# 《数据科学与工程算法》项目报告

报告题目： 基于局部敏感哈希的图节点相似性查询系统

姓 名： 温兆和

学 号： 10205501432

完成日期： 2023.04.28

摘要 [中文]：

局部敏感哈希是最为著名的寻找相似集合的算法之一，它的基本思想就是用一组哈希函数求出集合的哈希值，把哈希签名矩阵分块，并根据每一个分块中两个集合的哈希签名向量是否相等来判断这两个集合是否相似。在多探寻局部敏感哈希算法中，我们可以进一步放宽集合相似的标准，将哈希签名向量相似的集合判断为相似，让更多的集合称为目标集合的备选相似集合，从而提高算法的准确性。

Abstract [English]

Locality-Sensitive Hashing (LSH) is one of the most popular algorithms on similarity search. Its basic idea is to work out the hash values of the set on a group of hash functions, divide the hash value matrix into several parts and decide whether two sets are similar by comparing their hash value vectors in each part of the hash value matrix. In Multi-Probe Locality-Sensitive Hashing, the standard of the similarity between two sets is further loosen: two sets can be judged as ‘similar’ if their hash value vectors are similar with each other so that more sets can be selected to the candidate group of the goal set, thus enhancing the accuracy of the algorithm.

1. 项目概述（阐明项目的科学价值与相关研究工作，描述项目主要内容）

**1.1项目的科学价值**

在本次实验中，数据集是表示研究人员之间共同作者关系的无向图。我们需要构建一个LSH方案，争取在更短的时间内返回邻居集合与输入节点的邻居集合Jaccard相似度最高的十个节点（不包括输入节点自身）。这样做的意义在于，在海量高维数据的前提下牺牲一定的准确率，降低数据的维度，提升查询的速度。

**1.2项目使用的算法及其相关研究工作**

在课堂上，我们已经学习了最传统的LSH算法，即用多个哈希函数求出集合的哈希值，再对哈希签名矩阵进行分组，如果在任何一个分组中某个集合的哈希签名向量与目标集合的哈希签名向量相等，我们就认为这个集合与目标集合相似。这里的哈希函数是这样的：先把集合表示为一个只有0和1的向量，如果集合中有全集中的某个元素，那么相应元素就是1，否则就是0。接着对集合向量的行索引进行重排列，并将第一个不为0的行索引值作为集合的哈希值。近年来，有不少学术研究对传统LSH算法作出了改进：有些改进是针对哈希函数的；也有些改进，如多探寻局部敏感哈希[[1]](#footnote-1)，则是通过放大相似集合的范围来提升查询的准确率。

**1.3项目主要内容**

在本次实验中，我们首先对数据集进行预处理，随机生成尽可能多的哈希函数并求出每个集合的哈希值；然后，实现传统LSH算法并测试其运行时间和准确度；最后，实现多探寻LSH并通过比较探究它的性能（时间、空间、准确度）是否比传统LSH算法有所改进。

1. 问题定义（提供问题定义的语言描述与数学形式）

**2.1语言描述：**

最原始的数据集是这样的：每一行有两个数字，每个数字代表图中的一个节点，每一行代表两个节点间有一条边。对于一个节点来说，所有跟它有边的节点构成了它的邻居集合。对于我们要查询的节点，算法应该返回所有节点中邻居集合与查询的节点的Jaccard相似度最高的十个节点（不包括查询节点自身）。

**2.2数学形式：**

给定图

输入

其中节点的邻居集合与查询节点的邻居集合的Jaccard相似度定义为

需要返回数列中最大的十个数字的索引。

1. 方法（问题解决步骤和实现细节）

**3.1数据集预处理**

该部分的所有工作均在lab1\_dataprocess.ipynb文件中体现。

首先，逐行读取文件ca-AstroPh.txt中的数据，将其存入列表data中并将所有数据的类型从字符串转换为整型：

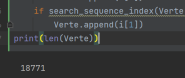
data = []  
file = open('ca-AstroPh.txt','r')  
file\_data = file.readlines()  
for i in range (1,len(file\_data)):  
 tmp\_list = file\_data[i].split(' ')  
 tmp\_list[-1] = tmp\_list[-1].replace('\n','')  
 data.append(tmp\_list)  
for i in data:  
 i[0]=int(i[0])  
 i[1]=int(i[1])

利用Python中的networkx工具创建无向图Charistic\_Matrix，并将data列表中的边逐一添加到这个无向图中：

Characteristic\_Matrix = nx.Graph()  
for i in data:  
 Characteristic\_Matrix.add\_edge(i[0]-1,i[1]-1)

通过读取无向图中的节点，我们可以发现这张图中有18771个节点：

Verte = []  
for i in data:  
 if search\_sequence\_index(Verte,i[0])==False:  
 Verte.append(i[0])  
 if search\_sequence\_index(Verte,i[1])==False:  
 Verte.append(i[1])  
print(len(Verte))

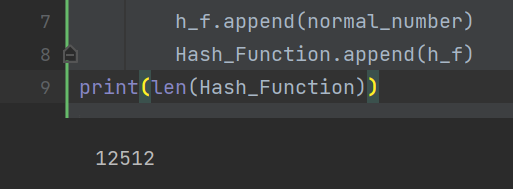


接着，我们需要构造尽可能多的哈希函数以对集合特征矩阵的行进行重排列。我们采用线性哈希函数，即

其中一次项系数必须与18771互素，常数项可以从0-18770之间随机生成。经过排查，我们发现1-18770之间共有12512个数字与18771互素，也就得到了12512个哈希函数：

def gcd(a, b):  
 if(a==1&b==1): # 两个正整数中，只有其中一个数值为1，两个正整数为互质数  
 return True  
 while True: #求出两个正整数的最大公约数  
 t = a%b  
 if(t == 0):  
 break  
 else:  
 a = b  
 b = t  
 if(b>1):  
 return False# 如果最大公约数大于1，表示两个正整数不互质  
 else:  
 return True# 如果最大公约数等于1,表示两个正整数互质

Hash\_Function = []  
for i in range (len(Verte)):  
 if (gcd(i+1,18771)):  
 h\_f = []  
 normal\_number = random.randint(0,18770)  
 h\_f.append(i+1)  
 h\_f.append(normal\_number)  
 Hash\_Function.append(h\_f)  
print(len(Hash\_Function))



接着，我们就能计算每个集合在这12512个哈希函数下的哈希值了。计算的方法是，从刚刚定义的无向图中读取每一个节点的邻居集合，将这个集合和某个哈希函数传入find\_msh()函数中后对集合中的每一个数字用传入的哈希函数作一次变换，最后返回变换后的所有数字中的最小值。

def find\_msh(neighbours,h\_f):  
 hash\_values = []  
 for i in neighbours:  
 hash\_values.append((i\*h\_f[0]+h\_f[1])%18771)  
 hash\_values.sort()  
 return hash\_values[0]

MSH = np.zeros((len(Verte),len(Hash\_Function)))  
for i in range (len(Verte)):  
 for j in range (len(Hash\_Function)):  
 neighbours = list(nx.all\_neighbors(Characteristic\_Matrix, i))  
 MSH[i][j]=find\_msh(neighbours,Hash\_Function[j])

最后，把数据预处理过程中生成的数据集、哈希函数和哈希签名矩阵存入csv文件，以便后续使用：

data\_in\_frame = DataFrame(data)  
data\_in\_frame.to\_csv('data.csv',index=0)

Hash\_Function\_in\_frame = DataFrame(Hash\_Function)  
Hash\_Function\_in\_frame.to\_csv('Hash\_Function.csv',index=0)

MSH\_in\_frame = DataFrame(MSH)  
MSH\_in\_frame.to\_csv('MSH.csv',index=0)

**3.2普通局部敏感哈希的实现**

该部分的所有工作均在lab1\_main.ipynb文件中体现。

首先，打开上一步生成的数据集、哈希签名矩阵并将数据集中的点添加到无向图Charactristic\_Graph中：

data = pd.read\_csv("data.csv")  
MSH = pd.read\_csv("MSH.csv")

Characteristic\_Graph = nx.Graph()  
for i in range(len(data)):  
 Characteristic\_Graph.add\_edge(data['0'][i]-1,data['1'][i]-1)  
Connection\_Matrix=np.array(nx.adjacency\_matrix(Characteristic\_Graph).todense())  
Verte = list(Characteristic\_Graph.nodes)

从所有18771个节点中任选五个作为本次实验的测试点：

test\_node = []  
for i in range (5):  
 test\_node+=[random.randint(0,18770)]

下面介绍普通LSH的实现思路。在函数LSH中，输入查询节点node、行条数b和每个行条的行数r。首先，根据b、r对哈希签名矩阵进行分块，然后遍历分块后的哈希签名矩阵，在每个行条中比较查询集合的签名向量和其它集合的签名向量。只要有一个行条中某个集合的签名向量与查询节点邻居集合的签名向量相等，就将其加入一个备选集。遍历完成后，调用similarest\_ten()函数，从备选集中找出与查询集合Jaccard相似度最高的十个集合并返回。

def LSH(node,b,r):# b个行条，每行r个哈希函数  
 global MSH  
 global G  
 global Verte  
 MSH\_tobe\_Verteilung = np.array(MSH.iloc[:, 0:b\*r])  
 volunteer = []  
 MSH\_Verteilung = np.array\_split(MSH\_tobe\_Verteilung, b, axis=1)  
 print("MSH\_Verteilung")  
 print(MSH\_Verteilung[0][node])  
 for i in range (len(MSH\_Verteilung)):  
 for j in range (len(Verte)):  
 if (difference\_less\_than\_one(MSH\_Verteilung[i][j],MSH\_Verteilung[i][node])==0) & (j!=node) & (j not in volunteer):  
 volunteer.append(j)  
 print("volunteer")  
 print(volunteer)  
 return similarest\_ten(node,volunteer)

在similarest\_ten()函数中，输入查询集合与备选集合。逐一计算查询集合与每个备选集合的Jaccard相似度并排序，返回Jaccard相似度最高的十个集合：

def similarest\_ten(node,Verte):  
 global Characteristic\_Graph  
 max\_index = []  
 Real\_Jaccard = pd.DataFrame(columns=['node','jaccard'],dtype=float)  
 for i in Verte:  
 df\_1 = pd.DataFrame([[i,jaccard(list(nx.all\_neighbors(Characteristic\_Graph, node)),list(nx.all\_neighbors(Characteristic\_Graph, i)))]],columns=['node','jaccard'],dtype=float)  
 Real\_Jaccard = pd.concat([Real\_Jaccard,df\_1], ignore\_index=True)  
 Real\_Jaccard=Real\_Jaccard.sort\_values(by="jaccard" , ascending=False)  
 print("Jaccard")  
 if len(Real\_Jaccard)>0:  
 print(Real\_Jaccard)  
 Real\_Jaccard\_list = np.array(Real\_Jaccard).tolist()  
 max\_ten = 10  
 if max\_ten>len(Real\_Jaccard\_list):  
 max\_ten=len(Real\_Jaccard\_list)  
 for i in range(max\_ten):  
 max\_index.append(Real\_Jaccard\_list[i][0])  
 if (len(max\_index)<10):  
 while (len(max\_index)<10):  
 max\_index.append(-1)  
 return max\_index

Jaccard相似度的计算方法是：两个集合交集中的元素个数除以两个集合并集中的元素个数：

def jaccard(A,B):  
 bigcap = len(list(set(A) & set(B)))  
 bigcup = len(list(set(A) | set(B)))  
 jaccard = bigcap/bigcup  
 return jaccard

这里我们还定义了一个difference\_less\_than\_one()函数。它在LSH()函数中被调用，是用来比较两个集合的哈希签名向量的。具体来说，它返回两个向量对应元素之差的绝对值的总和。

def difference\_less\_than\_one(a,b):  
 return np.abs(a-b).sum()

**3.3多探寻局部敏感哈希的实现**

该部分的所有工作均在lab1\_optimization.ipynb文件中体现。

多探寻局部敏感哈希的思路是这样的：除了根目标节点邻居集合签名向量完全相等的集合，签名向量相近的集合也被认为是相似的。比如，某个集合的签名向量是，那么签名向量为、或的集合也被认为与该集合相似。

所以说，多探寻局部敏感哈希的实现与普通局部敏感哈希相比只作了一点点改变：在LSH函数中多输入一个参数bound，只要两个哈希签名向量对应元素之差的绝对值小于bound就可以认为它们对应的两个集合相似。

def LSH(node,b,r,bound):# b个行条，每行r个哈希函数  
 global MSH  
 global G  
 global Verte  
 MSH\_tobe\_Verteilung = np.array(MSH.iloc[:, 0:b\*r])  
 volunteer = []  
 MSH\_Verteilung = np.array\_split(MSH\_tobe\_Verteilung, b, axis=1)  
 print("MSH\_Verteilung")  
 print(MSH\_Verteilung[0][node])  
 for i in range (len(MSH\_Verteilung)):  
 for j in range (len(Verte)):  
 if (difference\_less\_than\_one(MSH\_Verteilung[i][j],MSH\_Verteilung[i][node])<=bound) & (j!=node) & (j not in volunteer):  
 volunteer.append(j)  
 print("volunteer")  
 print(volunteer)  
 return similarest\_ten(node,volunteer)

另外，我们在多探寻局部敏感哈希的实现中沿用前面抽取到的五个测试点。它们在3.2的最后被存入csv文件。

selected\_node = pd.read\_csv("selected\_node.csv")

test\_node = []  
for i in range (5):  
 test\_node+=[int(selected\_node['node'][i])]

1. 实验结果（验证提出方法的有效性和高效性）

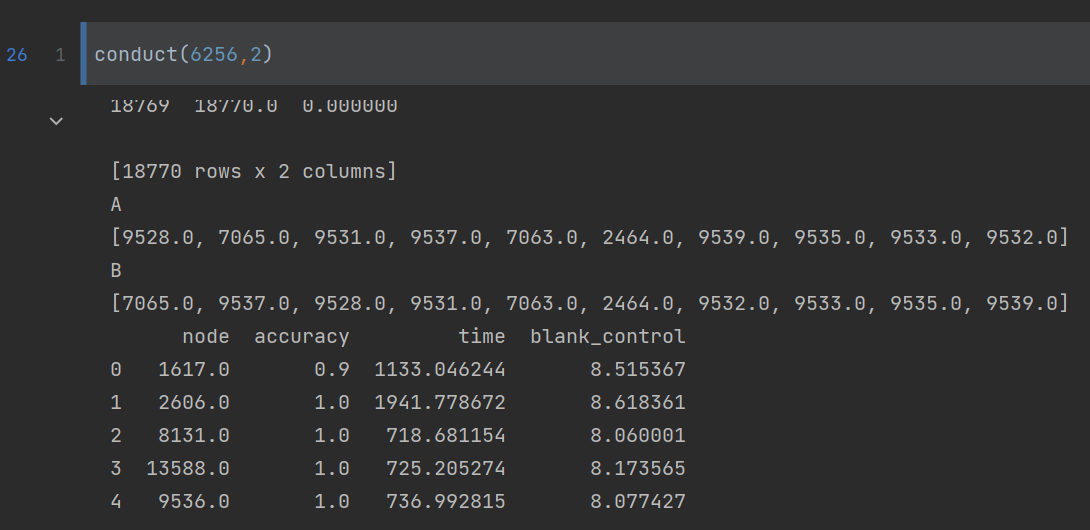
**4.1普通局部敏感哈希的运行结果**

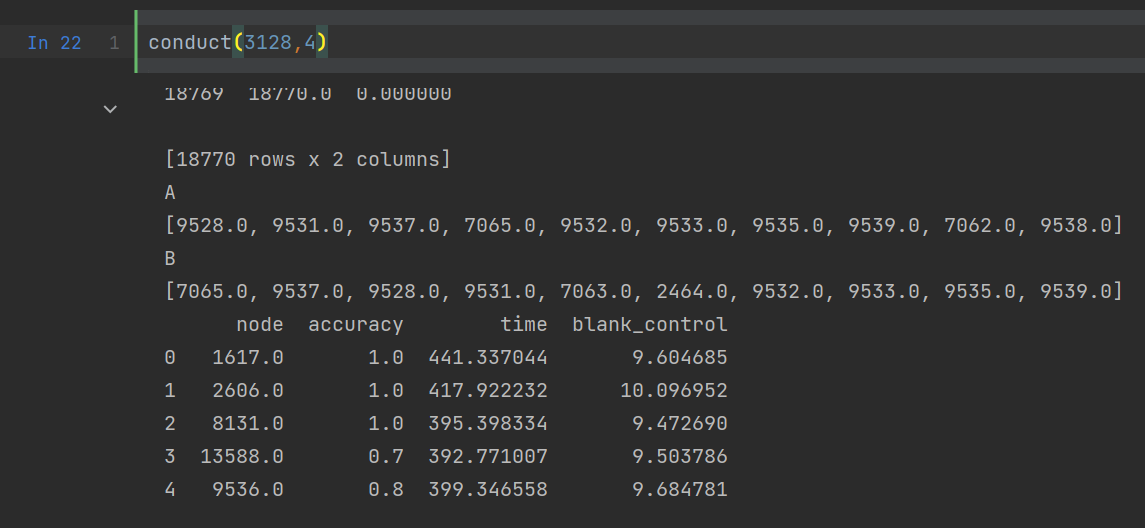
这里我们进行了三组测试，分别控制了三个不同的变量：b\*r、b和r。我们通过conduct()函数，分别计算局部敏感哈希与直接搜索最相近的集合的运行时间并计算正确率。正确率的计算方法是这样的：局部敏感哈希返回的十个节点组成集合A，直接搜索得到的十个节点（正确情况）组成集合B，用A与B交集中的元素个数除以B中元素的个数（十个）就是LSH的准确率。值得一提的是，A中有效元素的个数很可能少于十个，因为能够在LSH算法中进入候选集的节点就很有可能少于十个。

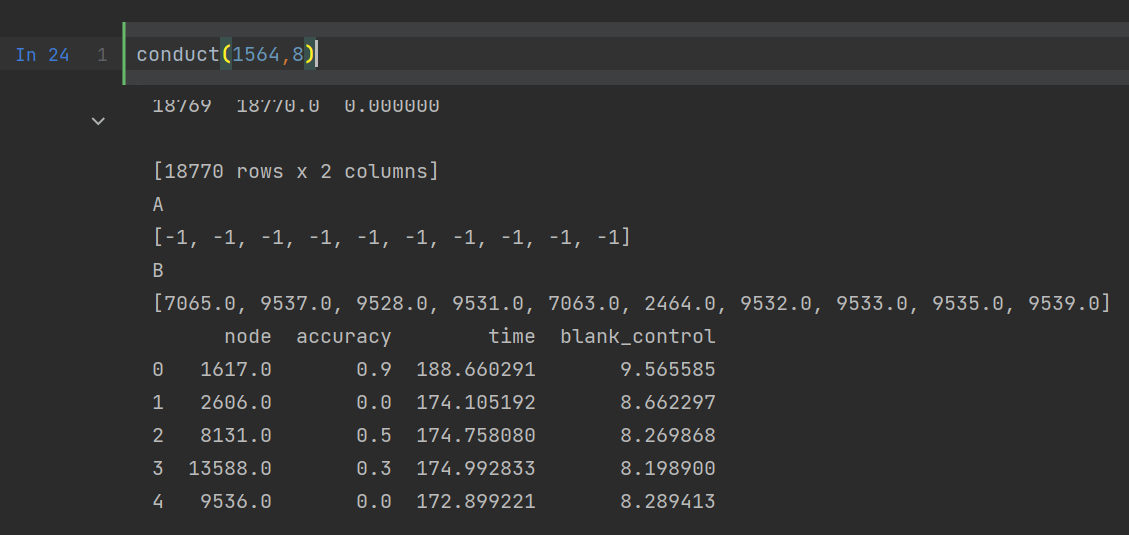
def conduct(b,r):  
 global test\_node  
 global Verte  
 df = pd.DataFrame(columns=['node','accuracy','time','blank\_control'],dtype=float)  
 for i in range (5):  
 print("-------------------")  
 print("i")  
 print(i)  
 t\_a=time.perf\_counter()  
 A = LSH(test\_node[i],b,r)  
 t\_b=time.perf\_counter()  
 Verte\_without\_i = list(Characteristic\_Graph.nodes)  
 if test\_node[i] in Verte\_without\_i:  
 Verte\_without\_i.remove(test\_node[i])  
 t\_c=time.perf\_counter()  
 B = similarest\_ten(test\_node[i],Verte\_without\_i)  
 t\_d=time.perf\_counter()  
 accuracy = len(list(set(A) & set(B)))/len(B)  
 run\_time = t\_b-t\_a  
 compare\_time = t\_d-t\_c  
 df1 = pd.DataFrame([[test\_node[i],accuracy,run\_time,compare\_time]],columns=['node','accuracy','time','blank\_control'],dtype=float)  
 df = pd.concat([df,df1], ignore\_index=True)  
 print("A")  
 print(A)  
 print("B")  
 print(B)  
 print(df)

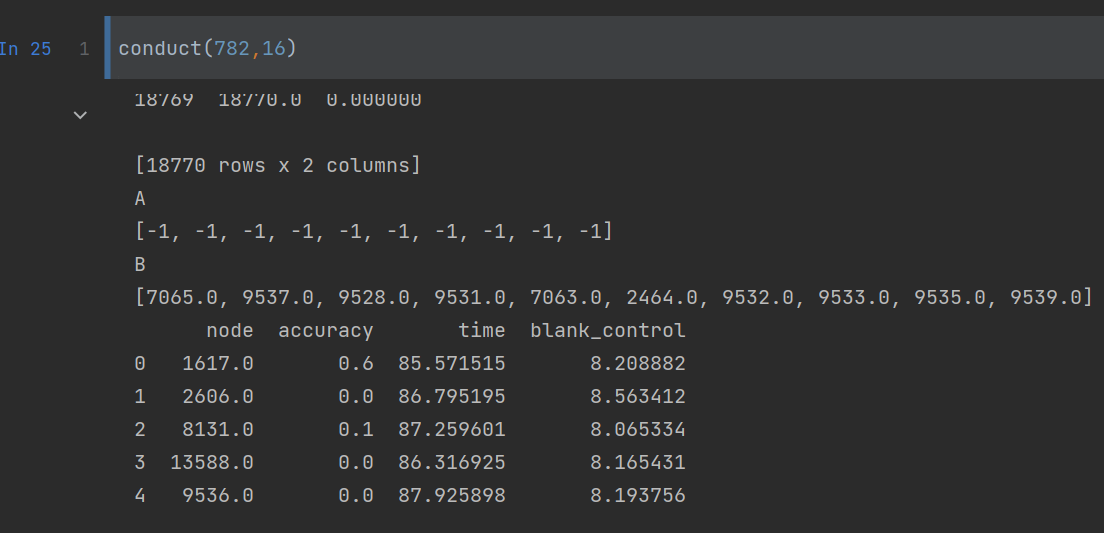
4.1.1 控制b\*r=12512

这里我们进行了四次实验，具体情况如下：

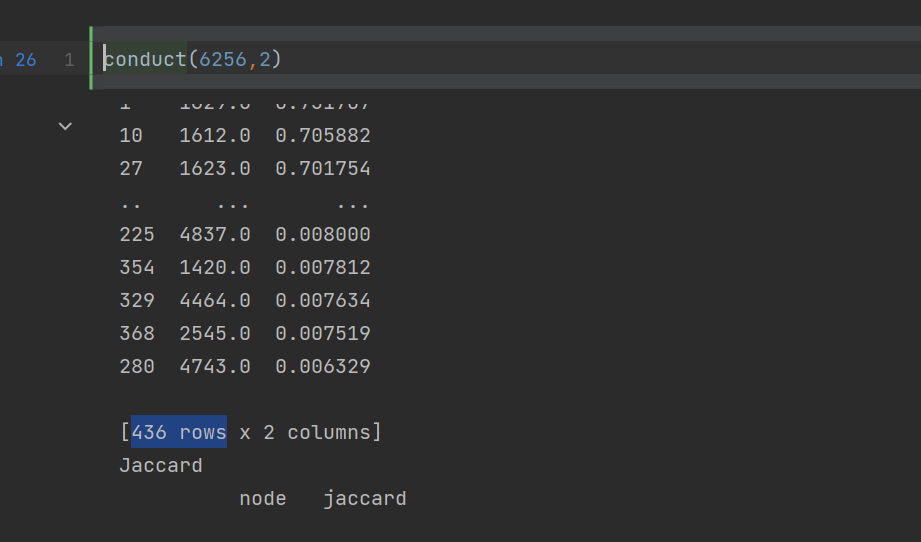




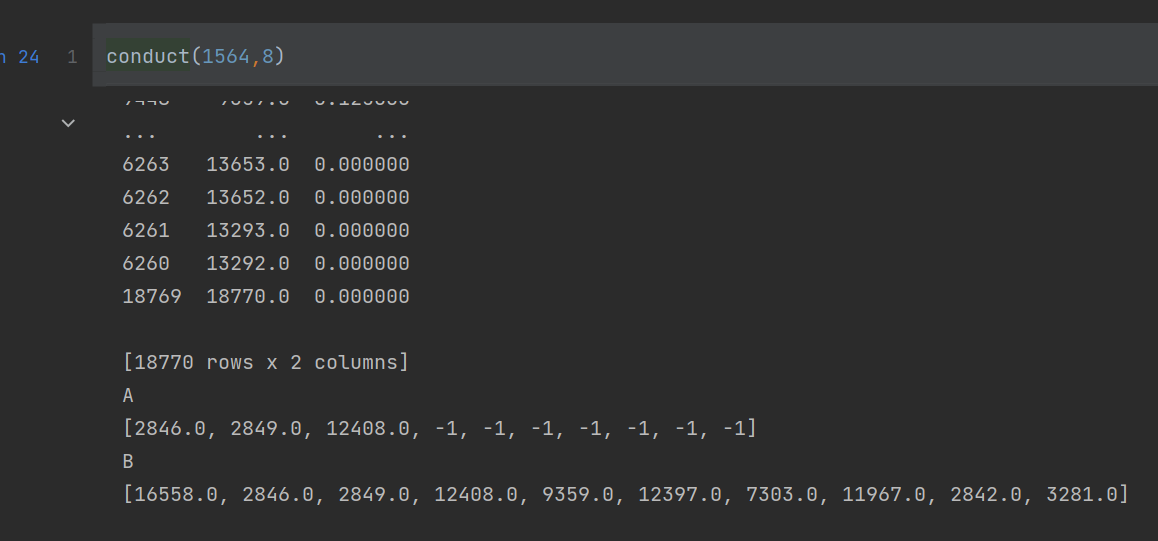




我们可以看到，随着行条数b的减少，运行时间也逐步减少，但无论在何种情况下，LSH的运行时间都远大于直接搜索的时间，原因在于直接搜索只调用了similarest\_ten()函数，其中最多只有一重for循环；而LSH在调用similarest\_ten()函数之前还要进行分桶操作，那里需要进行一个迭代次数为18771\*12512的双重循环，这耗费了大量的时间。但实际上，LSH还是在大大缩小了备选集范围的前提下达到了较高的正确率。以conduct(6256,2)的第一个测试节点为例，它直接把备选集的大小从原本的18770缩小到了436。如果集合所含有的不同元素再多一点（或者说集合特征向量的维数更大），或许LSH的性能就能得到更好的体现。

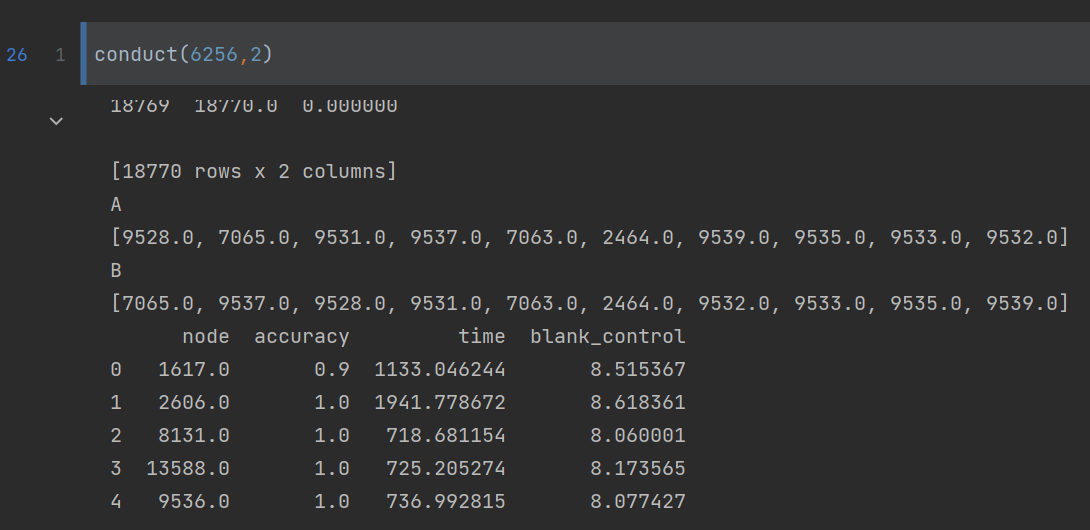


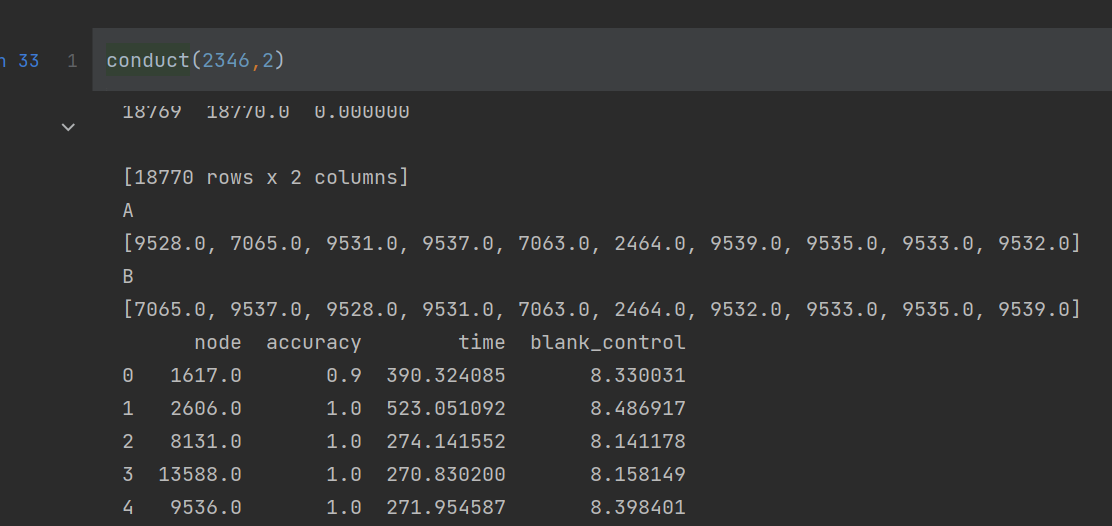
还值得注意的是，查询的正确率随着每个行条的行数r的增加而减少，这是因为两个集合被哈希到同一桶的概率随着r值的增加而减少。甚至当r比较大的时候，所有的节点无一进入备选集，导致得到的正确率为0。又如，在conduct(1564,8)的第四个测试节点中，备选集中只有三个节点入选。尽管入选的三个节点都是正确的，但由于数量不足本次实验要求的十个，得到的正确率还是0.3。

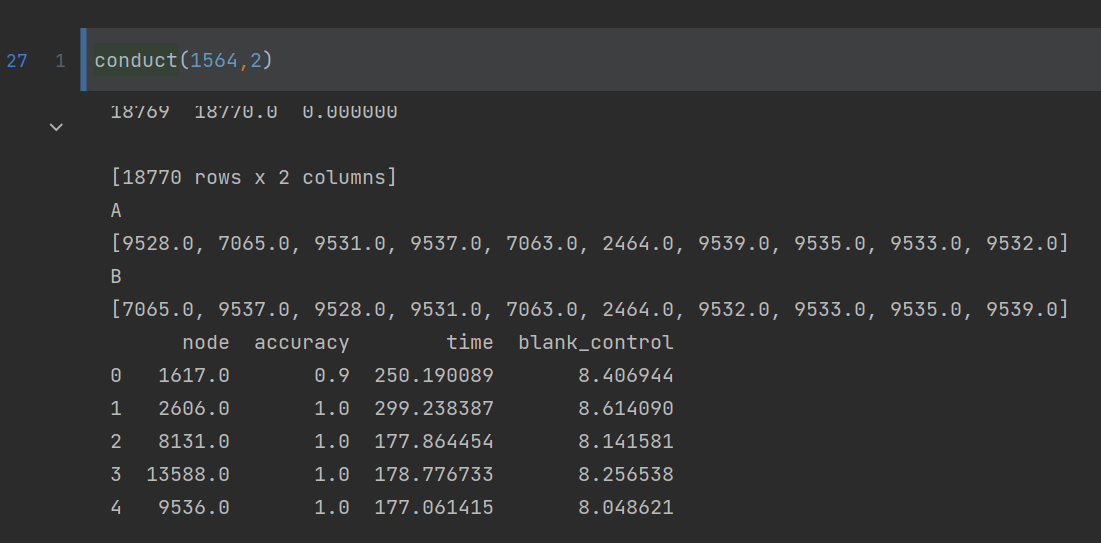


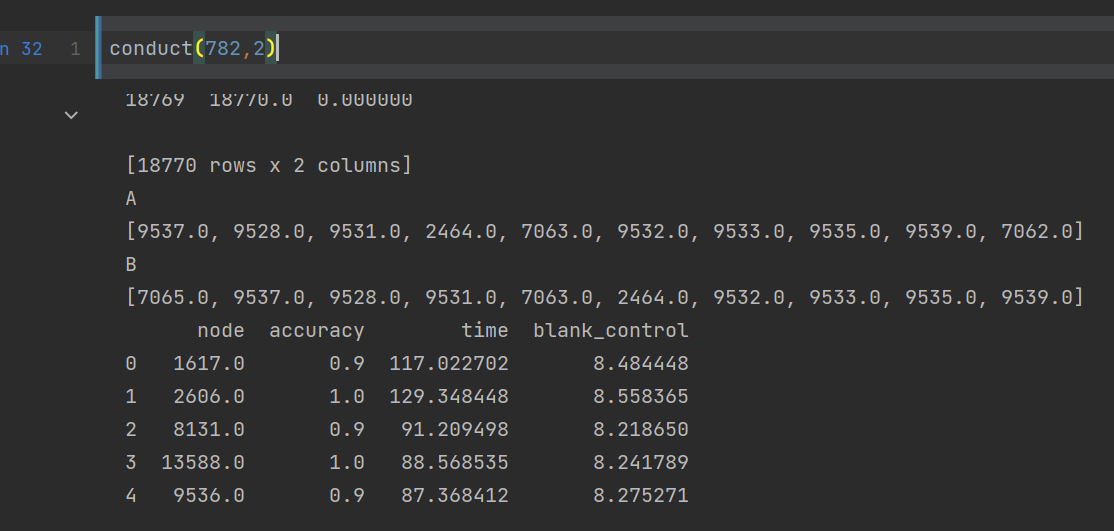
4.1.2 控制r=2

现在来探索控制每个行条的行数不变，改变行条数会对运行时间和正确率有什么影响。





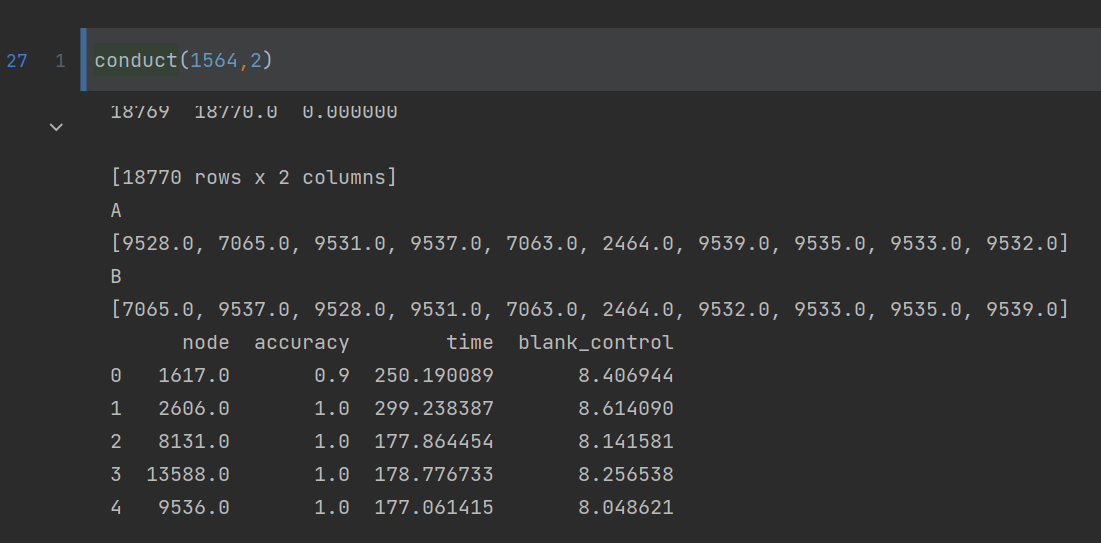


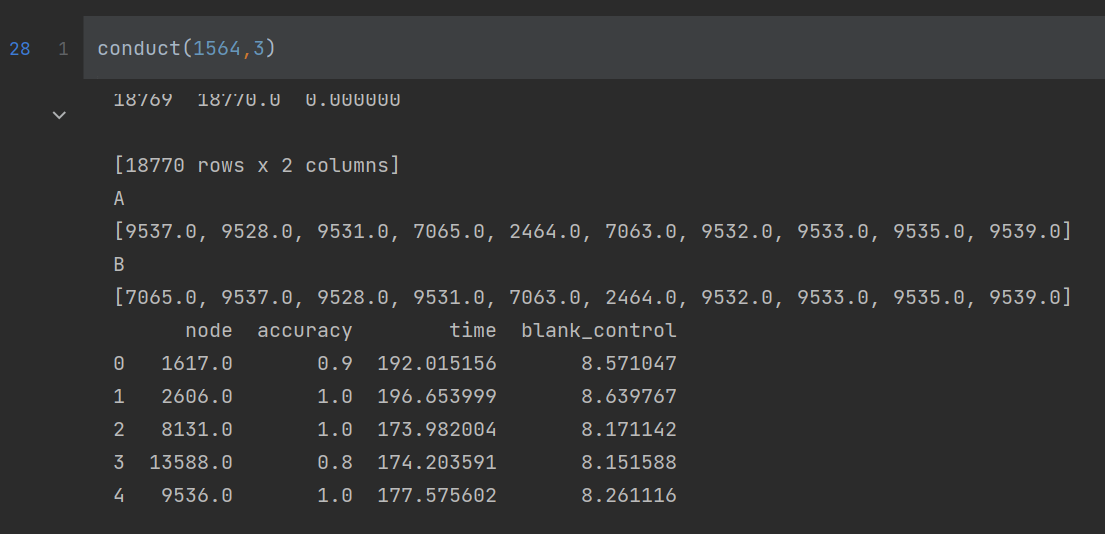


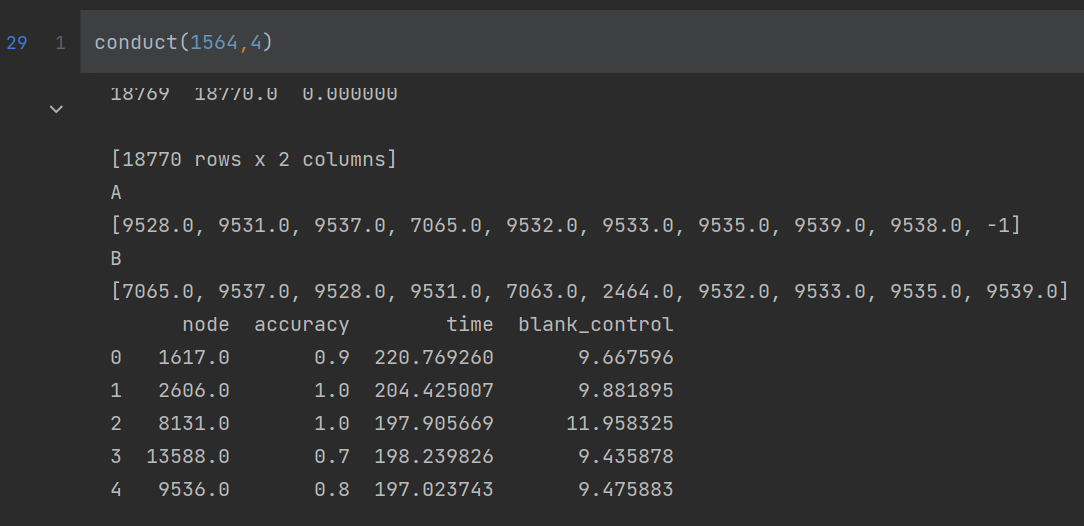
我们可以看到，随着b值的减少，正确率略有下降（因为哈希签名矩阵变小了，两个集合被哈希到同一个桶中的概率减少），但变化不大（因为如果两个集合真的相似，很可能在前面就已经遇到相同的哈希签名向量）。但是，运行的时间和算法所需的空间（即哈希签名矩阵的大小）都显著减少。

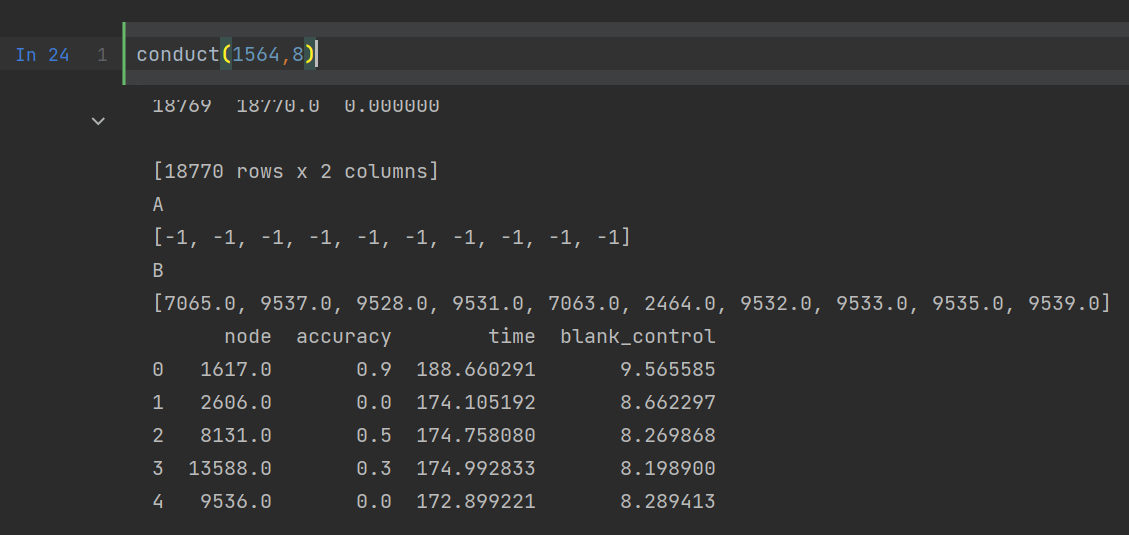
4.1.3 控制b=1564

最后研究控制行条数不变的情况下改变每一行的行数会对正确率和运行时间产生什么影响。





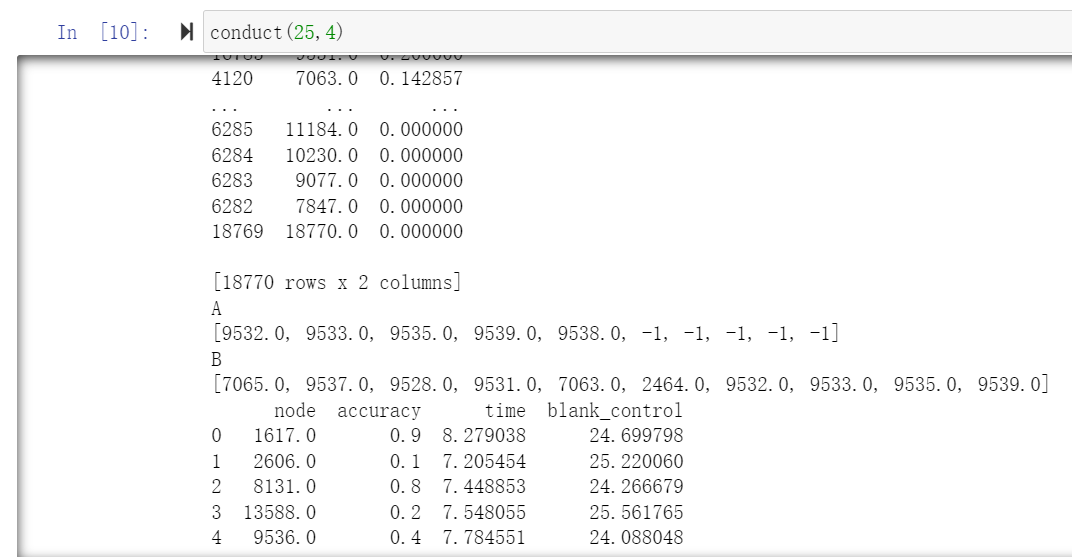


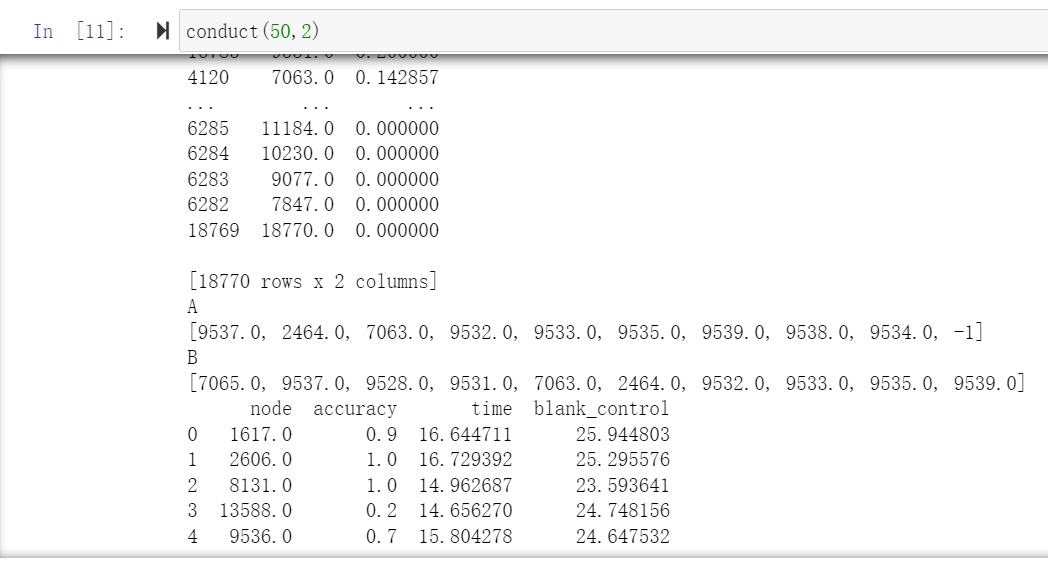


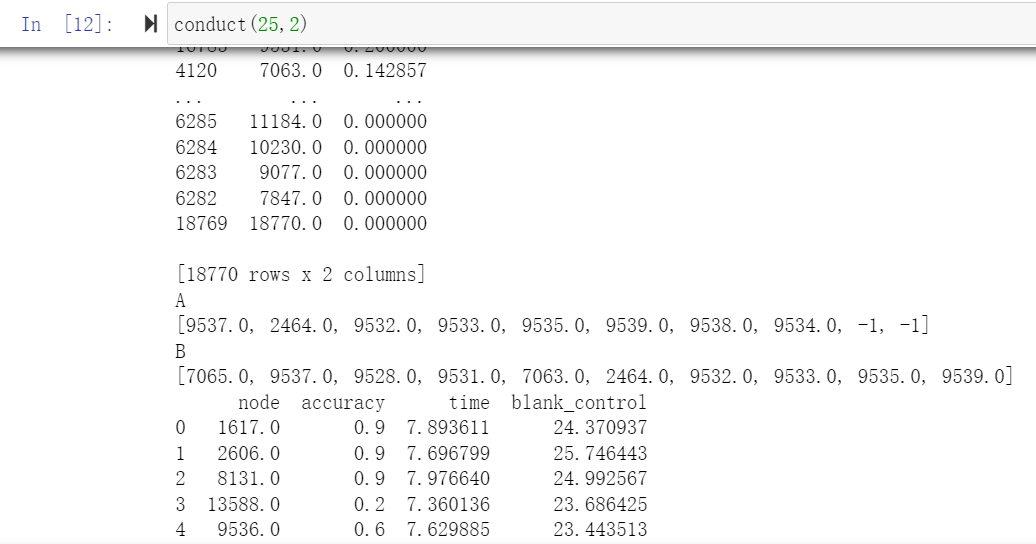
从结果来看，四个实验中的运行时间貌似变化不大。随着r值的增加，每次分桶需要比较的哈希签名向量维数增加，但进入候选集的节点变少，减少了similarest\_ten()函数中的循环次数。但随着r越来越大，正确率下降幅度很大，空间消耗也越来越大。

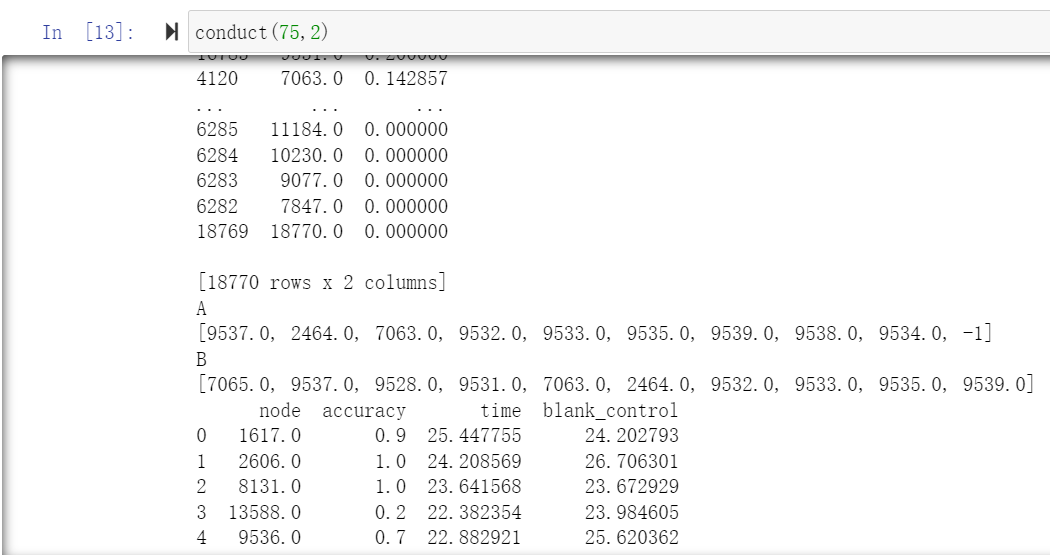
4.1.4 减少行条数量

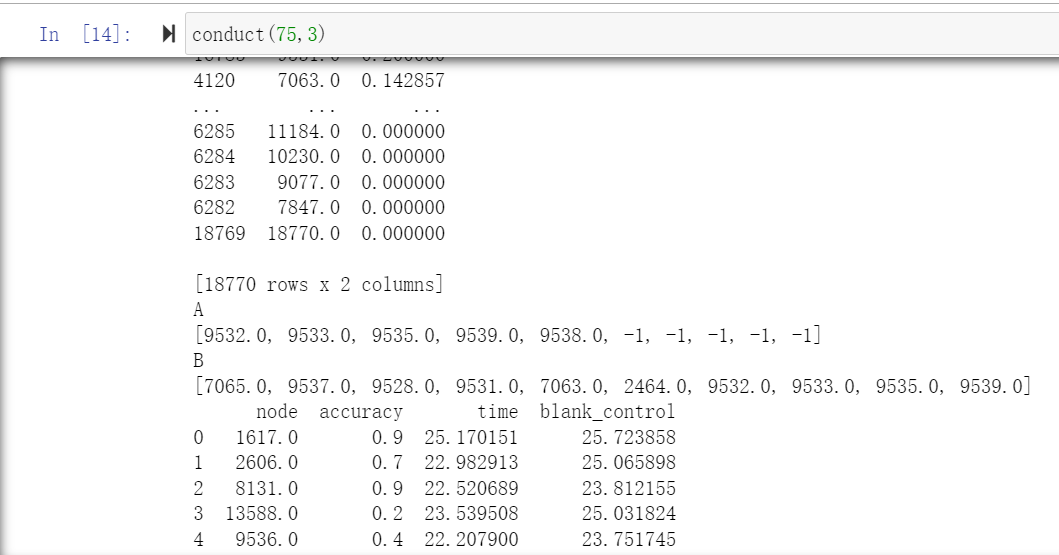
但是，即使正确率得到了保证，如果查询的时间反而远远大于暴力搜索的时间，那这个算法就没有什么意义了。注意到课件上只要两个集合的Jaccard相似度达到0.8，哪怕在行条数为20，每个行条有5行的情况下，这两个集合被哈希到同一个桶中的概率都大于99%。所以，我决定把行条数缩小到“几十”这个数量级，再做一次实验。

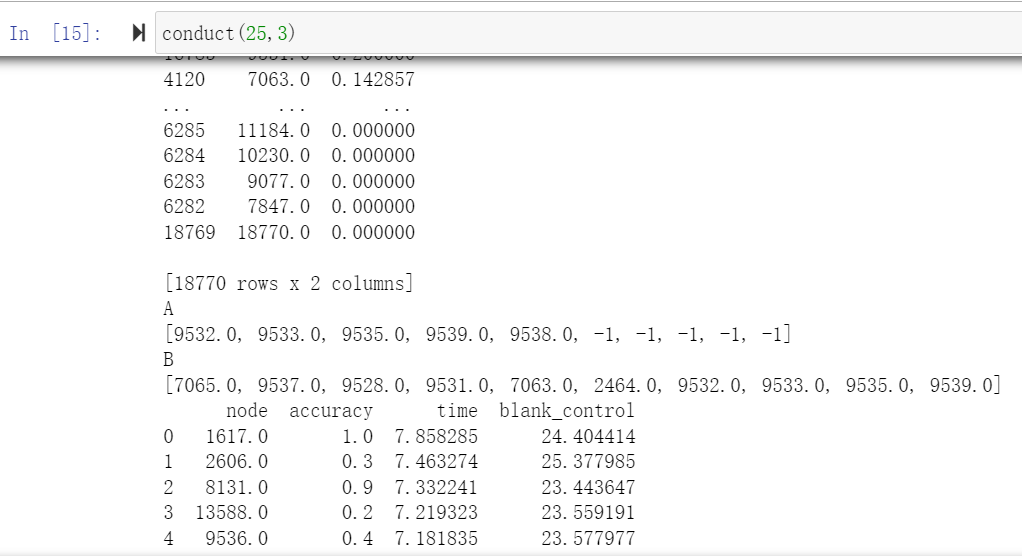


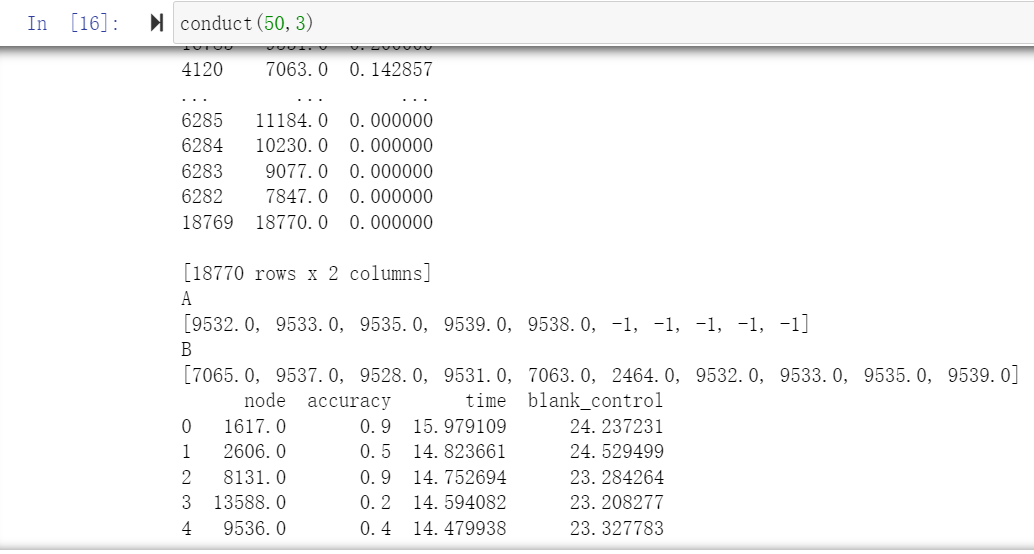


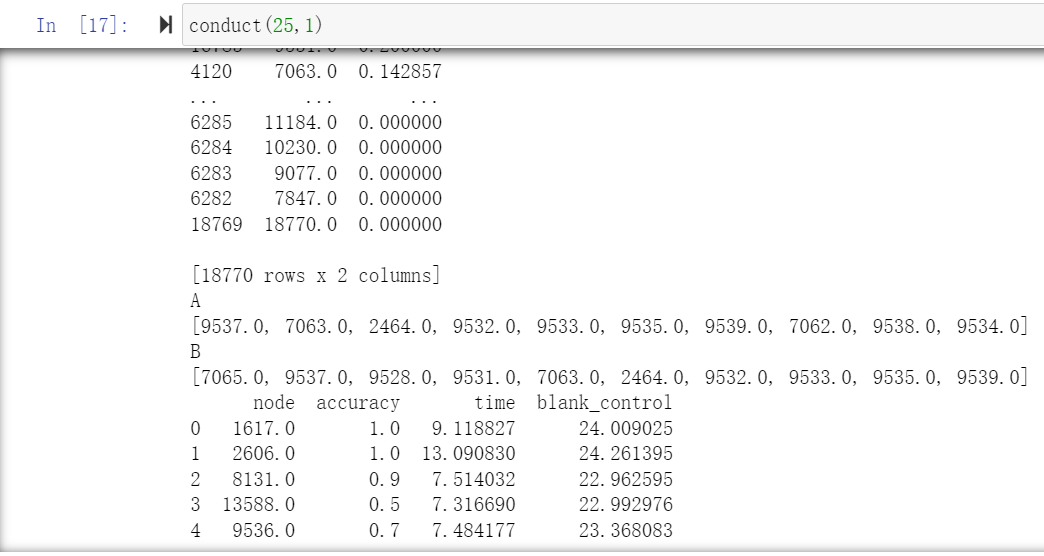


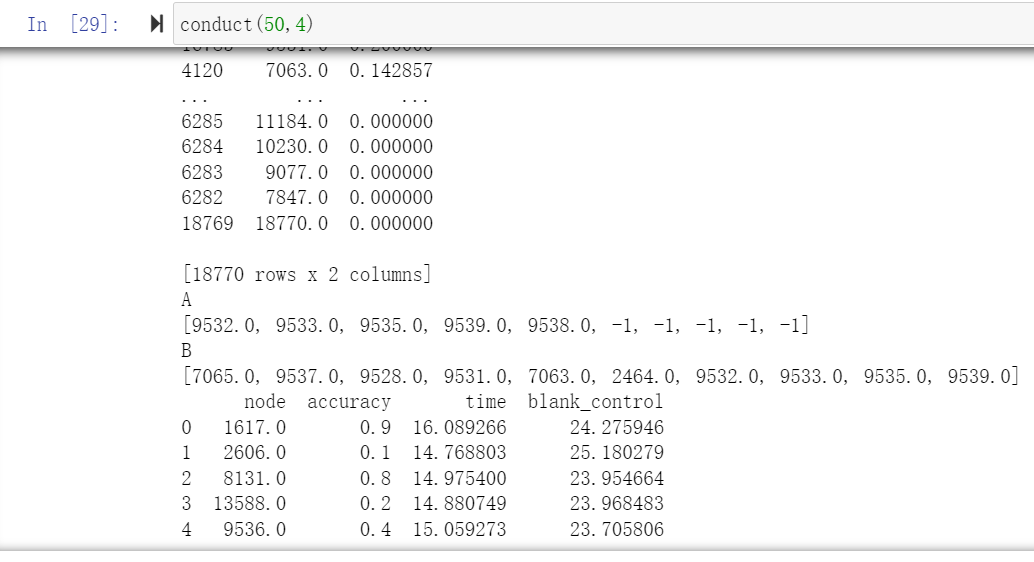






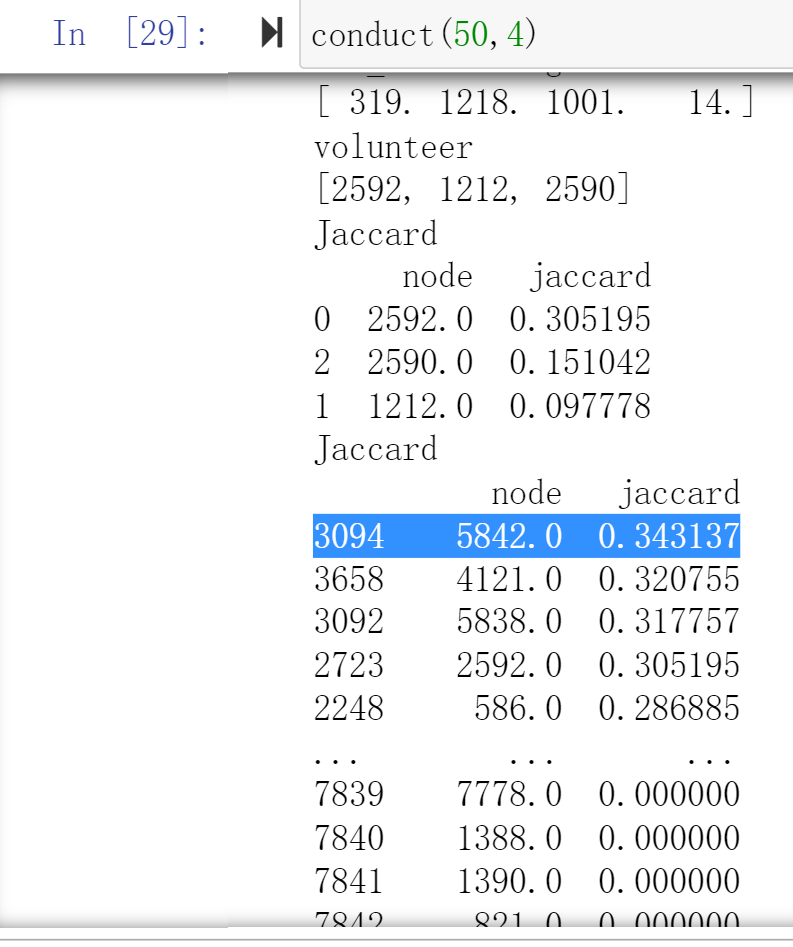






我们可以发现，只要行条数小于75，查询时间都小于暴力搜索的时间，对于大部分节点来说查询的正确率也都能得到保证。

但是，节点2606和13588的正确率无论在何种情况下都低于其他节点。以节点2606的邻居集合为例，所有集合中和2606的邻居集合Jaccard相似度最高的也只有0.34，这个数值与其他集合与2606的邻居集合被分到同一个桶中的概率相等。所以，其它集合与2606的邻居集合被哈希到同一个桶中的概率较低，导致进入备选集的节点数量少于十个，最终导致这个节点的准确率很低。13588的邻居集合也是同理。



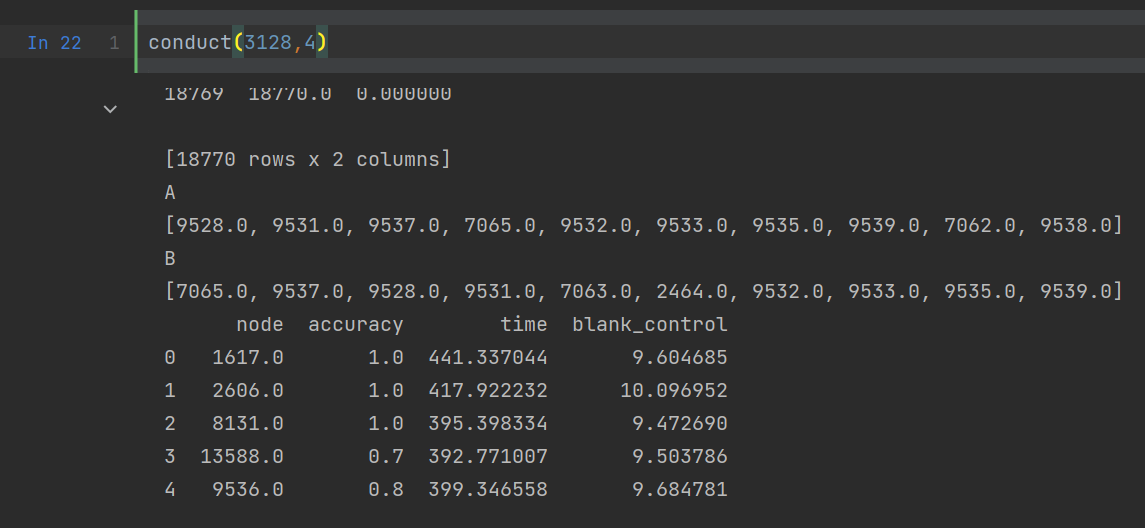
**4.2多探寻局部敏感哈希的运行结果**

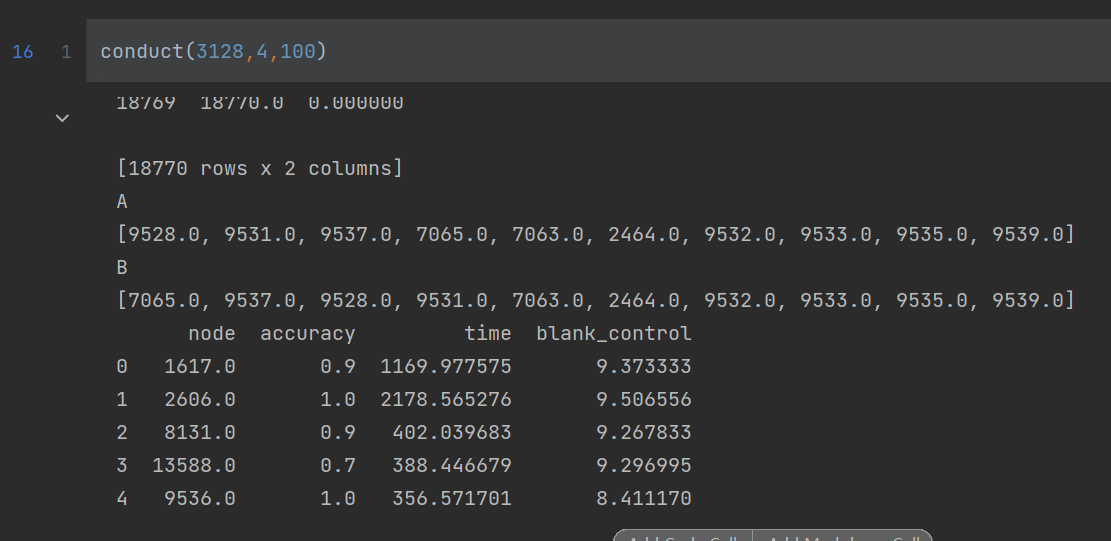
接着我们把普通LSH和多探寻LSH的正确率和运行时间作个比较，看看在空间消耗相等的情况下多探寻LSH是否能得到更好的结果。当然，多探寻LSH的conduct()函数要略作改动，因为它多了一个参数bound。

def conduct(b,r,bound):  
 global test\_node  
 global Verte  
 df = pd.DataFrame(columns=['node','accuracy','time','blank\_control'],dtype=float)  
 for i in range (5):  
 print("-------------------")  
 print("i")  
 print(i)  
 t\_a=time.perf\_counter()  
 A = LSH(test\_node[i],b,r,bound)  
 t\_b=time.perf\_counter()  
 Verte\_without\_i = list(Characteristic\_Graph.nodes)  
 if test\_node[i] in Verte\_without\_i:  
 Verte\_without\_i.remove(test\_node[i])  
 t\_c=time.perf\_counter()  
 B = similarest\_ten(test\_node[i],Verte\_without\_i)  
 t\_d=time.perf\_counter()  
 accuracy = len(list(set(A) & set(B)))/len(B)  
 run\_time = t\_b-t\_a  
 compare\_time = t\_d-t\_c  
 df1 = pd.DataFrame([[test\_node[i],accuracy,run\_time,compare\_time]],columns=['node','accuracy','time','blank\_control'],dtype=float)  
 df = pd.concat([df,df1], ignore\_index=True)  
 print("A")  
 print(A)  
 print("B")  
 print(B)  
 print(df)

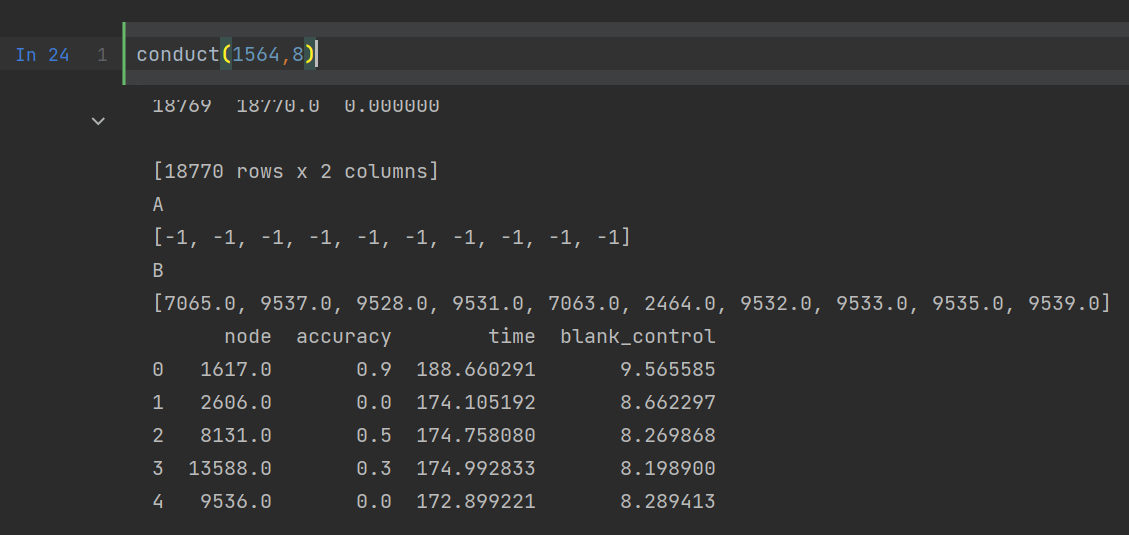
一开始我们尝试着把bound放宽到1和10，但根普通的LSH相比优化作用不够显著。最后，我们把bound放宽到了100，多探寻LSH和普通LSH的结果有了明显的不同。

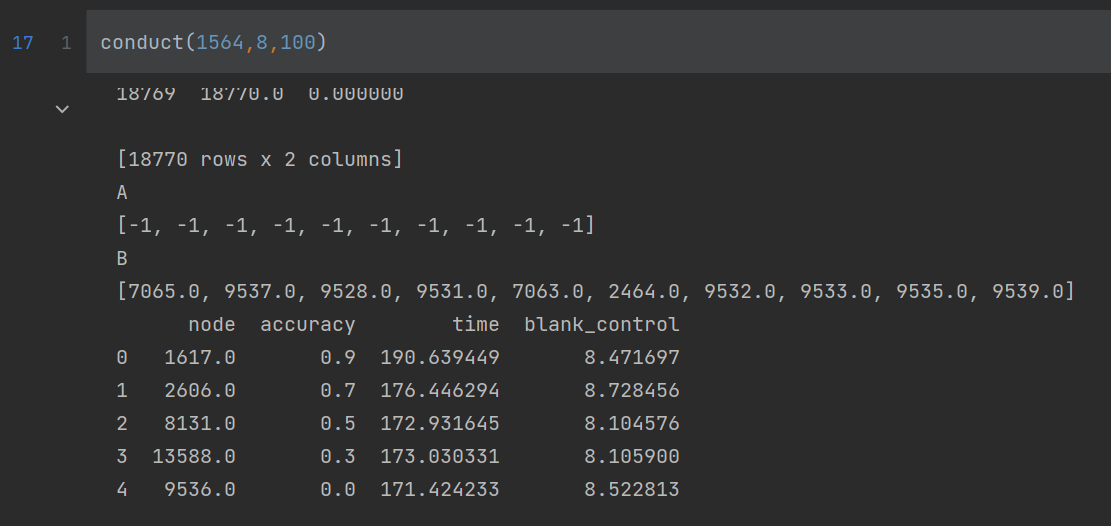
* (3128,4)



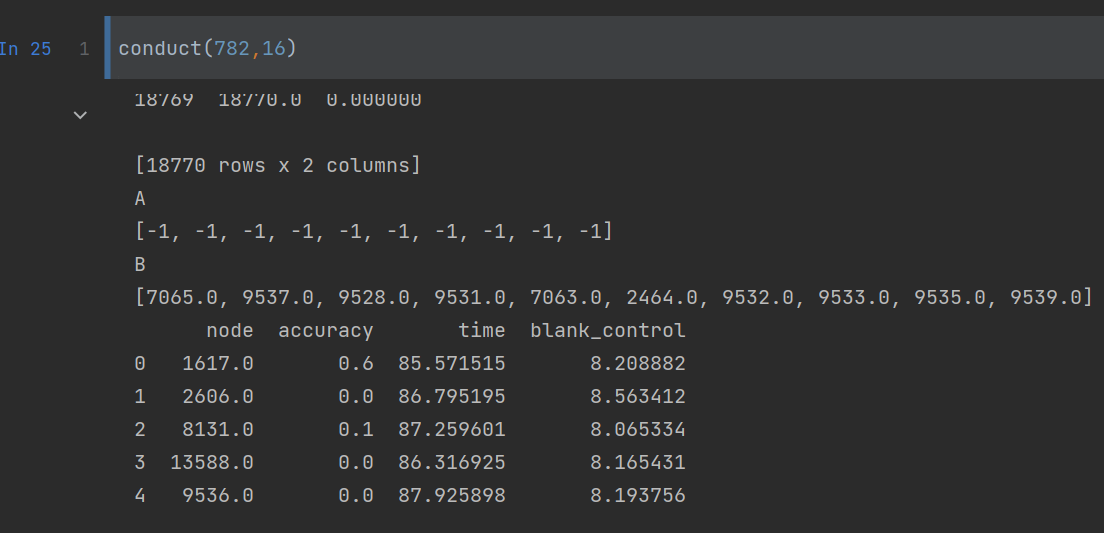


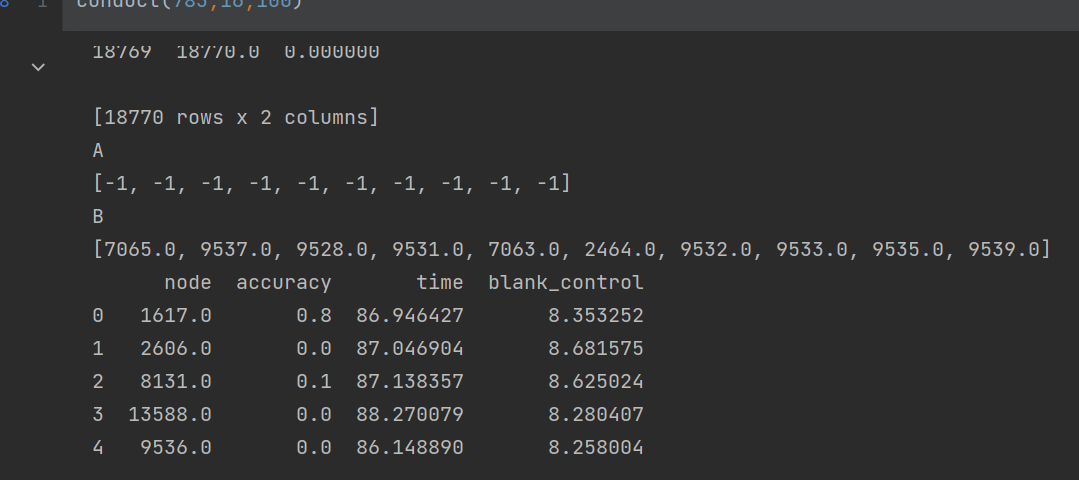
* (1564,8)





* (782,16)

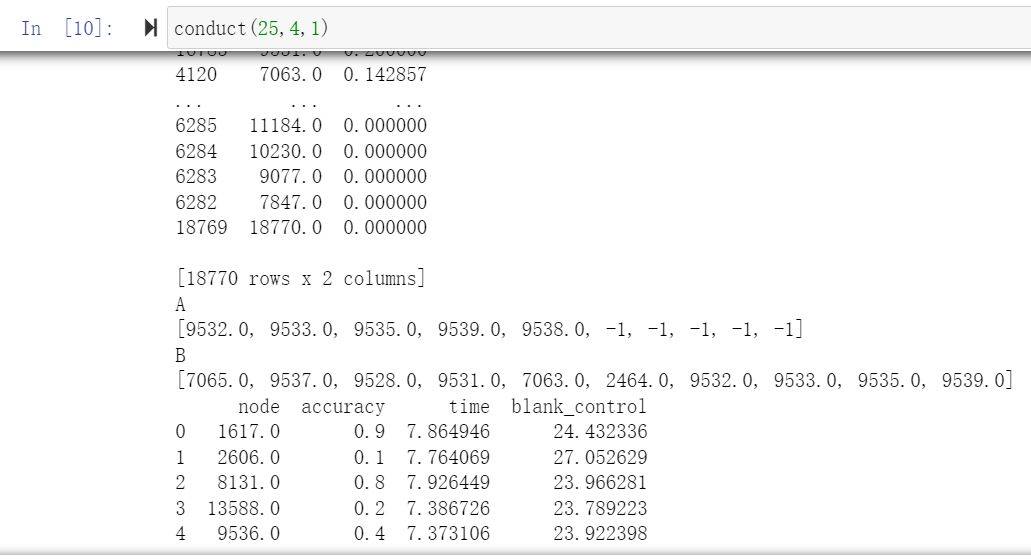


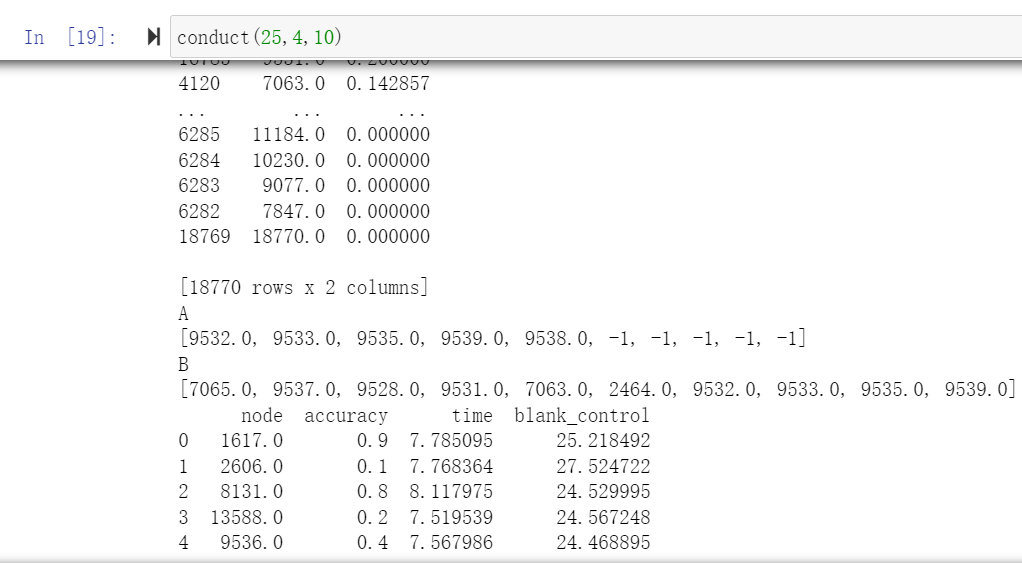


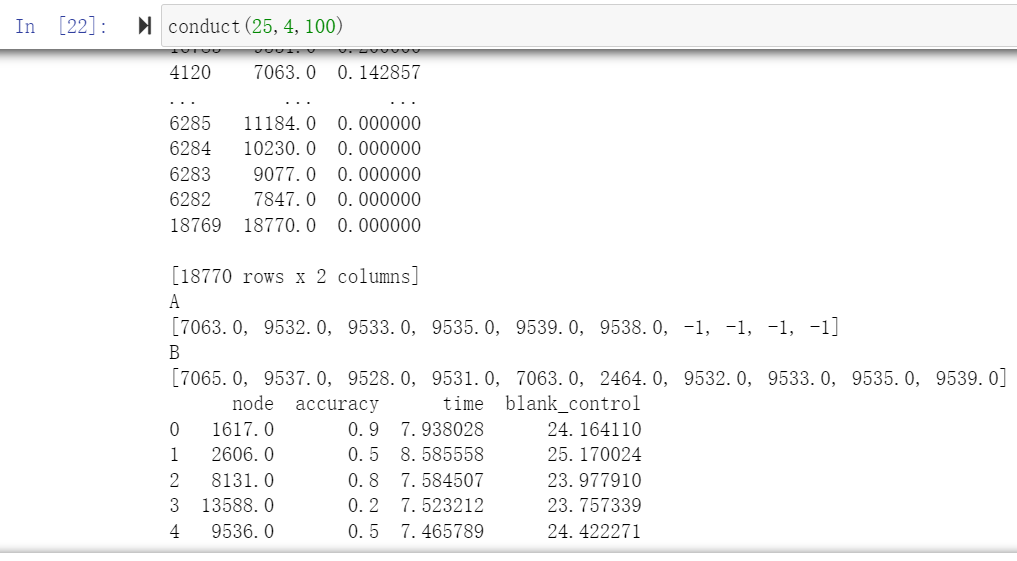
可以看到，当bound=100，在哈希签名矩阵分割情况相同的条件下，三组实验的正确率都有了不同程度的提升。但是，多探寻LSH的运行时间相比普通LSH要稍微高一些，因为加入候选集合的节点更多了，similarest\_ten()中的遍历次数也会变多，但是时间的增加不是特别多。

让我们在降低行条数目以后再做一次实验：

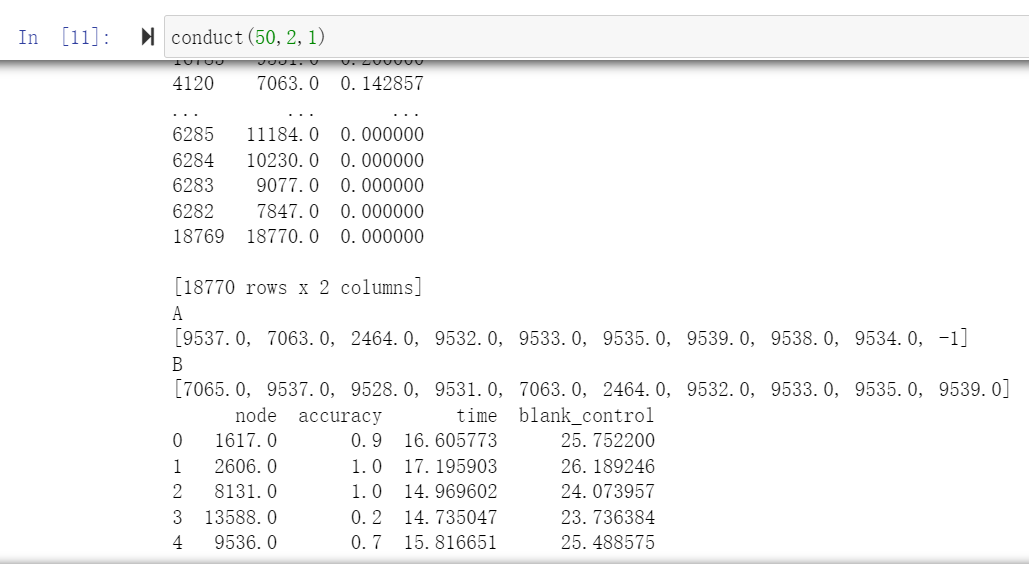
* (25,4)

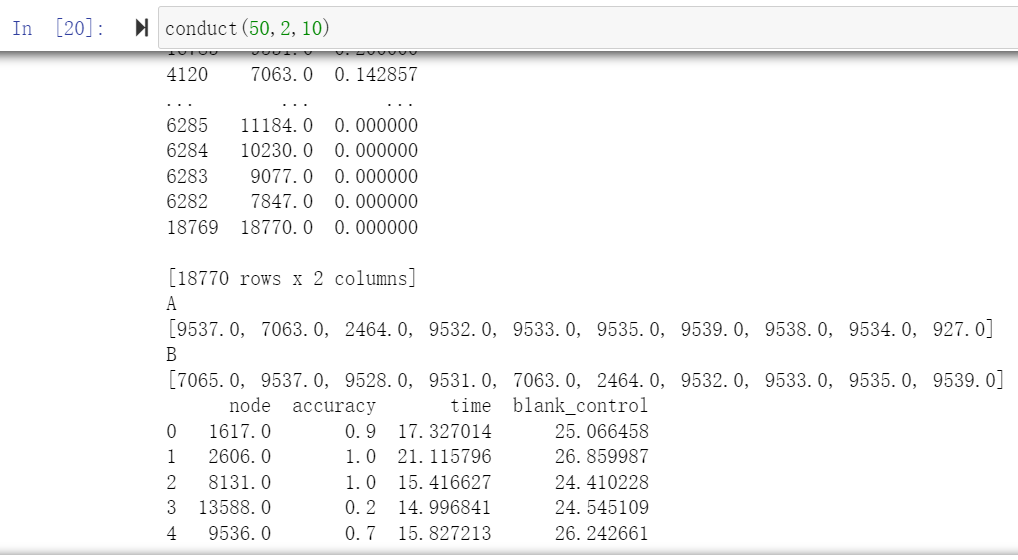


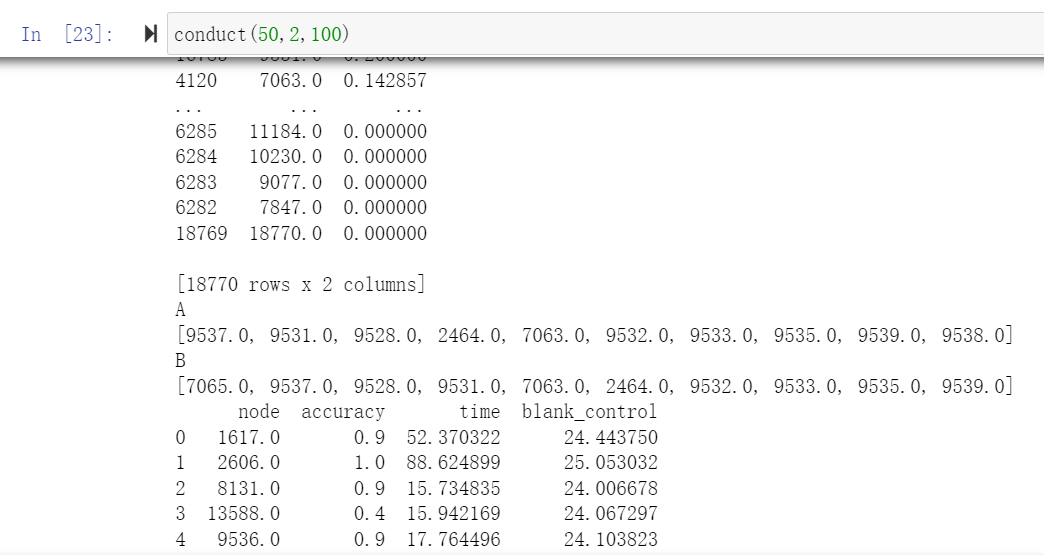




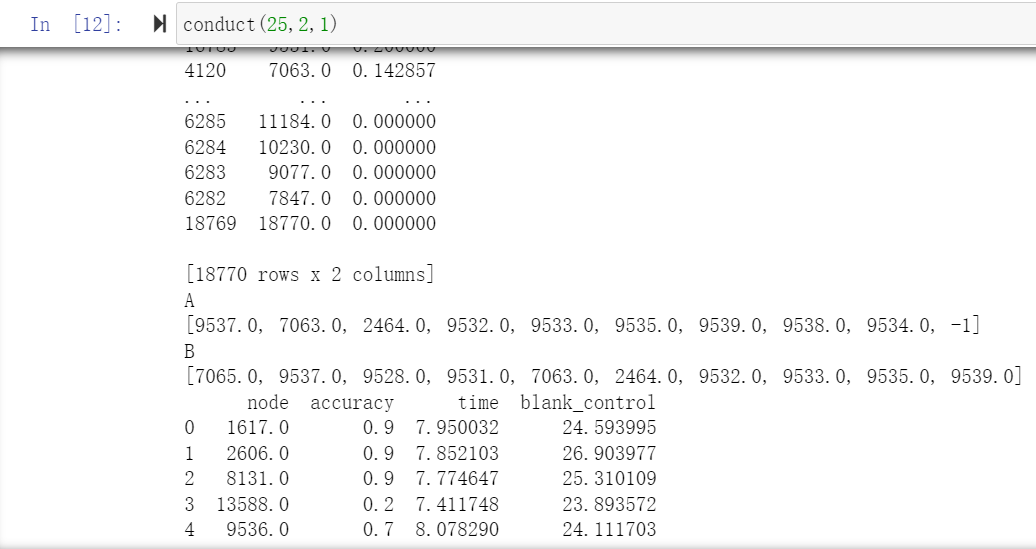
* (50,2)

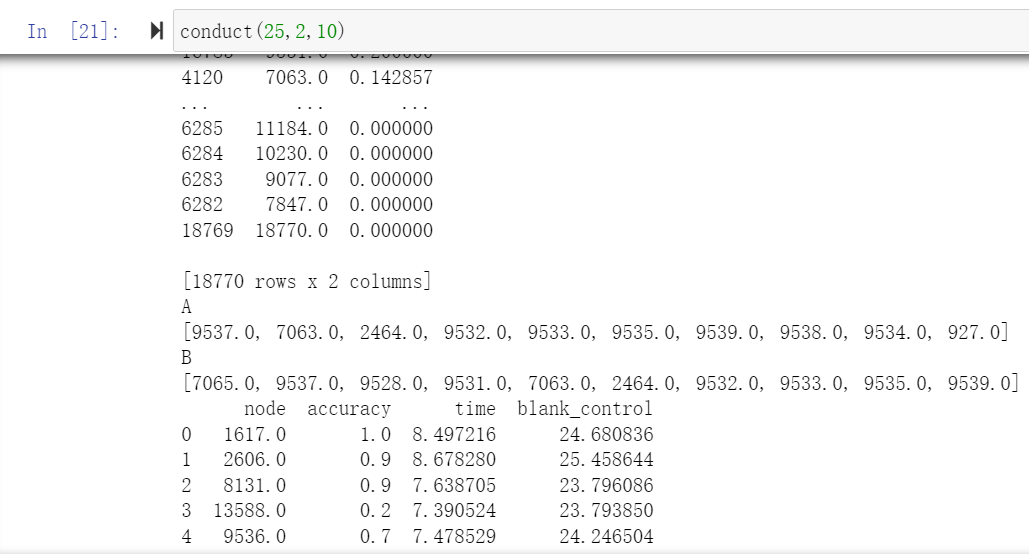


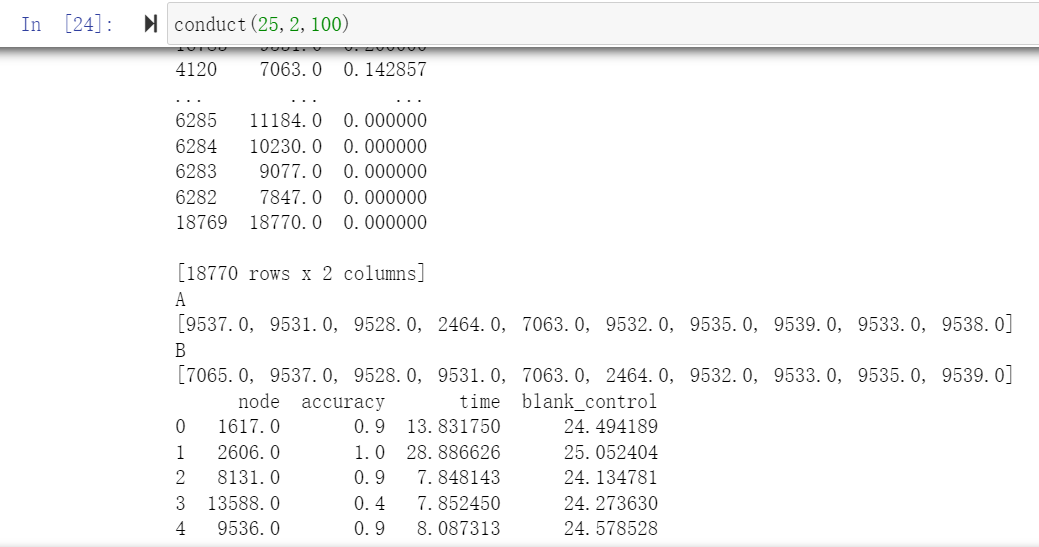




* (25,2)







我们发现，多探寻局部敏感哈希在行条数比较小的时候也能在时间成本没有太大变化的前提下提升正确率。值得一提的是，由于哈希签名向量的长度变短了，我们在放宽的bound比较小的情况下就能看到正确率的提升。比如，对节点9536来说，在行条数为25，每个行条有2行的前提下，只要把bound从0调到1，我们就能把正确率从0.6提升到0.7。

1. 结论（对使用的方法可能存在的不足进行分析，以及未来可能的研究方向进行讨论）

**5.1结论**

* 实践表明，行条数b对运行时间的影响最大，运行时间会随着b的增加显著上升；
* b对正确率的影响不大，所以为了在保证正确率的情况下缩短查询时间，我们应该把b调整到75以下；
* 每个行条的行数r对正确率的影响很大，正确率会随着r的上升急剧下降，但r的变化对运行时间影响不大；
* 所以，可以适当控制b的大小并不要把r设置得太大，就能在相对较短的时间内得到正确率较高的搜索结果，这同时还减少了算法的空间开销；
* 如果使用多探寻LSH，在b和r相同的情况下，还可以用略微增加一点运行时间的代价在r比较大的时候得到更高的正确率，这在哈希签名举着较大而不得不控制b的大小的时候非常有用。

**5.2可能存在的不足**

* 本次实验中，最大的遗憾就是LSH算法把大把的时间用在了分桶上，而分桶过程中还有着不少的重复计算；
* 此外，我们没有对哈希函数本身进行优化，使得相似的集合被哈希到同一个桶中的概率更大。

**5.3未来可能的研究方向**

* 可以对分桶的过程进行优化，从而减少循环次数、避免重复计算；
* 目前有人已经提出了用Hadamard变换的方法来优化哈希函数[[2]](#footnote-2)，但这种方法只能用于集合数量为偶数的情况，未来还可以研究对所有情况都适用的哈希函数优化方案。

1. Wei Dong. Zhe Wang. William Josephson. Moses Charikar. Kai Li. Modeling LSH for Performance Tuning [↑](#footnote-ref-1)
2. Anirban Dasgupta.Ravi Kumar.Tamás Sarlós.Fast Locality-Sensitive Hashing [↑](#footnote-ref-2)