Python Machine-Learning

42 ARTIFICIAL INTELLIGENCE

Module09 Regularization

Module09 - Regularization

Today you will fight overfitting!
You will discover the concepts of regularization and how to implement it into the algortihms you already saw until now.

Notions of the module

Regularization, overfitting. Regularized cost function, regularized gradient descent.
Regularized linear regression. Regularized logistic regression.

Useful Ressources

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

Week 3:

Solving the Problem of Overfitting:

- 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)

General rules

- The Python version to use is 3.7, you can check with the following command: `python -V`
- The norm: during this bootcamp you will follow the Pep8 standards
- The function `eval` is never allowed.
- The exercises are ordered from the easiest to the hardest.
- Your exercises are going to be evaluated by someone else, so make sure that your variable names and function names are appropriate and civil.
- Your manual is the internet.
- You can also ask questions in the `#bootcamps` channel in 42AI's Slack workspace.
- If you find any issues or mistakes in this document, please create an issue on our dedicated Github repository.

Helper

Ensure that you have the right Python version.

```
> which python
/goinfre/miniconda/bin/python
> python -V
Python 3.7.*
```

```
> which pip
/goinfre/miniconda/bin/pip
```

Exercise 00 - Logistic Regression

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

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.

Objectives:

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 had 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, thetas, alpha=0.001, n_cycle=1000):
        self.alpha = alpha
        self.max_iter = max_iter
        self.thetas = thetas
        # Your code here

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

You will add the following methods:

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

You have already written these functions, you will just need a 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]])
mylr = MyLR([2, 0.5, 7.1, -4.3, 2.09])

# Example 0:
```

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

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

# Example 2:
mylr.fit_(X, Y)
mylr.thetas
# Output:
array([[ 1.04565272],
       [ 0.62555148],
       [ 0.38387466],
       [ 0.15622435],
       [-0.45990099]])

# Example 3:
mylr.predict_(X)
# Output:
array([[0.72865802],
       [0.40550072],
       [0.45241588]])

# Example 4:
mylr.cost_(X,Y)
# Output:
0.5432466580663214
```

Exercise 01 - Polynomial models

Turn-in directory :	ex01
Files to turn in :	polynomial_model.py
Authorized modules :	numpy
Forbidden modules :	sklearn

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.

Objectives:

Create a function that takes a vector x of dimension $m * 1$ and an integer n as input, and returns a matrix of dimension $m * n$.
Each column of the matrix contains x raised to the power of j, for $j = 1, 2, ..., n$:

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

Instructions:

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

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

Examples:

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

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


# Example 2:
add_polynomial_features(x, 6)
```

```
# Output:
array([[    1,      1,      1,      1,      1,      1],
       [    2,      4,      8,     16,     32,     64],
       [    3,      9,     27,     81,    243,    729],
       [    4,     16,     64,    256,   1024,   4096],
       [    5,     25,    125,    625,   3125,  15625]])
```

Exercise 02 - AI Key Notions:

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

Are you able to clearly and simply explain:

1 - What is overfitting?

2 - What do you think underfitting might be?

3 - Why is it important to split the data set in a training and a test set?

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

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

Exercise 03 - Polynomial models II

Turn-in directory :	ex03
Files to turn in :	polynomial_model_extended.py
Authorized modules :	numpy
Forbidden modules :	sklearn

Objectives:

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

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, 5)
# 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]])
```

Interlude - Fighting Overfitting... With Regularization

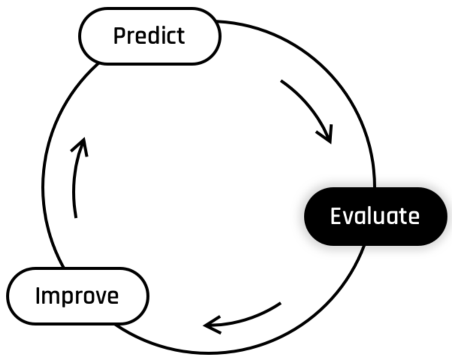

Figure 1: The Learning Cycle - Evaluate

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 you to 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 beahind regularization is to **penalize the model for putting too much weight on certain** (usually heavy polynomial) **features**. We do this by adding an extra term in the cost function:

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

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 cost.

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.

Please notice: 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 carefull!

Machine Learning was essentially developed 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 completly messes up 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 that you keep seeing that X' is of dimension $m * (n+1)$

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 cost 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 the 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.

Note:

$$\text{the notation } \sum_i \text{ means: "the sum for all } i\text{"}$$

There is no need to give explicitly 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 because it forces us to be 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 not a 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.

Interlude - 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:

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

Exercise 04 - L2 Regularization

Turn-in directory :	ex04
Files to turn in :	l2_reg.py
Authorized modules :	Numpy
Forbidden modules :	sklearn

Objectives:

You must implement the following formulas as functions:

I- Iterative:

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

Where:

- θ is a vector of dimension n * 1.

II - Vectorized:

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

Where:

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

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

Instructions:

In the `l2_reg.py` file, write the following function 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 dimension 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.
    """

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 dimension 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.
    """
```

Examples

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

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

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

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

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

Exercise 05 - Regularized Linear Cost Function

Turn-in directory :	ex05
Files to turn in :	linear_cost_reg.py
Authorized modules :	numpy
Forbidden modules :	sklearn

Objectives:

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 $m * 1$, the expected values,
- $\hat{y}$ is a vector of dimension $m * 1$, the predicted values,
- λ is a constant, the regularization hyperparameter,
- 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 `linear_cost_reg.py` file, write the following function as per the instructions given below:

```
def reg_cost_(y, y_hat, theta, lambda_):
    """Computes the regularized cost of a linear regression model from 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 of dimension m * 1.
      y_hat: has to be an numpy.ndarray, a vector of dimension m * 1.
      theta: has to be a numpy.ndarray, a vector of dimension n * 1.
      lambda_: has to be a float.
    Returns:
      The regularized cost 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 dimensions.
    Raises:
      This function should not raise any Exception.
    """
```

Hint: such situation is a good use case of decorators...

Examples

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

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

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

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

Exercise 06 - Regularized Logistic Cost Function

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

Objectives:

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 $m * 1$, the vector of predicted values,
- y is a vector of dimension $m * 1$, the vector of expected values,
- $\vec{1}$ is a vector of dimension $m * 1$, a vector full of ones,
- λ is a constant, the regularization hyperparameter,
- θ' is a vector of dimension $n * 1$, constructed using the following rules:

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

Instructions:

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

```
def reg_log_cost_(y, y_hat, theta, lambda_):
    """Computes the regularized cost of a logistic regression model from 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 of dimension m * 1.
      y_hat: has to be an numpy.ndarray, a vector of dimension m * 1.
      theta: has to be a numpy.ndarray, a vector of dimension n * 1.
      lambda_: has to be a float.
    Returns:
      The regularized cost 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 dimensions.
    Raises:
      This function should not raise any Exception.
    """
```

Hint: this is a great occasion to practice using decorators...

Examples

```
y = np.array([1, 1, 0, 0, 1, 1, 0])
y_hat = np.array([.9, .79, .12, .04, .89, .93, .01])
```

```
theta = np.array([1, 2.5, 1.5, -0.9])

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

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

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

Interlude - Regularized Gradient

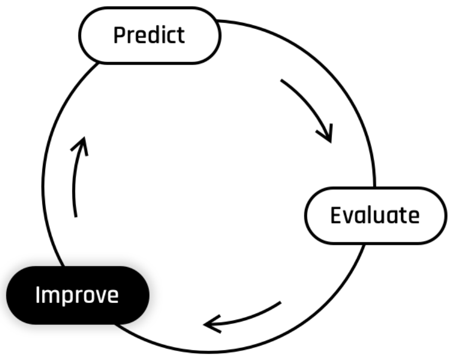

Figure 2: The Learning Cycle - Improve

To derive the gradient of the regularized cost 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^{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^{th} training example,
- $x^{(i)}$ is the feature vector of the i^{th} training example,
- $y^{(i)}$ is the expected target value for the i^{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 $(n+1)*1$, the gradient vector,
- m is the number of training examples used,
- X is a matrix of dimension $m*n$, the design matrix,

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

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

Linear Gradient vs Logistic Gradient

As before, we draw your attention on the only difference between linear regression and 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 07 - Regularized Linear Gradient

Turn-in directory :	ex07
Files to turn in :	reg_linear_grad.py
Authorized modules :	numpy
Forbidden modules :	sklearn

Objectives

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

I - 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^{th} component of $\nabla(J)$,
- $\nabla(J)$ is a vector of dimension $(n+1)*1$, 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^{th} training example,
- $x^{(i)}$ is the feature vector (of dimension $n*1$) of the i^{th} training example, found in the i^{th} row of the X matrix,
- X is a matrix of dimension $m*n$, the design matrix,
- $y^{(i)}$ is the i^{th} component of the y vector,
- y is a vector of dimension $m*1$, the vector of expected values,
- λ is a constant, the regularization hyperparameter,
- θ_j is the j^{th} parameter of the θ vector,
- θ is a vector of dimension $(n+1)*1$, the parameter vector,

II - Vectorized:

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

Where:

- $\nabla(J)$ is a vector of dimension $(n+1)*1$, the gradient vector,
- m is a constant, the number of training examples used,
- X is a matrix of dimension $m*n$, the design matrix,
- X' is a matrix of dimension $m*(n+1)$, the design matrix onto which a column of ones is added as a first column,
- X'^T is the transpose of tha matrix, with dimensions $(n+1)*m$,

- $h_\theta(X)$ is a vector of dimension $m * 1$, the vector of predicted values,
- y is a vector of dimension $m * 1$, the vector of expected values,
- λ is a constant, the regularization hyperparameter,
- θ is a vector of dimension $(n+1) * 1$, 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:

```
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 dimensions.
    Args:
      y: has to be a numpy.ndarray, a vector of dimension 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 dimension n * 1.
      lambda_: has to be a float.
    Returns:
      A numpy.ndarray, a vector of dimension 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 dimensions.
    Raises:
      This function should not raise any Exception.
    """

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 dimensions.
    Args:
      y: has to be a numpy.ndarray, a vector of dimension 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 dimension n * 1.
      lambda_: has to be a float.
    Returns:
      A numpy.ndarray, a vector of dimension 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 dimensions.
    Raises:
      This function should not raise any Exception.
    """
```

Hint: this is a great occasion to practice using 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]])
```

Exercise 08 - Regularized Logistic Gradient

Turn-in directory :	ex08
Files to turn in :	reg_logistic_grad.py
Authorized modules :	Numpy
Forbidden modules :	sklearn

Objectives

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

I - 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^{th} component of $\nabla(J)$,
- $\nabla(J)$ is a vector of dimension $(n+1)*1$, 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^{th} training example,
- $x^{(i)}$ is the feature vector (of dimension $n*1$) of the i^{th} training example, found in the i^{th} row of the X matrix,
- X is a matrix of dimension $m*n$, the design matrix,
- $y^{(i)}$ is the i^{th} component of the y vector,
- y is a vector of dimension $m*1$, the vector of expected values,
- λ is a constant, the regularization hyperparameter,
- θ_j is the j^{th} parameter of the θ vector,
- θ is a vector of dimension $(n+1)*1$, the parameter vector,

II - Vectorized:

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

Where:

- $\nabla(J)$ is a vector of dimension $(n+1)*1$, the gradient vector,
- m is a constant, the number of training examples used,
- X is a matrix of dimension $m*n$, the design matrix,
- X' is a matrix of dimension $m*(n+1)$, the design matrix onto which a column of ones is added as a first column,
- X'^T is the transpose of tha matrix, with dimensions $(n+1)*m$,

- $h_\theta(X)$ is a vector of dimension $m * 1$, the vector of predicted values,
- y is a vector of dimension $m * 1$, the vector of expected values,
- λ is a constant, the regularization hyperparameter,
- θ is a vector of dimension $(n + 1) * 1$, 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:

```
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 dimensions.
    Args:
      y: has to be a numpy.ndarray, a vector of dimension 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 dimension n * 1.
      lambda_: has to be a float.
    Returns:
      A numpy.ndarray, a vector of dimension 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 dimensions.
    Raises:
      This function should not raise any Exception.
    """

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 dimensions.
    Args:
      y: has to be a numpy.ndarray, a vector of dimension 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 dimension n * 1.
      lambda_: has to be a float.
    Returns:
      A numpy.ndarray, a vector of dimension 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 dimensions.
    Raises:
      This function should not raise any Exception.
    """
```

Hint: this is a great occasion to practice using 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]])
```

Interlude - Linear Regression to the Next Level: Ridge Regression

Until now we only talked about L2 regularization and its implication on the calculation of the cost function and gradient for both linear and logistic regression.

Now it's time to use proper terminology:
When we apply L2 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 that: linear regression regularized with L2.

We suggest that you watch this nice explanation very nice explanation of Ridge Regularization.
By the way, this Youtube channel, **StatQuest**, is very good to help you understand the gist of a lot of machine learning concepts.
You will not waste your time watching its statistics and machine learning playlists!

Exercise 09 - Ridge Regression

Turn-in directory :	ex09
Files to turn in :	ridge.py
Authorized modules :	numpy
Forbidden modules :	sklearn

Objectives:

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:

Your `MyRidge` class will have several methods:

- `__init__` , special method, identical to the one you wrote in `MyLinearRegression` (Day01),
- `get_params_` , which get the parameters of the estimator,
- `set_params_` , which set the parameters of the estimator,
- `predict_` , which generates predictions using a linear model,
- `fit_` , which fits Ridge regression model to a training dataset.

Except for `fit_`, the methods are identical to the ones in your `MyLinearRegression` class.
You should consider inheritance

The difference between `MyRidge`'s `fit_` method and the `fit_` method you implemented for your `MyLinearRegression` class is the use of a regularization term.

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

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

Hint: again, this is a great occasion for you to try using decorators...

Exercise 10 - Practicing Ridge Regression

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

Objectives:

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

Instructions:

Part 1: Data Splitting

Take your `spacecraft_data.csv` dataset and split it in a **training** and a **test** set.

Part 2: Training

- You will train 10 different models on the training set: **one** Linear Regression model and **nine** Ridge Regression models. All 10 models will use a polynomial hypothesis of **degree 3**. The Ridge Regression models will be trained with different λ values, ranging from 0.1 up to 1 (with increments of 0.1).
- Score the performance of each of the 10 models on the **test set** with the **Mean Squared Error** metric. You can use the `mse` function that you implemented in the `ex11` of `module05`.
- To properly visualize your results, make a bar plot showing the MSE score of the models given their λ value.

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

Part 3: Plots

- For each model you built in Part 2, plot its hypothesis function $h(\theta)$ on top of a scatter plot of the original data points (x, y).

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 be considered simply as a special case where $\lambda = 0$.

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

Exercise 11 - Regularized Logistic Regression

Turn-in directory :	ex11
Files to turn in :	my_logistic_regression.py
Authorized modules :	Numpy
Forbidden modules :	sklearn

Objectives:

In the last exercice, you implemented of 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` (L2 is default).

The goal of this exercice 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 :

- **add** a `penalty` parameter wich can take the following values:
  `{'l2', 'none'}, default = 'l2'`.

```
class MyLogisticRegression():
    """
    Description:
        My personnal logistic regression to classify things.
    """
    def __init__(self, theta, alpha=0.001, n_cycle=1000, penalty='l2'):
        self.alpha = alpha
        self.max_iter = max_iter
        self.theta = theta
        self.penalty=penalty
        # Your code here

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

- **update** the `fit_(self, x, y)` method:
  - `if penalty == 'l2'`:
    use a **regularized version** of the gradient descent.
  - `if penalty = 'none'`:
    use the **unregularized version** of the gradient descent from module08.

Hint: this is also a great use case for decorators...

Exercise 12 - Practicing Regularized Logistic Regression

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

Objectives:

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

Instructions:

Part 1: Split the Data

Take your `solar_system_census.csv` dataset and split it in a **training set** and a **test set**.

Part 2: Train different models

- Train **ten** different 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 one-vs-all method.
- Evaluate the **f1 score** of each of the ten models on the test set. You can use the `f1_score_` function that you wrote in the `ex11` of `module08`.
- To properly visualize your results, make a bar plot showing the score of the models given their λ value.

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

One Last Word - It's Just a Beginning. . .

Congratulation!!

You have finished this bootcamp and you can be proud of yourself!
We hope you liked it and that the material were understandable.

We tried our best to make it as accessible as possible to anyone, even for someone with little mathematical background. It was quite a challenge, and we hope we succeed to that difficult mission.

Equiped with your brand-new knowledge you are now able to tackle more challenging algorithm like **ensemble methods (random forest, gradient boosting)**, **support vector machine** or even **artificial neural networks !!**

An because we know that **a lot of you have neural networks in mind** when you started this journey into machine learning, let's talk a bit more about why you are now able to deep dive into it. . . fearlessly!

Neural networks are based on the same blocks you should now be familiar with. Essentially:

- matrix and vector operations,
- gradient descent,
- regularization,
- sigmoid (as activation functions, even if it is a bit outdated now)

Let's see what you can do now.

To go further

To keep learning Machine Learning, here are several options you should consider:

- To complete the entire Stanford's Machine Learning MOOC. It is a great ressource, **a classic** for those who want to study machine learning. This bootcamp followed thigthly the architecture of its first three weeks. This course is definitely worth your time! Also, someone did a great work to convert all the Octave assignments into Python notebooks.
- To take fast.ai Deep Learning MOOC. It's a great way to learn Deep Learning following a top-down approach.