Promotion 2023

Année A3

Projet DataScience & IA année 3

Souhail AIT LAHCEN

Rapport concernant le projet de début de semestre de DataScience & IA Année 3

Table des matières

Rapport du Projet de DataScience & IA Année 3 Année 2020-2021

Souhail Ait Lahcen

21 Mars 2021

I Présentation du Projet

Dans le cadre du premier problème de programmation de DataScience & IA nous devions réaliser un algorithme génétique pour déterminer un triplet (a,b et c) qui se rapproche le plus des vraies valeurs. L'objectif est donc de déterminer la combinaison (a,b,c) capable de rapprocher la fonction de **Weierstrass** au plus près des observations de températures effectuées à des instants données. Il fallait donc créer les méthodes et classes associées à ce projet.
Ainsi, l'objectif de ce rapport est de présenter les différentes méthodes et classes, que j'ai créé en lien avec le projet mais aussi faire part des problèmes que j'ai rencontrés.

Dans chaque sections et/ou sous-sections, nous pourrons utiliser des noms d'attributs associés aux classes appelées, ainsi veuillez les retrouver en annexe ou bien dans le programme Python.

Pour terminer cette introduction, vous devez savoir au préalable que vous pouvez démarrer le projet sur **Google collab** en suivant ce lien :
`https://colab.research.google.com/drive/1ziQhFqelcc9MOi5fY8BESVFEakJ_VfSW?usp=sharing`.

II Classe Individu

Avant de commencer, un individu dans notre algorithme génétique est un triplet (a,b,c) qui est crée de la manière suivante :

```
class Individu:
    def __init__(self, a=None, b=None, c=None):
        if(a==None or b == None or c== None ):
            h=list(numpy.random.standard_normal(1))
            self.a=h[0]
            while abs(self.a)>1:
                h=list(numpy.random.standard_normal(1))
                self.a=h[0]
            self.a=abs(self.a)
            self.b=random.randint(1,20)
            self.c=random.randint(1,20)
        else :
            self.a=a
            self.b=b
            self.c=c
```

Bien sûr, au début, je n'avais pas de classe Individu, car j'avais encore du mal à comprendre comment se crée une classe en Python, comme vous pouvez le voir sur la photo ci-dessous, je n'avais pas créé de classe mais une méthode :

```python
def Individu():
    # un couple a,b et c
    t=[]
    #a=random.gauss(0,1)
    h=list(numpy.random.standard_normal(1))
    a=h[0]
    while abs(a)>1:
        h=list(numpy.random.standard_normal(1))
        a=h[0]
    a=abs(a)
    b=random.randint(1,20)
    c=random.randint(1,20)
    t.append(a)
    t.append(b)
    t.append(c)
    return t
```

Cette méthode renvoie une liste composé, de la position 0 à 2, des valeurs de a, b et c. Mais comme nous allons le voir dans les prochaines parties, le fait d'utiliser cette méthode comme ça, pose de gros soucis dans la manipulation des données. En plus, aussi d'être dure à manipuler contrairement à l'utilisation d'une classe qui permet de manipuler "un par un" les individus, l'utilisation de cette méthode nous obliger à manipuler "trois par trois" les individus.
Ensuite, la visibilité du code est amélioré grâce à l'utilisation et création d'une classe.

Maintenant, passons à l'explication des créations du triplet (a,b et c). Tout d'abord, on vérifie si on manipule un individu s'il n'a pas déjà des valeurs attitrés si non on lui crée.
Pour a, qui d'après le sujet est compris entre [0,1] avec 0 et 1 exclu, j'ai choisi la loi normale et non la loi uniforme. Ce choix, vient du fait qu'elle permet de représenter la variabilité de nombreux phénomènes naturels (la glycémie à jeun, le taux de division bactérienne, etc.) (Elle fut découverte indépendamment par les mathématiciens Gauss en Allemagne (1809) et Laplace en France (1812)). Ainsi, comme nous étudions la température à la surface, d'une nouvelle étoile qui est un phénomène naturel, j'ai donc choisi l'utilisation de la normale, bien sûr comme certaines valeurs de la loi normale sont négatives ou supérieur à 1, j'ai crée une boucle afin d'éviter que la valeur de a soit en dehors de l'espace de définition.
Pour b et c, c'est simplement un **random.randint(1,20)** qui renvoie un entier entre 1 et 20 compris de manière aléatoire.
Enfin, la classe Individu a aussi la fonction Fitness de ce projet. Une fonction Fitness dans un algorithme génétique mesure la qualité de l'individu exprimée sous forme d'un nombre ou d'un vecteur. On dit qu'un individu i est meilleur que l'individu j quand i est plus proche de la solution que j.
Pour cette fonction au départ, comme dit précédemment, je l'ai crée sans faire partie d'aucune classe, cette fonction prenait un ensemble d'individus puis renvoyait sous forme de liste l'ensemble des comparaisons entre la somme totale des valeurs du fichiers CSV et la somme obtenu avec un triplet de la population, comme on peut le voir sur la photo ci-dessous :

```
def Fitness(Population): #On compare avec la somme avec les vrais valeurs a,b,c
    #a=0.14
    #b=19
    #c=15
    sommetheorique=0
    sommerelle=0
    ntheorique=0
    ensembledesvaleursthéoriques=[]
    ensembledescomparaison=[]
    indice=0
    df=LireCSV()
    i=0
    while indice < len(Population):
        for x in range(Population[indice+2]):
            sommetheorique = sommetheorique + (Population[indice]**x)*(cos((Population[indice+1]**x)
            ntheorique=ntheorique+1
        ensembledesvaleursthéoriques.append(sommetheorique)
        sommetheorique=0
        indice=indice+3
    i=0
    print(len(df))
    for y in range(int(len(df))):
        sommerelle = sommerelle + df["t"][y]
    i=0
    print(len(ensembledesvaleursthéoriques))
    print(int((len(Population)/3)))
    for z in range(len(ensembledesvaleursthéoriques)):
        ensembledescomparaison.append(abs(ensembledesvaleursthéoriques[z]-sommerelle))
    return ensembledescomparaison
```

Aussi pour information, la méthode **LireCSV()** est la méthode qui est associé au package **pandas** disponibles sur **Google Collab** ou **Spyder**, permet de lire le fichier CSV comme vous pouvez le voir ci-dessous :

```
def LireCSV():
    df=pd.read_csv("D:/ESILV/temperature_sample.csv",sep=';')
    return df
```

Pour la suite, seule la fonction Fitness a changée, comme vous pouvez le voir ci-dessous :

```
    def Fitness(self):
        ecart=0
        sommetheorique=0
        df=LireCSV()
        for i in range(len(df["#i"])):
            sommetheorique=0
            for x in range(self.c+1):
                sommetheorique = sommetheorique + (self.a**x)*(cos((self.b**x)*pi*df["#i"][i]))
            ecart+=abs(sommetheorique-df["t"][i])
        return ecart/len(df["t"])
```

Cette fonction qui est ma fonction actuelle permet de comparer un individu par rapport à chaque **T(i)** du fichier CSV et renvoie donc l'écart moyen entre le triplet étudié et les valeurs du fichier (donc le triplet qu'on doit approximer). L'écart est notamment obtenu grâce à la longueur du tableau contenant les valeurs **T(i)** du fichier CSV.

Pour finir, cette classe dispose aussi d'une fonction affichage :

```
    def __str__(self):
        return str(f"La valeur de a est :{self.a} , la valeur de b est  :{self.b} , et la valeur de c est  :{self.c}")
```

Nous allons donc passer à la partie de l'explication des méthodes qui sont en dehors de la classe Individu.

III Les méthodes Population et Evaluation

Comme précisé dans l'énoncé du projet, nous devons étudier des triplets de valeurs afin de se rapprocher des valeurs du fichier .csv donné en source sur **DVO**.

Dans un premier temps, pour étudier un ensemble de triplets, une classe Population a été crée afin de générer un ensemble d'individus. Tout d'abord, sans classe, je devais donc créer une méthode me permettant de générer selon un nombre de fois un triplet, ma première fonction rajoutait dans une liste, 100 fois un triplet en faisant appel à la fonction Individu (dont je fais référence dans la deuxième partie) ainsi j'avais donc une liste composé de 300 éléments propre :

```
def Population():
    i=0
    t=[]
    while i < 100:
        t.extend(Individu())
        i=i+1
    return t
```

Cependant, grâce à la classe Individu j'ai pu créer une fonction Population plus simple qui me permet selon un nombre donné de générer une population de telle ou telle taille :

```
def Population(nombre):
    return [Individu() for i in range(nombre)]
```

Ensuite, pour étudier les individus de la population créer je devais utiliser créer une fonction Evaluation qui permet d'étudier et de trier les Fitness de touts les individus afin de classer par ordre leurs qualités par rapport à ce que l'on recherche. Avant je n'avais pas créer de fonction Evaluation mais je faisais tout dans la méthode Selection (dont nous allons parler à la prochaine partie), cette fonction Evaluation se présente sous la forme suivante :

```
def Evaluation(population):
    population.sort(key=lambda x: x.Fitness())
    return population
```

Grâce à la fonction **lambda**, nous pouvons écrire très simplement cette méthode et optimiser la lisibilité et la complexité.

Nous allons donc passer à la partie de l'explication des méthodes qui nous permettent

de faire brasser mais aussi d'améliorer la qualité de nos individus.

IV Les méthodes Selection, Croisement et Mutation

Nous allons donc expliquer dans cette partie les opérateurs choisis afin de nous permettre d'arriver à approximer les valeurs voulus.

Commençons avec la création de la méthode Selection, qui dans la première version, créait lui même sa population puis triait la liste des fitness obtenues et enfin grâce aux fitness obtenues, il renvoyait les 5 meilleurs triplets. Cette méthode avait 2 importants problèmes, le premier est qu'elle prenait aucun paramètre et créait elle même sa population puis le 2ème qu'elle prenait que les 5 meilleurs or pour un meilleur brassage et évolution il ne faut pas délaisser les éléments les plus mauvais, cette fonction a été crée comme ceci :

```
def Selection():
    g=Population()
    t=Fitness(g)
    i=0
    h=list(t)
    h.sort()
    valeursSelectionnee=[]
    position=[]
    while i<5:
        valeursSelectionnee.append(h[i])
        i+=1
    i=0
    print(valeursSelectionnee)
    i=0
    a=0
    for z in range(5):
        a=0
        for a in range(len(t)):
            if(t[a]==valeursSelectionnee[z]):
                position.append(a)
    quintupletchoisie=[]
    i=0
    while i < 5 :
       quintupletchoisie.append(g[position[i]*3])
       quintupletchoisie.append(g[position[i]*3+1])
       quintupletchoisie.append(g[position[i]*3+2])
       i=i+1
    for a in range(len(quintupletchoisie)):
        print(quintupletchoisie[a])
    return quintupletchoisie
```

C’est pour ça que j’ai décidé de recommencer à zéro de créer une nouvelle méthode :

```
def Selection(population):
    selection=[]
    for i in range(5):
        selection.append(population[i])
    i=0
    i=len(population)-6
    while i<len(population):
        selection.append(population[i])
        i+=1
    return selection
```

Qui permet de prendre les 5 meilleurs mais aussi les 5 derniers afin de réaliser un meilleur brassage !, elle prend en paramètre une population et qui grâce à l’appel de la fonction Evaluation dans l’algorithme final permet de renvoyer une liste composé de 10 individus (les meilleurs et pires).

Ensuite, la création de la méthode Croisement, dans la 1ère version elle prenait en paramètre une population et aléatoirement modifier dans une nouvelle liste les triplets avec des valeurs a,b et c déjà présentes comme vous le pouvez voir ici :

```
def CrossOver2(individuselectionne):
    #On compare les trois premiers individu 0-1 0-2 et 1-2
    t=[]
    t=individuselectionne
    crossover=[]
    i=0
    a=[]
    b=[]
    c=[]
    compteur=0
    while i<len(t):
        a.append(t[i])
        b.append(t[i+1])
        c.append(t[i+2])
        i=i+3
    i=0
    z=0
    print(a)
    print("\n")
    print(b)
    print("\n")
    print(c)
    print("\n")
    while i<len(t):
        crossover.append(a[random.randint(0,len(a)-1)])
        crossover.append(b[random.randint(0,len(b)-1)])
        crossover.append(c[random.randint(0,len(c)-1)])
        if(Fitness(crossover)[z]<1):
           crossover.append(a[random.randint(0,len(a)-1)])
           crossover.append(b[random.randint(0,len(b)-1)])
           crossover.append(c[random.randint(0,len(c)-1)])
           compteur=compteur+1
        z=z+1
        i=i+3
    w=Fitness(crossover)
    print(compteur)
    return w
```

Sauf que, je me suis aperçu que cette méthode était fausse ou on pourra la considérer comme tronquer dans sa façon de réaliser un nouveau triplet, c'est pour cela que j'ai modifié en totalité cette méthode comme vous pouvez le voir ci-dessous :

```
def Croisement(individu1, individu2):
    c1=[]
    c2=[]
    value=random.randint(0,2)
    if(value==0):
        c1.append(individu1.a)
        c1.append(individu1.b)
        c1.append(individu2.c)
    if(value==1):
        c1.append(individu1.a)
        c1.append(individu2.b)
        c1.append(individu2.c)
    if(value==2):
        c1.append(individu2.a)
        c1.append(individu2.b)
        c1.append(individu1.c)
    croisement1=Individu(c1[0],c1[1],c1[2])
    value=random.randint(0,2)
    if(value==0):
        c2.append(individu1.a)
        c2.append(individu1.b)
        c2.append(individu2.c)
    if(value==1):
        c2.append(individu1.a)
        c2.append(individu2.b)
        c2.append(individu2.c)
    if(value==2):
        c2.append(individu2.a)
        c2.append(individu2.b)
        c2.append(individu1.c)
    croisement2=Individu(c2[0],c2[1],c2[2])
    return [croisement1,croisement2]
```

Cette nouvelle méthode prend en paramètre deux individus et renvoie une liste composé des deux individus nouvellement crées bien sûr même si on réutilise les mêmes ”parents” il y’aura 9 combinaisons diffférentes de couples d’individus ”enfants” en retour, grâce à la fonction **random.randint(0,2)** qui selon la valeurs va modifier la listes des deux individus nouvellement crées.

Enfin, la méthode Mutation, qui dans sa première version prenait en paramètre des

individus (les individus sélectionnés) et modifier de manière aléatoire leurs paramètres comme vous pouvez le voir ci-dessous :

```
def Mutation(individuselectionne):
    #On prend le meilleur et on peut voir si on change les valeurs si il devient meilleurs
    iteration =0
    t=[]
    t=individuselectionne
    nouveauquintuplechoisie=list(t)
    i=0
    a=[]
    b=[]
    c=[]
    while i<len(t):
        a.append(t[i])
        b.append(t[i+1])
        c.append(t[i+2])
        i=i+3
    i=0
    print(a)
    print("\n")
    print(b)
    print("\n")
    print(c)
    print("\n")
    while i<len(nouveauquintuplechoisie):
        nouveauquintuplechoisie[i]=a[random.randint(0,len(a)-1)]
        nouveauquintuplechoisie[i+1]=b[random.randint(0,len(b)-1)]
        nouveauquintuplechoisie[i+2]=c[random.randint(0,len(c)-1)]
        i=i+3
    w=Fitness(nouveauquintuplechoisie)
    return w
```

C'est ainsi, après m'être rendu de mes erreurs que j'ai modifié cette fonction comme vous le pouvez le voir :

```
def Mutation(individu):
    ind = Individu(individu.a,individu.b,individu.c)
    value=random.randint(0,2)
    if(value==0):
        ind.a = random.random()
    if(value==1):
        ind.b =random.randint(1,20)
    if(value==2):
        ind.c =random.randint(1,20)
    return ind
```

Qui prend en paramètre un individu et renvoie un nouveau individu crée et qui selon une valeur aléatoire (1/3 de chance) va modifier telle ou telle valeur de l'individu en paramètre pour renvoyer le nouveau individu crée.

Nous allons donc passer à la dernière partie qui est l'explication de l'algorithme final.

V La méthode Algo

Dans cette partie, nous allons expliquer l'algorithme finale, dans la première version il n'y avait pas cette méthode car je voulais tout inclure dans la méthode Selection (que vous pouvez voir dans la partie précédente). La méthode Algo que vous pouvez voir ci-dessous :

```python
def Algo(temps):
    if(temps==None):
      temps=20
    debut = time.time()
    #solutions=[]
    iteration=0
    taillepopulation=20
    population=Population(taillepopulation)
    a=time.time()
    while time.time() - a < temps:
        iteration+=1
        population=Evaluation(population)
        print(population[0])
        newpopulation=Selection(population)
        for i in range(0,len(newpopulation)-1,2):
            newpopulation.extend(Croisement(newpopulation[i],newpopulation[i+1]))
        for i in range(0,random.randint(1,len(newpopulation))):
            newpopulation.append(Mutation(newpopulation[i]))
        population+=newpopulation+Population(5)
    print("\n")
    print("Meilleur Fonction fitness :",population[0].Fitness(), "(qui correspond à la valeur moyenne de l'écart entre les valeurs réelles et les valeurs du meilleur t
    print("Avec une population de :",taillepopulation,"individus (triplets).")
    print("Il y'a eu :",iteration,"d'itérations pour aboutir à ce résultat.")
    print("Le meilleur triplet abc est :",round(population[0].a,2),",",population[0].b,"et",population[0].c,".")
    fin = time.time()
    print("Temps :",fin-debut,"secondes entre l'ouverture de la fonction et la fin.")
```

Prend un paramètre un entier, cet entier correspond au temps que vous souhaitez que l'algorithme tourne dans sa boucle **While** et ainsi constitue sa conditions d'arrêt et à environ 1 seconde près le temps écoulé depuis l'ouverture de la fonction (qui est aussi retourné), tout ceci grâce à la fonction **time**. Ensuite, on initialise la taille de la population puis aussi un nombre d'itérations réalisés dans cette méthode ainsi qu'une population. Puis, dans la boucle **While** nous allons ajouter successivement dans une nouvelle liste créée les résultats des méthodes Croisement et Mutation et ainsi rajouter ceci dans la population avec notamment une nouvelle population réduites (composé de 5 individus). A la fin du temps écoulé on retourne ceci :

```
Meilleur Fonction fitness : 0.036864645897240235 (qui co
Avec une population de : 20 individus (triplets).
Il y'a eu : 12 d'itérations pour aboutir à ce résultat.
Le meilleur triplet abc est : 0.35 , 15 et 2 .
Temps : 10.396415710449219 secondes entre l'ouverture de
```

Avec notamment le retour de la fonction Fitness du meilleur triplet et aussi le meilleur triplet. Enfin, comme vous pouvez le voir ci-dessous, les valeurs a,b et c convergent vers les valeurs a=0.35, b=15 et c=2 :

```
Algo(10)

La valeur de a est :0.26377646611630123, la valeur de b est :17, et la valeur de c est :19
La valeur de a est :0.26377646611630123, la valeur de b est :17, et la valeur de c est :19
La valeur de a est :0.17047276016003418, la valeur de b est :15, et la valeur de c est :6
La valeur de a est :0.17047276016003418, la valeur de b est :15, et la valeur de c est :6
La valeur de a est :0.2872428225741276, la valeur de b est :15, et la valeur de c est :6
La valeur de a est :0.2872428225741276, la valeur de b est :15, et la valeur de c est :6
La valeur de a est :0.3478428087684401, la valeur de b est :15, et la valeur de c est :6
La valeur de a est :0.3478428087684401, la valeur de b est :15, et la valeur de c est :2
La valeur de a est :0.3478428087684401, la valeur de b est :15, et la valeur de c est :2
La valeur de a est :0.3478428087684401, la valeur de b est :15, et la valeur de c est :2
La valeur de a est :0.3478428087684401, la valeur de b est :15, et la valeur de c est :2
La valeur de a est :0.3478428087684401, la valeur de b est :15, et la valeur de c est :2
La valeur de a est :0.3478428087684401, la valeur de b est :15, et la valeur de c est :2


Meilleur Fonction fitness : 0.03684077479738586 (qui correspond à la valeur moyenne de l'écart entre les valeurs réelles et les valeurs du meilleur triplet (a, b et c)) trouvée durant
Avec une population de : 20 individus (triplets).
Il y'a eu : 13 d'itérations pour aboutir à ce résultat.
Le meilleur triplet abc est : 0.35 , 15 et 2 .
Temps : 10.82547116279602 secondes entre l'ouverture de la fonction et la fin.
```

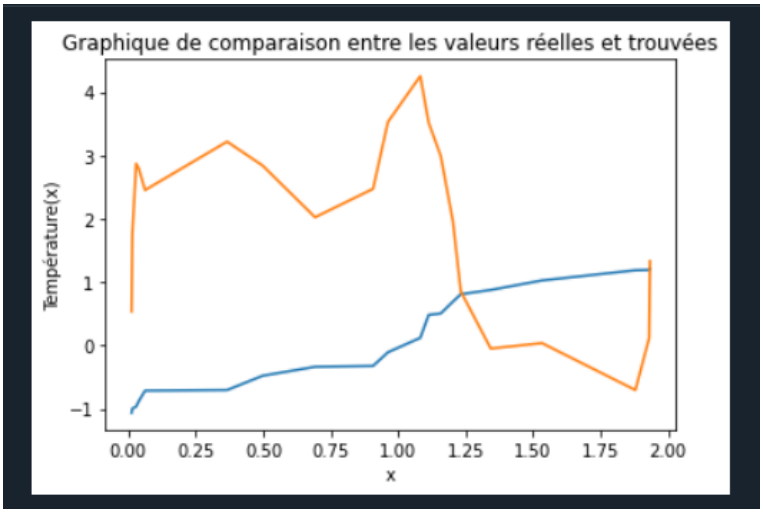
Innovations Nous allons donc passer à la partie Innovations.

VI Les Innovations

Dans un cadre de recherche, j'ai crée une méthode permettant d'afficher sur un graphique la comparaison entre les valeurs réelles et trouvées, cette fonction a été crée comme ceci :

```
def GraphiqueComparaison(individu):
    sommetheorique=0
    ensembledesvaleurstheoriques=[]
    df=LireCSV()
    for i in range(len(df["#i"])):
        for x in range(individu.c):
            sommetheorique = sommetheorique + (individu.a**x)*(cos((individu.b**x)*pi*df["#i"][i]))
        ensembledesvaleurstheoriques.append(sommetheorique)
    X=list(df["#i"])
    X.sort()
    Y=list(df["t"])
    Y.sort()
    plt.title("Graphique de comparaison entre les valeurs réelles et trouvées")
    plt.plot(X,Y)
    plt.plot(X,ensembledesvaleurstheoriques)
    plt.xlabel('x')
    plt.ylabel('Température(x)')
    plt.show()
```

et affiche :



Nous allons donc passer à la conclusion.

VII Conclusion et Critiques

Pour conclure, il y'a eu de nombreux problèmes dans l'écriture et tests des méthodes, entre les **sorties d'index**, l'efficacité des méthodes ou comme le fait que la première version avait des méthodes et fonctions qui n'avaient pas de lien entre eux (ou fonctionnaient mal entre eux) d'où des incohérences et une non fonctionnalité. Les problèmes étaient nombreux mais pas sans solutions, le projet à pu être construit correctement.