电工导实验报告 9

一、实验目的

1. 基于 LSH 索引的快速图像检索

二、实验内容

1. 了解为什么使用 LSH
2. 通过 Hash 函数提高效率
3. LSH 检索

三、实验环境

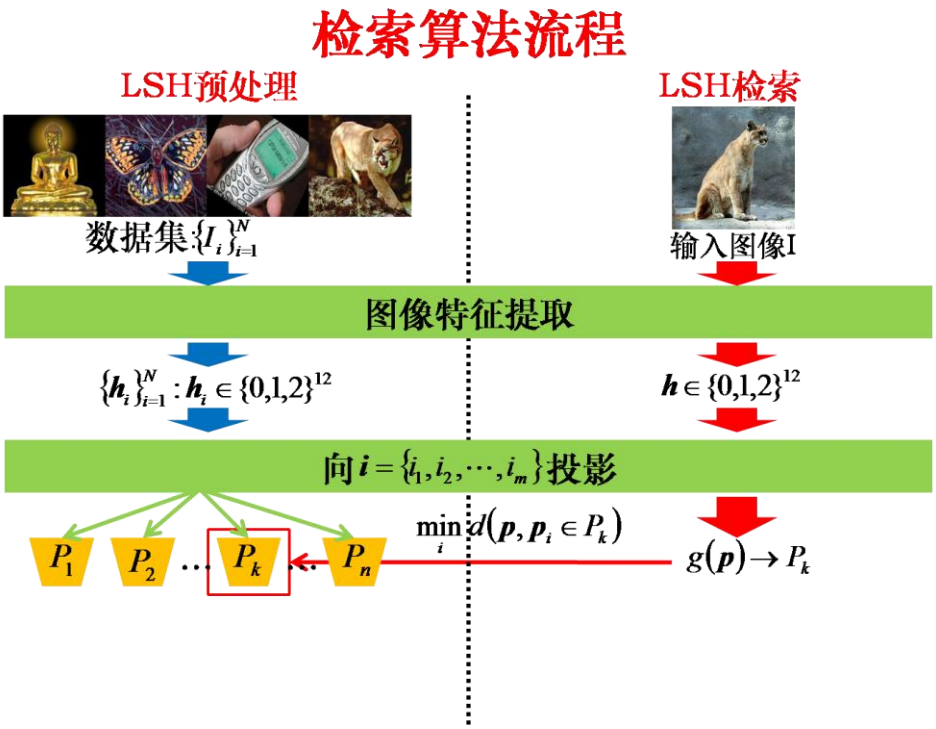
1. Python 2.7 + opencv +numpy

四、实验原理

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。

一幅图像可以分成四个部分，计算每个部分的 RGB 的比例，映射到 0，1，2 上，因此一幅图会得到一个由元素 {0，1，2} 组成的一个特征向量。通过哈希函数，将 12 维的特征向量投影到一个集合 I 上，相当于每一幅图都会得到一个投影的字符串，而一个字符串可能对应多幅颜色分布相近的图片。在很多图片中寻找最相似的图片后，通过哈希函数将需要比对的范围缩小到更小的范围，以提高匹配的效率。



五、算法实现及流程简介

对于一幅图，首先要分四个区域计算其颜色直方图，颜色直方图的计算之前已经非常熟练。然后将 RGB 三原色各自的比例投影到 0，1，2，以此来量化特征向量。将得到的四组三维向量组合到一起，得到一个由元素｛0，1，2｝构成的 12 维特征向量。

```
def cal(img,h1,h2,w1,w2): #按h1~h2,w1~w2区域计算p
    b, g, r = cv2.split(img)
    blue=0
    green=0
    red=0
    for i in range(h1,h2):
        for j in range(w1,w2):
            blue += b[i,j]
            green += g[i,j]
            red += r[i,j]
    total = blue+green+red
    p = [float(blue)/total, float(green)/total, float(red)/total] #颜色直方图

    for i in range(len(p)):
        if (p[i]<0.3):
            p[i] = 0
        elif (p[i]>=0.3 and p[i]<0.6):
            p[i] = 1
        else:
            p[i] = 2
    return p

def get_vector_p(img): #得到12维特征向量
    h = img.shape[0]
    w = img.shape[1]
    p = cal(img,0,h/2,0,w/2)
    p = p + cal(img,h/2+1,h,0,w/2)
    p = p + cal(img,0,h/2,w/2+1,w)
    p = p + cal(img,h/2+1,h,w/2+1,w)
    return p
```

之后是构造哈希函数，通过将 12 维的向量投影到 I 上，得到一个 len(I)的字符串。

```
def get_hash(p):
    proj_set = [1,3,5,7,9,11] #自定义
    vp = ""
    for i in p:
        if i==0:
            vp += "00"
        elif i==1:
            vp += "10"
        else:
            vp += "11"
    #print vp
    gp=""
    for i in proj_set:
        gp += vp[i-1]
    #print gp
    return gp
```

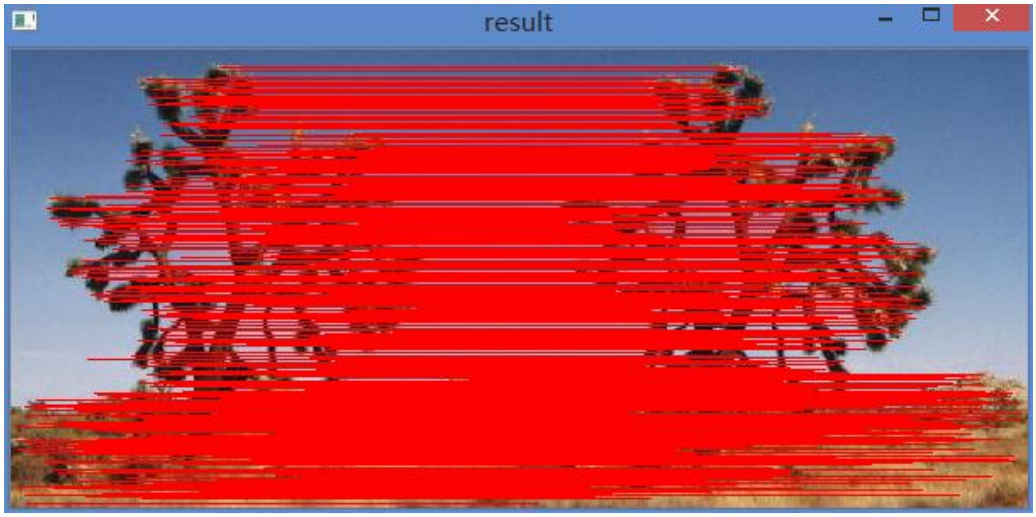
在 main 函数中，建立一个 dictionary,将所有得到的映射后的字符串作为 key,以此来对图片进行分类。当要寻找一幅图的时候，只要计算其经过 hash 后的字符串，到 dictionary 中寻找对应的一个小分类即可。如果出现找不到的情况，那就与所有的 key 进行比较，寻找相似度最高的 key，并与该 key 对应的图片进行比较，比较过程就通过上一次的 sift。由于

hash 函数的分类作用，能大大缩短匹配的时间，相当于把一些不太可能的图片直接滤去了，大幅度提升了效率和程序运行的速度。

六、运行效果

运行效果如下：

```
narrow the search scope to:
pic 12
pic 38
best matching result:
pic 38
time used:
0.408219253068
```



可见，通过 hash 分类以后，匹配的时候只需要和两幅相近的图片进行进一步的 sift 匹配，所需的时间也非常少。(注：时间计算的时候没有将建立 hash 的时间计算在内，计算的只是匹配的时间)。

上述结果我设定 proj_set = [1,3,5,7,9,11] 时的，集合的元素越多，相当于分类的类别越细，搜索的时候时间也就越短（可能建立 hash 时会多花一点点时间）；反之，如果集合元素的个数很少，如果只有一个元素，那只能分成两类，虽然比不分类快，但相比别的集合来说速度就慢了很多。

例如三个元素的时候，分类不是很细，需要匹配的图片也就相应增多。

```
narrow the search scope to:
pic 11
pic 12
pic 13
pic 17
pic 23
pic 26
pic 37
pic 38
pic 40
best matching result:
pic 38
time used:
0.85920775388
```

如果不通过 hash 来减少匹配的次数，每一幅图都匹配一次，那是时间会大大增加。对比运行结果如下：

```
best matching result:
pic 38
time used:
4.12456406022
```

结果是相同的，但是时间相差了 7 倍，可见 hash 函数的强大作用。

七、实验总结

LSH，局部敏感哈希算法通过给图片分类，使得在很多图像中匹配一幅图片的时间大幅缩短，大大节省了时间，增加了效率。dataset 中只有 40 幅图效率就相差很多，在成百上千幅图中匹配的话运用 hash 的优势将会更加被放大。之前网络爬虫的时候也用过 hash 函数，进而判断该 url 是否已经爬取过。这一次接触 hash,不仅更加熟悉和了解，也感叹算法对程序的优化效果如此惊人。

F1403023　5140309534　韩坤言

附源码：

```
# -*- coding: utf-8 -*-
import cv2
import numpy as np
import time

def cal(img,h1,h2,w1,w2): #按 h1~h2,w1~w2 区域计算 p
    b, g, r = cv2.split(img)
    blue=0
    green=0
    red=0
    for i in range(h1,h2):
        for j in range(w1,w2):
            blue += b[i,j]
            green += g[i,j]
            red += r[i,j]
    total = blue+green+red
    p = [float(blue)/total, float(green)/total, float(red)/total] #颜色直方图

    for i in range(len(p)):
        if (p[i]<0.3):
            p[i] = 0
        elif (p[i]>=0.3 and p[i]<0.6):
            p[i] = 1
        else:
            p[i] = 2
    return p

def get_vector_p(img): #得到 12 维特征向量
    h = img.shape[0]
    w = img.shape[1]
    p = cal(img,0,h/2,0,w/2)
    p = p + cal(img,h/2+1,h,0,w/2)
    p = p + cal(img,0,h/2,w/2+1,w)
    p = p + cal(img,h/2+1,h,w/2+1,w)
    return p

def get_hash(p):
    proj_set = [1,3,5,7,9,11] #自定义
    vp = ""
    for i in p:
        if i==0:
            vp += "00"
        elif i==1:
```

```
            vp += "10"
        else:
            vp += "11"
    #print vp
    gp=""
    for i in proj_set:
        gp += vp[i-1]
    #print gp
    return gp



def sift_match(target,imgs): #sift 以及匹配
    sift = cv2.SIFT()
    kp_targ, des_targ = sift.detectAndCompute(target,None)
    FLANN_INDEX_KDTREE = 0
    index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
    search_params = dict(checks=50)
    flann = cv2.FlannBasedMatcher(index_params,search_params)

    max_num = 0
    for img in imgs:
        kp, des = sift.detectAndCompute(img[1],None)
        matched = flann.knnMatch(des,des_targ,k=2) #和 target 比较
        #print matched
        accept = []
        for m,n in matched:
            if m.distance < 0.08*n.distance: #限定来找相似点 0.08
                accept.append(m)
        print len(accept)
        if len(accept)>max_num: #img 匹配的数量更多 刷新
            max_num = len(accept)
            best_img = img
            best_kp = kp
            best = accept
    print "best matching result:"
    print "pic", best_img[0]

    #将两幅图放在一幅图中
    h1, w1 = target.shape[:2]
    h2, w2 = best_img[1].shape[:2]
    result = np.zeros((max(h1, h2), w1 + w2, 3), np.uint8)
    result[:h1, :w1] = target[:,:]
    result[:h2, w1:] = best_img[1][:,:]
```

```
    #将匹配到的关键点连起来
    for m in best:
        color = (0,0,255)
        cv2.line(result, \
        (int(best_kp[m.queryIdx].pt[0]+w1),int(best_kp[m.queryIdx].pt[1])) , \
        (int(kp_targ[m.trainIdx].pt[0]), int(kp_targ[m.trainIdx].pt[1])), color)
    return result

def main():
    imgs = []
    for i in range(1,41):
        img = cv2.imread(str(i)+".jpg")
        imgs.append([i,img]) #把图片的名字一并记下

    hash_dic = {} #hash 函数值作为 key 用来分类
    for i in range(len(imgs)):
        p = get_vector_p(imgs[i][1])
        gp = get_hash(p)
        if hash_dic.has_key(gp):
            hash_dic[gp].append(imgs[i])
        else:
            hash_dic[gp] = [imgs[i]]

    target = cv2.imread("target.jpg")
    p = get_vector_p(target)
    gp = get_hash(p)

    t1 = time.clock()

    #通过哈希缩小需要匹配的范围
    if hash_dic.has_key(gp):
        pos_imgs = hash_dic[gp]
    else: #如果没有找到 寻找 hash 值最接近的
        pos_imgs =[]
        keys = hash_dic.keys()
        match = []
        for item in keys:
            cnt = 0
            for i in range(len(gp)):
                if (gp[i]==item[i]):
                    cnt = cnt + 1;
            match.append(cnt)
        maxi = max(match)
```

```
        for i in range(len(match)):
            if (match[i]==maxi):
                pos_imgs = pos_imgs + hash_dic[keys[i]]

    print "narrow the search scope to:"
    for i in pos_imgs:
        print "pic", i[0]
    result = sift_match(target,pos_imgs)

    t2 = time.clock()
    print "time used:"
    print t2-t1

    cv2.imshow('result',result)
    cv2.waitKey()
    cv2.destroyAllWindows()

if __name__ == '__main__':
    main()
```