华东师范大学软件工程学院上机实践报告

课程名称： 数据结构与算法实践	**年级：** 2023 级	**实践成绩：**
指导教师： ***	**姓名：** Eliot	
上机实践名称： 相似最近邻搜索	**学号：** ********	**实践日期：** 2024 春

一、背景介绍

嵌入 (有时也被称为向量， 后面我们会交替使用. Embedding/Vector) 是目前大语言模型如 ChatGPT 等背后的支撑技术。它的核心思想在于将我们日常接触的文本、 图片、视频等内容通过神经网络转换为高维度的向量表示 (这个向量就被称为嵌入)。 如图 1 所示：

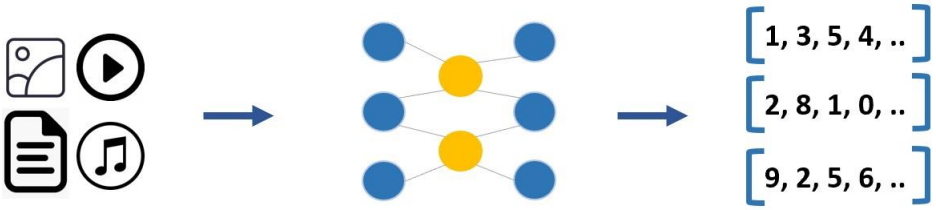


图 1: 文本/图片/视频/音频 –> 神经网络 –> 向量

我们可以以文本为例， King 这个单词通过神经网络可以转为向量 (维度为 6)：

$$\begin{bmatrix} 0.50451 & 0.68607 & -0.59517 & -0.022801 & 0.60046 & -0.13498 \end{bmatrix}$$

通过将文本内容转换为向量， 我们可以实现更加强大的语义搜索 (传统搜索使用关键词匹配， 字符串编辑距离)， 例如 <One happy dog> 和 <A playful hound>， 尽管它们没有共享任何一个相同的关键词， 但是它们的语义是相同的。传统的关键词匹配算法无法处理这种语义搜索问题。当我们将文本通过神经网络转换为向量之后， 那些在语义上接近的单词， 在向量空间中也会更加接近， 如图 2， 3 所示。当然你也可以在通过这些网站中更加具体的体会到这一点：

- Google 向量可视化

- 语义搜索
- 向量数据格式
- 语义图片搜索

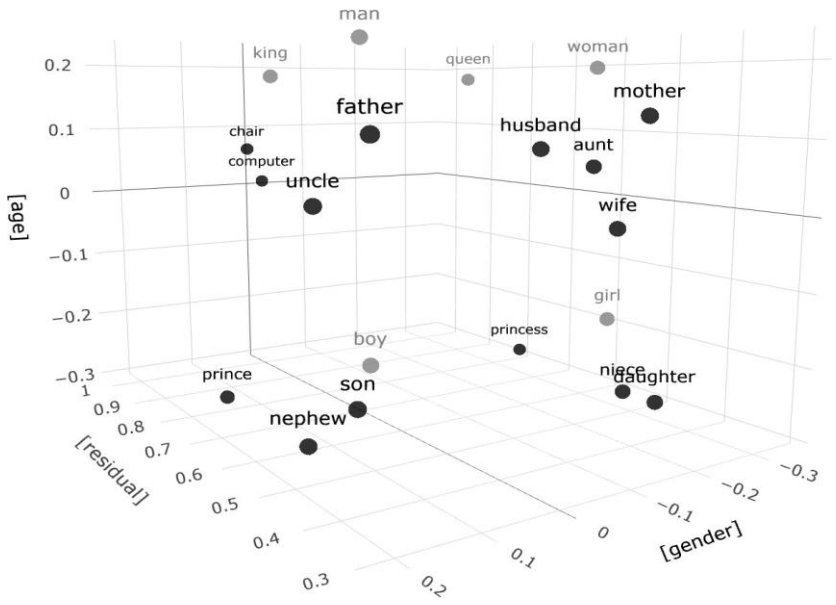


图 2： 词向量可视化

既然所有内容都能通过神经网络转换为向量， 那么通过向量进行语义搜索就变得愈发重要。向量搜索的通俗定义就是 **给定一个查询向量 (这个向量一般被称为 query)， 寻找与之最近邻 (接近) 的向量**。

目前向量搜索已经被广泛应用于如下领域：

- 推荐系统 例如在短视频平台中， 用户和短视频都被转换为高维度向量，为用户推荐视频的过程就是将用户向量作为查询向量， 寻找与之最近邻的短视频向量。
- 大语言模型 在与 ChatGPT 等大语言模型的交互过程中， 我们往往需要事先输入提示词 (Prompt)， 如果输入的提示词是" 达芬奇是谁?"， 那么大语言模型在处理过程中会首先将这段话转换成向量， 然后去知识库中查找与这个向量最近邻的向量 (例如这个向量表示的信息可能是" 达芬奇是欧洲文艺复新期间....."), 大语言模型综合上述搜索得到的附加信息进行处理，给出一个合理的回答。

- 搜索引擎　采用向量搜索进行语义搜索取代传统的关键词搜索。

向量搜索目前是学术界和工业界最热门的方向之一， 并且工业界的数据规模在十亿量级， 简单的穷举搜索不再适用于这个领域。在本次作业中， 你需要利用已有的知识设计一个合理的算法来处理十万到百万量级的向量搜索。**你可以自由地向 ChatGPT 等大语言模型求助，并在实验报告中备注， 但是抄袭行为是严格禁止的，请遵守学术道德。**

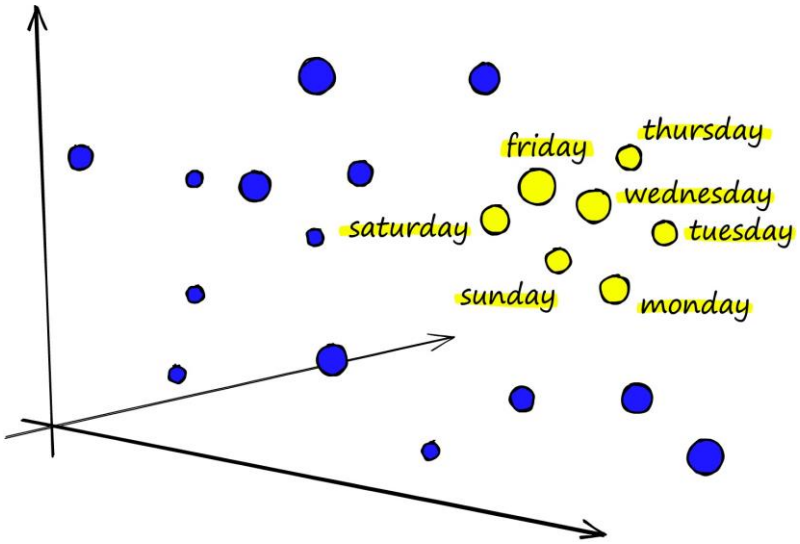


图 3： 词向量可视化

二、问题描述

在这一节中， 我们会形式化地定义向量搜索 (Nearest Neighbor Search) 这一问题：给定一个向量的集合 $X \in \mathbb{R}^{n \times d}$， 其中包含了 n 个 d 维的向量。对于一个查询向量 q， 向量最近邻搜索的目的是从 X 中找到一个向量 x^*， 满足

$$x^* = \arg\min_{x \in X} \delta(x, q)$$

即寻找 q 最近邻的向量。其中 $\delta(\cdot, \cdot)$ 表示向量之间的距离函数， 它的数学定义如下 (欧几里得距离)：

$$\delta(x, y) = \Sigma^d (x_i - y_i)^2$$

其中， x_i 和 y_i 分别表示 x 和 y 的第 i 维分量。.

如上定义可以非常简单地扩展到 K-最近邻搜索 (K-Nearest Neighbor Search)，即搜索 K 个最近邻的向量。但是当 n 和 d 的数目显著增加的时候， 精确最近邻搜索往往不再适用 (搜索成本太高)， 我们往往会在牺牲准确率的情况下， 提高搜索的速度， 这就是相似最近邻搜索 (Approximate Nearest Neighbor Search，ANN)。相似最近邻搜索的衡量指标 (召回率) 定义为

$$Recall@k = \frac{|R \cap \tilde{R}|}{k}$$

其中 R 是 q 真正的 k 个最近邻向量的集合， $\tilde{R}$ 是通过算法得到的近似最近邻向量的集合。

在本次作业中， 你需要设计一个算法来提高 ANN 搜索的效率， 我们要求你设计的算法需要保证 Recall@k 的召回率大于 90%.

输入数据的格式如下：

[输入] 向量集合 X 的大小 n， 向量的维度 d， 以及搜索最近邻的数目 k，紧接着是 n行 d 维的浮点数向量数据。然后是查询向量的个数 nq， 以及 nq 行 d 维的浮点数向量数据.

[输出] 对于每一个查询向量， 输出 k 个与之最近邻的 ID(ID 为该向量在 X 中所处的位置下标)

[示例]

[input]
5 3 2
2.3, 1.8, 4.5
3.7, 0.9, 2.1
1.5, 4.2, 3.6
2.9, 3.4, 0.7
4.1, 2.6, 1.3
2
2.3, 1.8, 4.3

3.7, 0.9, 2.5
[output]
0 2
1 4

注： n 的范围在 1w-100w 之间， d 的范围在 64-512 之间， nq 的范围在 100-1000 之间， k在 1-50 之间。

请根据本学期学习的知识，设计算法实现上述的查询功能，并尝试分析算法的空间复杂度和时间复杂度，可结合数据规模、原始数据的特性等分析查询影响因素等。

1）数据规模：包括 OJ 平台的数据和线下测试数据集（100 万级）；

2）查询任务的效率，可以统计不同的 k 值下的查询时间，例如在 k=(3，15，75，375…)时，不同数据规模下（例如：包含 200 个、4000 个、4 万个、40 万、100 万个向量的数据集）的查询时间。

3）任务说明：统计任务在不同规模数据下，不同维度、不同 k 值下的查询时间变化。查询时间和召回率的关系（可选）；或者你觉得有意义的统计指标；

4）书写要求：若采用教材内的算法实现查询，仅仅需要说明所用算法；若实践过程中涉及到自己设计的数据结构或者书本外的知识请在“实验记录和结果”中说明算法的基本思想。

5）代码提交：请在 EOJ 平台提交查询代码，将统一统计代码运行时间。注意：允许提交多次，不计罚时。

三、本地实验环境（CPU，内存，操作系统，编程语言，编译器）

CPU：13th Gen Intel(R) Core(TM) i5-1340P 1.90 GHz 16GB
Windows11 C++14 gdb

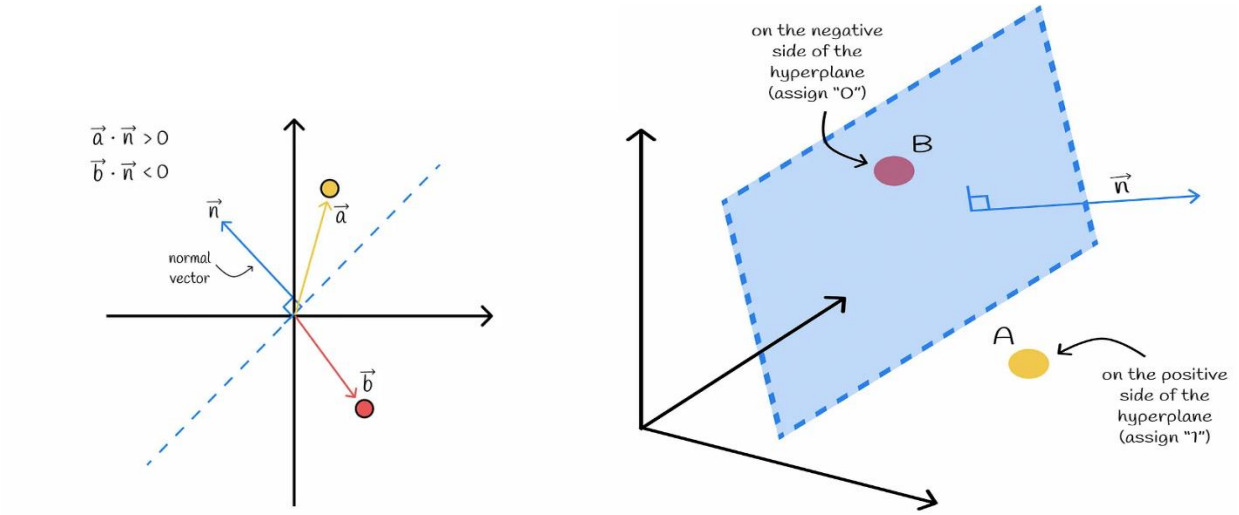
四、实验记录和结果

算法实验记录	
数据存储结构：	链式存储
查找算法：	Ensembled Hyperplane-based LSH 集成超平面局部敏感哈希
数据规模	oj test，sift small，sift
是否有课堂外的算法	是

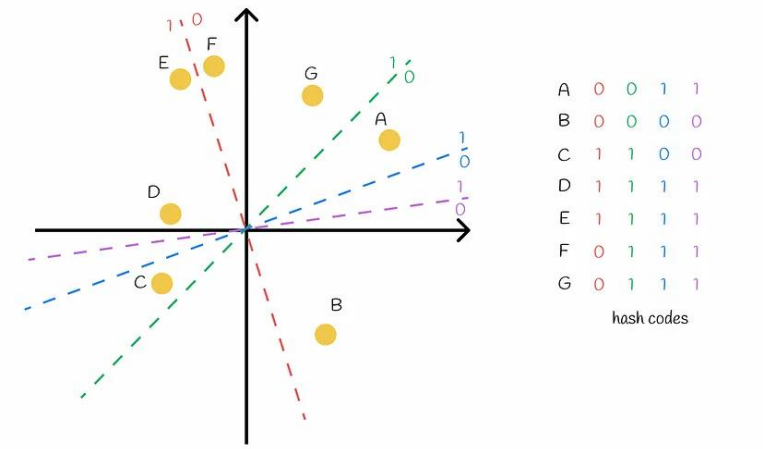
注：探索过程中未曾使用大模型，仅使用了搜索引擎。

Hyperplane-based LSH

我刚开始使用的算法是 Hyperplane-based LSH——基于超平面的局部敏感哈希以下为介绍。

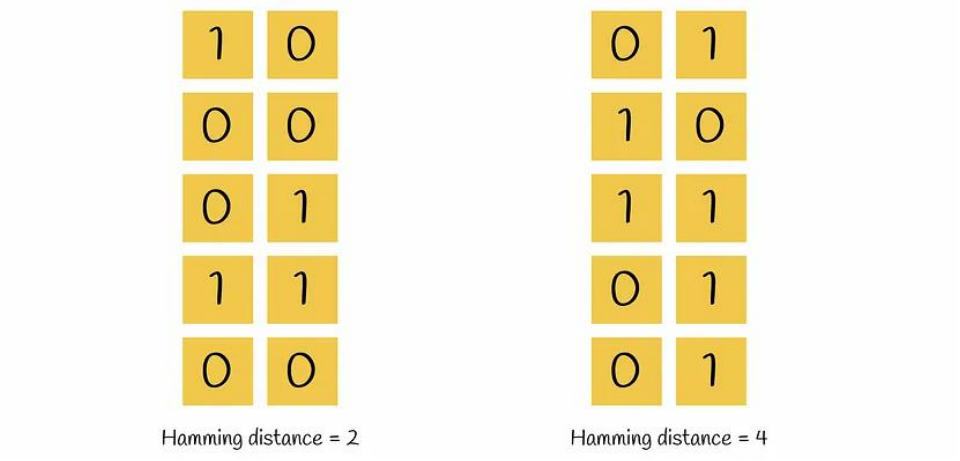


首先，在数据的高维空间中，按给定均值、方差的正态分布，生成一组随机超平面。将数据集中的embedding与这些超平面分别作内积计算。如图所示，如果该向量与超平面同向，则内积大于0，记为1，否则内积小于0，记为0。利用这个性质，我们便可以利用超平面切分高维空间。并给每个切分出的小空间分配一个二进制表示，二进制序列的长度等于超平面的个数。如下图所示。



由此，可以将数据集中的每个 embedding 划入不同的空间(即哈希桶)，用该空间的独特二进制表示来标识哈希桶。

接下来，定义两个 bitset 之间的汉明距离(Hamming Distance)。即异或值之和。汉明距离与两个 bitset 之间的差异度成正比。



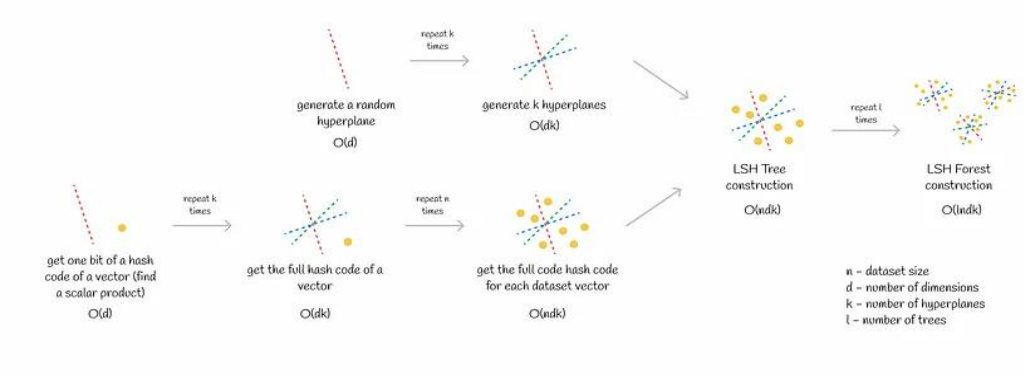
查询环节，获取 query embedding，将 query 与各个超平面作内积，获得二进制表示，转为十进制桶号，划分到对应的桶中。将该桶中所有的 embedding 作为候选集，与 query 作 L2-distance 欧几里得距离计算，pop 入一个 distance-based 的优先级队列。根据实验，通常一个桶的候选集大小不能达到很高的召回率，所以，可以根据汉明距离继续探测 Hamming distance=1,2,3,…的邻近桶，做好查询速度与召回率的 trade-off 即可。

最后，在查询的时候，根据 k 值，pop 出优先级队列存的数据即可。

以上为 vanilla hyperplane-based LSH，将作为后续实验结果的 baseline 算法。其近似时间复杂度为 O(ndk + nq*d*k + log(n/2^k))（n 为数据集大小，d 为维度，nq 为 query 个数，k 为 hyperplane 个数），前一部分为构建哈希表所需时间，中间部分为 query 划分到对应桶所需时间，后一部分为优先级队列建立近邻所需时间(未包含额外探测的候选集)，平均每个桶落入 n/2^k 个 embedding。

LSH forest/ensemble

接下来介绍 LSH forest，它的集成版本，也是我最终使用的版本。因为随机超平面的划分往往是不均匀的，就会导致漏掉一些近邻点，降低召回率，探测其他桶，又会增加时间成本。因而，通过将 query 对应的不同随机超平面组产生的哈希桶取并集，就可以获得一个范围不大但更精准的候选集。经过实验，在不需要额外探测其他汉明距离的桶的情况下，仅用 query 落入桶的集合作为候选集就能达到较高的召回率（实验中速度还有所提升, 因为待探测桶的数量很少）。近似时间复杂度为 O(lndk+nq*d*k + log(l*n/2^k)), l - number of trees 。



弊端

1. 需要调参。

   1.hyperplane的个数决定了空间划分的细致度，但实验中与召回率不一定成正比，需要和其他参数一并调试。

   2.随机超平面对应的高斯分布的均值和标准差也会带来几个点的召回率差异。在实验中，当均值和标准差与数据集一致时可以无痛提升几个点的召回率，直观上想就是因为超平面中心设在了靠近数据集中心的位置，更均匀地切分了数据。

   3.探测强度，探测不同汉明距离的桶，设定候选集阈值，需要平衡召回率和查询速度。

2. 内存成本。桶的数量n=2^k，k为hyperplane个数，呈指数级上涨，所以无法使用过多的超平面组建立forest。

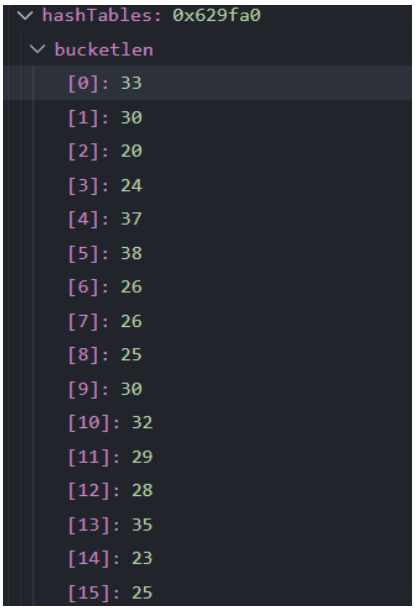
改进

使用正交超平面组

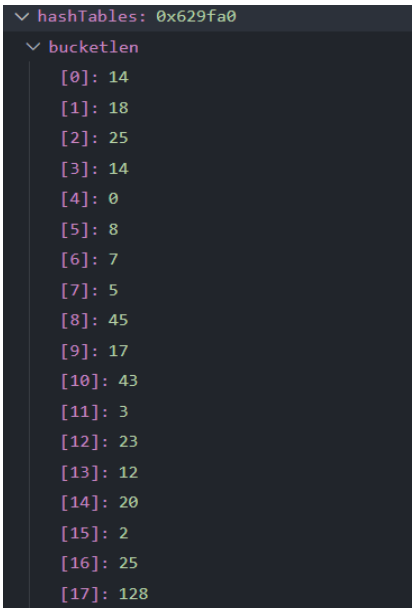
LSH的问题就在于仅用一个随机超平面组会导致空间划分不均，可能存在两个超平面之间贴的很近或很远。而LSH forest 在内存上有较大损失。那么有没有什么无痛提升召回率的方法呢？所以自然而然地想到了用正交向量集作为超平面，均匀切分空间。实验结果符合预想，通过调试观察每个桶落入的embedding个数，如下图所示，基本一致。而且在整体速度上有所提升，因为候选集的质量变高了(见实验统计表)。

正交超平面组一个简单的例子，当d=8时，可设hyperplane数为8，超平面向量分别设为(1,0,0,0,0,0,0,0), (0,1,0,0,0,0,0,0)...实验效果记录在下表。

正交向量组



随机向量组



现有问题在于当维度远大于超平面数时，如何构造出一个该维度下的正交超平面组？

由于随机生成的向量组不是线性无关的，所以就不能用**Gram-Schmidt**方法正交化处理。我在网上没有找到解决方法。

使用半精度存储,运算

经过观察，oj test的数据中，embedding的小数位数最多为10位，将小数截断到一定位数也不会影响精度，完全可以用半精度存储数据来加快一些内积，距离的计算。可以用C++ half float library来完成，但在实践中我没有尝试，只使用了float代替double。

并行思想

一开始想到了，去看了faiss，FALCONN等库的实现，4个query为一个batch,并行查找，速度更快。但时间有限，能力有限，写不出来…

尝试过的其他方法

暴力搜索：

query对每个embedding计算欧几里得距离，然后排序查询，精准近邻但是是最慢的算法。Oj上7分。

优先级队列:

在对每个embedding计算欧几里得距离后，将embedding push进优先级队列，按距离优先pop出k个近邻。精准近邻，但较慢，Oj上36分。

KDTree：

一种用于k近邻查询的空间划分树，一般用于低维数据，对于维度较高的数据，构建和查询的效率过慢。Oj上29分。主要还是受限于时间，我没有对其做剪枝操作，因为剪下来召回率损失太大…所以是精确搜索，召回率为1。

KDTree+BBF优化：

BBF（Best Bin First）是一种改进的k-d树最近邻查询算法。将“查询路径”上的节点进行排序，如按各自分割超平面（称为Bin）与查询点的距离排序。回溯检查总是从优先级最高的（Best Bin）的树节点开始。另外BBF还设置了一个运行超时限制，当优先级队列中的所有节点都经过检查或者超出时间限制时，算法返回当前找到的最好结果作为近似的最近邻。可以将KDTree拓展到高维空间。Oj上36分…在时间上有所改进。

基于点的point-based-LSH

一开始在没有找到hyperplane-based LSH方法的时候，我用了数据集的中心点来代替超平面，划分一个个”球”壳空间，但这样做的弊端很明显，空间划分极其不均，召回率自然也比较低。且构建哈希表和查询时都是直接用欧几里得距离，速度非常慢。Oj上14分。

实验结果统计						
算法	数据集	构建时间	查询时间	召回率	候选集大小	候选集质量
Hyperplane LSH (baseline)	0j test	0.678s	23.255s	0.778	1463	4.254
Orthogonal Hyperplane LSH	0j test	0.585s	18.203s(-5.05)	0.799	1156	5.529(+1.275)
LSH Forest (My Sota)	0j test	0.669s	18.095s(-5.16)	0.822	994	6.615(+2.361)
Point LSH	0j test	0.687s	683.1s(+659.8)	0.421	1524	2.210(-2.044)
KD Tree	0j test	0.817s	45.7s(+22.445)	1	-	-
KD Tree with BBF	0j test	0.806s	39.9s(+16.645)	1	-	-
Hyperplane LSH (baseline)	Sift Small	27.03s	0.827s	0.805	9166	2.196
LSH Forest (My Sota)	Sift Small	30.41s	0.561s(-0.266)	0.872	5484	3.975(+1.779)
Hyperplane LSH (baseline)	Sift	1443.5s	1813.5s	0.805	247320	3.255
LSH Forest (My Sota)	Sift	1340.6s	1367s(-446.5)	0.853	161824	5.271(+2.016)

注:

1.数据集oj test：n = 8000 d = 8 k = 30 nq = 45000

2.数据集sift small：n = 25000 d = 128 k = 100 nq = 100

3.数据集sift：n = 1000000 d = 128 k = 100 nq = 10000

4.构建时间：读入数据，构建哈希表/队列/树所需的时间

5.候选集质量 = (召回率 * n / 候选集大小) (自己发明的指标,与召回率成正比,与候选集大小成反比,同时消除了数据集大小的差异,用来衡量LSH算法的优越性) 在做召回率和速度的tradeoff时，候选集质量越高，越能保证召回率高且时间成本低。且由于LSH算法需要对数据集调参，所以召回率和时间往往不能准确反映算法优越性。但是这个指标不能跨数据集比较，因为每个数据集的搜索难度不同。

6.以上实验结果做了至少三次求平均的处理

7.关于内存占用情况并未做统计，但可以肯定的是，LSH forest占用的内存会比LSH多L倍，L为超平面组的个数。

五、结论

这是入学以来的第一次大作业，总的来说结果还算满意。过程中查了很多外文的资料，最后选择了 hyperplane-LSH，因为觉得这个方法挺有趣的，实现起来难度不大，而且潜在的改进空间也比较大。

从实验结果上分析，LSH forest 的优越性较 vanilla LSH 会随着数据集的规模上升而更加明显，更优质的候选集，更快的查询速度。唯一美中不足的就是这种方法会极大的消耗

内存资源来存储哈希表，有一定局限性。

有待解决的问题我想就在于如何构造高维的正交向量组吧，网上看到 matlab 有这种功能，但没找到实现代码，比较遗憾，不过数学上可能也需要一些高深的矩阵知识吧。随着实验的深入，我对算法效率越来越渴求，能无痛提升效率还是非常诱人的。另外一个问题就是，如果原先数据集没有对 embedding 作归一化处理到原点的话，LSH 生成向量组的时候就要调一下均值和方差，这个比较麻烦，可以考虑做一些适应数据集的功能。

收获上，对于一个问题的深入思考，实现，反思改进，再实现的过程更加熟练了。不断调试代码的 bug，调参，经历一些失败的实验，也锻炼了心态。

大作业还是养人啊,哈哈。

六、代码附录

```
#include <iostream>
#include <math.h>
#include <string.h>
#include <random>
#include <unordered_set>
#include <fstream>
#include <chrono>
#include "my_priorty_queue.h"
using namespace std;

#define maxd 130
#define maxn 25000
#define maxk 100
#define NUM_HASH_TABLES 13 //哈希表个数
#define NUM_BUCKETS 2048   //单表哈希桶个数 = 2 ^ NUM_HyperPlane
#define BUCKET_SIZE 5000   //哈希桶大小上限
#define NUM_HyperPlane 11  //超平面个数

int n, d, k;
int bitset[NUM_HASH_TABLES][NUM_BUCKETS][NUM_HyperPlane]; //embedding 的位表示
int querybitset[NUM_HyperPlane];                          //query 的位表示
bool havebit[NUM_HASH_TABLES][NUM_BUCKETS];      //用于记录该哈希桶是否标好对应位表示
//embedding
struct Point
{
    int id;
    float coordinates[maxd];
};
Point vec[NUM_HASH_TABLES][NUM_HyperPlane]; //超平面组


//数据集
struct Dataset{
    Point* points;
    int numPoints;
```

```
};

//哈希表
struct HashTable{
    int *bucketlen; //桶内元素计数
    int **bucket;
    ~HashTable();
};
HashTable::~HashTable() {
    for (int i = 0; i < NUM_BUCKETS; i++) {
        delete[] bucket[i];
    }
    delete[] bucket;
    delete[] bucketlen;
}

//记录最终结果
struct Result{
    int idx;
    float loss;
    Result();
    Result(int idx, float loss);
};
Result::Result(){
}
Result::Result(int idx, float loss){
    idx = idx;
    loss = loss;
}
bool operator < (const Result &x, const Result &y)
{
    return x.loss < y.loss;
}
bool operator >= (const Result &x, const Result &y)
{
    return x.loss >= y.loss;
}
bool operator > (const Result &x, const Result &y)
{
    return x.loss > y.loss;
}

//召回率测试
float cal_recall_k(const unordered_set<int> &gt, const unordered_set<int> &res,
size_t k) {
    float recall = 0.f;
    for (auto x: gt) {
```

```
        recall += res.count(x);
    }
    return recall / k;
}

//欧几里得距离计算
float ltwodistance(Point* point1, Point* point2){
    float sum1 = 0.0f;
    for (register int i = 0; i < d; ++i) {
        float diff = point1->coordinates[i] - point2->coordinates[i];
        sum1 += diff * diff;
    }
    return sum1;
}
//生成数据集
void generateDataset(Dataset* dataset) {
    Point* points = new Point[n+1];
    dataset->points = points;
    dataset->numPoints = n;
    for(register int i = 0; i < n; ++i) {
        points[i].id = i;
        for (register int j = 0; j < d; ++j) {
            cin >> points[i].coordinates[j];
        }
    }

    mt19937 e;
    normal_distribution <float> dis(30.0, 2000.0);// 期望为X，标准差为Y 的正态分布
    for(int idx=0;idx<NUM_HASH_TABLES;idx++){
        for(register int j=0;j<NUM_HyperPlane;j++){
            for(register int i=0;i<d;++i){
                vec[idx][j].coordinates[i] = dis(e);
            }
        }
    }
}

//生成哈希桶的位表示，并将数据集中的embedding 归入对应桶中
float BitGenerate(Point* point1, int tableIndex) {
    float sum = 0.0f;
    int bitrecord[NUM_HyperPlane];
    for(register int p=0;p<NUM_HyperPlane;p++){
        float sumset = 0.0f;
        for (int i = 0; i < d; ++i) {
            float diff = point1->coordinates[i] *vec[tableIndex][p].coordinates[i];
            sumset += diff;
        }
```

```
        if(sumset>=0){
            bitrecord[p] = 1;
            sum += pow(2, NUM_HyperPlane-1-p);
        }else{
            bitrecord[p] = 0;
        }
    }
    int bucket = (int)sum % NUM_BUCKETS;
    if(!havebit[tableIndex][bucket]){
        for(register int i=0;i<NUM_HyperPlane;i++){
            bitset[tableIndex][bucket][i] = bitrecord[i];
        }
        havebit[tableIndex][bucket] = true;
    }
    return sum;//二进制表示对应的十进制即为桶号
}

unsigned int hashFunction(Point* point, int tableIndex) {
    unsigned int hash = 0;
    hash = (unsigned int)BitGenerate(point, tableIndex);
    return hash;
}
//建哈希表
void buildHashTable(Dataset* dataset, int tableIndex, HashTable
hashTables[NUM_HASH_TABLES]) {
    for (int i = 0; i < NUM_HASH_TABLES; i++){
        HashTable* hashTable = &hashTables[i];
        hashTable->bucketlen = new int[NUM_BUCKETS+10]();
        hashTable->bucket = new int*[NUM_BUCKETS+10];       //分配行
        for (int i = 0; i < NUM_BUCKETS+10; i++) {
            hashTable->bucket[i] = new int[BUCKET_SIZE+10]; //分配列
        }
    }
    //初始化为0
    for(int i=0;i<NUM_BUCKETS;i++){
        hashTables[tableIndex].bucketlen[i] = 0;
        for(int j=0; j < BUCKET_SIZE ; j++){
            hashTables[tableIndex].bucket[i][j] = 0;
        }
    }
    //为embedding 分配桶，桶内元素计数
    for (register int i = 0; i < dataset->numPoints; ++i) {
        unsigned int hash = hashFunction(&dataset->points[i], tableIndex);
        hashTables[tableIndex].bucket[hash][hashTables[tableIndex].bucketlen[hash]]
= dataset->points[i].id;//在桶的末尾插入
        hashTables[tableIndex].bucketlen[hash]++;
    }
```

```
}

void searchNearestNeighbors(Dataset* dataset, Point* query, HashTable
hashTables[NUM_HASH_TABLES], unordered_set<int> &finalset) {
    List<Result> result; //优先级队列
    result.build_heap();

    int* record = new int[maxn]; //用于去重，记录 embedding 是否已在候选集中
    for(int i=0;i<n;i++){
        record[i] = 0;
    }
    for(int num=0; num < NUM_HASH_TABLES; ++num){
        int query_sum = 0;
        for(register int p = 0; p < NUM_HyperPlane; p++){
            float sumset = 0.0f;
            for (register int i = 0; i < d; ++i) {
                float diff = query->coordinates[i] * vec[num][p].coordinates[i];
                sumset += diff;
            }
            if(sumset>=0){
                querybitset[p] = 1;
                query_sum += pow(2, NUM_HyperPlane-1-p);
            }else{
                querybitset[p] = 0;
            }
        }
        // hamming distance == 0
        int hash = (int)query_sum;
        for(register int i=0;i<hashTables[num].bucketlen[hash];++i){
            int id = hashTables[num].bucket[hash][i];
            if(record[id]==0){
                Result r1 = Result();
                r1.idx = id;
                float distance = ltwodistance(query, &dataset->points[r1.idx]);
                r1.loss = distance;
                result.insert(result.size(), r1);
                record[id] = 1;
            }
        }
        // hamming distance == 1
        if(num < NUM_HASH_TABLES/2){
            int candidates[NUM_HyperPlane];
            //计算距离为1 时位表示对应的桶
            for(int i=0;i<NUM_HyperPlane;i++){
                if(querybitset[i]==1){
                    candidates[i] = query_sum - pow(2, NUM_HyperPlane-1-i);
                }else{
```

```
                    candidates[i] = query_sum + pow(2, NUM_HyperPlane-1-i);
                }
            }
            //插入候选集
            for(int i=0;i<NUM_HyperPlane;i++){
                int candidate = candidates[i];
                for(register int j=0;j < hashTables[num].bucketlen[candidate];++j){
                    int id = hashTables[num].bucket[candidate][j];
                    if(record[id]==0){
                        Result r1 = Result();
                        r1.idx = id;
                        float distance = ltwodistance(query,
&dataset->points[r1.idx]);
                        r1.loss = distance;
                        result.insert(result.size(), r1);
                        record[id] = 1;
                    }
                }
            }
        }
    }
    result.heap_sort();
    //cout<<"候选集大小："<<result.size()<<endl;
    int rr = 0;
    while(finalset.size()<k){
        finalset.insert(result.entry[rr].idx);
        rr++;
    }
}
```