

# Bootcamp Machine Learning



## Day02 Multivariate Linear Regression

# Day02 - Multivariate Linear Regression

Building on what you did on the previous days you will extend the linear regression to handle more than one features. Then you will see how to build polynomial models and how to detect overfitting.

## Notions of the Day

Multivariate linear hypothesis, multivariate linear gradient descent, polynomial models. Training and test sets, overfitting.

## 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 2:

#### Multivariate Linear Regression:

- Multiple Features (Video + Reading)
- Gradient Descent for Multiple Variables (Video + Reading)
- Gradient Descent in Practice I- Feature Scaling (Video + Reading)
- Gradient Descent in Practice II- Learning Rate (Video + Reading)
- Features and Polynomial Regression (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 - Linear Regression with Class

Exercise 01 - Ai Key Notions

Interlude - To the Multivariate Universe and Beyond!

Exercise 02 - Multivariate Hypothesis - Iterative Version

Interlude - Even More Linear Algebra Tricks!

Exercise 03 - Multivariate hypothesis - vectorized version

Interlude - Evaluate

Exercise 04 - Vectorized Cost Function

Interlude - Improve with the Gradient

Exercise 05 - Multivariate Linear Gradient

Interlude - Gradient Descent

Exercise 06 - Multivariate Gradient Descent

Exercise 07 - Multivariate Linear Regression with Class

Exercise 08 - Practicing Multivariate Linear Regression

Exercise 9 - Question Time!

Interlude - Introducing Polynomial Models

Exercise 10 - Polynomial models

Exercise 11 - Let's Train Polynomial Models!

Interlude - Plotting Curves With Matplotlib

Exercise 12 - Let's PLOT some Polynomial Models!

Interlude - Lost in Overfitting

Exercise 13 - DataSplitter

Exercise 14 - Machine Learning for Grown-ups: Training and Test Sets

Exercise 15 - Question Time!

# Exercise 00 - Linear Regression with Class

---

Turn-in directory :	ex00
Files to turn in :	mylinearregression.py
Authorized modules :	numpy
Forbidden modules :	sklearn

---

## AI Classics:

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

## Objectives:

- Write a class that contains all methods necessary to perform Linear Regression.

## Instructions:

In this exercise, you will not learn anything new but don't worry, it's for your own good!

You are expected to write your own `MyLinearRegression` class which looks similar to the `sklearn.linear_model.LinearRegression` class:

```
class MyLinearRegression():
    """
    Description:
        My personnal linear regression class to fit like a boss.
    """
    def __init__(self, theta):
        """
        Description:
            generator of the class, initialize self.
        Args:
            theta: has to be a list or a numpy array, it is a vector of dimension (number of
        → features + 1, 1).
        Raises:
            This method should noot raise any Exception.
        """
        ... your code here ...
        ... other methods ...
```

You will add the following methods:

- `predict_(self, x)`
- `cost_elem_(self, x, y)`
- `cost_(self, x, y)`
- `fit_(self, x, y, alpha, n_cycle)`

You have already implemented these functions, you just need a few adjustments so that they all work well within your `MyLinearRegression` class.

## Examples:

```
import numpy as np
from mylinearregression import MyLinearRegression as MyLR
x = np.array([[12.4956442], [21.5007972], [31.5527382], [48.9145838], [57.5088733]])
y = np.array([[37.4013816], [36.1473236], [45.7655287], [46.6793434], [59.5585554]])

lr1 = MyLR([2, 0.7])

# Example 0.0:
lr1.predict(x)
# Output:
array([[10.74695094],
       [17.05055804],
       [24.08691674],
       [36.24020866],
       [42.25621131]])

# Example 0.1:
lr1.cost_elem_(lr1.predict(x),y)
# Output:
array([[77.72116511],
       [49.33699664],
       [72.38621816],
       [37.29223426],
       [78.28360514]])

# Example 0.2:
lr1.cost_(lr1.predict(x),y)
# Output:
315.0202193084312

# Example 1.0:
lr2 = MyLR([0, 0])
lr2.fit(x, y, alpha=5e-8, n_cycle = 1500000)
lr2.theta
# Output:
array([[1.40709365],
       [1.1150909 ]])

# Example 1.1:
lr2.predict(x)
# Output:
array([[15.3408728 ],
       [25.38243697],
       [36.59126492],
       [55.95130097],
       [65.53471499]])

# Example 1.2:
lr2.cost_elem_(lr1.predict(x),y)
# Output:
array([[35.6749755 ],
       [ 4.14286023],
       [ 1.26440585],
       [29.30443042],
       [22.27765992]])

# Example 1.3:
```

```
lr2.cost_(lr1.predict(x),y)
# Output:
92.66433192085971
```

# Exercise 01 - AI Key Notions:

*These questions highlight key notions from the previous days. Making sure you can formulate a clear answer to each of them is necessary before you keep going. Discuss them with a fellow student if you can.*

## Are you able to clearly and simply explain:

- 1 - What is a *hypothesis* and what is its goal? (It's a second chance for you to say something intelligible, no need to thank us!)
- 2 - What is a *cost function* and what does it represent?
- 3 - What is *linear gradient descent* and what does it do? (hint: you have to talk about  $J$ , its gradient, and  $\theta$ )
- 4 - What happens if you choose a *learning rate* that is too large?
- 5 - What happens if you choose a very small *learning rate*, but still a sufficient *number of cycles*?
- 6 - Can you explain *MSE* and what it measures?

# Interlude - To the Multivariate Universe and Beyond!

Until now we've used a very simple hypothesis:  $h_{\theta}(x) = \theta_0 + \theta_1 x$

With this very simple hypothesis we found a way to evaluate and improve our predictions.

That's all very neat, but we live in a world full of complex phenomena that a model using this hypothesis would fail miserably at predicting. If we take weather forecasting for example, how easy do you think it would be to predict tomorrow's temperature with just one variable (say, the current atmospheric pressure)? A model based on just one variable is too simple to account for the complexity of this phenomenon.

Now what if, on top of the atmospheric pressure, we could take into account the current temperature, humidity, wind, sunlight, and any useful information we can get our hands on?

We'd need a model where more than one variable (or even thousands of variables) are involved. That's what we call a **multivariate model**. And that's today's topic!

## Predict

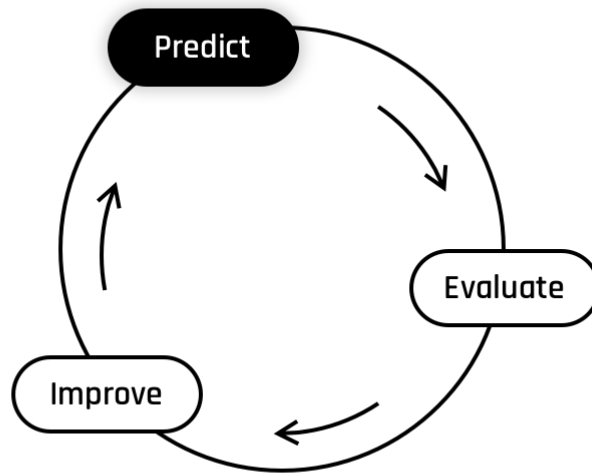


Figure 1: The Learning Cycle - Predict

## Representing the examples as an $m * n$ matrix

First we need to reconsider how we represent the training examples. Now that we want to characterize each training example with not just one, but many variables, we need more than a vector. We need a **matrix**!

So instead of an  $x$  vector of dimension  $m * 1$ , we now have a matrix of dimension  $m * n$ , where  $n$  is the number of **features** (or variables) that characterize each training example. We call it the **design matrix**, denoted by a



capital  $X$ .

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

Where:

- $x^{(i)}$  is the feature vector of the  $i^{th}$  training example, ( $i^{th}$  row of the  $X$  matrix)
- $x_j$  is the  $j^{th}$  column of the  $X$  matrix
- $x_j^{(i)}$  is the  $j^{th}$  feature of the  $i^{th}$  training example (at the intersection of the  $i^{th}$  row and the  $j^{th}$  column of the  $X$  matrix). It's a real number.

## The multivariate hypothesis

Then, we must update our hypothesis to take more than one feature into account.

$$\hat{y}^{(i)} = \theta_0 + \theta_1 x_1^{(i)} + \dots + \theta_n x_n^{(i)} \quad \text{for } i = 1, \dots, m$$

Where:

- $\hat{y}^{(i)}$  is the model's prediction for the  $i^{th}$  example,
- $x_1^{(i)}$  is the first feature of the  $i^{th}$  example,
- $x_n^{(i)}$  is the  $n^{th}$  feature of the  $i^{th}$  example
- $\theta$  is a vector of dimension  $(n + 1) * 1$ , the parameter vector.

You will notice that we end up with two indices:  $i$  and  $j$ . They should not be confused:

- $i$  refers to one of the  $m$  examples in the dataset (line number in the  $X$  matrix)
- $j$  refers to one of the  $n$  features that describe each example (column number in the  $X$  matrix)

# Exercise 02 - Multivariate Hypothesis - Iterative Version

---

Turn-in directory :	ex02
Files to turn in :	prediction.py
Forbidden functions :	None
Remarks :	n/a

---

## Objectives:

You must implement the following formula as a function:

$$\hat{y}^{(i)} = \theta_0 + \theta_1 x_1^{(i)} + \dots + \theta_n x_n^{(i)} \quad \text{for } i = 1, \dots, m$$

Where:

- $\hat{y}$  is a vector of dimension  $m * 1$ : the vector of predicted values
- $\hat{y}^{(i)}$  is the  $i^{th}$  component of the  $\hat{y}$  vector: the predicted value for the  $i^{th}$  example
- $\theta$  is a vector of dimension  $(n + 1) * 1$ : the parameter vector
- $\theta_j$  is the  $j^{th}$  component of the parameter vector
- $X$  is a matrix of dimension  $m * n$ : the design matrix
- $x^{(i)}$  is the  $i^{th}$  row of the  $X$  matrix: the feature vector of the  $i^{th}$  example
- $x_j$  is the  $j^{th}$  column of the  $X$  matrix
- $x_j^{(i)}$  is the element at the intersection of the  $i^{th}$  row and the  $j^{th}$  column of the  $X$  matrix: the  $j^{th}$  feature of the  $i^{th}$  example

## Instructions:

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

```
def simple_predict(x, theta):  
    """Computes the prediction vector y_hat from two non-empty numpy.ndarray.  
    Args:  
        x: has to be a numpy.ndarray, a matrix of dimension m * n.  
        theta: has to be a numpy.ndarray, a vector of dimension (n + 1) * 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(1,13).reshape((4,))
```

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

```
#Example 2:
theta2 = np.array([0, 1, 0, 0])
simple_predict(x, theta2)
# Output:
array([ 1.,  4.,  7., 10.])
# Do you understand why y_hat == x[:,0] here?
```

```
#Example 3:
theta3 = np.array([-1.5, 0.6, 2.3, 1.98])
simple_predict(X, theta3)
# Output:
array([ 9.64, 24.28, 38.92, 53.56])
```

```
#Example 4:
theta4 = np.array([-3, 1, 2, 3.5])
simple_predict(x, theta4)
# Output:
array([12.5, 32. , 51.5, 71. ])
```

# Interlude - Even More Linear Algebra Tricks!

As you already did before with the univariate hypothesis, the multivariate hypothesis can be vectorized as well.

If you add a column of 1's as the first column of the  $X$  matrix, you get what we'll call the  $X'$  matrix. Then, you can calculate  $\hat{y}$  by multiplying  $X'$  and  $\theta$ .

$$X' \cdot \theta = \begin{bmatrix} 1 & x_1^{(1)} & \dots & x_n^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & \dots & x_n^{(m)} \end{bmatrix} \cdot \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix} = \begin{bmatrix} \theta_0 + \theta_1 x_1^{(1)} + \dots + \theta_n x_n^{(1)} \\ \vdots \\ \theta_0 + \theta_1 x_1^{(m)} + \dots + \theta_n x_n^{(m)} \end{bmatrix} = \begin{bmatrix} \hat{y}^{(1)} \\ \vdots \\ \hat{y}^{(m)} \end{bmatrix} = \hat{y}$$

Another way of understanding this algebra trick is to pretend that each training example has an artificial  $x_0$  feature that is always equal to 1. This simplifies the equations because now, each  $x_j$  feature has its corresponding  $\theta_j$  parameter in the multiplication.

$$\theta_0 x_0^{(i)} + \theta_1 x_1^{(i)} + \dots + \theta_n x_n^{(i)} = \theta \cdot x'^{(i)}$$

# Exercise 03 - Multivariate hypothesis - vectorized version

---

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

---

## Objectives:

You must implement the following formula as a function:

$$\hat{y} = X' \cdot \theta = \begin{bmatrix} 1 & x_1^{(1)} & \dots & x_n^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & \dots & x_n^{(m)} \end{bmatrix} \cdot \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix} = \begin{bmatrix} \theta_0 + \theta_1 x_1^{(1)} + \dots + \theta_n x_n^{(1)} \\ \vdots \\ \theta_0 + \theta_1 x_1^{(m)} + \dots + \theta_n x_n^{(m)} \end{bmatrix}$$

Where:

- $\hat{y}$  is a vector of dimension  $m * 1$ : the vector of predicted values
- $X$  is a matrix of dimension  $m * n$ : the design matrix
- $X'$  is a matrix of dimension  $m * (n + 1)$ : the design matrix onto which a column of 1's was added as a first column
- $\theta$  is a vector of dimension  $(n + 1) * 1$ : the parameter vector
- $x^{(i)}$  is the  $i^{th}$  row of the  $X$  matrix
- $x_j$  is the  $j^{th}$  column of the  $X$  matrix
- $x_j^{(i)}$  is the intersection of the  $i^{th}$  row and the  $j^{th}$  column of the  $X$  matrix: the  $j^{th}$  feature of the  $i^{th}$  training example

Be careful:

- The `x` argument your function will receive as an input corresponds to  $X$ , the  $m * n$  matrix. Not  $X'$ .
- `theta` is an  $(n + 1) * 1$  vector.
- You have to transform `x` to fit `theta`'s dimension!

## Instructions:

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

```
def predict_(x, theta):
    """Computes the prediction vector y_hat from two non-empty numpy.ndarray.
    Args:
        x: has to be a numpy.ndarray, a vector of dimension m * n.
        theta: has to be a numpy.ndarray, a vector of dimension (n + 1) * 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(1,13).reshape((4,))  
  
#Example 1:  
theta1 = np.array([5, 0, 0, 0])  
predict_(x, theta1)  
# Output:  
array([5., 5., 5., 5.])  
# Do you understand why y_hat contains only 5's here?  
  
#Example 2:  
theta2 = np.array([0, 1, 0, 0])  
predict_(x, theta2)  
# Output:  
array([ 1.,  4.,  7., 10.])  
# Do you understand why y_hat == x[:,0] here?  
  
#Example 3:  
theta3 = np.array([-1.5, 0.6, 2.3, 1.98])  
predict_(X, theta3)  
# Output:  
array([ 9.64, 24.28, 38.92, 53.56])  
  
#Example 4:  
theta4 = np.array([-3, 1, 2, 3.5])  
predict_(x, theta4)  
# Output:  
array([12.5, 32. , 51.5, 71. ])
```

# Interlude - Evaluate

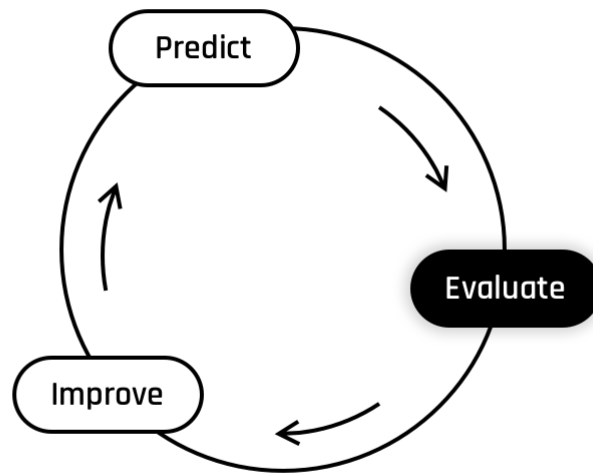


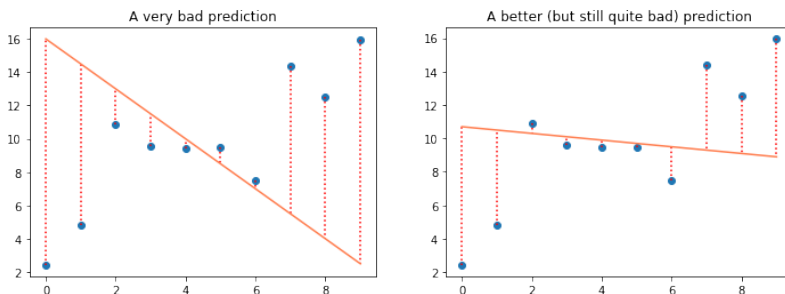
Figure 2: The Learning Cycle - Evaluate

## Back to the Cost Function

How is our model doing?

To evaluate our model, remember before we used a **metric** called the **cost function** (also known as **loss function**). The cost function is basically just a measure of how wrong the model is, in all of its predictions.

Two days ago, we defined the cost function as the average of the squared distances between each prediction and its expected value (distances represented by the dotted lines in the figure below) :



The formula was the following:

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

And its vectorized form:

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

**So, now that we moved to multivariate linear regression, what needs to change?** You may have noticed that variables such as  $x_j$  and  $\theta_j$  don't intervene in the equation. Indeed, the cost function only uses the predictions (  $\hat{y}$  ) and the expected values (  $y$  ), so the inner workings of the model don't matter to its evaluation metric.

This means we can use the exact same cost function as we did before!



# Interlude - Improve with the Gradient

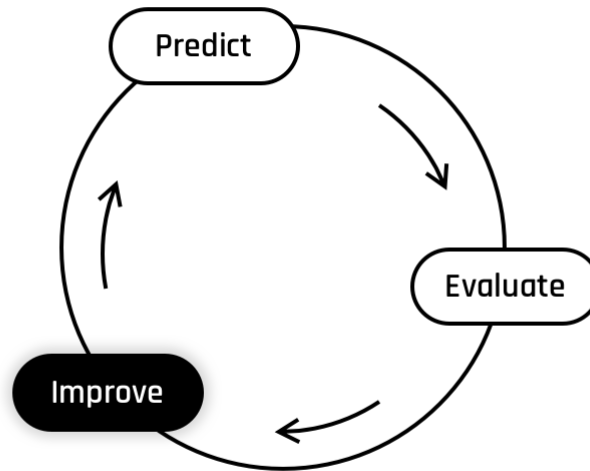


Figure 3: The Learning Cycle: Improve

## Multivariate Gradient

From our multivariate linear hypothesis we can derive our multivariate gradient. It looks a lot like the one we saw yesterday, but instead of having just two components, the gradient now has as many as there are parameters. This means that now we need to calculate  $\nabla(J)_0, \nabla(J)_1, \dots, \nabla(J)_n$

If we take the univariate equations we used yesterday and replace the formula for  $\nabla(J)_1$  by a more general  $\nabla(J)_j$ , we get the following:

$$\begin{aligned}\nabla(J)_0 &= \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) \\ \nabla(J)_j &= \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} \quad \text{for } j = 1, \dots, n\end{aligned}$$

Where:

- $\nabla(J)$  is a vector of size  $(n + 1) * 1$ , the gradient vector
- $\nabla(J)_j$  is the  $j^{th}$  component of  $\nabla(J)$ , the partial derivative of  $J$  with respect to  $\theta_j$
- $y$  is a vector of dimension  $m * 1$ , the vector of expected values
- $y^{(i)}$  is a scalar, the  $i^{th}$  component of vector  $y$
- $x^{(i)}$  is the feature vector of the  $i^{th}$  example
- $x_j^{(i)}$  is a scalar, the  $j^{th}$  feature value of the  $i^{th}$  example
- $h_{\theta}(x^{(i)})$  is a scalar, the model's estimation of  $y^{(i)}$ . (It can also be denoted  $\hat{y}^{(i)}$ )

## Vectorized Form

As usual, we can use some linear algebra magic to get a more compact (and computationally efficient) formula.

First we can use our convention that each training example has an extra  $x_0 = 1$  feature, and replace the gradient formulas above by one single equation that is valid for all  $j$  components:

$$\nabla(J)_j = \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} \quad \text{for } j = 0, \dots, n$$

And this generic equation can then be rewritten in a vectorized form:

$$\nabla(J) = \frac{1}{m} X'^T (X'\theta - y)$$

Where:

- $\nabla(J)$  is the gradient vector of size  $(n + 1) * 1$
- $X'$  is a matrix of dimension  $m * (n + 1)$ , the design matrix onto which a column of 1's was added as the first column
- $X'^T$  means the matrix has been transposed
- $\theta$  is a vector of size  $(n + 1) * 1$ , the parameter vector
- $y$  is a vector of size  $m * 1$ , the vector of expected values

The vectorized equation can output the entire gradient vector all at once, in one calculation! So if you understand the linear algebra operations, you can forget about the equations we presented at the top of the page and simply use the vectorized one.

# Exercise 05 - Multivariate Linear Gradient

---

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

---

## Objectives:

You must implement the following formula as a function:

$$\nabla(J) = \frac{1}{m} X'^T (X' \theta - y)$$

Where:

- $\nabla(J)$  is a vector of dimension  $(n + 1) * 1$ , the gradient vector
- $X$  is a matrix of dimension  $m * n$ , the design matrix
- $X'$  is a matrix of dimension  $m * (n + 1)$ , the design matrix onto which a column of 1's was added as a first column
- $\theta$  is a vector of dimension  $(n + 1) * 1$ , the parameter vector
- $y$  is a vector of dimension  $m * 1$ , the vector of expected values

## Instructions:

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

```
def gradient(x, y, theta):
    """Computes a gradient vector from three non-empty numpy.ndarray, without any for-loop.
    ↳ The three arrays must have the compatible dimensions.
    Args:
        x: has to be a numpy.ndarray, a matrix of dimension m * n.
        y: has to be a numpy.ndarray, a vector of dimension m * 1.
        theta: has to be a numpy.ndarray, a vector (n+1) * 1.
    Returns:
        The gradient as a numpy.ndarray, a vector of dimensions n * 1, containing the result of
    ↳ the formula for all j.
        None if x, y, or theta are empty numpy.ndarray.
        None if x, y and theta do not have compatible dimensions.
    Raises:
        This function should not raise any Exception.
    """
```

## Examples:

```
import numpy as np
x = np.array([
    [-6, -7, -9],
    [13, -2, 14],
```

```

        [-7, 14, -1],
        [-8, -4, 6],
        [-5, -9, 6],
        [ 1, -5, 11],
        [ 9, -11, 8]])
y = np.array([2, 14, -13, 5, 12, 4, -19])
theta1 = np.array([3,0.5,-6])

# Example :
gradient(x, y, theta1)
# Output:
array([-37.35714286, 183.14285714, -393.        ])

# Example :
theta2 = np.array([0,0,0])
gradient(x, y, theta2)
# Output:
array([ 0.85714286, 23.28571429, -26.42857143])

```

# Interlude - Gradient Descent

Now comes the fun part: *gradient descent*!

The algorithm is not that different from the one used in univariate linear regression. As you might have guessed, what will change is that the  $j$  indice needs to run from 0 to  $n$  instead of 0 to 1. So all you need is a more generic algorithm, which can be expressed in pseudocode as the following:

```
repeat until convergence      {
    compute  $\nabla(J)$ 
     $\theta_j := \theta_j - \alpha \nabla(J)_j$ 
} simultaneously update  $\theta$  for  $j = 0, 1, \dots, n$ 
```

If you started to like vectorized forms, you might have noticed that the  $\theta_j$  notation is actually redundant here, since all components of  $\theta$  need to be updated simultaneously.  $\theta$  is a vector,  $\nabla(J)$  also, they both have dimension  $(n + 1) * 1$ . So all we need to do is this:

```
repeat until convergence      {
    compute  $\nabla(J)$ 
     $\theta := \theta - \alpha \nabla(J)$ 
}
```

Where:

- $\theta$  is the entire parameter vector
- $\alpha$  (alpha) is the learning rate (a small number, usually between 0 and 1)
- $\nabla(J)$  is the entire gradient vector

**Note:** Do you still wonder why there is a subtraction in the equation?

By definition, the gradient indicates the direction towards which we should adjust the  $\theta$  parameters if we wanted to *increase* the cost. But since our optimization objective is to *minimize* the cost, we move  $\theta$  in the opposite direction of the gradient (hence the name *gradient descent*).

# Exercise 06 - Multivariate Gradient Descent

---

Turn-in directory :	ex06
Files to turn in :	fit.py
Authorized modules :	numpy
Forbidden functions :	any function that performs derivatives for you

---

## Objectives:

- Implement a function to perform linear gradient descent (LGD) for multivariate linear regression.

## Instructions:

In this exercise, you will implement linear gradient descent to fit your multivariate model to the dataset.

The pseudocode of the algorithm is the following:

```
repeat until convergence      {
    compute  $\nabla(J)$ 
     $\theta := \theta - \alpha \nabla(J)$ 
}
```

Where:

- $\nabla(J)$  is the entire gradient vector
- $\theta$  is the entire parameter vector
- $\alpha$  (alpha) is the learning rate (a small number, usually between 0 and 1)

You are expected to write a function named `fit__` as per the instructions bellow:

```
def fit_(x, y, theta, alpha, n_cycles):
    """
    Description:
        Fits the model to the training dataset contained in x and y.
    Args:
        x: has to be a numpy.ndarray, a matrix of dimension m * n: (number of training
    ↪ examples, number of features).
        y: has to be a numpy.ndarray, a vector of dimension m * 1: (number of training
    ↪ examples, 1).
        theta: has to be a numpy.ndarray, a vector of dimension (n + 1) * 1: (number of
    ↪ features + 1, 1).
        alpha: has to be a float, the learning rate
        n_cycles: has to be an int, the number of iterations done during the gradient
    ↪ descent
    Returns:
        new_theta: numpy.ndarray, a vector of dimension (number of features + 1, 1).
        None if there is a matching dimension problem.
    Raises:
        This function should not raise any Exception.
    """
    ... your code here ...
```

Hopefully, you have already implemented a function to calculate the multivariate gradient.

## Examples:

```
import numpy as np
x = np.array([[0.2, 2., 20.], [0.4, 4., 40.], [0.6, 6., 60.], [0.8, 8., 80.]])
y = np.array([[19.6], [-2.8], [-25.2], [-47.6]])
theta = np.array([[42.], [1.], [1.], [1.]])

# Example 0:
theta2 = fit_(X2, Y2, theta2, alpha = 0.0005, n_cycle=42000)
theta2
# Output:
array([[41.99...], [0.97...], [0.77...], [-1.20...]])

# Example 1:
predict_(X2, theta2)
# Output:
array([[19.5992...], [-2.8003...], [-25.1999...], [-47.5996...]])
```

## Remarks:

- You can create more training data by generating an  $x$  array with random values and computing the corresponding  $y$  vector as a linear expression of  $x$ . You can then fit a model on this artificial data and find out if it comes out with the same  $\theta$  coefficients that first you used.
- It is possible that  $\theta_0$  and  $\theta_1$  become “[nan]”. In that case, it means you probably used a learning rate that is too large.

# Exercise 07 - Multivariate Linear Regression with Class

---

Turn-in directory :	ex07
Files to turn in :	mylinearregression.py
Authorized modules :	Numpy
Forbidden modules :	sklearn

---

## Objectives:

- Upgrade your Linear Regression class so it can handle multivariate hypotheses.

## Instructions:

You are expected to upgrade your own `MyLinearRegression` class from **day01**. You will upgrade the following methods to support multivariate linear regression:

- `predict_(self, x)`
- `fit_(self, x, y)`

## Examples:

```
import numpy as np
from mylinearregression import MyLinearRegression as MyLR
X = np.array([[1., 1., 2., 3.], [5., 8., 13., 21.], [34., 55., 89., 144.]])
Y = np.array([[23.], [48.], [218.]])
mylr = MyLR([[1.], [1.], [1.], [1.], [1]])

# Example 0:
mylr.predict_(X)
# Output:
array([[8.], [48.], [323.]])

# Example 1:
mylr.cost_elem_(X,Y)
# Output:
array([[37.5], [0.], [1837.5]])

# Example 2:
mylr.cost_(X,Y)
# Output:
1875.0

# Example 3:
mylr.fit_(X, Y)
mylr.thetas
# Output:
array([[18.023..], [3.323..], [-0.711..], [1.605..], [-0.1113..]])

# Example 4:
mylr.predict_(X)
```



```
# Output:
array([[23.499..], [47.385..], [218.079...]])

# Example 5:
mylr.cost_elem_(X,Y)
# Output:
array([[0.041..], [0.062..], [0.001...]])

# Example 6:
mylr.cost_(X,Y)
# Output:
0.1056..
```

# Exercise 08 - Practicing Multivariate Linear Regression

---

Turn-in directory :	ex08
Files to turn in :	multivariate_linear_model.py
Authorized modules :	numpy, matplotlib
Forbidden modules :	sklearn
Remarks :	Read the doc

---

## Objectives:

- Fit a linear regression model to a dataset with multiple features.
- Plot the model's predictions and interpret the graphs.

## Instructions:

Yesterday you performed a univariate linear regression on a dataset to make predictions based on ONE feature (well done!). Now, it's time to dream bigger. Lucky you are, we give you a new dataset with multiple features that you will find in the resources attached.

The dataset is called `spacecraft_data.csv` and it describes a set of spacecrafts with their price, as well as a few other features. A description of the dataset is provided in the file named `spacecraft_data_description.txt`.

## Part One: Univariate Linear Regression

To start, we'll build on yesterday's work and see how a univariate model can predict spaceship prices. As you know, univariate models can only process ONE feature at a time. So to train each model, you need to select a feature and ignore the other ones.

### Instructions:

In the first part of the exercise, you will train three different univariate models to predict spaceship prices. Each model will use a different feature of the spaceships.

#### 1a) Age

Select the *Age* feature as your  $x$  vector, and *Sell\_price* as your  $y$  vector. Train a first model, `myLR_age`, and generate price predictions ( $\hat{y}$ ).

Output a scatter plot with both sets of data points on the same graph, as follows:

- The actual prices, given by  $(x_{age}^{(i)}, y^{(i)})$  for  $i = 0 \dots m$
- The predicted prices, represented by  $(x_{age}^{(i)}, \hat{y}^{(i)})$  for  $i = 0 \dots m$  (see example below),

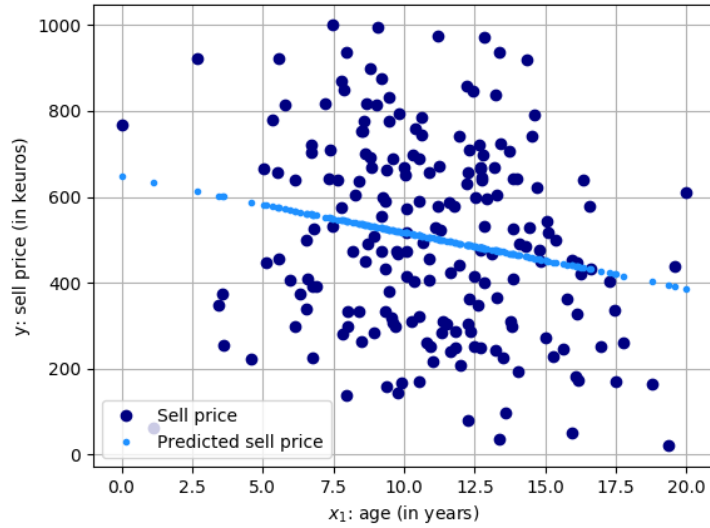


Figure 4: Plot of the selling prices of spacecrafts with respect to their age, as well as our first model's price predictions.

### 1b) Thrust

Select the *Thrust\_power* feature as your  $x$  vector, and *Sell\_price* as your  $y$  vector. Train a second model, `myLR_thrust`, and generate price predictions ( $\hat{y}$ ).

Output a scatter plot with both sets of data points on the same graph, as follows:

- The actual prices, given by  $(x_{thrust}^{(i)}, y^{(i)})$  for  $i = 0 \dots m$
- The predicted prices, represented by  $(x_{thrust}^{(i)}, \hat{y}^{(i)})$  for  $i = 0 \dots m$  (see example below),

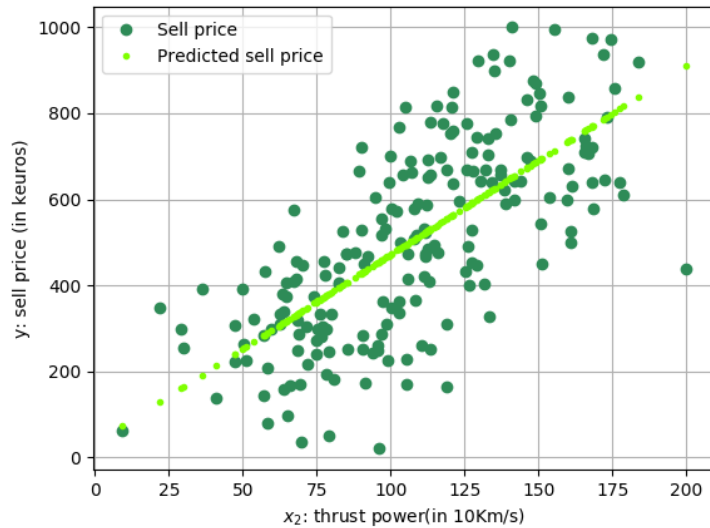


Figure 5: Plot of the selling prices of spacecrafts with respect to the thrust power of their engines, as well as our second model's price predictions.

### 1c) Total distance

Select the *Terameters* feature as your  $x$  vector, and *Sell\_price* as your  $y$  vector. Train a third model, `myLR_distance`, and make price predictions ( $\hat{y}$ ).

Output a scatter plot with both sets of data points on the same graph, as follows:

- The actual prices, given by  $(x_{distance}^{(i)}, y^{(i)})$  for  $i = 0 \dots m$
- The predicted prices, represented by  $(x_{distance}^{(i)}, \hat{y}^{(i)})$  for  $i = 0 \dots m$  (see example below),

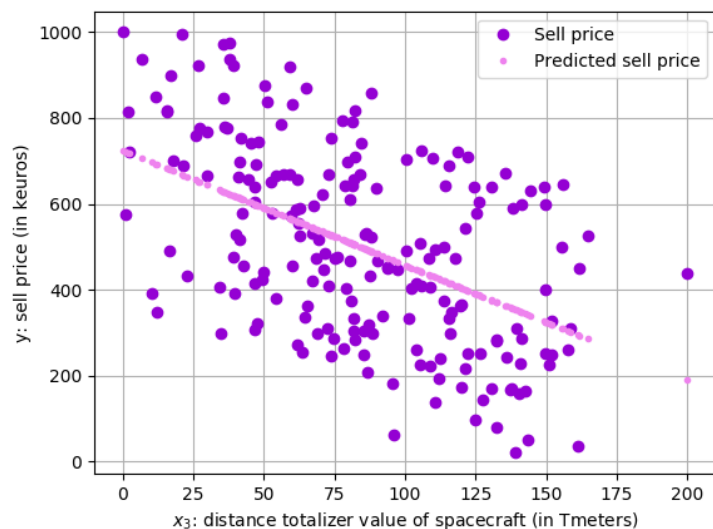


Figure 6: Plot of the selling prices of spacecrafts with respect to the terameters driven, as well as our third model's price predictions.

### Reminder:

- After executing the `fit_` method, you may obtain  $\theta = \text{array}([[ \text{nan}, \text{nan} ]])$ . If it happens, try reducing your learning rate.
- Be aware that you also need to set the appropriate number of cycles for the `fit_` function. If it's too low, you might not be leaving enough cycles for the gradient descent to carry out properly. Try to find a value that gets you the best score, but that doesn't make the training last forever.

### Hint:

First, try plotting the data points  $(x_j, y)$ . Then you can guess initial theta values that are not too far off. This will help your algorithm converge more easily.

### Examples:

```
import pandas as pd
import numpy as np
from mylinearregression import MyLinearRegression as MyLR

data = pd.read_csv("spacecraft_data.csv")
X = np.array(data[['Age']])
Y = np.array(data[['Sell_price']])
myLR_age = MyLR([1000.0], [-1.0])
myLR_age.fit(X[:,0].reshape(-1,1), Y, alpha = 2.5e-5, n_cycle = 100000)
```

```
RMSE_age = myLR_age.mse_(X[:,0].reshape(-1,1),Y)
print(RMSE_age)
57636.77729...
```

How accurate is your model when you only take one feature into account?

## Part Two: Multivariate Linear Regression (A New Hope)

Now, it's time for your first multivariate linear regression! Here, you will train a single model that will take all features into account.

### Instructions:

#### 2a) Training the model

- Train a single multivariate linear regression model on all three features.
- Display and interpret the resulting theta parameters. What can you say about the role that each feature plays in the price prediction?
- Evaluate the model with the Mean Squared Error. How good is your model doing, compared to the other three that you trained in Part One of this exercise?

**Remarks:** You can obtain a better fit if you increase the number of cycles.

### Examples:

```
import pandas as pd
import numpy as np
from mylinearregression import MyLinearRegression as MyLR

data = pd.read_csv("spacecraft_data.csv")
X = np.array(data[['Age', 'Thrust_power', 'Terameters']])
Y = np.array(data[['Sell_price']])
my_lreg = MyLR([1.0, 1.0, 1.0, 1.0])

my_lreg.mse_(X,Y)
# 144044.877...

my_lreg.fit(X,Y, alpha = 1e-4, n_cycle = 600000)
my_lreg.theta
# array([[334.994...], [-22.535...], [5.857...], [-2.586...]])

my_lreg.mse_(X,Y)
# 586.896999...
```

#### 2b) Plotting the predictions

Here we'll plot the model's predictions just like we did in Part One. We'll make three graphs, each one displaying the predictions and the actual prices as a function of ONE of the features.

- On the same graph, plot the actual and predicted prices on the  $y$  axis , and the *age* feature on the  $x$  axis. (see figure below)

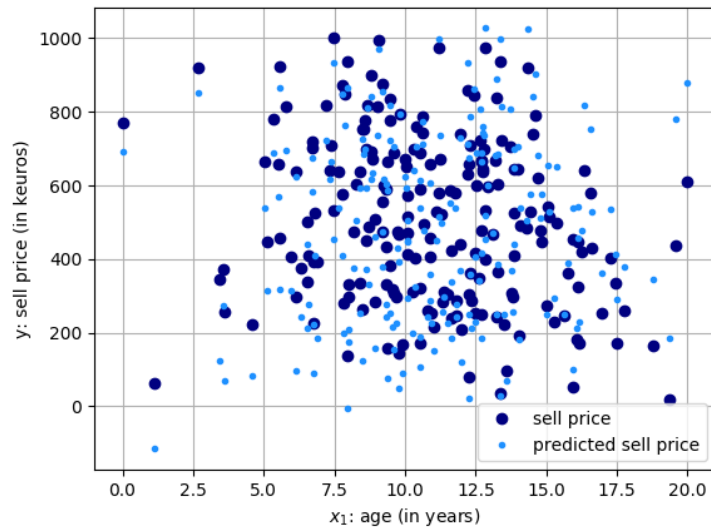


Figure 7: Spacecraft sell prices of and predicted sell prices with the multivariate hypothesis, with respect to the *age* feature

- On the same graph, plot the actual and predicted prices on the  $y$  axis , and the *thrustpower* feature on the  $x$  axis. (see figure below)

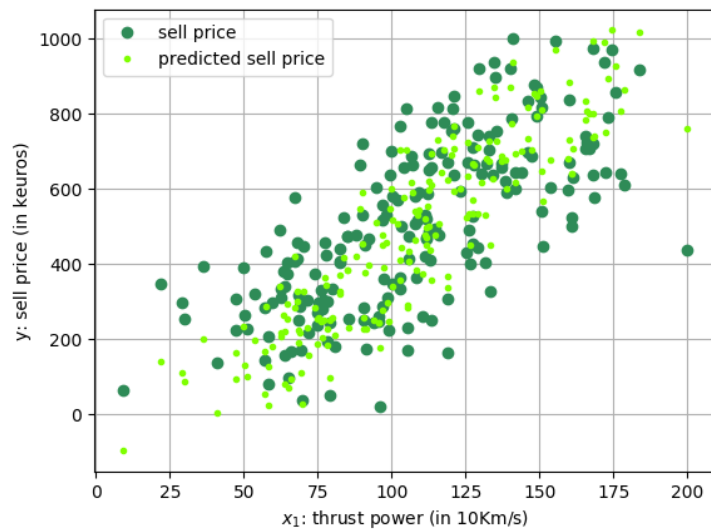


Figure 8: Spacecraft sell prices predicted sell prices with the multivariate hypothesis, with respect to the thrust power of the engines

- On the same graph, plot the actual and predicted prices on the  $y$  axis, and the *distance* feature on the  $x$  axis. (see figure below)

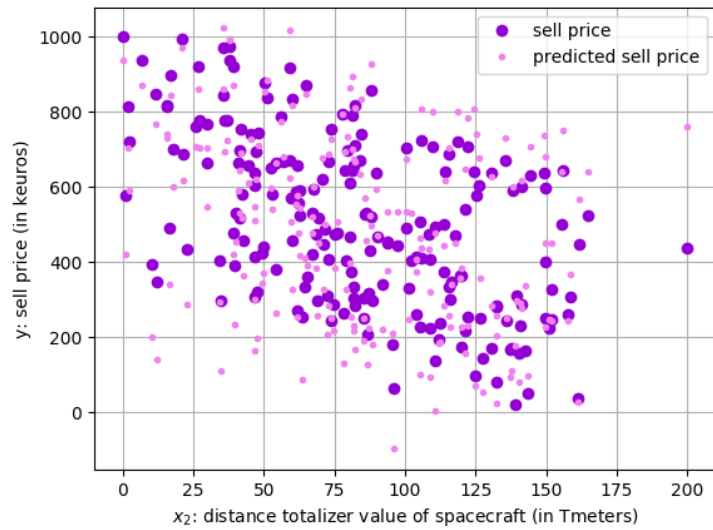


Figure 9: Spacecraft sell prices and predicted sell prices with the multivariage hypothesis, with respect to the driven distance (in terameters)

Can you see any improvement on these three graphs, compared to the three that you obtained in Part One?  
 Can you relate your observations to the MSE value that you just calculated?

# Exercise 9 - Question Time!

**Are you able to clearly and simply explain:**

- 1 - What is the main difference between univariate and multivariate linear regression, in terms of variables?
- 2 - Is there a minimum number of variables needed to perform a multivariate linear regression? If yes, which one?
- 3 - Is there a maximum number of variables needed to perform a multivariate linear regression? If yes, which one?
- 4 - Is there a difference between univariate and multivariate linear regression in terms of performance evaluation?
- 5 - What does it mean geometrically to perform a multivariate gradient descent with two variables?



# Interlude - Introducing Polynomial Models

You probably noticed that the method we use is called *linear regression* for a reason: the model generates all of its predictions on a straight line. However, we often encounter features that don't have a linear relationship with the predicted variable, like in the figure below:

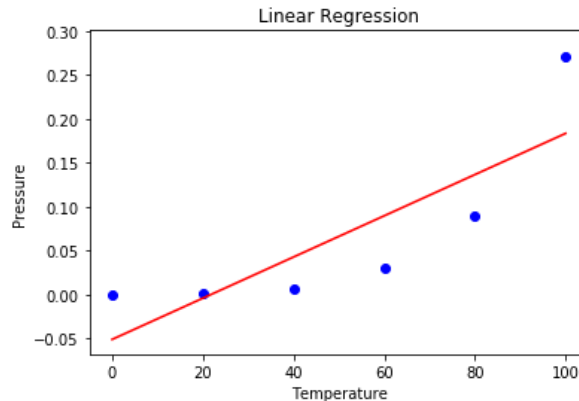


Figure 10: Non-linear relationship

In that case, we are stuck with a straight line that can't fit the data points properly. In this example, what if we could express  $y$  not as a function of  $x$ , but also of  $x^2$ , and maybe even  $x^3$  and  $x^4$ ? We could make a hypothesis that draws a nice **curve** that would better fit the data. That's where polynomial features can help!

## Polynomial features

First we get to do some *feature engineering*. We create new features by raising our initial  $x$  feature to the power of 2, and then 3, 4... as far as we want to go. For each new feature we need to create a new column in the dataset.

## Polynomial Hypothesis

Now that we created our new features, we can combine them in a linear hypothesis that looks just the same as what we're used to:

$$\hat{y} = \theta_0 + \theta_1 x + \theta_2 x^2 + \dots + \theta_n x^n$$

It's a little strange because we are building a linear combination, not with different features but with different powers of the same feature. This is a first way of introducing non-linearity in a regression model!

# Exercise 10 - Polynomial models

---

Turn-in directory :	ex10
Files to turn in :	polynomial_model.py
Authorized modules :	numpy
Forbidden modules :	sklearn

---

## Objectives:

Create a function that takes a vector  $x$  of dimension  $m * 1$  and an integer  $n$  as input, and returns a matrix of dimension  $m * n$ .

Each column of the matrix contains  $x$  raised to the power of  $j$ , for  $j = 1, 2, \dots, n$ :

$$x \quad | \quad x^2 \quad | \quad x^3 \quad | \quad \dots \quad | \quad x^n$$

## Instructions:

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

```
def add_polynomial_features(x, power):  
    """Add polynomial features to vector x by raising its values up to the power given in  
    → argument.  
    Args:  
        x: has to be a numpy.ndarray, a vector of dimension m * 1.  
        power: has to be an int, the power up to which the components of vector x are going to  
    → be raised.  
    Returns:  
        The matrix of polynomial features as a numpy.ndarray, of dimension m * n, containing  
    → polynomial feature values for all training examples.  
        None if x is an empty numpy.ndarray.  
    Raises:  
        This function should not raise any Exception.  
    """
```

## Examples:

```
import numpy as np  
x = np.arange(1,6).reshape(-1, 1)  
  
# Example 1:  
add_polynomial_features(x, 3)  
# Output:  
array([[ 1,  1,  1],  
       [ 2,  4,  8],  
       [ 3,  9, 27],  
       [ 4, 16, 64],  
       [ 5, 25, 125]])  
  
# Example 2:  
add_polynomial_features(x, 6)  
# Output:  
array([[ 1,  1,  1,  1,  1,  1],  
       [ 2,  4,  8, 16, 32, 64],  
       [ 3,  9, 27, 81, 243, 729],  
       [ 4, 16, 64, 256, 1024, 4096],  
       [ 5, 25, 125, 625, 3125, 15625]])
```

```
[ 2, 4, 8, 16, 32, 64],  
[ 3, 9, 27, 81, 243, 729],  
[ 4, 16, 64, 256, 1024, 4096],  
[ 5, 25, 125, 625, 3125, 15625]])
```

# Exercise 11 - Let's Train Polynomial Models!

---

Turn-in directory :	ex11
Files to turn in :	polynomial_train.py
Authorized modules :	numpy
Forbidden modules :	sklearn

---

## Objectives:

It's training time!

Let's train some polynomial models, and see if those with higher polynomial degree perform better!

## Instructions:

- Take your `are_blue_pills_magic.csv` dataset and train **nine** separate Linear Regression models with polynomial hypotheses with degrees ranging from 2 to 10.
- Evaluate the cost of each of the nine models.
- To properly visualize your results, make a bar plot showing the cost of the models given their polynomial degree.

According to your evaluations, what is the best hypothesis (or model) you can get?

**Terminology Note:** The **degree** of a polynomial expression is its highest exponent.

E.g.: The polynomial degree of  $5x^3 - x^6 + 2x^2$  is 6.

Here in this equation, you don't see any terms with  $x$ ,  $x^4$  and  $x^5$ , but we can still say they exist. It's just that their coefficient is 0. This means that a polynomial linear regression model can lower the impact of any term by bringing its corresponding  $\theta_j$  closer to 0.

# Interlude - Plotting Curves With Matplotlib

We asked you to plot straight lines in the day00. Now you are working with polynomial models, the hypothesis functions are not straight lines, but curves.

Plotting curves is a bit more tricky, because if you do not have enough data point, you will get an ugly broken line instead of a smooth curve.

Here's a way to do it.

Let's begin with a simple dataset:

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

x = np.arange(1,11).reshape(-1,1)
y = np.array([[ 1.39270298],
              [ 3.88237651],
              [ 4.37726357],
              [ 4.63389049],
              [ 7.79814439],
              [ 6.41717461],
              [ 8.63429886],
              [ 8.19939795],
              [10.37567392],
              [10.68238222]])

plt.scatter(x,y)
plt.show()
```

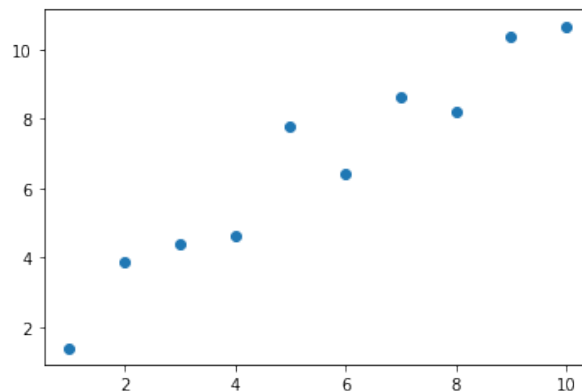


Figure 11: Scatter plot of a dataset

Now, we build a polynomial model of degree 3 and plot its hypothesis function  $h(\theta)$ .

```
from polynomial_model import add_polynomial_features
from mylinearregression import MyLinearRegression as MyLR

# Build the model:
x_ = add_polynomial_features(x, 3)
my_lr = MyLR(np.ones(4).reshape(-1,1)).fit_(x_, y)

# Plot:
## To get a smooth curve, we need a lot of data points
continuous_x = np.arange(1,10.01, 0.01).reshape(-1,1)
x_ = add_polynomial_features(continuous_x, 3)
```

```
y_hat = my_lr.predict_(continuous_x)

plt.scatter(x,y)
plt.plot(continuous_x, y_hat, color='orange')
plt.show()
```

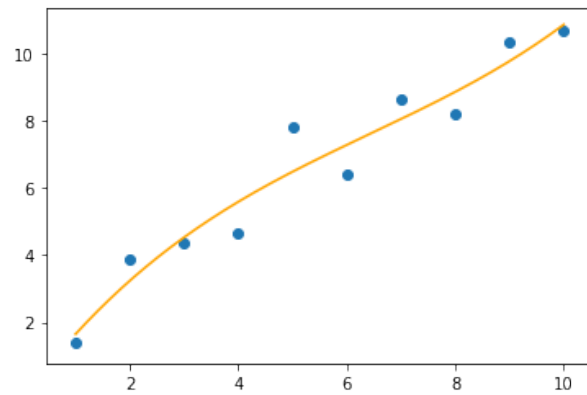


Figure 12: Scatter plot of a dataset, and on top, a plot of the polynomial hypothesis function

# Exercise 12 - Let's PLOT some Polynomial Models!

---

Turn-in directory :	ex12
Files to turn in :	polynomial_train.py
Authorized modules :	numpy
Forbidden modules :	sklearn

---

## Objectives:

It's plotting time!

## Instructions:

- For each model you built in the last exercise, plot its hypothesis function  $h(\theta)$  on top of a scatter plot of the original data points  $(x, y)$ .
- Answer the following questions:
  1. From a purely intuitive point of view, which hypothesis (i.e. which polynomial degree) seems to best represent the relationship between  $y$  and  $x$ ?
  2. Go back to your answer for the previous exercise. What polynomial degree had you identified as the most accurate on the dataset? Is it the same one as the curve you just picked as the most representative of the relationship between  $y$  and  $x$ ?
  3. Find the plot that illustrates that same hypothesis you had found was the most accurate in the previous exercise. Take a closer look at the curve. What do you think? Does it seem to properly represent the relationship between  $y$  and  $x$ ?

# Interlude - Lost in Overfitting

The two previous exercises lead you, dear reader, to a very dangerous territory: the realm of **overfitting**. You did not see it coming but now, you are in a bad situation...

By increasing the polynomial degree of your model, you increased its **complexity**.

Is it wrong?

Not always.

Some models are indeed very complex because the relationships they represent are very complex as well.

But, if you look at the plots for the previous exercise's *best model*, you should feel that something is wrong.

## Something is rotten in the state of our model...

Take a look at the following plot.

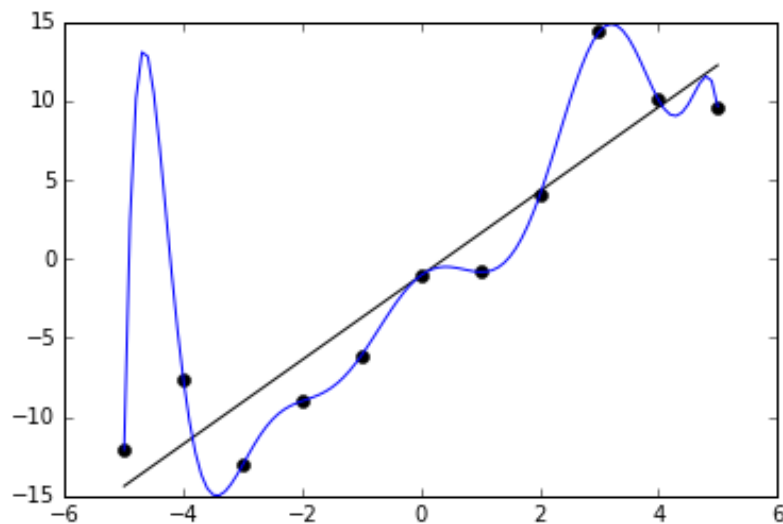


Figure 13: Overfitting hypothesis

You can see that the prediction line fits each data point perfectly, but completely misses out on capturing the relationship between  $x$  and  $y$  properly.

And now, if we add some brand new data points to the dataset, we see that the predictions on those new examples are way off.



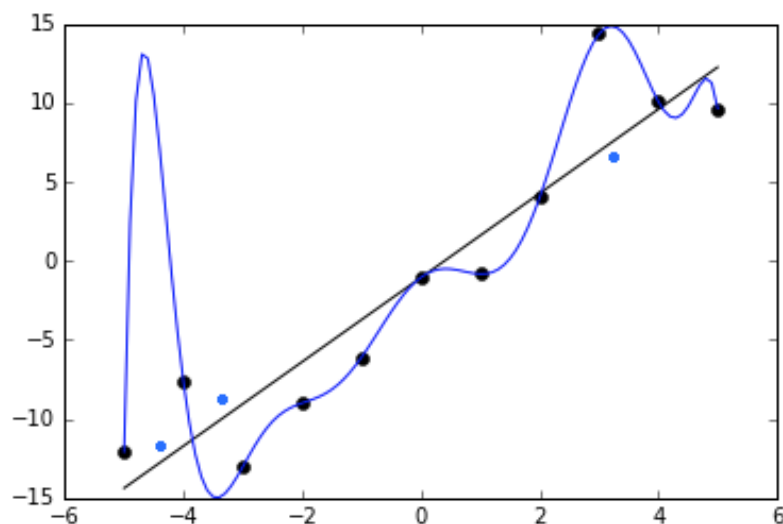


Figure 14: Generalization errors resulting from overfitting

This situation is called overfitting, because the model is doing an excessively good job at fitting the data. It is literally bending over backward to account for the data's minute details. But most the data's irregularities are just noise, and they should in fact be ignored. So because the model overfit, it can't generalize to new data.

## The training set, the test set, and the happy data scientist

To be able to detect overfitting, **you should always evaluate your model on new data.**

New data means, data that your model hasn't seen during training. It's the only way to make sure your model isn't *cheating*. To do so, now and forever, you must always divide your dataset in (at least) two parts: one for the training, and one for the evaluation of your model.

# Exercise 13 - DataSplitter

---

Turn-in directory :	ex13
Files to turn in :	data_splitter.py
Authorized modules :	Numpy
Forbidden modules :	sklearn

---

## Objectives:

Learn how to split a dataset into a **training set** and a **test set**.

## Instructions:

You must implement a function that **shuffles** and **splits** a dataset it in two parts: a **training set** and a **test set**.

- Your function will shuffle and split the  $X$  matrix while keeping a certain **proportion** of the examples for training, and the rest for testing.
- Your function will also shuffle and split the  $y$  vector while making sure that the order of the rows in the output match the order of the rows in the split  $X$  output.

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

```
def data_splitter(x, y, proportion):  
    """Shuffles and splits the dataset (given by x and y) into a training and a test set,  
    → while respecting the given proportion of examples to be kept in the training set.  
    Args:  
        x: has to be a numpy.ndarray, a matrix of dimension m * n.  
        y: has to be a numpy.ndarray, a vector of dimension m * 1.  
        proportion: has to be a float, the proportion of the dataset that will be assigned to  
    → the training set.  
    Returns:  
        (x_train, x_test, y_train, y_test) as a tuple of numpy.ndarray  
        None if x or y is an empty numpy.ndarray.  
        None if x and y do not share compatible dimensions.  
    Raises:  
        This function should not raise any Exception.  
    """
```

## Be careful!

- The dataset has to be randomly shuffled *before* it is split into training and test sets.
- Unless you use the same seed in your randomization algorithm, you won't get the same results twice.

## Examples:

The following examples are just an indication of possible outputs. As long as you have shuffled datasets with their corresponding  $y$  values, your function is working correctly.

```
import numpy as np  
x1 = np.array([1, 42, 300, 10, 59])  
y = np.array([0,1,0,1,0])
```

```

# Example 1:
data_splitter(x1, y, 0.8)
# Output:
(array([ 1, 59, 42, 300]), array([10]), array([0, 0, 0, 1]), array([1]))

# Example 2:
data_splitter(x1, y, 0.5)
# Output:
(array([59, 10]), array([ 1, 300, 42]), array([0, 1]), array([0, 1, 0]))

x2 = np.array([ [ 1, 42],
                [300, 10],
                [ 59, 1],
                [300, 59],
                [ 10, 42]])
y = np.array([0,1,0,1,0])

# Example 3:
data_splitter(x2, y, 0.8)
# Output:
(array([[ 10, 42],
        [300, 59],
        [ 59, 1],
        [300, 10]]), array([[ 1, 42]]), array([0, 1, 0, 1]), array([0]))

# Example 4:
data_splitter(x2, y, 0.5)
# Output:
(array([[59, 1],
        [10, 42]]), array([[300, 10],
        [300, 59],
        [ 1, 42]]), array([0, 0]), array([1, 1, 0]))

# Be careful! The way tuples of arrays are displayed could be a bit confusing...
#
# In the last example, the tuple returned contains the following arrays:
# array([[59, 1],
#        [10, 42]])
#
# array([[300, 10],
#        [300, 59]
#
# array([0, 0])
#
# array([1, 1, 0])

```

# Exercise 14 - Machine Learning for Grown-ups: Training and Test Sets

---

Turn-in directory :	ex14
Files to turn in :	data_splitter.py
Authorized modules :	numpy
Forbidden modules :	sklearn

---

## Objectives:

Let's do Machine Learning for real!

## Instructions:

- Split your `are_blue_pills_magic.csv` dataset into a training and a test set.
- Use your `polynomial_features` method on your training set
- Train nine Linear Regression models with polynomial hypotheses with degrees ranging from 2 to 10.
- Evaluate your models on the test set.

According to your model evaluations, what is the best hypothesis you can get?

# Exercise 15 - Question Time!

**Are you able to clearly and simply explain:**

- 1 - What is overfitting?
- 2 - What do you think underfitting might be?
- 3 - Why is it important to split the data set in a training and a test set?
- 4 - If a model overfits, what will happen when you compare its performance on the training set and the test set?
- 5 - If a model underfits, what do you think will happen when you compare its performance on the training set and the test set?