Machine Learning - Module 03

Logistic Regression

Summary: Discover your first classification algorithm: logistic regression. You will learn its loss function, gradient descent and some metrics to evaluate its performance.

Notions and ressources

Notions of the module

Logistic hypothesis, logistic gradient descent, logistic regression, multiclass classification. Accuracy, precision, recall, F1-score, confusion matrix.

Useful Ressources

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

Week 3

Classification and representation

- Classification (Video + Reading)
- Hypothesis Representation (Video + Reading)
- Decision Boundary (Video + Reading)

Logistic Regression Model

- Cost Function (Video + Reading)
- Simplified Cost Function and Gradient Descent (Video + Reading)

Multiclass Classification

- Mutliclass Classification: One-vs-all (Video + Reading)
- Review (Reading + Quiz)

Common Instructions

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

Contents

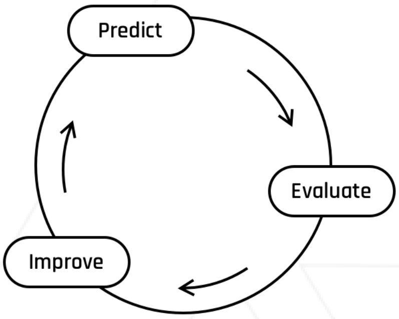
Chapter I

Exercise 00

Interlude

Classification: The Art of Labelling Things

Over the last three modules you have implemented your first machine learning algorithm. You also discovered the three main steps we follow when we build **learning algorithms**:



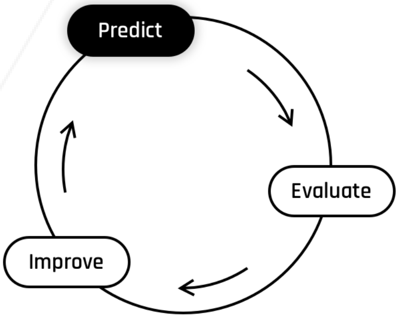
The first algorithm you discovered, **Multivariate Linear Regression**, can now be used to predict a numerical value, based on several features. This algorithm uses gradient descent to optimize its loss function.

Now let's introduce you to your first **classification algorithm**: it is named **Logistic Regression.** It peforms a *classification task*, which means that you are not predicting a numerical value (like price, age, grades...) but **categories**, or **labels** (like dog, cat, sick/healty...).

> ⚠ Don't be confused by the word *'regression'* in **Logistic Regression**. It really is a *classification task*! The name is a bit tricky but you will quickly get used to it.
> Once again: **Logistic Regression is a classification algorithm** which assigns a given example to a category.

> ⓘ In this module we will use the following terms interchangeably: **class**, **category**, and **label**. They all refer to the *groups* to which each training example can be assigned to, in a classification task.

Predict I: Introducing the Sigmoid Function



Formulating a Hypothesis

Remember that a hypothesis, denoted $h(\theta)$, is an equation that combines a set of **features** (that characterize an example) with **parameters** in order to output a **prediction**. Remember the hypothesis we used in linear regression?

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

It worked fine to predict continuous values, but could we also use it to tell, for example, if a patient is sick or not? That's a yes-or-no question, so the output from the hypothesis function should reflect that.

To get started, we'll assign each class a numerical value: sick patients will be assigned a value of 1, and healthy patients will be assigned a value of 0. The goal will be to build a hypothesis that outputs a probability that a patient is sick, as a float number within the range of 0 and 1.

The good news is that we can keep the linear equation we already worked with! All we need to do is squash its output through another function that is bounded between 0 and 1. That's the **Sigmoid function** and your next exercise is to implement it!

	Exercise : 00
Sigmoid	
Turn-in directory : *ex*00/	
Files to turn in : `sigmoid.py`	
Forbidden functions : `None`	

Objective

Introduction to the hypothesis in the case of logistic regression. You must implement the sigmoid function, given by the following formula:

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

Where:

- x is a scalar or a vector,
- e is the contracted form for exponential function. It is also a mathematical constant, named Euler's number.

This function is also known as **Standard logistic sigmoid function**. This explains the name *logistic regression*.

The sigmoid function transforms an input into a probability value, i.e. a value between 0 and 1. This probability value will then be used to classify the inputs.

Instructions

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

```
def sigmoid_(x):
    """
    Compute the sigmoid of a vector.
    Args:
        x: has to be a numpy.ndarray of shape (m, 1).
    Returns:
        The sigmoid value as a numpy.ndarray of shape (m, 1).
        None if x is an empty numpy.ndarray.
    Raises:
        This function should not raise any Exception.
    """
    ... Your code ...
```

Examples

```
# Example 1:
x = np.array([[-4]])
sigmoid_(x)
# Output:
array([[0.01798620996209156]])

# Example 2:
x = np.array([[2]])
sigmoid_(x)
# Output:
array([[0.8807970779778823]])

# Example 3:
x = np.array([[-4], [2], [0]])
sigmoid_(x)
# Output:
array([[0.01798620996209156], [0.8807970779778823], [0.5]])
```

Our sigmoid formula is a special case of the logistic function below, with $L = 1$, $k = 1$ and $x_0 = 0$:

$$f(x) = \frac{L}{1 + e^{-k(x-x_0)}}$$

Chapter II

Exercise 01

Interlude

Predict II : Hypothesis

We hope your curiosity led you to plot your sigmoid function. If you didn't, well here is what it looks like:

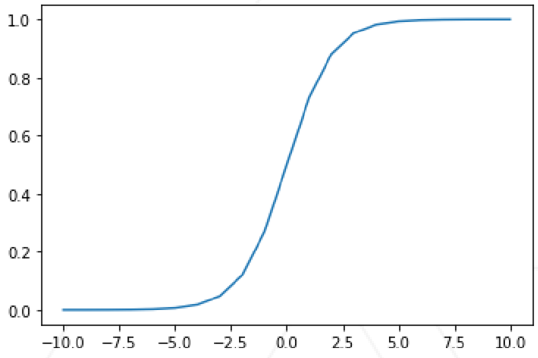


Figure II.1: Sigmoid

As you can see, **the sigmoid's output values range from** 0 **to** 1. You can input real numbers as big as you want (positive or negative), the output will always land within this range. This will be very helpful for the next part.

Logistic Hypothesis

Now you've written your sigmoid function, let's look at **the logistic regression hypothesis**.

$$\hat{y}^{(i)} \;=\; h_\theta(x^{(i)}) \;=\; \text{sigmoid}(\theta \cdot x'^{(i)}) \;=\; \frac{1}{1+e^{-\theta \cdot x'^{(i)}}} \qquad \text{for i = 1, \ldots, m}$$

This is simply the sigmoid function applied on top of the linear regression hypothesis!!

It can be vectorized as:

$$\hat{y} \;=\; h_\theta(X) \;=\; \text{sigmoid}(X'\theta) \;=\; \frac{1}{1+e^{-X'\theta}}$$

As we said before: the **sigmoid function** is just a way to **map the result of a linear equation onto a $[0,1]$ value range**.

This transformation allows us to interpret the result as a **probability that an individual is a member of a given class**.

Exercise : 01
Logistic Hypothesis
Turn-in directory : *ex*01/
Files to turn in : `log_pred.py`
Forbidden functions : `None`

Objective

Introduction to the hypothesis notion in case of logistic regression. You must implement the following formula as a function:

$$\hat{y} \;=\; \text{sigmoid}(X' \cdot \theta) \;=\; \frac{1}{1 + e^{-X' \cdot \theta}}$$

Where:

- 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 is added as a first column,
- $\hat{y}$ is a vector of dimension m, the vector of predicted values,
- θ is a vector of dimension $(n+1)$, the vector of parameters.

Be careful:

- the x your function will get as an input corresponds to X, the $(m \times n)$ matrix. Not X'.
- θ is an $(n+1)$ vector.

Instructions

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

```
def logistic_predict_(x, theta):
    """Computes the vector of prediction y_hat from two non-empty numpy.ndarray.
    Args:
      x: has to be an numpy.ndarray, a vector of dimension m * n.
      theta: has to be an 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.
    """
    ... Your code ...
```

Examples

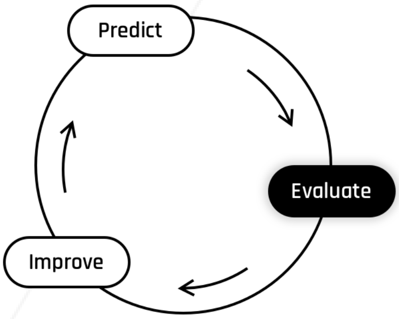
```
# Example 1
x = np.array([4]).reshape((-1, 1))
theta = np.array([[2], [0.5]])
logistic_predict_(x, theta)
# Output:
array([[0.98201379]])

# Example 1
x2 = np.array([[4], [7.16], [3.2], [9.37], [0.56]])
theta2 = np.array([[2], [0.5]])
logistic_predict_(x2, theta2)
# Output:
array([[0.98201379],
       [0.99624161],
       [0.97340301],
       [0.99875204],
       [0.90720705]])

# Example 3
x3 = np.array([[0, 2, 3, 4], [2, 4, 5, 5], [1, 3, 2, 7]])
theta3 = np.array([[-2.4], [-1.5], [0.3], [-1.4], [0.7]])
logistic_predict_(x3, theta3)
# Output:
array([[0.03916572],
       [0.00045262],
       [0.2890505 ]])
```

Chapter III

Exercise 02

Interlude

Evaluate



Our **model** can **predict the probability** for a given example **to be part of the class labeled as 1**. Now it's time to evaluate how good it is.

The previous loss function, used to evaluate linear regression, is not appropriate in a classification case.

Given the fact that classification tasks imply only two possible values:

- **zero**, if the element is not a member of the predicted class,
- **one**, if the element is a member of the predicted class,

measuring the `'distance'` between the prediction and the label is not going to be the best way to evaluate the performance of a classification model. We'll prefer the **logarithmic** function because it can penalize the wrong predictions even more harshly. But let's separate the two possible cases.

Case 1: The expected output is 1

In mathematical terms, we write:

$$y^{(i)} = 1$$

Here we need a function that will penalize the classifier with a high loss if its prediction ($\hat{y}$) gets close to 0. What do you think of this function? (Have a look at its plot).

$$loss_{y=1} = -\log(\hat{y})$$

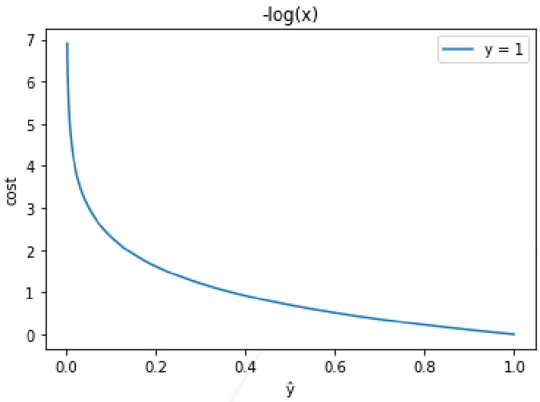


Figure III.1: Loss function when y = 1

You can see from the plot that:

- if the prediction ($\hat{y}$) is close to 0, the loss will be great,
- if the prediction ($\hat{y}$) is close to 1, the loss will be small.

So we got our function that can harshly penalize predictions that get close to 0. But sometimes, $y^{(i)}$ is NOT equal to 1. What if we *want* $\hat{y}$ to be closer to 0 instead?

Case 2: The expected output is 0

In this case we have:

$$y^{(i)} = 0$$

We just need to manipulate the last equation slightly in order to flip the curve the way we need:

$$loss_{y=0} = -\log(1 - \hat{y}^{(i)})$$

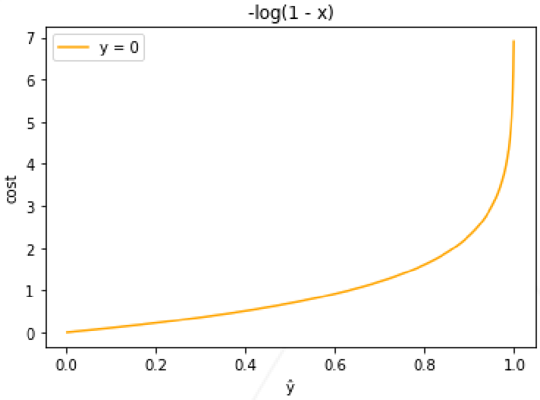


Figure III.2: Loss function when y = 0

You can see from the plot that:

- if the prediction is close to 1, the loss will be great,
- if the prediction is close to 0, the loss will be small.

So this second equation works like the first one, but penalizes the other way: this time $\hat{y}^{(i)}$ gets penalized harder when it gets close to 1.

Now, all we need is a smart way to automatically choose which loss function to use, depending on the value of $y^{(i)}$.

Putting it all together

Let's recap. We need a loss function that can alternate between these:

- If $y^{(i)} = 1$
$$loss = loss_{y=1} = -\log(\hat{y}^{(i)})$$
- If $y^{(i)} = 0$
$$loss = loss_{y=0} = -\log(1 - \hat{y}^{(i)})$$

And we can represent it like this:

How do you switch between $\text{loss}_{y=0}$ and $\text{loss}_{y=1}$ depending on the value of $y^{(i)}$? We could use an if-else statement in the code, but that's not very pretty and it doesn't provide a loss function that can be expressed as a single mathematical expression. It turns out there is a little mathematical trick we can use to make everything stand in one equation.

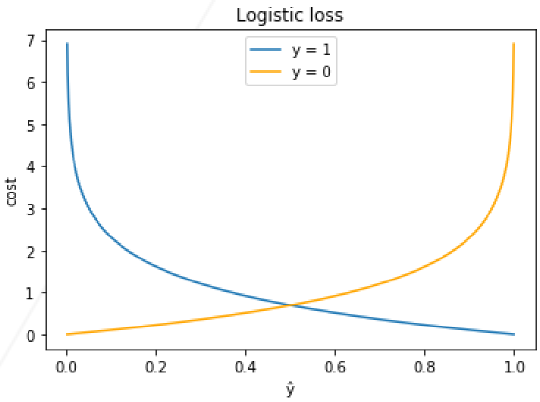


Figure III.3: loss_0 and loss_1

Building the equation for a single training example

For this part let's go step by step. The strategy is to sum both expressions:

$$loss = loss_{y=1} + loss_{y=0}$$

And then we need some kind of switch to "turn off" the term that shouldn't be use for the example i. It turns out we can use the $y^{(i)}$ value itself as a switch!

- When $y^{(i)} = 0$, we just multiply it with the term we don't want and we'll cancel it out:

$$\begin{aligned} loss &= y^{(i)} \cdot loss_{y=1} + loss_{y=0} \\ loss &= 0 \cdot loss_{y=1} + loss_{y=0} \\ loss &= loss_{y=0} \end{aligned}$$

- When $y^{(i)} = 1$, it's a little trickier. We have to multiply the term we want to cancel out by $(1 - y^{(i)})$:

$$\begin{aligned} loss &= loss_{y=1} + (1 - y^{(i)}) \cdot loss_{y=0} \\ loss &= loss_{y=1} + (1 - 1) \cdot loss_{y=0} \\ loss &= loss_{y=1} + 0 \cdot loss_{y=0} \\ loss &= loss_{y=1} \end{aligned}$$

Now, to make a generic equation that works without knowing in advance the value of $y^{(i)}$, all we need is to sum the two loss functions along with their "switches":

$$loss = y^{(i)} \cdot loss_{y=1} + (1 - y^{(i)}) \cdot loss_{y=0}$$

And then, if we develop loss_0 and loss_1:

$$loss = y^{(i)} \cdot (-\log(\hat{y}^{(i)})) + (1 - y^{(i)}) \cdot (-\log(1 - \hat{y}^{(i)}))$$

Finally, if we simplify the sign notation just a bit:

$$loss = -[y^{(i)} \cdot \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \cdot \log(1 - \hat{y}^{(i)})]$$

Cross-Entropy

We are reaching the goal! All we need to do is and average across all training examples and we end up with our final loss function. It has a name: **cross-entropy**. The equation is the following:

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

This formula allows you to calculate the overall loss of a complete set of predictions. If you have enough, you can stop here and move on to the exercise. If you'd like to better understand how it works and have "automatic switch" process broken down for you, here we go:

- If the given example $x^{(i)}$ is not part of the predicted class, $y^{(i)} = 0$:

$$\begin{array}{ccc} y^{(i)} & = & 0 \\ y^{(i)}\log(\hat{y}^{(i)})) & = & 0 \\ 1-y^{(i)} & = & 1 \\ (1-y^{(i)})\log(1-\hat{y}^{(i)}) & = & \log(1-\hat{y}^{(i)}) \end{array}$$

Therefore

$$J(\theta) = -\frac{1}{m}[\sum_{i=1}^{m} \overbrace{\cancel{y^{(i)}\log(\hat{y}^{(i)})}}^{0} + \overbrace{\cancel{(1-y^{(i)})}}^{1}\log(1-\hat{y}^{(i)})]$$

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

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

- If the given example $x^{(i)}$ is part of the predicted class, $y^{(i)} = 1$:

$$\begin{array}{ccc} y^{(i)} & = & 1 \\ y^{(i)}\log(\hat{y}^{(i)}) & = & \log(\hat{y}^{(i)}) \\ 1-y^{(i)} & = & 0 \\ (1-y^{(i)})\log(1-\hat{y}^{(i)}) & = & 0 \end{array}$$

Therefore

$$J(\theta) = -\frac{1}{m}[\sum_{i=1}^{m} \overbrace{\cancel{y^{(i)}}}^{1}\log(\hat{y}^{(i)}) + \overbrace{\cancel{(1-y^{(i)})\log(1-\hat{y}^{(i)})}}^{0}]$$

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

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

Interlude

Linear Algebra Strikes Again!

You've become quite used to vectorization by now. You may have already tried to vectorize the logistic loss function by yourself. Let's look one last time at the former equation:

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

Vectorized Logistic Loss Function

In the **vectorized version**, we remove the sum ($\sum$) because it is captured by the dot products:

$$J(\theta) = -\frac{1}{m}[y \cdot \log(\hat{y}) + (\vec{1} - y) \cdot \log(\vec{1} - \hat{y})]$$

Where:

- $\vec{1}$ is a vector full of 1's with the same dimension of y (m).

$$\vec{1} = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$$

Note: Operations Between Vectors and Scalars

We use the $\vec{1}$ notation to be rigorous, because **addition (or subtraction) between a vector and a scalar is not defined**. In other words, mathematically, you cannot write this: $1 - y$. The only operation defined between a scalar and a vector is multiplication, remember?

However...

`NumPy` is a bit permissive on vectors and matrix operations... The following instructions will get you the same results:

```
# Proper mathematical notation
y = np.array([[4], [7.16], [3.2], [9.37], [0.56]])
ones = np.ones(y.shape)
ones - y
# Output
array([[-3.  ],
       [-6.16],
       [-2.2 ],
       [-8.37],
       [ 0.44]])

# Incorrect mathematical notation
y = np.array([[4], [7.16], [3.2], [9.37], [0.56]])
1 - y
# Output
array([[-3.  ],
       [-6.16],
       [-2.2 ],
       [-8.37],
       [ 0.44]])
```

Strange, isn't it? It happens because of one of NumPy's permissive operations called **Broadcasting**. Broadcasting is a powerful feature whereby NumPy is able to figure out that you actually wanted to perform a subtraction on each element in the vector, so it does it for you automatically. It's very handy to write concise lines of code, but it can insert very sneaky bugs if you aren't 100% confident in what you're doing.

Many of the bugs you will encounter while working on Machine Learning problems will come from NumPy's permissiveness. Such bugs generaly don't throw any errors, but mess up the content of your vectors and matrices and you'll spend an awful lot of time looking for why your model doesn't learn. This is why we **strongly** suggest that you pay attention to your vector (and matrix) shapes and **stick as much as possible to the actual mathematical operations**.

For more information, you can watch this video on dealing with Broadcasting.

	Exercise : 02
Vectorized Logistic Loss Function	
Turn-in directory : *ex*02/	
Files to turn in : `vec_log_loss.py`	
Forbidden functions : `any function that performs derivatives for you`	

Objective

Understanding and manipulation of loss function concept in case of logistic regression. You must implement the following formula as a function:

$$J(\theta) = -\frac{1}{m}[y \cdot \log(\hat{y}) + (\vec{1} - y) \cdot \log(\vec{1} - \hat{y})]$$

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
- $\vec{1}$ is a vector of dimension m, a vector full of ones.

Instructions

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

```
def vec_log_loss_(y, y_hat, eps=1e-15):
    """
    Compute the logistic loss value.
    Args:
        y: has to be an numpy.ndarray, a vector of shape m * 1.
        y_hat: has to be an numpy.ndarray, a vector of shape m * 1.
        eps: epsilon (default=1e-15)
    Returns:
        The logistic loss value as a float.
        None on any error.
    Raises:
        This function should not raise any Exception.
    """
```

The purpose of epsilon (eps) is to avoid $log(0)$ errors, it is a very small residual value we add to y.

Examples

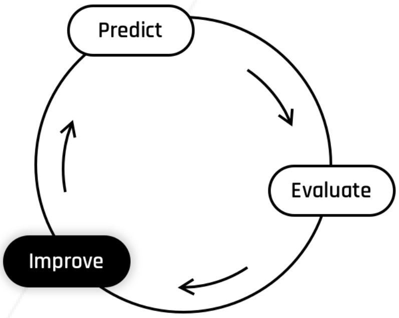
```
# Example 1:
y1 = np.array([1]).reshape((-1, 1))
x1 = np.array([4]).reshape((-1, 1))
theta1 = np.array([[2], [0.5]])
y_hat1 = logistic_predict_(x1, theta1)
vec_log_loss_(y1, y_hat1)
# Output:
0.018149927917808714

# Example 2:
y2 = np.array([[1], [0], [1], [0], [1]])
x2 = np.array([[4], [7.16], [3.2], [9.37], [0.56]])
theta2 = np.array([[2], [0.5]])
y_hat2 = logistic_predict_(x2, theta2)
vec_log_loss_(y2, y_hat2)
# Output:
2.4825011602472347

# Example 3:
y3 = np.array([[0], [1], [1]])
x3 = np.array([[0, 2, 3, 4], [2, 4, 5, 5], [1, 3, 2, 7]])
theta3 = np.array([[-2.4], [-1.5], [0.3], [-1.4], [0.7]])
y_hat3 = logistic_predict_(x3, theta3)
vec_log_loss_(y3, y_hat3)
# Output:
2.993853310859968
```

Chapter IV

Exercise 03

Interlude

Improve



Now we want to improve the algorithm's performance, or in other words, reduce the loss of its predictions. This brings us (again) to calculating the gradient, which will tell us how much and in which direction the theta parameters belonging to the model will be adjusted.

The logistic gradient

If you remember, to calculate the gradient, we start with the loss function and we derive it with respect to each of the theta parameters. If you know multivariate calculus you can try it for yourself, otherwise we've done it for you:

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

Where:

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

This formula should be very familiar to you, as it's the same as the linear regression gradient! The only difference is that $h_\theta(x^{(i)})$ corresponds to **the logistic regression hypothesis instead of the linear regression hypothesis**.

In other words:

$$h_\theta(x^{(i)}) = \text{sigmoid}(\theta \cdot x'^{(i)}) = \frac{1}{1 + e^{-\theta \cdot x'^{(i)}}}$$

Instead of:

$$\cancel{h_\theta(x^{(i)}) = \theta \cdot x'^{(i)}}$$

Interlude

Vectorized Logistic Gradient

Given the previous logistic gradient formula, it's quite easy to produce a vectorized version.

Actually, you almost already implemented it on module07!

As with the previous exercise, **the only thing you have to change is your hypothesis** in order to calculate your logistic gradient.

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

Vectorized Version

Can be vectorized the same way as you did before:

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

	Exercise : 03
Vectorized Logistic Gradient	
Turn-in directory : *ex*03/	
Files to turn in : `vec_log_gradient.py`	
Forbidden functions : `any function that performs derivatives for you`	

Objective

Understand and manipulation of the concept of gradient in the case of logistic formulation. You must implement the following formula as a function:

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

Where:

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

Instructions

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

```
def vec_log_gradient(x, y, theta):
    """Computes a gradient vector from three non-empty numpy.ndarray, without any for-loop. The three arrays must have comp
    Args:
      x: has to be an numpy.ndarray, a matrix of shape m * n.
      y: has to be an numpy.ndarray, a vector of shape m * 1.
      theta: has to be an numpy.ndarray, a vector (n +1) * 1.
    Returns:
      The gradient as a numpy.ndarray, a vector of shape n * 1, containg 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 shapes.
    Raises:
      This function should not raise any Exception.
    """
    ... Your code ...
```

Examples

```
# Example 1:
y1 = np.array([1]).reshape((-1, 1))
x1 = np.array([4]).reshape((-1, 1))
theta1 = np.array([[2], [0.5]])

vec_log_gradient(x1, y1, theta1)
# Output:
array([[-0.01798621],
       [-0.07194484]])

# Example 2:
y2 = np.array([[1], [0], [1], [0], [1]])
x2 = np.array([[4], [7.16], [3.2], [9.37], [0.56]])
theta2 = np.array([[2], [0.5]])

vec_log_gradient(x2, y2, theta2)
# Output:
array([[0.3715235 ],
       [3.25647547]])

# Example 3:
y3 = np.array([[0], [1], [1]])
x3 = np.array([[0, 2, 3, 4], [2, 4, 5, 5], [1, 3, 2, 7]])
theta3 = np.array([[-2.4], [-1.5], [0.3], [-1.4], [0.7]])

vec_log_gradient(x3, y3, theta3)
# Output:
array([[-0.55711039],
       [-0.90334809],
       [-2.01756886],
       [-2.10071291],
       [-3.27257351]])
```

Chapter V

Exercise 04

Exercise : 04
Logistic Regression
Turn-in directory : *ex*04/
Files to turn in : `my_logistic_regression.py`
Forbidden functions : `sklearn`

Objective

The time to use everything you built so far has come! Demonstrate your knowledge by implementing a logistic regression classifier using the gradient descent algorithm. You must have seen the power of `numpy` for vectorized operations. Well let's make something more concrete with that.

You may have to take a look at Scikit-Learn's implementation of logistic regression and noticed that the **sklearn.linear__model.LogisticRegression** class offers a lot of options.

The goal of this exercise is to make a simplified but nonetheless useful and powerful version, with fewer options.

Instructions

In the `my_logistic_regression.py` file, write a `MyLogisticRegression` class as in the instructions below:

```
class MyLogisticRegression():
        """
        Description:
                My personnal logistic regression to classify things.
        """
    def __init__(self, theta, alpha=0.001, max_iter=1000):
        self.alpha = alpha
        self.max_iter = max_iter
        self.theta = theta
        ... Your code here ...

        ... other methods ...
```

You will add at least the following methods:

- `predict_(self, x)`
- `loss_elem_(self, y, yhat)`
- `loss_(self, y, yhat)`
- `fit_(self, x, y)`

You have already written these functions, you will just need few adjustments so that they all work well within your `MyLogisticRegression` class.

Examples

```
import numpy as np
from my_logistic_regression import MyLogisticRegression as MyLR
X = np.array([[1., 1., 2., 3.], [5., 8., 13., 21.], [3., 5., 9., 14.]])
Y = np.array([[1], [0], [1]])
thetas = np.array([[2], [0.5], [7.1], [-4.3], [2.09]])
mylr = MyLR(thetas)

# Example 0:
mylr.predict_(X)
# Output:
array([[0.99930437],
       [1.        ],
       [1.        ]])

# Example 1:
mylr.loss_(X,Y)
# Output:
11.513157421577004

# Example 2:
mylr.fit_(X, Y)
mylr.theta
# Output:
array([[ 2.11826435]
       [ 0.10154334]
       [ 6.43942899]
       [-5.10817488]
       [ 0.6212541 ]])

# Example 3:
mylr.predict_(X)
# Output:
array([[0.57606717]
       [0.68599807]
       [0.06562156]])

# Example 4:
mylr.loss_(X,Y)
# Output:
1.4779126923052268
```

Chapter VI

Exercise 05

	Exercise : 05
Practicing Logistic Regression	
Turn-in directory : *ex*05/	
Files to turn in : `mono_log.py, multi_log.py`	
Forbidden functions : `sklearn`	

Objective

Now it's time to test your Logistic Regression Classifier on real data! You will use the **solar_system_census_dataset**.

Instructions

Some words about the dataset:

- You will work with data from the last Solar System Census.
- The dataset is divided in two files which can be found in the `resources` folder: `solar_system_census.csv` and `solar_system_census_planets.csv`.
- The first file contains biometric information such as the height, weight, and bone density of several Solar System citizens.
- The second file contains the homeland of each citizen, indicated by its Space Zipcode representation (i.e. one number for each planet... :)).

As you should know, Solar citizens come from four registered areas (zipcodes):

- The flying cities of Venus (0),
- United Nations of Earth (1),
- Mars Republic (2),
- The Asteroids' Belt colonies (3).

You are expected to produce 2 programs that will use Logistic Regression to predict from which planet each citizen comes from, based on the other variables found in the census dataset.

But wait... what? There are four different planets! How do you make a classifier discriminate between 4 categories? Let's go step by step...

One Label to Discriminate Them All

You already wrote a Logistic Regression Classifier that can discriminate between two classes. We can use it to solve the problem! Let's start by having it discriminate between citizens who come from your favorite planet and everybody else!

Your program (in `mono_log.py`) will:

1. Take an argument: `-zipcode=x` with x being 0, 1, 2 or 3. If no argument, usage will be displayed.
2. Split the dataset into a training and a test set.
3. Select your favorite Space Zipcode and generate a new `numpy.array` to label each citizen according to your new selection criterion:
    - 1 if the citizen's zipcode corresponds to your favorite planet.
    - 0 if the citizen has another zipcode.
4. Train a logistic model to predict if a citizen comes from your favorite planet or not, using your brand new label.
5. Calculate and display the fraction of correct predictions over the total number of predictions based on the test set.
6. Plot 3 scatter plots (one for each pair of citizen features) with the dataset and the final prediction of the model.

You can use normalization on your dataset. The question is: Should you?

You now have a model that can discriminate between citizens that come from one specific planet and everyone else. It's a first step, a good one, but we still have work to do before we can classify citizens among four planets!

So how does **Multiclass Logistic Regression** work?

One Versus All

The idea now is to apply what is called **one-versus-all classification**. It's quite straightforward:

Your program (in `multi_log.py`) will:

1. Split the dataset into a training and a test set.
2. Train 4 logistic regression classifiers to discriminate each class from the others (the way you did in part one).
3. Predict for each example the class according to each classifiers and select the one with the highest output probability.
4. Calculate and display the fraction of correct predictions over the total number of predictions based on the test set.
5. Plot 3 scatter plots (one for each pair of citizen features) with the dataset and the final prediction of the model.

Examples

If a cititzen got the following classification probabilities:

- Planet 0 vs all: 0.38
- Planet 1 vs all: 0.51
- Planet 2 vs all: 0.12
- Planet 3 vs all: 0.89

Then the citizen should be classified as coming from *Planet 3*.

Chapter VII

Exercise 06

Interlude

More Evaluation Metrics!

Once your classifier is trained, you want to evaluate its performance. You already know about *cross-entropy*, as you implemented it as your *loss function*. But when it comes to classification, there are more informative metrics we can use besides the loss function. Each metric focuses on different error types. But what is an error type?

A single classification prediction is either right or wrong, nothing in between. Either an object is assigned to the right class, or to the wrong class. When calculating performance scores for a multiclass classifier, we like to compute a separate score for each class that your classifier learned to discriminate (in a one-vs-all manner). In other words, for a given *Class A*, we want a score that can answer the question: "how good is the model at assigning *A* objects to *Class A*, and at NOT assigning *non-A* objects to *Class A*?"

You may not realize it yet, but this question involves measuring two very different error types, and the distinction is crucial.

Error Types

With respect to a given *Class A*, classification errors fall in two categories:

- **False positive:** when a *non-A* object is assigned to *Class A*. For example:
  - Pulling the fire alarm when there is no fire.
  - Considering that someone is sick when she isn't.
  - Identifying a face in an image when in fact it was a Teddy Bear.
- **False negative:** when an *A* object is assigned to another class than *Class A*. For example:
  - Not pulling the fire alarm when there is a fire.
  - Considering that someone is not sick when she is.
  - Failing to recognize a face in an image that does contain one.

It turns out that it's really hard to minimize both error types at the same time. At some point you'll need to decide which one is the most critical, depending on your use case. For example, if you want to detect cancer, of course it's not good if your model erroneously diagnoses cancer on a few healthy patients (**false positives**), but you absolutely want to avoid failing at diagnosing cancer on affected patients (**false negatives**) and let them go on with their lives while developing a potentially dangerous cancer.

Metrics

A metric is computed on a set of predictions along with the corresponding set of actual categories. The metric you choose will focus more or less on those two error types. If we come back to the **Class A** classifier:

- **Accuracy**: tells you the percentage of predictions that are accurate (i.e. the correct class was predicted). Accuracy doesn't give information about either error type.
- **Precision**: tells you how much you can trust your model when it says that an object belongs to *Class A*. More precisely, it is the percentage of the objects assigned to *Class A* that really were *A* objects. You use precision when you want to control for **False positives**.
- **Recall**: tells you how much you can trust that your model is able to recognize ALL *Class A* objects. It is the percentage of all **A** objects that were properly classified by the model as *Class A*. You use recall when you want to control for **False negatives**.
- **F1 score**: combines precision and recall in one single measure. You use the F1 score when want to control both **False positives** and **False negatives**.

	Exercise : 06
Other metrics	
Turn-in directory : $ex06/$	
Files to turn in : `other_metrics.py`	
Forbidden functions : `None`	

Objective

Understanding and manipulation of classification criteria (TP, FP, ...) and metrics. The goal of this exercise is to write four metric functions (which are also available in **sklearn.metrics**) and to understand what they measure and how they are constructed.

You must implement the following fomulas:

$$\text{accuracy} = \frac{\text{tp} + \text{tn}}{\text{tp} + \text{fp} + \text{tn} + \text{fn}}$$

$$\text{precision} = \frac{\text{tp}}{\text{tp} + \text{fp}}$$

$$\text{recall} = \frac{\text{tp}}{\text{tp} + \text{fn}}$$

$$\text{F1score} = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

Where:

- tp is the number of **true positives**,
- fp is the number of **false positives**,
- tn is the number of **true negatives**,
- fn is the number of **false negatives**.

Instructions

For the sake of simplicity, we will only ask you to use two parameters.

In the `other_metrics.py` file, write the following functions as per the instructions below:

```
def accuracy_score_(y, y_hat):
    """
    Compute the accuracy score.
    Args:
        y:a numpy.ndarray for the correct labels
        y_hat:a numpy.ndarray for the predicted labels
    Returns:
        The accuracy score as a float.
        None on any error.
    Raises:
        This function should not raise any Exception.
    """
    ... Your code ...

def precision_score_(y, y_hat, pos_label=1):
    """
    Compute the precision score.
    Args:
        y:a numpy.ndarray for the correct labels
        y_hat:a numpy.ndarray for the predicted labels
        pos_label: str or int, the class on which to report the precision_score (default=1)
    Returns:
        The precision score as a float.
        None on any error.
    Raises:
        This function should not raise any Exception.
    """
    ... Your code ...

def recall_score_(y, y_hat, pos_label=1):
    """
    Compute the recall score.
    Args:
        y:a numpy.ndarray for the correct labels
        y_hat:a numpy.ndarray for the predicted labels
        pos_label: str or int, the class on which to report the precision_score (default=1)
    Returns:
        The recall score as a float.
        None on any error.
    Raises:
        This function should not raise any Exception.
    """
    ... Your code ...

def f1_score_(y, y_hat, pos_label=1):
    """
    Compute the f1 score.
    Args:
        y:a numpy.ndarray for the correct labels
        y_hat:a numpy.ndarray for the predicted labels
        pos_label: str or int, the class on which to report the precision_score (default=1)
    Returns:
        The f1 score as a float.
        None on any error.
    Raises:
        This function should not raise any Exception.
    """
    ... Your code ...
```

Examples

```
import numpy as np
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# Example 1:
y_hat = np.array([1, 1, 0, 1, 0, 0, 1, 1]).reshape((-1, 1))
y = np.array([1, 0, 0, 1, 0, 1, 0, 0]).reshape((-1, 1))

# Accuracy
## your implementation
accuracy_score_(y, y_hat)
## Output:
0.5
## sklearn implementation
accuracy_score(y, y_hat)
## Output:
0.5

# Precision
## your implementation
precision_score_(y, y_hat)
## Output:
0.4
## sklearn implementation
precision_score(y, y_hat)
## Output:
0.4

# Recall
## your implementation
recall_score_(y, y_hat)
## Output:
0.6666666666666666
## sklearn implementation
recall_score(y, y_hat)
## Output:
0.6666666666666666

# F1-score
## your implementation
f1_score_(y, y_hat)
## Output:
0.5
## sklearn implementation
f1_score(y, y_hat)
## Output:
0.5
```

```
# Example 2:
y_hat = np.array([’norminet’, ’dog’, ’norminet’, ’norminet’, ’dog’, ’dog’, ’dog’, ’dog’])
y = np.array([’dog’, ’dog’, ’norminet’, ’norminet’, ’dog’, ’norminet’, ’dog’, ’norminet’])

# Accuracy
## your implementation
accuracy_score_(y, y_hat)
## Output:
0.625
## sklearn implementation
accuracy_score(y, y_hat)
## Output:
0.625

# Precision
## your implementation
precision_score_(y, y_hat, pos_label=’dog’)
## Output:
0.6
## sklearn implementation
precision_score(y, y_hat, pos_label=’dog’)
## Output:
0.6

# Recall
## your implementation
recall_score_(y, y_hat, pos_label=’dog’)
## Output:
0.75
## sklearn implementation
recall_score(y, y_hat, pos_label=’dog’)
## Output:
0.75

# F1-score
## your implementation
f1_score_(y, y_hat, pos_label=’dog’)
## Output:
0.6666666666666665
## sklearn implementation
f1_score(y, y_hat, pos_label=’dog’)
## Output:
0.6666666666666665
```

```
# Example 3:
y_hat = np.array(['norminet', 'dog', 'norminet', 'norminet', 'dog', 'dog', 'dog', 'dog'])
y = np.array(['dog', 'dog', 'norminet', 'norminet', 'dog', 'norminet', 'dog', 'norminet'])

# Precision
## your implementation
precision_score_(y, y_hat, pos_label='norminet')
## Output:
0.6666666666666666
## sklearn implementation
precision_score(y, y_hat, pos_label='norminet')
## Output:
0.6666666666666666

# Recall
## your implementation
recall_score_(y, y_hat, pos_label='norminet')
## Output:
0.5
## sklearn implementation
recall_score(y, y_hat, pos_label='norminet')
## Output:
0.5

# F1-score
## your implementation
f1_score_(y, y_hat, pos_label='norminet')
## Output:
0.5714285714285715
## sklearn implementation
f1_score(y, y_hat, pos_label='norminet')
## Output:
0.5714285714285715
```

Chapter VIII

Exercise 07

Exercise : 07
Confusion Matrix
Turn-in directory : *ex*07/
Files to turn in : `confusion_matrix.py`
Forbidden functions : `None`

Objective

Manipulation of confusion matrix concept. The goal of this exercise is to reimplement the function `confusion_matrix` available in **sklearn.metrics** and to learn what does the confusion matrix represent.

Instructions

For the sake of simplicity, we will only ask you to use three parameters. Be careful to respect the order, true labels are rows and predicted labels are columns:

		predicted labels	
		label 1	label 2
true label	label 1		
	label 2		

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

```
def confusion_matrix_(y_true, y_hat, labels=None):
    """
    Compute confusion matrix to evaluate the accuracy of a classification.
    Args:
        y_true: numpy.ndarray for the correct labels
        y_hat: numpy.ndarray for the predicted labels
        labels: Optional, a list of labels to index the matrix.
                This may be used to reorder or select a subset of labels. (default=None)
    Returns:
        The confusion matrix as a numpy ndarray.
        None on any error.
    Raises:
        This function should not raise any Exception.
    """
    ... Your code ...
```

Examples

```
import numpy as np
from sklearn.metrics import confusion_matrix

y_hat = np.array([['norminet'], ['dog'], ['norminet'], ['norminet'], ['dog'], ['bird']])
y = np.array([['dog'], ['dog'], ['norminet'], ['norminet'], ['dog'], ['norminet']])

# Example 1:
## your implementation
confusion_matrix_(y, y_hat)
## Output:
array([[0 0 0]
       [0 2 1]
       [1 0 2]])
## sklearn implementation
confusion_matrix(y, y_hat)
## Output:
array([[0 0 0]
       [0 2 1]
       [1 0 2]])

# Example 2:
## your implementation
confusion_matrix_(y, y_hat, labels=['dog', 'norminet'])
## Output:
array([[2 1]
       [0 2]])
## sklearn implementation
confusion_matrix(y, y_hat, labels=['dog', 'norminet'])
## Output:
array([[2 1]
       [0 2]])
```

VIII.1 Optional part

VIII.1.1 Objective(s):

For a more visual version, you can add an option to your previous confusion_matrix_ function to return a `pandas.DataFrame` instead of a numpy array.

VIII.1.2 Instructions:

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

```
def confusion_matrix_(y_true, y_hat, labels=None, df_option=False):
    """
    Compute confusion matrix to evaluate the accuracy of a classification.
    Args:
        y_true: a numpy.ndarray for the correct labels
        y_hat: a numpy.ndarray for the predicted labels
        labels: optional, a list of labels to index the matrix. This may be used to reorder or select a subset of labels. (
        df_option: optional, if set to True the function will return a pandas DataFrame instead of a numpy array. (default=
    Returns:
        Confusion matrix as a numpy ndarray or a pandas DataFrame according to df_option value.
        None on any error.
    Raises:
        This function should not raise any Exception.
    """
    ... Your code ...
```

VIII.2 Examples:

```
import numpy as np
y_hat = np.array(['norminet', 'dog', 'norminet', 'norminet', 'dog', 'bird'])
y = np.array(['dog', 'dog', 'norminet', 'norminet', 'dog', 'norminet'])

# Example 1:
confusion_matrix_(y, y_hat, df_option=True)
# Output:
           bird  dog  norminet
 bird         0    0         0
 dog          0    2         1
 norminet     1    0         2

# Example 2:
confusion_matrix_(y, y_hat, labels=['bird', 'dog'], df_option=True)
# Output:
           bird  dog
 bird         0    0
 dog          0    2
```

If you fail this exercise on your first attempt, Norminet will curse you forever. Yeah, you'd better do it right or you are in trouble my friend, big trouble!

Chapter IX

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. Why do we use logistic hypothesis for a classfication problem rather than a linear hypothesis?

2. What is the decision boundary?

3. In the case we decide to use a linear hypothesis to tackle a classification problem, why the classification of some data points can be modified by considering more examples (for example, extra data points with extrem ordinate)?

4. In a one versus all classification approach, how many logisitic regressor do we need to distinguish between N classes?

5. Can you explain the difference between accuracy and precision? What is the type I and type II errors?

6. What is the interest of the F1-score?

Contact

You can contact 42AI association by email: contact@42ai.fr
You can join the association on 42AI slack and/or posutale to one of the association teams.

Acknowledgements

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

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

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

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

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

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

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