
# Machine Learning - Module 02

## Multivariate Linear Regression

*Summary: Building on what you did on the previous modules 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 and ressources

## Notions of the module

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

## Useful Ressources

You are strongly advise to use the following resource: Machine Learning MOOC - Stanford These videos are available at no cost; simply log in, select "Enroll for Free", then click "Audit" at the bottom of the pop-up window. The following sections of the course are pertinent to today's exercises:

### Week 2: Regression with multiple input variables

#### Multiple linear regression

- Multiple features
- Gradient descent for multiple linear regression

#### Gradient descent in practice

- Feature scaling part 1
- Feature scaling part 2
- Checking gradient descent for convergence
- Choosing the learning rate
- Feature engineering
- Polynomial regression

*All videos above are available also on this Andrew Ng's YouTube playlist, videos 21 and from 24 to 30*

# Chapter I

# Common Instructions

- The version of Python recommended to use is 3.7, you can check the version of Python with the following command: `python -V`
- The norm: during this bootcamp, it is recommended to follow the PEP 8 standards, though it is not mandatory. You can install pycodestyle which is a tool to check your Python code.
- The function `eval` is never allowed.
- The exercises are ordered from the easiest to the hardest.
- Your exercises are going to be evaluated by someone else, so make sure that your variable names and function names are appropriate and civil.
- Your manual is the internet.
- If you are a student from 42, you can access our Discord server on 42 student's associations portal and ask your questions to your peers in the dedicated Bootcamp channel.
- You can learn more about 42 Artificial Intelligence by visiting our website.
- If you find any issue or mistake in the subject please create an issue on 42AI repository on Github.
- We encourage you to create test programs for your project even though this work **won't have to be submitted and won't be graded**. It will give you a chance to easily test your work and your peers' work. You will find those tests especially useful during your defence. Indeed, during defence, you are free to use your tests and/or the tests of the peer you are evaluating.
- We are constantly looking to improve these bootcamps, and your feedbacks are essential for us to do so !
  You can tell us more about your experience with this module by filling this form.
  Thank you in advance and good luck for this bootcamp !

# Contents

# Chapter II

# Exercise 00

## Interlude - To the Multivariate Universe and Beyond!

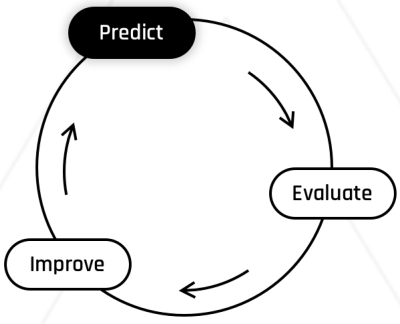
Until now we've used a very simple hypothesis: $h_\theta(x) = \theta_0 + \theta_1 x$. With this very simple hypothesis we found a way to evaluate and improve our predictions.

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!

## Interlude - Predict




## Representing the examples as an $(m \times n)$ 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 $m$, we now have a matrix of dimension $(m \times n)$, 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$^{\text{th}}$ training example, (i$^{\text{th}}$ row of the $X$ matrix),
- $x_j$ is the j$^{\text{th}}$ column of the $X$ matrix,
- $x_j^{(i)}$ is the j$^{\text{th}}$ feature of the i$^{\text{th}}$ training example (at the intersection of the i$^{\text{th}}$ row and the j$^{\text{th}}$ 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.

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

Where:

- $\hat{y}^{(i)}$ is the model's prediction for the i$^{\text{th}}$ example,
- $x_1^{(i)}$ is the first feature of the i$^{\text{th}}$ example,
- $x_n^{(i)}$ is the n$^{\text{th}}$ feature of the i$^{\text{th}}$ example,
- $\theta$ is a vector of dimension $(n + 1)$, 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 : 00 |
|---|---|
| Multivariate Hypothesis - Iterative Version | |
| Turn-in directory : $ex00/$ | |
| Files to turn in : `prediction.py` | |
| Forbidden functions : `None` | |

# Objective

Manipulate the hypothesis to make prediction. You must implement the following formula as a function:

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

Where:

- $\hat{y}$ is a vector of dimension $m$: the vector of predicted values,
- $\hat{y}^{(i)}$ is the i$^{\text{th}}$ component of the $\hat{y}$ vector: the predicted value for the i$^{\text{th}}$ example,
- $\theta$ is a vector of dimension $(n+1)$: the parameter vector,
- $\theta_j$ is the j$^{\text{th}}$ component of the parameter vector,
- $X$ is a matrix of dimensions $(m \times n)$: the design matrix,
- $x^{(i)}$ is the i$^{\text{th}}$ row of the $X$ matrix: the feature vector of the i$^{\text{th}}$ example,
- $x_j$ is the j$^{\text{th}}$ column of the $X$ matrix,
- $x_j^{(i)}$ is the element at the intersection of the i$^{\text{th}}$ row and the j$^{\text{th}}$ column of the $X$ matrix: the j$^{\text{th}}$ feature of the i$^{\text{th}}$ 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.array.
    Args:
      x: has to be an numpy.array, a matrix of dimension m * n.
      theta: has to be an numpy.array, a vector of dimension (n + 1) * 1.
    Return:
      y_hat as a numpy.array, a vector of dimension m * 1.
      None if x or theta are empty numpy.array.
      None if x or theta dimensions are not matching.
      None if x or theta is not of expected type.
    Raises:
      This function should not raise any Exception.
    """
    ... Your code ...
```

# Examples

```
import numpy as np
x = np.arange(1,13).reshape((4,-1))

# Example 1:
theta1 = np.array([5, 0, 0, 0]).reshape((-1, 1))
simple_predict(x, theta1)
# Ouput:
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]).reshape((-1, 1))
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]).reshape((-1, 1))
simple_predict(x, theta3)
# Output:
array([[ 9.64], [24.28], [38.92], [53.56]])


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

# Chapter III

# Exercise 01

## 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 $\theta$.

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

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.

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

| | Exercise : 01 |
|---|---|
| Multivariate hypothesis - vectorized version | |
| Turn-in directory : $ex01/$ | |
| Files to turn in : `prediction.py` | |
| Forbidden functions : `None` | |

# Objective

Manipulate the hypothesis to make prediction. You must implement the following formula as a function:

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

Where:

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

Be careful:

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

# 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.array.
    Args:
      x: has to be an numpy.array, a vector of dimensions m * n.
      theta: has to be an numpy.array, a vector of dimensions (n + 1) * 1.
    Return:
      y_hat as a numpy.array, a vector of dimensions m * 1.
      None if x or theta are empty numpy.array.
      None if x or theta dimensions are not appropriate.
      None if x or theta is not of expected type.
    Raises:
      This function should not raise any Exception.
    """
    ... Your code ...
```

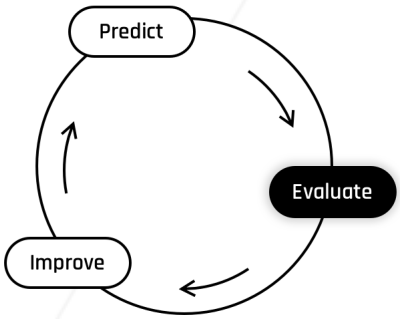
# Examples

```
import numpy as np
x = np.arange(1,13).reshape((4,-1))

# Example 1:
theta1 = np.array([5, 0, 0, 0]).reshape((-1, 1))
predict_(x, theta1)
# Ouput:
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]).reshape((-1, 1))
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]).reshape((-1, 1))
predict_(x, theta3)
# Output:
array([[ 9.64], [24.28], [38.92], [53.56]])


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

# Chapter IV

# Exercise 02

## Interlude - Evaluate




## Back to the Loss Function

How is our model doing?

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

Two modules ago, we defined the loss 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):

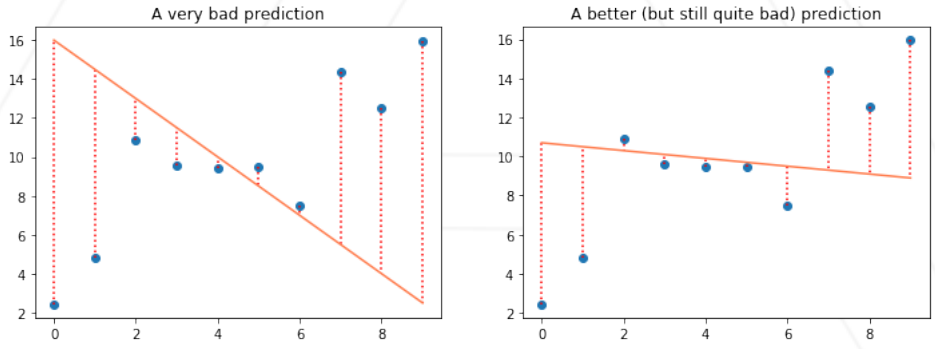



Figure IV.1: Distances between predicted and expected values

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 intervened in the equation. Indeed, the loss 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 loss function as we did before!

| | Exercise : 02 |
|---|---|
| Vectorized Loss Function | |
| Turn-in directory : $ex02/$ | |
| Files to turn in : `loss.py` | |
| Forbidden functions : `None` | |

# Objective

Understand and manipulate loss function for multivariate linear regression. You must implement the following formula as a function:

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

Where:

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

# Instructions

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

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

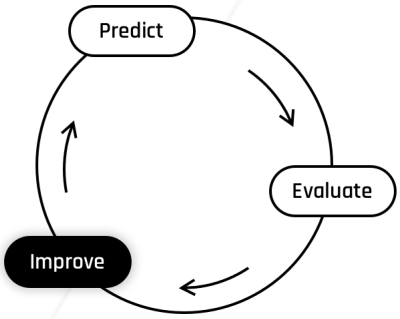
# Examples

```
import numpy as np
X = np.array([0, 15, -9, 7, 12, 3, -21]).reshape((-1, 1))
Y = np.array([2, 14, -13, 5, 12, 4, -19]).reshape((-1, 1))

# Example 1:
loss_(X, Y)
# Output:
2.142857142857143

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

# Chapter V

# Exercise 03

## Interlude - Improve with the Gradient




## Multivariate Gradient

From our multivariate linear hypothesis we can derive our multivariate gradient. It looks a lot like the one we saw during the previous module, 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 $\nabla(J)_0, \nabla(J)_1, \ldots, \nabla(J)_n$.

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

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

Where:

- $\nabla(J)$ is a vector of dimension $(n+1)$, the gradient vector,
- $\nabla(J)_j$ is the $\text{j}^{\text{th}}$ component of $\nabla(J)$, the partial derivative of $J$ with respect to $\theta_j$,
- $y$ is a vector of dimension $m$, the vector of expected values,
- $y^{(i)}$ is a scalar, the $\text{i}^{\text{th}}$ component of vector $y$,
- $x^{(i)}$ is the feature vector of the $\text{i}^{\text{th}}$ example,

- $x_j^{(i)}$ is a scalar, the j$^{\text{th}}$ feature value of the i$^{\text{th}}$ 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:

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

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 dimension $(n+1)$,
- $X'$ is a matrix of dimension $(m \times (n+1))$, 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 dimension $(n+1)$, the parameter vector,
- $y$ is a vector of dimension $m$, 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 : 03 |
|---|---|
| Multivariate Linear Gradient | |
| Turn-in directory : $ex03/$ | |
| Files to turn in : `gradient.py` | |
| Forbidden functions : `None` | |

# Objective

Understand and manipulate concept of gradient in the case of multivariate formulation. 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 $(n+1)$, the gradient vector,
- $X$ is a matrix of dimensions $(m \times n)$, the design matrix,
- $X'$ is a matrix of dimensions $(m \times (n+1))$, the design matrix onto which a column of 1's was added as a first column,
- $\theta$ is a vector of dimension $(n+1)$, the parameter vector,
- $y$ is a vector of dimension $m$, 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.array, without any for-loop.
    The three arrays must have the compatible dimensions.
    Args:
      x: has to be an numpy.array, a matrix of dimension m * n.
      y: has to be an numpy.array, a vector of dimension m * 1.
      theta: has to be an numpy.array, a vector (n +1) * 1.
    Return:
      The gradient as a numpy.array, a vector of dimensions n * 1,
        containg the result of the formula for all j.
      None if x, y, or theta are empty numpy.array.
      None if x, y and theta do not have compatible dimensions.
      None if x, y or theta is not of expected type.
    Raises:
      This function should not raise any Exception.
    """
    ... Your code ...
```

# 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]).reshape((-1, 1))
theta1 = np.array([0, 3, 0.5, -6]).reshape((-1, 1))

# Example :
gradient(x, y, theta1)
# Output:
array([[ -33.71428571], [ -37.35714286], [183.14285714], [-393.]])


# Example :
theta2 = np.array([0, 0, 0, 0]).reshape((-1, 1))
gradient(x, y, theta2)
# Output:
array([[ -0.71428571], [  0.85714286], [23.28571429], [-26.42857143]])
```

# Chapter VI

# Exercise 04

## 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**
$\nabla(J)\ \theta_j \leftarrow \theta_j - \alpha\nabla(J)_j$
*simultaneously update*
$\theta$ for j=0,1,...,n

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

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

If you started to like vectorized forms, you might have noticed that 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 $(n+1)$. So all we need to do is this:

**repeat** until convergence
**compute**
$\nabla(J)\ \theta \leftarrow \theta - \alpha\nabla(J)$

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 loss. But since our optimization objective is to minimize the loss, we move $\theta$ in the opposite direction of the gradient (hence the name gradient descent).

|  | Exercise : 04 |
|---|---|
| Multivariate Gradient Descent | |
| Turn-in directory : $ex04/$ | |
| Files to turn in : `fit.py` | |
| Forbidden functions : `any function that performs derivatives for you` | |

# Objective

Understand and manipulate the concept of gradient descent in the case of multivariate linear regression. 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:

$$\begin{aligned} &\text{repeat until convergence } \{ \\ &\quad \text{compute } \nabla(J) \\ &\quad \theta := \theta - \alpha\nabla(J) \\ &\} \end{aligned}$$

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, max_iter):
  """
  Description:
    Fits the model to the training dataset contained in x and y.
  Args:
    x: has to be a numpy.array, a matrix of dimension m * n:
                (number of training examples, number of features).
    y: has to be a numpy.array, a vector of dimension m * 1:
                (number of training examples, 1).
    theta: has to be a numpy.array, a vector of dimension (n + 1) * 1:
                (number of features + 1, 1).
    alpha: has to be a float, the learning rate
    max_iter: has to be an int, the number of iterations done during the gradient descent
  Return:
    new_theta: numpy.array, a vector of dimension (number of features + 1, 1).
    None if there is a matching dimension problem.
    None if x, y, theta, alpha or max_iter is not of expected type.
  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_(x, y, theta,  alpha = 0.0005, max_iter=42000)
theta2
# Output:
array([[41.99..],[0.97..], [0.77..], [-1.20..]])

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

- 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.

# Chapter VII

# Exercise 05

|  | Exercise : 05 |
|---|---|
| Multivariate Linear Regression with Class | |
| Turn-in directory : $ex05/$ | |
| Files to turn in : `mylinearregression.py` | |
| Forbidden functions : `sklearn` | |

## Objective

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

## Instructions

You are expected to upgrade your own `MyLinearRegression` class from **Module01**. You will upgrade (at least) the following methods to support multivariate linear regression:

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

Depending on how you implement your methods, you might need to update other methods.

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


# Example 1:
mylr.loss_elem_(Y, y_hat)
# Output:
array([[225.], [0.], [11025.]])


# Example 2:
mylr.loss_(Y, y_hat)
# Output:
1875.0


# Example 3:
mylr.alpha = 1.6e-4
mylr.max_iter = 200000
mylr.fit_(X, Y)
mylr.thetas
# Output:
array([[18.188..], [2.767..], [-0.374..], [1.392..], [0.017..]])


# Example 4:
y_hat = mylr.predict_(X)
# Output:
array([[23.417..], [47.489..], [218.065...]])


# Example 5:
mylr.loss_elem_(Y, y_hat)
# Output:
array([[0.174..], [0.260..], [0.004..]])


# Example 6:
mylr.loss_(Y, y_hat)
# Output:
0.0732..
```

# Chapter VIII

# Exercise 06

| | Exercise : 06 |
|---|---|
| Practicing Multivariate Linear Regression | |
| Turn-in directory : $ex06/$ | |
| Files to turn in : `multivariate_linear_model.py` | |
| Forbidden functions : `sklearn` | |

## Objective

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

## Dataset

During last module, 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 the previous module'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. For each feature, your program has to perform a gradient descent from a new set of thetas, plot or generate a plot, print the final value of the thetas and the MSE of the corresponding model.

### 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 $(x^{(i)}_{age}, y^{(i)})$ for $i = 0....m$,
- The predicted prices, represented by $(x^{(i)}_{age}, \hat{y}^{(i)})$ for $i = 0....m$ (see example below).

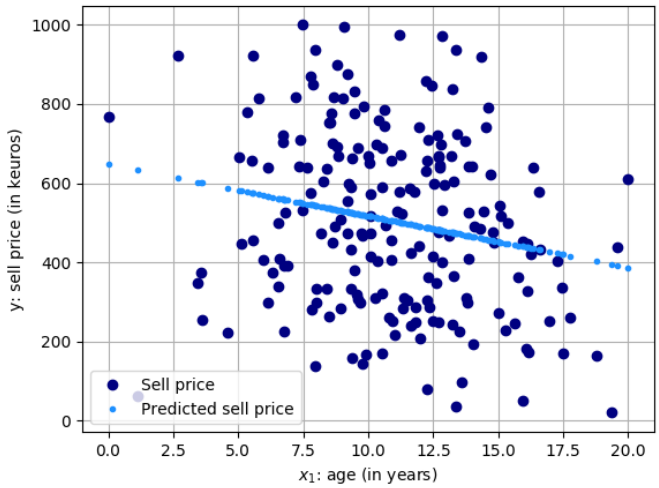



Figure VIII.1: Plot of the selling prices of spacecrafts with respect to their age, as well as our first model's price predictions.

### 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 $(x^{(i)}_{thrust}, y^{(i)})$ for $i = 0....m$,
- The predicted prices, represented by $(x^{(i)}_{thrust}, \hat{y}^{(i)})$ for $i = 0....m$ (see example below).

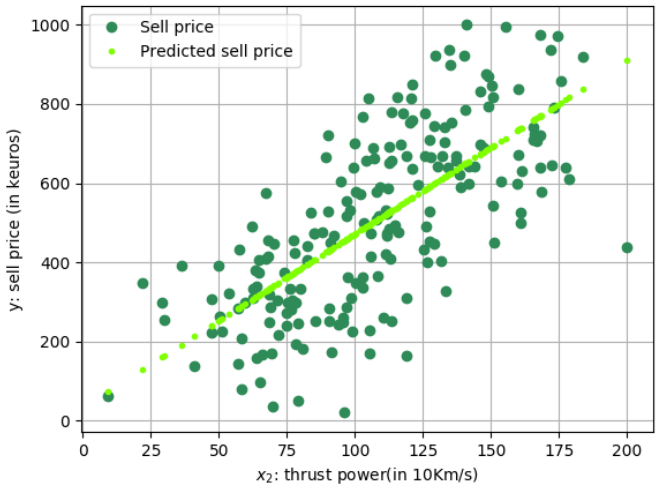



Figure VIII.2: Plot of the selling prices of spacecrafts with respect to the thrust power of their engines, as well as our second model's price predictions.

### 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 $(x^{(i)}_{distance}, y^{(i)})$ for $i = 0....m$,
- The predicted prices, represented by $(x^{(i)}_{distance}, \hat{y}^{(i)})$ for $i = 0....m$ (see example below),

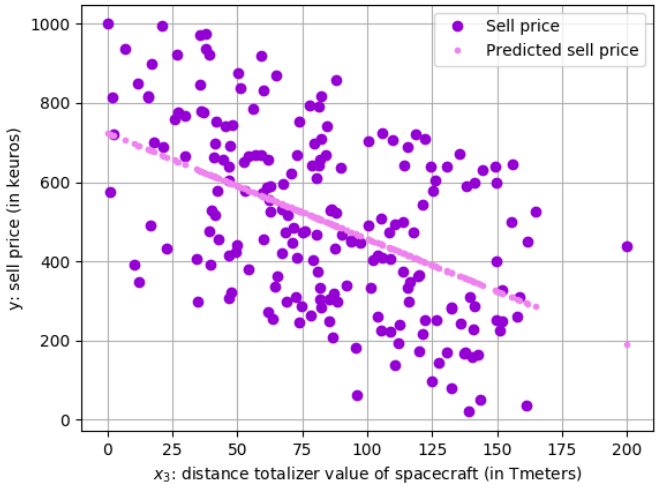



Figure VIII.3: Plot of the selling prices of spacecrafts with respect to the terameters driven, as well as our third model's price predictions.

### Reminder

- After executing the `fit_` method, you may obtain $\theta = array([[”nan”, ”nan”]])$. 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 have allowed 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.

`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(thetas = [[1000.0], [-1.0]], alpha = 2.5e-5, max_iter = 100000)
myLR_age.fit_(X, Y)

y_pred = myLR_age.predict_(X)
myLR_age.mse_(Y, y_pred)
#Output
55736.867198...
```

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!

### Instructions

Here, you will train a single model that will take all features into account. Your program is expected to perform steps similar to the ones in the part one (fitting, displaying or generating 3 graphs, printing the thetas and the MSE).

### 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?

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(thetas=[1.0, 1.0, 1.0, 1.0], alpha=9e-5, max_iter=500000)


# Example 0:
my_lreg.mse_(Y, my_lreg.predict_(X))
# Output:
144044.877...


# Example 1:
my_lreg.fit_(X, Y)
my_lreg.thetas
# Output:
array([[367.28849...]
       [-23.69939...]
       [  5.73622...]
       [ -2.63855...]])


# Example 2:
print(my_lreg.mse_(Y, my_lreg.predict_(X)))
# Output:
435.9325695...
```

### 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)
- 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)
- 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)

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?

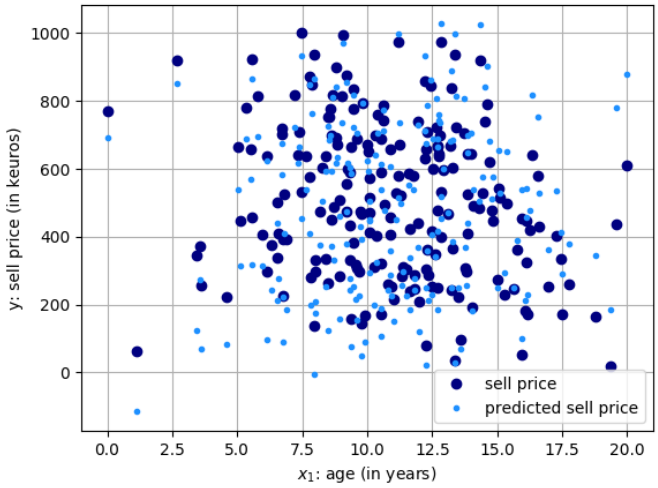



Figure VIII.4: Spacecraft sell prices of and predicted sell prices with the multivariate hypothesis, with respect to the *age* feature

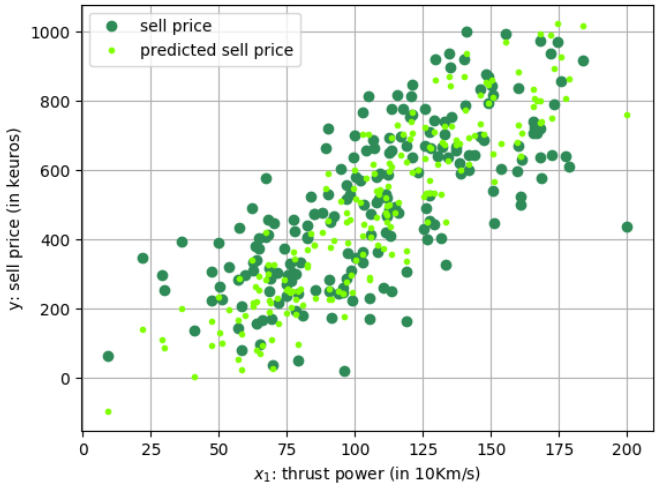



Figure VIII.5: Spacecraft sell prices predicted sell prices with the multivariate hypothesis, with respect to the thrust power of the engines

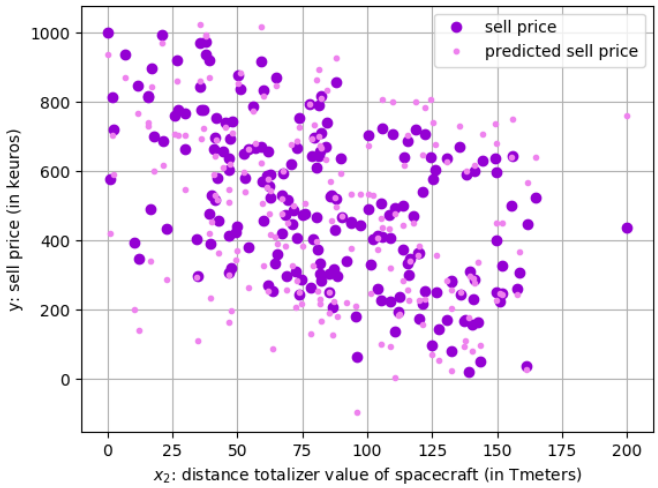



Figure VIII.6: Spacecraft sell prices and predicted sell prices with the multivariage hypothesis, with respect to the driven distance (in terameters)

# Chapter IX

# Exercise 07

## 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 do not have a linear relationship with the predicted variable, like in the figure below:

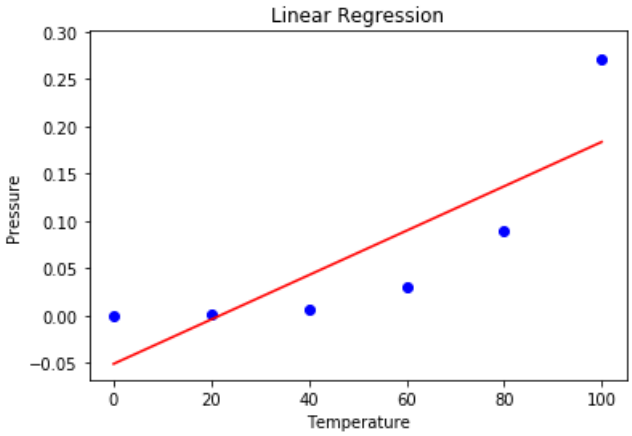



Figure IX.1: Non-linear relationship

In that case, we are stuck with a straight line that cannot 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!

## Interlude - 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.

## Interlude - 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:

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

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 : 07 |
|---|---|
| Polynomial models | |
| Turn-in directory : $ex07/$ | |
| Files to turn in : `polynomial_model.py` | |
| Forbidden functions : `sklearn` | |

# Objective

Broaden the comprehension of the notion of hypothesis. Create a function that takes a vector $x$ of dimension $m$ and an integer $n$ as input, and returns a matrix of dimensions $(m \times n)$. Each column of the matrix contains $x$ raised to the power of $j$, for $j = 1, 2, ..., n$:

$$x \mid x^2 \mid x^3 \mid \dots \mid x^n$$

Such a matrix is called a **Vandermonde matrix**.

# 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 an numpy.array, 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.
    Return:
      The matrix of polynomial features as a numpy.array, of dimension m * n,
        containing the polynomial feature values for all training examples.
      None if x is an empty numpy.array.
      None if x or power is not of expected type.
    Raises:
      This function should not raise any Exception.
    """
    ... Your code ...
```

# Examples

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


# Example 0:
add_polynomial_features(x, 3)
# Output:
array([[  1,   1,   1],
       [  2,   4,   8],
       [  3,   9,  27],
       [  4,  16,  64],
       [  5,  25, 125]])


# Example 1:
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]])
```

# Chapter X

# Exercise 08

## Interlude - Plotting Curves With Matplotlib

We asked you to plot straight lines in the `module05`. 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()
```

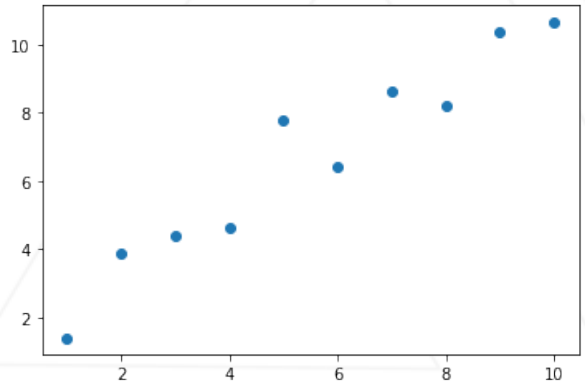



Figure X.1: Scatter plot of a dataset

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()
```

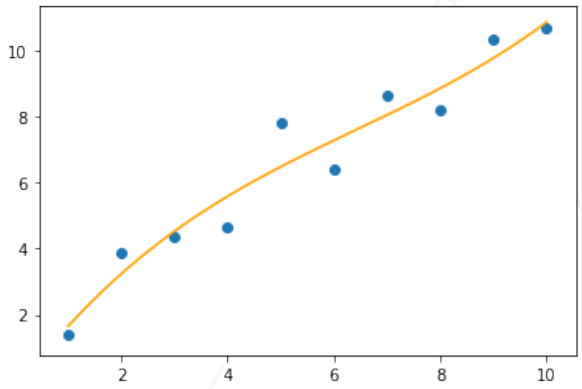



Figure X.2: Scatter plot of a dataset, and on top, a plot of the polynomial hypothesis function

| | Exercise : 08 |
|---|---|
| Let's Train Polynomial Models! | |
| Turn-in directory : $ex08/$ | |
| Files to turn in : `polynomial_train.py` | |
| Forbidden functions : `sklearn` | |

# Objective

Manipulation of polynomial hypothesis. It's training time! Let's train some polynomial models, and see if those with higher polynomial degree perform better!

Write a program which:

- Reads and loads `are_blue_pills_magics.csv` dataset,
- Trains **six** separate Linear Regression models with polynomial hypothesis with degrees ranging from 1 to 6,
- Evaluates and prints evaluation score (MSE) of each of the six models,
- Plots a bar plot showing the MSE score of the models in function of the polynomial degree of the hypothesis,
- Plots the 6 models and the data points on the same figure. Use lineplot style for the models and scaterplot for the data points. Add more prediction points to have smooth curves for the models.

You will use `Micrograms` as feature and `Score` as target. The implementation of the method `fit_` based on the simple gradient descent lakes of efficiency and sturdiness, which will lead to the impossibility of converging for polynomial models with high degree or with features having several orders of magnitude of difference. See the starting values for some thetas below to help you to get acceptable parameters values for the models.

According to evaluation score only, what is the best hypothesis (or model) between the trained models? According to the last plot, why it is not true? Which phenomenom do you observed here?

## Starting points

You will not be able to get acceptable parameters for models 4, 5 and 6. Thus you can start the fit process for those models with:

```
theta4 = np.array([[-20],[ 160],[ -80],[ 10],[ -1]]).reshape(-1,1)
theta5 = np.array([[1140],[ -1850],[ 1110],[ -305],[ 40],[ -2]]).reshape(-1,1)
theta6 = np.array([[9110],[ -18015],[ 13400],[ -4935],[ 966],[ -96.4],[ 3.86]]).reshape(-1,1)
```

## Teminology 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.

## Remark

When you will be evaluated, it will be wised to run your program at the beginning of the evaluation as it can take several minutes to train the different models.

# Chapter XI

# Exercise 09

## 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.

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

Take a look at the following plot.

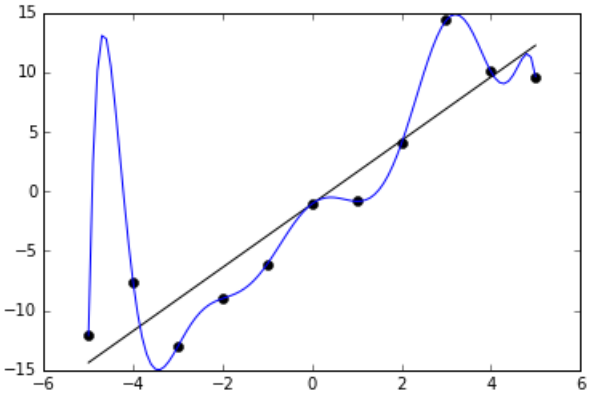



Figure XI.1: Overfitting hypothesis

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.

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

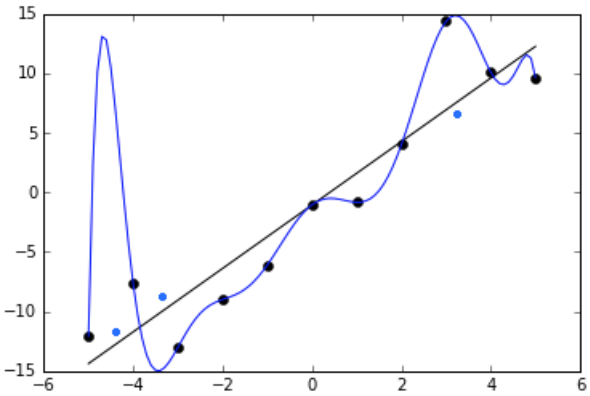



Figure XI.2: Generalization errors resulting from overfitting

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.

## Interlude - 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 *recalling*. 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 : 09 |
|---|---|
| DataSpliter | |
| Turn-in directory : $ex09/$ | |
| Files to turn in : `data_spliter.py` | |
| Forbidden functions : `sklearn` | |

# Objective

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

```
def data_spliter(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 an numpy.array, a matrix of dimension m * n.
      y: has to be an numpy.array, 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.
    Return:
      (x_train, x_test, y_train, y_test) as a tuple of numpy.array
      None if x or y is an empty numpy.array.
      None if x and y do not share compatible dimensions.
      None if x, y or proportion is not of expected type.
    Raises:
      This function should not raise any Exception.
    """
    ... Your code ...
```

- 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]).reshape((-1, 1))
y = np.array([0, 1, 0, 1, 0]).reshape((-1, 1))

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

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

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

# Example 3:
data_spliter(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_spliter(x2, y, 0.5)
# Output:
(array([[59,  1],
        [10, 42]]),
 array([[300,  10],
        [300,  59],
        [  1,  42]]),
 array([0, 0]),
 array([1, 1, 0]))
```

# Chapter XII

# Exercise 10

| | Exercise : 10 |
|---|---|
| Machine Learning for Grown-ups: Trantor guacamole business | |
| Turn-in directory : $ex10/$ | |
| Files to turn in : `space_avocado.py, benchmark_train.py, models.[csv/yml/pickle]` | |
| Forbidden functions : `sklearn` | |

## Objective

Let's do Machine Learning for "real"!

## Introduction

The dataset is constituted of 5 columns:

- **index**: not relevant,
- **weight**: the avocado weight order (in ton),
- **prod_distance**: distance from where the avocado ordered is produced (in Mkm),
- **time_delivery**: time between the order and the receipt (in days),
- **target**: price of the order (in trantorian unit).

It contains the data of all the avocado purchase made by Trantor administration (guacamole is a serious business there).

## Instructions

You have to explore different models and select the best you find. To do this:

- Split your `space_avocado.csv` dataset into a training and a test set.
- Use your `polynomial_features` method on your training set.
- Consider several Linear Regression models with polynomial hypothesis with a maximum degree of 4.
- Evaluate your models on the test set.

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

- Plot the evaluation curve which help you to select the best model (evaluation metrics vs models).
- Plot the true price and the predicted price obtain via your best model (3D representation or 3 scatterplots).

The training of all your models can take a long time. Thus you need to train only the best one during the correction. But, you should return in `benchmark_train.py` the program which perform the training of all the models and save the parameters of the different models into a file. In `models.[csv/yml/pickle]` one must find the parameters of all the models you have explored and trained. In `space_avocado.py` train the model based on the best hypothesis you find and load the other models from `models.[csv/yml/pickle]`. Then evaluate and plot the different graphics as asked before.

# Chapter XIII

# Conclusion - What you have learnt

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

1. What is the main (obvious) difference between univariate and multivariate linear regression?

2. Is there a minimum number of variables needed to perform a multivariate linear regression?

3. Is there a maximum number of variables needed to perform a multivariate linear regression? In theory and in practice?

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?

6. Can you explain what is overfitting?

7. Can you explain what is underfitting?

8. Why is it important to split the data set in a training and a test set?

9. If a model overfits, what will happen when you compare its performance on the training set and the test set?

10. If a model underfits, what do you think will happen when you compare its performance on the training set and the test set?

## Contact

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

Thank you for attending 42AI's Machine Learning Bootcamp module02 !

## Acknowledgements

The Python & ML bootcamps are the result of a collective effort. We would like to thank:

- Maxime Choulika (cmaxime),
- Pierre Peigné (ppeigne),
- Matthieu David (mdavid),
- Quentin Feuillade–Montixi (qfeuilla, quentin@42ai.fr)
- Mathieu Perez (maperez, mathieu.perez@42ai.fr)

who supervised the creation and enhancements and the present transcription.

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

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

- All prior participants who took a moment to provide their feedbacks, and help us improve these bootcamps !