Chapter 2

BASIC PROGRAMMING LANGUAGES

CONTENTS

2.1 What is R?

R is a free and integrated programming language for statistical computing and graphical display. It is widely used among probabilists, statisticians and data scientists, some elementary statistical tests and functions are already built in **R**, such that no extra libraries have to be called out. The name **R**[1] partially came after the first names of the first two **R** creators: Ross Ihaka and Robert Gentleman, and another reason is to supersede and to be on the top of **S**, another programming language.

2.2 What is Python?

Python is used *very* widely as a scripting language - Pixar[2] uses it in all their tools for making a computer-animated movie such as *Toy Story* 3. Python bears some resemblance to Java, but generally is cleaner

[1] Kurt Hornik. The **R** FAQ: Why **R**?. ISBN 3-900051-08-9. Retrieved 12 January 2021.
[2] Pixar Animation Studios, a subsidiary of The Walt Disney Company. Company website: `https://www.pixar.com/`.

and easier to read. The name of Python came from a British comedy group performers *Monty Python*[3], it reflects the goal of Python's developers - keep it funny.

Particularly in Python, we can enjoy the following features:

1. Do not need to declare the types of variables, for example, write `x = 1` rather than `int x = 1`. You can first write `x = 1` as an integer, then `x = "statistics"` as a string now in the later parts but still in the same programme;
2. The environment, in which you work with, lets you type bits of code and see what happens without an intermediate compiling step, making experimentation and testing very simple;
3. It is so easy to read, and uses English keywords and natural-sounding syntax.

   Example

   ```
   x = 1
   if x > 0:
       print("x is positive")
       print("Time to begin with machine learning!")
   ```

4. Instead of using curly braces to delimit code blocks, Python uses, for instance 4 (actually can be of any length as long as by consistent throughout the programme) whitespace indentation, and correctly nesting your code now has a semantic meaning.

Python is similar to Java in the sense that it also handles your memory management: allocates memory for different uses, and the capacity of freeing up that memory once it is no longer needed. In the rest of this book, the Python codes shown are consistent with the Python 3.7 version.

2.3 Package Management in R and Python

R and Python are two most popular programming languages for machine learning and deep learning developers, and there are hundreds of packages and libraries available online. In **R**, the installation and the importation of packages are done directly in the console:

```
> install.packages("e1071")
> library(e1071)
```

Here "e1071" is one of the packages for building a support vector machine (SVM) model in **R**, and this SVM model will be further discussed in Section 7.2.

However, in Python, one cannot install the packages directly in the console. Using Anaconda[4], one of the most popular data science platforms, we first locate *Anaconda Prompt* in the computer machine. If you are

[3] "General Python FAQ". Python v2.7.3 documentation. Docs.python.org. Retrieved 4 June 2020.

[4] Anaconda is a distribution mainly for Python and **R**, it consists of different Integrated Development Environments (IDE), including Spyder, Jupyter Notebook, RStudio, Visual Studio Code. We can imagine Anaconda as a computer system between Windows, Mac OS, or Linux, and the IDEs as the web browsers, let say Chrome, Microsoft Edge, or Safari. These computer systems provide us a platform to search online, in which we can choose different web browsers. We can install Anaconda from `https://www.anaconda.com/`.

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

using the Windows computer system, it can be searched in the *Windows Search Bar*. Then, we install the packages in the *Anaconda Prompt* by typing "pip install tensorflow" as in Figure 2.3.1.

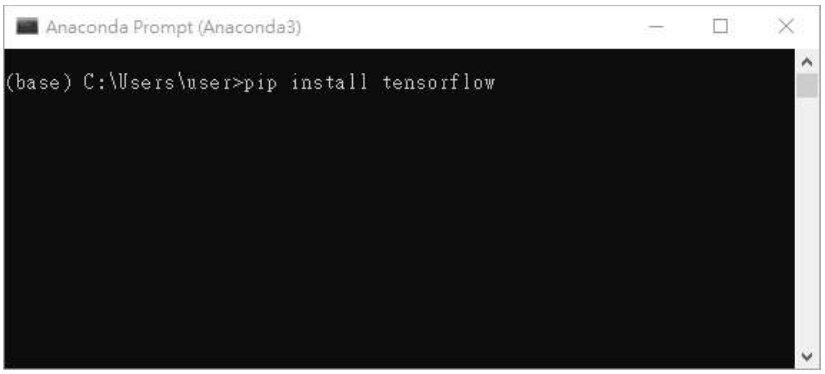


Figure 2.3.1: Typing "pip install tensorflow" in Anaconda prompt.

Here `pip` is used to install the Python related packages. There is another command `conda`, which is used to install packages written in any languages available in Anaconda. After the installation, we can import the package in the console by typing:

```
1 import tensorflow as tf
```

Here `tf` is used as a self-defined, short-handed notation for `tensorflow`. The tensorflow package is widely used for deep learning, it has an advanced version for GPU (Graphical Processing Unit), which is called the `tensorflow-gpu`. It significantly reduced the training time of a deep learning model, but it requires a decent GPU and additional software installation CUDA (Compute Unified Device Architecture) and cuDNN (CUDA Deep Neural Network library). Moreover, the importation of `tensorflow-gpu` is by typing `import tensorflow as tf`, GPU will automatically be used instead of CPU. We can check our CPU and GPU by:

```
1 from tensorflow.python.client import device_lib
2 print(device_lib.list_local_devices())
```

```
1  [name: "/device:CPU:0"
2  device_type: "CPU"
3  memory_limit: 268435456
4  locality {
5  }
6  incarnation: 14792078817090895925
7  , name: "/device:GPU:0"
8  device_type: "GPU"
9  memory_limit: 7020285133
10 locality {
11     bus_id: 1
12     links {
13     }
14 }
15 incarnation: 1319059476946399408
16 physical_device_desc: "device: 0, name: GeForce RTX 2070 SUPER, pci bus id:
       0000:01:00.0, compute capability: 7.5"
17 ]
```

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

Here "GeForce RTX 2070 SUPER" is the GPU used and the reference index for this GPU is 0. If there exists another GPU, the reference index for the second GPU is 1, and so on. The memory limits for CPU and GPU are shown in bytes, and we see that, for instance, the limit of the present GPU is 30 times more than that of CPU.

2.4 Basic Operations in R and Python

In this section, the dataset `fin-ratio.csv` (See Subsection 6.5.1) is used as an illustration for the first and the second items:

1. **Read and Write csv/txt Files**:
   CSV (Comma Separated Values) files or txt (TeXT) files are two common files for data storage. In **R**, there is no need to import any libraries for this action.

```
> # Read csv in R
> data <- read.csv("fin-ratio.csv")
> # Write csv in R
> write.csv(data, "fin-ratio_new.csv")
```

Programme 2.4.1: Read and write csv file in **R**.

   In Python, we need the `pandas` (derived its name from the PANel DAta, *i.e.* observations over multiple time periods for the same object) library:

```
import pandas as pd

# Read csv in Python
data = pd.read_csv('fin-ratio.csv')
# Write csv in Python
data.to_csv("fin-ratio_new.csv")
```

Programme 2.4.2: Read and write csv file in Python.

   Note that `data` must be a `pandas.DataFrame` object in order to use the `to_csv()` function.

2. **Sample Mean, Sample Variance, and Sample Covariance Matrix**:
   In **R**, there is no need to import any libraries for this action.

```
> # Assign data to x except the label y = HSI stock or not
> x <- data[-ncol(data)]
> # Compute sample means and sample variances for each columns
> apply(x, MARGIN=2, FUN=mean)
        EY        CFTP       ln_MV          DY        BTME         DTE
-0.6502403 -0.2338956  6.2668068  2.4961735  1.9082626  0.7097322
> apply(x, MARGIN=2, FUN=var)
       EY       CFTP      ln_MV         DY       BTME        DTE
18.497907  3.693061  2.743936 13.871563 68.308197 12.992907
>
```

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

```
> # Compute sample covariance matrix
> cov(x)
               EY       CFTP        ln_MV           DY        BTME           DTE
EY     18.497907  2.9089644   1.16018856   1.9203766   1.4781279   0.33795296
CFTP    2.908964  3.6930613   0.76629950   1.2371466   1.8228390   0.32879079
ln_MV   1.160189  0.7662995   2.74393616   0.9720714  -0.7734227  -0.07413221
DY      1.920377  1.2371466   0.97207145  13.8715626  -0.2575337   0.15815280
BTME    1.478128  1.8228390  -0.77342270  -0.2575337  68.3081966   1.96176520
DTE     0.337953  0.3287908  -0.07413221   0.1581528   1.9617652  12.99290723
```

Programme 2.4.3: Basic statistics measures in **R**.

Here the `apply()` function "applies" a function specified by the parameter `FUN` on the dimension specified by the parameter `MARGIN`, in which the argument `MARGIN=2` indicates the function is applied on the second dimension, *i.e.* column. In Python, the `pandas` library have built-in sample means, same variances, and sample covariances.

```python
# Assign data to x except the label y = HSI stock or not
x = data.drop(data.columns[-1], axis=1)
# data.columns[-1] = HSI, and axis=1 means column (axis=0 for row)
# Compute sample means and sample variances
print(x.mean())
print(x.var())

# Compute sample covariance matrix
print(x.cov())
```

```
EY       -0.650240
CFTP     -0.233896
ln_MV     6.266807
DY        2.496174
BTME      1.908263
DTE       0.709732
dtype: float64
EY        18.497907
CFTP       3.693061
ln_MV      2.743936
DY        13.871563
BTME      68.308197
DTE       12.992907
dtype: float64
               EY       CFTP      ln_MV           DY        BTME         DTE
EY     18.497907   2.908964   1.160189    1.920377    1.478128    0.337953
CFTP    2.908964   3.693061   0.766300    1.237147    1.822839    0.328791
ln_MV   1.160189   0.766300   2.743936    0.972071   -0.773423   -0.074132
DY      1.920377   1.237147   0.972071   13.871563   -0.257534    0.158153
BTME    1.478128   1.822839  -0.773423   -0.257534   68.308197    1.961765
DTE     0.337953   0.328791  -0.074132    0.158153    1.961765   12.992907
```

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

Programme 2.4.4: Basic statistics measures in Python.

3. **Building Mathematical Function**:

   In **R**, there is no need to import any libraries for the `exp` operation.

```
> MyFun <- function(x){
+     return (exp(x))
+ }
```

Programme 2.4.5: A simple exponential function `MyFun()` saved as `My_Function.R` in **R**.

Within the same folder, we can call back our `MyFun()` function in another file with command:

```
> source("My_Function.R")
>
> MyFun(2)
[1] 7.389056              # = e^2
```

Programme 2.4.6: Calling the function `MyFun()` saved in Programme 2.4.5 in **R**.

In Python, we need to import the `math` library or the `numpy` (stands for NUMerical PYthon) library for the `exp` operation.

```
import numpy as np

def MyFun(x):
    return (np.exp(x))
```

Programme 2.4.7: A simple exponential function `MyFun()` saved as `My_Function.py` in Python.

Similarly, within the same folder, we can call back our `MyFun()` function in another file with command:

```
from My_Function import MyFun

MyFun(2)
```

```
7.38905609893065          # = e^2
```

Programme 2.4.8: Calling the function `MyFun()` saved in Programme 2.4.7 in Python.

For both **R** and Python, due to limited space for display, the printed results can only show a few number of significant figures, while in calculations, more significant figures are used. Actually, Python uses more digits in calculation than **R**.

4. **Recursion Operation - For Loop**:

   In **R**, the iteration starts from 1:

```
> # For loop in R
> for (i in 1:5){
+     print(i)
```

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

```
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

Programme 2.4.9: A simple for loop in **R**.

However, in Python, the iteration starts from zero:

```
# For loop in Python
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

Programme 2.4.10: A simple for loop in Python.

Similarly, the index in **R** starts with 1 while the index in Python starts with 0 as pointed before. Therefore, extreme care has to be taken in translating the indexing between **R** and Python.

5. **Help Function**: In **R**, you can always call `help(function)` or `?function` for documentation. For example, `mean` or `?mean` for the documentation of the mean function. In Python, you can also call `help(function)`. Moreover, under Spyder, one of the most popular programming platform for the Python language, one can just put the cursor under the function for documentation (See Figure 2.4.1).

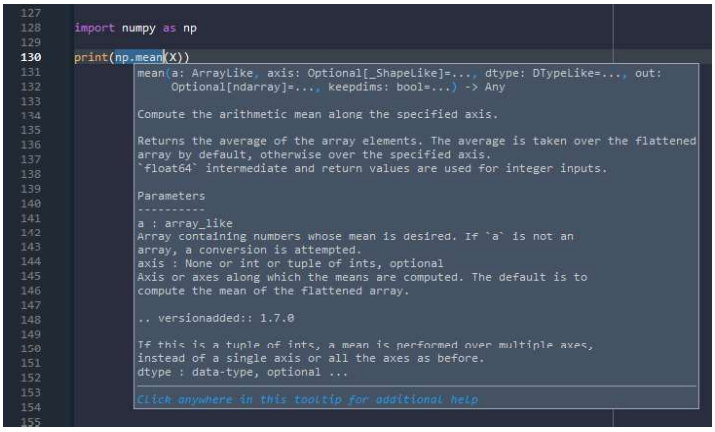


Figure 2.4.1: Function documentation can be shown when the curser is put on the top of that function.

6. **Matrix Computation**: In **R**, we do not need to import any libraries:

```
> A <- matrix(c(1, 3, 5, 2, 6, 4, 2, 3, 1), nrow=3, ncol=3, byrow=T)
> B <- matrix(c(3, 1, 2, 4, 2, 8, 1, 3, 1), nrow=3, ncol=3, byrow=T)
```

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

```
> A
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    6    4
[3,]    2    3    1
> B
     [,1] [,2] [,3]
[1,]    3    1    2
[2,]    4    2    8
[3,]    1    3    1
>
> # Matrix Multiplication
> A%*%B
     [,1] [,2] [,3]
[1,]   20   22   31
[2,]   34   26   56
[3,]   19   11   29
>
> # Hadamard Product
> A * B
     [,1] [,2] [,3]
[1,]    3    3   10
[2,]    8   12   32
[3,]    2    9    1
>
> # Inverse
> solve(A)
           [,1]       [,2]       [,3]
[1,]  0.3333333 -0.6666667  1.0000000
[2,] -0.3333333  0.5000000 -0.3333333
[3,]  0.3333333 -0.1666667  0.0000000
```

Programme 2.4.11: Simple matrix operation in **R**.

In Python, we need to import the `numpy` library for the matrix operation:

```
A = np.array([1, 3, 5, 2, 6, 4, 2, 3, 1]).reshape(3,3)
B = np.array([3, 1, 2, 4, 2, 8, 1, 3, 1]).reshape(3,3)
print(A)
print(B)

# Matrix Multiplication
print(np.dot(A, B))
print(A @ B)

# Hadamard Product
print(np.multiply(A, B))
print(A * B)

```

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

```
14 # Inverse
15 print(np.linalg.inv(A))          # LINear ALGebra
```

```
1  [[1 3 5]
2   [2 6 4]
3   [2 3 1]]
4
5  [[3 1 2]
6   [4 2 8]
7   [1 3 1]]
8
9  [[20 22 31]
10  [34 26 56]
11  [19 11 29]]
12
13 [[20 22 31]
14  [34 26 56]
15  [19 11 29]]
16
17 [[ 3  3 10]
18  [ 8 12 32]
19  [ 2  9  1]]
20
21 [[ 3  3 10]
22  [ 8 12 32]
23  [ 2  9  1]]
24
25 [[ 0.33333333 -0.66666667  1.        ]
26  [-0.33333333  0.5        -0.33333333]
27  [ 0.33333333 -0.16666667  0.        ]]
```

Programme 2.4.12: Simple matrix operation in Python (Line breaks are added for easy identification).

2.5 Simulation in R and Python

The idea of simulations originated from the building of atomic bomb dated back to Manhattan Project in 1942 by the time of World War II. In the 1930s, scientists such as Einstain, Oppenheimer and von Neumann, discovered the phenomenon of nuclear fission where an atom is decomposed into two, when it is hit by another atom, by then a huge amount of energy is released as a byproduct: particularly though the energy released by a single nuclear fission is negligible, a chain reaction of collisions makes a great difference by triggering nearby atomic nuclear fission and spreading throughout the whole radiactive substance, all results in a domino effect.

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

2.5.1 Pseudo-Random Number Generator

One obstacle faced in simulating atomic behavior is the generation of random numbers from $[0, 1]$; indeed, true randomness cannot be achievable by desktop machines. *Pseudo Random Number Generator* is used in order to mimic random phenomena by generating seemingly random values but through a deterministic mechanism.

2.5.1.1 Chaos Theory

In a chaotic system, apparent random and appearance of its states is actually led by deterministic initial conditions through a nonlinear sensitive generating mechanism; indeed, small disturbances in the initial conditions, even after a short time, can cause resulting states to be significantly different from each other, as a consequence, this unfortunate unpredictability can serve as a way of producing randomness.

2.5.1.2 Linear Congruential Generator

The linear congruential generator is one of the most common and fundamental Pseudo Random Number Generators; it is actually a nonlinear and one of the simplest chaotic dynomial systems, which generates a sequence of states x_n by:

$$x_{n+1} \equiv (ax_n + c) \ (mod \ m) \, . \tag{2.5.1}$$

With an initial seed x_0, while m is chosen to be a large prime number, c is a small positive number in comparison to m. The value of $x_{n+1} \equiv (ax_n + c) \ (mod \ m)$ is very sensitive to the initial seed x_0 even with a modest value of n. According to standard results in Chaos Theory, after a long run, the number x_n will significantly uncorrelate with x_0 or even any x_k, for $n \gg k$, obtained long before.

Given x_0, a sample of the uniform random variable $U[0, 1]$ can be obtained by using x_k/m, for $k = 1, \cdots, n$, where x_k follows the modular arithmetic dynamics (2.5.1); further in practice, we may drop the first portion of x_k's, but only use those x_k's with $k \geq k_0$ for some large enough k_0, let say, $k_0 = 1{,}000$. The feasibility of this sampling is warranted by a well-known result in Ergodic Theory:

$$\lim_{n\to\infty} \frac{1}{n} \sum_{j=0}^{n-1} 1_{(a,b)} \left(\frac{x_j}{m} \right) = b - a, \qquad \text{for any} \quad 0 \leq a < b \leq 1 \, . \tag{2.5.2}$$

where the indicator $1_{(a,b)}(z) = 1$ whenever $z \in (a, b)$ or equal to 0 otherwise. We now shows two respective programmes in **R** and Python, on how we implement the pseudo random number generator for uniform random varaibles on $[0, 1]$.

```
> library(numbers)      # For mod function
>
> # Linear Congruential Generator
> pse_uni_gen = function(mult=16807, modu=(2^31)-1, seed=123456789, size=1){
+     u = matrix(0,size,1)
+     x = mod((seed*mult+1),modu)
```

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

```
+      u[1] = x/modu
+      for (i in 1:size){
+          x = mod((x*mult+1),modu)
+          u[i] = x/modu
+      }
+      return(u)
+ }
>
> # Pseudo Uniform Generator
> pse_uniform_gen = function(low=0, up=1, seed=123456789, size=1){
+      return(low+(up-low)*pse_uni_gen(seed=seed,size=size))
+ }
>
> unif_pse = pse_uniform_gen(size=11000)[1001:11000]
> # R built-in uniform random variable generator
> unif_rand = runif(n=10000)
```

Programme 2.5.1: Pseudo uniform generator and built-in uniform generator in **R**.

Here the `numbers` library is imported to use the `mod` function for calculating the modulo of two numbers, and the `runif` (Random UNIForm) is the **R** built-in function for generating standard uniform random numbers, $U[0,1]$ by default, in which the argument `n=10000` indicates that 10,000 random numbers are generated. Under this settings, the initial seed is $x_0 = 123456789$, $a = 16807$, $c = 1$, and the prime number is $m = 2^{31} - 1$. Similarly in Python:

```
import numpy as np
from numpy.random import rand

# Linear Congruential Generator
def pse_uni_gen(mult=16807, mod=2**31-1,seed=123456789,size=1):
    u = np.zeros(size)
    x = (seed*mult+1)%mod
    u[0] = x/mod
    for i in range(1,size):
        x = (x*mult+1)%mod
        u[i] = x/mod
    return u

# Pseudo Uniform Generator
def pse_uniform_gen(low=0,up=1,seed=123456789,size=1):
    return low+(up-low)*pse_uni_gen(seed=seed,size=size)

pse_sample = pse_uniform_gen(low=0,up=1,size=11000)[1001:11000]
# Uniform random variable generator in numpy library
built_in_sample = rand(10000)
```

Programme 2.5.2: Pseudo uniform generator and numpy standard uniform generator in Python.

Here `numpy.random` is imported for generating standard uniform random numbers by using the `rand()` function, where the additional argument `10000` indicates that 10,000 random numbers are generated. In comparison to `runif()` in **R**, `rand()` can only generate standard uniform random numbers on the interval

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

[0,1] and it takes only size or shape as the arguments. The `zeros()` function under the `numpy` library generates a multi-dimensional array of zeros with shape equals to the `size` parameter, where the shape argument must be a list of numbers, let say `[2, 3]` for generating a 2×3 array of zeros, or a number, let say `10` for generating a one-dimensioal array of zeros with size 10. Finally, the `%` operator calculates the modulus of two numbers.

Next, we plot the histograms for the pseudo uniform generator and the built-in uniform generator in **R** and Python with the `hist()` (HISTogram) function:

```
> hist(unif_pse, col=rgb(red=0,green=0,blue=1.0,alpha=0.5), main="Histogram")
> hist(unif_rand, col=rgb(red=1.0, green=0, blue=0, alpha=0.5), add=TRUE)
> legend('bottomright',legend=c("Pseudo Generator","R Generator"),
+        fill=c(rgb(red=0,green=0,blue=1.0,alpha=0.5),
+               rgb(red=1.0,green=0,blue=0,alpha=0.5)))
```

Programme 2.5.3: Histogram for pseudo uniform generator and built-in uniform generator in **R**.

Here the `rgb()` function sets the color intensities for each red, green, and blue color ranging from 0 to 1, in which a small value indicates that such color is light, where the additional parameter `alpha` sets the transparency of the colors ranging from 0 to 1 (0 being fully transparent that is no color at all and 1 being "solid"). The `add=TRUE` in the second line indicates that the second histogram is plotted by "adding" to the same figure as the first histogram. The `legend()` function adds a figure legend to the histogram, in which "bottomright" indicates that the figure legend will be located in the bottom right hand corner.

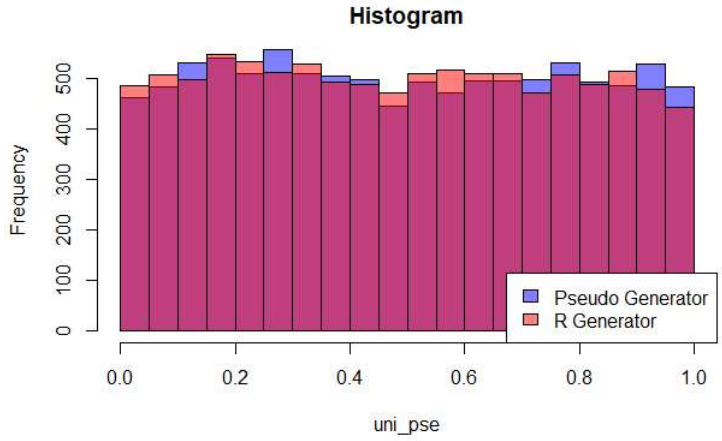


Figure 2.5.1: Histogram of the **R** generator and pseudo generator samples.

```
import matplotlib.pyplot as plt      # A plotting library

plt.hist(pse_sample, bins=20, alpha=0.5, label="Pseudo Generator")
plt.hist(built_in_sample, bins=20, alpha=0.5, label="Numpy Generator")
plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
plt.legend()
plt.savefig("prnguni.png")
plt.show()
```

Programme 2.5.4: Histogram with original settings in Python.

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

Here `matplotlib.pyplot` (MATlab-PLOTting-LIBrary.PYthon-PLOT) is imported for plotting the histogrames. The argument `bins=20` indicates that 20 number of bins are used, and `alpha` is again the transparency of the colors as same as in **R**. The figure legend is added through the `legend()` function. Lastly, the `savefig()` function saves the histogram in the same folder where this programme is stored with a name specified as "prnguni.png". The `show()` function "shows" the histogram on the console. Note that the `savefig()` function must be called before the `show()` function, otherwise, a blank white plot will be saved.

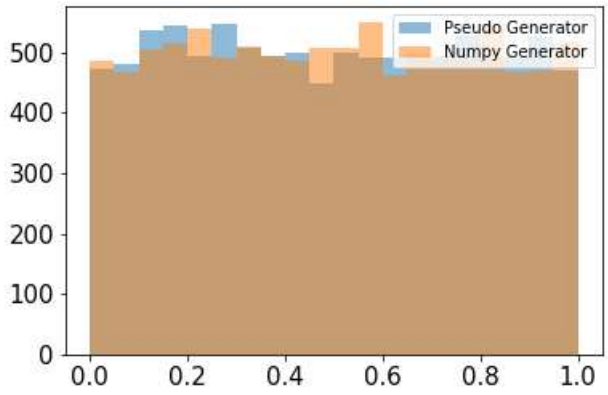
```
plt.hist(pse_sample, alpha=0.5, label="Pseudo Generator",
         color="b", ec='black', bins="sturges")
plt.hist(built_in_sample, alpha=0.5, label="Numpy Generator",
         color="r", ec='black', bins="sturges")
plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
plt.legend()
plt.savefig("prnguni_colorR.png")
plt.show()
```

Programme 2.5.5: Histogram with settings similar to **R** in Python.

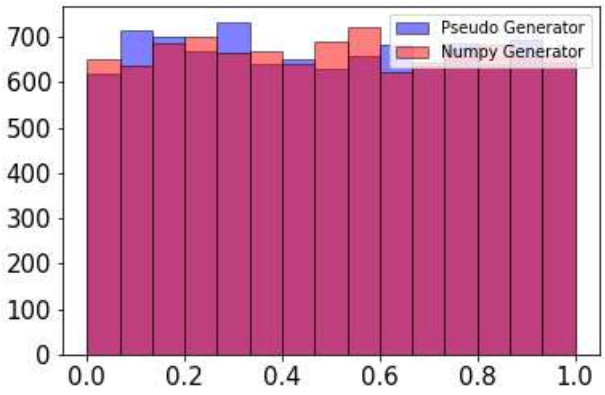
Here the name `ec` stands for Edge Color and the `bins` parameter can take either a number or a string. In **R**, "sturges" is the default bining method used in histogram. This method calculates the number of bins based on the formula:

$$k = \lceil \log_2 n \rceil + 1 \,, \tag{2.5.3}$$

where n is the total number of samples, $\log_2$ is the binary logarithm with base 2, and the $\lceil x \rceil$ function returns the smallest integer that is larger than x.



(a) Original histogram plotted by Programme 2.5.4.



(b) Histogram similar to **R** format plotted by Programme 2.5.5.

Figure 2.5.2: Histograms of the Python generator and pseudo generator samples.

Figure 2.5.2(a) and Figure 2.5.2(b) show the same pseudo uniform distribution generated by Python. However, the former figure seems to be elementary, while the later figure appears to be more neat and informative.

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

2.5.2 Inverse Function Transform

Given $f_{\mathrm{x}}(x)$ the probability density function of a random variable x, also denote $F_{\mathrm{x}}(x)$ as its cumulative distribution function. If one can generate a uniform distribution, we can simulate x through an explicit expression of the inverse of F_{x}; indeed, if $\mathrm{u} \sim U(0,1)$, *i.e.* $\mathbb{P}(\mathrm{u} < u) = u$ for $u \in (0,1)$, then $\mathrm{x} = F_{\mathrm{x}}^{-1}(\mathrm{u}) \sim F_{\mathrm{x}}$; and this can be seen by noting that $\mathbb{P}(F_{\mathrm{x}}^{-1}(\mathrm{u}) \leq x) = \mathbb{P}(\mathrm{u} \leq F_{\mathrm{x}}(x)) = F_{\mathrm{x}}(x)$. Conversely, $F_{\mathrm{x}}(x) \sim U[0,1]$. This inverse function transform technique can simulate quite a class of random variables, for example the exponential distribution, and see the programmes in **R** and Python below.

```
> # Pseudo Exponential Generator
> pse_exp_gen = function(lamb, size=1){
+     t = as.integer(Sys.time())
+     u = pse_uni_gen(size=size, seed=t)
+     x = -(1/lamb)*log(1-u)
+     return(x)
+ }
>
> pse_sample = pse_exp_gen(lamb=1, size=11000)[1001:11000]
> # R built-in exponential random variable generator
> built_in_sample = rexp(n=10000)
```

Programme 2.5.6: Pseudo exponential generator and built-in exponential generator in **R**.

Here the `rexp()` (Random-EXPonential) function is the **R** built-in function for generating standard exponential random numbers, Exp(1) by default, in which the argument `n=10000` indicates that 10,000 random numbers are generated. The `Sys.time()` function returns an absolute date-time value, and the `as.integer()` function further transforms this value into the total number of seconds since January 1, 1970 (midnight UTC/GMT). The `log()` (LOGarithmic) function returns the natural logarithmic value. Similarly in Python,

```
import numpy as np
import time
from numpy.random import exponential

# Pseudo Exponential Generator
def pseudo_exp_gen(lamb,size=1):
    t = time.perf_counter()
    seed = int(10**9*((t-int(t))))
    u = pse_uniform_gen(size=size, seed=seed)
    x = -(1/lamb)*np.log(1-u)
    return x

pse_sample = pseudo_exp_gen(scale=1, size=11000)[1001:11000]
# Exponential random variable generator in numpy library
built_in_sample = exponential(scale=1, size=10000)
```

Programme 2.5.7: Pseudo exponential generator and numpy exponential generator in Python.

Here the `perf_counter()` (PERFormance-COUNTER) function under the `time` library returns the float value of time in seconds between the moment when you open the Spyder application or other Python IDE

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

and the moment when you compile this line of the code. Since this float value will change over time, the initial seed calculated based on this float value will also change over time. The `exponential()` function in `numpy.random` generates exponential random numbers in which the `scale=1` argument indicates the hazard rate $\lambda = 1$ and the argument `size=10000` indicates that 10,000 random numbers are generated. Finally, the `log()` (LOGarithm) function from the `numpy` library returns the natural logarithmic value.

Next, we plot the histograms for the pseudo exponential generator and the built-in exponential generator in **R** and Python:

```
> hist(pse_sample, col=rgb(red=0,green=0,blue=1.0,alpha=0.5), main="Histogram")
> hist(built_in_sample, col=rgb(red=1.0, green=0, blue=0, alpha=0.5), add=TRUE)
> curve(10000*dexp(x, rate=1), add=TRUE, col='black', lwd=2.5)
> legend('topright',legend=c("Pseudo Generator","R Generator"),
+        fill=c(rgb(red=0, green=0, blue=1.0, alpha=0.5),
+               rgb(red=1.0, green=0, blue=0, alpha=0.5)))
```

Programme 2.5.8: Histogram for pseudo exponential generator and built-in exponential generator in **R**.

Here the `curve()` function plots a "curve" based on the specified function `10000*dexp(x, rate=1)`, in which the argument `lwd=2.5` indicates that the Line WiDth of the curve is scaled up by a factor of 2.5, where the base, *e.g.* `lwd=1` will change relative to the plot window in the console. The higher is the number assigned to `lwd`, the thicker is the "curve". Lastly, the `dexp()` (Density-EXPonential) function returns the density value of the exponential distribution evaluated at point x, in which the argument `rate=1` indicates that the exponential distribution has hazard rate $\lambda = 1$.

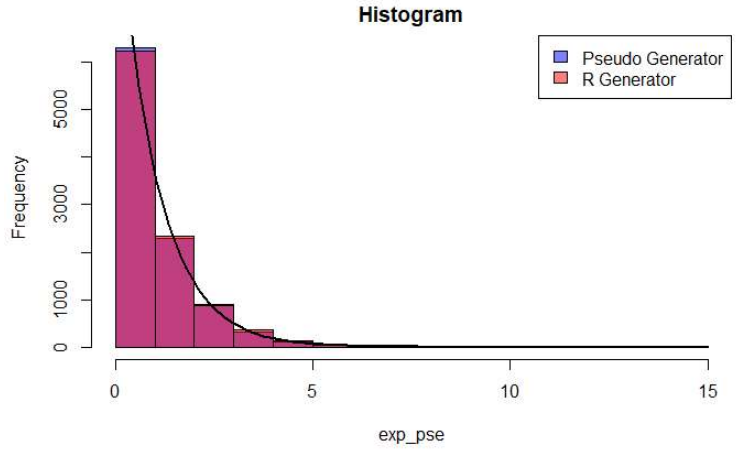


Figure 2.5.3: Histogram of the **R** generator and pseudo generator samples.

In Python:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import expon   # SCIentific PYthon

x = np.linspace(start=expon.ppf(0.01), stop=expon.ppf(0.99), num=10000)
plt.hist(pse_sample, alpha=0.5, color="b", label="Pseudo Generator")
plt.hist(built_in_sample, alpha=0.5, color="r", label="Numpy Generator")
```

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

```
plt.plot((x),(expon.pdf(x))*10000)
plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
plt.legend()
plt.show()
```

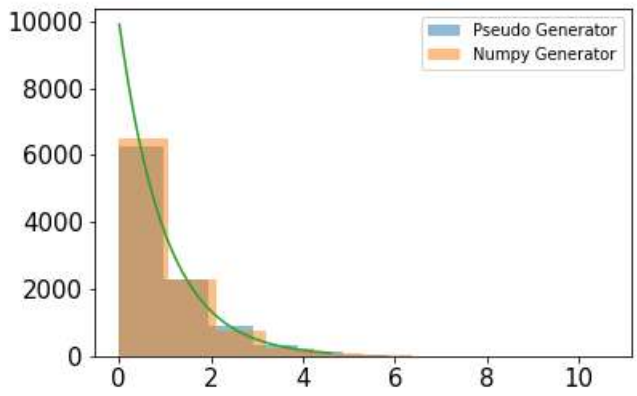
Programme 2.5.9: Histogram with original settings in Python.

Here the `linspace()` (LINear-SPACE) function in the `numpy` library returns an array of evenly spaced steps over a specified interval `[start, stop]`, in which the argument `num=10000` indicates the total number of steps. Finally, the `expon.ppf()` (EXPONential-Percent-Point-Function) in `scipy.stats` (SCIentific-PYthon.STATisticS) returns the inverse cumulative density value $F_x^{-1}(q)$ with the argument $q \in [0, 1]$ (Quantile).

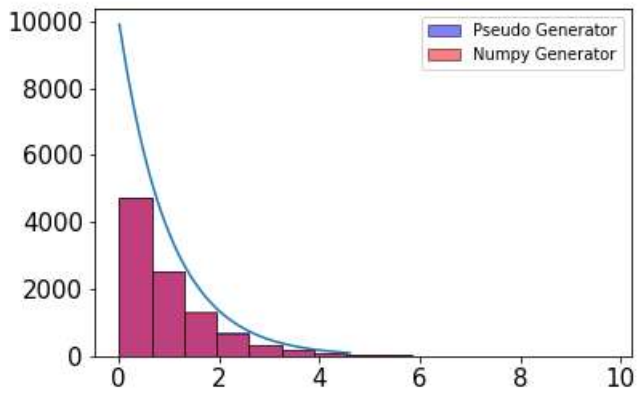
```
_, bins, _ = plt.hist(pse_sample, alpha=0.5, label="Pseudo Generator",
                      color="b", ec='black', bins="sturges")
plt.hist(built_in_sample, alpha=0.5, label="Numpy Generator",
                      color="r", ec='black', bins=bins)
plt.plot((x),(expon.pdf(x))*10000)
plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
plt.legend()
plt.savefig("prngexp_colorR.png")
plt.show()
```

Programme 2.5.10: Histogram with settings similar to **R** in Python.

Here the `hist()` function from `matplotlib.pyplot` returns three results. The first result is the total number of samples falling in each bins; the second result is the starting point of each bins; the third result is a "BarContainer" object containing the individual artists used to create this histogram. Since the exponential random numbers generated by the pseudo random number generator and by the numpy exponential random number generator are different, the bins generated by these two methods have different starting points. We can pass the `bins` result as an argument for the second histogram.



(a) Original histogram plotted by Programme 2.5.9 with different starting points for each bin.



(b) Histogram similar to **R** format plotted by Programme 2.5.10 with the same starting point for each bin.

Figure 2.5.4: Histograms of the Python generator and pseudo generator samples.

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

2.5.3 Box-Muller Transformation

Although Inverse Function Transform is the most fundamental tool for distribution simulation, a potential limitation to its use is especially when F_{x}^{-1} cannot be expressed out explicitly. Normal distribution is certainly the case since the inverse of its cumulative distribution function cannot be written as a sum of finitely many elementary functions. Fortunately, an alternative approach via the celebrated *Box-Muller Transformation* makes a simulation of a pair of normal random variables plausible. Let u_1 and u_2 be independent and follow uniform distribution $U[0,1]$, we write

$$\mathrm{z}_1 := \sqrt{-2\ln \mathrm{u}_1}\cos(2\pi \mathrm{u}_2) \qquad \text{and} \qquad \mathrm{z}_2 := \sqrt{-2\ln \mathrm{u}_1}\sin(2\pi \mathrm{u}_2)\,, \qquad \text{(Box-Muller Transformation)}$$

both of which follow $N(0,1)$ independently. To see this, by a simple change-of-variable, we can express u_1 and u_2 as follows:

$$\begin{cases} \mathrm{u}_1 = \exp\left\{-\dfrac{\mathrm{z}_1^2+\mathrm{z}_2^2}{2}\right\} =: \exp\left\{\dfrac{-q}{2}\right\}, \\ \mathrm{u}_2 = \dfrac{1}{2\pi}\arctan\left\{\dfrac{\mathrm{z}_2}{\mathrm{z}_1}\right\}, \end{cases}$$

with the Jacobian matrix being

$$J = \begin{pmatrix} -z_1\exp\{-q/2\} & -z_2\exp\{-q/2\} \\ \frac{-z_2}{2\pi(z_1^2+z_2^2)} & \frac{z_1}{2\pi(z_1^2+z_2^2)} \end{pmatrix}.$$

Since the joint probability density function of u_1 and u_2 is simply

$$f(u_1,u_2) = 1, \qquad \text{for} \quad 0<u_1,u_2<1\,,$$

it follows that the joint probability density function of z_1 and z_2 should be

$$g(z_1,z_2) = \left|\frac{-1}{2\pi}\exp(-q/2)\right|\cdot(1) = \frac{1}{2\pi}\exp\left\{-\frac{z_1^2+z_2^2}{2}\right\}, \qquad \text{for} \quad -\infty<z_1,z_2<\infty\,.$$

Note that there is some difficulty with the definition of this transformation, particularly when $z_1 = 0$. However, these difficulties occur with probability zero, which causes no problem at all. See Programme 2.5.11 and Programme 2.5.12 for an illustration of programmes in **R** and Python respectively.

```
> # Pseudo Normal Generator (Box-Muller Transform)
> pse_norm_gen <- function(mu=0.0, sigma=1.0, size=1){
+     t <- (as.integer(Sys.time()))
+     u1 <- pse_uni_gen(seed=t,size=size)
+     t <- as.integer(Sys.time())
+     u2 <- pse_uni_gen(seed=t,size=size)
+     z0 <- sqrt(-2*log(u1))*cos(2*pi*u2)
+     z1 <- sqrt(-2*log(u1))*sin(2*pi*u2)
+     z0 <- z0*sigma + mu
+     return(z0)
+ }
>
> norm_pse <- pse_norm_gen(size=11000)[1001:11000]
> # R built-in normal random variable generator
> norm_rand <- rnorm(n=10000)
```

Programme 2.5.11: Pseudo normal generator and built-in normal generator in **R**.

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

Here the `rnorm()` (Random NORMal) function is the **R** built-in function for generating normal random numbers, by default $\mathcal{N}(0,1)$, in which `n=10000` indicates that 10,000 random numbers are generated. Similarly in Python:

```
import numpy as np
import time
from numpy.random import normal

# Pseudo Normal Generator (Box-Muller Transform)
def pseudo_normal_gen(mu=0.0, sigma=1.0, size=1):
    t = time.perf_counter()
    seed1 = int(10**9*(t-int(t)))
    u1 = pse_uniform_gen(seed=seed1,size=size)
    t = time.perf_counter()
    seed2 = int(10**9*((t-int(t))))
    u2 = pse_uniform_gen(seed=seed2,size=size)
    z0 = np.sqrt(-2*np.log(u1))*np.cos(2*np.pi * u2)
    z1 = np.sqrt(-2*np.log(u1))*np.sin(2*np.pi * u2)
    z0 = z0*sigma+mu
    return z0

pse_sample = pseudo_normal_gen(mu=0, sigma=1, size=11000)[1001:11000]
# Normal random variable generator in numpy library
built_in_sample = normal(loc=0,scale=1,size=10000)
```

Programme 2.5.12: Pseudo normal generator and numpy normal generator in Python.

Here the `normal()` function from `numpy.random` generates normal random numbers in which the `loc=1` (LOCation) argument indicates that it has a mean $\mu = 0$, the `scale=1` argument indicates that it has a standard deviation $\sigma = 1$, and the argument `size=10000` indicates that 10,000 random numbers are generated.

Next, we plot the histograms for the pseudo normal generator and the built-in normal generator in **R**:

```
> hist(norm_pse, col=rgb(red=0, green=0, blue=1.0, alpha=0.5), main="Histogram")
> hist(norm_rand, col=rgb(red=1.0, green=0, blue=0, alpha=0.5), add=TRUE)
> legend('topright',legend=c("Pseudo Generator","R Generator"),
+        fill=c(rgb(red=0,green=0,blue=1.0,alpha=0.5),
+               rgb(red=1.0,green=0,blue=0,alpha=0.5)))
```

Programme 2.5.13: Histogram for pseudo normal generator and built-in normal generator in **R**.

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

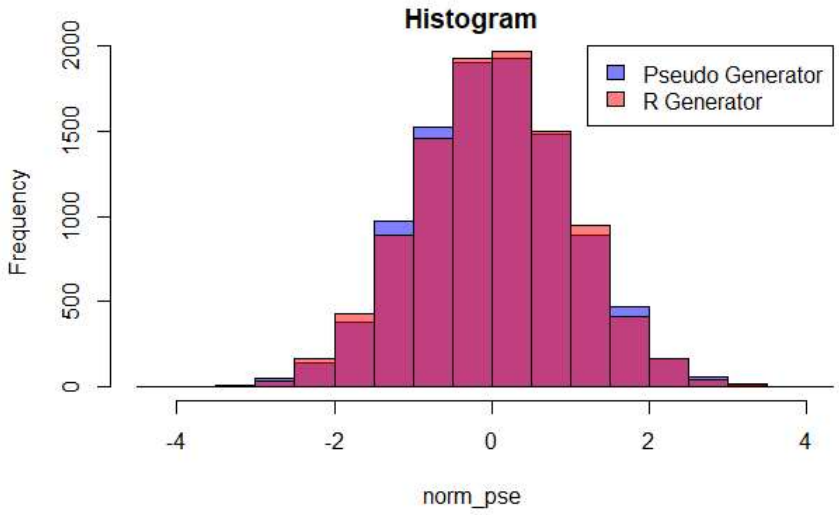


Figure 2.5.5: Histogram of the **R** generator and pseudo generator samples

Similarly in Python:

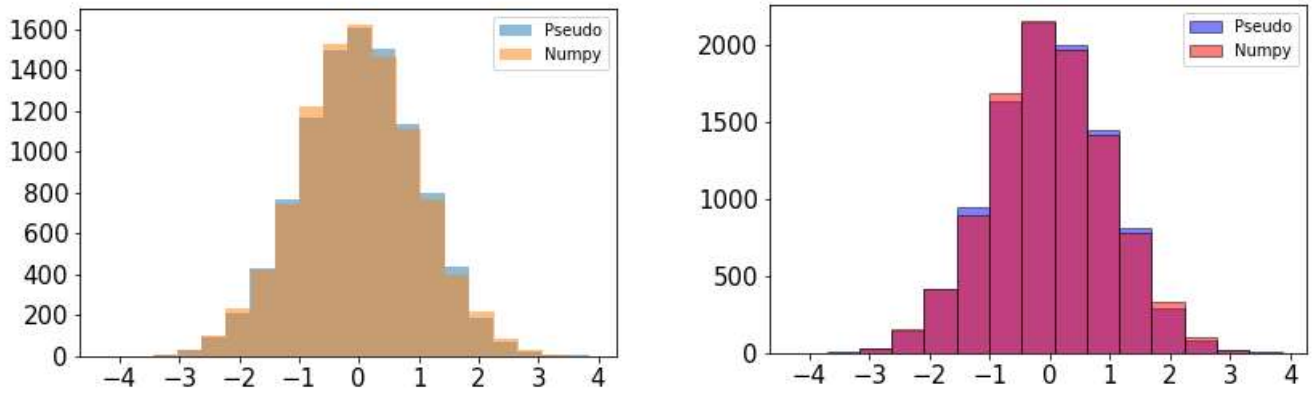
```
import matplotlib.pyplot as plt

plt.hist(pse_sample, bins=20, alpha=0.5, color="b", label="Pseudo")
plt.hist(built_in_sample, bins=20, alpha=0.5, color="r", label="Numpy")
plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
plt.legend()
plt.show()
```

Programme 2.5.14: Histogram with original settings in Python.

```
_, bins, _ = plt.hist(pse_sample, alpha=0.5, label="Pseudo",
                      color="b", ec='black', bins="sturges")
plt.hist(built_in_sample, alpha=0.5, label="Numpy",
         color="r", ec='black', bins=bins)
plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
plt.legend()
plt.savefig("prngnorm_colorR.png")
plt.show()
```

Programme 2.5.15: Histogram with settings similar to **R** in Python.

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.



(a) Original Histogram plotted by Programme 2.5.14 with different starting points for each bin.

(b) Histogram similar to **R** format plotted by Programme 2.5.15 with the same starting point for each bins.

Figure 2.5.6: Histograms of the Python generator and pseudo generator samples.

2.5.4 Normality Test in R and Python

The basic assumption under the celebrated **Black-Scholes-Merton** model (Black and Scholes (1972)) is that the percentage change in the stock price over a short period of time δt is normally distributed, *i.e.*

$$\delta S/S \sim \mathcal{N}(\mu\delta t, \sigma^2\delta t)\,, \tag{2.5.4}$$

where δS is the change in the stock price S from t to $t+\delta t$, $\mu\delta t$ is the mean of percentage change and $\sigma\sqrt{\delta t}$ is the standard deviation of this percentage change; μ is called the drift rate and σ stands for the volatility (σ^2 for the variance rate). Usually the stock price is observed over a time period (*e.g.* the closing price is observed daily, so that $\delta t = 1/252$, normally around 252 trading days in a calender year). Consider N observed daily stock prices $S_1, \ldots, S_N$, and define the log return

$$u_n = \ln\left(\frac{S_n}{S_{n-1}}\right) \sim \mathcal{N}(\mu\tau, \sigma^2\tau), \qquad n = 1, \cdots, N\,, \tag{2.5.5}$$

where $\tau = 1/252$ in place of δt from now on. To see whether this assumption is correct, we can use some graphical methods to test for normality.

The file "0005.HK.csv" contains the daily prices for HSBC (0005) from 1/1/2020 to 31/12/2020. Let us first read in these stock prices and compute u_n for HSBC according to (2.5.5). According to (2.5.5), u_n is expected to be normally distributed. The easiest graphical method is to construct a histogram and see whether it is a bell-shaped curve; surely, it is too rough, and a more sophisticated graphical method is the normal Q-Q plot. The normal Q-Q plot is a graphical method for testing univariate normality, and it is essentially the most effective one as it is the graphical version of the very powerful Shapiro-Walk test[5] (Shapiro and Wilk (1965)) . To this end, we order the u_n in ascending order denoted by $u_{(n)}$, *i.e.* $u_{(1)} \le u_{(2)} \le \cdots \le u_{(N)}$.

[5]The Shapiro-Wilk test is a way to formally test whether a sample is normally distributed. The test statistic is $W = \frac{(\sum_{n=1}^{N} a_n x_{(n)})^2}{\sum_{n=1}^{N}(\bar{x}_N - x_n)^2}$, where $x_{(n)}$ are the ordered sample values and vector $a = m^\top V^{-1}/C$ where C is the vector norm $C = (m^\top V^{-1} V^{-1} m)^{1/2}$ and the vector m is made up of the expected values of the order statistics of iid random variables sampled from the standard normal distribution; finally, V is the covariance matrix of those normal order statistics.

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

The normal Q-Q plot is the plot of these ordered $u_{(n)}$ against $q_{(n)}$, the n-th quantile of the standard normal distribution, *i.e.* $\mathbb{P}\{z < q_{(n)}\} = (n - 0.5)/N$ (with a continuity adjustment); conceptually, the right hand side should be n/N, but a reduction of 0.5 is used to avoid the difficulty when $n = N$. If these u_n's were normally distributed, this plot should be close to a straight line. However for **R**, the actual plots illustrate that the proportional changes are usually heavily tailed for large changes. Next, we look at the histogram and the normal Q-Q plot of u_n. Figure 2.5.7 for **R** and Figure 2.5.8 for Python shows that the distribution of u_n has a fatter tail than a normal distribution.

```
> HSBC <- read.csv("0005.HK.csv")
> X <- HSBC$Adj.Close
> log_return <- diff(log(X))
>
> hist(log_return)
> qqnorm(log_return)
> qqline(log_return)
```

Programme 2.5.16: QQ-Norm test for HSBC in **R**.

Here the `diff()` (lagged-DIFFerence) function returns the successive differences, by default `differences=1`, of the numeric vector X.

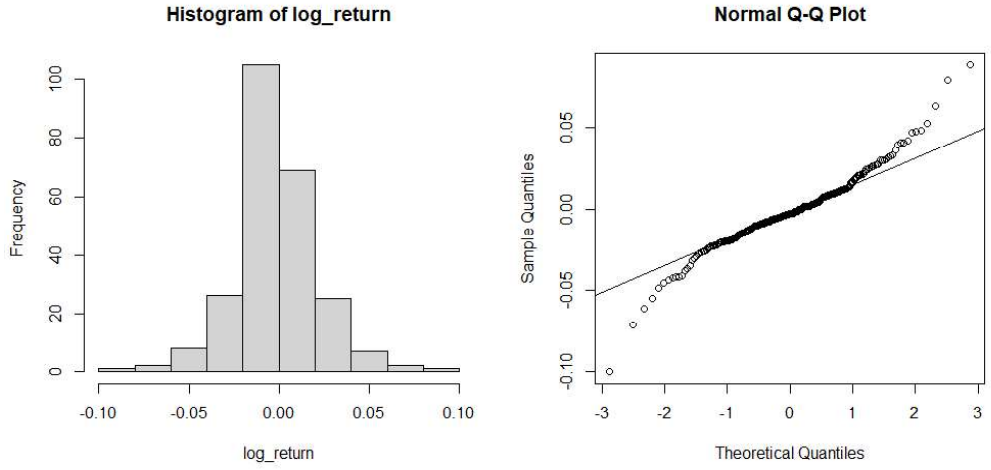


Figure 2.5.7: Histogram (Left) and Normal Q-Q Plot(Right) for HSBC log-return in **R**.

Similarly in Python:

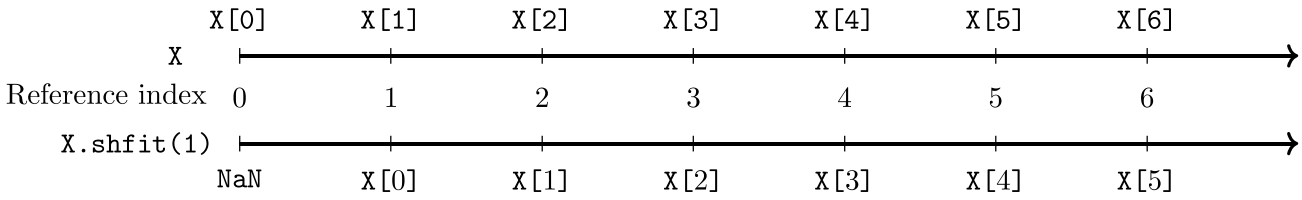
```
import matplotlib.pyplot as plt
import scipy.stats as stats

HSBC = pd.read_csv("0005.HK.csv")
X = HSBC["Adj Close"]
log_returns = np.log(X) - np.log(X.shift(periods=1))
log_returns = log_returns.iloc[1:]

fig = plt.figure(figsize=(13,5))
plt.subplot(1, 2, 1)
plt.hist(log_returns, bins="sturges", ec='black')
```

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

```
plt.title("Histogram_HSBC")

plt.subplot(1, 2, 2)
stats.probplot(log_returns, dist="norm", plot=plt)
plt.title("QQNorm_HSBC")
plt.savefig("HSBC Norm python.png")
plt.show()
```

Programme 2.5.17: QQ-Norm test for HSBC in Python.

Here the `shift()` function from the `pandas` library "shifts" the reference indexes of our vector `X` by adding `periods=1` to each reference index.



From the figure above, the first item `X.shift(1)[0] = NaN`. Therefore, `iloc[1:]` (Integer-LOCation) function from the `pandas` library is used to select the data starting from index 1, and then are assigned to `log_returns`. The `figsize=(13,5)` (FIGure-SIZE) argument in the `matplotlib.pyplot.figure` function controls respectively the width and height in inches. The argument `1,2,1` in the `subplot()` function controls respectively the number of plots in a row (`1`), the number of plots in a column (`2`), and the reference position (`1`) of the current plot in the figure. Under the current settings, the last number of this argument cannot be larger than 2, because there are at most $1 \times 2 = 2$ number of plots. Lastly, the `probplot()` (PROBability PLOT) function from `scipy.stats` return the probability plot of the given dataset against the theoretical distribution specified by the parameter `dist`, in which the argument `dist="norn"` indicates that the normal distribution is used.

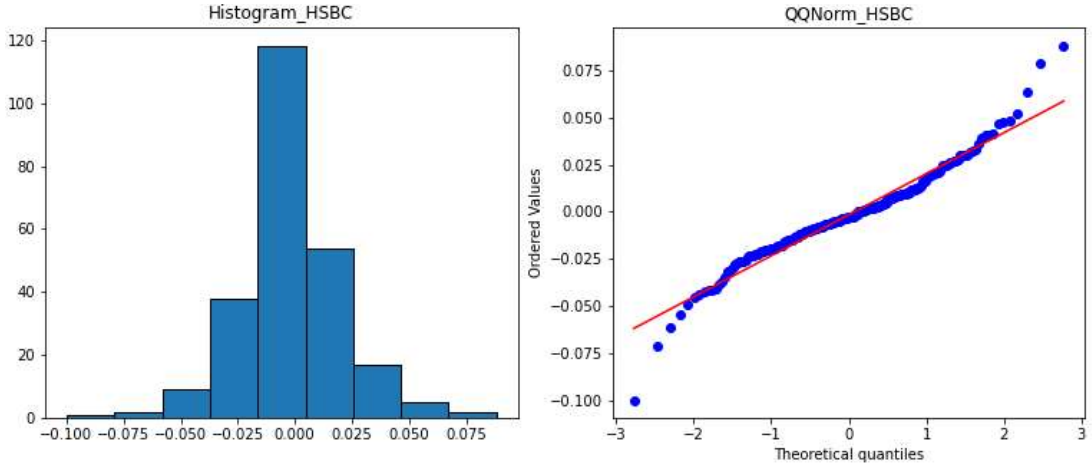


Figure 2.5.8: Histogram (Left) and Normal Q-Q Plot(Right) for HSBC log-return in Python.

Although we set the same bining methods for both **R** and Python, the histograms in Figure 2.5.7 (Left) and Figure 2.5.8 (Left) are still different. One can easily see that the histogram in Figure 2.5.7 (Left) has 10 bins but the histogram in Figure 2.5.8 (Left) has only 9 bins. The total number of entries is 247, where 1 data is used up for calculating the log-return. Using (2.5.3), $k = \lceil \log_2(247-1) \rceil + 1 = 9$, but **R** has 10

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

bins. Therefore, extreme care is needed to read the histogram produced.

There are other formal statistical tests for testing normality. The **Kolmogorov-Smirnov (KS)** test can be used to test whether a random sample is coming from a specific distribution. The test statistic is $D_N = \sup_{x\in\mathbb{R}} |F_N(x) - F(x)|$, where $F_N(x)$ and $F(x)$ are the empirical, with a sample of size N, and theoretical distribution functions, respectively; see Figure 2.5.9. Note that the distribution of $\sqrt{N}D_N$ converges to that of the absolute maximum of a standardized Brownian bridge on $[0,1]$ as N goes to infinity [3], and it is independent to the theoretical distribution function $F(x)$. In practice, the empirical distribution function $F_N(x)$ is:

$$F_N(x) := \frac{1}{N}\sum_{n=1}^{N} \mathbb{1}_{\{x_n \le x\}},$$

where $\mathbb{1}_A$ is an indicator function taking value 1 if the event A occurs and 0 otherwise. For example, in testing whether the dataset $z_1, \cdots, z_N$ follows a standard normal distribution, the KS test statistics is given by:

$$D_n := \sup|F(z) - S_N(z)| = \max_{i=1,\cdots,N} \left[\frac{i}{N} - \Phi(z_{(i)}), \Phi(z_{(i)}) - \frac{i-1}{N} \right],$$

where $z_{(1)} < \cdots < z_{(N)}$ are the sorted values of $z_1, \cdots, z_N$, and $\Phi(\cdot)$ is the standard normal continuous distribution function CDF. The critical value at $\alpha \in [0,1]$ ($100(1-\alpha)^{th}$ percentile of the KS statistics) is given by:

$$c(\alpha) = \sqrt{\frac{-0.5 \cdot \ln(\alpha/2)}{N}}.$$

Therefore, we reject the null hypothesis that $z_1, \cdots, z_N$ follows a standard normal distribution if $D_N > c(\alpha)$ or the p-value is less than 0.05. Of course, the standard normal CDF $\Phi(\cdot)$ can be replaced by any other arbitrary CDFs $F(\cdot)$ for testing whether the dataset follows the corresponding distribution.

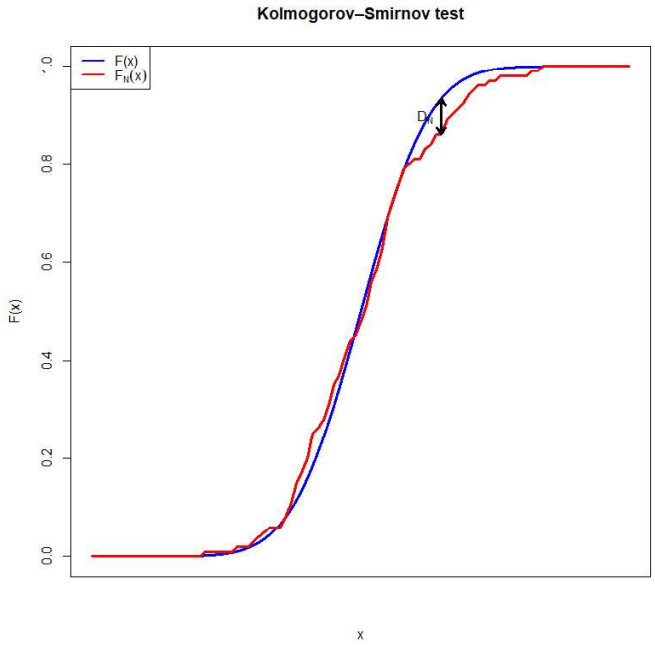


Figure 2.5.9: An illustrative example on KS test statistics.

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

```
> ks.test(log_return, y=pnorm, mean=mean(log_return), sd=sd(log_return))

One-sample Kolmogorov-Smirnov test

data:  log_return
D = 0.079562, p-value = 0.0888
alternative hypothesis: two-sided
>
> KS_test <- function(u, func=pnorm, ...){
+     n <- length(u)
+     S_n <- (0:n)/n
+     z <- sort(u)
+     Phi_z <- func(z, ...)
+     Term1 <- S_n[2:(n+1)] - Phi_z
+     Term2 <- Phi_z - S_n[1:n]
+     KS <- max(Term1, Term2)
+     crit <- sqrt(-0.5*log(0.05/2)/n)
+     list(KS.stat=KS, KS.crit=crit)
+ }
>
> KS_test(log_return, func=pnorm, mean=mean(log_return), sd=sd(log_return))
$KS.stat
[1] 0.07956208

$KS.crit
[1] 0.08658939
```

Programme 2.5.18: Kolmogorov-Smirnov test for HSBC daily return in **R**.

Here the `ks.test()` and the self-defined `KS_test()` function returns the hypothesis test results of the Kolmogorov-Smirnov (KS) test against a theoretical continuous distribution specified by the parameter `y` (resp. `func`), in which the argument `y=pnorm` (resp. `func=pnorm`) indicates that the normal distribution is used.

```
import numpy as np
import scipy.stats as stats

u = log_returns
print(stats.kstest(u, "norm", args=(np.mean(u), np.std(u))))

def KS_test(x, func=stats.norm, *args):
    n = len(x)
    S_n = np.arange(n+1)/n
    z = np.sort(x)
    Phi_z = func.cdf(z, *args)
    Term1 = S_n[1:] - Phi_z
    Term2 = S_n[:-1] - Phi_z
```

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

```
    KS = np.max([Term1, Term2])
    crit = np.sqrt(-0.5*np.log(0.05/2)/n)
    return({"KS-stat": KS, "critical value":crit})

print(KS_test(u, stats.norm, np.mean(u), np.std(u)))
```

```
KstestResult(statistic=0.07918316613234533, pvalue=0.08662726861876724)
{'KS-stat': 0.07918316613234533, 'critical value': 0.08658939004122418}
```

Programme 2.5.19: Kolmogorov-Smirnov test for HSBC daily return in Python.

In particular, the `stats.kstest()` function accepts an argument `args`, which is a tuple object, indicating the extra arguments passing to the theorectical function; while the self-defined `KS_test()` function accepts an argument `*args`, which can be a sequence of arguments, indicating the extra arguments that are passed to the function `func`.

Another commonly used normality test is the **Jarque-Bera** (**JB**) test. It is based on the fact that

$$\text{skewness} = 0 \qquad \text{and} \qquad \text{kurtosis} = 3\,,$$

for the standard normal distribution. The test statistic is

$$JB = N\left(\frac{\hat{\zeta}_1^2}{6} + \frac{(\hat{\zeta}_2^2 - 3)^2}{24}\right) \sim \chi_2^2\,,$$

asymptotically, where $\hat{\zeta}_1^2$ and $\hat{\zeta}_2^2$ are the sample standardized skewness and kurtosis:

$$\text{Skew}(\boldsymbol{x}) = \frac{\frac{1}{N}\sum_{n=1}^{N}(x_n - \bar{x}_N)^3}{\left(\frac{1}{N}\sum_{n=1}^{N}(x_n - \bar{x}_N)^2\right)^{3/2}} \qquad \text{and} \qquad \text{Kurt}(\boldsymbol{x}) = \frac{\frac{1}{N}\sum_{n=1}^{N}(x_n - \bar{x}_N)^4}{\left(\frac{1}{N}\sum_{n=1}^{N}(x_n - \bar{x}_N)^2\right)^{2}}\,.$$

We write the following function `JB_test()` to perform this test.

```
> library("normtest")
> jb.norm.test(log_return)

Jarque-Bera test for normality

data:  log_return
JB = 94.359, p-value < 2.2e-16

>
> JB_test <- function(x) {              # function for JB-test
+     u <- x - mean(x)                  # remove mean
+     n <- length(u)                    # sample size
+     s <- sd(u)*sqrt((n-1)/n)          # compute population sd
+     sk <- sum(u^3)/(n*s^3)            # compute skewness
+     ku <- sum(u^4)/(n*s^4)-3          # excess kurtosis
+     JB <- n*(sk^2/6+ku^2/24)          # JB test stat
+     p <- 1-pchisq(JB,2)               # p-value
+     list(JB.stat=JB, p.value=p)       # output
```

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

```
+ }
>
> JB_test(log_return)
$JB.stat
[1] 94.3587

$p.value
[1] 0
```

Programme 2.5.20: JB test for HSBC daily return in **R**.

Next, the `normtest` (NORMal TEST) library is imported for the `jb.norm.test()` function, which returns the hypothesis test results of the Jarque-Bera (JB) test against the normal distribution. Similarly in Python, we import `scipy.stats` for the `kstest()` function for the Kolmogorov-Smirnov (KS) test and the `jarque_bera()` function for the Jarque-Bera test.

```
import numpy as np
import scipy.stats as stats

print(stats.jarque_bera(log_returns))

def JB_test(x):                                    # function for JB-test
    u = x - np.mean(x)                             # remove mean
    n = len(u)                                     # sample size
    s = np.std(u)                                  # compute population sd
    sk = sum(u**3)/(n*s**3)                        # compute skewness
    ku = sum(u**4)/(n*s**4)-3                      # excess kurtosis
    JB = n*(sk**2/6+ku**2/24)                      # JB test stat
    p = 1 - stats.chi2.cdf(JB,2)                   # p-value
    return({"JB-stat": JB, "p-value": p})          # output

print(JB_test(log_returns))
```

```
Jarque_beraResult(statistic=94.35870436809749, pvalue=0.0)
{'JB-stat': 94.35870436809799, 'p-value': 0.0}
```

Programme 2.5.21: JB test for HSBC daily return in Python.

The p-values for the HSBC log-return u from both the **KS** test and **JB** tests are small, less that 0.05, thus we conclude the normality assumption for the return of HSBC is not valid, which is consistent with the observation from the normal Q-Q plots.

2.5.5 Basic Regression in R and Python

In this subsection, a famous historical data from Karl Pearson[6] is used for linear regression in **R** and Python. The dataset consists of the height measurements from 1078 fathers and their fully grown sons in England, circa 1900. Pearson's original data was measured and reported in the nearest inch. In **R**, we use the `lm()` (Linear Model) function for this simple linear regression model.

[6]Orginal data can be found in Pearson K and Lee A. (1903). On the laws of inheritance in man. Biometrika, 2:357-462.

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

```
> data <- read.csv("Pearson.txt", sep="\t")
> model <- lm(Son~., data=data)
> model$coefficients
(Intercept)      Father
 33.8928005   0.5140059
> plot(Son~., data=data,
 main="Heights of fathers and their full grown son (in inches)")
> abline(model)
```

Programme 2.5.22: Linear regression in **R**.

In Python, we need to import the `sklearn` (Scikit-learn) library, which includes various machine learning algorithms in classification, clustering, and regression. For example, support vector machines (`sklearn.svm()`), K-means clustering (`sklearn.cluster.KMeans()`), K-NN (`sklearn.neighbors.KNeighborsClassifier()`), and random forests (`sklearn.ensemble.RandomForestClassifier()`), where these machine learning algorithms will be further discussed in later chapters. In this simple linear regression model, we need to import the `LinearRegression()` function from `sklearn.linear_model`.

```
1  from sklearn.linear_model import LinearRegression
2  import numpy as np
3  import pandas as pd
4  import matplotlib.pyplot as plt
5
6  data = pd.read_csv("Pearson.txt", sep="\t")
7  model = LinearRegression()
8  x = data.iloc[:, 0].values.reshape(-1, 1)
9  y = data.iloc[:, 1].values.reshape(-1, 1)
10 model.fit(x, y)
11 print({"intercept": model.intercept_, "slope": model.coef_})
12 y_pred = model.predict(x)
13
14 plt.figure(figsize=(5, 5))
15 plt.scatter(x, y, edgecolors="black", color="white", s=20)
16 plt.plot(x, y_pred, color="red")
17 plt.xlabel("Father")
18 plt.ylabel("Son")
19 plt.title("Heights of fathers and their full grown son (in inches)")
20 plt.savefig("Pearson python.png")
21 plt.show()
```

```
{"intercept": array([33.89280054]), "slope": array([[0.51400591]])}
```

Programme 2.5.23: Linear regression in Python.

Referring to Line 7 of Programme 2.5.23. The `iloc(:,0)` function locates the first (0-th) column across all rows of `data`, whose values are then retrieved by the `values()` function and further reshaped to become a vector by the `reshape(-1,1)` function, here the two arguments represent the numbers of rows and columns, respectively, and the -1 in the row argument is a designated value that instructs Python to automatically determine the number of rows by itself. The argument `edgecolors="black"` indicates that the edge color of each marker is black; meanwhile the argument `s=20` indicates that the size of each marker in points square.

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

Therefore, together with the argument `color="white"`, we can plot a scatter plot in Python similar to that of **R**. The respective plots in **R** and Python are:

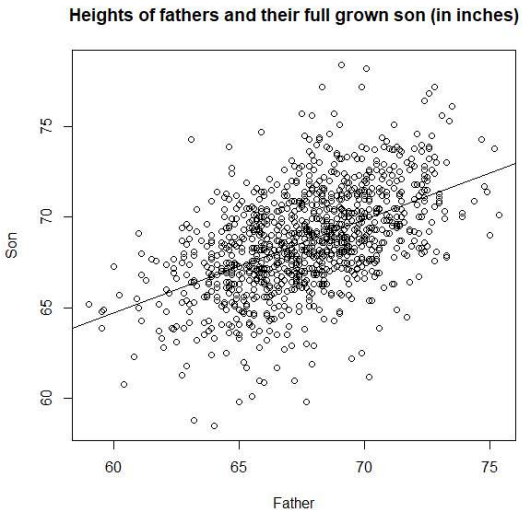


Figure 2.5.10: Linear regression with Pearson data in **R**.

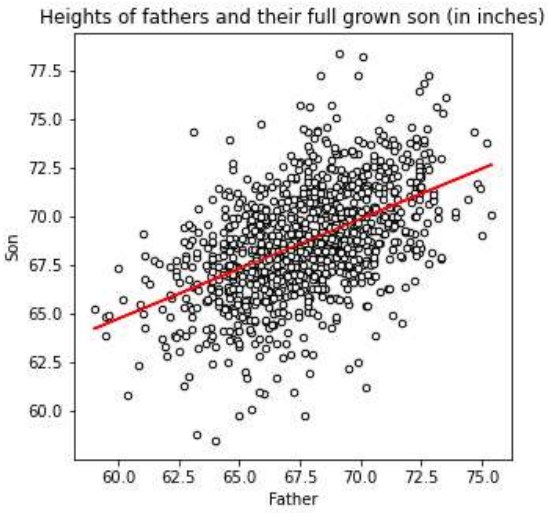


Figure 2.5.11: Linear regression with Pearson data in Python.

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

2.6 One-way ANOVA and Tukey's HSD with Stock Market Indices

One-way analysis of variance (ANOVA) is used to compare the means of two or more $|\mathcal{C}| := |\{c_1, \cdots, c_K\}| \geq 2$ categorical and independent groups of data with only one continuous feature (independent) variable[7]. It requires the assumption that all observations are normally and independently distributed with a common variance. Statistically, denote $\mu_1, \cdots, \mu_{|\mathcal{C}|}$ be the respective population mean of each categorical group governing by the only one continuous feature variable $x^{(1)}$, the null hypothesis of one-way ANOVA is:

$$H_0 : \mu_1 = \mu_2 = \cdots = \mu_{|\mathcal{C}|} ,$$

where, as always, we reject the null hypothesis that the population means are not all equal if the p-value is lower than 0.05. However, if the null hypothesis is rejected, one-way ANOVA cannot provide us much knowledge on which population mean(s) μ_k is/are significantly different from others. Tukey (1949) proposed the Tukey's HSD (Honest Significant Difference) which considers all possible pairs of population means. The score of a Tukey's HSD between the group c_i and the group c_j, where $i \neq j$, is given by:

$$q_{\text{tukey},ij} = \frac{\left|\bar{x}_i^{(1)} - \bar{x}_j^{(1)}\right|}{\text{SE}_{\text{ANOVA}}} , \tag{2.6.1}$$

where $\bar{x}_i^{(1)}$ and $\bar{x}_j^{(1)}$ are respectively the sample means of the group c_i and the group c_j, SE_{ANOVA} is the standard error computed from the ANOVA under H_0. Indeed, under H_0, since all population means are assumed to be equal, we can simplify q_{tukey} to

$$q_{\text{tukey},ij} = \frac{\left|\bar{x}_i^{(1)} - \bar{x}_j^{(1)}\right|}{\sqrt{\frac{1}{\bar{n}} \cdot \frac{\text{SSE}_{\text{within}}}{N - |\mathcal{C}|}}} , \tag{2.6.2}$$

where $\text{SSE}_{\text{within}}$ is the within sum of squared error:

$$\text{SSE}_{\text{within}} = \sum_{k=1}^{|\mathcal{C}|} \sum_{\substack{1 \leq n \leq N \\ c(x_n^{(1)}) = c_k}} (x_n^{(1)} - \bar{x}_k^{(1)})^2 \qquad \text{and} \qquad \bar{n} = \frac{|\mathcal{C}|}{\frac{1}{N_1} + \cdots + \frac{1}{N_{|\mathcal{C}|}}} ,$$

in which the harmonic mean of $N_1, \cdots, N_{|\mathcal{C}|}$ is adopted to compute $\bar{n}$ with N_k being the total number of samples in group $c_k \in \mathcal{C}$, and each $c(x_n^{(1)})$ denotes the corresponding class of the observation $x_n^{(1)}$. Notice that $\text{SE}_{\text{ANOVA}} = \sqrt{\sigma^2/(N - |\mathcal{C}|)}$ and under H_0 that all population means are equal, then

$$\sigma^2 = \sum_{n=1}^{N} \left(x_n^{(1)} - \frac{1}{N} \sum_{n=1}^{N} x_n^{(1)} \right)^2 .$$

Next, we shall illustrate the application of one-way ANOVA and Tukey HSD for the three stock market indices, namely the S&P500, HSI, and FTSE in both **R** and Python. `SPX.csv`, `HSI.csv`, and `FTSE.csv` are respectively the adjusted closing prices of S&P500, HSI, and FTSE dating from 4 January, 1984 to 26 April, 2021. In particular, 10 years of data from 2011 to 2020 are analysed.

```
> SPX <- read.csv("SPX.csv", header=TRUE, row.names=1)
> HSI <- read.csv("HSI.csv", header=TRUE, row.names=1)
> FTSE <- read.csv("FTSE.csv", header=TRUE, row.names=1)
>
> SPX$Year <- format(as.Date(rownames(SPX), format="%d/%m/%Y"), "%Y")
```

[7] For two or more continuous independent variables, this ANOVA analysis is extended to one-way multivariate analysis of variance (MANOVA); see more details in Timm (2002).

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

```
> HSI$Year <- format(as.Date(rownames(HSI), format="%d/%m/%Y"), "%Y")
> FTSE$Year <- format(as.Date(rownames(FTSE), format="%d/%m/%Y"), "%Y")
>
> SPX$log_return <- c(NA, diff(log(SPX$Price)))
> HSI$log_return <- c(NA, diff(log(HSI$Price)))
> FTSE$log_return <- c(NA, diff(log(FTSE$Price)))
>
> # Remove first day of data for log return
> SPX <- SPX[-1,]
> HSI <- HSI[-1,]
> FTSE <- FTSE[-1,]
>
> Remove_Outlier <- function(index, outlier_factor=1.5){
+     q25 <- quantile(index$log_return, probs=.25, na.rm=FALSE)
+     q75 <- quantile(index$log_return, probs=.75, na.rm=FALSE)
+     iqr <- q75 - q25  #Inter-quartile range
+     lower_bound <- q25 - outlier_factor*iqr
+     upper_bound <- q75 + outlier_factor*iqr
+     pos <- index$log_return > lower_bound & index$log_return < upper_bound
+     return (index[pos,])
+ }
>
> # Hyperparameter
> outlier_removal <- TRUE
> start_year <- 2011
> end_year <- 2020
>
> if (outlier_removal){
+     SPX <- Remove_Outlier(SPX)
+     HSI <- Remove_Outlier(HSI)
+     FTSE <- Remove_Outlier(FTSE)
+ }
>
> year_name <- paste0(" Year: ", start_year, "-", end_year)
> Chosen_Year <- start_year:end_year
> Chosen_SPX <- SPX[SPX$Year%in%Chosen_Year,]
> Chosen_HSI <- HSI[HSI$Year%in%Chosen_Year,]
> Chosen_FTSE <- FTSE[FTSE$Year%in%Chosen_Year,]
```

Programme 2.6.1: Stock market indices cleansing for one-way ANOVA and Tukey HSD in **R**.

Here `as.Date()` function converts the date character string from the index name of the dataframe into a datetime object, in which the argument `format="%d/%m/%Y"` indicates that the input string has a format of date (two digits) / month (two digits) / year (four digits). After that, `format()` function is immediately applied with an argument `"%Y"` indicating that only the year is returned, and this is augmented into the

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

stock market index dataframe in a new column named as `Year`. The logarithmic return is computed by the `log()` function, then calculate the difference of consecutive terms, or equivalently the logarithm of ratio of consecutive prices, via the `diff()` function. Note that the logarithmic return on the first day, *i.e.* 4 January, 1984, is unavailable, because we do not have the adjusted closing price on the previous day. Therefore, we augment `NA` in the beginning of the logarithmic return, and we locate it in a new column named as `log_return`. We then remove the data on the very first day by using `[-1,]`.

In the `Remove_Outlier()` function in either **R** or Python, one possible way, any data points which are lower than the lower quartile minus 1.5 times the inter-quartile range or greater than the upper quartile plus 1.5 times the inter-quartile range are regarded as the outliers. If each stock market index follows a standard normal distribution, then the 25% (resp. 75%) quantile represents a quantile value of -0.6745 (resp. 0.6745) under $\mathcal{N}(0,1)$. With a 1.5 `outlier_factor`, the `lower_bound` (resp. `upper_bound`) represents the quantile value $-0.6745 + 1.5 \cdot (0.6745 - (-0.6745)) = -2.6960$ (resp. 2.6960). Therefore, under $\mathcal{N}(0,1)$, we are selecting the middle proportion of $\Phi(2.6960) - \Phi(-2.6960) = 0.9930$, where $\Phi(\cdot)$ is the c.d.f. of a standard normal distribution. Therefore, if the stock market index follows a heavier tailed distribution than $\mathcal{N}(0,1)$, then the outliers accounts for roughly 1% of the data.

Similarly, in Python,

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

SPX = pd.read_csv("SPX.csv", index_col=0)
HSI = pd.read_csv("HSI.csv", index_col=0)
FTSE = pd.read_csv("FTSE.csv", index_col=0)

SPX["Year"] = pd.to_datetime(SPX.index.values).year
HSI["Year"] = pd.to_datetime(HSI.index.values).year
FTSE["Year"] = pd.to_datetime(FTSE.index.values).year

SPX["log_return"] = np.log(SPX.Price) - np.log(SPX.Price.shift(1))
HSI["log_return"] = np.log(HSI.Price) - np.log(HSI.Price.shift(1))
FTSE["log_return"] = np.log(FTSE.Price) - np.log(FTSE.Price.shift(1))

SPX = SPX.iloc[1:]
HSI = HSI.iloc[1:]
FTSE = FTSE.iloc[1:]

def Remove_Outlier(index, outlier_factor=1.5):
    q25 = np.quantile(index.log_return, q=0.25)
    q75 = np.quantile(index.log_return, q=0.75)
    iqr = q75 - q25    # Inter-quartile range
    lower_bound = q25 - outlier_factor*iqr
    upper_bound = q75 + outlier_factor*iqr
    boolean = (lower_bound<index.log_return) & (index.log_return<upper_bound)
    return index[boolean]

# Hyperparameter
outlier_removal = True
```

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

```
start_year = 2011
end_year = 2020

if outlier_removal:
    SPX = Remove_Outlier(SPX)
    HSI = Remove_Outlier(HSI)
    FTSE = Remove_Outlier(FTSE)

year_name = " Year: " + str(start_year) + "-" + str(end_year)
Chosen_Year = range(start_year, end_year + 1)
Chosen_SPX = SPX[SPX.Year.isin(Chosen_Year)]
Chosen_HSI = HSI[HSI.Year.isin(Chosen_Year)]
Chosen_FTSE = FTSE[FTSE.Year.isin(Chosen_Year)]
```

Programme 2.6.2: Stock market indices cleansing for one-way ANOVA and Tukey HSD in Python.

Next, we extract the data from 2011 to 2020 for the later analysis. Again, we shall first test the resulting separately whether the three resulting stock market indices follow a normal distribution using Kolmogorov Smirnov test.

```
> par(mfrow=c(1,3))
> hist(Chosen_SPX$log_return, breaks=20, xlab="SPX", cex.lab=1.5, cex.axis=1.5,
+      cex.main=1.5, main=paste0("SPX daily log-return.", year_name))
> hist(Chosen_HSI$log_return, breaks=20, xlab="HSI", cex.lab=1.5, cex.axis=1.5,
+      cex.main=1.5, main=paste0("HSI daily log-return.", year_name))
> hist(Chosen_FTSE$log_return, breaks=20, xlab="FTSE", cex.lab=1.5,cex.axis=1.5,
+      cex.main=1.5, main=paste0("FTSE daily log-return.", year_name))
>
> u1 <- Chosen_SPX$log_return
> u2 <- Chosen_HSI$log_return
> u3 <- Chosen_FTSE$log_return
> ks.test(u1, y=pnorm, mean=mean(u1), sd=sd(u1))

One-sample Kolmogorov-Smirnov test

data:  u1
D = 0.046118, p-value = 8.124e-05
alternative hypothesis: two-sided

> ks.test(u2, y=pnorm, mean=mean(u2), sd=sd(u2))

One-sample Kolmogorov-Smirnov test

data:  u2
D = 0.035725, p-value = 0.004404
alternative hypothesis: two-sided

> ks.test(u3, y=pnorm, mean=mean(u3), sd=sd(u3))

```

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

```
One-sample Kolmogorov-Smirnov test

data:  u3
D = 0.025369, p-value = 0.08945
alternative hypothesis: two-sided
```

Programme 2.6.3: Continuation of Programme 2.6.1: Normality test of stock market indices in **R**.

In the following normality test, we test whether the logarithmic return for each stock market index is normally distributed. Figure 2.6.1 shows the histograms for the daily logarithmic return for each stock market index, and apparently we observe that all three stock market indices' daily logarithmic returns acquire a bell-shaped distribution. We next adopt the Kolmogorov-Smirnov (KS) test, and the two small p-values (< 0.05) for SPX and HSI returns indicate that we reject the null hypothesis, in other words, we are 95% confident that the daily logarithmic returns of SPX and HSI are actually **NOT** normally distributed. On the other hand, since the p-value for FTSE return is greater than 0.05, we still accepts that its logarithmic return is normally distributed. Nevertheless, as long as the underlying distribution is symmetric and light-tailed, which is really the case for the presence as illustrated in Figure 2.6.1, with a finite and small enough variance, the celebrated Central Limit Theorem can warrant that the sample means to closely follow a normal distribution, even with a dosen of sample size, thanks to the Berry-Esséen bound [6, 7]. Therefore, the results of one-way ANOVA and Tukey HSD test are still robust enough to cover the present consideration.

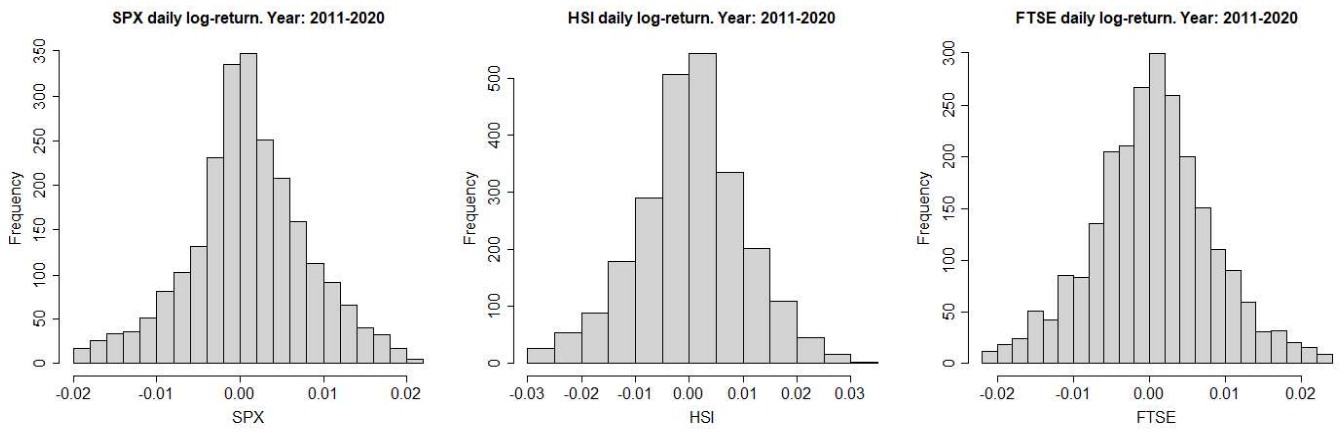


Figure 2.6.1: Histograms of the daily logarithmic return for SPX (left), HSI (middle), and FTSE (right) in **R**.

Similarly in Python:

```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20,7))
ax1.set_title("SPX daily log-return." + year_name)
ax2.set_title("HSI daily log-return." + year_name)
ax3.set_title("FTSE daily log-return." + year_name)
# Use the same binnimg method sturges as in R
ax1.hist(Chosen_SPX.log_return, bins="sturges", ec='black')
ax2.hist(Chosen_HSI.log_return, bins="sturges", ec='black')
ax3.hist(Chosen_FTSE.log_return, bins="sturges", ec='black')
fig.savefig("Index log_return histogram.png")
```

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

```
10
11 import scipy.stats as stats
12 import seaborn as sns
13 from statsmodels.stats.multicomp import MultiComparison
14
15 u1, u2, u3 = Chosen_SPX.log_return, Chosen_HSI.log_return, Chosen_FTSE.log_return
16 print(stats.kstest(u1, 'norm', args=(np.mean(u1), np.std(u1))))
17 print(stats.kstest(u2, 'norm', args=(np.mean(u2), np.std(u2))))
18 print(stats.kstest(u3, 'norm', args=(np.mean(u3), np.std(u3))))
```

```
1 KstestResult(statistic=0.046076444280916035, pvalue=7.99316826044075e-05)
2 KstestResult(statistic=0.0356928557459143, pvalue=0.004342608539308407)
3 KstestResult(statistic=0.02533315697981553, pvalue=0.08870422968828784)
```

Programme 2.6.4: Continuation of Programme 2.6.2: Normality test of stock market indices in Python.

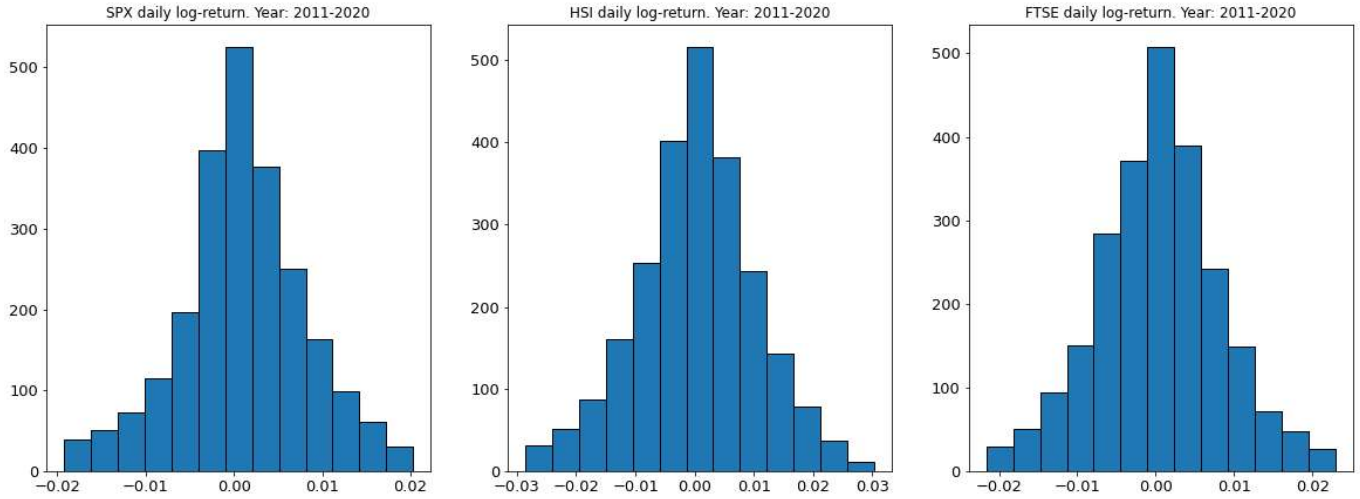


Figure 2.6.2: Histograms of the daily logarithmic return for SPX (left), HSI (middle), and FTSE (right) in Python.

```
1 > par(mfrow=c(1,3))
2 > boxplot(Chosen_SPX$log_return, Chosen_HSI$log_return, Chosen_FTSE$log_return,
3 +         names=c("SPX", "HSI", "FTSE"), xlab="Index", ylab="Daily log-return",
4 +         frame=FALSE, col=c("#00AFBB", "#E7B800", "#FC4E07"), boxwex=0.75,
5 +         main=paste0("Boxplot.", year_name))
6 >
7 > Chosen_SPX$Index <- "SPX"
8 > Chosen_HSI$Index <- "HSI"
9 > Chosen_FTSE$Index <- "FTSE"
10 > AllIndex <- rbind(Chosen_SPX, Chosen_HSI, Chosen_FTSE)
11 > AllIndex$Index <- factor(AllIndex$Index, c("SPX", "HSI", "FTSE"))
12 >
13 > library("gplots")
14 > plotmeans(log_return~Index, data=AllIndex, ylab="Daily log-return",
15 +           xlab="Index", main=paste0("Mean Plot with 95% CI.", year_name))
```

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

```
>
> anova.test <- aov(log_return~Index, data=AllIndex)
> summary(anova.test)
               Df  Sum Sq   Mean Sq F value  Pr(>F)
Index           2  0.0006 0.0003206   4.529  0.0108 *
Residuals    7185  0.5086 0.0000708
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
> # Reject the null hypothesis?
> summary(anova.test)[[1]][[1, "Pr(>F)"]] < 0.05
[1] TRUE
>
> tukey.test <- TukeyHSD(anova.test)
> tukey.test
Tukey multiple comparisons of means
95% family-wise confidence level

Fit: aov(formula = log_return ~ Index, data = AllIndex)

$Index
                   diff           lwr           upr     p adj
HSI-SPX   -6.691266e-04 -0.0012400420 -9.821133e-05 0.0165817
FTSE-SPX  -5.941240e-04 -0.0011640375 -2.421048e-05 0.0386687
FTSE-HSI   7.500264e-05 -0.0004937116  6.437169e-04 0.9486808

> plot(tukey.test)
```

Programme 2.6.5: Continuation of Programme 2.6.3: One-way ANOVA and Tukey HSD with three stock market indices in **R**.

Here the `aov()` function is used for the one-way ANOVA. For the categorical feature variable, we have the name of the stock market index, namely `SPX`, `HSI`, and `FTSE`; for the label, we have the daily logarithmic returns. The small p-value (< 0.5) indicates the we reject the null hypothesis, *i.e.* the corresponding population means $\mu_{\text{SPX}}, \mu_{\text{FTSE}}, \mu_{\text{HSI}}$ are not all equal. However, with ANOVA test only, we are not able to tell how these population means for these three stock market indices are different from each other. We therefore adopt the Tukey HSD test by using the `TukeyHSD()` function. Since there are 3 stock market indices, there are $\binom{3}{2} = 3$ possible pairs. The corresponding small p-values for `HSI` against `SPX`, `FTSE` against `SPX` indicate that the population means for `HSI` and `SPX` are not the same, and it is also the case between `FTSE` and `SPX`. However, for `FTSE` against `HSI`, the larger p-value ($0.9486808 > 0.5$) indicate that it is fine to assume the equality for $\mu_{\text{FTSE}} = \mu_{HSI}$.

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

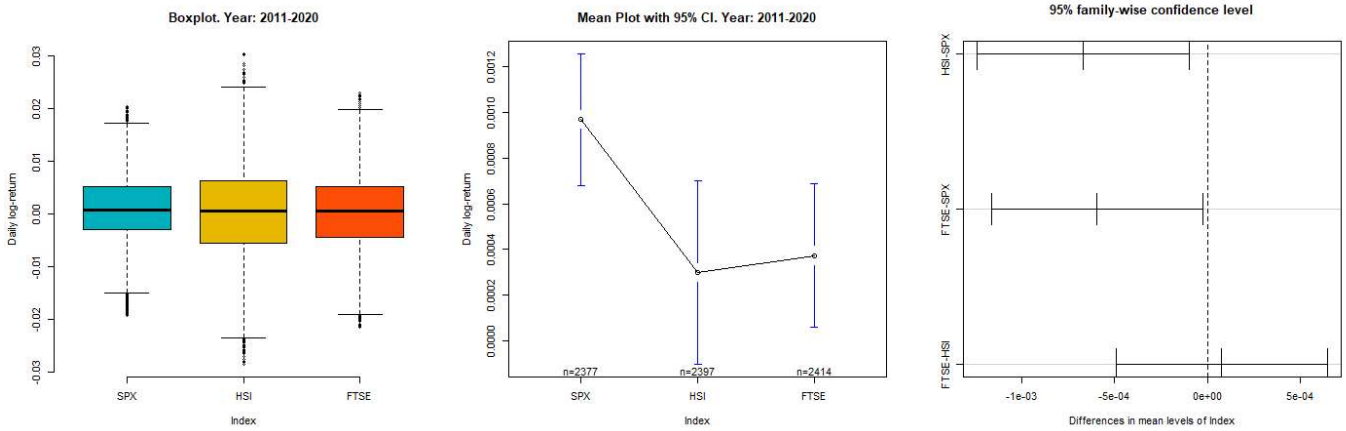


Figure 2.6.3: Boxplot (left), meanplot (middle), Tukey HSD plot (right) for three stock market indices in **R**.

From the middle plot of Figure 2.6.3, the mean of the daily logarithmic return for `SPX` is significantly higher than those of `HSI` and `FTSE`, which is in agreement with the result obtained from the Tukey HSD test.

In Python,

```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(22,7))
plt.subplots_adjust(wspace=0.4)      # horizontal spacing between plots
ax1.set_title("Boxplot." + year_name, fontsize=20)
ax1.set_xlabel("Location", fontsize=15)
ax1.set_ylabel("Daily log-return", fontsize=15)
ax1.boxplot([Chosen_SPX.log_return, Chosen_HSI.log_return, Chosen_FTSE.log_return],
            widths=0.7, labels=["SPX", "HSI", "FTSE"])

Chosen_SPX.loc[:, "Index"] = "SPX"
Chosen_HSI.loc[:, "Index"] = "HSI"
Chosen_FTSE.loc[:, "Index"] = "FTSE"

AllIndex = Chosen_SPX.append([Chosen_HSI, Chosen_FTSE])

sns.pointplot(x='Index', y="log_return", data=AllIndex, ci=95, ax=ax2)
ax2.set_title("Mean Plot with 95% CI." + year_name, fontsize=20)
ax2.set_xlabel("Index", fontsize=15)
ax2.set_ylabel("Daily log-return", fontsize=15)

F, p = stats.f_oneway(Chosen_SPX.log_return, Chosen_HSI.log_return,
                      Chosen_FTSE.log_return)
# Reject the null hypothesis?
print(p < 0.05)

# ANOVA
comp = MultiComparison(data=AllIndex.log_return, groups=AllIndex.Index,
                       group_order=["SPX", "HSI", "FTSE"])
TurkeyHSD_result = comp.tukeyhsd()
print(TurkeyHSD_result)

# Tukey
TurkeyHSD_result.plot_simultaneous(figsize=(22, 7), ax=ax3)
ax3.title.set_fontsize(20)
ax3.set_xlabel("Daily log-return", fontsize=15)
ax3.set_ylabel("Index", fontsize=15)
fig.savefig("Boxplot, Meanplot, Tukey HSD.png")
```

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

```
True
Multiple Comparison of Means - Tukey HSD, FWER=0.05
===================================================
group1 group2 meandiff p-adj   lower   upper  reject
---------------------------------------------------
   SPX    HSI   -0.0007 0.0166 -0.0012 -0.0001   True
   SPX   FTSE   -0.0006 0.0387 -0.0012    -0.0   True
   HSI   FTSE    0.0001    0.9 -0.0005  0.0006  False
---------------------------------------------------
```

Programme 2.6.6: Continuation of Programme 2.6.4: One-way ANOVA and Tukey HSD with three stock market indices in **R**.

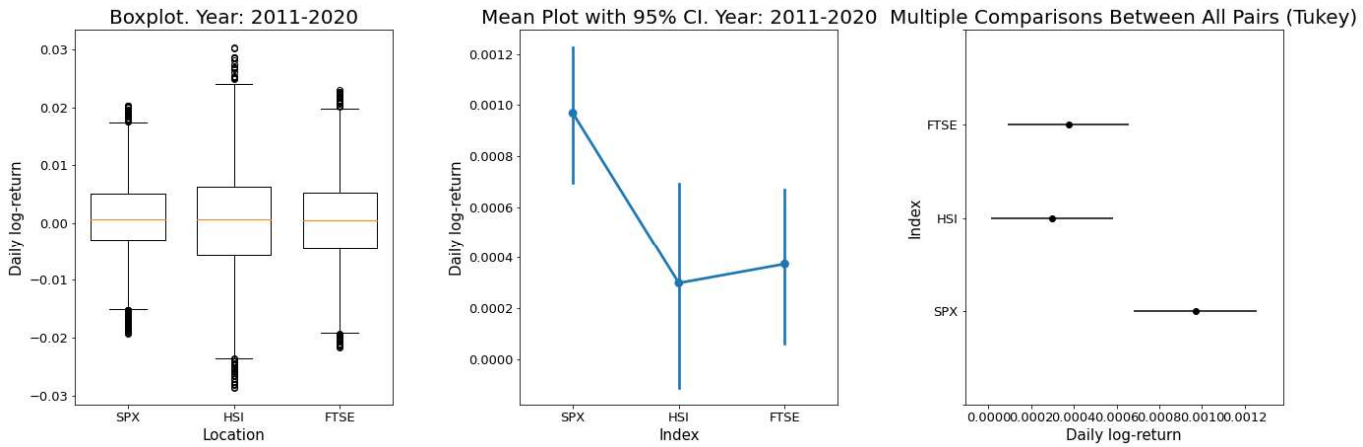


Figure 2.6.4: Boxplot (left), meanplot (middle), Tukey HSD plot (right) for three stock market indices in Python.

To replicate the result of `tukey.test`, from (2.6.2), we first compute the individual sample means, the individual sum of square errors $\mathrm{SSE}_{\mathrm{SPX}}, \mathrm{SSE}_{\mathrm{FTSE}}, \mathrm{SSE}_{\mathrm{HSI}}$, the degrees of freedom ($\ell = N_{\mathrm{SPX}} + N_{\mathrm{FTSE}} + N_{\mathrm{HSI}} - 3 + 1 - 1 = 7185$), and the harmonic mean number of each stock market index $\bar{n}$. `Within_group_MSE` is the within group mean square error $\mathrm{MSE}_{\mathrm{within}} = (\mathrm{SSE}_{\mathrm{SPX}} + \mathrm{SSE}_{\mathrm{FTSE}} + \mathrm{SSE}_{\mathrm{HSI}})/\ell$. The former SSE's and the latter MSE are obtained by `Residual`, `df`, and `Mean Sq` respectively via the one-way ANOVA from Programme 2.6.5. Moreover, the `ptukey()` function is used to compute the probability that the HSD is smaller than the critical value indicated by the quantile parameter `q`. Since, Tukey HSD is a one-sided test, the p-value is obtained by $p = 1 - \mathbb{P}(q_{\mathrm{tukey,ij}} \leq q)$.

```
> mu_SPX <- mean(Chosen_SPX$log_return)
> mu_HSI <- mean(Chosen_HSI$log_return)
> mu_FTSE <- mean(Chosen_FTSE$log_return)
>
> SSE_SPX <- sum((Chosen_SPX$log_return - mu_SPX)^2)
> SSE_HSI <- sum((Chosen_HSI$log_return - mu_HSI)^2)
> SSE_FTSE <- sum((Chosen_FTSE$log_return - mu_FTSE)^2)
```

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

```
>
> n_SPX <- length(Chosen_SPX$log_return)
> n_HSI <- length(Chosen_HSI$log_return)
> n_FTSE <- length(Chosen_FTSE$log_return)
>
> df <- length(AllIndex$log_return) - 3 + 1 - 1  # df: degrees of freedom
> df
[1] 7185
>
> Within_group_MSE <- (SSE_SPX + SSE_HSI + SSE_FTSE) / df
> Within_group_MSE
[1] 7.079026e-05
>
> # Harmonic mean used for the two sample sizes
> SPX_HSI_SE_ANOVA <- sqrt(Within_group_MSE / (2/(1/n_SPX + 1/n_HSI)))
> FTSE_SPX_SE_ANOVA <- sqrt(Within_group_MSE / (2/(1/n_FTSE + 1/n_SPX)))
> FTSE_HSI_SE_ANOVA <- sqrt(Within_group_MSE / (2/(1/n_FTSE + 1/n_HSI)))
>
> SPX_vs_HSI <- abs(mu_SPX - mu_HSI) / SPX_HSI_SE_ANOVA
> FTSE_vs_SPX <- abs(mu_FTSE - mu_SPX) / FTSE_SPX_SE_ANOVA
> FTSE_vs_HSI <- abs(mu_FTSE - mu_HSI) / FTSE_HSI_SE_ANOVA
>
> 1 - ptukey(q=SPX_vs_HSI, nmeans=3, df=df)
[1] 0.01658168
> 1 - ptukey(q=FTSE_vs_SPX, nmeans=3, df=df)
[1] 0.03866867
> 1 - ptukey(q=FTSE_vs_HSI, nmeans=3, df=df)
[1] 0.9486808
```

Programme 2.6.7: Continuation of Programme 2.6.5: Replication of the Tukey HSD test in **R**.

Similarly, we have the similar coding via Python:

```
mu_SPX = np.mean(Chosen_SPX.log_return)
mu_HSI = np.mean(Chosen_HSI.log_return)
mu_FTSE = np.mean(Chosen_FTSE.log_return)

SSE_SPX = np.sum((Chosen_SPX.log_return - mu_SPX)**2)
SSE_HSI = np.sum((Chosen_HSI.log_return - mu_HSI)**2)
SSE_FTSE = np.sum((Chosen_FTSE.log_return - mu_FTSE)**2)

n_SPX = len(Chosen_SPX.log_return)
n_HSI = len(Chosen_HSI.log_return)
n_FTSE = len(Chosen_FTSE.log_return)

df = len(AllIndex.log_return) - 3 + 1 - 1   # df: degrees of freedom
print(df)

Within_group_MSE = (SSE_SPX + SSE_HSI + SSE_FTSE) / df
print(Within_group_MSE)
```

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

```
18
19 # Harmonic mean used for the two sample sizes
20 SPX_HSI_SE_ANOVA = np.sqrt(Within_group_MSE / (2/(1/n_SPX + 1/n_HSI)))
21 FTSE_SPX_SE_ANOVA = np.sqrt(Within_group_MSE / (2/(1/n_FTSE + 1/n_SPX)))
22 FTSE_HSI_SE_ANOVA = np.sqrt(Within_group_MSE / (2/(1/n_FTSE + 1/n_HSI)))
23
24 SPX_vs_HSI = np.abs(mu_SPX - mu_HSI) / SPX_HSI_SE_ANOVA
25 FTSE_vs_SPX = np.abs(mu_FTSE - mu_SPX) / FTSE_SPX_SE_ANOVA
26 FTSE_vs_HSI = np.abs(mu_FTSE - mu_HSI) / FTSE_HSI_SE_ANOVA
27
28 from statsmodels.stats.libqsturng import psturng
29
30 print(psturng(q=SPX_vs_HSI, r=3, v=df)[0])
31 print(psturng(q=FTSE_vs_SPX, r=3, v=df)[0])
32 print(psturng(q=FTSE_vs_HSI, r=3, v=df))
```

```
1 7185
2 7.079026465287093e-05
3 0.016587032757677256
4 0.038681202320859254
5 0.9
```

Programme 2.6.8: Continuation of Programme 2.6.6: Replication of the Tukey HSD test in Python.

All rights reserved. Do not distribute without permission from the authors, Kaiser Fan and Phillip Yam.

BIBLIOGRAPHY

[1] Black, F., and Scholes, M. (1972). The Pricing of Options and Corporate Liabilities. *The Journal of Political Economy*. Vol. 81, No. 3. 637-654.

[2] Shapiro, S. S., and Wilk, M.B. (1965). An analysis of variance test for normality (complete samples). *Biometrika*. Vol. 52, issue 3-4. 591-611.

[3] Kolmogorov A (1933). "Sulla determinazione empirica di una legge di distribuzione". G. Ist. Ital. Attuari. 4: 83–91

[4] Timm, N. H. (2002). Applied multivariate analysis. Springer.

[5] Tukey, J. W. (1949). Comparing individual means in the analysis of variance. Biometrics, 99-114.

[6] Berry, A. C. (1941). The accuracy of the Gaussian approximation to the sum of independent variates. Transactions of the american mathematical society, 49(1), 122-136.

[7] Esseen, C.-G (1942). On the Liapunoff limit of error in the theory of probability. Arkiv för Matematik, Astronomi och Fysik. A28: 1–19. ISSN 0365-4133.