

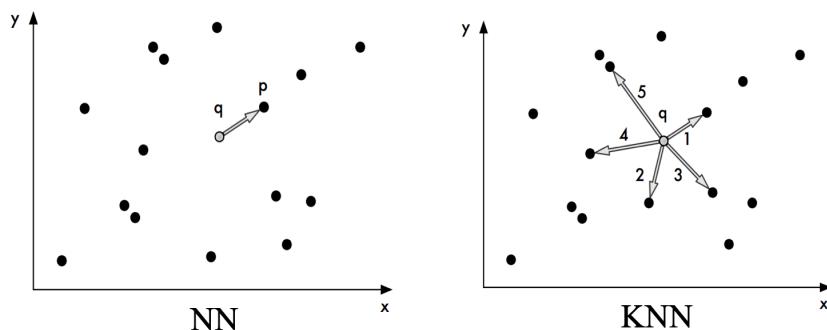
LSH图片索引

1. 原理简述

- 引入——海量比较相似度

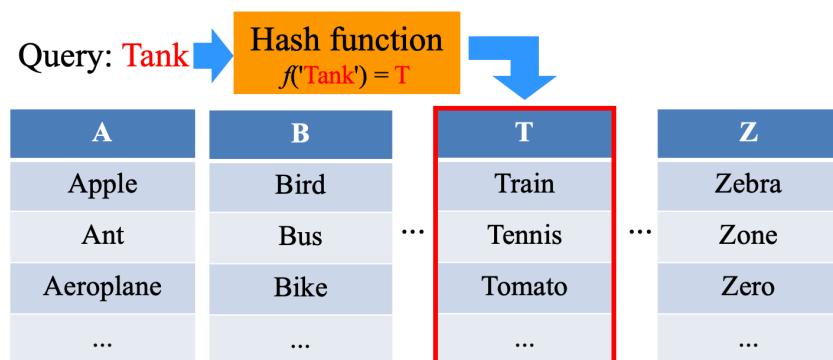
在之前的学习中我们已经学会了提取图片特征，也就是用一个（或一组）向量来表示图片特征，比如FAST、SURF、SIFT、ORB等。那么如何判断两个图片的相似度呢？自然地我们就想到算两个向量间的距离（广义的），距离越小，表示图像的特征相似度越高。而如果是两组向量，就有上次实验用到的BruteForce穷举法、FLANN近似K近邻算法（包含了多种高维特征向量匹配的算法，例如随机森林等）算法，等等。

而现在问题来了，我们不仅要比较两张图片的相似度，而是要在海量的图片中找出与待比对图片相似度最高的一组，怎么办呢？如果用Nearest neighbor(NN)或k-nearest neighbor(KNN)等充满暴力美学的方法穷举，这是O(N)算法，即使用搜索树优化了，也只是O(logN)。而感受一下平时用谷歌以图搜图的速度，对比全网图片量N，显然这是不现实的。



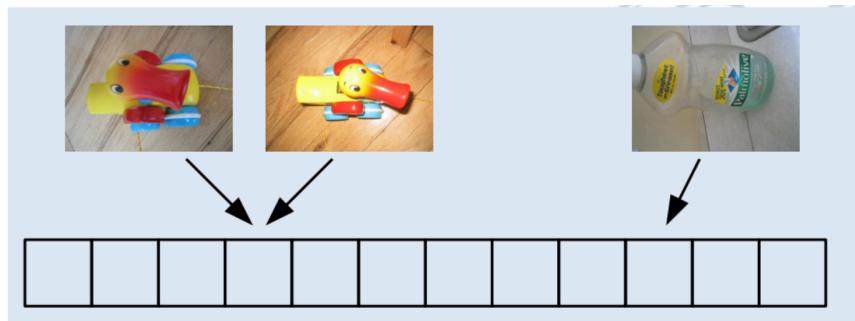
- 使用哈希方法

往往在需要搜索加速时我们会想到哈希索引。在之前的实验（对爬取网页文本进行索引，并用布隆过滤器优化储存）时我们已经了解过哈希函数的概念和特性。实际就是先用某种规则（Hash函数）把数据库中的数据分类，整个Table分为若干个不同的bucket，哈希值相同的元素放入一个bucket中；查找某元素是否在Table中，可以先算出其哈希值，类似于字典中的索引，这一步是O(1)的。然后在bucket中再去查找，由于一个bucket中的数据量远小于全部数据量，搜寻范围被极大缩小，则搜索速度加快。这叫做普通Hash方法（传统Hash方法）。



用查字典类比传统Hash方法

而这次实验中我们使用的LSH和传统Hash方法有所不同，所谓Locality-Sensitive Hashing，强调的是局部敏感。直观来说，如果两组数据原本距离近，那么在经历过Hash函数之后也更可能被放进相同的桶。反之亦然。



lsh的感性认识

• lsh中的hash function

那么如何定义“更可能”这个概率问题呢？我们这样描述hash function:

- 1) 如果 $d(x,y) \leq d_1$, 则 $h(x) = h(y)$ 的概率至少为 p_1 ;
- 2) 如果 $d(x,y) \geq d_2$, 则 $h(x) = h(y)$ 的概率至多为 p_2 ;

其中 $d(x,y)$ 表示 x 和 y 之间的距离, $d_1 < d_2$, $h(x)$ 和 $h(y)$ 分别表示对 x 和 y 进行hash变换。

满足以上两个条件的hash functions称为 (d_1, d_2, p_1, p_2) -sensitive。而通过一个或多个 (d_1, d_2, p_1, p_2) -sensitive的hash function对原始数据集合进行hashing生成一个或多个hash table的过程称为Locality-sensitive Hashing。

ok, 现在我们已经明确了hash function中概率的定义, 还剩最后一个要点, 如何度量两点间距离 $d(x,y)$ 。要注意, 不是每种距离度量方式都有其对应的hash function, 比如欧式距离就没有常规对应至桶号的。以下几种是在lsh中常见的距离度量。由于对这次实验比较感兴趣, 对其中部分距离度量进行了代码实现。

• lsh中的距离度量

- Jaccard distance

Jaccard distance: $(1 - \text{Jaccard similarity})$, 而Jaccard similarity = $(A \cap B) / (A \cup B)$ 。Jaccard similarity通常用来判断两个集合的相似性。

```
1 #计算 Jaccard distance
2 def jaccard(s1, s2):
3     x = set(s1)
4     y = set(s2)
5     return float(len(x&y))/len(x|y)
```

Jaccard distance对应的LSH hash function为minhash, 是 $(d_1, d_2, 1-d_1, 1-d_2)$ -sensitive的。

```

1 #minhash
2 def calculate_minhash(self,hash_function,s):
3     minhash = float("inf")
4     for item in s:
5         value = hash_function(item)
6         #其中生成hash_function为随机, lambda x:(a*x+b)%c
7         if value < minhash:
8             minhash = value
9     return int(minhash)
10 #若要生成某向量的signature,常常是一组hash function对应的minhash的串联

```

- Cosine distance

Cosine distance: $\cos(\theta) = \frac{A \cdot B}{\|A\| \|B\|}$, 常用来判断两个向量之间的夹角, 夹角越小, 表示它们越相似。

```

1 #计算 Cosine distance
2 def cos(s1,s2):
3     return (np.dot(s1,s2)/(np.linalg.norm(s1)*np.linalg.norm(s2)))

```

Cosine distance对应的LSH hash function为: $H(V) = \text{sign}(V \cdot R)$, R 是一个随机向量。 $V \cdot R$ 可以看做是将 V 向 R 上进行投影操作。理解成利用随机的超平面将原始数据空间进行划分, 每一个数据被投影后会落入超平面的某一侧, 经过多个随机的超平面划分后, 原始空间被划分为了很多cell, 而位于同一个cell内的数据被认为具有很大可能是相邻的。其是 $(d_1, d_2, (180-d_1)/180, (180-d_2)/180)$ -sensitive的。

- normal Euclidean distance

就是欧式距离。但是其LSH hash function需要借助引入桶宽参数。 $H(V) = \|V \cdot R + b\| / a$, R 是一个随机向量, a 是桶宽, b 是一个在 $[0,a]$ 之间均匀分布的随机变量。 $V \cdot R$ 可以看做是将 V 向 R 上进行投影操作。理解成将数据投影到一条随机直线上, 并且该直线由很多长度等于 a 的线段组成, 每一个数据被投影后会落入该直线上的某一个线段上(对应的桶内), 将所有数据都投影到直线上后, 位于同一个线段内的数据将被认为具有很大可能是相邻的。其是 $(a/2, 2a, 1/2, 1/3)$ -sensitive的。

- Hamming distance

Hamming distance: 两个具有相同长度的向量中对应位置处值不同的次数。也就是这次实验中使用的方法。

```

1 #计算 Hamming distance
2 def hamming(s1,s2):
3     return sum(ch1!=ch2 for ch1,ch2 in zip(s1, s2))/ float(len(s1))

```

Hamming distance对应的LSH hash function为: $H(V) = \text{向量 } V \text{ 的第 } i \text{ 位上的值}$, 其是 $(d_1, d_2, 1-d_1/d, 1-d_2/d)$ -sensitive的。

2.实现

- 获取图像特征向量

按照PPT的说明，一张图只需要用12维的颜色直方图来表示。一是因为数据集比较小，二是因为只是为了演示lsh算法的基本思想。实际上，简单想想就知道，12维的颜色直方图一定不能足够完备地描述一张图。

```
1 def getInt(x):
2     if x<=0.3:return 0
3     if x<=0.6:return 1
4     return 2
5
6 def getFeatureVec(imgdir):
7     img=cv2.imread(imgdir, cv2.IMREAD_COLOR)
8     FeatureVec=[]
9     row,col=img.shape[0],img.shape[1]
10    Hset=[img[:row//2,:col//2],img[:row//2,col//2+1:],img[row//2+1:,:col//2],img[row//2+1:,:col//2+1:],
11        for quater in Hset:
12            imgSum=float(np.sum(quater))
13            bgr=[np.sum(quater[:, :, 0])/imgSum,np.sum(quater[:, :, 1])/imgSum,np.sum(quater[:, :, 2])/imgSum]
14            quaterRes=[getInt(x) for x in bgr]
15            FeatureVec.extend(quaterRes)
16    return FeatureVec
```

这样就可以返回一张图的12维特征向量。为了后续步骤，要将每个分量量化到整数域，3个区间分别用0 1 2表示。量化方法现在用的是PPT上的说明。

- lsh哈希函数计算

为了使lsh方法具有更高的拓展性，这里我定义了使用Hamming距离的lsh哈希类，以后可以方便地适应不同的特征向量、哈希函数。

```
1 class hammingHasher(object):
2     def __init__(self, inLength=12, outLength=4, maxUnitValue=2, mySeeds=None):
3         if mySeeds: self.hashSeeds=mySeeds
4         else:
5             self.hashSeeds=random.sample(range(1,inLength*maxUnitValue+1),outLength)
6             self.inLength, self.outLength, self.maxUnitValue=inLength, outLength, maxUnitValue
```

仔细解读hamming距离对应的哈希函数后，我是这样理解的：比p=(0,1,2,1,0,2)，inLength=6,maxUnitValue=2,那么哈希投影向量(hashSeeds)中每个分量的是1至inLength*maxUnitValue的整数值。原向量如何投影呢？对每个分量，除maxUnitValue得到的商就是原向量中待比较值的位置，而考虑到Unary向量的定义，则余数就是目前的比较子——若待比较值≥比较子，计算结果的那一位置1，否则置0。

```

1 def hashIndex(self,inVector):
2     hashStr=""
3     for oneSeed in self.hashSeeds:
4         ind=(oneSeed-1)//self.maxUnitValue
5         left=oneSeed-ind*self.maxUnitValue
6         hashStr+=(str(int(inVector[ind]>=left)))
7     return int(hashStr,2)

```

需要注意的是，由于计算出的向量是对hashSeeds每一维的投影，为0或1，则结果是一串长度为outLength的二进制数，这里我直接将二进制数化成了10进制，目的是方便索引储存。这样，整个哈希函数就是将一个向量映射到一个0至 $2^{outLength}-1$ 的整数。

离线为整个数据库建立哈希索引：

```

1 def makeHash(self,dataset):
2     resList=[]
3     for i in range(2**self.outLength):resList.append([])
4
5     for dirpath,dirnames,filenames in os.walk(dataset):
6
7         for file in filenames:
8             imgdir=os.path.join(dirpath,file)
9             featureVec=getFeatureVec(imgdir)
10            hashInd=self.hashIndex(featureVec)
11            featureStr=' '.join([str(x) for x in featureVec])
12            resList[hashInd].append({
13                "img":imgdir,
14                "featureVec":featureStr
15            })
16    hashData=[{
17        "inLength":self.inLength,
18        "outLength":self.outLength,
19        "maxUnitValue":self.maxUnitValue,
20        "hashSeeds":self.hashSeeds
21    }
22
23    storeData=[hashData,resList]
24    with codecs.open("./dataset.json",'w')as f:
25        json.dump(storeData,f,ensure_ascii=False,indent=4)

```

这样将整个索引后的数据库储存在json文件里，后面可以在线查找。对于在线查找，这里我又建立了lshTable类，通过读取离线储存的数据库，建立lsh方法，并可以对新的数据进行查询、添加等操作。这里我们主要看看查找操作：

```
1 | class lshTable(object):
2 |     #.....
3 |     def lookup(self,imgdir):
4 |         featureVec=np.array(getFeatureVec(imgdir))
5 |         hashInd=self.hashIndex(featureVec)
6 |         minNow=float("inf")
7 |         possibleMatch=[]
8 |         for toCompare in self.resList[hashInd]:
9 |             vec=np.array([int(x) for x in toCompare["featureVec"].split(' ')])
10 |             euclid=np.linalg.norm(featureVec-vec)
11 |             if(euclid<minNow):
12 |                 minNow=euclid
13 |                 possibleMatch=[toCompare["img"]]
14 |             elif(euclid==minNow):
15 |                 possibleMatch.append(toCompare["img"])
16 |         return possibleMatch
```

先计算待查图片的向量并将其哈希至对应的索引桶内，再在桶内搜索。由于数据量较小，没有做优化，直接对桶内每个元素暴搜，按照特征向量的欧氏距离来定义距离。

3.结果

和暴力搜索作比较，我选用的方法是先用内置方法提取orb特征，然后建立基于汉明距离的暴力搜索器 `bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)`，两张图片的局部匹配体现在对两张图的orb向量进行 `bf.match`，得出的每个匹配点都有距离特征，计算所有匹配点的距离平均值，作为两张图的相似度（越小越好）。遍历文件夹中所有文件，找出相似度最好的。

```

1 #暴力搜索
2 def searchClumsy(target,dataset):
3     orb = cv2.ORB_create()
4     tarimg=cv2.imread(target, cv2.IMREAD_GRAYSCALE)
5     keypoint1, descriptor1 = orb.detectAndCompute(tarimg, None)
6     minDis=float('inf')
7     bestMatch=""
8     global kp2,nice_match
9     start=time.clock()
10    for dirpath,dirnames,filenames in os.walk(dataset):
11        for file in filenames:
12            imgdir=os.path.join(dirpath,file)
13            if not imgdir.endswith('jpg'):continue
14            toCompare=cv2.imread(imgdir, cv2.IMREAD_GRAYSCALE)
15            keypoint2, descriptor2 = orb.detectAndCompute(toCompare, None)
16            bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
17            matches = bf.match(descriptor1, descriptor2)
18            matchDis=[x.distance for x in matches]
19            avgDis=sum(matchDis)/(len(matchDis)+0.0000001)
20            if avgDis<minDis:
21                minDis=avgDis
22                bestMatch=imgdir
23                kp2=keypoint2
24                nice_match=sorted(matches, key=lambda x: x.distance)
25    end=time.clock()
26    print("best match:"+bestMatch)
27    print("costs % seconds."%(end-start))
28    colorimg1=cv2.imread(target,cv2.IMREAD_COLOR)
29    colorimg2=cv2.imread(bestMatch,cv2.IMREAD_COLOR)
30    img_match = cv2.drawMatches(colorimg1, keypoint1,colorimg2,kp2,nice_match[:30],colorimg2)
31    cv2.imwrite('img_match.jpg',img_match)

```

• 速度

搜索target.jpg，使用lsh索引的结果：

```

1 ['/Dataset/12.jpg', '/Dataset/38.jpg']
2 costs 0.00254199999999933 seconds.

```

使用暴力搜索的结果：

```

1 best match:/Users/markdana/Desktop/Exp12/Dataset/38.jpg
2 costs 0.61607 seconds.

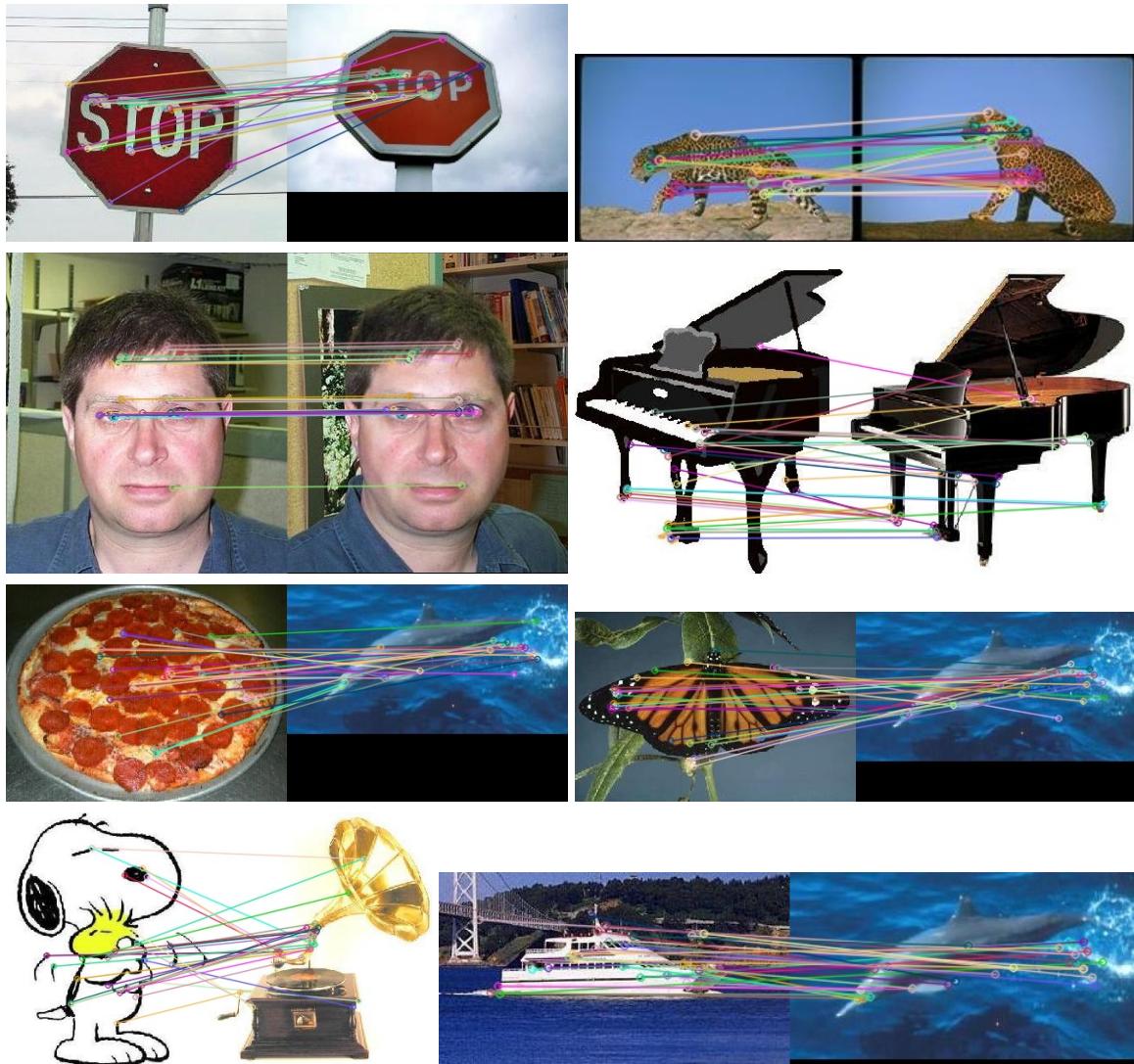
```

可以看到使用lsh的方法比暴力搜索快了25倍左右。当然直接这样比是不精确的，因为两者获得图片特征向量的方法就不一样，速度自然也不一样。但是我们可以直观思考，随着数据库越来越大，暴力搜索耗时也会越来越大，但lsh

在适当的哈希函数下，增多桶数，可以维持每个桶内数据较少而均匀，这样采用lsh索引的时间优势将越来越大。

- 精确度

首先展示暴力搜索得出的几组结果（从数据集中去除了原图）：

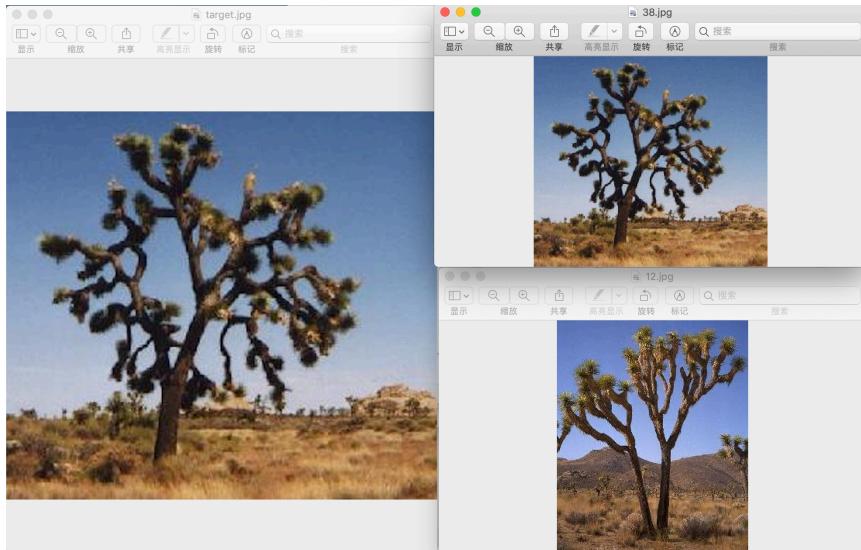


可以发现，有一些是匹配得正确的，有一些则不好。我测试了一下，20组图片里配得上的仅有15组。具体原因和改进措施会在下一部分中提及。

再展示用lsh搜索的结果，由于在桶内找12维向量间的最小距离，会返回一组可能的匹配图像。

其中有一些可以很好地匹配到：

```
1 | >> lsh.lookup("target.jpg")
2 | >> ['/Dataset/12.jpg', '/Dataset/38.jpg']
```



搜出来两组结果，都是相似的

有一些则会搜出多种结果，不有效：

```
1 | >> lsh.lookup("/Dataset/2.jpg")
2 | >> ['/Dataset/2.jpg', '/Dataset/29.jpg', '/Dataset/15.jpg', '/Dataset/21.jpg', '/Dataset/32.
```

而有一些，自己对应的匹配图像（只是背景有微调的）都匹配不到：

```
1 | >> lsh.lookup("/Dataset/26.jpg")
2 | >> ['/Dataset/23.jpg']
```



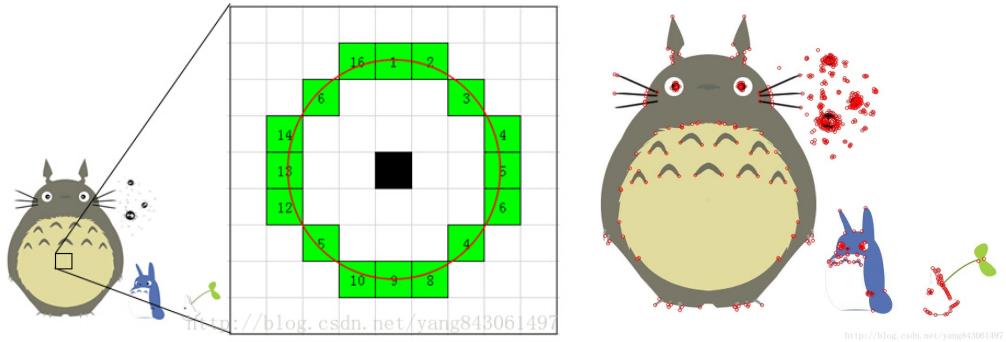
搜出来的结果并非匹配的

简单统计了一下，20组图片，能精确搜出（能搜到且精确）对应的只有10组左右。具体原因下面还会分析。

4.思考与分析

- 为何orb会出现匹配错误

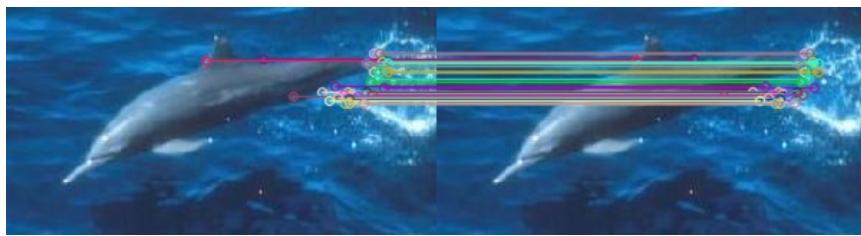
ORB采用FAST (features from accelerated segment test) 算法来检测特征点。



用FAST算法找到的特征点

然后对这些特征点，用类似于我们上次sift算子的方法，转移到物体坐标系使得特征点具有旋转不变性。但是通过上面对orb暴力搜索结果的分析，我们发现有一些是错误匹配的，为什么呢？

原因之一在于对轮廓、纹理的刻画不够强。看前四组都是轮廓比较简单、背景单一的（钢琴、路牌、豹子等），而后四组则是纹理复杂的（蝴蝶、披萨、船等，额..史努比..假装没看见它，或许是由于卡通图像轮廓点识别方法不同）。所以复杂的背景和物体本身纹理会造成较大的失真噪声。而很奇妙的是，他们都匹配到了同一张海豚图片。



看看海豚的orb找到的特征点，主要集中在水花处，而由于主体与背景相差不大，找到的特征点的向量具有较大的随机普适性，其他的图片里某些噪声或杂乱信息也很容易匹配到。

为了解决这个问题，我们可以在使用orb方法时首先对图片高斯模糊去噪，另外这些类似海豚的图片，容易让搜索进入无效的鞍点，在数据库中应格外注意。

原因之二在于丧失了图片的色彩信息。解决办法是在实际建立图片向量时，是一组轮廓（空间）特征点，一组色彩（颜色、饱和度、明亮度）特征点，同时表征一张图片。

当然，这次实验的重点并不是优化orb，因此也没有对上述的改进措施进行代码实现。

- 哈希函数族的选取

在我建立的类中，如果没有传入 `mySeeds`，则会随机生成一组哈希函数族：

```
1 | self.hashSeeds=random.sample(range(1,inLength*maxUnitValue+1),outLength)
```

但是实际测试几组后，有些效果较好，有些效果一般（很多图片被放进了一个桶中）。于是我针对40组图片的数据库，遍历了所有可能的哈希函数组，选出了几组效果最好的。所谓效果好，就是能把图片尽可能多地放进不同的桶中。找出了几组，在总共16个桶中，能放进8个及以上的：

```
1 #some so-so seeds
2 [8, [1, 3, 5, 13]]
3 [8, [1, 3, 5, 19]]
4 [9, [1, 3, 7, 13]]
5 [9, [1, 3, 7, 19]]
6 [10, [1, 3, 13, 19]]
7 [10, [1, 5, 13, 19]]
8 [12, [1, 7, 13, 19]]
```

好的哈希函数往往能申请到专利。但实际上，这几组离“好”还差很远，其一在于数据库只针对了40张图片，其二在于不能简单用桶的数目表征效果：试想，如果应该匹配的两张图却被放进了不同的桶中，那桶数确实多了，但有用吗？当然，这也是由于图像特征向量本身影响的。

另外，我们看到，在1至24的哈希函数族里，效果较好的都是奇数，原因我也会在后面解释。

- **用12维向量表示图片，为何不精确**

很容易想到，这个向量只能表征图片的色彩特征，并且只能表示rgb三原色的比重，对于许多色彩比较均匀，没有那么强的偏向（如全蓝天、全绿草、全红草莓）的图片，这个色彩向量会都是一样的。另外对于本应该匹配的两个物体，如果背景、光线不同，这个颜色向量也会有很大区别。比如下图：



向量分别为：

```
1 {"img": "3.jpg",
2 "featureVec": "0 1 1 0 1 1 0 1 1 0 1 1"}
3 {"img": "8.jpg",
4 "featureVec": "1 1 1 1 1 1 1 1 1 0 1 1"}
```

另外，由于H1-H2-H3-H4的顺序，也不具有旋转不变性。

还有一点，看似是12维，其实仔细想，真的能有 3^{12} 种不同的向量吗？看我们前面的定义，色彩占比在0-0.3的为0，0.3-0.6的为1，0.6-1的为2。那考虑一个1/4块的三维直方图，如果第一位为2，则后两位都不能为2；如果同时再第二位为1，则第三位肯定在0.1以下，也就是0。也就是说，这三维并不是相互独立的，通过其中部分可以推导出剩余的部分，那么实际上涵盖的信息并没有12维那么多，至多8维，很不全面的。

• 色彩直方图向量的优化

我们暂时先不考虑上面的这些劣势，而是看看已经获得的向量，发现其中所有的值都集中在0和1中。看看定义，色彩占比在0-0.3的为0，0.3-0.6的为1，0.6-1的为2。而对于数据集中的图片，rgb中某一色大于0.6的可能为0，也就是并不能使0，1，2平均分散开。尤为严重的是，大量图片向量为全1。

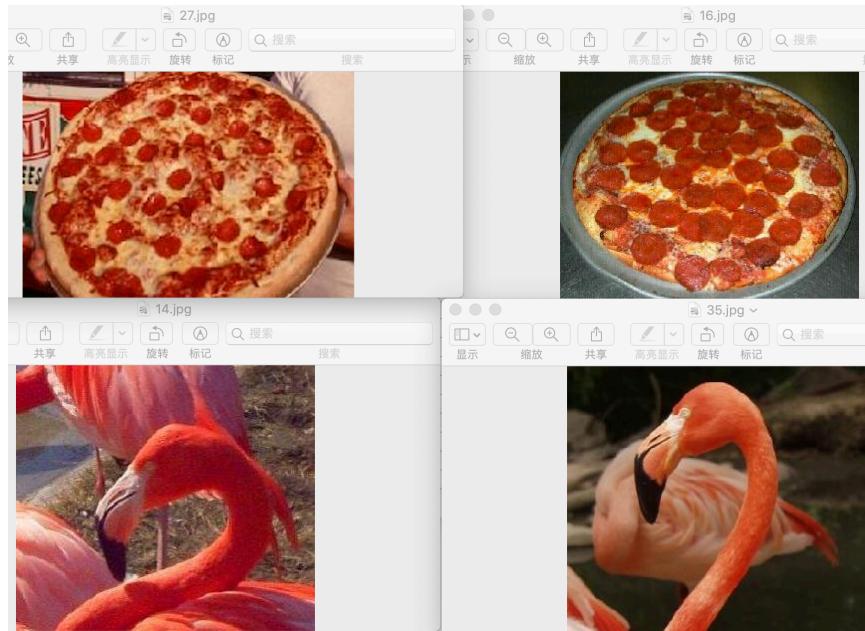
于是我统计了数据集中所有的色彩分量，用直方图的方法平均分为3类，得到新的分割方法（当然只是针对这个数据库，并且也隔得很近）：

```
1 def getInt(x):
2     if x<=0.32022380752609653: return 0
3     if x<=0.33984590482268179: return 1
4     return 2
```

用和刚才类似的方法我找了一些能够使放进尽可能多的桶的哈希函数族。但同样的，能放进不同的桶多，那匹配的图片会不会被放进了不同的桶呢？而且现在确实让0 1 2的分布均匀了，但相应的阈值也变精确了，两张本应该匹配的图现在或许由于光线的微小差别就被放在了不同的桶内，反而效果或许会变差。具体效果我们来测试几组——

比如 [4, 6, 9, 19]，数据分散到了13个桶里，有些桶内放着的图片是匹配的：

```
1 {
2     "img": "/Dataset/14.jpg",
3     "featureVec": "0 0 2 0 0 2 0 0 2 0 0 2"
4 },
5 {
6     "img": "/Dataset/16.jpg",
7     "featureVec": "0 0 2 0 0 2 0 0 2 0 0 2"
8 },
9 {
10    "img": "/Dataset/35.jpg",
11    "featureVec": "0 0 2 0 0 2 0 0 2 0 0 2"
12 },
13 {
14    "img": "/Dataset/27.jpg",
15    "featureVec": "0 0 2 0 0 2 0 0 2 0 0 2"
16 }
```



都很红的在一个桶内

但也有一些桶内只有一张图，也就是出现了刚才猜测的由于微小差别被识别进了不同的哈希值。对于得到的能进入较多桶的哈希函数族，我再次挑选出了一些匹配也比较好的：

```

1  match={5: 10, 10: 5, 21: 30, 30: 21, 15: 28, 28: 15, 25: 39, 39: 25, 16: 27, 27: 16, 22: 31,
2  #手动找的匹配
3  for mySeed in seedslist:
4      h=hammingHasher(mySeeds=mySeed)
5      h.makeHash(dataset)
6      with open(datadir,'r') as f:
7          storeData=json.load(f)
8          resList=storeData[1]
9          matchlen=0
10         for bucket in resList:
11             numlist=[]
12             for dict in bucket:
13                 string=dict["img"]
14                 num=int(re.findall(".*Dataset/(.*).jpg.*",string)[0])
15                 numlist.append(num)
16                 for x in numlist:
17                     if match[x] in numlist:
18                         matchlen+=1
19             print(matchlen,mySeed)

```

在挑选的过程中也发现，桶数少时更有利于匹配的在一个桶中，但是搜索时或许会出来很多结果；而桶数多时，匹配的或许由于微小差别进入了不同的桶，反而又搜不到了。所以要找到两方面都较符合的折中的哈希函数族。一些比较好的结果：

```
1 [3, 8, 17, 20]
2 [3, 11, 13, 20]
3 [2, 6, 10, 18]
4 [2, 10, 18, 19]
5 [1, 6, 15, 18]
6 [1, 2, 6, 18]
7 [1, 2, 4, 15]
8 [4, 6, 15, 19]
```

- 哈希索引值优化

在这里进去12维的向量，出来4维的二进制向量，也就是16个桶。数据库里40张图像对应到16个桶，这里只是做演示用。实际上哈希值（桶数）应该随着数据量的变化有相应的变化，比如十万量级以上的数据库，四维16个桶肯定是不够的。

另外，还有一种方法就是“桶中桶”，在第一层哈希索引到相应的桶后，如果还有较多的数据，可以再用另一种哈希规则，建立桶中桶，使再次加速。

5.SIFT+BOW+LSH

- 图片特征向量的优化

分析前面的改进，其实都是小修小补，因为本质上对图片特征的12维表示太差劲了。现在想用更好的方法来表示图片，因此专门拿出来作为一节。

实际上对图片做索引时，往往用一个空间轮廓特征向量和一个色彩向量同时描述一张图片。我们也可以用如之前做过的sift向量、orb向量来表示。但有一个问题，这里并不像这次实验是12维的定长向量，而是一组向量，并且互相也是无序的（出来的KeyPoints类顺序并不是特征敏感的），不能连缀成一个向量，那么怎么做hamming-lsh索引呢？

这时我们就引出一个新的概念，词袋，BOW，Bag-Of-Words。以下引用自[CSDN](#)

Bag-of-words模型是信息检索领域常用的文档表示方法。在信息检索中，BOW模型假定对于一个文档，忽略它的单词顺序和语法、句法等要素，将其仅仅看作是若干个词汇的集合，文档中每个单词的出现都是独立的，不依赖于其它单词是否出现。也就是说，文档中任意一个位置出现的任何单词，都不受该文档语意影响而独立选择的。例如有如下两个文档：

1: Bob likes to play basketball, Jim likes too.

2: Bob also likes to play football games.

基于这两个文本文档，构造一个词典：

Dictionary = {1:"Bob", 2. "like", 3. "to", 4. "play", 5. "basketball", 6. "also", 7. "football", 8. "games", 9. "Jim", 10. "too"}

这个词典一共包含10个不同的单词，利用词典的索引号，上面两个文档每一个都可以用一个10维向量表示

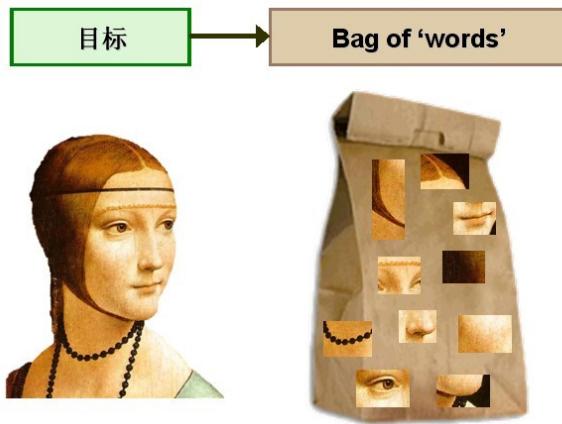
(用整数数字0~n (n为正整数) 表示某个单词在文档中出现的次数) :

1: [1, 2, 1, 1, 1, 0, 0, 0, 1, 1]

2: [1, 1, 1, 1, 0, 1, 1, 1, 0, 0]

向量中每个元素表示词典中相关元素在文档中出现的次数(下文中, 将用单词的直方图表示)。不过, 在构造文档向量的过程中可以看到, 我们并没有表达单词在原来句子中出现的次序 (这是本Bag-of-words模型的缺点之一, 不过瑕不掩瑜甚至在此处无关紧要)。

同样的, 为了表示一幅图像, 我们可以将图像看作文档, 即若干个“视觉词汇”的集合, 同样的, 视觉词汇相互之间没有顺序。



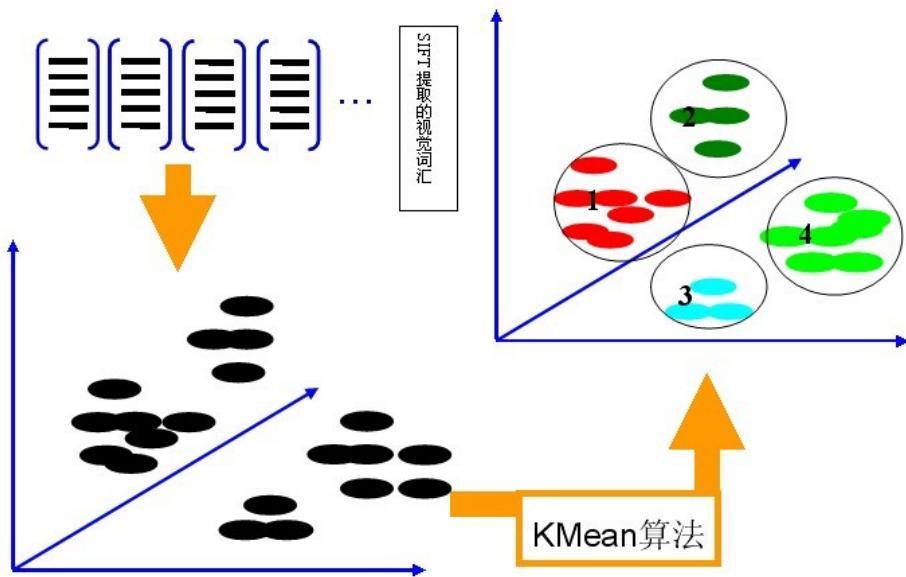
将Bag-of-words模型应用于图像表示

但是由于图片中的词汇不像文本文档那样是现成的, 需要首先从图像中提取出相互独立的视觉词汇, 也就是特征的检测和表示, 然后生成单词表。这里我们用SIFT从图像中提取不变特征点, 作为视觉词汇, 并构造单词表, 用单词表中的单词表示一幅图像。接下来的例子假设有三个目标类, 分别是人脸、自行车和吉他, 演示如何将图像表示成数值向量。



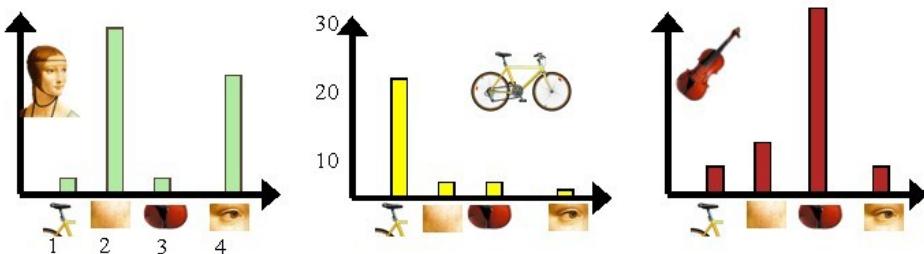
从图像中提取出相互独立的视觉词汇

对数据库中所有图像的sift特征点提取后, 放在一个集合内, 然后使用k-means算法对所有词汇进行聚类。比如这里设K=4, 则N个特征向量被分为4类, 近的被合并。



利用K-Means算法构造单词表

然后我们再回到每一幅图像，其所有特征向量都被合并到了这K维里，统计K维里各自单词出现的频率，就可以将图片用直方图向量表示：



利用K-Means算法构造单词表

于是，每一张图原本为 $x \times 256$ 维（ x 在不同图不一样）的SIFT向量们，都可以用同维向量表示了。我们实际上利用BOW完成了一次降维。

- 1 | 人脸: [3,30,3,20]
- 2 | 自行车: [20,3,3,2]
- 3 | 吉他: [8,12,32,7]

另外，在使用K-means算法的步骤，也可以用flann寻找高维数据的最近邻。

这是用调用 `sklearn` 库中的k-means：

```

1  sklearn.cluster.KMeans(
2      n_clusters=16,
3      init='k-means++',
4      n_init=10,
5      max_iter=300,
6      tol=0.0001,
7      precompute_distances='auto',
8      verbose=0,
9      random_state=None,
10     copy_x=True,
11     n_jobs=1,
12     algorithm='auto'
13 )
14 '''其中比较关键的参数：
15 n_clusters: 簇的个数，即你想聚成几类，比如这里我想让图片用16维向量表示出来
16 n_init: 获取初始簇中心的更迭次数，为了弥补初始质心的影响，算法默认会初始10个质心，实现算法，然后返回
17 max_iter: 最大迭代次数（因为kmeans算法的实现需要迭代）
18 tol: 容忍度，即kmeans运行准则收敛的条件
19 algorithm: kmeans的实现算法，有：'auto'，'full'，'elkan'，其中‘full’表示用EM方式实现
20 ...

```

假设 `data` 储存了数据库中所有图片的SIFT向量，对其KNN得到对每个向量的索引（至K维）：

```

1  from sklearn.cluster import KMeans
2
3  #data = []
4  estimator=KMeans(n_clusters=16)
5  res=estimator.fit_predict(data)
6  # 预测类别标签结果
7  label_pred=estimator.labels_
8  # 各个类别的聚类中心值
9  centroids=estimator.cluster_centers_
10 # 聚类中心均值向量的总和
11 inertia=estimator.inertia_

```

然后对每张图的每个向量，索引至k维直方图，再用适当的方法整值化就可以进行下一步：lsh索引了。

- 适用于sift特征的lsh函数

- 基于随机位抽样的局部敏感哈希 LSH Based on Random Bits Sampling
- 基于随机超平面的局部敏感哈希 LSH Based on Random Hyperplane
- 基于 p 稳定分布的局部敏感哈希 LSH Based on p-Stable Distributions
- 基于阈值的局部敏感哈希 LSH Based on Thresholding
- 谱哈希 Spectral Hashing (SH)
- 迭代量化 Iterative Quantization (ITQ)

这些都是在sift特征中较有效的lsh函数选择。很多人专门从事这方面的研究，较深奥，因此这次我就暂时不再继续了。坑先挖着，以后填。

可以看看其中一些成果：[LSHKIT](#),[LSHBOX](#)



LSHBOX项目中，SIFT+BOW+ITQ的结果