第 3 章　机器学习基础实践

机器学习是人工智能领域内的一个重要分支，旨在通过计算的手段，利用经验来改善计算机系统的性能，这里的经验通常指历史数据。机器学习的原理是从大量的数据中抽象出一个算法模型，然后将新数据输入到模型中，得到模型对其的判断（例如类型、预测实数值等），也就是说，机器学习是一门主要研究学习算法的学科。

3.1　实践一：基于线性回归/Lasso 回归/多项式回归实现房价预测

基于线性回归-Lasso 回归-多项式回归实现房价预测

回归算法是机器学习领域一个非常经典的学习算法，主要用于对输入自变量产生一个对应的输出因变量值，通常，因变量为实数范围内的数值类型数据。在形式上，对于一个点集，用一条曲线去拟合其分布的过程，就叫作回归。线性回归算法是最简单的回归算法，其表达形式为 $y=\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}+b$，$\boldsymbol{w}$ 即为所学习的参数，$\boldsymbol{x}$、y 分别为自变量与因变量，在机器学习任务中，称之为输入特征与输出结果。线性回归算法是指自变量之间通过一个线性组合便可得到因变量的预测结果的算法，对于一些线性可分的数据集，可以尝试使用线性回归模型进行建模。

在机器学习建模的过程中，通常会出现两种情况，一种是欠拟合，另一种是过拟合。欠拟合是指模型在训练数据和测试数据上都不能很好地拟合数据的特征，常出现在模型较为简单，但是数据的分布形式较复杂的情况。对于欠拟合，可以通过增加模型的复杂度来解决。过拟合是指模型在训练数据上可以很好地拟合数据的特征，但是却无法很好地拟合测试数据的特征，常出现在模型较为复杂，但是数据的分布形式较简单的情况。对于过拟合，可以通过使用正则化的策略来解决。常用的正则化策略包括 L1 正则和 L2 正则。L1 正则限制模型各个参数的绝对值之和，倾向于产生稀疏的特征。L2 正则限制模型各个参数的平方和的开方值，倾向于使得各个特征趋近于 0。

岭回归算法可以看作是带有 L2 正则的线性回归算法，其是一种专用于共线性数据分析的有偏估计回归方法，实质上是一种改良的最小二乘估计法，通过放弃最小二乘法的无偏性，以损失部分信息、降低精度为代价获得回归系数更为符合实际、更可靠的回归方法，对病态数据的拟合要强于最小二乘法。

Lasso 回归算法可以看作是带有 L1 正则的线性回归算法，其是一种压缩估计，通过构造一个惩罚函数得到一个较为精炼的模型，使得它压缩一些回归系数，即强制系数绝对值之和小于某个固定值，同时设定一些回归系数为零。因此，Lasso 回归算法保留了子集收缩的优点，是一种处理具有复共线性数据的有偏估计。

多项式回归算法是线性回归算法的一种扩展。在一些场景中，使用直线无法拟合全部的数据，则需要使用高阶的曲线进行拟合，如二次模型等，而多项式回归算法能够对一些更复杂的、非线性可分的数据集进行建模。具体来说，在多项式回归算法中，引入了特征的更高次方，例如平方项或立方项，通过增加模型的自由度来捕获数据中非线性的变化。

回归任务最常用的性能度量方式为均方误差，即计算真实值与预测值之间的差平方的均值，也就是真实值与预测值之间的欧氏距离，最小化该值可以使预测误差尽可能小，并且对均方误差值的优化是一个凸优化过程（二次损失函数，可以求得最小值），可以使用最小二乘法对模型进行求解，使得所有样本到所拟合曲线上的距离之和最小。

本书就回归算法模型进行代码演示，在波士顿房价数据集上进行建模，对于模型未见过的数据，使用建模的回归模型预测其房价。该建模过程主要分为以下四个步骤：数据加载、模型配置、模型训练、模型评估。本节实践的平台为 AI Studio，实践环境为 Python 3.7。

步骤 1：数据加载

获取数据集：直接通过 sklearn 的 datasets 模块获取波士顿房价数据集，并打印数据集的 shape。其中：boston.data 为获取数据集的特征信息部分，boston.target 为获取数据集的房价标签部分。

```
# 加载相关包
import numpy as np
import os
import matplotlib
import matplotlib.pyplot as plt
import pandas as pd

from sklearn import datasets, linear_model
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_squared_error


boston = sklearn.datasets.load_boston()

print(boston.data.shape)
print(boston.target.shape)

print(boston.feature_names)
print(boston.DESCR)
```

输出结果如下所示：

```
(506, 13)
(506,)
['CRIM' 'ZN' 'INDUS' 'CHAS' 'NOX' 'RM' 'AGE' 'DIS' 'RAD' 'TAX' 'PTRATIO'
'B' 'LSTAT']
```

通过代码输出可以看出，该数据集包含 506 条数据，每条数据包含 13 个输入变量和 1

个输出变量，输入变量包含房屋以及房屋周围的详细信息，例如城镇犯罪率、一氧化氮浓度、住宅平均房间数、到中心区域的加权距离以及自住房平均房价等。

步骤 2：数据预处理

对于下载的数据集，由于该数据集中原始的特征尺度不一，因此首先需要对原始数据进行归一化操作，方可进行后续的模型训练。本实践使用 sklearn. preprocessing. StandardScaler()函数进行归一化处理，其通过去除均值和缩放为单位变量实现特征标准化。在完成归一化后，我们将数据集切分为训练集与测试集两个子集以进行后续的训练过程。

```
x = boston.data
y = boston.target

ss = StandardScaler()
x = ss.fit_transform(x)

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.2, random_state = 3)
```

之后我们调用 pandas 库函数，实现清晰的数据展示。

```
bos = pd.DataFrame(x_train)
print(bos.head())
```

输出结果如图 3-1-1 所示，可以看出数据已经进行了归一化的处理。

```
          0         1         2         3         4         5         6  \
0  0.686614 -0.487722  1.015999 -0.272599  1.367490  0.631645  0.907687
1  0.049445 -0.487722  1.015999 -0.272599 -0.196047 -0.079260  0.786781
2 -0.416206  0.370669 -1.139082 -0.272599 -0.965723  0.973563 -1.115709
3  1.327804 -0.487722  1.015999 -0.272599  0.512296 -1.397069  1.021481
4 -0.405253 -0.487722 -0.375976 -0.272599 -0.299707 -0.224575  0.591198

          7         8         9        10        11        12
0 -0.617477  1.661245  1.530926  0.806576 -3.837460  0.849024
1 -0.330735  1.661245  1.530926  0.806576  0.423838  0.030409
2  0.689122 -0.523001 -1.141751 -1.643945  0.389848 -1.130230
3 -0.805438  1.661245  1.530926  0.806576 -0.078878  1.718101
4 -0.795123 -0.523001 -0.143951  1.130230  0.340070  0.201421
```

图 3-1-1　预处理后的波士顿房价数据集

步骤 3：模型配置

本实践调用 sklearn. linear_model 类实现线性回归、Lasso 回归和多项式回归算法，首先实例化模型，之后调用 fit()函数训练模型。模型训练结束后，根据训练好的模型，在测试数据上调用 score()函数、mean_squared_error()函数计算均方根误差进行评估。

(1) 基于线性回归的波士顿房价预测。

```
# 线性回归:
def li_model():
    model = linear_model.LinearRegression()
    return model
```

```
model1 = li_model()
model1.fit(x_train, y_train)
train_score1 = model1.score(x_train, y_train)
test_score1 = model1.score(x_test, y_test)
print("训练集上的 train_score: ", train_score1)
print("测试集上的 train_score: ", test_score1)

y_pred1 = model1.predict(x_test)
rmse1 = (np.sqrt(mean_squared_error(y_test, y_pred1)))
print("测试集上的 rmse: ", rmse1)
```

输出结果如下所示：

```
训练集上的 train_score:  0.7239410298290112
测试集上的 train_score:  0.7952617563243858
测试集上的 rmse:  4.116196425564963
```

（2）基于 Lasso 回归的波士顿房价预测。

```
# 岭回归:
def la_model():
    model = linear_model.LassoCV()
    return model

model2 = la_model()
model2.fit(x_train, y_train)

# Lasso 系数
print(model2.alpha_)
# 相关系数
print(model2.coef_)

train_score2 = model2.score(x_train, y_train)
test_score2 = model2.score(x_test, y_test)
print("训练集上的 train_score: ", train_score2)
print("测试集上的 train_score: ", test_score2)

y_pred2 = model2.predict(x_test)
rmse2 = (np.sqrt(mean_squared_error(y_test, y_pred2)))
print("测试集上的 rmse: ", rmse2)
```

输出结果如下所示：

```
0.006848057478670192
[ -1.04627962  1.0952031  -0.33778052  0.8531124  -1.77967908  2.53107055
 -0.25071085 -3.0417173  2.55200506 -1.73738179 -1.95608356  0.90793361
 -3.41528279]
训练集上的 train_score:  0.7239147183349263
测试集上的 train_score:  0.7953129722912958
测试集上的 rmse:  4.115681553106814
```

（3）基于多项式回归的波士顿房价预测。

```
# 多项式回归:
def pn_model(degree = 1):
```

```
    pn_feature = PolynomialFeatures(degree = degree, include_bias = False)
    li = linear_model.LinearRegression()
    pipeline = Pipeline([('pn',pn_feature),('li',li)])
    return pipeline

model3 = pn_model(degree = 2)
model3.fit(x_train, y_train)
train_score3 = model3.score(x_train, y_train)
test_score3 = model3.score(x_test, y_test)
print("训练集上的 train_score: ", train_score3)
print("测试集上的 train_score: ", test_score3)

y_pred3 = model3.predict(x_test)
rmse3 = (np.sqrt(mean_squared_error(y_test, y_pred3)))
print("测试集上的 rmse: ", rmse3)
```

输出结果如下所示：

```
训练集上的 train_score:  0.9305468799409318
测试集上的 train_score:  0.8600492818189005
测试集上的 rmse:  3.403174122380949
```

通过以上模型的预测结果我们可以看出，多项式回归模型取得了最佳的回归效果，显著降低了使用线性回归模型的均方根误差值，但是通过观察模型在训练集和测试集的表现可知，模型面临了过拟合的情况，也就是模型在训练集上性能较好，但是在测试集上性能较差。

步骤 4：模型结果可视化

在此步骤中，将训练好的多项式回归模型的预测效果进行可视化展示。理想状态下，模型的预测值与真实值相等，即 $y'=y$，两者应该在直线 $y=x$ 上分布。绘制以真实值为横坐标，预测值为纵坐标的散点图，观察预测值与 $y=x$ 直线的分布差异，可直观判断回归模型的性能。

```
plt.scatter(y_test, y_pred3, label = "test")
plt.plot([y_test.min(), y_test.max()],
         [y_test.min(), y_test.max()],
         'k -- ',
         lw = 3,
         label = "predict"
        )

plt.show()
```

输出结果如图 3-1-2 所示。

由图可以看出，多项式回归模型在小于 35 时的房价预测的较为准确。在超过 35 后，预测值会小于真实值。

在以上实践中，我们对比了几种简单的回归算法在波士顿房价预测数据集上的表现。对于回归算法来说，线性回归算法只能处理线性可分的数据，对于线性不可分数据，需要使用对数线性回归、广义线性回归或者其他回归算法，感兴趣的读者可以自行查阅资料学习。

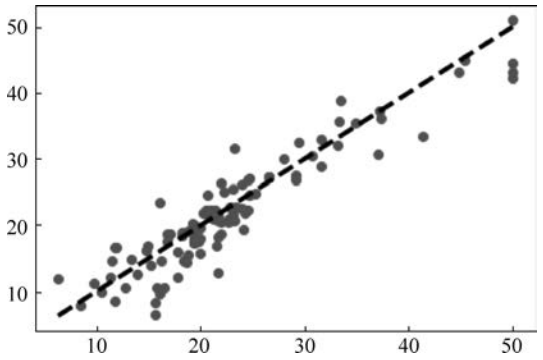

图 3-1-2　多项式回归模型的预测效果

基于朴素贝叶斯实现文本分类

3.2　实践二：基于朴素贝叶斯实现文本分类

分类问题是机器学习领域的另一个经典问题，和回归问题预测输入数据对应的连续值相对。分类问题即预测一系列的离散值，有很多经典的分类算法，例如贝叶斯算法、支持向量机以及逻辑回归算法等。贝叶斯分类算法是以贝叶斯定理为基础的一系列分类算法，包含朴素贝叶斯算法与树增强型朴素贝叶斯算法。朴素贝叶斯算法是最简单但是十分高效的贝叶斯分类算法，因为其假设输入特征之间相互独立，因此得名“朴素”。

在文本分类中，根据贝叶斯定理 $P(c|d)=\dfrac{P(d|c)*P(c)}{P(d)}$，文档 d 属于类型 c 的概率等于文档 d 对类型 c 的条件概率乘以类型 c 的出现概率，再除以文档 d 的出现概率，取概率最大的类型作为文本的判别类型，可形式化为 $y'=\underset{c\in C}{\operatorname{argmax}}\dfrac{P(d|c)P(c)}{P(d)}$，其中同一文档计算概率大小时，$P(d)$相同，故可省略，因此 $y'=\underset{c\in C}{\operatorname{argmax}}P(d|c)P(c)$。假设文档的特征为 $d=(x_1,x_2,x_3,\cdots,x_n)$，根据朴素贝叶斯的核心思想，各变量之间相互独立，则有 $P(d|c)=P(x_1|c)P(x_2|c)P(x_3|c)\cdots P(x_n|c)$，因此最终的分类结果变为：$y'=\underset{c\in C}{\operatorname{argmax}}P(x_1|c)P(x_2|c)P(x_3|c)\cdots P(x_n|c)P(c)=\underset{c\in C}{\operatorname{argmax}}P(c)\prod_{x\in d}P(x|c)$。根据上述观察，只需在全局数据集上统计 $P(c)$以及 $P(x|c)$，便可轻松获得文本的类型。

本节依旧使用 Sklearn 包中封装好的朴素贝叶斯算法，实现文本分类。本节实践的平台为 AI Studio，实践环境为 Python 3.7。

步骤 1：数据集简介

本实践采用的数据集为网上公开的从中文新闻网站上爬取 56821 条新闻摘要数据，数据集中包含 10 个类型（各类型数据量统计如表 3-2-1 所示），本节实践将其中 90%作为训练集，10%作为验证集。

表 3-2-1　新闻数据集样本数统计

国际	4354	汽车	7469
文化	5110	教育	8066
娱乐	6043	科技	6017
体育	4818	证券	3654
财经	7432	房产	3858

步骤 2：文本数据预处理

文本数据由于其自然语言形式，无法直接输入到计算机进行处理，需要对其进行自然语言到数字的转化。本实践最终将文本表示为 one-hot 形式，即对于给定词表，若文本中出现了词表中的词，则将与词表大小相同的向量中该词对应的位置置为 1，否则为 0。因此，需要在全局语料上构建一个词表，首先使用结巴分词对语料进行分词，为了不使词表过大造成过度复杂的计算，本实践只采样一定数量的高频词作为词表集合，同时为了避免一些高频无意义的词干扰文本表示，在构建词表时，首先也会将上述高频无意义的停用词去除。

```
# 导入必要的包
import random
import jieba                                                    # 处理中文
from sklearn import model_selection
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score,classification_report
import re,string
```

首先，加载文本，过滤其中的特殊字符。

```
# 结巴分词,将文本转化为词列表
def text_to_words(file_path):
    sentences_arr = []
    lab_arr = []
    with open(file_path,'r',encoding = 'utf8') as f:
        for line in f.readlines():
            lab_arr.append(line.split('_!_')[1])                # 文本所属标签
            sentence = line.split('_!_')[-1].strip()
            # 去除标点符号
            sentence = re.sub("[\s+\.\!\/_,$%SymbolYCp*(+\"\')]+|[+——()?
【】""!,.?、~@#¥%……&*()«»:]+", "",sentence)
            sentence = jieba.lcut(sentence, cut_all = False)
            sentences_arr.append(sentence)
return sentences_arr, lab_arr
```

加载停用词表，对文本词频进行统计，过滤掉停用词及词频较低的词，构建词表。

```
# 加载停用词表
def load_stopwords(file_path):
    stopwords = [line.strip() for line in open(file_path, encoding = 'UTF-8').readlines()]
return stopwords
# 词频统计
def get_dict(sentences_arr,stopswords):
    word_dic = {}
    for sentence in sentences_arr:
```

```
            for word in sentence:
                if word != ' ' and word.isalpha():
                    if word not in stopswords:                    # 停用词处理
                        word_dic[word] = word_dic.get(word,1) + 1
    # 按词频序排列
    word_dic = sorted(word_dic.items(),key = lambda x:x[1],reverse = True)
    return word_dic

    # 构建词表,过滤掉频率低于 word_num 的单词
    def get_feature_words(word_dic,word_num):
        '''
        从词典中选取 N 个特征词,形成特征词列表
        return: 特征词列表
        '''
        n = 0
        feature_words = []
        for word in word_dic:
            if n < word_num:
                feature_words.append(word[0])
            n += 1
    return feature_words

    # 文本特征表示
    def get_text_features(train_data_list, test_data_list, feature_words):
            # 根据特征词,将数据集中的句子转化为特征向量
            def text_features(text, feature_words):
            text_words = set(text)
            features = [1 if word in text_words else 0 for word in feature_words]
            return features                                       # 返回特征
        train_feature_list = [text_features(text, feature_words) for text in
    train_data_list]
        test_feature_list = [text_features(text, feature_words) for text in test_data_list]
    return train_feature_list, test_feature_list

    # 调用上述函数,完成词表构建
    sentences_arr, lab_arr = text_to_words('data/data6826/news_classify_data.txt')
    # 加载停用词
    stopwords = load_stopwords('data/data43470/stopwords_cn.txt')
    # 生成词典
    word_dic = get_dict(sentences_arr,stopwords)
    # 生成特征词列表,此处使用词维度为 10000
    feature_words = get_feature_words(word_dic,10000)
```

切分数据集,并将文本数据转化为固定长度的 ID 向量。

```
# 数据集划分
train_data_list, test_data_list, train_class_list, test_class_list = model_selection.train_
test_split(sentences_arr,lab_arr, test_size = 0.1)
# 生成特征向量
train_feature_list,test_feature_list = get_text_features(train_data_list,test_data_list,
feature_words)
```

步骤 3：模型定义与训练

上述概率计算中,可能存在某一个单词在某个类型中从来没有出现过,即某个属性的条

件概率为 0($P(x|c)=0$),此时会导致整体概率为 0。为了避免这种情况出现,引入拉普拉斯平滑参数,将条件概率为 0 的属性的概率设定为固定值,具体地,对每个类型下所有单词的计数加 1,当训练样本集数量充分大时,并不会对结果产生影响。下面调用接口的参数中,alpha 为 1 时,表示使用拉普拉斯平滑方式,若设置为 0,则不使用平滑;fit_prior 代表是否学习先验概率 $P(Y=c)$,如果设置为 False,则所有的样本类别输出都有相同的类别先验概率;class_prior 为各类型的先验概率,如果没有给出具体的先验概率则自动根据数据进行计算。

```
# 获取朴素贝叶斯分类器
classifier = MultinomialNB(alpha = 1.0,                    # 拉普拉斯平滑
                           fit_prior = True,                # 是否要考虑先验概率
                           class_prior = None)

# 进行训练
classifier.fit(train_feature_list, train_class_list)
```

步骤 4:模型验证

模型训练结束后,可使用验证集测试模型的性能,同第 3.2 节,输出准确率的同时,对各个类型的精确率、召回率以及 F1 值也进行输出。

```
# 在验证集上进行验证
test_accuracy = classifier.score(test_feature_list, test_class_list)
print(test_accuracy)
predict = classifier.predict(test_feature_list)
print(classification_report(test_class_list, predict))
```

输出结果如图 3-2-1 所示。

```
accuracy_score: 0.7700
Classification report for classifier:
              precision    recall  f1-score   support

           0       0.73      0.70      0.72       522
           1       0.74      0.86      0.79       558
           2       0.89      0.82      0.86       504
           3       0.64      0.66      0.65       784
           4       0.82      0.79      0.81       371
           5       0.85      0.85      0.85       733
           6       0.82      0.83      0.83       847
           7       0.71      0.69      0.70       572
           8       0.78      0.66      0.72       433
           9       0.76      0.81      0.78       359

    accuracy                           0.77      5683
   macro avg       0.77      0.77      0.77      5683
weighted avg       0.77      0.77      0.77      5683
```

图 3-2-1　朴素贝叶斯文本分类结果

步骤 5:模型预测

使用上述训练好的模型,对任意给定的文本数据,可进行预测,观察模型的泛化性能。

```
# 加载句子,对句子进行预处理:去除标点、分词
def load_sentence(sentence):
```

```
    # 去除标点符号
    sentence = re.sub("[\s+\.\!\/_,$%SymbolYCp*(+\"\')]+|[+——()?【】""!,.?、~@#
¥%…&*()«»:]+", "",sentence)
    sentence = jieba.lcut(sentence, cut_all=False)
return sentence

lab = ['文化', '娱乐', '体育', '财经','房产', '汽车', '教育', '科技', '国际', '证券']

p_data = '【中国稳健前行】应对风险挑战必须发挥制度优势'
sentence = load_sentence(p_data)
sentence = [sentence]
print('分词结果:', sentence)
# 形成特征向量
p_words = get_text_features(sentence,sentence,feature_words)
res = classifier.predict(p_words[0])
print(lab[int(res)])
```

输出结果如图 3-2-2 所示。

```
分词结果: [['中国', '稳健', '前行', '应对', '风险', '挑战', '必须', '发挥', '制度', '优势']]
所属类型:  财经
```

图 3-2-2　文本分类预测结果展示

基于逻辑回归模型实现手写数字识别

3.3　实践三：基于逻辑回归模型实现手写数字识别

逻辑回归是线性回归的一个变体版本，即建模函数 $\ln\frac{y}{1-y}=\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}+\boldsymbol{b}$。此处，$y$ 为样本 x 作为正样本的可能性，$1-y$ 为其为负样本的可能性，两者的比值 $\frac{y}{1-y}$ 称为几率，反映了 x 作为正样本的相对可能性，因此逻辑回归又称作对数几率回归。

逻辑回归虽然称作回归，但实际上是一种分类学习算法，无须事先假设数据的分布即可进行建模，避免了先验假设分布偏差带来的影响，并且得到的是近似概率预测，对需要概率结果辅助决策的任务十分友好。逻辑回归使用极大似然估计进行参数学习，即最大化模型的对数似然值，使得每个样本属于真实标签的概率越大越好。该优化目标可以通过牛顿法、梯度下降法等求得最优解。

Sklearn 是 Python 的一个机器学习库，它有比较完整的监督学习与非监督学习的算法实现，本节将利用 Sklearn 中的逻辑回归算法，实现 MNIST 手写数字识别。本节实践的平台为 AI Studio，实践环境为 Python 3.7。

步骤 1：数据集加载及预处理

MNIST 数据集来自美国国家标准与技术研究所，训练集由来自 250 个不同人手写的数字构成，其中，50％是高中学生，50％是人口普查局的工作人员，测试集也包含同样比例人群的手写数字图片。数据集总共包含 60000 个训练集和 10000 测试数据集，分为图片和标签，图片是 28×28 的像素矩阵，标签为 0～9 共 10 个数字。由于数据集存储格式为二进制，因

此在读取时需要逐字节进行解析。首先将数据集挂载到当前工作空间下，然后解压（在 AI Studio 可编辑 Notebook 界面中，若要执行 Linux 命令，只需在命令前加“!”即可），读取图片数据。

```
!unzip data/data7869/mnist.zip
!gzip -dfq mnist/train-labels-idx1-ubyte.gz
!gzip -dfq mnist/t10k-labels-idx1-ubyte.gz
!gzip -dfq mnist/train-images-idx3-ubyte.gz
!gzip -dfq mnist/t10k-images-idx3-ubyte.gz

# 导入相关包
import struct,os
import numpy as np
from array import array as pyarray
from numpy import append, array, int8, uint8, zeros
from sklearn.metrics import accuracy_score,classification_report
import matplotlib.pyplot as plt

# 定义加载 MNIST 数据集的函数
def load_mnist(image_file, label_file, path="mnist"):
    digits=np.arange(10)

    fname_image = os.path.join(path, image_file)
    fname_label = os.path.join(path, label_file)

    flbl = open(fname_label, 'rb')                          # 读取标签文件
    magic_nr, size = struct.unpack(">II", flbl.read(8))
    lbl = pyarray("b", flbl.read())
    flbl.close()

    fimg = open(fname_image, 'rb')                          # 读取图片文件
    magic_nr, size, rows, cols = struct.unpack(">IIII", fimg.read(16))
    img = pyarray("B", fimg.read())
    fimg.close()

    ind = [ k for k in range(size) if lbl[k] in digits ]
    N = len(ind)

    images = zeros((N, rows*cols), dtype=uint8)
    labels = zeros((N, 1), dtype=int8)
    for i in range(len(ind)):                               # 将图片转化为像素矩阵格式
        images[i] = array(img[ ind[i]*rows*cols : (ind[i]+1)*rows*cols ]).reshape((1,
rows*cols))
        labels[i] = lbl[ind[i]]

    return images, labels

# 定义图片展示函数
def show_image(imgdata, imgtarget, show_column, show_row):
    # 注意这里的 show_column*show_row==len(imgdata)
    for index,(im,it) in enumerate(list(zip(imgdata,imgtarget))):
        xx = im.reshape(28,28)
```

```
        plt.subplots_adjust(left = 1, bottom = None, right = 3, top = 2, wspace = None, hspace =
None)
        plt.subplot(show_row, show_column, index + 1)
        plt.axis('off')
        plt.imshow(xx , cmap = 'gray',interpolation = 'nearest')
        plt.title('label: % i' % it)

# 调用函数,加载训练集数据
train_image, train_label = load_mnist("train - images - idx3 - ubyte", "train - labels - idx1 -
ubyte")
# 调用函数,加载测试集数据
test_image, test_label = load_mnist("t10k - images - idx3 - ubyte","t10k - labels - idx1 -
ubyte")
# 显示训练集前 50 个数字
show_image(train_image[:50], train_label[:50], 10,5)
```

输出结果如图 3-3-1 所示。

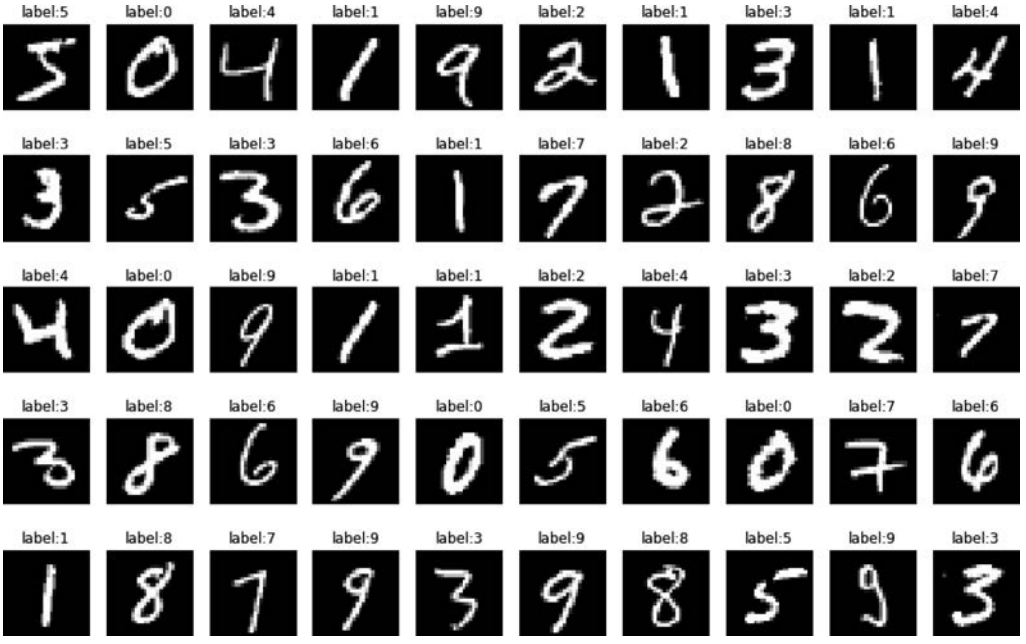

图 3-3-1　MINST 手写数字

步骤 2：模型定义

此处直接将 sklearn.linear_model 中的 LogisticRegression 导入即可，注意，虽然逻辑回归并没有直接建模输出 y 与输入特征 $\boldsymbol{x}$ 之间的映射关系，但它本质上是线性回归算法的一种变体，且回归参数 $\boldsymbol{w}$ 对于输入特征而言仍是线性的，因此也属于线性模型的范畴。

```
# 导入 LogisticRegression 类
from sklearn.linear_model import LogisticRegression
# 实例化 LogisticRegression 类
lr = LogisticRegression()
```

步骤 3：模型学习

由于图片数据的像素值取值范围为 0～255，过大的计算值可能导致计算结果非常大，或者梯度变化剧烈，因此不利于模型的学习与收敛。为避免上述情况出现，首先需要对训练数据做预处理，也就是尺度缩放，比如对每个像素值都除以其最大像素值 255，将所有像素值压缩到 0～1 的范围内，然后再进行学习。

```
# 数据缩放
train_image = [im/255.0 for im in train_image]
# 训练模型
lr.fit(train_image,train_label)
```

步骤 4：模型验证

模型训练结束后，可在验证集或测试集上测试其性能，对于分类任务，最常见的评价指标包括准确率(accuracy)、精确率(precision)、召回率(recall)、F1 值(F1-score)等。其中，精确率反映正样本的判断准确率，召回率反映正样本中被实际识别的样本比例，而 F1 值则是精确率与召回率的折中，在各类型样本数量不均衡时，该指标很好地反映模型的性能。

```
# 数据缩放
test_image = [im/255.0 for im in test_image]
# 测试集结果预测
predict = lr.predict(test_image)
# 打印准确率及各分类评价指标
print("accuracy_score: %.4lf" % accuracy_score(predict,test_label))
print("Classification report for classifier %s:\n%s\n" % (lr, classification_report(test_
label, predict)))
```

各指标输出如图 3-3-2 所示。

```
accuracy_score: 0.9257
Classification report for classifier LogisticRegression
              precision    recall  f1-score   support

           0       0.95      0.98      0.96       980
           1       0.96      0.98      0.97      1135
           2       0.93      0.90      0.91      1032
           3       0.90      0.91      0.91      1010
           4       0.94      0.93      0.93       982
           5       0.91      0.88      0.89       892
           6       0.94      0.95      0.94       958
           7       0.94      0.92      0.93      1028
           8       0.87      0.88      0.88       974
           9       0.91      0.92      0.91      1009

    accuracy                           0.93     10000
   macro avg       0.92      0.92      0.92     10000
weighted avg       0.93      0.93      0.93     10000
```

图 3-3-2　逻辑回归手写数字识别结果

基于 SVM/决策树/XGBoost 算法实现鸢尾花

3.4　实践四：基于 SVM/决策树/XGBoost 算法实现鸢尾花

支持向量机(SVM)的主要思想为最大化不同类型的样本到分类超平面之间的距离和。

当数据完全线性可分时，得到的最大间隔是硬间隔，即两个平行的超平面（间隔带）之间不存在样本点；当数据部分线性可分时，两个超平面之间允许存在一些样本点，此时得到的最大间隔平面是软间隔平面。对于完全线性不可分的数据，一般的支持向量机算法无法满足要求，但是适当使用核技巧，将非线性样本特征映射到高维线性可分空间，便可应用支持向量机进行分类，此时的支持向量机称为非线性支持向量机。常用的核技巧包括：线性核函数、多项式核函数、高斯核函数（径向基函数）。其中，高斯核函数需要进行调参，即核变换的带宽，它控制径向作用范围。

决策树算法也是一种典型的分类算法，其首先对数据进行处理，利用归纳算法生成可读的规则和决策树，然后使用决策对新数据进行分析。生成的决策树是一种树形结构，其中，每个内部节点表示一个属性上的测试，每个分支代表一个测试输出，每个叶节点代表一种类别。本质上决策树是通过一系列规则对数据进行分类的过程。具体来说，对于决策的过程，首先从根节点开始，测试待分类项中相应的特征属性，并按照其值选择输出分支，直至到达叶节点，将叶节点存放的类别作为决策结果。

集成学习在分类任务中是另一种经典的算法，其通过将多个弱分类器集成在一起，使它们共同完成学习任务以构建一个强分类器。潜在的哲学思想是“三个臭皮匠，赛过诸葛亮”。集成学习中有两类集成方法，分别为 Bagging（Bootstrap Aggregating）和 Boosting 方法。Bagging 方法基于数据随机重抽样的思想，利用 Bootstrap 抽样（有放回的随机抽样）方法从整体数据集中抽样得到 N 个数据集，之后在每个数据集上学习出一个弱分类器模型，再利用 N 个模型的输出投票得到最后的强分类器的预测结果。对于 Bagging 方法，可以使用决策树作为其基分类器。Boosting 方法基于错误提升分类器性能的思想，通过集中关注被已有分类器分类错误的样本，构建新的分类器。也就是说，每一次迭代时训练集的选择都与前面各轮的学习结果有关，而且每次都是通过更新各个样本权重的方式来改变数据分布。在 Boosting 方法中，最终生成的强分类器中的各个弱分类器也具有不同的权重，预测效果越好的弱分类器的权重越高，预测效果越差的弱分类器的权重越低。Boosting 方法的代表算法包括随机森林、GBDT（Gradient Boosting Decision Tree）梯度提升决策树算法以及 xgboost 算法等。

Boosting 方法采用的是加法模型和前向分步算法来解决分类和回归问题，而以决策树作为基函数的提升方法称为提升树。GBDT 是提升树算法的一种，其使用 CART（Classification and Regression Tree）分类和回归树中的回归树作为基分类器。GBDT 是一种迭代的决策树算法，通过多轮迭代，每轮学习都在上一轮训练的残差（损失函数的负梯度）的基础上进行训练。在回归问题中，每轮迭代产生一棵 CART 回归树，迭代结束时将得到多棵 CART 回归树，然后把所有的树加总起来就得到了最终的提升树。XGBoost 算法是 Boosting 集成学习算法中的另一个经典算法，其基于 GBDT 算法改进而来的，二者本质上都利用了 Boosting 算法中拟合残差的思想。

本节使用 Sklearn 中封装好的支持向量机、决策树算法以及优化的分布式梯度增强库 XGBoost 实现鸢尾花数据集的分类任务，并绘制分类超平面，可视化分类效果。本节实践的平台为 AI Studio，实践环境为 Python 3.7。

步骤 1：数据集加载

在第 3.1 节是直接从 sklearn.datasets 中加载集成的数据集，而本小节采用另一种数据加载方式，从挂载在当前目录下的数据集文件中读取数据，用于训练。

```
# 加载相关包
import numpy as np
from matplotlib import colors
from sklearn import svm
from sklearn import model_selection
import matplotlib.pyplot as plt
import matplotlib as mpl

# 将字符串转化为整形
def iris_type(s):
    it = {b'Iris-setosa':0, b'Iris-versicolor':1, b'Iris-virginica':2}
    return it[s]

# 加载数据
data = np.loadtxt('/home/aistudio/data/data2301/iris.data',
dtype=float,                                                    # 数据类型
                  delimiter=',',                                # 数据分割符
                  converters={4:iris_type})                     # 将标签用 iris_type 进行转换
# 数据分割,将样本特征与样本标签进行分割
x, y = np.split(data, (4, ), axis=1)
x = x[:, :2]                                                    # 取前两个特征进行分类
# 调用 model_selection 函数进行训练集、测试集切分
x_train, x_test, y_train, y_test = model_selection.train_test_split(x, y, random_state=1,
test_size=0.2)
```

步骤 2：模型配置及训练

调用 fit()函数构造训练函数。

```
# 训练函数
def train(clf, x_train, y_train):
    clf.fit(x_train, y_train.ravel())
```

(1) 构造 SVM 分类器。

sklearn.svm.SVC()函数提供多个可配置参数。其中，C 为错误项的惩罚系数。C 越大，对训练集错误项的惩罚越大，模型在训练集上的准确率越高，越容易过拟合；C 越小，越允许训练样本中有一些误分类错误的样本，泛化能力越强。对于训练样本带有噪声的情况，一般采用较小的 C，把训练样本集中错误分类的样本作为噪声。kernel 为采用的核函数，可选的为 linear/poly/rbf/sigmoid/precomputed，默认为线性核；decision_function_shape 设置为 ovr 时表示一对多分类决策函数，设置为 ovo 时表示一对一分类决策函数。

```
# SVM 分类器构建
from sklearn import svm
# 构建 SVM 分类器
def SVM_classifier():
```

```
    clf = svm.SVC(C = 0.8, kernel = 'rbf', decision_function_shape = 'ovo')
    return clf

# 生成 SVM 模型以及调用函数训练
clf1 = SVM_classifier()
train(clf1, x_train, y_train)
```

（2）构造决策树分类器。

调用 sklearn.tree.DecisionTreeClassifier()函数构造决策树模型，其中包含了一系列的可选参数。例如：criterion 为特性选择的标准，默认设置为 gini，即基尼系数，其是 CART 算法中采用的度量标准，该参数还可以设置为 entropy，表示信息增益；splitter 为特征节点划分标准，默认设置为 best，其表示在所有特征上递归，一般用于训练样本数据量不大的场合，该参数还可以设置为 random，表示随机选择一部分特征进行递归，一般用于训练数据量较大的场合，可以减少计算量；max_depth 为设置决策树的最大深度，默认为 None，None 表示不对决策树的最大深度作约束，直到每个叶节点上的样本均属于同一类，或者少于 min_samples_leaf 参数指定的叶节点上的样本个数，也可以指定一个整型数值，设置树的最大深度，在样本数据量较大时，可以通过设置该参数提前结束树的生长；min_samples_split 为当对一个内部节点划分时，要求该节点上的最小样本数，默认为 2；min_samples_leaf 为设置叶节点上的最小样本数，默认为 1。

```
from sklearn.tree import DecisionTreeClassifier
# 构建决策树分类器
def dtree_classifier():
    clf = DecisionTreeClassifier(criterion = "entropy", splitter = 'best',
                                 max_depth = None,
                                 min_samples_split = 5,
                                 min_samples_leaf = 1,
                                 min_weight_fraction_leaf = 0.0,
                                 max_features = None,
                                 random_state = None,
                                 max_leaf_nodes = None,
                                 min_impurity_decrease = 0.0,
                                 min_impurity_split = None,
                                 class_weight = None,
                                 presort = False)
    return clf

# 生成决策树模型定义以及调用函数训练
clf2 = dtree_classifier()
train(clf2, x_train, y_train)
```

（3）构造 xgboost 分类器。

调用 xgboost.XGBClassifier 构造 xgboost 模型，其中包含了一系列的可选参数。例如：learning_rate 为学习率，用于控制每次迭代更新权重时的步长，默认为 0.3；n_estimatores 为总共迭代的次数，即决策树的个数；max_depth 表示决策树的最大深度，默认值为 6；objective 用于指定训练任务的目标，参数默认为 reg：squarederror，表示以平方损失为损失函数的回归模型，还可以设置为 binary:logistic 以及 multi：softmax 等；binary:logistic 表

示二分类逻辑回归模型(输出为概率,即 sigmoid 函数值);multi:softmax 表示使用 softmax 作为目标函数的多分类模型。

```
from xgboost import XGBClassifier
def xgb_classifier():
    clf = XGBClassifier(learning_rate = 0.001,
                        n_estimators = 3,           # 树的个数, 10 棵树建 xgboost
                        max_depth = None,           # 树的深度
                        min_child_weight = 1,       # 叶节点最小权重
                        gamma = 0,                  # 惩罚项中叶节点个数前的参数
                        subsample = 1,              # 所有样本建立决策树
                        colsample_btree = 1,        # 所有特征建立决策树
                        scale_pos_weight = 0.7,     # 解决样本个数不均衡的问题
                        random_state = 27,          # 随机数
                        objective = 'multi:softprob',
                        slient = 0)
    return clf

# 生成 xgboost 模型定义以及调用函数训练
clf3 = xgb_classifier()
train(clf3, x_train, y_train)
```

步骤 3:模型验证

在划分好的测试集上测试模型的准确率,使用 Sklearn 中机器学习模型封装好的方法 score()计算模型预测结果的准确率。对于 SVM 模型,同时输出样本 x 到各个决策超平面的距离。

```
def print_accuracy(clf, x_train, y_train, x_test, y_test, model_name):
    print(model_name + ': ')
    print('\t training prediction: %.3f' %(clf.score(x_train, y_train)))
    print('\t test prediction: %.3f' %(clf.score(x_test, y_test)))

    if model_name == 'SVM':
        print('\t decision_function:\n \t\t', clf.decision_function(x_train)[:1])
```

输出 SVM、决策树以及 xgboost 模型的模型验证结果。

```
print_accuracy(clf1, x_train, y_train, x_test, y_test, 'SVM')
print_accuracy(clf2, x_train, y_train, x_test, y_test, 'DecisionTree')
print_accuracy(clf3, x_train, y_train, x_test, y_test, 'XGBoost')
```

通过图 3-4-1 可以看到,SVM 算法取得了最好的模型预测效果。

```
SVM:
        training prediction:0.800
        test prediction:0.833
        decision_function:
                [[-1.13785175 -1.09754144 -0.16640687]]
DecisionTree:
        training prediction:0.892
        test prediction:0.733
XGBoost:
        training prediction:0.850
        test prediction:0.767
```

图 3-4-1　鸢尾花分类的各个模型的预测结果

步骤 4：模型效果可视化展示

针对 SVM 模型的预测效果，绘制可视化结果。若要绘制各个类型对应的空间区域，需要采样大量的样本点，但是本数据集仅包含 150 条数据，绘制的区域不太精细。因此，需要生成大规模的样本数据，并根据生成的数据进行分类区域的绘制，过程如下（本实践采用样本的前两维特征进行分类）：首先在各维特征的最大值与最小值区间内进行采样，生成行相同矩阵（矩阵每行向量中各元素值都相同）与列相同矩阵（矩阵每列向量中各元素值都相同），然后将两矩阵拉平为两个长向量，两个长向量每个元素分别作为样本的第一个特征与第二个特征，使用训练好的 SVM 模型对生成的样本点进行预测，将生成的样本点使用不同的颜色散落在坐标空间中，当样本点足够多时，分类边界便会显示得更加精细。

```
def draw(clf, x):
    iris_feature = 'sepal length', 'sepal width', 'petal length', 'petal width'
    x1_min, x1_max = x[:, 0].min(), x[:, 0].max()
    x2_min, x2_max = x[:, 1].min(), x[:, 1].max()

    x1, x2 = np.mgrid[x1_min:x1_max:200j, x2_min:x2_max:200j]
    grid_test = np.stack((x1.flat, x2.flat), axis = 1)
    print("grid_test:\n", grid_test[:2])

    grid_hat = clf.predict(grid_test)

    # 预测分类值 得到[0, 0, ..., 2, 2]
    print('grid_hat:\n', grid_hat)
    # 使得 grid_hat 和 x1 形状一致
    grid_hat = grid_hat.reshape(x1.shape)
    cm_light = mpl.colors.ListedColormap(['#A0FFA0', '#FFA0A0', '#A0A0FF'])
    cm_dark = mpl.colors.ListedColormap(['g', 'b', 'r'])

    plt.pcolormesh(x1, x2, grid_hat, cmap = cm_light)
    plt.scatter(x[:, 0], x[:, 1], c = np.squeeze(y), edgecolor = 'k', s = 50, cmap = cm_dark )
    plt.scatter(x_test[:, 0], x_test[:, 1], s = 120, facecolor = 'none', zorder = 10 )
    plt.xlabel(iris_feature[0], fontsize = 20)              # 注意单词的拼写 label
    plt.ylabel(iris_feature[1], fontsize = 20)
    plt.xlim(x1_min, x1_max)
    plt.ylim(x2_min, x2_max)
    plt.title('Iris data classification via SVM', fontsize = 30)
    plt.grid()
    plt.show()
draw(clf1, x)
```

输出结果如图 3-4-2 和图 3-4-3 所示。

```
grid_test:
 [[4.3       2.       ]
 [4.3       2.0120603]]
grid_hat:
 [0. 0. 0. ... 2. 2. 2.]
```

图 3-4-2　SVM 鸢尾花分类—预测结果

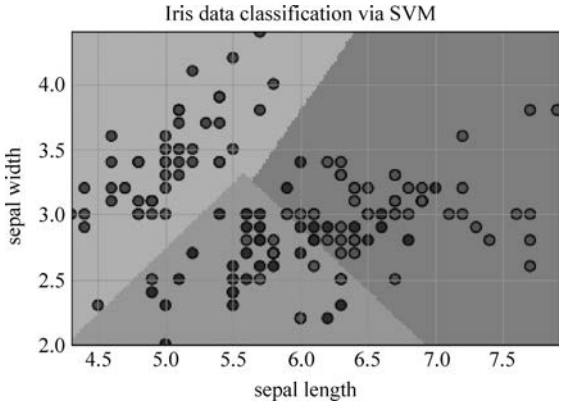

图 3-4-3　SVM 鸢尾花分类可视化

3.5　实践五：基于 K-means/层次聚类算法实现自制数据集聚类

基于 K-means/层次聚类算法实现自制数据集聚类

聚类问题是无监督学习的问题，算法的思想在于“物以类聚，人以群分”，聚类算法通过感知样本间的相似度，进行类别归纳，对新的输入进行输出预测。经典的聚类算法包括 K-means 算法以及层次聚类算法等。

K-means 算法是一种经典的无监督聚类算法，对于给定的样本集，按照样本之间的距离大小，将样本集划分为 K 个簇，让簇内的点尽量紧密的连在一起，而让簇间的距离尽量的大。K-means 的学习过程本质上是不停更新簇心的过程，一旦簇心确定，该算法便完成了学习过程。K 的取值也需要人为定义，K 很大时，模型趋向于在训练集上表现地好，即过拟合，但在测试集上性能可能较差；K 过小时，可能导致簇心不准确，在训练集与测试集上的性能均较差。因此，虽然 K-means 算法较为简单，但是也存在天然的弊端，且对离群点很敏感。具体来说，K-means 算法首先随机初始化或随机抽取 K 个样本点作为簇心，然后以这 K 个簇心进行聚类，聚类后重新计算簇心（一般为同一簇内样本的均值），重复上述操作，直至簇心趋于稳定或者达到指定迭代次数时停止迭代。

层次聚类算法是第二类重要的聚类方法。层次聚类方法是对给定的数据集进行层次的分解，直到满足某种条件为止。层次聚类方法可以分为两类：“自底向上”的聚合策略和“自顶向下”的分拆策略。对于“自底向上”的聚合策略，每一个对象都是一个聚类簇，选最近的聚类簇合并，最终所有的对象都属于一个聚类簇。对于“自顶向下”的分拆策略，所有的对象都属于一个聚类簇，按一定规则将聚类簇分拆，最终每一个对象都是一个聚类簇。

本书使用 K-means 算法以及层次聚类算法实现聚类。本节实践的平台为 AI Studio，实践环境为 Python 3.7。

步骤 1：创建数据集

使用 sklearn.datasets.make_blobs()函数构建聚类数据集，其中包含一些可设置的参数。例如：n_samples 表示样本数量；n_features 表示每一个样本包含的特征值数量；centers 表示聚类中心点的数量，即样本中包含的类别数；random_state 为随机数种子，可以

用于固定随机生成的数据；cluster_std 用于设置每个类别的方差。

在本节实践代码中，调用 make_blobs()函数生成 600 个样本点，并设置 4 类的聚类簇。

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt

# 自己创建聚类数据集
X, y = make_blobs(n_samples = 600, n_features = 3, centers = 4, random_state = 1)
fig, ax1 = plt.subplots(1)
```

将生成的所有样本点进行可视化，结果如图 3-5-1 所示。

```
ax1.scatter(X[:, 0], X[:, 1], marker = 'o', s = 8)
plt.savefig('./1.png')
plt.show()
```

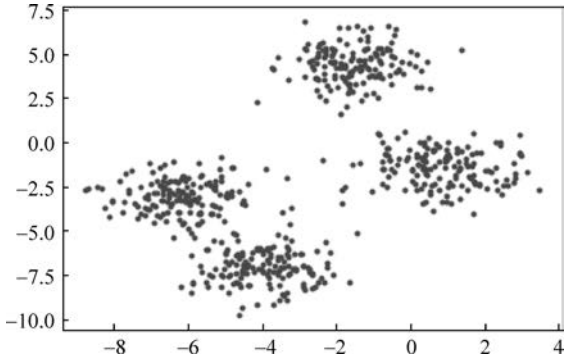

图 3-5-1　自制数据集样本点可视化

将样本点所属的聚类簇进行可视化，结果如图 3-5-2 所示。

```
# 绘制二维数据分布图,每个样本使用两个特征,绘制其二维数据分布图
color = ["red","pink","orange","gray"]
fig, ax1 = plt.subplots(1)
for i in range(4):
    ax1.scatter(X[y == i, 0], X[y == i, 1], marker = 'o', s = 8, c = color[i])
plt.savefig('./2.png')
plt.show()
```

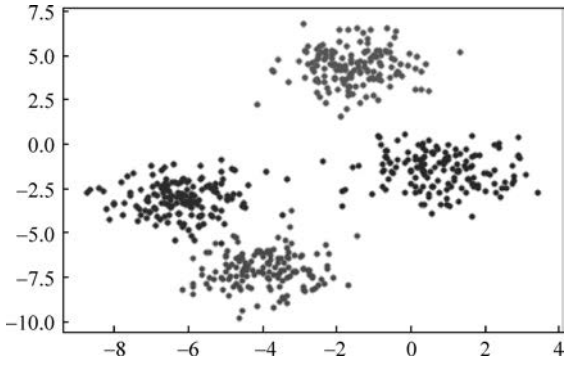

图 3-5-2　样本点所属簇类可视化

步骤 2：模型配置及训练

调用 fit()函数构造训练函数。

```
# 训练函数
def train(estimator):
    estimator.fit(X)                                    # 聚类
```

(1) 构造 K-means 聚类模型，且设置聚类簇数为 4。

```
from sklearn.cluster import KMeans

# 构造 K-Means 聚类模型
def Kmeans_model(n_clusters):
    model = KMeans(n_clusters = n_clusters)             # 构造聚类器
    return model

# 初始化实例，并开启训练拟合
model1 = Kmeans_model(4)
train(model1)
```

(2) 构造层次聚类模型，且设置聚类簇数为 4。

```
from sklearn.cluster import AgglomerativeClustering

# 构造层次聚类模型
def Agg_model(n_clusters):
    model = AgglomerativeClustering(linkage = 'ward', n_clusters = n_clusters)
    return model

# 初始化实例，并开启训练拟合
model2 = Agg_model(4)
train(model2)
```

步骤 3：模型效果可视化展示

模型训练结束后，通过 labels 获取聚类标签进行可视化。聚类结果如图 3-5-3 和图 3-5-4 所示，可以看到，两类聚类模型都取得了较好的聚类效果，二者聚类出的样本点的分布形式接近于其真实分布。

(1) K-means 聚类模型。

```
label_pred = model1.labels_                             # 获取聚类标签
# 绘制 K-means 聚类结果
x0 = X[label_pred == 0]
x1 = X[label_pred == 1]
x2 = X[label_pred == 2]
x3 = X[label_pred == 3]
plt.scatter(x0[:, 0], x0[:, 1], c = "red", marker = 'o', label = 'label0')
plt.scatter(x1[:, 0], x1[:, 1], c = "green", marker = '*', label = 'label1')
plt.scatter(x2[:, 0], x2[:, 1], c = "blue", marker = '+', label = 'label2')
plt.scatter(x3[:, 0], x3[:, 1], c = "purple", marker = 'SymbolYCp', label = 'label3')
plt.xlabel('Feature1')
```

```
plt.ylabel('Feature2')
plt.legend(loc = 2)
plt.savefig('./3.png')
plt.show()
```

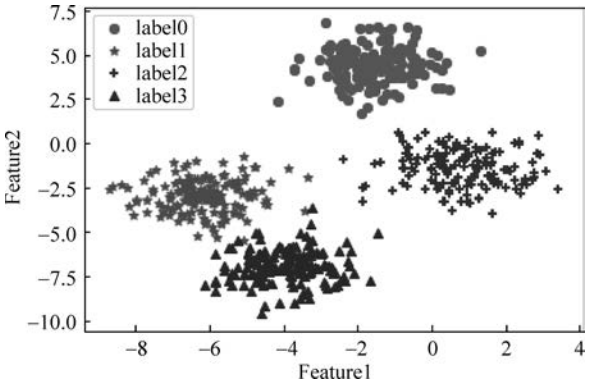

图 3-5-3　K-means 模型聚类结果

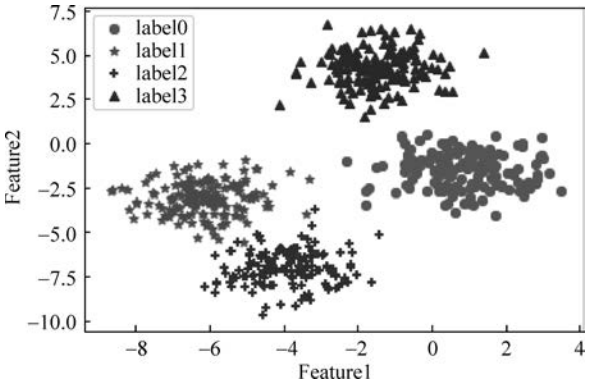

图 3-5-4　层次聚类模型聚类结果

（2）层次聚类模型。

```
label_pred = model2.labels_                                    # 获取聚类标签
# 绘制层次聚类结果
x0 = X[label_pred == 0]
x1 = X[label_pred == 1]
x2 = X[label_pred == 2]
x3 = X[label_pred == 3]
plt.scatter(x0[:, 0], x0[:, 1], c = "red", marker = 'o', label = 'label0')
plt.scatter(x1[:, 0], x1[:, 1], c = "green", marker = '*', label = 'label1')
plt.scatter(x2[:, 0], x2[:, 1], c = "blue", marker = '+', label = 'label2')
plt.scatter(x3[:, 0], x3[:, 1], c = "purple", marker = 'SymbolYCp', label = 'label3')
plt.xlabel('Feature1')
plt.ylabel('Feature2')
plt.legend(loc = 2)
plt.savefig('./4.png')
plt.show()
```

步骤 4：手动实现 K-means 算法

在以上步骤中，调用了 Sklearn 封装好的库快速实现了 K-means 算法。下面，将手动实现 K-means 以更好地了解其实现过程。

（1）首先定义距离测量标准，本书使用欧氏距离衡量两样本之间的距离。

```
# 欧氏距离计算
def distEclud(x,y):
    return np.sqrt(np.sum((x-y)**2))                    # 计算欧氏距离
```

（2）定义簇心，此处使用随机抽取的 K 个样本点为簇心，进行后续计算。

```
# 为给定数据集构建一个包含 K 个随机质心 centroids 的集合
def randCent(dataSet,k):
    m,n = dataSet.shape
    centroids = np.zeros((k,n))
    for i in range(k):
        index = int(np.random.uniform(0,m))
        centroids[i,:] = dataSet[index,:]
    return centroids
```

（3）实现 K-means 算法：首先初始化簇心，然后遍历所有点，找到其对应的簇，更新簇心，重复迭代上述过程，直到簇心不再发生变化。

```
# K 均值聚类算法
def KMeans(dataSet,k):
    m = np.shape(dataSet)[0]

    clusterAssment = np.mat(np.zeros((m, 2)))
    clusterChange = True

    # 1.初始化质心 centroids
    centroids = randCent(dataSet,k)
    while clusterChange:
        # 样本所属簇不再更新时停止迭代
        clusterChange = False

        # 遍历所有的样本
        for i in range(m):
            minDist = 100000.0
            minIndex = -1

            # 遍历所有的质心
            # 2.找出最近的质心
            for j in range(k):
                # 计算该样本到质心的欧式距离，找到距离最近的那个质心 minIndex
                distance = distEclud(centroids[j,:],dataSet[i,:])
                if distance < minDist:
                    minDist = distance
                    minIndex = j
```

```
                # 3.更新该行样本所属的簇
                if clusterAssment[i,0] != minIndex:
                    clusterChange = True
                    clusterAssment[i,:] = minIndex,minDist**2
            # 4.更新质心
            for j in range(k):
                # np.nonzero(x)返回值不为零的元素的下标,它的返回值是一个长度为 x.ndim(x 的
轴数)的元组
                # 元组的每个元素都是一个整数数组,其值为非零元素的下标在对应轴上的值
                # 矩阵名.A 代表将矩阵转化为 array 数组类型

                # 这里取矩阵 clusterAssment 所有行的第一列,转为一个 array 数组,与 j(簇类标签
值)比较,返回 true or false
                # 通过 np.nonzero 产生一个 array,其中是对应簇类所有的点的下标值(x 个)
                # 再用这些下标值求出 dataSet 数据集中的对应行,保存为 pointsInCluster(x*4)
                pointsInCluster = dataSet[np.nonzero(clusterAssment[:,0].A == j)[0]]
                                                        # 获取对应簇类所有的点(x*4)
                centroids[j,:] = np.mean(pointsInCluster,axis=0)
                                                        # 求均值,产生新的质心

    print("cluster complete")
    return centroids,clusterAssment
```

(4) 可视化展示函数定义,分别取前两个维度的特征与后两个维度的特征绘图,便于观察聚类效果。

```
def draw(data,center,assment):
    length=len(center)
    fig=plt.figure
    data1=data[np.nonzero(assment[:,0].A == 0)[0]]
    data2=data[np.nonzero(assment[:,0].A == 1)[0]]
    data3=data[np.nonzero(assment[:,0].A == 2)[0]]
    data4=data[np.nonzero(assment[:,0].A == 3)[0]]
    # 选取前两个维度绘制原始数据的散点图
    plt.scatter(data1[:,0],data1[:,1],c="red",marker='o',label='label0')
    plt.scatter(data2[:,0],data2[:,1],c="green", marker='*', label='label1')
    plt.scatter(data3[:,0],data3[:,1],c="blue", marker='+', label='label2')
    plt.scatter(data4[:,0],data4[:,1],c="purple", marker='SymbolYCp', label='label3')
    # 绘制簇的质心点
    for i in range(length):
        plt.annotate('center',xy=(center[i,0],center[i,1]),xytext=\
        (center[i,0]+1,center[i,1]+1),arrowprops=dict(facecolor='yellow'))
    plt.savefig('./5.png')
    plt.show()
```

(5) 执行 K-means 过程,实现样本点的聚类,此处同样直接设置聚类簇数 $K=4$。

```
k = 4
centroids,clusterAssment = KMeans(X, k)
draw(dataSet, centroids, clusterAssment)
```

可视化结果如图 3-5-5 所示，其中箭头指向簇心。

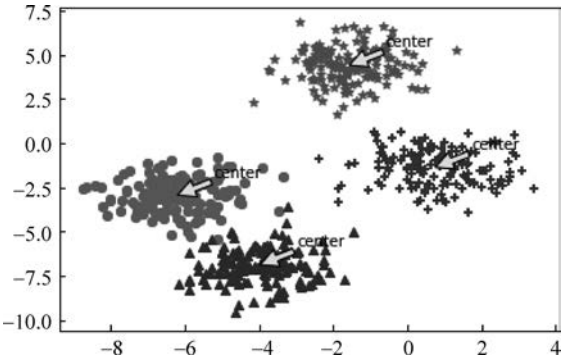

图 3-5-5　手动实现 K-Means 聚类模型的聚类效果