Bootcamp
Machine Learning

Module06
Univariate Linear Regression

Module06 - Univariate Linear Regression

Today you will implement a method to improve your model's performance: **gradient descent**. Then you will discover the notion of normalization.

Notions of the module

Gradient descent, linear regression, normalization.

Useful Resources

You are strongly advise to use the following resources: Machine Learning MOOC - Stanford

Week 1:

Linear Regression with One Variable:

- Gradient Descent (Video + Reading)
- Gradient Descent Intuition (Video + Reading)
- Gradient Descent For Linear Regression (Video + Reading)
- Review (Reading + Quiz)

Week 2:

Multivariate Linear Regression

- Gradient Descent in Practice 1 - Feature Scaling (Video + Reading)

General Rules

- The version of Python to use is 3.7, you can check the version of Python 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 - Prediction

Turn-in directory :	ex00/
Files to turn in :	prediction.py
Forbidden functions :	None
Remarks :	n/a

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

Objective:

Understand and manipulate the notion of hypothesis in machine learning.

You must implement the following formula as a function:

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

Where:

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

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

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

Be careful:

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

You have to transform x to fit the dimension of θ !

Instructions:

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

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

Examples:

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

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


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


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


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

Exercise 01 - Vectorized Cost Function

Turn-in directory :	ex01/
Files to turn in :	cost.py
Forbidden functions :	None
Remarks :	n/a

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

Objective:

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

You must implement the following formula as a function:

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

Where:

- y is a vector of dimension m,
- $\hat{y}$ is a vector of dimension m.

Instructions:

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

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

Examples:

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

# Example 1:
cost_(X, Y)
# Output:
2.142857142857143
```

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

Exercise 02 - AI Key Notions

These questions highlight key notions from the previous module. 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 - When we pre-process the training examples, why are we adding a column of *ones* to the left of the x vector (or X matrix) when we use the linear algebra trick?

2 - Why does the cost function square the distance between the data points and their predicted values?

3 - What does the cost function value represent?

4 - Toward which value would you like the cost function to tend to? What would it mean?

Interlude - Improve

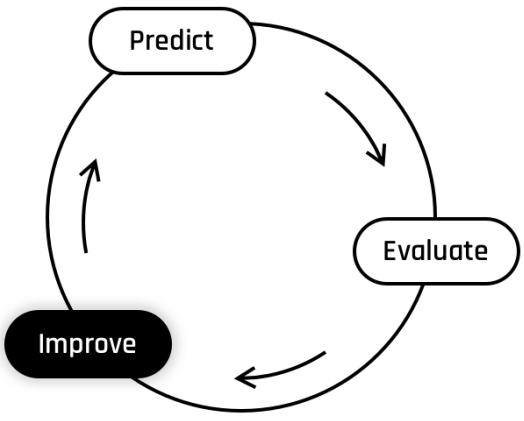

Figure 1: The Learning Cycle - Improve

Yesterday, you discovered the first two steps of the learning process: starting with a model that makes naive predictions and evaluating it. Now we are going to tackle the third part: improving it!

Lets take a new dataset:

Given our measure of performance, improvement entails **reducing the cost (or loss)** measured by the cost function. If we plot the cost of a model's predictions as a function of its θ_1 parameter (with a fixed value for θ_0),

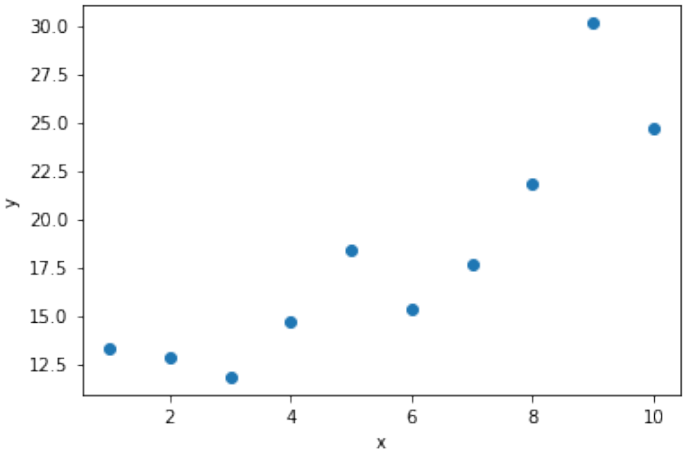

Figure 2: Scatter plot of a given dataset

we obtain a curve like this one:

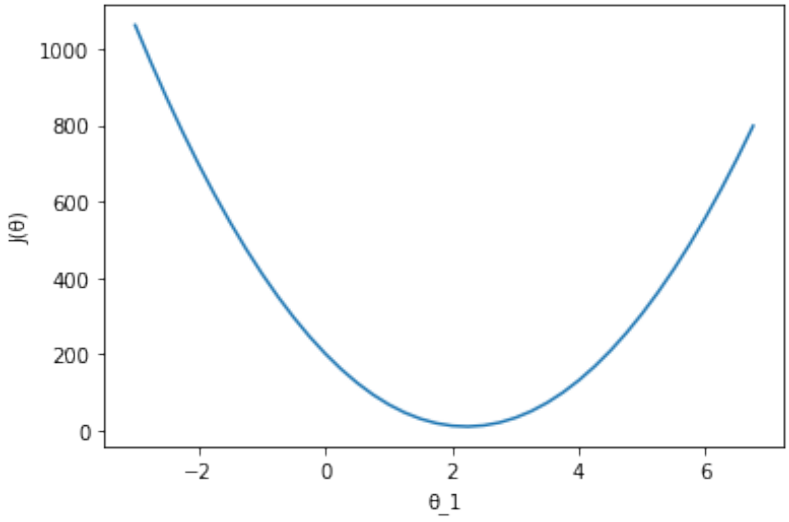

Figure 3: Cost function given theta_1

On the graphs below, you can see the we can see that extreme θ_1 values (which modifies the slope of the hypothesis curve - in orange) correspond to a very high cost. On the other hand, as we get closer to the bottom of the curve, the cost is reduced.

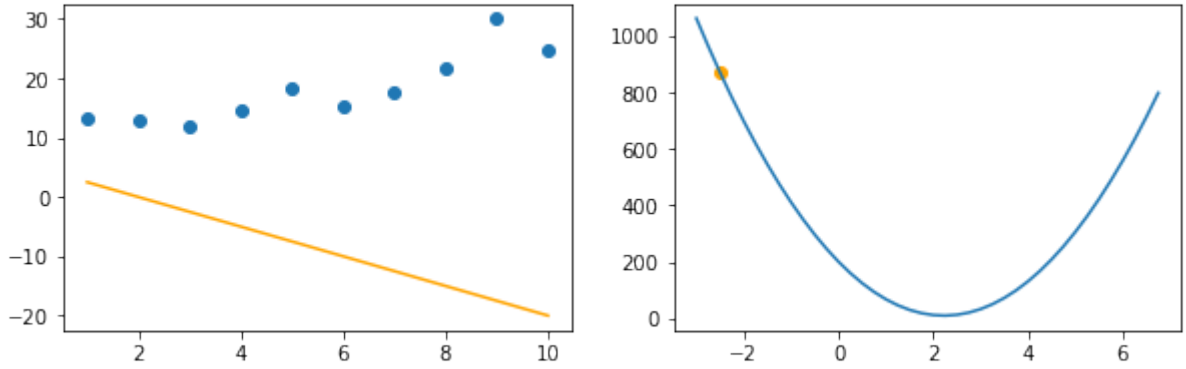

Figure 4: A quite bad model

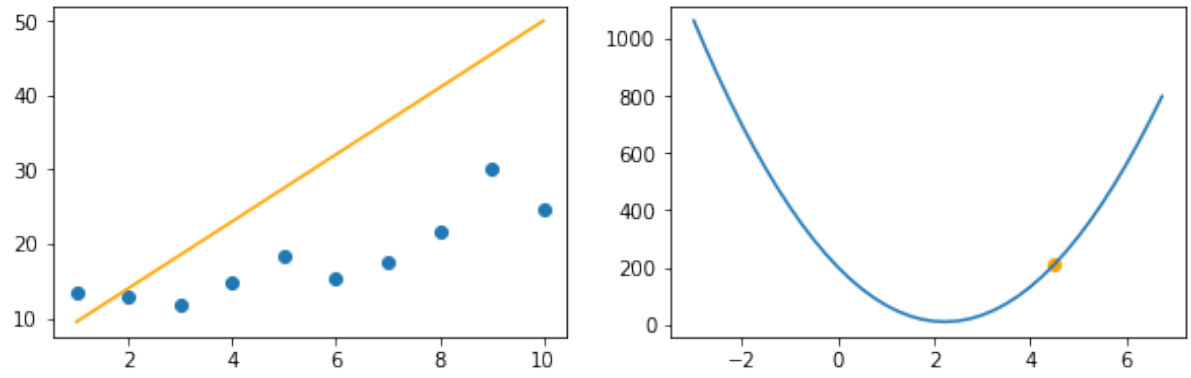

Figure 5: A better (but still bad) model

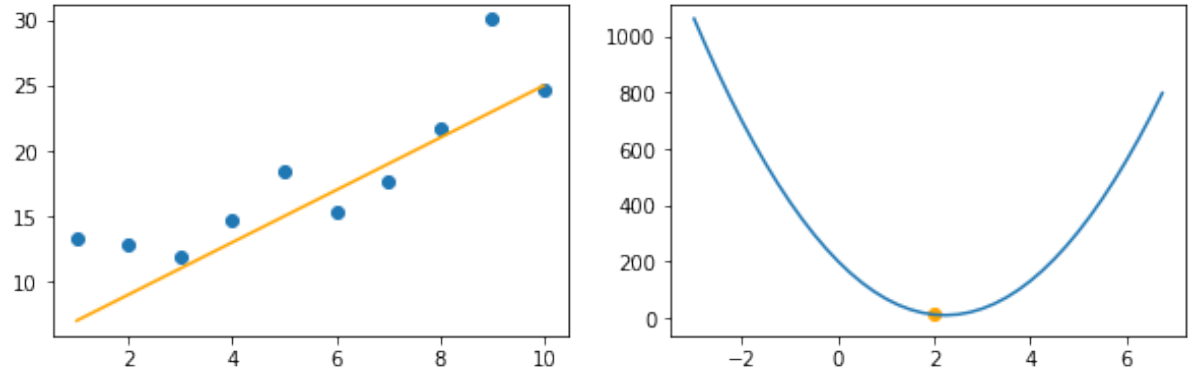

Figure 6: A good model

The cost function's minimum corresponds to the bottom of the curve. We want θ_1 to get to this sweet spot. It means that wherever θ_1 starts at, as the training goes on, it needs to get closer to the value that matches $J(\theta)$'s minimum.

But how to get closer to the minimum?

Excellent question dear reader. We're glad you asked!
First, the algorithm needs to figure out in what direction θ_1 should be moved (i.e. increased or decreased). It does so by calculating the ***slope*** of the $J(\theta)$ curve at the current position of θ_1. If the slope is positive, θ_1 must be decreased. If the slope is negative, it must be increased. If you have studied calculus, you probably sense that all of this involves calculating the derivative of the cost function.

The story gets a little more complicated, however, because we have two parameters to adjust: θ_0 and θ_1. Not just θ_1 (as we showed in our example to simplify). This means the $J(\theta)$ function doesn't have one derivative, but two ***partial derivatives***. One that computes the slope of J with respect to θ_0, and a second one for the slope of J with respect to θ_1. Finally, we package those partial derivatives in a vector of dimension 2×1, which is called ***gradient*** (noted ∇).

Don't worry if you don't master multivariate calculus yet, we have calculated the partial derivatives for you, all you will need to do is write them in Python.

Exercise 03 - Linear Gradient - Iterative Version

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

Objective:

Understand and manipulate the notion of gradient and gradient descent in machine learning.

You must write a function that computes the ***gradient*** of the cost function.
It must compute a partial derivative with respect to each theta parameter separately, and return the gradient vector.
The partial derivatives can be calculated with the following formulas:

$$\nabla(J)_0 = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})$$

$$\nabla(J)_1 = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x^{(i)}$$

Where:

- $\nabla(J)$ is the gradient vector of size 2×1, (this strange symbol : ∇ is called nabla)
- x is a vector of dimension m,
- y is a vector of dimension m,
- $x^{(i)}$ is the i^{th} component of vector x,
- $y^{(i)}$ is the i^{th} component of vector y,
- $\nabla(J)_j$ is the j^{th} component of $\nabla(J)$,
- $h_\theta(x^{(i)})$ corresponds to the model's prediction of $y^{(i)}$.

Hypothesis Notation:

$h_\theta(x^{(i)})$ is the same as what we previously noted $\hat{y}^{(i)}$.
The two notations are equivalent. They represent the model's prediction (or estimation) of the $y^{(i)}$ value. If you follow Andrew Ng's course material on Coursera, you will see him using the former notation.

As a reminder : $h_\theta(x^{(i)}) = \theta_0 + \theta_1 x^{(i)}$

Instructions:

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

```
def simple_gradient(x, y, theta):
    """Computes a gradient vector from three non-empty numpy.ndarray, without any for-loop.
    ↪  The three arrays must have compatible dimensions.
    Args:
      x: has to be an numpy.ndarray, a vector of dimension m * 1.
      y: has to be an numpy.ndarray, a vector of dimension m * 1.
      theta: has to be an numpy.ndarray, a 2 * 1 vector.
    Returns:
      The gradient as a numpy.ndarray, a vector of dimension 2 * 1.
      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.
```

```
    """
    ... Your code ...
```

Examples:

```
import numpy as np
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])

# Example 0:
theta1 = np.array([2, 0.7])
simple_gradient(x, y, theta1)
# Output:
array([-19.0342574, -586.66875564])

# Example 1:
theta2 = np.array([1, -0.4])
simple_gradient(x, y, theta2)
# Output:
array([-57.86823748, -2230.12297889])
```

Interlude - Linear Algebra Tricks II

If you tried to run your code on a very large dataset, you'd find that it takes a long time to execute! That's because it doesn't use the power of Python libraries that are optimized for matrix operations.

Remember the linear algebra trick of yesterday? Let's use it again!
If you concatenate a column of 1's to the left of the x vector, you get what we called matrix X'.

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

This transformation is very convenient because we can rewrite each 1 as $x_0^{(i)}$, and each $x^{(i)}$ as $x_1^{(i)}$. So now the X' matrix looks like this:

$$X' = \begin{bmatrix} x_0^{(1)} & x_1^{(1)} \\ \vdots & \vdots \\ x_0^{(m)} & x_1^{(m)} \end{bmatrix}$$

Notice that each $x^{(i)}$ example becomes a vector made of $(x_0^{(i)}, x_1^{(i)})$.
The 0 and 1 indices on the x features correspond to the indices of the θ parameters with which they will be multiplied.

Why does this matter? Well, if we take the equation from the previous exercise:

$$\nabla(J)_0 = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})$$

We can multiply it by 1 without changing its value:

$$\nabla(J)_0 = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot 1$$

And rewrite 1 as $x_0^{(i)}$:

$$\nabla(J)_0 = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)}$$

This means that now the equation for $\nabla(J)_0$ is no different from the equation we had for $\nabla(J)_1$, so they can both be captured by ONE **generic equation**:

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

And as you probably suspected, a generic equation opens the door to vectorization...

Vectorizing the Gradient Calculation

Now it's time to learn how to calculate the entire gradient in one short, pretty, linear algebra equation!

- First, we'll use the X' matrix and our vectorized hypothesis equation: $h_\theta(x) = X'\theta$

$$\nabla(J)_j = \frac{1}{m}(X'\theta - y)X'_j \qquad \text{for j = 0, 1}$$

- Second, we need to tweak the equation a bit so that it directly returns a $\nabla(J)$ vector containing both $\nabla(J)_0$ and $\nabla(J)_1$.

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

If the equation does not seems obvious, play a bit with your vectors, on paper and in your code, until you get it.

Notation Remark:

X'^{T} : You might wonder what the ^{T} is for. It means the X' matrix is **transposed**. Transposing a matrix flips it on its diagonal so that its rows become its columns and vice versa. Here we need to do it so that matrix dimensions are appropriate multiplication and to multiply the right elements together.

Exercise 04 - Linear Gradient - Vectorized Version

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

Objective:

Understand and manipulate the notion of gradient and gradient descent in machine learning.

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 2×1.
- X' is a **matrix** of dimension $m \times 2$.
- X'^T is the transpose of X'. Its dimensions are $2 \times m$.
- y is a vector of dimension m.
- θ is a vector of dimension 2×1.

Be careful:

- the x you will get as an input is an m vector.
- θ is a 2×1 vector. You have to transform x to fit the dimension of θ!

Instructions:

You have to code the following function as per the instructions given below in gradient.py:

```
def gradient(x, y, theta):
    """Computes a gradient vector from three non-empty numpy.ndarray, without any for loop.
    ↪  The three arrays must have compatible dimensions.
    Args:
      x: has to be a numpy.ndarray, a matrix of dimension m * 1.
      y: has to be a numpy.ndarray, a vector of dimension m * 1.
      theta: has to be a numpy.ndarray, a 2 * 1 vector.
    Returns:
      The gradient as a numpy.ndarray, a vector of dimension 2 * 1.
      None if x, y, or theta is an 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([12.4956442, 21.5007972, 31.5527382, 48.9145838, 57.5088733])
y = np.array([37.4013816, 36.1473236, 45.7655287, 46.6793434, 59.5585554])

# Example 0:
theta1 = np.array([2, 0.7])
gradient(x, y, theta1)
# Output:
array([-19.0342574, -586.66875564])

# Example 1:
theta2 = np.array([1, -0.4])
gradient(x, y, theta2)
# Output:
array([-57.86823748, -2230.12297889])
```

Interlude - Gradient Descent

So far we've calculated the *gradient*, which indicates whether and by how much we should increase or decrease θ_0 and θ_1 in order to reduce the cost.
What we have to do next is update the theta parameters accordingly, step by step, until we reach the minimum. This iterative process, called **Gradient Descent**, will progressively improve the performance of your regression model on the training data.

The gradient descent **algorithm** can be summarized like this: for a certain number of cycles, at each step, both θ parameters are slightly moved in the opposite directions than what the gradient indicates.

The algorithm can be expressed in pseudocode as the following:

repeat until convergence: $\{$**compute**$\nabla(J) \quad \theta_0 \leftarrow \theta_0 - \alpha\nabla(J)_0 \quad \theta_1 \leftarrow \theta_1 - \alpha\nabla(J)_1\}$

A few remarks on this algorithm:

- If you directly subtracted the gradient from θ, your steps would be too big and you would quickly overshoot past the minimum. That's why we use α (alpha), called the *learning rate*. It's a small float number (usually between 0 and 1) that decreases the magnitude of each update.
- The pseudocode says "repeat until convergence", but in your implementation, you will not actually check for convergence at each iteration. You will instead set a number of cycles that is sufficient for your gradient descent to converge.
- When training a linear regression model on a new dataset, you will have to choose appropriate alpha and the number of cycles through trial and error.

Exercise 05 - Gradient Descent

Turn-in directory :	ex05/
Files to turn in :	fit.py
Authorized modules :	numpy
Forbidden functions :	any function that calculates derivatives for you

Objective:

Understand and manipulate the notion of the gradient and gradient descent in machine learning. Be able to explain what it means to ***fit*** a Machine Learning model to a dataset Implement a function that performs **Linear Gradient Descent** (LGD).

Instructions:

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

The pseudocode for the algorithm is the following:

repeat until convergence: $\{$**compute**$\nabla(J) \quad \theta_0 \leftarrow \theta_0 - \alpha\nabla(J)_0 \quad \theta_1 \leftarrow \theta_1 - \alpha\nabla(J)_1\}$

Where:

- α (alpha) is the *learning rate*. It's a small float number (usually between 0 and 1),
- For now, "reapeat until convergence" will mean to simply repeat for max_iter (a number that you will choose wisely).

You are expected to code a function named `fit_` as per the instructions below:

```
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.ndarray, a vector of dimension m * 1: (number of training
↪   examples, 1).
        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 2 * 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
    Returns:
        new_theta: numpy.ndarray, a vector of dimension 2 * 1.
        None if there is a matching dimension problem.
    Raises:
        This function should not raise any Exception.
    """
        ... your code here ...
```

Hopefully, you have already written a function to calculate the linear gradient.

Examples:

```
import numpy as np
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]])
theta= np.array([1, 1])

# Example 0:
theta1 = fit_(x, y, theta, alpha=5e-8, max_iter=1500000)
theta1
# Output:
array([[1.40709365],
       [1.1150909 ]])

# Example 1:
predict(x, theta1)
# Output:
array([[15.3408728 ],
       [25.38243697],
       [36.59126492],
       [55.95130097],
       [65.53471499]])
```

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 θ coefficients that first you used.
- It is possible that θ_0 and θ_1 become "[nan]". In that case, it means you probably used a learning rate that is too large.

Exercise 06 - Linear Regression with Class

Turn-in directory :	ex06/
Files to turn in :	my_linear_regression.py
Authorized modules :	numpy
Forbidden modules :	sklearn

Objective:

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 class available in Scikit-learn: `sklearn.linear_model.LinearRegression`

```
class MyLinearRegression():
    """
    Description:
        My personnal linear regression class to fit like a boss.
    """
    def __init__(self,  thetas, alpha=0.001, max_iter=1000):
            self.alpha = alpha
            self.max_iter = max_iter
            self.thetas = thetas

    #... other methods ...
```

You will add the following methods:

- `fit_(self, x, y)`
- `predict_(self, x)`
- `cost_elem_(y, y_hat)`
- `cost_(y, y_hat)`.

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:
MyLR.cost_elem_(y, lr1.predict(x))
# Output:
array([[710.45867381],
       [364.68645485],
       [469.96221651],
       [108.97553412],
       [299.37111101]])

# Example 0.2:
MyLR.cost_(y, lr1.predict(x))
# Output:
195.34539903032385

# Example 1.0:
lr2 = MyLR([1, 1], 5e-8, 1500000)
lr2.fit_(x, y)
lr2.thetas
# 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:
MyLR.cost_elem_(y, lr2.predict(x))
# Output:
array([[486.66604863],
       [115.88278416],
       [ 84.16711596],
       [ 85.96919719],
       [ 35.71448348]])

# Example 1.3:
MyLR.cost_(y, lr2.predict(x))
# Output:
80.83996294128525
```

Exercise 07 - Practicing Linear Regression

Turn-in directory :	ex07/
Files to turn in :	linear_model.py

Authorized modules :	numpy, matplotlib
Forbidden modules :	sklearn
Remarks :	Read the doc

Objective:

Evaluate a linear regression model on a very small dataset, with a given hypothesis function h. Manipulate the cost function J, plot it, and briefly analyze the plot.

Instructions:

You can find in the `resources` folder a tiny dataset called `are_blue_pills_magic.csv` which gives you the driving performance of space pilots as a function of the quantity of the "blue pills" they took before the test. You have a description of the data in the file named `are_blue_pills_magic.txt`.
As your hypothesis function h, you will choose:

$$h_\theta(x) = \theta_0 + \theta_1 x$$

Where x is the variable, and θ_0 and θ_1 are the coefficients of the hypothesis. The hypothesis is a function of x.

You are strongly encouraged to use the class you have implement in the previous exercise.

Your program must:

- Read the dataset from the csv file,
- perform a linear regression,

Then you will model the data and plot 2 different graphs:

- A graph with the data and the hypothesis you get for the spacecraft piloting score versus the quantity of "blue pills" (see example below)

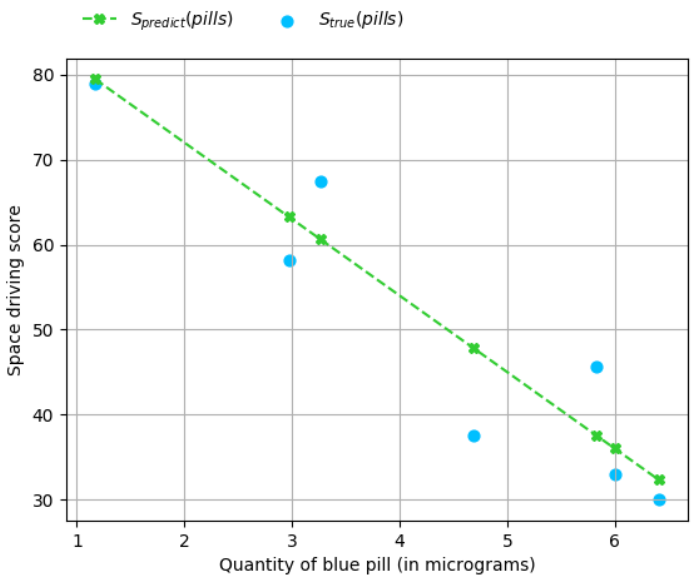

Figure 7: Space driving score as a function of the quantity of blue pill (in micrograms). In blue the real values and in green the predicted values.

- The cost function $J(\theta)$ in function of the θ values (see example below),
- Your program will also calculate and display the MSE associated to the model (you know how to do it already).

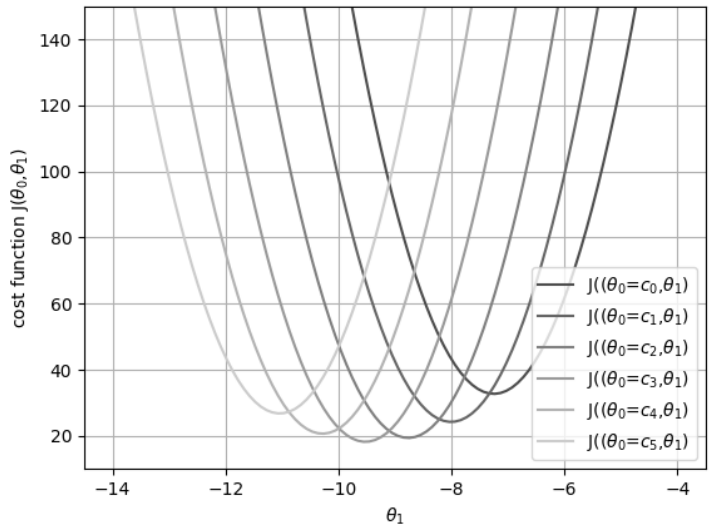

Figure 8: Evolution of the cost function J as a function of θ_1 for different values of θ_0.

Examples:

```
import pandas as pd
import numpy as np
from sklearn.metrics import mean_squared_error
from mylinearregression import MyLinearRegression as MyLR

data = pd.read_csv("are_blue_pills_magic.csv")
Xpill = np.array(data[Micrograms]).reshape(-1,1)
Yscore = np.array(data[Score]).reshape(-1,1)

linear_model1 = MyLR(np.array([[89.0], [-8]]))
linear_model2 = MyLR(np.array([[89.0], [-6]]))
Y_model1 = linear_model1.predict_(Xpill)
Y_model2 = linear_model2.predict_(Xpill)

>>>print(MyLR.mse_(Yscore, Y_model1))
# 57.60304285714282
>>>print(mean_squared_error(Yscore, Y_model1))
# 57.603042857142825
>>>print(MyLR.mse_(Yscore, Y_model2))
# 232.16344285714285
>>>print(mean_squared_error(Yscore, Y_model2))
# 232.16344285714285
```

Here, the use of scikit learn is to ensure that our code is performing as expected. The use of scikit learn is forbidden in the code you will turn-in.

Clarification and Hints:

There is no method named `.mse_` in sklearn's LinearRegression class, but there is a method named `.score`. The `.score` method corresponds to the R^2 score. The metric MSE is available in the `sklearn.metrics` module.

Exercise 8 - Question Time!

Are you able to clearly explain:

1 - What is a hypothesis and what is its goal?

2 - What is the 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 the theta parameters...)

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 - Normalization

The values inside the x vector can vary quite a lot in magnitude, depending on the type of data you are working with. For example, if your dataset contains distances between planets in km, the numbers will be huge. On the other hand, if you are working with planet masses expressed as a fraction of the solar system's total mass, the numbers will be very small (between 0 and 1)
Both cases may slow down convergence in Gradient Descent (or even sometimes prevent convergence at all). To avoid that kind of situation, normalization is a very effective way to proceed.

The idea behind this technique is straightforward: **scaling the data**.

With normalization, you can transform your x vector into a new x' vector whose values range between $[-1, 1]$ more or less. Doing this allows you to see much more easily how a training example compares to the other ones:

- If an x' value is close to 1, you know it's among the largest in the dataset
- If an x' value is close to 0, you know it's average
- If an x' value is close to -1, you know it's among the smallest.

So with the upcoming normalization techniques, you'll be able to map your data to two different value ranges: $[0, 1]$ or $[-1, 1]$. Your algorithm will like it and thank you for it.

Exercise 09 - Normalization I: Z-score Standardization

Turn-in directory :	ex09/
Files to turn in :	z-score.py
Forbidden functions :	None
Remarks :	n/a

Objective:

Introduction to standardization/normalization methods. You must implement the following formula as a function:

$$x'^{(i)} = \frac{x^{(i)} - \frac{1}{m}\sum_{i=1}^{m} x^{(i)}}{\sqrt{\frac{1}{m-1}\sum_{i=1}^{m}(x^{(i)} - \frac{1}{m}\sum_{i=1}^{m} x^{(i)})^2}} \qquad \text{for } i \text{ in } 1, ..., m$$

Where:

- x is a vector of dimension m,
- $x^{(i)}$ is the i^{th} component of the x vector,
- x' is the normalized version of the x vector.

The equation is much easier to understand in the following form:

$$x'^{(i)} = \frac{x^{(i)} - \mu}{\sigma} \qquad \text{for } i \text{ in } 1, ..., m$$

This should remind you something from **TinyStatistician**...

Nope?

Ok let's do a quick recap:

- μ is the mean of x,
- σ is the standard deviation of x.

Instructions:

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

```
def zscore(x):
    """Computes the normalized version of a non-empty numpy.ndarray using the z-score
    ↪  standardization.
    Args:
      x: has to be an numpy.ndarray, a vector.
    Returns:
      x' as a numpy.ndarray.
      None if x is a non-empty numpy.ndarray or not a numpy.ndarray.
    Raises:
      This function shouldn't raise any Exception.
    """
    ... Your code ...
```

Examples:

```
# Example 1:
X = numpy.array([0, 15, -9, 7, 12, 3, -21])
zscore(X)
# Output:
array([-0.08620324,  1.2068453 , -0.86203236,  0.51721942,  0.94823559,
        0.17240647, -1.89647119])

# Example 2:
Y = np.array([2, 14, -13, 5, 12, 4, -19])
zscore(Y)
# Output:
array([ 0.11267619,  1.16432067, -1.20187941,  0.37558731,  0.98904659,
        0.28795027, -1.72770165])
```

Exercise 10 - Normalization II: Min-max Standardization

Turn-in directory :	ex10/
Files to turn in :	minmax.py
Forbidden functions :	None
Remarks :	n/a

Objective:

Introduction to standardization/normalization methods. Implement another normalization method.

You must implement the following formula as a function:

$$x'^{(i)} = \frac{x^{(i)} - min(x)}{max(x) - min(x)} \qquad \text{for } i = 1, ..., m$$

Where:

- x is a vector of dimension m,
- $x^{(i)}$ is the i^{th} component of vector x,
- $min(x)$ is the minimum value found among the components of vector x,
- $max(x)$ is the maximum value found among the components of vector x.

You will notice that this min-max standardization doesn't scale the values to the $[-1, 1]$ range. What do you think the final range will be?

Instructions:

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

```
def minmax(x):
    """Computes the normalized version of a non-empty numpy.ndarray using the min-max
    ↪  standardization.
    Args:
      x: has to be an numpy.ndarray, a vector.
    Returns:
      x' as a numpy.ndarray.
      None if x is a non-empty numpy.ndarray or not a numpy.ndarray.
    Raises:
      This function shouldn't raise any Exception.
    """
```

Examples:

```
# Example 1:
X = np.array([0, 15, -9, 7, 12, 3, -21])
minmax(X)
# Output:
array([0.58333333, 1.        , 0.33333333, 0.77777778, 0.91666667,
       0.66666667, 0.        ])

# Example 2:
```

```
Y = np.array([2, 14, -13, 5, 12, 4, -19])
minmax(Y)
# Output:
array([0.63636364, 1.        , 0.18181818, 0.72727273, 0.93939394,
       0.6969697 , 0.        ])
```