前言

資料科學由數學與計算機科學的多種子領域組成。統計、線性代數、資料庫、機器智慧、資料視覺化是資料科學領域所融合的部分主題。資料科學實務工具與技術的發展非常快。本書專注於以 Java 物件導向程式實作核心基本概念。在啟發你著手進行資料科學實務的同時，我也希望能夠引導你打造下一代的資料科學技術。

本書讀者

本書的對象是已經熟悉應用程式開發的科學家與工程師，內容涵蓋資料科學程序、數學理論以及程式範例。這本書是深入研究的起點。

寫作動機

我寫這本書是為了推動。資料科學由 R 與 Python 推動，僅少數人探索 Java 領域。解譯語言顯然是主流資料工具，但還有工程與科學領域對擴展性、可靠性、方便性的需求，Java 或許是一個答案。若本書能帶給你啟發，我希望你會對支援資料科學的開源 Java 專案做出貢獻。

對現今資料科學的看法

資料科學持續變化中，不僅是範圍，還包括實作方式。技術演變非常快，主流演算法來去以年甚或是月計。長久以來的標準化做法被實務淘汰，而成功方案經常受到量化科學

領域之外的人的干擾。資料科學已經是一門大學科系。未來要成功只有一個辦法：認識數學、程式設計、重要主題。

導讀

這本書依資料科學程序的邏輯安排。第一章討論取得、清理、安排資料成最純形式的多種方法，包括基本資料輸出。第二章討論視資料為矩陣的重要概念，包括矩陣操作。取得資料並知道應該採用的格式後，第三章介紹檢驗資料的基本概念。第四章使用第二章與第三章的概念將資料轉換成穩定可用的數值。第五章討論監督式與非監督式學習演算法以及評估方法。第六章使用適用於資料科學演算法的自定元件安裝與執行MapReduce。附錄 A 說明一些實用資料集。

本書編排慣例

本書以下列各種字體來達到強調或區別的效果：

斜體字（*Italic*）

代表新名詞、網址 URL、電子郵件、檔案名稱及檔案屬性。中文以楷體表示。

定寬字（`Constant width`）

用於標示程式碼，或是在本文段落中標註程式片段，如變數或函式名稱、資料庫、資料型別、環境變數、陳述式、關鍵字等等。

定寬粗體字（**`Constant width bold`**）

標示指令或其他由使用者輸入的文字。

定寬斜體字（*`Constant width italic`*）

標示應以使用者輸入或是依前後文決定內容來取代的文字。

這個圖示代表提示或建議。

這個圖示代表一般註解。

第一章

資料 I/O

事件持續在我們周遭發生。我們通常在特定時間點與空間記錄離散事件，然後將某人（或某物）以各種格式所做的記錄定義為資料。作為資料科學家，我們在檔案、資料庫、網路服務上運用資料。通常有人克服種種困難定義出精確描述所有變數的名稱、類型、範圍、關係的資料格式或模型，但不一定能以指定格式獲取資料。真實的資料（甚至是經過設計的資料庫）經常帶有缺漏、拼寫錯誤、不正確的格式、重複值，以及最糟的狀況：多個變數連在一起。雖然你可以實作機器學習演算法並產生精美的圖表，但資料科學中最重要與最耗時的部分是準備資料並確保其正確性。

資料是什麼？

你的終極目標是從來源讀取資料，透過統計分析或學習減少資料，然後展示學習到的某種知識，通常是以圖表的方式呈現。但就算結果是加總、個別使用者或數量等單一值，你還是必須依循相同的程序：輸入資料→還原分析→輸出資料。

實務資料科學是由商業需求驅動，因此從右向左檢視此程序是有利的。首先將你嘗試回答的問題正規化。舉例來說，你需要依地區列出最常使用者、預測下一週的日營業額或庫存項目分類圖表嗎？接下來探索可回答問題的分析鏈。最後，決定採用的方法後，需要什麼資料來達成這個目標？你可能會發現沒有所需的資料。通常你會發現（比原來想象）較簡單的工具就能產生所需的輸出。

這一章探索從各種資料來源讀寫資料的細節。每個次步驟需要什麼資料模型是很重要的問題。或許建構一系列數值陣列型別（例如 `double[][]`、`int[]`、`String[]`）就足以保存資料。另一方面，建構容器類別來保存每一組資料，然後產生這些物件的 `List` 或

Map 可能會比較好。還有一種資料模型是將每一筆記錄做成 JavaScript Object Notation（JSON）文件中的鍵值對。資料模型的選擇大部分依後續資料消耗程序的輸入需求而定。

資料模型

進來的資料是什麼格式？要轉換成什麼格式才能送出？假設 somefile.txt 內有 id、year、city 資料列。

單變量陣列

此時最簡單的資料模型是建構 id、year、city 三個變數的一系列陣列：

```
int[] id = new int[1024];
int[] year = new int[1024];
String[] city = new String[1024];
```

BufferedReader 會逐行讀入檔案，遞增計數以將值加到陣列中。此資料模型或許適合已知維度的乾淨資料（clean data），程式最後產生一個可執行的類別。將此資料餵給各種統計分析或學習演算法很簡單。但你或許會想要將程式模組化，並建構適合各種資料來源與資料模型的類別與方法。在這種情況下，必須改變現有方法以適應新參數時處理陣列會很麻煩。

多變量陣列

此時你想要每一列保存一筆記錄的所有資料，但它們必須是相同型別！在這種情況下，只有將城市指定為整數值才可行：

```
int[] row1 = {1, 2014, 1};
int[] row2 = {2, 2015, 1};
int[] row3 = {3, 2014, 2};
```

你也可以將它做成二維陣列：

```
int[][] data = {{1, 2014, 1}, {2, 2015, 1}, {3, 2014, 2}};
```

一開始的資料集可能是複雜的資料模型，或者混合了文字、整數、雙精度與日期時間。理想中，在你找出什麼資料會進入統計分析或學習演算法後，此資料會轉換成雙精度二維陣列。但這需要相當多的工作。一方面，機器學習處理資料矩陣很方便，另一方面，你可能不知道遺漏或產生了什麼錯誤。

資料物件

另一個選項是建構容器類別然後產生該容器的 List 或 Map 集合。優點是它能集中特定記錄的值，且加入新成員到類別中而不會破壞以該類別為參數的方法。somefile.txt 檔案中的資料可以下列類別表示：

```
class Record {
    int id;
    int year;
    String city;
}
```

盡可能保持類別的輕量化，因為這些類別的集合（List 或 Map）會累積成巨大的資料集！操作 Record 的任何方法可以是其類別內命名類似 RecordUtils 的靜態方法。

集合結構 List 用於保存所有 Record 物件：

```
List<Record> listOfRecords = new ArrayList<>();
```

以 BufferReader 讀取資料檔案，解析每一行並將內容儲存在新的 Record 實例中，然後將新的 Record 實例加入 List<Record> listOfRecords。若需要一個鍵以快速的查詢與讀取個別 Record 實例，則使用 Map：

```
Map<String, Record> mapOfRecords = new HashMap<>();
```

每一筆記錄的鍵應該是獨特的，例如記錄的 ID 或 URL。

矩陣與向量

矩陣與向量是二維與一維陣列組成的高階資料結構。資料集通常帶有多個欄與列，我們可以說這些變數組成有 m 列與 n 欄的二維陣列（或矩陣）X。我們選擇 i 作為列索引，j 作為欄索引，使 $m \times n$ 矩陣的元素為 $x_{i,j}$

$$\begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m,1} & x_{m,2} & \cdots & x_{m,n} \end{pmatrix}$$

將值放到矩陣等資料結構會增加方便性。通常我們會對資料執行數學運算，矩陣實例可帶有執行這些運算的抽象方法，而實作細節則適用於目前的工作。第二章會深入討論矩陣與向量。

JSON

JavaScript Object Notation（JSON）是目前流行的資料表示形式。一般來說，JSON 資料以 *json.org* 的簡單規則表示：雙引號！後面沒有逗號！ JSON 物件外層有大括弧，包含以逗號分隔的任意組數鍵值對（不保證內容的順序，因此要視為 `HashMap` 型別）：

```
{"city":"San Francisco", "year": 2020, "id": 2, "event_codes":[20, 22, 34, 19]}
```

JSON 陣列外層有方括號，裡面是以逗號分隔的有效 JSON（保證陣列內容的順序，因此要視為 `ArrayList` 型別）：

```
[40, 50, 70, "text", {"city":"San Francisco"}]
```

你會發現兩種主要類型。有些資料檔案帶有完整的 JSON 物件或陣列，通常是組態檔案。但另一種常見的資料結構是每一行一個獨立 JSON 物件的文字檔案。請注意，這種資料結構（JSON 資料列）技術上並非 JSON 物件或陣列，因為行之間沒有括弧或逗號，所以將整個資料結構當做一個 JSON 物件（或陣列）解析會失敗。

處理真實資料

真實資料混亂、不完整、不正確有時還不一致。若你操作的是“完美”的資料集，那是因為別人花了時間整理好。事實上你的資料很有可能不完美，而你分析的是垃圾資料。唯一可以確定的辦法是自行從來源取得資料並加以處理。如此，若有錯誤，你會知道誰該負責。

空

空值以各種形式出現。若資料在 Java 中傳遞，則有可能帶有 `null`。若解析來自文字檔案的字串，`null` 可能以 `"null"`、`"NULL"`、`"na"` 等各種文字或句號表示。不論是哪一種（空型別或空文字），我們都必須處理：

```
private boolean checkNull(String value) {
    return value == null || "null".equalsIgnoreCase(value);
}
```

空值通常記錄為空白或一系列空白。雖然這有時候很討厭，但它有多種用途，因為編成 0 不一定適合表示不存在的資料。舉例來說，若要記錄 0 與 1 的二進位資料，而有個項目的值不知道，指派值為 0（並寫入檔案中）可能會導致錯誤指派了實際上的負值。將空值寫入檔案時，我偏好零長度字串。

空白

實際資料有大量的空白。以 `String.isEmpty()` 方法檢查空字串很簡單，但要注意空白字串（甚至是一個空白）並非空！首先我們使用 `String.trim()` 方法刪除輸入值前後的空白，然後檢查它的長度。`String.isEmpty()` 僅於字串長度為零時回傳 `true`：

```
private boolean checkBlank(String value) {
    return value.trim().isEmpty();
}
```

解析錯誤

知道字串值並非空或空白後，將它解析成我們需要的型別。我們會排除將字串解析成字串，因為這並沒有東西要解析！

處理數值時，將字串轉換成 `double`、`int` 或 `long` 等原始型別是不智的。建議使用 `Double`、`Integer` 或 `Long` 等物件包裝類別，它們具有字串解析方法，能在有問題時拋出 `NumberFormatException`。我們可以捕捉此例外並更新解析錯誤計數。也可以輸出或記錄此錯誤：

```
try {
    double d = Double.parseDouble(value);
    // 處理 d
} catch (NumberFormatException e) {
    // 遞增解析錯誤計數等
}
```

同樣的，日期時間格式的字串可用 `OffsetDateTime.parse()` 方法解析；輸入字串有問題時捕捉 `DateTimeParseException` 並記錄錯誤：

```
try {
    OffsetDateTime odt = OffsetDateTime.parse(value);
    // 處理 odt
} catch (DateTimeParseException e) {
    // 遞增解析錯誤計數等
}
```

異常值

清理與解析資料後，可以檢查值是否符合需求。若預期值為 0 或 1 但收到 2，則該值明顯超出範圍，可以標示此資料點為異常值。如同空與空白，我們可以對值執行布林測試以判斷是否在可接受的值範圍內。這對數值、字串與日期都適用。

檢查數值範圍時，必須知道可接受的最大與最小值與是否包含界限值。舉例來說，若設定 `minValue = 1.0` 且 `minValueInclusive = true`，大於或等於 1.0 的值都能通過檢查。若設定 `minValueInclusive = false`，則只有大於 1.0 的值會通過檢查：

```
public boolean checkRange(double value) {
    boolean minBit = (minValueInclusive) ? value >= minValue : value > minValue;
    boolean maxBit = (maxValueInclusive) ? value <= maxValue : value < maxValue;
    return minBit && maxBit;
}
```

類似方法可用於整數型別。

我們也可以設定列舉字串以檢查字串值是否在可接受範圍內。這可以透過建立稱為 `validItems` 有效字串的 `Set` 實例，以 `Set.contains()` 方法檢查輸入值是否有效：

```
private boolean checkRange(String value) {
    return validItems.contains(value);
}
```

對 `DateTime` 物件，我們可以檢查日期是否在範圍內。在這種情況下，定義 `OffsetDateTime` 的最大與最小值，然後測試輸入日期時間是否介於最大與最小之間。請注意，範圍不包括 `OffsetDateTime.isBefore()` 與 `OffsetDateTime.isAfter()`。若輸入的日期時間等於最大或最小則不會通過測試。程式如下：

```
private boolean checkRange(OffsetDateTime odt) {
    return odt.isAfter(minDate) && odt.isBefore(maxDate);
}
```

管理資料檔案

資料科學的藝術就在這裡！如何建構資料集不只關於效率，還要有彈性。讀寫檔案有很多選項。至少整個檔案可以使用 `FileReader` 的實例讀入成 `String` 型別，然後 `String` 可以再解析成資料模型。對較大的檔案可以使用 `BufferedReader` 逐行讀取以避免 I/O 錯誤。這種策略是讀一行解析一行，僅保存必要的值並產生資料結構。若每一行有 1000

個變數且只需要其中的三個，則沒有必要全部保存。同樣的，若某一行的資料不合需求，則也無需保存。對較大的資料集，這麼做比讀取所有行到字串陣列（String[]）並於之後解析更節省資源。管理資料檔案的步驟越深思熟慮越好。後續的統計、學習、製作圖表都與建構資料集的決定有關。“垃圾進，垃圾出”這個格言絕對適用於此。

先認識檔案內容

資料檔案的格式有很多種，有些帶有不理想的特質。ASCII 檔案內容為 ASCII 字元組成的行，沒有規定數字的格式或精度、字串的使用或雙引號，或包含（或排除）空白、空與換行字元。總而言之，無論如何假設檔案內容，行內還是有可能包含任何東西。以 Java 讀取檔案前，先以文字編輯器或命令列看過。請注意，每個項目的數字、位置、與型別。仔細檢視缺值或空值如何表示。還要注意分隔與描述資料的表頭的形式。若檔案很小，可以用肉眼掃描缺漏與錯誤格式。舉例來說，可以用 Unix 的 bash 的 less 命令檢視 somefile.txt 檔案：

```
bash$ less somefile.txt

"id","year","city"
1,2015,"San Francisco"
2,2014,"New York"
3,2012,"Los Angeles"
...
```

我們看到逗號分隔（CSV）資料集，具有 id、year、city 欄。可以快速的檢查檔案的行數：

```
bash$ wc -l somefile.txt
1025
```

這表示有 1024 行資料與一行表頭。還有其他格式，例如以 tab 分隔的值（TSV）、所有值接在一起的“大字串”格式及 JSON。對較大的檔案，可以檢視前 100 或其他數量的行並輸出到樣本檔案供開發使用：

```
bash$ head -100 filename > new_filename
```

在某些情況下，資料檔案太大以至於無法以肉眼掃描結構或錯誤。1000 欄的資料檔案明顯很難檢視！同樣的，不可能從百萬行資料中找出錯誤。在這種情況下，基本上你會有描述欄格式與資料型別（例如整數、浮點數、文字）的資料字典。你可以用程式在解析檔案時檢查每一行資料；可能會拋出例外並輸出有問題行的內容以供你檢查。

讀取文字檔案

讀取文字檔案的一般方法是建構以 BufferedReader 包圍 FileReader 實例以逐行讀取。此處的 FileReader 取用 String 檔名參數，但 FileReader 也可以取用 File 物件作為參數。File 物件在檔名與路徑相依作業系統時很有用。以下是以 BufferedReader 逐行讀取檔案的通用形式：

```
try(BufferedReader br = new BufferedReader(new FileReader("somefile.txt")) ) {
    String columnNames = br.readline(); // 只能在它存在時如此做
    String line;
    while ((line = br.readLine()) != null) {
        /* 解析每一行 */
        // TODO
    }
} catch (Exception e) {
    System.err.println(e.getMessage()); // 或記錄錯誤
}
```

若檔案存在與遠端也可以執行相同的工作：

```
URL url = new URL("http://storage.example.com/public-data/somefile.txt");
try(BufferedReader br = new BufferedReader(
    new InputStreamReader(url.openStream())) ) {
    String columnNames = br.readline(); // 只能在它存在時如此做
    String line;
    while ((line = br.readLine()) != null) {
        // TODO：解析每一行
    }
} catch (Exception e) {
    System.err.println(e.getMessage()); // 或記錄錯誤
}
```

我們只需思考如何解析每一行。

解析大字串

以每一列都是多個值連接的“大字串”，而特定變數子字串以起點與終點位置編碼的檔案為例：

```
0001201503
0002201401
0003201202
```

前四位數是 id，接下來四位數是 year，最後兩位數是 city 編號。要記得每一行可數到千個字元長，而字元子字串的位置很重要。通常數字前面會補零，空白會以空值表示。請注意，浮點數（例如 32.456）的小數點如同其他“奇怪”的字元一樣會佔用一個空間！文字字串有時會編碼成值，例如 New York = 01 , Los Angeles = 02 , and San Francisco = 03。

此例中，每一行中的值可透過 String.substring(int beginIndex, int endIndex) 方法存取。請注意，子字串從 beginIndex 開始到 enIndex（不含）結束：

```
/* 解析每一行 */
int id = Integer.parseInt(line.substring(0, 4));
int year = Integer.parseInt(line.substring(4, 8));
int city = Integer.parseInt(line.substring(8, 10));
```

解析分隔字串

試算表與資料庫匯出的資料很有可能是 CSV 資料集。解析這種檔案不容易！以 CSV 格式的範例資料為例：

```
1,2015,"San Francisco"
2,2014,"New York"
3,2012,"Los Angeles"
```

我們只需以 String.split(",") 解析並使用 String.trim() 來刪除前後的空白。還需要以 String.replace("\", "") 刪除前後的引號：

```
/* 解析每一行 */
String[] s = line.split(",");
int id = Integer.parseInt(s[0].trim());
int year = Integer.parseInt(s[1].trim());
String city = s[2].trim().replace("\"", "");
```

下面的範例將 *somefile.txt* 中的資料以 tab 分隔：

```
1       2015        "San Francisco"
2       2014        "New York"
3       2012        "Los Angeles"
```

分割 tab 分隔的資料可透過替換前面的範例的 String.split(",") 的參數來執行：

```
String[] s = line.split("\t");
```

有時會遇到欄中帶有逗號的 CSV 檔案。一個例子是部落格中的文字，另一個例子是反正規化欄的資料 - 例如“San Francisco, CA”而非城市與州各一欄。這相當麻煩且需要動用 regex（正規表示式）。何不使用 Apache Commons CSV 解析函式庫？

```
/* 解析每一行 */
CSVParser parser = CSVParser.parse(line, CSVFormat.RFC4180);
for(CSVRecord cr : parser) {
    int id = cr.get(1); // 欄從 1 而非 0 開始
    int year = cr.get(2);
    String city = cr.get(3);
}
```

Apache Commons CSV 函式庫還可以處理 *CSVFormat.EXCEL*、*CSVFormat.MYSQL*、*CSVFormat.TDF* 等格式。

解析 JSON 字串

JSON 是 JavaScript 物件序列化的協定，可擴充各種型別的資料。此精簡、易讀的格式常用於網際網路資料 API（特別是 RESTful 服務）且是 MongoDB 與 CouchDB 等多種 NoSQL 方案的標準格式。PostgreSQL 資料庫的 9.3 版提供 JSON 資料型別並能夠查詢原生 JSON 欄。其明顯的好處是肉眼可讀；資料結構相對易讀，透過“美化輸出”會更好讀。在 Java 中，JSON 不過就是 `HashMaps` 與 `ArrayLists` 的集合，可以任何組態套疊。前面範例的每一行資料可加上鍵 - 值對而格式化成為 JSON 字串；字串要放在雙引號（非單引號）中且後面不能有逗號：

```
{"id":1, "year":2015, "city":"San Francisco"}
{"id":2, "year":2014, "city":"New York"}
{"id":3, "year":2012, "city":"Los Angeles"}
```

請注意，整個檔案本身在技術上並非是個 JSON 物件，解析整個檔案會失敗。合法的 JSON 格式每一行必須以逗號分開並全部包在方括號中，如此會組成一個 JSON 陣列。然而，撰寫這種結構沒有效率且不實用。現在這樣反而比較方便：逐行堆起以字串表示的 JSON 物件。請注意，JSON 解析程序並不知道鍵值對中值的型別，因此要取得 `String` 表示再用 box 方法解析成原始型別。接下來建構資料集就很簡單，使用 `org.simple.json`：

```
/* 在 while 迴圈外建構 JSON 解析程序 */
JSONParser parser = new JSONParser();
...

/* 轉換解析過的字串以建構物件 */
```

```
JSONObject obj = (JSONObject) parser.parse(line);
int id = Integer.parseInt(j.get("id").toString());
int year = Integer.parseInt(j.get("year").toString());
String city = j.get("city").toString();
```

讀取 JSON 檔案

這一節討論字串化 JSON 物件或陣列的檔案。你應該事先就知道檔案是否為 JSON 物件或陣列。舉例來說，若從命令列使用 `ls` 檢視檔案，你應該能夠分辨它是否有大括弧（物件）或方括號（陣列）：

```
{{"id":1, "year":2015, "city":"San Francisco"},
 {"id":2, "year":2014, "city":"New York"},
 {"id":3, "year":2012, "city":"Los Angeles"}}
```

然後使用 Simple JSON 函式庫：

```
JSONParser parser = new JSONParser();
try{
    JSONObject jObj = (JSONObject) parser.parse(new FileReader("data.json"));
    // TODO：使用 jObj
} catch (IOException|ParseException e) {
    System.err.println(e.getMessage());
}
```

若它是個陣列：

```
[{"id":1, "year":2015, "city":"San Francisco"},
 {"id":2, "year":2014, "city":"New York"},
 {"id":3, "year":2012, "city":"Los Angeles"}]
```

則可以解析整個 JSON 陣列：

```
JSONParser parser = new JSONParser();
try{
    JSONArray jArr = (JSONArray) parser.parse(new FileReader("data.json"));
    // TODO：使用 jObj
} catch (IOException|ParseException e) {
    System.err.println(e.getMessage());
}
```

若檔案是每一行一個 JSON 物件，則檔案技術上並非合法的 JSON 資料結構。讀取檔案與逐行解析 JSON 物件見“讀取文字檔案”一節。

讀取影像檔案

使用影像作為學習輸入時，必須將影像格式（例如 PNG）轉換成合適的資料結構，例如矩陣或向量。有很多部分要考慮。首先，影像是二維陣列，具有座標 $\{x_1, x_2\}$ 與相關聯的顏色或亮度值 $\{y_1...\}$，它們可儲存成單一整數值。若只需要儲存在 2D 整數陣列（data）中的原始值，則如此讀入影像：

```
BufferedImage img = null;
try {
    img = ImageIO.read(new File("Image.png"));
    int height = img.getHeight();
    int width = img.getWidth();
    int[][] data = new int[height][width];
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            int rgb = img.getRGB(i, j); // 負值
            data[i][j] = rgb;
        }
    }
} catch (IOException e) {
    // 處理例外
}
```

我們可能需要將整數進行位元位移，以轉換成 RGB（紅、藍、綠）分量：

```
int blue = 0x0000ff & rgb;
int green = 0x0000ff & (rgb >> 8);
int red = 0x0000ff & (rgb >> 16);
int alpha = 0x0000ff & (rgb >> 24);
```

但也可以如此取得該資訊：

```
byte[] pixels = ((DataBufferByte) img.getRaster().getDataBuffer()).getData();
for (int i = 0; i < pixels.length / 3 ; i++) {
    int blue = Byte.toUnsignedInt(pixels[3*i]);
    int green = Byte.toUnsignedInt(pixels[3*i+1]);
    int red = Byte.toUnsignedInt(pixels[3*i+2]);
}
```

顏色也許不重要。或許真正需要的是灰階：

```
// 轉換 0 到 255 階的 rgb 成灰階 (0 到 1)
double gray = (0.2126 * red + 0.7152 * green + 0.0722 * blue) / 255.0
```

還有，某些情況下不需要 2D 表示。將矩陣轉換成向量，連接矩陣的每一列到新的向量使 $x_n = x_1, x_2, ...$，使向量長度 n 為 $m \times p$ 矩陣，也就是列數乘以欄數。在著名的手寫影像 MNIST 資料集中，資料已經被更正（置中與裁切）然後轉換成二進位格式。因此讀取該資料需要特殊格式（見附錄 A），但它已經是向量（1D）而非矩陣（2D）格式。對 MNIST 資料集的學習技術通常涉及這種向量化格式。

寫入文字檔案

將資料寫入檔案有個使用 `FileWriter` 類別的通用形式，但再次建議使用 `BufferedWriter` 以避免任何 I/O 錯誤。一般概念是將要寫入檔案的所有資料格式化成單一字串。對我們的範例的三個變數，可以手動執行加上所選擇的分隔（逗號或 `\t`）：

```
/* 對每一筆記錄 */
String output = Integer.toString(record.id) + "," +
Integer.toString(record.year) + "," + record.city;
```

使用 Java 8 時，`String.join(delimiter, elements)` 很方便！

```
/* in Java 8 */
String newString = String.join(",", {"a", "b", "c"});

/* 或輸入 Iterator */
String newString = String.join(",", myList);
```

不然可在迴圈中使用 Apache Commons Lang 的 `StringUtils.join(elements, delimiter)` 或原生的 `StringBuilder` 類別：

```
/* in Java 7 */
String[] strings = {"a", "b", "c"};

/* 建構 StringBuilder 並加入第一個成員 */
StringBuilder sb;
sb.append(strings[0]);

/* 略過第一個字串，因為已經加入了 */
for(int i = 1; i < strings.length, i++){
    /* 選擇分隔 ... 也可以使用 \t 或 tab */
    sb.append(",");
    sb.append(strings[i]);
}

String newString = sb.toString();
```

請注意，連續使用 myString += myString_part 會呼叫 StringBuilder 類別，因此也可以使用 StringBuilder（或不這麼做）。無論是哪一種，字串都是逐行寫入。要記得 BufferedWriter.write(String) 方法不會寫入換行！若要讓每一筆資料記錄獨立一行則必須加上 BufferedWriter.newLine()：

```
try(BufferedWriter bw = new BufferedWriter(new FileWriter("somefile.txt")) ) {
    for(String s : myStringList){
        bw.write(s);
        /* 不要忘記加上換行！ */
        bw.newLine();
    }
} catch (Exception e) {
    System.out.println(e.getMessage());
}
```

前面的程式會覆寫檔名指定檔案中現有資料。有些情況下你想要加入現有檔案。FileWriter 類別以預設為 false 的 append 欄位選擇。若要開啟檔案供加入下一行：

```
/* 設定 FileWriter 的 append 以保存現有資料並加入新資料 */
try(BufferedWriter bw = new BufferedWriter(
    new FileWriter("somefile.txt", true))) {
    for(String s : myStringList){
        bw.write(s);
        /* 不要忘記加上換行！ */
        bw.newLine();
    }
} catch (Exception e) {
    System.out.println(e.getMessage());
}
```

還有一個選項是使用包裝 BufferedWriter 的 PrintWriter 類別。PrintWriter 有個 println() 方法使用目前作業系統原生的換行字元，因此程式可以不用加上 \n。這麼做的好處是不用擔心加上這些討厭的換行字元。這在你於自己的電腦（以及 OS）上產生文字檔案時很有用。下面是使用 PrintWriter 的範例：

```
try(PrintWriter pw = new PrintWriter(new BufferedWriter(
    new FileWriter("somefile.txt"))) ) {
    for(String s : myStringList){
        /* 幫你加入換行！ */
        pw.println(s);
    }
} catch (Exception e) {
    System.out.println(e.getMessage());
}
```

這些方法都能操作 JSON 資料。使用 JSONObject.toSTring() 方法轉換 JSON 物件到 String 並將 String 寫入。寫入組態檔案等單一 JSON 物件很簡單：

```
JSONObject obj = ...

try(BufferedWriter bw = new BufferedWriter(new FileWriter("somefile.txt")) ) {
    bw.write(obj.toString());
}
} catch (Exception e) {
    System.out.println(e.getMessage());
}
```

建構 JSON 資料檔案（一系列 JSON 物件）時，迭代 JSONObject 集合：

```
List<JSONObject> dataList = ...

try(BufferedWriter bw = new BufferedWriter(new FileWriter("somefile.txt")) ) {
    for(JSONObject obj : dataList){
        bw.write(obj.toString());
        /* 不要忘記加上換行！ */
        bw.newLine();
    }
} catch (Exception e) {
    System.out.println(e.getMessage());
}
```

若檔案是要新增資料則不要忘記設定 FileWriter 的 append ！設定 FileWriter 的 append 可於檔案後面加入更多的 JSON 記錄：

```
try(BufferedWriter bw = new BufferedWriter(
    new FileWriter("somefile.txt", true)) ) {
...
}
```

掌握資料庫操作

MySQL 等關聯式資料庫的堅實與彈性使其成為各種運用的理想技術。作為資料科學家，你很可能與更大的應用程式的關聯式資料庫互動，或者你會產生供資料科學群組使用的資料表。無論是哪一種，掌握命令列、結構化查詢語言（SQL）、與 Java Database Connectivity（JDBC）都是重要的技能。

命令列用戶端

命令列是管理資料庫與執行查詢的好環境。作為一個互動層，此用戶端能夠快速的執行探索命令。從命令列進行查詢後，可以將 SQL 轉入你的 Java 程式，而查詢可參數化以提升使用彈性。MySQL、PostgreSQL、SQLite 等常見資料庫都有命令列用戶端。在安裝 MySQL 以供開發的系統上（例如你的個人電腦），你應該能夠匿名與選擇性提供資料庫名稱連上：

```
bash$ mysql <database>
```

但你或許不能夠建構新的資料庫。你可以登入成資料庫管理員：

```
bash$ mysql -u root <database>
```

然後具有完整的存取權限。在其他狀況下（例如連接到實際資料庫、遠端實例、或雲端實例），則必須如下執行：

```
bash$ mysql -h host -P port -u user -p password <database>
```

連上後會看到 MySQL 提示，可透過它顯示你可以存取的資料庫、目前連接的資料庫、與使用者名稱：

```
mysql> SHOW DATABASES;
```

切換資料庫的命令是 USE *dbname*：

```
mysql> USE myDB;
```

你可以建構新資料表：

```
mysql> CREATE TABLE my_table(id INT PRIMARY KEY, stuff VARCHAR(256));
```

若有儲存在檔案中的資料表建構腳本，下面的命令會讀取並執行該檔案：

```
mysql> SOURCE <filename>;
```

當然，你會想要知道資料庫中有什麼資料表：

```
mysql> SHOW TABLES;
```

也會想要知道資料表的細節，包括欄名稱、資料型別、與限制：

```
mysql> DESCRIBE <tablename>;
```

結構化查詢語言

結構化查詢語言（SQL）是探索資料的工具。雖然物件關聯式對應（ORM）架構在企業應用程式中佔有一席之地，你還是會發現它們對資料科學的工作有很多限制。加強 SQL 技能並學習以下的基本操作是個好主意。

建構

要建構資料庫與資料表，使用下列 SQL：

```
CREATE DATABASE <databasename>;

CREATE TABLE <tablename> ( col1 type, col2 type, ...);
```

選取

普通的 SELECT 陳述具有下列形式：

```
SELECT
    [DISTINCT]
    col_name, col_name, ... col_name
    FROM table_name
    [WHERE where_condition]
    [GROUP BY col_name [ASC | DESC]]
    [HAVING where_condition]
    [ORDER BY col_name [ASC | DESC]]
    [LIMIT row_count OFFSET offset]
    [INTO OUTFILE 'file_name']
```

有一些技巧可能很好用。若資料集有數百萬筆資料，而你只想要取樣，可以使用 ORDER BY 回傳隨機樣本：

```
ORDER BY RAND();
```

並且可以設定 LIMIT 決定回傳樣本大小：

```
ORDER BY RAND() LIMIT 1000;
```

新增

新增資料到新資料列的做法如下：

```
INSERT INTO tablename(col1, col2, ...) VALUES(val1, val2, ...);
```

請注意！若需要所有欄而不是部分欄的值，則可以完全省略欄名：

```
INSERT INTO tablename VALUES(val1, val2, ...);
```

也可以一次新增多筆記錄：

```
INSERT INTO tablename(col1, col2, ...) VALUES(val1, val2, ...),(val1, val2, ...),
(val1, val2, ...);
```

修改

某些情況下，你必須修改現有資料。必須改正錯誤時，通常可以透過命令列快速的執行。雖然你會存取上線、分析、與測試資料庫，但有時候會進入 DBA 狀態。修改資料在處理真正的使用者與資料時很常見：

```
UPDATE table_name SET col_name = 'value' WHERE other_col_name = 'other_val';
```

在資料科學領域中，很難想象你會程式化的修改資料。當然有例外，例如改正打字錯誤或逐步建構資料表，但大部分情況下修改重要資料聽起來像是會引發災難，特別是多個使用者依靠同一個資料來源撰寫程式與進行分析時。

刪除

在儲存體很便宜的今天，刪除資料似乎是不必要的，但如同 UPDATE，犯了錯且不想要重新建構整個資料庫時，刪除資料很方便。通常會根據特定條件刪除資料，例如 user_id、record_id、或特定日期前後：

```
DELETE FROM <tablename> WHERE <col_name> = 'col_value';
```

另一個實用的命令是 TRUNCATE，它刪除資料表中所有資料但保持資料表不變。基本上，TRUNCATE 將資料表清乾淨：

```
TRUNCATE <tablename>;
```

拋棄

若要刪除資料表內容與資料表本身，必須 DROP 資料表。這會完全移除資料表：

```
DROP TABLE <tablename>;
```

這會刪除整個資料庫與其所有內容：

```
DROP DATABASE <databasename>;
```