《媒体数据管理》上机实验报告

姓名：xxxxxxxx
学号：xxxxxxxx

第 2 次实验 K-L 变换和矢量量化

K-L 变换

算法描述：
输入：m 条 n 维数据，目标降维维度 k
输出：m 条 k 维数据，降维后的数据能够尽可能好的表征原来数据的特性

1. 将原始数据按列组成 n 行 m 列矩阵 X
2. 将 X 的每一行进行 0 均值化，减去这一行的均值
3. 求出协方差矩阵

$$C=\frac{1}{m}XX^T$$

4. 求出协方差矩阵的特征值及对应的特征向量
5. 将特征向量按照对应特征值大小从大到小按行排列成矩阵，取前 k 行组成矩阵 P，P 为正交的低维线性子空间
6. Y = PX 即为降到 k 维之后的数据，Y 表示 X 在 P 下的坐标

核心源程序：

```
def klTransform(array, k):
    """
    对给定的输入数据 array，降维至 k 维
    :param array: 输入数据
    :param k: 目标维度
    :return: 降维之后的矩阵
    """
    #  求取均值
    m = np.mean(array, axis=1)  # 对每一行求取均值
    # 去除均值，对每一行进行零均值化
    for i in range(array.shape[0]):
        for j in range(array.shape[1]):
            array[i][j] -= m[i]
    #  求取协方差矩阵
    C = np.cov(array, bias=True)
    #  求取协方差矩阵的特征值、特征向量
    lamda, phi = np.linalg.eig(C)
    # numpy 计算出的特征向量为复数形式，提取实部
    phi = np.real(phi)
    #  对特征值从大到小排序
```

```
    sort_lamda = -np.sort(-lamda)
    #  从大到小排序，输出特征值序列的索引
    sort_index = np.argsort(-lamda)
    #  从特征向量中，找到对应特征值最大的前 k 个特征向量
    #  组成变换矩阵
    eigenVector = phi[:, sort_index[:k]]
    eigenValue = lamda[sort_index[:k]]

    #  将变换矩阵与输入矩阵进行点积，将原始输入数据投影到 k 维度空间内，完成降维
    Y = np.dot(eigenVector.T, array)
    # 返回变换矩阵与降维后的数据
    return (eigenVector, Y)
```

测试数据（输入、输出）：

输入：corel 数据集（）；

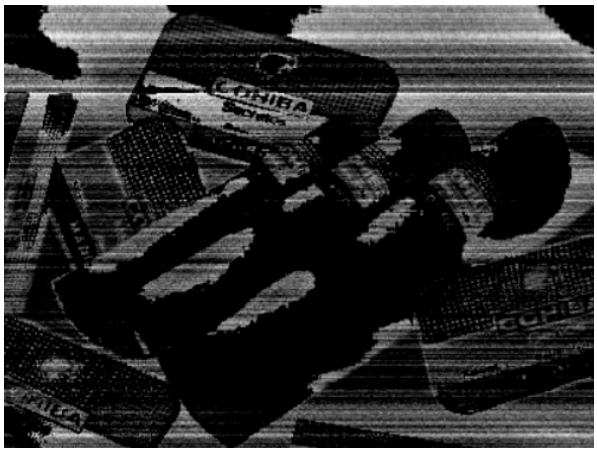
在实验中，输入数据为图像形式，原始图像为：

输出：

```
读取图像，并转换为灰度图 ...
原始输入矩阵大小为：(300, 400)
开始进行 K-L 变换 ...
变换后矩阵大小为：(100, 400)
变换后的矩阵为：
[-1091.433786305546, -1116.5510649727657, -1198.7705942272466, -1194.1126579914874, -1276.345075734262, -1314.20105480
[-1281.9825810666698, -1208.7669926872668, -1221.8832348117216, -1247.7305919341936, -1288.016922400759, -1279.3541981
[-492.9523399304907, -379.4114209274043, -332.3971781843806, -275.37372219466226, -238.74282116754625, -212.3982159745
[-350.4467817704677, -315.7939011496779, -332.4043546395803, -316.29310598555145, -352.8503719571272, -344.18872662506
[65.05544494645886, 108.44484572778981, 43.28790507384605, 8.941705692109508, -44.98623178074297, -60.877741870817374,
[354.2712217130813, 395.5855049592624, 389.5458482247532, 330.5776335774972, 382.9877241373573, 357.3344487464362, 350
[1071.491466315053, 1042.397848527877, 1020.1177621288563, 1007.4509766956171, 1000.006782742198, 936.0579076484141, 9
[-64.20687403754914, -72.10211226514986, -41.54577745679103, -14.45788820651768, -33.69637088667565, -23.0266823274153
[-354.4691417676889, -366.5043887016232, -343.11091852782454, -236.75653891944765, -168.95862962418573, -144.583649764
[-439.52972594107325, -447.19544625813995, -523.0585116315231, -521.7936294678609, -430.18647519397473, -380.2094951119
[603.3454894029875, 649.1343866958368, 568.5982480162886, 513.5141549850166, 504.6800391386521, 473.6662964759878, 470
[7.087569842196785, -0.1968915481867839, -3.2314040115055604, -64.84568782332099, -56.596610602730834, -36.29677201594
[-62.97239249696008, 20.72452063679313, -111.3115724492024, -76.78412529012064, -63.54226542238317, 9.021489805043814,
[74.34316465891673, -7.9836984767108845, -118.21920958870913, -177.58641621512132, -154.9128336134591, -228.1749245141
[-244.17142315964676, -308.3599553670148, -257.1728641567055, -277.9779804765676, -214.55468818882355, -259.2954510378
[-11.937479572685689, 51.01892989232308, 97.36479586597662, 122.62937967451596, 130.77785950459045, 111.82465641404306
[-281.6848799613583, -345.179136509726, -310.07763392172654, -350.19866377784894, -255.48189427014447, -221.9729633619
[-452.55425210098, -434.60242602628597, -373.30997915456845, -354.3591665237355, -273.08305813142744, -319.91979459650
```

将图像列向量从 300 维压缩到 200 维，并再次还原，生成的压缩后的图像为：

将图像列向量从 300 维压缩到 100 维，并再次还原，生成的压缩后的图像为：



矢量量化：

算法描述：
输入：原始待压缩图像，所需质心的数目 k
输出：压缩后的图像，该图像中的所有像素点只由 k 种像素点构成

1. 将输入的二维图像矩阵，展开变成一维矩阵
2. 使用 KMeans 算法，对图像中的每一个像素向量进行聚类，聚为 k 类
3. 对聚类后的每一类，求出其质心向量值
4. 将原始图像中的每一个像素点值替换为其同类质心值

核心源程序：

```
def compress_image(img, num_clusters):
    """
    使用矢量量化的方法，对输入图片进行压缩
    :param img: 输入图片 （四通道 R、G、B alpha）
    :param num_clusters: 矢量量化的质心个数
    :return:
    """
```

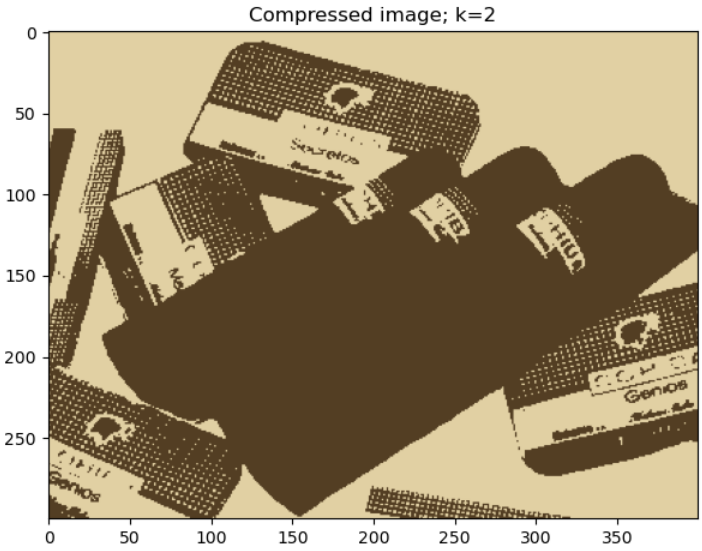
```
    # 原始输入图像为四通道图像，红、蓝、绿、透明度，将该图像所有的像素点进行重排列
    # 生成矩阵形式的数据，矩阵的每一行为一个像素点的 R,G,B,alpha 向量
    X = img.reshape((-1, 4))

    # 使用 KMeans 算法进行聚类，聚为 num_clusters 类
    kmeans = cluster.KMeans(n_clusters=num_clusters, n_init=4,
random_state=5)  # 配置聚类参数
    kmeans.fit(X)  # 开始聚类
    centroids = kmeans.cluster_centers_.squeeze()  # 求出质心向量值
    labels = kmeans.labels_  # 求出每个像素对应的类编号

    labels = labels.reshape((img.shape[0], img.shape[1]))
    # 将像素对应的类编号，重排列成原始图片大小
    result = np.zeros(img.shape)  # 初始化压缩后的图像
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            result[i][j] = centroids[labels[i][j]]
            # 将每个像素，使用其同类质心向量进行填充

    return result.astype(np.uint8)
```

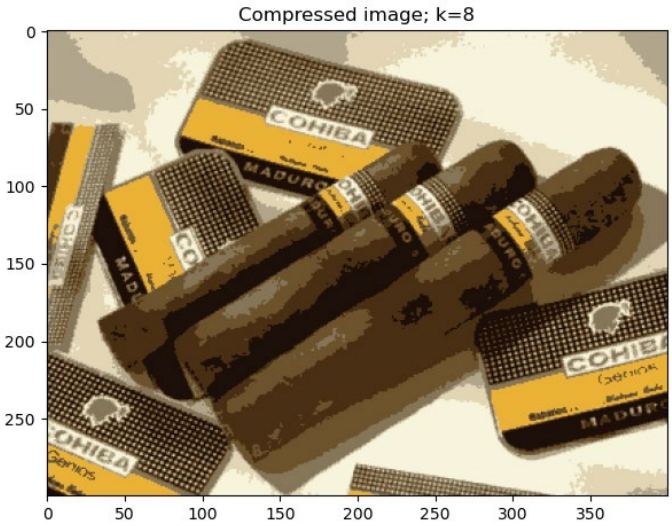
测试数据（输入、输出）：

输入：使用的原始图片；

原图

输出：


Compressed image; k=2
0
50
100
150
200
250
0
50
100
150
200
250
300
350


Compressed image; k=8
0
50
100
150
200
250
0
50
100
150
200
250
300
350


Compressed image; k=64
0
50
100
150
200
250
0
50
100
150
200
250
300
350

第 3 次实验　LSH 索引实现

本实验算法参考于 http://yangyi-bupt.github.io/ml/2015/08/28/lsh.html
使用了随机超平面投影方法实现局部敏感哈希

算法描述：

LSH 索引构建算法

输入：k 维向量数据集合
输出：LSH 索引

1. 初始化大小为 hash_size * input_dim 的，符合标准分布的二维矩阵 v，作为超平面（hash_size 输入数据待生成的哈希值的二进制长度，input_dim 为输入的数据维度）
2. 对输入的数据集中的每一个数据，执行以下算法：
   a. 计算输入向量数据 x 与超平面 v 的点积：x·v
   b. 使用 sgn 符号函数计算哈希值的每一个二进制位 h(v) = sgn(x·v)
      其中，sgn 函数为：

$$sgn(x) = \begin{cases} 1, x \geq 0 \\ 0, x < 0 \end{cases}$$

   c. 使用 sgn 函数计算出的哈希值作为索引表中的 key 值，原始的数据作为 value，将该记录在索引表中

LSH 索引查询算法

输入：一个 k 为的向量数据
输出：与该向量数据使用随机超平面投影方法实现的局部敏感哈希值结果相同的索引表中的数据集合

1. 对该向量数据使用与构建索引时相同的局部敏感哈希算法，计算哈希值
2. 以该哈希值作为 key，从索引表中查询具有相同 key 值的向量数据
3. 对查询出的所有数据一次计算与用户输入的查询数据的欧式距离
4. 按照欧式距离从小到大对查询结果排序，返回查询结果。

核心源程序：

LSH 索引构建

```
def index(self, input_point, extra_data=None):
    """
    为给定的单个数据 input_point 建立索引
    """

    if isinstance(input_point, np.ndarray):
        #将 numpy array 格式的数据转换为标准的 python list 格式数据
        input_point = input_point.tolist()
```

```
    if extra_data:
        value = (tuple(input_point), extra_data)
    else:
        value = tuple(input_point) # 转换为不可变元组

    for i, table in enumerate(self.hash_tables):
        table.append_val(self._hash(self.uniform_planes[i],
input_point), value) # 计算输入点的哈希值并将该哈希值作为 key， 原始数据作为
value 插入到索引表中
```

```
def _hash(self, planes, input_point):
    """
    为 input_point 生成二进制的哈希

    :param planes: 平面是尺寸为`hash_size` * `input_dim`的随机均匀平面
    :param input_point: 仅包含数字的输入数据
    """
    input_point = np.array(input_point)  # 转换为 numpy 数组，点积速度更
快
    projections = np.dot(planes, input_point)  # 使用随机超平面与输入数据
点积
    return "".join(['1' if i > 0 else '0' for i in projections]) #
使用 0 1 符号函数生成每个二进制位
```

LSH 索引查询

```
def query(self, query_point, num_results=None, distance_func=None):
    """
    查询

    :param query_point: 待查询数据
    :param num_results: 限制查询结果，只取前 num_results 个，若不限制，返回
所有哈希值相同的结果
    :param distance_func: 查询结果排序时使用的距离计算函数，默认采用欧式距离
    """

    candidates = set() # 初始化返回结果，当前为空集
        # 选择用户输入的距离计算函数
        if distance_func == "euclidean":
            d_func = LSHash.euclidean_dist_square # 使用欧式距离函数
        elif distance_func == "true_euclidean":
            d_func = LSHash.euclidean_dist
        elif distance_func == "centred_euclidean":
            d_func = LSHash.euclidean_dist_centred
        elif distance_func == "cosine":
            d_func = LSHash.cosine_dist
        elif distance_func == "l1norm":
            d_func = LSHash.l1norm_dist
        else:
```

```
            raise ValueError("The distance function name is
invalid.")

        for i, table in enumerate(self.hash_tables):
            binary_hash = self._hash(self.uniform_planes[i],
query_point) # 计算用户输入的待查找数据的哈希值
            candidates.update(table.get_list(binary_hash)) # 查找与查询
数据具有相同哈希值的数据
    # rank candidates by distance function
    candidates = [(ix, d_func(query_point, self._as_np_array(ix)))
                  for ix in candidates] # 对所有查询结果计算出与待查找数据的
欧式距离
    candidates.sort(key=lambda x: x[1]) # 按照欧式距离进行结果排序

    return candidates[:num_results] if num_results else candidates
```

测试数据（输入、输出）：

输入：corel 数据集 ColorHistogram （颜色直方图）数据，该数据共有 68040 行，对 corel 数据集中的每个图像计算颜色直方图信息，每行对应一张图片，每行有 32 维，每个维度的值是整个图像中每种颜色的密度 （对应 HSV 色彩空间的 32 种颜色）颜色直方图数据若相近，表明两个图像的色彩相似。

输出：构建 LSH 索引，并通过该索引查找输出与给定的颜色直方图数据哈希值相同的数据集中的数据。

```
Python 3.7.7 (default, May  6 2020, 11:45:54) [MSC v.1916 64 bit (AMD64)] on win32
>>> runfile('D:/workspace/DigitalMedia/LSH/lshTest.py', wdir='D:/workspace/DigitalMedi
从 asc 文件中读取数据...
删除数据中的第一列数据（该列为行号）...
初始化 LSH 索引表...
为前 1000 个数据建立 LSH 索引...
当前进度: 0 / 1000
当前进度: 100 / 1000
当前进度: 200 / 1000
当前进度: 300 / 1000
当前进度: 400 / 1000
当前进度: 500 / 1000
当前进度: 600 / 1000
当前进度: 700 / 1000
当前进度: 800 / 1000
当前进度: 900 / 1000
Time used:  0.02041190000000004
索引建立完成!
```

```
待查找的数据为: [8.77300e-03 1.20000e-05 1.00000e-06 0.00000e+00 1.45870e-02 0.00000e+00
 0.00000e+00 0.00000e+00 2.27100e-02 0.00000e+00 0.00000e+00 0.00000e+00
 6.78210e-02 1.04000e-04 0.00000e+00 0.00000e+00 6.35533e-01 1.62195e-01
 8.00000e-06 2.10000e-05 8.07300e-02 5.73000e-03 1.00000e-06 0.00000e+00
 1.66700e-03 0.00000e+00 0.00000e+00 0.00000e+00 2.10000e-04 1.00000e-06
 0.00000e+00 0.00000e+00]
```

```
查找到哈希值相同的数据共有 19 个，分别为:
((0.008773, 1.2e-05, 1e-06, 0.0, 0.014587, 0.0, 0.0, 0.0, 0.02271, 0.0, 0.0, 0.0, 0.067821, 0.0
((0.010208, 0.0, 0.0, 0.0, 0.019064, 0.0, 0.0, 0.0, 0.027613, 0.0, 0.0, 0.0, 0.072414, 0.0, 0.0
((0.013965, 1e-06, 0.0, 0.0, 0.010953, 1e-06, 0.0, 0.0, 0.007397, 0.0, 0.0, 0.0, 0.025732, 0.0
((0.004064, 0.000834, 0.000208, 0.000104, 0.008755, 0.004481, 0.002292, 0.000938, 0.051983, 0.0
((0.014793, 0.001876, 0.006563, 0.006251, 0.01, 0.000313, 0.0, 0.000208, 0.010417, 0.000104, 0
((0.017399, 0.001877, 2e-06, 0.00073, 0.022084, 0.003854, 0.000521, 0.000417, 0.005001, 0.0001
((0.000836, 0.000209, 0.000209, 0.000626, 0.009271, 0.010625, 0.00125, 0.002292, 0.044167, 0.0
((0.015109, 0.030627, 0.001355, 0.001667, 0.034796, 0.055733, 0.040012, 0.00054, 0.020523, 0.0
((0.008855, 0.000939, 2e-06, 2e-06, 0.025105, 0.003335, 1e-06, 1e-06, 0.035209, 0.001881, 7e-0
((0.008542, 0.001251, 0.001875, 0.01198, 0.014794, 0.000209, 0.0, 0.000104, 0.004585, 0.0, 0.0
((0.070529, 0.012613, 0.005108, 2e-06, 0.103341, 0.033545, 0.013126, 0.000938, 0.043338, 0.045
((0.018959, 0.008335, 0.00615, 0.003022, 0.065523, 0.117617, 0.042322, 0.005842, 0.141251, 0.0
((0.069586, 0.016668, 0.009167, 0.008959, 0.131566, 0.074794, 0.022084, 0.011563, 0.15386, 0.0
((0.002712, 0.001667, 0.001354, 0.000521, 0.003336, 0.001667, 0.000729, 0.000833, 0.005523, 0.0
((0.032614, 0.028575, 0.036678, 0.052712, 0.031356, 0.019381, 0.007709, 0.008854, 0.05448, 0.0
((0.003334, 0.004584, 0.002188, 0.00125, 0.086687, 0.004429, 0.000839, 0.000626, 0.06646, 0.00
((0.042709, 1e-06, 0.000209, 0.002292, 0.082191, 0.001586, 0.000839, 0.010105, 0.11271, 0.0026
((0.008237, 0.000729, 0.000104, 0.0, 0.000837, 2e-06, 1e-06, 1e-06, 0.001467, 9e-06, 3e-06, 5e
((0.021667, 0.033646, 0.020313, 0.00948, 0.057501, 0.065626, 0.021148, 0.007396, 0.049173, 0.1
Time used:  0.0002465000000002604
```

在计算过程中由于使用了 numpy 对向量计算进行加速，为 1000 个 32 维向量建立 LSH 索引，耗时仅为 0.02 秒，查询相同哈希值的数据，耗时仅为 0.000246 秒。

然而，由于给定的数据集中不包含标签信息，无法判断出哪些数据应该属于同一类，因此，无法计算出召回率和准确率。

第 4 次实验　SIFT 特征提取和匹配

算法描述：

SIFT 特征提取算法

输入：原始图像
输出：图像对应的 128 维的特征向量集

1. 对输入图像尺度空间的极点进行检测：
   - 尺度空间内核是高斯函数。因此假设 I(x,y)I(x,y)是原始图像，G(x,y,σ)G(x,y,σ)是尺度空间可变的高斯函数，则一个图像的尺度空间可以定义为 $L(x,y,\sigma)=G(x,y,\sigma)*I(x,y)$ 其中，*表示的是卷积运算，σ 表示尺度空间的大小，σ 越大则表示越模糊，表示图像的概貌，σ 越小则表示越清晰，表示图像的细节。高斯函数 G 的定义为：

     $$G(x,y,\sigma)=\frac{1}{2\pi\sigma^2}e^{-(x^2+y^2)/2\sigma^2}$$

   - 为了在尺度空间中找到稳定不变的极值点，在 SIFT 算法中使用了高斯差分(DOG)函数，定义为：

     $$\begin{aligned}D(x,y,\sigma) &= [G(x,y,k\sigma)-G(x,y,\sigma)]*I(x,y)\\ &= L(x,y,k\sigma)-L(x,y,\sigma)\end{aligned}$$

     其中 kσ 和 σ 是连续的两个图像的平滑尺度，所得到的差分图像在高斯差分金字塔中。
   - 对图像空间内的每一个点与其周围的 26 个点的高斯查分值进行比对，如果是最大值或最小值点，则该点为极值点
2. 对特征点进行精确定位：
   对得到的极值点进行三维泰勒展开

$$f\left(\begin{bmatrix}x\\y\\\sigma\end{bmatrix}\right)\approx f\left(\begin{bmatrix}x_0\\y_0\\\sigma_0\end{bmatrix}\right)+\left[\frac{\partial f}{\partial x}\ \frac{\partial f}{\partial y}\ \frac{\partial f}{\partial \sigma}\right]\left(\begin{bmatrix}x\\y\\\sigma\end{bmatrix}-\begin{bmatrix}x_0\\y_0\\\sigma_0\end{bmatrix}\right)+$$

$$\frac{1}{2}\left([x\quad y\quad \sigma]-[x_0\quad y_0\quad \sigma_0]\right)\begin{bmatrix}\frac{\partial^2 f}{\partial x\partial x} & \frac{\partial^2 f}{\partial x\partial y} & \frac{\partial^2 f}{\partial x\partial \sigma}\\ \frac{\partial^2 f}{\partial x\partial y} & \frac{\partial^2 f}{\partial y\partial y} & \frac{\partial^2 f}{\partial y\partial \sigma}\\ \frac{\partial^2 f}{\partial x\partial \sigma} & \frac{\partial^2 f}{\partial y\partial \sigma} & \frac{\partial^2 f}{\partial \sigma\partial \sigma}\end{bmatrix}\left(\begin{bmatrix}x\\y\\\sigma\end{bmatrix}-\begin{bmatrix}x_0\\y_0\\\sigma_0\end{bmatrix}\right)$$

   若写成矢量形式，则为：

$$f(X)=f(X_0)+\frac{\partial f^T}{\partial X}(X-X_0)+\frac{1}{2}(X-X_0)^T\frac{\partial^2 f}{\partial X^2}(X-X_0)$$

   在这里 X_0 表示离散的差值中心，XX 表示拟合后连续空间的差值点坐标，则设 $\hat{X}=X-X_0$，表示偏移量，带入上式，令求得的导数为 0，则有

$$\hat{X}=-\frac{\partial^2 f^{-1}}{\partial X^2}\frac{\partial f}{\partial X}$$

把极值点带入到原公式中，则有结果

$$f(\hat{X}) = f(X_0) + \frac{1}{2}\frac{\partial f^T}{\partial X}\hat{X}$$

若得到的偏移量大于 0.5，则认为偏移量过大，需要把位置移动到拟合后的新位置，继续进行迭代求偏移量，若迭代过一定次数后偏移量仍然大于 0.5，则抛弃该点。如果迭代过程中有偏移量小于 0.5，则停止迭代。另外，如果 13 式中得到 $f(\hat{X})$过小，则抛弃该点，论文中阈值为 0.03（设灰度值为 0~1）

3. 去除不稳定的极值点
   图像中的物体的边缘位置的点的主曲率一般会比较高，通过主曲率来判断该点是否在物体的边缘位置。若该点位于边缘位置，则进行去除
4. 根据特征点所在的高斯尺度图像中的局部特征计算特征点的方向
5. 生成特称点描述符
   - 计算各个像素点的梯度幅值和梯度幅角；
   - 根据该像素点距离特征点的距离进行加权（即第一次高斯加权），该像素点的幅值乘以加权值；
   - 根据该像素点在所在的小正方形区域内距中心的距离进行加权（即第二次高斯加权）

核心源程序：

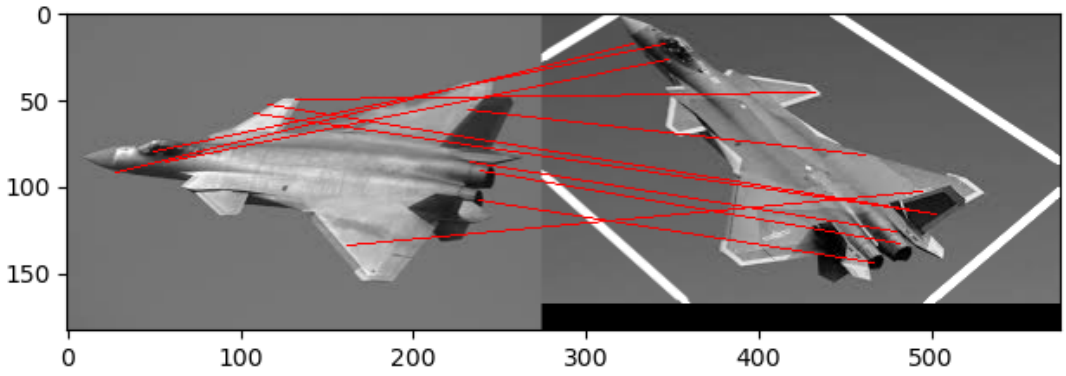
```
def computeKeypointsAndDescriptors(image, sigma=1.6,
num_intervals=3, assumed_blur=0.5, image_border_width=5):
    """
    为输入的图像计算 SIFT 描述符
    """
    image = image.astype('float32') # 图像类型预处理
    base_image = generateBaseImage(image, sigma, assumed_blur) #
对图像进行高斯模糊处理
    num_octaves = computeNumberOfOctaves(base_image.shape) # 计算
图像金字塔的层数
    gaussian_kernels = generateGaussianKernels(sigma,
num_intervals) # 计算金字塔中每层使用的高斯核
    gaussian_images = generateGaussianImages(base_image,
num_octaves, gaussian_kernels) # 生成图像金字塔
    dog_images = generateDoGImages(gaussian_images) # 使用高斯差分
函数对每层图像进行处理
    keypoints = findScaleSpaceExtrema(gaussian_images,
dog_images, num_intervals, sigma, image_border_width) # 寻找 DOG
图像中的极值点
    keypoints = removeDuplicateKeypoints(keypoints) #去除不稳定的极
值点
    keypoints = convertKeypointsToInputImageSize(keypoints) # 将
```

```
找到的极值点转换到原始图像尺度
    descriptors = generateDescriptors(keypoints, gaussian_images)
# 生成 128 维描述符
    return keypoints, descriptors
```

测试数据（输入、输出）：

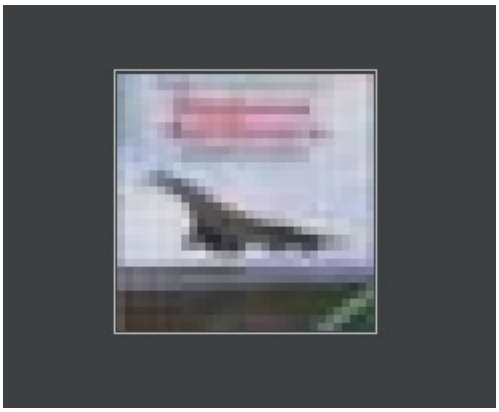
本程序首先对两幅飞行器的不同角度照片进行 SIFT 特征提取和匹配，判断程序的正确性：
输入：歼 20 飞行器的两张不同角度拍摄照片

输出：SIFT 特征提取和匹配之后的图像

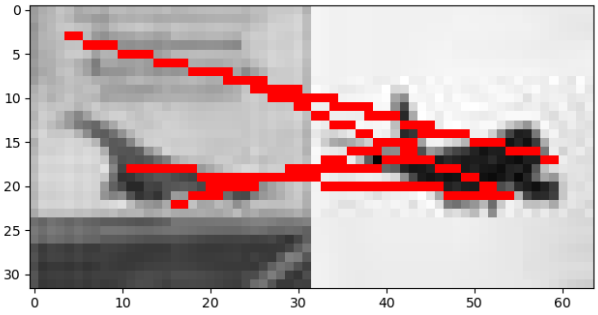


可以看到，成功对输入图像中的 SIFT 特征点进行了匹配，机翼、驾驶舱、机尾喷口等区域均实现了正确的匹配，并且识别出了经过透视变换后的图像区域。

之后按照题目要求，对给定数据集（cifar-10）中与测试集中 0 号类别中的第一个图像的进行 SIFT 特征提取，并从训练集中寻找其最匹配的图像：
输入：



输出：




```
>>> runfile('D:/workspace/DigitalMedia/SIFT/compare.py', wdir='D:/workspace/DigitalMedia/
compare with: 0_10008.jpg
寻找到更相似的图片：0_10008.jpg，描述符距离和为：1400.3079528808594
compare with: 0_10010.jpg
寻找到更相似的图片：0_10010.jpg，描述符距离和为：1303.9246826171875
compare with: 0_10020.jpg
compare with: 0_10024.jpg
寻找到更相似的图片：0_10024.jpg，描述符距离和为：1253.5840148925781
compare with: 0_10031.jpg
寻找到更相似的图片：0_10031.jpg，描述符距离和为：1194.7677001953125
compare with: 0_10043.jpg
compare with: 0_10050.jpg
寻找到更相似的图片：0_10050.jpg，描述符距离和为：1072.7540588378906
compare with: 0_10061.jpg
compare with: 0_10064.jpg
compare with: 0_10115.jpg
compare with: 0_1012.jpg
compare with: 0_10124.jpg
compare with: 0_10130.jpg
compare with: 0_10141.jpg
```

题目所给出的数据集，图片分辨率过低，导致提取出的 SIFT 特征点过少，且不准确，图像的匹配效果较差，为了与训练集上的图像进行匹配，实验中对训练集中的每个图像提取出 SIFT 描述符，并与原图像的 SIFT 描述符按照欧氏距离进行比较，从中选出欧式距离最小的三对 SIFT 描述符的欧式距离之和作为两幅图像的接近程度。欧式距离之和越小，两幅图像越接近。

第 5 次实验　时空数据查询

算法描述：

算法 1 ：kd 树构造算法

输入：k 维度空间数据集 $T=\{x_1,x_2,\cdots,x_N\}$

输出：kd 树

(1) 开始：构造根结点,根结点对应于包含 T 的 k 维空间的超矩形区域
选择 $x^{(1)}$ 为坐标轴,以 T 中所有实例的 $x^{(1)}$ 坐标的中位数为切分点,将根结点对应的超矩形区域切分为两个子区域。切分由通过切分点并与坐标轴 $x^{(1)}$ 垂直的超平面实现
由根结点生成深度为 1 的左、右子结点:左子结点对应坐标 $x^{(1)}$ 小于切分点的子区域,右子结点对应于坐标 $x^{(1)}$ 大于切分点的子区域

(2) 重复：对深度为 j 的结点,选择 $x^{(l)}$ 为切分的坐标轴,$l=j\ (\mathrm{mod}\ k)+1$,以该结点的区域中所有实例的 $x^{(l)}$ 坐标的中位数为切分点,将该结点对应的超矩形区域切分为两个子区域.切分由通过切分点并与坐标轴 $x^{(l)}$ 垂直的超平面实现
由该结点生成深度为 j+1 的左、右子结点:左子结点对应坐标 $x^{(l)}$ 小于切分点的子区域,右子结点对应坐标 $x^{(l)}$ 大于切分点的子区域。
将落在切分超平面上的实例点保存在该结点

(3) 直到两个子区域没有实例存在时停止，从而形成 kd 树的区域划分

算法 2 ：使用 kd 树的最近邻搜索算法
输入：已经构造好的 kd 树，目标点 x
输出：x 的最近邻

(1) 在树中找出包含目标点 x 的叶结点:从根结点出发,递归地向下访问 kd 树,若目标点 x 当前维的坐标小于切分点的坐标,则移动到左子结点,否则移动到右子结点.直到子结点为叶结点为止
(2) 以此叶结点为“当前最近点”
(3) 递归地向上回退,在每个结点进行以下操作
   (a) 如果该结点保存的实例点比当前最近点距离目标点更近,则以该实例点为“当前最近点”
   (b) 当前最近点一定存在于该结点一个子结点对应的区域.检查该子结点的父结点的另一子结点对应的区域是否有更近的点,具体地,检查另一子结点对应的区域是否与以目标点为球心、以目标点与“当前最近点”间的距离为半径的超球体相交
   如果相交,可能在另一个子结点对应的区域内存在距目标点更近的点,移动到另一个子结点,接着,递归地进行最近邻搜索如果不相交,向上回退
(4) 当回退到根结点时,搜索结束.最后的“当前最近点”即为 x 的最近邻点

核心源程序：

```
def create(self, dataSet, depth = 0):   #创建 kd 树，返回根结点
    """
    构建 kd 树
    :param dataSet: 以矩阵形式存储的数据集，每行为一个元素
    :param depth: kd 树当前深度 （递归时使用，外部调用时，该值为 0）
    :return:
    """
    if (len(dataSet) > 0):
        m, n = np.shape(dataSet)    #求出样本行，列
        midIndex = int(m / 2) #中间数的索引位置
        axis = depth % n    #判断以哪个轴划分数据
        sortedDataSet = dataSet[dataSet[:, axis].argsort()]  #进行排序
        node = Node(sortedDataSet[midIndex]) #将节点数据域设置为中位数，具体参考下书本
        leftDataSet = sortedDataSet[: midIndex] #将中位数的左边创建 2 改副本
        rightDataSet = sortedDataSet[midIndex+1 :]
        node.lchild = self.create(leftDataSet, depth+1) #将中位数左边样本传入来递归创建树
        node.rchild = self.create(rightDataSet, depth+1)
        return node
    else:
        return None
```

```
def search(self, tree, x):
    """
    搜索 kd 树中，x 的最近邻
    :param tree: 构造好的 kd 树
    :param x: 待搜索的 k 维度 数据
    :return: kd 树中 x 的最近邻
    """
    self.nearestPoint = None  #保存最近的点
    self.nearestValue = 0   #保存最近的值
    def travel(node, depth = 0):    #递归搜索
        if node != None:    #递归终止条件
            n = len(x)  #特征数
            axis = depth % n    #计算轴
            if x[axis] < node.data[axis]:   #如果数据小于结点，则往左结点找
                travel(node.lchild, depth+1)
            else:
                travel(node.rchild, depth+1)

            #以下是递归完毕后，往父结点方向回朔
            distNodeAndX = self.dist(x, node.data)  #目标和节点的距离判断
            if self.nearestPoint is None : #确定当前点，更新最近的点和最近的值
                self.nearestPoint = node.data
                self.nearestValue = distNodeAndX
```

```
            elif (self.nearestValue > distNodeAndX):
                self.nearestPoint = node.data
                self.nearestValue = distNodeAndX

            if (abs(x[axis] - node.data[axis]) <= self.nearestValue):
#确定是否需要去子节点的区域去找（圆的判断）
                if x[axis] < node.data[axis]:
                    travel(node.rchild, depth+1)
                else:
                    travel(node.lchild, depth + 1)
    travel(tree)
    return self.nearestPoint
```

测试数据（输入、输出）：

输入：数据集：CA ，BJ，数据的最后两列代表经度和纬度，采用最后两维构建 kd 树并搜索

输出：

```
Python 3.7.7 (default, May  6 2020, 11:45:54) [MSC v.1916 64 bit (AMD64)] (
>>> runfile('D:/workspace/DigitalMedia/KDTree/kdCore.py', wdir='D:/workspac
开始加载 CA 数据集 ...
开始构造 kd 树 ...
Time used:  0.2253311
搜索 [-119.568, 33.1364] 的最近邻 ...
最近邻为: [-119.156    33.7763]
Time used:  0.0031759000000000093
开始加载 BJ 数据集 ...
开始构造 kd 树 ...
Time used:  0.020141399999999976
搜索 [115.411, 40.9097] 的最近邻 ...
最近邻为: [115.993    40.5184]
Time used:  0.0005538000000000487
```

性能分析：

实验中使用的 CA、BJ 数据集为二维数据，其中，CA 数据集共有 22227 个实例点，BJ 数据集共有 399 个实例点，对这两个数据集中的实例构造 kd 树：
CA 数据集耗时 0.2253311 秒
BJ 数据集耗时 0.0201413 秒

在这两个数据集上分别搜索单个元素的最近邻：
CA 数据集耗时 0.00317 秒
BJ 数据集耗时 0.00055 秒

CA 数据集大小约为 BJ 数据集大小的 56 倍，而构造 kd 树耗时约为 10 倍，搜索最近邻耗时约为 6 倍，

构建出的 k-D Tree 的树高最多为 $O(\log n)$ N 是训练实例数，构建 k-D Tree 时间复杂度的瓶颈在于快速选出一个维度上的中位数，并将在该维度上的值小于该中位数的置于中位数的左边，其余置于右边。如果每次都使用 numpy 的排序函数对该维度进行排序，时间复杂度是 $O(n\log^2 n)$ 的。因此，本算法中构建 K-D 树的时间复杂度为 $O(n\log^2 n)$

而如果实例点是随机分布的, kd 树搜索的平均计算复杂度是 $O(\log n)$. kd 树更适用于训练实例数远大于空间维数时的k近邻搜索.当空间维数接近训练实例数时,它的效率会迅速下降,几乎接近线性扫描。

个人收获：

简单说一下学习本课程的收获，或者对本课程有什么建议。