École Polytechnique de l'Université de Tours
64, Avenue Jean
Portalis 37200
TOURS, FRANCE
(33)2-47-36-14-14
www.polytech.univ-tours.fr

Parcours des Écoles d'Ingénieurs Polytech

Année 2020/ année 2021

Statistiques descriptives

Étudiants

Félix Baubriaud
Célian Chicon
felix.baubriaud@etu.univ-tours.fr
celian.chicon@etu.univ-tours.fr

Encadrant

Christophe Lenté
christophe.lente@univ-tours.fr

Avertissement

Ce document a été rédigé par **Félix Baubriaud** et **Célian Chicon** susnommés les auteurs. L'École Polytechnique de l'Université François Rabelais de Tours est représentée par **Christophe Lenté** susnommé le tuteur académique.

Par l'utilisation de ce modèle de document, l'ensemble des intervenants du projet acceptent les conditions définies ci-après.

Les auteurs reconnaissent assumer l'entière responsabilité du contenu du document ainsi que toutes suites judiciaires qui pourraient en découler du fait du non-respect des lois ou des droits d'auteur.

Les auteurs attestent que les propos du document sont sincères et assument l'entière responsabilité de la véracité des propos.

Les auteurs attestent ne pas s'approprier le travail d'autrui et que le document ne contient aucun plagiat.

Les auteurs attestent que le document ne contient aucun propos diffamatoire ou condamnable devant la loi.

Les auteurs reconnaissent qu'ils ne peuvent diffuser ce document en partie ou en intégralité sous quelque forme que ce soit sans l'accord préalable du tuteur académique et de l'entreprise.

Les auteurs autorisent l'école polytechnique de l'université François Rabelais de Tours à diffuser tout ou partie de ce document, sous quelque forme que ce soit, y compris après transformation en citant la source. Cette diffusion devra se faire gracieusement et être accompagnée du présent avertissement.

Table des matières

Introduction

L’objectif de ce projet est de réaliser une librairie (bibliothèque) de statistiques avec pour langage python.
Une telle librairie peut s'avérer très utile dès qu’il s’agit d’étudier une liste de valeurs. Par exemple, on pourra étudier une liste des notes obtenues à un examen en utilisant la librairie pour calculer la moyenne, la médiane etc. afin d’observer comment sont réparties les notes. On pourra aussi à l’aide de cette librairie comparer les températures dans une région par rapport aux moyennes saisonnières, en calculant sa variance ou encore ses coefficients de symétrie et d’aplatissement.
Ceci pour dire que le domaine d’application des statistiques descriptives est très vaste et que ces dernières peuvent permettre de modéliser simplement des ensembles de données parfois difficiles à décrire.
Nous nous sommes intéressés à plusieurs types de représentation des échantillons. D’abord nous avons vu les séries statistiques brutes, qui constituent le modèle le plus simple pour décrire des échantillons de caractères discrets. Puis nous sommes passés à la représentation sous forme dépouillée, qui s’avère plus pratique lorsque les valeurs contenues dans un échantillon se répètent plusieurs fois. Enfin, nous avons étudié la représentation des séries statistiques en classes, qu’on peut représenter à l’aide d’un histogramme. Ce dernier type de représentation est réservé à des échantillons de taille importante, et dont les éléments ne sont pas décrits par une valeur précise mais seulement par un intervalle.
On pourra remarquer qu’on peut passer d’une série dépouillée à une série brute et inversement. On peut également transformer une série en classes en une série brute, ce que nous avons d’ailleurs fait pour faciliter le code.

1 Série statistique sous forme brute

1.1. Définition

Une série statistique brute se présente sous la forme $(v_1,v_2,...,v_N)$ où $v_1,v_2,...,v_N$ représentent des nombres réels, pas forcément distincts. En python, on représentera une série statistique brute par une liste $L=[i_0,i_1,...,i_{N-1}]$ avec $i_0,i_1,...,i_{N-1}$ de type *int* ou *float*. Ces valeurs sont associées à un caractère observé et pourront permettre d'effectuer une enquête statistique sur ce caractère.

1.2. Paramètre de position

1.2.1 Le mode

Le mode d'une série statistique est la valeur apparaissant le plus de fois. Elle n'est pas forcément unique. Pour trouver cette/ces valeurs numériquement, nous avons créé la fonction ci-dessous:

```
def mode(self):
    """prend en entrée une liste (série brute) et retourne une liste contenant la/les valeurs apparaissant le plus de fois"""
    if self.l!=[]:
        maxi=0
        m=[]
        for i in self.l:
            n=0
            for j in self.l:
                if i==j:
                    n+=1
            if n>maxi and i not in m:
                maxi=n
                m.clear()
                m.append(i)
            if n==maxi and i not in m:
                m.append(i)
        return m
    else:
        print("self.l est vide")
```

Figure 1 - Fonction mode()

Le principe de cette fonction est le suivant : on parcourt la liste à l'aide d'une boucle for. Pour chaque valeur, on regarde combien de fois elle apparaît dans la liste entière. Si son nombre d'apparition est égal à celui de la valeur apparaissant le plus de fois parmi celles déjà parcourues par la liste for (qu'on appelle maxi), on ajoute sa valeur à la liste m (le mode). Si ce nombre dépasse maxi, alors il devient maxi et on vide, grâce à la fonction prédéfinie clear(), la liste m à laquelle on ajoute la valeur associée au nombre d'apparition.

1.2.2. La moyenne

Pour calculer la moyenne d'une liste brute, il suffit de sommer tous les éléments de la liste puis de diviser cette somme par le nombre de termes. Nous avons créé une fonction en python qui calcule la moyenne. Pour alléger le code, nous avons utilisé les fonctions prédéfinies "sum()" et "len()".

```python
def moyenne(self):
    """prend en entrée une liste (série brute) et retourne un nombre (la moyenne de la série)"""
    if self.l!=[]:
        self.moy=(sum(self.l)/len(self.l),True)
        return self.moy[0]
    else:
        print("self.l est vide")
```

Figure 2 - Fonction moyenne()

1.2.3. Médiane et quantiles

Les quantiles sont les valeurs séparant la population d'une liste triée par ordre croissant en plusieurs parties. La médiane sépare la liste en 2, les quartiles en 4, les centiles en 100 etc. Lorsqu'il n'est pas possible de partager la liste avec un nombre de la liste (par exemple pour la médiane dans une liste possédant un nombre pair d'éléments), il existe deux conventions. La première consiste à arrondir à la valeur supérieure tandis que la deuxième prend la moyenne des valeurs inférieure et supérieure. Dans notre fonction python, nous avons utilisé la deuxième convention, qui semble plus précise:

```python
def quantiles(self,q):
    """prend en entrée une liste (série brute) ainsi que le coefficient (q) qui sert à découper
    la série et retourne une liste des nombres qui découpent la série en q parties (ex q=4 pour quartile)"""
    if self.l!=[] and type(q)==int:
        lq=[] #liste qui retournera les quantiles
        for i in range(1,q):
            k=i*(len(self.l)+1)/(q) #découpage de la liste en q parties
            if int(k)==k: #si k est entier, on prend la k-ième valeur de la liste classée par ordre croissant (d'indice k-1)
                lq.append(sorted(self.l)[int(k)-1])
            else: #si k n'est pas entier, on prend la moyenne de la partie entière de k et du terme suivant.
                lq.append((sorted(self.l)[int(k)-1]+sorted(self.l)[int(k)])/2)
        return lq
    else:
        print("self.l est vide ou q n'est pas entier")
```

Figure 3 - Fonction mode()

Pour comprendre cette fonction on peut considérer l'exemple suivant:

L=[7 , 9 , 12 , 3 , 17 , 5 , 2 , 14 , 3]

La méthode prédéfinie sorted(L) renvoie alors la liste :

sorted(L)=[2 , 3 , 3 , 5 , 7 , 9 , 12 , 14 , 17]

Pour calculer les quartiles de cette liste, on commence par diviser sa longueur+1 par 4 : (9+1)/4=2.5. Dans la fonction, on parcourt donc les valeurs k_1=2.5, k_2=5 et k_3=7.5 avec la liste for. Comme k_2=5 est entier, le 2ème quartile (ou la médiane) est la 5ème valeur de la liste triée, c'est-à-dire 7, qui a pour indice 4 dans la liste. Quant aux nombres k_1=2.5 et k_3=7.5, ils ne sont pas entiers donc on prend la moyenne de la 2ème et de la 3ème valeur pour le 1er quartile ((3+3)/2=3) et la moyenne de la 7ème et de la 8ème valeur pour le 3ème quartile ((12+14)/2=13). Ainsi la fonction quantile(L,4) renvoie la liste des quartiles [3,5,13].

1.3.Paramètre de dispersion

1.3.1. L'étendue

L'étendue correspond à la longueur de l'intervalle auquel appartiennent les nombres de la liste.
Autrement dit, pour une série brute, c'est la différence entre la plus grande valeur et la plus petite, ce qui donne en python:

```python
def etendue(self):
    """prend en entrée une liste (série brute) et retourne un nombre (l'étendue de la série)"""
    if self.l!=[]:
        return max(self.l)-min(self.l)
    else:
        print("self.l est vide")
```

Figure 4 - Fonction etendue()

1.3.2. L'écart arithmétique moyen

Il s'agit de calculer les écarts (en valeur absolue) entre la moyenne et chacun des nombres de la liste puis d'en faire la moyenne. Pour la valeur absolue, on a préféré ne pas utiliser les modules "math" ou "numpy" dans notre fonction. Pour cela nous avons d'abord mis au carré l'expression, puis nous l'avons passée à la puissance ½ ce qui revient à appliquer la racine carré :

```python
def ecartMoyen(self):
    """prend en entrée une liste (série brute) et retourne un nombre (l'écart moyen de la série)"""
    if self.l!=[]:
        return sum([((self.moyenne()-i)**2)**(1/2) for i in self.l])/len(self.l)
    else:
        print("self.l est vide")
```

Figure 5 - Fonction ecartMoyen()

1.3.3. La variance

Si on voulait la transcrire en langage français, la variance serait "la moyenne des écarts à la moyenne au carré". En python, on peut donc la représenter sous cette forme :

```python
def variance(self):
    """prend en entrée une liste (série brute) et retourne un nombre (la variance de la série)"""
    if self.l!=[]:
        return sum([(i-self.moyenne())**2 for i in self.l])/len(self.l)
    else:
        print("self.l est vide")
```

Figure 6 - Fonction variance()

1.3.4. L'écart type

C'est la racine carrée de la variance. Comme pour l'écart arithmétique moyen, plutôt que de faire la racine carrée, on met à la puissance ½ :

```python
def ecartType(self):
    """prend en entrée une liste (série brute) et retourne un nombre (l'écart type de la série)"""
    if self.l!=[]:
        return self.variance()**(1/2)
    else:
        print("self.l est vide")
```

Figure 7 - Fonction ecartType()

1.4.Moments et paramètres de formes

1.4.1.Moment d'ordre k

Pour calculer le moment d'ordre k d'une série statistique, k étant un entier positif à saisir par l'utilisateur, on somme chaque élément qu'on aura au préalable mis à la puissance k, puis on divise cette somme par le nombre d'éléments de la liste.

```
def moments(self,k):
    """prend en entrée une liste (série brute) ainsi que k un nombre qui est
    l'ordre du moment calculé et retourne un nombre (le moment de la série)"""
    if self.l!=[] and type(k)==int and k>=0:
            return sum([i**k for i in self.l])/len(self.l)
    else:
        print("self.l est vide ou k n'est pas entier positif")
```

Figure 8 - Fonction moments()

On peut remarquer que le moment d'ordre 1 donne la moyenne de la série et que celui d'ordre 0 est toujours égal à 1.

1.4.2.Moment centré d'ordre k

De même, on définit le moment centré d'ordre k comme la moyenne des écarts à la moyenne mis à la puissance k. La fonction en python est donc semblable à celle de la variance.

```
def momentsCentre(self,k):
    """prend en entrée une liste (série brute) ainsi que k un nombre qui est
    l'ordre du moment calculé et retourne un nombre (la moment centré de la série)"""
    if self.l!=[] and type(k)==int and k>=0:
            return sum([(i-self.moyenne())**k for i in self.l])/len(self.l)
    else:
        print("self.l est vide ou k n'est pas entier positif")
```

Figure 9 - Fonction momentsCentre()

Cette fois, le moment centré d'ordre 2 donne la variance tandis que le moment centré d'ordre 0 et celui d'ordre 1 font respectivement 1 et 0 quelle que soit la série.

1.4.3.Les coefficients de Fisher

Le premier coefficient de Fisher est appelé le coefficient d'asymétrie. Il permet de distinguer comment sont réparties les valeurs de la série autour de la moyenne : si ce coefficient est positif, cela signifie que les valeurs s'étalent davantage à droite de la moyenne. S'il est négatif, l'étalement est à gauche. Enfin, si la répartition de l'échantillon ou de la distribution est symétrique autour de la moyenne, le coefficient d'asymétrie est nul. Pour calculer ce coefficient, on effectue le quotient du moment centré d'ordre 3 et du moment centré d'ordre 2 mis à la puissance 3/2 :

```
def coefFisher1(self):
    """prend en entrée une liste (série brute) et retourne un nombre (le premier coefficient de Fisher de la série)"""
    if self.l!=[]:
        return self.momentsCentre(3)/(self.momentsCentre(2)**(3/2))
    else:
        print("self.l est vide")
```

Figure 10 - Fonction coefFisher1()

Le second coefficient de Fisher est le coefficient d'aplatissement. Il indique la dispersion des valeurs autour de la moyenne. Plus la valeur de ce coefficient est grande, plus les valeurs de la série se resserrent autour de la moyenne et il est égal à 3 pour une distribution normale. On l'obtient en divisant le moment centré d'ordre 4 par le moment centré d'ordre 2 au carré :

```
def coefFisher2(self):
    """prend en entrée une liste (série brute) et retourne un nombre (le second coefficient de Fisher de la série)"""
    if self.l!=[]:
        return self.momentsCentre(4)/(self.momentsCentre(2)**2)
    else:
        print("self.l est vide")
```

Figure 11 - Fonction coefFisher2()

2 Série statistique sous forme dépouillée

2.1. Définition

Une série statistique dépouillée est une famille dont les éléments sont de la forme (xi,ni) ou (xi,fi), i appartenant à [1,p] où p est l'indice de chaque valeurs présentes dans la série et ni et fi étant respectivement l'effectif et la fréquence d'apparition de ces valeurs. Dans notre librairie nous avons adopté la première définition d'une série sous forme dépouillée. Pour définir une série statistique dépouillée dans notre programme, on utilise une liste de tuples $L=[(x_0,n_0),(x_1,n_1),...,(x_{p-1},n_{p-1})]$ avec $x_0,x_1,...,x_{p-1}$ de type *float* ou *int* et $n_0,n_1,...,n_{p-1}$ strictement positif de type *int*.

2.2. Transcription sous python

La forme d'une série dépouillée est différente par rapport à une série brute, ce qui est bien sûr dûe à une définition différente. Donc nous avons réalisé des méthodes propres à une série sous forme dépouillée ainsi qu'une méthode de tri. Nous avions également créé une méthode de transformation pour passer d'une série dépouillée à une série brute, mais nous ne l'avons pas utilisée dans notre bibliothèque finale malgré le fait que ce soit pratique et efficace.
Dans cette section nous allons seulement voir ou revoir les méthodes qui ont un changement majeur observable entre la version brute et la version pour une série dépouillée.

2.2.1. Méthode de tri

Nous avons réalisé une méthode de tri car ceci nous était nécessaire pour le calcul d'autres méthodes propres aux séries statistiques sous forme dépouillée.

```
def tri_d(self):
    """tri une liste dépouillée"""
    if self.l!=[]:
        l1,l2=[i[0] for i in self.l],[]
        l1.sort()
        for i in l1:
            for j in self.l:
                if j[0]==i:
                    l2.append(j)
        return l2
    else:
        print("self.l est vide")
```

Figure 12 - Fonction tri_d()

C’est une simple méthode de recréation des paires composant une série dépouillée, à partir des effectifs qui ont auparavant été triés dans l’ordre croissant grâce à la méthode sort(), ici appliquée à la liste des effectifs définies par compréhension.

2.2.2. Quantiles

Il est plus difficile de calculer les quantiles dans une série dépouillée que dans une série brute. Notamment on ne peut pas utiliser les méthodes prédéfinies len() et sorted(). On définit donc au début de la fonction la variable “long” qui est la longueur de la liste, et on appellera la fonction tri_d() créée précédemment pour considérer la liste triée par ordre croissant selon la première valeur du tuple. Comme pour notre fonction quantile pour les séries brutes, on parcourt les k_i qui représentent les indices des quantiles +1. Mais on définit également pour chaque k_i un compteur qui sommera les effectifs des tuples, et une variable x correspondant à l’indice du tuple considéré. Avec une boucle while, on parcourt alors les tuples de la liste et on s’arrête à celui contenant la k_i-ème valeur (d’indice k_i-1). Si k_i est entier ou si le tuple contient aussi la (k_i+1)-ème valeur alors le quantile correspond à la valeur de ce tuple. Sinon, le quantile est la moyenne des valeurs de ce tuple et du suivant.

```
def quantiles_d(self,q):
    """prend en entrée une liste (série dépouillée)
    ainsi que le coefficient (q) qui sert à découper
    la série et retourne une liste des nombres qui
    découpent la série en q parties (ex q=4 pour quartile)"""
    if self.l!=[] and type(q)==int and q>0:
        lq=[]
        long=sum(a[1] for a in self.l) #nombre de valeurs dans la liste : somme des effectifs
        for i in range(1,q):
            k=i*(long+1)/q
            compteur,x=0,-1 #on cherche quel nombre se situe à la k-ième place
                            # dans la liste de tuples triée par ordre croissant.
            while compteur<=k-1:
                x+=1
                compteur+=self.tri_d()[x][1]
            if int(k)==k or compteur>k: #si k est entier ou compris entre
                    # deux même valeur, on prend la valeur du x-ième tuple de la liste triée.
                lq.append(self.tri_d()[x][0])
            else: #si k-1<compteur<k, on prend la moyenne
                #des valeurs du x-ième tuple et du (x+1)ième tuple.
                lq.append((self.tri_d()[x][0]+self.tri_d()[x+1][0])/2)
        return lq
    else:
        print("self.l est vide ou q n'est pas entier positif")
```

Figure 13 - Fonction quantiles_d()

3 Série statistique en classes

3.1. Définition

Les statistiques en classes se font par découpage en sous-intervalles (classes) du domaine auquel appartiennent les valeurs étudiées. Chaque élément de la série se présente alors sous la forme ([borne inférieure, borne supérieure], effectif présent dans l'intervalle). En python, on définit donc une série statistique en classes comme ceci : $L=[([a_0,a_1],n_0),([a_2,a_3],n_1),...,([a_{2N},a_{2N+1}],n_N)]$ où la suite a_n est croissante, les a_i sont de type int ou float et les n_i sont strictement positifs et de type int.

Les statistiques en classes ne sont plus représentées sous forme de bâtons mais sous forme d'histogramme, c'est-à-dire de rectangle dont la base est proportionnelle à la largeur de la classe, et dont la hauteur est proportionnelle à l'effectif ou la fréquence (fréquence = effectif / effectif total) de la classe.

3.2. Transcription sous python

Pour les statistiques en classe nous avons réalisé une méthode de recherche de la classe modale et une méthode de transformation qui permet de revenir à une série sous forme brute.

3.2.1. Classe modale

La classe modale est la plus "haute" de toutes les classes sous forme d'histogramme, si on en revient à la définition c'est la classe avec le plus grand effectif.

```
def classModale(self):
    """prend en entrée un hisogramme et retourne une liste contenant la/les classes de hauteur maximale"""
    if self.l!=[]:
        cm=[]
        c=0
        for i in self.lh:
            print(i)
            if i[1]>c:
                cm.clear()
                cm.append(i[0])
                c=i[1]
            elif i[1]==c:
                cm.append(i[0])
        return cm
    else:
        print("self.l est vide")
```

Figure 14 - Fonction classModale()

On applique dans cette fonction un simple algorithme de recherche d'un maximum, ici l'effectif maximum en position i[1] dans un tuple qui correspond à une classe de la

forme ([borne inf,borne sup],effectif).

3.2.2. Méthode de transformation

Le principe est de revenir à une série brute dont on a déjà réalisé les méthodes de calculs de statistiques. Donc pour cela on a rédigé une méthode de passage (passpassHisto()).

```
def passpassHisto(self,l):
    lb=[]
    for i in l:
        for _ in range(i[1]):
            lb.append((i[0][0]+i[0][1])/2)
    return lb
```

Figure 15 - Fonction passpassHisto()

On parcourt la liste sous forme d'histogramme afin de calculer la moyenne de chacune des classes, puis on place ces moyennes dans la nouvelle série proportionnellement à leur effectif.

4 Statistiques doubles

4.1. Définition

Les statistiques doubles sont le lien entre deux séries statistiques. On étudie comment le couple de série est lié ou non, comment varient ensemble ou non les séries.

4.2. Transcription en python

Nous avons pour les statistiques doubles les méthodes de calcul de la covariance et du coefficient de corrélation.

4.2.1. La covariance

La covariance est la variance couplée entre deux séries statistiques, la variance étant pour rappel la moyenne des écarts à la moyenne au carré de la série.

```
def covariance(self,l2):
    """prend en entrée deux listes (séries brutes) et retourne un nombre (la covariance de ces deux séries)"""
    if self.l!=[] and l2.l!=[] and len(self.l)==len(self.l):
        return sum([(i-self.moyenne())for i in self.l][k]*[(j-l2.moyenne()) for j in l2.l][k] for k in range(len(self.l)))/len(l2.l)
    else:
        print("les listes sont vides ou ne sont pas de même longueur")
```

Figure 16 - Fonction covariance()

On applique dans notre code la formule de la covariance telle qu'elle est définie en mathématiques. La seule différence est qu'elle est ici définie par compréhension.

4.2.2. Le coefficient de corrélation

On obtient le coefficient de corrélation en divisant la covariance de deux séries par le produit des écarts types de ces deux séries.

```
def correlation(self,l2):
    """prend en entrée deux listes (séries brutes) et retourne un nombre (la corrélation de ces deux séries)"""
    if self.l!=[] and l2.l!=[] and len(self.l)==len(self.l):
        return self.covariance(l2)/(self.ecartType()*l2.ecartType())
    else:
        print("les listes sont vides ou ne sont pas de même longueur")
```

Figure 17 - Fonction correlation()

Pour le calcul on utilise les fonctions précédemment créées pour l'écart type et la covariance.

5 Structure python

5.1. Structure

Dans la création de notre librairie python nous avons d'abord fait un programme sous forme d'une liste de fonction, mais comme cela ne convient pas à la situation nous avons opté pour une classe regroupant les méthodes propres à tous les types de séries statistiques que l'on traite dans le projet.

5.2. Initialisation

L'initialisation de notre classe comprend 2 parties:

- un test pour savoir à quel type de séries statistiques appartient l'instance de la classe, et si cette série est ou non bien définie.
- une attribution au variable de stockage de la classe ainsi que des valeurs par défaut (liste vide).

```
def __init__(self,liste):
    c=0
    d=False
    if type(liste)==list and liste!=[]:#l doit être une liste non vide
        if type(liste[0])==int or type(liste[0])==float:#cas d'une liste brute
            c=len(liste)#condition pour pouvoir créer la liste
            for i in liste:
                if type(i)!=int and type(i)!=float:#l doit contenir des réels
                    c=0
        if type(liste[0])==tuple:#cas des listes dépouillées et des histogrammes
            if type(liste[0][0])==int or type(liste[0][0])==float:#cas d'une liste dépoulliée
                a=True
                for x in liste: #l doit contenir des tuples de 2 éléments : un réel et un entier positif non nul
                    if type(x)!=tuple or (type(x[0])!=int and type(x[0])!=float) or \
                        type(x[1])!=int or len(x)!=2 or x[1]<=0:
                        a=False
                if a:
                    for x in liste:
                        for i in liste:
                            if x[0]==i[0]:
                                c+=1 #si le 1er élément de chaque tuple est bien unique, on aura c=len(liste)
            elif type(liste[0][0]==list):#cas d'un histogramme
                b=liste[0][0][0]
                d=True
                for x in liste:#l doit contenir des tuples contenant une liste de deux réels et un entier positif non nul.
                                #On vérifie que le 1er nombre de la liste du tuple soit compris
                                # entre le 2ème de la liste du tuple précédent et le 2ème de la liste du même tuple.
                    if type(x)!=tuple or type(x[0])!=list or type(x[1])!=int or\
                        (type(x[0][0])!=int and type(x[0][0])!=float) or \
                    (type(x[0][1])!=int and type(x[0][1])!=float) or len(x)!=2 or \
                        len(x[0])!=2 or x[1]<=0 or x[0][1]<x[0][0] or x[0][0]<b:
                        d=False
                    b=x[0][1]
    if c==len(liste):#Vrai si la liste est brute ou dépouillée
        self.l=liste
    elif d:#Vrai seulement si la liste est un histogramme: on crée alors une liste brute à partir de l'histogramme
        self.l,self.lh=self.passpassHisto(liste),liste
    else:
        self.l=[]
    self.moy=(0,False)
```

Figure 18 - Fonction __init__()

Dans tous les cas, l'instance à créer doit être une liste non vide.

- Si le premier élément de la liste est un nombre réel, on considère que l'instance est une série statistique brute. On vérifie alors simplement que chaque élément de la liste est un réel. Si tel est le cas, on aura dans le programme c=len(liste) qui est la condition nécessaire pour créer l'instance.
- Si le premier élément de la liste est un tuple et que le premier élément de ce tuple est un réel, on considère que l'instance est une série statistique dépouillée. On vérifie donc que chaque élément de la liste est de la forme (x_i,n_i) avec x_i réel et n_i un entier strictement positif. On vérifie également que tous les x_i sont distincts. Pour cela on regarde pour chaque x_i combien de fois il apparaît dans la liste entière. Chaque x_i est censé apparaître une seule et unique fois et il y a autant de x_i que de tuple donc en sommant le nombre d'apparition de chaque x_i on devrait obtenir la longueur de la liste, c'est-à-dire c=len(liste) dans le code.
- Enfin, si le premier élément de la liste est un tuple et que le premier élément de ce tuple est une liste, on considère que l'instance est une série statistique en classe. On vérifie alors que chaque élément est de la forme $([a_i,a_{i+1}],n_i)$ où les a_i forment une suite croissante et les n_i sont des entiers strictement positifs. Si tel est le cas dans le code on laisse la valeur de c à 0 mais le booléen d prend la valeur "True".

Après cette série de tests, c=len(l) signifie que l'instance est une série brute ou dépouillée bien définie donc on peut la créer. Si cette condition n'est pas vérifiée mais que d="True", on est dans le cas d'une série statistique en classe bien définie. On définit alors self.l la série brute associée obtenue par la méthode "passpassHisto()" ainsi que self.lh la série en classes telle quelle qui permettra notamment d'utiliser la fonction "classModale()".Si ni l'une ni l'autre des conditions est vérifiée, l'instance est mal définie et on crée à la place de celle-ci une liste vide.

5.3. Les fonctions de la classe

Après l'initialisation, on définit dans la classe toutes les fonctions vues précédemment pour les listes brutes puis dépouillée, sans oublier les méthodes propres aux séries statistiques en classe ("passpassHisto()" et "classModale") et aux statistiques doubles ("covariance()" et "corrélation()"). Pour chaque fonction nous regardons d'abord si l'instance est vide ou non. D'autres vérifications peuvent s'avérer nécessaires. Par exemple, dans la fonction "quantile()", on vérifie que le paramètre k est bien entier et strictement positif. Dans le cas contraire, plutôt que d'exécuter le programme de la fonction, on renvoie un message d'erreur à l'utilisateur.

5.4. Test sur le fonctionnement de la bibliothèque

Pour s'assurer du bon fonctionnement de notre classe, nous avons fait de nombreux tests. Nous avons créé des fonctions retournant des séries de nombre aléatoire, en utilisant le module random.
La première prend en paramètre le nombre d'éléments qu'on veut dans la série et un intervalle puis renvoie une liste brute de nombres entiers compris dans cette intervalle.

```
def listealeatoire(longueur,intervalle):
    L=[]
    for i in range(0,longueur):
        L.append(random.randint(intervalle[0],intervalle[1]))
    return L
```

Figure 19 - Fonction listealeatoire()

La deuxième prend en plus en paramètre un effectif maximum et renvoie une série dépouillée dont chaque tuple est composé d'un nombre entier aléatoire mais distincts de celui des autres tuples compris dans l'intervalle et d'un nombre entier strictement positif ne dépassant pas l'effectif maximum.

```
def listealeatoiredepouille(longueur,intervalle,effectifmax):
    L=[]
    for i in range(0,longueur):
        a=random.randint(intervalle[0],intervalle[1])
        while a in [i[0] for i in L]:
            a=random.randint(intervalle[0],intervalle[1])
        L.append((a,random.randint(1,effectifmax)))
    return L
```

Figure 20 - Fonction listealeatoiredepouille()

La dernière, pour les statistiques en classe prend les mêmes paramètres que la deuxième puis découpe l'intervalle en parties égales selon la longueur entrée. La liste renvoyée est alors de la forme $[([a_0,a_1],n_0),([a_1,a_2],n_1),...,([a_{N-1},a_N],n_{N-1})]$ où N représente la longueur, les n_i sont des nombres entiers strictement positifs ne dépassant pas l'effectif maximum et
a_i=premier nombre de l'intervalle + i*longueur de l'intervalle/longueur de la liste.

```
def histogrammealeatoire(longueur,intervalle,effectifmax):
    L=[]
    for k in range(longueur):
        i=([intervalle[0]+k*(intervalle[1]-intervalle[0])/longueur,intervalle[0]\
        +(k+1)*(intervalle[1]-intervalle[0])/longueur],random.randint(1,effectifmax))
        L.append(i)
    return L
```

Figure 21 - Fonction histogrammealeatoire()

A l'aide de ces fonctions, nous avons pu créer des instances puis appeler les différentes fonctions contenues dans la classe. Nous avons comparé certaines valeurs

obtenues par ces fonctions avec les méthodes équivalentes fournies par le module statistics pour s'assurer qu'elles coïncidaient.

5.5. Les ajouts possibles

Dans un souci d'optimisation pour diminuer le nombre de calculs effectués on peut ajouter un booléen de vérification lié à une méthode souvent présente dans les calculs. La moyenne est un bon exemple car nous avons commencé à implémenter ce système en stockant la moyenne dans un tuple de la forme (moyenne, booléen). Ce système permet lors l'appel de la moyenne de vérifier si celle-ci a déjà été calculée avant (booléen = true si déjà calculée et false sinon).

```
self.moy=(0,False)
```

Figure 22 - initialisation de self.moy

Partie dans l'initialisation de ce système.

```
self.moy=(sum(self.l)/len(self.l),True)
return self.moy[0]
```

Figure 23 - corps de la méthode moyenne

Partie de la méthode moyenne qui remplace le tuple par défaut par la valeur de la moyenne ainsi que le booléen témoin.

Ce système bien qu'utile nécessite une modification dans toutes nos méthodes appelant la méthode moyenne. On doit ajouter un test afin de vérifier si celle-ci a été calculée ou non au préalable. Dans un souci de temps et car nous manipulons des séries courtes nous avons laissé cette amélioration possible de côté.
On peut appliquer cette méthode de vérification à toutes les méthodes usuelles de statistiques.

Nous pouvons conclure sur des résultats assez satisfaisants. En effet, la totalité des fonctions que nous avons conçues a l'air de marcher. La classe créée correspond donc bien à une librairie de méthodes statistiques descriptives qui fonctionnent pour les trois représentations les plus courantes : statistiques brutes, dépouillées et en classes. Plusieurs librairies de ce type comme le module statistics ou encore numpy existent déjà mais souvent ne prennent que des listes brutes en paramètre (c'est le cas pour le module statistics). Ces librairies aussi ne contiennent pas toujours autant de méthodes, notamment les coefficients de Fisher et les moments.

Bibliographie

https://www.soft-concept.com/surveymag/definition-fr/definition-coefficient-de-fisher.html

http://www.itse.be/statistique2010/co/22141_Cours_quantile_1.html

https://docs.python.org/3/library/statistics.html

https://fr.wikipedia.org/wiki/Statistique_descriptive

+PDF fournie par notre encadrant

Comptes rendus hebdomadaires

Compte rendu n°1

Nous avons écrit en python les premières fonctions: moyenne, médiane, écarts moyen, variance, étendue, mode, moment et moment et moment centré, coefficient de Fisher. Nous sommes en train de réfléchir à une fonction permettant de calculer tous les quantiles.

Compte rendu n°2

Aujourd'hui nous avons écrit la fonction quantiles et commencé à adapter les fonctions pour une liste dépouillée. Nous avons aussi utilisé le module statistics pour vérifier nos fonctions mais les coefs de Fisher sont indisponible dedans et les quantiles ainsi que le mode du module ne fonctionne pas.

Compte rendu n°3

Aujourd'hui, nous avons corrigé la fonction quantile car il y avait encore quelques problèmes. Nous avons également changé les fonctions mode et effectifs et créé des fonctions permettant de générer des listes aléatoires. Nous avons commencé à commenter notre programme et nous avons continué les fonctions pour les listes dépouillées.

Compte rendu n°4

Aujourd'hui, nous avons fini les fonctions, les commentaires et nous avons commencé la création de notre classe.

Compte rendu n°5

Aujourd'hui nous avons créé 2 classes : une pour les séries statistiques brutes et une autre pour les séries statistiques dépouillées. Dans ces classes sont regroupées toutes les fonctions faites précédemment. Nous essayons maintenant de gérer les exceptions concernant la création d'une instance via ces classes.

Compte rendu n°6

Aujourd'hui nous avons fait un point sur le travail effectué jusqu'à présent et nous avons continué nos classes qui comprenaient quelques erreurs. Nous avons commencé l'ajout d'un traitement d'exception tel que des problèmes d'entrées.

Compte rendu n°7

Aujourd'hui nous avons regroupé nos deux classes (séries brutes et dépouillées), ce qui a nécessité des changements dans les fonctions et dans l'init. Nous avons commencé à regarder les histogrammes et avons créé une fonction permettant de passer d'un histogramme à une liste brute.

Compte rendu n°8

Aujourd'hui nous avons avancé sur les stats en classes (histogrammes), commencé les statistiques doubles et amélioré l'init plus quelques petites corrections.

Compte rendu n°9

Aujourd'hui nous avons testé nos fonctions pour chaque type de liste (brute, dépouillée, histogramme) pour vérifier qu'il n'y avait pas de problème dans la classe. Nous avons également créé les fonctions calculant la covariance et la corrélation entre 2 listes mais nous n'arrivons pas à les inclure dans la classe.

Stats descriptives

Résumé

Le but de ce projet était de créer une librairie python permettant d'utiliser des méthodes de statistiques sur différents types de série. Nous avons donc conçu de nombreuses fonctions pour calculer les moments, les paramètres de position, de dispersion et de forme d'abord pour des séries statistiques brutes, puis pour des séries statistiques dépouillées. Ensuite, nous avons étudié les histogrammes et trouvé une méthode permettant de passer d'une série statistique en classes à une série brute. Enfin, nous nous sommes intéressés aux statistiques doubles, notamment pour décrire la covariance et la corrélation. Nous avons finalement créé une classe regroupant toutes les méthodes et vérifiant si une série statistique est bien définie ou non, et avons effectué quelques tests pour s'assurer du bon fonctionnement de notre classe.

Mots-clés

-statistiques -échantillon -programmation
-fonctions -dénombrement -modélisation

Abstract

This project goal was to make a python lib which supports different types of statistical series. To make it work, we have made many methods to calculate moments, positional parameters, dispersion and form parameters for brute type series. Then we've done it for "dépouillée" and we've studied histograms. We've made a switching method to go from a classes serie to a brute serie. Then we've worked on pair statistics. Finally we've made a python class with every method made before and we've made tests to know if a series was well defined according to our math definition on python. To end the project we've run tests to check if everything was working well.

Keywords

-statistics -functions -method
-programming -modeling -sample

Encadrant académique	Étudiant(e)s
Christophe Lenté	*Félix Baubriaud* *Célian Chicon*