Régressions linéaire et polynômiale avec Python

Exercice 1 (régression linéaire) : Le premier TP est un extrait et une adaptation du cours en ligne de Guillaume Saint-Cirgue. Il consiste à réaliser une première régression linéaire avec le langage Python. Pour cela, il faut effectuer les 4 étapes principales de cette réalisation. Une étape supplémentaire est nécessaire pour évaluer les performances de notre modèle. L'environnement de programmation utilisé est Anaconda/Jupyter. Pour ce premier TP, les étapes ainsi que les commandes nécessaires sont décrites.

Première étape : Réalisation d'un *dataset* : créer un *dataset* aléatoire de 1500 lignes et une colonne ainsi qu'un vecteur résultat/label/target/étiquette.

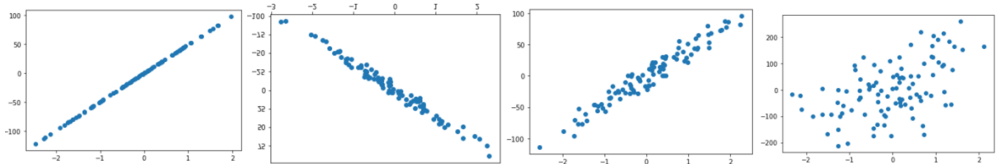
Importation des librairies nécessaires pour le calcul matriciel (numpy), la régression linéaire (*make_regression*) et les courbes (*matplotlib*) :

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

Création et affichage de *dataset*. Ici 200 échantillons avec une seule caractéristique/feauture est créé. Le bruit correspond à l'application d'une perturbation sur les données représentées par une droite (régression linéaire).

```
x, y = make_regression (n_samples=200, n_features=1, noise=10)
plt.scatter(x,y)
```

Exemples avec de bruits 0, 5, 10 et 80[1] :



Il faut s'assurer que les dimensions de nos données x et y sont cohérents avec nos attentes. Donc on doit faire une vérification de leurs dimensions :

```
print (x.shape)
print (y.shape) ➔ problème ? alors voir ci-après
y=y.reshape(y.shape[0], 1)
print(y.shape)
```

[1] Les deux premiers ensembles de figures concernent un dataset de 1500 échantillons.

En effet, la valeur de retour de *make_regression ()* pour *y*, est d'un tableau de m valeurs mais sa forme est incomplète : (*n_samples,*). Pour plus de précision voir ici. Pour compléter les dimensions de y, il suffit d'appliquer la commande *y=y.reshape(y.shape[0], 1)* qui précise que les dimensions de y : nombre de lignes= n_samples et nombre de colonnes=1.

Création de la matrice de *dataset* (tableau 2D contenant la colonne de x et une colonne supplémentaire constituée que de 1) :

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

Initialisation des paramètres *a* et *b* de la régression linéaire (un tableau 1D de 2 éléments *a* et *b* - appelé vecteur *theta*) :

```
theta=np.random.randn(2,1)
theta
```

Deuxième étape ; définition de modèle et son affichage :

Définition du modèle régression linéaire :

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

Tester notre modèle:

```
model (X, theta)
```

Affichage de l'application du modèle sur le *dataset* :

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

On remarque qu'avec le *theta* choisi aléatoirement (*a* et *b* choisis aléatoirement), le modèle ne répond pas du tout à nos attentes. Autrement dit, il ne représente pas du tous nos données (notre nuage de points). Comment peut-on améliorer le modèle ? La réponse est qu'il faut améliorer les paramètres initiaux *a* et *b* (autrement dit le *theta* intial). Cette amélioration peut se faire en calculant la fonction d'erreur (de coût). Nous avons vu en cours comment minimiser cette fonction par l'algorithme de descente de gradient.

Troisième étape ; définition de fonction de coût (on utilisera cette fonction lors d'évaluation des performances de notre modèle).

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

Quatrième étape ; algorithme de descente de gradient :

Calcul de dérivée (gradient) :

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

Algorithme itératif de descente de gradient :

```
def DG (X,y, theta, learning_rate, n_iterations):
    for i in range(0, n_iterations):
        theta=theta-learning_rate*grad(X,y, theta)
        return theta
```

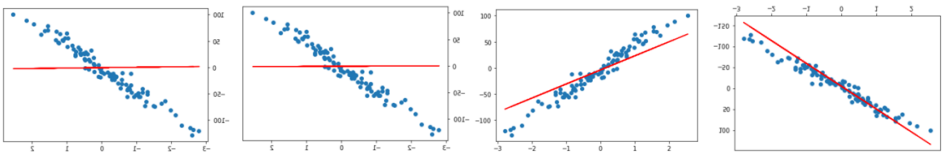
Le test de DG avec notre modèle sur dataset :

```
thetaF=DG(X,y, theta, learning_rate =0.001, n_iterations=1000)
thetaF
```

Affichage de la prediction :

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

avec *n_iterations=1000* et *learning_rate =0.001, 0.01, 0.5, 1*, nous avons respectivement :



Évaluation de performances :

Évolution de coûts/erreurs : Les détails du calcul des dérivées (gradients) sont donnés à la page suivante. Deux méthodes d'évaluer les performances de notre modèle sont données par la suite. La première consiste à regarder comment nous arrivons à diminuer l'erreur de notre au fur et à mesure des 1000 itérations de l'algorithme de descente de gradient. Pour cela, il suffit de calculer l'erreur à chaque itération et dessiner sa variation au cours des itérations :

```
def DG (X,y, theta, learning_rate, n_iterations):
    histCout=np.zeros(n_iterations)
    for i in range(0, n_iterations):
        theta=theta-learning_rate*grad(X,y, theta)
        histCout[i]= cout(X,y,theta)
    return theta, histCout
```

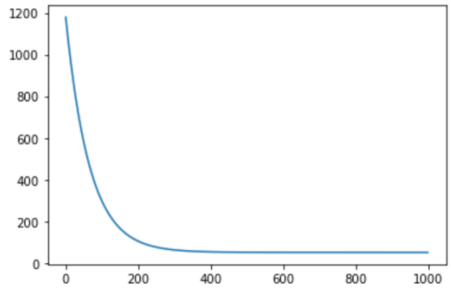
Voici les commandes pour tester l'algorithme de descente de gradient pendant 1000 itérations avec un taux d'apprentissage 0.01, tout en calculant l'erreur à chaque itération (mise dans le tableau *histCout*) :

```
thetaF, histCout=DG(X,y, theta, learning_rate =0.01, n_iterations=1000)
```

thetaF, histCout

En dessinant la courbe de variation d'erreur au cours des itérations, on obtient (pour notre cas) :

plt.plot(range(1000),histCout)



On remarque la diminution des erreurs au cours des itérations et qu'environ à partir de l'itération 350, l'erreur est stabilisée et il ne sert à rien de faire des itérations supplémentaires.

Coefficient de détermination : Une autre métrique pour évaluer les performances d'un modèle est le coefficient de détermination. Ce coefficient est le R^2défini par

$$R^2 = 1 - \frac{\sum(f(x)-y)^2}{\sum(\bar{y}-y)^2}$$

Où $\bar{y}$ est la moyenne des labels de nos échantillons. Plus ce coefficient (qui donne le degré de confiance) est proche d'un, meilleur est notre modèle. La fonction suivante calcule ce coefficient :

```
def coefDet(y, prediction):
    u=((y-prediction)**2).sum()
    v=((y-y.mean())**2).sum()
    return 1-u/v
```

Alors, l'appel à la fonction : *coefDet(y, prediction)*, nous retourne : 0.9554333976514042. Ce qui est assez proche de 1.

Exercice 2 (régression linéaire et polynômiale) : Considérons le *dataset* présenté à la fin de l'exercice. Il s'agit de décrire la corrélation entre le taux de DDT d'un Brochet (variable à prédire) et l'âge du Brochet (variable prédictive).

1. Définissez le tableau 2D dont la 1ère colonne représente l'âge des Brochets et la 2ème des 1''. Dessinez le nuage de points qui représente le taux de DDT en fonction de l'âge des Brochets.
2. Appliquez un modèle de régression linéaire à ce nuage de point.
3. Quel algorithme d'apprentissage utilisez-vous pour la minimisation de la fonction de coût ?
4. Évaluez les performances de votre modèle.
5. Est-ce la régression linéaire est un bon modèle pour la prédiction de taux de DDT en fonction de l'âge des Brochets ? Utilisez un modèle de régression polynomiale (la plus simple : polynôme de seconde degré). Pour cela, il suffit qu'on adapte le *dataset* et le nombre de coefficient de régression (a, b et c). Le reste de l'algorithme reste le même.

Quelle est votre conclusion sur le choix de ces modèles ? Quel est le rôle de *dataset* sur ce choix ?

Exercice 3 (régression polynomiale) : Considérons un *dataset* comme celui présenté à l'exercice 1, en y ajoutant une dose de "non-linéarité". Appliquez à ce *dataset* une régression polynômiale de degré 2.

1. **Création de *dataset*** : Vous pourrez utiliser la méthode suivante pour créer un dataset de 200 échantillons :

   ```
   x, y = make_regression (n_samples=200, n_features=1, noise=10)
   y = y + abs(y/2)
   ```

   Dessinez le nuage de points représentant ce *dataset*.

2. **Création de modèle** : Un modèle de régression linéaire convient-il à ce nuage de point ? Argumentez votre réponse. Que proposez-vous comme modèle de régression ?

   Supposons que le modèle polynômial est de second degré ($a.x^2 + b.x + c$). On peut alors constituer la matrice du modèle avec les instructions suivantes :

   ```
   X=np.hstack((x, np.ones (x.shape)))
   X=np.hstack((x**2, X))
   ```

3. **Fonction de coût et algorithme d'apprentissage :** Quel algorithme d'apprentissage utilisez-vous pour la minimisation de la fonction de coût (la recherche des valeurs optimales pour a, b, et c) ?
4. **Évaluation de performances** : Évaluez les performances de votre modèle en dessinant la courbe d'erreur et en calculant le coefficient de détermination.

Exercice 4 (régression linéaire multiple) : En suivant la même démarche que celle de l'exercice 1, créez un *dataset* d'un ensemble d'échantillons dont chacun possède deux caractéristiques/*features*. Adaptez les tableaux et exécutez les instructions de l'exercice 1. Dessinez les courbes de performance de la prédiction en fonction de chacun des *features* ainsi qu'en fonction des deux *features* en même temps (plot en 3D).

Exercice 5 (régression linéaire multiple) : Il s'agit ici de prédire la qualité de vins en utilisant ses caractéristiques chimiques et/ou physiques. Les données peuvent être téléchargées du site : http://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-white.csv

Les variables d'entrée sont les caractéristiques et les sorties sont les labels/étiquettes correspondant à la bonne ou mauvaise qualité de vin.

Annexes

Annexe 1 : Calcul des dérivées (régressions linéaire et polynômiale)

Régression linéaire (RL) :

Soit $f(x) = a.x + b$,. Alors, $f'_x = a$.

Soit $f(x) = a.x + b$,. Alors, $f'_a = x$ et $f'_b = 1$.

Soit $f(x) = x^2$, alors $f'_x = 2.x.x'$

Soit $f(x) = (a.x + b)^2$, alors $f'_a = 2.(a.x + b).(a.x + b)'_a = 2.x.(a.x + b)$ et

$$f'_b = 2.(a.x + b).1 = 2.(a.x + b)$$

$$J(a,b)=\frac{1}{2.m}\sum_{i=1}^{i=m}(a.x^{(i)} + b - y^{(i)})^2$$

$$\frac{\partial J(a,b)}{\partial a} = \frac{1}{m}\sum_{i=1}^{i=m}(a.x^{(i)} + b - y^{(i)}).x^{(i)}$$

$$\frac{\partial J(a,b)}{\partial b} = \frac{1}{m}\sum_{i=1}^{i=m}(a.x^{(i)} + b - y^{(i)})$$

Régression polynômiale simple (de degré 2) :

Soit $f(x) = a.x^2 + b.x + c$,. Alors, $f'_a = x^2$, $f'_b = x$ et $f'_c = 1$

$$J(a,b)=\frac{1}{2.m}\sum_{i=1}^{i=m}(a.x^{(i)^2} + b.x^{(i)} + c - y^{(i)})^2$$

Avec,

$$\frac{\partial J(a,b)}{\partial a} = \frac{1}{m}\sum_{i=1}^{i=m}2.x^{(i)^2}(a.x^{(i)^2} + b.x^{(i)} + c - y^{(i)})$$

$$\frac{\partial J(a,b)}{\partial b} = \frac{1}{m}\sum_{i=1}^{i=m} x^{(i)}.(a.x^{(i)^2} + b.x^{(i)} + c - y^{(i)})$$

$$\frac{\partial J(a,b)}{\partial c} = \frac{1}{m}\sum_{i=1}^{i=m} (a.x^{(i)^2} + b.x^{(i)} + c - y^{(i)})$$

Pour simplifier les calculs, comme pour la régression linéaire on peut transformer nos calculs de gradients en calcul matriciels. Nous aurons ainsi, $F = X.\theta$ où $\theta = (a,b,c)^T$ et X étant une matrice a 3 colonnes et m lignes. Les 1ère, 2ème et 3ème colonne sont constituées des $x^{(i)^2}$, $x^{(i)}$ et 1 respectivement.