Rapport_TP4_Corvisier_Kheldouni

May 21, 2017

1 Perceptron, SVMs

authors : Jean-Christophe CORVISIER, Amine KHELDOUNI

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        from tools import *
        import numpy.random as rd
        import random as r
```

1.1 Implémentation du perceptron

```
In [2]: # Question 1
        class Perceptron:
            def __init__(self,max_iter = 1000,eps=1e-3,projection = None):
                self.max_iter = max_iter
                self.eps = eps
                self.projection = projection or (lambda x: x) #projection fonction identite par

            def fit(self,data,y):
                data = self.projection(data)
                self.w = np.random.random((1,data.shape[1]))
                self.histo_w  = np.zeros((self.max_iter,data.shape[1]))
                self.histo_f = np.zeros((self.max_iter,1))
                ylab=set(y.flat)
                if len(ylab)!=2:
                    print("pas bon nombres de labels (%d)" % (ylab,))
                    return
                self.labels = {-1: min(ylab), 1:max(ylab)}
                y = 2*(y!=self.labels[-1])-1
                i=0
                while i<self.max_iter:
                    idx = range(len(data))
                    for j in idx:
                        self.w = self.w - self.get_eps()*self.loss_g(data[j],y[j:(j+1)])
                    self.histo_w[i]=self.w
                    self.histo_f[i]=self.loss(data, y)
```

```
                #if i % 100==0: print(i,self.histo_f[i])
                i+=1
        def predict(self,data):
            data = self.projection(data)
            return np.array([self.labels[int(x)] for x in np.sign(data.dot(self.w.T)).flat])
        def score(self,data,y):
            return np.mean(self.predict(data)==y)
        def get_eps(self):
            return self.eps
        def loss(self,data,y):
            return hinge(self.w,data,y)
        def loss_g(self,data,y):
            return grad_hinge(self.w,data,y)


    def hinge(w, data, y, alpha=0):
        data, y, w = data.reshape(len(y), -1), y.reshape(-1, 1), w.reshape(1, -1)
        return np.mean(np.maximum(0, -y * data.dot(w.T)))

    def grad_hinge(w, data, y, alpha=0):
        data, y, w = data.reshape(len(y), -1), y.reshape(-1, 1), w.reshape(1, -1)
        return -np.mean(((-y * data.dot(w.T) >= 0)) * data * y, 0)
```

Dans cette première partie, nous implémentons la fonction de coût ainsi que son gradient pour étudier une convergence par descente de gradient (et certaines variantes). Une fois ces fonctions implémentées, notre Perceptron pourra effectuer un apprentissage sur des données d'entraînement pour ensuite faire de la prédiction grâce à la convergence d'une descente de gradient.

La fonction de coût considérée est une fonction "hinge loss" :

$$l^{\alpha}(y, f_w(x)) = max(0, \alpha - y f_w(x))$$

en considérant que le paramètre α est nul.

Les dérivées partielles de cette fonction de coût s'exprime alors comme suit :

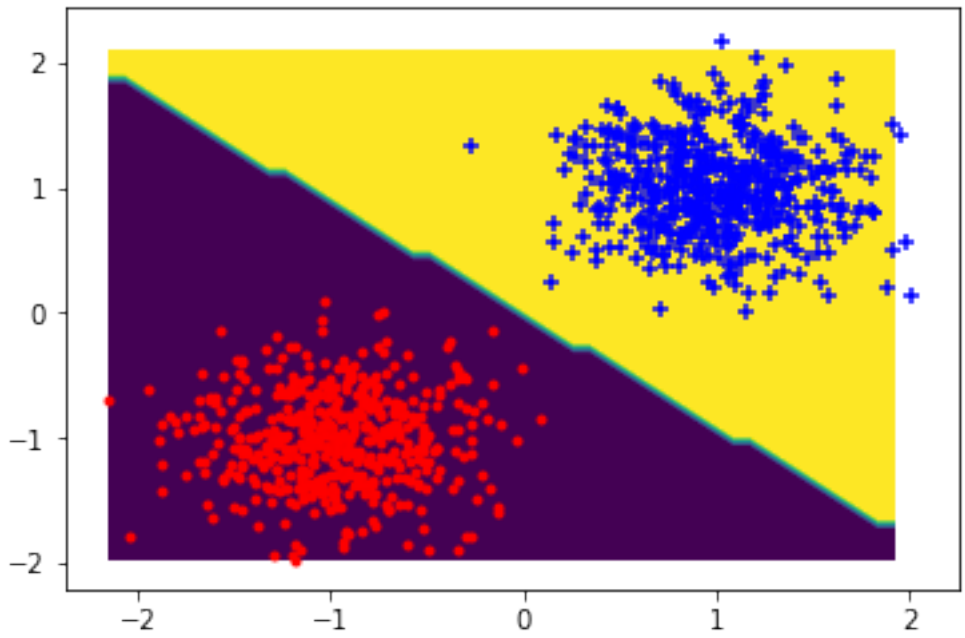
$$\frac{\partial l_{hinge}}{\partial w} = \begin{cases} -yx & \text{si } y f_w(x) < 0 \\ 0 & \text{sinon} \end{cases}$$

```
In [3]: ### Test des fonctions hinge, grad_hinge
        w = np.random.random((3,))
        data = np.random.random((100,3))
        y = np.random.randint(0,2,size = (100,1))*2-1
        print('Test de la fonction Hinge et de son gradient : ')
        print(hinge(w,data,y), hinge(w,data[0],y[0]), hinge(w,data[0,:],y[0]).shape)
        print(grad_hinge(w,data,y), grad_hinge(w,data[0],y[0]).shape,grad_hinge(w,data[0,:],y[0]
```

```
Test de la fonction Hinge et de son gradient :
0.338140862259 0.0 ()
[ 0.23763595  0.19999729  0.19958149] (3,) (3,)
```

In [4]:
```
# Question 1.2

### Generation de donnees de type 0
xtrain,ytrain = gen_arti(data_type=0,epsilon=0.2)
xtest,ytest = gen_arti(data_type=0,epsilon=0.2)

plt.ion()

### Apprentissage et test
model=Perceptron(max_iter=200, eps=1e-3)
model.fit(xtrain,ytrain)
print("score en train : ",model.score(xtrain,ytrain))
print("score en test : ",model.score(xtest,ytest))

# Question 1.3
#### Tracer de frontiere
plt.figure()
plot_frontiere(xtrain, model.predict,50)
plot_data(xtrain,ytrain)
```

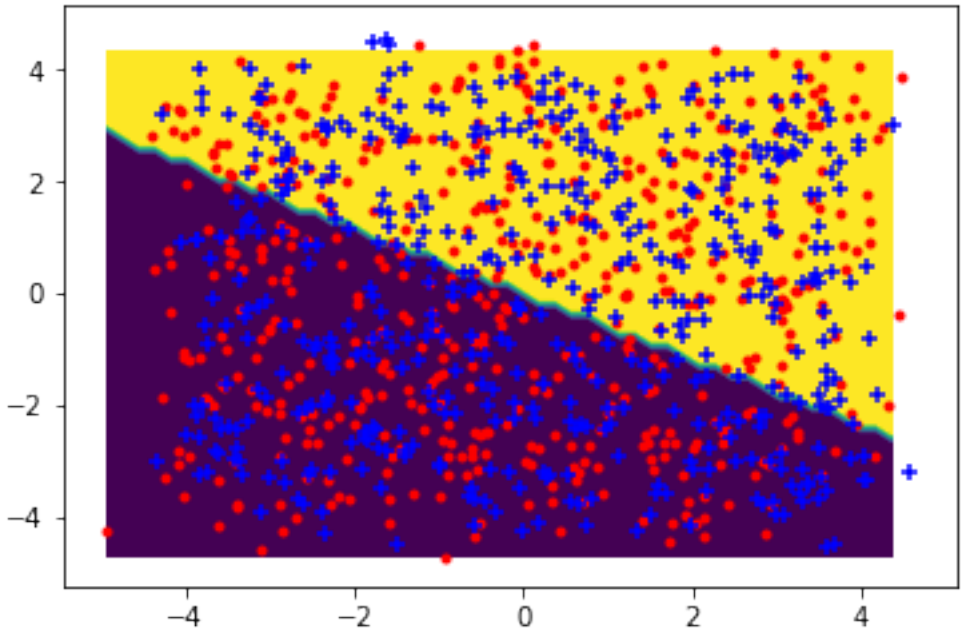
```
score en train :  1.0
score en test :  1.0
```



In [5]:
```
### Generation de donnees 2
xtrain,ytrain = gen_arti(data_type=2,epsilon=0.4)
```

```
        xtest,ytest = gen_arti(data_type=2,epsilon=0.4)

        plt.ion()

        ### Apprentissage
        model=Perceptron(max_iter=400, eps=1e-3)
        model.fit(xtrain,ytrain)
        print("score en train : ",model.score(xtrain,ytrain))
        print("score en test : ",model.score(xtest,ytest))

        #### Tracer de frontiere
        plt.figure()
        plot_frontiere(xtrain, model.predict,50)
        plot_data(xtrain,ytrain)
```

```
score en train :  0.497
score en test :  0.468
```



Pour un jeu de données de type 0, les ensembles de points sont bien séparables. On obtient alors une droite séparant les deux classes.Dans ce premier cas, les scores d'apprentissage et d'entraînement sont à 100% car la frontière sépare très bien l'espace en deux sous catégories.

Par contre, pour un jeu de données plus difficile à séparer de plus grande variance, la classification par notre Perceptron est beaucoup moins performante. En effet, les exemples étant mélangés, il devient impossible de classifier les données à l'aide d'une droite. Les scores d'apprentissage et de test sont alors plus de l'ordre des 50% pour des données du type 2.

Pour des données plus complexes, il faut donc augmenter la dimentionnalité grâce à un plongement des données qu'on verra plus tard en section 3, au lieu d'utiliser le classifieur linéaire basique.

In [6]:
```
# Question 1.4

xtrain,ytrain = gen_arti(data_type=0,epsilon=0.3)
xtest,ytest = gen_arti(data_type=0,epsilon=0.3)

def batchGrad(data, y, max_iter=5000, alpha = 0.001, eps=10**(-4)):
    """ Descente de gradient (batch)"""
    w = np.random.random((data.shape[1],))
    k = 1
    grad = grad_hinge(w, data, y)
    while (k < max_iter and np.linalg.norm(grad) > eps):
        w = w - alpha * grad
        grad = grad_hinge(w, data, y)
        k += 1
    print('La norme du gradient batchgrad à la dernière itération vaut : ')
    print(grad)
    print('Dernière itération de la descente : ')
    print(k)
    return w

wBatch_opt = batchGrad(xtrain, ytrain)
print('Le vecteur w optimal batchgrad trouvé est : ')
print(wBatch_opt)


def batch_iter(y, tx, batch_size, num_batches=1, shuffle=True):
    """ Fonction retournant un sous vecteur de y et une sous matrice des données
    mélanger par permutations """
    data_size = tx.shape[0]
    if shuffle:
        shuffle_indices = np.random.permutation(np.arange(data_size))
        shuffled_y = y[shuffle_indices]
        shuffled_tx = tx[shuffle_indices]
    else:
        shuffled_y = y
        shuffled_tx = tx
    for batch_num in range(num_batches):
        start_index = batch_num * batch_size
        end_index = min((batch_num + 1) * batch_size, data_size)
        if start_index != end_index:
            return shuffled_y[start_index:end_index], shuffled_tx[start_index:end_index]

def stochasticGrad(data, y, batch_size, max_iter=1000, eta = 0.01, eps=10**(-10)):
    """ Descente de gradient stochastique """
```

```
        w = rd.random((data.shape[1],))
        k = 1
        grad = grad_hinge(w, data, y)
        while (k < max_iter and np.linalg.norm(grad) > eps):
            ly, lx = batch_iter(y, data, batch_size)
            for i in range(data.shape[0]):
                w = w - eta * grad_hinge(w, lx[i], np.array([ly[i]]))
            grad = grad_hinge(w, data, y)
            k += 1

        print('La norme du gradient stochastique à la dernière itération vaut : ')
        print(grad)
        print('Dernière itération de la descente : ')
        print(k)
        return w

    wStochastic_opt = stochasticGrad(xtrain, ytrain, xtrain.shape[0])
    print('Le vecteur w optimal du gradient stochastique trouvé est : ')
    print(wStochastic_opt)

    # Question 1.5

    def step(t):
        return 1./(t+1)**2

    def VariableGradient(data, y, max_iter=1000, eps=10**(-6)):
        """ Gradient à pas variable """
        w = w = np.random.random((data.shape[1],))
        k = 1
        grad = grad_hinge(w, data, y)
        while (k < max_iter and np.linalg.norm(grad) > eps):
            w = w - step(k) * grad
            grad = grad_hinge(w, data, y)
            k += 1

        print('La norme du gradient à la dernière itération vaut : ')
        print(grad)
        print('Dernière itération de la descente : ')
        print(k)
        return w


    wVarStep_opt = VariableGradient(xtrain, ytrain)
    print('Le vecteur w optimal trouvé est : ')
    print(wVarStep_opt)
```

```
La norme du gradient batchgrad à la dernière itération vaut :
[-0. -0.]
```

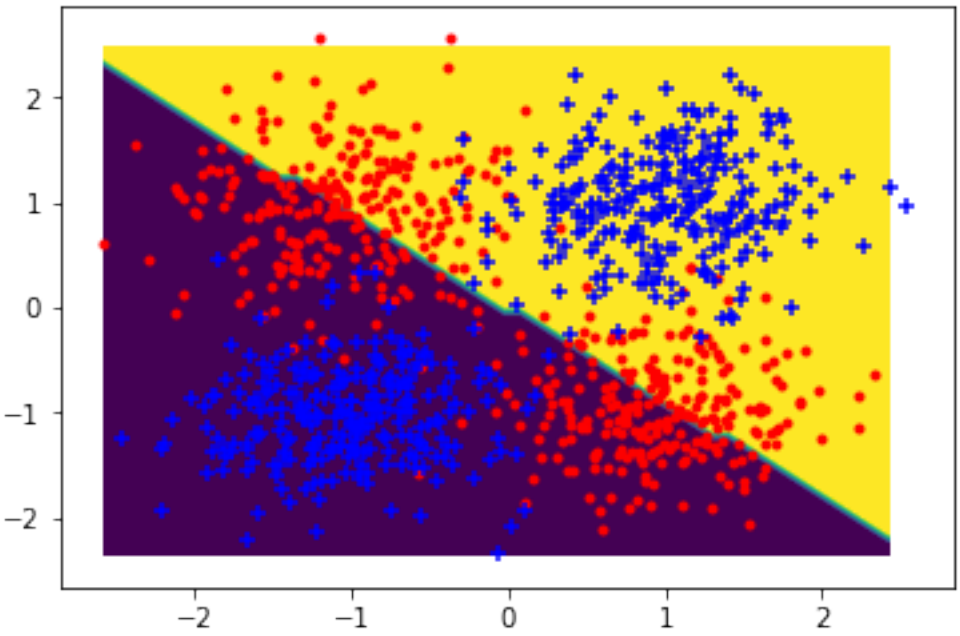
```
Dernière itération de la descente :
1
Le vecteur w optimal batchgrad trouvé est :
[ 0.93201097  0.73660559]
La norme du gradient stochastique à la dernière itération vaut :
[-0. -0.]
Dernière itération de la descente :
137
Le vecteur w optimal du gradient stochastique trouvé est :
[ 0.57942856  0.56546205]
La norme du gradient à la dernière itération vaut :
[-0. -0.]
Dernière itération de la descente :
1
Le vecteur w optimal trouvé est :
[ 0.61492781  0.57980384]
```

Les fonctions de descente de gradient programmées ci-dessus fonctionnent correctement. En effet, la valeur du gradient trouvée est très faible (approximant 0). On remarque évidemment une convergence plus rapide pour la descente de gradient stochastique en comparaison à la descente batch.

```
In [7]: #Création d'une classe utilisant les fonctions d'optimisation développées
        class GradientTestPerceptron:
            def __init__(self,max_iter = 1000,eps=1e-3,projection = None):
                self.max_iter = max_iter
                self.eps = eps
                self.projection = projection or (lambda x: x) #projection fonction identite par

            def fit(self,data,y):
                self.histo_w  = np.zeros((self.max_iter,data.shape[1]))
                self.histo_f = np.zeros((self.max_iter,1))
                ylab=set(y.flat)
                if len(ylab)!=2:
                    print("pas bon nombres de labels (%d)" % (ylab,))
                    return
                self.labels = {-1: min(ylab), 1:max(ylab)}
                y = 2*(y!=self.labels[-1])-1
                self.w=VariableGradient(data, y, max_iter=1000, eps=10**(-6))

            def predict(self,data):
                data = self.projection(data)
                return np.array([self.labels[int(x)] for x in np.sign(data.dot(self.w.T)).flat])
            def score(self,data,y):
                return np.mean(self.predict(data)==y)
            def get_eps(self):
                return self.eps
```

```
        def loss(self,data,y):
            return hinge(self.w,data,y)
        def loss_g(self,data,y):
            return grad_hinge(self.w,data,y)


    ### Generation de donnees 1
    xtrain,ytrain = gen_arti(data_type=1,epsilon=0.4)
    xtest,ytest = gen_arti(data_type=1,epsilon=0.4)

    plt.ion()

    ### Apprentissage
    model=GradientTestPerceptron(max_iter=400, eps=1e-3)
    model.fit(xtrain,ytrain)
    print("score en train : ",model.score(xtrain,ytrain))
    print("score en test : ",model.score(xtest,ytest))

    #### Tracer de frontiere
    plt.figure()
    plot_frontiere(xtrain, model.predict,50)
    plot_data(xtrain,ytrain)
```

```
La norme du gradient à la dernière itération vaut :
[ 0.29169604  0.34747375]
Dernière itération de la descente :
1000
score en train :  0.514
score en test :  0.49
```



L'ajout des méthodes de descente de gradient au Perceptron initial fournit en effet une bonne convergence des données vers le vecteur w_{opt}. On obtient alors les mêmes résultats pour des données de type 0 (séparables).

Néanmoins, pour des données plus compliquées, les descentes de gradient ne fournissent aucune solution à la multi dimensionnalité de l'espace et ne fournit donc pas de solution plus performante que celles mentionnées plus haut.

Enfin, ces algorithmes permettent une convergence peut-être plus rapide vers l'optimum, mais pour contourner le problème du mélange de données (ou du bruit sur les données), il faudra utiliser une autre méthode augmentant la dimentionnalité de notre frontière.

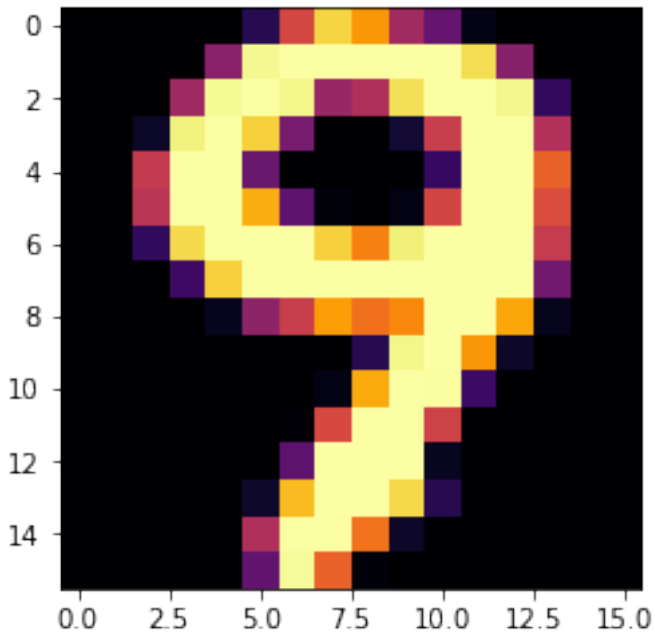
1.2 Données USPS

```
In [8]: def load_usps(filename):
            with open(filename,"r") as f:
                f.readline()
                data =[ [float(x) for x in l.split()] for l in f if len(l.split())>2]
            tmp = np.array(data)
            return tmp[:,1:],tmp[:,0].astype(int)

        def get_usps(l,datax,datay):
            """ l : liste des chiffres a extraire"""
            if type(l)!=list:
                resx = datax[datay==l,:]
                resy = datay[datay==l]
                return resx,resy
            tmp =   list(zip(*[get_usps(i,datax,datay) for i in l]))
```

```
        tmpx,tmpy = np.vstack(tmp[0]),np.hstack(tmp[1])
        idx = np.random.permutation(range(len(tmpy)))
        return tmpx[idx,:],tmpy[idx]

    def show_usps(data):
        plt.imshow(data.reshape((16,16)),interpolation="nearest",cmap="inferno")
```

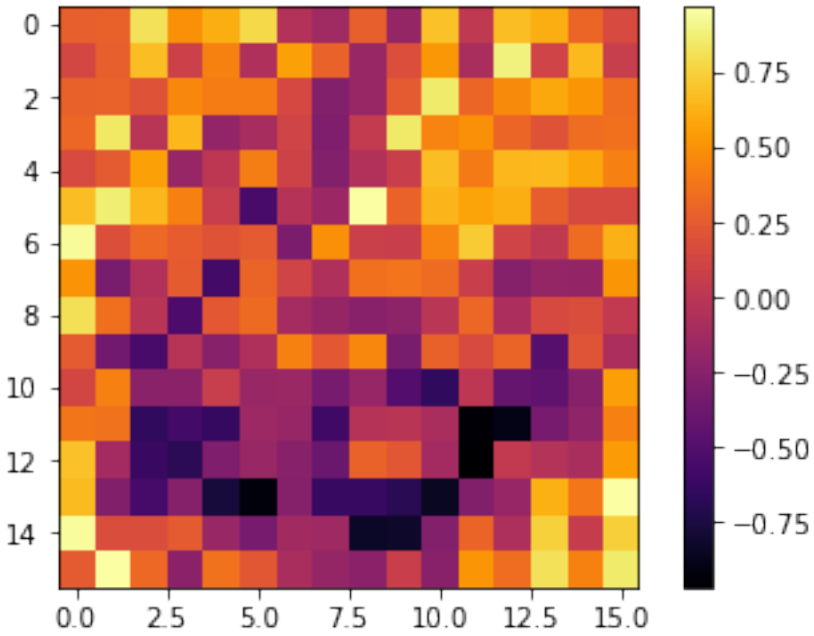
In [9]:
```
#Question 2.1
xuspstrain,yuspstrain = load_usps("USPS_train.txt")
xuspstest,yuspstest = load_usps("USPS_test.txt")
```

In [10]:
```
# 6 vs 9
xtrain,ytrain = get_usps([6,9],xuspstrain,yuspstrain)
xtest,ytest = get_usps([6,9],xuspstest,yuspstest)
show_usps(xtest[0])
plt.show()

model=Perceptron(max_iter=300, eps=1e-3)
model.fit(xtrain, ytrain)
print("score en train : ",model.score(xtrain,ytrain))
print("score en test : ",model.score(xtest,ytest))

show_usps(model.w)
plt.colorbar()
```



```
score en train :  1.0
score en test :  0.99711815562
```

```
Out[10]: <matplotlib.colorbar.Colorbar at 0x1ae88d05e80>
```

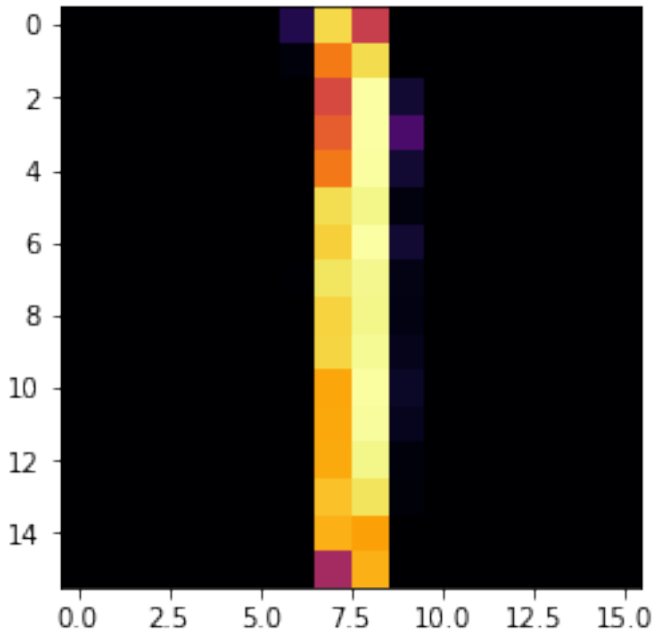


En confrontant un chiffre contre un autre (6 vs 9), on remarque que la convergence est très rapide. En effet, au bout d'une centaine d'itérations, on obtient déjà une erreur presque nulle. L'apprentissage se fait parfaitement (avec un score de 100%) et le test du Perceptron fournit un très bon score également voisinant les 99.42%.

La matrice de poids obtenue permet en effet de deviner le chiffre (6 par exemple sur la matrice de poids ci-dessus).

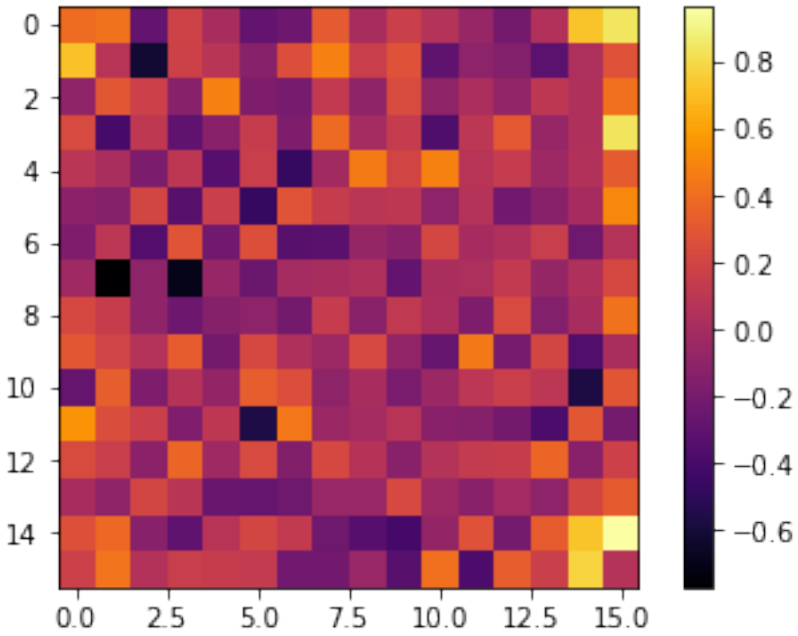
```
In [11]: # 1,2 vs 6,8

         def train1268(val):
             if (val == 1 or val == 2):
                 return 1
             else:
                 return 0

         train1268 = np.vectorize(train1268)

         xtrain,ytrain = get_usps([1,2,6,8],xuspstrain,yuspstrain)
         xtest,ytest = get_usps([1,2,6,8],xuspstest,yuspstest)
         ytrain = train1268(ytrain)
         ytest = train1268(ytest)
```

```
show_usps(xtest[0])
plt.show()

model=Perceptron(max_iter=300, eps=1e-3)
model.fit(xtrain, ytrain)
print("score en train : ",model.score(xtrain,ytrain))
print("score en test : ",model.score(xtest,ytest))

show_usps(model.w)
plt.colorbar()
```




```
score en train :  0.972127804215
score en test :  0.924812030075
```

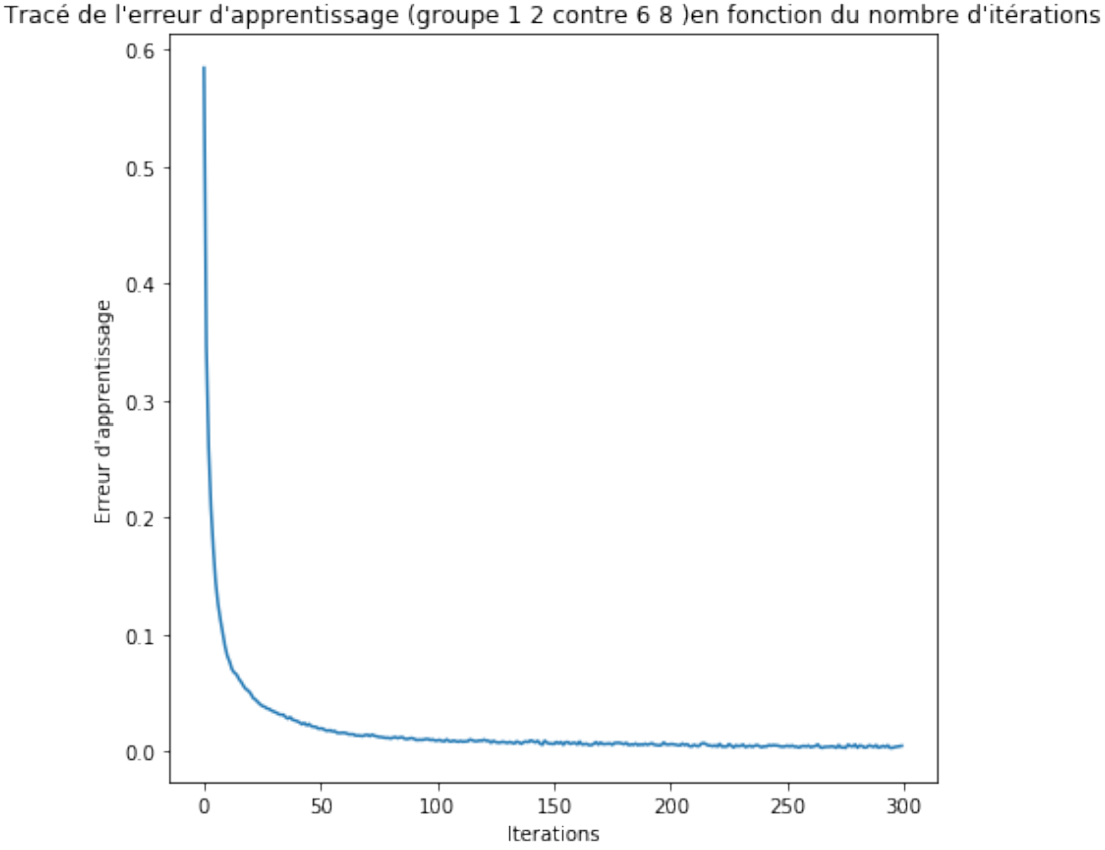
```
Out[11]: <matplotlib.colorbar.Colorbar at 0x1ae88b8e748>
```

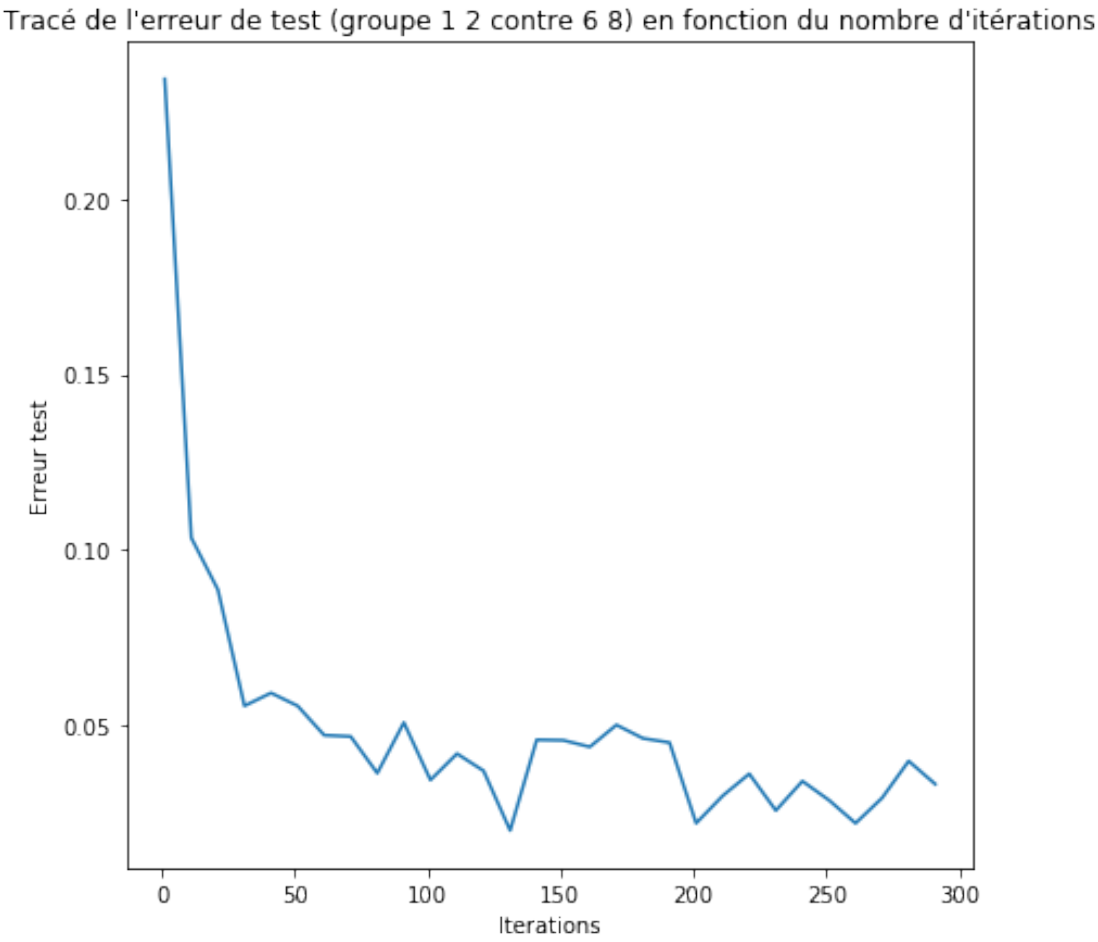


Par ailleurs, en confrontant un sous ensemble de chiffre avec un autre (1,2 vs 6,8), l'entraînement se fait toujours aussi bien (99%). Et les résultats fournits par le Perceptron pour des données de test est un peu moins bon, mais reste très correct (91.85%).

In [12]:
```
# Question 2.2 : Tracé des erreurs

#le cas donné ici est celui du groupe de chiffre 1 2 contre le groupe 6 8
#erreur d'apprentissage sur cette configuration
plt.figure(figsize=(7,7))
plt.plot(model.histo_f)
plt.xlabel('Iterations')
plt.ylabel("Erreur d'apprentissage")
plt.title("Tracé de l'erreur d'apprentissage (groupe 1 2 contre 6 8 )en fonction du nom
plt.show()
# Erreur de test
def testError(nb_iter):
    plt.figure(figsize=(7,7))
    model=Perceptron(max_iter=1, eps=1e-3)
    errorTest = []
    for k in range(1, nb_iter, 10):
        xtrain,ytrain = get_usps([1,2,6,8],xuspstrain,yuspstrain)
        xtest,ytest = get_usps([1,2,6,8],xuspstest,yuspstest)
        ytrain = train1268(ytrain)
        ytest = train1268(ytest)
        model.max_iter = k
        model.fit(xtrain, ytrain)
```

```
            errorTest.append(hinge(model.w, xtest, ytest))
        plt.plot(range(1, nb_iter, 10), errorTest)
        plt.xlabel('Iterations')
        plt.ylabel('Erreur test')
        plt.title("Tracé de l'erreur de test (groupe 1 2 contre 6 8) en fonction du nombre
        plt.show()

testError(300)
```





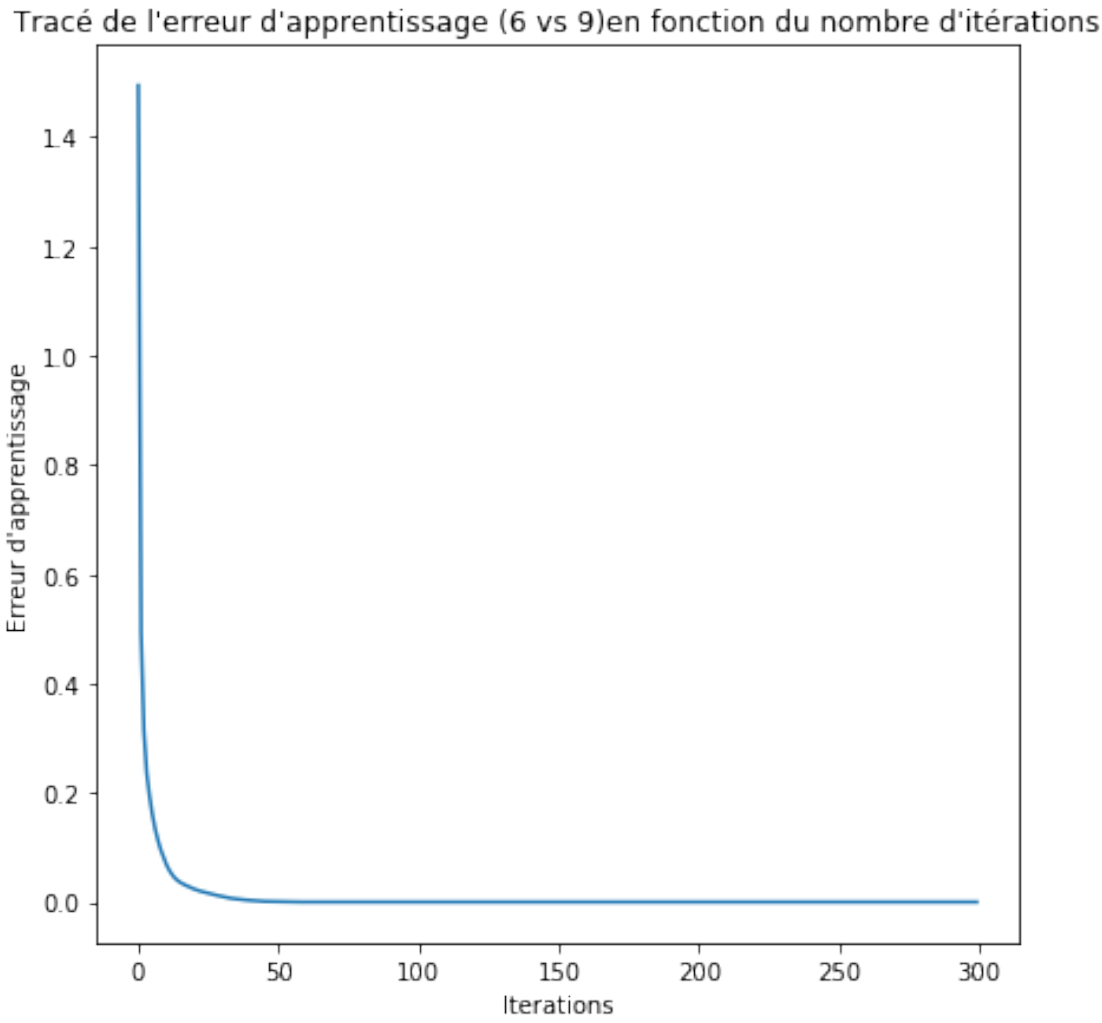
On remarque que les erreurs d'apprentissage et de test décroissent rapidement sur les 100 premières itérations. L'erreur d'apprentissage a en effet un forte convergence vers 0. Par contre, l'erreur de test, bien que décroissante vers la même valeur, fluctue notablement par la suite (le minimum de l'erreur de test est atteint à 260 itérations puis l'erreur de test réaugmente).

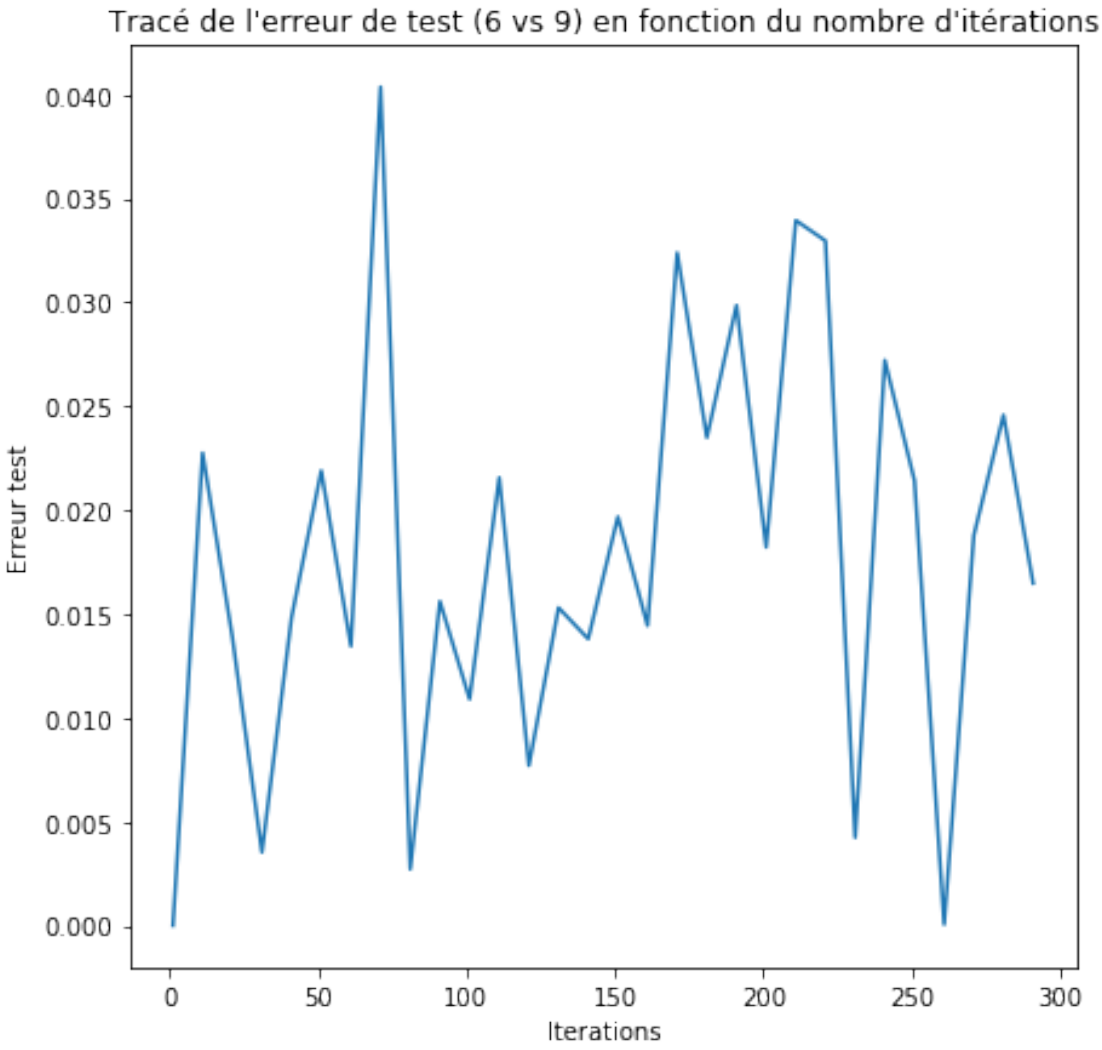
On a donc bien un phénomène de sur-apprentissage des données.

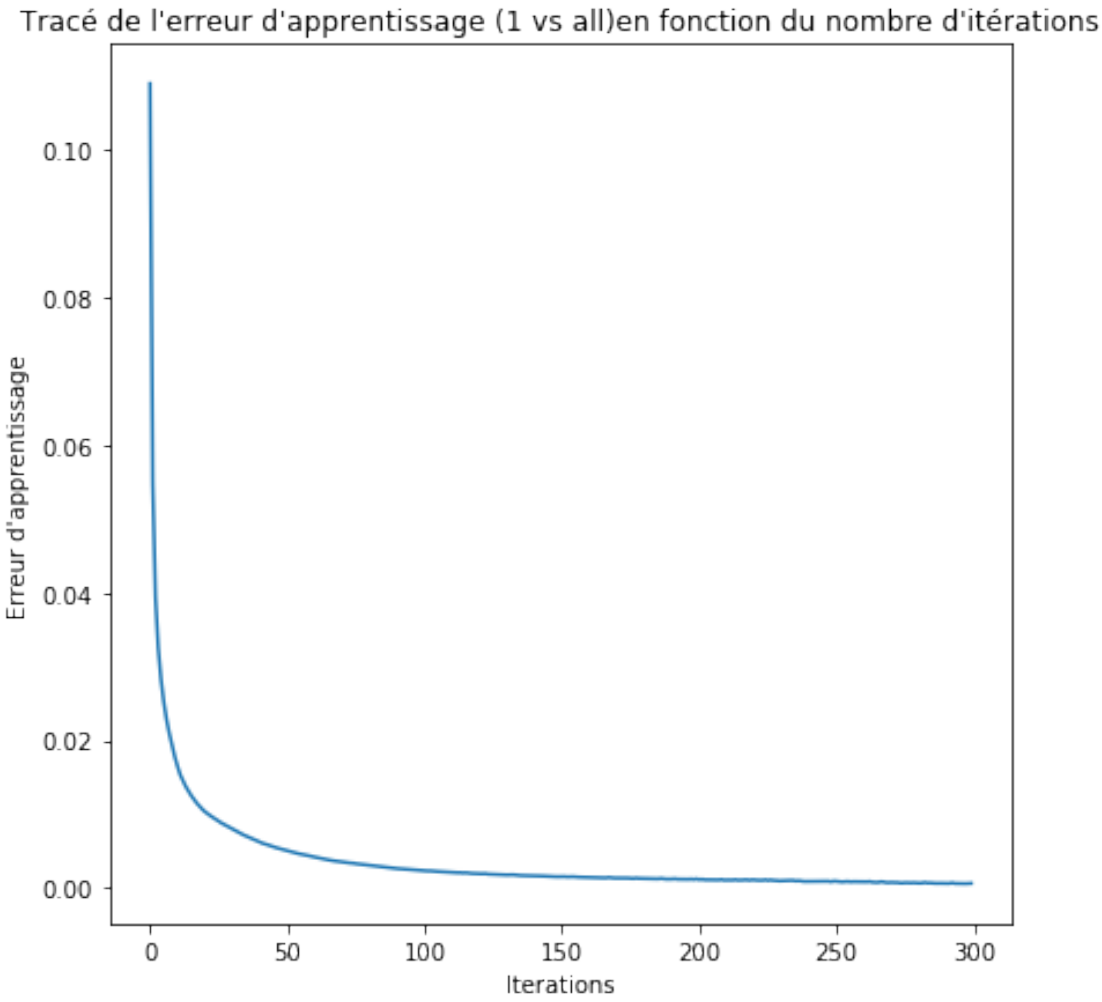
```
In [26]: #Test des autres configurations
         #1 chiffre contre 1 autre :  6 contre 9
         # 6 vs 9

         #élaboration de fonctions qui permettront de calculer l'erreur de test plus tard


         def train6vs9(val):
             if (val == 6):
                 return 1
             else:
                 return 0
```

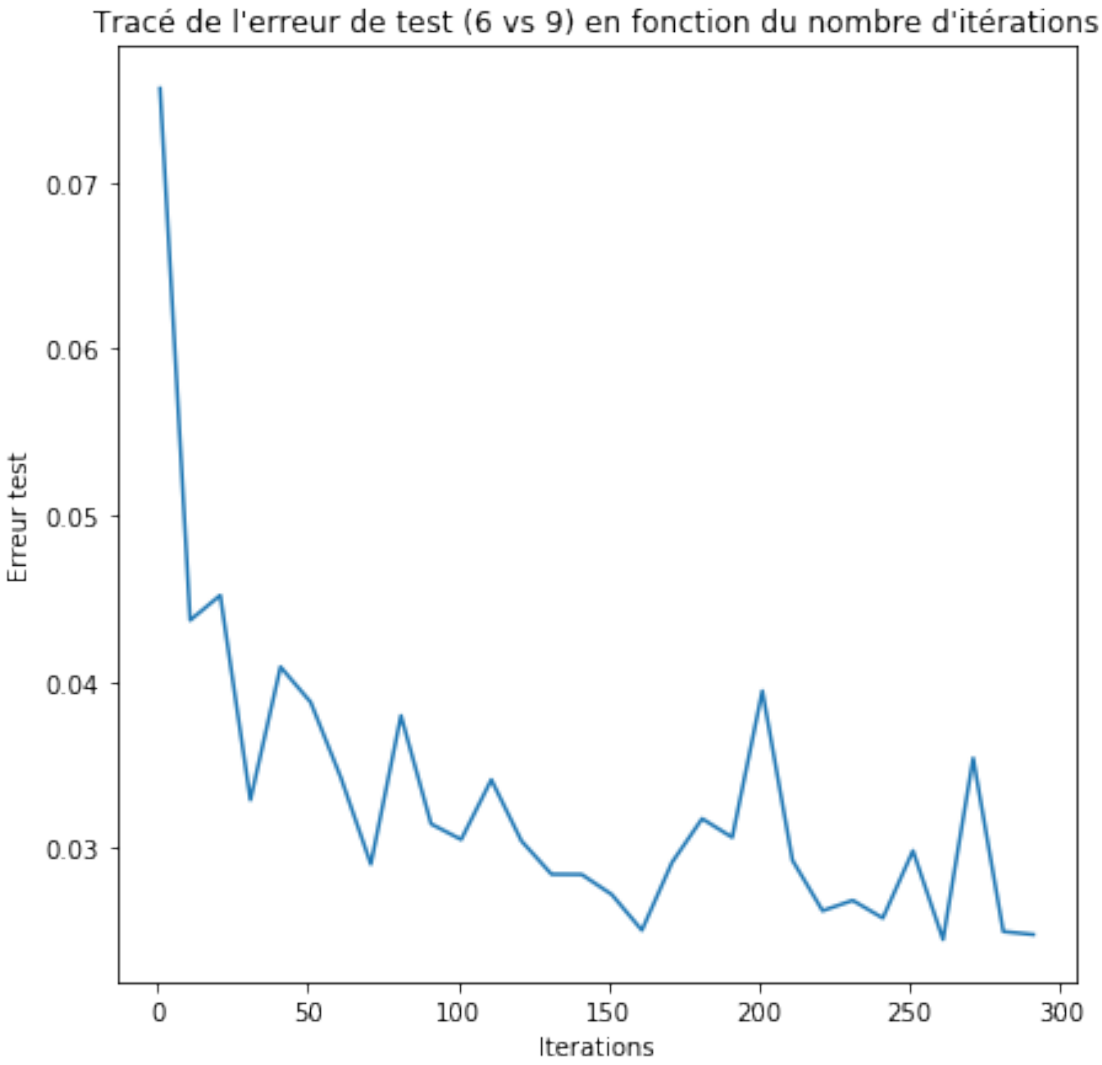
```
train6vs9 = np.vectorize(train6vs9)
xtrain,ytrain = get_usps([6,9],xuspstrain,yuspstrain)
xtest,ytest = get_usps([6,9],xuspstest,yuspstest)
ytrain=train6vs9(ytrain)
ytest=train6vs9(ytest)

model=Perceptron(max_iter=300, eps=1e-3)
model.fit(xtrain, ytrain)

plt.figure(figsize=(7,7))
plt.plot(model.histo_f)
plt.xlabel('Iterations')
plt.ylabel("Erreur d'apprentissage")
plt.title("Tracé de l'erreur d'apprentissage (6 vs 9)en fonction du nombre d'itérations
plt.show()
# Erreur de test
def testError6vs9(nb_iter):
    plt.figure(figsize=(7,7))
    model=Perceptron(max_iter=1, eps=1e-3)
    errorTest = []
    for k in range(1, nb_iter, 10):
        xtrain,ytrain = get_usps([6,9],xuspstrain,yuspstrain)
        xtest,ytest = get_usps([6,9],xuspstest,yuspstest)
        ytrain=train6vs9(ytrain)
        ytest=train6vs9(ytest)
        model.max_iter = k
        model.fit(xtrain, ytrain)
        errorTest.append(hinge(model.w, xtest, ytest))
    plt.plot(range(1, nb_iter, 10), errorTest)
    plt.xlabel('Iterations')
    plt.ylabel('Erreur test')
    plt.title("Tracé de l'erreur de test (6 vs 9) en fonction du nombre d'itérations")
    plt.show()

testError6vs9(300)



#Test:un chiffre contre tous les autres
def train1contretous(val):
    if (val == 1):
        return 1
    else:
        return 0

train1contretous = np.vectorize(train1contretous)

xtrain,ytrain = get_usps([1,2,3,4,5,6,7,8,9],xuspstrain,yuspstrain)
```

```
xtest,ytest = get_usps([1,2,3,4,5,6,7,8,9],xuspstest,yuspstest)

ytrain=train1contretous(ytrain)
ytest=train1contretous(ytest)
model=Perceptron(max_iter=300, eps=1e-3)
model.fit(xtrain, ytrain)

#tracer de l'erreur d'apprentissage
plt.figure(figsize=(7,7))
plt.plot(model.histo_f)
plt.xlabel('Iterations')
plt.ylabel("Erreur d'apprentissage")
plt.title("Tracé de l'erreur d'apprentissage (1 vs all)en fonction du nombre d'itératio
plt.show()

# Erreur de test
def testError1vsall(nb_iter):
    plt.figure(figsize=(7,7))
    model=Perceptron(max_iter=1, eps=1e-3)
    errorTest = []
    for k in range(1, nb_iter, 10):
        xtrain,ytrain = get_usps([1,2,3,4,5,6,7,8,9],xuspstrain,yuspstrain)
        xtest,ytest = get_usps([1,2,3,4,5,6,7,8,9],xuspstest,yuspstest)
        ytrain=train1contretous(ytrain)
        ytest=train1contretous(ytest)
        model.max_iter = k
        model.fit(xtrain, ytrain)
        errorTest.append(hinge(model.w, xtest, ytest))
    plt.plot(range(1, nb_iter, 10), errorTest)
    plt.xlabel('Iterations')
    plt.ylabel('Erreur test')
    plt.title("Tracé de l'erreur de test (6 vs 9) en fonction du nombre d'itérations")
    plt.show()

testError1vsall(300)
```


Tracé de l'erreur d'apprentissage (6 vs 9)en fonction du nombre d'itérations
Erreur d'apprentissage
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
50
100
150
200
250
300
Iterations


Tracé de l'erreur de test (6 vs 9) en fonction du nombre d'itérations
Erreur test
0.040
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
0
50
100
150
200
250
300
Iterations


Tracé de l'erreur d'apprentissage (1 vs all)en fonction du nombre d'itérations
Erreur d'apprentissage
0.10
0.08
0.06
0.04
0.02
0.00
0
50
100
150
200
250
300
Iterations



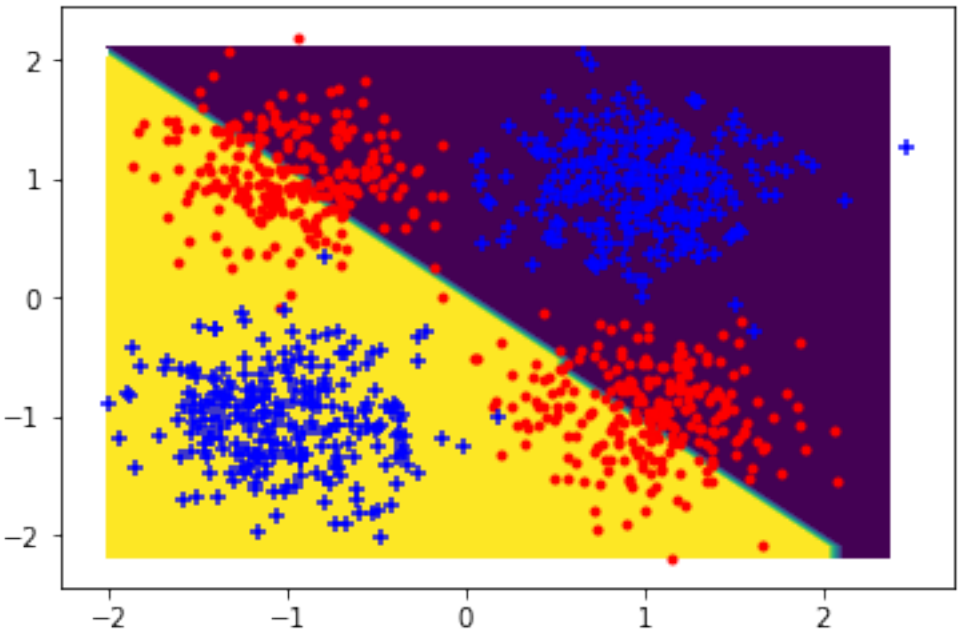
On observe pour tous les tests une courbe d'erreur d'apprentissage diminuant en fonction du nombre d'itérations, avec une erreur d'apprentissage quasi constante à partir de 50 itérations. Pour le chiffre 1 contre les autres on observe une courbe d'erreur de test globalement décroissante jusqu'à 150 itérations , puis une augmentation et des fluctuations jusqu'à 300 itérations, ce qui traduit du surapprentissage à partir de 150 itérations. Pour le chiffre 6 contre le chiffre 9 on observe des fluctuations importantes de l'erreur de test, bien qu'elle soit très faible. Le minimum de l'erreur d'apprentissage est atteint à 250 itérations pour le test chiffre 6 contre le chiffre 9.

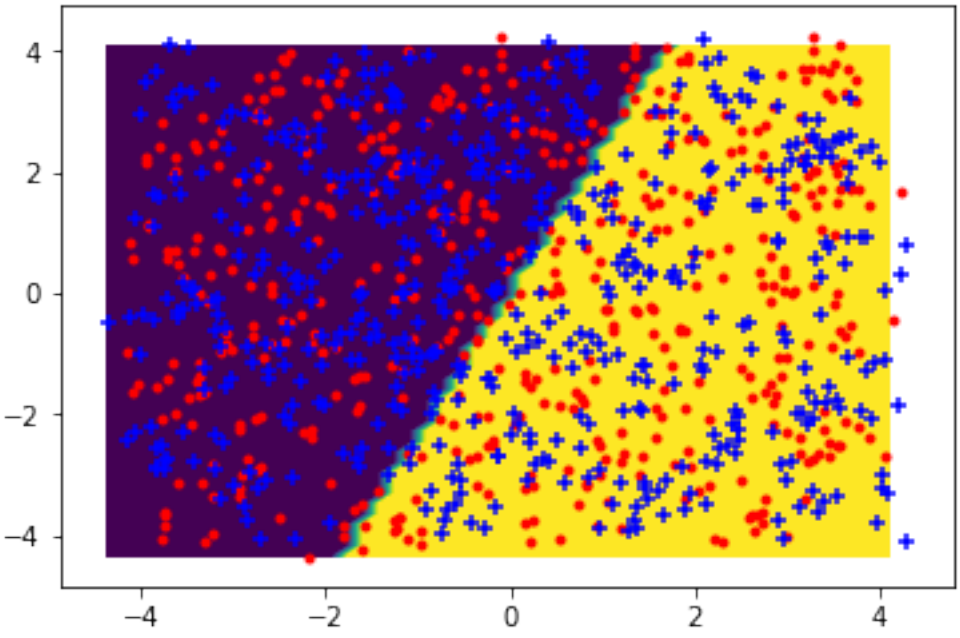
1.3 Expressivité et feature map

```
In [14]: # Question 3.1

         #test data de type 1
         xtrain,ytrain = gen_arti(data_type=1,epsilon=0.2)
         xtest,ytest = gen_arti(data_type=1,epsilon=0.2)
```

```
        plt.ion()
        model=Perceptron(max_iter=400, eps=1e-3)
        model.fit(xtrain,ytrain)
        print("score en train datatype 1: ",model.score(xtrain,ytrain))
        print("score en test datatype 1 : ",model.score(xtest,ytest))

        #### Tracer de frontiere
        plt.figure()
        plot_frontiere(xtrain, model.predict,50)
        plot_data(xtrain,ytrain)

        #test data de type 2
        xtrain,ytrain = gen_arti(data_type=2,epsilon=0.2)
        xtest,ytest = gen_arti(data_type=2,epsilon=0.2)

        plt.ion()
        model=Perceptron(max_iter=400, eps=1e-3)
        model.fit(xtrain,ytrain)
        print("score en train datatype 2: ",model.score(xtrain,ytrain))
        print("score en test datatype 2 : ",model.score(xtest,ytest))

        #### Tracer de frontiere
        plt.figure()
        plot_frontiere(xtrain, model.predict,50)
        plot_data(xtrain,ytrain)
```

```
score en train datatype 1:  0.521
score en test datatype 1 :  0.505
score en train datatype 2:  0.474
score en test datatype 2 :  0.515
```





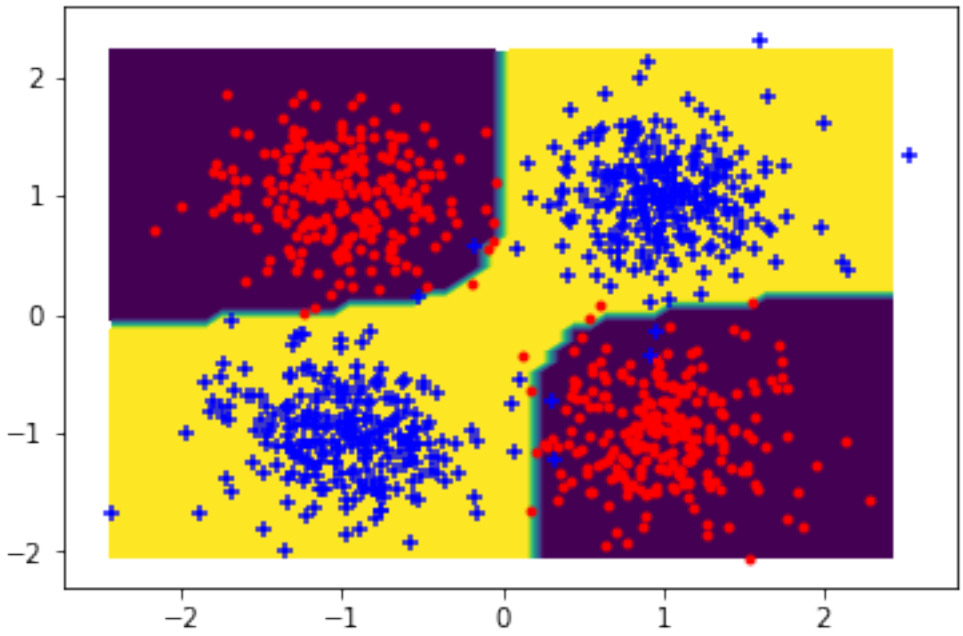
Comme mentionné plus haut lors de la section 1, les données de type 1 ou 2 ne peuvent être séparées par une frontière linéaire. Le score est de 50% aussi bien pour l'entraînement que pour le test, ce qui ne constitue pas un résultat satisfaisant pour notre Perceptron.

Pour remédier à ce problème, nous allons dans cette section, augmenté l'expressivité de notre classifieur en considérant des "feature map" ou encore des fonctions de plongement.

On considére un plongement polynomial des données. Les grands degrés du plongement polynomial pouvant résulté à du sur-apprentissage, nous nous limitons ici au cas bi-dimensionnel.

In [15]:
```
# Question 3.2

def poly(x):
    vect_poly = np.zeros((x.shape[0],6))
    vect_poly[:,0] = np.array([1]*x.shape[0])
    vect_poly[:,1] = x[:,0]
    vect_poly[:,2] = x[:,1]
    vect_poly[:,3] = x[:,0] * x[:,1]
    vect_poly[:,4] = x[:,0] * x[:,0]
    vect_poly[:,5] = x[:,1] * x[:,1]
    return vect_poly

xtrain,ytrain = gen_arti(data_type=1,epsilon=0.2)
xtest,ytest = gen_arti(data_type=1,epsilon=0.2)

model = Perceptron(max_iter=1000, eps=1e-3, projection=poly)
model.fit(xtrain, ytrain)

print("score en train : ",model.score(xtrain,ytrain))
print("score en test : ",model.score(xtest,ytest))

plt.figure()
plot_frontiere(xtrain, model.predict)
plot_data(xtrain,ytrain)
```

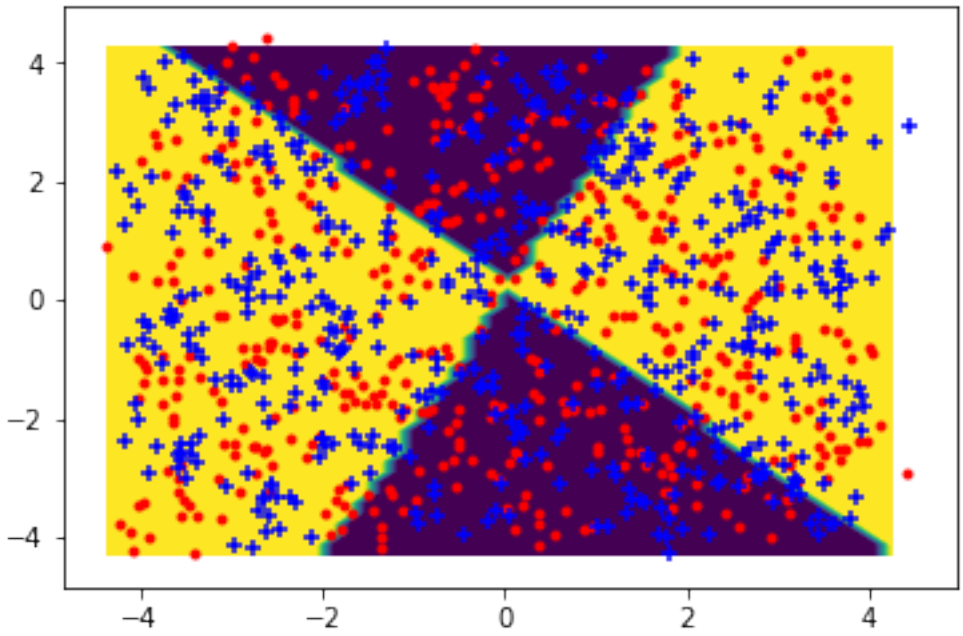
```
score en train :  0.989
score en test :  0.987
```



Les frontières de décision doivent correspondre, avec ce plongement polynomial, à des projetés polynomiaux sur l'espace des données. En effet, pour des jeux de données de type 1, chaque classe possède deux nuages de points. Avec la bi-dimensionnalité de ce modèle, des frontières polynomiales délimitent chacune des dimensions de l'un des classes offrant ainsi une séparation quasi-parfaite des données.

En effet, on obtient un score de test de 98.4% pour un entraînement de 99%.

```
In [16]: xtrain,ytrain = gen_arti(data_type=2,epsilon=0.2)
         xtest,ytest = gen_arti(data_type=2,epsilon=0.2)

         model = Perceptron(max_iter=1000, eps=1e-3, projection=poly)
         model.fit(xtrain, ytrain)

         print("score en train : ",model.score(xtrain,ytrain))
         print("score en test : ",model.score(xtest,ytest))

         plt.figure()
         plot_frontiere(xtrain, model.predict)
         plot_data(xtrain,ytrain)

score en train :  0.509
score en test :  0.522
```



Néanmoins, pour un jeu de donnée encore moins séparable (de type 2 par exemple), le plongement polynomial reste inefficace (51% d'apprentissage et 52% de test) car incapable de délimiter l'une des classes par des projetés polynomiaux. Les points sont beaucoup trop mélangés et nous avons besoin de faire un plongement dans un espace de dimension plus grande.

```
In [25]: # Question 3.3

         def k(x,xp,sigma):
             xp = xp.reshape((1, xp.shape[0]))
             return np.exp(-(np.linalg.norm((x-xp), axis=1)**2)/(2*sigma**2))



         #cas où les points sont tirés au hasard

         def gauss_B_1(data,data_train,Nbre_exemple,sigma):
             K=np.zeros((data.shape[0],Nbre_exemple))
             for i in range(Nbre_exemple):
                 K[:,i]=k(data,data_train[i,:],sigma)
             return K


         Nbre_exemple=300
         xtrain,ytrain = gen_arti(data_type=2,epsilon=0.1)
         xtest,ytest = gen_arti(data_type=2,epsilon=0.1)
         #generation de la matrice des points d'entrainement
```

```
B=np.random.rand(Nbre_exemple,xtrain.shape[1])
#creation de la fonction de projection


fonction_projection = lambda x: gauss_B_1(x,B,Nbre_exemple,0.4)


#test du modèle avec des points aléatoirement choisis
model = Perceptron(max_iter=2000, eps=1e-3, projection=fonction_projection)
model.fit(xtrain, ytrain)

print("score en train avec des points de description aléatoirement choisis: ",model.sco
print("score en test avec des points de description aléatoirement choisis: ",model.scor

plt.figure()
plot_frontiere(xtest, model.predict)
plot_data(xtest,ytest)




#Cas de points utilisés pour la description issus de l'ensemble d'entrainement

def gauss_B(data,data_train,liste_indice,Nbre_exemple,sigma):
    K=np.zeros((data.shape[0],Nbre_exemple))
    for i in range(Nbre_exemple):
        K[:,i]=k(data,data_train[liste_indice[i],:],sigma)
    return K


Nbre_exemple=300

xtrain,ytrain = gen_arti(data_type=2,epsilon=0.1)
xtest,ytest = gen_arti(data_type=2,epsilon=0.1)

#génération des
liste_indice=[]
for i in range(Nbre_exemple):
    indice=np.random.randint(0,xtrain.shape[0])
    securite=0
    while(indice in liste_indice and (securite<10000)):
        indice=np.random.randint(0,data.shape[0])
        securite+=1
    liste_indice.append(indice)


fonction_projection = lambda x: gauss_B(x,xtrain,liste_indice,Nbre_exemple,0.4)
```

```
        model = Perceptron(max_iter=2000, eps=1e-3, projection=fonction_projection)
        model.fit(xtrain, ytrain)

        print("score en train avec des points de description issus de l'ensemble d'entrainement
        print("score en test avec des points de description issus de l'ensemble d'entrainement:

        plt.figure()
        plot_frontiere(xtest, model.predict)
        plot_data(xtest,ytest)
```

```
score en train avec points aléatoirement choisis:  0.513
score en test avec points aléatoirement choisis:  0.517
score en train avec des points issus de l'ensemble d'entrainement :  0.919
score en test avec des points issus de l'ensemble d'entrainement:  0.79
```