Aprendizaje Automático (2017-2018)
Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada

Memoria Práctica 1

Simón López Vico

22 de marzo de 2018

Índice

1. Ejercicio sobre la búsqueda iterativa de óptimos

Gradiente Descendente (7 puntos).

1.1. Implementar el algoritmo de gradiente descendente.

El algoritmo de gradiente descendente ha sido implementado para ser aplicado sobre una función y el gradiente de ésta. Los parámetros serán: `w, lr, f_error, f_grad, epsilon, max_iters`.

- `w`: peso de la regresión; toma como valor el punto inicial desde el que buscar el mínimo.
- `lr`: learning rate, tasa de aprendizaje.
- `f_error`: función sobre la que vamos a buscar el mínimo.
- `f_grad`: gradiente de la función `f_error`.
- `epsilon`: lo usaremos para denotar el mínimo error que queremos alcanzar; se compara con el valor absoluto de la función de error en `w`.
- `max_iters`: número máximo de iteraciones, por si la función de error no llega a un valor menor a `epsilon`.

El código del algoritmo será el siguiente:

```
def gradiente_descendente(w, lr, f_error, f_grad, epsilon, max_iters):
        i=0
        array_error = [f_error(w)]
        while True and i<max_iters:
                w = w - lr * f_grad(w)
                i+=1
                #print(f_error(w))
                array_error.insert(len(array_error),f_error(w))

                if np.abs(f_error(w)) < epsilon:
                        break

        sol=[w,i,array_error]
        return sol
```

La función devolverá el valor de `w`, el número de iteraciones y un array con los valores de la función de error (para el ejercicio 3).

1.2. Considerar la función $E(u,v) = (u^3e^{v-2} - 4v^3e^{-u})^2$. Usar gradiente descendente para encontrar un mínimo de esta función, comenzando desde el punto $(u,v) = (1,1)$ y usando una tasa de aprendizaje $\eta = 0{,}05$

Para ejecutar la función, usaremos:

```
gradiente_descendente([1,1],0.05,f_error,f_grad,10**(-5),100000)
```

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

El gradiente de la función dada será:

$$\begin{pmatrix} 2*(u^3e^{v-2} - 4v^3e^{-u})*(3u^2e^{v-2} + 4v^3e^{-u}) \\ 2*(u^3e^{v-2} - 4v^3e^{-u})*(u^3e^{v-2} - 12v^2e^{-u}) \end{pmatrix}$$

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)

Tras ejecutar el código, obtendremos que el algoritmo **tardará 38 iteraciones** en obtener un valor menor al épsilon dado.

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

En las coordenadas $(u,v) = (1{,}1195439, 0{,}65398806)$.

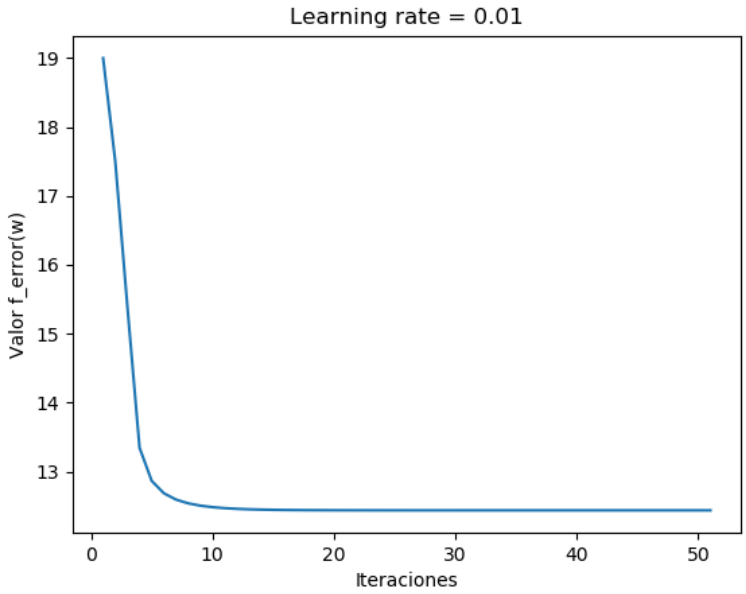
1.3. Considerar ahora la función $f(x,y) = (x-2)^2 + 2(y+2)^2 + 2sin(2\pi x)sin(2\pi y)$:

a) Usar gradiente descendente 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.

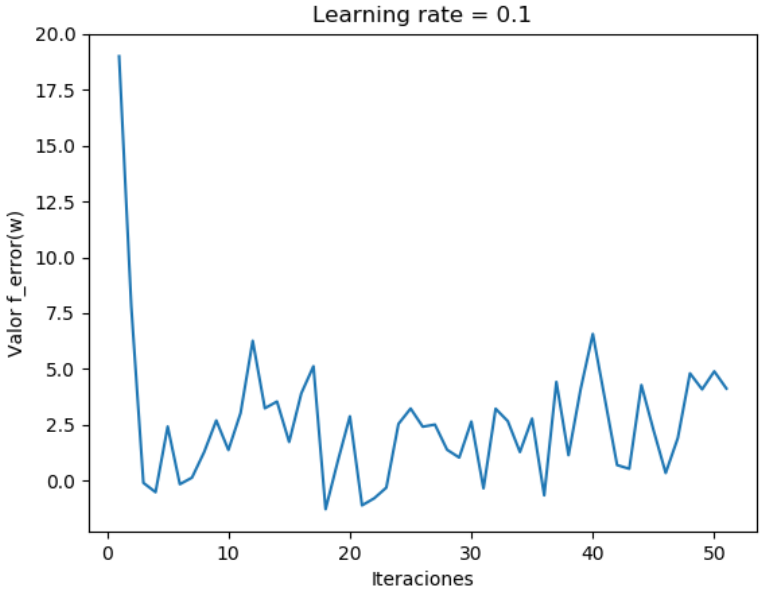
Para este ejercicio, tendremos que usar el siguiente código:

```
b=gradiente_descendente([1,1],0.01,funcion_2,grad_funcion_2,10**(-5),50)

y3a = b[2]
x3a = range(1, len(y3a)+1)
plt.plot(x3a, y3a)
plt.xlabel('Iteraciones')
plt.ylabel('Valor f_error(w)')
plt.title('Learning rate = 0.01')
plt.show()
```

Dichas líneas generarán la siguiente gráfica:



Por otra parte, haremos los mismos pasos para calcular el gradiente descendente con un *learning rate* de 0.1, obteniendo esta gráfica:



Como podemos comprobar en el caso de `lr=0.1`, la función no llega a converger nunca,

pues en las distintas iteraciones del algoritmo irá tomando valores positivos y negativos, por exceso y por defecto, sin llegar a un valor menor al del épsilon requerido en la llamada de la función.

Por otra parte, podemos ver que con una tasa de aprendizaje de 0.01, la función converge y alcanza un valor de error menor al especificado.

Por tanto, la elección de la tasa de aprendizaje es importante en el estudio de los mínimos para que así nuestra función converja.

b) Obtener el valor mínimo y los valores de las variables (x, y) en donde se alcanzan cuando el punto de inicio se fija: (2.1, -2.1), (3, -3), (1.5, 1.5), (1,-1). Generar una tabla con los valores obtenidos

Obtendremos los valores mediante el siguiente código:

```
b1=gradiente_descendente([2.1,-2.1],0.01,funcion_2,grad_funcion_2,10**(-5),50)
b2=gradiente_descendente([3,-3],0.01,funcion_2,grad_funcion_2,10**(-5),50)
b3=gradiente_descendente([1.5,1.5],0.01,funcion_2,grad_funcion_2,10**(-5),50)
b4=gradiente_descendente([1,-1],0.01,funcion_2,grad_funcion_2,10**(-5),50)
```

Y los resultados serán los siguientes:

Punto inicial	Valor mínimo
(2.1, -2.1)	(2.24380497, -2.23792582)
(3, -3)	(2.73093565, -2.71327913)
(1.5, 1.5)	(1.77792447, 1.03205687)
(1, -1)	(1.26906435, -1.28672087)

Tabla 1.1: Comparación de valores mínimos dependiendo del punto de inicio.

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

Para empezar, la tasa de aprendizaje (*learning rate*) escogida para la resolución del gradiente determinaría la convergencia o no convergencia de la función a su mínimo, por lo que habría que estudiar detenidamente el valor de esta tasa.

Por otra parte, si escogemos un valor cercano a un mínimo local, el algoritmo del gradiente descendente convergerá a éste sin opción a encontrar el mínimo global, como podemos ver en el ejercicio anterior, en el que dependiendo del w inicial que escojamos, el algoritmo convergerá a un mínimo u otro.

Finalmente, si la función es "muy plana", la convergencia al mínimo será muy lenta, teniendo que realizar un gran número de iteraciones y haciendo que la ejecución sea más

duradera.

Aún así, el método del gradiente descendente es una buena opción para buscar mínimos de una función, siempre que se ajusten bien sus parámetros.

2. Ejercicio sobre Regresión Lineal

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

2.1. Estimar un modelo de regresión lineal a partir de los datos proporcionados de dichos números (Intensidad promedio, Simetria) usando tanto el algoritmo de la pseudo-inversa como Gradiente descendente 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). (usar *Regress_Lin(datos, label)* como llamada para la función (opcional)).

Para empezar, usaremos la función `selecciona_necesarios(n1,n2,tipo)`, la cual leerá de los archivos `X_train.npy` e `y_train.npy` (si la variable `tipo='train'`) o de los archivos `X_test.npy` e `y_test.npy` (si la variable `tipo='test'`). La función devolverá dos arrays `X` e `y` que tendrán los valores de `n1` y `n2` almacenados en los ficheros anteriormente mencionados, y en el array `y` un -1 para el valor `n1` y un 1 para el valor `n2`.

El código será el siguiente:

```
def selecciona_necesarios(n1,n2,tipo):
        if tipo=='train':
                todo_X=np.load('datos/X_train.npy')
                todo_y=np.load('datos/y_train.npy')
        else:
                if tipo=='test':
                        todo_X=np.load('datos/X_test.npy')
                        todo_y=np.load('datos/y_test.npy')
                else:
                        print('Tipo valido train o test.')
                        return

        X=[]
```

```
    y=[]

    i=0
    for i in range(len(todo_X)):
            if todo_y[i]==n1:
                    X.insert(len(X),[1,todo_X[i][0],todo_X[i][1]])
                    y.insert(len(y),-1)
            if todo_y[i]==n2:
                    X.insert(len(X),[1,todo_X[i][0],todo_X[i][1]])
                    y.insert(len(y),1)

    solX=np.array(X, np.float64)
    solY=np.array(y, np.float64)

    return solX, solY
```

Usaremos esta función para obtener los valores de simetría e intensidad promedio de los números 1 y 5 llamando a la función con `selecciona_necesarios(1,5,'train')`.

Tras obtener los datos, aplicaremos el algoritmo de la pseudo-inversa y gradiente descendiente estocástico.

Pseudo-inversa:
Implementaremos el algoritmo mediante una función muy simple a la cual pasaremos la matriz `X` (valores de intensidad y simetría) y el vector `y` (etiquetas de las soluciones) y devolverá $w = ((X^T X)^{-1} X^T) y$.

```
def pseudo_inversa(X,y):
        pseudoX=np.dot(np.linalg.inv(
                        np.dot(np.transpose(X),X)),np.transpose(X))
        return np.dot(pseudoX,y)
```

Gradiente descendente estocástico:
Este algoritmo usará dos funciones auxiliares:

- `set_minibatches(X,y,longitud)`, que devolverá dos vectores `_X_` e `_y_` de tamaño `longitud` que contendrán valores aleatorios contenidos en `X` e `y`, y los usaremos para calcular sobre ellos el gradiente descendente.
- `h(x_,w_)`: devuelve el producto de $w^T X$, el cual se corresponde a la función $h(x)$ que aparece en la sumatoria del algoritmo.

Códigos:

```
def set_minibatches(X,y,longitud):
```

```
        array_index=np.random.permutation(len(X))

        Xtemp=[]
        ytemp=[]

        i=0
        for i in range(longitud):
                Xtemp.insert(len(Xtemp),X[array_index[i]])
                ytemp.insert(len(ytemp),y[array_index[i]])

        _X_=np.array(Xtemp,np.float64)
        _y_=np.array(ytemp,np.float64)

        return _X_, _y_

def h(x_,w_):
        sol=np.zeros(len(x_),np.float64)
        i=0
        for i in range(len(sol)):
                sol[i]=np.dot(np.transpose(w_),x_[i])
        return sol
```

Con estas dos funciones, realizaremos el gradiente descendente estocástico, el cual aplicará el algoritmo del gradiente descendente sobre los minibatches generados con la función `set_minibatches(X,y,longitud)`.

La función tomará como parámetros `X` e `y` datos para la regresión, `lr` tasa de aprendizaje, `epsilon` valor mínimo de error a alcanzar, `longitud` tamaño de los minibatches sobre los que aplicar el gradiente y `max_iter` número máximo de iteraciones a hacer si no se llega a un valor menor a `epsilon`.

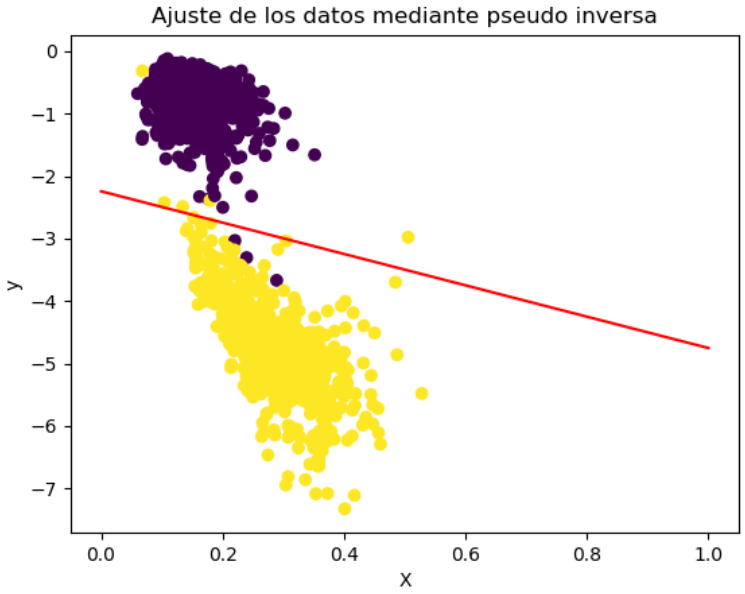
```
def gradiente_descendente_estocastico(X,y,lr,epsilon,longitud,max_iter):
        _w_ = np.zeros(len(X[1]),np.float64)
        #mejor_w=np.zeros(len(X[1]),np.float64)

        _X_, _y_=set_minibatches(X,y,longitud)

        i=0
        while True and i < max_iter:
                suma=np.zeros(len(X[1]), np.float64)

                suma += np.dot(np.transpose(_X_),( h(_X_,_w_)-_y_ ))
                _w_=_w_-lr*(2.0/longitud)*suma

                if np.abs(error_matriz(X,y,_w_)) < epsilon:
```

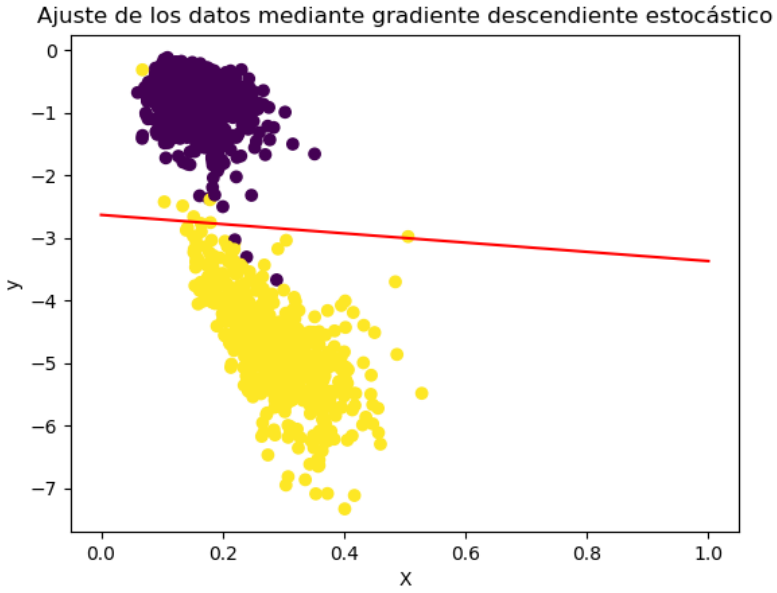
```
                    break

                _X_, _y_=set_minibatches(X,y,longitud)
                i+=1

                '''if np.abs(error_matriz(X,y,_w_)) <
                        np.abs(error_matriz(X,y,mejor_w)):
                                mejor_w=np.array(_w_,np.float64)'''

        return _w_
        #return mejor_w
```

El algoritmo del gradiente descendiente estocástico no tiene por qué converger, y por tanto suele terminar de ejecutarse cuando llega al máximo de iteraciones. Ésto puedo generar problemas, ya que el valor de w al terminar el algoritmo puede que no sea el óptimo.

Podemos ver que el código de arriba tiene unas líneas comentadas; si usamos estas líneas de código, el algoritmo guardará el valor de w para el que se obtiene el menor error, asegurando así un valor un valor óptimo de éste.

Las soluciones obtenidas para los distintos algoritmos serán las siguientes:





Finalmente, hablemos del error de cada uno de los algoritmos. Para calcular E_{in} y E_{out} usaremos las siguiente funciones:

```
def error_matriz(X,y,w): #E_in
        N=len(X)
        producto=np.dot(X,w)-y
        normalizado=np.linalg.norm(producto)
        return 1/N * normalizado * normalizado

def error_out(X,y,w):
        total=len(X)
        fallos=0
        i=0
        for i in range(total):
                producto=np.dot(X[i],w)
                if producto>0 and y[i]==-1:
                        fallos+=1
                if producto<0 and y[i]==1:
                        fallos+=1

        porcentaje_fallo=1.0*fallos/total

        return porcentaje_fallo
```

Calcularemos los errores de la pseudo-inversa y del SGD llamando a estas funciones con

los w obtenidos de cada uno y los plasmaremos en la siguiente tabla:

	Pseudoinversa	SGD
E_{in}	0.07918658628900395	0.08105255333099275
E_{out}	0.01650943396226415	0.018867924528301886

Tabla 2.1: Comparación de error entre pseudo-inversa y SGD.

Como podemos comprobar, el método de la pseudo-inversa tiene un error menor al del gradiente descendiente estocástico. Además, si usamos el algoritmo del SGD guardando el mejor valor de w (como hemos comentado antes), podremos comprobar que el error del SGD se acerca mucho al de la pseudo-inversa, pero sin llegar a rebajarlo. Por tanto, podremos concluir que para los datos proporcionados, el algoritmo de la pseudo-inversa ajusta mejor la recta de regresión que el gradiente descendente estocástico.

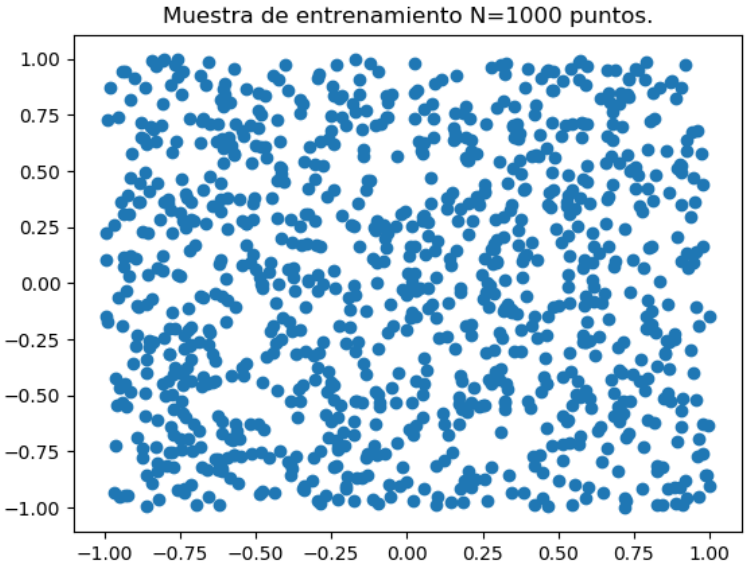
2.2. En este apartado exploramos como se transforman los errores E_{in} y E_{out} cuando aumentamos la complejidad del modelo lineal usado. Ahora hacemos uso de la función `simula_unif(N, 2, size)` que nos devuelve N coordenadas 2D de puntos uniformemente muestreados dentro del cuadrado definido por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

a) Generar una muestra de entrenamiento de N = 1000 puntos en el cuadrado $\chi = [-1, 1] \times [-1, 1]$. Pintar el mapa de puntos 2D. (ver función de ayuda)

Usaremos la función `simula_unif(N, 2, size)` que se encuentra en el archivo `funciones_utils.py` proporcionado por el profesor, y realizaremos los siguientes pasos para usarla y pintar la distribución.

```
#Generate data
X = simula_unif(N=1000, dims=2, size=(-1, 1))
#Plot data
plt.title('Muestra de entrenamiento N=1000 puntos.')
plt.scatter(X[:, 0], X[:, 1])
plt.show()
```

La gráfica obtenida será:

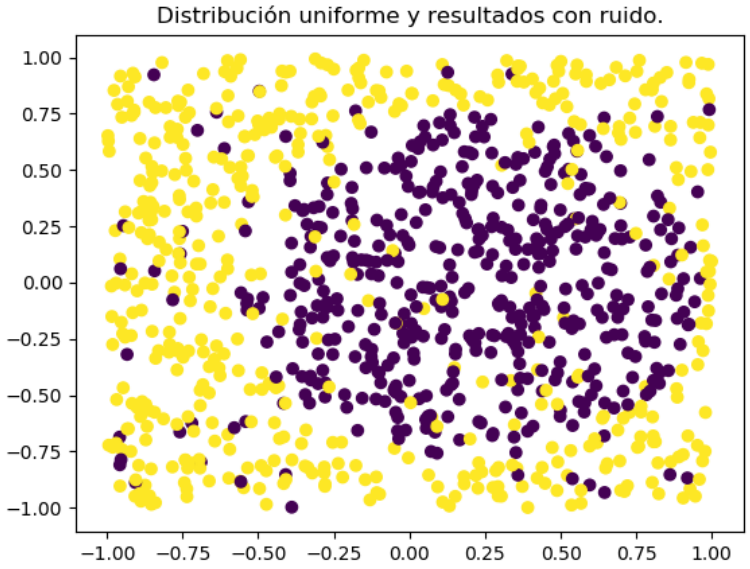


b) Consideremos la función $f(x_1, x_2) = sign((x_1 - 0,2)^2 + x_2^2 - 0,6)$ que usaremos para asignar una etiqueta a cada punto de la muestra anterior. Introducimos ruido sobre las etiquetas cambiando aleatoriamente el signo de un 10 % de las mismas. Pintar el mapa de etiquetas obtenido.

Definiremos la función `f_signo(x1,x2)` y usaremos el código `label_data(x1,x2)` proporcionado por el profesor.

```
def f_signo(x1,x2):
        return np.sign((x1-0.2)*(x1-0.2) + x2*x2-0.6)

def label_data(x1, x2):
        y = f_signo(x1,x2)
        idx = np.random.choice(range(y.shape[0]),
                size=(int(y.shape[0]*0.1)), replace=True)
        y[idx] *= -1
        return y
```

A continuación, generaremos los valores de `y` y dibujaremos la gráfica, obteniendo el siguiente resultado:

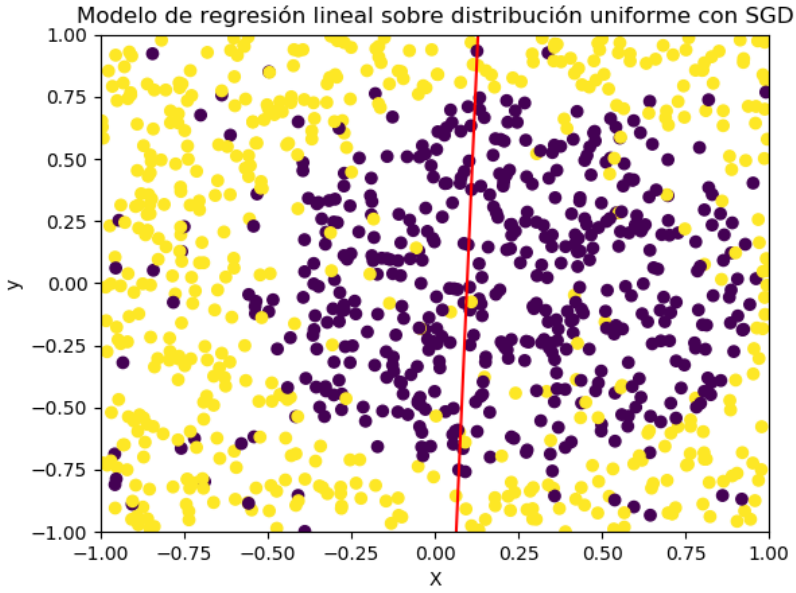


c) Usando como vector de características $(1, x_1, x_2)$ ajustar un modelo de regresión lineal al conjunto de datos generado y estimar los pesos w. Estimar el error de ajuste E_{in} usando Gradiente Descendente Estocástico (SGD).

Para realizar este apartado, primero hemos creado una función `aniade_coordenada(X)` que pondrá cada uno de los elementos de X generados con `simula_unif(N, 2, size)` de la forma $(1, x_1, x_2)$. El código será el siguiente:

```
def aniade_coordenada(X):
        Xtemp=[]

        i=0
        for i in range(len(X)):
                Xtemp.insert(len(Xtemp),[1,X[i][0],X[i][1]])

        solX=np.array(Xtemp, np.float64)

        return solX
```

Después, llamaremos a dicha función con `X=aniade_coordenada(X)`, calcularemos la recta de regresión con `w=gradiente_descendente_estocastico(X,y,0.1,0.01,64,1000)` y pintaremos la gráfica obtenida, la cuál será:



El error E_{in} obtenido para la distribución uniforme con SGD será de 0.9350981763094321, un valor de error muy alto teniendo en cuenta que las etiquetas son {-1,1}; se puede ver gráficamente que la recta de regresión generada no separa el conjunto de datos sobre el que trabajamos.

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.

Para esto, usaremos la función `media_error_uniforme()`, que hará un bucle de 1000 iteraciones generando datos X e y, ajustándolos con SGD y sumando el valores de los errores E_{in} y E_{out} en distintas variables para finalmente calcular la media del error. Para que la ejecución no se demore mucho, hemos establecido el máximo de iteraciones a realizar por cada muestra en 100.

```
def media_error_uniforme():
        i=0
        suma_ein=0
        suma_eout=0
        for i in range(1000):
                X = simula_unif(N=1000, dims=2, size=(-1, 1))
                y = label_data(X[:, 0], X[:, 1])
                X=aniade_coordenada(X)
                w=gradiente_descendente_estocastico(X,y,0.05,0.01,64,100)
                suma_ein+=error_matriz(X,y,w)
```

```
            suma_eout+=error_out(X,y,w)
            print(i)

    media_ein=suma_ein/1000.0
    media_eout=suma_eout/1000.0

    return media_ein, media_eout
```

Tras ejecutarlo, obtendremos $E_{in} = 0{,}9264763313659891$ (que no se diferencia demasiado al obtenido con una única ejecución) y $E_{out} = 0{,}3975289999999996$.

e) 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}:

Con los valores obtenidos en el apartado anterior, y a la vista distribución de los datos en la gráfica, podemos asegurar que un ajuste lineal no es adecuado para el conjunto de datos sobre el que trabajamos, pues los valores de y están distribuidos de forma esférica, por lo que ajustaría mejor los puntos una regresión circular.

Si quisiéramos ajustar los datos con una regresión lineal, deberíamos de cambiar el dominio de éstos; ya que la distribución se encuentra en $\chi = [-1, 1] \times [-1, 1]$, no sería una mala idea tomar los datos como el cuadrado de cada uno de ellos, pues los valores de y quedarían en una esquina de la gráfica, y la regresión lineal sobre ésta sería más óptima.