Bootcamp
Machine Learning

Bootcamp ML

Day03 - Regularization and Feature engineering

Welcome to the day03 of the machine learning bootcamp ! Today you will see the L2 regularization and how to implement it in both linear and logistic regression. Then we will see few ways to process your data in order to improve significantly the performance of your models.

Notions of the day

- Regularization
- Regularized linear regression
- Regularized logistic regression
- z-score standardization
- min-max standardization
- Polynomial features
- Interaction terms

General rules

- The version of Python to use is 3.7.x, you can check the version of Python with the following command: python -V
- The norm: during this bootcamp you will follow the Pep8 standards https://www.python.org/dev/peps/pep-0008/
- 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 variables and functions names are appropriated.
- Your man is internet.

- You can also ask question in the dedicated channel in Slack: 42-ai.slack.com.
- If you find any issue or mistakes in the subject please create an issue on our dedicated repository on Github: https://github.com/42-AI/bootcamp_machine-learning/issues.

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
```

Mathematical delights (continued)

Exercice 00 - Ridge - iterative version

Exercice 01 - Ridge - vectorized version

Exercice 02 - Regularized MSE

Exercice 03 - Regularized Linear Gradient - iterative version

Exercice 04 - Regularized Linear Gradient - vectorized version

Exercice 05 - Regularized Logistic Loss

Exercice 06 - Regularized Logistic Gradient - iterative version

Exercice 07 - Regularized Logistic Gradient - vectorized version

Algorithm

Exercice 08 - Ridge regression

Exercice 09 - Regularized Logistic regression

Feature Engineering

Exercice 10 - Z-score standardization

Exercice 11 - Min-max standardization

Exercice 12 - Polynomial features + interaction terms

Exercise 00 - Regularization - iterative version

Turnin directory :	ex00
Files to turn in :	reg.py
Forbidden function :	None
Remarks :	n/a

You must implement the following formula as a function:

$$\lambda \sum_{j=1}^{n} \theta_j^2$$

Where

- θ is a vector of dimensions n * 1,
- λ is a constant

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

```
def regularization(theta, lambda_):
    """Computes the regularization term of a non-empty numpy.ndarray,
without any for-loop.    Args:
     theta: has to be a numpy.ndarray, a vector of dimension n * 1.
     lambda: has to be a float.
    Returns:
     The regularization term of theta.
     None if theta is an empty numpy.ndarray.
    Raises:
     This function should not raise any Exception.
    """
```

Examples

```
>>> X = np.array([0, 15, -9, 7, 12, 3, -21])
>>> regularization(X, 0.3)
40.67142857142857
>>> regularization(X, 0.01)
1.3557142857142856
>>> regularization(X, 0)
0.0
```

Exercise 01 - Regularization - vectorized version

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

You must implement the following formula as a function:

$$\lambda\theta \cdot \theta = \lambda\theta^T\theta$$

Where

- θ is a vector of dimension n * 1,
- λ is a constant

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

```
def vectorized_regularization(theta, lambda_):
    """Computes the regularization term of a non-empty numpy.ndarray,
without any for-loop.    Args:
     theta: has to be a numpy.ndarray, a vector of dimension n * 1.
     lambda: has to be a float.
    Returns:
     The regularization term of theta.
     None if theta is an empty numpy.ndarray.
    Raises:
     This function should not raise any Exception.
    """
```

Examples

```
>>> X = np.array([0, 15, -9, 7, 12, 3, -21])
>>> vectorized_regularization(X, 0.3)
40.67142857142857
>>> vectorized_regularization(X, 0.01)
1.3557142857142856
>>> vectorized_regularization(X, 0)
0.0
```

Exercise 02 - Regularized Mean Squared Error

Turnin directory :	ex02
Files to turn in :	reg_mse.py
Forbidden function :	None
Remarks :	n/a

Objective:
You must implement the following formula as a function:

$$regMSE = \frac{1}{m}\left((X\theta - y)^T(X\theta - y) + \lambda\theta \cdot \theta\right)$$

where:

- X is a matrix of dimension m * n,
- θ is a vector of dimension n * 1,
- y is a vector of dimension m * 1,
- λ is a constant,

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

```
def reg_mse(y, x, theta, lambda_):
    """Computes the regularized mean squared error 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:
     The mean squared error as a float.
     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.
    """
```

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])
>>> Z = np.array([3,0.5,-6])
>>> reg_mse(X, Y, Z, 0)
2641.0
>>> reg_mse(X, Y, Z, 0.1)
2642.508333333333
>>> reg_mse(X, Y, Z, 0.5)
2648.5416666666665
```

Exercise 03 - Regularized Linear Gradient - iterative version

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

You must implement the following formula as a function:

$$\nabla(J)_0 = \frac{\alpha}{m}\sum_{i=1}^{m}(h_\theta(x_i) - y_i)x_i^{(j)}$$

$$\nabla(J)_j = \frac{\alpha}{m}\sum_{i=1}^{m}(h_\theta(x_i) - y_i)x_i^{(j)} + \lambda\theta_j$$

where

- $\nabla(J)$ is a vector of dimension n * 1
- x is a matrix of dimension m * n (i.e. a matrix containing m vectors of dimension n * 1)
- y is a vector of dimension m * 1
- θ is a vector of dimension n * 1
- x_i is the ith component of x, a vector of dimension n * 1
- y_i is the ith component of y
- $\nabla(J)_j$ is the jth component of $\nabla(J)$
- $h_\theta(x_i)$ is the result of the dot product of the vector θ and the vector x_i
- α is a constant
- λ is a constant

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

```
def reg_linear_grad(y, x, theta, alpha, 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.
     alpha: has to be a float.
     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.
    """
```

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])
>>> Z = np.array([3,0.5,-6])
>>> reg_linear_grad(X, Y, Z, 0.5, 0.5)
array([ -18.67857143, 91.58928571, -196.71428571])
>>> reg_linear_grad(X, Y, Z, 0.1, 0.5)
array([ -3.73571429, 18.31785714, -39.34285714])
>>> reg_linear_grad(X, Y, Z, 0.5, 0.1)
array([ -18.67857143, 91.575     , -196.54285714])
```

Exercise 04 - Regularized Linear Gradient - vectorized version

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

You must implement the following formula as a function:

$$\nabla(J) = \frac{\alpha}{m} X^T (X\theta - y)$$

$$\nabla(J)_0 := \nabla(J)_0$$
$$\nabla(J)_j := \nabla(J)_j + \frac{\lambda}{m}\theta_j$$

where

- $\nabla(J)$ is a vector of dimension n * 1
- X is a matrix of dimension m * n (i.e. a matrix containing m vectors of dimension n * 1)
- y is a vector of dimension m * 1
- θ is a vector of dimension n * 1
- $\nabla(J)_j$ is the jth component of $\nabla(J)$
- α is a constant
- λ is a constant

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

```
def vec_reg_linear_grad(y, x, theta, alpha, 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.
     alpha: has to be a float.
     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.
    """
```

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])
>>> Z = np.array([3,0.5,-6])
>>> vec_reg_linear_grad(X, Y, Z, 0.5, 0.5)
array([ -18.67857143, 91.58928571, -196.71428571])
>>> vec_reg_linear_grad(X, Y, Z, 0.1, 0.5)
array([ -3.73571429, 18.31785714, -39.34285714])
>>> vec_reg_linear_grad(X, Y, Z, 0.5, 0.1)
array([ -18.67857143, 91.575     , -196.54285714])
2641.0
>>> reg_mse(X, Y, Z, 0.1)
2642.508333333333
>>> reg_mse(X, Y, Z, 0.5)
2648.5416666666665
```

Exercise 05 - Regularized Logistic Loss Function

Turning directory :	ex05
Files to turn in :	reg_log_loss.py
Forbidden function :	None
Remarks :	n/a

Objectives:

You must implement the following formula as a function:

$$J(\theta) = \frac{1}{m}\left((-y \cdot \log(h)) - (1-y) \cdot (1 - log(h)) + \lambda\theta \cdot \theta\right)$$

$$J(\theta) = \frac{1}{m}\left((-y^T \log(h)) - (1-y)^T (1 - log(h)) + \lambda\theta \cdot \theta\right)$$

where:

- $h = g(X\theta)$ is a vector of dimension m * 1,
- θ is a vector of dimension n * 1,
- y is a vector of dimension m * 1,
- λ is a constant,

Instructions:

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

```
def reg_log_loss_(y_true, y_pred, m, lambda_):
    """
    Compute the logistic loss value.
    Args:
        y_true: a scalar or a numpy ndarray for the correct labels
        y_pred: a scalar or a numpy ndarray for the predicted labels
        m: the length of y_true (should also be the length of y_pred)
        lambda_: a float for the regularization parameter
        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.
    """
```

Examples:

```
import numpy as np
from sigmoid import sigmoid_
from reg_log_loss import reg_log_loss_

# Test n.1
x_new = np.arange(1, 13).reshape((3, 4))
y_true = np.array([1, 0, 1])
theta = np.array([-1.5, 2.3, 1.4, 0.7])
y_pred = sigmoid_(np.dot(x_new, theta))
m = len(y_true)
print(reg_log_loss_(y_true, y_pred, m, theta, 0.0))
# 7.233346147374828

# Test n.2
x_new = np.arange(1, 13).reshape((3, 4))
y_true = np.array([1, 0, 1])
theta = np.array([-1.5, 2.3, 1.4, 0.7])
y_pred = sigmoid_(np.dot(x_new, theta))
m = len(y_true)
print(reg_log_loss_(y_true, y_pred, m, theta, 0.5))
# 7.441471049799478

# Test n.3
x_new = np.arange(1, 13).reshape((3, 4))
y_true = np.array([1, 0, 1])
theta = np.array([-1.5, 2.3, 1.4, 0.7])
y_pred = sigmoid_(np.dot(x_new, theta))
m = len(y_true)
print(reg_log_loss_(y_true, y_pred, m, theta, 1))

# 8.065846049799477

# Test n.4
x_new = np.arange(1, 13).reshape((3, 4))
y_true = np.array([1, 0, 1])
theta = np.array([-5.2, 2.3, -1.4, 8.9])
y_pred = sigmoid_(np.dot(x_new, theta))
m = len(y_true)
print(reg_log_loss_(y_true, y_pred, m, theta, 1))
# 11.478071825561033

# Test n.5
x_new = np.arange(1, 13).reshape((3, 4))
y_true = np.array([1, 0, 1])
theta = np.array([-5.2, 2.3, -1.4, 8.9])
y_pred = sigmoid_(np.dot(x_new, theta))
m = len(y_true)
print(reg_log_loss_(y_true, y_pred, m, theta, 0.3))
# 12.329321825561033

# Test n.6
x_new = np.arange(1, 13).reshape((3, 4))
y_true = np.array([1, 0, 1])
theta = np.array([-5.2, 2.3, -1.4, 8.9])
y_pred = sigmoid_(np.dot(x_new, theta))
```

```
m = len(y_true)
print(reg_log_loss_(y_true, y_pred, m, theta, 0.9))
# 19.139321825561034
```

Exercise 06 - Regularized Logistic Gradient - iterative version

Turn-in directory :	ex06
Files to turn in :	reg_logistic_grad.py
Forbidden functions :	None
Remarks :	n/a

This exercice is almost the same as the ex03, except that $h_\theta(x_i)$ is now the logistic regression hypothesis instead of the linear regression one.

$h_\theta(x_i) = g(\theta \cdot x_i)$ where g is the sigmoid function applied to the result of $\theta \cdot x_i$.

You must implement the following formula as a function:

$$\nabla(J)_0 = \frac{\alpha}{m}\sum_{i=1}^{m}(h_\theta(x_i) - y_i)x_i^{(j)}$$

$$\nabla(J)_j = \frac{\alpha}{m}\sum_{i=1}^{m}(h_\theta(x_i) - y_i)x_i^{(j)} + \lambda\theta_j$$

where

- $\nabla(J)$ is a vector of dimension n * 1
- x is a matrix of dimension m * n (i.e. a matrix containing m vectors of dimension n * 1)
- y is a vector of dimension m * 1
- θ is a vector of dimension n * 1
- x_i is the ith component of x, a vector of dimension n * 1
- y_i is the ith component of y
- $\nabla(J)_j$ is the jth component of $\nabla(J)$
- $h_\theta(x_i)$ is the result of the sigmoid function applied to the result of the dot product of the vector θ and the vector x_i
- α is a constant
- λ is a constant

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, alpha, 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.
     alpha: has to be a float.
     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.
    """
```

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([1,0,1,1,1,0,0])
>>> Z = np.array([1.2,0.5,-0.32])
>>> reg_logistic_grad(Y, X, Z,, 0.5 0.5)
array([-514754.11712633, -373142.24641195, 423580.96005031])
>>> reg_logistic_grad(Y, X, Z,, 0.9 0.1)
array([-926557.4108274 , -671656.06925579, 762445.7445477 ])
>>> reg_logistic_grad(Y, X, Z,, 0.5 0.5)
array([-102950.82342527, -74628.44642525, 84716.19018149])
```

Exercise 07 - Regularized Logistic Gradient - vectorized version

Turn-in directory :	ex07
Files to turn in :	vec_reg_logistic_grad.py
Forbidden functions :	None
Remarks :	n/a

This exercice is almost the same as the ex04, except that we are replacing the linear regression one $X\theta$ by the logistic regression hypothesis $g(X\theta)$, where g is the sigmoid function applied element-wise to the result of $X\theta$.

You must implement the following formula as a function:

$$\nabla(J) = \frac{\alpha}{m} X^T (g(X\theta) - y)$$

$$\nabla(J)_0 := \nabla(J)_0$$

$$\nabla(J)_j := \nabla(J)_j + \frac{\lambda}{m}\theta_j$$

where

- $\nabla(J)$ is a vector of dimension n * 1
- X is a matrix of dimension m * n (i.e. a matrix containing m vectors of dimension n * 1)
- y is a vector of dimension m * 1
- θ is a vector of dimension n * 1
- g is the sigmoid function applied element-wise to a vector
- $\nabla(J)_j$ is the jth component of $\nabla(J)$
- α is a constant
- λ is a constant

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

```
def vec_reg_logistic_grad(y, x, theta, alpha, 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.
     alpha: has to be a float.
     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.
    """
```

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([1,0,1,1,1,0,0])
>>> Z = np.array([1.2,0.5,-0.32])
>>> vec_reg_logistic_grad(Y, X, Z,, 0.5 0.5)
array([-514754.11712633, -373142.24641195, 423580.96005031])
>>> vec_reg_logistic_grad(Y, X, Z,, 0.9 0.1)
array([-926557.4108274 , -671656.06925579, 762445.7445477 ])
>>> vec_reg_logistic_grad(Y, X, Z,, 0.5 0.5)
array([-102950.82342527, -74628.44642525, 84716.19018149])
```

Exercise 08 - Ridge Regression Method

Turn-in directory :	ex08
Files to turn in :	ridge_reg.py
Forbidden functions :	*.sum()
Forbidden classes :	sklearn.linear_model.Ridge
Authorized classes :	sklearn.model_selection.LeaveOneOut
	sklearn.model_selection.KFold
Remarks :	n/a

Objectives:

- Understand the limit of a hypothesis.
- Understand of the notions of biais and variance.
- Implementation of a class named **MyRidge** similar to the class of the same name in **sklearn.linear_model**.

Part One - Back to the *Future* Past

If I had an hour to solve a problem I would spend 55 minutes thinking about the problem and 5 minutes thinking about the solutions. (Albert Einstein)

Thus, be $lazy$ smart ! This should remind you something you already coded.

Instructions:
For this part, your class **MyRidge** will have several methods:

- __init__ , special method, identical to the one of the class **MyLinearRegression** (Day01),
- get_params_ , which get the parameters of the estimator,
- set_params_ , which set the parameters of the estimator,
- predict_ , which predict output using a linear model,
- fit_ , which fit Ridge regression model
- a few metric methods: mse_ , rmse_ , rscore_ .

Except for `.fit_` the methods are identicals to the ones of your class **MyLinearRegression**.
You should consider inheritance

The difference between the method `.fit_` of **MyRidge** class and the method `.fit_` of **MyLinearRegression** is the use of a regularization term.
The cost function:

$$J(\theta) = \frac{1}{2M}\sum_{i=1}^{M}(h_\theta(x^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2M}\sum_{j=1}^{N}\theta_j^2$$

where:

- M is the number of training examples (or the length of the dataset),
- N is the number of features,
- $x^{(i)}$ is the input (all features values) of the ith training example,
- $y^{(i)}$ is is the output of the ith training example,
- λ is regularization parameter (or Tikhonov parameter).

Beware of in the second summation term, the index j start at 1 and not 0 (θ_0 is not penalized).

Code your class **MyRidge** as per the instructions below:

```
class MyRidge(ParentClass):
    fit_(self, lambda=1.0, max_iter=1000, tol=0.001):
        """
        Fit the linear model by performing Ridge regression (Tikhonov
regularization).
        Args:
         lambda: has to be a float. max_iter: has to be integer.
         tol: has to be float.
        Returns:
         Nothing.
        Raises:
         This method should not raise any Exception.
        """
```

Regularization parameter must be a positive float.

Try your method with the dataset named **dataset.csv**.
The dataset is made of 3 columns: 2 features and the output (x1, x2 and y).
You have representations of the dataset in the following figure:

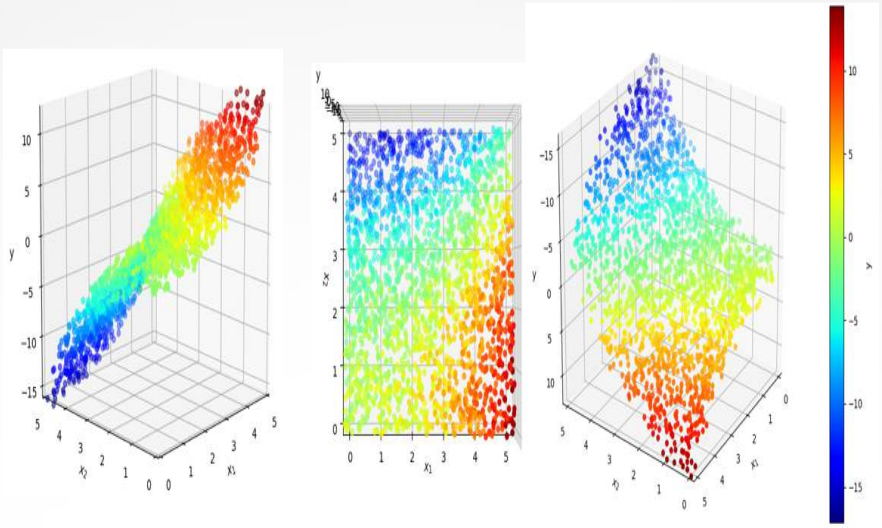

Figure 1: 3D representations of the dataset y(x1,x2).

The best hypothesis for your data has the following form:

$$h(x_1, x_2) = \theta_0 + \theta_1.x_1 + \theta_2.x_1^3 + \theta_3.x_2 + \theta_4.x_2^2$$

Good practice:

In **day01** you probably fit your data with the entire dataset.
This is not a good practice !
You should at list divide the dataset in 2 sets : a training set and a test set.
A better practice is to divide your dataset in 3 sets: training set, cross-validation set and a test set.
Several techniques exist to do a good training/cross-validation process: the *LeaveOneOut*, *KFold* and so on(which you can find in sklearn: **sklearn.model_selection.LeaveOneOut** or **sklearn.model_selection.KFold**).
See documentation on internet about it to get more information.

For the exercise, we will simplify the process:

- You will split your data set into 3 sets: training set, cross-validation set and testing set.
- The training set will be represent 60 % of the dataset and the cross-validation set will correspond to 20 % of the dataset.
- Perform few training with different values of λ (choose the values but stay coherent !).
- Choose the best set of θ coefficients based on the error results (RMSE and R2 score for example) of your cross-validation.
- And verify if your model is a good generalization by calculating the error results on the testing set.

Plot a graph with 3 curves:

- The output with respect to one feature,
- The model you get when you fit the data with the method **.fit()** of your class **MyLinearRegression**,
- The model you get when you fit the data with the method **.fit_()** of your class **MyRidge**,

Answer to the following question (with quantitative arguments):

- Which model is better ?

Questions:

- What is the biais and the variance ?
- A variation of the regularization parameter leads to an evolution of the biais and the variance of a model. How the variance and the biais evolve with the regularization parameter ?

Exercise 09 - Regularized Logistic Regression

Turning directory :	ex09
Files to turn in :	log_reg.py
Forbidden function :	None
Remarks :	n/a

Objectives:

In the last exercice, you implemented of a regularized version of the linear regression algorithm, called Ridge regression. Now it is time to update your own logistic regression as well !

In the scikit-learn library, the logistic regression implementation have a regularization policy by default which can be selected using the parameter **penalty**.

The goal of this exercice is to update your old LogisticRegressionBatchGD class to take that into account.

Instructions:

In the log_reg.py file change the class **LogisticRegressionBatchGD** according to the following :

- add a **penalty** parameter wich can take the following values: {'l2', 'none'}, default = 'l2'.
- update the **.fit()** method:

if **penalty = 'l2'**: use a regularized version of the gradient descent.if **penalty = 'none'**: use the unregularized version of the gradient descent from day02.

Now that you have done this, you will test your model with the same datasets from day02 and compare the results with and without regularization.

Exercise 10 - Z-score standardization

Turnin directory :	ex10
Files to turn in :	z-score.py
Forbidden function :	None
Remarks :	n/a

Objective:
You must implement the following formula as a function:

$$x'_i = \frac{x_i - \frac{1}{m}\sum_{i=1}^{m} x_i}{\sigma = \sqrt{\frac{1}{m-1}\sum_{i=1}^{m}(x_i - \frac{1}{m}\sum_{i=1}^{m} x_i)^2}}$$

where:

- x is a vector of dimension m * 1
- x_i is the ith component of the vector x

Which is way easier to understand in the following form:

$$x'_i = \frac{x_i - \mu}{\sigma}$$

This should remind you something from day00...

Nope?

Ok lets do a quick recap:

- μ is the mean of the components of x
- σ is the standard deviation of the components of x

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

```
def zscore(x):
    """Computes the z-score standardization lambda function of a non-empty
numpy.ndarray.
    Args:
     x: has to be an numpy.ndarray, a vector.
    Returns:
     The z-score standardisation lambda function, which takes a number as
input and returns the normalized value of this number for the vector x.
     None if x is a non-empty numpy.ndarray.
    Raises:
     This function shouldn't raise any Exception.
    """
```

Examples

```
>>> X = numpy.array([0, 15, -9, 7, 12, 3, -21])
>>> z_x = zscore(X)
>>> z_x(15)
1.2068453023747985
>>> z_x(-21)
-1.896471189446112
>>> z_x(0)
-0.08620323588391418
>>>
>>> Y = np.array([2, 14, -13, 5, 12, 4, -19])
>>> z_y = zscore(Y)
>>> z_y(15)
1.2519577145903358
>>> z_y(-21)
-1.9029757261773106
>>> z_y(0)
-0.06259788572951679
```

Exercise 11 - min-max standardization

Turnin directory :	ex11
Files to turn in :	minmax.py
Forbidden function :	None
Remarks :	n/a

Objective:
You must implement the following formula as a function:

$$x'_i = \frac{x_i - min(x)}{max(x) - min(x)}$$

where:

- x is a vector
- x_i is the ith component of the vector x
- $min(x)$ is the minimum value found in the components of the vector x
- $max(x)$ is the maximum value found in the components of the vector x

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

```
def minmax(x):
    """Computes the min-max standardization lambda function of a non-empty
numpy.ndarray, using a for-loop.
    Args:
     x: has to be an numpy.ndarray, a vector.
    Returns:
     The min-max standardisation lambda function, which takes a number as
input and returns the normalized value of this number for the vector x.
     None if x is a non-empty numpy.ndarray.
    Raises:
     This function shouldn't raise any Exception.
    """
```

Examples

```
>>> X = [0, 15, -9, 7, 12, 3, -21]
>>> m = minmax(X)
>>> m(15)
1.0
>>> m(-21)
0.0
>>> m(0)
0.5833333333333334
```

Exercise 12 - PolynomialFeatures

Turnin directory :	ex12
Files to turn in :	polynomial_feat.py
Forbidden function :	None
Remarks :	n/a

Objective:
Original features does not always encapsulate the true relationships between the data and the prediction. As you have seen in some previous exercices on Linear Regression and Ridge, sometimes the relationship is better represented using non-linear parameters, like **polynomial features**, or **interraction terms**.

- 'Polynomial feature' refers to a feature build by putting an original feature to some polynomial degree.
- 'Interraction term' refers to a feature build by multiplying together two features (originals or polynomials)

In such cases it is necessary to **build** your own non-linear features by yourself.

You will have to create a function able to produce all the polynomial features possible for a dataset up to a given polynomial degree and all the interaction terms possible for this dataset up to the given polynomial degree -1. Your function must also add an intercept column of 1 as the first column of the dataset.

Take a look at the class sklearn.preprocessing.PolynomialFeature here: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html?highlight=polynomial#sklearn.preprocessing.PolynomialFeatures.
We are doing a light mimic of it as a simple function.

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

```
def polynomialFeatures(x, degree=2, interaction_only= False,
include_bias=True):
    """Computes the polynomial features (including interraction terms) of a
non-empty numpy.ndarray.
    Args:
     x: has to be an numpy.ndarray, a vector.
    Returns:
     The polynomial features matrix, a numpy.ndarray.
     None if x is a non-empty numpy.ndarray.
    Raises:
     This function shouldn't raise any Exception.
    """
```

Examples

```
>>> X = np.arange(6).reshape(3, 2)
>>> X
array([[0, 1],
     [2, 3],
     [4, 5]])
>>> polynomialFeatures(X)
array([[ 1., 0., 1., 0., 0., 1.],
     [ 1., 2., 3., 4., 6., 9.],
     [ 1., 4., 5., 16., 20., 25.]])
>>>
>>> polynomialFeatures(X, 3)
array([[ 1., 0., 1., 0., 0., 1., 0., 0., 0., 1.],
     [ 1., 2., 3., 4., 6., 9., 8., 12., 18., 27.],
     [ 1., 4., 5., 16., 20., 25., 64., 80., 100., 125.]])
>>>
>>> polynomialFeatures(X, 3, interaction_only=True, include_bias=False)
array([[ 0., 1., 0.],
     [ 2., 3., 6.],
     [ 4., 5., 20.]])
>>> m = minmax(X)
```