目录

1 实验介绍

1.1 实验内容

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.

1.2 实验环境

操作系统: Ubuntu 18.04 LTS

Python 版本: 3.7

NumPy: 1.14.5

OpenCV-Python: 3.4.3.18

2 实验过程

2.1 LSH 原理

LSH (Locality-sensitive hashing, 局部敏感哈希), 可以有效地对高维数据进行降维处理. 不同于通常的哈希算法,LSH 致力将相似的高维输入数据映射为相同的低维结果, 而不相似的输入则产生不同的哈希结果. 利用此特点, 我们可以将其用于图像的检索. 其主要步骤为:

1. 提取图像的特征向量

2.LSH 预处理

3.LSH 检索

2.2 图像特征提取

在前几次的实验中, 我们实现了图像的直方图特征提取、SIFT 图像特征提取等, 而在本次实验中, 我选用了 ORB 特征.

ORB 具有旋转不变性, 对噪声不敏感, 同时计算速度很快 (速度约为 SIFT 的 100 倍,SURF 的 10 倍), 适合在本次实验中使用. 由于本次实验的重点不在于此, 直接使用 OpenCV 所提供的 API, 如下:

```python
def orb_feature(img_path, max_kp):
    img = cv2.imread(img_path)
    detector = cv2.ORB_create(max_kp)  # limit the number of keypoints
    kp, des = detector.detectAndCompute(img, None)
    return des
```

2.3 LSH 预处理

在上一步的特征提取后, 对于一张图片, 我们得到了形状为 $(max\_kp, 32)$ 的特征张量, 将其展开为 $d = max\_kp \times 32$ 维的特征向量, 进行 LSH 预处理.

根据 LSH 的思想, 我们将 d 维非负整数向量 $\mathbf{p}$ 映射到 $d^{'} = d * C$ 维的 Hamming 空间:

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

式中,C 为 p_i 的最大值,$\mathbf{Unary}_C(p_i)$ 表示一个前 p_i 位为 1, 其余 $C - p_i$ 位为 0 的二进制数.

然后, 选取合适的投影集合 $I = \{i_1, i_2, \dots, i_m\}, 1 \le i_i < i_2 < \dots < i_m \le d^{'}$, 定义 $v(\mathbf{p})$ 在 I 上的投影为:

$$g(\mathbf{p}) = p_1 p_2 \dots p_m$$

其中 p_j 为 $v(\mathbf{p})$ 的第 i_j 个元素 (取值为 0 或 1).

在具体的程序实现中, 其实可以直接根据 $\mathbf{p}$ 计算出 $g(\mathbf{p})$, 无需将 $\mathbf{p}$ 转换到 Hamming 空间, 做法如下:

```python
def compute_LSH(feature, I):
    feature = feature.reshape((-1,))
    d, C = feature.shape[0], 255
    result = list()
```

```
    for I_i in I:
        i = int((I_i - 1)//C + 1)  # i = 1,2,...,d
        x_i = feature[i - 1]
        if I_i <= x_i + C * (i - 1):
            result.append(1)
        else:
            result.append(0)
    result = np.array(result)
    return result
```

经过 LSH 处理, 我们得到了一个相对低维的向量, 再进一步, 我们二次哈希, 将其转换为一个一维整数, 加快检索速度:

```
def compute_hash(vector, mod):
    result = 0
    for i, x in enumerate(vector):
        result = (result + (i + 1) * x) % mod
    return result
```

2.4 LSH 检索

对于给定的目标图片, 我们计算其 ORB 特征与哈希结果; 对于众多的候选图片, 我们同样地计算每一张图片的 ORB 特征与哈希结果. 那么,LSH 检索便只需比较其哈希结果, 若出现相同的哈希结果, 则认为这张图片极有可能相似于目标图片:

```
def LSH_match(target_img, dataset_folder):
    # preprocessing...
    target_hash, target_feature = LSH(target_img)
    dataset_hash = list()
    print("The Hash value of the target image:", target_hash)
    for dir_path, dir_names, file_names in os.walk(dataset_folder):
        for file_name in file_names:
            img_path = os.path.join(dir_path, file_name)
            img_hash, img_feature = LSH(img_path)
            dataset_hash.append((img_path, img_hash, img_feature))
    # searching...
    tick = time.time()
    possible_result = list()
    for img_path, img_hash, img_feature in dataset_hash:
        if(img_hash == target_hash):
            possible_result.append((img_path, img_hash, img_feature))
    if not possible_result:
```

```
        print("No image matched!")
        return
    # omitted here
    # ...
```

然而, 可能的情况是会出现多张图片具有相同的哈希值. 那么, 需要进一步的特征比对以确认. 考虑到一般情况下, 哈希函数分布基本均匀, 具有相同哈希值的图片数量不会太多, 所以在这里, 我采用了最朴素的暴力搜索匹配法:

```
def LSH_match(target_img, dataset_folder):
    # ...
    # omitted here
    best_match = None
    for img_path, img_hash, img_feature in possible_result:
        bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
        matches = bf.match(target_feature, img_feature)
        if not best_match or len(matches) > best_match[1]:
            best_match = (img_path, len(matches))
    print("The best match is:", best_match[0])
```

2.5 NN 检索

作为本次实验的对照, 我同样实现了简单的 NN (Nearest neighbor, 最近邻) 检索. 同样地, 将 NN 检索分为预处理与检索两部分.

在预处理时, 计算目标图片与所有候选图片的 ORB 特征; 而在检索时, 则比较目标图像的特征与每一张候选图片特征的相似程度, 取最高相似度的候选图片作为检索的结果. 具体程序实现如下:

```
def NN_match(target_img, dataset_folder):
    # preprocessing...
    target_feature = orb_feature(target_img, 100)
    dataset_feature = list()
    for dir_path, dir_names, file_names in os.walk(dataset_folder):
        for file_name in file_names:
            img_path = os.path.join(dir_path, file_name)
            img_feature = orb_feature(img_path, 100)
            dataset_feature.append((img_path, img_feature))
    # searching...
    best_match = None
    for img_path, img_feature in dataset_feature:
        bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
        matches = bf.match(target_feature, img_feature)
        if not best_match or len(matches) > best_match[1]:
```

```
            best_match = (img_path, len(matches))
    print("The best match is:", best_match[0])
```

3 实验结果

3.1 投影集合与搜索结果

在 LSH 算法中, 投影集合 I 的选取是一件相对复杂的事情. 一个合适的投影集合, 应当有合适的元素个数, 若元素过少, 则无法充分表现图像的特征, 若元素过多, 则 LSH 预处理的时间复杂度会增加, 同时也不一定能更好地表现图像特征; 此外, 投影集合还需要使得哈希结果有着相对均匀的分布, 否则会造成算法的退化, 无法充分发挥哈希的速度优势.

在实验中, 一开始, 我设置投影集合内元素为一个等差数列, 均分 0 到 d' 的空间, 如下:

```
I = np.linspace(0, 100*32*255, 100, dtype="int32")
```

但这样做在运行时会时不时出错, 原因便在于 ORB 特征提取并不一定每次都能提取到 max_kp 个特征点. 那么, 保险起见, 我假定只能提取到 max_kp 一半的特征点, 也就是说, 投影元素的最大值设置为 $50 \times 32 \times 255$. 这样之后, 程序便可顺利运行, 不再出错了.

然后再考虑投影集合的元素个数. 对于本次实验, 由于图片总量并不大, 故投影集合无需太大, 因此我设置为 100. 实验验证, 这样做的效果已经令人满意.

程序运行结果如下:

```
The Hash value of the target image: 52
dataset\13.jpg 46
dataset\38.jpg 100
dataset\39.jpg 43
The best match is: dataset\38.jpg
```

可以看到, 与目标图像具有相同哈希值的图像仅有 3 张, 我们只需在其中进行暴力匹配, 从而提高了检索的效率. 而最后的结果也是完全正确的.

3.2 LSH 与 NN 运行效率比较

若计算 LSH 与 NN 检索的总用时, 即包括预处理与检索两个阶段, 使用装饰器实现运行时间计算:

```
def timer(func):
    @functools.wraps(func)
    def wrapper(*args, **kw):
        tick = time.time()
        func(*args, **kw)
        tock = time.time()
        print("Time: {:.4f}s".format(tock-tick))
    return wrapper

@timer
def LSH_match(target_img, dataset_folder):
    # omitted
```

运行结果为:

```
LSH Time: 0.5326s
 NN Time: 0.2394s
```

出乎意料地,LSH 检索并未显出优势. 在仔细确认之后, 我认为问题在于 LSH 在预处理阶段比 NN 具有更高的时间复杂度, 尤其是本次实验中, 使用 Python 实现 LSH 预处理, 相对于 OpenCV 中 C++ 实现的 ORB 特征提取, 运行时占了较多的时间, 以至于完全抵消了 LSH 在检索时的优势.

为了确认上述想法, 修改程序, 仅统计程序在检索阶段的用时, 不考虑预处理用时, 则结果如下:

```
LSH search: 0.008976s
NN search: 0.005985s
```

显而易见, 本次实验中 LSH 搜索的主要耗时在于预处理阶段, 粗略计算, 约占 83%. 在去除了预处理用时后,LSH 的优势方才显现.

4 总结与分析

在本次实验中, 我实现了基本的 LSH 检索. 对于 LSH 检索, 其优势在于大数据量情境下的快速检索, 相较于 NN 检索, 虽然准确度有所下降, 但效率大幅提升.

虽然本次实验中候选图片集较小,LSH 检索与 NN 检索用时相差无几, 但在更大数据量时,LSH 显然是更优的选择.