PageRank 实验报告

# 数据集分析

共有节点 8297 个， 矩阵有8274列， 8297行， 要采取方法把矩阵归一化为方阵。

共有记录103689条。矩阵稀疏度0.0015

# 算法分析

## 稀疏矩阵构建

由于网络矩阵的稀疏性，构建稀疏矩阵以减少内存使用，加速计算。

基于不要重复造轮子的考虑，我们直接采用了Python科学计算库scipy中的sparse类，采取了基于三元组保存的稀疏矩阵

矩阵构建和Page向量构建

1. **class** Page(object):
2. **def** \_\_init\_\_(self, Row, Col):
3. data = np.ones(len(Col))
4. self.matrix = csr\_matrix((data, (Row, Col)))
5. self.page = np.ones(self.matrix.shape[0])
6. self.page = self.page/len(self.page)

矩阵正则化，每行和为1。

1. **def** regularize(self):
2. # max = np.max(self.matrix.shape)
3. # self.matrix.reshape((max, max))
4. s = self.matrix.nonzero()
5. Row = s[0]
6. Row = set(Row)
7. **for** i **in** Row:
8. row = self.matrix.getrow(i)
9. col = row.nonzero()[1]
10. mean = row.getnnz()
11. **for** j **in** col:
12. self.matrix[i, j] = 1 / mean

## 基于矩阵分块计算的PageRank算法

假设RAM足够存储新向量和旧向量，但是不足以用来做矩阵运算，就需要将矩阵存在磁盘中，将矩阵分块读进内存，分多次计算得到新向量。由于数据集并不是很大，稀疏矩阵可以在内存中完成运算，但是为了测试算法，将矩阵分为10块。在本地测试环境中，矩阵分块运算产生的时间损耗约为19%。

收敛判断

1. **def** issame(self, page, newpage):
2. d = self.differ(page, newpage)
3. **if** d < 0.000001:
4. **print**("convergence at "+str(d))
5. **return** True
6. **return** False

如果新向量和旧向量相似度低于（该参数作为相似度S，在下文中尝试了修改对效果的影响）则认为收敛。

相似度计算函数

1. **def** differ(self, oldrank, newrank):
2. dif = oldrank - newrank
3. dif = dif/oldrank
4. dif = dif\*\*2
5. **return** np.mean(dif)

矩阵分块存储函数

1. **def** save(self):
2. #parts=10
3. page = self.page
4. l=len(page)
5. partlen=int(l/parts)
6. Use\_mit = self.matrix.T
8. **for** i **in** range(parts-1):
9. begin=i\*partlen
10. end=begin+partlen
11. save\_npz('Use\_mit'+str(i), Use\_mit[begin:end])
13. begin=(parts-1)\*partlen
14. end=l;
15. save\_npz('Use\_mit'+str(parts-1), Use\_mit[begin:end])

PageRank算法

1. **def** rank(self, k, beta):
2. page = self.page
3. l=len(page)
4. partlen=int(l/parts)
5. **for** i **in** range(k):
6. #  针对dead ends和spider trap做出调整
8. **print**(str(i)+'/'+str(k))
9. new\_page = np.ones(l)
10. **for** j **in** range(parts-1):
11. begin=j\*partlen
12. end=begin+partlen
13. part=load\_npz('Use\_mit'+str(j)+'.npz')
14. temp=part.dot(page)
15. new\_page[begin:end] = temp \* beta + (1 - beta)\*np.ones(partlen) / l
16. #每次读取一块矩阵并计算new\_page中对应位置的值
17. begin=(parts-1)\*partlen
18. end=l
19. part=load\_npz('Use\_mit'+str(parts-1)+'.npz')
20. temp=part.dot(page)
21. #print(temp)
22. new\_page[begin:end] = temp \* beta + (1 - beta)\*np.ones(end-begin) / l
23. #new\_page = (Use\_mit.dot(page)) \* beta + (1 - beta)\*np.ones(len(page)) / len(page)
25. **if** i == k - 1:
26. d = self.differ(page, new\_page)
27. **print**("deffer = "+str(d))
28. page = new\_page
29. **return** page

## 针对Dead End 和 Spider Trap的调整

不考虑Dead End和 Spider Trap 时， 迭代公式为

考虑Dead End和 Spider Trap ，

Dead End 是一类只有入度没有出度的节点， 如果不对网络中的这类节点做处理， 网络中的能量将从 Dead End中流失，最终导致所有节点的能量都是零。

Spider Trap 是一类有入度，且出边只指向自己的节点，这类节点会将能量困在自己身上，导致最终自己成为高分节点。

我们使用“抽税”法一举解决以上两个问题，即，认为每一个冲浪者有很小的机会随机的跳转到任意节点，这样，迭代公式为

公式中

β为收敛阻尼，

e为单位向量，向量长度和v一致。

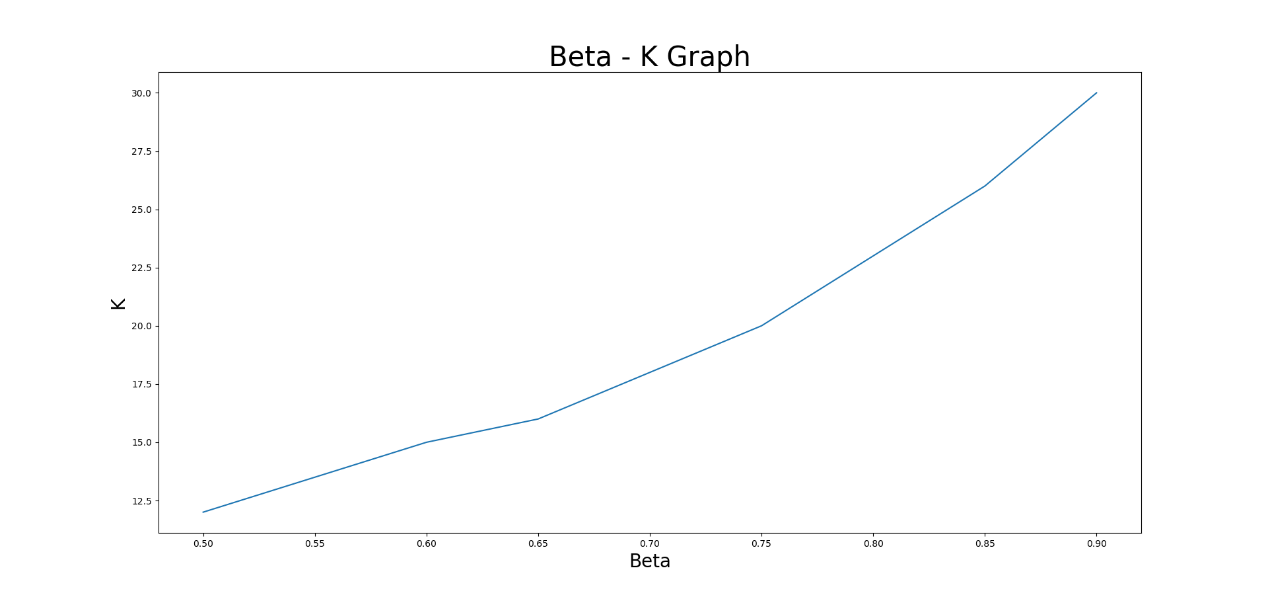
## 对β和K的参数尝试

对于迭代式中的和，定义相似度

固定相似度时认为收敛

对于收敛阻尼β， 尝试改变β，查看对K的影响。

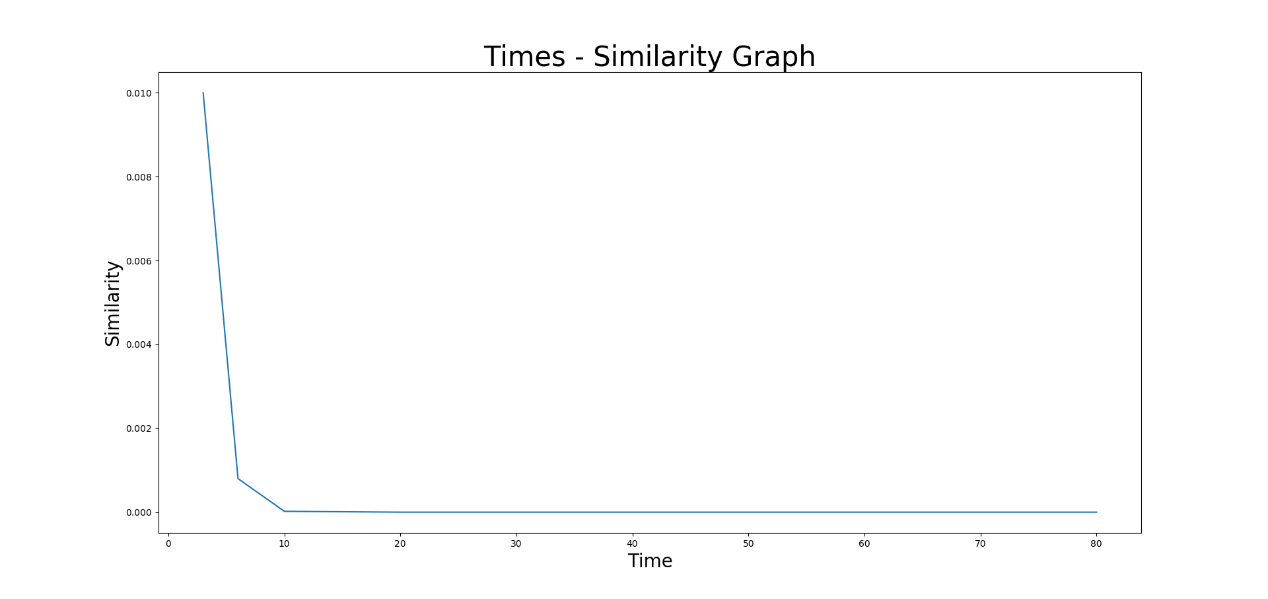
尝试使用不同β，查看收敛速度，结论是β越大收敛越慢



不同β， 收敛时得到的TopPage绝大部分相同，存在少数顺序不同的现象。

固定β=0.85，尝试不同k下，相似度S的变化，结论是收敛速度很快。

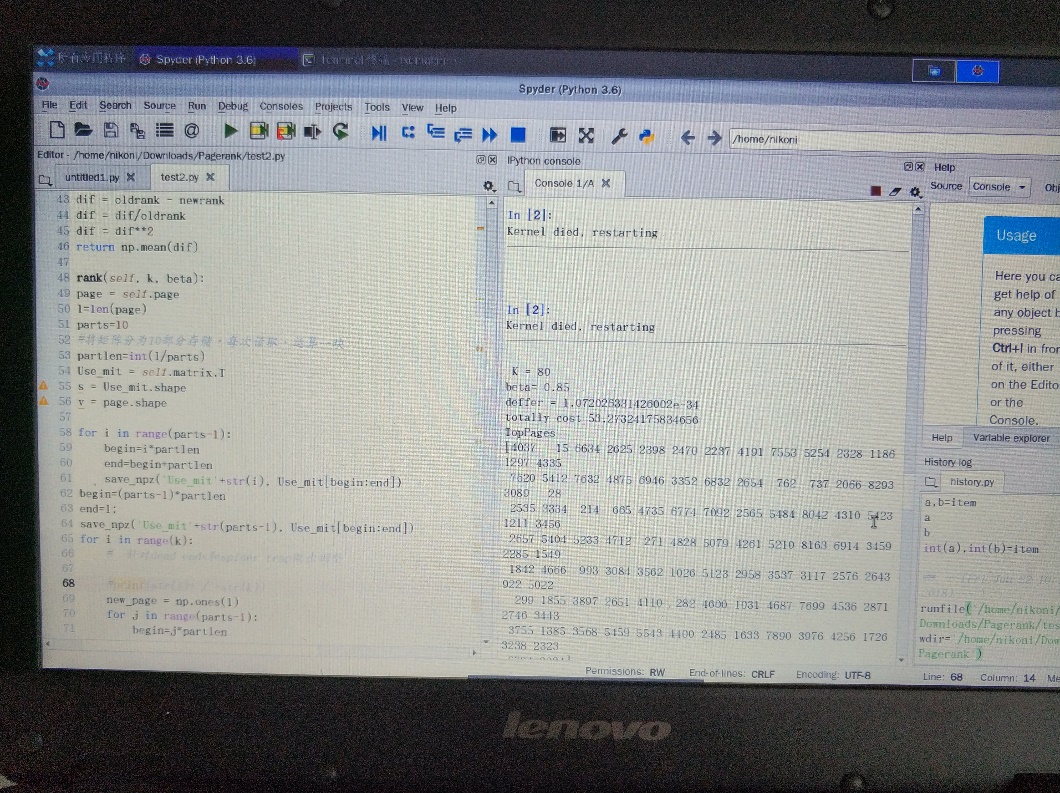
在k=100时完全收敛， 最终我们采用了k=100.



# Linux运行截图

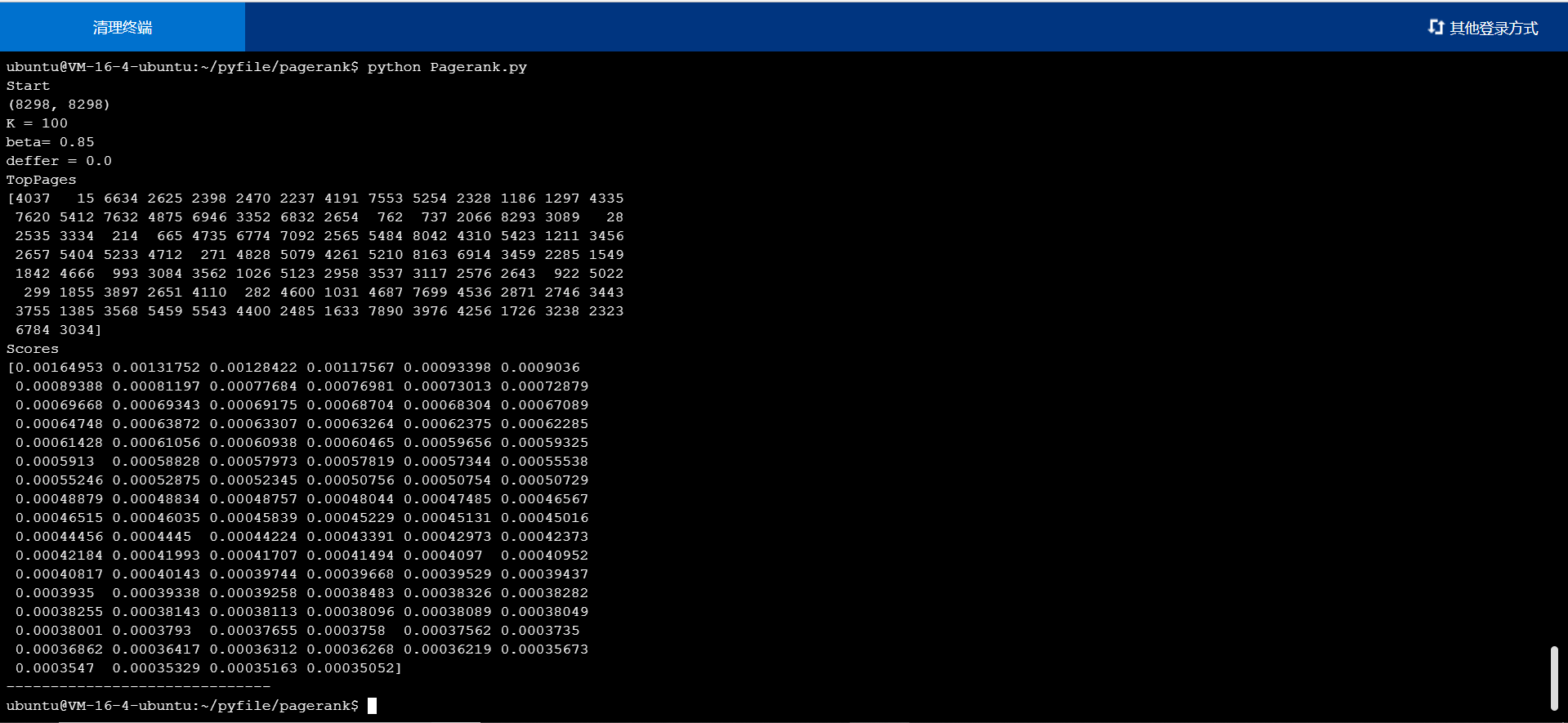
## 运行环境一

本地Chrombook下运行截图



## 运行环境二

腾讯云Ubuntu 16.04 64位运行截图



未分块

K = 100 ， β = 0.85

此时已经完全收敛



分块计算

参数同上

total cost显示的是运行时间

# Future Work

## 收敛条件选择

在代码中，differ 的大小可以用来判断向量是否收敛

取K=100， β=0.85

可以看见，最后differ= 0

已经完全收敛。

在之前“[*对β和K的参数尝试*](#_对β和K的参数尝试)” 一节中我们讨论过K对differ的影响，

K越大differ越小，但是计算代价也越大。一定存在一个工程上比较合适的differ，使得计算代价适中的情况下求出比较收敛的Toplist向量。但是这个问题似乎只是单纯的工程取舍，我们也没有查到一般取值。所以最后我们认为differ=0才收敛。这个问题我们希望以后能进一步探究。

## 分块矩阵相乘并行

在之前” [*基于矩阵分块计算的PageRank算法*](#_基于矩阵分块计算的PageRank算法)”一节中我们将矩阵进行分割。在CPU多核，或者拥有多台服务器的情况下，矩阵分块计算是可以而且易于并行的。实际上这也是PageRank算法能应用在实战中的一大原因。

但是考虑到本次实验的服务器是单核。数据量也不大。我们并没有做并行优化。这个功能我们希望以后在多核或多服务器下尝试。