# TESTS TME3:

# SAHLI OUSSAMA

Plot_error (trainx, trainy, mse).

Trainx = Ensemble de données 2D généré selon un mélange de deux lois gaussiennes.

Trainy = Les labels associés aux données (50% de 1 et 50% de-1).

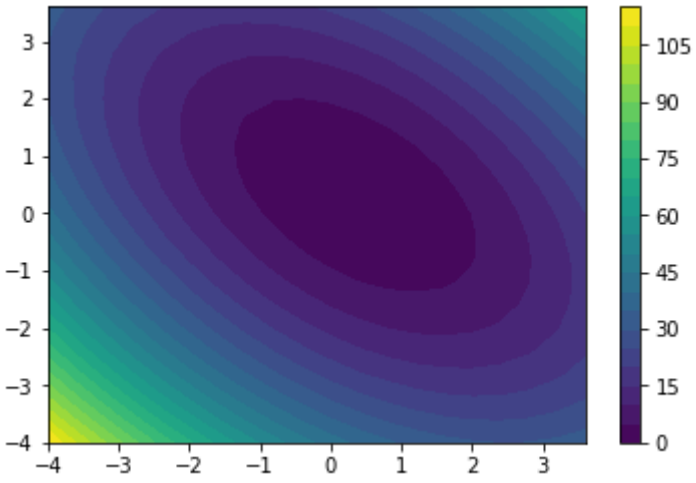
L'ensemble de données 2D peut être généré soit par un mélange de deux lois gaussiennes, ou soit par un mélange de 4 lois gaussienne (paramètre data_type de la fonction gen_arti).

On peut aussi gérer le bruit dans les données avec le paramètre epsilon. C'est-à-dire qu'on ajoute à chaque dimension de l'ensemble de données 2D les valeurs contenues dans un échantillon généré selon une loi normale dont l'écart-type est epsilon.

La fonction plot_error définit trois variables : grid, x1list, x2list. La variable grid contient plusieurs points 2D. Ces points 2D sont les différents vecteurs de poids W, à partir desquelles on calculera la valeur de la fonction de coût passée en paramètre. Ici la fonction de coût utilisée est la MSE (la moyenne de l'erreur quadratique). Ainsi pour chaque vecteur de poids W contenu dans grid on calcule la valeur de la fonction de coût à partir de ce W et de l'ensemble trainx et trainy.

On obtient une représentation graphique, où l'axe des abscisses (x1list) représente la valeur de la 1ère dimension du vecteur poids W, et l'axe des ordonnées représente la valeur prise par la 2ème dimension de W. Les couleurs obtenues sur le graphique nous permettent de visualiser l'erreur calculée par la fonction de coût en fonction du vecteur de poids W utilisé. Ainsi à partir des couleurs, on peut observer pour quelles valeurs de poids on obtient une valeur de coût minimale.

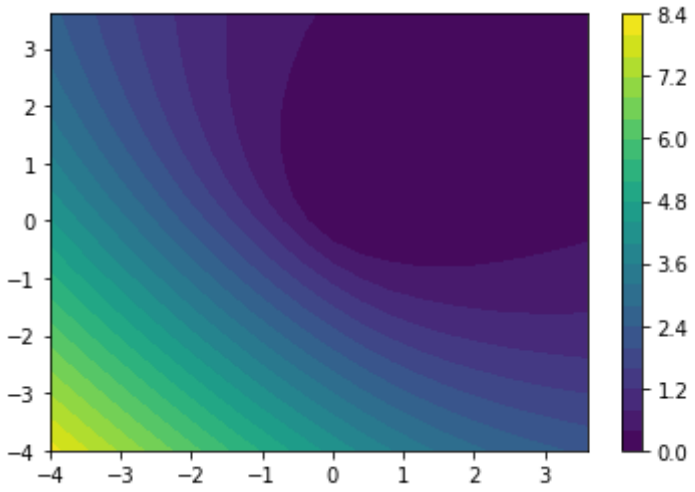
plot_error (trainx, trainy, mse):




Ici la zone mauve correspond à une valeur minimale de l'erreur calculé. On remarque que la moyenne de l'erreur quadratique est proche de 0 sur la zone en mauve du dessin (plus

particulièrement au milieu du dessin). En utilisant cette information, on peut par exemple dire que le vecteur de poids w : [0.5,0.5] permet de réduire considérablement la valeur de la moyenne de l'erreur quadratique calculé à partir des prédictions faites sur l'ensemble de données trainx (associé aux labels définis dans trainy).

plot_error (trainx, trainy, hinge):




Ici la fonction de coût utilisé permet de calculer la moyenne de l'erreur hinge. On remarque que par rapport au cas précèdent, les valeurs des erreurs calculés sont moins grandes (ici la valeur max de la moyenne de l'erreur hinge s'approche de 8.4, tandis que dans le cas précèdent la valeur max de la moyenne de l'erreur quadratique s'approchait de 105).

On observe que la zone mauve correspond à l'endroit où la moyenne de l'erreur hinge est la plus faible. Ainsi, on peut dire que, par exemple, le vecteur de poids w : [2,1.5] permet de réduire considérablement la valeur de la moyenne de l'erreur hinge calculé à partir des prédictions faites sur l'ensemble de données trainx (associé aux labels définis dans trainy).

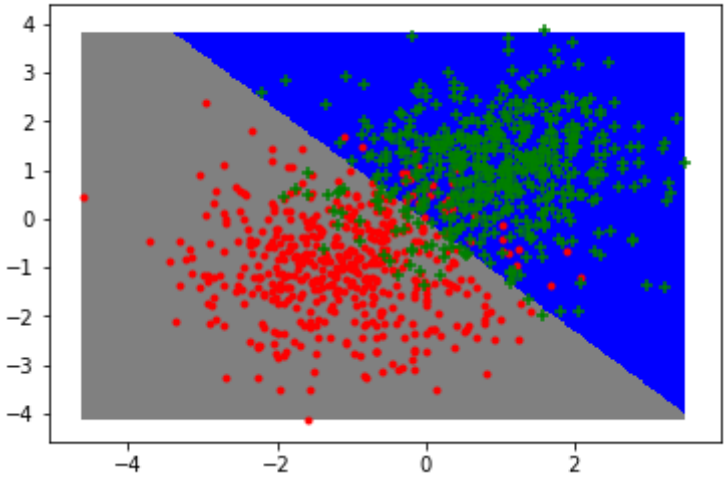
# Perceptron

Maintenant, on va entrainer notre modèle.

perceptron = Lineaire (hinge, hinge_g, max_iter=1000, eps=0.1).

Ici, la fonction de coût hinge est passé en paramètre de notre modèle, donc le modèle représentera un perceptron.

Les frontières obtenues dans l'espace de représentation des exemples :




Scores obtenus (Pour le Perceptron) :

Erreur : train 91.700000 %, test 90.300000 %

Ici le critère utilisé comme critère de convergence dans notre algorithme de descente de gradient est le paramètre max_iter de notre modèle. On va faire varier la valeur de cet hyperparamètre afin d'observer l'évolution du score de prédictions sur les ensembles train et test.

trainx,trainy = at.gen_arti(nbex=1000,data_type=0,epsilon=1)

testx,testy = at.gen_arti(nbex=1000,data_type=0,epsilon=1)

Variations de max_iter entre 10 et 1000 :

Pour chaque valeur de max_iter, on répète le processus d'entrainement 10 fois pour avoir une accuracy moyenne de l'accuracy calculé pour l'ensemble train et l'ensemble test.

```
for i in range (10,1100,100):
    meantrain=[]
    meantest=[]
    for j in range(0,10):
        perceptron = Lineaire(hinge,hinge_g,max_iter=i,eps=0.1)
        perceptron.fit(trainx,trainy,"batch")
        meantrain.append(perceptron.score(trainx,trainy))
        meantest.append(perceptron.score(testx,testy))
    print("Erreur moyenne : train %f, test %f, max_iter %d"% (np.mean(meantrain),np.mean(meantest),i))
```

```
Erreur moyenne : train 90.040000, test 90.670000, max_iter 10
Erreur moyenne : train 87.420000, test 87.380000, max_iter 110
Erreur moyenne : train 91.350000, test 91.340000, max_iter 210
Erreur moyenne : train 91.310000, test 91.420000, max_iter 310
Erreur moyenne : train 90.850000, test 91.130000, max_iter 410
Erreur moyenne : train 83.500000, test 83.660000, max_iter 510
Erreur moyenne : train 90.800000, test 91.340000, max_iter 610
Erreur moyenne : train 87.000000, test 87.130000, max_iter 710
Erreur moyenne : train 91.070000, test 91.380000, max_iter 810
Erreur moyenne : train 91.170000, test 91.400000, max_iter 910
Erreur moyenne : train 91.310000, test 91.400000, max_iter 1010
```

On remarque que pour un nombre d'itérations égale à 310, on obtient une accuracy moyenne de 91.31% pour l'ensemble de train, et une accuracy moyenne de 91.42% pour l'ensemble de test. J'ai décidé de privilégier l'accuracy moyenne calculé sur l'ensemble de test. Ainsi, on peut dire que dans le cas de ce perceptron, max_iter=310 peut être un bon critère de convergence.

Variations de max_iter entre 1000 et 10000 :

```
Erreur moyenne : train 92.070000, test 89.900000, max_iter 10
Erreur moyenne : train 93.060000, test 90.600000, max_iter 1010
Erreur moyenne : train 93.030000, test 90.500000, max_iter 2010
Erreur moyenne : train 93.000000, test 90.640000, max_iter 3010
Erreur moyenne : train 93.030000, test 90.520000, max_iter 4010
Erreur moyenne : train 83.640000, test 81.710000, max_iter 5010
Erreur moyenne : train 88.250000, test 86.120000, max_iter 6010
Erreur moyenne : train 90.670000, test 88.230000, max_iter 7010
Erreur moyenne : train 92.750000, test 90.300000, max_iter 8010
Erreur moyenne : train 92.610000, test 90.230000, max_iter 9010
Erreur moyenne : train 92.800000, test 90.060000, max_iter 10010
```

On peut constater que plus le nombre d'itérations à tendance devenir très grand, plus l'accuracy moyenne calculée sur l'ensemble d'apprentissage à tendance aussi à augmenter (elle se situe dans les 93%), tandis que l'accuracy moyenne calculée sur l'ensemble de test diminue (on était à 91% dans les cas précédents, avec max_iter <=1000, et là nous sommes dans les 90%). Mais, on remarque aussi que cette tendance ne s'applique à certaines valeurs de max_iter, car on peut constater une légère baisse de l'accuracy pour des valeurs de max_iter=5010, max_iter=6010 et max_iter=7010. Mais la tendance reprend au-delà de ces nombres d'itérations.

Ainsi, je pense qu'il est préférable de garder max_iter=310 pour notre perceptron (je privilégie l'accuracy moyenne définit sur l'ensemble de test).

Précédemment on a appliqué une descente de gradient batch, maintenant on va utiliser une descente de gradient stochastique puis mini-batch.

Ici je veux observer si le choix d'une descente de gradient batch, stochastique ou mini-batch à une influence sur l'accuracy moyenne calculé.

Descente de gradient mini-batch :

```
for i in range (10,1100,100):
     meantrain=[]
     meantest=[]
     for j in range(0,10):
         perceptron = Lineaire(hinge,hinge_g,max_iter=i,eps=0.1)
         perceptron.fit(trainx,trainy,"mini-batch")
         meantrain.append(perceptron.score(trainx,trainy))
         meantest.append(perceptron.score(testx,testy))
     print("Erreur moyenne : train %f, test %f, max_iter %d"% (np.mean(meantrain),np.mean(meantest),i))
```

```
Erreur moyenne : train 91.210000, test 90.400000, max_iter 10
Erreur moyenne : train 89.500000, test 88.720000, max_iter 110
Erreur moyenne : train 89.730000, test 89.020000, max_iter 210
Erreur moyenne : train 91.030000, test 90.170000, max_iter 310
Erreur moyenne : train 90.150000, test 89.270000, max_iter 410
Erreur moyenne : train 89.840000, test 88.980000, max_iter 510
Erreur moyenne : train 90.790000, test 89.970000, max_iter 610
Erreur moyenne : train 90.660000, test 90.070000, max_iter 710
Erreur moyenne : train 89.910000, test 89.150000, max_iter 810
Erreur moyenne : train 89.580000, test 88.810000, max_iter 910
Erreur moyenne : train 90.940000, test 90.100000, max_iter 1010
```

Descente de gradient stochastique :

```
for i in range (10,1100,100):
     meantrain=[]
     meantest=[]
     for j in range(0,10):
         perceptron = Lineaire(hinge,hinge_g,max_iter=i,eps=0.1)
         perceptron.fit(trainx,trainy,"stochastique")
         meantrain.append(perceptron.score(trainx,trainy))
         meantest.append(perceptron.score(testx,testy))
     print("Erreur moyenne : train %f, test %f, max_iter %d"% (np.mean(meantrain),np.mean(meantest),i))
```

```
Erreur moyenne : train 90.620000, test 90.400000, max_iter 10
Erreur moyenne : train 90.820000, test 90.690000, max_iter 110
Erreur moyenne : train 90.760000, test 90.630000, max_iter 210
Erreur moyenne : train 90.590000, test 90.700000, max_iter 310
Erreur moyenne : train 90.310000, test 90.320000, max_iter 410
Erreur moyenne : train 90.400000, test 90.430000, max_iter 510
Erreur moyenne : train 90.720000, test 90.720000, max_iter 610
Erreur moyenne : train 90.540000, test 90.440000, max_iter 710
Erreur moyenne : train 90.560000, test 90.440000, max_iter 810
Erreur moyenne : train 90.620000, test 90.460000, max_iter 910
Erreur moyenne : train 90.780000, test 90.600000, max_iter 1010
```

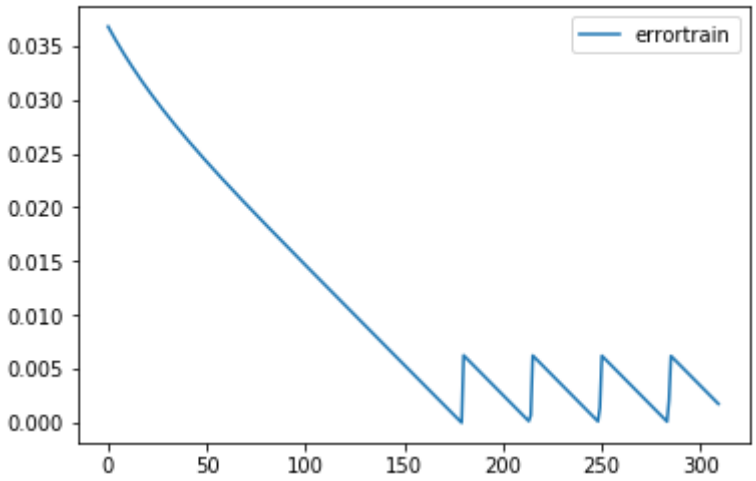
A priori, aux vues des résultats obtenues ci-dessus, l'accuracy moyenne calculé selon les différentes variantes de descentes de gradient, ne change pas considérablement. Même si, nous pouvons dire que sur la variante batch, les résultats obtenues, à partir d'un ensemble d'apprentissage contenant 1000 exemples, sont un tout petit peu meilleur.

A présent, je vais augmenter la taille de mon ensemble d'apprentissage et de mon ensemble de test. Au lieu de considérer seulement 1000 exemples, je vais en considérer 1 millions. Ainsi j'observerai l'allure des courbes de l'erreur en apprentissage sur chacune des variantes de la descente de gradient.

```
trainx,trainy =  at.gen_arti(nbex=1000000,data_type=0,epsilon=1)
testx,testy =  at.gen_arti(nbex=1000000,data_type=0,epsilon=1)
```

```
perceptron = Lineaire(hinge,hinge_g,max_iter=310,eps=0.1,biais=False)
perceptron.fit(trainx,trainy,"batch")
x = [i for i in range(0,perceptron.max_iter)]
y1 = perceptron.errorstrain
plt.plot(x, y1, label="errortrain")
plt.legend()
plt.show()
```

<u>Courbe de l'erreur d'apprentissage en utilisant une descente de gradient batch sur un ensemble contenant 1 millions d'exemples :</u>

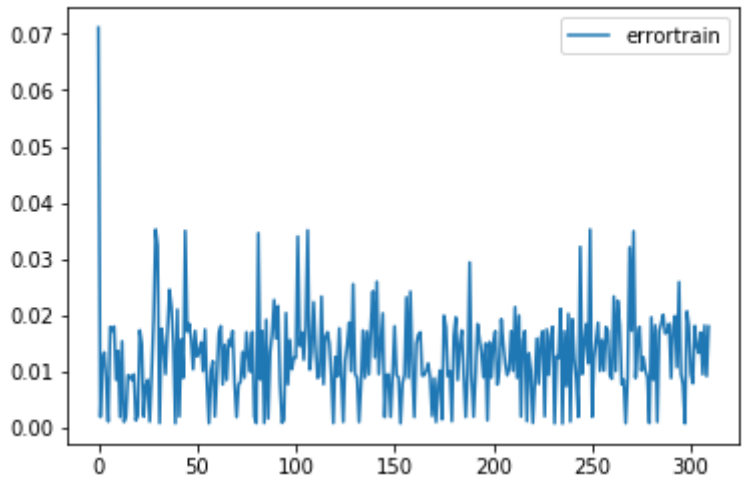



Temps de calcul de la descente de gradient batch sur un ensemble de 1 millions de données : 31 min.

Maintenant on va observer l'allure de la courbe en utilisant une descente de gradient  stochastique.

```
perceptron = Lineaire(hinge,hinge_g,max_iter=310,eps=0.1,biais=False)
perceptron.fit(trainx,trainy,"stochastique")
x = [i for i in range(0,perceptron.max_iter)]
y1 = perceptron.errorstrain
plt.plot(x, y1, label="errortrain")
plt.legend()
plt.show()
```

Courbe de l'erreur d'apprentissage en utilisant une descente de gradient stochastique sur un ensemble contenant 1 millions d'exemples :




Temps de calcul de la descente de gradient stochastique sur un ensemble de 1 millions de données : 118 min.

Observations :

La descente de gradient batch se dirige directement vers une solution optimale, car on peut constater que la courbe obtenue est assez lisse. Le fait d'obtenir une courbe assez lisse est dû au fait que nous faisons la moyenne de tous les gradients des données d'entrainement à chaque itération.

Le problème est que sur un ensemble de données très grand (ici 1 million), pour faire seulement un pas, le modèle devra calculer les gradients de tous les 1 millions d'exemples. Cela ne semble pas très efficace. Ainsi, sur un ensemble de données trop énorme, il est préférable d'utiliser une descente de gradient stochastique. La descente de gradient nous permettra de résoudre le problème identifier précédemment.

Dans le cas de la descente de gradient stochastique, on considère 1 seul exemple à la fois pour calculer le gradient. Puisque nous considérons un seul exemple à la fois, le coût fluctuera au fil des exemples de formation et ne diminuera pas nécessairement. Mais à long terme, on voit le coût diminuer avec les fluctuations.

En résumé la descente de gradient batch est utilisée pour des courbes plus lisses. La descente de gradient batch converge directement vers le minimum. Tandis que la descente de gradient stochastique converge plus rapidement pour les jeux de données plus volumineux. Le problème avec la descente de gradient stochastique est qu'elle ralentit le temps de calcul.

Pour résoudre ce problème on fait un mélange entre la descente de gradient stochastique et la descente de gradient batch. Cela permettra de réduire le temps de calcul et d'obtenir les avantages des deux précédentes variantes. Ce modèle s'appelle la descente de gradient mini-batch.

Le biais :

Maintenant On va s'intéresser au rôle du biais pour le perceptron.

Hypothèse : Grâce au biais, la fonction d'activation (c'est la fonction calculée en faisant le produit scalaire entre w et un exemple de l'ensemble de données) va être décalée (car le biais est soustrait à la Somme des entrées pondérées par les poids) et le perceptron aura donc de plus grandes opportunités d'apprentissage.

Variations de max_iter entre 10 et 1000 avec biais :

```
for i in range (10,1100,100):
    meantrain=[]
    meantest=[]
    for j in range(0,10):
        perceptron = Lineaire(hinge,hinge_g,max_iter=i,eps=0.1,biais=True)
        perceptron.fit(trainx,trainy,"batch")
        meantrain.append(perceptron.score(trainx,trainy))
        meantest.append(perceptron.score(testx,testy))
    print("Erreur moyenne : train %f, test %f, max_iter %d"% (np.mean(meantrain),np.mean(meantest),i))
```

```
Erreur moyenne : train 86.360000, test 84.840000, max_iter 10
Erreur moyenne : train 91.310000, test 89.980000, max_iter 110
Erreur moyenne : train 90.990000, test 90.380000, max_iter 210
Erreur moyenne : train 91.000000, test 90.260000, max_iter 310
Erreur moyenne : train 84.970000, test 84.460000, max_iter 410
Erreur moyenne : train 83.470000, test 82.640000, max_iter 510
Erreur moyenne : train 90.980000, test 90.440000, max_iter 610
Erreur moyenne : train 90.910000, test 90.270000, max_iter 710
Erreur moyenne : train 90.940000, test 90.390000, max_iter 810
Erreur moyenne : train 90.930000, test 90.300000, max_iter 910
Erreur moyenne : train 82.870000, test 82.410000, max_iter 1010
```

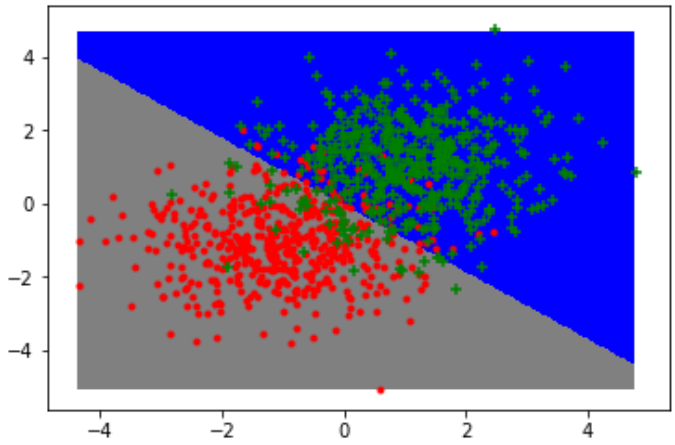
D'après les résultats obtenus ci-dessus, le biais n'a pas l'air d'augmenter les accuracy moyenne calculé par le perceptron.

# Régression linéaire :

On va maintenant s'intéresser à la régression linéaire, en utilisant comme fonction de coût l'erreur quadratique moyenne pour appliquer la descente de gradient.

Les frontières obtenues dans l'espace de représentation des exemples :

```
Erreur : train 90.200000, test 91.800000
```




Variations de max_iter entre 10 et 1000 avec biais :

```
for i in range (10,1100,100):
    meantrain=[]
    meantest=[]
    for j in range(0,10):
        regression = Lineaire(mse,mse_g,max_iter=i,eps=0.1,biais=False)
        regression.fit(trainx,trainy,"batch")
        meantrain.append(regression.score(trainx,trainy))
        meantest.append(regression.score(testx,testy))
    print("Erreur moyenne : train %f, test %f, max_iter %d"% (np.mean(meantrain),np.mean(meantest),i))
```

```
Erreur moyenne : train 91.400000, test 92.060000, max_iter 10
Erreur moyenne : train 91.500000, test 92.200000, max_iter 110
Erreur moyenne : train 91.500000, test 92.200000, max_iter 210
Erreur moyenne : train 91.500000, test 92.200000, max_iter 310
Erreur moyenne : train 91.500000, test 92.200000, max_iter 410
Erreur moyenne : train 91.500000, test 92.200000, max_iter 510
Erreur moyenne : train 91.500000, test 92.200000, max_iter 610
Erreur moyenne : train 91.500000, test 92.200000, max_iter 710
Erreur moyenne : train 91.500000, test 92.200000, max_iter 810
Erreur moyenne : train 91.500000, test 92.200000, max_iter 910
Erreur moyenne : train 91.500000, test 92.200000, max_iter 1010
```

On peut constater une très petite amélioration de l'accuracy moyenne calculée pour l'ensemble de test, par rapport aux résultats obtenue avec le perceptron dans les cas précédent. Sur une base de 1000 exemples 2D, en utilisant la régression linéaire on peut obtenir une accuracy moyenne de 92.2%, tandis qu'avec le perceptron on plafonnait à 91.4%.

# DONNEES USPS :

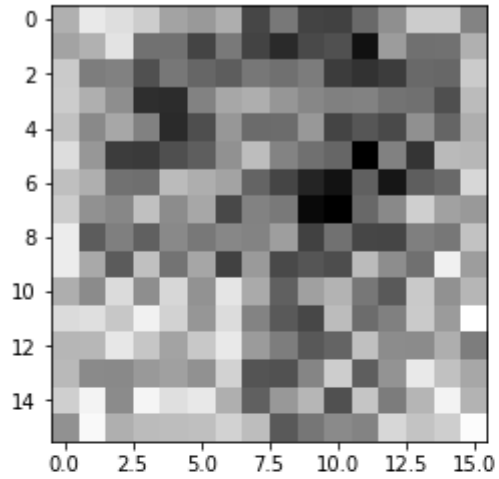
Dans les données USPS, on se retrouve face à un problème Multi classe, et non binaire. Il y a 10 classes (allant de 0 à 9). Pour pouvoir entrainer nos modèles, on sélectionne 2 classes parmi les 10, et on récupère les données correspondantes. Puis on change le nom de ces classes en -1 et 1, pour permettre à notre modèle de pouvoir apprendre.

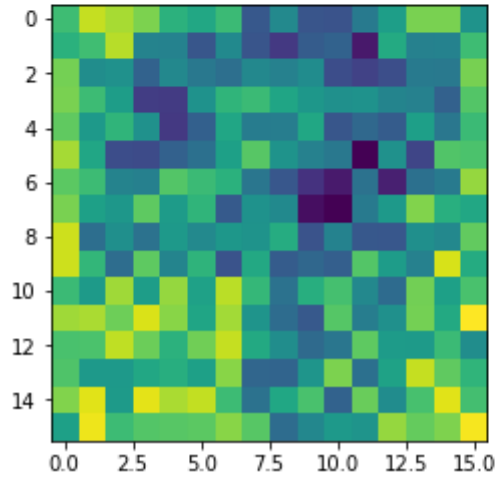
Comme modèle, je vais utiliser le perceptron dans mes expériences.

Pour chaque classe je vais montrer le vecteur de poids obtenue avec la fonction show_usps.

Classe 6 vs classe 9 :

```
Erreur : train 100.000000, test 99.135447
```

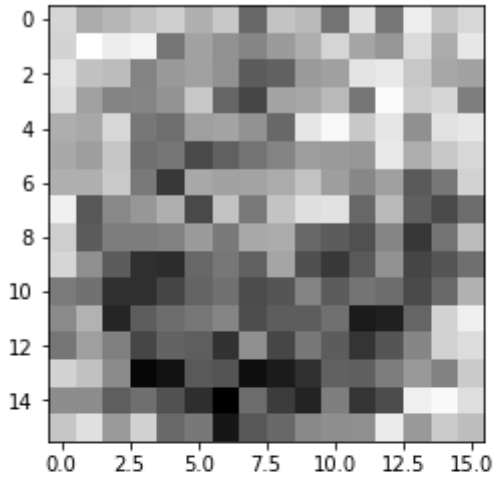


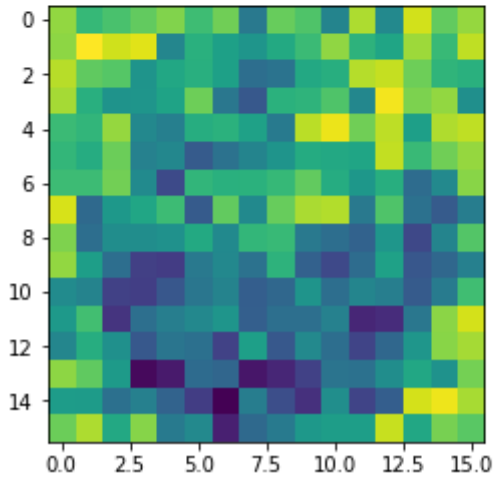




On peut observer que la matrice de poids w prend la forme d'un 9. Ainsi après apprentissage, le perceptron est en mesure de reconnaître les chiffres qui sont des 9 et ceux qui ne le sont pas.

Classe 9 vs 6 :

```
Erreur : train 100.000000, test 99.135447
```

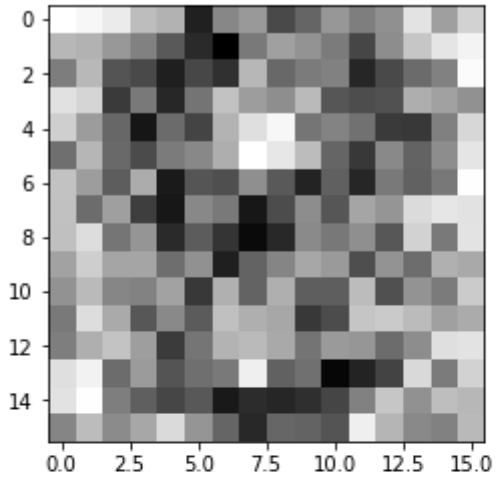


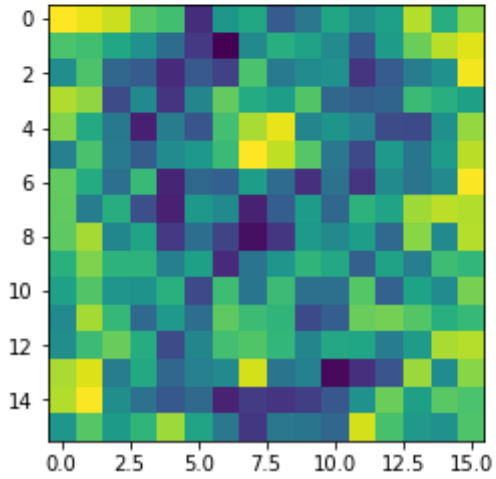




On peut observer que la matrice de poids w prend la forme d'un 6. Ainsi après apprentissage, le perceptron est en mesure de reconnaître les chiffres qui sont des 6 et ceux qui ne le sont pas.

Classe 1 vs classe 8 :

```
Erreur : train 99.935359, test 97.441860
```

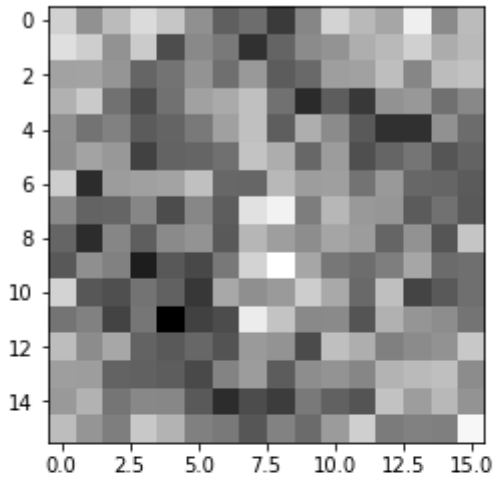


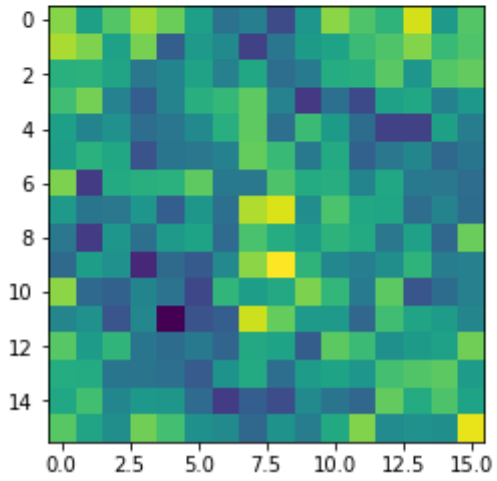




On peut observer que la matrice de poids w prend la forme d'un 8. Ainsi après apprentissage, le perceptron est en mesure de reconnaître les chiffres qui sont des 8 et ceux qui ne le sont pas.

Classe 5 vs classe 0 :

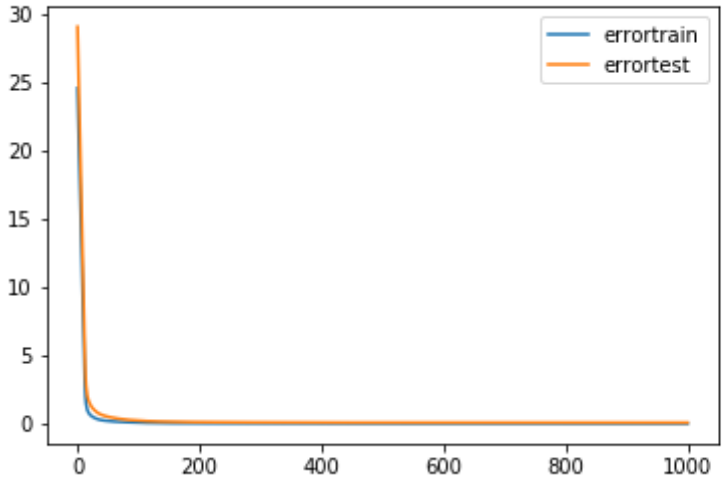
```
Erreur : train 98.971429, test 96.917148
```




On peut observer que la matrice de poids w prend la forme d'un 0. Ainsi après apprentissage, le perceptron est en mesure de reconnaître les chiffres qui sont des 0 et ceux qui ne le sont pas.

Courbe de l'erreur en apprentissage et en test en fonction du nombre d'itérations (pour la classe 1 vs classe 8) :

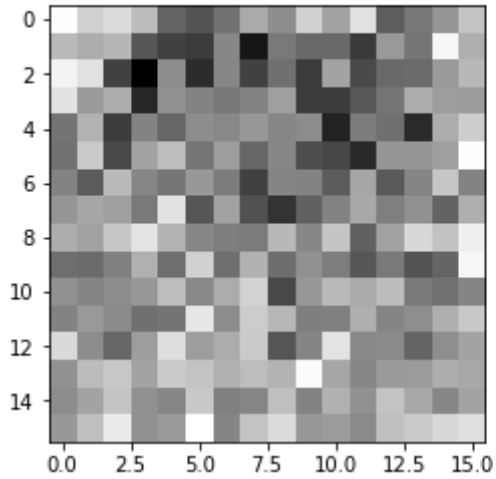


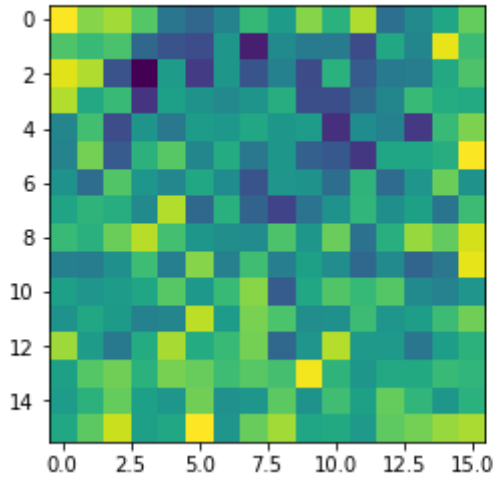

On peut observer la courbe sur l'erreur en test colle presque parfaitement avec celle de l'erreur en apprentissage. Nous ne sommes donc pas dans un cas de sur-apprentissage.

Désormais on va entrainer notre perceptron avec une classe contre toutes les autres.

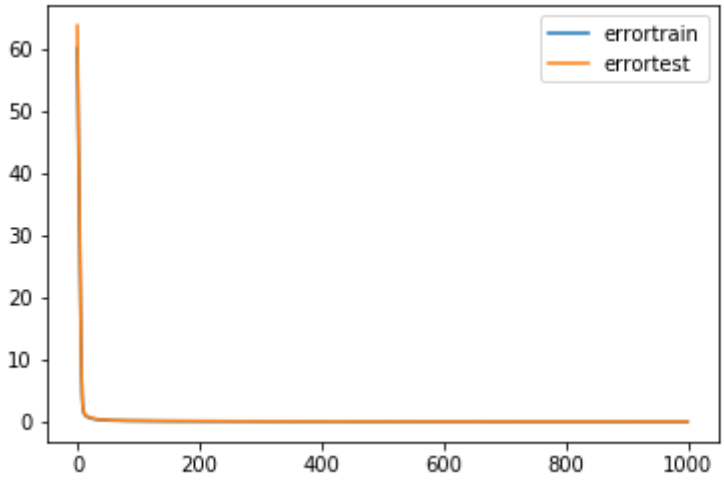
Classe 6 vs les autres :

```
Erreur : train 98.450144, test 97.558545
```



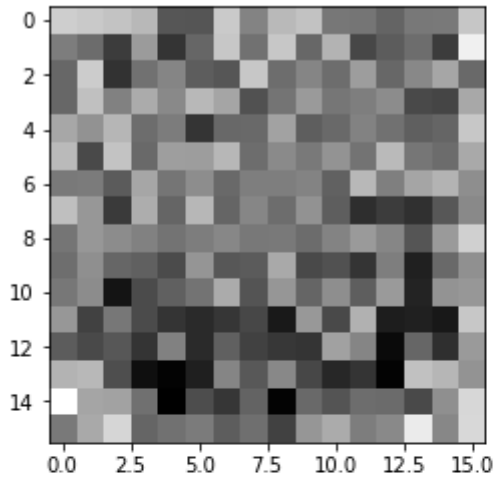
Courbe de l'erreur en apprentissage et en test en fonction du nombre d'itérations (pour la classe 6 vs les autres) :

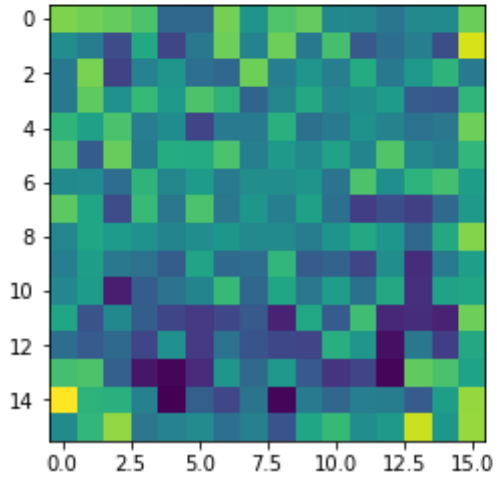



On peut observer la courbe sur l'erreur en test colle presque parfaitement avec celle de l'erreur en apprentissage. Nous ne sommes donc pas dans un cas de sur-apprentissage.

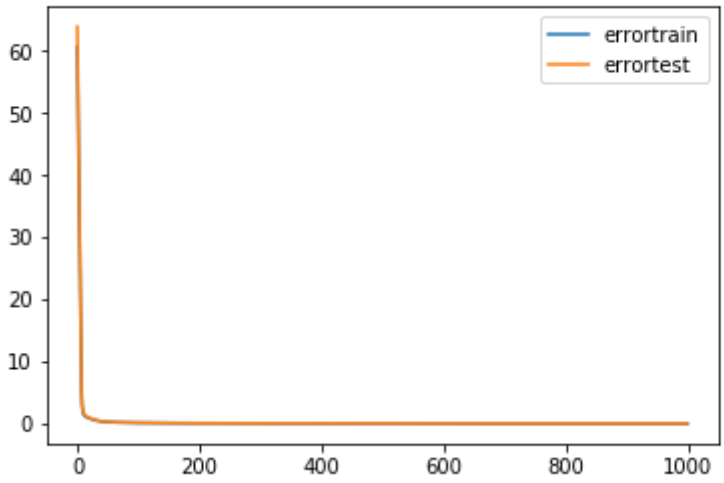
Classe 9 vs les autres :

```
Erreur : train 96.790564, test 96.412556
```



Courbe de l'erreur en apprentissage et en test en fonction du nombre d'itérations (pour la classe 9 vs les autres) :




Sur cet exemple aussi, la courbe de l'erreur en test colle parfaitement avec celle de l'erreur en apprentissage. Donc il n'y a pas de sur-apprentissage ici aussi.

# Projection :

En appliquant une projection polynomiale sur les données, on obtient les résultats suivants :

```
Erreur moyenne : train 92.380000, test 90.400000
```

En ce qui concerne la projection gaussienne, j'ai tout d'abord chercher à optimiser le paramètre sigma ( représenté ici par j):

```
Erreur: train 89.000000, test 88.400000, j: 0.500000
Erreur: train 90.400000, test 90.000000, j: 1.000000
Erreur: train 90.600000, test 90.000000, j: 1.500000
Erreur: train 89.400000, test 88.500000, j: 2.000000
Erreur: train 90.700000, test 89.900000, j: 2.500000
Erreur: train 87.600000, test 89.400000, j: 3.000000
Erreur: train 81.200000, test 79.600000, j: 3.500000
Erreur: train 88.200000, test 89.400000, j: 4.000000
Erreur: train 91.600000, test 90.700000, j: 4.500000
```

```
Erreur: train 87.500000, test 86.500000, j: 1.000000
Erreur: train 87.100000, test 85.700000, j: 2.000000
Erreur: train 86.700000, test 87.500000, j: 3.000000
Erreur: train 86.700000, test 85.500000, j: 4.000000
Erreur: train 87.100000, test 87.100000, j: 5.000000
Erreur: train 91.400000, test 90.700000, j: 6.000000
Erreur: train 90.600000, test 90.200000, j: 7.000000
Erreur: train 91.300000, test 90.800000, j: 8.000000
Erreur: train 91.200000, test 90.800000, j: 9.000000
```

Et ensuite en appliquant une projection gaussienne sur les données, on obtient les résultats suivants :

```
Erreur moyenne : train 89.990000, test 88.180000
```