project2

February 26, 2021

1 Project 2 - Exploring graph data

- Dave Miller
- CSCI 347

1.1 Part 1 - Think about the data

This dataset constists of around 4000 Wikipedia articles and the links between them. It is interesting to me because there is a game call Wikispeedia where you have to get from one topic (cheese) to another (europe) only by clicking links from page to page. It could be thought of as a sort of semantic graph for topics. I initially tried to use to whole graph (~4000 nodes) but ended up sampling 1000 and taking the largest connected component for the last part of this assignment since computing the average shortest path length and betweeness centrality were taking too long. For the nodes with the highest centrality measures, I expect more general topics that are one word and often searched. I think these nodes would get more links on more pages and therefore be most central. I could reasonably expect the graph to exhibit both small-world and scale-free behavior since this is a real world dataset that was created by many people over time.

[149]:
```
import numpy as np
import networkx as nx
import random
```

[8]:
```
PATH = 'datasets/wikispeedia_paths-and-graph/links.tsv'
```

[144]:
```
G = nx.read_edgelist('datasets/wikispeedia_paths-and-graph/links.tsv')
cc = max(nx.connected_components(G), key=len)
G = G.subgraph(cc)
```

[107]:
```
sample = random.sample(list(G.nodes), 1000)
sampleG = G.subgraph(sample)
smallG = sampleG.subgraph(max(nx.connected_components(sampleG), key=len))
```

[97]:
```
print(len(G.nodes()))
print(len(G.edges()))
```

```
4583
106529
```

1.2 Part 2 - Functions for graph analysis

[16]:
```
# Function to count number of vertices, assumes a tab separated file (.tsv)
def numVertices(edgelist):
    vertices = []
    with open(edgelist) as f:
        lines = f.readlines()

    for line in lines:
        v = line.split('\t')
        for vertex in v:
            vertex = vertex.strip('\n')
            if vertex not in vertices:
                vertices.append(vertex)

    return len(vertices)
```

[18]:
```
numVertices(PATH) == len(G.nodes)
```

[18]:
```
True
```

[67]:
```
# Function to get the degree of a given vertex (string)
def degree(edgelist, target):
    degrees = {}
    with open(edgelist) as f:
        lines = f.readlines()

    for line in lines:
        v = line.split('\t')
        v[1] = v[1].replace('\n', '')

        if v[0] not in degrees:
            degrees[v[0]] = [v[1]]
        else:
            if v[1] not in degrees[v[0]]:
                degrees[v[0]].append(v[1])

        if v[1] not in degrees:
            degrees[v[1]] = [v[0]]
        else:
            if v[0] not in degrees[v[1]]:
                degrees[v[1]].append(v[0])

    return len(degrees[target]), degrees
```

[69]:
```
degree(PATH, 'Dove')[0] == G.degree('Dove')
```

[69]:
```
True
```

[72]:
```
# Function to get clustering coefficient of a given vertex (string)
def clusterCoefficient(edgelist, target):
    d, degrees = degree(edgelist, target)
    neighbors = degrees[target]

    actualEdges = 0
    possibleEdges = len(neighbors) * (len(neighbors)-1)

    if d < 2:
        return 0

    for n in neighbors:
        for secondNighbor in degrees[n]:
            if secondNighbor in neighbors:
                actualEdges += 1

    return actualEdges / possibleEdges
```

[78]:
```
clusterCoefficient(PATH, 'Blokus') == nx.clustering(G, 'Blokus')
```

[78]:
```
True
```

[84]:
```
# Function to compute betweeness centrality for given vertex (this takes␣
 ↪forever)
def betweenessCentrality(edgelist, target):
    centrality = 0
    paths = nx.all_pairs_shortest_path(G)

    for path in paths:
        if target in path:
            centrality += 1

    return centrality / 4586
```

[86]:
```
nx.betweenness_centrality(G)['Dove']
```

[86]:
```
0.00015379497370143967
```

[147]:
```
# Function to compute the average shortest path length
def avgShortestPathLength(edgelist):
    nodes = list(G.nodes)
    avgPath = 0

    for node1 in nodes:
        for node2 in nodes:
            if node1 != node2:
                avgPath += nx.shortest_path_length(G, node1, node2)
    return avgPath / len(nodes)
```

[135]:
```
# Function that takes an edgelist and gives an adjacentcy matrix (symmetric for␣
↪undirected graph)
def adjacentcyMatrix(edgelist):
    nodes = list(G.nodes)
    adj = np.zeros((len(nodes), len(nodes)))

    with open(edgelist) as f:
        lines = f.readlines()

    for line in lines:
        v = line.split('\t')
        v[1] = v[1].strip('\n')

        adj[nodes.index(v[0])][nodes.index(v[1])] = 1
        adj[nodes.index(v[1])][nodes.index(v[0])] = 1

    return adj
```

[138]:
```
adj = adjacentcyMatrix(PATH)
print(adj[:10])
```

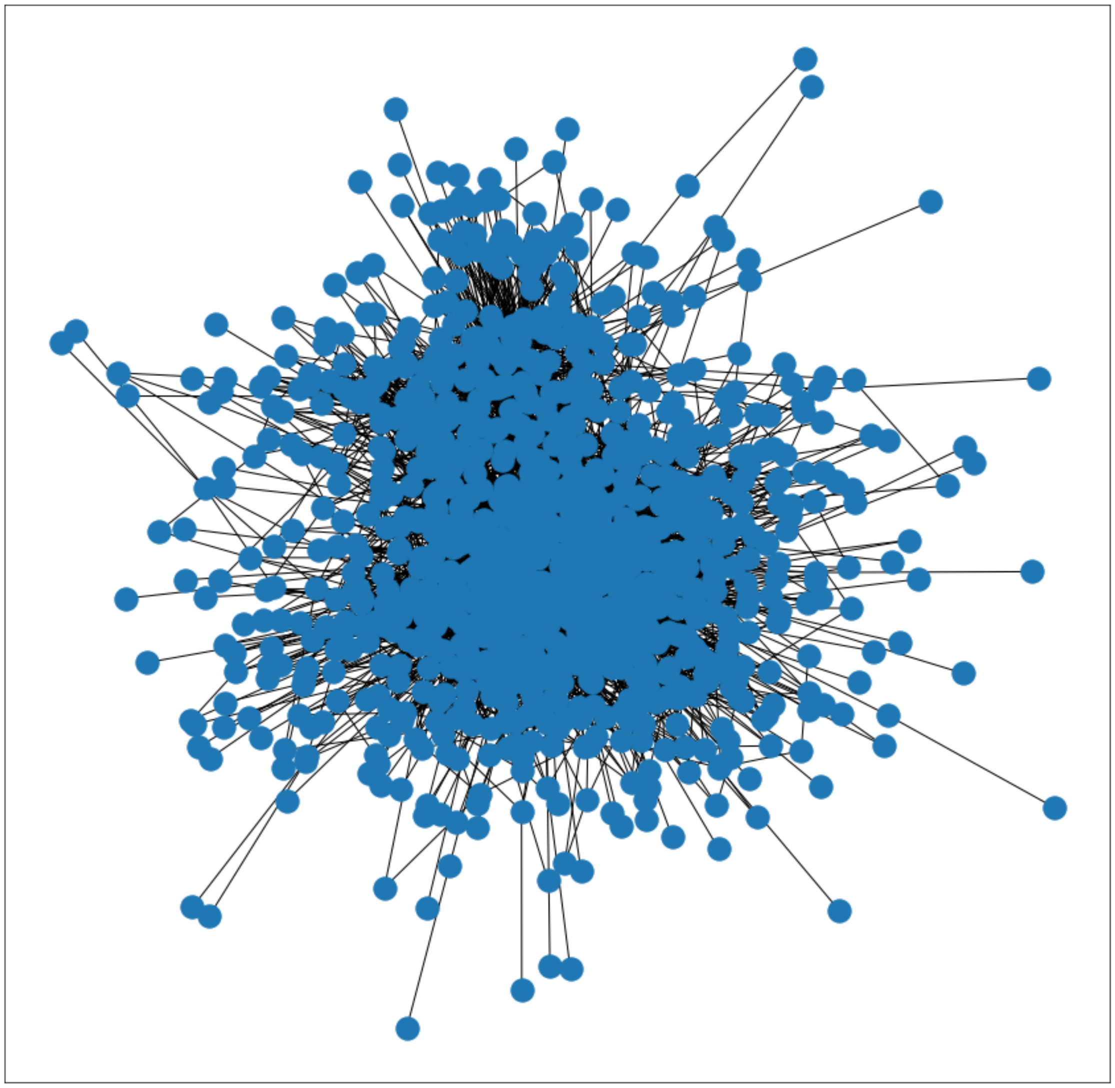
```
[[0. 1. 1. … 0. 0. 0.]
 [1. 0. 1. … 0. 0. 0.]
 [1. 1. 0. … 0. 0. 0.]
 …
 [1. 1. 1. … 0. 0. 0.]
 [1. 0. 0. … 0. 0. 0.]
 [1. 1. 0. … 0. 0. 0.]]
```

1.3 Part 3 - Analyze the graph data

[124]:
```
print(nx.info(smallG))
```

```
Name:
Type: Graph
Number of nodes: 949
Number of edges: 4832
Average degree:  10.1834
```

[108]:
```
import matplotlib.pyplot as plt
pos = nx.spring_layout(smallG)
plt.figure(figsize=(15,15))
nx.draw_networkx(smallG, pos=pos, with_labels=False)
```



```
[121]: from operator import itemgetter
       degrees = dict(nx.degree(smallG))
       result = dict(sorted(degrees.items(), key = itemgetter(1), reverse = True)[:10])
       result
```

```
[121]: {'Europe': 209,
        'Animal': 113,
        'Time_zone': 83,
        'French_language': 79,
        'New_York_City': 78,
        '20th_century': 76,
        'Binomial_nomenclature': 72,
        'New_Zealand': 70,
        'Brazil': 70,
        'Iran': 69}
```

[120]:
```
bc = dict(nx.betweenness_centrality(smallG))
result = dict(sorted(bc.items(), key=itemgetter(1), reverse=True)[:10])
result
```

[120]:
```
{'Europe': 0.23676253980686926,
 '20th_century': 0.059738215200946576,
 'Animal': 0.059240450101437366,
 'New_York_City': 0.05240446153384433,
 'French_language': 0.04492701568174148,
 'New_Zealand': 0.037453198365534485,
 'Brazil': 0.029922577443738795,
 'Binomial_nomenclature': 0.028038305236302557,
 'Oxygen': 0.027343403247141126,
 'Time_zone': 0.026958869776994958}
```

[119]:
```
cc = dict(nx.clustering(smallG))
result = dict(sorted(cc.items(), key=itemgetter(1), reverse=True)[:10])
result
```

[119]:
```
{'Hurricane_John_%282006%29': 1.0,
 'Konrad_Lorenz': 1.0,
 'Angel_shark': 1.0,
 'Milk_thistle': 1.0,
 'Constructivism_%28art%29': 1.0,
 'African_clawed_frog': 1.0,
 'London_Zoo': 1.0,
 'Tropical_Storm_Franklin_%282005%29': 1.0,
 'The_Catlins': 1.0,
 'Hurricane_Gloria': 1.0}
```

[118]:
```
ec = dict(nx.eigenvector_centrality(smallG))
result = dict(sorted(ec.items(), key=itemgetter(1), reverse=True)[:10])
result
```

[118]:
```
{'Europe': 0.3046768158405792,
 'Time_zone': 0.19732243282127068,
 'List_of_countries': 0.16560771606125035,
 'Iran': 0.158675533006163,
 'List_of_sovereign_states': 0.15620844913938087,
 'Albania': 0.15129051561806178,
 'List_of_circulating_currencies': 0.14871922922838582,
 'Syria': 0.14629795648390087,
 'Bulgaria': 0.14581537632238734,
 'Brazil': 0.14338483313377196}
```

```
[117]: pr = dict(nx.pagerank(smallG))
       result = dict(sorted(pr.items(), key=itemgetter(1), reverse=True)[:10])
       result
```

```
[117]: {'Europe': 0.018927159193490452,
        'Animal': 0.012211185302670466,
        'Binomial_nomenclature': 0.008090859122843661,
        'New_York_City': 0.007444707832002265,
        'French_language': 0.007333931120706987,
        '20th_century': 0.006981096459513071,
        'Bird': 0.006563413821991096,
        'Time_zone': 0.006512277879544247,
        'New_Zealand': 0.006135700729910397,
        'Brazil': 0.005833612139067835 5}
```

It appears that most of the highly ranked nodes are the same with the exception of clustering coefficient nodes. This could be because there are a lot of ties for 1.0. For this, tiebreaking could involve looking at the nodes that have more neighbors. Europe is at the top of the list for all of the other metrics. Overall, these lists make sense in that they differ slightly because they are different metrics.

```
[125]: np.log(len(smallG.nodes))
```

```
[125]: 6.855408798609928
```

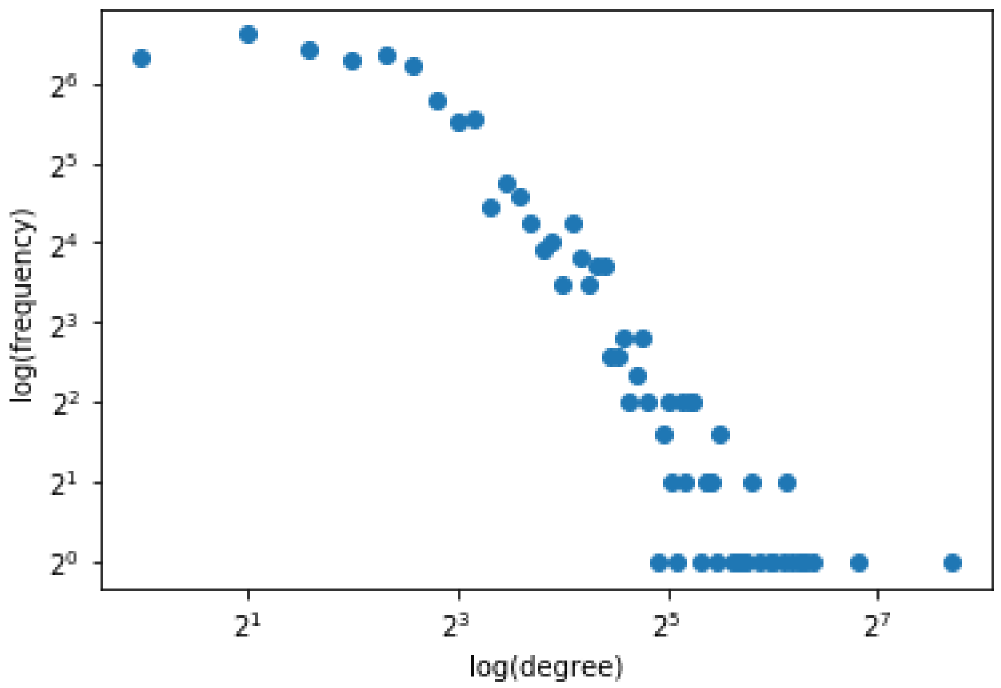
```
[126]: nx.average_shortest_path_length(smallG)
```

```
[126]: 3.1786290699070308
```

This graph does exhibit small-world behavior. This is defined as the average shortest path length scaling logrithmically with the number of nodes. As you can see, the average shortest path length is rather small for such a large graph and the numbers above are proportional.

```
[128]: degrees = dict(nx.degree(smallG)).values()
       unique, count = np.unique(list(degrees), return_counts=True)
       plt.scatter(unique, count)
       plt.xscale('log', basex=2)
       plt.yscale('log', basey=2)
       plt.xlabel('log(degree)')
       plt.ylabel('log(frequency)')
```

```
[128]: Text(0, 0.5, 'log(frequency)')
```



It appears that this graph also exhibits the scale-free property, as shown by the power law distribution between degree and frequency of that degree. This shows that there are a few nodes with very high degree, but most nodes have a lower degree.