

# 上机实验十三 实验报告

刘瀚文

517030910294

2018 年 12 月 25 日

# 目录

<b>I 实验十三</b>	<b>3</b>
<b>1 实验准备</b>	<b>4</b>
1.1 环境配置 . . . . .	4
1.1.1 Python . . . . .	4
1.2 背景知识——图像检索 . . . . .	5
1.2.1 Locality-sensitive Hashing . . . . .	5
1.2.2 Hashing 的基本思想 . . . . .	5
1.2.3 数据的表示 . . . . .	6
1.2.4 LSH 预处理 . . . . .	6
1.2.5 哈希函数计算 . . . . .	7
1.2.6 LSH 检索 . . . . .	7
1.2.7 检索算法流程 . . . . .	7
<b>2 实验</b>	<b>8</b>
2.1 实验要求 . . . . .	8
2.2 实验过程 . . . . .	8
2.2.1 Basic Process . . . . .	9
2.2.2 哈希函数 LSHsearch() . . . . .	10
2.2.3 哈希预处理 PreProcess() . . . . .	11
2.2.4 正常预处理 PreProcessMax() . . . . .	12
2.2.5 NN 暴力搜索函数 NN_search(vec, aimSet) . . . . .	12
2.2.6 哈希比较函数 LSHCompare() . . . . .	13
2.3 结果展示 . . . . .	14
2.3.1 使用暴力搜索图片的时间 . . . . .	14

2.3.2	使用 [1, 3, 7, 8] 维向量搜索图片的时间 . . . . .	15
2.3.3	使用 [1, 3, 7, 11] 维向量搜索图片的时间 . . . . .	15
2.3.4	使用 mod6 搜索图片的时间 . . . . .	16
2.3.5	额外实验 . . . . .	16

## II 实验总结 17

## Part I

## 实验十三

# Chapter 1

## 实验准备

### 1.1 环境配置

#### 1.1.1 Python

环境说明：Python 2.7.15

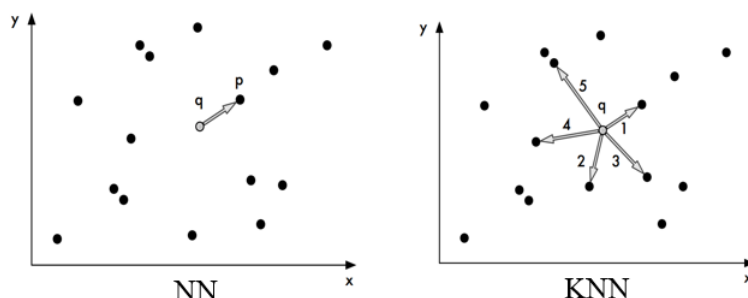
使用 Ubuntu 14.04 VMware 14

## 1.2 背景知识——图像检索

### 1.2.1 Locality-sensitive Hashing

Why use LSH

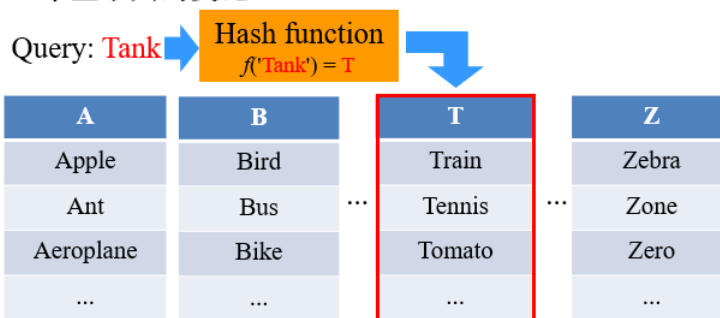
用Nearest neighbor (NN) 或k-nearest neighbor (KNN)在数据库中检索和输入数据距离最近的1个或k个数据，一般情况下算法复杂度为 $O(n)$ （例如暴力搜索），优化情况下可达到 $O(\log n)$ （例如二叉树搜索），其中 $n$ 为数据库中的数据量。当数据库很大（即 $N$ 很大时），搜索速度很慢。



### 1.2.2 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：



### 1.2.3 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

$$\mathbf{p} = (p_1, p_2, \dots, p_d)$$

- 其中 $p_i$ 是整数, 满足  $0 \leq p_i \leq C$ , 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 $\mathbf{p}$ 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  $\mathbf{p} = (p_1, \dots, p_{12})$
- 每个分量满足  $0 \leq p_j \leq 1$  将其量化成
- 3个区间分别用0 1 2表示:

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

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  $p_j \in \{0,1,2\}$

### 1.2.4 LSH 预处理

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

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

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

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

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

$$\mathbf{v}(\mathbf{p}) = 001011100011$$

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

$$I_i = \{i_1, i_2, \dots, i_m\}: 1 \leq i_1 < i_2 < \dots < i_m \leq d'$$

上的投影为  $g_i(\mathbf{p}) = p_{i_1} p_{i_2} \cdots p_{i_m}$ , 其中 $p_{i_j}$ 为 $\mathbf{v}(\mathbf{p})$ 的第 $i_j$ 个元素。对于上述 $\mathbf{p}$ , 它在 $\{1,3,7,8\}$ 上的投影为 $(0,1,1,0)$

ing

4

### 1.2.5 哈希函数计算

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

• $I|i$ 表示 $I$ 中范围在 $(i-1)*C+1 \sim i*C$ 中的坐标:

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

• $v(p)$ 在 $I$ 上的投影即是 $v(p)$ 在 $I|i(i=1,2,\dots,d)$ 上的投影串联,  $v(p)$ 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:

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

•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;

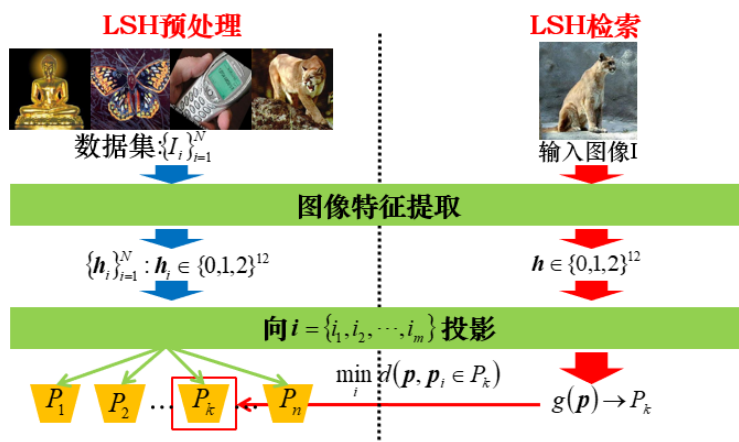
•串联得到: (0,1,1,0)

### 1.2.6 LSH 检索

$g(p)$ 被称作Hash函数, 对于容量为 $N$ 的数据集 $\{p_i\}_{i=1}^N$ ,  $g(p)$ 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 $p$ , 先计算 $g(p)$ , 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。

### 1.2.7 检索算法流程





## Chapter 2

# 实验

### 2.1 实验要求

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。

### 2.2 实验过程

按实验的思路，结合代码的顺序进行分析。

### 2.2.1 Basic Process

基本处理环节：

通过将图像分成 4 各块，然后分别获取每小块的 R G B 的数据。

使用归一化条件，获得图像的十二维向量。

接着对图像进行处理，获得图像的 Hamming 码。

```
# Basic Process
def basicProcess(img):
    H = len(img)
    W = len(img[1])

    MH = int(H / 2)
    MW = int(W / 2)

    l1 = getPartInfo(img, 0, MH, 0, MW)
    l2 = getPartInfo(img, 0, MH, MW, W)
    l3 = getPartInfo(img, MH, H, 0, MW)
    l4 = getPartInfo(img, MH, H, MW, W)

    l = []
    for i in l1:
        l.append(reset(i))
    for i in l2:
        l.append(reset(i))
    for i in l3:
        l.append(reset(i))
    for i in l4:
        l.append(reset(i))

    # LSH PreProcess
    res = []
    for i in l:
        if i == 0:
            res.extend([0, 0])
        elif i == 1:
            res.extend([1, 0])
        elif i == 2:
            res.extend([1, 1])
    return res
```

其中用到了 `getPartInfo()` 函数，用这个函数来获取归一化的部分小块的图像的图片信息。

```
def getPartInfo(img, start1, end1, start2, end2):
    r = 0
    g = 0
    b = 0
    for i in range(start1, end1):
        for j in range(start2, end2):
            r += img[i][j][0]
            g += img[i][j][1]
            b += img[i][j][2]

    count = r + g + b
    r = round(r / count, 2)
    g = round(g / count, 2)
    b = round(b / count, 2)
    l = []
    l.append(r)
    l.append(g)
    l.append(b)
    return l
```

### 2.2.2 哈希函数 LSHsearch()

我使用了两种方法的哈希函数。

#### 第一种使用投影的方法

使用 12 维向量的参数投影，并且计算对应的值。

使用  $\text{count} = \text{count} * 2 + i$  的方法，尽可能地可以减少冲突。

```
# LSH Search
def LSHsearchN(vec):
    l = []
    count = 0
    list = [1, 7, 12]
    for i in list:
        l.append(vec[i])
    for i in l:
        count = count * 2 + i
    return count
```

## 第二种使用同余取模的方法

由于 12 维的向量，加和的最大值为  $2 * 12 = 24$ ，我用总和模 6 来，来将数据分到不同的组里面去。

```
# LSH Search
def LSHsearch(vec):
    count = 0
    for i in vec:
        count += vec[i]
    count %= 6
    return count
```

### 2.2.3 哈希预处理 PreProcess()

中规中矩的方式，读取每一张图片，然后根据图片信息放入不同的列表中。

比较巧妙的是：使用了 pk 的文件写入方式，能够更有效的写入数据。

```
# PreProcess
def PreProcess():
    print("Processing...")
    dataset = []
    for i in range(6):
        dataset.append([])
    for i in range(1, 41):
        imgname = "Dataset/{i}.jpg".format(i)
        print("Processing Pic {i} ...".format(i))
        img = cv2.imread(imgname)
        vec = basicProcess(img)
        hash = LSHsearch(vec)
        dataset[hash].append(tuple([imgname, vec]))
    file = open("Data.pkl", "wb")
    pk.dump(dataset, file)
    file.close()
    print("Data PreProcess Done !")
```

## 2.2.4 正常预处理 PreProcessMax()

正常图片的预处理就是直接记录图片的信息 + 图片的名字。

```
def PreProcessMax():
    print("Processing...")
    dataset = []
    for i in range(1, 41):
        imgname = "Dataset/{0}.jpg".format(i)
        print("Processing Pic {0} ...".format(i))
        img = cv2.imread(imgname)
        vec = basicProcess(img)

        dataset.append(tuple([imgname, vec]))
    file = open("DataMax.pkl", "wb")
    pk.dump(dataset, file)
    file.close()
    print("DataMax PreProcess Done !")

PreProcessMax()
```

## 2.2.5 NN 暴力搜索函数 NN\_search(vec, aimSet)

对于给定的 12 维向量和目标搜索集合。对于内在元素进行比对搜索。

```
def NN_search(vec, aimSet):
    result = []
    len1 = len(aimSet)
    len2 = len(vec)
    for i in range(len1):
        for j in range(len2):
            if vec[j] != aimSet[i][1][j]:
                break

        if j == len2 - 1:
            result.append(aimSet[i][0])

    num = len(result)
    for i in range(num):
        img = cv2.imread(result[i])
        cv2.imshow("Matched Target", img)

    if (num == 1):
        print("The Matched Photo is {}".format(result[0]))
    else:
```

```

    for i in range(num):
        print("The Sim Photos are {}".format(result[i]))

# LSHCompare()

```

### 2.2.6 哈希比较函数 LSHCompare()

在有了 NN 暴力搜索函数 NN\_search(vec, aimSet) 之后，其实这部分就很简单了。

就是对应于之前方法使用的哈希函数，获取对应的目标探测集合。

对于 target.jpg 进行处理，获取对应的信息。

然后投入到暴力搜索函数 NN\_search(vec, aimSet) 中就可以得出结果啦。

同时这部分也展示原始图片和搜索出来的结果进行可视化的展示。

这部分中也使用了 time.clock() 函数进行计时，以比对时间。

对于正常照片处理也有一个相似的函数，由于没有什么额外的东西，就不占用空间进行展示啦。

```

def LSHCompare():
    a = time.clock()
    img = cv2.imread("target.jpg")
    cv2.imshow("Raw Target", img)
    vec = basicProcess(img)
    hash = LSHserarch(vec)
    with open("Data.pkl", "rb") as f:
        dataset = pk.load(f)

    aimSet = dataset[hash]
    print("\nVector of Target : ")
    print(vec)
    # print(aimSet[0][0]) # name
    # print(aimSet[0][1]) # vector

    NN_search(vec, aimSet)
    b = time.clock()
    print("Using {} seconds.".format(b - a))
    cv2.waitKey(0)

    cv2.destroyAllWindows()

```

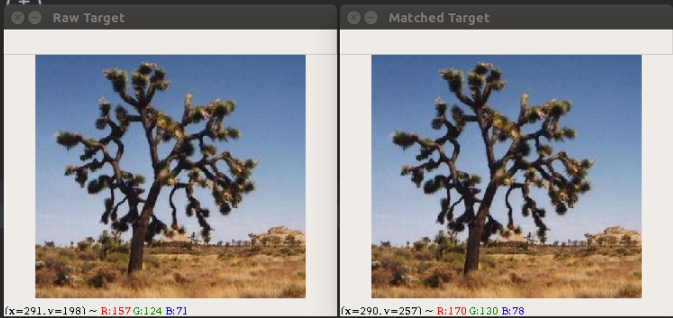
## 2.3 结果展示

结果中呈现的是搜索时间。

总时间由于使用的是写入文件的方式，所以就先不对那部分的时间进行比较。

### 2.3.1 使用暴力搜索图片的时间

```
148 def Compare():
149     a = time.clock()
150     img = cv2.imread("target.jpg")
151     cv2.imshow("Raw Target", img)
152     vec = basicProcess(img)
153
154     with open("DataMax.pkl", "rb") as f:
155         dataset = pk.load(f)
156
157     aimSet = dataset
158     print("\nVector of Ta
159     print(vec)
160     # print(aimSet[0][0])
161     # print(aimSet[0][1])
```

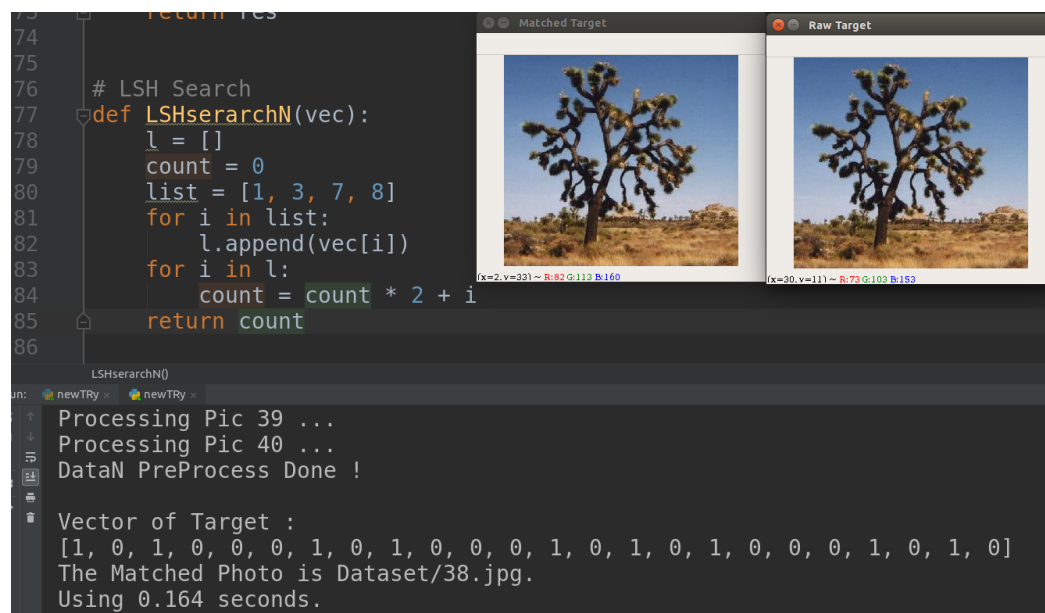


```
Processing Pic 39 ...
Processing Pic 40 ...
DataMax PreProcess Done !

Vector of Target :
[1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0]
The Matched Photo is Dataset/38.jpg.
Using 0.358404 seconds.
```

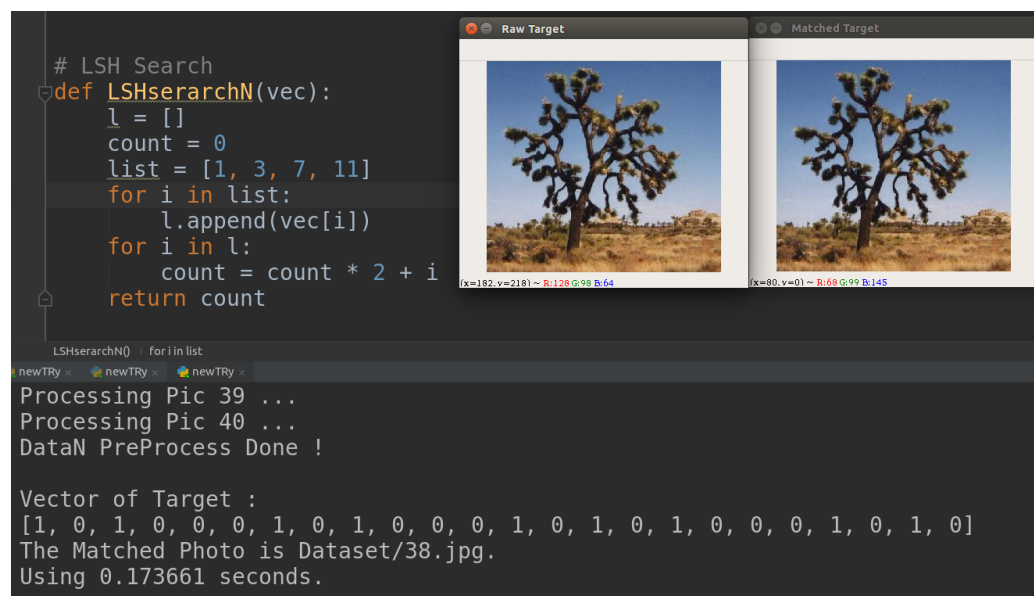
用时 0.3584s

### 2.3.2 使用 [1, 3, 7, 8] 维向量搜索图片的时间



用时 0.164s

### 2.3.3 使用 [1, 3, 7, 11] 维向量搜索图片的时间



用时 0.1736s

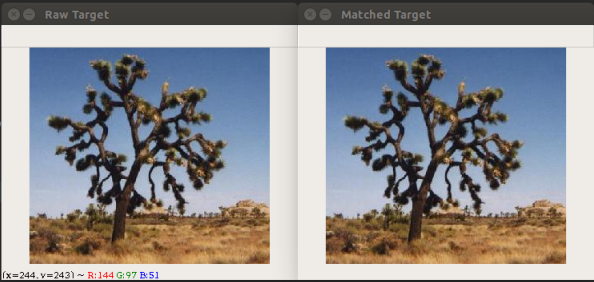


### 2.3.4 使用 mod6 搜索图片的时间

```
def NN_search(vec, aimSet):
    result = []
    len1 = len(aimSet)
    len2 = len(vec)
    for i in range(len1):
        for j in range(len2):
            if vec[j] != aimSet[i]:
                break
        if j == len2 - 1:
            result.append(aimSet[i])

NN_search()
if (num == 1):
    try(1)
Processing Pic 39 ...
Processing Pic 40 ...
Data PreProcess Done !

Vector of Target :
[1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0]
The Matched Photo is Dataset/38.jpg.
Using 0.193545 seconds.
```



用时 0.1935s

### 2.3.5 额外实验

同时这次也对于大作业中爬取的 3k 张图片进行了搜索。

之前的大作业图片以图识图的搜索引擎是基于上次实验的代码，但是学习了哈希搜索之后，发现对于 3K 级别的数据搜索有了明显的提升。

搜索的结果由原本的几秒的数量级（肉眼明显可以感觉到的延迟）降低到只需要零点几秒，所以也是有了巨大的提升。

## Part II

# 实验总结

这次实验深入学习了图像匹配的算法，并且用自己的代码水平进行了方法的实现，很有趣！

在这次实验中，最大的收获，就是可以对自己大作业中的代码进行优化，有了进几十倍的效率提高，真的是一件很好的事，也学习到了好多知识。

这是最后一次实验啦，感谢老师和助教老师一直的陪伴，我也会好好利用这学期学到的知识，不断进步！

谢谢！