随机算法课程

实验报告

实验一：用 minHash 实现集合的相似性连接

姓名：袁野

学号：1190200122

班级：1903102

评分表：（由老师填写）

最终得分：	
对实验题目的理解是否透彻：	
实验步骤是否完整、可信 ：	
代码质量 ：	
实验报告是否规范 ：	
趣味性、难度加分 ：	
特 色：	1
	2
	3

一、实验题目概述

minhash 算法可以用来快速计算集合的相似度，在本实验中，通过实现二重循环判断算法和 minhash 算法来实现几何相似度的求解问题，比较两种算法的差别，同时对比不同参数设定后的效果区别。进一步地，通过对 minhash 进行改进来加速 minhash 算法，加深对 minhash 随机算法地认知。

二、对实验步骤的详细阐述

1、首先进行数据分析，将所有的数据读入，并计算不同数据集的集合个数、元素个数、集合大小分布等描述集族的特征信息，用以后续分析。根据分析，数据是以集合-元素的形式给定在文件里，因此将所有这样的二元组按照集合为第一关键字，元素为第二关键字进行排序，然后使用 C++ STL 中的 vector 储存集合，这样做可以很方便得保证所有集合对应得 vector 内的元素都是有序的，同时降低代码难度，便于后续使用不同方法进行处理。

2、naïve 算法的实现：两层循环分别枚举集合，对于两个不同的集合，我们采用线性比较的方式，分别建立两个指针指向两个集合所对应的 vector 第一个元素，由于集合元素是有序的，因此我们只需要将两个指针中指向元素较小的指针向后移动，并沿途寻找相同的元素对数，然后计算相似度即可。最后统计相似度大于阈值的集合对数以及运行时间。

3、minhash 算法的实现：由于在这里我们更在意的是一个元素被哪些集合包含，因此我们可以转变储存思路，用 vector 记录每个元素被包含的集合序号，这样的话我们可以在计算某一个元素在通过某一个哈希函数映射之后，直接更新所有相关集合对应的签名矩阵里的信息，简化代码过程。

而关于若干哈希函数的设定，由于 minhash 的实质是将所有的元素随机排列后统计每个集合出现最早的元素的行号，哈希函数只是将随机排列元素这个过程简化，所以对于所有元素个数 n，如果我们要设置 m 个哈希函数，那么根据同余的相关知识，只需要找到 m 个与 n 互质的互不相同的数字a_i，通过公式 $y = a_i x + b_i (mod\ n)$（y 是映射之后的数字，x 是元素本身，a_i如上文所述，b_i为随机生成的正数）计算，那么所有与元素在同一个哈希函数的映射下的结果都小于 n 且互不相同，即在线性时间内产生了一个 n 的全排列。而对于每个元素映射之后的值，只需要在签名表中的对应函数、对应集合位置与该处的原数字取较小值即可得到一张签名表。

最后我们只需二次循环来枚举集合，并检查结果一致的函数所占的比例，即作为这两个集合的相似度。由于对于部分集合来说，集合大小可能远远小于哈希函数的个数，上述的过程实际上是在“升维”，所以我在代码中对于集合较小的元素对采用 naïve 算法计算其相似度。

4、改进的 minhash 算法：lsh 算法对于 minhash 中得到 h*m 的签名表(h 为哈希函数个数，m 为集合个数)，我们可以将其纵向分为 b 个 brand，每个 brand 大小为 r，即满足 h=r*b，然后对于每一个 brand 来说，将每一个集合在该 brand 内的签名通过具有 local sensitive 的哈希函数进行再次映射，并通过桶统计映射值相同的集合对，这些映射值相同的集合对我们认为他们是有极大概率相似的集合对，反之我们认为这些集合大概率不相似，那么就不对其进行任何操作。将所有的 brand 进行这样的操作之后，就会得到一个包含所有“可能相似”的集合对的表，根据这上边的内容进行比较，这样比较次数就会减少，我们只会对“可

能相似”的集合对进行比较。而关于 lsh 算法更进一步的讨论将在第五部分中提及。

三、实验数据

1. 实验设置

实验环境：

Ubuntu20.04.4 LTS(GNU/Linux 5.10.102.1-microsoft-standard-WSL2 x86_64)

数据：

数据集	集合个数	元素个数	最大集合
E1_kosarak_100k.txt	99999	23496	1136
E1_Booking-out.txt	532751	105283	1921
E1_AOL-out.txt	65516	1216654	3755

2. 实验结果

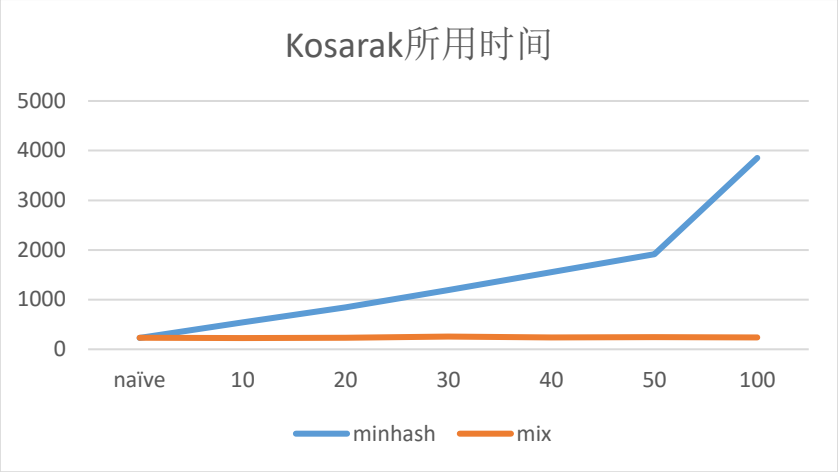
由于在实验过程中发现 minhash 算法中对于较小集合的 naïve 替代实际上会影响我对该算法时间效率的分析，因此以下的时间为未采用 naïve 替代的 minhash 算法。

	Kosarak		Booking		AOL	
	运行时间	答案超出比例	运行时间	答案超出比例	运行时间	答案超出比例
naive	228.753s	0.00%	205.413s	0.00%	69.3798s	0%
minHash_10	539.186s	580.22%	504.447s	0.42%	205.055s	56.48%
minHash_20	843.58s	215.24%	791.719s	0.10%	346.179s	69.70%
minHash_30	1194.77s	98.72%	2148.96s	0.01%	484.401s	29.83%
minHash_40	1550.94s	46.91%	3738.12s	0.03%	658.124s	15.91%
minHash_50	1912.53s	80.67%	4539.35s	0.0025%	878.076s	5.52%
minHash_100	3853.28s	8.38%	10517s	0.001889%	1549.75s	1.55%

而当 minhash 和 naïve 结合的算法也是有一定分析价值的，时间如下：

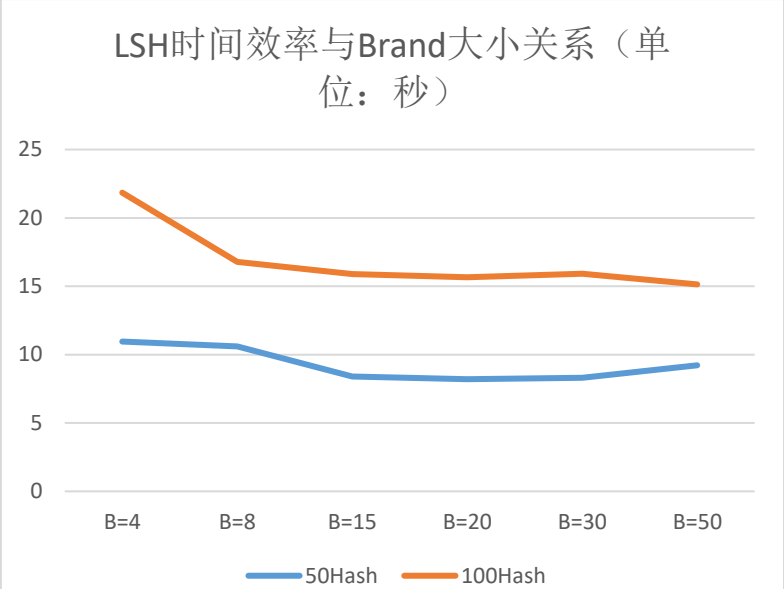
	Kosarak		Booking		AOL	
	运行时间	答案超出比例	运行时间	答案超出比例	运行时间	答案超出比例
naive	228.753s	0.00%	205.413s	0.00%	228.753s	0.00%
mix_10	226.158s	0.07%	573.242s	0.0000107%	65.9929s	0.01%
mix_20	229.514s	0.000006%	580.412s	0.00%	68.4324s	0.00055%
mix_30	256.094s	0.00%	590.313s	0.00%	71.702s	0.00%
mix_40	238.131s	0.00%	591.103s	0.00%	78.8407s	0.00%
mix_50	245.673s	0.00%	588.519s	0.00%	83.2951s	0.00%
mix_100	239.124s	0.00%	600.535s	0.00%	106.447s	0.00%

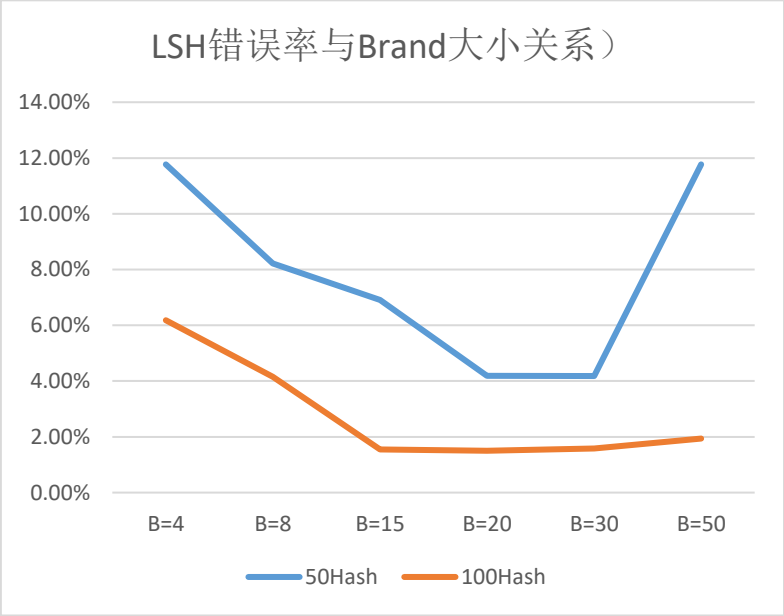
我们将 Kosarak 的效果作图



对于 LSH 部分，我运行了 AOL 数据集，对于不同的 brand 大小 b，结果如下：

	50 哈希函数		100 哈希函数	
	运行时间	答案超出比例	运行时间	答案超出比例
B=4	10.9545s	11.77%	21.8453s	6.18%
B=8	10.6085s	8.22%	16.7815s	4.15%
B=15	8.39167s	6.91%	15.8881s	1.55%
B=20	8.20263s	4.19%	15.6489s	1.50%
B=30	8.31025s	4.18%	15.9148s	1.58%
B=50	9.22031s	11.77%	15.1348s	1.94%



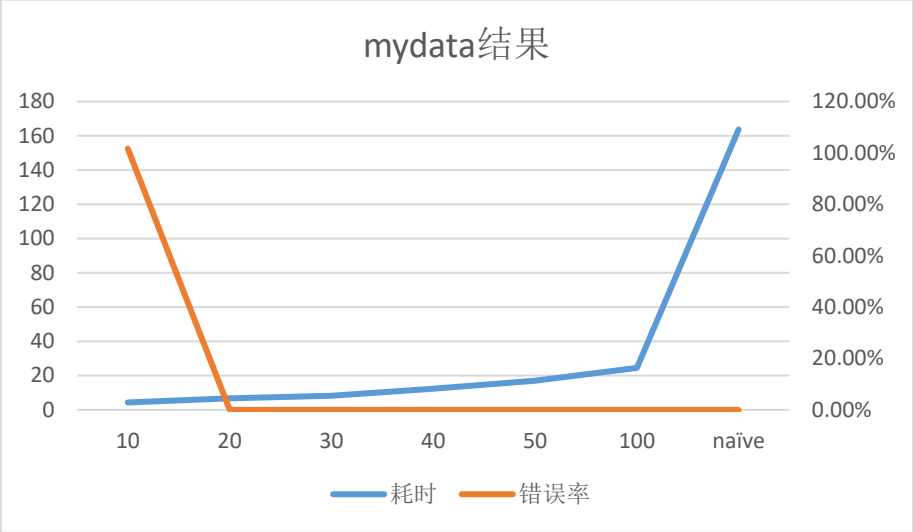


四、对实验结果的理解和分析

1、minhash 运行时间与哈希函数的个数基本呈线性关系，而为什么均比 naïve 要快的原因，我的猜想是实际上数据集较小的集合占大多数，所以实际上 minhash 对数据集造成了前面提到的升维负优化，而二者混合的 mix 算法时间效率上与 naïve 如出一辙也可以很好的印证这一点。我自己生成了一个集合大小在 1000 左右的数据集，得到的结果如下：

	Mydata	
	运行时间	答案超出比例
naive	163.674s	0.00%
minHash_10	4.3322s	101.62%
minHash_20	6.7239s	0.00024%
minHash_30	8.2398s	0.0001%
minHash_40	12.3755s	0.00002%

minHash_50	16.962s	0.00%
minHash_100	24.4559s	0.00%



可以看到对于大一些的数据集，minhash 的优化效果还是比较明显的，而且正确率也相对可观。Booking 这个数据过于庞大，因此我截取了前 100000 个集合进行计算，发现其 minhash 的正确率时三个数据集中最好的，但是时间效率是最差的。

2、我们观察错误率可以发现，错误率与哈希函数个数是呈指数关系分布的，哈希函数越多，错误率越低，但是错误率降低的速度也越来越慢。

3、对于不同哈希函数个数的 minhash 算法，我们很容易发现，哈希函数的个数越少，运行时间越快，但错误率极高，哈希函数越多，运行速度越慢，错误率越低，综合来看，对于这四个数据集，哈希函数个数在 30 到 40 之间为最佳。但是实际在使用时，应当结合数据集的具体特点来进行分析，比如数据集集合大小的分布等，这是十分重要的。

4、由于我实现的 LSH 算法十分依赖内存，因此只能执行 AOL 数据集，我们观察结果发现，哈希函数越多，其时间消耗越大，但是其错误率越小 ；而关于每一个 brand 的大小，brand 越小，导致后面比较的次数越多，时间效率越低，brand 越大，前边用桶维护的时间效率就越低，因此 brand 的大小要找到合适的区间。通过观察我们发现，哈希函数为 100 且 brand 大小在 15 到 30 之间时的时间稳定在 16s 左右，错误率不超过 2%，是一个相对比较优秀的情况。由于我自己实现的问题，实际上效果可以更优。

五、实验过程中最值得说起的几个方面

1、 本次实验中的三个数据集其实各有特点，而无论是 naïve 还是 minhash 都无法体现出其压倒性的优势，因此在实际应用中需要将这两种算法相结合，显然这样的效果是最好的。
2、 在 minhash 的实现过程中采用了倒排索引，通过元素索引包含它的集合，这样在更新 hash 签名矩阵时可以更方便、高效地找到需要更新的地方，效率会更好。
3、 由于三个数据集均有大部分集合很小的特点，因此在这些数据集上的对比实际上有些吃力，实际上的效率也不尽如人意，如果是一个相对集合较大的数

据集，比如几篇文章，minhash 或 lsh 的优势会被很有效地放大。

4、 这里展开分析一下 lsh 算法：首先 minhash 算法是对原数据的降维，而降维之后仍需要进行平方时间规模的对比，所以效率实际上并没有提高多少，因此我们考虑如何将平方级别规模的比较时间省略。在求出 hash 签名矩阵之后，lsh 算法将对哈希签名矩阵行进行分割，分割成 B 个 brand（一个 brand 有 $r = k/b$ 行）。对于每一个 brand，我们将 n 个人的在当前 brand 中的 $r * 1$ 的子矩阵采用具有 local sensitive 特性的函数进行再次哈希，按照哈希值将其放进桶内，此时我们保证桶数量尽可能多。此时，我们根据 local sensitive 的特点可以知道，最终哈希值落在临近桶内的原列对应的集合是“可能相似”的，这样的话我们只需要比较每个 brand 内“可能相似”的集合即可，这样可以大大减少无效对比（即比较完之后发现是不相似的集合对）。设两个集合的相似度为 s ，那么在所有 brand 中至少存在一个 brand 使得二者的哈希值相同的概率为$Pr = 1 - (1 - s^r)^B, s \in [0,1]$。可预见的是，根据阈值可以调整合适的参数，使得对于两个相似度超过阈值的集合可以有极大概率在某一个 brand 内被比较。

5、 说一些与实验内容无关，但是与实验方式有关的东西。我采用的是 C++语言编写实验代码，期间为了可以更快速得到实验的结果数据，我采用了-Ofast 优化，虽然这个命令确实可以加快我的实验进程，但是并不利于我对实验的观察。由于-Ofast 优化开关的原理是对代码的底层逻辑和一些实现细节进行优化，因此，当程序体量越大，数据量越大时，它的优化效果就越明显，换言之，原本运行时间与输入数据量成正比的一份代码，在优化之后就不在成线性，而是随数据量增大逐渐偏离线性。甚至会有，由于输出量太大，我将输出删掉，原本几百秒的代码两三秒就跑完，而我错以为是 IO 时间太长，实际上是当输出删除掉之后，编译器的优化开关会认为这部分代码运行与否不影响最终的输出（事实也的确是这样），因此这二重循环便没有运行，从而导致了“效率骤升”的假象。因此，在观察实验现象时，应该及时考虑到编译器等无关因素对于实验结果的影响。