SISMID Module 9
Lab 4: Network epidemiology

Thomas J. Hladish, Joel C. Miller and Lauren Ancel Meyers

July 2014

Modeling the spread of infectious diseases

Whether in humans or other host organisms, there are clear, practical reasons why we might be interested in how diseases spread. Since there are also clear, practical reasons why we usually don't want to experimentally infect real populations, we turn to mathematical and computational modeling.

In general, we try to answer three kinds of questions through modeling:

- What will happen? (e.g. How many people will get the flu next year?)
- What should we do? (e.g. What intervention measure is most likely to reduce that number?)
- What factors matter? (e.g. Is it host birth rate or pathogen evolution that drives some phenomenon?)

Epidemiological models range from the abstract, where we only keep track of proportions of a population that are susceptible, infectious, and recovered (the SIR model) using a set of differential equations, to the very explicit, where every individual in a population has specific interactions with other individuals (an agent-based network model). Models can become arbitrarily complex depending on what you include. In general, however, there are major trade-offs: simple, abstract models may be quick to run and easy to interpret but not very realistic, while complex, explicit models may be time consuming to make and run, and hard to interpret, but might be necessary to accurately reproduce real phenomena. In general, it's best to start simple and see if you can describe the real world without dozens of parameters.

The percolation model

One approach to network epidemiology comes from physics, where researchers wanted to describe how a liquid can stochastically percolate through a semi-porous medium, like water percolating through sandstone. As it happens, this model can be readily adapted to describe how an infectious disease can "percolate" through a population.

To use a percolation model, we need to know two things: the contact network of our population, and the probability T that a contact will result in disease transmission, given contact between an infected and a susceptible individual. The contact network is both a property of the population and of the disease; for example, if we are modeling an STD, "contact" may mean sexual interaction, whereas if we are modeling influenza, "contact" might mean handling the same shopping cart. The probability of transmission T, called **transmissibility**, is something that we generally try to estimate for a given disease based on data from real epidemics.

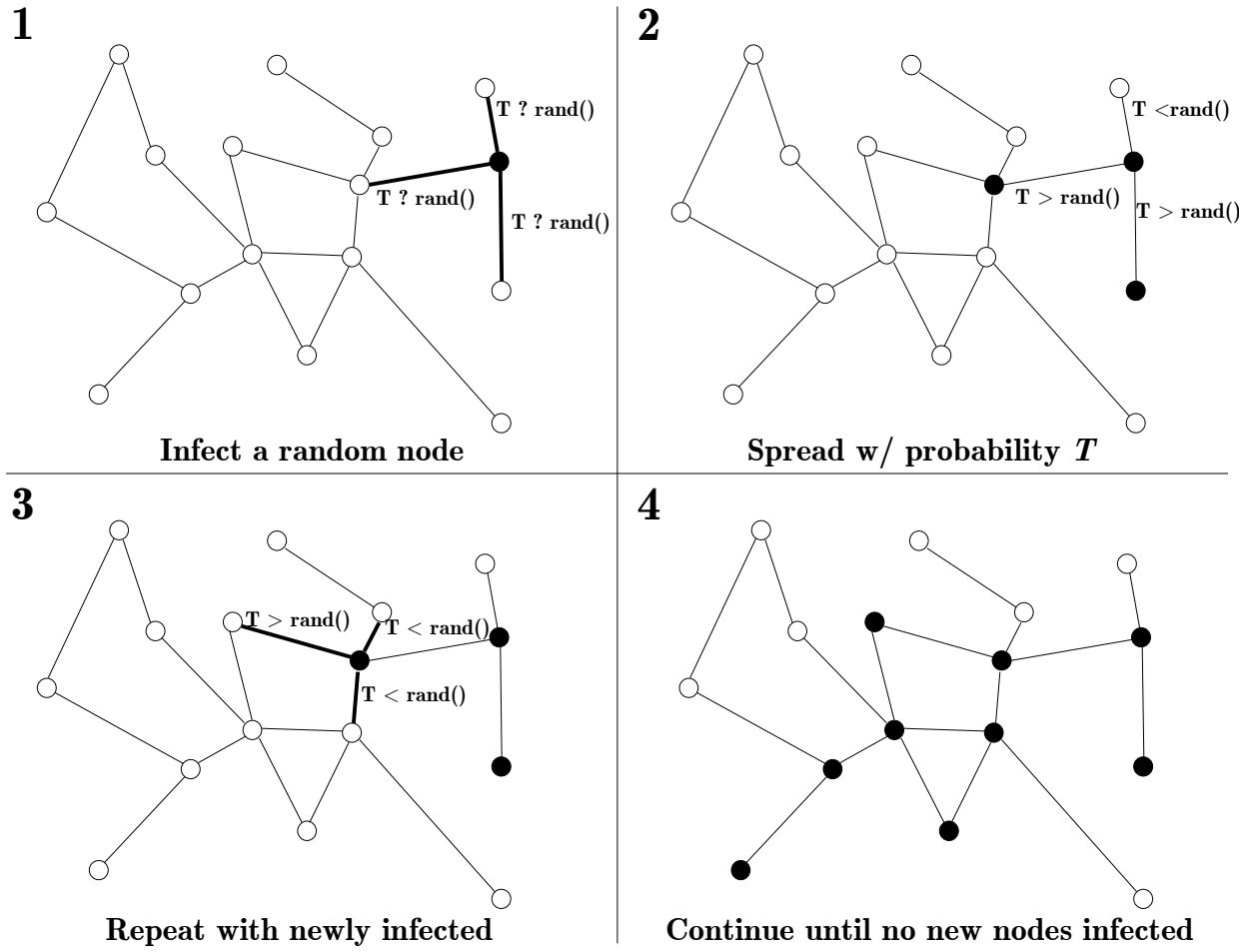

Figure 1: Cartoon of the steps in a percolation simulation

In Figure 1, we see a cartoon of how a percolation simulation works.

Step 1 We begin with a network of susceptible (= white) nodes. An infection is introduced by selecting a random node to infect; in epidemiology, this first infected individual is called **patient zero**. To determine whether the infection spreads from patient zero, we will compare the value of T (0.5 in this case) with a randomly drawn number[1] (uniform on [0,1)) for each of patient zero's outbound edges, shown in bold. If T is greater, the infection spreads; otherwise, it does not.

Step 2 After drawing a random number for each of the three edges, we found that T was greater in two cases and we therefore propagate the infected state to those two nodes. If T had been smaller for all three, the simulation would end at this step: each edge connected to an infected individual is tested exactly once in the percolation model.

Step 3 We now look for susceptible neighbors of the nodes infected in the last iteration. For each of those neighbors, we repeat the process of drawing a random number and comparing it to T; if T is greater, we change the state of the corresponding nodes to "infected."

Step 4 Repeat Step 3 until an iteration when no new nodes get infected.

Note that while edges are tested at most once, a given node may have multiple opportunities to get infected if its degree is > 1.

[1] To save space, Figure 1 uses a rand() function, although we will use Python's random.random() function.

When the simulation is complete, we tally the nodes that got infected; this is the **epidemic size**. Epidemic size may be absolute (a count of infected nodes) or relative (the fraction of nodes that were infected); usually we will use the latter, so that networks of different sizes can be compared. Knowing the epidemic size from one simulation run does not tell us the likelihood of getting that particular epidemic size, however. Generally we repeat the simulation many times, since any given run could result in anything from one node being infected (patient zero) to all nodes. By looking at a large number of runs, we can calculate the average epidemic size, as well as the probability[2] that any given outcome will occur.

1 Working with list variables

In order to do a simulation like this, we need some more tools for working with lists. Note that this is merely an introduction to the really cool things you can do with lists in Python. You can see more examples at `http://docs.python.org/tutorial/datastructures.html`

List assignment

As you know from when we looked at the `range()` function, lists are a kind of ordered, "container" variable, used to group together other values. Some functions like `range()` create and return lists, but you can also create them yourself as a list of comma-separated values (items) between square brackets. List items need not all have the same type (e.g. strings, integers, even other lists).

```
>>> a = ['pi', 3.141592653, 100, 1234]   # Assign a mixture of strings and numbers to 'a'
>>> a
['pi', 3.141592653, 100, 1234]
>>> a[1] = 'ice cream'                   # Overwrite one of the values (the second one--
>>> a                                    #      remember, indexing starts at 0)
['pi', 'ice cream', 100, 1234]
```

It is important to note that you can only overwrite a value in a list if that position in a list exists in the first place. For example, if a list has four elements, we can overwrite the first one, but not the fifth. See "Modifying lists" below for how to do that.

List access and length

You can access the stored values in lists using *indices*, and get the number of elements in the list using `len()`.

```
>>> a[0]
'pi'
>>> len(a)                               # How many elements are in a?
4
>>> a[3]                                 # 3 is therefore the index of the last element
1234
>>> a[-1]                                # An idiom for the last element in the list
1234
```

[2] Technically, we can only estimate this probability.

Modifying lists

We can add elements to the end of lists, remove elements from anywhere, and sort the list

```
>>> a = ['pi', 'ice cream', 100, 1234]
>>> a.append(2.17)                      # Place 2.17 in position 5 using append()
>>> a
['pi', 'ice cream', 100, 1234, 2.17]
>>> a.pop()                             # Remove the last element (the default) using pop()
2.17
>>> a
['pi', 'ice cream', 100, 1234]
>>> a.pop(1)                            # Remove the second element in the list using pop(1)
'ice cream'
>>> a
['pi', 100, 1234]
>>> a.sort()                            # Sort the contents of the list alphanumerically
>>> a
[100, 1234, 'pi']
```

List membership testing

Python has a very handy way of testing whether something is an element in a list: the `in` keyword.

```
>>> 'pi' in a
True
>>> 'Pi' in a                # Case sensitive!
False
```

2 Making things stochastic with `random()`, `choice()`, and `shuffle()`

In order to introduce randomness in our simulation, we need a few tools from the `random` library:

```
>>> from random import *
>>> random()                 # Draw a random float on [0,1)
0.79593921591157424
>>> random()
0.63001049048895807
>>> x = range(10)            # Store [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] in x
>>> choice(x)                # Sample (with replacement) an element of a list
7                            # (this doesn't change the list)
>>> choice(x)
4
>>> shuffle(x)               # Shuffle the order of the elements in a list
>>> x
[9, 8, 3, 6, 0, 7, 1, 5, 2, 4]
```

The `random` library uses a fairly fast and very high quality pseudo-random number generator called Mersenne Twister.

3 Implementing a percolation simulation in Python

Now we finally know everything we need to write a simulation! The percolation algorithm we looked at on page 1 can be decomposed into the following functional blocks of code (a sort of road map):

0. Import any functions we need from other libraries
1. Define the parameters we'll need—it's good practice to put these at the top of your program
2. Construct the network
3. Store node states somehow, marking one (patient zero) infected, and the rest susceptible
4. Do the simulation
5. Tally the results

We can implement these steps in following way[3]:

```
#!/usr/bin/python
# 0.) Import functions
from networkx import *
from random import *

# 1.) Define parameters
T = 0.9        # probability of transmission

# 2.) Construct network
G=Graph()
##################################################
# Insert code to construct the network
# e.g. what you developed for Lab 3, Section 3
##################################################
net_size = float(G.order())

# 3.) Store node states
p_zero = choice(G.nodes()) # Randomly choose one node
infected    = [p_zero]
recovered   = []

# 4.) Run the simulation
while len(infected) > 0:
    v = infected.pop(0)
    for u in G.neighbors(v):
        # We need to make sure Node u is still susceptible
        if u not in infected and u not in recovered:
            if random() < T:
                infected.append(u)
    recovered.append(v)

# 5.) Tally the total fraction that got infected
epi_size = len(recovered)/net_size
print epi_size
```

[3]Note that this is a fairly simple implementation, at least in terms of the Python constructs and variable types it uses. We can optimize it significantly and make it 5-10 times faster by using slightly more sophisticated Python. For an example of such an implementation, see Appendix A in this lab.

The only thing that is new in the percolation code is the `while` loop. Just like a `for` loop, `while` loops run a chunk of code multiple times. The difference is that `for` loops run the code a specified number of times, whereas `while` loops run the code as long as a certain statement is true. In this case, we are saying that the simulation continues as long as the `infected` list contains one or more infected nodes.

You can download the above code from the "Lab 3: Simple percolation code" link. Complete this program by finishing the second block of code, so that we construct the real sexual network that you analyzed in Lab 2, Section 5. Note that we are using a high transmissibility value, which we would expect for an STD like chlamydia or gonorrhea. Once you are done modifying your code, run your program 10 or more times. How variable do the results seem to be?

3.1 Determining the epidemic size distribution

Let's investigate the results in a more systematic manner by modifying the program in the following ways. **If you're not sure how to make these changes, please ask me, or refer to the solution in Appendix B.**

- **Make the simulation automatically run 100 times by using a `for` loop.** Indent everything in Blocks 3, 4, and 5; then add a loop statement just before Block 3.
- **Keep track of the results of each run.** In Block 1 of the code, add an empty list variable where we will ultimately store the results (I suggest you call it `results`). At the end of Block 5, append the outcome to the results list.
- **Plot the results.** Import the `hist()` and `show()` functions from `pylab`, and then at the end of the program, create a histogram of the values stored in the `results` list.

3.2 Analysis of epidemic size distribution

How would you describe the distribution you generated from the real sexual network data? Uni- or bi-modal? Is there a clear separation between small outbreaks and big epidemics?

Now run your program again, but change the input file so that you use the randomized (shuffled) sexual network. Remember, this network contains the same number of nodes, edges, and even degree sequence as the real network (refer back to Section 4 of Lab 2 for graphical representations of the two networks).

How is the distribution different? What do you think results in the difference, when there is so much in common between the two networks?

Additional exercise: Transmissibility vs. epidemic size

Let's investigate the relationship between transmissibility `T` and epidemic size `S` a little more thoroughly. We're going to run the percolation simulation a lot of times for several different parameter values, to see how those parameters affect the outcome. For all of the runs with each set of parameter values, we're going to average the outcomes (epidemic sizes). That way we can reduce the effect of random variations in outcome and instead focus on important trends. In order to visualize these trends, we're going to create a scatter plot with the resulting mean epidemic sizes. Modify the program you've been using to do the following:

1. Using either a real or generated network, run the simulation 100 times for each of the T values[4] in (0.0, 0.05, 0.1, 0.15, 0.2, ... , 0.95, 1.0). Store the mean epidemic size resulting from each T value in a list called `mean_epi_sizes`.
2. When you have generated the `mean_epi_sizes` for the network, use the `pylab.scatter()` function to create the scatter plot. The code to generate the plot could look something like this:

```
1 from pylab import scatter, show
2 scatter(T_values, mean_epi_sizes)
3 show()
```

[4]You can generate this list of T values using `t_values = [i/20.  for i in range(21)]`

Appendix A: Optimizing the percolation simulation

In the revised percolation simulation below, we keep track of who is susceptible using a dictionary variable (called `states`). In this case, we are using a generated random graph, but we could just as easily use a graph generated from an edgelist.

```
#!/usr/bin/python
from networkx import *
from random import random, choice

def percolate(G, T):
    states = dict([(node, 's') for node in G.nodes()])

    p_zero = choice(G.nodes())
    states[p_zero] = 'i'
    infected = [p_zero]
    recovered = []

    while len(infected) > 0:
        v = infected.pop(0)
        for u in G.neighbors(v):
            if states[u] == 's' and random() < T:
                states[u] = 'i'
                infected.append(u)
        states[v] = 'r'
        recovered.append(v)

    return len(recovered)

net_size = 400
mean_k = 5
p = mean_k / (net_size - 1)
T = 0.3
G = erdos_renyi_graph(net_size, p)

for i in range(100):
    print percolate(G, T)
```

Appendix B: Solution to Section 3.1

```
#!/usr/bin/python
# 0.) Import functions
from networkx import *
from random import *
from pylab import hist, show

# 1.) Define parameters
T = 0.9        # probability of transmission

# 2.) Construct network
G = Graph()
file = open('sexual_net.csv')
for edge in file:
  node1, node2 = edge.strip().split(',')
  G.add_edge(node1, node2)

net_size = float(G.order())
results = []

for i in range(100):
    # 3.) Store node states
    p_zero = choice(G.nodes()) # Randomly choose one node
    infected    = [p_zero]
    recovered   = []

    # 4.) Run the simulation
    while len(infected) > 0:
        v = infected.pop(0)
        for u in G.neighbors(v):
            # We need to make sure Node u is still susceptible
            if u not in infected and u not in recovered:
                if random() < T:
                    infected.append(u)
        recovered.append(v)

    # 5.) Tally the total fraction that got infected
    epi_size = len(recovered)/net_size
    results.append( epi_size )

hist(results, bins = 20)
show()
```