中国人民大学信息学院

大数据实践 实验报告

实验名称 算法原理及 MapReduce 实现

实验内容 PageRank / K-means

专　　业 计算机应用技术

学　　号 2017100955

姓　　名 陶友贤

指导老师 覃雄派

2018 年 3 月 19 日

Github: https://github.com/taoyouxian/py-voucher/bigdata

目录

1 前言

PageRank 的 Page 可是认为是网页，表示网页排名，PageRank 算法计算每一个网页的 PageRank 值，然后根据这个值的大小对网页的重要性进行排序。它的思想是模拟一个悠闲的上网者，上网者首先随机选择一个网页打开，然后在这个网页上呆了几分钟后，跳转到该网页所指向的链接，这样无所事事、漫无目的地在网页上跳来跳去，PageRank 就是估计这个悠闲的上网者分布在各个网页上的概率。

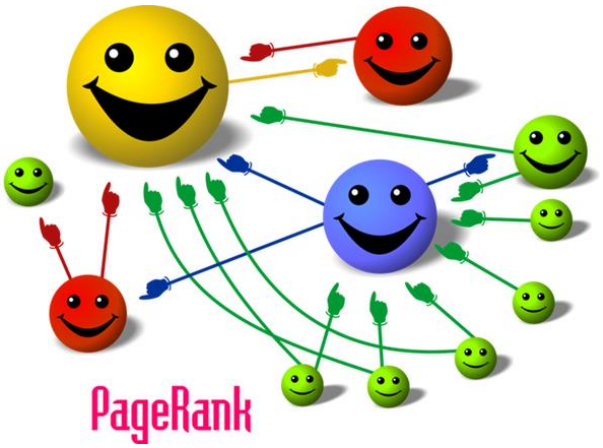
2 算法原理

2.1 核心思想

借鉴了学术界评判学术论文重要性的通用方法，那就是看论文的引用次数。由此想到网页的重要性也可以根据这种方法来评价，其核心思想如下

- 如果一个网页被很多其他网页链接到的话说明这个网页比较重要，也就是 PageRank 值会相对较高
- 如果一个 PageRank 值很高的网页链接到一个其他的网页，那么被链接到的网页的 PageRank 值会相应地因此而提高

就如下图所示（一个概念图）



图一 PageRank 核心思想概念图

2.2 算法原理

PageRank 算法总的来说就是预先给每个网页一个 PR 值（下面用 PR 值指代 PageRank 值），由于 PR 值物理意义上为一个网页被访问概率，所以一般是 1/N，其中 N 为网页总数。另外，一般情况下，所有网页的 PR 值的总和为 1。如果不为 1 的话也不是不行，最后算出来的不同网页之间 PR 值的大小关系仍然是正确的，只是不能直接地反映概率了。

预先给定 PR 值后，通过下面的算法不断迭代，直至达到平稳分布为止。

为了将这一分析数学化，我们 pi(n)表示虚拟用户在进行第 n 次浏览时访问网页 Wi 的几率。显然，上述假设可以表述为：

$$pi(n+1) = \Sigma j\, pj(n)pj \to i/Nj$$

这里 $p_{j\to i}$ 是一个描述互联网链接结构的指标函数，其定义是：如果网页 W_j 有链接指向网页 W_i，则 $p_{j\to i}$ 取值为 1，反之则为 0。显然，这条假设所体现的正是前面提到的佩奇和布林的排序原则，因为右端求和式的存在表明与 W_i 有链接的所有网页 W_j 都对 W_i 的排名有贡献，而求和式中的每一项都正比于 p_j，则表明来自那些网页的贡献与它们的自身排序有关，自身排序越靠前(即 p_j 越大)，贡献就越大。

为符号简洁起见，我们将虚拟用户第 n 次浏览时访问各网页的几率合并为一个列向量 $\mathbf{p}_n$，它的第 i 个分量为 $p_i(n)$，并引进一个只与互联网结构有关的矩阵 **H**，它的第 i 行 j 列的矩阵元为 $H_{ij}=p_{j\to i}/N_j$，则上述公式可以改写为：

$$\mathbf{p}n+1=\mathbf{H}\mathbf{p}n$$

注：更详细的算法原理介绍可以参考相关文献。

3 算法实现

3.1 迭代法

详见 github 地址。

3.2 MapReduce 实现

具体流程图：

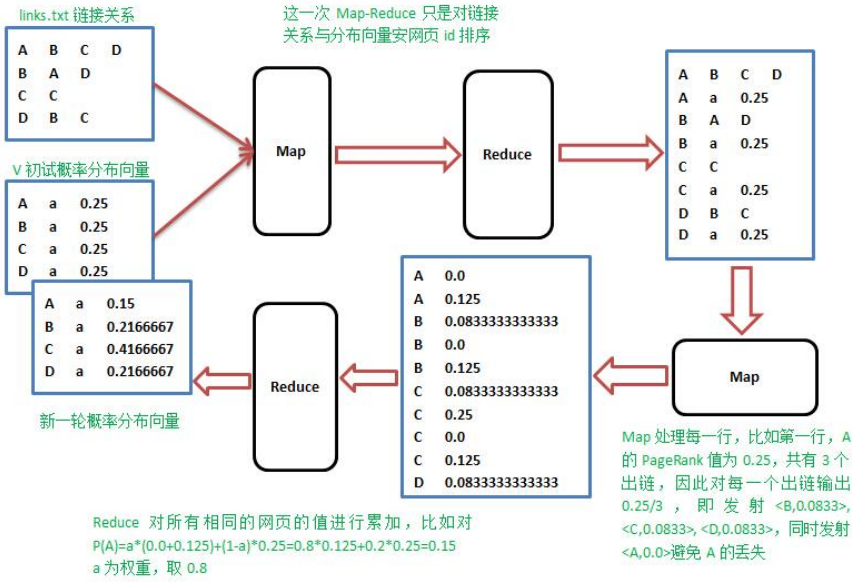


图 2 MapReduce 实现流程图

下面是使用 MapReduce 实现 PageRank 的具体代码。首先是通用的 map 与 reduce 模块。

```
class MapReduce:
    __doc__ = '''提供 map_reduce 功能'''

    @staticmethod
    def map_reduce(i, mapper, reducer):
        """
        map_reduce 方法
        :param i: 需要 MapReduce 的集合
        :param mapper: 自定义 mapper 方法
        :param reducer: 自定义 reducer 方法
        :return: 以自定义 reducer 方法的返回值为元素的一个列表
        """
        intermediate = []  # 存放所有的(intermediate_key,
intermediate_value)
        for (key, value) in i.items():
            intermediate.extend(mapper(key, value))

        # sorted 返回一个排序好的 list，因为 list 中的元素是一个个的 tuple，key
设定按照 tuple 中第几个元素排序
        # groupby 把迭代器中相邻的重复元素挑出来放在一起,key 设定按照 tuple 中第
几个元素为关键字来挑选重复元素
```

```
        # 下面的循环中groupby返回的key是intermediate_key，而group是个list，是1个或多个
        # 有着相同intermediate_key的(intermediate_key, intermediate_value)
        groups = {}
        for key, group in itertools.groupby(sorted(intermediate, key=lambda im: im[0]), key=lambda x: x[0]):
            groups[key] = [y for x, y in group]
        # groups是一个字典，其key为上面说到的intermediate_key，value为所有对应intermediate_key的intermediate_value
        # 组成的一个列表
        return [reducer(intermediate_key, groups[intermediate_key]) for intermediate_key in groups]
```

接着是计算 PR 值的类，其中实现了用于计算 PR 值的 mapper 和 reducer：

```
class PRMapReduce:
    __doc__ = '''计算PR值'''

    def __init__(self, dg):
        self.damping_factor = 0.85  # 阻尼系数,即α
        self.max_iterations = 100  # 最大迭代次数
        self.min_delta = 0.00001  # 确定迭代是否结束的参数,即ϵ
        self.num_of_pages = len(dg.nodes())  # 总网页数

        # graph表示整个网络图。是字典类型。
        # graph[i][0]存放第i网页的PR值
        # graph[i][1]存放第i网页的出链数量
        # graph[i][2]存放第i网页的出链网页，是一个列表
        self.graph = {}
        for node in dg.nodes():
            self.graph[node] = [1.0 / self.num_of_pages, len(dg.neighbors(node)), dg.neighbors(node)]

    def ip_mapper(self, input_key, input_value):
        """
        看一个网页是否有出链，返回值中的1没有什么物理含义，只是为了在map_reduce中的groups字典的key只有1，对应的value为所有的悬挂网页的PR值
        :param input_key: 网页名，如A
        :param input_value: self.graph[input_key]
        :return: 如果没有出链，即悬挂网页，那么就返回[(1,这个网页的PR值)]；否则就返回[]
        """
        if input_value[1] == 0:
```

```
            return [(1, input_value[0])]
        else:
            return []

    def ip_reducer(self, input_key, input_value_list):
        """
        计算所有悬挂网页的 PR 值之和
        :param input_key: 根据 ip_mapper 的返回值来看，这个 input_key 就是:1
        :param input_value_list: 所有悬挂网页的 PR 值
        :return: 所有悬挂网页的 PR 值之和
        """
        return sum(input_value_list)

    def pr_mapper(self, input_key, input_value):
        """
        mapper 方法
        :param input_key: 网页名，如 A
        :param input_value: self.graph[input_key]，即这个网页的相关信息
        :return: [(网页名, 0.0), (出链网页 1, 出链网页 1 分得的 PR 值), (出链网
页 2, 出链网页 2 分得的 PR 值)...]
        """
        return [(input_key, 0.0)] + [(out_link, input_value[0] /
input_value[1]) for out_link in input_value[2]]

    def pr_reducer_inter(self, intermediate_key, intermediate_value_list,
dp):
        """
        reducer 方法
        :param intermediate_key: 网页名，如 A
        :param intermediate_value_list: A 所有分得的 PR 值的列表:[0.0,分得的
PR 值,分得的 PR 值...]
        :param dp: 所有悬挂网页的 PR 值之和
        :return: (网页名，计算所得的 PR 值)
        """
        return (intermediate_key,
                self.damping_factor * sum(intermediate_value_list) +
                self.damping_factor * dp / self.num_of_pages +
                (1.0 - self.damping_factor) / self.num_of_pages)

    def page_rank(self):
        """
        计算 PR 值，每次迭代都需要两次调用 MapReduce。一次是计算悬挂网页 PR 值之
和，一次
        是计算所有网页的 PR 值
```

```
        :return: self.graph，其中的 PR 值已经计算好
        """
        iteration = 1  # 迭代次数
        change = 1  # 记录每轮迭代后的 PR 值变化情况，初始值为 1 保证至少有一次迭代
        while change > self.min_delta:
            print("Iteration: " + str(iteration))

            # 因为可能存在悬挂网页，所以才有下面这个 dangling_list
            # dangling_list 存放的是[所有悬挂网页的 PR 值之和]
            # dp 表示所有悬挂网页的 PR 值之和
            dangling_list = MapReduce.map_reduce(self.graph, self.ip_mapper, self.ip_reducer)
            if dangling_list:
                dp = dangling_list[0]
            else:
                dp = 0

            # 因为 MapReduce.map_reduce 中要求的 reducer 只能有两个参数，而我们
            # 需要传 3 个参数（多了一个所有悬挂网页的 PR 值之和,即 dp），所以采用
            # 下面的 lambda 表达式来达到目的
            # new_pr 为一个列表，元素为:(网页名，计算所得的 PR 值)
            new_pr = MapReduce.map_reduce(self.graph, self.pr_mapper, lambda x, y: self.pr_reducer_inter(x, y, dp))

            # 计算此轮 PR 值的变化情况
            change = sum([abs(new_pr[i][1] - self.graph[new_pr[i][0]][0]) for i in range(self.num_of_pages)])
            print("Change: " + str(change))

            # 更新 PR 值
            for i in range(self.num_of_pages):
                self.graph[new_pr[i][0]][0] = new_pr[i][1]
            iteration += 1
        return self.graph
```

4 算法缺点

- 没有区分站内导航链接

很多网站的首页都有很多对站内其他页面的链接，称为站内导航链接。这些链接与不同网站之间的链接相比，肯定是后者更能体现 PageRank 值的传递关系。

- 没有过滤广告链接和功能链接

这些链接通常没有什么实际价值，前者链接到广告页面，后者常常链接到某个社交网站首页。

- 对新网页不友好

一个新网页的一般入链相对较少，即使它的内容的质量很高，要成为一个高 PR 值的页面仍需要很长时间的推广。

5 结论

在此之前，尽管对于搜索结果的排序感到很吃惊，但是没有详细去研究过，总是以为科技就是这样先进，却不知道这些技术当年也是由爱思考专研的研究生得来的。今天，开始学习这一被誉为“数据挖掘十大优秀算法之一”的 PageRank，慢慢开始敬佩搜索引擎内部的优秀技术，当然也激励自己更加认真地学习。

6 参考文献

1. 《这就是搜索引擎：核心技术详解》，张俊林
2. 当年 PageRank 诞生的论文：The PageRank Citation Ranking: Bringing Order to the Web
3. 维基百科 PageRank
4. PageRank 算法简介及 Map-Reduce 实现
5. 博客《谷歌背后的数学》，卢昌海
6. 博客 PageRank 背后的数学
7. 深入浅出 PageRank 算法
8. MapReduce 原理与设计思想
9. Using MapReduce to compute PageRank

7 K-means 实现

7.1 算法原理

K-Means 算法是硬聚类算法，是典型的基于原型的目标函数聚类方法的代表。它是将数据点到原型的某种距离作为优化的目标函数，利用函数求极值的方法得

到迭代运算的调整规则（如图 3 所示）。K-Means 算法以欧氏距离作为相似度测度，求对应某一初始聚类中心向量 V 最优分类，使得评价指标最小。算法采用误差平方和准则函数作为聚类准则函数。

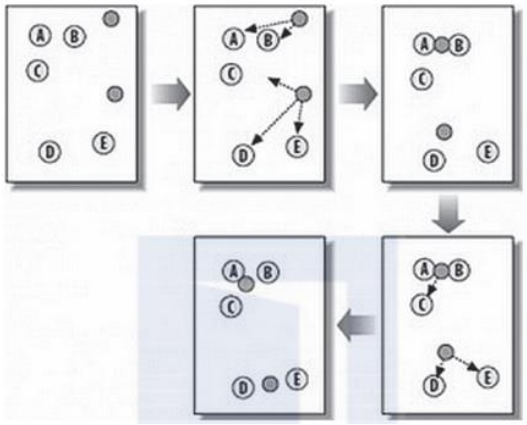


图 3 K-means 算法聚类过程图

具体的算法步骤如下：

1）随机在图中取 K（这里 K=2）个种子点。

2）然后对图中的所有点求到这 K 个种子点的距离，假如点 Pi 离种子点 Si 最近，那么 Pi 属于 Si 点群。图 2-45 中，我们可以看到 A、B 属于上面的种子点，C、D、E 属于下面中部的种子点。

3）接下来，我们要移动种子点到属于它的“点群”的中心。见图 4 中的第 3 步。

4）然后重复第 2）和第 3）步，直到种子点没有移动。我们可以看到图 4 中的第 4 步上面的种子点聚合了 A、B、C，下面的种子点聚合了 D、E。

图 4 所示为 K-Means 算法的流程图。

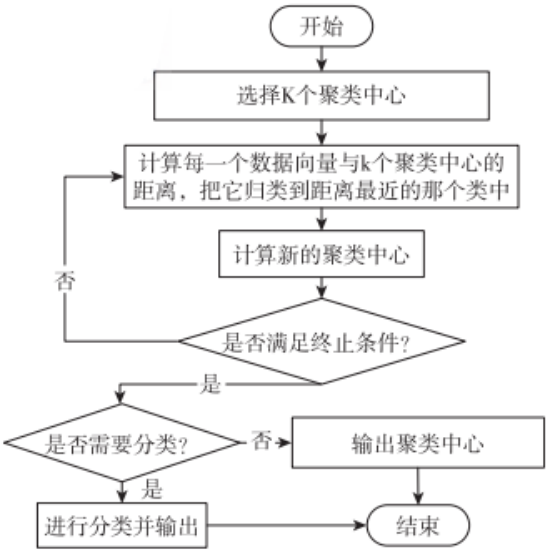


图 4 K-means 算法流程图

7.2 MapReduce 实现

1. 用一个全局变量存放上一次迭代后的质心

2. map 里，计算每个质心与样本之间的距离，得到与样本距离最短的质心，以这个质心作为 key，样本作为 value，输出

3. reduce 里，输入的 key 是质心，value 是其他的样本，这时重新计算聚类中心，将聚类中心 put 到一个全部变量 t 中。

4. 在 main 里比较前一次的质心和本次的质心是否发生变化，如果变化，则继续迭代，否则退出。

注：具体见 Github 代码实现部分