DataScientest • com

NumPy for Data Science

Operations on Numpy arrays

1. Arithmetic operators

`Numpy` allows you to perform mathematical operations on arrays in an optimized way.

- Applying one of the basic arithmetic operations (/ , ∗ , − , + , ∗∗) between an array and a value will apply the operation to **each of the elements** of the array.
- It is also possible to perform an arithmetic operation **between two arrays**. This will apply the operation between **each pair of elements**.

```
# Creation of two arrays with 2 values
a = np.array([4, 10])
b = np.array([6, 7])

# Multiplication between two arrays
print(a * b)
>>> [24, 70]
```

- **(a)** Import the package **`numpy`** under the name **`np`** .
- **(b)** Create an array of dimensions 10x4 filled with ones.
- **(c)** Using a `for` loop and the `enumerate` function, multiply each row by its index. In order to modify the matrix, **it must be accessed through indexing**.

In [2]:

```python
# Insert your code here
import numpy as np

X = np.ones((10,4))

for n, arr in enumerate(X):
    X[n,:] = n*arr
X
```

Out[2]:

```
array([[0., 0., 0., 0.],
       [1., 1., 1., 1.],
       [2., 2., 2., 2.],
       [3., 3., 3., 3.],
       [4., 4., 4., 4.],
       [5., 5., 5., 5.],
       [6., 6., 6., 6.],
       [7., 7., 7., 7.],
       [8., 8., 8., 8.],
       [9., 9., 9., 9.]])
```

In [33]:

```python
# If the array of the value is bigger than 1 then compute power of values
X = np.ones((10,4))
X = X * [i for i in range(4)]
print(X[:2])

for i, row in enumerate(X):
    for j in row:
        if j > 1:
            X[i,int(j)] **= 2

X
```

```
[[0. 1. 2. 3.]
 [0. 1. 2. 3.]]
```

Out[33]:

```
array([[0., 1., 4., 9.],
       [0., 1., 4., 9.],
       [0., 1., 4., 9.],
       [0., 1., 4., 9.],
       [0., 1., 4., 9.],
       [0., 1., 4., 9.],
       [0., 1., 4., 9.],
       [0., 1., 4., 9.],
       [0., 1., 4., 9.],
       [0., 1., 4., 9.]])
```

Hide solution

In [3]:

```
import numpy as np

M = np.ones((10, 4))

# For each row of the matrix M
for i, row in enumerate(M):
    # We multiply the row by its index
    M[i,:] = row*i
    # Alternatively M[i,:]* = i

print(M)
```

```
[[0. 0. 0. 0.]
 [1. 1. 1. 1.]
 [2. 2. 2. 2.]
 [3. 3. 3. 3.]
 [4. 4. 4. 4.]
 [5. 5. 5. 5.]
 [6. 6. 6. 6.]
 [7. 7. 7. 7.]
 [8. 8. 8. 8.]
 [9. 9. 9. 9.]]
```

As explained above, the $*$ operator allows you to compute an element-wise product between arrays.

For example:

$$\begin{pmatrix} 5 & 1 \\ 3 & 0 \end{pmatrix} * \begin{pmatrix} 2 & 4 \\ 0 & 8 \end{pmatrix} = \begin{pmatrix} 10 & 4 \\ 0 & 0 \end{pmatrix}$$

The matrix product, in the mathematical sense of the term, can be performed using the **`dot`** method of a `numpy` array:

```
# Creation of two arrays of size 2x2
M = np.array([[5, 1],
              [3, 0]])
```

In [14]:

```
# Insert your code here


M = np.array([[5, 1],
              [3, 0]])


N = np.array([[2, 4],
              [0, 8]])



A = np.array([[1, 0],
              [0, 1]])


B = np.array([[1, -1],
              [-1, 1]])


def powerA(n, X, Y):

    for i in range(n):
        X = X.dot(Y)
        at = X @ Y
    return X, at


powerA(3, A, B)
```

Out[14]:

```
(array([[ 4, -4],
        [-4,  4]]), array([[ 8, -8],
        [-8,  8]]))
```

Hide solution

In [38]:

```
def powerA(n):
    # A is initialized to the identity matrix
    A = np.array([[1, 0],
                  [0, 1]])

    # This matrix B will be used to calculate the powers of A
    B = np.array([[1, -1],
                  [-1, 1]])

    # We multiply A by B n times to get A**n
    for i in range(n):
        A = A.dot(B)

    return A

print ("A**2: \n", powerA(2), "\n")
print ("A**3: \n", powerA(3), "\n")
print ("A**4: \n", powerA(4), "\n")

print ("A general formula of A**n is given by:")
print ("[2**(n-1), -2**(n-1)]")
print ("[-2**(n-1), 2**(n-1)]")
```

```
A**2:
 [[ 2 -2]
 [-2  2]]

A**3:
 [[ 4 -4]
 [-4  4]]

A**4:
 [[ 8 -8]
 [-8  8]]

A general formula of A**n is given by:
[2**(n-1), -2**(n-1)]
[-2**(n-1), 2**(n-1)]
```

In a two-dimensional plane, rotations around the origin are represented by the matrices of the form:

$$A(\theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}$$

where θ defines the angle of the rotation **in radians**. Thus, the rotation of a point with coordinates $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ is calculated thanks to the formula

$\tilde{x} = A(\theta)x$

- **(f)** Define a function named **`rotation_matrix`** taking as argument a number θ (`theta`) and returning the associated $A(\theta)$ matrix. You can calculate the $\cos$ and $\sin$ functions using the `np.cos` and `np.sin` functions of numpy.

In [42]:

```
# Insert your code here

def rotation_matrix(theta, X):
    Y = np.array([[np.cos(theta), -np.sin(theta)],
                  [np.sin(theta), np.cos(theta)]])
    return Y.dot(X)

#g)
x = np.array([1,1])

x_pi = rotation_matrix(np.pi, x)

print(x, x_pi )

#h)
a_pi_4 = rotation_matrix(np.pi/4, x)
a_3pi_4 = rotation_matrix(3*np.pi/4, x)

print(a_pi_4.dot(a_3pi_4), x_pi)
```

```
[1 1] [-1. -1.]
0.0 [-1. -1.]
```

In [70]:

```
# Insert your code here

def rotation_matrix(theta):
    Y = np.array([[np.cos(theta), -np.sin(theta)],
                  [np.sin(theta), np.cos(theta)]])
    return Y

#g)
x = np.array([1,1])

x_pi = rotation_matrix(np.pi)

print(x, x_pi.dot(x))

#h)
x = np.array([1,1])
a_pi_4 = rotation_matrix(np.pi/4)
a_3pi_4 = rotation_matrix(3*np.pi/4)

print(x, a_pi_4.dot(a_3pi_4.dot(x)), "\n")

#i)
pi = 100*np.pi
pi_2 = 200*np.pi

a_theta1 = rotation_matrix(pi)
a_theta2 = rotation_matrix(pi_2)
a_sum = rotation_matrix(pi + pi_2)

print("a_theta1 \n\n",a_theta1, "\n a_theta2 \n\n", a_theta2, "\n a_sum \n\n", a_sum, "\n")
print(a_theta1.dot(a_theta2.dot(x)))
print(a_sum.dot(x))
```

```
[1 1] [-1. -1.]
[1 1] [-1. -1.]

a_theta1

 [[ 1.00000000e+00 -1.96438672e-15]
 [ 1.96438672e-15  1.00000000e+00]]
 a_theta2

 [[ 1.00000000e+00 -3.92877345e-15]
 [ 3.92877345e-15  1.00000000e+00]]
 a_sum

 [[ 1.00000000e+00  5.09502587e-14]
 [-5.09502587e-14  1.00000000e+00]]

[1. 1.]
[1. 1.]
```

Hide solution

In [59]:

```python
# First question
def rotation_matrix(theta):
    A = np.array([[np.cos(theta), -np.sin(theta)],
                  [np.sin(theta), np.cos(theta)]])

    return A

# Second question
x = np.array([1, 1])
A_pi = rotation_matrix(np.pi)

print("x =", x)
print("A(pi)x =", A_pi.dot(x))

# Third question
A_pi_4 = rotation_matrix(np.pi/4)
A_3pi_4 = rotation_matrix(3*np.pi/4)

print("A(pi/4) A(3pi/4) x =", A_pi_4.dot(A_3pi_4.dot(x)))

# Fourth question
print("\n")
print("Intuitively, A(theta_1) A(theta_2) = A(theta_1 + theta_2) because applying a rotation of angle theta_1 then" )
print("applying another rotation of angle theta_2 is exactly the same as applying a rotation whose angle is the sum of the t
```

```
x = [1 1]
A(pi)x = [-1. -1.]
A(pi/4) A(3pi/4) x = [-1. -1.]


Intuitively, A(theta_1) A(theta_2) = A(theta_1 + theta_2) because applying a rotation of angle theta_1 then
applying another rotation of angle theta_2 is exactly the same as applying a rotation whose angle is the sum of the two.
```

2. Broadcasting between a matrix and a value

When performing an operation between elements of different dimensions, `Numpy` performs what is called **Broadcasting** to understand the operation and execute it.

The term broadcasting is used because one of the arrays is "broadcasted" into an array of larger dimensions so that the two arrays have compatible dimensions. This definition will be illustrated below.

In this section, we will try to understand numpy's broadcasting rules in the following cases:

- Operation between a matrix and a constant
- Operation between a matrix and a vector

An arithmetic operation such as the **sum between a matrix and a constant** does not make mathematical sense. With `Numpy`, the broadcasting rule in this case is to **sum the constant to each term** of the matrix.

$$M = \begin{pmatrix} 3 & 1 & 2 \\ -2 & 1 & 5 \end{pmatrix}, c = 10$$

$$\begin{aligned} M + c &= \begin{pmatrix} 3+10 & 1+10 & 2+10 \\ -2+10 & 1+10 & 5+10 \end{pmatrix} \\ &= \begin{pmatrix} 13 & 11 & 12 \\ 8 & 11 & 15 \end{pmatrix} \end{aligned}$$

What really happens is that the constant c is broadcasted into a matrix C with the same dimensions as M:

$$c \xrightarrow{\text{broadcasting}} C = \begin{pmatrix} c & c & c \\ c & c & c \end{pmatrix}$$

Thus, $M + C$ is mathematically well defined and can be computed with basic operations.

3. Broadcasting between a matrix and a vector

Similarly, `numpy` allows us to perform arithmetic operations between a matrix and a vector. However, there are some **constraints** which determine whether the vector can be *broadcasted* into a matrix with **compatible** dimensions.

In order to determine if the dimensions of the vector and the matrix are compatible, `numpy` will compare **each dimension** of the two arrays and determine if:

- the dimensions are equal.
- one of the dimensions is equal to 1.

If for each dimension, one of these conditions is verified, then the dimensions are **compatible** and the operation has been understood. Otherwise, a `ValueError: operands could not be broadcast together` error will be displayed.

Let us consider the following objects :

$$M = \begin{pmatrix} 3 & 1 & 2 \\ -2 & 1 & 5 \end{pmatrix}, v = \begin{pmatrix} 2 \\ 5 \end{pmatrix}$$

Do M and v have compatible dimensions for broadcasting?

M is a 2x3 dimensional matrix. v is a vector with 2 elements, but `numpy` will instead see v as a **matrix of dimensions 2x1**, that is, a matrix with two rows and one column.

The first dimension of M and v is equal to 2. They are **equal** so the compatibility condition is **verified** for this dimension.

The second dimension of M is equal to 3 and that of v is equal to 1. The compatibility condition is still **verified** because **one of the dimensions is equal to 1**.

Therefore M and v **have compatible dimensions** for broadcasting.

The vector v will then be broadcasted along the axis where the dimension of v is equal to 1. In our case, it is the axis of the **columns**. The broadcasting of v will therefore be given by:

$$v = \begin{pmatrix} 2 \\ 5 \end{pmatrix} \xrightarrow{\text{broadcasting}} V = \begin{bmatrix} v & v & v \end{bmatrix} = \begin{pmatrix} 2 & 2 & 2 \\ 5 & 5 & 5 \end{pmatrix}$$

The result of $M * v$ will then be given by:

$$\begin{aligned} M * v &\xrightarrow{\text{broadcasting}} M * V \\ &= \begin{pmatrix} 3 * 2 & 1 * 2 & 2 * 2 \\ -2 * 5 & 1 * 5 & 5 * 5 \end{pmatrix} \\ &= \begin{pmatrix} 6 & 2 & 4 \\ -10 & 5 & 25 \end{pmatrix} \end{aligned}$$

Now suppose we have a line vector $u = \begin{pmatrix} 3 & 4 \end{pmatrix}$.

For `numpy`, this vector has dimensions 1x2 (one row and 2 columns). The vectors u and v are compatible for broadcasting because on each axis one of the vectors has a dimension equal to 1.

How and **on which object** is the broadcasting carried out in this case?

The broadcasting will be carried out on the two vectors and the resulting matrix of the broadcasting will have the largest dimension between the two vectors:

$$v = \begin{pmatrix} 2 \\ 5 \end{pmatrix} \xrightarrow{\text{broadcasting}} V = \begin{pmatrix} 2 & 2 \\ 5 & 5 \end{pmatrix}$$

and

$$u = \begin{pmatrix} 3 & 4 \end{pmatrix} \xrightarrow{\text{broadcasting}} U = \begin{pmatrix} 3 & 4 \\ 3 & 4 \end{pmatrix}$$

Thus, the result of $v + u$ is given by:

$$\begin{aligned} v + u &= \begin{pmatrix} 2 \\ 5 \end{pmatrix} + \begin{pmatrix} 3 & 4 \end{pmatrix} \\ &\xrightarrow{\text{broadcasting}} V + U \\ &= \begin{pmatrix} 2 & 2 \\ 5 & 5 \end{pmatrix} + \begin{pmatrix} 3 & 4 \\ 3 & 4 \end{pmatrix} \\ &= \begin{pmatrix} 5 & 6 \\ 8 & 9 \end{pmatrix} \end{aligned}$$

These rules allow us to understand and predict the result of an operation between two arrays which do not have the same shape. They will be useful for the following exercise:

Min-Max normalization is a method that is used to **rescale the variables of a database to the interval** $[0, 1]$.

Assume our database contains 3 individuals and 2 variables:

In [129]:

```
l = [0,10, 15,4]
x = np.array([[1,2,3],
              [5,6,7]])

min(l)
max(l)
x.T

#l = [0,10, 15,25]
l2 = []

l = np.array([1,2,3,4])

for i in l:
    new_i = (i - min(l)) / (max(l) - min(l))
    l2.append(round(new_i, 3))
print(l, l.ndim)
print(l2)
```

```
[1 2 3 4] 1
[0.0, 0.333, 0.667, 1.0]
```

In [154]:

```
def normal(X):
    X_tilde = np.zeros(shape = X.shape)
    X_tilde = X_tilde.T
    X = X.T
    for j, row in enumerate(X):
        min_Xj = min(row)
        max_Xj = max(row)
        print(min_Xj, max_Xj)
        X_tilde[:, j] = (X[:, j] - min_Xj)/(max_Xj - min_Xj)

    return X_tilde.T

y = np.array([[24, 1.88],
              [18, 1.68],
              [14, 1.65]])
normal(y)
```

```
14.0 24.0
1.65 1.88
```

Out[154]:

```
array([[ 1.        , -1.212     ],
       [71.08695652,  0.13043478],
       [ 0.        ,  0.        ]])
```

In [141]:

```python
# Insert your code here

def normal(X):
    if X.ndim == 1:
        l2 = []
        for i in X:
            new_i = (i - min(X)) / (max(X) - min(X))
            l2.append(round(new_i, 2))
        return l2
    else:
        X = X.T
        for i, row in enumerate(X):
            mi = min(row)
            ma = max(row)
            for j, value in enumerate(row):
                new_value = (value - mi) / (ma - mi)
                X[i,j] = round(new_value, 2)

        return X.T

l = np.array([1,2,3,4])

X = np.array([[1, 0.4, 0],
              [1, 0.13, 0]])

y = np.array([[24, 1.88],
              [18, 1.68],
              [14, 1.65]])
normal(l)
normal(y)
#normal(X)
```

Out[141]:

```
array([[1.        , 1.        ],
       [0.4       , 0.13043478],
       [0.       , 0.        ]])
```

Hide solution

In [138]:

```
def normalization_min_max(X):
    # Initialization of X_tilde
    X_tilde = np.zeros(shape = X.shape)

    # For each column of X
    for j, column in enumerate(X.T):
        # Initialization of the minimum and maximum of the column
        min_Xj = column[0]
        max_Xj = column[0]

        # For each value in the column
        for value in column:
            # If the value is SMALLER than the min
            if value < min_Xj:
                # We overwrite the min with this value
                min_Xj = value

            # If the value is GREATER than the max
            if value > max_Xj:
                # We overwrite the max with this value
                max_Xj = value

        # We can now calculate X_tilde for this column
        # Broadcasting allows us to do this without a for loop
        X_tilde[:, j] = (X[:, j] - min_Xj)/(max_Xj - min_Xj)

    return X_tilde


X = np.array([[24, 1.88],
              [18, 1.68],
              [14, 1.65]])

X_tilde = normalization_min_max(X)

print(X_tilde)
```

```
[[1.         1.        ]
 [0.4        0.13043478]
 [0.         0.        ]]
```

4. Statistical methods

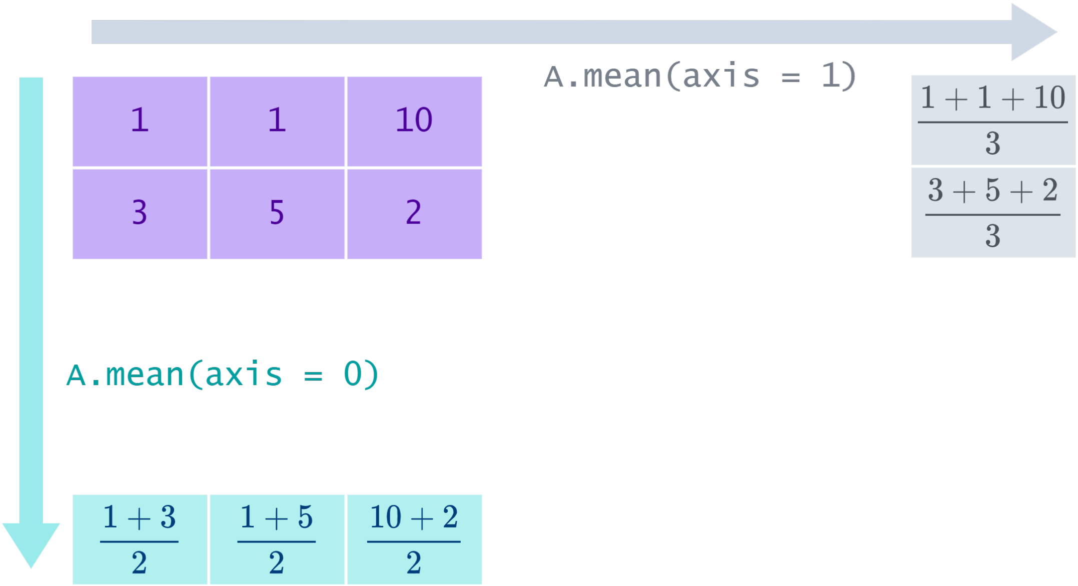
In addition to common math operations, `numpy` arrays also have several [methods (https://docs.scipy.org/doc/numpy-1.12.0/reference/arrays.ndarray.html#array-methods)](https://docs.scipy.org/doc/numpy-1.12.0/reference/arrays.ndarray.html#array-methods) for more complex operations on arrays.

One of the most used operations is the computation of an average using the **`mean`** method of an array:

```
A = np.array([[1, 1, 10],
              [3, 5, 2]])

# Computation of the mean over ALL the values of X
print(A.mean())
>>> 3.67

# Computation of the mean of each COLUMN of X
print(A.mean(axis = 0))
>>> [2, 3, 6]

# Computation of the mean of each ROW of X
print(A.mean(axis = 1))
>>> [4, 3.33]
```

The argument **`axis`** determines **which dimension will be scanned** to compute the mean:

- `axis = 0` means that the dimension scanned will be that of **rows**, which means that the result will be **the average of each column**.
- `axis = 1` means that the dimension scanned will be that of **columns**, which means that the result will be **the average of each row**.



The `axis` argument is **very often** used for operations on matrices, and **not only for `Numpy`**. It is very important to understand its effect.

There are other statistical methods that behave like the **`mean`** method, such as:

> - **`sum`** : Computes the sum of the elements of an array.
> - **`std`** : Computes the standard deviation.
> - **`min`** : Finds the minimum **value** among the elements of an array.
> - **`max`** : Finds the maximum **value** among the elements of an array.
> - **`argmin`** : Returns the index of the minimum **value**.
> - **`argmax`** : Returns the index of the maximum **value**.

These methods are useless for databases if you do not provide a value for the `axis` argument.

In general, we will use the value **`axis = 0`** to get the result **for each column**, that is, **for each variable in the database**.

Thus, we can calculate the Min-Max normalization very quickly using the **`min`** and **`max`** methods along with **broadcasting**:

```
X_tilde = (X - X.min(axis = 0))/(X.max(axis = 0) - X.min(axis = 0))

print (X_tilde)
>>> [[1, 1]
>>>  [0.4, 0.13043478]
>>>  [0, 0]]
```

The [Mean Squared Error (https://en.wikipedia.org/wiki/Mean_squared_error)](https://en.wikipedia.org/wiki/Mean_squared_error) is a metric to quantify the prediction error obtained by a regression model. This notion will be seen in more detail later in your training.

The formula for the mean squared error, abbreviated by MSE, is calculated with the following formula:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

where:

> - $\hat{y}$ and y are **vectors** with length n.
> - $\hat{y}$ is given by the matrix product between a X matrix and a *regression vector* β, ie:
> $$\hat{y} = X\beta$$

In [155]:

```
X = np.array([[24, 1.88],
              [18, 1.68],
              [14, 1.65]])

X_tilde = (X - X.min(axis = 0))/(X.max(axis = 0) - X.min(axis = 0))
X_tilde
```

Out[155]:

```
array([[1.        , 1.        ],
       [0.4       , 0.13043478],
       [0.        , 0.        ]])
```

In [156]:

```
# Insert your code here

def mean_squared_error(X, beta, y):
    y_hat = X.dot(beta)

    err_square = (y_hat - y)**2

    mse = mse.mean()

    return mse
```

Hide solution

In [157]:

```
def mean_squared_error(X, beta, y):
    # Computation of ^y
    y_hat = X.dot(beta)

    # Computation of (^y_i - y_i)**2
    mse = (y_hat - y)**2

    # MSE
    mse = mse.mean()

    return mse
```

Our database contained 3 individuals and 2 variables:

> - Jacques: 24 years old, height 1.88m.
> - Mathilde: 18 years old, height 1.68m.
> - Alban: 14 years old, height 1.65m.

We will try to find a model able to **predict the height of an individual based on his age**. Thus, we define:

$$X = \begin{pmatrix} 24 \\ 18 \\ 14 \end{pmatrix}$$

$$y = \begin{pmatrix} 1.88 \\ 1.68 \\ 1.65 \end{pmatrix}$$

Our goal will be to find an **optimal** β^* such that:

$$y \approx X\beta^*$$

- **(b)** For `beta` taking the values 0.01, 0.02, ..., 0.13, 0.14 and 0.15, compute the associated MSE using the previously defined `mean_squared_error` function. Store the values in a list.

To create the list `[0.01, 0.02, ..., 0.13, 0.14, 0.15]`, you can use the `np.linspace` function which has a signature similar to the `range` function:

In [210]:

```python
X = np.array([24,
              18,
              14])

y = np.array([1.88,
              1.68,
              1.65])

# Insert your code here


lst = np.linspace(0.01, 0.15, 15)
result = []
for beta in  lst:
    result.append([mean_squared_error(X, beta, y), beta])

#print(result)
print("Mininmun MSE and Optimal Beta :",min(result))
```

```
Mininmun MSE and Optimal Beta : [0.07803333333333334, 0.08999999999999998]
```

Hide solution

In [173]:

```python
X = np.array([24,
              18,
              14])

y = np.array([1.88,
              1.68,
              1.65])

# List containing the mse
errors = []

# List containing the betas to be tested
betas = np.linspace(start = 0.01, stop = 0.15, num = 15)

# For all the values of beta
for beta in betas:
    # Compute the associated MSE
    errors.append(mean_squared_error(X, beta, y))
```

- **(c)** Convert the list containing the MSE's to a numpy array.

In [213]:

```python
# Insert your code here

errs_arr = np.array(errors)
print("Array of MSE : \n", errs_arr, "\n")

min_err_index = errs_arr.argmin()
print("Min MSE index :", min_err_index, "\n")

print("Minimum MSE = ",errs_arr[min_err_index], "\n")
print("Optimal Beta = ", betas[min_err_index])
```

```
Array of MSE :
 [2.40656667 1.85976667 1.38603333 0.98536667 0.65776667 0.40323333
 0.22176667 0.11336667 0.07803333 0.11576667 0.22656667 0.41043333
 0.66736667 0.99736667 1.40043333]

Min MSE index : 8

Minimum MSE =  0.07803333333333334

Optimal Beta =  0.08999999999999998
```

Hide solution

In [201]:

```python
# Array containing the MSE for each beta
errors = np.array(errors)

# List containing the betas that have been tested
betas = np.linspace(start = 0.01, stop = 0.15, num = 15)

# Index of the beta that minimizes the MSE
index_beta_optimal = errors.argmin()

# Optimal beta
beta_optimal = betas[index_beta_optimal]

print("The optimal beta is:", beta_optimal)
```

```
The optimal beta is: 0.08999999999999998
```

- **(e)** What are the heights predicted by this optimal β^*? The heights predicted by the model are given by the vector $\hat{y} = X\beta^*$.
- **(f)** Compare the predicted heights to the actual heights of the individuals. For example, you can compute the average absolute difference between the predicted and true values using the absolute value (`np.abs`).

In [222]:

```
# Insert your code here

y_predict = X.dot(beta_optimal)
print("Predicted heights : ",y_predict)


print("Real heights      : ", y)

errors_ = np.abs(y - y_predict)

print("Erros             : ", errors_)

print("Average of errors : ", errors_.mean())
```

```
Predicted heights :  [2.16 1.62 1.26]
Real heights      :  [1.88 1.68 1.65]
Erros             :  [0.28 0.06 0.39]
Average of errors :  0.24333333333333334
```

Hide solution

In [214]:

```
y_hat = X.dot(beta_optimal)
print("Predicted heights: \n", y_hat)

print("\n Real heights: \n", y)

print("\n The model makes an average mistake of", np.abs(y - y_hat).mean(), "metres.")

# The predicted heights are incorrect but approximately very close to the real sizes.
```

```
Predicted heights:
 [2.16 1.62 1.26]

 Real heights:
 [1.88 1.68 1.65]

 The model makes an average mistake of 0.24333333333333334 metres.
```

In [ ]: