

APRENDIZAJE AUTOMÁTICO



UNIVERSIDAD
DE GRANADA

Práctica 1: Programación

Alba Casillas Rodríguez
76738108B
albacaro@correo.ugr.es

Ejercicio 1: Búsqueda iterativa de óptimos

El **descenso de gradiente** es una técnica general de optimización iterativa que alcanza un óptimo local siguiendo la dirección del vector gradiente en cada punto.

Implementamos el algoritmo de gradiente descendente, donde establecemos como "**w**" **el punto inicial**, que tomará un valor a elegir (p.e: (1,1)) y una **tasa de aprendizaje** arbitraria que afectará a la velocidad de descenso del algoritmo, la cual en nuestro ejercicio tomará valores como 0.1 o 0.01.

$$w_j := w_j - \eta \frac{\partial E(u,v)}{\partial w_j} \quad (\text{ECUACIÓN GENERAL})$$

Como se puede observar de la ecuación general del gradiente descendente, el **gradiente** estará compuesto por las **derivadas parciales** de w (es decir, la derivada del gradiente en el parámetro u y la derivada del gradiente en v).

Calcularemos el valor de w hasta que se cumpla el máximo de iteraciones declarado o hasta cumplir un tope ($E(u,v) = 10^{-14}$).

```
def Gradiente_descendente:  
    w := [u, v]  
    contador := 0  
    tasa_aprendizaje :=  
    tope :=  
  
    while (contador < MAXIMO) and (funcion(w.u, w.v) < tope)  
        contador +=1  
        w := w - tasa_aprendizaje * gradiente(w.u, w.v)  
    end
```

Consideramos la función:

$$E(u,v) = (ue^v - 2ve^{-u})^2$$

Para encontrar un mínimo de esta función, comenzaremos desde el punto $(u,v) = (1,1)$, una tasa de aprendizaje de 0.1, y un gradiente que estará formado por la expresión:
 $E(u,v) = [\partial/\partial u, \partial/\partial v]$, donde:

$$\partial/\partial u[f(u)] = f'(u) = 2(e^v u - 2ve^{-u})(2ve^{-u} + e^{-v})$$

$$\partial/\partial v[f(v)] = f'(v) = 2(ue^v - 2e^{-u})(ue^v - 2e^{-u} v)$$

El algoritmo tardará 23 iteraciones en alcanzar un valor inferior a 10^{-14} , en las coordenadas $w=[0.30249569478234256, 0.2674254631241295]$.

Consideramos la función:

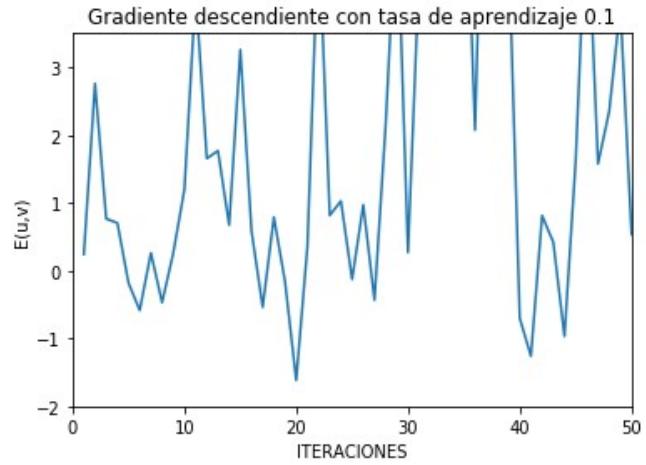
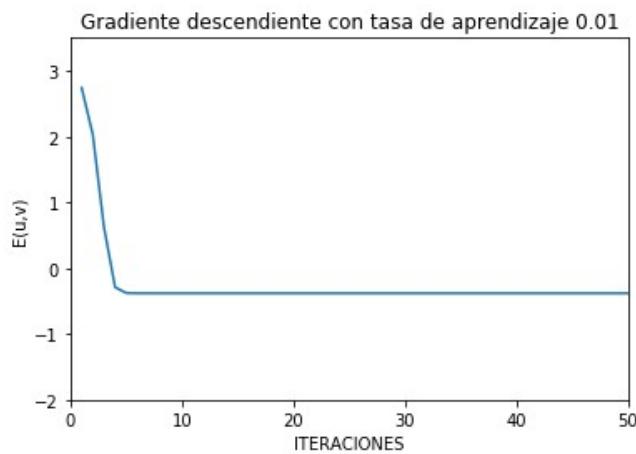
$$f(x, y) = (x - 2)^2 + 2(y + 2)^2 + 2 \sin(2\pi x) \sin(2\pi y)$$

Para encontrar un mínimo de esta función, comenzaremos desde el punto $(u,v) = (1,-1)$ y un máximo de 50 iteraciones. Ahora compararemos la tasa de aprendizaje con los valores 0.01 y 0.1.

El gradiente: $E(u,v) = [\partial/\partial x, \partial/\partial y]$, será:

$$\begin{aligned}\partial/\partial x[f(x)] &= f'(x) = 4\pi \sin(2\pi y) \cos(2\pi x) + 2(x-2) \\ \partial/\partial y[f(y)] &= f'(y) = 4\pi \sin(2\pi x) \cos(2\pi y) + 4(y+2)\end{aligned}$$

Tras ejecutar las 50 iteraciones, obtenemos los siguientes gráficos:



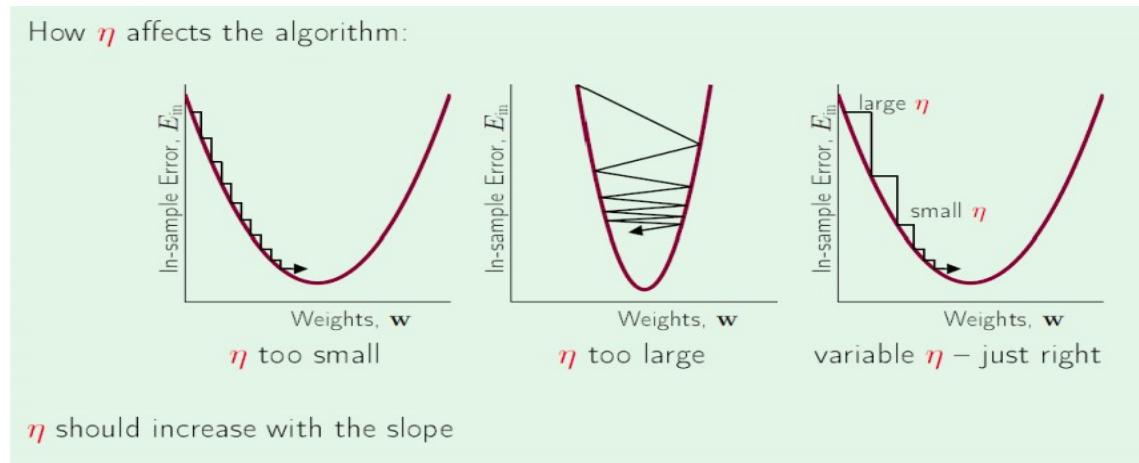
Esto nos demuestra la **importancia de elegir una buena tasa de aprendizaje**. En la gráfica de la izquierda, donde la tasa de aprendizaje es del 0.01, observamos que encuentra un mínimo local antes de llegar incluso a hacer las 10 primeras iteraciones y, sin embargo, los valores se ven estancados y no se sale de este mínimo. En contraste, en la gráfica de la derecha y con una tasa de aprendizaje del 0.1, obtenemos valores de w que pasan de un extremo a otro (pasan de valores más pequeños a valores muy grandes), con los que no podemos conseguir llegar a ningún mínimo.

A continuación obtenemos el valor mínimo y sus valores(x,y) cuando los puntos de inicio son fijados en: (2.1,-2.1), (3,-3) y (1,-1)

PUNTO INICIAL	COORDENADAS PUNTO	VALOR FUNCIÓN
[2.1,-2.1]	[2.257186, -2.187067]	-1.7076569338471193
[3,-3]	[2.275360,-2.250701]	-1.7731184350193216
[1.5, 1.5]	[1.777924, 1.032056]	18.042078009957635
[1,-1]	[1.2690643, -1.286720]	-0.3812494974381009

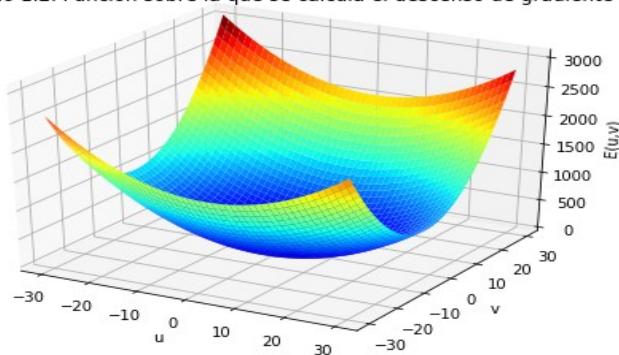
Como conclusión, con una **tasa de aprendizaje muy grande**, los cambios en w serán también muy grandes y aumentará la dificultad para encontrar coeficientes que minimicen la función proporcionada. Una **tasa de aprendizaje muy pequeña** ralentizará la convergencia del algoritmo para encontrar una solución adecuada. Por lo cual, es adecuado encontrar un valor lo suficientemente pequeño como para que decrezca en cada iteración.

Esta exactitud a la hora de elegir la tasa de aprendizaje proporciona una gran dificultad a la hora de encontrar un mínimo, estando la posibilidad de estancarse en un mínimo local, por lo que nunca llegaríamos al mínimo global.



Por último, mostramos una representación 3D de nuestro algoritmo gradiente descendente:

Ejercicio 1.2. Función sobre la que se calcula el descenso de gradiente



Ejercicio 2: Regresión Lineal

Para este ejercicio, se pide implementar el algoritmo de **descenso del gradiente estocástico**, para estimar un modelo de regresión lineal a partir de dos características (valor medio del nivel de gris y simetría del número respecto de su eje vertical) de vectores de dígitos manuscritos; seleccionando las imágenes de los números 1 y 5.

Además, calcularemos también el algoritmo de la pseudo-inversa, para comparar ambos modelos de regresión lineal.

Calculamos la pseudo-inversa mediante:

$$\mathbf{w} = \mathbf{X}' \mathbf{y}$$

donde:

$$\mathbf{X}' = (\mathbf{X}' \mathbf{X})^{-1} \mathbf{X}'$$

siendo X la matriz que contiene las características de nuestro conjunto de datos.

```
def pseudoinversa:  
    return (inversa(x_traspuesta*x)*x_traspuesta)*y
```

Implementamos el algoritmo de gradiente descendente estocástico:

$$\frac{\partial E_{in}(\mathbf{w})}{\partial w_j} = 2/M \sum_{n=1}^M \mathbf{x}_{nj} (\mathbf{h}(\mathbf{x}_n) - \mathbf{y}_n)$$

donde nuestra X_n representa la característica del ejemplo n e Y_n la etiqueta del ejemplo n y $\mathbf{h}(\mathbf{x}_n)$ será $\mathbf{x}_n \mathbf{w}^T$

A la hora de implementar el algoritmo, dividiremos el conjunto de datos en bloques de datos más pequeños (**"minibatches"**), los cuales tendrán tamaño M (donde M=128).

```

def SGD:
    para i  $\in \{0, M\}$ :
        minibatches_x, minibatches_y = generar_minibatches(tam=128)

        para cada dato "j" de minibatches_x:
             $h_x = \text{minibatches}_x[j] X^w$ 

            para cada característica "c" de nuestro dato del minibatche:
                 $w[c] := (2/M) * (\text{minibatches}_x[j][c] * h_x - \text{minibatches}_y[j])$ 
            w = w[c]

```

A parte de ambos algoritmos, también implementamos una función para calcular el Ein y Eout de la muestra a partir del error cuadrático:

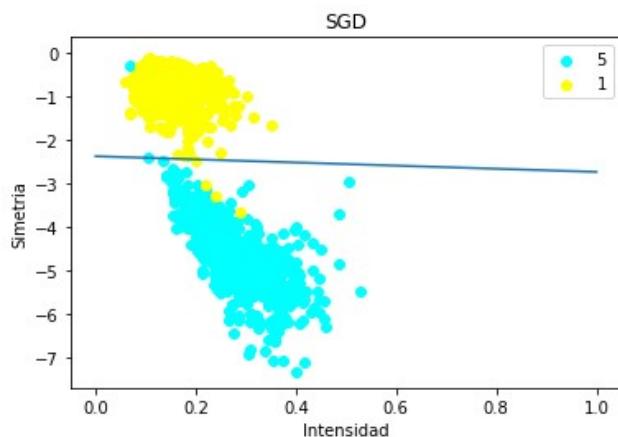
$$Ein(w) = 1/N \sum_{i=1}^N (w^T x_i - y_i)^2$$

```

def Err:
    para cada dato de x:
        calculamos el error cuadrático
        calculamos la media de los errores cuadráticos

```

Tras ejecutar ambos algoritmos, obtenemos:

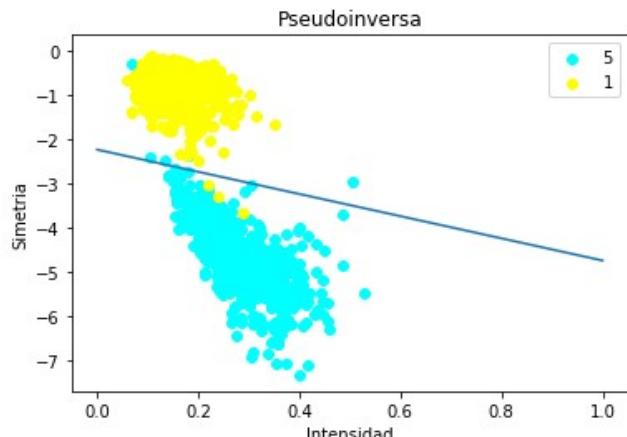


Gradiente descendente:

$$w = [-1.20373088 \quad -0.18144151 \quad -0.5048346]$$

Bondad de ajuste de:

$$Ein = 0.10966 \quad Eout = 0.1827$$



Pseudo-inversa:

$$w = [-1.11588016 \quad -1.24859546 \quad -0.49753165]$$

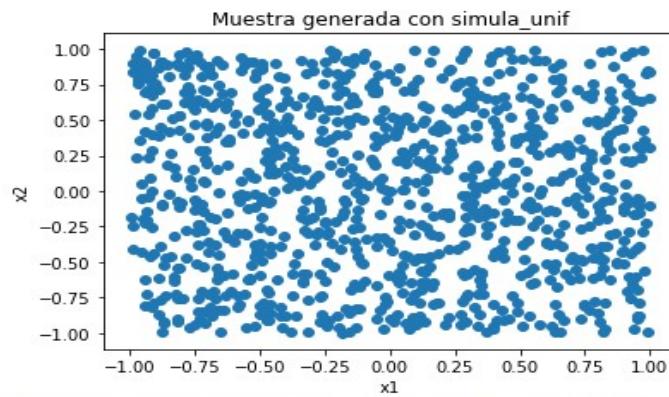
Bondad de ajuste de:

$$Ein = 0.0791 \quad Eout = 0.1309$$

Aunque en conclusión ambos modelos separan bien las clases y se obtengan errores pequeños, notamos en la pseudo-inversa obtenemos una recta con algo más de pendiente y con un error de entrada algo mejor que el del gradiente descendente.

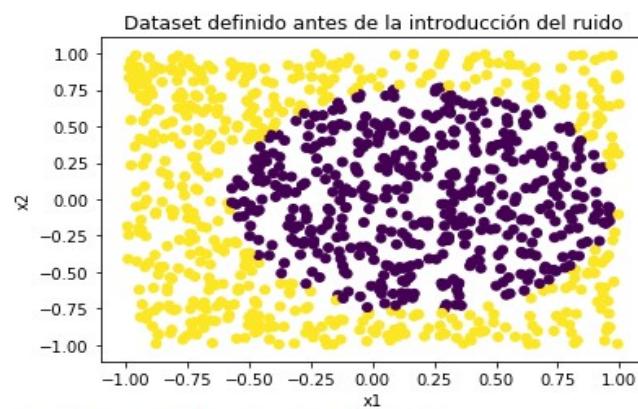
En el siguiente apartado, realizaremos un experimento:

Primero, generaremos una muestra de entrenamiento de $N = 1000$ puntos en el cuadrado $X = [-1, 1] \times [-1, 1]$. Para ello, utilizamos la función “**simula_unif**” proporcionada por el profesorado.



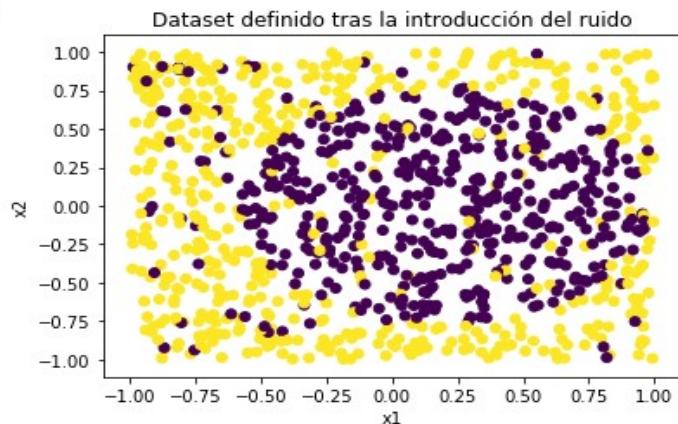
Consideraremos la función: $f(x_1, x_2) = \text{sign}((x_1 - 0,2)^2 + x_2^2 - 0,6)$ para asignar una etiqueta a cada punto de la muestra anterior.

```
def funcion_y:
    for i ∈{0..len(x)}
        y ← np.sign((x[i][0] - 0.2)² + x[i][1]² - 0.6)
```

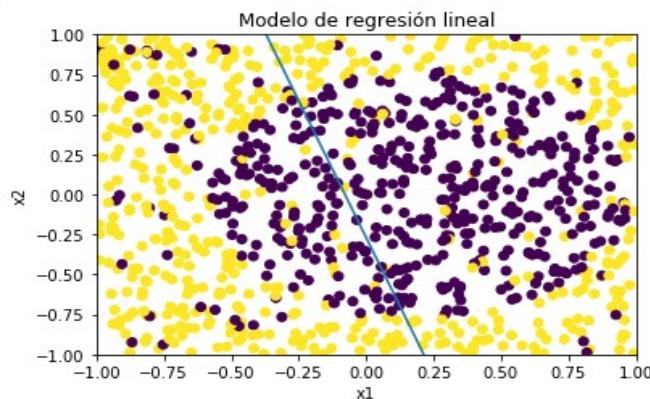


E introducimos ruido a un 10% aleatorio de las muestras:

```
def anadir_ruido:  
    for i ∈{0..len(x)*0.10}:  
        numero ← random(0..1000)  
        y[numero] ← -y[numero]
```



Por último, ajustamos el modelo aplicando nuestro algoritmo SGD a este conjunto de datos:



Obtenemos:

$$\mathbf{w} = [-0.03989143 \ -0.50616608 \ -0.14787035]$$

y una bondad del resultado de $E_{in} = 0.93181$

A continuación, repetiremos este experimento 1000 veces con 1000 conjuntos de pruebas aleatorios que se generarán en cada iteración, donde calcularemos el valor medio de los errores Ein y Eout.

Bondad del resultado para grad. descendente estocástico:

Ein medio 0.9326

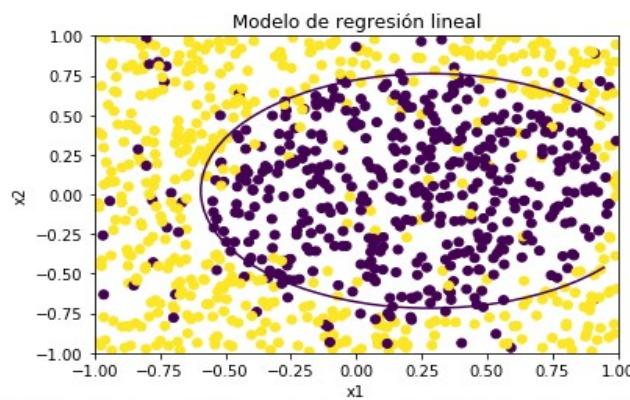
Eout medio 0.906

Es evidente que los valores de los errores son muy malos ya que están próximos a 1, conclusión también intuitiva al ver la representación de los puntos y su ajuste. Por tanto, podemos concluir que para problemas cuyos datos no estén separados con etiquetas de clases linealmente separables, como es en este caso donde los datos adoptan una figura de “elipse” (donde una parte de los datos están introducidos en la otra), la regresión lineal será poco útil al no ser capaz de encontrar una recta que divida los datos correctamente.

Cabe destacar que el Ein, en este caso, es mayor al Eout debido a que al conjunto de datos de entrenamiento le hemos añadido el 10% del ruido, al contrario del conjunto test; por lo que es de esperar que se obtenga un resultado peor.

Ahora, realizaremos el mismo experimento pero con características no lineales. Usando como vector de características: $\Phi_2(x) = (1, x_1, x_2, x_1x_2, x_1^2, x_2^2)$.

Repetimos los mismos pasos que en el experimento anterior, y obtenemos los resultados:



Bondad del resultado para grad. descendente estocástico:

Ein medio: 0.5679

Eout medio : 0.5741

Notamos que los errores medios mejoran con respecto al experimento anterior.

Podemos concluir así, que para obtener un mejor resultado en problemas de regresión lineal, es preferible adoptar una función con características no lineales, ya que de esta manera se puede encontrar una recta que divida de una forma mas fiel los datos.

Ejercicio 3: Bonus – Método de Newton

Implementamos el algoritmo del Método de Newton para la función del ejercicio 1, apartado 3:

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

De esta manera, el método de newton será un algoritmo muy similar al gradiente descendente, donde utilizaremos la **matriz Hessiana invertida** la cual premultiplica a la dirección de máximo descenso.

La matriz hessiana de una función f de n variables, es la matriz cuadrada nxn de las **segundas derivadas parciales**, así que el valor de w se basará en las derivadas parciales de segundo orden.

Por tanto, nuestra matriz hessiana en este caso será:

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial xy} \\ \frac{\partial^2 f}{\partial xy} & \frac{\partial^2 f}{\partial y^2} \end{bmatrix}$$

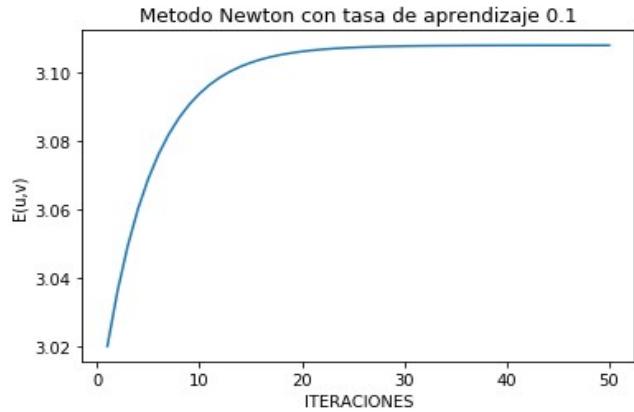
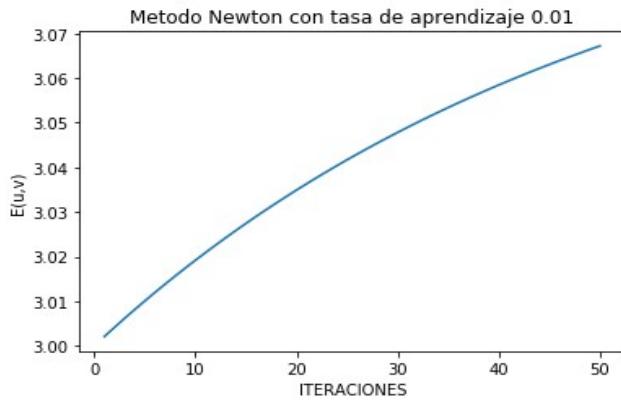
donde:

$$\begin{aligned}\frac{\partial^2 f}{\partial x^2}(x) &= f''(x) = 2 - 8\pi^2 \sin(2\pi y) \sin(2\pi x) \\ \frac{\partial^2 f}{\partial y^2}(y) &= f''(y) = 4(1 - 2\pi^2) \sin(2\pi x) \sin(2\pi y) \\ \frac{\partial^2 f}{\partial xy}(xy) &= f''(xy) = 8\pi^2 \cos(2\pi x) \cos(2\pi y)\end{aligned}$$

Implementamos el Método de Newton

```
def Metodo Newton:  
    w := [u, v]  
  
    while (contador < MAXIMO):  
        contador += 1  
        hessiana ← inversa(herssiana)  
        gradiente ← gradiente(w.u, w.v)  
  
        w := w - tasa_aprendizaje * (hessiana*gradiente)  
    end
```

Ejecutamos el algoritmo para un máximo de 50 iteraciones, valores iniciales [1,-1] y una tasa de aprendizaje tanto del 0.01 como del 0.1, siendo estos los mismos valores utilizados en el ejercicio 3.

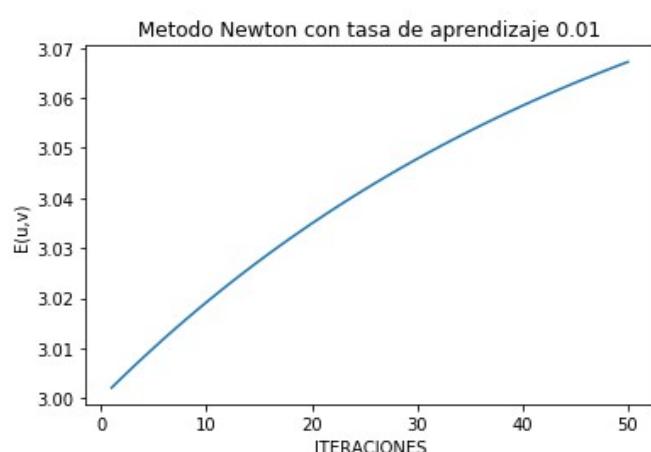


Al contrario que en el gradiente descendente, notamos que los valores que toma nuestra función a medida que avanza las iteraciones fluctúa más suavemente, siendo unos cambios más pequeños.

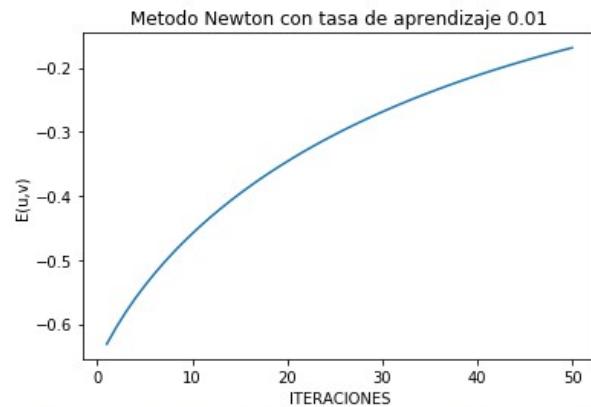
Comparamos gráficamente el algoritmo de Gradiente descendente con el Método de Newton:

-Tasa de aprendizaje: 0.01

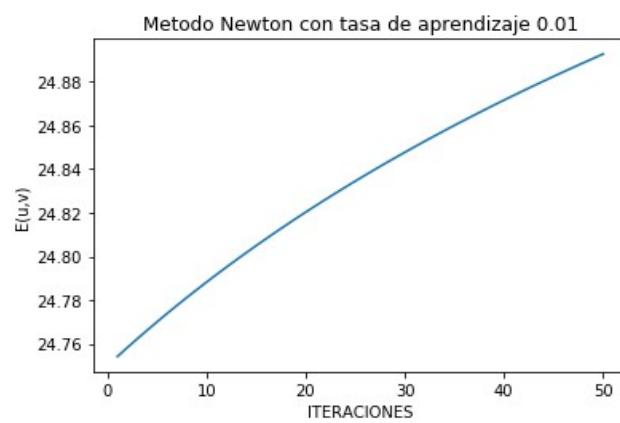
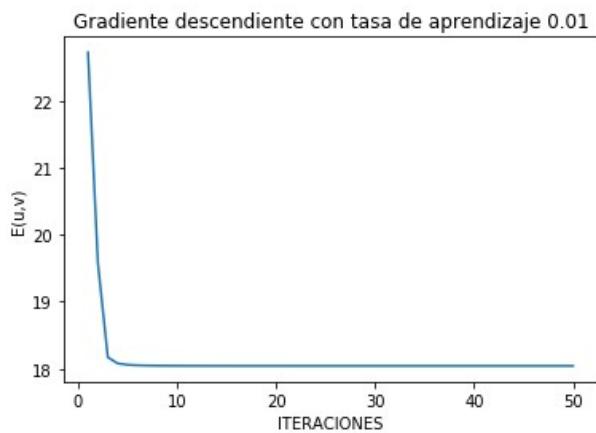
Puntos iniciales : [1,-1]



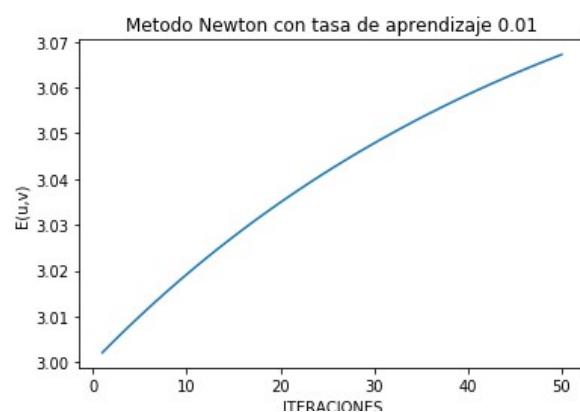
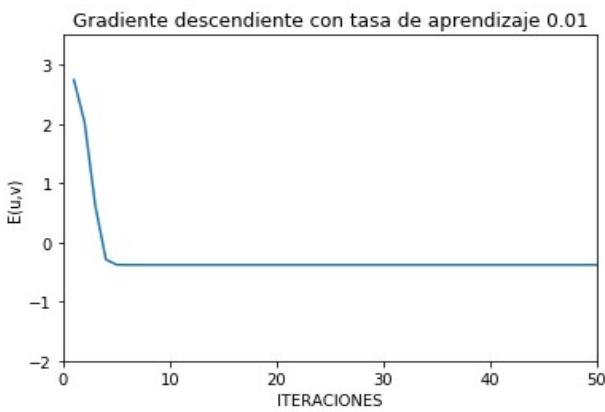
Puntos iniciales : [2.1,-2.1]



Puntos iniciales : [1.5,1.5]

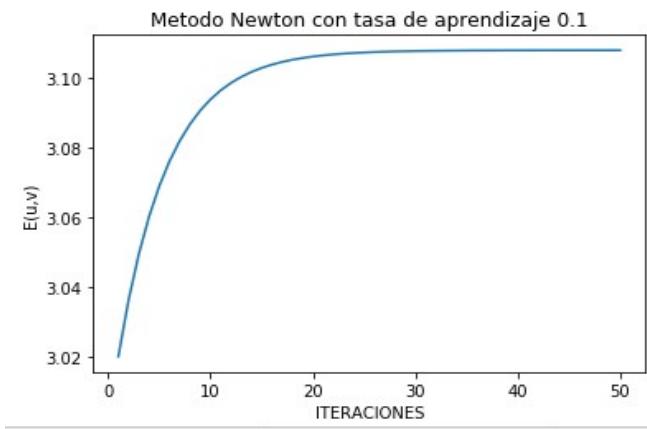
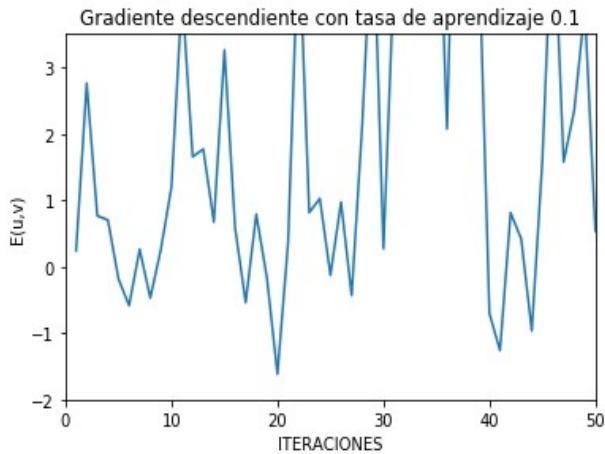


Puntos iniciales : [3,-3]

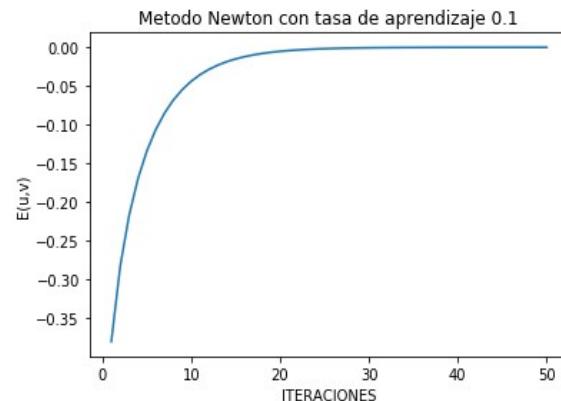
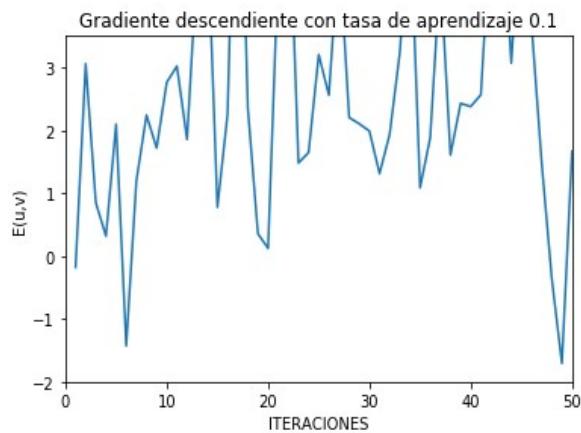


-Tasa de aprendizaje 0.1

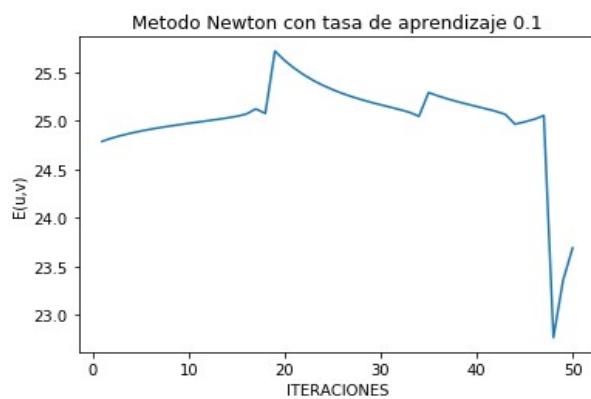
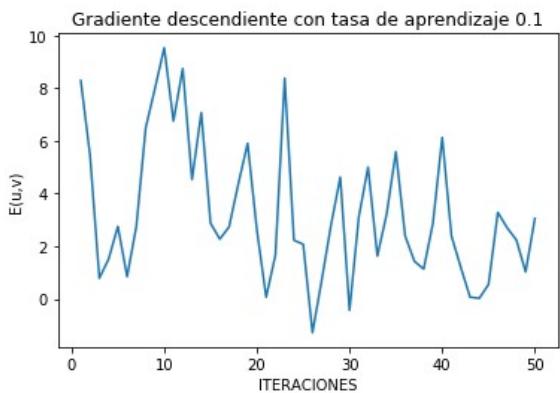
Puntos iniciales : [1,-1]



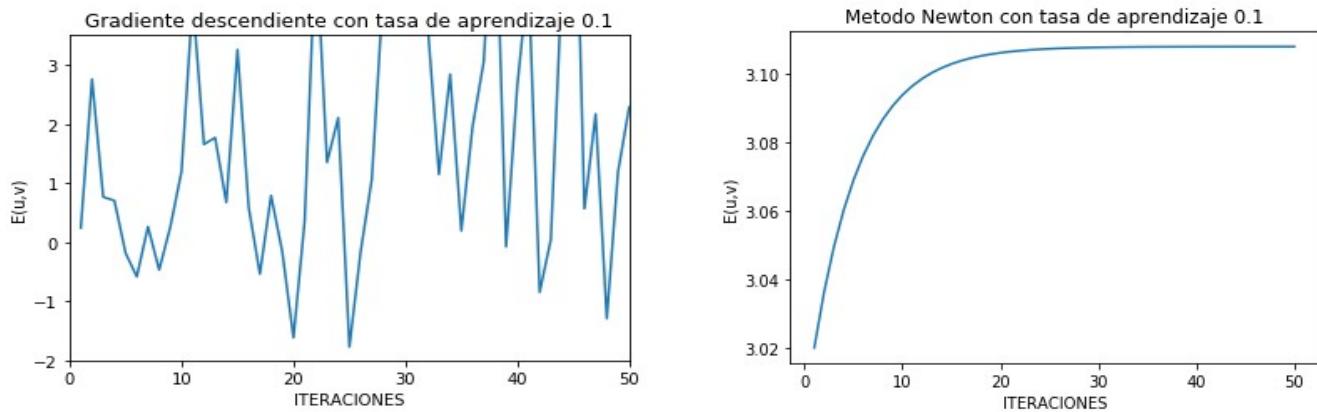
Puntos iniciales : [2.1,-2.1]



Puntos iniciales : [1.5,1.5]



Puntos iniciales : [3,-3]



Observamos que en el método de Newton, aunque variemos el valor de la tasa de aprendizaje, no obtenemos unos resultados muy buenos, ya que cambia de valores muy lentamente a lo largo de las iteraciones, al contrario del gradiente descendente, que es capaz de encontrar óptimos (aunque se estanque en ellos o de demasiados “saltos”).

Por tanto, aunque utilicemos la matriz Hessiana, y que los valores de las segundas derivadas pueden ofrecer una mayor velocidad de convergencia, no tenemos la seguridad de converger en un óptimo local (como hemos podido observar en las gráficas); además de tampoco saber si nos acercamos a un mínimo, máximo o punto de silla; por tanto, nos decidimos por el Gradiente Descendente como mejor algoritmo para este tipo de problemas.

Bibliografía

<https://www.calculadora-de-derivadas.com/>

<https://docs.python.org/3/library/>

<https://www.youtube.com/watch?v=hUfHUPPCxZ4>

<https://www.programcreek.com/python/example/56587/matplotlib.pyplot.title>

<https://iartificial.net/gradiente-descendiente-para-aprendizaje-automatico/>

[#La importancia de elegir un buen ratio de aprendizaje](#)

<https://planetachatbot.com/conceptos-fundamentales-en-machine-learning-funci%C3%B3n-de-perdida-y-optimizaci%C3%B3n-e30c25404622>

https://rua.ua.es/dspace/bitstream/10045/19734/3/Optimizacion_Problemas_sin_restricciones.pdf

https://es.wikipedia.org/wiki/Matriz_hessiana

https://www.math.ucla.edu/~wotaoyin/math273a/slides/Lec4_newton_methods_273a_2015_f.pdf