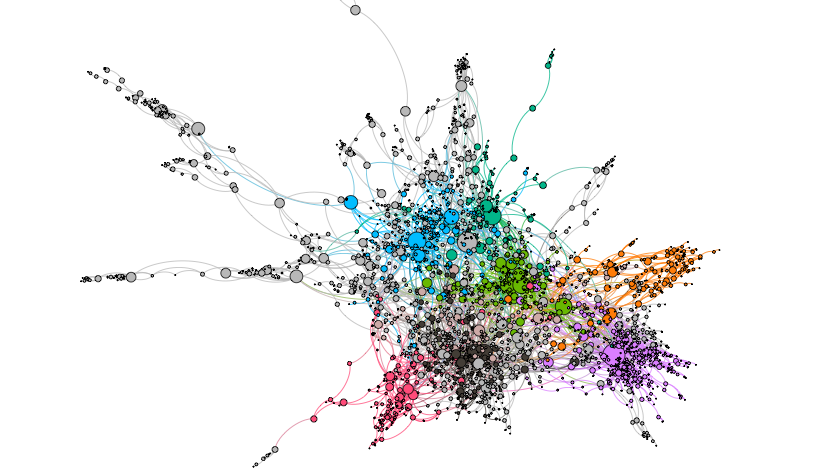


A Report on

**NETWORK GRAPH ANALYSIS**



Under esteemed guidance of

Ms. Ashwairya Saxena

(Machine Learning Project)

By:

Manoj Konda,

NIT Warangal

**INDEX**

|  |  |
| --- | --- |
| **Sl.No.** | **Topic** |
| 1 | INTRODUCTION |
| 2 | OBJECTIVE |
| 3 | BACKGROUND |
| 4 | HARDWARE & SOFTWARE REQUIREMENTS |
| 5 | CODING |
| 6 | OUTPUT SCREENSHOTS |
| 7 | FUTURE SCOPE |
| 8 | CONCLUSION |
| 9 | BIBLIOGRAPHY AND REFERENCES |

**INTRODUCTION**

Network Graph Analysis has real broad applications in the field of networking. Two main areas are involved in the analysis of the application of network graphs, which are graph-based representation and network theory. Complex systems, such as a power grid, the World Wide Web, activity in different regions of the brain, or people within a community, can be understood, studied and visualized based on their connections in a network.

Networks are everywhere.

Networks or Graphs are a set of objects (called nodes) having some relationship with each other (called edges).

We live in a connected world and generate a vast amount of connected data. Social networks, financial transaction systems, biological networks, transportation systems, and a telecommunication nexus are all examples.

**OBJECTIVE**

To Explore and Analyse Network Graph data ,Node Classification, Link prediction(predicting missing links ) and Community detection.

**BACKGROUND**

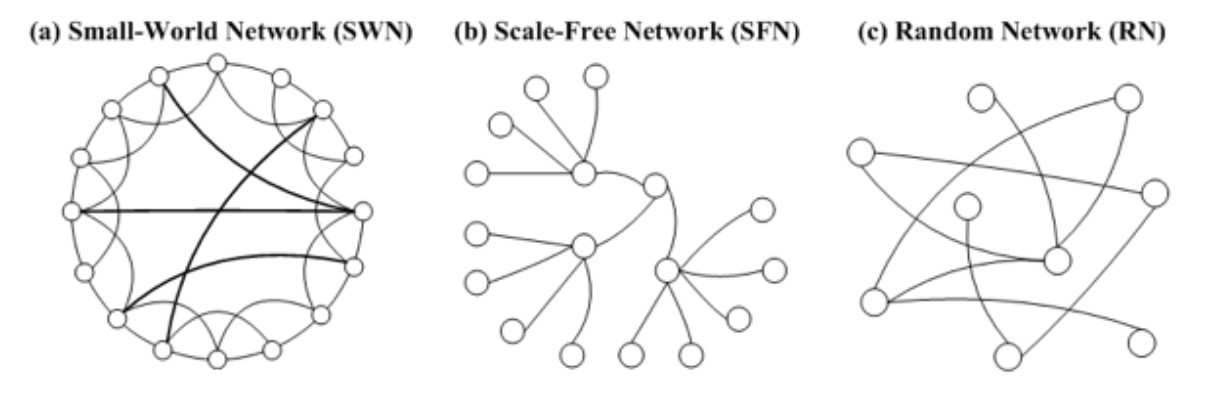
A graph has two components which are nodes and edges. In a graphical representation, these components have natural correspondences with the elements of the problem. In general, nodes in a graph represent features and edges represent interactions between features.

Nodes (A,B,C,D,E in the example) are usually representing entities in the network, and can hold self-properties (such as weight, size, position and any other attribute) and network-based properties (such as Degree- number of neighbors or Cluster- a connected component the node belongs to etc.).

Edges represent the connections between the nodes, and might hold properties as well (such as weight representing the strength of the connection, direction in case of asymmetric relation or time if applicable).

These two basic elements can describe multiple phenomena, such as social connections, virtual routing network, physical electricity networks, roads network, biology relations network and many other relationships.

Real-world networks and in particular social networks have a unique structure which often differs them from random mathematical networks:

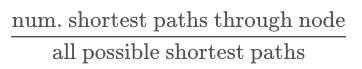


* The Small-World Network phenomenon claims that real networks often have very short paths (in terms of number of hops) between any connected network members. This applies for real and virtual social networks (the six handshakes theory) and for physical networks such as airports or electricity or web-traffic routings.
* Scale Free networks with power-law degree distribution have a skewed population with a few highly-connected nodes (such as social-influences) and a lot of loosely-connected nodes.
* Homophily is the tendency of individuals to associate and bond with similar others, which results in similar properties among neighbors.

**Centrality**:

In network analysis, measures of the importance of nodes are referred to as centrality measures. Highly central nodes play a key role of a network, serving as hubs for different network dynamics. However the definition and importance of centrality might differ from case to case, and may refer to different centrality measures:

* Degree : the amount of neighbors of the node
* EigenVector / PageRank : iterative circles of neighbors
* Closeness : the level of closeness to all of the nodes
* Betweenness: the amount of short path going through the node

**Degree** is the simplest and the most common way of finding important nodes. A node’s degree is the sum of its edges. If a node has three lines extending from it to other nodes, its degree is three. Five edges, its degree is five. It’s really that simple. Since each of those edges will always have a node on the other end, you might think of degree as the number of people to which a given person is directly connected. The nodes with the highest degree in a social network are the people who know the most people. These nodes are often referred to as hubs, and calculating degree is the quickest way of identifying hubs

Degree centrality =

**Eigenvector** centrality is a kind of extension of degree—it looks at a combination of a node’s edges and the edges of that node’s neighbors. Eigenvector centrality cares if you are a hub, but it also cares how many hubs you are connected to. It’s calculated as a value from 0 to 1: the closer to one, the greater the centrality. Eigenvector centrality is useful for understanding which nodes can get information to many other nodes quickly.

Eigenvector centrality computes the centrality for a node based on the centrality of its neighbors. The eigenvector centrality for node i is the i-th element of the vector x defined by the equation

Ax=λx

where A is the adjacency matrix of the graph G with eigenvalue λ. By virtue of the Perron–Frobenius theorem, there is a unique solution x, all of whose entries are positive, if λ is the largest eigenvalue of the adjacency matrix A^2

.

**Betweenness centrality** is a bit different from the other two measures in that it doesn’t care about the number of edges any one node or set of nodes has. Betweenness centrality looks at all the shortest paths that pass through a particular node. To do this, it must first calculate every possible shortest path in your network, so keep in mind that betweenness centrality will take longer to calculate than other centrality measures (but it won’t be an issue in a dataset of this size). Betweenness centrality, which is also expressed on a scale of 0 to 1, is fairly good at finding nodes that connect two otherwise disparate parts of a network. If you’re the only thing connecting two clusters, every communication between those clusters has to pass through you. In contrast to a hub, this sort of node is often referred to as a broker. Betweenness centrality is not the only way of finding brokerage (and other methods are more systematic), but it’s a quick way of giving you a sense of which nodes are important not because they have lots of connections themselves but because they stand between groups, giving the network connectivity and cohesion.

Betweenness centrality of a node v is the sum of the fraction of all-pairs shortest paths that pass through v



Where  is the total number of shortest paths from node  to node  and  is the number of those paths that pass through .

**Libraries Used**:

* numpy
* matplotlib
* networkx
* warnings
* pandas
* Subprocess - check\_output
* Networkx.algorithms - community
* Networkx.algorithms - node\_classification
* **Numpy**: NumPy, which stands for Numerical Python, is a library consisting of multidimensional array objects and a collection of routines for processing those arrays. Using NumPy, mathematical and logical operations on arrays can be performed
* **Matplotlib:** Matplotlib is an amazing visualization library in Python for 2D plots of arrays. Matplotlib is a multi-platform data visualization library built on NumPy arrays and designed to work with the broader SciPy stack
* **Pandas**: Pandas is an open-source, BSD-licensed Python library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language. Python with Pandas is used in a wide range of fields including academic and commercial domains including finance, economics, Statistics, analytics, etc.
* **Networkx**: NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.

**NetworkX provides**:

* tools for the study of the structure and dynamics of social, biological, and infrastructure networks;
* a standard programming interface and graph implementation that is suitable for many applications;
* a rapid development environment for collaborative, multidisciplinary projects;
* an interface to existing numerical algorithms and code written in C, C++, and FORTRAN; and
* the ability to painlessly work with large nonstandard data sets.

With NetworkX you can load and store networks in standard and nonstandard data formats, generate many types of random and classic networks, analyze network structure, build network models, design new network algorithms, draw networks, and much more.

Applications: It helps to visualize Network data i.e., graph data , Node classification, Link Prediction(predicting relation between nodes), Community detection of a node etc.,

**Algorithms Used**:

* **Node Classification** - there is an inbuilt module in networkx i.e., node\_classification.

This module provides the functions for node classification problems.

The functions in this module are not imported into the top level `networkx` namespace.

You can access these functions by importing the `networkx.algorithms.node\_classification` modules, then accessing the functions as attributes of `node\_classification`

We use harmonic function algorithm in Node classification.

* **Link prediction** - There are various algorithms for Link prediction

1. Triadic closure
2. Jaccard Coefficient
3. Resource Allocation Index
4. Adamic Adar Index
5. Preferential Attachment
6. Community Common Neighbor
7. Community Resource Allocation

In these we use Jaccard Coefficient, Resource Allocation Index, Adamic Adar Index, Preferential Attachment,

**Jaccard Coefficient** :

It is calculated by the number of common neighbors normalized by the total number of neighbors. It is used to measure the similarity between two finite sample sets, and is defined as the size of the intersection divided by the size of the union of the sample sets.

Jaccard Coefficient(X, Y) = 

**Resource Allocation Index** :

In pair of nodes u, v that have no direct link, node u can allocate some resources to the node v through their common neighbor. Their common neighbors assume the role of passers. In the simplest case, we assume that each passer has a unit of resources; it assigns these resources to its neighbors evenly

Among a number of similarity-based methods to predict missing links in a complex network, Research Allocation Index performs well with lower time complexity. It is defined as a fraction of a resource that a node can send to another through their common neighbors.

Research Allocation Index(X, Y) = 

**Adamic Adar Index** :

This similarity index assigns a higher similarity function value to a small degree node. Adamic-Adar algorithm believes that an affair owned by less objects, compared to owned by more objects, has greater effect on link prediction.

This measure was introduced in 2003 to predict missing links in a Network, according to the amount of shared links between two nodes. It is calculated as follows:

Adamic Adar Index(X, Y) = 

**Preferential Attachment** :

Preferential attachment means that the more connected a node is, the more likely it is to receive new links. Nodes with higher degree get more neighbors.

Preferential attachment mechanism can be used to generate scale-free network evolution models. The probability of generating a new link of node u is directly proportional to the degree of the node

Preferential Attachment(X, Y) =|N(X)|.|N(Y)|

* **Community detection:**

1. Kernighan–Lin bipartition algorithm- Partition a graph into two blocks using the Kernighan–Lin algorithm.
2. K-Clique- Find k-clique communities in a graph using the percolation method.
3. Modularity-based communities-
4. Find communities in a graph using the greedy modularity maximization.
5. Find communities in graphs using Clauset-Newman-Moore greedy modularity maximization.
6. Lukes Algorithm for exact optimal weighted tree partitioning.
7. Label propagation community detection algorithms. etc.,

I used greedy\_modularity\_communities(G) and label\_propagation\_communities(G)

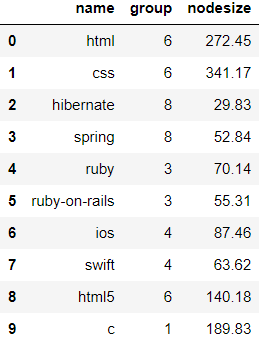
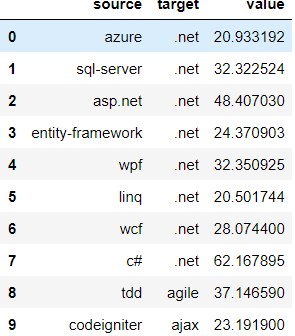
Common thing to ask about a network dataset is what the subgroups or communities are within the larger social structure. Is your network one big, happy family where everyone knows everyone else? Or is it a collection of smaller subgroups that are only connected by one or two intermediaries? The field of community detection in networks is designed to answer these questions. There are many ways of calculating communities, cliques, and clusters in your network, but the most popular method currently is modularity. Modularity is a measure of relative density in your network: a community (called a module or modularity class) has high density relative to other nodes within its module but low density with those outside. Modularity gives you an overall score of how fractious your network is, and that score can be used to partition the network and return the individual communities.13

Very dense networks are often more difficult to split into sensible partitions. Luckily, as you discovered earlier, this network is not all that dense. There aren’t nearly as many actual connections as possible connections, and there are several altogether disconnected components. It's worthwhile partitioning this sparse network with modularity and seeing if the result makes historical and analytical sense.

Community detection and partitioning in NetworkX requires a little more setup than some of the other metrics.

**Dataset**: My datasets “stack\_network\_links.csv” and stack\_network\_nodes.csv” are the info regarding different technologies(like html, css, c#, react ect.,) are considered as nodes whereas links are source and target

Converted datasets as dataframes below:

.

**CODING**

**import networkx as nx**

**import matplotlib.pyplot as plt**

**import pandas as pd**

**from operator import itemgetter**

**import seaborn as sns**

**import numpy as np**

**from networkx.algorithms import node\_classification**

**from networkx.algorithms import community**

**import warning**s

warnings.filterwarnings('ignore')

# Reading datasets and importing data

G = nx.Graph()

df\_nodes = pd.read\_csv("dataset/stack\_network\_nodes.csv")

df\_edges = pd.read\_csv("dataset/stack\_network\_links.csv")

for index, row in df\_nodes.iterrows():

G.add\_node(row['name'], group=row['group'], nodesize=row['nodesize'])

for index, row in df\_edges.iterrows():

G.add\_weighted\_edges\_from([(row['source'], row['target'], row['value'])])

#Nodes dataframe

df\_nodes.head(10)

# edges dataframe

df\_edges.head(10)

#general info of network

print(nx.info(G))

print("\nList of all {} nodes present\n{}".format(len(G.nodes()), G.nodes()))

# check whether our network is connected or not connected

if nx.is\_connected(G):

print('Connected Graph')

else:

print("Not connected")

# Network Density

print("\nNetwork density:", nx.density(G))

# network density of 0.037376 says that it is not completely connected.

# May not all the nodes are connected

# diameter of largest component

components = nx.connected\_components(G)

largest\_component = max(components, key=len)

print(largest\_component)

subgraph = G.subgraph(largest\_component)

diameter = nx.diameter(subgraph)

print("\nNetwork diameter of largest component:", diameter)

# Triadic closure(Local Clustering coefficient). This is one of the link prediction methods

triadic\_closure = nx.transitivity(G)

print("\nTriadic closure:", triadic\_closure)

# Centrality

# Degree Centrality

# Top three nodes having the highest Degree Centrality

def find\_nodes\_with\_highest\_deg\_cent(G):

# Compute the degree centrality of G: deg\_cent

deg\_cent = nx.degree\_centrality(G)

# Compute the maximum degree centrality: max\_dc

max\_1\_dc = max(list(deg\_cent.values()))

max\_2\_dc = list(sorted(deg\_cent.values()))[-2]

max\_3\_dc = list(sorted(deg\_cent.values()))[-3]

maxnode1 = set()

maxnode2 = set()

maxnode3 = set()

# Iterate over the degree centrality dictionary

for k, v in deg\_cent.items():

# Check if the current value has the maximum degree centrality

if v == max\_1\_dc:

# Add the current node to the set of nodes

maxnode1.add(k)

if v == max\_2\_dc:

# Add the current node to the set of nodes

maxnode2.add(k)

if v == max\_3\_dc:

# Add the current node to the set of nodes

maxnode3.add(k)

return maxnode1, maxnode2, maxnode3

top\_deg\_dc, top2\_deg\_dc, top3\_deg\_dc = find\_nodes\_with\_highest\_deg\_cent(G)

print("\nTop three nodes having the highest degree centrality :", top\_deg\_dc, top2\_deg\_dc,

top3\_deg\_dc)

# Degree Centrality

# degree of each node

degree=G.degree(G.nodes())

for i in degree:

print(i,end=" ")

# plot node degree bar graph

from collections import Counter

degree\_dic = dict(G.degree(G.nodes()))

nx.set\_node\_attributes(G, degree\_dic, 'degree')

degree\_hist = pd.DataFrame({"degree": list(degree\_dic.values()),

"Nodes": list(degree\_dic.keys())})

plt.figure(figsize=(10,10))

clrs = ['darkblue' if (x < max(degree\_dic.values())) else 'red' for x in degree\_dic.values() ]

sns.barplot(x = 'degree', y = 'Nodes',

data = degree\_hist,

palette=clrs)

plt.ylabel('Node', fontsize=30)

plt.xlabel('Degree', fontsize=30)

plt.tick\_params(axis='both', which='major',labelsize=10)

plt.show()

# clearly 'jquery' has highest degree

# Degree centrality

dc\_dict = nx.degree\_centrality(G)

nx.set\_node\_attributes(G, dc\_dict, 'degree')

sorted\_dc = sorted(dc\_dict.items(), key=itemgetter(1), reverse=True)

print("Order of nodes According to their importance by using Degree centrality")

for i in sorted\_dc:

print(i)

# TOp 3 nodes with highest Degree centrality

print("\nTop three nodes having highest Degree centrality")

for i in sorted\_dc[:3]:

print(i)

# Eigenvector Centrality

eigenvector\_dict = nx.eigenvector\_centrality(G)

nx.set\_node\_attributes(G, eigenvector\_dict, 'eigenvector')

sorted\_eigenvector = sorted(eigenvector\_dict.items(), key=itemgetter(1), reverse=True)

print("Order of nodes According to their importance by using EigenVector Centrality")

for i in sorted\_eigenvector:

print(i)

# Top 3 nodes with highest Eigenvector centrality

print("\nTop three nodes having highest Eigenvector centrality")

for i in sorted\_eigenvector[:3]:

print(i)

# Betweenness Centrality

betweenness\_dict = nx.betweenness\_centrality(G)

nx.set\_node\_attributes(G, betweenness\_dict, 'betweenness')

sorted\_betweenness = sorted(betweenness\_dict.items(), key=itemgetter(1), reverse=True)

print("Order of nodes According to their importance by using Betweenness Centrality")

for i in sorted\_betweenness:

print(i)

# Top 3 nodes with highest Betweenness centrality

print("\nTop three nodes having highest Betweenness centrality")

for b in sorted\_betweenness[:3]:

print(b)

# shortest path between nodes

nx.shortest\_path(G,'jquery','c#')

nx.shortest\_path(G,'jquery','redux')

nx.shortest\_path(G,'linq','xml')

# from all the 3 centralities common node is 'jquery'

# since it is not connected

# shortest path lengths from highest centrality node 'jquery' are

nx.shortest\_path\_length(G,'jquery')

# Building a subgroup

# We can find the distance of a node from every other node in the network using breadth-first search algorithm, starting from that node. networkX provides the function bfs\_tree to do it.

sub1 = nx.bfs\_tree(G,'jquery')

sub2 = nx.bfs\_tree(G,'css')

# Subgroup (an oriented tree constructed from of a breadth-first-search starting at "jquery")

plt.figure(figsize=(25, 25))

options = {

'edge\_color': '#BAB0AD',

'width': 1,

'with\_labels': True,

'font\_weight': 'normal',

'font\_size': 15,

'style': 'dashed'

}

sizes = [G.nodes[node]['nodesize'] \* 10 for node in G]

nx.draw\_networkx(sub1, pos=nx.spring\_layout(G, k=0.25, iterations=50), \*\*options)

nx.draw\_networkx(sub1.subgraph('jquery'), pos=nx.spring\_layout(G, k=0.25, iterations=50),node\_color='red', \*\*options)

ax = plt.gca()

ax.collections[0].set\_edgecolor("#555555")

plt.show()

# Subgroup (an oriented tree constructed from of a breadth-first-search starting at "css")

plt.figure(figsize=(25, 25))

options = {

'edge\_color': '#BAB0AD',

'width': 1,

'with\_labels': True,

'font\_weight': 'normal',

'font\_size': 15,

'style': 'dashed'

}

sizes = [G.nodes[node]['nodesize'] \* 10 for node in G]

nx.draw\_networkx(sub2, pos=nx.spring\_layout(G, k=0.25, iterations=50), \*\*options)

nx.draw\_networkx(sub2.subgraph('css'), pos=nx.spring\_layout(G, k=0.25, iterations=50),node\_color='red', \*\*options)

ax = plt.gca()

ax.collections[0].set\_edgecolor("#555555")

plt.show()

# Nodes which are connected to important node

print(nx.node\_connected\_component(G,'jquery'))

print("\nTotal {} nodes are connected with main important 'jquery' node".format(len(nx.node\_connected\_component(G,'jquery'))))

# Nodes which are not connected to important node

print(G.nodes()-list(nx.node\_connected\_component(G,'jquery')))

# Node Classification

# harmonic\_function

G.nodes['angular']['label']='web API framework'

G.nodes['css']['label']='web design '

G.nodes['c++']['label']='programming language'

G.nodes['git']['label']='command shell'

G.nodes['linux']['label']='OS'

G.nodes['qt']['label']='GUI'

G.nodes['hibernate']['label']='database'

classs = node\_classification.harmonic\_function(G)

nodes=list(G.nodes())

node\_class={nodes[i]:classs[i] for i in range(len(nodes))}

for i in node\_class:

print(i,"-------->",node\_class.get(i))

# Link prediction

# jaccard coefficient

threshold\_j=0.45

jaccard=list(nx.jaccard\_coefficient(G))

for i in jaccard:

if i[2]>threshold\_j:

print(i)

# Resource Allocation Index (predict missing links, similarity between two nodes)

threshold\_RAI=0.45

RAI=list(nx.resource\_allocation\_index(G))

for i in RAI:

if i[2]>threshold\_RAI:

print(i)

# Adamic adar Index predict missing links in a Network, according to the amount of shared links between two nodes

threshold\_AAI=1

AAI=list(nx.adamic\_adar\_index(G))

for i in AAI:

if i[2]>threshold\_AAI:

print(i)

#preferential Attachment

degree\_dic = dict(G.degree(G.nodes()))

minn=min(degree\_dic.values())

maxx=max(degree\_dic.values())

avg=sum(degree\_dic.values())/len(degree\_dic)

print("Minimum degree = {}\nMaximum degree={}\nAverage Degree={}".format(minn,maxx,avg))

thres\_d=maxx\*avg

PA=list(nx.preferential\_attachment(G))

for i in PA:

if i[2]>thres\_d:

print(i)

# Community Detection

# Greedy Modularity

communities = community.greedy\_modularity\_communities(G)

print(communities)

modularity\_dict = {} # Create a blank dictionary

for i,c in enumerate(communities): # Loop through the list of communities, keeping track of the number for the community

for name in c: # Loop through each person in a community

modularity\_dict[name] = i # Create an entry in the dictionary for the person, where the value is which group they belong to.

# Now you can add modularity information like we did the other metrics

nx.set\_node\_attributes(G, modularity\_dict, 'modularity')

# First get a list of just the nodes in that class

class0 = [n for n in G.nodes() if G.nodes[n]['modularity'] == 0]

# Then create a dictionary of the eigenvector centralities of those nodes

class0\_eigenvector = {n:G.nodes[n]['eigenvector'] for n in class0}

# Then sort that dictionary and print the first 5 results

class0\_sorted\_by\_eigenvector = sorted(class0\_eigenvector.items(), key=itemgetter(1), reverse=True)

print("Modularity Class 0 Sorted by Eigenvector Centrality:")

for node in class0\_sorted\_by\_eigenvector[:5]:

print("Name:", node[0], "| Eigenvector Centrality:", node[1])

for i,c in enumerate(communities): # Loop through the list of communities

if len(c) > 2: # Filter out modularity classes with 2 or fewer nodes

print('Class '+str(i)+':', list(c)) # Print out the classes and their members

# Label Propagation communities

label\_prop\_comm=community.label\_propagation\_communities(G)

j=0

for i in label\_prop\_comm:

print("Class [{}] ----> {}".format(j,i))

j+=1

# Drawing Networks

color\_map = {1: '#f09494', 2: '#eebcbc', 3: '#72bbd0', 4: '#91f0a1', 5: '#629fff', 6: '#bcc2f2',

7: '#eebcbc', 8: '#f1f0c0', 9: '#d2ffe7', 10: '#caf3a6', 11: '#ffdf55', 12: '#ef77aa',

13: '#d6dcff', 14: '#d2f5f0', 15: '#2B2B40', 16: '#e6bbaa', 17: '#c158fd'}

plt.figure(figsize=(25, 25))

options = {

'edge\_color': '#BAB0AD',

'width': 1,

'with\_labels': True,

'font\_weight': 'normal',

'font\_size': 15,

'style': 'dashed'

}

colors = [color\_map[G.nodes[node]['group']] for node in G]

sizes = [G.nodes[node]['nodesize'] \* 20 for node in G]

"""

Using the spring layout :

- k controls the distance between the nodes and varies between 0 and 1

- iterations is the number of times simulated annealing is run

default k=1 and iterations=50

"""

# nx.spring\_layout(G, k=0.25, iterations=50)

nx.draw(G, node\_color=colors, node\_size=sizes, pos=nx.spring\_layout(G, k=1, iterations=50), \*\*options)

ax = plt.gca()

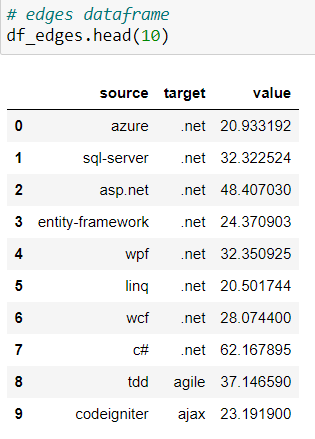
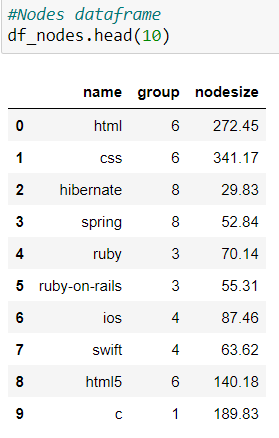
ax.collections[0].set\_edgecolor("#555555")

plt.show()

plt.savefig("Network.png")

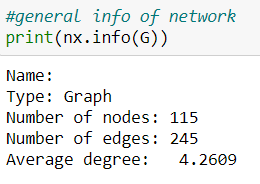
**OUTPUT SCREENSHOTS**

**Datasets are converted into Dataframes using pandas:**

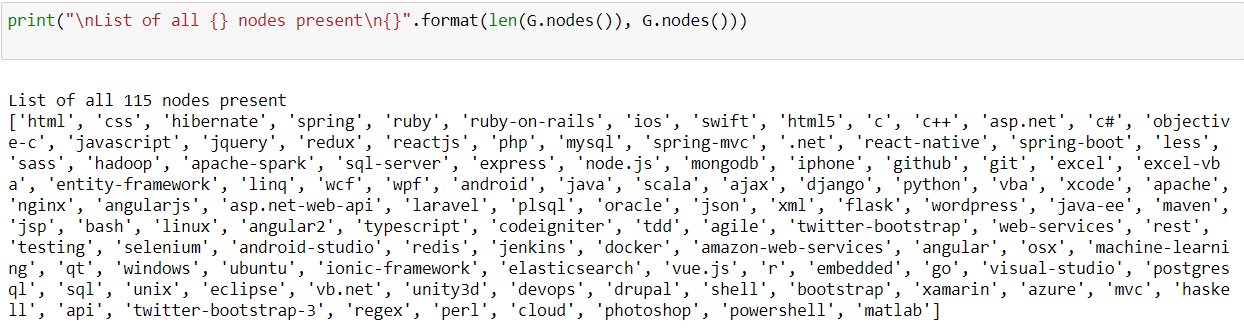
****

1. Nodes, Links dataset are converted into Network using networkx module

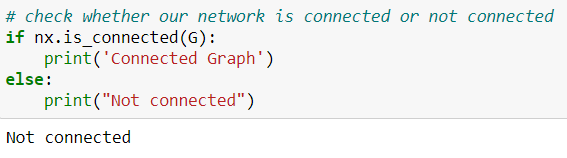
Basic Info of the network is given by “nx.info(G)”.



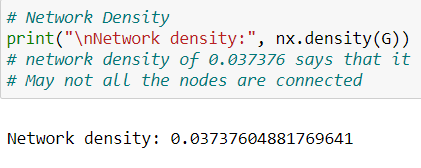
1. List of all nodes present in network given by “G.nodes()”.



1. Check whether the network is connected or not , check whether it is directed or not, check whether the network is weighted or not.



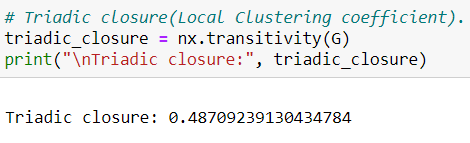
1. Lets see how dense our network is



1. Since our network is not connected. We can determine the diameter of our Network . So, we determine the diameter of a subgroup containing Largest\_component in our Network graph.

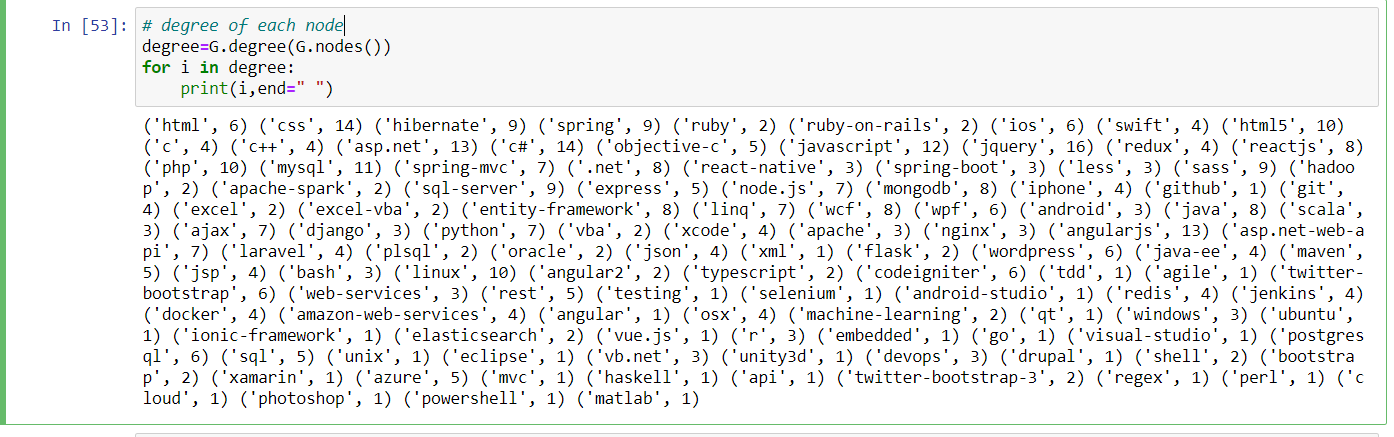


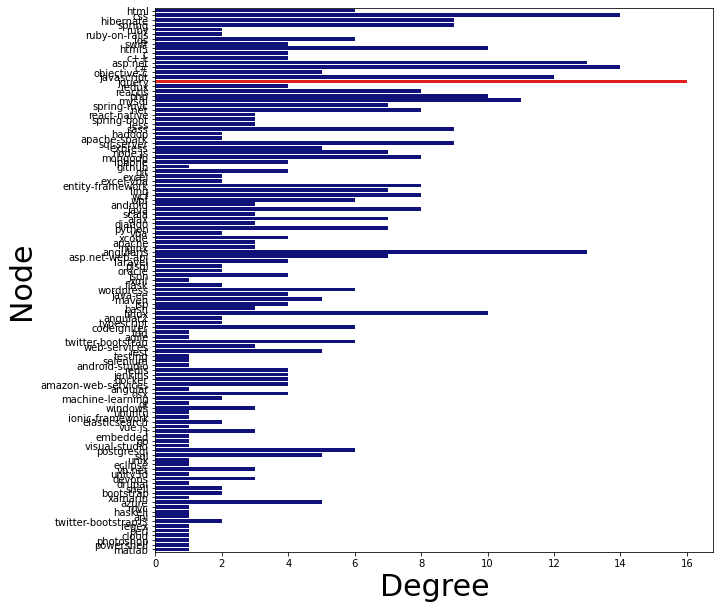
1. Transitivity is the ratio of all triangles over all possible triangles.



Here we can see Triadic Closure is < 0.5 i.e., we can say not well connected.

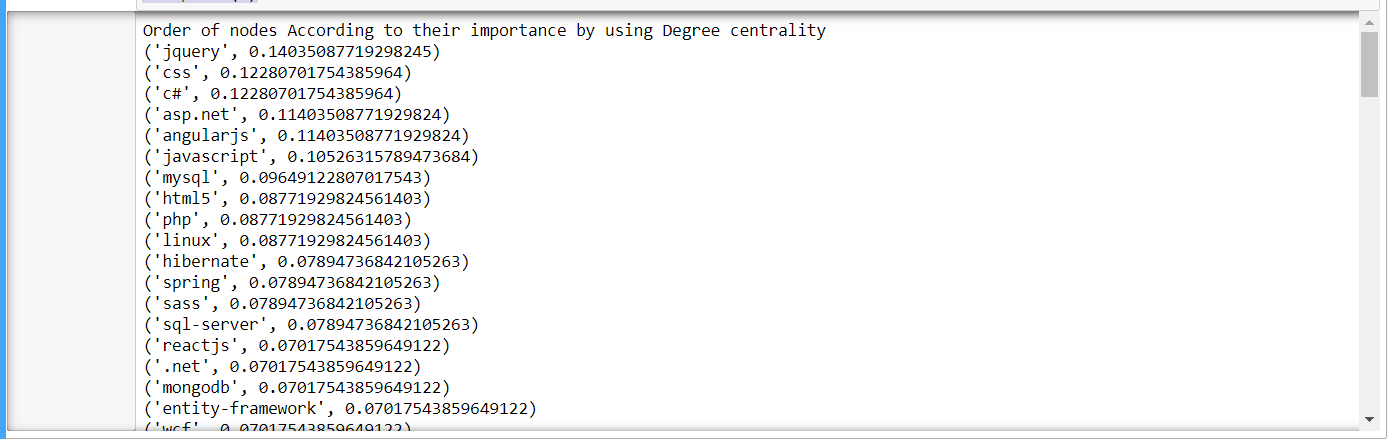
1. Now Let's find centre/important nodes from the Network using Degree centrality, Eigenvector Centrality, Betweenness Centrality.
2. **Degree centrality** - Let's plot a bar graph of all nodes with their node degree .



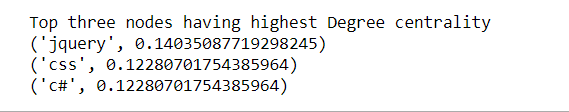


Red colored bar has the highest degree . Therefore, we can say thay ‘jquery’ node has the highest importance.

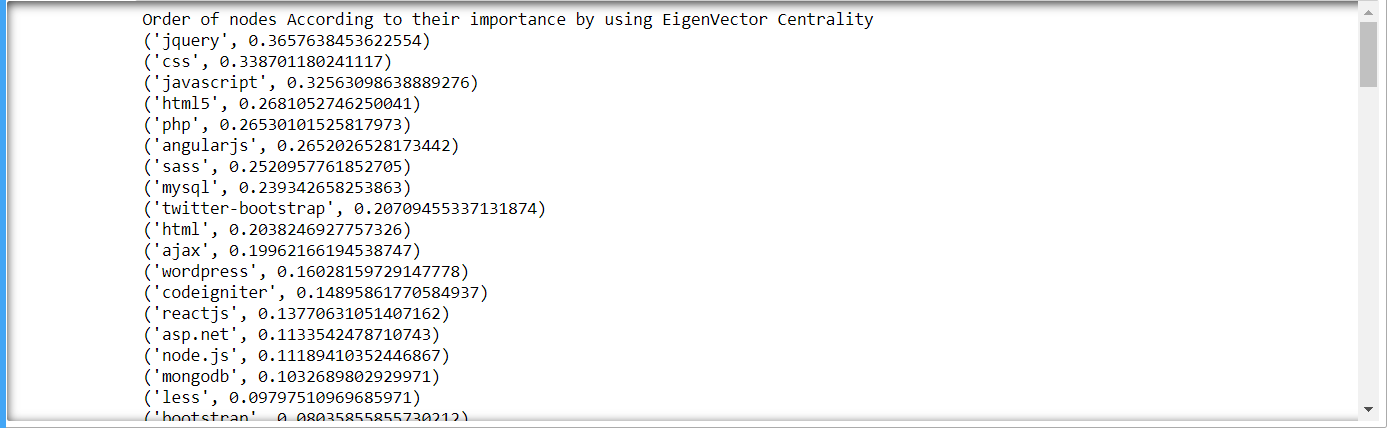
Now let's see the degree centrality of each node.

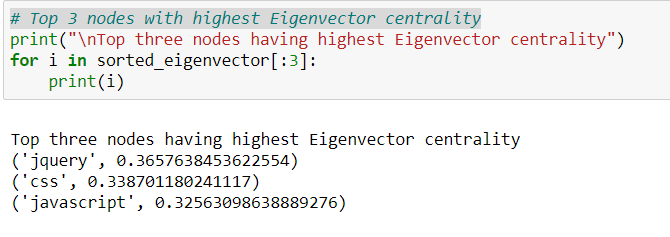


Printing Top 3 important nodes by degree centrality.

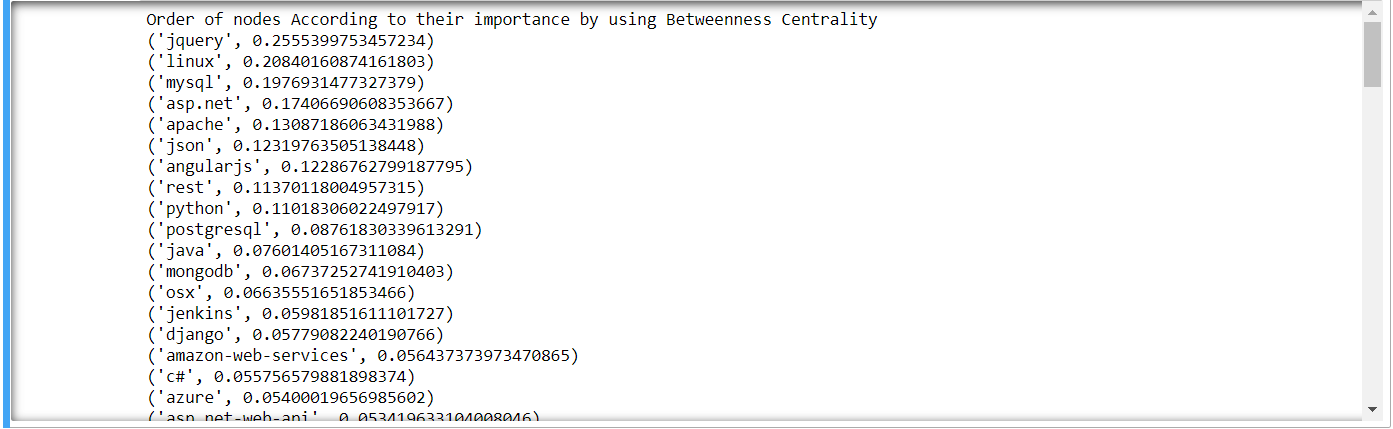


1. **Eigenvector Centrality**

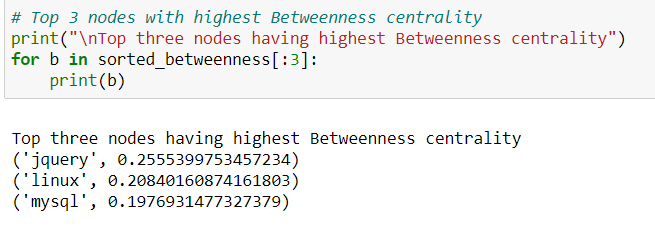
****

Printing Top 3 important nodes by EigenVector centrality.****

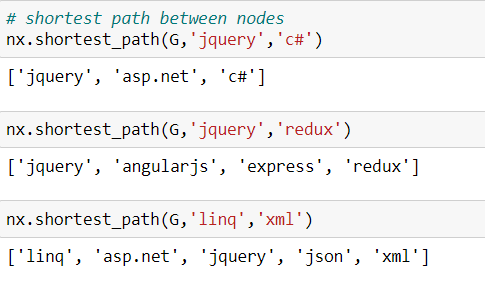
1. **Betweenness Centrality**



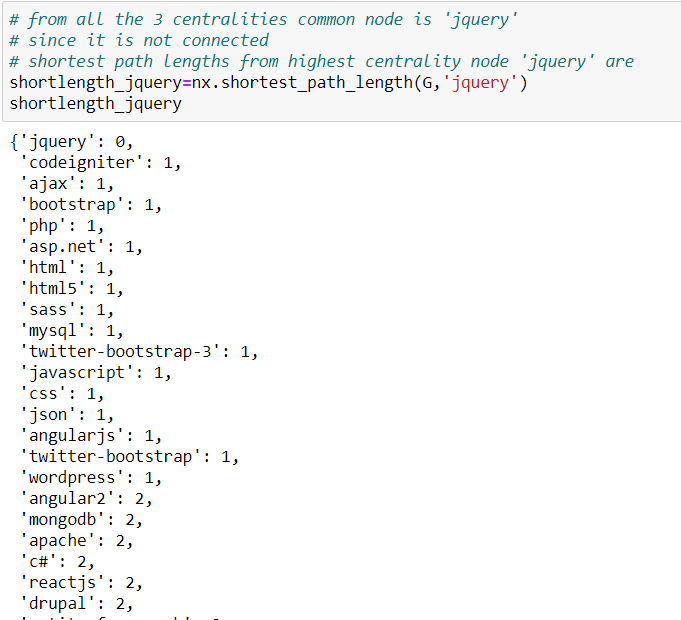
Printing Top 3 important nodes by Betweenness centrality.



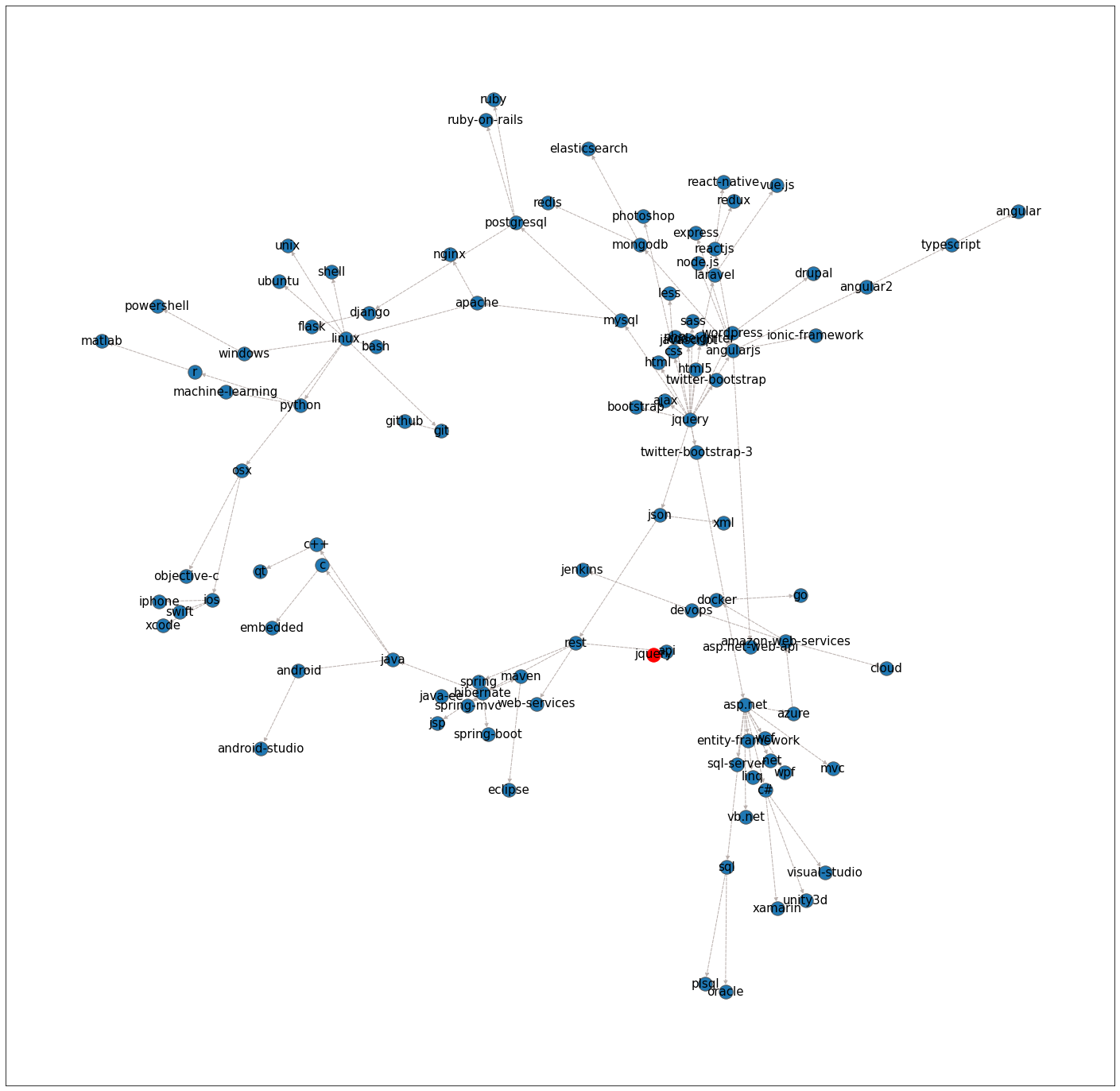
8. We can find the shortest path and its length between any two nodes using “nx.shortest\_path(G,src,des), nx.shortest\_path\_len(G,src,des)”.



9. Important node is ‘jquery’. Printing all shortest paths length from node ‘jquery’.

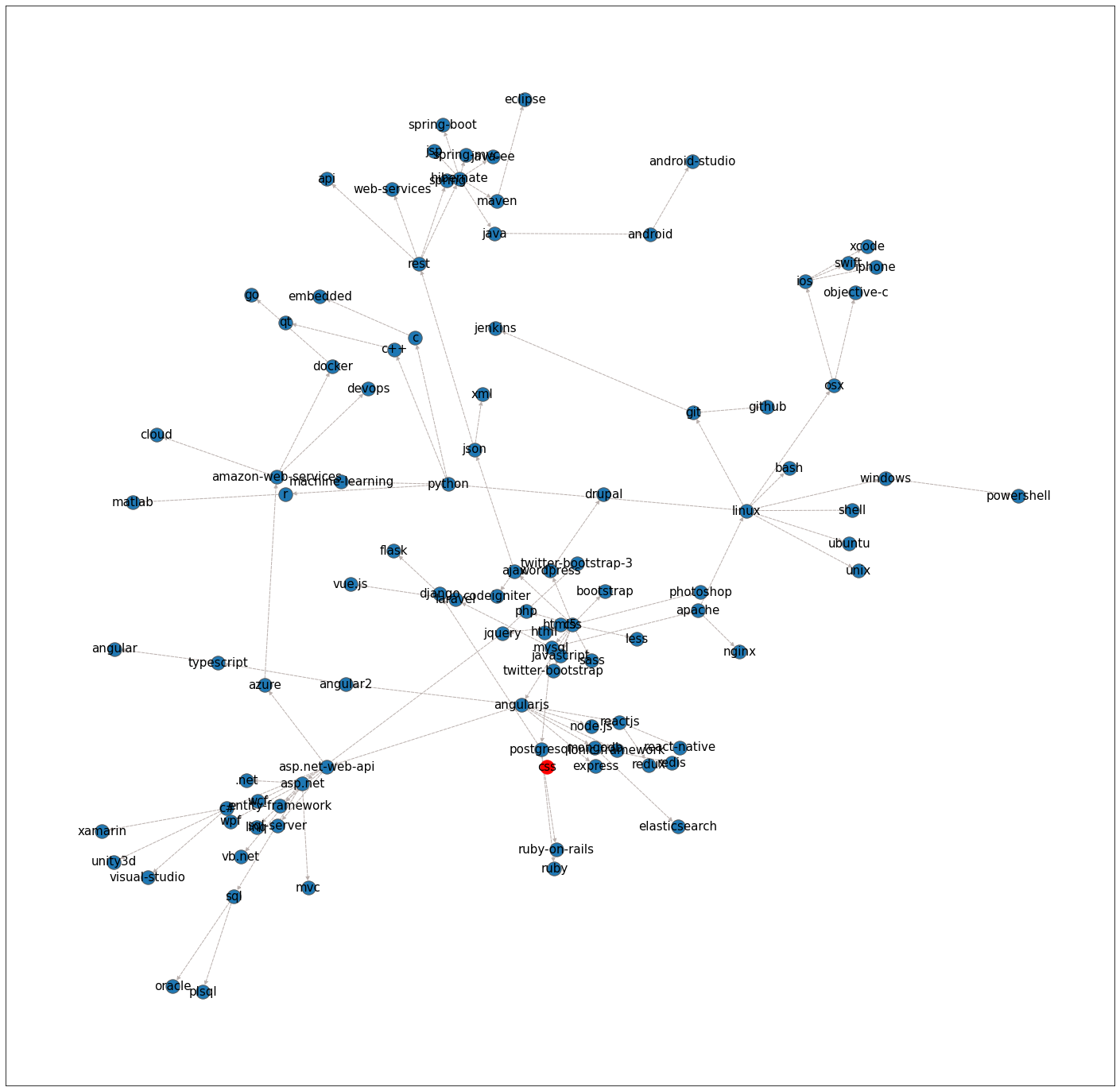


10. Plotting paths of node ‘jquery’ and ‘css’ from every other node in the network using breadth-first search algorithm, starting from that node. networkX provides the function bfs\_tree to do it.



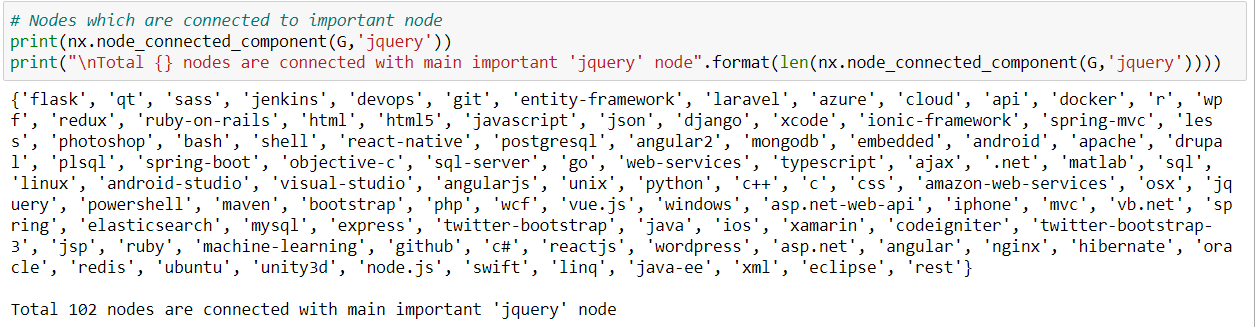
All paths possible from ‘Jquery’ to all other nodes.

All paths possible from node ‘css’ to all other nodes.

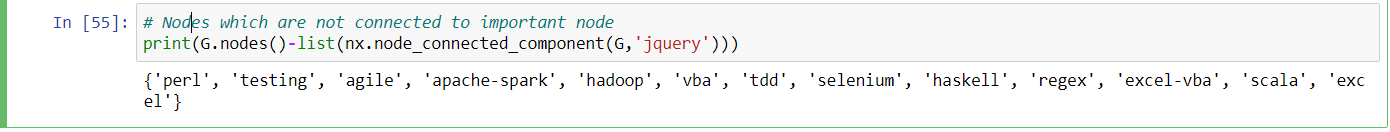


11. Let us print which nodes are not connected and not connected to the important node ‘jquery’.

Connected to ‘jquery’

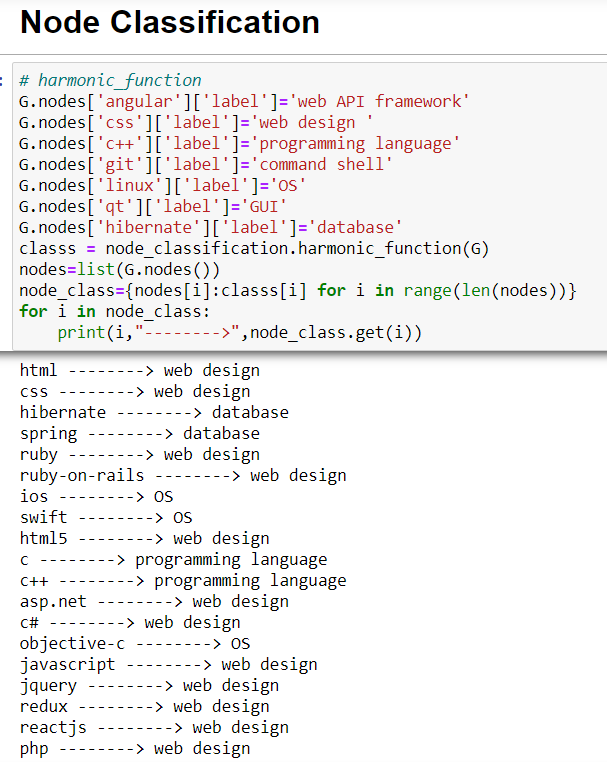


Not Connected to ‘jquery’



12. Node Classification using Harmonic Function algorithm.

Few nodes are labeled but many nodes are not labelled this harmonic function labels the nodes which are not labels at begin.

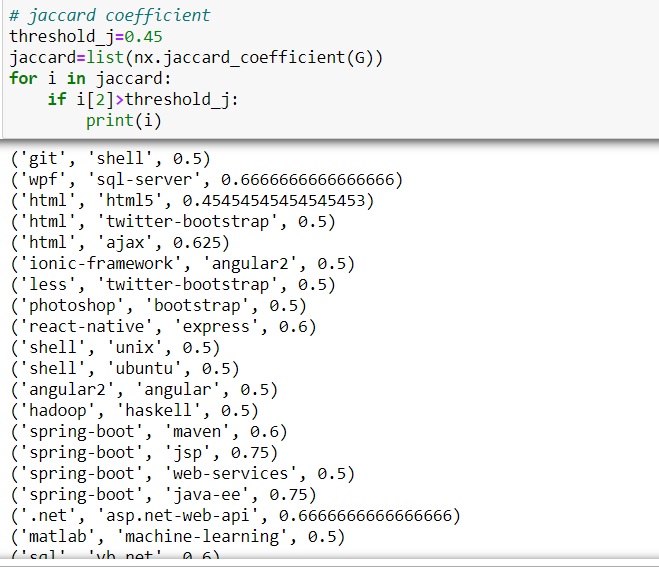




13. **Link Prediction**

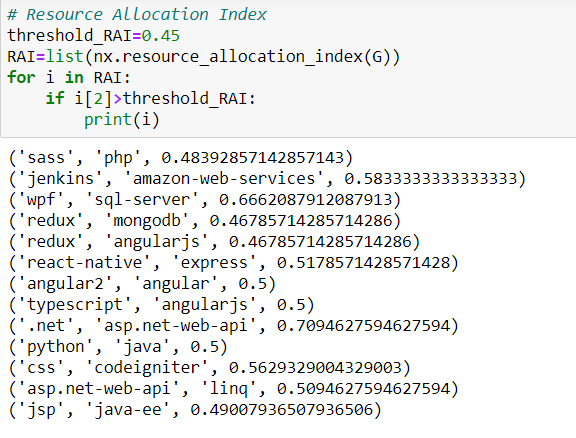
1. **Jaccard coefficient:**

threshold\_value=0.45 , the new links with jaccard coefficient greater than 0.45 edges are printed.



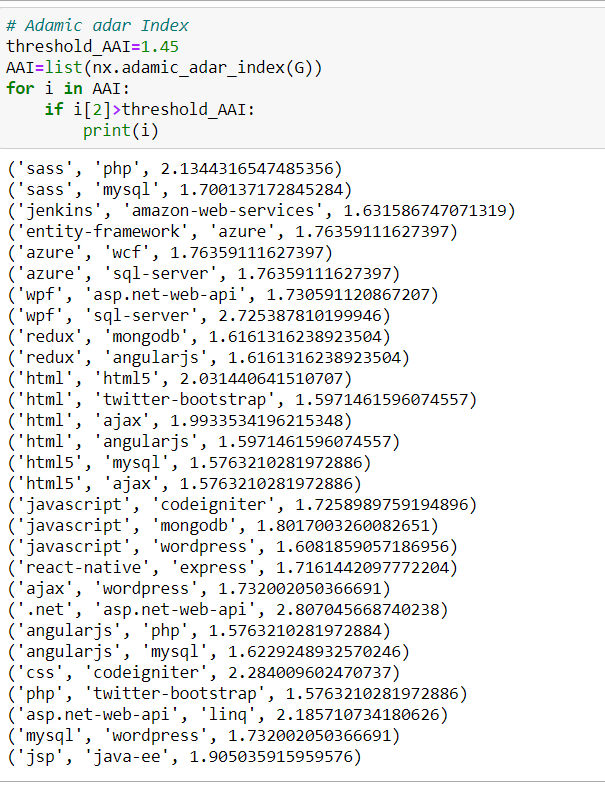
1. **Resource Allocation Index(RAI):**

threshold=0.45, the new edges with RAI greater than 0,45 are considered.



1. **Adamic Adar Index(AAI):**

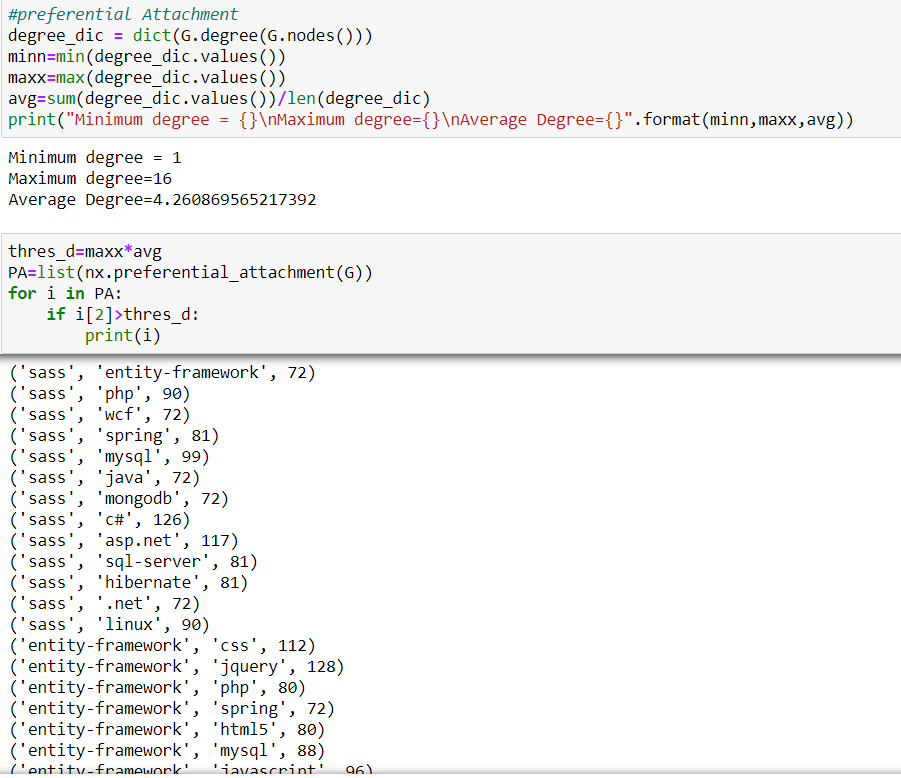
threshold=1.45, the new links with AAI > 1.45 are considered.



1. **Preferential Attachment(PA):**

threshold=maximun\_degree \* average\_degree\_of\_all\_nodes

The new links with PA > threshold are taken into consideration.



**14. Community Detection:**

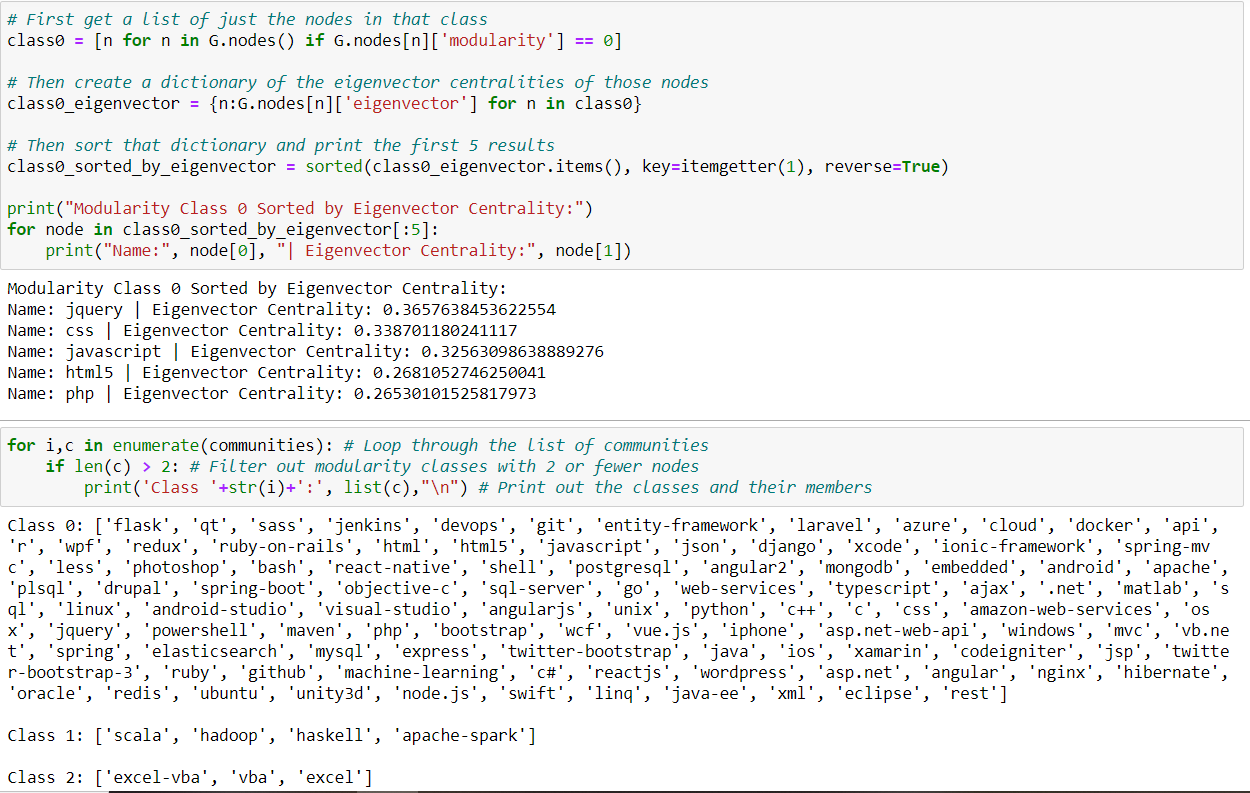
Grouping nodes into subgroups/communities using Modularity and Label Propagation.

1. Community detection using Greedy modularity.

This is an inbuilt algorithm.



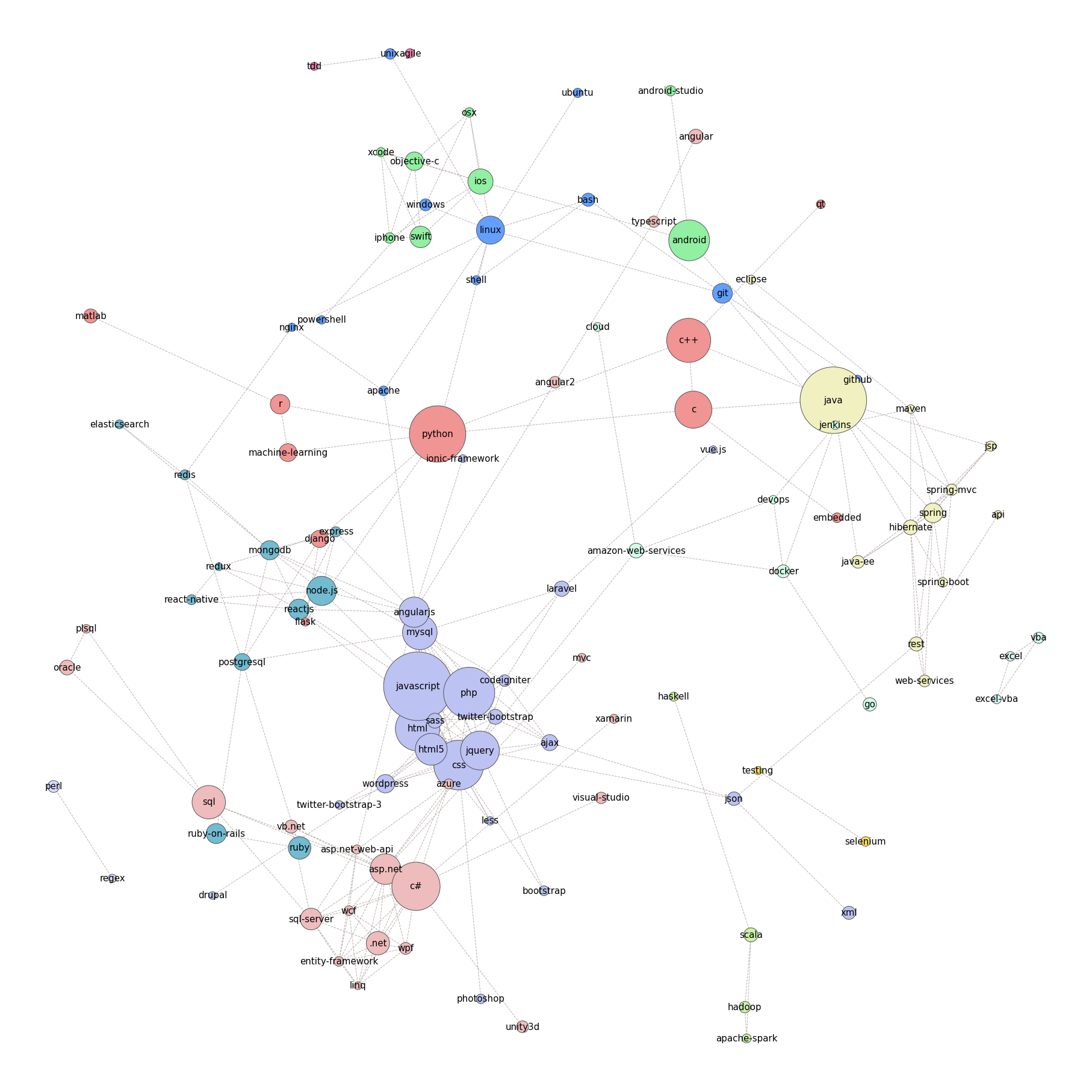
**This is done using eigenvector centrality.**

****

Another Community detection technique used is **Label Propagation Community:**

****

**15. Plotting Network:**

****

**HARDWARE AND SOFTWARE REQUIREMENTS**

|  |  |
| --- | --- |
| **HARDWARE TOOLS** | **MINIMUM REQUIREMENTS** |
| Processor | i5 or above |
| Hard disk | 10GB |
| RAM | 8GB |
| Monitor | 17’’ Coloured |
| Mouse | Optical |
| Keyboard | 122 Keys |

|  |  |
| --- | --- |
| **SOFTWARE TOOLS** | **MINIMUM REQUIREMENTS** |
| Platform | Windows/Linux/MacOS |
| Operating System | Windows/Linux/MacOS |
| Technology | Machine learning – Python |
| Scripting language | Python |
| IDE | Pycharm & Jupyter notebook |

**FUTURE SCOPE**

Now-a-days trending apps like twitter, facebook, maps, netflix, and many others are networks and these functionalities can be improved by network graph analysis. Friend suggestions, Nearest Person to you, People you may know these all are derived using Network analysis.

And also fields like marketing, stocks, blockchain, Telecom are like networks these can be improvised more using Network analysis. These marketing and other strategies can be done using link prediction , communities, marketing and advertising will be easier

**CONCLUSION**

We conclude that using Network Graph Analysis, We can predict the future node, relation between nodes, divide nodes into communities, and also analysing Network data give you more information in a short time.

By using social network analysis techniques to identify information about people, their role in the network, and their relationships can be useful to help sort, aggregate, and filter information in social media.

People can share content directly by sharing links on their social networking pages, posting them on social-sharing websites like reddit, or more passively sharing with social readers that show friends everything a person has looked at. These methods highlight information that a person’s friends have found interesting, and that filter is often very useful for identifying good content. Further user input, like votes up or down on content, can further help sort and filter information shared in this way.

Recommender systems move up a level, aggregating ratings or behavior and using that to personalize suggestions for items that a person might like. Collaborative filtering systems use similarity estimates to show items that people similar to the user like. Social recommender systems replace or enhance the similarity measures with social features, like trust relationships, to recommend items. This leverages social information in several ways, using ratings that users supply and their social connections to highlight interesting items.

Leveraging social information is already effective for finding information, and as the amount of information people encounter online grows, it will be important to develop new methods that incorporate social information and techniques for filtering, sorting, and aggregating content.

**BIBLIOGRAPHY AND REFERENCES**

* Zhu, X., Ghahramani, Z., & Lafferty, J. (2003, August). Semi-supervised learning using gaussian fields and harmonic functions. In ICML (Vol. 3, pp. 912-919).
* networkx(<http://networkx.lanl.gov/>)
* Link detection Algorithms networkx (<https://www.geeksforgeeks.org/link-prediction-predict-edges-in-a-network-using-networkx/>)
* Node Detection Algorithms (https://networkx.org/documentation/stable/reference/algorithms/node\_classification.html)
* Community Detection Algorithms networkx (<https://networkx.org/documentation/stable/reference/algorithms/community.html#module-networkx.algorithms.community.kernighan_lin>)
* Dataset(<https://github.com/>)
* Algorithms(<https://www.hindawi.com/journals/mpe/2013/125123/>)
* matpltolib(<https://matplotlib.org/>)
* Cordasco, G., & Gargano, L. (2010, December). Community detection via semi-synchronous label propagation algorithms. In Business Applications of Social Network Analysis (BASNA), 2010 IEEE International Workshop on (pp. 1-8). IEEE.