实 验 报 告

实验 13：位置敏感哈希算法（LSH&NN）

517030910374
郭嘉宋

目录：

注：在编程过程中我也遇到了很多困难和 Bug，还有很多思考了很久得出的结论与解释，以及一些需要注释的内容，都在下述 “2. 实验过程” 中，用加粗的 NOTE 加以了说明。

正文:

1. 实验准备

(1) 实验环境

本实验主要采用 Ubuntu 系统下的 python2.7 进行，使用 miniconda 环境，Ubuntu 系统安装在 VMware Workstation 提供的虚拟机中，同时使用 OpenCV、Numpy 等库对图像进行分析与后续处理。

(2) 实验目的

实验 13:

了解 LSH 和 NN 算法原理，体会哈希检索带来的复杂度降低与效率提升。尝试自己实现 LSH 和 NN 算法，对图片内容进行检索，完善图片搜索功能。

(3) 实验原理

实验 13:

LSH(Location Sensitive Hash)，即位置敏感哈希函数。与一般哈希函数不同的是位置敏感性，也就是散列前的相似点经过哈希之后，也能够在一定程度上相似，并且具有一定的概率保证。

形式化定义:

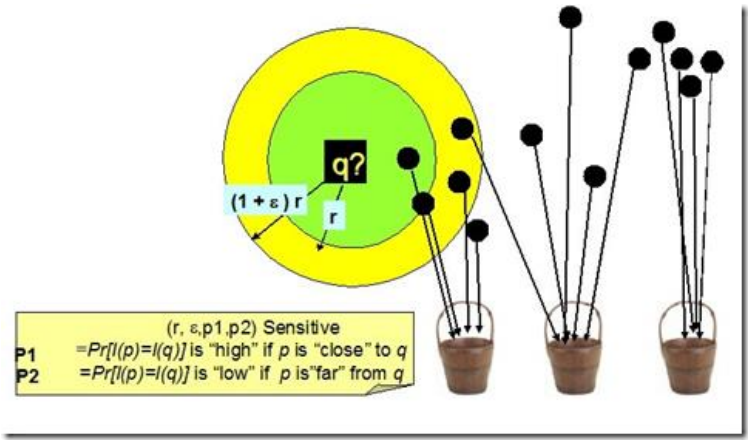
对于任意 q, p 属于 S，若从集合 S 到 U 的函数族 H={h1,h2...hn}对距离函数 D，如欧式距离、曼哈顿距离等等，满足条件:

若D(p,q) <= r,且Pro[h(p) = h(q)] >= p1

若D(p,q) > r(1+ε)，且Pro[h(p) = h(q)] <= p2

则称 D 是位置敏感的。

如下图，空间上的点经位置敏感哈希函数散列之后，对于 q，其 rNN 有可能散列到同一个桶（如第一个桶）,即散列到第一个桶的概率较大，会大于某一个概率阈值 p1；而其(1+emxilong) rNN 之外的对象则不太可能散列到第一个桶，即散列到第一个桶的概率很小，会小于某个阈值 p2.



LSH 的作用:

◆高维下近似查询

相似性检索在各种领域特别是在视频、音频、图像、文本等含有丰富特征信息领域中的应用变得越来越重要。丰富的特征信息一般用高维向量表示，由此相似性检索一般通过 K 近邻或近似近邻查询来实现。一个理想的相似性检索一般需要满足以下四个条件:

1. 高准确性。即返回的结果和线性查找的结果接近。

2. 空间复杂度低。即占用内存空间少。理想状态下，空间复杂度随数据集呈线性增长，但不会远大于数据集的大小。

3. 时间复杂度低。检索的时间复杂度最好为 O（1）或 O（logN)。

4. 支持高维度。能够较灵活地支持高维数据的检索。

传统主要方法是基于空间划分的算法——tree 类似算法, 如 R-tree, Kd-tree, SR-tree。这种算法返回的结果是精确的，但是这种算法在高维数据集上的时间效率并不高。实验[1]指出维度高于 10 之后，基于空间划分的算法时间复杂度反而不如线性查找。LSH 方法能够在保证一定程度上的准确性的前提下，时间和空间复杂度得到降低，并且能够很好地支持高维数据的检索。

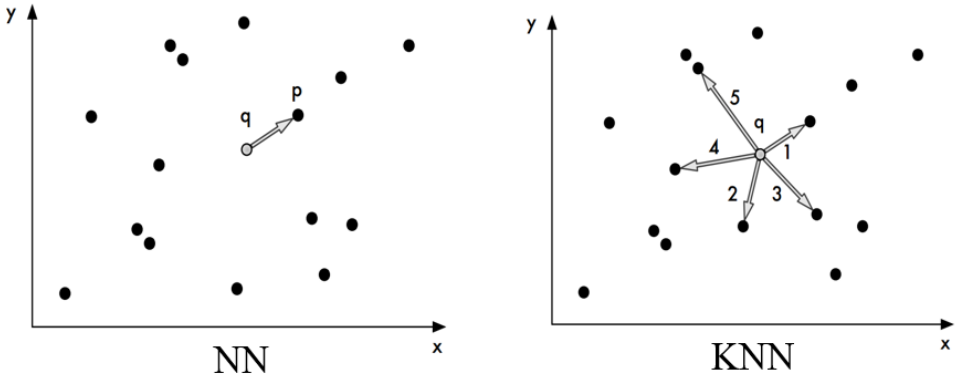
◆分类和聚类

根据 LSH 的特性，即可将相近（相似）的对象散列到同一个桶之中，则可以对图像、音视频、文本等丰富的高维数据进行分类或聚类。

◆数据压缩。如广泛地应用于信号处理及数据压缩等领域的 Vector Quantization 量子化技术。

1. Why use LSH?

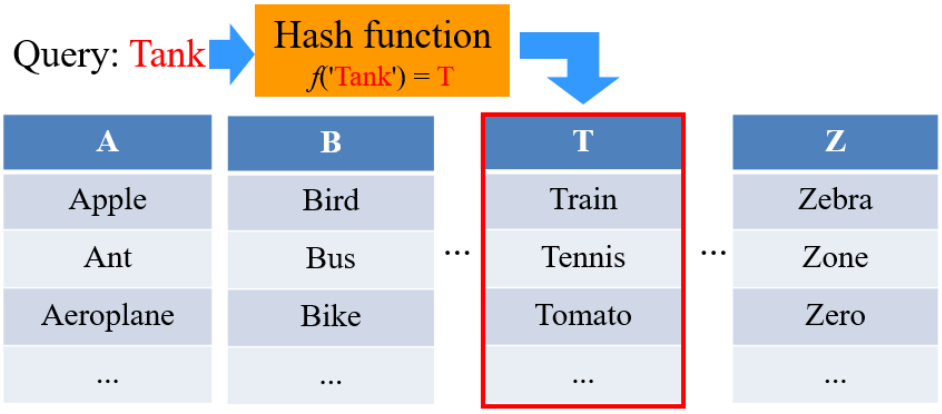
用Nearest neighbor (NN) 或k-nearest neighbor (KNN)在数据库中检索和输入数据距离最近的1个或k个数据，一般情况下算法复杂度为O(n)（例如暴力搜索），优化情况下可达到O(log n)（例如二叉树搜索），其中n为数据库中的数据量。当数据库很大（即N 很大时），搜索速度很慢。



Hashing的基本思想

Hashing的基本思想是按照某种规则（Hash函数）把数据库中的数据分类，对于输入数据，先按照该规则找到相对应的类别，然后在其中进行搜索。由于某类别中的数据量相比全体数据少得多，因此搜索速度大大加快。

一个查字典的类比：



数据的表示

•数据(图像、视频、音频等)都表示成一个d维的整数向量

$$\boldsymbol{p}=(p_1,p_2,\cdots p_d)$$

•其中p_i是整数，满足 $0\le p_i\le C$，这里C是整数的上限。

•在本实验中，每幅图像用一个12维的颜色直方图$\boldsymbol{p}$表示，构成方式如右图所示。其中 $H_i, i=1,2,3,4$

•是3维颜色直方图。

•特征向量的量化

•上述得到的特征向量 $\boldsymbol{p}=(p_1,\cdots p_{12})$

•每个分量满足 $0\le p_j\le 1$ 将其量化成

•3个区间分别用0 1 2表示：

$$p_j=\begin{cases}0, \text{if } 0\le p_j<0.3\\ 1, \text{if } 0.3\le p_j<0.6\\ 2, \text{if } 0.6\le p_j\end{cases}$$

也可以用别的量化方法，目的是使0 1 2的分布尽可能平均

•于是最终得到的特征向量的每个元素满足 $p_j\in\{0,1,2\}$

LSH预处理

d维整数向量$\boldsymbol{p}$可用$d'=d*C$维的Hamming码表示：

$$v(\boldsymbol{p})=\text{Unary}_C(p_1)\cdots\text{Unary}_C(p_d)$$

其中 $\text{Unary}_C(p_1)$ 表示C个二进制数，前p1个为1，后C-p1个为0。如当C=10：

$$\text{Unary}_C(5)=1111100000 \qquad \text{Unary}_C(3)=1110000000$$

如$\boldsymbol{p}$=(0,1,2,1,0,2)，这里d=6,C=2,于是

$$v(\mathbf{p})=00\,10\,11\,10\,00\,11$$

选取集合 {1, 2, …, d'} 的L个子集 $\{I_i\}_{i=1}^{L}$，定义v($\boldsymbol{p}$)在集合

$$I_i=\{i_1,i_2,\cdots i_m\}: 1\le i_1<i_2<\cdots<i_m\le d'$$

上的投影为 $g_i(\boldsymbol{p})=p_{i1}p_{i2}\cdots p_{im}$，其中$p_{ij}$为v($\boldsymbol{p}$)的第$i_j$个元素。对于上述$\boldsymbol{p}$，它在{1,3,7,8}上的投影为(0,1,1,0)

哈希函数计算

•不必显式的将d维空间中的点$\boldsymbol{p}$映射到d'维Hamming空间向量v($\boldsymbol{p}$)。

•I|i表示I中范围在(i-1)*C+1~i*C中的坐标：

$$I = \{1,3,7,8\}, I|1 = \{1\}, I|2 = \{3\},$$
$$I|3 = \phi, I|4 = \{7,8\}, I|5 = I|6 = \phi$$

•v($\boldsymbol{p}$)在I上的投影即是v($\boldsymbol{p}$)在I|i(i=1,2,…,d)上的投影串联，v($\boldsymbol{p}$)在I|i上的投影是一串1紧跟一串0的形式，需要求出1的个数：

$$|\{I|i\} - C * (i-1) \le x_i|$$

•比如$\{I|1\}$中小于等于$x_1 = 0$的个数为0，投影：0；

•$\{I|2\} - 2$中小于等于$x_2 = 1$的个数为1，投影：1；

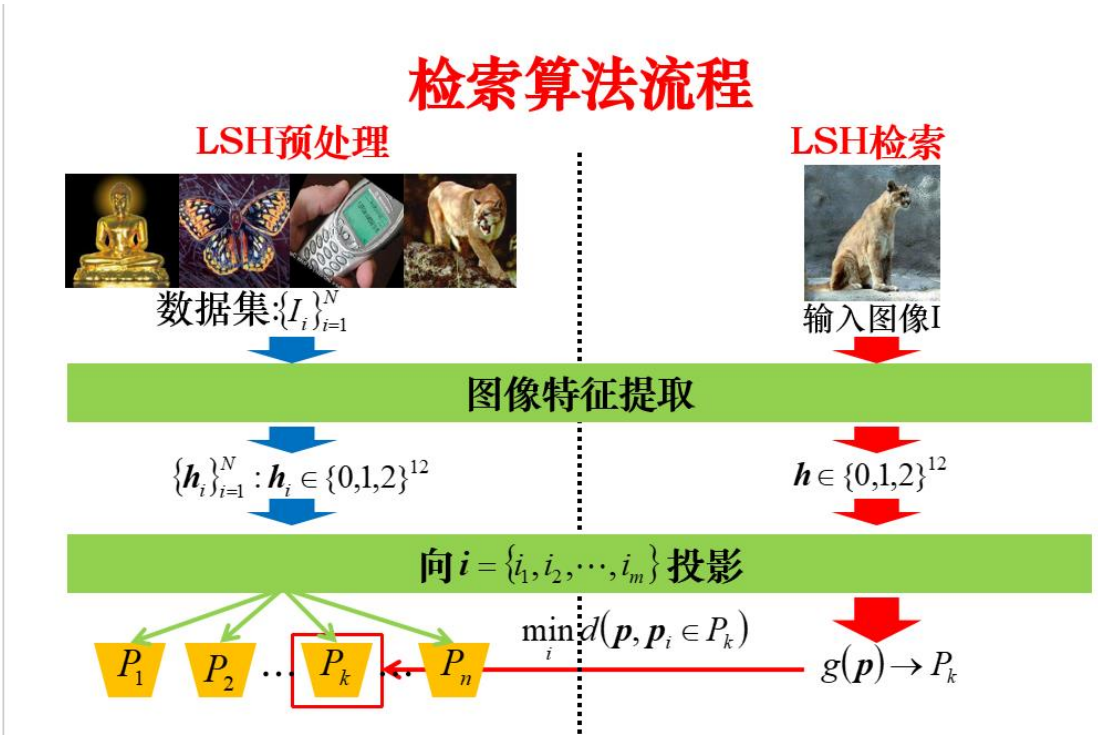
•$\{I|4\} - 3 * 2$中小于等于$x_4 = 1$的个数为1，投影：10；

•串联得到：(0,1,1,0)

LSH检索

g($\boldsymbol{p}$)被称作Hash函数，对于容量为N的数据集$P = \{\boldsymbol{p}_i\}_{i=1}^{N}$，g($\boldsymbol{p}$)可能的输出有n个，n远小于N，这样就将原先的N个数据分成了n个类别，其中每个类别中的数据具有相同的Hash值，不同类别的数据具有不同的Hash值。

对于待检索的输入$\boldsymbol{p}$，先计算g($\boldsymbol{p}$)，找到其对应的类别，然后在该类别的数据集中进行搜索，速度能够大大加快。



2. 实验过程

1.1 实验步骤

首先我定义了一个函数 colorHistogram，用来求得图片中每 1/4 区的 RGB 颜色比例，储存在'BGR'这个 list 中返回，然后又定义了函数 extractFeature，把整幅图片的完整向量信息保存在 feature 中返回，如下图所示：

```
def colorHistogram(img, init_row, init_col):
    ROW, COL, CLR = img.shape
    BGR = [0] * 3
    for i in range(init_row, init_row + ROW / 2):
        for j in range(init_col, init_col + COL / 2):
            BGR[0] += img[i][j][0]
            BGR[1] += img[i][j][1]
            BGR[2] += img[i][j][2]
    total = BGR[0] + BGR[1] + BGR[2]
    for c in range(3):
        BGR[c] = float(BGR[c]) / total
    return BGR
```

```
def extractFeature(img):
    ROW, COL, CLR = img.shape
    feature = colorHistogram(img, 0, 0) + colorHistogram(img, 0, COL / 2) \
  + colorHistogram(img, ROW / 2, 0) + colorHistogram(img, ROW / 2, COL / 2)
    return feature
```

接下来就是哈希转换哈数 hashFeature，我将原先的图片完整信息向量 raw_feature 传入函数，mask 为需要映射到的哈希集合，将 RGB 分部比例转换为 0、1、2 的整数，通过哈希函数归类到 val 类中，最后返回类名 val，如下图所示：

```
def hashFeature(raw_feature, mask):
    feature = [0] * 12
    for i in range(12):
        if raw_feature[i] < 0.3:
            feature[i] = 0
        elif raw_feature[i] < 0.6:
            feature[i] = 1
        else:
            feature[i] = 2
    result = []
    for n in mask:
        C = 2
        i = n / C
        result.append(n % C < feature[i])
    val = 0
    for i in range(len(result)):
        val += result[i] * (2 ** i)
    return val
```

NOTE：我的哈希函数先求了投影结果，最后将权重求和归类。即我将 C 取为 2，先将 mask 投影向量的每个分量的对应投影数值求出，如上图所示，求出 n 个投影分量对应的数值，储存在 result 中，因为投影结果只可能是 0 或 1，之后我将每一位视作二进制位，以相应位权求和，最终得到一个可以唯一表示投影结果的十进制数 val，将 val 作为哈希类的类名传出函数。

之后就是主函数了，将文件夹 dataset、目标图片 target 和投影向量 mask 传入函数，稍作异常处理，若输入的投影分量不是整数则进行异常处理，取整即可，如下图所示：

```
target = cv2.imread(sys.argv[1])
# target=cv2.imread('target.jpg')
dataset = sys.argv[2]
# dataset='Dataset'
mask = sys.argv[3:]
# mask=[4,12,20]

for i in range(len(mask)):
    mask[i] = int(mask[i])
```

为了对比 LSH 和 NN 的算法效率，我在预处理时新建了 2 个字典 lib1 和 lib2，lib1 中储存了哈希操作后的对应哈希类名和每个类下的对应图片，而 lib2 直接将图片名作为类，并不经过哈希函数操作，如下图所示：

```
lib1 = {}
lib2 = {}
for root, dirs, files in os.walk(dataset):
    for filename in files:
        img = cv2.imread(os.path.join(root, filename))
        feature = extractFeature(img)
        hash_val = hashFeature(feature, mask)
        lib2[filename] = feature
        d = lib1.get(hash_val, {})
        d[filename] = feature
        lib1[hash_val] = d
```

预处理工作完成，之后再对搜索图片进行搜索，先 LSH 算法搜索，求出哈希值，这里我把搜索准确的最小距离 min_d 取为 4，即若所有图片都超过 4 则没有找到，再比对同一个哈希类中距离最小的点，完整的 LSH 搜索如下图所示：

```
t0 = time.clock()
for t in range(50):
    feature = extractFeature(target)
    hash_val = hashFeature(feature, mask)
    d = lib1[hash_val]
    min_d = 4
    name = ''
    for key, value in d.items():
        dis = 0
        for i in range(12):
            dis += (value[i] - feature[i]) ** 2
        dis = dis ** 0.5
        if dis < min_d:
            min_d = dis
            name = key
t1 = time.clock()
print name
print 'LSH:', (t1 - t0)/50
```

NOTE：程序中我对距离的定义是，两个向量差值的模，如上图，最后的 dis 即为相对距离的大小。

NOTE：因为每次程序执行的时间具有随机性，总有小范围波动，为了更精准的比较，我令程序先后搜索 50 次，最后输出的时间是平均搜索的时间。

之后就是普通的 NN 算法搜索，即在没有哈希分类的 lib2 中直接搜索，如下图所示，我直接比较了两个向量的距离，以 min_d=11 作为最小距离，只要小于距离 11 的点都判定为可选项，再寻找其中距离最小的点，如下图所示：

```
t0 = time.clock()
for t in range(50):
    feature = extractFeature(target)
    min_d = 11
    name = ''
    for key, value in lib2.items():
        dis = 0
        for i in range(12):
            dis += (value[i] - feature[i]) ** 2
        dis = dis ** 0.5
        if dis < min_d:
            min_d = dis
            name = key
t1 = time.clock()
print name
print 'NN:', (t1 - t0)/50
```

NOTE：因为每次程序执行的时间具有随机性，总有小范围波动，为了更精准的比较，同 LSH 一样，先后搜索 50 次，最后输出的时间是平均搜索的时间，name 即为最佳匹配图。

1.2 实验结果：

1. 取投影集合为 {4、12、20}，在 Dataset 中搜索图片 target.jpg。

```
guo@ubuntu:~/PycharmProjects/lab13$ python lab13.py target.jpg Dataset 4 12 20
38.jpg
LSH: 0.30003504
38.jpg
NN: 0.30868156
```

LSH 和 NN 都找到了正确的图片，LSH 的时间略短，每次短 0.009s 左右，如果进行 100 次试验则快了 0.9s。

2. 取投影集合为 {2、8、10、18}，在 Dataset 中搜索图片 target.jpg。

```
guo@ubuntu:~/PycharmProjects/lab13$ python lab13.py target.jpg Dataset 2 8 10 18
38.jpg
LSH: 0.24403766
38.jpg
NN: 0.25524176
```

LSH 和 NN 都找到了正确的图片，LSH 的时间更短，且短的时间更多，每次快 0.011s，如果进行 100 次试验则快了 1.1s。

3. 取投影集合为 {1、6、10、14、18、22}，在 Dataset 中搜索图片 target.jpg。

```
guo@ubuntu:~/PycharmProjects/lab13$ python lab13.py target.jpg Dataset 1 6 10 14 18 22
38.jpg
LSH: 0.23401974
38.jpg
NN: 0.24929352
```

LSH 和 NN 都找到了正确的图片，LSH 的时间更短，且短的时间再次增加，每次快 0.015s，如果进行 100 次试验则快了 1.5s。

4. 取投影集合为 {1、5、10、12、14、18、20、22}，在 Dataset 中搜索图片 target.jpg。

```
guo@ubuntu:~/PycharmProjects/lab13$ python lab13.py target.jpg Dataset 1 5 10 12 14 18 20 22
38.jpg
LSH: 0.2609125
38.jpg
NN: 0.27383642
```

LSH 和 NN 都找到了正确的图片，LSH 的时间更短，但我们发现，LSH 的时间反而较上次增加，不像之前的递减。

NOTE：我们发现 NN 算法每次的时间也不一样，但是理论分析应该完全相同，查阅资料后发现这是计算机系统硬件运算的随机误差，比如锁存器中已经有大量数导致之后的 NN 也变慢了，但是 NN 的变化很小，在误差范围之内。同时因为在虚拟机中进行运算，导致系统误差更大了一些，NN 算法时间有波动也是正常情况。

1.3 OpenCV 自带库函数 LSH 与 NN 效率对比

Cv2 中的 LSH、NN 算法使用如下：

```
sift =cv2.xfeatures2d.SIFT_create()
kp1, des1 = sift.detectAndCompute(img1, None)
min=100
answer='none'
t0=time.clock()
for root, dirs, files in os.walk(dataset):
    for filename in files:
        img2 = cv2.imread(os.path.join(root, filename),cv2.IMREAD_COLOR)
        kp2, des2 = sift.detectAndCompute(img2, None)
        bf = cv2.BFMatcher(cv2.NORM_L1, crossCheck=True)
        matches = bf.match(des1, des2)
        re = matches[0].distance
        if re<min:
            min=re
            answer=filename
t1=time.clock()
```

```
t0=time.clock()
kp1, des1 = orb.detectAndCompute(img1, None)
min=100
answer='none'
for root, dirs, files in os.walk(dataset):
    for filename in files:
        img2 = cv2.imread(os.path.join(root, filename),cv2.IMREAD_COLOR)
        kp2, des2 = orb.detectAndCompute(img2, None)
    # print(kp1)
        bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
        matches = bf.match(des1, des2)
        re = matches[0].distance
        if re<min:
            min=re
            answer=filename
t1=time.clock()
```

NOTE：为使用 cv2.xfeatures2d.SIFT_create()函数需要新安装库名为 opencv-contrib-python，才可以使用此函数得到 sift 关键点。

结果如下:

```
/home/guo/miniconda2/bin/python /home/guo/PycharmProjects/lab13/test.py
NN:
distance: 0.0
best picture: 38.jpg
0.81253
LSH:
distance: 0.0
best picture: 38.jpg
0.690063
```

结果表明 LSH 依旧快于 NN 很多，同我的实验结果相同，实验成功。

3. 实验总结

映射集合的维度越大，分类的种类越多，每一个类别的样本就更少，计算哈希值所需要时间更长。理论上 LSH 检索所需要的时间大致等于计算目标的特征向量的时间 + 计算哈希值的时间 + 在一个类别中找出与目标的特征向量距离最小的特征向量的时间，所以映射集合的维度太大或者太小都不利于缩短检索时间。在上述 1.2 实验结果中可以由我的实验结果看出。时间和哈希映射表的长度的增加大致呈二次函数关系，先减少后增加。所以要合理选择映射集合的长度，需要不大不小适中最好，可由调参实现。

同时这次试验 LSH 相比 NN 的快速并没有怎么提现，是因为数据库中仅有 40 张图片，我自己实验的时候对每张图片信息提取都比较基础，效率差距体现不出来。

系统自带的两个函数都提取了关键点，信息量更大，所以时间差距更加明显。

此次试验让我对 OpenCV 的了解更深了，上学期计算导论使用过 OpenCV 制作过人脸识别，对 OpenCV 的有一定的认识，这次试验让我学习到了哈希算法对复杂度的优化，也让我知道了很多时候程序并不是可以一次写好的，需要后序调整参数才能获得最好的运算效率，而这个最佳点往往是第一次写的时候无法估算的，只有一次一次的调整尝试才可以找到，同时希望在以后的学习过程中继续深入了解图像处理技术。

注：在编程过程中我也遇到了很多困难和 Bug，还有很多思考了很久得出的结论与解释，以及一些需要注释的内容，都在下述“2. 实验过程”中，用加粗的 NOTE 加以了说明。源码见附带文件即可，lab13_1.py 是我自己写的 LSH 和 NN 算法，而 lab13_2.py 是对现有 cv 库内函数的使用。

继续努力！

o(*≧▽≦)ツ！

517030910374

郭嘉宋