Polycopié de la ressources *R5.VCOD.08 Renforcement informatique* dans le cadre de la formation *SD* Grenoble.

Département SD IUT2 de Grenoble

Cette ressource s'appuie sur les ressources :

- R1.02 - Bases de données relationnelles 1
- R1.03 - Bases de la programmation 1
- R1.04 - Statistique Descriptive 1 : Polycopié disponible à l'adresse https://membres-ljk.imag.fr/Vincent.Brault/Cours/Cours_1A.pdf
- R2.03 - Bases de la programmation 2
- R2.08 - Statistique inférentielle
- R3.06 - Tests d'hypothèses pour l'analyse bi-variée
- R4.02 - Méthodes factorielles
- R5.01 - Bases de données NoSQL

Table des matières

Avant propos

Le but de cette ressource est de permettre aux étudiant·e·s de (re)voir un certain nombre d'outils d'analyse de données en utilisant le langage Python. Par conséquent, il n'y aura que très peu de nouvelles notions ce qui permettra à chaque élève de renforcer ses connaissances théoriques tout en les mettant en application sur de nouveaux types de jeux de données. En plus des ressources du BUT mises sur la page de garde, ce polycopié s'appuie sur les références suivantes :

- *Analyse de données avec R* de Husson et al. (2016).
- *Python pour les SHS. Introduction à la programmation pour le traitement de données* de Schultz et Bussonnier (2021).

Nous vous encourageons à les consulter si vous voulez plus de précisions ou une vision différente des notions présentées.

De plus, je remercie les lectrices et les lecteurs attentif·ve·s pour les corrections de coquilles. À ce titre, merci donc à Jérémy Macé et Célia Tropel (BUT3 - 2023/2024).

Chapitre 1

Automatisation de scripts

"Ordinateur : moyen conçu pour accélérer et automatiser les erreurs."
Anonyme

Dans ce chapitre, nous aborderons le principe d'automatisation des procédures. Dans son travail, les data-analystes et les data-scientistes sont amené·e·s à faire des tâches répétitives et les automatiser permet de gagner un temps parfois précieux. Nous commencerons par rappeler le principe d'une fonction puis comment faire une automatisation dans Python directement et nous conclurons par une généralisation en utilisant des outils comme `taskschd` par exemple.

Ce chapitre reprend une grande partie des notions abordées dans les ressources *R1.03 - Bases de la programmation 1* et *R2.03 - Bases de la programmation 2*.

1.1 Fonctions en python

Commençons par quelques définitions :

Définition 1 (Fonction informatique)
En informatique, une **fonction** est un ensemble d'instructions qui seront réalisées et qui affichera une liste de résultat ou transformera des *éléments extérieurs* à la fonction. Comme en mathématique, elle peut prendre une ou plusieurs information(s) en entrée.

1.1.1 Fonctions basiques

Dans cette partie, nous faisons un rappel sur la base des fonctions telle qu'elles ont été vues en *R2.03 - Bases de la programmation 2*.

Codage en Python

En Python, une fonction se définie à l'aide de `def`, il faut ensuite mettre le nom de la fonction ainsi que les variables utilisées (s'il y en a) et la valeur qui sortira. Par exemple, la fonction suivante :

```
def Bonjour():
    print("Bonjour, ravi que tu sois avec nous")
```

renvoie toujours le texte "*Bonjour, ravi que tu sois avec nous*" quand on l'utilise :

```
Bonjour()
```

On peut personnaliser ce texte en mettant un nom comme entrée :

```
def Bonjour(Prénom):
    print("Bonjour "+Prénom+", ravi que tu sois avec nous")
```

renvoie un texte incluant le nom de la personne. Ainsi, la commande

```
Bonjour("Zelda")
```

renverra le texte "*Bonjour Zelda, ravi que tu sois avec nous*". Si on essaie d'utiliser la commande sans mettre de prénom :

```
Bonjour()
```

on obtient l'erreur :

```
TypeError: Bonjour() missing 1 required positional argument: 'Prénom'
```

Afin de corriger cela, on peut mettre une valeur par défaut mais il serait vexant de donner un mauvais prénom à quelqu'un. Mais si on met `""` :

```
Bonjour("")
```

on obtient un espace après *Bonjour*. Le choix fait est donc de mettre `None` et de décomposer suivant la valeur du prénom :

```
def Bonjour(Prénom=None):
    if (Prénom is None):
        print("Bonjour, ravi que tu sois avec nous")
    else:
        print("Bonjour "+Prénom+", ravi que tu sois avec nous")
```

Ainsi, notre fonction semble enfin prête.

Remarque

Notons que pour une simple fonction, plusieurs réflexions ont été nécessaires. Il est important de bien prendre du recul sur ce qu'on fait afin d'anticiper un maximum d'erreurs.

Dans l'exemple précédent, la sortie était un texte. Généralement, nous avons besoin d'un résultat qui pourra être repris plus tard.

Codage en Python

Par exemple, nous pouvons faire une fonction qui, étant donné un texte, renvoie le nombre d'occurrences de chaque lettre. Pour ce faire, nous utilisons l'attribut `lower` permettant de transformer toutes les lettres d'un texte en minuscule.

```
import string
import numpy as np
import pandas as pd

def freq_lettre(Texte):
    # On transforme en minuscule
    Texte = Texte.lower()
    Alphabet = list(string.ascii_lowercase)
    # On calcule le nombre d'occurrences par lettre
    Somme = [Texte.count(Alphabet[it]) for it in range(len(Alphabet))]
    return pd.DataFrame({"Lettre": Alphabet,
                         "Freq": Somme/np.sum(Somme)})
```

Ainsi, nous pouvons voir que

```
freq_lettre("Mon pauvre zébu ankylosé choque deux fois ton wagon jaune")
```

possède toutes les lettres tandis que

```
freq_lettre("L'humain avait un hot−dog qui avait un parfum divin." +
            "L'animal noir sauta sur l'humain" +
            "lui arrachant ainsi l'hot−dog." +
            "Mais un lapin biscornu sauta" +
            "du saucisson faisant fuir l'humain." +
            "L'animal noir poussa un long cri aigu qui arracha" +
            "l'animal biscornu du sol mouvant.")
```

ne possède pas les lettres "e", "j", "k", "w", "x", "y" ou "z". En utilisant ce qu'on a vu dans le chapitre **??**, nous pouvons coupler avec une représentation en diagramme empilé :

```
def Barplot_texte(Texte):
    temp = freq_lettre(Texte)
    empile = pd.DataFrame({temp.iloc[it, 0]: [temp.iloc[it, 1]]
                           for it in range(temp.shape[0])})
    empile.plot.bar(stacked=True, figsize=(11, 10))
```

ou une comparaison de deux textes :

```
def Comp_texte(Texte1, Texte2):
    temp1 = freq_lettre(Texte1)
    temp2 = freq_lettre(Texte2)
    empile = pd.DataFrame({temp1.iloc[it, 0]: [temp1.iloc[it, 1],
                                               temp2.iloc[it, 1]]
                           for it in range(temp1.shape[0])})
    empile.plot.bar(stacked=True,figsize=(11, 10))
```

Nous avons représenté les résultats pour les deux textes étudiés dans la figure 1.1.

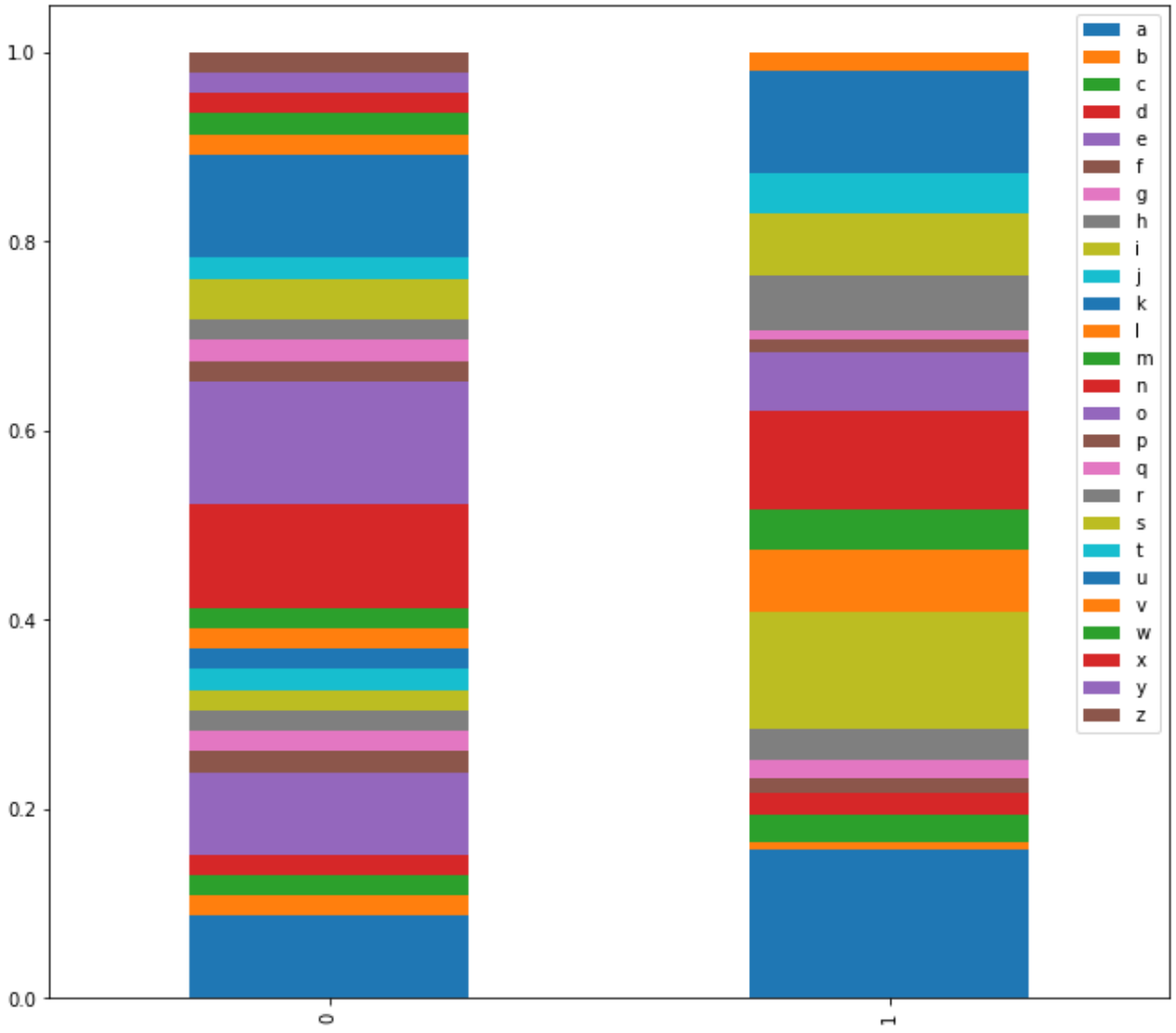

FIGURE 1.1 – Diagrammes empilés des fréquences de lettres dans les deux textes utilisés dans l'exemple.

1.1.2 Vérifications et sécurités

Les sécurités sont importantes pour éviter les erreurs qui bloqueraient tout ou une partie du système. Cette sécurité est importante pour anticiper des erreurs qui pourraient faire bugguer le programme alors qu'une solution pourrait être donnée ou, pire, un comportement qui ne serait pas cohérent avec le résultat attendu.

Codage en Python

Afin de calculer les n premiers entiers naturels, on pourrait créer un vecteur de valeurs allant de 1 à n et faire la somme :

```
def n_entier(n):
    return np.sum(range(n+1))
```

Dans ce cas, si nous tapons :

```
n_entier(10)
```

nous obtenons bien 55 mais si nous mettons

```
n_entier("Zelda")
```

une erreur apparaît car c'est du texte et que la fonction `range` ne peut pas prendre en compte du texte. Toutefois, deux problèmes ne sont pas pris en compte :

```
n_entier(−10)
```

renvoie 0 alors qu'il devrait y avoir une erreur et

```
n_entier(10.)
```

renvoie une erreur car c'est un `float` alors qu'il devrait renvoyer 55 aussi. Une façon de contourner ces deux problèmes est de mettre la sécurité suivante :

```
def n_entier(n):
    # On vérifie qu'on a un numérique
    if (isinstance(n, (int, float))):
        # si oui, on vérifie que c'est un entier positif
        if (((n % 1) != 0) | (n < 0)):
            raise ValueError("n doit être un entier positif")
    else:
        # si non, on renvoie une erreur
        raise ValueError("n doit être un entier positif")
    # Si c'est un float mais qui est entier positif, on le met en int
    if (isinstance(n, float)):
        n = int(n)
    return np.sum(range(n+1))
```

Toutefois, pour gagner du temps, on peut utiliser une formule de maths permettant de calculer la somme des n premiers entiers naturels :

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Dans ce cas, nous avons le code :

```
def n_entier_formule(n):
    return n*(n+1)/2
```

et, cette fois, elle fonctionne avec `10.`. Néanmoins, elle fonctionne aussi avec 10.5 qui renverra `60.375` qui n'est pas un résultat correct (mais un programme utilisant cette valeur ne verrait pas forcément le problème). Ainsi, la vérification précédente est d'autant plus importante :

```
def n_entier_formule(n):
    if (((n % 1) != 0) | (n < 0)):
        raise ValueError("n doit être un entier positif")
    return n*(n+1)/2
```

La (mauvaise) utilisation de la statistique à travers les âges

Lors du décollage de la fusée Ariane 5, la vitesse *horizontale* de la fusée (c'est-à-dire la vitesse à laquelle la fusée quitte la verticale par rapport à son point de lancement) a dépassé la limite inscrite dans le logiciel du calculateur d'après les rapporteurs de la commission suite à l'explosion de la fusée. Ceci a entraîné une succession d'événements qui ont abouti à la destruction de la fusée. [1]

Il est donc nécessaire de faire des tests à son code, même les plus étranges car nous ne savons jamais comment vont être utilisés nos fonctions.

Remarque

Pour donner une idée de où placer la barre, j'aime beaucoup l'extrait du livre *Anomalie* de Le Tellier (2020) que je remets ici :

> Rien ne leur échappe ; le Pentagone leur aurait-il demandé de présenter toutes les réponses possibles à un pile ou face qu'ils en auraient envisagé trois : pile, face, et le cas rare où la pièce déciderait de s'immobiliser sur sa tranche, à la verticale. Mais dix jours après la remise du rapport, en avril 2002, le DoD leur renvoie, avec une question inscrite au feutre rouge : "Et si nous sommes confrontés à un cas n'obéissant à aucune situation étudiée ?"
> Tina hausse les yeux au ciel : va pour l'hypothèse où la pièce lancée resterait suspendue en l'air.

1.1.3 Optimisation d'un code

Comme vous le verrez dans les ressources *Ressource R6.01 : Big Data : enjeux, stockage et extraction* et *Ressource R6.02 : Méthodes statistiques pour le Big Data*, à partir d'un moment, le codage brut est trop lent. Il est important de comprendre le plus tôt possible comment optimiser ses codes en comparant différentes méthodes. Pour ce faire, vous pouvez, par exemple, stocker l'heure au début de la procédure et le comparer à l'heure à la fin de la procédure à l'aide de la fonction `time` de la bibliothèque `time`.

Codage en Python

Pour ce faire, il est important de prendre l'heure de début :

```
import time

heure_deb = time.time()
```

L'idée est de prendre l'heure avant, exécuter la procédure et prendre l'heure après :

```
heure_deb = time.time()
print(n_entier(10**8))
heure_fin = time.time()
heure_fin-heure_deb
```

Le temps obtenu au final est donné sous forme de secondes. Au passage, si on souhaite obtenir la date au format classique, nous pouvons utiliser la bibliothèque `datetime`

```
from datetime import datetime
```

1. Voir par exemple les articles de Libération (https://www.liberation.fr/futurs/1996/07/24/l-explosion-d-ariane-5-c-est-la-faute-au-logicielnous-sommes-tous-coupables-a-affirme-le-directeur-g_176415/) ou l'humanité (https://www.humanite.fr/-/-/les-raisons-de-lechec-de-la-fusee-dariane-5-01) pour plus de détails.

```
date = datetime.fromtimestamp(heure_deb)
date.strftime("%Y-%m-%d %H:%M:%S")
```

À l'opposé, on peut transformer une date en `timestamp` à l'aide de la fonction `strptime` de la bibliothèque `datetime` en

```
datetime.strptime("2023-12-31 23:59:59", "%Y-%m-%d %H:%M:%S").timestamp()
```

Néanmoins, cette façon de faire n'est pas très précise et nous recommandons plutôt la bibliothèque `timeit`.

Codage en Python

Il existe plusieurs façons de calculer le temps, je vous en proposer une consistant à créer deux fonctions avec chacune des procédures :

```
# Avec liste
def time_entier_formule():
    for n in range(1000):
        n_entier_formule(n)
    return 0
# Avec vecteurs
def time_entier_liste():
    for n in range(1000):
        n_entier(n)
    return 0
```

puis d'utiliser la fonction `Timer` de la bibliothèque `timeit` pour préciser la procédure qui sera testée (si vous ne voulez pas utiliser les fonctions précédentes, vous pouvez écrire en texte la procédure directement) :

```
import timeit

test_formule = timeit.Timer(time_entier_formule)
test_liste = timeit.Timer(time_entier_liste)
```

Enfin, on utilise la fonction `timeit` de la bibliothèque `timeit` qui calculera le temps moyen de n lancers de la procédure ou la fonction `repeat` de la bibliothèque `timeit` qui fera plusieurs fois le calcul :

```
# 20 calculs de moyenne sur 5 lancers
temps_execution_formule = test_formule.repeat(repeat=20, number=5)
temps_execution_vecteur = test_liste.repeat(repeat=20, number=5)
```

De plus, si nous souhaitons comparer les temps, nous pouvons les afficher sous forme de boxplot par exemple (voir la figure ‘1.2) :

```
from matplotlib import pyplot as plt

fig = plt.figure(figsize=(9, 3))
ax = plt.subplot(1, 2, 1)
ax.boxplot([temps_execution_formule, temps_execution_vecteur])
ax.set_title('Echelle normale')
ax = plt.subplot(1, 2, 2)
ax.boxplot([temps_execution_formule, temps_execution_vecteur])
ax.set_yscale('log')
ax.set_title('Echelle logarithmique')
plt.show()
```

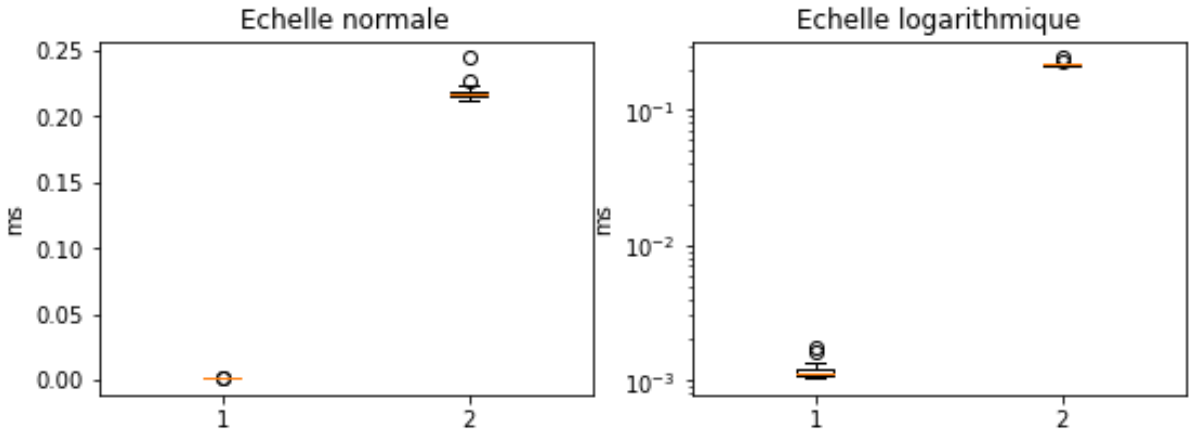

FIGURE 1.2 – Boxplots des temps mis pour calculer la somme des n premiers entiers avec n allant de 1 à 1000 avec une échelle normale (à gauche) et une échelle logarithmique (à droite) : pour chaque graphe, la méthode avec la formule est à gauche et celle en générant le vecteur de tous les entiers en faisant la somme est à droite.

Exemple fil rouge

La différence de temps de l'exemple est assez importante car dans un cas, on doit générer toutes les valeurs puis les additionner ; on dit qu'on est d'une complexité linéaire en fonction de la valeur n, notée $\mathcal{O}(n)$. A l'opposé, dans l'autre fonction, on doit juste faire une addition, une multiplication et une division donc le même nombre d'opérations qu'importe la valeur de n ; on parle de complexité constante notée $\mathcal{O}(1)$. Dans un cas, nous sommes de l'ordre du millième de milliseconde pour faire les 1000 calculs tandis que, dans l'autre, nous sommes de l'ordre du dixième de milliseconde.

1.2 Planification d'une tâche

Dans cette partie, nous allons voir plusieurs façons de planifier une tâche.

1.2.1 Planification avec Python ouvert

Une des façons de procéder est d'utiliser la bibliothèque `schedule`. Voici un exemple de code :

Codage en Python

Pour commencer, il faut écrire la tâche qui sera exécutée :

```
def Renf():
    print("Renforcement informatique 2, c'est trop cool")
```

Ensuite, il faut définir le planificateur ainsi que le temps d'attente entre deux tâches :

```
import schedule
planificateur = schedule.Scheduler()
# On va planifier toutes les 5 secondes
planificateur.every(5).seconds.do(Renf)
```

Ainsi, à chaque fois que le planificateur sera appelé, il vérifiera si 5 secondes ont été écoulées ou pas. Par exemple, nous pouvons faire une boucle `for` en mettant des temps de pause :

```
import time
# Boucle d'exécution
for it in range(50):
    # Lancement du planificateur
    planificateur.run_pending()
    # Attente avant nouveau lancement
    time.sleep(5)
```

La procédure va s'exécuter 49 fois.

Attention au piège

Si on met un temps réduit par rapport au temps d'exécution, le planificateur attendra avant de faire son action. Par exemple, le code suivant :

```
# Boucle d'exécution réduite
for it in range(50):
    # Lancement du planificateur
    planificateur.run_pending()
    # Attente avant nouveau lancement
    time.sleep(1)
```

n'effectueras la tâche que 10 fois car le planificateur sera appelé plusieurs fois sans pouvoir être exécuté.

De plus, il est possible de programmer les lancements à des heures spécifiques [2]. Une des difficultés sera de faire communiquer deux tâches entre elles car elles ne renvoient pas de résultats. Une solution proposée sur internet est d'utiliser les bibliothèques `threading` et `queue` mais elles ont fait crasher mon PC donc je recommande de sauvegarder avant. Une autre solution consiste à passer par un fichier intermédiaire.

Codage en Python

Pour ce faire, il faut une tâche qui crée ou modifie un fichier et une autre qui va récupérer les information dans ce ficher. Par exemple, on peut aller chercher les informations des actualités d'une API d'information :

```
import requests


def rec_actu():
    # Récupération
    data = requests.get(url)
    # Vérification que ça a fonctionné
    if (data.status_code == 200):
        # Récupération du dernier titre
        titre = data.json()['articles'][0]['title']
        # On écrit dans un fichier
        with open("titre.txt", 'w') as fichier:
            fichier.write(titre)
    else:
        print("Echec de la récupération")
```

Ici, nous avons récupéré les données et vérifié qu'elles fonctionnaient (car j'utilise un API demandant une clef) puis j'ai récupéré les informations sous forme de JSON (voir la ressource *R5.01 : Bases de données NoSQL*) et j'ai été cherché le titre pour le stocker dans un fichier. Nous pouvons faire une autre tâche qui vient lire le titre et l'afficher s'il a changé depuis la dernière fois :

```
import os


def news():
    if os.path.exists("ancien.txt"):
        with open("ancien.txt", 'r') as fichier:
            ancien = fichier.read()
    else:
        ancien = ""
    with open("titre.txt", 'r') as fichier:
        titre = fichier.read()
    if (ancien != titre):
```

2. Voir par exemple https://schedule.readthedocs.io/en/stable/examples.html pour un aide mémoire

```
        print("\nDernière nouvelle\n" + titre + "\n")
        with open("ancien.txt", 'w') as fichier:
            fichier.write(titre)
```

Et enfin, nous pouvons gérer la génération des deux :

```
from tqdm import tqdm

# Créez un planificateur
planificateur1 = schedule.Scheduler()
planificateur2 = schedule.Scheduler()
# Planifiez la fonction às'exécuter toutes les 5 secondes
planificateur1.every(30).seconds.do(rec_actu)
planificateur2.every(1).minutes.do(news)

for it in tqdm (range(50)):
    # Lancement du planificateur
    planificateur1.run_pending()
    planificateur2.run_pending()
    # Attente avant nouveau lancement
    time.sleep(30)
```

Une autre solution est d'utiliser une variable *extérieure* et d'ajouter *global* dans notre fonction. De plus, nous pouvons utiliser le même planificateur pour plusieurs opérations.

Codage en Python

```
ancien = ""

def rec_actu():
    global titre
    # Récupération
    data = requests.get(url)
    # Vérification que ça a fonctionné
    if (data.status_code == 200):
        # Récupération du dernier titre
        titre = data.json()['articles'][0]['title']
    else:
        print("Echec de la récupération")
        titre =""


def news():
    global ancien, titre
    if (ancien != titre):
        print("\nDernière nouvelle\n" + titre + "\n")
        ancien = titre


# Créez un planificateur
planificateur = schedule.Scheduler()
# Planifiez la fonction às'exécuter toutes les 5 secondes
planificateur.every(30).seconds.do(rec_actu)
planificateur.every(1).minutes.do(news)

for it in tqdm (range(50)):
    # Lancement du planificateur
```

```
    planificateur.run_pending()
    # Attente avant nouveau lancement
    time.sleep(30)
```

1.3 Planification automatique sur Windows avec le planificateur de tâches

Pour créer une tâche automatique sur Windows, il faut procéder en plusieurs temps :

1. Créer un script qui s'exécute tout seul (donc, par exemple, qui renvoie le résultat dans un fichier). Vérifier que ça fonctionne en fermant tout et double cliquant sur votre script.
2. Il faut ensuite planifier la tâche, pour cela, il faut ouvrir le **planificateur de tâches** en faisant au choix :
   - Win+R sur votre clavier.
   - En allant chercher **invite commande**.
3. Dans la fenêtre, taper **taskschd.msc**.
4. Le planificateur s'ouvre avec toutes les tâches qui s'exécutent déjà suite à l'installation de différents logiciels.
5. Créer une nouvelle tâche dans le menu à droite :
   (a) Dans l'onglet *Général*, mettre un nom et une description claire et précise (pour éviter de se demander ce que fait la tâche quand on ouvre à nouveau trois ans après)
   (b) Dans l'onglet *Déclencheurs*, préciser les conditions qui déclencheront la tâche (tous tes heures, tous les jours, à chaque fois que l'ordinateur s'allume...)
   (c) Dans l'onglet *Action*, préciser l'emplacement de votre script.
6. Sauvegarder et lancer la tâche.

Remarque

Il est possible que Windows vous demande de confirmer le logiciel utilisé à chaque fois qu'il lance la tâche. Si, comme moi, votre chemin comporte des points, il est possible que windows interprète mal le logiciel à utiliser. Il vaut donc mieux éviter ceci.

Attention au piège

Il est possible que le python que vous utilisez pour créer vos scripts (par exemple avec Anaconda) et celui utilisé par Windows ne soit pas le même. Dans ce cas, le principal problème sera la mise à jour des packages à faire des deux côtés.

Chapitre 2

Modèles linéaires

"La nature est écrite en langage mathématique."

Galilée, physicien et astronome italien (1564-1642)

Dans cette partie, nous abordons le principe de modèle linéaire et on se place ainsi dans le prolongement des ressources *R1.04 - Statistique Descriptive 1*, *R2.05 - Statistique descriptive 2* et *R3.06 - Tests d'hypothèses pour l'analyse bi-variée*. Cette partie n'est pas abordée dans le parcours *VCOD* et la théorie présentée dans la ressource *R4.EMS.08 : Modèle linéaire* n'est donc pas à connaître. En revanche, la connaissance du moment où une modélisation linéaire est judicieuse et comment l'utiliser est à connaître.

2.1 Théorie

Commençons par définir un modèle linéaire.

Définitions 2 (Modèle linéaire)

Étant donnés n observations $(Y_1, \dots, Y_n)$ et np covariables $(x_{i,1}, \dots, x_{i,p})_{1 \leq i \leq n}$, nous parlons de **modèle linéaire** si nous supposons qu'il existe $p+1$ paramètres scalaires $(\beta_0, \dots, \beta_p) \in \mathbb{R}^{p+1}$ et n variables aléatoires indépendantes $(\varepsilon_1, \dots, \varepsilon_n)$ gaussiennes centrées et de même variance telles que :

$$\forall i \in \{1, \dots, n\},\ Y_i = \beta_0 + \sum_{j=1}^{p} \beta_j x_{i,j} + \varepsilon_i.$$

Nous parlons de modèle linéaire car si nous prenons les notations suivantes :

$$\boldsymbol{Y} = \begin{pmatrix} Y_1 \\ \vdots \\ Y_n \end{pmatrix}, \boldsymbol{\beta} = \begin{pmatrix} \beta_0 \\ \vdots \\ \beta_n \end{pmatrix}, \boldsymbol{X} = \begin{pmatrix} 1 & x_{1,1} & \cdots & x_{1,j} & \cdots & x_{1,p} \\ 1 & x_{2,1} & \cdots & x_{2,j} & \cdots & x_{2,p} \\ \vdots & \vdots & \ddots & \vdots & & \vdots \\ 1 & x_{i,1} & \cdots & x_{i,j} & \cdots & x_{i,p} \\ \vdots & \vdots & & \vdots & \ddots & \vdots \\ 1 & x_{n,1} & \cdots & x_{n,j} & \cdots & x_{n,p} \end{pmatrix} \text{ et } \varepsilon = \begin{pmatrix} \varepsilon_1 \\ \vdots \\ \varepsilon_n \end{pmatrix},$$

le problème se résume par :

$$\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{\beta} + \varepsilon. \tag{2.1}$$

Attention au piège

Le vocabulaire *linéaire* vient du fait que l'équation (2.1) est un modèle linéaire donc que nous avons une combinaison linéaire et pas du fait que les co-variables sont linéaires. Par exemple, le modèle :

$$Y_i = \beta_0 + \beta_1 x^2 + \varepsilon_i$$

est un modèle linéaire car, si nous regardons $X = x^2$, nous avons :

$$Y_i = \beta_0 + \beta_1 X + \varepsilon_i.$$

Ainsi la méthode pour analyser des données en admettant qu'il existe un lien linéaire entre les variables est la suivante :

Point méthode

1. Faire une représentation graphique ou proposer une représentation basée sur une analyse en composante principale (voir par ou la ressource *R4.02 : Méthodes factorielles*).
2. Faire une estimation des paramètres.
3. Vérifier si certains paramètres peuvent être considérés comme nuls et refaire une estimation dans ce cas.
4. Représenter *résidus*, c'est-à-dire la différence entre les observations et les estimations, pour vérifier s'ils sont gaussiens, centrés et de même variance ou pas.

2.2 Application

Dans cette partie, nous prenons le même exemple que pour la ressource *R5.10 - Python pour la Science des Données* sur les notations des jeux de Nintendo [1].

Codage en Python

Nous supposons que nous avons chargé les données :

```
import pandas as pd
Nintendo = pd.read_csv("NintendoGames.csv", header=0)
```

Nous commençons par afficher la représentation entre le score donné par les utilisateurs (en ordonnées) en fonction du score donné par le site (en abscisse) sous forme de nuage de points pour les 690 jeux dont nous avions les informations pour les deux variables (voir la figure 2.1).

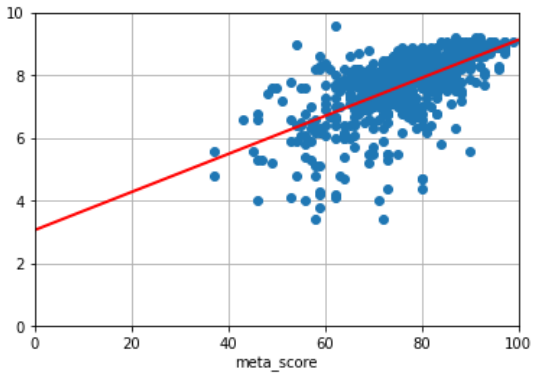

FIGURE 2.1 – Représentation sous forme de nuage de points du score donné par les utilisateurs (en ordonnées) en fonction du score donné par le site `metacritic.com` en abscisse. Le trait rouge correspond à une estimation par une droite de régression (voir le chapitre 2). 690 jeux concernés (63,07% de répondants).

Codage en Python

Les codes pour étudier les variables `meta_score` et `user_score` sont les suivants :

```
from scipy.stats import linregress
# Enlever les valeurs NaN
```

1. Les élèves intéressé·e·s pour explorer d'avantage ce jeu de données peuvent retrouver les informations sur le polycopié disponible sur mon site internet : `https://membres-ljk.imag.fr/Vincent.Brault/`

```
mask = ~np.isnan(Nintendo['meta_score']) & ~np.isnan(
    Nintendo['user_score'])
# Répondants
np.sum(mask)/Nintendo.shape[0]*100
# Régression linéaire
slope, intercept, r_value, p_value, std_err = linregress(
    Nintendo['meta_score'][mask], Nintendo['user_score'][mask])
# Affichage
ax = Nintendo.plot(x="meta_score", y="user_score", style='o', legend=False)
ax.set_xlim(0, 100)
ax.set_ylim(0, 10)
# Ajout du trait rouge
ax.plot([0.0, 100.0], [intercept, intercept+100*slope], 'r-', lw=2)
ax.grid()
```

Le modèle ainsi présenté est le suivant :

$$\forall i \in \{1,\dots,n\}, \underbrace{\texttt{user\_score}_i}_{Y_i} = \underbrace{\texttt{intercept}}_{\beta_0} + \underbrace{\texttt{slope}}_{\beta_1} \times \underbrace{\texttt{meta\_score}_i}_{X_{1,i}} + \varepsilon_i \text{ où } \varepsilon_i \overset{iid}{\sim} \mathcal{N}\left(0, \sigma^2\right) \tag{2.2}$$

et nous retrouvons la forme de l'équation (2.1) avec une seule co-variable. De plus, nous avons les informations suivantes :

- `r_value` est le **coefficient linéaire de Pearson** vu en ressource *R1.04 - Statistique descriptive 1.*
- `p_value` est la **p-valeur** du **test de corrélation linéaire de Pearson** vu en ressource *R3.06 - Tests d'hypothèses pour l'analyse bi-variée.*
- `std_err` est l'**écart-type** sur l'estimation du paramètre β_1. Sans entrer trop dans les détails, l'estimation $\widehat{\beta_1}$ est une variable aléatoire dépendant des données dont nous l'estimation est une observation. Ainsi, la théorie nous permet d'avoir une *plage* de valeurs possibles dont $\widehat{\beta_1}$ est la valeur centrale.

Remarque

Notons que nous pouvons également accéder à l'écart-type de l'estimation de l'*intercept* en faisant la commande suivante :

```
linregress(Nintendo['meta_score'][mask], Nintendo['user_score'][mask]).intercept_stderr
```

Pour l'estimation, comme nous n'avons que deux variables, nous pourrions utiliser le code précédent. Néanmoins, nous utiliserons la librairie `statsmodels.api`. L'avantage et, en même temps, l'inconvénient de la fonction `OLS` de la bibliothèque `statsmodels.api` est qu'elle ne propose pas de base l'estimation de l'*intercept* c'est-à-dire la valeur si toutes les co-variables sont nulles. Pour contrer cela, nous devons corriger légèrement la matrice des co-variables en ajoutant une colonne composée uniquement de 1.

Codage en Python

Les codes pour préparer les données sont les suivants :

```
import numpy as np

# On ne garde que les individus où on a l'information
Nintendo2 = Nintendo.loc[mask,:]
n = Nintendo2.shape[0]

# On ajoute une colonne avec que des 1
X_intercept = np.column_stack((np.ones(n),Nintendo2['meta_score']))
```

TABLE 2.1 – Sortie de la commande `results.summary()` en admettant que `results` est l'estimation par la fonction `OLS` de la bibliothèque `statsmodels.api`

```
                            OLS Regression Results
==============================================================================
Dep. Variable:             user_score   R-squared:                       0.391
Model:                            OLS   Adj. R-squared:                  0.390
Method:                 Least Squares   F-statistic:                     441.7
Date:                Thu, 07 Dec 2023   Prob (F-statistic):           3.91e-76
Time:                        18:58:22   Log-Likelihood:                -829.58
No. Observations:                 690   AIC:                             1663.
Df Residuals:                     688   BIC:                             1672.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          3.0729      0.222     13.830      0.000       2.637       3.509
x1             0.0606      0.003     21.017      0.000       0.055       0.066
==============================================================================
Omnibus:                      185.453   Durbin-Watson:                   1.592
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              591.439
Skew:                          -1.276   Prob(JB):                    3.72e-129
Kurtosis:                       6.749   Cond. No.                          557.
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

Ensuite, nous faisons l'estimation et en demandant un résumé des informations, nous obtenons le tableau 2.1.

Codage en Python

Les codes estimer les paramètres sont les suivants :

```
import statsmodels.api as sm

# Modélisation
model = sm.OLS(Nintendo2['user_score'], X_intercept)
results = model.fit()

# Affichage des informations
print(results.summary())
```

Les informations du tableau 2.1 sont les suivantes :

- Les coefficients (`coef`) donnent les valeurs estimées des $\boldsymbol{\beta}$. Ici, β_0, l'*intercept*, vaut environ 3.0729 et β_1 environ 0.0606.
- La p-valeur (`P>|t|`) donne la probabilité que chaque coefficient peut être considéré comme nul. Plus cette valeur est petite, moins il est plausible que nous puissions supposer que le coefficient est nul. Ici, les valeurs sont plus petites que 10^{-4} donc on rejette l'hypothèse que les valeurs sont nulles.
- La valeur du R^2 ajusté (`Adj. R-squared`) donne une indication s'il existe effectivement un lien linéaire entre la variable réponse et les autres variables. Sa valeur est comprise entre 0 et 1 et plus elle est proche de 1, plus il est plausible qu'il y a effectivement une relation linéaire. Ici, nous voyons que la valeur n'est pas vraiment proche de 1 donc la relation ne semble pas totalement linéaire.

Nous pouvons vérifier la cohérence de l'estimation en ajoutant la droite estimée au nuage de points (voir la figure 2.2). Nous voyons sur cette figure que la droite semble suivre le nuage de points mais certaines observations semblent très éloignées et, de plus, les points du bas semblent plus éloignés. Ceci peut être dû au fait que nos scores sont bloqués par une valeur maximale qui tasse les valeurs.

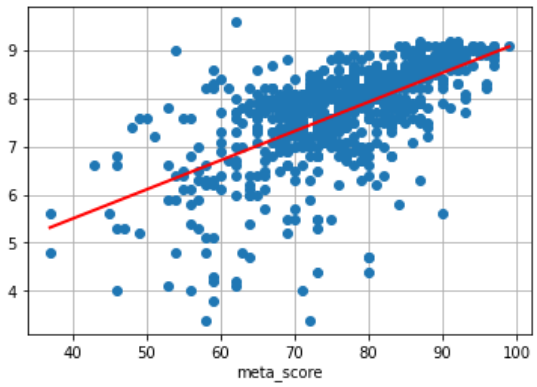

FIGURE 2.2 – Représentation sous forme de nuage de points du score donné par les utilisateurs (en ordonnées) en fonction du score donné par le site metacritic.com en abscisse. Le trait rouge correspond à une estimation par une droite de régression faite à l'aide de la bibliothèque `statsmodels.api`. 690 jeux concernés (63,07% de répondants).

Ensuite, nous devons regardes les **résidus**, notés $\widehat{\varepsilon}_i$, qui se calculent en enlevant la tendance estimée aux observations :

$$\forall i \in \{1, \dots, n\},\ \widehat{\varepsilon}_i = Y_i - \left(\widehat{\beta}_0 + \sum_{j=1}^{p} \widehat{\beta}_j x_{i,j} \right).$$

En python, pour le type `statsmodels.regression.linear_model.RegressionResultsWrapper`, c'est la variable `resid`. La première chose que nous pouvons faire est un histogramme des résidus (voir la figure 2.3).

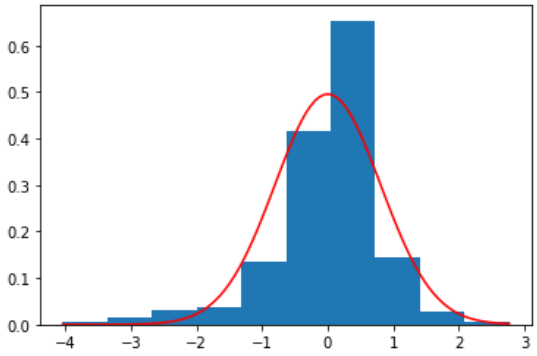

FIGURE 2.3 – Représentation sous forme d'histogramme des résidus. Le trait rouge correspond à une estimation de la densité si les résidus étaient gaussiens.

Nous avons ajouté une estimation de la densité pour voir si l'histogramme s'en approche ou pas. Nous pouvons remarquer que la distribution est asymétrique avec de fortes valeurs négatives ; ceci est en accord avec la valeur du Skewness du tableau 2.1. Ceci laisse penser que l'hypothèse que les résidus soient gaussiens n'est pas vérifiée.

Codage en Python

Le code utilisé pour faire un histogramme des résidus est le suivant :

```
from matplotlib import pyplot as plt

# Récupération des résidus
```

```
residus = np.array(results.resid)
x = np.linspace(min(residus), max(residus), 100)

# Création de la densité de comparaison
sigma = np.var(residus)
ecart_type = np.sqrt(sigma)
courbe_normale = (1 / (ecart_type * np.sqrt(2 * np.pi))) * np.exp(
    -0.5 * ((x - 0) / ecart_type)**2)

# Affichage
fig, ax = plt.subplots()
ax.hist(residus, density=True)
plt.plot(x, courbe_normale, color='red', label='Courbe Normale')
plt.show()
```

Afin de mieux comprendre ce à quoi on peut s'attendre dans le cas d'un bruit gaussien, nous avons représenté sur la figure 2.4 les histogrammes d'une simulation de variables gaussiennes en augmentant le nombre d'observations ($n \in \{20, 200, 2000\}$). Nous voyons que, bien que les résidus sont gaussiens, l'histogramme n'épouse pas parfaitement la densité. Néanmoins, en comparaison avec la figure 2.3, les observations sont plus symétriques et ne vont pas prendre de très grandes ou très petites valeurs.

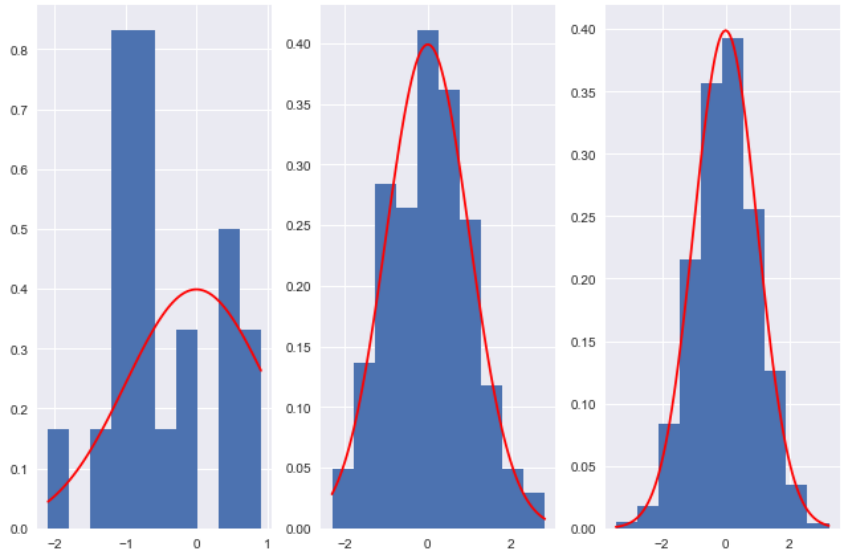

FIGURE 2.4 – Représentation sous forme d'histogramme de simulations gaussiennes avec 20 observations (à gauche), 200 observations (au milieu) et 2000 observations (à droite). Le trait rouge est la densité.

Codage en Python

Le code utilisé pour faire la figure 2.4 est :

```
fig, axs = plt.subplots(1, 3, figsize=(9, 6))
ecart_type = 1
# X1
noise = npr.standard_normal(20)
x = np.linspace(min(noise), max(noise), 100)
courbe_normale = (1 / (ecart_type * np.sqrt(2 * np.pi))) * np.exp(
    -0.5 * ((x - 0) / ecart_type)**2)
axs[0].hist(noise, density=True)
axs[0].plot(x, courbe_normale, color='red', label='Courbe Normale')
# X2
noise = npr.standard_normal(200)
```

```
x = np.linspace(min(noise), max(noise), 100)
courbe_normale = (1 / (ecart_type * np.sqrt(2 * np.pi))) * np.exp(
    -0.5 * ((x - 0) / ecart_type)**2)
axs[1].hist(noise, density=True)
axs[1].plot(x, courbe_normale, color='red', label='Courbe Normale')
# X3
noise = npr.standard_normal(2000)
x = np.linspace(min(noise), max(noise), 100)
courbe_normale = (1 / (ecart_type * np.sqrt(2 * np.pi))) * np.exp(
    -0.5 * ((x - 0) / ecart_type)**2)
axs[2].hist(noise, density=True)
axs[2].plot(x, courbe_normale, color='red', label='Courbe Normale')

plt.tight_layout()
plt.show()
```

Ensuite, nous pouvons regarder la distribution sous forme de *qqplot* (voir la figure 2.5).

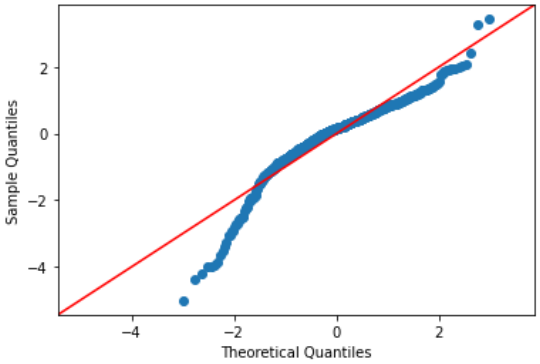

FIGURE 2.5 – Représentation sous forme de qqplot des résidus. Plus les points sont proches du trait rouge, plus nous pouvons considérer que les résidus sont gaussiens.

Plus les points sont proches de la droite, plus nous pouvons considérer que les résidus sont gaussiens. Ici, la distribution ressemble à un S ce qui laisse penser que la distribution n'est pas gaussienne.

Codage en Python

Le code utilisé pour faire un *qqplot* des résidus est le suivant :

```
import pylab

# Récupération des résidus normalisés
noise = np.array(results.resid_pearson)

# Affichage du qqplot
sm.qqplot(noise, line='45')
pylab.show()
```

De même, sur la figure 2.6, nous avons représenté les *qqplots* pour plusieurs simulations de loi gaussiennes ($n \in \{20, 200, 2000\}$). A nouveau, les points ne sont pas exactement sur le trait noir en pointillés mais ne sont pas très écartés (surtout quand nous avons beaucoup d'observations). De plus, les valeurs extrêmes s'écartent peu (par opposition à ce que nous voyons sur la figure 2.5).

Codage en Python

Le code utilisé pour faire la figure 2.6 est :

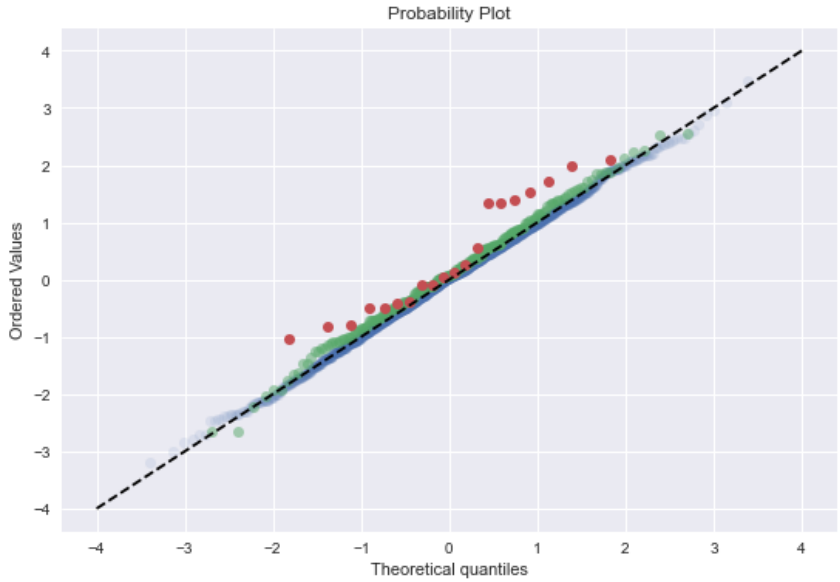

FIGURE 2.6 – Représentation sous forme de qqplot de simulations gaussiennes avec 20 observations (en rouge), 200 observations (en vert) et 2000 observations (en bleu). Le trait en pointillés noir correspond à l'objectif.

```
import scipy.stats as stats

plt.style.use('seaborn')

fig, ax = plt.subplots(1, 1, figsize=(9, 6))
stats.probplot(npr.standard_normal(2000), plot=plt, fit=False)
stats.probplot(npr.standard_normal(200), plot=plt, fit=False)
stats.probplot(npr.standard_normal(20), plot=plt, fit=False)

# Change colour of scatter
ax.get_lines()[0].set_markerfacecolor('C0')
ax.get_lines()[0].set_alpha(0.1)
ax.get_lines()[1].set_markerfacecolor('C1')
ax.get_lines()[1].set_alpha(0.5)
ax.get_lines()[2].set_markerfacecolor('C2')
ax.get_lines()[2].set_alpha(1)

# Add on y=x line
ax.plot([−4, 4], [−4, 4], c='black', ls="−−")
```

Chapitre 3

Aide mémoire en Python

Dans cette annexe, nous remettons des commandes de bases en Python nécessaires à la survie.

3.1 Manipulation pour les environnements de travail

Pour manipuler les environnements, la bibliothèque `os` permet d'interagir avec le système d'exploitation :

```
import os
```

Notamment, il est possible de savoir à tout moment où on se trouve à l'aide de la fonction `getcwd` :

```
os.getcwd()
```

Pour changer l'environnement de travail, c'est la fonction `chdir` :

```
# Si pwd est une variable str contenant le chemin
os.chdir(pwd)
```

3.2 Barre de progression

Lorsque les calculs sont longs, il peut être intéressant de savoir où en est le code. Pour cela, et si vous avez un boucle `for`, il peut être intéressant d'ajouter une barre de progression. Ceci peut se faire avec la fonction `tqdm` de la bibliothèque du même nom :

```
from tqdm import tqdm
j = 0
for i in tqdm(range(10**8)):
    j += i
```

3.3 Recherche de valeurs particulières

Pour rechercher des valeurs particulières, nous savons qu'il est possible d'utiliser l'opérateur `"=="` qui nous renvoie des réponses logiques `True` et `False`. Souvent, c'est la position qui nous intéresse. Dans ce cas, nous pouvons utiliser la fonction `flatnonzero` de la bibliothèque `numpy`. Par exemple, pour chercher les positions des valeurs manquantes d'une colonne d'un tableau `Tableau` de type `DataFrame`, nous avons le code suivant :

```
import pandas as pd
import numpy as np
np.flatnonzero(pd.isna(Tableau["Nom colonne"]))
```

3.4 Enlever une valeur

Dans Python, la valeur `None` (sans guillemets) représente une valeur manquante. En remplaçant une valeur par `None`, on peut généralement continuer à utiliser les autres fonctions sans incidences.

Bibliographie

F. Husson, S. Lê, et J. Pagès. Analyse de données avec R. Presses universitaires de Rennes, 2016.

H. Le Tellier. L'Anomalie. Gallimard, 2020.

É. Schultz et M. Bussonnier. Python pour les SHS. Introduction à la programmation pour le traitement de données. Pratique de la statistique. Presses universitaires de Rennes, 2021.

Table des figures

Liste des tableaux

Index