

2019 级

《大数据存储与管理》

课 程 报 告

——基于 LSH 的设计和实现

姓 名 邓至廷

学 号 U201915104

班 号 IOT1901 班

日 期 2022.04.16

目 录

一、内容介绍	1
二、LSH 原理	1
四、实验设计	2
五、实验过程	4
5.1 内存占用和运行时间	4
5.2 错误率	6
六、实验总结	6
参考文献	7

一、内容介绍

局部敏感哈希(LSH), 英文为 locality-sensitive hashing。LSH 常常也会被称做位置敏感哈希。LSH 是一种哈希算法, 最早在 1998 年由 Indyk 在上提出。不同于以往的哈希算法, 哈希最开始是为了减少冲突方便快速增删改查, 在这里 LSH 恰恰相反, 它利用的正式哈希冲突加速检索, 并且效果极其明显。

对于低维数据, 往往采用简单的线性查找, 但随着大数据时代的到来, 数据量往往巨大, 且大部分情况下都是高维数据。如果用线性查找高维数据, 那么效率就不够理想了。LSH 主要运用到高维海量数据的快速近似查找。近似查找便是比较数据点之间的距离或者是相似度。因此, 很明显, LSH 是向量空间模型下的东西。一切数据都是以点或者说以向量的形式表现出来的。所以, LSH 最大的特点便是高效处理海量高维数据的最近邻问题

二、LSH 原理

LSH 定义如下:

一个哈希函数族满足如下条件时, 被称为是 (R, cR, P_1, P_2) -sensitive, 对于任意两个点 $p, q \in R^d$:

If $\|p-q\| \leq R$, then $\Pr_H[h(q)=h(p)] \geq P_1$;

If $\|p-q\| \geq cR$, then $\Pr_H[h(q)=h(p)] \leq P_2$;

(通常要满足 $c > 1, P_1 > P_2$)

LSH 算法的核心思想是, 将高维数据降维到低维数据。在高维空间, 若两点距离很近, 设计一种哈希函数对两点计算哈希值, 使得他们的哈希值大概率一样; 反之, 若距离较远, 他们哈希值相同的概率也会很小。

在实现 LSH 的过程中, 选择距离计算的方法不同也会分为不同的算法。常见的有 Hamming 距离、欧式距离等等, 此处我所采用的是欧式距离, 实现 E^2 LSH 算法。

E^2 LSH 使用基于 p -stable 分布的哈希函数族, p -stable 分布定义如下:

对于一个实数集 R 上的分布 D , 如果存在 $P > 0$, 对任何 n 个实数 v_1, \dots, v_n 和 n 个满足 D 分布的变量 X_1, \dots, X_n , 随机变量 $\sum_i v_i X_i$ 和 $(\sum_i |v_i|^p)^{1/p} X$ 有相同的分布, 其中 X 是服从 D 分布的一个随机变量, 则称 D 为一个 p 稳定分布。

$p=1$ 时, 这个分布就是标准的柯西分布。概率密度函数: $c(x) = \frac{1}{\pi} \frac{1}{1+x^2}$

$p=2$ 时, 这个分布就是标准的正态分布。概率密度函数: $c(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$

三、LSH 流程以及减少空间开销

定义 g 函数 $g_i(v) = (h_1(v), h_2(v), \dots, h_k(v))$, $1 \leq i \leq L$, 对应 L 个哈希表, 每个 g 函数随机独立产生。每个 g 函数由在哈希函数族 $h_{a,b}$ 中随机独立选取的 k 个 h 函数组成。 g 函数的值对应具体的 hash bucket。但以此构建 hash table 时, 以上述组的

值作为 bucket 的标识的话，会出现空间复杂度大和不易查找的缺陷，同时为了满足选题要求，考虑减少 LSH 空间开销，将上述过程的末尾进行修改。

此时，若要减少 LSH 的空间开销，参考论文后，可以使用产生近邻点的方法来提高空间效率。考虑使用另外两个哈希函数 H1、H2，第一个由上述一个个组映射到 hash table 的任意 i 位上，作为哈希表 2 的索引，第二个则作为链表中桶的索引，两个函数具体表达式如下：

$$H_1(x_1 \sim x_k) = ((\sum_{i=1}^k r_i x_i) \bmod C) \bmod L$$

$$H_2(x_1 \sim x_k) = (\sum_{i=1}^k r'_i x_i) \bmod C$$

(C 为大素数，L 为组数)

由 $g_1 \sim g_L$ ，对应每个 $h_1 \sim h_k$ ，都被 H_1 映射为 $x_1 \sim x_k$ ， H_2 由上得出索引和数据向量的标识，大致结构如图 3.1，因此通过上述步骤可以减少 LSH 的空间开销。

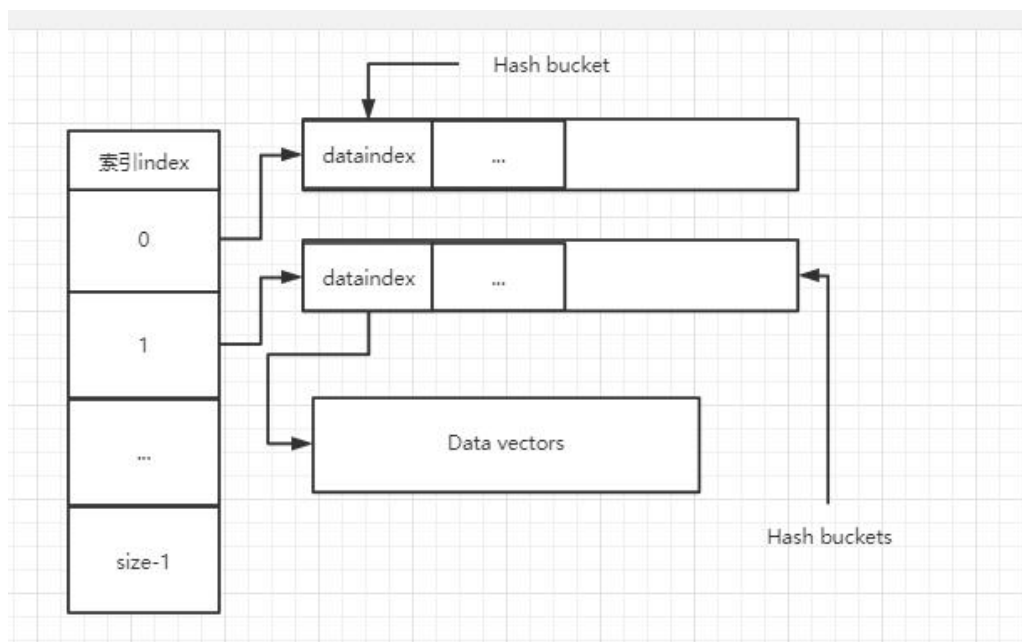


图 3.1 双哈希函数映射关系图

四、实验设计

本次实验，实现 LSH 的语言为 Python 3.10

实验环境：

处理器：Intel(R) Core(TM) i5-9400F CPU @ 2.90GHz 2.90 GHz

内存：16.0GB

编译器：PyCharm 2021.2.2

所用数据为 14 维，数据样例个数约为 200；

数据来源：大三上学期 大数据分析课程的白葡萄酒数据

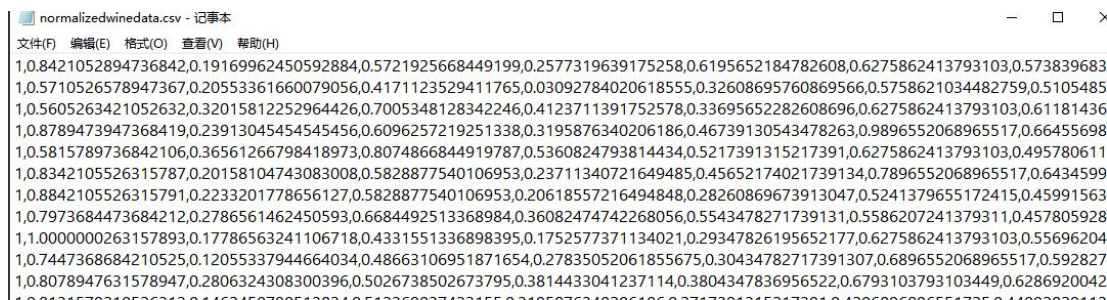


图 3.2 数据集截图

存储的结构设计:

```
class TableNode(object):
    def __init__(self, index):
        self.val = index
        self.buckets = {}
```

图 3.3 数据存储的结构图

参考官方 LSH 源码, E2LSH 主要实现过程如下:

```
def e2LSH(dataSet, k, L, r, tableSize):
    hashTable = [TableNode(i) for i in range(tableSize)]
    n = len(dataSet[0])
    m = len(dataSet)
    C = pow(2, 32) - 5
    hashFuncs = []
    fpRand = [random.randint(-10, 10) for i in range(k)]
    for times in range(L):
        e2LSH_family = gen_e2LSH_family(n, k, r)
        # hashFuncs: [[h1, ...hk], [h1, ..hk], ..., [h1, ...hk]]
        hashFuncs.append(e2LSH_family)
        for dataIndex in range(m):
            # generate K hash values
            hashVals = gen_HashVals(e2LSH_family, dataSet[dataIndex], r)
            # generate fingerprint
            fp = H2(hashVals, fpRand, k, C)
            # generate index
            index = fp % tableSize
            # find the node of hash table
            node = hashTable[index]
            # node.buckets is a dictionary: {fp: vector_list}
            if fp in node.buckets:
                # bucket is vector list
                bucket = node.buckets[fp]
                # add the data index into bucket
                bucket.append(dataIndex)
            else:
                node.buckets[fp] = [dataIndex]
    return hashTable, hashFuncs, fpRand
```

图 3.4 主要算法实现过程

Search 方法实现过程

```
def search(dataSet, query, k, L, r, tableSize):
    result = set()
    temp = e2LSH(dataSet, k, L, r, tableSize)
    C = pow(2, 32) - 5
    hashTable = temp[0]
    hashFuncGroups = temp[1]
    fpRand = temp[2]
    for hashFuncGroup in hashFuncGroups:
        queryFp = H2(gen_HashVals(hashFuncGroup, query, r), fpRand, k, C)
        # get the index of query in hash table
        queryIndex = queryFp % tableSize
        # get the bucket in the dictionary
        if queryFp in hashTable[queryIndex].buckets:
            result.update(hashTable[queryIndex].buckets[queryFp])
    return result
```

图 3.5 查找方法实现过程

五、实验过程

5.1 内存占用和运行时间

此处我使用 psutil 库计算程序占用内存，time 库计算运行时间，方法如下：

```
pid = os.getpid()
p = psutil.Process(pid)
info_start = p.memory_full_info().uss / 1024
```

```
print("程序运行了%s秒" % total_time)
info_end = p.memory_full_info().uss / 1024
print("程序占用了内存" + str(info_end - info_start) + "KB")
```

图 3.5 计算内存和时间方法

以查找如下数据

```
[2,0.20526318421052656,0.2727272924901186,0.7379679679144386,0.56185567525
7732,0.695652175,0.21379313793103452,0.1371308227848101,0.0188681132075471
83,0.36277605678233443,0.10409557167235495,0.3821139024390244,0.3626373992
673992,0.24750356640513554]
```

为例，在原数据所在行为 97，对应 index 为 96，程序将输出欧式距离相似的数据
最后的程序运行结果局部截图如下：

```

67:1.0201030034120233
68:1.1151658524813686
70:0.7087160489244738
72:0.8616169371693594
76:1.0103318920346125
77:0.7399755043143335
79:1.047818112652513
81:0.8896290388976577
82:0.9154324720870907
83:1.1940001270170855
85:0.8378801769113478
86:0.8566561697020533
88:0.8990647354263565
91:0.9155029290512355
93:0.9182135097772373
95:0.955481054926344
96:0.0
97:0.9807116186922465
99:1.2237721140159235
100:0.9074219170761209
102:0.8459148190495089
103:0.9133721501323858
105:1.2339805616466006
107:0.9414238433159675
108:0.9730008423880953
110:1.1709014113155558
111:0.8430792767404993
112:1.0213789529279276
113:0.8987414524522391
115:1.2515451061400031
116:0.8603801388825579

```

图 3.6 程序运行结果

```

程序运行了0.2909886837005615秒
程序占用了内存384.0KB

```

图 3.7 占用内存和时间

结果分析:

可以很明显的观察到, 所查结果索引为 96, 在 200 个维度为 14 的数据中查找时, 占用内存很小, 只占用了 384KB, 并且运行速度很快, 约用时 0.3 秒

5.2 错误率

参考文献[6], 给出如下 LSH 错误率计算公式:

$$\text{错误率} = \frac{1}{|Q|K} \sum_{q \in Q} \sum_{k=1}^K \frac{d_{\text{LSH}_k}}{d_k^*}$$

其中, Q 是实验中检索点的集合, K 为错误率和召回率的衡量标准, d_{LSH_k} 是通过 LSH 算法找到的第 k ($0 \leq k \leq K$) 个近邻点到查询点的距离, d_k^* 是真实的第 k 个近邻到查询点的距离。根据我的理解, 此公式表示 LSH 找到的 k 个近邻和真实 k 个近邻的对比结果, 所以在理想状态下, 错误率为 1.0, 结果越差, 错误率越大
根据以上数据计算错误率约为 1.02603

六、实验总结

在阅读许多文献后, 了解到 LSH 算法的强大, 以及对 LSH 各种的改进, 比如: 用树形结构代替哈希表, 使其具有自我校正参数的能力; 产生近邻查询点的方法提高空间效率, 不过会降低算法的空间效率; 以及用多重探测的方法改进欧式空间的 LSH 算法, 同时提高时间效率和空间效率。此外, 还了解到 LSH 的应用非常广泛, 近似检测、图像音频检索、聚类等等。不过由于我个人能力有限, 还无法熟练的把 LSH 算法用于更加复杂的数据集研究。在这学期学完此次课程以后, 我对存储有了很深刻的认识, 因为不管是做什么样的程序, 数据的存储是离不开的。有了存储, 就有了查询, 在海量的数据里缓慢查询是不现实的, LSH 给了我一种新的眼界, 比如这次尽管只查找了 200 个数据样例, 但是维度却很高, 如果以线性查找 14 维数据, 恐怕时间复杂度会非常高, 所以 LSH 的强大令我十分震撼, 此次课程我也收获匪浅。

参考文献

- [1] ZHENG Q, CHEN H, WANG Y 等. COSBench: A Benchmark Tool for Cloud Object Storage Services[C]//2012 IEEE Fifth International Conference on Cloud Computing. 2012: 998 - 999.
- [2] ARNOLD J. OpenStack Swift[M]. O' Reilly Media, 2014.
- [3] WEIL S A, BRANDT S A, MILLER E L 等. Ceph: A Scalable, High-performance Distributed File System[C]//Proceedings of the 7th Symposium on Operating Systems Design and Implementation. Berkeley, CA, USA: USENIX Association, 2006: 307 - 320.
- [4] Dean J, Barroso L A. Association for Computing Machinery, 2013. The Tail at Scale[J]. Commun. ACM, 2013, 56(2): 74 - 80.
- [5] Delimitrou C, Kozyrakis C. Association for Computing Machinery, 2018. Amdahl's Law for Tail Latency[J]. Commun. ACM, 2018, 61(8): 65 - 72.
- [6] 蔡衡, 李舟军, 孙健, 李洋. 基于 LSH 的中文文本快速检索[J]. 计算机科学, 2009, 36(8): 201-204.