Data Science in Practice-Assignment 2

March 31, 2019

1 Problem Set 2

Imports and definition of helper fucntions such as plots

In [1]:
```
!python -m pip install --upgrade pip
!pip install python-louvain
```

```
Collecting pip
  Downloading https://files.pythonhosted.org/packages/d8/f3/413bab4ff08e1fc4828dfc59996d721917d
    100% || 1.4MB 6.6MB/s ta 0:00:011
Installing collected packages: pip
  Found existing installation: pip 18.1
    Uninstalling pip-18.1:
      Successfully uninstalled pip-18.1
Successfully installed pip-19.0.3
Collecting python-louvain
  Downloading https://files.pythonhosted.org/packages/96/b2/c74bb9023c8d4bf94f5049e3d705b3064c0
Requirement already satisfied: networkx in /Users/genmur/anaconda3/lib/python3.6/site-packages
Requirement already satisfied: decorator>=4.1.0 in /Users/genmur/anaconda3/lib/python3.6/site-p
Building wheels for collected packages: python-louvain
  Building wheel for python-louvain (setup.py) ... done
  Stored in directory: /Users/genmur/Library/Caches/pip/wheels/f9/74/a9/14f051b00dddd46d71529dl
Successfully built python-louvain
Installing collected packages: python-louvain
Successfully installed python-louvain-0.13
```

In [2]:
```
!pip install powerlaw
```

```
Collecting powerlaw
  Downloading https://files.pythonhosted.org/packages/d5/4e/3ceab890fafff8e78a5fd7f5340c232c38l
Requirement already satisfied: scipy in /Users/genmur/anaconda3/lib/python3.6/site-packages (f
Requirement already satisfied: numpy in /Users/genmur/anaconda3/lib/python3.6/site-packages (f
Requirement already satisfied: matplotlib in /Users/genmur/anaconda3/lib/python3.6/site-package
Requirement already satisfied: mpmath in /Users/genmur/anaconda3/lib/python3.6/site-packages (
Requirement already satisfied: cycler>=0.10 in /Users/genmur/anaconda3/lib/python3.6/site-packa
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /Users/genmur/anacon
Requirement already satisfied: python-dateutil>=2.1 in /Users/genmur/anaconda3/lib/python3.6/si
```

```
Requirement already satisfied: pytz in /Users/genmur/anaconda3/lib/python3.6/site-packages (fro
Requirement already satisfied: six>=1.10 in /Users/genmur/anaconda3/lib/python3.6/site-packages
Requirement already satisfied: kiwisolver>=1.0.1 in /Users/genmur/anaconda3/lib/python3.6/site-
Requirement already satisfied: setuptools in /Users/genmur/anaconda3/lib/python3.6/site-package
Building wheels for collected packages: powerlaw
  Building wheel for powerlaw (setup.py) ... done
  Stored in directory: /Users/genmur/Library/Caches/pip/wheels/e0/27/02/08d0e2865072bfd8d7c655e
Successfully built powerlaw
Installing collected packages: powerlaw
Successfully installed powerlaw-1.4.6
```

In [3]:
```
import sys
import numpy as np
import pandas as pd
import seaborn as sns
import networkx as nx
from networkx.algorithms.community import k_clique_communities
import community as community
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.colors import ListedColormap, LinearSegmentedColormap
import collections
import operator
import powerlaw as pl
from multiprocessing import Pool
import scipy.sparse as sp
import itertools
import pprint
import matplotlib.style as style
import random, time
from matplotlib.colors import rgb2hex
import io
sns.set()
style.use('Solarize_Light2')
sns.set_style("whitegrid")
```

In [4]:
```
def nx_hist(x, bins, normed = False, xscale = 'linear', yscale = 'linear', use_log = Fa
    density = None
    if(normed):
        density = 1
        ylabel = 'Probability'
    fig = plt.figure(figsize=(15,6))
    plt.hist(x, bins=bins, normed=density, log=use_log)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    #plt.xscale(yscale)
    plt.yscale(yscale)
```

```python
    plt.title(title)
    plt.grid(True)
    plt.show()

def print_in_degree_dist(G):
    degree_sequence = sorted([d for n, d in G.in_degree()], reverse=True)  # degree se
    degreeCount = collections.Counter(degree_sequence) # degree count
    print("Empirical Degree Distribution for network: "+G.name)
    nx_hist(degree_sequence,len(degreeCount))
    nx_hist(degree_sequence,len(degreeCount),yscale='log')
   # nx_hist(degree_sequence,len(degreeCount),yscale='log',use_log=True)

def print_out_degree_dist(G):
    degree_sequence = sorted([d for n, d in G.out_degree()], reverse=True)  # degree s
    degreeCount = collections.Counter(degree_sequence) # degree count
    print("Empirical Degree Distribution for network: "+G.name)
    nx_hist(degree_sequence,len(degreeCount))
    nx_hist(degree_sequence,len(degreeCount),yscale='log')
    #nx_hist(degree_sequence,len(degreeCount),yscale='log',use_log=True)

def print_degree_dist(G):
    degree_sequence = sorted([d for n, d in G.degree()], reverse=True)  # degree seque
    degreeCount = collections.Counter(degree_sequence) # degree count
    print("Empirical Degree Distribution for network: "+G.name)
    nx_hist(degree_sequence,len(degreeCount))
    nx_hist(degree_sequence,len(degreeCount),yscale='log')
    #nx_hist(degree_sequence,len(degreeCount),yscale='log',use_log=True)

def plot_centrality(x,y,xlabel,ylabel,xmin,ymin,xmax,ymax):
    plt.figure(figsize=(15,9))
    plt.scatter(np.array(list(x.values())),np.array(list(y.values())))
    plt.xlim(xmin,xmax)
    #plt.ylim(ymin,ymax)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.title("Centrality Comparison for the selected Measures")
    plt.show()

def cc_cdf(cc_dict):
    plt.figure(figsize=(15,10))
    plt.plot(sorted(cc_dict.values()),np.arange(0,len(cc_dict))/len(cc_dict))
    plt.xlabel("Local Clustering Coefficient (c)")
    plt.ylabel("P(x<=c)")
    plt.title("Empirical Clustering Coefficient CDF")
    plt.show()
    return

def w_degree_centrality(G):
```

```
    keys = list(nx.get_edge_attributes(G,'weight').keys())
    tot = np.array(list(nx.get_edge_attributes(G,'weight').values())).sum()
    vals = list(np.array(list(dict(G.degree(weight='weight')).values())/tot))
    return dict(zip(keys, vals))

def in_degree_centrality(G,weights):
    values_list = list(G.edges.data(weights))
    val_num = np.array([values_list[i][2] for i in range(9651)])
    vals = list(np.array(list(dict(G.in_degree(weight=weights)).values()))/np.sum(val_r
    keys = list(idc_G.keys())
    return dict(zip(keys, vals))

def out_degree_centrality(G,weights):
    values_list = list(G.edges.data(weights))
    val_num = np.array([values_list[i][2] for i in range(9651)])
    vals = list(np.array(list(dict(G.out_degree(weight=weights)).values()))/np.sum(val_
    keys = list(idc_G.keys())
    return dict(zip(keys, vals))

def print_node_data(G,max):
    i = 0
    for n, d in G.nodes(data=True):
        if i < max:
            print(n,d)
        i = i + 1
    return

def ret_nodelist(DM_G,G_GCC,N):

    G_GCC_degs = dict(sorted(G_GCC.degree, key=lambda x: x[1], reverse=True))
    G_GCC_min_degs = dict([(key, value) for key,value in G_GCC_degs.items() if (value =

    #G_GCC_top10_names = [key for key, value in G_GCC_degs.items()][0:N]
    #G_GCC_top10 = [value for key, value in G_GCC_degs.items()][0:N]

    print("Top 10 highest degree nodes and values:")
    i = 0
    for k, v in G_GCC_degs.items():
        if i < N:
            print(k,v)
        i = i + 1
    #return [G_GCC_top10_names, G_GCC_top10]
```

```
In [5]: G = nx.read_edgelist('SMS-network.txt')
```

Basic Network statistics and degree distribution

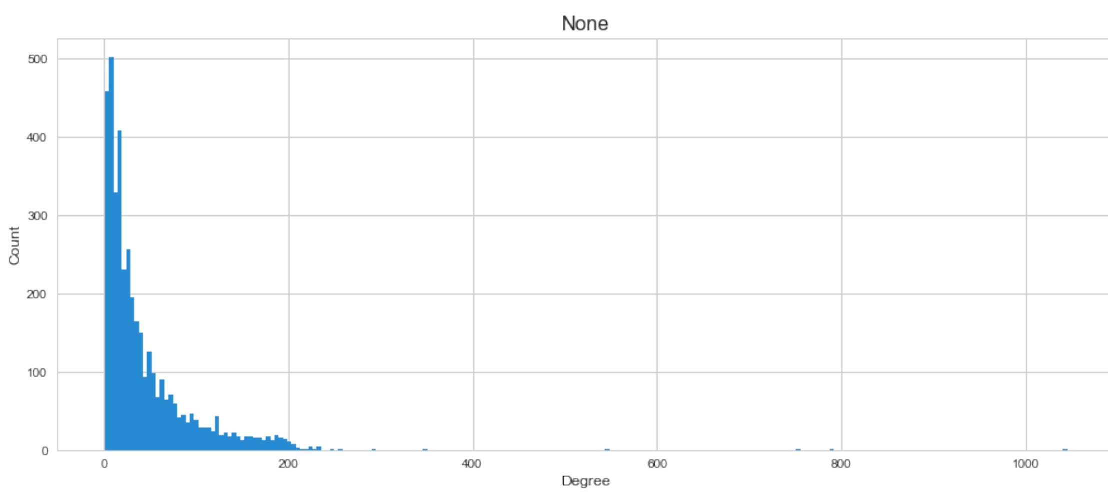
```
In [6]: print(nx.info(G))
```

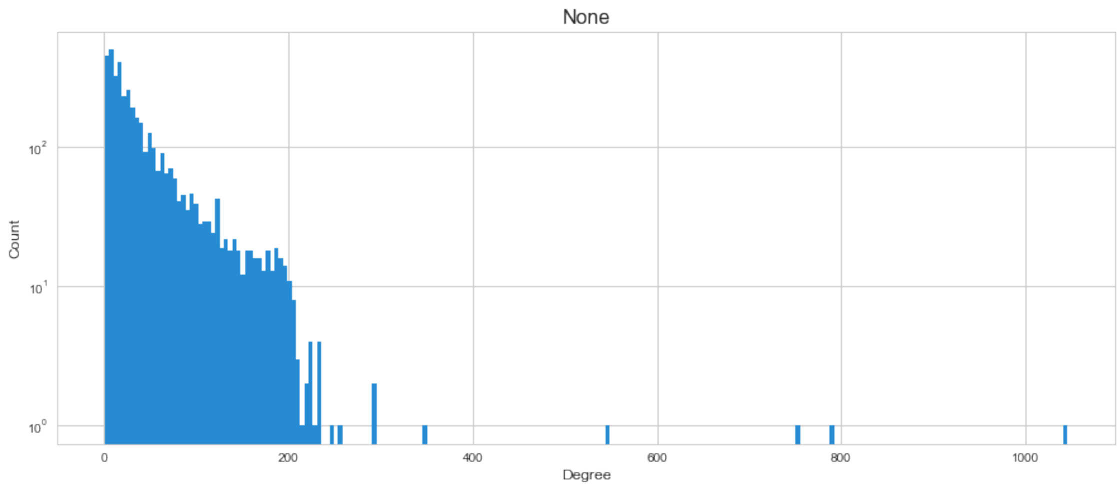
```
Name: 
Type: Graph
Number of nodes: 4039
Number of edges: 88234
Average degree:  43.6910
```

```
In [7]: G_GCC = max(nx.connected_component_subgraphs(G), key=len)
        #G_game_GCC = max(nx.connected_component_subgraphs(G_game), key=len)
        #G_game_uw_GCC = max(nx.connected_component_subgraphs(G_game_uw), key=len)
        #nx.set_node_attributes(G_game_uw_GCC,cat_dict)
```

```
In [8]: print(nx.info(G_GCC))
        print_degree_dist(G_GCC)
```

```
Name: 
Type: Graph
Number of nodes: 4039
Number of edges: 88234
Average degree:  43.6910
Empirical Degree Distribution for network:
```



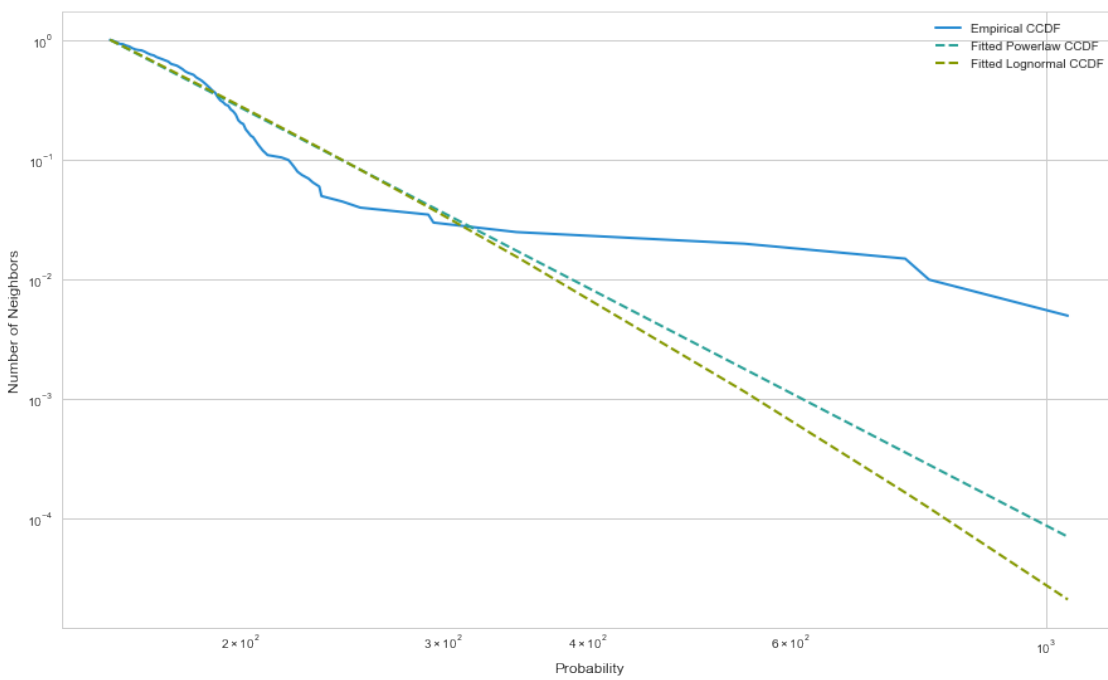



```
In [9]: def calculate_powerlaw(G_GCC):
            G_deg = sorted([d for n, d in G_GCC.degree()], reverse=True)
            print('{:^30}\t{:^7}'.format('Minimum number of neighbors:', G_deg[0]))
            print('{:^30}\t{:^7}'.format('Maximum number of neighbors:', G_deg[-1]))
            print('{:^30}\t{:^7.3f}'.format('Mean number of neighbors:', sum(G_deg)/len(G_deg)

            fit_G = pl.Fit(G_deg)

            print("Power Law vs Exponential:")
            R, p = fit_G.distribution_compare('power_law', 'exponential')
            print('R = {:.5f}'.format(R))
            print('p = {:.5f}'.format(p))

            print("Power Law vs Lognormal:")
            R, p = fit_G.distribution_compare('power_law', 'lognormal')
            print('R = {:.5f}'.format(R))
            print('p = {:.5f}'.format(p))

            print("Lognormal vs Exponential:")
            R, p = fit_G.distribution_compare('lognormal_positive', 'exponential')
            print('R = {:.5f}'.format(R))
            print('p = {:.5f}'.format(p))

            plt.figure(figsize=(15,9))
            fig = fit_G.plot_ccdf(linewidth=2, label='Empirical CCDF')
            fit_G.power_law.plot_ccdf(linestyle='--',ax=fig, label='Fitted Powerlaw CCDF')
            fit_G.lognormal_positive.plot_ccdf(linestyle='--',ax=fig, label='Fitted Lognormal C
            plt.xlabel("Probability")
            plt.ylabel("Number of Neighbors")
            plt.legend()
            plt.show()
```

```
            print('Fitted parameter alpha of the power law distribution: {:.3f}'.format(fit_G.p
            print('Standard error of alpha: {:.3f}'.format(fit_G.power_law.sigma))
```

In [10]: calculate_powerlaw(G_GCC)

```
 Minimum number of neighbors:          1045
 Maximum number of neighbors:             1
  Mean number of neighbors:           43.691
Power Law vs Exponential:
R = 23.37320
p = 0.06533
Power Law vs Lognormal:


Calculating best minimal value for power law fit
/Users/genmur/anaconda3/lib/python3.6/site-packages/powerlaw.py:700: RuntimeWarning: invalid va
  (Theoretical_CDF * (1 - Theoretical_CDF))
/Users/genmur/anaconda3/lib/python3.6/site-packages/powerlaw.py:1605: RuntimeWarning: invalid v
  CDF = CDF/norm
'nan' in fit cumulative distribution values.
Likely underflow or overflow error: the optimal fit for this distribution gives values that are


R = -0.92839
p = 0.18551
Lognormal vs Exponential:
R = 21.77665
p = 0.05531
```



```
Fitted parameter alpha of the power law distribution: 5.993
Standard error of alpha: 0.351
```

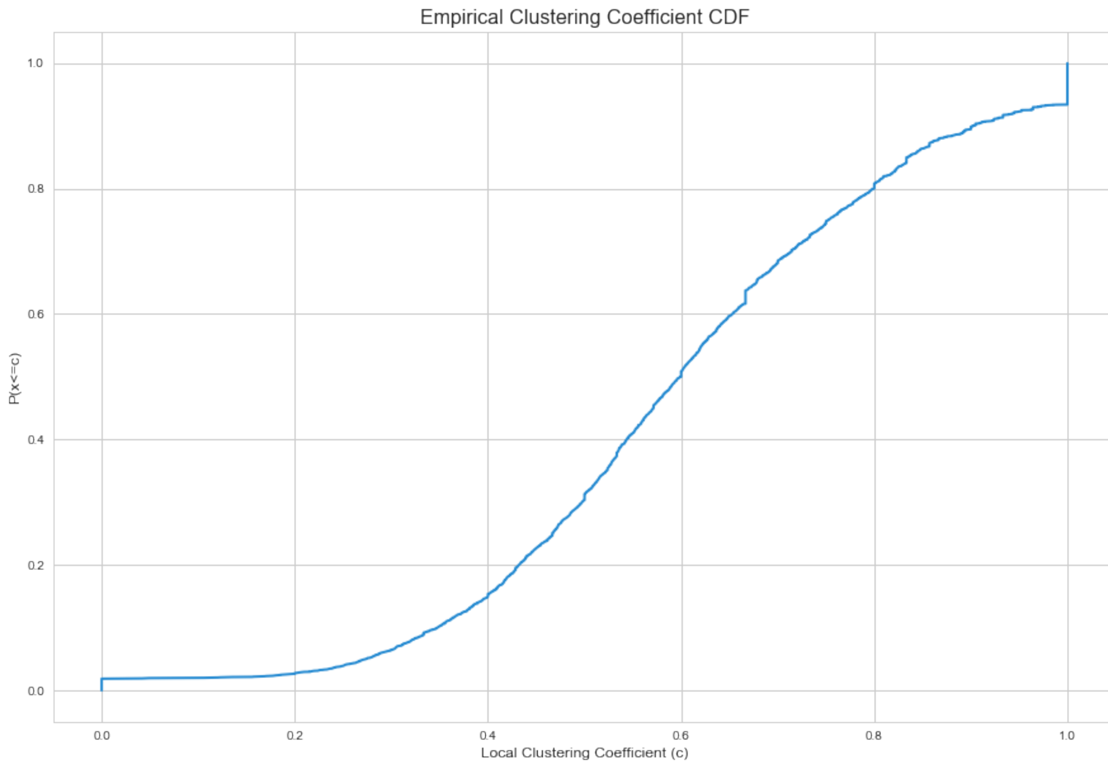
Exponential, Lognormal, and powerlaw distributions were compared through distribution distance. The best fit for the network would appear to be a lognormal distribution, but as we can see the powerlaw distribution is close enough to the true TCDF (and indeed the R coefficient is only 0.59 in favor of the lognormal) to asset that the network follows a power law distribution.

This implies that the network does exhibit some scale free properties, which is relatively expected of networks of this type where their embedding is extremely high dimensional, and no prior structural constraints other than the network topology are placed: users can freely navigate from any page to any other page.

Clustering and Other Network Figures

```
In [11]: def calculate_clustering(G_GCC):
             clust_G = nx.clustering(G_GCC) #network clustering coeff for all nodes
             C_G = float(sum(clust_G.values())/len(clust_G)) #average clustering coefficient
             T_G = nx.transitivity(G_GCC) #transitivity
             D_G = nx.density(G_GCC) #density
             cc_cdf(clust_G) #clustering coefficient cdf
             print("Average Clustering:",C_G)
             print("Overall Clustering (Transitivity):",T_G)
             print("Network Density",D_G)
             print("Avg Clustering to Density Ratio:",C_G/D_G)

             df = pd.DataFrame(columns=['Density','Overall clust','Average clust','Avg Cluster
             df = df.append({'Density':D_G,'Overall clust': T_G, 'Average clust': C_G,'Avg Clus
             return df
```

```
In [12]: df = calculate_clustering(G_GCC)
         display(df)
```




```
Average Clustering: 0.6055467186200865
Overall Clustering (Transitivity): 0.5191742775433075
Network Density 0.010819963503439287
Avg Clustering to Density Ratio: 55.96568957257614
```

	Density	Overall clust	Average clust	Avg Clustering to Density Ratio
0	0.01082	0.519174	0.605547	55.96569

As can be seen from the table, the network is sparse: the density is 0.01 << 1. However, the network does seem to exhibit a high clustering property in that the average clustering is 55 times the density (where the density corresponds to the would-be average clustering if the network were randomly generated).

Given that the CDF tails off towards the end, this implies that the nodes in the network have a rather wide range of clustering coefficients, which given the context is relatively unsurprising.

Note: The Floyd Warshall Matrix takes a LONG time to compute (i7 8700k overclocked), and weighs 500mb approximately. Loading it in compressed is a whole other deal.

```
In [13]: def calculate_distances(G_GCC,flag,filename):
             if flag:
                 DM_G = nx.floyd_warshall_numpy(G_GCC)
                 np.savez_compressed(filename, DM_G.astype(int))
```

```
            else:
                with np.load(filename) as data:
                    DM_G = data['arr_0']
            dia_G = np.max(DM_G) # network diameter
            avg_spl_G = np.sum(DM_G)/(len(G_GCC.nodes)*(len(G_GCC.nodes)-1)) #avg shortest pa
            print("Diameter:",dia_G)
            print("Average Shortest Path Length:",avg_spl_G)

            df = pd.DataFrame(columns=['Diameter','Average Shortest Path Length'])
            df = df.append({'Diameter':dia_G,'Average Shortest Path Length': avg_spl_G,}, igno
            display(df)
            return DM_G
```

In [14]:
```
DM_G = calculate_distances(G_GCC,0,'dm_mat.npz')
```

```
Diameter: 8
Average Shortest Path Length: 3.6925068496963913
```

	Diameter	Average Shortest Path Length
0	8.0	3.692507

In [15]:
```
ret_nodelist(DM_G,G_GCC,10)
```

```
Top 10 highest degree nodes and values:
107 1045
1684 792
1912 755
3437 547
0 347
2543 294
2347 291
1888 254
1800 245
1663 235
```

2 3. Centrality Measures

In [16]:
```
def print_top_10(D,G): #where G is the graph and D is the centrality measure dict
    df = pd.DataFrame(columns=['Node ID','Centrality Score'])
    #nodeDict = dict(G.nodes(data=True))
    D_sorted = dict(sorted(D.items(), key=operator.itemgetter(1), reverse=True))
    D_list = list(D_sorted)
    L = D_list[0:10]
    #top_10_dict = {k:nodeDict[k] for k in L if k in nodeDict}
    top_10_vals = np.array(list(D_sorted.values())[:10])
```

```
    #print(top_10_dict)
    i = 0
    for l in L:
        df = df.append({'Node ID': l, 'Centrality Score':top_10_vals[i]}, ignore_inde
        i = i+1
    display(df)
```

In [17]:
```
def store_top_200(D,G):
    df = pd.DataFrame(columns=['Node ID','Centrality Score'])
    D_sorted = dict(sorted(D.items(), key=operator.itemgetter(1), reverse=True))
    D_list = list(D_sorted)

    S_top200 = D_list[0:200]
    S_top200_value = np.array(list(D_sorted.values())[:200])

    i = 0
    for l in S_top200:
        df = df.append({'Node ID': l, 'Centrality Score':S_top200_value[i]}, ignore_in
        i = i+1
    return df
```

In [18]:
```
#Networkx: Parallel Implementation of Betweenness Centrality
def chunks(l, n):
    """Divide a list of nodes `l` in `n` chunks"""
    l_c = iter(l)
    while 1:
        x = tuple(itertools.islice(l_c, n))
        if not x:
            return
        yield x


def _betmap(G_normalized_weight_sources_tuple):
    """Pool for multiprocess only accepts functions with one argument.
    This function uses a tuple as its only argument. We use a named tuple for
    python 3 compatibility, and then unpack it when we send it to
    `betweenness_centrality_source`
    """
    return nx.betweenness_centrality_source(*G_normalized_weight_sources_tuple)


def betweenness_centrality_parallel(G, processes=None):
    """Parallel betweenness centrality  function"""
    p = Pool(processes=processes)
    node_divisor = len(p._pool) * 4
    node_chunks = list(chunks(G.nodes(), int(G.order() / node_divisor)))
    num_chunks = len(node_chunks)
```

```
        bt_sc = p.map(_betmap,
                      zip([G] * num_chunks,
                          [True] * num_chunks,
                          [None] * num_chunks,
                          node_chunks))

        # Reduce the partial solutions
        bt_c = bt_sc[0]
        for bt in bt_sc[1:]:
            for n in bt:
                bt_c[n] += bt[n]
        return bt_c
```

```
In [19]: bc_G = betweenness_centrality_parallel(G_GCC)
```

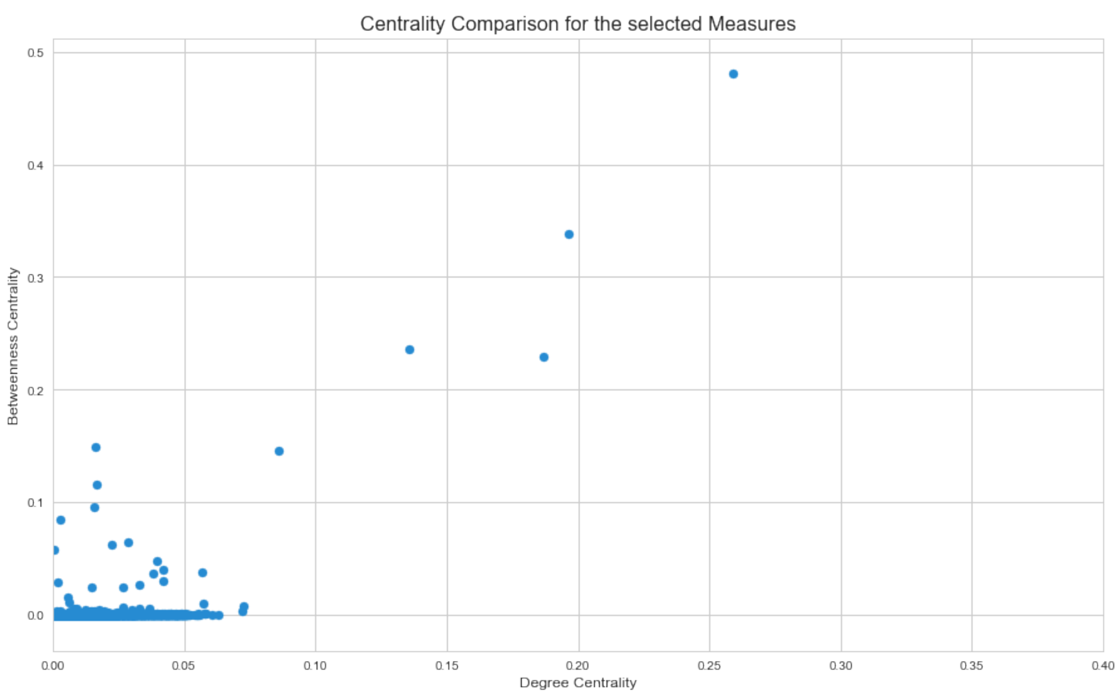
```
In [20]: dc_G = nx.degree_centrality(G_GCC)
         pr_G = nx.pagerank(G_GCC)
```

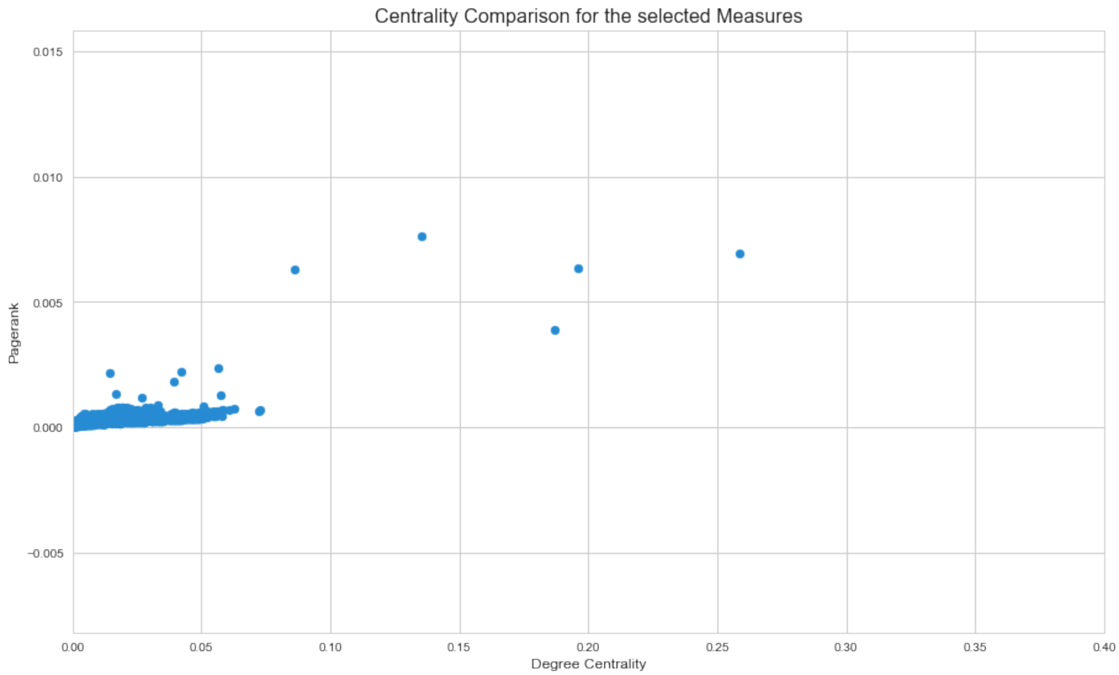
```
In [21]: ev_G = nx.eigenvector_centrality_numpy(G_GCC)
         katz_G = nx.katz_centrality_numpy(G_GCC)
```

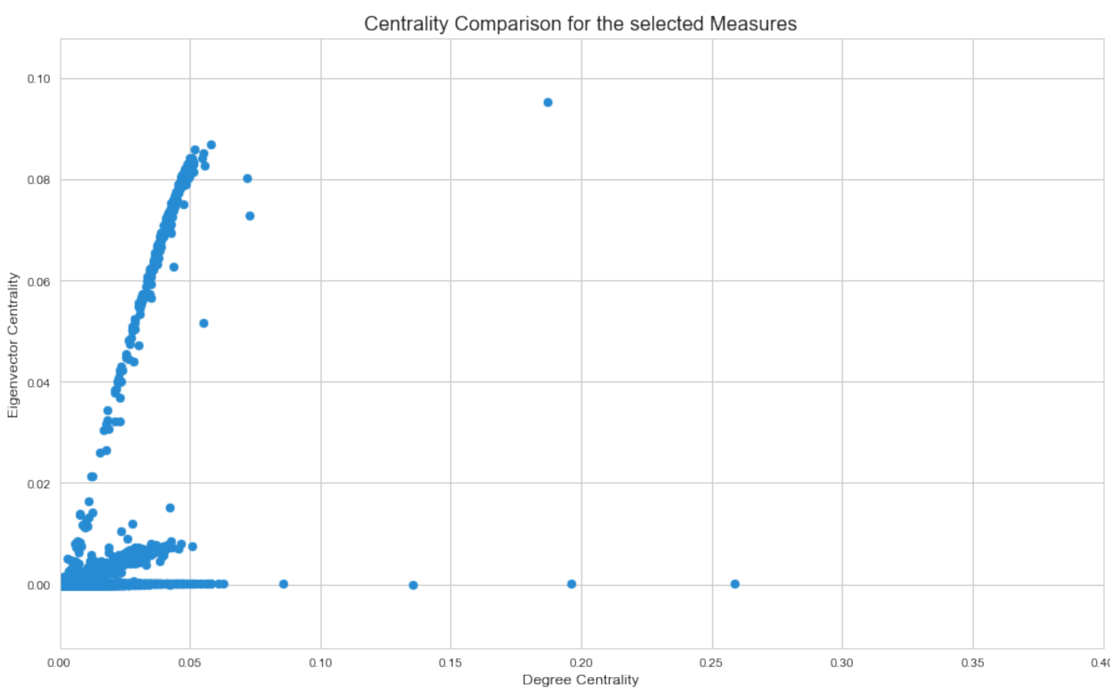
```
In [22]: cc_G = nx.closeness_centrality(G_GCC)
```

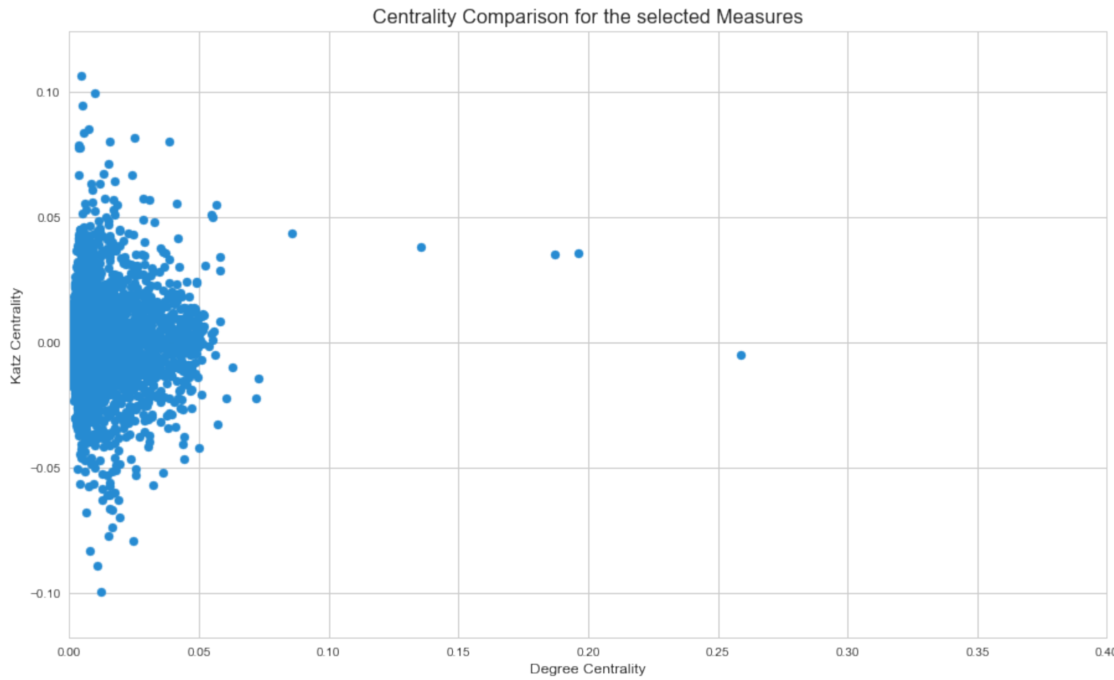
```
In [23]: print("Degree vs Unweighted Centralities:")
         plot_centrality(dc_G,bc_G,"Degree Centrality","Betweenness Centrality",0,0,0.4,0.02)
         plot_centrality(dc_G,pr_G,"Degree Centrality","Pagerank",0,0,0.4,0.02)
         plot_centrality(dc_G,ev_G,"Degree Centrality","Eigenvector Centrality",0,0,0.4,0.02)
         plot_centrality(dc_G,katz_G,"Degree Centrality","Katz Centrality",0,0,0.4,0.02)
```

```
Degree vs Unweighted Centralities:
```



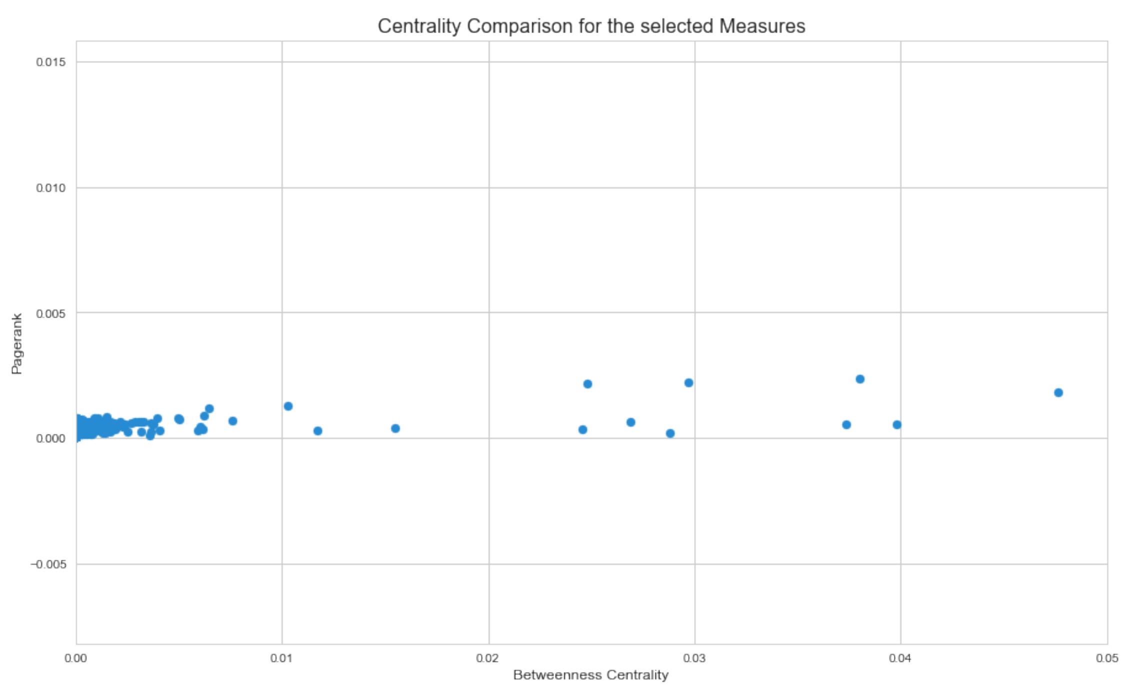

Centrality Comparison for the selected Measures
Pagerank
0.015
0.010
0.005
0.000
−0.005
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
Degree Centrality

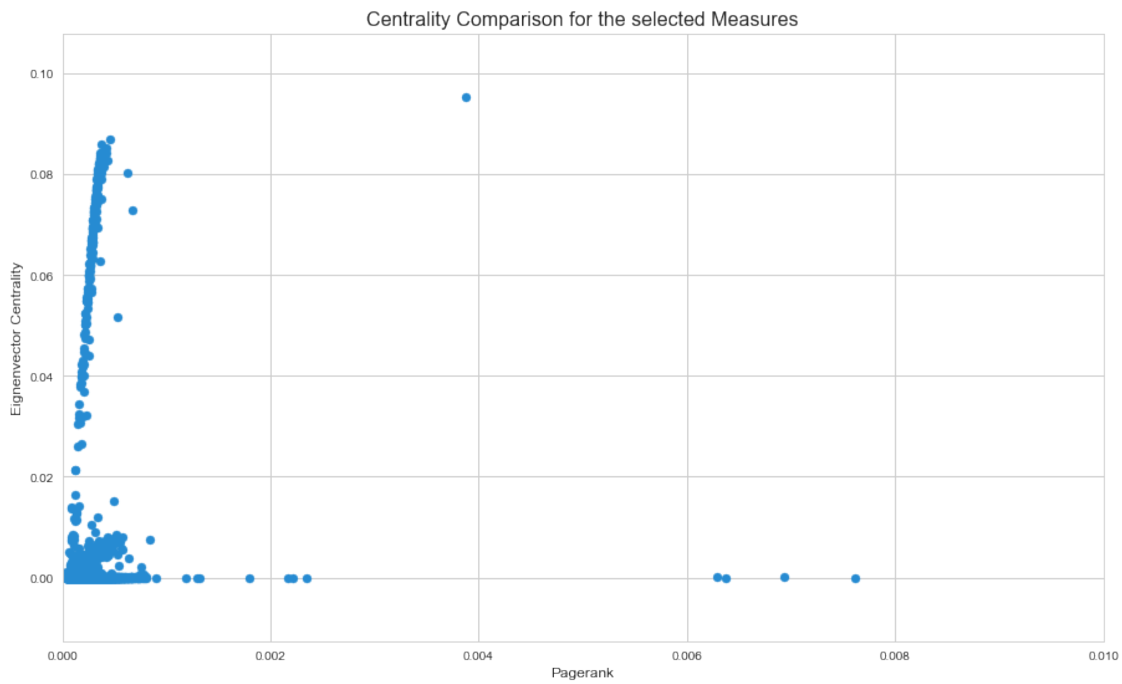

Centrality Comparison for the selected Measures
Eigenvector Centrality
0.10
0.08
0.06
0.04
0.02
0.00
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
Degree Centrality

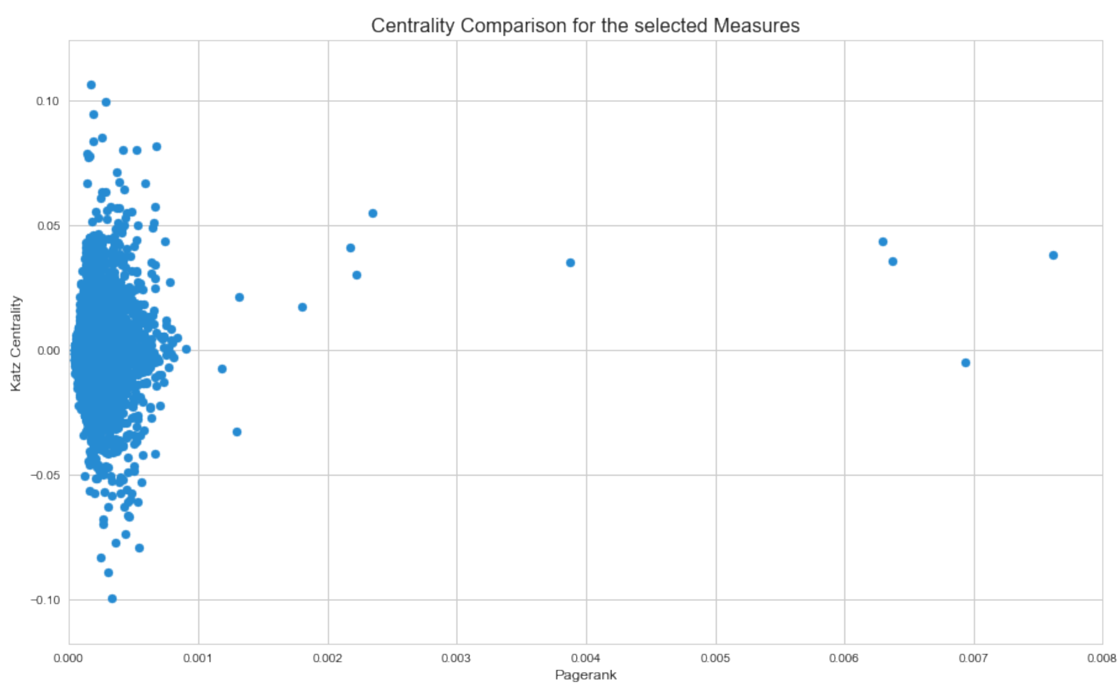


In [24]:
```
print("Centralitiy Comparison:")
plot_centrality(bc_G,pr_G,"Betweenness Centrality","Pagerank",0,0,0.05,0.02)
plot_centrality(pr_G,ev_G,"Pagerank","Eignenvector Centrality",0,0,0.01,0.02)
plot_centrality(pr_G,katz_G,"Pagerank","Katz Centrality",0,0,0.008,0.02)
```

```
Centralitiy Comparison:
```



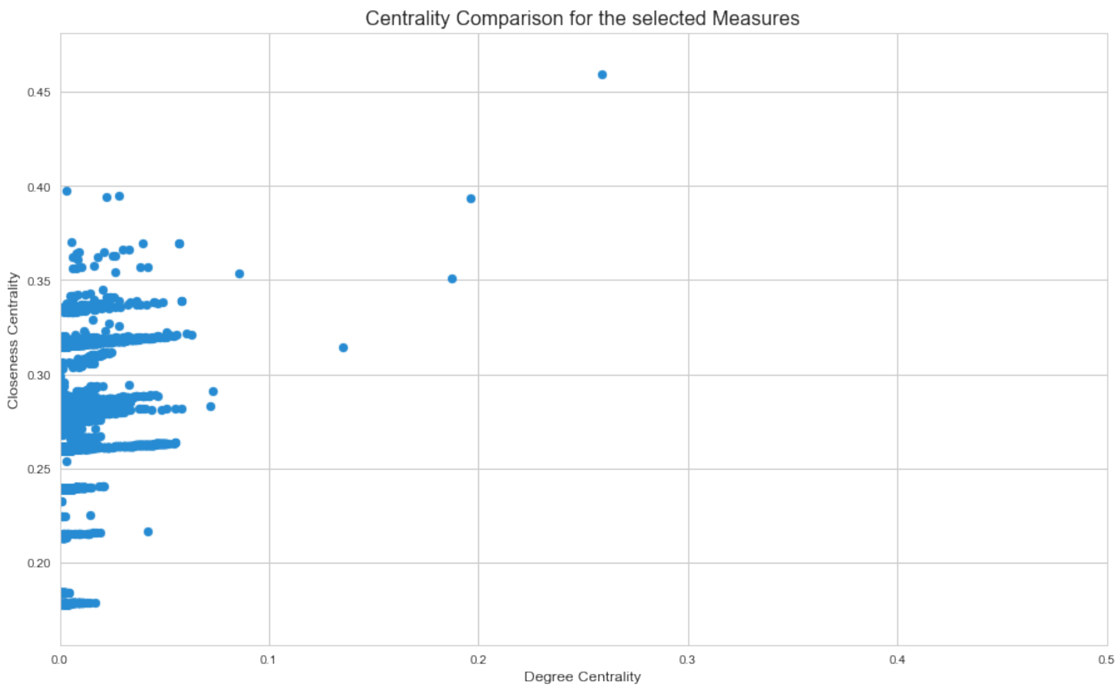


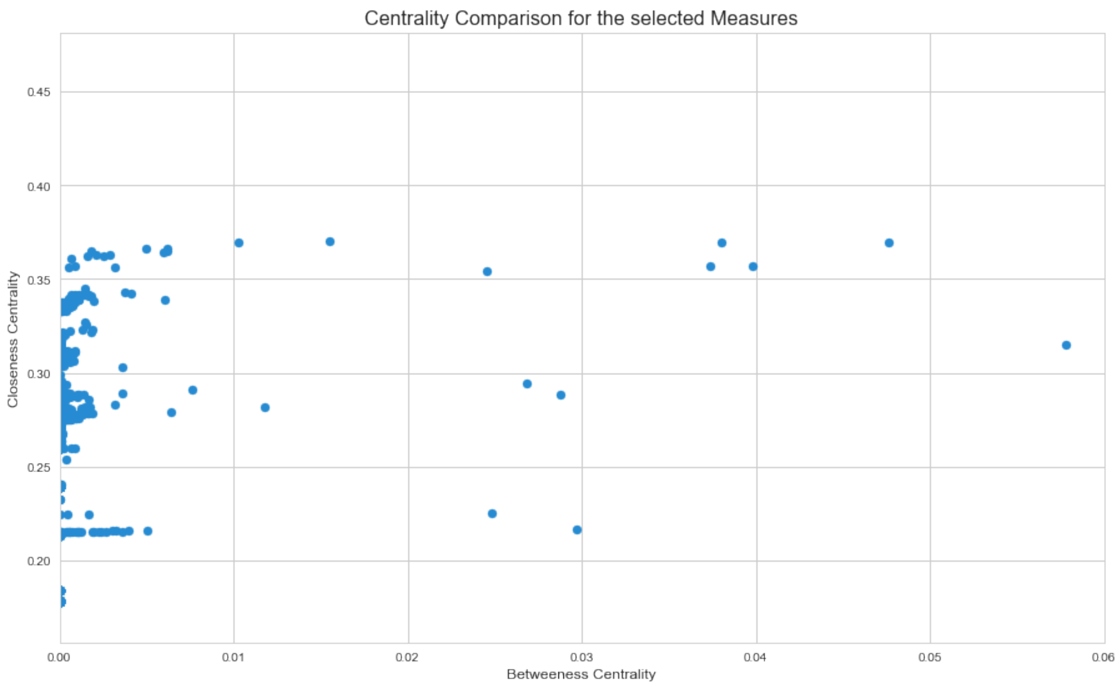


As shown above, except for katz, measures agree with each other in a propotional manner.

```
In [25]: print("Closeness Centrality:")
         plot_centrality(dc_G,cc_G,"Degree Centrality","Closeness Centrality",0,0,0.5,0.02)
         plot_centrality(bc_G,cc_G,"Betweeness Centrality","Closeness Centrality",0,0,0.06,0.02
```

```
Closeness Centrality:
```





In the plots above, relating both degree and betweenness centrality to closeness centrality we get a more skewed linear relationship, with closeness attributing value to some nodes where betweenness and degree give very little. One could intepret this as, in the overall sense, there being some relationship between the number of shortest paths going through a node being proportional to the inverse of the distance of those shortest paths, in other words, how effective a player is in getting from source to sink is ultimately dependent on how much a given node acts as a hub (how many links to other pages a given article has, and how common of a topic is that article about).

Print top 10 entries according to measures

In [26]:
```
print("Degree Centrality")
print_top_10(dc_G,G_GCC)
S_dc=store_top_200(dc_G,G_GCC)
print(S_dc)

print("Closeness Centrality")
print_top_10(cc_G,G_GCC)
S_cc=store_top_200(cc_G,G_GCC)
print(S_cc)

print("Betweenness Centrality")
print_top_10(bc_G,G_GCC)
S_bc=store_top_200(bc_G,G_GCC)
print(S_bc)

print("Page Rank")
print_top_10(pr_G,G_GCC)
S_pr=store_top_200(pr_G,G_GCC)
print(S_pr)

print("Eigenvector Centrality")
print_top_10(ev_G,G_GCC)
S_ev=store_top_200(ev_G,G_GCC)
print(S_ev)

print("Katz Centrality")
print_top_10(katz_G,G_GCC)
S_katz=store_top_200(katz_G,G_GCC)
print(S_katz)

#S_xx stores the possible targets
```

```
Degree Centrality


  Node ID  Centrality Score
```

0	107	0.258791
1	1684	0.196137
2	1912	0.186974
3	3437	0.135463
4	0	0.085934
5	2543	0.072808
6	2347	0.072065
7	1888	0.062902
8	1800	0.060674
9	1663	0.058197

	Node ID	Centrality Score
0	107	0.258791
1	1684	0.196137
2	1912	0.186974
3	3437	0.135463
4	0	0.085934
5	2543	0.072808
6	2347	0.072065
7	1888	0.062902
8	1800	0.060674
9	1663	0.058197
10	2266	0.057949
11	1352	0.057949
12	483	0.057207
13	348	0.056711
14	1730	0.055968
15	1985	0.055473
16	1941	0.055225
17	2233	0.054978
18	2142	0.054730
19	1431	0.054482
20	1199	0.053739
21	1584	0.052254
22	2206	0.052006
23	1768	0.051758
24	2410	0.051263
25	2229	0.051263
26	2611	0.051263
27	1589	0.050768
28	2047	0.050768
29	2218	0.050768
..	...	...
170	2283	0.039871
171	2446	0.039871
172	2154	0.039624
173	2112	0.039624

```
174    2333          0.039624
175    1554          0.039624
176    2289          0.039624
177    2482          0.039624
178    2133          0.039376
179     414          0.039376
180    1516          0.039376
181    2121          0.039128
182    2370          0.039128
183    2020          0.039128
184    1184          0.039128
185    2551          0.039128
186    2253          0.038881
187    1522          0.038633
188    1736          0.038633
189    2087          0.038633
190    1185          0.038633
191    1813          0.038633
192    2308          0.038633
193    1718          0.038385
194    2299          0.038385
195    2005          0.038385
196    2257          0.038385
197    2561          0.038385
198    2095          0.038138
199     993          0.038138

[200 rows x 2 columns]
Closeness Centrality


  Node ID  Centrality Score
0     107          0.459699
1      58          0.397402
2     428          0.394837
3     563          0.393913
4    1684          0.393606
5     171          0.370493
6     348          0.369916
7     483          0.369848
8     414          0.369543
9     376          0.366558


    Node ID  Centrality Score
0       107          0.459699
1        58          0.397402
2       428          0.394837
```

3	563	0.393913
4	1684	0.393606
5	171	0.370493
6	348	0.369916
7	483	0.369848
8	414	0.369543
9	376	0.366558
10	475	0.366192
11	566	0.364967
12	1666	0.364704
13	1534	0.364605
14	484	0.363162
15	353	0.363097
16	1171	0.362445
17	651	0.362282
18	420	0.361019
19	1085	0.357852
20	1687	0.357250
21	1577	0.357187
22	1718	0.356651
23	1165	0.356493
24	1136	0.356305
25	1465	0.354615
26	0	0.353343
27	1912	0.350947
28	580	0.345040
29	1505	0.342755
..	...	...
170	1431	0.320425
171	1199	0.320349
172	629	0.320324
173	649	0.320324
174	917	0.320298
175	1377	0.320197
176	1612	0.320146
177	1589	0.320044
178	1078	0.320019
179	1746	0.319968
180	1827	0.319943
181	896	0.319943
182	1813	0.319918
183	1277	0.319892
184	1487	0.319867
185	1238	0.319816
186	1622	0.319816
187	1804	0.319791
188	1390	0.319740
189	1645	0.319715

190	1604	0.319715
191	1616	0.319690
192	1833	0.319690
193	1620	0.319664
194	1783	0.319664
195	1844	0.319639
196	1367	0.319588
197	1714	0.319588
198	1559	0.319563
199	1707	0.319538

[200 rows x 2 columns]
Betweenness Centrality

	Node ID	Centrality Score
0	107	0.480518
1	1684	0.337797
2	3437	0.236115
3	1912	0.229295
4	1085	0.149015
5	0	0.146306
6	698	0.115330
7	567	0.096310
8	58	0.084360
9	428	0.064309

	Node ID	Centrality Score
0	107	0.480518
1	1684	0.337797
2	3437	0.236115
3	1912	0.229295
4	1085	0.149015
5	0	0.146306
6	698	0.115330
7	567	0.096310
8	58	0.084360
9	428	0.064309
10	563	0.062780
11	860	0.057826
12	414	0.047633
13	1577	0.039785
14	348	0.037998
15	1718	0.037343
16	686	0.029722
17	594	0.028803
18	136	0.026870

```
19     3980          0.024820
20     1465          0.024572
21      171          0.015492
22      862          0.011748
23      483          0.010308
24     2543          0.007605
25     3830          0.006437
26      376          0.006196
27     1666          0.006175
28     1420          0.006057
29     1534          0.005948
..      ...               ...
170     772          0.000472
171     213          0.000468
172    3750          0.000464
173    1537          0.000458
174    2081          0.000455
175    2384          0.000449
176    1617          0.000448
177    3721          0.000444
178    2512          0.000441
179    3550          0.000439
180    1338          0.000436
181    3872          0.000434
182    3651          0.000434
183     601          0.000427
184    1540          0.000422
185    3886          0.000420
186    1702          0.000416
187     391          0.000412
188    2111          0.000412
189     500          0.000410
190     169          0.000402
191     422          0.000386
192     908          0.000385
193     525          0.000385
194     921          0.000382
195    3556          0.000370
196     857          0.000369
197    2054          0.000362
198     804          0.000362
199    2176          0.000358

[200 rows x 2 columns]
Page Rank


  Node ID  Centrality Score
```

0	3437	0.007615
1	107	0.006936
2	1684	0.006367
3	0	0.006290
4	1912	0.003877
5	348	0.002348
6	686	0.002219
7	3980	0.002170
8	414	0.001800
9	698	0.001317

	Node ID	Centrality Score
0	3437	0.007615
1	107	0.006936
2	1684	0.006367
3	0	0.006290
4	1912	0.003877
5	348	0.002348
6	686	0.002219
7	3980	0.002170
8	414	0.001800
9	698	0.001317
10	483	0.001297
11	3830	0.001184
12	376	0.000901
13	2047	0.000841
14	56	0.000804
15	25	0.000800
16	828	0.000789
17	322	0.000787
18	475	0.000785
19	428	0.000780
20	67	0.000772
21	3596	0.000766
22	2313	0.000754
23	713	0.000749
24	271	0.000746
25	563	0.000740
26	917	0.000733
27	119	0.000732
28	3545	0.000727
29	3938	0.000727
..	...	...
170	170	0.000507
171	1078	0.000507
172	3611	0.000506
173	465	0.000505

174	3677	0.000505
175	3256	0.000504
176	1399	0.000503
177	1622	0.000503
178	272	0.000503
179	1471	0.000502
180	3150	0.000502
181	1104	0.000502
182	1610	0.000500
183	98	0.000500
184	3019	0.000500
185	199	0.000497
186	3877	0.000496
187	40	0.000496
188	1235	0.000496
189	3758	0.000495
190	1391	0.000495
191	3680	0.000495
192	1204	0.000493
193	2007	0.000493
194	3201	0.000493
195	2054	0.000492
196	3076	0.000492
197	2598	0.000491
198	3002	0.000490
199	1211	0.000489

[200 rows x 2 columns]
Eigenvector Centrality

	Node ID	Centrality Score
0	1912	0.095406
1	2266	0.086983
2	2206	0.086053
3	2233	0.085173
4	2464	0.084279
5	2142	0.084193
6	2218	0.084156
7	2078	0.084136
8	2123	0.083672
9	1993	0.083533

	Node ID	Centrality Score
0	1912	0.095406
1	2266	0.086983
2	2206	0.086053

3	2233	0.085173
4	2464	0.084279
5	2142	0.084193
6	2218	0.084156
7	2078	0.084136
8	2123	0.083672
9	1993	0.083533
10	2410	0.083518
11	2244	0.083342
12	2507	0.083273
13	2240	0.083057
14	2340	0.083053
15	2229	0.083008
16	1985	0.082738
17	2088	0.082470
18	2073	0.082256
19	2220	0.082156
20	2131	0.082133
21	2604	0.082071
22	2059	0.082065
23	2309	0.082022
24	2590	0.082002
25	2369	0.081757
26	2611	0.081547
27	2602	0.081494
28	2607	0.081245
29	2090	0.080998
..	...	...
170	2237	0.054839
171	2329	0.054647
172	1925	0.054589
173	2506	0.053337
174	2579	0.052451
175	2418	0.052315
176	2098	0.051790
177	1941	0.051681
178	2306	0.050950
179	2631	0.050503
180	2477	0.050104
181	2552	0.050094
182	2563	0.048744
183	2532	0.048299
184	2489	0.047606
185	2300	0.047274
186	2213	0.045565
187	2574	0.044670
188	2060	0.044455
189	2056	0.044077

190	2591	0.043035
191	2210	0.042375
192	2554	0.042372
193	2179	0.041943
194	2392	0.040720
195	2462	0.040067
196	2407	0.039973
197	2164	0.039766
198	1989	0.038698
199	1963	0.038467

[200 rows x 2 columns]
Katz Centrality

	Node ID	Centrality Score
0	2696	0.106398
1	2921	0.099801
2	2934	0.094739
3	3275	0.085035
4	2870	0.083750
5	2951	0.081817
6	1718	0.080273
7	3274	0.080043
8	3246	0.078651
9	2697	0.077689

	Node ID	Centrality Score
0	2696	0.106398
1	2921	0.099801
2	2934	0.094739
3	3275	0.085035
4	2870	0.083750
5	2951	0.081817
6	1718	0.080273
7	3274	0.080043
8	3246	0.078651
9	2697	0.077689
10	2820	0.077537
11	3299	0.071543
12	3385	0.067596
13	3244	0.067080
14	366	0.066822
15	3017	0.064491
16	434	0.063463
17	3204	0.063426
18	2751	0.060786

19	637	0.057535
20	3224	0.057394
21	1570	0.057091
22	3182	0.056790
23	3001	0.056197
24	1672	0.055629
25	925	0.055428
26	3360	0.054966
27	348	0.054768
28	3342	0.053044
29	3221	0.052917
..	...	...
170	1170	0.029274
171	3662	0.029247
172	372	0.029208
173	968	0.029003
174	528	0.029001
175	3242	0.028975
176	1663	0.028756
177	2996	0.028719
178	3549	0.028638
179	3546	0.028595
180	3265	0.028532
181	2841	0.028365
182	3236	0.028215
183	2556	0.028202
184	2539	0.028024
185	1337	0.028004
186	1549	0.027728
187	479	0.027690
188	2246	0.027650
189	3624	0.027637
190	924	0.027598
191	2724	0.027492
192	428	0.027369
193	1735	0.027175
194	641	0.027164
195	2616	0.027057
196	2208	0.027004
197	1280	0.026987
198	1453	0.026975
199	3020	0.026908

[200 rows x 2 columns]

As can be seen in these tables, except for katz centrality all measures agree on what one would intuitively attribute as important nodes: the nodes with the highest degree correspond to the ones with the highest centrality scores which correspond to the nodes most commonly known to the majority of the player base.

Degree correlation over Focus Categories

```
In [27]: print("\nOverall Correlation Coefficient:",np.round(nx.degree_pearson_correlation_coe
```

```
Overall Correlation Coefficient: 0.064
```

3 5. Community Detection

In this section the Louvain algorthim is used in order to detect communities. The random seed parameter had to be fixed as the algorithm's stochastic component would sometimes generate and extra (or a couple extra) communities with a very small number of nodes, which is nonsensical. The resolution parameter was also adjusted as a result of this section's analysis, to ensure that communities were balanced in relative terms. This is not due to wanting to induce some prior belief on the number of communities, but rather that when the algorthm's output resulted in a pair communities with identical composition half the size of all the others there was no need to consider them as individual communities.

As anticipated, this section is primarily intended to compare and contrast with the hard-generated grouping based on wikipedia categories analyzed in section 4.

```
In [28]: # NOT IMPLEMENTED - TOO SLOW
         #list_clique_communities = list(k_clique_communities(G2_GCC, k=15))
         #list_clique_communities
```

```
In [29]: np.random.seed(5000)
```

```
In [30]: #nx.greedy_modularity_communities(G_GCC, weight=None)
```

```
In [31]: partition = community.best_partition(G_GCC, resolution=1.2,  random_state=5000)
         labels = set(partition.values())
         community_counts = {i: list(partition.values()).count(i) for i in labels}
         community_counts
```

```
Out[31]: {0: 350,
          1: 465,
          2: 439,
          3: 554,
          4: 442,
          5: 206,
          6: 237,
          7: 226,
          8: 25,
          9: 61,
          10: 315,
```

```
11: 548,
12: 53,
13: 73,
14: 45}
```

```
In [32]: #print(partition)
```

For the given parameterization, the algorithm finds six communities of approximately equal size. The contents of these shall be examined below.

```
In [33]: T = []
         for i in range(len(labels)):
             c = [nodes for nodes in partition.keys() if partition[nodes] == i]
             T.append(G_GCC.subgraph(c))
```

```
In [34]: corr_list2 =[nx.degree_pearson_correlation_coefficient(T[l]) for l in range(len(T))]
         a = [print('Community:',l,'has correlation coefficient:',corr_list2[l]) for l in range
```

```
Community: 0 has correlation coefficient: -0.14047538438748622
Community: 1 has correlation coefficient: -0.045146108781109556
Community: 2 has correlation coefficient: 0.005612735431508522
Community: 3 has correlation coefficient: -0.07357425101046838
Community: 4 has correlation coefficient: 0.014110833129858153
Community: 5 has correlation coefficient: -0.11409190052678135
Community: 6 has correlation coefficient: 0.0013672180351790464
Community: 7 has correlation coefficient: 0.1365525266388471
Community: 8 has correlation coefficient: -0.12420387339962306
Community: 9 has correlation coefficient: -0.2840661133551881
Community: 10 has correlation coefficient: 0.1290968873760748
Community: 11 has correlation coefficient: -0.09288000605967495
Community: 12 has correlation coefficient: 0.4558897443512828
Community: 13 has correlation coefficient: -0.07462914279980881
Community: 14 has correlation coefficient: -0.06957281819667319
```

Community Layout Plot and Induced Community Graph NOTE: Takes a relatively long time to plot (2-5 min)

```
In [35]: # Some fancy code for inferring node position based on communities
         # https://stackoverflow.com/questions/43541376/how-to-draw-communities-with-networkx/
         def community_layout(g, partition):
             """
             Compute the layout for a modular graph.


             Arguments:
             ----------
             g -- networkx.Graph or networkx.DiGraph instance
                 graph to plot
```

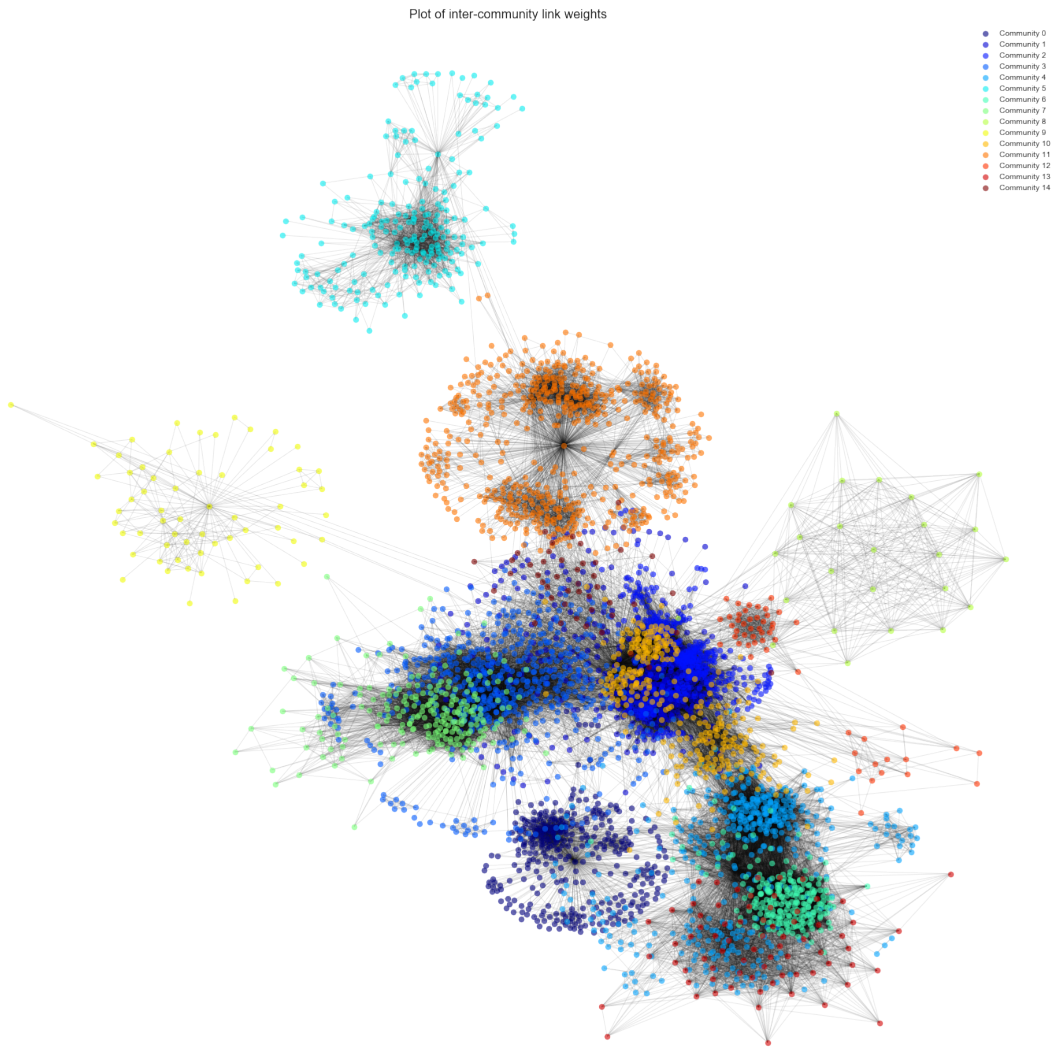
```
    partition -- dict mapping int node -> int community
        graph partitions


    Returns:
    --------
    pos -- dict mapping int node -> (float x, float y)
        node positions

    """

    pos_communities = _position_communities(g, partition, scale=3.)

    pos_nodes = _position_nodes(g, partition, scale=1.)

    # combine positions
    pos = dict()
    for node in g.nodes():
        pos[node] = pos_communities[node] + pos_nodes[node]

    return pos

def _position_communities(g, partition, **kwargs):

    # create a weighted graph, in which each node corresponds to a community,
    # and each edge weight to the number of edges between communities
    between_community_edges = _find_between_community_edges(g, partition)

    communities = set(partition.values())
    hypergraph = nx.DiGraph()
    hypergraph.add_nodes_from(communities)
    for (ci, cj), edges in between_community_edges.items():
        hypergraph.add_edge(ci, cj, weight=len(edges))

    # find layout for communities
    pos_communities = nx.spring_layout(hypergraph, **kwargs)

    # set node positions to position of community
    pos = dict()
    for node, community in partition.items():
        pos[node] = pos_communities[community]

    return pos

def _find_between_community_edges(g, partition):

    edges = dict()
```

```
    for (ni, nj) in g.edges():
        ci = partition[ni]
        cj = partition[nj]

        if ci != cj:
            try:
                edges[(ci, cj)] += [(ni, nj)]
            except KeyError:
                edges[(ci, cj)] = [(ni, nj)]

    return edges

def _position_nodes(g, partition, **kwargs):
    """
    Positions nodes within communities.
    """

    communities = dict()
    for node, community in partition.items():
        try:
            communities[community] += [node]
        except KeyError:
            communities[community] = [node]

    pos = dict()
    for ci, nodes in communities.items():
        subgraph = g.subgraph(nodes)
        pos_subgraph = nx.spring_layout(subgraph, **kwargs)
        pos.update(pos_subgraph)

    return pos
```

```
In [36]: def plot_communities(G):
    #colormap2 = np.array(["magenta", "yellow", "red", "white", "black", "cyan"])#, "
    p = cm.get_cmap('jet', len(set(partition.values())))
    colormap = np.array(p(np.arange(len(set(partition.values())))))[:len(set(partitio
    colormap2 = [rgb2hex(colormap[i,:]) for i in range(colormap.shape[0])]
    plt.figure(figsize=(25,25))
    pos = community_layout(G, partition)

    count = 0
    for com in set(partition.values()) :
        list_nodes = [nodes for nodes in partition.keys() if partition[nodes] == com]
        nx.draw_networkx_nodes(G, pos, list_nodes, node_size = 50, alpha=0.6, node_co
        count = count + 1

    nx.draw_networkx_edges(G, pos, alpha=0.10,  edge_color='k')
```

```
        plt.legend()
        plt.axis('off')
        plt.title('Plot of inter-community link weights')
        plt.show()
```

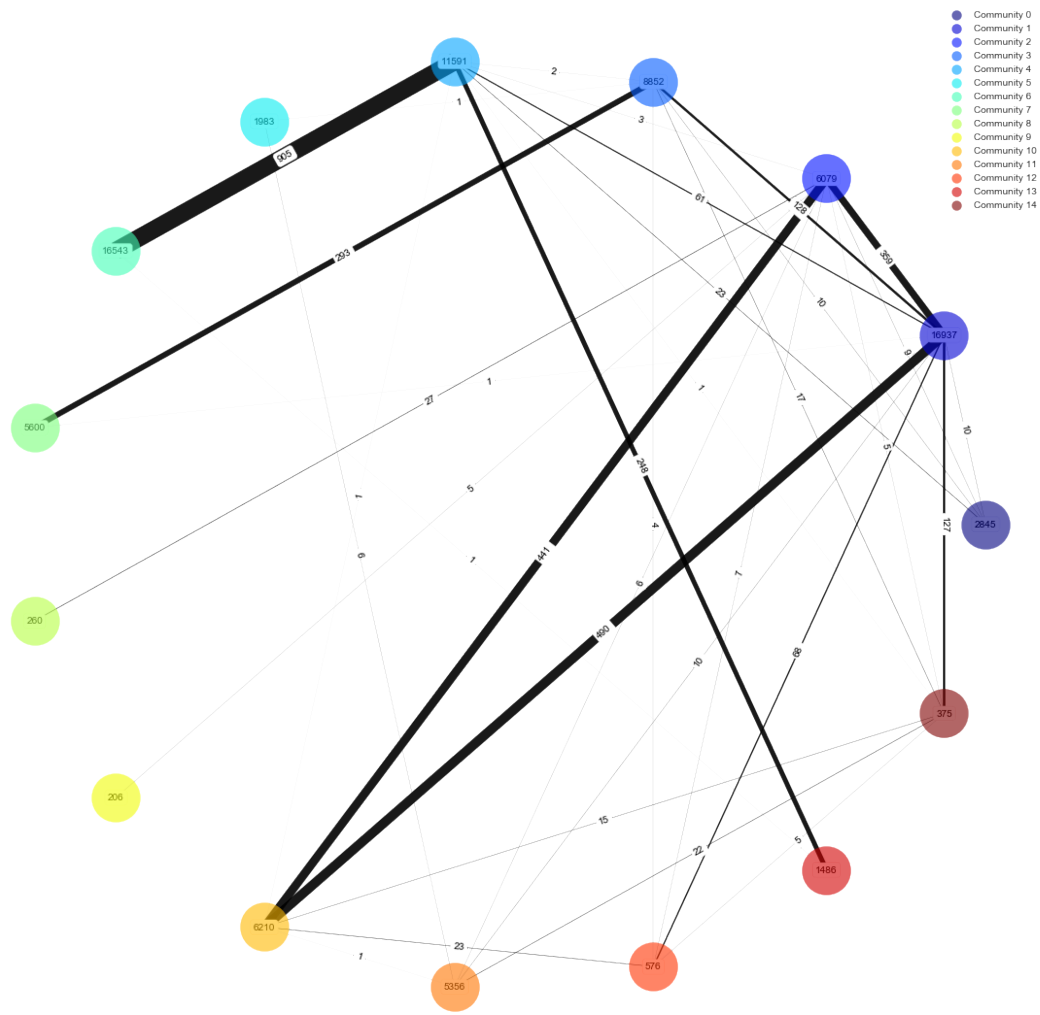
```
In [37]: plot_communities(G_GCC)
```



The (arguably pictoral) plot above gives us a good first representation of how inter-community links are occurring. Whereas the induced graph representation below will give insight into the interlink structure numerically, this one is arguably more useful in terms of seeing between each pair of communities the relative proportion of high weight links to low weight links. The weighted network was used in this case. Again, given the ad-hoc construction of weights this may not be entirely representative of the true weight structure, however using the unweighted network results in uniformly colored links, which renders this plot uninformative.

In the plot, darker links correspond to links with low weight, the lighter they get, the higher the weight. From this, we see that a large portion of links between each community has a low weight, this implies that when going from source article to sink article community traversal a large number of articles are sparsely visited, whereas a low number of articles (the occasional lighter edges) are visited very often. The lighter connections between each community are therefore representative of those two articles in disparate areas which act as common ground between communities (ie. two articles about bioengineering to relate the biology and tech community).

In [38]:
```
T_com = community.induced_graph(partition, G_GCC, weight='weight')
def plot_graph(G):
    plt.figure(figsize=(20,20))
    pos = nx.circular_layout(G)

    p = cm.get_cmap('jet', len(set(partition.values())))
    colormap = np.array(p(np.arange(len(set(partition.values())))))[:len(set(partitio
    colormap2 = [rgb2hex(colormap[i,:]) for i in range(colormap.shape[0])]
    #colormap2 = np.array(["magenta", "yellow", "red", "white", "black", "cyan"])#, "

    # plot nodes
    #nx.draw_networkx_nodes(G, pos, node_color='orange',node_size=2000, alpha=0.5)
    count = 0
    for com in set(partition.values()) :
        #list_nodes = [nodes for nodes in partition.keys() if partition[nodes] == com
        nx.draw_networkx_nodes(G_GCC, pos, nodelist=[count], node_size = 2500, alpha=
        count = count + 1

    # plot edges with widths depending on weights
    nx.draw_networkx_edges(G, pos, edge_color='k', alpha=0.9, width = [d['weight']/50

    # add edges' labels with weights
    edge_labels=dict([((u,v), d['weight']) for u,v,d in G.edges(data=True)])
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)
    plt.axis('off')
    plt.legend(markerscale=0.2)
    plt.show()

plot_graph(T_com)
```


Community 0
Community 1
Community 2
Community 3
Community 4
Community 5
Community 6
Community 7
Community 8
Community 9
Community 10
Community 11
Community 12
Community 13
Community 14
11591
8852
1983
6079
16543
16937
5600
2845
260
375
206
1486
6210
576
5356