PageRank 实验内容与要求

第一个部分：从概率转移矩阵计算得到 pagerank 值

一、 实验目的

- 熟悉、理解和掌握 PageRank 算法结构，能够利用编程实现概率转移矩阵计算得到每个节点对应的 rank 值，并在此基础上理解采集器陷阱；
- 熟悉、理解和掌握避免采集器陷阱的“抽税”算法，并能在已经实现的 PageRank 程序上进行相应的修改得到相应的“抽税”算法程序；
- 更进一步，理解 PageRank 算法的收敛性分析（此为补充内容）。

二、 实验工具和环境

最好使用 Python 3 版本，利用其他程序语言实现也可以。需要用到的 python 包为 numpy

三、 算法过程

- 输入：概率转移矩阵（M），最大迭代次数（max_iterations），收敛阈值（min_epsilon）
- 初始化 page_rank 值（可以使用向量储存，）：page_rank[i]=1/n 其中 n 是图中节点个数
- 当迭代次数小于最大迭代次数，且前后所有 page_rank 值差的绝对值的和小于阈值，则：新的 page_rank 值计算结果为前一个 page_rank 向量左乘概率转移矩阵

  否则，跳出循环
- 输出每一个节点对应的 page_rank 值。

四、 实验步骤（也可以不完全按照此步骤执行，理解上面算法过程并将其实现即可）

1. 创建一个 python 项目文件，并添加一个 python 格式的文件
2. 该程序用到 numpy 包，在文件头输入“import numpy”
3. 定义一个 pagerank 迭代器（可命名为 PRIterator）的类，并在类中定义变量最大迭代次数，收敛阈值以及概率转移矩阵，并将前两个设置默认值为 100,0.00001

注：在这里也可以不用类的方法，直接用函数来执行也是可以的，这个不作限制。

4. 在类中只需要定义一个 page_rank 的方法即可，方法主要实现算法的第二步和第三步，初始化代码如下：

```
graph_size = numpy.size(self.matrix, 0)  # 得到图节点个数，即转移矩阵的行数
page_rank = numpy.ones([graph_size, 1])/graph_size
```

循环过程代码如下：

```
flag = False
for i in range(self.max_iterations):
    old_page_rank = page_rank
    page_rank = numpy.dot(self.matrix, page_rank)  # 左乘转移矩阵更新 PageRank 值
    change = numpy.abs(page_rank - old_page_rank)  # 计算迭代前后值的变化

    if numpy.sum(change) < self.min_epsilon:
        flag = True
        break
```

5. 利用 print 函数输出结果
6. 输入课本例 5.1、例 5.3、例 5.4 或者给你们的 data.txt 文件中的概率转移矩阵，并调用类中的方法，看输出结果

第二个部分：采用“抽税”算法

五、 “抽税”算法过程（划线部分为与之前算法不同部分）：

- 输入：概率转移矩阵（M），最大迭代次数（max_iterations），收敛阈值（min_epsilon），beta 常数
- 初始化 page_rank 值（可以使用向量储存）：page_rank[i]=1/n 其中 n 是图中节点个数
- 当迭代次数小于最大迭代次数，且前后所有 page_rank 值差的绝对值的和小于阈值，则：page_rank = beta*M*pagerank + （1-beta）*e/n
  否则，跳出循环
- 输出每一个节点对应的 page_rank 值。

六、 实验步骤（主要提及需要修改的地方）：

1. 在类中需要加入一个 beta 变量
2. 初始化时候，添加一个 e 向量，代码如下

```
e = numpy.ones([graph_size, 1])
```

3. 循环语句中需要修改迭代式，代码如下：

```
page_rank = self.beta * numpy.dot(self.matrix, page_rank) + (1-self.beta) * e/graph_size
```

4. 重新输入例 5.1、例 5.3 和例 5.4 中的概率转移矩阵，看输出结果

第三个部分（以下都是补充内容，不作为硬性要求）：对于大规模数据的处理和算法的收敛性分析

大规模数据的处理：

这里推荐一个 Stanford 的网站，http://snap.stanford.edu/data/。这是 Stanford 大学的一个数据集站点，在里面可以找到各种网络图数据。我在这里随便下载了一个：

Web graphs

Name	Type	Nodes	Edges	Description
web-BerkStan	Directed	685,230	7,600,595	Web graph of Berkeley and Stanford
web-Google	Directed	875,713	5,105,039	Web graph from Google
web-NotreDame	Directed	325,729	1,497,134	Web graph of Notre Dame
web-Stanford	Directed	281,903	2,312,497	Web graph of Stanford.edu

即上图的 web-BerkStan。打开这个数据集可以看到其文件头部对该数据集有一个说明，如下所示：

```
# Directed graph (each unordered pair of nodes is saved once): web-BerkStan.txt
# Berkely-Stanford web graph from 2002
# Nodes: 685230 Edges: 7600595
# FromNodeId    ToNodeId
1       2
1       5
1       7
1       8
1       9
1       11
1       17
1       254913
1       438238
254913  255378
254913  255379
254913  255383
254913  255384
254913  255392
254913  255393
254913  255394
254913  255396
254913  255399
254913  255401
254913  255402
254913  255561
254913  255562
254913  255637
254913  255638
```

这个数据集的表示方式其实和我们所需要的概率转移矩阵是有差的，因此在将该

数据集用在我们自己的程序上跑需要先对数据集进行处理，将其转换成我们想要的矩阵形式。

但是这个数据集如果转换为一个六十多万维且稀疏的概率矩阵的话，且直接按照左乘矩阵形式来迭代，对电脑性能要求实在太高了，也不是一个明智的选择。所以一般这种操作用到分布式处理方式，这里可以参考书本后面的内容。

重点推荐： SNAP http://snap.stanford.edu，全称 Standford Network Analysis Project，是斯坦福大学提供的一个功能非常强大的开源工具。这个工具主要用于复杂网络领域的研究工作，是一个集高质量论文、数据集、源码于一体的网站，资源数量不多，但文章质量非常的高。

SNAP 本身用于大规模数据与复杂网络分析，由 C++写成，性能高，能轻松处理成百上千、甚至十亿规模的节点。目前支持两种语言：C++与 Python，Snap.py 就是为 python 提供的一个开源接口。在下载使用其中的 Snap.py 时候注意一下版本对应就可以了。

算法收敛性分析：

此部分为补充内容，主要是如何证明算法一的收敛性：

- **Claim:** Sequence $\boldsymbol{M}\cdot\boldsymbol{r}^{(0)}, \boldsymbol{M}^{2}\cdot\boldsymbol{r}^{(0)}, \ldots \boldsymbol{M}^{k}\cdot\boldsymbol{r}^{(0)}, \ldots$ approaches the dominant eigenvector of $\boldsymbol{M}$
- **Proof:**
  - Assume $\boldsymbol{M}$ has $\boldsymbol{n}$ linearly independent eigenvectors, $x_1, x_2, \ldots, x_n$ with corresponding eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$, where $\lambda_1 > \lambda_2 > \cdots > \lambda_n$
  - Vectors $x_1, x_2, \ldots, x_n$ form a basis and thus we can write: $r^{(0)} = c_1 x_1 + c_2 x_2 + \cdots + c_n x_n$
  - $$\boldsymbol{Mr^{(0)}} = \boldsymbol{M}(\boldsymbol{c_1 x_1} + \boldsymbol{c_2 x_2} + \cdots + \boldsymbol{c_n x_n}) = c_1(Mx_1) + c_2(Mx_2) + \cdots + c_n(Mx_n) = c_1(\lambda_1 x_1) + c_2(\lambda_2 x_2) + \cdots + c_n(\lambda_n x_n)$$
  - **Repeated multiplication on both sides produces** $M^k r^{(0)} = c_1(\lambda_1^k x_1) + c_2(\lambda_2^k x_2) + \cdots + c_n(\lambda_n^k x_n)$

- $M^k r^{(0)} = \lambda_1^k \left[c_1 x_1 + c_2 \left(\frac{\lambda_2}{\lambda_1}\right)^k x_2 + \cdots + c_n \left(\frac{\lambda_2}{\lambda_1}\right)^k x_n \right]$
- Since $\lambda_1 > \lambda_2$ then fractions $\frac{\lambda_2}{\lambda_1}, \frac{\lambda_3}{\lambda_1} \ldots < 1$
  and so $\left(\frac{\lambda_i}{\lambda_1}\right)^k = 0$ as $k \to \infty$ (for all $i = 2 \ldots n$).
- **Thus: $M^k r^{(0)} \approx c_1 (\lambda_1^k x_1)$**
  - Note if $c_1 = 0$ then the method won't converge