PageRank算法实现和理论作业

汪楚文 2018202114 06/20/2020

实验概述

[实验环境]

Ubuntu 18.04.4 LTS (GNU/Linux 4.15.0-88-generic x86_64)

Or

macOS Catalina 10.15.1

[Python版本]

Python 3.7.4

[Documents]

```
2018202114-汪楚文-PageRank/
|
|-- 2018202114-PageRank.py
|-- 2018202114-PageRank实验报告.pdf
```

超链接：点击此处可在github中查看该项目]

showCase：点击此处可在bilibili观看基础演示视频1

showCase：点击此处可在bilibili观看Epinion演示视频1

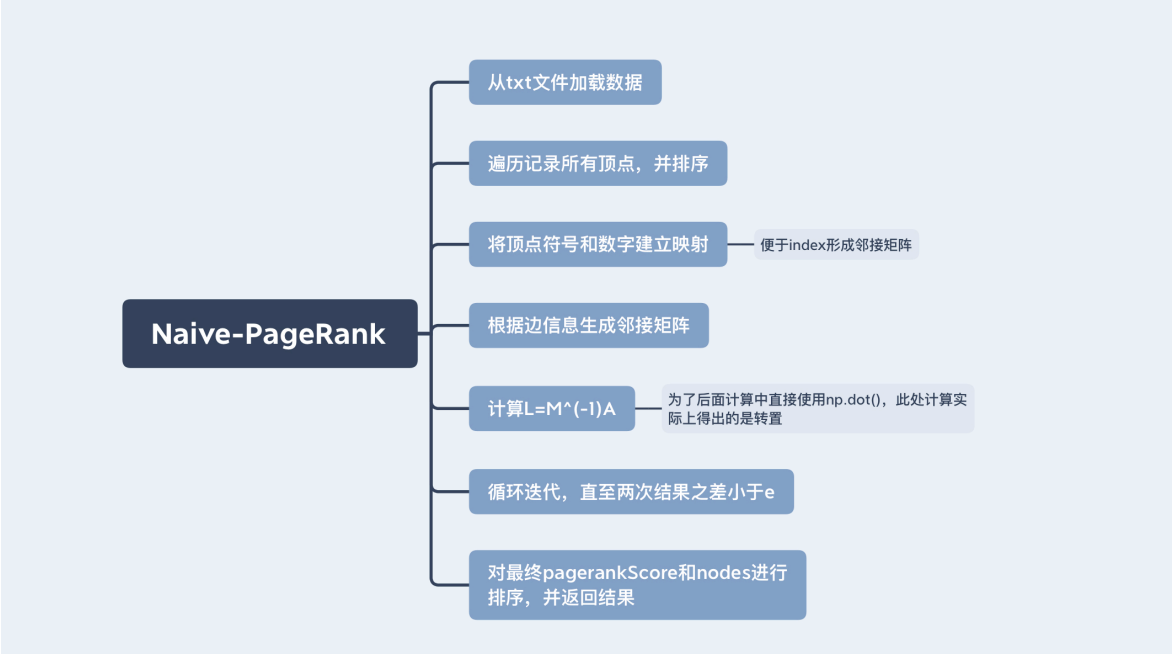
showCase：点击此处可在bilibili观看Epinion演示视频2

一共三个演示视频

实验内容

一.基本的PageRank 函数实现

[思路框架]



[具体思路]

（1）用edges = [line.strip('\n').split(',') for line in f]保存有向图的边

（2） 遍历edges[0]和edges[1]，将所有顶点加入nodes=[]

（3）由于nodes中保存的是顶点字符，为了形成邻接矩阵，需要对这些顶点进行编号

（4）首先按照字典顺序将顶点名称排序

（5）用字典node_to_num = {}来建立nodes到0～N-1的一个一一映射

（6）用for edge in edges:S[edge[1], edge[0]] = 1初始化邻接矩阵

（7） S[i, j] /= sum_of_col来生成L矩阵

（8）A = damping_factor * S + (1 - damping_factor) / N * np.ones([N, N])用于迭代

（9）P_n1 = np.dot(A, P_n)迭代至e <= 0.000001

(10)P_n,node_list_in_descending_order=zip(*sorted(zip(P_n,nodes),reverse=True))

依照P_n的大小顺序，排列nodes的顺序，并返回得到结果

[Remark]

在第（7）步中sum_of_col可能为0，其对应一个顶点没有出度的情况。在这种情况下，将这个顶点指向其他所有顶点：if sum_of_col==0: S[i,j] = 1/N

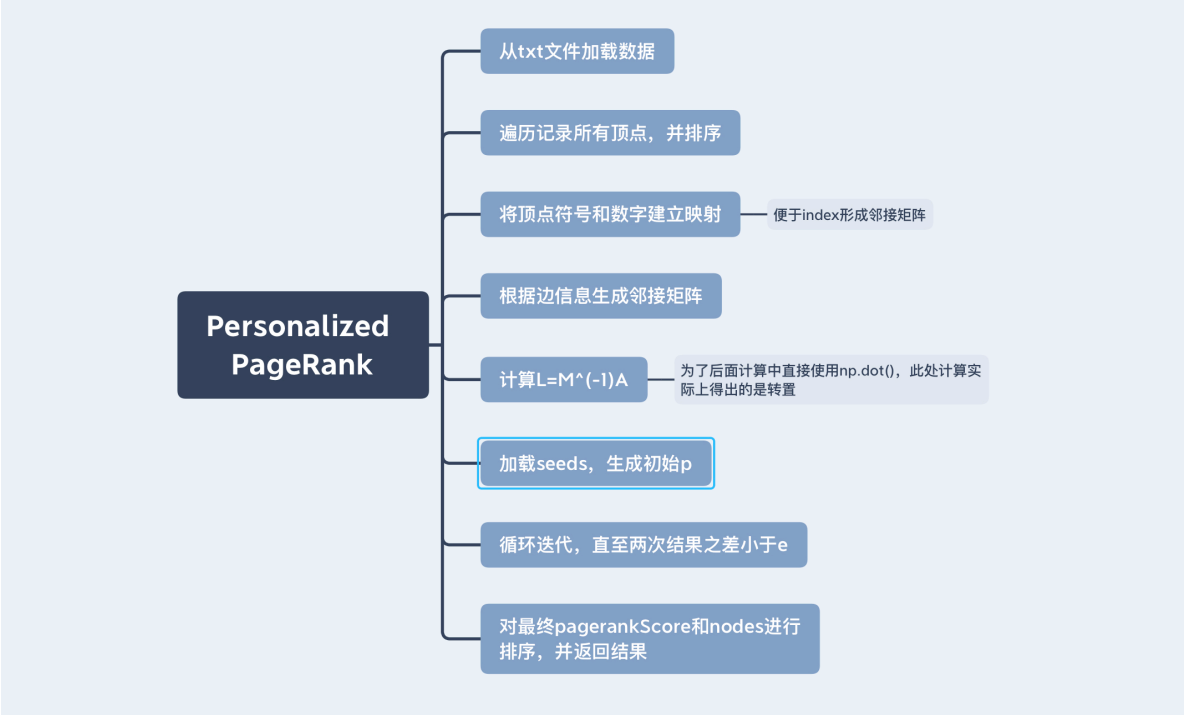
在（10）步中根据P_n来排列nodes，P_n[i]和P_n[j]相等时，由于nodes原来已经按字典顺序sort过，故此时仍为字典顺序

[实验结果]PageRank Score: [0.35895541 0.34261258 0.18311004 0.11532197]

PageRank: ['a', 'b', 'c', 'd']

二.Personalized PageRank 函数实现

[思路框架]（基本同上）



[具体思路]（基本同上）

（1）用edges = [line.strip('\n').split(',') for line in f]保存有向图的边
（2） 遍历edges[0]和edges[1]，将所有顶点加入nodes=[]
（3）由于nodes中保存的是顶点字符，为了形成邻接矩阵，需要对这些顶点进行编号
（4）首先按照字典顺序将顶点名称排序
（5）用字典node_to_num = {}来建立nodes到0～N-1的一个一一映射
（6）用for edge in edges:S[edge[1], edge[0]] = 1初始化邻接矩阵
（7） S[i, j] /= sum_of_col来生成L矩阵
（8）A = damping_factor * S 用于迭代
（9）加载seeds，seeds = [line.strip('\n').split(',') for line in f2]
（10）初始向量for seed in seeds:p_init[node_to_num[seed[0]]]=seed[1]
（11）np.dot(A, P_n)+(1-damping_factor)*p_init迭代至e <= 0.000001
(12)P_n,node_list_in_descending_order=zip(*sorted(zip(P_n,nodes),reverse=True))
依照P_n的大小顺序，排列nodes的顺序，并返回得到结果

[Remark]（同Naive-PageRank上）

[实验结果]PPR Score: [0.35895541 0.34261258 0.18311004 0.11532197]
PPR: ['a', 'b', 'c', 'd']

Epinions数据集上的实验结果

数据预处理：

程序编号	1	文件名	data.py	说明	用于对Epinion数据集进行预处理

```
if __name__ == '__main__':
    f = open('soc-Epinions1.txt', 'r')
    f.readline()
    f.readline()
    f.readline()
    f.readline()
    edges = [line.strip('\n').split('\t') for line in f]
    f1 = open('data.txt','w')
    for edge in edges:
        f1.write(edge[0]+','+edge[1]+'\n')
```

写在前面

对于稀疏图，采用邻接矩阵效率过低，运算量很大。对于稀疏图，考虑用邻接链表代替邻接矩阵。运算以数为单位，不以矩阵为单位，可大大降低运算量。

程序编号	2	文件名	hello.py	说明	用于对Epinion数据集的pagerank

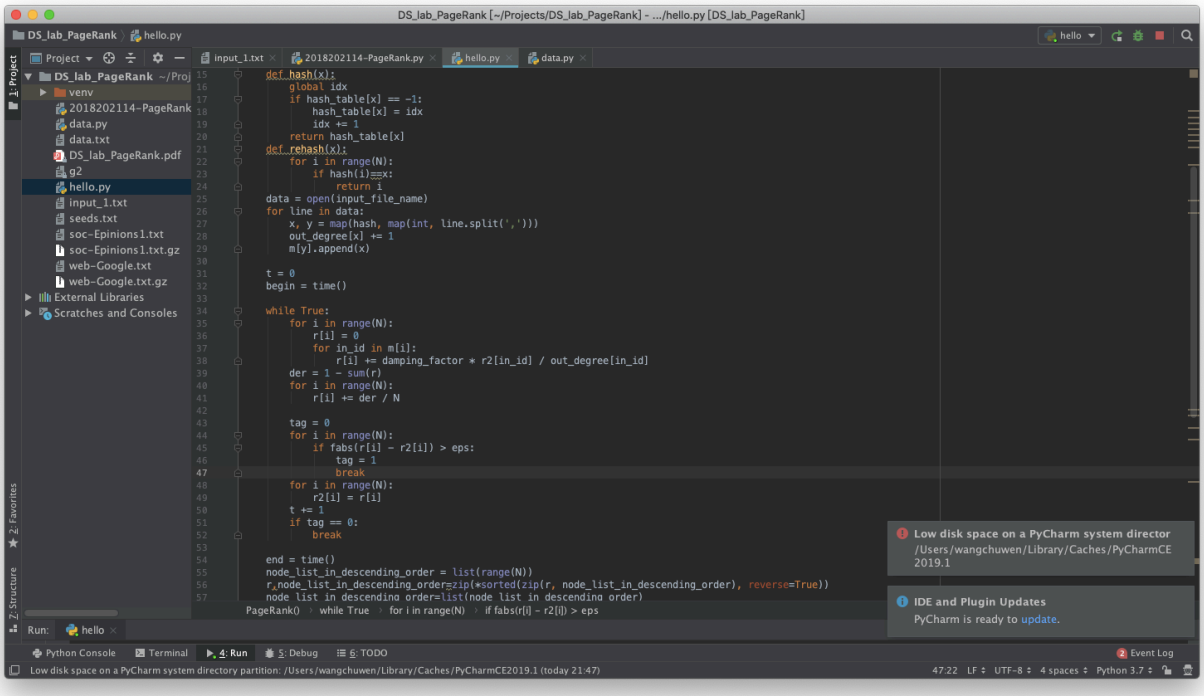
```
from math import fabs
from time import time

idx = 0

def PageRank(input_file_name='data.txt',damping_factor=0.85):
    N = 75879
    r = [1. / N for i in range(N)]
    r2 = [1. / N for i in range(N)]
    out_degree = [0 for i in range(N)]

    hash_table = [-1 for i in range(N * 2)]
    m = [[] for i in range(N * 2)]
    eps = 1e-6
    def hash(x):
        global idx
        if hash_table[x] == -1:
            hash_table[x] = idx
            idx += 1
        return hash_table[x]
```

```
    def rehash(x):
        for i in range(N):
            if hash(i)==x:
                return i
    data = open(input_file_name)
    for line in data:
        x, y = map(hash, map(int, line.split(',')))
        out_degree[x] += 1
        m[y].append(x)

    t = 0
    begin = time()

    while True:
        for i in range(N):
            r[i] = 0
            for in_id in m[i]:
                r[i] += damping_factor * r2[in_id] / out_degree[in_id]
        der = 1 - sum(r)
        for i in range(N):
            r[i] += der / N

        tag = 0
        for i in range(N):
            if fabs(r[i] - r2[i]) > eps:
                tag = 1
                break
        for i in range(N):
            r2[i] = r[i]
        t += 1
        if tag == 0:
            break

    end = time()
    node_list_in_descending_order = list(range(N))
    r,node_list_in_descending_order=zip(*sorted(zip(r,
node_list_in_descending_order), reverse=True))
    node_list_in_descending_order=list(node_list_in_descending_order)
    for i in range(10):
        node_list_in_descending_order[i]=rehash(node_list_in_descending_order[i])
    for i in range(10):
        print('TOP %s\t' %(i+1))
        print(node_list_in_descending_order[i])
        print('Score:')
        print(r[i])
        print('\n')
    print('total iteration is %d' % t)
    print('total time is %f' % (end - begin))
if __name__ == '__main__':
    PageRank()
```

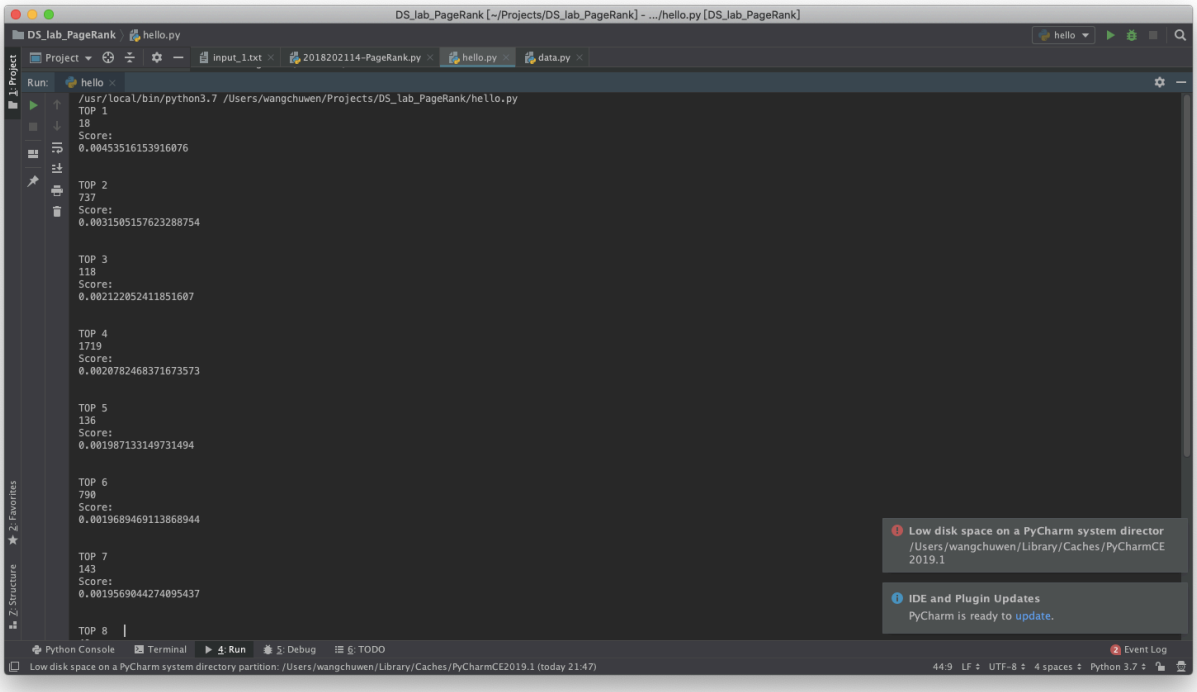
一.Naive PageRank结果

showCase：点击此处可在bilibili观看Epinion演示视频1

[运行截图]



[结果截图]:



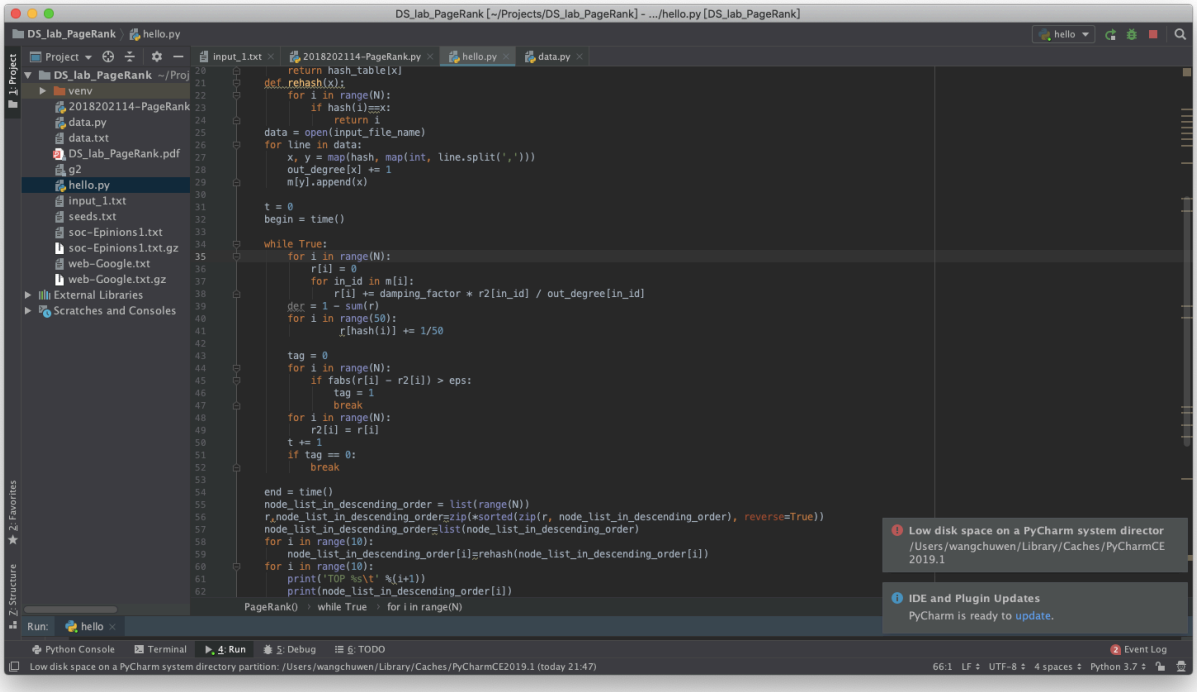
[top10]

TOP 10	SCORE	
18	0.00453516153916076	total iteration is 32
737	0.00315051576232888	
118	0.00212205241185161	
1719	0.00207824683716736	
136	0.00198713314973149	
790	0.00196894691138689	total time is 8.740246s
143	0.00195690442740954	
40	0.0018249277784487	
1619	0.0015362938978724	
725	0.00149605923367339	

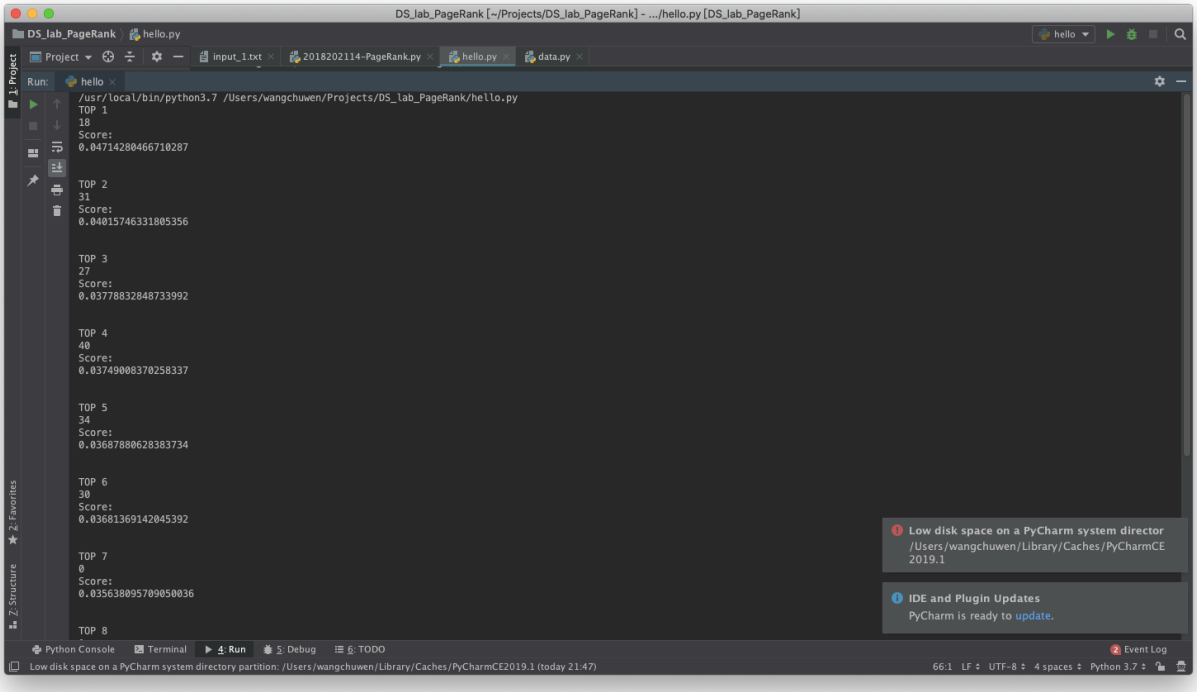
二.Personalized PageRank结果

showCase：点击此处可在bilibili观看Epinion演示视频2

[运行截图]



[结果截图]:



[top10]

TOP 10	SCORE	
18	0.0471428046671029	total iteration is 49
31	0.0401574633180536	
27	0.0377883284873399	
40	0.0374900837025834	
34	0.0368788062838373	
30	0.0368136914204539	total time is 11.296616s
0	0.0356380957090500	
1	0.0339927888780729	
12	0.0335797110693822	
28	0.0322095716245708	

应对大规模图数据的设想和实现

应对大规模图数据，除了使用邻接列表代替链接矩阵外，还可采用Map-Reduce法[实现][replica]

Mapper 的输入格式为：
（节点，*PageRank* 值）->（该节点的外部链接节点列表）
Mapper 的输出格式为：
（节点） ->（该节点的反向链接节点，反向节点的 *PankRank* 值/反向节点的外链个数）
Reducer 的输入格式（Mapper 的计算输出）为：
（节点） ->（该节点的反向链接节点，反向节点的 *PankRank* 值/反向节点的外链个数）
Reducer 的输出为：
（节点， 新的 *PageRank* 值）
假设有物理节点 *A*，*B* 参与计算，其中网页*1*、*2*保存于 *A*，网页*3*、*4*保存于 *B*，各个网页的 *PageRank* 初始值是*1.0*，*alpha* 值取 *0.85*，那么运算过程如下：
节点 A 上做 Mapper：
输入为：
（*1*，*1.0*）->（*2*，*3*，*4*）
（*2*，*1.0*）->（*3*，*4*）
输出为：
（*2*）->（*1*，*1.0/3*）
（*3*）->（*1*，*1.0/3*）
（*4*）->（*1*，*1.0/3*）
（*3*）->（*2*，*1.0/2*）
（*4*）->（*2*，*1.0/2*）
节点 B 上做 Mapper：
输入为：
（*3*，*1.0*）->（*4*）
（*4*，*1.0*）->（*2*）
输出为：
（*4*）->（*3*，*1.0*）
（*2*）->（*4*，*1.0*）
Reducer:
输入为 节点 A 和 B 的输出。
计算新的 PageRank 结果如下：
（*1*）： *0* +（*1-0.85*）*/4*
（*2*）： *0.85**（*1.0/3+1.0*）+（*1-0.85*）*/4*
（*3*）： *0.85**（*1.0/3+1.0/2*）+（*1-0.85*）*/4*
（*4*）： *0.85**（*1.0/3+1.0/2+1.0*）+（*1-0.85*）*/4*
多次迭代得出稳定的 PageRank 值。

Personalized PageRank线性可加证明

Proof：

由于平凡图$G=<V,E>$全连通，故知向量唯一收敛于p^*，且p^*满足等式：

$$p^* = \alpha p^* \mathrm{L} + (1-\alpha)p^{(0)}$$

对于$i = 1,2,3...n$，$p_i^* \leftarrow PPR(p_i^{(0)}, G, \alpha)$有：

$$\begin{cases} p_i^* = \alpha p_i^* \mathrm{L} + (1-\alpha)p_i^{(0)} \\ p^* = \alpha p^* \mathrm{L} + (1-\alpha)p^{(0)} \end{cases}$$

即

$$\begin{cases} (1-\alpha)p_i^{(0)} = p_i^* - \alpha p_i^* \mathrm{L} \\ (1-\alpha)p^{(0)} = p^* - \alpha p^* \mathrm{L} \end{cases} \quad (i = 1,2,3...n) \quad (\#)$$

由于$p^{(0)} = [\lambda_1, \lambda_2 \ldots \lambda_n]$，故$p^{(0)} = \sum_{i=1}^{n} \lambda_i p_i^{(0)}$

故：

$$(1-\alpha)p^{(0)} = \sum_{i=1}^{n} (1-\alpha)\lambda_i p_i^{(0)}$$

将(#)式带入得：

$$\begin{aligned} (1-\alpha)p^{(0)} &= \sum_{i=1}^{n} \lambda_i (p_i^* - \alpha p_i^* \mathrm{L}) \\ &= \sum_{i=1}^{n} \lambda_i p_i^* - \alpha (\sum_{i=1}^{n} \lambda_i p_i^*)\mathrm{L} \end{aligned}$$

故可知当$p^{(0)} = \sum_{i=1}^{n} \lambda_i p_i^*$时，只需一步可收敛$\sum_{i=1}^{n} \lambda_i p_i^* \leftarrow PPR(p^{(0)}, G, \alpha)$

又由于$p^* \leftarrow PPR(p^{(0)}, G, \alpha)$

故又Markov链收敛唯一性可得：

$$p^* = \sum_{i=1}^{n} \lambda_i p_i^*$$

证毕

实验小结

这次实验让我复习了一下PageRank的工作原理，同时也学会了在不直接调用库函数的情况下Python编程实现PageRank。

在实验中，出现最多的错误是TypeError，经常没注意类型就直接进行运算了，后来根据提示信息也逐一进行了修改。

这次实验让我从浅入深，一开始我设计的程序相当于是把pagerank的数学计算方法直接用python实现，依赖于矩阵的计算。原始程序对于给的例子g2运行效果还不错，但一旦运行稀疏图，矩阵的计算就要花费很长时间，这让我在Epinions数据集上测试的时候不得不换一种方法。用邻接列表来代替邻接矩阵，把每次迭代会影响到某顶点的其他顶点放在list里，对这个顶点的pagerank迭代，只需要依赖于list里的顶点就好了，于是计算时间大大减少，从原来的460s，缩减到了6s。

除此之外，实验中还做了一道算法证明题，这也是我的第一次尝试，感觉良好。

总而言之，我不仅掌握了pagerank的算法实现，还了解了算法落实到具体程序上的变化。这次实验让我受益匪浅。

附

提交的代码由于采用的是更直观的矩阵计算，故其只在小规模图的计算上比较好。如果测试需要在大规模稀疏图上进行的话，我的大规模稀疏图的pagerank算法实现以提交至github，见开头超链接，文件名为hello.py.也可参考我上传至bilibili的视频。