Machine Learning
Quelques algorithmes

Said Gounane

05/02/2022

Contents

1 Regression linéaire

- **Hypothèses :** $h_\theta(x) = \theta_0 + \theta_1 x$
- **Paramètres du model** : θ_0, θ_1
- **Fonction du coût :**

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=0}^{m} (h_\theta(x(i)) - y(i))^2$$

- **Objectif :** Trouver les meilleurs paramètres θ_0 et θ_1 revient à minimiser (trouver le minimum) de la fonction du coût.

$$\min_{\theta_0, \theta_1} J(\theta_0, \theta_1)$$

1.1 La méthode des moindres carée

Théorème : Si la variance $Var(X)$ de la série statistique $X = (x_i)$ est non-nulle, il existe une unique droite qui minimise la quantité $J(a, b)$. Elle vérifie

$$a = \frac{Cov(X, Y)}{Var(X)} \text{ et } b = \bar{Y} - a\bar{X},$$

où $Cov(X, Y)$ désigne la covariance de X et de Y, $\bar{X}$ la moyenne de (x_i) et $\bar{Y}$ la moyenne de (y_i).

$$COV(x, y) = \frac{\sum (x_i - \overline{X})(y_i - \overline{Y})}{n - 1}$$

1.2 La méthode de la déscente du gradient

- Initialize randomly the values of θ_0, θ_1
- Keep chaging θ_0,θ_1 to reduce $J(\theta_0, \theta_1)$ until we hopefully end up at a minimum

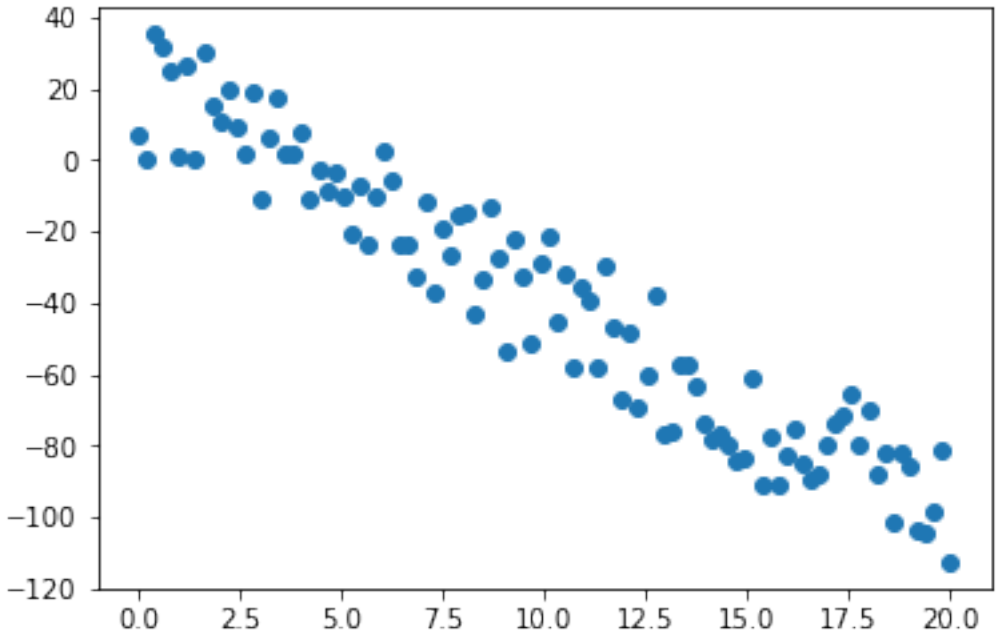
$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

- return θ_0, θ_1
- fin algorithm

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

[84]:
```
a=-6
b=20
X=np.linspace(0,20,100)
h=a*X+b
Y=h+np.random.uniform(-20,20,size=100)
plt.scatter(X,Y)
```

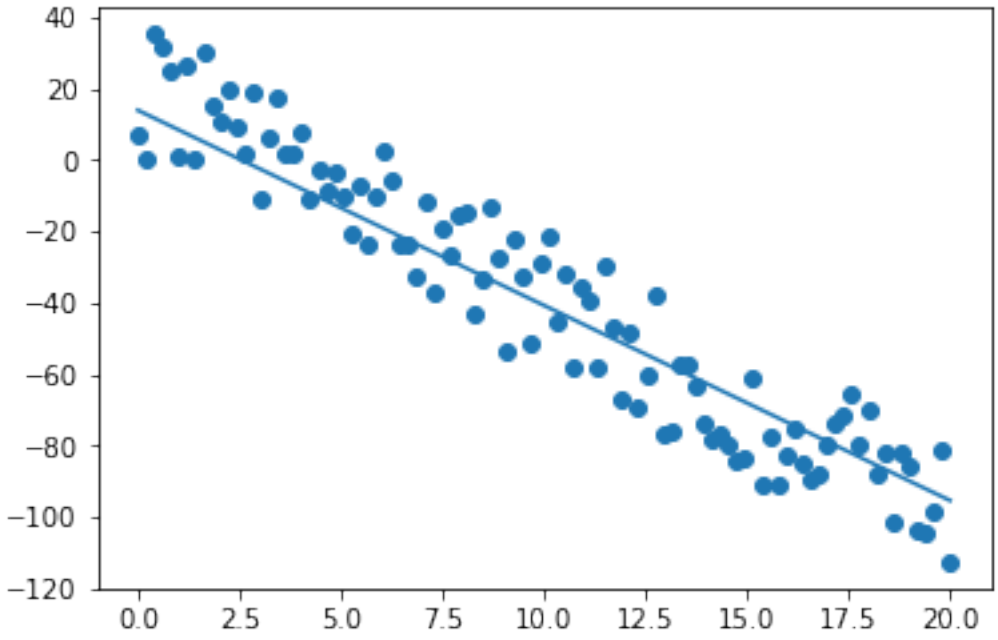
[84]:
```
<matplotlib.collections.PathCollection at 0x7fd870402700>
```

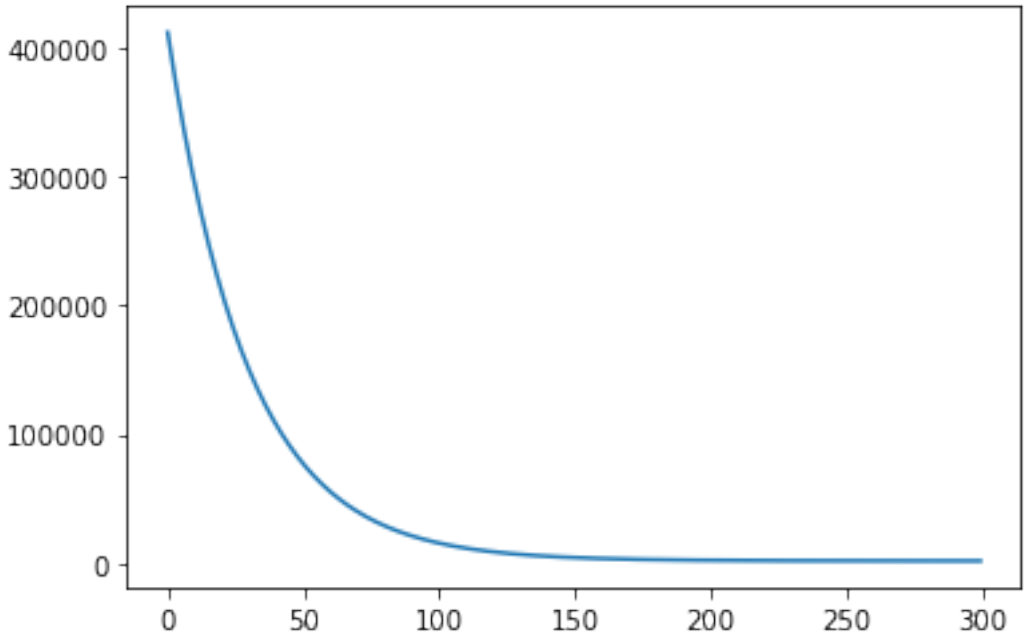


[91]:
```
Jh=[]
alpha=0.00001
a=10
b=15
for i in range(300):
    Yh=a*X+b
    E=Y-Yh
    J=E.dot(E)/8
    dJa=-X.dot(E)/8
    dJb=-sum(E)/8
    a=a-alpha*dJa
    b=b-alpha*dJb
```

```
    Jh.append(J)
plt.scatter(X,Y)
plt.plot(X,a*X+b)
plt.figure()
plt.plot(Jh)
print(a,b)
```

```
-5.467253618160841 13.935031971367351
```





[64]:
```
x=np.array([20, 22, 18, 7, 7, 11, 17, 15])
y=np.array([10, 15, 60, 100, 120, 90, 78, 12])
X=np.matrix([np.ones(x.shape[0]),x]).T
Y=np.matrix(y).T
w=np.linalg.inv(X.T.dot(X)).dot(X.T).dot(Y)
w
```

[64]:
```
matrix([[152.69331158],
        [ -6.29526917]])
```

[68]:
```
plt.scatter(x,y)
a=w[0,0]
b=w[1,0]
print(b,a)
plt.plot(x,a*x+b)
```

```
-6.295269168026092 152.6933115823815
```

[68]:
```
[<matplotlib.lines.Line2D at 0x7fd853af41f0>]
```

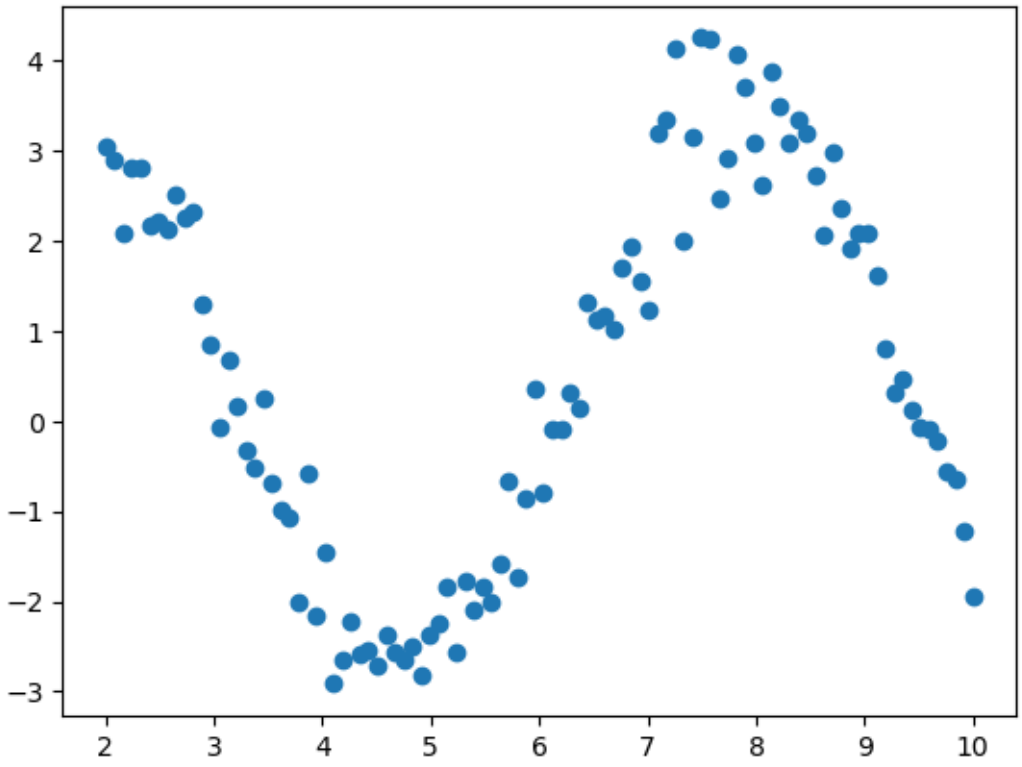
2 Régression Polynomial

[63]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
```

```
from sklearn.model_selection import train_test_split

w=np.array([0.4,0.002,3])
x=np.linspace(2,10,100)
y=w[0]+w[1]*x+w[2]*np.sin(x)+np.random.randn(100)*0.43
plt.scatter(x,y)
```

[63]: <matplotlib.collections.PathCollection at 0x145e34200>



[ ]:

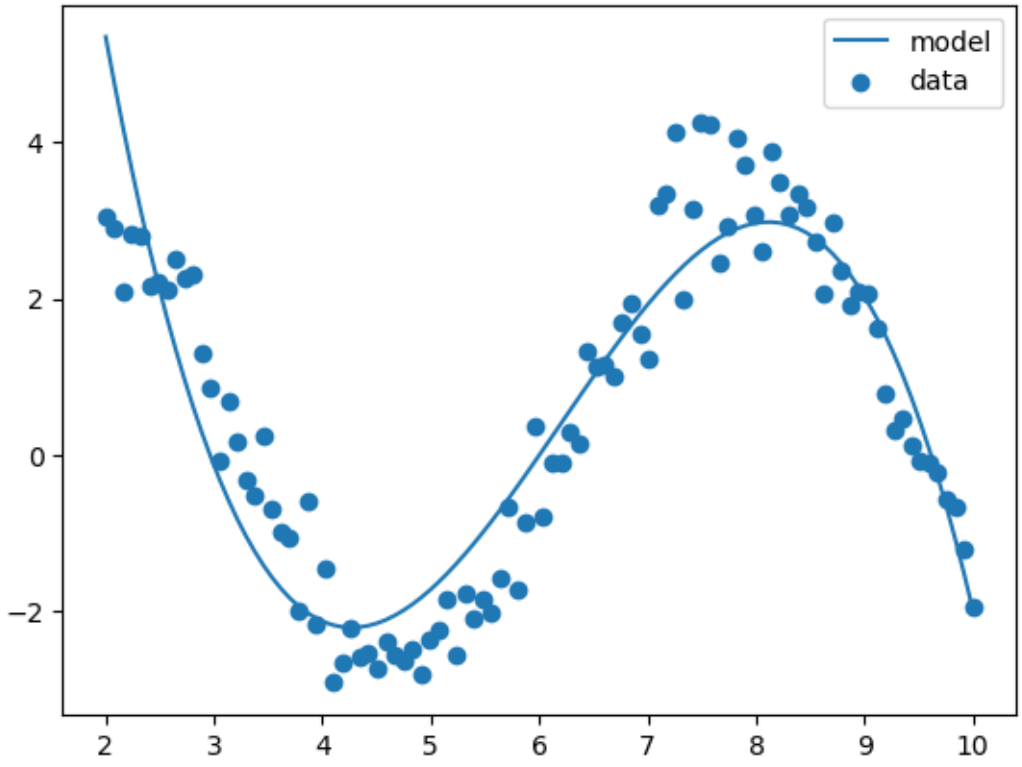
[64]:
```
X=np.array([x,x*x,x*x*x]).T
X_train,X_test ,y_train, y_test=train_test_split(X,y,test_size=0.2,␣
 ↪random_state=42)
model=LinearRegression().fit(X_train,y_train)
W=np.array([model.intercept_, model.coef_[0],model.coef_[1],model.coef_[2]])
W
```

[64]: array([31.20012046, -18.97735499,   3.39157251, -0.18256535])

[65]:
```
yy=W[0]+W[1]*x+W[2]*x*x+W[3]*x*x*x
plt.plot(x,yy,label="model")
```

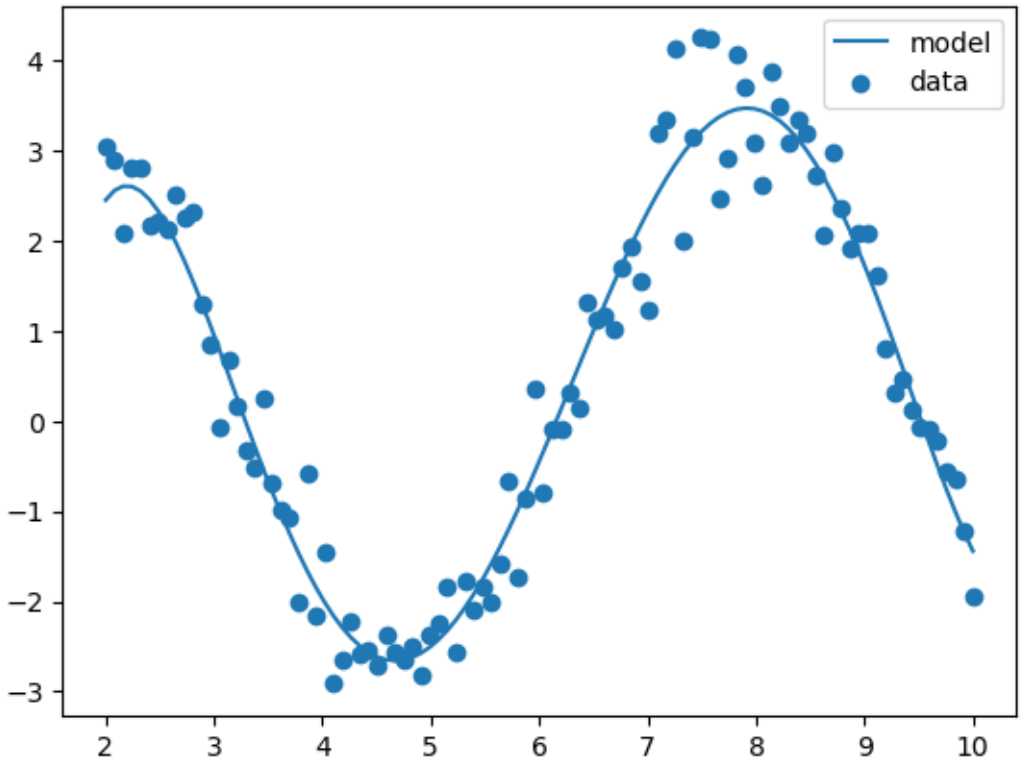
```
plt.scatter(x,y,label="data")
plt.legend()
```

[65]: <matplotlib.legend.Legend at 0x145eaa540>



[66]:
```
X=np.array([x,x*x,x*x*x,np.power(x,4),np.power(x,5)]).T
X_train,X_test ,y_train, y_test=train_test_split(X,y,test_size=0.2,␣
 ↪random_state=42)
model=LinearRegression().fit(X_train,y_train)
yy=model.intercept_+model.coef_[0]*x + model.coef_[1]*x*x + model.coef_[2]*x*x*x␣
 ↪+ model.coef_[3]*x*x*x*x + + model.coef_[4]*x*x*x*x*x
plt.plot(x,yy,label="model")
plt.scatter(x,y,label="data")
plt.legend()
```

[66]: <matplotlib.legend.Legend at 0x145e63740>



[67]:
```
model.score(X_test,y_test)
```

[67]:
```
0.9638511872793611
```

[68]:
```
#predire la sortie pour x=12
xx=4.5
s=model.predict([[xx,xx*xx,np.power(xx,3),np.power(xx,4),np.power(xx,5)]])
s
```

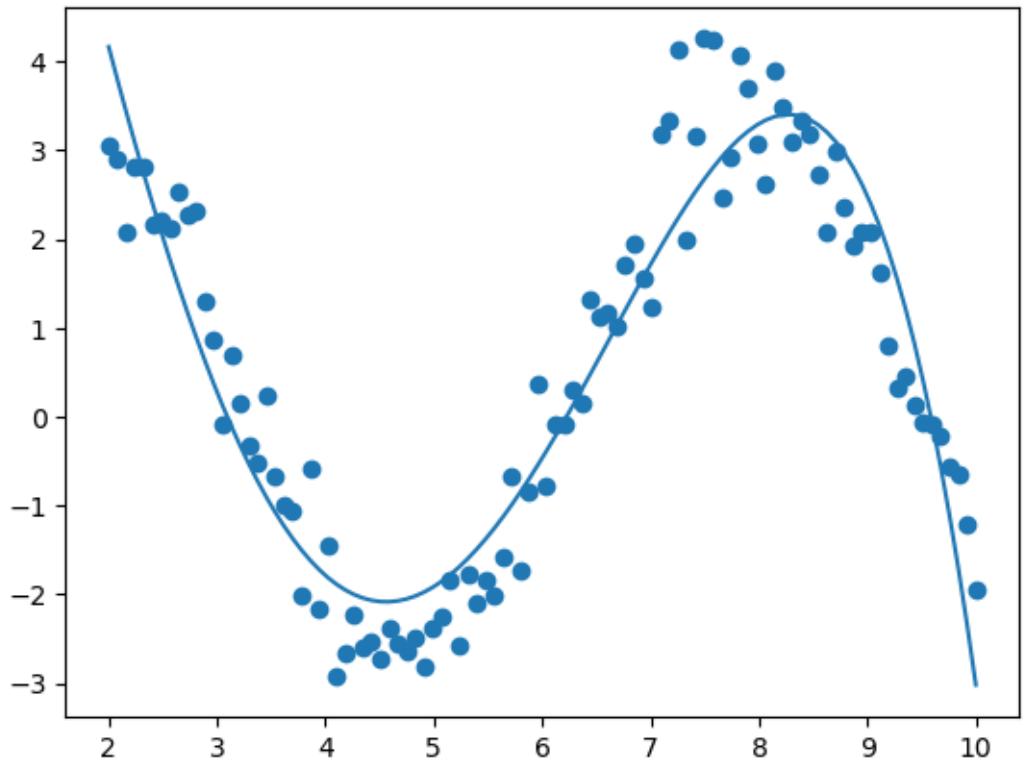
[68]:
```
array([-2.61317718])
```

[89]:
```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LinearRegression
model= Pipeline([
 ('poly', PolynomialFeatures(degree=4)),
 ('linear', LinearRegression(fit_intercept=False))
]).fit(np.array([x[:]]).T,y)

yy=model.predict(np.array([x[:]]).T)
plt.plot(x,yy)
plt.scatter(x,y)
```

```
#print(np.array([x[:]]).shape,y.shape)
```

[89]: <matplotlib.collections.PathCollection at 0x146bacc50>



2.1 House price predection

[61]:
```
import pandas as pd
df=pd.read_csv("data.csv")
df.head()
```

[61]:

	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	\
0	2014-05-02 00:00:00	313000.0	3.0	1.50	1340	7912	
1	2014-05-02 00:00:00	2384000.0	5.0	2.50	3650	9050	
2	2014-05-02 00:00:00	342000.0	3.0	2.00	1930	11947	
3	2014-05-02 00:00:00	420000.0	3.0	2.25	2000	8030	
4	2014-05-02 00:00:00	550000.0	4.0	2.50	1940	10500	

	floors	waterfront	view	condition	sqft_above	sqft_basement	yr_built	\
0	1.5	0	0	3	1340	0	1955	
1	2.0	0	4	5	3370	280	1921	
2	1.0	0	0	4	1930	0	1966	
3	1.0	0	0	4	1000	1000	1963	

```
4     1.0           0      0           4        1140            800       1976

   yr_renovated                    street       city  statezip country
0          2005      18810 Densmore Ave N  Shoreline  WA 98133     USA
1             0           709 W Blaine St    Seattle  WA 98119     USA
2             0  26206-26214 143rd Ave SE       Kent  WA 98042     USA
3             0           857 170th Pl NE   Bellevue  WA 98008     USA
4          1992         9105 170th Ave NE    Redmond  WA 98052     USA
```

3 Regression Logistique

$$h_w(x) = \frac{1}{1 + e^{-wx}}$$

$$J(w) = -\frac{1}{m}\sum_{i=1}^{m}(y^i log(h_w(x^i)) + (1 - y^i)log(1 - h_w(x^i))$$

$$w_j := w_j - \alpha\frac{\delta J(w_j)}{\delta w_j} = w_j - \frac{\alpha}{m}\sum_{i=1}^{m}(h_w(x^i) - y_i)x_j^i$$

x^i: le i ieme exemple

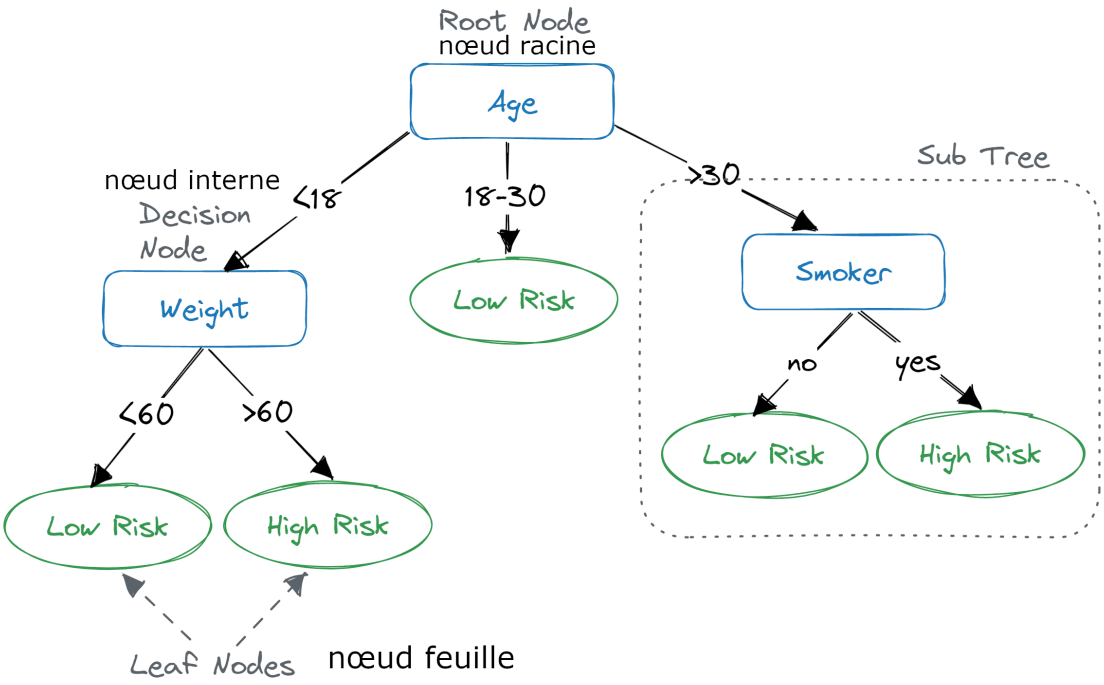
x_j: La j eme composante de x

4 Arbres de decision

4.1 1-Introduction

Un arbre de décision est une structure arborescente de type organigramme dans laquelle un **nœud interne** représente une **caractéristique** (ou un attribut), la **branche** représente une **règle de décision** et chaque **nœud feuille** représente le résultat.

Le nœud le plus élevé d'un arbre de décision est appelé **nœud racine**. Il apprend à partitionner sur la base de la valeur de l'attribut. Il partitionne l'arbre de manière récursive, appelée partition récursive.

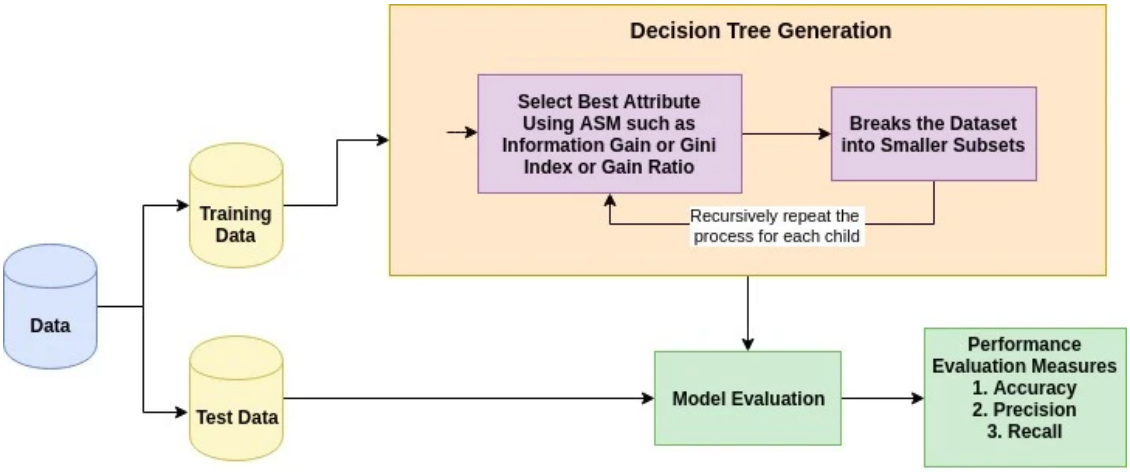


Un arbre de décision est un type d'algorithme ML à boîte blanche. Il partage une logique décisionnelle interne, qui n'est pas disponible dans les algorithmes de type "boîte noire", tels que les réseaux neuronaux. Son temps d'apprentissage est plus rapide que celui de l'algorithme de réseau neuronal.

La complexité temporelle des arbres de décision est fonction du nombre d'enregistrements et d'attributs dans les données données données. Les arbres de décision peuvent traiter des données de haute dimension avec une bonne précision.

4.2 2-Comment fonctionne l'algorithme de l'arbre de décision ?

L'idée de base de tout algorithme d'arbre de décision est la suivante : 1. Sélectionner le meilleur attribut à l'aide des mesures de sélection d'attributs (Attribute Selection Measures- ASM) pour diviser les données. 2. Faire de cet attribut un nœud de décision et diviser l'ensemble de données en sous-ensembles plus petits. 3. Commencer la construction de l'arbre en répétant ce processus de manière récursive pour chaque enfant jusqu'à ce que l'une des conditions corresponde : - Tous les tuples appartiennent à la même classe. - Il n'y a plus d'attributs restants. - Il n'y a plus d'instances.



4.3 3-Mesures de sélection d'attributs (ASM)

La mesure de sélection d'attributs (Attribute Selection Measures- ASM) permet de sélectionner le critère de division qui partitionne les données de la meilleure manière possible.

Elle est également connue sous le nom de règles de division, car elle nous aide à déterminer les points de rupture pour les tuples sur un nœud donné.

ASM attribue un rang à chaque caractéristique (ou attribut) en expliquant l'ensemble de données donné.

L'attribut ayant le meilleur score sera sélectionné comme attribut de partitionnement.

Dans le cas d'un attribut à valeur continue, les points de séparation des branches doivent également être définis.

Les mesures de sélection les plus courantes sont le **gain d'information**, le **ratio de gain** et l'**indice de Gini**.

4.3.1 a-Gain d'information

Le gain d'information est la diminution de l'entropie. L'entropie mesure l'impureté de l'ensemble des données d'entrée. En théorie de l'information, elle désigne l'impureté d'un groupe d'exemples.

Le gain d'information calcule la différence entre l'entropie avant la division et l'entropie moyenne après la division de l'ensemble de données sur la base de valeurs d'attributs données.

L'entropie de l'ensemble S est définie comme suit: 1. Aavant le partitionnement

$$H(S) = -\sum_{i=1}^{m} p_i Log_2(p_i)$$

Où p_i est la probabilité qu'un exemple arbitraire dans S appartienne à la classe C_i.

2. Après le partitionnement à base de valeurs d'un attributs A:

$$H_A(S) = \sum_{j=1}^{v} \frac{|S_j|}{|S|} H(S_j)$$

Le Gain d'information:

$$Gain(S, A) = H(S) - H_A(S)$$

L'attribut A ayant le **gain d'information le plus élevé**, $Gain(S, A)$, est choisi comme attribut de séparation.

L'algorithme d'arbre de décision ID3 (Iterative Dichotomiser) utilise le gain d'information.

4.3.2 b-Rapport de gain

Le gain d'information préfère l'attribut ayant un grand nombre de valeurs de sortie distinctes. Par exemple, considérons un attribut avec un identifiant unique, qui a zéro $ H(S)$ en raison d'une partition pure. Cela maximise le gain d'information et crée un partitionnement inutile.

C4.5, une amélioration d'ID3, utilise une extension du gain d'information connue sous le nom de ratio de gain (gain ratio). Le rapport de gain traite le problème du biais en normalisant le gain d'information à l'aide de Split Info.

$$splitInfo_A(S) = -\sum_{j=1}^{v} \frac{|S_j|}{|S|} log_2(\frac{|S_j|}{|S|})$$

$$GainRatio(S, A) = \frac{Gain(S, A)}{splitInfo_A(S)}$$

L'attribut présentant le **rapport de gain le plus élevé** est choisi comme attribut de séparation.

4.3.3 c-Gini Index

L'algorithme d'arbre de décision, CART (Classification and Regression Tree), utilise la méthode de Gini pour créer des points de séparation.

$$Gini(S) = 1 - \sum_{i=1}^{m} p_i^2$$

L'indice de Gini considère une sépartition binaire pour chaque attribut. Vous pouvez calculer une somme pondérée de l'impureté de chaque partition. Si une division binaire sur l'attribut A divise les données S en S1 et S2, l'indice de Gini de S est :

$$Gini_A(S) = \frac{|S_1|}{|S|} Gini(S1) + \frac{|S_2|}{|S|} Gini(S2)$$

$$GI(S, A) = Gini(S) - Gini_A(S)$$

L'attribut ayant **l'indice de Gini le plus faible** est choisi comme attribut de partitionnement.

4.4 4-Exemple

Definir l'attribut de partitionnement des exemples dans le tableau ci-dessous, en utilisant les 3 mesures de sélection d'attributs: Gain d'information, rapport de gain et Gini index.

Jour	Attributs des exemples				Classe
	Prévisions	Température	Humidité	Vent	
1	Ensoleillé	Chaud	Élevée	Faible	Non
2	Ensoleillé	Chaud	Élevée	Fort	Non
3	Nuageux	Chaud	Élevée	Faible	Oui
4	Pluvieux	Moyen	Élevée	Faible	Oui
5	Pluvieux	Frais	Normale	Faible	Oui
6	Pluvieux	Frais	Normale	Fort	Non
7	Nuageux	Frais	Normale	Fort	Oui
8	Ensoleillé	Moyen	Élevée	Faible	Non
9	Ensoleillé	Frais	Normale	Faible	Oui
10	Pluvieux	Moyen	Normale	Faible	Oui
11	Ensoleillé	Moyen	Normale	Fort	Oui
12	Nuageux	Moyen	Élevée	Fort	Oui
13	Nuageux	Chaud	Normale	Faible	Oui
14	Pluvieux	Moyen	Élevée	Fort	Non

Ensemble d'exemples pour playTennis

[2]:
```
import numpy as np
```

[5]:
```
S=np.array([[0,0,1,0,0],
            [0,0,1,1,0],
            [1,0,1,0,1],
            [2,1,0,1,1],
            [2,2,0,0,1],
```

```
            [2,2,0,1,0],
            [1,2,0,1,1],
            [0,1,1,0,0],
            [0,2,0,0,1],
            [2,1,1,1,0]
           ])
S.shape
```

[5]: (10, 5)

[ ]:

5 Random Forest

- Random forest signifie « forêt aléatoire ». Proposé par Leo Breiman en 2001,
- c'est un algorithme qui se base sur l'assemblage d'arbres de décision.
- Il est assez intuitif à comprendre, rapide à entraîner et il produit des résultats généralisables.

5.1 I- Principe de fonctionnement du random forest

Un random forest est constitué d'un ensemble d'arbres de décision indépendants.

Chaque arbre dispose d'une vision parcellaire du problème du fait d'un:

- **Tree bagging** : un tirage aléatoire avec remplacement sur les exemples.
- **Feature sampling** : un tirage aléatoire sur les variables.

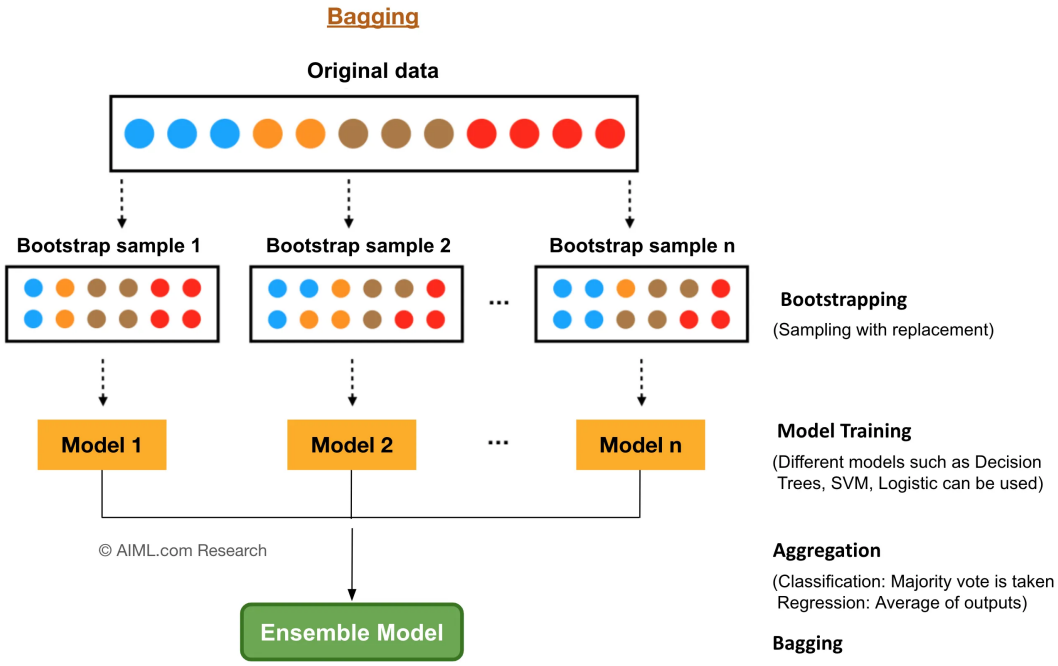
A la fin, tous ces arbres de décisions indépendants sont assemblés. La prédiction faite par le random forest pour des données inconnues est alors la moyenne (ou le vote, dans le cas d'un problème de classification) de tous les arbres.

$$Random forest : tree bagging + feature sampling$$

5.1.1 1- Tree bagging

Le Bagging signifie "bootstrap aggregation". C'est un processus de tirage aléatoire sur les observations (lignes de données) déterminé par 3 étapes clés :

- Construction de n arbres de décisions en tirant aléatoirement n échantillons d'observations,
- Entraînement de chaque arbre de décision,
- Pour faire une prévision sur de nouvelles données, il faut appliquer chacun de n arbres et prendre la majorité parmi les n prévisions.



2-

Feature sampling

C'est un processus de tirage aléatoire sur les variables (Features). Par défaut, on tire $\sqrt{n}$ variables pour un problème à n variables au total.

5.2 II- Exemple

[28]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
import numpy as np
```

[29]:
```
X, y = make_classification(n_samples=1000, n_features=4,n_informative=2,␣
 ↪n_redundant=0,random_state=0, shuffle=False)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33,␣
 ↪random_state=42)
```

1- A partire du dataset (X,y) Créer 10 sous ensemble en utilsant le Bootstraping

[70]:
```
Xs=[]
Ys=[]
for i in range(10):
    featurs = np.random.choice(X.shape[1], int(np.sqrt(X.shape[1])))
    x=X[:,featurs]
    #print(x)
    Xs.append(x.copy())
    Ys.append(y.copy())
    #print(featurs)
```

```
    index_t = np.random.choice(X.shape[0], 20)
    index_s = np.random.choice(X.shape[0], 20)
    Xs[i][index_t,:]=Xs[i][index_s,:].copy()
    Ys[i][index_t]=Ys[i][index_s].copy()
```

[71]:
```
Xs[0]
```

[71]:
```
array([[-0.60362044, -1.66853167],
       [ 1.13181196, -1.42616056],
       [-1.07836109, -1.00842811],
       ...,
       [ 0.409295  , -0.99828435],
       [-0.61980477,  0.98048001],
       [ 0.28791543, -1.28980655]])
```

2- Créer 10 arbres de decisions à 2 features pour ces 10 sous ensembles

[ ]:
```
```

3- Calculer la prediction y_pred du y_test et afficher classification_report

[ ]:
```
```

4- Utilisation directe de la classe RandomForestClassifier sklearn.ensemble

[19]:
```
clf = RandomForestClassifier(max_depth=2,n_estimators=10, random_state=0)
clf.fit(X_train, y_train)
y_pred=clf.predict(X_test)
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
[[152  14]
 [  8 156]]
              precision    recall  f1-score   support

           0       0.95      0.92      0.93       166
           1       0.92      0.95      0.93       164

    accuracy                           0.93       330
   macro avg       0.93      0.93      0.93       330
weighted avg       0.93      0.93      0.93       330
```

5.2.1 II- Faire les mèmes étapes pour la regression

Remarque: Dans un problème de regression, en utilise d'autres metrics que la matrice de confusion (voir: https://scikit-learn.org/stable/modules/model_evaluation.html) comme mean_squared_error

[20]:
```
from sklearn.datasets import make_regression
from sklearn.metrics import mean_squared_error

X, y = make_regression(n_samples=5, n_features=2, noise=1, random_state=42)
```

[ ]:
```

```

[52]:
```
np.random.choice(20,4)
```

[52]:
```
array([17, 12,  4, 11])
```

[ ]:
```

```

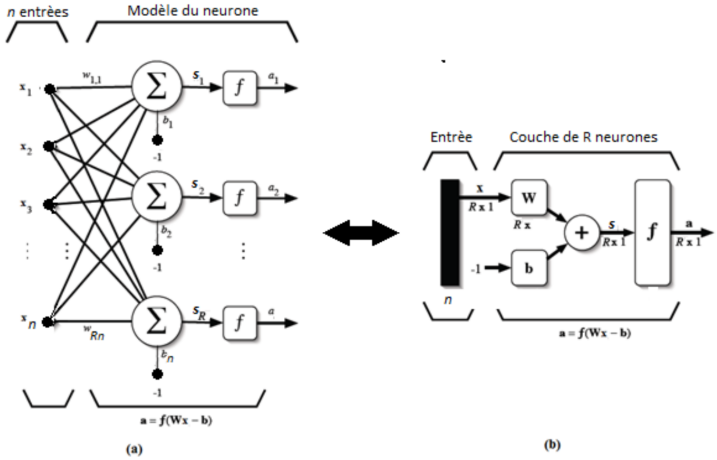
6 Réseaux de neurones

6.1 1. Introduction

- Le perceptron multicouche (PMC) et les fonctions à base radiale (RBF) sont les structures de réseaux de neurones les plus utilisées en classification supervisée, parmi une multitude de structures.
- Ces structures ont pu attirer une grande part d'intérêt dans la recherche et ont été utilisées dans une multitude d'applications dans divers domaines.
- Il a été aussi démontré que ces deux types de réseaux de neurones sont des approximateurs universels, grâce à leurs habilité d'approximer n'importe quelle fonction de n'importe quelle dimension avec une précision arbitraire, à condition de leur présenter un nombre de données d'entraînement suffisant et adéquat et un bon choix de leur structure.

6.2 2. Modèle de neurones et réseaux

L'idée des réseaux de neurones artificiels est inspirée des réseaux de neurones biologiques. Un réseau de neurones biologique est un ensemble de neurones massivement interconnectés. L'unité principale d'un réseau de neurone est le neurone. Ce dernier se compose de quatre parties principales

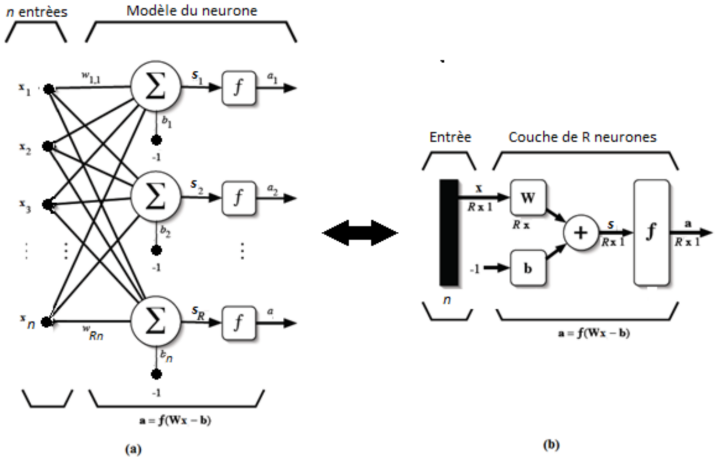


: 1. Les **dendrites** : Ils forment un maillage de récepteurs nerveux qui permettent d'acheminer des signaux électriques en provenance d'autres neurones vers le corps du neurone. 2. Le **soma** : Le corps de la

cellule. Il agit comme une espèce d'intégrateur en accumulant des charges électriques. Lorsque le neurone devient suffisamment excité, il engendre un potentiel électrique qui se propage à travers son axone pour exciter d'autres neurones. 3. L'**axone** : Sert à transmettre les informations vers d'autres neurones ; 4. Les **synapses** : Une interface entre l'axone du neurone et les dendrites des autres neurones. La fonction précise d'un réseau de neurones biologique est essentiellement déterminée par l'arrangement spatial des neurones et de leur axone, ainsi que la qualité des connexions synaptiques individuelles.

6.3 3. Le modèle d'un neurone

Un modèle d'un neurone artificiel est montré dans la figure ci-dessous. Il se compose essentiellement d'un ensemble d'entrées x_i muni d'un poids pour chacune, une unité de calcul représentant le corps du neurone et une seule sortie.

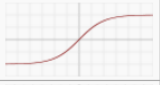


La taille du vecteur d'entrée est souvent augmentée en ajoutant une autre entrée $x_0 = -1$ de poids b. Cette entrée supplémentaire s'appelle un biais. La sortie du neurone j est la somme des x_i pondérés par les qui alimente l'entrée d'une **fonction d'activation** f. Cette sortie est donnée par l'équation suivante:

$$a = f(W^T X - b) = f\big(\sum_{i=0}^{i=n} w_i x_i\big)$$

Avec $w_0 = b$ et $x_0 = -1$

Il existe plusieurs possibilités pour le choix de la fonction d'activation. Le tableau suivant illustre un ensemble d'exemple des fonctions d'activation utilisables.

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1-f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$

[93]:
```
import numpy as np
```

[94]:
```
#definition de la fonction step
def step(z):
    if(z<0):
        return 0
    return 1
```

[95]:
```
#definition de la fonction logistique
def logistique(z):
    return 1/(1+np.exp(-z))
```

[96]:
```
#definition de la fonction step ReLU
def relu(z):
    if(z<0):
        return 0
    return z
```

[97]:
```
#Definition d'un neurone
def neurone(w,x,b,f):
    return f(w.dot(x)-b)
```

[98]:
```
#Tester le neurone avec les differentes fonctions d'activation
w=np.array([1,4,1])
x=np.array([1,2,1])
out1=neurone(w,x,6,step)
out2=neurone(w,x,6,logistique)
out3=neurone(w,x,6,relu)
print(out3)
```

4

6.3.1 3.1 Entrainement d'un perceptron

Rosenblat a montré qu'un perceptron peut apprendre à produire n'importe quelle séparation linéaire, en utilisant un algorithme d'entraînement où les poids sont aléatoirement déterminés puis modifiés itérativement par l'équation suivante

$$w(t+1) = w(t) + \Delta w$$

avec

$$\Delta w = \eta(y_i - \hat{y}_i)x_i$$

Où est le taux d'apprentissage.

[123]:
```
def train(neurone, x_train, y_train, alpha,epoche):
    w=np.random.uniform(size=x_train.shape[1])
    b=np.random.uniform()
    for i in range(epoche):
        for j in range(len(y_train)):
            yh=neurone(w,x_train[j],b,step)
            E=y_train[j]-yh
            dw=alpha*E*x_train[j]
            w=w+dw

            b=b-alpha*E
    return (w,b)
```

[124]:
```
x_train=np.array([[1,4,5],[2,5,9],[1,-1,6],[6,7,-9]])
y_train=np.array([1,0,0,1])
w,b=train(neurone,x_train,y_train,0.01,100)
```

[127]:
```
neurone(w,x_train[3],b,step)
```

[127]: 1

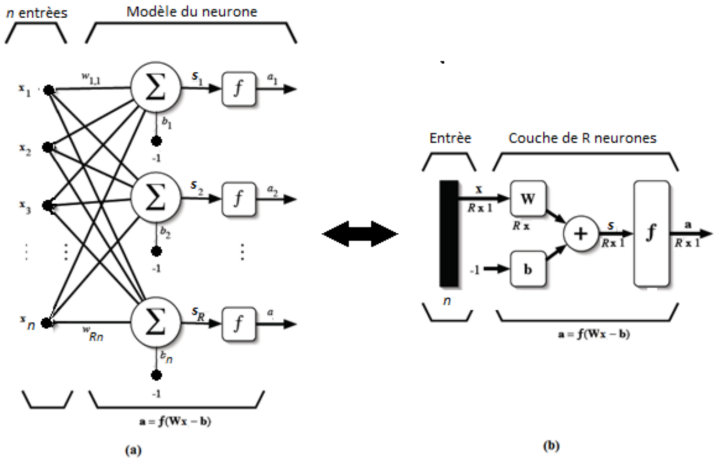
Un seul perceptron n'a pas de grande utilité puisque qu'il ne converge pas dans le cas des problèmes non linéairement séparables.

En revanche, une combinaison appropriée de plusieurs perceptrons est d'une puissance considérable pour ce genre de problèmes.

Une structure bien connue dans le domaine, appelée le perceptron multicouche (PMC), où les neurones sont structurés sous forme de couches et l'information passe d'une couche à l'autre jusqu'à ce qu'elle attient sa destination, a bien montré ses performances de grandes qualités.

6.4 4. le réseau de neurones

En général, un réseau de neurones est un maillage de plusieurs neurones organisés en couches. Chaque couche contient R neurones connectés aux n entrées. On dit alors que la couche est totalement connectée. Un poids est associé à chacune des connexions. L'ensemble des poids d'une couche forme une matrice W de dimension S×R



6.5 5. Le perceptron multicouche (PMC)

[1]:
```
import numpy as np
```

[2]:
```
def step(s):
    if s>0:
        return 1
    else:
        return 0

def logistic(s):
    return 1/(1+np.exp(s))

def relu(s):
    if s>0:
        return s
    else:
        return 0
```

[3]:
```
class Neuron:
    def __init__(self,input_size, activation):
        self.input_size=input_size
        self.activation=activation
```

```
        self.W=np.random.rand(input_size)
        self.b=np.random.rand()

    def forward(self,x):
        x=np.array(x)
        s=x.dot(self.W)+self.b
        a=self.activation(s)
        return a
    #def fit(_self, X,Y)
```

[4]:
```
n0=Neuron(3,lambda s:s)
n1=Neuron(3,step)
n2=Neuron(3,logistic)
n3=Neuron(3,relu)
x=np.array([[1,4,-1]])
print(n0.forward(x))
print(n1.forward(x))
print(n2.forward(x))
print(n3.forward(x))
```

```
[1.91039783]
1
[0.07591876]
[1.27397768]
```

[5]:
```
class Layer:
    def __init__(self, input_size, output_size, activation):
        self.neurons=[]
        for _ in range(output_size):
            self.neurons.append(Neuron(input_size,activation))
    def forward(self,x):
        out=[]
        for neuron in self.neurons:
            out.append(neuron.forward(x))
        return out
```

[6]:
```
l=Layer(3,10,logistic)
l.forward([1,1,-3])
```

[6]:
```
[0.7449584704456438,
 0.49328490846842077,
 0.2995324092976883,
 0.34900488473026675,
 0.7026984239503654,
 0.34196097725374763,
 0.5863275877593777,
 0.1895859464885219,
 0.7020840909021232,
```

```
0.5830739367872044]
```

[7]:
```python
class Nnet:
    def __init__(self, input_size,hidens, output_size,f, activation):
        self.layers=[]
        inputs=input_size
        for n in hidens:
            self.layers.append(Layer(inputs,n,f))
            inputs=n
        self.layers.append(Layer(inputs,output_size,activation))
    def forward(self,x):
        print("L0:",x)
        for l in self.layers:
            out=l.forward(x)
            print("L:",out)
            x=out
        return out
```

[8]:
```python
net=Nnet(3,[2,3],2,relu,relu)
net.forward([1.2,.1,1])
```

```
L0: [1.2, 0.1, 1]
L: [1.3327050910936262, 1.6916270214192226]
L: [2.618729044203357, 2.9561409959175196, 1.2282203494373447]
L: [5.771683570673411, 2.8438457746370362]
```

[8]:
```
[5.771683570673411, 2.8438457746370362]
```

[ ]:
```python

```

6.6 Backpropagation (Rétropropagation)

- NOM et prenom: Nom renom, nom prenom

L'algorithme de rétropropagation (Backpropagation) est une technique d'apprentissage supervisé utilisée pour entraîner les réseaux de neurones artificiels. Il repose sur la descente de gradient pour ajuster les poids du réseau et minimiser l'erreur entre la sortie prédite et la sortie réelle.

6.6.1 Principe de la Rétropropagation

L'algorithme suit trois étapes principales :

```
1. Propagation avant (Forward Propagation)
2. Calcul de l'erreur
3. Propagation arrière (Backward Propagation) et mise à jour des poids
```

6.6.2 Algorithme

1- Initialisation des poids par veleurs aléatoires 2- Pour chaque exemple x du dataset

1.a- Calculer les activations ($a_{ij}^{(l)}$ de tous le reseau 1.b- Calculer les $\delta_k^{(L)}$ des neurones de sortie

$$\delta_k^{(L)} := (a_k^{(L)} - y_k)\sigma'(s_k^{(L)})$$

1.c- Calculer les $\delta_k^{(l)}$ des neurone cachés

$$\delta_k^{(l)} = \sum_j w_{jk}^{(l+1)} \delta_j^{(l+1)} \sigma'(s_j^{(l)})$$

1.d- Calculer le gradient de la fonction coût par rapport aux $w_{ki}^{(l)}$ et aux $b_k^{(l)}$

$$\frac{\delta C_x}{\delta w_{ki}^{(l)}} = \delta_k^{(l)}.a_i^{(l-1)}$$

$$\frac{\delta C_x}{\delta b_k^{(l)}} = \delta_k^{(l)}.1$$

3- Calculer la moyenne des gradients sur la totalité des exemples d'entrainement

$$\frac{\delta C}{\delta w_{ki}^{(l)}} = \frac{1}{m} \sum_x \frac{\delta C_x}{\delta w_{ki}^{(l)}}$$

$$\frac{\delta C}{\delta b_k^{(l)}} = \frac{1}{m} \sum_x \frac{\delta C}{\delta b_k^{(l)}}$$

4- Mise à jour des poids

$$w_{ki}^{(l)} = w_{ki}^{(l)} - \eta \frac{\delta C}{\delta w_{ki}^{(l)}}$$

$$b_k^{(l)} = b_k^{(l)} - \eta \frac{\delta C}{\delta b_k^{(l)}}$$

5- répéter 2-4 jusqu'a l'obtention d'un coût acceptable ou un nombre d'iteration maximal

6.6.3 3. Application

Appliquer l'algoritm de la retropropagation d'erreur sur un réseau de la forme [4, 6, 6, 3]

1- Importion des modules

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
```

2- Utiliser **sklearn.datasets.load_iris** pour generer un jeux de donnée valide pour la structure du réseau ci-dessus

[10]:
```
d=2 #exemple size
H=[4,4,3]
```

```
X, y = make_blobs(n_samples=100, centers=H[-1], n_features=d,random_state=0)
Y=np.zeros((y.shape[0],H[-1]))
for i in range(y.shape[0]):
    Y[i,y[i]]=1
Y[:4]
```

[10]:
```
array([[0., 1., 0.],
       [1., 0., 0.],
       [0., 1., 0.],
       [1., 0., 0.]])
```

[ ]:
```

```

6.6.4 I. Implementation directe de l'algorithme

3- Créer une liste des poid adaptèe au réseau

[13]:
```
W=[np.random.rand(H[0],d),np.random.rand(H[1],H[0]),np.random.rand(H[2],H[1])]
W
```

[13]:
```
[array([[0.60169977, 0.16547036],
        [0.07256997, 0.20220087],
        [0.04061593, 0.71871332],
        [0.15232647, 0.7149863 ]]),
 array([[0.2648955 , 0.63972162, 0.03153269, 0.52198426],
        [0.2019343 , 0.50564292, 0.67334111, 0.06550604],
        [0.68865928, 0.39475166, 0.32880275, 0.97096667],
        [0.90022132, 0.29785344, 0.2470379 , 0.51679045]]),
 array([[0.9229129 , 0.85693302, 0.64787237, 0.36714316],
        [0.61790652, 0.36137875, 0.25835893, 0.93199829],
        [0.445199  , 0.1876389 , 0.12275359, 0.54158359]])]
```

4- Créer un fonction *forward()* qui Calculer les activation $a_k^{(l)}$

[23]:
```
def foward(X,W,H):
    a=[np.zeros((H[i],)) for i in range(len(H)) ]
    delta=[np.zeros((H[i],)) for i in range(len(H)) ]
    for i in range(len(H)):
        a[i]=X.dot(W[i])
```

5- Créer un fonction *backward()* qui Calculer les $\delta_k^{(l)}$

[22]:
```
a
```

[22]:
```
[array([0., 0., 0., 0.]), array([0., 0., 0., 0.]), array([0., 0., 0.])]
```

6- Créer la fonction *fit(X,Y)* pour trouver les valeur optimales des poids

[ ]:

7- Calculer la matrice de confusion du resultat obtenue

[ ]:

6.6.5 II- Utilisation de MLPClassifier

8- Utiliser **sklearn.neural_network.MLPClassifier** pour créer, entrainer et tester le reseau ci-dessus

[ ]:

9- Calculer la matrice de confusion du resultat obtenue

[ ]:

6.6.6 III- Utilisation Keras

ref: https://rtavenar.github.io/teaching/neuralnets_td/html/keras1.html

[ ]:

6.6.7 IV- Utilisation pytorche

ref: https://pageperso.lis-lab.fr/benoit.favre/pstaln/01_pytorch-mlp.html

[ ]: