Práctica 1 | Modelos lineales

Ejercicio 1. Búsqueda iterativa de óptimos

> **Ejercicio 1.1.** Implementar el algoritmo de gradiente descendiente.

Para abordar este ejercicio, he pensado en hacer una versión suficientemente general del algoritmo de descenso de gradiente. La cabecera de la función `gradient_descent` queda, por tanto, como sigue:

```
256 def gradient_descent(w0, f, gf, lr, sc, log):
```

En primer lugar, aclararemos el nombre de algunas de las variables que aparecen en el cuerpo de la función. Teniendo en cuenta que $k \in \mathbb{N}$ es el la dimensión del dominio de f, tenemos que:

- $\mathbf{w}(i) \in \mathbb{R}^k$ es el **punto** correspondiente a la i-ésima iteración (`wi` en el cuerpo de la función). Al vector con todos los puntos para cada iteración lo denotaremos como $\mathbf{w} \equiv (\mathbf{w}(0), \mathbf{w}(1), ...)$.
- $\mathbf{g}(i) = \nabla f(\mathbf{w}(i))$ es el **gradiente** de la función a optimizar evaluado en el punto correspondiente a la i-ésima iteración (`gi` en el cuerpo de la función).

Una vez aclarado esto, explicamos a continuación cada uno de los parámetros:

- `w0` (**punto inicial**, $\mathbf{w}(0) \in \mathbb{R}^{d+1}$). Es el punto donde comenzará el algoritmo de gradiente descendiente. Puede ser fijo o calculado.
- `f` (**función a optimizar**, f), toma valores $f : \mathbb{R}^k \longrightarrow \mathbb{R}$. Normalmente, $f = E_{\text{in}}$ o $f = E_{\text{out}}$.
- `gF` (**función gradiente** de f, ∇f), toma valores $\nabla f : \mathbb{R}^k \longrightarrow \mathbb{R}$. Se calcula como

$$[\nabla f(\mathbf{v})]_i = \frac{\partial f}{\partial \mathbf{v}_i}(\mathbf{v}).$$

- `lr` (***learning rate* o tasa de aprendizaje**, $\eta(i)$). Es una función cuyos parámetros son i, $\mathbf{w}$, f y ∇f, y que devuelve un valor real: $\eta(i) \equiv \texttt{lr}(i, \mathbf{w}, f, \nabla f) \in \mathbb{R}$. Puede ser fijo o depender de dichos parámetros. Por ejemplo:
  - **Fijo.** En el caso de que el *learning rate* sea fijo, η, la función `lr` será:

$$\texttt{lr}(i, \mathbf{w}, f, \nabla f) = \eta.$$

    Es el que usaremos en esta práctica.

- **Variable.** El problema de un *learning rate* fijo es que un η demasiado pequeño es ineficiente cuando estamos lejos del mínimo (toma demasiadas iteraciones), y un η demasiado grande causa vaivenes, aumentando f en algunos casos. Lo ideal sería tomar pasos largos al estar lejos del mínimo, e ir reduciéndolos conforme nos acercamos a éste (que sea proporcional a $\mathbf{g}(i)$). Una forma posible sería:

$$\texttt{lr}(i, \mathbf{w}, f, \nabla f) = \eta \|\nabla f(\mathbf{w}(i))\|.$$

  El incluir estos parámetros permite aplicar una gran variedad de heurísticas al problema.

- `sc` (***stop condition***, **condición de parada**). Es una función que toma como parámetros, de nuevo, i, $\mathbf{w}$, f y ∇f, y que devuelve un valor booleano: $\texttt{sc}(i, \mathbf{w}, f, \nabla f) \in \{\texttt{True}, \texttt{False}\}$. Es comprobada en cada iteración. Si devuelve `True`, el algoritmo para. Algunas condiciones posibles:

  - **Contar iteraciones.** Podemos hacer que el algoritmo pare cuando ha alcanzado la T-ésima iteración:

$$\texttt{sc}(i, \mathbf{w}, f, \nabla f) = (\!| \, i \geq T \, |\!).^{1}$$

  - **Umbral de error.** El algoritmo podría parar en cuanto f llegue a un cierto umbral δ, en tal caso:

$$\texttt{sc}(i, \mathbf{w}, f, \nabla f) = (\!| f(\mathbf{w}(i)) \leq \delta \, |\!).$$

  - **Umbral en diferencia.** También podría parar cuando la diferencia entre los valores de los pesos en dos iteraciones consecutivas sea menor que un cierto umbral δ:

$$\text{sc}(i, \mathbf{w}, f, \nabla f) = (\!| \, \|\mathbf{w}(i) - \mathbf{w}(i-1)\| \leq \delta \, |\!).^{2}$$

- `log`. Función para imprimir por pantalla. Se llama una vez al final de cada iteración. Tiene como parámetros, de nuevo, i, $\mathbf{w}$, f y ∇f.

Lo interesante de definir de esta forma la función de descenso de gradiente es que podemos implementar multitud de tipos de este algoritmo, modificando el cálculo de `w0` y la implementación de `lr` y `sc`.

[1] Usamos la notación $(\!|\cdot|\!)$ como una *función de evaluación booleana*, que da `True` en caso de ser la sentencia verdadera y `False` en caso contrario. Es similar a la notación $[\![\cdot]\!]$, usada en el libro de la bibliografía *Learning from Data*, en el que la expresión booleana se evalúa como +1 o -1 en caso de que sea la sentencia verdadera o falsa, respectivamente.

[2] Nótese que al llamar a `sc` es siempre $i \leq 1$, luego tiene sentido usar $\mathbf{w}(i-1)$.

Ejercicio 1.2. Considerar la función

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

Usar gradiente descendiente para encontrar un mínimo de esta función, comenzando desde el punto $(u,v) = (1,1)$ y usando una tasa de aprendizaje $\eta = 0.1$.

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

b) ¿Cuántas iteraciones tarda el algoritmo en obtener por primera vez un valor de $E(u,v)$ inferior a 10^{14}? (Usar flotantes de 64 bits)

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

Apartado a)

Calculamos a continuación el gradiente $\nabla E(u,v)$. Por definición, $\nabla E(u,v) = \begin{pmatrix} \frac{\partial E}{\partial u}(u,v) \\ \frac{\partial E}{\partial v}(u,v) \end{pmatrix}$. Calculamos individualmente tales derivadas parciales:

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

Ya tenemos la expresión del gradiente:

$$\nabla E(u,v) = 2(u^3e^{v-2} - 2v^2e^{-u}) \begin{pmatrix} 3u^2e^{v-2} + 2v^2e^{-u} \\ u^3e^{v-2} - 4ve^{-u} \end{pmatrix}$$

Apartado b)

En vista de lo requerido, implementaremos el algoritmo de gradiente descendiente a partir del algoritmo general (ejercicio 1.1), siendo $\mathbf{w}(i) \equiv (u(i), v(i)), f \equiv E$, y teniendo en cuenta los siguientes criterios:

- $\mathbf{w}(0) = (1,1)$.
- **Tasa de aprendizaje.** Es fija, $\eta = 0.1$. Por tanto, nuestra función *tasa de aprendizaje*, `lr`, será

$$\texttt{lr}(i, \mathbf{w}, f, \nabla f) = \eta = 0.1.$$

- **Criterio de parada.** Necesitaremos al menos llegar a la iteración i donde $E(u(i), v(i)) \leq 10^{-14}$. Por tanto, la función *criterio de parada*, `sc`, será

$$\texttt{sc}(i, \mathbf{w}, f, \nabla f) = (\!| f(\mathbf{w}(i)) \leq 10^{-14} |\!).$$

De este modo, estamos en condiciones de implementar nuestro algoritmo para este apartado (y para el siguiente). Para poder obtener información en consola haremos uso de la función `log`, que en nuestro caso imprimirá i, $\mathbf{w}(i)$ y $f(\mathbf{w}(i))$, justo cuando se verifique la condición de parada (por tanto sólo se imprimirá una vez). Este *print* contiene toda la información necesaria para resolver el apartado. Basta ver el output tras ejecutar la función:

```
Pausado en iteración: 10, con E(u,v)=3.1139605842768533e-15
```

Por tanto, el algoritmo tarda 10 iteraciones en alcanzar tal valor de la función.

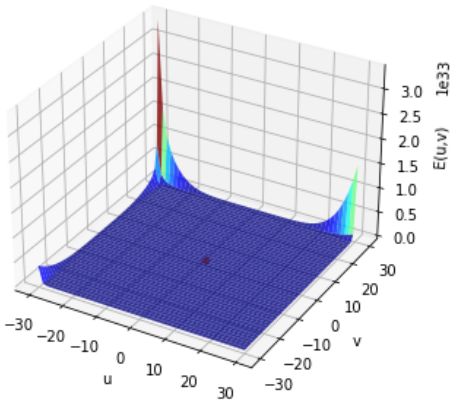
Apartado c)

Haciendo uso de `log` también podemos mostrar las coordenadas donde se alcanza tal valor:

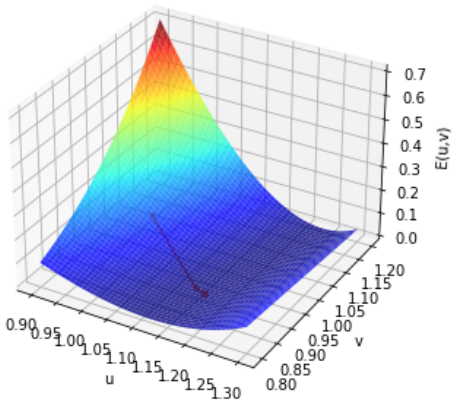
```
Coordenadas: (u,v)=(1.1572888496465497, 0.9108383657484799)
```

Es decir, se alcanzó en el punto $(u, v) = (1.1572888496465497, 0.9108383657484799)$.

Podemos ver el progreso del algoritmo en la *Gráfica 1.2.1*. Dado que el algoritmo no es visible en este rango, podemos ver los puntos con más detalle en la *Gráfica 1.2.2*. Vemos que los puntos, en efecto, van acercándose a un mínimo.



Gráfica 1.2.1



Gráfica 1.2.2

Ejercicio 1.3. Considerar ahora la función

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

a) Usar gradiente descendiente para minimizar esta función. Usar como punto inicial ($x_0 = -1, y_0 = 1$), 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 η.

b) Obtener el valor mínimo y los valores de las variables (x, y) en donde se alcanzan cuando el punto de inicio se fija: $(-0.5, 0.5), (1, 1), (2.1, -2.1), (-3, 3), (-2, 2)$. Generar una tabla con los valores obtenidos.

Apartado a)

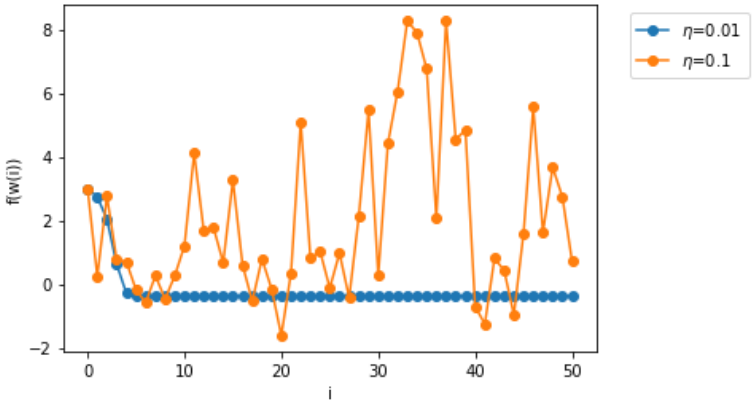
En este caso, tenemos que:

$$\nabla f(x,y) = \begin{pmatrix} 2(x+2) + 4\pi\cos(2\pi x)\sin(2\pi y) \\ 4(y-2) + 4\pi\sin(2\pi x)\cos(2\pi y) \end{pmatrix}$$

Implementaremos una función `gd_13` que nos permitirá modificar el punto inicial y el η, para de este modo usarlo en todo el ejercicio, con una tasa de aprendizaje fija y criterio de parada en función de la iteración (para parar al llegar a $i = 50$). De este modo, obtenemos los siguientes resultados en la última iteración, ambos con punto inicial ($x_0 = -1, y_0 = 1$):

η	(x, y)	$f(x, y)$
0.01	(–1.269064351751895, 1.2867208738332965)	–0.3812494974381
0.1	(–2.939376479816701, 1.607795422435394)	0.7241149424996057

El progreso de cómo evolucionan los valores de f para cada η podemos verlo en la *Gráfica 1.3.1*.



Gráfica 1.3.1

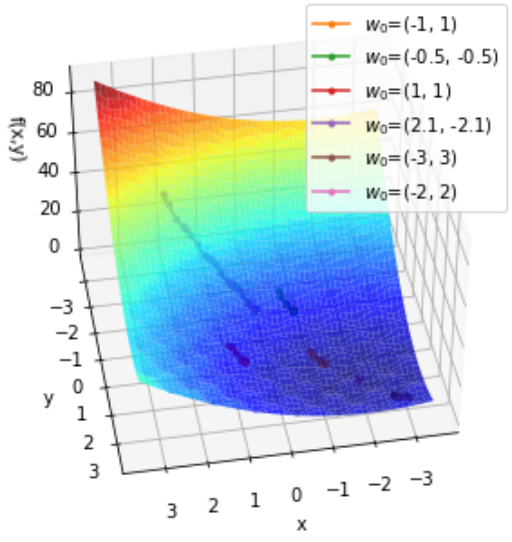
Esta prueba evidencia la importancia de la tasa de aprendizaje a la hora de converger a un mínimo. Como comentamos antes, el valor de $\eta = 0.01$, más pequeño, progresa adecuadamente hacia el mínimo, llegando a estabilizarse el algoritmo. En cambio, para $\eta = 0.1$, el punto oscila, llegando a aumentar el valor de la función.

Apartado b)

A continuación, tomaremos $\eta = 0.01$, dado que hemos visto que converge adecuadamente a los mínimos. Haciendo de nuevo uso de `gd_13`, obtenemos los siguientes resultados (en la última iteración del algoritmo):

$\mathbf{w}(0) = (x_0, y_0)$	(x, y)	$f(x, y)$
$(-0.5, -0.5)$	$(-0.7934994705090673, -0.12596575869895063)$	9.12514666
$(1, 1)$	$(0.6774387808772109, 1.290469126542778)$	6.43756960
$(2.1, -2.1)$	$(0.14880582855887767, -0.09606770499224294)$	12.49097144
$(-3, 3)$	$(-2.7309356482481055, 2.7132791261667037)$	–0.38124950
$(-2, 2)$	$(-2.0, 2.0)$	0.00000000

En vista de los resultados, vemos que la convergencia también depende del punto inicial. Vemos que el mejor resultado se obtiene en $(x_0, y_0) = (-2, 2)$, pues se trata de un mínimo. En el resto, depende de lo lejos que estemos de este punto (se van acercando a él), así como de lo "abrupta" que sea la superficie (la superficie es muy "rugosa", en tanto que tiene muchos mínimos locales). En la *Gráfica 1.3.2* podemos ver cómo actúa el algoritmo para cada punto inicial.[3]



Gráfica 1.3.2

[3]Desgraciadamente, los colores no se distinguen demasiado bien. Esta es, sin embargo, una limitación conocida de `matplotlib`.

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

Encontrar el mínimo de una función es un problema difícil, pues en muchas ocasiones no puede resolverse de forma analítica. Para resolver esto, tenemos como posible solución el algoritmo de gradiente descendiente que acabamos de abordar, que es un algoritmo iterativo.

Es claro que el algoritmo de gradiente descendiente es sumamente útil, además, es aplicable a cualquier función diferenciable y con ciertas propiedades favorables (como es la convexidad). En un algoritmo iterativo nos interesa la convergencia (en este caso, al mínimo). Sin embargo, para ello la principal dificultad estriba en poder asegurar que estas condiciones se cumplen, pues como hemos visto en los ejemplos anteriores:

- En el caso de E, el algoritmo funciona perfectamente, convergiendo de forma rápida al mínimo, dado que E es una función convexa.
- En el caso de f, una función con muchos mínimos locales, el algoritmo será exitoso sólo en caso de que escojamos parámetros adecuados.

En definitiva, dado que en muchas ocasiones nuestra función a minimizar no tendrá propiedades tan benevolentes como la de E, es esencial elegir los diversos parámetros de forma conveniente –punto inicial y tasa de aprendizaje–. Es interesante mencionar que, como comentamos anteriormente, puede implementarse una tasa de aprendizaje que varíe de forma proporcional al módulo del gradiente (que vaya acelerando cuando hay mucha pendiente y decelerando al llegar al mínimo, de esta forma evitando las fluctuaciones que evidenciamos para valores más grandes de η). Además, no olvidemos que es posible que no podamos obtener con facilidad las características de la función (monotonía, curvatura...) para comprobar si este algoritmo es deseable.

Ejercicio 2. Regresión lineal

> **Ejercicio 2.1.** Estimar un modelo de regresión lineal a partir de los datos proporcionados de dichos números (Intensidad promedio, Simetría) usando tanto el algoritmo de la pseudoinversa como gradiente descendiente estocástico (SGD). Las etiquetas serán $\{1, 1\}$, una para 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).

Para aplicar los algoritmos deberemos implementar primero la función de error, `reg_err`, así como su derivada, `d_reg_err`. Además, también implementaremos la pseudoinversa (de Moore-Penrose), podemos hacer el cálculo matricial o usar la función `numpy.linalg.pinv`.

Una vez hecho esto, podemos implementar los algoritmos especificados. En el caso de la **pseudoinversa**, es inmediato, podemos calcular el vector de pesos con un simple producto:

$$\texttt{linear\_regression}(X, y) = X^{\dagger} y$$

En el caso del **descenso de gradiente estocástico** (SGD), implementamos la función correspondiente, `stochastic_gradient_descent`, que de nuevo es general (para poder pasarle diversas tasas de aprendizaje y condiciones de parada, y así programar numerosas variaciones). La cabecera de esta función es como sigue:

```
415 def stochastic_gradient_descent(X, y, lr, sc, bs, log):
```

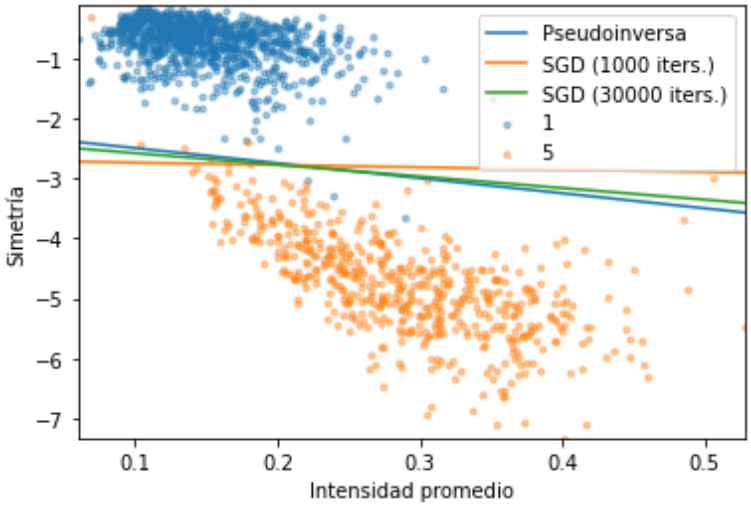
Los parámetros son similares al caso de `gradient_descent`, pero en este caso las funciones `lr` y `sc` tienen como parámetros a i, $\mathbf{w}$ y $\mathbf{e}$, siendo $\mathbf{e}$ un vector con los errores para cada iteración, a partir de $i = 1$. Además, le pasamos los arrays de datos y etiquetas para la regresión. Adicionalmente tenemos el parámetro `bs` (*batch size*), el tamaño del lote; es decir, cuántos elementos se tendrán en cuenta en cada iteración del algoritmo. Por defecto, `bs` es 32, un tamaño aceptable está en el rango $[32, 128]$.

La principal diferencia entre este algoritmo y el descenso de gradiente implementado en ejercicios anteriores es que se va teniendo en cuenta un subconjunto de la muestra en cada iteración. Los elementos a tener en cuenta son aleatorios –la distribución de los *batches* o lotes viene debidamente comentada en el código–.

En nuestro caso, implementaremos una función `sgd_iter_eta` que tendrá una tasa de aprendizaje fija, `eta`, y parará al llegar a un número máximo de iteraciones `max_iter`.

A continuación, hacemos uso de la función auxiliar `read_data` para leer los datos y etiquetas de entrenamiento y prueba proporcionados. Finalmente, usamos las funciones recién implementadas para calcular las rectas que nos dividen el espacio en las dos características. Obtenemos los siguientes errores, pudiendo ver las rectas y los datos en la *Gráfica 2.1*.

Algoritmo	Máx. iteraciones	E_{in}	E_{out}
Pseudoinversa	~	0.07918658628900395	0.1309538372005258
SGD	1000	0.08168988774965068	0.13581861169470594
SGD	30000	0.07946282656475578	0.13086538679463428



Gráfica 2.1

Podemos ver que la pseudoinversa da un E_{in} menor que SGD. Esto es algo esperable, ya que estamos minimizando el error cuadrático medio en el primer caso. Sin embargo, a pesar de esto, en muchas ocasiones es preferible utilizar SGD: por una parte, porque es más eficiente (la multiplicación de matrices es costosa, en especial conforme aumenta el set de datos); por otra, porque la aleatoriedad puede permitir la salida de mínimos locales (algo que no hace el algoritmo de gradiente descendiente convencional).

En el caso de los E_{out} vemos que también es mayor en el caso de SGD pero, lo que es más importante, son considerablemente mayores que los E_{in}. Esto también es esperable, pues el algoritmo no se ajusta a los datos de test, sino únicamente a los que ha sido entrenado.

Por otra parte, dentro de las dos ejecuciones de SGD que hemos hecho, vemos cómo, conforme aumentamos las iteraciones, el algoritmo se va aproximando al ajuste de mínimos cuadrados que proporciona la pseudoinversa, tanto en la gráfica como en los errores, que se van aproximando a dicho ajuste.

Ejercicio 2.2.1.

a) Generar una muestra de entrenamiento de $N = 1000$ puntos en el cuadrado $\mathcal{X} = [-1, 1] \times [-1, 1]$. Pintar el mapa de puntos 2D.

b) Consideremos la función

$$f(x_1, x_2) = \text{sign}((x_1 - 0.2)^2 + x_2^2 - 0.6)$$

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.

c) 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 descendiente estocástico (SGD).

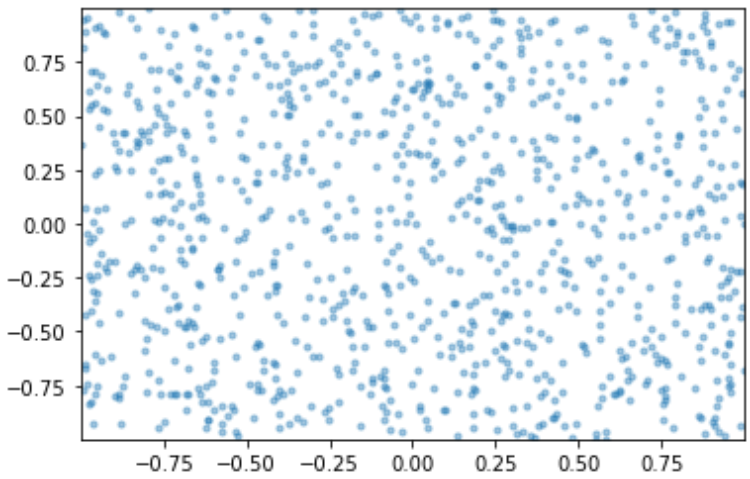
d) Ejecutar todo el experimento definido por (a)-(c) 1000 veces (generamos 1000 muestras diferentes) y

- 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.

e) Valore qué 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}.

Apartado a)

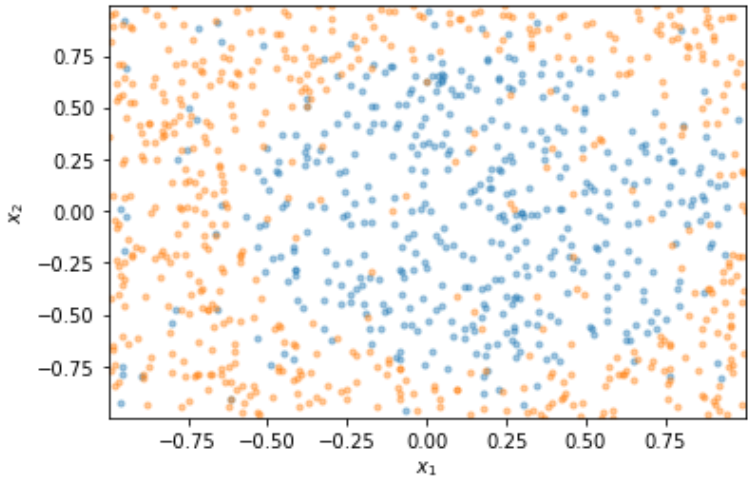
Usando la función auxiliar `simula_unif`, obtenemos el mapa que puede verse en la *Gráfica 2.2.1*.



Gráfica 2.2.1

Apartado b)

Tras seguir el enunciado, obtenemos las etiquetas que podemos ver en la *Gráfica 2.2.2*.

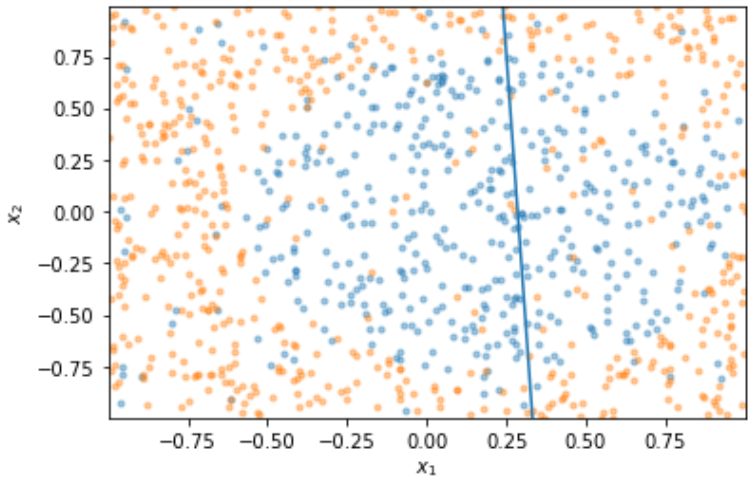


Gráfica 2.2.2

Apartado c)

Haciendo uso de SGD, obtenemos la recta de regresión visible en la *Gráfica 2.2.3*, con

$$E_{\text{in}} = 0.9378539529654695.$$



Gráfica 2.2.3

Este error es muy cercano a 1 (el máximo posible).

Apartado d)

Tras una ejecución del experimento, obtenemos los siguientes errores medios:

$$E_{\text{in}} = 0.9268380951116494$$
$$E_{\text{out}} = 0.9326801103022467$$

De nuevo, vemos cómo los errores son cercanos a 1.

Apartado e)

Dado que los errores son cercanos al máximo posible (1), tenemos que los ajustes son muy malos. Esto es evidente si tenemos en cuenta que una recta no es capaz de aproximar una circunferencia correctamente. Para ello, deberemos realizar un ajuste no lineal o una transformación (como veremos en el ejercicio siguiente).

> **Ejercicio 2.2.2.** 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}.

Simplemente ejecutaremos el algoritmo SGD haciendo uso de la transformada indicada en el enunciado. Por tanto, bastará hacer

$$\hat{w} = \texttt{sgd\_iter\_eta}(\hat{X}, y)$$

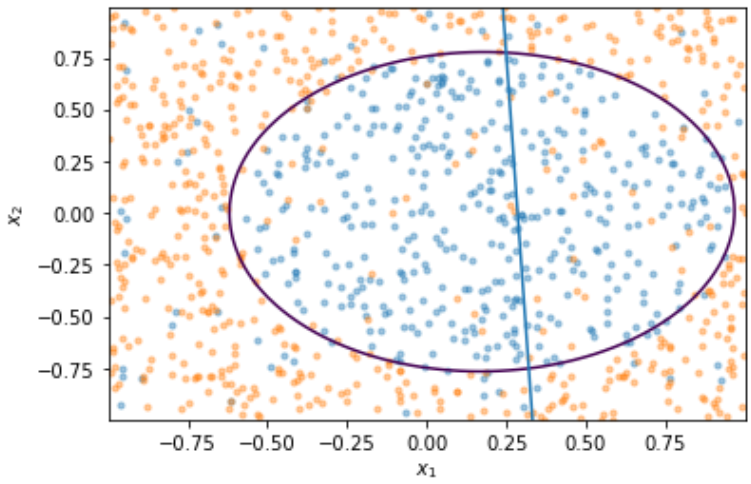
donde

$$X = \begin{pmatrix} 1 & x_{11} & x_{12} \\ \vdots & \vdots & \vdots \\ 1 & x_{N1} & x_{N2} \end{pmatrix}_{N\times 3} \Rightarrow \hat{X} = \begin{pmatrix} 1 & x_{11} & x_{12} & x_{11}x_{12} & x_{11}^2 & x_{12}^2 \\ \vdots & \vdots & \vdots & & & \\ 1 & x_{N1} & x_{N2} & x_{N1}x_{N2} & x_{N1}^2 & x_{N2}^2 \end{pmatrix}_{N\times 6}$$

luego $\hat{w}$ será un vector de dimensión 6, que nos permitirá sacar un ajuste más verosímil – basta comparar el error de la regresión lineal, E_{in}, con el nuevo error tras aplicar esta transformación, $\hat{E}_{\text{in}}$:

$$E_{\text{in}} = 0.9431763202682004$$
$$\hat{E}_{\text{in}} = 0.6053452094057804$$

Vemos una comparación de ambos ajustes en la *Gráfica 2.2.4*. De nuevo, notamos que el nuevo ajuste es más apropiado.



Gráfica 2.2.4

Ejercicio 2.2.3. A la vista de los resultados de los errores promedios E_{in} y E_{out} obtenidos en los dos experimentos, ¿qué modelo considera que es el más adecuado? Justifique la decisión.

Con un sencillo análisis del error como el que hemos realizado en el último ejercicio, tenemos razones considerables para decantarnos por el modelo no lineal. Por otra parte, es bien conocido que una recta no logrará aproximar unos datos separables mediante una ecuación cuadrática (como es nuestro caso, en el que los datos están mayoritariamente agrupados en una elipse). El modelo no lineal que hemos ajustado es cuadrático, y por tanto logra estimar nuestros datos de forma notablemente mejor que uno lineal. En definitiva, el modelo cuadrático es el más adecuado para este problema.

Bonus. Método de Newton

> **Ejercicio Bonus.** Implementar el algoritmo de minimización de Newton y aplicarlo a la función $f(x, y)$ dada en el ejercicio 1.3. Desarrolle los mismos experimentos usando los mismos puntos de inicio.
>
> - Generar un gráfico de cómo 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.

La implementación del algoritmo es muy similar a la de los algoritmos anteriores, pues se trata de un método iterativo. En cada paso, actualizamos $\mathbf{w}(i)$ de forma que $\Delta\mathbf{w} \propto -H_f^{-1}\nabla f$ como sigue:

$$\mathbf{w}(i) = \mathbf{w}(i-1) - \eta(i)\, H_f^{-1}(\mathbf{w}(i-1))\nabla f(\mathbf{w}(i-1))$$

Si $f : \mathbb{R}^k \longrightarrow \mathbb{R}$, la *matriz hessiana* H_f es una función $H_f : \mathbb{R}^k \longrightarrow \mathbb{R}^{k\times k}$, que se calcula como $[H_f(\mathbf{v})]_{i,j} = \frac{\partial^2 f}{\partial \mathbf{v}_i \partial \mathbf{v}_j}(\mathbf{v})$. Por tanto, para nuestro problema, queda:

$$H_f(x,y) = \begin{pmatrix} \frac{\partial^2 f}{\partial x^2}(x,y) & \frac{\partial^2 f}{\partial x \partial y}(x,y) \\ \frac{\partial^2 f}{\partial y \partial x}(x,y) & \frac{\partial^2 f}{\partial y^2}(x,y) \end{pmatrix} = \begin{pmatrix} 2 - 8\pi^2 \sin(2\pi x)\sin(2\pi y) & 8\pi^2 \cos(2\pi x)\cos(2\pi y) \\ 8\pi^2 \cos(2\pi x)\cos(2\pi y) & 4 - 8\pi^2 \sin(2\pi x)\sin(2\pi y) \end{pmatrix}$$

Antes de nada, comentaremos brevemente la cabecera de la función implementada, `newton`:

```
652 def newton(w0, f, gf, hf, lr, sc, log):
```

Es similar a las cabeceras de las funciones anteriores, con la salvedad de que también pasamos la recién comentada función hessiana de f, `hf`. Además, las funciones `lr`, `sc` y `log` tendrán como parámetros a i, $\mathbf{w}$, f, ∇f y H_f (añadimos también la hessiana como parámetro en caso de que queramos usarla para cualquier heurística). En nuestro caso, usaremos un η fijo y pararemos en un cierto número de iteraciones: para ello hemos implementado la función `newton_iter_eta` (que en lugar de `lr` y `sc` acepta los parámetros `eta` y `max_iter`; el η fijado y el número máximo de iteraciones, respectivamente).

Para analizar el comportamiento de este método, seguiremos la siguiente estrategia: variaremos los puntos iniciales (w_0) y las tasas de aprendizaje (η), comparando los algoritmos de Newton y GD. Para ello, ejecutaremos ambos algoritmos varias veces, en dos lotes de ejecuciones:

1) Fijando un w_0 y variando los η, una vez por cada algoritmo.

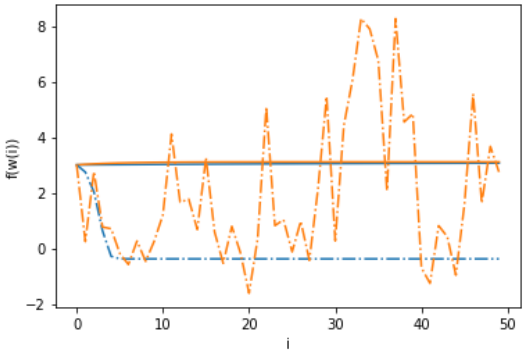
   $w_0 = (-1, 1),\ \eta \in \{0.1, 0.01\}$

2) Fijando un η y variando los w_0, una vez por cada algoritmo.

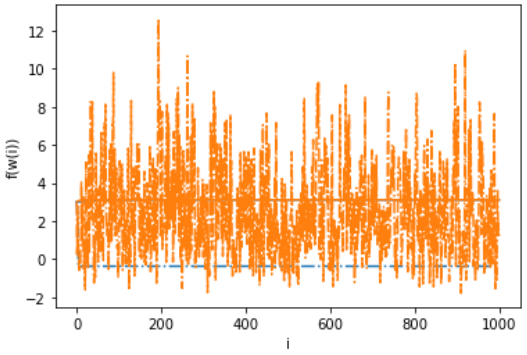
   $\eta = 0.01,\ w_0 \in \{(-1, 1), (-0.5, -0.5), (1, 1), (2.1, -2.1), (-3, 3), (-2, 2)\}$

Los resultados que obtenemos son los siguientes:

Lote 1. Fijar w_0, variar η

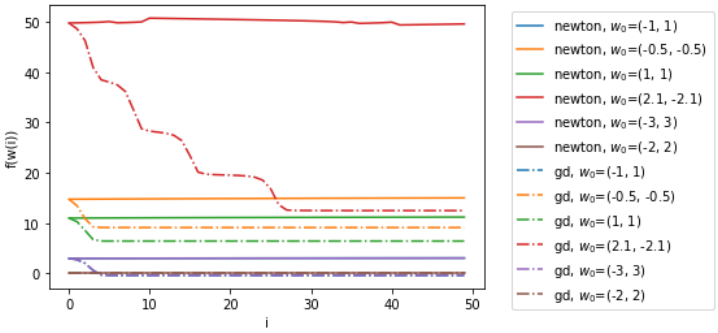


Gráfica B.1 (50 iteraciones)

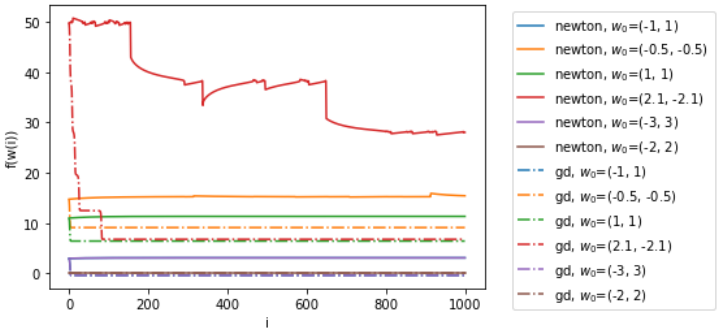


Gráfica B.3 (1000 iteraciones)

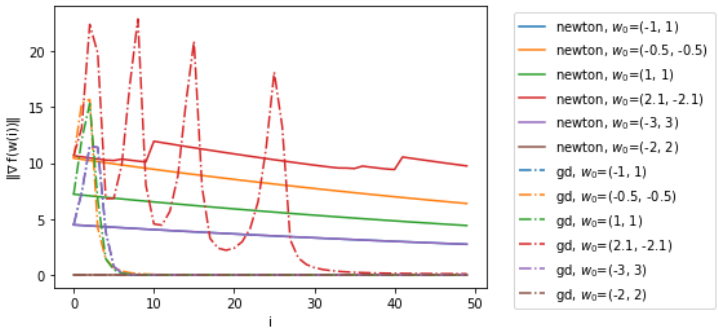
Lote 2. Fijar η, variar w_0



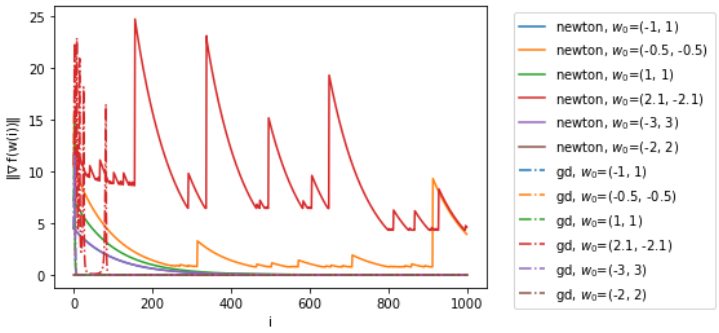
Gráfica B.2 (50 iteraciones)



Gráfica B.4 (1000 iteraciones)



Gráfica B.5 (50 iteraciones)



Gráfica B.6 (1000 iteraciones)

Vemos que este algoritmo no minimiza nuestra función, independientemente de que varíen los η o los puntos de inicio. Esto puede deberse a que estamos cayendo en puntos donde el gradiente se anula, y por tanto quedándonos en puntos de silla de f. Por otra parte, el método de Newton asegura convergencia local, por lo que si estamos lejos de un mínimo es difícil que el algoritmo llegue a él.

Un hecho interesante a notar es que este método lo que sí que hace es minimizar el gradiente de la función (aunque de forma un tanto abrupta), como podemos ver en la *Gráfica B.5* y la *Gráfica B.6*. Pero, de nuevo, que esto ocurra no garantiza que el algoritmo converja hacia un mínimo.

Evidentemente, en comparación con el gradiente descendiente, el algoritmo de Newton es mucho más inestable. Sin embargo, es adecuado en funciones con propiedades más agradables, o cuando conocemos entornos de los mínimos.