# Bootcamp
# Machine Learning


## Day01

## Stepping Into Machine Learning

# Day00 - Stepping Into Machine Learning

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 of the Day

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

## Useful Ressources

We strongly advise you 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 tomorow ;)*

**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)

## General rules

- The Python version to use is 3.7, you can check with the following command: `python -V`
- The norm: during this bootcamp you will follow the Pep8 standards

- 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 42AI's Slack workspace.
- If you find any issues or mistakes in this document, please create an issue on our dedicated Github repository.

## Helper

Ensure that you have the right Python version.

```
> which python
/goinfre/miniconda/bin/python
> python -V
Python 3.7.*
> which pip
/goinfre/miniconda/bin/pip
```

# Exercise 00 - The Vector.

| | |
|---|---|
| Turn-in directory : | ex00 |
| Files to turn in : | vector.py, test.py |
| Forbidden functions : | None |
| Forbidden libraries : | Numpy |
| Remarks : | n/a |

**AI Classics:**

*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.*

You will provide a testing file to prove that your class works as expected.
You will have to create a helpful class, with more options and providing enhanced ease of use for the user.

In this exercise, you have to create a `Vector` class. The goal is to create vectors and be able to perform mathematical operations with them.

```
>> v1 = Vector([0.0, 1.0, 2.0, 3.0])
>> v2 = v1 * 5
>> print(v2)
(Vector [0.0, 5.0, 10.0, 15.0])
```

It has 2 attributes:

- `values` : list of floats
- `size` : size of the vector -> `Vector([0.0, 1.0, 2.0, 3.0]).size == 4`

You should be able to initialize the object with:

- a list of floats: `Vector([0.0, 1.0, 2.0, 3.0])`
- a size: `Vector(3)` -> the vector will have `values = [0.0, 1.0, 2.0]`
- a range: `Vector((10,15))` -> the vector will have `values = [10.0, 11.0, 12.0, 13.0, 14.0]`

You will implement all the following built-in functions (called 'magic methods') for your `Vector` class:

```
    __add__
    __radd__
    # add : scalars and vectors, can have errors with vectors.
    __sub__
    __rsub__
    # sub : scalars and vectors, can have errors with vectors.
    __truediv__
    __rtruediv__
    # div : only scalars.
    __mul__
    __rmul__
    # mul : scalars and vectors, can have errors with vectors,
    # return a scalar if we perform Vector * Vector (dot product)
    __str__
    __repr__
```

## Authorized vector operations are:

- Addition between two vectors of same dimension (m * 1)

$$x + y = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} + \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} = \begin{bmatrix} x_1 + y_1 \\ \vdots \\ x_m + y_m \end{bmatrix}$$

- Subtraction between two vectors of same dimension (m * 1)

$$x - y = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} - \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} = \begin{bmatrix} x_1 - y_1 \\ \vdots \\ x_m - y_m \end{bmatrix}$$

- Multiplication and division between one vector (m * 1) and one scalar (1 * 1)

$$x \cdot a = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \cdot a = \begin{bmatrix} x_1 \cdot a \\ \vdots \\ x_m \cdot a \end{bmatrix}$$

- Dot product between two vectors of same dimension (m * 1)

$$x \cdot y = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} = \sum_{i=1}^{m} x_i \cdot y_i = x_1 \cdot y_1 + \cdots + x_m \cdot y_m$$

Don't forget to handle all types of error properly!

# Exercise 01 - The Matrix.

| | |
|---|---|
| Turn-in directory : | ex01 |
| Files to turn in : | matrix.py, test.py |
| Forbidden functions : | None |
| Forbidden libraries : | Numpy |
| Remarks : | n/a |

**AI Classics:**

*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.*

You will provide a testing file to prove that your class works as expected.
You will have to create a helpful class, with more options and providing enhanced ease of use for the user.

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

```
>> 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]])
>> print(m1 * m2)
(Matrix [[28., 34.], [56., 68.]])
```

It has 2 attributes:

- `data` : list of lists → the elements stored in the matrix
- `shape` : by shape we mean the dimensions of the matrix as a tuple (rows, columns) → `Matrix([[0.0, 1.0], [2.0, 3.0], [4.0, 5.0]]).shape == (3, 2)`

You should be able to initialize the object with:

- the elements of the matrix as a list of lists: `Matrix([[0.0, 1.0, 2.0, 3.0], [4.0, 5.0, 6.0, 7.0]])` → the dimensions of this matrix are then (2, 4)
- a shape `Matrix((3, 3))` → the matrix will be filled by default with zeroes
- the expected elements and shape `Matrix([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]], (3, 3))`

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

```
    __add__
    __radd__
    # add : vectors and matrices, can have errors with vectors and matrices.
    __sub__
    __rsub__
    # sub : vectors and matrices, can have errors with vectors and matrices.
    __truediv__
    __rtruediv__
    # div : only scalars.
    __mul__
    __rmul__
    # mul : scalars, vectors and matrices , can have errors with vectors and matrices,
    # return a Vector if we perform Matrix * Vector (dot product)
```

```
__str__
__repr__
```

## Matrix - vector authorized operations are:

- Multiplication between a (m * n) matrix and a (n * 1) vector

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

In other words:

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

## Matrix - matrix authorized operations are:

- Addition between two matrices of same dimension (m * 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 * 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 * n) and one scalar (1 * 1)

$$Xa = \begin{bmatrix} x_1^{(1)} & \dots & x_n^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(m)} & \dots & x_n^{(m)} \end{bmatrix} \cdot a = \begin{bmatrix} x_1^{(1)}a & \dots & x_n^{(1)}a \\ \vdots & \ddots & \vdots \\ x_1^{(m)}a & \dots & x_n^{(m)}a \end{bmatrix}$$

- Mutiplication between two matrices of compatible dimension: (m * n) and (n * 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:

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

Don't forget to handle all kind of errors properly!

# Exercise 02 - TinyStatistician

| | |
|---|---|
| Turn-in directory : | ex02 |
| Files to turn in : | TinyStatistician.py |
| Forbidden function : | any function that calculates mean, median, quartiles, variance or standard deviation at your place |
| Forbidden library : | Numpy |
| Remarks : | n/a |

**AI Classics:**

*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.*

Create a class named `TinyStatistician` which implements the following methods.
All methods take in an array and return a new modified one.
We are assuming that all inputs are correct, i.e. you don't have to protect your functions against input errors.

- `mean(x)` : computes the mean of a given non-empty array `x`, using a for-loop and returns the mean as a float, otherwise None if `x` is an empty array. This method should not raise any Exception.

Given a vector $x$ of dimension m * 1, the mathematical formula of its mean is:

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

- `median(x)` : computes the median, also called the 50th percentile, of a given non-empty darray `x`, using a for-loop and returns the median as a float, otherwise None if `x` is an empty array. This method should not raise any Exception.
- `quartiles(x, percentile)` : computes the 1st and 3rd quartiles, also called the 25th percentile and the 75th percentile, of a given non-empty array `x`, using a for-loop and returns the quartile as a float, otherwise None if `x` is an empty array. The first parameter is the array and the second parameter is the expected percentile. This method should not raise any Exception.
- `var(x)` : computes the variance of a given non-empty array `x`, using a for-loop and returns the variance as a float, otherwise None if `x` is an empty array. This method should not raise any Exception.

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

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

- `std(x)` : computes the standard deviation of a given non-empty array `x`, using a for-loop and returns the standard deviation as a float, otherwise None if `x` is an empty array. This method should not raise any Exception.

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

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

## Examples

```
>>> from TinyStatistician import TinyStatistician
>>> tstat = TinyStatistician()
>>> a = [1, 42, 300, 10, 59]

>>> tstat.mean(a)
```

```
82,4

>>> tstat.median(a)
42.0

>>> tstat.quartile(a, 25)
10.0

>>> tstat.quartile(a, 75)
59.0

>>> tstat.var(a)
12279.439999999999

>>> tstat.std(a)
110.81263465868862
```

# 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
'''
```

To be said 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.

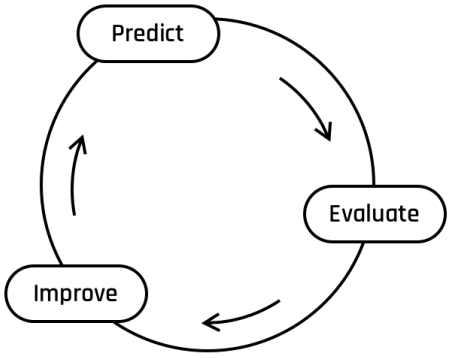


Figure 1: cycle_neutral

# Predict

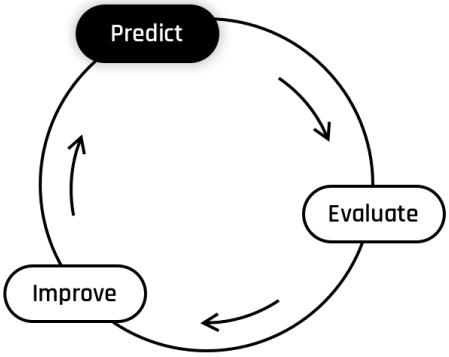


Figure 2: cycle_predict

## 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.*

$$hypothesis(data) = predictions$$

### 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 '^' 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 $\theta_0$ and $\theta_1$ 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:

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

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 experessed 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 $\theta$ notation?

You might have two questions at the moment:

- **WTF is that weird symbol?**
  This strange symbol, $\theta$, 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: $\hat{y} = ax_1 + bx_2 + cx_3 + ... + yx_{25} + z$
  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 $\theta$, you just have to increment the number to name the parameter:
  $\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_{2468} x_{2468}$ ... Easy right?

### Another common notation:

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

Because $\hat{y}$ is calculated with our linear hypothesis using $\theta$ 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 theta"*.

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 03 - Simple Prediction

| | |
|---|---|
| Turn-in directory : | ex03 |
| Files to turn in : | prediction.py |
| Forbidden functions : | None |
| Remarks : | n/a |

## Objective:

You must implement the following formula as a function:

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

Where:

- $x$ is a vector of dimension m * 1, the vector of examples (without the $y$ values)
- $\hat{y}$ is a vector of dimension m * 1, the vector of predicted values
- $\theta$ 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 an numpy.ndarray, a vector of dimension m * 1.
      theta: has to be an 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.
    """
```

## Examples:

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

#Example 1:
theta1 = np.array([5, 0])
simple_predict(x, theta1)
# Ouput:
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.])
```

# 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 $\theta$ 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 04 - Add Intercept

| | |
|---|---|
| Turn-in directory : | ex04 |
| Files to turn in : | tools.py |
| Forbidden functions : | None |
| Remarks : | n/a |

## Objective:

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.ndarray x.
    Args:
      x: has to be an numpy.ndarray, a vector of dimension m * 1.
    Returns:
      X as a numpy.ndarray, a vector of dimension m * 2.
      None if x is not a numpy.ndarray.
      None if x is a empty numpy.ndarray.
    Raises:
      This function should not raise any Exception.
    """
```

## 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.]])
```

# Exercise 05 - Prediction

| | |
|---|---|
| Turn-in directory : | ex05 |
| Files to turn in : | prediction.py |
| Forbidden functions : | None |
| Remarks : | n/a |

## Objective:

You must implement the following formula as a function:

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

Where:

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

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

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

**Be careful:**

- the $x$ you will get as an input is an $m * 1$ vector
- $\theta$ is a $2 * 1$ vector.

You have to transform $x$ into $X'$ to fit the dimension of $\theta$ !

## 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.ndarray.
    Args:
      x: has to be an numpy.ndarray, a vector of dimension m * 1.
      theta: has to be an 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 Exceptions.
    """
```

## Examples:

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

#Example 1:
theta1 = np.array([5, 0])
predict_(x, theta1)
# Ouput:
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.])
```

# Exercise 06 - Let's Make Nice Plots

| | |
|---|---|
| Turn-in directory : | ex06 |
| Files to turn in : | plot.py |
| Forbidden functions : | None |
| Remarks : | Matplotlib is your friend |

**It is plot time!**

For you rinformation, 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 ($h_\theta$).

## 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.ndarray.
    Args:
      x: has to be an numpy.ndarray, a vector of dimension m * 1.
      y: has to be an numpy.ndarray, a vector of dimension m * 1.
      theta: has to be an numpy.ndarray, a vector of dimension 2 * 1.
    Returns:
        Nothing.
    Raises:
      This function should not raise any Exceptions.
    """
```

## 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:
```

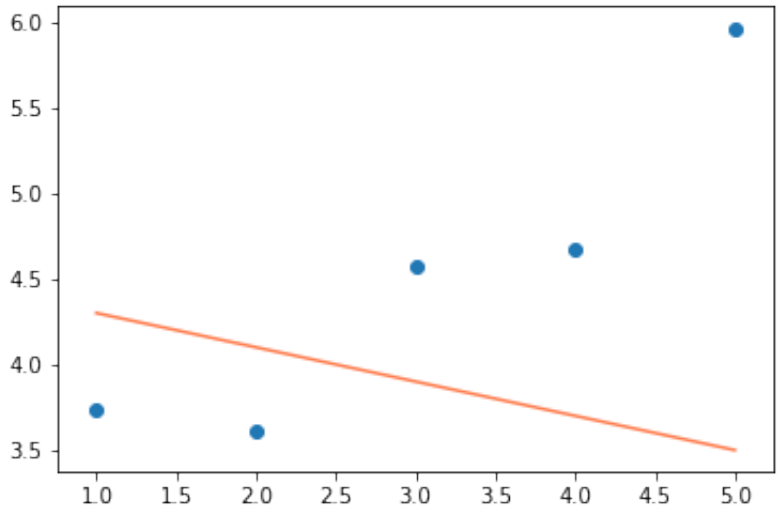


Figure 3: plot1

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

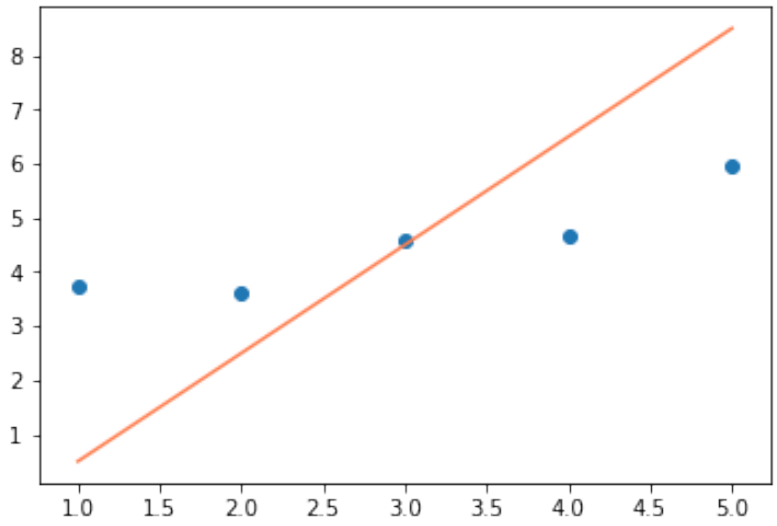


Figure 4: plot2

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

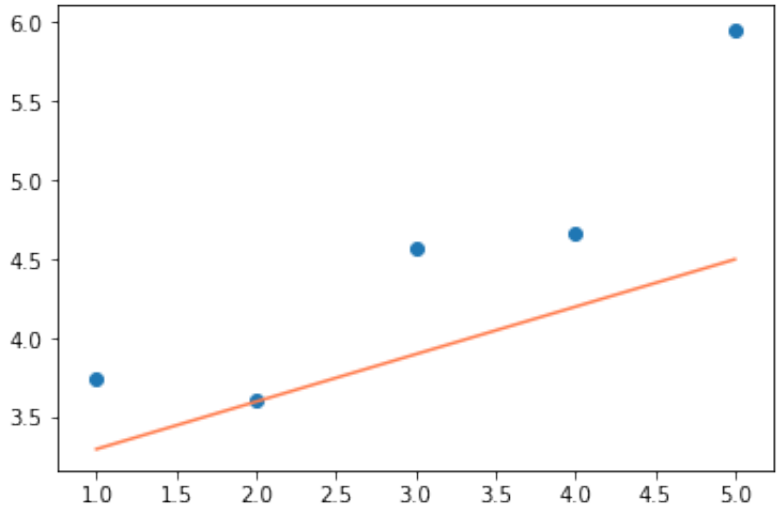


Figure 5: plot3

# Interlude - Evaluate

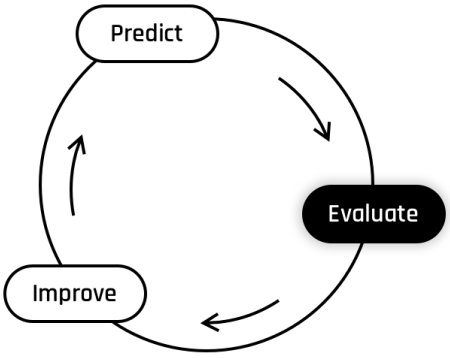


Figure 6: cycle_evaluate

## Introducing the cost 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.

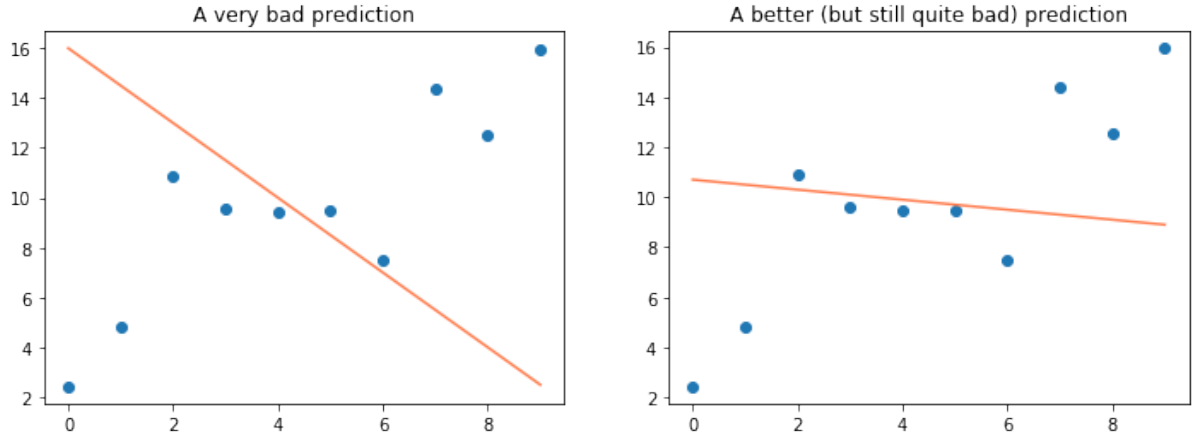


Figure 7: plot_bad_prediction

To evaluate our model, we are going to use a **metric** called **cost function** (sometimes called **loss function**). The cost 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 it will cost us a lot!

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!

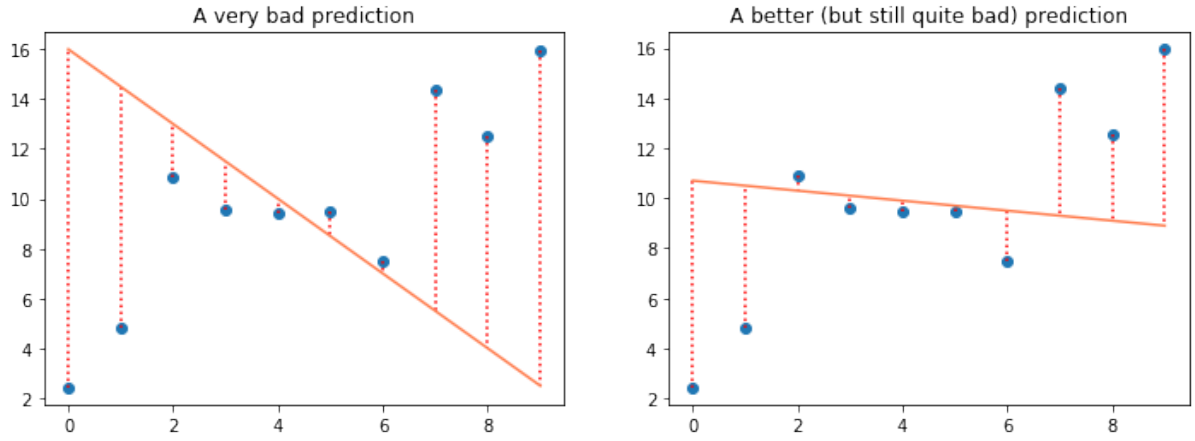


Figure 8: plot_bad_pred_with_distance

# Exercise 07 - Cost Function

| | |
|---|---|
| Turn-in directory : | ex07 |
| Files to turn in : | cost.py |
| Forbidden functions : | None |
| Remarks : | n/a |

## Objective:

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

## Instructions:

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

- *cost_elem_( )*, which computes the squared distances for all examples
- *cost_( )*, which averages the distances across all examples

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

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

def cost_(y, y_hat):
    """
    Description:
        Calculates the value of cost function.
    Args:
      y: has to be an numpy.ndarray, a vector.
      y_hat: has to be an numpy.ndarray, a vector.
    Returns:
        J_value : has to be a float.
```

```
        None if there is a dimension matching problem between X, Y or theta.
    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:
cost_elem_(y1, y_hat1)

# Output:
array([[0.], [0.1], [0.4], [0.9], [1.6]])

# Example 2:
cost_(y1, y_hat1)

# Output:
3.0

x2 = np.array([[0.2, 2., 20.], [0.4, 4., 40.], [0.6, 6., 60.], [0.8, 8., 80.]])
theta2 = np.array([[0.05], [1.], [1.], [1.]])
y_hat2 = predict_(x2, theta2)
y2 = np.array([[19.], [42.], [67.], [93.]])

# Example 3:
cost_elem_(y2, y_hat2)

# Output:
array([[1.3203125], [0.7503125], [0.0153125], [2.1528125]])

# Example 4:
cost_(y2, y_hat2)

# Output:
4.238750000000004

x3 = np.array([0, 15, -9, 7, 12, 3, -21])
theta3 = np.array([[0.], [1.]])
y_hat3 = predict_(x3, theta3)
y3 = np.array([2, 14, -13, 5, 12, 4, -19])

# Example 5:
cost_(y3, y_hat3)

# Output:
4.285714285714286

# Example 6:
cost_(y3, y3)
```

```
# Output:
0.0
```

## More Information:

This cost 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 rigourously 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 tomorow. Be patient, and trust us ;)

# Interlude - Fifty Shades of Linear Algebra

In the last exercise, we implemented the cost 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 cost 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 - Vectorized Cost Function

| | |
|---|---|
| Turn-in directory : | ex08 |
| Files to turn in : | vec_cost.py |
| Forbidden functions : | None |
| Remarks : | n/a |

## Objective:

You must implement the following formula as a function:

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

Where:

- $\hat{y}$ is a vector of dimension $m * 1$, the vector of predicted values
- $y$ is a vector of dimension $m * 1$, the vector of expected values

## Instructions:

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

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

## 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:
cost_(X, Y)
# Output:
4.285714285714286

# Example 2:
cost_(X, X)
# Output:
0.0
```

# Exercise 09 - Lets Make Nice Plots Again

| | |
|---|---|
| Turn-in directory : | ex09 |
| Files to turn in : | plot.py |
| Forbidden functions : | None |
| Remarks : | Matplotlib is your friend |

**It's plot time again!**

## Objective:

You must implement a function which plots the data, the prediction line, and the cost.
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 cost ($J$) 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_cost(x, y, theta):
    """Plot the data and prediction line from three non-empty numpy.ndarray.
    Args:
      x: has to be an numpy.ndarray, a vector of dimension m * 1.
      y: has to be an numpy.ndarray, a vector of dimension m * 1.
      theta: has to be an numpy.ndarray, a vector of dimension 2 * 1.
    Returns:
        Nothing.
    Raises:
      This function should not raise any Exception.
    """
```

## 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_cost(x, y, theta1)
# Output:
```

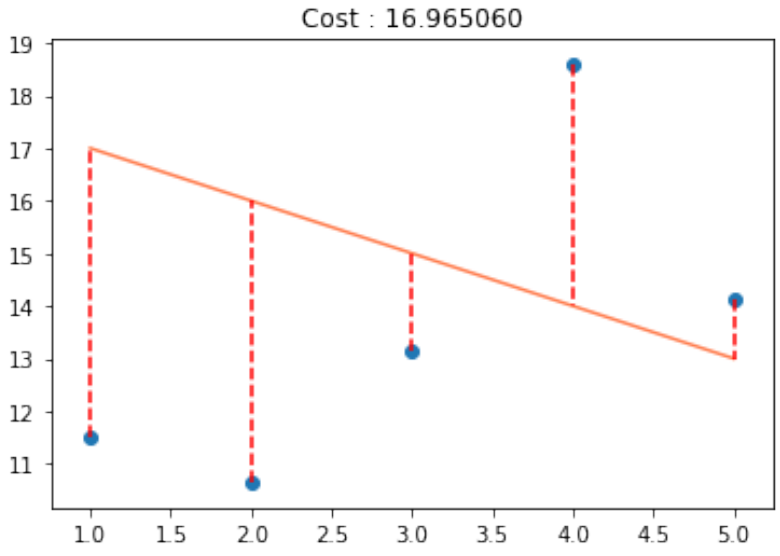


Figure 9: plot_cost1

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

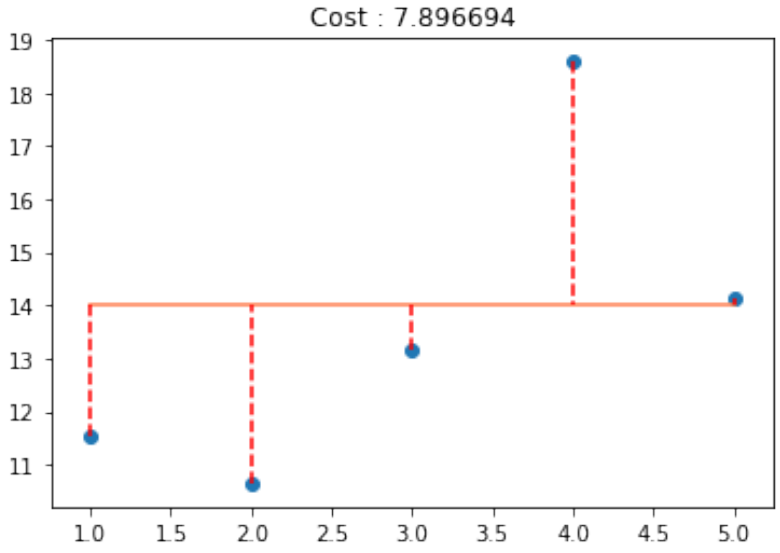


Figure 10: plot_cost2

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

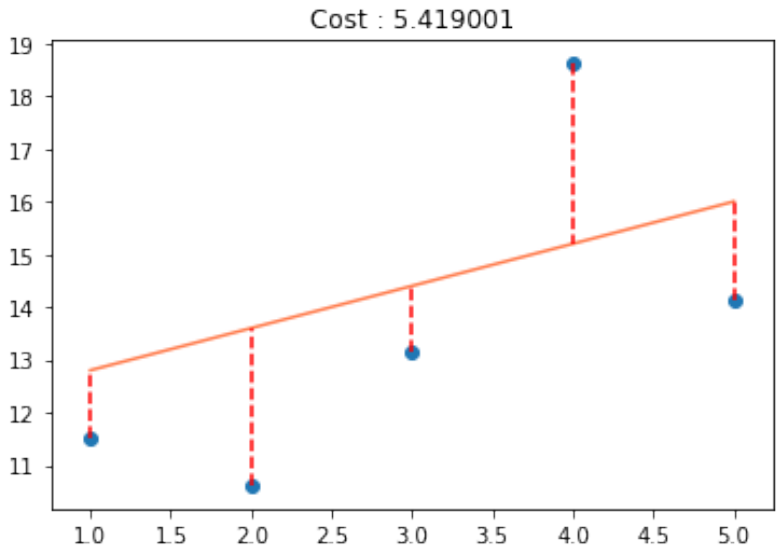


Figure 11: plot_cost3

# Exercise 10 - Question time!

## Are you able to clearly and simply explain:

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 cost function square the distances between the data points and their predicted values?

3 - What does the cost function's output represent?

4 - Toward which value do we want the cost function to tend? What would that mean?

5 - Do you understand why are matrix multiplications are not commutative?

# Exercise 11 - Other Cost Functions

| | |
|---|---|
| Turn-in directory : | ex11 |
| Files to turn in : | other_costs.py |
| Forbidden functions : | None |
| Remarks : | n/a |

You certainly had a lot of fun implementing your cost function. Remember we told you it was **one among many possible ways of measuring the cost**. 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:

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

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

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

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

Where:

- $y$ is a vector of dimension $m * 1$,
- $\hat{y}$ is a vector of dimension $m * 1$,
- $y^{(i)}$ 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_costs.py` file, create the following functions: 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.ndarray, a vector of dimension m * 1.
        y_hat: has to be a numpy.ndarray, 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.ndarray, a vector of dimension m * 1.
        y_hat: has to be a numpy.ndarray, 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.ndarray, a vector of dimension m * 1.
        y_hat: has to be a numpy.ndarray, 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.ndarray, a vector of dimension m * 1.
        y_hat: has to be a numpy.ndarray, 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 ...
```

## Remarks:

You might consider implementing four more methods, similar to what you did for the cost 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
```