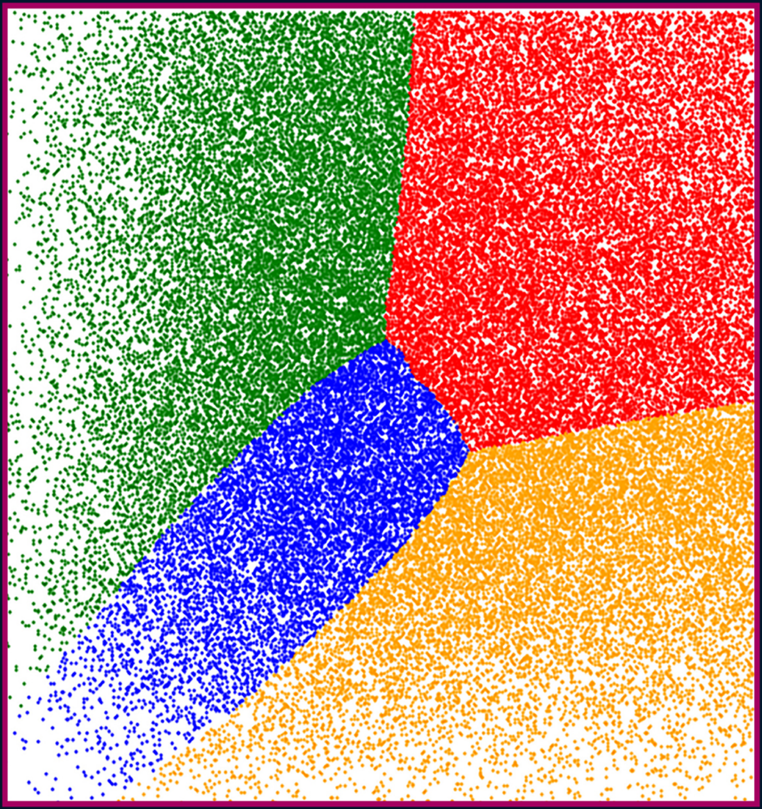
DATA SCIENCE SERIES

A TOUR OF DATA SCIENCE

Learn R and Python in Parallel



NAILONG ZHANG

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

A Tour of Data Science

A Tour of Data Science

Learn R and Python in Parallel

Nailong Zhang

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

First edition published 2021
by CRC Press
6000 Broken Sound Parkway NW, Suite 300, Boca Raton, FL 33487-2742

and by CRC Press
2 Park Square, Milton Park, Abingdon, Oxon, OX14 4RN

Library of Congress Cataloging-in-Publication Data

Names: Zhang, Nailong, author.
Title: A tour of data science : learn R and Python in parallel / Nailong Zhang.
Description: First edition. | Abingdon, Oxon ; Boca Raton, FL : CRC Press, 2021. | Series: Chapman & Hall/CRC data science series | Includes bibliographical references and index.
Identifiers: LCCN 2020025166 | ISBN 9780367895860 (paperback) | ISBN 9780367897062 (hardback) | ISBN 9781003020646 (ebook)
Subjects: LCSH: Data mining. | R (Computer program language) | Python (Computer programlanguage)
Classification: LCC QA76.9.D343 Z458 2021 | DDC 006.3/12--dc23
LC record available at https://lccn.loc.gov/2020025166

ISBN: 9780367895860 (hbk)

Typeset in Computer Modern font
by Cenveo Publisher Services

To my parents

Contents

Preface

0.1 ASSUMPTIONS ABOUT THE READER'S BACKGROUND

It is hard to give a clear definition of data science because it is not clear where the border is. Actually, many data scientists are working in different industries with different skill sets. Generally speaking, data science requires broad knowledge in statistics, machine learning, optimization and programming. It is impossible to cover even one of these four areas in depth in a short book, and it is also beyond the scope of this book. However, I think it might be useful to have a book that talks about data science with a broad range of topics and a moderate amount of technical detail. And that is one of the motivations for writing this book. I chose to make the book short with minimum mathematical theories introduced, and hope reading through the book could help the readers get a sense of what data science is about.

I assume the readers have basic knowledge in statistics and linear algebra, and calculus, such as normal distribution, sample size, gradient, matrix inversion, etc. Previous programming knowledge is not required. However, as the title of the book says, we will learn R and Python in parallel, if you're already familiar with either R or Python it would help learning the other based on side-by-side comparison.

Besides the comparison of the two popular languages used in data science, this book also focuses on the translation from mathematical models to codes. In the book, the audience could find applications/implementations of some important algorithms from scratch, such as maximum likelihood estimation, inversion sampling, copula simulation, simulated annealing, bootstrapping, linear regression (lasso/ridge regression), logistic regression, gradient boosting trees, etc.

0.2 BOOK OVERVIEW

The book consists of six chapters designed to introduce core ideas in data science.

Chapter 1: Introduction to R/Python Programming - R and Python are the two most popular programming languages used in data science. In this chapter, we go through the basics of R and Python programming in parallel with examples. Specifically, the topics of this chapter include variable, type, function, control flow, data structures, and object-oriented programming. The chapter is designed for both beginners and intermediate audience.

Chapter 2: More on R/Python Programming - Following the first chapter, the topics of this chapter include debugging, vectorization, parallelism, working with C++ in R/Python, and functional programming. These topics are chosen to help the audience to get familiar with some intermediate/advanced topics in R/Python programming. Mastering these topics will greatly help with coding skills. Like the first chapter, in this chapter, I try to emphasize the differences between R and Python with coding examples.

Chapter 3: data.table and pandas - In the first two chapters, we focus on general-purpose programming techniques. In this chapter, we introduce the very basics of data science, i.e., data manipulation. For the audience with little experience in data science, we start from a brief introduction to SQL. The major part of this chapter focuses on the two widely used data.frame packages, i.e., data.table in R and pandas in Python. Side-by-side examples using the two packages not only enables the audience to learn basic usages of these tools but also can be used as a quick reference manual.

Chapter 4: Random Variables, Distributions & Linear Regression - In this chapter, we focus on statistics and linear regression, which is the foundation of data science. To better follow this chapter, I recommend any introductory level statistics course as a prerequisite. The topics of this chapter include random variable sampling methods, distribution fitting, joint distribution/copula simulation, confidence interval calculation, and hypothesis testing. In later sections, we also talk about linear regression models from scratch. Many textbooks introduce the theories behind linear regressions but still don't help much on the implementation. We will see how the linear regression is implemented as a toy example in both R and Python with the help of linear algebra. I will also show how the basic linear regression model can be used for L2 penalized linear regression, i.e., ridge regression.

Chapter 5: Optimization in Practice - Most machine learning models rely on optimization algorithms. In this chapter, we give a brief introduction to optimization. Specifically, we will talk about convexity, gradient descent, general-purpose optimization tools in R and Python, linear programming and metaheuristic algorithms, etc. Based on these techniques, we will see coding examples about maximum likelihood estimation, linear regression, logistic regression, portfolio construction, traveling salesman problem.

Chapter 6: Machine Learning – A gentle introduction - Machine learning is a huge topic. In this chapter, I try to give a very short and gentle introduction to machine learning. It starts with a brief introduction of supervised learning, unsupervised learning and reinforcement learning, respectively. For supervised learning, we will see the gradient boosting regression with a pure Python implementation from scratch, from which the audience could learn the translation from the mathematical models to object-oriented programs. For unsupervised learning, the finite Gaussian mixture model and PCA are discussed. And for reinforcement learning, we will also use a simple game as an example to show the usage of deep Q-networks. The goal of this chapter is two-fold. First, I would like to give the audience an impression of what machine learning looks like. Second, reading the code snippets in this chapter could help the audience review/recap the topics in previous chapters.

CHAPTER 1

Introduction to R/Python Programming

In this chapter, I will give an introduction to general R and Python programming in a parallel fashion.

1.1 CALCULATOR

R and Python are general-purpose programming languages that can be used for writing softwares in a variety of domains. But for now, let us start with using them as basic calculators. The first thing is to have them installed. R[1] and Python[2] can be downloaded from their official website. In this book, I will be using R 3.5 and Python 3.7.

To use R/Python as basic calculators, let's get familiar with the interactive mode. After the installation, we can type R or Python (it is case insensitive so we can also type r/python) to invoke the interactive mode. Since Python 2 is installed by default on many machines, in order to avoid invoking Python 2 we type python3.7 instead.

R

```

~ $R

R version 3.5.1 (2018−07−02) −− "Feather Sp\footnoteray"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64−apple−darwin15.6.0 (64−bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

```

[1]https://www.r-project.org
[2]https://www.python.org

```
  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

Python

```
~ $python3.7
Python 3.7.1 (default, Nov  6 2018, 18:45:35)
[Clang 10.0.0 (clang-1000.11.45.5)] on darwin
Type "help", "copyright", "credits" or "license" for more
    information.
>>>
```

The messages displayed by invoking the interactive mode depend on both the version of R/Python installed and the machine. Thus, you may see different messages on your local machine. As the messages said, to quit R we can type `q()`. There are 3 options prompted by asking the user if the workspace should be saved or not. Since we just want to use R as a basic calculator, we quit without saving workspace.

To quit Python, we can simply type `exit()`.

R

```
> q()
Save workspace image? [y/n/c]: n
~ $
```

Once we are inside the interactive mode, we can use R/Python as a calculator.

R

```
> 1+1
[1] 2
> 2*3+5
[1] 11
> log(2)
```

```
[1] 0.6931472
> exp(0)
[1] 1
```

Python

```
>>> 1+1
2
>>> 2*3+5
11
>>> log(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'log' is not defined
>>> exp(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'exp' is not defined
```

From the code snippet above, R is working as a calculator perfectly. However, errors are raised when we call `log(2)` and `exp(2)` in Python. The error messages are self-explanatory - `log` function and `exp` function don't exist in the current Python environment. In fact, `log` function and `exp` function are defined in the `math` module in Python. A module[3] is a file consisting of Python code. When we invoke the interactive mode of Python, a few built-in modules are loaded into the current environment by default. But the `math` module is not included in these built-in modules. That explains why we got the `NameError` when we try to use the functions defined in the `math` module. To resolve the issue, we should first load the functions to use by using the `import` statement as follows.

Python

```
>>> from math import log, exp
>>> log(2)
0.6931471805599453
>>> exp(0)
1.0
```

1.2 VARIABLE AND TYPE

In the previous section we have seen how to use R/Python as calculators. Now, let's see how to write real programs. First, let's define some variables.

[3]https://docs.python.org/3/tutorial/modules.html

R

```
> a=2
> b=5.0
> x='hello world'
> a
[1] 2
> b
[1] 5
> x
[1] "hello world"
> e=a*2+b
> e
[1] 9
```

```
>>> a=2
>>> b=5.0
>>> x='hello world'
>>> a
2
>>> b
5.0
>>> x
'hello world'
>>> e=a*2+b
>>> e
9.0
```

Here, we defined 4 different variables `a, b, x, e`. To get the type of each variable, we can utilize the function `typeof()` in R and `type()` in Python, respectively.

```
> typeof(x)
[1] "character"
> typeof(e)
[1] "double"
```

```
>>> type(x)
<class 'str'>
>>> type(e)
<class 'float'>
```

The type of `x` in R is called character, and in Python is called str.

1.3 FUNCTIONS

We have seen two functions `log` and `exp` when we use R/Python as calculators. A function is a block of code which performs a specific task. A major purpose of wrapping a block of code into a function is to reuse the code.

It is simple to define functions in R/Python.

```
> fun1=function(x){return(x
  *x)}
> fun1
function(x){return(x*x)}
> fun1(2)
[1] 4
```

```
>>> def fun1(x):
...     return x*x # note the
   indentation
...
>>> fun1(2)
4
```

Here, we defined a function `fun1` in R/Python. This function takes `x` as input and returns the square of `x`. When we call a function, we simply type the function name followed by the input argument inside a pair of parentheses. It is worth noting that input or output are not required to define a function. For example, we can define a function `fun2` to print `Hello World`! without input and output.

One major difference between R and Python codes is that Python codes are structured with indentation. Each logical line of R/Python code belongs to a certain group. In R, we use `{}` to determine the grouping of statements. However, in Python we use leading whitespace (spaces and tabs) at the beginning of a logical line to compute the indentation level of the line, which is used to determine the statements' grouping. Let's see what happens if we remove the leading whitespace in the Python function above.

Python

```
>>> def fun1(x):
... return x*x # note the indentation
  File "<stdin>", line 2
    return x*x # note the indentation
         ^
IndentationError: expected an indented block
```

We got an `IndentationError` because of missing indentation.

```
> fun2=function(){print('Hello World!')}
> fun2()
[1] "Hello World!"
```

```
>>> def fun2(): print('Hello World!')
...
>>> fun2()
Hello World!
```

Let's go back to `fun1` and have a closer look at the `return`. In Python, if we want to return something we have to use the keyword `return` explicitly. `return` in R is a function but it is not a function in Python and that is why no parenthesis follows `return` in Python. In R, `return` is not required even though we need to return something from the function. Instead, we can just put the variables to return in the last line of the function defined in R. That being said, we can define `fun1` as follows.

R

```
> fun1=function(x){x*x}
```

Sometimes we want to give a default value to an argument for a function, and both R and Python allow functions to have default values.

R

Python

```r
> log_fun =
    function(x, base=2){
+ return(log(x, base))
+ }
> log_fun(5, base=2)
[1] 2.321928
> log_fun(5, 2)
[1] 2.321928
> log_fun(base=2, 5)
[1] 2.321928
>
```

```python
>>> def log_fun(x, base=2):
...
    return math.log(x, base)
...
>>> log_fun(5,2)
2.321928094887362
>>> log_fun(5, base=2)
2.321928094887362
>>> log_fun(base=2, 5)
  File "<stdin>", line 1
SyntaxError: positional
    argument follows keyword
    argument
```

In Python we have to put the arguments with default values at the end, which is not required in R. However, from a readability perspective, it is always better to put them at the end. You may have noticed the error message above about positional argument. In Python there are two types of arguments, i.e., positional arguments and keyword arguments. Simply speaking, a keyword argument must be preceded by an identifier, e.g., base in the example above. And positional arguments refer to non-keyword arguments.

1.4 CONTROL FLOWS

To implement a complex logic in R/Python, we may need control flows.

1.4.1 If/else

Let's define a function to return the absolute value of input.

```r
> fun3=function(x){
+   if (x>=0){
+     return(x)}
+   else{
+     return(-x)}
+ }
> fun3(2.5)
[1] 2.5
> fun3(-2.5)
[1] 2.5
```

```python
>>> def fun3(x):
...   if x>=0:
...     return x
...   else:
...     return -x
...
>>> fun3(2.5)
2.5
>>> fun3(-2.5)
2.5
```

The code snippet above shows how to use `if/else` in R/Python. The subtle difference between R and Python is that the condition after `if` must be embraced by parenthesis in R but it is optional in Python.

We can also put `if` after `else`. But in Python, we use `elif` as a shortcut.

```
> fun4=function(x){
+   if (x==0){
+     print('zero')}
+   else if (x>0){
+     print('positive')}
+   else{
+     print('negative')}
+ }
> fun4(0)
[1] "zero"
> fun4(1)
[1] "positive"
> fun4(-1)
[1] "negative"
```

```
>>> def fun4(x):
...   if x==0:
...     print('zero')
...   elif x>0:
...     print('positive')
...   else:
...     print('negative')
...
>>> fun4(0)
zero
>>> fun4(1)
positive
>>> fun4(-1)
negative
```

1.4.2 For loop

Similar to the usage of `if` in R, we also have to use parenthesis after the keyword `for` in R. But in Python there should be no parenthesis after `for`.

```
> for (i in 1:3){print(i)}
[1] 1
[1] 2
[1] 3
```

```
>>> for i in range(1,4):print(i)
...
1
2
3
```

There is something more interesting than the `for loop` itself in the snippets above. In the R code, the expression `1:3` creates a vector with elements 1, 2 and 3. In the Python code, we use the `range()` function for the first time. Let's have a look at them.

```
> typeof(1:3)
[1] "integer"
```

```
>>> type(range(1,4))
<class 'range'>
```

`range()` function returns a `range` type object, which represents an immutable sequence of numbers. `range()` function can take three arguments, i.e., `range(start, stop, step)`. However, `start` and `step` are both optional. It's critical to keep in mind that the `stop` argument that defines the upper limit of the sequence is exclusive. And that is why in order to loop through 1 to 3 we have to pass 4 as the `stop` argument to `range()` function. The `step` argument specifies how much to increase from one number to the next. The default values of `start` and `step` are 0 and 1, respectively.

1.4.3 While loop

R

```
> i=1
> while (i<=3){
+   print(i)
+   i=i+1
+ }
[1] 1
[1] 2
[1] 3
```

```
>>> i=1
>>> while i<=3:
...   print(i)
...   i+=1
...
1
2
3
```

You may have noticed that in Python we can do `i+=1` to add 1 to `i`, which is not feasible in R by default. Both for loop and while loop can be nested.

1.4.4 Break/continue

Break/continue helps if we want to break the for/while loop earlier, or to skip a specific iteration. In R, the keyword for continue is called `next`, in contrast to `continue` in Python. The difference between `break` and `continue` is that calling `break` would exit the innermost loop (when there are nested loops, only the innermost loop is affected); while calling `continue` would just skip the current iteration and continue the loop if not finished.

R

```
> for (i in 1:3){
+   print(i)
+   if (i==1) break
+ }
[1] 1
> for (i in 1:3){
+   if (i==2){next}
+   print(i)
+ }
[1] 1
[1] 3
```

Python

```
>>> for i in range(1,4):
...   print(i)
...   if i==1: break
...
1
>>> for i in range(1,4):
...   if i==2: continue
...   print(i)
...
1
3
```

1.5 SOME BUILT-IN DATA STRUCTURES

In the previous sections, we haven't seen much difference between R and Python. However, regarding the built-in data structures, there are some significant differences we will see in this section.

1.5.1 Vector in R and list in Python

In R, we can use function `c()` to create a vector; A vector is a sequence of elements with the same type. In Python, we can use `[]` to create a list, which is also a sequence of elements. But the elements in a list don't need to have the same type. To get the number of elements in a vector in R, we use the function `length()`; and to get the number of elements in a list in Python, we use the function `len()`.

```
> x=c(1,2,5,6)
> y=c('hello','world','!')
> x
[1] 1 2 5 6
> y
[1] "hello" "world" "!"
> length(x)
[1] 4
> z=c(1,'hello')
> z
[1] "1"     "hello"
```

```
>>> x=[1,2,5,6]
>>> y=['hello','world','!']
>>> x
[1, 2, 5, 6]
>>> y
['hello', 'world', '!']
>>> len(x)
4
>>> z=[1,'hello']
>>> z
[1, 'hello']
```

In the code snippet above, the first element in the variable `z` in R is coerced from 1 (numeric) to "1" (character) since the elements must have the same type.

To access a specific element from a vector or list, we could use `[]`. In R, sequence types are indexed beginning with the one subscript. In contrast, sequence types in Python are indexed beginning with the zero subscript.

```
> x=c(1,2,5,6)
> x[1]
[1] 1
```

Python

```
>>> x=[1,2,5,6]
>>> x[1]
2
>>> x[0]
1
```

What if the index to access is out of boundary?

R

```r
> x=c(1,2,5,6)
> x[-1]
[1] 2 5 6
> x[0]
numeric(0)
> x[length(x)+1]
[1] NA
> length(numeric(0))
[1] 0
> length(NA)
[1] 1
```

Python

```python
>>> x=[1,2,5,6]
>>> x[-1]
6
>>> x[len(x)+1]
Traceback (most recent call
    last):
  File "<stdin>", line 1,
      in <module>
IndexError:
    list index out of range
```

In Python, negative index number means indexing from the end of the list. Thus, `x[-1]` points to the last element and `x[-2]` points to the second-last element of the list. But R doesn't support indexing with negative numbers in the same way as Python. Specifically, in R `x[-index]` returns a new vector with `x[index]` excluded.

When we try to access with an index out of boundary, Python would throw an `IndexError`. The behavior of R when indexing out of boundary is more interesting. First, when we try to access `x[0]` in R we get a `numeric(0)` whose length is also 0. Since its length is 0, `numeric(0)` can be interpreted as an empty numeric vector. When we try to access `x[length(x)+1]` we get an `NA`. In R, there are also `NaN` and `NULL`.

`NaN` means "Not A Number" and it can be verified by checking its type - "double". `0/0` would result in an `NaN` in R. `NA` in R generally represents missing values. And `NULL` represents a NULL (empty) object. To check if a value is `NA`, `NaN` or `NULL`, we can use `is.na()`, `is.nan()` or `is.null`, respectively.

```r
> typeof(NA)
[1] "logical"
> typeof(NaN)
[1] "double"
> typeof(NULL)
[1] "NULL"
> is.na(NA)
[1] TRUE
> is.null(NULL)
[1] TRUE
> is.nan(NaN)
```

Python

```python
>>> type(None)
<class 'NoneType'>
>>> None is None
True
>>> 1 == None
False
```

In Python, there is no built-in `NA` or `NaN`. The counterpart of `NULL` in Python is

None. In Python, we can use the is keyword or == to check if a value is equal to None.

From the code snippet above, we also notice that in R the boolean type value is written as "TRUE/FALSE", compared with "True/False" in Python. Although in R "TRUE/FALSE" can also be abbreviated as "T/F", I don't recommend using the abbreviation.

There is one interesting fact that we can't add a NULL to a vector in R, but it is feasible to add a None to a list in Python.

```r
> x=c(1,NA,NaN,NULL)
> x
[1]   1  NA NaN
> length(x)
[1] 3
```

```python
>>> x=[1,None]
>>> x
[1, None]
>>> len(x)
2
```

Sometimes we want to create a vector/list with replicated elements, for example, a vector/list with all elements equal to 0.

```r
> x=rep(0, 10)
> x
 [1] 0 0 0 0 0 0 0 0 0 0
> y=rep(c(0,1), 5)
> y
 [1] 0 1 0 1 0 1 0 1 0 1
```

```python
>>> x=[0]*10
>>> x
[0, 0, 0, 0, 0, 0, 0, 0, 0,
    0]
>>> y=[0, 1]*5
>>> y
[0, 1, 0, 1, 0, 1, 0, 1, 0,
    1]
```

When we use the ∗ operator to make replicates of a list, there is one caveat - if the element inside the list is mutable then the replicated elements point to the same memory address. As a consequence, if one element is mutated other elements are also affected.

Python

```python
>>> x=[0] # x is a list which is mutable
>>> y=[x]*5 # each element in y points to x
>>> y
[[0], [0], [0], [0], [0]]
>>> y[2]=2 # we point y[2] to 2 but x is not mutated
>>> y
[[0], [0], 2, [0], [0]]
>>> y[1][0]=-1 # we mutate x by changing y[1][0] from 0 to -1
>>> y
[[-1], [-1], 2, [-1], [-1]]
>>> x
[-1]
```

How to get a list with replicated elements but pointing to different memory addresses?

Python

```
>>> x=[0]
>>> y=[x[:] for _ in range(5)] # [:] makes a copy of the list x;
    another solution is [list(x) for _ in range(5)]
>>> y
[[0], [0], [0], [0], [0]]
>>> y[0][0]=2
>>> y
[[2], [0], [0], [0], [0]]
```

Besides accessing a specific element from a vector/list, we may also need to do slicing, i.e., to select a subset of the vector/list. There are two basic approaches of slicing:

- Integer-based

  R

```
> x=c(1,2,3,4,5,6)
> x[2:4]
[1] 2 3 4
> x[c(1,2,5)] # a vector of indices
[1] 1 2 5
> x[seq(1,5,2)] # seq creates a vector to be used as
    indices
[1] 1 3 5
```

  Python

```
>>> x=[1,2,3,4,5,6]
>>> x[1:4] # x[start:end] start is inclusive but end is
    exclusive
[2, 3, 4]
>>> x[0:5:2] # x[start:end:step]
[1, 3, 5]
```

The code snippet above uses hash character # for comments in both R and Python. Everything after # on the same line would be treated as comment (not executable). In the R code, we also used the function `seq()` to create a vector. When I see a function that I haven't seen before, I might either google it or use the built-in helper mechanism. Specifically, in R use `?` and in Python use `help()`.

```
> ?seq
```

```
>>> help(print)
```

- Condition-based

  Condition-based slicing means to select a subset of the elements which satisfy certain conditions. In R, it is quite straightforward by using a boolean vector whose length is the same as the vector to be sliced.

R

```
> x=c(1,2,5,5,6,6)
> x[x %% 2==1] # %% is the modulo operator in R; we select
    the odd elements
[1] 1 5 5
> x %% 2==1 # results in a boolean vector with the same
    length as x
[1]  TRUE FALSE  TRUE  TRUE FALSE FALSE
```

The condition-based slicing in Python is quite different from that in R. The prerequisite is list comprehension which provides a concise way to create new lists in Python. For example, let's create a list of squares of another list.

Python

```
>>> x=[1,2,5,5,6,6]
>>> [e**2 for e in x] # ** is the exponent operator, i.e.,
    x**y means x to the power of y
[1, 4, 25, 25, 36, 36]
```

We can also use `if` statement with list comprehension to filter a list to achieve list slicing.

Python

```
>>> x=[1,2,5,5,6,6]
>>> [e for e in x if e%2==1] # % is the modulo operator in Python
[1, 5, 5]
```

It is also common to use `if/else` with list comprehension to achieve more complex operations. For example, given a list x, let's create a new list y so that the non-negative elements in x are squared and the negative elements are replaced by 0s.

Python

```
>>> x=[1,−1,0,2,5,−3]
>>> [e∗∗2 if e>=0 else 0 for e in x]
[1, 0, 0, 4, 25, 0]
```

The example above shows the power of list comprehension. To use `if` with list comprehension, the `if` statement should be placed in the end after the `for` loop statement; but to use `if/else` with list comprehension, the `if/else` statement should be placed before the `for` loop statement.

We can also modify the value of an element in a vector/list variable.

```
> x=c(1,2,3)
> x[1]=−1
> x
[1] −1  2  3
```

```
>>> x=[1,2,3]
>>> x[0]=−1
>>> x
[−1, 2, 3]
```

Two or multiple vectors/lists can be concatenated easily.

```
> x=c(1,2)
> y=c(3,4)
> z=c(5,6,7,8)
> c(x,y,z)
[1] 1 2 3 4 5 6 7 8
```

```
>>> x=[1,2]
>>> y=[3,4]
>>> z=[5,6,7,8]
>>> x+y+z
[1, 2, 3, 4, 5, 6, 7, 8]
```

As the list structure in Python is mutable, there are many things we can do with list.

Python

```
>>> x=[1,2,3]
>>> x.append(4) # append a single value to the list x
>>> x
[1, 2, 3, 4]
>>> y=[5,6]
>>> x.extend(y) # extend list y to x
>>> x
[1, 2, 3, 4, 5, 6]
>>> last=x.pop() # pop the last element from x
>>> last
6
>>> x
[1, 2, 3, 4, 5]
```

I like the list structure in Python much more than the vector structure in R. list in Python has a lot more useful features which can be found from the python official documentation[4].

1.5.2 Array

Array is one of the most important data structures in scientific programming. In R, there is also an object type "matrix", but according to my own experience, we can almost ignore its existence and use array instead. We can definitely use list as array in Python, but lots of linear algebra operations are not supported for the list type. Fortunately, there is a Python package `numpy` off the shelf.

R

```
> x=1:12
> array1=array(x, c(4,3)) # convert vector x to a 4 rows * 3
    cols array
> array1
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
> y=1:6
> array2=array(y, c(3,2)) # convert vector y to a 3 rows * 2
    cols array
> array2
     [,1] [,2]
```

[4]https://docs.python.org/3/tutorial/datastructures.html

```
[1,]    1    4
[2,]    2    5
[3,]    3    6
> array3 = array1 %*% array2 # %*% is the matrix multiplication
    operator
> array3
     [,1] [,2]
[1,]   38   83
[2,]   44   98
[3,]   50  113
[4,]   56  128
> dim(array3) # get the dimension of array3
[1] 4 2
```

Python

```
>>> import numpy as np # we import the numpy module and alias it
     as np
>>> array1=np.reshape(list(range(1,13)),(4,3)) # convert a list
    to a 2d np.array
>>> array1
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
>>> type(array1)
<class 'numpy.ndarray'>
>>> array2=np.reshape(list(range(1,7)),(3,2))
>>> array2
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> array3=np.dot(array1, array2) # matrix multiplication using
    np.dot()
>>> array3
array([[ 22,  28],
       [ 49,  64],
       [ 76, 100],
       [103, 136]])
>>> array3.shape # get the shape(dimension) of array3
(4, 2)
```

You may have noticed that the results of the R code snippet and Python code snippet are different. The reason is that in R the conversion from a vector to an array is by-column; but in numpy the reshape from a list to a 2D numpy.array is by-row. There are two ways to reshape a list to a 2D numpy.array by column.

Python

```
>>> array1=np.reshape(list(range(1,13)),(4,3),order='F') # use order='F'
>>> array1
array([[ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11],
       [ 4,  8, 12]])
>>> array2=np.reshape(list(range(1,7)),(2,3)).T # use .T to transpose an array
>>> array2
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> np.dot(array1, array2) # now we get the same result as using R
array([[ 38,  83],
       [ 44,  98],
       [ 50, 113],
       [ 56, 128]])
```

To learn more about numpy, the official website[5] has great documentation/tutorials.

The term broadcasting describes how arrays with different shapes are handled during arithmetic operations. A simple example of broadcasting is given below.

R

```
> x = c(1, 2, 3)
> x+1
[1] 2 3 4
```

Python

```
>>> import numpy as np
>>> x = np.array([1, 2, 3])
>>> x + 1
array([2, 3, 4])
```

However, the broadcasting rules in R and Python are not exactly the same.

R

```
> x = array(c(1:6), c(3,2))
> y = c(1, 2, 3)
> z = c(1, 2)
```

Python

```
>>> import numpy as np
>>> x = np.array([[1, 2],
    [3, 4], [5, 6]])
```

[5]http://www.numpy.org

```
# point−wise multiplication
> x * y
     [,1] [,2]
[1,]    1    4
[2,]    4   10
[3,]    9   18
> x*z
     [,1] [,2]
[1,]    1    8
[2,]    4    5
[3,]    3   12
```

```
>>> y = np.array([1, 2, 3])
>>> z = np.array([1, 2])
>>> # point−wise multiplication
>>> x * y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (3,2) (3,)
>>> x * z
array([[ 1,  4],
       [ 3,  8],
       [ 5, 12]])
```

From the R code, we see the broadcasting in R is like recycling along with the column. In Python, when the two arrays have different dimensions, the one with fewer dimensions is padded with ones on its leading side. According to this rule, when we do $x * y$, the dimension of x is $(3, 2)$ but the dimension of y is 3. Thus, the dimension of y is padded to $(1, 3)$, which explains what happens when $x * y$.

1.5.3 List in R and dictionary in Python

Yes, in R there is also an object type called list. The major difference between a vector and a list in R is that a list could contain different types of elements. list in R supports integer-based accessing using `[[]]` (compared to `[]` for vector).

R

```
> x=list(1,'hello world!')
> x
[[1]]
[1] 1

[[2]]
[1] "hello world!"

> x[[1]]
[1] 1
> x[[2]]
[1] "hello world!"
> length(x)
[1] 2
```

list in R could be named and support accessing by name via either `[[]]` or `$` operator. But vector in R can also be named and support accessing by name.

R

```
> x=c('a'=1,'b'=2)
> names(x)
[1] "a" "b"
> x['b']
b
2
> l=list('a'=1,'b'=2)
> l[['b']]
[1] 2
> l$b
[1] 2
> names(l)
[1] "a" "b"
```

However, elements in list in Python can't be named as R. If we need the feature of accessing by name in Python, we can use the dictionary structure. If you used Java before, you may consider dictionary in Python as the counterpart of HashMap in Java. Essentially, a dictionary in Python is a collection of key:value pairs.

Python

```
>>> x={'a':1,'b':2} # {key:value} pairs
>>> x
{'a': 1, 'b': 2}
>>> x['a']
1
>>> x['b']
2
>>> len(x) # number of key:value pairs
2
>>> x.pop('a') # remove the key 'a' and we get its value 1
1
>>> x
{'b': 2}
```

Unlike dictionary in Python, list in R doesn't support the `pop()` operation. Thus, in order to modify a list in R, a new one must be created explicitly or implicitly.

1.5.4 data.frame, data.table and pandas

data.frame is a built-in type in R for data manipulation. In Python, there is no such built-in data structure since Python is a more general-purpose programming language. The solution for data.frame in Python is the `pandas`[6] module.

Before we dive into data.frame, you may be curious as to why we need it. In other words, why can't we just use vector, list, array/matrix and dictionary for all data manipulation tasks? I would say yes - data.frame is not a must-have feature for most of ETL (extraction, transformation and Load) operations. But data.frame provides a very intuitive way for us to understand the structured dataset. A data.frame is usually flat with 2 dimensions, i.e., row and column. The row dimension is across multiple observations and the column dimension is across multiple attributes/features. If you are familiar with relational database, a data.frame can be viewed as a table.

Let's see an example of using data.frame to represent employees' information in a company.

R

```
> employee_df = data.
    frame(name=c("A", "B", "C
    "),department=c("
    Engineering","Operations
    ","Sales"))
> employee_df
  name  department
1    A Engineering
2    B  Operations
3    C       Sales
```

Python

```
>>> import pandas as pd
>>> employee_df=pd.DataFrame
    ({'name':['A','B','C'],'
    department':["Engineering
    ","Operations","Sales"]})
>>> employee_df
  name   department
0    A  Engineering
1    B   Operations
2    C        Sales
```

There are quite a few ways to create data.frame. The most commonly used one is to create data.frame object from array/matrix. We may also need to convert a numeric data.frame to an array/matrix.

R

```
> x=array(rnorm(12), c(3,4))
> x
           [,1]       [,2]      [,3]       [,4]
[1,] -0.8101246 -0.8594136 -2.260810  0.5727590
[2,] -0.9175476  0.1345982  1.067628 -0.7643533
[3,]  0.7865971 -1.9046711 -0.154928 -0.6807527
> random_df=as.data.frame(x)
> random_df
          V1         V2        V3         V4
1 -0.8101246 -0.8594136 -2.260810  0.5727590
```

[6]https://pandas.pydata.org/

```
2 -0.9175476  0.1345982  1.067628 -0.7643533
3  0.7865971 -1.9046711 -0.154928 -0.6807527
> data.matrix(random_df)
             V1         V2        V3         V4
[1,] -0.8101246 -0.8594136 -2.260810  0.5727590
[2,] -0.9175476  0.1345982  1.067628 -0.7643533
[3,]  0.7865971 -1.9046711 -0.154928 -0.6807527
```

Python

```
>>> import numpy as np
>>> import pandas as pd
>>> x=np.random.normal(size=(3,4))
>>> x
array([[-0.54164878, -0.14285267, -0.39835535, -0.81522719],
       [ 0.01540508,  0.63556266,  0.16800583,  0.17594448],
       [-1.21598262,  0.52860817, -0.61757696,  0.18445057]])
>>> random_df = pd.DataFrame(x)
>>> random_df
          0         1         2         3
0 -0.541649 -0.142853 -0.398355 -0.815227
1  0.015405  0.635563  0.168006  0.175944
2 -1.215983  0.528608 -0.617577  0.184451
>>> np.asarray(random_df)
array([[-0.54164878, -0.14285267, -0.39835535, -0.81522719],
       [ 0.01540508,  0.63556266,  0.16800583,  0.17594448],
       [-1.21598262,  0.52860817, -0.61757696,  0.18445057]])
```

In general, operations on an array/matrix are much faster than those on a data frame. In R, we may use the built-in function `data.matrix` to convert a data.frame to an array/matrix. In Python, we could use the function `asarray` in `numpy` module.

Although data.frame is a built-in type, it is not quite efficient for many operations. I would suggest using data.table[7] whenever possible. `dplyr`[8] is also a very popular package in R for data manipulation. Many good resources are available online to learn data.table and pandas.

1.6 REVISIT OF VARIABLES

We have talked about variables and functions so far. When a function has a name, its name is also a valid variable. After all, what is a variable?

In mathematics, a variable is a symbol that represents an element, and we do

[7]https://cran.r-project.org/web/packages/data.table/index.html
[8]https://dplyr.tidyverse.org/

not care whether we conceptualize a variable in our mind, or write it down on paper. However, in programming a variable is not only a symbol. We have to understand that a variable is a name given to a memory location in computer systems. When we run `x = 2` in R or Python, somewhere in memory has the value 2, and the variable (name) points to this memory address. If we further run `y = x`, the variable `y` points to the same memory location pointed to by `(x)`. What if we run `x=3`? It doesn't modify the memory which stores the value 2. Instead, somewhere in the memory now has the value 3 and this memory location has a name `x`. And the variable `y` is not affected at all, as well as the memory location it points to.

1.6.1 Mutability

Almost everything in R or Python is an object, including these data structures we introduced in previous sections. Mutability is a property of objects, not variables, because a variable is just a name.

A list in Python is mutable, meaning that we could change the elements stored in the list object without copying the list object from one memory location to another. We can use the `id` function in Python to check the memory location for a variable. In the code below, we modified the first element of the list object with name `x`. And since Python list is mutable, the memory address of the list doesn't change.

Python

```
>>> x=list(range(1,1001)) # list() convert a range object to a
    list
>>> hex(id(x)) # print the memory address of x
'0x10592d908'
>>> x[0]=1.0 # from integer to float
>>> hex(id(x))
'0x10592d908'
```

Is there any immutable data structure in Python? Yes, for example tuple is immutable, which contains a sequence of elements. The element accessing and subset slicing of tuple follows the same rules as list in Python.

Python

```
>>> x=(1,2,3,) # use () to create a tuple in Python, it is
    better to always put a comma in the end
>>> type(x)
<class 'tuple'>
>>> len(x)
3
>>> x[0]
1
>>> x[0]=-1
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

If we have two Python variables pointed to the same memory, when we modify the memory via one variable the other is also affected as we expect (see the example below).

Python

```
>>> x=[1,2,3]
>>> id(x)
4535423616
>>> x[0]=0
>>> x=[1,2,3]
>>> y=x
>>> id(x)
4535459104
>>> id(y)
4535459104
>>> x[0]=0
>>> id(x)
4535459104
>>> id(y)
4535459104
>>> x
[0, 2, 3]
>>> y
[0, 2, 3]
```

In contrast, the mutability of vector in R is more complex and sometimes confusing. First, let's see the behavior when there is a single name given to the vector object stored in memory.

R

```
> a=c(1,2,3)
> .Internal(inspect(a))
@7fe94408f3c8 14 REALSXP g0c3 [NAM(1)] (len=3, tl=0) 1,2,3
> a[1]=0
> .Internal(inspect(a))
@7fe94408f3c8 14 REALSXP g0c3 [NAM(1)] (len=3, tl=0) 0,2,3
```

It is clear in this case the vector object is mutable since the memory address doesn't change after the modification. What if there is an additional name given to the memory?

R

```
> a=c(1,2,3)
> b=a
> .Internal(inspect(a))
@7fe94408f238 14 REALSXP g0c3 [NAM(2)] (len=3, tl=0) 1,2,3
> .Internal(inspect(b))
@7fe94408f238 14 REALSXP g0c3 [NAM(2)] (len=3, tl=0) 1,2,3
> a[1]=0
> .Internal(inspect(a))
@7fe94408f0a8 14 REALSXP g0c3 [NAM(1)] (len=3, tl=0) 0,2,3
> .Internal(inspect(b))
@7fe94408f238 14 REALSXP g0c3 [NAM(2)] (len=3, tl=0) 1,2,3
> a
[1] 0 2 3
> b
[1] 1 2 3
```

Before the modification, both variable `a` and `b` point to the same vector object in the memory. But surprisingly, after the modification the memory address of variable `a` also changed, which is called "copy on modify" in R. And because of this unique behavior, the modification of `a` doesn't affect the object stored in the old memory and thus the vector object is immutable in this case. The mutability of R `list` is similar to that of R `vector`.

1.6.2 Variable as function argument

Most functions/methods in R and Python take some variables as argument. What happens when we pass the variables into a function?

In Python, the variable, i.e., the name of the object, is passed into a function. If the variable points to an immutable object, any modification to the variable, i.e., the name, doesn't persist. However, when the variable points to a mutable object, the modification of the object stored in memory persists. Let's see the examples below.

```
>>> def g(x):
...   print(id(x))
...   x-=1
...   print(id(x))
...   print(x)
>>> a=1
>>> id(a)
```

```
>>> def f(x):
...   id(x)
...   x[0]-=1
...   id(x)
>>> a=[1,2,3]
>>> id(a)
4535423616
```

```
4531658512
>>> g(a)
4531658512
4531658480
0
>>> a
1
```

```
>>> f(a)
4535423616
4535423616
>>> a
[0, 2, 3]
```

We see that the object is passed into function by its name. If the object is immutable, a new copy is created in memory when any modification is made to the original object. When the object is immutable, no new copy is made and thus the change persists out of the function.

In R, the passed object is always copied on a modification inside the function, and thus no modification can be made to the original object in memory.

R

```
> f=function(x){
+   print(.Internal(inspect(x)))
+   x[1]=x[1]-1
+   print(.Internal(inspect(x)))
+   print(x)
+ }
>
> a=c(1,2,3)
> .Internal(inspect(a))
@7fe945538688 14 REALSXP g0c3 [NAM(1)] (len=3, tl=0) 1,2,3
> f(a)
@7fe945538688 14 REALSXP g0c3 [NAM(3)] (len=3, tl=0) 1,2,3
[1] 1 2 3
@7fe945538598 14 REALSXP g0c3 [NAM(1)] (len=3, tl=0) 0,2,3
[1] 0 2 3
[1] 0 2 3
> a
[1] 1 2 3
```

People may argue that R functions are not as flexible as Python functions. However, it makes more sense to do functional programming in R since we usually can't modify objects passed into a function.

1.6.3 Scope of variables

What is the scope of a variable and why does it matter? Let's first have a look at the code snippets below.

R

```
> x=1
> var_func_1 =
    function(){print(x)}
> var_func_1()
[1] 1
> var_func_2 = function(){x=
    x+1; print(x)}
> var_func_2()
[1] 2
> x
[1] 1
```

Python

```
>>> x=1
>>>
    def var_func_1():print(x)
>>> var_func_1()
1
>>> def var_func_2():x+=1
...
>>> var_func_2()
Traceback (most recent call
    last):
  File "<stdin>", line 1,
      in <module>
  File "<stdin>", line 1,
      in var_func_2
UnboundLocalError: local
    variable 'x
    ' referenced before
    assignment
```

The results of the code above seem strange before knowing the concept of variable scope. Inside a function, a variable may refer to a function argument/parameter or it could be formally declared inside the function which is called a local variable. But in the code above, `x` is neither a function argument nor a local variable. How does the `print()` function know what the identifier `x` points to?

The scope of a variable determines where the variable is available/accessible (can be referenced). Both R and Python apply lexical/static scoping for variables, which set the scope of a variable based on the structure of the program. In static scoping, when an 'unknown' variable is referenced, the function will try to find it from the most closely enclosing block. That explains how the `print()` function could find the variable `x`.

In the R code above, `x=x+1` the first `x` is a local variable created by the = operator; the second `x` is referenced inside the function so the static scoping rule applies. As a result, a local variable `x` which is equal to 2 is created, which is independent with the `x` outside of the function `var_func_2()`. However, in Python when a variable is assigned a value in a statement, the variable would be treated as a local variable and that explains the `UnboundLocalError`.

Is it possible to change a variable inside a function which is declared outside the function without passing it as an argument? Based on the static scoping rule only, it's impossible. But there are workarounds in both R/Python. In R, we need the help of environment; and in Python we can use the keyword `global`.

So what is an environment in R? An environment is a place where objects are stored. When we invoke the interactive R session, an environment named `.GlobalEnv` is created automatically. We can also use the function `environment()` to get the present environment. The `ls()` function can take an environment as the argument to list all objects inside the environment.

R

```
$r
> typeof(.GlobalEnv)
[1] "environment"
> environment()
<environment: R_GlobalEnv>
> x=1
> ls(environment())
[1] "x"
> env_func_1=function(x){
+   y=x+1
+   print(environment())
+   ls(environment())
+ }
> env_func_1(2)
<environment: 0x7fc59d165a20>
[1] "x" "y"
> env_func_2=function(){print(environment())}
> env_func_2()
<environment: 0x7fc59d16f520>
```

The above code shows that each function has its own environment containing all function arguments and local variables declared inside the function. In order to change a variable declared outside of a function, we need the access of the environment enclosing the variable to change. There is a function `parent_env(e)` that returns the parent environment of the given environment `e` in R. Using this function, we are able to change the value of `x` declared in `.GlobalEnv` inside a function which is also declared in `.GlobalEnv`. The `global` keyword in Python works in a totally different way, which is simple but less flexible.

```
> x=1
> env_func_3=function(){
+   cur_env=environment()
+   par_env=
    parent.env(cur_env)
+   par_env$x=2
+ }
> env_func_3()
> x
[1] 2
```

Python

```
>>> def env_func_3():
...   global x
...   x = 2
...
>>> x=1
>>> env_func_3()
>>> x
2
```

I seldom use the `global` keyword in Python, if ever. But the environment in R could be very handy on some occasions. In R, environment could be used as a purely mutable version of the `list` data structure. Let's use the R function `tracemem` to trace the copy of an object. It is worth nothing that `tracemem` can't trace R functions.

```
# list is not purely mutable
> x=list(1)
> tracemem(x)
[1] "<0x7f829183f6f8>"
> x$a=2
> tracemem(x)
[1] "<0x7f828f4d05c8>"
```

```
# environment is purely
  mutable
> x=new.env()
> x
<environment: 0x7f8290aee7e8
  >
> x$a=2
> x
<environment: 0x7f8290aee7e8
  >
```

Actually, the object of an R6 class type is also an environment.

R

```
> # load the Complex class that we defined in chapter 1
> x = Complex$new(1,2)
> typeof(x)
[1] "environment"
```

In Python, we can assign values to multiple variables in one line.

```
>>> x,y = 1,2
>>> x
1
>>> y
2
```

```
>>> x,y=(1,2)
>>> print(x,y)
1 2
>>> (x,y)=(1,2)
>>> print(x,y)
1 2
>>> [x,y]=(1,2)
>>> print(x,y)
1 2
```

Even though in the left snippet above there aren't parentheses embracing `1, 2` after the `=` operator, a tuple is created first and then the tuple is unpacked and assigned to `x, y`. Such a mechanism doesn't exist in R, but we can define our own multiple assignment operator with the help of environment.

R chapter1/multi_assignment.R

```
‘%=%‘ = function(left, right) {
  # we require the RHS to be a list strictly
  stopifnot(is.list(right))
  # dest_env is the desitination environment enclosing the
     variables on LHS
  dest_env = parent.env(environment())
  left = substitute(left)

  recursive_assign = function(left, right, dest_env) {
    if (length(left) == 1) {
      assign(x = deparse(left),
             value = right,
             envir = dest_env)
      return()
    }
    if (length(left) != length(right) + 1) {
      stop("LHS and RHS must have the same shapes")
    }

    for (i in 2:length(left)) {
      recursive_assign(left[[i]], right[[i - 1]],  dest_env)
    }
  }

  recursive_assign(left, right, dest_env)
}
```

Before going more deeply into the script, first let’s see the usage of the multiple assignment operator we defined.

R

```
> source('multi_assignment.R')
> c(x,y,z) %=% list(1,"Hello World!",c(2,3))
> x
[1] 1
> y
[1] "Hello World!"
> z
[1] 2 3
> list(a,b) %=% list(1,as.Date('2019-01-01'))
> a
[1] 1
> b
```

```
[1] "2019-01-01"
```

In the `%=%` operator defined above, we used two functions `substitute, deparse` which are very powerful but less known by R novices. To better understand these functions as well as some other less known R functions, the Rchaeology[9] tutorial is worth reading.

It is also interesting to see that we defined the function `recursive_assign` inside the `%=%` function. Both R and Python support the concept of first class functions. More specifically, a function in R/Python is an object, which can be

1. stored as a variable;
2. passed as a function argument;
3. returned from a function.

The essential idea behind the `recursive_assign` function is a depth-first search (DFS), which is a fundamental graph traversing algorithm[10]. In the context of the `recursive_assign` function, we use DFS to traverse the parse tree of the `left` argument created by calling `substitute(left)`.

1.7 OBJECT-ORIENTED PROGRAMMING (OOP) IN R/PYTHON

All the codes we wrote above follow the procedural programming paradigm[11]. We can also do functional programming (FP) and OOP in R/Python. In this section, let's focus on OOP in R/Python.

Class is the key concept in OOP. In R there are two commonly used built-in systems to define classes, i.e., S3 and S4. In addition, there is an external package R6[12] which defines R6 classes. S3 is a light-weight system but its style is quite different from OOP in many other programming languages. S4 system follows the principles of modern object-oriented programming much better than S3. However, the usage of S4 classes is quite tedious. I would ignore S3/S4 and introduce R6, which is closer to the class in Python.

Let's build a class in R/Python to represent complex numbers.

R

```
> library(R6) # load the R6 package
>
> Complex = R6Class("Complex",
+ public = list( # only elements declared in this list are
    accessible by the object of this class
```

[9]https://cran.r-project.org/web/packages/rockchalk/vignettes/Rchaeology.pdf
[10]https://en.wikipedia.org/wiki/Depth-first_search
[11]https://en.wikipedia.org/wiki/Comparison_of_programming_paradigms
[12]https://cran.r-project.org/web/packages/R6/index.html

```
+ real = NULL,
+ imag = NULL,
+ # the initialize function would be called automatically when
    we create an object of the class
+ initialize = function(real, imag){
+     # call functions to change real and imag values
+   self$set_real(real)
+   self$set_imag(imag)
+ },
+ # define a function to change the real value
+ set_real = function(real){
+   self$real=real
+ },
+ # define a function to change the imag value
+ set_imag = function(imag){
+   self$imag = imag
+ },
+ # override print function
+ print = function(){
+   cat(paste0(as.character(self$real),'+',as.character(self$
    imag),'j'),'\n')
+ }
+ )
+ )
> # let's create a complex number object based on the Complex
    class we defined above using the new function
> x = Complex$new(1,2)
> x
1+2j
> x$real # the public attributes of x could be accessed by $
    operator
[1] 1
```

Python

```
>>> class Complex:
...   # the __init__ function would be called automatically when
     we create an object of the class
...   def __init__(self, real, imag):
...     self.real = real
...     self.imag = imag

...   def __repr__(self):
...     return "{0}+{1}j".format(self.real, self.imag)
```

```
...
>>> x = Complex(1,2)
>>> x
1+2j
>>> x.real # different from the $ operator in R, here we use .
    to access the attribute of an object
1
```

By overriding the print function in the R6 class, we can have the object printed in the format of `real+imag j`. To achieve the same effect in Python, we override the method `__repr__`. In Python, we refer to the functions defined in classes as methods. And overriding a method means changing the implementation of a method provided by one of its ancestors. To understand the concept of ancestors in OOP, one needs to understand the concept of inheritance[13].

You may be curious about the double underscore surrounding the methods, such as `__init__` and `__repr__`. These methods are well-known as special methods[14]. In the definition of the special method `__repr__` in the Python code, the `format` method of `str` object[15] is used.

Special methods can be very handy if we use them in suitable cases. For example, we can use the special method `__add__` to implement the + operator for the `Complex` class we defined above.

Python

```
>>> class Complex:
...   def __init__(self, real, imag):
...     self.real = real
...     self.imag = imag
...   def __repr__(self):
...     return "{0}+{1}j".format(self.real, self.imag)
...   def __add__(self, another):
...     return Complex(self.real + another.real, self.imag +
    another.imag)
...
>>> x = Complex(1,2)
>>> y = Complex(2,4)
>>> x+y # + operator works now
3+6j
```

We can also implement the + operator for `Complex` class in R as we have done for Python.

[13]https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)
[14]https://docs.python.org/3/reference/datamodel.html#specialnames
[15]https://docs.python.org/3.7/library/string.html

R

```
>  `+.Complex` = function(x, y){
+    Complex$new(x$real+y$real, x$imag+y$imag)
+ }
> x=Complex$new(1,2)
> y=Complex$new(2,4)
> x+y
3+6j
```

The most interesting part of the code above is `` `+.Complex` ``. First, why do we use ` `` ` to quote the function name? Before getting into this question, let's have a look at the Python 3's variable naming rules[16].

```
Within the ASCII range (U+0001..U+007F), the valid characters
    for identifiers (also referred to as names) are the same as
    in Python 2.x: the uppercase and lowercase letters A through
     Z, the underscore _ and, except for the first character,
    the digits 0 through 9.
```

According to the rule, we can't declare a variable with name `2x`. Compared with Python, in R we can also use `.` in the variable names[17]. However, there is a workaround to use invalid variable names in R with the help of ` `` `.

R

```
> 2x = 5
Error: unexpected symbol in
    "2x"
> .x = 3
> .x
[1] 3
> `+2x%` = 0
> `+2x%`
[1] 0
```

Python

```
>>> 2x = 5
  File "<stdin>", line 1
    2x = 5
     ^
SyntaxError: invalid syntax
>>> .x = 3
  File "<stdin>", line 1
    .x = 3
    ^
SyntaxError: invalid syntax
```

Now it is clear the usage of ` `` ` in `` `+.Complex` `` is to define a function with invalid name. Placing `.Complex` after + is related to S3 method dispatching, which will not be discussed here.

[16]https://docs.python.org/3.3/reference/lexical_analysis.html
[17]https://cran.r-project.org/doc/manuals/r-release/R-lang.html#Identifiers

1.7.1 Member accessibility in Python

In some programming languages the members (variable or methods) of a class can be declared with access modifiers which specify the accessibility or scope of a member. In Python, class members don't have explicit access modifiers, but it is still possible to specify the accessibility. By default, the class member can be accessed inside or outside the class definition. If the name of the member starts with a single underscore, the member should not be accessed outside the class definition by convention, but it is not enforced. If the name of the member starts with double underscore, the member name is mangled and thus the member can't be accessed by its original member outside the class definition. But inside the class definition these variables can always be accessed. Let's see the example below.

Python

```
>>> class ProtectedMemberClass:
...     """
...     _x and _func1 are protected and are recommended not to
    be accessed out of the class definition, but not enforced.
...     __y and __func2 are private and the names would be
    mangled
...     """
...     def __init__(self, val1, val2):
...         self._x = val1
...         self.__y = val2
...     def _func1(self):
...         print("protected _func1 called")
...     def __func2(self):
...         print("private __func2 called")
...     def func3(self):
...         # inside the class definition, we can access all
    these members
...         print("self._x is {0} and self.__y is{1}".format(
...             self._x, self.__y))
...         self._func1()
...         self.__func2()
...
>>> p = ProtectedMemberClass(0, 1)
>>> p._x
0
>>> p.__y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'ProtectedMemberClass' object has no attribute '
    __y'
>>> p._func1()
```

```
protected _func1 called
>>> p.__func2()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'ProtectedMemberClass' object has no attribute '
    __func2'
>>> p.func3()
self._x is 0 and self.__y is1
protected _func1 called
private __func2 called
```

In this example, an error is thrown when we try to access `__y` or `__func2` outside the class definition. But they are reachable within the class definition and these fields are usually called private fields.

1.8 MISCELLANEOUS

There are some items that I haven't discussed so far, which are also important in order to master R/Python.

1.8.1 Package/module installation

- Use `install.packages()` function in R
- Use R IDE to install packages
- Use `pip`[18] to install modules in Python

1.8.2 Virtual environment

Virtual environment is a tool to manage dependencies in Python. There are different ways to create virtual environments in Python. I suggest using the `venv` module shipped with Python 3. Unfortunately, there is nothing like a real virtual environment in R as far as I know although there are quite a few management tools/packages.

1.8.3 <- vs. =

If you have known R before, you probably have heard of the advice[19] to use <– to rather than = for value assignment. Let's see an example when <– makes a difference when we do value assignment.

```
> x=1
> a=list(x <- 2)
```

```
> x=1
> a=list(x = 2)
```

[18]https://packaging.python.org/tutorials/installing-packages/
[19]https://google.github.io/styleguide/Rguide.xml

```
> a
[[1]]
[1] 2

> x
[1] 2
```

```
> a
$x
[1] 2

> x
[1] 1
```

CHAPTER 2

More on R/Python Programming

We learned quite a few of the basics of R/Python programming in the previous chapter. I hope this chapter can be used as an intermediate level R/Python programming tutorial. There are a few topics to cover, including debugging, vectorization and some other useful features of R/Python.

2.1 WORK WITH R/PYTHON SCRIPTS

In Chapter 1 we were coding within the interactive mode of R/Python. When we are working on real-world projects, using an integrated development environment (IDE) is a more pragmatic choice. There are not many choices for R IDE, and among them RStudio[1] is the best one I have used so far. As for Python, I would recommend either Visual Studio Code[2] or PyCharm[3]. But of course, you could use any text editor to write R/Python scripts.

Let's write a simple script to print `Hello World!` in R/Python. I have made a directory `chapter2` on my disk, the R script is created with a text editor and then saved as `hello_world.R` and the Python script is created and saved as `hello_world.py`, inside the directory.

chapter2/hello_world.R

```r
print("Hello World!")
```

chapter2/hello_world.py

```python
print("Hello World!")
```

There are a few ways to run an R script. For example, we can run the script from the console with the `r −f filename` command. We can also use the `Rscript filename` command from the console. If we use `r −f filename` to run a script, the content of the script is also printed. With `Rscript filename`, the script content would not be

[1]https://www.rstudio.com/products/rstudio/
[2]https://code.visualstudio.com
[3]https://www.jetbrains.com/pycharm/

printed. Also, we can open the R interactive session and use the `source()` function. As for the Python script, we can run it from the console.

```
chapter2 $ls
hello_world.R hello_world.py
chapter2 $r -f hello_world.R
> print("Hello World!")
[1] "Hello World!"

chapter2 $r
> source('hello_world.R')
[1] "Hello World!"
```

```
chapter2 $ls
hello_world.R hello_world.py

chapter2 $python3.7
    hello_world.py
Hello World!
```

Below is another example of scripts, in which we first create a function to print a triangle and then call the function for printing.

chapter2/print_triangle.R

```
# define a function to print
    triangle
print_triangle =
    function(n){
  for (i in 1:n){
    print(strrep("*",i))
  }
}
# call the function for
    printing
print_triangle(5)
```

chapter2/print_triangle.py

```
# define a function to print
    triangle
def print_triangle(n):
  for i in range(n+1):
    print("*"*i)
# now we call the function
print_triangle(5)
```

Let's run these two scripts to see the results.

```
chapter2
    $Rscript print_triangle.R
[1] "*"
[1] "**"
[1] "***"
[1] "****"
[1] "*****"
```

```
chapter2 $python
    print_triangle.py

*
**
***
****
*****
```

The major reason for using scripts is that we don't want to limit ourselves by using

R/Python as basic calculators. In a script, we can put as many lines of R/Python code as we need. In Python, a script is also referred to as module. It is common to see a project consisting of multiple scripts to better organize the code structure. For example, in a data science project, data loading/cleaning functions could be organized inside one script, and model building functions could be organized in another script. Model testing/validation can also have its own script. It is also valid to put different scripts into directories. For example, below is the directory structure of a toy project.

```
toy_project/
|-- src
|   |-- data_processing
|   |   |-- data_clean.R
|   |   |__ data_clean.py
|   |-- model_building
|   |   |-- linear_model.R
|   |   |-- linear_model.py
|   |   |-- tree_model.R
|   |   |__ tree_model.py
|   |__ model_validation
|__ test
```

We could put scripts into these directories accordingly. Please note that the toy project is just for illustration and your projects don't need to follow this toy project structure.

An immediate problem of putting functions into different scripts is working out how R/Python could find/use a function defined in another script. For example, in the `linear_model.R` script we need to call a function `impute` defined in `data_clean.R`. The solution is the `source` function in R and `import` statement in Python. In R, sourcing a file is straightforward by using the file path. However, there are some caveats using `source` in Python, which are out of this book's scope.

When we use an IDE (RStudio or Visual Studio Code), not only can we work on scripts, but also we can work on projects. But a project has nothing to do with the programming language itself. A project is IDE-specific. For example, with RStudio we can create an RStudio project to organize the scripts as well as some configurations[4]. To work with an IDE-specific project, the IDE should be installed. But running the scripts doesn't require any IDE since we can even run the scripts with a command line interface.

[4]https://support.rstudio.com/hc/en-us/articles/200526207-Using-Projects

2.2 DEBUGGING IN R/PYTHON

Debugging is one of the most important aspects of programming. What is debugging in programming? The programs we write might include errors/bugs and debugging is a step-by-step process to find and remove the errors/bugs in order to get the desired results.

If you are smart enough or the bugs are evident enough then you can debug the program on your mind without using a computer at all. But in general we need some tools/techniques to help us with debugging.

2.2.1 Print

Most programming languages provide the functionality of printing, which is a natural way of debugging. By trying to place `print` statements at different positions we may finally catch the bugs. When I use `print` to debug, it's feeling like playing the game of Minesweeper. In Python, there is a module called `logging`[5] which could be used for debugging like the `print` function, but in a more elegant fashion.

2.2.2 Browser in R and pdb in Python

In R, there is a function `browser()` which interrupts the execution and allows the inspection of the current environment. Similarly, there is a module `pdb` in Python that provides more debugging features. We will only focus on the basic usages of `browser()` and the `set_trace()` function in `pdb` module. The essential difference between debugging using `print()` and `browser()` and `set_trace()` is that the latter function allows us to debug in an interactive mode.

Let's write a function which takes a sorted vector/list `v` and a target value `x` as input and returns the leftmost index `pos` of the sorted vector/list so that `v[pos]>=x`. Since `v` is already sorted, we may simply loop through it from left to right to find `pos`.

chapter2/find_pos.R

```r
find_pos=function(v,x){
  for (i in 1:length(v)){
    if (v[i]>=x){
      return(i)
    }
  }
}

v=c(1,2,5,10)
print(find_pos(v,-1))
print(find_pos(v,4))
print(find_pos(v,11))
```

chapter2/find_pos.py

```python
def find_pos(v,x):
  for i in range(len(v)):
    if v[i]>=x:
      return i

v=[1,2,5,10]
print(find_pos(v,-1))
print(find_pos(v,4))
print(find_pos(v,11))
```

[5]https://docs.python.org/3/library/logging.html

Now let's run these two scripts.

```
chapter2 $r
> source('find_pos.R')
[1] 1
[1] 3
NULL
```

```
chapter2 $python3.7 find_pos
    .py
0
2
None
```

When `x=11`, the function returns `NULL` in R and `None` in Python because there is no such element in `v` larger than `x`. The implementation above is trivial, but not efficient. If you have some background in data structures and algorithms, you probably know this question can be solved by binary search. The essential idea of binary search comes from the concept of divide-and-conquer[6]. Since `v` is already sorted, we may divide it into two partitions by cutting it from the middle, and then we get the left partition and the right partition. `v` is sorted implying that both the left partition and the right partition are also sorted. If the target value `x` is larger than the rightmost element in the left partition, we can just discard the left partition and search `x` within the right partition. Otherwise, we can discard the right partition and search `x` within the left partition. Once we have determined which partition to search, we may apply the idea recursively so that in each step we reduce the size of `v` by half. If the length of `v` is denoted as `n`, in terms of big O notation[7], the run time complexity of binary search is $\mathcal{O}(\log n)$, compared with $\mathcal{O}(n)$ of the for-loop implementation.

The code below implements the binary search solution to our question. (It is more intuitive to do it with recursion but here I write it with iteration since tail recursion optimization[8] in R/Python is not supported.)

R chapter2/find_binary_search_buggy.R

```r
binary_search_buggy=function(v, x){
  start = 1
  end = length(v)
  while (start<end){
    mid = (start+end) %/% 2 # %/% is the floor division operator
    if (v[mid]>=x){
      end = mid
    }else{
      start = mid+1
    }
  }
```

[6]https://en.wikipedia.org/wiki/Divide-and-conquer_algorithm
[7]https://en.wikipedia.org/wiki/Big_O_notation
[8]https://en.wikipedia.org/wiki/Tail_call

```
  return(start)
}
v=c(1,2,5,10)
print(binary_search_buggy(v,−1))
print(binary_search_buggy(v,5))
print(binary_search_buggy(v,11))
```

Python chapter2/find_binary_search_buggy.py

```python
def binary_search_buggy(v, x):
  start, end = 0, len(v)−1
  while start<end:
    mid = (start+end)//2  # // is the floor division operator
    if v[mid]>=x:
      end = mid
    else:
      start = mid+1
  return start

v=[1,2,5,10]
print(binary_search_buggy(v,−1))
print(binary_search_buggy(v,5))
print(binary_search_buggy(v,11))
```

Now let's run these two binary_search scripts.

```
chapter2 $r
> source('
   binary_search_buggy.R')
[1] 1
[1] 3
[1] 4
```

```
chapter2 $python3.7
    binary_search_buggy.py
0
2
3
```

The binary search solutions don't work as expected when `x=11`. We write two new scripts.

R chapter2/find_binary_search_buggy_debug.R

```r
binary_search_buggy=function(v, x){
  browser()
  start = 1
  end = length(v)
  while (start<end){
    mid = (start+end)
    if (v[mid]>=x){
      end = mid
    }else{
      start = mid+1
    }
  }
  return(start)
}
v=c(1,2,5,10)
print(binary_search_buggy(v, 11))
```

Python chapter2/find_binary_search_buggy_debug.py

```
from pdb import set_trace
def binary_search_buggy(v, x):
  set_trace()
  start, end = 0, len(v)-1
  while start<end:
    mid = (start+end)//2
    if v[mid]>=x:
      end = mid
    else:
      start = mid+1
  return start

v=[1,2,5,10]
print(binary_search_buggy(v,11))
```

Let's try to debug the programs with the help of `browser()` and `set_trace()`.

R

```
> source('binary_search_buggy_debug.R')
Called from: binary_search_buggy(v, 11)
Browse[1]> ls()
[1] "v" "x"
Browse[1]> n
debug at binary_search_buggy_debug.R#3: start = 1
Browse[2]> n
debug at binary_search_buggy_debug.R#4: end = length(v)
Browse[2]> n
debug at binary_search_buggy_debug.R#5: while (start < end) {
    mid = (start + end)%/%2
    if (v[mid] >= x) {
        end = mid
    }
    else {
        start = mid + 1
    }
}
Browse[2]> n
debug at binary_search_buggy_debug.R#6: mid = (start + end)%/%2
Browse[2]> n
debug at binary_search_buggy_debug.R#7: if (v[mid] >= x) {
    end = mid
} else {
    start = mid + 1
}
Browse[2]> n
debug at binary_search_buggy_debug.R#10: start = mid + 1
Browse[2]> n
debug at binary_search_buggy_debug.R#5: (while) start < end
Browse[2]> n
debug at binary_search_buggy_debug.R#6: mid = (start + end)%/%2
Browse[2]> n
debug at binary_search_buggy_debug.R#7: if (v[mid] >= x) {
    end = mid
} else {
    start = mid + 1
}
Browse[2]> n
debug at binary_search_buggy_debug.R#10: start = mid + 1
Browse[2]> n
debug at binary_search_buggy_debug.R#5: (while) start < end
Browse[2]> start
[1] 4
Browse[2]> n
debug at binary_search_buggy_debug.R#13: return(start)
Browse[2]> n
[1] 4
```

In the R code snippet above, we placed the `browser()` function on the top of the function `binary_search_buggy`. Then when we call the function we enter into the debugging environment. By calling `ls()` we see all variables in the current debugging scope, i.e., `v, x`. Typing `n` will evaluate the next statement. After typing `n` a few times, we finally exit from the while loop because `start = 4` such that `start < end` is FALSE. As a result, the function just returns the value of start, i.e., 4. To exit from the debugging environment, we can type `Q`; to continue the execution we can type `c`.

The root cause is that we didn't deal with the corner case when the target value `x` is larger than the last/largest element in `v` correctly.

Let's debug the Python function using `pdb` module.

Python

```
chapter2 $python3.7 binary_search_buggy_debug.py
> chapter2/binary_search_buggy_debug.py(4)binary_search_buggy()
-> start,end = 0,len(v)-1
(Pdb) n
> chapter2/binary_search_buggy_debug.py(5)binary_search_buggy()
-> while start<end:
(Pdb) l
  1   from pdb import set_trace
  2   def binary_search_buggy(v,x):
  3     set_trace()
  4     start,end = 0,len(v)-1
  5  ->   while start<end:
  6       mid = (start+end)//2
  7       if v[mid]>=x:
  8         end = mid
  9       else:
 10         start = mid+1
 11     return start
(Pdb) b 7
Breakpoint 1 at chapter2/binary_search_buggy_debug.py:7
(Pdb) c
> chapter2/binary_search_buggy_debug.py(7)binary_search_buggy()
-> if v[mid]>=x:
(Pdb) c
> chapter2/binary_search_buggy_debug.py(7)binary_search_buggy()
-> if v[mid]>=x:
(Pdb) mid
2
(Pdb) n
> chapter2/binary_search_buggy_debug.py(10)binary_search_buggy()
-> start = mid+1
(Pdb) n
> chapter2/binary_search_buggy_debug.py(5)binary_search_buggy()
-> while start<end:
(Pdb) start
3
(Pdb) n
```

```
> chapter2/binary_search_buggy_debug.py(11)binary_search_buggy()
-> return start
```

Similar to R, command `n` would evaluate the next statement in `pdb`. Typing command `l` would show the current line of current execution. Command `b line_number` would set the corresponding line as a break point; and `c` would continue the execution until the next breakpoint (if it exists).

In R, besides the `browser()` function there are a pair of functions `debug()`and `undebug()` which are also very handy when we try to debug a function; especially when the function is wrapped in a package. More specifically, the `debug` function will invoke the debugging environment whenever we call the function to debug. See the example below which explains how we invoke the debugging environment for the `sd` function (standard deviation calculation).

R

```
> x=c(1,1,2)
> debug(sd)
> sd(x)
debugging in: sd(x)
debug: sqrt(var(if (is.vector(x) || is.factor(x)) x else as.
    double(x),
     na.rm = na.rm))
Browse[2]> ls()
[1] "na.rm" "x"
Browse[2]> Q
> undebug(sd)
> sd(x)
[1] 0.5773503
```

The binary_search solutions are fixed below.

R chapter2/find_binary_search.R

```
binary_search=function(v,x){
  if (x>v[length(v)]){return(NULL)}
  start = 1
  end = length(v)
  while (start<end){
    mid = (start+end)
    if (v[mid]>=x){
      end = mid
    }else{
      start = mid+1
    }
  }
  return(start)
}
```

Python chapter2/find_binary_search.py

```python
def binary_search(v, x):
  if x>v[-1]: return
  start, end = 0, len(v)-1
  while start<end:
    mid = (start+end)//2
    if v[mid]>=x:
      end = mid
    else:
      start = mid+1
  return start
```

2.3 BENCHMARKING

By benchmarking, I mean measuring the entire operation time of a piece of program. There is another term called profiling which is related to benchmarking. But profiling is more complex since it commonly aims at understanding the behavior of the program and optimizing the program in terms of time elapsed during the operation.

In R, I like using the `microbenchmark` package. And in Python, `timeit` module is a good tool to use when we want to benchmark small bits of Python code.

As mentioned before, the run time complexity of binary search is better than that of a for-loop search. We can do benchmarking to compare the two algorithms.

R chapter2/benchmark.R

```r
library(microbenchmark)
source('binary_search.R')
source('find_pos.R')

v=1:10000

# call each function 1000 times;
# each time we randomly select an integer as the target value

# for-loop solution
set.seed(2019)
print(microbenchmark(find_pos(v, sample(10000,1)),times=1000))
# binary-search solution
```

```
set.seed(2019)
print(microbenchmark(binary_search(v, sample(10000,1)),times
    =1000))
```

In the R code above, `times=1000` means we want to call the function `1000` times in the benchmarking process. The `sample()` function is used to draw samples from a set of elements. Specifically, we pass the argument `1` to `sample()` to draw a single element. It's the first time we use `set.seed()` function in this book. In R/Python, we draw random numbers based on the pseudorandom number generator (PRNG) algorithm[9]. The sequence of numbers generated by PRNG is completed determined by an initial value, i.e., the seed. Whenever a program involves the usage of PRNG, it is better to set the seed in order to get replicable results (see the example below).

R

```
> set.seed(2019)
> rnorm(1)
[1] 0.7385227
> set.seed(2019)
> rnorm(1)
[1] 0.7385227
```

R

```

> set.seed(2019)
> rnorm(1)
[1] 0.7385227
> rnorm(1)
[1] -0.5147605
```

Now let's run the R script to see the benchmarking result.

R

```
> source('benchmark.R')
Unit: microseconds
                              expr  min       lq      mean    median
                                        uq      max
 find_pos(v, sample(10000, 1)) 3.96 109.5385 207.6627 207.5565
    307.8875 536.171
 neval
  1000
Unit: microseconds
                                   expr   min      lq    mean median
                                            uq     max
 binary_search(v, sample(10000, 1)) 5.898 6.3325 14.2159 6.6115
    7.3635 6435.57
 neval
  1000
```

[9]https://en.wikipedia.org/wiki/Pseudorandom_number_generator

The binary_search solution is much more efficient based on the benchmarking result. Doing the same benchmarking in Python is a bit complicated.

Python chapter2/benchmark.py

```python
from binary_search import binary_search
from find_pos import find_pos
import timeit
import random

v=list(range(1,10001))

def test_for_loop(n):
  random.seed(2019)
  for _ in range(n):
    find_pos(v, random.randint(1,10000))

def test_bs(n):
  random.seed(2019)
  for _ in range(n):
    binary_search(v, random.randint(1,10000))

# for-loop solution
print(timeit.timeit('test_for_loop(1000)',setup='from __main__ 
    import test_for_loop',number=1))
# binary_search solution
print(timeit.timeit('test_bs(1000)',setup='from __main__ import 
    test_bs',number=1))
```

The most interesting part of the Python code above is `from __main__ import`. Let’s ignore it for now; we will revisit it later.

Below is the benchmarking result in Python (the unit is second).

Python

```
chapter2 $python3 benchmark.py
0.284618441
0.003966589000000002
```

2.4 VECTORIZATION

In parallel computing, automatic vectorization[10] means a program in a scalar implementation is converted to a vector implementation which processes multiple pairs of

[10]https://en.wikipedia.org/wiki/Automatic_vectorization

operands simultaneously by compilers that feature auto-vectorization. For example, let's calculate the element-wise sum of two arrays `x` and `y` of the same length in C programming language.

```c
int x[4] = {1,2,3,4};
int y[4] = {0,1,2,3};
int z[4];
for (int i=0;i<4;i++){
    z[i]=x[i]+y[i];
}
```

The C code above might be vectorized by the compiler so that the actual number of iterations performed could be less than 4. If 4 pairs of operands are processed at once, there would be only 1 iteration. Automatic vectorization may make the program run much faster in some languages like C. However, when we talk about vectorization in R/Python, it is different from automatic vectorization. Vectorization in R/Python usually refers to the human effort paid to avoid for-loops. First, let's see some examples of how for-loops may slow your programs in R/Python.

R

chapter2/vectorization_1.R

```r
library(microbenchmark)

# generate n standard normal r.v
rnorm_loop = function(n){
x=rep(0,n)
for (i in 1:n) {x[i]=rnorm(1)}
}

rnorm_vec = function(n){
x=rnorm(n)
}

n=100
# for loop
print(microbenchmark(rnorm_loop(n),times=1000))
# vectorize
print(microbenchmark(rnorm_vec(n),times=1000))
```

Running the R code produces the following result on my local machine.

R

```
> source('vectorization_1.R')
```

```
Unit: microseconds
          expr     min       lq      mean   median       uq      max
     neval
 rnorm_loop(n) 131.622 142.699 248.7603 145.3995 270.212 16355.6
     1000
Unit: microseconds
         expr   min    lq     mean median    uq      max neval
 rnorm_vec(n) 6.696 7.128 10.87463  7.515 8.291 2422.338  1000
```

Python

```python
import timeit
import numpy as np

def rnorm_for_loop(n):
  x=[0]*n # create a list with n 0s
  np.random.seed(2019)
  for _ in range(n):
    np.random.normal(0,1,1)

def rnorm_vec(n):
  np.random.seed(2019)
  x = np.random.normal(0,1,n)

print("for loop")
print(f'{timeit.timeit("rnorm_for_loop(100)",setup="from
    __main__ import rnorm_for_loop",number=1000):.6f} seconds')
print("vectorized")
print(f'{timeit.timeit("rnorm_vec(100)",setup="from __main__
    import rnorm_vec",number=1000):.6f} seconds')
```

Please note that in this Python example we are using the `random` submodule of numpy module instead of the built-in `random` module since `random` module doesn't provide the vectorized version of random number generation function. Running the Python code results in the following on my local machine.

Python

```
chapter2 $python3.7 vectorization_1.py
for loop
0.258466 seconds
vectorized
0.008213 seconds
```

Also, the `timeit.timeit` measures the total time to run the main statements `number` times, but not the average time per run.

In either R or Python, the vectorized version of random normal random variable (r.v.) is significantly faster than the scalar version. It is worth noting the usage of the `print(f'')` statement in the Python code, which is different from the way that we print the object of `Complex` class in Chapter 1. In the code above, we use the `f−string`[11] which is a literal string prefaced with `'f'` containing expressions inside `{}` which would be replaced with their values. `f−string` is a feature introduced since Python 3.6.

It's also worth noting that lots of built-in functions in R are already vectorized, such as the basic arithmetic operators, comparison operators, `ifelse()`, element-wise logical operators `&,|`. But the logical operators `&&, ||` are not vectorized.

In addition to vectorization, there are also some built-in functions which may help to avoid the usages of for-loops. For example, in R we might be able to use the `apply` family of functions to replace for-loops; and in Python the `map()` function can also be useful. In the Python `pandas` module, there are also many usages of `map/apply` methods. But in general the usage of `apply/map` functions has little or nothing to do with performance improvement. However, appropriate usages of such functions may help with the readability of the program. Compared with the `apply` family of functions in R, I think the `do.call()` function is more useful in practice. We will spend some time in `do.call()` later.

Application - Biham–Middleton–Levine (BML) traffic model

Considering the importance of vectorization in scientific programming, let's try to get more familiar with vectorization thorough the Biham–Middleton–Levine (BML) traffic model[12]. The BML model is very important in modern studies of traffic flow since it exhibits a sharp phase transition from free flowing status to a fully jammed status. A simplified BML model could be characterized as follows:

- Initialized on a 2-D lattice, each site of which is either empty or occupied by a colored particle (blue or red);
- Particles are distributed randomly through the initialization according to a uniform distribution; the two colors of particles are equally distributed.
- On even time steps, all blue particles attempt to move one site up and an attempt fails if the site to occupy is not empty;
- On Odd time steps, all red particles attempt to move one site right and an attempt fails if the site to occupy is not empty;
- The lattice is assumed periodic which means when a particle moves out of the lattice, it will move into the lattice from the opposite side.

[11]https://www.python.org/dev/peps/pep-0498/
[12]https://en.wikipedia.org/wiki/Biham-Middleton-Levine_traffic_model

The BML model specified above is implemented in both R/Python as follows to illustrate the usage of vectorization.

R chapter2/BML.R

```r
library(R6)
BML = R6Class(
  "BML",
  public = list(
    # alpha is the parameter of the uniform distribution to
        control particle distribution's density
    # m*n is the dimension of the lattice
    alpha = NULL,
    m = NULL,
    n = NULL,
    lattice = NULL,
    initialize = function(alpha, m, n) {
      self$alpha = alpha
      self$m = m
      self$n = n
      self$initialize_lattice()
    },
    initialize_lattice = function() {
      # 0 -> empty site
      # 1 -> blue particle
      # 2 -> red particle
      u = runif(self$m * self$n)
      # the usage of L is to make sure the elements in particles
          are of type integer;
      # otherwise they would be created as double
      particles = rep(0L, self$m * self$n)
      # doing inverse transform sampling
      particles[(u > self$alpha) &
                  (u <= (self$alpha + 1.0) / 2)] = 1L
      particles[u > (self$alpha + 1.0) / 2] = 2L
      self$lattice = array(particles, c(self$m, self$n))
    },
    odd_step = function() {
      blue.index = which(self$lattice == 1L, arr.ind = TRUE)
      # make a copy of the index
      blue.up.index = blue.index
      # blue particles move 1 site up
      blue.up.index[, 1] = blue.index[, 1] - 1L
      # periodic boundary condition
      blue.up.index[blue.up.index[, 1] == 0L, 1] = self$m
      # find which moves are feasible
      blue.movable = self$lattice[blue.up.index] == 0L
      # move blue particles one site up
      # drop=FALSE prevents the 2D array degenerates to 1D array
      self$lattice[blue.up.index[blue.movable, , drop = FALSE]]
         = 1L
      self$lattice[blue.index[blue.movable, , drop = FALSE]] = 0
         L
    },
    even_step = function() {
      red.index = which(self$lattice == 2L, arr.ind = TRUE)
      # make a copy of the index
      red.right.index = red.index
      # red particles move 1 site right
      red.right.index[, 2] = red.index[, 2] + 1L
      # periodic boundary condition
      red.right.index[red.right.index[, 2] == (self$n + 1L), 2]
         = 1
      # find which moves are feasible
      red.movable = self$lattice[red.right.index] == 0L
      # move red particles one site right
      self$lattice[red.right.index[red.movable, , drop = FALSE]]
          = 2L
      self$lattice[red.index[red.movable, , drop = FALSE]] = 0L
    }
  )
)
```

Now we can create a simple BML system on a 5 × 5 lattice using the R code above.

R

```
> source('BML.R')
> set.seed(2019)
> bml=BML$new(0.4,5,5)
> bml$lattice
     [,1] [,2] [,3] [,4] [,5]
[1,]    2    0    2    1    1
[2,]    2    2    1    0    1
[3,]    0    0    0    2    2
[4,]    1    0    0    0    0
[5,]    0    1    1    1    0
> bml$odd_step()
> bml$lattice
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    2    0    2    1    0
[2,]    2    2    1    0    1
[3,]    1    0    0    2    2
[4,]    0    1    1    1    0
[5,]    0    0    0    0    1
> bml$even_step()
> bml$lattice
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    2    2    1    0
[2,]    2    2    1    0    1
[3,]    1    0    0    2    2
[4,]    0    1    1    1    0
[5,]    0    0    0    0    1
```

In the initialization step, we used the inversion sampling approach[13] to generate the status of each site. Inversion sampling method is a basic but powerful approach to generate r.v. from any probability distribution given its cumulative distribution function (CDF). We will talk more about sampling technique in Chapter 4.

Python

```python
import numpy as np

class BML:
    def __init__(self, alpha, m, n):
        self.alpha = alpha
        self.shape = (m, n)
        self.initialize_lattice()

    def initialize_lattice(self):
        u = np.random.uniform(0.0, 1.0, self.shape)
        # instead of using default list, we use np.array to
            create the lattice
        self.lattice = np.zeros_like(u, dtype=int)
        # the parentheses below can't be ignored
        self.lattice[(u > self.alpha) & (u <= (1.0+self.alpha)
            /2.0)] = 1
        self.lattice[u > (self.alpha+1.0)/2.0] = 2

    def odd_step(self):
        # please note that np.where returns a tuple which is
            immutable
```

[13]https://en.wikipedia.org/wiki/Inverse_transform_sampling

```
        blue_index = np.where(self.lattice == 1)
        blue_index_i = blue_index[0] - 1
        blue_index_i[blue_index_i < 0] = self.shape[0]-1
        blue_movable = self.lattice[(blue_index_i, blue_index
            [1])] == 0
        self.lattice[(blue_index_i[blue_movable],
                      blue_index[1][blue_movable])] = 1
        self.lattice[(blue_index[0][blue_movable],
                      blue_index[1][blue_movable])] = 0

    def even_step(self):
        red_index = np.where(self.lattice == 2)
        red_index_j = red_index[1] + 1
        red_index_j[red_index_j == self.shape[1]] = 0
        red_movable = self.lattice[(red_index[0], red_index_j)]
            == 0
        self.lattice[(red_index[0][red_movable],
                      red_index_j[red_movable])] = 2
        self.lattice[(red_index[0][red_movable],
                      red_index[1][red_movable])] = 0
```

The Python implementation is also given.

R

```
>>> import numpy as np
>>> np.random.seed(2019)
>>> from BML import BML
>>> bml=BML(0.4,5,5)
>>> bml.lattice
array([[2, 0, 1, 1, 2],
       [0, 2, 2, 2, 1],
       [1, 0, 0, 2, 0],
       [2, 0, 1, 0, 2],
       [1, 1, 0, 2, 1]])
>>> bml.odd_step()
>>> bml.lattice
array([[2, 0, 0, 1, 2],
       [1, 2, 2, 2, 1],
       [0, 0, 1, 2, 0],
       [2, 1, 0, 0, 2],
       [1, 0, 1, 2, 1]])
>>> bml.even_step()
>>> bml.lattice
array([[0, 2, 0, 1, 2],
       [1, 2, 2, 2, 1],
       [0, 0, 1, 0, 2],
       [2, 1, 0, 0, 2],
       [1, 0, 1, 2, 1]])
```

Please note that although we have imported `numpy` in BML.py, we import it again in the code above in order to set the random seed. If we change the line to `from BML import *`, we don't need to import `numpy` again. But it is not recommended to `import *` from a module.

2.5 EMBARRASSINGLY PARALLELISM IN R/PYTHON

According to the explanation in wikipedia[14], single-threading is the processing of one command at a time, and its opposite is multithreading. A process is the instance of a computer program executed by one or many threads[15]. Multithreading is not the same as parallelism. In a single processor environment, the processor can switch execution between threads, which results in concurrent execution. However, it is possible a process with multithreads runs on a multiprocessor environment and each of the threads on a separate processor, which results in parallel execution.

Both R and Python are single-threaded. In Python, there is a `threading` package[16], which supports multithreading on a single core. It may suit some specific tasks. For example, in web scraping multithreading on a single core may be able to speed up the program if the download time is in excess of the CPU processing time.

Now let's talk about embarrassingly parallelism by multiprocessing. An embarrassingly parallel problem is one where little or no effort is needed to separate the problem into a number of parallel tasks[17]. In R there are various packages supporting multiprocessing on multiple CPU cores, for example, the `parallel` package, which is my favorite one. In Python, there are also some available modules, such as `multiprocessing`, `joblib` and `concurrent.futures`.

Let's see an application of embarrassingly parallelism to calculate π using Monte Carlo simulation[18].

Application - Monte Carlo simulation to estimate π via parallelization

Monte Carlo simulation provides a simple and straightforward way to estimate π. We know the area of a circle with radius 1 is just π. Thus, we can convert the original problem of π calculation to a new problem, i.e., how to calculate the area of a circle with radius 1. We also know the area of a square with side length 2 is equal to 4. Thus, π can be calculated as $4r_{c/s}$ where $r_{c/s}$ denotes the ratio of areas of a circle with radius 1 and a square with side length 2. Now the problem is how to calculate the ratio $r_{c/s}$? When we randomly throw

[14] https://en.wikipedia.org/wiki/Thread_(computing)
[15] https://en.wikipedia.org/wiki/Process_(computing)
[16] https://docs.python.org/3.7/library/threading.html
[17] https://en.wikipedia.org/wiki/Embarrassingly_parallel
[18] https://en.wikipedia.org/wiki/Monte_Carlo_method

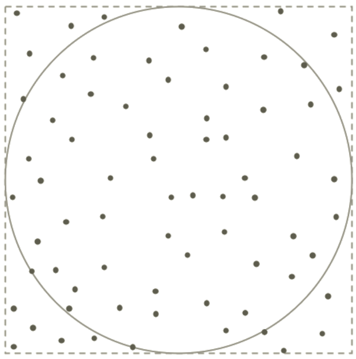


Figure 2.1: Generate points within a square and count how many times these points fall into the inscribed circle

n points into the square and m of these points fall into the inscribed circle (see Figure 2.1), then we can estimate the ratio as m/n. As a result, a natural estimate of π is $4m/n$. This problem is an embarrassingly parallel problem by its nature. Let's see how we implement the idea in R/Python.

R chapter2/pi.R

```r
library(parallel)
count_inside_point = function(n){
  m = 0
  for (i in 1:n){
    p_x = runif(1, -1, 1)
    p_y = runif(1, -1, 1)
    if (p_x^2 + p_y^2 <=1){
      m = m+1
    }
  }
  m
}

# now let's use the mcapply for parallelization
generate_points_parallel = function(n){
  # detectCores() returns the number of cores available
  # we assign the task to each core
  unlist(mclapply(X = rep(n %/% detectCores(), detectCores()),
     FUN=count_inside_point))
}

# now let's use vectorization
generate_points_vectorized = function(n){
  p = array(runif(n*2,-1,1), c(n,2))
  sum((p[,1]^2+p[,2]^2)<=1)
}

pi_naive = function(n) cat('naive: pi -', 4*count_inside_point(n
    )/n, '\n')

pi_parallel = function(n) cat('parallel: pi -', 4*sum(
    generate_points_parallel(n))/n, '\n')

pi_vectorized = function(n) cat('vectorized: pi -', 4*sum(
    generate_points_vectorized(n))/n, '\n')
```

In the above R code snippet, we use the function `mclapply` which is not currently available on some operation systems[19]. When it is not available, we may consider using `parLapply` instead.

Python chapter2/pi.py

```python
# now let's try the parallel approach
# each process uses the same seed, which is not desired
def generate_points_parallel(n):
    pool = mp.Pool()
    # we ask each process to generate n//mp.cpu_count() points
    return pool.map(count_inside_point, [n//mp.cpu_count()]*mp.
        cpu_count())

# set seed for each process
# first, let's define a helper function
def helper(args):
    n, s = args
    seed(s)
    return count_inside_point(n)

def generate_points_parallel_set_seed(n):
    pool = mp.Pool() # we can also specify the number of
        processes by Pool(number)
    # we ask each process to generate n//mp.cpu_count() points
    return pool.map(helper, list(zip([n//mp.cpu_count()]*mp.
        cpu_count(),range(mp.cpu_count()))))

# another optimization via vectorization
def generate_points_vectorized(n):
```

[19]https://stat.ethz.ch/R-manual/R-devel/library/parallel/doc/parallel.pdf

```
    p = uniform(−1, 1, size=(n,2))
    return np.sum(np.linalg.norm(p, axis=1) <= 1)

def pi_naive(n):
    print(f'pi: {count_inside_point(n)/n*4:.6f}')

def pi_parallel(n):
    print(f'pi: {sum(generate_points_parallel_set_seed(n))/n
        *4:.6f}')

def pi_vectorized(n):
    print(f'pi: {generate_points_vectorized(n)/n*4:.6f}')
```

In the above Python code snippet, we defined a function `generate_points_parallel` which returns a list containing the numbers of inside points. But the problem with this function is that each process in the pool shares the same random state. As a result, we will obtain a list of duplicated numbers by calling this function. To overcome the issue, we defined another function `generate_points_parallel_set_seed`. Let's see the results of our estimation for π running on a laptop.

R

```
> source('pi.R')
> system.time(pi_naive(1e6))
naive: pi − 3.144592
   user  system elapsed
  8.073   1.180   9.333
> system.time(pi_parallel(1e6))
parallel: pi − 3.1415
   user  system elapsed
  4.107   0.560   4.833
> system.time(pi_vectorized(1e6))
vectorized: pi − 3.141224
   user  system elapsed
  0.180   0.031   0.214
```

Python

```
>>> import timeit
>>> from pi import pi_naive, pi_parallel, pi_vectorized
>>> print(f'naive − {timeit.timeit("pi_naive(1000000)",setup="
    from __main__ import pi_naive",number=1):.6f} seconds')
pi: 3.141056
naive − 4.363822 seconds
```

```
>>> print(f'parallel - {timeit.timeit("pi_parallel(1000000)",
    setup="from __main__ import pi_parallel",number=1):.6f}
    seconds')
pi: 3.141032
parallel - 2.204697 seconds
>>> print(f'vectorized - {timeit.timeit("pi_vectorized(1000000)
    ",setup="from __main__ import pi_vectorized",number=1):.6f}
    seconds')
pi: 3.139936
vectorized - 0.148950 seconds
```

We see the winner in this example is vectorization, and the parallel solution is better than the naive solution. However, when the problem cannot be vectorized we may use parallelization to achieve better performance.

2.6 EVALUATION STRATEGY

R is a lazy programming language since it uses lazy evaluation by default [13]. Lazy evaluation strategy delays the evaluation of an expression until its value is needed. When an expression in R is evaluated, it follows an outermost reduction order in general. But Python is not a lazy programming language and the expression in Python follows an innermost reduction order. The code snippets below illustrate the innermost reduction vs. outermost reduction.

R

```
> divide = function(x, y){
    stopifnot(y!=0);return(x
    /y)}
> power = function(x, p){
    if (p==0) 1 else x^p}
> power(divide(1,0),2)
Error in divide(1, 0) : y !
    = 0 is not TRUE
> power(divide(1,0),0)
[1] 1
```

Python

```
>>> def divide(x, y):
    return x/y
>>> def power(x, p):
    return 1 if p==0
    else x**p
>>> power(divide(1,0),0)
Traceback (most recent call
    last):
  File "<stdin>", line 1,
    in <module>
  File "<stdin>", line 1,
    in divide
ZeroDivisionError: division
    by zero
```

Because of the outermost reduction order, the R code snippet evaluates the function `power` first and since if the second argument is 0 the first argument is not required to evaluate. Thus, the function call returns 1. But the Python code snippet first evaluates the function call of `divide` and an exception is raised because of division by zero.

Although Python is an eager language, it is possible to simulate the lazy evaluation behavior. For example, the code inside a generator[20] is evaluated when the generator is consumed but not evaluated when it is created. We can create a sequence of Fibonacci numbers with the help of `generator`.

Python

```
>>> def fib():
...     '''
...     a generator to create Fibonacci numbers less than 10
...     '''
...     a, b = 0, 1
...     while a<10:
...         a, b = b, a+b
...         yield a
...
>>> f = fib()
>>> print(f)
<generator object fib at 0x116dfc570>
>>> for e in f: print(e)
...
1
1
2
3
5
8
13
>>>
```

In the code snippet above, we create a generator which generates the sequence of Fibonacci numbers less than 10. When the generator is created, the sequence is not generated immediately. When we consume the generator in a loop, each element is then generated as needed. It is also common to consume the generator with the `next` function.

Python

```
>>> f = fib()
>>> next(f)
1
>>> next(f)
1
>>> next(f)
2
```

[20]https://www.python.org/dev/peps/pep-0255

```
>>> next(f)
3
>>> next(f)
5
```

2.7 SPEED UP WITH C/C++ IN R/PYTHON

The purpose of this section is not to teach C/C++. If you know how to write C/C++ then it may help to improve the performance of R/Python program.

It is recommended to use vectorization technique to speed up your program whenever possible. But what if it's infeasible to vectorize the algorithm? One potential option is to use C/C++ in the R/Python code. According to my limited experience, it is much easier to use C/C++ in R with the help of `Rcpp` package than in Python. In Python, more options are available but not as straightforward as the solution of `Rcpp`. In this section, let's use `Cython`[21].

Actually, `Cython` itself is a programming language written in Python and C. In `Rcpp` we can write C++ code directly and use it in native R code. With the help of `Cython`, we can write python-like code which is able to be compiled to C or C++ and automatically wrapped into python-importable modules. `Cython` could also wrap independent C or C++ code into python-importable modules.

Let's see how to calculate Fibonacci numbers with these tools. The first step is to install `Rcpp` package in R and the `Cython` module (use `pip`) in Python. Once they are installed, we can write the code in C++ directly for R. As for `Cython`, we write the python-like code which will be compiled to C/C++ later.

chapter2/Fibonacci.cpp

```cpp
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
int Fibonacci(int n) {
  if (n==1 or n==2){
    return 1;
  }
  return Fibonacci(n-1)+
      Fibonacci(n-2);
}
```

chapter2/Fibonacci.pyx

```python
#!python
#cython: language_level=3

cdef int Fibonacci_c(int n):
    if n==1 or n==2:
        return 1
    return Fibonacci_c(n-1)
        + Fibonacci_c(n-2)

def Fibonacci_static(n):
    return Fibonacci_c(n)

def Fibonacci(n):
    if n==1 or n==2:
        return 1
```

[21]https://en.wikipedia.org/wiki/Cython

```
    return Fibonacci(n-1) +
        Fibonacci(n-2)
```

It's worth noting the extension of the `Cython` file above is `pyx`, not `py`. And in `Fibonacci.pyx`, the function we defined follows the native Python syntax. But in `Cython` we can add static typing declarations which are often helpful to improve the performance, although not it's mandatory. The keyword `cdef` makes the function `Fibonacci_c` invisible to Python and thus we have to define the `Fibonacci_static` function as a wrapper which can be imported into Python. The function `Fibonacci` is not static typed for benchmarking.

Let's also implement the same functions in native R/Python for benchmarking purposes.

chapter2/Fibonacci.R

```
Fibonacci_native =
    function(n){
  if (n==1 || n==2){
    1
  }else {
    Fibonacci_native(n-1) +
        Fibonacci_native(n-2)
  }
}
```

chapter2/Fibonacci.py

```
def Fibonacci_native(n):
    if n==1 or n==2:
        return 1
    return Fibonacci_native(
        n-1) +
        Fibonacci_native(n-2)
```

Now let's compare the performance of different implementations.

R

```
> library(microbenchmark)
> library(Rcpp)
> source('Fibonacci_native}.R')
> sourceCpp('Fibonacci.cpp')
> # the native implementation in R
> microbenchmark(Fibonacci_native(20), times=1000)
Unit: milliseconds
                 expr      min       lq     mean   median
                      uq      max
 Fibonacci_native(20) 3.917138 4.056468 4.456824 4.190078
    4.462815 26.30846
 neval
  1000
> the C++ implementation
> microbenchmark(Fibonacci(20), times=1000)
Unit: microseconds
```

```
           expr    min      lq     mean median     uq     max
              neval
 Fibonacci(20) 14.251 14.482 15.81708 14.582 14.731 775.095
     1000
```

The native implementation in R is much slower than the C++ implementation (see the difference of units).

Python

```
>>> import pyximport
>>> pyximport.install()
(None, <pyximport.pyximport.PyxImporter object at 0x1056f9f98>)
>>> from Fibonacci import Fibonacci_static, Fibonacci
>>> from Fibonacci_native import Fibonacci_native
>>> import timeit
>>> print(f'{timeit.timeit("Fibonacci_native(20)",setup="from
    __main__ import Fibonacci_native",number=1000):.6f} seconds'
    )
1.927359 seconds
>>> print(f'{timeit.timeit("Fibonacci(20)",setup="from __main__
    import Fibonacci",number=1000):.6f} seconds')
0.316726 seconds
>>> print(f'{timeit.timeit("Fibonacci_static(20)",setup="from
    __main__ import Fibonacci_static",number=1000):.6f} seconds'
    )
0.012741 seconds
```

The results show the static typed implementation is the fastest, and the native implementation in pure Python is the slowest. Again the time measured by `timeit.timeit` is the total time to run the main statement 1000 times. The average time per run of `Fibonacci_static` function is close to the average time per run of `Fibonacci` in R.

2.8 A FIRST IMPRESSION OF FUNCTIONAL PROGRAMMING

Functional (programming) languages define programs as mathematical functions and treat them as first-class[22]. In pure functional programming, a function takes input and returns output without making any side effect. A side effect means a state variable outside the function is modified. For example, printing the value of a variable inside a function is a side effect.

Both R and Python are not purely functional languages per se, although R behaves more functionally than Python [5]. If you wonder why the `return` function

[22]https://en.wikipedia.org/wiki/List_of_programming_languages_by_type#Functional_languages

could be ignored to return a value in an R function, now you get the explanation — a function should return the output automatically from an FP perspective.

I don't think it makes any sense to debate whether we should choose OOP or FP using R/Python. And both languages support multiple programming paradigms (FP, OOP, etc.). If you want to do purist FP, neither R nor Python would be the best choice. For example, purist FP should avoid loop because loop always involves an iteration counter whose value changes over iterations, which is obviously a side effect. Thus, recursion is very important in FP. But tail recursion optimization is not supported in R and Python although it can be implemented via trampoline[23].

Let's introduce a few tools that could help to get a first impression of FP in R and Python.

2.8.1 Anonymous function

We have seen anonymous functions before. Anonymous functions don't have names given to them. They are commonly used in FP. To recap, let's see how to define a useless anonymous function.

```
> function(x) x*x
function(x) x*x
```

```
>>> lambda x: x**2
<function <
    lambda> at 0x1033d5510>
```

Usually, we define anonymous functions because we want to pass them to other functions. And thus, the functions defined above are useless as they are not passed to any other function and after the definition they are deleted automatically from the memory.

Of course, we can assign names to the anonymous functions — but what is the point to naming anonymous functions?

```
> f1 = function(x) x*x
> f1
function(x) x*x
```

```
>>> f1 = lambda x: x**2
>>> f1(2)
4
```

2.8.2 Map

Map could be used to avoid loops. The first argument to map is a function.

In R, we use `Map` which is shipped with base R; and in Python we use `map` which is also a built-in function.

[23]https://en.wikipedia.org/wiki/Tail_call#Through_trampolining

R

```
> Map(function(x) x*x,
    c(1:3))
[[1]]
[1] 1

[[2]]
[1] 4

[[3]]
[1] 9
> Map(function(x, y) x*y,
    c(1:3), c(4:6))
[[1]]
[1] 4

[[2]]
[1] 10

[[3]]
[1] 18
```

```
>>> map(
    lambda e: e**2, [1,2,3])
<map object at 0x1033e3160>
>>> list(map(
    lambda x: x**2, [1,2,3]))
[1, 4, 9]
>>> list(map(lambda x, y: x*
    y, [1,2,3], [4,5,6]))
[4, 10, 18]
```

There are a few things to note from the code snippets above. First, the returned value from `Map` in R is a list rather than a vector. That is because `Map` is just a wrapper of `mapply` function in R, and the `SIMPLIFY` argument is set to `FALSE`. If we want to get a vector value returned, just use `mapply` instead.

R

```
> mapply(function(x, y) x*y, c(1:3), c(4:6), SIMPLIFY=TRUE)
[1]  4 10 18
```

The returned value from `map` in Python is a `map` object which is an iterator. The `generator` function we have seen before is also an iterator. To use an iterator, we could convert it to a list, or get the next value using the `next` function, or just use a for loop.

Python

```
>>> # use next on an iterator
>>> o=map(lambda x, y: x*y, [1,2,3], [4,5,6])
>>> next(o)
4
>>> next(o)
10
```

```
>>> o=map(lambda x, y: x*y, [1,2,3], [4,5,6])
>>> # use for loop on an iterator
>>> for e in o:print(e)
...
4
10
18
>>> # convert an iterator to a list
>>> o=map(lambda x, y: x*y, [1,2,3], [4,5,6])
>>> list(o)
[4, 10, 18]
```

2.8.3 Filter

Similar to map, the first argument to `Filter` in R and `filter` in Python is also a function. This function is used to get the elements which satisfy a certain condition specified in the first argument. For example, we can use it to get the even numbers.

R

```
> Filter(function(x) x
    %%2==0, c(0:10))
[1]  0  2  4  6  8 10
```

Python

```
>>> list(filter(
    lambda e: e%2==0,
    range(11)))
[0, 2, 4, 6, 8, 10]
```

2.8.4 Reduce

`Reduce` behaves a bit differently from `Map` and `Filter`. The first argument is a function with exactly two arguments. The second argument is a sequence, on each element of which the first function argument will be applied from left to right. And there is one optional argument called initializer. When the initializer is provided, it will be used as the first argument of the function argument of `Reduce`. The examples below depict how it works.

R

```
> Reduce(function(x, y) x+2
    *y, c(1,2,3))
[1] 11
> Reduce(function(x, y) x+2
    *y, c(1,2,3), 1)
[1] 13
```

Python

```
>>> from functools import
    reduce
>>> reduce(lambda x, y:x+2*y
    , [1,2,3])
11
>>> reduce(
    lambda x, y:x+2*y,
    range(1,4), 1)
13
```

Please note that in order to use `reduce` in Python, we have to `import` it from the `functools` module.

Utilizing these functions flexibly may help to make the code more concise, at the cost of readability.

2.9 MISCELLANEOUS

We have introduced the basics of R/Python programming so far. There is much more to learn in becoming an advanced user of R/Python. For example, the appropriate usages of `iterator, generator, decorator` could improve both the conciseness and readability of your Python code. The `generator`[24] is commonly seen in machine learning programs to prepare training/testing samples. `decorator` is a kind of syntactic sugar to allow the modification of a function's behavior in a simple way. In R there are no built-in `iterator, generator, decorator`, but you may find some third-party libraries to mimic these features; or you may try to implement your own.

One advantage of Python over R is that there are some built-in modules containing high-performance data structures or commonly used algorithms implemented efficiently. For example, I enjoy using the `deque` structure in the Python `collections` module[25], but there is no built-in counterpart in R. We have written our own binary search algorithm earlier in this chapter, which can also be replaced by the functions in the built-in module `bisect`[26] in Python.

Another important aspect of programming is testing. Unit testing is a typical software testing method that is commonly adopted in practice. In R there are two third-party packages `testthat` and `RUnit`. In Python, the built-in `unittest` is quite powerful. Also, the third-party module `pytest`[27] is very popular.

[24]https://docs.python.org/3/howto/functional.html
[25]https://docs.python.org/3/library/collections.html
[26]https://docs.python.org/3.7/library/bisect.html
[27]https://docs.pytest.org/en/latest/

CHAPTER 3

data.table and pandas

Upon receiving feedback and requests from readers on the first few chapters, I decided to devote a whole chapter to the introduction of `data.table` and `pandas`. Of course, there are many other great data analysis tools in both R and Python. For example, many R users like using `dplyr` to build up data analysis pipelines. The performance of `data.table` is superior and that is the main reason I feel there is no need to use other tools for the same tasks in R. But if you are a big fan of the pipe operator `%>%` you may use `data.table` and `dplyr` together. Regarding the big data ecosystem, Apache Spark has API in both R and Python. Recently, there are also some emerging projects aiming at better usability and performance, such as `Apache Arrow`[1], `Modin`[2].

3.1 SQL

Similar to the previous chapters, I will introduce the tools side by side. However, I feel before diving into the world of `data.table` and `pandas`, it is better to talk a little bit about `SQL`[3]. `SQL` is a Query language designed for managing data in relational database management system (RDBMS). Some of the most popular RDBMSs include `MS SQL Server`, `MySQL`, `PostgreSQL`, etc. Different RDBMSs may use SQL languages with major or subtle differences.

If you have never used RDBMS you may wonder why we need it.

- first, we need a system to store the data;
- second, we also need a system that allows us to easily access, manage and update the data.

In this chapter, we will use the public `mtcars` dataset as an example. This dataset was available in R[4] and originally it was reported in [11].

Let's assume there is a table `mtcars` in a database (I'm using sqlite3 in this book) and see some simple tasks we can do with `SQL` queries.

[1]https://arrow.apache.org
[2]https://github.com/modin-project/modin
[3]https://en.wikipedia.org/wiki/SQL
[4]https://stat.ethz.ch/R-manual/R-devel/library/datasets/html/mtcars.html

Table 3.1: Partial mtcars data loaded from R 3.5.1

name	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.00	6.00	160.00	110.00	3.90	2.62	16.46	0.00	1.00	4.00	4.00
Datsun 710	22.80	4.00	108.00	93.00	3.85	2.32	18.61	1.00	1.00	4.00	1.00
Valiant	18.10	6.00	225.00	105.00	2.76	3.46	20.22	1.00	0.00	3.00	1.00
Duster 360	14.30	8.00	360.00	245.00	3.21	3.57	15.84	0.00	0.00	3.00	4.00
Merc 240D	24.40	4.00	146.70	62.00	3.69	3.19	20.00	1.00	0.00	4.00	2.00
Merc 230	22.80	4.00	140.80	95.00	3.92	3.15	22.90	1.00	0.00	4.00	2.00
Merc 280	19.20	6.00	167.60	123.00	3.92	3.44	18.30	1.00	0.00	4.00	4.00
Merc 280C	17.80	6.00	167.60	123.00	3.92	3.44	18.90	1.00	0.00	4.00	4.00
Merc 450SE	16.40	8.00	275.80	180.00	3.07	4.07	17.40	0.00	0.00	3.00	3.00

SQL

```
sqlite> select * from mtcars limit 2;
name,mpg,cyl,disp,hp,drat,wt,qsec,vs,am,gear,carb
"Mazda RX4",21,6,160,110,3.9,2.62,16.46,0,1,4,4
"Mazda RX4 Wag",21,6,160,110,3.9,2.875,17.02,0,1,4,4
```

In the example above, I select two rows from the table using the syntax `select from`. The keyword `limit` in sqlite specifies the number of rows to return. In other RDBMSs, we may need to use `top` instead. It is straightforward to select on conditions with the `where` keyword.

SQL

```
sqlite> select mpg,cyl from mtcars where name = 'Mazda RX4 Wag';
mpg,cyl
21,6
```

We can definitely use more conditions with the `where` clause.

SQL

```
sqlite> .mode column -- make the output aligned; and yes we use
    '--' to start comment in many SQL languages
sqlite> select name, mpg, cyl,vs,am from mtcars where vs=1 and
    am=1;
name        mpg         cyl         vs          am
----------  ----------  ----------  ----------  ----------
Datsun 710  22.8        4           1           1
Fiat 128    32.4        4           1           1
Honda Civi  30.4        4           1           1
Toyota Cor  33.9        4           1           1
Fiat X1-9   27.3        4           1           1
```

```
Lotus Euro   30.4         4            1            1
Volvo 142E   21.4         4            1            1
```

We are just accessing specific rows and columns from the table in database with `select from where`. We can also do something a bit more fancy, for example, to get the maximum, the minimum and the average of `mpg` for all vehicles grouped by the number of cylinders.

SQL

```
sqlite> select cyl, max(mpg) as max, min(mpg) as min, avg(mpg)
   as avg from mtcars group by cyl order by cyl;
cyl          max          min          avg
----------   ----------   ----------   ----------------
4            33.9         21.4         26.6636363636364
6            21.4         17.8         19.7428571428571
8            19.2         10.4         15.1
```

In the above example, there are a few things worth noting. We use `as` to create an alias for a variable; we group the original rows by the number of cylinders with the keyword `group by`; and we sort the output rows with the keyword `order by`. `max`, `min` and `avg` are all built-in functions that we can use directly.

It is also possible to have user-defined functions in `SQL` as what we usually do in other programming languages.

SQL queries could be very complex and long. Usually a long query might contain subqueries or Common Table Expressions (CTE). First, let's see an example of subquery to get the average mpg for cars with 6 cylinders as below.

SQL

```
sqlite> select am, avg(mpg) avg_mpg from (select * from mtcars
   where cyl==6) group by am;
am           avg_mpg
----------   ----------
0            19.125
1            20.5666666
```

The subquery in the parentheses specifies the fact that we are only interested in the rows with cyl==6. The same results could be obtained by CTE as below.

SQL

```
sqlite> with temp_table as (select * from mtcars where cyl==6)
   select am, avg(mpg) avg_mpg from temp_table group by am;
am           avg_mpg
```

```
─────────────   ─────────────
0               19.125
1               20.5666666
```

Basically, a CTE can be thought of as a temporary result set which can be referenced multiple times in one query. Multiple CTEs can be used in the same query. Using CTE improves the readability of the query with similar performance to subquery in complex queries. In some cases, CTE could be a bottleneck in performance and you may consider using temporary table which will not be discussed in this chapter.

3.2 GET STARTED WITH DATA.TABLE AND PANDAS

`SQL` is a very powerful tool for data analysis, but it works on RDBMS and generally we can't apply R or Python functions to the database tables directly. Many practitioners in data science have to work with database at times but more often they need to work in a programming languages such as R or Python. We have introduced the data.frame in both R and Python in previous chapters. A data.frame is just like a database table that you may operate within the corresponding language. Usually a data.frame is stored in memory, but of course it can also be deserialized for storage in hard disks.

With the data.frame-like objects, we could build up a better data processing pipeline by only reading the original data from the database and storing the final output to the database if necessary. Most of works we can do with data.frame-like objects may also be done in RDBMS with `SQL`. But they may require intense interactions with a database, which is not preferred if it can be avoided.

Now, let's get started with `data.table` and `pandas`. In this book, I will use `data.table 1.12.0` and `pandas 0.24.0`.

R

```
> library(data.table)
data.table 1.12.0  Latest
    news: r—datatable.com
}
```

Python

```
>>> import pandas as pd
>>> pd.__version__
'0.24.2'
```

I have saved the `mtcars` data to a csv[5] file (code/chapter3/mtcars.csv), which is loaded from R 3.5.1. Although the mtcars data is loaded into R environment by default, let's load the data by reading the raw csv file for learning purposes.

R

```
> mtcars_dt=fread('mtcars.csv')
> head(mtcars_dt)
```

[5]https://en.wikipedia.org/wiki/Comma-separated_values

```
                    name  mpg cyl disp  hp drat    wt  qsec vs am
                        gear carb
1:          Mazda RX4 21.0   6  160 110 3.90 2.620 16.46  0  1
      4    4
2:      Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1
      4    4
3:         Datsun 710 22.8   4  108  93 3.85 2.320 18.61  1  1
      4    1
4:     Hornet 4 Drive 21.4   6  258 110 3.08 3.215 19.44  1  0
      3    1
5: Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0
      3    2
6:            Valiant 18.1   6  225 105 2.76 3.460 20.22  1  0
      3    1
```

Python

```
>>> import pandas as pd
>>> mtcars_df=pd.read_csv('mtcars.csv')
>>> mtcars_df.head(5)
                    name   mpg  cyl   disp   hp  ...   qsec  vs  am
                         gear  carb
0          Mazda RX4  21.0    6  160.0  110  ...  16.46   0   1
       4     4
1      Mazda RX4 Wag  21.0    6  160.0  110  ...  17.02   0   1
       4     4
2         Datsun 710  22.8    4  108.0   93  ...  18.61   1   1
       4     1
3     Hornet 4 Drive  21.4    6  258.0  110  ...  19.44   1   0
       3     1
4  Hornet Sportabout  18.7    8  360.0  175  ...  17.02   0   0
       3     2

[5 rows x 12 columns]
```

The type of `mtcars_dt` is `data.table`, not `data.frame`. Here we use the `fread` function from `data.table` to read a file and the output type is a `data.table` directly. Regarding reading csv in R, a very good package is `readr` for very large files, but the output has a `data.frame` type. In practice, it is very common to convert a `data.frame` to `data.table` with the function `as.data.table`.

3.3 INDEXING & SELECTING DATA

Before introducing the indexing rules in `data.table` and `pandas`, it's better to understand the `key` in `data.table` and the `index` in `pandas`.

What is the `key` in a `data.table`? We have talked about RDBMS and `SQL` in the previous section. With `select from where` we can easily access specific rows satisfying certain conditions. When the database table is too large, a database `index` is used to improve the performance of data retrieval operations. Essentially, a database `index` is a data structure, and to maintain the data structure, additional cost (for example, space) may be required. The reason to use `key` in `data.table` and `index` in `pandas` is very similar. Now let's see how to set `key` and `index`.

R

```
> setkey(mtcars_dt,name)
> key(mtcars_dt)
[1] "name"
> head(mtcars_dt,5)
                 name  mpg cyl disp  hp drat    wt  qsec vs am
       gear carb
1:        AMC Javelin 15.2   8  304 150 3.15 3.435 17.30  0  0
      3    2
2: Cadillac Fleetwood 10.4   8  472 205 2.93 5.250 17.98  0  0
      3    4
3:         Camaro Z28 13.3   8  350 245 3.73 3.840 15.41  0  0
      3    4
4:  Chrysler Imperial 14.7   8  440 230 3.23 5.345 17.42  0  0
      3    4
5:         Datsun 710 22.8   4  108  93 3.85 2.320 18.61  1  1
      4    1
```

Python

```
>>> mtcars_df.set_index('name',inplace=True)
>>> mtcars_df.index.name
'name'
>>> mtcars_df.head(5)
                   mpg  cyl   disp   hp  drat  ...   qsec  vs
        am  gear  carb
name                                           ...
Mazda RX4         21.0    6  160.0  110  3.90  ...  16.46   0
     1     4     4
Mazda RX4 Wag     21.0    6  160.0  110  3.90  ...  17.02   0
     1     4     4
Datsun 710        22.8    4  108.0   93  3.85  ...  18.61   1
     1     4     1
```

```
Hornet 4 Drive      21.4    6  258.0  110  3.08  ...  19.44   1
      0     3     1
Hornet Sportabout  18.7    8  360.0  175  3.15  ...  17.02   0
      0     3     2
```

There are quite a few things worth noting from the above code snippets. When we use the `setkey` function the quotes for the column name is optional. So `setkey(mtcars_dt, name)` is equivalent to `setkey(mtcars_dt, 'name')`. But in `pandas`, quotes are required. The effect of `setkey` is in place, which means no copies of the data are made at all. But in `pandas`, by default `set_index` set the index on a copy of the data and the modified copy is returned. Thus, in order to make the effect in place, we have set the argument `inplace=True` explicitly. Another difference is that `setkey` would sort the original data in place automatically but `set_index` does not. It's also worth noting every `pandas` data.frame has an index; and by default it is `numpy.arange(n)` where `n` is the number of rows. But there is no default key in a `data.table`.

In the above example, we only use a single column as the key/index. It is possible to use multiple columns as well.

R

```
> setkeyv(mtcars_dt,c('cyl',
    'gear'))
> key(mtcars_dt)
[1] "cyl"  "gear"
```

Python

```
>>> mtcars_df.set_index(['
    cyl','gear'],inplace=
    True)
>>> mtcars_df.index.names
FrozenList(['cyl', 'gear'])
```

To use multiple columns as the key in `data.table`, we use the function `setkeyv`. It is also interesting that we have to use `index.names` rather than `index.name` to get the multiple column names of the index (which is called MultiIndex) in `pandas`. There are duplicated combinations of (cyl, gear) in the data, which implies key or index could be duplicated.

Once the key/index set, we can access rows with given indices fast.

R

```
> mtcars_dt['Merc 230']
       name  mpg cyl  disp hp drat   wt qsec vs am gear carb
1: Merc 230 22.8   4 140.8 95 3.92 3.15 22.9  1  0    4    2
> mtcars_dt[c('Merc 230','Camaro Z28')] # multiple values of a
    single index
         name  mpg cyl  disp  hp drat   wt  qsec vs am gear carb
1:   Merc 230 22.8   4 140.8  95 3.92 3.15 22.90  1  0    4    2
2: Camaro Z28 13.3   8 350.0 245 3.73 3.84 15.41  0  0    3    4
> setkeyv(mtcars_dt,c('cyl','gear'))
> mtcars_dt[.(6,4)] # work with key vector using .()
```

```
           name  mpg cyl  disp  hp drat    wt  qsec vs am gear
               carb
1:     Mazda RX4 21.0   6 160.0 110 3.90 2.620 16.46  0  1    4
      4
2: Mazda RX4 Wag 21.0   6 160.0 110 3.90 2.875 17.02  0  1    4
      4
3:      Merc 280 19.2   6 167.6 123 3.92 3.440 18.30  1  0    4
      4
4:     Merc 280C 17.8   6 167.6 123 3.92 3.440 18.90  1  0    4
      4
> mtcars_dt[.(c(6,8),c(4,3))] # key vector with multiple values
                   name  mpg cyl  disp  hp drat    wt  qsec vs
                     am gear carb
 1:           Mazda RX4 21.0   6 160.0 110 3.90 2.620 16.46  0
     1    4    4
 2:       Mazda RX4 Wag 21.0   6 160.0 110 3.90 2.875 17.02  0
     1    4    4
 3:            Merc 280 19.2   6 167.6 123 3.92 3.440 18.30  1
     0    4    4
 4:           Merc 280C 17.8   6 167.6 123 3.92 3.440 18.90  1
     0    4    4
 5:         AMC Javelin 15.2   8 304.0 150 3.15 3.435 17.30  0
     0    3    2
 6:  Cadillac Fleetwood 10.4   8 472.0 205 2.93 5.250 17.98  0
     0    3    4
 7:          Camaro Z28 13.3   8 350.0 245 3.73 3.840 15.41  0
     0    3    4
 8:   Chrysler Imperial 14.7   8 440.0 230 3.23 5.345 17.42  0
     0    3    4
 9:    Dodge Challenger 15.5   8 318.0 150 2.76 3.520 16.87  0
     0    3    2
10:          Duster 360 14.3   8 360.0 245 3.21 3.570 15.84  0
    0    3    4
11:   Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0
    0    3    2
12: Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0
    0    3    4
13:          Merc 450SE 16.4   8 275.8 180 3.07 4.070 17.40  0
    0    3    3
14:          Merc 450SL 17.3   8 275.8 180 3.07 3.730 17.60  0
    0    3    3
15:         Merc 450SLC 15.2   8 275.8 180 3.07 3.780 18.00  0
    0    3    3
16:    Pontiac Firebird 19.2   8 400.0 175 3.08 3.845 17.05  0
    0    3    2
```

Here is a bit of explanation for the code above. We can simply use `[]` to access the rows with the specified key values if the key has a character type. But if the key has a numeric type, `list()` is required to enclose the key values. In `data.table`, `.()` is just an alias of `list()`, which means we would get the same results with `mtcars_dt[list (6,4)]`. Of course, we can use also do `mtcars_dt[.('Merc 230')]` which is equivalent to `mtcars_dt[.('Merc 230')]`.

Python

```
>>> mtcars_df.loc['Merc 230']
mpg      22.80
cyl       4.00
disp    140.80
hp       95.00
drat      3.92
wt        3.15
qsec     22.90
vs        1.00
am        0.00
gear      4.00
carb      2.00
Name: Merc 230, dtype: float64
>>> mtcars_df.loc[['Merc 230','Camaro Z28']] # multiple values
    of a single index
             mpg  cyl   disp   hp  drat    wt   qsec  vs  am
                gear  carb
name
Merc 230    22.8    4  140.8   95  3.92  3.15  22.90   1   0
      4     2
Camaro Z28  13.3    8  350.0  245  3.73  3.84  15.41   0   0
      3     4

mtcars_df.set_index(['cyl','gear'],inplace=True)
>>> mtcars_df.loc[(6,4)] # work with MultiIndex using ()
           mpg   disp   hp  drat     wt   qsec  vs  am  carb
cyl gear
6   4     21.0  160.0  110  3.90  2.620  16.46   0   1     4
    4     21.0  160.0  110  3.90  2.875  17.02   0   1     4
    4     19.2  167.6  123  3.92  3.440  18.30   1   0     4
    4     17.8  167.6  123  3.92  3.440  18.90   1   0     4
>>> # you may notice that the name column disappeared; that
    would be explained later
```

```
>>> mtcars_df.loc[[(6,4),(8,3)]] # MultiIndex with multiple
    values
              mpg   disp   hp  drat     wt   qsec  vs  am  carb
cyl gear
6   4       21.0  160.0  110  3.90  2.620  16.46   0   1     4
    4       21.0  160.0  110  3.90  2.875  17.02   0   1     4
    4       19.2  167.6  123  3.92  3.440  18.30   1   0     4
    4       17.8  167.6  123  3.92  3.440  18.90   1   0     4
8   3       18.7  360.0  175  3.15  3.440  17.02   0   0     2
    3       14.3  360.0  245  3.21  3.570  15.84   0   0     4
    3       16.4  275.8  180  3.07  4.070  17.40   0   0     3
    3       17.3  275.8  180  3.07  3.730  17.60   0   0     3
    3       15.2  275.8  180  3.07  3.780  18.00   0   0     3
    3       10.4  472.0  205  2.93  5.250  17.98   0   0     4
    3       10.4  460.0  215  3.00  5.424  17.82   0   0     4
    3       14.7  440.0  230  3.23  5.345  17.42   0   0     4
    3       15.5  318.0  150  2.76  3.520  16.87   0   0     2
    3       15.2  304.0  150  3.15  3.435  17.30   0   0     2
    3       13.3  350.0  245  3.73  3.840  15.41   0   0     4
    3       19.2  400.0  175  3.08  3.845  17.05   0   0     2
```

Compared to `data.table`, we need to use the `loc` method when accessing rows based on index. The `loc` method also takes boolean conditions.

Python

```
>>> mtcars_df.loc[mtcars_df.mpg>30] # select the vehicles with
    mpg>30
              mpg  disp   hp  drat     wt   qsec  vs  am  carb
cyl gear
4   4       32.4  78.7   66  4.08  2.200  19.47   1   1     1
    4       30.4  75.7   52  4.93  1.615  18.52   1   1     2
    4       33.9  71.1   65  4.22  1.835  19.90   1   1     1
    5       30.4  95.1  113  3.77  1.513  16.90   1   1     2
```

When using boolean conditions, `loc` could be ignored for convenience.

Python

```
>>> mtcars_df[mtcars_df.mpg>30] # ignore loc with boolean
    conditions
              mpg  disp   hp  drat     wt   qsec  vs  am  carb
cyl gear
4   4       32.4  78.7   66  4.08  2.200  19.47   1   1     1
    4       30.4  75.7   52  4.93  1.615  18.52   1   1     2
    4      33.9  71.1   65  4.22  1.835  19.90   1   1     1
    5      30.4  95.1  113  3.77  1.513  16.90   1   1     2
```

If the key/index is not needed, we can remove the key or reset the index. For `data.table` we can set a new key to override the existing one which then becomes a column. But in `pandas`, `set_index` method removes the existing index which also disappears from the data.frame.

R

```
> key(mtcars_dt)
[1] "cyl"  "gear"
> setkey(mtcars_dt, NULL) # remove the existing key
> key(mtcars_dt)
NULL
> setkey(mtcars_dt, 'gear')
> key(mtcars_dt)
[1] "gear"
> setkey(mtcars_dt, 'name') # override the existing key
> key(mtcars_dt)
[1] "name"
```

Python

```
>>> mtcars_df.index.names
FrozenList(['cyl', 'gear'])
>>> mtcars_df.reset_index(inplace=True) # remove the existing
    index
>>> mtcars_df.index.names
FrozenList([None])
>>> mtcars_df.set_index(['gear'], inplace=True)
>>> mtcars_df.index.name
'gear'
>>> mtcars_df.columns # list all columns
Index(['cyl', 'name', 'mpg', 'disp', 'hp', 'drat', 'wt', 'qsec',
     'vs', 'am',
       'carb'],
      dtype='object')
>>> mtcars_df.set_index('name', inplace=True)
>>> mtcars_df.columns # the name column disappears
Index(['cyl', 'mpg', 'disp', 'hp', 'drat', 'wt', 'qsec', 'vs', '
    am', 'carb'], dtype='object')
>>>
```

In Chapter 1, we introduced integer-based indexing for list/vector. It is also applicable to `data frame` and `data.table`.

R

```
> mtcars_dt=fread('mtcars.csv')
> mtcars_dt[c(1,2,6),]
            name  mpg cyl disp  hp drat    wt  qsec vs am gear carb
1:     Mazda RX4 21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
2: Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
3:       Valiant 18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

Python

```
>>> mtcars_df=pd.read_csv('mtcars.csv')
>>> mtcars_df.iloc[[0,1,5]] # again, indices in python are zero-based.
            name   mpg  cyl   disp   hp  ...   qsec  vs  am  gear  carb
0      Mazda RX4  21.0    6  160.0  110  ...  16.46   0   1     4     4
1  Mazda RX4 Wag  21.0    6  160.0  110  ...  17.02   0   1     4     4
5        Valiant  18.1    6  225.0  105  ...  20.22   1   0     3     1
```

So far we have seen how to access specific rows. What about columns? Accessing columns in `data.table` and `pandas` is quite straightforward. For `data.table`, we can use `$` sign to access a single column or a vector to specify multiple columns inside `[]`. For data.frame in `pandas`, we can use `.` to access a single column or a list to specify multiple columns inside `[]`.

R

```
> head(mtcars_dt$mpg,5) # access a single column
[1] 21.0 21.0 22.8 21.4 18.7}
> mtcars_dt[1:5,c('mpg','gear')]
    mpg gear
```

Python

```
>>> mtcars_df.iloc[0:5].mpg.values
array([21. , 21. , 22.8, 21.4, 18.7])
Name: mpg, dtype: float64
>>> mtcars_df[['mpg','gear']].head(5)
```

```
1: 21.0     4
2: 21.0     4
3: 22.8     4
4: 21.4     3
5: 18.7     3
```

```
    mpg  gear
0  21.0     4
1  21.0     4
2  22.8     4
3  21.4     3
4  18.7     3
```

In addition to passing a vector to access multiple columns in `data.table`, we can also use the `.(variable_1,variable_2,...)`.

R

```
> mtcars_dt[1:5,.(mpg,cyl,hp)] # without quotes for variables
    mpg cyl hp
1: 21.5   4 97
2: 22.8   4 93
3: 24.4   4 62
4: 22.8   4 95
5: 32.4   4 66
```

It is also possible to do integer-based columns slicing.

R

```
> mtcars_dt[1:5,c(2,5,6)] # the 2nd, the 5th and the 6th column
    mpg  hp drat
1: 21.0 110 3.90
2: 21.0 110 3.90
3: 22.8  93 3.85
4: 21.4 110 3.08
5: 18.7 175 3.15
```

Python

```
>>> mtcars_df.iloc[0:5,[1,4,5]] # the 2nd, the 5th and the 6th column
    mpg   hp  drat
0  21.0  110  3.90
1  21.0  110  3.90
2  22.8   93  3.85
3  21.4  110  3.08
4  18.7  175  3.15
```

To access specific rows and specific columns, there are two strategies:

1. select rows and then select columns in a chain;
2. select rows and columns simultaneously.

Let's see some examples.

R

```
> mtcars_dt=fread('mtcars.csv')
> setkey(mtcars_dt,'gear')
> mtcars_dt[.(6),c('mpg','cyl','hp')] # use strategy 2;
    mpg cyl  hp
1: 21.4   6 110
2: 18.1   6 105
3: 21.0   6 110
4: 21.0   6 110
5: 19.2   6 123
6: 17.8   6 123
7: 19.7   6 175
> mtcars_dt[.(6)][,c('mpg','cyl','hp')] # use strategy 1;
    mpg cyl  hp
1: 21.4   6 110
2: 18.1   6 105
3: 21.0   6 110
4: 21.0   6 110
5: 19.2   6 123
6: 17.8   6 123
7: 19.7   6 175
```

Python

```
>>> mtcars_df=pd.read_csv('mtcars.csv')
>>> mtcars_df.set_index('cyl', inplace=True)
>>> mtcars_df.loc[6,['mpg','hp']] # use strategy 2; we can't
    list the index as a normal column; while a key in data.table
     is still a normal column
      mpg   hp
cyl
6    21.0  110
6    21.0  110
6    21.4  110
6    18.1  105
6    19.2  123
6    17.8  123
6    19.7  175
>>> mtcars_df.loc[6][['mpg','hp']] # us strategy 1
      mpg   hp
cyl
6    21.0  110
6    21.0  110
6    21.4  110
6    18.1  105
6    19.2  123
6    17.8  123
```

```
6     19.7  175
```

As we have seen, using the `setkey` function for `data.table` sorts the `data.table` automatically. However, sorting is not always desired. In `data.table`, there is another function `setindex/setindexv` which has similar effects to `setkey/setkeyv` but doesn't sort the `data.table`. In addition, one `data.table` could have multiple indices, but it cannot have multiple keys.

R

```
> setindex(mtcars_dt,'cyl')
> head(mtcars_dt,5) # not sorted by the index
                 name  mpg cyl disp  hp drat    wt  qsec vs am
                    gear carb
1:          Mazda RX4 21.0   6  160 110 3.90 2.620 16.46  0  1
      4    4
2:      Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1
      4    4
3:         Datsun 710 22.8   4  108  93 3.85 2.320 18.61  1  1
      4    1
4:     Hornet 4 Drive 21.4   6  258 110 3.08 3.215 19.44  1  0
      3    1
5: Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0
      3    2
> mtcars_dt[.(6), on='cyl'] # we use on to specify the index
              name  mpg cyl  disp  hp drat    wt  qsec vs am gear
                  carb
1:       Mazda RX4 21.0   6 160.0 110 3.90 2.620 16.46  0  1    4
       4
2:   Mazda RX4 Wag 21.0   6 160.0 110 3.90 2.875 17.02  0  1    4
       4
3: Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3
       1
4:         Valiant 18.1   6 225.0 105 2.76 3.460 20.22  1  0    3
       1
5:        Merc 280 19.2   6 167.6 123 3.92 3.440 18.30  1  0    4
       4
6:       Merc 280C 17.8   6 167.6 123 3.92 3.440 18.90  1  0    4
       4
7:    Ferrari Dino 19.7   6 145.0 175 3.62 2.770 15.50  0  1    5
       6
> setindexv(mtcars_dt,c('cyl','gear'))
> mtcars_dt[.(6,3),on=c('cyl','gear')]
              name  mpg cyl disp  hp drat    wt  qsec vs am gear
                 carb
```

```
1: Hornet 4 Drive 21.4   6  258 110 3.08 3.215 19.44  1  0    3
    1
2:        Valiant 18.1   6  225 105 2.76 3.460 20.22  1  0    3
    1

> mtcars_dt[.(4),on='cyl'] # the index 'cyl' still works after
   set c('cyl','gear') as indexv
             name  mpg cyl  disp  hp drat    wt  qsec vs am
               gear carb
 1:      Datsun 710 22.8   4 108.0  93 3.85 2.320 18.61  1  1
     4    1
 2:       Merc 240D 24.4   4 146.7  62 3.69 3.190 20.00  1  0
     4    2
 3:        Merc 230 22.8   4 140.8  95 3.92 3.150 22.90  1  0
     4    2
 4:        Fiat 128 32.4   4  78.7  66 4.08 2.200 19.47  1  1
     4    1
 5:     Honda Civic 30.4   4  75.7  52 4.93 1.615 18.52  1  1
     4    2
 6: Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1
     4    1
 7:  Toyota Corona 21.5   4 120.1  97 3.70 2.465 20.01  1  0
     3    1
 8:       Fiat X1-9 27.3   4  79.0  66 4.08 1.935 18.90  1  1
     4    1
 9:  Porsche 914-2 26.0   4 120.3  91 4.43 2.140 16.70  0  1
     5    2
10:   Lotus Europa 30.4   4  95.1 113 3.77 1.513 16.90  1  1
    5    2
11:      Volvo 142E 21.4   4 121.0 109 4.11 2.780 18.60  1  1
    4    2
```

3.4 ADD/REMOVE/UPDATE

First, let's see how to delete a single column.

R

```
> mtcars_dt=fread('mtcars.
   csv')
> 'cyl' %in%
   colnames(mtcars_dt)
[1] TRUE
> mtcars_dt$cyl=NULL #
   method 1
```

Python

```
>>> mtcars_df=pd.read_csv('
   mtcars.csv')
>>> 'cyl'
   in mtcars_df.columns
True
>>> mtcars_df.drop(columns=[
   'cyl'],inplace=True)
```

```
> 'cyl' %in%
  colnames(mtcars_dt)
[1] FALSE
> mtcars_dt=fread('mtcars.
  csv')
> mtcars_dt[,cyl:=NULL] #
  method 2
> 'cyl' %in%
  colnames(mtcars_dt)
[1] FALSE
```

```
>>> 'cyl'
  in mtcars_df.columns
False
```

The := operator in `data.table` can be used to add/remove/update columns, by reference. Thus, when we use := no copies of the data are created. Getting familiar with this operator is critical to mastering `data.table`. Next, let's see how to delete multiple columns at the same time.

R

```
> mtcars_dt=fread('mtcars.
  csv')
> mtcars_dt[,c('cyl','hp'):=
  NULL]
> c('cyl','hp') %in%
  colnames(mtcars_dt)
[1] FALSE FALSE
```

Python

```
>>> mtcars_df=pd.read_csv('
  mtcars.csv')
>>> mtcars_df.drop(columns=[
  'cyl','hp'],inplace=True)
>>> [e in mtcars_df.columns
  for e in ['cyl','hp']]
[False, False]
```

The interesting fact of the code above is that in R the `%in%` function is vectorized, but the `in` function in Python is not.

Adding a single column to an existing `data.table` or `DataFrame` is as straightforward as removing.

R

```
> mtcars_dt$new_col=1 # method 1
> head(mtcars_dt, 2)
                name  mpg disp drat    wt  qsec vs am gear carb
                  new_col
1:         Mazda RX4 21.0  160 3.90 2.620 16.46  0  1    4    4
         1
2:     Mazda RX4 Wag 21.0  160 3.90 2.875 17.02  0  1    4    4
         1
> mtcars_dt$new_col=NULL
> mtcars_dt[, new_col:=1] # method 2
> head(mtcars_dt, 2)
            name mpg disp drat    wt  qsec vs am gear carb
              new_col
1:     Mazda RX4  21  160  3.9 2.620 16.46  0  1    4    4
         1
```

```
2: Mazda RX4 Wag   21  160   3.9 2.875 17.02  0  1    4    4
           1
```

Python

```
>>> mtcars_df['new_col']=1
>>> mtcars_df.head(2)
             name   mpg   disp  drat     wt   qsec  vs  am  gear
                  carb  new_col
0      Mazda RX4  21.0  160.0   3.9  2.620  16.46   0   1     4
       4        1
1  Mazda RX4 Wag  21.0  160.0   3.9  2.875  17.02   0   1     4
       4        1
```

Adding multiple columns is a bit tricky compared with adding a single column.

R

```
> mtcars_dt=fread('mtcars.csv')
> mtcars_dt[, `:=`(nc1=1,nc2=2)]
> head(mtcars_dt, 2)
             name mpg cyl disp  hp drat    wt  qsec vs am gear
                 carb nc1 nc2
1:      Mazda RX4  21   6  160 110  3.9 2.620 16.46  0  1    4
      4   1   2
2: Mazda RX4 Wag   21   6  160 110  3.9 2.875 17.02  0  1    4
      4   1   2
```

Python

```
>>> mtcars_df=mtcars_df.assign(**{'nc1':1, 'nc2':2})
>>> mtcars_df.head(2)
             name   mpg  cyl   disp   hp  ...  am  gear  carb
                 nc1  nc2
0      Mazda RX4  21.0    6  160.0  110  ...   1     4     4
   1    2
1  Mazda RX4 Wag  21.0    6  160.0  110  ...   1     4     4
   1    2

[2 rows x 14 columns]
```

In the R code, we use `:=` to create multiple columns. In the Python code, we put the new columns into a dictionary and use the assign function with the dictionary unpacking operator ∗∗. To learn the dictionary unpacking operator, please refer to official document[6]. The `assign` method of a `DataFrame` doesn't have `inplace` argument so we need to assign the modified `DataFrame` to the original one explicitly.

Now let's see how to update values. We can update the entire column or just the column on specific rows.

R

```
> mtcars_dt=fread('mtcars.csv')
> mtcars_dt[, ‘:=‘(nc1=1, nc2=2)]
> mtcars_dt[, nc1:=10] # update the entire column c1
> head(mtcars_dt,2)
            name mpg cyl disp  hp drat    wt  qsec vs am gear
                carb nc1 nc2
1:     Mazda RX4  21   6  160 110  3.9 2.620 16.46  0  1    4
      4  10   2
2: Mazda RX4 Wag  21   6  160 110  3.9 2.875 17.02  0  1    4
      4  10   2
> mtcars_dt[cyl==6, nc1:=3] # update the nc1 for rows with cyl=6
> head(mtcars_dt,2)
            name mpg cyl disp  hp drat    wt  qsec vs am gear
                carb nc1 nc2
1:     Mazda RX4  21   6  160 110  3.9 2.620 16.46  0  1    4
      4   3   2
2: Mazda RX4 Wag  21   6  160 110  3.9 2.875 17.02  0  1    4
      4   3   2
```

Python

```
>>> mtcars_df=pd.read_csv('mtcars.csv')
>>> mtcars_df=mtcars_df.assign(**{'nc1':1,'nc2':2})
>>> mtcars_df['nc1']=10
>>> mtcars_df.head(2)
            name   mpg  cyl   disp   hp  ...  am  gear  carb
                nc1  nc2
0      Mazda RX4  21.0    6  160.0  110  ...   1     4     4
   10    2
1  Mazda RX4 Wag  21.0    6  160.0  110  ...   1     4     4
   10    2

[2 rows x 14 columns]
>>> mtcars_df.loc[mtcars_df.cyl==6,'nc1']=3
>>> mtcars_df.head(2)
```

[6]https://www.python.org/dev/peps/pep-0448

```
              name   mpg  cyl   disp   hp  ...  am  gear  carb
                   nc1  nc2
0      Mazda RX4  21.0    6  160.0  110  ...   1     4     4
    3    2
1  Mazda RX4 Wag  21.0    6  160.0  110  ...   1     4     4
    3    2

[2 rows x 14 columns]
```

We can also combine the technique of rows indexing with column update.

R

```
> mtcars_dt=fread('mtcars.csv')
> setkey(mtcars_dt,'cyl')
> mtcars_dt[,`:=`(nc1=1,nc2=2)]
> mtcars_dt[.(4),nc2:=4] # change nc2 for rows with cyl=4
> head(mtcars_dt,2)
           name  mpg cyl  disp hp drat    wt  qsec vs am gear carb
               nc1 nc2
1: Datsun 710 22.8   4 108.0 93 3.85 2.32 18.61  1  1    4    1
      1   4
2:  Merc 240D 24.4   4 146.7 62 3.69 3.19 20.00  1  0    4    2
      1   4
```

Python

```
>>> mtcars_df=pd.read_csv('mtcars.csv')
>>> mtcars_df.set_index('cyl', inplace=True)
>>> mtcars_df=mtcars_df.assign(**{'nc1':1,'nc2':2})
>>> mtcars_df.loc[4, 'nc2']=4 # change nc2 for rows with cyl=4
>>> mtcars_df.loc[4].head(2)
             name   mpg   disp  hp  drat  ...  am  gear  carb  nc1
                  nc2
cyl                                       ...
4    Datsun 710  22.8  108.0  93  3.85  ...   1     4     1    1
       4
4     Merc 240D  24.4  146.7  62  3.69  ...   0     4     2    1
       4
```

In addition to `:=`, we can also use `set` function to modify values in a `data.table`. When used properly, the performance gain could be significant. Let's use a fictional case.

R

```
> arr=array(0, c(1000,1000)) # create a big array
> dt=data.table(arr) # create a data.table based on the array
> system.time(for (i in 1:nrow(arr)) arr[i,1L] = i) # modify the
    array
   user  system elapsed
  0.003   0.000   0.003
> system.time(for (i in 1:nrow(arr)) dt[i,V1:=i]) # use := for
    data.table
   user  system elapsed
  1.499   0.017   0.383
> system.time(for (i in 1:nrow(arr)) set(dt,i,2L,i)) # use set
    for data.table
   user  system elapsed
  0.003   0.000   0.003
```

We see that updating the values with the `set` function in this example is as fast as updating the values in an array.

3.5 GROUP BY

At the beginning of this chapter, we saw an example with `group by` in SQL query. `group by` is a very powerful operation, and it is also available in `data.table` and `pandas`. Let's try to get the average mpg grouped by cyl.

R

```
> mtcars_dt[, mean(mpg),
    by=cyl]
   cyl       V1
1:   6 19.74286
2:   4 26.66364
3:   8 15.10000
> # it works well but it's
    better to give it a name
> mtcars_dt[, .(mean_mpg=
    mean(mpg)), by=cyl] #
    with the .() or list()
   cyl mean_mpg
1:   6 19.74286
2:   4 26.66364
3:   8 15.10000
```

Python

```
>>> mtcars_df.groupby('cyl
    ').mpg.mean()
cyl
4    26.663636
6    19.742857
8    15.100000
Name: mpg, dtype: float64
>>> mtcars_df.groupby('cyl
    ').mpg.mean().reset_index
    ().rename(columns={'mpg':
    'mean_mpg'})
   cyl   mean_mpg
0    4  26.663636
1    6  19.742857
2    8  15.100000
```

`Group by` also works on multiple columns.

R

```
> mtcars_dt[,.(mean_mpg=
  mean(mpg)),
  by=.(cyl, gear)] # by=c('
  cyl', 'gear') also works
   cyl gear mean_mpg
1:   6    4   19.750
2:   4    4   26.925
3:   6    3   19.750
4:   8    3   15.050
5:   4    3   21.500
6:   4    5   28.200
7:   8    5   15.400
8:   6    5   19.700
```

Python

```
>>> mtcars_df.groupby(['cyl
  ', 'gear']).mpg.mean().
  reset_index().rename(
  columns={'mpg':'mean_mpg
  '})
   cyl  gear  mean_mpg
0    4     3    21.500
1    4     4    26.925
2    4     5    28.200
3    6     3    19.750
4    6     4    19.750
5    6     5    19.700
6    8     3    15.050
7    8     5    15.400
```

We can also create multiple columns with `group by`. My feeling is `data.table` is more expressive.

R

```
> mtcars_dt[,.(mean_mpg=mean(mpg), max_hp=max(hp)), by=.(cyl,
  gear)]
   cyl gear mean_mpg max_hp
1:   6    4   19.750    123
2:   4    4   26.925    109
3:   6    3   19.750    110
4:   8    3   15.050    245
5:   4    3   21.500     97
6:   4    5   28.200    113
7:   8    5   15.400    335
8:   6    5   19.700    175
```

Python

```
>>> mtcars_df.groupby(['cyl','gear']).apply(lambda e:pd.Series({
  'mean_mpg':e.mpg.mean(), 'max_hp':e.hp.max()}))
          mean_mpg  max_hp
cyl gear
4   3       21.500    97.0
    4       26.925   109.0
    5       28.200   113.0
6   3       19.750   110.0
    4       19.750   123.0
    5        19.700   175.0
8   3        15.050   245.0
    5        15.400   335.0
```

In data.table, there is also a keyword called **keyby** which enables **group by** and **sort** operations together.

R

```
> mtcars_dt[,.(mean_mpg=mean(mpg),max_hp=max(hp)), keyby=.(cyl,
   gear)]
   cyl gear mean_mpg max_hp
1:   4    3   21.500     97
2:   4    4   26.925    109
3:   4    5   28.200    113
4:   6    3   19.750    110
5:   6    4   19.750    123
6:   6    5   19.700    175
7:   8    3   15.050    245
8:   8    5   15.400    335
```

It is even possible to add expressions after the by keyword.

R

```
> mtcars_dt[,.(mean_mpg=mean(mpg),max_hp=max(hp)), by=.(cyl==8,
   gear==4)]
     cyl  gear mean_mpg max_hp
1: FALSE  TRUE 24.53333    123
2: FALSE FALSE 22.85000    175
3:  TRUE FALSE 15.10000    335
```

3.6 JOIN

Join[7] combines columns from one or more tables for RDBMSs. We also have the Join operation available in data.table and **pandas**. We only talk about three different types of joins here, i.e., inner join, left join, and right join. Left join and right join are also referred to as outer join.

Let's make two tables to join.

R

[7]https://en.wikipedia.org/wiki/Join_(SQL)

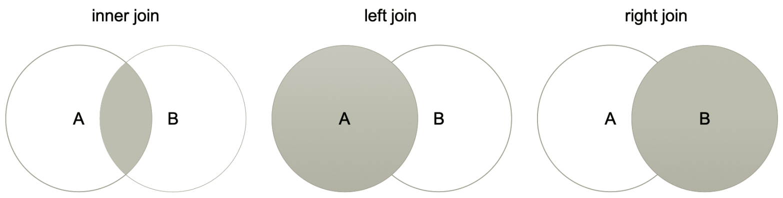


Figure 3.1: Inner join, left (outer) join and right (outer) join

```
> department_dt=data.table(department_id=c(1, 2, 3),
   department_name=c("Engineering","Operations","Sales"))
> department_dt
   department_id department_name
1:             1     Engineering
2:             2      Operations
3:             3           Sales
> employee_dt=data.table(employee_id=c(1,2,3,4,5,6),
   department_id=c(1,2,2,3,1,4))
> employee_dt
   employee_id department_id
1:           1             1
2:           2             2
3:           3             2
4:           4             3
5:           5             1
6:           6             4
```

Python

```
>>> department_df=pd.DataFrame({'department_id':[1,2,3],'
   department_name':["Engineering","Operations","Sales"]})
>>> department_df
   department_id department_name
0              1     Engineering
1              2      Operations
2              3           Sales
>>> employee_df=pd.DataFrame({'employee_id':[1,2,3,4,5,6],'
   department_id':[1,2,2,3,1,4]})
>>> employee_df
   employee_id  department_id
0            1              1
1            2              2
2            3              2
```

```
3              4              3
4              5              1
5              6              4
```

We can join tables with or without the help of index/key. In general, joining on index/key is more fast. Thus, I recommend always to set key/index for join operations.

R

```
> setkey(employee_dt, 'department_id')
> setkey(department_dt, 'department_id')
> department_dt[employee_dt] # employee_dt left join
    department_dt
   department_id department_name employee_id
1:             1     Engineering           1
2:             1     Engineering           5
3:             2      Operations           2
4:             2      Operations           3
5:             3           Sales           4
6:             4            <NA>           6
> department_dt[employee_dt, nomatch=0] # employee_dt left join
    department_dt
   department_id department_name employee_id
1:             1     Engineering           1
2:             1     Engineering           5
3:             2      Operations           2
4:             2      Operations           3
5:             3           Sales           4
> employee_dt[department_dt] # employee_dt right join
    department_dt
   employee_id department_id department_name
1:           1             1     Engineering
2:           5             1     Engineering
3:           2             2      Operations
4:           3             2      Operations
5:           4             3           Sales
```

To join `data.table` A and B, the syntax `A[B]` and `B[A]` only work when the keys of A and B are set. The function `merge` from `data.table` package works with or without key.

R

```
> # merge function works with keys
> merge(employee_dt,department_dt, all.x=FALSE, all.y=FALSE) #
    inner join
   department_id employee_id department_name
1:             1           1     Engineering
2:             1           5     Engineering
3:             2           2      Operations
4:             2           3      Operations
5:             3           4           Sales
> merge(employee_dt, department_dt, all.x=TRUE, all.y=FALSE) #
    employee_dt left join department_dt
   department_id employee_id department_name
1:             1           1     Engineering
2:             1           5     Engineering
3:             2           2      Operations
4:             2           3      Operations
5:             3           4           Sales
6:             4           6            <NA>
> merge(employee_dt, department_dt, all.x=FALSE, all.y=TRUE) #
    employee_dt right join department_dt
   department_id employee_id department_name
1:             1           1     Engineering
2:             1           5     Engineering
3:             2           2      Operations
4:             2           3      Operations
5:             3           4           Sales
> # merge function works without keys
> setkey(employee_dt, NULL)
> setkey(department_dt, NULL)
> department_dt[employee_dt]
Error in `[.data.table`(department_dt, employee_dt) :
  When i is a data.table (or character vector), the columns to
      join by must be specified either using 'on=' argument (see
       ?data.table) or by keying x (i.e. sorted, and, marked as
      sorted, see ?setkey). Keyed joins might have further speed
       benefits on very large data due to x being sorted in RAM.

> merge(employee_dt, department_dt, all.x=FALSE, all.y=FALSE, by
    .x='department_id', by.y='department_id')
   department_id employee_id department_name
1:             1           1     Engineering
2:             1           5     Engineering
3:             2           2      Operations
4:             2           3      Operations
5:             3           4           Sales
>
```

Table 3.2: Syntax of join in data.table

type	syntax 1	syntax 2
A inner join B	A[B, nomatch=0] or B[A, nomatch=0]	merge(A, B, all.x=FALSE, all.y=FALSE)
A left join B	B[A]	merge(A, B, all.x=TRUE, all.y=FALSE)
B right join A	A[B]	merge(A, B, all.x=FALSE, all.y=TRUE)

The join operations for `data.table` are summarized in Table 3.2.

In `pandas`, there are also different ways to join `DataFrame`. Let's just focus on the basic method with the `merge` function (other methods may also be based on this function).

Python

```
>>> department_df.set_index('department_id', inplace=True)
>>> employee_df.set_index('department_id', inplace=True)
>>> pd.merge(employee_df, department_df, how='inner', left_index
    =True, right_index=True) # inner join on index
               employee_id department_name
department_id
1                        1     Engineering
1                        5     Engineering
2                        2      Operations
2                        3      Operations
3                        4           Sales
>>> pd.merge(employee_df, department_df, how='left', left_index=
    True, right_index=True) # left join on index
               employee_id department_name
department_id
1                        1     Engineering
1                        5     Engineering
2                        2      Operations
2                        3      Operations
3                        4           Sales
4                        6             NaN
>>> pd.merge(employee_df, department_df, how='right', left_index
    =True, right_index=True) # right join on index
               employee_id department_name
department_id
1                        1     Engineering
1                        5     Engineering
2                        2      Operations
```

```
2                         3      Operations
3                         4           Sales
>>> employee_df.reset_index()
>>> department_df.reset_index()
>>> pd.merge(employee_df, department_df, how='inner', left_on='
    department_id', right_on='department_id') # join with
    columns directly (not on index)
               employee_id department_name
department_id
1                         1     Engineering
1                         5     Engineering
2                         2      Operations
2                         3      Operations
3                         4           Sales
```

We have learned the very basics of `data.table` and `pandas`. In fact, there are lots of other useful features in both tools which are not covered in this chapter. For example, the `.I/.N` symbol in `data.table`, and the `stack/unstack` method in `pandas`.

CHAPTER 4

Random Variables, Distributions & Linear Regression

We will see a few topics related to random variables (r.v.) and distributions in this chapter. Based on the concepts of random variables, we will further introduce linear regression models.

4.1 A REFRESHER ON DISTRIBUTIONS

Distributions describe the random phenomenon in terms of probabilities. Distributions are connected to random variables. A random variable is specified by its distribution. We know an r.v. can be either discrete or continuous. The number of calls received by a call center is an example of discrete r.v., and the amount of time taken to read this book is a continuous random variable.

There are infinite numbers of distributions. Many important distributions fall into some distribution families (i.e., a parametric set of probability distributions of a certain form). For example, the multivariate normal distribution belongs to exponential distribution family[1].

Cumulative distribution function (CDF) specifies how a real-valued random variable X is distributed, i.e.,

$$F_X(x) = P(X \leq x). \tag{4.1}$$

It is worth noting that CDF is always monotonic.

When the derivative of CDF exists, we call the derivative of F the probability density function (PDF) which is denoted by the lower case f_X. PDF is associated with continuous random variables. For discrete random variables, the counterpart of PDF is called probability mass function (PMF). PMF gives the probability that a

[1]https://en.wikipedia.org/wiki/Exponential_family

random variable is equal to some value, but PDF does not represent probabilities directly.

The quantile function is the inverse of CDF, i.e.,

$$Q_X(p) = inf\{x \in \mathbb{R} : p \leq F_X(x)\}. \tag{4.2}$$

PDF, CDF and quantile functions are all heavily used in quantitative analysis. In R and Python, we can find a number of functions to evaluate these functions. The best known distribution should be univariate normal/Gaussian distribution. Let's use Gaussian random variables for illustration.

R

```
> # for reproducibility
> set.seed(42)
> # draw samples
> x = rnorm(10, mean=0, sd=2)
> print(x)
 [1]  2.7419169 -1.1293963  0.7262568  1.2657252  0.8085366
 [6] -0.2122490  3.0230440 -0.1893181  4.0368474 -0.1254282
> print(mean(x))
[1] 1.094594
> print(sd(x))
[1] 1.670898
> # evaluate PDF
> d = dnorm(x, mean=0, sd=2)
> print(d)
 [1] 0.07793741 0.17007297 0.18674395 0.16327110 0.18381928
 [6] 0.19835103 0.06364496 0.19857948 0.02601446 0.19907926
> # evalute CDF
> p = pnorm(x, mean=0, sd=2)
> print(p)
 [1] 0.9148060 0.2861395 0.6417455 0.7365883 0.6569923 0.4577418
 [7] 0.9346722 0.4622928 0.9782264 0.4749971
> # evaluate quantile
> q = qnorm(p, mean=0, sd=2)
> print(q)
 [1]  2.7419169 -1.1293963  0.7262568  1.2657252  0.8085366
 [6] -0.2122490  3.0230440 -0.1893181  4.0368474 -0.1254282
```

We see that $qnorm(pnorm(x)) = x$. In Python, we use the functions in `numpy` and `scipy.stats` for the same purpose.

Python

```
>>> import scipy.stats
>>> import numpy as np
>>> import scipy.stats
>>> np.random.seed(42)
>>> x = np.random.normal(loc=0.0, scale=2.0, size=10)
>>> print(x)
[ 0.99342831 -0.2765286   1.29537708  3.04605971 -0.46830675
   -0.46827391
  3.15842563  1.53486946 -0.93894877  1.08512009]
>>> norm = scipy.stats.norm(loc=0.0, scale=2.0)
>>> d = norm.pdf(x)
>>> print(d)
[0.17632116 0.19757358 0.16172856 0.06254333 0.19407713
   0.19407788
 0.05732363 0.1485901  0.17865677 0.17217026]
>>> p = norm.cdf(x)
>>> print(p)
[0.69030468 0.44501577 0.74140679 0.93612438 0.40743296
   0.40743933
 0.94285637 0.77858846 0.31936529 0.70628362]
>>> q = norm.ppf(p) # ppf is the quantile function
>>> print(q)
[ 0.99342831 -0.2765286   1.29537708  3.04605971 -0.46830675
   -0.46827391
  3.15842563  1.53486946 -0.93894877  1.08512009]
```

A random variable could also be multivariate. In fact, the univariate normal distribution is a special case of multivariate normal distribution whose PDF is given in

$$p(\boldsymbol{x};\boldsymbol{\mu},\boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{m/2}|\boldsymbol{\Sigma}|^{1/2}} \exp(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})), \tag{4.3}$$

where $\boldsymbol{\mu}$ is the mean and $\boldsymbol{\Sigma}$ is the covariance matrix of the random variable $\boldsymbol{x}$.

Sampling from distributions are involved in many algorithms, such as Monte Carlo simulation. First, let's see a simple example in which we draw samples from a 3-dimensional normal distribution.

R

```
> library(mvtnorm)
> mu = c(1, 2)
> cov = array(
   c(1.0, 0.5, 0.5, 2.0),
   c(2, 2))
```

Python

```
>>> from scipy.stats import
    multivariate_normal
    as mvn
>>> mu = np.array([1,2])
```

```
> set.seed(42)
> x = rmvnorm(n = 1,
    mean = mu, sigma = cov)
> print(x)
         [,1]     [,2]
[1,] 2.221431 1.498778
> d = dmvnorm(x,
    mean = mu, sigma = cov)
> print(d)
[1] 0.04007949
> p = pmvnorm(lower=-Inf,
    upper = x[1,],
    mean = mu, sigma = cov)
> print(p)
[1] 0.344201
attr(,"error")
[1] 1e-15
attr(,"msg")
[1] "Normal Completion"
```

```
>>> cov = np.array([[1,
    0.5],[0.5, 2]])
>>> mvn.random_state.seed
    (42)
>>> dist = mvn(mean = mu,
    cov = cov)
>>> x = dist.rvs(size=1)
>>> print(x)
[1.16865045 2.72887797]
>>> d = dist.pdf(x)
>>> print(d)
0.10533537790438291
>>> p = dist.cdf(x)
>>> print(p)
0.44523665187828293
```

Please note in the example above, we do not calculate the quantiles. For multivariable distributions, the quantiles are not necessarily fixed points.

4.2 INVERSION SAMPLING & REJECTION SAMPLING

4.2.1 Inversion sampling

Sampling from the Gaussian or other famous distributions could be as simple as just calling a function. What if we want to draw samples from any distribution with its CDF? Inversion sampling is a generic solution. The key idea of inversion sampling is that $F_X(x)$ always follows a uniform distribution between 0 and 1. There are two steps in drawing a sample with the inversion sampling approach.

1. draw independent and identical (i.i.d.) distribution random variables $u_1, ..., u_n$ which are uniformly distributed from 0 to 1;
2. calculate the quantiles $q_1, ..., q_n$ for $u_1, ..., u_n$ based on the CDF, and return $q_1, ..., q_n$ as the desired samples to draw.

Let's see how to use the inversion sampling technique to sample from exponential distribution with CDF $f_X(x; \lambda) = 1 - e^{-\lambda x}$.

R

```
> n=1000
> set.seed(42)
> # step 1, draw from
    uniform dist
```

Python

```
>>> import numpy as np
>>> np.random.seed(42)
>>> n=1000
```

```r
> u = runif(n=n, min=0,
    max=1)
> # step 2, calculate the
    quantiles
> lambda = 2.0
> x = qexp(u, rate = lambda)
> # the mean is close to 1/
    lambda
> mean(x)
[1] 0.4818323
```

```python
>>> # step 1, generate from
    uniform(0,1)
... u = np.random.uniform(
    low=0, high=1, size=n)
>>> # step 2, calculate the
    quantile
... lamb = 2.0
>>> q = -np.log(1-u)/lamb
>>> np.mean(q)
0.4862529739826122
```

In the above R implementation, we used the built-in quantile function in step 2; however, for many distributions there are no built-in quantile functions available and thus we need to specify the quantile function ourselves (illustrated in the Python implementation).

4.2.2 Rejection sampling

Rejection sampling is also a basic algorithm to draw samples for a random variable X given its PDF f_X. The basic idea of rejection sampling is to draw samples for a random variable Y with PDF f_Y and accept the samples with probability $f_X(x)/(Mf_Y(x))$. M is selected such that $f_X(x)/(Mf_Y(x)) \leq 1$. If the sample generated is rejected, the sampling procedure is repeated until an acceptance. More theoretical details of rejection sampling can be found from wikipedia[2]. The distribution f_Y is called proposal distribution.

Let's try to draw samples from an exponential distribution truncated between 0 and b. The PDF of this random variable is specified by $f_X(x;\lambda,b) = \lambda e^{-\lambda x}/(1-e^{-\lambda b})$.

A naive approach is to sample from the untruncated exponential distribution and only accept samples smaller than b, which is implemented as follows.

R

```r
n=1000
set.seed(42)
lambda = 1.0
b = 2.0
x = rep(0, n)
i = 0
while (i<n){
  y = rexp(n = 1, rate =
      lambda)
  if (y<=b) {
    i=i+1
    x[i] = y
  }
```

Python

```python
import numpy as np
np.random.seed(42)
lamb, b = 1.0, 2.0
n=1000
x = []
i = 0
while i<n:
  y = np.random.exponential(
      scale = lamb, size = 1)
  if y<=b:
    x.append(y[0])
    i+=1
```

[2]https://en.wikipedia.org/wiki/Rejection_sampling

```
}
```

After running the code snippets above, we have the samples stored in x from the truncated exponential distribution.

Now let's use the rejection sampling technique for this task. Since we want to sample the random variable between 0 and b, one natural choice of the proposal distribution f_Y is a uniform distribution between 0 and b and we choose $M = b\lambda/(1-e^{-\lambda b})$. As a result, the acceptance probability $f_X(x)/(Mf_Y(x))$ becomes $e^{-\lambda x}$.

R

```r
n=1000
set.seed(42)
lambda = 1.0
b = 2.0
x = rep(0, n)
i = 0
while (i<n){
  # sample from the proposal distribution
  y = runif(1, min = 0, max = b)
  # calculate the acceptance probability
  p = exp(-lambda*y)
  if (runif(1)<=p) {
    i=i+1
    x[i] = y
  }
}
```

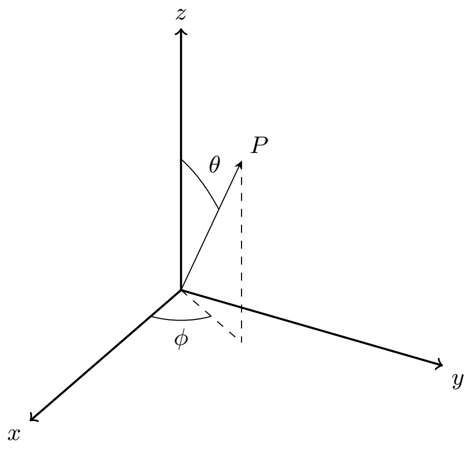
Python

```python
import numpy as np
np.random.seed(42)
lamb, b = 1.0, 2.0
n = 1000
x = []
i = 0
while i < n:
    # sample from the proposal distribution
    y = np.random.uniform(low=0, high=b)
    # calculate the acceptance probability
    p = np.exp(-lamb*y)
    if p >= np.random.uniform():
        x.append(y)
        i += 1
```

We have seen the basic examples on how to draw samples with inversion samples and truncated samples. Now let's work on a more challenging problem.

Application - Draw samples from a sphere

Without loss of generality, let's consider a unit sphere, i.e., the radius $r = 1$. We want to draw i.i.d. points from a unit sphere.



The problem appears simple at first glance - we could utilize the spherical coordinates system and draw samples for ϕ and θ. Now the question is how to sample for ϕ and θ. A straightforward idea is to draw independent and uniform samples ϕ from 0 to 2π and θ from 0 to π, respectively. However, this idea is incorrect which will be analyzed below.
Let's use $f_P(\phi,\theta)$ to denote the PDF of the joint distribution of (ϕ,θ). We integrate this PDF, then

$$1=\int_0^{2\pi}\int_0^{\pi} f_P(\phi,\theta)d\phi d\theta=\int_0^{2\pi}\int_0^{\pi} f_\Phi(\phi)f_{\Theta|\Phi}(\theta|\phi)d\phi d\theta. \tag{4.4}$$

If we enforce Φ has a uniform distribution between 0 and 2π, then $f_\Phi(\phi)=1/2\pi$, and

$$1=\int_0^{\pi} f_{\Theta|\Phi}(\theta|\phi)d\theta. \tag{4.5}$$

One solution to (4.5) is $f_{\Theta|\Phi}(\theta|\phi)=sin(\theta)/2$.
Thus, we could generate the samples of Φ from the uniform distribution and the samples of Θ from the distribution whose PDF is $sin(\phi)/2$. Sampling for Φ is trivial, but how about Θ? We could use the inversion sampling technique. The CDF of Θ is $(1-cos(\theta))/2; 0\le\theta\le\pi$, and the quantile function is $Q_\Theta(p)=arccos(1-2p)$.
The implementation of sampling from unit sphere is implemented below.

R

```r
n=2000
set.seed(42)
# sample phi
phi = runif(n, min = 0,
    max = 2*pi)
# sample theta by inversion
    cdf
u = runif(n, min = 0,
    max = 1)
theta = acos(1-2*u)
```

Python

```python
import numpy as np
np.random.seed(42)
n = 2000
# sample phi
phi = np.random.uniform(low
    =0, high=2*np.pi, size=n)
# sample theta by inversion
    cdf
u = np.random.uniform(low=0,
     high=1, size=n)
```

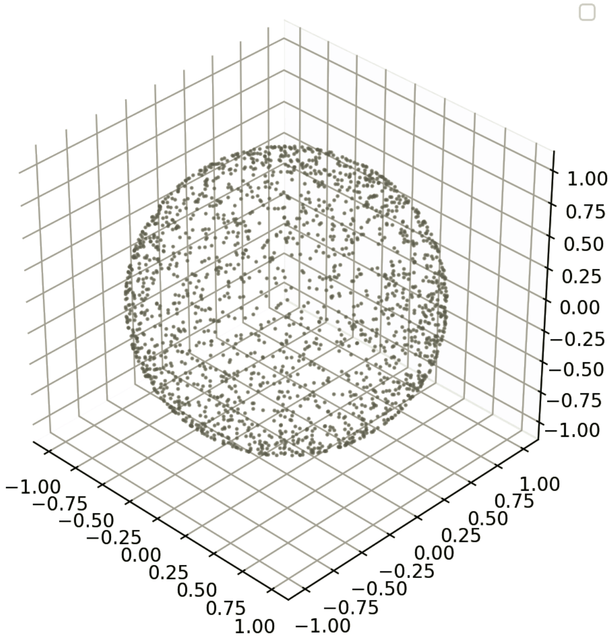


Figure 4.1: Uniform samples from a unit sphere

```r
# now we calculate the
    Cartesian coordinates
x = sin(theta)*cos(phi)
y = sin(theta)*sin(phi)
z = cos(theta)
```

```python
theta = np.arccos(1-2*u)
# we calculate the Cartesian
    coordinates
x = np.sin(theta)*np.cos(phi
    )
y = np.sin(theta)*np.sin(phi
    )
z = np.cos(theta)
```

There are also other solutions to this problem, which won't be discussed in this book. A related problem is to draw samples inside a sphere. We could solve the inside sphere sampling problem with a similar approach, or by using the rejection sampling approach, i.e., sampling from a cube with acceptance ratio $\pi/6$.

4.3 JOINT DISTRIBUTION & COPULA

We are not trying to introduce these concepts from scratch. This section is more like a recap.

In the previous section, we saw the PDF for multivariate normal distribution in (4.3). A multivariate distribution is also called joint distribution, since the multivariate random variable can be viewed as a joint of multiple univariate random variables. Joint PDF gives the probability density of a set of random variables. Sometimes we may only be interested in the probability distribution of a single random variable from a set. And that distribution is called marginal distribution. The PDF of a marginal distribution can be obtained by integrating the joint PDF over all the other random variables. For example, the integral of (4.3) gives the PDF of a univariate normal distribution.

The joint distribution is the distribution about the whole population. In the context of a bivariate Gaussian random variable (X_1, X_2), the joint PDF $f_{X_1,X_2}(x_1, x_2)$ specifies the probability density for all pairs of (X_1, X_2) in the 2-dimension plane. The marginal distribution of X_1 is still about the whole population because we are not ruling out any points from the support of the distribution function. Sometimes we are interested in a subpopulation only, for example, the subset of (X_1, X_2) where $X_2 = 2$ or $X_2 > 5$. We can use conditional distribution to describe the probability distribution of a subpopulation. To denote conditional distribution, the symbol | is frequently used. We use $f_{X_1|X_2=0}(x_1|x_2)$ to represent the distribution of X_1 conditional on $X_2 = 0$. By the rule of conditional probability $P(A|B) = P(A, B)/P(B)$, the calculation $f_{X_1|X_2}(x_1|x_2)$ is straightforward, i.e., $f_{X_1|X_2}(x_1|x_2) = f_{X_1,X_2}(x_1, x_2)/f_{X_2}(x_2)$.

The most well-known joint distribution is the multivariate Gaussian distribution. Multivariate Gaussian distribution has many important and useful properties. For example, given the observation of $(X_1, ..., X_k)$ from $(X_1, ..., X_m)$, $(X_k + 1, ..., X_m)$ is still following a multivariate Gaussian distribution, which is essential to Gaussian process regression[3].

We have seen the extension from univariate Gaussian distribution to multivariate Gaussian distribution, but how about other distributions? For example, what is the joint distribution for two univariate exponential distribution? We could use copula[4] for such a purpose. For the random variable $(X_1, ..., X_m)$, let $(U_1, ..., U_m) = (F_{X_1}(X_1), ..., F_{X_m}(X_m))$ where F_{X_k} is the CDF of X_k. We know U_k is following a uniform distribution. Let $C(U_1, ..., U_m)$ denote the joint CDF of $(U_1, ..., U_m)$ and the CDF is called copula.

There are different copula functions, and one commonly used is the Gaussian copula. The standard Gaussian copula is specified as below.

$$C_{\Sigma}^{Gauss}(u_1, ..., u_m) = \Phi_{\Sigma}(\Phi^{-1}(u_1), ..., \Phi^{-1}(u_m)), \tag{4.6}$$

where Φ denotes the CDF of the standard Gaussian distribution, and Φ_Σ denotes the CDF of a multivariate Gaussian distribution with mean $\mathbf{0}$ and correlation matrix Σ.

Let's see an example to draw samples from a bivariate exponential distribution constructed via Gaussian copula. The basic idea of sampling multivariate random variables via copula is to sample $U_1, ..., U_m$ first and then transform it to the desired random variables.

[3]http://www.gaussianprocess.org/gpml/chapters/RW2.pdf

[4]https://en.wikipedia.org/wiki/Copula_(probability_theory)

R

```
> library(mvtnorm)
> n = 10000
> set.seed(42)
> rates = c(1.0, 2.0)
> mu = c(0, 0)
> rho = 0.6
> correlation = array(c(1, rho, rho, 1), c(2, 2))
> # sample R from multivariate Gaussian
> r = rmvnorm(n = n, mean = mu, sigma = correlation)
> # generate U
> u = pnorm(r)
> # calculate the quantile for X
> x1 = qexp(u[, 1], rate = rates[1])
> x2 = qexp(u[, 2], rate = rates[2])
> x = cbind(x1, x2)
> cor(x)
          x1        x2
x1 1.0000000 0.5476137
x2 0.5476137 1.0000000
> apply(x, mean, MARGIN = 2)
       x1        x2
0.9934398 0.4990758
```

Python

```
>>> import numpy as np
>>> from scipy.stats import multivariate_normal as mvn
>>> from scipy.stats import norm
>>> mu = np.array([0, 0])
>>> rho = 0.6
>>> cov = np.array([[1, rho],[rho, 1]])
>>> mvn.random_state.seed(42)
>>> n = 10000
>>> uvn = norm(loc=0.0, scale=1.0)
>>> dist = mvn(mean = mu, cov = cov)
>>> rates = [1.0, 2.0]
>>> # sample R from multivariate Gaussian
... r = dist.rvs(size=n)
>>> # generate U
... u = uvn.cdf(r)
>>> # calculate the quantile for X
... def qexp(u, rate): return -np.log(1-u)/rate
...
```

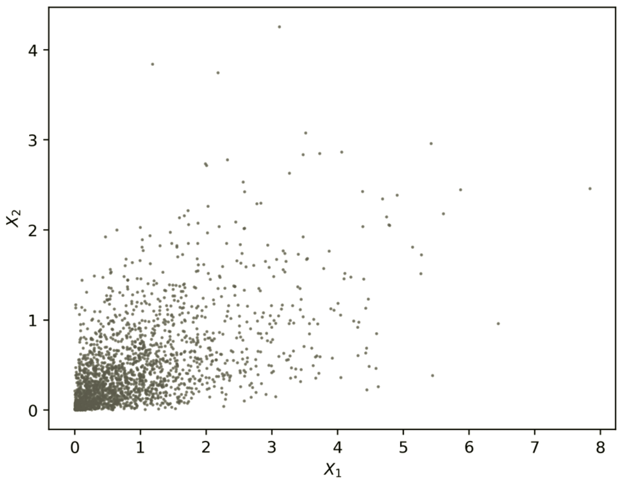


Figure 4.2: Samples from a bivariate exponential distribution constructed via Gaussian copula

```
>>> x1 = qexp(u[:, 0], rate = rates[0])
>>> x2 = qexp(u[:, 1], rate = rates[1])
>>> x = np.column_stack((x1, x2))
>>> np.corrcoef(x.T)
array([[1.        , 0.57359023],
       [0.57359023, 1.        ]])
>>> np.mean(x, axis=0)
array([0.99768825, 0.50319692])
```

We plot 2000 samples generated from the bivariate exponential distribution constructed via copula in Figure 4.2.

With the help of copula, we can even construct joint distribution with marginals from different distributions. For example, let's make a joint distribution of a uniform distributed random variable and an exponential distributed random variable.

R

```
> library(mvtnorm)
> n = 10000
> set.seed(42)
> mu = c(0, 0)
> rho = 0.6
> correlation = array(c(1, rho, rho, 1), c(2, 2))
> # sample R from multivariate Gaussian
> r = rmvnorm(n = n, mean = mu, sigma = correlation)
> # generate U
> u = pnorm(r)
```

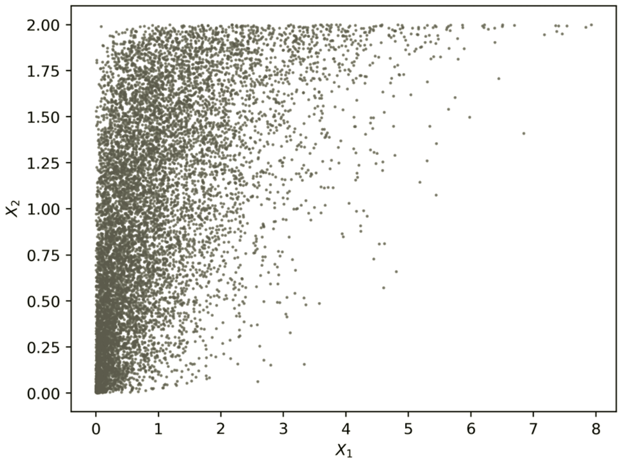


Figure 4.3: Samples from a joint distribution of a uniform marginal and an exponential marginal

```
> # calculate the quantile for X
> # X1 ~ exp(1.0); X2 ~ unif(0, 2)
> x1 = qexp(u[, 1], rate = 1.0)
> x2 = qunif(u[, 2], min = 0, max = 2)
> cor(x1, x2)
[1] 0.5220363
```

4.4 FIT A DISTRIBUTION

Statistics is used to solved real-world problems with data. In many cases we may have a collection of observations for a random variable and want to know the distribution which the observations follow. In fact, there are two questions involved in the process of fitting a distribution. First, which distribution to fit? And second, given the distribution, how to estimate the parameters. These two questions are essentially the same questions that we have to answer in supervised learning. In supervised learning, we need to choose a model and estimate the parameters (if the model has parameters). We can also refer to these two questions as model selection and model fitting. Usually, model selection is done based on the model fitting.

Two widely-used methods in distribution fitting are method of moments and the maximum likelihood method. In this section we will see the method of moments. The maximum likelihood method will be introduced in Chapter 6. The k^{th} moment of a random variable is defined as $\mu_k = E(x^k)$. If there are m parameters, usually we derive the first m theoretical moments in terms of the parameters, and by equating these theoretical moments to the sample moments $\hat{\mu}_k = 1/n\sum_1^n x_i^k$ we will get the estimate.

Let's take the univariate Gaussian distribution as an example. We want to estimate the mean μ and variance σ^2. The first and second theoretical moments are μ and $\mu^2+\sigma^2$. Thus, the estimate $\hat{\mu}$ and $\hat{\sigma}^2$ are $1/n \quad {}_1^n x_i$ and $1/n \quad {}_1^n x_i^2 - (1/n \quad {}_1^n x_i)^2 = 1/n \quad {}_1^n (x_i - \hat{\mu})^2$. The code snippets below show the implementation.

R

```r
> set.seed(42)
> n = 1000
> mu = 2.5
> var = 1.6
> x = rnorm(n, mu,
    sqrt(var))
> mu_est = mean(x)
> var_est =
    mean((x-mu_est)^2)
> print(mu_est)
[1] 2.467334
> print(var_est)
[1] 1.60647
```

Python

```python
>>> import numpy as np
>>> np.random.seed(42)
>>> n = 1000
>>> mu, var = 2.5, 1.6
>>> x = np.random.normal
    (2.5, 1.6**0.5, size=n)
>>> mu_est = np.mean(x)
>>> var_est = np.mean((x-
    mu_est)**2)
>>> print(mu_est)
2.524453331300827
>>> print(var_est)
1.5326479835704276
```

We could also fit another distribution to the data generated from a normal distribution. But which one is better? One answer is to compare the likelihood functions evaluated at the fitted parameters and choose the one that gives the larger likelihood value.

Please note that different methods to fit a distribution may lead to different parameter estimates. For example, the estimate of population variance using the maximum likelihood method is different from that using the moments method. Actually, the estimator for population mean is biased using the methods method but the estimator using the maximum likelihood method is unbiased.

4.5 CONFIDENCE INTERVAL

In the previous section we have seen the parameter estimation of a distribution. However, the parameter estimates from either the method of moments or the maximum likelihood estimation are not the exact values of the unknown parameters. There are uncertainties associated with distribution fitting because the data to which the distribution is fit usually is just a random sample rather than a population. Suppose we could repeat the distribution fitting process m times and each time we collect a random sample of size n (i.e., a collection of n observations for the random variable of interest), then we will get n estimates of the distribution parameters. Which estimate is the best to use? In fact, all these n estimates are observations of the estimator random variable. Estimator is a function of the random sample, and itself is a random variable. For example, eq1 is an estimator for the μ parameter in a Gaussian distribution.

4.5.1 Central limit theorem

Now we know when we fit a distribution, the parameter estimates are not the exact values of the unknown parameters. The question is how to quantify the uncertainties? To answer this question, we better know the distribution of the estimator. In the example of the Gaussian distribution, what distribution does the estimator $\hat{\mu}$ follow? It is straightforward to see that the estimator is still following a Gaussian distribution since each X_i is following a Gaussian distribution (sum of independent Gaussian random variables still follows a Gaussian distribution). But what if X_i is from an arbitrary distribution? We may still derive an exact distribution of the sample mean $\hat{\mu}$ for an arbitrary distribution, but sometimes the derivation may not be easy. A simple idea is to use the central limit theorem (CLT), which states that the distribution of the mean of a random sample from a population with finite variance is approximately normally distributed when the sample size is large enough. More specifically, $\sqrt{n}(\bar{X}-\mu)/\sigma \xrightarrow{d} N(0,1)$ where μ and σ are the population mean and standard deviation, respectively. Sometimes we do not have the actual value of the population standard deviation and the sample standard deviation S can be used instead and thus $\sqrt{n}(\bar{X}-\mu)/S \xrightarrow{d} N(0,1)$.

We know if Z is a Gaussian random variable, $P(-Z_{(1+\alpha)/2} < Z \leq Z_{(1+\alpha)/2}) = \alpha$ where Z_u denotes the quantile of u. By CLT, $P(-Z_{(1+\alpha)/2} \leq \sqrt{n}(\bar{X}-\mu)/S \leq Z_{(1+\alpha)/2}) = \alpha$, which further leads to $P(\bar{X} - Z_{(1+\alpha)/2}S/\sqrt{n} \leq \mu \leq \bar{X} + Z_{(1+\alpha)/2}S/\sqrt{n}) = \alpha$. The interval $\bar{X} \pm Z_{(1+\alpha)/2}S/\sqrt{n}$ is called an α confidence interval (CI) for the population mean μ. For example, since $Z_{(1+0.95)/2} = 1.96$ the 95% CI is constructed as $\bar{X} \pm 1.96S/\sqrt{n}$. Of course, if we know the exact value of σ, we could use $\bar{X} \pm 1.96\sigma/\sqrt{n}$ instead.

We show an example for the confidence interval calculation of population mean in normal distribution.

R

```r
> set.seed(42)
> n = 1000
> mu = 2.5
> var = 1.6
> x = rnorm(n, mu, sqrt(var))
> mu_est = mean(x)
> # we can also calculate S with sd(x) in R
> S = sqrt(sum((x - mu_est) ^ 2) / (n - 1))
> alpha = 0.95
> Z_alpha = qnorm((1 + alpha) / 2)
> CI = c(mu_est - Z_alpha * S / sqrt(n), mu_est + Z_alpha * S /
    sqrt(n))
> print(CI)
[1] 2.388738 2.545931
```

Python

```python
>>> import numpy as np
>>> import scipy
>>> np.random.seed(42)
>>> n = 1000
>>> mu, var = 2.5, 1.6
>>> norm = scipy.stats.norm(loc=0, scale=1)
>>> x = np.random.normal(mu, var**0.5, size=n)
>>> mu_est = np.mean(x)
>>> S = np.sqrt(np.sum((x-mu_est)**2)/(n-1))
>>> alpha = 0.95
>>> Z_alpha = norm.ppf((1+alpha)/2)
>>> CI = [mu_est - Z_alpha*S/np.sqrt(n), mu_est + Z_alpha*S/np.
    sqrt(n)]
>>> print(CI)
[2.4476842124240363, 2.6012224501776178]
```

The interpretation of CI is tricky. A 95% CI does not mean the probability that the constructed CI contains the true population mean is 0.95. Actually, a constructed CI again is a random variable because the CI is created based on each random sample collected. Following the classic explanation from textbooks, when we repeat the procedures to create CI multiple times, the probability that the true parameter falls into the CI is equal to α. Let's do an example to see that point.

R

```r
> set.seed(42)
> B = 1000
> n = 1000
> mu = 2.5
> var = 1.6
> alpha = 0.95
> Z_alpha = qnorm((1 + alpha) / 2)
> coverage = rep(0, B)
> for (i in 1:B) {
+   x = rnorm(n, mu, sqrt(var))
+   mu_est = mean(x)
+   S = sqrt(sum((x - mu_est) ^ 2) / (n - 1))
+   CI = c(mu_est - Z_alpha * S / sqrt(n), mu_est + Z_alpha * S
    / sqrt(n))
+   coverage[i] = (mu >= CI[1]) && (mu <= CI[2])
+ }
> # the coverage probability should be close to alpha
> print(mean(coverage))
[1] 0.941
```

4.5.2 Bootstrap

So far we have seen how to create the CI for sample mean. What if we are interested in quantifying the uncertainty of other parameters, for example, the variance of a random variable? If we estimate these parameters with the maximum likelihood method, we can still construct the CI in a similar approach with the large sample theory. However, we will not discuss it in this book. Alternatively, we could use the bootstrap technique.

Bootstrap is simple yet powerful. It is a simulation-based technique. If we want to estimate a quantity θ, first we write the estimator for θ as a function of a random sample i.e., $\hat{\theta} = g(X_1, ..., X_n)$. Next, we just draw a random sample and calculate $\hat{\theta}$ and repeat this process B times to get a collection of $\hat{\theta}$ denoted as $\hat{\theta}^{(1)}, ..., \hat{\theta}^{(B)}$. From these simulated $\hat{\theta}$, we could simply use the percentile $\hat{\theta}_{(1-\alpha)/2}$ and $\hat{\theta}_{(1+\alpha)/2}$ to construct the α CI. There are also other variants of the bootstrapping method with similar ideas.

Let's try to use bootstrap to construct a 95% CI for the population variance of a Gaussian distributed random variable.

R

```r
> set.seed(42)
> B = 1000
> n = 1000
> mu = 2.5
> var = 1.6
> alpha = 0.95
> Z_alpha = qnorm((1 + alpha) / 2)
> var_sample = rep(0, B)
> for (i in 1:B) {
+   x = rnorm(n, mu, sqrt(var))
+   var_sample[i] = sum((x-mean(x))^2)/(n-1)
+ }
> quantile(var_sample, c((1-alpha)/2, (1+alpha)/2))
    2.5%    97.5%
1.466778 1.738675
```

4.6 HYPOTHESIS TESTING

We have talked about confidence interval, which is used to quantify the uncertainty in parameter estimation. The root cause of uncertainty in parameter estimation is that we do the inference based on random samples. Hypothesis testing is another technique related to confidence interval calculation.

A hypothesis test is an assertion about populations based on random samples. The assertion could be for a single population or multiple populations. When we collect a random sample, we may try to use the random sample as evidence to judge the hypothesis. Usually a hypothesis testing consists of two hypotheses.

- H_0: the null hypothesis;
- H_1: the alternative hypothesis.

When we perform a hypothesis testing, there are two possible outcomes, i.e., a) reject H_0 if the evidence is likely to support the alternative hypothesis, and b) do not reject H_0 because of insufficient evidence.

The key point to understand in hypothesis testing is the significant level, which is usually denoted as α. When the null hypothesis is true, the rejection of null hypothesis is called type I error. And the significance level is the probability of committing a type I error. When the alternative hypothesis is true, the acceptance of null hypothesis is called type II error. And the probability of committing a type II error is denoted as β. $1-\beta$ is called the power of a test.

To conduct a hypothesis testing, there are a few steps to follow. First we have to specify the null and alternative hypotheses, and the significance level. Next, we calculate the test statistic based on the data collected. Finally, we calculate the p-value. If the p-value is smaller than the significance level, we reject the null hypothesis; otherwise we accept it. Some books may describe a procedure to compare the test statistic with a critic region, which is essentially the same as the p-value approach. The real challenge to conduct a hypothesis testing is to calculate the p-value, whose calculation depends on which hypothesis test to use. Please note that the p-value itself is a random variable since it is calculated from the random sample. And when the null hypothesis is true, the distribution of p-value is uniform from 0 to 1.

p-value is also a conditional probability. A major misinterpretation about p-value is that it is the conditional probability that given the observed data the null hypothesis is true. Actually, p-value is the probability of the observation given the null hypothesis is true.

For many reasons, we will not go in-depth into the calculation of p-values in this book. But the basic idea is to figure out the statistical distribution of the test statistic. Let's skip all the theories behind and go to the tools in R/Python.

4.6.1 One-sample t test

Probably one-sample t test is the most basic and useful hypothesis tests. It can determine if the sample mean is statistically different from a hypothesized population mean for continuous random variables. To use one-sample t test some assumptions are usually required. For example, the observations should be independent. Another assumption is the normality, i.e., the population should be normal distributed or approximately normal distributed. The normality assumption is controversial but it is beyond the scope of this book.

In one-sample t test, the alternative hypothesis could be two-sided, or one-sided.

A two-sided H_1 does not specify if the population mean is greater or smaller than the hypothesized population mean. In contrast, a one-sided H_1 specifies the direction.

Now let's see how to perform one-sample t test in R/Python.

R

```
> set.seed(42)
> n = 50
> mu = 2.5
> sigma = 1.0
> # set the significance level
> alpha = 0.05
> # sample 50 independent observations from a normal
    distribution
> x = rnorm(n, mu, sigma)
> mu_test = 2.5
>
> # first, let's do two-sided t test
> # H0: the population mean is equal to mu_test
> # H1: the population mean is not equal to mu_test
> t1 = t.test(x, alternative = "two.sided", mu = mu_test)
> print(t1)

  One Sample t-test

data:  x
t = -0.21906, df = 49, p-value = 0.8275
alternative hypothesis: true mean is not equal to 2.5
95 percent confidence interval:
 2.137082 2.791574
sample estimates:
mean of x
 2.464328

> # based on the p-value (> alpha), we accept the null
    hypothesis stated above
> t1$value
NULL
>
> # next, let's do a one-sided t test
> # H0: the population mean is equal to mu_test
> # H1: the population mean is greater than mu_test
> t2 = t.test(x, alternative = "greater", mu = mu_test)
> print(t2)

```

```
  One Sample t-test

data:  x
t = -0.21906, df = 49, p-value = 0.5862
alternative hypothesis: true mean is greater than 2.5
95 percent confidence interval:
 2.191313      Inf
sample estimates:
mean of x
 2.464328

> # based on the p-value (> alpha), we accept the null
    hypothesis stated above
> t2$p.value
[1] 0.5862418
>
> # let's change mu_test and perform a one-sided test again
> mu_tes_new = 2.0
> t3 = t.test(x, alternative = "greater", mu = mu_tes_new)
> print(t3)

  One Sample t-test

data:  x
t = 2.8514, df = 49, p-value = 0.003177
alternative hypothesis: true mean is greater than 2
95 percent confidence interval:
 2.191313      Inf
sample estimates:
mean of x
 2.464328

> # based on the p-value (< alpha), we reject the null
    hypothesis stated above, i.e., we conclude the population
    mean is greater than 2.
> t3$p.value
[1] 0.003177329
```

Python

```
>>> import numpy as np
>>> from scipy.stats import ttest_1samp
>>> np.random.seed(42)
>>> n = 50
>>> mu, sigma = 2.5, 1.0
>>> # generate 50 independent samples from a normal distribution
... x = np.random.normal(mu, sigma, size=n)
>>> alpha = 0.05
>>> mu_test = 2.5
>>> # H0: population mean equals mu_test
... # H1: population mean does not equal to mu_test
... # perform a two-sided t test
... t1 = ttest_1samp(x, mu_test)
>>> # based on the p-value (> alpha), we accept the null
   hypothesis stated aboveprint(t1.pvalue)
... print(t1.pvalue)
0.09403837414922567
```

In the R code snippet we show both one-sided and two-sided one-sample t tests. However, we only show a two-sided test in the Python program. It is feasible to perform a one-sided test in an indirect manner with the same function, but I don't think it's worth discussing here. For hypothesis testing, it seems R is a better choice than Python.

4.6.2 Two-sample t test

Two-sample t test is a bit more complex than one-sample t test. There are two types of t tests commonly used in practice, i.e., paired t test and unpaired t test. In paired t test, the samples are paired together. For example, we may want to know if there is a significant difference in human blood pressures in morning and in evening. Hypothesis testing may help. To do so, we may conduct an experiment and collect the blood pressures in morning and in evening from a number of participants, respectively. Let X_i denote the morning blood pressure and Y_i denote the evening blood pressure of participant i. We should pair X_i and Y_i since the pair is measured from the same person. Then the paired t test can be used to compare the population means. The null hypothesis usually states that the difference of two population means is equal to a hypothesized value. Just like the one-sample t test, we could do a one-sided or two-sided paired t test.

Now let's see how to do paired t test in R/Python.

R

```
> library(mvtnorm)
> set.seed(42)
> n = 50
> # we sample from a bivariate normal distribution
> mu = c(2.0, 1.0)
> sigma = diag(c(1.0, 1.5))
> # significance level
> alpha = 0.95
> # sample 50 independent observations from the normal
    distribution
> mu_diff = 1.2
> x = rmvnorm(n, mu, sigma)
>
> # first, let's do a two-sided t test
> # H0: the population means' difference is equal to mu_diff
> # H1: the population means' difference is not equal to mu_diff
> t.test(x=x[,1], y=x[,2], mu=mu_diff, paired = TRUE,
    alternative = "two.sided")

  Paired t-test

data:  x[, 1] and x[, 2]
t = -0.86424, df = 49, p-value = 0.3917
alternative hypothesis: true difference in means is not equal to
     1.2
95 percent confidence interval:
 0.5417625 1.4623353
sample estimates:
mean of the differences
               1.002049

> # we can't reject H0 based on the fact p-value>alpha
>
> # next, let's do an one-sided t test
> # H0: the population means' difference is equal to mu_diff
> # H1: the population means' difference is less than mu_diff
> t.test(x=x[,1], y=x[,2], mu=mu_diff, paired = TRUE,
    alternative = "less")

  Paired t-test

data:  x[, 1] and x[, 2]
t = -0.86424, df = 49, p-value = 0.1958
alternative hypothesis: true difference in means is less than
    1.2
95 percent confidence interval:
     -Inf 1.386057
sample estimates:
mean of the differences
               1.002049

> # we can't reject H0 based on the fact p-value>alpha
```

Paired t test can also be done in Python, but we will not show the examples.

Unpaired t test is also about two population means' difference, but the samples are not paired. For example, we may want to study if the average blood pressure of men is higher than that of women. In the unpaired t test, we also have to specify if we assume the two populations have equal variance or not.

R

```
> set.seed(42)
> n = 50
> alpha = 0.05
> mu1 = 2.0
> mu2 = 1.0
> sigma1 = 1.0
> sigma2 = 1.5
> # generate two samples independently
> x1 = rnorm(n, mu1, sigma1)
> x2 = rnorm(n, mu2, sigma2)
> # first, let's do a two-sided t test
> # assume equal variance for the two populations
> mu_diff = 0.5
> # H0: the population means' difference is equal to mu_diff
> # H1: the population means' difference is not equal to mu_diff
> t.test(x1, x2, alternative = "two.sided", mu = mu_diff, paired
    = FALSE, var.equal = TRUE)

  Two Sample t-test

data:  x1 and x2
t = 1.2286, df = 98, p-value = 0.2222
alternative hypothesis: true difference in means is not equal to
    0.5
95 percent confidence interval:
 0.3072577 1.3192945
sample estimates:
mean of x mean of y
 1.964328  1.151052

> # we accept the null hypothesis since p-value>alpha
>
> # now, we don't assume the equal variance
> t.test(x1, x2, alternative = "two.sided", mu = mu_diff, paired
    = FALSE, var.equal = FALSE)

  Welch Two Sample t-test

data:  x1 and x2
t = 1.2286, df = 94.78, p-value = 0.2223
alternative hypothesis: true difference in means is not equal to
    0.5
95 percent confidence interval:
 0.3070428 1.3195094
sample estimates:
mean of x mean of y
 1.964328  1.151052

> # there is no big change for p-value, we also accept the null
   hypothesis since p-value>alpha
>
> # let's try a one sided test without equal variance
> # H0: the population means' difference is equal to mu_diff
> # H1: the population means' difference larger than mu_diff
> t.test(x1, x2, alternative = "less", mu = mu_diff, paired =
   FALSE, var.equal = FALSE)

  Welch Two Sample t-test

data:  x1 and x2
t = 1.2286, df = 94.78, p-value = 0.8889
alternative hypothesis: true difference in means is less than
   0.5
95 percent confidence interval:
     -Inf 1.236837
sample estimates:
mean of x mean of y
 1.964328  1.151052

> # we accept the null hypothesis since p-value>alpha
```

There are many other important hypothesis tests, such as the chi-squared test[5], likelihood-ratio test[6].

[5]https://en.wikipedia.org/wiki/Chi-squared_test
[6]https://en.wikipedia.org/wiki/Likelihood-ratio_test

4.7 BASICS OF LINEAR REGRESSION

There are numerous books available on the theory of linear regression, for example, [9] and [10]. So what is the purpose in writing a few sections about these models in another book on data science? Many audiences would be interested in how to implement their own regression models rather than using the off-the-shelf software packages. By the end of this chapter, we will build up our own linear regression models in R/Python. We will also see how to reuse some functions in different regressions, such as linear regression, and those regressions with L2 penalties.

We start from the matrix form of linear regression.

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}, \tag{4.7}$$

where the column vector $\boldsymbol{y}$ contains n observations on the dependent variable, $\boldsymbol{X}$ is a $n \times (p+1)$ matrix ($n > p$) of independent variables with constant vector $\mathbf{1}$ in the first column, $\boldsymbol{\beta}$ is a column vector of unknown population parameters to estimate based on the data, and $\boldsymbol{\epsilon}$ is the error term (or noise). For the sake of illustration, (4.7) can be extended as in

$$\begin{bmatrix} \boldsymbol{y}_1 \\ \boldsymbol{y}_2 \\ \vdots \\ \boldsymbol{y}_n \end{bmatrix} = \begin{bmatrix} 1 & \boldsymbol{X}_{11} & \boldsymbol{X}_{21} & \cdots & \boldsymbol{X}_{p1} \\ 1 & \boldsymbol{X}_{12} & \boldsymbol{X}_{22} & \cdots & \boldsymbol{X}_{p2} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \boldsymbol{X}_{1n} & \boldsymbol{X}_{2n} & \cdots & \boldsymbol{X}_{pn} \end{bmatrix} \begin{bmatrix} \boldsymbol{\beta}_0 \\ \boldsymbol{\beta}_1 \\ \vdots \\ \boldsymbol{\beta}_p \end{bmatrix} + \begin{bmatrix} \boldsymbol{\epsilon}_1 \\ \boldsymbol{\epsilon}_2 \\ \vdots \\ \boldsymbol{\epsilon}_n \end{bmatrix} \tag{4.8}$$

We apply the ordinary least squares (OLS)[7] approach to estimate the model parameter $\boldsymbol{\beta}$ since it requires fewer assumptions than other estimation methods such as maximum likelihood estimation[8]. Suppose the estimated model parameter is denoted as $\hat{\boldsymbol{\beta}}$; we define the residual vector of the system as $\boldsymbol{e} = \boldsymbol{y} - \boldsymbol{X}\hat{\boldsymbol{\beta}}$. The idea of OLS is to find $\hat{\boldsymbol{\beta}}$ which can minimize the sum of squared residuals (SSR), i.e.,

$$\min_{\hat{\boldsymbol{\beta}}} \quad \boldsymbol{e}'\boldsymbol{e} \tag{4.9}$$

Now the question is how to solve the optimization problem (4.8). First, let's expand the SSR.

$$\begin{aligned} \boldsymbol{e}'\boldsymbol{e} &= (\boldsymbol{y} - \boldsymbol{X}\hat{\boldsymbol{\beta}})'(\boldsymbol{y} - \boldsymbol{X}\hat{\boldsymbol{\beta}}) \\ &= \boldsymbol{y}'\boldsymbol{y} - 2\hat{\beta}'\boldsymbol{X}'\boldsymbol{y} + \hat{\beta}'\boldsymbol{X}'\boldsymbol{X}\hat{\boldsymbol{\beta}} \end{aligned} \tag{4.10}$$

The first- and second-order derivatives are calculated as follows

$$\begin{cases} \frac{\partial \boldsymbol{e}'\boldsymbol{e}}{\partial \hat{\boldsymbol{\beta}}} = -2\boldsymbol{X}'\boldsymbol{y} + 2\boldsymbol{X}'\boldsymbol{X}\hat{\boldsymbol{\beta}} \\ \frac{\partial^2 \boldsymbol{e}'\boldsymbol{e}}{\partial \hat{\boldsymbol{\beta}} \partial \hat{\boldsymbol{\beta}}'} = 2\boldsymbol{X}'\boldsymbol{X}. \end{cases} \tag{4.11}$$

[7]https://en.wikipedia.org/wiki/Ordinary_least_squares
[8]https://en.wikipedia.org/wiki/Maximum_likelihood_estimation

We see that the second-order derivative is positive semidefinite which implies the SSR in OLS is a convex function (see Section 3.1.4 in [3]) and for an unconstrained convex optimization problem, the necessary as well as sufficient condition for optimality is that the first-order derivative equals 0 (see Section 4.2.3 in [3]). Optimization of convex function is very important in machine learning. Actually, the parameter estimations of many machine learning models are convex optimization problems.

Based on the analysis above, the solution of (4.9) is given in (4.12).

$$\hat{\boldsymbol{\beta}} = (\boldsymbol{X}'\boldsymbol{X})^{-1}\boldsymbol{X}'\boldsymbol{y} \tag{4.12}$$

Now it seems we are ready to write our own linear regression model in R/Python. The solution in (4.12) involves matrix transportation, multiplication and inversion, all of which are supported in both R and Python. In Python, we can use the `numpy` module for the matrix operations.

However, in practice we don't solve linear regression with (4.12) directly. Why? Let's see an example with

$$x = \begin{bmatrix} 1e+6 & -1 \\ -1 & 1e-6 \end{bmatrix}.$$

R

```
> x=array(
    c(10^6,-1,-1,10^-6),
    c(2,2))
> # solve() would return the
      inverse matrix
> solve(t(x) %*% x)
Error in solve.default(
    t(x) %*% x) :
  system is computationally
      singular: reciprocal
      condition number =
      2.22044e-28
```

Python

```
>>> import numpy as np
>>> x=np.array([[1e6
    ,-1],[-1,1e-6]])
>>> np.linalg.inv(np.dot(x.
    transpose(),x))
array([[4.50359963e+03,
    4.50359963e+09],
        [4.50359963e+09,
            4.50359963e+15]])
```

The R code above throws an error because of the singularity of $\boldsymbol{X}'\boldsymbol{X}$. It's interesting that the corresponding Python code doesn't behave in the same way as R, which has been reported as an issue on github[9].

When the matrix $\boldsymbol{X}'\boldsymbol{X}$ is singular, how to solve the OLS problem? In this book, we will focus on the QR decomposition based solution. Singular value decomposition (SVD) can also be used to solve OLS, which will not be covered in this book.

In linear algebra, a QR decomposition[10] of matrix $\boldsymbol{X}$ would factorize $\boldsymbol{X}$ into a product, i.e., $\boldsymbol{X} = \boldsymbol{QR}$ where $\boldsymbol{Q}$ are orthogonal matrices and $\boldsymbol{R}$ is an upper triangular matrix. Since the matrix $\boldsymbol{Q}$ is orthogonal ($\boldsymbol{Q}' = \boldsymbol{Q}^{-1}$), we have

[9]https://github.com/numpy/numpy/issues/10471
[10]https://en.wikipedia.org/wiki/QR_decomposition

$$
\begin{aligned}
\hat{\beta} &= (\boldsymbol{X}'\boldsymbol{X})^{-1}\boldsymbol{X}'\boldsymbol{y} \\
&= (\boldsymbol{R}'\boldsymbol{Q}'\boldsymbol{Q}\boldsymbol{R})^{-1}\boldsymbol{R}'\boldsymbol{Q}'\boldsymbol{y} \\
&= (\boldsymbol{R}'\boldsymbol{R})^{-1}\boldsymbol{R}'\boldsymbol{Q}'\boldsymbol{y} \\
&= \boldsymbol{R}^{-1}\boldsymbol{Q}'\boldsymbol{y}
\end{aligned}
\tag{4.13}
$$

Now we are ready to write our simple R/Python functions for linear regression with the help of QR decomposition according to (4.13).

R chapter4/qr_solver.R

```r
qr_solver=function(x,y){
  qr.coef(qr(x),y)
}
```

Python chapter4/qr_solver.py

```python
import numpy as np

def qr_solver(x,y):
  q,r=np.linalg.qr(x)
  p = np.dot(q.T,y)
  return np.dot(np.linalg.
      inv(r),p)
```

Of course, we don't need to implement our own OLS solvers in a production environment; but if you do, still you may find some well-written and well-tested functions such as `np.linalg.lstsq` to save time and effort from doing it from scratch.

Ok, now we have finished the training part of a linear regression model in both R and Python. After we train a model we want to use it, i.e., to make predictions based on the model. For most machine learning models, training is much more complex than prediction (exceptions include Lazy-learning models such as KNN). Let's continue developing our own linear regression model by including the prediction function and enclose everything in an object.

R chapter4/linear_regression.R

```r

library(R6)
LR = R6Class(
  "LR",
  public = list(
    coef = NULL,
    initialize = function() {

    },
    fit = function(x, y) {
      self$qr_solver(cbind(1, x), y)
    },
    qr_solver = function(x, y) {
      self$coef = qr.coef(qr(x), y)
    },
    predict = function(new_x) {
      cbind(1, new_x) %*% self$coef
    }
  )
)
```

Python chapter4/linear_regression.py

```python
import numpy as np

class LR:
    def __init__(self):
        self.coef = None

    def qr_solver(self, x, y):
        q, r = np.linalg.qr(x)
        p = np.dot(q.T, y)
        return np.dot(np.linalg.inv(r), p)

    def fit(self, x, y):
        self.coef = self.qr_solver(np.hstack((np.ones((x.shape
            [0], 1)), x)), y)

    def predict(self, x):
        return np.dot(np.hstack((np.ones((x.shape[0], 1)), x)),
            self.coef)
```

Now let's try to use our linear regression model to solve a real regression problem with the Boston dataset[11], and check the results.

R

```r
> source('linear_regression.R')
>
> library(MASS) # load Boston data from this package
>
> lr = LR$new()
> # -i means excluding the ith column from the data.frame
> lr$fit(data.matrix(Boston[,-ncol(Boston)]),Boston$medv)
> print(lr$coef)
```

[11]https://www.cs.toronto.edu/ delve/data/boston/bostonDetail.html

```
                      crim           zn         indus
                                   chas
 3.645949e+01 -1.080114e-01  4.642046e-02  2.055863e-02
    2.686734e+00
          nox             rm           age           dis
                          rad
-1.776661e+01  3.809865e+00  6.922246e-04 -1.475567e+00
   3.060495e-01
          tax        ptratio         black         lstat
-1.233459e-02 -9.527472e-01  9.311683e-03 -5.247584e-01
> # let's make prediction on the same data
> pred=lr$predict(data.matrix(Boston[,-ncol(Boston)]))
> print(pred[1:5])
[1] 30.00384 25.02556 30.56760 28.60704 27.94352
>
> # compare it with the R built-in linear regression model
> rlm = lm(medv ~ ., data=Boston)
> print(rlm$coef)
  (Intercept)          crim            zn         indus
             chas
 3.645949e+01 -1.080114e-01  4.642046e-02  2.055863e-02
    2.686734e+00
          nox             rm           age           dis
                          rad
-1.776661e+01  3.809865e+00  6.922246e-04 -1.475567e+00
   3.060495e-01
          tax        ptratio         black         lstat
-1.233459e-02 -9.527472e-01  9.311683e-03 -5.247584e-01
> print(rlm$fitted[1:5])
       1        2        3        4        5
30.00384 25.02556 30.56760 28.60704 27.94352
```

Python

```
>>> from sklearn.datasets import load_boston
>>> from linear_regression import LR
>>> boston = load_boston()
>>> X, y = boston.data, boston.target
>>> # first, let's run our own linear regression
... lr = LR()
>>> lr.fit(X, y)
>>> print(lr.coef)
[ 3.64594884e+01 -1.08011358e-01  4.64204584e-02  2.05586264e-02
  2.68673382e+00 -1.77666112e+01  3.80986521e+00  6.92224640e-04
```

```
 -1.47556685e+00  3.06049479e-01 -1.23345939e-02 -9.52747232e-01
  9.31168327e-03 -5.24758378e-01]
>>> print(lr.predict(X)[:5])
[30.00384338 25.02556238 30.56759672 28.60703649 27.94352423]
>>>
>>> # now, use sklearn's linear regression model
... from sklearn.linear_model import LinearRegression
>>> reg = LinearRegression().fit(X, y)
>>> print(reg.coef_)
[-1.08011358e-01  4.64204584e-02  2.05586264e-02  2.68673382e+00
 -1.77666112e+01  3.80986521e+00  6.92224640e-04 -1.47556685e+00
  3.06049479e-01 -1.23345939e-02 -9.52747232e-01  9.31168327e-03
 -5.24758378e-01]
>>> print(reg.predict(X)[:5])
[30.00384338 25.02556238 30.56759672 28.60703649 27.94352423]
```

The results from our own linear regression models are almost identical to the results from `lm()` function or the `sklearn.linear_model` module, which means we have done a great job so far.

4.8 RIDGE REGRESSION

What is the problem of solving linear regression model specified by (4.9)? There is nothing wrong at all with that approach. But I would like to cite my favorite quote — "Essentially, all models are wrong, but some are useful"[12]. (4.9) provides one solution to model (4.7). There are some alternative solutions, such as lasso regression and ridge regression. In this section, let's focus on ridge regression.

What is ridge regression? Ridge regression doesn't change the model itself. Instead, it changes the way to estimate model parameters. By naive OLS, we minimize the SSR directly. In ridge regression, we change the objective function (which is commonly called loss function in machine learning models) by adding an additional penalty

$$\min_{\hat{\boldsymbol{\beta}}} \quad \boldsymbol{e}'\boldsymbol{e} + \lambda \boldsymbol{\beta}'\boldsymbol{\beta}. \tag{4.14}$$

The optimization problem (4.14) is still an unconstrained optimization problem and it is convex. It can also be formulated as a constrained convex optimization problem as

$$\begin{aligned} &\min_{\hat{\boldsymbol{\beta}}} \quad \boldsymbol{e}'\boldsymbol{e} \\ &subject \quad to \quad \boldsymbol{\beta}'\boldsymbol{\beta} \leq t. \end{aligned} \tag{4.15}$$

[12]https://en.wikipedia.org/wiki/All_models_are_wrong

The theory behind ridge regression can be found from The Elements of Statistical Learning [9]. Let's turn our attention to the implementation of ridge regression. The solution to (4.15) can be obtained in the same way as the solution to (4.9), i.e.,

$$\hat{\boldsymbol{\beta}} = (\boldsymbol{X}'\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}'\boldsymbol{y}. \tag{4.16}$$

Again, in practice we don't use (4.16) to implement ridge regression for the same reasons that we don't use (4.12) to solve linear regression without penalty.

Actually, we don't need new techniques to solve (4.14). Let's make some transformation on the objective function in (4.14):

$$\boldsymbol{e}'\boldsymbol{e} + \lambda\boldsymbol{\beta}'\boldsymbol{\beta} = \sum_{i=1}^{n}(\boldsymbol{y}_i - \boldsymbol{x}_i'\boldsymbol{\beta})^2 + \sum_{i=1}^{p}(0 - \sqrt{\lambda}\boldsymbol{\beta}_i)^2 \tag{4.17}$$

, where $\boldsymbol{x}_i = [1, \boldsymbol{X}_{1i}, \boldsymbol{X}_{2i}, ..., \boldsymbol{X}_{pi}]'$.

Let's define an augmented dataset:

$$\boldsymbol{X}_\lambda = \begin{bmatrix} 1 & \boldsymbol{X}_{11} & \boldsymbol{X}_{21} & \cdots & \boldsymbol{X}_{p1} \\ 1 & \boldsymbol{X}_{12} & \boldsymbol{X}_{22} & \cdots & \boldsymbol{X}_{p2} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \boldsymbol{X}_{1n} & \boldsymbol{X}_{2n} & \cdots & \boldsymbol{X}_{pn} \\ \sqrt{\lambda} & 0 & 0 & \cdots & 0 \\ 0 & \sqrt{\lambda} & 0 & \cdots & 0 \\ 0 & 0 & \sqrt{\lambda} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \sqrt{\lambda} \end{bmatrix}, \quad \boldsymbol{y}_\lambda = \begin{bmatrix} \boldsymbol{y}_1 \\ \boldsymbol{y}_2 \\ \vdots \\ \boldsymbol{y}_n \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \tag{4.18}$$

If we regress $\boldsymbol{y}_\lambda$ on $\boldsymbol{X}_\lambda$, the OLS solution is just what we are looking for. However, usually the penalty is not applied on the intercept. Thus, we modify $\boldsymbol{y}_\lambda$ and $\boldsymbol{X}_\lambda$ to

$$\boldsymbol{X}_\lambda = \begin{bmatrix} 1 & \boldsymbol{X}_{11} & \boldsymbol{X}_{21} & \cdots & \boldsymbol{X}_{p1} \\ 1 & \boldsymbol{X}_{12} & \boldsymbol{X}_{22} & \cdots & \boldsymbol{X}_{p2} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \boldsymbol{X}_{1n} & \boldsymbol{X}_{2n} & \cdots & \boldsymbol{X}_{pn} \\ 0 & \sqrt{\lambda} & 0 & \cdots & 0 \\ 0 & 0 & \sqrt{\lambda} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \sqrt{\lambda} \end{bmatrix}, \quad \boldsymbol{y}_\lambda = \begin{bmatrix} \boldsymbol{y}_1 \\ \boldsymbol{y}_2 \\ \vdots \\ \boldsymbol{y}_n \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \tag{4.19}$$

Now we are ready to implement our own ridge regression model based on the description above. It is also common to normalize[13] the independent variables before applying ridge regression to make different variables in the same order of magnitude.

R chapter4/linear_regression_ridge.R

[13]https://en.wikipedia.org/wiki/Normalization_(statistics)

```r
library(R6)
LR_Ridge = R6Class(
  "LR_Ridge",
  public = list(
    coef = NULL,
    mu = NULL,
    sd = NULL,
    lambda = NULL,
    initialize = function(lambda) {
      self$lambda = lambda
    },
    scale = function(x) {
      self$mu = apply(x, 2, mean)
      self$sd = apply(x, 2, function(e) {
        sqrt((length(e) - 1) / length(e)) * sd(e)
      })
    },
    transform = function(x) {
      return(t((t(x) - self$mu) / self$sd))
    },
    fit = function(x, y) {
      self$scale(x)
      x_transformed = self$transform(x)
      x_lambda = rbind(x_transformed, diag(rep(sqrt(self$lambda)
          , ncol(x))))
      y_lambda = c(y, rep(0, ncol(x)))
      self$qr_solver(cbind(c(rep(1, nrow(
        x
      )), rep(0, ncol(
        x
      ))), x_lambda), y_lambda)
    },
    qr_solver = function(x, y) {
      self$coef = qr.coef(qr(x), y)
    },
    predict = function(new_x) {
      new_x_transformed = self$transform(new_x)
      cbind(rep(1, nrow(new_x)), new_x_transformed) %*% self$
          coef
    }
  )
)
```

Python chapter4/linear_regression_ridge.py

```python
import numpy as np
from sklearn.preprocessing import StandardScaler


class LR_Ridge:
    def __init__(self, l):
        self.l = l
        self.coef = None
        self.scaler = StandardScaler()

    def qr_solver(self, x, y):
        q, r = np.linalg.qr(x)
        p = np.dot(q.T, y)
        return np.dot(np.linalg.inv(r), p)

    def fit(self, x, y):
        x_transformed = self.scaler.fit_transform(x)
        x_lambda = np.vstack(
            (x_transformed, np.diag([self.l**0.5]*x.shape[1])))
        x_lambda = np.hstack(
            (np.vstack((np.ones((x.shape[0], 1)), np.zeros((x.
                shape[1], 1)))), x_lambda))
        y_lambda = np.hstack((y, np.zeros((x.shape[1],))))
        self.coef = self.qr_solver(x_lambda, y_lambda)

    def predict(self, x):
        new_x_transformed = self.scaler.transform(x)
        new_x_transformed = np.hstack(
            (np.ones((x.shape[0],1)), new_x_transformed)
        )
        return np.dot(new_x_transformed, self.coef)
```

In R, we implement our own scaler; but in the Python implementation, we use `StandardScaler` from `sklearn.preprocessing` module, which is very handy. Please pay attention to how we calculate the standard deviation[14] in R - we are using the formula of population standard deviation rather than the formula of sample standard deviation. Actually it doesn't matter which formula we use to calculate the standard deviation. I made the choice to use population standard deviation formula in order to generate consistent result of the `StandardScaler` since `StandardScaler` uses the formula of population standard deviation.

The selection of best λ requires solving the OLS problem repeatedly with different

[14]https://en.wikipedia.org/wiki/Standard_deviation

values of λ, which implies the QR decomposition procedure would be called multiple times. Using SVD decomposition could be more efficient in terms of selection of best λ. But it won't be covered in the current version of this book.

We are ready to run our own ridge regression on the Boston dataset.

R

```
> source('linear_regression_ridge.R')
>
> library(MASS) # load Boston data from this package
>
> # let's try lmabda = 0.5
> ridge = LR_Ridge$new(0.5)
> ridge$fit(data.matrix(Boston[,-ncol(Boston)]),Boston$medv)
> print(ridge$coef)
                    crim          zn       indus        chas
                             nox
22.53280632 -0.92396151  1.07393055  0.12895159  0.68346136
    -2.04275750
         rm         age         dis         rad         tax
                ptratio
 2.67854971  0.01627328 -3.09063352  2.62636926 -2.04312573
     -2.05646414
      black       lstat
 0.84905910 -3.73711409
> # let's make prediction on the same data
> pred=ridge$predict(data.matrix(Boston[,-ncol(Boston)]))
> print(pred[1:5])
[1] 30.01652 25.02429 30.56839 28.61521 27.95385
```

Python

```
>>> from sklearn.datasets import load_boston
>>> from linear_regression_ridge import LR_Ridge
>>>
>>> boston = load_boston()
>>> X, y = boston.data, boston.target
>>> # first, let's run our own linear regression
...
>>> ridge = LR_Ridge(0.5)
>>> ridge.fit(X, y)
>>> print(ridge.coef)
[ 2.25328063e+01 -9.23961511e-01  1.07393055e+00  1.28951591e-01
  6.83461360e-01 -2.04275750e+00  2.67854971e+00  1.62732755e-02
 -3.09063352e+00  2.62636926e+00 -2.04312573e+00 -2.05646414e+00
```

```
  8.49059103e-01 -3.73711409e+00]
>>> print(ridge.predict(X)[:5])
[30.01652397 25.02429359 30.56839459 28.61520864 27.95385422]
```

It's exciting to see the outputs from R and Python are quite consistent.

Linear regression is not as simple as it seems. To learn more about it, I recommend reading [10].

CHAPTER 5

Optimization in Practice

We talk about mathematical optimization in this chapter. Mathematical optimization is the selection of a best element from some set of available alternatives. Why do we talk about optimization in this book on data science? Data science is used to help make better decisions, and so is optimization. Operations research, with optimization at its core, is a discipline that deals with the application of advanced analytical methods to make better decisions[1]. I always feel data science and operations research are greatly intertwined. The training of a variety of (supervised) machine learning models is to minimize the loss functions, which is essentially an optimization problem. In optimization, the loss functions are usually referred as objective functions.

Mathematical optimization is a very broad field and we ignore its theories in this book and only touch some of the simplest applications. Actually, in Chapter 4 we have seen one of its important applications in linear regression.

5.1 CONVEXITY

The importance of convexity cannot be overestimated. A convex optimization problem [3] has the following form

$$\begin{aligned} &\max_{\boldsymbol{x}} f_0(\boldsymbol{x}) \\ &subject \quad to \quad f_i(\boldsymbol{x}) \leq \boldsymbol{b}_i; i = 1, ..., m, \end{aligned} \tag{5.1}$$

where the vector $\boldsymbol{x} \in \mathbf{R}^n$ represents the decision variable; $f_i; i = 1, ..., m$ are convex functions ($\mathbf{R}^n \rightarrow \mathbf{R}$). A function f_i is convex if

$$f_i(\alpha \boldsymbol{x} + (1-\alpha)\boldsymbol{y}) \leq \alpha f_i(\boldsymbol{x}) + (1-\alpha) f_i(\boldsymbol{y}), \tag{5.2}$$

for all $\boldsymbol{x}$, $\boldsymbol{y} \in \mathbf{R}^n$, and all $\alpha \in \mathbf{R}$ with $0 \leq \alpha \leq 1$. (5.1) implies that a convex optimization problem requires both the objective function and the set of feasible solutions to be convex. Why do we care about the convexity? Because a convex optimization problem has a very nice property - a local minimum (which is minimal

[1] https://en.wikipedia.org/wiki/Operations_research

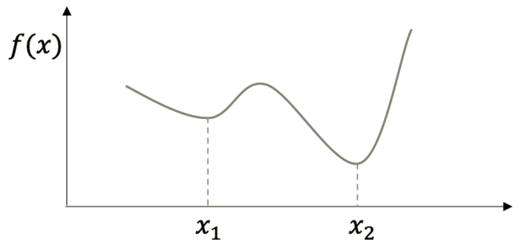


Figure 5.1: A non-convex function f. Its local optimum $f(x_2)$ is a global optimum, but the local optimum $f(x_1)$ is not

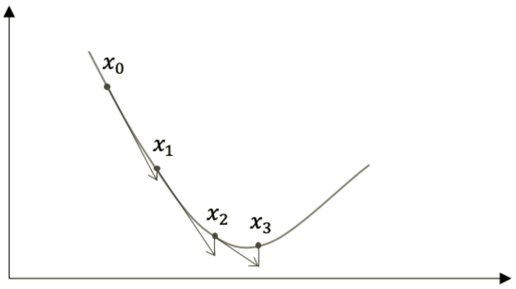


Figure 5.2: Apply gradient descent step by step for minimization

among a neighboring set of candidate solutions) is also a global minimum (which is the minimal solution among all feasible solutions). Thus, if we can find a local minimum we get the global minimum for convex optimization problems.

Many methods/tools we will introduce in the following sections won't throw an error if you feed them a non-convex objective function; but the solution returned by these numerical optimization methods is not necessary to be a global optimum. For example, the algorithm may get trapped in a local optimum.

5.2 GRADIENT DESCENT

What is the most commonly used optimization algorithm in machine learning? Probably the answer is gradient descent (and its variants). The gradient descent is also an iterative method, and in step i the update of $\boldsymbol{x}$ follows

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} - \gamma \nabla f(\boldsymbol{x}^{(k)}), \tag{5.3}$$

where f is the objective function and γ is called the step size or learning rate. Start from an initial value $\boldsymbol{x}_0$ and follow (5.3); we will have a monotonic sequence $f(\boldsymbol{x}^{(0)})$, $f(\boldsymbol{x}^{(1)})$, ..., $f(\boldsymbol{x}^{(n)})$ in the sense that $f(\boldsymbol{x}^{(k)}) >= f(\boldsymbol{x}^{(k+1)})$. When the problem is convex, $f(\boldsymbol{x}^{(i)})$ converges to the global minimum (see Figure 5.2).

Many machine learning models can be solved by the gradient descent algorithm, for example, the linear regression (with or without penalty) introduced in Chapter

4. Let's see how to use the vanilla (standard) gradient descent method in the linear regression that we introduced in Chapter 4.

According to the gradient derived in (4.11), the update for the parameters becomes:

$$\hat{\boldsymbol{\beta}}^{(k+1)} = \hat{\boldsymbol{\beta}}^{(k)} + 2\gamma \boldsymbol{X}'(\boldsymbol{y} - \boldsymbol{X}\hat{\boldsymbol{\beta}}^{(k)}). \tag{5.4}$$

R chapter5/linear_regression_gradient_descent.R

```r
library(R6)
LR_GD = R6Class(
  "LR_GD",
  public = list(
    coef = NULL,
    learning_rate = NULL,
    x = NULL,
    y = NULL,
    seed = NULL,
    initialize = function(x, y, learning_rate = 0.001, seed =
        42) {
      # we add 1s to x
      self$x = cbind(1, x)
      self$y = y
      self$seed = seed
      set.seed(self$seed)
      # we use a fixed learning rate here, but it can also be
         adaptive
      self$learning_rate = learning_rate
      # let's initialize the coef
      self$coef = runif(ncol(self$x))
    },
    fit = function(max_iteration = 1000) {
      for (i in 1:max_iteration) {
        self$update()
      }
    },
    gradient = function() {
      y_hat = self$predict(self$x)
      # we normalize the gradient by the sample size
      - 2 * (t(self$x) %*% (self$y - y_hat)) / nrow(self$x)
    },
    update = function() {
      self$coef = self$coef - self$learning_rate * self$gradient
         ()
    },
    predict = function(new_x) {
      new_x %*% self$coef
    }
  )
)
```

Accordingly, we have the Python implementation as follows.

Python chapter5/linear_regression_gradient_descent.py

```python
class LR_GD:
    """
    linear regression with vanilla gradient descent
    """

    def __init__(self, x, y, learning_rate=0.005, seed=42):
        # we put 1s in the first column of x
        self.x = np.hstack((np.ones((x.shape[0], 1)), x))
        self.y = y[:, None]
        self.seed = seed
        np.random.seed(self.seed)
        self.coef = np.random.uniform(size=(self.x.shape[1], 1))
        self.learning_rate = learning_rate

    def predict(self, new_x):
        # let's use @ for matrix multiplication
        return new_x @ self.coef

    def gradient(self):
        y_hat = self.predict(self.x)
        return -2.0*self.x.T @ (self.y-y_hat)/self.x.shape[0]

    def update(self):
        self.coef -= self.learning_rate*self.gradient()

    def fit(self, max_iteration=1000):
        for _ in range(max_iteration):
            self.update()
```

Let's use simulated dataset to test our algorithm.

R

```r
> source('gradient_descent.R
  ')
```

Python

```python
>>> from gradient_descent
   import LR_GD
```

```r
> n = 1000
> set.seed(42)
> x = cbind(rnorm(n),
    runif(n))
> b = 6 # intercept
> coef = c(2, 3)
> err = runif(n, 0, 0.2)
> y = x %*% coef + b + err
> lr_gd = LR_GD
    $new(x = x, y = y,
    learning_rate = 0.05)
> lr_gd$fit(1000)
> lr_gd$coef
          [,1]
[1,] 6.106233
[2,] 2.001042
[3,] 2.992593
```

```python
>>> import numpy as np
>>> n = 1000
>>> np.random.seed(42)
>>> x = np.column_stack((np.
    random.normal(size=n), np
    .random.uniform(size=n)))
>>> b = 6 # intercept
>>> coef = np.array([2, 3])
>>> err = np.random.uniform(
    size=n, low=0, high=0.2)
>>> y = x @ coef + b + err
>>> lr_gd = LR_GD(x=x, y=y,
    learning_rate=0.05)
>>> lr_gd.fit(1000)
>>> print(lr_gd.coef)
[[6.09887817]
 [2.00028507]
 [2.99996673]]
```

The results show that our implementation of gradient descent update works well on the simulated dataset for linear regression.

However, when the loss function is non-differentiable, the vanilla gradient descent algorithm (4.11) cannot be used. In Chapter 5, we added the L^2 norm (also referred to as Euclidean norm[2]) of $\boldsymbol{\beta}$ to the loss function to get the ridge regression. What if we change the L^2 norm to L^1 norm in (4.14)?

$$\min_{\hat{\boldsymbol{\beta}}} \quad \boldsymbol{e}'\boldsymbol{e} + \lambda \sum_{i=1}^{p} |\boldsymbol{\beta}_i|, \tag{5.5}$$

where $\lambda > 0$.

Solving the optimization problem specified in (5.5), we will get the Lasso solution of linear regression. Lasso is not a specific type of machine learning model. Actually, Lasso refers to least absolute shrinkage and selection operator. It is a method that performs both variable selection and regularization. What does variable selection mean? When we build up a machine learning model, we may collect as many data points (also called features, or independent variables) as possible. However, if only a subset of these data points are relevant for the task, we may need to select the subset explicitly for some reasons. First, it's possible to reduce the cost (time or money) to collect/process these irrelevant variables. Second, adding irrelevant variables into some machine learning models may impact the model performance. But why? Let's take linear regression as an example. Including an irrelevant variable into a linear regression model is usually referred to as model misspecification. (Of course, omitting a relevant variable also results in a misspecified model.) In theory, the estimator of

[2]https://en.wikipedia.org/wiki/Norm_(mathematics)

coefficients is still unbiased ($\hat{\theta}$ is an unbiased estimator of θ if $E(\hat{\theta}) = \theta$) when an irrelevant variable is included. But in practice, when the irrelevant variable has non-zero sample correlation coefficient with these relevant variables or the response variable $\boldsymbol{y}$, the estimate of coefficient may get changed. In addition, the estimator of coefficients is inefficient[3]. Thus, it's better to select the relevant variables for a linear regression.

The Lasso solution is very important in machine learning because of its sparsity, which results in the automated variable selection. As for why the Lasso solution is sparse, the theoretical explanation could be found from [9]. Let's focus on how to solve it. Apparently, gradient descent algorithm cannot be applied directly to solve (5.5) although it is still a convex optimization problem since the regularized loss function is non-differentiable because of $||$.

Proximal gradient is one of the algorithms that can be used to solve (5.5) is called proximal gradient descent [14]. Let's skip the theory of proximal methods and the tedious linear algebra, and jump to the solution directly. By proximal gradient descent, the update is given as follows in each iteration

$$\begin{aligned}\boldsymbol{\beta}^{(k+1)} &= S_{\lambda\gamma}(\boldsymbol{\beta}^{(i)} - 2\gamma\nabla(\boldsymbol{e}'\boldsymbol{e}))) \\ &= S_{\lambda\gamma}(\boldsymbol{\beta}^{(i)} + 2\gamma\boldsymbol{X}'(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}^{(i)})),\end{aligned} \tag{5.6}$$

where $S_\theta(\cdot)$ is called soft-thresholding operator given by

$$[S_\theta(\boldsymbol{z})]_i = \begin{cases} \boldsymbol{z}_i - \theta & \boldsymbol{z}_i > \theta; \\ 0 & |\boldsymbol{z}_i| \leq \theta; \\ \boldsymbol{z}_i + \theta & \boldsymbol{z}_i < -\theta. \end{cases} \tag{5.7}$$

Now we are ready to implement the Lasso solution of linear regression via (5.7). Similar to ridge regression, we don't want to put penalty on the intercept, and a natural estimate of the intercept is the mean of response, i.e., $\bar{\boldsymbol{y}}$. The learning rate γ in each update could be fixed or adapted (change over iterations). In the implementation below, we use the largest eigenvalue of $\boldsymbol{X}'\boldsymbol{X}$ as the fixed learning rate.

R chapter5/lasso.R

```r
library(R6)

# soft-thresholding operator
sto = function(z, theta){
  sign(z)*pmax(abs(z)-rep(theta,length(z)), 0.0)
}

Lasso = R6Class(
  "LassoLinearRegression",
  public = list(
```

[3]https://en.wikipedia.org/wiki/Efficient_estimator

```
    intercept = 0,
    beta = NULL,
    lambda = NULL,
    mu = NULL,
    sd = NULL,
    initialize = function(lambda) {
      self$lambda = lambda
    },
    scale = function(x) {
      self$mu = apply(x, 2, mean)
      self$sd = apply(x, 2, function(e) {
        sqrt((length(e)-1) / length(e))*sd(e)
      })
    },
    transform = function(x) t((t(x) - self$mu) / self$sd),
    fit = function(x, y, max_iter=100) {
      if (!is.matrix(x)) x = data.matrix(x)
      self$scale(x)
      x_transformed = self$transform(x)
      self$intercept = mean(y)
      y_centered = y - self$intercept
      gamma = 1/(eigen(t(x_transformed) %*% x_transformed, only.
          values=TRUE)$values[1])
      beta = rep(0, ncol(x))
      for (i in 1:max_iter){
        nabla = - t(x_transformed) %*% (y_centered -
            x_transformed %*% beta)
        z = beta - 2*gamma*nabla
        print(z)
        beta = sto(z, self$lambda*gamma)
      }
      self$beta = beta
    },
    predict = function(new_x) {
      if (!is.matrix(new_x)) new_x = data.matrix(new_x)
      self$transform(new_x) %*% self$beta + self$intercept
    }
  )
)
```

Python chapter5/lasso.py

```

import numpy as np
```

```python
from sklearn.preprocessing import StandardScaler

def sto(z, theta):
    return np.sign(z)*np.maximum(np.abs(z)-np.full(len(z),theta)
        , 0.0)

class Lasso:
    def __init__(self, l):
        self.l = l # l is lambda
        self.intercept = 0.0
        self.beta = None
        self.scaler = StandardScaler()

    def fit(self, x, y, max_iter=100):
        self.intercept = np.mean(y)
        y_centered = y-self.intercept
        x_transformed = self.scaler.fit_transform(x)
        gamma = 1.0/np.linalg.eig(x_transformed.T.dot(
            x_transformed))[0].max()
        beta = np.zeros(x_transformed.shape[1])
        for _ in range(max_iter):
            nabla = - np.dot(x_transformed.T, y_centered-np.dot(
                x_transformed, beta))
            z = beta - 2*gamma*nabla
            beta = sto(z, self.l*gamma)
            print(z)
        self.beta = beta

    def predict(self, new_x):
        new_x_transformed = self.scaler.transform(new_x)
        return np.dot(new_x_transformed, self.beta) + self.
            intercept
```

Now, let's see the application of our own Lasso solution of linear regression on the Boston house-prices dataset.

R

```r
> source('lasso.R')
> library(MASS)
> lr = Lasso$new(200)
> lr$fit(data.matrix(Boston[,-ncol(Boston)]), Boston$medv, 100)
> print(lr$beta)
               [,1]
crim    -0.34604592
```

```
zn        0.38537301
indus    -0.03155016
chas      0.61871441
nox      -1.08927715
rm        2.96073304
age       0.00000000
dis      -1.74602943
rad       0.02111245
tax       0.00000000
ptratio  -1.77697602
black     0.67323937
lstat    -3.71677103
```

Python

```
>>> from lasso import Lasso
>>> from sklearn.datasets import load_boston
>>> boston = load_boston()
>>> X, y = boston.data, boston.target
>>> lr = Lasso(200.0)
>>> lr.fit(X, y, max_iter=100)
>>> lr.beta
array([-0.34604592,  0.38537301, -0.03155016,  0.61871441,
    -1.08927715,
        2.96073304, -0.        , -1.74602943,  0.02111245, -0.
                  ,
       -1.77697602,  0.67323937, -3.71677103])
```

In the example above, we set the $\lambda = 200.0$ arbitrarily. The results show that the coefficients of `age` and `tax` are zero. In practice, the selection of λ is usually done by cross-validation which will be introduced later in this book. If you have used the off-the-shelf Lasso solvers in R/Python you may wonder why the λ used in this example is so big. One major reason is that in our implementation the first item in the loss function, i.e., $\boldsymbol{e}'\boldsymbol{e}$ is not scaled by the number of observations.

In practice, one challenge to apply Lasso regression is the selection of parameter λ. We will talk about that in Chapter 6 in more detail. The basic idea is to select the best λ to minimize the prediction error on the unseen data.

5.3 ROOT-FINDING

There is a close relation between optimization and root-finding. To find the roots of $f(x) = 0$, we may try to minimize $f^2(x)$ alternatively. Under specific conditions, to minimize $f(x)$ we may try to solve the root-finding problem of $f'(x) = 0$ (e.g., see linear regression in Chapter 5). Various root-finding methods are available in both R and Python.

Let's see an application of root-finding in finance.

Application - Internal rate of return

Internal rate of return(IRR) is a measure to estimate an investment's potential profitability. In fact, it is a discount rate which makes the net present value (NPV) of all future cashflows (positive for inflow and negative for outflow) equal to zero. Given the discount rate r, NPV is calculated as

$$\sum_{t=0}^{T} C_i/(1+r)^t, \tag{5.8}$$

where C_i denotes the cashflow and t denotes the duration from time 0. Thus, we can solve the IRR by finding the root of NPV.

Let's try to solve IRR in R/Python.

R chapter5/xirr.R

```r
NPV = function(cashflow_dates, cashflow_amounts, discount_rate){
  stopifnot(length(cashflow_dates)==length(cashflow_amounts))
  days_in_year = 365 # we assume there are 365 days in each year
  times= as.numeric(difftime(cashflow_dates, cashflow_dates[1],
     units="days"))/days_in_year
  sum(cashflow_amounts/(1.0+discount_rate)^times)
}
# we name this function as 'xirr' to follow the convention of MS
    Excel
xirr = function(cashflow_dates, cashflow_amounts){
  # we use uniroot function in R for root-finding
  uniroot(f=function(x) NPV(cashflow_dates,cashflow_amounts,x),
     interval= c(-1,100))$root
}
```

Python chapter5/xirr.py

```python
from datetime import datetime
from scipy.optimize import root
def NPV(cashflow_dates, cashflow_amounts, discount_rate):
  assert len(cashflow_dates)==len(cashflow_amounts), "
     inconsistent lengths of cashflows dates and amounts"
  days_in_year=365.0
  times = [(e-cashflow_dates[0]).days/days_in_year for e in
     cashflow_dates]
  return sum(cashflow_amounts[i]/(1.0+discount_rate)**e for i,e
     in enumerate(times))
def xirr(cashflow_dates, cashflow_amounts):
  # we use the scipy.optimize.root function to find the root
  return root(fun=lambda x:NPV(cashflow_dates,cashflow_amounts,x
     ),x0=[0.0]).x
```

R

```
> source('xirr.R')
> cashflow_dates = c(
   as.Date('2010-01-01'),
   as.Date('2010-06-01'),
   as.Date('2012-01-01'))
> cashflow_amounts =
   c(-1000, 200, 1000)
> xirr(cashflow_dates,
   cashflow_amounts)
[1] 0.1120706
```

Python

```
>>> from xirr import xirr
>>> from datetime
   import datetime
>>> cashflow_dates=[datetime
   (2010,1,1), datetime
   (2010,6,1), datetime
   (2012,1,1)]
>>> cashflow_amounts=[-1000,
    200, 1000]
>>> xirr(cashflow_dates,
   cashflow_amounts)
array([0.1120734])
```

Please note the R function `uniroot` works on one-dimensional functions only, while scipy.optimize.root function also works on multi-dimensional functions. `scipy.optimize.root` requires an initial guess for the solution, but `uniroot` does not. To find roots of multi-dimensional functions in R, the `optim` function introduced in the next section can also be used. Also, the package `rootSolve` could be useful. There are other useful features from both functions; for example, we can control the tolerance for termination.

5.4 GENERAL PURPOSE MINIMIZATION TOOLS IN R/PYTHON

Implementing our own algorithms from scratch could be fun and may help us better understand how the algorithms work, but they may not be always efficient and stable. There are established packages ready to use for many algorithms. For general purpose minimization problems, we can use the `optim` function in R and `scipy.optimize.minimize` function in Python as a black box. Various optimization algorithms have been implemented as the workhorse of these two functions. Some of these optimization algorithms (e.g., Quasi-Newton methods[4]) require the gradients, and some are derivate-free (e.g., Nelder–Mead method[5]). First, let's see a simple example.

[4]https://en.wikipedia.org/wiki/Quasi-Newton_method
[5]https://en.wikipedia.org/wiki/Nelder-Mead_method

Application - Maximum likelihood estimation of normal distribution

Maximum likelihood estimation (MLE)[6] is a very important topic in statistics. I will give a very brief introduction of what MLE is. To understand MLE, first we need to understand the likelihood function. Simply speaking, the likelihood function $L(\theta|x)$ is a function of the unknown parameters θ to describe the probability or odds of obtaining the observed data x (x is a realization of random variable X). For example, when the random variable X follows a continuous probability distribution with probability density function (pdf) f_θ, $L(\theta|x) = f_\theta(x)$. It's common to observe a random sample consisting of multiple independent observations rather than a single observation. In that case, the likelihood function is

$$L(\theta|x) = \prod_{i=1}^{n} f_\theta(x_i). \tag{5.9}$$

Given the observed data and a model on which the data are generated, we may minimize the corresponding likelihood function to estimate the model parameters. There are some nice properties of MLE, such as its consistency and efficiency. To better understand MLE and its properties, I recommend reading [4].

In practice, it's common to minimize the logarithm of the likelihood function, i.e., the log-likelihood function. When $X \sim \mathcal{N}(\mu, \sigma^2)$, the pdf of X is given as below

$$f(x|(\mu, \sigma^2)) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2}. \tag{5.10}$$

Taking the logarithm, the log-likelihood function is equal to

$$\mathcal{L}(\theta|x) = -\frac{n}{2}\log(2\pi) - n\log\sigma - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(x_i - \mu)^2. \tag{5.11}$$

Since the first item in (5.11) is a constant, we can simply set the log-likelihood function to

$$\mathcal{L}(\theta|x) = -n\log\sigma - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(x_i - \mu)^2. \tag{5.12}$$

It's worth noting (5.12) is convex. Let's implement the MLE for normal distribution in R/Python.

R chapter5/normal_mle.R

```r
log_lik = function(theta, x){
  mu = theta[1]
  sigma = theta[2]
```

[6]https://en.wikipedia.org/wiki/Maximum_likelihood_estimation

```
  # we return the negative log—likelihood since optim is for
      minimization
  —(—length(x)*log(sigma) — 0.5/sigma^2*sum((x—mu)^2))
}

normal_mle = function(x){
  # par=c(0,1) as the initial values
  # lower=c(—Inf, 1e—6) for the lower bounds
  optim(par=c(0, 1), fn=log_lik, x=x, lower=c(—Inf, 1e—6),
      method="L—BFGS—B")$par
}
```

Python chapter5/normal_mle.py

```
from scipy.optimize import minimize
from numpy.random import normal
from math import log

def log_lik(theta, x):
  mu = theta[0]
  sigma = theta[1]
  # return the negative log—likelihood for minimization
  return —(—len(x)*log(sigma) — 0.5/sigma**2*sum((e—mu)**2 for e
      in x))

def normal_mle(x):
  return minimize(fun=lambda e:log_lik(e,x),x0=[0,1],method='L—
      BFGS—B',bounds=((None,None),(1e—6,None))).x
```

There is no bound set for the mean parameter μ, while we need to set the lower bound for the standard deviation σ. We chose 'L-BFGS-B' as the optimization algorithm in this example, which requires the gradient of the objective function. When the gradient function is not provided, the gradient is estimated in the numeric approach. In general, providing the gradient function may speed up the work. 'L-BFGS-B' is a quasi-Newton method. Newton method requires the Hessian matrix for optimization, but the calculation of Hessian matrix may be expensive or even unavailable in some cases. The quasi-Newton methods approximate the Hessian matrix of the objective function instead of calculation. Now let's use these functions to estimate the parameters of normal distribution.

R

```
> source('normal_mle.R')
```

Python

```
>>> from numpy.random
    import seed, normal
```

```r
> n=100 # generate a random sample with size = 100
> set.seed(42)
> x=rnorm(n, mean=1.5, sd=0.8)
> normal_mle(x)
[1] 1.5260111 0.8289101
```

```python
>>> from normal_mle import normal_mle.R
>>> seed(42)
>>> n=100
>>> x=normal(1.5, 0.8, n)
>>> normal_mle(x)
array([1.41692178, 0.72289325])
```

Let's see another application of the general purpose minimization - logistic regression.

Application - Logistic regression

We have seen how linear regression works in Chapter 4. Logistic regression works similarly to linear regression. The major difference is that logistic regression is used in a classification problem. Classification means to identify to which of a set of categories a new observation belongs. Among classification problems, if each observation belongs to one of two disjoint categories, it is a binary classification problem. Binary classification has lots of real-world applications, such as spam email filter, loan default prediction, and sentiment analysis.
In logistic regression, the prediction output of an observation is the probability that the observation falls into a specific category. Let $\boldsymbol{X}$ represent a $n \times (p+1)$ matrix ($n > p$) of independent variables (predictor) with constant vector $\mathbf{1}$ in the first column, and $\boldsymbol{y}$ represent a column vector of the response variable (target). Please note $\boldsymbol{y}$ is a binary vector, i.e., $\boldsymbol{y}_i \in \{0, 1\}$. Formally, a logistic regression is specified as

$$Pr(\boldsymbol{y}_i = 1|\boldsymbol{x}_i, \boldsymbol{\beta}) = \frac{1}{1+e^{-\boldsymbol{x}_i'\boldsymbol{\beta}}}, \tag{5.13}$$

where $\boldsymbol{x}_i = [1, \boldsymbol{X}_{1i}, \boldsymbol{X}_{2i}, ..., \boldsymbol{X}_{pi}]$, and $\boldsymbol{\beta}$ denotes the parameter of the logistic model.
What does (5.13) mean? It implies the response variable $\boldsymbol{y}_i$ given $\boldsymbol{x}_i$ and $\boldsymbol{\beta}$ follows a Bernoulli distribution. More specifically, $\boldsymbol{y}_i \sim Bern(1/(1+\exp(-\boldsymbol{x}_i\boldsymbol{\beta})))$. Based on the assumption that the observations are independent, the log-likelihood function is

$$\begin{aligned}
\mathcal{L}(\beta|\boldsymbol{X}, \boldsymbol{y}) &= \log \prod_{i=1}^{n} \Big(Pr(\boldsymbol{y}_i = 1|\boldsymbol{x}_i, \boldsymbol{\beta})^{\boldsymbol{y}_i} Pr(\boldsymbol{y}_i = 0|\boldsymbol{x}_i, \boldsymbol{\beta})^{1-\boldsymbol{y}_i}\Big) \\
&= \sum_{i=1}^{n} \Big(\boldsymbol{y}_i \log Pr(\boldsymbol{y}_i = 1|\boldsymbol{x}_i, \boldsymbol{\beta}) + (1-\boldsymbol{y}_i) \log Pr(\boldsymbol{y}_i = 0|\boldsymbol{x}_i, \boldsymbol{\beta})\Big) \\
&= \sum_{i=1}^{n} \Big(\boldsymbol{y}_i \boldsymbol{x}_i \boldsymbol{\beta} - \log(1+e^{\boldsymbol{x}_i'\boldsymbol{\beta}})\Big).
\end{aligned} \tag{5.14}$$

Given the log-likelihood function, we can get the maximum likelihood estimate of logistic regression by minimizing (5.14) which is also convex. The minimization can be done similarly to linear regression via the iteratively re-weighted least square method (IRLS) [9]. However, in this section we will not use the IRLS method and instead let's use the `optim` function and `scipy.optimize.minimize` function in R and Python, respectively.

R chapter5/logistic_regression.R

```r
library(R6)

LogisticRegression = R6Class(
  "LogisticRegression",
  public = list(
    coef = NULL,
    initialize = function() {
    },
    sigmoid = function(x) {
      1.0/(1.0 + exp(-x))
    },
    log_lik = function(beta, x, y) {
      linear = x %*% beta
      ll = sum(linear * y) - sum(log(1 + exp(linear)))
      return(-ll) # return negative log-likelihood
    },
    fit = function(x, y) {
      if (!is.matrix(x)) x = data.matrix(x)
      self$coef = optim(par=rep(0, 1+ncol(x)), fn=self$log_lik,
          x=cbind(1, x), y=y, method="L-BFGS-B")$par
    },
    predict = function(new_x) {
      if (!is.matrix(new_x)) new_x = data.matrix(new_x)
      linear = cbind(1, new_x) %*% self$coef
      self$sigmoid(linear)
    }
  )
)
```

Python chapter5/logistic_regression.py

```python
import numpy as np
from scipy.optimize import minimize

class LogisticRegression:
    def __init__(self):
        self.coef = None

    # we use '_' as the prefix of the method name to indicate
        the method is private
    def _sigmoid(self, x):
        return 1.0/(1 + np.exp(-x))

    def _log_lik(self, beta, x, y):
        linear = np.dot(x, beta)
        ll = np.sum(linear.dot(y)) - np.sum(np.log(1.0+np.exp(
            linear)))
        return -ll

    def fit(self, x, y):
        self.coef = minimize(fun=self._log_lik, args=(np.insert(
            x, 0, 1.0, axis=1),y), method='L-BFGS-B', x0=np.
            zeros(1+x.shape[1])).x

    def predict(self, new_x):
        linear = np.insert(new_x, 0, 1.0, axis=1) # add 1 to
            new_x
        return self._sigmoid(linear)
```

Now let's see how to use our own logistic regression. We use the banknote dataset[7] in this example. In the banknote dataset, there are four different predictors extracted from the image of a banknote, which are used to predict if a banknote is genuine or forged.

R

```
> library(Metrics)
> banknote_data = read.csv('data_banknote_authentication.txt',
   header = FALSE)
> colnames(banknote_data) = c('variance', 'skewness', 'curtosis'
   , 'entropy', 'class')
>
> lr=LogisticRegression$new()
> lr$fit(banknote_data[,1:4], banknote_data$class)
> print(lr$coef)
[1]  7.3216635 -7.8590497 -4.1907866 -5.2872185 -0.6052346
> print(lr$predict(banknote_data[,1:4])[1:5]) # see the first 5
   predictions
[1] 4.071505e-19 6.741755e-22 2.187781e-10 1.604733e-16
[5] 4.579230e-01
```

Python

[7]https://archive.ics.uci.edu/ml/datasets/banknote+authentication

```
>>> from sklearn.metrics import log_loss
>>> from logistic_regression import LogisticRegression
>>> banknote_data=pd.read_csv('data_banknote_authentication.txt'
    ,header=None)
>>> from logistic_regression import LogisticRegression
>>> lr=LogisticRegression()
>>> lr.fit(banknote_data.iloc[:,0:4].values,banknote_data.iloc
    [:,4])
>>> print(lr.coef)
[ 7.32157876 -7.8591774  -4.19090781 -5.28734934 -0.60536929]
>>> lr.predict(banknote_data.iloc[:,0:4].values)[:5] # predict
    the probabilities of y=1
array([4.06674238e-19, 6.73409431e-22, 2.18663270e-10,
    1.60365564e-16,
       4.57910123e-01])
```

The above implementation of logistic regression works well. But it is only a pedagogical toy to illustrate what a logistic regression is and how it can be solved with general purpose optimization tool. In practice, there are fast and stable off-the-shelf software tools to choose. It's also worth noting most of the iterative optimization algorithms allow to specify the max number of iterations as well as the tolerance. Sometimes it's helpful to tune these parameters.

In practice, the penalized (L^1, L^2) versions of logistic regression are more commonly used than the vanilla logistic regression we introduced above.

5.5 LINEAR PROGRAMMING

In linear programming (LP)[8] both the objective function and the constraints are linear. LP can be expressed as follows,

$$\begin{aligned} &\max_{\boldsymbol{x}} \boldsymbol{c}'\boldsymbol{x} \\ &\textit{subject} \quad \textit{to} \quad \boldsymbol{Ax} \leq \boldsymbol{b}, \end{aligned} \tag{5.15}$$

where the vector $\boldsymbol{x}$ represents the decision variable; $\boldsymbol{A}$ is a matrix, $\boldsymbol{b}$ and $\boldsymbol{c}$ are vectors. We can do minimization instead of maximization in practice. Also, it's completely valid to have equality constraints for LP problems. All LP problems can be converted into the form of (5.15). For example, the equality constraint $\boldsymbol{Cx} = \boldsymbol{d}$ can be transformed to inequality constraints $\boldsymbol{Cx} \leq \boldsymbol{d}$ and $-\boldsymbol{Cx} \leq -\boldsymbol{d}$.

Every LP problem falls into three categories:

1. infeasible - no solution that satisfies all constraints exists;

[8]https://en.wikipedia.org/wiki/Linear_programming

Table 5.1: Available investment opportunities

investment	vintage	maturity	rate of return (annualized)
A	year 1	year 2	0.08
B	year 2	year 5	0.12
C	year 3	year 5	0.16
D	year 1	year 4	0.14

2. unbounded - for every feasible solution, there exists another feasible solution that improves the objective function;
3. having an optimal solution - a unique maximum (or minimum) value of the objective function exists.

When an LP problem has an optimal solution, there might be different sets of decision variables that lead to the same optimal value of the objective function. LP problems are also convex problems [3]. The algorithms to solve LP problems have been well-studied. The basic and popular algorithm is Dantzig's simplex method which was invented in 1947 [7]. More recent interior point methods have better worst-case complexity than the simplex method. In R/Python, we can find packages that implement both algorithms.

LP has been extensively used in business, economics, and some engineering problems. First, let's see a simple application of LP in regression analysis.

Application - Portfolio construction

Let's assume there are 4 different investment opportunities, A, B, C, and D, listed in Table ??. There is no intermediate payment between the vintage year and the maturity year. Payments received on maturity can be reinvested. We also assume the risk-free rate is 0. The goal is to construct a portfolio using these investment opportunities with $10,000 at year 1 so that the net asset value at year 5 is maximized.
Let x_1, x_2, x_3 and x_4 denote the amount to invest in each investment opportunity; and let x_0 denote the cash not invested in year 1. The problem can be formulated as

$$\begin{aligned}
\max_{\boldsymbol{x}} \quad & \boldsymbol{x}_0 + \boldsymbol{x}_1(1+0.08) - \boldsymbol{x}_2 - \boldsymbol{x}_3 + \boldsymbol{x}_2(1+0.12)^3 + \boldsymbol{x}_3(1+0.16)^2 + \boldsymbol{x}_4(1+0.14)^3 \\
& \textit{subject to} \quad \boldsymbol{x}_0 + \boldsymbol{x}_1 + \boldsymbol{x}_4 = 10000; \\
& \qquad \boldsymbol{x}_0 + \boldsymbol{x}_1(1+0.08) \geq \boldsymbol{x}_2 + \boldsymbol{x}_3; \\
& \qquad \boldsymbol{x}_i \geq 0; i = 0, ..., 4.
\end{aligned} \tag{5.16}$$

In Python, there are quite a few tools that can be used to solve LP, such as `ortools`[9], `scipy.optimize.linprog`[10], and `CVXPY`[11]. We will show the solution using `ortools` in Python and `lpSolve`[12] in R.

R chapter5/portfolio_construction.R

```r
library(lpSolve)

total = 10000.0
rate_of_return = c(0, 0.08, 0.12, 0.16, 0.14)
# define the objective function
f.obj = c(1, 1+rate_of_return[2], -1+(1+rate_of_return[3])^3,
    -1+(1+rate_of_return[4])^2, (1+rate_of_return[5])^3)
# define the constraints
f.con = t(array(c(1, 1, 0, 0, 1, 1, 1+rate_of_return[2], -1, -1,
     0), c(5, 2)))
# define the direction in the constraints
f.dir = c("=", ">=")
# define the right hand side of the constraints
f.rhs = c(total, 0)

solution = lp("max", f.obj, f.con, f.dir, f.rhs)
cat('x:', solution$solution, '\n')
cat('obj:', solution$objval, '\n')
```

Python chapter5/portfolio_construction.py

```python
from ortools.linear_solver import pywraplp

total = 10000.0
rate_of_return = [0.0, 0.08, 0.12, 0.16, 0.14]
solver=pywraplp.Solver('PortfolioConstruction',pywraplp.Solver.
    GLOP_LINEAR_PROGRAMMING)

# create the variables
x=[None]*5
for i in range(5):
    x[i]=solver.NumVar(0.0, solver.infinity(), 'x'+str(i))

# create the constraints
constraints = [None]*2
# set the equality constraint
```

[9]https://developers.google.com/optimization/
[10]https://scipy.org/
[11]https://www.cvxpy.org/
[12]https://cran.r-project.org/web/packages/lpSolve/lpSolve.pdf

```
constraints[0] = solver.Constraint(total, total)
constraints[0].SetCoefficient(x[0], 1.0)
constraints[0].SetCoefficient(x[1], 1.0)
constraints[0].SetCoefficient(x[4], 1.0)
# set the inequality constraint
constraints[1] = solver.Constraint(0.0, solver.infinity())
constraints[1].SetCoefficient(x[0], 1.0)
constraints[1].SetCoefficient(x[1], 1.0+rate_of_return[1])
constraints[1].SetCoefficient(x[2], -1.0)
constraints[1].SetCoefficient(x[3], -1.0)

# set objective function
objective = solver.Objective()
objective.SetCoefficient(x[0], 1.0)
objective.SetCoefficient(x[1], 1.0+rate_of_return[1])
objective.SetCoefficient(x[2], -1.0+(1+rate_of_return[2])**3)
objective.SetCoefficient(x[3], -1.0+(1+rate_of_return[3])**2)
objective.SetCoefficient(x[4], (1+rate_of_return[4])**3)
# we want to maximize the objective function
objective.SetMaximization()
status = solver.Solve()
if status == solver.OPTIMAL:
    sol = [e.solution_value() for e in x]
    print("x: {0}".format(sol))
    print("objective: {0}".format(objective.Value()))
```

Now let's run our code to get the results.

R

```
> source('portfolio_construction.R')
x: 0 10000 10800 0 0
obj: 15173.22
```

Python

```
chapter5 $python3.7 portfolio_construction.py
x: [0.0, 10000.0, 10800.0, 0.0, 0.0]
objective: 15173.222400000004
```

We see that the optimal portfolio allocation results in a net asset value of 15173.22 in year 5. You may notice that the Python solution is lengthy compared to the R solution. That is because the `ortools` interface in Python is in an OOP fashion and we add the constraint one by one. But for `lpSolve`, we utilize the compact matrix form (5.15) to specify an LP problem. In `ortools`, we specify the bounds of a decision variable during its declaration. In `lpSolve`, all decisions are non-negative by default.

LP also has applications in machine learning; for example, least absolute deviations (LAD) regression[13] can be solved by LP. LAD regression is just another type of linear regression. The major difference between LAD regression and the linear regression we introduced in Chapter 4 is that the loss function to minimize in LAD is the sum of the absolute values of the residuals rather than the SSR loss.

5.6 MISCELLANEOUS

5.6.1 Stochasticity

The update schedule of gradient descent is straightforward. But what to do if the dataset is too big to fit into memory? That is a valid question, especially in the era of big data. There is a naive and brilliant solution - we can take a random sample from all the observations and only evaluate the loss function on the random sample in each iteration. Actually, this variant of gradient descent algorithm is called stochastic gradient descent.

5.6.2 Coordinate descent

You may wonder that if we can randomly take a subset from all the observations, can we also just take a subset of the variables $\boldsymbol{x}$? Yes, and sometimes it is quite useful. For example, in coordinate descent, in each iteration we can update a single coordinate $\boldsymbol{x}^j$ rather than the entire $\boldsymbol{x}$. Coordinate descent has been widely applied for the Lasso solution of the generalized linear models [8].

5.6.3 Newton method

Newton method looks similar to gradient descent, but it requires the Hessian matrix of the objective function (see (5.17)). In theory, the convergence of Newton method is faster than the gradient descent algorithm. But not all differentiable objective functions have second-order derivatives; and the evaluation of Hessian matrix may also be computationally expensive.

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} - \gamma H(\boldsymbol{x}^{(k)})^{-1} \nabla f(\boldsymbol{x}^{(k)}) \tag{5.17}$$

We have seen the proximal gradient descent method for Lasso. Actually, the second-order derivatives can also be incorporated into the proximal gradient descent method, which leads to the proximal Newton method.

5.6.4 Constrained optimization

In general, constrained optimization has the following form.

$$\begin{aligned} & \max_{\boldsymbol{x}} f(\boldsymbol{x}) \\ subject \quad to \quad & g_i(\boldsymbol{x}) \leq \boldsymbol{b}_i; i = 1, ..., m, \\ & h_i(\boldsymbol{x}) = \boldsymbol{c}_j; j = 1, ..., k, \end{aligned} \tag{5.18}$$

[13]https://en.wikipedia.org/wiki/Least_absolute_deviations

where g_i is the inequality constraint and h_i is the equality constraint, both of which could be linear or nonlinear. A constrained optimization problem may or may not be convex. Although there are some tools existing in R/Python for constrained optimization, they may fail if you just throw the problem into the tools. It is better to understand the very basic theories for constrained optimization problems, such as the Lagrangian, and the Karush-Kuhn-Tucker (KKT) conditions [3].

5.6.5 Quadratic programming

Quadratic programming (QP)[14] is also commonly seen in machine learning. For example, the optimization problems in vanilla linear regression, ridge linear regression and Lasso solution of linear regression all fall into the QP category. To see why the Lasso solution of linear regression is also QP, let's convert the optimization problem from the unconstrained form (5.5) to the constrained form (5.19) similar to the ridge regression.

$$\begin{aligned} &\min_{\hat{\boldsymbol{\beta}}} \quad \boldsymbol{e}'\boldsymbol{e} \\ &subject \quad to \quad \lambda \sum_{i=1}^{p} |\boldsymbol{\beta}_i| \leq t. \end{aligned} \tag{5.19}$$

The || operator in the constraint in (5.19) can be eliminated by expanding the constraint to 2^p linear constraints.

$$\begin{aligned} &\min_{\hat{\boldsymbol{\beta}}} \quad \boldsymbol{e}'\boldsymbol{e} \\ &subject \quad to \\ &\lambda\boldsymbol{\beta}_1 + \lambda\boldsymbol{\beta}_2 + ... + \lambda\boldsymbol{\beta}_n \leq t \\ &-\lambda\boldsymbol{\beta}_1 + \lambda\boldsymbol{\beta}_2 + ... + \lambda\boldsymbol{\beta}_n \leq t \\ &... \\ &-\lambda\boldsymbol{\beta}_1 - \lambda\boldsymbol{\beta}_2 - ... - \lambda\boldsymbol{\beta}_n \leq t. \end{aligned} \tag{5.20}$$

It is clear the problem specified in (5.20) is a QP problem. The optimization of support vector machine (SVM) can also be formulated as a QP problem.

In R and Python, there are solvers[15] designed specifically for QP problems, which can be used as a black box.

5.6.6 Metaheuristic optimization

Some optimization algorithms are specific to problems. For example, simplex method is mostly used for linear programming problems. In contrast, metaheuristic optimization algorithms are not problem-specific. Actually, metaheuristic algorithms are

[14]https://en.wikipedia.org/wiki/Quadratic_programming

[15]https://cran.r-project.org/web/views/Optimization.html, https://cvxopt.org

strategies to guide the search process for the solutions. There is a wide variety of meta-heuristic algorithms in the literature, and one of my favorites is simulated annealing (SA) [16]. SA algorithm can be used for both continuous and discrete optimization problems (or hybrid, such as mixed integer programming[16]). The `optim` function in R has the SA algorithm implemented (set `method = 'SANN'`) for general purpose minimization with continuous variables. For discrete optimization problems, it usually requires customized implementation. The pseudocode of SA is given as follows.

```
• define the cooling schedule $T_k$, an energy function $E$, and initial solution $s_0$
• for $k = 1, ..., m$
    – pick a random neighbor $s_k$ based on $s_{k-1}$
    – calculate the energy change $\delta = E(s_k) - E(s_{k-1})$
    – calculate the acceptance probability $P = e^{-\delta/T_k}$
    – if $P < random(0, 1)$, $s_k = s_{k-1}$
• return $s_m$.
```

Let's see how we can apply SA for the famous traveling salesman problem.

Application - Traveling salesman problem

Given a list of cities, what is the shortest route that visits each city once and returns to the original city? This combinatorial optimization problem is the traveling salesman problem (TSP). When there are K cities, the number of feasible solutions is equal to $(K-1)!/2$. It is difficult to enumerate and evaluate all solutions for a large number of cities. SA algorithm fits this problem quite well.
Let's consider a problem with only 10 cities. To apply the SA algorithm, we need to determine a cooling schedule. For simplicity, we chose $T_k = T_0\alpha^k$. As for the energy function, the length of the route is a natural choice. For many problems, the challenge to apply SA is how to generate new solutions from the incumbent. And most of the time there are various ways to do that, but some may not be efficient. In this TSP problem, we adopt a simple strategy, i.e., randomly select two different cities from the current route and switch them. If the switch results in a shorter route we would accept the switch; if the new route is longer, we still could accept the switch with the acceptance probability.

The implementation of SA with this simple strategy is implemented in both R/Python as follows.

R chapter5/TSP.R

```r
library(R6)

TSP = R6Class(
```

[16]https://en.wikipedia.org/wiki/Integer_programming

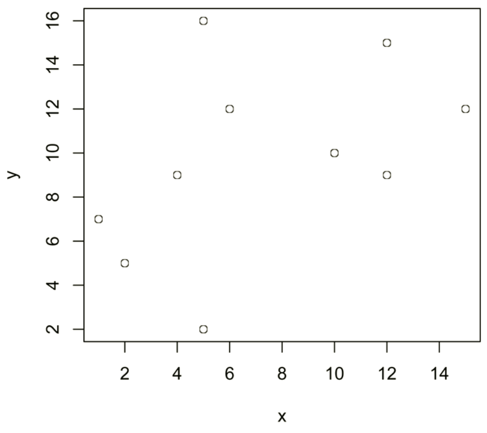


Figure 5.3: TSP with 10 cities represented by blue dots

```
  "TSP",
  public = list(
    cities = NULL,
    distance = NULL,
    initialize = function(cities, seed = 42) {
      set.seed(seed)
      self$cities = cities
      self$distance = as.matrix(dist(cities))
    },
    calculate_length = function(path) {
      l = 0.0
      for (i in 2:length(path)) l = l+self$distance[path[i-1],
          path[i]]
      l + self$distance[path[1],path[length(path)]]
    },
    accept = function(T_k, energy_old, energy_new) {
      delta = energy_new - energy_old
      p = exp(-delta/T_k)
      p>=runif(1)
    },
    solve = function(T_0, alpha, max_iter) {
      T_k = T_0
      # create the initial solution s0
      s = sample(nrow(self$distance), nrow(self$distance),
          replace=FALSE)
      length_old = self$calculate_length(s)
      for (i in 1:max_iter){
        T_k = T_k*alpha
        # we randomly exchange 2 cities in the route
        candidates = sample(nrow(self$distance), 2, replace=
          FALSE)
        temp = s[candidates[1]]
        s[candidates[1]] = s[candidates[2]]
        s[candidates[2]] = temp
        length_new = self$calculate_length(s)
        # check if we accept or reject the exchange
        if (self$accept(T_k, length_old, length_new)){
          # accept
          length_old = length_new
        }else{
          # reject
          s[candidates[2]] = s[candidates[1]]
          s[candidates[1]] = temp
        }
      }
      if (s[1]==1) return(list('s'=s,'length'=length_old))
      start = which(s==1)
      s_reordered = c(s[start:length(s)], s[1:start-1])
      list('s'=s_reordered,'length'=length_old)
    }
  )
)
```

Python chapter5/TSP.py

```python
import numpy as np
from scipy.spatial.distance import cdist

class TSP:
    def __init__(self, cities, seed=42):
        np.random.seed(seed)
        self.cities = cities
        # let's calculate the pairwise distance
        self.distance = cdist(cities, cities)

    def calculate_length(self, path):
        l = 0.0
        for i in range(1, len(path)):
            l += self.distance[path[i-1], path[i]]
        return l + self.distance[path[-1], path[0]]

    def accept(self, T_k, energy_old, energy_new):
        delta = energy_new - energy_old
        p = np.exp(-delta/T_k)
        return p >= np.random.uniform(low=0.0, high=1.0, size=1)
            [0]

    def solve(self, T_0, alpha, max_iter):
        T_k = T_0
        # let's create an initial solution s0
        s = np.random.permutation(len(self.cities))
        length_old = self.calculate_length(s)
        for i in range(max_iter):
            T_k *= alpha
            # we randomly choose 2 cities and exchange their
                positions in the route
            pos_1, pos_2 = np.random.choice(len(s), 2, replace=
                False)
            s[pos_1], s[pos_2] = s[pos_2], s[pos_1]
            length_new = self.calculate_length(s)
            # check if we want to accept the new solution or not
            if self.accept(T_k, length_old, length_new):
                # we accept the solution and update the old
                    energy
                length_old = length_new
            else:
                # we reject the solution and reverse the switch
                s[pos_1], s[pos_2] = s[pos_2], s[pos_1]
        # let's reorder the result
        if s[0] == 0:
            return s, length_old
        start = np.argwhere(s == 0)[0, 0]
        return np.hstack((s[start:], s[:start])), length_old
```

Let's run our TSP solver to see the results.

R

```
> cities = read.csv('
   chapter5/cities.csv')
> cities = cities[,c('x','y
   ')]
> tsp = TSP$new(cities)
> tsp$solve(2000,0.99,2000)
$s
 [1]  1  3  2 10  5  6  9  4
      7  8
```

Python

```
>>> from TSP import TSP
>>> cities = pd.read_csv('
    chapter5/cities.csv')
>>> cities = list(
    zip(cities.x, cities.y))
>>> tsp = TSP(cities)
>>> tsp.solve(2000, 0.99,
    2000)
(array([1, 2, 0, 7, 6, 3, 8,
```

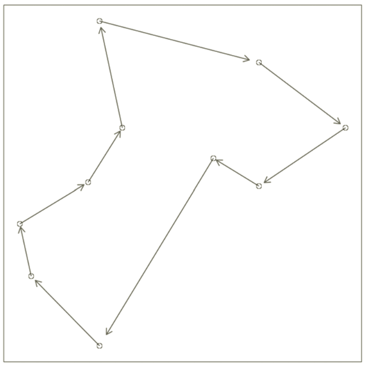


Figure 5.4: Route found by our TSP solver

```

$length
[1] 47.37255
```

```
      5, 4, 9]),
    47.372551972154646)
```

In the implementation above, we set the initial temperature to 2000, the cooling rate α to 0.99. After 2000 iterations, we get the same routes from both implementations. In practice, it's common to see metaheuristic algorithms get trapped in local optimums.

CHAPTER 6

Machine Learning - A gentle introduction

Machine LEARNING is such a huge topic that we will only touch on it a bit in this book. There are different learning paradigms.

Basically, supervised learning refers to the learning task in which we have input data and output data and the purpose is to learn a functional map from the input data to the output data. In contrast to supervised learning, there is unsupervised learning in which only the input data is available. In addition to supervised/unsupervised learning, there are also other learning paradigms, such as reinforcement learning[1] and transfer learning[2].

There is no strict distinction between different learning paradigms. For example, semi-supervised learning[3] falls between supervised learning and unsupervised learning, and supervised learning paradigms can also be found inside some modern reinforcement learning algorithms.

6.1 SUPERVISED LEARNING

In this book, we have talked about quite a few supervised learning models, such as linear regression and its variants, logistic regression. We will also spend some effort in tree-based models in this chapter. There are many other interesting supervised learning models that we don't cover in this book, such as support vector machine[4], linear discriminant analysis[5]. Almost all well-known supervised learning models' implementations can be found online and I recommend learning from reading the source code. As for the usage, there is an abundance of off-the-shelf options in both R and Python. Algorithm-wise, my favorite supervised learning models include linear models, gradient boosting trees, and (deep) neural network models. Linear models are simple with great interpretability; and it is rare if gradient boosting tree models or

[1]https://en.wikipedia.org/wiki/Reinforcement_learning
[2]https://en.wikipedia.org/wiki/Transfer_learning
[3]https://en.wikipedia.org/wiki/Semi-supervised_learning
[4]https://en.wikipedia.org/wiki/Support-vector_machine
[5]https://en.wikipedia.org/wiki/Linear_discriminant_analysis

deep neural network models cannot match the prediction accuracy of other models for a specific problem.

6.1.1 Population & random sample

A population is a complete set of elements of interest for a specific problem. It is usually defined by the researcher of the problem. A population can be either finite or infinite. For example, if we define the set of real numbers as our population, it is infinite. But if we are only interested in the integers between 1 and 100, then we get a finite population.

A random sample is a set of elements selected from a population. The size of a random sample could be larger than the population since an element can be taken multiple times. Why do we care about random samples? Because we are interested in the population from which the random sample is taken, and the random sample could help to make inference on the population.

In supervised learning, each element in the population has a set of attributes which are usually called features or covariates, as well as a label. For example, a bank may use the mortgage applicant's personal information (FICO score, years of employment, debt to income ratio, etc.) as the covariates, and the status of the mortgage (default, or paid off) as the label. A model can be used to predict the final status of the mortgage for a new applicant, and such kinds of models are classification models. When a mortgage is in default status, the applicant may have already made payments partially. Thus, a model to predict the amount of loss for the bank is also useful, and such kinds of models are regression models.

But why do we need supervised learning? If we know the labels of the entire population, nothing is needed to learn. All we need is a database table or a dictionary (HashMap) for lookup. The issue is that many problems in the real world don't allow us to have the labels of the entire population. And thus, we need to learn or infer based on the random sample collected from the unknown population.

6.1.2 Universal approximation

The universal approximation theorem says that a single hidden layer neural network can approximate any continuous functions ($\mathbf{R}^n \rightarrow \mathbf{R}$) with sufficient number of neurons under mild assumptions on the activation function (for example, the sigmoid activation function) [1]. There are also other universal approximators, such as the decision trees.

Not all machine learning models can approximate universally, for example, the linear regression without polynomial items. If we have the data from the entire population, we may fit the population with a universal approximator. But as we have discussed earlier, when the entire population is available there is no need to fit a machine learning model for prediction. But if only a random sample is available, is a universal approximator still the best choice? It depends.

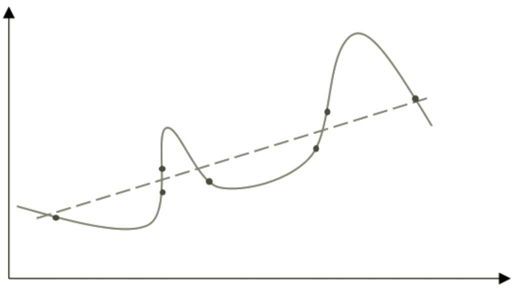


Figure 6.1: A random sample from a population follows a linear model (the dashed line) is overfit by the solid curve

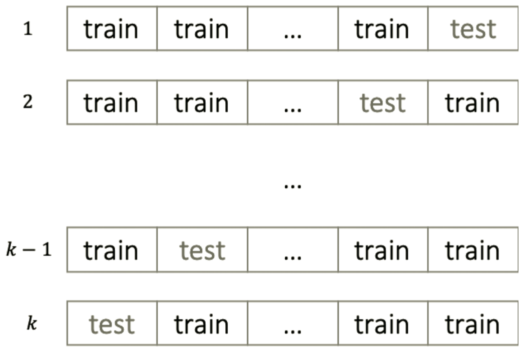


Figure 6.2: k fold cross-validation

6.1.3 Overfitting & cross-validation

One of the risks of fitting a random sample with a predictive model is overfitting (see Figure 6.1). Using universal approximators as the predictive model may even amplify such risk.

To mediate the risk of overfitting, we usually use cross-validation to assess how accurately a predictive model is able to predict for unseen data. The following steps specify how to perform a cross-validation.

- divide the training data into k partitions randomly
- for $i = 1, ..., k$
  - train a model using all partitions except partition i
  - record the prediction accuracy of the trained model on partition i
- calculate the average prediction accuracy.

Cross-validation can also be considered as a metaheuristic algorithm since it is not problem-specific because it doesn't matter what type of predictive models we use.

There are some ready-to-use tools in R and Python modules for cross-validation.

But for pedagogical purposes, let's do it step by step. We do the cross validation using the Lasso regression we built in a previous chapter on the Boston dataset.

R

```r
source('../chapter5/lasso.R')

library(caret)
library(MASS)
library(Metrics) # we use the rmse function from this package
k = 5

set.seed(42)
# if we set returnTrain = TRUE, we get the indices for train
    partition
test_indices = createFolds(Boston$medv, k = k, list = TRUE,
    returnTrain = FALSE)
scores = rep(NA, k)

for (i in 1:k){
  lr = Lasso$new(200)
  # we exclude the indices for test partition and train the
     model
  lr$fit(data.matrix(Boston[-test_indices[[i]], -ncol(Boston)]),
      Boston$medv[-test_indices[[i]]], 100)
  y_hat = lr$predict(data.matrix(Boston[test_indices[[i]], -ncol
     (Boston)]))
  scores[i] = rmse(Boston$medv[test_indices[[i]]], y_hat)
}
print(mean(scores))
```

Python

```python
import sys
sys.path.append("..")

from sklearn.metrics import mean_squared_error
from sklearn.datasets import load_boston
from sklearn.model_selection import KFold
from chapter5.lasso import Lasso


boston = load_boston()
X, y = boston.data, boston.target

```

```
# create the partitions with k=5
k = 5
kf = KFold(n_splits=k)
# create a placeholder for the rmse on each test partition
scores = []

for train_index, test_index in kf.split(X):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
    # let's train the model on the train partitions
    lr = Lasso(200.0)
    lr.fit(X_train, y_train, max_iter=100)
    # now test on the test partition
    y_hat = lr.predict(X_test)
    # we calculate the root of mean squared error (rmse)
    rmse = mean_squared_error(y_test, y_hat) ** 0.5
    scores.append(rmse)

# average rmse from 5-fold cross-validation
print(sum(scores)/k)
```

Run the code snippets; we have the 5-fold cross-validation accuracy as follows.

R

```
chapter6 $r -f cv.R
[1] 4.978324
```

Python

```
chapter6 $python3.7 cv.py
5.702339699398128
```

We add the line `sys.path.append("..")` in the Python code, otherwise it would throw an error because of the `import` mechanism[6].

6.1.4 Evaluation metrics

In the example above, we measure the root of mean squared error (RMSE) as the accuracy of the linear regression model. There are various metrics to evaluate the accuracy of predictive models.

- Metrics for regression

  For regression models, RMSE, mean absolute error (MAE)[7] and mean absolute percentage error (MAPE)[8] are some of the commonly-used evaluation metrics. You may have heard of the coefficient of determination (R^2 or adjusted R^2) in statistics. But from a predictive modelling perspective, R^2 is not a metric that

[6]https://docs.python.org/3/reference/import.html
[7]https://en.wikipedia.org/wiki/Mean_absolute_error
[8]https://en.wikipedia.org/wiki/Mean_absolute_percentage_error

Table 6.1: Some metrics for regression models

metric	**formula**
RMSE	$\sqrt{\frac{{}_{i=1}^{n}(\hat{y}_i - y_i)^2}{n}}$
MAE	$\frac{{}_{i=1}^{n}\lvert\hat{y}_i - y_i\rvert}{n}$
MAPE	$\frac{1}{n}\quad{}_{i=1}^{n}\left\lvert\frac{\hat{y}_i - y_i}{y_i}\right\rvert$

evaluates the predictive power of the model since its calculation is based on the training data. But what we are actually interested in is the model performance on the unseen data. In statistics, goodness of fit is a term to describe how good a model fits the observations, and R^2 is one of these measures for goodness of fit. In predictive modelling, we care more about the error of the model on the unseen data, which is called generalization error. But of course, it is possible to calculate the counterpart of R^2 on testing data.

- Metrics for classification

  The most intuitive metric for classification models is the accuracy, which is the percentage of corrected classified instances. To calculate the accuracy, we need to label each instance to classify. Recall the logistic regression we introduced in Chapter 5, the direct output of a logistic regression are probabilities rather than labels. In that case, we need to convert the probability output to the label for accuracy calculation. For example, let's consider a classification problem where the possible labels of each instance are 0, 1, 2 and 3. If the predictive probabilities of each label are 0.2, 0.25, 0.5, 0.05 for label 0, 1, 2 and 3, respectively, then the predictive label is 2 since its corresponding probability is the largest.

  But actually we don't always care about the labels of an instance. For example, a classification model for mortgage default built in a bank may only be used to calculate the expected monetary loss. Another example is the recommendation system that predicts the probabilities which are used for ranking of items. In that case, the model performance could be evaluated by `logloss`, `AUC`, etc., using the output probabilities directly. We have seen in Chapter 5 the loss function of logistic regression is the log-likelihood function.

  Actually, `logloss` is just the average evaluated log-likelihood on the testing data, and thus it can also be used for classification models with more than 2 classes (labels) because likelihood function is not restricted to Bernoulli distribution (extended Bernoulli distribution is called categorical distribution[9]). Another name for `logloss` is `cross−entropy loss`.

  In practice, `AUC` (Area Under the ROC Curve) is a very popular evaluation metric for binary-class classification problems. AUC is bounded between 0 and 1. A perfect model leads to an AUC equal to 1. If a model's predictions are

[9]https://en.wikipedia.org/wiki/Categorical_distribution

100% wrong, the resulting AUC is equal to 0. But if we know a binary-class classification model always results in 100% wrong predictions, we can instead use $1-\hat{y}$ as the corrected prediction and as a result we will get a perfect model and the AUC based on the corrected prediction becomes 1. Actually, a model using a completely random guess as the prediction leads to an AUC equal to 0.5. Thus, in practice the evaluated AUC is usually between 0.5 and 1.

There are also many other metrics, such as recalls, precisions, and F1 score[10].

The selection of evaluation metrics in predictive modelling is important but also subjective. Sometimes we may also need to define a customized evaluation metric.

Many evaluation metrics can be found from the R package `Metrics` and the Python module `sklearn.metrics`.

R

```
> set.seed(42)
> # regression metrics
> y = rnorm(n = 10,
  mean = 10, sd = 2)
> y
 [1] 12.741917  8.870604
    10.726257 11.265725
    10.808537  9.787751
 [7] 13.023044  9.810682
    14.036847  9.874572
> # we use random numbers as
    the predictions
> y_hat = rnorm(n = 10,
  mean = 10.5, sd = 2.2)
> y_hat
 [1] 13.370713 15.530620
    7.444506  9.886665
    10.206693 11.899091
 [7]  9.874644  4.655798
    5.130973 13.404249
>
> rmse(actual = y, predicted
    = y_hat)
[1] 4.364646
> mae(actual = y, predicted
   = y_hat)
[1] 3.540164
> mape(actual = y, predicted
    = y_hat)
[1] 0.3259014
```

Python

```
>>> import numpy as np
>>> from sklearn.metrics
    import mean_squared_error
    , mean_absolute_error,
    log_loss, roc_auc_score
>>> np.random.seed(42)
>>> # regression metrics
...
>>> y = np.random.normal(10,
     2, 10)
>>> y_hat = np.random.normal
    (10.5, 2.2, 10)
>>> mean_squared_error(y,
    y_hat) ** 0.5 #
    rmse
3.0668667318485165
>>> mean_absolute_error(y,
    y_hat) #
    mae
2.1355703394788237
>>> # let's define mape
    since it's not available
...
>>> def mape(y, y_hat):
    return np.mean(np.
    abs(y-y_hat)/y_hat)
...
>>> mape(y, y_hat)
0.292059554974094
>>> # classification metrics
>>> y = np.random.binomial
    (1, 0.25, 10)
```

[10]https://en.wikipedia.org/wiki/Precision_and_recall

```r
> # classification metrics
> y = rbinom(n = 10, size =
    1, prob=0.25)
> y
 [1] 0 0 0 1 0 1 1 0 1 0
> y_hat = runif(10, 0, 1)
> logLoss(y, y_hat)
[1] 0.4553994
> auc(y, y_hat)
[1] 0.8333333
```

```python
>>> y_hat = np.random.
    uniform(0, 1, 10)
>>> log_loss(y, y_hat)
0.47071363776285635
>>> roc_auc_score(y, y_hat)
    # auc
0.8095238095238095
```

6.1.5 Feature engineering & embedding

According to the explanation of feature engineering[11] on wikipedia, feature engineering is the process to use domain knowledge to create new features based on existing features. In reality, it is not rare to see feature engineering leads to better prediction accuracy. And sometimes I use feature engineering too. But I think feature engineering will be less and less useful in the future as the machine learning algorithms become more and more intelligent.

Let's consider three features x_1, x_2 and x_3 and assume the actual model is specified as $y = f(x_1, g(x_2, x_3))$. After all, y is still a function of x_1, x_2 and x_3, and we can write it as $y = h(x_1, x_2, x_3)$. Thus, even without creating the new feature $x_4 = g(x_2, x_3)$ explicitly, a universal approximator should be able to learn (i.e., approximate) the function h from the data ideally. This idea is also supported by the Kolmogorov–Arnold representation theorem[12] which says any continuous real-valued multivariate functions can be written as a finite composition of continuous functions of a single variable.

$$f(x_1, ..., x_m) = \sum_{q=0}^{2m} \Phi_q\Big(\sum_{p=1}^{m} \phi_{q,p}(x_p)\Big). \tag{6.1}$$

As of today, since the machine learning algorithms are not that intelligent, it is worth trying feature engineering especially when domain knowledge is available.

If you are familiar with dimension reduction, embedding can be considered as something similar. Dimension reduction aims at reducing the dimension of $\boldsymbol{X}$. It sounds interesting and promising if we can transform the high-dimensional dataset into a low-dimensional dataset and feed the dataset in a low dimension space to the machine learning model. However, this may not be a good idea in general. For example, in supervised learning it is not guaranteed that the low-dimensional predictors will keep all the information related to the response variable. Actually, many machine learning models are capable of handling the high-dimensional predictors directly.

Embedding transforms the features into a new space, which usually has a lower

[11]https://en.wikipedia.org/wiki/Feature_engineering

[12]https://en.wikipedia.org/wiki/Kolmogorov-Arnold_representation_theorem

dimension. But generally embedding is not done by the traditional dimension reduction techniques (for example, principal component analysis). In natural language processing, a word can be embedded into a vector space by word2vec [12] (or other techniques). When an instance is associated with an image, we may consider using the penultimate layer's output from a (deep) neural network-based image classification model to encode/embed the image into a space with lower dimension. It is also popular to use embedding layers for embedding.

6.1.6 Collinearity

Collinearity is one of the cliches in machine learning. For non-linear models, collinearity is usually not a problem. For linear models, I recommend reading this discussion[13] to see when it is not a problem.

6.1.7 Feature selection & parameter tuning

We have seen how the Lasso solutions of linear models can be used for feature selection in Chapter 6. What about non-linear models? There are some model-specific techniques for feature selection. Also, there is a metaheuristic approach to select features - cross-validation. Specifically, we can try different combinations of the features and use cross-validation to select the set of features which results in the best cross-validation evaluation metric. However, the major problem of this approach is its efficiency. When the number of features is too large, it is impossible to try all the different combinations with limited computational resources. Thus, it is better to use the model-specific feature selection techniques in practice.

To tune model parameters, such as the λ in Lasso, we can also use cross-validation. But again, the efficiency is our major concern.

Can we have feature selection and parameter tuning done automatically? Actually, automated machine learning[14] has been a hot research topic in both academia and industry. With the advance of machine learning theories, more intelligent algorithms will be available which require less and less work on feature engineering and parameter tuning (or neural network architecture searching).

6.2 GRADIENT BOOSTING MACHINE

In this section we will talk about one of the greatest supervised learning models - gradient boosting machine.

6.2.1 Decision tree

A decision tree consists of a bunch of nodes. In a decision tree there is a node with no parent nodes, which is called root node. The node without any children is called leaf node.

The length (number of nodes) of the longest path from the root node to a leaf

[13]https://statisticalhorizons.com/multicollinearity
[14]https://en.wikipedia.org/wiki/Automated_machine_learning

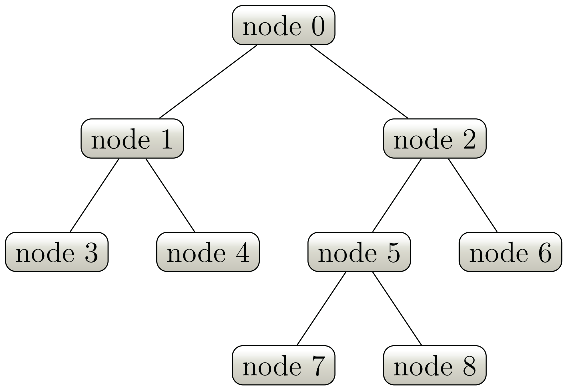


Figure 6.3: A single decision tree

node is called the depth of the tree. For example, the depth of the tree above is 4. Each leaf node has a label. In regression tasks, the label is a real number, and in classification tasks the label could be a real number which is used to get the class indirectly (for example, fed into a sigmoid function to get the probability), or an integer representing the predicted class directly.

Each node except the leaves in a decision tree is associated with a splitting rule. These splitting rules determine to which leaf an instance belongs. A rule is just a function taking a feature as input and returns true or false as output. For example, a rule on the root could be $x_1 < 0$ and if it is true, we go to the left node; otherwise we go to the right node. Once we arrive at a leaf, we can get the predicted value based on the label of the leaf.

To get a closer look, let's try to implement a binary tree structure for regression tasks in R/Python from scratch.

Let's implement the binary tree as a recursive data structure, which is composed partially of similar instances of the same data structure. More specifically, a binary tree can be decomposed into three components, i.e., its root node, the left subtree under the root, and the right subtree of the root. To define a binary (decision) tree, we only need to define these three components. And to define the left and right subtrees, this decomposition is applied recursively until the leaves.

Now we have the big picture on how to define a binary tree. However, to make the binary tree a decision tree, we also need to define the splitting rules. For simplicity, we assume there is no missing value in our data and all variables are numeric. Then a splitting rule of a node is composed of two components, i.e., the variable to split on, and the corresponding breakpoint for splitting.

There is one more component we need to define in a decision tree; that is, the prediction method which takes an instance as input and returns the prediction.

Now we are ready to define our binary decision tree.

R chapter6/tree.R

```
library(R6)
Tree = R6Class(
  "Tree",
  public = list(
    left = NULL,
    right = NULL,
    variable_id = NULL,
    break_point = NULL,
    val = NULL,
    initialize = function(left, right, variable_id, break_point,
        val) {
      self$left = left
      self$right = right
      self$variable_id = variable_id
      self$break_point = break_point
      self$val = val
    },
    is_leaf = function() {
      is.null(self$left) && is.null(self$right)
    },
    depth = function() {
      if (self$is_leaf()) {
        1
      } else if (is.null(self$left)) {
        1 + self$right$depth()
      } else if (is.null(self$right)) {
        1 + self$left$depth()
      } else{
        1 + max(self$left$depth(), self$right$depth())
      }
    },
    predict_single = function(x) {
      # if x is a vector
      if (self$is_leaf()) {
        self$val
      } else{
        if (x[self$variable_id] < self$break_point) {
          self$left$predict_single(x)
        } else{
          self$right$predict_single(x)
        }
      }
    },
    predict = function(x) {
      # if x is an array
      preds = rep(0.0, nrow(x))
      for (i in 1:nrow(x)) {
        preds[i] = self$predict_single(x[i, ])
      }
      preds
    },
    print = function() {
      # we can call print(tree), similar to the special method
         in Python
      cat("variable_id:", self$variable_id, "\n")
      cat("break at:", self$break_point, "\n")
      cat("is_leaf:", self$is_leaf(), "\n")
      cat("val:", self$val, "\n")
      cat("depth:", self$depth(), "\n")
      invisible(self)
    }
  )
)
```

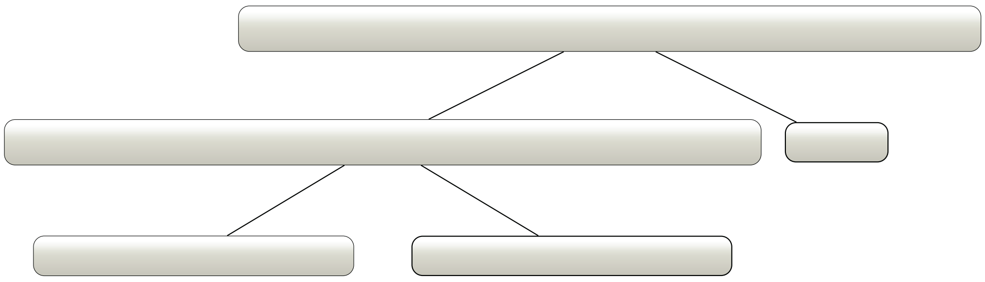
Python chapter6/tree.py

```python
class Tree:
    def __init__(self, left, right, variable_id, break_point,
        val):
        self.left = left
        self.right = right
        self.variable_id = variable_id
        self.break_point = break_point
        self.val = val

    @property
    def is_leaf(self):
        return self.left is None and self.right is None

    def _predict_single(self, x):
        if self.is_leaf:
            return self.val
        if x[self.variable_id] < self.break_point:
            return self.left._predict_single(x)
        else:
            return self.right._predict_single(x)

    def predict(self, x):
        return [self._predict_single(e) for e in x]

    @property
    def depth(self):
        if self.is_leaf:
            return 1
        elif self.left is None:
            return 1 + self.right.depth
        elif self.right is None:
            return 1 + self.left.depth
        return 1 + max(self.left.depth, self.right.depth)

    def __repr__(self):
        return "variable_id: {0}\nbreak_at: {1}\nval: {2}\
           nis_leaf: {3}\nheight: {4}".format(
            self.variable_id, self.break_point, self.val, self.
                is_leaf, self.depth)
```

You may have noticed the usage `@property` in our Python implementation. It is one of the built-in decorators in Python. We won't talk too much of decorators. Basically, adding this decorator makes the method `depth` behave like a property, in the sense that we can call `self.depth` instead of `self.depth()` to get the depth.

In the R implementation, the `invisible(self)` is returned in the print method which seems strange. It is an issue of `R6` class due to the `S3` dispatch mechanism which is not introduced in this book[15].

The above implementation doesn't involve the training or fitting of the decision tree. In this book, we won't talk about how to fit a traditional decision tree model due to its limited usage in the context of modern machine learning. Let's see how to use the decision tree structures we defined above by creating a pseudo decision tree illustrated below.



R

```
> source('tree.R')
> node_2 = Tree$new(NULL,
    NULL, NULL, NULL, 0.6)
> node_3 = Tree$new(NULL,
    NULL, NULL, NULL, 2.4)
```

Python

```
>>> from tree import Tree
>>> node_2 = Tree(None, None
    , None, None, 0.6)
```

[15]https://github.com/r-lib/R6/issues/140

```
> node_4 = Tree$new(NULL,
    NULL, NULL, NULL, 4.5)
> node_1 = Tree$new(node_3,
    node_4, 1, 10.5,
    NULL)
> node_0 = Tree$new(node_1,
    node_2, 2, 3.2,
    NULL)
> print(node_0)
variable_id: 2
break at: 3.2
is_leaf: FALSE
val:
depth: 3
> print(node_4)
variable_id:
break at:
is_leaf: TRUE
val: 4.5
depth: 1
> x = array(c(10, 0.5),
    c(1, 2))
> node_0$predict(x)
[1] 2.4
```

```
>>> node_3 = Tree(None, None
    , None, None, 2.4)
>>> node_4 = Tree(None, None
    , None, None, 4.5)
>>> node_1 = Tree(node_3,
    node_4, 0, 10.5, None)
>>> node_0 = Tree(node_1,
    node_2, 1, 3.2, None)
>>>
>>> print(node_0)
variable_id: 1
break_at: 3.2
val: None
is_leaf: False
depth: 3
>>> print(node_4)
variable_id: None
break_at: None
val: 4.5
is_leaf: True
depth: 1
>>> x = [[10, 0.5]]
>>> node_0.predict(x)
[2.4]
```

It's worth noting decision trees can approximate universally.

6.2.2 Tree growing in gradient boosting machine

What is a gradient boosting machine (GBM) (or gradient boosting regression)? Essentially, a GBM is just a forest of decision trees. If you have heard of random forest (RF), you may know that a random forest is also a bunch of trees. What is the difference between a GBM and RF?

Looking at the fitted trees from RF and GBM, there is no way to tell if the trees are fitted by an RF or a GBM. The major difference is how these trees are trained, rather than the trees themselves. A minor difference is how these trees are used for prediction. In many RF implementations, the prediction for an instance is the average prediction of each tree within the forest. If it is a classification task, there are two ways to get the final prediction — (a) predict the class with majority voting directly, i.e., the predicted class is the one with highest frequency among the predicted classes of all trees; (b) predict the probability based on the frequencies, for example, if among five trees there are three trees output class 1 then the predicted probability of class 1 is equal to 3/5. In many GBM implementations, the prediction (for regression tasks) is the sum of the predictions of individual trees.

GBM fits trees sequentially, but RF fits trees independently. The obvious advantage of fitting trees independently is that it can be done in parallel. But accuracy-wise, GBM usually performs better according to my limited experience.

We have seen the structure of a single decision tree in GBM. Now it's time to see how to get these trees fitted in GBM. Let's start from the first tree.

To grow a tree, we start from its root node. In GBM fitting, usually we predetermine a maximum depth d for each tree to grow. And the final tree's depth may be equal to or less than the maximum depth d. At a high level, the tree is grown in a recursive fashion. Specifically, first we attempt to split the current node and if the splitting improves the performance we grow the left subtree and the right subtree under the root node. When we grow the left subtree, its maximum depth is $d-1$, and the same applies to the right subtree. We can define a tree grow function for such purpose which takes a root node $Node_{root}$ (it is also a leaf) as input. The pseudo code of a tree grow function is illustrated below.

```
• if $d > 1$:
    – call split function on $Node_{root}$
    – if true is returned:
        ∗ call grow function on the empty $Node_{left}$ with $d-1$ maximum depth
        ∗ call grow function on the empty $Node_{right}$ with $d-1$ maximum depth
• return
```

In fact, the algorithm of tree growing is just a DFS algorithm.

To complete the pseudo algorithm above, we need to have a split function which takes a leaf node as input and returns a boolean value as output. If true is returned, we will do the splitting, i.e., to grow the left/right subtree. So now the challenge is how to define the split function, which requires the understanding of the theories behind GBM.

6.2.3 Optimization of GBM

Similar to other regression models we have seen so far, GBM with K trees has a loss function which is defined below.

$$\mathcal{L} = \mathop{}\limits_{i=1}^{n} (y_i - \hat{y}_i)^2, \tag{6.2}$$

where

$$\hat{y}_i = \mathop{}\limits_{t=1}^{K} f_t(\boldsymbol{x}_i). \tag{6.3}$$

f_t denotes the prediction of t^{th} tree in the forest. As we mentioned previously, the fitting is done sequentially. When we fit the t^{th} tree, all the previous $t-1$ trees are fixed. And the loss function for fitting the t^{th} tree is given below.

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} (y_i - \sum_{l=1}^{t-1} f_l(\boldsymbol{x}_i) - f_t(\boldsymbol{x}_i))^2 \tag{6.4}$$

In practice, a regularized loss function (6.5) is used instead of (6.3) to reduce overfitting.

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} (y_i - \sum_{l=1}^{t-1} f_l(\boldsymbol{x}_i) - f_t(\boldsymbol{x}_i))^2 + \Phi(f_t). \tag{6.5}$$

Let's follow the paper [6] and use the number of leaves as well as the L2 penalty of the values (also called weights) of the leaves for regularization. The loss function then becomes

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} (y_i - \sum_{l=1}^{t-1} f_l(\boldsymbol{x}_i) - f_t(\boldsymbol{x}_i))^2 + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} \omega_j^2, \tag{6.6}$$

where ω_j is the value associated with the j_{th} leaf of the current tree.

Again, we get an optimization problem, i.e., to minimize the loss function (6.6). The minimization problem can also be viewed as a quadratic programming problem. However, it seems different from the other optimization problems we have seen before, in the sense that the decision tree f_t is a non-parametric model. A model is non-parametric if the model structure is learned from the data rather than pre-determined.

A common approach used in GBM is the second-order approximation. By second-order approximation, the loss function becomes

$$\mathcal{L}^{(t)} \approx \sum_{i=1}^{n} (y_i - \sum_{l=1}^{t-1} f_l(\boldsymbol{x}_i))^2 + \sum_{i=1}^{n} (g_i f_t(\boldsymbol{x}_i) + \frac{1}{2} h_i f_t(\boldsymbol{x}_i)^2) + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} \omega_j^2, \tag{6.7}$$

where $g_i = 2(f_t(\boldsymbol{x}_i) + \sum_{l=1}^{t-1} f_l(\boldsymbol{x}_i) - y_i)$ and $h_i = 2$ are the first- and the second-order derivatives of the function $(y_i - \sum_{l=1}^{t-1} f_l(\boldsymbol{x}_i) - f_t(\boldsymbol{x}_i))^2$ with respect to the function $f_t(\boldsymbol{x}_i)$.

Let's implement the function to calculate g and h and put them into `util.py`.

Python chapter6/util.py

```python
import numpy as np

def gh_lm(actual, pred):
    '''
    gradient and hessian for linear regression loss
    '''
    return 2*(pred-actual), 2.0
```

Since the first item is a constant, let's ignore it.

$$\mathcal{L}^{(t)} \approx \underset{i=1}{\overset{n}{}} (g_i f_t(\boldsymbol{x}_i) + \frac{1}{2} h_i f_t(\boldsymbol{x}_i)^2) + \gamma T + \frac{1}{2}\lambda \underset{j=1}{\overset{T}{}} \omega_j^2. \tag{6.8}$$

Let's think of the prediction of an instance of the current tree. The training data, i.e., instances fall under the leaves of a tree. Thus, the prediction of an instance is the value ω associated with the corresponding leaf that the instance belongs to. Based on this fact, the loss function can be further rewritten as follows.

$$\mathcal{L}^{(t)} \approx \underset{j=1}{\overset{T}{}} (\omega_j \underset{i \in I_j}{} g_i + \frac{1}{2}\omega_j^2 (\underset{i \in I_j}{} h_i + \lambda)) + \gamma T. \tag{6.9}$$

When the structure of the tree is fixed the loss function (6.9) is a quadratic convex function of ω_j, and the optimal solution can be obtained by setting the derivative to zero.

$$\omega_j = -\frac{{}_{i \in I_j} g_i}{{}_{i \in I_j} h_i + \lambda} \tag{6.10}$$

Plugging (6.10) into the loss function results in the minimal loss of the current tree structure

$$-\frac{1}{2} \underset{j=1}{\overset{T}{}} \frac{({}_{i \in I_j} g_i)^2}{{}_{i \in I_j} h_i + \lambda} + \gamma T. \tag{6.11}$$

Now let's go back to the split function required in the tree grow function discussed previously. How to determine if we need a splitting on a leaf? (6.11) gives the solution — we can calculate the loss reduction by splitting which is given below

$$\frac{1}{2}\Big(\frac{({}_{i \in I_{left}} g_i)^2}{{}_{i \in I_{left}} h_i + \lambda} + \frac{({}_{i \in I_{right}} g_i)^2}{{}_{i \in I_{right}} h_i + \lambda} - \frac{({}_{i \in I} g_i)^2}{{}_{i \in I} h_i + \lambda} \Big) - \gamma. \tag{6.12}$$

If the loss reduction is positive, the split function returns true; otherwise it returns false.

So far, we have a few ingredients ready to implement our own GBM, which are listed below:

- the structure of a single decision tree in the forest, i.e., the `Tree` class defined;
- the node splitting mechanism, i.e., (6.12);
- the tree growing mechanism, i.e., the pseudo algorithm with the leaf value calculation (6.10).

However, there are a few additional items we need to go through before the implementation.

In Chapter 6, we have seen how stochasticity works in iterative optimization algorithms. The stochasticity technique is very important in the optimization of GBM.

More specifically, we apply the stochasticity both instance-wise and feature-wise. Instance-wise, when we fit a new tree, we randomly get a subsample from the training sample. And feature-wise, we randomly select a few features/variables when we fit a new tree. The stochasticity technique could help to reduce overfitting. The extent of the stochasticity can be controlled by arguments.

Like the gradient decent algorithm, in GBM there is also a learning rate parameter which scales the values of the leaves after a tree is fitted.

In practice, we may also not want to have too few instances under a leaf, to reduce potential overfitting. When there are two few instances under a leaf, we may just stop the splitting process.

Now we have almost all the ingredients to make a working GBM. Let's define the `split_node` function in the code snippet below.

Python chapter6/grow.py

```python
from tree import Tree
import numpy as np
import pdb


def split_node(f_in_tree, x_in_node, x_val_sorted,
    x_index_sorted, g_tilde, h_tilde, lam, gamma, min_instances)
    :
    '''
    f_in_tree: a list of booleans indicating which variable/
        feature is selected in the tree
    x_in_ndoe: a list of booleans indicating which instance is
        used in the tree
    x_val_sorted: a nested list, x_val_sorted[feature index] is
        a list of instance values
    x_index_sorted: a nested list, x_index_sorted[feature index]
         is a list of instance indexes
    g_tilde: first order derivative
    h_tilde: second order derivative
    lam: lambda for regularization
    gamma: gamma for regularization
    min_instances: the minimal number of instances under a leaf
    at the beginning we assume all instances are on the right
    '''
    if sum(x_in_node) < min_instances:
        return False, None, None, None, None, None, None
    best_break = 0.0
    best_feature, best_location = 0, 0
    ncol, nrow = len(f_in_tree), len(x_in_node)
    g, h = 0.0, 0.0
    for i, e in enumerate(x_in_node):
        if e:
            g += g_tilde[i]
            h += h_tilde[i]
    base_score = g*g/(h+lam)
    score_reduction = -np.inf
    best_w_left, best_w_right = None, None
    for k in range(ncol):
        if f_in_tree[k]:
            # if the feature is selected for this tree
            best_n_left_k = 0
            n_left_k = 0
            # feature is in the current tree
            g_left, h_left = 0.0, 0.0
            g_right, h_right = g-g_left, h-h_left
            # score reduction for current feature k
            score_reduction_k = -np.inf
            for i in range(nrow):
                # for each in sample, we try to split on it
                index = x_index_sorted[k][i]
                if x_in_node[index]:
                    n_left_k += 1
                    best_n_left_k += 1
                    g_left += g_tilde[index]
                    g_right = g-g_left
                    h_left += h_tilde[index]
                    h_right = h-h_left
                    # new score reduction
                    score_reduction_k_i = g_left*g_left/(h_left+lam) + \
                        (g_right*g_right)/(h_right+lam)-base_score
                    if score_reduction_k <= score_reduction_k_i:
                        best_n_left_k = n_left_k
                        best_break_k = x_val_sorted[k][i]
                        best_location_k = i
                        score_reduction_k = score_reduction_k_i
                        w_left_k = -g_left/(h_left+lam)
                        w_right_k = -g_right/(h_right+lam)

            # if the score reduction on feature k is a better candidate
            if score_reduction_k >= score_reduction:
                score_reduction = score_reduction_k
                best_feature = k
```

```python
                best_break = best_break_k
                best_location = best_location_k
                best_w_left = w_left_k
                best_w_right = w_right_k
    return 0.5*score_reduction >= gamma, best_feature, best_break, best_location, best_w_left, best_w_right, score_reduction


def grow_tree(current_tree, f_in_tree, x_in_node, max_depth, x_val_sorted, x_index_sorted, y, g_tilde, h_tilde, eta, lam, gamma, min_instances):
    '''
    current_tree: the current tree to grow, i.e., a node
    f_in_tree, x_in_node, x_val_sorted, x_index_sorted: see split_node function
    max_depth: maximinum depth to grow
    eta: learning rate
    y: the response variable
    '''
    nrow = len(y)
    if max_depth == 0:
        return
    # check if we need a split
    do_split, best_feature, best_break, best_location, w_left, w_right, _ = split_node(
        f_in_tree, x_in_node, x_val_sorted, x_index_sorted, g_tilde, h_tilde, lam, gamma, min_instances)

    if do_split:
        # update the value/weight with the learning rate eta
        w_left_scaled = w_left*eta
        w_right_scaled = w_right*eta
        current_tree.variable_id = best_feature
        current_tree.break_point = best_break
        current_tree.val = None

        # initialize the left subtree
        current_tree.left = Tree(None, None, None, None, w_left_scaled)
        # initialize the right subtree
        current_tree.right = Tree(None, None, None, None, w_right_scaled)
        # update if an instance is in left or right
        x_in_left_node = [False]*len(x_in_node)
        x_in_right_node = [False]*len(x_in_node)
        for i in range(nrow):
            index = x_index_sorted[best_feature][i]
            if x_in_node[index]:
                if i <= best_location:
                    x_in_left_node[index] = True
                else:
                    x_in_right_node[index] = True
        # recursively grow its left subtree
        grow_tree(current_tree.left, f_in_tree, x_in_left_node, max_depth-1,
                  x_val_sorted, x_index_sorted, y, g_tilde, h_tilde, eta, lam, gamma, min_instances)
        # recursively grow its right subtree
        grow_tree(current_tree.right, f_in_tree, x_in_right_node, max_depth-1,
                  x_val_sorted, x_index_sorted, y, g_tilde, h_tilde, eta, lam, gamma, min_instances)
    else:
        # current node is a leaf, so we update the value/weight of the leaf
        g, h = 0.0, 0.0
        for i, e in enumerate(x_in_node):
            if e:
                g += g_tilde[i]
                h += h_tilde[i]
        w_left_scaled = -g/(h+lam)*eta
        current_tree.val = w_left_scaled
```

And the implementation of the `GBM` class is given below.

Python chapter6/gbm.py

```python
from tree import Tree
from grow import grow_tree, split_node
from utils import gh_lm, rmse
import numpy as np


class GBM:
    def __init__(self, x_train, y_train, depth, eta, lam, gamma, sub_sample, sub_feature, min_instances=2):
        '''
        x_train, y_train: training data
        depth: maximum depth of each tree
        eta: learning rate
        lam and gamma: regularization parameters
        sub_sample: control the instance-wise stochasticity
        sub_feature: control the feature-wise stochasticity
        min_instances: control the mimimum number of instances
            under a leaf
        '''
        self.n = len(x_train)
        self.m = len(x_train[0])
        self.x_train = x_train
        self.y_train = y_train
        self.x_test, self.y_test = None, None
        self.depth = depth
        self.eta = eta
        self.lam = lam
        self.gamma = gamma
        self.sub_sample = sub_sample
        self.sub_feature = sub_feature
        self.min_instances = min_instances

        self.y_tilde = [0]*len(y_train)
        self.g_tilde, self.h_tilde = [0] * \
            len(self.y_tilde), [0]*len(self.y_tilde)
        for i in range(len(self.y_tilde)):
            self.g_tilde[i], self.h_tilde[i] = gh_lm(
                y_train[i], self.y_tilde[i])
        x_columns = x_train.transpose()
        self.nf = min(self.m, max(1, int(sub_feature*self.m)))
        self.x_val_sorted = np.sort(x_columns)
        self.x_index_sorted = np.argsort(x_columns)
        self.forest = []

    def set_test_data(self, x_test, y_test):
        self.x_test = x_test
        self.y_test = y_test

    def predict(self, x_new):
        y_hat = np.array([0.0]*len(x_new))
        for tree in self.forest:
            y_hat += np.array(tree.predict(x_new))
        return y_hat

    def fit(self, max_tree, seed=42):
        np.random.seed(seed)
        self.forest = []
        i = 0
        while i < max_tree:
            # let's fit tree i
            # instance-wise stochasticity
            x_in_node = np.random.choice([True, False], self.n,
               p=[
                                         self.sub_sample, 1-self
                                            .sub_sample])
            # feature-wise stochasticity
            f_in_tree_ = np.random.choice(
                range(self.m), self.nf, replace=False)
            f_in_tree = np.array([False]*self.m)
            for e in f_in_tree_:
                f_in_tree[e] = True
            del f_in_tree_
            # initialize the root of this tree
            root = Tree(None, None, None, None, None)
            # grow the tree from root
            grow_tree(root, f_in_tree, x_in_node, self.depth-1,
               self.x_val_sorted,
                      self.x_index_sorted, self.y_train, self.
                         g_tilde, self.h_tilde, self.eta, self.
                         lam, self.gamma, self.min_instances)
            if root is not None:
                i += 1
                self.forest.append(root)
            else:
                next
            for j in range(self.n):
                self.y_tilde[j] += self.forest[-1].
                   _predict_single(
                    self.x_train[j])
                self.g_tilde[j], self.h_tilde[j] = gh_lm(
                    self.y_train[j], self.y_tilde[j])
            if self.x_test is not None:
                # test on the testing instances
                y_hat = self.predict(self.x_test)
                print("iter: {0:>4}   rmse: {1:1.6f}".format(
                    i, rmse(self.y_test, y_hat)))
```

Now let's see the performance of our GBM implemented from scratch.

Python chapter6/test_gbm.py

```
from gbm import GBM
```

```python
import numpy as np
from sklearn import datasets
from sklearn.utils import shuffle


def get_boston_data(seed=42):
    boston = datasets.load_boston()
    X, y = shuffle(boston.data, boston.target, random_state=seed
        )
    X = X.astype(np.float32)
    offset = int(X.shape[0] * 0.8)
    X_train, y_train = X[:offset], y[:offset]
    X_test, y_test = X[offset:], y[offset:]
    return X_train, y_train, X_test, y_test

if __name__ == "__main__":
    X_train, y_train, X_test, y_test = get_boston_data(42)
    gbm = GBM(X_train, y_train, depth=6, eta=0.05, lam=1.0,
              gamma=1, sub_sample=0.5, sub_feature=0.7)
    gbm.set_test_data(X_test, y_test)
    gbm.fit(max_tree=200)
```

Running the code above, we have output as below.

Python

```
chapter6 $python3.7 test_gbm.py
iter:    1   rmse: 23.063019
iter:    2   rmse: 21.997972
iter:    3   rmse: 21.026602
iter:    4   rmse: 20.043397
iter:    5   rmse: 19.210746
...
iter:  196   rmse: 2.560747
iter:  197   rmse: 2.544847
iter:  198   rmse: 2.541102
iter:  199   rmse: 2.537366
iter:  200   rmse: 2.535143
```

We don't implement the model in R, but it is not difficult to do based on the Python implementation above.

GBM can be used with various loss functions, and the major difference is the implementation of the first-/second-order derivatives, i.e., g and h.

Regardless of the performance, there are two major missing features in our implementation: a) cross-validation and b) early stopping. We have talked about cross-validation, but what is early stopping? It is a very useful technique in GBM. Usually, the cross-validated loss decreases when we add new trees at the beginning and at a certain point, the loss may increase when more trees are fitted (due to overfitting). Thus, we may select the best number of trees based on the cross-validated loss. Specifically, stop the fitting process when the cross-validated loss doesn't decrease. In practice, we don't want to stop the fitting immediately when the cross-validated loss starts increasing. Instead, we specify a number of trees, e.g., 50, as a buffer, after which the fitting process should stop if cross-validated loss doesn't decrease.

Early stopping is also used in other machine learning models, for example, neural network. Ideally, we would like to have early stopping based on the cross-validated loss. But when the training process is time-consuming, it's fine to use the loss on a testing dateset[16].

The commonly used GBM packages include `XGBoost`[17], `LightGBM`[18] and `CatBoost` [19]. Let's see how to use `XGBoost` for the same regression task on the Boston dataset.

R chapter6/xgb.R

```r
library(xgboost)
library(MASS)
library(Metrics)
set.seed(42)
train_index = sample(nrow(Boston), floor(0.8 * nrow(Boston)),
    replace = FALSE)
Boston = data.matrix(Boston)
target_col = which(colnames(Boston) == 'medv')
X_train = Boston[train_index, -target_col]
y_train = Boston[train_index, target_col]
X_test = Boston[-train_index, -target_col]
y_test = Boston[-train_index, target_col]
# prepare the data for training and testing
dTrain = xgb.DMatrix(X_train, label = y_train)
dTest = xgb.DMatrix(X_test)
params = list(
  "booster" = "gbtree",
  "objective" = "reg:linear",
  "eta" = 0.1,
  "max_depth" = 5,
  "subsample" = 0.6,
  "colsample_bytree" = 0.8,
  "min_child_weight" = 2
)
```

[16]https://en.wikipedia.org/wiki/Early_stopping
[17]https://github.com/dmlc/xgboost
[18]https://github.com/microsoft/LightGBM
[19]https://github.com/catboost/catboost

```
# run the cross-validation
hist = xgb.cv(
  params = params,
  data = dTrain,
  nrounds = 500,
  early_stopping_rounds = 50,
  metrics = 'rmse',
  nfold = 5,
  verbose = FALSE
)
# since we have the best number of trees from cv, let's train
    the model with this number of trees
model = xgb.train(params, nrounds = hist$best_iteration, data =
    dTrain)
pred = predict(model, dTest)

cat(
  "rmse on testing instances is",
  rmse(y_test, pred),
  "with",
  hist$best_iteration,
  "trees"
)
```

Python chapter6/xgb.py

```
import xgboost as xgb
from sklearn import datasets
from sklearn.utils import shuffle
from sklearn.metrics import mean_squared_error
import numpy as np

seed = 42
boston = datasets.load_boston()
X, y = shuffle(boston.data, boston.target, random_state=seed)
X = X.astype(np.float32)
offset = int(X.shape[0] * 0.8)
X_train, y_train = X[:offset], y[:offset]
X_test, y_test = X[offset:], y[offset:]

params = {'booster': 'gbtree', 'objective': 'reg:linear', '
    learning_rate': 0.1,
          'max_depth': 5, 'subsample': 0.6, 'colsample_bytree':
              0.8, 'min_child_weight': 2}
```

```

# prepare the data for training and testing
dtrain = xgb.DMatrix(data=X_train, label=y_train, missing=None)
dtest = xgb.DMatrix(X_test)

# run 5-fold cross-validation with maximum 1000 trees, and try
    to minimize the metric rmse
# early stopping 50 trees
hist = xgb.cv(params, dtrain=dtrain, nfold=5,
              metrics=['rmse'], num_boost_round=1000, maximize=
                 False, early_stopping_rounds=50)

# find the best number of trees from the cross-validation
    history
best_number_trees = hist['test-rmse-mean'].idxmin()

# since we have the best number of trees from cv, let's train
    the model with this number of trees
model = xgb.train(params, dtrain, num_boost_round=
    best_number_trees)
pred = model.predict(dtest)
print(
    f"rmse on testing instances is {mean_squared_error(pred,
        y_test)**0.5:.6f} with {best_number_trees} trees")
```

The parameters `subsample` and `colsample_bytree` control the stochasticity, within the range of $[0, 1]$. If we set these two parameters to 1, then all instances and all features are selected for fitting every tree.

The two code snippets illustrate a minimal workflow of fitting a GBM model. First, we conduct (hyper) parameter tuning (such as learning rate, number of trees, regularization parameters, stochasticity parameters) with cross-validation, and next we train the model with the tuned parameters.

Running the code snippets, we have the following results.

R

```
> source('xgb.R')
rmse on testing instances
    is 2.632298 with 83 trees
```

Python

```
chapter6 $python3.7 xgb.py
rmse on testing instances is
    2.736038 with 179 trees
```

In `XGBoost`, we could also use linear regression models as the booster (or base learner) instead of decision trees. However, when `'booster':'gblinear'` is used, the sum of the prediction from all boosters in the model is equivalent to the prediction from a single (combined) linear model. In that sense, what we get is just a Lasso solution of a linear regression model.

GBM can be used in different tasks, such as classification, ranking, survival analysis, etc. When we use GBM for predictive modelling, missing value imputation is not required, which is one big advantage over linear models. But in our own implementation we don't consider missing values for simplicity. In GBM, if a feature is categorical we could do label-encoding[20], i.e., mapping the feature to integers directly without creating dummy variables (such as one-hot encoding). Of course one-hot encoding[21] can also be used. But when there are too many new columns created by one-hot encoding, the probability that the original categorical feature is selected is higher than these numerical variables. In other words, we are assigning a prior weight to the categorical feature regarding the feature-wise stochasticity.

For quite a few real-world prediction problems, the monotonic constraints are desired. Monotonic constraints are either increasing or decreasing. The increasing constraint for feature x_k refers to the relationship that $f(x_1, ..., x_k, ..., x_m) \leq f(x_1, ..., x'_k, ..., x_m)$ if $x_k \leq x'_k$. For example, an increasing constraint for the number of bedrooms in a house price prediction model makes lots of sense. Using gradient boosting tree regression models we can enforce such monotonic constraints in a straightforward manner. Simply, after we get the best split for the current node, we may check if the monotonic constraint is violated by the split. The split won't be adopted if the constraint is broken.

6.3 UNSUPERVISED LEARNING

For many supervised learning tasks, we could formulate the problem as an optimization problem by writing down the loss function as a function of training input and output. In unsupervised learning problems there are no label/output. It is more difficult to formulate unsupervised learning problems in a unified approach. Still, some unsupervised learning problems can still be formulated as an optimization problem. Let's see a few examples briefly.

6.3.1 Principal component analysis (PCA)

PCA is a very popular technique in many engineering disciplines. As you may know, PCA is a technique for dimension reduction. More specifically, let $\boldsymbol{x}$ denote an $n * m$ matrix, and each row of $\boldsymbol{x}$ denoted as $\boldsymbol{x_i'}$ represents a point in $\mathbb{R}^m$. Sometimes the dimension m could be relatively large and we don't like that. In order to represent the data in a more compact way, we want to transform the raw data points into a new coordinate system in $\mathbb{R}^p$ where $p < m$. The $k^{th}; k = 1, ..., p$ coordinate of $\boldsymbol{x_i}$ in the new coordinate system can be written $\boldsymbol{w_k' x_i}$. However, there are infinite $\boldsymbol{w_k}$ for the transformation and we have to make a guidance for such transformation. The key to our guidance is to make the data points projected onto the first transformed

[20]https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html
[21]https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html

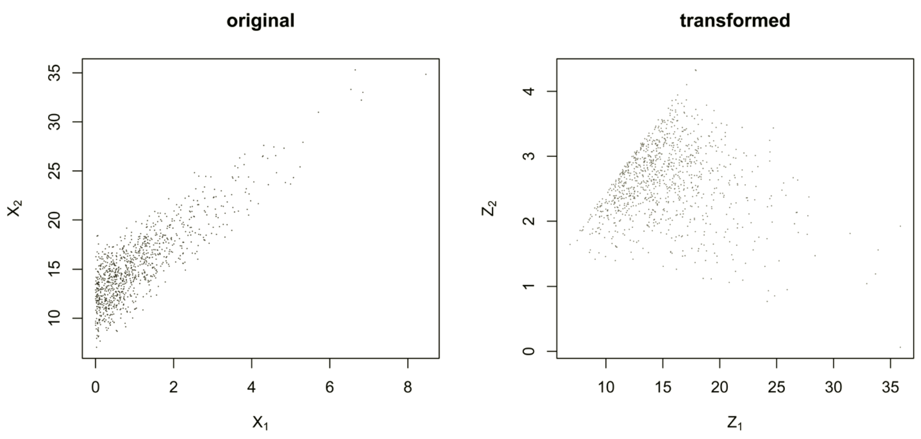


Figure 6.4: Original vs. transformed data points via PCA

coordinate have the largest variance, and $\boldsymbol{w_1'x_i}$ is called the first principal component. And we can find the remaining transformations and principal components iteratively.

So now we see how the PCA is formulated as an optimization problem. However, under the above setting, there are infinite solutions for $\boldsymbol{w_k}$. We usually add a constraint on $\boldsymbol{w_k}$ in PCA, i.e., $|\boldsymbol{w_k}| = 1$. The solution to this optimization problem is surprisingly elegant - the optimal $\boldsymbol{w_k}$ is the eigenvectors of the covariance matrix of $\boldsymbol{x}$. Now let's try to conduct a PCA with eigen decomposition (it can also be done with other decompositions).

R

```
> set.seed(42)
> n = 1000
> # we simulate some data points on a 2d plane
> x1 = rexp(n)
> x2 = x1*3 + rnorm(n, 12, 2)
> x = cbind(x1, x2)
> # total marginal variance
> sum(diag(cov(x)))
[1] 15.54203
> pca_vectors = eigen(cov(x))$vectors
> # find the projection on the new coordinate system
> z = x %*% pca_vectors
> # total marginal variance after transformation
> sum(diag(cov(z)))
[1] 15.54203
```

In fact, it does not matter if the raw data points are centered or not if the eigen decomposition is on the covariance matrix. If you prefer to decompose $\boldsymbol{x'x}$ directly, centering is a necessary step. Many times, we don't want to perform PCA in this way since there are a lot of functions/packages available in both R and Python.

6.3.2 Mixture model

In previous chapters we have seen how to fit a distribution to data. What if the data points actually come from multiple distributions, for example, a mixture of two Gaussian distributions. If we know from which distribution each observed data point comes from it is not difficult to estimate the parameters. But in some situations it is impossible to tell the actual distribution a point is sampled from. Usually, a mixture of multiple distributions can be estimated by the maximum likelihood method. We can derive the likelihood function of the observed data points and then we have an optimization problem.

Suppose we have a random size with sample size n, and each sample $x_i; i = 1, ..., n$ is from one of the K multivariate Gaussian distributions. The k^{th} distribution is denoted as $\mathcal{N}_k(\mu_k, \Sigma_k)$. We want to estimate the parameters in these Gaussian distributions.

As we discussed in Chapter 4, there are two commonly used approaches for distribution fitting, i.e., method of moments and maximum likelihood estimation. In this case, we use the maximum likelihood estimation because the likelihood function can be easily derived as below.

$$\mathcal{P} = \mathop{}\limits_{i=1}^{n} \mathop{}\limits_{k=1}^{K} \pi_k f(x_i|\mu_k, \Sigma_k), \tag{6.13}$$

where π_k represents the probability that a randomly selected data point belongs to distribution k. And thus, the log-likelihood function becomes:

$$\mathcal{L} = \mathop{}\limits_{i=1}^{n} log(\mathop{}\limits_{k=1}^{K} \pi_k f(x_i|\mu_k, \Sigma_k)). \tag{6.14}$$

Let's try to implement the above idea in R with the `optim` function.

R

```r
> library(mvtnorm)
> set.seed(42)
> n = 1000
> # 60% samples are from distribution 1 and the remainings are
    from distribution 2
> p = 0.6
> n1 = n * p
> n2 = n - n1
> mu1 = c(-1, -1)
> mu2 = c(1, 1)
> sigma1 = array(c(1, 0.5, 0.5, 1), c(2, 2))
> sigma2 = array(c(1, -0.2, -0.2, 1), c(2, 2))
> x1 = rmvnorm(n = n1, mean = mu1, sigma = sigma1)
> x2 = rmvnorm(n = n2, mean = mu2, sigma = sigma2)
> x = rbind(x1, x2)
> # let's permute x
> x_permuted = x[sample(n), ]
>
>
> log_lik = function(theta, x) {
+   # theta is a vector of length 9
+   # first let's reparametrize the parameters
+   mu1 = theta[1:2]
+   mu2 = theta[3:4]
+   sigma1 = array(c(theta[5], theta[6], theta[6], theta[5]), c
    (2, 2))
+   sigma2 = array(c(theta[7], theta[8], theta[8], theta[7]), c
    (2, 2))
+   pi_1 = theta[9]
+   pi_2 = 1 - pi_1
+   # we return the negative log-likelihood
+   - sum(log(pi_1 * dmvnorm(x, mu1, sigma1) + pi_2 * dmvnorm(x,
     mu2, sigma2)))
+ }
>
> gaussian_mixture_mle = function(x) {
+   # par as the initial values
+   optim(
+     par = c(0, 0, 0, 0, 1, 0, 1, 0, 0.5),
+     fn = log_lik,
+     x = x,
+     method = "L-BFGS-B",
+     control = list(trace = 1, maxit = 1000)
+   )$par
+ }
> res = gaussian_mixture_mle(x_permuted)
iter   10 value 3299.463920
final  value 3299.463917
converged
> mu1_hat = res[1:2]
> mu2_hat = res[3:4]
> sigma1_hat = array(c(res[5], res[6], res[6], res[5]), c(2, 2))
> sigma2_hat = array(c(res[7], res[8], res[8], res[7]), c(2, 2))
> pi = res[9]
> print(mu1_hat)
[1] -0.2221727 -0.2171368
> print(mu2_hat)
```

```
[1] −0.2221727 −0.2171368
> print(sigma1_hat)
          [,1]     [,2]
[1,] 1.986965 1.196158
[2,] 1.196158 1.986965
> print(sigma2_hat)
          [,1]     [,2]
[1,] 1.986965 1.196158
[2,] 1.196158 1.986965
> print(pi)
[1] 0.5
```

The estimates of parameters are not satisfactory, why? Remember we have emphasized the importance of convexity in Chapter 6. Actually, the log-likelihood function given in (6.14) is not convex. For non-convex problems, the `optim` function may not converge. In practice, EM algorithm[22] is frequently applied for mixture model.

The theory of EM algorithm is out of the scope of this book. Let's have a look at the implementation for this specific problem. Basically, there are two steps, i.e., E-step and M-step which run in an iterative fashion. In the t^{th} E-step, we update the membership probability $w_{i,k}$ that represents the probability that x_i belongs to distribution k, based on the current parameter estimates as follows.

$$w_{i,k}^{(t)} = \frac{\alpha_k^{(t)} f(x_i|\mu_k^{(t)}, \Sigma_k^{(t)})}{{}_{k=1}^{K} \alpha_k^{(t)} f(x_i|\mu_k^{(t)}, \Sigma_k^{(t)})}. \tag{6.15}$$

And in the t^{th} M-step, for each k we update the parameters as follows.

$$\begin{aligned}
\alpha_k^{(t+1)} &= \frac{{}_{i=1}^{n} w_{i,k}^{(t)}}{n} \\
\mu_k^{(t+1)} &= \frac{{}_{i=1}^{n} w_{i,k}^{(t)} x_i}{{}_{i=1}^{n} w_{i,k}^{(t)}} \\
\Sigma_k^{(t+1)} &= \frac{{}_{i=1}^{n} w_{i,k}^{(t)} (x_i - \mu_k^{(t)})(x_i - \mu_k^{(t)})'}{{}_{i=1}^{n} w_{i,k}^{(t)}}
\end{aligned} \tag{6.16}$$

The Python code below implements the above EM update schedule for a Gaussian mixture model.

Python chapter6/gmm.py

```
import numpy as np

class GaussianMixture:
```

[22]https://en.wikipedia.org/wiki/Expectation-maximization_algorithm

```python
    """
    X - n*m array
    K - the number of distributions/clusters
    seed - random seed for reproducibility
    """

    def __init__(self, X, K):
        self.X = X
        self.K = K
        self.n, self.p = X.shape
        self.mu_list = [np.random.uniform(-1, 1, self.p)
                        for _ in range(self.K)]
        self.sigma_list = [np.diag([1.0]*self.p) for _ in range(self.K)]
        self.alphas = np.ones(self.K)/self.K

    def E_step(self):
        # first, we update the membership weight for each data point, i.e., which distribution x_i belongs to
        # we compute the pdf for each data point and each distribution, stored in w(i,j)
        pdf_list = np.zeros((self.n, self.K))
        c = (2*np.pi)**(self.p/2)
        for i in range(self.K):
            x_centered = self.X - self.mu_list[i]
            sigma_inversed = np.linalg.inv(self.sigma_list[i])
            sigma_det = np.sqrt(np.linalg.det(self.sigma_list[i]))
            for j in range(self.n):
                pdf_list[j, i] = 1.0/(c*sigma_det)*
                    np.exp(-0.5 * x_centered[j, :][None, :] @ sigma_inversed @ x_centered[j, :][:, None])
        # we calculate the posterior probability
        posterior_prob = pdf_list*self.alphas
        # now let's update the memembership probability
        w = posterior_prob/np.sum(posterior_prob, axis=1)[:, None]
        return w

    def M_step(self, w):
        # we update the mu and sigma based on the updated weight from the E step
        for i in range(self.K):
            self.alphas[i] = np.mean(w[:, i])
            self.mu_list[i] = np.sum(
                self.X*w[:, i][:, None], axis=0)/np.sum(w[:, i])
            x_centered = self.X - self.mu_list[i]
            self.sigma_list[i] = x_centered.transpose() @ (
                x_centered*w[:, i][:, None])/np.sum(w[:, i])

    def fit(self, maxit=100, verbose=False):
        for _ in range(maxit):
            w = self.E_step()
            self.M_step(w)
            if verbose:
                print("mu: {}".format(self.mu_list))
                print("sigma: {}".format(self.sigma_list))

    def __repr__(self):
        return "mu: {}\n, sigma: {}\n, alpha: {}\n".format(self.
           mu_list, self.sigma_list, self.alphas)
```

For space-saving, we use a third-party package in R rather than implementing from scratch.

R

```
> library(mixtools)
> fit = mvnormalmixEM(x_permuted, maxit = 1000, k = 2)
number of iterations= 278
> summary(fit)
summary of mvnormalmixEM object:
          comp 1    comp 2
lambda  0.669674  0.330326
mu1    -0.893166 -0.859121
mu2     1.137988  1.084453
loglik at estimate:  -3211.291
> print(fit$sigma)
[[1]]
          [,1]      [,2]
[1,] 1.1086853 0.6670415
[2,] 0.6670415 1.2275707

[[2]]
           [,1]       [,2]
[1,]  0.9821523 -0.3748227
[2,] -0.3748227  1.0194300

# let's also infer the distribution for each data point
```

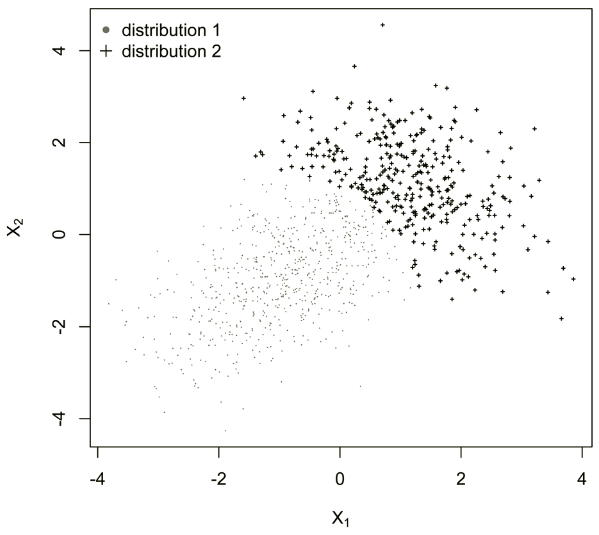


Figure 6.5: Data points clustered by a Gaussian mixture model

```
z = as.integer(
  fit$lambda[1] * dmvnorm(x_permuted, fit$mu[[1]], fit$sigma
      [[1]]) > fit$lambda[2] *
    dmvnorm(x_permuted, fit$mu[[2]], fit$sigma[[2]])
) + 1
```

In the above code, z_i represents the membership probability. We plot the data points with their clusters in Figure 6.5.

6.3.3 Clustering

Clustering is the task of grouping similar objects together. Actually in the example above we have used the Gaussian mixture model for clustering. There are many clustering algorithms and one of the most famous clustering algorithms might be K-means[23].

6.4 REINFORCEMENT LEARNING

Different from supervised/unsupervised learning, reinforcement learning (RL) is trying to learn a strategy which is used for decision-making. AlphaGo[24] and OpenAI Five[25] are two of the most famous use cases in the real world. There are many good

[23]https://en.wikipedia.org/wiki/K-means_clustering
[24]https://deepmind.com/research/case-studies/alphago-the-story-so-far
[25]https://openai.com/projects/five

resources for learning RL (for example, see [15]), and it is impossible to give an in-depth introduction on RL in this short section. But we could learn from a specific example for a first impression.

Application - A simple game

Let's play a hypothetical game, in which the player is asked to make a sequence of decisions. Each time, the decision is to choose one number from 1 and 2. If number 1 is chosen, the next decision-making time would be 1 time unit later and number 2 is chosen, the next decision-making time would be 2 time units later. No matter which number is chosen, the player always get $1 as the reward immediately. The game would end in 100 time units, and the goal is to collect as many rewards as possible.

The game seems very simple and the best strategy is to choose number 1 in each step. This problem cannot be solved by supervised or unsupervised learning techniques. And one obvious distinction in this problem is that we don't have any training data to learn from. This problem falls into the agent-environment interaction paradigm.

$$\begin{tikzcd}
\boxed{\text{agent}} \arrow[d, bend left] \\
\boxed{\text{environment}} \arrow[u, bend left]
\end{tikzcd}$$

In step t, the agent takes an action and the action $a^{(t)}$ in turn would have an effect on the environment. As a result, the state of the environment goes from $s^{(t)}$ to $s^{(t+1)}$ and returns a reward $r^{(t+1)}$ to the agent. Usually, the goal in RL is to pick up actions so that the cumulative reward $\;^{m}_{t=0} r^{(t+1)}$ is maximized. When $m \to \infty$, the cumulative reward may not be bounded. In that case, the future reward can be discounted by $\lambda \in [0, 1]$ and we want to maximize $\;^{\infty}_{t=0} \lambda^t r^{(t+1)}$. And in the following step, the agent would pick up the next action $a^{(t+1)}$ based on the information collected. In general, the environment state and reward at step t are random variables. If the probability distribution of the next state and reward only depends on the current environment state and the action picked up, the Markov property holds which results in a Markov decision process (MDP). A majority of RL studies focus on Markov decision process MDP.

With the advance of deep learning, deep learning-based RL has made significant progress recently. Many DRL algorithms have been developed, such as deep Q-networks (DQN) algorithm which we will discuss in more detail in the next section.

As we mentioned, usually there are no data to learn in RL. Both supervised and unsupervised learning need data to perform optimization on some objective functions. So does RL, in a different approach. RL can also be viewed as a simulation-based optimization approach, because RL runs simulations for the agent-environment interaction.

There are many RL algorithms available online. In the next section we will have a look at the DQN algorithm.

6.5 DEEP Q-NETWORKS

To introduce DQN we first define an action-value function $Q_\pi(s,a)$ and the optimal action-value function $Q_\pi^*(s,a)$. The action-value function is defined as

$$Q_\pi(s,a) = E[R|s,a,\pi], \tag{6.17}$$

where R denotes the random cumulative (discounted) rewards to obtain if the agents start from state s and take the action a and then always follow the policy (strategy) π.

Based on the definition of the action-value function, the optimal action-value function is defined as

$$Q_\pi^*(s,a) = Q_\pi(s,a). \tag{6.18}$$

The optimal action-value function represents the optimal expected cumulative (discounted) rewards to obtain if the agent starts from state s, action a. If we know the exact optimal action-value function, it is easy to derive the optimal policy π - we can just choose the action that maximizes $Q_\pi^*(s,a)$. But how to get the optimal action-value function is the key problem. And DQN aims at approximating the optimal action-value function by (deep) neural networks. Let $Q(s,a;\theta)$ denote the neural networks parameterized by θ.

Compared with the vanilla DQN, double DQN is more commonly used due to its advantages. The training of these neural networks is still done in a supervised manner, i.e., specify a loss function and update the parameters based on gradient-based methods. An important idea to improve the performance of DQN is the experience replay which is a brilliant idea and worth learning, but we will not cover it here.

Let's see how to do RL in practice. Although there are some RL tools in R, I recommend using Python because there are many mature tools in the deep learning ecosystem. To find the optimal strategy of the game we discussed in the previous section, we will use two packages, i.e., `gym`[26] and `stable_baselines`[27]. The `gym` package helps us define the environment and the `stable_baselines` package contains functionalities for building the DQN agent.

In `gym` there is a class `Env` based on which we can define our own environment class. More specifically, our environment class has to have three methods customized to fit the problem, i.e., the `__init__` method, the `reset` method and the `step` method. The `reset` method specifies how to reset the environment before we start the learning. The `step` method specifies how the environment reacts to the action and it should return the next state and the reward. Also, `step` method should return if the current game is finished or not. Regarding our game, the state that the agent observes in each step could be the current time in the game, and the reward is always 1. If the

[26]https://gym.openai.com
[27]https://stable-baselines.readthedocs.io/en/master

current environment time is greater than the time horizon, the game is finished. We could define the environment class in the following code snippet.

Python chapter6/game_env.py

```python
import numpy as np
from gym import Env, spaces

class GameEnv(Env):
    """
    H: the finite horizon of the game to specify
    """
    def __init__(self, H):
        self.H = H
        # there are two actions available — 0 and 1, thus we use
            a discrete(2) space
        self.action_space = spaces.Discrete(2)
        # the observation/state has a single dimension to
           reflect the current environment time
        self.observation_space = spaces.Box(low=np.array([0]),
           high=np.array([self.H]))
        self.current_time = 0.0

    def reset(self):
        self.current_time = 0.0
        return np.array([self.current_time])

    def step(self, action):
        if action == 0:
            self.current_time += 1.0
        elif action == 1:
            self.current_time += 2.0
        done = self.current_time >= self.H
        reward = 1
        info = {}
        return np.array([self.current_time]), reward, done, info
```

And the code below shows how to use a DQN agent to play the game with the help of the `stable_baselines` package.

Python chapter6/game_env_run.py

```python
import numpy as np
from game_env_run import GameEnv
from stable_baselines import DQN
from stable_baselines.common.vec_env import DummyVecEnv
from stable_baselines.deepq.policies import MlpPolicy

if __name__ == "__main__":
    # the horizon of the game is 100 time units
    H = 100
    # create the instance of our game environment
    env = GameEnv(H)
    env.reset()
    # create the DQN agent with MlpPolicy which is a predefined
        neural network and our env instance
    model = DQN(MlpPolicy, env, gamma=1.0, verbose=0)
    # let's train the model with 20000 timesteps
    model.learn(total_timesteps=20000)

    # now let's test our agent with 1000 games
    n = 1000
    env.reset()
    r, a = [], []
    for _ in range(n):
        current_r = 0.0
        obs = env.reset()
        done = False
        while not done:
            action, _ = model.predict(obs)
            obs, rewards, done, _ = env.step(action)
            current_r += rewards
            a.append(action)
        r.append(current_r)
    print("the average cumulative rewards from {0} games is {1}"
        .format(
        n, np.mean(r)))
```

By running the code above, we have the following output.

Python

```
chapter6 $python3.7 game_env_run.py
the average cumulative rewards from 1000 games is 100.0
```

In the game we discussed above, the environment does not have stochasticity and the optimal policy does not depend on the environment's state at all as it is a simplified example to illustrate the usage of the tools. The real-world problems are usually much more complicated than the example, but we could follow the same procedure, i.e., define the environment for the problem and then build the agent for learning.

6.6 COMPUTATIONAL DIFFERENTIATION

In Chapter 6 we talked about the gradient-based optimization algorithms, such as the vanilla gradient descent. We implemented the gradient descent method for linear regression. If you have used modern machine learning frameworks such as `TensorFlow`[28] and `PyTorch`[29], you may remember in `TensorFlow` there is no need to define the gradient update step manually and instead the framework handled the update magically. That is because these frameworks have automatic differentiation implemented.

It is worth distinguishing numerical differentiation, symbolic differentiation and automatic differentiation. Symbolic differentiation[30] aims at finding the derivative of a given formula with respect to a specified variable. It takes a symbolic formula as input and also returns a symbolic formula. For example, by using symbolic differentiation we could simplify $\partial(x^2 + y^2)/\partial x$ to $2x$. Numerical differentiation is often used when the expression/formula of the function is unknown (or in a black box, for example, a computer program). Compared with symbolic differentiation, numerical differentiation and automatic differentiation try to evaluate the derivative of a function at a fixed point. For example, we may want to know the value of $\partial f(x, y)/\partial x$ when $x = x_0; y = y_0$. The most basic formula for numerical differentiation is $\lim_{h \to 0}(f(x + h)) - f(x))/h$. In implementation, this limit could be approximated by setting h to a very small value, for example $1e - 6$. Automatic differentiation also aims at finding the numerical values of derivatives, in a more efficient approach based on chain rule. However, unlike the numerical differentiation, automatic differentiation requires the exact function formula/structure. Also, automatic differentiation does not need the approximation for the limit which is done in numerical differentiation. At the lowest level, it evaluates the derivatives based on symbolic rules. Thus automatic differentiation is partly symbolic and partly numerical [2].

Let's see an example of how to do symbolic differentiation in R and Python.

R

```r
> D(expression(x^2 + y^2), "
    x")
2 * x
```

Python

```python
>>> import sympy
>>> x = sym.Symbol('x')
>>> y = sym.Symbol('y')
>>> f = x**2 + y**2
>>> sym.diff(f, x)
2*x
```

It's worth noting there are many other symbolic operations we could do in R/Python, for example, symbolic integration.

Automatic differentiation is extremely important in modern machine learning. Many deep learning models can be viewed as function compositions. Let's take a two-hidden layer neural network illustrated below as an example. Let x denote the

[28]https://www.tensorflow.org/

[29]https://pytorch.org/

[30]https://www.cs.utexas.edu/users/novak/asg-symdif.html

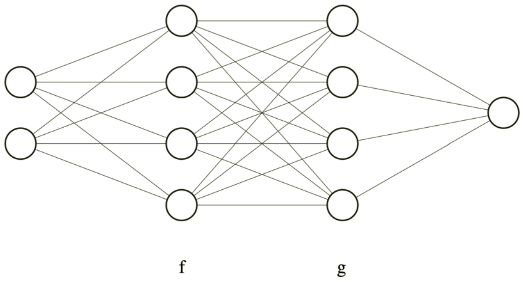


Figure 6.6: A neural network with two hidden layers f and g

input and f_u, g_v denote the first and second hidden layer, respectively. The output y is written as $y = g_v(f_u(x))$. The learning task is to estimate the parameters u, v in these hidden layers (see, Figure 6.6). If we use the gradient descent approach for the parameter estimation, the evaluation of gradients for both u and v should be calculated. However, there might be overlapped operations if we simply perform two numerical differentiation operations independently. By utilizing automatic differentiation technique, it is possible to reduce the computational complexity.

Let's see how we could utilize the automatic differentiation to simplify our linear regression implementation.

Python chapter6/linear_regression_ad.py

```python
from autograd import grad
import autograd.numpy as np

class LR_AD:
    """
    linear regression using automatic differentiation from
        autograd
    """
    def __init__(self, x, y, learning_rate=0.005, seed=42):
        self.seed = seed
        self.learning_rate = learning_rate
        self.x = np.hstack((np.ones((x.shape[0], 1)), x))
        self.y = y
        self.coef = None

    def loss(self):
        def loss_with_coef(coef):
            y_hat = self.x @ coef
            err = self.y - y_hat
            return err @ err.T / self.x.shape[0]
        # return the loss function with respect to the coef
        return loss_with_coef

    def fit(self, max_iteration=1000):
        # create the gradient function with the help of
            automatic differentiation
        grad_func = grad(self.loss())
        coef = np.random.uniform(size=(self.x.shape[1]))
        for _ in range(max_iteration):
            coef −= self.learning_rate*grad_func(coef)
        self.coef = coef
```

In the implementation above, we use `autograd.numpy` to replace the ordinary `numpy` package in order to perform automatic differentiation on the functions of interest. In the `loss` method, we return a function `loss_with_coef` which captures the data required for the computation. This type of function is also called closure[31].

6.7 MISCELLANEOUS

In this chapter, we have talked about machine learning in a superficial manner. We have seen some implementation of machine learning algorithms from scratch through out this book, which is mainly for teaching and illustration purposes. When we work on real-world data science projects we usually prefer to use the robust off-the-shelf machine learning libraries unless there is a need to customize the existing models or even to invent new models. In both R and Python, there are many such tools ready to use. Particularly, the CRAN task view[32] provides an up-to-date review of machine learning tools in R.

Many popular machine learning tools have API available in both languages. There also exist some tools for interoperability between Python and R. These tools allow us to use R in Python and vice versa, for example, the `reticulate`[33] library in R.

It is worth noting that machine learning is not only about the learning theories. It is also an engineering problem indeed.

[31]https://en.wikipedia.org/wiki/Closure_(computer_programming)
[32]https://cran.r-project.org/web/views/MachineLearning.html
[33]https://rstudio.github.io/reticulate

Bibliography

[1] Andrew R Barron. Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions on Information Theory*, 39(3):930–945, 1993.

[2] Atılım Günes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *The Journal of Machine Learning Research*, 18(1):5595–5637, 2017.

[3] Stephen Boyd and Lieven Vandenberghe. *Convex optimization.* Cambridge University Press, 2004.

[4] George Casella and Roger L Berger. *Statistical inference*, volume 2. Duxbury Pacific Grove, CA, 2002.

[5] John M Chambers et al. Object-oriented programming, functional programming and r. *Statistical Science*, 29(2):167–180, 2014.

[6] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016.

[7] George Dantzig. *Linear programming and extensions.* Princeton University Press, 2016.

[8] Jerome Friedman, Trevor Hastie, and Rob Tibshirani. glmnet: Lasso and elastic-net regularized generalized linear models. *R package version*, 1(4), 2009.

[9] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning.* Springer series in statistics New York, NY, USA:, second edition, 2009.

[10] William H Greene. *Econometric analysis.* Pearson Education India, 2003.

[11] Harold V Henderson and Paul F Velleman. Building multiple regression models interactively. *Biometrics*, pages 391–411, 1981.

[12] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[13] Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. Evaluating the design of the r language. In *European Conference on Object-Oriented Programming*, pages 104–131. Springer, 2012.

[14] Neal Parikh, Stephen Boyd, et al. Proximal algorithms. *Foundations and Trends in Optimization*, 1(3):127–239, 2014.

[15] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT Press, 2018.

[16] Peter JM Van Laarhoven and Emile HL Aarts. Simulated annealing. In *Simulated annealing: Theory and applications*, pages 7–15. Springer, 1987.

Index