Lab3-LSH

1. 实验概览

1.1 实验目的

- 理解并实现局部敏感哈希（LSH）用于近似最近邻（Approximate Nearest Neighbor, ANN）检索。
- 将 LSH 与暴力最近邻（Brute-force NN）在检索精度与速度上进行比较。
- 掌握如何把图像表示为低维、量化的特征向量并用哈希表进行检索。

1.2 实验原理

- **特征表示**
  将每张图像转化为 **12 维量化颜色能量直方图**（把图像缩放到 200×200，分为 4 个象限，每个象限计算 B/G/R 能量占比并量化为 {0,1,2}）。该表示简单、紧凑且便于用 Hamming哈希处理。
- **LSH 基本思想**
  将高维离散特征通过若干随机投影/选择（实现为选择特定的二进制位索引集合）构成多个哈希表（每个哈希表对应一个 `g` 函数），查询时从这些表取候选集并在候选集中做精确距离度量以返回近似最近邻。
- **评估指标**
  - **准确性/召回（Recall）**：LSH 返回的结果是否等同于暴力 NN（是否检索到真正的最近邻）。
  - **查询时间**：暴力搜索 vs LSH（平均查询时间，ms）。

2 算法设计与实现

下面逐步说明代码中每个主要函数的功能、输入输出、关键参数与实现要点。

2.1 特征提取：`compute_color_histogram(image)`

- 将输入 BGR 图像缩放到 200×200，分成 4 个象限（左上、右上、左下、右下），每个象限分别计算 B、G、R 三通道的能量（像素值之和），归一化为每通道占比，然后对每个象限的

3 个占比做 33%/66% 分位点量化，最终返回长度最多 12（4×3）的整数向量（值为 0、1、2）。返回 `np.array`，dtype=int，形状 (12,)。

```
    def compute_color_histogram(image):
        image = cv2.resize(image, (200, 200))
        h, w = image.shape[:2]
        mid_h = h // 2
        mid_w = w // 2
        regions = [image[0:mid_h, 0:mid_w], image[0:mid_h, mid_w:w],
                   image[mid_h:h, 0:mid_w], image[mid_h:h, mid_w:w]]
        hists = []
        for region in regions:
            b, g, r = cv2.split(region)
            b_energy = np.sum(b)
            g_energy = np.sum(g)
            r_energy = np.sum(r)
            total_energy = b_energy + g_energy + r_energy
            hist = [b_energy / total_energy, g_energy / total_energy, r_energy /
total_energy]
            bins = np.percentile(hist, [33, 66])
            quantized = np.digitize(hist, bins)
            hists.extend(list(quantized[:3]))
        return np.array(hists[:12], dtype=int)
```

2.2 哈希函数：`lsh_hash(p, C, index_set)`

- **思想**：把原始 d 维特征 `p` 看作分段拼接后的位序（d' = d * C），对 `index_set` 内每个索引 `t`，计算对应的位是否为 1（依据 `p` 在其所属段的位置）。返回一个二进制元组 `g` 作为哈希键。
- **参数说明**：
  - `p`：输入特征（整数向量），长度 d。
  - `C`：每个原始维度展开的复制数（在代码中 `C=3`）。
  - `index_set`：选取的位索引集合（长度 = m）。

```
    def lsh_hash(p, C, index_set):
        g = []
        for t in index_set:
            i = (t - 1) // C
            v_t = 1 if t <= i * C + p[i] else 0
            g.append(v_t)
        return tuple(g)
```

2.3 建表：`build_lsh_table(data, C=3, L=4, m=6, seed=0)`

- **功能**：构建 `L` 个哈希表，每个哈希表使用不同的 `index_set`（随机选取 `m` 个索引），并把所有数据条目索引存入对应哈希桶。为了保证实验可重复性，函数接受 `seed` 参数（默认 `seed=0`），保持 `seed` 固定以便复现实验结果和对比不同 `(L,m)` 配置的影响
- **返回**：`hash_tables`（长度 L 的 list，每项为 dict mapping key->list(indices)）和 `index_sets`（对应的索引集合列表）。
- **参数影响**：
  - `C` 控制投影位的展开（影响哈希位的语义）。
  - `L`（哈希表数）越大，召回率越高但内存与候选数量上升。
  - `m`（每表的哈希位数）越大，哈希越精确（候选变少），但召回可能下降。

  通过改变L，m达到尝试不同投影集合的搜索的效果的目的。

```
    def build_lsh_table(data, C=3, L=4, m=6, seed=0):
        np.random.seed(seed)
        d, d_prime = data.shape[1], data.shape[1] * C
        hash_tables, index_sets = [], []
        for _ in range(L):
            idx = np.random.choice(np.arange(1, d_prime + 1), size=m,
replace=False)
            index_sets.append(sorted(idx))
            table = defaultdict(list)
            for idx_data, p in enumerate(data):
                key = lsh_hash(p, C, idx)
                table[key].append(idx_data)
            hash_tables.append(table)
        return hash_tables, index_sets
```

2.4 查询与检索

- `query_lsh(query, hash_tables, index_sets, C)`：对每个哈希表计算键，然后合并所有表中对应桶的候选项集合。

```
    def query_lsh(query, hash_tables, index_sets, C):
        candidates = set()
        for table, idx_set in zip(hash_tables, index_sets):
            key = lsh_hash(query, C, idx_set)
            candidates.update(table.get(key, []))
        return list(candidates)
```

- `brute_force_search(query, data)`：在所有数据上计算 L1 距离，返回全局最小的索引。

```
def brute_force_search(query, data):
    dists = [l1_distance(query, x) for x in data]
    return np.argmin(dists)
```

- `lsh_search(query, data, hash_tables, index_sets, C)`：获取候选后在候选集合上计算 L1 距离并返回最小距离对应的索引；若候选为空，返回 `None`。

```
def lsh_search(query, data, hash_tables, index_sets, C):
    candidates = query_lsh(query, hash_tables, index_sets, C)
    if not candidates:
        return None
    dists = [l1_distance(query, data[i]) for i in candidates]
    return candidates[np.argmin(dists)]
```

2.5 可视化：`visualize_results(query_img_path, nn_img_path, lsh_img_path=None, save_path=None)`

- 将目标图、暴力 NN 结果、LSH 结果并列展示并保存为 `result_compare.png`。若 LSH 未返回结果，会在第三幅子图显示文字提示。

3. 实验结果及分析

3.1 实验结果

建立哈希表时

- 固定参数m=6，改变参数L，即保持表中hash长度，增加建表的数量，分别为4、8、16时运行结果如下：

```
=== 检索结果 ===
目标图像: target.jpg
暴力 NN 检索结果: 12.jpg
LSH 检索结果: 12.jpg
=== 时间对比 ===
暴力 NN 搜索时间: 0.21251996 ms
LSH 搜索时间: 0.17085125 ms
可视化结果已保存: result_compare.png
```

```
=== 检索结果 ===
目标图像: target.jpg
暴力 NN 检索结果: 12.jpg
LSH 检索结果: 12.jpg
=== 时间对比 ===
暴力 NN 搜索时间: 0.21688139 ms
LSH 搜索时间: 0.18816800 ms
可视化结果已保存: result_compare.png
```

```
=== 检索结果 ===
目标图像: target.jpg
暴力 NN 检索结果: 12.jpg
LSH 检索结果: 12.jpg
=== 时间对比 ===
暴力 NN 搜索时间: 0.21903248 ms
LSH 搜索时间: 0.25807633 ms
可视化结果已保存: result_compare.png
```

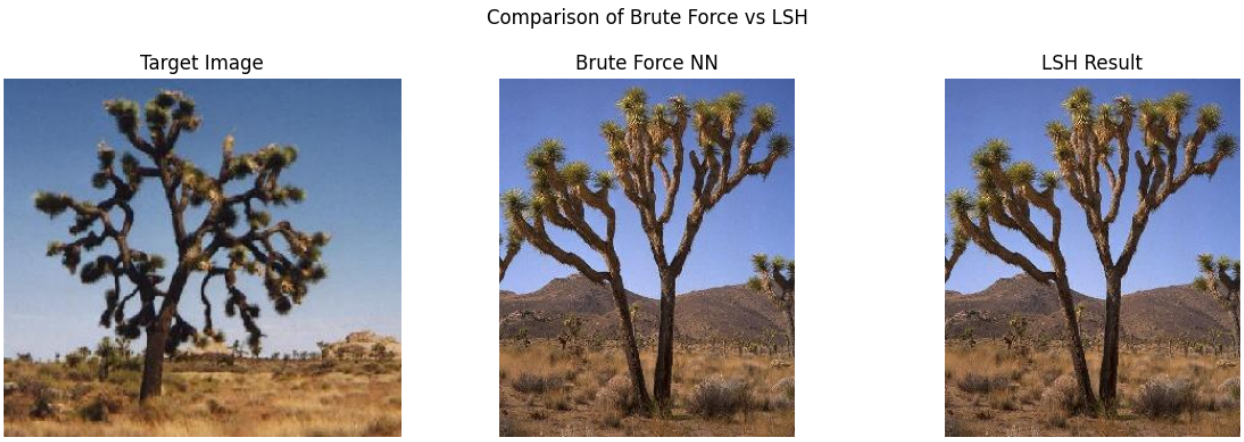
- 固定参数L=8，改变参数m，即保持建表的数量，增加表中hash长度，分别为3、 6、 9时运行结果如下：

```
=== 检索结果 ===
目标图像: target.jpg
暴力 NN 检索结果: 12.jpg
LSH 检索结果: 12.jpg
=== 时间对比 ===
暴力 NN 搜索时间: 0.21459913 ms
LSH 搜索时间: 0.23474982 ms
可视化结果已保存: result_compare.png
```

```
=== 检索结果 ===
目标图像: target.jpg
暴力 NN 检索结果: 12.jpg
LSH 检索结果: 12.jpg
=== 时间对比 ===
暴力 NN 搜索时间: 0.22217469 ms
LSH 搜索时间: 0.18921471 ms
可视化结果已保存: result_compare.png
```

```
=== 检索结果 ===
目标图像: target.jpg
暴力 NN 检索结果: 12.jpg
LSH 检索结果: 45.jpg
=== 时间对比 ===
暴力 NN 搜索时间: 0.23607388 ms
LSH 搜索时间: 0.04325192 ms
可视化结果已保存: result_compare.png
```

- 参数合适时运行所得图片如下：



3.2 结果分析

3.2.1 改变投影集合对LSH的影响

- `L`：哈希表（hash tables）的数量
  - L 大 → 更稳但查询慢
  - L 小 → 速度快但容易漏掉匹配
- `m`：每个表的哈希键长度（即随机采样多少个索引维度）
  - m 大 → 更严格的匹配 → 候选少 → 搜索更快但可能漏匹配，例如L=8，m=9时出现了搜索结果错误的情况。
  - m 小 → 更宽松的匹配 → 候选多 → 搜索更慢但不容易漏

调整恰当的参数值，两者平衡决定了LSH的性能。

3.2.2 运行时间分析

在这次实验中，我使用了`time`模块来计算查询时间，但是由于查询时间较短，我通过运行多次查询来计算平均查询时间，以减小误差。最终得到的结果如下：

- 在运行10000次查询后
  - LSH算法（L=8，m=6）的平均查询时间为： 0.18330920 ms
  - NN算法的平均查询时间为：0.20548279 ms

如果数据库中的图像数量较多，LSH算法的优势可能会更加明显。在小规模数据库中由于哈希表构建和索引开销，LSH 与暴力搜索的时间差异不明显。然而，随着数据库规模增大，暴力 NN 的查询时间随样本数线性增长，而 LSH 的查询时间基本保持稳定。这种加速源于 LSH 通过随机投影将相似样本映射到相同桶，从而避免了对所有样本计算距离。

4. 思考题

1. 本练习中使用了颜色直方图特征信息，检索效果符合你的预期吗？检索出的图像与输入图像的相似性体现在哪里？

    - 检索效果符合预期
    - 相似性体现在颜色直方图的相似性，即颜色直方图的能量分布相似，从视觉上看就是目标图像与匹配图象个在四块分区上的色调分别相似

2. 能否设计其他的特征？

    - 可以使用其他特征：如梯度直方图、SIFT特征等。

5. 附录

```
import os
import cv2
import numpy as np
from collections import defaultdict
import matplotlib.pyplot as plt
from time import time

def compute_color_histogram(image):
    image = cv2.resize(image, (200, 200))
    h, w = image.shape[:2]
    mid_h = h // 2
    mid_w = w // 2
    regions = [image[0:mid_h, 0:mid_w], image[0:mid_h, mid_w:w],
               image[mid_h:h, 0:mid_w], image[mid_h:h, mid_w:w]]
    hists = []
    for region in regions:
        b, g, r = cv2.split(region)
        b_energy = np.sum(b)
        g_energy = np.sum(g)
        r_energy = np.sum(r)
        total_energy = b_energy + g_energy + r_energy
        hist = [b_energy / total_energy, g_energy / total_energy, r_energy /
total_energy]
        bins = np.percentile(hist, [33, 66])
        quantized = np.digitize(hist, bins)
        hists.extend(list(quantized[:3]))
    return np.array(hists[:12], dtype=int)
```

```python
def lsh_hash(p, C, index_set):
    g = []
    for t in index_set:
        i = (t - 1) // C
        v_t = 1 if t <= i * C + p[i] else 0
        g.append(v_t)
    return tuple(g)

def build_lsh_table(data, C=3, L=4, m=6, seed=0):
    np.random.seed(seed)
    d, d_prime = data.shape[1], data.shape[1]*C
    hash_tables, index_sets = [], []
    for _ in range(L):
        idx = np.random.choice(np.arange(1, d_prime+1), size=m, replace=False)
        index_sets.append(sorted(idx))
        table = defaultdict(list)
        for idx_data, p in enumerate(data):
            key = lsh_hash(p, C, idx)
            table[key].append(idx_data)
        hash_tables.append(table)
    return hash_tables, index_sets

def query_lsh(query, hash_tables, index_sets, C):
    candidates = set()
    for table, idx_set in zip(hash_tables, index_sets):
        key = lsh_hash(query, C, idx_set)
        candidates.update(table.get(key, []))
    return list(candidates)

def l1_distance(a, b):
    return np.sum(np.abs(a - b))

def brute_force_search(query, data):
    dists = [l1_distance(query, x) for x in data]
    return np.argmin(dists)

def lsh_search(query, data, hash_tables, index_sets, C):
    candidates = query_lsh(query, hash_tables, index_sets, C)
    if not candidates:
        return None
    dists = [l1_distance(query, data[i]) for i in candidates]
    return candidates[np.argmin(dists)]

def visualize_results(query_img_path, nn_img_path, lsh_img_path=None, save_path=None):
    # 读取图像
    query_img = cv2.imread(query_img_path)
    nn_img = cv2.imread(nn_img_path)
    lsh_img = cv2.imread(lsh_img_path) if lsh_img_path else None

    # 转 RGB
```

```
    query_img = cv2.cvtColor(query_img, cv2.COLOR_BGR2RGB)
    nn_img = cv2.cvtColor(nn_img, cv2.COLOR_BGR2RGB)
    if lsh_img is not None:
        lsh_img = cv2.cvtColor(lsh_img, cv2.COLOR_BGR2RGB)

    # 画图
    plt.figure(figsize=(12, 4))
    plt.subplot(1, 3, 1)
    plt.imshow(query_img)
    plt.title('Target Image')
    plt.axis('off')

    plt.subplot(1, 3, 2)
    plt.imshow(nn_img)
    plt.title('Brute Force NN')
    plt.axis('off')

    plt.subplot(1, 3, 3)
    if lsh_img is not None:
        plt.imshow(lsh_img)
        plt.title('LSH Result')
        plt.axis('off')
    else:
        plt.text(0.5, 0.5, 'No LSH Match', ha='center', va='center', fontsize=12)
        plt.axis('off')

    plt.suptitle('Comparison of Brute Force vs LSH')
    plt.tight_layout()

    if save_path:
        plt.savefig(save_path)
        print(f"可视化结果已保存: {save_path}")
    else:
        plt.show()

if __name__ == "__main__":
    folder = "img/dataset"
    image_paths = [
        os.path.join(folder, f)
        for f in os.listdir(folder)
        if f.lower().endswith(('.jpg', '.jpeg', '.png', '.jfif'))
    ]
    image_paths.sort()

    features = []
    for path in image_paths:
        img = cv2.imread(path)
        if img is None:
            print(f"无法读取图片: {path}")
            continue
        feat = compute_color_histogram(img)
```

```
        features.append(feat)
    data = np.vstack(features)

    # 读取目标图片
    target_path = os.path.join(os.getcwd(), "img/target.jpg")
    if not os.path.exists(target_path):
        raise FileNotFoundError(f"未找到目标图片: {target_path}")
    query_img = cv2.imread(target_path)
    query_feat = compute_color_histogram(query_img)

    hash_tables, index_sets = build_lsh_table(data, C=3, L=8, m=6)
    nn_time = 0
    lsh_time = 0
    # 检索
    for i in range(0, 10000):
        # 暴力 NN
        start = time()
        nn_result_idx = brute_force_search(query_feat, data)
        nn_time += time() - start

        # LSH
        start = time()
        lsh_result_idx = lsh_search(query_feat, data, hash_tables, index_sets, C=3)
        lsh_time += time() - start

    # 路径
    lsh_img_path = image_paths[lsh_result_idx] if lsh_result_idx is not None else None
    nn_img_path = image_paths[nn_result_idx]

    print("=== 检索结果 ===")
    print("目标图像:", os.path.basename(target_path))
    print("暴力 NN 检索结果:", os.path.basename(nn_img_path))
    if lsh_img_path:
        print("LSH 检索结果:", os.path.basename(lsh_img_path))
    else:
        print("LSH 未找到候选")
    print("=== 时间对比 ===")
    print(f"暴力 NN 搜索时间: {nn_time / 10:.8f} ms")
    print(f"LSH 搜索时间: {lsh_time / 10:.8f} ms")

    # 显示结果
    save_path = "result_compare.png"
    visualize_results(target_path, nn_img_path, lsh_img_path, save_pat
```