PageRank 算法实现与改进

1 算法背景

Pagerank算法，即网页排名，又或Google排名，是Google创始人Sergey Brin和Lawrence Page在1997年构建早期的搜索系统原型提出的链式分析算法。根据网页被引用的次数以及引用该网页的网页的重要性来决定其重要性。

2 数据分析

在`WikiData.txt`文档中，存储了连接的数据，数据有两列。表示了从 FromNodeID 到 ToNodeID 的连接。

3 实验原理

pagerank的核心算法基于以下两种假设

1. 数量假设：如果有很多页面指向同一页面，那么这一页面就很重要。
2. 质量假设：如果一高质量页面指向某一页面，则该页面也可能很重要。

我们定义每一个页面的排名与指向该页面的数量和质量有关，但是每个页面都可能指向多个页面，就需要考虑该页面的出度。对于任一个页面j，可以求得它的排名

$$r_j$$

，其中

$$d_i$$

是页面i的出度

$$r_j = \sum_{i \to j} \frac{r_i}{d_i}$$

化为矩阵运算可以得到

$$\begin{vmatrix} r_1 \\ r_2 \\ \vdots \\ r_n \end{vmatrix} = \begin{vmatrix} M_{11} & M_{12} & \cdots & M_{1n} \\ M_{21} & M_{22} & \cdots & M_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ M_{n1} & M_{n2} & \cdots & M_{nn} \end{vmatrix} * \begin{vmatrix} r_1 \\ r_2 \\ \vdots \\ r_n \end{vmatrix}$$

其中

$$M_{ij} = \begin{cases} \frac{i}{d_j} & if \quad (j \to i) \\ 0 & others \end{cases}$$

记为

$$R = MR$$

3.1 解决spider trap

为了解决spider traps，引入了一个阻尼系数d，它表示人们在当前网页继续浏览的概率，而(1-d)就是随机打开一个网页的概率，这样整个图就是强连通的[1]

$$r_j = d(\sum_{i \to j} \frac{r_i}{di}) + (1 - d)$$

3.2 解决dead ends

dead ends会导致迭代过程中R的和不断减少，收敛不到正确结果。dead ends反映在矩阵M上就是某一列全为0，这会导致R所有元素都为0。因此在每次迭代都要加上失去的Rank值。也可以在对矩阵M进行预处理，将全0的列全部换为1/n。

$$R_{new} = d(MR_{old}) + [1 - d * sum]_{n*1}$$

这里的sum是MR_{old}各元素的求和。这样就保证了在迭代过程中R各元素之和始终为1。

3.3 优化稀疏矩阵

如果按照常规方式存储数据，大小为8297*8297，为了优化，我们选择按照原始数据那样存储，即from-to的形式

3.4 实现分块计算

由于在矩阵计算过程中，实际上，部分计算是重复的，因此，利用分块矩阵去修改它的值，会减少计算量，能够加快计算的效率。对于从同一点出发的边统一处理，仅需从A中读取从某 一点出发的所有边即可，实现了分块矩阵

4 实验步骤

4.1 引入包

我们只引入了numpy包来做矩阵运算处理

```
import numpy as np
```

4.2 全局变量

f为数据的路径，`NUM`为点的数量，也就是最大的节点ID

```
f = 'WikiData.txt'
global NUM
NUM=0
```

4.3 加载数据

调用`np.loadtxt()`方法来读取文件，并把数据按照从小到大排序，以便分块处理，然后让节点ID从0开始。加载数据的时候的就是按照from-to的格式来存的

```
# 加载数据
def load_data(f):
    # f: 数据路径
    readtxt = np.loadtxt(open(f), dtype=int)
    edges = readtxt[:, :]
    # 从小到大排序，目的是让相同起点聚集在一起
    edges = sorted(edges, key=(lambda x: x[0]))
    global NUM
    NUM = np.max(edges)
    # 使下标与节点ID相同
    edges = list(map(lambda x: x - 1, edges))
    # 返回值为排序后的数据
    return edges
```

4.4 寻找deadend

`deadend`是没有出度的点，所以只需要遍历数据，看哪些点有就标记为1，没有就是0了。

```
# 寻找dead_end
def dead_end(data):
    # data为所有数据
    global NUM
    dead_flag = np.zeros(NUM)
    # 若数据的起点没有该节点，则为0
    for d in data:
        dead_flag[d[0]] = 1
    # 返回值为一个01数组，0为dead_end
    return dead_flag
```

4.5 核心方法

该方法就是PageRank核心方法，首先初始化v为1/NUM，然后不断迭代，在里面实现了分块矩阵，因为数据是由小到大排好的，因此从第一个from点开始知道找到所有的to，然后把它rank值分配给出度的点，当所有出度的点分配完后，就找到下一个from点继续遍历。当所有from点遍历完，就去找到deadend把这些deadend的值加回总池里面，然后再用阻尼系数解决掉spidertrap，最后进行一下判断，即收敛或者达到迭代最大次数

```
# 核心函数
def block_stripe_pagerank(A, r, w, beta):
    # A: 原始数据集，r为迭代次数，w为dead_end标签，beta:阻尼系数
    global NUM
```

```
    v_new = np.ones(NUM) * 1 / NUM  # 初始值
    v_old = v_new
    B = v_new
    rank = 1  # 迭代次数
    while 1:
        v_old = v_new  # 每次更新旧值
        x_old = A[0]  # 从第一个点开始
        start = end = 0  # 起点与终点
        v_new = np.zeros(NUM)
        for x in A:  # 遍历每条边
            # 对于从同一点出发的边统一处理,实现了分块矩阵
            if x[0] != x_old[0]:  # 遇到不同的起始点
                for i in range(start, end):  # 遍历相同起点的边
                    v_new[A[i][1]] += v_old[x_old[0]] / (end -
start)  # 更新不同终点的入度
                start = end
                x_old = x  # 到下一个起始点
                end += 1
            else:
                end += 1  # 相同起点继续遍历
        index = 0
        sum = 0
        # 对于dead end进行random teleporting（心灵转移），就是我们认为
在任何一个页面浏览的用户都有可能以一个极小的概率瞬间转移到另外一个随机页面
        for x in w:
            if (x == 0):  # 若是dead_end则把本来的值保留下来
                sum += v_old[index]
                index += 1
            else:
                index += 1
        # 加入到总的矩阵里面
        v_new += sum * np.ones(NUM) * 1 / NUM
        # 根据公式计算 beta为阻尼系数
        v_new = beta * v_new + (1 - beta) * B
        # 迭代次数加一
        rank += 1
        # 如果求的矩阵收敛或者迭代次数达到最大值则停止迭代
        if np.sum(np.abs(v_new - v_old)) < 1.0e-6 and rank < r:
            break;
    # 返回迭代次数，最后的矩阵
    return rank, v_new
```

4.6 回写数据

通过循环把数据写入到`β+result.txt`里面，间隔为`,`。

```
def write_data(node_id, score, beta):
    f = open(str(beta) + 'result.txt', 'w')
    for i in range(100):
        f.write(str(node_id[i] + 1) + ' ' +
str(score[node_id[i]]) + '\n')
    f.close()
    return
```

4.7 主函数

主函数主要调用上面的方法，然后获取结果从大到小的顺序，然后传入方法，把结果写出。

```
if __name__ == '__main__':
    data = load_data(f)
    dead_flag = dead_end(data)
    r = 100
    beta = input("please input β: ")
    beta = float(beta)
    rank, re = block_stripe_pagerank(data, r, dead_flag, beta)
    re = re / np.sum(re)
    index = np.argsort(-re)
    write_data(index, re, beta)
```

5 实验结果

当β=0.85时，前20项

```
4037 0.0043478912026883336
15 0.0034727700812874373
6634 0.0033849964474663282
2625 0.003098860861503943
2398 0.002461828927040283
2470 0.0023817378380743947
2237 0.002356121152125842
4191 0.002140222904878855
7553 0.002047623859348148
5254 0.0020290981293788132
2328 0.0019244954446397278
1186 0.0019209776541996072
1297 0.0018363341778702838
4335 0.0018277671129415729
7620 0.0018233479682458465
5412 0.0018109253156389235
7632 0.0018003804894792855
4875 0.0017683598661553692
6946 0.0017066514014120167
3352 0.001683559924398068
```

当β=0.75时，前20项

```
4037 0.004112821971018394
15 0.0031700595338187477
6634 0.002765411268361496
2625 0.002744284256757444
2470 0.002352040469726903
2237 0.002277468781436758
2398 0.0021377294285309916
4191 0.0019274295701037696
1186 0.001885109058889225
5254 0.0018456790728876174
7553 0.0018006292897053279
2328 0.0017388869358659926
7620 0.001638531426133147
1297 0.0016146759409541899
4875 0.0016074814945350882
4335 0.0015841585456706092
2654 0.0015749465819829263
8293 0.0015537241864138618
7632 0.0015379912118150655
665 0.00153042330087271
```

当β=0.9时，前20项为

```
4037 0.004439848671551332
6634 0.0037499802146872237
15 0.0036139204124339664
2625 0.003278344127204988
2398 0.002631654028119673
2237 0.002376384318996887
2470 0.002369305016743573
4191 0.002246316405358422
7553 0.0021687457835321473
5254 0.0021167733224489063
2328 0.002015654465023306
5412 0.001966152151807405
4335 0.0019563771228313133
1297 0.0019479959678167793
7632 0.00193895864549683
1186 0.0019202550338986084
7620 0.0019145173862178508
6946 0.0019017816870990085
4875 0.0018478438952323022
6832 0.0017817472925882754
```

当β=0.5时，前20项为

```
4037 0.0032261873367280783
15 0.002300204877439733
2470 0.0019836475523792763
2625 0.0018735458636152925
2237 0.0018653208676929836
6634 0.0016285480311152482
```

```
1186 0.0015883235218259219
2398 0.0013991524288094106
4191 0.0013788765159828292
5254 0.0013406246752178956
665 0.0012984138775478662
8293 0.001260239520998325
2328 0.00125040268705561
2654 0.0012412264096556002
6774 0.0012313566090085574
7553 0.0011794913083527964
4875 0.001179243735774769
214 0.00117745824045006 1
28 0.0011627256893478135
7620 0.0011584134227185711
```

---

1. Brin S, Page L. The anatomy of a large-scale hypertextual web search engine[J]. 1998. ↩