《数据科学与工程算法》项目报告LSH

摘要

在高维空间中寻找最近的邻居是很多应用中的一个重要问题，例如多媒体检索，机器学习，生物和地质科学等。为了避免维数灾难，一个解决方法是寻找近似解，即近似版本的目标最近邻问题，也叫c-approximate Nearest Neighbor search(ANN)，而Locality Sensitive Hashing 是最流行的近似最近邻（ANN）解决方案之一，LSH通过使用随机哈希函数将高维数据映射到低维，将数据点分配到哈希桶中。

本报告的数据集是表示研究人员之间共同作者关系的无向图，需构建一个LSH方案，以进行相似性搜索：对于任何查询节点，应该找到与查询节点邻居集合的Jaccard相似度得分最高的前10个节点（不包括查询节点本身）。并对相似性搜索的性能，如准确性、索引时间、查询时间和空间使用率等进行评估。

通过观察实验结果发现：对本实验数据集，选取b=50,r=1可以得到一个很好的效果：其总共用时仅仅5.827878s , 就已经达到了0.968421053的准确率 。

Abstract

Finding nearest neighbors in high-dimensional space is an important problem in many applications, such as multimedia retrieval, machine learning, biological and geological sciences, etc. In order to avoid the curse of dimensionality, one solution is to find an approximate solution, that is, an approximate version of the target nearest neighbor problem, also called c-approximate Nearest Neighbor search (ANN), and Locality Sensitive Hashing is the most popular approximate nearest neighbor (ANN) solution In one of the schemes, LSH maps high-dimensional data to low-dimensional data by using a random hash function, and assigns data points to hash buckets.

The data set of this report is an undirected graph representing the co-author relationship between researchers. It is necessary to construct an LSH scheme for similarity search: for any query node, the Jaccard similarity score with the highest Jaccard similarity score to the query node's neighbor set should be found. The first 10 nodes (excluding the query node itself). And evaluate the performance of similarity search, such as accuracy, indexing time, query time and space usage, etc.

By observing the experimental results, it is found that for this experimental data set, choosing b=50 and r=1 can get a very good effect: the total time spent is only 5.827878s, and the accuracy rate of 0.968421053 has been achieved.

一、 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）

在高维空间中寻找最近的邻居是很多应用中的一个重要问题，例如多媒体检索，机器学习，生物和地质科学等。对于低维（<10）数据，流行的是基于树的索引结构，如 KD-tree 、SR-tree等是有效的，但对于更高的维数，这些索引结构遭受众所周知的问题，维数灾难。一个解决方案就是寻找近似结果。在许多应用中不需要100%的准确性，搜索足够接近的结果比搜索精确结果快得多 。近似解决方案以更快的性能换取准确性。近似版本的目标最近邻问题，也叫c-approximate Nearest Neighbor search。

Locality Sensitive Hashing 是最流行的近似最近邻（ANN）解决方案之一。 Locality Sensitive Hashing (LSH)通过使用随机哈希函数将高维数据映射到低维，数据点被分配到每个哈希桶中。这种方法背后的想法是原始高维空间中更接近的数据点将被以高概率映射到低维投影空间中的相同哈希桶。自从它首次被引入以来，已经提出了许多 Locality Sensitive Hashing 的变体，它们主要侧重于提高给定查询的搜索准确性和/或搜索性能。性能/查询的准确性权衡取决于用户提供的成功保证率。

二、 问题定义（提供问题定义的语言描述与数学形式）

数据集是表示研究人员之间共同作者关系的无向图，共有18771个人，之间的共同作者关系共有198050条，可以据此建立图模型，研究人员为图的节点，研究人员间共同作者关系为图的边。

$$G = G(V, E)$$
$$|V| = 18771, |E| = 198050$$

每一个**节点** v_i **也称为一个对象，一列(col)**，对应一个**邻居集合向量A[i]**，这个向量记录了v_i与v_1~v_{18771}之间有无边，若v_i与v_j之间有边，则A[i] [j]=1,否则A[i] [j]=0。

现在对于任何查询节点，应该找到与查询节点邻居集合的Jaccard相似度得分最高的前10个节点（不包括查询节点本身）,即需要查找与向量A[i] Jaccard相似度最高的前十个向量A[j] (i != j)

采用的方法为LSH：会生成n个**签名向量sig_vector**，然后将其组成**签名矩阵sig_matrix。**

再对其进行分桶(genHashBucket)，把每一个节点的最小哈希签名分成b段，每段长度为r(故n=b*r)，将长度为r的签名段根据其数据本身与所在层数映射到不同的桶中。一个节点所在桶中的所有元素被称为这个节点的**候选集**

然后对于一个查询节点query，找到其候选集，将其候选集中元素根据其与query的jaccard相似度大小进行从大到小排序，取其前10，即为估计的与查询节点邻居集合的Jaccard相似度得分最高的前10个节点。

三、 方法（问题解决步骤和实现细节）

数据预处理

原始数据集如下：

	vertex	neighbour
0	1	2
1	1	3
2	1	4
3	1	5
4	1	6
...	...	...
198045	18765	18766
198046	18765	18767
198047	18766	18767
198048	18768	18769
198049	18770	18771

198050 rows × 2 columns

每条边仅在原始数据集中出现一次，即有了1，2, 原始数据中便不会记录2，1，但是2，1间也有边，因为是无向图，其邻接矩阵必然是对称矩阵。自然而然地，我采取用邻接矩阵来存储这种数据。

每一个节点 v_i 为一个对象，对应一个属性向量A[i]，这个向量记录了v_i与$v_1 - v_{18771}$之间有无边，若v_i与v_j之间有边，则A[i] [j]=1,否则A[i] [j]=0，注意因为是对称矩阵，所以如有A[i] [j]=1，则还要给A[j] [i]赋值为1。按照这种方法构造邻接矩阵。

```
original_matrix = np.zeros((18771, 18771)) #注意第0列对应vertex 1
for i in range(198050):
    original_matrix[data.iloc[i]['vertex']-1][data.iloc[i]['neighbour']-1]+=1
    original_matrix[data.iloc[i]['neighbour']-1][data.iloc[i]['vertex']-1]+=1
```

得到原始矩阵original_matrix如下：

```
array([[0., 1., 1., ..., 0., 0., 0.],
       [1., 0., 0., ..., 0., 0., 0.],
       [1., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 1.],
       [0., 0., 0., ..., 0., 1., 0.]])
```

可以看到这个矩阵中0的数目多于1的数目，是个稀疏矩阵(邻接矩阵一般都是稀疏矩阵)

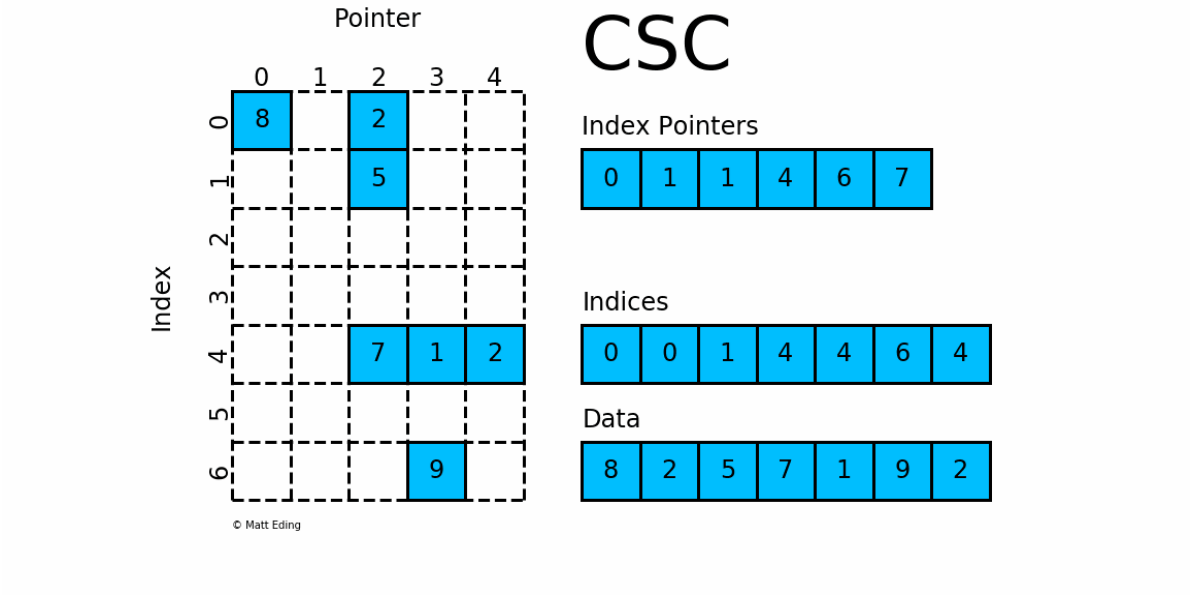
稀疏矩阵按列压缩（优化

original_matrix为稀疏矩阵，在后续的运算中回造成许多不必要的遍历，之前我用original_matrix直接去生成签名，生成一个签名向量用时2min , 这个速度令人难以接受，故舍弃这种方案。

所以需要从遍历邻接矩阵转变为遍历邻接表，这样就不用遍历为0的元素了，而生成最小哈希签名里用到的是第一个不为0的元素对应的行号，所以是不用去遍历为0元素的。

```
csc= csc_matrix(original_matrix)
index_pointer=csc.indptr
indices=csc.indices
```

csc的索引方法[1]:



可以看到csc类似于邻接表，v_i的邻居即为indices[index_pointer[i]:index_pointer[i+1]]，而这里的data向量是一个全为1的向量。

构造哈希签名矩阵(1)

生成哈希签名向量

思路:

维护一个长度为节点总数的数据，作为签名向量result，初始化为[-1,-1,-1,...-1]

然后随机选一行randomRow，对于在行号为randomRow处为1的列号j

如果result[j]==-1，说明这一列还没有遇到最小哈希签名，则result[j]=randomRow，并且count++

当count=节点总数时，一个签名向量找齐了，退出函数

(注意此处用生成不重合的、在范围 [0,节点总数) 里的randomRow来得到一个行号的随机排列，以此来替代哈希函数簇

此处是对特定行进行索引，所以采用csr

```
#acquire signature vector
import random

def genSigv(csr):
    #generate signature vector
    #parameter matrix: a ndarray var
    #return a signature vector(a list)
    indptr = csr.indptr
    indices = csr.indices
    rows=csr.shape[0] #行数
    cols=csr.shape[1] #列数
    #initialize the signature vector as [-1,-1...-1]
```

```
    result = [-1 for i in range(cols)]
    #rowset为行号的集合(从1开始)
    rowset= [i for i in range(rows)] #行号就从0开始了

    #选定一行处理后便把这行从rowset里面移除，直到rowset为空
    #其中选定某行的顺序就等价于一个哈希函数(都是映射)
    count = 0
    while len(rowset)>0:
        #随机选一行
        randomRow = random.choice(rowset)
        for j in indices[indptr[randomRow]:indptr[randomRow+1]]:
                if result[j]==-1:
                    result[j]=randomRow
                    count += 1
        if count == cols: #如果提前找齐了签名向量就退出
            break
        rowset.remove(randomRow)

    # return a list
    return result
```

缺点

这样找最小哈希签名向量，每一个节点的最小哈希签名找到了之后仍然需要等其它节点找到其最小哈希签名，并且已经找到最小哈希签名的节点仍然要继续被遍历，这样增加了时间开销。

计算一个最小哈希签名向量的平均用时为1s。

把哈希签名向量组合成哈希签名矩阵

```
#生成签名矩阵
def sigMatrixGen(input_matrix, n):
    """
    generate the sig matrix
    :param input_matrix: naarray var
    :param n: the row number of sig matrix which we set
    :return sig matrix: ndarray var
    """

    result = []

    for i in range(n):
        sig = genSigv(input_matrix)
        result.append(sig)

    # return a ndarray
    return np.array(result)
```

构建哈希签名矩阵(2)（改进版

构造随机排列

由于此处哈希函数的作用是打乱行号，本质上是生成一个行号的随机排列；再加上生成哈希函数簇，再将行号用哈希函数进行打乱较为麻烦，所以此处用直接生成随机排列来代替。

```
def genRandomPermutationMatrix(index, numOfhash):  #generate a random sequence
    random_permutation_matrix = index
    index_to_shuffle = random_permutation_matrix.copy()
    new_index = [shuffle(index_to_shuffle, random_state=i +
np.random.randint(low=1, high=2**30)) for i in range(numOfhash)]
    random_permutation_matrix = np.c_[new_index].T
    return random_permutation_matrix
```

由于一次shuffle函数可以得到一个行号的排列，即一个permutation_vector,那么用n次shuffle便可得到n个permutation_vector，将这些permutation_vector拼接起来即可得到random_permutation_matrix.

构造签名矩阵

用上面那个random_permutation_matrix来得到最小哈希签名矩阵：

```
def genSigMatrix(csc, numOfhash, index_pointer, indices):
    index = np.arange(csc.shape[0]) #arange return a ndarray
    random_permutation_matrix = genRandomPermutationMatrix(index, numOfhash)
    col_num = csc.shape[1]
    random_col = random_permutation_matrix.shape[1] #randomly pick a col
    sig_matrix = np.zeros((random_col, col_num))

    for j in range(random_col):
        for i in range(col_num):
            sig_matrix[j][i] = min(random_permutation_matrix[:, j]
[indices[index_pointer[i]:index_pointer[i+1]]])
    return sig_matrix
```

此处的原理为：大循环j:构造第j个哈希签名sig_vector[j]

j是一个行号的随机排列的迭代器

小循环i:构造第j个哈希签名的第i位sig_vector[j] [i]

其中小循环中的关键代码解释：

```
sig_matrix[j][i] = min(random_permutation_matrix[:, j]
[indices[index_pointer[i]:index_pointer[i+1]]])
```

random_permutation_matrix[: , j]索引了第j个random_permutation_vector，而indices[index_pointer[i]:index_pointer[i+1]]对应了v_i的所有邻居节点的下标，则random_permutation_$matrix[:, j]$[indices[index_pointer[i]:index_pointer[i+1]]]就是vi的为1的位置对应的行号的集合，最小哈希签名便是这一集合中最小的那一个，所以再取min，即可得到vi的第j个最小哈希签名，即sig_matrix[j] [i]。

优点(较构造哈希签名矩阵(1))

解决了哈希签名矩阵(1)的缺点，即一个一个地算出每个节点的最小哈希签名，而不是所有节点一起遍历、直到找齐了哈希签名向量再退出。这样减少了时间开销。

构造哈希签名矩阵(1)计算一个最小哈希签名向量的平均时间为1s，而改进版构造哈希签名矩阵(2)计算一个最小哈希签名向量的平均用时为0.06416342s

分桶

大桶buckets是一个字典,其key为tag，value为具有相同tag的col组成的list，即buckets={tag:col_list}, buckets里的每一个元素都是一个小桶bucket，用于存储tag相同的col

```
{'e7ebc54f099a4d76218f10d12e15c6c3': [0],
 '406a7716b60994c187171054e7d7e064': [1, 31],
 'd90c90ea0f65479d9d6db33e98aeb9e5': [2],
 'a9f4e06d597aefde5f24cb4d17ae2751': [3],
 '38fdba08d00fddbd8d99b21026673e23': [4],
 '439b0c4281f7d65f635a3a3ea0e94a77': [5],
 '952efcccadae14076ac55e3cc44379cd': [6],
 '886cccba632f1f785ac09d1788374133': [7],
 '9b087b7fb5a60415001ca29426b9c0b3': [8],
 '90b807d73a70d88d547fcd3eb26c098c': [9],
 'd9d61af0fa973750812d4a96abfa516b': [10],
 'b6f6223f968e986a3cc82ebd5e41b4e6': [11],
 '9119808466b844c4985925641c59a042': [12],
 'dc6de4d9bb515ae5d933e34827cc89a3': [13],
 '1d6b1d97440oaa59419oc80ac5f44393': [14],
 'a59108f783c2fb7847a837e9af184cd3': [15],
 '589145127a71c469f2fa4e9a807288ff': [16],
 '84a43abd57431e007dbc43720ffe5e9c': [17, 638, 1415, 4004, 8617],
...
 'b07d60c0185a90666b3030d3bcef3702': [1143],
 'ce5308a7522a6223801c60bdc88762e6': [1144],
 '70c61f5002775c46f216d1726a88b596': [1145],
 '0f665f356efc45640f08662ea4fcbf04': [1146],
 ...}
```

把sig_matrix分为b个长度为r的段(所以要生成b*r个签名向量), 每一个长度为r的段根据其本身的数据与所在层数(count), 采用md5加密算法生成一个特定的tag

对于每一个tag，如果tag没在buckets.keys中，则创建一个key为tag的桶

如果该tag已经有对应的小桶bucket了，即tag在buckets.keys中，则把该tag对应的col加入buckets[tag]

另外维护一个index字典，为每一个col创建一个它所有的tag组成的集合，方便后续查询某一个col都在哪些小桶中

```
import hashlib

def LSH(sig_matrix, r, b):
    rownum = sig_matrix.shape[0]
    colnum = sig_matrix.shape[1]
    begin , end = 0, r
```

```
    count = 0

    index = {}
    buckets={}
    while end <= rownum:
        count += 1

        for col in range(colnum):
            hash_obj = hashlib.md5()
            band = sig_matrix[begin: end, col].tobytes() +
np.array([count]).tobytes()
            hash_obj.upda，te(band)
            tag = hash_obj.hexdigest()
            if col not in index:
                index[col] = set()
                index[col].add(tag)
            else:
                index[col].add(tag)

            if tag in bucket:
                buckets[tag].add(col)
            else:
                buckets[tag] = set()
                buckets[tag].add(col)
        begin += r
        end += r

    return bucket, index
```

查询

查询节点(查询的列)为query，

searchsimilar()函数: 输入：query 输出：query的相似候选集，即跟query在同一个桶里的col组成的集合

遍历query对应的所有的tag，对于每一个小桶buckets[tag]，把其中的col加入query的相似候选集

然后对相似候选集中的col按照其与query的jaccard相似度由大到小排列，返回排名前十的col

```
from collections import Counter #用于构造一个拿来排序的集合
def searchsimilar(csc,index,hashBucks,query):
    result=set()  #返回跟query在一个桶里的对象(用set保证了不重复)
    for tag in index[query]:
        if query in hashBucks[tag]:
            for i in hashBucks[tag]:
                result.add(i)
    result.remove(query) #去除查询对象自己

    jaccard_dict={}
    for i in result:
        jaccard_dict[i]=jaccard(csc,query,i)
    sorted_list=sorted(jaccard_dict.items(),key=lambda x:-x[1])
    result.clear()
    count=0
    while(count<len(sorted_list)):
```

```
        if(count==10):
            break
        result.add(sorted_list[count][0])
        count+=1
    return result
```

计算jaccard相似度

```
def jaccard(csc,col1,col2):
    indptr = csc.indptr
    indices = csc.indices
    a=set()
    b=set()
    for i in indices[indptr[col1]:indptr[col1+1]]:
        a.add(i)
    for j in indices[indptr[col2]:indptr[col2+1]]:
        b.add(j)
    return len(set(a).intersection(set(b))) / len(set(a).union(set(b)))
```

计算col1与col2的相似度，用其交集元素个数/并集元素个数即为jaccard(col1,col2)

举一例，query=132,b=400,r=5

按照给定数据集生成sig_matrix与hashBuckets之后，用LSH查询query=132的相似度排名前十的col

但是与其在同一个桶里的只有9个，所以只输出了9个col

col	jaccard(query,col)
112	0.4095940959409594
65	0.39603960396039606
119	0.3282208588957055
125	0.3151862464183381
147	0.31213872832369943
371	0.3105263157894737
374	0.2702702702702703
395	0.17333333333333334
362	0.13392857142857142

query=132的候选集：{65, 362, 395, 112, 147, 371, 374, 119, 125}

四、实验结果（验证提出方法的有效性和高效性）

最终采用csc与构建哈希签名矩阵(2)，下面的测试结果均为这个最终最好的版本的

测试指标

时间开销

用python time库里面的time()函数进行测量，单位默认为s。

由此可以测出**生成签名矩阵时间:genSigMatrix, 分桶时间:gen_hashBuckets, 查询时间:similarSearch**

准确度测量

候选集：similarset

根据计算节点v_i与其它18770个节点的jaccard相似度再排序得到的**真实的jaccard相似度前10的节点集合：top10**

真实的jaccard相似度前k的集合: topk

定义两个准确度：

precision_top10= $len(top10 \bigcap similarset)/10$

precision_topk=$len(topk \bigcap similarset)/k$

可以看出precision_top10更符合这个问题的要求：“应该找到与查询节点邻居集合的jaccard相似度得分最高的前10个节点”。而precision_topk是为候选集中没有10个邻居节点的节点准备的，如果其候选集中邻居节点数小于10，那么其precision_top10准确度必然会被拉低，所以准备precision_topk来辅助观察其性能。

测试代码

测量时间开销代码：

```
time_start=time.time()
#代码块
time_end=time.time()
time_c= time_end - time_start   #运行所花时间
print(time_c)
```

测试查询准确度代码

因为有18771个节点，全部算一遍查询准确度取平均值时间复杂度是$O(n^2)$，用时过久，所以采用系统抽样的方法抽取97个节点来测量准确度。

```
#用系统抽样的方法抽取一些节点来求searchsimilar的平均用时，与平均precision

#precision_top10=len(similarset & real_top10_set)/10
#当找到的候选集similarset的大小k小于10时，
#precision_topk=len(similarset & real_topk_set)/k # k=len(similarset)

precision_top10,precision_topk=0,0
query_list=np.arange(0,NUM,200)
print("查询节点个数：",end='')
print(len(query_list))
sum_search_time=0
for query in query_list:
```

```
    time_start=time.time()
    similarset=searchsimilar(csc,index,hashBucks,query)
    time_end=time.time()
    sum_search_time+=time_end-time_start
    k=len(similarset)
    top10,topk=precision(csc,query,similarset,k)
    precision_top10+=top10
    precision_topk+=topk
print("average search time:",end='')
print(sum_search_time/len(query_list))
print("precison_top10:",end=' ')
print(precision_top10/len(query_list))
print("precision_topk:",end=' ')
print(precision_topk/len(query_list))
```

测试结果

测量b和r取不同值时的时间开销与准确度。

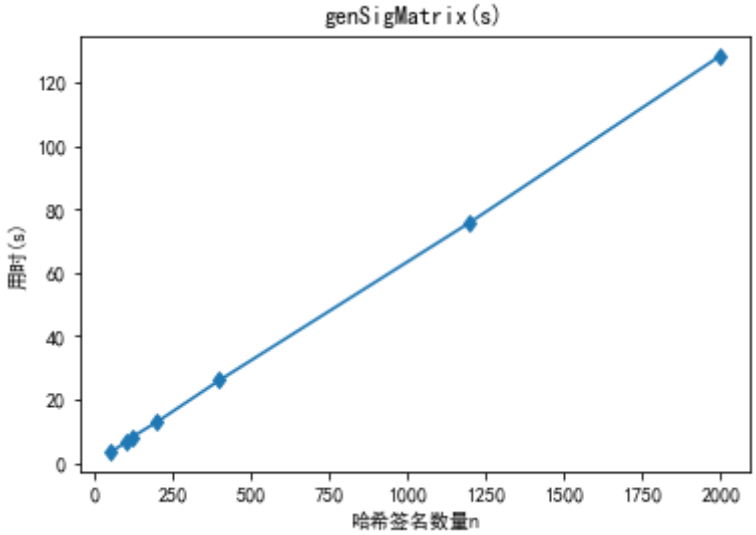
生成签名矩阵时间:genSigMatrix, 分桶时间:gen_hashBuckets, 单个节点平均查询时间:similarSearch
单位：s

	n	genSigMatrix(s)	gen_hashBuckets(s)	similarSearch(s)	precision_top10	precision_topk
b=50,r=1	50	3.258874178	2.563181639	0.005821938	0.968421053	0.978947368
b=50,r=2	100	6.487015247	3.47612071	0.000731935	0.780851064	0.867291456
b=100,r=1	100	6.534632683	5.233114719	0.007314271	0.929787234	0.962765957
b=120,r=1	120	8.097911835	6.447575092	0.007822737	0.936170213	0.965957447
b=60,r=2	120	8.097911835	3.203719139	0.001415861	0.846808511	0.918085106
b=40,r=3	120	8.097911835	2.461671591	0.000341943	0.55106383	0.829677474
b=30,r=4	120	8.097911835	1.654126883	0	0.372340426	0.62933975
b=24,r=5	120	8.097911835	1.599802494	0	0.244680851	0.450861196
b=20,r=6	120	8.097911835	1.448031425	0	0.208510638	0.379078014
b=15,r=8	120	8.097911835	1.148053169	0	0.121276596	0.261170213
b=100,r=2	200	12.9117105	7.067448139	0.000521721	0.861702128	0.917764269
b=200,r=1	200	13.22928143	10.37401056	0.009079593	0.931914894	0.965957447
b=200,r=2	400	26.06638837	13.34110022	0.002169229	0.891489362	0.941252955
b=400,r=3	1200	75.79899144	26.39799571	0.001241882	0.8	0.882877406
b=400,r=5	2000	128.3268397	29.52940798	0.00068767	0.476595745	0.823775751

生成哈希签名矩阵用时

n	genSigMatrix(s)
50	3.258874178
100	6.487015247
120	8.097911835
200	12.9117105
400	26.06638837

n	genSigMatrix(s)
1200	75.79899144
2000	128.3268397



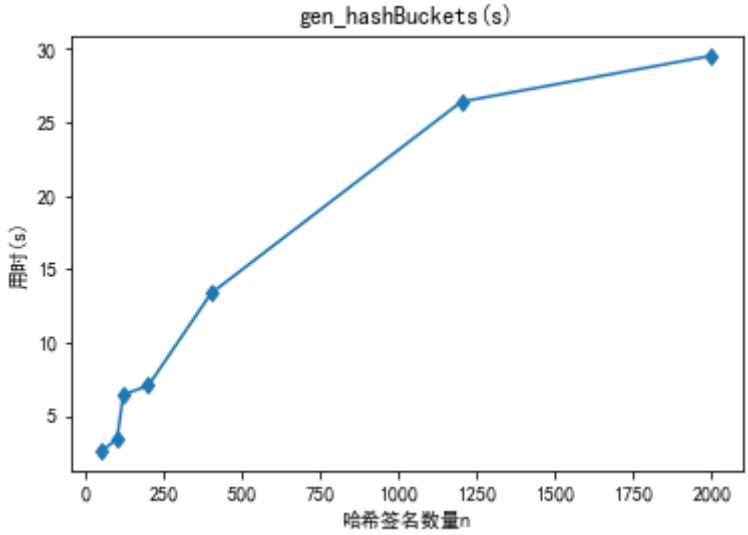
可以看出genSigMatrix用时与哈希签名数量n间是线性的

生成一个哈希签名用时都差不多是0.065s

genSigMatrix时间复杂度为O(n)

分桶用时

n	gen_hashBuckets(s)
50	2.563181639
100	3.47612071
120	6.447575092
200	7.067448139
400	13.34110022
1200	26.39799571
2000	29.52940798



分桶用时随着哈希签名数量递增而递增，增长形势大概为ln(n)

分桶用时不仅与n有关，还与r的选取有关，所以用时随n增长的形势不仅与n有关，还与r有关，需要综合分析

查询用时

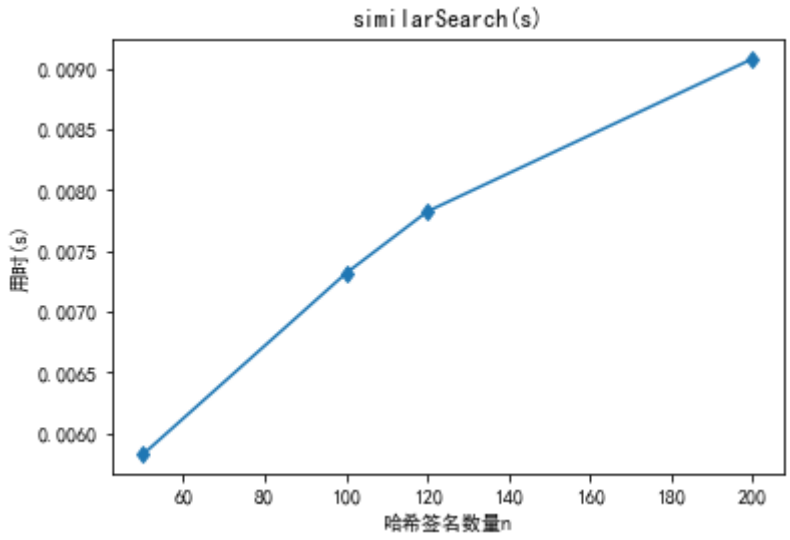
n=b*r

b	r	n	similarSearch(s)
50	1	50	0.005821938
50	2	100	0.000731935
100	1	100	0.007314271
120	1	120	0.007822737
60	2	120	0.001415861
40	3	120	0.000341943
30	4	120	0
24	5	120	0
20	6	120	0
15	8	120	0
100	2	200	0.000521721
200	1	200	0.009079593
200	2	400	0.002169229
400	3	1200	0.001241882
400	5	2000	0.00068767

可以看出查询用时普遍很短，

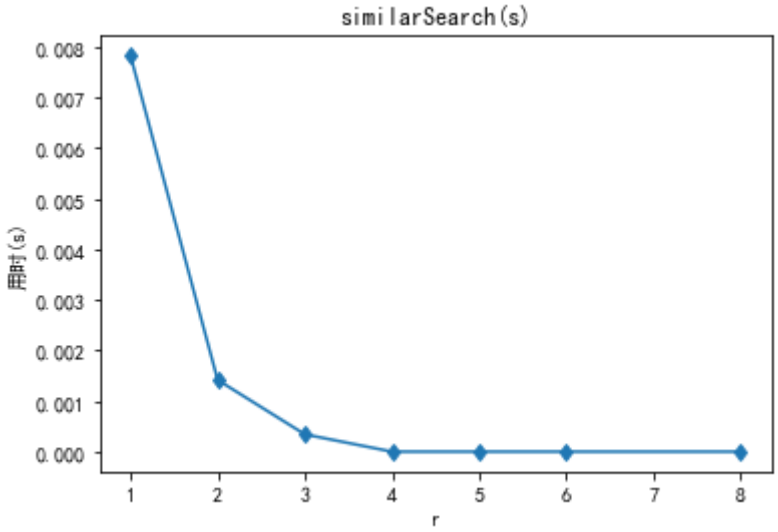
当r固定时，n越大，查询用时越大

当r=1时:



当n固定时，r越大，查询时间越短

当n=120时:



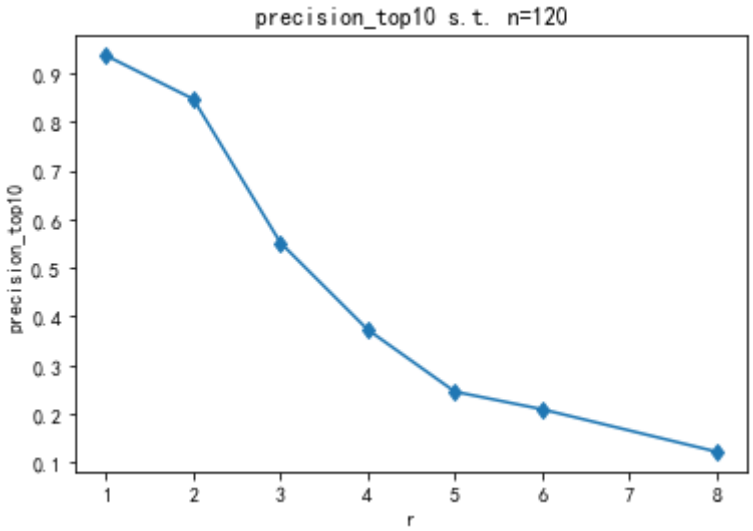
precision

b	r	n	precision_top10	precision_topk
50	1	50	0.968421053	0.978947368
50	2	100	0.780851064	0.867291456
100	1	100	0.929787234	0.962765957
120	1	120	0.936170213	0.965957447
60	2	120	0.846808511	0.918085106
40	3	120	0.55106383	0.829677474
30	4	120	0.372340426	0.62933975
24	5	120	0.244680851	0.450861196

b	r	n	precision_top10	precision_topk
20	6	120	0.208510638	0.379078014
15	8	120	0.121276596	0.261170213
100	2	200	0.861702128	0.917764269
200	1	200	0.931914894	0.965957447
200	2	400	0.891489362	0.941252955
400	3	1200	0.8	0.882877406
400	5	2000	0.476595745	0.823775751

1. 可以看出r=1或2时准确率普遍接近甚至大于0.9，n固定时，随着r增大，两种准确率都逐渐减小

b	r	n	precision_top10	precision_topk
120	1	120	0.936170213	0.965957447
60	2	120	0.846808511	0.918085106
40	3	120	0.55106383	0.829677474
30	4	120	0.372340426	0.62933975
24	5	120	0.244680851	0.450861196
20	6	120	0.208510638	0.379078014
15	8	120	0.121276596	0.261170213



2. r=1时：在n=50时，准确率就已经可以达到0.97，后面随着n的增大，precision_top10反而降到了0.93左右

所以对本实验数据集，选取b=50,r=1可以得到一个很准确的预测结果，而且因为n=50,其用时也短

b	r	n	precision_top10	precision_topk

b	r	n	precision_top10	precision_topk
50	1	50	0.968421053	0.978947368
100	1	100	0.929787234	0.962765957
120	1	120	0.936170213	0.965957447
200	1	200	0.931914894	0.965957447

3. 由于有一些节点候选集里节点数小于10，因此precision_topk总大于precision_top10

空间复杂度理论分析

记：数据集为图G=G(V,E)，节点数为18771，生成了n个签名向量(n=b*r)

1. 生成邻接矩阵oringinal_matrix:$O(|V|^2)$,后面以csc的形式存储，csc本质为邻接表，其空间复杂度为$O(|E|)$
2. genSigMatrix: $O(|V|*n)$，易得
3. genHashBuckets:$O(|V|*b)$，因为一共有$|V|*b$个长度为r的签名段，所以桶里面一共要装$|V|*b$个元素
4. similarSearch: $O(|V|)$，若要获得更紧的上界的话需要进行概率分析，求得一个节点的平均候选集个数

五、 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）

经过了稀疏矩阵按列压缩与构建哈希签名矩阵(2)的优化，本算法已经在时间复杂度，准确率方面取得一个很好的效果。

具体来说，对本实验数据集，选取b=50,r=1可以得到一个很好的效果：

b	r	n	genSigMatrix	gen_hashBuckets	similarSearch	precision_top10	precision_topk
50	1	50	3.258874178	2.563181639	0.005821938	0.968421053	0.978947368

总共用时仅仅5.827878s , 就已经达到了0.968421053的precision_top10 。

主要待改进的地方：

1. 空间复杂度的分析仅仅在理论层面上，应该用实验来检验
2. gen_hashBuckets, similarSearch用时、presicion受**n的大小**和**r与b比例**的影响，应该进行优化来确定最优的

   的n与r/b
3. 未探讨相似度阈值取值的选取与优化

六、 Reference

[1] (206条消息) python scipy 稀疏矩阵详解*scipy稀疏矩阵*|efferyLLLLL的博客-CSDN博客

[2] O Jafari · 2021 A Survey on Locality Sensitive Hashing Algorithms and their Applications