林禹臣 5140309507

December 26, 2015 (Week 15)

电类工程导论C 实验报告

——LSH搜索

目录页

1.引言

在这次实验里我主要实现了LSH 的方法检索图片，更加深刻的理解了 LSH 的原理和实际效果，并且与 NN 的方法进行比较，还引入了余弦近似度的计算，从而可以根据相似度进行排序。

2.实验环境

Mac OS X 10.11.2 + opencv 2.4.12 + numpy1.10.1 + Python 2.7.10 (64bit)

3.实验原理

LSH，局部敏感哈希函数。

哈希函数的概念就是通过一个函数将某个属性（可以是一个值，也可以是一个向量）映射为另外一个属性。通常我们也把这个属性直观的理解为一个桶。所以哈希函数的过程也可以看做，将一群元素根据某种规则分配到不同的桶中，这样以后查找他的时候，可以先计算一下它在哪个桶中，然后在这个桶里面找。

普通的哈希函数就是这样了，但是我们希望哈希函数能够更强悍一点。

第一，我们希望这个函数可以把所有的元素均匀的分布在每个桶中。

第二，我们希望两个相似的元素分配在相似的桶中。

如果满足了第二点，那么这个哈希函数就是局部敏感哈希函数。

用数学的语言描述就是这样的：

令d1<d2是定义在距离测定d下得两个距离值，如果一个函数族的每一个函数f满足：

（1）如果d(x,y)<=d1,则f(x)=f(y)的概率至少为p1，即P(f(x)=f(y)) >= p1；

（2）如果d(x,y)>=d2,则f(x)=f(y)的概率至多为p2，即p(f(x)=f(y)) <= p2；

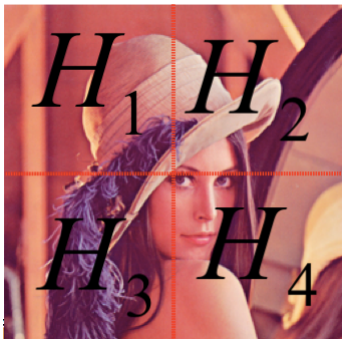
则称F为(d1,d2,p1,p2)-敏感的函数族。

4.实验过程

4.1. 特征向量的计算和量化

我们要对一个图片进行特征提取，这里为了方便，选择了一种非常简单的方式来构造这个特征向量。

如右图所示，我们将一个图片分隔成四个部分，然后对每个部分计算 RGB 三中颜色的能量比例，把这个比例进行离散量化处理成0，1，2三个档，所以一共又3*4 = 12 维。



处理每一块的时候代码如下：

```
#计算img的RGB能量比例
def CalcRGB(img):
    b,g,r = cv2.split(img)  #分散三种颜色，注意顺序
    #分别计算三种颜色的能量
    energy_b = np.sum(b)
    energy_g = np.sum(g)
    energy_r = np.sum(r)
    energy_all  = energy_b + energy_g + energy_r #计算总能量
    #输出三个能量的比例
    res = [0.0,0.0,0.0]
    res[0] = (energy_b+0.0) / energy_all
    res[1] = (energy_g+0.0) / energy_all
    res[2] = (energy_r+0.0) / energy_all
    return res
```

计算特征向量，并且量化。

```
#计算一个图片的特征向量
def CalcP(imgurl):
    P = []
    img = cv2.imread(imgurl,1)  #读入图像
    h = img.shape[0]
    w = img.shape[1]
    P += CalcRGB(img[0:h/2,0:w/2])
    P += CalcRGB(img[h/2:h,0:w/2])
    P += CalcRGB(img[0:h/2,w/2:w])
    P += CalcRGB(img[h/2:h,w/2:w])

    #为了使 0 1 2 分布均匀测试得到的比例
    low = 0.32
    high = 0.35

    for i,p in enumerate(P):
        P[i] = 0
        if(p>low and p<high):
            P[i] = 1
        elif(p>=high):
            P[i] = 2
    return P
```

这里要注意一点就是 low = 0.32 和 high = 0.35 的处理是通过我的实验得到的，PPT 中给的0.3 和 0.6 使得分布非常不均匀。我通过多次实验得到了这两个数值，可以保证 0 有116个，1 有 144个，2 有 112 个，相对分布均匀一点，这样对哈希函数的平稳性有很大的好处。

4.2. 计算LSH

这里可以显式地去计算海明码然后通过海明码向指定的位置进行映射，把映射结果串联当做最后的哈希值。但是在效率上有些低，所以我们采用了隐式地计算。

首先我们把需要映射的位置分块处理，因为原来一共有 d*C 个位置需要考虑，我们分成 d 个桶，把这些需要映射的位置进行分类。分类的标准是，将这些位置的序号按照除以 C，得到商数放在对应的桶中。然后计算哈希值的时候找到非空的桶，按顺序进行计算，把映射结果进行串联即可，代码如下：

```
#p 是一个特征向量
#Sub 是一个需要映射的位置的集合 比如 [1,7,10,15,20]
def CalcLSH(p,Sub):
    #制备12个空集合的集合
    I = [[] for i in range(12)]

    for i in Sub:    #对每个元素进行分类
        I[(i-1)/2].append(i)

    lsh = []

    #I[ind] 即是 PPT 中的 I|ind 这个集合
    for ind in range(12):
        if(len(I[ind])==0):
            continue
        for i in I[ind]:
            lsh.append(0+(i-2*ind<=p[ind]))

    return lsh
```

4.3. LSH 检索预处理

为了检索，我们首先可以把 Dataset 文件夹里所有的图片进行 LSH 的计算，然后存储，这样可以方便检索，并且提高效率。

构造两个 List 来帮助检索。第一个 List allHash：用来存储所有不同的哈希值，注意这是一个数学意义上的集合，有互异性。第二个 List 叫做allHashFileID，这个 List 的第 i 项存储的是，以 allHash[i] 为哈希值的图片的序号的集合。即：

$$allHashFileID[i] = \{x \mid LSH(file[x]) = allHash[i]\}$$

直观的理解，第一个 List 的元素是桶的 id，第二个List的每一项是某个特定桶里元素的集合。

这样我们检索的时候就可以非常方便的得到检索结果了。

代码如下：

```
#Sub 是固定选取的子集
def PreProcessing(Sub):
    #files 是图片库
    #allHash 所有图片库里图片的 lsh 值组成的集合
    #allHashFileID[ind] 存储了所有的以 allHash[ind]为 lsh 值的图片的坐标
    #allP 是所有图片的特征向量集合
    import os
    files = os.listdir('Dataset')
    allHash = []
    allHashFileID = []
    allP = []

    for i in range(len(files)):
        imgurl = 'Dataset/' + files[i]
        p = CalcP(imgurl)
        allP.append(p)
        lsh = CalcLSH(p,Sub)

        if(lsh in allHash):
            allHashFileID[allHash.index(lsh)].append(i)
        else:
            allHash.append(lsh)
            allHashFileID.append([i])

    return files,allHash,allHashFileID,allP
```

4.4. 相似度计算

为了比较两个每个检索结果的质量，从而进行排序，我们可以利用余弦定理的思想来比较两个向量的相似程度。如果两个向量的夹角余弦越大，则夹角越小，则他们相似程度越大。计算时，可以先把两个向量进行归一化，从而方便计算，代码如下：

```
#归一化
def Normalize(vec):
    res = [0.0]*12
    s = 0
    for i in vec:
        s += i**2
    s = s**0.5
    if(s>0):
        for i in range(12):
            res[i] = float(vec[i]) / s
    return res

#计算两个向量的余弦相似度  余弦值越大  夹角越小  相似度越大
def CalcSimilarity(A,B):
    A = Normalize(A)
    B = Normalize(B)
    res = 0.0
    for i in range(12):
        res += A[i]*B[i]
    return res
```

4.5. LSH 检索

有了之前的基础，接下来的检索就是非常简单的一件事了。注意排序时利用了 lambda表达式来构造对比函数，从而对 res 的排序是由第二个关键词进行的，代码见下一页：

```
#利用 LSH 来进行搜索
#imgurl 是要查询的图片的链接
#files 是图片库
#allHash 所有图片库里图片的 lsh 值组成的集合
#allHashFileID[ind] 存储了所有的以 allHash[ind]为 lsh 值的图片的坐标
#allP 是所有图片的特征向量集合
#Sub 是固定选取的子集
def Search_LSH(imgurl,files,allHash,allHashFileID,allP,Sub):
    res = []
    p = CalcP(imgurl)
    lsh = CalcLSH(p,Sub)
    if lsh not in allHash:
        return res #如果不存在可能的图片就直接返回空集
    ind = allHash.index(lsh)#返回第一次出现lsh的坐标

    for i in allHashFileID[ind]:#所有的可能的图片
        res.append((files[i],CalcSimilarity(p,allP[i])))
    res.sort(lambda x,y:cmp(x[1],y[1]),reverse=True)
    return res
```

4.6. NN 检索

NN 检索就是对所有的图片都进行计算相似度，然后进行排序的过程，也就是暴力搜索的方法，代码很简单，如下：

```
#暴力搜索
def Search_NN(imgurl,Sub,allP):
    res = []
    p = CalcP(imgurl)
    import os
    files = os.listdir('Dataset')
    for i in range(len(files)):
        imgurl = 'Dataset/' + files[i]
        res.append((files[i],CalcSimilarity(p,allP[i])))
    res.sort(lambda x,y:cmp(x[1],y[1]),reverse=True)
    return res
```

5. 实验结果 和 实验分析

5.1. 对不同的投影集合进行测试

当 I = [1,7,10,13,15,20] 的时候，搜索结果如下

```
Search_LSH Result:
('38.jpg', 1.0)
('25.jpg', 0.8366600265340756)
('21.jpg', 0.7627700713964738)
time cost:  0.00355696678162
```

当 I = [1,7,10,13,15,20,23,4] 的时候，搜索结果如下:

```
Search_LSH Result:
('38.jpg', 1.0)
('25.jpg', 0.8366600265340756)
('21.jpg', 0.7627700713964738)
time cost:  0.00294494628906
```

搜索结果没有变，但是当 I 的维度增大的时候，搜索时间提高了很多。

当 I = [2,4,9,11,13,21]，结果发生了较大的改变，但是仍然也可以搜索到非常好的结果。而且比上一个 I 结果更好，因为排名第二的结果相似度为0.95，可见提高维度不一定可以提高准确度。

```
Search_LSH Result:
('38.jpg', 1.0)
('12.jpg', 0.948683298050514)
time cost:  0.00239586830139
```

5.2. 对比 NN 和 LSH 的检索结果和检索效率

在这里我让 NN搜索的结果只显示前10个相似度最高的项。

```
Search_LSH Result:
('38.jpg', 1.0)
('25.jpg', 0.8366600265340756)
('21.jpg', 0.7627700713964738)
time cost:  0.00372195243835

Search_NN Result:
('38.jpg', 1.0)
('12.jpg', 0.948683298050514)
('25.jpg', 0.8366600265340756)
('8.jpg', 0.808290376865476)
('26.jpg', 0.7768985960673558)
('15.jpg', 0.7745966692414834)
('28.jpg', 0.7745966692414834)
('30.jpg', 0.7745966692414834)
('7.jpg', 0.7745966692414834)
('21.jpg', 0.7627700713964738)
time cost:  0.00511980056763
```

可以看出NN 算法缺点是耗时，优点是召回率和准确率都更高，而且比较方便控制召回的数量。

LSH的优点是效率高，可以找到最相似的几个图片，但是缺点是召回的准确性不是那么好，可以看到在 LSH 的结果中排在第二位和第三位的 25.jpg 和21.jpg 的真实排名是第3和第10。而且 LSH 的结果数量不是很方便控制，如果需要返回大量结果则要找到相邻的桶中结果，这样又会降低一次准确率。

6. 实验扩展

6.1 多个投影集

在这个实验中我只是采取了单个投影集的情况，实际上如果我们重复采用多个不同的投影集，然后将结果取并集，最后进行排序，效果会更好。但是实践中发现这样的耗时会增加很多，甚至超过了NN，所以最后我没有采用这个功能。

6.2 更好的特征向量和量化方法

我们这里采用的12维特征向量是最简单的特征向量了，但是它有很多问题，一是维度太低，只把图像分成了4块。第二是它只有颜色特征，没有很多其他的特征，比如纹理等等。

6.3 更加智能的投影集选取方案

我们不能对每个查询都去调整投影集让结果更好，所以我觉得应该要找到大量的输入然后通过大量的测试，收敛到一个最合适的投影集。

7. 参考

http://www.cnblogs.com/fengfenggirl/p/lsh.html

http://blog.csdn.net/alvine008/article/details/45367727

源代码在同一个压缩包内。

非常感谢助教和何老师的耐心指导和讲解。

林禹臣

yuchenlin@sjtu.edu.cn

5140309507

2015.12.25