Lab 2: Numpy

The objective of this notebook is to learn about the Numpy library (official documentation). You can find a good guide at this link.

Exploit the Numpy library and avoid explicit for loops and python lists for all the exercises of this lab.

Outline

- 1. Numpy arrays creation
- 2. Operations with Numpy arrays
- 3. Accessing Numpy arrays
- 4. Min-Max normalization with numpy

First, run the following cell to import some useful libraries to complete this Lab. If not already done, you must install them in your virtual environment

In [2]:

```
import numpy as np
import matplotlib.pyplot as plt
import random
```

If the previous cell outputs one of the following errors: `ModuleNotFoundError: No module named 'numpy'` or `ModuleNotFoundError: No module named 'matplotlib'`, then, you have to install the numpy or the matplotlib packages. If you don't remember how to install a Python package, please retrieve the guide on Anaconda-Navigator.

To install **numpy** you can use one of the following commands from the terminal of your virtual environment:
`conda install numpy`
`pip install numpy`

To install **matplotlib** you can use one of the following commands from the terminal of your virtual environment:
`conda install matplotlib`
`pip install matplotlib`

Please run the following cell containing useful functions already implemented for you to plot some charts.

In [3]:

```
def plot_distributions(my_list, names):
    fig, ax = plt.subplots(1, len(my_list), figsize=(14, 6))

    fig.suptitle("Frequency Histograms X, Y", fontsize=20)

    for i, x in enumerate(my_list):
        ax[i].hist(x, 25)
        ax[i].axvline(x.mean(), color='k', linestyle='dashed', linewidth=2)
        ax[i].set_xlabel(names[i], fontsize=14)
        ax[i].set_ylabel('Frequency', fontsize=14)

    plt.tight_layout()
    plt.show()
    return


def plot_2d_points(X,Y, norm_flag=False):
    fig, ax = plt.subplots(figsize=(10, 5))

    if norm_flag:
        ax.set_xlabel('Size of the house norm', fontsize=14)
        ax.set_ylabel('Price', fontsize=14)
    else:
        ax.set_xlabel('Size of the house', fontsize=14)
        ax.set_ylabel('Price', fontsize=14)
    ax.scatter(X, Y)

    plt.show()
    return

def plot_3d_points(X, Y, norm_flag=False):
    fig, ax = plt.subplots(figsize=(20, 10))
    ax = fig.add_subplot(projection='3d')


    if norm_flag:
        ax.set_xlabel('Size of the house norm', fontsize=14)
        ax.set_ylabel('Number of rooms norm', fontsize=14)
        ax.set_zlabel('Price', fontsize=14)
    else:
        ax.set_xlabel('Size of the house', fontsize=14)
        ax.set_ylabel('Number of rooms', fontsize=14)
        ax.set_zlabel('Price', fontsize=14)

    ax.scatter(X[:,0], X[:,1], Y)

    plt.tight_layout()
    plt.show()
    return
```

1. Numpy arrays creation

Exercise 1.1

Create a Numpy array from the following list: `[[1, 2, 3], [4, 5, 6], [7, 8, 9]]`. Print the the **values** of the **array** and the **shape**.

In [4]:

```
#### START CODE HERE ####
#### Ideally 3 line ####
my_arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(my_arr)
print(my_arr.shape)
#### END CODE HERE ####
```

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

Expected output

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

Exercise 1.2

Create a Numpy array `my_arr` filled with all **ones** with **3 rows** and **4 columns**.

▶ **Hints**

In [5]:

```
#### START CODE HERE ####
#### Ideally 1 line ####
my_arr = np.ones((3, 4))
#### END CODE HERE ####

print(my_arr)
print(my_arr.shape)
```

```
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
(3, 4)
```

Expected output

```
[[1. 1. 1. 1.]
[1. 1. 1. 1.]
[1. 1. 1. 1.]]
(3, 4)
```

Exercise 1.3

Create a Numpy array `my_arr` filled with all **zeros** with **5 rows** and **2 columns**.

▶ **Hints**

In [6]:

```
#### START CODE HERE ####
#### Ideally 1 line ####
my_arr = np.zeros((5, 2))
#### END CODE HERE ####

print(my_arr)
print(my_arr.shape)
```

```
[[0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]]
(5, 2)
```

Expected output

```
[[0. 0.]
[0. 0.]
[0. 0.]
[0. 0.]
[0. 0.]]
(5, 2)
```

Exercise 1.4

Create a Numpy array `my_arr` filled with all **0.5** with **2 rows** and **2 columns**.

▶ **Hints**

In [7]:

```
#### START CODE HERE ####
#### Ideally 1 line ####
my_arr = np.full((2, 2), 0.5)
#### END CODE HERE ####

print(my_arr)
print(my_arr.shape)
```

```
[[0.5 0.5]
 [0.5 0.5]]
(2, 2)
```

Expected output

```
[[0.5 0.5]
[0.5 0.5]]
(2, 2)
```

2. Operations with Numpy arrays

Exercise 2.1

Add the value 5 to **each element** of the Numpy array `my_arr`.

In [8]:

```
my_arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
#### START CODE HERE ####
#### Ideally 1 line ####
my_arr = my_arr + 5
#### END CODE HERE ####
print(my_arr)
```

```
[[ 6  7  8]
 [ 9 10 11]
 [12 13 14]]
```

Expected output

`[[ 6  7  8]`

`[ 9 10 11]`

`[12 13 14]]`

Exercise 2.2

Perform the element-wise **logarithm** of the Numpy array `my_arr`.

In [10]:

```python
my_arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
#### START CODE HERE ####
#### Ideally 1 line ####
my_arr = np.log(my_arr)
#### END CODE HERE ####
print(my_arr)
```

```
[[0.         0.69314718 1.09861229]
 [1.38629436 1.60943791 1.79175947]
 [1.94591015 2.07944154 2.19722458]]
```

Expected output

`[[0.         0.69314718 1.09861229]`

`[1.38629436 1.60943791 1.79175947]`

`[1.94591015 2.07944154 2.19722458]]`

Exercise 2.3

Compute the **mean** in `arr_mean`, the **standard deviation** in `arr_std`, the **sum** in `arr_sum`, the **max** value in `arr_max`, and the **index of the max** value in `arr_max_idx` of the array `my_arr`.

In [9]:

```python
my_arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
#### START CODE HERE ####
#### Ideally 5 line ####
arr_mean = np.mean(my_arr)
arr_std = np.std(my_arr)
arr_sum = np.sum(my_arr)
arr_max = np.max(my_arr)
arr_max_idx = np.argmax(my_arr)
#### END CODE HERE ####

print(f"mean: {arr_mean}")
print(f"standard deviation: {arr_std}")
print(f"sum: {arr_sum}")
print(f"max value: {arr_max}")
print(f"index of the max value: {arr_max_idx}")
```

```
mean: 5.0
standard deviation: 2.581988897471611
sum: 45
max value: 9
index of the max value: 8
```

Expected output

```
mean: 5.0
standard deviation: 2.581988897471611
sum: 45
max value: 9
index of the max value: 8
```

Exercise 2.4

Compute the **mean** along the **rows axis** in `arr_mean_rows` of the array `my_arr`.

▶ **Hints**

In [11]:

```
my_arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(my_arr)

#### START CODE HERE ####
#### Ideally 1 line ####
arr_mean_rows = np.mean(my_arr, axis=1)
#### END CODE HERE ####

print(f"\nmean of the rows: {arr_mean_rows}")
```

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

mean of the rows: [2. 5. 8.]
```

Expected output

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

mean of the rows: [2. 5. 8.]
```

Exercise 2.5

Compute the **mean** along the **columns axis** in `arr_mean_cols` of the array `my_arr`.

▶ **Hints**

In [12]:

```
my_arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(my_arr)
#### START CODE HERE ####
#### Ideally 1 line ####
arr_mean_cols = np.mean(my_arr, axis=0)
```

```
#### END CODE HERE ####
print(f"\nmean of the columns: {arr_mean_cols}")
```

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

mean of the columns: [4. 5. 6.]
```

Expected output

`[[1 2 3]`

`[4 5 6]`

`[7 8 9]]`

`mean of the columns: [4. 5. 6.]`

You can see that both the mean along the rows axis and the mean along the columns axis return a **row vector**.

3. Accessing Numpy arrays

Exercise 3.1

Assign the value of the element in the **fourth row** and the **second column** of the array `my_arr` into a variable `value`.

▶ **Hints**

In [13]:

```
my_arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10], [11, 12, 13, 14, 15],
print(my_arr)

#### START CODE HERE ####
#### Ideally 1 line ####
value = my_arr[3, 1]
#### END CODE HERE ####

print(f"\nThe element in the fourth row and second column is {value}")
```

```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]
 [16 17 18 19 20]]

The element in the fourth row and second column is 17
```

Expected output

`[[ 1  2  3  4  5]`

`[ 6  7  8  9 10]`

`[11 12 13 14 15]`

`[16 17 18 19 20]]`

`The element in the fourth row and second column is 17`

Exercise 3.2

Assign the values of the slice corresponding to the **rows from 0 to 2 (both included)** and the **columns from 1 to 2 (both included)** into a variable `slice_arr`.

▶ **Hints**

In [14]:

```
my_arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10], [11, 12, 13, 14, 15],
print(my_arr)

#### START CODE HERE ####
#### Ideally 1 line ####
slice_arr = my_arr[:3, 1:3]
#### END CODE HERE ####

print(f"\nSlice:")
print(slice_arr)
```

```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]
 [16 17 18 19 20]]

Slice:
[[ 2  3]
 [ 7  8]
 [12 13]]
```

Expected output

```
[[ 1  2  3  4  5]
[ 6  7  8  9 10]
[11 12 13 14 15]
[16 17 18 19 20]]

Slice:
[[ 2  3]
[ 7  8]
[12 13]]
```

Exercise 3.3

Assign the values of the slice with **all the columns** of the **last 3 rows** into a variable `slice_arr`.

▶ **Hints**

In [15]:

```
my_arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10], [11, 12, 13, 14, 15],
print(my_arr)

#### START CODE HERE ####
#### Ideally 1 line ####
slice_arr = my_arr[-3:, :]
#### END CODE HERE ####
```

```
print(f"\nSlice:")
print(slice_arr)
```

```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]
 [16 17 18 19 20]]

Slice:
[[ 6  7  8  9 10]
 [11 12 13 14 15]
 [16 17 18 19 20]]
```

Expected output

```
[[ 1  2  3  4  5]
[ 6  7  8  9 10]
[11 12 13 14 15]
[16 17 18 19 20]]

Slice:
[[ 6  7  8  9 10]
[11 12 13 14 15]
[16 17 18 19 20]]
```

Exercise 3.4

Assign the values of the slice with **all the columns of the last 3 rows** into a variable `slice_arr`. Then assign to **all the elements** of `slice_arr` the value `-1`.

▶ **Hints**

In [16]:

```
my_arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10], [11, 12, 13, 14, 15],
print("original array:")
print(my_arr)

#### START CODE HERE ####
#### Ideally 2 line ####
slice_arr = my_arr[-3:, :]
slice_arr[:, :] = -1
#### END CODE HERE ####

print(f"\n array after the modification of the slice:")
print(my_arr)
```

```
original array:
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]
 [16 17 18 19 20]]

 array after the modification of the slice:
[[ 1  2  3  4  5]
 [-1 -1 -1 -1 -1]
 [-1 -1 -1 -1 -1]
 [-1 -1 -1 -1 -1]]
```

Expected output

```
original array:
[[ 1  2  3  4  5]
[ 6  7  8  9 10]
[11 12 13 14 15]
[16 17 18 19 20]]

Slice:
[[ 1  2  3  4  5]
[-1 -1 -1 -1 -1]
[-1 -1 -1 -1 -1]
[-1 -1 -1 -1 -1]]
```

You can see that the modifications on the slice also affect the original array.

Exercise 3.5

Assign the values of the slice with **all the columns of the last 3 rows** into a variable `slice_arr`. This time, `slice_arr` should **not be a view but a new array** (i.e., the modification of `slice_arr` should **not affect** the original array). Then **assign to all the elements** of `slice_arr` the value `-1`.

▶ **Hints**

In [18]:

```
my_arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10], [11, 12, 13, 14, 15],
print("original array:")
print(my_arr)

#### START CODE HERE ####
#### Ideally 2 line ####
slice_arr = my_arr[-3:, :] .copy()
slice_arr[:, :] = -1
#### END CODE HERE ####

print(f"\n slice:")
print(slice_arr)
print(f"\n array after the modification of the slice:")
print(my_arr)
```

```
original array:
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]
 [16 17 18 19 20]]

 slice:
[[-1 -1 -1 -1 -1]
 [-1 -1 -1 -1 -1]
 [-1 -1 -1 -1 -1]]

 array after the modification of the slice:
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]
 [16 17 18 19 20]]
```

Expected output

```
original array:
[[ 1  2  3  4  5]
[ 6  7  8  9 10]
[11 12 13 14 15]
[16 17 18 19 20]]

slice
[[-1 -1 -1 -1 -1]
[-1 -1 -1 -1 -1]
[-1 -1 -1 -1 -1]]

array after the modification of the slice:
[[ 1  2  3  4  5]
[ 6  7  8  9 10]
[11 12 13 14 15]
[16 17 18 19 20]]
```

This time, the modifications of the slice do **not affect** anymore the original array.

Exercise 3.6

Define a **mask** of the array into a variable `mask` with **all the elements greater or equal than 5 and less or equal than 10**. Then, assign to **all the masked elements** of the original array the value `-1`.

▶ **Hints**

In [19]:

```
my_arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10], [11, 12, 13, 14, 15],
print("original array:")
print(my_arr)

#### START CODE HERE ####
#### Ideally 2 line ####
mask = (my_arr >= 5) & (my_arr <= 10)
my_arr[mask] = -1
#### END CODE HERE ####

print(f"\n array after the modification of the masked elements:")
print(my_arr)
```

```
original array:
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]
 [16 17 18 19 20]]

 array after the modification of the masked elements:
[[ 1  2  3  4 -1]
 [-1 -1 -1 -1 -1]
 [11 12 13 14 15]
 [16 17 18 19 20]]
```

Expected output

```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]
 [16 17 18 19 20]]

array after the modification of the masked elements:
[[ 1  2  3  4 -1]
 [-1 -1 -1 -1 -1]
 [11 12 13 14 15]
 [16 17 18 19 20]]
```

4. Min-Max normalization with numpy

In this exercise, you have to implement the **Min-Max normalization**. It is a simple method that rescales the range of features into `[0, 1]`. The formula for the **Min-Max normalization** is the following:

$$x\_norm = \frac{(x - x_{min})}{(x_{max} - x_{min})}$$

Some learning algorithms require **input features** to be **normalized** or **standardized** to work correctly. In the next labs, we will learn some libraries (e.g., *scikit-learn*) that perform **normalization** or **standardization** with one single line of code. These libraries exploit *Numpy* internally. However, it is important to understand how to work with arrays.

4.1 Min-Max normalization of one-dimensional data

We first focus on a **synthetic dataset** (i.e., artificially generated) with **one input feature** (i.e., `X` has shape `(n_samples, 1)`) and a **continuous target variable** `Y` (with shape `(n_samples, 1)`).

We will use, as **one-dimensional** example, a dataset with `n_samples` (records) composed of couples containing the `size of the house` and the relative `price` (in thousands of €). We would like to train a Machine Learning model (we will learn how to train a model in the next labs) that takes as input the `size of the house` and predicts the most probable `price`. This paradigm is called **supervised learning** because you provide to the model both the **input features** `X` and the **expected output** `Y`. During learning, the model will try to predict, for each input $x_i \in X$ the corresponding output y_i (e.g., for each house, it will try to predict the price given the size). Then, the predicted output $\hat{y}_i$ is compared with the real output y_i by computing a measure that quantifies the error of the model (e.g., a distance between the predicted and real values). Then, the model will update the internal weights based on the error. For example, if the error is very low, it means that the model is good at predicting the price for that house. Therefore, it should **not** update its internal weights. If the error is high, the model is **not** good at predicting the price of that house. Therefore, it should update its internal weight. This procedure is repeated for all the samples in your dataset.

This is just a simplification to understand how Machine Learning algorithms work. However, we will learn more in the next labs. So, if it is not so clear to you, don't worry. The purpose is to begin to understand why some exercises.

In this example, `X` is of shape `(n_samples, 1)`. It is one-dimensional because, for each house, the model takes as input only the size of the house. For notation, we write the input `X` as a column vector (i.e., `(n_samples, 1)` is a column vector with `n_samples` rows).

Sythetic dataset generation: one-dimensional input features

Firstly, you will create a vector `X` containing the size of the houses with `500` samples generated by a **normal distribution** given a **mean** `mu` and a **standard deviation** `sigma`. This means that the sizes will have **mean** = `mu` and **standard deviation** = `sigma`. The code for generating the samples is already available to you. Please run the next cell to generate the samples.

In [20]:

```
mu, sigma = 120, 50 # mean and standard deviation
X = np.random.normal(mu, sigma, (500,)) # generate a gaussian distribution w
X = np.abs(X) # only positive values make sense for the size of the house

print("X Shape", X.shape)
print("X Minimum value:", X.min())
print("X Maximum value:", X.max())
print("X mean:", X.mean())
print("X standard deviation:", X.std())
print("First ten element of x: ", X[:5])
```

```
X Shape (500,)
X Minimum value: 0.5025244923023138
X Maximum value: 285.6666080041605
X mean: 120.8444591505394
X standard deviation: 49.79067742074923
First ten element of x:  [ 92.3703539    6.54663671 191.74560346 189.1849622
7 160.96362765]
```

Now compute the price of each house as a linear function of the input features `X` with an error term. The price of the house is computed as follows:

$$price = size * (price\_per\_mq \pm error)$$

Therefore:

$$y_i = x_i * (price\_per\_mq \pm error)$$

Instead of computing `Y` with **explicit for loops**, we exploit Numpy.

Run the next cell to compute `Y`.

In [21]:

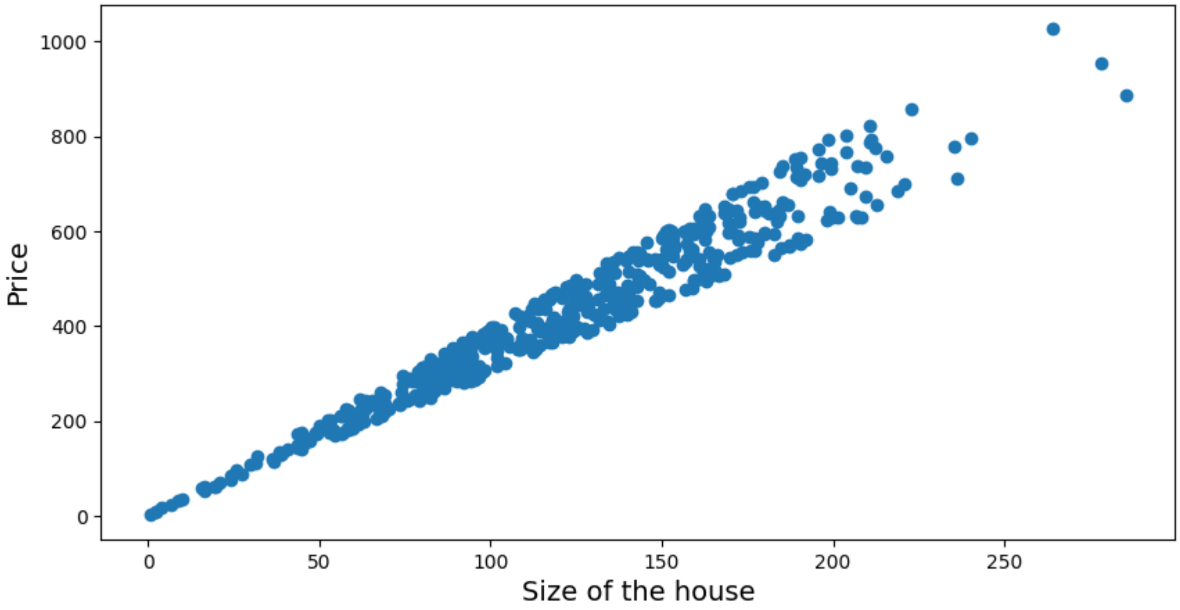
```
price_per_mq = 3.5 # thousands of euros per square meter
error = np.random.random((500,)) - 0.5 # generate a random number in range [
Y =  X*(price_per_mq + error) # compute the price for each house. Notice tha

print("Y Shape", Y.shape)
print("Y Minimum value: {:.2f}".format(Y.min()))
print("Y Maximum value: {:.2f}".format(Y.max()))
print("Y mean: {:.2f}".format(Y.mean()))
print("Y standard deviation: {:.2f}".format(Y.std()))
print("First ten element of Y: ", Y[:5])
```

```
Y Shape (500,)
Y Minimum value: 1.66
Y Maximum value: 1026.98
Y mean: 422.94
Y standard deviation: 177.55
First ten element of Y:  [358.36393775  22.75803392 719.55747059 713.0445834
4 540.26841245]
```

Run the next cell to **plot the generated points in the plane**, with the size of the houses on the *x* axis and the prices on the *y* axis.

In [22]:

```
plot_2d_points(X,Y)
```

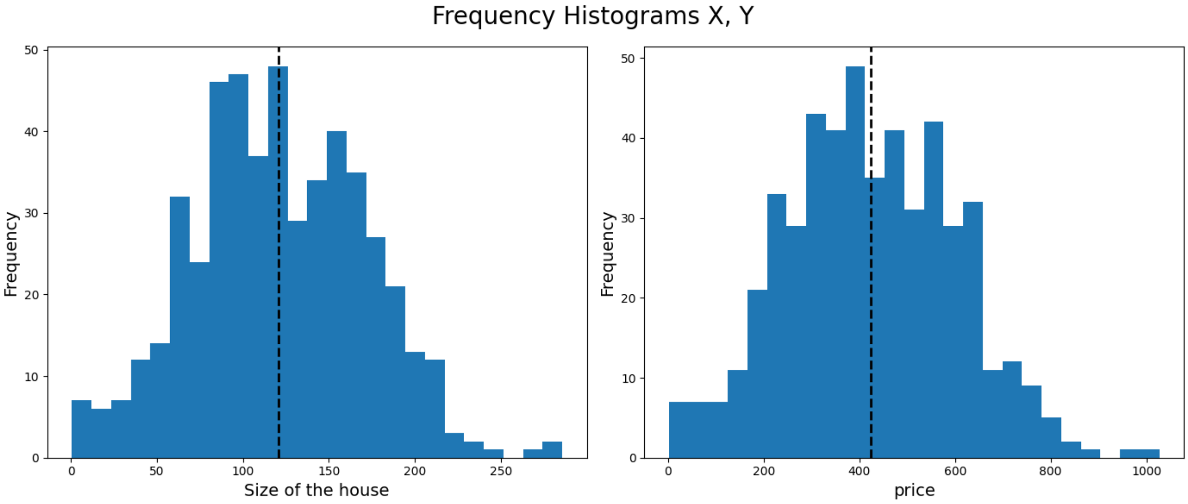


You can notice that the value of `X` are in a range `[0, 300]` (with some randomness).

Run the next cell to plot the frequency histograms of the size of the houses and prices. The values follow a normal distribution.

In [24]:

```
plot_distributions([X,Y], ["Size of the house", "price"])
```



Exercise 4.1

Now, you have to perform the **Min-Max normalization** of the input features. After the normalization, all the values of `X` must be in the range `[0, 1]`.

Perform the **Min-Max normalization** of the vector `X` and assign the normalized vector into a variable `X_norm`. Remember that the formula for the **Min-Max normalization** is the following:

$$X\_norm = \frac{(x - x_{min})}{(x_{max} - x_{min})}$$

NOTE: Exploit Numpy instead of lists or explicit for loops!

▶ **Hints**

In [26]:

```
#### START CODE HERE ####
#### Ideally 1 line ####
X_norm = (X - X.min()) / (X.max() - X.min())
#### END CODE HERE ####

print("Minimum value after norm: {:.2f}".format(X_norm.min()))
print("Maximum value after norm: {:.2f}".format(X_norm.max()))
print("Mean after norm: {:.2f}".format(X_norm.mean()))
print("Standard deviation after norm: {:.2f}".format(X_norm.std()))
print("First ten element of X_norm: ", X_norm[:5])
```

```
Minimum value after norm: 0.00
Maximum value after norm: 1.00
Mean after norm: 0.42
Standard deviation after norm: 0.17
First ten element of X_norm:  [0.32215778 0.02119521 0.67064224 0.6616627
0.56269745]
```

Expected output

`Minimum value after norm: 0.0`

`Maximum value after norm: 1.0`

`Mean after norm: 0.5` (with some randomness)

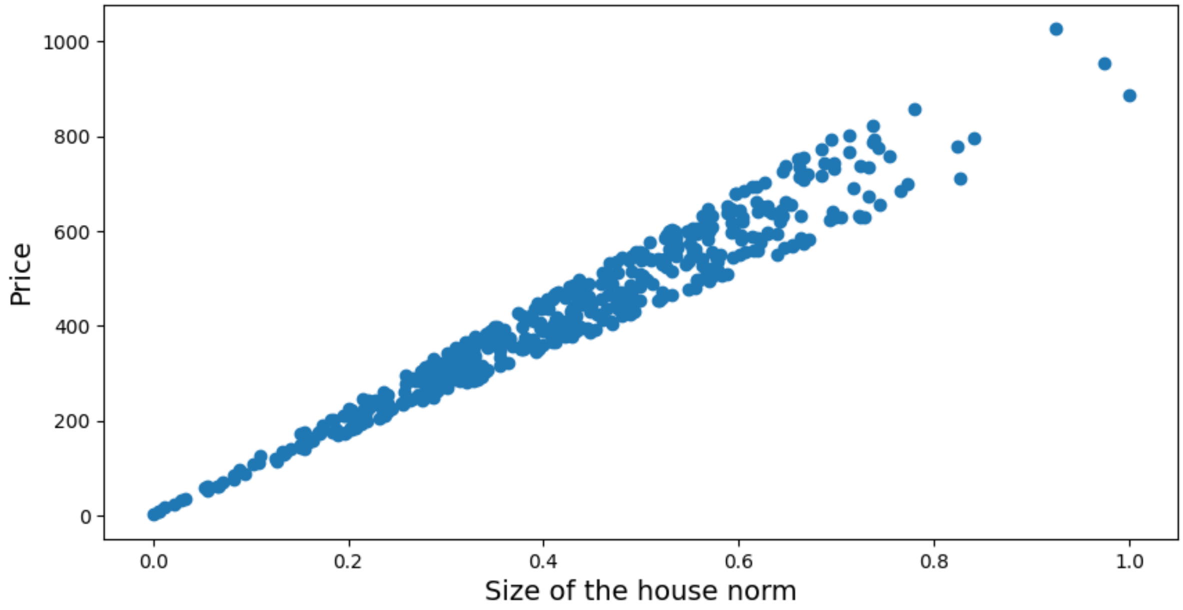
`Standard deviation after norm: 0.15` (with some randomness)

`First ten element of X_norm: [list with 5 floats between 0 and 1]` (with some randomness)

If you implemented the normalization correctly, the min and the max values of `X_norm` must be 0 and 1, respectively. Now, run the next cell to plot the **normalized** points in the plane, with the size of the houses on the *x* axis and the prices on the *y* axis.

In [27]:

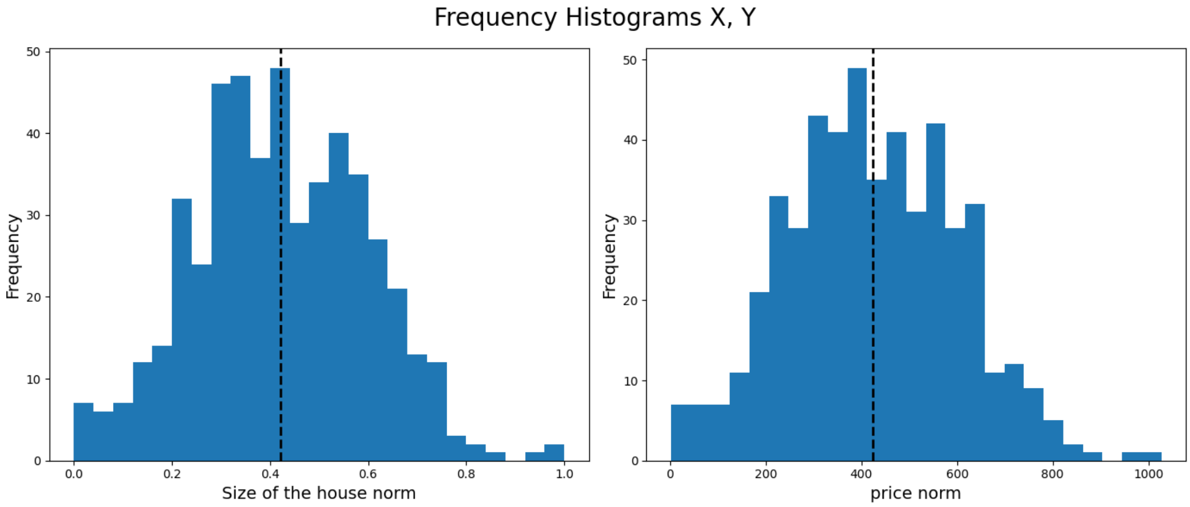
```
plot_2d_points(X_norm,Y,True)
```



You can notice that the values in the *x* axis are now in the range `[0, 1]`. But the relations between data are preserved (i.e., the points are distributed in the same way in the plane).

Run the next cell to plot the frequency histograms of the **normalized** size of the houses and prices.

```
In [28]: plot_distributions([X_norm,Y], ["Size of the house norm", "price norm"])
```



After the **Min-Max normalization**, the values of `X` are rescaled in the range [0, 1]. Again, the distribution is preserved.

Congratulations! you have correctly normalized a one-dimensional array of features using Numpy.

Now let's try a two-dimensional array.

4.2 Min-Max normalization of two-dimensional data

Now we will move to a **2-dimensional case**. We create an artificial dataset `X` with 2 dimensions: the **size of the house (mq)** (in the first column of `X`) and the **number of**

rooms (in the second column of X). The target amount that a Machine Learning algorithm would like to estimate is still the **price** (in thousands of €) (in the Y). In this case, the task of the algorithm is to predict the price given the size of the house and the number of rooms. Therefore, the input features X is a two-dimensional array. We will see how to train a Machine Learning algorithm to predict the price of the house Y given the size of the house and the number of rooms X in the next lectures.

Firstly, you will create a 2-dimensional array X containing the size of the houses and the number of rooms for 500 synthetic generated samples. The following cells will create the dataset X and the target value of Y .

In [29]:

```
n_samples = 500

mu, sigma = 120, 50 # mean and standard deviation
X_size = np.random.normal(mu, sigma, (500,)) # generate a gaussian distribut
X_size = np.abs(X_size) # only positive values make sense for the size of th

mu, sigma = 3, 2 # mean and standard deviation of the houses' number of room
X_rooms = np.random.normal(mu, sigma, n_samples) # Generate 500 samples (num
X_rooms = X_rooms.astype(int)
X_rooms = X_rooms+np.min(X_rooms)*-1+1 # move the samples with a minumum num

X = np.hstack((X_size.reshape(-1, 1), X_rooms.reshape(-1, 1)))

print("X Shape", X.shape)
print("X size of the house - Minimum value: {:.2f}".format(X[:,0].min()))
print("X size of the house - Maximum value: {:.2f}".format(X[:,0].max()))
print("X size of the house - mean: {:.2f}".format(X[:,0].mean()))
print("X size of the house - standard deviation: {:.2f}".format(X[:,0].std()
print("First ten element of X size of the house: {}".format(X[:5,0]))
print("X n rooms - Minimum value: {:.2f}".format(X[:,1].min()))
print("X n rooms - Maximum value: {:.2f}".format(X[:,1].max()))
print("X n rooms - mean: {:.2f}".format(X[:,1].mean()))
print("X n rooms - standard deviation: {:.2f}".format(X[:,1].std()))
print("First ten element of X n rooms: {}".format(X[:5,1]))
```

```
X Shape (500, 2)
X size of the house - Minimum value: 2.51
X size of the house - Maximum value: 274.84
X size of the house - mean: 122.90
X size of the house - standard deviation: 48.97
First ten element of X size of the house: [178.11975759 186.8402547  204.865
39829 169.39274102 141.94816472]
X n rooms - Minimum value: 1.00
X n rooms - Maximum value: 12.00
X n rooms - mean: 6.67
X n rooms - standard deviation: 1.86
First ten element of X n rooms: [ 5.  4.  5.  6. 10.]
```

Notice that the input features in X (i.e., the size of the house and the number of rooms) have a **different scale**. This can be **problematic** when training a learning algorithm.

Now compute the price of each house as a linear function of the input features X with an error term. The price of the house is computed as follows:

$$price = size * (price\_per\_mq \pm error\_size) + n\_rooms * (increment\_per\_room \pm$$

Therefore:

$$y_i = x_i[0] * (price\_per\_mq \pm error\_size) + x_i[1] * (increment\_per\_room \pm error\_r$$

Instead of computing `Y` with **explicit for loops**, we exploit Numpy.

Run the next cell to compute `Y`.

In [30]:

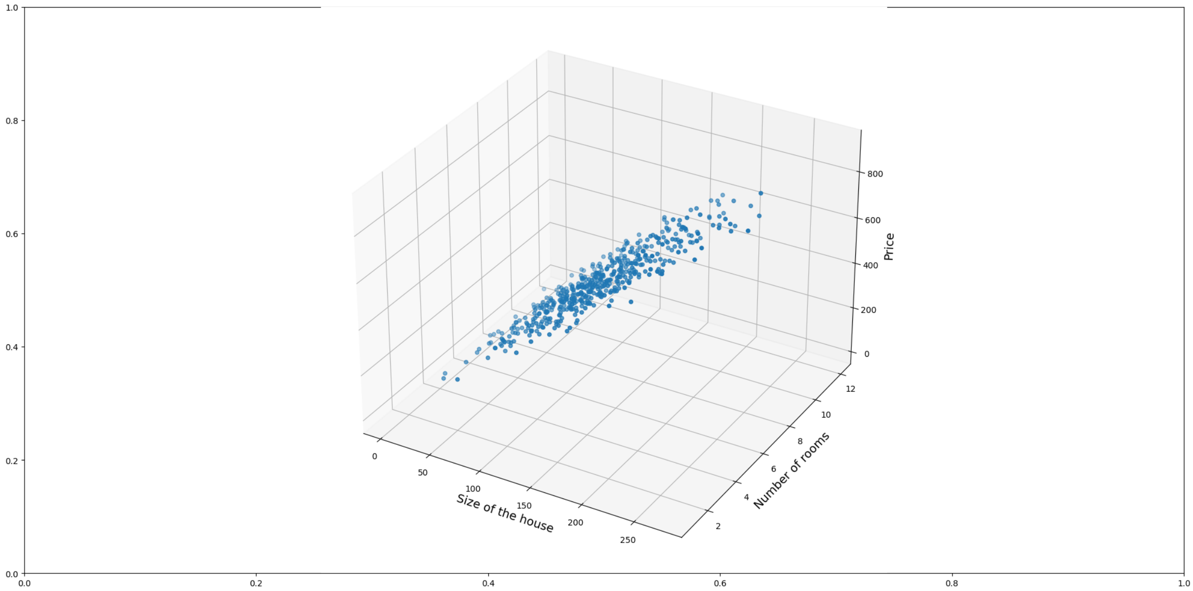
```
price_per_mq = 3.5 # tousands of euros per square metre
increment_per_room = 0.05
error_size = np.random.random((500,)) - 0.5 # generate a random number in ra
error_rooms = (np.random.random((500,)) - 0.5)/10 # generate a random number
Y =  X[:,0]*(price_per_mq + error_size) + X[:,1]*(increment_per_room + error

print("Y Shape", Y.shape)
print("Y Minimum value: {:.2f}".format(Y.min()))
print("Y Maximum value: {:.2f}".format(Y.max()))
print("Y mean: {:.2f}".format(Y.mean()))
print("Y standard deviation: {:.2f}".format(Y.std()))
print("First ten element of Y: ", Y[:5])
```

```
Y Shape (500,)
Y Minimum value: 9.23
Y Maximum value: 912.65
Y mean: 426.97
Y standard deviation: 170.64
First ten element of Y:  [633.04293852 636.28084786 812.52547715 573.4092383
5 563.1302792 ]
```

Run the next cell to **plot the generated points in the space**, with the size of the houses on the *x* axis, the number of rooms on the *y* axis, and the prices on the *z* axis.

In [31]:

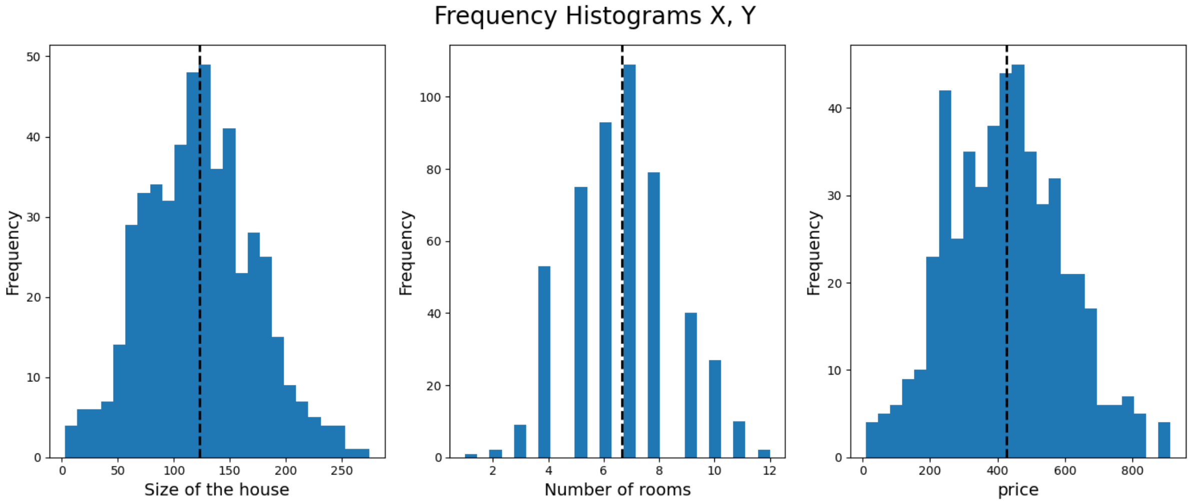
```
plot_3d_points(X, Y)
```



Run the next cell to plot the frequency histograms of the size of the houses, the number of rooms, and the prices. The values follow a normal distribution.

In [32]:

```
plot_distributions([X[:,0], X[:,1], Y], ["Size of the house", "Number of roo
```



Exercise 4.2

Now, you have to perform the **Min-Max normalization** of the input features. After the normalization, all the values of `X` (in both dimensions) must be in the range `[0, 1]`.

Perform the **Min-Max normalization** of the vector `X` and assign the normalized vector into a variable `X_norm`. Remember that the formula for the **Min-Max normalization** is the following:

$$X\_norm = \frac{(x - x_{min})}{(x_{max} - x_{min})}$$

Important: **this time, you should normalize each column separately** (i.e., the column with the size of the houses must be normalized with the mean and the standard deviation of the size of the houses in the dataset, while the column with the number of rooms must be normalized with the mean and the standard deviation of the number of rooms in the dataset).

NOTE: Exploit Numpy instead of lists or explicit for loops!

▶ **Hints**

In [33]:

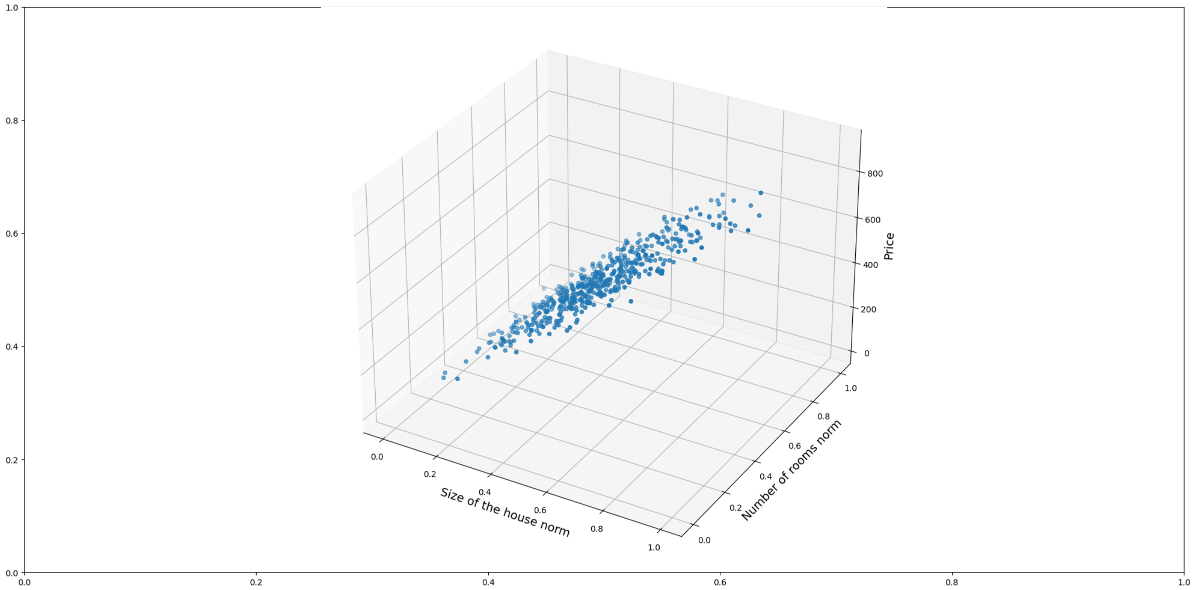
```
#### START CODE HERE ####
#### Ideally 1 line ####
X_norm = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
#### END CODE HERE ####

print("X_norm Shape", X.shape)
print("X_norm size of the house - Minimum value: {:.2f}".format(X_norm[:,0].
print("X_norm size of the house - Maximum value: {:.2f}".format(X_norm[:,0].
print("X_norm size of the house - mean: {:.2f}".format(X_norm[:,0].mean()))
print("X_norm size of the house - standard deviation: {:.2f}".format(X_norm[
print("First ten element of X_norm size of the house: {}".format(X_norm[:5,0
print("X_norm n rooms - Minimum value: {:.2f}".format(X_norm[:,1].min()))
print("X_norm n rooms - Maximum value: {:.2f}".format(X_norm[:,1].max()))
print("X_norm n rooms - mean: {:.2f}".format(X_norm[:,1].mean()))
print("X_norm n rooms - standard deviation: {:.2f}".format(X_norm[:,1].std()
print("First ten element of X_norm n rooms: {}".format(X_norm[:5,1]))
```

```
X_norm Shape (500, 2)
X_norm size of the house - Minimum value: 0.00
X_norm size of the house - Maximum value: 1.00
X_norm size of the house - mean: 0.44
X_norm size of the house - standard deviation: 0.18
First ten element of X_norm size of the house: [0.64484042 0.67686293 0.7430
53   0.61279397 0.51201483]
X_norm n rooms - Minimum value: 0.00
X_norm n rooms - Maximum value: 1.00
X_norm n rooms - mean: 0.52
X_norm n rooms - standard deviation: 0.17
First ten element of X_norm n rooms: [0.36363636 0.27272727 0.36363636 0.454
54545 0.81818182]
```

If you implemented the normalization correctly, the min and the max values of `X_norm` must be 0 and 1, respectively, for both dimensions (i.e., the size of the houses and the number of rooms).

Now, run the next cell to plot the **normalized** points in the space, with the size of the houses on the *x* axis, the number of rooms on the *y* axis, and the prices on the *z* axis.

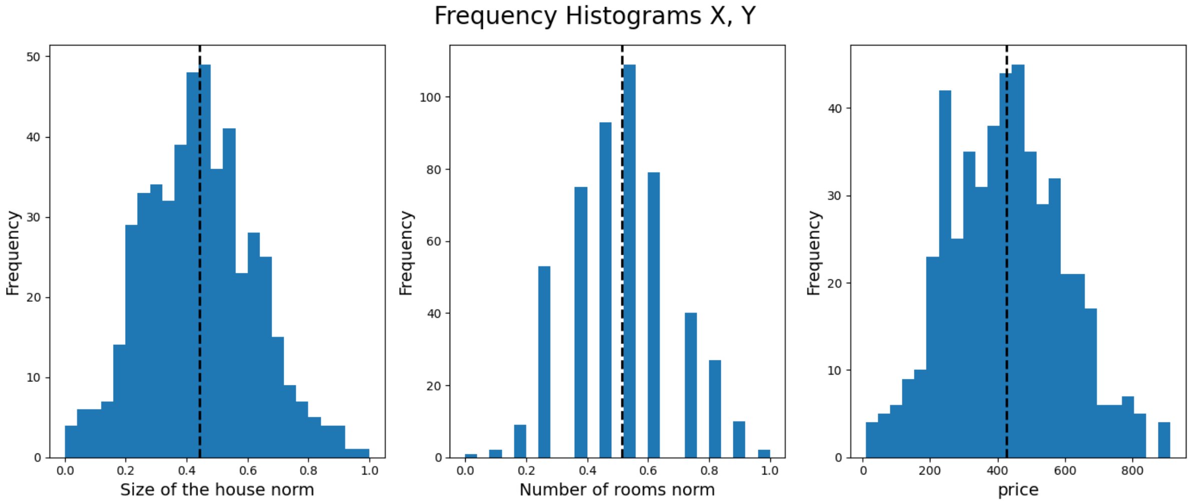
```
In [34]: plot_3d_points(X_norm, Y, True)
```



You can notice that the values in the *X* axis are now in the range `[0, 1]` (i.e., *x* and *y* in the space). But the relations between data are preserved (i.e., the points are distributed in the same way in the plane).

Run the next cell to plot the frequency histograms of the **normalized** size of the houses, number of rooms, and prices.

```
In [35]: plot_distributions([X_norm[:,0], X_norm[:,1], Y], ["Size of the house norm",
```



After the **Min-Max normalization**, the values of `X` in both axis (i.e., size of the house and number of rooms) are rescaled in the range [0, 1]. Again, the distributions are preserved.

Congratulations! you have also correctly normalized a two-dimensional array of features using Numpy.

```
In [ ]:
```