Machine Learning TP1: Régression linéaire

Melvin DUBEE - Tanguy ROUDAUT FIPASE 24

16 octobre 2023

Vous pouvez trouver nos codes dans les dossiers "code" et "code2" et les résultats dans le dossier "output" qui se trouve à la racine de l'archive.

1 Régression linéaire avec une variable

1.1 Affichage des données

Une première fonction *plotData()*, qui permet d'afficher le profit du food truck en fonction de sa population sous forme de point.
L'objectif de cette partie sera de prédire le profit d'un food truck dans une nouvelle ville grâce à une descente de gradient en comprenant sa méthodologie.

```
1 def plotData(X,y):
2     fig = plt.figure()
3     plt.plot(X, y, 'rx')
4     plt.grid(True)
5     plt.ylabel("Profit in $10,000s")
6     plt.xlabel("Population of City in 10,000s")
7     fig.show()
```

Listing 1 – Fonction plotData

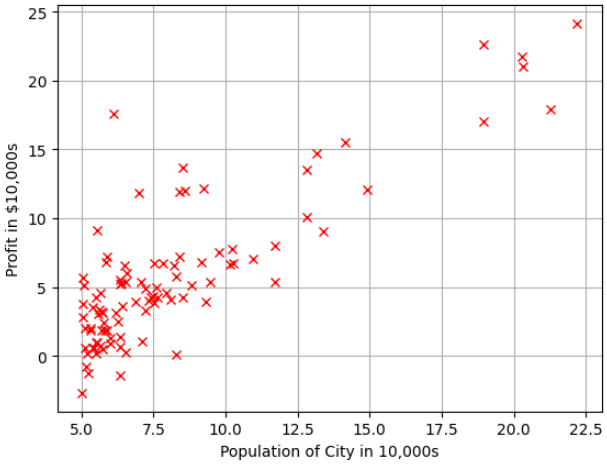

FIGURE 1 – Diagramme de dispersion des données d'entraînement

1.2 Descente de gradient

Le modèle de régression linéaire est représenté par l'équation 1. Cette équation nous permet d'obtenir une prédiction en fonction d'une entrée x_1 et de θ.

$$h_\theta(x) = x^T\theta = \theta_0 + \theta_1 x_1 \tag{1}$$

Pour que cette prédiction soit optimale, il est important de déterminer correctement les paramètres de notre modèle : θ. Pour cela, nous devons réaliser deux étapes : Le calcul du coût $J(\theta)$ et une descente de gradient.

1.2.1 Calcul du coût $J(\theta)$

Le calcul du coût $J(\theta)$ permet de mesurer la qualité de la prédiction, si le coût est faible alors notre prédiction est proche des valeurs réelles et inversement si le coût est important.

$$J(\theta) = \underbrace{\frac{1}{2m}\sum_{i=0}^{m-1}}_{(b)}\underbrace{(h_\theta(x^{(i)}) - y^{(i)})}_{(a)}^2 \tag{2}$$

(a) **Différence entre la prédiction et la valeur réelle**, ce qui revient à déterminer l'erreur de la prédiction. On met le résultat au carré pour obtenir que des erreurs positives.
(b) **Moyenne des erreurs**, ce qui nous permet d'obtenir le coût $J(\theta)$

On remarque ici avec l'équation 3 qui utilise l'équation 1, que la seule valeur qui puisse influencer notre coût est θ. Effectivement, les valeurs restantes : x, y et m ; sont les valeurs de notre problème qui sont déterminées et non modifiables.

Mise en application

```
1  def computeCost(X, y, theta):
2      m = y.size
3      J = 0
4
5      for i in range(0, m):
6          J += (1/(2*m)) * (X[i].T @ theta -
           ↪  y[i])**2
7
8      return J
```

Listing 2 – Fonction computeCost

```
With theta = [0 ; 0] Cost computed = 32.0727
Expected cost value (approx) 32.07

 --------------------------

With theta = [-1 ; 2] Cost computed = 54.242455
Expected cost value (approx) 54.24
```

Listing 3 – Output fonction computeCost

Suite à la mise en application, on constate effectivement l'influence de θ sur notre coût, on en déduit facilement qu'il est nécessaire de trouver la bonne valeur de θ pour minimiser ce coût. C'est le rôle de la descente de gradient.

1.2.2 Descente de gradient

La descente de gradient permet de minimiser le coût et donc d'obtenir les bonnes valeurs de theta pour réaliser une prédiction optimale.

$$\theta_j := \theta_j - \alpha \underbrace{\frac{1}{m}\sum_{i=0}^{m-1}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}}_{\text{Gradient du coût } J(\theta)} \tag{3}$$

Pour mener à bien la descente de gradient, on réalise à nouveau la moyenne des erreurs de prédiction, mais cette fois-ci on la multiplie par un pas d'apprentissage α. Ce pas permet de déterminer le taux d'apprentissage du modèle, plus il est grand plus l'apprentissage sera rapide, mais si ce taux est trop important alors la descente de gradient peut diverger d'un minimum. Il est donc important de choisir correctement ce pas. Ce gradient de la fonction de coût est ensuite soustrait aux θ_j simultanément.
Il faut réaliser cette étape un certain nombre d'itérations pour converger vers un minimum.

Mise en application

```python
def gradientDescent(X, y, theta, alpha, num_iters):
    m = y.size  # number of training examples
    n = theta.size # number of parameters
    cost_history = np.zeros(num_iters) # cost over iters
    theta_history = np.zeros((n,num_iters)) # theta over iters

    for n_iter in range(num_iters):
        for j in range(n):
            res = 0
            for i in range(m):
                res += (X[i].T @ theta - y[i]) * X[i][j]

            theta[j] = theta[j] - alpha * (1/m) * res

        cost_history[n_iter] = computeCost(X, y, theta)
        theta_history[:,n_iter] = theta.reshape((2,))

    return theta, cost_history, theta_history
```

Listing 4 – Fonction gradientDescent

Après application de la fonction 4 on obtient les résultats du listing 5 et figure 2, ce qui nous permet de conclure que nos valeurs de θ sont correctes et que l'on a un modèle satisfaisant pour réaliser les prédictions optimales suivantes :

1. *For population = 35,000, we predict a profit of 4483.9858*
2. *For population = 70,000, we predict a profit of 45328.6063*

```
Theta found by gradient descent:
-3.6360634754795016 1.1669891581648786
Expected theta values (approx)
 -3.6303  1.1664
```

Listing 5 – Output fonction gradientDescent

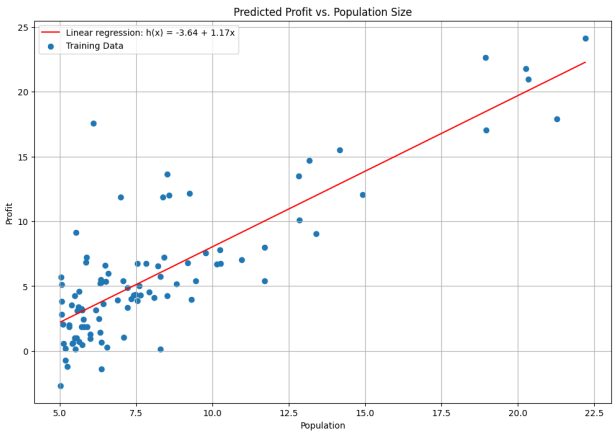

FIGURE 2 – Profit prédit vs population

1.3 Visualisation de $J(\theta)$

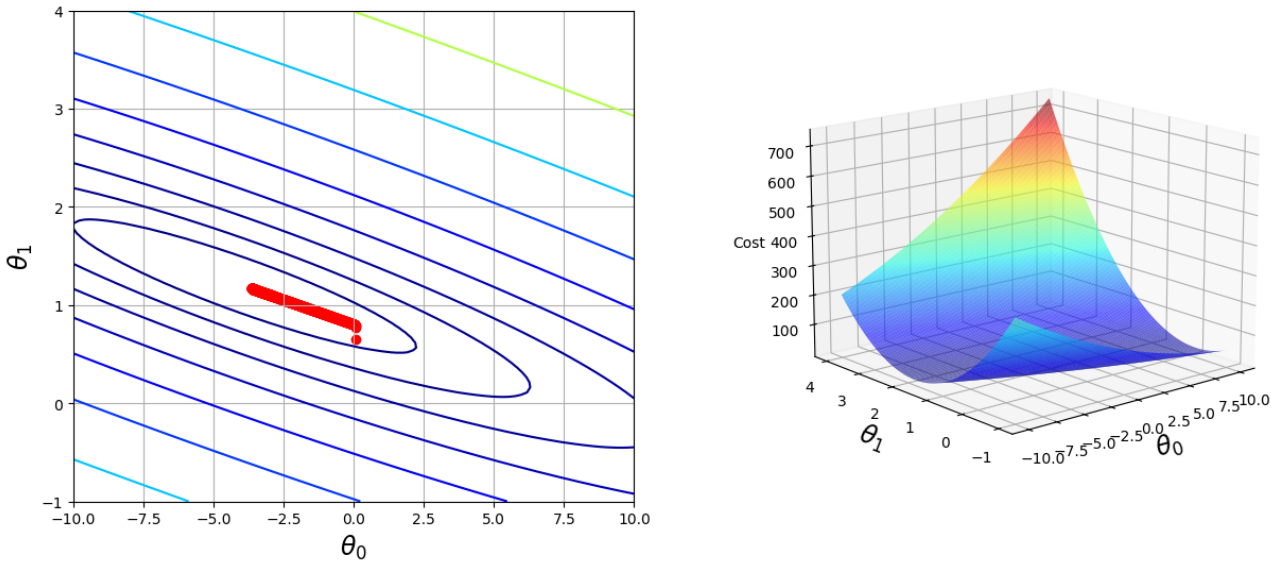

FIGURE 3 – Visualisation de $J(\theta)$

Ce graphique nous permet de visualiser plus facilement de comment le coût $J(\theta)$ évolue, on remarque que les valeurs de θ sont modifiées au fur et à mesure pour converger vers le centre de l'éclipse qui représente le minimum optimal.

2 Régression linéaire avec plusieurs variables

2.1 Normalisation des caractéristiques

Dans ce problème, nous avons des caractéristiques qui ont des échelles radicalement différentes *(exemple 2100 sq-ft pour 3 chambres)*. Il est intéressant de normaliser les données pour les ramener à une échelle commune, ce qui permet d'interpréter plus facilement les résultats et rendre les calculs plus stables et rapides. Grâce à la normalisation, on peut être sûr que chaque caractéristique contribue équitablement à la prédiction du modèle, indépendamment de son échelle initiale.

Pour réaliser la normalisation, nous pouvons établir les formules suivantes :

$$X_{norm} = \frac{X - \mu}{\sigma} \quad \Big| \quad \mu = \frac{1}{m}\sum_{i=0}^{m-1} x_j^{(i)} \quad \Big| \quad \sigma = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(x_j^{(i)} - \mu\right)^2}$$

— μ La moyenne de chaques caractéristiques
— σ L'écart type de chaques caractéristiques
— X_{norm} Caractéristiques normalisées
— $x_j^{(i)}$ La caractéristique de la ligne j à la colonne i

Mise en application

```python
def featureNormalize(X):
    mu = np.mean(X, axis=0)
    sigma = np.std(X, axis=0)
    X_norm = (X - mu) / sigma

    return X_norm, mu, sigma
```

$$\mu = [2000.68085106 \quad 3.17021277]$$
$$\sigma = [7.86202619e+02 \quad 7.52842809e-01]$$

Listing 6 – Fonction featureNormalize

2.2 Descente de gradient

La descente de gradient de ce second problème suit la même méthodologie. La seule différence est qu'ici on à plusieurs caractéristiques.
Nos fonctions précédentes sont adaptées pour un calcul de coût et une descente de gradient à plusieurs caractéristiques, nous pouvons réutiliser les mêmes fonctions.

```python
def computeCostMulti(X, y, theta):
   m = y.size
   J = 0

   for i in range(0, m):
      J += (1/(2*m)) * (X[i].T @ theta - y[i])**2

   return J
```

Listing 7 – Fonction computeCostMulti

```python
def gradientDescentMulti(X, y, theta, alpha, num_iters):
    m = y.size  # number of training examples
    n = theta.size # number of parameters
    cost_history = np.zeros(num_iters)
    theta_history = np.zeros((n,num_iters))

    for n_iter in range(num_iters):
        for j in range(n):
            res = 0
            for i in range(m):
                res += (X[i].T @ theta - y[i]) * X[i][j]

            theta[j] = theta[j] - alpha * (1/m) * res

        cost_history[n_iter] = computeCostMulti(X, y, theta)
        theta_history[:,n_iter] = theta.reshape((n,))

    return theta, cost_history, theta_history
```

Listing 8 – Fonction gradientDescentMulti

Suite à notre descente de gradient, on constate sur la figure 4 que le coût $J(\theta)$ converge vers un minimum. On peut donc considérer que nos valeurs de θ sont optimales pour réaliser une prédiction proche de la réalité.

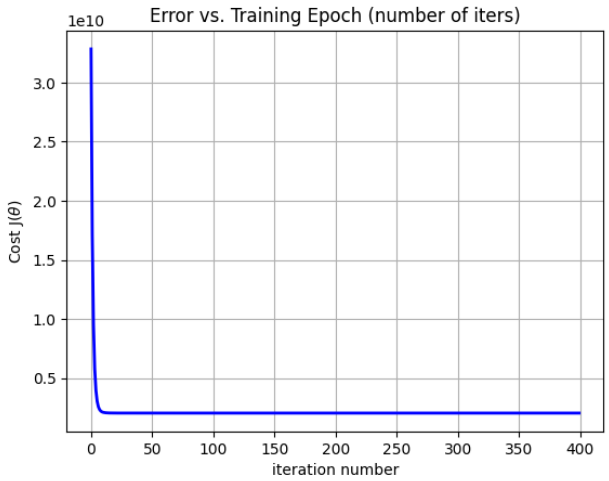

FIGURE 4 – Convergence de $J(\theta)$ en fonction des itérations avec $\alpha = 0.3$

$$\theta_0 = 340412.65957447$$
$$\theta_1 = 109447.79$$
$$\theta_2 = -6578.35$$

Prédiction pour une maison de 1650 *sq-ft* et 3 chambres de 293081.46$

2.2.1 Sélection du taux d'apprentissage

Comme expliqué précédemment il est important de sélectionner correctement le taux d'apprentissage, si celui-ci est trop important alors la descente de gradient peut diverger d'un minimum, s'il est trop petit alors le processus peut échouer dû à des valeurs trop importantes.

Trop petit, $\alpha = 0.001$

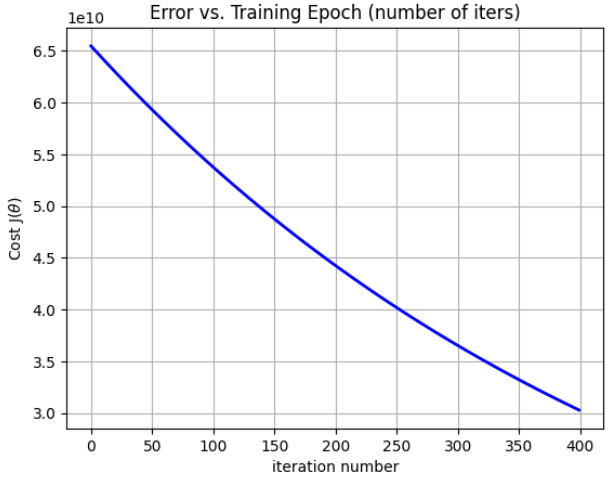

FIGURE 5 – Convergence de $J(\theta)$ en fonction des itérations avec $\alpha = 0.001$

$$\theta_0 = 112272.89$$
$$\theta_1 = 33255.72$$
$$\theta_2 = 14509.19$$

Prédiction pour une maison de 1650 *sq-ft* et 3 chambres de 94158.94$

Dans ce cas le taux d'apprentissage choisi est trop faible, on le constate grâce à l'allure de la courbe, mais également avec les résultats obtenus. Un résultat cohérent se rapproche de l'allure de la figure 4.

2.3 Exercice facultatif : Équations normales

La régression linéaire par équations normales donne un résultat similaire à la descente de gradient, mais ce n'est pas un algorithme itératif. L'équation 4 nous permet d'obtenir directement une valeur optimale de θ.

$$\theta = (X^T X)^{-1} X^T y \tag{4}$$

Mise en application

```python
def normalEqn(X,y):
    theta = np.linalg.inv((X.T @ X)) @ X.T @ y

    return theta
```

Listing 9 – Fonction normalEqn

Prédiction pour une maison de 1650 *sq-ft* et 3 chambres de 293081.46$ avec la méthode des équations normales.
Le résultat est similaire de celui obtenu par la méthode de la descente de gradient avec une valeur de $\alpha = 0.3$.

3 Questions

1. **Définissez les termes :**
   - **Approches supervisées**
     L'approche supervisée est utilisée quand on a un certains nombres de données connus et correctes. C'est données sont utilisées pour entrainer le modèle pour que celui-ci puisse nous donner une prévision proche de la réalité. Nous avons donc appliqué une approche supervisée dans ce TP.
   - **Approches non supervisées**
     Il s'agit de l'inverse de l'approche supervisée, ici on lui demande un résultat à l'aide de données aléatoires.
   - **Régression**
     Utilisé pour déterminer une prédiction optimale à l'aide d'une relation linéaire entre une variable dépendante et plusieurs variables indépendantes.
   - **Classification**
     Utilisé pour déterminer à qu'elle classe appartient une observation à partir de précédentes connus.
2. **Représenter en un schéma général, les processus d'apprentissage et de prédiction ?**

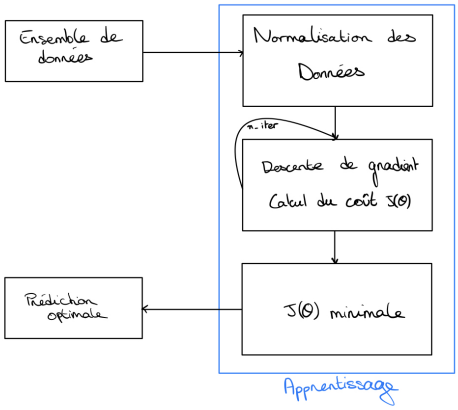

FIGURE 6 – Schéma général d'apprentissage

3. **Comment fonctionne l'apprentissage ? Par quels moyens ? A quoi sert la fonction de coût ? Comment est-résolu le problème ? Connaissez vous d'autres moyens de le résoudre ?**
   La méthode d'apprentissage par régression linéaire fonctionne grâce à l'ajustement du paramètre de notre modèle θ. Pour ce faire, il faut minimiser la fonction de coût *(erreur entre la prédiction et la réalité)* à l'aide d'une descente de gradient ce qui permet d'obtenir un θ pour réaliser une prédiction optimale.
   On peut résoudre ce problème avec une descente de gradient, mais également avec les équations normales.
4. **Pourquoi faut-il parfois normaliser les descripteurs (features) ?**
   Il est intéressant de normaliser les données pour les ramener à une échelle commune, ce qui permet d'interpréter plus facilement les résultats et rendre les calculs plus stables et rapides. Grâce à la normalisation, on peut être sûr que chaque caractéristique contribuent équitablement à la prédiction du modèle, indépendamment de son échelle initiale.