Iterative search and lineal regression

Gradient descent iterative method and lineal regression

Blanca Cano Camarero
Department: DECSAI
University: ETSIIT, Granada university
Country: Spain
Date: April 3, 2021
Mathematics and computer engineering degrees,
Doble Grado matemática e informática

Contents

Chapter 1

Gradient descent

1.1 Gradient descent 's algorithm

1.1.1 Introduction

Gradient descent is a general technique for minimizing twice-differentiable functions through its slope. [1] It is used to find local minimums. The start point is crucial in the search.

The basic idea is to update the weights using the gradients until it is not possible to continuous minimizing the error.

1.1.2 Math

In order to understand the algorithm we are going to define:

Let $w(0) \in \mathbb{R}^d$ be an arbitrary initial point, $E : \mathbb{R}^d \times \mathbb{R}^d \longrightarrow \mathbb{R}$ a class C^1 function. The learning rate or step size $\eta \in \mathbb{R}^+$ is a experimental coefficient about how much are we going to follow the slope to obtain the new weight. Let $w(t) \in \mathbb{R}^d \quad t \in \mathbb{N}$ be the weight for t iteration which is defined as

$$w(t+1) = w(t) - \eta \nabla E_{in}(w(t))$$

Properties

- This algorithm gives local minimums.
- Convergence it is not assured, so it would be necessary some stop criteria.
- For a convex function it would be a unique global minimum.
- The learning rate η variable in time is important: fixes learning gradient descent algorithm.

1.1.3 Algorithm

The following code snippet implement the algorithm, where $w(0)$ is the *initial_point*, E is the error, $\nabla E_{in}(w)$ is *gradient_function* and finally η is *eta*. The value $\eta = 0.1$ is a heuristic basic on purely practical observation [1].

In order to avoid an infinite search, our stop criteria are a limit in the number of iterations *max_sitter* and an error tolerance.

```
def gradient_descent(initial_point, loss_function,
                gradient_function,  eta, max_iter, target_error):
    '''
    initicial point: w_0
    E: error function
    gradient_function
    eta:  step size

    ### stop conditions ###
    max_iter
    target_error

    #### return ####
    (w,iterations)
    w: the coordenates that minimize E
    it: the numbers of iterations needed to obtain w

    '''

    iterations = 0
    error = E( initial_point[0], initial_point[1])
    w = initial_point

    while ( (iterations < max_iter) and(error > target_error)):

        w = w - eta * gradient_function(w[0], w[1])

        iterations += 1
        error = loss_function(w[0], w[1])


    return w, iterations
```

1.1.4 Problem 1

We want to solve the following problem:

Use gradient descent's algorithm to find a minimum for the function

$$E(u,v) = (u^3 e^{(v-2)} - 2 * v^2 e^{-u})^2.$$

Set $(u,v) = (1,1)$ as initial point and use learning rate $\eta = 0.1$.

Compute analytically the gradient of $E(u,v)$

$$\nabla E(u,v) = \left(\frac{\partial}{\partial u}(u^3 e^{(v-2)} - 2 * v^2 e^{-u})^2, \frac{\partial}{\partial v}(u^3 e^{(v-2)} - 2v^2 e^{-u})^2 \right) =$$
$$= \Big(2(u^3 e^{(v-2)} - 2 * v^2 e^{-u})(3u^2 e^{(v-2)} + 2v^2 e^{-u}), 2(u^3 e^{(v-2)} - 2 * v^2 e^{-u})(u^3 e^{(v-2)} - 4ve^{-u}) \Big)$$

Number of iterations and final coordinates.

Firstable we need to use 64-bits float, so we are going to use the data type $float64$ of numpy library [2].

The functions' declaration are:

```
  def dEu(u,v):
    '''
    Partial derivate of E with respect to the variable u
    '''
    return np.float64(
        2
        *( 3* u**2 * np.e**(v-2) + 2*v**2 * np.e**(-u) )
        *( u**3 * np.e**(v-2) - 2*v**2 * np.e**(-u))
    )

def dEv(u,v):
    '''
    Partial derivate of E with respect to the variable v
    '''
    return np.float64(
        2*
        ( u**3 * np.e**(v-2) - 2*v**2 * np.e**(-u) )
        *( u**3 * np.e**(v-2) - 4*v * np.e**(-u))
    )


def gradE(u,v):
    '''
        gradient of E
    '''
    return np.array([dEu(u,v), dEv(u,v)])
```

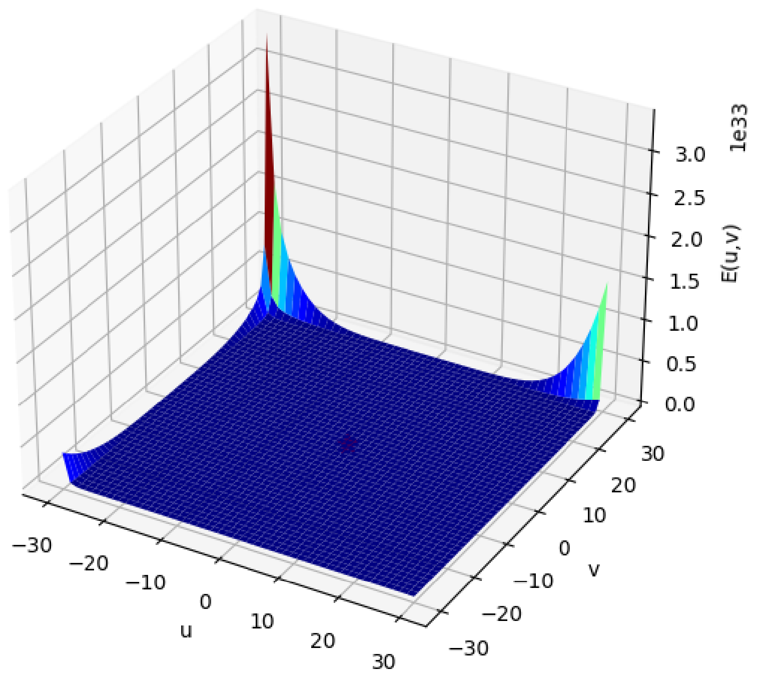
To obtain the number of iterations and the final coordinates, the only thing we need to do is to call *gradien_descent* function with the initial conditions:

```
eta = 0.01
max_iter = 10000000000
target_error = 1e-14
initial_point = np.array([1.0,1.0])
w, it = gradient_descent( initial_point,
                          E,
                          gradE,
                          eta,
                          max_iter,
                          target_error )
```

The result are:

- Numbers of iterations: 178.
- Final coordinates: $(1.162, 0.924)$.

A 3d graph with the result is



1.1.5 Problem 2

For function $f(x, y) = (x + 2)^2 + 2(y - 2)^2 + 2\sin(2\pi x)\sin(2\pi y)$

Use gradient descent to minimize f

The initial point is $(x_0 = -1, y_0 = 1)$, learning rate is $\eta = 0.01$ and the maximum number of iterations must be 50. Plot the result and repeat the experiment with $\eta = 0.1$.

Firstly we are going to calculate partial derivatives and gradient of f.

$$\frac{\partial}{\partial x} f = 2(x+2) + 2\sin(2\pi y)\cos(2\pi x)2\pi = 2(x+2) + 4\pi\sin(2\pi y)\cos(2\pi x)$$

$$\frac{\partial}{\partial y} f = 2(y-2) + 4\pi\sin(2\pi x)\cos(2\pi y)$$

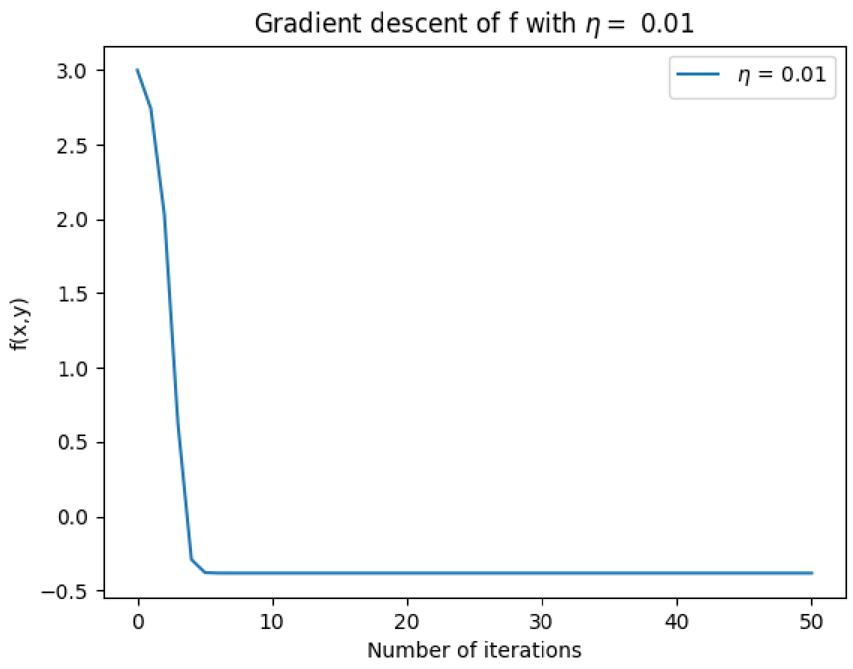
It is important to realise that $f(x, y) < 0$ for some values in $\mathbb{R}^3$ so the error target has been omitted in this algorithm.

Now the new algorithm is

```python
def gradient_descent_trace(initial_point, loss_function,
  gradient_function,  eta, max_iter):
  '''
  initicial point: w_0
  loss_function: error function
  gradient_function
  eta:  step size

  ### stop conditions ###
  max_iter

  #### return ####
  (w,iterations)
  w: the coordenates that minimize loss_function
  it: the numbers of iterations needed to obtain w

  '''

  iterations = 0
  error = loss_function( initial_point[0], initial_point[1])
  w = [initial_point]

  while iterations < max_iter:

      new_w = w[-1] - eta * gradient_function(w[-1][0], w[-1][1])


      iterations += 1
```

```
        error = loss_function(new_w[0], new_w[1])
        w.append( new_w )

    return w, iterations
```

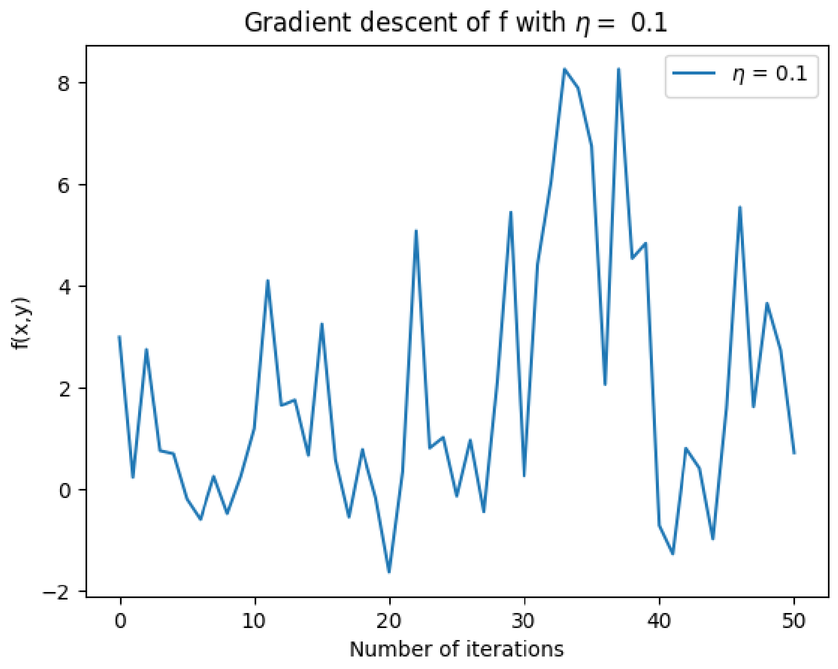
After 50 iterations for $\eta = 0.01$, the final coordinates are $(-1.269, 1.287)$ and their value is -0.381. The graph, which shows the relation between iterations and the function minimization is



As far as we have seen, before the 10^{th} iteration we are really close to the minimum and stay there without fluctuate.

On the other hand, after 50 iterations for $\eta = 0.1$ the final coordinates are $(-2.939, 1.608)$ and their value is 0.724, so as we can see this result is worse than the last one.

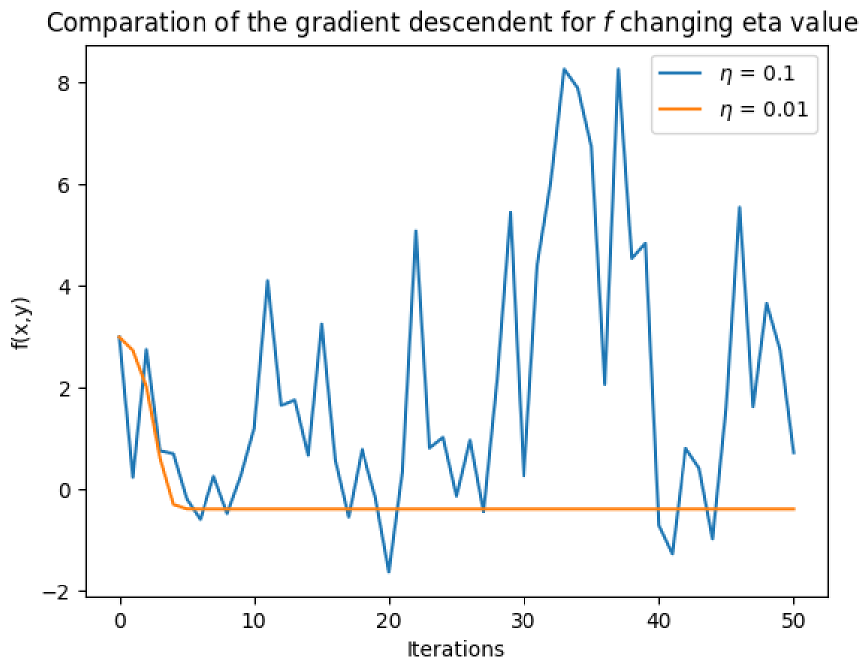
In the following graph we can see the evolution fluctuation of the images iteration by iteration



.

The reason for this irregularity is that the step size is too big, so it skips the minimum.

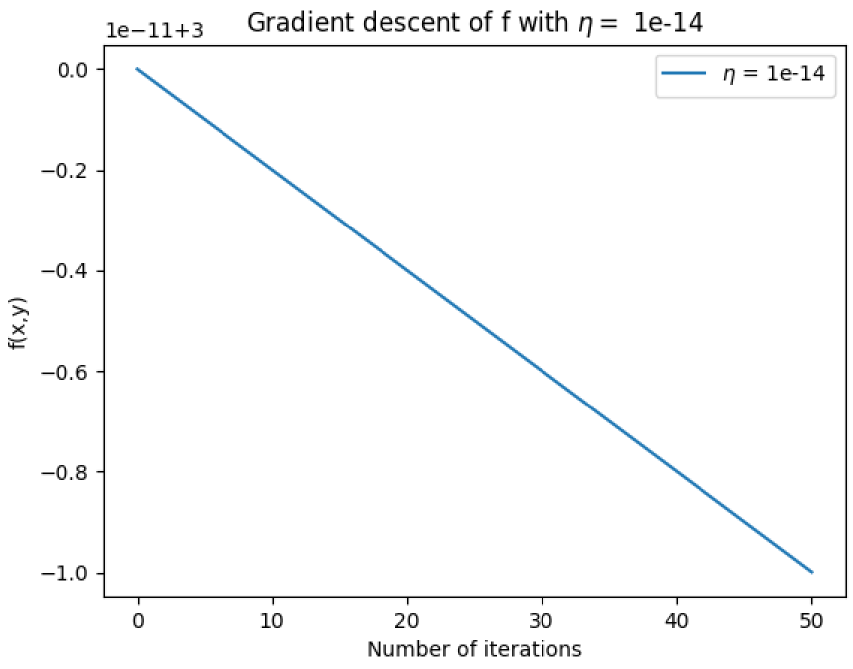
We can also compare the two experiment in the following graph.

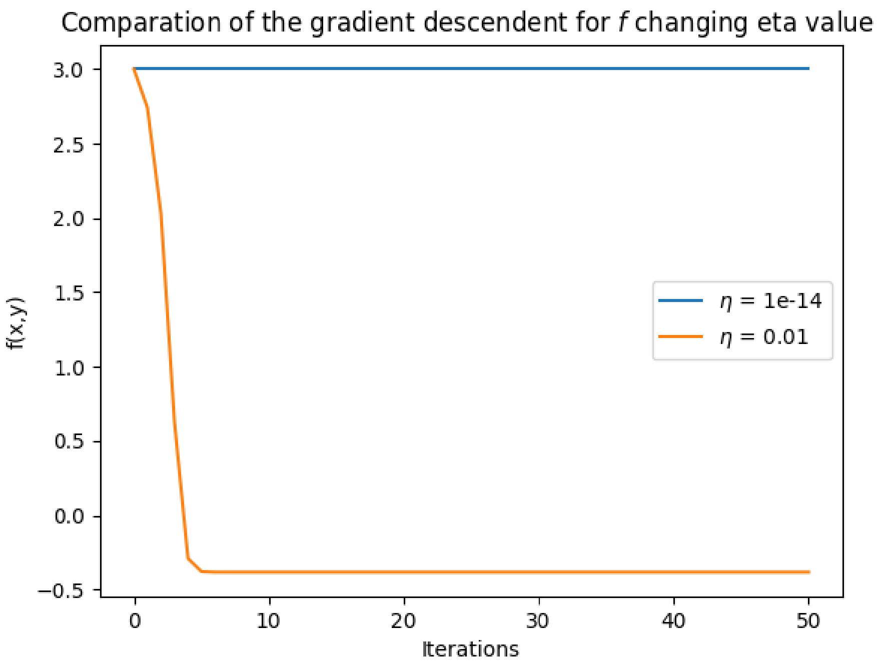


.

Moreover, basic on its mathematics proof, which use Taylor's series, we know that it should be small, but if it is too small the algorithm will never reach the minimum in a right time.

Let see a new example, now $\eta = 10^{-14}$, after 50 iteration the final coordinates are $(-1, 1)$ and the value is 3, so this new selection is even worse that the one with the bigger step size, although it goes without oscillate.





As a conclusion, a priory, it is difficult to select a step size value, each problem should have an appropriate one, and the selection must be empirical. Even though some heuristic [1] tell that $\eta = 0.01$ its a good try.

Minimum value

Before running the algorithm is important to think about a good value for the learning rate η. Based on the last section $\eta = 0.1$ is a good one.

After run the program the results are

Initial point	Final coordinates	Final value
(-0.5 -0.5)	(-0.793 -0.126)	9.125
(1 1)	(0.677 1.29)	6.437
(2.1 -2.1)	(0.149 -0.096)	12.491
(-3 3)	(-2.7315 2.713)	-0.381
(-2 2)	(-2. 2.)	0

This example gives the idea that the local minimums found depend on the start point and a priory, unless we know some properties of the function such as convexity or monotony we are not able to assure that the minimum found is global.

For a mathematical study we need differentiable functions, and a technique to find where their zeros are, so we need to solve their equations. Some time this is not possible and we use numeric methods.

1.1.6 Final conclusion about finding global functions' minimum

- Properties differentiable, arbitrary initial point, eta, computational cost...
TO-DO

Chapter 2

Linear Regression

2.1 Linear regresion

2.1.1 Stochastic gradient descendent

Stochastic gradient descendent (SGD) is a sequential version of the gradient descent. Instead of considering the full batch gradient on all N training data points, we condider a stochastic version of the gradient. Firts, pick a training data point (x_n, y_n) uniformly random (hence the name 'stochastic') and consider only the error on that data point. [1]

The gradient of this single data point's error is used for the weight update in exactly the same that the gradient was used in batch gradient descent.

Another variants are the **mini-batch gradient descent** and the **batch gradient descendent**, the differences among them are the size of the batch: one for the pure stochastic gradient descendent, between 32 or 64 for the mini-batch variation and more than that for the batch gradient descendent.

2.1.2 Pseudo - inverse algorithm

Pseudo inverse algorithm also known as **linear regression algorithm** or **ordinary least squares**(OLS) is based on minimizing the squared error between the proyection matrix $h(x) = w^T x$ and y, the target vector, where $x \in \mathbb{R}^{N\times(d+1)}$ is the feature matrix and $N \in \mathbb{N}$ the training data size.

$$E_{in}(w) = \frac{1}{N}\sum_{n=1} N(w^T x_n - y)^2 = \frac{1}{N}\|Xw - y\|^2$$

Where $\|.\|$ is the Euclidean norm of a vector.

Since $E_{in}(w)$ is differenciable we can use standard matrix calculus to find the w that minimizes E_{in} with respect to w is the zero vector:

$$\nabla E_{in}(w) = \frac{2}{N}(X^T X w - X^T y) = 0$$

Finally to get $\nabla E_{in}(w)$ to be 0, one should solve for w that satisfies

$$X^T X w = X^y$$

If $X^T X$ is invertible, $w = X^\dagger y$ where $X^\dagger = (X^T X)^{-1}$ is the `pseudo-inverse` of X. The resulting w is the unique optimal solution that minimizes E_{in}. Otherwise a pseudo-inverse can still be defined, but the solution will not be unique.

In practice, $X^T X$ is invertible in most of the cases since N is often much bigger than $d+1$, so there wil likely be $d+1$ linerly indepent vector x_n.

2.1.3 Exercise 1

Estimate a linear regression model from the data provided by the feature vectors (Average intensity, Symmetry) using both the pseudo-inverse algorithm and the Stochastic Gradient Descent (SGD). The labels will be $\{-1, 1\}$, one for each feature vector of each number. Draw the solutions obtained together with the data used in the fitting. Assess the goodness of the result using Ein and Eout (for Eout calculate the predictions using the data from the test file).

Error

As we have said the error is the mean squared error:

$$E_{out}(h) = \mathbb{E}[(h(x) - y)^2]$$

$$E_{in}(w) = \frac{1}{N}\|Xw - y\|^2$$

A direct implementation is

```python
def Error(x,y,w):
    '''quadratic error
    INPUT
    x: input data matrix
    y: target vector
    w:  vector to

    OUTPUT
    quadratic error >= 0
    '''
    error_times_n = np.linalg.norm(x.dot(w) - y.reshape(-1,1))**2

    return error_times_n/len(x)
```

For the euclidian norm we have used `np.linalg.norm` [3] numpy function. The gradient computation is direct too:

$$\nabla E_{in}(w) = \frac{2}{N}(X^T X w - X^T y) = \frac{2}{N}(X^T(Xw - y))$$

```
def dError(x,y,w):
''' gradient
OUTPUT
column vector
  '''

  return (2/len(x)*(x.T.dot(x.dot(w) - y.reshape(-1,1))))
```

Interpretation of the mean squared error, E

The mean squared error funcion $E : \mathbb{R}^d \longrightarrow \mathbb{R}_0^+$ measures the average of the squared difference between the estimated values ant the actual value[4]. Hence, the nearer to zero, the better.

Pseudo-inverse algorithm

As we have described in pseudo inverse introduction, firstly we need to compute the pseudo-inverse. For that we have use `np.linalg.pinv` [5]function from numpy library.

```
def pseudoInverseMatrix ( X ):
  '''
  INPUT
  X: is a matrix (must be a np.array) to use transpose and dot method
  OUTPUT
  hat matrix
  '''

  '''
  #S =( X^TX ) ^{-1}
  simetric_inverse = np.linalg.inv( X.T.dot(X) )

  # S X^T = ( X^TX ) ^{-1} X^T
  return simetric_inverse.dot(X.T)
  '''
  return np.linalg.pinv(X)
```

Finally we have to compute $w = X^{\dagger} y$

```
def pseudoInverse(X, Y):
    '''
    INPUT
    X is the feature matrix
    Y is the target vector (y_1, ..., y_m)

    OUTPUT:
    w: weight vector
    '''
    X_pseudo_inverse = pseudoInverseMatrix ( X )
    Y_transposed = Y.reshape(-1, 1)

    w = X_pseudo_inverse.dot( Y_transposed)

    return w
```

Pseudo-inverse linear regression model

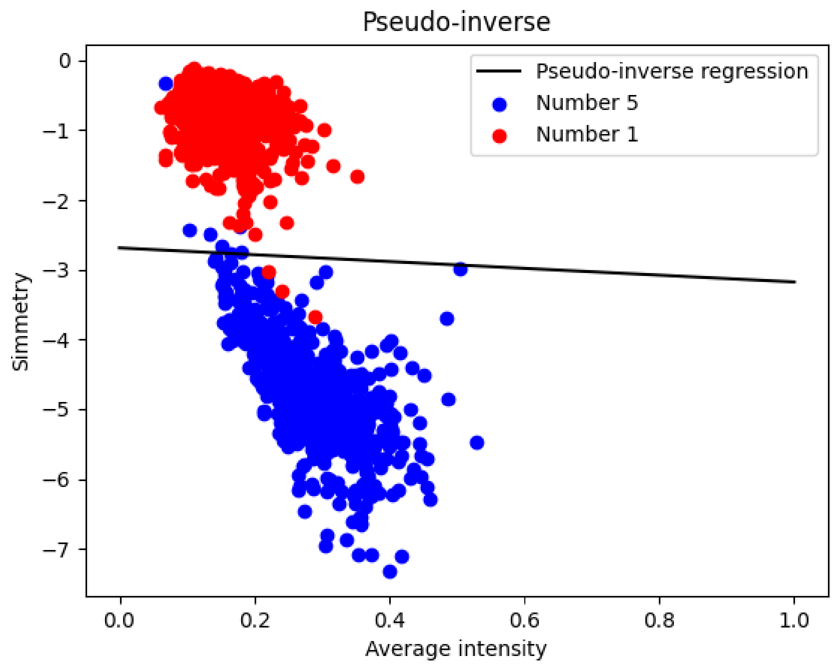
After execute the algorithm we obtain:

```
___ Goodness of the Pseudo-inverse fit ___

  Ein:   0.07918658628900395
  Eout:  0.1309538372005258

Evaluating output training data set
Input size:  1561
Bad negatives : 7
Bad positives : 3
Accuracy rate : 99.35938500960923 %

Evaluating output test data set
Input size:  424
Bad negatives : 1
Bad positives : 7
Accuracy rate : 98.11320754716981 %
```

Which means that the w calcs by our pseudo-inverse algorithm we the training data set has a $E_{in}(w) = 0.079$ and with the test data $E_{out}(w) = 0.131$, for our experiment it is a good fit, because it is close enough to zero. In addition, if we evaluate the data classification, from 1561 traing data it only misclassify 10, whereof 7 was truely positives. The accuracy rate ($\frac{\text{good classified data}}{\text{data set size}}$) is 99.358%, which continues being really good.

Initially, we could think that it classifies the positives values better, but if we analyse the output from test data set, there are more negatives values misclassified, so we cannot stablish any relation. Moreover here the accuracy rate is 98.113%

Finally, a graphic representation for the solutions is



How have we plotted the regression line.

Firstable, to draw a line we need two points.

Therefore we use the signum to classify the numbers, we are going to find two points in $\mathbb{R}^2$ that their stimation is zero, which means that they are in the middle of classification (the regression line).

The obtainied weight vector $w^T = (w_1, w_2, w_3)^T$, means that for a point $(x, y) \in \mathbb{R}^2$ its stimation is $h(x, y) = (1, x, y)w = w_1 + w_2 x + w_3 y$.

To calcule the two points we are going to equal $h(x, y) = 0$ and from the infinities solution we can calculate:

if $x = 0$ then $y = \frac{-w_1}{w_3}$, and if $x = 1$ then $y = \frac{-w_1 - w_2}{w_3}$.

The related code is

```
# regression line
# x= 0
symmetry_for_cero_intensity = -w[0]/w[2]

#  x = 1, 0 = w0 + w1 * w2 * x2
```

```
    # then y = (-w0 - w1) /w2
    symmetry_for_one_intensity= (-w[0] - w[1])/w[2]

    # plotting order
    plt.plot([0, 1],
             [symmetry_for_cero_intensity,symmetry_for_one_intensity],
             'k-',
             label=(title+ ' regression'))
```

Initial point

training data

Motivo de 1, -1 para compensar la etiqueta

HAcer 200

W no importa en la práctica (podemos manipularlo)

VISUALIZAR MODELO CON SCATTER PLOT (DIAPOSITVA 22, CON LOS DATOS ENTRENAMIENTO Y TEST)t

E w deberá de ser la misma

Incluir valor de E_inE_oit y eroro de clasificación (lo normal es que el error en test sea un poquito peor que los de entrenamiento.

Gradiente descendiente minimizar eterativamente:

En el segunto no tiene porqué darse que una iteración sea mejor (otra cosa es que mejore la tendencia mejor)

Si se utilizan todos sí que debería de ir mejorando de manera global.

2.1.4 How to plot

As we know $w = (cte, intensity - coeffinient, symmetry - coefficient) = (c, i, s)$ have three parameters. And we are going to plot in a 2D grap where

$c + xi + ys = 0$, it is a line, so we are going to compute two points, the one that x

$(x = 0) : y = frac{-c}y$ $(x = 1) : y = frac{-c - i}y$

2.2 Experiment

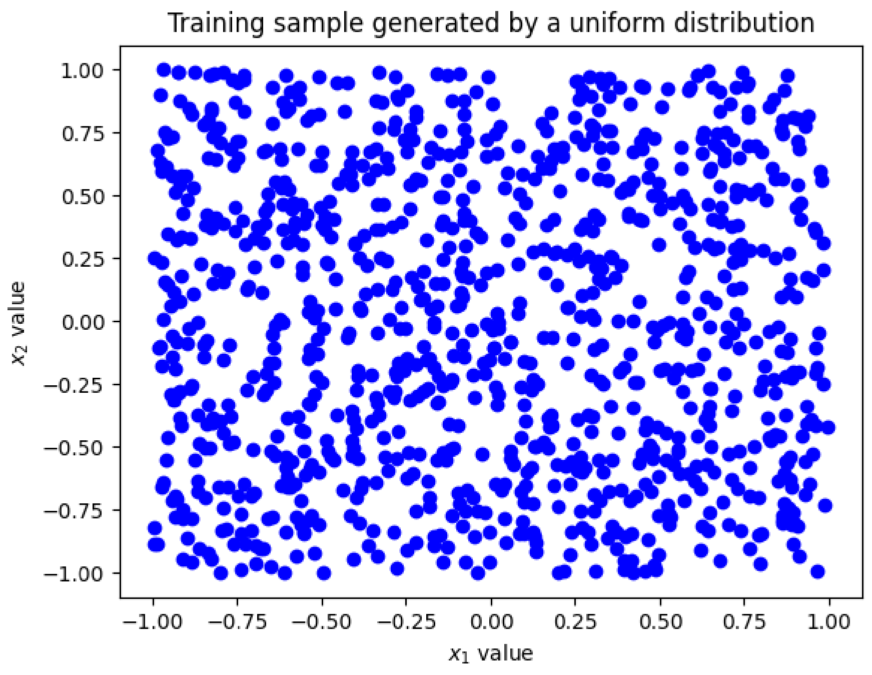
2.2.1 a) Generate a training sample

We are going to use a uniform generation:

```
def simula_unif(N, d, size):
        ''' generate a trining sample of N  points
in the square [-size,size]x[-size,size]
'''
        return np.random.uniform(-size,size,(N,d))
```

After fixed radom seed to 1.

The final 2D map is



2.2.2 b) Labels, noise and map

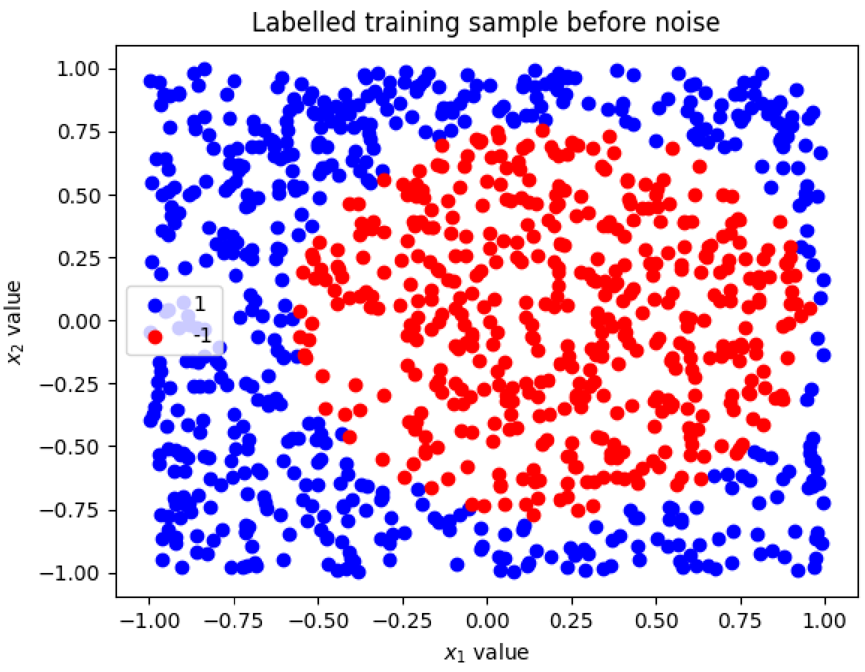
b) Let's consider the function $f(x_1, x_2) = sign((x_1 - 0.2)^2 + x_2^2 - 0.6)$ that we will use to assign a label to each point of the previous sample. We introduce noise on the labels, randomly changing the sign of 10 % of them. Draw the obtained labels map.

Before plotting, while analysing the function is important to have in mind that a ciercumference with radius r and center $(c_1, c_2) \in \mathbb{R}^2$ are the points $(x_1, x_2) \in \mathbb{R}^2$ that verify

$$(x_1 - c_1)^2 + (x_2 - c_2)^2 = r^2$$

Therefore looking at f it is easy to think that we are goint to see a circle of radius $\sqrt{0.6}$ and center $(0.2, 0)$.

The plotting is

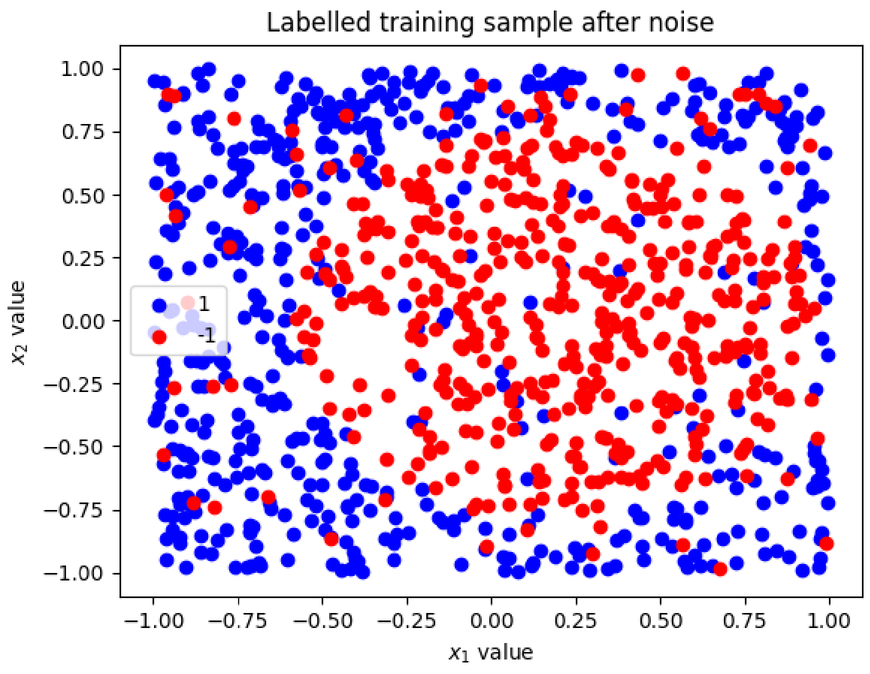


In order to introduce noise on the label we are going to change randomly the sign of the 10% of the labels obtained by b.

```
#labels
y = np.array( [f(x[0],x[1]) for x in training_sample ])

index = list(range(size_training_example))
np.random.shuffle(index)

percent_noisy_data = 10.0
size_noisy_data = int((size_training_example *percent_noisy_data)/ 100 )


noisy_y = np.copy(y)
for i in index[:size_noisy_data]:
    noisy_y[i] *= -1
```

As we can see the idea behind the snippet is simple: The noised labels would be a copy of the original one and 10% of the data would change their sign.

The final map is :



2.2.3 Estimate the fitting error of E_{in} using SDG

c) Using $(1, x_1, x_2)$ as feature vector, fit a linear regression model to the generated dataset and estimate the weights w. Estimate the fitting error of Ein using Stochastic Gradient Descent (SGD).

Having in mind the observation in the last subsention that the labels follows a circumference equation with a bit of noise, a linear regression model it is not going to be the best approach.

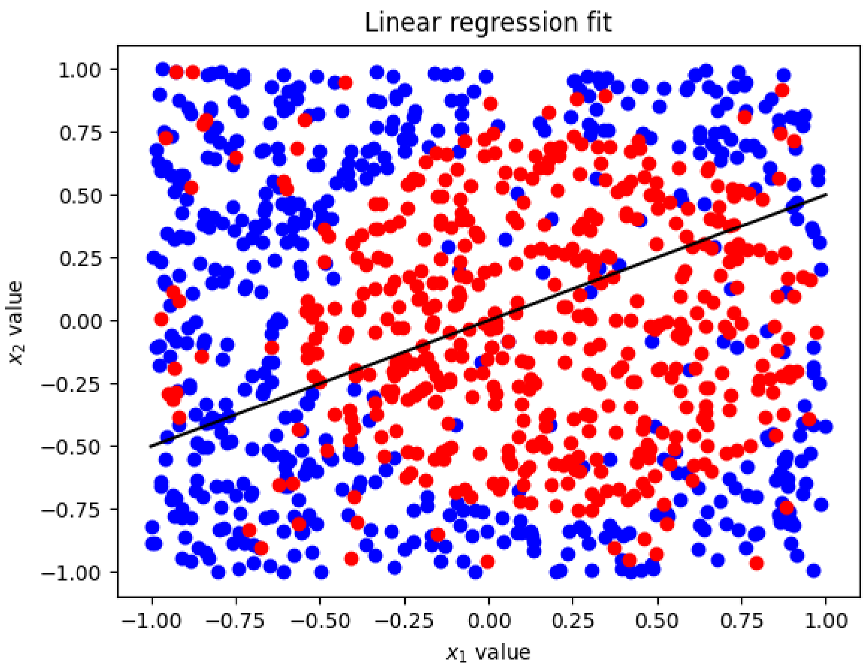
The experiment result are: That for a linear SGD with batch size 32 the E_{in} is 0.930

And valuating output training data set Input size: 1000 Bad negatives : 187 Bad positives : 224 Accuracy rate : 58.9 %

Remember that the bad negatives are the points that the regression classify in negative but they are positives and the bad positives are the negatives one that are classify as a positives.

Finally the accuracy rate was the corrected classify data divided by the input size, so it is not a good model due to the fact that the accuracy rate is so close to a random one, that it would theoretically have a 50% of a accuracy rate.

A visual representation of de fix is



execution:

```
>>> _______LINEAR REGRESSION EXERCISE _______

Exercise 1


 Enter to start


___ Goodness of the Stochastic Gradient Descendt (SGD) fit ___


SGD, batch size 2
Ein:  0.0822091904727966
Eout:  0.13268719301475448

Evaluating output training data set
Input size:  1561
Bad negatives : 8
Bad positives : 3
Accuracy rate : 99.29532351057014 %

Evaluating output test data set
```

```
Input size:  424
Bad negatives : 1
Bad positives : 7
Accuracy rate : 98.11320754716981 %

SGD, batch size 32
Ein:  0.08145689246982911
Eout:  0.13524833395770525

Evaluating output training data set
Input size:  1561
Bad negatives : 6
Bad positives : 3
Accuracy rate : 99.4234465086483 %

Evaluating output test data set
Input size:  424
Bad negatives : 1
Bad positives : 7
Accuracy rate : 98.11320754716981 %

SGD, batch size 200
Ein:  0.0815451181277193
Eout:  0.13602151565370962

Evaluating output training data set
Input size:  1561
Bad negatives : 6
Bad positives : 3
Accuracy rate : 99.4234465086483 %

Evaluating output test data set
Input size:  424
Bad negatives : 1
Bad positives : 7
Accuracy rate : 98.11320754716981 %

SGD, batch size 15000
Ein:  0.08144533934205953
Eout:  0.13486901151111522

Evaluating output training data set
Input size:  1561
Bad negatives : 7
Bad positives : 3
Accuracy rate : 99.35938500960923 %

Evaluating output test data set
Input size:  424
Bad negatives : 1
Bad positives : 7
Accuracy rate : 98.11320754716981 %

___ Goodness of the Pseudo-inverse fit ___

  Ein:   0.07918658628900395
  Eout:  0.1309538372005258

Evaluating output training data set
Input size:  1561
Bad negatives : 7
Bad positives : 3
Accuracy rate : 99.35938500960923 %

Evaluating output test data set
Input size:  424
Bad negatives : 1
Bad positives : 7
Accuracy rate : 98.11320754716981 %

--- Type any key to continue ---
º
Exercise 2


EXPERIMENT (a)


EXPERIMENT (b)


EXPERIMENT (c)


SGD, batch size 32
Ein:  0.9301633062413852

Evaluating output training data set
```

```
Input size:  1000
Bad negatives : 187
Bad positives : 224
Accuracy rate : 58.9 %

 EXPERIMENT (d), lineal regression

The mean value of E_in in all 1000 experiments is: 0.0009131004444134428
The mean value of E_out in all 1000 experiments is: 0.000905505366572186

EXPERIMENT (e)


For one experiment:

 SGD, batch size 32
Ein:  0.6158558061444251

Evaluating output training data set
Input size:  1000
Bad negatives : 88
Bad positives : 50
Accuracy rate : 86.2 %

The mean value of E_in in all 1000 experiments is: 0.0003827441225550457
The mean value of E_out in all 1000 experiments is: 0.00037845910961602556
>>>
```

Bibliography

[1] Hsuan-Tien Lin Yaser S. Abu-Mostafa, Malik Magdon-Ismail. *Learing From Data. A Short Course.* AMLbook, 2012.

[2] Numpy documentation. Numpy basic data types documentation, 2021.

[3] Numpy documentation. norm, documentation, 2021.

[4] wikipedia. Mean squared error wikipedia, 2021.

[5] Numpy documentation. Numpy pseudo-inverse matrix, documentation, 2021.