K-means 算法实现实验报告

201250070 郁博文

一、 数据预处理

首先在网上随机找一张图片。我选用的是 1600*900 的尺寸。为了方便后续计算，将其裁剪为 256*256 大小。

由于图片采用 RGB 三通道，所以先把图片中每个像素点的通道值归约到[0,1].，结果以 numpy 的矩阵存储。

```
def load_data(file_path):
    '''导入数据
    input:  file_path(string):文件的存储位置
    output: data(mat):数据
    '''
    f = open(file_path, "rb")  # 以二进制的方式打开图像文件
    data = []
    im = image.open(f)  # 导入图片
    m, n = im.size  # 得到图片的大小
    print(m, n)
    for i in range(m):
        for j in range(n):
            tmp = []
            x, y, z = im.getpixel((i, j))
            tmp.append(x / 256.0)
            tmp.append(y / 256.0)
            tmp.append(z / 256.0)
            data.append(tmp)
    f.close()
    return np.mat(data)
```

二、 K-means 算法原理

由于我们实现的是 K-means 算法而不是 K-means++，所以在第一步我们先随机初始化中心点。使用 np.random 方法在通道每一维的最大值和最小值之间随

机生成。

```python
def randCent(data, k):
    '''随机初始化聚类中心
    input:  data(mat):训练数据
        k(int):类别个数
    output: centroids(mat):聚类中心
    '''
    n = np.shape(data)[1]  # 属性的个数
    centroids = np.mat(np.zeros((k, n)))  # 初始化k个聚类中心
    for j in range(n):  # 初始化聚类中心每一维的坐标
        minJ = np.min(data[:, j])
        rangeJ = np.max(data[:, j]) - minJ
        # 在最大值和最小值之间随机初始化
        centroids[:, j] = minJ * np.mat(np.ones((k, 1))) + np.random.rand(k, 1) * rangeJ
    return centroids
```

我们采用欧式距离作为评判距离的标准。具体实现如下：将两个向量做差后乘上他们的转置向量就得到欧氏距离的平方。

```python
def distance(vecA, vecB):
    '''计算vecA与vecB之间的欧式距离的平方
    input:  vecA(mat)A点坐标
        vecB(mat)B点坐标
    output: dist[0, 0](float)A点与B点距离的平方
    '''
    dist = (vecA - vecB) * (vecA - vecB).T
    return dist[0, 0]
```

接下来就是 K-means 的具体实现。首先我们初始化一个矩阵，行代表像素点，第一列代表该像素点当前所属的类，第二列代表该像素点与那个类对应的中心的距离。

```python
m, n = np.shape(data)  # m：样本的个数，n：特征的维度
subCenter = np.mat(np.zeros((m, 2)))  # 初始化每一个样本所属的类别
```

然后就是循环遍历中心，计算每个像素点到中心的距离然后取距离最小的那个中心所在类为像素点所在类。

```
for i in range(m):
    minDist = np.inf  # 设置样本与聚类中心之间的最小的距离，初始值为争取穷
    minIndex = 0  # 所属的类别
    for j in range(k):
        # 计算i和每个聚类中心之间的距离
        dist = distance(data[i, ], centroids[j, ])
        if dist < minDist:
            minDist = dist
            minIndex = j
    # 判断是否需要改变
    if subCenter[i, 0] != minIndex:  # 需要改变
        change = True
        subCenter[i, ] = np.mat([minIndex, minDist])
```

在完成了对所有像素点的遍历后，我们需要计算新的中心点。这里采用了简单的取平均方式，遍历每个类别中每个像素点的三维坐标然后对每一维分别取平均。

```
# 重新计算聚类中心
for j in range(k):
    sum_all = np.mat(np.zeros((1, n)))
    r = 0  # 每个类别中的样本的个数
    for i in range(m):
        if subCenter[i, 0] == j:  # 计算第j个类别
            sum_all += data[i, ]
            r += 1
    for z in range(n):
        try:
            centroids[j, z] = sum_all[0, z] / r
            print(r)
        except:
            print(" r is zero")
```

循环退出的条件是在经历了类别判断的循环后，每个像素点的类别是否发生变化，如果有一个像素点的类别发生变化就说明整个算法还没有收敛，需要进一

步计算。

三、　图像生成

在完成了 K-means 算法的计算之后，我们已经获取了最后形成的 k 个聚类的像素点的三维坐标，不过这个坐标在第一部分中被归约到了[0,1]，所以需要再*256 以回到原本通道值。然后我们取出每个像素值对应的类别，根据类别将它们的 RGB 通道值改为对应中心的 RGB 通道值，最后用 PIL 把这些像素点拼成新的图片。这样就完成了 K-means 算法。

以下是结果展示：

原图：

K=2:

K=5: