

**Laboratory Manual**

**Subject: Machine Learning–IV (PCCS7010T) Semester: VII Class: B. Tech Experiment No. :5**

**Title :** Compute stochastic matrix from a given graph, compute PageRank vector and return the results.

**Pre-requisite:** Basics of Machine Learning

Hardware Requirements (if any): Windows 64 bit processor

Theory/Concept Explanation (Compulsory):

Introduction:

PageRank (PR) is an algorithm used by Google Search to rank websites in their search engine results. PageRank was named after Larry Page, one of the founders of Google. PageRank is a way of measuring the importance of website pages. According to Google:

*PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites.*

It is not the only algorithm used by Google to order search engine results, but it is the first algorithm that was used by the company, and it is the best-known.  
The above centrality measure is not implemented for multi-graphs.

***Algorithm***   
The PageRank algorithm outputs a probability distribution used to represent the likelihood that a person randomly clicking on links will arrive at any particular page. PageRank can be calculated for collections of documents of any size. It is assumed in several research papers that the distribution is evenly divided among all documents in the collection at the beginning of the computational process. The PageRank computations require several passes, called “iterations”, through the collection to adjust approximate PageRank values to more closely reflect the theoretical true value.

**Simplified algorithm**   
Assume a small universe of four web pages: A, B, C, and D. Links from a page to itself, or multiple outbound links from one single page to another single page, are ignored. PageRank is initialized to the same value for all pages. In the original form of PageRank, the sum of PageRank over all pages was the total number of pages on the web at that time, so each page in this example would have an initial value of 1. However, later versions of PageRank, and the remainder of this section, assume a probability distribution between 0 and 1. Hence the initial value for each page in this example is 0.25.  
The PageRank transferred from a given page to the targets of its outbound links upon the next iteration is divided equally among all outbound links.  
If the only links in the system were from pages B, C, and D to A, each link would transfer 0.25 PageRank to A upon the next iteration, for a total of 0.75.



Suppose instead that page B had a link to pages C and A, page C had a link to page A, and page D had links to all three pages. Thus, upon the first iteration, page B would transfer half of its existing value, or 0.125, to page A and the other half, or 0.125, to page C. Page C would transfer all of its existing value, 0.25, to the only page it links to, A. Since D had three outbound links, it would transfer one-third of its existing value, or approximately 0.083, to A. At the completion of this iteration, page A will have a PageRank of approximately 0.458.



In other words, the PageRank conferred by an outbound link is equal to the document’s own PageRank score divided by the number of outbound links L ( ).



In the general case, the PageRank value for any page u can be expressed as:



i.e. the PageRank value for a page u is dependent on the PageRank values for each page v contained in the set Bu (the set containing all pages linking to page u), divided by the number L(v) of links from page v. The algorithm involves a damping factor for the calculation of the PageRank. It is like the income tax which the govt extracts from one despite paying him itself.

**Implementation:**

def pagerank(G, alpha=0.85, personalization=None,

max\_iter=100, tol=1.0e-6, nstart=None, weight='weight',

dangling=None):

"""Return the PageRank of the nodes in the graph.

PageRank computes a ranking of the nodes in the graph G based on

the structure of the incoming links. It was originally designed as

an algorithm to rank web pages.

Parameters

----------

G : graph

A NetworkX graph. Undirected graphs will be converted to a directed

graph with two directed edges for each undirected edge.

alpha : float, optional

Damping parameter for PageRank, default=0.85.

personalization: dict, optional

The "personalization vector" consisting of a dictionary with a

key for every graph node and nonzero personalization value for each node.

By default, a uniform distribution is used.

max\_iter : integer, optional

Maximum number of iterations in power method eigenvalue solver.

tol : float, optional

Error tolerance used to check convergence in power method solver.

nstart : dictionary, optional

Starting value of PageRank iteration for each node.

weight : key, optional

Edge data key to use as weight. If None weights are set to 1.

dangling: dict, optional

The outedges to be assigned to any "dangling" nodes, i.e., nodes without

any outedges. The dict key is the node the outedge points to and the dict

value is the weight of that outedge. By default, dangling nodes are given

outedges according to the personalization vector (uniform if not

specified). This must be selected to result in an irreducible transition

matrix (see notes under google\_matrix). It may be common to have the

dangling dict to be the same as the personalization dict.

Returns

-------

pagerank : dictionary

Dictionary of nodes with PageRank as value

Notes

-----

The eigenvector calculation is done by the power iteration method

and has no guarantee of convergence. The iteration will stop

after max\_iter iterations or an error tolerance of

number\_of\_nodes(G)\*tol has been reached.

The PageRank algorithm was designed for directed graphs but this

algorithm does not check if the input graph is directed and will

execute on undirected graphs by converting each edge in the

directed graph to two edges.

"""

if len(G) == 0:

return {}

if not G.is\_directed():

D = G.to\_directed()

else:

D = G

# Create a copy in (right) stochastic form

W = nx.stochastic\_graph(D, weight=weight)

N = W.number\_of\_nodes()

# Choose fixed starting vector if not given

if nstart is None:

x = dict.fromkeys(W, 1.0 / N)

else:

# Normalized nstart vector

s = float(sum(nstart.values()))

x = dict((k, v / s) for k, v in nstart.items())

if personalization is None:

# Assign uniform personalization vector if not given

p = dict.fromkeys(W, 1.0 / N)

else:

missing = set(G) - set(personalization)

if missing:

raise NetworkXError('Personalization dictionary '

'must have a value for every node. '

'Missing nodes %s' % missing)

s = float(sum(personalization.values()))

p = dict((k, v / s) for k, v in personalization.items())

if dangling is None:

# Use personalization vector if dangling vector not specified

dangling\_weights = p

else:

missing = set(G) - set(dangling)

if missing:

raise NetworkXError('Dangling node dictionary '

'must have a value for every node. '

'Missing nodes %s' % missing)

s = float(sum(dangling.values()))

dangling\_weights = dict((k, v/s) for k, v in dangling.items())

dangling\_nodes = [n for n in W if W.out\_degree(n, weight=weight) == 0.0]

# power iteration: make up to max\_iter iterations

for \_ in range(max\_iter):

xlast = x

x = dict.fromkeys(xlast.keys(), 0)

danglesum = alpha \* sum(xlast[n] for n in dangling\_nodes)

for n in x:

# this matrix multiply looks odd because it is

# doing a left multiply x^T=xlast^T\*W

for nbr in W[n]:

x[nbr] += alpha \* xlast[n] \* W[n][nbr][weight]

x[n] += danglesum \* dangling\_weights[n] + (1.0 - alpha) \* p[n]

# check convergence, l1 norm

err = sum([abs(x[n] - xlast[n]) for n in x])

if err < N\*tol:

return x

raise NetworkXError('pagerank: power iteration failed to converge '

'in %d iterations.' % max\_iter)

The above code is the function that has been implemented in the networkx library.

To implement the above in networkx, you will have to do the following:

>>> import networkx as nx

>>> G=nx.barabasi\_albert\_graph(60,41)

>>> pr=nx.pagerank(G,0.4)

>>> pr

**Output:**

**{0: 0.012774147598875784, 1: 0.013359655345577266, 2: 0.013157355731377924,**

**3: 0.012142198569313045, 4: 0.013160014506830858, 5: 0.012973342862730735,**

**6: 0.012166706783753325, 7: 0.011985935451513014, 8: 0.012973502696061718,**

**9: 0.013374146193499381, 10: 0.01296354505412387, 11: 0.013163220326063332,**

**12: 0.013368514624403237, 13: 0.013169335617283102, 14: 0.012752071800520563,**

**15: 0.012951601882210992, 16: 0.013776032065400283, 17: 0.012356820581336275,**

**18: 0.013151652554311779, 19: 0.012551059531065245, 20: 0.012583415756427995,**

**21: 0.013574117265891684, 22: 0.013167552803671937, 23: 0.013165528583400423,**

**24: 0.012584981049854336, 25: 0.013372989228254582, 26: 0.012569416076848989,**

**27: 0.013165322299539031, 28: 0.012954300960607157, 29: 0.012776091973397076,**

**30: 0.012771016515779594, 31: 0.012953404860268598, 32: 0.013364947854005844,**

**33: 0.012370004022947507, 34: 0.012977539153099526, 35: 0.013170376268827118,**

**36: 0.012959579020039328, 37: 0.013155319659777197, 38: 0.013567147133137161,**

**39: 0.012171548109779459, 40: 0.01296692767996657, 41: 0.028089802328702826,**

**42: 0.027646981396639115, 43: 0.027300188191869485, 44: 0.02689771667021551,**

**45: 0.02650459107960327, 46: 0.025971186884778535, 47: 0.02585262571331937,**

**48: 0.02565482923824489, 49: 0.024939722913691394, 50: 0.02458271197701402,**

**51: 0.024263128557312528, 52: 0.023505217517258568, 53: 0.023724311872578157,**

**54: 0.02312908947188023, 55: 0.02298716954828392, 56: 0.02270220663300396,**

**57: 0.022060403216132875, 58: 0.021932442105075004, 59: 0.021643288632623502}**