Quentin Rod 2ème ISC
Thomas Dagier

Rapport sur l'Optimisation

Introduction

Dans le cadre du cours de Mathématiques en technologies de l'information nous souhaitons réaliser de l'optimisation. Nous voulons, à partir d'un nuage de points, trouver un modèle permettant de représenter cet ensemble sous la forme d'une fonction mathématique. De nombreuses modélisations sont alors possibles. Dans ce travail, nous nous intéressons à effectuer une régression linéaire soit l'obtention de la meilleure droite possible.
Pour cela, nous devons faire un projet en C qui implémentera cette régression linéaire. Une librairie réalisée en C et en python nous est fournie et permet d'exporter nos données dans des fichiers au format « .vec » pour ensuite les afficher à l'écran.
Dans ce rapport, nous allons évoquer notre approche théorique qui nous a permis de réaliser ce travail en abordant les différentes méthodes permettant de trouver la meilleure droite. Les résultats obtenus seront ensuite expliqués et développés de sorte à bien comprendre notre méthodologie de travail et la réalisation qui en découle.

Théorie :

En admettant que notre modèle permet de trouver la meilleure droite d'un nuage de points, on pose son équation : $y = a * x + b$
Un nuage de points est un ensemble de points. Chacun est donc un couple (x, y). En C, nous avons choisi de les représenter sous la forme d'une structure $point\_2d$.

```
typedef struct {
    double x,y;
} point_2d;
```

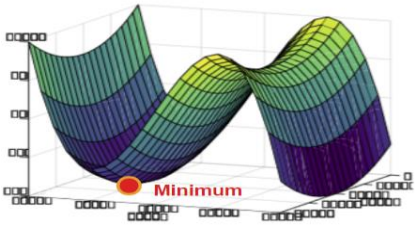
Nous avons choisi de représenter le nuage de points sous la forme d'un tableau de $point\_2d$ contenant successivement chacun des couples de notre ensemble.

```
typedef struct {
    point_2d* array;
    int length;
} arrayPoints;
```

La première étape est de définir une fonction de coût. Nous la définissons comme suivant:

$$E(a,b) = \sum_{i=1}^{N} (a * x_i + b - y_i)^2 \text{ avec le point } (x_i, y_i) \text{ à l'indice } i \text{ du tableau}$$

Pour tout a et b qui vérifient l'équation de la droite en chaque point de notre ensemble, on cherche cette droite qui est la plus proche du y_i de chaque point. Pour cela, on somme la différence entre l'image d'une droite quelconque en chacun des points et la véritable image y_i du point. On recherche donc les coefficients a et b pour lesquels le coût est le plus faible. On parle alors de minimisation de la fonction de coût.
Ce minimum est le point (a, b). Pour le trouver, nous utilisons le gradient de la fonction de coût. Le gradient en un point de la fonction correspond à un vecteur indiquant la direction où la variation est la plus forte au voisinage de ce point.



Obtention du minimum global

$$\vec{\nabla} f = \begin{pmatrix} \frac{df}{dx_1} \\ \vdots \\ \frac{df}{dx_n} \end{pmatrix}$$

Formule du gradient

Les composantes de ce vecteur sont les dérivées partielles en chacune des dimensions. Dans notre cas nous sommes limités à un gradient à 2 composantes.
La droite que nous cherchons à obtenir peut se trouver de plusieurs manières. Une solution simple, qui nous permet de vérifier que notre programme fonctionne, est de trouver la pente a et l'ordonnée à l'origine b de manière analytique.

Solution analytique

Cette solution est applicable dans notre optimisation car elle se fait en 2 dimensions. Il est donc possible de calculer la valeur exacte de a et b puisque nous avons 2 équations à 2 inconnues à résoudre.

Le point (a, b) a donc un gradient nul ce qui revient à écrire : $\vec{\nabla} E(a, b) = 0$.
Mathématiquement, on pose notre fonction de coût :

$$E(a, b) = \sum_{i=1}^{N} (a * x_i + b - y_i)^2$$

On a donc une fonction à deux inconnues. Le gradient en chaque point de la fonction correspond donc à :

$$\vec{\nabla} E(a, b) = \begin{pmatrix} \frac{dE}{dx_i} \\ \frac{dE}{dy_i} \end{pmatrix} = \begin{pmatrix} 2 * (a \sum_{i=1}^{N} x_i^2 \quad + \quad b * \sum_{i=1}^{N} x_i \quad - \quad \sum_{i=1}^{N} x_i * y_i) \\ 2 * (a * \sum_{i=1}^{N} x_i \quad + \quad b * N \quad - \quad \sum_{i=1}^{N} y_i) \end{pmatrix}$$

Pour rappel, nous cherchons à résoudre l'équation $\vec{\nabla} E(a, b) = 0$. Cela revient à résoudre un système à deux inconnues a et b. On obtient alors les valeurs suivantes.

$$a = \frac{N * \sum_{i=1}^{N} x_i * y_i \; - \sum_{i=1}^{N} x_i * \sum_{i=1}^{N} y_i}{N * \sum_{i=1}^{N} x_i^2 \; - \left(\sum_{i=1}^{N} x_i\right)^2}$$

$$b = \frac{\sum_{i=1}^{N} x_i * y_i \; - \; a * \sum_{i=1}^{N} x_i^2}{\sum_{i=1}^{N} x_i}$$

Dans notre programme, nous utilisons ces dernières formules dans les fonctions :

- **double calculationCoefficientA(arrayPoints array),** pour trouver la pente a
- **double calculationCoefficientB(arrayPoints array, double A),** pour trouver l'ordonnée à l'origine b.

Dans le cas où notre équation de coût est à N inconnues, nous devons résoudre N équations à N inconnues. Cela devient alors très difficile ou impossible et il faut opter pour une méthode numérique. C'est-à-dire, trouver une approximation de ces deux valeurs.

Solution numérique

La méthode utilisée est la descente de gradient. Elle consiste à partir d'un point de départ (a_0, b_0) et à se déplacer dans la direction inverse du gradient en ce point, soit - $\vec{\nabla}E(a,b)$. Avancer dans la direction opposée du gradient signifie que l'on se déplace où la fonction descend le plus fortement. L'objectif est donc de se rapprocher le plus possible du point minimum (a, b).

Chaque nouveau point est donc défini ainsi avec $0 < \lambda < 1$.

$$\begin{pmatrix} a_{i+1} \\ b_{i+1} \end{pmatrix} = \begin{pmatrix} a_i \\ b_i \end{pmatrix} - \lambda * \vec{\nabla}E(a_i, b_i)$$

Également, nous définissons Epsilon qui représente la précision. Il correspond à la distance maximale souhaitée entre le dernier et l'avant dernier point.

$$\left|\left| \begin{pmatrix} a_{i+1} \\ b_{i+1} \end{pmatrix} - \begin{pmatrix} a_i \\ b_i \end{pmatrix} \right|\right| < \epsilon$$

Dans notre programme, nous utilisons ces dernières formules dans les fonctions :

- **point_2d descenteGradient(double epsilon, arrayPoints arrayOfPoints, point_2d Xi, double lambda),** pour réaliser la descente de gradient et retourner le dernier point (a, b) calculé lorsque la distance est inférieure à ϵ
- **point_2d gradientOnPoint(double a, double b, arrayPoints data)**, pour trouver le gradient en un point de notre ensemble (x_i, y_i).

Afin de calculer le gradient en un point de l'ensemble, on utilise la formule du gradient montrée précédemment. Soit :

$$\vec{\nabla}E(a,b) = \begin{pmatrix} \frac{dE}{dx_i} \\ \frac{dE}{dy_i} \end{pmatrix} = \begin{pmatrix} 2 * (a \sum_{i=1}^{N} x_i^2 + b * \sum_{i=1}^{N} x_i - \sum_{i=1}^{N} x_i * y_i) \\ 2 * (a * \sum_{i=1}^{N} x_i + b * N - \sum_{i=1}^{N} y_i) \end{pmatrix}$$

Cependant, la fonction de coût étant constituée de nombreuses sommes, si le nombre de points est très grand, il est possible d'avoir un dépassement de capacité lors du calcul. Il faudrait donc adapter en fonction de l'ensemble étudié les valeurs de lambda et epsilon.

Afin de contrer ce problème, nous avons normalisé le point initial, le vecteur de x ainsi que le vecteur de y avant la phase de calcul de la fonction **descenteGradient()**. Le point final obtenu est donc lui aussi normalisé. L'opération inverse est effectuée sur les vecteurs et sur ce point à la fin de la fonction afin d'avoir les valeurs non normalisées de a et b.

Résultats

Notre optimisation nous permet de trouver les coefficients a et b qui définissent la "meilleure droite" de notre nuage de points. Notre implémentation analytique, trouvant les valeurs exactes de ces coefficients devient complexe à utiliser pour une fonction de coût ayant plus de paramètres. Notre implémentation numérique, elle, est utile pour n'importe quelle fonction de coût mais ne garantit pas de trouver les coefficients de la meilleure droite.
Il faut donc comparer les deux implémentations pour être sûr du résultat obtenu.

Premièrement, dans notre programme, nous générons aléatoirement un nuage de points qui est borné entre $(x_{min}, y_{\min})$ et $(x_{max}, y_{\max})$. Ce sont 4 points choisis arbitrairement pour délimiter notre ensemble. À partir de ce nuage, nous calculons les valeurs de la pente a et l'ordonnée à l'origine b avec les deux implémentations :

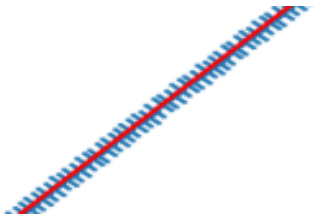
```
Solution analytique : a=0.01, b=4.98

Solution numérique : a=0.01, b=4.98
```

Si dans les deux cas il est retourné les mêmes valeurs, on estime que les coefficients sont corrects. Une manière de vérifier que notre programme fonctionne correctement est d'ajouter du bruit.
On pose donc $y_i = a * x_i + b + r_i$ avec r_i, le bruit aléatoire, à ajouter à notre précédente équation de droite.

Dans ce test, nous choisissons $a = 6.0, b = 15.0\ et\ -5 < r_i < 5$:

```
Test avec du bruit :
        Analytiquement: a=6.00, b=15.07, error=5859.00
        Numériquement:  a=6.00, b=15.01, error=5861.00
```



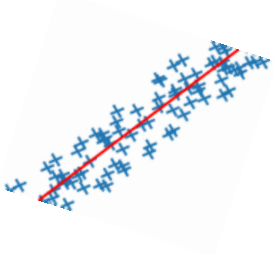
Ce test nous permet de vérifier que nos implémentations sont correctes même en présence de bruit mais aussi que les coefficients retournés soient très proches de ceux choisis.
Cependant, ces tests nous ont aussi permis de voir qu'en agrandissant notre ensemble, un dépassement de capacité est obtenu lors du calcul des coefficients. La normalisation a permis de résoudre ce problème.

Afin d'établir une notion d'erreur du modèle, nous définissons arbitrairement une formule pour la calculer: $\sum_{i=1}^{N}(a * x_i + b - y_i)^2$. Nous effectuons alors plusieurs tests en modifiant les paramètres a, b et r_i.

Sur plusieurs droites générées avec $a = 6.0, b = 15.0\ et\ -5 < r_i < 5$, l'erreur moyenne reste aux alentours de 5800. Cette valeur ne change véritablement que si la valeur de r_i n'est plus comprise dans le même intervalle. Observons le modèle obtenu avec des bruits différents :

Pour $a = 6.0, b = 15.0\ et\ -2 < r_i < 2$, notre erreur moyenne avoisine 500.

```
Analytiquement: a=6.00, b=14.92, error=488.00
Numériquement:  a=6.00, b=14.86, error=489.00
```

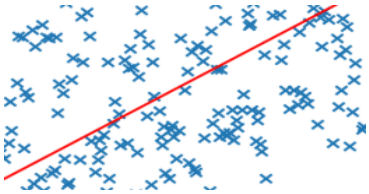


Pour $a = 6.0, b = 15.0\ et\ -300 < r_i < 300$:

```
Analytiquement: a=6.01, b=11.97, error=28987328.00
Numériquement:  a=6.01, b=11.91, error=28987329.00
```

Pour $a = 6.0, b = 15.0\ et\ -1000 < r_i < 1000$:

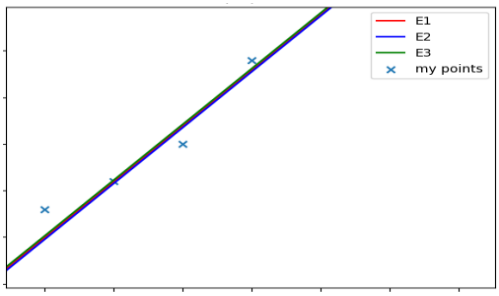
```
Analytiquement: a=6.01, b=21.73, error=335566393.00
Numériquement:  a=6.01, b=21.67, error=335566394.00
```

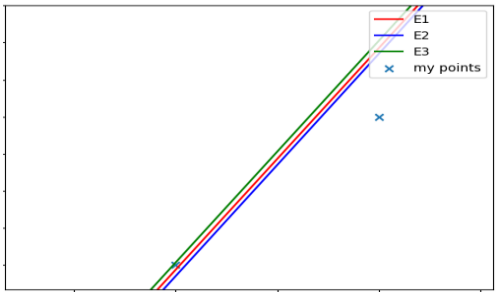


Comme nous pouvons le voir, plus le bruit est grand, plus l'erreur sera grande. Or, l'erreur moyenne donnant une indication sur la qualité de notre modèle, plus l'erreur est petite, plus notre modèle est proche des valeurs du nuage de points. De ce fait, si le bruit est important, notre erreur sera très grande et on obtiendra de mauvais coefficients.

Un autre moyen de tester le modèle est de faire une validation croisée. Toujours avec le même calcul d'erreur, nous regardons si les coefficients a et b continuent de rester cohérent sur un ensemble réduit de points. Après avoir séparé notre ensemble en trois portions égales, nous calculons le coefficient a et b à partir de deux sous-ensembles puis regardons si ces valeurs sont cohérentes sur le troisième. Si l'erreur est proche dans les 3 cas, c'est que notre modèle est cohérent pour l'ensemble du nuage même si nous y ajoutons des points. L'entraînement est effectué sur 1000 points avec les valeurs suivantes: $a = 6.0, b = 15.0\ et\ -5 < r_i < 5$.

```
Entrainement :
        G1 U G2 sur G3 : a=6.00, b=15.08, error=2014.00
        G1 U G3 sur G2 : a=6.00, b=15.15, error=1917.00
        G2 U G3 sur G1 : a=6.00, b=14.99, error=1939.00
```





Nous pouvons observer une erreur moyenne qui est relativement proche pour les trois entraînements. Les coefficients trouvés le sont aussi. Par conséquent, nous pouvons considérer que notre modèle est correct.

Conclusion

À travers ce travail pratique, nous avons renforcé les notions de mathématiques vues en cours. Nous avons observé par la pratique les différences concrètes entre une méthode analytique et numérique. Egalement, les tests et les entraînements nous ont servi à comprendre nos modèles tout en tirant parti des résultats obtenus. Nous avons donc pu comprendre les enjeux et déduire les bonnes conclusions.
Cette implémentation est un cas très simplifié de l'optimisation puisque notre fonction de coût reste une fonction affine. Il pourrait alors être intéressant de regarder comment implémenter une solution permettant de faire un modèle plus complexe avec des fonctions de coût à plusieurs degrés. Sans changer l'optique du travail, nous pourrions aussi envisager de faire des tests différents et d'observer le comportement et les changements obtenus.