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

Práctica 2: Programación

Simón López Vico

April 19, 2018

1 Ejercicio sobre la complejidad de H y el ruido (6 puntos)

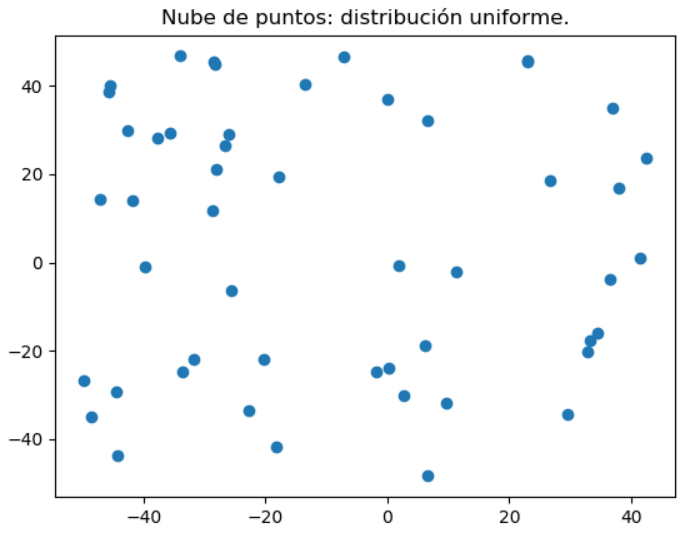
En este ejercicio debemos aprender la dificultad que introduce la aparición de ruido en las etiquetas a la hora de elegir la clase de funciones más adecuada. Haremos uso de tres funciones ya programadas:

- *simula_unif(N, dim, rango)*, que calcula una lista de N vectores de dimensión *dim*. Cada vector contiene *dim* números aleatorios uniformes en el intervalo *rango*.
- *simula_gaus(N, dim, sigma)*, que 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*.
- *simula_recta(intervalo)*, que simula de forma aleatoria los parámetros, $v = (a, b)$ de una recta, $y = ax + b$, que corta al cuadrado $[-50, 50] \times [-50, 50]$.

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

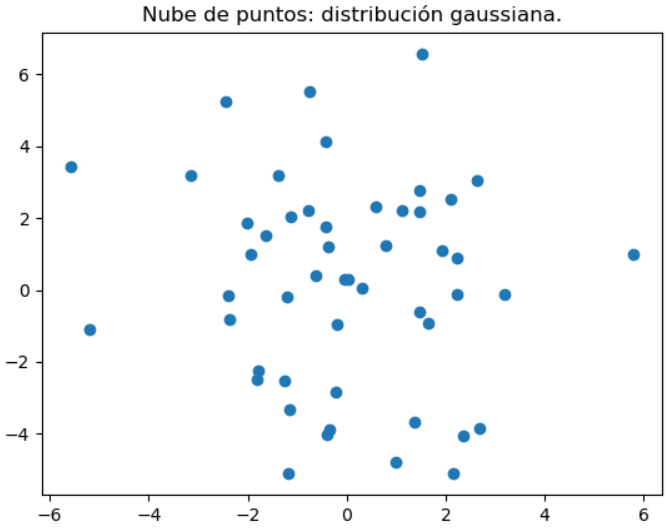
   (a) Considere $N = 50$, $dim = 2$, $rango = [-50, +50]$ con *simula_unif(N, dim, rango)*.

   Solución: Llamaremos a la función con `X_unif=simula_unif(N=50, dims=2, size=(-50, +50))`:



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

Solución: Llamaremos a la función con `X_gauss= simula_gaus((50,2), [5, 7])`:



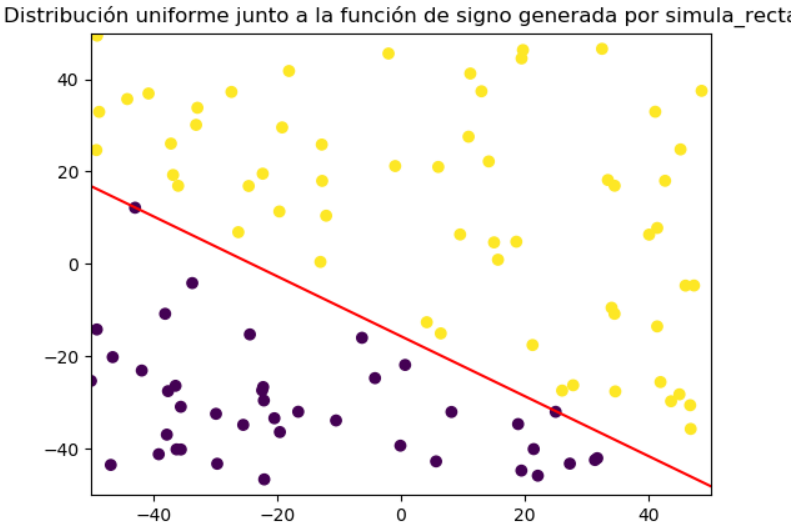
2. (2 puntos) Con ayuda de la función *simula_unif()* generar una muestra de puntos 2D a los que vamos añadir una etiqueta usando el signo de la función $f(x, y) = y - ax - b$, es decir el signo de la distancia de cada punto a la recta simulada con *simula_recta()*.

   (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.)

   Solución: *(Para este ejercicio, en vez de usar 50 valores he usado 100)* Comenzamos generando una distribución uniforme y los coeficientes a y b de la recta que separará la distribución con el siguiente código:

   ```
   X=simula_unif(100,2,(-50,+50))
   a,b=simula_recta((-50,50))
   ```

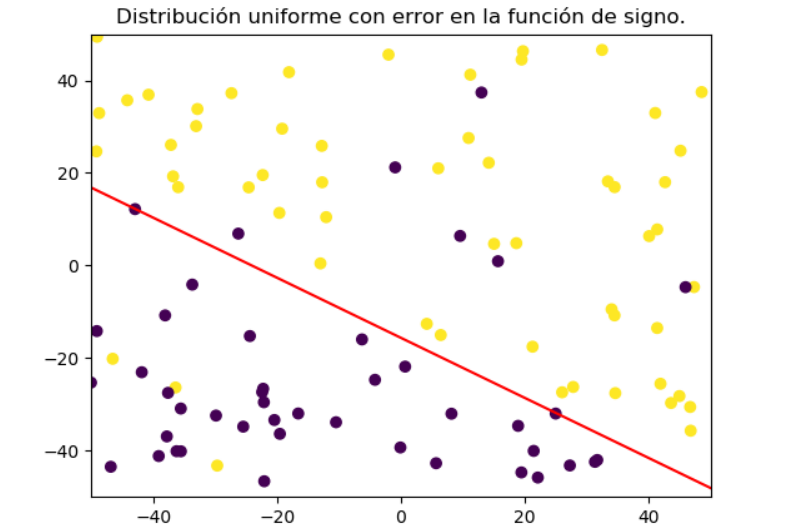
   Tras ello, etiquetaremos los valores respecto de la recta usando el método `y=label_data(X[:,0],X[:,1],a,b)`, que devolverá el signo de la función $f(x, y) = y - ax - b$, y finalmente *plotearemos* los datos generados, obteniendo la siguiente gráfica:



(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.)

Solución: Para cambiar las etiquetas aleatoriamente, usaremos la función `y=modif_aleatorio(y,10)`, donde `10` denota el porcentaje de etiquetas a modificar. Esta función guardará dos arrays `y_neg` e `y_pos` con los índices del vector `y` donde hay valores negativos y positivos respectivamente. Tras ello, se elegirán `len(y_neg)*(porcentaje/100)` cantidad de valores aleatorios del array y se modificará el valor almacenado en dichos índices en `y` (análogo con `y_pos`).

Tras aplicar la función, la gráfica quedará de la siguiente manera:



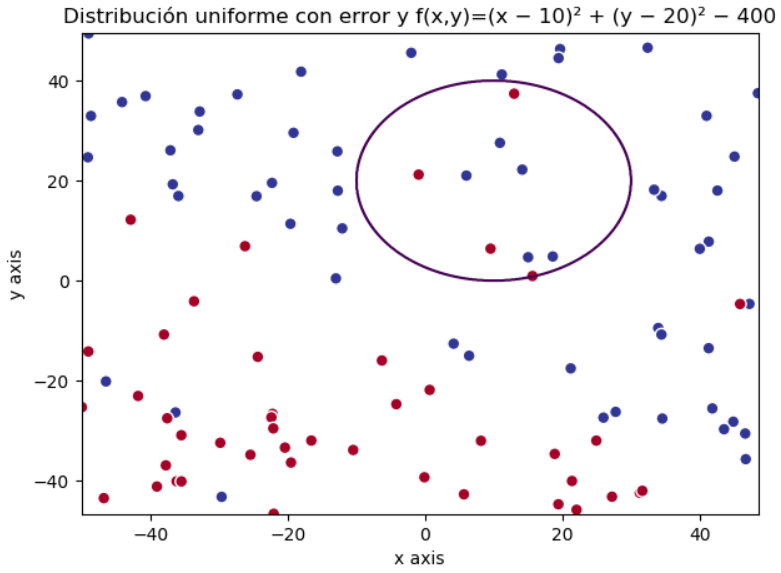
3. (3 puntos) Supongamos ahora que las siguientes funciones definen la frontera de clasificación de los puntos de la muestra en lugar de una recta:

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

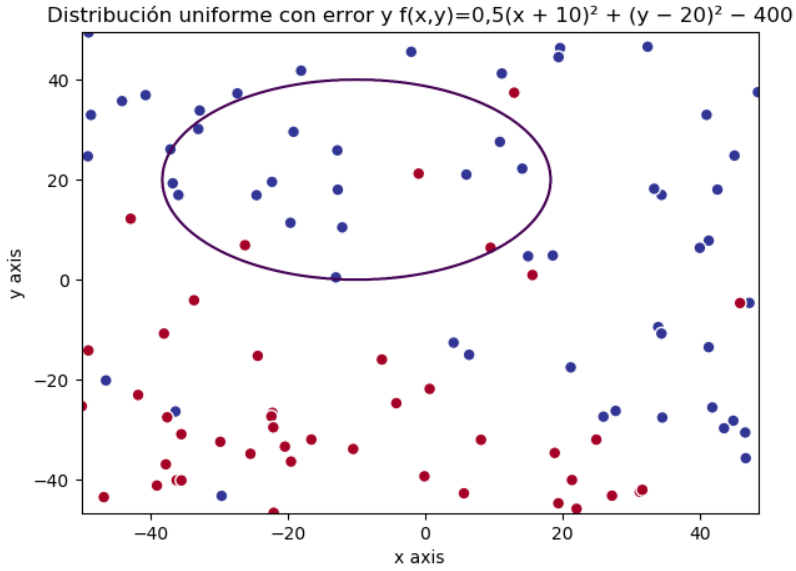
   Visualizar el etiquetado generado en *2b* junto con cada una de las gráficas de cada una de las funciones. Comparar las formas de las regiones positivas y negativas de estas nuevas funciones con las obtenidas en el caso de la recta ¿Son estas funciones más complejas mejores clasificadores que la función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

   Solución:
   Para visualizar las funciones en este ejercicio, usaremos la función `plot_datos_cuad([...])` facilitada por el profesor, la cual hemos modificado ligeramente para representar mejor los resultados. Las gráficas generadas serán las siguientes:

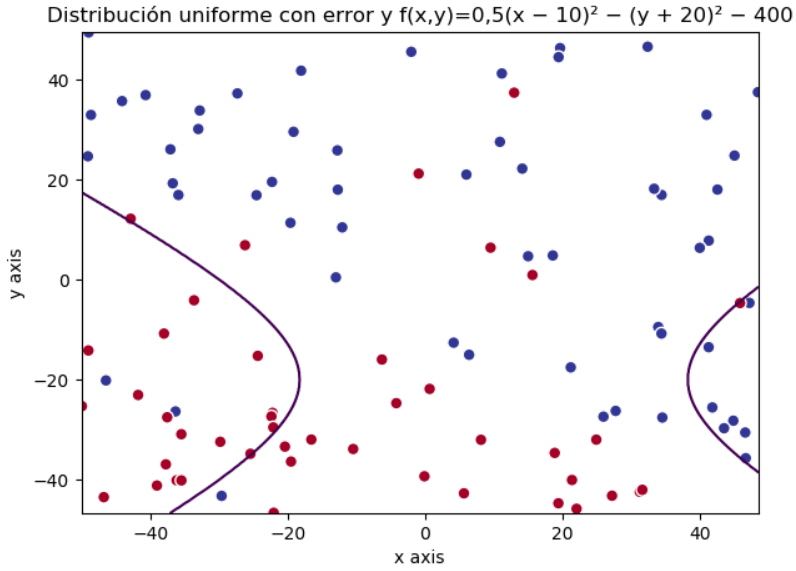
   - $f(x, y) = (x - 10)^2 + (y - 20)^2 - 400$



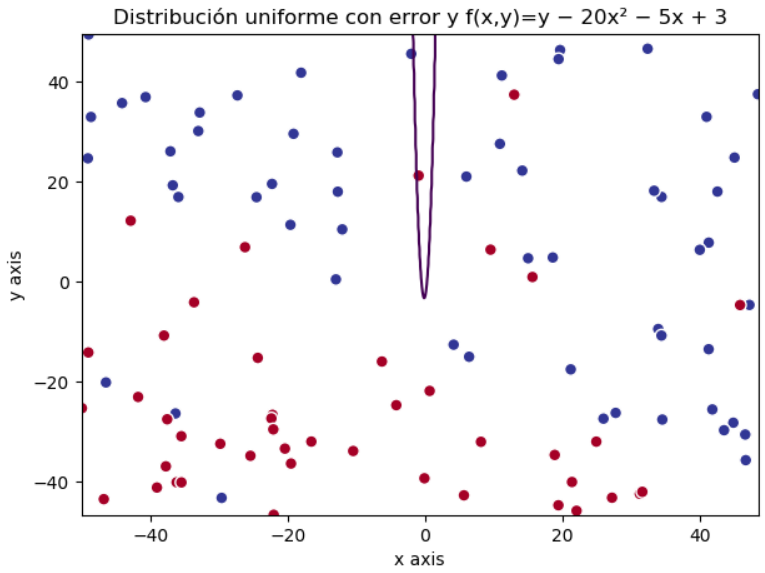
   - $f(x, y) = 0,5(x + 10)^2 + (y - 20)^2 - 400$



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



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



Como podemos ver, estas nuevas funciones no son mejores clasificadoras que la función lineal para la distribución generada, pues las etiquetas de la distribución están calculadas respecto de la recta $f(x, y) = y - ax - b$, haciendo que ésta siempre sea la mejor clasificadora de los datos (aunque un 10% de ellos hayan sido modificados).

2 Modelos Lineales (7 puntos)

1. **(3 puntos) Algoritmo Perceptron:** Implementar la función *ajusta_PLA(datos, label, max_iter, wini)* 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 *wini* el valor inicial del vector. La función devuelve los coeficientes del hiperplano.

   Solución: El algoritmo PLA implementado será el siguiente:

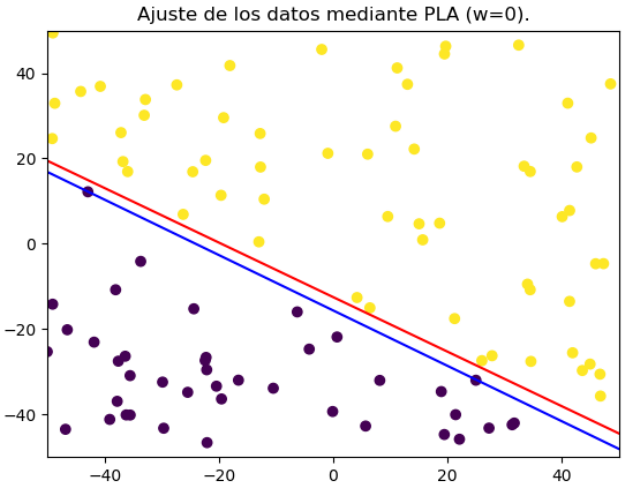
```
def ajusta_PLA(datos, label, max_iter, wini):
    iteraciones=0
    wold=np.copy(wini)

    while iteraciones < max_iter:
        i=0
        for i in range(len(datos)):
            if np.sign(np.dot(np.transpose(wini),datos[i])) != label[i]:
                wini=wold+np.dot(label[i],datos[i])

        if np.array_equal(wini,wold):
            break
        wold=np.copy(wini)
        iteraciones+=1
    return wini,iteraciones
```

   Además, tendremos que usar la función `aniade_coordenada(X)` implementada en la práctica 1, la cual añade una fila de unos a la matriz de datos.

   (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 y, 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.

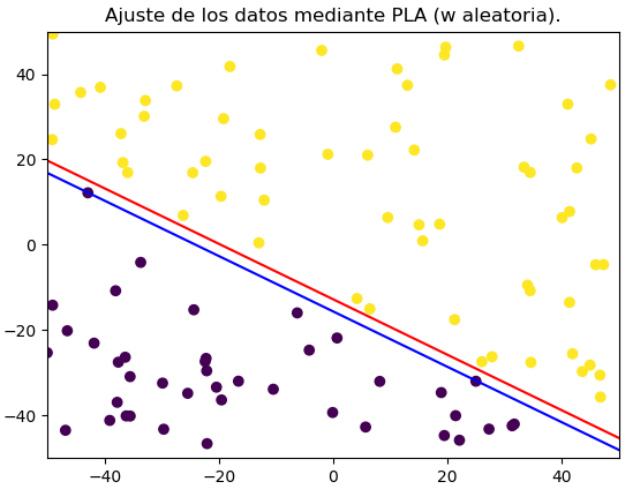
   Solución: Primero, ejecutaremos el algoritmo PLA y mostraremos en una gráfica sus resultados. Notemos que la recta azul es $f(x, y) = y - ax - b$ y la recta roja la función ajustada respecto los pesos de w.

   - Si el valor inicial de $w_{ini} = 0$, tendremos los siguientes resultados:



Para este ajuste se han necesitado 81 iteraciones y el valor de $w = (814, 41.62232603, 65.06305507)$.

- Si el valor inicial de w es aleatorio, el cual obtendremos con `wini=np.random.rand(3)`, tendremos los siguientes resultados:



Para este ajuste, el valor aleatorio inicial ha sido $w_{ini} = (0.83146036, 0.01393227, 0.1153776)$ se han necesitado 77 iteraciones y el valor de $w = (787.83146036, 40.08783301, 61.59960626)$.

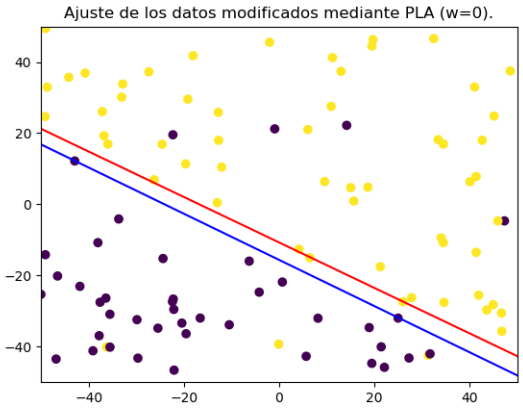
Además, tras 10 iteraciones del algoritmo con distintos valores iniciales aleatorios, obtendremos 73.9 iteraciones de media.

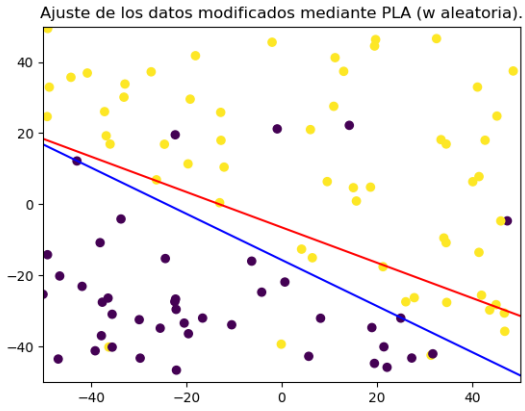
Podemos ver que las iteraciones hasta la convergencia no difieren demasiado, siendo menor el número de iteraciones con un valor inicial aleatorio. Aún así, la elección de un valor aleatorio o de un vector de ceros como w_{ini} no es muy relevante en la ejecución sobre dicha distribución, pues el dominio de los datos es $[-50, 50] \times [-50, 50]$ y el valor aleatorio se selecciona en $[0, 1] \times [0, 1]$, es decir, un valor cercano a 0, haciendo que la ejecución sobre dichos pesos

iniciales sea similar.

(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.

Solucion: Para esta cuestión, usaremos la función `y=modif_aleatorio(y,10)` usada en el ejercicio *2b* del apartado anterior. Tras ello, aplicaremos el algoritmo PLA obteniendo los siguientes resultados:





Podemos ver que las rectas no se llegan ajustar, además de que el número de iteraciones que hace es el número máximo de iteraciones pasado a la función (en nuestro caso 10000). Esto es debido a que w no llega a converger, pues el algoritmo PLA asegura una solución en tiempo finito para un conjunto de datos **separable linealmente**, lo cual no se cumple en la distribución con las etiquetas modificadas.

2. **(4 puntos) 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 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 $d = 2$ para que los datos sean visualizables, y sea $\mathcal{X} = [0, 2] \times [0, 2]$ con probabilidad uniforme de elegir cada $x \in \mathcal{X}$. 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.

Solución:
Para generar dicha muestra, usaremos la función `X=simula_unif(100,2,(0,2))`, y los coeficientes de la recta que corte la distribución generada los obtendremos con `a,b=simula_recta((0,2))`. Por último, etiquetaremos los datos con la función `y=label_data(X[:,0],X[:,1],a,b)`, de la que hablamos en el ejercicio 2 del anterior apartado.

(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 $\eta = 0,01$

Solución:
Reutilizaremos el código del SGD de la práctica 1 y modificaremos el gradiente de la función de error por el de la Regresión Logística, por lo que la función de los pesos será:

$$w_{current} = w_{old} - \eta \frac{1}{N} \sum_{i=0}^{N} -y_i x_i \sigma(-y_i w^T x_i)$$

donde σ (*sigma*) es la función logística, $\sigma(x) = \frac{1}{1-e^{-x}}$.

El código para realizar este algoritmo será el siguiente:

```
def sigma(y,w,X):
    omega_x=np.dot(np.transpose(w),X)
    return 1/(1+np.exp(-np.dot(-y,omega_x)))

def regresion_logistica_SGD(X_,y_,wini,lr,epsilon,longitud,max_iter):
    w=np.copy(wini)
    X, y=set_minibatches(X_,y_,longitud)
    wold=np.copy(w)

    i=0
    while i < max_iter:
        suma=np.zeros(len(X[1]), np.float64)
        j=0
        for j in range(len(X)):
            suma += np.dot(np.dot(-y[j],X[j]), sigma(y[j],w,X[j]))

        w=wold-lr*suma
        if np.linalg.norm(wold-w) < epsilon:
            break

        wold=np.copy(w)
        X, y=set_minibatches(X_,y_,longitud)
        i+=1

    print('Iteraciones necesarias:',i)

    return w
```

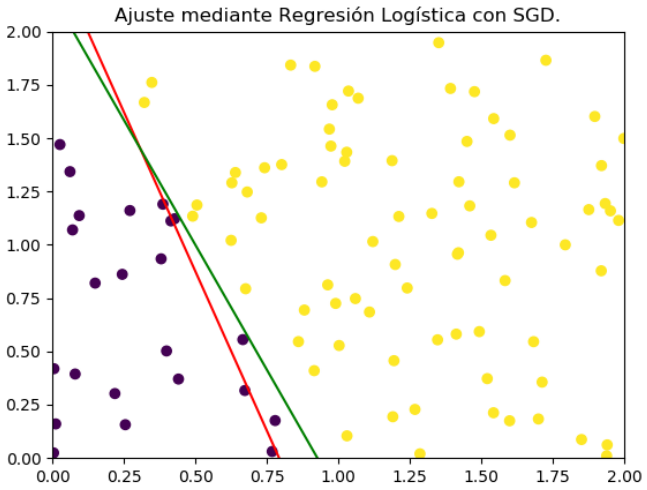
donde `X_` será la muestra, `y_` las etiquetas, `wini` el valor inicial de w, `lr` el valor del *learning rate*, `epsilon` el valor asignado al criterio de parada, `longitud` la cantidad de elementos para cada *minibatch* y `max_iter` el número máximo de iteraciones a realizar si no converge antes la solución.
Por otra parte, la función `set_minibatches(X,y,longitud)` será la misma función usada en la práctica 1 para el Gradiente Descendente Estocástico, la cual se encargará de hacer una permutación aleatoria sobre el conjunto de datos y sus etiquetas y escogerá `longitud` elementos sobre los que se aplicará el algoritmo.

(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).

Solución:
Para ejecutar el algoritmo con los valores que se piden, llamaremos a la función de la siguiente manera:

```
X=simula_unif(100,2,(0,2))
a,b=simula_recta((0,2))
y=label_data(X[:,0],X[:,1],a,b)
wini=[0,0,0]
X=aniade_coordenada(X)

w=regresion_logistica_SGD(X,y,wini,0.01,0.01,64,10000)
```

Tras **329 iteraciones**, obtendremos el valor de w, el cual generará la siguiente recta (en verde la recta que denota la frontera y en rojo la recta que genera el w ajustado):



A continuación, generaremos una nueva muestra de 5000 elementos, la etiquetaremos respecto de la misma recta usada anteriormente y comprobaremos su E_{out} respecto de la recta ajustada mediante Regresión Logística; además, será necesario reetiquetar los elementos de `y`, poniendo los -1 a 0. El código para ello será el siguiente:

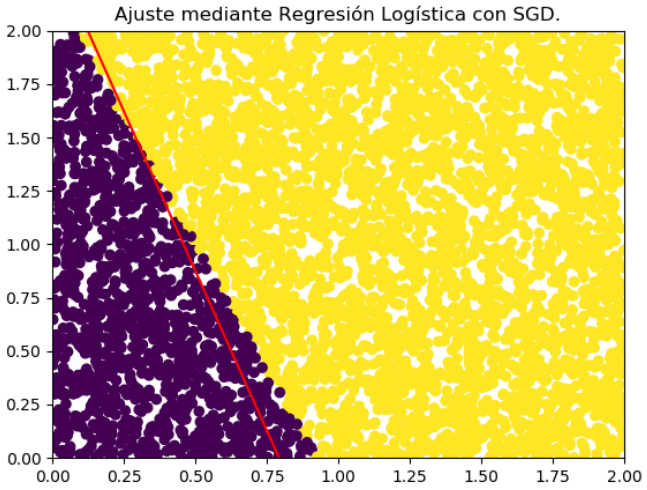
```
X=simula_unif(5000,2,(0,2))
y=label_data(X[:,0],X[:,1],a,b)
X=aniade_coordenada(X)
y=reetiqueta(y)
tasa=tasa_error(X,y,w)
```

donde `tasa_acierto(X,y,w)` se encargará de comprobar los errores respecto del sigmoide $\sigma(x) = \frac{1}{1-e^{-x}}$, y la implementación del algoritmo será:

```
def tasa_error(X,y,w):
    total=len(X)
    fallos=0
    i=0
    for i in range(total):
```

```
        omega_t=np.dot(np.transpose(w),X[i])
        valor=1/(1+np.exp(-omega_t))
        if valor>0.5 and y[i]==0:
            fallos+=1
        if valor<0.5 and y[i]==1:
            fallos+=1

    porcentaje_fallo=1.0*fallos/total
    return porcentaje_fallo
```

Gráficamente, tendremos la siguiente imagen:



en la cual podemos ver que hay elementos mal ajustados respecto de la recta generada con el valor de w.

Finalmente, el valor de tasa de error obtenido tras hacer el cálculo será $E_{out} = 0.0278$, es decir, un 2% de los datos estarán mal etiquetados respecto de la recta ajustada mediante Regresión Logística.