

**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](#_Toc14954)

[二、LSH原理 1](#_Toc1520)

[四、实验设计 2](#_Toc8220)

[五、实验过程 4](#_Toc15313)

[5.1 内存占用和运行时间 4](#_Toc18617)

[5.2 错误率 6](#_Toc26073)

[六、实验总结 6](#_Toc22306)

[参考文献 7](#_Toc11450)

# 一、内容介绍

局部敏感哈希(LSH)，英文为locality-sensetive 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。

对于低纬数据，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纬数据。如果用线性查找高纬数据，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,P1,P2)-sensitvie，对于任意两个点  p，q∈Rd：

If ||p-q|| ≤ R ,then PrH[h(q)=h(p)]≥P1;

If ||p-q|| ≥c R ,then PrH[h(q)=h(p)]≤P2;

(通常要满足c>1,P1>P2)

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。常见的也Hamming距离、欧式距离等等，此处我所采用的是欧式距离，实现E2LSH算法。

E2LSH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v1,…,vn和n个满足D分布的变量X1,…,Xn，随机变量和()1/pX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)=

p=2时，这个分布就是标准的正态分布。概率密度函数：c(x)=

**三、LSH流程以及减少空间开销**

定义g函数gi(v)=(h1(v),h2(v),...,hk(v))，1≤i≤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 ha,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：

(~)=) mod ) mod L

(C为大素数，L为组数)

由g1~gL，对应每个h1~hk,都被H1映射为x1~xk,H2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。

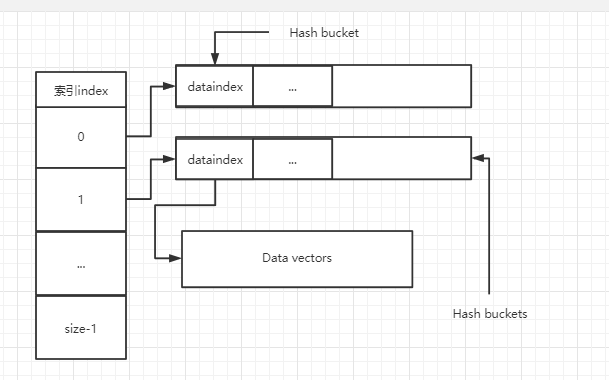


图3.1双哈希函数映射关系图

# 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

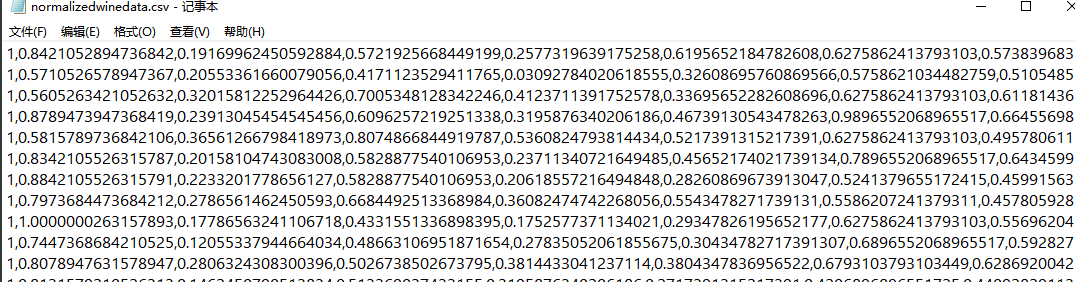


图3.2数据集截图

存储的结构设计：

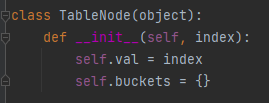


图3.3数据存储的结构图

参考官方LSH源码，E2LSH主要实现过程如下：

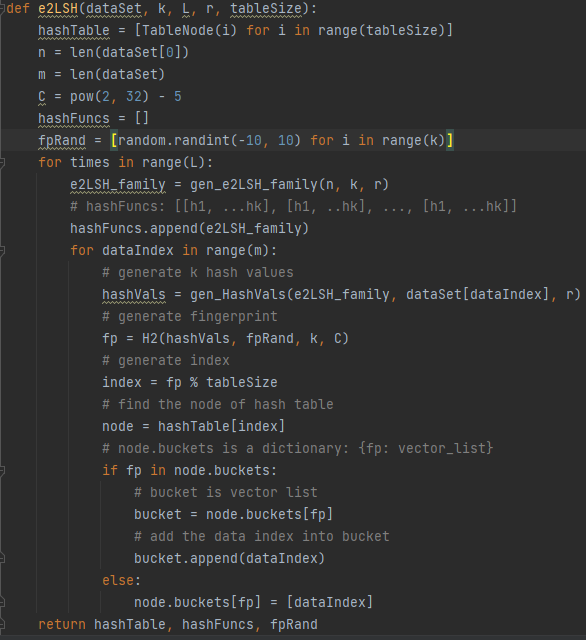


图3.4主要算法实现过程

Search方法实现过程

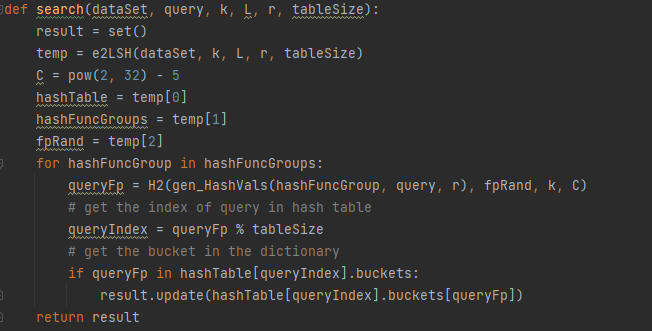
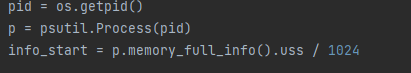


图3.5查找方法实现过程

# 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：



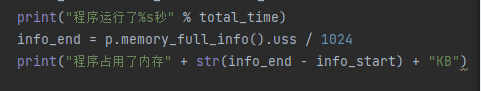


图3.5计算内存和时间方法

以查找如下数据

[2,0.20526318421052656,0.2727272924901186,0.7379679679144386,0.561855675257732,0.695652175,0.21379313793103452,0.1371308227848101,0.018868113207547183,0.36277605678233443,0.10409557167235495,0.3821139024390244,0.3626373992673992,0.24750356640513554]

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：

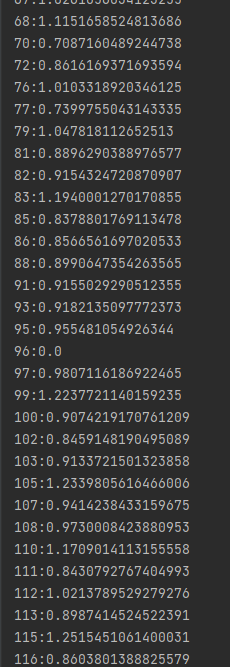


图3.6程序运行结果



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：

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，是通过LSH算法找到的第k（0≤k≤K）个近邻点到查询点的距离，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.