Les objectifs de ce TP

1) Charger les données pour la régression
   a. Simulé des données par **make_regression** du module **sklearn.datasets**
2) Mettre en œuvre le modèle de régression linéaire par
   a. Equations normales
   b. Descente de gradient (effet du learning rate)
   c. **LinearRegression** du module **sklearn.linear_model**
3) Faire des figures dans le cas 1D et 2D
4) Calculer le coefficient de performance **r2_score manuellement avec numpy**
5) Séparer les données en trainset et testset par **train_test_split** du module **sklearn.model_selection**
6) **Evaluer la qualité du modèle de régression par mean_squared_error et r2_score du module sklearn.metrics**

Partie 1 : Régression Linéaire Simple Numpy

```
import numpy as np
from sklearn.datasets import make_regression
import matplotlib.pyplot as plt
```

1.1. Dataset

Génération de données aléatoires avec une tendance linéaire avec make_regression: on a un dataset (x,y) qui contient 100 exemples, et une seule variable x.

Note: chaque fois que la cellule est executée, des données différentes sont générer. Utiliser np.random.seed(0) pour reproduire le même Dataset à chaque fois.

Récuperer les features x et le target y.

```
np.random.seed(0) # pour toujours reproduire le meme dataset
    x, y = make_regression(n_samples=100, n_features=1, noise=10)
```

Les commentaires sous python sont précédés par le symbole #.

Important: vérifier les dimensions de x et y. On remarque que y n'a pas les dimensions (100, 1). On corrige le problème avec **np.reshape**

```
print(x.shape)
print(y.shape)

# redimensionner y
y = y.reshape(y.shape[0], 1)

print(y.shape)
```

1.2. Pour visualiser les données, nous utilisons la librairie **matplotlib**. Le code suivant permet de tracer la sortie y en fonction de l'entrée x (cas avec un seul feature).

```
# afficher les résultats. x en abscisse et y en ordonnée
plt.scatter(x, y)
plt.xlabel ("x")
plt.ylabel (" y")
plt.title(" Le Titre")
plt.show()
```

1.3. Création de la matrice X qui contient la colonne de Biais. Pour ca, on colle l'un contre l'autre le vecteur x et un vecteur 1 (avec np.ones) de dimension égale a celle de x.

```
X = np.hstack((np.ones(x.shape),x))
print(X.shape)
```

1.4. Finalement, création d'un vecteur parametre a, initialisé avec des coefficients aléatoires. Ce vecteur est de dimension (2, 1). Si on désire toujours reproduire le meme vecteur a, on utilise comme avant **np.random.seed(0).**

```
np.random.seed(0) # pour produire toujours le même vecteur a aléatoire
a = np.random.randn(2, 1)
a
```

1.5. **Modèle Linéaire** : On implémente un modèle $F=X.a$, puis on teste le modèle pour voir s'il n'y a pas de bug (bonne pratique oblige). En plus, cela permet de voir à quoi ressemble le modèle initial, défini par la valeur de a

```
def model(X, a):
    return X.dot(a)
```

```
plt.scatter(x, y)
plt.plot(x, model(X, a), c='r')
```

1.6. Fonction Cout: Erreur Quadratique moyenne

On mesure les erreurs du modele sur le Dataset X, y en implémenterl'erreur quadratique moyenne, **Mean Squared Error (MSE)** en anglais.

$$J(a) = \frac{1}{2m} \sum (X.a - y)^2$$

Ensuite, on teste notre fonction, pour voir s'il n'y a pas de bug.

```
def cost_function(X, y, a):
    m = len(y)
    return 1/(2*m) * np.sum((model(X, a) - y)**2)

cost_function(X, y, a)
```

1.7. Gradient et Descente de Gradient

On implémente la formule du gradient pour la **MSE :**

$$\nabla_a J = \frac{1}{m} X^T.(X.a - y)$$

Ensuite on utilise cette fonction dans la descente de gradient:

$$a = a - \alpha \nabla_a J$$

```
def grad(X, y, a):
    m = len(y)
    return 1/m * X.T.dot(model(X, a) - y)
```

```
def gradient_descent(X, y, a, learning_rate, n_iterations):

    # création d'un tableau de stockage pour enregistrer l'évolution du Cout du modele
    cost_history = np.zeros(n_iterations)

    for i in range(0, n_iterations):
        # mise a jour du parametre theta (formule du gradient descent)
        a = a - learning_rate * grad(X, y, a)
        cost_history[i] = cost_function(X, y, a)
        # on enregistre la valeur du Cout au tour i dans cost_history[i]

    return a, cost_history
```

1.8. Phase d'entrainement

On définit un **nombre d'itérations**, ainsi qu'un **pas d'apprentissage** α**.**

Une fois le modele entrainé, on observe les résultats par rapport à notre Dataset

```
n_iterations = 1000
learning_rate = 0.01
```

```
theta_final, cost_history = gradient_descent(X, y, a, learning_rate,
n_iterations)

# voici les parametres du modele une fois que la machine a été entrainée
theta_final
```

```
# création d'un vecteur prédictions qui contient les prédictions de notre modele final
predictions = model(X, theta_final)

# Affiche les résultats de prédictions (en rouge) par rapport a notre Dataset (en bleu)
plt.scatter(x, y)
plt.plot(x, predictions, c='r')
```

1.9. Courbes d'apprentissage

Pour vérifier si notre algorithme de Descente de gradient a bien fonctionné, on observe l'évolution de la fonction cout a travers les itérations. On est sensé obtenir une courbe qui diminue a chaque itération jusqu'a stagner a un niveau minimal (proche de zéro). Si la courbe ne suit pas ce motif, alors le pas **learning_rate** est peut-etre trop élevé, il faut prendre un pas plus faible.

Pour cela, utiliser la variable host_history pour faire ce plot.

```
plt.plot()
```

1.10. Evaluation finale

Pour évaluer la réelle performance de notre modele avec une métrique populaire, on peut utiliser le **coefficient de détermination**, aussi connu sous le nom R^2. Il nous vient de la méthode des moindres carrés. Plus le résultat est proche de 1, meilleur est votre modèle.

Définire une fonction qui retourne R^2.

```
def coef_determination(y, pred):
    u =
    v =
    return 1 - u/v
```

```
coef_determination(y, predictions)# afficher le coefficient de performan
```

Partie 2 : Régression Linéaire Multiple et Polynomiale Numpy

2.1. Régression Polynomiale: 1 variable x_1

2.1.1 Dataset

Pour développer un modèle polynomial à partir des équations de la régression linéaire, il suffit d'ajouter des degrés de polynome dans les colonnes de la matrice X ainsi qu'un nombre égal de lignes dans le vecteur a.

Ici, nous allons développer un ploynome de degré 2: $f(x)=a_2x^2+a_1x+a_0$. Pour celà, il faut développer les matrices suivantes:

$$X=\begin{bmatrix} 1 & x^{(1)} & x^{2(1)} \\ \vdots & \vdots & \vdots \\ 1 & x^{(m)} & x^{2(m)} \end{bmatrix}$$

$$a=\begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix}$$

$$y=\begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$ ***note :*** *le vecteur y reste le meme que pour la régression linéaire*

```
np.random.seed(0) # permet de reproduire l'aléatoire

# creation d'un dataset (x, y) linéaire
x, y = make_regression(n_samples=100, n_features=1, noise = 10)
# modifie les valeurs de y pour rendre le dataset non-linéaire
y = y + abs(y/2)

plt.scatter(x, y) # afficher les résultats. x en abscisse et y en ordonnée
```

Question : Comment peut-on utiliser la première partie pour faire la régression polynomiale? (suivre les mêmes démarches que la partie 1)

2.2. Régression Multiples Variables

C'est lorsqu'on intègre plusieures variables x_1, x_2, x_3, *etc*.

à notre modèle que les choses commencent à devenir vraiment intéressantes. C'est peut-être aussi à ce moment que les gens commencent parfois à parler *d'intelligence artificielle*, car il est difficile pour un être humain de se représenter dans sa tête un modèle à plusieurs dimensions (nous n'évoluons que dans un espace 3D). On se dit alors que la machine, quant à elle, arrive à se réprésenter ces espaces, car elle y trouve le meilleur modèle (avec la descente de gradient) et les gens disent donc qu'elle est intelligente, alors que ce ne sont que des mathématiques.

2.2.1 Dataset

Maintenant, nous allons créer un modèle à 2 variables x_1, x_2. Pour cela, il suffit d'injecter les différentes variables x_1, x_2 (les **features** en anglais) dans la matrice X, et de créer le vecteur a qui s'accorde avec:

$$X = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} \\ \vdots & \vdots & \vdots \\ 1 & x_1^{(m)} & x_1^{(m)} \end{bmatrix}$$

$$a = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix}$$

$$y = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$ ***note** : le vecteur y reste le meme que pour la régression linéaire*.

```
np.random.seed(0) # permet de reproduire l'aléatoire

# creation d'un dataset (x, y) linéaire
x, y = make_regression(n_samples=100, n_features=2, noise = 10)

# afficher les résultats. x_1 en abscisse et y en ordonnée
plt.scatter(x[:,0], y)
```

Ce Dataset ne contenant que 2 variables x_1 *et* x_2, il est possible de le visualiser dans un espace 3D. Comme vous pouvez le voir, ce modèle peut être représenté par une surface. Au passage, cette surface est plane car make_regression nous retourne des données linéaires. Si on veut créer une surface non plane, il suffit de modifier la valeur de y comme nous l'avons fait au début.

```
from mpl_toolkits.mplot3d import Axes3D
#%matplotlib notebook #activez cette ligne pour manipuler le graph 3D

ax = fig.add_subplot(111, projection='3d')
```

```
ax.scatter(x[:,0], x[:,1], y) # affiche en 3D la variable x_1, x_2, et la target y

# affiche les noms des axes
ax.set_xlabel('x_1')
ax.set_ylabel('x_2')
ax.set_zlabel('y')
```

Question : Généraliser la partie 1 pour deux features.

Partie 3: Régression Linéaire avec la Librairie sklearn

La méthode de régression linéaire est implémentée dans la librairie **sklearn.linear_model** sous le nom **LinearRegression**. Utiliser cette fonction pour programmer la régression linéaire de vos données, puis évaluer la qualité de votre modèle.

```
from sklearn.linear_model import LinearRegression
model = LinearRegression()
model.fit(x, y)  # apprentissage
y_pred = model.predict(x) # prediction
```

La fonction **PolynomialFeatures** du module **sklearn.preprocessing** génère une nouvelle matrice de features composée de toutes les combinaisons polynômiales des features de degré inférieur ou égale au degré indiqué. Pour degree=2 avec deux features x1 et x2, cette fonction renvoie [1, x1, x2, x1x2, $x1^2$, $x2^2$].

```
from sklearn.preprocessing import PolynomialFeatures
polynomial_features= PolynomialFeatures(degree=1) # polynomial degree
X = polynomial_features.fit_transform(x)
```

Nous utilisons aussi la fonction **train_test_split** de la librairie **sklearn.model_selection** pour séparer les données en deux parties : une partie pour le training et l'autre pour le test.

```
from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_test = train_test_split(x, y, random_state=4)
```

On considère les données non linéaires suivantes :

```
np.random.seed(0)
x = np.sort(7 * np.random.rand(80, 1), axis=0)
y = np.sin(x).ravel() + 0.1*np.random.normal(0,1,len(x))
```

Le but est d'appliquer ce que nous avons vu dans les deux parties précédentes.

On suit le plan suivant :

- Visualiser les données
- Séparer les données en deux parties (training set and test set)
- Faire l'apprentissage du modèle `LinearRegression` sur les données du training set
- Evaluer le modèle obtenu sur les données du test set
- Faire augmenter le degré du polynôme, puis refaire l'apprentissage et l'évaluation
- Comparer les résultats des modèles obtenus
- Visualiser les données de training set et testset de deux couleurs différentes
- Tracer les courbes de régression correspondantes à chaque modèle.