华中科技大学

课 程 设 计 报 告

课程名称： 大数据存储系统与管理

报告选题： 基于 LSH 的设计与实现

专业班级：物联网 1901

学　　号：U201915040

姓　　名：高世文

指导教师：施展　华宇

报告日期：2022.04.18

计算机科学与技术学院

目录

基于 LSH 的设计与实现

高世文 U201915040

IOT1901 计算机学院

摘要 LSH(locality sensitivity hashing)算法是一种大量数据中进行相似度计算的算法。在传统的协同过滤算法中，无论是基于用户或物品的，都存有一个关键的问题是：如何计算两者之间的相似度。相似度的计算有很多种方式，常见的如欧式距离、余弦相似度或 Jaccard 相似度等等。在数据维度较小时，我们可以直接遍历每一个 pair 去计算，但当数据维度增大到一定程度时,这种计算方式的复杂度将大大提升,同时需要花费相当高的成本。因此需要采用 LSH 近似算法，以牺牲精度的代价大大节省计算效率。本文将粗浅的概述 LSH 算法的设计与实现。

关键词 LSH 最小哈希 p-stable hash

1 引言

LSH 算法的思想是，利用 hash 函数，将原始数据点映射到一个新的空间中，并且使得在原空间中距离相近的点会以很大的概率产生 hash 碰撞。当进行最邻近查找时，只需要计算查询点的 hash 值，然后提取所有与查询点产生 hash 碰撞的数据点，这些数据点可以在一个较大的概率下保证是与查询点相似的。

这样一来，我们只需要在这些相似的殿中寻找那个最邻近的，而无需遍历整个数据库。因此，LSH 算法能够帮助我们从整个数据库中找到一个子集，该子集的数据点会以很大概率与查询点相临近。LSH 算法中采用的 hash 函数并不同于传统的用于密码学中的 hash 函数。在密码学中，我们期望尽量避免 hash 碰撞，而在 LSH 算法中，我们希望能够最大化碰撞概率。

2 LSH 的基本思想

2.1 传统哈希算法

同常情况下通过建立哈希表的方式,我们可以得到时间复杂度为 1 的查找效率,该方法的关键之处在于选定一个哈希函数,将原始数据映射到相应的桶(Hash Bucket)中——哈希表中同一个位置可能存有多个元素，以应对哈希冲突问题，在不同数据被映射到同一个桶中时,可以再次哈希将数据映射到其他空桶中解决。

2.2 LSH 算法思想

LSH 算法的基本思想是：如果原始数据空间中的两个目标数据点相邻，那么

通过相同的映射或投影变换后，这两个数据点在新的数据空间中依然相邻的概率很大，而不相邻的数据点被映射到同一个桶的概率很小。也就是说，如果我们对原始数据进行了哈希映射后，我们希望原先相邻的两个数据被哈希映射到相同的桶里，具有相同的桶号。

首先已知的是，对原始数据集合中所有的数据都进行哈希映射后，可以得到一个哈希表，这些原始数据被分散到了哈希表的桶内，属于同一个桶内的数据很大可能是相邻的，但也会存在不相邻的数据被分配进同一个哈希桶内的情况。因此，问题的关键是找到一个这样的 LSH 函数——原始相邻数据在经其哈希映射后能落入同一个桶中。由此一来，我们在该数据集合中进行近邻查找就变得容易了，只需要将查询数据进行哈希映射得到其桶号，然后取出该桶号对应桶内的所有数据，再进行线性匹配即可查找到与查询数据相邻的数据。

也就是说，我们通过哈希函数映射变换操作，将原始数据集合分成了多个子集合，每个子集合中是相邻且个数较少的数据。这个过程就是将一个在海量数据中查找相邻元素的问题转化为了在一个很小的集合内查找相邻元素的问题，显然计算量下降了很多。

2.3 LSH 函数设计

由上分析我们可以看出 LSH 函数应该有如下性质：距离较远的点产生哈希碰撞的概率较小；距离较近的点，产生哈希碰撞的概率较高。从这一性质出发，我们可以这样定义 LSH 函数：

在距离空间 X(d)中，d 为距离，函数簇 H 是从 X 到 U 的映射，对任意 X 中的点 p，q，f 都满足：

（1）如果 d(x，y) ≤ d1， 则 h(x) = h(y)的概率>=p1；

（2）如果 d(x，y) ≥ d2， 则 h(x) = h(y)的概率<=p2；

其中 d(x，y)表示 x 和 y 之间的距离，d1 < d2， h(x)和 h(y)分别表示对 x 和 y 进行 hash 变换。满足上述条件的哈希函数称之为是(d1,d2,p1,p2)-sensitive。而通过一个或多个(d1,d2,p1,p2)-sensitive 的哈希函数对原始数据集合进行 hashing 生成一个或多个哈希表的过程就称为 Locality-sensitive Hashing。

2.4 LSH 算法查找过程

使用 LSH 进行对海量数据建立哈希表，并通过索引来进行近似最近邻查找的过程如下：

1. 离线建立索引

（1）选取满足(d1,d2,p1,p2)-sensitive 的 LSH 哈希函数；

（2）根据对查找结果的准确率（即相邻的数据被查找到的概率）确定哈希表的个数 L，每个哈希表内哈希函数的个数 K，以及跟 LSH 哈希函数自身有关的参数；

（3）将所有数据经过 LSH 哈希函数映射到相应的桶内，构成了一个或多个哈希表；

2. 在线查找

（1）将查询数据经过 LSH 哈希函数映射得到的相应桶号；

（2）将桶号中对应的数据取出，为了保证查找速度，通常只需要取出前 2L 个数据；

（3）计算查询数据与这 2L 个数据之间的相似度或距离，返回最近邻的数据；

LSH 在线查找时间由两个部分组成：

（1）通过 LSH 哈希函数计算 h 哈希值（桶号）的时间；

（2）将查询数据与桶内的数据进行比较计算的时间。因此，LSH 的查找时间至少是一个亚线性函数时间，因为我们可以通过对桶内的数据建立索引来加快匹配速度，这时第（2）部分的耗时就从 O(N)变成了 O(logN)或 O(1)（取决于采用的索引方法）。

LSH 为我们提供了一种在海量的高维数据集中查找与查询数据点（query data point）近似最相邻的某个或某些数据点的方法，但同时依旧需要注意的是，LSH 并不能保证一定能够查找到与带查询数据最相邻的数据点，而是减少需要匹配的数据点个数的同时保证查找到最近邻的数据点的概率很大。

3 LSH 具体设计与实现与空间开销的优化

3.1 p 稳定分布

不同的相似度判别方法，对应着不同的 LSH。在两种最常用的相似度下，有两种不同的 LSH，分别是：使用 jaccard 系数度量数据相似度的 min-hash 和使用欧氏几何距离度量数据相似度的 p-stable hash, 这里我们尝试着实现 p-stable hash.

首先需要明确 p 稳定分布的概念：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 v1,…,vn 和 n 个满足 D 分布的变量 X1,…,Xn，随机变量 Σ iviXi 和(Σ i|vi|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。对任何 p∈(0,2]存在稳定分布：

p=1 时是柯西分布，概率密度函数为 c(x)=1/[π (1+x2)]；

p=2 时是高斯分布，概率密度函数为 g(x)=1/(2π)1/2*e-x^2/2。

利用 p-stable 分布可以有效的近似高维特征向量，并在保证度量距离的同时，对高维特征向量进行降维，其关键思想是，产生一个 d 维的随机向量 a，随机向量 a 中的每一维随机的、独立的从 p-stable 分布中产生。对于一个 d 维的特征向量 v，如定义，随机变量 a • v 具有和(Σ i|vi|p)^(1/p)X 一样的分布，因此可以用 a • v 表示向量 v 来估算||v||p。

3.2 LSH 哈希函数构造与空间开销的优化

在 p 稳定的局部敏感 hash 中，我们将利用可以估计长度的性质来构建 hash 函数簇。具体如下：

（1）将空间中的一条直线分为长度为 r 且彼此等长的若干段。

（2）通过某映射函数（待构造的哈希函数），将空间中的点映射到这条直线

上，给映射到同一段的点赋予相同的 hash 值，与之前提到的理论一致，若空间中的两个点距离较近，他们被映射到同一段的概率也就越高。

综上，可以得到这样一个结论：空间中两个点距离||v1−v2||p||v1−v2||p，接近到一定程度时，应该被映射成同一 hash 值，而向量点积的性质，正好保持了这种局部敏感性。因此，可以用点积来设计 hash 函数簇。

参考文献中提出了这样一种 hash 函数簇：

$$h_{a,b}(v) = \lfloor \frac{a \cdot v + b}{r} \rfloor$$

可见，若要空间中的两个点映射为同一 hash 值，需要满足的条件为：这两点的点积加上随机值的计算结果在同一条线段上。

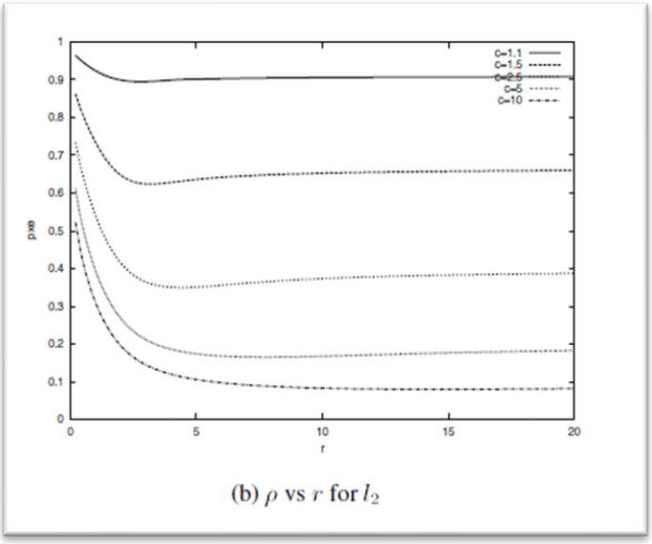
现在估计一下这个概率。设同分布，则概率公式如下：

$$p(c) = \Pr\left[h_{a,b}(v_1) = h_{a,b}(v_2)\right] = \int_0^r \frac{1}{c} f_p\left(\frac{t}{c}\right)\left(1 - \frac{t}{r}\right) dt$$

当 r 值取定的时候，这个公式可以看做是一个仅与 r 的取值相关的函数。c 越大，函数值越小（碰撞的概率越低）；c 越小，函数值越大（碰撞的概率越高）。

关于 r 的取值，在参考文献中，并没有给出一个确定的值。这需要我们根据实际情况需要来设定。

试想，因为我们设定的 LSH 是距离敏感的，所以不难推出：选取合适的 r 值，能够使得 c 尽可能的小。这里面的理论非常复杂，我并未完全搞清楚，但根据参考文献上提供的概率关系图，依旧可以看出以下几点信息：



(b) ρ vs r for l_2

（1）c 的取值不同时，对于相同的 r， ρ 也不同

（2）r 的取值大于某一点后， ρ 对 r 的变化不再敏感

（3）r 越大， ρ 越小。但是 r 的取值也不能过大，否则会导致 p1,p2 都接近于 1，增大搜索时间

综上可知，r 的取值要根据实际情况自行设定，空间开销优化的关键就体现在这里。

3.3 查询最相邻

根据 3.2 中的分析，可以设计两个哈希函数，H1 作为哈希表索引，H2 作为链表中的关键字：

$$H_1(x_1,\dots,x_k)=\left(\left(\sum_{i=1}^{k} r_i\, x_i\right)\ mod\ C\right) mod\ size$$

$$H_2(x_1,\dots,x_k)=\left(\sum_{i=1}^{k} r_i' x_i\right)\ mod\ C$$

通过如上两个哈希函数可以构建出哈希表，最终实现将索引相同的数据映射到同一个哈希桶中。

4 LSH 的应用

LSH 的应用场景很多，凡是需要进行大量数据之间的相似度计算的地方都可以使用 LSH 来加快查找匹配速度，下面是一些常见的应用：

（1）查找网络上的重复网页

目前的互联网环境中存在很多重复的网页，为了提高搜索引擎的检索质量或避免重复建立索引，需要查找出重复的网页，从而及时清除。这时可以将网页文档用一个集合或词袋向量来表征，然后通过一些 hash 运算来判断两篇文档之间的相似度，LSH 算法即可解决这类问题。

（2）检测图片相似性

在视觉数据处理平台，拥有从海量图片中学习并理解其内容的能力是非常重要的。为了检测重复或相似的图片，可以设计基于 LSH 搜索器的系统来完成这项任务。

（3）相似音乐查找

对于一段音乐或音频信息，可以根据其音频指纹来表征该音频片段，为了快速检索到与查询音频或歌曲相似的歌曲，我们可以对数据库中的所有歌曲的音频指纹建立 LSH 索引，然后通过该索引来加快检索速度。

（4）指纹匹配

在需要进行指纹识别的领域，指纹库一般是十分庞大的。当我们需要在海量指纹数据中定位到需要查找的那张时，我们可以对指纹的细节特征建立 LSH 索引，从而加快指纹的匹配速度。

等等。

5 附录

```
//p-stable hash LSH
import random
import numpy as np
```

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


def genPara(n, r):
    """

    :param n: length of data vector
    :param r:
    :return: a, b
    """

    a = []
    for i in range(n):
        a.append(random.gauss(0, 1))
    b = random.uniform(0, r)

    return a, b


def gen_e2LSH_family(n, k, r):
    """

    :param n: length of data vector
    :param k:
    :param r:
    :return: a list of parameters (a, b)
    """
    result = []
    for i in range(k):
        result.append(genPara(n, r))

    return result


def gen_HashVals(e2LSH_family, v, r):
    """

    :param e2LSH_family: include k hash funcs(parameters)
    :param v: data vector
```

```
        :param r:
        :return hash values: a list
        """

        # hashVals include k values
        hashVals = []

        for hab in e2LSH_family:
            hashVal = (np.inner(hab[0], v) + hab[1]) // r
            hashVals.append(hashVal)

        return hashVals


def H2(hashVals, fpRand, k, C):
        """

        :param hashVals: k hash vals
        :param fpRand: ri', the random vals that used to generate fingerprint
        :param k, C: parameter
        :return: the fingerprint of (x1, x2, ..., xk), a int value
        """
        return int(sum([(hashVals[i] * fpRand[i]) for i in range(k)]) % C)


def e2LSH(dataSet, k, L, r, tableSize):
        """
        generate hash table

        * hash table: a list, [node1, node2, ... node_{tableSize - 1}]
        ** node: node.val = index; node.buckets = {}
        *** node.buckets: a dictionary, {fp:[v1, ..], ...}

        :param dataSet: a set of vector(list)
        :param k:
        :param L:
        :param r:
        :param tableSize:
        :return: 3 elements, hash table, hash functions, fpRand
        """

        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 include L hash functions group, and each group contain k hash functions
        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
```

```
def nn_search(dataSet, query, k, L, r, tableSize):
    """

    :param dataSet:
    :param query:
    :param k:
    :param L:
    :param r:
    :param tableSize:
    :return: the data index that similar with query
    """

    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:

        # get the fingerprint of query
        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
```

//test.py

```
import e2LSH
from test_helper import *


if __name__ == "__main__":

    C = pow(2, 32) - 5
    dataSet = readData("test_data.csv")
    query = [-2.7769, -5.6967, 5.9179, 0.37671, 1]

    indexes = e2LSH.nn_search(dataSet, query, k=20, L=5, r=1, tableSize=20)
    for index in indexes:
        print(euclideanDistance(dataSet[index], query))
```

6 参考文献

• "Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions" (by Alexandr Andoni and Piotr Indyk). Communications of the ACM, vol. 51, no. 1, 2008, pp. 117-122.
• Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi- Probe LSH: Efficient Indexing for High-Dimensional Similarity Search," Proc. 33rd Int'l Conf. Very Large Data Bases (VLDB '07), pp. 950-961, 2007.
• Yu Hua, Bin Xiao, Bharadwaj Veeravalli, Dan Feng. "Locality-Sensitive Bloom Filter for Approximate Membership Query", IEEE Transactions on Computers (TC), Vol. 61, No. 6, June 2012, pages: 817-830.
• Yu Hua, Xue Liu, Dan Feng, "Data Similarity-aware Computation Infrastructure for the Cloud", IEEE Transactions on Computers (TC), Vol.63, No.1, January 2014, pages: 3-16
• LSH 系列一：p 稳定分布 LSH 算法初探 http://t.csdn.cn/Dl001
• 3.2&3.3 标题下全部数据、公式、图表均来自转载，已私信获得原作者同意
• 5.附录内代码仅供参考，为同一参考文献中作者所附。