Regression en C++

Prérequis : Pour mener à bien ce travail, il est nécessaire d'avoir installé les outils classiques pour le C++ (compilateur, CMake, etc.) ainsi que la bibliothèque matplotlib-cpp permettant l'affichage graphique (via Python).

Contexte du travail : On s'intéresse ici à l'implémentation en C++ d'un algorithme ultra-classique en *Machine Learning*, la régression linéaire. Pour plus d'information, je conseille l'excellent cours d'Andrew Ng, fondateur du site coursera (voir https://www.coursera.org/learn/machine-learning/), plateforme qui dispense de nombreux MOOCs de qualité sur cette thématique. Le contexte général du *Machine Learning* est d'appliquer une méthode dite d'*apprentissage* sur des *données* afin de *prédire* des *informations* utiles ou de déterminer des corrélations entre les données.

La méthode de régression

On suppose dans cette partie que l'on dispose d'un ensemble de m couples $(x_i, y_i)_{i=1\dots m}$ où $x_i \in \mathbb{R}$ et $y_i \in \mathbb{R}$. Ces points suivent une relation donnée par une fonction f inconnue que l'on souhaite approcher, c'est-à-dire que l'on a $y_i = f(x_i)$ pour tout i. Le but de ce TP est de proposer une méthode pour approcher la fonction f à partir des données (x_i, y_i). La méthode proposée ici repose sur la determination des paramètres $\theta \in \mathbb{R}$ d'une certaine fonction hypothèse notée h_θ que l'on pose sur la relation entre les données. Autrement dit, on suppose que la fonction f peut être approchée par l'hypothèse h_θ (pour des valeurs de θ bien choisies) :

$$h_\theta(x) = \hat{y} \simeq y = f(x)$$

où $\hat{y}$ est appelée la prédiction de x idéalement proche de y.

Le choix de l'hypothèse est crucial pour la précision de l'approximation. Si la relation entre x et y est linéaire, la fonction h_θ suivante est alors une bonne approximation de f :

$$\hat{y} = h_\theta(x) = \theta_0 + \theta_1 x,$$

où $x \in \mathbb{R}$. Les paramètres θ_0 et θ_1 sont donc à determiner. De même, si la relation entre les données est par exemple un polynôme de degré 2, l'hypothèse pertinente a la forme suivante :

$$\hat{y} = h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2,$$

et les paramètres θ_0, θ_1 et θ_2 sont ici à determiner.

L'ensemble des prédictions est noté $Y \in \mathbb{R}^m$ et l'ensemble des caractéristiques (ou *features*) servant à la prédiction est noté $X \in \mathbb{R}^m$. Le vecteur de paramètres θ est à n composantes, i.e. $\theta = \{\theta_i\} \in \mathbb{R}^n$ et c'est l'inconnue de notre problème qu'il nous faut découvrir. Une fois ce vecteur connu, il est alors possible d'utiliser l'hypothèse h_θ en prédiction. L'idée est de déterminer le vecteur θ de manière à ce que l'écart entre y_i et notre hypothèse $h_\theta(x_i)$ soit le plus petit possible pour l'ensemble des données. Autrement dit, on cherche θ tel que :

$$h_\theta(x_i) = \hat{y}_i \simeq y_i = f(x_i) \qquad \forall\, 0 \leq i \leq m.$$

On espère ensuite que l'hypothèse se *généralise*, c'est-à-dire que la prédiction d'une donnée inconnue x vérifie bien $\hat{y} \simeq y = f(x)$.

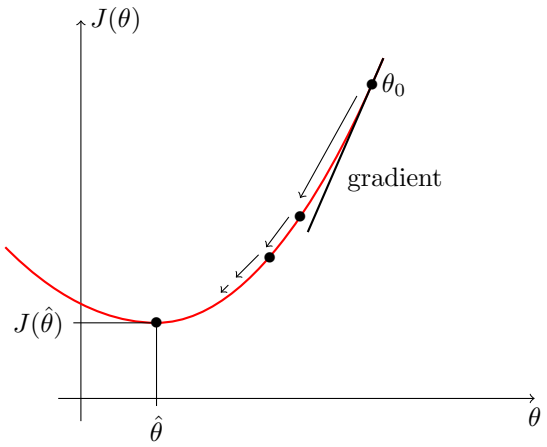
Pour évaluer la performance de l'hypothèse, nous utilisons une fonction dite *de coût* permettant pour un vecteur θ donné de mesurer l'écart entre l'hypothèse et les données. A moins d'avoir une hypothèse parfaitement adaptée aux données, cet écart n'est jamais nul mais on cherche à le minimiser. La fonction de coût considérée est l'erreur quadratique moyenne :

$$J(\theta) = \frac{1}{2m} \sum_{i=1\dots m} (h_\theta(x_i) - y_i)^2.$$

Notons qu'il existe d'autres fonctions de coût utilisables.

Ainsi, notre but est de trouver les meilleures valeurs de θ minimisant la fonction $J(\theta)$. On note ce minimum $\hat{\theta}$, on suppose qu'il existe et qu'il est atteignable. Pour cela, on utilise une méthode itérative de *descente de gradient*. L'idée de cette méthode est de minimiser en suivant les *gradients* de la fonction.

C'est un algorithme massivement utilisé en *Machine Learning* car il est peu coûteux et dispose de bonnes propriétés pour cette discipline. On parle alors d'*apprentissage* car d'une manière imagée, on apprend à partir des données pour fixer les paramètres afin de prédire efficacement.



L'algorithme de descente considéré consiste en une succession de corrections opérées de manière itérative sur le paramètre θ :

$$\theta := \theta - \frac{\alpha}{m} \sum_{i=1}^{m} (h_\theta(x_i) - y_i) \, \nabla h_\theta(x_i),$$

où on note α un paramètre dit d'*apprentissage*. Autrement dit, l'algorithme en pseudo-code est le suivant :

Algorithm 1 Calcul θ minimisant $J(\theta)$

Require: θ, α, n_{max}
1: $c = \frac{\alpha}{m}$
2: **for** $k = 1...n_{max}$ **do**
3: $\quad \Delta\theta = 0$
4: $\quad$ **for** $i = 1...m$ **do**
5: $\quad\quad h = h_\theta(x_i) - y_i$
6: $\quad\quad$ **for** $j = 1...n$ **do**
7: $\quad\quad\quad \nabla h = \frac{\partial h}{\partial \theta_i}(x_i)$
8: $\quad\quad\quad \Delta\theta_j = \Delta\theta_j + ch\nabla h$
9: $\quad\quad$ **end for**
10: $\quad$ **end for**
11: $\quad \theta = \theta - \Delta\theta$
12: **end for**

Travail demandé : Le but de ce travail est d'implémenter en `C++` l'algorithme de régression présenté dans la partie précédente :

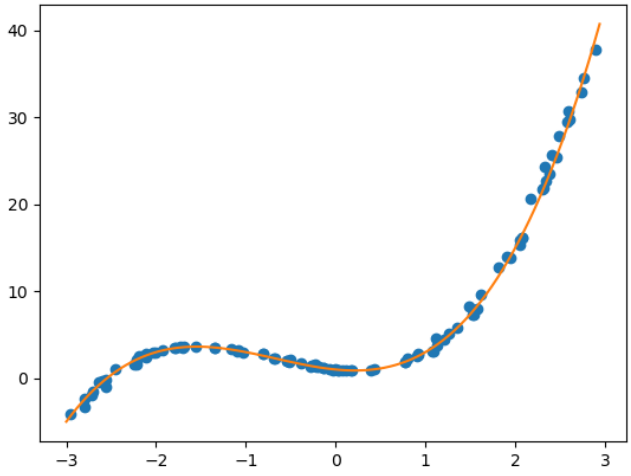
- L'implémentation devra utiliser au maximum la librairie standard `stl` et les possibilités du langage offertes par les normes `C++11/14/17`.
- Il conviendra d'utiliser la programmation orientée objet. Pour cela, il sera nécessaire de définir les classes suivantes :
  - `Hypothesis` : classe permettant de modéliser les hypothèses h_θ. Cette classe devra contenir les paramètres θ_i en attribut (i.e. un tableau de `double`). Une méthode devra permettre le calcul de l'hypothèse $h_\theta(x)$ pour une nouvelle données x (i.e. un `double`, ligne 5.). Une méthode devra permettre le calcul du gradient de l'hypothèse $\frac{\partial h}{\partial \theta_i}(x)$ en un point donnée x par rapport à un paramètre (ligne 7.).
  - `Optimizer` : classe permettant de gérer les incréments des paramètres durant le processus itératif. Ainsi, la classe devra contenir les incréments $\Delta\theta$ (i.e. un tableau de `double`, ligne 3.) et une méthode devra permettre la mise à jour des paramètres du modèle (voir ligne 11.).
  - `LossFunction` : classe permettant de modéliser la fonction de coût $J(\theta)$. Une méthode devra permettre de calculer un élément de la fonction de coût par rapport à un couple de données (x, y) (i.e. deux `double`, ligne 5.). De plus, une méthode devra permettre la mise à jour des incréments de gradient $\Delta\theta$ (ligne 5. à 9.).

- `Trainer` : classe permettant de coordonner les différents objets durant la double boucle d'apprentissage (ligne 2. et ligne 4.).

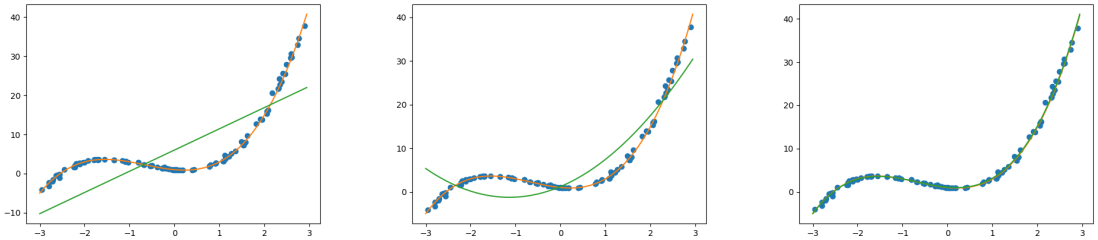
- Les données devront être générées de manière aléatoire pour tester votre algorithme. Typiquement, vous pourrez générer différentes fonctions (polynomiales ou approchables par un polynômes) pour tester plusieurs hypothèses. Je conseille de tester les fonctions $f_0(x) = ax + b$, $f_1(x) = ax^2 + bx + c$ et éventuellement $f_2(x) = cos(2\pi x)$. Ainsi, vous pourrez générer m points aléatoires x_i dans un intervalle puis générer les points $y_i = f(x_i)$. Il est aussi possible d'ajouter un bruit aléatoire à la génération $y_i = f(x_i + \epsilon)$.

Attention, ceci ne sont que des indications, vous avez toute latitude pour proposer votre modélisation. Vous pouvez notamment modifier les noms des classes à votre convenance. Vous pouvez ajouter des espaces de nommage par exemple.

Quelques exemples : Dans la figure ci-dessous, voici les points générés avec une petite perturbation à partir de la fonction exacte en orange.



Dans les figures ci-dessous, trois exemples d'hypothèses (linéaire, polynôme de degré 2 et 3). On remarque que le polynôme de degré 3 colle parfaitement à la courbe de référence.



Liens utiles : Voici quelques ressources qui devraient vous inspirer :

- `https://fr.wikipedia.org/wiki/Erreur_quadratique_moyenne`
- `https://fr.wikipedia.org/wiki/Algorithme_du_gradient`

et pour les plus courageux :

- `https://ruder.io/optimizing-gradient-descent/`
- `https://arxiv.org/abs/1609.04747`