42 | ARTIFICIAL INTELLIGENCE

Machine Learning Bootcamp - Module 04

Regularization

*Summary: Today you will fight **overfitting**! You will discover the concepts of **regularization** and how to implement it into the algorithms you know.*

Notions and ressources

Notions of the module

- Regularization
- Overfitting
- Regularized loss function
- Regularized gradient descent
- Regularized linear regression
- Regularized logistic regression

Useful Ressources

You are recommended to use the following material: Machine Learning MOOC - Stanford

This series of videos is available at no cost: simply log in, select "Enroll for Free", and click "Audit" at the bottom of the pop-up window.

The following sections of the course are particularly relevant to today's exercises:

Week 3: Classification

Classification with logistic regression (already seen in module 03)

- Motivations
- Logistic regression
- Decision boundary

Cost function for logistic regression (already seen in module 03)

- Cost function for logistic regression
- Simplified Cost Function for Logistic Regression

Gradient descent for logistic regression (already seen in module 03)

- Gradient Descent Implementation

The problem of overfitting (New !!!)

- The problem of overfitting
- Addressing overfitting
- Cost function with regularization
- Regularized linear regression
- Regularized logistic regression

All videos above are available also on this Andrew Ng's YouTube playlist, videos 31 to 36 (already seen in module 03) and 37 to 41 (new !!!).

Chapter I

Common Instructions

- The version of Python recommended to use is 3.7. You can check your Python's version 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 or Black, which are convenient packages to check your 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're planning on using an AI assistant such as a LLM, make sure it is helpful for you to **learn and practice**, not to provide you with hands-on solution ! Own your tool, don't let it own you.
- 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.

Contents

Chapter II

Exercise 00

	Exercise : 00
Polynomial models II	
Turn-in directory : $ex00/$	
Files to turn in : `polynomial_model_extended.py`	
Forbidden functions : `sklearn`	

Objective

Create a function that takes a matrix X of dimensions $(m \times n)$ and an integer p as input, and returns a matrix of dimension $(m \times (np))$.

For each column x_j of the matrix X, the new matrix contains x_j raised to the power of k, for $k = 1, 2, ..., p$:

$$x_1 \mid \ldots \mid x_n \mid x_1^2 \mid \ldots \mid x_n^2 \mid \ldots \mid x_1^p \mid \ldots \mid x_n^p$$

Instructions

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

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

Examples

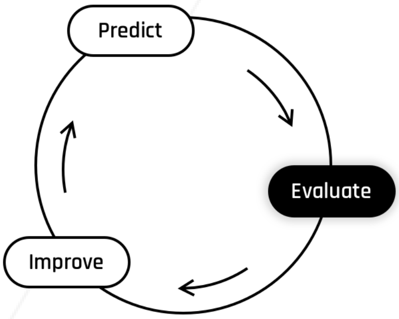
```
import numpy as np
x = np.arange(1,11).reshape(5, 2)

# Example 1:
add_polynomial_features(x, 3)
# Output:
array([[   1,    2,    1,    4,    1,    8],
       [   3,    4,    9,   16,   27,   64],
       [   5,    6,   25,   36,  125,  216],
       [   7,    8,   49,   64,  343,  512],
       [   9,   10,   81,  100,  729, 1000]])

# Example 2:
add_polynomial_features(x, 4)
# Output:
array([[    1,     2,     1,     4,     1,     8,     1,    16],
       [    3,     4,     9,    16,    27,    64,    81,   256],
       [    5,     6,    25,    36,   125,   216,   625,  1296],
       [    7,     8,    49,    64,   343,   512,  2401,  4096],
       [    9,    10,    81,   100,   729,  1000,  6561, 10000]])
```

Chapter III

Exercise 01

Interlude

Fighting Overfitting... With Regularization



In the **module07**, we talked about the problem of **overfitting** and the necessity of splitting the dataset into a **training set** and a **test set** in order to spot it.

However, being able to detect overfitting does not mean being able to avoid it.

To address this important issue, it is time to introduce a new technique: **regularization**. If you remember well, overfitting happens because the model takes advantage of irrelevant signals in the training data.

The basic idea behind regularization is to **penalize the model for putting too much weight on certain** (usually heavy polynomials) **features**.

We do this by adding an extra term to the loss function formula:

$$\text{regularized loss function} = \text{loss function} + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

By doing so, **we are encouraging the model to keep its θ values as small as possible**.
Indeed, the values of θ *themselves* are now taken into account when calculating the loss.

λ (called *lambda*) is the parameter through which you can modulate how reglarization should impact the model's construction.

- If $\lambda = 0$, there is no regularization (as we did until now).
- If λ is very large, it will drive all the θ parameters to 0.

> In the regularization term, the sum starts at $j = 1$ because we do NOT want to penalize the value of θ_0 (the y-intercept, which doesn't depend on a feature).

Be careful!

Machine Learning was essentially developped by computer scientists (not mathematicians).

This can cause problems when we try to represent things mathematically. For example: using the θ_0 notation to represent the y-intercept makes things easy when we apply the linear algebra trick, **but** it completely messes up with the overall matrix notation!

According to that notation, the X' matrix has the following properties:

- its rows, $x'^{(i)}$, follow the mathematical indexing: starting at 1.
- its columns, x'_j, follow the computer science indexing: starting at 0.

$$X' = \underbrace{\begin{bmatrix} 1 & x_1^{(1)} & \dots & x_n^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & \dots & x_n^{(m)} \end{bmatrix}}_{j = 0, \dots, n} = \left.\begin{bmatrix} x_0^{(1)} & x_1^{(1)} & \dots & x_n^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ x_0^{(m)} & x_1^{(m)} & \dots & x_n^{(m)} \end{bmatrix}\right\} i = 1, \dots, m$$

It's precisely for this reason you keep seeing that X' is of dimension $(m \times (n+1))$.

Terminology

The regularization technique we are introducing here is named **L_2 regularization**, because it adds the squared **L_2 norm** of the θ vector to the loss function.

The L_2 norm of a given vector x, written

$$L_2(x) = ||x||_2 = \sqrt{\sum_i x_i^2} L_2(x)^2 = ||x||_2^2 = \sum_i x_i^2$$

is its **euclidean norm** (i.e. the sum of its components squared).

There is an infinite variety of norms that could be used as regularization terms, depending on the desired regularization effect.
Here, we will only use L_2, the most common one.

The notation $\sum_i$ means: "the sum for all i"
There is no need to explicitly specify the start and the end of the summation index if we want to sum over all the values of i. However, it is better to do it anyway as it forces us to make sure of what we are doing. And in our case, we do not want to sum over θ_0 ...

Our old friend vectorization ...

It is no surprise, we can use vectorization to calculate $\sum_{j=1}^{n} \theta_j^2$ more efficiently.

It could be a good exercise for you to try to figure it out by yourself.

We suggest you give it a try and then check the answer on the next page.

Answers to the Vectorization Problem

So, how do you vectorize the following?

$$\sum_{i=j}^{n} \theta_j^2$$

It's very similar to a **dot product** of θ with itself. The only problem here is to find a way to not take θ_0 into account.

Let's construct a vector θ' with the following rules:

$$\begin{aligned} \theta'_0 &= 0 \\ \theta'_j &= \theta_j \quad \text{for } j = 1, \dots, n \end{aligned}$$

In other words:

$$\theta' = \begin{bmatrix} 0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix}$$

This way, we can perform the dot product without having θ_0 interfering in our calculations:

$$\begin{aligned} \theta' \cdot \theta' &= \begin{bmatrix} 0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix} \cdot \begin{bmatrix} 0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix} \\ &= 0 \cdot 0 + \theta_1 \cdot \theta_1 + \cdots + \theta_n \cdot \theta_n \\ &= \textstyle\sum_{j=1}^{n} \theta_j^2 \end{aligned}$$

	Exercise : 01
L2 Regularization	
Turn-in directory : *ex*01/	
Files to turn in : `l2_reg.py`	
Forbidden functions : `sklearn`	

Objective

You must implement the following formulas as functions:

Iterative

$$L_2(\theta)^2 = \sum_{j=1}^{n} \theta_j^2$$

Where:

- θ is a vector of dimension $(n+1)$.

Vectorized

$$L_2(\theta)^2 = \theta' \cdot \theta'$$

Where:

- θ' is a vector of dimension $(n+1)$, constructed using the following rules:

$$\begin{aligned} \theta'_0 &= 0 \\ \theta'_j &= \theta_j \quad \text{for } j = 1, \dots, n \end{aligned}$$

Instructions

In the `l2_reg.py` file, write the following functions as per the instructions given below:

```
def iterative_l2(theta):
        """Computes the L2 regularization of a non-empty numpy.ndarray, with a for-loop.
        Args:
                theta: has to be a numpy.ndarray, a vector of shape n * 1.
        Returns:
                The L2 regularization as a float.
                None if theta in an empty numpy.ndarray.
        Raises:
                This function should not raise any Exception.
        """
        ... Your code ...

def l2(theta):
        """Computes the L2 regularization of a non-empty numpy.ndarray, without any for-loop.
        Args:
                theta: has to be a numpy.ndarray, a vector of shape n * 1.
        Returns:
                The L2 regularization as a float.
                None if theta in an empty numpy.ndarray.
        Raises:
                This function should not raise any Exception.
        """
        ... Your code ...
```

Examples

```
x = np.array([2, 14, -13, 5, 12, 4, -19]).reshape((-1, 1))

# Example 1:
iterative_l2(x)
# Output:
911.0

# Example 2:
l2(x)
# Output:
911.0

y = np.array([3,0.5,-6]).reshape((-1, 1))
# Example 3:
iterative_l2(y)
# Output:
36.25

# Example 4:
l2(y)
# Output:
36.25
```

Chapter IV

Exercise 02

	Exercise : 02
Regularized Linear Loss Function	
Turn-in directory : *ex*02/	
Files to turn in : `linear_loss_reg.py`	
Forbidden functions : `sklearn`	

Objective

You must implement the following formula as a function:

$$J(\theta) = \frac{1}{2m}[(\hat{y} - y) \cdot (\hat{y} - y) + \lambda(\theta' \cdot \theta')]$$

Where:

- y is a vector of dimension m, the expected values
- $\hat{y}$ is a vector of dimension m, the predicted values
- λ is a constant, the regularization hyperparameter
- θ' is a vector of dimension n, constructed using the following rules:

$$\begin{aligned} \theta'_0 & = 0 \\ \theta'_j & = \theta_j \quad \text{for } j = 1, \ldots, n \end{aligned}$$

Instructions

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

```
def reg_loss_(y, y_hat, theta, lambda_):
        """Computes the regularized loss of a linear regression model from two non-empty numpy.array,
without any for loop. The two arrays must have the same dimensions.
        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.
                theta: has to be a numpy.ndarray, a vector of shape n * 1.
                lambda_: has to be a float.
        Returns:
                The regularized loss as a float.
                None if y, y_hat, or theta are empty numpy.ndarray.
                None if y and y_hat do not share the same shapes.
        Raises:
                This function should not raise any Exception.
        """
        ... Your code ...
```

`such a situation could be a good use case for decorators...`

Examples

```
y = np.array([2, 14, -13, 5, 12, 4, -19]).reshape((-1, 1))
y_hat = np.array([3, 13, -11.5, 5, 11, 5, -20]).reshape((-1, 1))
theta = np.array([1, 2.5, 1.5, -0.9]).reshape((-1, 1))

# Example :
reg_loss_(y, y_hat, theta, .5)
# Output:
0.8503571428571429

# Example :
reg_loss_(y, y_hat, theta, .05)
# Output:
0.5511071428571429

# Example :
reg_loss_(y, y_hat, theta, .9)
# Output:
1.116357142857143
```

Chapter V

Exercise 03

	Exercise : 03
Regularized Logistic Loss Function	
Turn-in directory : $ex03/$	
Files to turn in : `logistic_loss_reg.py`	
Forbidden functions : `sklearn`	

Objective

You must implement the following formula as a function:

$$J(\theta) = -\frac{1}{m}[y \cdot \log(\hat{y}) + (\vec{1} - y) \cdot \log(\vec{1} - \hat{y})] + \frac{\lambda}{2m}(\theta' \cdot \theta')$$

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
- λ is a constant, the regularization hyperparameter
- θ' is a vector of dimension n, constructed using the following rules:

$$\begin{aligned} \theta'_0 &= 0 \\ \theta'_j &= \theta_j \quad \text{for } j = 1, \dots, n \end{aligned}$$

Instructions

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

```
def reg_log_loss_(y, y_hat, theta, lambda_):
        """Computes the regularized loss of a logistic regression model from two non-empty numpy.ndarray,
        without any for loop. The two arrays must have the same shapes.
        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.
                theta: has to be a numpy.ndarray, a vector of shape n * 1.
                lambda_: has to be a float.
        Returns:
                The regularized loss as a float.
                None if y, y_hat, or theta is empty numpy.ndarray.
                None if y and y_hat do not share the same shapes.
        Raises:
                This function should not raise any Exception.
        """
        ... Your code ...
```

Here again, seems to be a good use case for decorators ...

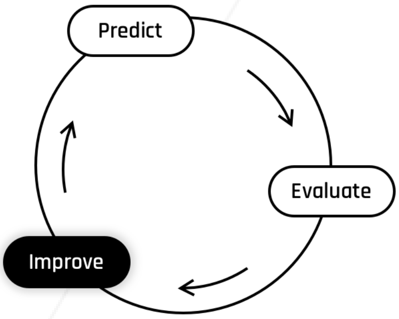
Examples

```
y = np.array([1, 1, 0, 0, 1, 1, 0]).reshape((-1, 1))
y_hat = np.array([.9, .79, .12, .04, .89, .93, .01]).reshape((-1, 1))
theta = np.array([1, 2.5, 1.5, -0.9]).reshape((-1, 1))

# Example :
reg_log_loss_(y, y_hat, theta, .5)
# Output:
0.43377043716475955

# Example :
reg_log_loss_(y, y_hat, theta, .05)
# Output:
0.13452043716475953

# Example :
reg_log_loss_(y, y_hat, theta, .9)
# Output:
0.6997704371647596
```

Chapter VI

Exercise 04

Interlude - Regularized Gradient



To derive the gradient of the regularized loss function, $\nabla(J)$ you have to change a bit the formula of the unregularized gradient.

Given the fact that we are not penalizing θ_0, the formula will remain the same as before for this parameter. For the other parameters $(\theta_1, \ldots, \theta_n)$, we must add the partial derivative of the regularization term: $\lambda\theta_j$.

Therefore, we get:

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

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

Where:

- $\nabla(J)_j$ is the j$^{\text{th}}$ component of the gradient vector $\nabla(J)$
- m is the number of training examples used
- $h_\theta(x^{(i)})$ is the model's prediction for the i$^{\text{th}}$ training example
- $x^{(i)}$ is the feature vector of the i$^{\text{th}}$ training example
- $y^{(i)}$ is the expected target value for the i$^{\text{th}}$ example

- λ is a constant, the regularization hyperparameter
- θ_j is the j^{th} parameter of the θ vector

Which can be vectorized as:

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

Where:

- $\nabla(J)$ is a vector of dimension $(n+1)$, the gradient vector
- m is the number of training examples used
- X is a matrix of dimension $(m \times n)$, the design matrix
- X' is a matrix of dimension $(m \times (n+1))$, the design matrix onto which a column of ones is added as a first column
- y is a vector of dimension m, the vector of expected values
- $h_\theta(X)$ is a vector of dimension m, the vector of predicted values
- λ is a constant
- θ is a vector of dimension $(n+1)$, the parameter vector
- θ' is a vector of dimension $(n+1)$, constructed using the following rules:

$$\begin{aligned} \theta'_0 &= 0 \\ \theta'_j &= \theta_j \quad \text{for } j = 1, \ldots, n \end{aligned}$$

Linear Gradient vs Logistic Gradient

As before, we draw your attention on the only difference between the linear regression's and the logistic regression's gradient equations: **the hypothesis function** $h_\theta(X)$.

- In the linear regression: $h_\theta(X) = X'\theta$
- In the logistic regression: $h_\theta(X) = \text{sigmoid}(X'\theta)$

	Exercise : 04
Regularized Linear Gradient	
Turn-in directory : *ex*04/	
Files to turn in : `reg_linear_grad.py`	
Forbidden functions : `sklearn`	

Objective

You must implement the following formulas as functions for the **linear regression hypothesis**

Iterative

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

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

Where:

- $\nabla(J)_j$ is the j$^{\text{th}}$ component of $\nabla(J)$
- $\nabla(J)$ is a vector of dimension $(n+1)$, the gradient vector
- m is a constant, the number of training examples used
- $h_\theta(x^{(i)})$ is the model's prediction for the i$^{\text{th}}$ training example
- $x^{(i)}$ is the feature vector (of dimension n) of the i$^{\text{th}}$ training example, found in the i$^{\text{th}}$ row of the X matrix
- X is a matrix of dimensions $(m \times n)$, the design matrix
- $y^{(i)}$ is the i$^{\text{th}}$ component of the y vector
- y is a vector of dimension m, the vector of expected values
- λ is a constant, the regularization hyperparameter
- θ_j is the j$^{\text{th}}$ parameter of the θ vector
- θ is a vector of dimension $(n+1)$, the parameter vector

Vectorized

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

Where:

- $\nabla(J)$ is a vector of dimension $(n+1)$, the gradient vector
- m is a constant, the number of training examples used
- 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 ones is added as a first column
- X'^{T} is the transpose of tha matrix, with dimensions $((n+1) \times m)$
- $h_\theta(X)$ is a vector of dimension m, the vector of predicted values
- y is a vector of dimension m, the vector of expected values
- λ is a constant, the regularization hyperparameter
- θ is a vector of dimension $(n+1)$, the parameter vector
- θ' is a vector of dimension $(n+1)$, constructed using the following rules:

$$\begin{aligned} \theta'_0 &= 0 \\ \theta'_j &= \theta_j \quad \text{for } j = 1, \dots, n \end{aligned}$$

Instructions

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

```python
def reg_linear_grad(y, x, theta, lambda_):
    """Computes the regularized linear gradient of three non-empty numpy.ndarray,
       with two for-loop. The three arrays must have compatible shapes.
    Args:
      y: has to be a numpy.ndarray, a vector of shape m * 1.
      x: has to be a numpy.ndarray, a matrix of dimesion m * n.
      theta: has to be a numpy.ndarray, a vector of shape (n + 1) * 1.
      lambda_: has to be a float.
    Return:
      A numpy.ndarray, a vector of shape (n + 1) * 1, containing the results of the formula for all j.
      None if y, x, or theta are empty numpy.ndarray.
      None if y, x or theta does not share compatibles shapes.
      None if y, x or theta or lambda_ is not of the expected type.
    Raises:
      This function should not raise any Exception.
    """
    ... Your code ...

def vec_reg_linear_grad(y, x, theta, lambda_):
    """Computes the regularized linear gradient of three non-empty numpy.ndarray,
       without any for-loop. The three arrays must have compatible shapes.
    Args:
      y: has to be a numpy.ndarray, a vector of shape m * 1.
      x: has to be a numpy.ndarray, a matrix of dimesion m * n.
      theta: has to be a numpy.ndarray, a vector of shape (n + 1) * 1.
      lambda_: has to be a float.
    Return:
      A numpy.ndarray, a vector of shape (n + 1) * 1, containing the results of the formula for all j.
      None if y, x, or theta are empty numpy.ndarray.
      None if y, x or theta does not share compatibles shapes.
      None if y, x or theta or lambda_ is not of the expected type.
    Raises:
      This function should not raise any Exception.
    """
    ... Your code ...
```

this is a good use case for decorators...

Examples

```
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]])
theta = np.array([[7.01], [3], [10.5], [-6]])

# Example 1.1:
reg_linear_grad(y, x, theta, 1)
# Output:
array([[ -60.99      ],
                [-195.64714286],
                [ 863.46571429],
                [-644.52142857]])

# Example 1.2:
vec_reg_linear_grad(y, x, theta, 1)
# Output:
array([[ -60.99      ],
                [-195.64714286],
                [ 863.46571429],
                [-644.52142857]])

# Example 2.1:
reg_linear_grad(y, x, theta, 0.5)
# Output:
array([[ -60.99      ],
                [-195.86142857],
                [ 862.71571429],
                [-644.09285714]])

# Example 2.2:
vec_reg_linear_grad(y, x, theta, 0.5)
# Output:
array([[ -60.99      ],
                [-195.86142857],
                [ 862.71571429],
                [-644.09285714]])

# Example 3.1:
reg_linear_grad(y, x, theta, 0.0)
# Output:
array([[ -60.99      ],
                [-196.07571429],
                [ 861.96571429],
                [-643.66428571]])

# Example 3.2:
vec_reg_linear_grad(y, x, theta, 0.0)
# Output:
array([[ -60.99      ],
                [-196.07571429],
                [ 861.96571429],
                [-643.66428571]])
```

Chapter VII

Exercise 05

	Exercise : 05
Regularized Logistic Gradient	
Turn-in directory : *ex*05/	
Files to turn in : `reg_logistic_grad.py`	
Forbidden functions : `sklearn`	

Objective

You must implement the following formulas as functions for the **logistic regression hypothesis**

Iterative

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

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

Where:

- $\nabla(J)_j$ is the j$^{\text{th}}$ component of $\nabla(J)$
- $\nabla(J)$ is a vector of dimension $(n+1)$, the gradient vector
- m is a constant, the number of training examples used
- $h_\theta(x^{(i)})$ is the model's prediction for the i$^{\text{th}}$ training example
- $x^{(i)}$ is the feature vector of dimension (n) of the i$^{\text{th}}$ training example, found in the i$^{\text{th}}$ row of the X matrix
- X is a matrix of length $(m \times n)$, the design matrix

- $y^{(i)}$ is the i$^{\text{th}}$ component of the y vector
- y is a vector of dimension m, the vector of expected values
- λ is a constant, the regularization hyperparameter
- θ_j is the j$^{\text{th}}$ parameter of the θ vector
- θ is a vector of dimension $(n+1)$, the parameter vector

Vectorized

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

Where:

- $\nabla(J)$ is a vector of dimension $(n+1)$, the gradient vector
- m is a constant, the number of training examples used
- 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 ones is added as a first column
- X'^T is the transpose of tha matrix, with dimensions $((n+1) \times m)$
- $h_\theta(X)$ is a vector of dimension m, the vector of predicted values
- y is a vector of dimension m, the vector of expected values
- λ is a constant, the regularization hyperparameter
- θ is a vector of dimension $(n+1)$, the parameter vector
- θ' is a vector of dimension $(n+1)$, constructed using the following rules:

$$\begin{aligned} \theta'_0 &= 0 \\ \theta'_j &= \theta_j \quad \text{for } j = 1, \dots, n \end{aligned}$$

Instructions

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

```python
def reg_logistic_grad(y, x, theta, lambda_):
        """Computes the regularized logistic gradient of three non-empty numpy.ndarray, with two for-loops.
        The three arrays must have compatible shapes.
        Args:
                y: has to be a numpy.ndarray, a vector of shape m * 1.
                x: has to be a numpy.ndarray, a matrix of dimesion m * n.
                theta: has to be a numpy.ndarray, a vector of shape n * 1.
                lambda_: has to be a float.
        Returns:
                A numpy.ndarray, a vector of shape n * 1, containing the results of the formula for all j.
                None if y, x, or theta are empty numpy.ndarray.
                None if y, x or theta does not share compatibles shapes.
        Raises:
                This function should not raise any Exception.
        """
        ... Your code ...

def vec_reg_logistic_grad(y, x, theta, lambda_):
        """Computes the regularized logistic gradient of three non-empty numpy.ndarray, without
        any for-loop. The three arrays must have compatible shapes.
        Args:
                y: has to be a numpy.ndarray, a vector of shape m * 1.
                x: has to be a numpy.ndarray, a matrix of shape m * n.
                theta: has to be a numpy.ndarray, a vector of shape n * 1.
                lambda_: has to be a float.
        Returns:
                A numpy.ndarray, a vector of shape n * 1, containing the results of the formula for all j.
                None if y, x, or theta are empty numpy.ndarray.
                None if y, x or theta does not share compatibles shapes.
        Raises:
                This function should not raise any Exception.
        """
        ... Your code ...
```

this is a good use case for decorators...

Examples

```
x = np.array([[0, 2, 3, 4],
                                [2, 4, 5, 5],
                                [1, 3, 2, 7]])
y = np.array([[0], [1], [1]])
theta = np.array([[-2.4], [-1.5], [0.3], [-1.4], [0.7]])

# Example 1.1:
reg_logistic_grad(y, x, theta, 1)
# Output:
array([[-0.55711039],
                [-1.40334809],
                [-1.91756886],
                [-2.56737958],
                [-3.03924017]])

# Example 1.2:
vec_reg_logistic_grad(y, x, theta, 1)
# Output:
array([[-0.55711039],
                [-1.40334809],
                [-1.91756886],
                [-2.56737958],
                [-3.03924017]])

# Example 2.1:
reg_logistic_grad(y, x, theta, 0.5)
# Output:
array([[-0.55711039],
                [-1.15334809],
                [-1.96756886],
                [-2.33404624],
                [-3.15590684]])

# Example 2.2:
vec_reg_logistic_grad(y, x, theta, 0.5)
# Output:
array([[-0.55711039],
                [-1.15334809],
                [-1.96756886],
                [-2.33404624],
                [-3.15590684]])

# Example 3.1:
reg_logistic_grad(y, x, theta, 0.0)
# Output:
array([[-0.55711039],
                [-0.90334809],
                [-2.01756886],
                [-2.10071291],
                [-3.27257351]])

# Example 3.2:
vec_reg_logistic_grad(y, x, theta, 0.0)
# Output:
array([[-0.55711039],
                [-0.90334809],
                [-2.01756886],
                [-2.10071291],
                [-3.27257351]])
```

Chapter VIII

Exercise 06

Interlude - Next Level: Ridge Regression

Until now we only talked about L_2 regularization and its implications on the calculation of the loss function and gradient for both the linear and the logistic regression.

Now it's time to use the proper terminology :

When we apply L_2 regularization on a linear regression model, the new model is called a ***Ridge Regression*** *model. Besides that brand-new name, Ridge regression is nothing more than linear regression regularized with L_2.*

We suggest you watch this very nice explanation of Ridge Regularization.

By the way, this Youtube channel, ***StatQuest***, is very helpful to understand the gist of a lot of machine learning concepts.
You will not waste your time watching its statistics and machine learning playlists!

	Exercise : 06
Ridge Regression	
Turn-in directory : *ex*06/	
Files to turn in : `ridge.py`	
Forbidden functions : `sklearn`	

Objective

Now it's time to implement your `MyRidge` class, similar to the class of the same name in `sklearn.linear_model`.

Instructions

In the `ridge.py` file, create the following class as per the instructions given below:

```
class MyRidge(ParentClass):
        """
        Description:
                My personnal ridge regression class to fit like a boss.
        """
        def __init__(self,  thetas, alpha=0.001, max_iter=1000, lambda_=0.5):
                self.alpha = alpha
                self.max_iter = max_iter
                self.thetas = thetas
                self.lambda_ = lambda_
                ... Your code here ...

        ... other methods ...
```

Your `MyRidge` class will have at least the following methods:

- `__init__`, special method, similar to the one you wrote in `MyLinearRegression` (module06)
- `get_params_`, which gets the parameters of the estimator
- `set_params_`, which sets the parameters of the estimator
- `loss_`, which returns the loss between 2 vectors (numpy arrays)
- `loss_elem_`, which returns a vector corresponding to the squared diffrence between 2 vectors (numpy arrays)
- `predict_`, which generates predictions using a linear model
- `gradient_`, which calculates the vectorized regularized gradient
- `fit_`, which fits Ridge regression model to a training dataset

You should consider inheritance from MyLinearRegression.

If `MyRidge` inheritates from `MyLinearRegression`, you may not need to reimplement the `predict_` method.

The difference between the `MyRidge`'s implementations of `loss_elem_`, `loss_`, `gradient_` and `fit_` and the ones in your `MyLinearRegression` class (implemented in module 02) is the use of a regularization term.

again, this is a good use case for decorators...

Chapter IX

Exercise 07

	Exercise : 07
Practicing Ridge Regression	
Turn-in directory : *ex*07/	
Files to turn in : `space_avocado.py, benchmark_train.py,` `models.[csv/yml/pickle]`	
Forbidden functions : `sklearn`	

Objective

It's training time! Let's practice our brand new Ridge Regression with a polynomial model.

Introduction

You have already used the dataset `space_avocado.csv`. The dataset is made of 5 columns:

- **index**: not relevant
- **weight**: the avocado weight order (in tons)
- **prod_distance**: distance from where the avocado ordered is produced (in Mkms)
- **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 purchases 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, a cross-validation and a test sets
- Use your `polynomial_features` method on your training set
- Consider several Linear Regression models with polynomial hypotheses with a maximum degree of 4
- For each hypothesis consider a regularized factor ranging from 0 to 1 with a step of 0.2
- Evaluate your models on the cross-validation set
- Evaluate the best model on the test set

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

- Plot the evaluation curve which will help you to select the best model (evaluation metrics vs models + λ factor).
- Plot the true price and the predicted price obtained via your best model with the different λ values (meaning the dataset + the 5 predicted curves).

`The training of all your models can take a long time.`
`Therefore you need to train only the best one during the correction.`

Nevertheless, you should return in `benchmark_train.py` the program which performs the training of all the models and saves 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 the best model on the right set and plot the different graphics as asked before.

Chapter X

Exercise 08

Interlude

Regularized Logistic Regression is still Logistic Regression

As opposed to linear regression, **regularized logistic regression is still called logistic regression**.

Working without regularization parameters can simply be regarded as a special case where $\lambda = 0$.

if $\lambda = 0$:

$$\begin{aligned}\nabla(J) &= \frac{1}{m}[X'^T(h_\theta(X) - y) + \lambda\theta'] \\ &= \frac{1}{m}[X'^T(h_\theta(X) - y) + 0 \cdot \theta'] \\ &= \frac{1}{m}[X'^T(h_\theta(X) - y)]\end{aligned}$$

Exercise : 08

Regularized Logistic Regression

Turn-in directory : *ex*08/

Files to turn in : `my_logistic_regression.py`

Forbidden functions : `sklearn`

Objective

In the last exercise, you implemented a regularized version of the linear regression algorithm, called Ridge regression.

Now it's time to update your logistic regression classifier as well!

In the `scikit-learn` library, the logistic regression implementation offers a few regularization techniques, which can be selected using the parameter `penalty` (L_2 is default). The goal of this exercise is to update your old `MyLogisticRegression` class to take that into account.

Instructions

In the `my_logistic_regression.py` file, update your `MyLogisticRegression` class according to the following instructions:

```
class MyLogisticRegression():
        """
        Description:
                My personnal logistic regression to classify things.
        """
  supported_penalities = ['l2'] #We consider l2 penalities only. One may want to implement other penalities

        def __init__(self, theta, alpha=0.001, max_iter=1000, penality='l2', lambda_=1.0):
                # Check on type, data type, value ... if necessary
    self.alpha = alpha
                self.max_iter = max_iter
                self.theta = theta
                self.penality = penality
    self.lambda_ = lambda_ if penality in self.supported_penalities else 0
                #... Your code ...

        ... other methods ...
```

- **add** a `penalty` parameter which can take the following values:`'l2'`, `'none'` (default value is `'l2'`).
- **update** the `fit_(self, x, y)` method:
  - `if penality == 'l2'`: use a **regularized version** of the gradient descent.

- `if penality = ’none’`: use the **unregularized version** of the gradient descent from `module03`.

Examples

```
from my_logistic_regression import MyLogisticRegression as mylogr

theta = np.array([[-2.4], [-1.5], [0.3], [-1.4], [0.7]])

# Example 1:
model1 = mylogr(theta, lambda_=5.0)

model1.penality
# Output
’l2’

model1.lambda_
# Output
5.0

# Example 2:
model2 = mylogr(theta, penality=None)

model2.penality
# Output
None

model2.lambda_
# Output
0.0

# Example 3:
model3 = mylogr(theta, penality=None, lambda_=2.0)

model3.penality
# Output
None

model3.lambda_
# Output
0.0
```

`this is also a great use case for decorators...`

Chapter XI

Exercise 09

	Exercise : 09
Practicing Regularized Logistic Regression	
Turn-in directory : *ex*09/	
Files to turn in : `solar_system_census.py, benchmark_train.py, models.[csv/yml/pickle]`	
Forbidden functions : `sklearn`	

Objective

It's training time! Let's practice our updated Logistic Regression with polynomial models.

Introduction

You have already used the dataset `solar_system_census.csv` and `solar_system_census_planets.cs`

- 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)

Instructions

Split the Data

Take your `solar_system_census.csv` dataset and split it into a **training set**, a **cross-validation set** and a **test set**.

Training and benchmark

One part of your submission will be located in the `benchmark_train.py` and `models.[csv/yml/pickle]` files. You have to:

- Train different regularized logistic regression models with a polynomial hypothesis of **degree 3**. The models will be trained with different λ values, ranging from 0 to 1. Use the one-vs-all method.
- Evaluate the **f1 score** of each of the models on the cross-validation set. You can use the `f1_score_` function that you wrote in the `ex11` of `module08`.
- Save the different models into a `models.[csv/yml/pickle]`.

Solar system census program

The second and last part of your submission is in `solar_system_census.py`. You have to:

- Load the differents models from `models.[csv/yml/pickle]` and train from scratch only the best one on a training set.
- Visualize the performance of the different models with a bar plot showing the score of the models given their λ value.
- Print the **f1 score** of all the models calculated on the test set.
- Visualize the target values and the predicted values of the best model on the same scatterplot. Make some efforts to have a readable figure.

i For the second script `solar_system_census.py`, only a train and test set are necessary as one is simply looking at the performance.

Chapter XII

Conclusion - What you have learnt

This Machine Learning Bootcamp is over, congratulations !

Based on all the notions and problems tackled today, you should be able to discuss and answer the following questions:

1. Why do we use the 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 does considering more examples (for example, extra data points with extreme ordinate) can alter the classification of some data points ?

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

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

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

i

`Your feedbacks are essential for us to improve these bootcamps ! Please take a few minutes to tell us about your experience in this module by filling this form. Thank you in advance !`

Contact

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

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

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 of 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 in the creation and development of these modules.

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