```
In [1]: import numpy as np
        import matplotlib.pyplot as plot
```

Classification multi-classes

Mise en situation

Je suis travaille pour le bureau de poste. Je dois trouver un algorithme pour lire les chiffres des codes postaux.

Une régression logistique, c'est pour trouver une classe. Mais si j'en ai 10 ?

Classification sur N classes

Le principe

Jusque-là, la régression logistique nous a permis de classer les données en deux catégories : c'est un match, ou pas.

Pour plusieurs (= N) catégories, le principe est très simple : on va faire N modèles de régression logistique, et sortir le meilleur.

A partir du jeu de données, on entrainera donc N modèles, un pour la 1ere classe, un pour la 2nde, etc...

Gestion des probabilités

La régression logistique nous donne la probabilité du match sur la classe reconnue. Là je vais me retrouver avec N probabilités. Et à la question "à quelle classe appartient mon objet", je vais avoir N réponses, toutes probabilistes, et dont la somme ne fait pas 1. Exemple: cette photo est un chien à 40% ou un chat à 50% ou un poisson rouge à 70%. Ca fait un total de 160% :)

La technique brutale pour rédoudre ça consiste à dire : je prends le max, il vaut 1, et les autres 0. On aura "poisson rouge à 100%".

Et de manière un peu plus souple : on fait le ratio "probabilité de la classe c" / "somme des probabilités des N classes". Un produit en croix tout bête.

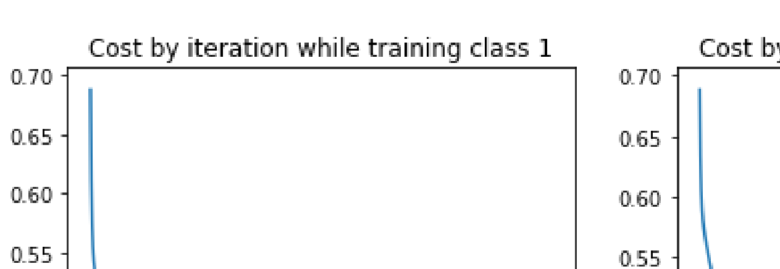
*Note : la technique **softmax** consiste à faire la même chose, mais sur les exponentielles des probabilités et pas les probabilités elles-mêmes. C'est surtout utilisé dans les dernières couches des réseaux de neurones, pas trop dans notre cas : vu que tous les N résultats sont des probabilités, et donc entre 0 et 1, ça va d'avantage réduire la probabilité max que la faire ressortir. Mais dans un réseau de neurones, cette couche de normalisation fait partie du modèle et donc prise en compte dans le training.*

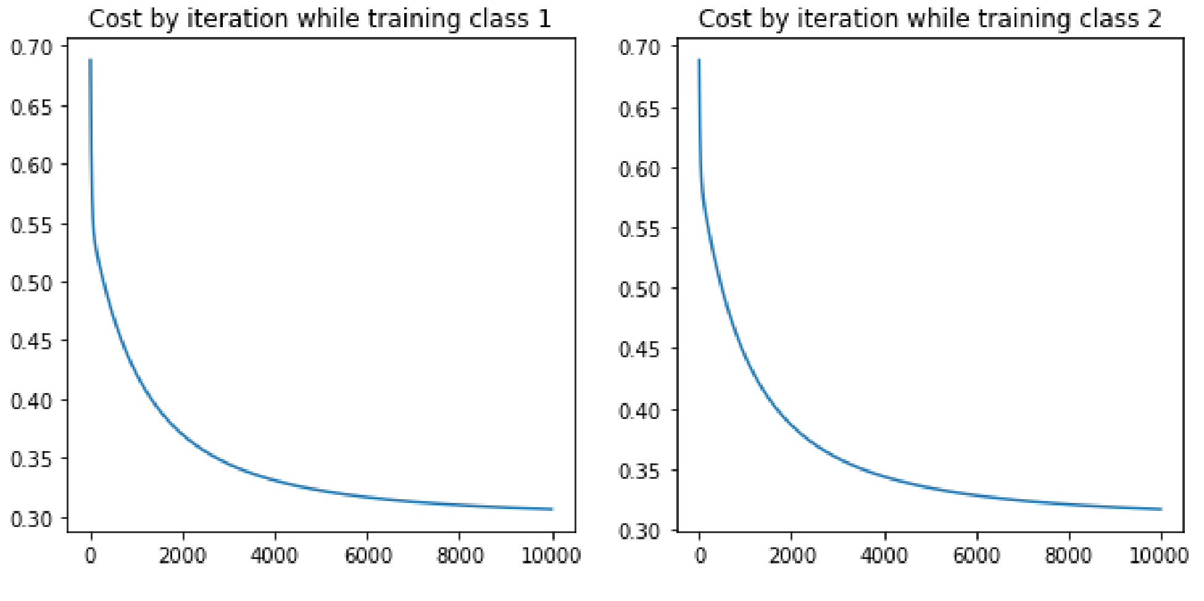
Simulation rapide

Validons rapidement le concept avec un exemple simple.

```
In [2]: def sigmoid(x): return 1 / (1+np.exp(-x))
        def f(x, theta): return sigmoid(np.dot(x, theta))

        def model_cost(x, y, theta):
            evals = f(x, theta)
            return -np.sum(y*np.log(evals) + (1-y)*np.log(1-evals)) / m

        def train_model_on_class(x, y, klass, alpha = 0.01, iterations = 1000):
            m, n = x.shape

            x = np.concatenate([np.ones((m, 1)), (x-mus)/sigmas], axis = 1)
            n += 1

            #valeurs {0,1} pour y : 1 si class == "klass", 0 sinon
            y = (y == klass)

            costs = np.zeros((iterations, 1))
            theta = np.random.rand(n, 1) * 0.01
            for i in range(iterations):
                errs = f(x, theta) - y
                grad = np.dot(x.T, errs) / m
                theta = theta - alpha * grad
                costs[i,0] = model_cost(x, y, theta)
            return theta, costs

        def classifier(x, thetas, mus, sigmas):
            m = x.shape[0]

            classes = len(thetas)
            probabilities = np.zeros((m, classes))

            for k in range(classes):
                x_k = np.concatenate([np.ones((m, 1)), (x-mus)/sigmas], axis = 1)
                probabilities[:, k] = f(x_k, thetas[k]).reshape(-1)
            return probabilities / np.sum(probabilities, axis = 1).reshape(-1,1)
```

```
In [3]: np.random.seed(1)

        # On va générer 1000 exemples, avec 50 erreurs (volontaires)
        m = 1000
        noisy_count = 50

        x_train = np.random.rand(m, 2) * 20 - 10
        y_train = np.zeros((m,1))
        y_train[np.where(np.all([x_train[:,0] <= x_train[:,1], x_train[:,0] >= -x_train[:,1]],
        y_train[np.where(np.all([x_train[:,0] <= x_train[:,1], x_train[:,0] < -x_train[:,1]],
        y_train[np.where(np.all([x_train[:,0] > x_train[:,1], x_train[:,0] >= -x_train[:,1]],
        y_train[m-noisy_count:, 0] = 3-y_train[m-noisy_count:, 0]
        #0 : partie basse
        #1 : partie haute
        #2 : partie gauche
        #3 : partie droite

        plot.scatter(x_train[:, 0], x_train[:,1], c=y_train)
        t = plot.title('Data visualization')

        #normalisation
        mus = np.mean(x_train, axis = 0)
        sigmas = np.std(x_train, axis = 0)
        x_train_norm = (x_train - mus)/sigmas

        thetas = []
        costs = []

        fig = plot.figure(figsize=(10,10))
        for klass in range(4):
            theta, cost = train_model_on_class(x_train_norm, y_train, klass, alpha=0.1, iterat
            thetas.append(theta)
            costs.append(cost)
            fig.add_subplot(2,2,klass + 1)
            plot.title('Cost by iteration while training class ' + str(klass+1))
            plot.plot(range(len(cost)), cost)

        plot.show()

        samples = np.array([[0,8],[8,0],[-8,0],[0,-8]])
        probabilities = classifier(samples, thetas, mus, sigmas)
        maxs = np.max(probabilities, axis = 1)
        target_classes = np.argmax(probabilities, axis = 1)
        for i in range(samples.shape[0]):
            print('Point (%d, %d) has been identified to be in class %s with probability %d%%
                samples[i, 0],
                samples[i, 1],
                ['bottom', 'top', 'left', 'right'][target_classes[i]],
                maxs[i]*100
            ))
```





```
Point (0, 8) has been identified to be in class top with probability 87%
Point (8, 0) has been identified to be in class right with probability 69%
Point (-8, 0) has been identified to be in class left with probability 70%
Point (0, -8) has been identified to be in class bottom with probability 68%
```

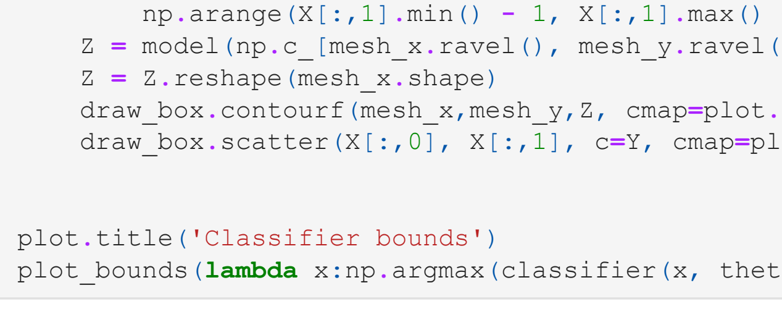
Magnifique! (même avec 5% de bruit dans les mesures)

On récapitule:

- 4 classes, 4 modèles, 4 trainings
- un "lisseur de probabilité" pour renvoyer la meilleure catégorie

Une petite visualisation des limites des zones détectées pour se faire plaisir :)

```
In [4]: def plot_bounds(model, X, Y, draw_box):
            h=0.01
            mesh_x, mesh_y = np.meshgrid(
                np.arange(X[:,0].min() - 1, X[:,0].max() + 1, h),
                np.arange(X[:,1].min() - 1, X[:,1].max() + 1, h))
            Z = model(np.c_[mesh_x.ravel(), mesh_y.ravel()])
            Z = Z.reshape(mesh_x.shape)
            draw_box.contourf(mesh_x,mesh_y,Z, cmap=plot.cm.Spectral)
            draw_box.scatter(X[:,0], X[:,1], c=Y, cmap=plot.cm.Spectral)


        plot.title('Classifier bounds')
        plot_bounds(lambda x:np.argmax(classifier(x, thetas, mus, sigmas), axis = 1), x_train,
```



Retour à la mise en situation

Chargement des données

Pour une fois, les données seront des vraies données. Elles proviennent du dataset de Yann Le Cun http://yann.lecun.com/exdb/mnist/

Il s'agit d'images 28x28 en noir et blanc, 60.000 données d'entrainement et 10.000 données de validation.

```
In [5]: def load(file):
            data = np.load(file)
            return data['x'], data['y']

        x_train, y_train = load('data/d09_train_data.npz')
        x_test , y_test  = load('data/d09_test_data.npz')

        print('%i training samples loaded of size %i' %(x_train.shape[0], x_train.shape[1]))
        print('%i test samples loaded' %(x_test.shape[0]))

        #normalisation
        #note: le 1e-9 est pour éviter des divisions par 0...
        mus = np.mean(x_train, axis = 0)
        sigmas = np.std(x_train, axis = 0) + 1e-9
        x_norm = (x_train-mus)/sigmas
```

```
60000 training samples loaded of size 784
10000 test samples loaded
```

On va garder toutes nos fonctions précédentes. On va simplement changer *train_model_on_class* par une version mini-batch, pour traiter les 60.000 exemples plus rapidement (plus lentement en fait, mais la taille mémoire nécessaire pour 60k exemples va faire du swap et c'est pas bon)

```
In [6]: def mbgd_train_model_on_class(x, y, klass, alpha = 0.01, epochs = 200, batch_size=128)
            m, n = x.shape

            x = np.concatenate([np.ones((m, 1)), (x-mus)/sigmas], axis = 1)
            n += 1

            #valeurs {0,1} pour y : 1 si class == "klass", 0 sinon
            y = (y == klass)

            theta = np.random.rand(n, 1) * 0.01
            for e in range(epochs):
                for start in range(0, m, batch_size):
                    x_batch = x[start:start+batch_size, :]
                    y_batch = y[start:start+batch_size, :]
                    errs = f(x_batch, theta) - y_batch
                    grad = np.dot(x_batch.T, errs) / m
                    theta = theta - alpha * grad
            return theta
```

Y'a plus qu'à entrainer le modèle, et à tester !

L'entrainement va prendre quelques minutes, mais enfin c'est pas énorme, et on pourra toujours sauver notre modèle pour l'utiliser : l'entrainement, on ne le fait qu'une seule fois !

```
In [7]: thetas = []
        for klass in range(10):
            print('Training class %i...'%klass)
            thetas.append(mbgd_train_model_on_class(x_norm, y_train, klass, alpha=0.05))
```

```
Training class 0...
Training class 1...
Training class 2...
Training class 3...
Training class 4...
Training class 5...
Training class 6...
Training class 7...
Training class 8...
Training class 9...
```

Le test maintenant. Lui, c'est beaucoup plus rapide : y'a qu'une seule passe

```
In [8]: #On calcule les résultats sur le jeu de training
        train_results = np.argmax(classifier(x_train, thetas, mus, sigmas), axis = 1).reshape
        print('Accuracy on train set : %f' % (np.sum(train_results == y_train)/y_train.shape[(

        #On calcule les résultats sur le jeu de test
        test_results = np.argmax(classifier(x_test, thetas, mus, sigmas), axis = 1).reshape(-1
        print('Accuracy on test set : %f' % (np.sum(test_results == y_test)/y_test.shape[0]))
```

```
Accuracy on train set : 0.722550
Accuracy on test set : 0.733700
```

On arrive donc à reconnaitre des chiffres manuscrits sur des images jamais vues à l'entrainement, avec une fiabilité de 73.37%, et même mieux que sur le jeu de training (72.26%), on n'a probablement pas de souci d'overfit - en tout cas la vairance est très faible. Y'a par contre un sacré biais...

Mais quoi qu'il en soit c'est vraiment pas si mal en fait, pour un modèle aussi simple que la régression logistique. Avec des modèles plus complexes, à base de réseaux de neurones, on verra qu'on fera mieux, mais c'est plus gros à mettre en place...

Une autre différence : avec un réseau de neurones, on pourra faire un seul modèle à 10 sorties, plutôt que 10 modèles à sortie unique

L'avantage, là, c'est que clean et concis :) Le coeur du code tient en 3 lignes:

- error = sigmoid($x.\theta$) - y
- grad = x^T.error / m
- θ = θ - α.grad

Et on boucle...

Analyse des erreurs

On peut pousser l'analyse un peu plus loin : quels sont les chiffres les moins bien reconnus ?

```
In [9]: for i in range(10):
            invalid_class_i = np.sum(np.all([y_test == i, test_results != i], axis = 0))
            total_class_i = np.sum(y_test == i)
            print('Failure rate for class %i: %d%%' %(i,invalid_class_i * 100 / total_class_i)
```

```
Failure rate for class 0: 8%
Failure rate for class 1: 2%
Failure rate for class 2: 32%
Failure rate for class 3: 21%
Failure rate for class 4: 41%
Failure rate for class 5: 48%
Failure rate for class 6: 12%
Failure rate for class 7: 13%
Failure rate for class 8: 55%
Failure rate for class 9: 35%
```

Y'a clairement un souci avec les '8' :D

Et l'autre sens : quels sont les chiffres reconnus par erreur le plus souvent ?

```
In [10]: for i in range(10):
             invalid_class_i = np.sum(np.all([y_test != i, test_results == i], axis = 0))
             total_class_i = np.sum(y_test == i)
             print('Failure rate for class %i: %d%%' %(i,invalid_class_i * 100 / total_class_i)
```

```
Failure rate for class 0: 34%
Failure rate for class 1: 51%
Failure rate for class 2: 12%
Failure rate for class 3: 34%
Failure rate for class 4: 8%
Failure rate for class 5: 15%
Failure rate for class 6: 23%
Failure rate for class 7: 37%
Failure rate for class 8: 14%
Failure rate for class 9: 28%
```

Là c'est plutôt les '1' qui couinent.

Pour résumer, mon modèle marche pas trop mal, mais il a la manie de reconnaitre les 1 là où y'en n'a pas, et de rater les 8.

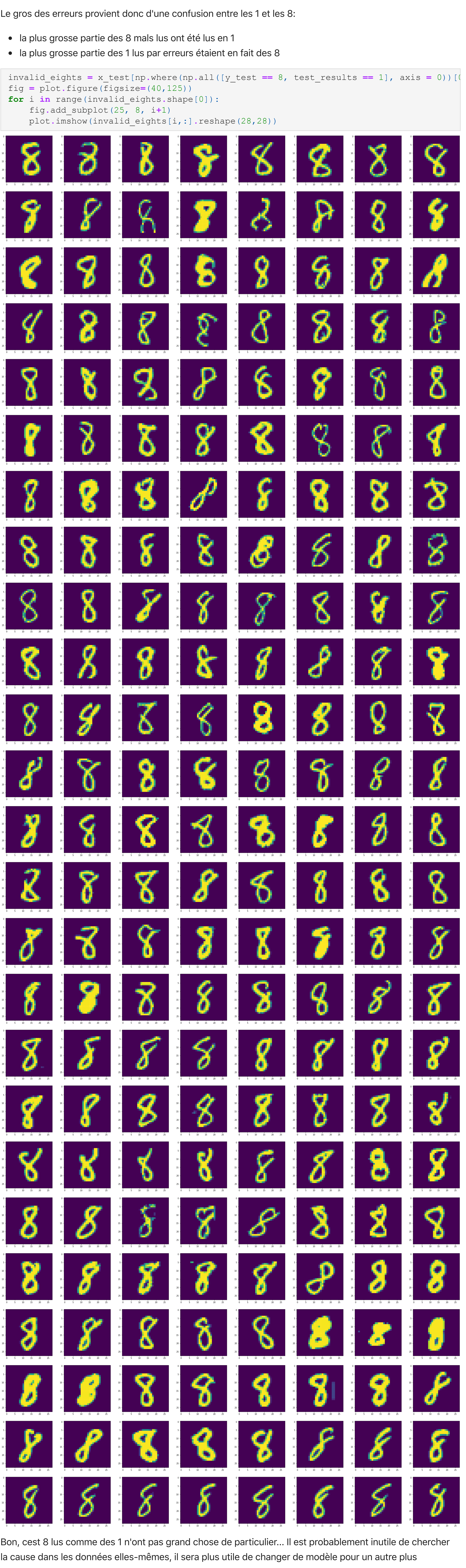
```
In [11]: for i in range(10): print('8 is read as %i in %i cases' %(i, np.sum(test_results[y_tes
         print()
         for i in range(10): print('%i is read as 1 in %i cases' %(i, np.sum(test_results[y_tes
```

```
8 is read as 0 in 50 cases
8 is read as 1 in 200 cases
8 is read as 2 in 19 cases
8 is read as 3 in 84 cases
8 is read as 4 in 20 cases
8 is read as 5 in 83 cases
8 is read as 6 in 14 cases
8 is read as 7 in 37 cases
8 is read as 8 in 431 cases
8 is read as 9 in 36 cases

0 is read as 1 in 2 cases
1 is read as 1 in 1110 cases
2 is read as 1 in 68 cases
3 is read as 1 in 59 cases
4 is read as 1 in 98 cases
5 is read as 1 in 32 cases
6 is read as 1 in 21 cases
7 is read as 1 in 65 cases
8 is read as 1 in 200 cases
9 is read as 1 in 43 cases
```

Le gros des erreurs provient donc d'une confusion entre les 1 et les 8:

- la plus grosse partie des 8 mals lus ont été lus en 1
- la plus grosse partie des 1 lus par erreurs étaient en fait des 8

```
In [12]: invalid_eights = x_test[np.where(np.all([y_test == 8, test_results == 1], axis = 0))[(
         fig = plot.figure(figsize=(40,125))
         for i in range(invalid_eights.shape[0]):
             fig.add_subplot(25, 8, i+1)
             plot.imshow(invalid_eights[i,:].reshape(28,28))
```



Bon, cest 8 lus comme des 1 n'ont pas grand chose de particulier... Il est probablement inutile de chercher la cause dans les données elles-mêmes, il sera plus utile de changer de modèle pour un autre plus complexe. (On n'a quasiment pas de variance, on peut se le permettre...), mais enfin ça sera pour un autre jour !