



Rapport de mini projet

Algorithmes de perceptron et de pocket

Réalisé par:

**FARAH Manal
ATMANI Houda
HANDI Kaoutar
MAKHCHOUN Khadija
OUZOUGAGH Chaimaa**



Table des matières

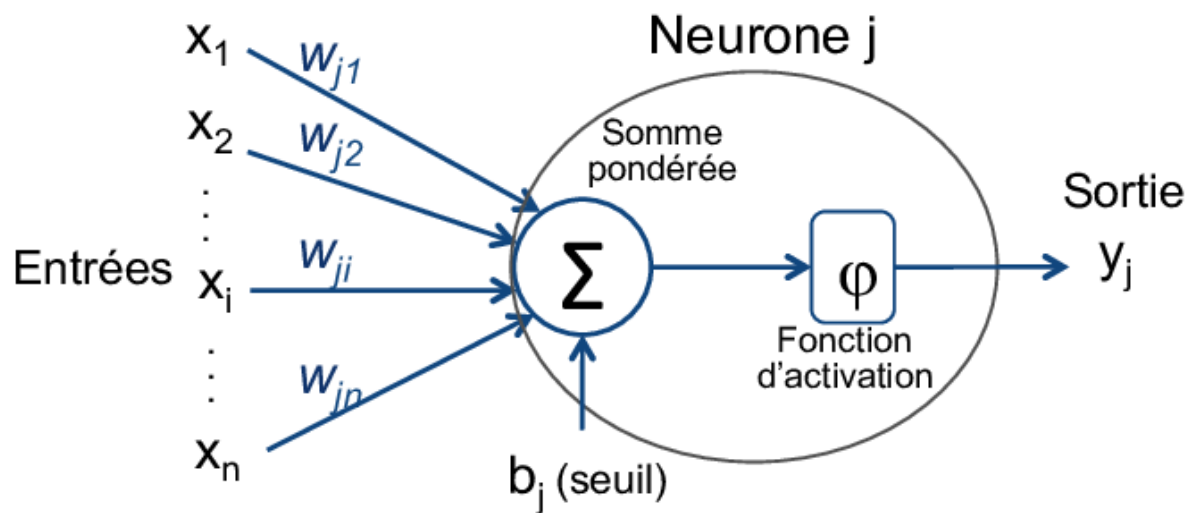
1. Introduction	3
2. Modèles de perceptron et de pocket.....	5
2.1 L'algorithme	5
2.2 Principe	6
2.3 Avantages/Désavantages.....	6
3. Programme de perceptron	6
3.1 Implémentation en python	6
3.2 Implémentation de l'algorithme.....	7
3.3 Visualisation des données.....	10
4. Programme de pocket	12
4.1 Implémentation en Python :	12
4.2 Visualisation des données.....	13
5. Variation de score en fonction de nombre des itérations: ...	14
6. Variation de score en fonction de la taille des données :	15

1. Introduction

Les réseaux de neurones <Neural Networks> représentent l'une des méthodes d'apprentissage supervisés, permettant de résoudre des problématiques d'apprentissage de la machine <Machine Learning>.

Cette méthode s'appuie sur une imitation du fonctionnement des neurones dans le cerveau humain et elle s'applique dans différents domaines tels que la reconnaissance d'image et l'identification des objets.

Un neurone j à la structure suivante :



- ✚ Les entrées (x_1, x_2, \dots, x_n) représentent des variables d'études .
- ✚ Les poids (w_{j1}, \dots, w_{jn}) reliés aux n entrées, qui sont l'objet de l'étude.
- ✚ $\sum w_{ji}x_i + b_j$: combinaisons des entrées pondérées par les poids et ajout d'un biais optionnel b_j
- ✚ ϕ : fonction d'activation appliquée sur la fonction de combinaison, permettant de déduire la sortie y_j

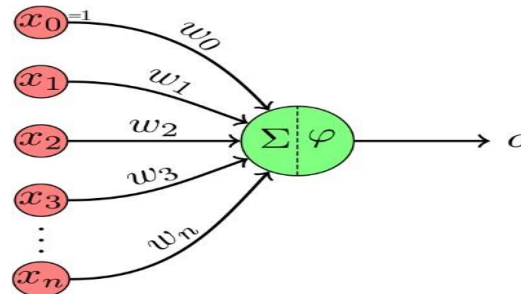
Il existe différents types de réseaux de neurones, tel que chacun se caractérise par son architecture, objectifs et domaine d'application.

On va s'intéresser dans ce devoir, au modèle de perceptron et de Pocket, en mettant l'accent sur :

- Architecture de réseau de neurones
- Le principe de l'algorithme
- L'implémentation en python
- Visualisation des résultats des algorithmes.

2. Modèles de perceptron et de pocket

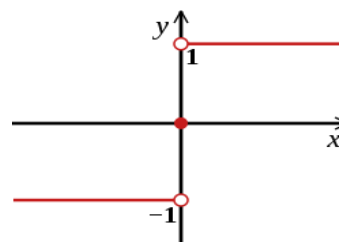
Le perceptron est un modèle linéaire de l'apprentissage supervisé, Il est le réseau de neurones le plus simple vu qu'il se compose d'un seul neurone.



Cet algorithme sert à calculer une somme pondérée par des poids $Z = \sum(x_i w_i)$, puis la passer à travers une fonction d'activation ou de seuil ϕ et renvoyer la valeur de la sortie.

La fonction d'activation dans le modèle de perceptron est appelée la fonction signe, elle se définit comme suit :

$$\text{Sign}(Z) = \Phi(z) = \begin{cases} -1 & z \leq 0 \\ +1 & z > 0 \end{cases}$$



Le fait d'avoir w_0 comme seuil revient à ajouter w_0 à la somme comme biais et avoir à la place un seuil de 0. Autrement dit, nous considérons un signal d'entrée supplémentaire x_0 qui est toujours mis à 1.

2.1 L'algorithme

Soit un fichier d'apprentissage D de N éléments, linéairement séparables :

$$D = \{(x_i, y_i)\}, \text{ où } x_i \in \mathbb{R}^d \text{ et } y_i \in \{-1, 1\},$$

L'algorithme de perceptron s'écrit :

1. Initialiser le vecteur $w \leftarrow 0$
2. Pour itération = 1 à N faire
3. Pour chaque exemplaire $(x_i, y_i) \in D$ faire
4. Calculer la prédiction $\hat{y}_i = \text{signe}(\langle w^t, x_i \rangle)$
5. Si $\hat{y}_i * y_i \leq 0$ alors
6. Ajuster $w : w \leftarrow w + x_i * y_i$
7. Fin si
8. Fin pour
9. Fin pour

2.2 Principe

Le principe de cet algorithme est de vérifier pour chaque exemplaire s'il est mal classé, c'est-à-dire que la valeur prédite \hat{y}_i et la valeur réelle y_i ont des signes différents. Si c'est bien le cas, le vecteur des poids va être mis à jour, jusqu'à avoir toutes les données bien classées, et renvoie à la fin le vecteur de pondération w .

2.3 Avantages/Désavantages

Il est vrai que l'algorithme de perceptron est le plus simple des réseaux de neurones. Cependant, il atteint rapidement ses limites techniques pour la raison qu'il peut uniquement séparer les données si elles sont linéairement séparables.

3. Programme de perceptron

3.1 Implémentation en python

Pour implémenter l'algorithme de perceptron en python on utilisera la bibliothèque numpy pour faire les opérations matrice-vecteur. Nous l'implémenterons une classe appelée Perceptron contenant 3 méthodes : `.fit()`, `.predict()` et `.score()` :

- ✚ La méthode `.fit()` sera utilisée dans la phase d'apprentissage. Elle attend comme paramètres le fichier de données réparti en 2 paramètres X et y : X étant la matrice

contenant les observations et y est un tableau d'une seule dimension contenant le label de chaque ligne de la matrice X . Cette méthode retourne le vecteur poids w qui va servir pour tracer la droite de séparation des données.

✚ La méthode `.predict()` sera utilisée pour faire la prédiction sur de nouvelles données. Elle vérifie d'abord si l'attribut poids existe, sinon cela signifie qu'on a pas encore fait la phase d'apprentissage, et nous affichons un message d'avertissement et retournons. La méthode attend comme paramètre le fichier de test sur lequel on fera la prédiction et retourne le y prédit.

✚ La méthode `.score()` calcule et renvoie la précision des prédictions.

3.2 Implémentation de l'algorithme

Tout d'abord, nous importons les bibliothèques à utiliser : **Numpy** pour manipuler les matrices et les tableaux, **Scikit-learn** comme bibliothèque d'outils dédiés au machine learning et à la data-science et **Matplotlib** pour visualiser les données graphiquement.

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
```

Ensuite, nous avons défini la classe Perceptron ayant comme première méthode `.fit()`, elle met à jour les poids après vérification de la condition de mal classification de $(y_n, x_n) : (y_n(w^t x_n) \leq 0)$

```
def fit(self, X, y, n_iter=1000):

    n_samples = X.shape[0] #nbre des lignes de la matrice
    n_features = X.shape[1] #nbre des colonnes de la matrice

    #Initialisation du vecteur des poids par des zéros
    self.weights = np.zeros((n_features+1,))

    # Ajouter une colonne des 1 et la concaténer avec X
    X = np.concatenate([X, np.ones((n_samples,1))], axis=1)

    for i in range(n_iter):
        for j in range(n_samples):

            #La condition de la mal classification
            if y[j]*np.dot(self.weights, X[j, :]) <= 0:

                #Ajouter la nouvelle valeur du poids au vecteur self.weights
                self.weights += y[j]*X[j, :]
```

La deuxième méthode `.predict()` permet de prédire les résultats de Y et les classer en des valeurs 1 ou -1, à partir des données X et à partir des poids w déjà calculés par la méthode `.fit()`

```
def predict(self, X):
    #Vérification si la phase d'apprentissage est déjà faite
    if not hasattr(self, 'weights'):
        print('The model is not trained yet!')
        return

    #la phase de prédiction
    n_samples = X.shape[0]
    X = np.concatenate([X, np.ones((n_samples,1))], axis=1)
    y = np.matmul(X, self.weights)
    y = np.vectorize(lambda val: 1 if val > 0 else -1)(y)
    return y
```


La méthode `.score()` permet de trouver la précision des résultats , en comparant la valeur des vrais y et des y_prédits, puis la moyenne des résultats de cette comparaison est retournée.

```
def score(self, X, y):  
    pred_y = self.predict(X)  
    return np.mean(y == pred_y)
```

➤ Les fonctions de la bibliothèque numpy utilisées dans l'implémentation de l'algorithme :

➔ `np.matmul()` , `np.dot()` : ces deux fonctions servent à calculer le produit matriciel. Elles ont été utilisées pour calculer le produit matriciel du vecteur des poids avec la matrice X

```
#La condition de la mal classification  
if y[j]*np.dot(self.weights, X[j, :]) <= 0:
```

```
y = np.matmul(X, self.weights)
```

➔ `np.vectorize(lambda val: 1 if val > 0 else -1)(y)`: cette fonction a été utilisée pour retourner le vecteur des valeurs de Y qui prennent soit la valeur 1 ou -1.

➔ `np.mean(y == pred_y)`: vérifie si la valeur du y prédit est égal à la vraie valeur de y en donnant le résultat 1 ou 0 , puis calcule la moyenne de ces valeurs trouvées et retourne une valeur compris entre 0 et 1.

3.3 Visualisation des données

Dans cette phase on commence par la génération d'un fichier de données d'une manière aléatoire on se basant sur la fonction *make_classification* qui est déjà défini dans la bibliothèque *scikit-learn*.

```
##### Création de données #####
X, y = make_classification(
    n_features=2, #represente le nombre de paramètres (colonnes de X)
    n_classes=2, #les classifications de y
    n_samples=200, #nombre des données (les lignes de X )
    n_redundant=0, # les combinaisons linéaire entre les données
    n_clusters_per_class=1, #le regroupement des données dans chaque classe
    random_state=55, # aide à generer les même données aleatoire
    class_sep=2 # pour separer les données lineairement
)
#####
```

On génère maintenant depuis ces données les parties utilisées dans l'apprentissage et aussi dans le test d'un pourcentage de 20% (*test_size=0.2*).

```
#permuter les 0 par des -1 dans le vecteur y
y=np.vectorize(lambda val: 1 if val > 0 else -1)(y)

# diviser les données en 2 parties ( train et test )
X_train, X_test, y_train, y_test = train_test_split( X, y, test_size=0.2, random_state=6)

p=Perceptron() #creation d'un objet p de type perceptron
p.fit(X_train, y_train,100) #executer la phase d'apprentissage des poids
```

Pour tracer la droite qui sépare nos données on la représente par l'équation suivante :

$$w_2 + w_1 \cdot x_2 + w_0 \cdot x_1 = 0$$

Pour cela on prend la valeur de x_2 depuis l'équation :

$$x_2 = (-w_2 - w_0 \cdot x_1) / w_1$$

```

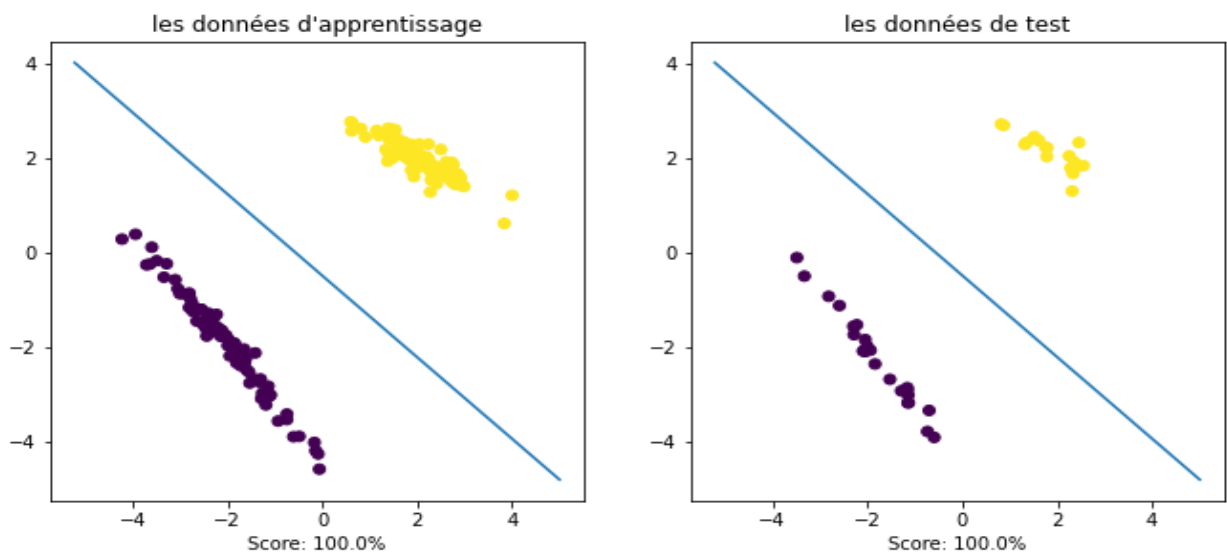
x1=np.min(X[:, 0]) - 1 #definir un min de la droite
x2=np.max(X[:, 0]) + 1 #definir un max de la droite

#definir l'equation de notre droite séparatrice
xx=np.linspace(x1,x2,1000)
L = [(-p.weights[2]-p.weights[0]*elem)/p.weights[1] for elem in xx]

### tracer les données et la droite
fig, axs = plt.subplots(1,2, figsize=(11, 5))
#tracer les données d'apprentissage
axs[0].scatter(X_train[:,0],X_train[:,1],c=y_train)
#tracer les données de test
axs[1].scatter(X_test[:,0],X_test[:,1],c=y_test)
#tracer la droite pour l'apprentissage et aussi pour le test
axs[0].plot(xx,L)
axs[1].plot(xx,L)
#### ajout des titres pour chaque graphe
axs[0].set_title("les données d'apprentissage" )
axs[1].set_title("les données de test")
##ajouter le score de chaque graphe
axs[0].set_xlabel("Score: "+str(p.score(X_train, y_train)*100)+"%")
axs[1].set_xlabel("Score: "+str(p.score(X_test, y_test)*100)+"%")

```

➤ Résultat de l'exécution



Rq: Ce programme de perceptron donne toujours un score de 100% lorsque les données sont linéairement séparables.

4. Programme de pocket

4.1 Implémentation en Python :

L'algorithme de pocket est utilisé dans le cas du non linéarité des données. La principale distinction entre cet algorithme et le précédent est de garder juste les valeurs des poids générant l'erreur minimale. Pour son implémentation, les mêmes fonctions sont utilisées avec l'ajout de la fonction `.test_score()` et la modification dans les valeurs des poids à garder dans la fonction `.fit()`.

➔ `.test_score()` : cette fonction est ajoutée pour calculer la précision avec la nouvelle valeur du poids et la comparer avec l'ancienne valeur.

```
if self.test_score(X,y,w) >= self.test_score(X,y,self.weights):
```

- C'est une fonction similaire à `.score()` avec les différences suivantes :

- Elle prend comme paramètre de plus `w`, pour retourner la précision en fonction de la valeur du poids donnée ce qui permettra de comparer les valeurs des précisions en fonction du poids.

- Elle ne fait pas la concaténation à chaque fois comme c'est le cas avec la fonction `.score()` qui appelle la fonction `.predict()` ; elle qui fait la concaténation de la colonne des 1 avec `X`.

```
def test_score(self,X,y,w):  
    y_ = np.matmul(X,w)  
    y_ = np.vectorize(lambda val: 1 if val > 0 else -1)(y_)  
    return np.mean(y==y_)
```

➔ `.fit()` : elle met à jour les poids gardés dans la « poche » après vérification de la condition de mal classification de (y_n, x_n) : $(y_n(w^t x_n) \leq 0)$ et après la comparaison de la précision de la nouvelle valeur avec la précédente , pour ne garder que les poids avec le minimum d'erreur généré.

```
def fit(self, X, y, n_iter=20):
    n_samples = X.shape[0] #nbre des lignes de la matrice

    n_features = X.shape[1] #nbre des colonnes de la matrice

    #Initialisation par 0 du vecteur où seront stockés les poids à garder
    self.weights = np.zeros((n_features+1,))

    #Initialisation du vecteur des poids par des zéros
    w = np.zeros((n_features+1,))

    # Ajouter une colonne des 1 et la concaténer avec X
    X = np.concatenate([X, np.ones((n_samples,1))], axis=1)
    for i in range(n_iter):
        for j in range(n_samples):

            #La condition de la mal classification
            if y[j]*np.dot(w, X[j, :]) <= 0:

                #Ajouter la nouvelle valeur du poids au vecteur w
                w += y[j]*X[j,:]

            #Comparaison des précisions
            if self.test_score(X,y,w) >= self.test_score(X,y,self.weights):

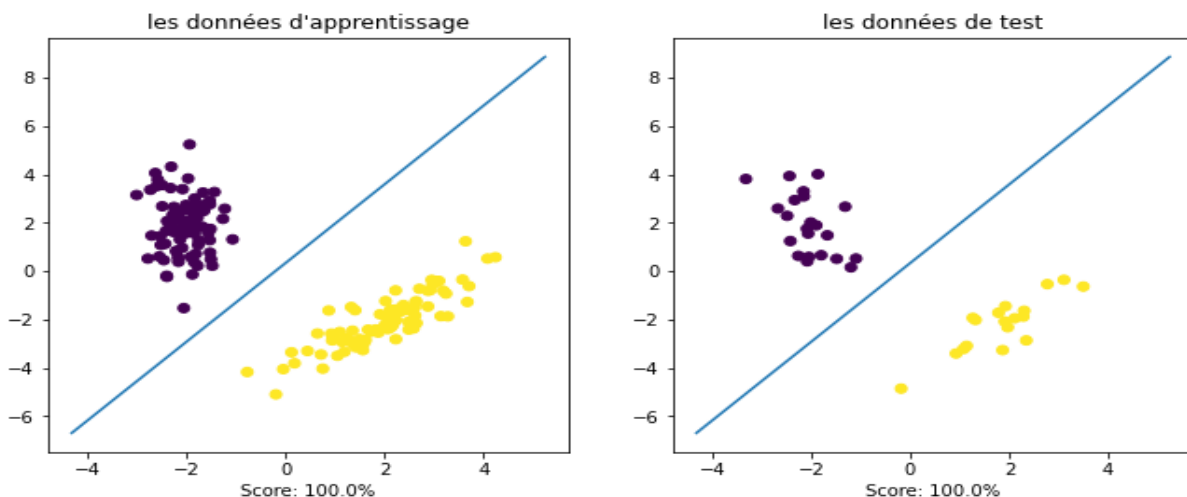
                #la valeur des poids se met à jour et prend la nouvelle valeur de w
                #elle est définie ainsi pour modifier les valeurs du vecteur self.weights
                #et ne pas impacter la valeur du vecteur w par la suite
                self.weights = [w[i] for i in range(len(w))]
```

4.2 Visualisation des données

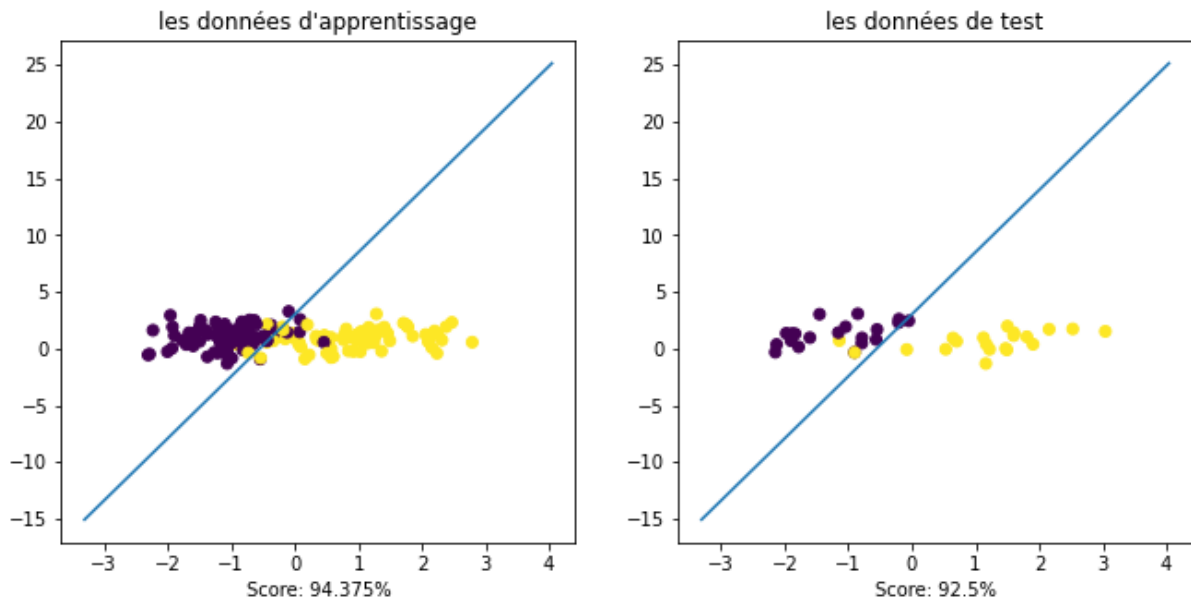
On travaille avec le même programme python déjà utilisé dans la visualisation des données de perceptron.

Si les données sont linéairement séparables Pocket nous donne un score de 100% et sinon, dans ce cas Pocket nous donne une droite avec une erreur minimale et il donne toujours des résultats beaucoup plus mieux que celles de perceptron.

- **Données linéairement séparables :**



- **Données linéairement non séparables :**



5. Variation de score en fonction de nombre des itérations:

Dans cette partie, on visualise la variation du score en fonction de nombre des itérations pour les deux programmes de perceptron et Pocket.

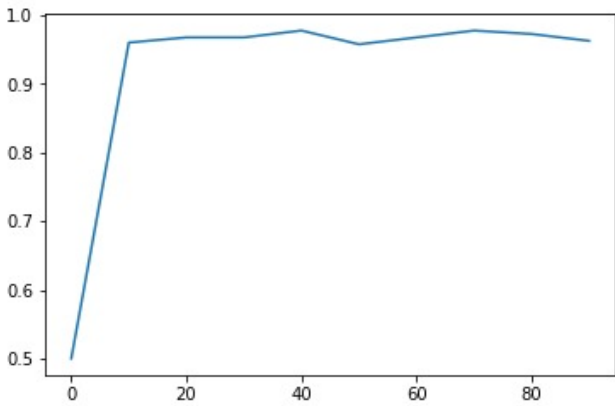
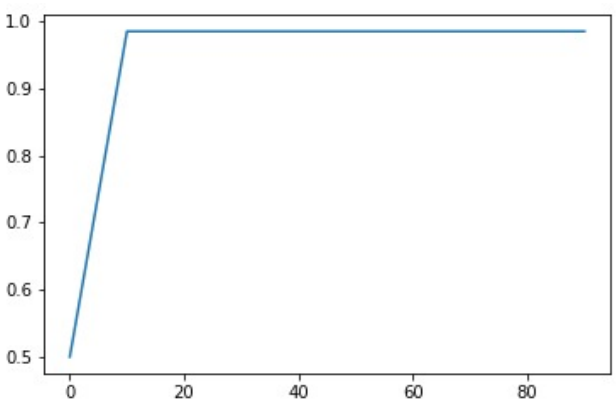
```
#####variation score#####
def create_graph(max_samples) :
    #liste des itérations
    L = [i*10 for i in range(max_samples)]
    p = Pocket()
    L_scores_train = []
    for elem in L:
        p.fit(X_train, y_train,elem)
        #calcul pour chaque itération de la liste L
        L_scores_train.append(p.score(X_train, y_train))
    #Visualisation des résultats
    plt.plot(L, L_scores_train)

create_graph(10)
```

Cette fonction permet de calculer le score de chaque itération entre 0 et le *max_samples* avec un pas de 10.

➤ **Résultat d'exécution :**

La visualisation du graphe de variation dans les deux programmes pour les mêmes données utilisées dans la phase d'apprentissage

<p>Perceptron</p> 	<p>Pocket</p> 
<p>On constate que la variation du score est instable car dans le programme de perceptron on modifie les poids sans prendre en considération le score des itérations précédentes.</p>	<p>Contrairement au programme de perceptron, Pocket modifie les poids en condition d'améliorations du score c'est pour cela le score s'améliore ou se stabilise lorsqu'on avance dans les itérations.</p>

6. Variation de score en fonction de la taille des données :

Dans cette partie, on visualise la variation du score en fonction de la taille des données utilisées dans la phase d'apprentissage pour les deux programmes de perceptron et Pocket.

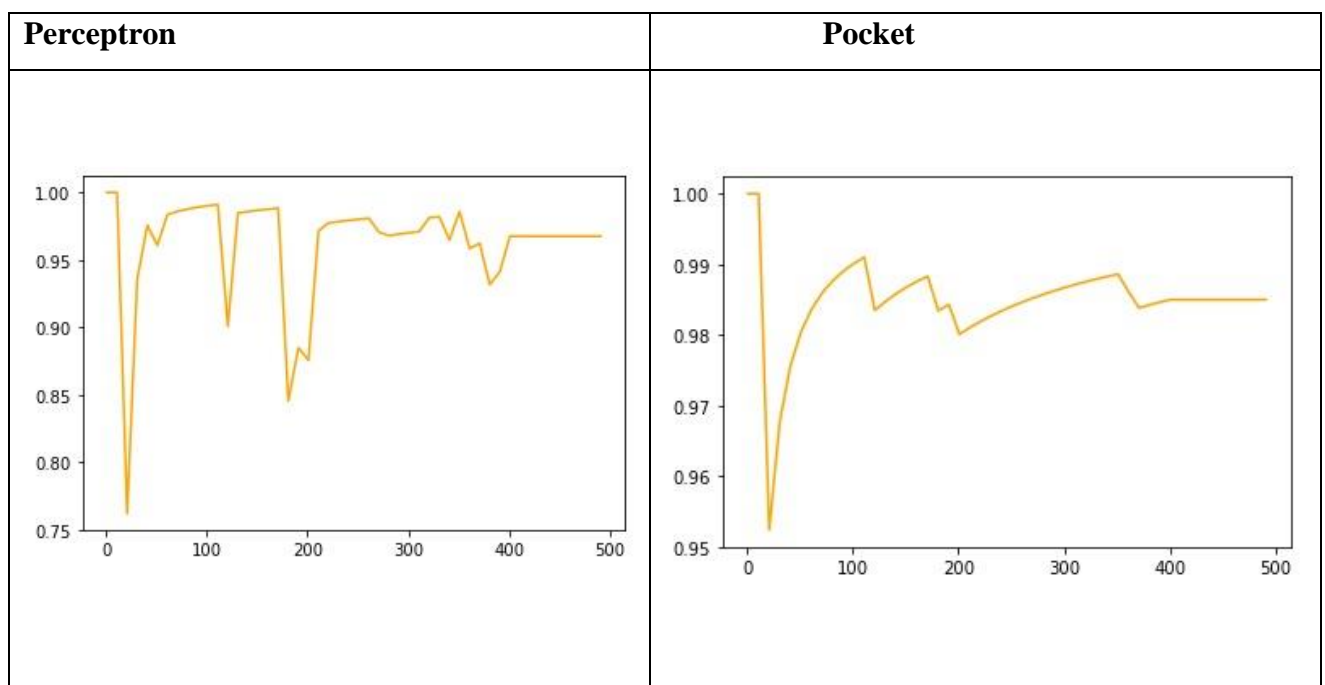
```
def create_graph2(max_samples) :
    #la liste des tailles des données utilisées dans l'apprentissage
    L = [i*10+1 for i in range(max_samples)]
    p = Pocket()
    L_scores_train = []
    for elem in L:
        p.fit(X_train[:elem], y_train[:elem],100)
        #calcul pour chaque taille de données de la liste L
        L_scores_train.append(p.score(X_train[:elem], y_train[:elem]))
    #Visualisation des résultats
    plt.plot(L, L_scores_train,color='orange')

create_graph2(20)
```

Cette fonction permet de calculer le score pour chaque taille de données commençant par 1 jusqu'à *max_samples* avec un pas de 10.

➤ **Résultat d'exécution :**

La visualisation du graphe de variation dans les deux programmes pour les mêmes données utilisées dans la phase d'apprentissage.



Remarque1 :

En utilisant les mêmes données, le score de Pocket est mieux que celui de perceptron

- ✓ Score du Pocket =0.994 (99,4%)
- ✓ Score du Perceptron =0.98 (98%)

Remarque2 :

Lorsque les données sont de taille petite, on remarque l'instabilité du score qui donne des valeurs stochastiques. Et en ajoutant les données et à un certain moment le score devient plus stable.

