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

Test, biais et variance, régularisation - partie 2

Mise en situation (rappel)

J'ai implémenté un modèle, mais j'aimerais savoir s'il est efficace. Le coût est bien faible, c'est super !

Mais quand je l'utilise, c'est pas tout à fait ça :(

Régularisation

Le principe

Que se passe t'il quand on part en l'overfit ? Mathématiquement parlant, θ prend des valeurs un peu étranges pour pouvoir coller au mieux au training.

L'idée de la régularisation est de limiter l'overfit en général - il y a plusieurs méthodes de régularisation. On va en considérer une, assez simple : L2.

La pratique

Au lieu d'avoir, pour une régression linéaire,

$J(\theta) = \frac{1}{2m}\sum(x^{(i)}.\theta - y^i)^2$

on aura maintenant, pour une régularisation L2

$J(\theta) = \frac{1}{2m}\sum(x^{(i)}.\theta - y^i)^2 + \frac{\lambda}{2m}\sum_{j>0}\theta_j^2$

Notons le nouveau λ, on en reparlera

Pour une régularisation L2, on ajoute en gros le carré de la norme L2 (euclidienne, quoi), à un détail près : on ne compte pas θ_0.

En effet, pour tous les tests, on a $x_0 = 1$ (qui permet de prendre en compte une constante dans nos calculs). Contrairement aux autres indices, x_0 est fixé. Contraindre θ_0 en plus créera un trop gros biais.

Pour mieux saisir cette idée, supposons que tous nos test soient bon, à une constante additive près. Par exemple, mon modèle renvoie 5, 8, 12 au lieu de 9, 12, 16 : mon modèle a un biais de 4, mais on peut facilement obtenir le modèle parfait en ajoutant 4 à θ_0.

Par contre, pour tous les autres paramètres, on demande à ce qu'ils soient aussi petits que possible, pour minimiser la somme des carrés.

Pour une régression logistique, pareil on aura : $J_{reg}(\theta) = J_{classique}(\theta) + \frac{\lambda}{2m}\sum\theta_j^2$.

Les gradients

Bien entendu, les gradients changent aussi, quand on essaye de minimiser la fonction.

Le terme supplémentaire est $\frac{\lambda}{2m}\sum_{j>0}\theta_j^2$, sa dérivée par rapport à $\partial\theta_j$ se calcule facilement : tous les termes de la somme sont constants (donc dérivée nulle), à l'exception de θ_j^2 dont la dérivée est $2\theta_j$.

La dérivé du terme supplémentaire est donc $\frac{\lambda}{m}\sum\theta_j$

Le test

Lorsqu'on mesure le coût sur un jeu de test, il faut le faire **sans** prendre en compte la régularisation.

Ce terme en plus n'a qu'un seul objectif : "orienter" la convergence vers le sens qui m'intéresse. Si maintenant j'obtiens, avec ce terme, un modèle parfait, qui répond toujours correctement sur des données de tests, son coût doit clairement être 0, par principe (puisqu'on ne peut pas faire mieux), c'est bien plus exact que de dire qu'il est égal au terme de régularisation.

Autrement, ça voudrait dire qu'entre un modèle parfait en test, et un modèle presque parfait avec des plus petits θ_j, on préfère le second :(

λ : ce mystérieux paramètre

Reste ce mystérieux λ: c'est quoi ? On récapitule :

- Ca ne fait pas partie du modèle (on élimine le terme de régularisation lors du test et lors de l'utilisation...)
- Ca aide à faire converger le modèle.

Par définition (cf un peu en arrière dans le temps), ce n'est pas un paramètre du modèle, mais un nouvel hyperparamètre, tout simplement.

- Si λ vaut 0 : on ne régularise pas le modèle
- Plus λ augmente, plus l'impact de la régularisation sur le coût total est important

On se sert donc de ce λ pour jouer avec l'effet de la régularisation.

Jeux de train, dev et test : comment trouver le bon λ

- J'ai mon modèle, je l'entraine sur un jeu de train pour une valeur de λ, et je teste sur mon jeu de test - je mesure la performance de mon modèle.
- J'essaye une autre valeur de λ, et je reteste, etc...

Et au final, je trouve le "meilleur" λ. Est-ce bien ?

La réponse est "pas top". Ce que j'ai fait avec λ et le jeu de test est équivalent à ce que je fais avec θ et le jeu de train : je trouve la mailleure valeur. Et puisque θ à besoin d'être testé sur un jeu inconnu, λ devrait aussi être testé sur un autre jeu que test, inconnu aussi.

Un autre découpage des données répandu est :

- training: 60%
- dev : 20 %
- test : 20%

Le nouveau jeu de dev (ou de cross-validation) permet de regarder les impacts des hyperparamètres. Au final:

- Le modèle est entrainé avec certaines valeurs d'hyperparamètres, sur training
- La performance du modèle "hyperparamétré" est mesurée par le jeu de dev
- On fait varier les hyperparamètres et on recommence
- On trouve le "meilleur modèle hyperparamétré" et on mesure sa performance sur test

Concernant la dernière étape : on le fait sur test et pas sur dev, puisqu'on sait déjà par construction que la performance sur dev est bonne (la meilleure possible même) - de la même manière qu'on ne teste pas sur training puisque la perfomance sur training est par construction la meilleure possible.

Retour à la mise en situation

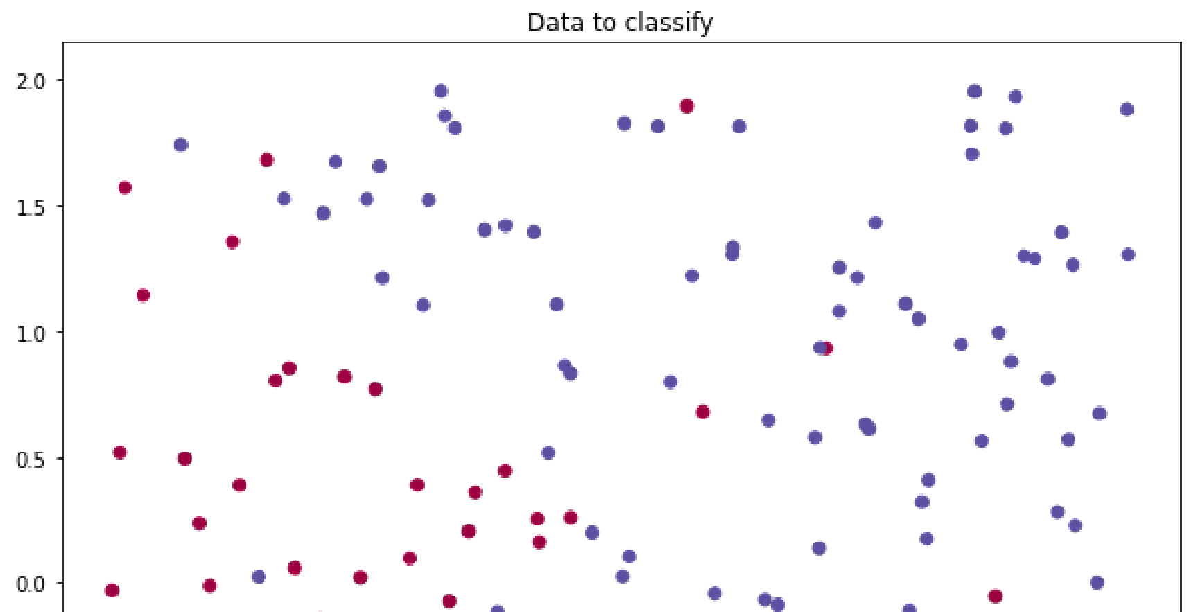
Je dois classifier des données. Plus mon modèle est complexe, et plus j'arrive à réduire le coût pendant l'entrainement. Mais ma fonction ne fait pas forcément mieux :(

```
In [2]: #nos outis

        # Transforme mes données avec des colonnes de degrés supplémentaires
        def pre_process_data(X, degree):
            m = X.shape[0]
            res = np.ones((m, 1))
            for i in range(1, degree+1):
                for j in range(i+1):
                    res = np.concatenate([res, (X[:,0]**j * X[:,1]**(i-j)).reshape(m, 1)], ax
            return res

        # Fonction sigmoide
        def sigmoid(x) : return 1/(1+np.exp(-x))

        # Fonction de prédiction
        def f(X, theta):
            return sigmoid(np.dot(X, theta))

        # Fonction de coût - sans régularisation
        def cost(X, Y, theta):
            predictions = f(X, theta)
            return -np.sum(Y * np.log(predictions) + (1-Y) * np.log(1-predictions)) / X.shape

        # Descente de gradient
        def regression_logistique(X, Y, degree, alpha = 0.001, lambd = 0, iterations = 1000):
            X = pre_process_data(X, degree)
            m, n = X.shape

            mus = np.mean(X, axis = 0)
            sigmas = np.std(X, axis = 0)

            #On ne normalise pas la colonne x0
            mus[0] = 0
            sigmas[0] = 1

            X = (X-mus) / (sigmas)

            theta = np.ones((n, 1))
            costs = []
            for i in range(iterations):
                errors = f(X, theta) - Y
                grad_no_regul = np.dot(X.T, errors)
                regul = lambd * (np.sum(theta) - theta[0])
                grad = (grad_no_regul + regul) / m
                theta = theta - alpha * grad
                costs.append(cost(X, Y, theta))
            return theta, costs, mus, sigmas

        # Visualisation de la classification
        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)
```

Chargement des données

Chargeons les données, et visualisons les rapidement

```
In [3]: data = np.load('data/d08_data.npy')
        Xtrain = data[:, 0:-1]
        Ytrain = data[:, -1].reshape(-1, 1)

        fig = plot.figure(figsize=(10,10))
        plot.scatter(Xtrain[:,0], Xtrain[:,1], c=Ytrain, cmap=plot.cm.Spectral)
        t = plot.title('Data to classify')
```

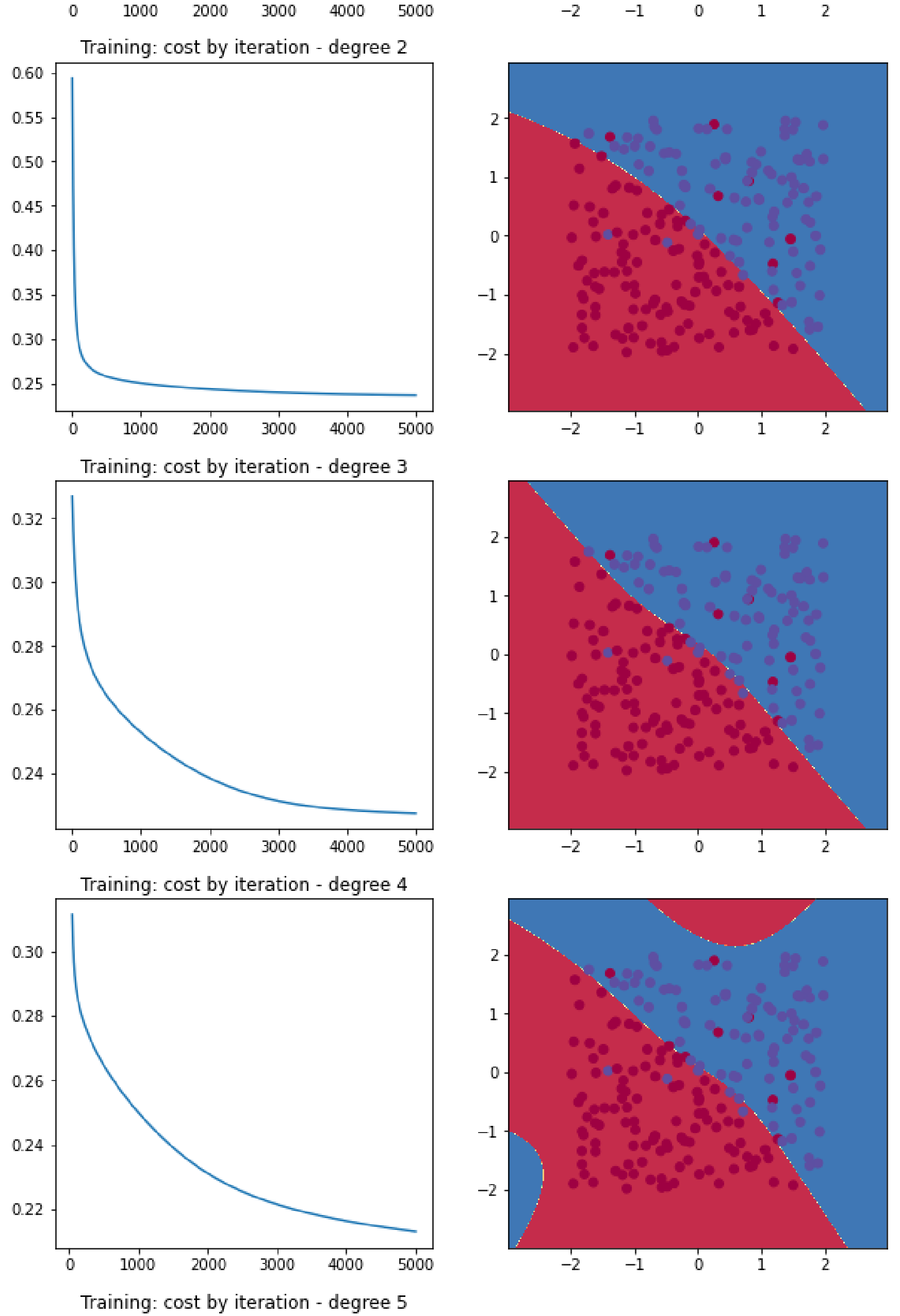


A quelques erreurs près, une diagonale suffirait à bien classer nos données !

Entraînons le modèle :

```
In [4]: thetas = []
        final_costs = []
        max_degree = 5
        fig = plot.figure(figsize=(10,5*max_degree))

        for degree in range(1, max_degree + 1):
            theta, costs, mus, sigmas = regression_logistique(Xtrain, Ytrain, degree, alpha =
            fig.add_subplot(max_degree,2, degree*2 - 1)
            plot.plot(range(5000), costs)
            plot.title('Training: cost by iteration - degree ' + str(degree))
            ax = fig.add_subplot(max_degree,2, degree * 2)
            plot_bounds(lambda x:np.round(f((pre_process_data(x, degree) - mus)/(sigmas), thet
            thetas.append(theta)
            final_costs.append(costs[-1])
```

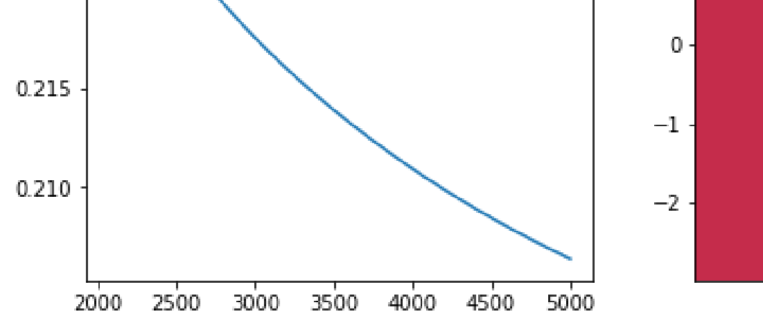
```
<ipython-input-2-9ea1a79099e4>:22: RuntimeWarning: divide by zero encountered in log
  return -np.sum(Y * np.log(predictions) + (1-Y) * np.log(1-predictions)) / X.shape[0]
<ipython-input-2-9ea1a79099e4>:22: RuntimeWarning: invalid value encountered in multip
ly
  return -np.sum(Y * np.log(predictions) + (1-Y) * np.log(1-predictions)) / X.shape[0]
```



Plus le degré augmente, plus la séparation devient... exotique. Mais le modèle essaye de faire au mieux pour inclure les différents points "invalides" (le petit point rouge en haut va se faire "avaler" par la zone rouge qui descend par exemple).

Pourtant, le coût est de plus en plus intéressant :

```
In [5]: p = plot.plot(range(1,max_degree+1), final_costs, "b*")
```



Et malgré celà, si on prend un point comme (0, 3), qui est clairement "bleu" en théorie :

```
In [6]: for d in range(max_degree):
            print("Degree %i, probability of being blue: %f" %(d+1, 100*f(pre_process_data(np
```

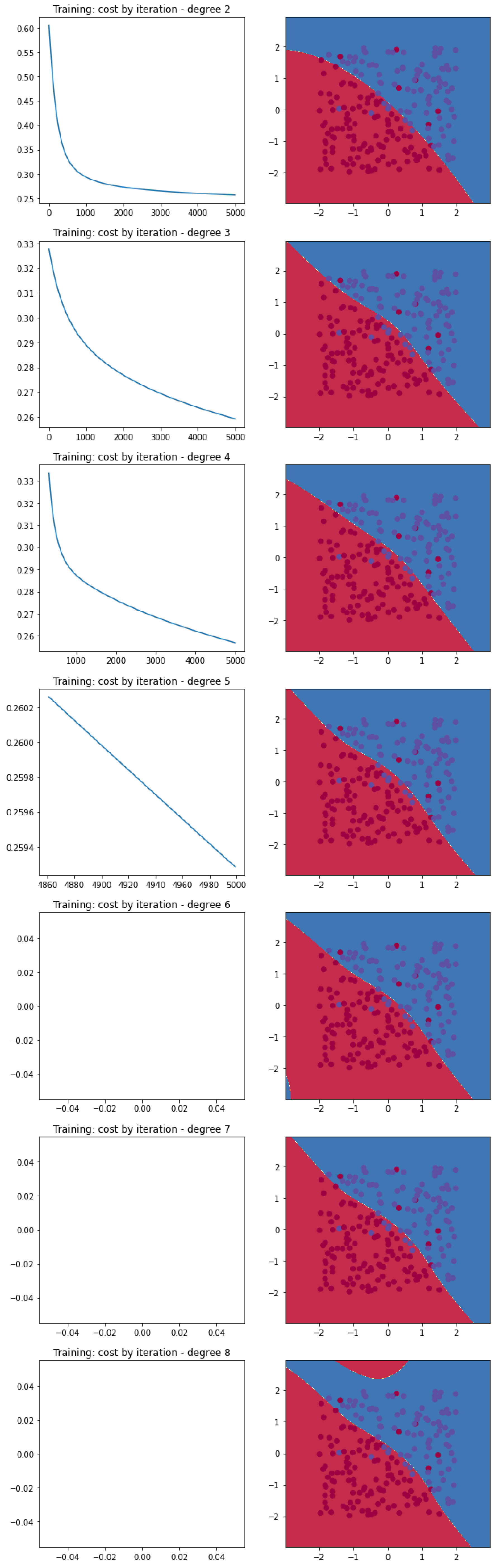
```
Degree 1, probability of being blue: 99.969360
Degree 2, probability of being blue: 75.410825
Degree 3, probability of being blue: 100.000000
Degree 4, probability of being blue: 0.000000
Degree 5, probability of being blue: 0.000000
```

Les degrés 4 et plus sont bien meilleurs à l'entrainement, mais ils sont hyper formels : (0,3) est rouge... alors qu'il est bleu :P

On va maintenant tout refaire, avec un peu de "lambda" pour faire de la régularisation

```
In [7]: thetas = []
        final_costs = []
        max_degree = 8

        fig = plot.figure(figsize=(10,5*max_degree))

        for degree in range(1, max_degree + 1):
            theta, costs, mus, sigmas = regression_logistique(
                Xtrain, Ytrain, degree, alpha = 0.01, lambd = .1, iterations = 5000)
            fig.add_subplot(max_degree,2, degree*2 - 1)
            plot.plot(range(5000), costs)
            plot.title('Training: cost by iteration - degree ' + str(degree))
            ax = fig.add_subplot(max_degree,2, degree * 2)
            plot_bounds(lambda x:np.round(f((pre_process_data(x, degree) - mus)/(sigmas), thet
            thetas.append(theta)
            final_costs.append(costs[-1])
```

```
<ipython-input-2-9ea1a79099e4>:22: RuntimeWarning: divide by zero encountered in log
  return -np.sum(Y * np.log(predictions) + (1-Y) * np.log(1-predictions)) / X.shape[0]
<ipython-input-2-9ea1a79099e4>:22: RuntimeWarning: invalid value encountered in multip
ly
  return -np.sum(Y * np.log(predictions) + (1-Y) * np.log(1-predictions)) / X.shape[0]
```



Idéalement, il faut faire tester plusieurs λ, et différement suivant les degrés. Mais on voit bien l'impact de la régularisation : on essaye d'aplatir les séparations.

Note: On remarque quand même qu'au degré 8, la régularisation ne suffit plus : le modèle a trop de libertés, il faut augmenter λ probablement.

Si on affiche les thetas, on voit bien que la priorité est donnée aux 2 premiers termes (les seuls qui servent vraiment)

```
In [8]: for i in range(max_degree):
            print("d=" + str(i+1) + ":" + np.str(np.round(thetas[i].T, 1)))
```

```
d=1:[[-0.5  2.3  2.2]]
d=2:[[-0.3  2.3  2.3  0.   0.1  0.4]]
d=3:[[0.1 1.5 1.3 0.2 0.5 0.7 1.2 0.5 0.5 1.4]]
d=4:[[ 0.3  1.5  1.3 -0.   0.2  0.3  1.3  0.7  0.6  1.5 -0.1  0.3  0.6  0.5
   0.4]]
d=5:[[ 0.5  1.3  1.1 -0.   0.2  0.3  0.9  0.3  0.5  1.  -0.   0.4  0.7  0.6
   0.5  0.6  0.2  0.7  0.9  0.3  0.9]]
d=6:[[ 0.6  1.3  1.  -0.1  0.   0.2  0.9  0.4  0.5  1.  -0.1  0.1  0.5  0.4
   0.3  0.6  0.3  0.7  0.9  0.3  1.  -0.2  0.2  0.5  0.4  0.5  0.4  0.3]]
d=7:[[ 0.6  1.2  1.  -0.1  0.   0.2  0.7  0.2  0.3  0.8 -0.1  0.1  0.5  0.4
   0.3  0.4  0.1  0.5  0.7  0.2  0.7 -0.2  0.2  0.5  0.4  0.5  0.4  0.4
   0.2  0.1  0.6  0.5  0.4  0.7  0.2  0.7]]
d=8:[[ 0.5  1.2  1.  -0.1 -0.   0.1  0.8  0.2  0.3  0.8 -0.2 -0.   0.3  0.3
   0.2  0.4  0.1  0.6  0.7  0.2  0.7 -0.3  0.   0.3  0.2  0.3  0.3  0.2
   0.2  0.1  0.6  0.5  0.4  0.7  0.2  0.7 -0.3  0.1  0.3  0.2  0.4  0.2
   0.4  0.3  0.3]]
```

Si on retrace les termes correspondants, on voit que les termes en x et y sont les plus importants pour les 2 premiers modèles, ainsi que les termes en x^3 et y^3 pour les termes supérieurs, et plus généralement les termes en puissance impaire.

Explications :

- tous les termes en puissance paire sont inutiles et donc éliminés
- la plupart des termes en produits croisés x et y pèsent un peu plus dans la balance - même s'ils auraient du être éliminés
- les termes en puissance impaire sont gardés, bien que seuls les premiers soient utiles : simplement le jeu de données était assez proche de 1 (en x et en y), le modèle n'a pas pu voir que seule la première puissance compte !

Regularisation et équation normale

En régression logistique, on avait le gradient :

[illegible]