Machine Learning TP1: Régression linéaire

Melvin DUBEE - Tanguy ROUDAUT FIPASE 24

1er octobre 2023

1 Régression linéaire avec une variable

1.1 Affichage des données

Une pemière 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

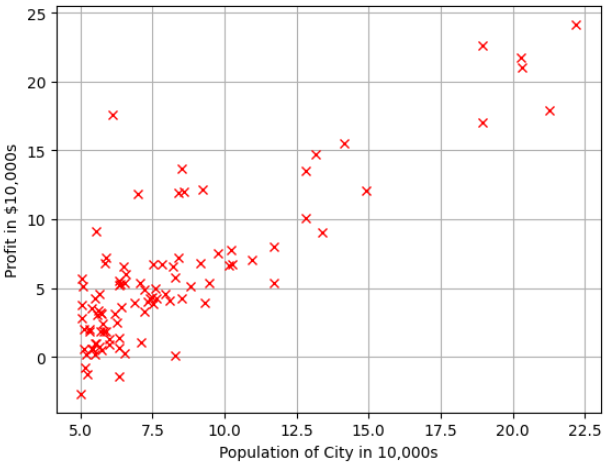

FIGURE 1 – Profit en fonction de la population

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 entré x_1 et de θ.

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

Pour que cette prédiction soit optimal, 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 réel 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 valeurs réel**, ce qui revient à déterminer l'erreur de la prédiction. On éléve 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 seul valeur qui puisse influencer notre coût est θ. Effectivement, les valeurs restante : x, y et m ; sont les valeurs de notre problèmes qui sont déterminé et non modifiable.

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 optimal.

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=0}^{m-1} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \tag{3}$$

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 divergé 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 simulanément.
Il faut réaliser cette étape un certains nombres 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 correct et que l'on à un modèle satisfaisant pour réaliser des prédictions optimal.

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

Listing 5 – Output fonction gradientDescent

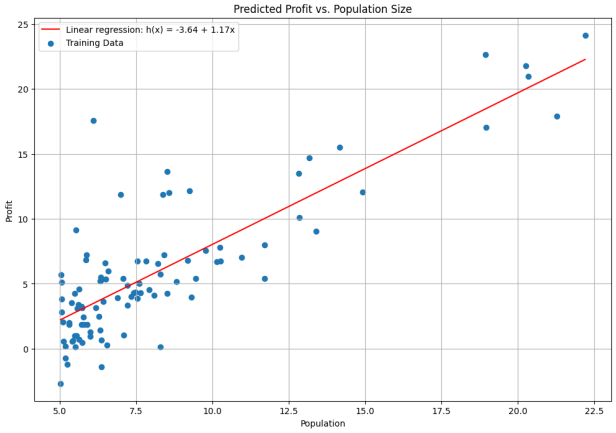

FIGURE 2 – Profit prédit vs population

```
For population = 35,000, we predict a profit of 4483.9858
For population = 70,000, we predict a profit of 45328.6063
```

Listing 6 – Prédiction

1.3 Visualisation de $J(\theta)$

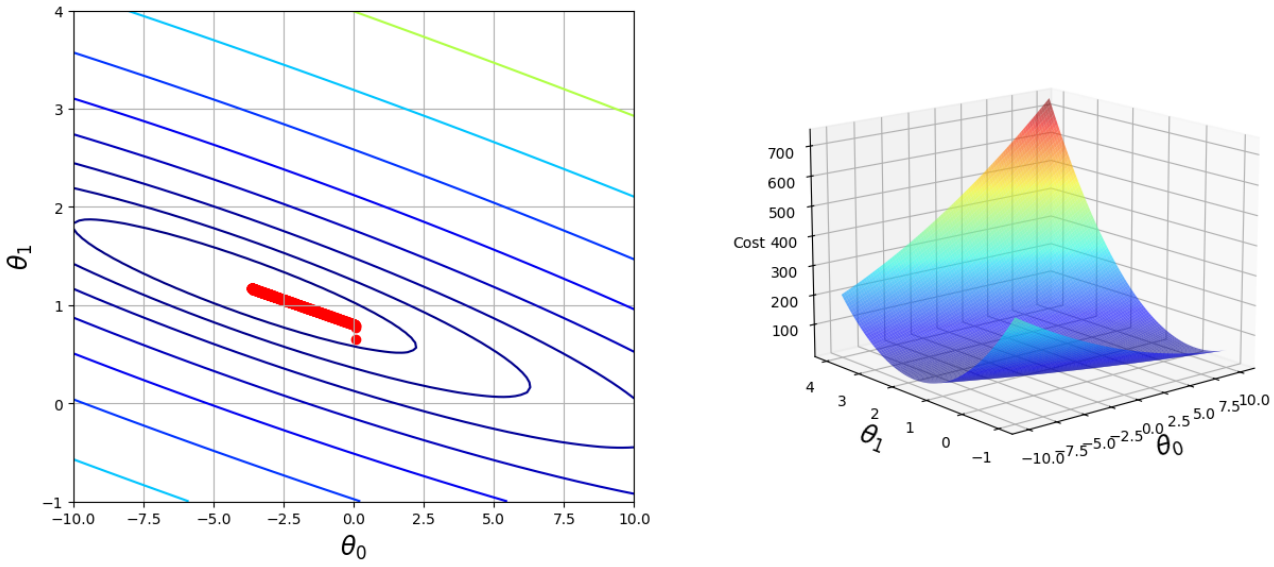

FIGURE 3 – Visualisation de $J(\theta)$

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

2 Régression linéaire avec plusieurs variables

2.1 Normalisation des caractéristiques

2.2 Descente de gradient

2.2.1 Calcul du coût $J(\theta)$

2.2.2 Descente de gradient

2.3 Exercice facultatif : Equations normales

3 Questions