第5章 模型优化

CHAPTER 5

第4章讲解了不同结构的模型，并概述了其适用的问题类型。本章将讲解如何优化模型，使其接近最优的预测效果，以及如何根据预测结果选择更适用于某个特定问题的模型。

5.1 损失函数和衡量指标

损失函数是模型训练时优化的目标，而衡量指标可以为我们提供关于模型有效程度的信息。模型训练时，目标是尽量降低所选损失函数；而人工选择和优化模型时，目标是结合衡量指标所提供信息，最大化模型预测收益。

本节讲解如何根据问题本质选择合适的损失函数和衡量指标（简称函数和指标），以及常用的函数和指标。不同的实际应用中，可能涉及更多函数和指标，又或可以根据问题需求自己定义函数和指标，但大多数情况下，使用本节中讲解的内容可以训练出效果不错的模型。在需要自定义函数和指标时，也可以根据本节所讲述的选择思路搭建。

优化模型之前，首先需要从问题的定义出发，确定衡量指标。在分类问题和回归问题中，可供选择的衡量指标亦不相同。

5.1.1 分类问题的衡量指标

第4章使用“准确率”一词泛指衡量指标。分类问题中，准确率（accuracy）的定义为

$$\text{准确率} = \frac{\text{预测值等于真实取值的数据个数}}{\text{所有预测数据个数}} \tag{5.1}$$

这一衡量指标适用于许多分类问题，直观地展示模型预测正确的概率，但准确率这一指标并不适用于所有分类问题。

从简单的二分类问题出发，举一个直观的例子——类别不平衡问题。若问题中负类别占总样本数的95%，正类别仅占5%，一个朴素的、将所有输入皆预测为0的模型就可以取得高达95%的准确率，而这样的模型显然不能满足我们的分类需求。这时，想象一个训练有效的模型，当输入真实目标取值为正类别数据时，其预测准确率达到90%；当输入真实目标取值为负类别数据时，其预测准确率同样达到90%。这个模型整体的准确率将为90%，低于朴素模型95%的准确率。在这种情况下，对比准确率的高低无法直接衡量模型的好坏。

若类别相对平衡，准确率是否能成为万能的衡量指标呢？答案是否定的。假设数据集中包含某疾病患者初次就诊时的身体状况，以及其是否在3年内再次因该疾病就诊，且3年内复发的占比为50%。我们想要根据该历史数据预测未来初次就诊的患者的病症是否会在3年内复发，并增加对可能复发疾病患者的身体情况跟进。假设两个预测模型的整体准确率皆达到85%，分别称二者为模型A和模型B。当输入真实目标取值为正类别（患者疾病会复发）数据时，模型A的预测准确率为95%，模型B的预测准确率为75%；当输入真实目标取值为负类别数据时，模型A的预测准确率为75%，模型B的预测准确率为95%。那么当模型A预测错误时，多数情况是将病情不会复发的患者预测为病情会复发，增加了健康跟进；当模型B预测错误时，多数情况是将病情会复发的患者预测为病情不会复发，省略了必要的健康跟进。然而，我们无法从准确率的高低中获取这一重要差异。

从模型A和模型B的对比中可以看出，设定衡量指标时，应该从预测后果的角度出发，例如“预测正确所能带来的收益”和“预测错误将承担的风险”，寻找可以为该后果提供有效度量的指标。这并不说明准确率这一指标完全无用，准确率在一定程度上还是可以说明模型整体的有效程度，但在更多衡量指标的结合评估下，我们可以更全面地了解模型的预测能力。

首先介绍4个二分类问题中常用于辅助准确率的衡量指标：精度（precision）、召回率（recall）、F1值和特效率（specificity）。回顾2.1.3节中介绍的混合矩阵，其各数值含义如表5.1所示。

表5.1 混合矩阵中各数值含义

真实数值	预测数值为0	预测数值为1
0	真阴性数量	假阳性数量
1	假阴性数量	真阳性数量

精度的公式定义为

$$\text{精度}=\frac{\text{真阳性数量}}{\text{真阳性数量}+\text{假阳性数量}}=\frac{\text{真阳性数量}}{\text{预测数值为1的总个数}} \tag{5.2}$$

在混合矩阵中，精度为右下角数值于右列数值之和的占比。当预测问题中真阳性所带收益较大，而假阳性所带损失也较严重时，可以参考精度，衡量利弊。例如在一个视频推荐算法中，预测用户是否喜欢某个视频。真阳性的收益在于吸引该用户继续使用平台，而过多的假阳性可能会导致用户放弃使用该平台，因此，预测这一问题的模型应该达到较高的精度。

召回率也称敏感度（sensitivity），其公式定义为

$$\text{召回率}=\frac{\text{真阳性数量}}{\text{真阳性数量}+\text{假阴性数量}}=\frac{\text{真阳性数量}}{\text{真实数值为1的总个数}} \tag{5.3}$$

在混合矩阵中，召回率为右下角数值于下行数值之和的占比。当预测问题中假阴性的损失较严重时，可以参考召回，判断该损失是否在可接受范围内，以此判断模型能否被实际应用。例如在预测病人是否会疾病复发的例子中，将一个真阳性患者错误预测成阴性可能会省略必要的健康跟进，而将一个真阴性患者错误预测成阳性会增加不必要的健康跟进。相比两种类别预测错误的后果，也许我们希望尽可能提高召回率。

有些情况下，单纯使用精度或召回率还不足以说明模型的效益。在介绍精度时举的例

子中，也许视频数据库中存储了上万个视频，因此我们并不在意模型拥有较低的召回率，但同时我们也不希望召回率过低，导致模型仅能为用户推荐寥寥无几的视频。在介绍召回率时举的例子中，也许我们并不在意稍微花费人力，去跟进一部分病情不会复发的患者的健康情况，但我们也不需要一个永远将预测为 1 的模型，尽管其召回率将达到 100%。F1 衡量指标结合了对高精度和高召回率的需求，其公式定义为

$$\mathrm{F1}=\frac{2\times \text{精度}\times \text{召回率}}{\text{精度}+\text{召回率}} \tag{5.4}$$

当精度和召回率皆为最高值 1.0 时，F1 亦为最高值，$\mathrm{F1}=\frac{2.0}{2.0}=1.0$；当精度或召回率中的一者为 0 时，不论另一者为何取值，式(5.4)的分子皆为 0，F1 皆为最低值 0。F1 仅在精度和召回率皆偏高时达到较高值，若其中一者偏低，对应的 F1 取值也将较低，因此，使用 F1 衡量指标可以有效结合对高精度和高召回率的需求。

特效率的公式定义为

$$\text{特效率}=\frac{\text{真阴性数量}}{\text{真阴性数量}+\text{假阳性数量}}=\frac{\text{真阴性数量}}{\text{真实数值为 0 的总个数}} \tag{5.5}$$

在混合矩阵中，特效率为左上角数值于上行数值之和的占比。类似于召回率，当预测问题中假阳性的损失较严重时，可以参考特效率，判断该损失是否在可接受范围内，以此判断模型能否被实际应用。

模型的有效性也可以根据绘图的方式分析。常用的接受者操作特征曲线(receiver operating characteristic curve)，也称 ROC 曲线，其横坐标为假阳率(false positive rate，FPR)，表达式为

$$\text{假阳率}=\frac{\text{假阳性数量}}{\text{真阴性数量}+\text{假阳性数量}}=\frac{\text{假阳性数量}}{\text{真实数值为 0 的总个数}} \tag{5.6}$$

纵坐标为真阳率(True Positive Rate，TPR)，表达式为

$$\text{真阳率}=\frac{\text{真阳性数量}}{\text{假阴性数量}+\text{真阳性数量}}=\frac{\text{真阳性数量}}{\text{真实数值为 1 的总个数}} \tag{5.7}$$

曲线图如图 5.1 所示。

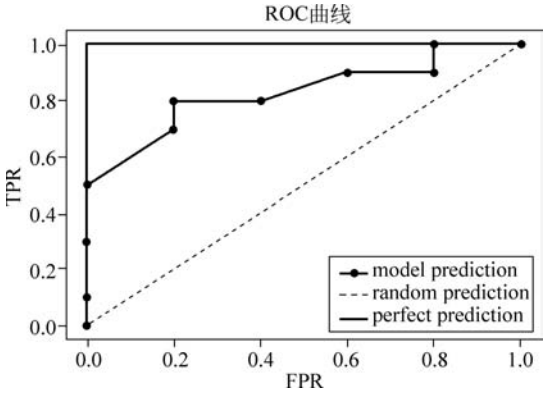

图 5.1　ROC 曲线示意图

图 5.1 中包含 3 种模型的 ROC 曲线。其中，虚线为一个预测完全随机的模型，对应一

条穿过原点的直线；不带任何圆形节点的实线为一个预测完全准确的模型，对应一个从原点竖直上升，而后平行于横轴向右的曲线；带有圆形节点的实线为大多数现实中可能遇到的模型，位于前两种线条之间。

预测二分类问题的模型输出为该数据点为 0 或 1 的概率，而后使用一个阈值，将概率转换为一个或 0 或 1 的预测值。多数情况下，默认的阈值为 0 和 1 的平均值 0.5，但这一阈值可以根据问题的需要调整。ROC 曲线中每个点对应的是使用不同阈值所取得的假阳率和真阳率。由图 5.1 中的 3 个模型曲线对比可以看出，预测能力更强的模型对应的曲线下面积越大，而 ROC 曲线下方面积的大小也确实是一种衡量模型预测能力的指标，简称 AUC (area under curve)。不同模型对应的 ROC 无法使用计算机直接对比，需要人为观察，但 AUC 作为一个标量数值，有直观的高低之分。从某种角度看，AUC 是一个概述 ROC 曲线的数值。由于 ROC 曲线使用不同阈值绘制，AUC 这一衡量指标将不受阈值的影响。

使用 sklearn 的 metrics.roc_curve 函数绘制 ROC 曲线时，需输入目标真实取值和模型所预测的数据点为正类别的概率。执行：

```
#Chapter5/classification_metrics.ipynb

import numpy as np
import matplotlib.pyplot as plt
from sklearn import metrics

#目标值
y = np.array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
          1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
#模型预测每个点为阳性的概率
model_pred_probabilities = np.array([0.01, 0.3, 0.6, 0.4, 0.45,
                                     0.03, 0.2, 0.55, 0.3, 0.24,
                                     0.9, 0.75, 0.3, 0.51, 0.6,
                                     0.7, 0.55, 0.1, 0.9, 1.0])
#一个毫无预测能力的模型预测的概率——全部为 50%
random_pred = np.array([0.5] * 20)
#一个完全预测正确的模型
perfect_pred = np.array([0.0] * 10 + [1.0] * 10)

fpr_model, tpr_model, thresholds_model = \
            metrics.roc_curve(y, model_pred_probabilities, pos_label = 1)
fpr_rand, tpr_rand, thresholds_rand = \
            metrics.roc_curve(y, random_pred, pos_label = 1)
fpr_perfect, tpr_perfect, thresholds_perfect = \
            metrics.roc_curve(y, perfect_pred, pos_label = 1)

#绘制 ROC 曲线
plt.plot(fpr_model, tpr_model, marker = 'o', label = 'model prediction')
plt.plot(fpr_rand, tpr_rand, linestyle = ': ', label = 'random prediction')
plt.plot(fpr_perfect, tpr_perfect, label = 'perfect prediction')
plt.xlabel('FPR')
plt.ylabel('TPR')
```

```
plt.title('ROC curve')
plt.legend()
plt.show()
```

输出为图 5.1 所示的 3 条 ROC 曲线。metrics.roc_curve 函数包含 3 项输出，其中第 3 项为 ROC 曲线中使用的所有阈值，第 1 项为该模型使用不同阈值所对应的 FPR，第 2 项为该模型使用不同阈值所对应的 TPR。使用 FPR 和 TPR，我们可以进一步使用 sklearn 中的 metrics.auc 函数计算 AUC，在下一个 cell 中执行：

```
#在 metrics.auc 函数中的第 1 项输入不同阈值对应的 FPR
#第 2 项输入不同阈值对应的 TPR
print('完全随机模型的 AUC: ', metrics.auc(fpr_rand, tpr_rand))
print('贴近现实模型的 AUC: ', metrics.auc(fpr_model, tpr_model))
print('完美模型的 AUC: ', metrics.auc(fpr_perfect, tpr_perfect))
```

输出结果如下：

```
完全随机模型的 AUC: 0.5
贴近现实模型的 AUC: 0.8300000000000001
完美模型的 AUC: 1.0
```

由此可见，AUC 取值范围为[0.5, 1]，整体预测能力越强的模型对应越高的 AUC。这里提一个技术细节：由于数据点数量有限，使用的阈值数也有限，最终的 ROC 曲线其实是由几个确定的点和连接点的直线构成的。曲线中真正确定的点数量有限，而算法假设确定点之间的 ROC 曲线可以由直线估算。AUC 的计算也基于这一估算之上。

与 ROC 类似的曲线还有精度召回率曲线(precision recall curve)，又称 PR 曲线。PR 曲线同样使用不同的阈值，计算模型在该阈值下的精度和召回率，并以召回率作为横轴，精度作为纵轴进行绘制。使用 sklearn 中的 metrics.precision_recall_curve 函数可以进行绘制，其使用方法与 metrics.roc_curve 函数类似，在下一个 cell 中执行：

```
#Chapter5/ classification_metrics.ipynb

from sklearn.metrics import precision_recall_curve
precision_model, recall_model, thresholds_model = \
        metrics.precision_recall_curve(y, model_pred_probabilities, pos_label = 1)
precision_rand, recall_rand, thresholds_rand = \
        metrics.precision_recall_curve(y, random_pred, pos_label = 1)
precision_perfect, recall_perfect, thresholds_perfect = \
        metrics.precision_recall_curve(y, perfect_pred, pos_label = 1)

#绘制 PR 曲线
plt.plot(recall_model, precision_model, marker = 'o', label = 'model prediction')
plt.plot(recall_rand, precision_rand, linestyle = ': ', label = 'random prediction')
plt.plot(recall_perfect, precision_perfect, label = 'perfect prediction')
plt.xlabel('Recall')
```

```
plt.ylabel('precision')
plt.title('precision recall curve')
plt.legend()
plt.show()
```

显示结果如图 5.2 所示。

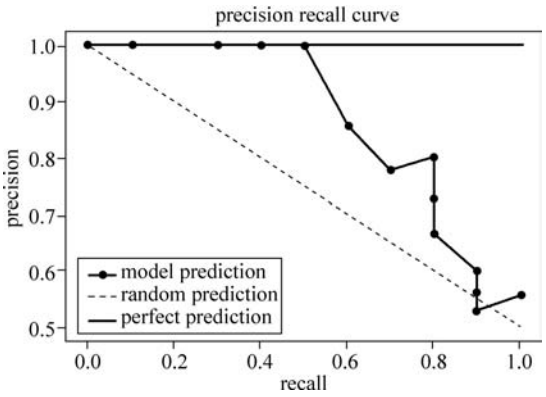

图 5.2 代码输出

其中，虚线为一个预测完全随机的模型，对应一条穿过点(0,1)的直线；不带任何圆形节点的实线为一个预测完全准确的模型，对应一个平行于横轴向右，纵轴值为 1 的直线；带有圆形节点的实线为大多数现实中可能遇到的模型，多数情况下位于前两种线条之间，但也有可能偶尔低于随机模型所对应的直线。同样，预测能力更强的模型对应的曲线下面积越大。PR 曲线下的面积名为 AUPRC(area under precision recall curve)。正如 AUC 概述了 ROC 曲线，可以将 AUPRC 看作一个概述 PR 曲线的标量数值。在 AUC 的计算中，sklearn 做出了点与点之间为直线连接的估算，而 sklearn 的文档中表明，直线连接这一估算放在 AUPRC 的计算中会导致过于乐观的数值结果，因此，sklearn 不对 AUPRC 进行直接计算，而是使用平均精度(average precision)这一标量数值概述 PR 曲线。平均精度的计算公式为

$$\text{平均精度} = \sum_n (R_n - R_{n-1}) P_n \tag{5.8}$$

其中，R_n 为第 n 个阈值所对应的召回率，P_n 为第 n 个阈值所对应的精度。本质上，平均精度是一个以召回率作为权重值的精度加权和。

使用 sklearn 中的 metrics.average_precision_score 可以计算平均精度，在下一个 cell 中执行：

```
#Chapter5/m classification_etrics.ipynb

#在 metrics.average_precision_score 函数中的第 1 项输入目标真实取值
#第 2 项输入模型预测该数据点为正类的概率
print('完全随机模型的平均精度：',
      metrics.average_precision_score(y, random_pred))
print('贴近现实模型的平均精度：',
```

```
        metrics.average_precision_score(y, model_pred_probabilities))
print('完美模型的平均精度: ',
        metrics.average_precision_score(y, perfect_pred))
```

输出如下：

```
完全随机模型的平均精度: 0.5
贴近现实模型的平均精度: 0.859047619047619
完美模型的平均精度: 1.0
```

需要提醒的是，尽管 AUC、AUPRC 和平均精度皆为不根据阈值变化的衡量指标，但在某些应用中，也许我们愿意在一定程度上牺牲召回率，提高精度，或反之。举个例子，某着重于精度提高的二分类中，模型 A 的平均精度为 0.75，而同一问题中模型 B 的平均精度为 0.7。也许我们愿意在保证精度高于 0.9 时，允许召回率降至任何不低于 0.5 的数值。单纯从模型 A 和 B 的平均精度来看，并不能直接说明是否存在能使该模型同时满足精度高于 0.9 且召回率不低于 0.5 这两个条件。在这种问题中，分别分析精度和召回率比分析平均精度更为合适。

5.1.2 回归问题的衡量指标

回归问题中的目标为连续变量，因此，一个常用的衡量指标是预测值与目标值之间的均方差，也是第 4 章所用过的 MSE，其计算公式为

$$\mathrm{MSE}=\frac{1}{N}\sum_{i=1}^{N}(y_i-f(x_i))^2 \tag{5.9}$$

其中，y_i 为第 i 个数据点的真实目标取值，$f(x_i)$ 为模型针对第 i 个数据点的预测值，N 为预测数据点总数。

另一个计算预测值与目标值之间距离的衡量指标是绝对平均误差（mean absolute error，MAE），其计算公式为

$$\mathrm{MAE}=\frac{1}{N}\sum_{i=1}^{N}|y_i-f(x_i)| \tag{5.10}$$

相比 MAE，使用 MSE 会更大程度地惩罚较大的误差，较大的误差进行平方处理后会更为显著。举个例子，假设 $N=2$，考虑两种模型预测结果：若使用模型 A，$y_1-f_{\mathrm{A}}(x_1)=0.5$，$y_2-f_{\mathrm{A}}(x_2)=-8.5$；若使用模型 B，$y_1-f_{\mathrm{B}}(x_1)=4.5$，$y_2-f_{\mathrm{B}}(x_2)=-5.5$，那么

$$\mathrm{MSE}_{\mathrm{A}}=\frac{1}{2}(0.5^2+(-8.5)^2)=36.25 \tag{5.11}$$

$$\mathrm{MAE}_{\mathrm{A}}=\frac{1}{2}(|0.5|+|-8.5|)=4.5 \tag{5.12}$$

$$\mathrm{MSE}_{\mathrm{B}}=\frac{1}{2}(4.5^2+(-5.5)^2)=25.25 \tag{5.13}$$

$$\mathrm{MAE}_{\mathrm{B}}=\frac{1}{2}(|4.5|+|-5.5|)=5 \tag{5.14}$$

如果使用 MAE 作为衡量指标，那么模型 A 是相比 MAE 更小的模型，为更优选择；如果使

用 MSE 作为衡量指标，那么模型 B 是相比 MAE 更小的模型，为更优选择。MSE 和 MAE 之间的选择取决于我们是否愿意接受某些数据点误差较大，从而相应地在某些数据点的预测上取得非常小的误差。如果答案是肯定的，那么应选择 MAE 作为衡量指标，反之则选择 MSE 作为衡量指标。另外，当数据中存在明显的离群值(outlier)时，MAE 相较 MSE 更加稳健，不易因离群值的存在而飞速上升。

实践中常使用 MSE 的一种变形式——均方根误差(root mean squared error，RMSE)。其表达式为

$$\mathrm{RMSE}=\sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i-f(x_i))^2} \tag{5.15}$$

式(5.15)在 MSE 的基础上简单地开了一个二次根号，这样可以保证误差和预测目标的单位相同，即与 MSE 本质效果相同。

MSE、MAE 和 RMSE 中存在一个共同点，3 个指标均不在乎预测值与目标之间的大小关系。不论 $y_i-f(x_i)$ 是正数或负数并不影响以上 3 种指标的数值，但在某些应用中，也许欠预测($y_i>f(x_i)$)所带来的实际损失大于过预测($y_i<f(x_i)$)。这类情况可以使用均方根对数误差(Root Mean Squared Log Error，RMSLE)，其表达式为

$$\mathrm{RMSLE}=\sqrt{\frac{1}{N}\sum_{i=1}^{N}(\log(y_i+1)-\log(f(x_i)+1))^2} \tag{5.16}$$

举个例子展示 RMSLE 和 RMSE 在面对欠预测和过预测时的差异：假设 $N=1$，$y_1=100$，使用模型 C 的预测结果为 $f_{\mathrm{C}}(x_1)=75$，使用模型 D 的预测结果为 $f_{\mathrm{D}}(x_1)=125$，那么

$$\mathrm{RMSE}_{\mathrm{C}}=\sqrt{25^2}=25 \tag{5.17}$$

$$\mathrm{RMSLE}_{\mathrm{C}}=\sqrt{(\log(100+1)-\log(75+1))^2}=0.284 \tag{5.18}$$

$$\mathrm{RMSE}_{\mathrm{D}}=\sqrt{25^2}=25 \tag{5.19}$$

$$\mathrm{RMSLE}_{\mathrm{D}}=\sqrt{(\log(100+1)-\log(125+1))^2}=0.221 \tag{5.20}$$

两个模型所得 RMSE 完全相等，但由于模型 C 对 x_1 的预测为欠预测，而模型 D 为过预测，因此模型 D 所得 RMSLE 较低。

MSE、MAE、RMSE 和 RMSLE 可以用来对比不同模型之间的优劣关系，但四者皆无法直接说明单个模型的优劣。接下来介绍的 R 方(R-squared)和调整 R 方(adjusted R-squared)指标，可以提供一个关于单个模型优劣的数值衡量。

R 方的数学表达式为

$$R^2=1-\frac{\frac{1}{N}\sum_{i=1}^{N}(y_i-f(x_i))^2}{\frac{1}{N}\sum_{i=1}^{N}(y_i-\bar{y})^2} \tag{5.21}$$

其中，$\bar{y}=\frac{1}{N}\sum_{i=1}^{N}y_i$。不难看出，$R$ 方表达式第 2 项的分子为 MSE，分母为真实目标值的方差。R 方的取值范围为[0，1]，MSE 越大意味着 R^2 越小，同时也意味着模型的预测力越差。在 R 方的基础上，调整 R 方的计算公式中考虑到模型特征的数量，其表达式为

$$R^2_{\text{adj}} = 1 - (1 - R^2)\frac{N-1}{N-D-1} \tag{5.22}$$

其中，D 为总特征数。R^2_{adj} 一定小于或等于 R^2。使用 R 方的问题在于，每个新增的特征都将会提升 R 方，尽管新增的特征并不在已有的特征基础上帮助模型预测目标。使用 Python 举一个例子，首先创建一个简单的回归问题数据集，执行：

```
#Chapter5/regression_metrics.ipynb

import pandas as pd
import numpy as np

np.random.seed(42)
#目标值 y 与 x1 的关系为 y = 2 * x1,存在少量噪声
df = pd.DataFrame({'x1': [1, 3, 2, 5, 9],
                   'y': [2, 6, 4.1, 10, 18.1]})

#特征 x2 不包含更多信息,为随机数值
df['x2'] = np.random.rand(5)
df = df[['x1', 'x2', 'y']]#为列重新排序,特征于目标左侧

display(df)
```

	x1	x2	y
0	1	0.374540	2.0
1	3	0.950714	6.0
2	2	0.731994	4.1
3	5	0.598658	10.0
4	9	0.156019	18.1

图 5.3　代码输出

显示结果如图 5.3 所示。

使用 sklearn 中的直线回归模型 LinearRegression 进行训练和预测，使用 metrics.r2_score 函数进行 R 方的计算，并使用自定义的函数进行调整 R 方的计算，在下一个 cell 中执行：

```
#Chapter5/regression_metrics.ipynb

from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score

def adjusted_R2(R2, N, D):
    '''返回调整 R 方数值
输入:
        R2: R 方数值
        N: 预测点总数
        D: 特征数
    输出:
        调整 R 方
    '''
    return 1 - (1-R2) * (N-1)/(N-D-1)

lr_1 = LinearRegression()     #不使用特征 x2 训练
lr_2 = LinearRegression()     #使用特征 x2 训练
```

```
lr_1.fit(df[['x1']], df['y'])
lr_2.fit(df[['x1', 'x2']], df['y'])
y_pred_1 = lr_1.predict(df[['x1']])
y_pred_2 = lr_2.predict(df[['x1', 'x2']])
y_true = list(df['y'])
R2_1 = r2_score(y_true, y_pred_1)
R2_2 = r2_score(y_true, y_pred_2)
print('不使用特征 x2 所得 R 方: ', R2_1)
print('使用特征 x2 所得 R 方: ', R2_2)
print('不使用特征 x2 所得调整 R 方: ', adjusted_R2(R2_1, len(y_true), 1))
print('使用特征 x2 所得调整 R 方: ', adjusted_R2(R2_2, len(y_true), 2))
```

输出如下：

```
不使用特征 x2 所得 R 方: 0.9999395206312185
使用特征 x2 所得 R 方: 0.9999408313812255
不使用特征 x2 所得调整 R 方: 0.9999193608416247
使用特征 x2 所得调整 R 方: 0.9998816627624509
```

由此输出可见，加入 $x2$ 后，R 方从 0.9999395206312185 上升至 0.9999408313812255，而调整 R 方从 0.9999193608416247 降至 0.9998816627624509。加入随机特征 $x2$ 后，调整 R 方的改变方向相较 R 方更加合理。

MSE、MAE、RMSE 和 RMSLE 皆可以使用 sklearn 自带函数计算，使用上段代码中的模型输出和真实目标值，在下一个 cell 中执行：

```
#Chapter5/regression_metrics.ipynb

from sklearn.metrics import mean_squared_error,\
                            mean_absolute_error,\
                            mean_squared_log_error

#计算 lr_1 模型输出各个衡量指标下的数值
#当函数 mean_squared_error 中的参数 squared = True 时计算 MSE
#当 squared = False 时计算 RMSE,默认值为 squared = True
print('MSE: ', mean_squared_error(y_true, y_pred_1, squared = True))
print('MAE: ', mean_absolute_error(y_true, y_pred_1))
print('RMSE: ', mean_squared_error(y_true, y_pred_1, squared = False))
print('RMSLE: ', mean_squared_log_error(y_true, y_pred_1))
```

输出如下：

```
MSE: 0.0019500000000000181
MAE: 0.03900000000000041
RMSE: 0.04415880433163944
RMSLE: 5.894588133682514e-05
```

5.1.3 损失函数

5.1.1 节和 5.1.2 节讲解了如何提供方便人工解读模型效益的数值，本节讲解如何为模型设定合适的优化目标。

回归问题中，5.1.2 节中讲解的衡量指标也可以用作损失函数。最常用的损失函数包含 MSE、MAE、RMSE 和 RMSLE，四者之间的选择与 5.1.2 节中四者作为衡量指标的选择方法相同。

分类问题中，最常用的损失函数是 4.5.2 节介绍的交叉熵损失函数。回顾 4.5.2 节中交叉熵损失函数的公式：

$$L(g,y)=\begin{cases}-\log(g), & y=1\\ -\log(1-g), & y=0\end{cases} \tag{5.23}$$

$$g=\sigma(z)=\frac{1}{1+\mathrm{e}^{-z}} \tag{5.24}$$

$$z=\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}+b \tag{5.25}$$

其中，$\boldsymbol{w}$ 为权重，b 为偏移项。使用交叉熵损失函数，解决了 Sigmoid 函数中偏导数易消失这一问题。式(5.23)～式(5.25)仅适用于二分类问题，面对多分类问题时，式(5.24)中的 Sigmoid 函数将被 Softmax 函数替代，回归算法与逻辑斯蒂回归类似，被称为 Softmax 回归。假设某多分类问题中类别总数为 K，且模型输出为一个长度为 K 的向量，向量中的每项代表模型预测数据点属于该类的概率。Softmax 回归中输入与输出之间的关系如式(5.26)～式(5.28)所示：

$$\boldsymbol{z}=\boldsymbol{w}\boldsymbol{x}+\boldsymbol{b} \tag{5.26}$$

$$g_k=\mathrm{Softmax}(z_1,z_2,\cdots,z_K)_k=\frac{\mathrm{e}^{z_k}}{\sum\limits_{k'}\mathrm{e}^{z_{k'}}} \tag{5.27}$$

$$L(\boldsymbol{g},\boldsymbol{y})=-\sum_{k=1}^{K}y_k\log(g_k)=-\boldsymbol{y}^{\mathrm{T}}\log(\boldsymbol{g}) \tag{5.28}$$

式(5.26)为 Softmax 函数的表达式，式(5.27)为多分类问题中交叉熵损失函数的表达式。

合页损失(hinge loss)是分类问题中另一个常见的损失函数，假设正类别用数值+1 表示，负类别用数值−1 表示，合页损失数学表达式为

$$L(g,y)=\max(0,1-g\cdot y) \tag{5.29}$$

其图像如图 5.4 所示，就像正在翻合的书页。

图 5.4 以 $g\cdot y$ 作为横轴，绘制了不同 y 和 g 之间距离所对应的合页损失。$g\cdot y$ 可以看作真实目标值 y 和模型预测值 g 之间的距离。由于正类别使用数值+1 表示，负类别使用数值−1 表示，而模型输出范围为实数，一个合理的阈值为 0。若 $y\geqslant 0$，预测结果设定为+1；若 $y<0$，预测结果设定为−1。直觉上看，当预测值 g 与 y 的符号相同时，使用 0 这一阈值可以得到正确的类别预测。换言之，当 $g\cdot y>0$ 时，模型将预测正确类别，但合页损失在 $g\cdot y$ 略大于 0 时仍贡献少量损失数值，并随着 $g\cdot y$ 的降低呈直线上升，只有在 $g\cdot y\geqslant 1$ 时，合页损失值降至 0。

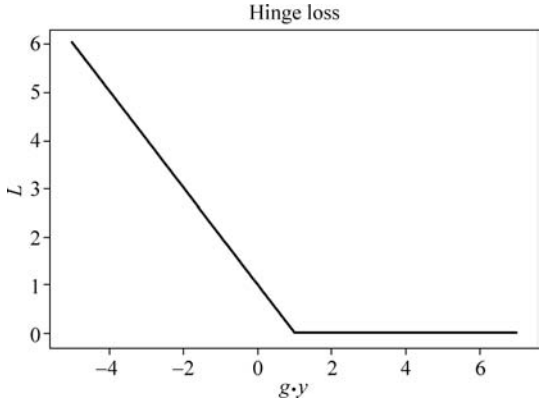

图 5.4 合页损失函数示意图

由这一特点可以看出，合页损失鼓励模型对应的决策边界尽可能远离数据点。我们可以在 $g \geqslant 0$ 时将 g 看作模型预测数据点类别为正类的确信程度，在 $g < 0$ 时将 g 看作模型预测数据点类别为负类的确信程度，那么合页损失在惩罚预测错误类别的同时，同样惩罚了预测正确但对于预测结果不确信的模型。

5.2 K 折交叉验证

3.2 节讲解了训练集、验证集和测试集在模型预测中的分工。其中，验证集起到模拟测试集的作用，在隐藏测试集真实目标值的前提下，模型在验证集上的表现可以用来预估其在测试集上的表现。

回顾 4.8.1 节使用随机森林预测波士顿房价的例子。这个例子中使用 MSE 作为损失函数和衡量指标，最终模型在 3 个集上取得的 MSE 分别为

```
训练集 MSE: 8.262145505468627
验证集 MSE: 10.079003086534906
测试集 MSE: 21.377303939606175
```

例子中，虽然验证集和测试集都是从历史数据集中随机分割出来的 15% 数据点，但 MSE 有较为明显的差异。这是因为，随机分割的验证集可能恰好比测试集的分布更加接近训练集，而由于模型根据训练集建立，其在验证集上的表现自然优于测试集，但 MSE 在验证集和测试集上的差异会导致我们无法准确预估模型对未来数据的预测能力。

使用 K 折交叉验证（K-fold cross-validation）可以有效缓解这一问题。K 折交叉验证使用 k 个不同的训练集和验证集，获得 k 个不同的训练集下模型所得验证集的 MSE，或其他衡量指标数值。K 折交叉验证的过程如下。

（1）从总历史数据集中分割出测试集，称测试集以外的数据点集合为交叉验证集 S。

（2）将交叉验证集分为 k 个子集，称为 $S_1, S_2, \cdots, S_k$。

（3）使用 S_1 作为验证集，S/S_1 作为训练集，训练模型并记录模型在验证集的 MSE，或其他衡量指标数值。

(4) 重复第 3 步,并将 S_1 改为 S_2,S_3,…,S_k,直至 k 个子集中的每个子集都被选作过验证集。

(5) 计算 k 个子集对应的验证集 MSE 的平均值。

k 个验证集的 MSE 平均数可以更有效地说明模型面对未知数据的预测力。sklearn 官网对于 K 折交叉验证的图示如图 5.5 所示。

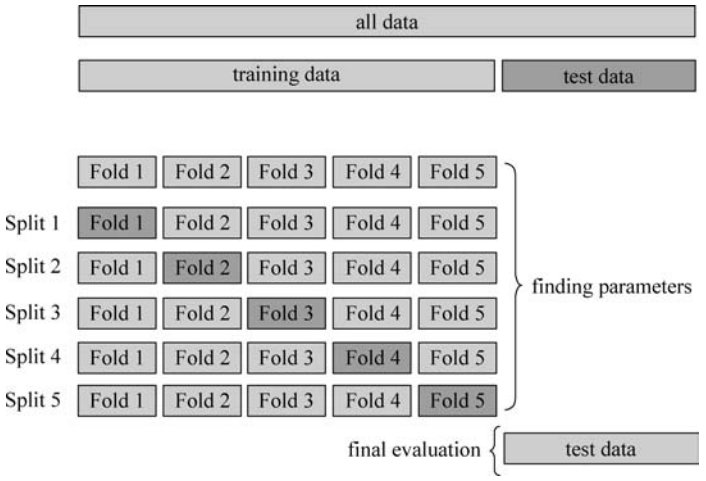

图 5.5 交叉验证图示(图片来源于 sklearn 官网)

图 5.5 中,all data(全部数据)为完整历史数据集。根据 K 折交叉验证过程中的第 1 步,完整历史数据集被分为 test data(测试数据)和 training data(训练数据)。图 5.5 中划为训练集的部分与前文中提到的交叉验证集 S 对应。虽然测试集以外的数据点集合称为 training data,但为了避免和“用于模型训练的数据集”这一定义冲突,下文将继续称这一部分数据为交叉验证集。图 5.5 中使用的折(Fold)为 5。根据 K 折交叉验证过程中的第 2 步,交叉验证集被分为 $k=5$ 个子集,Fold 1(第 1 折)到 Fold 5(第 5 折)分别对应 S_1,S_2,…,S_5。Split 1(分割 1)至 Split 5(分割 5)为 5 次不同验证集的选择,每次选择中,Fold 1(第 1 折)到 Fold 5(第 5 折)被分别选作验证集,其余折作为该次分割的训练集,进行模型训练和评估。Split 1 至 Split 5 这一流程对应 K 折交叉验证过程中的第 3 和 4 步。图 5.5 在 5 次分割旁边注释到,这个步骤可以用于 finding parameters,也就是调参,5.3 节将详细讲解如何调参。

回到 4.8.1 节随机森林预测的波士顿房价问题,并使用 sklearn 中基础的 model_selection.KFold 函数进行分割及交叉验证,执行:

```
#Chapter5/cross-validation.ipynb

import pandas as pd
from sklearn.datasets import load_boston
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import KFold
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
```

```
#读取数据集,数据特征详情可参考 4.6 节
boston_dataset = load_boston()
df = pd.DataFrame(boston_dataset['data'])
df.columns = boston_dataset['feature_names']
df['price'] = boston_dataset['target']

#分割交叉验证集(占 85%)和测试集(占 15%)
df_train_val, df_test = train_test_split(df, test_size = 0.15, random_state = 42)
df_train_val.reset_index(drop = True, inplace = True)#方便索引

#n_splits 用于控制 k 值
kf = KFold(n_splits = 5)
#初始化一个空的 Python 列表,存储每次交叉验证分割后所得验证集 MSE
mse_lst = []
#循环每个训练集和验证集的组合
#kf.split 函数输出为数据点指数,而非数据点本身,因此需要根据指数索引取得数据点
for train_index, val_index in kf.split(df_train_val):
    #使用 iloc 进行指数索引
    df_train, df_val = df_train_val.iloc[train_index],\
                       df_train_val.iloc[val_index]
    #初始化一个随机森林模型
    #参数设置为 n_estimators = 10: 森林中使用 10 棵决策树
    #max_depth = 5: 每棵决策树最深允许延伸 5 层
    #max_features = 'sqrt': 每个节点将从大小为总特征数平方根的特征子集选择分割特征
    reg = RandomForestRegressor(n_estimators = 10,
                                max_depth = 5,
                                max_features = 'sqrt',
                                random_state = 42)
    #训练随机森林
    reg.fit(df_train.drop(columns = ['price']), df_train['price'])

    #预测评估
    pred_val = reg.predict(df_val.drop(columns = ['price']))
    mse_val = mean_squared_error(list(df_val['price']), pred_val)
    mse_lst.append(mse_val)
```

上段代码的执行将完成 K 折交叉验证过程中的第 1～4 步,得到一个包含 5 次分割分别对应的验证集 MSE 的 Python 列表。分析此列表,在下一个 cell 中执行:

```
#Chapter5/cross - validation.ipynb

import numpy as np

mse_lst = np.array(mse_lst)
#分析 mse_lst
print('5 次分割分别对应的验证集 MSE: ', mse_lst)
print('5 次分割的验证集 MSE 的最小值: ', mse_lst.min())
print('5 次分割的验证集 MSE 的最大值: ', mse_lst.max())
print('5 次分割的验证集 MSE 的平均值: ', mse_lst.mean())
print('5 次分割的验证集 MSE 的标准差: ', mse_lst.std())
```

输出如下:

```
5 次分割分别对应的验证集 MSE: [25.33013227 22.2121525  14.36727223 16.55122968 18.60053958]
5 次分割的验证集 MSE 的最小值: 14.367272229658111
5 次分割的验证集 MSE 的最大值: 25.33013227255939
5 次分割的验证集 MSE 的平均值: 19.41226525299553
5 次分割的验证集 MSE 的标准差: 3.9282794287521425
```

从输出中,我们不仅能得出平均 MSE 约为 19.41 这一信息,同时认识到该数据集的不同分割之间 MSE 大致的差异,得出模型预测未来数据时所得 MSE 的大致范围。最后,使用完整的交叉验证集训练模型并对测试集进行预测评估,在下一个 cell 中执行:

```
reg.fit(df_train_val.drop(columns = ['price']), df_train_val['price'])

pred_test = reg.predict(df_test.drop(columns = ['price']))
mse_test = mean_squared_error(list(df_test['price']), pred_test)
print('测试集所得 MSE 为', mse_test)
```

输出如下:

```
测试集所得 MSE 为 7.458895070212933
```

由于执行过 K 折交叉验证,我们可以合理解释测试集 MSE 与 5 个验证集 MSE 平均数之间的差异。不同分割所得 MSE 存在明显差异,因此,模型在验证集上取得的平均 MSE 只能说明,在预测大量未知数据时,可以期待一个均值约为 19.41 的 MSE,但每个数据点预测所得 MSE 可能与均值相差较大。

基础的 KFold 分割将根据数据点的指数顺序将其分割为 k 个折。而面对某些类不平衡的分类问题时,例如负类别占比 80%,正类别占比 20%的二分类问题,也许我们希望分割后的每个训练集和验证集也保持这样的类别占比,以此让每次分割都最大程度地模拟未知数据的期待分布。这种情况下可以使用 sklearn 中的 model_selection.StratifiedKFold 函数,代替基础的 model_selection.KFold 函数。为演示这一函数,首先创建一个负类别占比 80%,正类别占比 20%的二分类 DataFrame,执行:

```
#Chapter5/stratified_split.ipynb

import pandas as pd
import numpy as np

#为对比 KFold 和 StratifiedKFold,直接创建交叉验证集,省略分割测试集的步骤
df_train_val = pd.DataFrame({'x1': np.arange(1, 101),
                             'y': [0] * 80 + [1] * 20})
#打乱顺序
df_train_val = df_train_val.sample(frac = 1).reset_index(drop = True)

print('负类别占比: ', 1 - df_train_val['y'].mean())
print('正类别占比: ', df_train_val['y'].mean())
```

使用 StratifiedKFold 进行分割,并与 KFold 进行对比,在下一个 cell 中执行:

```
#Chapter5/stratified_split.ipynb

from sklearn.model_selection import StratifiedKFold, KFold

#初始化 StratifiedKFold 和 KFold 分割器
skf = StratifiedKFold(n_splits=5)
kf = KFold(n_splits=5)

#使用 StratifiedKFold 进行分割,并打印每次分割的类别比例
#StratifiedKFold.split 需依次输入 X 和 y
X, y = df_train_val[['x1']], df_train_val['y']
count = 1#记录迭代数
for train_index, val_index in skf.split(X, y):
    #使用 iloc 进行指数索引
    train_y, val_y = y.iloc[train_index], y.iloc[val_index]
    print('使用 StratifiedKFold 的第{}次分割'.format(count))
    print('当前分割的训练集中负类别占比:', 1 - train_y.mean())
    print('当前分割的训练集中正类别占比:', train_y.mean())
    print('当前分割的验证集中负类别占比:', 1 - val_y.mean())
    print('当前分割的验证集中正类别占比:', val_y.mean())
    print()
    count += 1

#使用 KFold 进行分割,并打印每次分割后训练集和验证集的类别比例
count = 1#记录迭代数
for train_index, val_index in kf.split(df_train_val):
    #使用 iloc 进行指数索引
    df_train, df_val = df_train_val.iloc[train_index],\
                    df_train_val.iloc[val_index]
    print('使用 KFold 的第{}次分割'.format(count))
    print('当前分割的训练集中负类别占比:', 1 - df_train['y'].mean())
    print('当前分割的训练集中正类别占比:', df_train['y'].mean())
    print('当前分割的验证集中负类别占比:', 1 - df_val['y'].mean())
    print('当前分割的验证集中正类别占比:', df_val['y'].mean())
    print()
    count += 1
```

输出如下：

```
使用 StratifiedKFold 的第 1 次分割
当前分割的训练集中负类别占比: 0.8
当前分割的训练集中正类别占比: 0.2
当前分割的验证集中负类别占比: 0.8
当前分割的验证集中正类别占比: 0.2

使用 StratifiedKFold 的第 2 次分割
当前分割的训练集中负类别占比: 0.8
当前分割的训练集中正类别占比: 0.2
```

```
当前分割的验证集中负类别占比：0.8
当前分割的验证集中正类别占比：0.2

使用 StratifiedKFold 的第 3 次分割
当前分割的训练集中负类别占比：0.8
当前分割的训练集中正类别占比：0.2
当前分割的验证集中负类别占比：0.8
当前分割的验证集中正类别占比：0.2

使用 StratifiedKFold 的第 4 次分割
当前分割的训练集中负类别占比：0.8
当前分割的训练集中正类别占比：0.2
当前分割的验证集中负类别占比：0.8
当前分割的验证集中正类别占比：0.2

使用 StratifiedKFold 的第 5 次分割
当前分割的训练集中负类别占比：0.8
当前分割的训练集中正类别占比：0.2
当前分割的验证集中负类别占比：0.8
当前分割的验证集中正类别占比：0.2

使用 KFold 的第 1 次分割
当前分割的训练集中负类别占比：0.7875
当前分割的训练集中正类别占比：0.2125
当前分割的验证集中负类别占比：0.85
当前分割的验证集中正类别占比：0.15

使用 KFold 的第 2 次分割
当前分割的训练集中负类别占比：0.8
当前分割的训练集中正类别占比：0.2
当前分割的验证集中负类别占比：0.8
当前分割的验证集中正类别占比：0.2

使用 KFold 的第 3 次分割
当前分割的训练集中负类别占比：0.8375
当前分割的训练集中正类别占比：0.1625
当前分割的验证集中负类别占比：0.65
当前分割的验证集中正类别占比：0.35

使用 KFold 的第 4 次分割
当前分割的训练集中负类别占比：0.7875
当前分割的训练集中正类别占比：0.2125
当前分割的验证集中负类别占比：0.85
当前分割的验证集中正类别占比：0.15

使用 KFold 的第 5 次分割
当前分割的训练集中负类别占比：0.7875
当前分割的训练集中正类别占比：0.2125
当前分割的验证集中负类别占比：0.85
当前分割的验证集中正类别占比：0.15
```

由输出可见，使用 StratifiedKFold 分割的每个训练集和验证集的类别比例皆保留了原数据集中的比例，而使用 KFold 分割则无法保证这一比例。

5.3 超参数调试

使用多数机器学习模型时需要根据问题类型和数据特点设定超参数(hyperparameter)，调试超参数这一过程也被简称为调参。不同于权重这类模型训练时学习到的参数，超参数需要在模型开始训练之前设定，某种程度上更加详细地设定了模型结构。

某些模型可调试超参数繁多，调试范围也较广，例如决策树和森林集成类模型。4.6 节深度神经网络、4.8.1 节随机森林、4.8.2 节极端随机树及 4.9.2 节 XGBoost 和 LightGBM 中均使用波士顿房价预测这一回归问题作为示例，进行模型训练、预测和评估。然而，尽管预测的问题和评估指标皆相同，在获得每次模型的评估结果后，我们并没有直接与其他模型的结果进行对比。这是因为第 4 章中每次模型初始化时，超参数的定义仅根据我们对超参数范围的理解和实践经验，并没有经过任何优化。尽管我们尽量赋予不同集成模型中起相似(甚至相同)作用的超参数相同值，但由于模型之间运行原理的差异，这样的设定仍无法公平地选择解决此问题的最优模型。

这就像要求计算机课程班上的 5 位同学在一周内解答一道难题，并根据解题结果选择代表学校参加市级计算机比赛的同学。5 位同学在自己状态最佳的条件下会给出不同的解答。这 5 份不同的解答在课程所制定的衡量标准下有高有低，却分别对应 5 位同学本人根据衡量标准有能力给出的最优解。然而每个同学达到最佳状态的条件不同，有些同学可能需要睡够 8h、早睡早起，有些同学可能需要喝几杯咖啡、熬夜找灵感，有些同学可能需要解题时听钢琴曲。如果强行让每个同学都执行同一种作息，例如规定同学们每天喝一杯咖啡，那么一部分同学的状态将优于其他同学，且更接近自己的最佳状态。这种设定下，无法公平判断每个同学给出的解答是否接近其最佳潜能，也无法评估哪个同学最适合代表学校参加市级计算机比赛。

解决这一问题的方法是让每个同学达到自己的最佳解题状态，并在该状态下做出解答。然而，学生达到最优状态的条件可能不为人知，学生本人和身边的人可能都不完全了解如何发挥该学生最大的潜能，因此，找寻进入最佳解题状态可能需要多次尝试，在合理的起居习惯范围内尝试不同的组合，并评估该组合相伴的状态所对应的解题效果。这里的合理指的是根据对人类起居习惯的理解进行预估。例如睡眠时间的选择中，一个合理的范围可能是 6～9h，而不去考虑或尝试更高或更低的设定，避免在尝试中浪费时间。

同理，判断最优模型之前，需保证该模型达到最适合解决该问题的状态。在模型训练中，这一状态可以使用不同的超参数调试。首先需要在可以调整的超参数中选择合理的调试范围，并组合使用不同超参数范围内的值，训练模型并评估该组合对应的预测效果。多次组合尝试后，便可以大致得到模型预测该问题的最优结果。

本节将使用随机森林、极端随机树和 XGBoost 作为示例进行超参数调试。由于可调试超参数较多、范围较广，基本可以排除人工搜寻组合这一尝试。本节讲解 3 种不同的自动最优超参数搜寻算法。

5.3.1 网格搜索法

网格搜索法(grid search)是一种穷举搜索的方法。在不同超参数的设定范围内,算法将使用所有不同组合,训练对应模型并在验证集上进行评估。算法输出为评估结果最优的模型或前 n 个最优模型所对应的超参数。

网格搜索法中的网格来源于其对超参数组合的搜索方式。以随机森林为例,假设我们想找到最优的 n_estimators、max_depth 和 max_features 的组合,并事先设定每个超参数的范围,那么搜索空间的三维可视化图如图 5.6 所示。

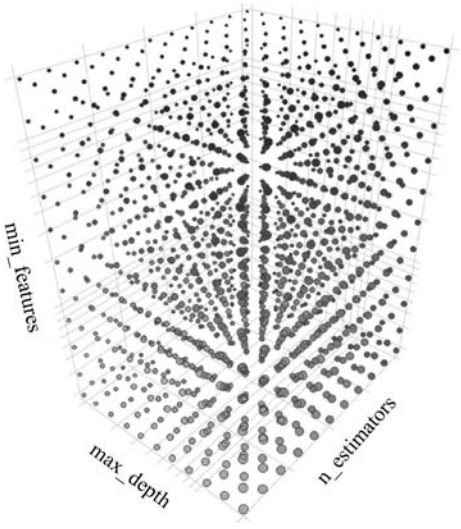

图 5.6 网格搜索空间三维可视化图

空间中的每个点代表一个可能的组合,不同的组合在空间中的位置仿若网格中的格点。

使用网格搜索法前不仅需要设定网格范围,也需设定每个超参数在相应范围内所有值得尝试的取值。当超参数为连续变量时,可以尝试取值范围内间隔相同的 k 个数值;当超参数为离散变量时,可以尝试取值范围内全部数值,或每间隔 k 个数值加入一个数值。

sklearn 中的 model_selection.GridSearchCV 使用 5.2 节中介绍的 K 折交叉验证,在输入的网格中寻找最优超参数组合。K 折交叉验证和网格搜索组合的搜寻过程如下:

(1) 循环每种超参数组合:进行 K 折交叉验证,并存储该超参数组合下的衡量指标信息,如 5.2 节中的平均 MSE 或折之间 MSE 的标准差。

(2) 分析每种组合所对应的衡量指标信息,选择最优组合。

GridSearchCV 函数中需要输入:

(1) estimator:待进行超参数调试的模型。

(2) param_grid:包含所有模型中需调试的超参数,以及序列候选尝试值,用于定义网格中格点位置的 Python 字典。

(3) scoring:指定衡量指标,可为衡量指标对应的特定字符串或一个列表的字符串,默认使用模型自带的衡量指标。

(4) cv:用于设定 K 折交叉验证中的 k 值。

(5) refit：设定是否结束交叉验证时，使用全部的交叉验证集和搜寻到的最优超参数重新训练模型，默认值为 True。当 scoring 为一个字符串列表时，需要在 refit 中输入用于为超参数组合优劣排序的衡量指标。

回顾 4.8.1 节和 4.8.2 节中森林模型的参数：n_estimators＝10、max_depth＝5 及 max_features＝'sqrt'。

在 4.8.2 节的结尾提到，该参数设定下，极端随机树的表现略差于随机森林，但这并不足以说明随机森林在预测波士顿房价问题上优于极端随机树。我们需要分别调试两个模型的参数，并对比其最优表现。根据以上对网格搜索的基本了解，首先对随机森林模型中的 n_estimators、max_depth 和 max_features 进行调试，执行：

```
#Chapter5/grid_search.ipynb

import pandas as pd
from sklearn.datasets import load_boston
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV, train_test_split
import time

#读取数据集，数据特征详情可参考 4.6 节
boston_dataset = load_boston()
df = pd.DataFrame(boston_dataset['data'])
df.columns = boston_dataset['feature_names']
df['price'] = boston_dataset['target']

#初始化一个随机森林模型，暂时不设定特定的超参数，完全使用默认值
reg = RandomForestRegressor(random_state = 42)
#设定每个待调试超参数的候选取值，放入一个 Python 字典，为方便讲解，此处仅使用少量格点
parameters = {'n_estimators': [5, 10, 15],
          'max_depth': [5, 10, 15],
          'max_features': [3, 6, 9]}

#分割交叉验证集(占 85 %)和测试集(占 15 %)
df_train_val, df_test = train_test_split(df, test_size = 0.15, random_state = 42)
df_train_val.reset_index(drop = True, inplace = True) #方便索引

#记录网格搜索总共花费时间
start = time.time()
#初始化一个网格搜索器，使用 - 1 * MSE 作为 scoring
grid_search_cv = GridSearchCV(reg, parameters,
                              scoring = 'neg_mean_squared_error')
grid_search_cv.fit(df_train_val.drop(columns = ['price']),
                   df_train_val['price'])
end = time.time()
print('本次网格搜索耗时: ', round(end - start, 2), 's')
print(grid_search_cv.cv_results_.keys()) #打印 GridSearchCV 存储的信息名称
```

输出如下：

```
本次网格搜索耗时: 2.6 s
dict_keys(['mean_fit_time', 'std_fit_time', 'mean_score_time', 'std_score_time', 'param_max_
depth', 'param_max_features', 'param_n_estimators', 'params', 'split0_test_score', 'split1_test
_score', 'split2_test_score', 'split3_test_score', 'split4_test_score', 'mean_test_score', 'std
_test_score', 'rank_test_score'])
```

这个例子中,在网格搜索和 K 折交叉验证的过程中,GridSearchCV 陆续记录了 16 条信息,并保存在 grid_search_cv.cv_results_字典中。根据 k 值和调试超参数数量的不同,字典中可能记录更多或更少的信息。针对这个例子,这 16 条信息记录的内容如下。

(1) params:超参数组合测试顺序,其他信息皆对应这一顺序排列。

(2) mean_fit_time:每个超参数组合在 K 折交叉验证中的平均训练时间。

(3) std_fit_time:每个超参数组合在 K 折交叉验证中训练时间之间的标准差。

(4) mean_score_time:每个超参数组合在 K 折交叉验证中评估所使用的平均时间。

(5) std_score_time:每个超参数组合在 K 折交叉验证中评估所使用的时间之间的标准差。

(6) param_max_depth:每个组合分别对应的 max_depth 数值。

(7) param_max_features:每个组合分别对应的 max_features 数值。

(8) param_n_estimators:每个组合分别对应的 n_estimators 数值。

(9) split0_test_score,…,split4_test_score:第 1～5 次分割分别对应的验证集负 MSE。sklearn 提供负 MSE 作为衡量指标是为了方便为 mean_test_score 排序。rank_test_score 中,第 1 名的 mean_test_s 核数值最大,而负 MSE 最大的超参数组合也是 MSE 最小的组合,为最优组合。

(10) mean_test_score:5 次分割的验证集平均负 MSE。

(11) std_test_score:5 次分割的验证集负 MSE 标准差。

(12) rank_test_score:超参数根据验证集负 MSE 大小优劣排序。

其中,较为重要的几个信息包括 mean_test_score、std_test_score 和 rank_test_score。打印字典中的这 3 条信息,以及其对应的超参数组合,在下一个 cell 中执行:

```
#Chapter5/grid_search.ipynb

print('\n超参数组合测试顺序: \n',
      grid_search_cv.cv_results_['params'])
print('\n超参数组合优劣排序: \n',
      grid_search_cv.cv_results_['rank_test_score'])
print('\n超参数组合验证集平均 MSE: \n',
       -grid_search_cv.cv_results_['mean_test_score'].round(3))
print('\n超参数组合验证集 MSE 标准差: \n',
      grid_search_cv.cv_results_['std_test_score'].round(3))

#找到排名第一的组合及其 MSE
#由于使用负 MSE 作为衡量指标,因此打印 -1 * mean_test_score
#排名第一的组合为 rank_test_score 最高的组合,也就是负 MSE 最高(MSE 最低)的最优组合
rank_1_index = grid_search_cv.cv_results_['rank_test_score'].argmin()
```

```
print('\n排名第一的超参数组合为\n',
      grid_search_cv.cv_results_['params'][rank_1_index])
print('\n排名第一的超参数组合对应的平均验证集 MSE: \n',
       - grid_search_cv.cv_results_['mean_test_score'][rank_1_index])
print('\n排名第一的超参数组合对应的验证集 MSE 之间的标准差: \n',
      grid_search_cv.cv_results_['std_test_score'][rank_1_index])
```

输出如下：

```
超参数组合测试顺序:
[{'max_depth': 5, 'max_features': 3, 'n_estimators': 5}, {'max_depth': 5, 'max_features': 3, 'n
_estimators': 10}, {'max_depth': 5, 'max_features': 3, 'n_estimators': 15}, {'max_depth': 5, 'max
_features': 6, 'n_estimators': 5}, {'max_depth': 5, 'max_features': 6, 'n_estimators': 10}, {'max
_depth': 5, 'max_features': 6, 'n_estimators': 15}, {'max_depth': 5, 'max_features': 9, 'n_
estimators': 5}, {'max_depth': 5, 'max_features': 9, 'n_estimators': 10}, {'max_depth': 5, 'max
_features': 9, 'n_estimators': 15}, {'max_depth': 10, 'max_features': 3, 'n_estimators': 5}, {'
max_depth': 10, 'max_features': 3, 'n_estimators': 10}, {'max_depth': 10, 'max_features': 3, 'n
_estimators': 15}, {'max_depth': 10, 'max_features': 6, 'n_estimators': 5}, {'max_depth': 10, '
max_features': 6, 'n_estimators': 10}, {'max_depth': 10, 'max_features': 6, 'n_estimators': 15},
{'max_depth': 10, 'max_features': 9, 'n_estimators': 5}, {'max_depth': 10, 'max_features': 9, '
n_estimators': 10}, {'max_depth': 10, 'max_features': 9, 'n_estimators': 15}, {'max_depth': 15,
'max_features': 3, 'n_estimators': 5}, {'max_depth': 15, 'max_features': 3, 'n_estimators': 10},
{'max_depth': 15, 'max_features': 3, 'n_estimators': 15}, {'max_depth': 15, 'max_features': 6, '
n_estimators': 5}, {'max_depth': 15, 'max_features': 6, 'n_estimators': 10}, {'max_depth': 15,
'max_features': 6, 'n_estimators': 15}, {'max_depth': 15, 'max_features': 9, 'n_estimators': 5},
{'max_depth': 15, 'max_features': 9, 'n_estimators': 10}, {'max_depth': 15, 'max_features': 9, '
n_estimators': 15}]

超参数组合优劣排序:
[27 26 25 22 15 13 19 10  7 23 18 16 17  9  2 24 11  4 21 14  8 12  6  3
20  5  1]

超参数组合验证集平均 MSE:
[23.489 19.412 19.232 16.834 15.189 14.629 15.762 14.349 13.94  16.965
15.592 15.266 15.41  14.297 13.323 17.27  14.419 13.757 16.383 14.949
14.028 14.617 13.897 13.612 16.249 13.789 13.267]

超参数组合验证集 MSE 标准差:
[3.051 3.928 4.646 2.422 3.017 2.892 3.239 1.684 2.189 5.171 3.781 2.971
3.363 2.746 2.69  3.417 2.494 3.076 3.66  2.448 2.55  2.415 3.567 3.738
3.605 2.75  3.169]

排名第一的超参数组合为
{'max_depth': 15, 'max_features': 9, 'n_estimators': 15}

排名第一的超参数组合对应的平均验证集 MSE:
13.2671422721554

排名第一的超参数组合对应的验证集 MSE 之间的标准差:
3.169208929623358
```

rank_test_score 可以较为直观地提供最优超参数组合，在这个例子中，排名第一的超参

数组合是 max_depth＝15、max_features＝9 和 n_estimators＝15。结合 mean_test_score 列表可以看出这 3 个参数的调试对 MSE 的影响。从 std_test_score 中可以看出该组合下的模型面对不同验证集的 MSE 波动大小。多数情况下，我们希望模型预测不同验证集时表现略同，因此，在平均 MSE 相差较小的选择中，可以考虑加入 std_test_score，选择何为最优组合。其他信息也可以作为参考，例如在超参数范围较大且对模型训练速度产生较大影响时，mean_fit_time 也可能会是选择最优超参数组合时需要考虑的因素之一。

接下来，使用同样的网格搜索极端随机树的最优参数，在下一个 cell 中执行：

```
#Chapter5/grid_search.ipynb

from sklearn.ensemble import ExtraTreesRegressor

#初始化一个极端随机树模型，暂时不设定特定的超参数，完全使用默认值
reg = ExtraTreesRegressor(random_state = 42)
#设定每个待调试超参数的候选值，放入一个 Python 字典，为方便讲解，此处仅使用少量格点
parameters = {'n_estimators': [5, 10, 15],
          'max_depth': [5, 10, 15],
          'max_features': [3, 6, 9]}

#记录网格搜索总共花费的时间
start = time.time()
#初始化一个网格搜索器，使用 ExtraTreesRegressor 自带的衡量指标(MSE)作为 scoring
grid_search_cv2 = GridSearchCV(reg, parameters,
                               scoring = 'neg_mean_squared_error')
grid_search_cv2.fit(df_train_val.drop(columns = ['price']),
                df_train_val['price'])
end = time.time()
print('本次网格搜索耗时：', round(end - start, 2), 's')
```

输出如下：

```
本次网格搜索耗时：1.77 s
```

正如 4.8.2 节所述，极端随机树的训练速度相较随机森林更快，这一优势在使用更多格点进行网格搜索时会更加显著。打印 grid_search_cv2.cv_results_中存储的有效信息，在下一个 cell 中执行：

```
#Chapter5/grid_search.ipynb

#超参数组合顺序与上文中随机森林相同，这里省略对其打印
print('\n 超参数组合优劣排序：\n',
      grid_search_cv2.cv_results_['rank_test_score'])
print('\n 超参数组合验证集平均 MSE：\n',
       - grid_search_cv2.cv_results_['mean_test_score'].round(3))
print('\n 超参数组合验证集 MSE 标准差：\n',
      grid_search_cv2.cv_results_['std_test_score'].round(3))
```

```
#找到排名第一的组合及其 MSE
#由于使用负 MSE 作为衡量指标,打印 - 1 * mean_test_score
#排名第一的组合为 rank_test_score 最高的组合,也就是负 MSE 最高(MSE 最低)的最优组合
rank_1_index = grid_search_cv2.cv_results_['rank_test_score'].argmin()
print('\n 排名第一的超参数组合为\n',
      grid_search_cv2.cv_results_['params'][rank_1_index])
print('\n 排名第一的超参数组合对应的平均验证集 MSE: \n',
      - grid_search_cv2.cv_results_['mean_test_score'][rank_1_index])
print('\n 排名第一的超参数组合对应的验证集 MSE 之间的标准差: \n',
      grid_search_cv2.cv_results_['std_test_score'][rank_1_index])
```

输出如下：

```
超参数组合优劣排序:
[25 27 26 24 23 21 22 20 18 16 11 12  7  3  5 13  9  6 19 14 10 17  8  2
15  4  1]

超参数组合验证集平均 MSE:
[22.314 25.1   24.332 20.431 19.563 18.075 19.043 15.861 15.374 14.335
12.759 12.865 11.533 10.622 10.959 13.641 11.823 11.162 15.716 14.163
12.55  14.975 11.601 10.452 14.171 10.647 10.264]

超参数组合验证集 MSE 标准差:
[4.858 4.872 5.003 4.062 3.735 3.394 3.466 3.503 3.967 4.096 3.413 3.252
1.86  1.814 2.182 4.738 2.795 3.309 2.29  2.769 2.753 2.746 2.068 2.142
3.935 3.263 3.312]

排名第一的超参数组合为
{'max_depth': 15, 'max_features': 9, 'n_estimators': 15}

排名第一的超参数组合对应的平均验证集 MSE:
10.264269102677938

排名第一的超参数组合对应的验证集 MSE 之间的标准差:
3.3122587085283897
```

使用极端随机树，排名第一的超参数组合同样是 max_depth＝15、max_features＝9 和 n_estimators＝15，但在该组合下，平均的验证集 MSE 约为 10.264，低于随机森林中约为 13.27 的平均验证集 MSE。

最后，使用最优超参数组合和完整的交叉验证集训练的模型，对测试集进行预测和评估。直接使用 GridSearchCV 中的.score 函数，依次输入测试集的特征和目标值，在下一个 cell 中执行：

```
#使用最优超参数组合,对测试集进行预测
print('随机森林使用最优组合后,预测测试集所得负 MSE: ',
      grid_search_cv.score(df_test.drop(columns = ['price']),
                           df_test['price']))
print('极端随机树使用最优组合后,预测测试集所得负 MSE: ',
```

```
        grid_search_cv2.score(df_test.drop(columns = ['price']),
                              df_test['price']))
```

输出如下：

```
随机森林使用最优组合后,预测测试集所得负 MSE: -7.6210021055723
极端随机树使用最优组合后,预测测试集所得负 MSE: -5.154347667839
```

由此可见，在随机森林和极端随机树各自最优的超参数组合下，极端随机树在测试集取得约为 5.15 的 MSE，低于随机森林取得的约为 7.62 的测试集 MSE。

实际的调参和模型对比中，需要在计算力允许的范围内使用更多的格点，或调试更多的参数，而后对比不同模型最优参数下的表现。

5.3.2 随机搜索法

5.3.1 节中介绍的网格搜索法可以完整地搜索指定格点内的所有组合，但这一做法的缺陷将在每个超参数的候选值增加时，或需调试的超参数数量增加时显现。

5.3.1 节中的两个模型皆调试了 3 个参数，每个参数的候选值为 3 个。这意味着，每个模型尝试了 27 种不同的超参数组合。在这样的组合量级下，网格搜索法在随机森林的参数寻找上花费 2.6s，在极端随机树的参数寻找上花费 1.77s。实践中，假设每个参数的候选值增加为 20 个，每个模型将尝试 8000 种不同的超参数组合，网格搜索法将在随机森林的参数寻找上花费$\frac{8000}{27}\times 2.6\text{s}\approx 770.37\text{s}$，大约 12.8min；在极端随机树的参数寻找上花费$\frac{8000}{27}\times 1.77\text{s}\approx 524.4\text{s}$，也就是 8.74min。假设再加入待调试参数，例如 min_samples_split、max_leaf_nodes 和 min_impurity_descrease，每个新增的参数各有 10 个候选值，组合数将会从 8000 增加至 8000000，搜索所需时间也会增至原来的 1000 倍，随机森林的超参数搜索将花费约 213h，极端随机树的超参数搜索将花费约 146h。这惊人的时长还是建立在模型较为简单且数据集较小的基础上，实践中遇到的许多数据集比仅含 506 个数据点的波士顿数据集大。

考虑到网格搜索法的这一缺陷，遇到组合量级较大的问题时，可以采用随机搜索法（random search）。使用随机搜索法需要输入每个待调试参数的候选值范围。算法会在每个待调试参数的候选值范围内随机选取一个值加入组合，而我们可以根据搜索空间的大小设定搜索的总组合数。5.3.1 节中提到，当超参数为连续变量时，网格搜索法将尝试取值范围内间隔相同的 k 个数值。这是因为连续变量在某个范围内会存在无穷大的可选数值。而使用随机搜索，在面对取值为连续变量的超参数时，可以尝试到搜索空间内许多网格搜索无法触及的组合。

举一个二维的例子，假设共有两个待调试的超参数，其取值范围皆为连续变量。网格搜索的组合分布与随机搜索的组合分布如图 5.7 所示，二者皆尝试 25 个超参数组合。

sklearn 中的 model_selection.RandomizedSearchCV 使用 5.2 节中介绍的 K 折交叉验证，在输入的超参数范围中使用特定分布采样，寻找最优超参数组合。K 折交叉验证和随机搜索组合的搜寻过程如下。

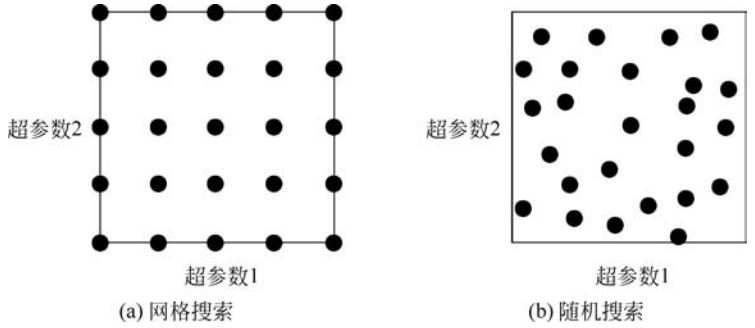

图 5.7 网格搜索和随机搜索尝试组合分布示意图

(1) 设定愿意尝试的总组合数 N。

(2) 循环 N 次：

(a) 在每个输入的超参数范围内采样，得到一组随机抽取的组合。

(b) 进行 K 折交叉验证，并存储该超参数组合下的衡量指标信息，如 5.2 节中的平均 MSE 或折之间 MSE 的标准差。

(3) 分析每种组合所对应的衡量指标信息，选择最优组合。

RandomizedSearchCV 函数中需要输入：

(1) estimator：待进行超参数调试的模型。

(2) param_distributions：包含所有模型中需调试的超参数，以及每个超参数的取值范围或分布的 Python 字典。若取值范围输入为列表，则使用均匀分布在列表中采样；若使用特定分布采样，则需要在字典中对应的值内输入分布函数。

(3) n_iter：用于设定愿意尝试的总组合数 N，默认值为 10。

(4) scoring：指定衡量指标，可为衡量指标对应的特定字符串或一个列表的字符串，默认使用模型自带的衡量指标。

(5) cv：用于设定 K 折交叉验证中的 k 值。

(6) refit：设定是否结束交叉验证时，使用全部的交叉验证集和搜寻到的最优超参数重新训练模型，默认值为 True。当 scoring 为一个字符串列表时，需要在 refit 中输入用于为超参数组合优劣排序的衡量指标。

(7) return_train_score：是否存储训练集表现，默认值为 False。若设定为 True，则 cv_results 将包含模型在交叉验证时训练集上的表现。

使用随机搜索，对随机森林模型中的 n_estimators、max_depth、max_features 和 min_impurity_decrease 进行调试，并对波士顿数据集进行预测，执行：

```
#Chapter5/random_search.ipynb

import pandas as pd
from sklearn.datasets import load_boston
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import RandomizedSearchCV, train_test_split
```

```
import numpy as np
from scipy.stats import uniform

#读取数据集,数据特征详情可参考 4.6 节
boston_dataset = load_boston()
df = pd.DataFrame(boston_dataset['data'])
df.columns = boston_dataset['feature_names']
df['price'] = boston_dataset['target']

#初始化一个随机森林模型,暂时不设定特定的超参数,完全使用默认值
reg = RandomForestRegressor(random_state = 42)
#设定每个待调试超参数的候选取值范围或采样分布,放入一个 Python 字典
#由于使用的是随机搜索(而非网格搜索),可以扩大采样取值范围
#n_estimators、max_depth 和 max_features 皆适用 Numpy array 定义采样范围
#min_impurity_decrease 使用 scipy.stats.uniform 设定均匀分布采样
parameters = {'n_estimators': np.arange(3, 20),       #[3, 19]中每个整数
              'max_depth': np.arange(3, 20),          #[3, 19]中每个整数
              'max_features': np.arange(3, 14),       #[3, 13]中每个整数
              #均匀分布采样,范围为[0, 0.1]
              'min_impurity_decrease': uniform(loc = 0, scale = 0.1)}

#分割交叉验证集(占 85 %)和测试集(占 15 %)
df_train_val, df_test = train_test_split(df, test_size = 0.15, random_state = 42)
df_train_val.reset_index(drop = True, inplace = True) #方便索引

#初始化一个网格随机搜索器,使用 - 1 * MSE 作为 scoring
random_search_cv = RandomizedSearchCV(reg, parameters,
                                      n_iter = 27, #测试 27 组范围内的超参数随机组合
                                      scoring = 'neg_mean_squared_error',
                                      random_state = 42)
random_search_cv.fit(df_train_val.drop(columns = ['price']),
                     df_train_val['price'])
```

random_search_cv 中的 cv_results_存储的信息类型和格式与 5.3.1 节 grid_search_cv 中的 cv_results_基本相同,打印 random_search_cv.cv_results_中存储的有效信息,在下一个 cell 中执行:

```
#Chapter5/random_search.ipynb

#超参数组合顺序与 5.3.1 节例子中打印的格式相同,这里省略打印
print('\n 超参数组合优劣排序: \n',
      random_search_cv.cv_results_['rank_test_score'])
print('\n 超参数组合验证集平均 MSE: \n',
      - random_search_cv.cv_results_['mean_test_score'].round(3))
print('\n 超参数组合验证集 MSE 标准差: \n',
      random_search_cv.cv_results_['std_test_score'].round(3))

#找到排名第一的组合及其 MSE
#由于使用负 MSE 作为衡量指标,打印 - 1 * mean_test_score
```

```
#排名第一的组合为 rank_test_score 最高的组合,也就是负 MSE 最高(MSE 最低)的最优组合
rank_1_index = random_search_cv.cv_results_['rank_test_score'].argmin()
print('\n 排名第一的超参数组合为\n',
      random_search_cv.cv_results_['params'][rank_1_index])
print('\n 排名第一的超参数组合对应的平均验证集 MSE: \n',
      - random_search_cv.cv_results_['mean_test_score'][rank_1_index])
print('\n 排名第一的超参数组合对应的验证集 MSE 之间的标准差: \n',
      random_search_cv.cv_results_['std_test_score'][rank_1_index])
```

输出如下：

```
超参数组合优劣排序:
[12  8 10 16 27  2  5 18 13 17  3  6 14 19 11  1 15  9 23 21  7 20 24 22
26 25  4]

超参数组合验证集平均 MSE:
[14.247 13.481 14.217 15.214 21.728 12.825 13.139 15.589 14.454 15.401
12.887 13.173 15.072 16.031 14.247 12.725 15.12  13.486 18.004 17.403
13.445 16.94  19.549 17.909 20.552 19.623 13.003]

超参数组合验证集 MSE 标准差:
[2.723 1.937 1.989 3.334 2.683 2.216 3.027 4.524 3.465 3.437 2.847 2.711
4.052 2.554 2.982 2.424 3.555 3.203 3.898 5.515 2.771 7.324 3.006 5.513
3.788 5.527 3.336]

排名第一的超参数组合为
{'max_depth': 16, 'max_features': 10, 'min_impurity_decrease': 0.05704439744053994,
'n_estimators': 10}

排名第一的超参数组合对应的平均验证集 MSE:
12.725258999905279

排名第一的超参数组合对应的验证集 MSE 之间的标准差:
2.4243041417429576
```

对比 5.3.1 节网格搜索法中的最优超参数组合所得约为 13.27 的 MSE,同样在 27 组超参数的尝试中,随机搜索法找到了取得约为 12.72 更优 MSE 的超参数组合：

- n_estimators=13
- max_depth=9
- max_features=5
- min_impurity_decrease=0.03572802662812243

这一优化并不是必然的,这样微小的提升也并不一定能在测试集或未来数据预测中体现。并且这样对比两种搜索方法的最优 MSE 也不公平,在随机搜索中,我们加大了 n_estimator、max_depth 和 max_features 的取值范围,并增加了 min_ impurity _decrease 这一超参数进行调试。这个对比只是为了说明,随机搜索可以在有限的迭代中,在一个较大的超参数取值空间内搜索到较为不错的组合,而网格搜索法中的迭代数将随着超参数取值空间的扩大而

飞速上升，不像随机搜索法一般受我们的控制。

最后，使用最优超参数组合和完整的交叉验证集训练的模型，对测试集进行预测和评估。在下一个 cell 中执行：

```
#使用最优超参数组合,对测试集进行预测
print('随机森林使用最优组合后,预测测试集所得负 MSE: ',
    random_search_cv.score(df_test.drop(columns = ['price']),
                           df_test['price']))
```

输出如下：

```
随机森林使用最优组合后,预测测试集所得负 MSE: -8.659385764532018
```

随机搜索法在调试 4 个随机森林参数后，取得最优超参数组合约为 8.66 的 MSE，略高于网格搜索法在调试 3 个随机森林参数后取得的约为 7.62 的测试集 MSE。正如上文所述，验证集平均 MSE 的降低并不一定能在测试集 MSE 的对比上体现。从交叉验证 MSE 的标准差中也分析过，数据集本身的分布存在子集与子集之间的差异，这种情况下，验证集平均 MSE 的参考价值可能高于参考一个较小的测试集上的 MSE。

5.3.3 遗传算法

网格搜索法和随机搜索法存在一个共性，每组超参数组合的优劣这一信息并不被加以利用。

举个例子，在调试随机森林模型的超参数时，假设待调试参数 n_estimators 的候选值为[2，7，12]，max_depth 的候选值为[2，5，8，11，14，17，20]，并且在这个例子中，n_estimators 为 2 的随机森林预测效果皆不佳。网格搜索法会依次尝试以下超参数组合：

- n_estimators=2，max_depth=2
- n_estimators=2，max_depth=5
- n_estimators=2，max_depth=8
- ……
- n_estimators=2，max_depth=20

尽管在得知{n_estimators=2，max_depth=2}、{ n_estimators=2，max_depth=5}的表现皆不佳的情况下，因这一信息不被加以利用，网格搜索法仍然会尝试{n_estimators=2，max_depth=8}、{n_estimators=2，max_depth=11}等 n_estimators=2 的组合。

随机搜索法遇到的情况大同小异。在多次随机尝试后，随机搜索法也许会尝试{n_estimators=2，max_depth=8}、{n_estimators=2，max_depth=17}这两种组合，并发现二者对应模型的表现皆不佳，但同样由于这一信息不被加以利用，随机搜索法并不会在下一次随机抽取超参数数值时避开 n_estimators=2 这一选择。

可视化一个网格搜索中各个超参数组合所对应的模型表现。使用 5.3.1 节中的部分代码，并将不同 n_estimators 和 max_depth 的组合所对应的 MSE 绘制成一个热图（Heat map）。使用 GridSearchCV，定义 n_estimators 和 max_depth 的取值范围并进行网格搜索，执行：

```
#Chapter5/genetic_algorithm.ipynb

import pandas as pd
from sklearn.datasets import load_boston
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV, train_test_split
import numpy as np
import time

#读取数据集,数据特征详情可参考 4.6 节
boston_dataset = load_boston()
df = pd.DataFrame(boston_dataset['data'])
df.columns = boston_dataset['feature_names']
df['price'] = boston_dataset['target']

#初始化一个随机森林模型,暂时不设定特定的超参数,完全使用默认值
reg = RandomForestRegressor(random_state = 42)
#设定每个待调试超参数的候选取值,放入一个 Python 字典
#为方便二维可视化,仅调试两个超参数
parameters = {'n_estimators': np.arange(5, 15),
          'max_depth': np.arange(5, 15)}

#分割交叉验证集(占 85%)和测试集(占 15%)
df_train_val, df_test = train_test_split(df, test_size = 0.15, random_state = 42)
df_train_val.reset_index(drop = True, inplace = True) #方便索引

#初始化一个网格搜索器,使用 -1 * MSE 作为 scoring
grid_search_cv = GridSearchCV(reg, parameters,
                              scoring = 'neg_mean_squared_error')
grid_search_cv.fit(df_train_val.drop(columns = ['price']),
                df_train_val['price'])
```

使用 Pandas 中的.pivot_table 函数,建立一个行为不同 n_estimator 取值,列为不同 max_depth 取值,项为该行、列对应的超参数组合所得 MSE 的 DataFrame。.pivot_table 函数用于重新设定 DataFrame 的行、列、值,并对原 DataFrame 的值使用某种函数进行合计,如求和、求平均数、求最小值等。使用需输入:

(1) data:原 DataFrame,存储所有未合计数值。

(2) values:被合计的列名。

(3) index:原 DataFrame 中的列名,可为字符串存储的单个列名或列表存储的多个列名。输入的列将作为新 DataFrame 中的行。

(4) columns:原 DataFrame 中的列名,可为字符串存储的单个列名或列表存储的多个列名。输入的列将作为新 DataFrame 中的列。

(5) aggfunc:用于合计的函数,默认值为 Numpy.mean,用于计算平均值。

(6) fill_value:用于填补合计后 DataFrame 中的空缺值,默认值为 None。

将 cv_results_中存储的信息放入一个 DataFrame,并对其使用.pivot_table。在下一个

cell 中执行：

```
#Chapter5/genetic_algorithm.ipynb

import pandas as pd

pd.options.display.max_columns = 8          #限制显示的列数
cv_results_df = pd.DataFrame(grid_search_cv.cv_results_)
print('原 cv_results_ DataFrame: ')
display(cv_results_df.head())

pd.options.display.max_columns = 20         #放松显示列数的限制
#使用 param_n_estimators 作为新 DataFrame 的行
#param_n_estimators 作为新 DataFrame 的列
#mean_test_score 为被合计的列
pvt = pd.pivot_table(cv_results_df, values = 'mean_test_score',
                     index = 'param_n_estimators',
                     columns = 'param_max_depth')
pvt * = -1                                  #将负 MSE 转化为 MSE

print('\n 重新定义行、列、值后的 DataFrame: ')
display(pvt.head())
```

显示如图 5.8 所示。

原cv_results_ DataFrame:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	...	split4_test_score	mean_test_score	std_test_score	rank_test_score
0	0.010733	0.000396	0.002261	0.000217	...	-11.145454	-15.979853	3.935793	100
1	0.011317	0.000624	0.002304	0.000206	...	-10.291700	-15.751602	4.320214	99
2	0.013518	0.000318	0.002090	0.000024	...	-10.194655	-14.700617	3.424163	78
3	0.015109	0.000452	0.002349	0.000245	...	-11.083657	-15.024449	3.119830	87
4	0.016940	0.000362	0.002505	0.000368	...	-10.986084	-14.883442	2.984429	83

5 rows × 15 columns

重新定义行、列、值后的DataFrame:

param_max_depth	5	6	7	8	9	10	11	12	13	14
param_n_estimators										
5	15.979853	15.577434	15.483880	15.134917	15.213864	15.006325	15.610121	15.504613	14.991463	15.136669
6	15.751602	15.379505	15.044945	14.826077	14.970105	14.830353	15.201063	15.054643	14.659807	14.835580
7	14.700617	14.297051	14.011480	13.817132	13.769147	13.636176	14.069989	14.009666	13.653688	13.864863
8	15.024449	14.680066	14.226623	14.081457	14.204806	13.973938	14.457631	14.274901	13.994875	14.249426
9	14.883442	14.410011	14.098739	13.835465	13.975585	13.829152	14.169710	14.066470	13.750084	14.043134

图 5.8　代码输出

本质上，.pivot_table 使用 param_n_estimators 和 param_max_depth 的所有不同值定义新 DataFrame 的行与列，而后根据每行、每列对应的 param_n_estimators 和 param_max_depth，用 aggfunc 计算该项对应的值。以 param_n_estimators＝5，param_max_depth＝5 所对应的项为例，.pivot_table 实际执行的计算如下，在下一个 cell 中执行：

```
#Chapter5/genetic_algorithm.ipynb

print('索引该行和列对应的所有 mean_test_score: ')

#当前行和列取值
param_n_estimators = 5
param_max_depth = 5

#这个例子中每行、每列组合只对应一个 mean_test_score,这是 cv_results_的特点
#在不同类型的 DataFrame 中进行以下索引可能输出许多列
display(cv_results_df[(cv_results_df['param_max_depth'] == param_max_depth) &\
          (cv_results_df['param_n_estimators'] == \
                        param_n_estimators)]['mean_test_score'])

#由于行、列组合只对应一个 mean_test_score,
#使用.mean()函数本质上只是打印该 mean_test_score
print('\n 索引该行和列对应的 mean_test_score 平均数: ')
print(cv_results_df[(cv_results_df['param_max_depth'] == param_max_depth) &\
          (cv_results_df['param_n_estimators'] == \
param_n_estimators)]['mean_test_score'].mean())
```

输出如下：

```
索引该行和列对应的所有 mean_test_score:
0    -15.979853
Name: mean_test_score, dtype: float64

索引该行和列对应的 mean_test_score 平均数:
-15.979853215094366
```

输出中的平均数等于 pvt DataFrame 乘以－1 之前该行和列的对应值。可视化不同超参数组合所对应的 MSE，在下一个 cell 中执行：

```
#Chapter5/genetic_algorithm.ipynb

import matplotlib.pyplot as plt
fig, ax = plt.subplots(figsize=(10, 10))
im = ax.imshow(pvt)

#显示每个格点对应的超参数
ax.set_xticks(np.arange(len(parameters['n_estimators'])))
ax.set_yticks(np.arange(len(parameters['max_depth'])))
ax.set_xticklabels(parameters['n_estimators'])
ax.set_yticklabels(parameters['max_depth'])
ax.set_xlabel('n_estimators', size=16)
ax.set_ylabel('max_depth', size=16)
ax.set_title('Grid Search MSE', size=20)

#循环所有超参数组合,并在网格内填充该格点对应的 MSE
```

```
for depth_idx in range(len(parameters['max_depth'])):
    for n_idx in range(len(parameters['n_estimators'])):
        depth = parameters['max_depth'][depth_idx]
        n = parameters['n_estimators'][n_idx]
        MSE = -cv_results_df[(cv_results_df['param_max_depth'] == \
                    depth) &\
                (cv_results_df['param_n_estimators'] == \
                    n)]['mean_test_score'].mean()
        MSE = round(MSE, 3)
        text = ax.text(n_idx, depth_idx, MSE,
                ha = "center", va = "center", color = "w")


plt.show()
```

输出如图 5.9 所示。

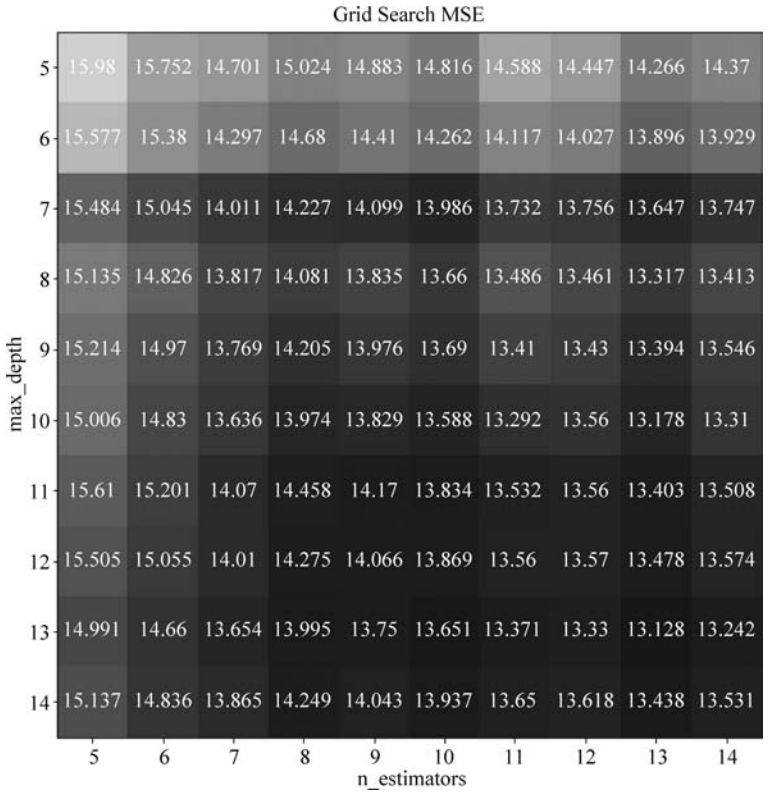

图 5.9　代码输出

由图 5.9 中的 MSE 趋势可见，在[5，14]的范围内，总体上看，较大的 n_estimators 和 max_depth 的组合所得 MSE 较低。注意，这一规律仅存在于[5，14]这一调试取值范围内。也许更大的 max_depth 和 n_estimators 组合所得 MSE 会回升。一般来讲，超参数组合会存在区域性的规律，如图 5.9 所示的局部 MSE 低谷。我们往往无法确定这个低谷是否属于全局低谷，但合理设定的调试范围可以让这一局部低谷接近全局低谷。

遗传算法(genetic algorithm)运用超参数组合中可能存在的规律,试图有方向性地尝试不同的超参数组合。遗传算法的设计灵感来源于自然中生物进化的规律,使用计算机运算模仿生物进化中染色体基因的交叉、变异等过程。该算法并不仅限于解决超参数调试这个问题,许多优化问题都可以放入遗传算法的框架进行运算。在超参数组合优化这一应用中,模型本身可以被看作"进化中的单染色体生物",每个待调试参数可以被看作一个"染色体中的基因",其取值可以看作该染色体基因的具体信息。生物染色体中的所有基因信息确定后,生物需要在环境中生存。模型在指定衡量指标下的表现即为"生物对环境的适应程度"。

遗传算法的超参数组合优化过程如下:

(1) 初始化(initialization):初始化 M 个生物个体,即 M 个取值范围内随机生成的超参数组合,其集合为初始群体 P_0。

(2) 个体选择(selection):群体中的一部分个体将被选中进行下一代个体的"繁衍",对环境适应性更强的个体将有更大的概率被选中。超参数的选择中,"适应性"这一数值为指定的衡量指标下该组合对应模型的表现,如 MSE、精度等数值。

(3) 遗传算子(genetic operator):根据多种遗传算子生成下一代超参数组合,常用的两种算法为交叉运算(crossover)和变异运算(mutation):

(a) 交叉运算:被选中的个体两两配对,成为下一代的"父母"。交叉运算中选择交换基因的方式有多种,本节将使用单点交叉。单点交叉会从染色体基因序列中选择一个分割点,父与母染色体位于分割点左侧的超参数将互相交替,右侧的超参数不变。交替后两个新的超参数组合为该父母的两个"孩子",加入下一代的群体中,如图 5.10 所示。交叉运算的目的在于保存当前群体中适应性强的基因并进行重组,尝试是否能繁衍出适应性更强的后代。

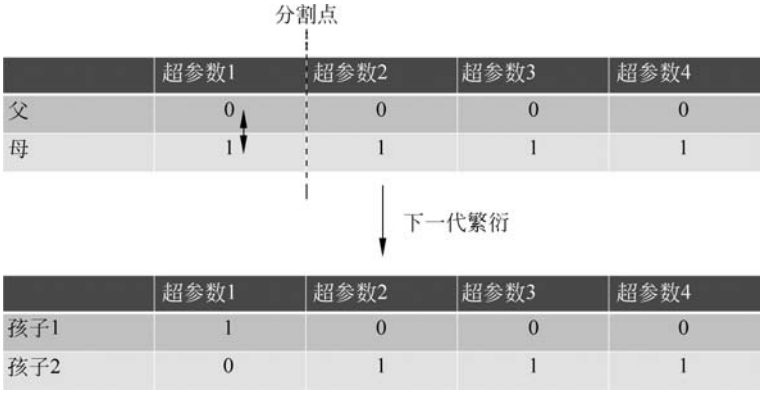

图 5.10 单点交叉示意图

(b) 变异运算:繁衍过程中,下一代个体有小概率衍生出父母基因中皆不存在的基因,如图 5.11 所示。变异的目的在于增加群体的多样性,尝试是否能变异出适应性更强的后代。

(4) 终止(termination):相比父代,当子代的适应性不再有显著提升,或当繁衍代数达到一个提前决定的量时,可以终止算法。所有代个体中适应性最强的个体基因,对应遗传算

法搜寻到的最优超参数组合。

	超参数1	超参数2	超参数3	超参数4
孩子1	1	0	0	0
孩子2	0	1	1	1

↓ 变异

	超参数1	超参数2	超参数3	超参数4
孩子1	1	0	0	0
孩子2	0	2	1	1

图 5.11　个体变异示意图

使用 Python 写出一套遗传算法的流程，搜索最优的 n_estimators 和 max_depth 的取值组合。首先进行第 1 步初始化，在下一个 cell 中执行：

```
#Chapter5/genetic_algorithm.ipynb

#初始化
def initialization(M):
    '''初始化 M 个生物个体
    输入:
        M: 初始化群体中总个体数
    输出:
        初始化的群体
    '''
    np.random.seed(42)     #为了保证本段代码可复制性添加,应用时可删除

    #使用向量存储染色体中不同基因的值
    n_estimators = np.zeros([M, 1], dtype = np.int)
    max_depth = np.zeros([M, 1], dtype = np.int)

    for i in range(M):
        n_estimators[i] = np.random.randint(5, 15)       #范围为[5, 14]
        max_depth[i] = np.random.randint(5, 15)          #范围为[5, 14]

    population = np.concatenate((n_estimators, max_depth), axis = 1)
    return population

#设定 M = 20
population = initialization(20)
print('初始化个体数为 20 的群体: \n', population)
```

输出如下：

```
初始化个体数为 20 的群体:
[[11  8]
[12  9]
[11 14]
[ 7 11]
[12  9]
[ 8 12]
[12  7]
[10  9]
[ 6 12]
[10  6]
[ 9  5]
[14 10]
[13  5]
[14  7]
[11  8]
[13  7]
[ 9  7]
[11  9]
[13 11]
[ 6  8]]
```

输出中每行代表一个单染色体个体，每个染色体包含 n_estimators 和 max_depth 两个基因，两个基因的取值范围均为[5, 14]中的整数。

第 2 步，进行个体选择。将交叉验证的平均负 MSE 作为适应性，适应性越高对应的负 MSE 越高，也是更优的模型。在下一个 cell 中执行：

```
#Chapter5/genetic_algorithm.ipynb

from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error

def get_individual_fitness(population, df_train_val):
    '''计算群体中每个个体的适应性
    输入:
        population: 包含所有个体染色体基因信息的群体
        df_train_val: 交叉验证集
    输出:
        群体中每个个体对应的适应性分数
    '''
    fitness_scores = []
    for i in range(population.shape[0]):
        curr_n = population[i][0]
        curr_depth = population[i][1]
        #交叉验证
        kf = KFold(n_splits=5)
        mse_lst = []
        for train_index, val_index in kf.split(df_train_val):
```

```
            df_train, df_val = df_train_val.iloc[train_index],\
                               df_train_val.iloc[val_index]
            reg = RandomForestRegressor(n_estimators = curr_n,
                                        max_depth = curr_depth,
                                        random_state = 42)
            #训练随机森林
            reg.fit(df_train.drop(columns = ['price']), df_train['price'])

            #预测评估
            pred_val = reg.predict(df_val.drop(columns = ['price']))
            mse_val = mean_squared_error(list(df_val['price']), pred_val)
            mse_lst.append(mse_val)

        #计算平均负 MSE 并录为个体适应性
        mse_lst = np.array(mse_lst)
        fitness_scores.append( - round(mse_lst.mean(), 3))

    return fitness_scores

fitness_scores = get_individual_fitness(population, df_train_val)
print('群体中每个个体所对应的适应性: \n', fitness_scores)
```

输出如下：

```
群体中每个个体所对应的适应性:
[ - 13.486, - 13.43, - 13.65, - 14.07, - 13.43, - 14.275, - 13.756, - 13.69, - 15.055,
 - 14.262, - 14.883, - 13.31, - 14.266, - 13.747, - 13.486, - 13.647, - 14.099, - 13.41,
 - 13.403, - 14.826]
```

群体中 20 个个体所对应的适应性皆不同，在这其中选择 12 个适应性较高的个体进行繁衍。上文中介绍遗传算法的步骤时提到，对环境适应性更强的个体将有更大的概率被选中作为下一代的父母。选择算法良多，本节将简单地选择适应性最强的前 k 个个体作为下一代的父母。某些选择算法中，适应性一般，无法进入排名前 k 的个体也有一定概率繁衍。使用这一方法并设定 $k=12$，在下一个 cell 中执行：

```
#Chapter5/genetic_algorithm.ipynb

def parents_selection(population, fitness_scores, num_parents):
    '''在群体中寻找适应性最高的个体
    输入:
        population: 包含所有个体染色体基因信息的群体
        fitness_scores: 群体中每个个体对应的适应性分数
        num_parents: 最终选择的最优个体数
    输出:
        在群体中寻找适应性最高的个体
    '''
    worst_fitness = min(fitness_scores) - 1#设定一个低于群体中所有个体的适应性
    #初始化一个用于存储适应性较高父代个体的 Numpy array
```

```
    selected_parents = np.zeros((num_parents, population.shape[1]),
                                dtype = np.int)

    #在 population 中寻找适应性最高的个体,共 num_parents 个
    for idx in range(num_parents):
        highest_fitness_idx = fitness_scores.index(max(fitness_scores))
        selected_parents[idx, : ] = population[highest_fitness_idx, : ]
        #保证选择过的个体不再被选择
        fitness_scores[highest_fitness_idx] = worst_fitness

    return selected_parents

parents = parents_selection(population, fitness_scores, 12)
print('群体中适应性最高的 12 个个体: \n', parents)
```

输出如下：

```
群体中适应性最高的 12 个个体:
[[14 10]
[13  8]
[13  8]
[12 10]
[13  9]
[12  8]
[11  8]
[14  9]
[11 14]
[13  6]
[14  6]
[11  6]]
```

第 3 步,对优选出来的父代个体进行交叉运算和变异运算,繁衍下一代个体。在下一个 cell 中执行：

```
#Chapter5/genetic_algorithm.ipynb

def crossover(parents, num_children):
    '''交叉运算
    输入:
        parents: 父代 k 个最优个体
        num_children: 新一代个体数
    输出:
        新一代个体
    '''
    #存储下一代个体基因
    after_crossover = np.zeros((num_children, parents.shape[1]),
                               dtype = np.int)
    for i in range(num_children):
```

```
            #第 i 个孩子的父母为父母列表中的第 i 和第 i+1 位
            #若 i>总父母个数,则取 i % parents.shape[0]
            parent1_idx = i % parents.shape[0]
            #若 i+1>总父母个数,则取(i+1) % parents.shape[0]
            parent2_idx = (i+1) % parents.shape[0]

            #由于仅有两个超参数,使用中点作为交叉点
            #这样的运算相当于仅保留图 5.6 中的"孩子 2"
            after_crossover[i, 0] = parents[parent1_idx, 0]
            after_crossover[i, 1] = parents[parent2_idx, 1]

    return after_crossover

def mutation(after_crossover):
    '''变异运算
    输入:
        after_crossover: 交叉运算后所得新一代个体
    输出:
        变异后的新一代个体
    '''
    np.random.seed(42)

    mutated = after_crossover.copy()
    #n_estimators 变异值,范围为[-1, 1],拥有 1/3 的概率不变异
    mutation_val_n = np.random.randint(-1, 2,
                                       after_crossover.shape[0])
    #max_depth 变异值,范围为[-1, 1],拥有 1/3 的概率不变异
    mutation_val_depth = np.random.randint(-1, 2,
                                           after_crossover.shape[0])

    #加入变异值
    mutated[:, 0] += mutation_val_n
    mutated[:, 1] += mutation_val_depth

    #防止变异数值超出指定范围
    #n_estimators 变异后不能低于 5
    mutated[:, 0] = np.maximum(5, mutated[:, 0])
    #n_estimators 变异后不能高于 14
    mutated[:, 0] = np.minimum(14, mutated[:, 0])
    #max_depth 变异后不能低于 5
    mutated[:, 1] = np.maximum(5, mutated[:, 1])
    #max_depth 变异后不能高于 14
    mutated[:, 1] = np.minimum(14, mutated[:, 1])

    return mutated

#执行交叉算法
children_after_crossover = crossover(parents, parents.shape[0])
children_after_mutation = mutation(children_after_crossover)
print('经过交叉运算和变异运算后的下一代个体: \n', children_after_mutation)
```

输出如下：

```
经过交叉运算和变异运算后的下一代个体:
[[14  7]
[12  9]
[14 10]
[13  8]
[12  8]
[11  8]
[12  9]
[14 14]
[12  5]
[14  5]
[14  6]
[12 10]]
```

以上代码为一轮繁衍的结果。使用所有定义的函数，完整执行 10 个迭代的进化。在下一个 cell 中执行：

```
#Chapter5/genetic_algorithm.ipynb

initial_population = 10                    #初始群体大小
num_pairs_parents = 5                      #每一代成为父母的个体数,也是下一代群体大小
num_generations = 10                       #总代数
num_parameters = 2                         #待调试参数量

#初始化群体,此为第 0 代
population = initialization(initial_population)
#存储每一代个体的染色体基因数值
population_history = [population]
#存储进化过程中不同代的个体适应性
curr_fitness = get_individual_fitness(population, df_train_val)
fitness_history = [curr_fitness]
curr_best_generation = 0                   #记录当前最优组合所在代数
curr_best_fitness = max(curr_fitness)      #记录当前最优组合的负 MSE
print('第 0 代群体: ')
print('最高负 MSE 为', curr_best_fitness)
print()

#开始进化

for generation in range(num_generations):

    print("第{}代群体: ".format(generation + 1))

    #选择最优的个体成为父母
    parents = parents_selection(population,
                                curr_fitness,
                                num_pairs_parents)
```

```
    #执行交叉算法,并设定为下一代群体
    children_after_crossover = crossover(parents, num_pairs_parents)
    population = mutation(children_after_crossover)
    population_history.append(population.copy())

    #计算新一代的适应性
    curr_fitness = get_individual_fitness(population, df_train_val)
    fitness_history.append(curr_fitness.copy())
    gen_best_fitness = max(curr_fitness)
    print('最高负 MSE 为', gen_best_fitness)
    print()

    #若找到更优组合,则更新最优组合和最优负 MSE
    if gen_best_fitness > curr_best_fitness:
        curr_best_fitness = gen_best_fitness
        curr_best_generation = generation + 1
```

输出如下：

```
第 0 代群体:
最高负 MSE 为 -13.43

第 1 代群体:
最高负 MSE 为 -13.317

第 2 代群体:
最高负 MSE 为 -13.292

第 3 代群体:
最高负 MSE 为 -13.178

第 4 代群体:
最高负 MSE 为 -13.438

第 5 代群体:
最高负 MSE 为 -13.31

第 6 代群体:
最高负 MSE 为 -13.128

第 7 代群体:
最高负 MSE 为 -13.478

第 8 代群体:
最高负 MSE 为 -13.178

第 9 代群体:
最高负 MSE 为 -13.438
```

```
第 10 代群体:
最高负 MSE 为 -13.128
```

最后,使用 11 代群体中最优个体对应参数训练模型并预测试集数据,在下一个 cell 中执行:

```
#Chapter5/genetic_algorithm.ipynb

#通过最优代数和进化历史的记录,寻找最优组合
best_generation_fitness = fitness_history[curr_best_generation]
best_index = \
        best_generation_fitness.index(max(best_generation_fitness))
best_estimators, best_depth = \
             population_history[curr_best_generation][best_index]

print('最优超参数组合:\n',
    'n_estimators = {0}, max_depth = {1}\n'.format(best_estimators,
                                                   best_depth))

#使用交叉验证集训练模型,并在测试集上进行评估
reg = RandomForestRegressor(n_estimators = best_estimators,
                            max_depth = best_depth,
                            random_state = 42)
#训练随机森林
reg.fit(df_train_val.drop(columns = ['price']), df_train_val['price'])

#预测评估
pred_test = reg.predict(df_test.drop(columns = ['price']))
mse_test = mean_squared_error(list(df_test['price']), pred_test)
print('随机森林使用最优组合后,预测测试集所得 MSE: ', round(mse_test, 3))
```

输出如下:

```
最优超参数组合:
n_estimators = 13, max_depth = 13

随机森林使用最优组合后,预测测试集所得 MSE: 13.703
```

最后,根据进化历史,分析并可视化遗传算法不同代中个体的变化趋势,在下一个 cell 中执行:

```
#Chapter5/genetic_algorithm.ipynb

#绘制进化过程群体中 n_estimators 和 max_depth 的平均值
plt.plot([p[:, 0].mean() for p in population_history],
       linestyle = '-.', label = 'n_estimators')
plt.plot([p[:, 1].mean() for p in population_history],
       label = 'max_depth')
```

```
plt.title('Genetic evolution of parameter mean')
plt.xlabel('generation')
plt.ylabel('parameter value')
plt.legend()
plt.show()
```

输出如图 5.12 所示。

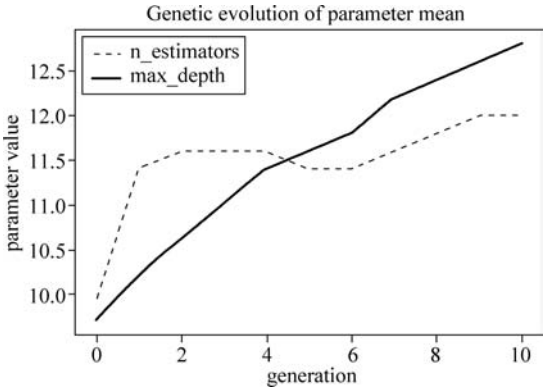

图 5.12　代码输出

由此可见，群体繁衍、进化的过程中，遗传算法的搜索空间一步步接近 n_estimators 和 max_depth 偏大的值，也是本节开篇所示的 MSE 较小区域。当搜索空间较大时，遗传算法的这一特质将有效地发掘表现较佳的组合区域，可以节省网格搜索与随机搜索中执行的许多结果大概率不佳的探索。

5.4　函数正则化

第 4 章中的许多模型存在过拟合的风险。4.5.1 节线性回归模型中，若 m 值过小，则模型呈现欠拟合；若 m 值过大，则模型呈现过拟合。4.5.1 节中，已知数据的本质为 3 次多项式的前提下，使用 $m=3$ 得到了在训练集范围内拟合较好，且预测超出训练集范围数据点的能力较强的函数模型。同时，我们发现 $m=19$ 的模型呈现过拟合。

简单的解决方案是选择较小的 m 值，也就是减少模型本身可以学习的权重数量，但这种简单的方式限制了模型可以表达的函数类型。另外，某些模型，例如深度神经网络，注定包含许多权重，若想利用这类模型的优势，单纯地减少可学习权重数量往往不可行。

正则化(regularization)在保持原有权重数量的前提下，有效减缓模型呈现的过拟合问题。正则化的核心在于惩罚取值过大的权重。回顾线性回归的成本函数 J：

$$J=\frac{1}{2N}\|\boldsymbol{X}\boldsymbol{w}-\boldsymbol{y}\|^2 \tag{5.30}$$

模型训练的目的在于降低这一成本函数。在函数中加入关于权重的函数项 $R(\boldsymbol{w})$，也被称为正则化项(regularizer)，如式(5.31)所示：

$$J=\frac{1}{2N}\|\boldsymbol{X}\boldsymbol{w}-\boldsymbol{y}\|^2+\lambda R(\boldsymbol{w}) \tag{5.31}$$

其中，λ 决定正则化强度，λ 越大对应的正则化越强。降低成本函数意味着降低正则化项 $R(\boldsymbol{w})$。常用的正则化项包含 L1 正则化项和 L2 正则化项。L2 正则化中，$R(\boldsymbol{w})$定义为所有权重平方之和，其表达式为

$$R(\boldsymbol{w})=\sum_j w_j^2 \tag{5.32}$$

L1 正则化中，$R(\boldsymbol{w})$定义为所有权重绝对值之和，其表达式为

$$R(\boldsymbol{w})=\sum_j |w_j| \tag{5.33}$$

成本函数中加入 L1 或 L2 正则项后，需要在降低 $\frac{1}{2N}\|\boldsymbol{X}\boldsymbol{w}-\boldsymbol{y}\|^2$ 预测值与目标之间距离的同时，保证 $R(\boldsymbol{w})$不要过大。多数情况下，降低 $\frac{1}{2N}\|\boldsymbol{X}\boldsymbol{w}-\boldsymbol{y}\|^2$ 的代价是提升 $R(\boldsymbol{w})$，模型呈现过拟合；而降低 $R(\boldsymbol{w})$的代价是提升 $\frac{1}{2N}\|\boldsymbol{X}\boldsymbol{w}-\boldsymbol{y}\|^2$，模型呈现欠拟合。两者之间的平衡由超参数 λ 控制。若 λ 较大，则表示模型愿意牺牲对 $\frac{1}{2N}\|\boldsymbol{X}\boldsymbol{w}-\boldsymbol{y}\|^2$ 项的优化，而着重优化正则项 $R(\boldsymbol{w})$，过大的 λ 会由于过度限制权重的大小造成模型欠拟合；若 λ 较小，则表示模型愿意牺牲对正则项 $R(\boldsymbol{w})$的优化，而着重优化 $\frac{1}{2N}\|\boldsymbol{X}\boldsymbol{w}-\boldsymbol{y}\|^2$ 项，过小的 λ 会由于过度轻视对权重大小的控制造成模型过拟合。

最后，简单讲解一下 L1 正则化和 L2 正则化的区别。从式(5.32)和式(5.33)的对比中可以看出，L2 正则化随着权重绝对值的平方增加，而 L1 正则化随着权重绝对值的增加仅呈直线增加。这意味着，在 L2 正则化中，一个较小权重绝对值的降低并不能与一个较大权重绝对值的提升相抵消，而在 L1 正则化中，两者可以相抵消。

举个例子，假设 $w_1=0.5$，$w_2=50$。在其余权重不变的前提下，设想将 w_1 的绝对值降低 0.5，将 w_2 的绝对值提升 0.5，变化前 w_1 和 w_2 对 L2 正则项的总贡献为 $0.5^2+50^2=2500.25$，对 L1 正则项的总贡献为 $|0.5|+|50|=50.5$；变化后 w_1 和 w_2 对 L2 正则项的总贡献为 $0^2+50.5^2=2550.25$，对 L1 正则项的总贡献为 $|0|+|50.5|=50.5$。若使用 L1 正则项，则 w_1 和 w_2 的变化对于正则项没有影响，也就意味着，这一变化降低了 $\frac{1}{2N}\|\boldsymbol{X}\boldsymbol{w}-\boldsymbol{y}\|^2$ 项，该变化从降低总成本的角度来讲是可取的。若使用 L2 正则项，则 w_1 和 w_2 的变化提高了正则项，这意味着，即使这一变化降低了 $\frac{1}{2N}\|\boldsymbol{X}\boldsymbol{w}-\boldsymbol{y}\|^2$ 项，该变化从降低总成本的角度来讲也不一定可取。

从这个例子中可以看出，相比 L2 正则化，L1 正则化更容易使更多权重接近于 0。当特征数较多时，也许我们希望通过设定许多接近 0 的权重进行特征筛选，这时可以选择 L1 正则化；若希望尽量保留所有特征，则选择 L2 正则化。

使用 L1 正则化的回归算法称为稀疏回归(lasso regression)，使用 L2 正则化的回归算法称为岭回归(ridge regression)。在 sklearn 中，调用 linear_model.Lasso 即可使用稀疏回归，调用 linear_model.Ridge 即可使用岭回归。由于 Lasso 和 Ridge 的调用方式并无大的区别，本节最后将使用 Ridge 模型重新预测 4.5.1 节中的例子。回顾 4.5.1 节中的例子，使

用 $m=19$，并可视化模型在稍微超出训练集范围的数据上的预测结果，执行：

```
#Chapter5/regularization.ipynb

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import Ridge, LinearRegression

np.random.seed(42)

#建立 4.5.1 节中的 3 次多项式
x_arr = np.linspace(-10, 10, 50)
#刨除噪声因素,x 与 y 呈 y = x^3 + 2x^2 + x 的关系
df_poly = pd.DataFrame({'x': x_arr,
                        'y': x_arr ** 3 + 2 * x_arr ** 2 + x_arr +\
                           np.random.rand(50) * 150 - 75})

#创建特征 x^2 到 x^19
for m in range(2, 20):
    df_poly['x^' + str(m)] = df_poly['x'] ** m

#L2 正则化, alpha 为文中的 lambda,用于调试正则化强度
reg = Ridge(alpha=1.0)
reg.fit(df_poly, df_poly['y'])

#创建稍微超出训练集范围的数据点
x_arr_expanded = np.linspace(-10.5, 10.5, 50)
df_poly_expanded = pd.DataFrame({'x': x_arr_expanded,
                        'y': x_arr_expanded ** 3 + 2 * x_arr_expanded ** 2 + x_arr_expanded
+\
                           np.random.rand(50) * 150 - 75})
#创建特征 x^2 到 x^19
for m in range(2, 20):
    df_poly_expanded['x^' + str(m)] = df_poly_expanded['x'] ** m

#使用相应 m 值训练好的模型进行预测并绘制预测结果
plt.scatter(df_poly_expanded['x'], df_poly_expanded['y'], label='data')
plt.plot(df_poly_expanded['x'], reg.predict(df_poly_expanded),
                               label='predictions')
plt.title('m=' + str(m))
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```

输出如图 5.13 所示。

对比 4.5.1 节在相同范围内使用普通的 LinearRegression 模型输出，如图 5.14 所示。由此可见，使用正则化确实可以降低过拟合的风险，提高模型的泛化性。

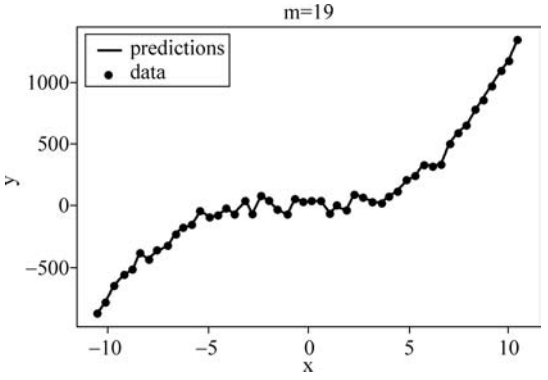

图 5.13　代码输出

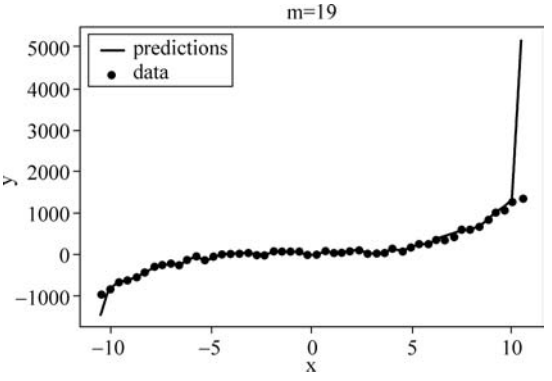

图 5.14　LinearRegression 模型同范围预测输出

这里需要注意的是，正则化并不是一剂万能药。在这个例子中，数据的真实规律是一个3次多项式，而我们使用了 $m=19$ 这样一个过于复杂的假设。上段代码中，训练集的 x 取值范围为[−10，10]，而可视化的预测范围为[−10.5，10.5]，这说明，模型在稍微超出训练集范围的数据点上尚能做出较好的预测，但当预测范围与训练集的 x 取值范围增加时，使用了正则化的模型仍会呈现明显的过拟合。在下一个 cell 中可视化预测范围为[−15，15]的数据点，执行：

```
#Chapter5/regularization.ipynb

#创建超出训练集范围较大的数据点
x_arr_expanded2 = np.linspace(-15, 15, 50)
df_poly_expanded2 = pd.DataFrame({'x': x_arr_expanded2,
                    'y': x_arr_expanded2 ** 3 + 2 * x_arr_expanded2 ** 2 + x_arr_
expanded2 + \
                      np.random.rand(50) * 150 - 75})
#创建特征 x^2 到 x^19
for m in range(2, 20):
    df_poly_expanded2['x^' + str(m)] = df_poly_expanded2['x'] ** m
```

```
#使用相应 m 值训练好的模型进行预测并绘制预测结果
plt.scatter(df_poly_expanded2['x'], df_poly_expanded2['y'], label = 'data')
plt.plot(df_poly_expanded2['x'], reg.predict(df_poly_expanded2),
                              label = 'predictions')
plt.title('m = ' + str(m))
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```

输出如图 5.15 所示。

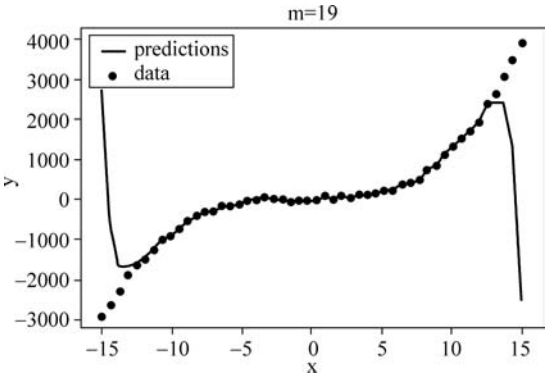

图 5.15　代码输出