实验 12-LSH

姓名：王煌基　　学号：519030910100　　班级：F1903004

一、实验原理

将高维空间中的元素视为点并赋以坐标值，坐标值为正整数。通过一族哈希函数 将空间所有点映射到 n 个哈希表T_i中，$n = |F|$，即每个哈希函数$f \in F$对应一个哈希表，每个哈希表都存放着空间所有的点。对于给定的查询子 q，分别计算$f_1(q)$、$f_2(q)$、…、$f_n(q)$，$f_i \in F$，i=1,2,…,n 。以所有$f_i(q)$落入的哈希表T_i中的桶中所有点作为候选集，比较其与 q 之间的距离，选出距离最近的 K 个点（K-NNS）

二、实验环境

1.个人笔记本电脑

2.操作系统：windows10 专业版

3.使用软件：Visual Studio Code；Docker Desktop

三、实验算法

1. 图像特征向量的获取：利用颜色直方图来获取图像特征向量，并且将其量化成 0、1、2 的类型以使得 0、1、2 尽可能均匀分布；

2. 利用 Hamming 法或者 Hash 法计算得到在 g 方向上的投影并保存；

3. 对搜索库中的图像进行索引的建立，并保留其特征向量集以及 Hash 桶；

4. 利用 LSH 或(K)NN 算法检索图片，进行图片的查找，并保存结果。

四、实验流程

练习一：利用 LSH 算法在图片数据库中搜索与目标图片最相似的图片。自行设计投影集合，尝试不同投影集合的搜索的效果。对比 NN 与 LSH 搜索的执行时间、搜索结果。

【解】

本次实验的流程可以按照实验算法的步骤一步一步来进行，以简化实验难度。

①图像特征向量的获取：利用颜色直方图来获取图像特征向量，并且将其量化成 0、1、2 的类型以使得 0、1、2 尽可能均匀分布

这里，每副图像均用一个 12 维的颜色直方图 P 来表示，从而可以得到相应的特征向量，并且为了方便表达，可以将 P 内部的元素限制在 0、1、2 之中，而且尽可能使他们平均，借助之前实验中所学的颜色直方图的知识，我们可以较为容易地写出相应的代码块，这里将这个函数取名为 color，是为了得到 filename.jpg 的特征向量。下面是简要代码：

```
def color(filename):
    # 读入图并且进行 RGB、BGR 的转化
    # 需要注意是 4 块区域，分为 BGR
    blue, green, red = [0,0,0,0], [0,0,0,0], [0,0,0,0]
    # 宽 len(image),长 len(image[0])，读入的图片按照 r,g,b 的顺序
    # 分成四块来进行计算 blue、green、red 并且存入 p 向量中
    # 归一化处理
    # 计算出 0、1、2 类型的特征向量
    for i in range(len(p)):
        if p[i] < 0.3:
            p[i] = 0
        elif p[i] < 0.6:
            p[i] = 1
        else:
            p[i] = 2
    return p
```

②利用 Hamming 法或者 Hash 法计算得到在 g 方向上的投影并保存；

LSH 可以利用 Hamming 空间映射或者哈希函数族的映射方法，需要对其进行分别实现。

i)Hamming 空间映射：

d 维整数向量 p 可用 d’=d*C 维的 Hamming 码表示，通过对编码的分析可以发现，由于 p 是 12 维的向量，构造 v(p)需要 24 维的数组，并且，由于当前位置只有 00,10,11 三种情况（对应二进制的 0、1、2），第一二位分别对应 2*i,2*i+1，因此我们可以快速的得出 p 向量在 Hamming 空间映射中对 g 向量的映射值的实现如下：

```
# 计算 Hamming 码来获取投影，其中 g 是需要被投影的方向
def Hamming(p,g):
    # 由于 p 是 12 维的向量，构造 v(p)需要 24 维的数组
    lsh = []
    v = [0] * 24
    for i in range(12):
        # 当前位置只有 00,10,11 三种情况，第一二位分别对应 2*i,2*i+1
        if p[i] == 1:
            v[2*i] = 1
        if p[i] == 2:
            v[2*i] = 1
            v[2*i+1] = 1
    # eg.v(p)=001011100011
    # 对于上述 p，它在{1,3,7,8}上的投影为(0,1,1,0)
    for i in g:
        lsh.append(v[i-1])
    return lsh
```

通过这样的方式，我们就可以得到 Hamming 法下得到的投影向量。

ii)哈希函数族的映射方法：

这里通过定义 I={1,2,…,d’ }，且定义 p|I 为向量 p 在坐标集 I 上的投影，即以坐标集 I 中每个坐标为位置索引，取向量 p 对应位置的比特值并将结果串联起来。我们将空间 P 中的所有点利用哈希函数族 都存入哈希表的哈希桶中。由于哈希桶可能很大，可以通过对某一个桶“扩容”的方式来存下大量的数据，并且提高访问的效率

```
# 哈希函数计算 lsh
def hash(p,g):
    # I 是 Hash 桶
    I  = [[] for i in range(12)]
    # print(I)
    lsh = []
    # I|i 表示 I 中范围在(i-1)*C+1~i*C 中的坐标：
    for i in g:
        # print((i-1)//2)
        I[(i-1)//2].append(i)
    for i in range(12):
        if I[i] != []:
            for Ii in I[i]:
                if Ii-2*i <= p[i]:
                    lsh.append(1)
                else:
                    lsh.append(0)
    return lsh
```

通过这样的方式，我们实现了哈希函数族的映射法。

③对搜索库中的图像进行索引的建立，并保留其特征向量集以及 Hash 桶（即预处理）；

实际上，在前面两个步骤的准备基础之上，这一个步骤是水到渠成的。

我们最后希望在预处理结束后我们能得到哈希的桶以及每张图的特征，因此只需要按照：读入图片->计算图片的特征向量 p->将其投影到 g 上形成 LSH 向量，将其封装至哈希桶中方便 O（n）时间的查找，根据这个思路，可以写出预处理的函数：

```
# 对搜索库中的图像（共 40 张，均以 i.jpg 为文件名）建立索引
def pre(g):
    Hash, Hash_id, character = [], [], []
    for i in range(1,41):
        # 读图处理
        # 获取该图片的特征向量并保存到 character 内
        character.append(p)
        # 可用 I 的方式获取投影
        # 也可以用汉明法获取投影即 lsh = Hamming(p,g)
        lsh = hash(p,g)
        # 需要注意的是，由于循环的时候文件用的是 1.jpg->40.jpg
        # 然而数组 append 只能是从 0 开始，因而在具体实现的时候需要-1，同样回答了输出的+1 原因
        # 尝试将 LSH 扔入哈希桶中，并且保存
```

```
    return Hash, Hash_id, character
```

④利用 LSH 或(K)NN 算法检索图片，进行图片的查找，并保存结果。

(K)NN 被称为邻近算法，或者说 K 最近邻（KNN，K-NearestNeighbor）分类算法，本质上可以认为是暴力枚举的思想，对于每一个需要查询的信息，通过枚举计算出离其最近的 1 个(K 个)信息点，并且将其视作这个信息的代表。对于本次实验而言，可以将“距离”的计算转化为判断是否相同，因为数据单一，每个维度只有 0、1、2，并且计算等号可以简化非常多的计算过程，增加效率，故可直接利用判断是否相同来实现 KNN 算法：

```
# 不使用 LSH，直接利用 NN 法进行搜索
def nn_search(character):
    result = []
    p = color('target')
    for i in range(0,40):
        # 直接枚举
        if character[i] == p:
            result.append('dataset/'+str(i+1)+'.jpg')
    return result
```

对于 LSH(局部敏感度哈希)，本质上就是利用哈希桶来实现类似于 KNN 算法的暴力枚举的想法，只是这个暴力枚举的过程利用了哈希桶之后，可以极大地增加效率，从而使得 LSH 算法的匹配得到最大程度的优化，对于具体的图，我们获取了其特征向量之后，只需要判断是否在哈希桶内，如果在，在具体的那只小桶内查找即可，效率可谓非常之高：

```
# 利用 lsh 检索图片，进行图片索引的查找
def lsh_search(g, Hash, Hash_id, character):
    result = []
    p = color('target')
    lsh = hash(p,g)
    # 判断该图片的特征向量是否合理
    if lsh not in Hash:
        return result
    ID = Hash.index(lsh)
    for i in Hash_id[ID]:
        if character[i] == p:
            result.append('dataset/'+str(i+1)+'.jpg')
    return result
```

进而，本次实验的内容基本完成，最后只需要实现主程序的简单效率判断以及结果输出即可，这里通过 time 库的 time()函数来记录时间（单位为 s（秒）），并且以实现的时间差作为效率的对比即可。

五、实验结果

运行 LSH 可以在终端得到如下结果：

```
When using lsh_search......
The efficiency of lsh_search is 378.8013458251953ms
There is(are) 2 result(s), and they are(it is):
        dataset/12.jpg
        dataset/38.jpg
When using nn_search......
The efficiency of nn_search is 395.8172798156738ms
There is(are) 2 result(s), and they are(it is):
        dataset/12.jpg
        dataset/38.jpg
```

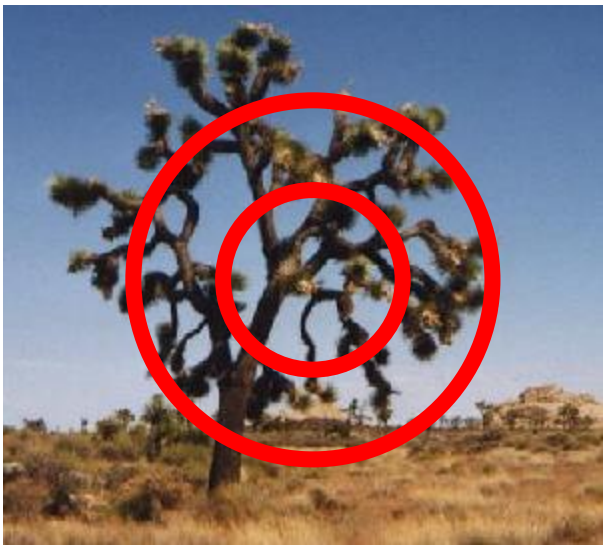
然而，在多次运行程序之后，可以发现，二者的时间花费是时高时低的，为了减少、避免这样的偶然性造成的误差，我通过多次运行这次程序，记录其所花费的时间，取平均，以作为实际（虽然可能仍有误差）的算法效率：

LSH/ms	378.8	359.4	309.2	312.6	295.6	295.8	347.6	340.5	329.9375
KNN/ms	395.8	292.5	314.2	299.1	331.2	322.7	386.4	315.2	332.1375

从这样取平均的方式来看，可以发现 LSH 在小数据规模上是小优于 KNN 算法的，那么当数据规模放大时，LSH 的优势则可以体现的一览无余。

当然，这里 KNN 的效率非常高，还有一个很大的原因，就是我们的 KNN 实现并不是利用传统的距离匹配的方式，而是直接利用每个维度相等的判别方式，这样减少了平方、根号的计算，效率提高了非常之多（如果是其他的应用场景，可能 KNN 并不能达到这么高效）。

需要注意到的是，在实验输出结果方面，我们得到的是 12.jpg 和 38.jpg，然而，理想状态应该只有 38.jpg（虽然 12.jpg 与 38.jpg 非常相似），这说明我们的算法仍旧存在一定的精度问题（左图 12，右图 38）。为了进一步增加本次实验的精度、效果以及提升对具体图片的匹配度，我在**六、实验体会及拓展思考**进行了进一步的探讨！



六、实验体会及拓展思考

本次接触的 LSH 难度适中，并且能够很好地结合之前的如颜色直方图等知识，能够较好地独立自主完成，并且有切实的获得感和体验感，收获颇多。

对于实验的结果精度进一步加强方面，为了将图 12 与图 38 区分开来，我通过改变 p 的分布方式（即 0、1、2 的区间不同）来加强本次的代码实现，经过不断的尝试，最后获得的结果是：

```
for i in range(len(p)):
    if p[i] < 0.38:
        p[i] = 0
    elif p[i] < 0.68:
        p[i] = 1
    else:
        p[i] = 2
```

程序运行结果为：

```
When using lsh_search......
The efficiency of lsh_search is 312.3178482055664ms
There is(are) 1 result(s), and they are(it is):
        dataset/38.jpg
When using nn_search......
The efficiency of nn_search is 290.26007652282715ms
There is(are) 1 result(s), and they are(it is):
        dataset/38.jpg
```

虽然不可否认的是，通过修改 0、1、2 的区间的方法并不是一个特别好的方法，但是在颜色直方图的条件下目前的方法可能是最优的。

【拓展思考】

一、本练习中使用了颜色直方图特征信息，检索效果符合你的预期吗？检索出的图像与输入图像的相似性体现在哪里？

答：总体来说符合预期，但是存在轻微的误差，具体在于两张图（12.jpg,38.jpg）可能提取的基于颜色直方图的特征向量是一样的，即色调上是及其相似的，由于范围的划分（0.3、0.6）导致了出现重合情况，特征向量相同。

二、能否设计其他的特征？

答：理论上是可以的，例如可以采取灰度直方图、梯度直方图作为特征向量，如果有必要，甚至可以采取 SIFT 作为特征向量，只是实现起来较为麻烦，而且不是很好处理，故略。

对于灰度直方图，由于灰度直方图计算时是利用 256 维的灰度值，然而 p 向量最好是 12 维的，因此我将灰度直方图计算时调整为 264=12*22 的大小方便生成特征向量，并且在生成

均匀分布的 0、1、2 时需要着重注意的是，此时我们并不是将一张图分块去计算的，因而我们计算出的 p 普遍均小于 1，那么为了尽可能均匀分布，就应该以每一维的概率 1/12=0.083333 的情况去作为分界线，从而我们可以得到相应的 color 函数，核心的灰度直方图计算是与之前的实验完全相同的，故不作讲述：

```
# 首先获取图像对应的特征向量
# 通过颜色直方图计算相应的特征向量
def color(filename):
    # 读入图（目标的特征向量）
    in_img = cv2.imread(filename+'.jpg')
    image = cv2.cvtColor(in_img, cv2.COLOR_BGR2GRAY)
    # 创建 0~255 的灰度值像素点数
    # 为了与 12 配套，故此处设为 264 = 12 * 22
    gray = [0]*264
    tot  = 0
    for i in range(0,len(image)):
        for j in range(0,len(image[0])):
            # 直接将 image[i][j]点的灰度像素值加到我们的 gray 上
            gray[image[i][j]] += 1
            tot += 1
    # 建立具体比例，存入 p 中
    p = []
    for i in range(12):
        cnt = 0
        for j in range(22):
            cnt += gray[i*22+j]
        p.append(cnt*1.0/tot)
    # 计算出 0、1、2 类型的特征向量
    for i in range(len(p)):
        if p[i] < 0.08:
            p[i] = 0
        elif p[i] < 0.16:
            p[i] = 1
        else:
            p[i] = 2
    return p
```

运行结果如下：

```
When using lsh_search......
The efficiency of lsh_search is 48.505544662475586ms
There is(are) 1 result(s), and they are(it is):
        dataset/38.jpg
When using nn_search......
The efficiency of nn_search is 48.165082931518555ms
There is(are) 1 result(s), and they are(it is):
        dataset/38.jpg
```

可以发现，采取不同的特征提取方法之后，精度反而变高了，而且速度变快了。

对于梯度直方图，其实也是几乎相同的道理，而且梯度值有区间范围，大致在 360 左右，正好为 12*30，因此，在计算方面实际上就是对灰度图的 color 进行更改便可以得到结果了。

```
# 首先获取图像对应的特征向量
# 通过颜色直方图计算相应的特征向量
def color(filename):
    # 读入图（目标的特征向量）
    in_img = cv2.imread(filename+'.jpg')
    image = cv2.cvtColor(in_img, cv2.COLOR_BGR2GRAY)
    Ix, Iy = np.zeros((len(image),len(image[0]))), np.zeros((len(image),len(image[0])))
    M, B = np.zeros((len(image),len(image[0]))), np.zeros((len(image),len(image[0])))
    # 创建 0~255 的灰度值像素点数
    # 为了与 12 配套，故此处设为 264 = 12 * 22
    N = [0]*361
    tot = 0
    for i in range(1,len(image)-1):
        for j in range(1,len(image[0])-1):
            # Ix 是 x 方向的梯度，Iy 即 y 方向的
            Ix[i][j] = int(image[i+1][j])-int(image[i-1][j])
            Iy[i][j] = int(image[i][j+1])-int(image[i][j-1])
            # M 为梯度强度
            M[i][j]=(Ix[i][j]**2+Iy[i][j]**2)**0.5
            # N 记录实际的 i 区间内的值
            N[math.floor(M[i][j])] += 1
            tot += 1
    # 建立具体比例，存入 p 中
    p = []
    for i in range(12):
        cnt = 0
        for j in range(30):
            cnt += N[i*30+j]
        p.append(cnt*1.0/tot)
    # 计算出 0、1、2 类型的特征向量
    for i in range(len(p)):
        if p[i] < 0.08:
            p[i] = 0
        elif p[i] < 0.16:
            p[i] = 1
        else:
            p[i] = 2
    return p
```

运行结果为：

```
When using lsh_search......
The efficiency of lsh_search is 448.7764835357666ms
There is(are) 18 result(s), and they are(it is):
        dataset/1.jpg
        dataset/4.jpg
        dataset/5.jpg
        dataset/6.jpg
        dataset/7.jpg
        dataset/8.jpg
        dataset/10.jpg
        dataset/14.jpg
        dataset/17.jpg
        dataset/18.jpg
        dataset/20.jpg
        dataset/23.jpg
        dataset/25.jpg
        dataset/26.jpg
        dataset/33.jpg
        dataset/36.jpg
        dataset/38.jpg
        dataset/40.jpg
When using nn_search......
The efficiency of nn_search is 439.1303062438965ms
There is(are) 18 result(s), and they are(it is):
        dataset/1.jpg
        dataset/4.jpg
        dataset/5.jpg
        dataset/6.jpg
        dataset/7.jpg
        dataset/8.jpg
        dataset/10.jpg
        dataset/14.jpg
        dataset/17.jpg
        dataset/18.jpg
        dataset/20.jpg
        dataset/23.jpg
        dataset/25.jpg
        dataset/26.jpg
        dataset/33.jpg
        dataset/36.jpg
        dataset/38.jpg
        dataset/40.jpg
```

其实效果并不会太好，出现了非常多相似的图片，说明梯度并不适合作为一个 LSH 匹配的典型提取特征。

综上所述，图片的特征选取对其产生的特征向量至关重要，好的提取法甚至可以提高运算效率和准确度（例如灰度），不好的匹配方式不仅会降低程序效率，而且正确率未必能得到好的保证。

上述两个颜色提取方式存放在 LSH_with_dif_color.py 和 LSH_with_dif_color2.py 之中，没有在**七、实验代码**中贴出，前者为灰度直方图的提取方式，后者为梯度直方图的提取方式。

七、实验代码

LSH 算法的具体实现 LSH.py：

```
import cv2
from matplotlib import pyplot as plt
import time
# 首先获取图像对应的特征向量
# 通过颜色直方图计算相应的特征向量
def color(filename):
    # 读入图（目标的特征向量）
    in_img = cv2.imread(filename+'.jpg')
    image = cv2.cvtColor(in_img, cv2.COLOR_BGR2RGB)
    # 需要注意是 4 块区域，分为 BGR
    blue, green, red = [0,0,0,0], [0,0,0,0], [0,0,0,0]
    # 宽 len(image),长 len(image[0])，读入的图片按照 r,g,b 的顺序
    # 分成四块来进行计算
    for i in range(0,len(image)//2):
        for j in range(0,len(image[0])//2):
            blue[0] += image[i][j][2]
            green[0]+= image[i][j][1]
            red[0]  += image[i][j][0]
    # 归一化处理
    tot = blue[0] + green[0] + red[0]
    blue[0], green[0], red[0] = blue[0]*1.0/tot, green[0]*1.0/tot, red[0]*1.0/tot
    for i in range(0,len(image)//2):
        for j in range(len(image[0])//2,len(image[0])):
            blue[1] += image[i][j][2]
            green[1]+= image[i][j][1]
            red[1]  += image[i][j][0]
    # 归一化处理
    tot = blue[1] + green[1] + red[1]
    blue[1], green[1], red[1] = blue[1]*1.0/tot, green[1]*1.0/tot, red[1]*1.0/tot
    for i in range(len(image)//2,len(image)):
        for j in range(0,len(image[0])//2):
            blue[2] += image[i][j][2]
            green[2]+= image[i][j][1]
            red[2]  += image[i][j][0]
```

```
        # 归一化处理
        tot = blue[2] + green[2] + red[2]
        blue[2], green[2], red[2] = blue[2]*1.0/tot, green[2]*1.0/tot, red[2]*1.0/tot
        for i in range(len(image)//2,len(image)):
            for j in range(len(image[0])//2,len(image[0])):
                blue[3] += image[i][j][2]
                green[3]+= image[i][j][1]
                red[3]  += image[i][j][0]
        # 归一化处理
        tot = blue[3] + green[3] + red[3]
        blue[3], green[3], red[3] = blue[3]*1.0/tot, green[3]*1.0/tot, red[3]*1.0/tot
        p = []
        for i in range(4):
            p.append(blue[i])
            p.append(green[i])
            p.append(red[i])
        # 计算出0、1、2类型的特征向量
        for i in range(len(p)):
            if p[i] < 0.3:
                p[i] = 0
            elif p[i] < 0.6:
                p[i] = 1
            else:
                p[i] = 2
        return p

    # 计算Hamming码来获取投影，其中g是需要被投影的方向
    def Hamming(p,g):
        # 由于p是12维的向量，构造v(p)需要24维的数组
        lsh = []
        v = [0] * 24
        for i in range(12):
            # 当前位置只有00,10,11三种情况，第一二位分别对应2*i,2*i+1
            if p[i] == 1:
                v[2*i] = 1
            if p[i] == 2:
                v[2*i] = 1
                v[2*i+1] = 1
        # eg.v(p)=001011100011
        # 对于上述 p，它在{1,3,7,8}上的投影为(0,1,1,0)
        for i in g:
            lsh.append(v[i-1])
        return lsh


    # 哈希函数计算lsh
```

```
81.     def hash(p,g):
82.         # I 是 Hash 桶
83.         I  = [[] for i in range(12)]
84.         # print(I)
85.         lsh = []
86.         # I|i 表示 I 中范围在(i-1)*C+1~i*C 中的坐标：
87.         for i in g:
88.             # print((i-1)//2)
89.             I[(i-1)//2].append(i)
90.         for i in range(12):
91.             if I[i] != []:
92.                 for Ii in I[i]:
93.                     if Ii-2*i <= p[i]:
94.                         lsh.append(1)
95.                     else:
96.                         lsh.append(0)
97.         return lsh
98.
99.
100.    # 对搜索库中的图像（共 40 张，均以 i.jpg 为文件名）建立索引
101.    def pre(g):
102.        Hash, Hash_id, character = [], [], []
103.        for i in range(1,41):
104.            filename ='dataset/'+str(i)
105.            p = color(filename)
106.            character.append(p)
107.            # 可用 I 的方式获取投影
108.            # 也可以用汉明法获取投影即 lsh = Hamming(p,g)
109.            lsh = hash(p,g)
110.            # 需要注意的是，由于循环的时候文件用的是 1.jpg->40.jpg
111.            # 然而数组 append 只能是从 0 开始，因而在具体实现的时候需要-1，同样回答了输出的+1 原因
112.            try:
113.                Hash_id[Hash.index(lsh)].append(i - 1)
114.            except:
115.                Hash.append(lsh)
116.                Hash_id.append([i - 1])
117.        return Hash, Hash_id, character
118.
119.    # 利用 lsh 检索图片，进行图片索引的查找
120.    def lsh_search(g, Hash, Hash_id, character):
121.        result = []
122.        p = color('target')
123.        lsh = hash(p,g)
124.        # 判断该图片的特征向量是否合理
125.        if lsh not in Hash:
126.            return result
```

```
        ID = Hash.index(lsh)
        # 这里的 NN 法可以直接判断向量是否相等，这样直接且高效
        # print(Hash_id[ID])
        for i in Hash_id[ID]:
            if character[i] == p:
                result.append('dataset/'+str(i+1)+'.jpg')
        return result

# 不使用 LSH，直接利用 NN 法进行搜索
def nn_search(character):
    result = []
    p = color('target')
    for i in range(0,40):
        # 直接枚举
        if character[i] == p:
            result.append('dataset/'+str(i+1)+'.jpg')
    return result

g = [1,7,10,15,20]

# 首先进行预处理，为所有待查图像建立相应索引
Hash, Hash_id, character = pre(g)

start = time.time()
print("When using lsh_search......")
result = lsh_search(g, Hash, Hash_id, character)
end = time.time()
print("The efficiency of lsh_search is {}ms".format((end-start)*1000))
if result == []:
    print("No matching picture!")
else:
    print("There is(are) {} result(s), and they are(it is):".format(len(result)))
    for i in result:
        print('\t' + i)

start = time.time()
print("When using nn_search......")
result = nn_search(character)
end = time.time()
print("The efficiency of nn_search is {}ms".format((end-start)*1000))
if result == []:
    print("No matching picture!")
else:
    print("There is(are) {} result(s), and they are(it is):".format(len(result)))
    for i in result:
        print('\t' + i)
```

八、参考文献

①《LSH（局部敏感度哈希）》——Bob_tensor

https://blog.csdn.net/weixin_43461341/article/details/105603825

②《LSH(Locality Sensitive Hashing)原理与实现》——guoziqing506

https://blog.csdn.net/guoziqing506/article/details/53019049

③《Similarity Search in High Dimension via Hashing》