PageRank

实验报告

2020.06.22

数据科学导论

思子华

2018202181

目录

1 实验要求

1.1 作业内容

- 实现 PageRank
- 实现 Personalized PageRank
- 证明 Personalized PageRank 的线性可加性
- 考虑如何存储稀疏图，并对 PageRank 算法计算的部分进行优化

2 实现思路

2.1 PageRank

2.1.1 核心功能

- 读取文件
- 从图中提取出 dead ends
- 进行 PageRank
  - 初始化向量$P^{(0)}$
  - 迭代计算
    - $P^{(i+1)} = \frac{1\text{-}\alpha}{n} P^{(i)}E$
    - $P^{(i+1)} += \alpha PL$
  - 判断是否 converge
- 返回根据分数降序排序的 list

2.1.2 代码结构

```
                |--> read_file
                |--> deadEnds
PageRank ----
                |--> mypagerank (iterations)
                |--> sort (sorted by value)
```

2.1.3 代码实现思路

- 从输入文件中提取出临接链表（使用哈希表存储）、提取所有结点的名称（存储在哈希表中）
- 根据临接链表从所有结点中挑选出只有入度没有出度的 deadEnds
- 迭代计算 pagerank
  - 按1-α的比例计算$P^{(i+1)} = \frac{1-\alpha}{n} P^{(i)} E$
  - 考虑到 deadEnd 的存在，将 deadEnds 中的值均匀加到所有结点中
  - 按α的比例$P^{(i+1)} += \alpha PL$
- 计算误差是否达到 10e-6

为什么需要考虑 dead ends

课上的 ppt 中没有处理 dead end，但是作业的图中有 1w 多个结点是 dead ends。如果不考虑 dead ends，会导致向量 P 的各个维度之和从 1 不断减小，因为 dead ends 中值没有进一步传播出去。

根据实际测试，如果不考虑 dead ends 不会影响最终的迭代结果，但是会导致迭代次数增加。因为不考虑 dead ends 的情况下每一次迭代总 P 的各个维度的值都小于等于考虑 dead ends 的情况下 P 中各个维度的值，进而导致两次迭代之间的差距较小，因此迭代速度较慢。

如 Figure1 所示，图上方不考虑 dead ends 的时候迭代了 43 次，用时 6.4s，并且排序结果的得分较低；图下方考虑 dead ends 以后，迭代了 35 次，用时 5.56s，排序结果与上方相同，但是得分较高。(注：这里的图是 PPR 的运行结果)

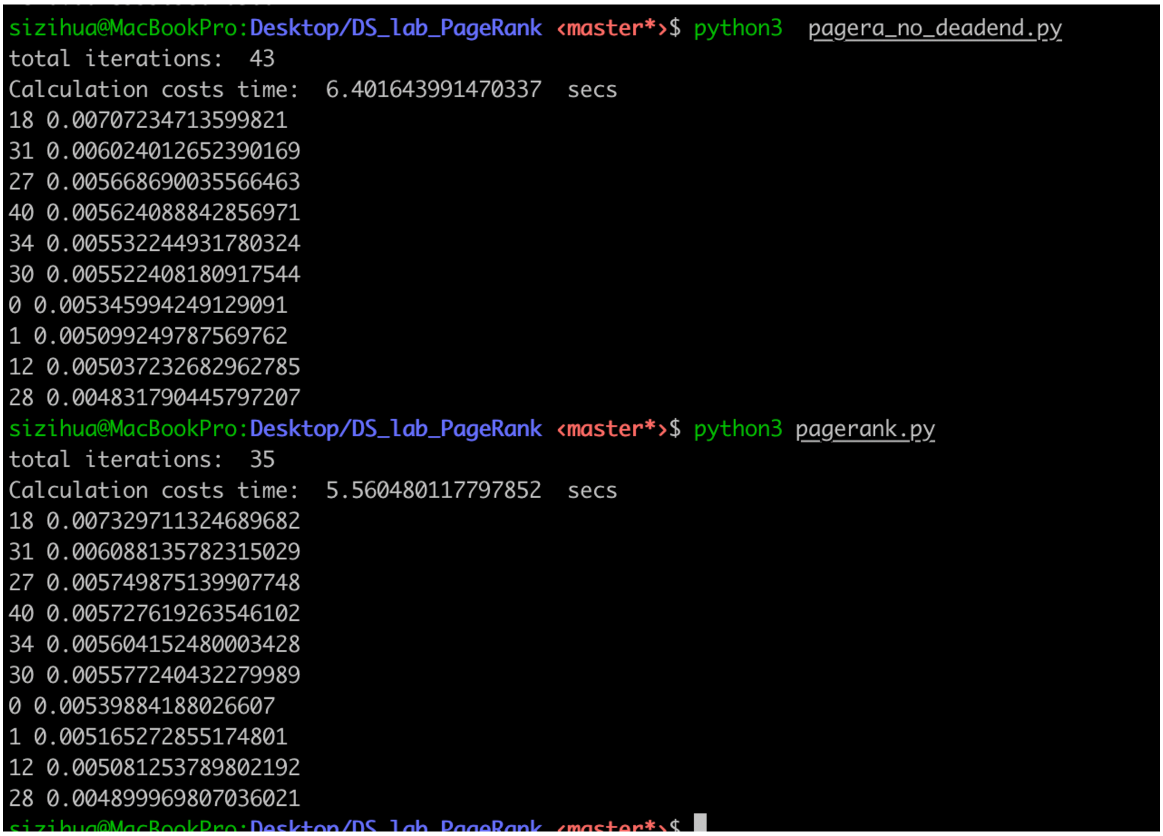


Figure 1

为什么使用哈希表存储临接链表

由于本次实验的数据为稀疏图（边数 508837 远小于结点数的平方75879^2），使用邻接链表存储图的数据。在迭代计算 P 与 L 的乘积时，从 pagerank 的实际含义出发，遍历临接链表，设遍历到结点u_i时，将$\frac{val\ (u_i)}{out_degree(u_i)}$添加到$v_j$中（其中，val 表示结点到值，out_degree 表示出度，存在边（u_i, v_j）。

程序中使用 python 自带的 dict 存储临接链表存储数据，由于 dict 是使用哈希表实现的，查询元素、修改 value 值的时间复杂度是 O(1)。因此，本程序运行速度较快。

2.1.4 核心代码展示

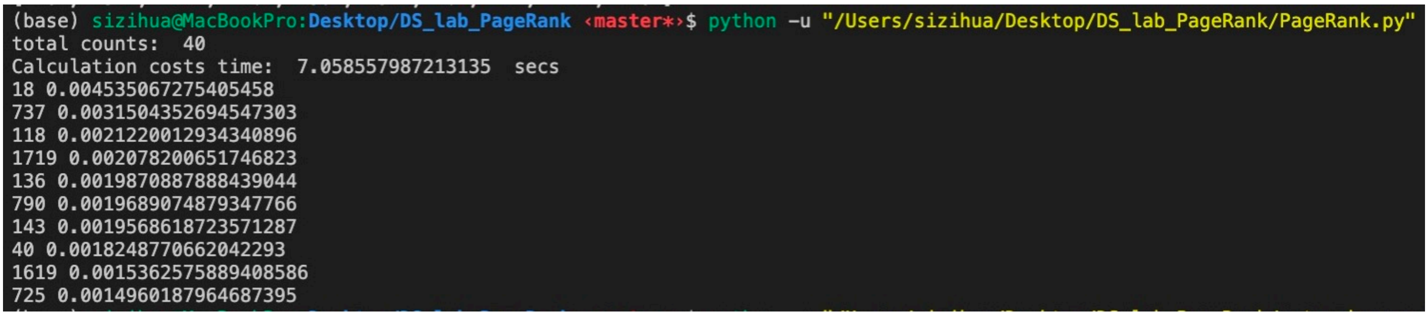
```python
def mypagerank(link_dict, pagelist, dead_end_list, beta=0.85):
    #迭代计算
    n_pages = len(pagelist)  # 页面总数

    r_old = dict()
    r_new = dict()
    for each_page in pagelist.keys():
        r_old[each_page] = 1/n_pages
    convergence = False
    count = 0
    while not convergence:
        dead_end_sum = 0.0
        for each_end in dead_end_list:
            dead_end_sum += beta * r_old[each_end] / n_pages

        # 初始化 r_new
        for each_page in pagelist.keys():
            r_new[each_page] = (1 - beta) / n_pages + dead_end_sum
        for src in link_dict.keys():
            dest_list = link_dict[src]
            src_out_degree = len(dest_list)
            for each_dest in dest_list:
                r_new[each_dest] += beta * r_old[src] / src_out_degree
        #判断是否 converge
        err = 0
        threshold = 10e-6
        for each_page in pagelist.keys():
            err += abs(r_old[each_page] - r_new[each_page])
        for each_page in pagelist.keys():
            r_old[each_page] = r_new[each_page]
        convergence = err < threshold
        count += 1
        #print('iteration: ', count, end='\r')
    #print('total iterations: ', count)
    return r_new
```

时间复杂度分析

- 初始化时间复杂度 O(V)
- 迭代循环中
- 计算 dead ends 中所有结点值的和需要 O(V)
- 计算$P^{(i+1)} = \frac{1-\alpha}{n} P^{(i)} E$ 并且加入 dead ends 中的值需要遍历一遍临接链表，时间复杂度 O(V+E)
- 按α的比例$P^{(i+1)} += \alpha PL$，时间复杂度为 O(V)
- 由于一共迭代常数次，总的时间复杂度为 O(V+E)

2.1.5 结果展示

- Page Rank 运行结果



```
(base) sizihua@MacBookPro:Desktop/DS_lab_PageRank ‹master*›$ python -u "/Users/sizihua/Desktop/DS_lab_PageRank/PageRank.py"
total counts:  40
Calculation costs time:  7.058557987213135  secs
18 0.004535067275405458
737 0.0031504352694547303
118 0.0021220012934340896
1719 0.002078200651746823
136 0.0019870887888439044
790 0.0019689074879347766
143 0.0019568618723571287
40 0.0018248770662042293
1619 0.0015362575889408586
725 0.0014960187964687395
```

Figure 2

2.1.6 如何应对大规模图数据的设想和实现

一般情况下的 PageRank 时间、空间开销

- 1）时间开销：算法主要时间开销将会是每轮迭代中做$P^{(i+1)} += \alpha PL$的矩阵乘法上，这是 $O(V^2)$ 的时间开销(V 代表图中结点个数，下同)，再乘上算法需要迭代 k 轮完成收敛，因此 PageRank 的时间开销是 $O(V^2)$. 不过，一般来说，这个收敛需要次数 k 会是在 10~100 之间的数值，不会特别大.
- 2) 空间开销：算法最大的空间开销来自于存储整个 P 矩阵到内存中，这是$O(V^2)$的空间开销. 因此如果假设有 10^6 个结点的图，需要的 M 矩阵大小是 10^12，按照 int 型 4byte 来存储，这相当于 4TB 的内存开销，这是无法承受的空间开销.

对矩阵空间优化的办法

- 已知大多数情况下，M 矩阵十分稀疏，那么我们可以使用邻接链表或者类似形式，只存储非零元素的值.
- 比如，在 Python 中，可以通过构造字典数据类型来实现.
  - G = {1: [2, 3, 4], 2: [1, 4], 3: [1], 4: [2, 3]}
    G[1] = [2, 3, 4]表示结点 1 和 2，3，4 是有一条有向边.

- 那么这种情形下，空间开销就是 O(V+E) 的，因为邻接链表存储了所有的点和有向边. 在稀疏图中，O(V+E) 往往远小于 O(V^2).

对时间的优化办法

- 为了实现从 O(V^2) 下降到 O(V+E) 的优化，我们需要重新定义一般 PageRank 中的 PL 矩阵乘法操作.
- 我们知道，PL 乘法实际上完成的目的是算出 P*列向量，也就是每个结点新的 PageRank 值. 按照之前所述，我们有如下观察：
- Observation ：结点的新 PageRank 值 = Σ （来源结点的 PageRank 值 • 本结点所分享到的权重）

```
for src in link_dict.keys():
            dest_list = link_dict[src]
            src_out_degree = len(dest_list)
            for each_dest in dest_list:
                r_new[each_dest] += beta * r_old[src] / src_out_degree
```

- 由此，我们的时间开销变成了 O(V + E)，在稀疏图中，这样的时间开销远比 O(V^2) 小

设想

本次实验的数据规模并没有特别的大，如果真的遇到特别大的数据规模，可以考虑使用并行、分布式等方式进行计算，并且可以使用特定的方式存储稀疏矩阵（比如三元组存储，十字链表存储法）

2.2 Personalized Page Rank

2.2.1 核心功能

- 与 PageRank 基本一致
- 核心代码只有一处不同

$P = \alpha PL + \frac{1-\alpha}{n}PE$ 变为 $P = \alpha PL + (1-\alpha)\mathrm{P}^{(0)}$

那一部分代码更改如下

```
for item in input_seed.keys():
        r_init[item] = (1 - beta) * input_seed[item]
```

2.2.2 运行结果

- PPR

```
sizihua@MacBookPro:Desktop/DS_lab_PageRank <master*>$ python3 pagerank.py
total iterations:  35
Calculation costs time:  5.560480117797852  secs
18 0.007329711324689682
31 0.006088135782315029
27 0.005749875139907748
40 0.005727619263546102
34 0.005604152480003428
30 0.005577240432279989
0 0.005398841880266607
1 0.005165272855174801
12 0.005081253789802192
28 0.004899969807036021
```

Figure 3

3 PPR 线性可加证明题

PPR 公式如下：

$$p = \alpha pL + (1-\alpha)p^0$$

不妨设 $L = \begin{bmatrix} l_1 \\ l_2 \\ \vdots \\ \vdots \\ l_n \end{bmatrix}$, 其中，$l_i = [l_{i1} \quad l_{i2} \quad l_{i3} \quad l_{i4} \quad l_{i5}]$。

下面用**数学归纳法**证明：

$$p^k = \sum_i^n \lambda_i p_i^k \text{ (其中 k} \in [1 \quad \cdots \quad *])$$

设 $p^m = [\lambda_1^m \quad \lambda_2^m \quad \cdots \quad \cdots \quad \lambda_n^m], p_i^m = [a_{i1} \quad a_{i2} \quad \cdots \quad \cdots \quad a_{in}]$。

显然，m = 1 时，满足 $p^1 = \sum_i^n \lambda_i p_i^1$。

归纳假设：若 m = k 时，有$p^k = \sum_i^n \lambda_i p_i^k$成立。

则有：

$$[\lambda_1^k \quad \lambda_2^k \quad \cdots \quad \cdots \quad \lambda_n^k] = \sum_i^n [\lambda_i a_{i1} \quad \lambda_i a_{i2} \quad \cdots \quad \cdots \quad \lambda_i a_{in}] \tag{1}$$

那么 m = k + 1 时：

$$p^{k+1} = \sum_i^n \alpha \lambda_i^k l_i + (1-\alpha)p^0$$

$$\sum_i^n \lambda_i p_i^{k+1} = \sum_i^n \left(\sum_j^n \alpha \lambda_i a_{ij} l_j + (1-\alpha)\lambda_i p_i^0\right)$$

上式 $= \sum_i^n \sum_j^n \alpha \lambda_i a_{ij} l_j + \sum_j^n (1-\alpha)\lambda_i p_i^0$

$= \sum_j^n \sum_i^n \alpha \lambda_i a_{ij} l_j + \sum_i^n (1-\alpha)\lambda_i p_i^0$

$= \sum_j^n \sum_i^n \alpha \lambda_i a_{ij} l_j + (1-\alpha)\lambda_i p^0$

根据（1）式，

上式 $= \sum_i^n \alpha \lambda_i^k l_i + (1-\alpha)p^0$

即$p^{k+1} = \sum_i^n \lambda_i p_i^{k+1}$成立，由数学归纳法可知$p^k = \sum_i^n \lambda_i p_i^k$得证。

综上所述，$p^* = \sum_i^n \lambda_i p_i^*$成立

4 实验小结

本次实验，我认为有三个 trick 的点

- 使用 python 自带的 dict（哈希表）存储临接链表，使得随机访问元素和修改 value 的代价都是 O(1)
- 迭代计算矩阵乘法时，结合 page rank 的原理进行了优化，使得矩阵乘法的时间复杂度降为了 O(E+V)
- 计算时考虑到 dead ends 的影响，减少了迭代次数，增加了运行速度