APRENDIZAJE AUTOMÁTICO

DOCUMENTACIÓN PRÁCTICA 2:

Ejercicio sobre la complejidad de H y el ruido Modelos Lineales

Universidad de Granada

GRUPO A3
(Viernes 17:30-19:30)

CURSO 2018-2019

Realizado por:
Jesús Medina Taboada
DNI:
jemeta@correo.ugr.es

ÍNDICE

Ejercicio 1: complejidad de H y el ruido

En este ejercicio comprobaremos la dificultad que introduce la aparición de ruido en las etiquetas a la hora de elegir la clase de funciones más adecuada.

Haremos uso de tres funciones ya programadas:

simula_unif(N, dim, rango):calcula una lista de N vectores de dimensión dim. Cada vector contiene dim números aleatorios uniformes en el intervalo rango.

```
def simula_unif(N, dim, rango):
    return np.random.uniform(rango[0],rango[1],(N,dim))
```

simula_gaus(N, dim, sigma): calcula una lista de longitud N de vectores de dimensión dim, donde cada posición del vector contiene un número aleatorio extraído de una distribución Gaussiana de media 0 y varianza dada, para cada dimensión, por la posición del vector sigma.

```
def simula_gaus(N, dim, sigma):
    media = 0
    out = np.zeros((N,dim),np.float64)
    for i in range(N):
        # Para cada columna dim se emplea un sigma determinado. Es decir, para
        # la primera columna (eje X) se usará una N(0,sqrt(sigma[0]))
        # y para la segunda (eje Y) N(0,sqrt(sigma[1]))
        out[i,:] = np.random.normal(loc=media, scale=np.sqrt(sigma), size=dim)

    return out
```

simula_recta(intervalo):simula de forma aleatoria los parámetros, v = (a, b) de una recta, y = ax + b, que corta al cuadrado [−50, 50] × [−50, 50].

```
def simula_recta(intervalo):
    points = np.random.uniform(intervalo[0], intervalo[1], size=(2, 2))
    x1 = points[0,0]
    x2 = points[1,0]
    y1 = points[0,1]
    y2 = points[1,1]
    # y = a*x + b
    a = (y2-y1)/(x2-x1) # Calculo de la pendiente.
    b = y1 - a*x1       # Calculo del termino independiente.

    return a, b
```

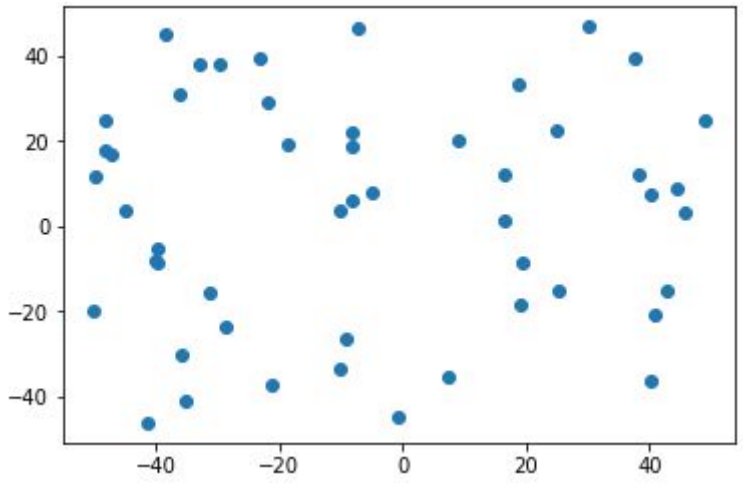
APARTADO 1

Dibujar una gráfica con la nube de puntos de salida correspondiente.

a) Considere N = 50, dim = 2, rango = [−50, +50] con simula_unif(N, dim, rango)

```
x = simula_unif(50, 2, [-50,50])
plt.scatter(x[:, 0], x[:, 1])
plt.show()
```

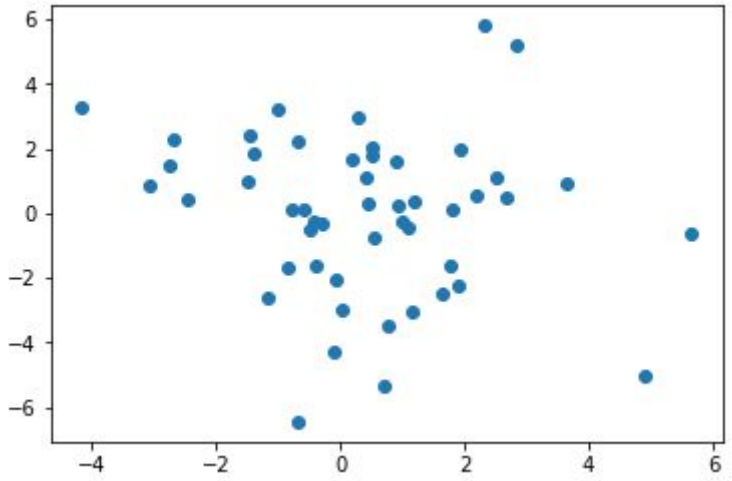
DADA UNA DISTRIBUCIÓN UNIFORME



b) Considere N = 50, dim = 2 y sigma = [5, 7] con simula_gaus(N, dim, sigma)

```
x = simula_gaus(50, 2, np.array([5,7]))
plt.scatter(x[:, 0], x[:, 1])
plt.show()
```

DADA UNA DISTRIBUCIÓN NORMAL



APARTADO 2

Con ayuda de la función simula_unif() generar una muestra de puntos 2D a los que vamos añadir una etiqueta usando el signo de la función f(x, y) = y − ax − b, es decir el signo de la distancia de cada punto a la recta simulada con simula_recta().

NOTA: la función np.sign al asignar etiqueta al 0 devuelve 0 en vez de 1 o -1 lo que nos puede ocasionar problemas más adelante, por ello usaremos la siguiente función auxiliar.

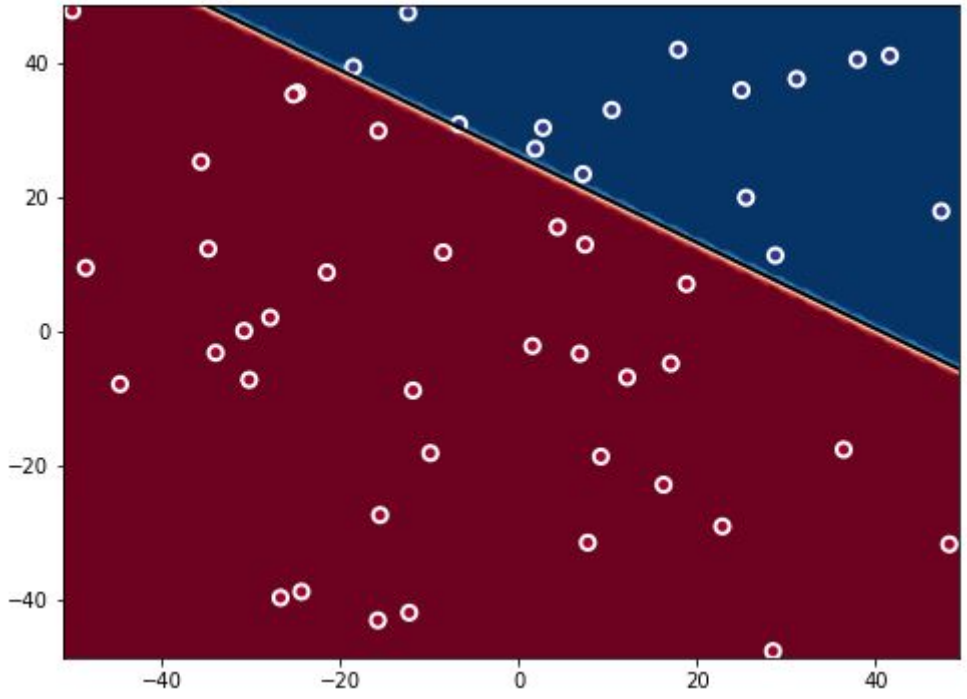
En este caso usaremos un N=50 porque es el que hemos estado usando aunque podriamos usar uno de 100, al igual que con el intervalo podriamos usar un intervalo=[-1,1].

```
# La funcion np.sign(0) da 0, lo que nos puede dar problemas
def signo(x):
    if x >= 0:
        return 1
    return -1

def f(x, y, a, b):
    return signo(y - a*x - b)

# simulamos una muestra de puntos 2D
x = simula_unif(50,2, [-50, 50])

# simulamos una recta
intervalo=[-50,50]
[a,b] = simula_recta(intervalo)

# agregamos las etiquetas usando el signo de la funcion f
y = []

for i in range(0,x.shape[0]):
    y.append(f(x[i][0], x[i][1], a, b))
```

Para los siguientes apartados realizamos una función que servirá para dibujar la gráfica representativa. Donde ‘x’ son los datos, ‘y’ son las etiquetas, ‘a’ es la pendiente de la recta y ‘b’ es el término independiente de la recta.

```python
#Funcion que dibuja una gráfica de los datos etiquetados y la recta

def plot_datos_recta(x, y, a, b, title='Point clod plot', xaxis='x axis', yaxis='y axis'):
    #Primero transformamos lo parámetros de la recta en coeficientes de w
    #siendo a la pendiente de la recta y b el término independeinte
    w = np.zeros(3, np.float64)
    w[0] = -a
    w[1] = 1.0
    w[2] = -b

    #Preparar datos
    min_xy = x.min(axis=0)
    max_xy = x.max(axis=0)
    border_xy = (max_xy-min_xy)*0.01

    #Generar grid de predicciones
    xx, yy = np.mgrid[min_xy[0]-border_xy[0]:max_xy[0]+border_xy[0]+0.001:border_xy[0],
                      min_xy[1]-border_xy[1]:max_xy[1]+border_xy[1]+0.001:border_xy[1]]
    grid = np.c_[xx.ravel(), yy.ravel(), np.ones_like(xx).ravel()]
    pred_y = grid.dot(w)
    pred_y = np.clip(pred_y, -1, 1).reshape(xx.shape)

    #Generamos puntos(plot)
    f, ax = plt.subplots(figsize=(8, 6))
    ax.contourf(xx, yy, pred_y, 50, cmap='RdBu',
                      vmin=-1, vmax=1)

    ax.scatter(x[:, 0], x[:, 1], c=y, s=50, linewidth=2,
               cmap="RdYlBu", edgecolor='white')
    ax.plot(grid[:, 0], a*grid[:, 0]+b, 'black', linewidth=2.0)
    ax.set(
       xlim=(min_xy[0]-border_xy[0], max_xy[0]+border_xy[0]),
       ylim=(min_xy[1]-border_xy[1], max_xy[1]+border_xy[1])
    )
    plt.show()
```

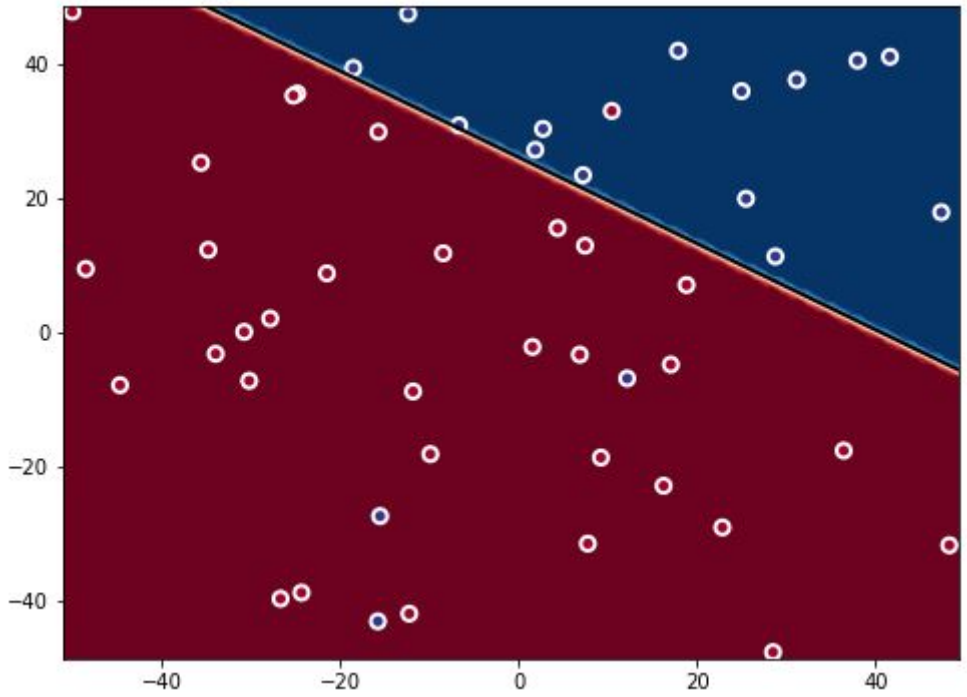
a) Dibujar una gráfica donde los puntos muestren el resultado de su etiqueta, junto con la recta usada para ello. (Observe que todos los puntos están bien clasificados respecto de la recta).



b) Modifique de forma aleatoria un 10 % etiquetas positivas y otro 10 % de negativas y guarde los puntos con sus nuevas etiquetas. Dibuje de nuevo la gráfica anterior. (Ahora hay puntos mal clasificados respecto de la recta).

```
# Introducimos ruido en las etiquetas

# Vector de etiquetas 1 y -1
i_pos = []   #1
i_neg = []   #-1

# Vector de etiquetas que vamos a modificar
i_aux = np.array(y)

for i in range(0,len(y)):
    if y[i] == 1:
        i_pos.append(i)
    else:
        i_neg.append(i)

i_pos = np.array(i_pos)
i_neg = np.array(i_neg)

# Desordenamos las posiciones de los vectores
np.random.shuffle(i_pos)
np.random.shuffle(i_neg)

# Modificamos 10% de las etiquetas positivas
for i in range(0,i_pos.size//10):
    i_aux[i_pos[i]] = -1

# Modificamos 10% de las etiquetas negativas
for i in range(0,i_neg.size//10):
    i_aux[i_neg[i]] = 1

# Dibujamos la grafica
plot_datos_recta(x, i_aux, a, b)
```

Gráfica



Como podemos observar en esta gráfica si podemos ver algunos puntos mal clasificados, causados por ese 10% de ruido que hemos metido.

APARTADO 3

Supongamos ahora que las siguientes funciones definen la frontera de clasificación de los puntos de la muestra en lugar de una recta:

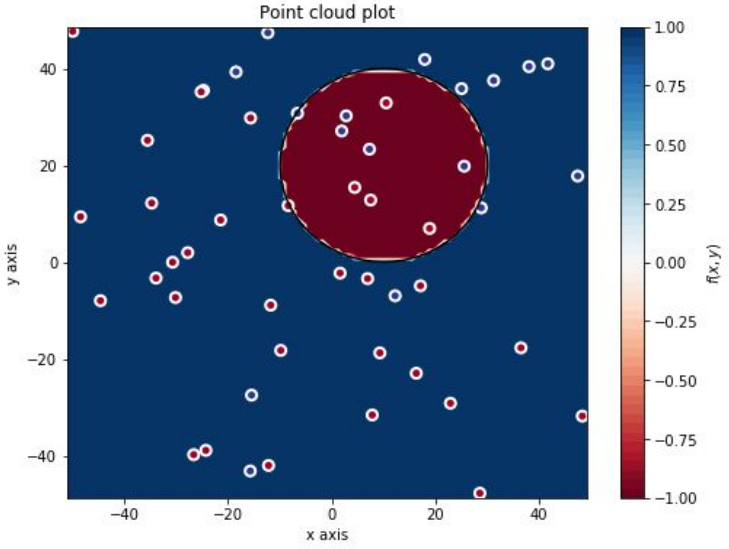
- $f(x,y) = (x-10)^2 + (y-20)^2 - 400$
- $f(x,y) = 0{,}5(x+10)^2 + (y-20)^2 - 400$
- $f(x,y) = 0{,}5(x-10)^2 - (y+20)^2 - 400$
- $f(x,y) = y - 20x^2 - 5x + 3$

Visualizar el etiquetado generado en 2b junto con cada una de las gráficas de cada una de las funciones. Comparar las formas de las regiones positivas y negativas de estas nuevas funciones con las obtenidas en el caso de la recta ¿Son estas funciones más complejas mejores clasificadores que la función lineal? ¿En qué ganan a la función lineal? Explicar el razonamiento.

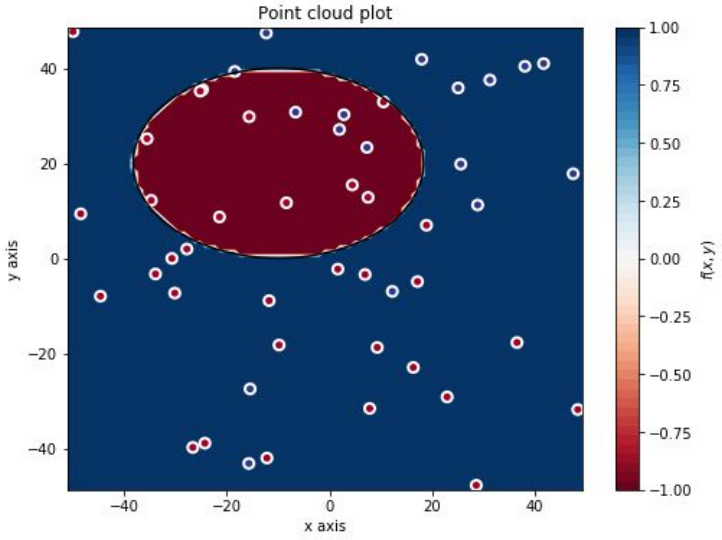
FUNCIONES

```
def f1(X):
    return (X[:,0]-10)**2+(X[:,1]-20)**2-400

def f2(X):
    return 0.5*(X[:,0]+10)**2+(X[:,1]-20)**2-400

def f3(X):
    return 0.5*(X[:,0]-10)**2-(X[:,1]+20)**2-400

def f4(X):
    return X[:,1]-20*X[:,0]**2-5*X[:,0]+3
```

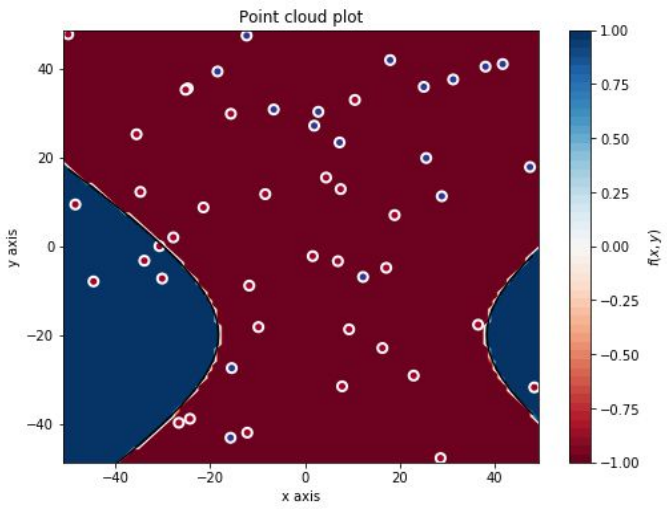
GRÁFICAS DE LAS FUNCIONES



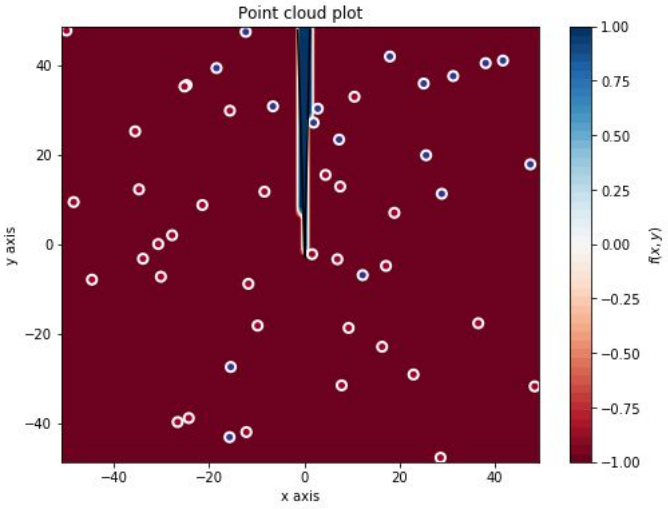
$f_1(x,y) = (x-10)^2 + (y-20)^2 - 400$



$f_2(x,y) = 0,5(x+10)^2 + (y-20)^2 - 400$



$f_3(x,y) = 0,5(x+10)^2 - (y+20)^2 - 400$



$f_4(x,y) = y - 20x^2 - 5x + 3$

ANÁLISIS

Las funciones para la elaboración de estas gráficas no son lineales a diferencia de las del primer ejercicio por lo tanto son mucho más complejas , que separan conjuntos de datos en un círculo o una elipse. La ventaja es que estas funciones son capaces de separar muchos casos que las lineales no pueden separar.

Todas tienen en común que tienen como máximo exponente 2, cuando en las lineales era 1. Si sumamos estas funciones más complejas a las que ya teníamos aumentando la clase seremos capaces de clasificar más datos.

Ejercicio 2: Modelos lineales

Algoritmo Perceptron:

Implementar la función ajusta_PLA(datos, label, max_iter, vini) que calcula el hiperplano solución a un problema de clasificación binaria usando el algoritmo PLA. La entrada datos es una matriz donde cada item con su etiqueta está representado 2 por una fila de la matriz, label el vector de etiquetas (cada etiqueta es un valor +1 o −1), max_iter es el número máximo de iteraciones permitidas y vini el valor inicial del vector. La función devuelve los coeficientes del hiperplano.

```
def ajusta_PLA(datos, label, max_iter, vini):
    w = vini
    x = np.concatenate((datos, np.ones((datos.shape[0], 1), np.float64)), axis=1)
    iters = 0
    while iters < max_iter:
        stop = True
        iters+=1
        for i in range(0,len(label)):
            if np.sign(w.dot(x[i])) != label[i]:
                w = w + label[i]*x[i]
                stop = False
        if stop:
            break
    return w, iters
```

El doble bucle en cada iteración hasta un maximo ‘max_iter’ comprueba si dado cada etiqueta coincide en signo en cuyo caso no se hace nada, pero si no coincide se actualiza el vector w. Si no se actualiza w ninguna vez para.

a) Ejecutar el algoritmo PLA con los datos simulados en los apartados 2a de la sección.1. Inicializar el algoritmo con:

a) el vector cero

b) con vectores de números aleatorios en [0, 1] (10 veces).

Anotar el número medio de iteraciones necesarias en ambos para converger. Valorar el resultado relacionando el punto de inicio con el número de iteraciones.

```
#Apartado a.1) con el vector de ceros
vini = np.zeros(x.shape[1]+1, np.float64)
w, iters = ajusta_PLA(x, y, 500, vini)
print ('Media de iteracione para converger con vector cero:', iters)
```

```
#Apartado a.2) con vectores de números aleatorios en [0, 1] (10 veces)
suma = 0
# Random initializations
for i in range(0,10):
    vini = simula_unif(1, 3,[0,1])
    w, iters = ajusta_PLA(x, y, 500, vini)
    suma = suma + iters
```

Tras ejecutarlo obtenemos los siguientes resultados para el número medio de iteraciones hasta converger:

RESULTADOS

```
Media de iteracione para converger con vector cero: 44
Valor medio de iteraciones necesario para converger:  67.7
```

En general podemos concluir que con el vector de ceros tenemos menos iteraciones aunque esto realmente podría no reflejar la verdad ya que el segundo caso al depender de la aleatoriedad necesitaríamos más pruebas para obtener datos fiables.

b) Hacer lo mismo que antes usando ahora los datos del apartado 2b de la sección.1. ¿Observa algún comportamiento diferente? En caso afirmativo diga cual y las razones para que ello ocurra.

```
# Ahora con los datos del ejercicio 1.2.b

vini = np.zeros(x.shape[1]+1, np.float64)
w, iters = ajusta_PLA(x, y_mod, 500, vini)
print (' Iteraciones con vector cero:', iters)

suma = 0

for i in range(0,10):
    vini = np.random.uniform(0, 1, x.shape[1]+1)
    w, iters = ajusta_PLA(x, y_mod, 500, vini)
    suma = suma + iters

print (' Iteraciones vector con aleatorios: ', suma/10)
```

Los datos obtenidos son:

```
Iteraciones con vector cero: 500
Iteraciones vector con aleatorios:  500.0
```

Estos resultados se deben a que los datos no son linealmente separables, por ello el algoritmo nunca terminará y siempre parará cuando llegue al número máximo de iteraciones que nuestro caso pusimos 500.

Regresión Logística:

En este ejercicio crearemos nuestra propia función objetivo f (una probabilidad en este caso) y nuestro conjunto de datos D para ver cómo funciona regresión logística. Supondremos por simplicidad que f es una probabilidad con valores 0/1 y por tanto que la etiqueta ‘y’ es una función determinista de ‘x’.

Consideremos d = 2 con probabilidad uniforme de elegir cada $x \in X$, luego elegimos una línea en el plano que pase por X como la frontera entre f(x) = 1 y f(x)=0 los cuales toma valores +1 y -1 respectivamente, para ello seleccionar dos puntos aleatorios del plano y calcular la línea que pasa por ambos. Seleccionamos 100 puntos aleatorios {xn} de X y evaluamos las respuestas {yn} de todos ellos respecto de la frontera elegida.

```
a, b = simula_recta(intervalo=(0,2))
X = simula_unif(100, 2,(0,2))
y = []

for i in range(0,x.shape[0]):
    y.append(f(x[i][0], x[i][1], a, b))
```

a)Implementar Regresión Logística(RL) con Gradiente Descendente Estocástico (SGD) bajo las siguientes condiciones:
- Inicializar el vector de pesos con valores 0.
- Parar el algoritmo cuando $|w^{(t-q)} - w^{(t)}| < 0,01$ |, donde $w^{(t)}$ denota el vector de pesos al final de la época t. Una época es un pase completo a través de los N datos.
- Aplicar una permutación aleatoria, 1, 2, ..., N , en el orden de los datos antes de usarlos en cada época del algoritmo.
- Usar una tasa de aprendizaje de η = 0,01

```
def sigmoide(x):
    return 1/(np.exp(-x)+1)

# Regresion Logistica Gradiente Descendente Estocastico
def rl_sgd(X, y, max_iters, tam_minibatch, lr = 0.01, epsilon = 0.01):
    x = np.concatenate((X, np.ones((X.shape[0], 1), np.float64)), axis=1)
    w = np.zeros(len(x[0]), np.float64)
    index = np.array(range(0,x.shape[0]))
    tam = tam_minibatch

    for k in range(0,max_iters):
        w_old = np.array(w)

        np.random.shuffle(index)
        for j in range(0,w.size):

            suma = np.sum(-y[index[0:tam:1]]*x[index[0:tam:1],j]*
            (sigmoide(-y[index[0:tam:1]]*(x[index[0:tam:1]].dot(w_old)))))

            w[j] -= lr*suma

        if np.linalg.norm(w-w_old) < epsilon:
            break

    return w

a, b = simula_recta(intervalo=(0,2))
X = simula_unif(100, 2, (0,2))
y = np.sign(f_(X,a,b))
```

b) Usar la muestra de datos etiquetada para encontrar nuestra solución g y estimar Eout usando para ello un número suficientemente grande de nuevas muestras (>999).

```
# Usar la muestra de datos etiquetada para encontrar nuestra solución g y estimar Eout
# usando para ello un número suficientemente grande de nuevas muestras (>999).
def Err(X,y,w):
    tam = X.shape[0]
    x = np.concatenate((X, np.ones((X.shape[0], 1), np.float64)), axis=1)
    return (1.0/tam)*np.sum(np.log(1+np.exp(-y[0:tam:1]*(x[0:tam:1].dot(w)))))

def reetiquetar(y):
    y_ = np.array(y)

    for i in range(0, y_.size):
        if y_[i] == -1:
            y_[i] = 0

    return y_

def error_acierto(X,y,w):
    x = np.concatenate((X, np.ones((X.shape[0], 1), np.float64)), axis=1)
    tam = y.size
    suma = 0

    for i in range(0,tam):
        if np.abs(sigmoide(x[i].dot(w))-y[i]) > 0.5:
            suma += 1

    return suma/tam
```

CONCLUSIÓN

Los resultados obtenidos han sido:

Ein= 0.01

Eout=0.02

Para calcular este error hemos hecho uso del “error de acierto” que hemos obtenido reetiquetando las etiquetas, es decir, pasando los -1 a 0 y se ha contado como fallo el que la distancia $\sigma(w^T x_i)$ era mayor de 0,5 respecto de ‘y’.

Los errores que hemos obtenido tanto en Ein como en Eout son muy bajo por lo que deducimos que hay un bajo porcentaje de fallo y que se ha realizado un buen ajuste.