第3章 基础建模

CHAPTER 3

第2章讲解了技术流程中的第1步——数据清理，介绍如何安装 Anaconda、Jupyter Notebook、Pandas、scikit-learn、XGBoost、LightGBM、CatBoost 和 TensorFlow，同时讲解如何使用 Pandas 解决表格数据常见的数据缺陷。本章专注于讲解如何快速搭建稳健的基准模型(benchmark model)。

基准模型的作用有二。第一，在进一步为项目投入资源之前，我们可以通过建立简单的模型大致了解数据的可预测性，并根据基准模型的表现与各步骤的可优化空间估算项目收益；第二，基准模型的表现可以为后续更加优化、复杂的模型提供基准，通过与基准模型表现的比对，可以更早发现预测管道可能存在的问题。设想若是跳过基础建模直接投入大量资源建立最优化的模型，第一，我们可能会发现当前拥有的数据并不能对目标进行准确的预测，因而需要收集更多的数据、特征或改变问题定义，这些改变可能会改变最优模型的选择，淘汰曾经调试的“最优”模型，浪费投入的资源；第二，若优化后的模型表行较基准模型更低，大概率是因为搭建的预测管道程序出错，而非优化模型本身不能取得更好的表现。由于实践中的预测管道包括从获取数据、数据处理到输入模型、获得预测结果等多个步骤，往往难以判断较低准确率是源自模型本身无法学习当前数据中的规律，还是因为预测管道某个环节的代码出现了问题。若与拥有基准模型的准确率做比对，当优化模型的准确率低于基准模型时，我们可以较为肯定是预测管道的某一步骤出现程序错误，尽早修复程序中的漏洞。

3.1 判断何为 X 和 y

2.1.4 节中提到了训练模型时需要用到 X_train 和 y_train，本节将定义 X、y，并讲解选择不同 X、y 的意义。

3.1.1 X 和 y 的定义

机器学习算法的目的在于学习将输入的变量映射到目标输出的函数。输入的变量由许多具有描述性质的特征组成，被称作变量 X；输出的变量是预测目标，被称作变量 y。X 的储存形式多为二维矩阵，每行代表一个独立数据点，而每列存储某个特征的取值。第 i 行 j 列存储的是第 i 个独立数据点 j 特征的取值。y 的存储形式根据预测的不同，可为一维数

组或二维矩阵。若预测目标为单个数值，如“3 日后冰激凌的销量是多少”，y 可用一维数组表示，其长度等于 X 的行数。若预测目标为多个数值，如“接下来一个星期每天冰激凌销量各为多少”，y 需要使用二维矩阵表示，矩阵行数等于 X 的行数，列数为 7，等于预测目标的个数。

模型将学习训练集中 X 与 y 之间的关系，并使用未来数据对应的 X 变量预测其 y 的取值，因此，选取 X 所包含特征时，需要确认可以在有效时间内获取未来数据相应特征的取值。假设预测目标为 3 日后冰激凌的销量，可以将训练集中 X 中的每行设定为历史数据中某一天，X 中的各列特征将描述决定当日销量的因素，相应位置的 y 为当日销量。我们可能会认为“昨日销量”这一特征可以帮助预测，便在 X 中加入一列表示昨日销量的数据。这一做法在提取作为训练集的历史数据时不会出现报错——若可以成功提取某一天，称为天数 k，销量作为某行 y 的取值，那么一般情况下天数 $k-1$ 的销量数据也将存在于数据库中，并可以作为该行 X 中昨日销量这一特征的取值，但昨日销量这一特征在预测目标时并不适用。预测目标为 3 日后冰激凌的销量，这意味着未来数据对应的 X 变量中昨日销量这一特征需以 2 日后冰激凌的销量填写，这显然无法实现。这一问题被称为数据泄露，常伴随着较实际情况更为乐观的模型表现。3.3 节将讲解如何规避此类问题。

面对某一问题时，采取不同解决问题的途径意味着对 y 的定义会有所不同。根据需要预测的问题，最直接的方法是让 y 的取值为这个问题的答案。某种程度上，y 的选择也决定了 X 中每行描述的具体对象。例如在预测 3 日后冰激凌销量时，训练集中 y 为某日销量，那么 X 中每行描述的具体对象就是该交易日。若预测目标为下一周销量的平均值，可以使用最直接的方式将 y 定义为某周销量平均值，这时 X 中每行描述的具体对象就是该交易周；也可以采取另一种途径，将 y 定义为从下周星期一开始计算每天冰激凌的销量各为多少，最后计算 7 个日期销量预测的平均值取得问题答案。X 中每行描述的具体对象为一个交易日。对于 y 的选择需根据项目的具体情况及得以收集到的数据内容。上述的例子中，若可以收集到具体形容某日特征的数据，如该日天气，并且这类特征对预测准确性可能起到影响，如夏日中较为凉爽的几天冰激凌销量大概率会降低，将描述对象定义为单个交易日可获得较大的收益。反之，若无法收集此类数据，若仍将每个交易周分割为 7 日，理论上的最大效益可能与分割前相差无几，长度却是分割前的 7 倍左右，因此为了尽可能压缩数据大小，减少计算资源的浪费，应该让每行代表一个交易周。

确定了 y 的定义及每行应当描述的具体对象后，X 的定义也逐渐明了。X 中每行用来描述已确定的、应当描述的对象。其特征应尽量与描述对象处于同一级——这里的等级指的是泛指程度，越低的等级表示越具体。例如当描述对象为一个交易日时，应尽量多选具体到该交易日的特征，而非该交易日所处的交易周的特征。少量的泛指性特征可用于做更高层的划分，如月份这一特征，虽无法对同一月份中的日与日进行区分，却可以区分不同月份的日期。若 X 中皆为高于描述对象等级的特征，例如在描述对象为交易日而特征皆描述交易周或更广的时间范围时，模型将无法通过数据学习同一交易周中不同交易日为何拥有不同销量，从而失去将描述对象划分为交易日的意义。

3.1.2 X 和 y 的选择对预测的影响

3.1.1 节对 X 与 y 选择的讲解也许会让你想起 2.4 节中讲的数据重新格式化。的确，

2.4 节展示了 X、y 的选择如何出现在实际应用中。本节将结合 3.1.1 节讲述的 X、y 选择思路，重新分析 2.4 节的例子，并讨论 X、y 的选择对于预测结果和训练模型所需计算力的影响。

在回顾 2.4 节的例子之前，先统一一下 2.4 节与 3.1.1 节中对同一概念的不同用词。2.4 节中提到，“表格数据格式转变的核心在于重新定义‘用于索引的列’”，这一转变的实质为改变每行描述的对象。

重新创建 2.4 节例子中的 DataFrame：

```
import pandas as pd

df = pd.DataFrame({'User Id': [0, 1, 2, 3, 4, 5, 6],
                   'Preference 1': ['food', 'tech', 'finance', 'finance', 'tech',
                                    'tech', 'tech'],
                   'Preference 2': ['tech', 'finance', 'tech', 'food', 'food',
                                    'finance', 'finance'],
                   'Age': [16, 21, 25, 31, 43, 29, 17],
                   'Click Ads': ['likely', 'unlikely', 'likely', 'likely',
                                 'unlikely', 'unlikely', 'likely']})

display(df)
```

显示如图 3.1 所示。

	User Id	Preference 1	Preference 2	Age	Click Ads
0	0	food	tech	16	likely
1	1	tech	finance	21	unlikely
2	2	finance	tech	25	likely
3	3	finance	food	31	likely
4	4	tech	food	43	unlikely
5	5	tech	finance	29	unlikely
6	6	tech	finance	17	likely

图 3.1 代码输出

假设需要预测的问题为“某用户是否会单击广告”，那么 y 的选择是一个储存单个用户是否单击广告的数组，每行描述的对象为单个用户。已有数据中 Click Ads 这一列数据可以用作 y。这时可以确定，X 中每行用来描述单个用户。已有数据中 User Id、Preference 1、Preference 2 和 Age 这 4 列用于描述单个用户的特征即可作为 X。分割 DataFrame 取得 X 与 y，在新的 cell 中执行：

```
X = df[['User Id', 'Preference 1', 'Preference 2', 'Age']]
y = df['Click Ads']
```

分割后的 X 是一个 Pandas DataFrame，y 是一个 Pandas Series，两者分别可以不经转换当作二维矩阵或一维数组输入 sklearn 的模型中训练模型。

若需要预测的问题为“某用户的年龄”，那么 y 的选择是一个储存单个用户年龄的数

组，每行描述的对象为单个用户。已有数据中 Age 这一列数据可以用作 y。这时同样可以确定，X 中每行用来描述单个用户。已有数据中 User Id、Preference 1、Preference 2 和 Click Ads 这 4 列用于描述单个用户的特征即可作为 X。分割 DataFrame 取得 X 与 y，在新的 cell 中执行：

```
X = df[['User Id', 'Preference 1', 'Preference 2', 'Click Ads']]
y = df['Age']
```

第 6 章将讲解如何为 X 制造更多高收益特征，但在基础建模这一步骤中，若非决定性特征隐藏于现有数据中而必须加以处理取得，暂时不需要对特征进行过多优化。针对基础建模中用于训练模型的数据，只需根据 2.3 节中的方法保证其无数据误差，X 中只保存有效特征，且使用特征在预测未来数据时可及时取得即可。关于 X 中的特征是否有效，更多鉴别方法也会在 6.5 节特征工程中讲解。本节将介绍一个简单的判断特征是否无效的方法。

观察数据及了解每列所代表的信息，可以发现 User Id 这一特征拥有许多不同的取值，并且属于定类数据。这类数据被称为高基数(high-cardinality)定类数据，往往不提供任何决定 y 取值的信息。这里回顾一下定类数据的特点：类别之间没有大小顺序的分类数据，这意味着不同取值之间没有顺序上的关联。关于高基数定类数据，考虑 3 种情况：第一，训练集中每一取值仅有一行，未知数据中不同数据点该特征取值皆不同且不等于训练集中该特征的任一取值；第二，训练集中每一取值的样本数皆小于可以分割取值为该数的数据点组成独立训练集所需的个数，且未知数据改特征取值有一定概率等于训练集中某些数据点该特征的取值；第三，训练集中某些取值的样本小于可以分割取值为该数的数据点组成独立训练集所需的个数，同时，某些取值的样本数大于或等于可以分割取值为该数的数据点组成独立训练集所需的个数，且未知数据改特征取值有一定概率等于训练集中某些数据点该特征的取值。

设想 User Id 被用作 X 中的一项特征输入模型训练，由于各用户编码之间没有任何关联，且每个用户编码仅在训练集中出现一次，模型能学习到的只有训练集中出现过的用户编码和其是否单击广告的联系。由于用户编码这一特征的本质，训练集中出现过的用户编码将不再出现在未知的需预测数据中，也就意味着模型在预测未知数据时，将无法有效运用从训练集中学习到的用户编码与其是否单击广告的关联，因此在上述两段代码例子中，我们皆可以选择在输入模型之前去除 User Id 这一特征。

用户编码这一高基数特征的例子较为特殊，因为一个用户编码只属于一个用户。第 2 种情况较为普遍，为方便讲解，定义一个符合第 2 种情况的特征并称为特征 T。模型将试图学习 T 中不同取值所对应的 y 值。若未知数据点中 T 的取值等于训练集某些数据点的 T 取值，由于每个取值的样本数皆较低，则模型将无法从相同的取值中学习到足够的信息。另外，由于不同取值之间并无关联，模型也无法从取值类似的数据点中学习到足够的信息。若未知数据点中 T 的取值不等于训练集中任一 T 的取值，同理，由于不同取值之间并无关联，则模型无法从 T 中学习到有效的信息。

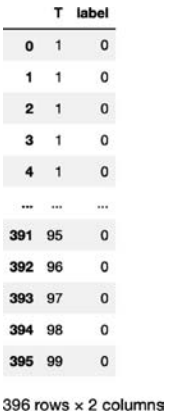
接下来，假设 T 符合第 3 种情况。对于未知数据中 T 取值不存在于训练集或在训练集中样本数量较小的数据点，同第 2 种情况的原理，模型将无法在训练时从 T 中学习到有助

于预测改数据点的信息，但训练集中同一 T 取值样本较多的数据点可以考虑单独切割，作为独立训练集训练模型。建立一个简单的、符合此情况的 DataFrame，在新的 cell 中执行：

```
df2 = pd.DataFrame({'T': [1] * 100 + [2] * 100 + [3] * 100 + list(range(4,100)),
                    'label': [0] * 98 + [1] * 2 + [0] * 95 + [1] * 5 +\
                    [1] * 100 + [0] * 96})
display(df2)
```

显示结果如图 3.2 所示。

这个简化的例子中 df2 变量的 T 列是一个符合第 3 种情况的特征，label(标签)是一个二分类的预测目标，两者皆无特殊含义。虽然通过 DataFrame 的定义可以得出 T 与 label 取值的具体分布，但在分析实际数据时，需要对两个 Series 分别使用 .value_counts()函数，清点 T 与 label 中各取值的个数，在新的 cell 中执行：



	T	label
0	1	0
1	1	0
2	1	0
3	1	0
4	1	0
...	...	...
391	95	0
392	96	0
393	97	0
394	98	0
395	99	0

396 rows × 2 columns

图 3.2 代码输出

```
print(df2['T'].value_counts())
print('\n----------\n')#分割线
print(df2['label'].value_counts())
```

输出如下：

```
1     100
2     100
3     100
37      1
28      1
     ...
69      1
70      1
71      1
72      1
50      1
Name: T, Length: 99, dtype: int64

----------

0    289
1    107
Name: label, dtype: int64
```

.value_counts()函数左列输出为取值，右列输出为该取值在 Series 中的个数，取值个数由大到小、自上到下排列。由此得出，396 个数据点中 T 取值为 1、2、3 的分别有 100 个，其余取值分别有 1 个，有 96 个不同于 1、2、3 的不同取值；label 为 0 的数据点有 289 个，label 为 1 的数据点有 107 个。当未知数据中 T 出现非 1、2、3 的取值时，模型只能根据训练集中的一个数据点进行预测，这样取得的结果是不可靠的。若将 T 取值为 1、2、3 的数据点分

割，在新的 cell 中执行：

```
T1 = df2[df2['T'] == 1]
T2 = df2[df2['T'] == 2]
T3 = df2[df2['T'] == 3]
```

打印 T1、T2 和 T3 中 label 的平均值，在下一个 cell 中执行：

```
print(T1['label'].mean(),
      T2['label'].mean(),
      T3['label'].mean())
```

输出如下：

```
0.02 0.05 1.0
```

这意味着在预测未知数据时，若未知数据 T 取值为 1 或 2，则该数据点 label 大概率为 0；若位置数据 T 取值为 3，则该数据点 label 大概率为 1。这样的推测源自 100 个数据点的取值分布，因此较为可靠。

实践中的预测比上述例子复杂许多，往往在定类数据特征之外有许多别类特征，因此训练集是否分割及如何分割这一问题并不平凡。考虑这个问题时，第一，考虑计算力的预算，使用分割的数据训练模型所需计算力较少，可以在低投入的同时估算预测准确性；第二，考虑分割后舍弃的数据点是否可能提高预测准确率——假设上述例子中在 T 这个特征之外还有许多非定类数据特征，若要将 T1 从 df2 中分割出来，需确认 T 值不等于 1 的数据点无法对 T 值等于 1 的数据点的 label 取值提供太多有效信息。举一个具体些的例子，商品的类别属于定类数据，因此方便面和洗衣粉属于不同类。假设数据中包含某日前 3～21 日的销量和产品类型作为特征，预测目标为 3 日后的销量。按照以上的方法分割，产品类型为方便面和洗衣粉的数据点将被分开输入不同的模型训练，但也许此零售公司洗衣粉的销售规律与方便面类似。理论上，模型可以通过学习方便面和洗衣粉两类产品的过往销售规律，以提高对方便面销量预测的准确性。在计算力允许的情况下，应将方便面、洗衣粉及所有过往销售规律类似的数据点聚为一类，训练专门预测此类销售规律数据点的模型。6.4 节将讲解如何聚类，获取最终用于训练的 X。

本节最后，接着 2.4 节的例子巩固对 y、X 和每行描述对象的定义思路。若以用户编码作为每行描述对象，即可直接从当前 df 中分割 X 与 y。这样定义的好处在于，若收集到的数据中形容单个用户的特征较多，特征对每个数据点的描述将较为详细。若以单个爱好及爱好拥有者组合作为每行描述对象，需先使用 2.4 节中介绍的.melt 函数，在新的 cell 中执行：

```
df = df.melt(id_vars = ['User Id', 'Age', 'Click Ads'],
             value_vars = ['Preference 1', 'Preference 2']).drop(\
             columns = ['variable']).rename(columns = {'value': 'Preference'})
display(df)
```

显示结果如图 3.3 所示。

	User Id	Age	Click Ads	Preference
0	0	16	likely	food
1	1	21	unlikely	tech
2	2	25	likely	finance
3	3	31	likely	finance
4	4	43	unlikely	tech
5	5	29	unlikely	tech
6	6	17	likely	tech
7	0	16	likely	tech
8	1	21	unlikely	finance
9	2	25	likely	tech
10	3	31	likely	food
11	4	43	unlikely	food
12	5	29	unlikely	finance
13	6	17	likely	finance

图 3.3 代码输出

如此定义每行描述对象的好处在于，若收集到的数据中形容单个爱好的特征较多，特征在详细描述用户的同时可以描述此用户的爱好。当 Preference 1 和 Preference 2 处于同一行时，若将描述两个喜好的特征直接附加到每行，多数模型无法有效认知到 X 中列与列的关系，效果不如重新定义每行的描述对象。

假设预测目标为“拥有某一喜好的用户是否单击广告”，可以确认，y 为 Click Ads。同时可以确定，X 中每行用来描述一个用户及其一个喜好的组合。重新格式化的数据中，User Id、Age 和 Preference 这 3 列用于描述单个用户及其一个喜好的特征即可作为 X。分割 DataFrame 取得 X 与 y，在新的 cell 中执行：

```
X = df[['User Id', 'Age', 'Preference']]
y = df['Click Ads']
```

3.2 训练集、验证集与测试集

3.1 节多次谈及训练集这一名词，本书虽未正式定义过训练集，却指出训练集为训练模型的数据集。验证集和测试集的定义没有训练集这样浅显易懂，两者之间也容易混淆。本节将讲解何为验证集、测试集，这二者和训练集 3 个数据集之间的关系。

3.2.1 三者的定义及关系

训练集(training set)，顾名思义，指的是用于训练模型的数据集。训练集中需包含预测目标的真实取值，用于分割为 y。就像人类学习如何解某类题目时，刚开始需要将自己的答案和正确答案进行比对，认识到自己推算答案中的错误并巩固正确解题的思路。不同于人类学习，想让机器学习规律往往需要提供大量示例，因此，训练集通常由大量历史数据组成。从本节开始将称训练集中分割出来的 y 为 y_train，X 为 X_train；验证集中分割出来的 y 为 y_val，X 为 X_val；测试集中分割出来的 y 为 y_test，X 为 X_test，以此区分不同数据集中分割出来的 X 与 y。

模型训练完成后，需要通过某种方式得知模型的准确率。验证集(validation set)用于核对模型的准确性。在完成训练的模型中输入 X_val，而后将模型根据 X_val 做出的预测与 y_val 比对，得出模型预测验证集中 y 的准确率。验证集中同样需要包含预测目标的真实取值，与训练集并无大差别，两者皆取自历史数据，但验证集对于数据点个数的要求较低，收集足够排除偶然因素的数据量即可。数据点过少，准确率可能与预测未知数据时得到的准确率均值偏差较大；数据点过多，浪费了本可加入训练集的数据。验证集可以被用于调试模型的超参数(hyperparameter)，进一步优化模型，这一步骤简称调参。第 5 章将讲解如

何调参。

使用验证集检测模型的准确率，就像是为准备一个无规定日期的考试做过大量习题后的模拟考试。模拟考试的结果用作决定是否结束训练进行真正考试的参考。模拟考试的次数太少，学生可能出于偶然得分过高而过早参加考试；次数太多，就会占据训练的时间。在反复训练的过程中，每次模拟考试的结果需要告知学生，并作为参考进行下一阶段的训练。

测试集(test set)用于模拟需要模型预测的未知数据，并对模型预测未知数据的准确率做出最终的估算。这里的未知，指的是对预测目标真实取值 y 的未知——真正需要预测的数据中无法分割出 y。而用于模拟未知数据的测试集中需要包含对应的 y 取值，与预测值进行比对提供准确率的参考。不论是测试集还是未知数据，X 皆为已知信息。测试集因为需要 y 的真实取值，也需取自历史数据。由此，可以将训练集、验证集和测试集看作分割于同一个历史数据集的 3 个不同功能的数据集。

若使用验证集检测的模型准确率达标，即可开始真正的预测。

测试集和验证集最大的区别在于，使用验证集得到的准确率可以用来调试模型，而测试集得到的准确率，不论好坏，都不应该用于调试模型。举个稍为具体的例子总结以上 3 个数据集用法的区别：假设有 10 个不同的模型可供选择，10 个模型皆使用同一个训练集训练。这时如果想要在 10 个模型中选出 1 个为最优模型，使用训练集进行评估明显不合适——将训练集中每个数据点的 y 取值“背过”的模型不一定有准确预测未知数据的能力，因此，10 个模型对验证集的预测准确率便成为评判更优模型的标准。由此筛选的最优模型，称其为 M，在某种程度上也取得了验证集中 y 的信息，换而言之，验证集对于 M 不算是绝对的未知信息，无法模拟未知数据，因此，M 对验证集预测的准确率不适于估算 M 预测未知数据时的准确率。

这里你可能会有所疑惑：M 既没有使用验证集进行训练，也没有直接观察到验证集中 X 所对应的 y，如何取得验证集中 y 的信息？这与 M 的定义有关。M 是多个不同模型中预测验证集准确率最高的模型。设想可供选择的模型不止 10 个，而是 1000 个或更多。每个模型或多或少存在偶然将某一数据点预测正确的概率，因此，在众多预选模型中挑出的最优模型，有一定概率是在没有学习到 X 与 y 之间真实关系的情况下，偶然对验证集取得较高的预测准确率。在选择 M 的同时，相当于透露了验证集中 y 取值最接近的一组预测。这就像一张只有选择题且所有选择题选项有限的模拟考卷。如果在不知答案的情况下做了 1000 遍，并根据答案选择最高得分考量学生的准备程度，这样的预估显然过于乐观。

训练集、验证集和测试集都属于模型调试阶段使用的数据集，而模型的调试往往需要多次迭代，因此，测试集中 y 的信息虽然没有如验证集一样直接用于选择最优模型，却也或多或少由于人为因素影响着最后模型的选择。例如，在调试的过程中可能遇到以下情况：训练后的模型经过验证集的筛选，选出了最优的模型 M1。在使用 M1 预测测试集中 y 的取值并与 y_test 进行比对时，却发现准确率远低于 M1 预测验证集时取得的准确率。一定程度上的准确率下降可能属于正常现象，正如上一个例子所述，M1 是多个不同模型中预测验证集准确率最高的模型，因此对于验证集的预测可能高于其他数据集，但需警惕大幅度的差异值。测试集与训练集、验证集分割自同一个历史数据集，理论上这 3 个数据集的 X 与 y 之间的关系相似，但也存在不同数据集数据分布不类似的可能性。若这种情况成立，且无法

判断这种分布差异的来源，则基本可以确定，现有的特征无法预测目标问题。不论在历史数据的预测中达到多高的准确率，都无法预测未来数据的规律会发生什么样的改变。

假设上述情况不成立，换而言之，数据在分布上并不存在集与集之间大的改变，那么M1得以在验证集预测中取得最优的成绩，而无法在预测测试集时取得类似的成绩，问题就出在模型上。经过验证集的调参，M1可能出于偶然，正好是一个没有掌握数据规律却猜对了验证集中大多数 y 取值的模型。了解这一问题后，我们可能会重新调参，筛选最优模型。假设类似情况出现多次迭代，每次选择的最优模型预测验证集的准确率都类似，而预测测试集时准确率时高时低，同时都与验证集有所差距。到了第5次迭代为最优模型，称为M5，在验证集和测试集的准确率终于接近，这时，我们是否能较为肯定地使用M5进行未来数据的预测，并预估其与预测测试集的准确率相近？答案是不能十分肯定。虽然M5没有使用测试集调参，但在人为观察到前4次迭代的最优模型预测测试集的准确率皆较低，而决定再次调参，直到测试集的预测准确率提升时，我们已经使用了测试集的信息筛选M5。

测试集有限时，若多次使用同一测试集预估未来准确率，并根据人为观察预测结果决定是否需要重新筛选最优模型，容易使预测测试集的准确率高于预测未来数据时可以取得的准确率。而多个迭代的筛选往往是难免的。在这个两难的局面下，为避免过分高估未来预测的准确率，第一，需要在每次进入下一次迭代的调试前思考两次迭代的差异。例如，下一次迭代是否使用到不同范围的参数，是否使用不同的模型等，并思考下一次迭代中做出的改变是否可能改善不同数据集准确率不一致的问题。合理地增加迭代，可以避免过多接触测试集 y 的信息。第二，每次迭代中若出现不同数据集准确率不一致的问题，需了解模型中什么样的结构引发了这一问题，并确认最终用于预测未来数据的最优模型中没有类似的结构。3.4节偏差与方差中将讲解一些可能造成此问题的原因。

3.2.2 如何使用 sklearn 分离 3 个集

本节讲解如何使用sklearn进行最基础的3集分离。由于模型的优化不属于基础建模这一步骤，关于更多分离验证集的方法将在第5章讲解。

3.1.1节提到，3个不同功能的数据集皆取自历史数据，因此，最直接的方法是为3集设定合理的比例，并根据比例从历史数据中分割。

训练集中的数据点相对较多，而用于优化和评估模型的验证集与测试集所需数据点相对较少。一个常用的训练集：验证集：测试集数据点个数的比例是70：15：15。使用sklearn的model_selection模块中的train_test_split函数可以将输入的数据集和比例分成两个数据集。如此分割两次，即可获得3个符合比例的数据集。首先，建立一个代表历史数据的DataFrame。这个DataFrame只有Id(编码)、Age(年龄)和target(目标)3列信息，可以更清楚地看到分割的情况，执行：

```
import pandas as pd

df = pd.DataFrame({'Id': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
                   'Age': [17, 30, 29, 69, 15, 58, 42, 32, 36, 70],
                   'target': [0, 1, 1, 0, 0, 0, 1, 1, 1, 0]})
display(df)
```

	Id	Age	target
0	1	17	0
1	2	30	1
2	3	29	1
3	4	69	0
4	5	15	0
5	6	58	0
6	7	42	1
7	8	32	1
8	9	36	1
9	10	70	0

图 3.4　代码输出

显示结果如图 3.4 所示。

使用 train_test_split 函数，首先分割出占比 70%的训练集，另外 30%将同时包含验证集和测试集；再从 30%的数据中分割出占比 50%的验证集，剩下的 50%作为测试集。在下一个 cell 中执行：

```
#导入 train_test_split 函数
from sklearn.model_selection import train_test_split

df_train, df_val_test = train_test_split(df, test_size = 0.3,
random_state = 42)
```

函数 train_test_split 参数第 1 项输入一个序列或者一个序列需要分割的 DataFrame，使用参数 train_size 或 test_size 进行分割。参数 random_state 作为伪随机数生成器的种子，在同一 random_state 下，“随机”分割的结果相同。train_size 决定了函数返回的 DataFrame 序列中第 1 个 DataFrame 的数据量占比；test_size 决定了函数返回的 DataFrame 序列中第 2 个 DataFrame 的数据量占比。上述代码中，设定 test_size＝0.3，将 70%的数据分入 df_train 中，30%的数据分入 df_val_test 中。至此，df_train 已经成为符合数据量占比的训练集，接下来分割验证集和测试集，并显示 3 个分割后的数据集。在下一个 cell 中执行：

```
df_val, df_test = train_test_split(
                  df_val_test, test_size = 0.5, random_state = 42)
display(df_train, df_val, df_test)
```

显示结果如图 3.5 所示。

	Id	Age	target
0	1	17	0
7	8	32	1
2	3	29	1
9	10	70	0
4	5	15	0
3	4	69	0
6	7	42	1

	Id	Age	target
5	6	58	0

	Id	Age	target
8	9	36	1
1	2	30	1

图 3.5　代码输出

原历史数据 df 中有 10 个数据点，分割后的训练集 df_train 占比 70%，拥有 7 个数据点；验证集和测试集各占比 15%，但因数据点个数需为整数，验证集中本应有 1.5 个数据点，向下取整得 1，测试集中本应有 1.5 个数据点，向上取整得 2。示例中显示的各集数据量符合这一计算。

函数 train_test_split 中 stratify 这一参数可用于控制某列不同取值在 3 个数据集中的比例。为演示这一参数的作用，定义一个包含 100 个数据点的 DataFrame。在下一个 cell 中执行：

```
df2 = pd.DataFrame({'Id': list(range(100)),
                    'type': [1, 2, 3, 4, 5] * 20,
                    'target': [0] * 50 + [1] * 50})
```

从 df2 的定义中可知，df2 中 type(类型)这一特征有 5 个取值，每个取值各占 20%的数据点；target 这一特征有 2 个取值，每个取值占 50%的数据点。重复上一个例子的分割方法，并打印分割后 3 个数据集中 type 和 target 两个特征的取值分布，在下一个 cell 中执行：

```
#Chapter3/stratify_0.ipynb

df_train, df_val_test = train_test_split(df2, test_size=0.3, random_state=42)
df_val, df_test = train_test_split(
                  df_val_test, test_size=0.5, random_state=42)

#打印 3 个分割后数据集中 type 和 target 的数据分布
print('df_train, type')
print(df_train['type'].value_counts(), '\n')
print('df_train, target')
print(df_train['target'].value_counts(), '\n')

print('df_val, type')
print(df_val['type'].value_counts(), '\n')
print('df_val, target')
print(df_val['target'].value_counts(), '\n')

print('df_test, type')
print(df_test['type'].value_counts(), '\n')
print('df_test, target')
print(df_test['target'].value_counts(), '\n')
```

输出如下：

```
df_train, type
2    16
5    15
3    15
4    14
1    10
Name: type, dtype: int64

df_train, target
1    37
0    33
Name: target, dtype: int64

df_val, type
1    7
4    3
```

```
3    3
2    2
Name: type, dtype: int64

df_val, target
0    8
1    7
Name: target, dtype: int64

df_test, type
5    5
4    3
1    3
3    2
2    2
Name: type, dtype: int64

df_test, target
0    9
1    6
Name: target, dtype: int64
```

从输出中可以看出，3 个数据集中 type 和 target 的取值分布皆没有保持 df2 中这两列的取值分布，接下来将分开讨论两列数据在不同集中取值分布不同的后果。首先需要注意的一点是，未来数据的任意一列取值分布，将有大概率与完整的历史数据集中该列取值分布相似。训练集占比较大，因此取值分布更大概率接近完整历史数据，而占比较小的验证集于测试集中就容易出现与原分布相差较远的取值分布。当 type 取值分布不同时，测试集相较完整历史数据集而言，会过分侧重某些类 type，而轻视某些类 type。例如上段代码输出中，df_test 中 type 为 5 的占比约为 33%，高于原占比的 20%；type 为 3 和 5 的占比分别约为 13%，低于原占比。这会导致预测测试集的准确率更接近预测 type 为 5 的数据的准确率。若模型 type 为 5 的数据的准确率高于预测 type 为 2 和 3 的数据，使用分布如此的测试集预估的未来数据准确率将过于乐观。

举一个较为极端的具体例子：假设模型对于 type 为 5 的数据的预测准确率为 90%，对于 type 为任意其他值的预测准确率为 35%。这里补充一条说明：对于不同 type 预测的准确率差异完全合理，因为不同 type 的数据规律存在或多或少的差异，而某些规律更容易被模型学习。由于完整历史数据中每种 type 占比相同，可以合理推测，未来数据中每种 type 的占比大概率也相同。预测未来数据的准确率可以由以下公式计算：$\frac{0.9+0.3\times4}{5}=0.42$。使用测试集预估的准确率为 $0.9\times\frac{1}{3}+0.3\times\frac{2}{3}=0.5$，是一个过于乐观的预估。

当 target 取值分布不同时，与 type 取值不同的后果类似。例如 df_test 中 target 取值为 0 的数据是取值为 1 的数据的 1.5 倍，而完整历史数据集中两者占比相等。这会导致预测测试集的准确率更接近预测 target 为 0 的数据的准确率。一个预测测试集准确率整体较高的模型，可能对真实 target 为 0 的数据点预测准确率较高，而对真实 target 为 1 的数据点

预测准确率较低。这也意味着，预测未来数据时，假阴性的比例将比假阳性高。

考虑到上述后果，使用 stratify(分层)参数保证分割的 3 集某列取值分布相似。稍微修改上段代码，在 stratify 中输入单列数据并执行，代码如下：

```
#Chapter3/stratify_1.ipynb

df_train, df_val_test = train_test_split(df2, test_size=0.3, random_state=42, stratify=
df2['type'])
df_val, df_test = train_test_split(
                  df_val_test, test_size=0.5, random_state=42,
stratify=df_val_test['type'])

#打印 3 个分割后数据集中 type 和 target 的数据分布
print('df_train, type')
print(df_train['type'].value_counts(), '\n')
print('df_train, target')
print(df_train['target'].value_counts(), '\n')

print('df_val, type')
print(df_val['type'].value_counts(), '\n')
print('df_val, target')
print(df_val['target'].value_counts(), '\n')

print('df_test, type')
print(df_test['type'].value_counts(), '\n')
print('df_test, target')
print(df_test['target'].value_counts(), '\n')
```

输出如下：

```
df_train, type
5    14
4    14
3    14
2    14
1    14
Name: type, dtype: int64

df_train, target
0    36
1    34
Name: target, dtype: int64

df_val, type
5    3
4    3
3    3
2    3
```

```
1    3
Name: type, dtype: int64

df_val, target
1    8
0    7
Name: target, dtype: int64

df_test, type
5    3
4    3
3    3
2    3
1    3
Name: type, dtype: int64

df_test, target
1    8
0    7
Name: target, dtype: int64
```

3 集中不同 type 的取值比例皆相等。

想要同时对多列执行嵌套分层，不建议使用 stratify 参数并输入多列特征。当 stratify 参数为多列时，train_test_split 会将多列中所有不同取值的总和作为不同类进行成比例分割。例如在 df2 中，在 stratify 参数中输入 df2[['target', 'type']]，train_test_split 会认为一共存在从 0～6 的 6 个取值，且不会将 target 和 type 中的 1 作区分。3.2.3 节将讲解如何手动进行多列执行嵌套分层。

若历史数据中存在时间关系，且预测问题也与时间有关，则需保证训练集中的数据时间前于验证集，而验证集中的数据时间前于测试集。例如预测 3 日后方便面销量这一问题，假设收集到 2017 年至 2019 年这 3 年的历史销量数据，并准备将其分割为训练集、验证集和测试集。直接使用 train_test_split，按比例随机抽取分别选入 3 集的数据点，会导致某些出现在测试集中的数据点时间前于训练集中的数据点。这就意味着，一个经过未来数据训练的模型在对过去的数据进行预测。例如在测试集中可能存在 2018 年 5 月的数据，而训练集中却有 2018 年 6 月的数据。这样的预测结果不能合理预估模型预测未知数据的准确率。为确保 3 集之间的时间顺序，可以使用 sklearn 中的 TimeSeriesSplit 交叉验证器进行分割。需要注意的是，TimeSeriesSplit 是一个 Python class，使用时需要先定义一个类为 TimeSeriesSplit 的对象，再使用对象所带的函数。

5.3 节会详细讲解交叉验证，以及如何在交叉验证中使用交叉验证器。本节仅作简单的介绍。交叉验证器会对输入的 DataFrame 进行多次分割。TimeSeriesSplit 将输入的 DataFrame 分为 $n+1$ 等份，并输出 n 对符合时间排序的 DataFrame。假设分为 $n+1$ 等份的 DataFrame 编码，编号小的 DataFrame 储存时间更靠前的数据，那么第 i 次分割的输出中一个数据集包含编码为 1～i 个等份，以及一个仅包含编码为 $i+1$ 等份的数据集。在下一个 cell 中定义一个数据点间存在时间关系的 DataFrame，执行：

```
import random

time_lst = list(range(100))
random.shuffle(time_lst) #打乱时间顺序
df_time = pd.DataFrame({'time': time_lst,
                        'sales': [random.randint(1,10) for _ in range(100)]})
```

在 df_time 中，time(时间)列记录每个数据点对应的时间，取值为 1～100 的一个整数，其中越小的数表示越靠前的时间点，时间不能重复；sales(销售)列记录该时间点的销售量。使用 TimeSeriesSplit 之前需保证 DataFrame 本身自上到下满足时间排序，而 df_time 中的时间点是随机的，因此需先对 df_time 进行排序。对一个 Pandas Series 使用.sort_values 函数，并在 by(根据)参数中输入用于排序的列，可以将所有行按照该列取值的大小顺序排列。在下一个 cell 中执行：

```
df_time.sort_values(by = ['time'], inplace = True)
df_time.reset_index(drop = True, inplace = True)
display(df_time)
```

为了之后使用指数索引时间更靠前的数据点，这里使用 reset_index 重新为指数排序。上段代码显示结果如图 3.6 所示。

	time	sales
0	0	7
1	1	5
2	2	10
3	3	6
4	4	5
...	...	...
95	95	2
96	96	5
97	97	10
98	98	8
99	99	7

100 rows × 2 columns

图 3.6 代码输出

对 df_time 使用 TimeSeriesSplit 方法如下：

```
#Chapter3/time_series_0.ipynb

#导入 class
from sklearn.model_selection import TimeSeriesSplit

#n_splits 决定分割次数，默认为 5
ts_split = TimeSeriesSplit(n_splits = 3)
for train_indices, val_test_indices in ts_split.split(df_time):
    #返回的一对数据集中左集大于或等于右集，因此选择左集作为训练集
    #右集作为验证集和测试集的总和
    df_train, df_val_test = df_time.loc[train_indices],\
                            df_time.loc[val_test_indices]
    print("训练集的最大时间值：", df_train['time'].max(),
          "验证集加测试集的最小时间值：", df_val_test['time'].min())
    print("训练集的大小：", df_train['time'].count(),
          "验证集加测试集的大小：", df_val_test['time'].count())
    print()
```

输出如下：

```
训练集的最大时间值：24          验证集加测试集的最小时间值：25
训练集的大小：25                验证集加测试集的大小：25
```

```
训练集的最大时间值: 49      验证集加测试集的最小时间值: 50
训练集的大小: 50            验证集加测试集的大小: 25

训练集的最大时间值: 74      验证集加测试集的最小时间值: 75
训练集的大小: 75            验证集加测试集的大小: 25
```

由此可见,每次分割的训练集中数据点的时间皆小于验证集和测试集中数据点的时间。这里的 n 为 3,输入的 DataFrame 被分为 4 等份,每一等份包含 25 个数据点。3 次分割中,第 3 次分割的训练集占比最接近 70%,第 2 次分割的占比最接近 50%。使用一个计数变量 count,确保 df_train 和 df_val_test 的赋值为第 3 次分割的一对数据集,修改上段代码,执行:

```
#Chapter3/time_series_1.ipynb

#导入 class
from sklearn.model_selection import TimeSeriesSplit

#n_splits 决定分割次数,默认为 5
ts_split = TimeSeriesSplit(n_splits=3)
count = 0

for train_indices, val_test_indices in ts_split.split(df_time):
#count 用于记录分割的次数
    count += 1
    if count != 3:
        continue#跳过此次分割
    else:
        df_train, df_val_test = df_time.loc[train_indices],\
                                df_time.loc[val_test_indices]

#分割结束后确认两集的大小
print("训练集的最大时间值: ", df_train['time'].max(),
      "验证集加测试集的最小时间值: ", df_val_test['time'].min())
print("训练集的大小: ", df_train['time'].count(),
      "验证集加测试集的大小: ", df_val_test['time'].count())
```

输出结果如下:

```
训练集的最大时间值: 74      验证集加测试集的最小时间值: 75
训练集的大小: 75            验证集加测试集的大小: 25
```

通过同样的方法,在下一个 cell 中将 df_val_test 分割成两个占比相等的数据集,分别为验证集和测试集,执行:

```
#Chapter3/time_series_1.ipynb

#使用 reset_index 重新为 df_val_test 的指数排序,为之后使用指数分割做准备
```

```
df_val_test.reset_index(drop = True, inplace = True)

#重新定义 TimeSeriesSplit 对象,设定 n = 2,分割 df_val_test
ts_split = TimeSeriesSplit(n_splits = 2)

#重置计数变量
count = 0

for val_indices, test_indices in ts_split.split(df_val_test):
    count += 1
    if count != 2:
        continue#跳过此次分割
    else:
        #唯有第 2 次分割的一对数据集使用到 df_val_test 中所有数据点(25 个)
        df_val, df_test = df_val_test.loc[val_indices],\
                          df_val_test.loc[test_indices]

#分割结束后确认两集的大小
print("验证集的最大时间值: ", df_val['time'].max(),
      "测试集的最小时间值: ", df_test['time'].min())
print("验证集的大小: ", df_val['time'].count(),
      "测试集的大小: ", df_test['time'].count())
```

输出结果如下:

```
验证集的最大时间值: 91      测试集的最小时间值: 92
验证集的大小: 17            测试集的大小: 8
```

若想让验证集和测试集使用到 df_val_test 中全部数据点,需要在最后一次分割时赋值。已知最后一次分割时的一对数据集大小比例为 n ∶ 1,且 TimeSeriesSplit 中的 n_split 最小可取值为 2,设定 $n=2$,使这个比例最接近 1 ∶ 1。

最后,在下一个 cell 中显示分割后的训练集、测试集和验证集,执行:

```
display(df_train.head(), df_val.head(), df_test.head())
```

显示结果如图 3.7 所示。

	time	sales
0	0	7
1	1	2
2	2	6
3	3	7
4	4	9

	time	sales
0	75	10
1	76	10
2	77	2
3	78	5
4	79	3

	time	sales
17	92	4
18	93	8
19	94	9
20	95	3
21	96	9

图 3.7 代码输出

3.2.3 如何使用 Pandas 手动分离 3 个集

通过 3.2.2 节的例子可以看出,在特定情况下,使用 sklearn 中的函数并不能完全按照

意愿分割训练集、验证集和测试集。例如在分割时间序列的例子中，若想使用原历史数据中的全部数据点，只能使用第 n 次分割的那对数据集，而该对数据集的比例为 $n:1$，因此，分割后的一对数据无法完全符合某些预想的比例。另外，如 3.2.2 节中提到，train_test_split 函数中的 stratify 参数无法针对多列执行嵌套分层。本节将讲解如何直接使用 Pandas，完成 3.2.2 节的随机分割和时间序列分割。

重新定义 3.2.2 节中的 df2，执行：

```
import pandas as pd

df2 = pd.DataFrame({'Id': list(range(100)),
                    'type': [1, 2, 3, 4, 5] * 20,
                    'target': [0] * 50 + [1] * 50})
```

接下来，执行对 Id 和 type 两列的嵌套分层。首先，定义何为单列分层及如何使用 Pandas 进行单列分层。3.2.2 节的例子中使用到的单列分层，根据 DataFrame 中某一列的数据取值比例分割数据集，保证分割后的数据集中该列不同取值之间比例不变。使用 Pandas 根据 type 列取值比例分层，代码如下：

```
#Chapter3/stratify_2.ipynb

train_ratio = 0.7          #定义训练集的比例
val_ratio = 0.15           #定义验证集的比例
test_ratio = 0.15          #定义测试集的比例

#定义 3 个空集，每个集会在接下来的 for 循环中逐步添加数据点
train_df = pd.DataFrame()
val_df = pd.DataFrame()
test_df = pd.DataFrame()

for value in df2['type'].unique():
#取特定比例(train_ratio)的 df2 中 type 取值为 value 的数据点，加入训练集
    train_df = train_df.append(df2[df2['type'] == value].sample(\
                               frac = train_ratio, random_state = 42))

#取特定比例(val_ratio)的 df2 中 type 取值为 value，
#且未加入训练集的数据点加入验证集
#因为某些数据点已经加入训练集，而 sample 函数中 frac 参数需为相对该 DataFrame 的样本比例
#重新计算比例
    new_val_ratio = val_ratio / (1 - train_ratio)
    val_df = val_df.append(df2[(df2['type'] == value) &\
                               (~df2['Id'].isin(train_df['Id'].unique()))].sample(\
                               frac = new_val_ratio, random_state = 42))

#取特定比例(test_ratio)的 df2 中 type 取值为 value，
#且未加入训练集或验证集的数据点加入验证集
#因为某些数据点已经加入训练集和验证集
#而 sample 函数中 frac 参数需为相对该 DataFrame 的样本比例
```

```
    #重新计算比例.若 train_ratio + val_ratio + test_ratio = 1,new_test_ratio = 1
        new_test_ratio = test_ratio / (1 - train_ratio - val_ratio)
        test_df = test_df.append(df2[(df2['type'] == value) &\
                                     (~df2['Id'].isin(train_df['Id'].unique())) &\
                                     (~df2['Id'].isin(val_df['Id'].unique()))].sample(\
                                     frac = new_test_ratio, random_state = 42))
```

此方法分割保持取值占比的原理在于,for 循环中每次迭代针对 df2 中 type 的一个取值,并从 type 满足该取值的数据点中抽取每集目标占比的样本数,按比例分别加入 3 个集。分割时需注意两点,第一,确保训练集、验证集和测试集的数据点不重合;第二,使用 .sample 函数抽取某 DataFrame 样本时,须确保用于设定抽取比例的 frac 参数准确。这两点皆在代码注释中有更详细的讲解。

在下一个 cell 中,验证分割后的 3 集取用了 df2 中的所有数据点,且 3 集之间数据点无重合。执行:

```
print(len(train_df.append(val_df).append(test_df)['Id'].unique()))
```

输出如下:

```
100
```

由于原数据集中不同行的 Id 不重复,如此证明,分割后的 3 集无重合的数据点,且 3 集的总和为完整的 df2。然后,验证训练集、验证集和测试集 3 集之间的数据点个数比例是否为 70∶15∶15,在下一个 cell 中执行:

```
print('训练集占比: ', len(train_df) / len(df2))
print('验证集占比: ', len(val_df) / len(df2))
print('测试集占比: ', len(test_df) / len(df2))
```

输出如下:

```
训练集占比: 0.7
验证集占比: 0.15
测试集占比: 0.15
```

证明代码运行结果中 df_train、df_val 和 df_test 相对 df2 的比例符合定义的 train_ratio、val_ratio 和 test_ratio。

最后,验证 3 集中 type 取值的占比等同于该取值在 df2 中的原占比。使用 .value_counts 函数,并设定 normalize 参数为 True,打印取值占比。在下一个 cell 中执行:

```
#Chapter3/stratify_2.ipynb

print('df2 中各取值占比: ')
print(df2['type'].value_counts(normalize = True))
```

```
print()
print('训练集中各取值占比: ')
print(train_df['type'].value_counts(normalize = True))
print()
print('验证集中各取值占比: ')
print(val_df['type'].value_counts(normalize = True))
print()
print('测试集中各取值占比: ')
print(test_df['type'].value_counts(normalize = True))
```

输出如下：

```
df2 中各取值占比:
5    0.2
4    0.2
3    0.2
2    0.2
1    0.2
Name: type, dtype: float64

训练集中各取值占比:
5    0.2
4    0.2
3    0.2
2    0.2
1    0.2
Name: type, dtype: float64

验证集中各取值占比:
5    0.2
4    0.2
3    0.2
2    0.2
1    0.2
Name: type, dtype: float64

测试集中各取值占比:
5    0.2
4    0.2
3    0.2
2    0.2
1    0.2
Name: type, dtype: float64
```

各集中 type 取值分布相等。

了解单列分层的思路后，可以开始定义嵌套分层并使用 Pandas 执行对 Id 和 type 两列的嵌套分层。嵌套分层根据 DataFrame 中多列数据取值比例分割数据集。多列数据的“取值”，指的是每列取值的组合，可以将每个取值想象为一个 Python tuple。若两个数据点每

列取值对应相等，则二者在分层问题中被归为同一取值的数据点。如单列分层问题，嵌套分层保证分割后的数据集中不同取值之间比例不变，而这里的取值，定义为用于分层的多个选定列取值。

回到 df2 这个具体例子中，type 和 target 两列作为用于分层的列。type 和 target 组合一共有 10 种取值：type＝1，target＝0；type＝1，target＝1；type＝2，target＝0；type＝2，target＝1；type＝3，target＝0；type＝3，target＝1；type＝4，target＝0；type＝4，target＝1；type＝5，target＝0；type＝5，target＝1。已知 df2 中每种取值占比均为 10％，分割后的 3 集中每个取值占比也应为 10％。使用与单列分层时类似的方法：

```
train_ratio = 0.7       #定义训练集的比例
val_ratio = 0.15        #定义验证集的比例
test_ratio = 0.15       #定义测试集的比例

#定义 3 个空集，每个集会在接下来的 for 循环中逐步添加数据点
train_df = pd.DataFrame()
val_df = pd.DataFrame()
test_df = pd.DataFrame()

for type_value in df2['type'].unique():
    for target_value in df2['target'].unique():
        #取特定比例(train_ratio)的 df2 中 type 取值为 type_value,
        #且 target 取值为 target_value 的数据点，加入训练集
        train_df = train_df.append(df2[(df2['type'] == type_value) &\
                                       (df2['target'] == target_value)].sample(\
                                       frac = train_ratio, random_state = 42))

        #取特定比例(val_ratio)的 df2 中 type 取值为 type_value,
        #target 取值为 target_value，且未加入训练集的数据点加入验证集
        #因为某些数据点已经加入训练集
        #而 sample 函数中 frac 参数需为相对该 DataFrame 的样本比例，重新计算比例：
        new_val_ratio = val_ratio / (1 - train_ratio)
        val_df = val_df.append(df2[(df2['type'] == type_value) &\
                                   (df2['target'] == target_value) &\
                               (~df2['Id'].isin(train_df['Id'].unique()))].sample(\
                                frac = new_val_ratio, random_state = 42))

        #取特定比例(test_ratio)的 df2 中 type 取值为 type_value,
        #target 取值为 target_value，且未加入训练集或验证集的数据点加入验证集
        #因为某些数据点已经加入训练集和验证集
        #而 sample 函数中 frac 参数需为相对该 DataFrame 的样本比例.重新计算比例：
        #若 train_ratio + val_ratio + test_ratio = 1，则 new_test_ratio = 1
        new_test_ratio = test_ratio / (1 - train_ratio - val_ratio)
        test_df = test_df.append(df2[(df2['type'] == type_value) &\
                                     (df2['target'] == target_value) &\
                                (~df2['Id'].isin(train_df['Id'].unique())) &\
                                (~df2['Id'].isin(val_df['Id'].unique()))].sample(\
                                 frac = new_test_ratio, random_state = 42))
```

两个嵌套的 for 循环，里层循环的每次迭代针对 df2 中 type 和 target 的一个取值组合，并从 type 和 target 满足该取值的数据点中抽取每集目标占比的样本数，按比例分别加入 3 个集。这里需要注意的点与单列分割时类似，在代码中有详细注释。

在下一个 cell 中，验证分割后的 3 集取用了 df2 中的所有数据点，且 3 集之间数据点无重合。执行：

```
print(len(train_df.append(val_df).append(test_df)['Id'].unique()))
```

输出如下：

```
100
```

由于原数据集中不同行的 Id 不重复，由此证明，分割后的 3 集无重合的数据点，且 3 集的总和为完整的 df2。验证训练集、验证集和测试集 3 集之间的数据点个数比例是否为 70∶15∶15，在下一个 cell 中执行：

```
print('训练集占比：', len(train_df) / len(df2))
print('验证集占比：', len(val_df) / len(df2))
print('测试集占比：', len(test_df) / len(df2))
```

输出如下：

```
训练集占比：0.7
验证集占比：0.1
测试集占比：0.2
```

由此可见，df_train、df_val 和 df_test 相对 df2 的比例不完全符合定义的 train_ratio、val_ratio 和 test_ratio。这是因为，使用两列数据分层时，每列不同取值只占据 10 个数据点。也就意味着，10 个数据点中 7 个数据点需分为训练集，剩下 3 个数据点的 50%分入验证集，50%分入测试集。3×0.5=1.5，验证集中本应有 1.5 个数据点，向下取整得 1；测试集中本应有 1.5 个数据点，向上取整得 2。

最后，验证 3 集中各 type 和 target 取值组合的占比等同于该取值在 df2 中的原占比。在下一个 cell 中执行：

```
#Chapter3/stratify_3.ipynb

for type_value in df2['type'].unique():
    for target_value in df2['target'].unique():
        print('type 取值为{t}, '.format(t = type_value) + \
              'target 取值为{v}的取值组合比例分别为'.format(v = target_value))
        print('df2: ', len(df2[(df2['type'] == type_value) &\
                        (df2['target'] == target_value)]) / len(df2),
              '; 训练集：', len(train_df[(train_df['type'] == type_value) &\
                              (train_df['target'] == target_value)]) / len(train_df),
              '; 验证集：', len(val_df[(val_df['type'] == type_value) &\
```

```
                        (val_df['target'] == target_value)]) / len(val_df),
            '; 测试集: ', len(test_df[(test_df['type'] == type_value) &\
                         (test_df['target'] == target_value)]) / len(test_df))
    print()
```

输出如下：

```
type 取值为 1, target 取值为 0 的取值组合比例分别为
df2: 0.1 ; 训练集: 0.1 ; 验证集: 0.1 ; 测试集: 0.1

type 取值为 1, target 取值为 1 的取值组合比例为
df2: 0.1 ; 训练集: 0.1 ; 验证集: 0.1 ; 测试集: 0.1

type 取值为 2, target 取值为 0 的取值组合比例为
df2: 0.1 ; 训练集: 0.1 ; 验证集: 0.1 ; 测试集: 0.1

type 取值为 2, target 取值为 1 的取值组合比例为
df2: 0.1 ; 训练集: 0.1 ; 验证集: 0.1 ; 测试集: 0.1

type 取值为 3, target 取值为 0 的取值组合比例为
df2: 0.1 ; 训练集: 0.1 ; 验证集: 0.1 ; 测试集: 0.1

type 取值为 3, target 取值为 1 的取值组合比例为
df2: 0.1 ; 训练集: 0.1 ; 验证集: 0.1 ; 测试集: 0.1

type 取值为 4, target 取值为 0 的取值组合比例为
df2: 0.1 ; 训练集: 0.1 ; 验证集: 0.1 ; 测试集: 0.1

type 取值为 4, target 取值为 1 的取值组合比例为
df2: 0.1 ; 训练集: 0.1 ; 验证集: 0.1 ; 测试集: 0.1

type 取值为 5, target 取值为 0 的取值组合比例为
df2: 0.1 ; 训练集: 0.1 ; 验证集: 0.1 ; 测试集: 0.1

type 取值为 5, target 取值为 1 的取值组合比例为
df2: 0.1 ; 训练集: 0.1 ; 验证集: 0.1 ; 测试集: 0.1
```

由以上输出可见，每个分割集中各取值组合比例皆保持为 0.1，与原 df2 中该组合取值比例相等。

3.3 数据泄露

数据泄露(data leakage)，指的是训练集中包含某些有关 y 取值，且无法在预测未来数据时获取的信息。这里所指的有关 y 取值的信息，包括任何被选作 X 中特征的数据。这可能导致模型预测训练集、验证集和测试集 3 集数据的准确率远高于预测未来数据的准确率。本节将列举各种类型的数据泄露，以及如何尽早发现并规避可能存在的数据泄露。

3.3.1 不同类型的数据泄露

数据泄露的方式分两大类。第一类，有关 y 取值的信息由直接或间接的方式出现在各集 X 的某些特征中，且该特征在预测未来数据时无法及时收集，这类数据泄露也被称为目标泄露(target leakage)；第二类，训练集的 X 中某些特征使用了测试集的信息，这类数据泄露也可以称为训练集与测试集相互污染(train-test contamination)。本节将分别讲解什么样的错误会导致这两类数据泄露。

最极端的目标泄露是将 y 的真实取值直接作为一个 X 中的特征并训练、验证、测试模型，此特征称为 t。在数据点充足的情况下，大多数模型可以将该特征取值与 y 取值联系起来。当下一次输入列数相同的 X 并使用模型预测其对应的 y 时，模型会直接输出该特征数值。这种情况听似离谱，但在实践中，由于 X 和 y 许多时候分割于同一个数据集，不经意间可能会在 X 中保留与 y 值相等的特征。显然，在预测未来数据时无法获取特征 t 的取值，而 t 的取值极大程度上包含了 y 取值的信息，因此，t 在训练集、验证集和测试集中的存在被称为目标泄露。

当特征与预测目标之间存在时间关系时，也需特别留意可能存在的目标泄露。例如 3.1.1 节中提到，预测 3 日后冰激凌的销量，若存在“昨日销量”这一特征，则涉及数据泄露。昨日销量这一特征称为 $t1$，$t1$ 或多或少影响相对其时间点的明日销量，也就是目标日销量 y，因此，将 $t1$ 设定为训练集、验证集和测试集中的特征将导致目标泄露。在实践中，由于训练集、验证集和测试集皆取自历史数据库，所以在设计特征时，若只考虑可以从历史数据库中提取的有关信息，而不考虑该信息与其对应的目标点之间的时间关系，则很有可能落入类似陷阱。

时间关系中的一个特例为因果关系。某些时候，现有的表格中存在一些看似对预测目标有益的特征。例如以下代码所创立的 DataFrame，执行：

```
#Chapter3/data_leakage_0.ipynb

import pandas as pd

df = pd.DataFrame({'Patient id': list(range(10)),
                   'Height (cm)': [160, 180, 170, 154,
                                   159, 174, 165, 177,
                                   184, 168],
                   'Gender': ['Female', 'Male', 'Male', 'Female',
                              'Female', 'Female', 'Female', 'Male',
                              'Male', 'Male'],
                   'Weight (kg)': [60, 90, 59, 49, 53,
                                   80,45, 70, 79, 68],
                   'Eats sugar products': [False, False, True, False,
                                           True, False, False, True,
                                           True, True],
                   '...': ['...'] * 10,#...用于表示省略许多其他特征
                   'Has diabetes': [True, True, False, True,
                                    False, True, False, False,
                                    False, False]})
display(df)
```

显示结果如图 3.8 所示。

	Patient id	Height (cm)	Gender	Weight (kg)	Eats sugar products	...	Has diabetes
0	0	160	Female	60	False	...	True
1	1	180	Male	90	False	...	True
2	2	170	Male	59	True	...	False
3	3	154	Female	49	False	...	True
4	4	159	Female	53	True	...	False
5	5	174	Female	80	False	...	True
6	6	165	Female	45	False	...	False
7	7	177	Male	70	True	...	False
8	8	184	Male	79	True	...	False
9	9	168	Male	68	True	...	False

图 3.8 代码输出

假设预测目标 y 为 Has diabetes(患糖尿病),Height (cm)(以厘米计位的身高)、Gender(性别)、Weight (kg)(以千克计量的体重)、Eats sugar products(是否吃含糖食品)等列皆被用作 X 中的特征。表格中的数据点取自个人收到检查结果一个月后,在调查问卷中的自主报告。从显示的 df 中可见,Eats sugar products 和目标 Has diabetes 有很强的相关性——除了 Patient id 为 6 的数据点,其余数据点中,此两列的取值正好相反。如果 X 中存在 Eats sugar products 这一与目标存在很强关系性的特征,则模型预测测试集的准确率将非常高。

但 Eats sugar products 这一特征与目标 Has diabetes 存在较强的因果关系,且 Has diabetes 的取值为因,Eats sugar products 的取值为果。单从这两列的列名无法得出这一因果结论,但表格的来源暗示了这一关系。每列数据的取值,来自于得知糖尿病检测结果的个体一个月后的自主报告。重点在于"得知糖尿病检测结果"和"一个月后",这意味着检测结果可能影响了个体的生活方式。而由常理推测,若个体得知患有糖尿病,则很有可能同时被告知不能再吃含糖食品,导致 Has diabetes 为 True 的数据点 Eats sugar products 特征取值皆为 False。日常生活中若不特别注意,人们或多或少会摄入含糖食品,因此,Has diabetes 为 False 的数据点 Eats sugar products 特征取值大概率为 True。图 3.8 中这两列的取值关系进一步证明了这一推测。

但在预测某个体是否患有糖尿病时,患者的生活方式并不受检测结果的影响,因此,糖尿病患者大概率会摄取含糖食品。换言之,形容该个体的数据点中,Eats sugar products 大概率为 True。而模型会因训练集中 Eats sugar products 和 Has diabetes 的关系,错误判断该个体不患有糖尿病。在这个例子中预测未来数据时,看似可以轻易取得患者"是否吃含糖食品"这一特征,但经过检查,"是否吃含糖食品"这一特征应被更准确地形容为"得知糖尿病检测结果后一个月是否吃含糖食品",而这一特征的取值无法在预知未来数据时获得,因此,Eats sugar products 在训练集、验证集和测试集中的存在被称为目标泄露。

相比目标泄露,训练集与测试集相互污染这一数据泄露方式更不容易被察觉。最极端的污染,是测试集的数据出现在训练集或验证集中。由于预测的未来数据大概率不会与训练集或验证集中数据完全相同,因此模型对测试集的预测结果无法准确预估其预测未来数据的准确率。

更微妙的污染情况是，训练集中的特征取值从某种程度上受到测试集数据的影响。在下一个 cell 中创建一个新的 DataFrame，执行：

```
#Chapter3/data_leakage_1.ipynb

import numpy as np

df2 = pd.DataFrame({'User id': list(range(10)),
                    'Preference': ['food', 'tech', 'finance', 'finance', 'tech',
                                   'tech', 'tech', 'food', 'tech', 'food'],
                    'Age': [16, 21, np.nan, 31, 43, 29, np.nan, 24, np.nan, 36],
                    'Click Ads': ['likely', 'unlikely', 'likely', 'likely',
                                  'unlikely', 'unlikely', 'likely',
                                  'unlikely', 'unlikely', 'likely']})

display(df2)
```

显示结果如图 3.9 所示。

预测目标为 Click Ads(是否单击广告)，使用 Preference(喜好)和 Age(年龄)作为用户特征进行预测。Age(年龄)这一特征存在空白值，若直接使用该特征平均值填补，在下一个 cell 中执行：

```
df2['Age'].fillna(df2['Age'].mean(), inplace=True)
display(df2)
```

显示结果如图 3.10 所示。

	User id	Preference	Age	Click Ads
0	0	food	16.0	likely
1	1	tech	21.0	unlikely
2	2	finance	NaN	likely
3	3	finance	31.0	likely
4	4	tech	43.0	unlikely
5	5	tech	29.0	unlikely
6	6	tech	NaN	likely
7	7	food	24.0	unlikely
8	8	tech	NaN	unlikely
9	9	food	36.0	likely

图 3.9　代码输出

	User id	Preference	Age	Click Ads
0	0	food	16.000000	likely
1	1	tech	21.000000	unlikely
2	2	finance	28.571429	likely
3	3	finance	31.000000	likely
4	4	tech	43.000000	unlikely
5	5	tech	29.000000	unlikely
6	6	tech	28.571429	likely
7	7	food	24.000000	unlikely
8	8	tech	28.571429	unlikely
9	9	food	36.000000	likely

图 3.10　代码输出

然后使用 sklearn 中的 train_test_split 函数以 70∶15∶15 的比例分割训练集、验证集和测试集，在下一个 cell 中执行：

```
from sklearn.model_selection import train_test_split

df_train, df_val_test = train_test_split(df2, test_size=0.3, random_state=42)
```

```
df_val, df_test = train_test_split(
                    df_val_test, test_size = 0.5, random_state = 42)
display(df_train, df_val, df_test)
```

显示结果如图 3.11 所示。

	User id	Preference	Age	Click Ads
0	0	food	16.000000	likely
7	7	food	24.000000	unlikely
2	2	finance	28.571429	likely
9	9	food	36.000000	likely
4	4	tech	43.000000	unlikely
3	3	finance	31.000000	likely
6	6	tech	28.571429	likely

	User id	Preference	Age	Click Ads
5	5	tech	29.0	unlikely

	User id	Preference	Age	Click Ads
8	8	tech	28.571429	unlikely
1	1	tech	21.000000	unlikely

图 3.11 代码输出

df2 中 User id 为 1 的数据点被分入测试集，User id 为 2 和 6 的数据点被分入训练集。而 User id 为 2 和 6 的数据点 Age 列原取值为空白值，填补时使用的平均值受到 User id 为 1 的数据点的影响。在实践中，当数据量较大时，类似以上分布的情况大概率会发生——训练集中原为空白值数据的填补值受到测试集的影响。而由于未知数据的 Age 列不影响训练集中空白值的填补，未知数据与训练集的关系或多或少异于测试集，因此，模型预测测试集的准确率无法合理预估预测未来数据的准确率。

3.3.2 发现并避免目标泄露

3.3.1 节中列举了几个存在目标泄露的例子。从这些例子可见，多数情况下，只要选择合理的特征就可以避免目标泄露。重新定义 3.3.1 节中的 df，执行：

```
#Chapter3/data_leakage_2.ipynb

import pandas as pd

df = pd.DataFrame({'Patient id': list(range(10)),
                   'Height (cm)': [160, 180, 170, 154,
                                   159, 174, 165, 177,
                                   184, 168],
                   'Gender': ['Female', 'Male', 'Male', 'Female',
                              'Female', 'Female', 'Female', 'Male',
                              'Male', 'Male'],
```

```
                    'Weight (kg)': [60, 90, 59, 49, 53,
                                    80, 45, 70, 79, 68],
                    'Eats sugar products': [False, False, True, False,
                                            True, False, False, True,
                                            True, True],
                    '...': ['...'] * 10,
                    'Has diabetes': [True, True, False, True,
                                     False, True, False, False,
                                     False, False]})
```

假设被省略的列中不含泄露目标的特征，且省略的列数较多，选择 X 时可执行：

```
# 当选择某 DataFrame 中大多数列而排出少数列时，可以使用此方法
X = df[[c for c in df.columns if c not in ['Patient id',
                                           'Eats sugar products',
                                           'Has diabetes']]]
display(X)
```

显示结果如图 3.12 所示。

	Height (cm)	Gender	Weight (kg)	...
0	160	Female	60	...
1	180	Male	90	...
2	170	Male	59	...
3	154	Female	49	...
4	159	Female	53	...
5	174	Female	80	...
6	165	Female	45	...
7	177	Male	70	...
8	184	Male	79	...
9	168	Male	68	...

图 3.12　代码输出

但在实践中，列与列之间的时间关系也许无法通过简单的列名和数据来源获得。在这种情况下，可以计算每列与目标之间的相关系数(Correlation coefficient)，找出明显异常的特征。相关系数一般以字母 r 表示，用于度量两个变量之间的线性关系。其计算公式为

$$r(X,Y)=\frac{\mathrm{cov}(X,Y)}{\sigma_X\sigma_Y} \tag{3.1}$$

其中 $\mathrm{cov}(X,Y)$代表变量 X 与 Y 的协方差，σ_X 代表变量 X 的标准差，σ_Y 代表变量 Y 的标准差。相关系数取值范围为$[-1, 1]$。当两个变量之间的 r 为 0 时，两者不相关；当两个变量之间的 r 为正数时，两者之间存在正相关关系，且 r 越大代表相关关系越强；当两个变量之间的 r 为负数时，两者之间存在负相关关系，且 r 越小代表相关关系越强。使用 NumPy 中的 corrcoef 函数计算相关系数，此函数的输出为一个相关系数矩阵，矩阵第 i 行 j 列表示第 i 个变量与第 j 个变量的相关系数。在下一个 cell 中执行：

```
# Chapter3/data_leakage_2.ipynb

import numpy as np

# 计算相关系数需保证两个变量皆为数值(Python 中的布尔值也属于数值)，
# 因此，首先将非数值列转换为数值
df_int = df.copy() # 复制 df
df_int.loc[df_int['Gender'] == 'Male', 'Gender'] = 0
df_int.loc[df_int['Gender'] == 'Female', 'Gender'] = 1
df_int['Gender'] = df_int['Gender'].astype(int)
```

```
#打印身高、体重、性别和是否摄入含糖食品 4 个特征
#以及目标本身与目标之间的相关系数
for c in ['Height (cm)', 'Weight (kg)',
          'Gender', 'Eats sugar products', 'Has diabetes']:
    print('{t}与 Has diabetes 之间的相关系数为'.format(t = c),
          np.corrcoef(df_int[c], df_int['Has diabetes'])[0][1])
```

输出如下：

```
Height (cm)与 Has diabetes 之间的相关系数为 -0.18501305048794728
Weight (kg)与 Has diabetes 之间的相关系数为 0.260856896854182
Gender 与 Has diabetes 之间的相关系数为 0.408248290463863
Eats sugar products 与 Has diabetes 之间的相关系数为 -0.816496580927726
Has diabetes 与 Has diabetes 之间的相关系数为 1.0
```

在此声明，以上数据纯属举例，因此以上特征与目标之间的相关系数不代表真实情况。由输出可见，Eats sugar products 这一特征与目标之间的相关系数约为－0.82，两者之间存在很强的负关系。目标 Has diabetes 与其本身的相关系数为 1，属于很强的正关系。将 X、y 输入模型训练前，可以打印 X 中每列与 y 之间的相关系数。若某列与 y 之间存在很强的正关系或负关系，则需特别注意该列特征是否存在目标泄露。

尽早搭建模拟预测未来数据的管道，可以有效侦查目标泄露。管道可以在模型训练完成之前搭建，与收集、创建特征同步。许多对于目标泄露的疏忽，源自错误判断历史数据中的某些特征是否可以在预测未来数据时获得。例如预测 3 日后方便面的销量时，历史数据中可以提取每个数据点对应的“昨日销量”，而在实际预测时无法获得这一特征。假设在训练模型之前，搭建一个预测未来数据的管道。管道模拟从数据库中提取形容未来数据的特征取值，并对其进行预处理。在这个问题定义中，管道将在预测当日，收集描述 3 日后这一时间点的所有训练时所用的特征取值。这也意味着，管道需要提取两日后的销量信息，因此，将无法搭建使用当前特征的预测管道。搭建管道遇到的这一障碍，可及时告诉我们训练集中存在目标泄露，需解决泄露问题后再进行模型的训练及优化。

另外，尽早设身处地预测未来数据，也可以让一些不易察觉的目标泄露浮出水面。也许在预测 3 日后方便面的销量时，我们会认为“3 日前方便面销量”这一特征不属于目标泄露，但在模拟预测的当日，假设预测时间为下午 5 点，可能会发现当日销量并未停止更新，仍有增加的可能性，因此，预测时提取的当日销量，对于 3 日后来讲，是“3 日前截止到下午 5 点的方便面销量”，而历史数据中储存并进入训练集的为“3 日前方便面总销量”。这时，根据历史数据中每日销量记录的结算时间，我们需要考虑是否可能在当日获得当日总销量这一数据。若答案为否，则“3 日前方便面总销量”在这个例子中属于目标泄露。

3.3.3 避免训练集与测试集的相互污染

相较目标泄露，训练集与测试集的相互污染更加难以被发现。以 3.1.1 节中的 df2 为例，若在数据预处理时使用过测试集的信息填补训练集中的空白值，可以视为一种微妙的污染，且没有简单的方法可以完全排查此类污染。你可能会考虑，是否可以从数据分布或训练

集与测试集的相似度入手，检测之前的预处理中是否存在类似污染。测试集的数据被用于填补训练集的空白值，理论上，测试集将较未来数据而言更接近训练集的数据分布——这也是引发顾虑的原因，若测试集更接近训练集，则模型对其预测结果将过于乐观，但是，不同数据集本身就存在数据分布上的差异，且例子中这样微妙的污染往往不会造成明显的数据分布差异，单纯对比训练集与测试集和未来数据集的数据分布差异，无法完全说明训练集与测试集之间存在相互污染。

最简单的方式，也许是在预处理数据和分离 3 集这两个步骤上多加思考，确保最后分离的测试集数据无法被训练集或验证集获取。需要确保的是，测试集能在最大程度上模拟未知数据。这也意味着，任何对测试集做的预处理，当使用到未来数据时也应当合理。举个例子，假设历史数据为以下表格，执行：

```
#Chapter3/data_leakage_3.ipynb

import pandas as pd
import random
import numpy as np

#设定随机种子，确保执行此段代码可以创建相同的 df
random.seed(42)

#Age(年龄)列取值范围为[20, 80]的一个整数
#random.randint(1, 10) = 1 的概率为 10%，因此 Age 列有 10% 概率为空白值
df = pd.DataFrame({'User id': list(range(100)),
                   'Age': [random.randint(20, 80) if random.randint(1, 10) != 1 \
                           else np.nan for _ in range(100)],
                   '...': ['...'] * 100,
                   'target': [0] * 50 + [1] * 50})
display(df)
```

显示结果如图 3.13 所示。

	User id	Age	...	target
0	0	21.0	...	0
1	1	35.0	...	0
2	2	28.0	...	0
3	3	63.0	...	0
4	4	25.0	...	0
...	...	...	...	...
95	95	40.0	...	1
96	96	21.0	...	1
97	97	79.0	...	1
98	98	76.0	...	1
99	99	35.0	...	1

100 rows × 4 columns

图 3.13　代码输出

从显示结果可见，Age(年龄)列多为有效数值。对 Series 使用.isna()和.sum()函数，计算空白值个数，在下一个 cell 中执行：

```
print(df['Age'].isna().sum())
```

输出结果如下：

```
10
```

在下一个 cell 中，以 70∶15∶15 的比例分割训练集、验证集和测试集，并分别打印训练集与测试集的空白值个数，执行：

```
#Chapter3/data_leakage_3.ipynb

from sklearn.model_selection import train_test_split

df_train, df_val_test = train_test_split(df, test_size=0.3, random_state=42)
df_val, df_test = train_test_split(
                    df_val_test, test_size=0.5, random_state=42)

print('训练集中空白值个数为: ', df_train['Age'].isna().sum())
print('测试集中空白值个数为: ', df_test['Age'].isna().sum())
```

输出结果如下：

```
训练集中空白值个数为: 7
测试集中空白值个数为: 2
```

现在考虑如何填补 Age 列中的空白值。使用全集的平均值会导致训练集与测试集相互污染，因此，填补某集时，可以考虑只使用该集数据的平均值填补该集，在下一个 cell 中执行：

```
#由于污染问题重点考虑训练集和测试集,暂时忽略验证集
df_train['Age'].fillna(df_train['Age'].mean(), inplace=True)
df_test['Age'].fillna(df_test['Age'].mean(), inplace=True)

#确认填补后的训练集和测试集中无空白值
print('训练集中空白值个数为: ', df_train['Age'].isna().sum())
print('测试集中空白值个数为: ', df_test['Age'].isna().sum())
```

输出结果如下：

```
训练集中空白值个数为: 0
测试集中空白值个数为: 0
```

如此做法可以保证训练集中的数据不受测试集的影响，避免相互污染，但这样处理的问题在于，若等待预测的未来数据个数较少或需分为每批数据点较少的批次预测，且存在空白值，那么未来数据中 Age 的平均值受偶然因素影响较大，大概率不接近历史数据的平均值。举个较为极端的例子，假设每次预测只能预测两个数据点，且其中一个数据点的 Age 列为空白值。使用与填补测试集同样的方法，该空白值将使用该次未来数据集中 Age 列有效取值的平均值填补。换言之，空白值将直接使用另一个数据点中 Age 的取值填补，这明显会存在较大的误差。

因此，填补测试集 Age 列空白值时，可以考虑使用训练集 Age 列有效取值的平均值。这样的填补不属于数据泄露，因为训练集的数值在预测未来数据时同样可以取得，且从理论角度，训练集与测试集的关系应等同于训练集与未来数据集的关系。如此填补，在测试集合理模拟了未来数据集的同时，使填补值更加接近总体均值。修改填补方式，执行：

```
df_train['Age'].fillna(df_train['Age'].mean(), inplace = True)
#填补 df_test 时,将 df_test['Age'].mean()改为 df_train['Age'].mean()
df_test['Age'].fillna(df_train['Age'].mean(), inplace = True)
```

当污染较为极端时,例如测试集的数据点直接出现在训练集中,有一定概率可以从对比模型预测训练集和测试集的准确率中,发现可能存在污染。若预测测试集的准确率过高,或过于接近训练集准确率,且已经确认不存在目标泄露,这时可以检查预处理和分割数据集的代码,是否存在训练集与测试集相互污染。

3.4 偏差与方差

训练一个机器学习模型时,目的是尽量提高模型预测测试集准确率。提高准确率,也可以称为降低预测误差。

预测误差由 3 部分组成,偏差(bias)、方差(variance)和不可约的误差(irreducible error)。不可约的误差来自于数据本质上的噪声,或是现有特征的不足。选择模型时,需要尽量降低偏差与方差,以降低最终对未来数据的预测误差。

3.4.1 定义偏差与方差

在预测某目标变量 Y 与一系列特征变量 X 的关系时,通常假设 Y 与 X 之间存在函数关系,可以用函数 f 表示:

$$Y = f^*(X) + e \tag{3.2}$$

其中,f^* 为一个准确反映 Y 与 X 之间关系的函数,而 e 是 X 经过 f^* 映射后,映像与真实取值之间的误差项。虽然 f^* 是一个理论上准确反映 Y 与 X 之间关系的函数,但若 X 中没有足够的特征,或数据中存在噪声,则 $f^*(X)$ 与 Y 之间还是会存在差异。定义 e 为 Y 与 $f^*(X)$ 之间的差异,$e = Y - f^*(X)$,从而得出式(3.2)。e 是数据本身的缺陷,无法用优化模型降低,也被称为贝叶斯误差(Bayes error)。

一个机器学习模型也是一个反映 Y 与 X 之间关系的函数,这里称为 g。训练模型时,目的是让函数 g 趋近于 f^*。换言之,我们希望在训练的过程中,让模型找到准确反映 Y 与 X 之间关系的函数。在实践中,f^* 往往过于复杂,因此,g 通常不等于 f^*。

偏差用于衡量真实关系函数 f^* 与模型学习到的函数 g 之间的差异。某数据点 x 的偏差值可由以下等式计算:

$$\text{bias}^2(x) = (E[g(x)] - f^*(x))^2 \tag{3.3}$$

$E[g(x)]$ 表示模型使用 g 函数预测数据点 x 的平均输出。根据不同的训练集,同样的模型可能会产生不同的预测结果,这些结果的平均值用 $E[g(x)]$ 表示。使用平均值,可以撇去使用不同训练集对输出的影响,因此,$E[g(x)]$ 可以表示模型本身的输出,而非限制于某训练集的输出。

方差用于衡量同一使用不同训练集后输出结果之间的差异。测试集中某数据点 x 的方差值可由以下式计算:

$$\text{var}(x) = E[(g(x) - E[g(x)])^2] \tag{3.4}$$

可以将预测目标 $f^*(x)$ 想象为靶心，将同一模型使用不同训练集训练后对 x 的预测结果想象为不同的击中点。低偏差意味着不同击中点的平均值与靶心距离较近；高偏差代表不同击中点的平均值与靶心距离较远；低方差意味着每个击中点距离不同击中点的平均值皆较近，击中点较为集中；高方差代表每个击中点距离不同击中点的平均值皆较远，击中点较为分散，如图 3.14 所示。

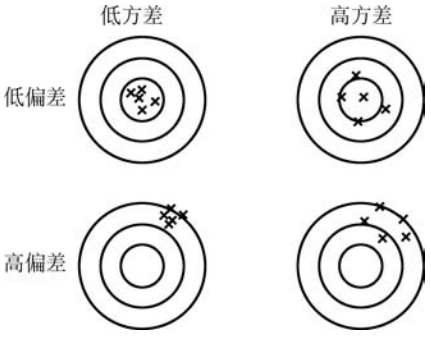


图 3.14 偏差与方差高低组合示例

由于 f^* 往往过于复杂，不同结构的模型在训练时，不同程度上简化了 f^*，从而得到 g。高偏差意味着模型过度简化了 f^*，从而无法学习到数据本质上的规律；低偏差意味着模型没有过多的简化 f^*。换言之，偏差较高的模型学习到的函数较为简单，而偏差较低的模型学习到的函数较为复杂。

从模型的角度看，低方差意味着使用不同训练集并不会过多改变模型所学函数；高方差意味着使用不同训练集将大幅度改变模型所学函数。理论上，关于同一问题分割的不同训练集应该存在同样的规律，因此，高方差往往预示模型并没有学习到训练集本质上的规律。

3.4.2 过拟合与欠拟合

当偏差较低而方差较高时，模型使用某些训练集训练后，预测训练集的准确率可能远高于同一模型使用其他训练集训练的准确率。正如 3.4.1 节所述，高方差预示着，模型并没有学习到训练集本质上的规律。结合较低的偏差，可以推测，某训练集中可能存在属于只存在于该训练集中的规律，该规律无法套用到别的训练集，也不存在于测试集或未来数据中。若该规律正好被模型学习，则模型预测该训练集的准确率，将会远高于其中规律无法被学习的训练集。

这种情况往往容易导致过拟合(overfitting)问题。过拟合，指的是模型在根据某一数量有限的训练集学习 X 与 y 之间的关系函数时，学习结果过分贴合该训练集的各个数据点，如图 3.15 所示。

在图 3.15 中，圆点表示某一训练集中的数据点，线表示模型学习到的 X 与 y 之间的关系函数。模型函数穿过训练集的每个数据点，这就意味着，模型预测训练集的误差低至 0%，但这样的准确率并不能说明模型能准确预测测试集数据。方块表示测试集数据，可见其与模型所学习的函数线之间的误差远大于 0%，如图 3.16 所示。

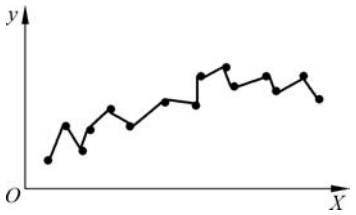


图 3.15　过拟合示例

图 3.16　测试集与模型函数存在差异

这是因为模型并没有学习到数据本质上的规律，只是“背过”了训练集中的每个数据点，或者如本节开头所述，模型只学习到了属于该训练集的规律，因此，遇到存在类似本质规律的测试集时，模型无法根据规律进行准确预测。

过拟合的后果在训练集存在噪声，或是测试集中存在稍微异于训练集的数据点时尤为明显。首先考虑第一种情况——训练集存在噪声，如图 3.17 所示。

点 a 是训练集中的一个噪声点，其取值并不符合数据本质的规律，但由于模型函数过度拟合于训练集数据，所学函数误将噪声认作有效数据，并穿过噪声点。点 b 是测试集中的一个点，其 X 与 y 取值的关系符合总体数据集的本质规律。使用此模型预测测试集中 X 特征与 a 相似的点 b，预测输出将接近点 a 的 y 取值，与点 b 的真实取值相差甚远。

接下来，考虑第二种情况——测试集中存在稍微异于训练集的数据点，如图 3.18 所示。

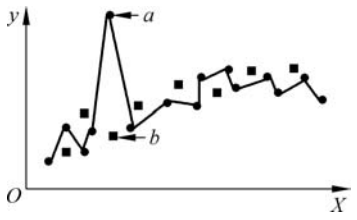


图 3.17　过拟合模型训练集中存在噪声

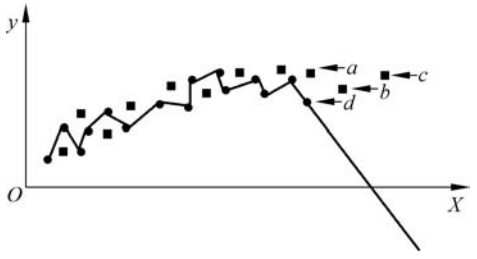


图 3.18　过拟合模型预测与训练集稍有差异的数据点

点 d 是训练集中 X 取值最右的数据点——这里使用“取值最右”而非“取值最大”形容点 d，是因为 X 中的数据点在多数情况下是一个数组，而数组之间的大小需要根据特定问题定义。点 a、b、c 是测试集中的 3 个数据点，其取值皆符合数据本质的规律，但 X 取值皆位于 d 之右。由于模型没有学习到数据本质的规律，我们可以合理假设，模型所学习函数穿过最右点 d 后，将直线下降。显然，模型预测 a、b、c 取值时，将存在非常大的误差。

与过拟合相对的另一个极端是欠拟合(underfitting)。欠拟合指的是模型学习到的函数与训练集数据拟合程度较低，往往与高偏差同时出现。模型函数过于简单，无法完全表达数据所呈现的规律，如图 3.19 所示。

在图 3.19 中，直线表示模型学习到的 X 与 y 之间的关系函数。函数完全没有捕捉到 X 取值较右数据点所表达的规律，因此，该模型在预测训练集数据时，所得误差较大。加入测试集数据，如图 3.20 所示。

简单的直线模型，同样无法准确预测测试集数据，且预测测试集的误差接近预测训练集的误差。

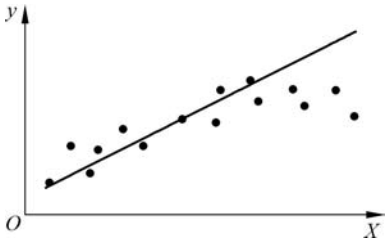


图 3.19 欠拟合示例

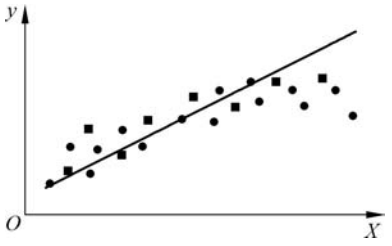
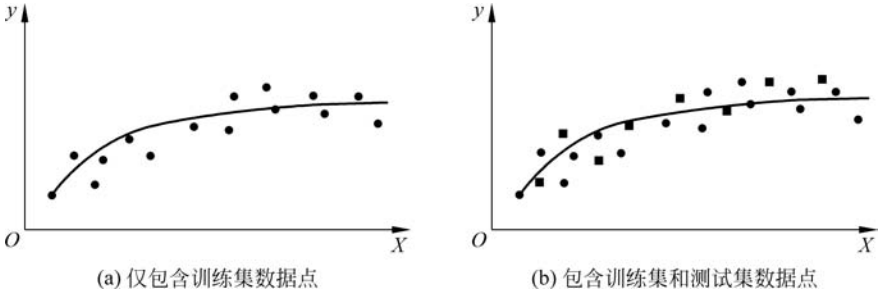


图 3.20 欠拟合示例加测试集数据

过拟合的模型相对数据本质规律过于复杂，而欠拟合模型过于简单。在实践中，这两者无法同时完全解决——模型无法在变得更加简单的同时变得更加复杂，因此，选择和优化模型的目标在于同时缓解过拟合和欠拟合现象，使模型所学函数接近数据本质规律，如图 3.21 所示。



(a) 仅包含训练集数据点　　(b) 包含训练集和测试集数据点

图 3.21 同时缓解过拟合与欠拟合

图 3.21 中的曲线表示模型所学函数。预测训练集时，模型对训练集的预测将存在少量误差，预测测试集时将存在数值相似的误差。由于模型捕捉到了数据本质的规律，我们可以合理推测，使用不同训练集获得的函数将较为相似，预测误差也均为较低值，因此，一个同时缓解了过拟合与欠拟合的模型，对应着低方差与低偏差。同时解决过拟合、欠拟合的过程，也被称作平衡偏差与方差。

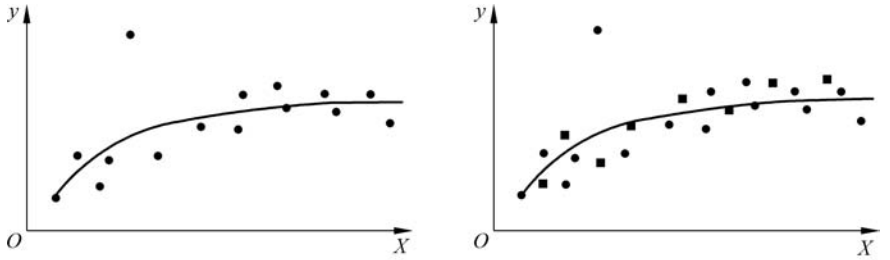
最后，回顾两个在模型过拟合时引起严重后果的情况，第一，训练集存在噪声；第二，测试集中存在稍微异于训练集的数据点。二者在一个平衡了偏差与方差的模型中，皆不构成大的问题。首先，由于大部分数据点符合本质规律，一个平衡了偏差与方差的模型，不会因为少量噪声的存在而大幅度改变模型函数，因此可以推测，当训练集中存在噪声时，模型函数仍能描述数据本质规律，如图 3.22 所示。

因此，当训练集存在噪声时，模型仍能在预测训练集和测试集时保持较高的准确率。

测试集中的数据，大概率符合总数据集本质规律。假设测试集中某些数据出现在训练集 X 取值最右点之右，如图 3.23 所示。

点 d 是训练集中 X 取值最右的数据点；点 a、b、c 是测试集中的 3 个数据点，其取值皆符合数据本质的规律。模型函数随着 X 取值向右增进，预测的 y 值首先大幅增加，而后增幅变得平缓，因此可以推测，该模型函数在 X 取值位于 d 点之右时，预测的 y 值会继续平缓

增加，因此，当测试集数据点稍微异于训练集时，模型仍能在预测测试集时保持较高的准确率。



(a) 仅包含训练集数据点　　(b) 包含训练集和测试集数据点

图 3.22　平衡偏差与方差模型遇到噪声的影响

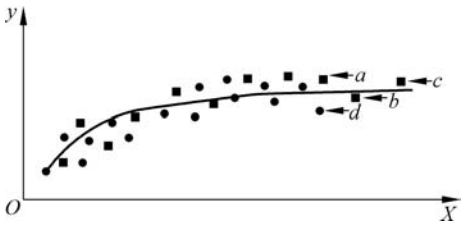


图 3.23　平衡偏差与方差模型遇到稍异于训练集的测试集数据点

3.4.3　实践中的过拟合与欠拟合

3.4.2 节中讲到，过拟合和欠拟合的模型，皆无法在预测未知数据时取得好的表现。本节将讨论如何在实践中发现模型存在过拟合或欠拟合，并有针对性地进行优化模型。

欠拟合往往容易被发现。之前提到过，我们需要尽量减少使用测试集数据的次数，因此，若模型预测训练集和验证集的准确率皆较低，且较为相似，基本可以推断模型欠拟合。甚至可以单纯使用训练后的模型预测验证集，若其准确率较低，可以推断模型欠拟合。

模型的欠拟合起因可归为两大类。第一，模型本身较数据而言过于简单。例如数据中的 X 与目标 y 存在二次多项式的关系，若模型只能建立直线关系，那么该模型结构将无法表达数据中的规律。这种情况下，可以考虑使用更加复杂的模型。第二，数据中的 X 与 y 也许不存在可以被模型解释的关系。假设收集到的数据中有 x1 和 x2 两个特征，预测目标为 y，x3 是与 y 有关但未被收集到的数据，如图 3.24 所示。

	x1	x2	x3	y
0	4	23	2	29
1	2	43	4	49
2	5	2	356	363
3	3	44	3	50
4	5	11	2	18
5	7	43	5	55
6	2	66	45	113
7	6	3	4	13

图 3.24　假想数据

y 与 x1、x2、x3 的关系为 y＝x1＋x2＋x3，但由于 x3 不属于收集到的特征之一，且 x1 和 x2 两个特征与 y 之间不存在本质上的规律，所以模型将无法从现有数据中学习并有效预测未知数据。在这种情况下，需要收集更多的特征，而非增加模型的复杂性。

区分这两类起因时，可以结合现有模型的复杂性和现有特征的信息量推断。

模型过拟合时，往往无法仅从训练集预测准确率判断。较高的训练集准确率，可能是因为模型学习到了数据的本质

规律，也可能是出现了过拟合。在这种情况下，可以根据验证集与训练集准确率或误差的差异，判断是否存在过拟合现象。举一个简单的例子，创建一个 DataFrame，其中 x1、x2、x3 为 X 中的 3 个特征，3 个特征与 y 之间的关系为 y＝x1＋x2＋x3，执行：

```
#Chapter3/overfitting_0.ipynb

import pandas as pd
import numpy as np

#这里使用 NumPy 代替 Python 自带的 random
#展示另一种创建随机数组的方法
np.random.seed(42)
df = pd.DataFrame({'x0': list(range(100)),
                   'x1': np.random.random(100) * 100,
                   'x2': np.random.random(100) * 100,
                   'x3': np.random.random(100) * 100})
df['y'] = df['x1'] + df['x2'] + df['x3']
display(df)
```

显示结果如图 3.25 所示。

	x0	x1	x2	x3	y
0	0	37.454012	3.142919	64.203165	104.800095
1	1	95.071431	63.641041	8.413996	167.126468
2	2	73.199394	31.435598	16.162871	120.797864
3	3	59.865848	50.857069	89.855419	200.578336
4	4	15.601864	90.756647	60.642906	167.001417
...	...	...	...	...	...
95	95	49.379560	34.920957	52.224326	136.524843
96	96	52.273283	72.595568	76.999355	201.868206
97	97	42.754102	89.711026	21.582103	154.047231
98	98	2.541913	88.708642	62.289048	153.539603
99	99	10.789143	77.987555	8.534746	97.311444

100 rows × 5 columns

图 3.25 代码输出

分割训练集、验证集和测试集，在下一个 cell 中执行：

```
from sklearn.model_selection import train_test_split

df_train, df_val_test = train_test_split(df, test_size=0.3, random_state=42)
df_val, df_test = train_test_split(df_val_test, test_size=0.5, random_state=42)
#使用.shape 打印 3 个 DataFrame 的形状，打印格式为(行数，列数)
print(df_train.shape, df_val.shape, df_test.shape)
```

输出如下：

```
(70, 5) (15, 5) (15, 5)
```

x0、x1、x2、x3 作为特征，预测目标为 y。创建一个过度拟合于该训练集的模型，在下一个 cell 中执行：

```
#Chapter3/overfitting_0.ipynb

class overfitting_model:
    train_X = None
    train_y = None
    joined_df = None
    fitted = False

    def fit(self, X, y):
        #训练时"背过"训练集每个数据点对应的 y 取值
        self.train_X = X
        self.train_y = y
        self.joined_df = X.join(y)
        self.fitted = True

    def predict(self, X):
        if not self.fitted:
            print('模型还未经过训练')
        #预测时，根据数据点的第一个特征(也就是 x0)的取值
        #直接输出训练集中该特征相等数据点的平均值
        #若训练集中不存在该特征取值相等的例子，预测为 0
        feature_1 = self.train_X.columns[0]
        output = []
        for i in range(len(X)):
            feature_1_val = list(X[feature_1])[i]
            if feature_1_val in list(self.train_X[feature_1]):
                output.append(self.joined_df[self.joined_df[feature_1] == \
                                             feature_1_val]['y'].mean())
            else:
                output.append(0)
        return output

#建立一个 overfitting_model 对象，称其为 overfitting_model_instance
#并使用 df_train 进行训练
overfitting_model_instance = overfitting_model()
overfitting_model_instance.fit(df_train[['x0', 'x1', 'x2', 'x3']],
                               df_train['y'])
```

这个模型“背过”了训练集中所有数据点的 y 取值。其中，模型使用训练集中与预测数据点 x0 取值相同点的平均 y 值之一行为，相当于提取仅属于该训练集，而无法泛论到其他数据集的规律。使用模型预测训练集。由于预测目标为连续变量，使用简单的均方误差(mean squared error，MSE)衡量预测结果与真实 y 取值的误差。均方误差，计算预测值与真实取值每一项差的平方的平均值，其计算公式为

$$\mathrm{MSE}=\frac{1}{n}\sum_{1}^{n}(y_i-y_i')^2 \tag{3.5}$$

其中，n 为数据点个数，y_i 表示第 i 个数据点的真实取值，y_i' 表示模型对第 i 个数据点的预测值。在下一个 cell 中，使用 sklearn 中的 mean_squared_error 函数，评估 overfitting_model_instance 对训练集的预测误差，执行：

```
from sklearn.metrics import mean_squared_error
prediction_train =   overfitting_model_instance.predict(df_train[['x0', 'x1',
                                                                   'x2', 'x3']])
print('模型预测训练集所得 MSE 为', mean_squared_error(df_train['y'], prediction_train))
```

输出如下：

```
模型预测训练集所得 MSE 为 476.4947309235288
```

在下一个 cell 中，评估 overfitting_model_instance 对验证集的预测误差，执行：

```
prediction_val =   overfitting_model_instance.predict(df_val[['x0', 'x1',
                                                               'x2', 'x3']])
print('模型预测验证集所得 MSE 为', mean_squared_error(df_val['y'], prediction_val))
```

输出如下：

```
模型预测验证集所得 MSE 为 11626.439792116962
```

在实践中，我们往往不能如例子中这样，得知模型的具体运行原理，但巨大的误差差异，足以说明模型出现了过拟合。附加一条说明：这个例子中，模型同时存在过拟合和欠拟合的现象。模型预测训练集的误差较大，但预测验证集所得误差更大。这样的情况对应高偏差和高方差。

在实践中，偶然的情况下，模型的过拟合也许无法用以上方法发现。若验证集相对训练集的数据量比例过小，有一定概率，则验证集中大部分数据点与训练集的某些数据类似。分割出一个满足这样条件的验证集，称其为 df_val_small，在下一个 cell 中执行：

```
#Chapter3/overfitting_0.ipynb

df_val_small = df_val[df_val['x0'].isin([14, 72, 87])]
prediction_val_small =   overfitting_model_instance.predict(df_val_small[['x0',
                                                                           'x1',
                                                                           'x2',
                                                                           'x3']])
print('模型预测小验证集所得 MSE 为', mean_squared_error(df_val_small['y'],
                                                        prediction_val_small))
print('模型预测小验证集结果为\n', prediction_val_small)
```

输入如下：

```
模型预测小验证集所得 MSE 为 405.7961962539237
模型预测小验证集结果为
 [138.97113429657134, 169.11995751410032, 102.54169051412093]
```

仅包含 3 个数据点的验证集 MSE 与预测训练集取得的 MSE 相似，但这明显不是因为过拟合问题被解决了，而是因为验证集过小，导致其包含的数据恰好与训练集某些数据类似——prediction_val_small 中不存在为 0 的预测，由此可知，df_val_small 中每个点的 x0 取值都存在于训练集中。一个过拟合的模型，在预测非常接近训练集的数据时，误差较低，但从原验证集 df_val 的预测误差看来，这样的数据点并不多见，因此，只要使用常规的验证集与训练集比例，或是 5.3 节中将要讲解的交叉验证——使用多个训练集和验证集组合，便可以避免这个问题。

3.5 小结

基础建模的步骤为第 1 步，根据问题定义和收集到的数据选择 X、y；第 2 步，分割训练集、验证集和测试集；第 3 步，使用训练集的 X、y 训练模型；第 4 步，使用验证集预估模型预测未来数据的准确率。最后，本章讲解了两个在基础建模和优化模型阶段都需要避免的问题——数据泄露和模型的过拟合或欠拟合。