華中科技大學

课 程 实 验 报 告

课程名称： 物联网数据存储与管理

选题名称： 基于 LSH 的设计和实现

专业班级： 物联网 1801 班

学　　号： U201814572

姓　　名： 黄世谱

指导教师： 华宇

报告日期： 2021 年 6 月 21 日

计算机科学与技术学院

目录

LSH 的设计和实现

一．选题背景及意义

在很多应用领域中，我们面对和需要处理的数据往往是海量并且具有很高的维度，怎样快速地从海量的高维数据集合中找到与某个数据最相似（距离最近）的一个数据或多个数据成为了一个难点和问题。如果是低维的小数据集，我们通过线性查找（Linear Search）就可以容易解决，但如果是对一个海量的高维数据集采用线性查找匹配的话，会非常耗时，因此，为了解决该问题，我们需要采用一些类似索引的技术来加快查找过程，通常这类技术称为最近邻查找(Nearest Neighbor,AN)，例如 K-d tree；或近似最近邻查找（Approximate Nearest Neighbor, ANN），例如 K-d tree with BBF, Randomized Kd-trees, Hierarchical K-means Tree。

从 NN 到 ANN

Nearest Neighbor （NN）近邻问题的定义：给定 n 点集合，搞个数据结构，当给定一个 query 时，返回最接近 query 的数据。

对于 NN，一般的解法如下：

1~2 维：Voronoi diagrams

10 维：kd trees, metric trees, ball-trees, spill trees

更高维：近似最近邻查找 Approximate Nearest Neighbor (ANN)，矢量量化方法等。

高维的 NN 会有维度灾难，所以有人提出了 Approximate Nearest Neighbor (ANN)。ANN 已经挺准啦，而且知道了 ANN 就可以算 NN，先用 ANN 找出所有近似近邻，然后选择最靠近的作为 NN 的解)。ANN 的时间复杂度是 sub-linear 的，不过空间复杂度还是挺高的。

从 ANN 到 LSH

局部敏感的哈希算法 locality-sensitive hashing （LSH）是 ANN 算法中的一种。LSH 能被用于去重，找相似文章，图像检索 ，推荐系统等。LSH 的基本思想是：把原始空间映射变换到 hash table 中，hash table 有很多桶，同一个桶里的数据很大可能是相邻的。假设原始空间有两相邻数据点，这两数据点在新的空间中相邻的概率（被 hash 到同一个桶中）很大，不相邻的数据点被映射到同一个桶的概率很小。

二．LSH 的定义

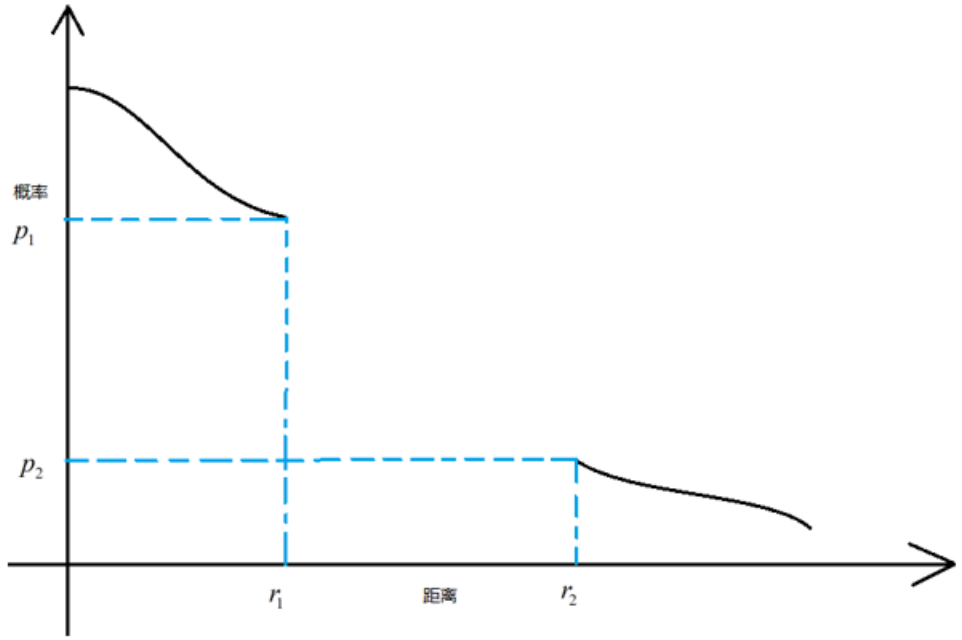
LSH 不像树形结构的方法可以得到精确的结果，LSH 所得到的是一个近似的结果，因为在很多领域中并不需非常高的精确度。即使是近似解，但有时候这个近似程度几乎和精准解一致。LSH 的主要思想是，高维空间的两点若距离很近，那么设计一种哈希函数对这两点进行哈希值计算，使得他们哈希值有很大的概率是一样的。同时若两点之间的距离较远，他们哈希值相同的概率会很小。给出 LSH 的定义如下：

我们将这样的一族 hash 函数 H={h:S→U}称为是 (r1,r2,p1,p2)敏感的，如果对于任意 H 中的函数 h，满足以下 2 个条件：

1. 如果 $d(O_1,O_2)<r_1$，那么 $P_r[h(O_1)=h(O_2)]\geqslant p_1$

2. 如果 $d(O_1,O_2)>r_2$，那么 $P_r[h(O_1)=h(O_2)]\leqslant p_2$

其中，$O_1,O_2\in S$，表示两个具有多维属性的数据对象，$d(O_1,O_2)$为 2 个对象的相异程度，也就是 1-相似度。其实上面的这两个条件说得直白一点，就是当足够相似时，映射为同一 hash 值的概率足够大；而足够不相似时，映射为同一 hash 值的概率足够小。



LSH 的基本思想是：将原始数据空间中的两个相邻数据点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仍然相邻的概率很大，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

行一些 hash 映射后，我们希望原先相邻的两个数据能够被 hash 到相同的桶内，具有相同的桶号。对原始数据集合中所有的数据都进行 hash 映射后，我们就得到了一个 hash table，这些原始数据集被分散到了 hash table 的桶内，每个桶会落入一些原始数据，属于同一个桶内的数据就有很大可能是相邻的，当然也存在不相邻的数据被 hash 到了同一个桶内。因此，如果我们能够找到这样一些 hash functions，使得经过它们的哈希映射变换后，原始空间中相邻的数据落入相同的桶内的话，那么我们在该数据集合中进行近邻查找就变得容易了，我们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。换句话说，我们通过 hash function 映射变换操作，将原始数据集合分成了多个子集合，而每个子集合中的数据间是相邻的且该子集合中的元素个数较小，因此将一个在超大集合内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。

三．LSH 建立索引过程

使用 LSH 进行对海量数据建立索引（Hash table）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 hash functions；

（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 hash table 的个数 L，每个 table 内的 hash functions 的个数 K，以及跟 LSH hash function 自身有关的参数；

（3）将所有数据经过 LSH hash function 哈希到相应的桶内，构成了一个或多个 hash table；

2. 在线查找

（1）将查询数据经过 LSH hash function 哈希得到相应的桶号；

（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 个数据即可）；

（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 hash functions 计算 hash 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 sublinear 时间。为什么是“至少”？因为我们可以通过对桶内的属于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。需要注意的是，LSH 并不能保证一定能够查找到与 query data point 最相邻的数据，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。

四．不同 LSH 设计

针对不同的相似度测量方法，局部敏感哈希的算法设计也不同，我们主要看看在两种最常用的相似度下，两种不同的 LSH：

1. 使用 Jaccard 系数度量数据相似度时的 min-hash

2. 使用欧氏距离度量数据相似度时的 P-stable hash

无论是哪种 LSH，都是将高维数据降维到低维数据，同时，还能在一定程度上，保持原始数据的相似度不变。LSH 不是确定性的，而是概率性的，也就是说有一定的概率导致原本很相似的数据映射成 2 个不同的 hash 值，或者原本不相似的数据映射成同一 hash 值。这是高维数据降维过程中所不能避免的（因为降维势必会造成某种程度上数据的失真），不过好在 LSH 的设计能够通过相应的参数控制出现这种错误的概率，这也是 LSH 为什么被广泛应用的原因。

1. min-hash

hash 函数的选择

Jaccard 系数：Jaccard 系数主要用来解决的是非对称二元属性相似度的度量问题，常用的场景是度量 2 个集合之间的相似度。

比如在下面的表格中写出了 4 个对象（文档）的集合情况，每个文档有相应的词项，用词典{w1,w2,…,w7}表示。若某个文档存在这个词项，则标为 1，否则标 0。

word	D_1	D_2	D_3	D_4
w_1	1	0	1	0
w_2	1	1	0	1
w_3	0	1	0	1
w_4	0	0	0	1
w_5	0	0	0	1
w_6	1	1	1	0
w_7	1	0	1	0

首先，我们现在将上面这个 word-document 的矩阵按行置换，比如可以置换成以下的形式：

word	D_1	D_2	D_3	D_4
w_2	1	1	0	1
w_1	1	0	1	0
w_4	0	0	0	1
w_3	0	1	0	1
w_7	1	0	1	0
w_6	1	1	1	0
w_5	0	0	0	1

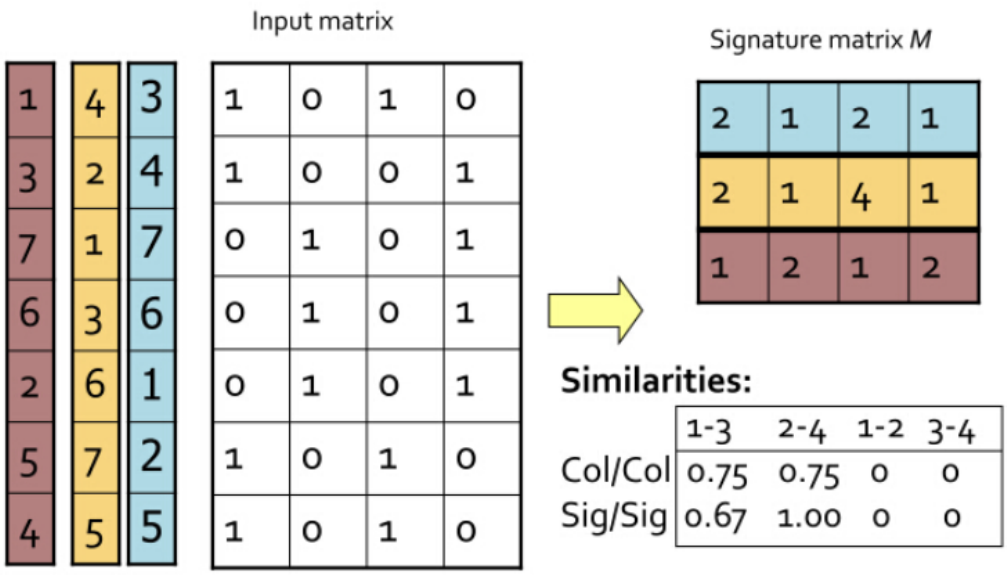
可以确定的是，这没有改变文档与词项的关系。现在做这样一件事：对这个矩阵按行进行多次置换，每次置换之后，统计每一列（其实对应的就是每个文档）第一个不为 0 的位置（行号），这样每次统计的结果能构成一个与文档数等大的向量，这个向量，我们称之为签名向量。

比如，如果对最上面的矩阵做这样的统计，得到[1,2,1,2]，对于下面的矩阵做统计，得到[1,1,2,1]。

简单来想这个问题，就拿上面的文档来说，如果两个文档足够相似，那也就是说这两个文档中有很多元素是共有的，换句话说，这样置换之后统计出来的签名向量，如果其中有一些文档的相似度很高，那么这些文档所对应的签名向量的相应的元素，值相同的概率就很高。

我们把最初始时的矩阵叫做 input matrix，由 m 个文档，n 个词项组成。而把由 t 次置换后得到的一个 t×m 的矩阵叫做 signature matrix。

流程如下图：



图中，4 个文档，做了 3 次置换，得到了一个 3x4 的签名矩阵。

需要注意的是，置换矩阵的行，在代码实现的时候，可以用这样的算法实现：

1. 在当下剩余的行中（初始时，剩余的行为全部行），随机选取任意一行，看看这一行哪些位置（这里的位置其实是列号）的元素是 1，如果签名向量中这个位置的元素还未被写入，则在这个位置写入随机选取的这个行的行号。并将这一行排除。

2. 持续进行 1 步的工作，直到签名向量全部被写满为止。

3. 以上 2 步的意义跟对整个矩阵置换、再统计，结果是一样的。函数如下：

```
def sigGen(matrix):
    """
    * generate the signature vector

    :param matrix: a ndarray var
    :return a signature vector: a list var
    """

    # the row sequence set
    seqSet = [i for i in range(matrix.shape[0])]

    # initialize the sig vector as [-1, -1, ..., -1]
    result = [-1 for i in range(matrix.shape[1])]

    count = 0

    while len(seqSet) > 0:

        # choose a row of matrix randomly
        randomSeq = random.choice(seqSet)

        for i in range(matrix.shape[1]):

            if matrix[randomSeq][i] != 0 and result[i] == -1:
                result[i] = randomSeq
                count += 1
        if count == matrix.shape[1]:
            break

        seqSet.remove(randomSeq)

    # return a list
    return result
```

现在给出一个定理。

定理：对于签名矩阵的任意一行，它的两列元素相同的概率是 x/n，其中 x 代表这两列所对应的文档所拥有的公共词项的数目。而 x/n 也就是这两个文档

的 Jaccard 系数。

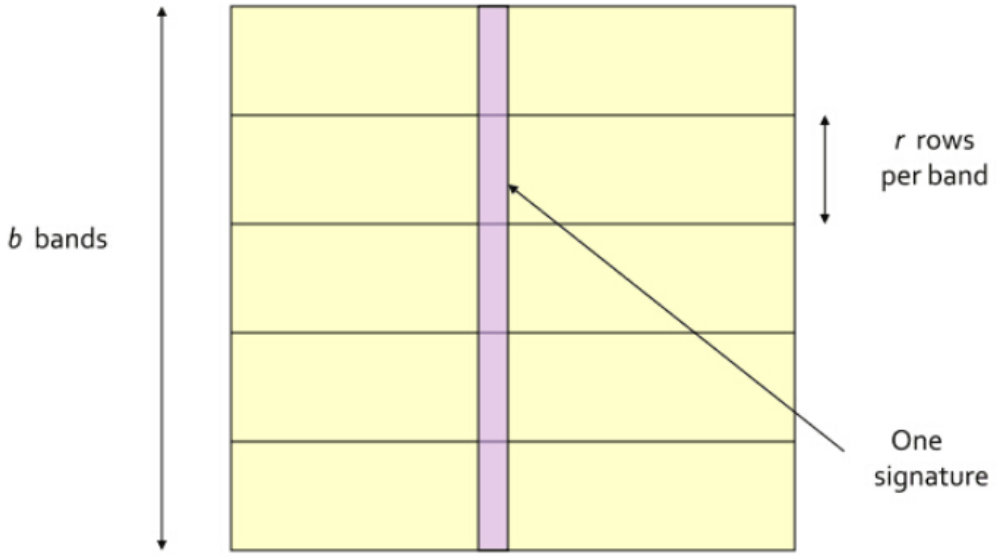
置换 input matrix 的行，取每列第一个非 0 元的做法，就是一个 hash 函数。这个 hash 函数成功地将多维数据映射成了一维数据。而从这个定理我们发现，这样的映射没有改变数据相似度。

需要注意的一点是，这里的 hash 函数只能对 Jaccard 系数定义数据相似度的情况起作用。不同的相似度模型，LSH 是不同的，目前，还不存在一种通用的 LSH。

构造 LSH 函数族

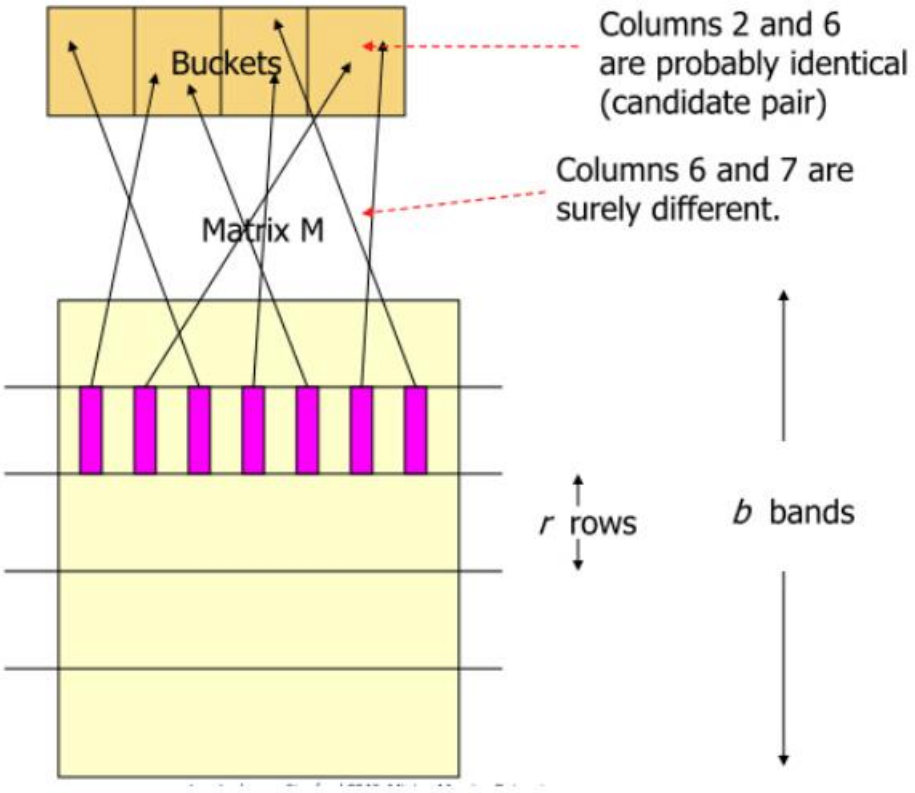
为了能实现前面 LSH 定义中的 2 个条件的要求，我们通过多次置换，求取向量，构建了一组 hash 函数。也就是最终得到了一个 signature matrix. 为了控制相似度与映射概率之间的关系，我们需要按下面的操作进行，一共三步。

(1) 将 signature matrix 水平分割成一些区块（记为 band），每个 band 包含了 signature matrix 中的 r 行。需要注意的是，同一列的每个 band 都是属于同一个文档的。如下图所示。



Signature matrix *M*

(2) 对每个 band 计算 hash 值，这里的 hash 算法没有特殊要求，MD5，SHA1 等等均可。一般情况下，我们需要将这些 hash 值做处理，使之成为事先设定好的 hash 桶的 tag，然后把这些 band“扔”进 hash 桶中。如下图所示。但是这里，我们只是关注算法原理，不考虑实际操作的效率问题。所以，省略处理 hash 值得这一项，得到每个 band 的 hash 值就 OK 了，这个 hash 值也就作为每个 hash bucket 的 tag。



(3) 如果某两个文档的，同一水平方向上的 band，映射成了同一 hash 值（如果你选的 hash 函数比较安全，抗碰撞性好，那这基本说明这两个 band 是一样的），我们就将这两个文档映射到同一个 hash bucket 中，也就是认为这两个文档是足够相近的。

对于两个文档的任意一个 band 来说，这两个 band 值相同的概率是：s^r，其中 $s \in [0,1]$是这两个文档的相似度。

也就是说，这两个 band 不相同的概率是 $1-s^r$

这两个文档一共存在 b 个 band，这 b 个 band 都不相同的概率是$(1-s^r)^b$

所以说，这 b 个 band 至少有一个相同的概率是 $1-(1-s^r)^b$

概率 $1-(1-s^r)^b$就是最终两个文档被映射到同一个 hash bucket 中的概率。这样一来，实际上可以通过控制参数 r,b 的值来控制两个文档被映射到同一个哈希桶的概率。而且效果非常好。比如，令 b=20,r=5。

当 s=0.8 时，两个文档被映射到同一个哈希桶的概率是：

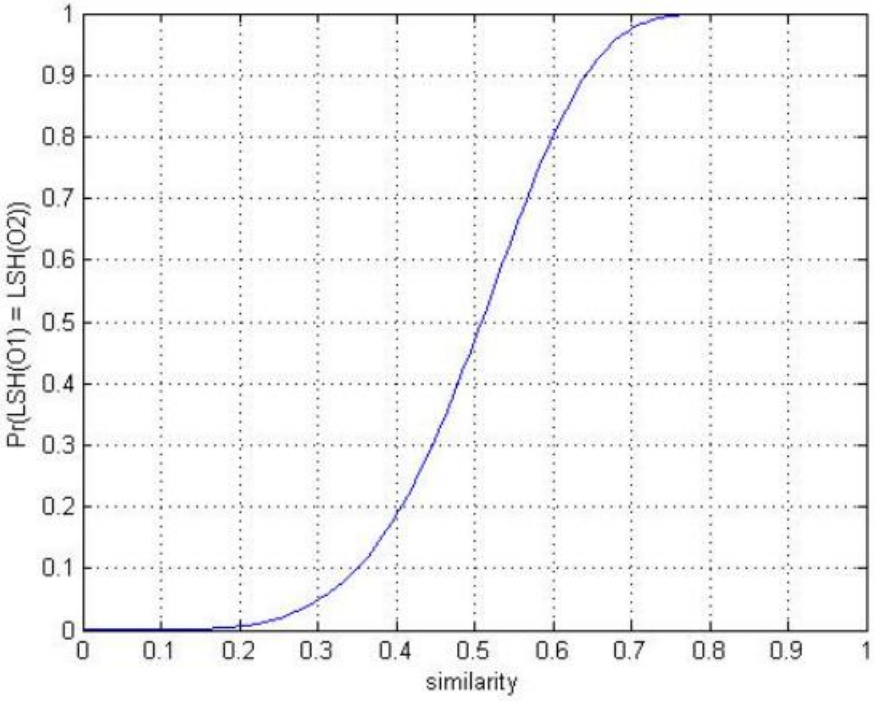
$Pr(LSH(O_1)=LSH(O_2))=1-(1-0.8^5)^5=0.9996439421094793$

当 s=0.2 时，两个文档被映射到同一个哈希桶的概率是：

$Pr(LSH(O_1)=LSH(O_2))=1-(1-0.2^5)^5=0.0063805813047682$

不难看出，这样的设计通过调节参数值，达到了“越相似，越容易在一个哈希桶；越不相似，越不容易在一个哈希桶”的效果。这也就能实现我们上边说的 LSH 的两个性质。

在 r=5,b=20 参数环境下的概率图如下。



当相似度高于某个值的时候，概率会变得非常大，并且快速靠近 1，而当相似度低于某个值的时候，概率会变得非常小，并且快速靠近 0。

另外， 需要注意的是，每一层的 band 只能和同一层的 band 相比，若 hash 值相同，则放入同一个哈希桶中。

2.P-stable hash

p 稳定分布：

定义：一个分布 D 称为 p 稳定分布，如果对于任意 n 个实数 $v_1,v_2,\cdots,v_n$ 和符合 D 分布的 n 个独立同分布的随机变量 $X_1,X_2,\cdots,Xn$，都存在一个 $p\geqslant 0$，使得 $\Sigma_i v_i X_i$ 和 $(\Sigma_i|v_i|p)^{1/p}$ 具有相同的分布，其中，X 是一个满足 D 分布的随机变量。

在 $p\in(0,2]$ 这个范围内存在稳定分布。我们最常见的是 p=1 以及 p=2 时的情况。

- $p=1$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：$c(x)=\frac{1}{\sqrt{2\pi}}e^{-x^2/2}$

当然，p 值不是仅能取 1 和 2,(0,2]中的小数也是可以的。

p 稳定分布可以估计给定向量 v 在欧式空间下的 p 范数的长度，也就是 $||v||_p$。

可以这样实现：对于一个向量 v（相当于上面公式中的(v1,v2,…,vn)），现在从 p 稳定分布中，随机选取 v 的维度个随机变量（相当于上面公式中的 X1,X2,…,Xn）构成向量 a，计算 $a \cdot v=\sum_i v_i X_i$，此时，a·v 与 $||v||_p X$ 同分布。我们就可以通过多给几个不同的向量 a，多计算几个 a·v 的值，来估计 $||v||_p$ 的值。

p-stable 分布 LSH 函数族构造

在 p 稳定的局部敏感 hash 中，我们利用 a·v 可以估计 $||v||_p$ 长度的性质来构建 hash 函数族。具体如下：

1.将空间中的一条直线分成长度为 r 的，等长的若干段。

2.通过一种映射函数（也就是我们要用的 hash 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。不难理解，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。

3.a·v 可以估计 $||v||_p$ 长度，那么，$(a \cdot v_1-a \cdot v_2)=a(v_1-v_2)$ 也就可以用来估计 $||v1-v2||_p$ 的长度。

综合上面的 3 条，可以得到这样一个结论：空间中两个点距离：$||v1-v2||_p$，近到一定程度时，应该被 hash 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 函数族。

p-stable 分布 LSH 相似性搜索算法

上面完成了对 p-stable 分布 LSH 函数族构造。那么接下来的问题是怎样具体实现 hash table 的构造以及查询最近邻。

我们构建 hash table 的过程就是要用这个函数族的每一个函数对每一个向量执行 hash 运算。为了减少漏报率 False Negative，一种解决方案是用多个 hash 函数对向量执行 hash 运算，比如说，对任意一个向量 v_i，现在准备了 k 个 hash 函数 $(h_1(),h_2(),\cdots,h_k())$，这 k 个 hash 函数是从 LSH 函数族中随机选取的 k 个，这样，通过计算，就得到了 k 个 hash 值：$(h_1(v_i),h_2(v_i),\cdots,h_k(v_i))$，而对于查询 q，用同样的 k 个 hash 函数，也能得到一组值 $(h_1(q),h_2(q),\cdots,h_k(q))$，这两组值之间，只要有一个对应位的值相等，我们就认为 v_i 是查询 q 的一个近邻。

但是，现在有一个问题，那就是上面这种做法的结果，确实减少了漏报率，但与此同时，也增加了误报率（本来不很相近的两条数据被认为是相近的）。所以，需要在上面方法的基础上，再增加一个措施。我们从 LSH 函数族中，随机选取 L 组这样的函数组，每个函数组都由 k 个随机选取的函数构成，

当然 L 个函数组之间不一定是一样的。现在这 L 组函数分别对数据处理，只要有一组完全相等，就认为两条数据是相近的。

现在假设 $P=Pr[h_{a,b}(v_1)=_{ha,b}(v_2)]$，那么，两条数据被认为是近邻的概率是：

$$1-(1-P^k)^L$$

构建 hash table 时，如果把一个函数组对向量的一组 hash 值 $(h_1(v_i),h_2(v_i),\cdots,h_k(v_i))$作为 hash bucket 的标识，有两个缺点：1.空间复杂度大；2.不易查找。为了解决这个问题，我们采用如下方法：

先设计两个 hash 函数：H1,H2

1.H1：Zk→{0,1,2,…,size-1}.简单说就是把一个 k 个数组成的整数向量映射到 hash table 的某一个位上，其中 size 是 hash table 的长度。

2.H2：Zk→{0,1,2,…,C}. $C=2^{32}-5$，是一个大素数。

这两个函数具体的算法如下，其中，r_i,r'_i是两个随机整数。

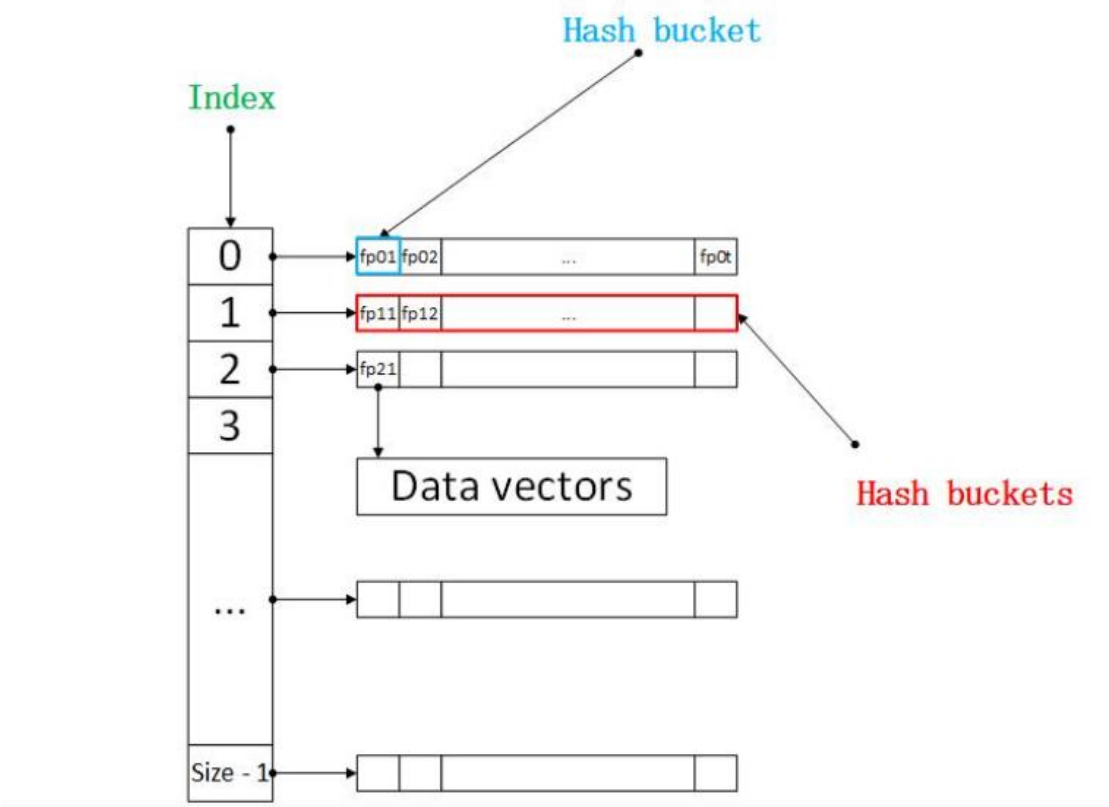
$$H_1(x_1,\ldots,x_k)=((\sum_{i=1}^{k}r_ix_i)\mod C)\mod size$$

$$H_2(x_1,\ldots,x_k)=(\sum_{i=1}^{k}r'_ix_i)\mod C$$

我们把 H2 计算的结果成为一个数据向量的“指纹”，它就是由数据向量的 k 个 hash 值计算得到的。而 H1 相当于是数据向量的指纹在 hash table 中的索引，这个算法跟基本的散列表算法是一个思路。

通过这两个新建的函数，我们可以将 hash table 的构建步骤作以下详细说明：

1. 从设计好的 LSH 函数族中，随机选取 L 组 hash 函数，每组由 k 个 hash 函数构成，记为$\{g_1(\cdot),g_2(\cdot),\cdots,g_L(\cdot)\}$，其中 $g_i(\cdot)=(h_1(\cdot),h_2(\cdot),\cdots,h_k(\cdot))$
2. 每个数据向量经过 $g_i(\cdot)$被映射成一个整型向量，记为$(x_1,\cdots,x_k)$
3. 将 2 步生成的(x1,…,xk)(x1,…,xk)通过 H1,H2H1,H2 计算得到两个数值：index,fpindex,fp，前者是 hash table 的索引，后者是数据向量对应的指纹。这里，为了方便描述这种 hash table 的结构，我将我们用的 hash table 的结构画出，如图所示。
4. 若其中有数据向量拥有相同的数据指纹，那么必然会被映射到同一个 hash bucket 当中。



用 L 组 hash 函数计算数据指纹及相应索引的时候，可能出现两个不相近的数据被两组不同的 hash 函数族映射为相同数据指纹的情况。这显然增加了误报率，所以一种可行的改进方法为：建立 L 个 hash 表，两个数据只要在任意一个 hash 表内被映射为相同的指纹，就认为二者是相近的。

数据向量由 H_2 生成数据指纹（图中的 fp_{01}, fp_{12}），每个数据指纹就是一个 hash bucket 的标识，存储着对应的数据向量。

可以得到相同 H1(·)值的 hash bucket 我们放在一个链表中，这个链表对应的就是 hash table 中相应的索引。

五、增强 LSH（Amplifying LSH）

通过 LSH hash functions 我们能够得到一个或多个 hash table，每个桶内的数据之间是近邻的可能性很大。我们希望原本相邻的数据经过 LSH hash 后，都能够落入到相同的桶内，而不相邻的数据经过 LSH hash 后，都能够落入到不同的桶中。如果相邻的数据被投影到了不同的桶内，我们称为 false negtive；如果不相邻的数据被投影到了相同的桶内，我们称为 false positive。因此，我们在使用 LSH 中，我们希望能够尽量降低 false negtive rate 和 false positive rate。

通常，为了能够增强 LSH，即使得 false negative rate 和/或 false positive rate 降低，我们有两个途径来实现：

1）在一个 hash table 内使用更多的 LSH hash function；

2）建立多个 hash table。

下面介绍一些常用的增强 LSH 的方法：

1. 使用多个独立的 hash table

每个 hash table 由 k 个 LSH hash function 创建，每次选用 k 个 LSH hash function（同属于一个 LSH function family）就得到了一个 hash table，重复多次，即可创建多个 hash table。多个 hash table 的好处在于能够降低 false positive rate。

2. AND 与操作

从同一个 LSH function family 中挑选出 k 个 LSH function，H(X) = H(Y)有且仅当这 k 个 Hi(X) = Hi(Y)都满足。也就是说只有当两个数据的这 k 个 hash 值都对应相同时，才会被投影到相同的桶内，只要有一个不满足就不会被投影到同一个桶内。

AND 与操作能够使得找到近邻数据的 p1 概率保持高概率的同时降低 p2 概率，即降低了 false negative rate。

3. OR 或操作

从同一个 LSH function family 中挑选出 k 个 LSH function，H(X) = H(Y)有且仅当存在一个以上的 Hi(X) = Hi(Y)。也就是说只要两个数据的这 k 个 hash 值中有一对以上相同时，就会被投影到相同的桶内，只有当这 k 个 hash 值都不相同时才不被投影到同一个桶内。

OR 或操作能够使得找到近邻数据的 p1 概率变的更大（越接近 1）的同时保持 p2 概率较小，即降低了 false positive rate。

4. AND 和 OR 的级联

将与操作和或操作级联在一起，产生更多的 hahs table，这样的好处在于能够使得 p1 更接近 1，而 p2 更接近 0。

除了上面介绍的增强 LSH 的方法外，有时候我们希望将多个 LSH hash function 得到的 hash 值组合起来，在此基础上得到新的 hash 值，这样做的好处在于减少了存储 hash table 的空间。常用方法如下：

1. 求模运算

new hash value = old hash value % N

2. 随机投影

假设通过 k 个 LSH hash function 得到了 k 个 hash 值：h1, h2..., hk。那么新的 hash 值采用如下公式求得：

*new hash value = h1*r1 + h2*r2 + ... +hk*rk*，其中 r1, r2, ..., rk 是一些随机数。

3. XOR 异或

假设通过 k 个 LSH hash function 得到了 k 个 hash 值：h1, h2..., hk。那么新的 hash 值采用如下公式求得：

new hash value= h1 XOR h2 XOR h3 ... XOR hk

六．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（或距离）计算的地方都可以使用 LSH 来加快查找匹配速度，下面列举一些应用：

（1）查找网络上的重复网页

互联网上由于各式各样的原因（例如转载、抄袭等）会存在很多重复的网页，因此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，以便进行一些处理。其大致的过程如下：将互联网的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，常用的有 minhash+LSH、simhash。

（2）查找相似新闻网页或文章

与查找重复网页类似，可以通过 hash 的方法来判断两篇新闻网页或文章是否相似，只不过在表达新闻网页或文章时利用了它们的特点来建立表征该文档的集合。

（3）图像检索

在图像检索领域，每张图片可以由一个或多个特征向量来表达，为了检索出与查询图片相似的图片集合，我们可以对图片数据库中的所有特征向量建立 LSH 索引，然后通过查找 LSH 索引来加快检索速度。目前图像检索技术在最近几年得到了较大的发展，有兴趣的读者可以查看基于内容的图像检索引擎的相关介绍。

（4）音乐检索

对于一段音乐或音频信息，我们提取其音频指纹（Audio Fingerprint）来表征该音频片段，采用音频指纹的好处在于其能够保持对音频发生的一些改变的鲁棒性，例如压缩，不同的歌手录制的同一条歌曲等。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。

（5）指纹匹配

一个手指指纹通常由一些细节来表征，通过对比较两个手指指纹的细节的相似度就可以确定两个指纹是否相同或相似。类似于图片和音乐检索，我们可以对这些细节特征建立 LSH 索引，加快指纹的匹配速度。

．参考文献

•http://people.csail.mit.edu/indyk/

•http://www.mit.edu/~andoni/LSH/

•"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