第 8 章 K-均值聚类

内容提要

- K-均值聚类算法
- 二分 K-均值聚类算法
- 对聚类得到的簇进行后处理
- 对地理位置进行聚类

8.1 K-均值聚类

聚类是一种无监督的学习，它将相似的对象归到同一簇中。聚类的方法几乎可以应用所有对象，簇内的对象越相似，聚类的效果就越好。K-means 算法中的 k 表示的是聚类为 k 个簇，means 代表取每一个聚类中数据值的均值作为该簇的中心，或者称为质心，即用每一个的类的质心对该簇进行描述。

聚类和分类最大的不同在于，分类的目标是事先已知的，而聚类则不一样，聚类事先不知道目标变量是什么，类别没有像分类那样被预先定义出来，所以，聚类有时也叫无监督学习。

聚类分析试图将相似的对象归入同一簇，将不相似的对象归为不同簇，那么，显然需要一种合适的相似度计算方法，我们已知的有很多相似度的计算方法，比如欧氏距离，余弦距离，汉明距离等。事实上，我们应该根据具体的应用来选取合适的相似度计算方法。

当然，任何一种算法都有一定的缺陷，没有一种算法是完美的，有的只是人类不断追求完美，不断创新的意志。K-means 算法也有它的缺陷，但是我们可以通过一些后处理来得到更好的聚类结果，接着会给出一个更加有效的的称为二分 K-均值聚类算法，这些在后面都会一一讲到。

K-means 算法虽然比较容易实现，但是其可能收敛到局部最优解，且在大规模数据集上收敛速度相对较慢。

K-均值是发现给定数据集的 k 个簇的算法。簇个数 k 是用户给定的，每一个簇通过其质心（centroid），即簇中所有点的中心来描述。K-均值算法的工作流程是这样的。首先，随机确定 k 个初始点作为质心。然后将数据集中的每个点分配到一个簇中，具体来讲，为每个点找距其最近的质心，并将其分配给该质心所对应的簇。这一步完成之后，每个簇的质心更新为该簇所有点的平均值。

```
创建k个点作为起始质心（经常是随机选择）
当任意一个点的簇分配结果发生改变时
    对数据集中的每个数据点
        对每个质心
            计算质心与数据点之间的距离
        将数据点分配到距其最近的簇
    对每一个簇，计算簇中所有点的均值并将均值作为质心
```

上面提到“最近”质心的说法，意味着需要进行某种距离计算。我们可以使用所喜欢的任意距离度量方法。数据集上 K-均值算法的性能会受到所选距离计算方法的影响。下面给出 K-均值算法的代码实现。

从学在浙大下载数据：

K-均值聚类支持函数

```
from numpy import *

def loadDataSet(fileName):   #general function to parse tab -delimited floats
    dataMat = []             #assume last column is target value
    fr = open(fileName)
    for line in fr.readlines():
        curLine = line.strip().split('\t')
        fltLine = list(map(float,curLine) #map all elements to float())
        dataMat.append(fltLine)
    return dataMat

def distEclud(vecA, vecB):
    return sqrt(sum(power(vecA - vecB, 2))) #la.norm(vecA-vecB)

def randCent(dataSet, k):
    n = shape(dataSet)[1]
    centroids = mat(zeros((k,n)))#create centroid mat
    for j in range(n):#create random cluster centers, within bounds of each dimension
        minJ = min(dataSet[:,j])
        rangeJ = float(max(dataSet[:,j]) - minJ)
        centroids[:,j] = mat(minJ + rangeJ * random.rand(k,1))
    return centroids

if __name__ == '__main__'
    dataSet = loadDataSet('testSet.txt')
    dataMat = mat(dataSet)
    range = randCent(dataMat, 2)
    print(range)
    print(dataMat[0])
    print(dataMat[1])
    print(distEclud(dataMat[0], dataMat[1])
```

randCent() 为给定数据集构建一个包含 k 个随机质心的集合。随机质心必须要在整个数据集的边界之内，这可以通过找到数据集每一维的最小和最大值来完成。然后生成 0 到 1.0 之间的随机数并通过取值范围和最小值，以便确保随机点在数据的边界之内。

K-均值聚类算法

```
def kMeans(dataSet, k, distMeas=distEclud, createCent=randCent):
    m = shape(dataSet)[0]
    clusterAssment = mat(zeros((m,2)))#create mat to assign data points
                                      #to a centroid, also holds SE of each point
    centroids = createCent(dataSet, k)
    clusterChanged = True
    while clusterChanged:
        clusterChanged = False
        for i in range(m):#for each data point assign it to the closest centroid
```

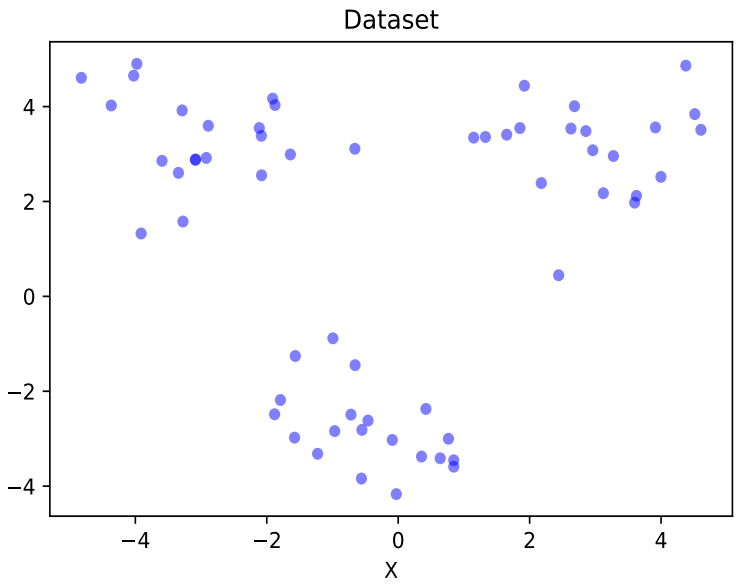

图 8.1: K 均值数据集

```
        minDist = inf; minIndex = -1
        for j in range(k):
            distJI = distMeas(centroids[j,:],dataSet[i,:])
            if distJI < minDist:
                minDist = distJI; minIndex = j
        if clusterAssment[i,0] != minIndex: clusterChanged = True
        clusterAssment[i,:] = minIndex,minDist**2
    print centroids
    for cent in range(k):#recalculate centroids
        ptsInClust = dataSet[nonzero(clusterAssment[:,0].A==cent)[0]]#get all the point in this
            cluster
        centroids[cent,:] = mean(ptsInClust, axis=0) #assign centroid to mean
return centroids, clusterAssment
```

kMeans() 函数接受 4 个输入参数。只有数据集及簇的数目是必选参数，而用来计算距离和创建初始质心的函数都是可选的。kMeans() 函数一开始确定数据集中数据点的总数，然后创建一个矩阵来存储每个点的簇分配结果。簇分配结果矩阵 clusterAssment 包含两列：一列记录簇索引值，第二列存储误差。这里的误差是指当前点到簇质心的距离，后边会使用该误差来评价聚类的效果。

按照上述方式（即计算质心 分配 重新计算）反复迭代，直到所有数据点的簇分配结果不再改变为止。程序中可以创建一个标志变量 clusterChanged，如果该值为 True，则继续迭代。上述迭代使用 while 循环来实现。接下来遍历所有数据找到距离每个点最近的质心，这可以通过对每个点遍历所有质心并计算点到每个质心的距离来完成。计算距离是使用 distMeas 参数给出的距离函数，默认距离函数是 distEclud()，该函数的实现已经在程序清单 10-1 中给出。如果任一点的簇分配结果发生改变，则更新 clusterChanged 标志。

最后，遍历所有质心并更新它们的取值。具体实现步骤如下：首先通过数组过滤来获得给定簇的所有点；然后计算所有点的均值，选项 axis = 0 表示沿矩阵的列方向进行均值计算；最

后，程序返回所有的类质心与点分配结果。

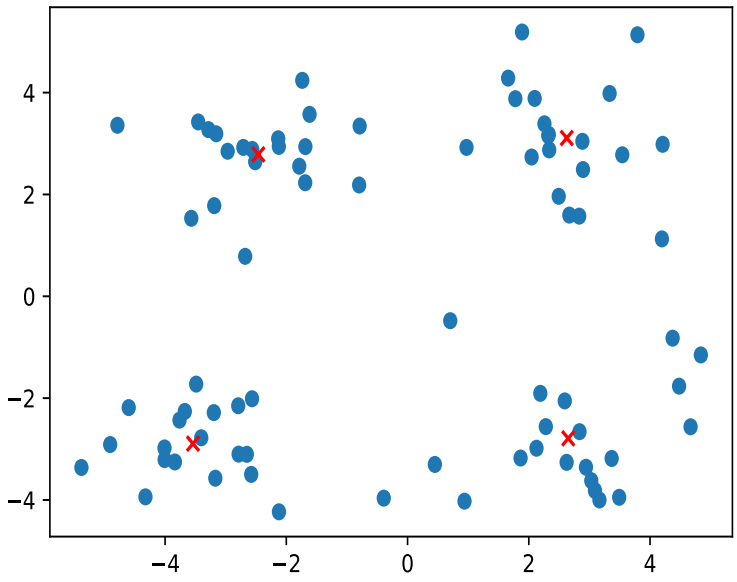

图 8.2: K 均值聚类结果

需要说明的是，在算法中，相似度的计算方法默认的是欧氏距离计算，当然也可以使用其他相似度计算函数，比如余弦距离；算法中，k 个类的初始化方式为随机初始化，并且初始化的质心必须在整个数据集的边界之内，这可以通过找到数据集每一维的最大值和最小值；然后最小值 + 取值范围 *0 到 1 的随机数，来确保随机点在数据边界之内。

在实际的 K-means 算法中，采用计算质心-分配-重新计算质心的方式反复迭代，算法停止的条件是，当然数据集所有的点分配的距其最近的簇不在发生变化时，就停止分配，更新所有簇的质心后，返回 k 个类的质心 (一般是向量的形式) 组成的质心列表，以及存储各个数据点的分类结果和误差距离的平方的二维矩阵。

上面返回的结果中，之所以存储每个数据点距离其质心误差距离平方，是便于后续的算法预处理。因为 K-means 算法采取的是随机初始化 k 个簇的质心的方式，因此聚类效果又可能陷入局部最优解的情况，局部最优解虽然效果不错，但不如全局最优解的聚类效果更好。所以，后续会在算法结束后，采取相应的后处理，使算法跳出局部最优解，达到全局最优解，获得最好的聚类效果。

有时候当我们观察聚类的结果图时，发现聚类的效果没有那么好，如8.3所示，K-means 算法在 k 值选取为 3 时的聚类结果，我们发现，算法能够收敛但效果较差。显然，这种情况的原因是，算法收敛到了局部最小值，而并不是全局最小值，局部最小值显然没有全局最小值的结果好。

那么，既然知道了算法已经陷入了局部最小值，如何才能够进一步提升 K-means 算法的效果呢?

一种用于度量聚类效果的指标是 **SSE**，即误差平方和，为所有簇中的全部数据点到簇中心的误差距离的平方累加和。SSE 的值如果越小，表示数据点越接近于它们的簇中心，即质心，聚类效果也越好。因为，对误差取平方后，就会更加重视那些远离中心的数据点。

显然，我们知道了一种改善聚类效果的做法就是降低 SSE，那么如何在保持簇数目不

变的情况下提高簇的质量呢?

一种方法是：我们可以将具有最大 SSE 值得簇划分为两个簇（因为，SSE 最大的簇一般情况下，意味着簇内的数据点距离簇中心较远），具体地，可以将最大簇包含的点过滤出来并在这些点上运行 K-means 算法，其中 k 设为 2.

同时，当把最大的簇 (上图中的下半部分) 分为两个簇之后，为了保证簇的数目是不变的，我们可以再合并两个簇。具体地：

一方面我们可以合并两个最近的质心所对应的簇，即计算所有质心之间的距离，合并质心距离最近的两个质心所对应的簇。

另一方面，我们可以合并两个使得 SSE 增幅最小的簇，显然，合并两个簇之后 SSE 的值会有所上升，那么为了最好的聚类效果，应该尽可能使总的 SSE 值小，所以就选择合并两个簇后 SSE 涨幅最小的簇。具体地，就是计算合并任意两个簇之后的总得 SSE，选取合并后最小的 SSE 对应的两个簇进行合并。这样，就可以满足簇的数目不变。

上面，是对已经聚类完成的结果进行改善的方法，在不改变 k 值的情况下，上述方法能够起到一定的作用，会使得聚类效果得到一定的改善。那么，下面要讲到的是一种克服算法收敛于局部最小值问题的 K-means 算法。即二分 k-均值算法。

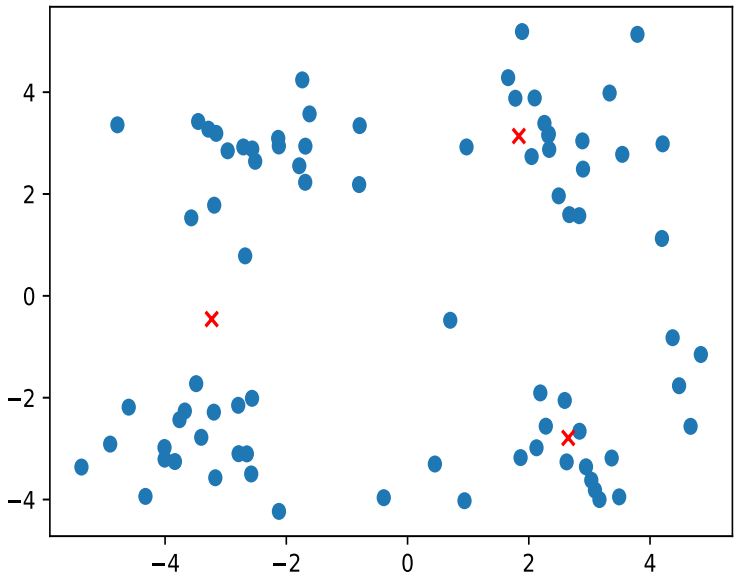

图 8.3: K 均值聚类结果

8.2 二分 K-means 算法

二分 K-means 算法首先将所有点作为一个簇，然后将簇一分为二。之后选择其中一个簇继续进行划分，选择哪一个簇取决于对其进行划分是否能够最大程度的降低 SSE 的值。上述划分过程不断重复，直至划分的簇的数目达到用户指定的值为止。

二分 K-means 算法的伪代码如下：

```
将所有点看成一个簇
当簇数目小于k时
    对于每一个簇
```

```
        计算总误差
        在给定的簇上面进行K-均值聚类 (k=2)
        计算将该簇一分为二之后的总误差
    选择使得误差最小的那个簇进行划分操作
```

另一种做法是选择 SSE 最大的簇进行划分，直到簇数目达到用户指定的数目为止。这个做法听起来并不难实现。下面就来看一下该算法的实际效果。

二分 K-均值聚类算法

```
def biKmeans(dataSet, k, distMeas=distEclud):
    m = shape(dataSet)[0]
    clusterAssment = mat(zeros((m,2)))
    centroid0 = mean(dataSet, axis=0).tolist()[0]
    centList =[centroid0] #create a list with one centroid
    for j in range(m):#calc initial Error
        clusterAssment[j,1] = distMeas(mat(centroid0), dataSet[j,:])**2
    while (len(centList) < k):
        lowestSSE = inf
        for i in range(len(centList)):
            ptsInCurrCluster = dataSet[nonzero(clusterAssment[:,0].A==i)[0],:]#get the data points
                currently in cluster i
            centroidMat, splitClustAss = kMeans(ptsInCurrCluster, 2, distMeas)
            sseSplit = sum(splitClustAss[:,1])#compare the SSE to the currrent minimum
            sseNotSplit = sum(clusterAssment[nonzero(clusterAssment[:,0].A!=i)[0],1])
            print "sseSplit, and notSplit: ",sseSplit,sseNotSplit
            if (sseSplit + sseNotSplit) < lowestSSE:
                bestCentToSplit = i
                bestNewCents = centroidMat
                bestClustAss = splitClustAss.copy()
                lowestSSE = sseSplit + sseNotSplit
        bestClustAss[nonzero(bestClustAss[:,0].A == 1)[0],0] = len(centList) #change 1 to 3,4, or
            whatever
        bestClustAss[nonzero(bestClustAss[:,0].A == 0)[0],0] = bestCentToSplit
        print 'the bestCentToSplit is: ',bestCentToSplit
        print 'the len of bestClustAss is: ', len(bestClustAss)
        centList[bestCentToSplit] = bestNewCents[0,:].tolist()[0]#replace a centroid with two best
            centroids
        centList.append(bestNewCents[1,:].tolist()[0])
        clusterAssment[nonzero(clusterAssment[:,0].A == bestCentToSplit)[0],:]= bestClustAss#reassign
            new clusters, and SSE
    return mat(centList), clusterAssment
```

在上述算法中，直到簇的数目达到 k 值，算法才会停止。在算法中通过将所有的簇进行划分，然后分别计算划分后所有簇的误差。选择使得总误差最小的那个簇进行划分。划分完成后，要更新簇的质心列表，数据点的分类结果及误差平方。具体地，假设划分的簇为 m（m<k）个簇中的第 i 个簇，那么这个簇分成的两个簇后，其中一个取代该被划分的簇，成为第 i 个簇，并计算该簇的质心；此外，将划分得到的另外一个簇，作为一个新的簇，成为第 m+1 个簇，并计算该簇的质心。此外，算法中还存储了各个数据点的划分结果和误差平方，此时也

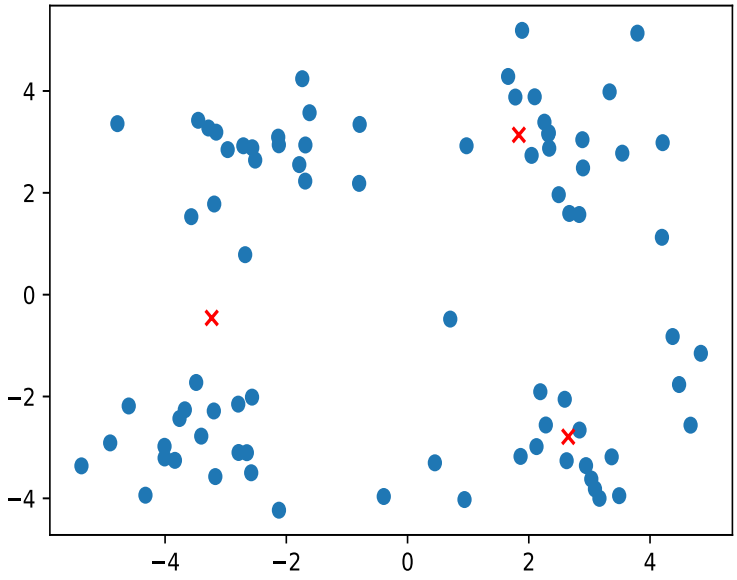

图 8.4: 二分 K 均值聚类结果

应更新相应的存储信息。这样，重复该过程，直至簇个数达到 k。

通过上述算法，之前陷入局部最小值的的这些数据，经过二分 K-means 算法多次划分后，逐渐收敛到全局最小值，从而达到了令人满意的聚类效果。

第 8 章 练习

1. 示例：对地图上的点进行聚类

   现在有一个存有 70 个地址和城市名的文本，而没有这些地点的距离信息。而我们想要对这些地点进行聚类，找到每个簇的质心地点，从而可以安排合理的行程，即质心之间选择交通工具抵达，而位于每个质心附近的地点就可以采取步行的方法抵达。显然，K-means 算法可以为我们找到一种更加经济而且高效的出行方式。

   通过地址信息获取相应的经纬度信息

   那么，既然没有地点之间的距离信息，怎么计算地点之间的距离呢？又如何比较地点之间的远近呢？

   从学在浙大上下载数据：

```
import matplotlib
import matplotlib.pyplot as plt
def clusterClubs(numClust=5):
    datList = []
    for line in open('places.txt').readlines():
        lineArr = line.split('\t')
        datList.append([float(lineArr[4]), float(lineArr[3])])
    datMat = mat(datList)
    myCentroids, clustAssing = biKmeans(datMat, numClust, distMeas=distSLC)
    fig = plt.figure()
    rect=[0.1,0.1,0.8,0.8]
    scatterMarkers=['s', 'o', '^', '8', 'p', \
```

```
                'd', 'v', 'h', '>', '<']
    axprops = dict(xticks=[], yticks=[])
    ax0=fig.add_axes(rect, label='ax0', **axprops)
    imgP = plt.imread('Portland.png')
    ax0.imshow(imgP)
    ax1=fig.add_axes(rect, label='ax1', frameon=False)
    for i in range(numClust):
        ptsInCurrCluster = datMat[nonzero(clustAssing[:,0].A==i)[0],:]
        markerStyle = scatterMarkers[i % len(scatterMarkers)]
        ax1.scatter(ptsInCurrCluster[:,0].flatten().A[0], ptsInCurrCluster[:,1].flatten().A[0],
            marker=markerStyle, s=90)
    ax1.scatter(myCentroids[:,0].flatten().A[0], myCentroids[:,1].flatten().A[0], marker='+', s
        =300)
    plt.show()
```

8.3 Sklearn 实现 K 均值聚类

8.3.1 利用肘部法选择 K 值

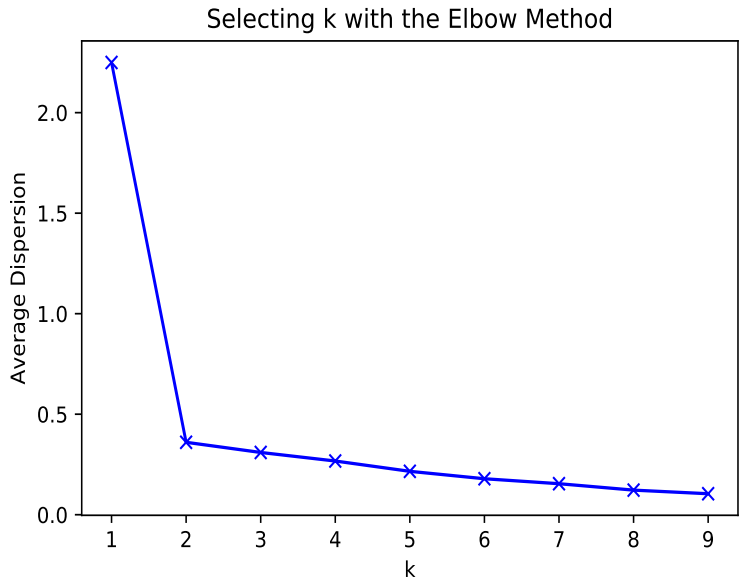

图 8.5: 利用肘部方法选择 K 值

```
import numpy as np
from sklearn.cluster import KMeans
from scipy.spatial.distance import cdist
import matplotlib.pyplot as plt

c1x = np.random.uniform(0.5, 1.5, (1, 10))
c1y = np.random.uniform(0.5, 1.5, (1, 10))
c2x = np.random.uniform(3.5, 4.5, (1, 10))
c2y = np.random.uniform(3.5, 4.5, (1, 10))
x = np.hstack((c1x, c2x))
```

```
y = np.hstack((c1y, c2y))
X = np.vstack((x, y)).T

K = range(1, 10)
meanDispersions = []
for k in K:
    kmeans = KMeans(n_clusters=k)
    kmeans.fit(X)
    meanDispersions.append(sum(np.min(cdist(X, kmeans.cluster_centers_, 'euclidean'), axis=1)) / X.
        shape[0])

plt.plot(K, meanDispersions, 'bx-')
plt.xlabel('k')
plt.ylabel('Average Dispersion')
plt.title('Selecting k with the Elbow Method')
plt.savefig('k-value.pdf', bbox_inches='tight')
plt.show()
```

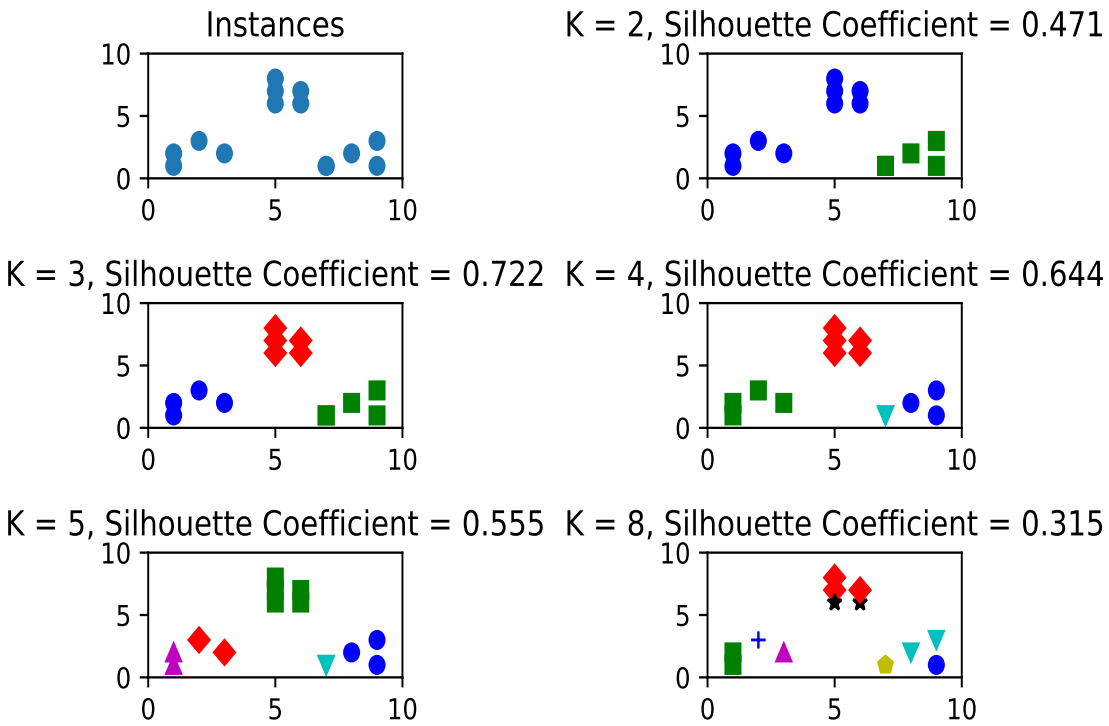

图 8.6: 不同 K 值下的轮廓系数

轮廓系数是对聚类紧密程度和稀疏程度的度量。当聚类的质量上升时，轮廓系数上升。当聚类内部很紧密且彼此之间距离很远时，轮廓系数很大；对于体积很大且相互重叠的聚类，轮廓系数很小。轮廓系数是在每一个实例上进行计算。对于实例集合，轮廓系数等于每个实例轮廓系数的平均值。轮廓系数计算公式为：

$$s = \frac{ab}{\max(a, b)} \tag{8.1}$$

其中 a 是聚类中实例之间的平均距离。b 是聚类的实例和最接近的聚类的实例之间的平均距离。

下例是从一个玩具数据集运行了 4 次 K 均值算法，创建了 2 个，3 个，4 个，5 个和 8 个聚类，并在每轮中计算轮廓系数。

```
import numpy as np
from sklearn.cluster import KMeans
from sklearn import metrics
import matplotlib.pyplot as plt

plt.subplot(3, 2, 1)
x1 = np.array([1, 2, 3, 1, 5, 6, 5, 5, 6, 7, 8, 9, 7, 9])
x2 = np.array([1, 3, 2, 2, 8, 6, 7, 6, 7, 1, 2, 1, 1, 3])

X = np.array(list(zip(x1, x2))).reshape(len(x1), 2)

plt.xlim([0, 10])
plt.ylim([0, 10])
plt.title('Instances')
plt.scatter(x1, x2)
colors = ['b', 'g', 'r', 'c', 'm', 'y', 'k', 'b']
markers = ['o', 's', 'D', 'v', '^', 'p', '*', '+']
tests = [2, 3, 4, 5, 8]
subplot_counter = 1
for t in tests:
    subplot_counter += 1
    plt.subplot(3, 2, subplot_counter)
    kmeans_model = KMeans(n_clusters=t).fit(X)
    for i, l in enumerate(kmeans_model.labels_):
        plt.plot(x1[i], x2[i], color=colors[l], marker=markers[l], ls='None')
    plt.xlim([0, 10])
    plt.ylim([0, 10])
    plt.title('K = %s, Silhouette Coefficient = %.03f' % (
        t, metrics.silhouette_score(X, kmeans_model.labels_, metric='euclidean')))

plt.subplots_adjust(wspace=1.2, hspace=1)

plt.savefig('couter.pdf', bbox_inches='tight')
plt.show()
```

当 K 值等于 3 时轮廓系数最大。将 K 值设置为 8 时，轮廓系数最小。

8.4 聚类的应用

8.4.1 图像量化

图像量化是一种有损压缩的方法，使用一种颜色来替换一张图片中一系列类似的颜色。在下面的例子中，我们使用聚类来找出包含一张图像中最重要颜色的压缩调色盘，然后使用压

缩调色盘重建图像。

首先读入图像并将其扁平化。

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.utils import shuffle
from PIL import Image

# First we read and flatten the image.
original_img = np.array(Image.open('tree.jpg'), dtype=np.float64) / 255
original_dimensions = tuple(original_img.shape)
width, height, depth = tuple(original_img.shape)
image_flattened = np.reshape(original_img, (width * height, depth))
```

然后，我们使用 K-均值算法从 1000 个随机选取的颜色样本中创建 64 个聚类。每个聚类都将成为压缩调色盘中的一个颜色。

```
image_array_sample = shuffle(image_flattened, random_state=0)[:1000]
estimator = KMeans(n_clusters=64, random_state=0)
estimator.fit(image_array_sample)
```

接下来，我们为原图中每个像素分配到哪个调色盘中。

```
cluster_assignments = estimator.predict(image_flattened)
```

最后，我们从压缩调色盘和聚类分配来创建压缩图片。

```
compressed_palette = estimator.cluster_centers_
compressed_img = np.zeros((width, height, compressed_palette.shape[1]))
label_idx = 0
for i in range(width):
    for j in range(height):
        compressed_img[i][j] = compressed_palette[cluster_assignments[label_idx]]
        label_idx += 1

plt.subplot(121)
plt.title('Original Image', fontsize=24)
plt.imshow(original_img)
plt.axis('off')
plt.subplot(122)
plt.title('Compressed Image', fontsize=24)
plt.imshow(compressed_img)
plt.axis('off')
plt.show()
```

8.4.2 通过聚类学习特征

在本节中，我们将通过聚类无标记数据来学习特征，然后使用学习到的特征建立监督分类器。

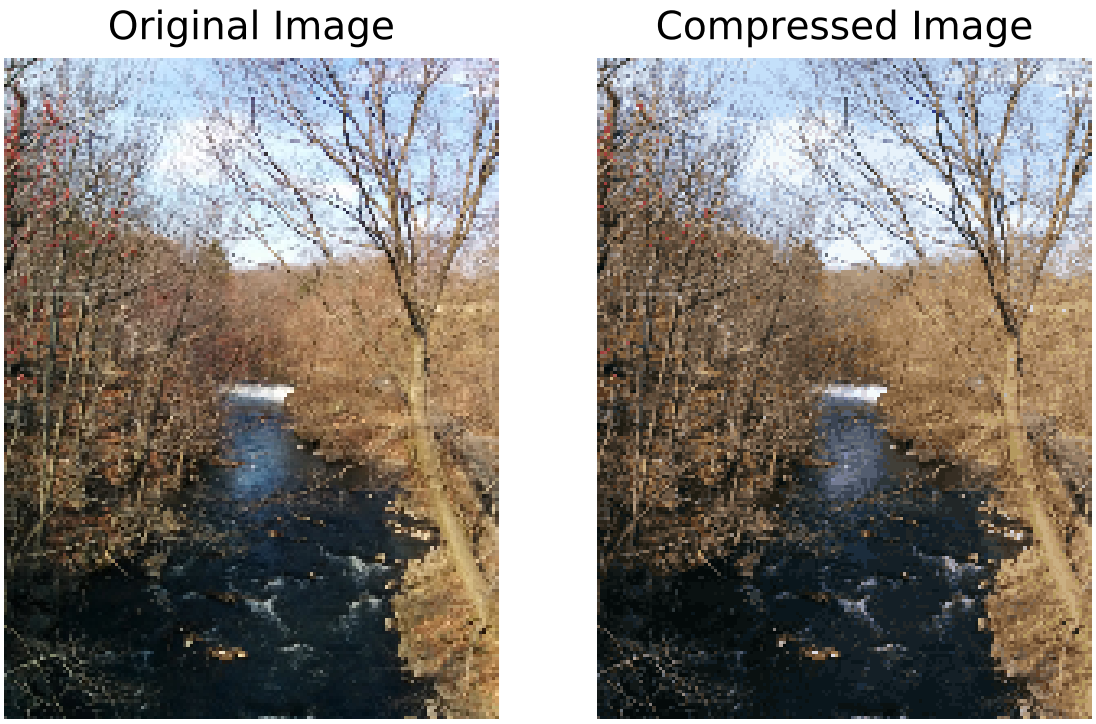

图 8.7: 利用 K 均值聚类实现图像压缩

我们从图像中提取 SURF 描述符，并将其聚类学习一个特征表示。SURF 描述符描述了一张图片感兴趣的区域，并且和图像缩放、旋转以及光照没有关系。然后，我们将使用一个向量表示一张图像，图像的每一个元素对应于描述符的一个聚类。这种方法有时也称为特征袋表示。

数据集：我们使用 Kaggle 网站中“狗 vs 猫”竞赛数据集中的 1000 张猫图像和 1000 张狗图像。该数据集可以从 Kaggle下载。我们用正类表示猫，负类表示狗。

注 这些图像有不同的尺寸。因为特征向量不表示像素，并不需要将图片调整为同样大小。

我们使用前 60% 训练，剩余图像用于测试。

```
import os
import glob
import numpy as np
import mahotas as mh
from mahotas.features import surf
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import *
from sklearn.cluster import MiniBatchKMeans
```

首先，我们加载图像，并转化为灰度图片，提取 SURF 特征。

```
all_instance_filenames = []
all_instance_targets = []

for f in glob.glob('cats-and-dogs-img/*.jpg'):
    target = 1 if 'cat' in os.path.split(f)[1] else 0
```

```
    all_instance_filenames.append(f)
    all_instance_targets.append(target)

surf_features = []
for f in all_instance_filenames:
    image = mh.imread(f, as_grey=True)
    surf_features.append(surf.surf(image)[:, 5:])

train_len = int(len(all_instance_filenames) * .60)
X_train_surf_features = np.concatenate(surf_features[:train_len])
X_test_surf_feautres = np.concatenate(surf_features[train_len:])
y_train = all_instance_targets[:train_len]
y_test = all_instance_targets[train_len:]
```

然后，我们将提取出来的描述符分配到 300 个聚类中。我们使用 MiniBatchKMeans，其收敛速度很快。

```
n_clusters = 300
estimator = MiniBatchKMeans(n_clusters=n_clusters)
estimator.fit_transform(X_train_surf_features)
```

接着，我们从训练数据和测试数据中组织特征向量，找出和每个提取的 SURF 描述符相关联的聚类，并使用 Numpy 类库 binCount 函数来计数。相关结果会将每个实例表示为一个 300 维的特征向量。

```
X_train = []
for instance in surf_features[:train_len]:
    clusters = estimator.predict(instance)
    features = np.bincount(clusters)
    if len(features) < n_clusters:
        features = np.append(features, np.zeros((1, n_clusters-len(features))))
    X_train.append(features)

X_test = []
for instance in surf_features[train_len:]:
    clusters = estimator.predict(instance)
    features = np.bincount(clusters)
    if len(features) < n_clusters:
        features = np.append(features, np.zeros((1, n_clusters-len(features))))
    X_test.append(features)
```

最后，我们在特征向量和目标上训练一个逻辑回归分类器，并计算准确率、精确率和召回率。

```
clf = LogisticRegression(C=0.001, penalty='l2')
clf.fit(X_train, y_train)
predictions = clf.predict(X_test)

print(classification_report(y_test, predictions))
```

```
             precision    recall  f1-score   support

          0       0.69      0.77      0.73       378
          1       0.77      0.69      0.72       420

avg / total       0.73      0.72      0.72       798
```