Veysseire Daniel TP2 IAA Université Paul Sabatier

TP2

IAA

Tests

1. En prenant mu=1/no_itération, on a un epsilon qui stagne vers 0.0170 après 725 itérations, autrement dit

   on a min(norm(erreurs)) ~ 0.0170

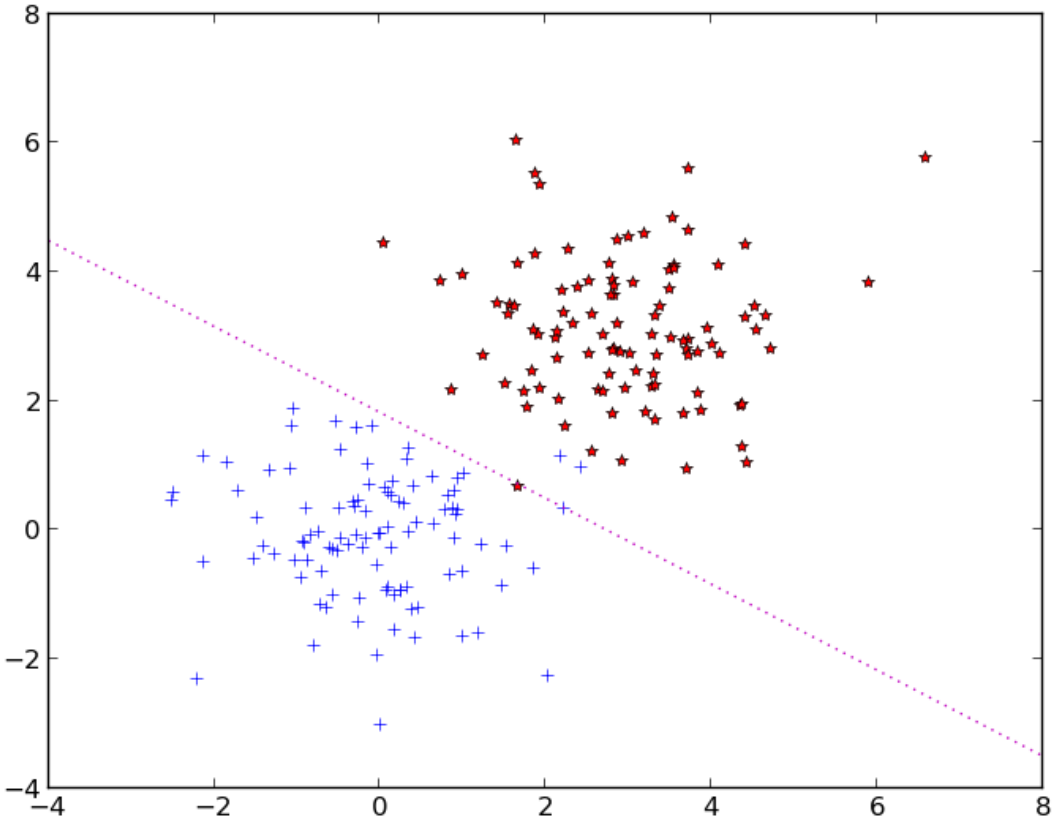
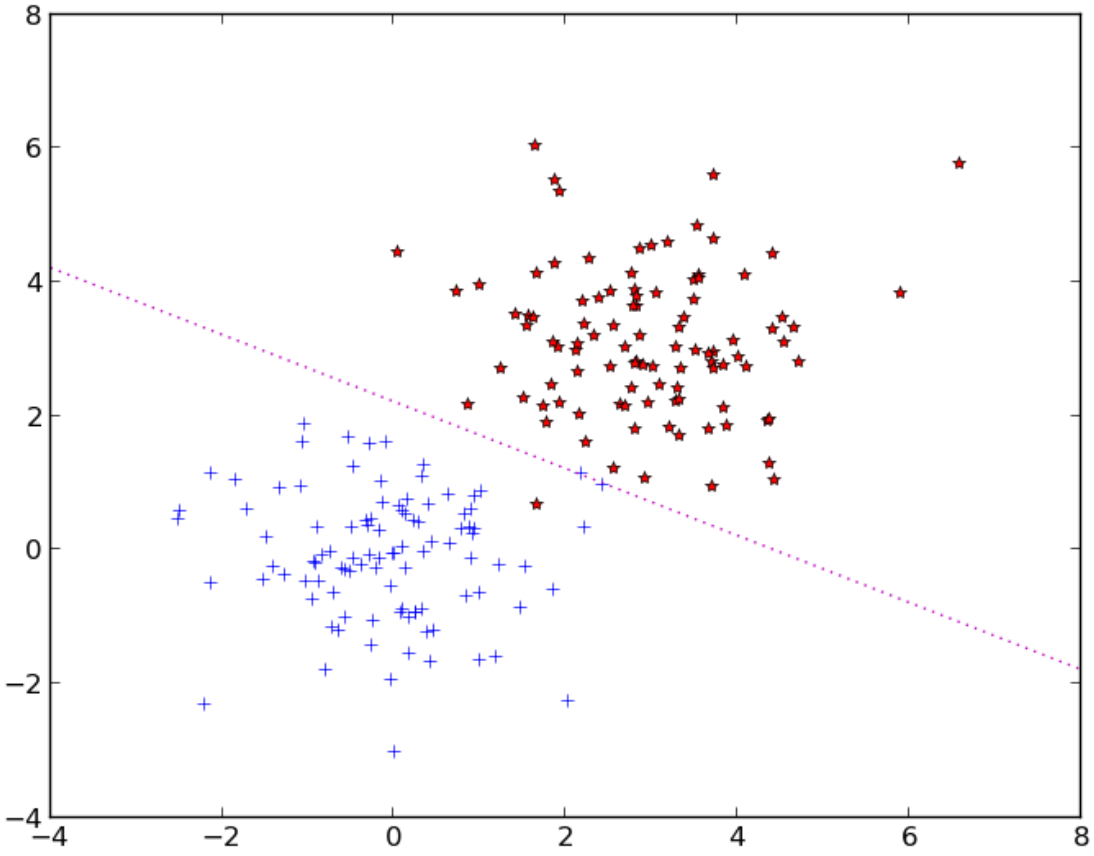


Illustration 1: Algorithme du perceptron sur les classes 1 et 2

Alors que en gardant mu=1. Tout du long, on a epsilon qui oscille autour de 0.0035 après 15 itérations.

Veysseire Daniel TP2 IAA Université Paul Sabatier



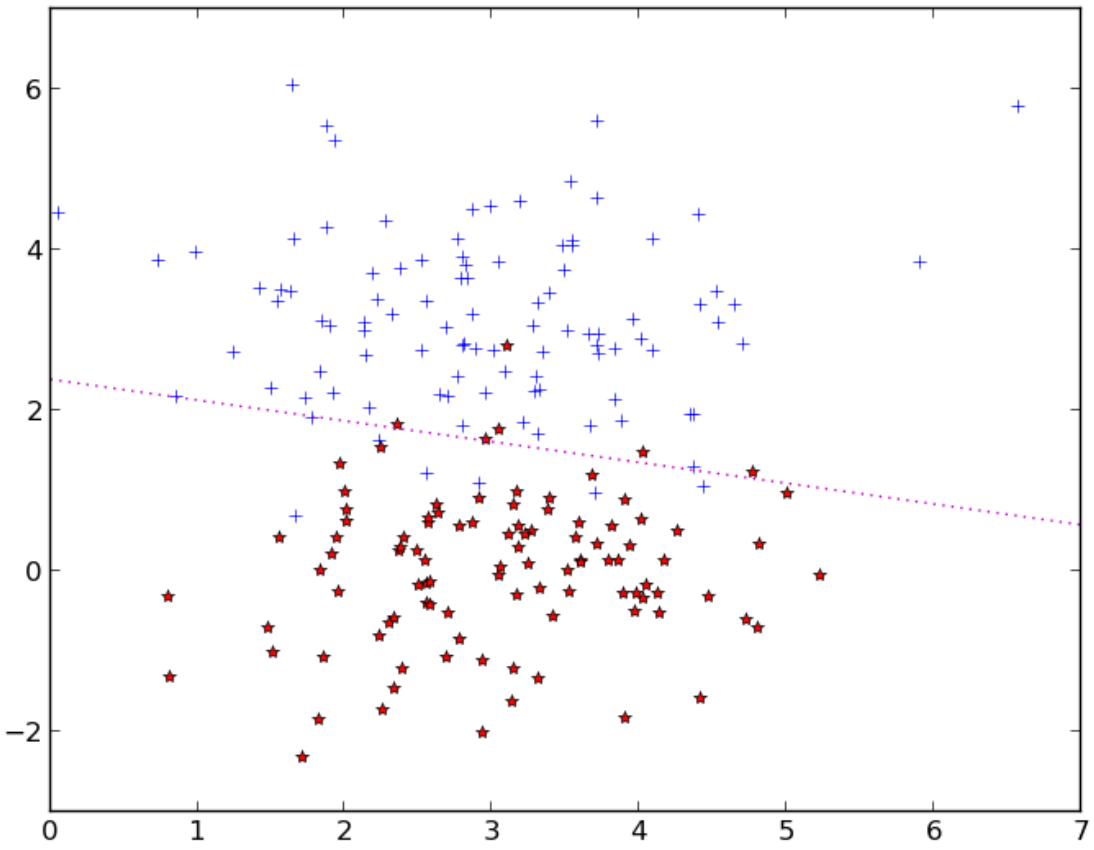
Avec mu = 2 tout du long, on arrive au même résultat après 22 itérations

La valeur de mu semble décroître trop vite si on fait mu=1./no_iteration.

J'opte pour prendre mu=mu / 1.01 pour que mu diminue plus lentement.

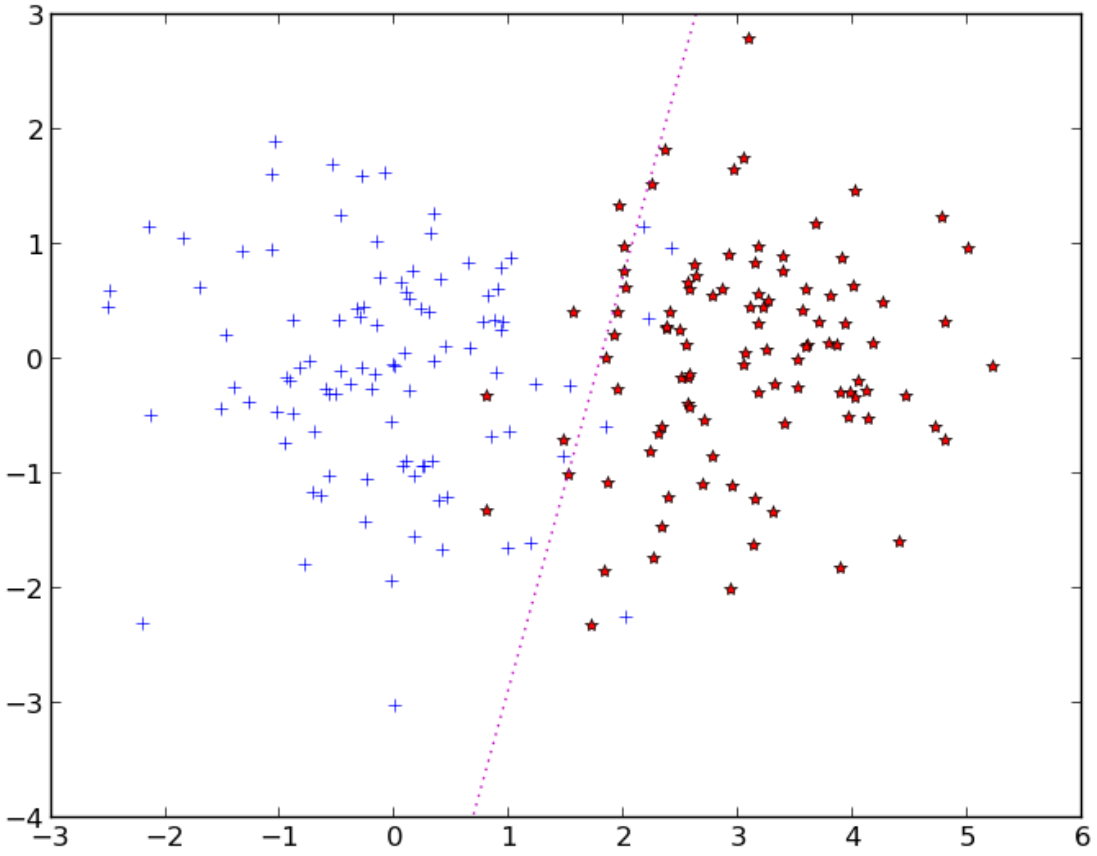
2. pour les classes 2 et 3, il faut epsillon = 0.011 pour avoir de bons resultats.

Veysseire Daniel TP2 IAA Université Paul Sabatier



Pour les classes 1 et 3, il faut epsilon = 0.011 pour avoir de bons résultats

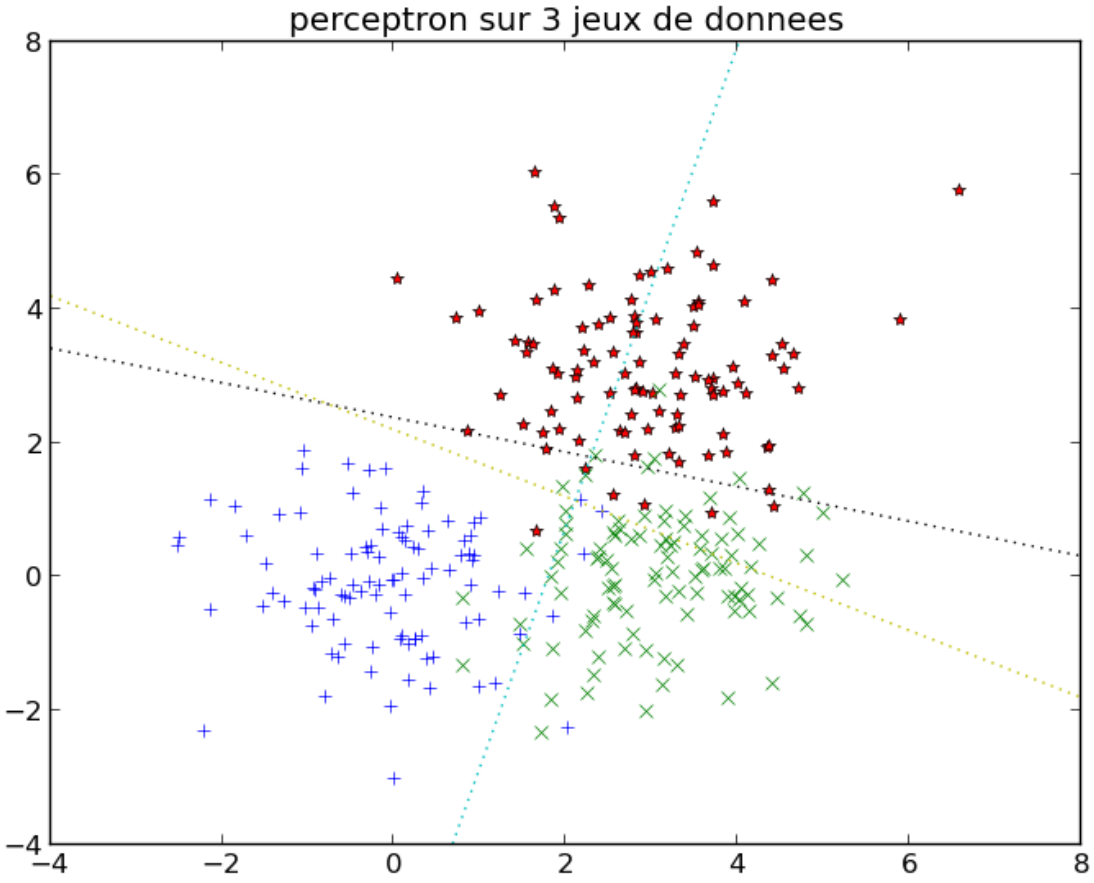
Veysseire Daniel TP2 IAA Université Paul Sabatier



pour avoir de bons résultats pour séparer nos trois classes, il nous faut epsilon = max(0.011,0.0035) = 0.011

on obtient :

Veysseire Daniel TP2 IAA Université Paul Sabatier

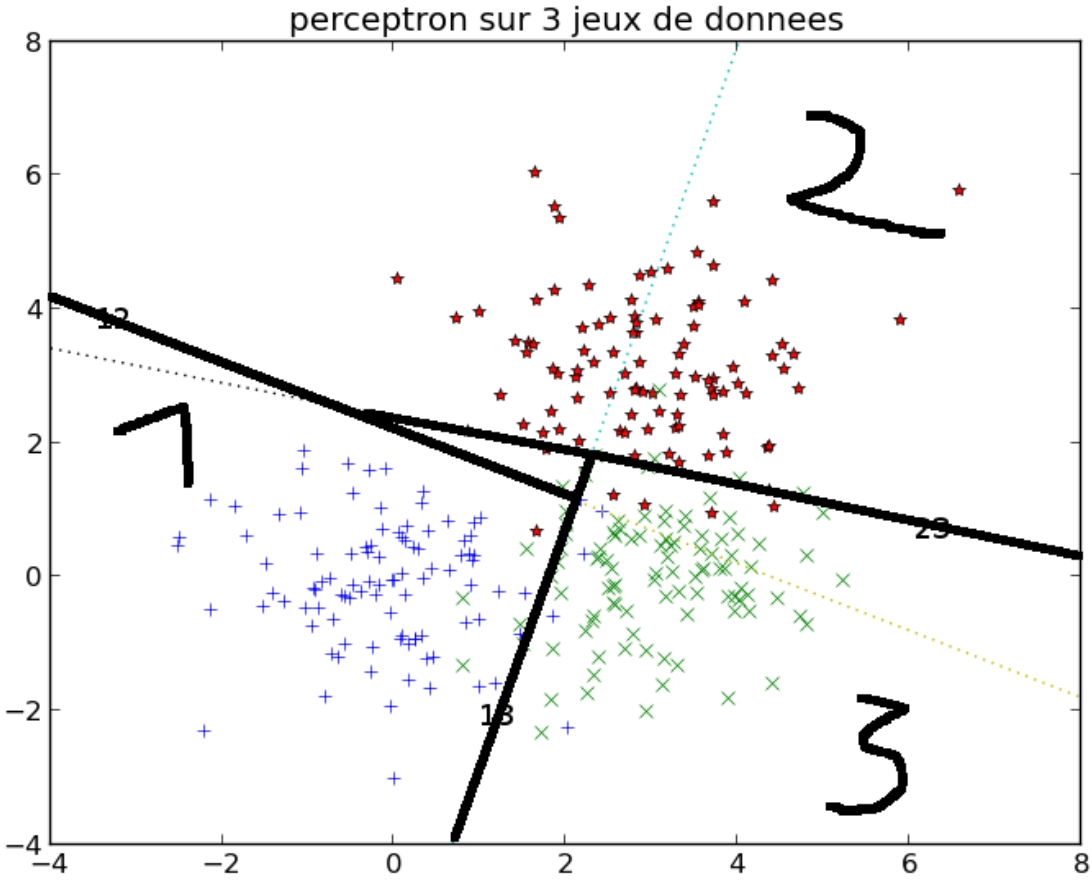


La classe 1 est bleu, la classe 2 rouge et la classe 3 verte.

La ligne jaune sépare la classe 1 et 2, la ligne verte sépare les classes 1 et 3, et la ligne noir sépare les classes 2 et 3.

On a donc trouvé les trois droites mais le problème n'est pas tout à fait résolu : il reste une zone indéterminée. En l' occurence celle se trouvant « au milieu » (cf. image ci dessous)

Veysseire Daniel TP2 IAA Université Paul Sabatier



Pour déterminer la classe à laquelle appartiennent les points situés dans cette zone, on peut prendre la demi-droite séparatrice la plus éloignée, et l'appliquer pour déterminer à quelle classe n'appartient pas le point. Il suffit ensuite d'appliquer la limite entre les deux classes restantes et regarder ou se situe le point par rapport à elle.

J'ai changé la fonction d'affichage pour que ce ne soit plus l'élément 0 de point[i] qui soit égale à 1 pour tout i, mais l'élément 2 afin de retomber sur la même formule que le cours.

J'ai changé la condition d'arrêt, tout en laissant l'ancienne en commentaire.

La nouvelle condition d'arrêt est :

(norme(ancienW – nouveauw) / norme(ancien W)) < 0.01 comme demandé en cours.

On obtient des résultats similaires.

3.C'est fait. Vérifions par des tests.

En prenant

classe4 =[[0.],[2.]]

et

```
classe5 = [[1.],[3.]]
```

on obtient

w=[-2.7027997230788285e-162, 3.5905873189199054e-162]

apres 37342 itérations.

Ces résultats me semblent étrange j'ai du mal à les interpréter en dimension 1. Cependant le programme compile et s'exécute sans problèmes.

Essayons en dimension 4 :

en prenant

```
classe4 =[[0.,0.,0.],[0.,1.,0.]]
```

```
classe5 = [[1.,1.,1.],[1.,0.,1.]]
```

on obtient au bout de 6 itérations :

w=[-0.48519752965395546, 0.50004877215493337, -0.48519752965395546, 9.7544309866748335e-05]

Je ne sais pas interpréter non plus ces résultats en dimension 4 mais ils me semblent plus plausibles que les résultats trouvés en dimension 2. Cependant le programme compile et s'exécute.

Remarque et astuce

Les listes sont assez difficilement manipulable, pour additionner ou soustraire deux vecteurs entre eux, ou multiplier par des scalaires, mieux vaut passer par des arrays.

Exemple :

```
In [32]: a=[1.,2.,3.]

In [33]: b=[4.,5.,6.]

In [34]: a+b
Out[34]: [1.0, 2.0, 3.0, 4.0, 5.0, 6.0]

In [35]: array(a)+array(b)
Out[35]: array([ 5.,  7.,  9.])

In [36]: list(array(a)+array(b))
Out[36]: [5.0, 7.0, 9.0]

In [37]: 3*array(a)
Out[37]: array([ 3.,  6.,  9.])

In [38]: 3*a
Out[38]: [1.0, 2.0, 3.0, 1.0, 2.0, 3.0, 1.0, 2.0, 3.0]

In [39]:
```

Pour les listes, le + concatène, et logiquement le * par un scalaire concatène le nombre de fois souhaité.

En revanche, pour les arrays, le + additionne membre à membre donc le * par un scalaire additionne membre à membre le nombre de fois souhaité donc réalise la vraie multiplication par un scalaire matriciel.

L'opération * entre deux matrices ne fonctionne pas pour les listes mais multiplie membre à membre pour les arrays, et ne réalise donc pas le produit matriciel.

```
In [44]: array(a)*array(b)
Out[44]: array([  4.,  10.,  18.])
```

Pour réaliser le produit matriciel, il faut utiliser la fonction dot, fonctionnant aussi bien sur les lists que sur les arrays

```
In [45]: dot(a,b)
Out[45]: 32.0

In [46]: dot(array(a),array(b))
Out[46]: 32.0
```

Une possibilité pour rajouter un élément 1. à chaque point d'une liste est de procéder ainsi :

On créer un vecteur colonne de taille len classe 1 en faisant :

ones((len(classe1),1))

Veysseire Daniel TP2 IAA Université Paul Sabatier

On obtient

```
array([[ 1.],
       [ 1.],
       [ 1.],
  ...
       [ 1.],
       [ 1.]])
```

Il suffit de concaténer ce vecteur à la classe1 pour obtenir ce qu'on désire

```
res=concatenate( (classe1, ones( (len(classe1) , 1) )  ),1)
```

Mais on obtient un array à cause de la fonction concatenate. Il faut donc le changer en list, et comme tous ces éléments sont des arrays il faut les rechanger en list eux aussi

```
final = list(res)
for i in range(len(final)):
        final[i] = list(final[i])
```

Cette expression étant lourde et difficile à comprendre vite, je ne l'ai pas utilisée dans le TP.

Veysseire Daniel TP2 IAA Université Paul Sabatier

Contact

Si il y a des erreurs, des remarques, des ajouts à faire, etc.
Veuillez m'en faire part à cette adresse :
wedg@hotmail.fr