TP : régression

- À rendre : un rapport aux format **pdf** ainsi que les 5 fichiers complétés `exoGD.py`, `exoLS.py`, `exoRLS.py`, `exoRLS2.py` et `exoLWLS.py`.
- Tout est à envoyer à l'adresse suivante : `perrin.isir@gmail.com`. N'oubliez pas d'inscrire vos noms quelque part (dans l'email ou le rapport).
- Le guide de référence de numpy : http://docs.scipy.org/doc/numpy/reference/

Introduction

L'objectif de la régression est de créer un modèle à partir de données observées. Elle s'effectue en ajustant les paramètres du modèle dont la structure est établie à l'avance. En apprentissage automatique, c'est une technique cruciale car l'ajustement des modèles utilisés permet une amélioration des performances au cours du temps.

La plupart du temps l'objectif est de modéliser des corrélations, ce qui permet notamment d'effectuer des prédictions en complétant des données nouvelles partielles. **Exemple :** on laisse tomber plusieurs fois un objet sur le sol, en mesurant à chaque fois la hauteur de chute et la vitesse lors de l'impact. Ces mesures constituent les données observées, et le but d'une régression pourra être de créer un modèle permettant de prédire la hauteur de chute en observant uniquement la vitesse à l'impact.

Typiquement, on tente de faire des régressions sur des données ayant de fortes corrélations, mais il existe de nombreuses variantes, avec différents niveaux de complexité, et parfois le coeur du problème est de modéliser l'incertitude et la variance des données. Le cas le plus simple est celui de la régression linéaire (qui devrait s'appeler régression affine), qui suppose une relation de la forme $\mathbf{y} = A\mathbf{x} + \mathbf{b}$ entre des données $\mathbf{x}$ et $\mathbf{y}$ (dans un cadre plus formel et probabiliste la relation porte sur l'espérance conditionnelle de $\mathbf{y}$, sachant $\mathbf{x}$). Plusieurs méthodes peuvent être utilisées pour ajuster les paramètres A et $\mathbf{b}$, la plus connue étant celle des moindres carrés.

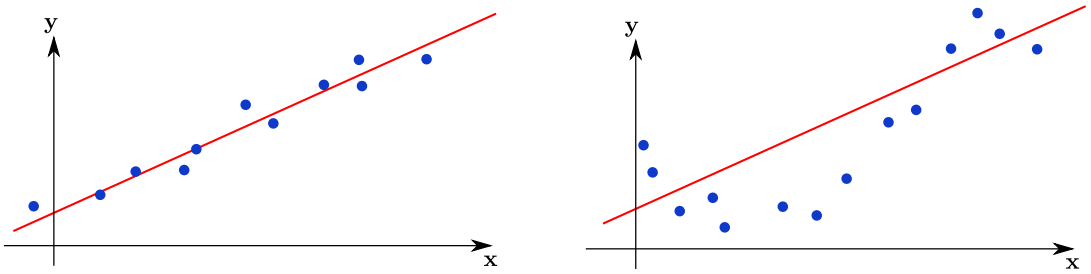

Figure 1: À gauche et à droite : des données plus ou moins adaptées à une régression linéaire

Souvent, les modèles linéaires sont insuffisants (cf. Figure ?, à droite). Dans ce TP, nous allons voir quelques méthodes simples permettant d'aller un peu plus loin (en restant dans un cadre non probabiliste). Il s'agit d'observer un ensemble des données $\{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}$, et d'ajuster un modèle $\mathbf{y} = f(\mathbf{x})$ qui s'écrit comme

une somme de k fonctions dépendant chacune d'un vecteur de paramètres θ_i:

$$f(\mathbf{x}) = \sum_{i=1}^{k} f_{\theta_i}(\mathbf{x})$$

On se place dans le cas où $\mathbf{y}$ est de dimension 1 (et l'on écrit dans ce qui suit y au lieu de $\mathbf{y}$).

1 Sommes pondérées de fonctions Gaussiennes

Dans cette partie, les fonctions f_{θ_i} ont pour paramètre un unique scalaire θ_i et s'écrivent :

$$f_{\theta_i} = \theta_i \phi_i(\mathbf{x}) = \theta_i \exp\left(-\frac{(\mathbf{x}-\mathbf{c}_i)^2}{\sigma_i^2}\right)$$

Les fonctions Gaussiennes $\phi_i(\mathbf{x})$ prennent des valeurs presques nulles partout sauf dans une région de l'espace d'entrée proche de $\mathbf{c}_i$: le *centre* fixé à l'avance pour chaque $\phi_i(x)$. Les paramètre σ_i sont également fixés (et ont généralement tous la même valeur).

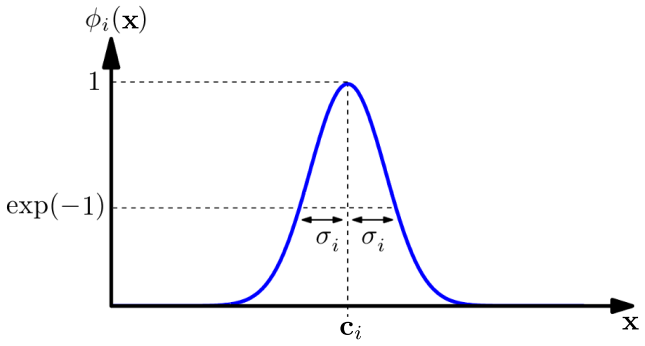

Figure 2: Une fonction Gaussienne en dimension 1

Si l'on pose $\phi(\mathbf{x}) = (\phi_1(\mathbf{x})\ \phi_2(\mathbf{x}) \cdots \phi_k(\mathbf{x}))^\mathsf{T}$ et $\theta = (\theta_1\ \theta_2 \cdots \theta_k)^\mathsf{T}$ on obtient :

$$f(x) = \phi(\mathbf{x})^\mathsf{T}\theta.$$

L'objectif de la régression est d'ajuster θ. Nous allons voir 3 méthodes : deux algorithmes incrémentaux et un de type "batch" qui traite l'ensemble des données observées en une fois.

1.1 Descente de gradient (méthode incrémentale)

Soit $\theta^{(t)}$ la valeur des paramètres à l'itération t. On observe une nouvelle donnée $(\mathbf{x}^{(t+1)}, y^{(t+1)})$. L'erreur de modélisation sur cette donnée est la suivante :

$$\epsilon^{(t+1)} = y^{(t+1)} - f_{\theta^{(t)}}\left(\mathbf{x}^{(t+1)}\right)$$

L'idée de la descente de gradient est de modifier légèrement θ pour améliorer l'erreur obtenue sur la dernière donnée observée. Pour cela, on considère la fonction $\theta \mapsto f_\theta\left(\mathbf{x}^{(t+1)}\right)$ en l'on calcule son gradient en $\theta^{(t)}$, que l'on nomme $\nabla_\theta^{(t+1)}$:

$$\nabla_\theta^{(t+1)} = \phi(\mathbf{x}^{(t+1)})$$

Remarque : la formule suivante permet de retrouver comment se calcule le gradient : $g(\mathbf{a}+\lambda\mathbf{b}) = g(\mathbf{a}) + \lambda(\nabla g(\mathbf{a}))^\mathsf{T}\mathbf{b} + o(\lambda)$.

Le gradient est orienté vers la pente maximale : il donne la direction et le sens permettant la plus abrupte augmentation de la fonction. Donc si $f_{\theta^{(t)}}\left(\mathbf{x}^{(t+1)}\right)$ doit être augmenté, c'est dans le sens de $\nabla_\theta^{(t+1)}$ qu'il

faut modifier θ. C'est le cas lorsque $\epsilon^{(t+1)}$ est positif. Inversement, si $\epsilon^{(t+1)}$ est négatif, il faut modifier θ dans le sens de $-\nabla_\theta^{(t+1)}$. La formule utilisée est la suivante :

$$\theta^{(t+1)} = \theta^{(t)} + \alpha\epsilon^{(t+1)}\nabla_\theta^{(t+1)} = \theta^{(t)} + \alpha\epsilon^{(t+1)}\phi(\mathbf{x}^{(t+1)}),$$

$\alpha > 0$ étant un coefficient appelé le "learning rate".

Instructions:

- Ouvrez le fichier `exoGD.py`. Il comprend la fonction `generateDataSample(x)` qui permet de générer une donnée bruitée y pour $\mathbf{x} \in [0,1]$ ($\dim(\mathbf{x}) = 1$), la fonction `phiOutput(input)` qui permet de générer le vecteur $\phi(\mathbf{x})$ ou une matrice de vecteurs $\phi(\mathbf{x}^{(i)})$ concaténés si l'entrée est un tuple, ainsi que la fonction `f(input, *user_theta)` qui permet de calculer $f(\mathbf{x})$. Les paramètres utilisés par `f` sont soit la variable globale `theta`, soit une valeur `*user_theta` fournie en entrée. Le nombre de composantes de $\phi(\mathbf{x})$ (c'est-à-dire le nombre k de fonctions Gaussiennes) est défini par la variable globale `numFeatures`.
- Implémentez la fonction `train_GD(maxIter)` qui ajustera la valeur de `theta` par descente de gradient à partir d'un nombre de données égal à `maxIter`. Lorsque le fichier est exécuté, les données observées sont affichées par des points, et la ligne rouge est la fonction "apprise", c'est-à-dire la fonction f correspondant aux paramètres `theta` (θ) ajustés par `train_GD(maxIter)`. Les autres courbes correspondent aux différents $f_{\theta_i}(\mathbf{x})$ et montrent comment la fonction f est décomposée.
- Essayez de trouver des valeurs de `maxIter`, `numFeatures` et du learning rate menant à de bons résultats (vous pouvez mettre des captures d'écran dans votre rapport).

1.2 Moindres carrés (méthode "batch")

Cette fois-ci on considère un ensemble de N données $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{1\leq i\leq N}$, et l'erreur suivante que l'on cherche à minimiser :

$$\epsilon(\theta) = \frac{1}{2N}\sum_{i=1}^{N}\left(y^{(i)} - f_\theta\left(\mathbf{x}^{(i)}\right)\right)^2$$

Si θ est un minimum local de la fonction ϵ, il correspond à un gradient nul. On cherche donc à résoudre :

$$\nabla\epsilon(\theta) = \mathbf{0}$$

Or, le gradient d'une somme est la somme des gradients, et on peut utiliser la formule suivant pour le gradient d'un produit de deux fonctions g et h : $\nabla(gh) = g\nabla h + h\nabla g$. On peut donc exprimer $\nabla\epsilon(\theta)$:

$$\nabla\epsilon(\theta) = \frac{1}{N}\sum_{i=1}^{N} -\left(y^{(i)} - f_\theta\left(\mathbf{x}^{(i)}\right)\right)\phi(\mathbf{x}^{(i)}) = -\frac{1}{N}\sum_{i=1}^{N}\phi(\mathbf{x}^{(i)})\left(y^{(i)} - \phi(\mathbf{x}^{(i)})^\mathsf{T}\theta\right)$$

Pour annuler $\nabla\epsilon(\theta)$, on tente donc d'obtenir:

$$\left(\sum_{i=1}^{N}\phi(\mathbf{x}^{(i)})\phi(\mathbf{x}^{(i)})^\mathsf{T}\right)\theta = \sum_{i=1}^{N}\phi(\mathbf{x}^{(i)})y^{(i)}$$

Ce que l'on réécrit :

$$A\theta = b$$

A n'étant pas forcément une matrice inversible, pour s'approcher de cette égalité le plus possible on peut utiliser la *pseudo-inverse* de A (`np.linalg.pinv()`), $A^\sharp$ et poser:

$$\theta = A^\sharp b$$

Instructions:

- Ouvrez le fichier `exoLS.py`. Sa structure est similaire à celle de `exoGD.py`, mais les données sont construites en une fois au lieu d'être créées progressivement. Ce sont les lignes `x = np.random.random(1000)` (le nombre de données peut être modifié) et `y = map(generateDataSample, x)`.
- Implémentez la fonction `train_LS()` qui calculera `theta` suivant la méthode des moindres carrés.
- Essayez de trouver à nouveau des paramètres menant à de bons résultats.
- Maintenant que vous avez pu tester une méthode incrémentale et une méthode "batch", quels sont selon vous les avantages et les inconvénients de la méthode des moindres carrés ?

1.3 Algorithme des moindres carrés récursifs (méthode incrémentale)

L'algorithme des moindres carrés récursifs est une variante incrémentale de la méthode des moindres carrés, dans laquelle A et b sont recalculés à chaque nouvelle donnée, ce qui se fait très simplement puisque A et b s'écrivent comme une somme sur les données. On a :

$$A^{(t+1)} = A^{(t)} + \phi(\mathbf{x}^{(t+1)})\phi(\mathbf{x}^{(t+1)})^\mathsf{T}$$
$$b^{(t+1)} = b^{(t)} + \phi(\mathbf{x}^{(t+1)})y^{(t+1)}$$

Les paramètres sont obtenus par $\theta^{(t+1)} = (A^{(t+1)})^\sharp b^{(t+1)}$, mais peuvent aussi être estimés en utilisant le lemme de Sherman-Morrison:

$$(A + uv^\top)^\sharp = A^\sharp - \frac{A^\sharp uv^\top A^\sharp}{1 + v^\top A^\sharp u} \tag{1}$$

Remarque : pour appliquer ce lemme il faut commencer avec une valeur non-nulle pour $A^{(0)}$.

Instructions:

- Ouvrez le fichier `exoRLS.py`. Sa structure est très similaire à celle de `exoGD.py`
- Implémentez la fonction `train_RLS()` qui ajustera `theta` de façon incrémentale en suivant la méthode des moindres carrés récursifs (sans utiliser le lemme de de Sherman-Morrison), et montrez dans votre rapport les résultats obtenus.
- Ouvrez le fichier `exoRLS2.py`, et implémentez cette fois la méthode des moindres carrés récursifs en utilisant le lemme de Sherman-Morrison.
- Comparez les deux variantes (avec ou sans le lemme de Sherman-Morrison). Quelle est la plus précise, quelle est la plus rapide, et pourquoi (vous pouvez inclure dans votre rapport des mesures de temps de calcul) ?

2 LWLS : "Locally-Weighted Least-Squares" (méthode "batch")

L'algorithme LWLS utilise une somme pondérée de modèles linéaires locaux, paramétrés par des vecteurs θ_i tels que $\dim(\theta_i) = \dim(\mathbf{x}) + 1 = d + 1$:

$$f(\mathbf{x}) = \sum_{i=1}^{k} \frac{\phi_i(\mathbf{x})}{\sum_{j=1}^{k} \phi_j(\mathbf{x})} m_{\theta_i}(\mathbf{x}),$$

avec $m_{\theta_i}(\mathbf{x}) = w(\mathbf{x})^\mathsf{T}\theta_i$ et $w(\mathbf{x}) = (\mathbf{x}_1\ \mathbf{x}_2 \cdots \mathbf{x}_d\ 1)^\mathsf{T}$.

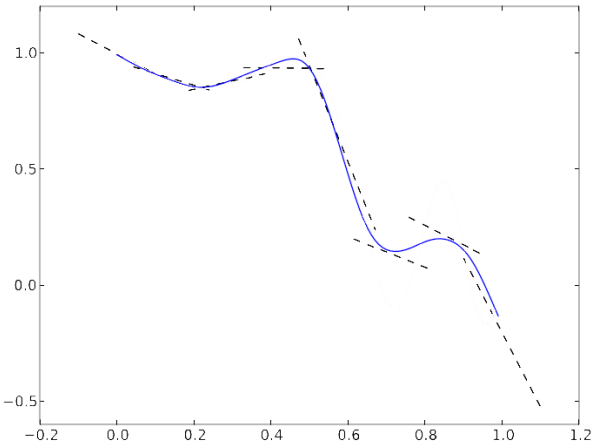

Figure 3: Une fonction obtenue avec LWLS

Chaque modèle local est calculé à partir de l'erreur localement pondérée suivante :

$$\epsilon_i(\theta_i) = \frac{1}{2N}\sum_{j=1}^{N} \phi_i(\mathbf{x}^{(j)}) \left(y^{(j)} - m_{\theta_i}(\mathbf{x}^{(j)})\right)^2 = \frac{1}{2N}\sum_{j=1}^{N} \phi_i(\mathbf{x}^{(j)}) \left(y^{(j)} - w(\mathbf{x}^{(j)})^\mathsf{T}\theta_i\right)^2$$

Comme pour la méthode des moindres carrés, on tente d'annuler son gradient, ce qui mène à essayer de résoudre :

$$-\frac{1}{N}\sum_{j=1}^{N} \phi_i(\mathbf{x}^{(j)}) w(\mathbf{x}^{(j)}) \left(y^{(j)} - w(\mathbf{x}^{(j)})^\mathsf{T}\theta_i\right) = 0$$

On pose donc $\theta_i = A_i^\sharp b_i$, avec :

$$A_i = \sum_{j=1}^{N} \phi_i(\mathbf{x}^{(j)}) w(\mathbf{x}^{(j)}) w(\mathbf{x}^{(j)})^\mathsf{T}$$

$$b_i = \sum_{j=1}^{N} \phi_i(\mathbf{x}^{(j)}) w(\mathbf{x}^{(j)}) y^{(j)}$$

Instructions:

- Ouvrez le fichier `exoLWLS.py`. Il comprend les fonctions `generateDataSample(x)`, `phiOutput(input)`, et la fonction `f(input)` qui est cette fois différente et fait appel à `w(input)` qui calcule les $w(\mathbf{x})$ pour une ou plusieurs valeurs de $\mathbf{x}$. Notez que désormais `theta` est une matrice formée de la concaténation horizontale des θ_i qui sont des vecteurs de 2 paramètres (car on est dans le cas où $\dim(\mathbf{x}) = 1$).
- Implémentez la fonction `train_LWLS()` qui calcule `theta`. Montrez à nouveau dans votre rapport les résultats obtenus.
- Pour des paramètres similaires, comparez les résultats obtenus avec la méthode LWLS et la méthode des moindres carrés (`exoLS.py`). Quelle méthode est la plus rapide, et quelle méthode donne selon vous les meilleurs résultats ? Quelles sont les différences principales si par exemple on augmente `numfeatures` ?
- Selon les cas de figure, comment choisisseriez-vous entre une méthode incrémentale et une méthode "batch" ?

- Quelles modifications (autres que les “méta-paramètres”) pourriez-vous apporter aux algorithmes pour obtenir des approximations encore plus précises ?