ITI 1521. Introduction à l'informatique II
Hiver 2018

Devoir 1

Échéance : 2 février 2018, 23 h 30

[PDF]

Objectifs d'apprentissages

- Maîtriser les concepts liés aux tableaux
- Manipuler des variables références
- Revisiter les concepts de base de la programmation orientée objet
- Éditer, compiler et exécuter des programmes Java
- Sensibiliser les étudiants face au problème du plagiat,
  et connaître les règlements de l'université à ce sujet

Introduction

L'intelligence artificielle est un sujet d'actualité. Tous les jours, les médias publient des articles sur ce sujet. Ces articles portent principalement sur l'apprentissage automatique qui est une branche de l'intelligence artificielle. C'est d'ailleurs un domaine où le Canada exerce un très grand leadership. Les chercheurs du domaine de l'apprentissage automatique développent des algorithmes qui apprennent à partir de données. Concrètement, ce sont des algorithmes qui déterminent les valeurs des paramètres d'un modèle à partir d'un jeu de données. Ces modèles sont alors utilisés pour faire des prédictions à partir de nouvelles observations. Pour ce devoir, nous nous intéressons à un modèle simple, la régression linéaire. C'est aussi un modèle bien établi en statistique. Nous concevons des programmes informatiques pour déterminer les valeurs «optimales» d'un modèle linéaire à partir de données. Ce devoir s'inspire du cours en ligne «*Machine Learning*» d'Andrew Ng. Notamment, les équations sont tirées de ce cours :

- https://www.coursera.org/learn/machine-learning.

Vous pouvez consulter une vidéo d'introduction sur YouTube :

- https://youtu.be/GnpUypfFkgo?t=3065

Algorithme du gradient et le modèle de régression linéaire

Imaginez que nous disposons d'un ensemble de points dans un plan à deux dimensions et que l'on souhaite tracer une droite représentative de ce jeu de données. De façon concrète, il s'agit d'une droite minimisant la distance moyenne de chaque point à la droite. Par exemple, sur la Figure 1, la droite en rouge est représentative du nuage de points blues.

Bien sûr, vous pourriez écrire un système d'équations et le résoudre afin de trouver cette droite. Mais il y a un autre moyen. Vous pouvez utiliser un algorithme fréquemment utilisé dans l'apprentissage automatique. Ce dernier trouvera une solution par essai et erreur.

Pour ce devoir, nous implémentons un tel algorithme, appelé *algorithme du gradient*. Nous traitons d'abord le cas d'une seule variable libre afin de trouver une droite dans un plan à deux dimensions. Nous généraliserons par la suite notre solution pour qu'elle puisse fonctionner avec un nombre arbitraire de variables. Cela signifie que notre ensemble de données peut comporter un nombre arbitraire de dimensions (*attributs ou observations*), et nous trouverons une équation linéaire qui correspond à l'ensemble des données.

À titre d'exemple, nous pourrions utiliser notre algorithme pour le marché de l'immobilier. Dans un premier temps, nous pourrions simplement nous intéresser à la taille des maisons. Nous devrions colliger un grand nombre

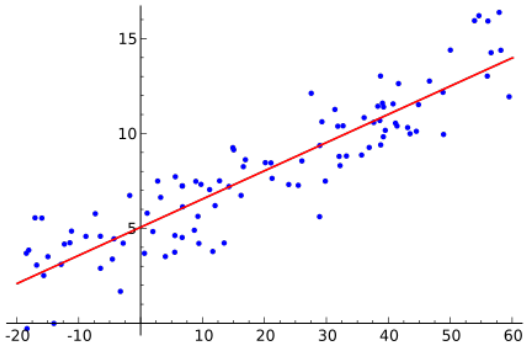

FIGURE 1 – Régression linéaire (Source : Sewaqu https://commons.wikimedia.org/w/index.php?curid=11967659.)

d'exemples de ventes de maisons dans la région, nous pourrions établir une base de données mettant en relation le prix de vente et la taille des maisons (une seule variable libre). Nous utilisons notre application afin de trouver une droite représentative des données. Par la suite, nous pourrions utiliser ce modèle linéaire pour établir le prix de vente d'une maison en fonction de sa taille.

Cela peut fonctionner dans une certaine mesure, mais nous allons bientôt réaliser que la taille seule ne suffit pas pour prédire le prix de vente avec précision. Un certain nombre d'autres facteurs sont importants : le nombre de pièces, le voisinage de la maison, l'état du bâtiment, l'année de construction, ayant un sous-sol fini, un garage, etc. Nous pouvons donc enrichir notre ensemble de données avec toutes ces variables (appelé *attributs ou observations*), et exprimer le prix de vente en fonction de toutes ces informations. Nous utiliser ensuite notre application pour trouver un bon ajustement. Nous serons maintenant en mesure de prédire le prix de vente, compte tenu de l'ensemble des caractéristiques de cette maison. Idéalement, cette prédiction sera maintenant plus précise.

Cette approche est assez limitée. Notamment, toutes les relations ne sont pas linéaires, donc une solution linéaire sera une mauvaise solution dans de nombreux cas. Mais l'objectif de ce devoir n'est pas d'étudier la régression linéaire ou la descente de gradient. Vous trouverez des informations complémentaires sur ces sujets plus tard au cours de vos études, si vous vous inscrivez aux cours correspondants. Cependant, si vous êtes curieux, vous pouvez facilement trouver des informations en ligne ; un bon point de départ, qui a été l'inspiration pour ce devoir, est le cours en ligne «*Machine Learning*» d'Andrew Ng (`https://www.coursera.org/learn/machine-learning`).

Algorithme du gradient pour une seule variable libre

Pour cette première application, nous travaillons dans le plan. Nous utilisons un ensemble de données que l'on appelle *ensemble d'apprentissage*. Nous avons un total de m échantillons de données. Chaque échantillon est une paire (x_i, y_i), ou x_i est l'entrée et y_i est la sortie (et donc le point (x_i, y_i) dans le plan).

Nous recherchons une ligne droite qui correspondrait le mieux à tous nos points de données. Comme vous le savez, l'équation d'une droite dans le plan a la forme $y = ax + b$. Nous cherchons une droite qui correspondrait le mieux à tous nos points de données, c'est-à-dire la meilleure équation. Nous nommons cette fonction h_θ. Nous cherchons donc une fonction $h_\theta(x_i) = \theta_0 + \theta_1(x_i)$.

Nous appelons la fonction h_θ la fonction *hypothèse*, puisque c'est la fonction supposée «expliquer» notre ensemble de données (pour s'y adapter). Nous allons commencer avec une certaine fonction d'hypothèse (probablement très mauvaise) et l'améliorer au fil du temps.

Afin d'améliorer l'hypothèse actuelle, nous devons mesurer sa précision. Nous utilisons une *fonction de coût* $J(\theta_0, \theta_1)$ telle que :

$$J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x_i) - y_i)^2$$

Cette fonction de coût nous indique à quel point notre fonction d'hypothèse est éloignée des valeurs réelles de notre ensemble d'apprentissage. Comme nous modifions θ_0 et θ_1, la valeur de $J(\theta_0, \theta_1)$ augmente ou diminue.

Le but est de modifier itérativement θ_0 et θ_1 afin de diminuer la valeur de $J(\theta_0, \theta_1)$ (c'est-à-dire, afin d'obtenir une meilleure fonction d'hypothèse). Pour nous guider, nous prenons la dérivée de $J(\theta_0, \theta_1)$, cela nous donne une «direction» dans laquelle aller pour réduire la valeur actuelle de J. Nous utiliserons cette dérivée pour réduire itérativement la valeur. À l'approche d'un minimum (local), la dérivée se rapprochera de 0 et nous cessons nos

déplacements[1].

Cette méthode s'appelle l'algorithme de la plus profonde descente («*gradient descent*»). Voici ses étapes :

$$\begin{aligned}&\text{jusqu'à la convergence : } \{\\ &\qquad \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1), \text{for } j = 0 \text{ and } j = 1\\ &\}\end{aligned}$$

Pour cet algorithme, on doit mettre à jour θ_0 et θ_1 simultanément. On utilise un paramètre α, le pas, qui contrôle le degré de correction apporté à θ_0 et θ_1 à chaque étape. Autrement dit, la «convergence» signifie que les nouvelles itérations de l'algorithme n'améliorent plus beaucoup la solution.

Il nous faut donc la dérivée partielle de J par rapport à θ_0 et θ_1 :

$$\frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) = \frac{2}{m} \sum_{i=1}^{m} (h_\theta(x_i) - y_i)$$

et

$$\frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) = \frac{2}{m} \sum_{i=1}^{m} ((h_\theta(x_i) - y_i) \times x_i)$$

Ainsi, l'algorithme du gradient pour un modèle de régression linéaire s'écrit comme suit :

$$\begin{aligned}&\text{jusqu'à la convergence : } \{\\ &\qquad \theta_0 := \theta_0 - \alpha \frac{2}{m} \sum_{i=1}^{m} (h_\theta(x_i) - y_i)\\ &\qquad \theta_1 := \theta_1 - \alpha \frac{2}{m} \sum_{i=1}^{m} ((h_\theta(x_i) - y_i) \times x_i)\\ &\}\end{aligned}$$

Implémentation

Pour la première partie, il y aura trois classes, **Display**, qui vous est fourni, **LinearRegression**, votre implémentation de l'algorithme, et **Assignment**, le programme principal.

La classe **Display** nous permettra de visualiser les droites et les points. Nous n'avez pas à comprendre tous les détails de cette classe, vous devez simplement savoir l'utiliser. Plus tard durant la session, nous discuterons l'implémentation des interfaces graphiques.

Si **linearRegression** est une variable référence qui désigne un objet de la classe **LinearRegression**, vous créez un objet de la classe **Display** comme suit :

```
Display graph;
graph = new Display(linearRegression);
```

Pour mettre à jour l'affichage, après une modification des paramètres du modèle linéaire, vous n'avez qu'à appeler la méthode **update()** de l'objet désigné par la variable référence **graph**.

```
graph.update();
```

Dans notre implémentation, nous allons choisir les valeurs initiales $\theta_0 = \theta_1 = 0$ pour toute régression linéaire, et notre méthode **gradientDescent** prendra deux arguments - un α comme décrit ci-dessus, ainsi qu'un nombre entier d'étapes exécuter (au lieu de «jusqu'à la convergence»).

[1] Notez que dans notre cas, la fonction d'erreur se trouve être une fonction *convexe* (une fonction parabolique) et a donc un seul minimum. Si ce n'était pas le cas, cette approche pourrait nous conduire à des minima locaux, ce qui n'est peut-être pas ce que nous voulons.

Question 1.1

Nous allons d'abord implémenter l'algorithme dans la classe **LinearRegression**. Le constructeur de cette classe reçoit en paramètre le nombre de points d'échantillonnage avec lesquels il travaillera. Ces échantillons sont ensuite fournis un par un en utilisant la méthode **addSample**. Une fois tous les exemples fournis, la méthode **gradientDescent** peut être appelée. Cette méthode a reçu deux paramètres : le pas α à utiliser, et le nombre d'itérations à effectuer lors de cet appel de la méthode (la méthode sera elle-même appelée plusieurs fois). La classe fournit un certain nombre d'autres méthodes qui sont nécessaires pour son propre usage ou par d'autres classes dont vous devrez fournir l'implémentation.

Afin de tester notre implémentation, nous utiliserons d'abord un jeu d'échantillons trivial composé de 1000 valeurs sur la droite $y = x$; nous allons sélectionner les valeurs de $x = 0$ à $x = 999$. La méthode **setLine** de la classe **Assignment** est utilisée pour ceci. En voici les détails :

- Créez un objet de la classe **LinearRegression** de la taille appropriée pour créer une droite composée des points (i, i) pour $0 \leq i \leq 999$ et créez un objet d'affichage correspondant.
- Itérer l'algorithme un total de 5 000 fois. Nous ferons cela en effectuant une descente de gradient 100 fois avec une petit valeur positive pour α, $\alpha = 0.000000003$ et **numOfSteps** mis à 100, bouclé 50 fois. À chaque itération de la boucle, vous devez mettre à jour le graphique et imprimer la valeur actuelle de la fonction d'hypothèse et de coût.

Si notre implémentation est correcte, la fonction d'hypothèse devrait tendre vers la droite $y = x$.
La sortie de votre console pour les premières itérations peut ressembler à ceci.

```
> java Assignment
setLine
Current hypothesis: 0.0+0.0x
Current cost: 332833.5
Press return to continue....

Current hypothesis: 0.0012974389329410189+0.8645276608877351x
Current cost: 6108.235980006332
Press return to continue....

Current hypothesis: 0.0014732015446818 95+0.9816455619876667x
Current cost: 112.09973445673143
Press return to continue....
```

Les figures 2, 3 et 4 montrent à quoi ressemble l'objet **Display** représentant le système après un nombre différent d'itérations. La droite en rouge est l'hypothèse actuelle. Comme vous pouvez le voir, initialement la droite rouge est $y = 0$, notre point de départ, et plus on itère, plus on se rapproche de $y = x$. Les classes et **JavaDoc** correspondants

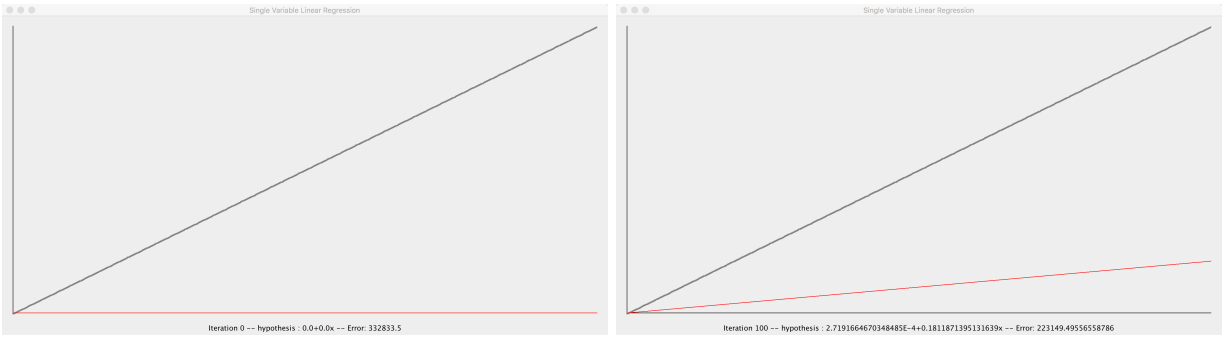

FIGURE 2 – Vue initiale et la vue après 100 itérations.

se trouvent ici :

- JavaDoc Documentation
- Assignment.java

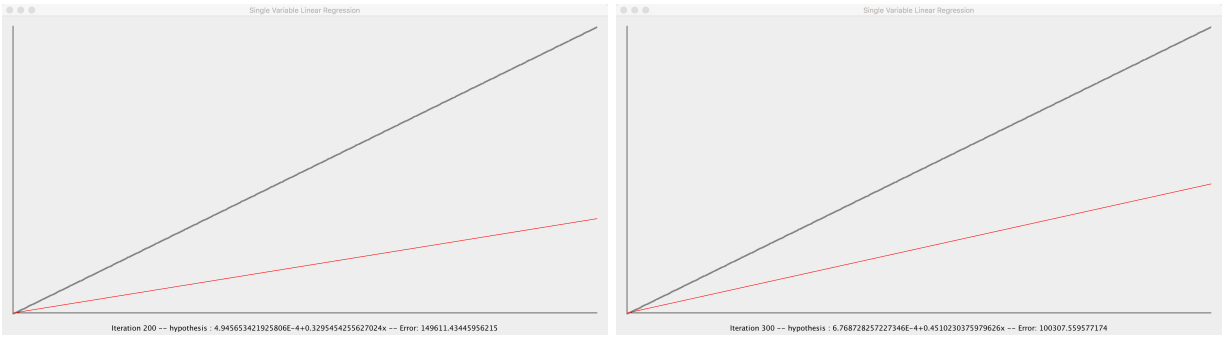

FIGURE 3 – Après 200 et 300 itérations.

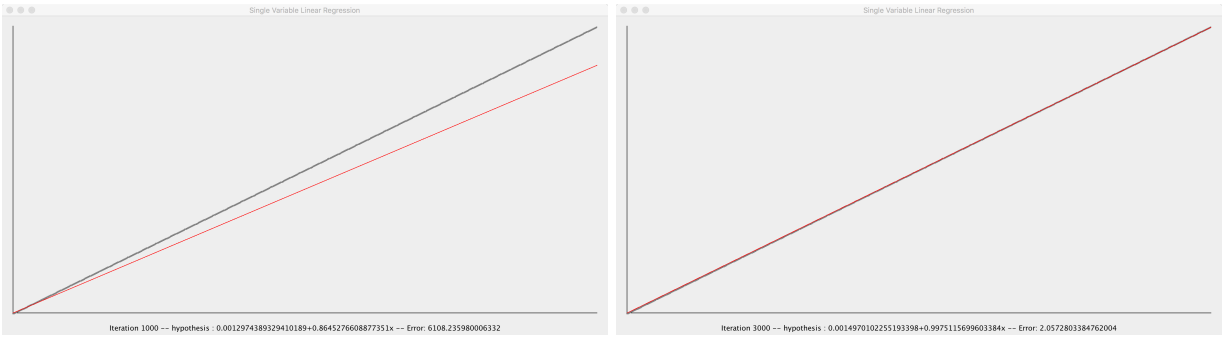

FIGURE 4 – Après 1000 et 3000 itérations.

- LinearRegression.java
- Display.java
- StudentInfo.java

Vous devez compléter les parties manquantes pour obtenir le résultat décris ci-dessus.

Question 1.2

Le test précédent était un début, mais pas particulièrement intéressant, car notre ensemble de points définit déjà une droite, donc nous allons générer une droite «aléatoire».

Pour générer des nombres aléatoires, vous allez créer un objet de la classe **java.util.Random** et appeler sa méthode **nextDouble**, qui retourne un nombre en réel dans l'intervalle 0 à 1. Vous devrez alors déterminer comment mettre à l'échelle ce nombre pour échantillonner à partir des intervalles spécifiés ci-dessous.

Le but est de fournir l'implémentation de la méthode **randomLine** pour la classe **Assignment**. Cette méthode doit faire ce qui suit.

- Créer un objet de la classe **LinearRegression** pour 500 points et un objet **Display** correspondant.
- Générer une droite aléatoire de la forme $y = ax + b$, où a est échantillonné aléatoirement à partir de l'intervalle $[-100, 100]$ et b est échantillonné aléatoirement à partir de l'intervalle $[-250, 250]$, et appelle **graph.setTarget(a, b)** pour dessiner sur l'écran - c'est la ligne que nous visons.
- Générez 500 points aléatoires satisfaisant aux conditions suivantes. La valeur x doit être échantillonnée aléatoirement à partir de $[-100, 300]$ et la valeur y correspondante est une valeur aléatoire ne dépassant pas 1000 par rapport au point réel de la ligne. C'est-à-dire, pour tout x, y devrait être échantillonné aléatoirement à partir de $[ax + b - 1000, ax + b + 1000]$. Le nombre 1000 ici est connu comme le *bruit*.
- Enfin, vous devez trouver un certain nombre d'itérations et une valeur pour α (indice : pensez petit, pensez positif) qui fonctionnent bien avec ces ensembles générés aléatoirement, puis effectuent la descente de la même manière qu'avant.

Les premières itérations d'un exemple peuvent ressembler à ceci.

```
> java Assignment
randomLine
Current hypothesis: 0.0+0.0x
Current cost: 1405392.4099890895
Aiming for: 123.72084222501928+6.347541365496268x_1
Press return to continue....

Current hypothesis: 0.044132282664095177+6.9297857925854x
Current cost: 338394.84501478786
Aiming for: 123.72084222501928+6.347541365496268x_1
Press return to continue....

Current hypothesis: 0.05875913751292656+7.020202855455091x
Current cost: 338210.92943203804
Aiming for: 123.72084222501928+6.347541365496268x_1

Press return to continue....
```

Les figures 5, 6 et 7 montrent le système après un nombre différent d'itérations sur un échantillon. Notez que la classe **Display** fournit une méthode **setTarget** qui peut être utilisée pour afficher sur l'écran la droite qui a été utilisée pour générer les données (la droite noire sur les figures).

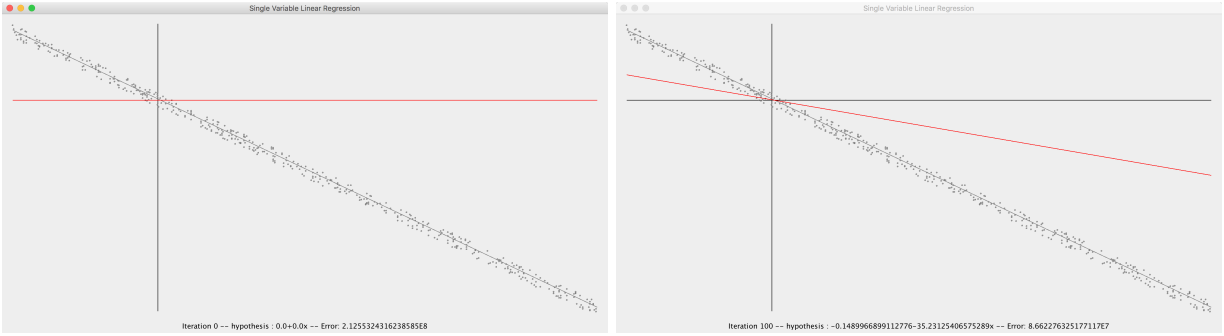

FIGURE 5 – Vue initiale et celle après 100 itérations.

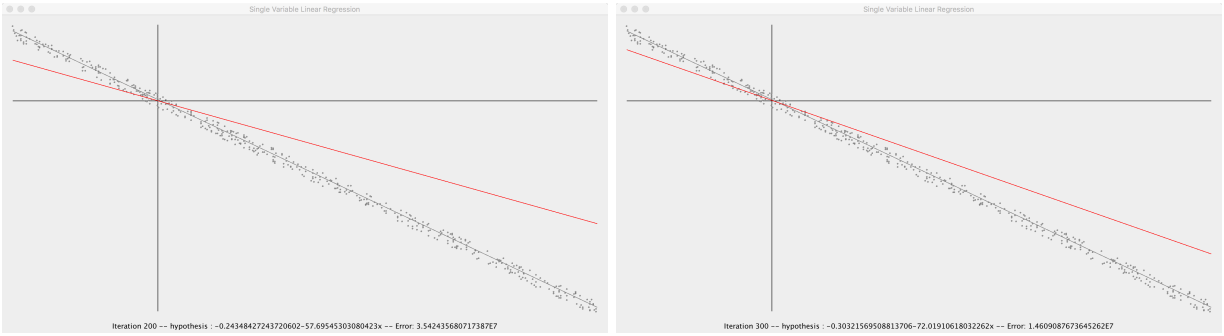

FIGURE 6 – Après 200 et 300 itérations.

Les classes et les **JavaDoc** se trouvent ici :

- JavaDoc Documentation
- Assignment.java
- LinearRegression.java

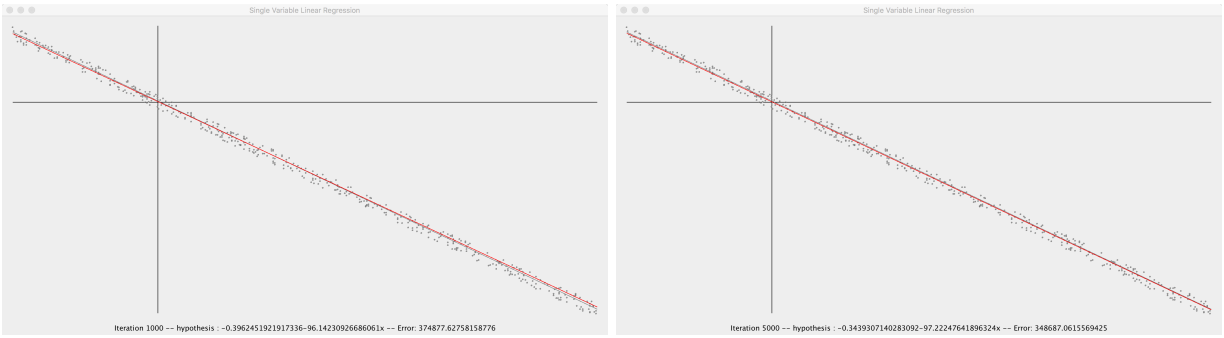

FIGURE 7 – Après 1000 et 5000 itérations.

- Display.java
- StudentInfo.java

Vous devez implémenter la méthode **randomLine** de la classe **Assignment**.

Modèle de régression linéaire généralisée ou encore multivariée

Si nous avons plus d'un attribut (dimension) (le «x» d'avant), nous pouvons utiliser la fonction d'hypothèse suivante. Supposons que nous ayons n attributs (dimensions) $x_1, x_2, \ldots, x_n$, la nouvelle fonction d'hypothèse est :

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \cdots + \theta_n x_n$$

Notations :

$$\begin{aligned} x_j^{(i)} &= \text{valeur de l'attributs } j \text{ du } i\text{-ème exemple de l'ensemble d'apprentissage} \\ x^{(i)} &= \text{l'entrée (attributs) du } i\text{-ème exemple de l'ensemble d'apprentissage} \\ m &= \text{le nombre d'exemples de l'ensemble d'apprentissage} \\ n &= \text{le nombre d'attributs} \end{aligned}$$

Pour plus de commodité, définissons $n+1$ «attributs» avec $x_0 = 1$ (c'est-à-dire $\forall i \in [1, \ldots, m], x_0^{(i)} = 1$). La fonction d'hypothèse $h_\theta(x)$ peut maintenant être réécrite comme suite :

$$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \cdots + \theta_n x_n$$

Algorithme du gradient pour un modèle de régression linéaire généralisée

Voici la nouvelle fonction de coûts :

$$J(\theta_0, \theta_, \ldots, \theta_n) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

L'algorithme du gradient devient alors :

jusqu'à la convergence : {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1, \ldots, \theta_n)$$

for $j \in [0, \ldots, n]$ (mise à jour simultanément)

}

et donc

jusqu'à la convergence : {

$$\begin{aligned} \theta_0 &:= \theta_0 - \alpha \frac{2}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_0^{(i)} \\ \theta_1 &:= \theta_1 - \alpha \frac{2}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_1^{(i)} \\ \theta_2 &:= \theta_2 - \alpha \frac{2}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_2^{(i)} \\ &\cdots \end{aligned}$$

}

et finalement :

jusqu'à la convergence : {

$$\theta_j := \theta_j - \alpha \frac{2}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

for $j \in [0, \ldots, n]$ (mise à jour simultanément)

}

Implémentation

Nous généraliserons l'implémentation de la question 1 pour calculer une régression linéaire multivariée. Cependant, nous n'utiliserons plus d'objets **Display**, pour des raisons de dimensionnalité. Vous devrez modifier votre classe **LinearRegression** pour fonctionner avec des entrées qui sont des vecteurs de doubles (représentés par des tableaux), au lieu de simples nombres. Vous devrez également mettre à jour la méthode de descente de gradient elle-même, ainsi que toutes les autres méthodes d'assistance nécessaires pour tenir compte des entrées de dimension supérieure. Nous procéderons ensuite aux trois tests ci-dessous dans la classe **Assignment**.

Question 2.1

Nous validons le code avec un plan fixe (c'est-à-dire une entrée bidimensionnelle ou deux attributs). Votre méthode **setPlane** doit procéder comme suit.

1. Créez un objet de la classe **LinearRegression** avec 2 attributs et 2000 points d'échantillonnage.
2. Ajoutez les points $((x, 2x), 5x)$ et $((2x, x), 4x)$ pour $0 \leq x \leq 999$. Notez que ces points satisfont tous l'équation. $z = x + 2y$
3. Comme précédemment, exécutez la descente avec $\alpha = 0.000000003$ et le nombre d'étapes définies sur 1000, soit un total de 10 fois, et imprimez l'hypothèse et le coût à chaque itération.

Voici les classes et les fichiers **JavaDoc**.

- JavaDoc Documentation
- Assignment.java
- LinearRegression.java
- StudentInfo.java

Question 2.2

Comme dans la question 1, nous allons maintenant passer à un plan avec des points aléatoires. Votre méthode **randomPlane** devrait faire ce qui suit.

- Créez un objet de la classe **LinearRegression** avec 2 attributs et 5000 points.
- Échantillonnez les coefficients aléatoires $a, b, c \in [-100, 100]$. Le plan que nous visons est $x_3 = c + ax_1 + bx_2$.
- Maintenant, ajoutez 5000 points qui satisfont aux conditions suivantes. L'entrée (x_1, x_2) doit être échantillonnée avec $x_i \in [50, 4000]$. Nous avons mis le bruit cette fois à 20, de sorte que pour tout (x_1, x_2), nous devrions avoir $z = ax_1 + bx_2 + c + \delta$, où δ est échantillonné aléatoirement à partir de $[-20, 20]$.
- Comme précédemment, choisissez des valeurs raisonnables pour α et le nombre d'étapes, et imprimez l'hypothèse actuelle, le coût actuel et le plan visé à chaque itération.

Voici les classes et les fichiers **JavaDoc**.

- JavaDoc Documentation
- Assignment.java
- LinearRegression.java
- StudentInfo.java

Question 2.3

Nous généralisons maintenant la partie précédente pour permettre une équation aléatoire pour une dimension donnée. La méthode **randomDimension** devrait prendre un entier n comme argument, en spécifiant la dimension requise n du vecteur d'entrée, et le programme principal testera ceci avec $n = 50$. Implémentez la méthode pour effectuer les opérations suivantes.

- Créez un objet de la classe **LinearRegression** avec n attributs et 5000 points. Un vecteur d'entrée général aura la forme $(x_1, x_2, \ldots, x_n)$.

- Nous générons une équation aléatoire comme suit. Nous échantillonnons au hasard les coefficients $t_0, t_1, \ldots, t_n$ à partir de $[-100, 100]$. L'équation que nous modélisons est

$$r = t_0 + \sum_{i=1}^{n} t_i x_i = t_0 + t_1 x_1 + t_2 x_2 + \ldots + t_n x_n. \quad (1)$$

- Ajoutez 5000 points qui satisfont aux conditions suivantes. Pour chaque vecteur d'entrée $(x_1, \ldots, x_n)$, échantillonnez x_i aléatoirement à partir de $[50, 4000]$. Alors, que le résultat r soit comme dans l'équation 1, plus ou moins un bruit de 20. La valeur ajoutée est alors $((x_1, \ldots, x_n), r)$.
- Encore une fois, choisissez des valeurs raisonnables pour α et le nombre d'étapes, et imprimez l'hypothèse actuelle, le coût actuel et l'équation visée à chaque itération.

Voici les classes et les fichiers **JavaDoc**.

- JavaDoc Documentation
- Assignment.java
- LinearRegression.java
- StudentInfo.java

Intégrité dans les études

Cette partie du devoir a pour but de sensibiliser les étudiants face au problème de fraude scolaire (plagiat). Lisez les deux documents qui suivent.

- http://www.uottawa.ca/administration-et-gouvernance/reglement-scolaire-14-autres-informations-importa
- https://www.uottawa.ca/vice-recteur-etudes/integrite-etudes
- http://web5.uottawa.ca/mcs-smc/integritedanslesetudes/documents/2011/integrite-dans-les-etudes-guide-
pdf
- http://www.uottawa.ca/vice-recteur-etudes/lintegrite-etudes/ressources-lintention-etudiants

Les règlements de l'université seront appliqués pour tout cas de plagiat.

En soumettant ce devoir, je certifie que :

1. **il s'agit d'un travail original ;**
2. **j'ai lu les documents ci-haut ;**
3. **je comprends les conséquences de la fraude scolaire.**

Avertissements

- Les soumissions qui ne sont pas conformes aux exigences ne fonctionneront pas avec les outils que nous utilisons pour valider vos classes et en conséquence ne seront pas corrigées.
- Nous utilisons un outil informatique pour détecter les cas de plagiat. Les soumissions de toutes les sections (anglaises et française) sont comparées. Les soumissions identifiées par cet outil recevront la note 0.
- Vous devez vous assurer que Brightspace a bien reçu votre soumission. Vous ne pourrez pas soumettre des documents après l'échéance.
- Les soumissions en retard ne sont pas acceptées.

Respect des règles et consignes

Veuillez suivre les consignes que vous trouverez sur la page des consignes aux devoirs. Tous les devoirs doivent être soumis à l'aide de `https://uottawa.brightspace.com`. Vous devez préférablement faire le travail en équipe de deux, mais vous pouvez aussi faire le travail seul. Vous devez au préalable vous inscrire à groupe d'étudiants sur Brightspace.

Fichiers

Vous devez remettre un fichier zip. Le répertoire principal a pour nom **a1_3000000_30000001**, où 300 000 et 300 001 sont les numéros d'étudiants des deux membres du groupe. Le nom du répertoire débute par la lettre minuscule **a** suivie du numéro du devoir, **1**. Les séparateurs sont les symboles soulignés («*underscore*») et non le tiret. Il n'y a pas d'espace dans le nom du répertoire. Si vous travaillez seul, répétez votre numéro d'étudiant deux fois. Le répertoire doit contenir les fichiers suivants :

- Un fichier texte README.txt qui contient les noms des deux partenaires pour ce devoir, leurs numéros étudiants, et une brève description du devoir (une ou deux lignes).
- Le code source de toutes vos classes. Chaque sous-répertoire doit contenir tous les fichiers nécessaires pour la compilation et l'exécution du devoir. En particulier, nous devons être en mesure de compiler et exécuter votre application sans avoir à ajouter des fichiers ou apporter des correctifs.
- Le répertoire **doc** et ses documents **JavaDoc**.
- **StudentInfo.java**, correctement complété et appelé à partir de votre programme principal.

Modifié le : 22 janvier 2018