Experiment 13

517030910356 谢知晖

目录

1 实验准备

1.1 实验环境

本次实验环境为 Windows 平台下的 Python(版本号 3.6)。
另外，实验中还用到的 Python 库有:

1. OpenCV(版本号 3.4.3)
2. NumPy(版本号 1.15.4)

1.2 实验目的

终于，我们的图像处理相关实验来到了最后一步也是至关重要的一步--图像检索。

在上一次的实验中，我们对于 SIFT 算法进行了分析。该算法能够提取出以任何图片作为输入的特征向量，达到特征提取的目的。而更早的实验我们也接触到了诸如 Canny 等图像特征处理方式，这都为我们提供了提取图像特征的手段和方法。

但我们的图像处理工作还没有完成。回归到最初的那个问题: 怎么找到在" 数据库" 和所需要查找的图片相似的图片？假设我们已经对" 数据库" 中的图片提取出了所有特征，而这正是我们之前的实验的目的。那么剩下的问题是如何匹配到所要寻找的数据，并尽可能地降低检索时间。

本次实验正是基于这样一个目的开展的。通过实现 LSH(Locality-sensitive Hashing) 算法，我们能够在理想情况下以十分优秀的时间代价查询到匹配结果，这对于我们的图像检索在实际使用场景中的应用效果而言是至关重要的。

1.3 实验原理

1.3.1 一些思考

我在开始本次实验前，针对 LSH 的原理做了如下思考:

对于我们建立的数据库，当数据库十分庞大时，实际上实现合理的搜索方案是十分有挑战的。而原因也是显然的:

1. 为了满足良好的相似度匹配，我们需要保证特征向量的" 信息熵" 足够大，这不可避免地增加了计算量
2. 为了提高匹配的速度，我们不得不缩小数据的容量，这则要求我们尽可能地减少特征向量的维度

通常情况下，我们不想舍弃精确度。这引导在保证检索结果正确性的前提下尽可能地提高匹配速度。

进一步分析，匹配的时间代价主要来源于比较次数上。而这就归于一般的查找问题上了，而主流的对于庞大数据量的查找方法首选自然是哈希算法。

但一般的哈希算法并不适用于图像检索。散列表本来是应用在集合的查找中，但我们需要实现的是找到相似的图像。一种直观的改进方法是利用哈希算法的映射能力，用哈希分类数据库图像，让数据库中相似的图像映射到同一个桶中，达到数据降维的目的。

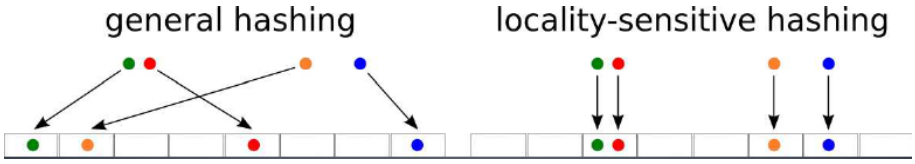


Fig1:LSH

到此为止，基本的思路已经理清，只剩下最后一点内容难以实现: 分类数据库中的相似图像信息。这终于将我引向这次实验的主题:LSH 算法

1.3.2 LSH

在这里，我不打算再讨论诸如 Hashing 思想等基本问题，而直奔主题，探究 LSH 算法究竟怎样处理相似性问题，即所谓"Locality Sensitive Hashing" 中"Locality Sensitive" 意义何在？

LSH 算法中的基本思想可以总结为以下两点: 在高维数据空间中的两个相邻的数据被映射到低维数据空间中后，将会有很大的概率仍然相邻；而原本不相邻的两个数据，在低维空间中也将有很大的概率不相邻。通过这样一映射，我们可以在低维数据空间来寻找相邻的数据点，避免在高维数据空间中寻找，因为在高维空间中会很耗时。有这样性质的哈希映射称为是局部敏感的。

由此衍生出局部敏感哈希的定义:

> 一个局部敏感哈希族 $\mathcal{H}\,(c, r, P_1, P_2)$，对于任意的 $p, q \in R_n$，有:
>
> 1. 如果 $||p-q|| \leq r$，则 $Pr_{\mathcal{H}}[h(p)=h(q)] \geq P_1$
> 2. 如果 $||p-q|| \geq cr$，则 $\mathrm{Pr}_{\mathcal{H}}[h(p)=h(p)] \leq P_2$

为了让局部敏感哈希函数族起作用，需要满足 $c>1$ 且 $P_1 > P_2$。

可以看出，LSH 算法的关键即在于这个" 局部敏感哈希函数族"。一种可行的构造方案利用了 Hamming 距离来实现上述条件。

我不打算详细讨论 Hamming 距离背后的数学意义，而仅在此指出其直观概念:

> **Hamming** 距离定义了两个位数相同的二进制数间不同的比特位数。Hamming 距离越大，我们则可以认为两个数 (或其代表的特征) 间的差异越大。

有了这样的认识，我们能够定义这样一个作用在二进制数据点上的哈希函数族 H。H 的每一个哈希函数随机地选择一个特定比特位上的值，而整个函数族包含所有从 $\{0,1\}^d$ 映射到 $\{0,1\}$ 的函数，并且有 $h_i(p)=p_i$。从 H 中随机地选择哈希函数 h，那么 $h_i(p)$ 将会随机地返回 p 的一个比特位。

对于这样的 H，$P_r[h(p)=h(q)]$ 就等于 p、q 中相同的比特位数的比例，因此有 $P_1 = 1-R/d$，$P_2 = 1-cR/d$。这样对于任意的 $c>1$，都满足 $P_1 > P_2$，即这种构造出来的哈希函数族是局部敏感的。

于是，对已有的特征向量集 (即我们的图像数据)，我们终于得到了这样一个完整的数据库构建流程:

将特征向量量化并转为 Hamming 码 → 选取 H 的适当子集作为投影 → 在投影下将所有特征向量映射到不同桶中

而检索过程也有如下实现:

计算图像特征向量 → 将特征向量投影到某一桶中 → 在该桶中找到相似度最高的特征向量

2 实验步骤

个人认为自己对于上述原理有了一个较为明确的思路，且实验内容并不复杂，实现起来不算困难。完整代码附于文末。

我依旧定义了一个类"MyLSH" 来完成实验的主体内容。该类实现了数据库的建立，即将一定量的图像数据提取特征、做 LSH，并提供查询方法。

但在介绍"MyLSH" 类之前，我首先定义了一些工具函数来辅助类的实现。

2.1 工具函数

2.1.1 get_imlist

函数"get_imlist" 只完成一项任务: 将目标文件夹"path" 下的所有文件的相对路径提取出来，以便于后续处理。

```
def get_imlist(path):
    return [os.path.join(path, f) for f in os.listdir(path) if f.endswith('.jpg')]
```

2.1.2 normalizing

函数的定义非常简单，仅仅将向量标准化，使其长度为 1。

```
def normalizing(vec):
    return vec / np.linalg.norm(vec)
```

2.1.3 get_similarity_cos

"get_similarity_cos" 用于计算向量间的余弦相似度。

该函数实际上仅仅进行了两个向量的内积计算，但出于提高代码可读性的目的，我还是将其封装为一个函数。

```
def get_similarity_cos(vec1, vec2):
    return np.inner(normalizing(vec1), normalizing(vec2))
```

2.1.4 RGB_split & get_RGB_feat

这两个函数都是用于实现最简单的特征向量的提取，即通过对图像的四个区域分别计算 RGB 能量直方图得到的 12 维向量。向量的每一维被均匀分配到 $\{0,1,2\}$ 上，其中切分的标准"low"、"high" 作为"get_RGB_feat" 的输入。

```
def RGB_split(img):
    # BGR
    b = np.sum(img[:, :, 0])
    g = np.sum(img[:, :, 1])
    r = np.sum(img[:, :, 2])
    total = r + g + b

    # RGB
    RGB = np.empty(3)
    RGB[0] = float(r) / total
    RGB[1] = float(g) / total
    RGB[2] = float(b) / total

    return RGB

def get_RGB_feat(img, low, high):
    feat_vec = np.empty(12)
    x = img.shape[0]
    y = img.shape[1]

    feat_vec[0:3] = RGB_split(img[0:int(x/2), 0:int(y/2)])
    feat_vec[3:6] = RGB_split(img[0:int(x/2), int(y/2):y])
    feat_vec[6:9] = RGB_split(img[int(x/2):x, 0:int(y/2)])
    feat_vec[9:12] = RGB_split(img[int(x/2):x, int(y/2):y])

    return feat_vec
```

2.1.5 get_vec_quantified

对于一个归一化后的特征向量，我定义了函数"get_vec_quantified" 来实现特征向量的量化。实验中我选取的 Hamming 码的量化参数为 $C = 2$。

比较特殊的是，这里我设置了一个"check_mode"，允许用户检查向量的量化效果。当在"check_mode" 下时，函数不返回量化后的向量，而是将量化的分布反馈给使用者。具体的使用场景将在下文的" 合理性检测" 中讨论。

```
def get_vec_quantified(vec, low, high, check_mode=False):
    res = np.ones(12) + 1 * (vec > high) - 1 * (vec < low)

    if check_mode:
        total_0 = np.sum(res == 0)
        total_1 = np.sum(res == 1)
        total_2 = np.sum(res == 2)
        return np.array([total_0, total_1, total_2])

    else:
        return res
```

2.1.6 get_LSH

"get_LSH" 是工具函数中较为重要的一个，它的目的在于隐式地将一个向量"vec" 通过特定的投影"proj" 映射到某一桶中，返回该桶"hash"。

如对量化到 $\{0,1,2\}$ 后的向量 $p=(0,1,2,1,0,2)$，有对应 Hamming 码 $v(p)=001011100011$。若选取投影 $\{1,3,7,8\}$，则得到结果 $(0,1,1,0)$。

```
def get_LSH(vec, proj):
    hash = []

    for i in proj:
        if i % 2 == 0:
            if vec[int((i - 1)/2)] == 2:
                hash.append(1)
            else:
                hash.append(0)

        else:
            if vec[int((i - 1)/2)] == 0:
                hash.append(0)
            else:
                hash.append(1)

    return hash
```

2.2 类的构造

构造函数接受数据的存放地址"data_path"、选取的投影集合"proj" 以及控制特征向量量化标准的"low"、"high"。

首先我对于对象的一些基本成员变量做出了初始化。

```
self.__path = data_path
self.__proj = proj
self.__imgs = get_imlist(self.__path)
self.__low = low
self.__high = high
```

然后，对象将对文件夹下的所有图像进行桶的划分。

我定义了三个成员变量来存储划分结果。"LSHs" 为包含所有桶的列表，每个 index 即为对应桶的编号；"feats_LSH" 是一个字典，它以 LSHs 的 index 为键值，每个键对应该桶中的所有特征向量 (这里除特征向量之外，我还额外存储了图像的路径，以便后续检索时能够直接显示匹配到的图像，下同)；"feats" 则为了测试 NN 搜索算法，而直接将所有特征向量放在一个桶中。

"LSHs" 和"feats_LSH" 中的桶一一对应，在查找时只需在"LSHs" 中找到桶的编号，然后对"feats_LSH" 中相应桶中的向量进行相似度计算。

```
self.__LSHs = []
self.__feats_LSH = {}
```

```
    self.__feats = []

    for img in self.__imgs:
        feat = get_RGB_feat(cv2.imread(img))
        self.__feats.append((feat, img))

        feat_quantified = get_vec_quantified(feat, self.__low, self.__high)
        LSH = get_LSH(feat_quantified, self.__proj)
        if LSH not in self.__LSHs:
            self.__LSHs.append(LSH)

        i = self.__LSHs.index(LSH)
        if i not in self.__feats_LSH:
            self.__feats_LSH[i] = []

        self.__feats_LSH[i].append((feat, img))
```

2.3 合理性检测

由于在特征向量量化时我们的划分存在一定随机性，有时候无法做到均匀划分，因此我添加了函数"check" 来简单地检测划分效果，这有助于我针对不同的数据库图像或选取的不同特征提取方法调整划分，以提高检索性能。

函数返回整个数据库中量化划分的总分布。使用 Numpy 的运算，并利用之前量化时定义的"check_mode"，该统计过程并不复杂。

```
    def check(self):
        total = np.zeros(3)
        for feat in self.__feats:
            feat = feat[0]
            total += get_vec_quantified(feat, self.__low, self.__high, check_mode=True)
        return total
```

2.4 LSH 搜索 & NN 搜索

在构建好数据库后，我便可以进行相应的检索。这里给出两种搜索方式"LSH 搜索" 和"NN 搜索"，分别由函数"LSH_search"、"NN_search" 实现。

2.4.1 LSH_search

LSH 搜索通过找到对应桶的方法滤去大量数据，仅对桶中的小部分数据进行搜索。

函数接受两个参数"img"、"display"，其中"img" 为待检索图像，"display" 为一布尔值，表示是否将最相近结果显示出来。

我首先使用同样的方式对输入图像进行特征提取，计算分桶。如果在数据库中没有这个桶，则认为不存在相似的结果，返回一空列表。

```
    img_feat = get_RGB_feat(img)
    img_feat_quantified = get_vec_quantified(img_feat, self.__low, self.__high)
    img_LSH = get_LSH(img_feat_quantified, self.__proj)
```

```
res = []
if img_LSH not in self.__LSHs:
    return res
```

如果找到了对应的桶，则直接在桶中进行查找。对所有数据库图像按照相关度进行排序，并得到相关度最高的图像路径"best_res_location"。

```
i = self.__LSHs.index(img_LSH)
for vec, location in self.__feats_LSH[i]:
    score = get_similarity_cos(img_feat, vec)
    res.append((score, location))

res.sort(key=lambda x: x[0], reverse=True)
best_res_location = res[0][1]
```

为了让使用者看到检索结果，以方便算法的调试，我对最佳结果做了可选的"display"。而函数的返回结果包含最佳匹配结果的路径以及排序后的相似度信息 (由于仅在一个桶中执行了检索，数据库中的其他图像相似度没有给出)。

```
if display:
    cv2.imshow("Result", cv2.imread(best_res_location))
    k = cv2.waitKey(0)
    if k == 'q':
        cv2.destroyAllWindows()

return [best_res_location, res]
```

2.4.2 NN_search

NN 搜索即对每一张图像进行相似度计算。在优化情况下能够达到 O(log n) 的时间复杂度，但这里我仅给出了 O(n) 的算法，即暴力搜索。

"NN_search" 的实现和"LSH_search" 类似，在此不再赘述。

3 实验结果

3.1 不同投影集合

我在数据集上"Dataset" 上进行此项测试。调试特征向量的量化划分标准为 $low = 0.32, high = 0.345$，得到 $(162, 158, 160)$ 的分布，在此基础上进行比较。

显然，我们所期望的理想投影集合应满足如下需求: 在保证一定召回率的前提下，尽量召回相似度高的图像。

但在实验中，我发现寻找这样的投影集合并不容易。例如，对于集合 $\{2, 4, 11, 13, 21\}$，我的 LSH 搜索返回了如下结果:

```
(0.9999999999999998, 'Dataset\\38.jpg')
(0.9987892968220986, 'Dataset\\12.jpg')
(0.991372190724903, 'Dataset\\7.jpg')
(0.9899322714672495, 'Dataset\\15.jpg')
(0.9898721547358234, 'Dataset\\28.jpg')
Time Cost: 0.0008967546543454465
```

Fig2:LSH(投影集合 1)

与 NN 搜索结果进行对比[1]，可以发现返回的并不是相似度最高的 5 个。
使用另一集合 $\{1, 6, 11, 16, 21\}$，得到了更为糟糕的结果。

```
(0.9999999999999998, 'Dataset\\38.jpg')
(0.9987892968220986, 'Dataset\\12.jpg')
(0.9921124735686665, 'Dataset\\26.jpg')
(0.9917060481455761, 'Dataset\\40.jpg')
(0.9898667198289556, 'Dataset\\17.jpg')
(0.910614812575912, 'Dataset\\37.jpg')
Time Cost: 0.00042893191045744307
```

Fig3:LSH(投影集合 2)

召回的结果中出现了相似程度很低的"37.jpg"，这对于分桶结果来说是十分不利的。
这可能由于投影集合过小，导致出现一些额外噪声，而实际上相似的图像并没有很多。为此，我又尝试了另一集合 $\{1, 2, 13, 16, 18, 21\}$，通过增加集合容量过滤掉噪声。

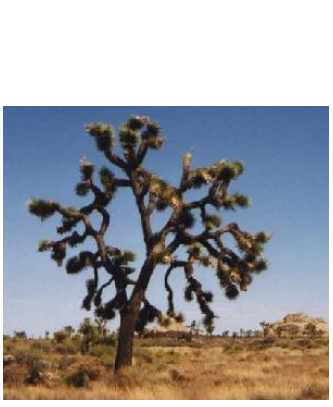
```
(0.9999999999999998, 'Dataset\\38.jpg')
(0.9987892968220986, 'Dataset\\12.jpg')
Time Cost: 0.00242255909990753
```

Fig4:LSH(投影集合 3)

这次我们仅召回了两张图像，而经校验这两张图确实是"Dataset" 中唯二与查找图像相似的。这正满足了我们的需求。
但同时值得注意的是，提高召回性能也以增加用时为代价，集合 3 的检索时间明显大于前两次检索。这提示我权衡好两者。

3.2 LSH 搜索和 NN 搜索的对比

为了更明显地体现两者之间的区别，我换用了较为庞大的数据集[2]。
搜索对象以及预期相似度最高的结果分别为:



(a) 搜索对象

(b) 最佳结果 (38.jpg)

(c) 次佳结果 (12.jpg)

Fig5: 期望搜索结果

[1]附在文末。
[2]由约 3800 张人脸图、500 张景点图、一些干扰项以及三张目标图片组成。这些附加的图像与我们所要检索的内容并没有关联。

同样地，我调整 $low = 0.3, high = 0.34$，使分布为 $(16355, 17323, 17334)$，投影集合则选用 $\{2, 4, 9, 11, 13, 17, 21\}$。

由于召回结果较多，我仅关注前 10 个结果。NN 搜索和 LSH 搜索的结果分别如下:

```
(0.9999999999999998, 'database\\38.jpg')
(0.9992793820886945, 'database\\431.jpg')
(0.9989788747916283, 'database\\499.jpg')
(0.9988519634029527, 'database\\427.jpg')
(0.9987892968220986, 'database\\12.jpg')
(0.9987466942560567, 'database\\98.jpg')
(0.9986146910763523, 'database\\279.jpg')
(0.9985984639128656, 'database\\158.jpg')
(0.9985473463910826, 'database\\119.jpg')
(0.9985428170065448, 'database\\4023.jpg')
Time: 0.04818800810590691
```

(a) NN

```
(0.9999999999999998, 'database\\38.jpg')
(0.9989788747916283, 'database\\499.jpg')
(0.9987892968220986, 'database\\12.jpg')
(0.9986146910763523, 'database\\279.jpg')
(0.9985984639128656, 'database\\158.jpg')
(0.998199046292549, 'database\\479.jpg')
(0.9979826429352612, 'database\\338.jpg')
(0.9979671931682186, 'database\\488.jpg')
(0.9977832446194431, 'database\\94.jpg')
(0.9977108419604147, 'database\\318.jpg')
Time: 0.0009526366285951838
```

(b) LSH

Fig6: NN & LSH

对于实验结果数据，我做了两方面分析:

3.2.1 准确度 & 召回率

首先最为直接的一个发现是: 我采用的特征提取算法有待改进。在 NN 搜索中，虽然 38.jpg 被正确地判断为最佳结果，但 12.jpg 并没有如预期那样作为第二好的结果被召回。当然，这也是情有可原的，毕竟实验中使用的特征向量仅为 12 维，且在提取中缺失了许多图像的特征信息。

如果假设我们的特征向量能代表图像的特征，NN 搜索的准确度无疑是比 LSH 高的，因为它对数据库中的所有图像都进行了一一比对，而 LSH 搜索仅比对了部分内容。与之对应的则有 LSH 搜索的召回率小于 NN 搜索。

但又有这样一种思考: 如果我们的特征提取存在一定误差 (即得到的特征向量无法精确地反映图像本质特征)，LSH 搜索是否在某一程度上是以降低召回率为代价提高结果可靠性呢? 我认为并不尽然[3]。

3.2.2 耗时

时间代价的优劣则毫无悬念。仅在 4000 多张图片时，LSH 搜索就比 NN 搜索快了 50 倍左右，而随着数据量的增大，之间的差距还将拉大。这说明 LSH 搜索在面对庞大数据量时有着很明显的速度优势，这也是我们采用这种算法的原因。

4 结果思考

可以说，这次实验的结果是较为良好的。但在实验过程中有不少回头优化的情况，也有许多能够改进之处尚未能够完成。

4.1 运算优化

在最开始初步实现我的 LSH 算法时，由于自己对于 Numpy 的接触不足，我没有使用 Numpy 的数组对象来处理向量，这让我承受了许多性能上的代价。于是，对于所有的向量结

[3]进一步讨论放在结果思考中。

构，我都换用数组来处理，这使得运算的便利性和效能都得到很大提升。

4.2 结构优化

一个很清楚的认识是，数据库的检索效率是我们关注的首要内容，因此检索的执行应该尽可能直接。

为此，我将数据库中的图像处理为最为直接的特征向量，查找时只需对待查询图像进行特征提取，并在两个向量间进行相似度计算。

4.3 LSH 召回结果

在之前的实验中我们可以看到，LSH 召回率低，但在结果中预期的次佳结果 12.jpg 的排名是要高于 NN 搜索中的排名的。

对于这样的结果，我最初认为 LSH 搜索在一定程度上加强了结果的可靠性。但经过一定分析，我发现了这种观点的错误。

首先，" 局部敏感" 的哈希方法潜在地隐含着对特征向量的局部特征提取。更直接地，我们可以认为在 LSH 算法的分桶过程中，原始的图像经过了两次" 采样"，导致两次" 失真"。最直接的依据是我们的图像特征经历了如下降维过程 (假设原始图像大小为 256×256，投影集合 Ω 有 $|\Omega| = 6$):

$$(256 \times 256 \times 3) \to (12 \times 1) \to (6 \times 1)$$

$(256 \times 256 \times 3)$ 的图像最终被降到 (6×1) !

另外，由于缺少先验，投影有时也可能完全失效。比如对于一个特征向量，如果投影集合中的元素都对应于图像中的某一个区域或某一部分特征上，使这一部分特征的权重变的很大，则将损坏原来的特征。

由此可见，LSH 结果可靠与否是有很大随机性的，但其结果可靠程度始终无法超越 NN 搜索中一一比对得到的结果。

4.4 合适的哈希

分析实验结果可以发现，合适的哈希函数 (即投影集合) 能够明显地改善检索结果，优化搜索召回率和准确率。

但是，寻找到最佳的投影集合并不是件容易的事情。正如上面讨论 LSH 召回结果时我提到的，我们能做到的只是让投影集合尽量保留前一步提取出的特征信息，这要求投影集合保持" 公正"。

一种简单的手段是使投影均匀地分布，使特征信息更多地得到考虑。但更好的方法应该建立在一定的先验知识上，通过映射更有效的特征区域来提高投影集合的可靠度，但先验知识的获取又是另一件难事了。

5 附页

5.1 NN 搜索结果

```
(0.9999999999999998, 'Dataset\\38.jpg')
(0.9987892968220986, 'Dataset\\12.jpg')
(0.993213225230763, 'Dataset\\23.jpg')
(0.9921124735686665, 'Dataset\\26.jpg')
(0.9917060481455761, 'Dataset\\40.jpg')
(0.991372190724903, 'Dataset\\7.jpg')
(0.9906729821815805, 'Dataset\\25.jpg')
(0.9905860191950103, 'Dataset\\8.jpg')
(0.9899322714672495, 'Dataset\\15.jpg')
(0.9898721547358234, 'Dataset\\28.jpg')
(0.9898667198289556, 'Dataset\\17.jpg')
(0.9897083135455338, 'Dataset\\21.jpg')
(0.989683528534769, 'Dataset\\30.jpg')
(0.9896239666865558, 'Dataset\\5.jpg')
(0.9892420789709906, 'Dataset\\32.jpg')
(0.9882097728247885, 'Dataset\\2.jpg')
(0.9864998494476148, 'Dataset\\29.jpg')
(0.9835693923648465, 'Dataset\\4.jpg')
(0.9826677503093731, 'Dataset\\24.jpg')
(0.9825827238294593, 'Dataset\\34.jpg')
(0.9821598963067601, 'Dataset\\31.jpg')
(0.9810835116347453, 'Dataset\\20.jpg')
(0.980005101723699, 'Dataset\\18.jpg')
(0.9774497471762597, 'Dataset\\33.jpg')
(0.9763414203807114, 'Dataset\\39.jpg')
(0.9749217942712048, 'Dataset\\10.jpg')
(0.9745766106333785, 'Dataset\\13.jpg')
(0.9720360733306072, 'Dataset\\9.jpg')
(0.9705675245105053, 'Dataset\\1.jpg')
(0.9685112198257312, 'Dataset\\19.jpg')
(0.9644171107357669, 'Dataset\\6.jpg')
(0.9573811980630943, 'Dataset\\36.jpg')
(0.956103996205684, 'Dataset\\3.jpg')
(0.9489669073602338, 'Dataset\\22.jpg')
(0.9259681191882936, 'Dataset\\35.jpg')
(0.9133307557306408, 'Dataset\\27.jpg')
(0.9120390111553263, 'Dataset\\14.jpg')
(0.910614812575912, 'Dataset\\37.jpg')
(0.9017529660458945, 'Dataset\\16.jpg')
(0.8657759153122305, 'Dataset\\11.jpg')
```

Fig2:NN 搜索结果

5.2 源代码

```
import cv2
import numpy as np
import os
import time



# Create a database based on a dataset
class MyLSH(object):
```

```
def __init__(self, data_path, proj, low=0.32, high=0.35):
    self.__path = data_path
    self.__proj = proj
    self.__imgs = get_imlist(self.__path)
    self.__low = low
    self.__high = high

# Get all LSHs
    self.__LSHs = []
    self.__feats_LSH = {}
    self.__feats = []

    for img in self.__imgs:
        feat = get_RGB_feat(cv2.imread(img))
        self.__feats.append((feat, img))

        feat_quantified = get_vec_quantified(feat, self.__low, self.__high)
        LSH = get_LSH(feat_quantified, self.__proj)
        if LSH not in self.__LSHs:
            self.__LSHs.append(LSH)

        i = self.__LSHs.index(LSH)
        if i not in self.__feats_LSH:
            self.__feats_LSH[i] = []

        self.__feats_LSH[i].append((feat, img))

def LSH_search(self, img, display=False):
    img_feat = get_RGB_feat(img)
    img_feat_quantified = get_vec_quantified(img_feat, self.__low, self.__high)
    img_LSH = get_LSH(img_feat_quantified, self.__proj)

    res = []
    if img_LSH not in self.__LSHs:
        return res

    i = self.__LSHs.index(img_LSH)
    for vec, location in self.__feats_LSH[i]:
        score = get_similarity_cos(img_feat, vec)
        res.append((score, location))

    res.sort(key=lambda x: x[0], reverse=True)
    best_res_location = res[0][1]

    if display:
        cv2.imshow("Result", cv2.imread(best_res_location))
        k = cv2.waitKey(0)
        if k == 'q':
            cv2.destroyAllWindows()

    return [best_res_location, res]
```

```
    def check(self):
        total = np.zeros(3)
        for feat in self.__feats:
            feat = feat[0]
            total += get_vec_quantified(feat, self.__low, self.__high, check_mode=True)
        return total

    def NN_search(self, img, display=False):
        img_feat = get_RGB_feat(img)
        res = []

        for vec, location in self.__feats:
            score = get_similarity_cos(img_feat, vec)
            res.append((score, location))

        res.sort(key=lambda x: x[0], reverse=True)

        best_res_location = res[0][1]

        if display:
            cv2.imshow("Result", cv2.imread(best_res_location))

            k = cv2.waitKey(0)
            if k == 'q':
                cv2.destroyAllWindows()
        # [best, allresult]
        return [best_res_location, res]


def get_imlist(path):
    return [os.path.join(path, f) for f in os.listdir(path) if f.endswith('.jpg')]


def normalizing(vec):
    return vec / np.linalg.norm(vec)


def RGB_split(img):
    # BGR
    b = np.sum(img[:, :, 0])
    g = np.sum(img[:, :, 1])
    r = np.sum(img[:, :, 2])
    total = r + g + b

    # RGB
    RGB = np.empty(3)
    RGB[0] = float(r) / total
    RGB[1] = float(g) / total
    RGB[2] = float(b) / total

    return RGB
```

```
def get_similarity_cos(vec1, vec2):
    return np.inner(normalizing(vec1), normalizing(vec2))


def get_RGB_feat(img):
    feat_vec = np.empty(12)
    x = img.shape[0]
    y = img.shape[1]

    feat_vec[0:3] = RGB_split(img[0:int(x/2), 0:int(y/2)])
    feat_vec[3:6] = RGB_split(img[0:int(x/2), int(y/2):y])
    feat_vec[6:9] = RGB_split(img[int(x/2):x, 0:int(y/2)])
    feat_vec[9:12] = RGB_split(img[int(x/2):x, int(y/2):y])

    return feat_vec


def get_vec_quantified(vec, low, high, check_mode=False):
    res = np.ones(12) + 1 * (vec > high) - 1 * (vec < low)

    if check_mode:
        total_0 = np.sum(res == 0)
        total_1 = np.sum(res == 1)
        total_2 = np.sum(res == 2)
        return np.array([total_0, total_1, total_2])

    else:
        return res

# Implicit
def get_LSH(vec, proj):
    hash = []

    for i in proj:
        if i % 2 == 0:
            if vec[int((i - 1)/2)] == 2:
                hash.append(1)
            else:
                hash.append(0)

        else:
            if vec[int((i - 1)/2)] == 0:
                hash.append(0)
            else:
                hash.append(1)

    return hash


def main():
    data_path = 'Dataset'
```

```
    target = 'target.jpg'
    search_mode = 'LSH'  # 'NN'
    res_num = 10
    img = cv2.imread(target)

    database = MyLSH(data_path, [2, 4, 9, 11, 13, 17, 21], low=0.3, high=0.34)

    if search_mode == 'LSH':
        count = 0
        start = time.clock()
        res = database.LSH_search(img, display=False)
        cost = time.clock() - start

        for i in res[1]:
            print(i)
            count += 1
            if count >= res_num:
                break

        print("Time: {0}".format(cost))

    if search_mode == 'NN':
        count = 0
        start = time.clock()
        res = database.NN_search(img, display=False)
        cost = time.clock() - start

        for i in res[1]:
            print(i)
            count += 1
            if count >= res_num:
                break

        print("Time: {0}".format(cost))


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

参考文献

[1] 局部敏感哈希: https://blog.csdn.net/yc461515457/article/details/48845775