UNIVERSIDAD DE GRANADA

PRÁCTICA 2: Complejidad de H y Modelos Lineales

Aprendizaje Automático || Curso 2018-2019

Alumno: José María Sánchez Guerrero
DNI: 76067801Q
Correo: jose26398@correo.ugr.es
Grupo: A3 – Viernes 17:30

CONTENIDO

EJERCICIO SOBRE LA COMPLEJIDAD DE H Y EL RUIDO

Ejercicio 1

El objetivo de este primer ejercicio es ver la diferencia o, mejor dicho, las dificultades que se nos presentan a la hora de sacar una función que nos permita clasificar las etiquetas de una muestra, con ruido y sin ruido, correctamente.

Para ello, vamos a utilizar tres funciones proporcionadas por los profesores que son las siguientes:

- **simula_unif (N, dim, rango).** Calcula una lista de *N* números aleatorios uniformes en el intervalo *rango* de dimensión *dim*.
- **simula_gaus (N, dim, sigma)**. Calcula una lista de *N* números aleatorios distribuidos por una Gaussiana con media 0 y varianza determinada por el *sigma* (también de dimensión *dim*).
- **simula_recta (intervalo)**. Simula de forma aleatoria los parámetros *a* y *b* de una recta *y = ax + b*, que corta al cuadrado [-50, 50] x [-50, 50].

Apartado a). Dibujar nube de puntos generados aleatoriamente de manera uniforme utilizando la función anterior con los siguientes parámetros:

simula_unif (N=50, dim=2, rango=[-50,50])

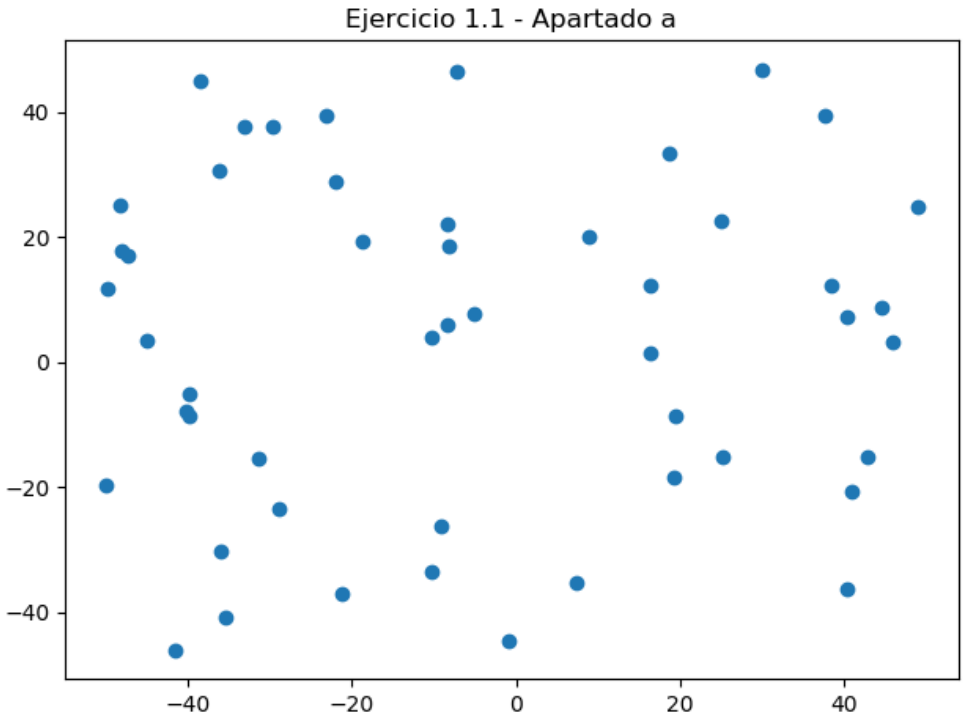

Ilustración 1. Nube de puntos generados aleatoriamente de manera uniforme

Apartado a). Dibujar nube de puntos generados aleatoriamente, y distribuidos por una Gaussiana, utilizando la función anterior con los siguientes parámetros:

simula_gauss (N=50, dim=2, sigma=[5,7])

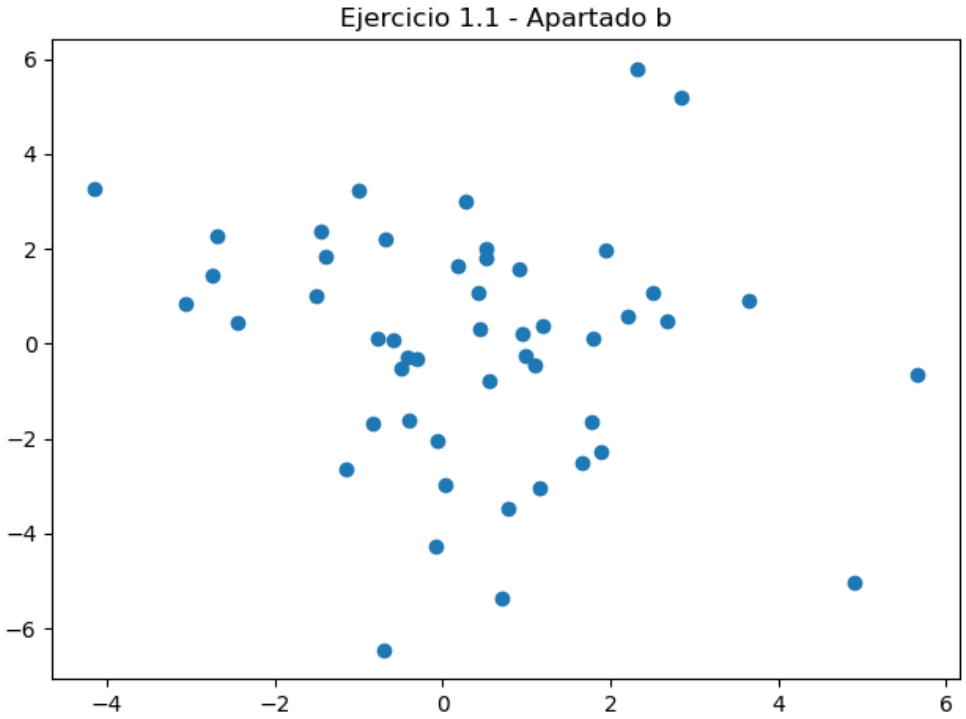

Ilustración 2. Nube de puntos generados aleatoriamente y distribuidos por una Gaussiana

Ejercicio 2

Para este ejercicio, vamos a generar una muestra de la misma forma que hemos hecho en el ejercicio 1 apartado a. A cada punto de la muestra le vamos a añadir una etiqueta utilizando el signo de la función $f(x, y) = y - ax - b$.

Después generamos la línea que los divide, que es la misma generada con *simula_recta().* Esta es su implementación:

```
# Generamos otra muestra de puntos 2D
x = simula_unif(50, 2, [-50, 50])
y = []

# Asignamos etiquetas a las muestras generadas anteriormenteutilizando
# las funciones simula_recta(), f() y signo() proporcionadas por el profesor
a, b = simula_recta([-50, 50])
for i in range(x.shape[0]):
    y.append(f(x[i, 0], x[i, 1], a, b))
y = np.array(y)

# Ahora generamos la linea que divide los datos
lineaX = np.linspace(-50, 50, y.size)
# Usamos la funcion f(x,y)=y-ax-b, que es la distancia de cada
# punto hasta la recta
lineaY = a * lineaX + b
```

Ilustración 3. Implementación de la asignación de etiquetas

Apartado a). Dibujar la gráfica de la muestra con etiquetas y la línea que separa los datos.

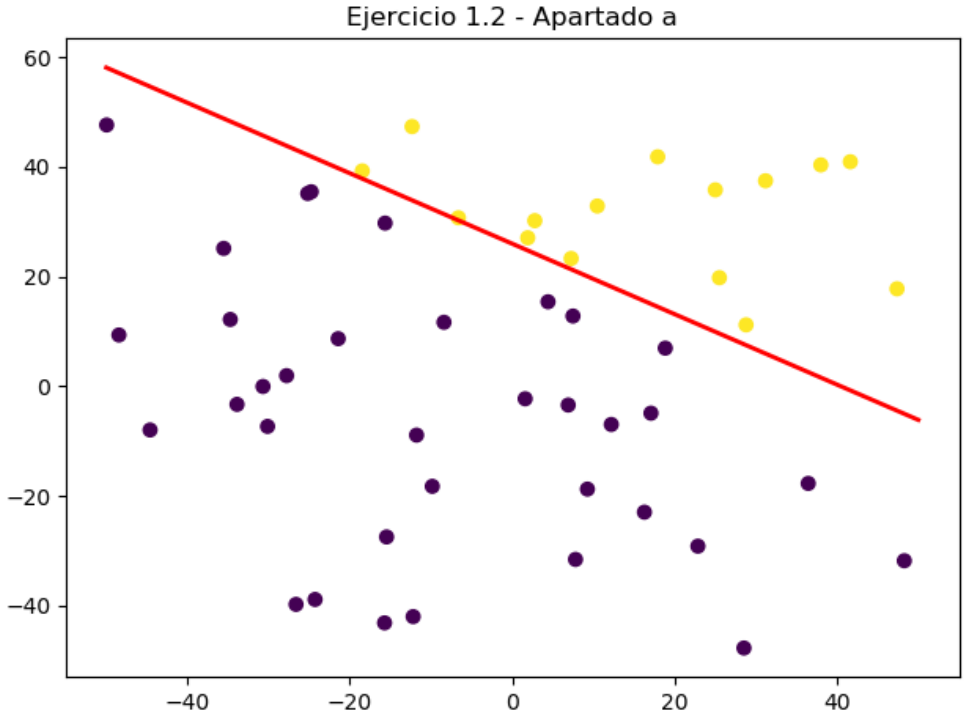

Ilustración 4. Nube de puntos etiquetados y línea que los separa

Apartado b). Ahora tenemos que añadirle ruido a la muestra de forma que el 10% de las positivas cambie a negativas y el 10% de las negativas cambie a positivas. Para ello he separado los índices de las positivas y negativas, he seleccionado el 10% de ellos y los he cambiado de array. La implementación es la siguiente:

```
# Inicializamos dos listas vacías para los indices positivos y negativos
indicesPositivos = []
indicesNegativos = []

# Le metemos los valores correspondientes del array y de etiquetas
for i in enumerate(y):
    if y[i[1]] == 1:
        indicesPositivos.append(i[0])
    else:
        indicesNegativos.append(i[0])

# Los convertimos en np.arrays
indicesPositivos = np.array(indicesPositivos)
indicesNegativos = np.array(indicesNegativos)

# Sacamos aleatoriamente un 10% de índices positivos y otro 10% de índices negativos
indexP = np.random.choice(int(indicesPositivos.size), size=int((indicesPositivos.size)/10), replace=False)
indexN = np.random.choice(int(indicesNegativos.size), size=int((indicesNegativos.size)/10), replace=False)

# Los volvemos a meter en listas
indicesPositivos = indicesPositivos.tolist()
indicesNegativos = indicesNegativos.tolist()
```

Ilustración 5. Implementación de la función para añadir ruido Pt. 1

```
# Intercambiamos el 10% de índices positivos sacados anteriormente por
# el otro 10% de índices negativos de la siguiente forma
for j in range(len(indicesPositivos)):
    if np.isin(j,indexP):
        # Introducimos el positivo en la lista de negativos
        indicesNegativos.append(indicesPositivos[j])
        # Borramos el positivo cambiado
        del indicesPositivos[j]

for k in range(len(indicesNegativos)):
    if np.isin(k, indexN):
        # Introducimos el negativo en la lista de positivos
        indicesPositivos.append(indicesNegativos[k])
        # Borramos el negativo cambiado
        del indicesNegativos[k]

# Los unimos todos en un nuevo array de etiquetas llamado ruido
ruido = np.zeros(y.size, np.int64)
for m in range(len(indicesPositivos)):
    ruido[indicesPositivos[m]] = 1
for n in range(len(indicesNegativos)):
    ruido[indicesNegativos[n]] = -1
```

Ilustración 6. Implementación de la función para añadir ruido Pt. 2

Ahora volvemos a dibujar la nube de puntos y la recta:

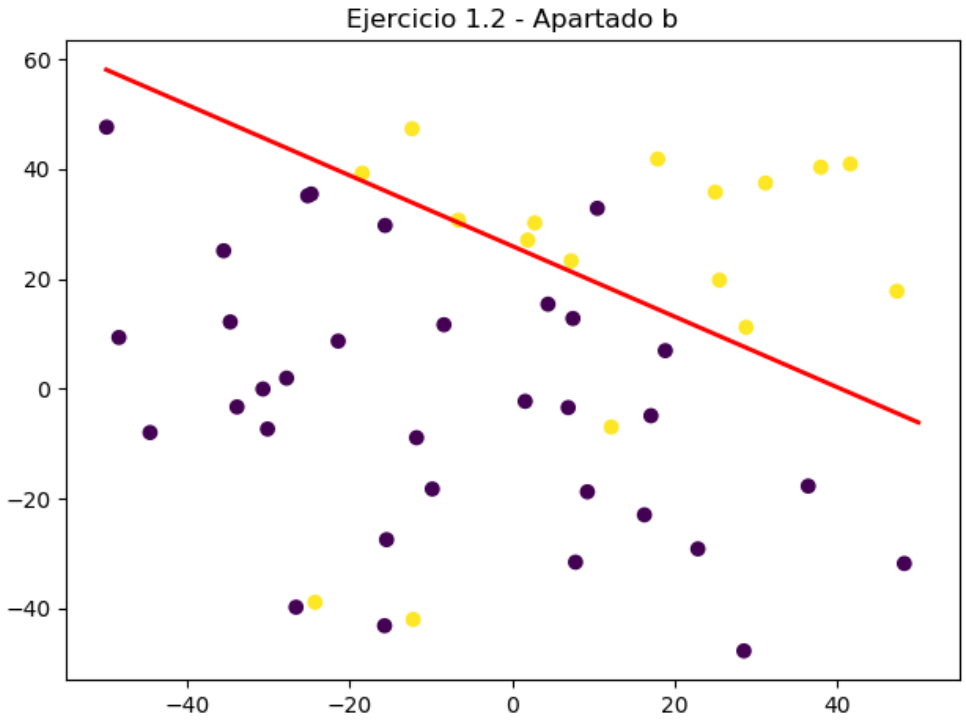

Ilustración 7. Nube de puntos etiquetados con ruido y línea que los separa

Ejercicio 3

En este ejercicio vamos a intentar dividir la nube de puntos con ruido utilizando funciones más complejas, que son las siguientes:

- $f1(x,y) = (x-10)^2 + (y-20)^2 - 400$
- $f2(x,y) = 0.5(x+10)^2 + (y-20)^2 - 400$
- $f3(x,y) = 0.5(x-10)^2 - (y+20)^2 - 400$
- $f4(x,y) = y - 20x^2 - 5x + 3$

Vamos a dibujar las gráficas gracias a la función proporcionada por el profesor llamada ***plot_datos_cuad (x, y, fz).*** Éste es el resultado:

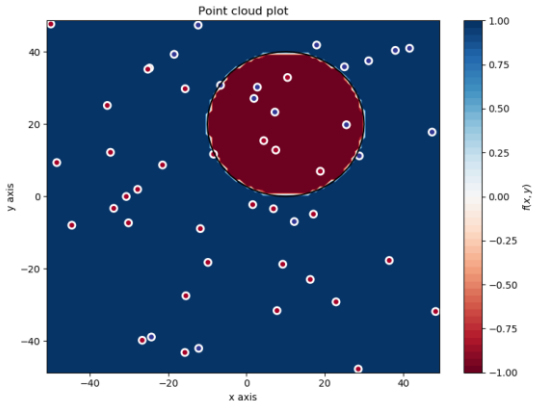

Ilustración 8. Función f1(x,y)

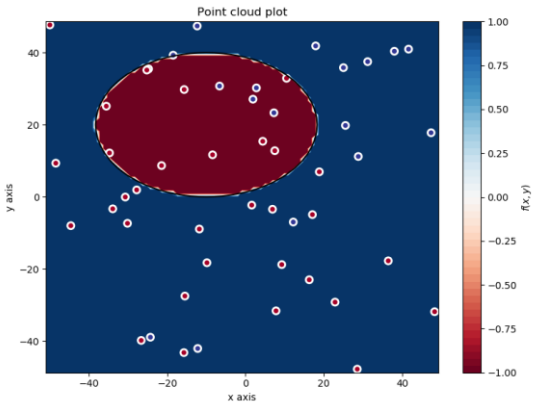

Ilustración 9. Función f2(x,y)

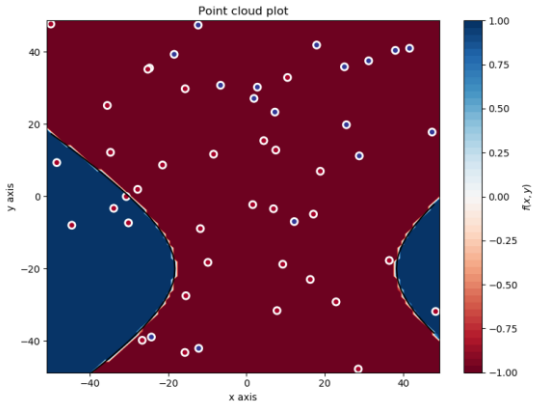

Ilustración 10. Función f3(x,y)

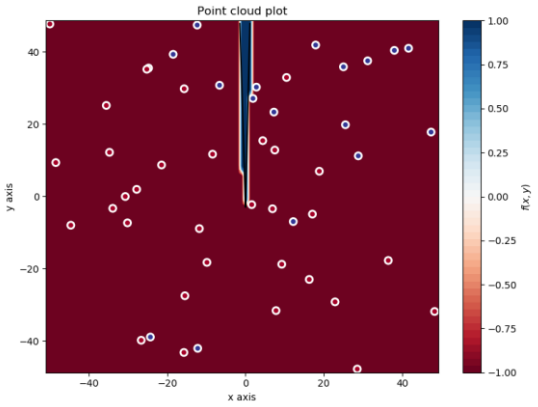

Ilustración 11. Función f4(x,y)

Podemos observar claramente que estas funciones, pese a ser más complejas, no son mejores clasificadoras, ya que los datos originalmente (pese a tener un poco de ruido) eran linealmente separables. En otros casos, donde la muestra no sea linealmente separable, estas funciones sí que serán mejores clasificadoras.

Sobre todo lo podremos observar en los casos donde estas funciones sean muy similares a la función objetivo, aunque la muestra contenga un poco de ruido.

MODELOS LINEALES

Algoritmo Perceptron (PLA)

Ejercicio 1

En este ejercicio tendremos que implementar el algoritmo del Perceptron. Este algoritmo calcula el hiperplano que divide una muestra clasificada de forma binaria. La función se llama ***ajusta_PLA (datos, label, max_iter, vini)*** y sus parámetros representan lo siguiente:

- **datos.** Es una matriz donde cada fila representa un dato y cada columna una característica suya. En este caso, las características son la posición de los puntos en un plano.
- **label.** Es un vector con las etiquetas de los datos anteriores.
- **max_iter.** Número máximo de iteraciones, ya que este algoritmo puede tardar mucho tiempo en converger.
- **vini.** Punto inicial del vector solución.

La implementación del algoritmo es la siguiente:

```
# Implementacion de la función Perceptrón que calcula el hiperplano solución
def ajusta_PLA(datos, label, max_iter, vini):
    w = np.array(vini)  # Copiamos en w el punto inicial
    iter = 0            # Creamos una variable para las iteraciones
    converge = False    # Booleano para detener la ejecución cuando converja

    # Comienzo del bucle
    while not converge:
        converge = True # Ponemos en True
        iter += 1       # Aumentamos el número de iteraciones

        # Recorremos todas las filas de datos (todos los puntos)
        for i in range(datos.shape[0]):
            # Realizamos el producto puntual de wT*xi
            prod = np.dot(w, datos[i])

            # Comprobamos que el signo del producto sea difeerente al de la etiqueta
            if (prod >= 0 and label[i] < 0) or (prod < 0 and label[i] > 0):
                w += label[i]*datos[i]  # Si cumple la condición, actualizamos w
                converge = False        # y seleccionamos que no converge

        # Corta el bucle en caso de que llegue a las iteraciones máximas
        if iter >= max_iter:
            break

    return w, iter
```

Ilustración 12. Implementación del PLA

Aparatado a). Tendremos que ejecutar el algoritmo de dos formas distintas: una con un vector de ceros como punto inicial; y la otra serán 10 ejecuciones con vectores aleatorios diferentes cada una.

Por pantalla tendremos que mostrar el número de iteraciones necesarias que ha necesitado para converger (en el segundo caso, haremos la media de las 10).

Los resultados han sido los siguientes:

- **(Array de ceros)** Valor de las iteraciones necesario para converger: 44

- **(Array aleatorio)** Valor medio de iteraciones necesario para converger: 58.4

Como conclusión podemos decir que el punto inicial también es importante a la hora de ejecutar el algoritmo, ya que podemos ver cómo ha variado el número de iteraciones dependiendo de él.

En mi caso, el valor obtenido con vectores aleatorios ha sido mayor, pero probablemente, en una de las 10 ejecuciones, el número de iteraciones haya sido menor que con el vector de ceros.

Adicionalmente, aquí muestro un gráfico de cómo divide el algoritmo y así comprobar visualmente que lo hace correctamente:

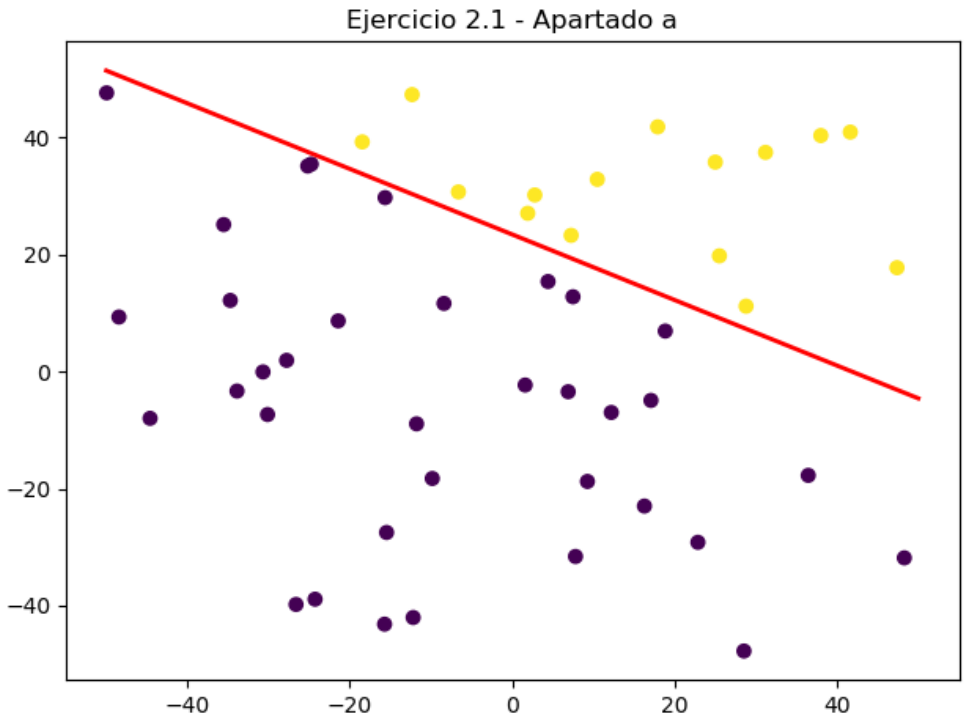

Ilustración 13. Demostración de cómo divide el PLA

Aparatado b). Vamos a hacer lo mismo que en el apartado a, pero esta vez utilizaremos las etiquetas con ruido. Los resultados son los siguientes:

- **(Array de ceros)** Valor de las iteraciones necesario para converger: 1000

- **(Array aleatorio)** Valor medio de iteraciones necesario para converger: 1000.0

Podemos observar cómo ha llegado al número máximo de iteraciones (en mi caso lo he puesto a 1000), tanto en el vector de ceros como en todas las ejecuciones con los vectores aleatorios. Esto se debe a que el ruido de la muestra utilizada no permite converger al algoritmo.

Regresión Logística

Ejercicio 2

En este ejercicio tendremos que generar nuestra propia función f a partir de un conjunto de datos D y sus etiquetas. Lo generaremos con la función **simula_unif(N=100, dim=2, rango=[0,2])** del ejercicio 1 y con esos parámetros.

Aparatado a). Ahora implementamos la función de **Regresión Logística (RL)** utilizando el **Gradiente Descendente Estocástico (SGD)**. Para la implementación he partido del sgd implementado en la práctica anterior, pero con las diferencias que comentaremos a continuación:

- Tras cada iteración, guardaremos el nuevo valor obtenido de *w* en la variable *w_ant*. El primero se inicializará a 0.

- Se realiza una permutación aleatoria de los datos en cada iteración.

- El bucle principal del algoritmo se detendrá cuando la **distancia euclídea** entre *w_ant* y *w* sea menor que 0.01.

- Y por último, el valor E_{in} que actualiza los pesos (y que en el *sgd* en la práctica anterior era la derivada del error) será un clasificador definido por:

$$\nabla E_{in} = -\frac{1}{N}\sum_{n=1}^{N}\frac{y_n x_n}{1+e^{y_n \boldsymbol{w^T(t)} x_n}}$$

Una vez hemos visto como es este clasificador vamos a ver su implementación:

```
# Función para la clasificación logística
def clasificadorRL(x, y, w):
    # Numerador de la fracción
    h = y*x
    # Calculo del valor del exponente de e
    z = y*w.dot(x.reshape(-1,))
    # Devolvemos el clasificador de la regresión logística
    return - (h) / (1 + np.exp(z))
```

Ilustración 14. Implementación del clasificador de RL

Una vez hemos explicado todo esto, ya podemos ver cómo he implementado este algoritmo de regresión logística utilizando el sgd:

```python
# Implementación del algoritmo de regresión logística utilizando el gradiente descendente estocástico
def sgdLR(x, y, initial_point, eta):
    w = np.copy(initial_point)  # Copiamos en w el punto inicial
    w = w.reshape(1,-1)         # Le asignamos una forma determinada al w

    # Calculamos el gradiente descendente
    while True:
        # Aplicamos una permutación aleatoria del tamaño de la muestra
        index = np.random.permutation(x.shape[0])
        # Asignamos los valores de los índices de 'x' e 'y' en los minibatches
        minibatch_x = x[index,:]
        minibatch_y = y[index]

        # Copiamos el w antes de modificarlo para comprobar posteriormente si parar
        w_ant = np.copy(w)

        # Calculamos el gradiente descendente
        for xn, yn in zip(minibatch_x, minibatch_y):
            gradiente = clasificadorRL(xn, yn, w)
            w -= eta * gradiente

        # Si la distancia euclídea entre el w_ant y este es menor que 0.01 paramos el bucle
        if np.linalg.norm(w_ant - w) < 0.01:
            break

    return w.reshape(-1,)
```

Ilustración 15. Implementación de la Regresión Logística con SGD

Adicionalmente, aquí muestro un gráfico de cómo divide el algoritmo sgdRL() y así comprobar visualmente que lo hace correctamente:

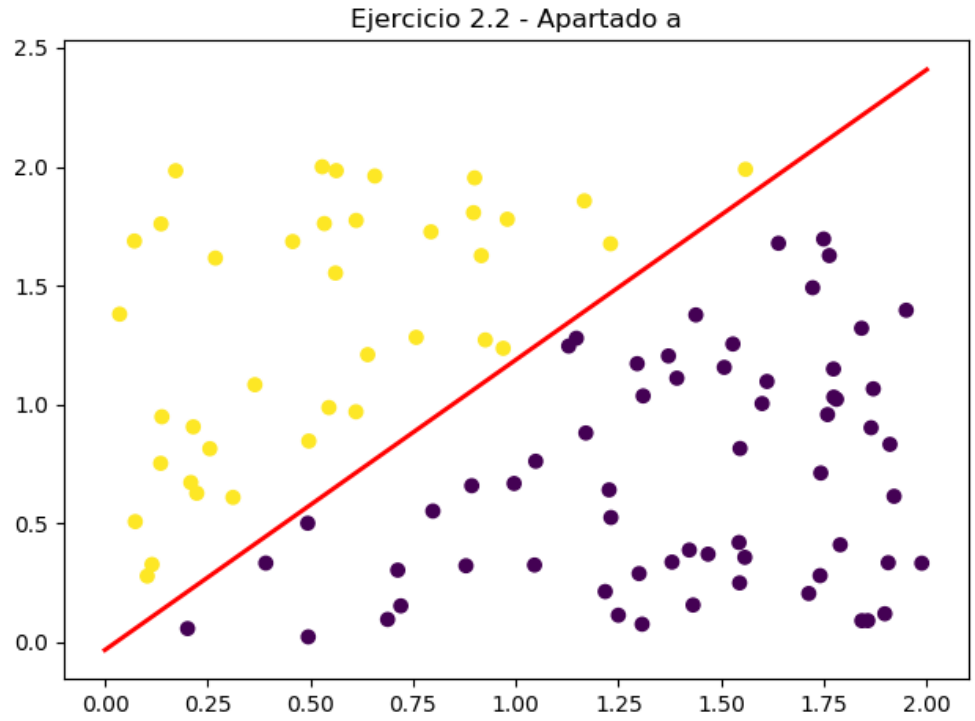

Ilustración 16. Demostración de cómo divide la Regresión Logística con SGD

Apartado b). A partir de la solución generada en el apartado anterior, clasificar una nueva muestra de N > 999 y con la misma dimensión y rango. Para mostrar el error que se produce he implementado la siguiente función:

```
# Función para estimar el error
def estimarError(x, y, w):
    y = y.reshape(-1,1) # Le asignamos una forma determinada al y
    w = w.reshape(-1,1) # Le asignamos una forma determinada al w

    # Realizamos el producto puntual de wT·x
    z = (x.dot(w))
    # Calculamos el ERM de la regresión logística
    Eout = np.log(1 + np.exp(-(y*z)))
    # Devolvemos la media de este valor
    return np.mean(Eout, axis=0)
```

Ilustración 17. Implementación de la función para estimar el error.

La función la he utilizado también para calcular el E_{in}, así que por pantalla mostraré los dos errores. El resultado es el siguiente:

- E_{in} = [0.06935797]

- E_{out} = [0.10728054]

Adicionalmente, también muestro un gráfico de cómo divide el algoritmo sgdRL() y así comprobar visualmente que lo hace correctamente

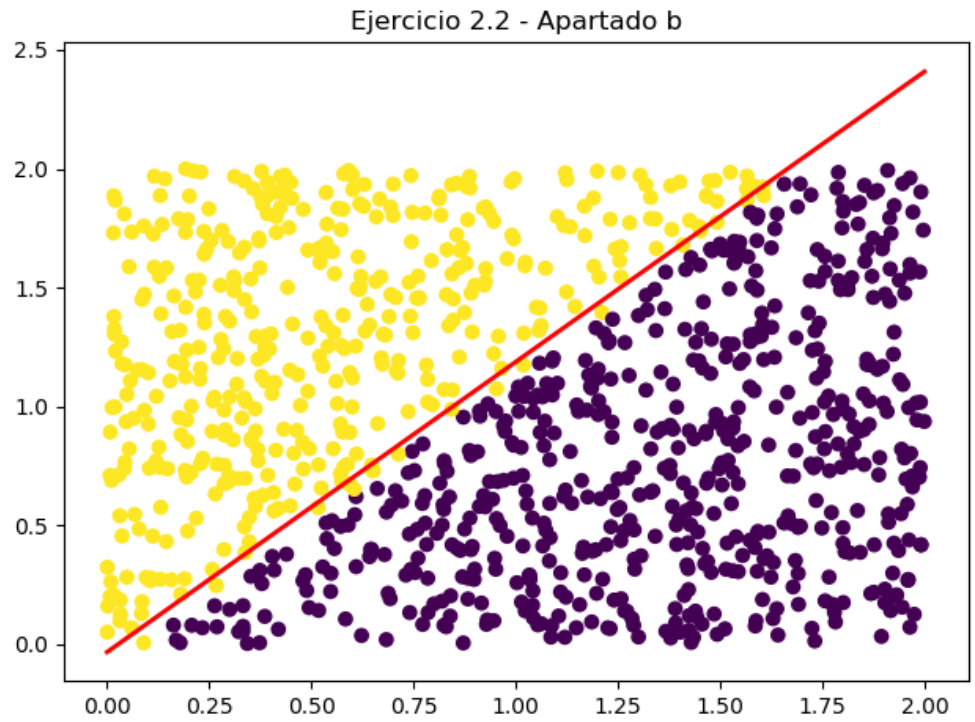

Ilustración 18. Demostración de cómo divide la Regresión Logística con SGD