2017_hw2_HillaryMiller-Copy2

November 12, 2017

1 BST 267: Introduction to Social and Biological Networks (2017)

2 Homework 2

In this homework, you'll get practice with commonly used Python data types and with some common Python operations. Consult lecture notes for lectures 3 and 4 for help. If you're stuck, use Google or your favorite search engine to look for help.

2.1 Question 1: Lists

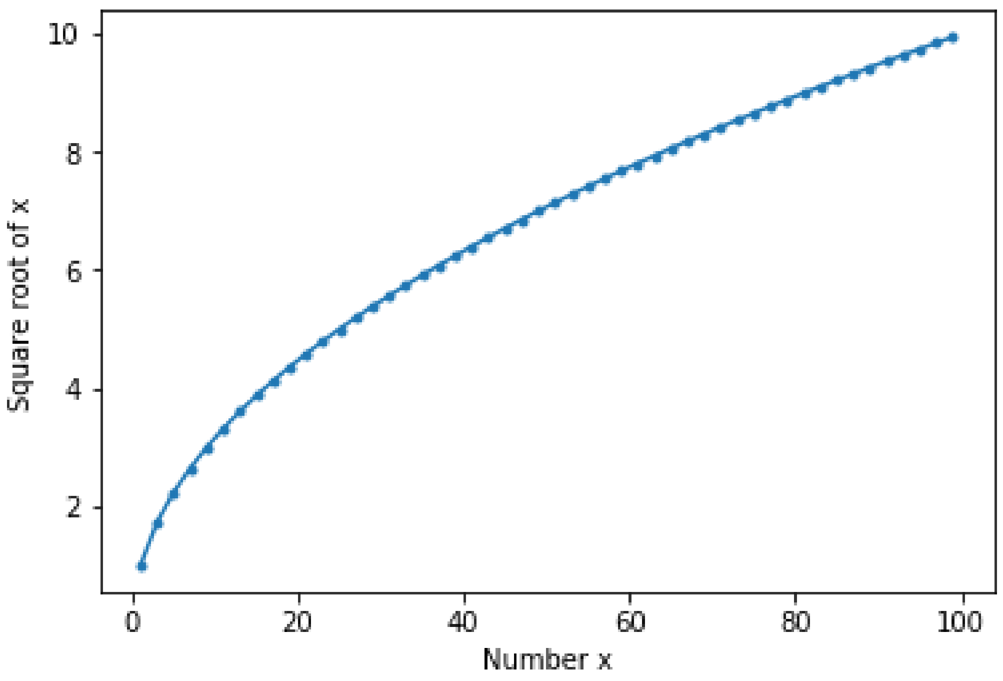
Create a list called `numbers` that contains numbers 1,3,5,...,99. In other words, the numbers in this list range from 1 to 99 in increments of 2. Compute the square root of each number on this list and store the square roots in a list called `roots`. Note that `numbers` and `roots` should have the same number of elements. Print out the elements of `numbers` and `roots`, i.e., a number and its square root. Once you're done, run the code underneath to create a plot.

```
In [10]: numbers = list(range(1,100,2))
         import numpy as np
         roots = list(np.sqrt(numbers))
         print(numbers, roots)
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49
```

```
In [11]: # run this code once you have `numbers` and `roots`
         import matplotlib.pyplot as plt
         % matplotlib notebook
         % matplotlib inline
         plt.plot(numbers, roots, '.-')
         plt.xlabel("Number x")
         plt.ylabel("Square root of x")
```

```
Out[11]: Text(0,0.5,'Square root of x')
```



2.2 Question 2: More lists

Python objects can be nested within other Python objects. Below we have an example of a list nested within another list. To create this list of lists, run the code below. Examine each line of code and make sure you understand what it does. Briefly, the code generates a so-called Barabasi-Albert graph and assigns it to variable `G`. The first loop is over all nodes of the graph and the second over all neighbors of each node. In the end, the list `neighbor_degrees` will contain as many lists are there are nodes in the graph, and each nested list will contain the degrees of the neighboring nodes of the given node.

```
In [12]: # run this code first
         import networkx as nx
         G = nx.barabasi_albert_graph(1000, 2, seed=123)
         node_degrees = []
         neighbor_degrees = []
         for node in G.nodes():
             node_degrees.append(G.degree(node))
             degrees = []
             for neighbor in G.neighbors(node):
                 degrees.append(G.degree(neighbor))
             neighbor_degrees.append(degrees)
```

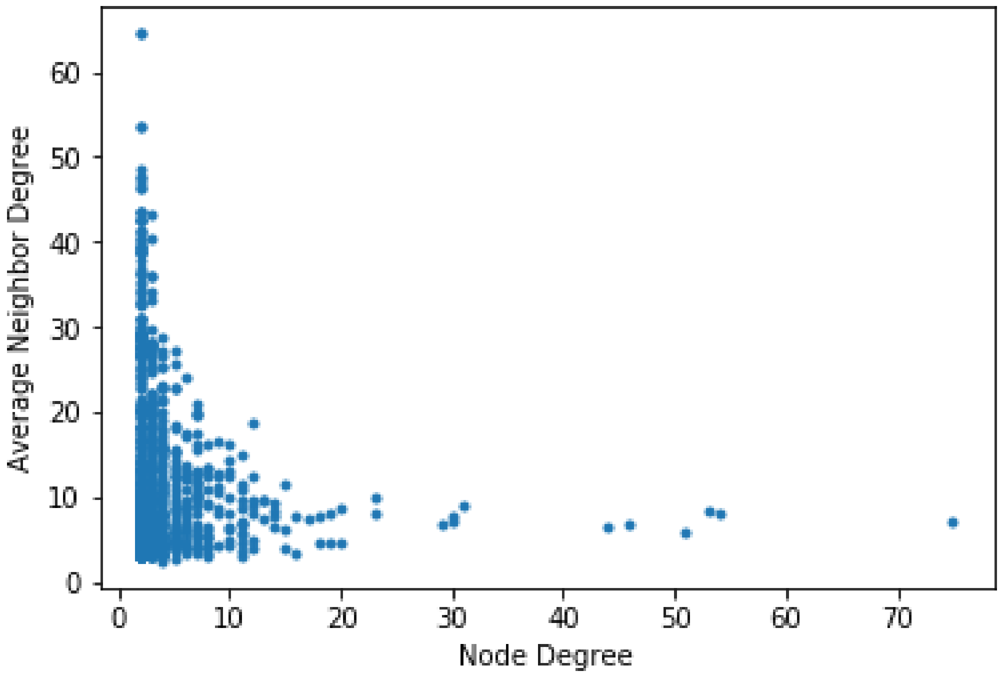
Now write the code to compute the average neighbor degree of each node, i.e., the average degree of the neighbors of each node. Use `plt.plot` to make a plot of degree on the x-axis and average neighbor degree on the y-axis. **Use `plt.xlabel` and `plt.ylabel` functions to label your plot.**

NOTE: Core Python language is lean, meaning that it has relatively few built-in functions, and the language becomes expressive when we use some of the many existing libraries, such as `networkx`. As a consequence, there is no built-in function for computing the mean of a sequence of numbers stored in a list. To compute the mean, you can simply add up the numbers in the list and then divide the result by the number of elements in your list. Alternatively, you can use the `mean` function in the `numpy` library.

```
In [13]: average_neighbor_degree = []
         for sublist in neighbor_degrees:
             average_neighbor_degree.append([np.mean(sublist)])
```

```
In [18]: plt.plot(node_degrees, average_neighbor_degree, '.')
         plt.xlabel("Node Degree")
         plt.ylabel("Average Neighbor Degree")
```

```
Out[18]: Text(0,0.5,'Average Neighbor Degree')
```



2.3 Question 3: Dictionaries and lists

There are several centrality measures that are used to quantify the centrality or importance of a node in a network. Degree, also known as degree centrality, is the most commonly used metric, but another centrality measure is so-called betweenness centrality. In NetworkX, if the network object is called `G`, then `G.degree()` returns a dictionary of node degrees: the keys are node IDs and the values are node degrees. You can compute betweenness centrality using the the `betweenness_centrality` function in the `nx` library. In other words, `nx.betweenness_centrality(G)` returns a dictionary of betweenness centrality values: the keys are node IDs and the values of node betweenness centrality values. (It's worth noting the following difference: the degree dictionary is available as a method of the network object `G` whereas the betweenness centrality dictionary is computed by calling a method or function in the `nx` library.) Generate two dictionaries, one for degree centrality and one for betweenness centrality. Then generate two lists called `x` and `y` corresponding to degree centrality and betweenness centrality, respectively, and plot them against one another. Make sure that the values in `x` and `y` for any given index are always for the same node as otherwise your plot will not be correct. **Use `plt.xlabel` and `plt.ylabel` functions to label your plot.**

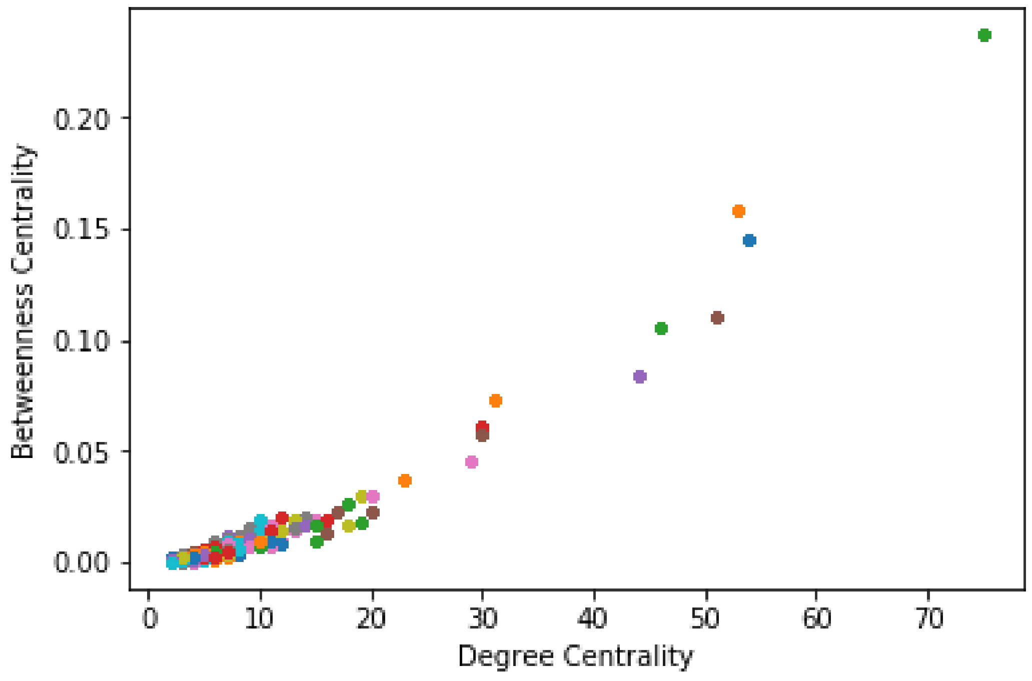
```
In [16]: betweenness =nx.betweenness_centrality(G)
         degree_centrality = dict(G.degree())

         x = []
         y = []

         for key in range(1000):
             x.append(list(degree_centrality.values()))
             y.append(list(betweenness.values()))


In [17]: plt.plot(x, y, '.-')
         plt.xlabel("Degree Centrality")
         plt.ylabel("Betweenness Centrality")

Out[17]: Text(0,0.5,'Betweenness Centrality')
```


Betweenness Centrality
Degree Centrality