第 18 章 数据科学的 Python 语言

本章着重介绍 Python 语言在数据科学方面的应用，包括文件读写、数据输入、数据清理(缺失值、重复观测值)、数据合并，以及 Sci-Kit Learn 模块的管道线(pipeline)。

18.1 何为数据科学

数据科学(data science)是从数据中发现模式、规律，得到洞见(insight)，从而为业界创造商业价值的一套科学方法。

数据科学的主旨是，从数据中挖掘有用信息，产生新知识(knowledge generation)，并为业界创造商业价值。

通俗地说，数据科学就是“数据科学家”(data scientist)所做的那些事情，即所谓的“数据科学项目”(data science project)。Wickham and Grolemund(2016)对数据科学项目的流程作了总结，参见图 18.1。

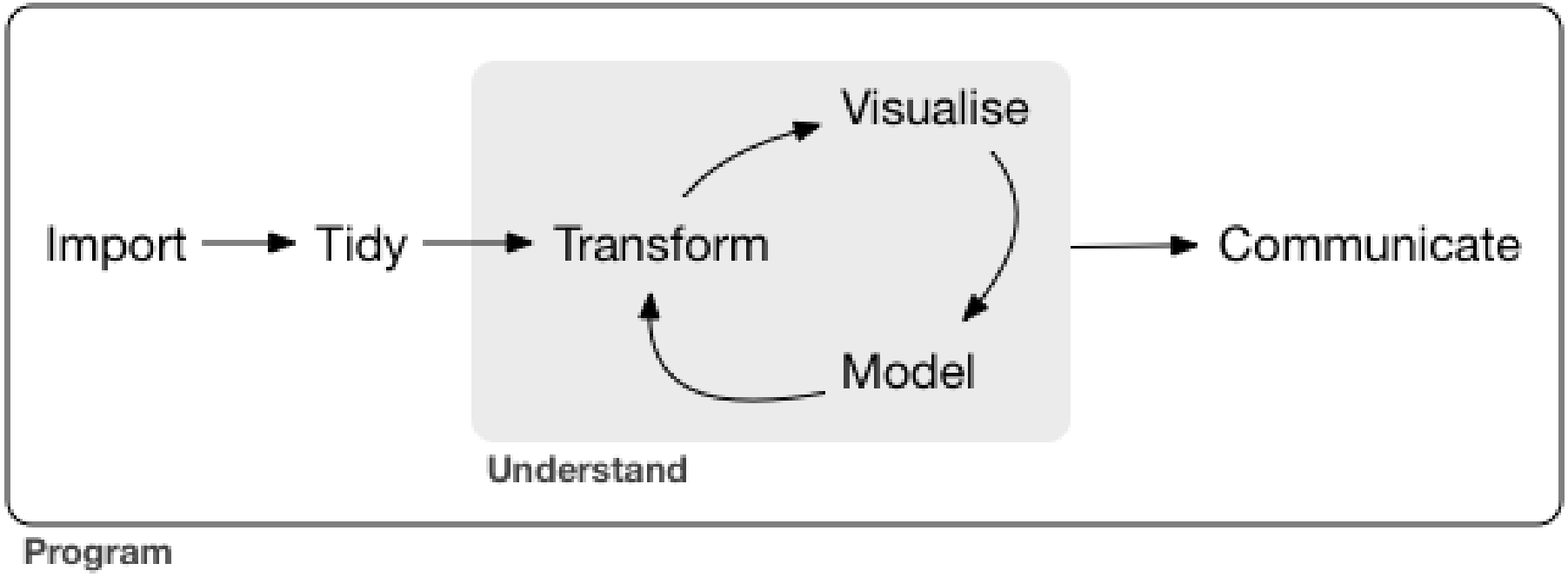

图 18.1 数据科学项目的流程

(1) 一个典型的数据科学项目始于“输入”(import)数据到软件，比如Python。如果没有数据，则巧妇难为无米之炊，无法进行数据科学项目。

(2) 将数据导入软件后，一般需要先进行“数据清理”(data cleaning)，也称“数据清洗”，使其变得“整洁”(tidy)。需要将数据整理为内部自洽的“一致结构”(consistent structure)，通常表现为“矩形数据”(rectangular data)，即以行表示样例(观测值)，而以列表示变量(特征)。

“整洁数据”(tidy data)一般有以下三个标准：
每个变量都占一列；
每个观测值都占一行；
每个观测值的每个变量取值都占据二维表的一格(cell)。

(3) 完成数据清理之后，一般须进一步对数据进行变换(transform)，这包括将变量标准化(standardization)，根据原有变量定义新的特征变量，选择原样本的某些子集，合并数据集，以及计算统计指标等。这些变换也称为“预处理”(pre-processing)。

(4) 在数据清理与预处理之后，即可开始从数据中“创造知识”(knowledge generation)的过程。此过程包括两大方法，即“可视化”(visualization)与“建模”(modeling)。

可视化能让你看到数据中意想不到的现象，或提出新问题。

建模则通过数学模型与算法，找到数据之间的内在联系与规律。无论可视化，还是建模，都可加深我们对于数据的理解(understand)。

(5) 数据科学项目的最后一步为“交流”(communicate)，即向有关领导、部门或用户汇报你的研究结果与发现。这是数据科学项目的“临门一脚”；若缺乏有效交流，则无法实现数据的商业价值。

围绕以上数据科学项目全过程的则是“编程”(programming)。

从输入数据、数据清理、数据变换，到可视化、建模，乃至交流，都离不开编程。

一个优秀的数据科学家一般需要具备三方面的知识与技能，即数理知识(例如机器学习)、编程能力(例如 Python 语言编程)与专业知识(熟悉所在行业业务)。

为便于执行以上数据科学项目流程，RStudio 首席科学家 Hadley Wickham 推出超级 R 包 `tidyverse`,包含一系列内在自洽的 R 包(a set of packages that work in harmony)，助力数据科学项目的全过程。

2019 年，Hadley Wickham 因此获得国际统计学会的 COPSS 奖(统计学最高奖)。

在 Python 中，可通过安装 Dplython 模块(`pip install dplython`)，来实现 R 包 `tidyverse` 的主要子包 `dplyr` 的类似功能。

18.2 读写文件

第 2 章介绍了 Python 的一些主要内建对象类型(major built-in object types)，包括数字、字符串、布尔型、列表、元组、字典、集合等。

Python 还有一个重要的内建对象类型，即“文件对象”(`file object`)，用于读写文件。

这些文件通常是一般的文本文件，不一定是像 CSV 文件那样的二维表数据。

在 Python 中，可使用内建的 open()函数创建一个文件对象，作为指向电脑硬盘某个文件的连接。

创建此文件对象后，可使用该文件对象的一系列方法(methods)，与电脑硬盘的外部文件进行数据交换，即读写字符串。

首先，导入本章前 6 节所需模块：

```
In [1]: import numpy as np
   ...: import pandas as pd
```

其次，作为示例，在当前工作路径，建立一个名为“myfile.txt”的文本文件：

```
In [2]: myfile = open('myfile.txt', 'w')
```

其中，参数“'w'”表示以“write”(覆盖写)的方式(mode)打开或创建文本文件“myfile.txt”。

如果文件 myfile.txt 原已存在，则覆盖其内容；若不存在，则创建一个新文件 myfile.txt。

输出结果“myfile”即为相应的“文件对象”，本质上是一个文本输入与输出的接口(text input-ouput wrapper)。

查看 myfile 的对象类型：

```
In [3]: type(myfile)
Out[3]: _io.TextIOWrapper
```

创建文件对象 myfile 之后，可使用 write()方法，向其中写入字符串，比如

```
In [4]: myfile.write('Stay Hungry.\n')
Out[4]: 13
```

其中，向文件对象 myfile 写入了“Stay Hungry”，一个句号“.”，及换行符“\n”(否则不会自动换行)。输出结果“13”为输入的字符个数。

再写入另一行字符：

```
In [5]: myfile.write('Stay Foolish.\n')
Out[5]: 14
```

结果显示，这次所输入的字符个数为 14。

打开或创建文件对象之后，若完成文件读写任务，可用 close()方法关闭此文件对象：

```
In [6]: myfile.close()
```

关闭文件对象意味着，关闭指向此文件的文本输入输出接口。

在关闭文件之前，前面输入的两行字符依然只在内存缓冲(buffer)，仍未写入硬盘文件。

此时硬盘文件处于占用状态，其他电脑进程无法对此文件进行操作。

在关闭文件时，文件对象所缓冲的文本内容被写入硬盘文件，然后关闭此输入输出接口，并将此硬盘文件释放给系统(其他电脑进程可对此文件进行操作)。

如果想将缓冲内容写入硬盘文件(`myfile.txt`)，但并不关闭文件对象(`myfile`)，可使用 `flush()`方法，比如 `myfile.flush()`。

在文件对象被回收时(包括退出 Python 程序)，Python 也会自动关闭文件(auto-close on collection)。

但使用 `close()`方法手工关闭文件依然是个好习惯，尤其对于较复杂的程序。

为避免忘记关闭文件，可使用 Python 的 `with` 语句，在完成文件读写任务后会自动关闭文件：

```
In [7]: with open('myfile.txt', 'w') as myfile:
   ...:     myfile.write('Stay Hungry.\n')
   ...:     myfile.write('Stay Foolish.\n')
```

其中，首行命令以“覆盖写”(参数`'w'`)的方式创建文件“`myfile.txt`”以及相应的文件对象“`myfile`”。

这行命令在功能上相当于上文的命令“`myfile = open('myfile.txt', 'w')`”。

二者的主要区别在于，在执行完 `with` 语句下面缩进(indented)的两行命令后，将自动关闭此文件。

如果想以“只读”(read)的方式打开此文件，可输入命令：

```
In [8]: myfile = open('myfile.txt', 'r')
```

其中，参数“`'r'`”表示**只读**(read)，即只能从文件读取内容，而无法写入内容。

参数“`'r'`”为默认参数，故可省略。

使用文件对象的 readline()方法，可逐行读取文件：

```
In [9]: myfile.readline()
Out[9]: 'Stay Hungry.\n'
```

再次使用 readline()方法，可读取下一行：

```
In [10]: myfile.readline()
Out[10]: 'Stay Foolish.\n'
```

尝试再用一次 readline()方法：

```
In [11]: myfile.readline()
Out[11]: ''
```

结果显示，输出了一个空的字符串(an empty string)，表示已至文件末尾。

若想一次性读取文件中的所有各行，并以列表形式展示，可使用 readlines()方法：

```
In [12]: myfile = open('myfile.txt')
    ...: myfile.readlines()
Out[12]: ['Stay Hungry.\n', 'Stay Foolish.\n']
```

若想一次性读取文件中的所有字符串，可使用 read()方法：

```
In [13]: open('myfile.txt').read()
Out[13]: 'Stay Hungry.\nStay Foolish.\n'
```

使用 print()函数，可得到更友好的显示效果：

```
In [14]: print(open('myfile.txt').read())
Stay Hungry.
Stay Foolish.
```

文件对象也是一种“可迭代对象”(iterable)，它以“行”为单位，从一行迭代至下一行，称为**文件迭代器**(file iterator)。

比如，使用如下 for 循环，可扫描并打印文件中的每一行：

```
In [15]: for line in open('myfile.txt'):
    ...:     print(line)
Stay Hungry.

Stay Foolish.
```

其中，由于 print()函数的默认参数 “end='\n'”，故自动在末尾加上一个新行(new line)，导致出现多余空行。

一种解决方法为，使用参数“end=''”：

```
In [16]: for line in open('myfile.txt'):
    ...:     print(line, end='')
Stay Hungry.
Stay Foolish.
```

其中，参数“end=''”表示以空的字符串作为结尾。

另一解决方法为，使用字符串的 rstrip()方法，去掉字符串右边的“空白”(whitespace)：

```
In [17]: for line in open('myfile.txt'):
    ...:     print(line.rstrip())
Stay Hungry.
Stay Foolish.
```

如果想在已有文件中追加写入内容(但不覆盖已有内容)，可使用“追加写”(append)的方法打开文件，比如

```
In [18]: with open('myfile.txt', 'a') as myfile:
    ...:     myfile.write('-- Steve Jobs\n')
```

其中，标题行的参数“'a'”表示追加写(append)。

如果 myfile.txt 不存在，则创建此文件；

反之，若 myfile.txt 已存在，则保留原有内容，然后追加写入新内容。

使用 read()方法读取 myfile.txt 所有内容，然后打印可得：

```
In [19]: print(open('myfile.txt').read())
Stay Hungry.
Stay Foolish.
-- Steve Jobs
```

作为另一例子，创建一个包含圆周率若干位数的文本文件：

```
In [20]: with open('pi_digits.txt', 'w') as file_object:
    ...:     file_object.write('3.1415926535\n')
    ...:     file_object.write('8979323846\n')
    ...:     file_object.write('2643383279\n')
```

其中，通过 with 语句，创建了一个文本文件“pi_digits.txt”，以及相应的文件对象“file_object”，然后写入 3 行字符串。

使用 read()方法读取全部字符串，并打印：

```
In [21]: print(open('pi_digits.txt').read())
3.1415926535
8979323846
2643383279
```

结果显示，圆周率的这些位数被分割在不同的三行。

若要将这三行数字连接在一起，可使用如下 for 循环：

```
In [22]: pi_string = ''
    ...: for line in open('pi_digits.txt'):
    ...:     pi_string += line.strip()
```

其中，首行命令初始化 pi_string 为一个空的字符串。

然后，通过“文件迭代器”(file iterator)，每次读取一行，并使用 strip() 方法去掉其前后的空白(whitespace)，再使用“增强型赋值”(augmented assignment) “+=”将该行字符串添加至 pi_string。

打印所得结果 pi_string：

```
In [23]: print(pi_string)
3.14159265358979323846264338327 9
```

展示字符串 pi_string 的长度：

```
In [24]: len(pi_string)
Out[24]: 32
```

结果显示，字符串 pi_string 包含圆周率的 32 个位数。

在使用 open()函数创建文件对象时，除了默认的“只读”('r')方式，以及“覆盖写”('w')与“追加写”('a')方式，另一方式为**创建写**('x')。

“创建写”('x')方式意味着，如果文件不存在，则创建此文件；反之，若文件已存在，则会报错(FileExistsError)。

若担心覆盖已有文件，可使用“创建写”('x')的方式。

18.3 输入数据

对于数据科学而言，更常见的文件格式为二维表形式的数据文件，比如 CSV 文件。

第 2 章已介绍过读写 CSV 文件的基本知识。

但现实数据可能比较杂乱(messy)，故本节进一步介绍 `read_csv()`函数，以及更一般的 `read_table()`函数。

本节使用了一系列来自 McKinney(2017)的 `csv` 文件与 `txt` 文件，可从本书网站(www.econometrics-stata.com)下载，并放于当前工作路径。

使用 pandas 模块的 read_csv()函数，载入第 1 个示例 ex1.csv 文件：

```
In [25]: df = pd.read_csv('ex1.csv')
    ...: df
Out[25]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

结果显示，这是一个很简单的示例数据，仅包含 3 个观测值与 5 个变量。read_csv()函数用于打开以逗号分隔的文本文件。

这可等价地通过 read_table()函数来实现：

```
In [26]: df = pd.read_table('ex1.csv', sep=',')
    ...: df
Out[26]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

其中，参数“sep=','”表示以逗号“,”作为分隔符(delimiter)。

read_table()函数适用于更一般的文本文件。

例如，若文本文件以分号“;”作为分隔符，则在调用 read_table()函数时，可输入参数“sep=';'”；以此类推。

在使用 read_csv()函数打开 CSV 文件时，默认“header=0”，这意味着将文件的第 0 行作为变量名。

若第 0 行并非变量名，可输入参数“header=None”。

以第 2 个示例文件 ex2.csv 为例：

```
In [27]: pd.read_csv('ex2.csv', header=None)
Out[27]:
   0   1   2   3      4
0  1   2   3   4  hello
1  5   6   7   8  world
2  9  10  11  12    foo
```

其中，参数“header=None”表示文件中并无变量名，故从第 0 行开始即为数据。

结果显示，自动加上了默认的变量名“0，1，2，3，4”。

在读取 CSV 文件时，还可以列表形式指定变量名，比如

```
In [28]: pd.read_csv('ex2.csv', names = ['a', 'b', 'c',
'd',
                                         'message'])
Out[28]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

其中，参数“names = ['a', 'b', 'c', 'd', 'message']”指定变量名为“a，b，c，d，message”。

第 3 个示例文件 ex3.txt 为文本文件。

先用 open()函数打开，并以 list()函数作为列表展示：

```
In [29]: list(open('ex3.txt'))
Out[29]:
['            A         B         C\n',
 'aaa -0.264438 -1.026059 -0.619500\n',
 'bbb  0.927272  0.302904 -0.032399\n',
 'ccc -0.264273 -0.386314 -0.217601\n',
 'ddd -0.871858 -0.348382  1.100491\n']
```

结果显示，此文本文件并未使用“固定的分隔符”(a fixed delimiter)，而是以长度不同的空白(a variable amount of whitespace)作为分隔。

对于此文本文件，在调用 `read_table()`函数时，可用正则表达式(regular expression)“`\s+`”来表示分隔符：

```
In [30]: df = pd.read_table('ex3.txt', sep='\s+')
    ...: df
Out[30]:
            A         B         C
aaa -0.264438 -1.026059 -0.619500
bbb  0.927272  0.302904 -0.032399
ccc -0.264273 -0.386314 -0.217601
ddd -0.871858 -0.348382  1.100491
```

其中，由于变量名只有 3 栏(A，B 与 C)，而数据有 4 列，故默认将第 1 列数据作为该数据框的“索引”(index)。

在第 4 个示例文件 ex4.csv 中，包括几行额外的注释。

先用 open()函数打开察看：

```
In [31]: list(open('ex4.csv'))
Out[31]:
['# hey!\n',
 'a,b,c,d,message\n',
 '# just wanted to make things more difficult for you\n',
 '# who reads CSV files with computers, anyway?\n',
```

```
 '1,2,3,4,hello\n',
 '5,6,7,8,world\n',
 '9,10,11,12,foo']
```

其中，第 0 行、第 2 行与第 3 行都是注释，并非变量名或数据。

为了在读取数据时跳过这几行，可输入以下命令：

```
In [32]: pd.read_csv('ex4.csv', skiprows=[0, 2, 3])
Out[32]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

其中，参数“skiprows=[0, 2, 3]”表示，跳过第 0 行、第 2 行与第 3 行。现实数据常包含缺失值。

pandas 模块默认以“NA”或“NULL”(用空的字符串表示)作为识别缺失值的**标识**(sentinel)。

以第 5 个示例文件 ex5.csv 为例，先用 open()函数打开考察：

```
In [33]: list(open('ex5.csv'))
Out[33]:
['something,a,b,c,d,message\n',
 'one,1,2,3,4,NA\n',
 'two,5,6,,8,world\n',
 'three,9,10,11,12,foo']
```

结果显示，若不考虑包含变量名的首行，此文件数据部分第 0 行的最后 1 个观测值为“NA”，而第 1 行有一个空的字符串(在数字 6 与 8 之间)。

使用 read_csv()函数打开此 CSV 文件：

```
In [34]: result = pd.read_csv('ex5.csv')
    ...: result
Out[34]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       two  5   6   NaN   8   world
2     three  9  10  11.0  12     foo
```

结果显示，第 0 行与第 1 行各有一个缺失值，显示为“NaN”。

可使用 isna()或 isnull()函数找出数据框的缺失值：

```
In [35]: pd.isna(result)
Out[35]:
   something     a     b     c     d  message
0      False  False  False  False  False     True
1      False  False  False   True  False    False
2      False  False  False  False  False    False
```

在现实数据中，可能以不同的标识(sentinel)表示缺失值。通过参数“na_values”，可指定额外的缺失值标识，比如

```
In [36]: pd.read_csv('ex5.csv', na_values=['foo'])
Out[36]:
  something  a   b    c   d message
0       one  1   2  3.0   4     NaN
1       two  5   6  NaN   8   world
2     three  9  10  11.0  12     NaN
```

其中，参数“na_values=['foo']”指定“foo”也是缺失值的标识。结果显示，数据框中现有 3 个缺失值。

有时，数据中的不同变量可能以不同的标识表示缺失值。此时，可先将不同变量的缺失值标识，定义为一个字典，然后传给参数“na_values”，比如

```
In [37]: sentinels = {'something': 'two', 'message':
                      'foo'}
   ...: pd.read_csv('ex5.csv', na_values=sentinels)
Out[37]:
  something  a   b   c    d  message
0       one  1   2   3.0   4      NaN
1       NaN  5   6   NaN   8    world
2     three  9  10  11.0  12      NaN
```

其中，第 1 行命令所定义的字典 sentinels，指定“two”为变量“something”的缺失值标识，而“foo”为变量“message”的缺失值标识。

第 6 个示例文件包含较多数据，使用 read_csv()函数载入此数据：

```
In [38]: pd.read_csv('ex6.csv')
Out[38]:
          one       two     three      four key
0    0.467976 -0.038649 -0.295344 -1.824726   L
1   -0.358893  1.404453  0.704965 -0.200638   B
2   -0.501840  0.659254 -0.421691 -0.057688   G
3    0.204886  1.074134  1.388361 -0.982404   R
```

```
4     0.354628 -0.133116  0.283763 -0.837063   Q
...        ...       ...       ...       ...  ..
9995  2.311896 -0.417070 -1.409599 -0.515821   L
9996 -0.479893 -0.650419  0.745152 -0.646038   E
9997  0.523331  0.787112  0.486066  1.093156   K
9998 -0.362559  0.598894 -1.843201  0.887292   G
9999 -0.096376 -1.012999 -0.657431 -0.573315   0

[10000 rows x 5 columns]
```

结果显示，该数据框共有 1 万行与 5 列数据。对于很大的数据集，读入全部数据可能较费时。若只想载入前 5 行数据，可输入以下命令：

```
In [39]: pd.read_csv('ex6.csv', nrows=5)
Out[39]:
        one       two     three      four key
0  0.467976 -0.038649 -0.295344 -1.824726   L
1 -0.358893  1.404453  0.704965 -0.200638   B
2 -0.501840  0.659254 -0.421691 -0.057688   G
3  0.204886  1.074134  1.388361 -0.982404   R
4  0.354628 -0.133116  0.283763 -0.837063   Q
```

其中，参数“nrows=5”表示仅载入前 5 行观测值。

对于大数据，还可将全样本分为若干“数据块”(chunks)，然后分块载入数据，比如：

```
In [40]: chunks = pd.read_csv('ex6.csv',
chunksize=2000)
```

其中，参数“chunksize=2000”表示数据块的容量(chunk size)为 2000 个观测值，即每次载入 2000 个观测值。

使用 type()函数，查看所得结果 chunks 的对象类型：

```
In [41]: type(chunks)
Out[41]: pandas.io.parsers.TextFileReader
```

结果显示，这是 pandas 的 io.parsers 子模块的“文本文件阅读器”(TextFileReader)对象。这也是一种可迭代对象(iterable)。

假设我们想根据这些载入的数据块，计算全样本中变量“one”的样本均值，最小值与最大值，可针对数据块使用 for 循环来实现：

```
In [42]: means = []; mins = []; maxs = []
    ...: for chunk in chunks:
    ...:     means.append(chunk.one.mean())
    ...:     mins.append(chunk.one.min())
    ...:     maxs.append(chunk.one.max())
```

其中，第 1 行命令初始化 means，mins 与 maxs 为空的列表。

在 for 循环中，针对 TextFileReader 对象 chunks 所包含的每个数据块 chunk，依次计算变量“one”的样本均值、最小值与最大值，并使用 append()方法添加至相应的 means，mins 与 maxs 列表中。

展示每个数据块的样本均值：

```
In [43]: means
Out[43]:
[0.04575016080420039,
 0.04575016080420039,
 0.04575016080420039,
 0.04575016080420039,
 0.04575016080420039]
```

结果显示，每个数据块的样本均值都相同。考察每个数据块的最小值：

```
In [44]: mins
Out[44]:
[-3.72686372556,
 -3.72686372556,
 -3.72686372556,
 -3.72686372556,
 -3.72686372556]
```

结果表明，每个数据块的最小值都一样。

继续考察每个数据块的最大值：

```
In [45]: maxs
Out[45]: [2.8338914628 2, 2.8338914628 2, 2.8338914628 2,
2.8338914628 2, 2.8338914628 2]
```

结果显示，每个数据块的最大值也一样。这是因为，ex6.csv 文件事实上包含重复的数据。统一显示整个样本的均值、最小值与最大值：

```
In [46]: np.mean(means), np.max(mins), np.max(maxs)
Out[46]: (0.0457501608042 0039, -3.7268637255 6,
2.8338914628 2)
```

18.4 缺失值

在第 2 章已初步介绍 Python 与 pandas 中的缺失值，可用 Python 原生的 None 或 Numpy 的 nan 来表示，并显示为 IEEE 的特别浮点值 NaN(Not a Number)。

进一步，可使用 isna()或 isnull()方法发现数据框的缺失值。

如果数据中有缺失值，应该如何处理？

在实践中，主要有三种方法。

第一种方法忽略缺失值的存在，不作任何处理，因为有些算法可以接受缺失值。

第二种方法去掉缺失值，即删除包含缺失值的行或列。

第三种方法则对缺失值进行填补或估算。

首先介绍如何去掉缺失值。在 R 语言中，以“NA”(Not Available)表示缺失值。若想以 NA 表示 Numpy 的 nan，可输入命令：

```
In [47]: from numpy import nan as NA
```

然后，即可用“NA”表示缺失值，例如

```
In [48]: df = pd.DataFrame([[1, NA, 3, 5], [7, 2, NA,
                NA], [NA, NA, NA, NA], [9, 4, 6, 3]],
                columns = ['x1', 'x2', 'x3', 'x4'])
    ...: df
Out[48]:
    x1   x2   x3   x4
0  1.0  NaN  3.0  5.0
1  7.0  2.0  NaN  NaN
2  NaN  NaN  NaN  NaN
3  9.0  4.0  6.0  3.0
```

如想去掉带有缺失值的任何行，可使用 dropna()方法：

```
In [49]: df.dropna()
Out[49]:
    x1   x2   x3   x4
3  9.0  4.0  6.0  3.0
```

其中，由于第 0-2 行均有缺失值，故最后仅剩下第 3 行。

若只想去掉全部为缺失值的行，可加上参数“how='all'”：

```
In [50]: df.dropna(how='all')
Out[50]:
    x1   x2   x3   x4
0  1.0  NaN  3.0  5.0
1  7.0  2.0  NaN  NaN
3  9.0  4.0  6.0  3.0
```

其中，由于第 2 行全为缺失值，故被去掉。

默认参数“how='any'”，即只要某行有任何缺失值，就被删去。

若想去掉带有缺失值的列，可用参数“axis=1”来实现；默认参数“axis=0”，即删除行(在实践中更为常见)。

为演示目的，先在数据框中加上全为缺失值的变量 x4：

```
In [51]: df['x4'] = NA
    ...: df
Out[51]:
    x1   x2   x3  x4
0  1.0  NaN  3.0 NaN
1  7.0  2.0  NaN NaN
2  NaN  NaN  NaN NaN
3  9.0  4.0  6.0 NaN
```

若要删去全为缺失的变量 x4，可输入命令：

```
In [52]: df.dropna(axis=1, how='all')
Out[52]:
    x1   x2   x3
0  1.0  NaN  3.0
1  7.0  2.0  NaN
2  NaN  NaN  NaN
3  9.0  4.0  6.0
```

在删除数据时，dropna()方法还允许指定一个“门槛”(threshold)，比如缺失 2 个以上观测值才删除：

```
In [53]: df.dropna(thresh=2)
Out[53]:
    x1   x2   x3  x4
0  1.0  NaN  3.0 NaN
1  7.0  2.0  NaN NaN
3  9.0  4.0  6.0 NaN
```

结果显示，第 2 行由于有 4 个缺失值(多于 2 个)而被删去，而其余行的缺失值均不超过 2 个，故被保留。

在去掉缺失值时，难免删除有用的信息，导致样本容量或变量数目减少。为此，处理缺失值的另一方法为填补数据框中这些缺失的“空洞”，这可使用 fillna()方法来实现。

若想将所有缺失值都变为 0，可输入命令：

```
In [54]: df.fillna(0)
Out[54]:
    x1   x2   x3   x4
0  1.0  0.0  3.0  0.0
1  7.0  2.0  0.0  0.0
2  0.0  0.0  0.0  0.0
3  9.0  4.0  6.0  0.0
```

若想将 x4 的缺失值设为 0，而将其余变量的缺失值都设为 1，可输入一个字典作为 fillna()方法的参数：

```
In [55]: df.fillna({'x1': 1, 'x2': 1, 'x3': 1, 'x4': 0})
Out[55]:
    x1   x2   x3   x4
0  1.0  1.0  3.0  0.0
1  7.0  2.0  1.0  0.0
2  1.0  1.0  1.0  0.0
3  9.0  4.0  6.0  0.0
```

若想“就地”(in-place)改变数据框，可输入参数“inplace=True”：

```
In [56]: df.fillna({'x1': 1, 'x2': 1, 'x3': 1, 'x4': 0},
inplace=True)
    ...: df
Out[56]:
    x1   x2   x3   x4
0  1.0  1.0  3.0  0.0
1  7.0  2.0  1.0  0.0
2  1.0  1.0  1.0  0.0
3  9.0  4.0  6.0  0.0
```

其中，参数“inplace=True”表示就地改变，默认为“inplace=False”(仅提供 copy 或 view，而不改变原数据框)。

有时我们并不想用某个固定数值来填补缺失值，而希望用更智能的**插值**(interpolation)来插补缺失值。

作为例子，随机生成一个6×3的数据框：

```
In [57]: np.random.seed(1)
    ...: matrix = np.random.randn(6, 3)
    ...: df = pd.DataFrame(matrix)
    ...: df
Out[57]:
          0         1         2
0  1.624345 -0.611756 -0.528172
1 -1.072969  0.865408 -2.301539
```

```
2  1.744812 -0.761207  0.319039
3 -0.249370  1.462108 -2.060141
4 -0.322417 -0.384054  1.133769
5 -1.099891 -0.172428 -0.877858
```

其次，将其第 1 个变量的 2-4 个观测值，以及第 2 个变量的第 4 个观测值设为缺失值：

```
In [58]: df.iloc[2:5, 1] = NA
    ...: df.iloc[4, 2] = NA
    ...: df
Out[58]:
          0         1         2
0  1.624345 -0.611756 -0.528172
1 -1.072969  0.865408 -2.301539
2  1.744812       NaN  0.319039
3 -0.249370       NaN -2.060141
4 -0.322417       NaN       NaN
5 -1.099891 -0.172428 -0.877858
```

若想使用变量的“滞后值”(即更早的变量取值)进行插值，可输入命令：

```
In [59]: df.fillna(method='ffill')
Out[59]:
          0         1         2
0  1.624345 -0.611756 -0.528172
1 -1.072969  0.865408 -2.301539
2  1.744812  0.865408  0.319039
3 -0.249370  0.865408 -2.060141
4 -0.322417  0.865408 -2.060141
5 -1.099891 -0.172428 -0.877858
```

其中，参数“`method='ffill'`”表示“forward fill”，即使用滞后值向前填充缺失值。

若用很久远的数据进行插值，可能并不合理。为此，可限制插值的范围：

```
In [60]: df.fillna(method='ffill', limit=2)
Out[60]:
          0         1         2
0  1.624345 -0.611756 -0.528172
1 -1.072969  0.865408 -2.301539
2  1.744812  0.865408  0.319039
3 -0.249370  0.865408 -2.060141
4 -0.322417       NaN -2.060141
5 -1.099891 -0.172428 -0.877858
```

其中，参数“limit=2”限制仅使用滞后值向前填充两个缺失值。

反之，也可使用“未来值”向后填充缺失值：

```
In [61]: df.fillna(method='bfill')
Out[61]:
          0         1         2
0  1.624345 -0.611756 -0.528172
1 -1.072969  0.865408 -2.301539
2  1.744812 -0.172428  0.319039
3 -0.249370 -0.172428 -2.060141
4 -0.322417 -0.172428 -0.877858
5 -1.099891 -0.172428 -0.877858
```

其中，参数“`method='bfill'`” 表示“back fill”；但较少见。

另一种常见插值方法为使用每个变量的均值进行插值，可输入命令：

```
In [62]: df.fillna(df.mean())
Out[62]:
          0         1         2
0  1.624345 -0.611756 -0.528172
1 -1.072969  0.865408 -2.301539
2  1.744812  0.027074  0.319039
3 -0.249370  0.027074 -2.060141
4 -0.322417  0.027074 -1.089734
5 -1.099891 -0.172428 -0.877858
```

其中，第 1 个变量的均值为 0.027074，故以之插值；以此类推。

类似地，也可使用每个变量的中位数进行插值：

```
In [63]: df.fillna(df.median())
Out[63]:
          0         1         2
0  1.624345 -0.611756 -0.528172
1 -1.072969  0.865408 -2.301539
2  1.744812 -0.172428  0.319039
3 -0.249370 -0.172428 -2.060141
4 -0.322417 -0.172428 -0.877858
5 -1.099891 -0.172428 -0.877858
```

其中，第 1 个变量的中位数为-0.172428，而第 2 个变量的中位数为-0.877858，故分别作为这两个变量的插值。

另一插值方法为**线性插值**(linear interpolation)，即假设数据为“等距间隔”(equally spaced)，并使用线性函数进行插值；这可通过 `interpolate()` 方法来实现：

```
In [64]: df.interpolate()
Out[64]:
          0         1         2
0  1.624345 -0.611756 -0.528172
1 -1.072969  0.865408 -2.301539
2  1.744812  0.605949  0.319039
3 -0.249370  0.346490 -2.060141
4 -0.322417  0.087031 -1.469000
5 -1.099891 -0.172428 -0.877858
```

其中，第 2 个变量第 4 个观测值的插值-1.469000，是其前后观测值-2.060141 与-2.060141 的算术平均数。

现实数据常有缺失值。如何正确处理缺失值，以达到最佳预测效果，可能并非易事。

下面，以 Kaggle 的泰坦尼克号存活数据作为处理缺失值的“实战”案例。Kaggle 是一个著名的机器学习竞赛平台，其 titanic 数据则是一个经典的竞赛数据集。

* 详见教材，以及配套 Python 程序（现场演示）。

18.5 重复观测值

数据清理的另一常见任务是考察数据中是否有“重复观测值”(duplicate observations)，即两个或多个观测值完全相同。

重复观测值有时是合理的，但也可能是原始数据有误。

作为示例，先建立一个有重复观测值的数据框：

```
In [94]: data = pd.DataFrame({'k1': ['one', 'two'] * 3
                              + ['two'], 'k2': [1, 1, 2, 3, 3, 4, 4]})
    ...: data
Out[94]:
    k1  k2
0  one   1
1  two   1
2  one   2
3  two   3
4  one   3
5  two   4
6  two   4
```

其中，第 5 与第 6 个观测值重复(完全一样)。

可使用 duplicated()方法，找出重复观测值：

```
In [95]: data.duplicated()
Out[95]:
0    False
1    False
2    False
3    False
4    False
5    False
6    True
dtype: bool
```

其中，第 6 个观测值的返回值为“True”，表明这是重复观测值。

如果确定重复观测值的数据有误，一种解决方法为通过查看原始数据来源，修正重复观测值，使之不再重复。

另一解决方法则直接去掉多余的重复观测值，可通过drop_duplicates()方法来实现：

```
In [96]: data.drop_duplicates()
Out[96]:
    k1   k2
0  one   1
1  two   1
2  one   2
3  two   3
4  one   3
5  two   4
```

其中，重复的第 6 个观测值已被去掉。

方法 drop_duplicates()默认保留第 1 个不重复的观测值。若想保留最后 1 个观测值，可输入参数“keep='last'”：

```
In [97]: data.drop_duplicates(keep='last')
Out[97]:
    k1  k2
0  one   1
1  two   1
2  one   2
3  two   3
4  one   3
6  two   4
```

其中，重复的第 5 个观测值被去掉，而第 6 个观测值则被保留。

另外，方法 `drop_duplicates()`默认根据所有列(变量)来判断是否为重复观测值。

若只想根据某一列(变量)来定义重复观测值，可以列表形式输入变量名，例如

```
In [98]: data.drop_duplicates(['k1'])
Out[98]:
   k1  k2
0  one   1
1  two   1
```

其中，仅根据变量 k1 的取值来判断观测值是否重复，故仅剩下 k1 的两个不同取值所对应的观测值。

18.6 合并数据

在实践中，数据常来自不同渠道，故有时需将这些数据框有意义地合并在一起。

比如，将工业企业数据与海关数据，根据产品进行匹配合并。

对于数组，可使用 Numpy 模块的 `concatenate()`函数进行多个数组的简单连接与并置。

为演示目的，随机生成两个2×2矩阵：

```
In [1]: np.random.seed(1)
   ...: matrix1 = np.random.randn(2, 2)
   ...: matrix1
Out[1]:
array([[ 1.62434536, -0.61175641],
       [-0.52817175, -1.07296862]])

In [2]: np.random.seed(12)
   ...: matrix2 = np.random.randn(2, 2)
   ...: matrix2
Out[2]:
array([[ 0.47298583, -0.68142588],
       [ 0.2424395 , -1.70073563]])
```

使用 Numpy 的 concatenate()函数合并矩阵 matrix1 与 matrix2：

```
In [3]: np.concatenate((matrix1, matrix2))
Out[3]:
array([[ 1.62434536, -0.61175641],
       [-0.52817175, -1.07296862],
       [ 0.47298583, -0.68142588],
       [ 0.2424395 , -1.70073563]])
```

其中，参数“(matrix1, matrix2)”以元组(tuple)形式输入需要合并的数组。

concatenate()函数默认进行纵向合并，即“axis=0”。若要进行横向合并，可输入参数“axis=1”：

```
In [4]: np.concatenate((matrix1, matrix2), axis=1)
Out[4]:
array([[ 1.62434536, -0.61175641,  0.47298583, -0.68142588],
       [-0.52817175, -1.07296862,  0.2424395 , -1.70073563]])
```

对于数组的纵向合并，也可等价地使用 Numpy 的 vstack()函数：

```
In [5]: np.vstack((matrix1, matrix2))
Out[5]:
array([[ 1.62434536, -0.61175641],
       [-0.52817175, -1.07296862],
       [ 0.47298583, -0.68142588],
       [ 0.2424395 , -1.70073563]])
```

其中，函数 vstack()表示“vertical stack”。

类似地，横向合并数组，也可等价地用 hstack()函数来实现：

```
In [6]: np.hstack((matrix1, matrix2))
Out[6]:
array([[ 1.62434536, -0.61175641,  0.47298583, -0.68142588],
       [-0.52817175, -1.07296862,  0.2424395 , -1.70073563]])
```

其中，函数 hstack()表示“horizontal stack”。

当然，数据框的合并在实践中更为常见。

pandas 的 concat()函数，在功能上类似于 Numpy 的 concatenate()函数，例如

```
In [7]: df1 = pd.DataFrame(matrix1)
   ...: df2 = pd.DataFrame(matrix2)
   ...: pd.concat([df1, df2])
Out[7]:
          0         1
0  1.624345 -0.611756
1 -0.528172 -1.072969
0  0.472986 -0.681426
1  0.242439 -1.700736
```

其中，先将矩阵 matrix1 与 matrix2 转换为数据框 df1 与 df2，然后以列表形式“[df1, df2]”作为参数输入 pd.concat()函数。

pd.concat()函数默认进行纵向合并，即“axis=0”。

若要进行横向合并，可输入参数“axis=1”：

```
In [8]: pd.concat([df1, df2], axis=1)
Out[8]:
          0         1         0         1
0  1.624345 -0.611756  0.472986 -0.681426
1 -0.528172 -1.072969  0.242439 -1.700736
```

另外，对于数据框进行纵向合并，也可使用 append()方法，例如：

```
In [9]: df1.append(df2)
Out[9]:
          0         1
0  1.624345 -0.611756
1 -0.528172 -1.072969
0  0.472986 -0.681426
1  0.242439 -1.700736
```

此命令将数据框 df2 纵向叠放在数据框 df1 之后。

作为另一示例，生成如下两个数据框：

```
In [10]: df3 = pd.DataFrame({'A': ['A1', 'A2'], 'B':
                             ['B1', 'B2'], 'C': ['C1', 'C2']})
    ...: df3
Out[10]:
    A   B   C
0  A1  B1  C1
1  A2  B2  C2

In [11]: df4 = pd.DataFrame({'B': ['B1', 'B2'], 'C':
                           ['C1', 'C2'], 'D': ['D1', 'D2'], })
    ...: df4
```

```
Out[11]:
    B   C   D
0  B1  C1  D1
1  B2  C2  D2
```

数据框 df3 与 df4 的共同变量包括 B 与 C，而 A 与 D 则为不同的变量。

将数据框 df3 与 df4 进行纵向合并：

```
In [12]: pd.concat([df3, df4])
Out[12]:
    A   B   C    D
0   A1  B1  C1  NaN
1   A2  B2  C2  NaN
0  NaN  B1  C1   D1
1  NaN  B2  C2   D2
```

其中，所得结果将数据框 df3 与 df4 的变量取了“并集”(union)，共包括 A，B，C 与 D 四个变量。而变量 A 与 D 的缺失值则显示为“NaN”。

在进行纵向合并时，若只想包括两个数据框的共同变量，即将数据框 df3 与 df4 的变量取“交集”(intersection)，可输入命令：

```
In [13]: pd.concat([df3, df4], join='inner')
Out[13]:
    B   C
0  B1  C1
1  B2  C2
0  B1  C1
1  B2  C2
```

其中，参数“join='inner'”表示对两个数据框的变量取交集，即仅保留共同变量。默认参数“join='outer'”，对两个数据框的变量取并集。

以上合并数据的方法只是将二维表数据简单地横向或纵向拼接。

另一常见的合并数据框方法为，将两个数据框的观测值按照某个共同变量的取值进行匹配与合并。

进行匹配所用的共同变量称为**关键字**(key)。

假设某企业需合并以下两个关于员工的数据框：

```
In [14]: df5 = pd.DataFrame({'employee': ['Bob', 'John',
                'Lisa', 'Sue'], 'group': ['Accounting',
                'Engineering', 'Engineering', 'HR']})
    ...: df5
Out[14]:
  employee        group
0      Bob   Accounting
1     John  Engineering
2     Lisa  Engineering
3      Sue           HR
```

```
In [15]: df6 = pd.DataFrame({'employee': ['Lisa', 'Bob',
                'Tom', 'Sue'], 'hire_date': [2004, 2008,
                2012, 2014]})
    ...: df6
Out[15]:
  employee  hire_date
0     Lisa       2004
1      Bob       2008
2      Tom       2012
3      Sue       2014
```

其中，数据框 df5 包含员工姓名与所属部门，而数据框 df6 包含员工姓名与雇佣年份，但两个数据框所包含的员工姓名不完全相同。

使用 pandas 的 merge()函数，以这两个数据框的共同变量“employee”作为“关键字”(key)进行匹配与合并：

```
In [16]: pd.merge(df5, df6, on='employee')
Out[16]:
  employee        group  hire_date
0      Bob   Accounting       2008
1     Lisa  Engineering       2004
2      Sue           HR       2014
```

其中，参数“on='employee'”表示根据变量 employee 进行匹配，然后合并这两个数据框。

如果两个数据框只有 1 个共同变量(在此为变量 employee)，也可省略参数“on='employee'”。

结果显示，数据框合并后的观测值，根据关键字 employee 的取值取了交集(intersection)，故既去掉了 df5 中“employee=John”的观测值，也去掉了 df6 中“employee=Tom”的观测值。

若想在合并数据框时，根据关键字 employee 的取值取并集(union)，即保留所有观测值，可输入参数“how='outer'”：

```
In [17]: pd.merge(df5, df6, on='employee', how='outer')
Out[17]:
  employee        group  hire_date
0      Bob   Accounting     2008.0
1     John  Engineering        NaN
2     Lisa  Engineering     2004.0
3      Sue           HR     2014.0
4      Tom          NaN     2012.0
```

其中，“employee=John”与“employee=Tom”的部分缺失值以“NaN”表示。

默认参数“how='inner'”表示，根据关键字的取值而选取观测值的交集。

如果输入参数“how='left'”，则将保留左边数据框(即 df5)的所有观测值，比如

```
In [18]: pd.merge(df5, df6, on='employee', how='left')
Out[18]:
  employee        group  hire_date
0      Bob   Accounting     2008.0
1     John  Engineering        NaN
2     Lisa  Engineering     2004.0
3      Sue           HR     2014.0
```

类似地，输入参数“how='right'”，则可保留右边数据框(即 df6)的所有观测值：

```
In [19]: pd.merge(df5, df6, on='employee', how='right')
Out[19]:
  employee        group  hire_date
0      Bob   Accounting       2008
1     Lisa  Engineering       2004
2      Sue           HR       2014
3      Tom          NaN       2012
```

有时据以匹配的关键字在两个数据框的变量名称不相同，此时可将参数“on”进一步细化为“left_on”与“right_on”。考虑以下数据框：

```
In [20]: df7 = pd.DataFrame({'name': ['Lisa', 'Bob',
                    'Tom', 'Sue'], 'hire_date': [2004,
                     2008, 2012, 2014]})
    ...: df7
Out[20]:
   name  hire_date
0  Lisa       2004
1   Bob       2008
2   Tom       2012
3   Sue       2014
```

其中，数据框 df7 在内容上与数据框 df6 相同，只是 df6 的变量名“employee”被改为 df7 的变量名“name”而已。

根据 df5 的变量 employee 与 df7 的变量 name，将 df5 与 df7 进行匹配与合并：

```
In [21]: pd.merge(df5, df7, left_on='employee', right_on='name')
Out[21]:
  employee        group  name  hire_date
0      Bob   Accounting   Bob       2008
1     Lisa  Engineering  Lisa       2004
2      Sue           HR   Sue       2014
```

其中，参数“left_on='employee'”表示 df5 的关键字变量为 employee，而参数“right_on='name'”表示 df7 的关键字变量为 name。

在合并后的数据框中，变量 employee 与变量 name 的信息重复，可使用 drop()方法去掉多余的变量 name：

```
In [22]: pd.merge(df5, df7, left_on='employee',
                  right_on='name').drop('name', axis=1)
Out[22]:
  employee        group  hire_date
0      Bob   Accounting       2008
1     Lisa  Engineering       2004
2      Sue           HR       2014
```

以上合并数据框的方法为“一对一”(one-to-one)，它将左边数据框的每个观测值与右边数据框的某个观测值进行匹配与合并。

也可以进行“多对一”(many-to-one)合并。

作为示例，生成以下两个数据框：

```
In [23]: df8 = pd.DataFrame({'key': ['b', 'b', 'a', 'c',
                                     'a', 'a', 'b'], 'var1': range(7)})
    ...: df8
Out[23]:
  key  var1
0   b     0
1   b     1
2   a     2
3   c     3
4   a     4
5   a     5
6   b     6
```

```
In [24]: df9 = pd.DataFrame({'key': ['a', 'b', 'd'],
                             'var2': range(3)})
    ...: df9
Out[24]:
  key  var2
0   a     0
1   b     1
2   d     2
```

其中，数据框 df8 包含 7 个观测值，而数据框 df9 包括 3 个观测值。

按照共同的关键字“key”，将这两个数据框进行“多对一”合并：

```
In [25]: pd.merge(df8, df9, on='key')
Out[25]:
  key  var1  var2
0   b     0     1
1   b     1     1
2   b     6     1
3   a     2     0
4   a     4     0
5   a     5     0
```

结果显示，合并后的数据框包含 6 个观测值。

在合并数据框时，也可将数据框的“索引”(index)作为关键字。作为示例，使用 set_index()方法，将 df5 与 df6 的变量 employee 设为索引，并将所得数据框分别记为 df5a 与 df6a：

```
In [26]: df5a = df5.set_index('employee')
    ...: df5a
Out[26]:
                group
employee
Bob          Accounting
John        Engineering
Lisa        Engineering
Sue                  HR
```

```
In [27]: df6a = df6.set_index('employee')
    ...: df6a
Out[27]:
          hire_date
employee
Lisa           2004
Bob            2008
Tom            2012
Sue            2014
```

然后，输入“left_index”与“right_index”作为merge()函数的参数：

```
In [28]: pd.merge(df5a, df6a, left_index=True,
                  right_index=True)
Out[28]:
             group   hire_date
employee
Bob       Accounting       2008
Lisa     Engineering       2004
Sue               HR       2014
```

其中，参数“left_index=True”与“right_index=True”表示，以df5a与df6a的索引作为关键字进行匹配与合并。

对于这种根据数据框索引进行的合并，也可使用 join()方法来实现：

```
In [29]: df5a.join(df6a)
Out[29]:
             group   hire_date
employee
Bob       Accounting     2008.0
John     Engineering        NaN
Lisa     Engineering     2004.0
Sue               HR     2014.0
```

在使用 merge()函数合并数据框时，也可混合使用参数“left_index=True”与“right_on='key'”，或者混合使用“left_on='key'”与“right_index=True”。

例如，若要合并数据框 df5a 与 df6，可输入命令：

```
In [30]: pd.merge(df5a, df6, left_index=True,
                  right_on='employee')
Out[30]:
        group employee  hire_date
1   Accounting      Bob       2008
0  Engineering     Lisa       2004
3           HR      Sue       2014
```

在合并数据框时，有时会遇到两个数据框的变量名冲突的情形。这意味着，两个数据框的变量名相同，而内容却不同。

作为示例，生成以下两个数据框：

```
In [31]: df10 = pd.DataFrame({'employee': ['Bob',
                                           'John','Lisa','Sue'],'rank':
                                           [1, 2, 3, 4]})
   ...: df10
Out[31]:
  employee  rank
0      Bob     1
1     John     2
2     Lisa     3
3      Sue     4
```

```
In [32]: df11 = pd.DataFrame({'employee': ['Bob', 'John',
                    'Lisa', 'Sue'], 'rank': [3, 1, 4, 2]})
    ...: df11
Out[32]:
  employee  rank
0      Bob     3
1     John     1
2     Lisa     4
3      Sue     2
```

其中，数据框 df10 与 df11 都有同名变量 rank，但内容不同；可能是关于员工的两种不同排名。

以变量 employee 作为关键字，将这两个数据框行合并：

```
In [33]: pd.merge(df10, df11, on='employee')
Out[33]:
  employee  rank_x  rank_y
0      Bob       1       3
1     John       2       1
2     Lisa       3       4
3      Sue       4       2
```

其中，所得结果已自动将同名变量加了后缀(suffix)，分别设为“rank_x”与“rank_y”。

若想使用其他后缀，可以列表形式输入参数“suffixes”，比如：

```
In [34]: pd.merge(df10, df11, on='employee',
                  suffixes=['_L', '_R'])
Out[34]:
  employee  rank_L  rank_R
0      Bob       1       3
1     John       2       1
2     Lisa       3       4
3      Sue       4       2
```

结果显示，同名变量“rank”被分别设为“rank_L”与“rank_R”。

18.7 Sci-Kit Learn 的管线类

在机器学习或数据科学的实践中，常需对原始数据进行一系列的变换，或依次使用若干算法，从而构成一个**算法链**(algorithm chains)。

比如，先对原始数据进行标准化(standardization)的预处理，然而再估计支持向量机的模型。

Sci-Kit Learn 模块提供了方便的**管线类**(`Pipeline class`)，使得创建“算法链”的过程更为简单。

而且，`Pipeline` 类可与 `GridSearchCV` 类相结合，针对整个算法链进行超参数的网格搜索。

以 sklearn 自带的威斯康辛乳腺癌数据 `breast_cancer` 为例(详见第 10 章)，使用支持向量机进行二分类的预测。

* 详见教材，以及配套 Python 程序（现场演示）。

18.8 结束语

本书介绍了不少模型，特别是监督学习的算法。是否存在任何情况下都最优秀的算法呢？

答案是否定的。

如果不对数据生成过程(data generating process)作任何假设，我们无法预先(a priori)判断任意两个算法的相对优越性。

这就是机器学习中著名的**无免费午餐定理** (No Free Lunch Theorem，简记 NFL)，由 Wolpert(1996)与 Wolpert and Macready(1997)提出。

直观上，如果数据生成过程为线性，则最简单的线性模型也可能胜过复杂的非线性模型，故无法预判孰优孰劣。

反之，如果真实的决策边界为矩形，则基于决策树的算法有望胜出。

因此，针对具体数据与问题，究竟哪个算法最优，还须用数据来回答。

或许，这正是机器学习的魅力所在吧。