GBENOU Merveille
JUGAND Tess

Université Paris Nanterre

PROJET – PROGRAMMATION AVANCÉE

IMPLÉMENTATION DE DEUX MÉTAHEURISTIQUES

Sommaire

Introduction

Ce projet s'inscrit dans le cadre de notre Master 1 IES – D3S (Innovation Entreprise et Société – Data Science for Social Sciences) dans l'enseignement « Programmation avancée ». Il consiste à la modélisation, à partir d'un jeu de données, de deux métaheuristiques, c'est-à-dire deux algorithmes d'optimisation visant à résoudre des problèmes complexes.

Ce projet s'effectue en langage de programmation Python.

Notre objectif est donc de trouver un optimum global pour notre jeu de données en nous familiarisant avec Python et les deux métaheuristiques proposées, à savoir :

- L'évolution différentielle : elle s'appuie sur l'algorithme génétique, avec les mêmes étapes (Sélection, Croisement, Mutation), et les stratégies évolutionnistes, avec des manipulations géométriques.
- L'apprentissage incrémental à base de population (ou « *PBIL : Population-based Incremental Learning*) : c'est également un type d'algorithme génétique où le génotype d'une population entière (vecteur de probabilité) évolue plutôt que des membres individuels.

Dans la première partie de notre rapport, nous décrirons le plan de nos fichiers de code de manière globale ainsi que la méthode pour lancer notre application. Deuxièmement, nous ferons une description de notre jeu de données choisi. Pour notre troisième partie, nous présenterons et analyserons les contenances des fichiers de code Python plus en détail ainsi que leurs résultats en fonction des paramètres choisis. Dans un quatrième temps nous comparerons les deux algorithmes entre eux et nous conclurons sur la meilleure méthode.

1. Plan et méthode

Pour les fichiers de code Python, il y en a deux : un pour chaque algorithme. Le plan est le même pour les deux fichiers, dans un premier temps nous implantons les packages et les algorithmes nécessaires (dont certains ont été vu en cours), ensuite nous codons l'algorithme principale d'optimisation et enfin la dernière partie est consacrée au choix des paramètres, au lancement de l'algorithme et à l'affichage des résultats.

Pour installer et lancer notre application, il faut avoir au préalable télécharger le langage Python sur votre ordinateur avec une IDE (*Intergrated Development and learning Environment*) comme PyCharm ou Spyder. Ensuite, il faut sauvegarder le fichier « projetinfo » contenant notre jeu de donnée dans le même dossier que les scripts ou savoir retrouver facilement le chemin conduisant au dossier dans lequel celui est enregistré, cette subtilité est détaillée dans les scripts lors de la fonction *main*. Enfin, il est nécessaire d'effectuer la vérification que tous les packages utiles sont déjà installés, à savoir :

- *pickle*
- *random*
- *numpy*
- *pandas*
- *heapq*
- *time*
- *matplotlib.pyplot*
- *timedelta from datetime*
- *LogisticRegression from sklearn.linear_model*
- *confusion_matrix from sklearn.metrics*
- *recall_score from sklearn.metrics*
- *model_selection from sklearn*
- *mean from statistics*

Pour lancer les programmes, il suffit d'appuyer sur Run. Lors de l'affichage vous aurez, pour chaque génération, le meilleur score ainsi que le temps d'exécution et deux graphiques regroupant les données des temps d'exécution pour l'un et les meilleurs scores pour l'autre, en fonction des générations.

2. Description du jeu de données

Notre jeu de données, s'intitulant « projetinfo », vient de la plateforme OpenML (*Open Media Library*) qui regroupe des jeux de données ouverts au public. Notre dataset a été créé par Nicholas Kushmerick en 1998.

Il représente un ensemble de publicités possibles sur internet, et plus précisément la géométrie de l'image (si elle est disponible) ainsi que les phrases apparaissant dans l'URL, le texte alt et ancrage de l'image ainsi que les mots apparaissant près du texte d'ancrage. Le but de ce jeu consiste à prédire si à partir des caractéristiques d'une image, est-ce que celle-ci est une publicité ("ad") ou non ("nonad").

Notre jeu de données contient 3279 observations, soit 3279 images analysées.

Il y a également 1558 variables, autrement dit les caractéristiques. Pour en citer quelques-unes, nous avons :

Nom de la variable	**Signification**	**Nature**
Height	Hauteur de l'image	Quantitative, Numérique
Width	Largeur de l'image	Quantitative, Numérique
Url.sunsetstrip.alley	Est-ce que cette phrase apparaît dans l'URL ? - 0 : Non - 1 : Oui	Qualitative, Nominale
Alt.logo	Est-ce que ce texte alt est présent ? - 0 : Non - 1 : Oui	Qualitative, Nominale

On remarquera que dans notre dataset, il y a 3 attributs continues, les autres étant discrets.

Il y a également notre variable qu'on cherche à prédire qui est la variable « class », elle est qualitative nominale et comporte deux possibilités :

- Ad (l'image a été prédite comme une pub)
- Nonad (ce n'est pas une pub)

Si on observe sa distribution, on aperçoit une grande majorité de *nonad*, 2 820 images ne sont pas des pubs, contre 459 *ad*.

3. Les algorithmes d'optimisation

A. Algorithme 1 : Evolution différentielle

- ### Analyse détaillée du fichier code

Tout d'abord, avant de rentrer dans les détails des différents algorithmes utilisés, nous définissons les paramètres, c'est-à-dire les éléments qu'on utilise lors de l'appel des diverses fonctions, afin d'avoir une meilleure compréhension du programme.

On a donc :

- *filename* : nom du fichier comprenant les données, ici ce sera « projetinfo.csv »
- *data* : notre dataset contenant toutes les données du fichier Excel original
- *target* : variable cible, ici elle s'appelle « class »
- *nfold* : nombre de *folder* qu'on souhaite pour la validation croisée
- *X* : dataset avec les valeurs de toutes les variables sauf la variable cible
- *Y* : dataset avec les valeurs de seulement la variable cible
- *model* : régression logistique
- *matrix* : matrice de taille 2x2 initialisée comme nulle et qui va comporter la somme des matrices de confusion
- *taille_pop* : taille de la population
- *taille_var* : nombre de variables par rapport aux dimensions des données
- *ind* : individu
- *d* : pour l'évolution différentielle, d = data (attention, dans l'autre métaheuristique PBIL ce n'est pas le cas)
- *pop* : population créée
- *number_of_choices* : nombre d'individus qu'on veut choisir de manière aléatoire
- *F* : facteur de mutation
- *R* : ici R = choix_indiv()
- *mutants* : la sortie de la fonction *create_mutants*
- *CR* : probabilité de croisement
- *G* : nombre de générations au total

Les paramètres *nfold, number_of_choices, F, CR, taille_pop, G* sont à définir par l'utilisateur.

Pour l'implantation de cet algorithme, nous avons eu besoin de plusieurs algorithmes vus en cours, à savoir :

Noms	**Fonctions**	**Paramètres**
read	Lire le fichier « projetinfo » avec ses données	*filename*
learning	Faie un apprentissage	*data, target*
cross_validation	Validation croisée pour garantir que l'apprentissage soit représentatif de l'ensemble des données	*nfold, X, Y, model, matrix*

create_population	Créer une population d'individus	*taille_pop, taille_var*
preparation	Sélectionner des colonnes en fonction de la valeur d'un individu	*data, ind, target*
fitness	Calcul du score pour chacun des individus	*d, pop, target*

Puis, nous avons ajouté d'autres algorithmes nécessaires, tels que :

- *choix_indiv* : cet algorithme a pour but de choisir des individus dans la population de manière aléatoire. Pour ce faire, cette fonction prend en paramètre : pop, number_of_choices, taille_pop, taille_var. On commence par redéfinir la taille de la population attendue, avec la dimension des mutants. On procède ensuite à la manipulation des lignes en choisissant des indices au hasard et tous différents pour avoir nos lignes choisies qui représenteront nos individus choisis aléatoirement.

- *create_mutants* : ici le but est de créer nos mutants. Comme paramètres, on trouve : pop, taille_pop, taille_var, F et R. Tout d'abord, on récupère les individus pour la mutation. Puis on convertit ces booléens récupérés en binaire (0 ou 1), on aura donc : r_1, r_2 et r_3. Ensuite, on calcule les mutants avec la formule : $m_j = r_{j1} + F * (r_{j2} - r_{j3})$. On convertit ensuite les valeurs supérieures à 1 en 1 et celles inférieures à 0 en 0, puis on reconvertit dans l'autre sens, c'est-à-dire en booléens. On obtient notre liste de mutants.

- *croisement* : c'est notre fonction de croisement avec les paramètres suivants : CR, pop, taille_pop, taille_var, mutants. Le but de ce croisement est d'augmenter la diversité des individus dans notre population avec un nouveau vecteur u. Sachant que r est un nombre aléatoire choisi de façon uniforme entre 0 et 1, le vecteur u est calculé avec : $u_{i,j} = m_{i,j}$ si r <= CR et sinon : $u_{i,j} = v_{i,j}$ avec v le vecteur de la population initial et m le vecteur des mutants.

Enfin, avec ces 9 algorithmes, nous pouvons former notre métaheuristique qui prend en paramètre : data, target, CR, G, F, taille_pop. La première étape de l'évolution différentielle est de créer les deux fichiers texte de sorties dans le répertoire EVOLUTION_DIFF_FILES, le chemin précis est détaillé dans le fichier Python. Ces deux fichiers contiendront pour l'un les moyennes des scores des meilleures individus de chaque génération et le meilleur individu toutes générations confondues pour l'autre. Tout le long de l'algorithme, le temps d'exécution de chaque génération est également retenu. Pour les étapes suivantes de création, mutation et croisement, nous avons suivi les étapes définies dans les consignes, à savoir, dans cet ordre :

- Définition du nombre de variable
- Création de la population initiale « pop »
- Évaluation de cette population initiale
- Calcul de la moyenne des scores sur la population
- Récupération du meilleur score et du meilleur individu de la population

Puis on va incrémenter le nombre de générations de 1 en 1 en répétant le même processus jusqu'à arriver à notre nombre de générations totale G, c'est-à-dire :

- Sélection des individus de la population
- Création de la population de mutants
- Croisement entre les mutants et la population initiale
- Évaluation de la nouvelle population issue du croisement
- Comparaison du score de la nouvelle population et de l'ancienne afin de ne garder que le meilleur
- Calcul de la moyenne des scores
- Récupération du meilleur individu de la nouvelle population pour la sauvegarde

Enfin, la métaheuristique se termine par l'affichage des deux graphiques concernant l'évolution du temps d'exécution et celle du score du meilleure individue par génération.

• Résultats et interprétations

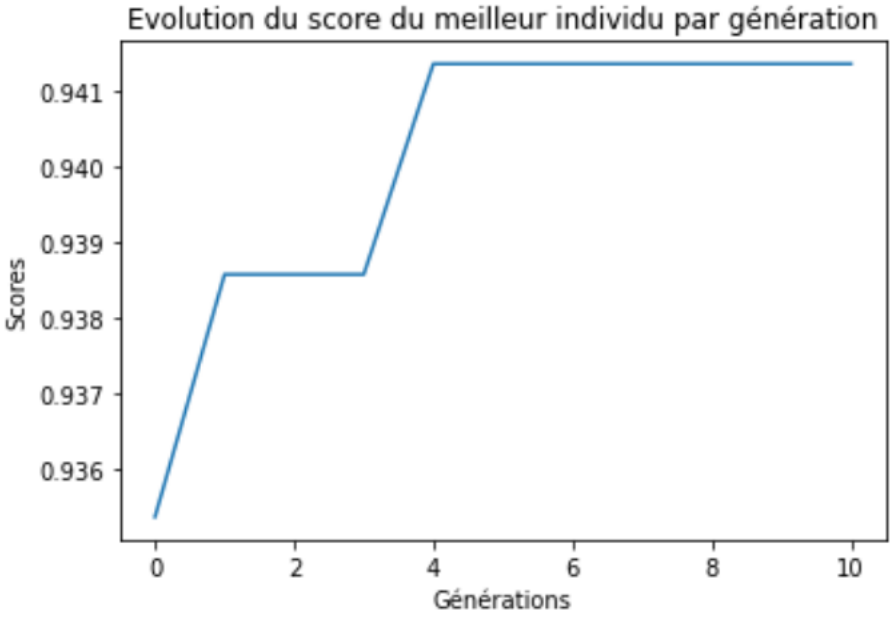
Par défaut, nous avons choisi les paramètres suivants :

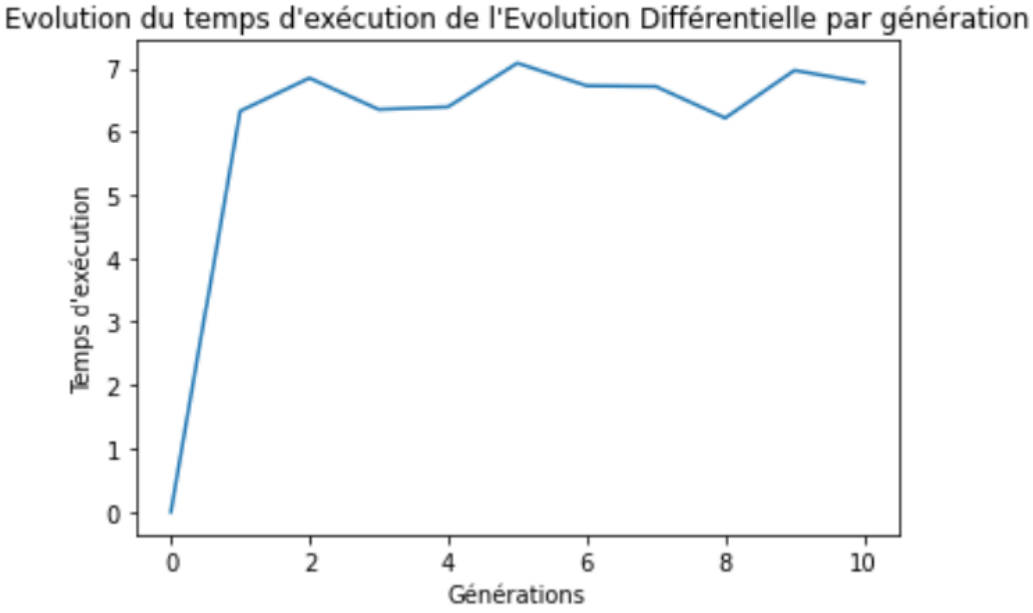
- *F* = 1
- *G* = 10
- *CR* = 0.5
- *taille_pop* = 10
- *number_of_choices* = 3
- *nfold* = 10

On obtient les résultats suivants :

```
génération : 1, meilleur score : 0.9385813721834027, temps: 0:00:06.326228
génération : 2, meilleur score : 0.9385813721834027, temps: 0:00:06.845844
génération : 3, meilleur score : 0.9385813721834027, temps: 0:00:06.351952
génération : 4, meilleur score : 0.9413722232871169, temps: 0:00:06.393184
génération : 5, meilleur score : 0.9413722232871169, temps: 0:00:07.080938
génération : 6, meilleur score : 0.9413722232871169, temps: 0:00:06.725346
génération : 7, meilleur score : 0.9413722232871169, temps: 0:00:06.713450
génération : 8, meilleur score : 0.9413722232871169, temps: 0:00:06.217588
génération : 9, meilleur score : 0.9413722232871169, temps: 0:00:06.966886
génération : 10, meilleur score : 0.9413722232871169, temps: 0:00:06.776268
```

Avec les graphiques :





Malgré un petit nombre de générations (=10), on peut quand même en tirer certaines conclusions :

- Le temps d'exécution de l'algorithme n'évolue pas vraiment au fil des générations, il se stabilise entre 6 et 7 secondes, ce qui est relativement rapide.
- L'évolution du score est impactée par le temps, puisqu'on observe des scores de plus en plus élevés : on commence avec 0.9386 et on finit avec 0.9414.
  Ce sont des scores relativement élevés et proches de 1, donc très bons.

Dans le fichier texte *fichier_meilleur_ttes_generations*, on obtient le meilleur individu associé au meilleur score de toutes générations confondues qui est d'environs 0.9414, qui correspond à la génération 4.

Cependant, avec seulement 10 générations, on s'aperçoit que les résultats varient d'une simulation à une autre, donc pour avoir une vision plus claire nous allons faire tourner la métaheuristiques avec 200 générations.

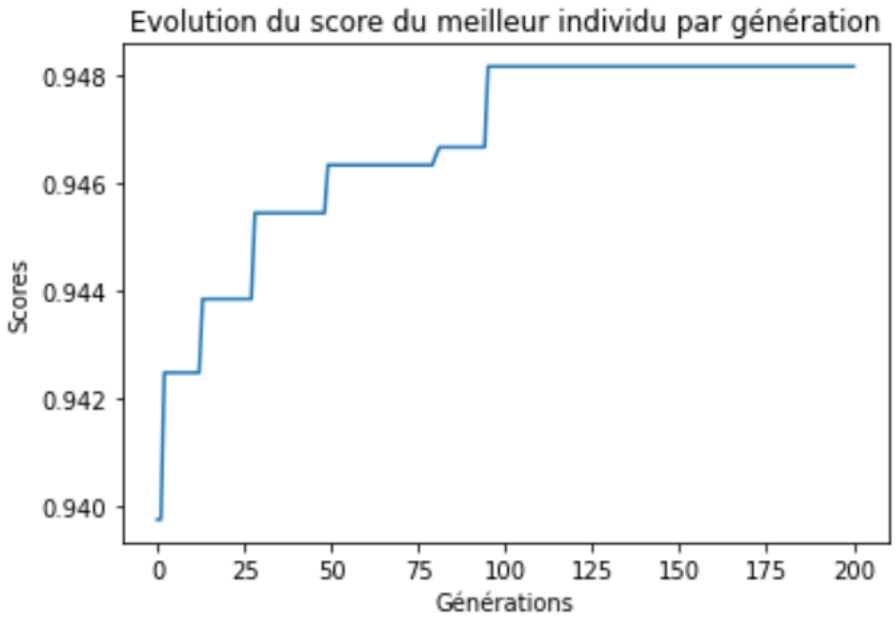
Premiers résultats :

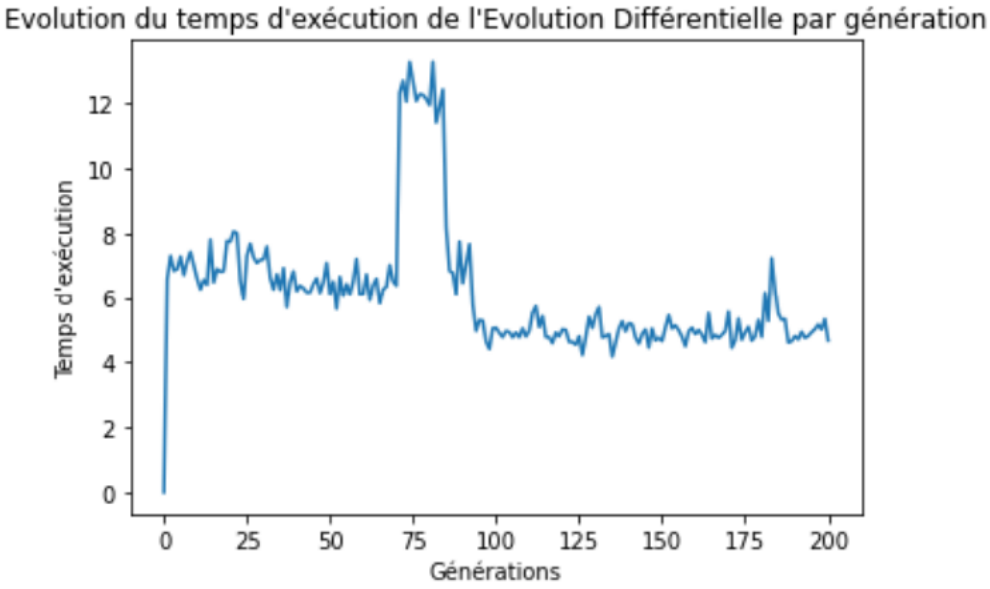
```
génération : 1, meilleur score : 0.9397355492686592, temps: 0:00:06.591615
génération : 2, meilleur score : 0.9424621799624013, temps: 0:00:07.310907
génération : 3, meilleur score : 0.9424621799624013, temps: 0:00:06.851043
génération : 4, meilleur score : 0.9424621799624013, temps: 0:00:06.903729
génération : 5, meilleur score : 0.9424621799624013, temps: 0:00:07.291501
génération : 6, meilleur score : 0.9424621799624013, temps: 0:00:06.718771
génération : 7, meilleur score : 0.9424621799624013, temps: 0:00:07.136961
génération : 8, meilleur score : 0.9424621799624013, temps: 0:00:07.431517
génération : 9, meilleur score : 0.9424621799624013, temps: 0:00:07.023481
génération : 10, meilleur score : 0.9424621799624013, temps: 0:00:06.603773
```

Derniers résultats :

```
génération : 190, meilleur score : 0.9481461470620991, temps: 0:00:04.803148
génération : 191, meilleur score : 0.9481461470620991, temps: 0:00:04.702418
génération : 192, meilleur score : 0.9481461470620991, temps: 0:00:04.937788
génération : 193, meilleur score : 0.9481461470620991, temps: 0:00:04.750290
génération : 194, meilleur score : 0.9481461470620991, temps: 0:00:04.810130
génération : 195, meilleur score : 0.9481461470620991, temps: 0:00:04.920835
génération : 196, meilleur score : 0.9481461470620991, temps: 0:00:05.017573
génération : 197, meilleur score : 0.9481461470620991, temps: 0:00:05.162188
génération : 198, meilleur score : 0.9481461470620991, temps: 0:00:05.006608
génération : 199, meilleur score : 0.9481461470620991, temps: 0:00:05.332730
génération : 200, meilleur score : 0.9481461470620991, temps: 0:00:04.669534
```

Graphiques :





Ici, nous pouvons faire les mêmes remarques que précédemment, avec l'avantage qu'on a une vision plus globale de notre score qui augmente en escaliers. En effet, il commence à 0.9397 puis finit à 0.9481, ce qui est encore mieux que le score obtenu avec 10 générations qui était de 0.9414.

Le temps d'exécution n'a quant à lui guère changer avec le temps, même si on peut tout de même remarquer quelques observations :

- Avant la génération 75, le temps d'exécution est entre 6 et 8 secondes
- Entre les générations 75 et 85 il augmente quasiment du double pour se situer autour de 12 secondes
- Puis après la 85ème génération et jusqu'à la fin il descend un peu pour être entre 4 et 5 secondes, ce qui est très rapide.

Dans le fichier texte *fichier_meilleur_ttes_generations*, on obtient le meilleur individu associé au meilleur score de toutes générations confondues qui est d'environs : 0.9481, qui correspond à la génération 95. Donc le score n'a pas évolué depuis cette même génération.

B. Algorithme 2 : PBIL

- ### Analyse détaillée du fichier code

Pour cette deuxième métaheuristique, les paramètres et les fonctions utilisés sont quasiment les mêmes, à quelques exceptions près.

Pour les paramètres utilisés lors de l'appel des fonctions, on trouve en plus :

- *length* : longueur du vecteur de probabilité
- *p0* : probabilité initiale
- *nb_var* : équivalent à notre *taille_var* dans l'algorithme précédent
- *probas* : vecteur de probabilité
- *n* : nombre d'individus dans lesquels on cherche le meilleur individu
- *score_list* : sortie de la fonction fitness, donc c'est le score pour chacun des individus
- *best_indiv* : meilleur individu obtenu de la population
- *LR* : taux d'apprentissage
- *new_probas* : nouveau vecteur de probabilité créer à partir de MS
- *MS* : l'impact qu'aura la mutation sur le vecteur
- *MP* : probabilité d'effectuer une mutation sur le vecteur de probabilité au cours d'une génération

Ensuite, on utilise ici une autre fonction vue en cours : *selection*. Cette fonction a comme paramètres pop, score_list, n. Elle cherche à obtenir le meilleur individu de la population sur n individus, tout d'abord en récupérant l'indice des n meilleurs scores puis en récupérant le plus haut de ceux-là.

Et d'autres implantées en plus pour le bon fonctionnement de cet algorithme :

- *vectprobas* : le but de cette fonction est de définir notre vecteur de probabilité *probas*. Elle prend en paramètre : length, p0. On initie simplement notre vecteur avec notre probabilité p0.

- *maj_vect_probas* : ici, nous mettons à jour notre vecteur de probabilité à partir du meilleur individu avec les paramètres : probas, best_indiv et LR. Pour ce faire, on implémente la fonction suivante : $probas_i = \big(probas_i * (1.0 - LR)\big) + (LR * \ best\_indiv)$.

- *mutation* : ceci définit la mutation du nouveau vecteur de probabilité à partir du paramètre MS, on utilise : new_probas, MS, avec la formule :

$$new\_probas_i = \Big(new_{probas_i} * (1.0 - MS)\Big) + (MS * random.choice([0,1]) * MS)$$

- *choix_mutation* : dans ce dernière algorithme, on détermine aléatoirement si on effectue une mutation ou non avec le paramètre MP. On choisit entre True ou False de manière aléatoire.

Après avoir eu toutes ces informations, on peut maintenant expliquer notre deuxième métaheuristique PBIL qui prend en paramètre : notre dataset, target, LR, MS, MP, taille_pop, G. La première étape de notre algorithme est, comme pour le précédent, de créer les deux fichiers texte de sorties dans le répertoire PBIL_FILES, le chemin précis est détaillé dans le fichier Python. Ensuite, après avoir initialiser le vecteur de probabilité et on procède comme suit, avec une incrémentation de 1 par 1 pour chaque génération :

- Création de la population
- Evaluation de cette population
- Récupérer le meilleur individu
- Mettre à jour notre vecteur de probabilité
- Effectuer une mutation (ou non)
- Comparer le score de la nouvelle population et de l'ancienne et garder le meilleur

De plus, comme pour l'évolution différentielle, on sauvegarde les moyennes des scores des meilleures individus de chaque génération, le meilleur individu toutes générations confondues et le temps d'exécution de chaque génération. Enfin, on affiche les graphiques.

• Résultats et interprétations

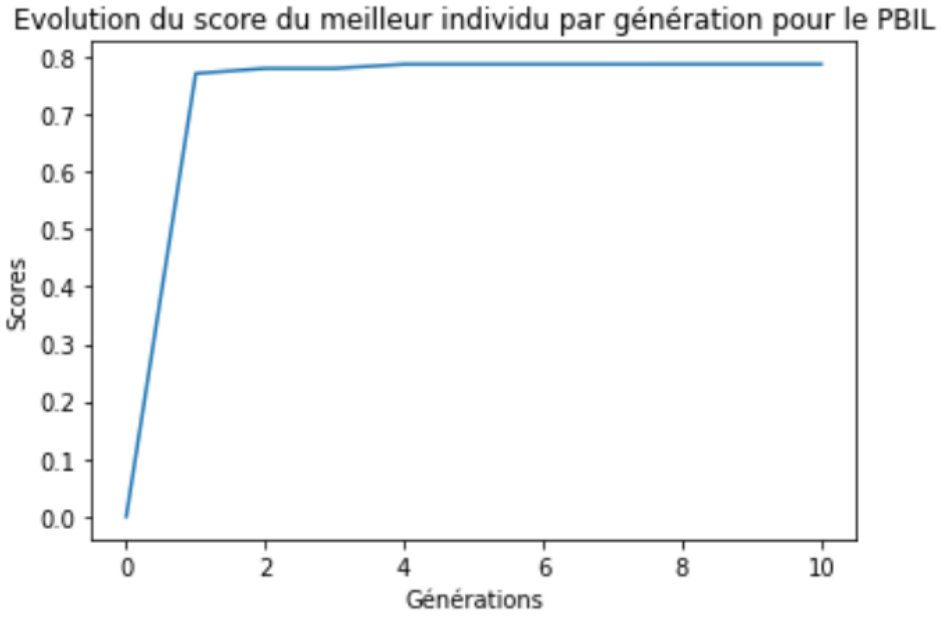
Par défaut, nous avons choisi les paramètres suivants :

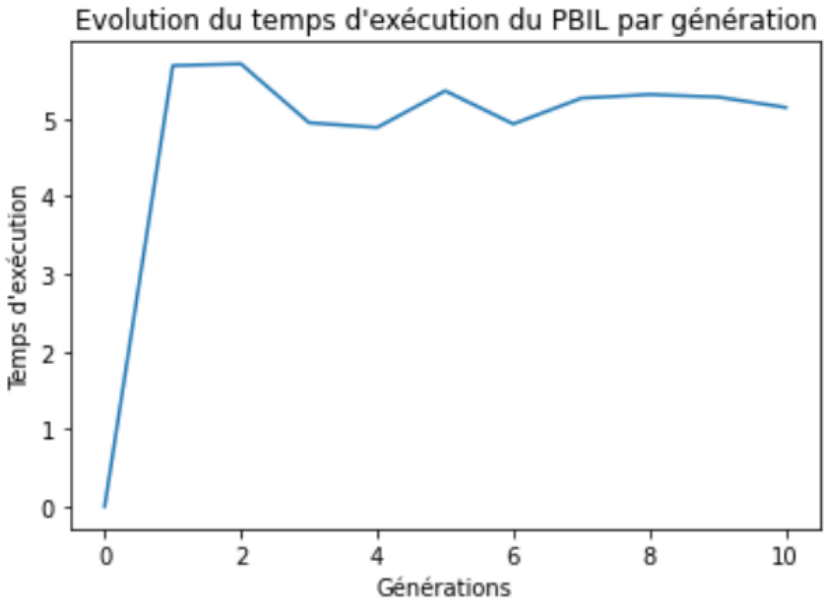
- n = 1
- p0 = 0.5
- LR = 0.1
- MS = 0.2
- MP = 0.2
- taille_pop = 10
- G = 10

On obtient :

```
génération : 1, meilleur score : 0.7704062949056691, temps: 0:00:05.688778
génération : 2, meilleur score : 0.7792981968200992, temps: 0:00:05.711718
génération : 3, meilleur score : 0.7792981968200992, temps: 0:00:04.951753
génération : 4, meilleur score : 0.7865688592221759, temps: 0:00:04.889915
génération : 5, meilleur score : 0.7865688592221759, temps: 0:00:05.363651
génération : 6, meilleur score : 0.7865688592221759, temps: 0:00:04.935792
génération : 7, meilleur score : 0.7865688592221759, temps: 0:00:05.268904
génération : 8, meilleur score : 0.7865943540536782, temps: 0:00:05.315776
génération : 9, meilleur score : 0.7865943540536782, temps: 0:00:05.283863
génération : 10, meilleur score : 0.7865943540536782, temps: 0:00:05.149223
```

Avec les graphiques :





On peut en déduire :

- Le temps d'exécution n'évolue pas vraiment avec le temps et est compris entre 4 et 6 secondes.
- Le score évolue avec le temps, en effet il augmente de quelques millièmes de seconde par génération en commençant à 0.7704 et finissant à 0.7866, ce qui représente un bon score.

Dans le fichier texte *fichier_meilleur_ttes_generations*, on obtient le meilleur individu associé au meilleur score de toutes générations confondues qui est d'environs : 0.7866, qui correspond à la génération 4.

Comme pour la métaheuristique précédente, nous allons maintenant faire tourner l'algorithme pendant 200 générations.

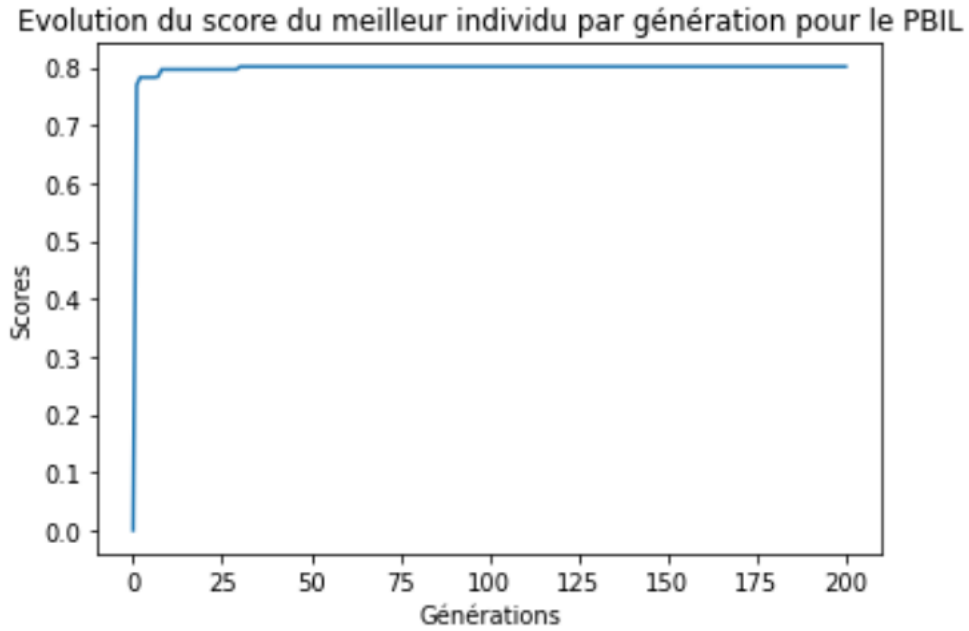
Premiers résultats :

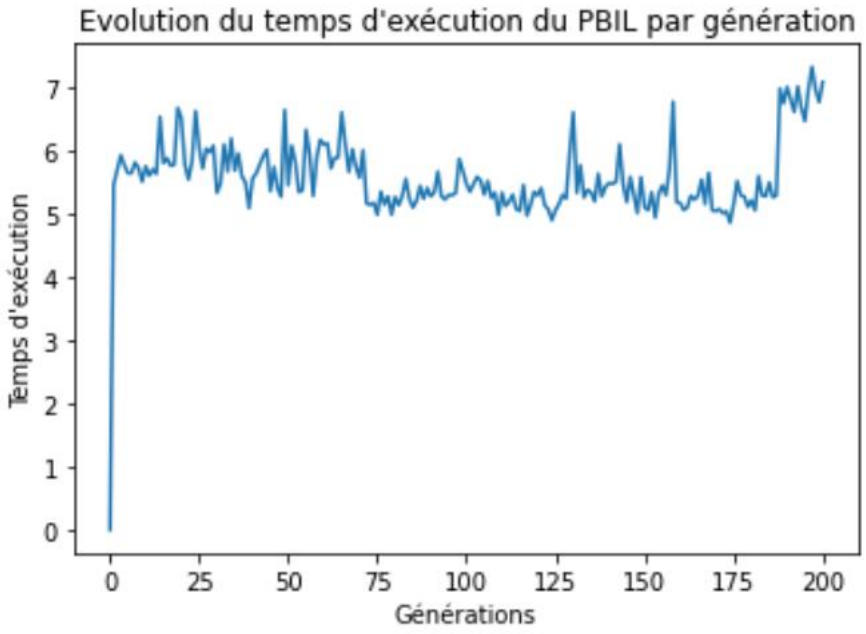
```
génération : 1, meilleur score : 0.7696715802160108, temps: 0:00:05.450380
génération : 2, meilleur score : 0.7829207806053864, temps: 0:00:05.681798
génération : 3, meilleur score : 0.7829207806053864, temps: 0:00:05.914178
génération : 4, meilleur score : 0.7829207806053864, temps: 0:00:05.760587
génération : 5, meilleur score : 0.7829207806053864, temps: 0:00:05.633926
génération : 6, meilleur score : 0.7829207806053864, temps: 0:00:05.634923
génération : 7, meilleur score : 0.7840101052241228, temps: 0:00:05.801477
génération : 8, meilleur score : 0.7965500857553424, temps: 0:00:05.713712
génération : 9, meilleur score : 0.7965500857553424, temps: 0:00:05.500284
génération : 10, meilleur score : 0.7965500857553424, temps: 0:00:05.742637
```

Derniers résultats :

```
génération : 190, meilleur score : 0.8014647939554072, temps: 0:00:06.994638
génération : 191, meilleur score : 0.8014647939554072, temps: 0:00:06.800517
génération : 192, meilleur score : 0.8014647939554072, temps: 0:00:06.602962
génération : 193, meilleur score : 0.8014647939554072, temps: 0:00:07.002207
génération : 194, meilleur score : 0.8014647939554072, temps: 0:00:06.671189
génération : 195, meilleur score : 0.8014647939554072, temps: 0:00:06.458637
génération : 196, meilleur score : 0.8014647939554072, temps: 0:00:06.963156
génération : 197, meilleur score : 0.8014647939554072, temps: 0:00:07.312696
génération : 198, meilleur score : 0.8014647939554072, temps: 0:00:06.947519
génération : 199, meilleur score : 0.8014647939554072, temps: 0:00:06.755754
génération : 200, meilleur score : 0.8014647939554072, temps: 0:00:07.070145
```

Graphiques :





On obtient les mêmes conclusions que précédemment, avec un temps d'exécution rapide compris entre 5 et 7 secondes, et un score qui augmente très vite et puis qui stagne à 0.8015.

Dans le fichier texte *fichier_meilleur_ttes_generations*, on obtient le meilleur individu associé au meilleur score de toutes générations confondues qui est d'environs : 0.8015, qui correspond à la génération 30. Donc le score n'a pas changé depuis.

Comparaison et conclusion

Dans cette partie, nous allons comparer nos deux métaheuristiques tout d'abord avec leurs meilleurs scores, puis par leurs temps d'exécutions et conclure.

Premièrement, les deux algorithmes ont des bons scores proches de 1, donc on peut déjà dire qu'ils fonctionnent bien tout les deux. Cependant, l'algorithme 1, à savoir l'évolution différentielle, renvoie un meilleur score : 94.81% comparé à l'algorithme 2 d'apprentissage incrémental à base de population qui est nous donne un score de : 80.15%. Même si l'algorithme 2 atteint plus vite son score « stable », il reste moins fort que celui de l'algorithme 1. Ainsi, par ce critère de comparaison, l'évolution différentielle est une meilleure méthode que l'algorithme PBIL pour notre jeu de données.

Dans un deuxième temps, regardons les temps d'exécutions. Pour l'évolution différentielle, on observe une légère baisse du temps d'exécution qui se stabilise entre 4 et 5 secondes. Avec l'apprentissage incrémental, on observe un temps d'exécution plutôt constant qui oscille entre 5 et 7 secondes. Ainsi, bien que la différence soit minime, on remarque que l'évolution différentielle tourne plus vite que l'autre algorithme. Donc, encore une fois, l'évolution différentielle est ici mieux adaptée.

Pour conclure, nous avons vu qu'avec nos deux critères de meilleurs scores et de temps d'exécution rapides, l'évolution différentielle convient le mieux à notre jeu de données.

On pourrait élargir notre réflexion en se demandant s'il n'existerait pas une métaheuristique avec les mêmes scores et temps que ceux de l'évolution différentielle, mais qui atteindrait ce « meilleur score » plus rapidement (comme l'algorithme PBIL qui pour 200 générations avait atteint son meilleur score à la génération 30). En somme, on pourrait peut-être trouver une métaheuristique qui garderait nos résultats mais qui optimiserait cette méthode.