第 5 章

CHAPTER 5

TensorFlow 聚类分析

本章使用第 4 章学习的数据转化操作及聚类(clustering)技术从给定的数据中发掘有趣的模式,将数据分组。

在处理过程中,会用到两个新的工具——scikit-learn 和 matplotlib 库。其中 scikit-learn 能够生成特定结构的数据集,而 matplotlib 库可以对数据和模型作图。

5.1 无监督学习

我们把使用带有标记的训练样本进行学习的算法称为监督学习(supervised learning)。监督学习的训练样本可以统一成如下形式,其中 x 为变量,y 为标记:

$$\text{Training set}:\{(x^{(1)},y^{(1)}),(x^{(2)},y^{(2)}),\cdots,(x^{(m)},y^{(m)})\}$$

显然,现实生活中不是所有数据都带有标记(或者说标记是未知的),所以我们需要对无标记的训练样本进行学习来揭示数据的内在性质及规律,我们把这种学习称为无监督学习(unsupervised learning)。所以,无监督学习的训练样本具有如下形式,它仅包含特征量:

$$\text{Training set}:\{x^{(1)},x^{(2)},\cdots,x^{(m)}\}$$

图 5-1 形象地表示了监督学习与无监督学习的区别。图 5-1(a)表示给带标记的样本进行分类,分界线两边为不同的类(一类为圈,另一类为叉);图 5-1(b)是基于变量 x_1 和 x_2 对无标记的样本(表面上看起来都是圈)进行聚类(clustering)。

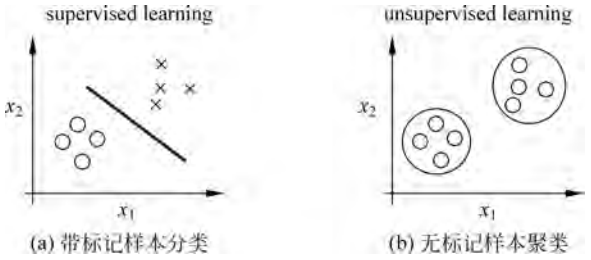

图 5-1 无监督学习与有监督学习的区别

无监督学习有很多应用,一个聚类的例子是对于收集到的论文,根据每个论文的特征量如词频、句子长、页数等进行分组。聚类还有许多其他应用,如图 5-2 所示。一个非聚类的例子是鸡尾酒会算法,即从带有噪声的数据中找到有效数据(信息),如在嘈杂的鸡尾酒会你

仍然可以注意到有人叫你，所以鸡尾酒会算法可以用于语音识别。

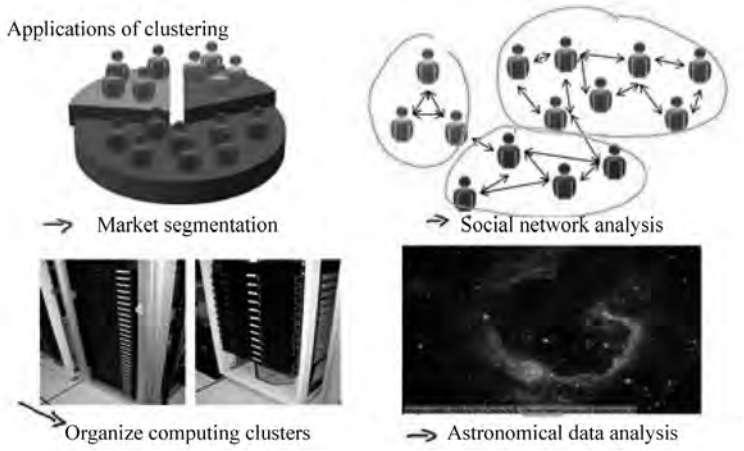

图 5-2　一些聚类的应用

5.2　聚类的概念

对于没有标签(unlabeled)的数据，我们首先能做的，就是寻找具有相同特征的数据，将它们分配到相同的组。

为此，数据集可以分成任意数量的段(segment)，其中每个段都可以用它的成员的质量中心(质心，centroid)来替代表示。

为了将不同的成员分配到相同的组中，需要定义一下，怎样表示不同元素之间的距离(distance)。在定义距离后，可以说，相对于其他的质心，每个类成员都更靠近自己所在类的质心。

数据可实现聚类需要以下几个典型要求。

(1) 可伸缩性：许多聚类算法在小于 200 个数据对象的小数据集合上工作得很好；但是，一个大规模数据库可能包含几百万个对象，在这样的大数据集合样本上进行聚类可能会导致有偏的结果。我们需要具有高度可伸缩性的聚类算法。

(2) 处理不同类型数据的能力：许多算法被设计用来聚类数值类型的数据，但应用可能要求聚类其他类型的数据，如二元类型(binary)、分类/标称类型(categorical/nominal)、序数型(ordinal)数据，或者这些数据类型的混合。

(3) 发现任意形状的聚类：许多聚类算法基于欧几里得或者曼哈顿距离度量来决定聚类。基于这样的距离度量的算法趋向于发现具有相近尺度和密度的球状簇。但是，一个簇可能是任意形状的，所以开发能发现任意形状簇的算法是很重要的。

(4) 决定输入参数的领域知识大小：许多聚类算法在聚类分析中要求用户输入一定的参数，如希望产生的簇的数目。聚类结果对于输入参数十分敏感。参数通常很难确定，特别是对于包含高维对象的数据集来说。这样不仅加重了用户的负担，也使得聚类的质量难以控制。

(5) 处理“噪声”数据的能力：绝大多数现实中的数据库都包含了孤立点、缺失或者错误的数据。一些聚类算法对于这样的数据敏感，可能导致低质量的聚类结果。

(6) 对于输入记录的顺序不敏感：一些聚类算法对于输入数据的顺序是敏感的。例如，同一个数据集合，当以不同的顺序交给同一个算法时，可能生成差别很大的聚类结果。所以开发对数据输入顺序不敏感的算法具有重要的意义。

(7) 高维度(high dimensionality)：一个数据库或者数据仓库可能包含若干维。许多聚类算法擅长处理低维的数据，可能只涉及二维或三维。人类的眼睛在最多三维的情况下能够很好地判断聚类的质量。在高维空间中聚类数据对象是非常有挑战性的，特别是考虑到这样的数据可能分布非常稀疏，而且高度偏斜。

(8) 基于约束的聚类：现实世界的应用可能需要在各种约束条件下进行聚类。假设你的工作是在一个城市中为给定数目的自动提款机选择安放位置，为了作出决定，可以对住宅区进行聚类，同时考虑如城市的河流和公路网、每个地区的客户要求等情况。要找到既满足特定的约束，又具有良好聚类特性的数据分组是一项具有挑战性的任务。

(9) 可解释性和可用性：用户希望聚类结果是可解释的、可理解的和可用的。也就是说，聚类可能需要和特定的语义解释及应用相联系。应用目标如何影响聚类方法的选择也是一个重要的研究课题。

在图 5-3 中，可以看到简单的聚类算法的输出结果。

图 5-3 简单聚类算法的输出

5.3 *k* 均值聚类算法

k 均值聚类算法是典型的基于距离的聚类算法，采用距离作为相似性的评价指标，即认为两个对象的距离越近，其相似度就越大。该算法认为簇是由距离靠近的对象组成的，因此把得到紧凑且独立的簇作为最终目标。

k 个初始聚类中心点的选取对聚类结果具有较大的影响，因为在该算法第一步中是随机地选取任意 *k* 个对象作为初始聚类中心，代表一个簇。该算法在每次迭代中对数据集中剩余的每个对象，根据其与各个簇中心的距离赋给最近的簇。当考查完所有数据对象后，一次迭代运算完成，新的聚类中心被计算出来。

5.3.1 *k* 均值聚类算法迭代判据

此方法的判据和目标是最小化簇成员到包含该成员的簇的实际质心的距离的平方的总和，这也称为惯性最小化，*k* 均值聚类算法的损失函数为：

$$\sum_{i=0}^{n} \lim_{\mu,j\in C} (\| x_j - \mu_i \|^2)$$

5.3.2 k 均值聚类算法的机制

k 均值聚类算法的机制可以由图 5-4 所示的流程图表示。

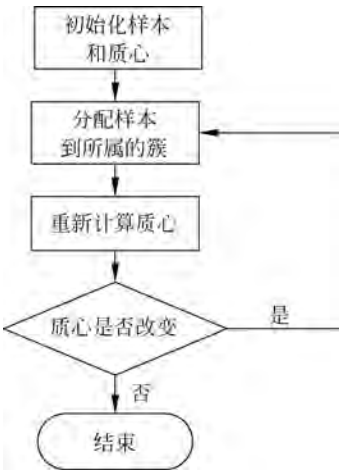

图 5-4 k 均值聚类算法简单流程图

k 均值聚类算法流程可以简化如下。

(1) 对于未分类的样本，首先随机以 k 个元素作为起始质心。为了简洁，也可以简化该算法，取元素列表中的前 k 个元素作为质心。

(2) 计算每个样本与质心的距离，并将该样本分配给距离它最近的质心所属的簇，重新计算分配好后的质心。

(3) 在质心改变后，它们的位移将引起各个距离改变，因此需要重新分配各个样本。

(4) 在停止条件满足前，不断重复步骤(2)和(3)。

我们可以使用不同类型的停止条件。

(1) 可以选择一个比较大的迭代次数 N，这样我们可能会遭遇到一些冗余的计算。也可以让 N 小一些，但是在这种情况下，如果本身质点不稳定，收敛过程慢，那么得到的结果就不能让人信服。这种停止条件也可以用作最后的手段，以防有一些非常漫长的迭代过程。

(2) 另外还有一种停止条件。如果已经没有元素从一个类转移到另一个类，意味着迭代的结束。

k 均值聚类算法示意图如图 5-5 所示。

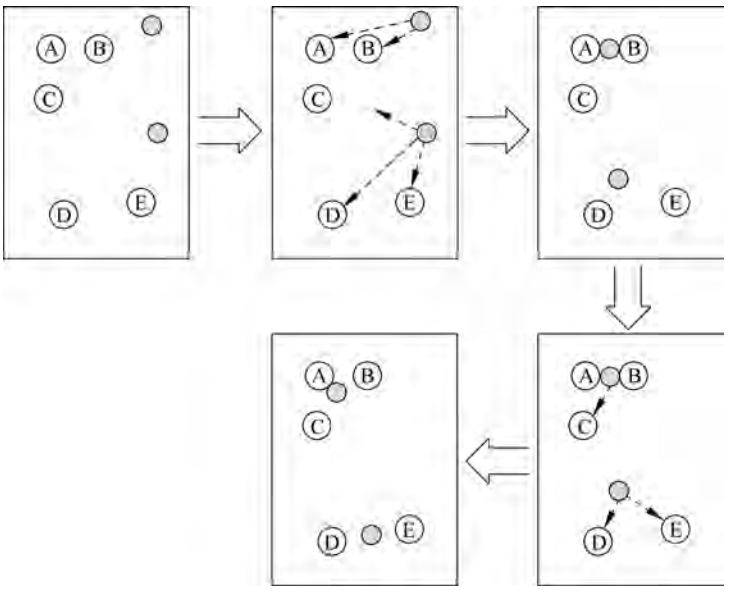

图 5-5 k 均值聚类算法示意图

5.3.3 *k* 均值聚类算法的优缺点

k 均值聚类算法的优点主要表现在：

- 扩展性很好(大部分的计算都可以并行计算)；
- 应用范围广。

但是简单是有成本的，其缺点表现在：

- 它需要先验知识(可能的聚类的数量应该预先知道)；
- 异常值影响质心的结果，因为算法并没有办法剔除异常值；
- 由于我们假设该图是凸的和各向图性的，所以对非圆状的簇，该算法表现不是很好。

5.3.4 *k* 均值聚类算法的实现

本节在实现过程中采用数据集 4k2_far.txt，聚类算法实现过程中默认的类别数量为 4。

(1) 辅助函数 myUtil.py。

```
from numpy import *
# 数据文件转矩阵
# path: 数据文件路径
# delimiter: 行内字段分隔符
def file2matrix(path, delimiter):
    fp = open(path, "rb") #读取文件内容
    content = fp.read()
    fp.close()
    rowlist = content.splitlines() #按行转换为一维表
    #逐行遍历,结果按分隔符分隔为行向量
    recordlist = [map(eval, row.split(delimiter)) for row in rowlist if row.strip()]
    #返回转换后的矩阵形式
    return mat(recordlist)
#随机生成聚类中心
def randCenters(dataSet, k):
    n = shape(dataSet)[1] #列数
    clustercents = mat(zeros((k, n))) #初始化聚类中心矩阵: k*n
    for col in xrange(n):
        mincol = min(dataSet[:, col])
        maxcol = max(dataSet[:, col])
#random.rand(k, 1):产生一个 0～1 的随机数向量,(k,1)表示产生 k 行 1 列的随机数
clustercents[:, col] = mat(mincol + float(maxcol - mincol) * random.rand(k, 1)) #按列赋值
   return clustercents
#欧式距离计算公式
def distEclud(vecA, vecB):
    return linalg.norm(vecA - vecB)
#绘制散点图
def drawScatter(plt, mydata, size=20, color='blue', mrkr='o'):
    plt.scatter(mydata.T[0], mydata.T[1], s=size, c=color, marker=mrkr)
#以不同颜色绘制数据集里的点
def color_cluster(dataindx, dataSet, plt):
```

```
    datalen = len(dataindx)
    for indx in xrange(datalen):
        if int(dataindx[indx]) == 0:
            plt.scatter(dataSet[indx, 0], dataSet[indx, 1], c = 'blue', marker = 'o')
        elif int(dataindx[indx]) == 1:
            plt.scatter(dataSet[indx, 0], dataSet[indx, 1], c = 'green', marker = 'o')
        elif int(dataindx[indx]) == 2:
            plt.scatter(dataSet[indx, 0], dataSet[indx, 1], c = 'red', marker = 'o')
        elif int(dataindx[indx]) == 3:
            plt.scatter(dataSet[indx, 0], dataSet[indx, 1], c = 'cyan', marker = 'o')
```

（2）k 均值聚类算法实现核心函数 kmeans.py。

```
from myUtil import *
def kMeans(dataSet, k):
    m = shape(dataSet)[0] #返回矩阵的行数
    #本算法核心数据结构:行数与数据集相同
    #列 1: 数据集对应的聚类中心; 列 2:数据集行向量到聚类中心的距离
    ClustDist = mat(zeros((m, 2)))
    #随机生成一个数据集的聚类中心:本例为 4 * 2 的矩阵
    #确保该聚类中心位于 min(dataSet[:,j]),max(dataSet[:,j])之间
    clustercents = randCenters(dataSet, k) #随机生成聚类中心
    flag = True #初始化标志位,迭代开始
    counter = [] #计数器
    #循环迭代直至终止条件为 False
    #算法停止的条件: dataSet 的所有向量都能找到某个聚类中心,到此中心的距离均小于
    #其他 k-1 个中心的距离
    while flag:
        flag = False #预置标志位为 False
    #1.构建 ClustDist: 遍历 DataSet 数据集,计算 DataSet 每行与聚类的最小欧式距离
    #将此结果赋值 ClustDist = [minIndex,minDist]
        for i in xrange(m):
            #遍历 k 个聚类中心,获取最短距离
            distlist = [distEclud(clustercents[j, :], dataSet[i, :]) for j in range(k)]
            minDist = min(distlist)
            minIndex = distlist.index(minDist)
            if ClustDist[i, 0] != minIndex: #找到了一个新聚类中心
                flag = True #重置标志位为 True,继续迭代
            #将 minIndex 和 minDist * * 2 赋予 ClustDist 第 i 行
            #含义是数据集 i 行对应的聚类中心为 minIndex,最短距离为 minDist
            ClustDist[i, :] = minIndex, minDist
        #2.如果执行到此处,说明还需要更改 clustercent 的值: 循环变量为 cent(0～k-1)
        #用聚类中心 cent 切分为 ClustDist,返回 dataSet 的行索引
        #并以此从 dataSet 中提取对应的行向量构成新的 ptsInClust
        #计算分隔后 ptsInClust 各列的均值,以此更新聚类中心 clustercents 的各项值
        for cent in xrange(k):
            #从 ClustDist 的第一列中筛选出等于 cent 值的行下标
            dInx = nonzero(ClustDist[:, 0].A == cent)[0]
            #从 dataSet 中提取行下标 == dInx 构成一个新数据集
            ptsInClust = dataSet[dInx]
            #计算 ptsInClust 各列的均值: mean(ptsInClust, axis = 0):axis = 0 按列计算
            clustercents[cent, :] = mean(ptsInClust, axis = 0)
    return clustercents, ClustDist
```

(3) k 均值聚类算法运行主函数 kmeans_test.py。

```
from kmeans import *
import matplotlib.pyplot as plt
dataMat = file2matrix("testData/4k2_far.txt", "\t") #从文件构建的数据集
dataSet = dataMat[:, 1:]  # 提取数据集中的特征列
k = 4  #外部指定 1,2,3...通过观察数据集有 4 个聚类中心
clustercents, ClustDist = kMeans(dataSet, k)
# 返回计算完成的聚类中心
print "clustercents:\n", clustercents
#输出生成的 ClustDist: 对应的聚类中心(列 1),到聚类中心的距离(列 2),行与 dataSet 一一对应
color_cluster(ClustDist[:, 0:1], dataSet, plt)
#绘制聚类中心
drawScatter(plt, clustercents, size = 60, color = 'red', mrkr = 'D')
plt.show()
```

在以上程序中,输出有以下几种结果。

(1) 正确的分类输出。

k 均值聚类算法输出如下,效果如图 5-6 所示。

```
clustercents:
[[ 6.99438039 5.05456275]
 [ 8.08169456 7.97506735]
 [ 3.02211698 6.00770189]
 [ 2.95832148 2.98598456]]
```

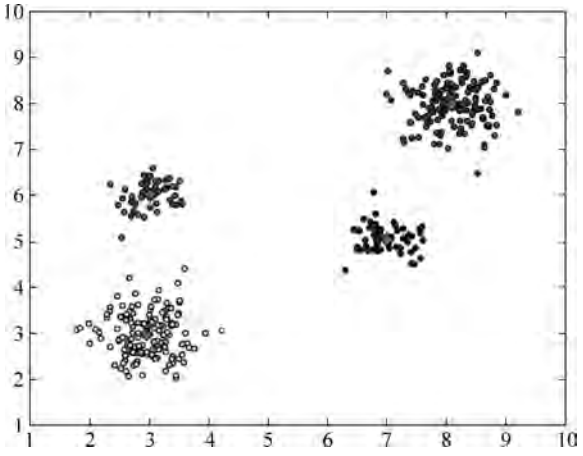

图 5-6 k 均值聚类算法正确分类结果

(2) 错误输出。

因为聚类中心是随机初始化的,k 均值聚类算法并不是总能够找到正确的聚类,下面是不能找到正确分类的情况。

情况一,局部最优收敛,输出如下,效果如图 5-7 所示。

```
clustercents:
[[ 2.9750599 3.77881139]
 [ 7.311725 5.00685 ]
 [ 6.7122963 5.09697407]
 [ 8.08169456 7.97506735]]
```

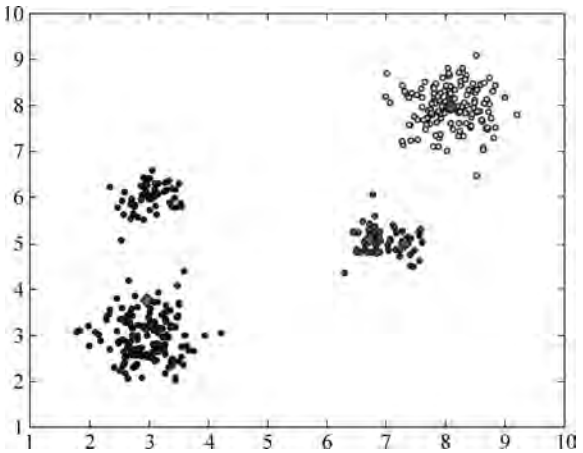

图 5-7　*k* 均值聚类算法错误分类结果 1

情况二，只收敛到三个聚类中心，输出如下，效果如图 5-8 所示。

```
clustercents:
[[ 6.99438039  5.05456275]
 [ 2.9750599  3.77881139]
 [ 8.08169456  7.97506735]
 [        nan         nan]]
```

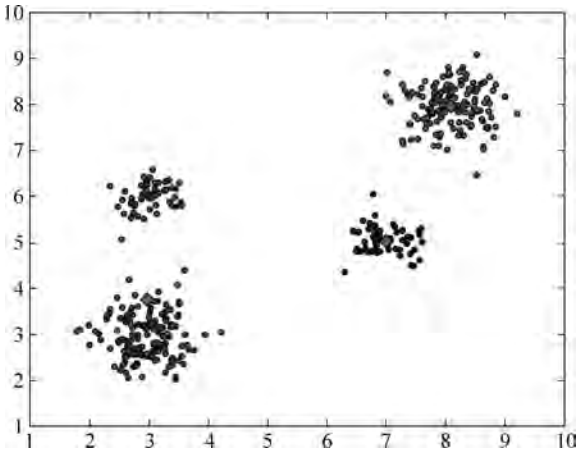

图 5-8　*k* 均值聚类算法错误分类结果 2

5.4 k 最近邻算法

k 最近邻(k-Nearest Neighbor,kNN)算法是一个理论上比较成熟的方法,也是最简单的机器学习算法之一。该方法的思路是:如果一个样本在特征空间中的 k 个最相似(即特征空间中最邻近)的样本中的大多数属于某一个类别,则该样本也属于这个类别。其流程图如图 5-9 所示。

5.4.1 实例分析

如图 5-10 所示,有两类不同的样本数据,分别用蓝色的小正方形和红色的小三角形表示,而图中正中间的那个绿色的圆所表示的数据则是待分类的数据。也就是说,现在我们不知道中间那个绿色的数据是从属于哪一类,下面我们就要解决这个问题:给这个绿色的圆分类。

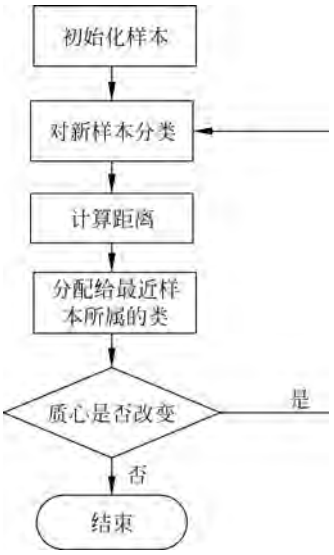

图 5-9 k 最近邻算法流程图

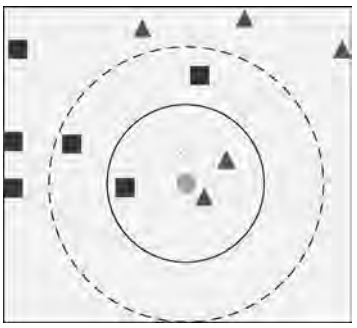

图 5-10 两类不同的样本数据(见彩插)

人们常说,物以类聚,人以群分,判别一个人是一个什么样品质特征的人,常常可以从他/她身边的朋友入手,所谓观其友而识其人。我们要判别上图中那个绿色的圆是属于哪一类数据,即要从它的邻居下手。但一次性看多少个邻居呢?从图 5-10 中还能看到:

(1) 如果 $k=3$,绿色圆点的最近的 3 个邻居是 2 个红色小三角形和 1 个蓝色小正方形,少数从属于多数,基于统计的方法,判定绿色的这个待分类点属于红色的三角形一类。

(2) 如果 $k=5$,绿色圆点的最近的 5 个邻居是 2 个红色三角形和 3 个蓝色的正方形,还是少数从属于多数,基于统计的方法,判定绿色的这个待分类点属于蓝色的正方形一类。

于此我们看到,当无法判定当前待分类点是从属于已知分类中的哪一类时,我们可以依据统计学的理论看它所处的位置特征,衡量它周围邻居的权重,而把它归(或分配)到权重更大的那一类。这就是 k 最近邻算法的核心思想。

5.4.2 k 最近邻算法概述

kNN是一种基本的分类和回归方法。kNN的输入是测试数据和训练样本的数据集，输出是测试样本的类别。kNN没有显式的训练过程，在测试时，计算测试样本和所有训练样本的距离，根据最近的 k 个训练样本的类别，通过多数投票的方式进行预测。算法描述如下。

输入：训练数据集 $T=\{(x_1,y_1),(x_2,y_2),\cdots,(x_n,y_n)\}$ 和测试数据 x。其中 $x_i=R^n$，$y_i=\{c_1,c_2,\cdots,c_K\}$。

输出：实例 x 所属的类别。

(1) 根据给定的距离度量，在训练集 T 中找到与 x 距离最近的 k 个样本，涵盖这 k 个点的 x 邻域记作 $N_k(x)$。

(2) 在 $N_k(x)$ 中根据分类规则(如多数表决)确定 x 的类别 y：

$$y=\text{argmax}_j \sum_{x_i\in N_k(k)} I\{(y_i=c_j),i=1,2,\cdots,n;j=1,2,\cdots,k\}$$

5.4.3 模型和三要素

从kNN的算法描述中可以发现，有三个元素很重要，分别是距离度量、k 的大小和分类规则，这便是kNN模型的三要素。在kNN中，当训练数据集和三要素确定后，相当于将特征空间划分成一些子空间，对于每个训练实例 x_i，距离该点比距离其他点更近的所有点组成了一个区域，每个区域的类别由决策规则确定且唯一，从而将整个区域划分。对于任何一个测试点，找到其所属的子空间，其类别即为该子空间的类别。

1. 距离度量

距离度量有很多方式，要根据具体情况选择合适的距离度量方式。常用的是闵可夫斯基距离(Minkowski Distance)，定义为：

$$D(x,y)=\left(\sum_{i=1}^{m}|x_i-y_i|^p\right)^{\frac{1}{p}}$$

其中，$p\geqslant 1$。当 $p=2$ 时，是欧氏距离；当 $p=1$ 时，是曼哈顿距离。

2. k 的选择

k 的选择会对算法的结果产生重大影响。

如果 k 值较小，就相当于用较小邻域中的训练实例进行预测，极端情况下 $k=1$，测试实例只和最接近的一个样本有关，训练误差很小(0)，但是如果这个样本恰好是噪声，预测就会出错，测试误差很大。也就是说，当 k 值较小的，会产生过拟合的现象。

如果 k 值较大，就相当于用很大邻域中的训练实例进行预测，极端情况是 $k=n$，测试实例的结果是训练数据集中实例最多的类，这样会产生欠拟合。

在应用中，一般选择较小的 k 值且 k 是奇数。通常采用交叉验证的方法来选取合适的 k 值。

3. 分类规则

kNN的分类决策规则通常是多数表决，即由测试样本的 k 个临近样本的多数类决定测

试样本的类别。多数表决规则有如下解释：给定测试样本 x，其最邻近的 k 个训练实例构成集合 $N_k(x)$，分类损失函数为 0-1 损失。如果涵盖 $N_k(x)$ 区域的类别为 c_j，则分类误差率为：

$$\frac{1}{k}\sum_{x_i \in N_k(x)} I\{y_i \neq c_j\} = 1 - \frac{1}{k}\sum_{x_i \in N_k(x)} I\{y_i = c_j\}$$

分类误差率小即经验风险小，所以多数表决等价于经验风险最小化。而 kNN 的模型相当于对任意的 x 得到 $N_k(x)$，损失函数是 0-1 损失，优化策略是经验风险最小。总的来说就是对 $N_k(x)$ 中的样本应用多数表决。

5.4.4 kNN 算法的不足

kNN 算法不仅可以用于分类，还可以用于回归。通过找出一个样本的 k 个最近邻居，将这些邻居的属性的平均值赋给该样本，就可以得到该样本的属性。更有用的方法是将不同距离的邻居对该样本产生的影响给予不同的权值(weight)，如权值与距离成反比。

该算法在分类时的一个主要的不足就是，当样本不平衡时，如一个类的样本容量很大，而其他类样本容量很小时，有可能导致当输入一个新样本时，该样本的 k 个邻居中大容量类的样本占多数。该算法只计算“最近的”邻居样本，某一类的样本数量很大，那么或者这类样本并不接近目标样本，或者这类样本很靠近目标样本。无论怎样，数量并不能影响运行结果。这时可以采用权值的方法(和该样本距离小的邻居权值大)来改进。

该方法的另一个不足之处是计算量较大，因为对每一个待分类的文本都要计算它到全体已知样本的距离，才能求得它的 k 个最近邻点。目前常用的解决方法是事先对已知样本点进行剪辑，事先去除对分类作用不大的样本。该算法比较适用于样本容量比较大的类域的自动分类，而那些样本容量较小的类域采用这种算法比较容易产生误分。

实现 k 近邻算法时，主要考虑的问题是如何对训练数据进行快速 k 近邻搜索，这在特征空间维数大及训练数据容量大时非常必要。

5.5 k 均值聚类算法的典型应用

前面章节对 k 均值聚类算法与 k 最近邻算法进行了介绍，下面通过实例来演示这两种方法的典型应用。

5.5.1 实例：对人工数据集使用 k 均值聚类算法

1. 数据集加载

实例中使用的是人工数据集，它的生成方式为：

```
centers = [(-2, -2), (-2, 1.5), (1.5, -2), (2, 1.5)]
data, features = make_blobs (n_samples = 200, centers = centers, n_features = 2, cluster_std
 = 0.8, shuffle = False, random_state = 42)
```

通过 matplotlib 绘制该数据集：

```
ax.scatter(np.asarray(centers).transpose()[0], np.asarray(centers).transpose()[1], marker =
'o', s = 250)
plt.plot()
```

得到的最终结果如图 5-11 所示。

2. 模型架构

Points 变量用来存放数据集的点的坐标，centroids 变量用于存放每个组质心的坐标，clustering_assignments 变量用来存放为每个数据元素分配的类的索引。

例如，clustering_assignments[2]=1 表示数据 data[2]的数据点被分配到 1 类，而 1 类的质心坐标通过访问 centroids[1]得到。

```
points = tf.Variable(data)
cluster_assignments = tf.Variable(tf.zeros([N], dtype = tf.int64))
centroids = tf.Variable(tf.slice(points.initialized_value(), [0,0], [K,2]))
```

然后，可以通过 matplotlib()库绘制出质心的位置：

```
fig, ax = plt.subplots()
ax.scatter(sess.run(points).transpose()[0], sess.run(points).transpose()[1], marker = 'o',
s = 200, c = assignments, cmap = plt.cm.coolwarm)
plt.show()
```

最终得到如图 5-12 所示的结果。

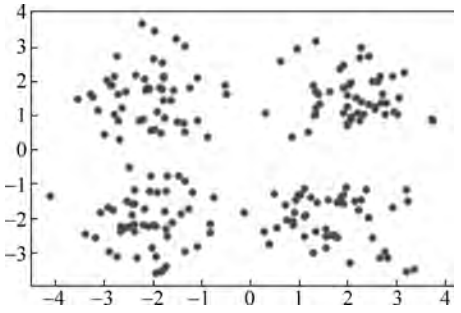

图 5-11　块状数据集散点图

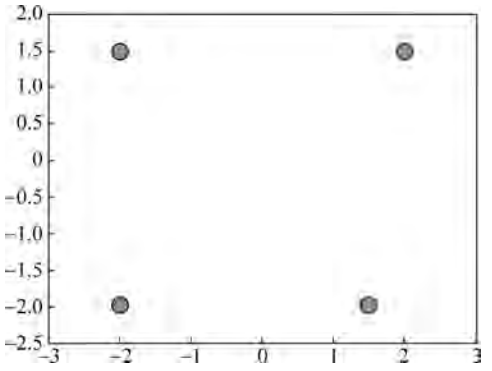

图 5-12　中心点的位置

3. 损失函数与优化循环

接着，对所有的质心做 N 次复制，对每个样本点做 K 次复制，这样样本点和质心的形状都是 $N \times K \times 2$，就可以计算每一个样本到每一个质心点之间在所有维度上的距离。

```
rep_centroids = tf.reshape(tf.tile(centroids, [N, 1]), [N, K, 2])
rep_points = tf.reshape(tf.tile(points, [1, K]), [N, K, 2])
sum_squares = tf.reduce_sum(tf.square(rep_points - rep_centroids), reduction_indices = 2)
```

然后，对所有维度求和，得到和最小的那个索引（这个索引就是每个点所属的新的类）：

```
best_centroids = tf.argmin(sum_squares, 1)
```

centroids 也会在每个迭代之后由 bucket_mean 函数更新，具体可查看完整的源代码。

4. 停止条件

实例的停止条件是所有的质心不再变化：

```
did_assignments_change = tf.reduce_any(tf.not_equal(best_centroids, cluster_assignments))
```

此处，使用 control_dependencies 来控制是否更新质心：

```
with tf.control_dependencies([did_assignments_change]):
    do_updates = tf.group(
    centroids.assign(means),
    cluster_assignments.assign(best_centroids))
```

运行程序，得到如图 5-13 所示的输出。

```
Found in 407.12 seconds 8 iterations
Centroids:
[[ 1.65289262 -2.04643427]
 [-2.0763623   1.61204964]
 [-2.08862822 -2.07255306]
 [ 2.09831502  1.55936014]]
Cluster assignments: [2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
```

图 5-13 k 均值聚类算法运行结果

图 5-14 是不同迭代中质心的变化。

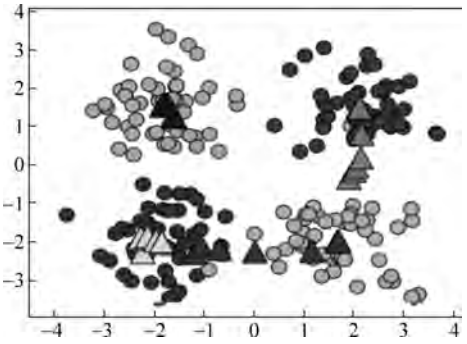

图 5-14 不同迭代中质心的变化（见彩插）

5. 完整的源代码

实现对人工数据集使用 k 均值聚类算法的完整源代码为：

```
import tensorflow as tf
import numpy as np
```

```
import time
import matplotlib
import matplotlib.pyplot as plt
from sklearn.datasets.samples_generator import make_blobs
from sklearn.datasets.samples_generator import make_circles

DATA_TYPE = 'blobs'
N = 200
#集群的数量,如果我们选择圆圈,只有 2 个就足够了
if (DATA_TYPE == 'circle'):
    K = 2
else:
    K = 4
#判断条件是否满足最大迭代次数
MAX_ITERS = 1000
start = time.time()
centers = [(-2, -2), (-2, 1.5), (1.5, -2), (2, 1.5)]
if (DATA_TYPE == 'circle'):
    data, features = make_circles(n_samples = 200, shuffle = True, noise = 0.01, factor = 0.4)
else:
    data, features = make_blobs (n_samples = 200, centers = centers,
n_features = 2, cluster_std = 0.8, shuffle = False, random_state = 42)
fig, ax = plt.subplots()
ax.scatter(np.asarray(centers).transpose()[0], np.asarray(centers).transpose()[1],
marker = 'o', s = 250)
plt.show()
fig, ax = plt.subplots()
if (DATA_TYPE == 'blobs'):
    ax.scatter(np.asarray(centers).transpose()[0], np.asarray(centers).transpose()[1],
marker = 'o', s = 250)
    ax.scatter(data.transpose()[0], data.transpose()[1], marker = 'o',
s = 100, c = features, cmap = plt.cm.coolwarm)
    plt.show()
points = tf.Variable(data)
cluster_assignments = tf.Variable(tf.zeros([N], dtype = tf.int64))
centroids = tf.Variable(tf.slice(points.initialized_value(), [0,0], [K,2]))
sess = tf.Session()
sess.run(tf.initialize_all_variables())
sess.run(centroids)
rep_centroids = tf.reshape(tf.tile(centroids, [N, 1]), [N, K, 2])
rep_points = tf.reshape(tf.tile(points, [1, K]), [N, K, 2])
sum_squares = tf.reduce_sum(tf.square(rep_points - rep_centroids),
reduction_indices = 2)
best_centroids = tf.argmin(sum_squares, 1)
did_assignments_change =
tf.reduce_any(tf.not_equal(best_centroids, cluster_assignments))
def bucket_mean(data, bucket_ids, num_buckets):
    total = tf.unsorted_segment_sum(data, bucket_ids, num_buckets)
    count = tf.unsorted_segment_sum(tf.ones_like(data), bucket_ids, num_buckets)
    return total / count
means = bucket_mean(points, best_centroids, K)
with tf.control_dependencies([did_assignments_change]):
    do_updates = tf.group(
```

```
        centroids.assign(means),
        cluster_assignments.assign(best_centroids))
    changed = True
    iters = 0
    fig, ax = plt.subplots()
    if (DATA_TYPE == 'blobs'):
        colourindexes = [2,1,4,3]
    else:
        colourindexes = [2,1]
    while changed and iters < MAX_ITERS:
        fig, ax = plt.subplots()
        iters += 1
        [changed, _] = sess.run([did_assignments_change, do_updates])
        [centers, assignments] = sess.run([centroids, cluster_assignments])
        ax.scatter(sess.run(points).transpose()[0], sess.run(points).transpose()[1],
    marker = 'o', s = 200, c = assignments, cmap = plt.cm.coolwarm)
        ax.scatter(centers[:,0],centers[:,1], marker = '^', s = 550,
        c = colourindexes, cmap = plt.cm.plasma)
        ax.set_title('Iteration ' + str(iters))
        plt.savefig("kmeans" + str(iters) + ".png")
    ax.scatter(sess.run(points).transpose()[0], sess.run(points).transpose()[1],
    marker = 'o', s = 200, c = assignments, cmap = plt.cm.coolwarm)
    plt.show()
    end = time.time()
    print("Found in %.2f seconds" % (end - start)), iters, "iterations"
    print("Centroids:")
    print(centers)
    print("Cluster assignments:", assignments)
```

5.5.2 实例：对人工数据集使用 k 最近邻算法

本实例使用的数据集是 k 均值聚类算法不能正确分类的数据集。

1. 数据集生成

实例中的数据集与 5.5.1 节一样，还是两类，但这次会加大数据的噪声(从 0.0 到 0.12)：

```
data, features = make_circles(n_samples = N, shuffle = True, noise = 0.12, factor = 0.4)
```

训练数据集的数据绘制如图 5-15 所示。

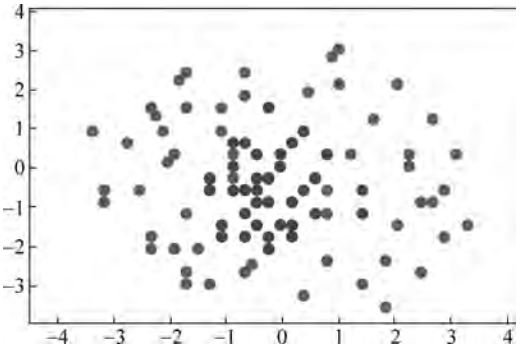

图 5-15 生成环状数据集

2. 模型结构

此处的变量除了存放原始数据外，还有一个列表，用来存放为每个测试数据预测的测试结果。

3. 损失函数

在聚类问题中，使用的距离描述跟前面一样，都是欧几里得距离。在每一个聚类的循环中，计算测试点与每个存在的训练点之

间的距离，找到最接近那个训练点的索引，使用该索引寻找最近邻的点的类：

```
distances = tf.reduce_sum(tf.square(tf.sub(i , tr_data)),reduction_indices = 1)
neighbor = tf.arg_min(distances,0)
```

4. 停止条件

实例中，当处理完测试集中所有的样本后，整个过程结束。

图 5-16 为 kNN 算法的结果。从图中可以看出，至少在有限数据集的范围内，该算法比无重叠、块状优化、k 均值聚类算法的效果好。

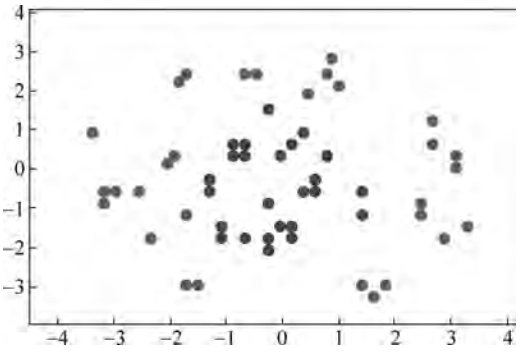

图 5-16　kNN 对环状数据集的结果

5. 完整的源代码

实现对人工数据集使用 k 最近邻算法的完整源代码为：

```
import tensorflow as tf
import numpy as np
import time
import matplotlib
import matplotlib.pyplot as plt
from sklearn.datasets.samples_generator import make_circles

N = 210
K = 2
# 判断条件是否满足最大迭代次数
MAX_ITERS = 1000
cut = int(N * 0.7)
start = time.time()
data, features = make_circles(n_samples = N, shuffle = True, noise = 0.12, factor = 0.4)
tr_data, tr_features = data[:cut], features[:cut]
te_data,te_features = data[cut:], features[cut:]
fig, ax = plt.subplots()
ax.scatter(tr_data.transpose()[0], tr_data.transpose()[1], marker = 'o',
                s = 100, c = tr_features, cmap = plt.cm.coolwarm)
plt.plot()
points = tf.Variable(data)
cluster_assignments = tf.Variable(tf.zeros([N], dtype = tf.int64))
sess = tf.Session()
sess.run(tf.initialize_all_variables())
test = []

for i, j in zip(te_data, te_features):
    distances = tf.reduce_sum(tf.square(tf.sub(i , tr_data)),reduction_indices = 1)
    neighbor = tf.arg_min(distances,0)
    #print tr_features[sess.run(neighbor)]
```

```
        #print j
        test.append(tr_features[sess.run(neighbor)])
    print test
    fig, ax = plt.subplots()
    ax.scatter(te_data.transpose()[0], te_data.transpose()[1], marker = 'o',
               s = 100, c = test, cmap = plt.cm.coolwarm)
    plt.plot()
    #rep_points_v = tf.reshape(points, [1, N, 2])
    #rep_points_h = tf.reshape(points, [N, 2])
    #sum_squares = tf.reduce_sum(tf.square(rep_points - rep_points), reduction_indices = 2)
    #print(sess.run(tf.square(rep_points_v - rep_points_h)))
    end = time.time()
    print("Found in %.2f seconds" % (end - start))
    print("Cluster assignments:", test)
```

5.5.3 实例：对图像识别使用 k 最近邻算法

k 最近邻算法也常用于图像识别。图像识别的“Hello World”数据集是 MNIST 手写数字样本数据集。MNIST 手写数字样本数据集由上万张 28×28 像素、已标注的图片组成。虽然该数据集不大，但是其包含有 784 个特征可供最近邻算法训练。我们将计算这类分类问题的最近邻预测，选用最近 k 邻域（实例中，$k=4$）模型。

1. 导入库

要实现图像识别，首先要导入必要的编程库。注意，导入 PIL（Python Image Library）模块绘制预测输出结果。TensorFlow 中有内建的函数加载 MNIST 手写数字样本数据集，实现代码为：

```
import random
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from PIL import Image
from tensorflow.examples.tutorials.mnist import input_data
from tensorflow.python.framework import ops
ops.reset_default_graph()
```

2. 创建图会话

接着，创建一个计算图会话，加载 MNIST 手写数字样本数据集，并指定 one-hot 编码，实现代码为：

```
sess = tf.Session()
mnist = input_data.read_data_sets("MNIST_data/", one_hot = True)
```

注意：one-hot 编码是分类类别的数值化，这样更利于后续的数值计算。实例包含 10 个类别（数字 0 到 9），采用长度为 10 的 0-1 向量表示。例如，类别“0”表示为向量：1,0,0,0,0,0,0,

0,0,0,0,类别“1”表示为向量：0,1,0,0,0,0,0,0,0,0。

3. 抽样处理

由于 MNIST 手写数字样本数据集较大，直接计算成千上万个输入的 784 个特征之间的距离是比较困难的，所以实例会抽样成小数据集进行训练。对测试集也进行抽样处理，为了后续绘图方便，选择测试集量可以被 6 整除。将绘制最后批次的 6 张图片来查看效果。代码为：

```
train_size = 1000
test_size = 102
rand_train_indices = np.random.choice(len(mnist.train.images), train_size, replace = False)
rand_test_indices = np.random.choice(len(mnist.test.images), test_size, replace = False)
x_vals_train = mnist.train.images[rand_train_indices]
x_vals_test = mnist.test.images[rand_test_indices]
y_vals_train = mnist.train.labels[rand_train_indices]
y_vals_test = mnist.test.labels[rand_test_indices]
```

4. 声明 k 值与批量

根据需要，声明 k 的值和批量的大小，代码为：

```
k = 4
batch_size = 6
```

5. 初始化定位符

根据需要，在计算图中开始初始化占位符，并赋值，实现代码为：

```
x_data_train = tf.placeholder(shape = [None, 784], dtype = tf.float32)
x_data_test = tf.placeholder(shape = [None, 784], dtype = tf.float32)
y_target_train = tf.placeholder(shape = [None, 10], dtype = tf.float32)
y_target_test = tf.placeholder(shape = [None, 10], dtype = tf.float32)
```

6. 声明距离度量

根据需要，声明距离度量函数。实例中使用 L1 范数(即绝对值)作为距离函数，实现代码为：

```
distance = tf.reduce_sum(tf.abs(tf.subtract(x_data_train, tf.expand_dims(x_data_test,
1))), axis = 2)
```

也可以把距离函数定义为 L2 范数。对应的代码为：

```
distance = tf.sqrt(tf.reduce_sum(tf.square(tf.subtract(x_data_train, tf.expand_dims(x_data
_test,1))), reduction_indices = 1))
```

7. 预测模型

根据需要，找到最接近的 top k 图片和预测模型。在数据集的 one-hot 编码索引上进行

预测模型计算，然后统计发生的数量，实现代码为：

```
top_k_xvals, top_k_indices = tf.nn.top_k(tf.negative(distance), k=k)
prediction_indices = tf.gather(y_target_train, top_k_indices)
# 预测模式类别
count_of_predictions = tf.reduce_sum(prediction_indices, axis=1)
prediction = tf.argmax(count_of_predictions, axis=1)
```

8. 计算预测值

在测试集上遍历迭代运行，计算预测值，并将结果存储，实现代码为：

```
num_loops = int(np.ceil(len(x_vals_test)/batch_size))
test_output = []
actual_vals = []
for i in range(num_loops):
    min_index = i*batch_size
    max_index = min((i+1)*batch_size,len(x_vals_train))
    x_batch = x_vals_test[min_index:max_index]
    y_batch = y_vals_test[min_index:max_index]
    predictions = sess.run(prediction, feed_dict={x_data_train: x_vals_train,
x_data_test: x_batch, y_target_train: y_vals_train, y_target_test: y_batch})
    test_output.extend(predictions)
    actual_vals.extend(np.argmax(y_batch, axis=1))
```

9. 计算模型训练准确度

现在已经保存了实际值和预测返回值，下面计算模型训练准确度。不过该结果会因为测试数据集和训练数据集的随机抽样而变化，但是其准确度为 80%～90%。实现代码为：

```
accuracy = sum([1./test_size for i in range(test_size) if test_output[i]==actual_vals[i]])
print('Accuracy on test set: ' + str(accuracy))
```

10. 绘制结果

绘制最后批次的计算结果，效果如图 5-17 所示，实现代码为：

```
actuals = np.argmax(y_batch, axis=1)
Nrows = 2
Ncols = 3
for i in range(len(actuals)):
    plt.subplot(Nrows, Ncols, i+1)
    plt.imshow(np.reshape(x_batch[i], [28,28]), cmap='Greys_r')
    plt.title('Actual: ' + str(actuals[i]) + 'Pred: ' + str(predictions[i]),
                                            fontsize=10)
    frame = plt.gca()
    frame.axes.get_xaxis().set_visible(False)
    frame.axes.get_yaxis().set_visible(False)
plt.show()
```

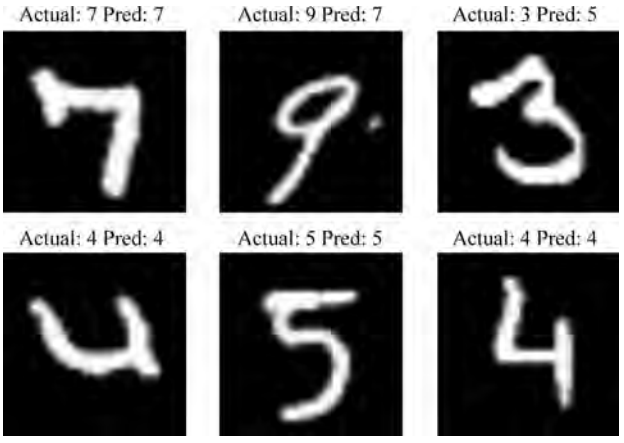

图 5-17　最近邻算法预测的最后批次的 6 张图片

5.6　小结

聚类就是对大量未知标注的数据集，按数据的内在相似性将数据集划分为多个类别，使类别内的数据相似度较大而类别间的数据相似度较小。本章主要介绍了无监督学习、聚类的概念、k 均值聚类算法、k 最近邻算法等内容，让读者全面了解聚类，并且利用 TensorFlow 实现相应聚类。最后，5.5 节介绍 k 均值聚类算法的典型应用，总结性地介绍聚类在人工智能中的应用。

5.7　习题

1. 什么是无监督学习？
2. 数据实现聚类需要哪几个典型要求？
3. k 均值聚类算法的优缺点主要表现在哪些方面？
4. kNN 模型的三要素是什么？
5. kNN 算法的不足主要表现在哪些方面？