Introduction à l'Apprentissage Automatique TP2

Le perceptron

ROUSSEL Hugo, M1 IARF TPA12

28/01/2019

Code :

Prédiction :

La prédiction pour un point *x* se fait très simplement si *x* est de la même dimension que *weights* :

```
return 1 if np.dot(x,self.weights) > 0 else -1
```

Sinon il suffit d'étendre x :

```
return 1 if np.dot(np.append(x,1),self.weights) > 0 else -1
```

Mais *np.append* ajoute un surcoût important, il est donc préférable de préparer les données d'entrées.

Pour faire une prédiction sur un vecteur d'entrée on utilise la prédiction sur un point, et on l'applique au vecteur avec un map :

```
return np.array(list(map(self.predict_point, X)))
```

Apprentissage :

Avec *Xp* (X prime) notre vecteur d'entrée, possiblement étendu (s'il l'est, alors *self.x_extended == True*).

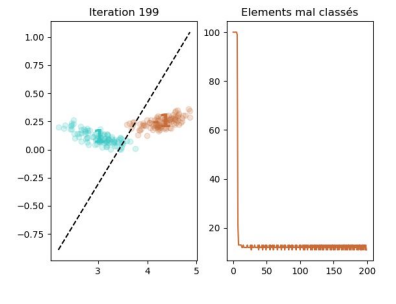
```
for iteration in range(self.max_iter):
            # variable stockant l'accumulation des coordonnées
            modif_w = np.zeros(len(self.weights))

            # prédiction des points (on peut le faire ici car batch)
            out = self.predict(Xp)

            # choisir l'équation de droite qui donne le moins d'erreur
            # permet de débloquer le perceptron dans certaines situations,
            # utile surtout si lr est petit.
            # sum(abs(y-out)/2) permet de compter le nombre d'erreur plus
              rapidement que np.sum(y != out)
            if sum(abs(y-out)/2) > len(y)/2 :
                self.weights = -self.weights
                out = self.predict(Xp)
```

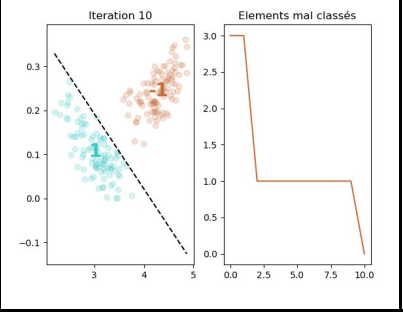
```
            if self.lr_decay:
                lr = self.lr / ((iteration%50)+1)
            else:
                lr = self.lr

            # batch learning
            for point, label, predicted in zip(Xp, y, out):
                # accumulation des coordonnées suivant la classe si les
 données sont mal classées
                if predicted != label:
                    errors[iteration] += 1
                    if self.x_extended :
                        modif_w += label*point
                    else:
                        modif_w += label*np.append(point, 1)
            self.weights += lr*modif_w
```

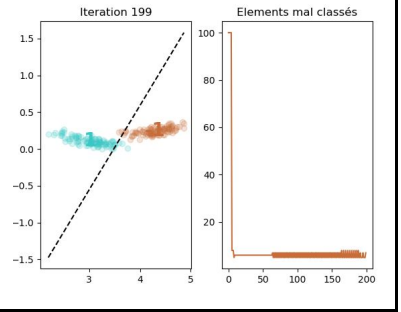
On peut détecter si l'algorithme a convergé avec :

```
stabilise = abs(sum(old_weights - self.weights)) < self.tol
```

Expérimentation :







f1 à 3 : Quelques exemples d'exécutions. lr=0.0001 constant, démarrage aléatoire. On remarque que le gros de l'apprentissage se passe durant les 20 premières itérations.

```
(base) D:\hugor\Documents\M1\iaa\TP2_etudiant>python main.py
100%|                                                                                | 100/100 [00:11<00:00,  7.64it/s]
mean score :  0.9178499999999998
```

f4 : 100 exécutions, avec lr=0.0001 constant, démarrage aléatoire. Le score est perfectible. Les mauvais scores viennent de cas tel f1, ou la droite se bloque dans une position proche d'une solution.

Pour améliorer l'apprentissage, on peut influer sur : les poids à l'initialisation, le taux d'apprentissage, et l'évolution de ce taux.

Les poids sont initialisés aléatoirement à partir d'une loi normale, c'est intéressant pour nos tests car cela permet l'apparition de valeur qui peuvent mettre en difficulté le modèle.

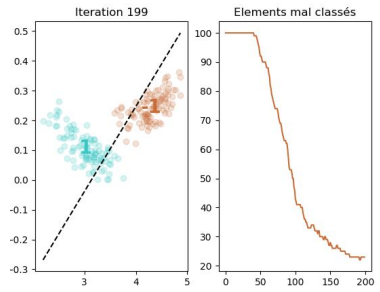
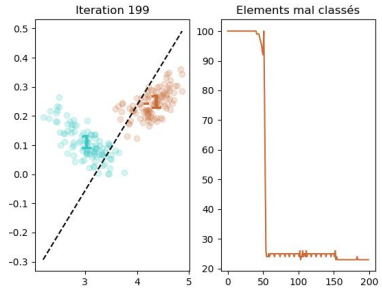
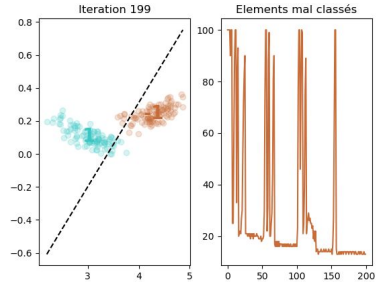
Evolution du taux d'apprentissage :

Trois façon de diminuer le taux d'apprentissage ont été expérimenté :

- en fonction du nombre d'itération.
- en fonction du nombre d'itération avec redémarrage.
- en fonction du logarithme du nombre d'itération avec redémarrage.

Un problème avec l'approche lr/itération est la lenteur d'apprentissage dans le cas ou le plan d'origine est éloigné des nuages de points (cf. f5). Pour contrer cet effet j'ai expérimenté avec 2 approches : le redémarrage (f6) et le logarithme (f7).

Valeurs initiales : lr = 0.001, weights = [-1.96813617 , 1.36967785 , 0.76232974]

		
f5 : lr = self.lr / (iteration+1)	*f6 : lr = self.lr / ((iteration%50)+1)*	*f7 : lr = self.lr/math.log10((iteration % 50)+2)*

L'idée du redémarrage vient du constat que l'apprentissage se fait principalements dans les premières itérations. Avec le logarithme on espère réduire la décroissance de lr pour rester dans des plages favorable à un apprentissage rapide.

Comparaison du score moyen sur 100 exécutions :

	lr/(iteration)	lr/(itération % 50)	lr/log10(itération % 50)
Score moyen	0.91	0.92	0.97

(lr initial=0.001, démarrage aléatoire)

Taux d'apprentissage initial :

Comparaison du score moyen sur 1000 exécutions, avec temps :

	1	0.1	0.01
lr/(iteration)	0.9999 (29s)	0.9954 (1m08s)	0.9407 (1m41s)
lr/(itération % 50)	1.0 (26s)	0.9986 (56s)	0.9849 (1m32s)
lr/log10(itération % 50)	1.0 (51s)	1.0 (54s)	0.9983 (1m16)

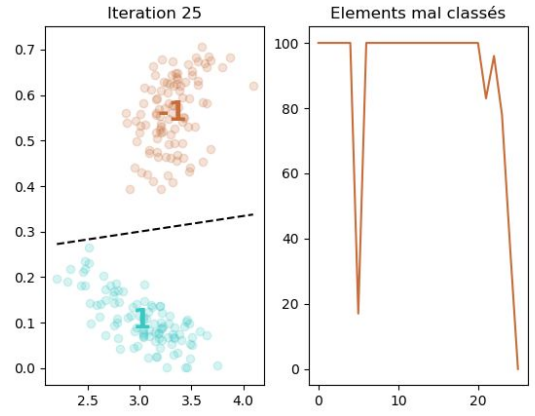
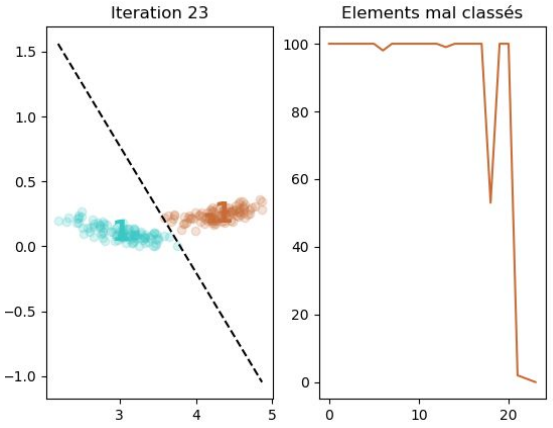
On remarque que plus le taux d'apprentissage initial est petit, plus le temps d'exécution est grand, cela est dû à l'augmentation du nombre d'itération nécessaire pour atteindre la solution.

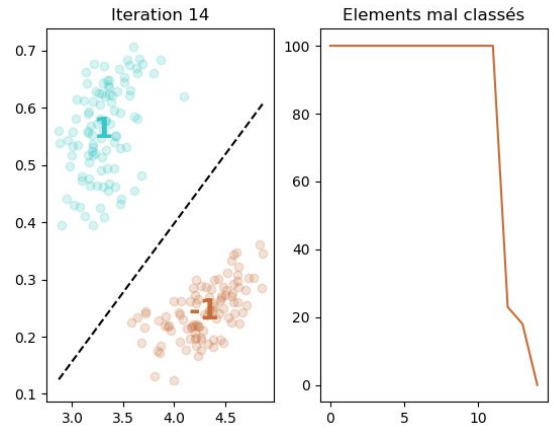
Le logarithme produit un résultat plus consistant que les 2 autres méthodes. Mais pour la suite on préfèrera utiliser la méthode redémarrage simple, avec un taux d'apprentissage initial d'un, car c'est la plus rapide.

Pour s'assurer de sa robustesse, on teste avec 10 000 exécution : le score est toujours 1.0 (temps : 4m24).

Applications :

Application de l'algorithme sur les différents groupes de données.

<table>
<tr>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td>Groupes 1 et 2, hyperplan : [2.60853419 -76.21460628 15.05180779]</td>
<td>Groupes 1 et 3, hyperplan : [-15.45995799 -15.80719731 58.66490939]</td>
</tr>
</table>

<table>
<tr>
<td> 
</td>
</tr>
<tr>
<td>Groupes 2 et 3, hyperplan : [-15.45954682 64.06061139 36.33634019]</td>
</tr>
</table>