



Machine Learning - Module 00

Stepping into Machine Learning

Summary: You will start by reviewing some linear algebra and statistics. Then you will implement your first model and learn how to evaluate its performances.

Notions and ressources

Notions of the module

Sum, mean, variance, standard deviation, vectors and matrices operations. Hypothesis, model, regression, loss function.

Useful Ressources

You are strongly advise to use the following resource: [Machine Learning MOOC - Stanford](#)
Here are the sections of the MOOC that are relevant for today's exercises:

Week 1

Introduction

- What is Machine Learning? (Video + Reading)
- Supervised Learning (Video + Reading)
- Unsupervised Learning (Video + Reading)
- Review (Reading + Quiz)

Linear Regression with One Variable

- Model Representation (Video + Reading)
- Cost Function (Video + Reading)
- Cost Function - Intuition I (Video + Reading)
- Cost Function - Intuition II (Video + Reading)
- *Keep what remains for tomorrow ;)*

Linear Algebra Review

- Matrices and Vectors (Video + Reading)
- Addition and Scalar Multiplication (Video + Reading)
- Matrix Vector Multiplication (Video + Reading)
- Matrix Matrix Multiplication (Video + Reading)
- Matrix Multiplication Properties (Video + Reading)
- Inverse and Transpose (Video + Reading)
- Review (Reading + Quiz)

Common Instructions


- The version of Python recommended to use is 3.7, you can check the version of Python with the following command: `python -V`
- The norm: during this piscine, it is recommended to follow the [PEP 8 standards](#), though it is not mandatory. You can install [pycodestyle](#) which is a tool to check your Python code.
- The function `eval` is never allowed.
- The exercises are ordered from the easiest to the hardest.
- Your exercises are going to be evaluated by someone else, so make sure that your variable names and function names are appropriate and civil.
- Your manual is the internet.
- You can also ask questions in the `#bootcamps` channel in the [42AI](#) or [42born2code](#).
- If you find any issue or mistakes in the subject please create an issue on [42AI repository on Github](#).
- We encourage you to create test programs for your project even though this work **won't have to be submitted and won't be graded**. It will give you a chance to easily test your work and your peers' work. You will find those tests especially useful during your defence. Indeed, during defence, you are free to use your tests and/or the tests of the peer you are evaluating.
- Submit your work to your assigned git repository. Only the work in the git repository will be graded. If Deepthought is assigned to grade your work, it will be run after your peer-evaluations. If an error happens in any section of your work during Deepthought's grading, the evaluation will stop.

Contents

I	Exercise 00	5
II	Exercise 01	10
III	Exercise 02	13
IV	Exercise 03	19
V	Exercise 04	23
VI	Exercise 05	26
VII	Exercise 06	29
VIII	Exercise 07	34
IX	Exercise 08	36
X	Exercise 09	40
XI	Conclusion - What you have learned	44

Chapter I

Exercise 00

	Exercise : 00
The Matrix	
Turn-in directory : <i>ex00/</i>	
Files to turn in : <code>matrix.py</code> , <code>test.py</code>	
Forbidden functions : <code>Numpy</code>	

Objective

Manipulation and understanding of basic matrix operations.

In this exercise, you have to create a **Matrix** and a **Vector** class. The goal is to have matrices and be able to perform both matrix-matrix operation and matrix-vector operations with them.

Instructions

You will provide a testing file to prove that your classes works as expected.

Your **Matrix** class must have the following 2 attributes:

- **data**: list of lists
- **shape**: the dimensions of the matrix as a tuple (rows, columns)

You should be able to initialize the object with either:

- the elements of the matrix as a list of lists: `Matrix([[1.0, 2.0], [3.0, 4.0]])`
- a shape: `Matrix((3, 3))` (the matrix will be filled with zeros by default)

You will implement all the following built-in functions (called **magic/special methods**) for your **Matrix** class:

```
# add : only matrices of same dimensions.
__add__
__radd__
# sub : only matrices of same dimensions.
__sub__
__rsub__
# div : only scalars.
__truediv__
__rtruediv__
# mul : scalars, vectors and matrices , can have errors with vectors and matrices,
# returns a Vector if we perform Matrix * Vector multiplication.
__mul__
__rmul__
__str__
__repr__
```

You will also implement:

- a **.T()** method which returns the transpose of the matrix (see examples below),

Then, you must create a **Vector** class that inherit from **Matrix**. At initialization, you must check that a column or a row vector is passed as the data argument. If not, you must send an error message :

```
v1 = Vector([[1, 2, 3]]) # create a row vector
v2 = Vector([[1], [2], [3]]) # create a column vector
v3 = Vector([[1, 2], [3, 4]]) # return an error
```

For **Vector**, you must implement:

- a **.dot(self, v: Vector)** method which returns the dot product between the current vector and **v**. If shapes don't match, you must properly display errors.



Caution: when you do operations between **Vector**, it must return a **Vector** and not a **Matrix**



`type(self)`

Examples

```
m1 = Matrix([[0.0, 1.0], [2.0, 3.0], [4.0, 5.0]])
m1.shape
# Output:
(3, 2)
```

```
m1.T()
# Output:
Matrix([[0., 2., 4.], [1., 3., 5.]])
```

```
m1.T().shape
# Output:
(2, 3)
```

```
m1 = Matrix([[0., 2., 4.], [1., 3., 5.]])
m1.shape
# Output:
(2, 3)
```

```
m1.T()
# Output:
Matrix([[0.0, 1.0], [2.0, 3.0], [4.0, 5.0]])
```

```
m1.T().shape
# Output:
(3, 2)
```

```
m1 = Matrix([[0.0, 1.0, 2.0, 3.0],
             [0.0, 2.0, 4.0, 6.0]])
```

```
m2 = Matrix([[0.0, 1.0],
             [2.0, 3.0],
             [4.0, 5.0],
             [6.0, 7.0]])
```

```
m1 * m2
# Output:
Matrix([[28., 34.], [56., 68.]])
```

```
m1 = Matrix([[0.0, 1.0, 2.0],
             [0.0, 2.0, 4.0]])
v1 = Vector([[1], [2], [3]])
```

```
m1 * v1
# Output:
Matrix([[8], [16]])
# Or: Vector([[8], [16]])
```

```
v1 = Vector([[1], [2], [3]])
v2 = Vector([[2], [4], [8]])
```

```
v1 + v2
# Output:
Vector([[3], [6], [11]])
```


Mathematical notions

Matrix - vector operations

- Multiplication between a $(m \times n)$ matrix and a vector of dimension n

$$Xy = \begin{bmatrix} x_1^{(1)} & \dots & x_n^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(m)} & \dots & x_n^{(m)} \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x^{(1)} \cdot y \\ \vdots \\ x^{(m)} \cdot y \end{bmatrix}$$

In other words:

$$Xy = \begin{bmatrix} \sum_{i=1}^n x_i^{(1)} \cdot y_i \\ \vdots \\ \sum_{i=1}^n x_i^{(m)} \cdot y_i \end{bmatrix}$$

Matrix - matrix operations

- Addition between two matrices of same dimension $(m \times n)$,

$$X + Y = \begin{bmatrix} x_1^{(1)} & \dots & x_n^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(m)} & \dots & x_n^{(m)} \end{bmatrix} + \begin{bmatrix} y_1^{(1)} & \dots & y_n^{(1)} \\ \vdots & \ddots & \vdots \\ y_1^{(m)} & \dots & y_n^{(m)} \end{bmatrix} = \begin{bmatrix} x_1^{(1)} + y_1^{(1)} & \dots & x_n^{(1)} + y_n^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(m)} + y_1^{(m)} & \dots & x_n^{(m)} + y_n^{(m)} \end{bmatrix}$$

- Substraction between two matrices of same dimension $(m \times n)$,

$$X - Y = \begin{bmatrix} x_1^{(1)} & \dots & x_n^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(m)} & \dots & x_n^{(m)} \end{bmatrix} - \begin{bmatrix} y_1^{(1)} & \dots & y_n^{(1)} \\ \vdots & \ddots & \vdots \\ y_1^{(m)} & \dots & y_n^{(m)} \end{bmatrix} = \begin{bmatrix} x_1^{(1)} - y_1^{(1)} & \dots & x_n^{(1)} - y_n^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(m)} - y_1^{(m)} & \dots & x_n^{(m)} - y_n^{(m)} \end{bmatrix}$$

- Multiplication or division between one matrix $(m \times n)$ and one scalar,

$$\alpha X = \alpha \begin{bmatrix} x_1^{(1)} & \dots & x_n^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(m)} & \dots & x_n^{(m)} \end{bmatrix} = \begin{bmatrix} \alpha x_1^{(1)} & \dots & \alpha x_n^{(1)} \\ \vdots & \ddots & \vdots \\ \alpha x_1^{(m)} & \dots & \alpha x_n^{(m)} \end{bmatrix}$$

- Multiplication between two matrices of compatible dimension: $(m \times n)$ and $(n \times p)$,

$$XY = \begin{bmatrix} x_1^{(1)} & \dots & x_n^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(m)} & \dots & x_n^{(m)} \end{bmatrix} \begin{bmatrix} y_1^{(1)} & \dots & y_p^{(1)} \\ \vdots & \ddots & \vdots \\ y_1^{(n)} & \dots & y_p^{(n)} \end{bmatrix} = \begin{bmatrix} x^{(1)} \cdot y_1 & \dots & x^{(1)} \cdot y_p \\ \vdots & \ddots & \vdots \\ x^{(m)} \cdot y_1 & \dots & x^{(m)} \cdot y_p \end{bmatrix}$$


In other words:

$$XY = \begin{bmatrix} \sum_{i=1}^n x_i^{(1)} \cdot y_1^{(i)} & \dots & \sum_{i=1}^n x_i^{(1)} \cdot y_p^{(i)} \\ \vdots & \ddots & \vdots \\ \sum_{i=1}^n x_i^{(m)} \cdot y_1^{(i)} & \dots & \sum_{i=1}^n x_i^{(m)} \cdot y_p^{(i)} \end{bmatrix}$$

Don't forget to handle all kinds of errors properly! *You are expected to demonstrate the errors management between Matrix object and Vector object and scalar plus the functionality of the accepted operations.*

Chapter II

Exercise 01

	Exercise : 01
TinyStatistician	
Turn-in directory : <i>ex01/</i>	
Files to turn in : <code>TinyStatistician.py</code>	
Forbidden functions : Any functions which calculates mean, median quartiles, percentiles, variance or standard deviation.	

Objective

These exercises are key assignments from the last bootcamp. If you haven't completed them yet, you should finish them first before you continue with today's exercises.

Instructions

Create a class named `TinyStatistician` with the following methods.

All methods take a list or a `numpy.array` as first parameter. You have to protect your functions against input errors.

- **mean(x)**: computes the mean of a given non-empty list or array x , using a for-loop. The method returns the mean as a float, otherwise `None` if x is an empty list or array, or a non expected type object. This method should not raise any Exception. Given a vector x of dimension $m * 1$, the mathematical formula of its mean is:

$$\bar{x} = \frac{\sum_{i=1}^m x_i}{m}$$

- **median(x)**: computes the median, which is also the 50th percentile, of a given non-empty list or array x . The method returns the median as a float, otherwise `None`

if x is an empty list or array or a non expected type object. This method should not raise any Exception.

- **quartile(x)**: computes the 1st and 3rd quartiles, also called the 25th percentile and the 75th percentile, of a given non-empty list or array x . The method returns the quartiles as a list of 2 floats, otherwise None if x is an empty list or array or a non expected type object. This method should not raise any Exception.
- **percentile(x, p)**: computes the expected percentile of a given non-empty list or array x . The method returns the percentile as a float, otherwise None if x is an empty list or array or a non expected type object. The second parameter is the wished percentile. This method should not raise any Exception.
- **var(x)**: computes the sample variance of a given non-empty list or array x . The method returns the sample variance as a float, otherwise None if x is an empty list or array or a non expected type object. This method should not raise any Exception.

Given a vector x of dimension $m * 1$ representing the a sample of a data population, the mathematical formula of its variance is:

$$\sigma^2 = \frac{\sum_{i=1}^m (x_i - \bar{x})^2}{m - 1} = \frac{\sum_{i=1}^m [x_i - (\frac{1}{m} \sum_{j=1}^m x_j)]^2}{m - 1}$$

- **std(x)**: computes the sample standard deviation of a given non-empty list or array x . The method returns the sample standard deviation as a float, otherwise None if x is an empty list or array or a non expected type object. This method should not raise any Exception.

Given a vector x of dimension $m * 1$, the mathematical formula of the sample standard deviation is:

$$\sigma = \sqrt{\frac{\sum_{i=1}^m (x_i - \bar{x})^2}{m - 1}} = \sqrt{\frac{\sum_{i=1}^m [x_i - (\frac{1}{m} \sum_{j=1}^m x_j)]^2}{m - 1}}$$

Examples

```
a = [1, 42, 300, 10, 59]
TinyStatistician().mean(a)
# Output:
82.4

TinyStatistician().median(a)
# Output:
42.0

TinyStatistician().quartile(a)
# Output:
[10.0, 59.0]

TinyStatistician().percentile(a, 10)
# Output:
4.6

TinyStatistician().percentile(a, 15)
# Output:
6.4

TinyStatistician().percentile(a, 20)
# Output:
8.2

TinyStatistician().var(a)
# Output:
15349.3

TinyStatistician().std(a)
# Output:
123.89229193133849
```



numpy uses a different definition of percentile, it does linear interpolation between the two closest list element to the percentile. Be sure to understand the difference between the population and the sample definition for the statistic metrics.

Chapter III

Exercise 02

Interlude - Predict, Evaluate, Improve

A computer program is said to learn from experience E , with respect to some class of tasks T , and performance measure P , if its performance at tasks in T , as measured by P , improves with experience E .

*Tom Mitchell,
Machine Learning, 1997*

In other words to learn you have to improve.

To improve you have to evaluate your performance.

To evaluate your performance you need to start performing on the task you want to be good at.

One of the most common tasks in Machine Learning is **prediction**.

This will be your algorithm's task.

This will be your task.

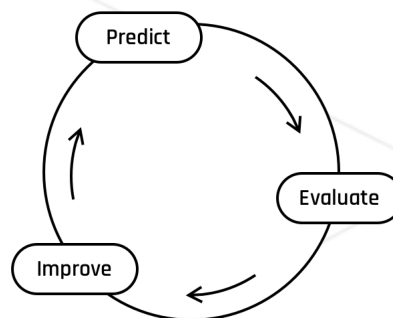


Figure III.1: cycle neutral

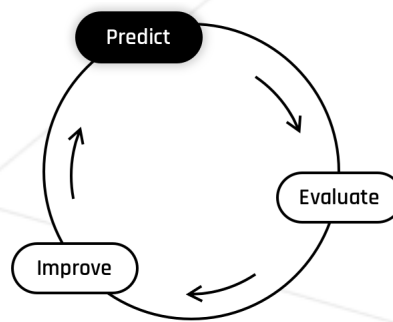


Figure III.2: cycle predict

Predict

A very simple model

We have some data. We want to model it.

- First we need to *make an assumption*, or hypothesis, *about the structure of the data and the relationship between the variables*.
- Then we can *apply that hypothesis to our data to make predictions*.

$$\text{hypothesis}(\text{data}) = \text{predictions}$$

Hypothesis

Let's start with a very simple and intuitive **hypothesis** on how the price of a spaceship can be predicted based on the power of its engines.

We will consider that *the more powerful the engines are, the more expensive the spaceship is*.

Furthermore, we will assume that the price increase is **proportional** to the power increase. In other words, we will look for a **linear relationship** between the two variables.

This means that we will formulate the price prediction with a **linear equation**, that you might be already familiar with:

$$\hat{y} = ax + b$$

We add the $\hat{}$ symbol over the y to specify that \hat{y} (*pronounced y-hat*) is a **prediction** (or estimation) of the real value of y . The prediction is calculated with the **parameters** a and b and the input value x .

For example, if $a = 5$ and $b = 33$, then $\hat{y} = 5x + 33$.

But in Machine Learning, we don't like using the letters a and b . Instead we will use the following notation:

$$\hat{y} = \theta_0 + \theta_1 x$$

So if $\theta_0 = 33$ and $\theta_1 = 5$, then $\hat{y} = 33 + 5x$.

To recap, this linear equation is our **hypothesis**. Then, all we will need to do is find the right values for our parameters θ_0 and θ_1 and we will get a fully-functional prediction **model**.

Predictions

Now, how can we generate a set of predictions on an entire dataset? Let's consider a dataset containing m data points (or space ships), called **examples**.

What we do is stack the x and \hat{y} values of all examples in vectors of length m . The relation between the elements in our vectors can then be represented with the following formula:

$$\hat{y}^{(i)} = \theta_0 + \theta_1 x^{(i)} \quad \text{for } i = 1, \dots, m$$

Where:

- $\hat{y}^{(i)}$ is the i^{th} component of vector y
- $x^{(i)}$ is the i^{th} component of vector x

Which can be expressed as:

$$\hat{y} = \begin{bmatrix} \theta_0 + \theta_1 \times x^{(1)} \\ \vdots \\ \theta_0 + \theta_1 \times x^{(m)} \end{bmatrix}$$

For example,

$$\text{given } \theta = \begin{bmatrix} 33 \\ 5 \end{bmatrix} \text{ and } x = \begin{bmatrix} 1 \\ 3 \end{bmatrix} :$$

$$\hat{y} = h_{\theta}(x) = \begin{bmatrix} 33 + 5 \times 1 \\ 33 + 5 \times 3 \end{bmatrix} = \begin{bmatrix} 38 \\ 48 \end{bmatrix}$$

More information

Why the θ notation?

You might have two questions at the moment:


- **WTF is that weird symbol?** This strange symbol, θ , is called "theta".
- **Why use this notation instead of a and b , like we're used to?** Despite its seeming more complicated at first, the theta notation is actually meant to simplify your equations later on. Why? a and b are good for a model with two parameters, but you will soon need to build more complex models that take into account more variables than just x . You could add more letters like this: $\hat{y} = ax_1 + bx_2 + cx_3 + \dots + yx_{25} + z$ But how do you go beyond 26 parameters? And how easily can you tell what parameter is associated with, let's say, x_{19} ? That's why it becomes more handy to describe all your parameters using the theta notation and indices. With θ , you just have to increment the number to name the parameter: $\hat{y} = \theta_0 + \theta_1x_1 + \theta_2x_2 + \dots + \theta_{2468}x_{2468} \dots$ Easy right?

Another common notation

$$\hat{y} = h_{\theta}(x)$$

Because \hat{y} is calculated with our linear hypothesis using θ and x , it is sometimes written as $h_{\theta}(x)$. The h stands for *hypothesis*, and can be read as "*the result of our hypothesis h given x and θ* ".

Then if $x = 7$, we can calculate: $\hat{y} = h_{\theta}(x) = 33 + 5 \times 7 = 68$ We can now say that according to our linear model, the **predicted value** of y given ($x = 7$) is 68.

	Exercise : 02
Simple Prediction	
Turn-in directory : <i>ex02/</i>	
Files to turn in : prediction.py	
Forbidden functions : any functions which performs prediction	

Objective

Understand and manipulate the notion of hypothesis in machine learning.

You must implement the following formula as a function:

$$\hat{y}^{(i)} = \theta_0 + \theta_1 x^{(i)} \quad \text{for } i = 1, \dots, m$$

Where:

- x is a vector of dimension m , the vector of examples/features (without the y values),
- \hat{y} is a vector of dimension $m * 1$, the vector of predicted values,
- θ is a vector of dimension $2 * 1$, the vector of parameters,
- $y^{(i)}$ is the i^{th} component of vector y ,
- $x^{(i)}$ is the i^{th} component of vector x .

Instructions

In the `prediction.py` file, write the following function as per the instructions given below:

```
def simple_predict(x, theta):  
    """Computes the vector of prediction y_hat from two non-empty numpy.ndarray.  
    Args:  
        x: has to be a numpy.ndarray, a vector of dimension m * 1.  
        theta: has to be a numpy.ndarray, a vector of dimension 2 * 1.  
    Returns:  
        y_hat as a numpy.ndarray, a vector of dimension m * 1.  
        None if x or theta are empty numpy.ndarray.  
        None if x or theta dimensions are not appropriate.  
    Raises:  
        This function should not raise any Exception.  
    """  
    ... Your code ...
```

Examples

```
import numpy as np
x = np.arange(1,6)

# Example 1:
theta1 = np.array([5, 0])
simple_predict(x, theta1)
# Output:
array([5., 5., 5., 5., 5.])
# Do you understand why y_hat contains only 5's here?

# Example 2:
theta2 = np.array([0, 1])
simple_predict(x, theta2)
# Output:
array([1., 2., 3., 4., 5.])
# Do you understand why y_hat == x here?

# Example 3:
theta3 = np.array([5, 3])
simple_predict(x, theta3)
# Output:
array([ 8., 11., 14., 17., 20.])

# Example 4:
theta4 = np.array([-3, 1])
simple_predict(x, theta4)
# Output:
array([-2., -1.,  0.,  1.,  2.])
```

Chapter IV

Exercise 03

Interlude - A Simple Linear Algebra Trick

As you know, vectors and matrices can be multiplied to perform linear combinations. Let's do a little linear algebra trick to optimize our calculation and use matrix multiplication. If we add a column full of 1's to our vector of examples x , we can create the following matrix:

$$X' = \begin{bmatrix} 1 & x^{(1)} \\ \vdots & \vdots \\ 1 & x^{(m)} \end{bmatrix}$$

We can then rewrite our hypothesis as:

$$\hat{y}^{(i)} = \theta \cdot x'^{(i)} = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} \cdot \begin{bmatrix} 1 & x^{(i)} \end{bmatrix} = \theta_0 + \theta_1 x^{(i)}$$

Therefore, the calculation of each $\hat{y}^{(i)}$ can be done with only one vector multiplication.

But we can even go further, by calculating the whole \hat{y} vector in one operation:

$$\hat{y} = X' \cdot \theta = \begin{bmatrix} 1 & x^{(1)} \\ \vdots & \vdots \\ 1 & x^{(m)} \end{bmatrix} \cdot \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} = \begin{bmatrix} \theta_0 + \theta_1 x^{(1)} \\ \vdots \\ \theta_0 + \theta_1 x^{(m)} \end{bmatrix}$$


We can now get to the same result as in the previous exercise with just a single multiplication between our brand new X' matrix and the θ vector!

A Note on Notation

In further Interludes, we will use the following convention:

- Capital letters represent matrices (e.g.: X)

- Lower case letters represent vectors and scalars (e.g.: $x^{(i)}$, y)

	Exercise : 03
Add Intercept	
Turn-in directory : <i>ex03/</i>	
Files to turn in : tools.py	
Forbidden functions : None	

Objective

Understand and manipulate the notion of hypothesis in machine learning. You must implement a function which adds an extra column of 1's on the left side of a given vector or matrix.

Instructions

In the tools.py file create the following function as per the instructions given below:

```
def add_intercept(x):  
    """Adds a column of 1's to the non-empty numpy.array x.  
    Args:  
        x: has to be a numpy.array of dimension m * n.  
    Returns:  
        X, a numpy.array of dimension m * (n + 1).  
        None if x is not a numpy.array.  
        None if x is an empty numpy.array.  
    Raises:  
        This function should not raise any Exception.  
    """  
    ... Your code ...
```

Examples


```
import numpy as np

# Example 1:
x = np.arange(1,6)
add_intercept(x)
# Output:
array([[1., 1.],
       [1., 2.],
       [1., 3.],
       [1., 4.],
       [1., 5.]])

# Example 2:
y = np.arange(1,10).reshape((3,3))
add_intercept(y)
# Output:
array([[1., 1., 2., 3.],
       [1., 4., 5., 6.],
       [1., 7., 8., 9.]])
```

Chapter V

Exercise 04

	Exercise : 04
Prediction	
Turn-in directory : <i>ex04/</i>	
Files to turn in : prediction.py	
Forbidden functions : None	

Objective

Understand and manipulate the notion of hypothesis in machine learning.

You must implement the following formula as a function:

$$\hat{y}^{(i)} = \theta_0 + \theta_1 x^{(i)} \quad \text{for } i = 1, \dots, m$$

Where:

- $\hat{y}^{(i)}$ is the i^{th} component of vector \hat{y}
- \hat{y} is a vector of dimension m , the vector of predicted values
- θ is a vector of dimension 2×1 , the vector of parameters
- $x^{(i)}$ is the i^{th} component of vector x
- x is a vector of dimension m , the vector of examples

But this time you have to do it with the linear algebra trick!

$$\hat{y} = X' \cdot \theta = \begin{bmatrix} 1 & x^{(1)} \\ \vdots & \vdots \\ 1 & x^{(m)} \end{bmatrix} \cdot \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} = \begin{bmatrix} \theta_0 + \theta_1 x^{(1)} \\ \vdots \\ \theta_0 + \theta_1 x^{(m)} \end{bmatrix}$$



- the argument x is an m vector
- θ is a 2×1 vector.

You have to transform x into X' to fit the dimension of θ !

Instructions

In the prediction.py file create the following function as per the instructions given below:

```
def predict_(x, theta):  
    """Computes the vector of prediction y_hat from two non-empty numpy.array.  
    Args:  
        x: has to be a numpy.array, a vector of dimension m * 1.  
        theta: has to be a numpy.array, a vector of dimension 2 * 1.  
    Returns:  
        y_hat as a numpy.array, a vector of dimension m * 1.  
        None if x and/or theta are not numpy.array.  
        None if x or theta are empty numpy.array.  
        None if x or theta dimensions are not appropriate.  
    Raises:  
        This function should not raise any Exceptions.  
    """  
    ... Your code ...
```

Examples

```
import numpy as np
x = np.arange(1,6)

# Example 1:
theta1 = np.array([[5], [0]])
predict_(x, theta1)
# Output:
array([[5.], [5.], [5.], [5.], [5.]])
# Do you remember why y_hat contains only 5's here?


# Example 2:
theta2 = np.array([[0], [1]])
predict_(x, theta2)
# Output:
array([[1.], [2.], [3.], [4.], [5.]])
# Do you remember why y_hat == x here?

# Example 3:
theta3 = np.array([[5], [3]])
predict_(X, theta3)
# Output:
array([[ 8.], [11.], [14.], [17.], [20.]])

# Example 4:
theta4 = np.array([[-3], [1]])
predict_(x, theta4)
# Output:
array([[-2.], [-1.], [ 0.], [ 1.], [ 2.]])
```

Chapter VI

Exercise 05

	Exercise : 05
Let's Make Nice Plots	
Turn-in directory : <code>ex05/</code>	
Files to turn in : <code>plot.py</code>	
Forbidden functions : <code>None</code>	



For you information, the task we are performing here is called **regression**. It means that we are trying to predict a continuous numerical attribute for all examples (like a price, for instance). Later in the bootcamp, you will see that we can predict other things such as categories.

Objective

You must implement a function to plot the data and the prediction line (or regression line). You will plot the data points (with their x and y values), and the prediction line that represents your hypothesis (h_θ).

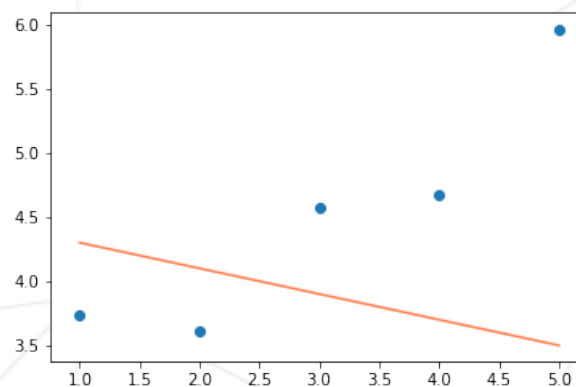
Instructions

In the plot.py file, create the following function as per the instructions given below:

```
def plot(x, y, theta):  
    """Plot the data and prediction line from three non-empty numpy.array.  
    Args:  
        x: has to be an numpy.array, a vector of dimension m * 1.  
        y: has to be an numpy.array, a vector of dimension m * 1.  
        theta: has to be an numpy.array, a vector of dimension 2 * 1.  
    Returns:  
        Nothing.  
    Raises:  
        This function should not raise any Exceptions.  
    """  
    ... Your code ...
```

Examples

```
import numpy as np  
x = np.arange(1,6)  
y = np.array([3.74013816, 3.61473236, 4.57655287, 4.66793434, 5.95585554])  
  
# Example 1:  
theta1 = np.array([[4.5],[-0.2]])  
plot(x, y, theta1)  
# Output:
```



```
# Example 2:  
theta2 = np.array([[-1.5],[2]])  
plot(x, y, theta2)  
# Output:
```

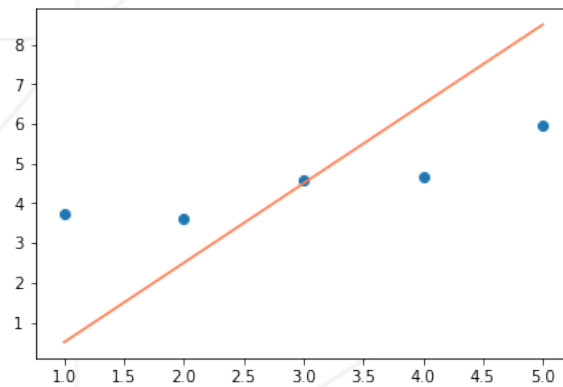


Figure VI.1: Example 2

```
# Example 3:  
theta3 = np.array([[3],[0.3]])  
plot(x, y, theta3)  
# Output:
```

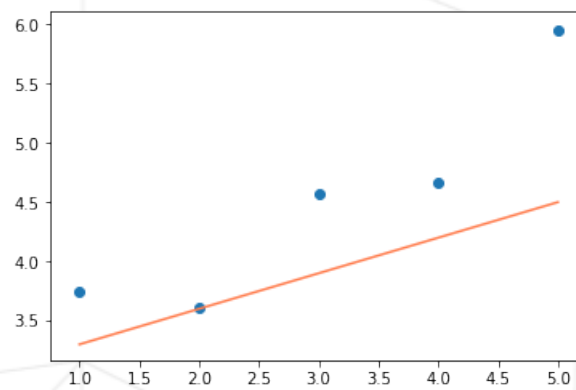


Figure VI.2: Example 3

Chapter VII

Exercise 06

Interlude - Evaluate

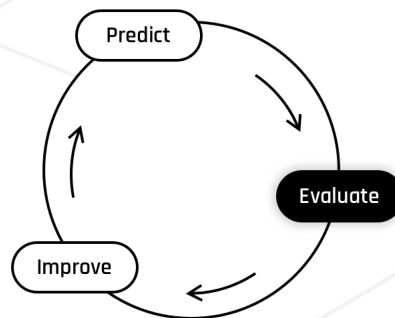


Figure VII.1: cycle evaluate

Introducing the loss function

How good is our model? It is hard to say just by looking at the plot. We can clearly observe that certain regression lines seem to fit the data better than others, but it would be convenient to find a way to measure it.

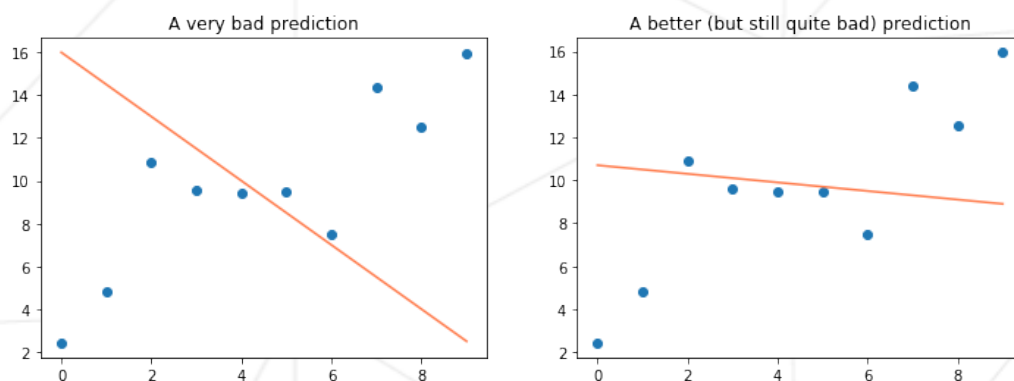


Figure VII.2: bad prediction

To evaluate our model, we are going to use a **metric** called **loss function** (sometimes called **cost function**). The loss function tells us how bad our model is, how much it *costs* us to use it, how much information we *lose* when we use it. If the model is good, we won't lose that much, if it's terrible, we have a high loss!

The metric you choose will deeply impact the evaluation (and therefore also the training) of your model.

A frequent way to evaluate the performance of a regression model is to measure the distance between each predicted value ($\hat{y}^{(i)}$) and the real value it tries to predict ($y^{(i)}$). The distances are then squared, and averaged to get one single metric, denoted J :

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)})^2$$

The smaller, the better!

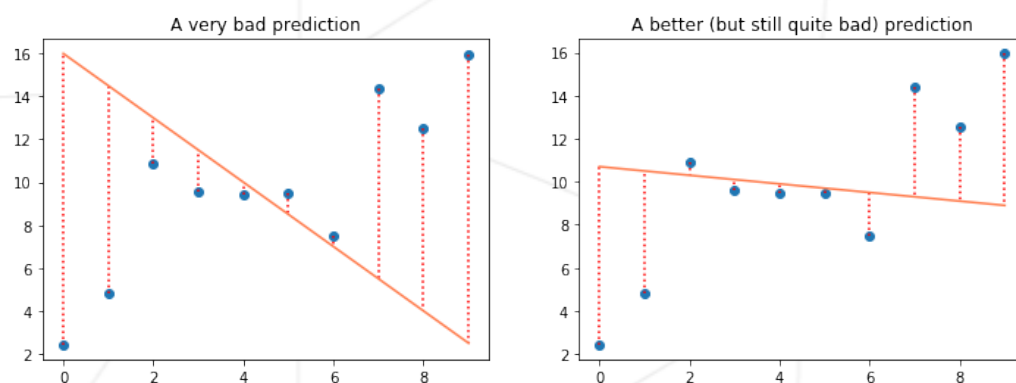



Figure VII.3: bad prediction with distance

	Exercise : 06
Loss function	
Turn-in directory : <i>ex06/</i>	
Files to turn in : loss.py	
Forbidden functions : None	

Objective

Understand and manipulate the notion of loss function in machine learning.

You must implement the following formula as a function (and another one very close to it):

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)})^2$$

Where:

- \hat{y} is a vector of dimension m , the vector of predicted values
- y is a vector of dimension $m \times 1$, the vector of expected values
- $\hat{y}^{(i)}$ is the i th component of vector \hat{y} ,
- $y^{(i)}$ is the i th component of vector y ,

Instructions

The implementation of the loss function has been split in two functions:

- `loss_elem_()`, which computes the squared distances for all examples $(\hat{y}^{(i)} - y^{(i)})^2$,
- `loss_()`, which averages the squared distances of all examples (the $J(\theta)$ above).

In the `loss.py` file create the following functions as per the instructions given below:

```
def loss_elem_(y, y_hat):
    """
    Description:
        Calculates all the elements (y_pred - y)^2 of the loss function.
    Args:
        y: has to be an numpy.array, a vector.
        y_hat: has to be an numpy.array, a vector.
    Returns:
        J_elem: numpy.array, a vector of dimension (number of the training examples,1).
        None if there is a dimension matching problem between X, Y or theta.
        None if any argument is not of the expected type.
    Raises:
        This function should not raise any Exception.
    """
    ... your code here ...

def loss_(y, y_hat):
    """
    Description:
        Calculates the value of loss function.
    Args:
        y: has to be an numpy.array, a vector.
        y_hat: has to be an numpy.array, a vector.
    Returns:
        J_value : has to be a float.
        None if there is a dimension matching problem between X, Y or theta.
        None if any argument is not of the expected type.
    Raises:
        This function should not raise any Exception.
    """
    ... your code here ...
```

Examples

```
import numpy as np

x1 = np.array([[0.], [1.], [2.], [3.], [4.]])
theta1 = np.array([[2.], [4.]])
y_hat1 = predict_(x1, theta1)
y1 = np.array([[2.], [7.], [12.], [17.], [22.]])

# Example 1:
loss_elem_(y1, y_hat1)

# Output:
array([[0.], [1.], [4.], [9.], [16.]])

# Example 2:
loss_(y1, y_hat1)

# Output:
3.0

x2 = np.array([0, 15, -9, 7, 12, 3, -21]).reshape(-1, 1)
theta2 = np.array([[0.], [1.]])
y_hat2 = predict_(x2, theta2)
y2 = np.array([2, 14, -13, 5, 12, 4, -19]).reshape(-1, 1)

# Example 3:
loss_(y2, y_hat2)

# Output:
2.142857142857143

# Example 4:
loss_(y2, y2)

# Output:
0.0
```



This loss function is very close to the one called "Mean Squared Error", which is frequently mentioned in Machine Learning resources. The difference is in the denominator as you can see in the formula of the $MSE = \frac{1}{m} \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)})^2$.

Except the division by $2m$ instead of m , these functions are rigorously identical: $J(\theta) = \frac{MSE}{2}$.


MSE is called like that because it represents the mean of the errors (i.e.: the differences between the predicted values and the true values), squared.

You might wonder why we choose to divide by two instead of simply using the MSE? (It's a good question, by the way.)

- First, it does not change the overall model evaluation: if all performance measures are divided by two, we can still compare different models and their performance ranking will remain the same.
- Second, it will be convenient when we will calculate the gradient tomorrow. Be patient, and trust us ;)

Chapter VIII

Exercise 07

	Exercise : 07
Vectorized loss function	
Turn-in directory : <i>ex07/</i>	
Files to turn in : vec_loss.py	
Forbidden functions : None	

Objective

Understand and manipulate the notion of loss function in machine learning.

You must implement the following formula as a function:

$$J(\theta) = \frac{1}{2m}(\hat{y} - y) \cdot (\hat{y} - y)$$

Where:

- \hat{y} is a vector of dimension m , the vector of predicted values,
- y is a vector of dimension m , the vector of expected values.

Instructions

In the `vec_loss.py` file, create the following function as per the instructions given below:

```
def loss_(y, y_hat):  
    """Computes the half mean squared error of two non-empty numpy.array, without any for loop.  
    The two arrays must have the same dimensions.  
    Args:  
        y: has to be an numpy.array, a vector.  
        y_hat: has to be an numpy.array, a vector.  
    Returns:  
        The half mean squared error of the two vectors as a float.  
        None if y or y_hat are empty numpy.array.  
        None if y and y_hat does not share the same dimensions.  
    Raises:  
        This function should not raise any Exceptions.  
    """  
    ... Your code ...
```

Examples

```
import numpy as np  
X = np.array([[0], [15], [-9], [7], [12], [3], [-21]])  
Y = np.array([[2], [14], [-13], [5], [12], [4], [-19]])  
  
# Example 1:  
loss_(X, Y)  
# Output:  
2.142857142857143  
  
# Example 2:  
loss_(X, X)  
# Output:  
0.0
```

Chapter IX

Exercise 08

Interlude - Fifty Shades of Linear Algebra

In the last exercise, we implemented the loss function in two subfunctions. It worked, but it's not very pretty. What if we could do it all in one step, with linear algebra?


As we did with the hypothesis, we can use a vectorized equation to improve the calculations of the loss function.

So now let's look at how squaring and averaging can be performed (more or less) in a single matrix multiplication!

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)})^2$$
$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m [(\hat{y}^{(i)} - y^{(i)})(\hat{y}^{(i)} - y^{(i)})]$$

Now, if we apply the definition of the dot product:

$$J(\theta) = \frac{1}{2m} (\hat{y} - y) \cdot (\hat{y} - y)$$

	Exercise : 08
Lets Make Nice Plots Again	
Turn-in directory : <i>ex08/</i>	
Files to turn in : plot.py	
Forbidden functions : None	

Objective

You must implement a function which plots the data, the prediction line, and the loss. You will plot the x and y coordinates of all data points as well as the prediction line generated by your theta parameters. Your function must also display the overall loss (J) in the title, and draw small lines marking the distance between each data point and its predicted value.

Instructions

In the `plot.py` file create the following function as per the instructions given below:

```
def plot_with_loss(x, y, theta):  
    """Plot the data and prediction line from three non-empty numpy.ndarray.  
    Args:  
        x: has to be a numpy.ndarray, a vector of dimension m * 1.  
        y: has to be a numpy.ndarray, a vector of dimension m * 1.  
        theta: has to be a numpy.ndarray, a vector of dimension 2 * 1.  
    Returns:  
        Nothing.  
    Raises:  
        This function should not raise any Exception.  
    """  
    ... Your code ...
```

Examples

```
import numpy as np
x = np.arange(1,6)
y = np.array([11.52434424, 10.62589482, 13.14755699, 18.60682298, 14.14329568])

# Example 1:
theta1= np.array([18,-1])
plot_with_loss(x, y, theta1)
# Output:
```

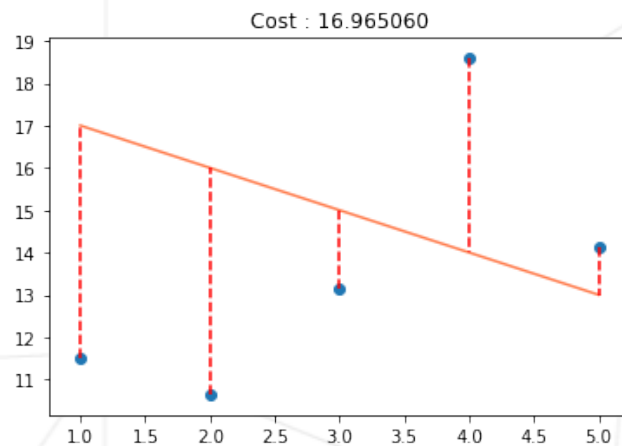


Figure IX.1: Example 1

```
# Example 2:
theta2 = np.array([14, 0])
plot_with_loss(x, y, theta2)
# Output:
```

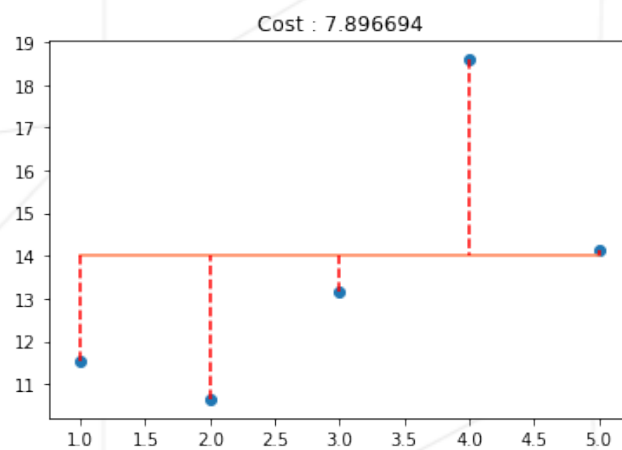


Figure IX.2: Example 2

```
# Example 3:  
theta3 = np.array([12, 0.8])  
plot_with_loss(x, y, theta3)  
# Output:
```

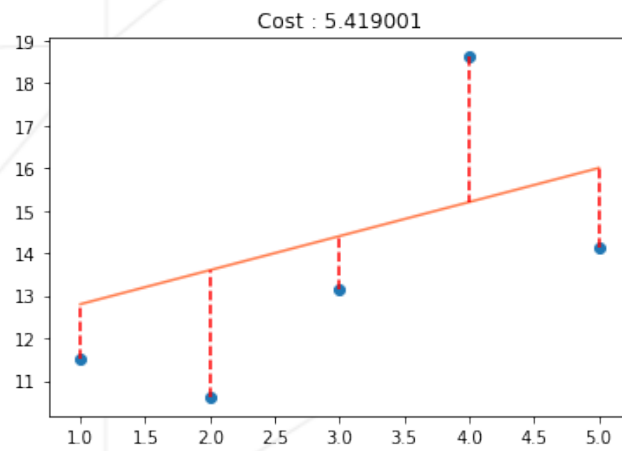



Figure IX.3: Example 3

Chapter X

Exercise 09

	Exercise : 09
Other loss functions	
Turn-in directory : <i>ex09/</i>	
Files to turn in : other_losses.py	
Forbidden functions : None	

Deepen the notion of loss function in machine learning.

You certainly had a lot of fun implementing your loss function. Remember we told you it was **one among many possible ways of measuring the loss**. Now, you will get to implement other metrics. You already know about one of them: **MSE**. There are several more which are quite common: **RMSE**, **MAE** and **R2score**.

Objective

You must implement the following formulas as functions:

$$MSE(y, \hat{y}) = \frac{1}{m} \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)})^2$$

$$RMSE(y, \hat{y}) = \sqrt{\frac{1}{m} \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)})^2}$$

$$MAE(y, \hat{y}) = \frac{1}{m} \sum_{i=1}^m |\hat{y}^{(i)} - y^{(i)}|$$

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)})^2}{\sum_{i=1}^m (y^{(i)} - \bar{y})^2}$$

Where:

- y is a vector of dimension m ,
- \hat{y} is a vector of dimension m ,
- $y^{(i)}$ is the i^{th} component of vector y ,
- $\hat{y}^{(i)}$ is the i^{th} component of \hat{y} ,
- \bar{y} is the mean of the y vector

Instructions

In the `other_losses.py` file, create the following functions: MSE, RMSE, MAE, R2score, as per the instructions given below:

```
def mse_(y, y_hat):  
    """  
    Description:  
        Calculate the MSE between the predicted output and the real output.  
    Args:  
        y: has to be a numpy.array, a vector of dimension m * 1.  
        y_hat: has to be a numpy.array, a vector of dimension m * 1.  
    Returns:  
        mse: has to be a float.  
        None if there is a matching dimension problem.  
    Raises:  
        This function should not raise any Exceptions.  
    """  
    ... your code here ...
```

```
def rmse_(y, y_hat):  
    """  
    Description:  
        Calculate the RMSE between the predicted output and the real output.  
    Args:  
        y: has to be a numpy.array, a vector of dimension m * 1.  
        y_hat: has to be a numpy.array, a vector of dimension m * 1.  
    Returns:  
        rmse: has to be a float.  
        None if there is a matching dimension problem.  
    Raises:  
        This function should not raise any Exceptions.  
    """  
    ... your code here ...
```

```
def mae_(y, y_hat):  
    """  
    Description:  
        Calculate the MAE between the predicted output and the real output.  
    Args:  
        y: has to be a numpy.array, a vector of dimension m * 1.  
        y_hat: has to be a numpy.array, a vector of dimension m * 1.  
    Returns:  
        mae: has to be a float.  
        None if there is a matching dimension problem.  
    Raises:  
        This function should not raise any Exceptions.  
    """  
    ... your code here ...
```

```
def r2score_(y, y_hat):  
    """  
    Description:  
        Calculate the R2score between the predicted output and the output.  
    Args:  
        y: has to be a numpy.array, a vector of dimension m * 1.  
        y_hat: has to be a numpy.array, a vector of dimension m * 1.  
    Returns:  
        r2score: has to be a float.  
        None if there is a matching dimension problem.  
    Raises:  
        This function should not raise any Exceptions.  
    """  
    ... your code here ...
```

You might consider implementing four more methods, similar to what you did for the loss function in exercise 07:



- `mse_elem()`,
- `rmse_elem()`,
- `mae_elem()`,
- `r2score_elem()`.

Examples

```
import numpy as np
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from math import sqrt

# Example 1:
x = np.array([0, 15, -9, 7, 12, 3, -21])
y = np.array([2, 14, -13, 5, 12, 4, -19])

# Mean squared error
## your implementation
mse_(x,y)
## Output:
4.285714285714286
## sklearn implementation
mean_squared_error(x,y)
## Output:
4.285714285714286

# Root mean squared error
## your implementation
rmse_(x,y)
## Output:
2.0701966780270626
## sklearn implementation not available: take the square root of MSE
sqrt(mean_squared_error(x,y))
## Output:
2.0701966780270626

# Mean absolute error
## your implementation
mae_(x,y)
# Output:
1.7142857142857142
## sklearn implementation
mean_absolute_error(x,y)
# Output:
1.7142857142857142

# R2-score
## your implementation
r2score_(x,y)
## Output:
0.9681721733858745
## sklearn implementation
r2_score(x,y)
## Output:
0.9681721733858745
```

Chapter XI

Conclusion - What you have learned

The excercises serie is finished, well done! Based on all the knowledges tackled today, you should be able to discuss and answer the following questions:

1. Why do we concatenate a column of ones to the left of the x vector when we use the linear algebra trick?
2. Why does the loss function square the distances between the data points and their predicted values?
3. What does the loss function's output represent?
4. Toward which value do we want the loss function to tend? What would that mean?
5. Do you understand why are matrix multiplications are not commutative?

These questions constitute an opportunity as peer community to explain for those you have not encountered these questions or to exchange and developped your comprehension with peers who have think about those ones.

Contact

You can contact 42AI association by email: contact@42ai.fr

You can join the association on [42AI slack](#) and/or posutale to [one of the association teams](#).

Acknowledgements

The modules Python & ML is the result of a collective work, we would like to thanks:

- Maxime Choulika (cmaxime),
- Pierre Peigné (ppeigne),
- Matthieu David (mdavid),
- Quentin Feuillade-Montixi (qfeuilla, quentin@42ai.fr)

who supervised the creation, the enhancement and this present transcription.

- Louis Develle (ldevelle, louis@42ai.fr)
- Owen Roberts (oroberts)
- Augustin Lopez (aulopez)
- Luc Lenotre (llenotre)
- Amric Trudel (amric@42ai.fr)
- Benjamin Carlier (bcarlier@student.42.fr)
- Pablo Clement (pclement@student.42.fr)

for your investment for the creation and development of these modules.

- Richard Blanc (riblanc@student.42.fr)
- Solveig Gaydon Ohl (sgaydon-@student.42.fr)
- Quentin Feuillade Montixi (qfeuilla@student.42.fr)

who betatest the first version of the modules of Machine Learning.