# APRENDIZAJE AUTOMÁTICO

## PRÁCTICAS - PRÁCTICA 1


Pedro Gallego López

Doble Grado de Ingeniería Informática y Matemáticas

*Universidad de Granada*
*Escuela Técnica Superior de Ingeniería Informática y Telecomunicaciones*


3 de abril de 2021

# Práctica 1

## Introducción

En esta práctica nos centramos en la búsqueda iterativa de óptimos a través del algoritmo del gradiente descendente y en la regresión lineal. Además habrá un bonus que tratará acerca del Método de Newton.

## Ejercicio 1. Búsqueda iterativa de óptimos.

Este Ejercicio está compuesto de varios apartados, todos relacionados con el uso del gradiente descendente. Hay un total de cuatro apartados que pasamos a comentar.

### Ejercicio 1.1. Implementar el algoritmo de gradiente descendente

Ya que el propio apartado pide la implementación del código, este se ve descrito en el cuadro de código 1.1.

Listing 1.1: Gradiente Descendente

```
def gradient_descent(w, lr, fun, gradf, iterations=50, error=-np.inf ):
   trayectoria=[w] # Puntos que vamos sacando con el gradiente descendente
   i=0
   umbral_superado=False
   while i<iterations and umbral_superado==False:

      f = fun(w[0],w[1])    # valor de la funcion
      if(f <= error):
         umbral_superado=True
      else:
         w_new = w - lr*gradf(w[0],w[1])

         if(w_new[0]==w[0] and w_new[1]==w[1]): # Si la solucion se ha estabilizado
            umbral_superado=True

         w=w_new

         trayectoria += [w.copy()]
         i = i+1

   return w, np.array(trayectoria), i, umbral_superado
```

El grueso de esta función está en la fórmula

$$w_{n+1} = w_n - lr * \nabla f(w_n) \tag{1.1}$$

La función $f$ y su gradiente $\nabla f$ son parámetros de la función: `fun` y `gradf` respectivamente. La ecuación recursiva 1.1 se introduce dentro de un bucle que hará tantas iteraciones como le digamos por parámetro (parámetro `iterations`) o hasta que se estabilice la solución, esto es que no haya cambio de una iteración a otra: $w_{n+1} == w_n \iff \nabla f(w_n) = 0$.

Con los parámetros `w` y `lr` establecemos el punto inicial del algoritmo y el learning rate.

Además se ha añadido el parámetro `error` para usarlo en ejercicios posteriores cuando haga falta. Este parámetro lo que hace es establecer un criterio de parada adicional. El criterio de parada se activa cuando

$$f(w_n) < error$$

y se refleja con el `if` de dentro del búcle del código 1.1.

**Ejercicio 1.2. Considerar la función $E(u,v) = (u^3e^{(v-2)} - 2v^2e^{-u})^2$ . Usar gradiente descendente para encontrar un mínimo de esta función, comenzando desde el punto $(u,v) = (1,1)$ y usando una tasa de aprendizaje $\eta = 0{,}1$**

En primer lugar se ha analizado la función haciendo un plot de ella que se puede observar en la figura 1.1.

Se puede intuir que por la forma que se ve que el algoritmo del gradiente descendente tiene que darnos en cada iteración un valor menor. Esto es así porque hay un amplio entorno con pendiente casi nula (en la gráfica 1.1 se ve como una base azul oscura) y por ello, aunque nuestro learning rate sea $\eta = 0{,}1$ los valores no van a variar mucho porque $\nabla f$ va a ser muy muy pequeño.

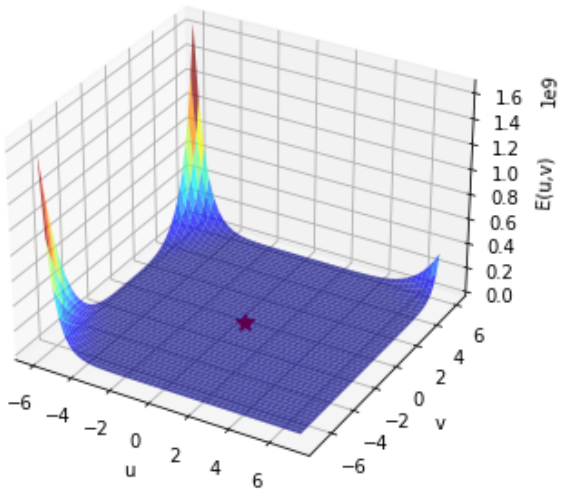


Figura 1.1: Función $E(u,v) = (u^3e^{(v-2)} - 2v^2e^{-u})^2$

**Ejercicio 1.2.a) Calcular analíticamente y mostrar la expresión del gradiente de la función $E(u,v)$**

Dada la función $E$ tenemos que sus derivadas parciales son:

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

Así el gradiente sería simplemente

$$\nabla f(u,v) = \left(\frac{\partial E}{\partial u}(u,v), \frac{\partial E}{\partial v}(u,v)\right) \tag{1.2}$$

En el script de python existen funciones denominadas `dEu, dEv, gradE` que hacen referencia a las derivadas parciales y al gradiente respectivamente.

**Ejercicio 1.2.b) ¿Cuántas iteraciones tarda el algoritmo en obtener por primera vez un valor de $E(u, v)$ inferior a $10^{-14}$ .**

**Ejercicio 1.2.c) ¿En qué coordenadas $(u, v)$ se alcanzó por primera vez un valor igual o menor a $10^{-14}$ en el apartado anterior?**

Se ha tardado 10 iteraciones en superar ese umbral. El valor que ha conseguido superar el umbral ha sido

$$w_{u,v} = (u, v) = (1{,}1572888496465497, 0{,}9108383657484799)$$

En las gráficas 1.2 y 1.3 se puede observar los distintos valores que se han ido obteniendo con el algoritmo de gradiente descendente.

Partimos del dato $E(1, 1) = 0{,}1353352832366127$, el primer punto de la trayectoria de nuestro algoritmo descendente que . Por lo dicho al inicio del apartado, lo esperado es que los valores vayan disminuyendo, y como nuestro learning rate es 0.1, un valor notorio, la actualización de los pesos durante el proceso son visibles y se puede observar como se cumple lo esperado, los valores van disminuyendo.

En las gráficas 1.3 se puede ver la trayectoria desde una vista plana, y se observa como al hacer zoom el comportamiento de los pesos parece "fractal": en la gráfica de la izquierda de la figura 1.3 se ve un punto muy alejado del resto, pero si nos acercamos a ese "resto" pasa lo que vemos en la gráfica de la derecha, que vuelve a haber uno más alejado. Esto pasa porque *"se decelera el proceso"*, esto es, el gradiente cada vez es menor en valor absoluto, haciendo que los cambios sean menores y provocando este comportamiento. Esto corrobora nuestra primera intuición acerca de la gráfica 1.1 sobre su *gran base azul oscura*.

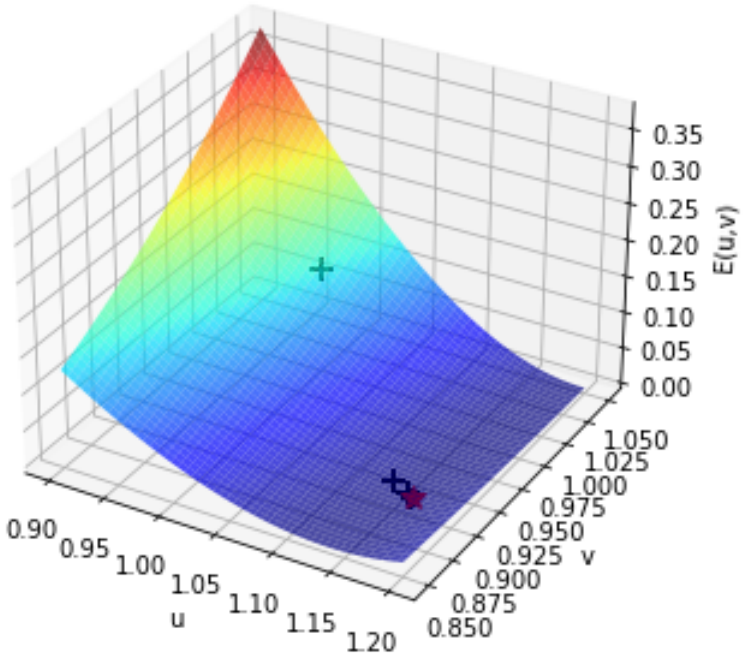


Figura 1.2: Función $E(u, v)$ del ejercicio 1.2 mostrando la trayectoria dada por el gradiente descendente. Se inicia en el punto (1,1). La estrella roja marca el punto final encontrado (donde se consiguió superar el umbral).

Otro aspecto a valorar sobre la *gran base azul oscura* es acerca de cómo "cambian de dirección" las curvas de nivel según hacemos zoom. Pero hay otra cuestión que genera duda, cuestión que se viene a la cabeza cuando observas la gráfica de la derecha de la Figura 1.3 y es: ¿Por qué la trayectoria no va de manera perpendicular a las curvas de nivel? La realidad es que estas dos cuestiones están íntimamente relacionadas: los cambios son tan pequeños y a un nivel tan local, que según nos acerquemos o nos alejemos veremos ligeramente perturbadas estas lineas de nivel puesto que se corresponden a escalas distintas, al ser tan minúsculos los cambios, las curvas de nivel no están muy bien definidas lo que provoca esta "incoherencia" visual.

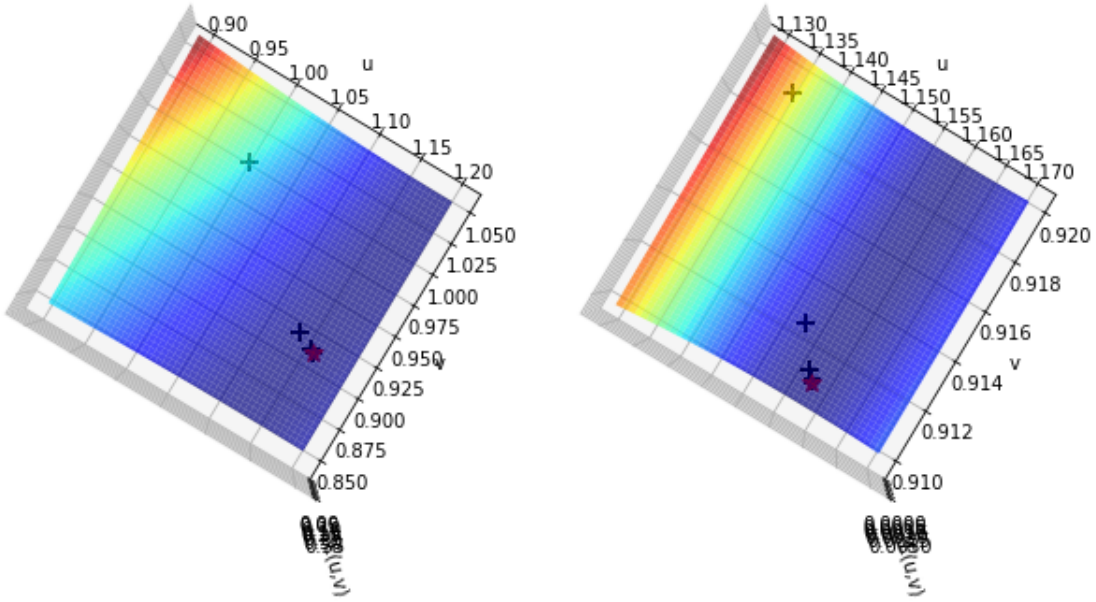


Figura 1.3: Misma gráfica que la representada en la figura 1.2 pero vista desde otra perspectiva. En la imagen de la izquierda se recogen todos los puntos. En la imagen de la derecha se ha hecho un zoom suprimiendo el valor más alejado (1,1) para poder ver mejor qué pasaba en los puntos que estaban más pegados.

**Ejercicio 1.3. Considerar ahora la función**

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

Vamos a observar esta función como hicimos en el apartado anterior. En la Figura 1.4 se puede ver un gráfico de nuestra función $f$ del enunciado. Salta a la vista la forma de la base de nuestro *"paraboloide"*. Esta forma es debida a los senos de la función, provoca que existan muchos mínimos locales que como se puede intuir a priori, va a ser el gran problema del ejercicio: encontrar un mínimo intentando que sea global y no únicamente local.

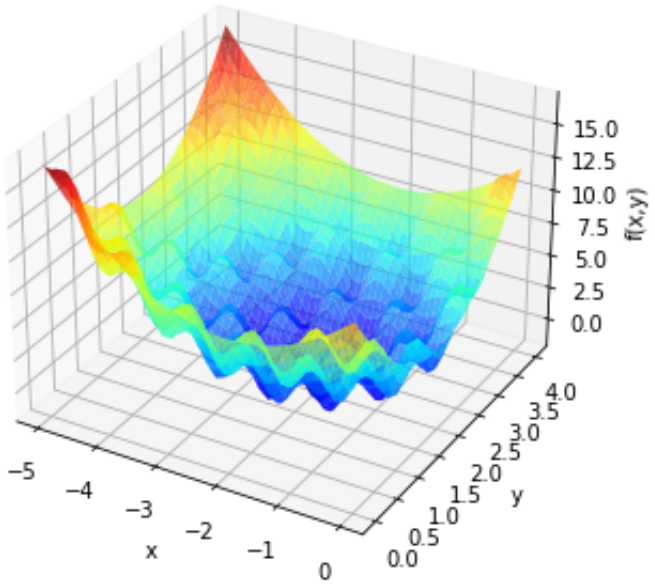


Figura 1.4: Función $f(x,y) = (x+2)^2 + 2(y-2)^2 + 2sin(2\pi x)sin(2\pi y)$

No hay que olvidar que la gráfica de nuestra función $f(x,y)$ es un paraboloide al que se le han añadido una perturbación $\phi(x,y) = 2sin(2\pi x)sin(2\pi y)$. Por lo que define claramente que los mínimos se van a encontrar en la bola $B((-2,2),\sqrt{2}) \subset \mathbb{R}^2$: el radio es $\sqrt{2}$ ya que $||\phi||_\infty = 2$ y es fácil ver a ojo que el mínimo global se va a alcanzar en un valor $w$ donde cumpla que:

- Está cerca del punto donde se alcanza el mínimo del paraboloide (-2,2).
- $(x,y)$ son tal que $\phi(x,y) = -2$

Es más, podemos afirmar que todos los mínimos locales candidatos se alcanzan en los puntos

$$(x,y) \in \mathbb{R}^2 : \phi(x,y) = -2 \tag{1.3}$$

podemos afirmarlo debido a que la perturbación $\phi$ añade "hoyos" y todos los hoyo añadidos dentro de la bola $B((-2,2),\sqrt{2}) \subset \mathbb{R}^2$ consiguen mínimos inferiores, y todos esos mínimos cumplen (1.3) . Luego podemos afirmar que el mínimo global será:

$$(x,y) \in \mathbb{R}^2 \; tal \; que \begin{cases} \phi(x,y) = -2 \\ ||(x,y) - (-2,2)||_2 = \text{mín}_{(x',y')} \, ||(x',y') - (-2,2)||_2 \end{cases}$$

**Ejercicio 1.3.a) Usar gradiente descendente para minimizar esta función. Usar como punto inicial $(x,y) = (-1,1)$ (tasa de aprendizaje $\eta = 0{,}01$ y un máximo de 50 iteraciones. Generar un gráfico de cómo desciende el valor de la función con las iteraciones. Repetir el experimento pero usando $\eta = 0{,}1$, comentar las diferencias y su dependencia de $\eta$.**

Los resultados de las ejecuciones se pueden ver en las figuras 1.5 ($\eta = 0{,}01$) y 1.6 ($\eta = 0{,}1$). Comentaré cada una por separado y luego haré un comentario común sacando conclusiones.

Para el caso $\eta = 0{,}01$ reflejado en la Figura 1.5 Vemos como encuentra un mínimo local y se dirige directo a él. Ese mínimo no es un mínimo global: esto se puede ver fácilmente si nos fijamos en la gráfica de la derecha de la Figura 1.6 en donde tenemos una visión más extensa

de la función, si nos vamos al punto $(x, y) = (-1, 1)$ vemos como corresponde a uno de los "mínimos exteriores". Sabiendo esto pues vemos que el valor del learning rate es adecuado para encontrar mínimos locales pero no un mínimo global. El valor mínimo encontrado ha sido:

$$f(w) = f(-1{,}269064351751895, 1{,}2867208738332965) = -0{,}3812494974381$$

Ahora cabría preguntarse cuál es el valor que debería de tener el learning rate para escapar de los mínimos locales y encontrar el mínimo global. Sin embargo, esta pregunta la dejaremos apartada hasta estudiar el siguiente caso con $\eta = 0{,}1$. Otra cuestión es la importancia del punto inicial. Si el punto inicial estuviese situado cerca del mínimo global, entonces probablemente lo alcanzaríamos.

Un último comentario acerca de las gráficas de la Figura 1.5 y tiene que ver con la cuestión del final del apartado 1.2 que se había comentado sobre las gráficas de la función $E(u, v)$ de ese mismo apartado. Esta cuestión iba sobre cómo la trayectoria de los puntos sacados con el gradiente descendente no se veía que trazasen una perpendicular con respecto a las curvas de nivel aparantemente y sobre cómo las curvas de nivel según cómo de cerca las viésemos cambiaban las direcciones de las curvas de nivel. Recordamos que la respuesta que dimos a esta cuestión fue que en esa zona de la función el gradiente era muy pequeño y que por lo tanto las curvas de nivel varían mucho a escalas tan pequeñas. Ahora nos fijamos en la Figura 1.5 y vemos como en este caso que sí que la trayectoria traza una línea que en todo momento parece ser ortogonal a las curvas de nivel, y no es difícil darse cuenta de que el gradiente de esta función en estos puntos tiene valores mucho más grandes que en la función del apartado anterior, que es lo que marca la diferencia.

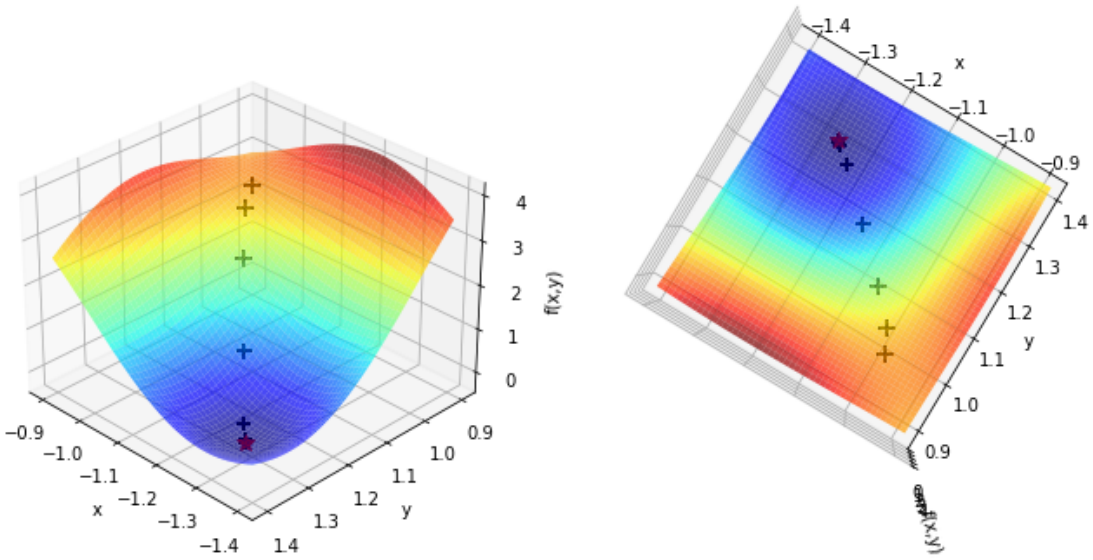


Figura 1.5: Resultado de ejecución del apartado 1.3.a) con learning rate $\eta = 0{,}01$

Ahora pasamos a comentar la Figura 1.6 donde se ha usado el learning rate $\eta = 0{,}1$. En este caso vemos como no pasa para nada lo anterior, huye de cualquier mínimo local. Esto es debido al valor del learning rate, que es tan alto que hace que el salto que pegan los pesos de una iteración a otra hace que pasen del entorno de un mínimo local al entorno de otro mínimo local distinto.

Vamos a razonar los movimientos que hacen los puntos (los puntos más grandes son los primeros y los más pequeños los últimos, la estrella roja es el último punto). Nos hará falta recordar que nuestra $f$ es un paraboloide con una perturbación $\phi$.

Al coger el learning rate tan grande, se observa como los pesos saltan de un extremo a otro de la gráfica sin aparente sentido. En realidad vemos como sale de los mínimos locales intentando centrarse en el mínimo global, haciendo más caso a la parte del paraboloide de $f$ que a la de la perturbación. Sin embargo, hay que darse cuenta de que el gradiente descendente hace uso del gradiente, y que con las perturbaciones de los senos de $\phi$ se consigue que el gradiente pueda llegar a ser en norma mucho mayor al que tendría el paraboloide sin la perturbación, esto hace que de una iteración del algoritmo a otra, los valores cambien drásticamente. Por lo general, los valores deberían oscilar dentro de la bola $B((-2,2),\sqrt{2}) \subset \mathbb{R}^2$, pudiendo salirse de ella en algún determinado momento, pero acabaría volviendo puesto que fuera de la bola el gradiente aportado por el paraboloide es muy grande y haría que volviese.

Y hay que señalar que estamos en un caso en el que el punto inicial (-1,1) se encuentra dentro de la bola $B((-2,2),\sqrt{2}) \subset \mathbb{R}^2$, es decir, cerca del mínimo. Pero hay que señalar que si el valor estuviese mucho más alejado, simplemente por el valor del gradiente dado por el paraboloide y el learning rate de 0.1, se llegaría rápidamente otra vez a la bola $B((-2,2),\sqrt{2}) \subset \mathbb{R}^2$.

Dicho este razonamiento, al final de las iteraciones el valor obtenido por el algoritmo ha sido:

$$f(w) = (-2{,}4174662279682364, 2{,}234353042462809) = -0{,}7023784790522758$$

Un valor menor que el apartado con el learning rate de 0.01.

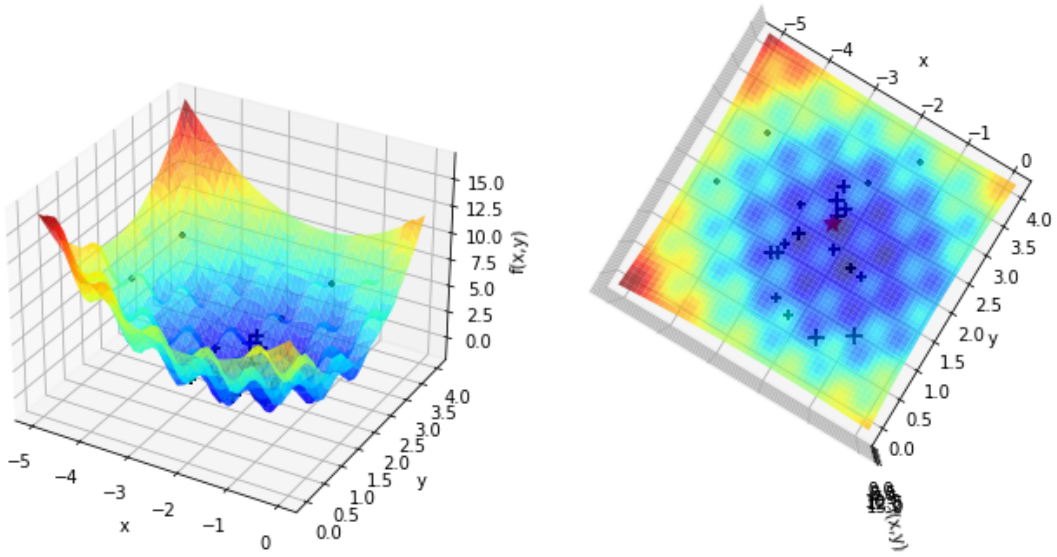


Figura 1.6: Resultado de ejecución del apartado 1.3.a) con learning rate $\eta = 0{,}1$

Pasamos ahora a la comparación de los learning rates y de los resultados obtenidos. A pesar de que al coger $\eta = 0{,}1$ parecía que nuestros nuevos pesos se asignaban casi sin sentido, se ha conseguido mínimo mejor que con $\eta = 0{,}01$, pero esto no quiere decir nada, quizás dejamos otra iteración y este valor se dispara.

En este caso concreto, el valor inicial de partida $w_0 = (-1, 1)$ es un valor muy bueno próximo al mínimo global, y por próximo entiéndase un valor lejano con respecto a esa superficie de perturbaciones. Con este valor inicial el learning rate $\eta = 0{,}01$ es un valor muy aconsejable puesto que te llevará a un óptimo local con total seguridad, aproximándose mucho al valor del mínimo global. Por contra, el learning rate de $\eta = 0{,}1$ es muy alto, y no nos dará certeza de

unos resultados buenos, puesto que de una iteración a otra hay mucho cambio.

Pero imaginemos que llegamos a coger un valor inicial lejano, las perturbaciones provocadas por $\phi$ podrían generar mínimos locales con valores súper altos, muy lejanos de la base del paraboloide (-2,2). En esta situación el learning rate $\eta = 0{,}01$ sería un valor pésimo puesto que interrumpiría la acción del paraboloide empujando los pesos hacia su base, y haría caso a las perturbaciones quedándose seguramente en ese mínimo local tan elevado. Por otro lado, el learning rate $\eta = 0{,}1$ conseguiría escapar de ese mínimo local elevado para llegar a la base del paraboloide y volver al mismo punto que estaría si hubiese empezado directamente desde el valor inicial (-1,1). Por lo que en este caso el $\eta = 0{,}1$ sería más recomendable.

Como conclusión se puede destacar la importancia de la elección conjunta del learning rate y del punto inicial. Lo ideal sería escoger un learning rate alto cuando cojamos un valor inicial lejos del mínimo global, y conforme nos vayamos acercando disminuir ese learning rate. Esto, evidentemente, no es tan fácil, puesto que el mínimo global no es conocido y aquí es donde gana importancia también estudiar la superficie siempre que sea posible, para hacernos una idea de los valores que va a arrojar el algoritmo del gradiente descendente.

**Ejercicio 1.3.b) Obtener el valor mínimo y los valores de las variables $(x, y)$ en donde se alcanzan cuando el punto de inicio se fija en: (-0,5, -0,5),(1, 1), (2,1, -2,1),(-3, 3),(-2, 2), Generar una tabla con los valores obtenidos. Comentar la dependencia del punto inicial.**

Learning rate $\eta = 0{,}01$

| $(w_0[0], w_0[1])$ | **(-0,5, -0,5)** | **(1, 1)** | **(2,1, -2,1)** | **(-3, 3)** | **(-2, 2)** |
|---|---|---|---|---|---|
| $f(w_0)$ | 14.75 | 11.0 | 49.739 | 3.0 | $4{,}79 * 10^{-31}$ |
| $w_f[0]$ | -0.7935 | 0.6774 | -0.3027 | -2.73 | -2.0 |
| $w_f[1]$ | -0.126 | 1.29 | 0.3441 | 2.713 | 2.0 |
| $f(w_f)$ | 9.1251 | 6.438 | 6.7945 | -0.3812 | $4{,}79 * 10^{-31}$ |

Learning rate $\eta = 0{,}1$

| $(w_0[0], w_0[1])$ | **(-0,5, -0,5)** | **(1, 1)** | **(2,1, -2,1)** | **(-3, 3)** | **(-2, 2)** |
|---|---|---|---|---|---|
| $f(w_0)$ | 14.75 | 11.0 | 49.739 | 3.0 | $4{,}79 * 10^{-31}$ |
| $w_f[0]$ | -1.6394 | -0.7 | -2.285 | -3.269 | -1.65584 |
| $w_f[1]$ | 1.2 | 1.61 | 1.12 | 0.938 | 2.2763 |
| $f(w_f)$ | 2.87 | 0.754 | 0.29 | 4.6 | 1.9 |

Las ejecuciones se han llevado a cabo con un máximo de 100 iteraciones para que diera tiempo a que el algoritmo llegase a un buen mínimo.

En las dos tablas de arriba se ve claramente lo explicado en el apartado anterior en donde ya se ha hablado de la importancia del punto inicial y su relación estrecha con el learning rate. Se observa como en los puntos iniciales que tienen valores más altos, al poner un learning rate mayor (de 0.1) se consigue llegar a mejores mínimos. Sin embargo, cuando estamos ya cerca del mínimo, como pasa con las dos últimas columnas, un learning rate pequeño nos asegura una buena convergencia al mínimo, mientras que con un learning rate alto corremos el peligro de salirnos fuera del mínimo, que es lo que pasa, en ambas columnas acabamos con unos pesos peores a los pesos iniciales.

Por ello se vuelve a remarcar la importancia del punto inicial y del learning rate. Haciendo énfasis en el punto inicial, que marca cómo de cerca estamos en un inicio y condicionando de principio a fin la búsqueda de ese mínimo.

### Ejercicio 1.4. ¿Cuál sería su conclusión sobre la verdadera dificultad de encontrar el mínimo global de una función arbitraria?

La dificultad es inmensa, acabamos de abordar dos ejercicios con dos funciones distintas y se ha visto la complejidad que reside en ellas y lo distintas que son. El hecho de elegir un learning rate correcto y unos buenos pesos iniciales hacen que la tarea sea compleja.

Si a esto le añadimos el hecho de no conocer la superficie sobre la que vamos a estudiar los mínimos, la complejidad aumenta exponencialmente. Los estudios analíticos de las funciones no servirían, el uso del gradiente se tendría que estimar de otra forma y todo sería más complejo. Los parámetros como el learning rate pasarían a ser parámetros a los que se les asignarían valores según experiencias empíricas en caso de que el problema sea suficientemente complejo, y en cuanto a los pesos iniciales nada te asegura que estés escogiendo unos buenos, por lo que también se resume en experiencias empíricas o incluso en asignar valores aleatorios.

De primeras el hecho de pensar en hallar los mínimos de una superficie que ni sabemos cómo es se hace ya complicado. Por lo tanto, la verdadera dificultad de encontrar el mínimo global de una función arbitraria es tratar de encontrar métodos que consigan aproximarnos a un valor que sea lo más parecido a un buen mínimo local.

## Ejercicio 2. Regresión Lineal.

Este ejercicio ajusta modelos de regresión a vectores de características extraídos de imágenes de dígitos manuscritos. En particular se extraen dos características concretas que miden: el valor medio del nivel de gris y la simetría del número respecto de su eje vertical. Solo se seleccionarán para este ejercicio las imágenes de los números 1 y 5

### Ejercicio 2.1. Estimar un modelo de regresión lineal a partir de los datos proporcionados por los vectores de características (Intensidad promedio, Simetría) usando tanto el algoritmo de la pseudo-inversa como el Gradiente descendente estocástico (SGD). Las etiquetas serán $\{-1, 1\}$, una por cada vector de cada uno de los números. Pintar las soluciones obtenidas junto con los datos usados en el ajuste. Valorar la bondad del resultado usando $E_{in}$ y $E_{out}$ (para $E_{out}$ calcular las predicciones usando los datos del fichero de test).

Vamos a empezar describiendo el desarrollo del código para llevar a cabo el ejercicio. Se hablará primero de los aspectos comunes de los dos (cálculo del error) y posteriormente se hablará primero del ejercicio realizado con el SGD y luego con la pseudo-inversa.

**Errores** $E_{in}, E_{out}$

Para implementar los errores lo que se ha hecho ha sido una función que calcula el error cuadrático medio, utilizando las funciones precisas de numpy para que sea rápido. También se ha implementado el Gradiente del error, para así poder aplicarlo al SGD. La implementación de ambas se puede ver en el listado de código de 1.2.

Listing 1.2: Error Cuadrático Medio

```
# Error cuadratico
def Err(x,y,w):
   # np.dot=producto matricial
   # np.square = calcula el cuadrado de sus elementos
   # np.mean = calcula la media
   return np.square(np.dot(x,w)-y).mean()

# Gradiente del error cuadratico
def gradErr(X,Y,w):
   grad=[]

   for j in range(X.shape[1]):
      res = 0
      for n in range(X.shape[0]):
         res = res + X[n][j]*(np.sum(X[n]*w) - Y[n])

      res = 2*res/X.shape[0]
      grad.append(res)

   return np.array(grad)
```

Las fórmulas son:

$$Err(w) = \frac{1}{len(X)} \sum_{i=1}^{len(X)} (w^T x_i - y_i)^2$$

$$\frac{\partial Err}{\partial w_j}(w) = \frac{2}{len(X)} \sum_{i=1}^{len(X)} x_{nj}(w^T x_i - y_i)$$

Donde $len(X)$ hace referencia al tamaño del batch que se le pase.

### Gradiente Descendente Estocástico - SGD

En este apartado también indicaremos el código del SGD. Este se encuentra en el listado de código 1.3. A la función le pasamos por parámetro el número de épocas que queremos que se ejecute. Una época equivale a un paso de todas las muestras del conjunto train, es decir de toda una partición de minibatchs. Además le pasamos los datos de entrenamiento, sus etiquetas, unos pesos iniciales, el tamaño del minibatch y el learning rate.

Se hace uso de `sklearn.utils.shuffle` para barajar las muestras y crear estocasticidad en el proceso. Así, en cada época, los minibatchs que son pasados serán distintos a otras épocas. Se ha considerado que en caso de que hubiese minibatchs con distinto tamaño (porque el batchsize no hace una división con resto cero con el tamaño de la muestra) se coge el último minibatch más grande, tanto como elementos queden para terminar la muestra.

Como condiciones de parada tenemos que se llegue a una solución estable, o bien que se termine el número de épocas.

Listing 1.3: Gradiente Descendente Estocástico

```
def sgd(x, y, w, batch_size, epochs, lr): # en el video dice que parametros podemos
    incluir aqui
   if(batch_size<=0 and batch_size>len(x)):
      print('ERROR: El size del batch tiene que ser mayor que 0 y menor que la
          muestra')

   trayectoria=[w] # Puntos que vamos sacando con el sgd

   n=0        # Cuenta el batch por el que vamos
   for i in range(epochs*(len(x)//batch_size)):
      inicio = (n*batch_size)
      fin = ((n+1)*batch_size)

      x_shuffled, y_shuffled= shuffle(x,y)

      # Condicion que controla que los indices tengan sentido (Cogemos el ultimo
          minibatch mas grande)
      if( len(x)-fin < batch_size):
         fin = len(x)- 1
         n=0

      x_batch = x_shuffled[ inicio : fin ]
      y_batch = y_shuffled[ inicio : fin ]

      w_new = w - lr*gradErr(x_batch, y_batch, w)

      iguales=True
      for i in range(len(w)):
         if w_new[i]!=w[i]:
            iguales = False

      if(iguales):  # Nos quedamos en un punto de estabilidad
         return w, np.array(trayectoria)

      w = w_new

      trayectoria += [w.copy()]

   return w, np.array(trayectoria)
```

### Pseudo-Inversa

El código para implementar este método es bastante sencillo gracias a las funciones de numpy. Se puede ver en el listado de código 1.4 como ocupa únicamente una línea. Sin embargo, las operaciones que hay por dentro requieren mucho más coste computacional a medida que el tamaño del problema aumenta. La fórmula de la pseudo-inversa puede escribirse como:

$$PseudoInversa(X) = \left(X^T X\right)^{-1} X^T$$

En el código directamente multiplicamos la pseudo-inversa por el vector $y$ para que nos devuelva directamente los pesos.

Listing 1.4: Pseudo-Inversa

```
def pseudoinverse(X, y):
    return np.dot(np.dot(np.linalg.inv((np.dot(X.T,X))),X.T), y)
```

**Ejecución y resultados**

Para la ejecución del ejercicio, se ha hecho el siguiente proceso:

1. Cargado los datos (en donde ha sido necesario añadirlos con el formato $[1, dato1, dato2]$ para su posterior trato con los pesos.
2. Se ha aplicado el algoritmo SGD ó pseudo-inversa con los datos de entrenamiento y se han almacenado los pesos obtenidos
3. Se ha calculaod $E_{in}$ con los datos de entrenamiento y el $E_{out}$ con los datos test.
4. Se han pintado las gráficas

Los parámetros aplicados al sgd han sido:

- Punto inicial: $0 \in \mathbb{R}^3$ se ha aconsejado que es un buen valor inicial. Además es buen valor puesto que el con estos pesos nuestra función de regresión arrojaría el valor 0, que equidista de las etiquetas $-1$ y 1, luego tiene sentido.
- Tamaño minibatch: 32. Un tamaño que he considerado correcto con el que se arrojan buenos resultados.
- Épocas: 50. Dejamos tiempo para que se ajuste bien la recta, asgurándonos que no hará cálculos innecesarios gracias a nuestra condición de parada de que se llegue a una solución estable.
- Learning Rate: 0.01. Vista la experiencia he considerado que este valor puede ser uno bueno para llegar de forma efectiva a un mínimo local.

Así los datos obtenidos han sido:

Error con Gradiente Descendente Estocástico (etiquetas $\{-1, 1\}$)

| | |
|---|---|
| **TRAIN** $E_{in}$ | 0.08171000959872528 |
| **TEST** $E_{out}$ | 0.13698357684822055 |

Error con Pseudo-Inversa (etiquetas $\{-1, 1\}$)

| | |
|---|---|
| **TRAIN** $E_{in}$ | 0.07918658628900395 |
| **TEST** $E_{out}$ | 0.13095383720052578 |

Donde se ve que la pseudo inversa mejora ligeramente el error. Ambos errores (independientemente del algoritmo que mires) dependen de la etiqueta que se le ha asignado a cada clase puesto que si las etiquetas fuesen distintas estos errores también lo serían (cambiarían su magnitud). Haciendo el ejemplo con las etiquetas $\{-5, 5\}$ se obtienen los resultados:

Error con Gradiente Descendente Estocástico (etiquetas $\{-5, 5\}$)

| | |
|---|---|
| **TRAIN** $E_{in}$ | 2.0427502399681314 |
| **TEST** $E_{out}$ | 3.424589421205513 |

Error con Pseudo-Inversa (etiquetas $\{-5, 5\}$)

| | |
|---|---|
| **TRAIN** $E_{in}$ | 1.9796646572250989 |
| **TEST** $E_{out}$ | 3.2738459300131444 |

Donde claramente se ha modificado la dimensionalidad del error. Luego cuando hablamos de errores, hay que hablar de forma relativa y sabiendo que dimensionalidad es adecuada. Porque ambos pares de tablas reflejan la misma regresión siendo errores distintos, y todo es debido a las etiquetas.

Volviendo a escribir las etiquetas como pide el enunciado, se ve en las gráficas 1.7 y 1.8 se pueden ver las gráficas de clasificación resultante de los datos train y test tanto del algoritmo SGD como de la pseudo inversa respectivamente.

Hay que recordar que nuestra regresión nos ha dado un plano:

$$f(x_1, x_2) = w_0 + w_1 x_1 + w_2 x_2$$

Como nuestra clasificación tiene las etiquetas $\{-1, 1\}$ las curvas de nivel de $f$ que mejor representan las dos clases son la $f(x_1, x_2) = -1$ y $f(x_1, x_2) = 1$, luego para separar estos valores valdría coger la curva de nivel que equidiste de los valores 1 y -1, la curva de nivel $f(x_1, x_2) = 0$, que es una recta. Esta recta puede observarse en las gráficas 1.7 y 1.8

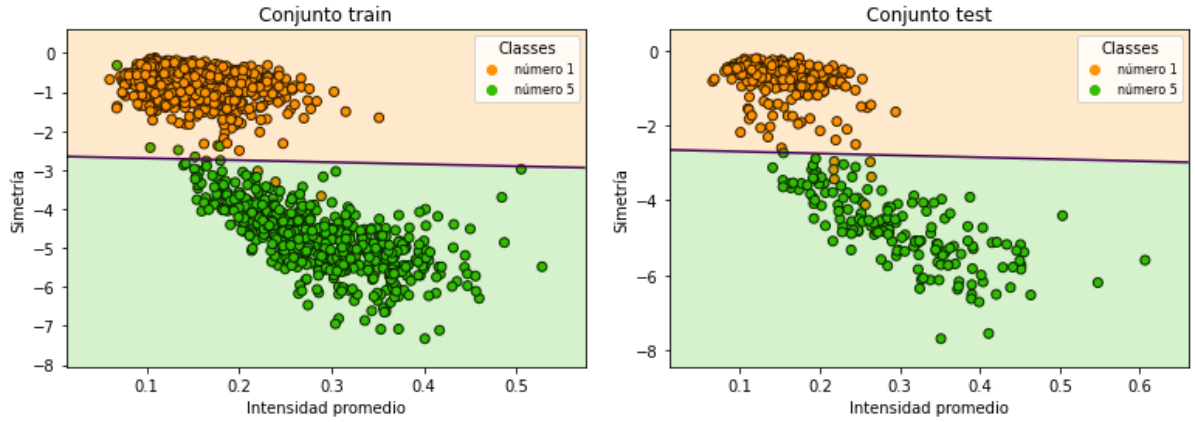


Figura 1.7: Resultado de ejecución del algoritmo de gradiente descendente estocástico para clasificar el número 1 y el número 5

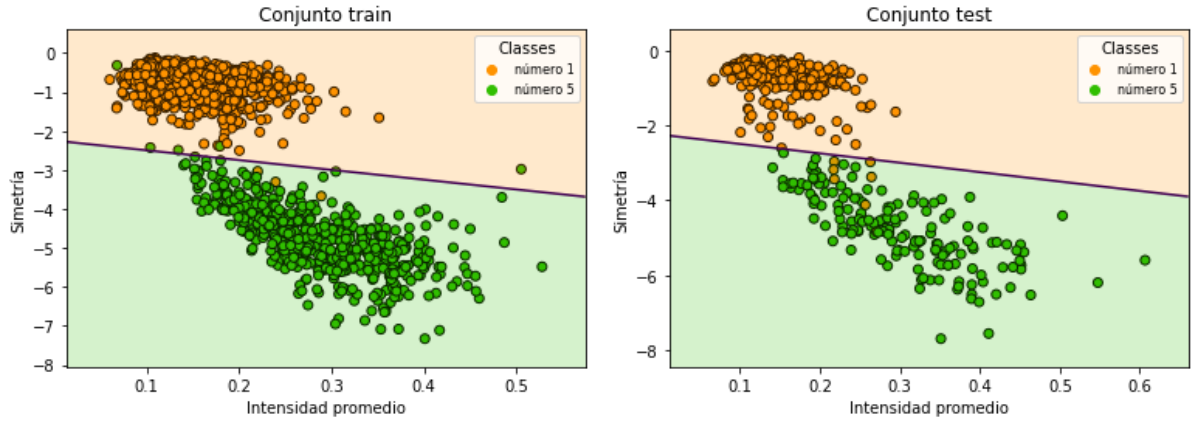


Figura 1.8: Resultado de ejecución del algoritmo de la pseudo-inversa para clasificar el número 1 y el número 5

En las gráficas 1.7 y 1.8 se puede observar como la pendiente de la recta separación de la pseudo inversa está algo más inclinada. Esta inclinación se puede apreciar que tiene sentido por

como están distribuidos visualmente los datos, y así lo corroboran los errores obtenidos, que mejoran con respecto al SGD.

### Nota respecto a duda surgida en clase

En una clase de prácticas se preguntó acerca de qué valores podrían tener las etiquetas, y se dijo que tendrían que ser siempre de la forma $\{-n.n\}$, porque sino las magnitudes afectarían a como se crea la recta. Sin embargo, ahondando en el tema hablamos por ejemplo si pusiésemos las etiquetas $\{1, 5\}$. He decidido desarrollar esta parte para aclarar un poco el tema.

Si escogemos las etiquetas $\{1, 5\}$ tendríamos que las curvas de nivel que mejor representan a las dos clases serían $f(x_1, x_2) = 1$ y $f(x_1, x_2) = 5$, y con el mismo razonamiento que antes, la curva de nivel que mejor separa ambas gráficas se encuentra en el nivel $\frac{1+5}{2} = 3$, luego si hacemos

$$3 = w_0 + w_1 x_1 + w_2 x_2$$

nos debería de salir una recta igual que en las gráficas 1.7 y 1.8, vamos a hacer la prueba solo con el sgd.

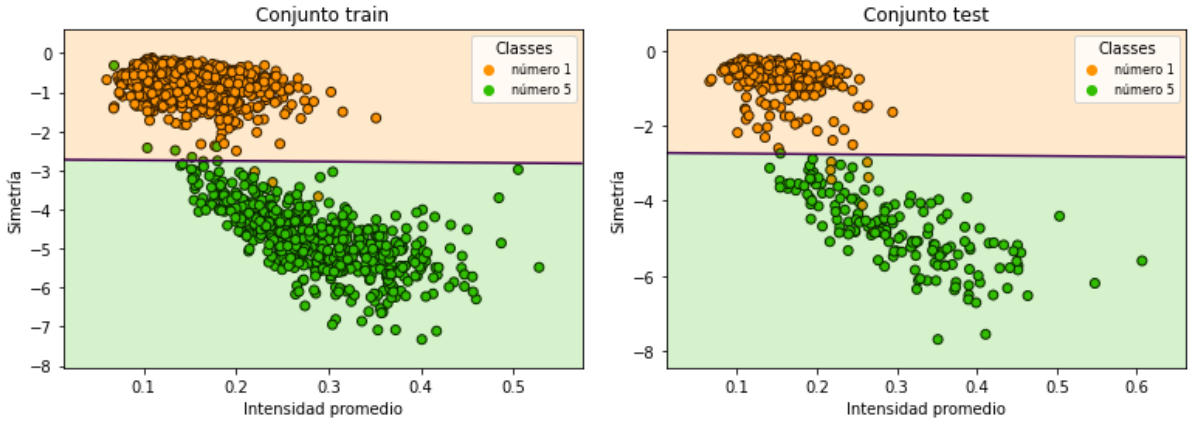


Figura 1.9: Resultado de ejecución del algoritmo sgd para clasificar el número 1 (label=1) y el número 5 (label=5)

Se ve en la figura 1.9 como la recta queda exactamente igual. Esto se debe a que el método de mínimos cuadrados solo se fija en la distancia que hay a cierto valor, luego lo importante es la **distancia**. Vamos a verlo matemáticamente. Supongamos que tenemos los dos problemas

$$label1_1 = -2, label5_1 = 2, \qquad f_1(x_1, x_2) = w_0 + w_1 x_1 + w_2 x_2 \tag{1.4}$$

$$label1_2 = 1, label5_2 = 5, \qquad f_2(x_1, x_2) = w_0 + w_1 x_1 + w_2 x_2 \tag{1.5}$$

Se ha puesto de etiqueta $\{-2, 2\}$ en (1.4) para que estén igual de distanciadas que $\{1, 5\}$, (recordemos que la magnitud del error cambia si distanciamos las etiquetas entre sí aunque los resultados de clasificación sean los mismos) y así que el error nos salga igual en una y otra y poder ver que efectivamente no ingluye poner las etiquetas $\{1, 5\}$. Como hemos dicho la recta que separaría bien las clases en (1.4) sería $f_1(x_1, x_2) = 0$ y en (1.5) sería $f_2(x_1, x_2) = 3$, pero además nos tenemos que fijar que

$$\begin{aligned} label1_2 &= 3 + label1_1 \\ label5_2 &= 3 + label5_1 \end{aligned} \tag{1.6}$$

Ahora calculemos los errores cuadráticos medios:

$$Err_2(X,Y) = \frac{1}{N}\sum_{(x_1,x_2)\in X} ||f_2(x_1,x_2) - labelN_2||^2 = \frac{1}{N}\sum_{(x_1,x_2)\in X} ||f_1(x_1,x_2)+3-(labelN_1+3)||^2 =$$

$$= \frac{1}{N}\sum_{(x_1,x_2)\in X} ||f_1(x_1,x_2)+3-3-labelN_1||^2 = Err_1(X,Y)$$

Siendo $Y$ un vector de $labelN_i$.

Vemos con esto que en ambos problemas se consigue exactamente el mismo error. Es más si nos damos cuenta, si cogemos el problema (1.4), calculamos los pesos por el algoritmo que sea, y obtenemos de pesos $\{w_0, w_1, w_2\}$ tendríamos que los pesos de (1.5) serían $\{(w_0 - 3), w_1, w_2\}$.

CONCLUSIÓN, el valor de las etiquetas es independiente, hay que estar atento para saber interpretar los resultados, tanto del error como de las curvas de nivel para la separación de clases.

**Ejercicio 2.2. En este apartado exploramos como se transforman los errores $E_{in}$ y $E_{out}$ cuando aumentamos la complejidad del modelo lineal usado. Ahora hacemos uso de la función $simula\_unif(N, 2, size)$ que nos devuelve $N$ coordenadas 2D de puntos uniformemente muestreados dentro del cuadrado definido por $[-size, size] \times [-size, size]$**

Se nos pide el siguiente experimento:

1. Generar una muestra de entrenamiento de N = 1000 puntos en el cuadrado X = [-1, 1] × [-1, 1]. Pintar el mapa de puntos 2D.

2. Consideremos la función $f(x_1, x_2) = sign((x_1 - 0,2)^2 + x_2^2 - 0,6)$ que usaremos para asignar una etiqueta a cada punto de la muestra anterior. Introducimos ruido sobre las etiquetas cambiando aleatoriamente el signo de un 10 % de las mismas. Pintar el mapa de etiquetas obtenido.

3. Usando como vector de características $(1, x_1, x_2)$ ajustar un modelo de regresión lineal al conjunto de datos generado y estimar los pesos w. Estimar el error de ajuste $E_{in}$ usando Gradiente Descendente Estocástico (SGD).

4. Ejecutar todo el experimento definido por (1)-(3) 1000 veces (generamos 1000 muestras diferentes) y

   - Calcular el valor medio de los errores $E_{in}$ de las 1000 muestras.
   - Generar 1000 puntos nuevos por cada iteración y calcular con ellos el valor de $E_{out}$ en dicha iteración. Calcular el valor medio de $E_{out}$ en todas las iteraciones.

5. Valore que tan bueno considera que es el ajuste con este modelo lineal a la vista de los valores medios obtenidos de $E_{in}$ y $E_{out}$

Para generar el dataset pedido en este ejercicio se ha definido una función que nos devuelve los conjuntos test y train con sus etiquetas. En el listado de código 1.5 se puede ver la implementación de este código. Para dar aleatoriedad se ha usado la función `np.random.choice` para cambiar las etiquetas de train y test.

Listing 1.5: Generación del dataset lineal

```
def generate_dataset_lineal(N,d,size, seed=1):
   # Generamos una muestra de entrenamiento de N=1000 puntos en [-1,1]x[-1,1]
   x = simula_unif(N, d, size)
   x_train = np.array([ [1, x[i][0],x[i][1]] for i in range(N) ])

   # Generamos una muestra de test de N=1000 puntos en [-1,1]x[-1,1]
   np.random.seed(seed)
   x = simula_unif(N, 2, 1)
   x_test = np.array([ [1, x[i][0],x[i][1]] for i in range(N) ])


   # Definimos las etiquetas
   y_train = np.array([f22b(x_train[i][1],x_train[i][2]) for i in range(N)])
   y_test = np.array([f22b(x_test[i][1],x_test[i][2]) for i in range(N)])

   v_change = np.random.choice([-1,1], N, p=[0.9,0.1])  # Aleatoriedad etiquetas
   y_train = y_train * v_change
   y_test = y_test * v_change

   return x_train, y_train, x_test, y_test
```

En una iteración de los puntos (1)-(3) en 50 épocas, con un learning rate de 0.1 y cogiendo como punto inicial el $w_0 = (0{,}5, 0{,}5, 0{,}5)$ se han obtenido los siguientes resultados:

Error con una ejecución de 50 épocas y características lineales

| | |
|---|---|
| **TRAIN** $E_{in}$ | 0.9576241420057854 |
| **TEST** $E_{out}$ | 0.9643022718800326 |

Con 50 épocas se han obtenido estos resultados. Los resultados son extremadamente malos a pesar de haberse entrenado un número de épocas considerable y con una tasa de aprendizaje alta para poder escapar de mínimos locales malos. En la gráfica 1.10 se ve el resultado de la regresión. Es visible que una recta es muy complicado que pueda separar estos dos conjuntos.

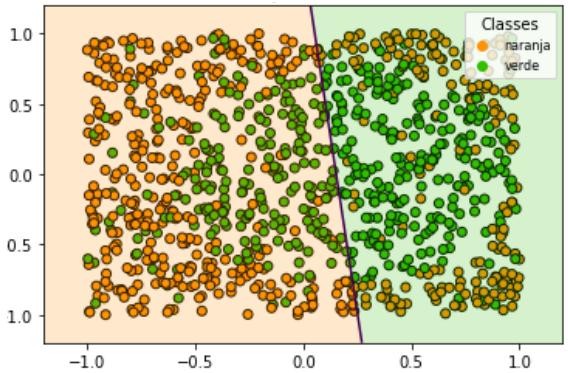


Figura 1.10: Resultado de ejecución de una iteración con 50 épocas y un lr=0.1 de los puntos del (1)-(3) del apartado 2.2. Con características lineales.

Para llevar a cabo todo el proceso vamos a rebajar el número de épocas de 50 a 15, puesto que requiere mucho coste computacional, y 1000 iteraciones de 50 épocas pueden llevarnos 40 minutos de ejecución. Un ejemplo de ejecución de 15 épocas es el que se puede observar en la gráfica 1.11.

Los datos han sido

Error con 1000 ejecuciones de 15 épocas y características lineales

| | |
|---|---|
| **TRAIN Media error** $\hat{E}_{in}$ | 0.931545749810305 |
| **TEST Media error** $\hat{E}_{out}$ | 0.9380155943717872 |

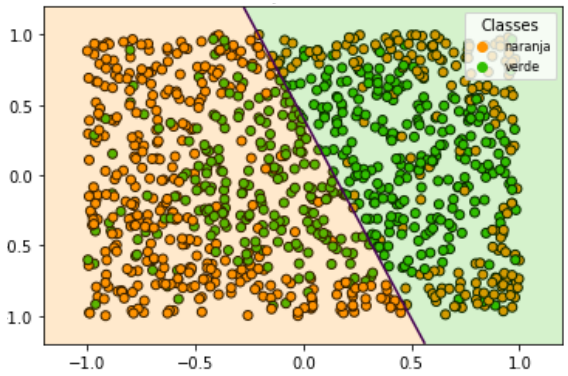


Figura 1.11: Resultado de ejecución de una iteración con 15 épocas y un lr=0.1, de los puntos del (1)-(3) del apartado 2.2. Características lineales. Ein: 0.959, Eout: 0.967

Con estos resultados se puede decir que no hemos sido de llegar a un método de clasificación, probablemente porque la clase de funciones que estamos usando no se ajusta al problema.

Ahora se nos pide repetir el mismo experimento anterior pero usando características no lineales. Ahora usaremos el siguiente vector de características: $\Phi_2(x) = (1, x_1, x_2, x_1x_2, x_1^2, x_2^2)$. Ajustar el nuevo modelo de regresión lineal y calcular el nuevo vector de pesos $\hat{w}$. Calcular los errores promedio de $E_{in}$ y $E_{out}$ .

Para crear este dataset se ha reutilizado el código 1.5. En el listado de código 1.6 se puede ver la función usada.

Listing 1.6: Generación del dataset no lineal

```
def generate_dataset_nl(N, d, size, seed):
    # Cogemos los datos de partida
    xo_train, y_train, xo_test, y_test = generate_dataset_lineal(N,d,size,seed)

    # Transformamos los datos
    x_train = np.array([[1, xo_train[i][1], xo_train[i][2],
        xo_train[i][1]*xo_train[i][2], xo_train[i][1]**2, xo_train[i][2]**2] for i in
        range(len(xo_train))])
    x_test = np.array([[1, xo_test[i][1], xo_test[i][2], xo_test[i][1]*xo_test[i][2],
        xo_test[i][1]**2, xo_test[i][2]**2] for i in range(len(xo_test))])

    return x_train, y_train, x_test, y_test
```

Con este nuevo vector de características aplicando también 50 épocas, un punto inicial $w_0 = (0{,}5, ..^{(6)}.., 0{,}5)$ un learning rate de 0.1, se ha conseguido mejorar los resultados como podemos ver en la siguiente tabla

Error con una ejecución de 50 épocas características no lineales

| | |
|---|---|
| **TRAIN** $E_{in}$ | 0.6002269393173719 |
| **TEST** $E_{out}$ | 0.6064915081151034 |

Comparándola con la primera tabla que se hizo con el mismo escenario solo que con características lineales, vemos que el error ha disminuido considerablemente, pero no sólo eso, si nos fijamos en la gráfica de esta ejecución (Figura 1.12) vemos como ahora nuestra clase de funciones si se ajusta mejor a nuestro modelo.

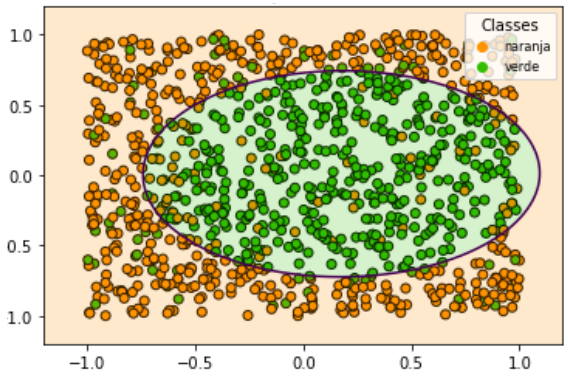


Figura 1.12: Resultado de ejecución de una iteración con 50 épocas y un lr=0.1, de los puntos del (1)-(3) del apartado 2.2. Características no lineales

Haciendo las 1000 ejecuciones de 15 épocas y características no lineales se obtienen los resultados de la siguiente tabla:

Error con 1000 ejecuciones de 15 épocas y características no lineales

| | |
|---|---|
| **TRAIN Media error** $\hat{E}_{in}$ | 0.5836724931988144 |
| **TEST Media error** $\hat{E}_{out}$ | 0.5916447114885925 |

Se observa como también mejora de la misma manera al del apartado anterior con características no lineales. Una ejecución de 15 épocas podemos verla en la gráfica 1.13

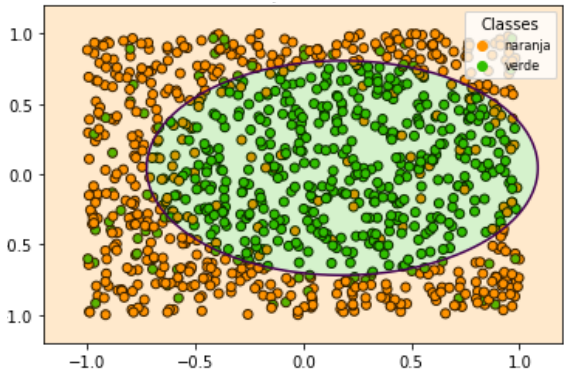


Figura 1.13: Resultado de ejecución de una iteración con 15 épocas y un lr=0.1, de los puntos del (1)-(3) del apartado 2.2. Características no lineales. Ein: 0.603, Eout: 0.608

NOTA: Todas las gráficas mostradas en este apartado son con el conjunto de entrenamiento.

### Conclusión

Para este conjunto de datos se considera que el modelo de características no lineales se aproxima mejor a una buena clasificación de puntos. A la vista de los errores está, pero además, visualmente se puede ver en las gráficas que la distribución de los puntos no sigue un modelo lineal, así que podemos sospechar que esa aproximación buena no va a ser.

Tampoco tenemos certeza de que la aproximación cuadrática que hemos usado en el segundo caso con $\Phi_2(x) = (1, x_1, x_2, x_1x_2, x_1^2, x_2^2)$ funcione correctamente, ya que a pesar de que la función $f$ con la que hemos asignado las etiquetas es una función cuadrática, la aleatoriedad que le hemos dado al 10 % de las etiquetas pueden modificar el modelo. Lo único que tenemos certeza es que aproximará mejor que la lineal, puesto que en este modelo se incluye el caso lineal. Efectivamente, si los pesos son $w = (w_0, w_1, w_2, w_3, w_4, w_5)$ y nuestra función

$$g(x_1, x_2) = w_0 + w_1x_1 + w_2x_2 + w_3x_1x_2 + w_4x_1^2 + w_5x_2^2$$

si cogemos $w_3 = 0, w_4 = 0, w_5 = 0$ tenemos que nuestra función se quedaría como

$$g(x_1, x_2) = w_0 + w_1x_1 + w_2x_2$$

que es precisamente la función lineal de la primera parte.

## Método de Newton-Raphson

**Implementar el algoritmo de minimización de Newton y aplicarlo a la función $f(x, y)$ dada en el ejercicio 1.3.**

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

**Desarrolle los mismos experimentos usando los mismos puntos de inicio. Generar un gráfico de como desciende el valor de la función con las iteraciones. Extraer conclusiones sobre las conductas de los algoritmos comparando la curva de decrecimiento de la función calculada en el apartado anterior y la correspondiente obtenida con gradiente descendente.**

En este método de actualización de pesos se hace uso, no solo de la pendiente de la superficie, sino también de su curvatura, teniendo así una información adicional. En concreto se utilizará el *Hessiano* de $f$ descrito en (1.7)

$$H(f)(x, y) = \begin{pmatrix} 2 - 8\pi^2 sin(2\pi x)sin(2\pi y) & 8\pi^2 cos(2\pi x)cos(2\pi y) \\ 8\pi^2 cos(2\pi x)cos(2\pi y) & 4 - 8\pi^2 sin(2\pi x)sin(2\pi y) \end{pmatrix} \tag{1.7}$$

Así el método de Newton-Raphson es:

$$w_{n+1} = w_n - H(f)^{-1}\nabla f \tag{1.8}$$

Sin embargo, usaremos un parámetro de learning rate para relajar el efecto del método y hacer el proceso más controlado. En la expresión (1.9) se ve como lo único que cambia con respecto a (1.8) es el learning rate.

$$w_{n+1} = w_n - \eta H(f)^{-1}\nabla f \tag{1.9}$$

He considerado un learning rate de 0.01 para compararlo con el método del gradiente descendente. Se ha hecho el proceso para los distintos puntos iniciales

$$\{(-0,5, -0,5), (1, 1), (2, 1, -2, 1), (-3, 3)\}$$

En las gráficas 1.14 y 1.15 se puede ver cómo se diferencian ambos métodos conforme avanzan las iteraciones con un learning rate de 0.01. Destaca lo mal que se desenvuelve el método

de Newton-Raphson en comparación con el del gradiente descendente. De la misma forma en las gráficas 1.16 y 1.17 podemos ver exactamente lo mismo pero con un learning rate de 0.1. En este caso en la gráfica de la derecha de la figura 1.17 se puede ver como ha alcanzado un mínimo local, por la estabilidad que tiene.

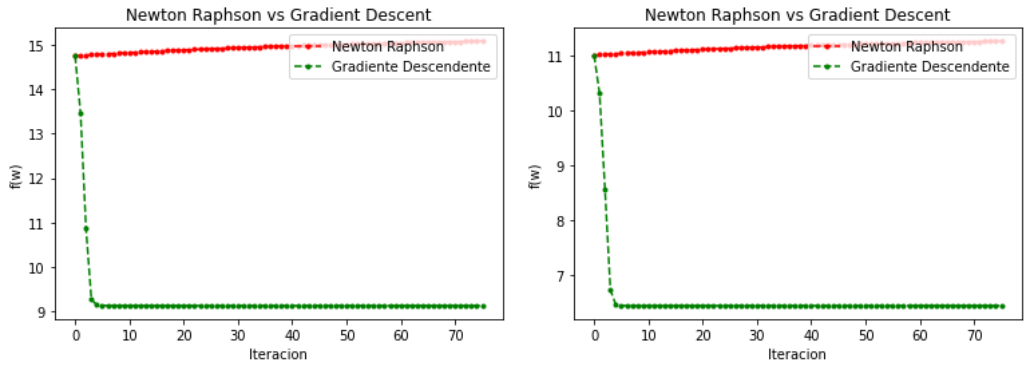


Figura 1.14: Punto inicial. Izquierda=(-0.5,0.5). Derecha=(1,1)

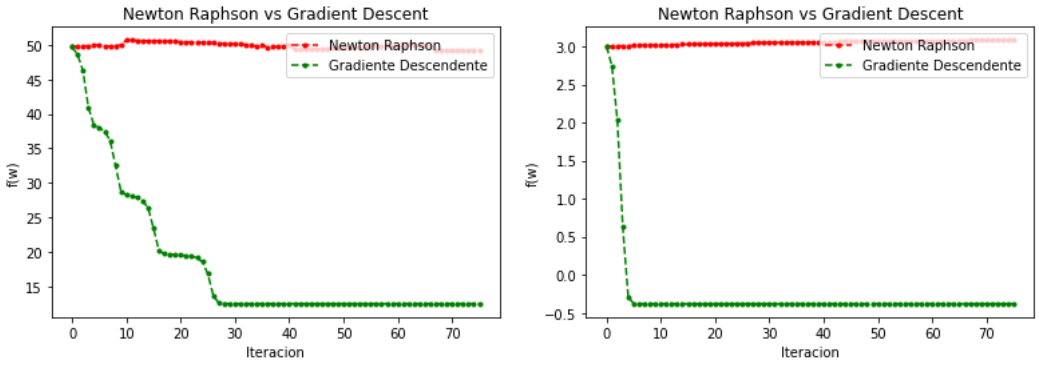


Figura 1.15: Punto inicial. Izquierda=(2.1,-2.1). Derecha=(-3,3)

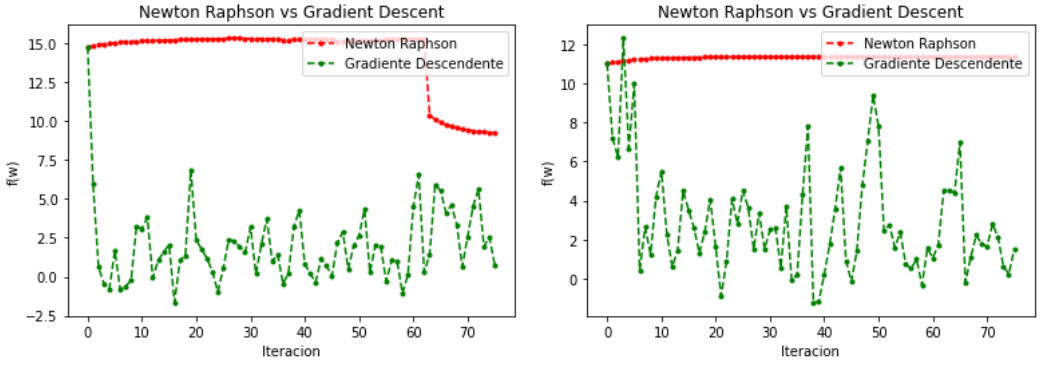


Figura 1.16: Punto inicial. Izquierda=(-0.5,0.5). Derecha=(1,1)

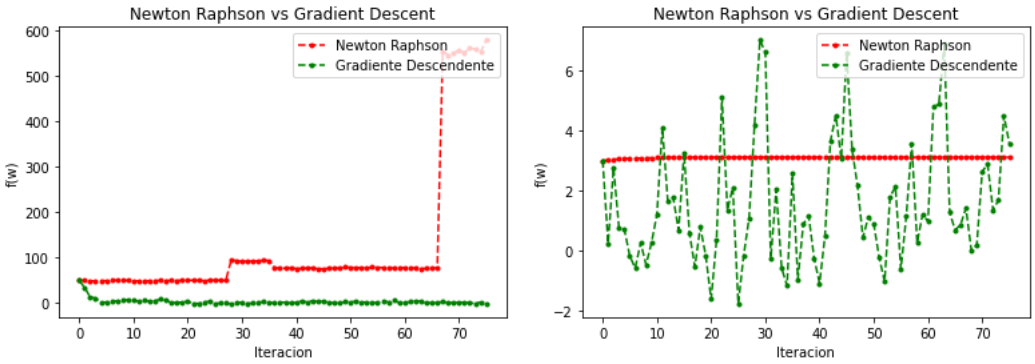


Figura 1.17: Punto inicial. Izquierda=(2.1,-2.1). Derecha=(-3,3)

Viendo estas ejecuciones surge la pregunta de por qué se desenvuelve tan mal. Se puede llegar a pensar incluso que está mal programado el método porque para nada parece que te busque mínimos. Pues la respuesta está en que este método hace uso de la información de la curvatura, y recordamos que nuestra función $f$ es un paraboloide al que se la han añadido unas perturbaciones $\phi$ de senos, esto provoca que la curvatura varíe rápidamente, provocando que en métodos como el de Newton-Raphson que hacen uso del *Hessiano* donde interviene la curvatura, arroje unos resultados tan malos. Es más, si consideramos

$$g(x, y) = (x + 2)^2 + 2(y - 2)^2 = f(x, y) - \phi(x, y)$$

Donde $g$ es el paraboloide de $f$ sin las perturbaciones $\phi$. Aplicando el método de Newton-Raphson sobre $g$ se han obtenido los resultados (con un learning rate de 0.1) se pueden ver en las gráficas comparativas 1.18 y 1.19. En estas 4 gráficas se observa como ahora sí el método de Newton-Raphson se mueve en buen sentido consiguiendo unos mínimos muy pequeños. En las gráficas 1.20, 1.21, 1.22 y 1.23 se puede ver la acción sobre el paraboloide que tiene el método de Newton-Raphson.

Como conclusión, el método de Newton-Raphson ha demostrado no desenvolverse bien con según qué tipo de funciones, siendo una alternativa increíblemente buena para algunas otras. Sin embargo, el gradiente descendente se impone ya que ha demostrado ser un método versátil con el que se asegura una cierta calidad del resultado siempre.

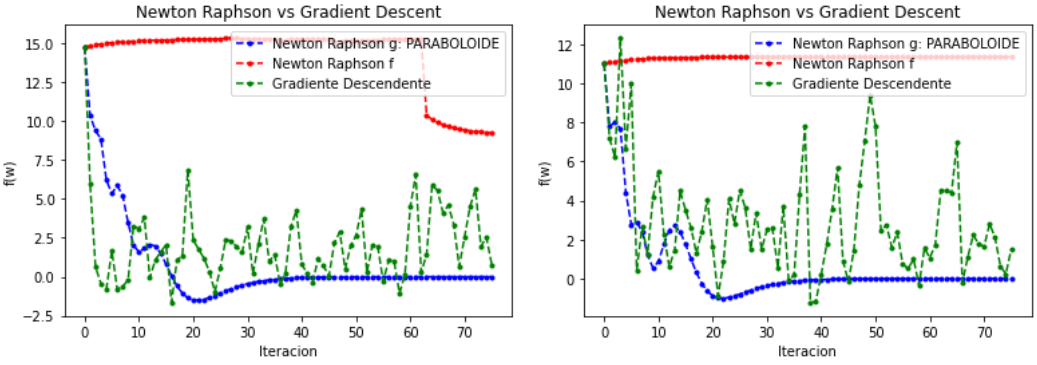


Figura 1.18: Punto inicial. Izquierda=(-0.5,0.5). Derecha=(1,1)

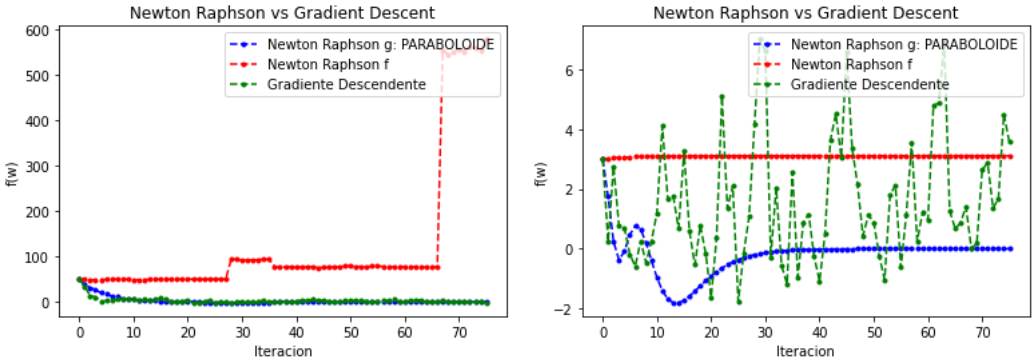


Figura 1.19: Punto inicial. Izquierda=(2.1,-2.1). Derecha=(-3,3)

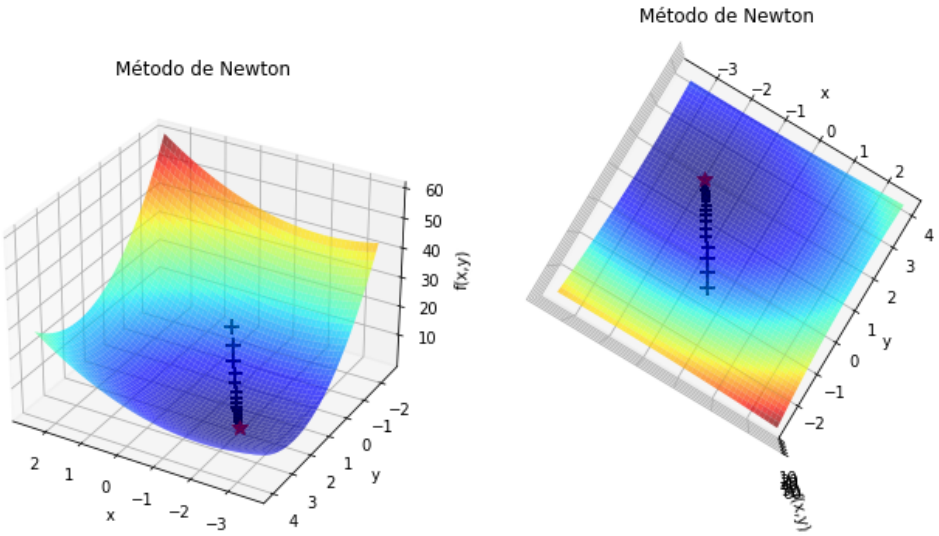


Figura 1.20: Método de Newton-Raphson sobre paraboloide $g$ con punto inicial (-0.5,0.5)

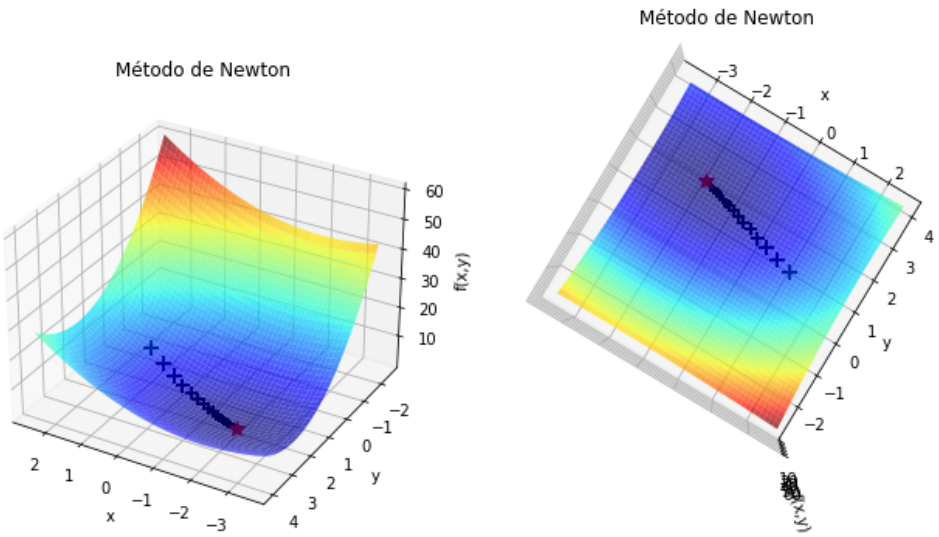


Figura 1.21: Método de Newton-Raphson sobre paraboloide $g$ con punto inicial (1,1)

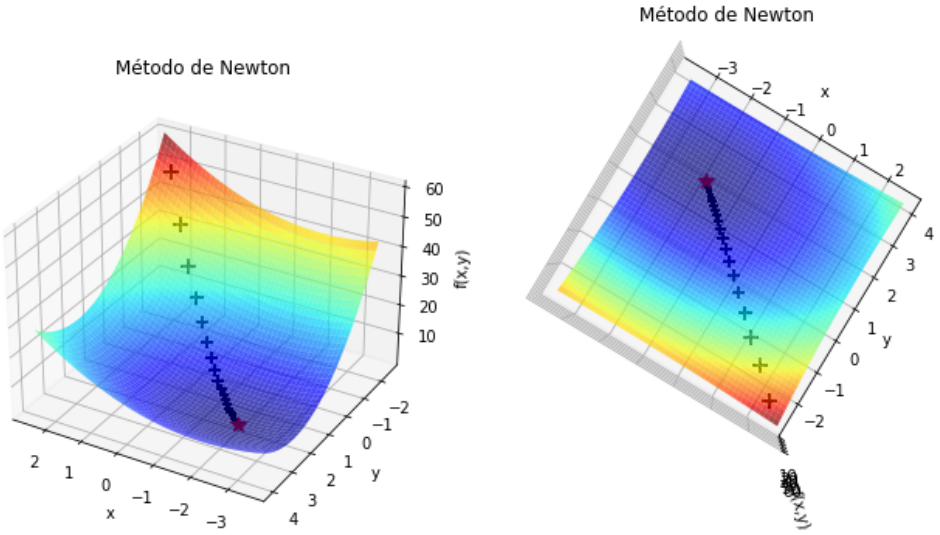


Figura 1.22: Método de Newton-Raphson sobre paraboloide $g$ con punto inicial (2.1,-2.1)

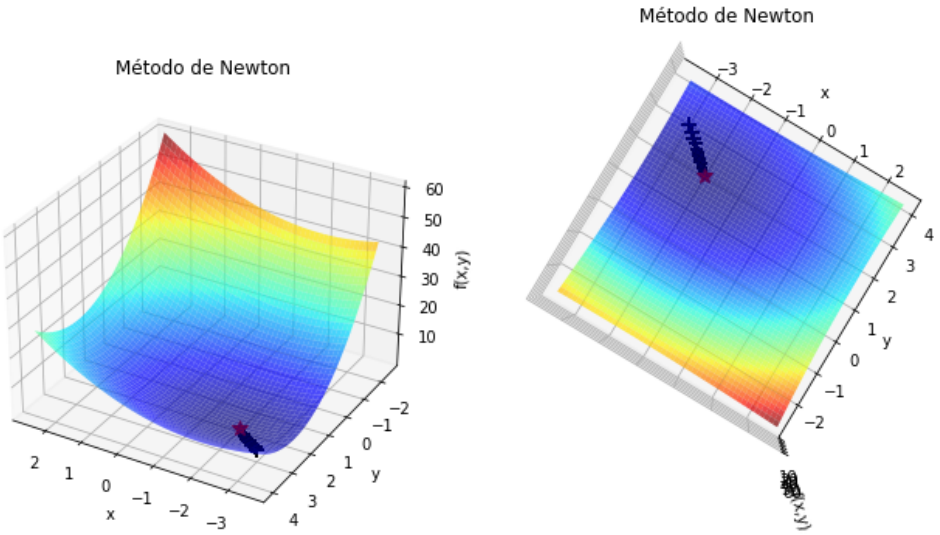


Figura 1.23: Método de Newton-Raphson sobre paraboloide $g$ con punto inicial (-3,3)