République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Alger 1 – BENYOUCEF Benkhedda

Faculté des Sciences
Département Informatique

Rapport

Module : Intelligence Artificielle

Réalisé par :

Moulay Abdellah Asma
Belabbas Rania

Section : M1 ISII groupe 3

Année universitaire : 2024/2025

La Régression Linéaire :

Nous avons un Dataset qui représente le prix moyen en fonction de la superficie. Nous allons appliquer une régression linéaire, qui se compose de quatre étapes principales :

- Préparation du Dataset
- Création du modèle
- La fonction du cout
- L'algorithme de minimisation

1- Préparation du Dataset

Tout d'abord on a commencé par lire et afficher le Dataset :

```
import pandas as pd
df=pd.read_excel("Prix-Moyen.xlsx")
print(df)
```

```
          prix  superficie
0     55776390         327
1     19786120         116
2     57482090         337
3     48612450         285
4     23538660         138
...        ...         ...
1494  39572240         232
1495  49465300         290
1496  28314620         166
1497  30020320         176
1498  60552350         355

[1499 rows x 2 columns]
```

```
print(df.head()) # afficher les 5 premieres colonnes
```

```
      Prix   Superficie
0  55776390         327
1  19786120         116
2  57482090         337
3  48612450         285
4  23538660         138
```

```
print(df.tail()) # afficher les dernieres colonnes
```

```
         Prix   Superficie
1494  39572240         232
1495  49465300         290
1496  28314620         166
1497  30020320         176
1498  60552350         355
```

Ensuite, nous avons entamé le nettoyage des données, où nous avons vérifié le type des données et recherché des valeurs manquantes

```
print(data.isnull().sum()) # verfier missing values
```

```
prix          0
superficie    0
dtype: int64
```

```
# Vérifier les types de données
print(df.dtypes)
```

```
prix          int64
superficie    int64
dtype: object
```

(En réalité, les données n'avaient pas réellement besoin de nettoyage, car il n'y avait ni doublons ni valeurs manquantes et toutes les données étaient du même types)

Puis, nous avons voulu afficher un nuage de points pour visualiser la relation entre les deux colonnes, 'superficie' et 'prix' :

```
import matplotlib.pyplot as plt
plt.scatter(df['Prix'], df['superficie'])
plt.title('Nuage de points de Colonne1 vs Colonne2')
plt.xlabel('Superficie')
plt.ylabel('Prix')
plt.grid()
plt.show()
```

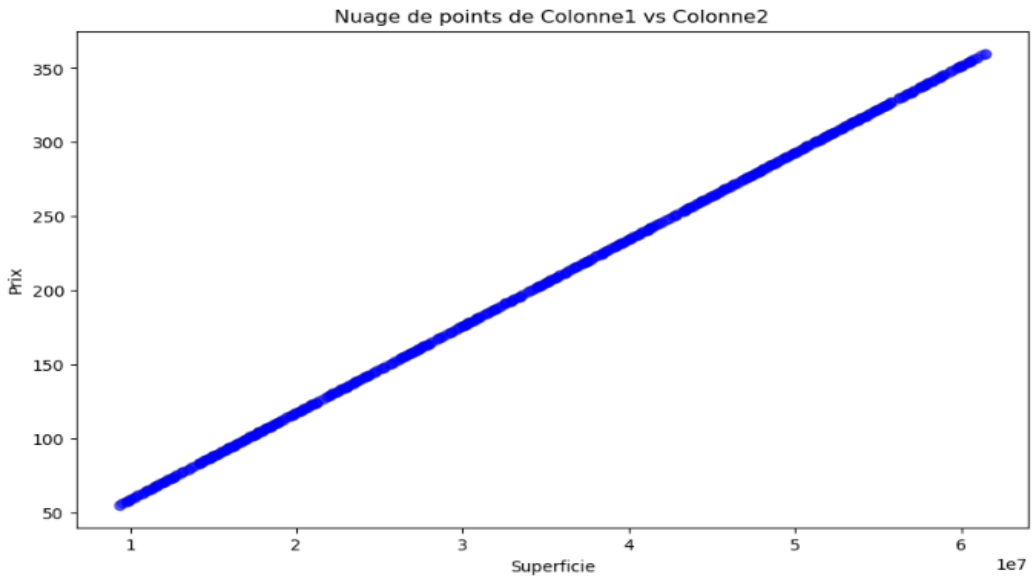
Nous avons rencontré l'erreur suivante : `KeyError: 'Prix'`

Explication : L'erreur indique que la colonne 'Prix' n'existe pas avec ce nom exact dans votre dataset.

Solution : nous avons vérifié le nom des colonnes avec la commande : **« df.columns »**, nous avons obtenu le résultat suivant: **Index(['Prix ', 'superficie'], dtype='object')** → ce qui indique qu'il y a un espace à la fin du nom de la colonne 'Prix'. Donc :

```
# Supprimer les espaces dans les noms de colonnes
df.columns = df.columns.str.strip()
```

Résultat : ensuite, cela a fonctionné et a affiché le nuage de points :



2- Le modèle :

Nous avons choisi le modèle de régression linéaire **y=ax+b** où **a** et **b** qui sont les paramètres, pour prédire une relation entre la superficie (x) et le prix(y). Il calcule ces paramètres, prédit les valeurs, et trace un graphique montrant les données réelles sous forme de points et la droite de régression ajustée pour visualiser la tendance.

```
import seaborn as sns
from sklearn.linear_model import LinearRegression
import numpy as np


# Extraire les colonnes
X = df["superficie"].values
y = df["prix"].values

# Créer le modèle de régression linéaire
model = LinearRegression()
model.fit(X, y)

# Obtenir les paramètres du modèle
a = model.coef_[0]
b = model.intercept_
print(f"Modèle : y = {a:.2f}x + {b:.2f}")

# Prédictions pour tracer la droite de régression
y_pred = model.predict(X)

plt.figure(figsize=(10, 6))
sns.scatterplot(x=df["superficie"], y=df["prix"], color="blue", label="Données réelles")
plt.plot(df["superficie"], y_pred, color="red", label=f"Modèle : y = {a:.2f}x + {b:.2f}")
plt.xlabel("Superficie")
plt.ylabel("Prix")
plt.legend()
plt.show()
```

Nous avons rencontré l'erreur suivante :

```
ValueError: Expected 2D array, got 1D array instead:
array=[327 116 337 ... 166 176 355].
Reshape your data either using array.reshape(-1, 1) if your data has a single feature or array.reshape(1, -1) if it contains a
single sample.
```

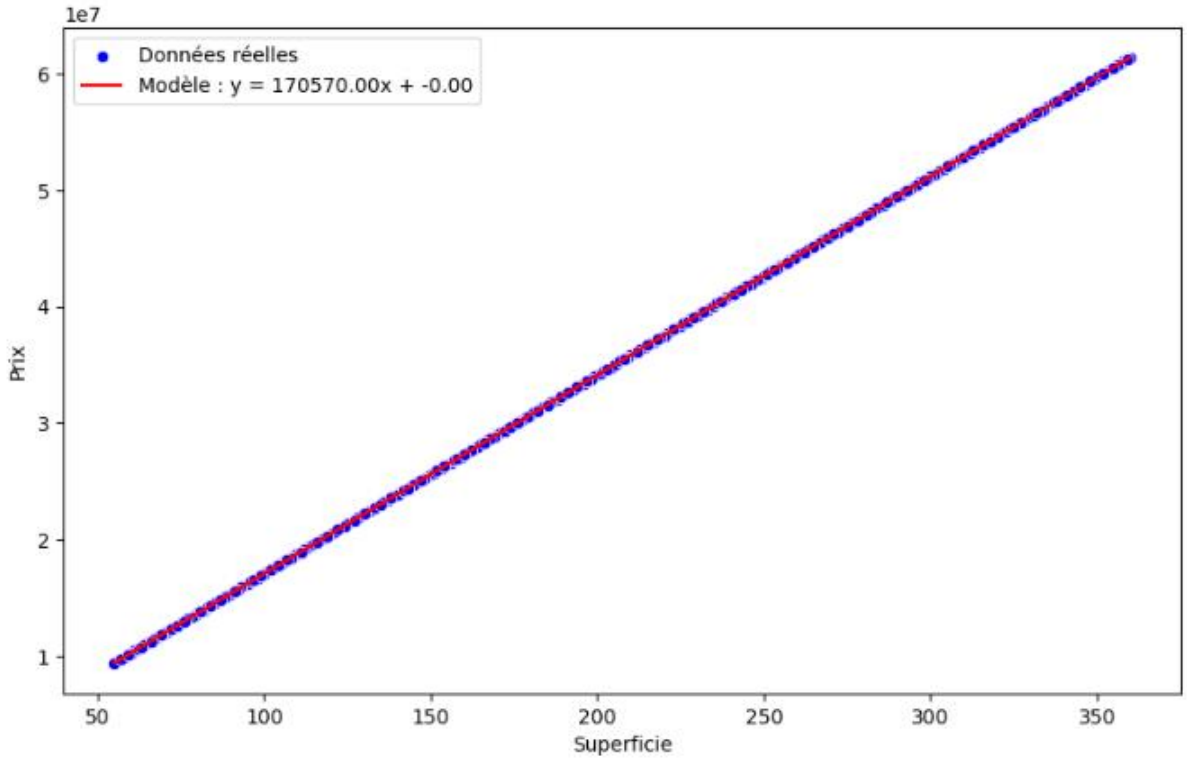
Explication : on a fourni des données sous la forme d'un tableau **1D** (une seule dimension) à une méthode ou un modèle comme la régression linéaire qui attend un tableau **2D** (deux dimensions).

Solution : Pour remédier à l'erreur on utiliser **« .reshape(1, -1) »** pour convertir un tableau 1D en 2D.

(Si nous travaillons avec plusieurs caractéristiques, nous n'utilisons pas reshape, car les données correspondent déjà à un tableau 2D.)

Résultat :

```
Modèle : y = 170570.00x + -0.00
```



3- **La fonction du cout :** Cette fonction mesure l'erreur globale entre les prédictions **y_pred** et les valeurs réelles **y**

```
X = df["superficie"].values
y = df["prix"].values

def fonction_cout(X, y, a, b):

    m = len(y)
    y_pred = a * X + b
    cout = (1 / (2 * m)) * np.sum((y_pred - y) ** 2)  # Formule de J(a, b)
    return cout
cout = fonction_cout(X, y, a, b)
print(f"Valeur de la fonction du coût : {cout:.2f}")
```

```
Valeur de la fonction du coût : 0.00
```

Cela signifie qu'il n'y a **aucune erreur** entre les valeurs réelles et les prédictions du modèle

4- L’algorithme de minimisation :

L’objectif est d’ajuster les paramètres d’un modèle pour réduire l'erreur entre les prédictions et les valeurs réelles (La fonction du cout).

La descente de gradient est l'algorithme de minimisation le plus utilisé pour la régression linéaire

```python
def descente_gradient(X, y, a_init, b_init, learning_rate, n_iterations):
    m = len(y)
    a, b = a_init, b_init  # Initialisation des paramètres

    for i in range(n_iterations):
        # Prédictions avec les paramètres actuels
        y_pred = a * X + b
        # Calcul des gradients
        grad_a = -(1 / m) * np.sum((y - y_pred) * X)
        grad_b = -(1 / m) * np.sum(y - y_pred)
        # Mise à jour des paramètres
        a = a - learning_rate * grad_a
        b = b - learning_rate * grad_b
        # Afficher l'état de convergence
        if i % 100 == 0:
            cout = fonction_cout(X, y, a, b)
            print(f"Iteration {i}: Cost = {cout:.2f}, a = {a:.2f}, b = {b:.2f}")

    return a, b

X = df["superficie"].values
y = df["prix"].values

# Paramètres initiaux
a_init = 0
b_init = 0
learning_rate = 0.001  # Réduire le taux d'apprentissage
n_iterations = 1000

# Exécution de la descente de gradient
a_opt, b_opt = descente_gradient(X, y, a_init, b_init, learning_rate, n_iterations)

# Résultat final
print(f"Paramètres optimaux : a = {a_opt:.2f}, b = {b_opt:.2f}")
```

Nous avons rencontré le résultat et l'erreur suivants :

```
Iteration 0: Cost = 1796839668177833984.00, a = 8608681.44, b = 35206.72
Iteration 100: Cost = inf, a = 22974559267919159186768004051772340619408443760431703334385018178806890955627730415318191180554086676496905953813258080728646237871621330093548034726247903560282027945077453619200, b = 939587931142718168673382995177197813024291565570927228805937087055855660088854546849284010239120081400777464854164120334352362488148396442543308344417193689044063717217533952.00
Iteration 200: Cost = nan, a = nan, b = nan
Iteration 300: Cost = nan, a = nan, b = nan
Iteration 400: Cost = nan, a = nan, b = nan
Iteration 500: Cost = nan, a = nan, b = nan
Iteration 600: Cost = nan, a = nan, b = nan
Iteration 700: Cost = nan, a = nan, b = nan
Iteration 800: Cost = nan, a = nan, b = nan
Iteration 900: Cost = nan, a = nan, b = nan
Paramètres optimaux : a = nan, b = nan
```

```
C:\Users\User\AppData\Local\Temp\ipykernel_18532\4177490537.py:7: RuntimeWarning: overflow encountered in square
  cout = (1 / (2 * m)) * np.sum((y_pred - y) ** 2)
C:\Users\User\AppData\Local\Temp\ipykernel_18532\796515688.py:12: RuntimeWarning: invalid value encountered in scalar subtract
  a = a - learning_rate * grad_a
```

Explication : Les données ne sont normalisée (X ou y contiennent des valeurs très grandes), ce qui amplifient les calculs des gradients et qui peut rapidement faire exploser le coût et les paramètres (a,b).

Solution : Ainsi la solution est de normaliser les données pour réduit l'amplitude des calculs, donc nous avons utilisé :

On soustrait la moyenne des données pour centrer les valeurs autour de zéro, Ensuite, on divise par l'écart-type pour rendre les valeurs comparables entre elles. Cela aide le modèle à mieux apprendre et à être plus stable.

```
# Normalisation
X = (X - np.mean(X)) / np.std(X)
y = (y - np.mean(y)) / np.std(y)
```

Résultat :

```
Iteration 0: Cost = 0.50, a = 0.00, b = 0.00
Iteration 100: Cost = 0.41, a = 0.10, b = 0.00
Iteration 200: Cost = 0.33, a = 0.18, b = 0.00
Iteration 300: Cost = 0.27, a = 0.26, b = 0.00
Iteration 400: Cost = 0.22, a = 0.33, b = 0.00
Iteration 500: Cost = 0.18, a = 0.39, b = 0.00
Iteration 600: Cost = 0.15, a = 0.45, b = 0.00
Iteration 700: Cost = 0.12, a = 0.50, b = 0.00
Iteration 800: Cost = 0.10, a = 0.55, b = 0.00
Iteration 900: Cost = 0.08, a = 0.59, b = 0.00
Paramètres optimaux : a = 0.63, b = 0.00
```

Conclusion :

En conclusion, nous avons appliqué les quatre étapes de la régression linéaire sur le dataset contenant des informations sur le prix et la superficie. Ces étapes incluent la préparation des données, l'ajustement du modèle, l'évaluation de la fonction de coût, et l'algorithme de minimisation : la descente de gradient.

Il est crucial de bien préparer les données (nettoyage) avant d'appliquer le modèle pour garantir des résultats fiables et efficaces.