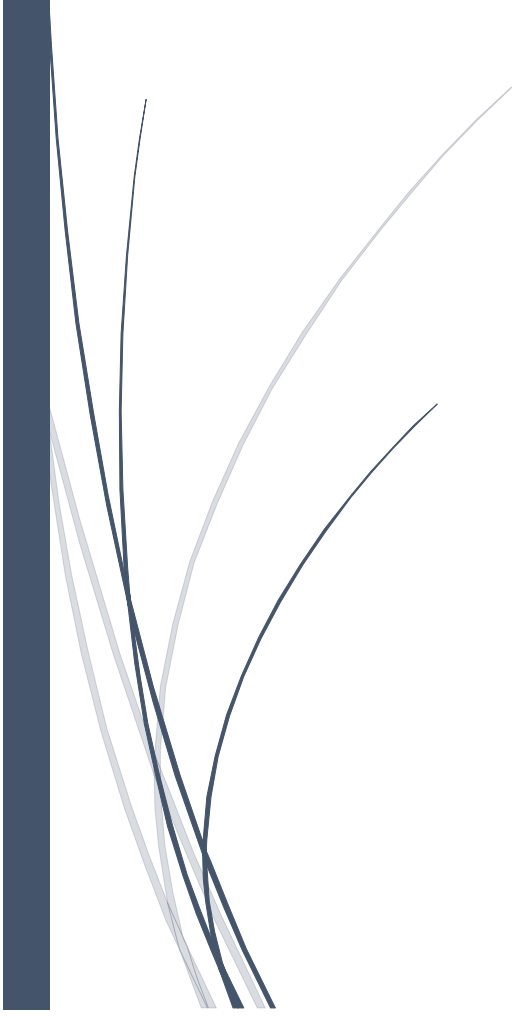
29/03/2023

Machine Learning

TP01



Ramiqi Andi

MASTER IS : HES-SO GENÈVE

Introduction

La décomposition QR est une technique algébrique largement utilisée pour résoudre des systèmes linéaires, elle est particulièrement très utile pour résoudre des problèmes de régression linéaire multiple où il y a plusieurs solutions possibles. En utilisant la décomposition QR, nous pouvons obtenir une solution optimale avec une complexité algorithmique plus faible que d'autres méthodes, telles que la méthode des moindres carrés ordinaires.

Dans ce travail pratique, l'utilisation de python offre de nombreuses bibliothèques pour effectuer facilement la décomposition QR et résoudre des systèmes linéaires, telles que NumPy et SciPy.

Ce rapport comprend la partie analytique des exercices. Le code sera fourni dans un Jupiter notebook[1] comprenant les différentes étapes, méthodes et exemples utilisées pour démontrer les résultats obtenus.

Linear independent attributes :

Ce premier exercice consistera à générer une matrice X de type « tall » de façon aléatoire, un vecteur w de manière aléatoire, et comparer le résultat de la fonction Xw = y avec le produit matriciel de X et w et la décomposition QR.

QR Decomposition :

La décomposition QR d'une matrice X prends la forme X = QR où Q est une matrice orthogonale et R une matrice triangulaire supérieure pour autant que les colonnes de X soient linéairement indépendantes.

Développement de la formule :

$$Xw = y \rightarrow QRw = y \rightarrow Q^{-1}QRw = Q^{-1}y \rightarrow Rw = Q^{-1}y \rightarrow Rw = Q^{T}y$$

Il est ensuite possible, grâce à la méthode Gram-Schmidt, de déterminer QR pour ainsi trouver un vecteur w à la résolution de l'équation Xw = y. Dans un premier temps, selon les dimensions de la matrice X [n,d], il est possible de déterminer les dimensions de la matrice Q et R. R étant une matrice carré, celle-ci aura pour taille [d,d]. La matrice Q aura ainsi les dimensions [n,d]

QR Decomposition

$$\begin{bmatrix} 2 & 3 \\ 2 & 4 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} \star & \star \\ \star & \star \\ \star & \star \end{bmatrix} \begin{bmatrix} \star & \star \\ 0 & \star \end{bmatrix}$$

Dans cet exemple, avec une matrice X de taille (3,2), la matrice Q aura les mêmes dimensions et R sera une matrice carré de taille (2,2)

[1] Voir en annexe : TP01.ipynb

Pour la résolution du calcul, il sera possible de réaliser les projections de X sur Q pour ainsi trouver les vecteurs Qi représentant les colonnes de la matrice Q :

$$Q_i = \frac{\widetilde{q_i}}{||\widetilde{q_i}||}$$

$$\widetilde{q_i} = a_i - \sum_{k=1}^{i-1} ({q_k}^T * a_i) * q_k$$

$$Pour\ la\ matrice\ R : \ selon\ les\ positions, le\ calcul\ sera\ diff\acute{e}rent$$
$$i > j = \ 0\ (repr\acute{e}sentent\ les\ 0\ de\ la\ matrice\ triangulaire\ sup\acute{e}rieure$$
$$i = j = R_{ii} = \left|\left|\widetilde{q_i}\right|\right|$$
$$i < j = R_{ij} = {q_i}^T a_j$$

Dans le premier exercice, lorsque nous comparons la solution définie avec celle produite par la décomposition qr (np.linalg.qr), celle-ci nous retourne bien le même résultat et les valeurs sont bien identiques.

Lorsqu'il est question d'utiliser la matrice pseudo-inverse, celle-ci est réalisable via la factorisation QR. En supposant que la matrice ait des colonnes linéairement indépendantes et que celle-ci soit « tall », il est possible de décomposer le résultat via QR de la façon suivante pour la fonction :

Détérminer Ax = b avec A = QR

$$A^T A = (QR)^T (QR) = R^T Q^T QR = R^T R$$

$$A^\dagger = (A^T A)^{-1} A^T = (R^T R)^{-1} (QR)^T = R^{-1} R^{-T} R^T Q^T = R^{-1} Q^T$$

$$\mathbf{x} = \mathbf{A}^\dagger \mathbf{b}$$

Lorsque nous appliquons la méthode en code en utilisant l'inverse de R et la transposé de Q, nous arrivons finalement aussi au même résultat. Le vecteur w est aussi identique dans ce cas d'utilisation.

Dans le troisième cas d'utilisation, nous allons cette fois ajouter du « bruit » aléatoire au vecteur y et comparer le résultat. Pour cela, il faudra modifier le vecteur y de sorte à perturber légèrement le résultat sur une loi normale centrée à 0 avec une variance de 1.

Le procédé pour déterminer le vecteur w en fonction de la matrice X et du vecteur « bruité » y reste le même que pour la première partie. Les résultats ne sont néanmoins pas identiques, mais nous pouvons observer qu'ils restent très proches. Cela montre que la décomposition QR a bel et bien été mis en place pour calculer le « w » avec un « y » légèrement bruité.

```
Les deux vecteurs ne sont pas identiques
Solution trouvée par QR :  [ 0.62  1.45  1.03 -0.04  1.6   0.63 -0.28 -1.25  0.58  0.4 ]
w généré aléatoirement :  [ 0.51  1.43  0.94  0.12  1.5   0.45 -0.25 -1.22  0.4   0.29]
```

Linear dependencies in the attributes

Dans cette deuxième partie, l'exercice consistera à répéter les éléments de la première partie en modifiant la matrice X de départ. Cette nouvelle matrice X' aura comme dimension les valeurs de X ainsi que l'ajout de 2 colonnes supplémentaires correspondants aux 2 premières colonnes de X. Cette première étape sera réalisable grâce à « numpy.concatenate »

Lorsque nous procédons au calcul de w', dans un premier temps avec la solution selon X' et y, puis à l'aide de la décomposition QR, les solutions suivantes sont trouvées :

```
Les deux vecteurs ne sont pas identiques
Solution trouvée par QR pour la matrice X':  [-0.18  0.01 -0.21  0.08 -1.11 -2.35  0.32  0.58 -0.06  0.5  -0.07  0.25]
w généré aléatoirement selon X'w = y:        [ 0.3   0.4  -0.21  0.08 -1.11 -2.35  0.32  0.58 -0.06  0.5  -0.55 -0.13]
```

Comme nous pouvons le constater, la solution n'est pas identique, mais celle-ci diffère seulement au niveau des colonnes dont les « features » sont linéarement dépendantes (Colonnes 1 et 2, ainsi que les 2 dernières colonnes)

Dans ce cas particulier, la solution que nous avons trouvée en utilisant QR n'est probablement pas unique puisque toutes les colonnes ne sont pas indépendantes dans X. Cela signifie qu'il peut y avoir plus d'un ensemble de poids w qui satisfont Xw = y. Cependant, la solution que nous avons trouvée est toujours la solution des moindres carrés, ce qui signifie qu'elle a la plus petite norme euclidienne parmi toutes les solutions qui satisfont Xw = y.

La solution que nous avons trouvée ici en utilisant QR devrait être similaire à celle que nous avons trouvée dans le cas d'attributs linéairement indépendants, puisque les colonnes dépendantes linéairement ajoutées n'affectent pas de manière significative la solution. Cependant, les poids peuvent ne pas être exactement les mêmes en raison d'erreurs numériques dans la décomposition QR et la résolution du système d'équations.

Visualising the $|\boldsymbol{Xw} - \boldsymbol{y}|^2$

En utilisant la méthode des moindres carrés , il est possible de trouver une solution pour l'interpolation d'un polynôme, même lorsque la matrice des coefficients est singulière. En utilisant cette méthode, nous allons chercher les coefficients du polynôme qui fournit l'erreur quadratique minimale lors de la comparaison de la courbe polynomiale à des points de données.

Dans un premier temps, nous allons répéter les exercices précédents et générer une matrice X de taille (100,2) aléatoirement ainsi qu'un vecteur w de taille (2,1) aléatoirement. Nous pouvons ensuite générer le y comme une fonction linéaire de X et w avec un bruit gaussien ajouté.

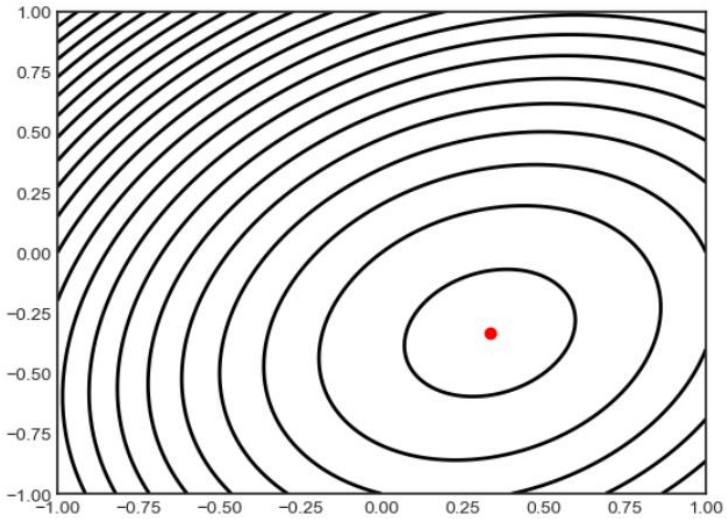
Pour résoudre le problème des moindres carrés, numpy.linalg fourni une fonction appelée lstsq() afin de résoudre le système « Xw = y » et détérminer le poids « w ».

Lorsque nous comparons la solution avec celle trouvée par la fonction, nous avons à disposition les résultats suivants :

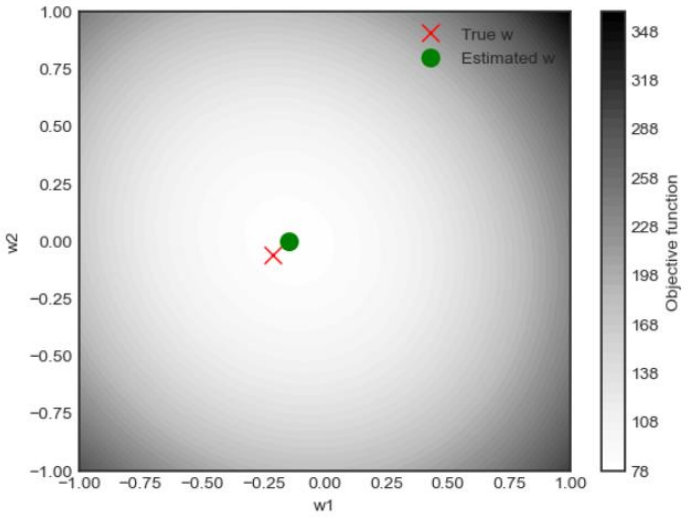
```
Solution générée aléatoirement :  [-0.61 -1.39]
Solution générée à l'aide de la fonction lstsq() :  [-0.67 -1.34]
```

Comme nous pouvons l'observer, la solution n'est pas exactement la même, mais celle-ci s'en rapproche le plus possible.

Dans un deuxième temps, nous allons générer une visualisation en 2D de la solution optimale afin d'observer et déterminer les valeurs de w permettant de minimiser la fonction objective. Pour visualiser la fonction objective dans l'espace w1 et w2, nous allons créer une grille de valeurs pour w1 et w2 et calculer la fonction objective pour chaque point de la grille. Nous utiliserons ensuite la fonction « contour » et « ax.contourf » de matplotlib pour créer deux graphiques permettant d'observer les valeurs de la fonction objectif.



Le premier plot utilise la fonction de contour « plt.contour » pour dessiner les lignes de niveau de la fonction LeastSquareFunction2(X, Y). Chaque ligne représente une valeur constante de la fonction, et l'argument « np.arange(0,100,1) » spécifie les valeurs de la fonction pour lesquelles les lignes de niveau doivent être dessinées.



Le deuxième plot utilise la fonction de contour rempli « ax.contourf » pour dessiner les mêmes lignes de niveau, mais cette fois-ci avec une couleur remplie pour chaque région de niveau. La fonction « plt.colorbar » ajoute une barre de couleur qui permet d'interpréter la valeur de la fonction pour chaque région.

Le premier plot est utile pour visualiser les contours de la fonction et la position de l'optimum global (point rouge), tandis que le deuxième plot est plus approprié pour voir la répartition des valeurs de la fonction sur la grille.