Iterative search and lineal regression

Gradient descent iterative method and lineal regession

Blanca Cano Camarero
Department: DECSAI
University: ETSIIT, Granada university
Country: Spain

Date: March 20, 2021
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

We are going to define algorithm:

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^2 function. The 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.
- 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_iter* and an error tolerance.

```
def gradient_descent(initial_point, E, 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
    '''

    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 = E(w[0], w[1])


    return w, iterations
```

1.1.4 Problem 1

We want to solve the following problem:

Run gradient descent's algorithm to find a minimum for function $E(u, v) = (u^3 e^{(v-s)} - 2 * v^2 e^{-u})^2$. Start with $(u, v) = (1, 1)$ and $\eta = 1.0$

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

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

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: $(1.162, 0.924)$.

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.

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

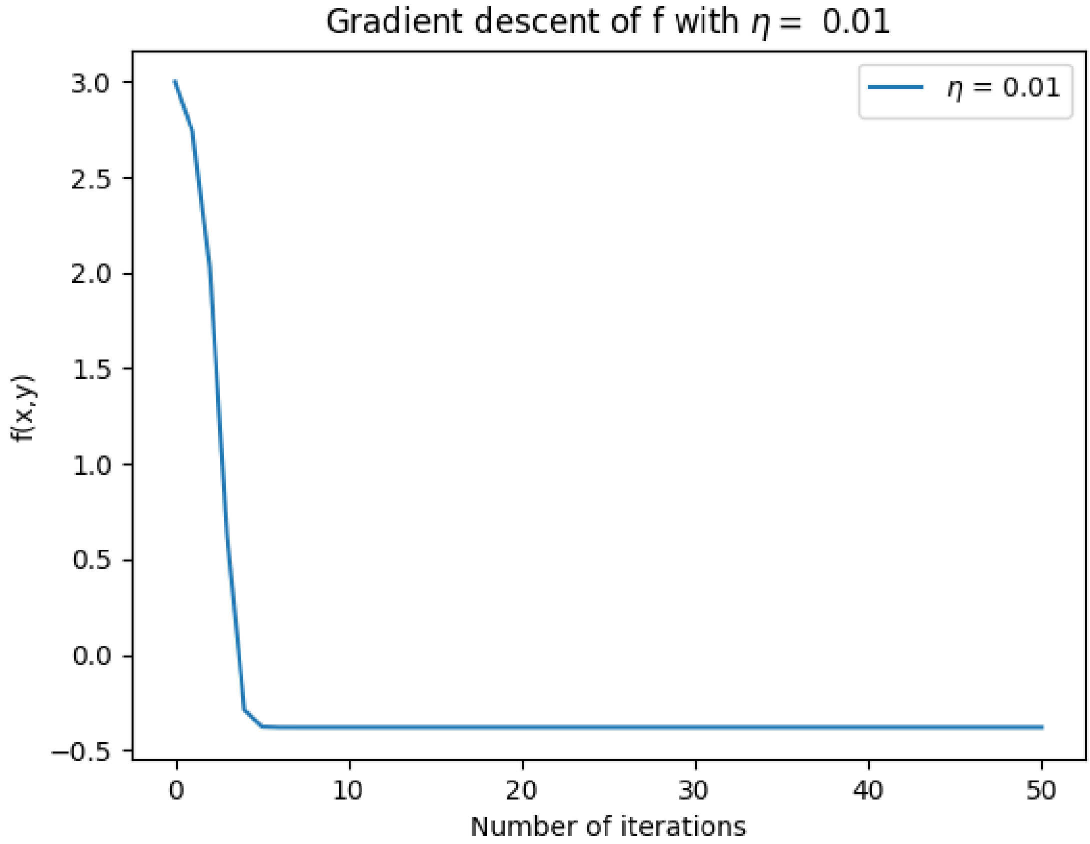
$$\frac{\partial}{\partial y} f = 2(y-2) + 4\pi\sin(2\pi x)\cos(2\pi y)$$

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

```
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
```

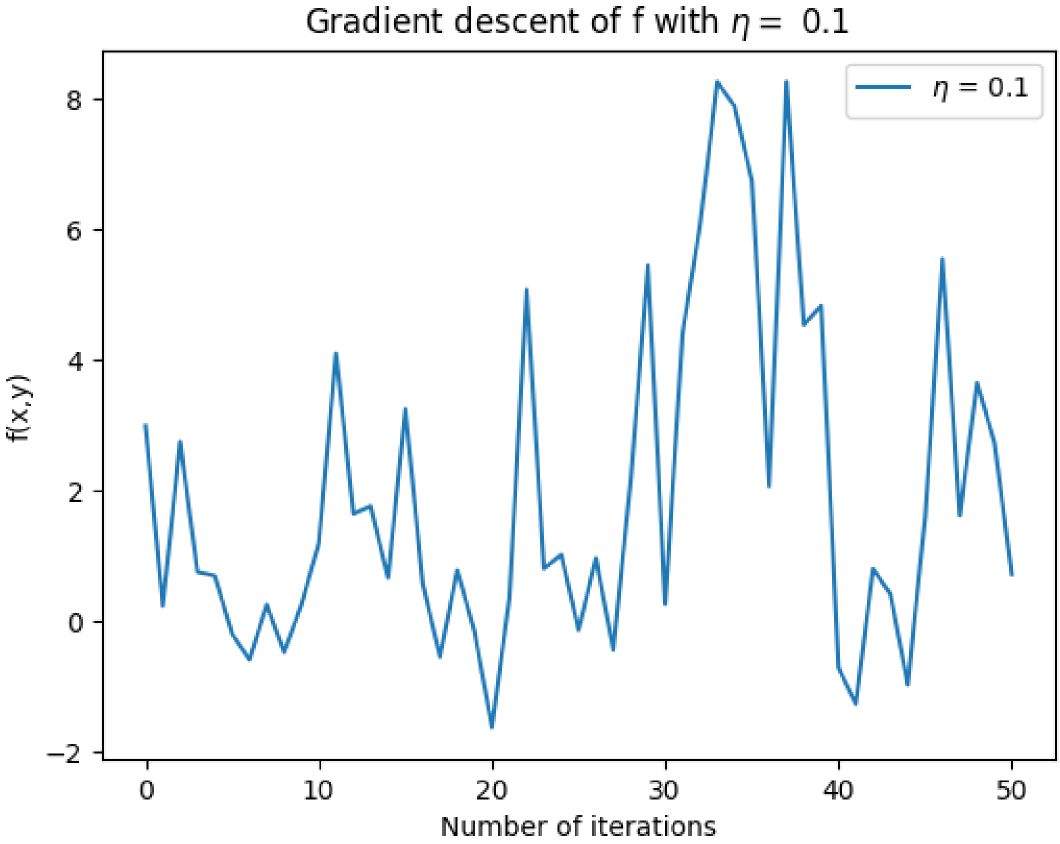
After 50 iterations for $\eta = 0.01$, the final coordinates are $(-1.269, 1.287)$ and their value is -0.381. The graph, which show the relation between iteration and the function minimization is



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.

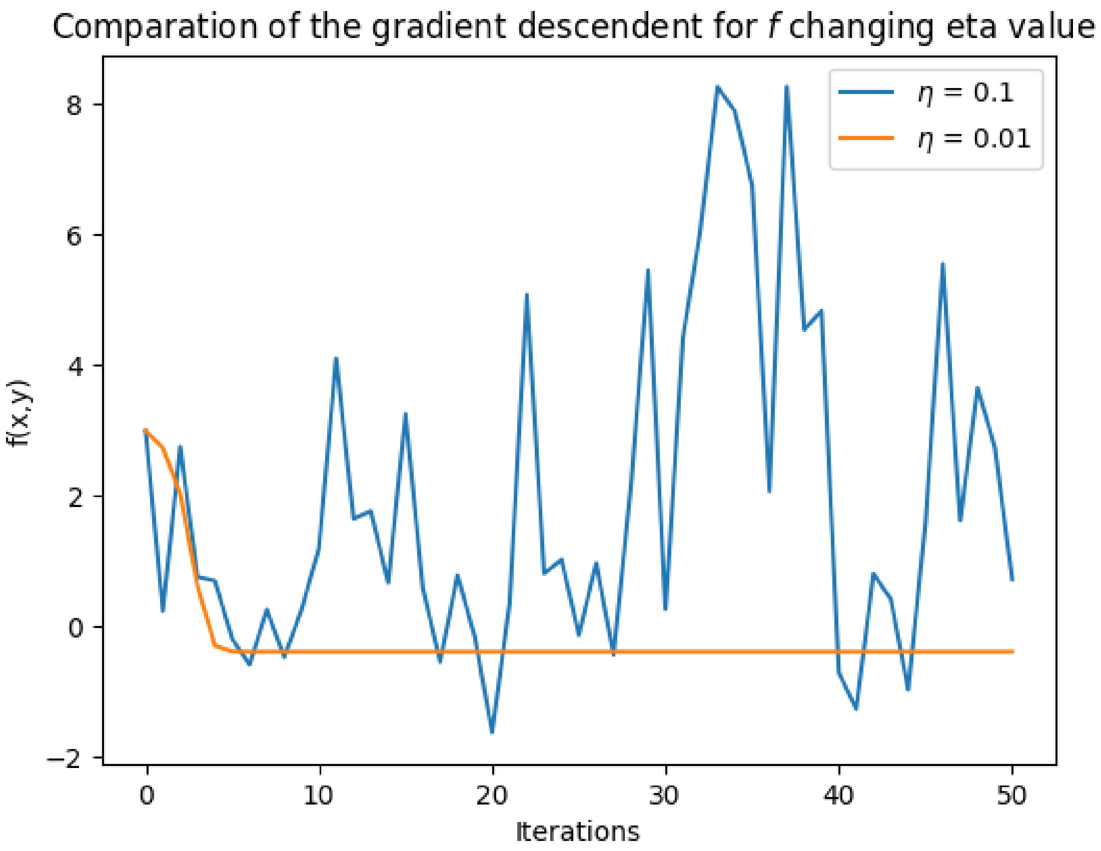
In the following graph we can see the evolution fluctuation of the images iteration by iteration



.

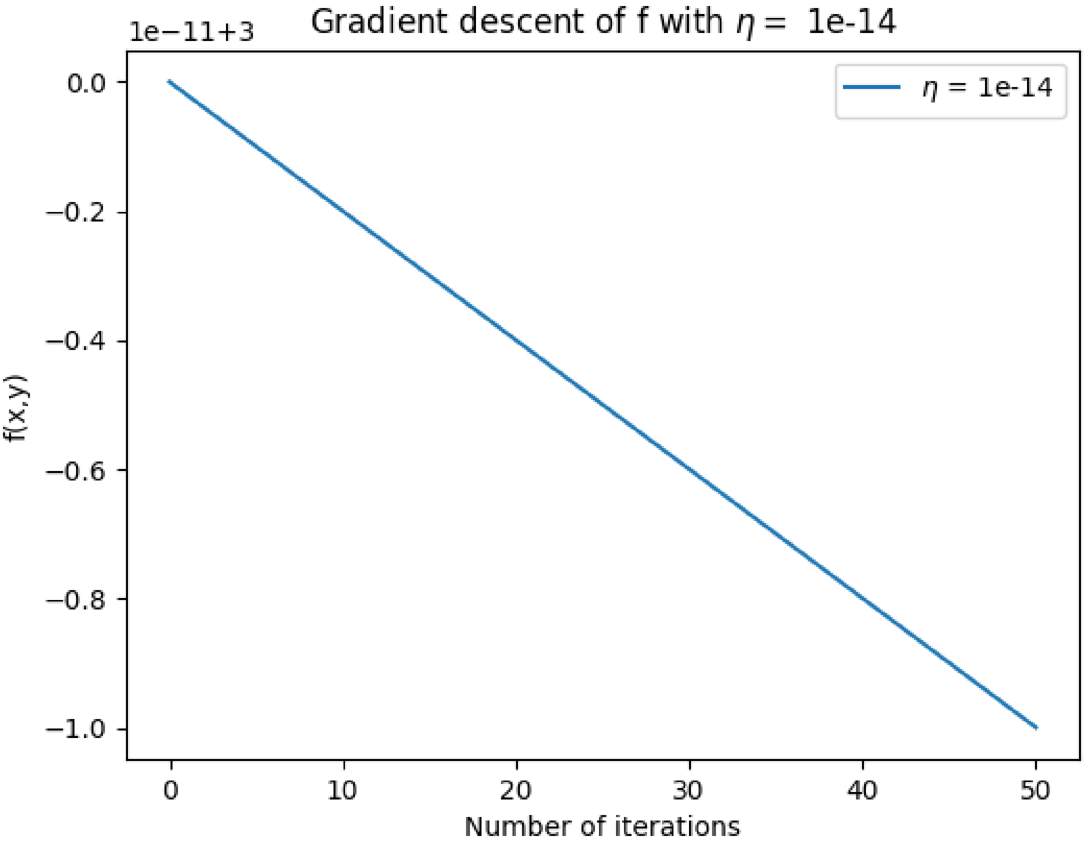
The reason for this irregularity is that the step size is too bigger so it skip the minimum.

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 one with the bigger step size, although it go 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 η, based on the last section $\eta = 0.1$ is a good one.

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 documentatation, 2021.