with S to I transitions
- During each time step, each infected (I) node selects one of its neighbors uniformly at random regardless of its status (S vs. I vs. R)
- If that neighboring node happens to be susceptible (S), then with probability p that neighboring node will become infected
- This means that each infected (I) node can only spread the infection to at most one of its neighbors during any given time step
- The function should take as its input the network `G` and two lists, `s_nodes` and `i_nodes`, which list the IDs of the nodes in these states
- The function should also take a fourth input argument `p`, the probability of infection (S to I transition)

```
1 #Carry out the S -> I spreading process for one time step.
2 def spread(G, s_nodes, i_nodes, p):
3     # YOU WILL NEED TO WRITE THIS CODE
```

Question 3: Putting it all together

- Let's now write function `simulate` to do the following:
  - Initialize and update `i_nodes`, `s_nodes`, `r_nodes` to keep track of the nodes in S, I, and R states during the current simulation round
  - Record the number of nodes in each state at every point in time in variables `num_s_nodes`, `num_i_nodes`, and `num_r_nodes`
  - Call `spread` and `recover` as many times as specified by `num_time_steps`
- The function should return thee lists: `num_s_nodes`, `num_i_nodes`, and `num_r_nodes`

```
1 # Simulate an epidemic.
2 def simulate(G, p_si, p_ir, num_seeds, num_time_steps):
3     # YOU WILL NEED TO WRITE THIS CODE
```

- Think carefully about the order in which you need to call `spread` and `recover`
- Python sets might be useful here

Question 4: Plotting

- Next we want to plot the proportion of S, I, and R nodes as a function of time
- Let's write the following function
- Note the use of `np.array`

```
# Plot the number of S, I, and R nodes as a function of time.
def make_plot(num_nodes, num_s_nodes, num_i_nodes, num_r_nodes, num_time_steps):
    h1, = plt.plot(np.array(num_s_nodes) / num_nodes)
    h2, = plt.plot(np.array(num_i_nodes) / num_nodes)
    h3, = plt.plot(np.array(num_r_nodes) / num_nodes)
    plt.xlabel("Time")
    plt.ylabel("Fraction of S, I, and R nodes")
    plt.legend([h1,h2,h3], ["S nodes","I nodes","R nodes"], loc="center left")
    plt.xlim([0, num_time_steps])
```

- Now run the simulation