Aprendizaje Automático (2018-2019)
Grado en Ingeniería Informática
Universidad de Granada

Informe práctica 2 : Programación

Montserrat Rodríguez Zamorano

22 de abril de 2019

Índice

1. Ejercicio sobre la complejidad de H y el ruido

En este ejercicio debemos aprender 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.

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

   a) Considere $N = 50, dim = 2, rango = [-50, 50]$ con $simula\_unif(N, dim, rango)$.

   Simplemente tenemos que hacer uso de la función ya programada. El código correspondiente a este apartado sería el que puede verse en (1), y su resultado, en (1.1).

```
def ejercicio11a():
    x = simula_unif(50, 2, [-50,50])
    plt.plot(x,'o',markersize=2)
    plt.title("Nube de puntos")
    plt.xlabel("eje x")
    plt.ylabel("eje y")
    plt.show()
    input("\n--- Pulsar enter para continuar ---\n")
```

Listing 1: Código simula_unif

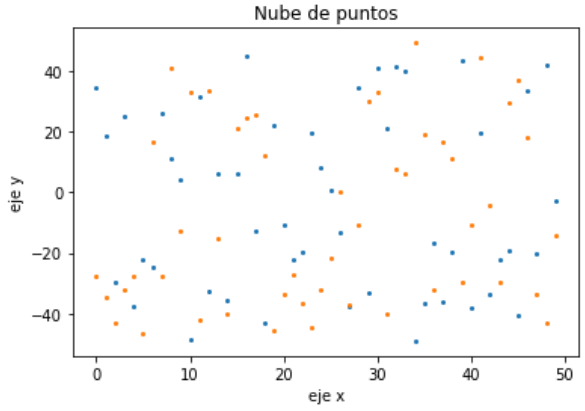

Figura 1.1: Resultado de la ejecución del código (1)

   b) Considere $N = 50, dim = 2, sigma = [5, 7]$ con $simula\_gaus(N, dim, sigma)$.

   De nuevo, tan solo tenemos que usar la función que nos dan. El código correspondiente a este apartado sería el que puede verse en (2), y su resultado, en (1.2).

```
def ejercicio11b():
    x = simula_gaus(50, 2, np.array([5,7]))
    plt.plot(x,'o',markersize=2)
    plt.title("Nube de puntos 2")
    plt.xlabel("eje x")
    plt.ylabel("eje y")
```

```
    plt.show()
    input("\n--- Pulsar enter para continuar ---\n")
```

Listing 2: Código simula_gaus

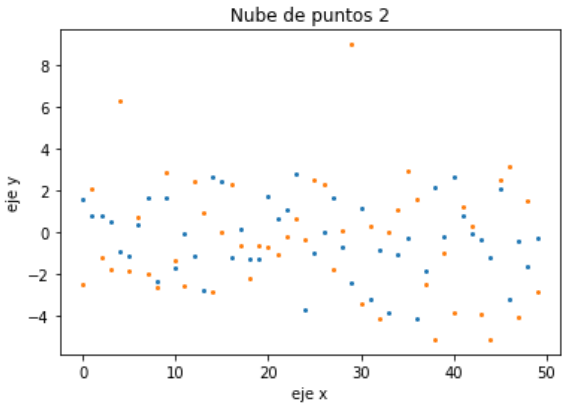

Figura 1.2: Resultado de la ejecución del código (2)

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

   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)

   Se programan en una misma función los ejercicios 2 y 3 (función *ejercicio*1_23). En este apartado sólo se revisa la parte correspondiente al ejercicio 2. El código que no pertenece a dicha función se ver en (3). Estas líneas de código se encargan de la inicialización de los datos (fuera de la función, para que se pueda acceder a ellos desde fuera de ella) y de la generación de la lista de etiquetas.

```
u = simula_unif(50, 2, [-50,50])
r = simula_recta([-50,50])
x_arr = np.array([-50,50])
y = r[0]*x_arr+r[1]
num_pos = 0
num_neg = 0

def generaEtiqueta(datos, recta):
    et = []
    np = 0
    nn = 0
    for i in range(len(datos)):
       if (f(datos[i,0],datos[i,1],recta[0],recta[1]) == 1)
          :
           np += 1
```

```
            et.append(1)
        else:
            nn += 1
            et.append(-1)
    return et, np, nn

etiq, num_pos, num_neg = generaEtiqueta(u,r)
etiq = np.array(etiq)
```

Listing 3: Código auxiliar ejercicio 2a

En (4) se pueden ver las líneas de código que se encargan de dibujar la nube de puntos junto con la recta usada para clasificar.

```
    e = np.copy(etiq)
    plt.scatter(u[:,0],u[:,1], c=e, s=3)
    plt.plot(x_arr, r[1]+r[0]*x_arr)
    #dibujamos recta
    plt.title("Clasificacion usando signo")
    plt.xlabel("eje x")
    plt.ylabel("eje y")
    plt.show()
    input("\n--- Pulsar enter para continuar ---\n")
```

Listing 4: Código ejercicio 1-2a

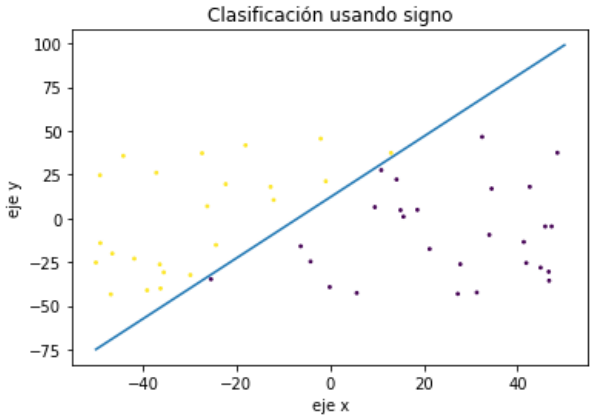

Figura 1.3: Resultado de la ejecución del código (4)

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)

Para modificar las etiquetas positivas y negativas en primer lugar se calculará el número de etiquetas que habrá que cambiar de cada tipo y luego se recorrerá el vector de etiquetas modificándolas hasta que se hayan cambiado tantas como queríamos (5). Se recorrerá el vector desde el principio y no de forma aleatoria para evitar modificar dos veces la misma etiqueta y generar así menos ruido.

```
def generarRuido(e, porc, np, nn):
    #modificar de forma aleatoria etiquetas
    num_mod_pos = int(porc*np)
    num_mod_neg = int(porc*nn)
    #print("Numero de etiquetas positivas a modificar: ",
       num_mod_pos, "\n")
    #print("Numero de etiquetas negativas a modificar: ",
       num_mod_neg, "\n")
    it_pos = 0
    it_neg = 0
    for i in range(np+nn):
        if e[i]==1:
            if it_pos<num_mod_pos:
                e[i]=-1
                it_pos += 1
                #print("Modificada etiqueta positiva\n")
        elif e[i]==-1:
            if it_neg<num_mod_neg:
                e[i]=1
                it_neg += 1
                #print("Modificada etiqueta negativa\n")
    return e
```

Listing 5: Código auxiliar ejercicio 1-2b

Como puede verse, ahora hay puntos mal clasificados con respecto a la recta (1.4). Esta es nuestra forma de simular el ruido que puede haber en los datos que encontremos en un problema real.

```
    e = generarRuido(e, 0.1, num_pos, num_neg)

    plt.scatter(u[:,0],u[:,1], c=e, s=3)
    plt.plot(x_arr, r[1]+r[0]*x_arr)
    #dibujamos recta
    plt.title("Clasificacion usando signo con ruido")
    plt.xlabel("eje x")
    plt.ylabel("eje y")
    plt.show()
    input("\n--- Pulsar enter para continuar ---\n")
```

Listing 6: Muestra de puntos con ruido

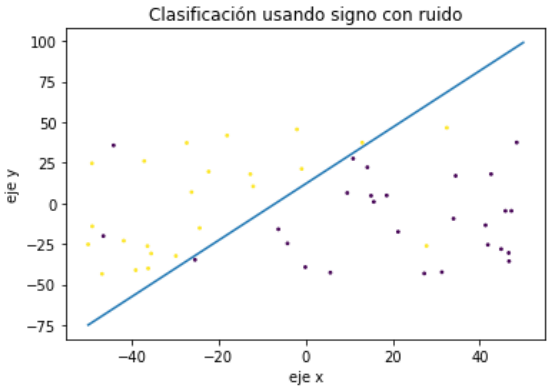

Figura 1.4: Resultado de la ejecución del código (6)

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

   Para este apartado se usará la función ya dada *plot_datos_cuad* como puede verse en el código (7). La región positiva estará en azul y la negativa en roja.

```
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
## parte del codigo dentro de la funcion ejercicio1\_23 ##
plot_datos_cuad(u,etiq,f1,'Clasificacion con ruido usando
   f1','eje x','eje y')
input("\n--- Pulsar enter para continuar ---\n")
plot_datos_cuad(u,etiq,f2,'Clasificacion con ruido usando
   f2','eje x','eje y')
input("\n--- Pulsar enter para continuar ---\n")
plot_datos_cuad(u,etiq,f3,'Clasificacion con ruido usando
   f3','eje x','eje y')
input("\n--- Pulsar enter para continuar ---\n")
plot_datos_cuad(u,etiq,f4,'Clasificacion con ruido usando
   f4','eje x','eje y')
input("\n--- Pulsar enter para continuar ---\n")
```

Listing 7: Clasificación usando funciones más complejas

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

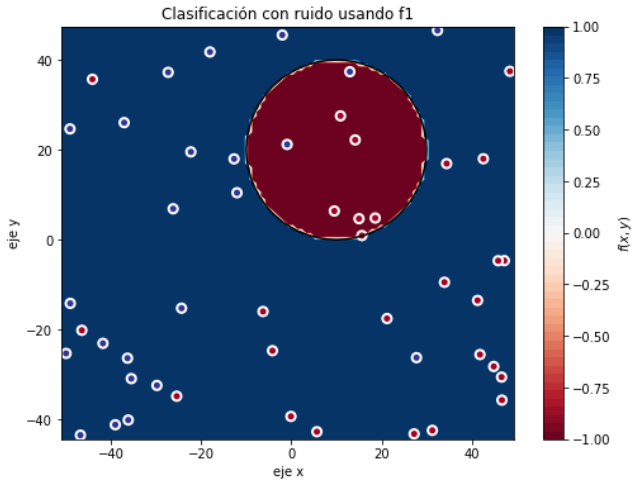

Figura 1.5: Frontera de clasificación $f(x, y) = (x - 10)^2 + (y - 20)^2 - 400$

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

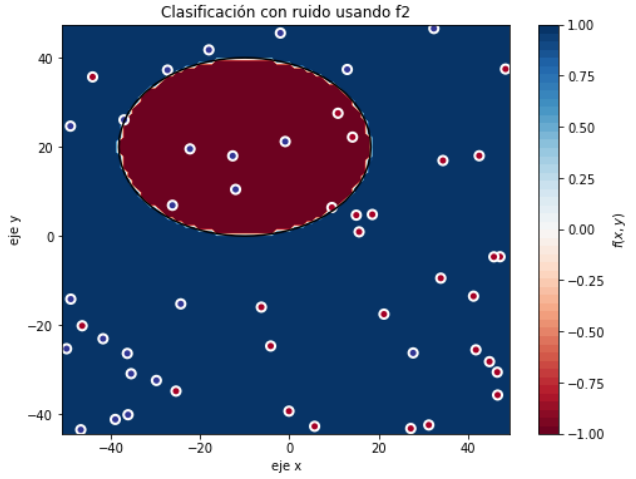

Figura 1.6: Frontera de clasificación $f(x, y) = 0{,}5(x + 10)^2 + (y - 20)^2 - 400$

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

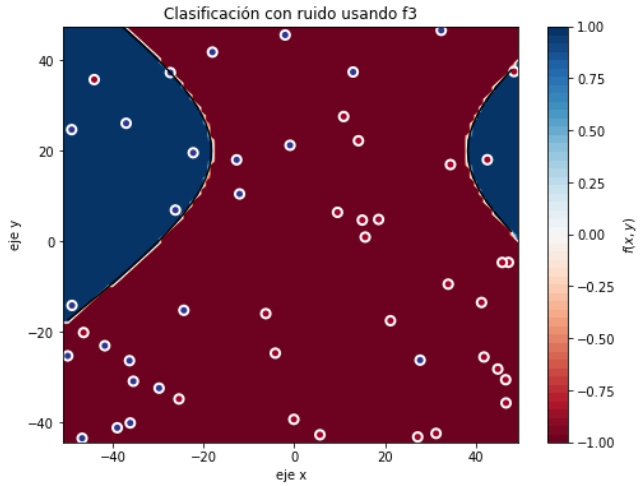

Figura 1.7: Frontera de clasificación $f(x, y) = 0{,}5(x - 10)^2 - (y + 20)^2 - 400$

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

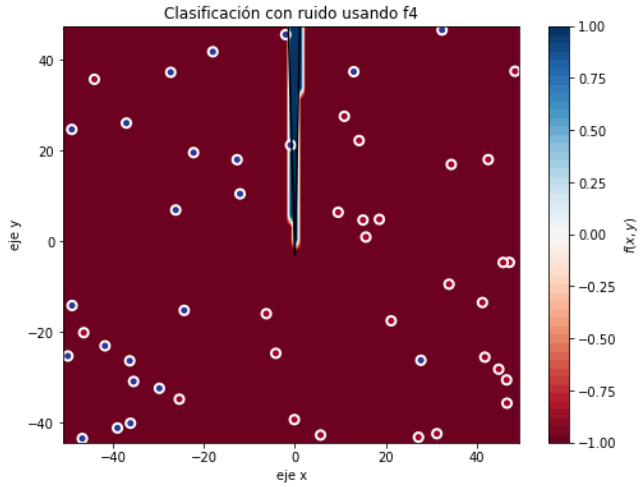

Figura 1.8: Frontera de clasificación $f(x, y) = y - 20x^2 - 5x + 3$

Como puede verse, aunque las funciones sean más complejas, la función lineal es la que mejor ha clasificado. Esto lleva a la conclusión de que una función por ser más complicada, no nos dará necesariamente mejores resultados. Deberíamos probar una funciones más simples, como las rectas, en primer lugar, ya que posiblemente nos dará un mejor resultado.

2. Modelos lineales

1. **Algoritmo Perceptrón:** 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 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.

a) Ejecutar el algoritmo PLA con los datos simulados en los apartados 2*a* de la sección 1. Inicializar el algoritmo con: a) el vector cero y, b) con los 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.

Se programa el algoritmo del perceptrón tal y como puede verse en el listing (8). Los resultados de la ejecución para datos sin ruido pueden verse en la tabla 2.1 y en la gráfica (2.1)

```
# datos: matriz donde cada item con su etiqueta esta
    representado
# por una fila de la matriz
# label: vector de etiquetas
# max_iter: numero maximo de etiquetas permitidas
# vini: valor inicial del vector

def ajusta_PLA(datos, label, max_iter, vini):
    it = 0
    error = 10**(-12)
    w_old = vini
    w_new = vini
    while it < max_iter:
        it +=1
        w_old = w_new
        for i in range(len(datos)):
            if signo(np.dot(np.transpose(w_new),datos[i]))!=
               label[i]:
                w_new = w_old + label[i]*datos[i]
                #print(w_new)
        #si converge, parar la iteracion
        if(np.linalg.norm(w_new-w_old)<error):
            break
    return w_old, it
```

Listing 8: Código algoritmo perceptrón

w_ini	w	iteraciones
[0, 0, 0]	[-2231, -428.0560087, 249.17641703]	5080
[0.35630533, 0.91945962, 0.20813803]	[-2206.64369467, -422.49812431, 245.36912102]	5054
[0.38075325, 0.02139767, 0.05984421]	[-2225.61924675, -425.38295084, 247.32343938]	5067
[0.83777432, 0.70216938, 0.09326867]	[-2249.16222568, -425.51764491, 246.47318211]	5199
[0.79134268, 0.39638455, 0.14529478]	[-2230.20865732, -427.76422967, 249.56845169]	5066
[0.98492727, 0.11899849, 0.14100426]	[-2198.01507273, -425.28232125, 247.74867013]	4992
[0.85391943, 0.72754044, 0.06912933]	[-2249.14608057, -425.49227385, 246.44904276]	5199
[0.62143003, 0.43298993, 0.88738095]	[-2203.37856997, -424.9610478, 247.59821969]	5007
[0.61800991, 0.66081048, 0.47444999]	[-2206.38199009, -422.75677345, 245.63543298]	5054
[0.00512159, 0.38433825, 0.29634551]	[-2249.99487841, -425.83547604, 246.67625894]	5199
[0.5495426, 0.48047421, 0.88940518]	[-2203.4504574, -424.91356351, 247.60024392]	5007

Tabla 2.1: Resultados ejecución algoritmo PLA para datos sin ruido

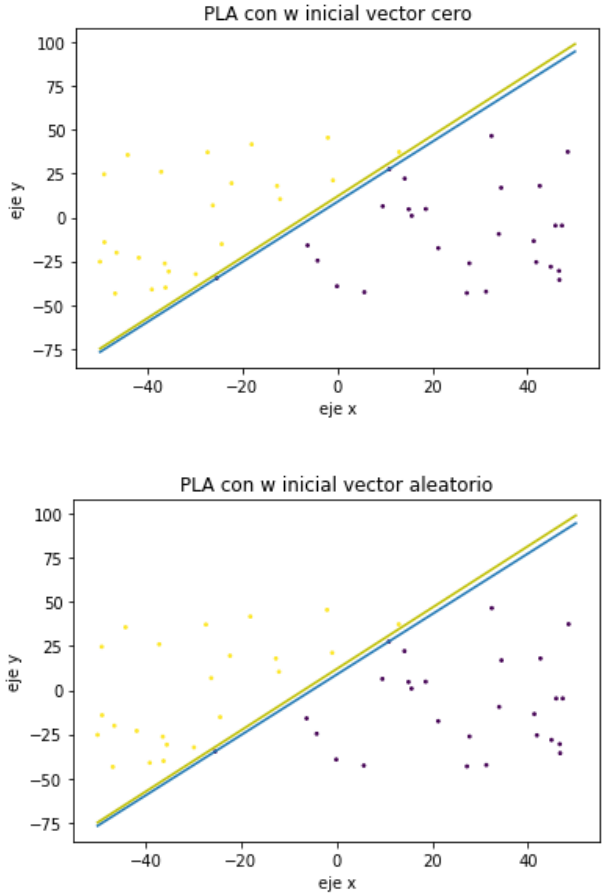

Figura 2.1: Gráfica ejecución del algoritmo perceptrón para datos sin ruido

El valor medio de las iteraciones hasta la convergencia para ejecuciones del algoritmo con vector inicial aleatorio son 5084. En el caso del vector inicial cero el número de iteraciones necesarias son 5080. Sin embargo, en todos los casos se llega a una solución parecida. Esto puede verse también con más detalle en las gráficas (puede verse la gráfica

de cada una de las iteraciones ejecutando el script completo).

Observando los resultados, podemos ver que el vector de inicio es importante a la hora de ejecutar el algoritmo, ya que determinará si la solución convergerá antes o después.

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

Hacemos uso de las mismas funciones que en el apartado anterior por lo que no se añadirá nuevo código. Se incluyen sólo los resultados de la ejecución en la tabla 2.2 y los gráficos (2.2).

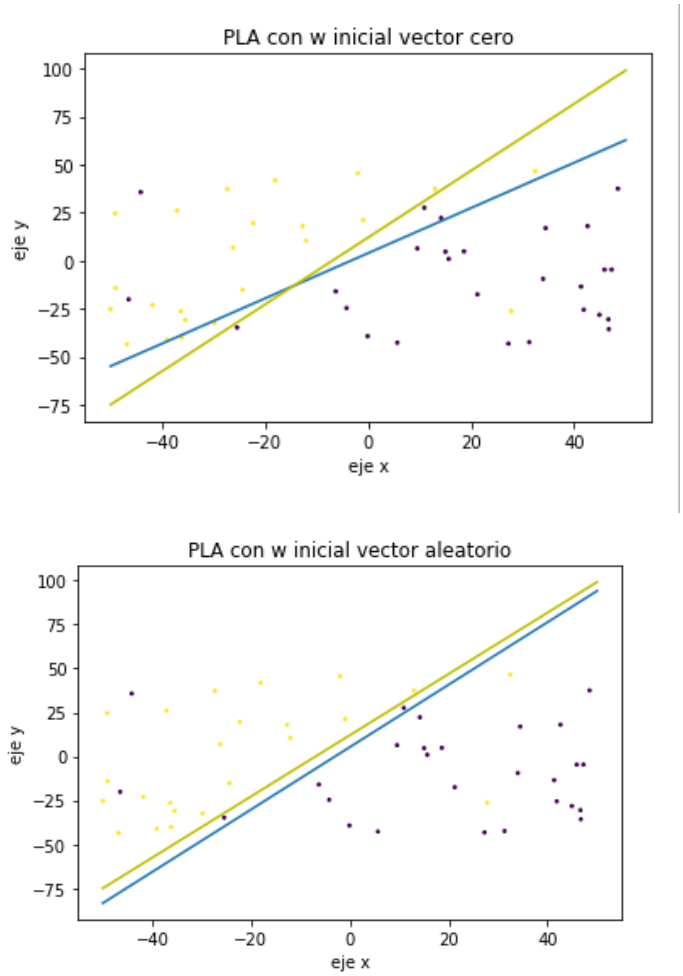

Figura 2.2: Gráfica ejecución del algoritmo perceptrón para datos con ruido

Como se puede ver, en este caso w nunca llegará a converger: las soluciones en cada caso además son muy diferentes y se agota el número máximo de iteraciones en todos los casos. Esto es porque sólo tendremos convergencia si las clases son linealmente separables, lo que no se da en este caso debido al ruido.

2. **Regresión logística:** En este ejercicio crearemos nuestra propia función objetivo f (una probabilidad en este caso) y nuestro conjunto de datos $\mathcal{D}$ para ver cómo funciona regresión

w_ini	w	iteraciones
[0, 0, 0]	[-235, -70.68549262, 60.03656901]	10000
[0.84567205, 0.22421247, 0.6889063]	[-232.15432795, -79.40860854, 44.45869203]	10000
[0.15349464, 0.20359814, 0.06759889]	[-234.84650536, -106.0979148, 12.69167004]	10000
[0.7494857, 0.1800794, 0.12426071]	[-232.2505143, -45.98063797, 12.66574369]	10000
[0.22480727, 0.27945955, 0.03382173]	[-232.77519273, -54.29167422, 34.15066119]	10000
[0.25439919, 0.66885471, 0.75991908]	[-234.74560081, -33.77080013, 21.43435711]	10000
[0.22601441 0.61184959 0.91225608]	[-232.77398559, -41.37416802, 11.46307759]	10000
[0.54161063, 0.37394509, 0.01320063]	[-230.45838937, -67.62177367, 57.17680926]	10000
[0.82931769, 0.81656153, 0.07972012]	[-238.17068231, -73.17524195, 60.33599546]	10000
[0.1716753, 0.83814903, 0.56416028]	[-232.8283247, -68.1333638, 57.47314422]	10000
[0.34978221, 0.14500296, 0.09851908]	[-224.65021779, -75.21013562, 42.38587413]	10000

Tabla 2.2: Resultados ejecución algoritmo PLA para datos con ruido

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 $\mathbf{x}$.

Consideremos $d = 2$ para que los datos sean visualizables, y sea $\mathcal{X} = [0, 2] \times [0, 2]$ con probabilidad uniforme de elegir cada $\mathbf{x} \in \mathcal{X}$. Elegir una línea en el plano que pase por $\mathcal{X}$ como la frontera entre $f(x) = 1$ (donde y toma valores $+1$) y $f(x) = 0$ (donde y toma valores -1), para ello seleccionar dos puntos aleatorios del plano y calcular la línea que pasa por ambos. Seleccionar $N = 100$ puntos aleatorios $\{x_n\}$ de $\mathcal{X}$ y evaluar las respuestas $\{y_n\}$ de todos ellos respecto de la frontera elegida.

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 $\|w^{(t-1)} - w^{(t)}\| < 0{,}01$, 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 $\eta = 0{,}01$.

En la función que puede verse en el fragmento de código (9) se implementa la regresión logística. Para ello se han tenido en cuenta las condiciones: se inicializa el vector inicial con valores 0 a la hora de ejecutar la función, se toma $error = 0{,}01$ y tasa de aprendizaje 0,01. Se aplica una permutación aleatoria en el orden de los datos antes del bucle que recorre los N datos.

No se utilizan minibatches ya que en las condiciones se habla de épocas (pases completos a través de los N datos), y por tanto no tiene sentido cogerlos en grupos. Simplemente se hace la permutación aleatoria.

```
def gradiente(y,w,x):
    temp0 = np.dot(-y,x)
    temp1 = sigmoide(np.dot(-y,np.dot(np.transpose(w),x)))
    return temp0*temp1

def sgdRL(datos, label, lr, max_iter, vini, error):
    it = 0
    w_ant = np.array(vini)
```

```
    w_new = np.array(vini)
    indices = np.arange(len(datos))

    while it < max_iter:
        it += 1
        #aplicamos una permutacion aleatoria en el orden de
            los datos
        np.random.shuffle(indices)
        datos = datos[indices]
        label = np.array(label)[indices]
        #pase de los N datos
        for i in range(len(datos)):
            gr = gradiente(label[i], w_new, datos[i])
            w_new = w_new - lr*gr
        #en caso de convergencia, detenemos la iteracion
        if np.linalg.norm(w_ant-w_new) < error:
            break
        #actualizamos
        w_ant = w_new
    return w_new, it
```

Listing 9: Código regresión logística con SGD

Se ejecuta el algoritmo y se muestran los datos en la gráfica (2.3). Nos adelantamos al siguiente apartado viendo la tasa de error para comprobar cómo de acertada ha sido la frontera calculada. Como podemos ver, es una solución muy buena.

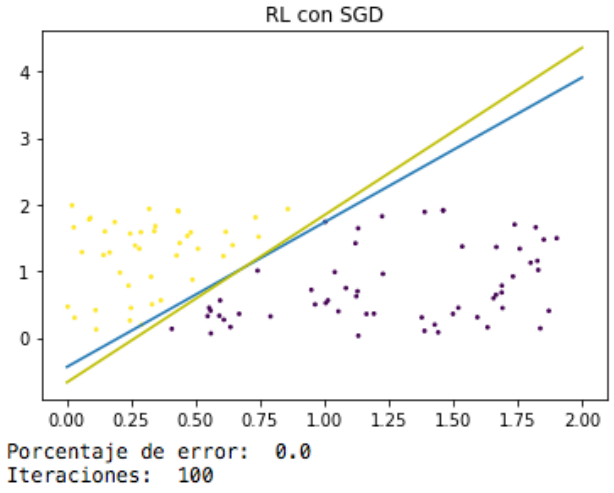

Figura 2.3: Gráfica ejecución regresión logística con SGD para $N = 100$

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

La función (10) usa la regla de Bayes para asignar las etiquetas predecidas. Cuando una de estas etiquetas no se corresponda con la etiqueta correcta, incrementamos los errores.

El número de errores obtenidos divididos entre el número total de datos y multiplicado por 100 nos dará el porcentaje de errores en nuestra solución.

```
def porcentajeError(y,w,x):
    error = 0
    h_arr = []
    for i in range(len(x)):
        #hallamos la prediccion
        h = sigmoide(np.dot(np.transpose(w),x[i]))
        if h>=0.5:
            h = 1
        else:
            h = -1
        h_arr.append(h)
        if(h != y[i]):
            error+= 1
    return (error/len(x))*100
```

Listing 10: Código regresión logística con SGD

Como puede verse en la gráfica (2.4), con $N = 1000$ se sigue teniendo un porcentaje de fallos muy bajo, prácticamente despreciable.

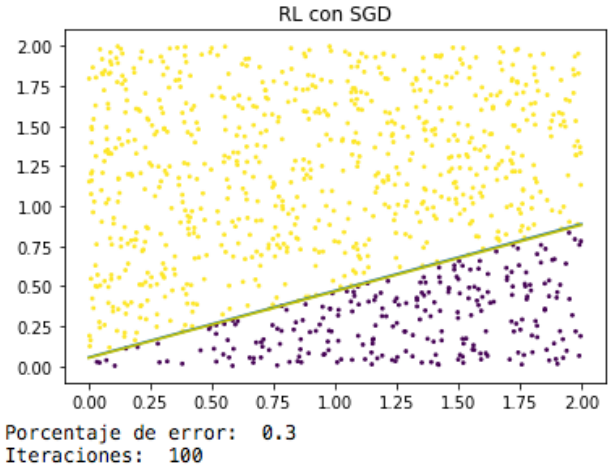

Figura 2.4: Gráfica ejecución regresión logística con SGD para $N = 1000$

Por curiosidad, observamos qué pasa si permitimos que el algoritmo itere mucho más, aumentando el número de iteraciones. Las soluciones obtenidas nos da un 0 % de fallos en ambos casos. Aunque el primer caso ($N = 100$) ya nos daba este porcentaje, podemos ver que itera mucho más hasta converger.

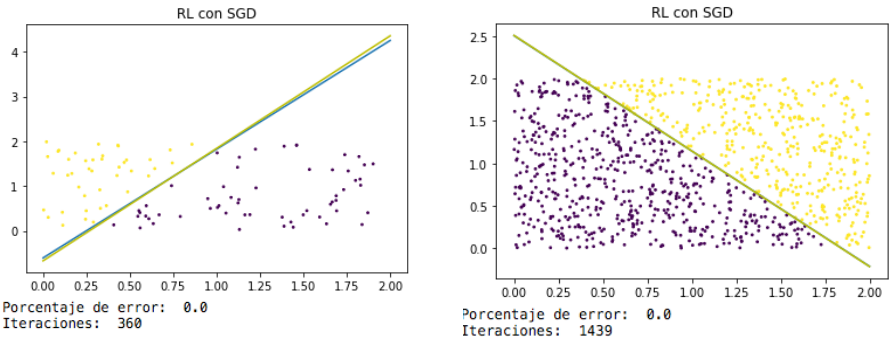

Figura 2.5: Gráfica ejecución regresión logística con SGD