第七章

最佳化：破解密碼

最佳化導論

目前為止，本書所使用的演算法都算是黑盒子，也就是說，程式的輸入資料與輸出結果都是我們所預期的。本質上，都是將機器學習演算法視為執行既定程序的函式庫。

本章我們即將檢視一些實作機器學習演算法的技巧。一開始，我們會對簡單的線性回歸模型建立一個誤差預測指標，這個範例可以告訴我們如何以最佳化的角度面對模型的擬合。最佳化問題就像是對一個機器進行調校動作，機器有一些設定可供調整，並且具有一個指標可以描述機器運作的效能，如何調整這台機器的效能指標達到最大，此時稱為**最適點**（*optimum*），試圖達到最適點的過程即為**最佳化**。

一旦對最佳化的運作有基本的概念，我們開始著手處理本章的工作：將破解密碼視為最佳化問題，並且建立一個簡易密碼破解系統。

因為我們將建立線性回歸函數，我們以之前所提的身高體重資料作為範例。如先前的做法，我們假設可以藉由身高預測體重，更明確地說，我們假設這是一個線性函數。在 R 語言中，這個函數如以下所示：

```
height.to.weight <- function(height, a, b)
{
  return(a + b * height)
}
```

在第五章裡，我們利用 `lm` 函式計算此函數的斜率與截距，在本例中，`a` 參數代表回歸直線的斜率，而 `b` 參數則是截距，代表的是身高為 0 的人的體重。

一旦定義此函數，我們如何得到最佳的 a 與 b 參數值？這就是最佳化問題所要解決的，我們需要先定義一個量測指標代表 a 與 b 的擬合狀況，接著嘗試調整 a 與 b 參數值，將這個量測指標最大化。

我們該如何進行呢？其實 lm 指令替我們做了所有的事情，它有一個簡單的誤差函數，並且試著將它最佳化，找出最佳的 a 與 b 參數值，其演算法僅使用一般的線性回歸函數。

我們直接執行 lm 指令，並且檢視所得到的最佳 a 與 b 參數值：

```
heights.weights <- read.csv('data/01_heights_weights_genders.csv')

coef(lm(Weight ~ Height, data = heights.weights))
#(Intercept)      Height
#-350.737192    7.717288
```

為什麼這一組 a 與 b 參數值是合理的？要回答這個問題，需要先了解 lm 指令所用的誤差函數，第五章曾簡略提及，lm 指令所用的誤差量測稱為「平方誤差」，其工作原理如下：

1. 針對一組 a 與 b
2. 給定一個身高值並猜測所對應的體重
3. 以真實體重減去猜測的體重，此值即為誤差
4. 將誤差平方
5. 對所有樣本點加總所有的誤差平方

為了便於理解，我們通常使用誤差平方的平均數而非總和，並且計算其平方根，但對於最佳化問題而言，這並不是必要的，我們直接計算總和以節省一些計算時間。

以上最後兩步是緊密相關的，如果我們不進行加總，那麼平方也沒甚麼幫助。平方的動作很重要，因為這讓結果不會互相抵銷而得到零。

要證明這一點並不難，不過需要一些代數運算，這不在本書涵蓋範圍。

我們以程式實現這個過程：

```
squared.error <- function(heights.weights, a, b)
{
  predictions <- with(heights.weights, height.to.weight(Height, a, b))
  errors <- with(heights.weights, Weight - predictions)
  return(sum(errors ^ 2))
}
```

為了了解其工作原理，我們針對幾組 a 與 b 參數值，計算其 squared.error 平方誤差值，如表 7-1：

```
for (a in seq(-1, 1, by = 1))
{
  for (b in seq(-1, 1, by = 1))
  {
    print(squared.error(heights.weights, a, b))
  }
}
```

表 7-1　各格點平方誤差

a	b	Squared error
-1	-1	536271759
-1	0	274177183
-1	1	100471706
0	-1	531705601
0	0	270938376
0	1	98560250
1	-1	527159442
1	0	267719569
1	1	96668794

如表 7-1 所示，有幾組 a 與 b 數值可以得到較小的平方誤差，這表示這個指標是一個有意義的誤差函數，可以用來找出最佳的 a 與 b 數值。這便是最佳化的第一步：設定一個指標，然後試著將其最大化或最小化。這個量測指標稱為**目標函數**（*objective function*）。這個最佳化問題變成尋找最佳的 a 與 b 數值，使得目標函數最大化或最小化。

一個最簡單的方法稱為格點搜尋法：如表 7-1 方式，計算一個適當範圍內的 a 與 b 數值，並且挑出具有最小的 squared.error 值。這個方法可以找出格點中的最佳值，所以還算可行的方法，不過卻有一些嚴重的問題：

- 這些格點應該距離多近？ a 的值應該是 0 至 3 抑或是 0 至 0.003 ？換而言之，最佳值搜尋的格點解析度應該設為多少？要回答這些問題需要實際對這些格點求值才能得知，這麼做會花費大量的計算時間，而且這又引發另一個最佳化問題：格點尺寸應設定為多少。這是一個惡性循環。
- 如果你要對兩個參數分別計算 10 個格點，則你需要求值的參數組合為 10^2=100，那如果你有 100 個參數的話，參數組合便有 10^100 個。參數組合隨著維度的增加式指數成長，這在機器學習領域十分常見，稱為「維數災難（Curse of Dimensionality）」。

因為我們想對數百至數千個輸入參數進行線性回歸，格點搜尋不是一個可行的最佳化演算法。那我們該怎麼做呢？幸好，科學家對最佳化問題已經研究了很久，提供了不少可行的演算法供我們使用。在 R 語言中，遇到最佳化問題可以先嘗試使用 optim 指令，它內建了許多常見的最佳化演算法。

為了介紹 optim 指令如何使用，我們將利用它進行線性回歸模型的擬合。我們預期它得到的 a 與 b 參數結果與 lm 指令的結果差不多。

```
optim(c(0, 0),
      function (x)
      {
        squared.error(heights.weights, x[1], x[2])
      })
#$par
#[1] -350.786736    7.718158
#
#$value
#[1] 1492936
#
#$counts
#function gradient
#     111       NA
#
#$convergence
#[1] 0
#
#$message
#NULL
```

如上式所示，optim 指令具有幾個引數。首先，需要傳入一參數向量作為起始猜測點；在本範例中，我們將 a 與 b 的起始猜測設為 c(0, 0)。接著，傳入一個引數為參數向量的函數，原則上這個函數就是要進行最佳化的目標，不過我們以匿名函數將之封裝，這是因為要進行最佳化的函數通常具有數個引數，但並非每個引數都需要進行最佳化。上式可以看到我們如何使用匿名函數包裝 squared.error 函數。

執行 optim 指令將回傳 par 向量，內含 a 與 b 的最佳值。這個結果與 lm 指令的輸出非常接近，這表示 optim 指令是正常作用的[1]。實作上 lm 指令使用專門用於線性回歸的演算法，如果你使用的是線性模型，則 lm 指令會比較準確；若非線性模型，optim 是較好的選擇。

optim 指令的其他輸出則較為其次。value 值告訴我們最佳化的函數值為何，對本例而言則是平方誤差總和。counts 值則代表 optim 指令執行此函數的次數（即上式 function 值）以及計算梯度的次數（即上式 gradient 值），梯度是 optim 指令的非必要引數，如果你懂得以微積分計算目標函數的梯度，那麼也可以傳入。

如果你對「梯度」這個名詞感到很陌生，別擔心！我們通常不會手動計算梯度，因此通常可以讓 optim 指令自動幫我們計算。在這個例子沒甚麼問題，不過不同情況下就很難說。

接著是 convergence 值，它告訴我們 optim 指令是否找到最佳解。如果沒特別問題的話，其值為 0。你可以在說明文件中找到不同 convergence 代碼所代表的意義。最後，message 則輸出警告訊息。

總之，optim 指令以微積分相關的數學方法執行最佳化，其數學推導已超出本書範圍，我們不深入探討其原理。不過 optim 的精神可以簡單地圖形化解釋。想像一下，如果固定 b 的值為 0，你想找出最佳的 a 值為何，你可以只改變 a 值計算下式平方差：

```
a.error <- function(a)
{
  return(squared.error(heights.weights, a, 0))
}
```

1 或是跟 lm 一樣失敗。

想知道最佳的 a 值在哪裡，可使用 R 語言中 curve 指令繪出平方誤差對 a 值的曲線。以下範例中，我們繪出 a.error 對不同 x 值的曲線，使用 sapply 指令是因為這是 R 語言表示式求值的行為。

```
curve(sapply(x, function (a) {a.error(a)}), from = -1000, to = 1000)
```

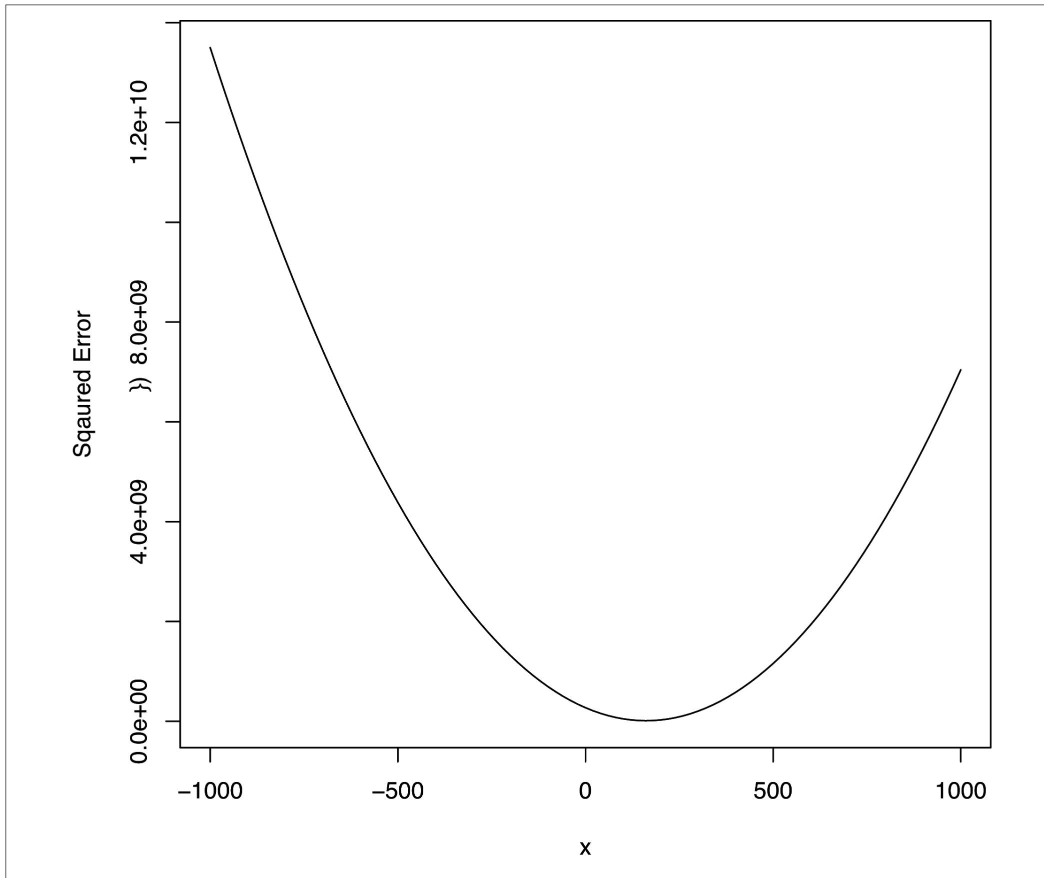


圖 7-1　平方誤差隨著 a 的改變

如圖 7-1 所示，有某一個特定的 a 值可使得平方誤差為最小值，比此數值大或小的 a 值都使得平方誤差變得更糟，此情形下，我們稱此問題具有**全域最適點**（*global optimum*）；此時，optim 指令將利用曲線的形狀判斷 a 值的方向為何；以此方法可以迅速地找出最適點。

要了解完整的回歸問題解法，我們還需要檢視當我們改變 b 值時，誤差函數將如何改變。

```
b.error <- function(b)
{
  return(squared.error(heights.weights, 0, b))
}

curve(sapply(x, function (b) {b.error(b)}), from = -1000, to = 1000)
```

如圖 7-2 所示，b 參數也有全域最適點，再加上 a 參數也具有全域最適點，則 optim 指令應該可以找到一個 a 與 b 參數組合使得誤差函數為最小值。

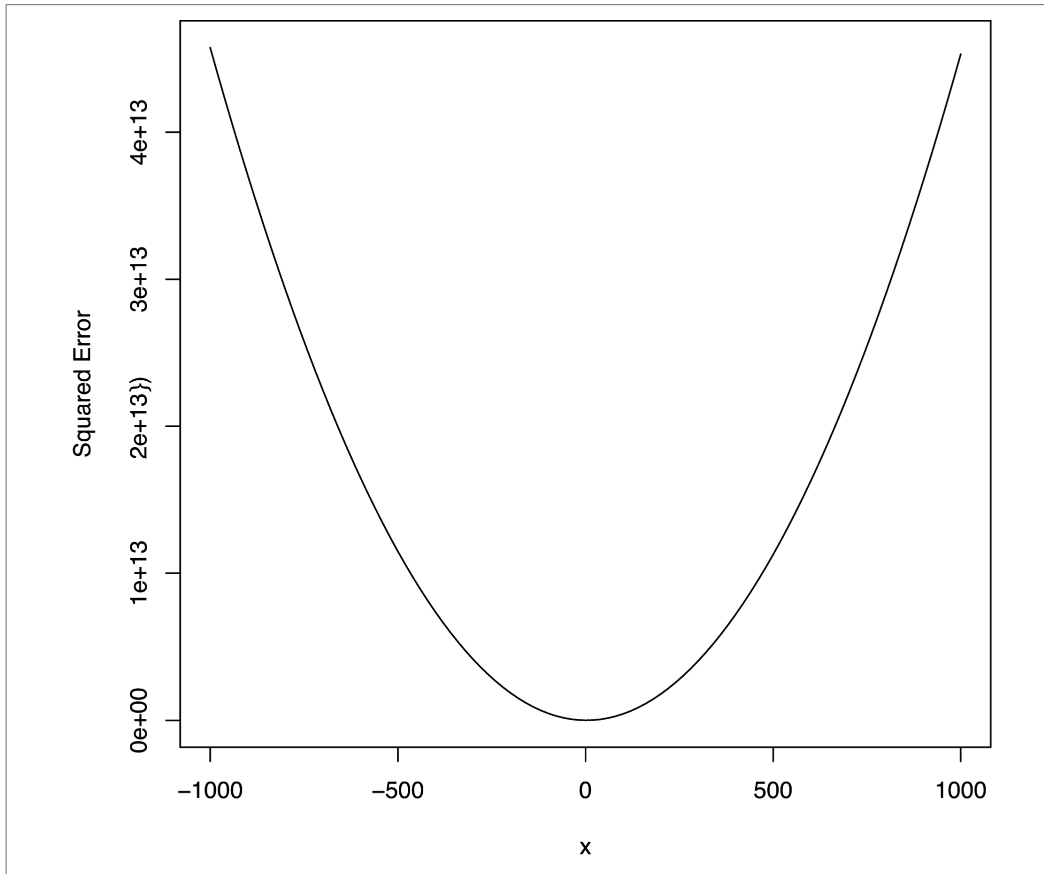


圖 7-2　平方誤差隨著 b 的改變

一般來說，optim 指令可以使用微積分同時對許多參數尋找最適點，它比格點搜尋法快得多，這是因為它能夠考慮鄰近點的資訊（即斜率），此資訊能決定最適點的搜尋方向，使得這個方法能夠比格點搜尋法更有效率。

山脊型回歸（Ridge Regression）

optim 指令的用法我們已經大致了解了，接下來我們可以使用此指令實現山脊型回歸。山脊型回歸是一種用來處理正則化的回歸種類，正則化我們已於第六章討論。山脊型回歸與一般的最小平方法回歸唯一的差別在於誤差函數，山脊型回歸的誤差函數額外考慮參數的大小，意即山脊型回歸將傾向於較小的參數值。以本範例來說，這將導致斜率與截距均傾向於零。

除了誤差函數考慮參數大小外，還有一個新增的複雜度，那便是參數 lambda，用來設定參數大小於誤差函數內的係數，以避免過度擬合的情況發生，這個額外的參數在正則化演算法中稱為超參數，第六章中有更深入的討論。在給定 lambda 值的情況下，山脊型回歸的誤差函數如下式表示：

```
ridge.error <- function(heights.weights, a, b, lambda)
{
  predictions <- with(heights.weights, height.to.weight(Height, a, b))
  errors <- with(heights.weights, Weight - predictions)
  return(sum(errors ^ 2) + lambda * (a ^ 2 + b ^ 2))
}
```

如同第六章的討論，我們可使用交叉驗證的方式選擇 lambda 值。本章接下來的部分，我們僅假設已經做完這個步驟，並且選擇 lambda 值為 1。

定義了山脊型回歸的誤差函數之後，我們重新呼叫 optim 指令，對此誤差函數最佳化，執行方式與一般最小平方法相同：

```
lambda <- 1

optim(c(0, 0),
      function (x)
      {
        ridge.error(heights.weights, x[1], x[2], lambda)
      })
#$par
#[1] -340.434108    7.562524
#
#$value
```

```
#[1] 1612443
#
#$counts
#function gradient
#     115       NA
#
#$convergence
#[1] 0
#
#$message
#NULL
```

從 optim 指令的輸出可以發現，這個結果比 lm 指令的結果還要再小一些，lm 輸出截距為 -350，而斜率為 7.7。這個簡易的範例可能還看不出來，但是對第六章中執行的多參數大型回歸分析來說，山脊型回歸會得到較有意義的結果。

除了檢查擬合係數之外，我們也可以重複之前的作圖，使用 curve 指令檢查為什麼 optim 指令也適用於山脊型回歸，結果如圖 7-3 與 7-4 所示。

```
a.ridge.error <- function(a, lambda)
{
  return(ridge.error(heights.weights, a, 0, lambda))
}

curve(sapply(x, function (a) {a.ridge.error(a, lambda)}), from = -1000, to = 1000)

b.ridge.error <- function(b, lambda)
{
  return(ridge.error(heights.weights, 0, b, lambda))
}

curve(sapply(x, function (b) {b.ridge.error(b, lambda)}), from = -1000, to = 1000)
```

希望這個範例能夠說服你只要能了解如何使用 optim 指令降低，預測誤差函數，你便能夠進行許多機器學習的演算法。我們建議你進行一些練習，並且嘗試不同的誤差函數，這會對你極有幫助。如果你嘗試建立絕對值誤差函數，函式可能如下所示：

```
absolute.error <- function
(heights.weights, a, b)
{
  predictions <- with(heights.weights, height.to.weight(Height, a, b))
  errors <- with(heights.weights, Weight - predictions)
  return(sum(abs(errors)))
}
```

由於一些數學上的理由，這個誤差函數在 optim 指令中並沒有辦法運作得很好，要解釋原因需要較複雜的微積分證明，不過如果繪出圖來會變好理解的，我們再次使用 curve 指令作圖：

```
a.absolute.error <- function(a)
{
  return(absolute.error(heights.weights, a, 0))
}

curve(sapply(x, function (a) {a.absolute.error(a)}), from = -1000, to = 1000)
```

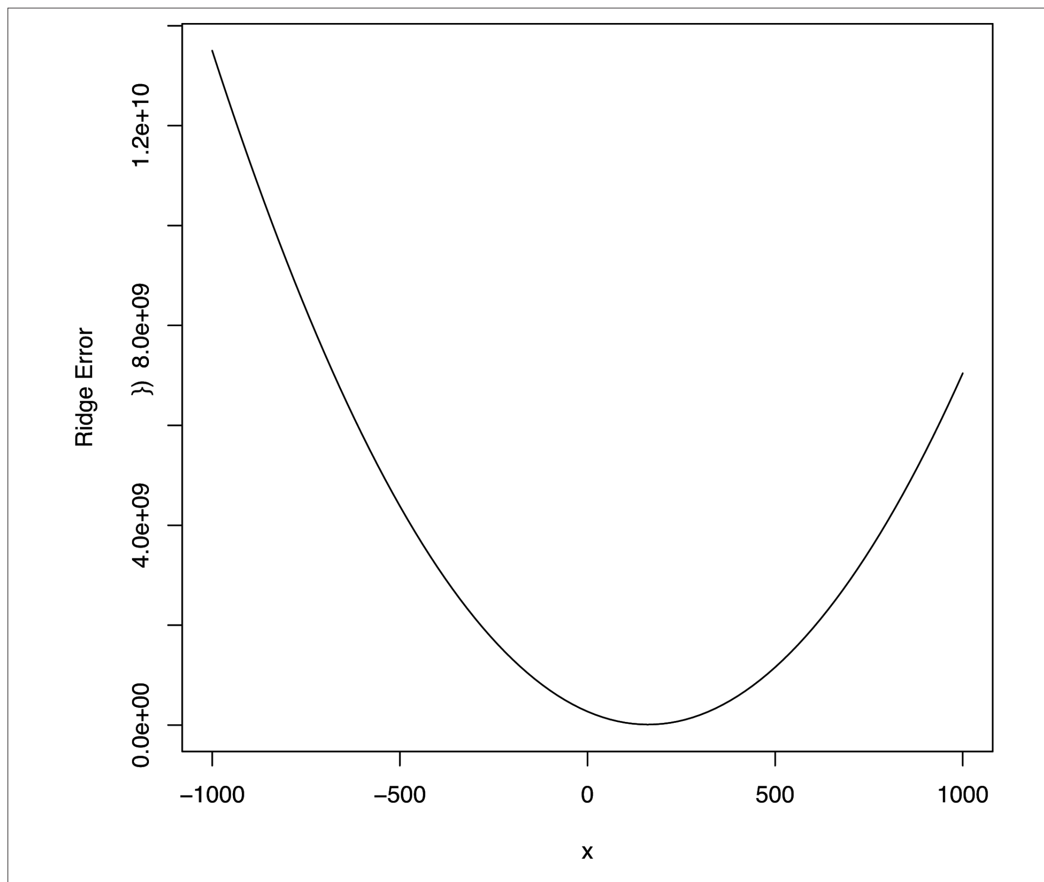


圖 7-3　回歸誤差隨著 a 的改變

如圖 7-5 所示，絕對值誤差曲線比平方誤差或山脊型誤差要尖銳得多，因此 `optim` 無法得到足夠資訊預測最適點的方向，即便圖形上清楚地顯示全域最適點的位置。

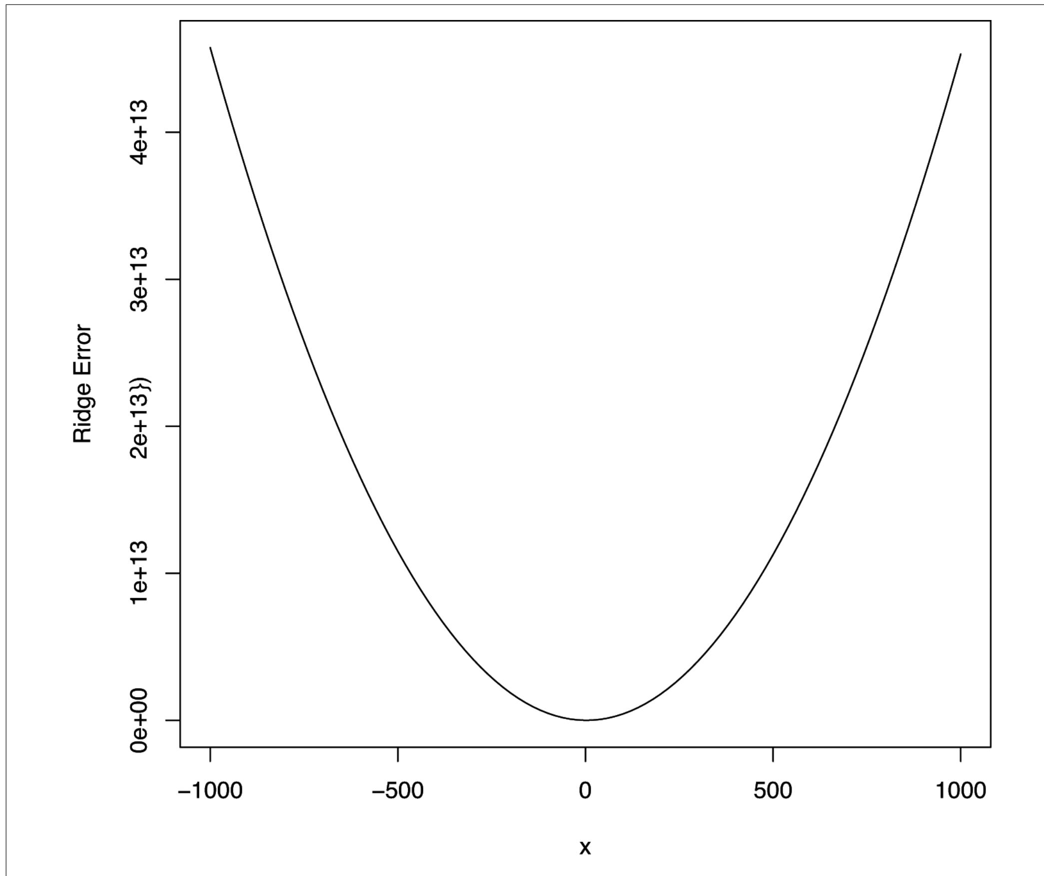


圖 7-4　回歸誤差隨著 b 的改變

有些強大的演算法並不適用特定型態的誤差函數，因此實作機器學習的精髓之一便是學習：甚麼時候用 `optim` 這種簡單的工具便足夠了，而甚麼時候需要用更強大的工具。事實上，有些演算法可用於絕對值誤差函數的最佳化，但這已超出本書範圍，如果你對這個課題感到興趣，可以請教你身邊的數學高手關於「凸面最佳化（convex optimization）」的資訊。

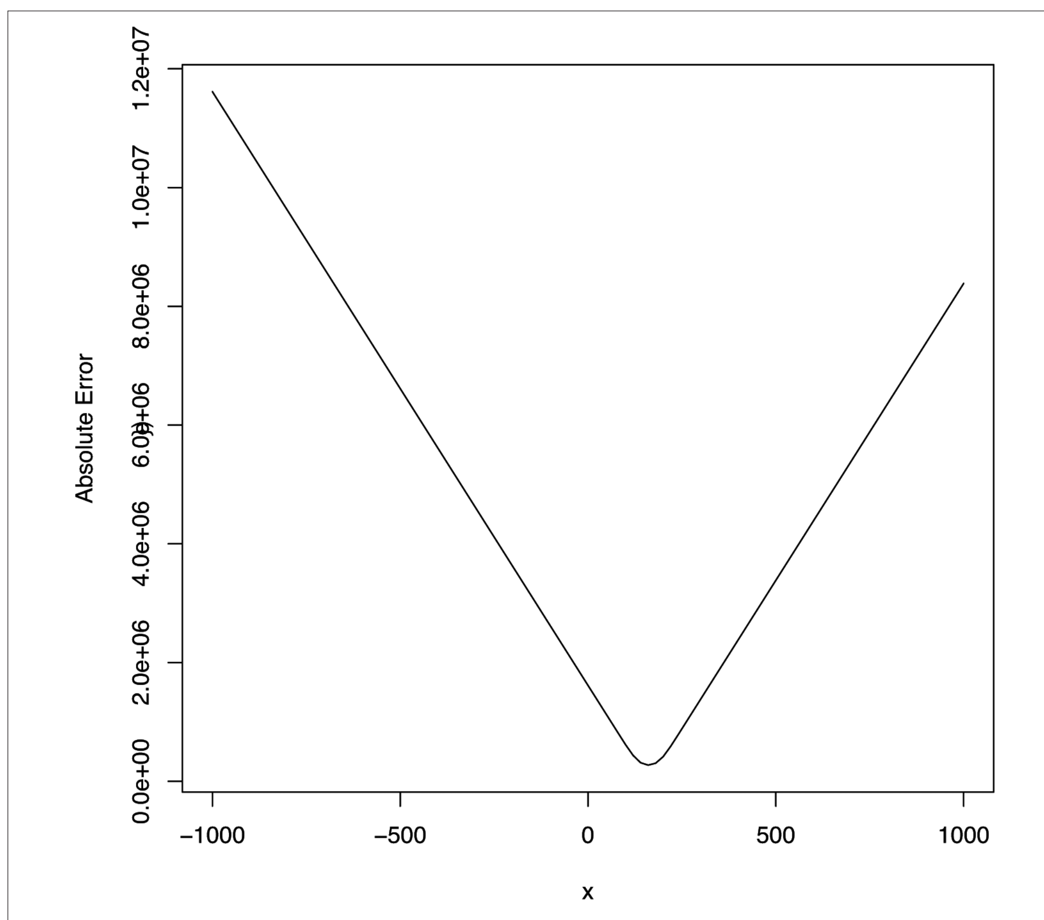


圖 7-5　絕對誤差隨著 a 的改變

將破解密碼視為最佳化問題

幾乎所有機器學習領域的演算法均可視為最佳化問題，目標都是將某預測誤差測度最小化。但有時候參數並不是簡單的數值，因此使用 `optim` 指令時，誤差函數並沒有辦法為一特定點提供足夠的鄰近點資訊（如斜率等），對這類問題，我們還有格點搜尋法可以使用，不過實際上還有更好的方法，我們將討論其中的一個直覺且強大的方法，此方法所使用的概念稱為隨機最佳化（stochastic optimization），其核心想法是在搜尋下一個參數組時加入些許隨機的成分， 重點是必須朝誤差函數減少的方向。這個方法與許多常見

的最佳化演算法有關聯，像是模擬退火演算法、基因演算法以及馬可夫鏈蒙地卡羅演算法（Markov chain Monte Carlo，MCMC）等，這些演算法你或許也曾耳聞。我們將使用的演算法稱為 Metropolis 演算法【譯註：這是為了紀念物理學家 Nicholas Metropolis】，這個方法的許多變形版本是當代的機器學習演算法的基礎。

我們將以本章主題密碼破解介紹 Metropolis 演算法， 使用此演算法進行密碼破解並非最有效的系統，而且真實世界的破解系統可能永遠不會選擇這個方法實作，不過這個範例能夠很清楚地展示如何使用 Metropolis 演算法；另外同樣重要的是，這個範例也告訴我們有些問題是無法透過一般的最佳化演算法（如 `optim` 指令）解決的。

讓我們定義清楚目前的問題：已知一字串，其使用替換式密碼（substitution cipher）進行加密，如何找出替換規則將該密文轉換回原文？替換式密碼是最簡單的加密系統，透過將原文中的每個字母替換為另一個字母轉換為密文。其中最著名的範例應是 ROT13[2]，或許你曾聽過凱撒密碼，這也是一個範例，它將原文的每一字母替換為下一個字母：「a」替換為「b」、「b」替換為「c」、「c」替換為「d」，以此類推，「z」則替換為「a」。

為了讓你清楚 R 語言中如何實作替換式密碼加密，我們先實作凱撒密碼：

```
english.letters <- c('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
                     'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v',
                     'w', 'x', 'y', 'z')

caesar.cipher <- list()
inverse.caesar.cipher <- list()

for (index in 1:length(english.letters))
{
  caesar.cipher[[english.letters[index]]] <- english.letters[index %% 26 + 1]
  inverse.caesar.cipher[[english.letters[index %% 26 + 1]]] <- english.letters[index]
}

print(caesar.cipher)
```

2　ROT13 將每個字母替換為該字母之後的第 13 個字母。「a」替換為「n」、「b」替換為「o」，以此類推。

實作加密規則之後，接著建立函式將原文加密：

```
apply.cipher.to.string <- function(string, cipher)
{
  output <- ''

  for (i in 1:nchar(string))
  {
    output <- paste(output, cipher[[substr(string, i, i)]], sep = '')
  }

  return(output)
}

apply.cipher.to.text <- function(text, cipher)
{
  output <- c()

  for (string in text)
  {
    output <- c(output, apply.cipher.to.string(string, cipher))
  }

  return(output)
}

apply.cipher.to.text(c('sample', 'text'), caesar.cipher)
```

準備好基本加密工具後，我們開始思考破解密碼可能遇到的問題。如同之前線性回歸的處理一樣，我們將破解替換式密碼分解為幾個步驟：

1. 基於某解密規則，定義其解密品質【譯註：即前文所述之誤差函數】。
2. 基於某解密規則，設計演算法隨機建立新解密規則。
3. 基於某解密規則，設計演算法搜尋下一個更佳的解密規則。

如何定義解密規則的品質呢？先假設我們已經擁有一段以替換式加密的密文。例如，凱撒可能傳這個加密訊息給你：「wfoj wjej wjdj」，以凱撒加密法解密後便是他的名言：「veni vidi vici」。【譯註：拉丁文「我來，我見，我征服」】

接著，我們假設所收到的密文是以標準英文撰寫的，那麼該如何對其解密呢？

所使用的解法，是將密文替換為標準英文的解密規則定義為好的規則。當給定一解密規則時，以此規則對密文解密並且觀察結果是否為正確的英文。舉例來說，假設我們有兩個解密規則 A 與 B，分別對前文的密文解密，所得如下：

- decrypt(T, A) = xgpk xkfk xkek
- decrypt(T, B) = veni vidi vici

看到這個結果後，很明顯地，規則 B 比規則 A 好多了！我們是如何做出此判斷的呢？因為 B 的結果比 A 看起來更像真實語言，而非不知所云。我們需要將這個直覺轉換為電腦程式的運算，這時需要一個語彙資料庫，提供所有的英文單字的出現機率，真實語言所使用的單字幾乎都是高機率單字，而假語言則充滿了低機率單字。剩下的複雜度便是處理不存在於語彙資料庫中的單字，其機率為零，但是我們又想要將所有的單字機率相乘，因此需要以一個極小的數目取代其零機率用於我們的演算法中，例如機器的最小浮點數，我們稱為 epsilon。處理完這個極端狀況後，便能夠以語彙資料庫評估兩段解密結果何者較佳：找出解密結果每個單字的出現機率，將之相乘變得出該結果於標準英文的出現機率，並以此衡量解密規則的優劣。

我們以語彙資料庫計算解密文的機率作為我們評估解密規則的誤差函數，因此這解密問題便轉化為最佳化問題了，我們只要找出解密結果機率最高的解密規則便成功了！

不幸的是，這個找出最高機率的解密規則演算法並無法使用 `optim` 指令直接解決，原因是誤差函數無法進行參數繪圖，而且也不具平滑性，所以 `optim` 指令無法朝向最適點方向前進。所以我們需要全新的最佳化演算法解決解密問題，這個方法便是前文所提的 Metropolis 演算法，此演算法對此問題是可行的，不過對一般長度的密文而言，它的計算速度十分緩慢[3]。

Metropolis 演算法的基本想法是，先從一個隨機決定的解密規則出發，經過反覆的改良，便能夠得到正確的解密規則。乍聽之下像是變魔術一般，不過實際上是可行的，你可以做個簡單的實驗驗證這個想法。一旦找出最佳的解密規則後，我們可以用直覺判斷前後文關聯性以及文法是否正確，如果正確，那麼這便是正確的解密規則。

---

3 不像 Perl 等專門處理字串的程式語言，R 語言字串處理算是慢的，這使得 Metropolis 演算法的低效能更加嚴重。

要產生好的解密規則，我們先隨機挑選一解密規則，並且重複地改善它，例如執行 50,000 次。因為每一步都朝向更好的方向前進，或許最後我們可以得到合理的結果，雖然說我們也不確定究竟需要 50,000 次或是 50,000,000 次才能得到正確的結果，這便是 Metropolis 演算法無法用於真實密碼破解系統的原因：你不確定這個演算法是否能在合理的時間內找到解答，而且你甚至無法判斷演算法是否朝向正確的方向找尋解答。本案例只能說是一個很簡單的範例，目的是示範如何使用最佳化演算法解決難解的複雜問題。

接下來讓我們仔細討論如何從目前的替換規則出發，找到下一個新的規則。我們以隨機方式改變目前規則的一處字母替換規則，以得到新的規則；假設說目前規則中「a」替換為「b」，那我們就隨機改成替換為「q」，不過因為原規則中還有另一個規則將另一個字母替換為「q」，假設此字母為「c」，那麼新規則中還需要一併修改「c」替換為「b」。因此每一次更新替換規則，都需要交換兩處字母的替換，其中一個是隨機選擇的，另一個則是因應此隨機更改所做的變更。

直觀來說，只有當新規則的標準英文機率提升時，我們才接受這個新的解密規則，此策略稱為貪婪最佳化（greedy optimization）。不幸地，此範例使用貪婪最佳化並無法為我們帶來最佳的替換規則，可能會陷入糟糕的替換規則中，所以我們將採用以下非貪婪規則，以決定隨機產生的新規則 B 是否被採納並取代原規則 A：

1. 如果 B 規則所解密的標準英文機率比 A 規則高，則以 B 取代 A。
2. 如果 B 規則所解密的標準英文機率比 A 規則低，我們偶而決定以 B 取代 A，但並非每次都這麼做。明確地說，如果 A 與 B 的機率分別是 `probability(T, A)` 與 `probability(T, B)`，那麼決定以 B 取代 A 的機率是 `probability(T, B) / probability(T, A)`。

如果你覺得這個取代機率有點莫名其妙，別擔心，重要的其實只是我們偶而接受較差的 B 規則，這可以避免陷入貪婪最佳化導致的糟糕規則中。

在開始使用 Metropolis 方法前，還需要建立一些隨機更改替換規則的函式，以下是其程式碼：

```
generate.random.cipher <- function()
{
  cipher <- list()

  inputs <- english.letters
  outputs <- english.letters[sample(1:length(english.letters),
```

```
  length(english.letters))]

  for (index in 1:length(english.letters))
  {
    cipher[[inputs[index]]] <- outputs[index]
  }

  return(cipher)
}

modify.cipher <- function(cipher, input, output)
{
  new.cipher <- cipher
  new.cipher[[input]] <- output
  old.output <- cipher[[input]]
  collateral.input <- names(which(sapply(names(cipher),
                      function (key) {cipher[[key]]}) == output))
  new.cipher[[collateral.input]] <- old.output
  return(new.cipher)
}

propose.modified.cipher <- function(cipher)
{
  input <- sample(names(cipher), 1)
  output <- sample(english.letters, 1)
  return(modify.cipher(cipher, input, output))
}
```

結合使用此產生新規則的函式與交換規則的程序，便可以減輕最佳化的貪婪程度，避免浪費時間陷於明顯糟糕的規則中。演算法如下：先計算 `probability(T, B) / probability(T, A)` 之數值，接著與一介於 0 與 1 之間的亂數進行比較，如果亂數的數值較高，我們便以 B 取代 A，反之則繼續保留 A 替換規則。

為了計算之前一直提及的標準英文機率，我們根據檔案 */usr/share/dic/words* 建立了語彙資料庫，此檔案記錄了維基百科所用單字的出現機率。將其載入 R 語言的程式碼如下：

```
load('data/lexical_database.Rdata')
```

你可以試著對此資料庫檢索一些簡單的單字（見表 7-2）：

```
lexical.database[['a']]
lexical.database[['the']]
lexical.database[['he']]
lexical.database[['she']]
lexical.database[['data']]
```

表 7-2　語彙資料庫

單字	出現機率
a	0.01617576
the	0.05278924
he	0.003205034
she	0.0007412179
data	0.0002168354

語彙資料庫準備好之後，我們接著需要計算文本的出現機率。我們先撰寫一函式讀取由資料庫中所取得的機率，以此函式可較容易地處理偽單字。偽單字的出現機率我們設定為機器最小浮點數，在 R 語言中，此數值可由系統變數 `.Machine$double.eps` 取得。

```
one.gram.probability <- function(one.gram, lexical.database = list())
{
  lexical.probability <- lexical.database[[one.gram]]

  if (is.null(lexical.probability) || is.na(lexical.probability))
  {
    return(.Machine$double.eps)
  }
  else
  {
    return(lexical.probability)
  }
}
```

找出個別單字機率的函式撰寫好之後，我們再建立一個函式將文本的每個單字拆開，分別計算其出現機率，再全部乘起來以取得文本整體的出現機率。不幸的是，直接將所有數值相乘將導致不穩定的數值輸出，因為浮點數的精確度是有極限的，因此，我們實際上是取個別機率的對數值，並且全部加總，此動作等同於原始機率相乘，這個方法將使函式的數值輸出較為穩定。

```
log.probability.of.text <- function(text, cipher, lexical.database = list())
{
  log.probability <- 0.0

  for (string in text)
  {
    decrypted.string <- apply.cipher.to.string(string, cipher)
    log.probability <- log.probability +
    log(one.gram.probability(decrypted.string, lexical.database))
  }
```

```
  return(log.probability)
}
```

Metropolis 方法所需的工具都有了，以下將其組合為單一函式：

```
metropolis.step <- function(text, cipher, lexical.database = list())
{
  proposed.cipher <- propose.modified.cipher(cipher)

  lp1 <- log.probability.of.text(text, cipher, lexical.database)
  lp2 <- log.probability.of.text(text, proposed.cipher, lexical.database)

  if (lp2 > lp1)
  {
    return(proposed.cipher)
  }
  else
  {
    a <- exp(lp2 - lp1)
    x <- runif(1)
    if (x < a)
    {
      return(proposed.cipher)
    }
    else
    {
      return(cipher)
    }
  }
}
```

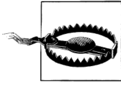
我們最佳化演算法的每個步驟都已經具備對應的函式，讓我們將其組合起來作為範例，我們先設定加密前的原文至一個字元向量中：

```
decrypted.text <- c('here', 'is', 'some', 'sample', 'text')
```

接著以凱撒法進行加密：

```
encrypted.text <- apply.cipher.to.text(decrypted.text, caesar.cipher)
```

接下來我們先隨機建立一個解密規則，並且以 Metropolis 方法重複進行 50,000 次，結果儲存至 `results` 變數中。運算過程中的每一步內容將記錄下來，包含該次運算解密文的出現機率對數值、解密後的文本以及記錄解密是否成功的變數。



當然，在真實破解密碼的案例中，你是無法得知解密是否成功，但是在這個示範性質的案例中，這個變數還挺好用的。

```
set.seed(1)
cipher <- generate.random.cipher()

results <- data.frame()

number.of.iterations <- 50000

for (iteration in 1:number.of.iterations)
{
  log.probability <- log.probability.of.text(encrypted.text, cipher, lexical.database)
  current.decrypted.text <- paste(apply.cipher.to.text(encrypted.text, cipher),
                                  collapse = ' ')
  correct.text <- as.numeric(current.decrypted.text == paste(decrypted.text,
                                                             collapse = ' '))
  results <- rbind(results,
                   data.frame(Iteration = iteration,
                              LogProbability = log.probability,
                              CurrentDecryptedText = current.decrypted.text,
                              CorrectText = correct.text))
  cipher <- metropolis.step(encrypted.text, cipher, lexical.database)
}

write.table(results, file = 'data/results.csv', row.names = FALSE, sep = '\t')
```

以上的計算將花上一些時間，在你等待計算完成的同時，先來看一下作者執行的結果，如表 7-3。

表 7-3　Metropolis 方法的運算過程

執行次數	解密文本機率的對數值	該次運算的解密文本
1	-180.218266945586	lsps bk kfvs kjvhys zsrz
5000	-67.6077693543898	gene is same sfmpwe text
10000	-67.6077693543898	gene is same spmzoe text
15000	-66.7799669880591	gene is some scmhbe text
20000	-70.8114316132189	dene as some scmire text
25000	-39.8590155606438	gene as some simple text
30000	-39.8590155606438	gene as some simple text
35000	-39.8590155606438	gene as some simple text
40000	-35.784429416419	were as some simple text

執行次數	解密文本機率的對數值	該次運算的解密文本
45000	-37.0128944882928	were is some sample text
50000	-35.784429416419	were as some simple text

如表所示，在運行 45,000 次後，離正確的原文已相差不遠，但還差那麼一點。如果你很仔細地檢視每一步過程，你會發現在第 45,609 次計算中已經解密為正確原文，但演算法還不滿意，繼續找尋更好的替換規則。這正是我們目標函數的致命缺點：我們只要每個單字都是正確英文即可，並不需要找到最標準的英文文本。只要改變替換規則可以將個別單字解密為更常見的單字，演算法就會往那個方向前進，而無視此文本並不符合文法或前後文。要避開這個缺點，你可以引入更多標準英文的資訊，例如考慮連續兩個單字一起出現的機率。不過到目前為止，至少我們點出了最佳化演算法中設定目標函數的複雜度，我們經常遇到演算法所得到的最佳結果，並不是我們所認知的最佳結果，所以人為的判斷對最佳化演算法還是必須的。

事實上，我們所面臨的問題比目標函數的英文程度不足還更嚴重。首先，Metropolis 方法是一個隨機找尋最佳化的方法，我們這次運氣不錯，可以成功找到不錯的結果，但下次可能跑個幾億次才能達到這個結果，你自己可以試著以不同亂數種子跑個 1000 次並觀察每一步的過程。

其次，Metropolis 方法容易偏離較佳的結果，這是非貪婪演算法的本質，所以只要你等待夠長的時間，它就會放棄你最想要的結果。圖 7-6 顯示機率對數值對運算次數作圖，你可以看出其過程是飄忽不定的。

目前有些常見的演算法對這個隨機過程進行改良，其中一個方法是隨著運行次數的增加，逐漸增加演算法貪婪程度，意即演算法愈來愈不能接受較差的結果，這稱為**模擬退火演算法**，這是非常強大的方法，你可以嘗試以此方法修改本破解密碼範例，只要改變接受新解密規則的部分即可[4]。

另一個方法是接受這個隨機性，並且以此獲得可能結果的機率分布，而非單一結果。不過本範例並不適用，當最佳化結果是一個數值時，這個方法可以產生各種可能的答案。

4 事實上，optim 指令有一個引數可以指定進行退火演算法，但是本範例無法直接使用，因為退火演算法僅適用於數值參數，本範例的解密規則所用資料結構並不適用。

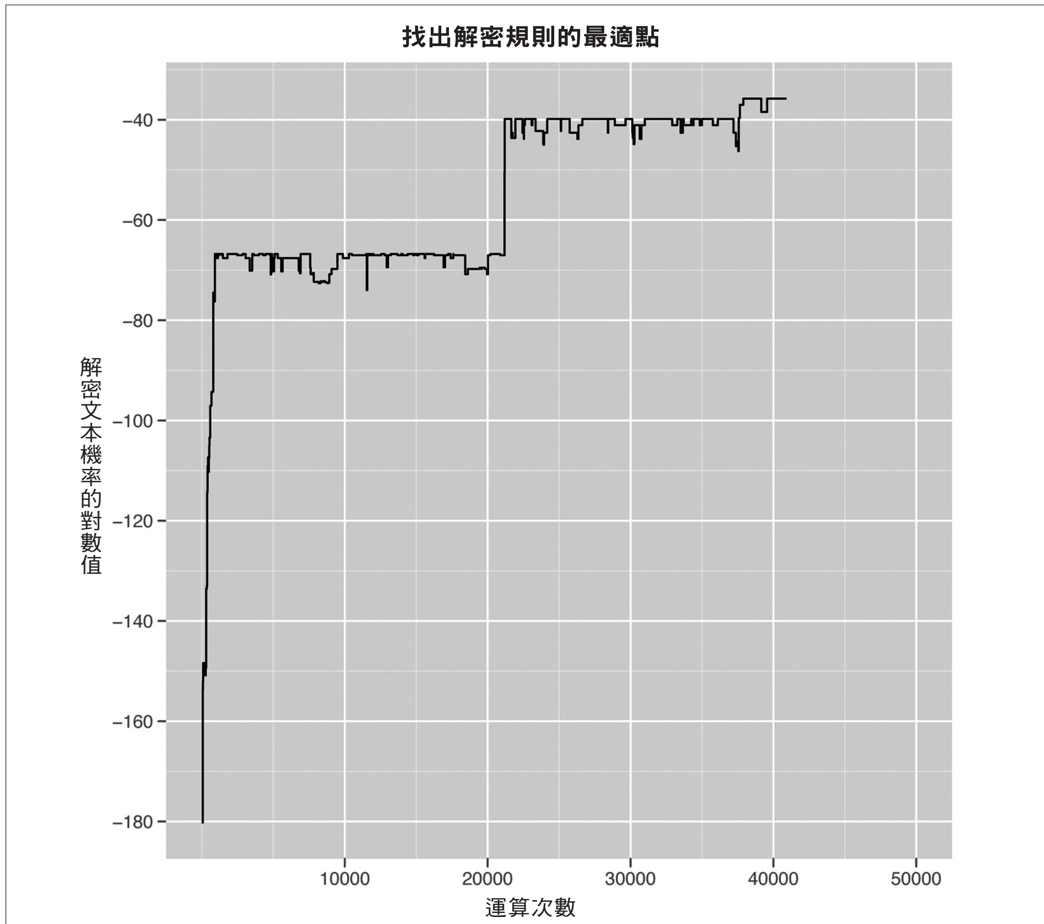


圖 7-6　Metropolis 方法運算過程：找出解密規則的最適點

本章的最後，我們希望你能夠對最佳化有所了解，並且能以此解決機器學習領域的複雜問題，在本書往後的章節，我們會繼續使用這些概念，特別是推薦系統的案例討論。