《媒体数据管理》上机实验报告

姓名：郑力南

第一次实验　算术编码

算法描述：

算术编码是一种无损数据压缩方法，也是一种熵编码的方法。和其它熵编码方法不同的地方在于，其他的熵编码方法通常是把输入的消息分割为符号，然后对每个符号进行编码，而算术编码是直接把整个输入的消息编码为一个数，一个满足（0.0 ≤ n < 1.0）的小数 n。在给定符号集和符号概率的情况下，算术编码可以给出接近最优的编码结果。使用算术编码的压缩算法通常先要对输入符号的概率进行估计，然后再编码。这个估计越准，编码结果就越接近最优的结果。

编码程序

输入：信源分布，原字符串；
输出：编码后的字符串；

算法流程：

1. 设定临时区间的起始位置初始化为 0，临时区间的长度初始化为 1；
2. 考察每一个字符 c_i，为第 i 个字符，其在信源分布中的概率为 p_i；
3. 则临时区间的起始位置加 $p_1 + p_2 + \ldots + p_i$，临时区间的长度乘 p_i；
4. 反复执行步骤 2、3 直到所有字符都已经考察过；
5. 找到临时区间的起始位置和临时区间的结尾位置之间的一个数，并满足其二进制位数最小，将二进制小数部分转化为字符串输出。

解码程序

输入：信源分布，编码后的字符串，原字符串长度；
输出：原字符串；

算法流程：

1. 设定临时区间的起始位置初始化为 0，临时区间的长度初始化为 1；
2. 将输入字符串转为二进制小数，设为 n；
3. 考察信源分布中的每一个字符，第 i 个字符 c_i，
4. 目标区间起始位置设为临时区间起始位置+临时区间长度×$(p_1 + p_2 + \ldots + p_i)$；
5. 目标区间结尾位置设为目标区间起始位置+临时区间长度×其在信源分布中的概率 p_i；
6. 如果 n 在目标区间内，则编码后的字符串加上字符 c_i，并将临时区间起始位置设为

目标区间起始位置，将临时区间长度设为目标区间结尾位置-目标区间起始位置，再从头开始遍历信源分布中的每一个字符；

7. 反复执行步骤 3、4、5、6 直到编码后的字符串的长度达到指定长度；
8. 输出编码后的字符串。

核心源程序 [github]

编码程序

```
string encode(string str_tmp, int len_c){
    string s;
    long double tmp_start = 0;
    long double tmp_len = 1;

    for (int i = 0; i < str_tmp.size(); i += len_c){
        s = str_tmp.substr(i, len_c);
        // 更新区间
        tmp_start += tmp_len * m[s].first;
        tmp_len = tmp_len * (m[s].second - m[s].first);
    }

    // bi函数将数字转化为二进制字符串
    string a = bi(tmp_start);
    cout << "start of last interval: 0." << a << endl;
    string b = bi(tmp_start + tmp_len);
    cout << "end of last interval:   0." << b << endl;

    // solve函数找到两个二进制数之间位数最少的数
    return solve(a, b);

}
```

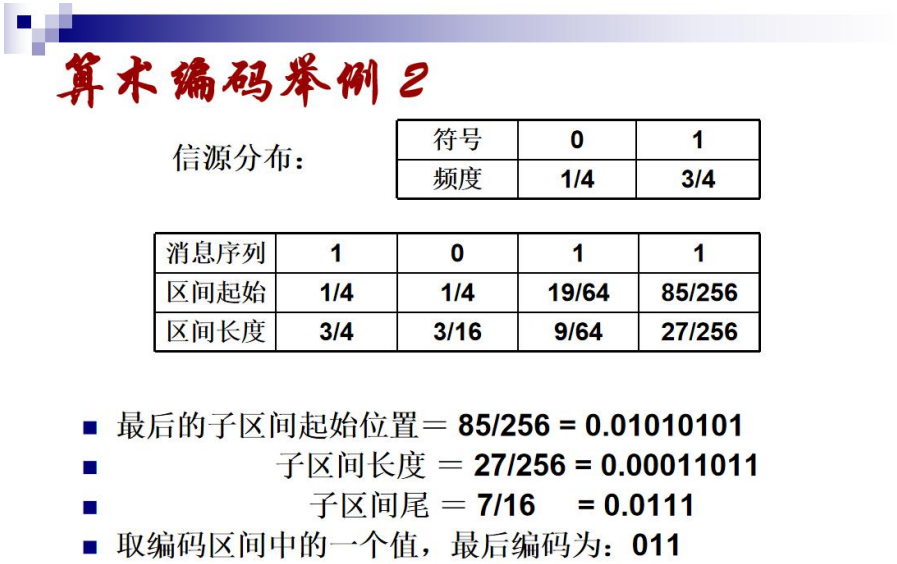
解码程序

```
string decode(string msg, int len){
    long double tmp_start = 0;
    long double tmp_len = 1;
    long double num = convert(msg);

    string s = "";
    while(1){
        if (s.size() == len){
            break;
        }

        // 遍历信源分布
        auto it = m.begin();
        for (; it != m.end(); ++it){
            auto range = it->second;
```

```
            // 更新目标区间（待考察）
            long double start = tmp_start + tmp_len * range.first;
            long double end = start + tmp_len * (range.second - range.first);

            // num在该区间内，意味着该区间对应的字符应是解码后的字符
            if (num >= start && num < end){
                s += it->first;
                tmp_start = start;
                tmp_len = end - start;
                break;
            }
        }
    }
    return s;
}
```

测试数据：

测试 1 符号单位长度为 1（课件 p21 示例）



输入命令

```
main -p input1.txt -s 1011 -e
```

main 为可执行文件；-p 表示输入信源分布，即 input1.txt；-s 表示输入字符串，即 1011；-e 表示运行编码

输出

```
D:\媒体数据管理作业\hw1>main -p input1.txt -s 1011 -e
raw str: 1011
start of last interval: 0.01010101
end of last interval:   0.0111
encode:  011
```

由输出可知区间首为 0.01010101，区间尾为 0.0111，编码后的字符串为 011

输入命令

```
main -p input1.txt -s 011 -d 4
```

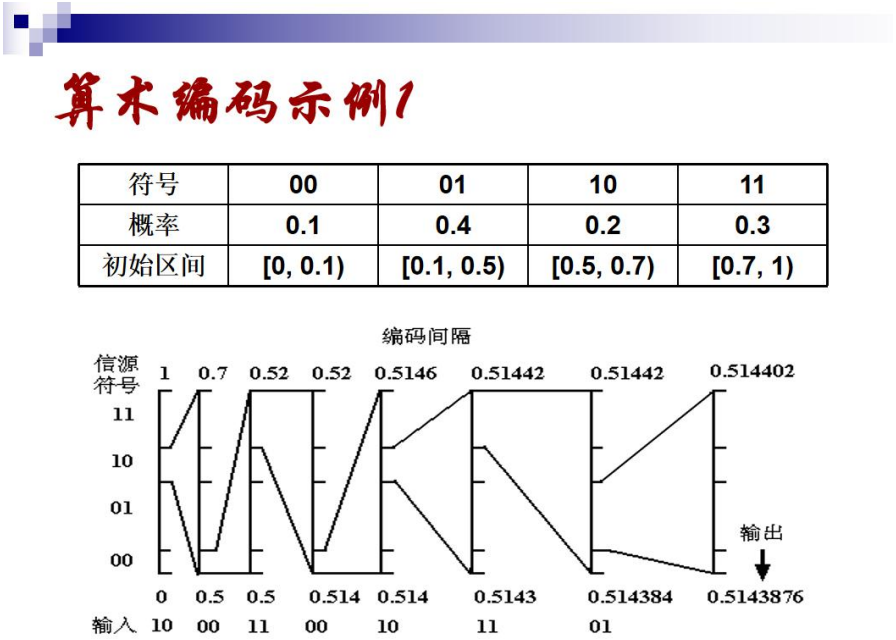
main 为可执行文件；-p 表示输入信源分布，即 input1.txt；-s 表示输入字符串，即 011；-d 表示运行解码，目标长度为 4

输出

```
D:\媒体数据管理作业\hw1>main -p input1.txt -s 011 -d 4
raw str: 011
decode:  1011
```

由输出可知将 011 解码为 1011

测试 2 符号单位长度为 2 （课件 p20 示例）



符号	00	01	10	11
概率	0.1	0.4	0.2	0.3
初始区间	[0, 0.1)	[0.1, 0.5)	[0.5, 0.7)	[0.7, 1)

输入命令

```
main -p input2.txt -s 10001100101101 -e
```

main 为可执行文件；-p 表示输入信源分布，即 input2.txt；-s 表示输入字符串，即 10001100101101；-e 表示运行编码

输出

```
D:\媒体数据管理作业\hw1>main -p input2.txt -s 10001100101101 -e
raw str: 10001100101101
start of last interval: 0.10000011101011101110011111011111011101111100101000111011111111
end of last interval:   0.10000011101011111101100101110110111111111001110101101111100010001
encode:  1000001110101111

D:\媒体数据管理作业\hw1>
```

由输出可知编码后的字符串为 10001100101101

输入命令

```
main -p input2.txt -s 1000001110101111 -d 14
```

main 为可执行文件；-p 表示输入信源分布，即 input1.txt；-s 表示输入字符串，即 1000001110101111；-d 表示运行解码，目标长度为 14

输出

```
D:\媒体数据管理作业\hw1>main -p input2.txt -s 1000001110101111 -d 14
raw str: 1000001110101111
decode:  10001100101101
```

由输出可知将 1000001110101111 解码为 10001100101101

第二次实验　K-L 变换和矢量量化

K-L 变换

主成分分析（PCA、K-L 变换）是一种统计分析、简化数据集的方法。它利用正交变换来对一系列可能相关的变量的观测值进行线性变换，从而投影为一系列线性不相关变量的值，这些不相关变量称为主成分。具体地，主成分可以看做一个线性方程，其包含一系列线性系数来指示投影方向。PCA 对原始数据的正则化或预处理敏感（相对缩放）。

算法描述：

输入：高维数据
输出：低维数据

算法流程：

1. 计算协方差矩阵；
2. 计算协方差矩阵的特征值和特征向量；
3. 对特征值从小到大排序；
4. 选取前 k 大的特征值对应的特征向量，组合成为一个矩阵；
5. 将原高维数据与这个矩阵相乘得到低维数据。

核心源程序 [github]：

```
# 计算协方差矩阵
covMat = np.cov(data, rowvar=False)
# 计算特征值和特征向量
eigVals, eigVects = np.linalg.eig(np.mat(covMat))
```

```
# 对特征值从小到大排序
eigValIndice = np.argsort(eigVals)
top = 3
# 最大top个特征值的下标
n_eigValIndice = eigValIndice[-1: -(top+1): -1]
n_eigVect = eigVects[:, n_eigValIndice]
# 得到低维特征空间的数据
lowDDataMat = np.matrix(data) * n_eigVect
```

测试数据：

输入：ColorHistogram 数据集（68040×32）

```
df = pd.read_csv("ColorHistogram.asc", header=None, sep=' ')
data = np.array(df)[:, 1:]
data.shape
```

```
(68040, 32)
```

```
df.iloc[:, 1:]
```

	1	2	3	4	5	6	7	8	9	10	.
0	0.002188	0.000000	0.000000	0.620521	0.010313	0.007083	0.043021	0.310729	0.000729	0.000000	.
1	0.002917	0.315417	0.188854	0.004440	0.000001	0.000001	0.000004	0.000032	0.000000	0.000000	.
2	0.000313	0.009825	0.008978	0.663125	0.002083	0.003542	0.006250	0.009688	0.000000	0.000000	.
3	0.111667	0.123855	0.078230	0.085486	0.015104	0.026563	0.015938	0.004064	0.003958	0.012708	.
4	0.329803	0.522930	0.034487	0.011571	0.005835	0.000315	0.000314	0.003542	0.000834	0.000418	.
5	0.153368	0.242450	0.246149	0.102189	0.007709	0.007083	0.005104	0.017084	0.000417	0.001042	.

输出：低维数据（68040×3）

```
# 低维特征空间的数据
lowDDataMat = np.matrix(data) * n_eigVect
lowDDataMat.shape
```

```
(68040, 3)
```

```
lowDDataMat
```

```
matrix([[-0.30735124, -0.08563712,  0.11602428],
        [-0.21053805, -0.10110636, -0.13301947],
        [-0.33944586, -0.07441226,  0.0433448 ],
        ...,
        [-0.2644657 , -0.16436316, -0.17503818],
        [-0.03750485, -0.07618829,  0.04120986],
        [ 0.01919037, -0.05732744,  0.24324975]])
```

矢量量化

算法描述：

矢量量化是一个在信号处理中的一个量化法，其为借由样本向量的训练来估算密度几率函数，并借由此密度函数推估最有效的量化方案。此技术原用于数据压缩，透过分割大数量的数据点，让每个小群集都有相同的数据点，而这些小群集的所有数据就由其正中央的点作

为代表，这点与 k-平均算法以及其他群集分析的特性相当。 矢量量化所使用的密度分布法的优势在于，此种压缩法对于高几率出现（密集）的数据误差小，而对低几率（稀疏）的数据误差大，故特别适用于大量且高维度的向量破坏性数据压缩。

输入：原图片
输出：压缩后的图片

算法流程：

1. 设定代表点个数，随机分布代表点；
2. 对于每个点计算到代表点的距离，判断属于哪个代表点；
3. 重新计算出新的代表点；
4. 重复步骤 2、若干次，知道代表点位置不再更新为止；
5. 将代表点所代表的全部点设为和代表点一样的数值（像素值）；
6. 输出处理后的图片。

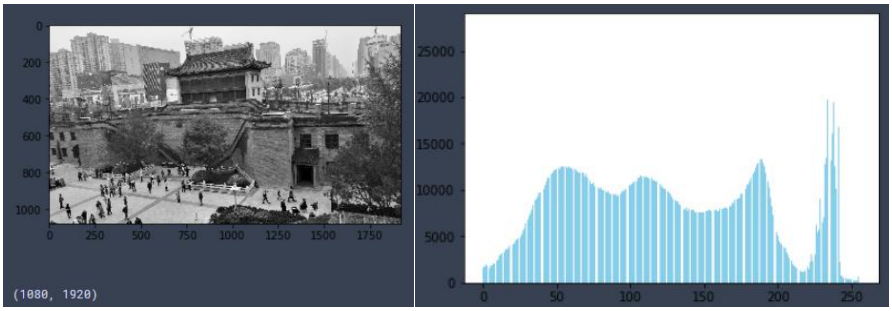
核心源程序 [github]：

```
K_nums = 4  # 代表点的个数

kmeans = KMeans(n_clusters=K_nums, n_init=10)
img_data = image.reshape((-1,1))
kmeans.fit(img_data)
centroids = kmeans.cluster_centers_.squeeze() # 代表点
labels = kmeans.labels_

# 将代表点所代表的全部点设为和代表点一样的数值
img2 = np.choose(labels, centroids).reshape(image.shape)
```

测试数据：

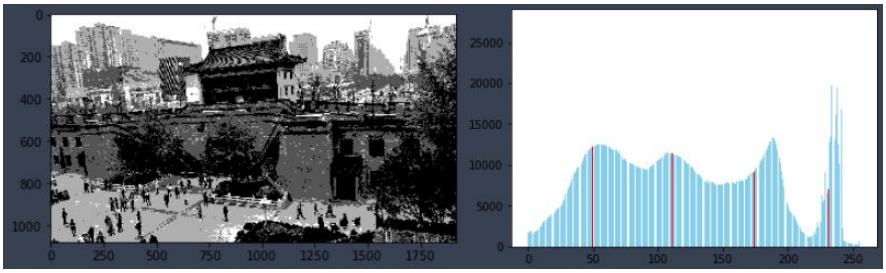
输入：原图（为方便可视化代表点，使用灰度图做例子）；

原图中颜色分布如右图所示，横坐标为灰度值，纵坐标为出现次数

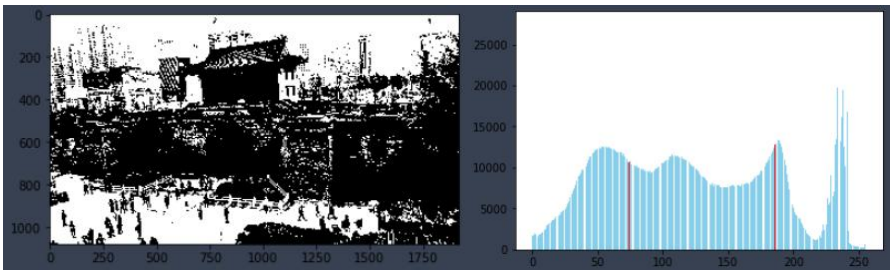


输出：

处理后的颜色分布如右图所示（k=4），横坐标为灰度值，纵坐标为出现次数，红色为代表点（173， 50， 110， 230）



处理后的颜色分布如右图所示（k=2），横坐标为灰度值，纵坐标为出现次数，红色为代表点（186， 74）



第三次实验　LSH 索引

算法描述：

输入：ColorHistogram 数据集（68040×32），1000 个查询
输出：每个查询的 10 个最近邻

算法流程：

1. 定义基础哈希函数 $h(o)=\left\lfloor \frac{\|a\cdot o+b\|}{W} \right\rfloor$；
2. 定义最大模长、线段长；
3. 定义哈希函数族 $H(o)=\{h_1(o),h_2(o),\ldots,h_3(o)\}$；
4. 对于任意两个输入，经哈希函数族运算后，发生哈希碰撞的次数越多则两个输入越相似。

核心源程序 **[github]**：

```
def h_k(o, w):
    # 基础哈希函数
    a = np.random.rand(data.shape[1])
    b = w * np.random.rand(data.shape[0] *
                           data.shape[1]).reshape(data.shape)
    return np.linalg.norm(a * o + b, axis=1) // w

# 最大模长
max_len = np.linalg.norm([np.max(data[:, _])
                          for _ in range(data.shape[1])])

# 最多n_bucket个桶
n_bucket = 256

# 根据n_bucket确定段长
W = max_len / n_bucket

def H(o, k_h):
    # 哈希函数族
    return np.array([h_k(o, W) for _ in range(k_h)]).T

k_h = 256       # k_h越大，每组中使用的哈希函数越多，结果越逼近p范数近邻的结果
data_hash = H(data, k_h)

prediction = np.zeros((n_qs, k), dtype=np.int64)
for i in range(n_qs):
    # 计算哈希碰撞的次数
    cnt = np.sum(data_hash == data_hash[i], axis=1)
    index = np.argsort(cnt)
    prediction[i] = index[-1*k :][::-1]
```

测试数据：

测试数据 1（k=10）

输入：ColorHistogram 数据集（68040×32），1000 个查询

输出：每个查询的 10 个最近邻

性能：

数据总量为 68040，查询数据集前 1000 点，前 10 个最近邻

准确率：0.9998197236919459

召回率：0.3867

精确率：0.3867

F 值：0.38669999999999993

测试数据 2（k=100）

输入：ColorHistogram 数据集（68040×32），1000 个查询

输出：每个查询的 100 个最近邻

性能：

数据总量为 68040，查询数据集前 1000 点，前 100 个最近邻

准确率：0.998274985302763

召回率：0.41314999999999996

精确率：0.41314999999999996

F 值：0.4131499999999999

测试数据 3（k=1000）

输入：ColorHistogram 数据集（68040×32），1000 个查询

输出：每个查询的 1000 个最近邻

性能：

数据总量为 68040，查询数据集前 1000 点，前 1000 个最近邻

准确率：0.9844951205173427

召回率：0.472524

精确率：0.472524

F 值：0.472524

第四次实验　SIFT 特征的近邻搜索

算法描述：

尺度不变特征转换(SIFT)是一种机器视觉的算法用来侦测与描述影像中的局部性特征，它在空间尺度中寻找极值点，并提取出其位置、尺度、旋转不变数，此算法由 David Lowe 在 1999 年所发表，2004 年完善总结。

输入：图片

输出：一系列特征向量

算法流程（SIFT）：

1. 预滤波；
2. 建立高斯金字塔和高斯差分金字塔；
3. 确定局部极值点；
4. 做子像素插值精确定位极值点；
5. 过滤具有较大的主曲率的极值点（去除边缘效应）；
6. 建立方向直方图；
5. 生成关键点描述子。

算法流程（近邻搜索）：

1. 对每张图提取 SIFT 特征；
2. 训练字典（聚类）；
3. 将图片用特征袋（直方图）表示；
4. 使用 TF-IDF 加权；
5. 根据余弦相似度查找近邻。

核心源程序 [github]：

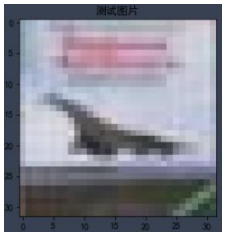
SIFT（get_feat_set 函数）具体实现的代码量过大，详见链接

近邻搜索

```
# 提取特征
desc_list = get_feat_set(s)
# 训练字典（聚类）
kmeans = clustering(desc_list)
# 图片直方图表示
train_feat = histogram(desc_list, kmeans)
# TF-IDF
tf = np.sum(train_feat, axis=0) / np.sum(train_feat)
idf = [np.log(len(train_feat) /
              (sum(train_feat[:, i] > 0) + 1)) \
                            for i in range(BINS)]
tfidf = tf * idf
train_feat = train_feat * tfidf

from sklearn.metrics.pairwise import cosine_similarity

def f(a, b):
    return cosine_similarity([a, b])[0][0]
k_ = 10
nbrs = NearestNeighbors(n_neighbors=k_+1,
                        algorithm='ball_tree').fit(train_feat)
distances, indices = nbrs.kneighbors(train_feat)

top_k = [int(s[int(i)].split('/')[-1][0])
         for i in indices[0][1:]]
```

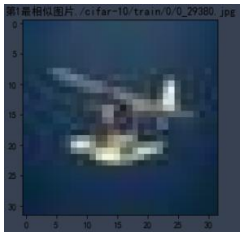
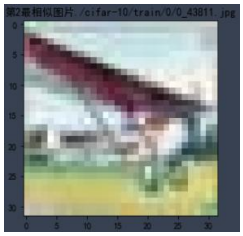
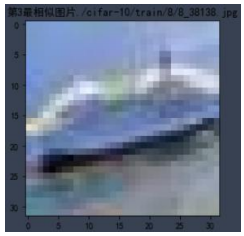
测试数据：

输入：



输出：

（以 3-最近邻为例，其中第 1、2 最相似图片是和测试数据同类别的，精确率为 66.7%）

第五次实验　k-d 树的近邻搜索

算法描述：

k-d 树是每个节点都为 k 维点的二叉树。所有非叶子节点可以视作用一个超平面把空间分割成两个半空间。节点左边的子树代表在超平面左边的点，节点右边的子树代表在超平面右边的点。选择超平面的方法如下：每个节点都与 k 维中垂直于超平面的那一维有关。因此，如果选择按照 x 轴划分，所有 x 值小于指定值的节点都会出现在左子树，所有 x 值大于指定值的节点都会出现在右子树。这样，超平面可以用该 x 值来确定，其法线为 x 轴的单位向量。

输入：大量数据点，查询
输出：每个查询的 k 个最近邻

算法流程（建树）：

1. 确定分割维度（交替）；
2. 按照选定维度的值排序，并获取中间索引；
3. 以中间为根递归建立 k-d 树。

算法流程（查找近邻）：

1. 确定分割维度（交替）；
2. 分割平面；
3. 检查目标点处于哪个区域，将两个区域标记为近邻区域（贪心）和父区域（回溯）；

4. 检查当前点到近邻点（若干）的距离是否更小，决定是否将该点归入近邻点；

5. 在近邻区域递归调用查找算法；

6. 如果在父区域内还有点，那么在另一半区域可能会存在近邻，则需要计算目标点到另一个区域的最近距离；

7. 如果第 6 步得到的距离比到近邻点（若干）的某个距离要小，那么父区域有望出现近邻点，在父区域递归调用查找算法；

核心源程序 [github]：

算法流程（建树）：

```python
def kdtree(point_list, depth: int = 0):
    """建树"""
    if not point_list:
        return None

    # 确定分割维度
    k = len(point_list[0])
    axis = depth % k

    # 按照选定维度的值排序，并获取中间索引
    point_list.sort(key=itemgetter(axis))
    median = len(point_list) // 2

    # 以中间为根递归建立kd树
    return Node(
        location=point_list[median],
        left_child=kdtree(point_list[:median], depth + 1),
        right_child=kdtree(point_list[median + 1:], depth + 1)
    )
```

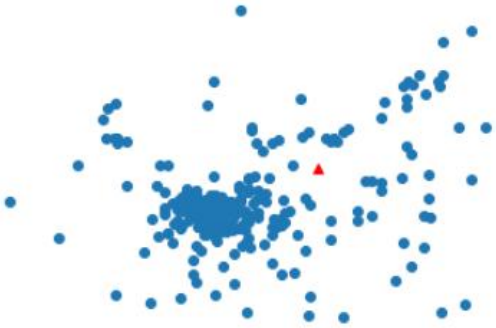
算法流程（查找近邻）：

```python
def search_kdtree(tree, target_point, result, hr=hr_max, depth=0, text=True):
    """查找近邻"""
    global search_steps
    if depth == 0:
        search_steps = 1
    else:
        search_steps += 1

    cur_node = tree.location
    left_branch = tree.left_child
    right_branch = tree.right_child
```

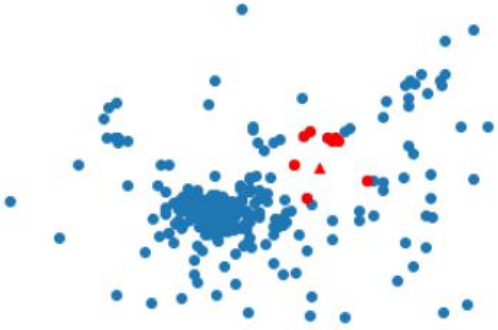
```
    nearer_kd = farther_kd = None
    nearer_hr = farther_hr = None

    # 选择分割维度 —— 0/1交替
    axis = depth % 2

    # 依据axis分割平面
    left_hr, right_hr = split_hyperplane(cur_node, hr, axis)

    # 检查目标点处于哪个区域
    if target_point[axis] <= cur_node[axis]:
        nearer_kd = left_branch
        farther_kd = right_branch
        nearer_hr = left_hr
        farther_hr = right_hr
    else:
        nearer_kd = right_branch
        farther_kd = left_branch
        nearer_hr = right_hr
        farther_hr = left_hr

    # 检查当前点到NNs的距离是否更小
    dist = result.compute_rel_distance(cur_node, target_point)
    new_result = result.update(dist, cur_node)

    # nearer_kd是最近的子树，nearer_hr是对应的区域，进一步搜索这个区域
    if nearer_kd:
        if text:
            print("@当前:", cur_node, hr,
                  "\n递归近邻:", nearer_kd.location, nearer_hr, '\n')
        search_kdtree(nearer_kd, target_point,
                      new_result, nearer_hr, depth+1, text)

    # 如果在另一半区域内还有点，
    # 那么在另一半区域可能会存在近邻，则需要计算目标点到另一个区域的最近距离
    if farther_kd:
        if text:
            print("@当前:", cur_node, hr,
                  "\n回溯另一半:", farther_kd.location, farther_hr, '\n')
        # pt是farther_hr区域上距离目标点最近的点（理想点）
        pt = compute_closest_point(target_point, farther_hr)

        # 检查
        dist = new_result.compute_rel_distance( pt, target_point)
        if new_result.can_contain(dist):
            # 如果目标点到区域上理想点的距离比暂存的某个距离要小
            # 那么这个区域有望出现近邻点，搜索这个区域
            new_result = search_kdtree(farther_kd, target_point,
                                       new_result, farther_hr, depth+1, text)

    return new_result
```

测试数据：

测试数据 1（BJ，k=10）
输入：BJ 数据集，查询点(116.8, 40.2)



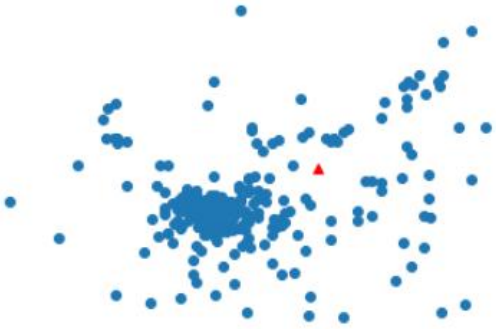
输出：查询点的 10 个最近邻



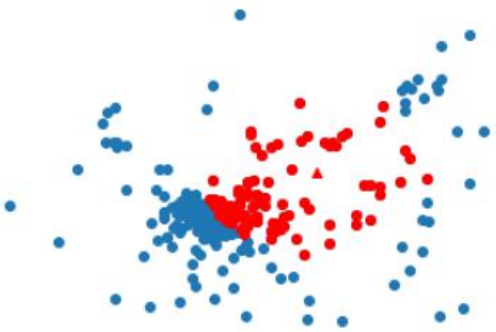
消耗时间：< 1ms

测试数据 2（BJ，k=100）

输入：BJ 数据集，查询点(116.8, 40.2)



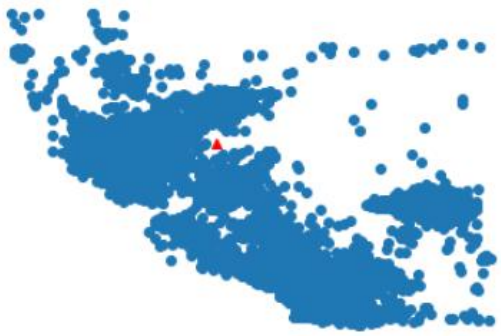
输出：查询点的 100 个最近邻



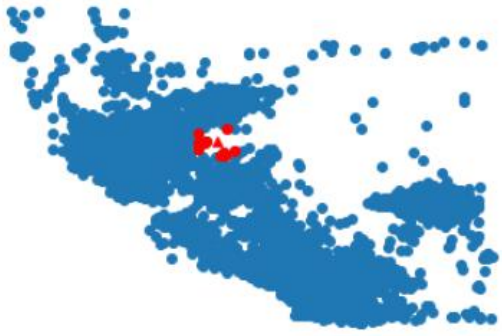
消耗时间：3ms

测试数据 3（CA，k=10）

输入：CA 数据集，查询点(-120, 38)



输出：查询点的 10 个最近邻



消耗时间：2ms

测试数据 4（CA，k=100）

输入：CA 数据集，查询点(-120, 38)

输出：查询点的 100 个最近邻

消耗时间：7ms

个人收获：

通过 5 次的实验，使我对数据的压缩、检索、匹配等技术有了一定的理解。作为大数据方向的学生，我觉得学习这门课还是十分必要的。5 次实验题目都是数据管理领域十分经典的算法，通过动手实践，不但锻炼了自己的编程能力，还对这些经典算法有了更深的理解，使我真切的体会到，学习算法不能仅仅停留在“看”，而是要实实在在地“写”，只有动手复现过算法，才能更深刻地理解算法背后的思想。

由于班上同学们编程能力水平不一，有些算法的还是比较复杂的，比如 SITF（不调库的话），如果有些同学是想自己实现算法，但是总是出现各种各样的问题，~~在 ddl 的催促下~~只好以一个比较低的质量完成。所以不如鼓励同学们在 ddl 之后将代码共享到公共仓库，方便同学们之间讨论学习，如果有的同学对之前的算法有不理解的地方，可以学习一下其他同学是如何解决的，虽然已经过了 ddl，但是分数不是最重要的，最重要的是从这门课中学到知识。