



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 et du secteur. 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
- Création d'une interface d'estimation de Prix par Superficie et Secteur

1. Préparation du Dataset

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

```
import pandas as pd
d=pd.read_excel("DataSets-02.xlsx")
print(d)
```

	Prix	Superficie	Secteur
0	2765952	56	campagne
1	3255785	65	campagne
2	3331295	55	campagne
3	3382160	67	campagne
4	3411387	63	campagne
...
6212	181503570	498	ville
6213	182202128	496	ville
6214	182230020	485	ville
6215	183393288	482	ville
6216	183863162	497	ville

[6217 rows x 3 columns]

Suppression des espaces supplémentaires dans le nom des colonnes

```
d.columns
```

```
Index(['Prix ', 'Superficie', 'Secteur'], dtype='object')
```

```
d.columns = d.columns.str.strip() #Supprimer les espaces des noms des colonnes
```

```
d.columns
```

```
Index(['Prix', 'Superficie', 'Secteur'], dtype='object')
```

Vérification des valeurs manquantes et affichage des types de données

```
print(d.isnull().sum()) # vérifier missing values
```

```
Prix          0  
Superficie    0  
Secteur       0  
dtype: int64
```

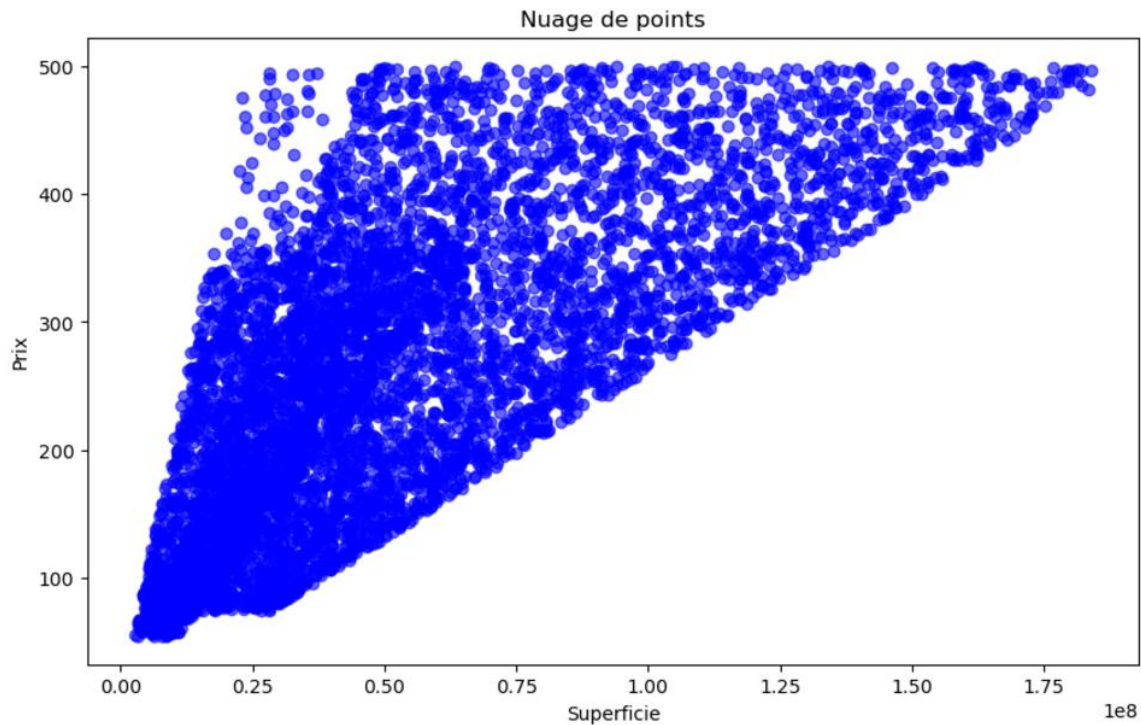
```
print(d.dtypes)
```

```
Prix          int64  
Superficie    int64  
Secteur       object  
dtype: object
```

Création du nuage de points

```
import matplotlib.pyplot as plt  
  
plt.figure(figsize=(10, 6))  
plt.scatter(d['Prix'], d['Superficie'], color='blue', alpha=0.6)  
plt.title('Nuage de points')  
plt.xlabel('Superficie')  
plt.ylabel('Prix')  
plt.show()
```

Résultat :



On a choisi de convertir la colonne 'Secteur' en valeurs numériques : attribuant la valeur 0 au secteur "campagne" et la valeur 1 au secteur "ville".

Ce type d'encodage est simple, rapide, et approprié pour une régression lorsque la variable n'a que deux valeurs distinctes.

```
#Convertir la colonne Secteur en numerique
ds={'campagne':0, 'ville':1}
d['Secteur']=d['Secteur'].map(ds)
print(d)
```

Résultat :

	Prix	Superficie	Secteur
0	2765952	56	NaN
1	3255785	65	NaN
2	3331295	55	NaN
3	3382160	67	NaN
4	3411387	63	NaN
...
6212	181503570	498	NaN
6213	182202128	496	NaN
6214	182230020	485	NaN
6215	183393288	482	NaN
6216	183863162	497	NaN

[6217 rows x 3 columns]

Dans l’affichage ci-dessous, on remarque que les valeurs de la colonne ‘Secteur’ sont **NaN** au lieu de 0 et 1. Cela signifie probablement que certaines valeurs dans cette colonne ne correspondent pas aux clés du dictionnaire **ds**, pour résoudre ce problème, nous appliquons une suppression des espaces et une conversion en minuscules :

```
#suppression des espaces et mettant tout en minuscule)
d['Secteur'] = d['Secteur'].str.strip().str.lower()
ds = {'campagne': 0, 'ville': 1}
d['Secteur'] = d['Secteur'].map(ds)
print(d)
```

Résultat d'affichage :

	Prix	Superficie	Secteur
0	2765952	56	0
1	3255785	65	0
2	3331295	55	0
3	3382160	67	0
4	3411387	63	0
...
6212	181503570	498	1
6213	182202128	496	1
6214	182230020	485	1
6215	183393288	482	1
6216	183863162	497	1

[6217 rows x 3 columns]

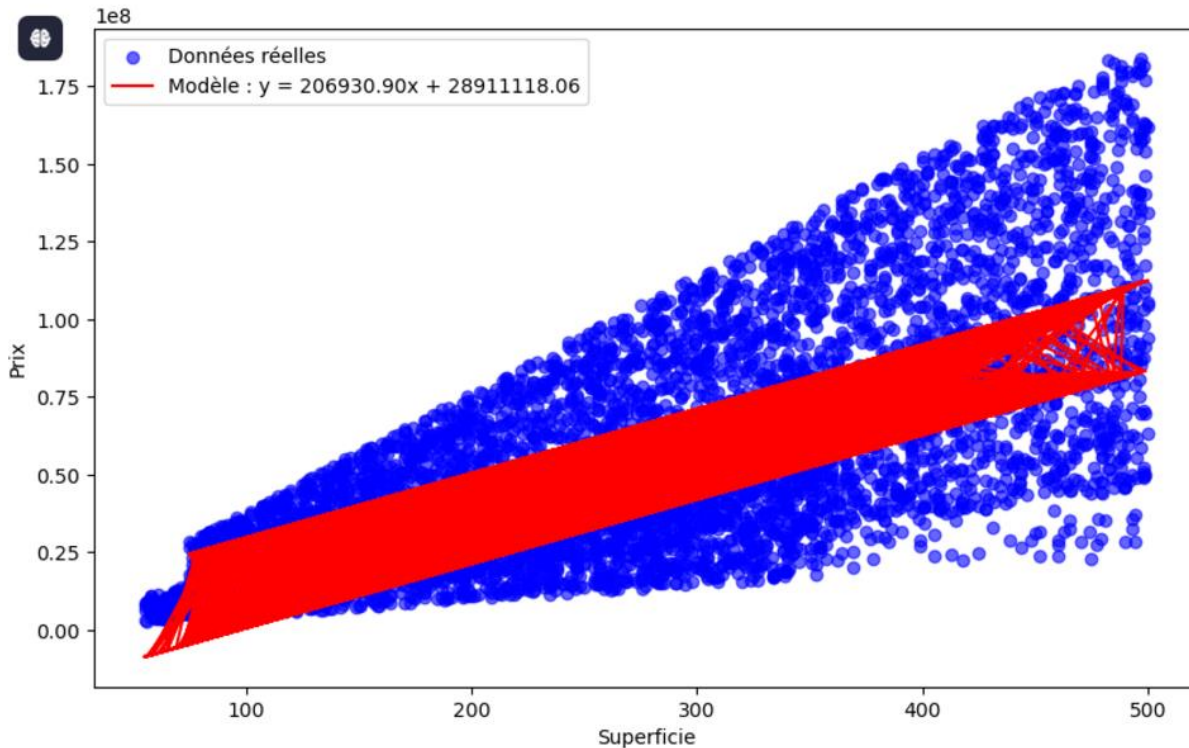
2. Le modèle :

La régression linéaire multiple est une extension de la régression linéaire simple qui permet de modéliser la relation entre une variable dépendante (y) et plusieurs variables indépendantes (x1, x2, ..., xn).

Nous avons choisi le modèle de régression linéaire multiple $y = a_1x_1 + a_2x_2 + b$. Les coefficients a_1 et a_2 sont calculés pour 'Superficie' et 'Secteur', respectivement, et b est l'ordonnée à l'origine pour prédire le prix (y) en fonction de la superficie (x1) et du secteur (x2). Il calcule les paramètres optimaux du modèle, effectue des prédictions et visualise les résultats à l'aide d'un graphique. La droite de régression ajustée pour visualiser la tendance.

Résultat :

Paramètres optimaux : a_1 (Superficie) = 206930.90, a_2 (Secteur) = 28911118.06, b = -20028509.83



L'erreur :

L'utilisation de **plt.plot** pour tracer la ligne de régression avec plusieurs variables entraîne un problème dans le graphique.

La difficulté est de chercher à relier une seule variable (Superficie) aux prédictions.

Avec plusieurs variables, cela n'a pas de sens, car la prédiction est également influencée par le Secteur.

Donc nous avons choisi d'afficher les résultats en séparant les données par secteur (campagne et ville) :

- Les prédictions spécifiques pour le secteur "campagne" sont calculées en utilisant la superficie et en remplaçant la valeur du secteur par 0.
- Les prédictions pour le secteur "ville" sont calculées de la même manière, mais avec la valeur du secteur remplacée par 1.

```

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Récupération des valeurs
X = d[['Superficie', 'Secteur']].values
y = d["Prix"].values

# Ajouter la colonne de biais (1)
X = np.hstack((X, np.ones((X.shape[0], 1))))

# Calcul des paramètres
theta = np.linalg.inv(X.T @ X) @ X.T @ y
print(f"Paramètres optimaux : a1 (Superficie) = {theta[0]:.2f}, a2 (Secteur) = {theta[1]:.2f}, b = {theta[2]:.2f}")

# Prédictions
y_pred = X @ theta

# Séparer les données par secteur
campagne = d[d['Secteur'] == 0]
ville = d[d['Secteur'] == 1]

# Prédictions spécifiques
y_pred_campagne = theta[0] * campagne['Superficie'] + theta[1] * 0 + theta[2]
y_pred_ville = theta[0] * ville['Superficie'] + theta[1] * 1 + theta[2]

plt.figure(figsize=(20, 12))
# On trace deux régressions distinctes pour chaque secteur
plt.scatter(campagne["Superficie"], campagne["Prix"], color="green", alpha=0.6, label="Campagne")
plt.scatter(ville["Superficie"], ville["Prix"], color="blue", alpha=0.6, label="Ville")

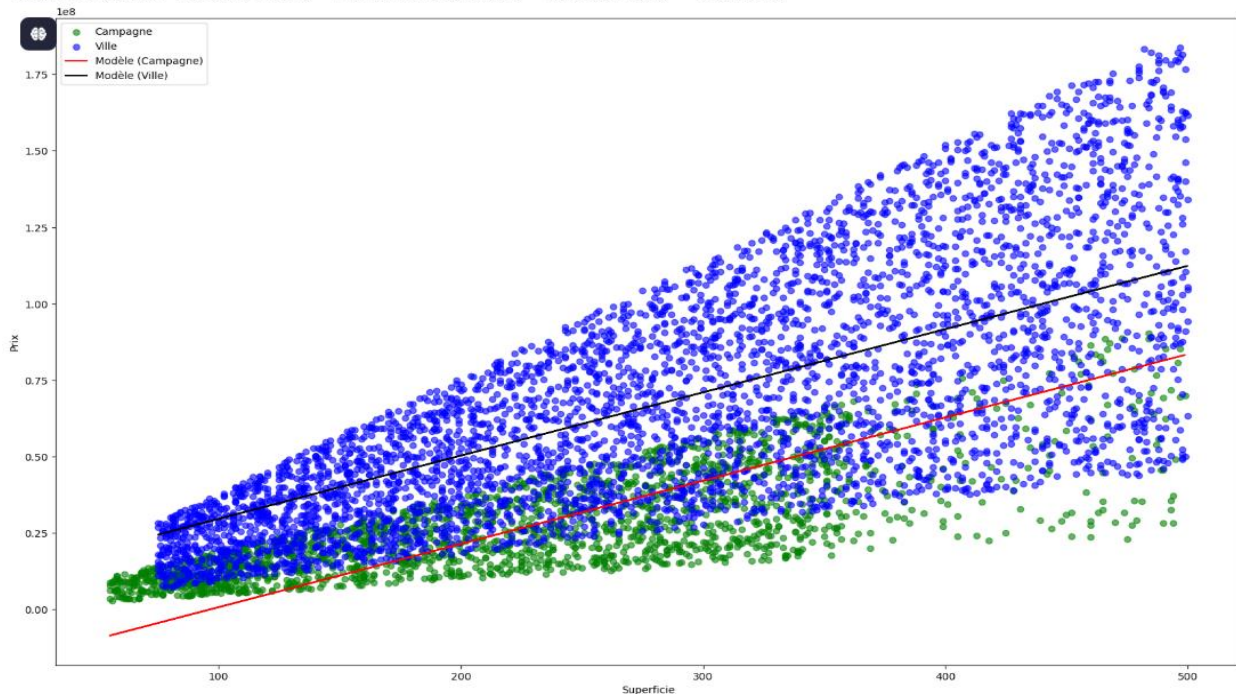
plt.plot(campagne["Superficie"], y_pred_campagne, color="red", label="Modèle (Campagne)")
plt.plot(ville["Superficie"], y_pred_ville, color="black", label="Modèle (Ville)")

plt.xlabel("Superficie")
plt.ylabel("Prix")
plt.legend()
plt.show()

```

Résultat :

Paramètres optimaux : a1 (Superficie) = 206930.90, a2 (Secteur) = 28911118.06, b = -20028509.83



(Note : verts pour la campagne et points bleus pour la ville ainsi que les lignes de régression rouge pour la campagne et noir pour la ville).

3. La fonction du cout :

Cette fonction mesure l'erreur globale entre les prédictions **y_pred** et les valeurs réelles **y**. Une valeur de coût plus faible indique un meilleur ajustement du modèle aux données.

```
def fonction_cout(X, y, theta):  
    m = len(y) # Nombre d'exemples  
    y_pred = X @ theta  
    erreur = y_pred - y # Erreurs de prédiction  
    cout = (1 / (2 * m)) * np.sum(erreur ** 2) # Fonction du coût J(θ)  
    return cout  
  
cout = fonction_cout(X, y, theta)  
print(f"Valeur de la fonction du coût : {cout:.2f}")  
  
Valeur de la fonction du coût : 269403657462891.81
```

La valeur de la fonction de coût est un indicateur clé de la performance du modèle. Une valeur élevée comme celle-ci nécessite une attention particulière pour identifier les problèmes potentiels et apporter des améliorations au modèle de régression linéaire.

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.

```
def descente_de_gradient(X, y, theta, learning_rate, iterations):  
    m = len(y)  
    historique_cout = []  
    for i in range(iterations):  
        # Calcul des prédictions  
        y_pred = X @ theta  
        # Calcul du gradient  
        gradient = (1 / m) * (X.T @ (y_pred - y))  
        # Mise à jour des paramètres  
        theta = theta - learning_rate * gradient  
        cout = fonction_cout(X, y, theta)  
        historique_cout.append(cout)  
        # Affichage des progrès toutes les 100 itérations  
        if i % 100 == 0:  
            print(f"Iteration {i}: Cost = {cout:.2f}")  
    return theta, historique_cout  
  
# Chargement des données  
X = d[['Superficie', 'Secteur']].values  
X = np.hstack((X, np.ones((X.shape[0], 1)))) # Ajout de la colonne de biais  
y = d["Prix"].values  
  
# Initialisation des paramètres theta à 0  
theta = np.zeros(X.shape[1])  
  
# Paramètres pour la descente de gradient  
learning_rate = 0.01  
iterations = 1000  
# Exécution de la descente de gradient  
theta_optimal, historique_cout = descente_de_gradient(X, y, theta, learning_rate, iterations)  
print(f"Paramètres optimaux : {theta_optimal}")
```


Ce code met en œuvre la méthode de descente de gradient pour optimiser les paramètres d'un modèle de régression linéaire. Il calcule les prédictions, le gradient, met à jour les paramètres, et suit l'évolution de la fonction de coût au fil des itérations. À la fin, il affiche les paramètres optimaux qui minimisent la fonction de coût.

Résultat :

```
Iteration 0: Cost = 1392562853158644350976.00
Iteration 100: Cost = inf
Iteration 200: Cost = nan
Iteration 300: Cost = nan
Iteration 400: Cost = nan
Iteration 500: Cost = nan
Iteration 600: Cost = nan
Iteration 700: Cost = nan
Iteration 800: Cost = nan
Iteration 900: Cost = nan
Paramètres optimaux : [nan nan nan]

C:\Users\User\AppData\Local\Temp\ipykernel_10828\2059198730.py:5: RuntimeWarning: overflow encountered in square
  cout = (1 / (2 * m)) * np.sum(erreur ** 2) # Fonction du coût J(θ)
C:\Users\User\anaconda3\lib\site-packages\numpy\core\fromnumeric.py:86: RuntimeWarning: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
C:\Users\User\AppData\Local\Temp\ipykernel_10828\4047802173.py:10: RuntimeWarning: overflow encountered in matmul
  gradient = (1 / m) * (X.T @ (y_pred - y))
C:\Users\User\AppData\Local\Temp\ipykernel_10828\4047802173.py:10: RuntimeWarning: invalid value encountered in matmul
  gradient = (1 / m) * (X.T @ (y_pred - y))
C:\Users\User\AppData\Local\Temp\ipykernel_10828\4047802173.py:13: RuntimeWarning: invalid value encountered in subtract
  theta = theta - learning_rate * gradient
```

Explication de l'erreur :

1. Valeur de Coût Énorme :

- La première valeur de coût est **1392562853158644350976.00**, ce qui est extrêmement élevé. Cela suggère que les prédictions sont très éloignées des valeurs réelles.

2. Coût Infini et NaN :

- Les valeurs de coût deviennent **inf** (infini) et **NaN** (not a number) dans les itérations suivantes. Cela se produit généralement lorsque les calculs dépassent la capacité de représentation des nombres en virgule flottante, souvent à cause d'un overflow.

3. Avertissements de Débordement :

- Les avertissements tels que **overflow encountered in square** et **invalid value encountered in matmul** indiquent que des opérations mathématiques produisent des résultats qui ne peuvent pas être représentés correctement.

Solutions Suggérées :

Normalisation des Données : Cela peut aider à éviter des valeurs extrêmes qui peuvent causer des débordements. On a utilisé la normalisation Min-Max ou la standardisation (soustraction de la moyenne et division par l'écart type). Ça permet d'assurer une meilleure stabilité numérique.

```
def descente_de_gradient(X, y, theta, learning_rate, iterations):
    m = len(y)
    historique_cout = []

    for i in range(iterations):
        # Calcul des prédictions
        y_pred = X @ theta

        # Calcul du gradient
        gradient = (1 / m) * (X.T @ (y_pred - y))

        # Mise à jour des paramètres
        theta = theta - learning_rate * gradient

        # Calcul et enregistrement du coût
        cout = fonction_cout(X, y, theta)
        historique_cout.append(cout)

        # Affichage des progrès toutes les 100 itérations
        if i % 100 == 0:
            print(f"Iteration {i}: Cost = {cout:.2f}")

    return theta, historique_cout

# Chargement des données
X = d[['Superficie', 'Secteur']].values
X = np.hstack((X, np.ones((X.shape[0], 1)))) # Ajout de la colonne de biais
y = d["Prix"].values

# Initialisation des paramètres theta à 0
theta = np.zeros(X.shape[1])

# Normalisation
X[:, 0] = (X[:, 0] - np.mean(X[:, 0])) / np.std(X[:, 0]) # Superficie
X[:, 1] = (X[:, 1] - np.mean(X[:, 1])) / np.std(X[:, 1]) # Secteur
# Cela signifie que les valeurs de la colonne Superficie et secteur auront désormais une moyenne de 0 et un écart type de 1.

y = (y - np.mean(y)) / np.std(y)

# Paramètres pour la descente de gradient
learning_rate = 0.001
iterations = 1000

# Exécution de la descente de gradient
theta_optimal, historique_cout = descente_de_gradient(X, y, theta, learning_rate, iterations)

print(f"Paramètres optimaux : {theta_optimal}")
```

Résultat :

```
Iteration 0: Cost = 0.50
Iteration 100: Cost = 0.43
Iteration 200: Cost = 0.38
Iteration 300: Cost = 0.34
Iteration 400: Cost = 0.30
Iteration 500: Cost = 0.28
Iteration 600: Cost = 0.26
Iteration 700: Cost = 0.24
Iteration 800: Cost = 0.23
Iteration 900: Cost = 0.22
Paramètres optimaux : [ 4.29526355e-01  2.72162209e-01 -3.50809046e-17]
```

5. Création de l'interface : Estimation de Prix Immobilier par Superficie et Secteur

Cette application permet aux utilisateurs d'estimer le prix d'un bien immobilier en fonction de sa superficie et de son secteur (campagne ou ville) à l'aide d'un modèle de régression linéaire.

```
import numpy as np
import pandas as pd
import tkinter as tk
from tkinter import ttk, messagebox

# Charger les données (déjà normalisées, theta calculé)
# Assurez-vous que d est le DataFrame contenant vos données initiales
d = pd.read_excel("DataSets-02.xlsx")

ds = {'campagne': 0, 'ville': 1}
d['Secteur'] = d['Secteur'].map(ds)

d.columns = d.columns.str.strip()

x = d[['Superficie', 'Secteur']].values
y = d['Prix'].values

# Calculer la normalisation (moyenne et écart-type)
mean_superficie = np.mean(X[:, 0])
std_superficie = np.std(X[:, 0])
mean_secteur = np.mean(X[:, 1])
std_secteur = np.std(X[:, 1])

# Normalisation des données
X[:, 0] = (X[:, 0] - mean_superficie) / std_superficie
X[:, 1] = (X[:, 1] - mean_secteur) / std_secteur

# Ajout de la colonne de biais
X_bias = np.hstack((X, np.ones((X.shape[0], 1))))

# Calcul des paramètres theta
theta = np.linalg.inv(X_bias.T @ X_bias) @ X_bias.T @ y

# Interface graphique
def estimer_prix():
    try:
        superficie_input = float(entry_superficie.get())
        secteur_input = 0 if combo_secteur.get() == "campagne" else 1

        # Préparer les données d'entrée
        X_input = np.array([[superficie_input, secteur_input]])
        X_input[:, 0] = (X_input[:, 0] - mean_superficie) / std_superficie
        X_input[:, 1] = (X_input[:, 1] - mean_secteur) / std_secteur

        # Ajouter la colonne de biais
        X_input = np.hstack((X_input, np.ones((X_input.shape[0], 1))))

        # Prédiction
        y_pred = X_input @ theta
        label_result['text'] = f"Prix estimé : {y_pred[0]:.2f}"
    except Exception as e:
        messagebox.showerror("Erreur", f"Une erreur est survenue : {e}")
```

```

# Création de L'interface
app = tk.Tk()
app.title("Estimation de prix")

frame = ttk.Frame(app, padding=20)
frame.grid()

ttk.Label(frame, text="Superficie :").grid(column=0, row=0, sticky=tk.W)
entry_superficie = ttk.Entry(frame)
entry_superficie.grid(column=1, row=0)

ttk.Label(frame, text="Secteur:").grid(column=0, row=1, sticky=tk.W)
combo_secteur = ttk.Combobox(frame, values=["campagne", "ville"])
combo_secteur.grid(column=1, row=1)
combo_secteur.current(0)

ttk.Button(frame, text="Calculer le prix", command=estimer_prix).grid(column=0, row=2, colspan=2)

label_result = ttk.Label(frame, text="Prix calculé : ")
label_result.grid(column=0, row=3, colspan=2)

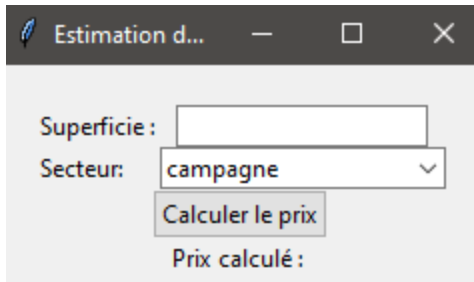
app.mainloop()

```

- Les bibliothèques **numpy** et **pandas** sont utilisées pour la manipulation des données, tandis que **tkinter** est utilisé pour créer l'interface graphique.
- Les données sont chargées à partir d'un fichier Excel. Assurez-vous que le fichier existe et que le chemin est correct.
- La moyenne et l'écart-type de chaque caractéristique sont calculés pour normaliser les données.
- Les paramètres du modèle sont calculés en utilisant la formule de la régression linéaire.
- **Fonction d'Estimation de Prix** : Cette fonction est appelée lorsque l'utilisateur clique sur le bouton pour estimer le prix. Elle récupère les entrées de l'utilisateur, normalise les données, ajoute la colonne de biais et effectue la prédiction.

Ce code crée une application simple pour estimer le prix d'un bien immobilier en fonction de sa superficie et de son secteur. Les données sont normalisées avant d'être utilisées pour faire des prédictions à l'aide d'un modèle de régression linéaire. L'interface graphique permet à l'utilisateur de saisir les informations nécessaires et d'afficher le prix estimé.

Résultat :



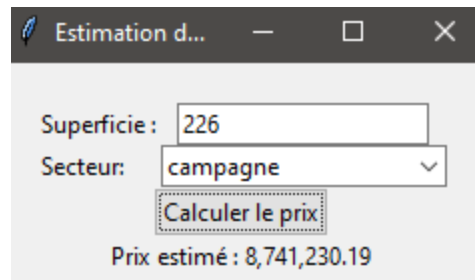
Estimation d...

Superficie :

Secteur: campagne

Calculer le prix

Prix calculé :



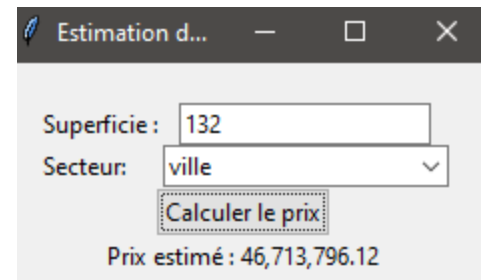
Estimation d...

Superficie : 226

Secteur: campagne

Calculer le prix

Prix estimé : 8,741,230.19



Estimation d...

Superficie : 132

Secteur: ville

Calculer le prix

Prix estimé : 46,713,796.12

Remarque : Lorsque la superficie d'un bien immobilier augmente, son prix tend également à augmenter, et pour une superficie équivalente, les biens situés en ville ont des prix plus élevés que ceux situés à la campagne.

Conclusion :

En résumé, nous avons conçu et développé un projet d'estimation des prix immobiliers qui intègre des méthodes d'apprentissage automatique pour fournir des évaluations précises basées sur des données de superficie et de secteur. Nous avons mis en place un modèle de régression linéaire qui analyse les relations entre ces variables et les prix. De plus, une interface utilisateur conviviale a été créée pour permettre aux utilisateurs d'interagir facilement avec le modèle et d'obtenir des estimations en temps réel.