Programmation et IA

Cours 7

Jean-Jacques Lévy

jean-jacques.levy@inria.fr

http://jeanjacqueslevy.net/prog-ia-25

Plan

- représentation des nombres réels
- norme flottant IEEE 754
- recherche de racines et d'extrema
- méthode par descente du gradient
- régression linéaire

[texte des programmes]

Pour apprendre Python:

w3schools: tutoriel et réference

`https://www.w3schools.com/python/default.asp`

programiz: tutoriel et réference

`https://www.programiz.com/python-programming`

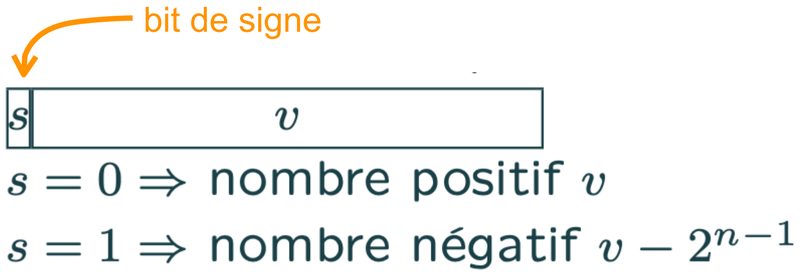
les nombres

Représentation des nombres entiers

• entiers: sont en binaire en complément à 2 sur *n* bits (*n* = 8, 16, 32, 64)

$$-2^{n-1} \leq x \leq 2^{n-1} - 1$$

10000000 01111111 en binaire



n = 8 $-128 \leq x \leq 127$

n = 16 $-32768 \leq x \leq 32767$

n = 32 $-2147483648 \leq x \leq 2147483647$

n = 64 $-9223372036854775808 \leq x \leq 9223372036854775807$ [9 milliards de milliards]

• entiers: grand nombres en précision arbitraire (GMP, Maple, Ocaml, **Python3**)

[les limites de leurs valeurs ne dépendent que de la taille de la mémoire de l'ordinateur]

Représentation des nombres réels

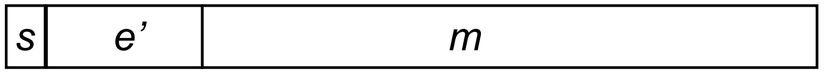
- les réels flottants sont des approximations des nombres réels

```
>>> 0.1 + 0.2 == 0.3
False
```

```
>>> 0.1 + 0.2
0.30000000000000004
```

- en virgule flottante : avec la norme *IEEE 754 standard for floating points* [W. Kahan 1985]

  bit de signe *s*, exposant *e'* sur 11 bits,
  mantisse *m* sur 52 bits



$s \in \{0, 1\} \quad e = e' - 1023$

$0 \leq e' < 2047, \quad -2^{52} \leq m \leq 2^{52} \implies x = \pm\, m\, 2^e$

$e' = 2047, \; m = 0. \implies x = \pm\infty$

$e' = 2047, \; m \neq 0. \implies x = \text{NaN}$

[texte des programmes]

Représentation des réels flottants

- les réels flottants en notation scientifique (en décimal)

```
>>> 1.234e3
1234
```

1.234×10^{3}

```
>>> 1.234e-13
.0000000000001234
```

1.234×10^{-13}

- les réels flottants sont représentés en binaire dans les processeurs

```
>>> 1.234e1
01100.01010111000010100011110101110000101000111 1
```

```
>>> 1.234e-3
0.0000000001010000110111110001010110100100101
```

```
>>> 0.5
000000000.1
```

```
>>> 0.75
00000000.11
```

[en puissances de 2 : $\cdots, 2^{3}, 2^{2}, 2^{1}, 2^{0} \cdot 2^{-1}, 2^{-2}, 2^{-3}, \cdots$]

[texte des programmes]

Représentation des réels flottants

- remarque: certains nombres décimaux finis ne peuvent être représentés en flottant

```
>>> 0.1
0.0001100110011001100110011001100110011001100110011
```

```
>>> -0.2
1.1100110011001100110011001100110011001100110011111
```

- un réel peut avoir 2 représentations différentes selon la place du point

$$0.000214 \times 10^2 = 0.214 \times 10^{-1}$$

- **représentation normalisée** (avec "01" ou "10" le plus à gauche dans la représentation binaire)

```
>>> 0.1
(0.1100110011001100110011001100110011001100110011000, -3)
```

```
>>> -0.2
(1.0011001100110011001100110011001100110011001100100, -2)
```

[le bit de signe différencie les 2 cas]

- cas spécial: zéro est représenté par tous les bits signe, exposant, mantisse valant zéro

[texte des programmes]

Erreur en calcul flottant

- erreur absolue d'arrondi par rapport au nombre réel exact (*p* chiffres significatifs après le point)

$$1/2 \times 2^{-p} \times 2^{e}$$

- erreur relative ou encore précision ϵ de la machine

$$\epsilon = 1/2 \times 2^{-p}$$

```
def precision () :
    a = 1.0
    while ((a + 1.0) - a) - 1.0 == 0.0 :
        a = 2 * a
    return a

epsilon = precision()
print (epsilon)
print (math.log2 (epsilon))
```

```
9007199254740992.0
53.0
```

- les opérations flottantes produisent des erreurs

- l'opération la plus redoutable est la soustraction

[texte des programmes]

Erreur en calcul flottant

- pour minimiser l'erreur, on change les formules par exemple quand $4ac \ll b^2$

$$\frac{-b+\sqrt{b^2-4ac}}{2a} \qquad \frac{2c}{-b-\sqrt{b^2-4ac}}$$

$$\frac{-b-\sqrt{b^2-4ac}}{2a} \qquad \frac{2c}{-b+\sqrt{b^2-4ac}}$$

- **Règle**: un test entre flottants n'a aucun sens si l'erreur de $|a-b|$ n'est pas estimée

```
if a > b :
    ...
else :
    ...
```

- série de livres Numerical recipes [1986 – 2007]

- bugs célèbres dus au calcul flottant: missile **Patriot** (1991), plate-forme **Sleipner A** (1991), **FDIV Intel** (1994), **Ariane 501** (1996), etc.

Exercice L'addition est-elle commutative ou associative en calcul flottant ?

[texte des programmes]

analyse numérique avec numpy

Module numpy

- analyse numérique en Python
- **vrais tableaux** multi-dimensionnels
- beaucoup de fonctions utiles en calculs numériques
- **efficace** et fonctionne sur les architectures **parallèles**
- *open source*
- tutoriel simple en `https://www.w3schools.com/python/numpy/default.asp`

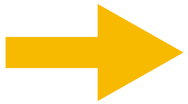
Module numpy

- analyse numérique en Python

```
import numpy as np
```

- vrais tableaux

```
a = [1, 2, 3, 4, 5]
b = np.array ([1, 2, 3, 4, 5])
print (a, b)
print (type(a), type(b))
```

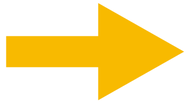


```
[1, 2, 3, 4, 5] [1 2 3 4 5]
<class 'list'> <class 'numpy.ndarray'>
```

N-dimensional array

- vrais tableaux multidimensionnels

```
c = np.array ([[1, 2, 3, 4], [5, 6, 7, 8]])
print (c)
print (c[1, 2])
print (type(c))
print (b.dtype, c.dtype)
print (b.ndim, c.ndim)
print (b.shape, c.shape)
```



```
[[1 2 3 4]
 [5 6 7 8]]
7
<class 'numpy.ndarray'>
int64 int64
1 2
(5,) (2, 4)
```

- les tableaux occupent une place contigüe en mémoire, et sont homogènes (tous leurs éléments ont un même type)

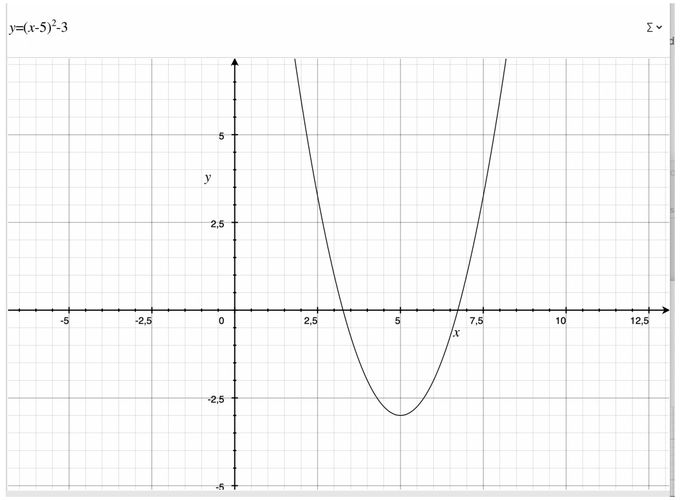
- attention aux ***alias*** ! on peut copier un tableau avec la méthode copy()

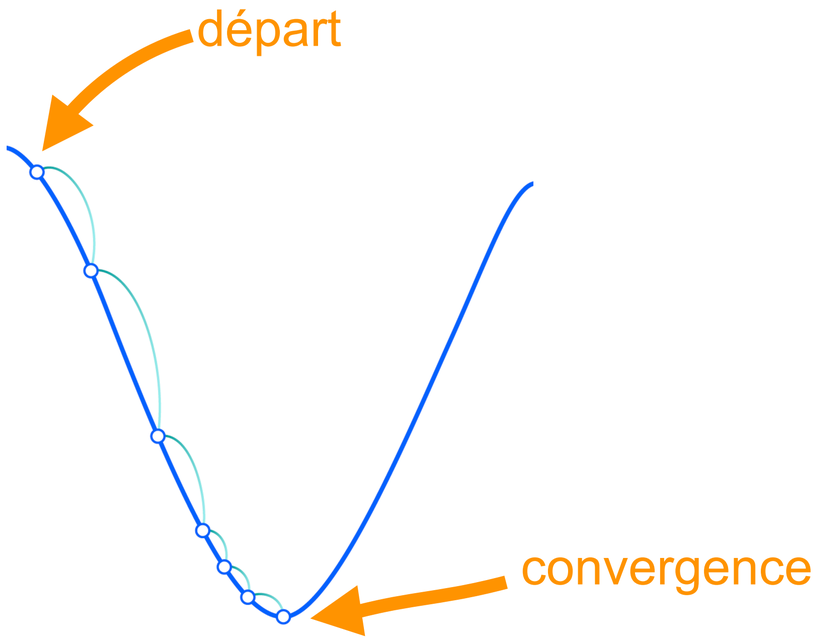
```
d = c
e = c.copy()
```

[texte des programmes]

Descente du gradient

• trouver le maximum ou minimum d'une fonction *f (x)*



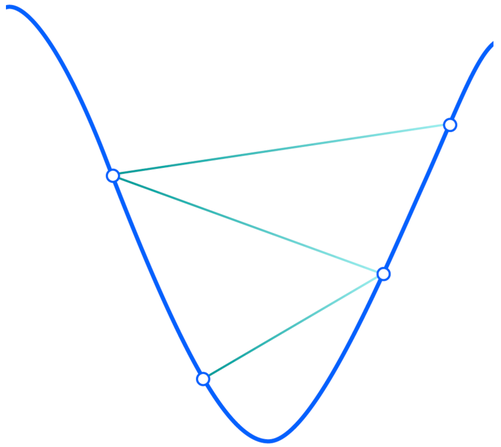


• on part d'un point sur la courbe `f(x)` et on descent vers le minimum en fonction de la pente de la courbe à ce point

• plus la pente est forte, plus on peut augmenter le **pas** de l'itération

• plus la pente est faible, plus on réduit le **pas** de l'itération

• si le pas de l'itération est trop grand, on peut osciller autour du minimum

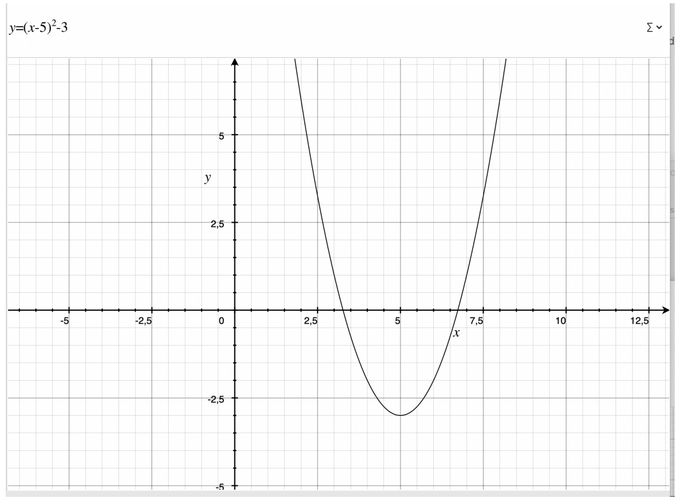


oscillations

[texte des programmes]

Descente du gradient

• trouver le maximum ou minimum d'une fonction *f (x)*




```
def f(x) :
    return (x-5)**2 - 3

def df (x) :
    return 2*(x-5)

alpha = 0.1

def min (f, x0) :
    x = x0
    for i in range (30) :
        x -= alpha * df(x)
    return x

print (min (f, 3))
```

• on part d'un point sur la courbe `f(x)` et on descent vers le minimum
en fonction de la pente de la courbe à ce point

• la pente se calcule avec la dérivée `df(x)` de la fonction en ce point

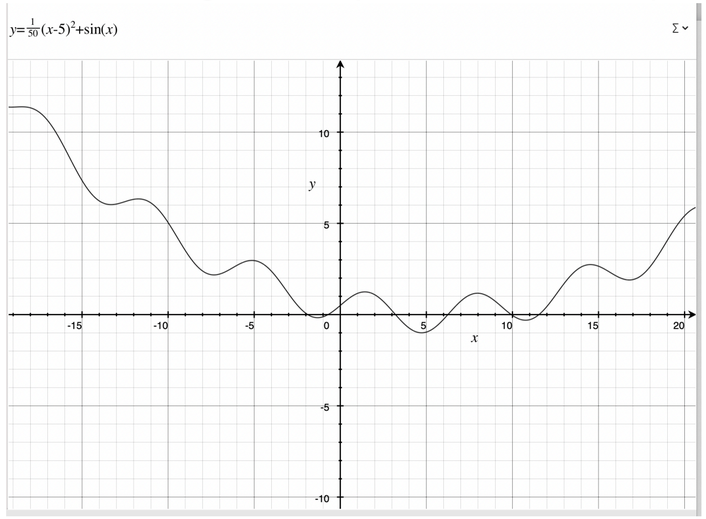
• on avance progressivement en choisissant un bon **taux** de progression `alpha`

• ce taux `alpha` ne doit être ni trop petit, ni trop grand

[texte des programmes]

Descente du gradient

- la descente du gradient peut ne trouver qu'un maximum ou minimum local




```
def f(x) :
    return 0.02 * (x-5)**2 + math.sin(x)

def df (x) :
    return 0.04*(x-5) + math.cos(x)

alpha = 0.1

def min (f, x0) :
    x = x0
    for i in range (30) :
        x -= alpha * df(x)
    return x

print (min (f, 10))
```

- ce minimum est global dans le cas d'une fonction parabolique

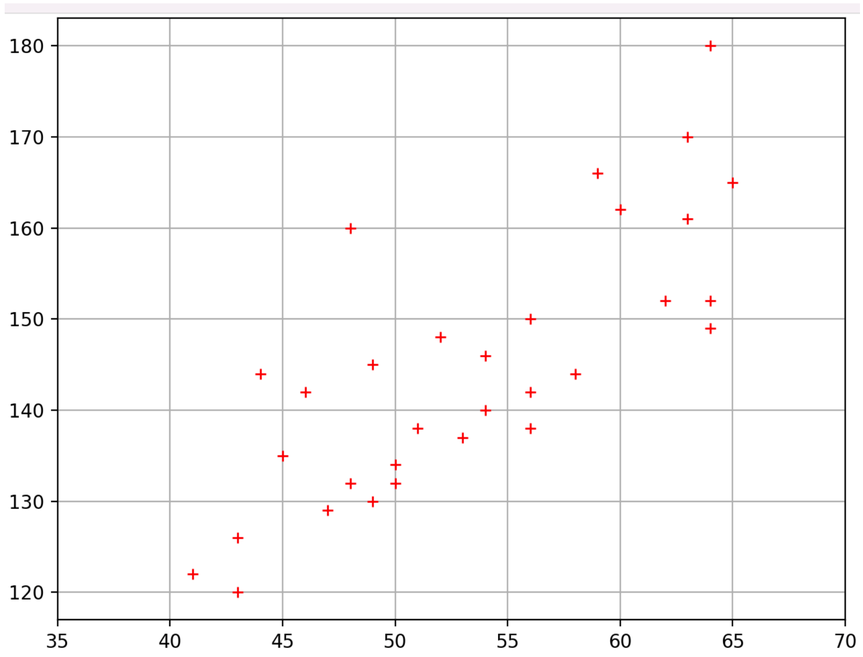
$$y = (x - a)^2$$

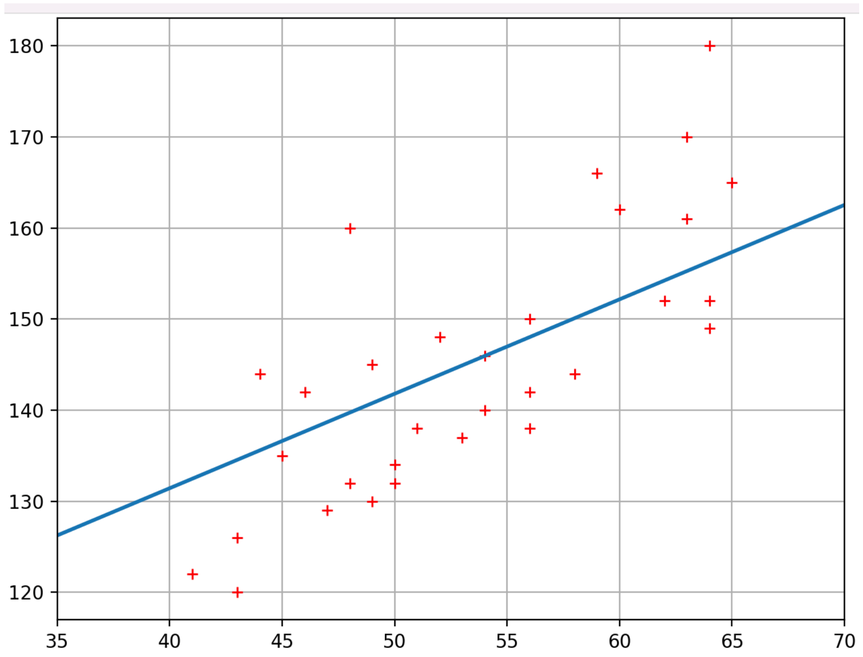
[texte des programmes]

régression linéaire

Régression linéaire

- on se donne un jeu de données (ici la pression artérielle en fonction de l'âge)
- on cherche une approximation linéaire de ces données

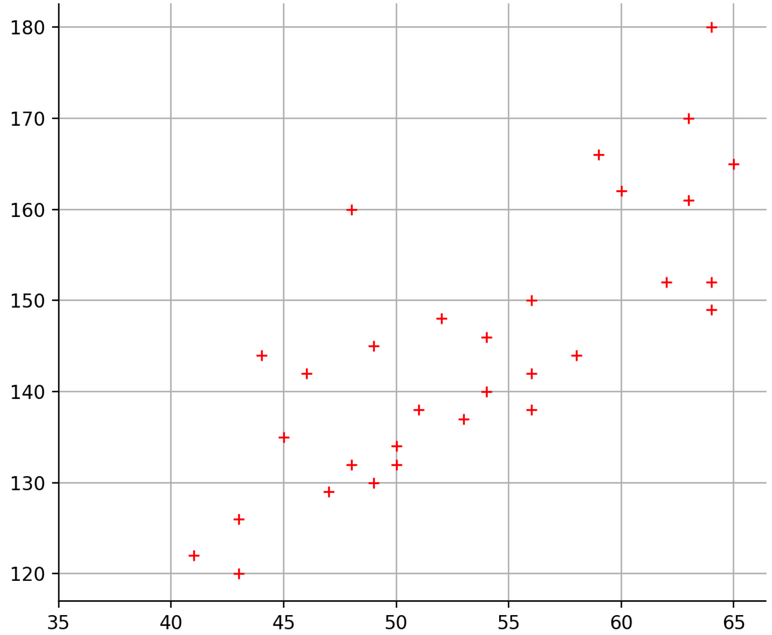




Régression linéaire

• le jeu de *n* données $(x^{(i)}, y^{(i)})$

```
dataset = [ (41, 122), (43, 120), (43, 126), (44, 144), (45, 135),
 (46, 142), (47, 129), (48, 132), (48, 160), (49, 130), (49, 145),
 (50, 132), (50, 134), (51, 138), (52, 148), (53, 137), (54, 140),
 (54, 146), (64, 149), (56, 138), (56, 142), (56, 150), (58, 144),
 (59, 166), (60, 162), (62, 152), (63, 161), (63, 170), (64, 152),
 (64, 180), (65, 165) ]
```



$x^{(0)} = 41,\ y^{(0)} = 122,\ x^{(1)} = 43,\ y^{(1)} = 120,\ x^{(2)} = 43,\ y^{(2)} = 126, \cdots$

[texte des programmes]

• on cherche une fonction $f(x) = \theta_0 + \theta_1 x$
telle que la valeur absolue de la différence $|f(x^{(i)}) - y^{(i)}|$
soit la plus petite possible pour tous les $(x^{(i)}, y^{(i)})$

```
def f (x, theta0, theta1) :
    return theta0 + theta1 * x
```

• la différence à minimiser est la somme :

$$J = \sum_{i=0,n-1} \frac{1}{2} (f(x^{(i)}) - y^{(i)})^2$$

[calcul des moindres carrés]

```
def J(dataset) :
    global theta0, theta1
    s = 0
    for (x, y) in dataset :
        s += 0.5 * (f (x, theta0, theta1) - y)**2
    return s
```

• *J* est aussi appelé le **coût** de l'approximation linéaire,
il dépend du jeu de données et des 2 **paramètres** θ_0 et θ_1

Régression linéaire

• on calcule les dérivées (partielles) du coût *J* par rapport à θ_0 et θ_1

$$\frac{\partial J}{\partial \theta_0} = \sum_{i=0,n-1} f(x^{(i)}) - y^{(i)}$$

$$\frac{\partial J}{\partial \theta_1} = \sum_{i=0,n-1} x^{(i)} \left(f(x^{(i)}) - y^{(i)}\right)$$

```
def derJ0 (dataset) :
    global theta0, theta1
    s = 0
    for (x, y) in dataset :
        s += f (x, theta0, theta1) - y
    return s
```

```
def derJ1 (dataset) :
    global theta0, theta1
    s = 0
    for (x, y) in dataset :
        s += x * (f(x, theta0, theta1) - y)
    return s
```

• on minimise le coût *J* par une descente du gradient en faisant varier les 2 paramètres θ_0 et θ_1

```
def descent (dataset) :
    global theta0, theta1, alpha
    for i in range (100) :
        theta0 -= alpha * derJ0 (dataset)
        theta1 -= alpha * derJ1 (dataset)
```

• on choisit un bon *alpha* et 2 valeurs initiales arbitraires pour θ_0 et θ_1

[texte des programmes]

Régression linéaire (résumé)

- on part du jeu de données

```
dataset = [ (41, 122), (43, 120), (43, 126), (44, 144), (45, 135),
 (46, 142), (47, 129), (48, 132), (48, 160), (49, 130), (49, 145),
 (50, 132), (50, 134), (51, 138), (52, 148), (53, 137), (54, 140),
 (54, 146), (64, 149), (56, 138), (56, 142), (56, 150), (58, 144),
 (59, 166), (60, 162), (62, 152), (63, 161), (63, 170), (64, 152),
 (64, 180), (65, 165) ]
```

- on cherche de bons θ_0 et θ_1 pour approximer ces données par une fonction linéaire

```
def f (x, theta0, theta1) :
    return theta0 + theta1 * x
```

- on calcule la fonction de coût *J* et ses dérivées par rapport à θ_0 et θ_1

```
def J(dataset) :
    global theta0, theta1
    s = 0
    for (x, y) in dataset :
        s += 0.5 * (f (x, theta0, theta1) - y)**2
    return s
```

```
def derJ0 (dataset) :
    global theta0, theta1
    s = 0
    for (x, y) in dataset :
        s += f (x, theta0, theta1) - y
    return s
```

```
def derJ1 (dataset) :
    global theta0, theta1
    s = 0
    for (x, y) in dataset :
        s += x * (f(x, theta0, theta1) - y)
    return s
```

- on fait une descente du gradient pour trouver les θ_0 et θ_1 qui minimisent le coût *J*

```
def descent (dataset) :
    global theta0, theta1, alpha
    for i in range (100) :
        theta0 -= alpha * derJ0 (dataset)
        theta1 -= alpha * derJ1 (dataset)
```

- on choisit un bon *alpha* et 2 valeurs initiales arbitraires pour θ_0 et θ_1

```
def run() :
    (theta0, theta1) = (90, 0); alpha = 0.00001
    print (theta0, theta1, J(dataset))
    descent (dataset)
    print (theta0, theta1, J(dataset))
```

[texte des programmes]

Régression linéaire (rappel)

• on calcule les dérivées (partielles) du coût *J* par rapport à θ_0 et θ_1

$$\frac{\partial J}{\partial \theta_0} = \sum_{i=0,n-1} f(x^{(i)}) - y^{(i)}$$

```
def derJ0 (dataset) :
    global theta0, theta1
    s = 0
    for (x, y) in dataset :
        s += f (x, theta0, theta1) - y
    return s
```

$$\frac{\partial J}{\partial \theta_1} = \sum_{i=0,n-1} x^{(i)} \left(f(x^{(i)}) - y^{(i)}\right)$$

```
def derJ1 (dataset) :
    global theta0, theta1
    s = 0
    for (x, y) in dataset :
        s += x * (f(x, theta0, theta1) - y)
    return s
```

• on minimise le coût *J* par une descente du gradient en faisant varier les 2 paramètres θ_0 et θ_1

```
def descent (dataset) :
    global theta0, theta1, alpha
    for i in range (100) :
        theta0 -= alpha * derJ0 (dataset)
        theta1 -= alpha * derJ1 (dataset)
```

• on choisit un bon *alpha* et 2 valeurs initiales arbitraires pour θ_0 et θ_1

[texte des programmes]

Régression linéaire (avec descente stochastique)

• on calcule les dérivées (partielles) du coût *J* par rapport à θ_0 et θ_1 pour chaque donnée

$$\frac{\partial J^{(i)}}{\partial \theta_0} = f(x^{(i)}) - y^{(i)}$$

$$\frac{\partial J^{(i)}}{\partial \theta_1} = x^{(i)} \left(f(x^{(i)}) - y^{(i)}\right)$$

```
def derJ0S (dataset, x, y) :
    global theta
    return (f (x, *theta) - y)
```

```
def derJ1S (dataset, x, y) :
    global theta
    return x * (f(x, *theta) - y)
```

• la descente stochastique ne fait la variation que pour chaque donnée en espérant obtenir un résultat proche sans osciller

• on itère la variation obtenue pour chaque donnée

```
def descentS (dataset) :
    global theta, alpha
    for i in range (100) :
        for (x, y) in dataset :
            theta[0] -= alpha * derJ0S (dataset, x, y)
            theta[1] -= alpha * derJ1S (dataset, x, y)
```

• la descente stochastique est donc plus rapide que la descente classique par lots

[texte des programmes]

Régression linéaire avec *m* variables

- le jeu de *n* données $(x^{(i)}, y^{(i)})$ où *x* est fabriqué de *m* composantes

- par exemple, la quantité de CO_2 en fonction du volume et du poids d'une voiture

  $x = (x_1, x_2)$ x_1 est le volume, x_2 est le poids

  y est la quantité de CO_2

- on cherche une fonction $f(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2$

Exercice 1 Trouver les paramètres $\theta_0, \theta_1, \theta_2$ par régression linéaire sur les données ci-contre.

En déduire la quantité de CO_2 pour une Renault Mégane: vol 1598, poids 1318

Car	Model	Volume	Weight	CO2
Toyota	Aygo	1000	790	99
Mitsubishi	Space-Star	1200	1160	95
Skoda	Citigo	1000	929	95
Fiat	500	900	865	90
Mini	Cooper	1500	1140	105
VW	Up!	1000	929	105
Skoda	Fabia	1400	1109	90
Mercedes	A-Class	1500	1365	92
Ford	Fiesta	1500	1112	98
Audi	A1	1600	1150	99
Hyundai	I20	1100	980	99
Suzuki	Swift	1300	990	101
Ford	Fiesta	1000	1112	99
Honda	Civic	1600	1252	94
Hundai	I30	1600	1326	97
Opel	Astra	1600	1330	97
BMW	1	1600	1365	99
Mazda	3	2200	1280	104
Skoda	Rapid	1600	1119	104
Ford	Focus	2000	1328	105
Ford	Mondeo	1600	1584	94
Opel	Insignia	2000	1428	99
Mercedes	C-Class	2100	1365	99
Skoda	Octavia	1600	1415	99
Volvo	S60	2000	1415	99
Mercedes	CLA	1500	1465	102
Audi	A4	2000	1490	104
Audi	A6	2000	1725	114
Volvo	V70	1600	1523	109
BMW	5	2000	1705	114
Mercedes	E-Class	2100	1605	115
Volvo	XC70	2000	1746	117
Ford	B-Max	1600	1235	104
BMW	2	1600	1390	108
Opel	Zafira	1600	1405	109
Mercedes	SLK	2500	1395	120

Régression linéaire avec *m* variables

```
dataset = [['Toyota', 'Aygo', 1000, 790, 99],
['Mitsubishi', 'Space-Star', 1200, 1160, 95],
['Skoda', 'Citigo', 1000, 929, 95],
['Fiat', '500', 900, 865, 90],
['Mini', 'Cooper', 1500, 1140, 105],
['VW', 'Up!', 1000, 929, 105],
['Skoda', 'Fabia', 1400, 1109, 90],
['Mercedes', 'A-Class', 1500, 1365, 92],
['Ford', 'Fiesta', 1500, 1112, 98],
['Audi', 'A1', 1600, 1150, 99],
['Hyundai', 'I20', 1100, 980, 99],
['Suzuki', 'Swift', 1300, 990, 101],
['Ford', 'Fiesta', 1000, 1112, 99],
['Honda', 'Civic', 1600, 1252, 94],
['Hundai', 'I30', 1600, 1326, 97],
['Opel', 'Astra', 1600, 1330, 97],
['BMW', '1', 1600, 1365, 99],
['Mazda', '3', 2200, 1280, 104],
['Skoda', 'Rapid', 1600, 1119, 104],
['Ford', 'Focus', 2000, 1328, 105],
['Ford', 'Mondeo', 1600, 1584, 94],
['Opel', 'Insignia', 2000, 1428, 99],
['Mercedes', 'C-Class', 2100, 1365, 99],
['Skoda', 'Octavia', 1600, 1415, 99],
['Volvo', 'S60', 2000, 1415, 99],
['Mercedes', 'CLA', 1500, 1465, 102],
['Audi', 'A4', 2000, 1490, 104],
['Audi', 'A6', 2000, 1725, 114],
['Volvo', 'V70', 1600, 1523, 109],
['BMW', '5', 2000, 1705, 114],
['Mercedes', 'E-Class', 2100, 1605, 115],
['Volvo', 'XC70', 2000, 1746, 117],
['Ford', 'B-Max', 1600, 1235, 104],
['BMW', '2', 1600, 1390, 108],
['Opel', 'Zafira', 1600, 1405, 109],
['Mercedes', 'SLK', 2500, 1395, 120]]
```

Car	Model	Volume	Weight	CO2
Toyota	Aygo	1000	790	99
Mitsubishi	Space-Star	1200	1160	95
Skoda	Citigo	1000	929	95
Fiat	500	900	865	90
Mini	Cooper	1500	1140	105
VW	Up!	1000	929	105
Skoda	Fabia	1400	1109	90
Mercedes	A-Class	1500	1365	92
Ford	Fiesta	1500	1112	98
Audi	A1	1600	1150	99
Hyundai	I20	1100	980	99
Suzuki	Swift	1300	990	101
Ford	Fiesta	1000	1112	99
Honda	Civic	1600	1252	94
Hundai	I30	1600	1326	97
Opel	Astra	1600	1330	97
BMW	1	1600	1365	99
Mazda	3	2200	1280	104
Skoda	Rapid	1600	1119	104
Ford	Focus	2000	1328	105
Ford	Mondeo	1600	1584	94
Opel	Insignia	2000	1428	99
Mercedes	C-Class	2100	1365	99
Skoda	Octavia	1600	1415	99
Volvo	S60	2000	1415	99
Mercedes	CLA	1500	1465	102
Audi	A4	2000	1490	104
Audi	A6	2000	1725	114
Volvo	V70	1600	1523	109
BMW	5	2000	1705	114
Mercedes	E-Class	2100	1605	115
Volvo	XC70	2000	1746	117
Ford	B-Max	1600	1235	104
BMW	2	1600	1390	108
Opel	Zafira	1600	1405	109
Mercedes	SLK	2500	1395	120

[texte des programmes]

Présentation algébrique

- on peut regrouper les paramètres, variables et données dans des matrices

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_d \end{bmatrix} \qquad x = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_d \end{bmatrix} \qquad X = \begin{bmatrix} {x^{(1)}}^T \\ \vdots \\ {x^{(n)}}^T \end{bmatrix} \qquad Y = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(n)} \end{bmatrix}$$

- calcul du coût

$$x^T = \begin{bmatrix} 1 & x_1 & \cdots & x_d \end{bmatrix}$$

$$f(x) = x^T\theta = \theta_0 + x_1\theta_1 + \cdots x_d\theta_d$$

$$f(X) - Y = \begin{bmatrix} f(x^{(1)}) - y^{(1)} \\ \vdots \\ f(x^{(n)}) - y^{(n)} \end{bmatrix} = X^T\theta - Y$$

$$J = \tfrac{1}{2}(f(X) - Y)^T(f(X) - Y) = \tfrac{1}{2}(X^T\theta - Y)^T(X^T\theta - Y)$$

- dérivée par rapport à θ

$$\nabla_\theta(J) = X^TX\theta - X^TY$$

- minimum **exact** quand la dérivée s'annule

$$\theta = (X^TX)^{-1}X^TY$$

Présentation algébrique

• minimum obtenu **exactement** en :

$$\theta = (X^T X)^{-1} X^T Y$$

Exercice 2 Trouver la solution exacte dans le cas du jeu de données avec une variable donnant la pression artérielle en fonction de l'âge. On a alors *d* = *2* et une seule variable *x*.

Le comparer au résultat obtenu par descente du gradient.

Rappel Dans le cas d'une matrice 2x2, on a:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad-bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

régression logistique

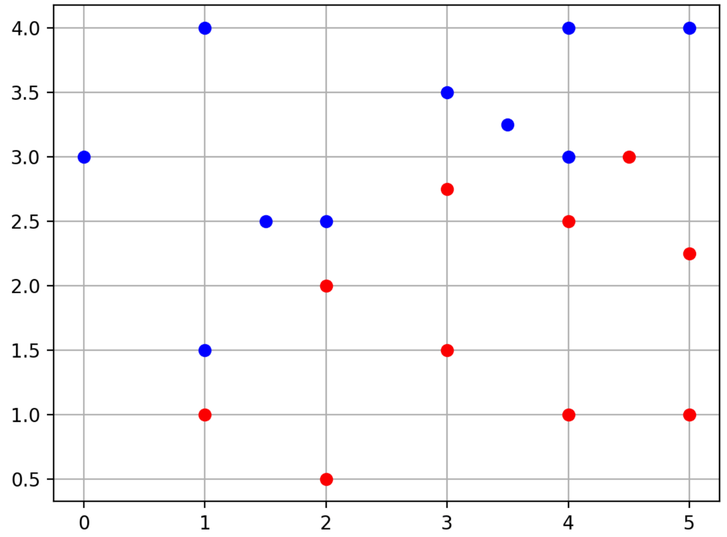
Classification

• le jeu de données est définit par deux ensembles de points rouges ou bleus dans le plan

```
bleus = [(0, 3), (1, 1.5), (1, 4), (1.5, 2.5), (2,2.5),
         (3, 3.5), (3.5, 3.25), (4,3), (4,4), (5,4)]

rouges = [(1,1), (2, 0.5), (2, 2), (3, 1.5), (3,2.75),
          (4,1), (4,2.5), (4.5, 3), (5, 1), (5, 2.25)]


dataset = [(0, 3, 0), (1, 1.5, 0), (1, 4, 0), (1.5, 2.5, 0),
           (2, 2.5, 0), (3, 3.5, 0), (3.5, 3.25, 0), (4, 3, 0),
           (4, 4, 0), (5, 4, 0), (1, 1, 1), (2, 0.5, 1),
           (2, 2, 1), (3, 1.5, 1), (3, 2.75, 1), (4, 1, 1),
           (4, 2.5, 1), (4.5, 3, 1), (5, 1, 1), (5, 2.25, 1)]
```



• pour chaque donnée $(x^{(i)}, y^{(i)}, z^{(i)})$ la valeur bleu correspond à 0 et la valeur rouge à 1 pour $z^{(i)}$

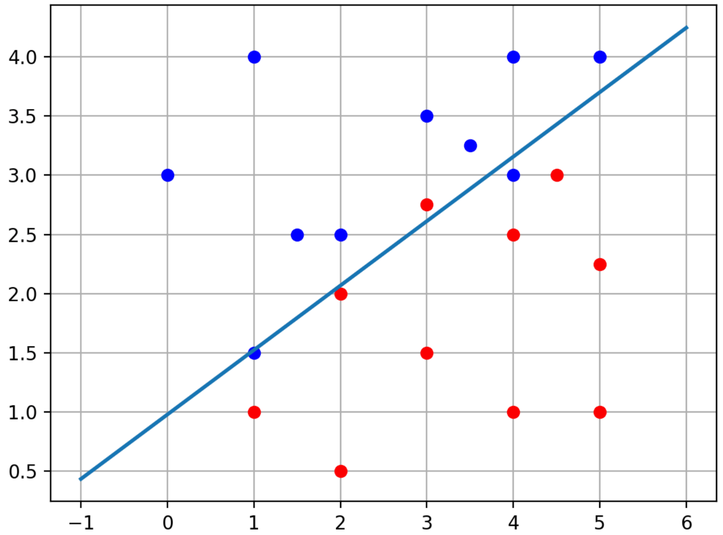
[texte des programmes]

Classification

- pour séparer par une **droite** les zones rouge et bleue, on cherche θ tel que :

(x, y) est rouge si $f(x, y) = \sigma(\theta_0 + \theta_1 x + \theta_2 y) > \frac{1}{2}$

(x, y) est bleu sinon



Remarque :

$$\sigma(\theta_0 + \theta_1 x + \theta_2 y) > \frac{1}{2} \quad \Longleftrightarrow \quad \theta_0 + \theta_1 x + \theta_2 y > 0$$

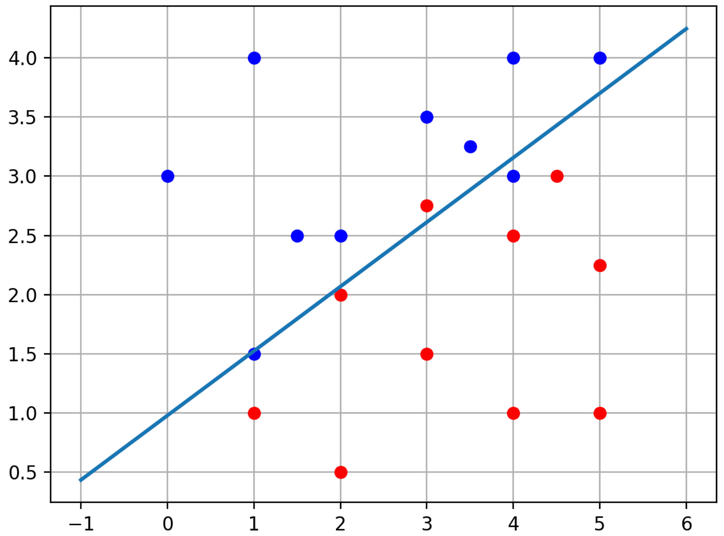
Classification

• l'erreur pour un θ donné est définie par la méthode des moindres carrés (*LMS - least mid squares*)

$$J = \sum_{i=0,n-1} \frac{1}{2} \left(f\left(x^{(i)}, y^{(i)}\right) - z^{(i)} \right)^2$$

• le minimum de l'erreur *J* est trouvé par descente du gradient
après 1000 itérations (stochastiques)

```
xlim = (-1, 6)
theta = [1.3, 0, 2]; alpha = 0.1
print (*theta, '--', J(dataset)); show(dataset)
descentS (dataset)
print (*theta, '--', J(dataset)); show(dataset)
```



Exercice 3 Programmer cette descente de gradient.

Exercice 4 Afficher en 3D les points *(x, y, z)* et le coût *J* en fonction de *x* et *y*.

[texte des programmes]

Prochain cours

- apprentissage
- réseau de neurones