Memoria Práctica 2

Álvaro Fernández García

Abril 2018

NOTA: utilizar python3 para ejecutar el script.

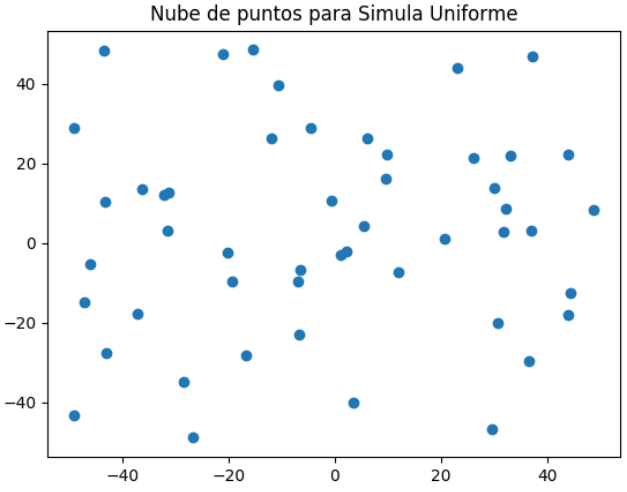
1 Ejercicio sobre la complejidad de H y el ruido

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:

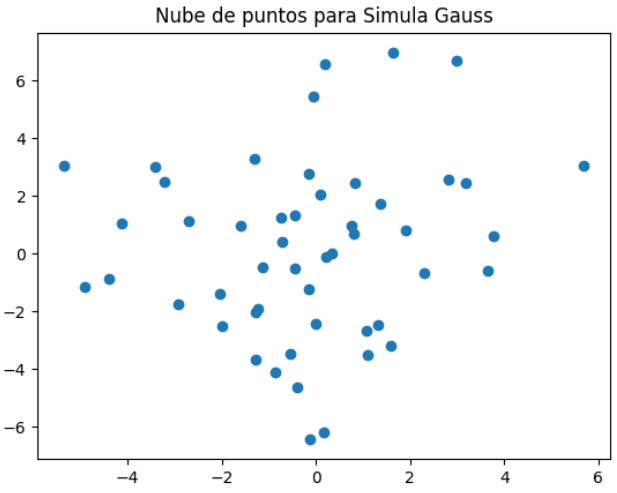
- $simula\_unif(N, dim, rango)$, que calcula una lista de N vectores de dimensión dim. Cada vector contiene dim números aleatorios uniformes en el intervalo rango.
- $simula\_gaus(N, dim, sigma)$, que calcula una lista de longitud N de vectores de dimensión dim, donde cada posición del vector contiene un número aleatorio extraido de una distribucción 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. Dibujar una gráfica con la nube de puntos de salida correspondiente.

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



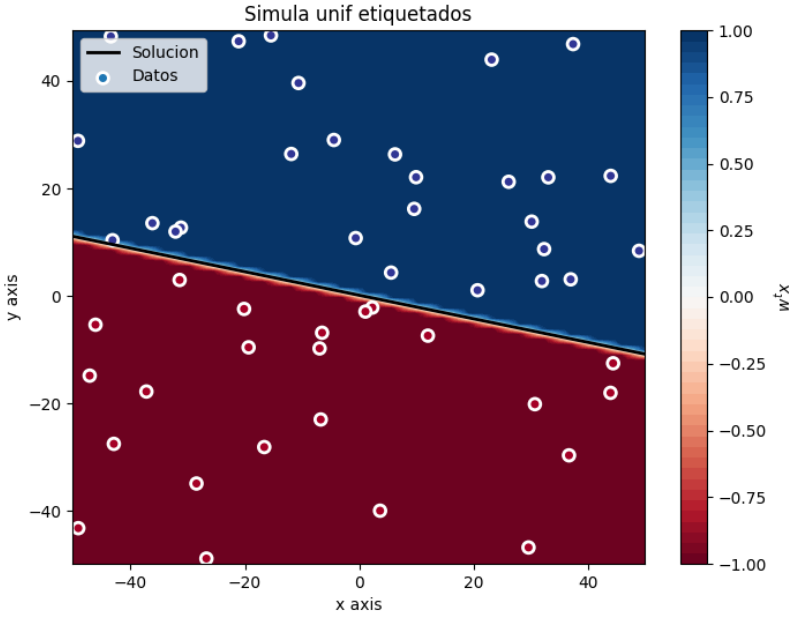
(b) Considere N = 50, dim = 2 y sigma = [5, 7] con $simula\_gaus(N, dim, sigma)$.



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 $simula\_recta()$.

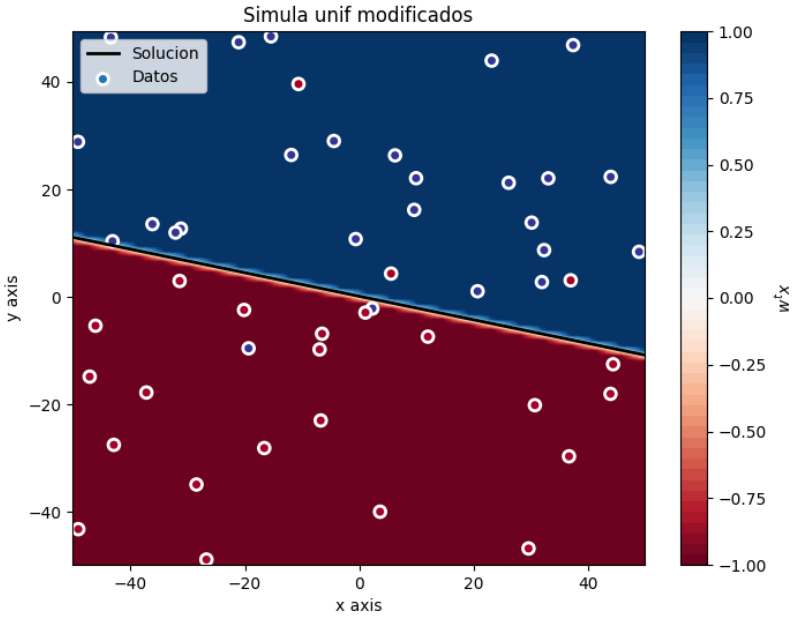
   (a) Dibujar una gráfica donde los puntos muestren el resultado de su etiqueta, junto con la recta usada para ello. (Observe que todos los puntos están bien clasificados respecto de la recta).

   Nota: se ha utilizado la muestra generada en el apartado a) del ejercicio anterior.



   (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).

   Hay un total de 23 etiquetas para -1, y 27 para 1. Redondeando, el 10% de 23 es 2 y el 10% de 27 es 3. Nos da un total de 5 etiquetas modificadas como se ve en la gráfica.

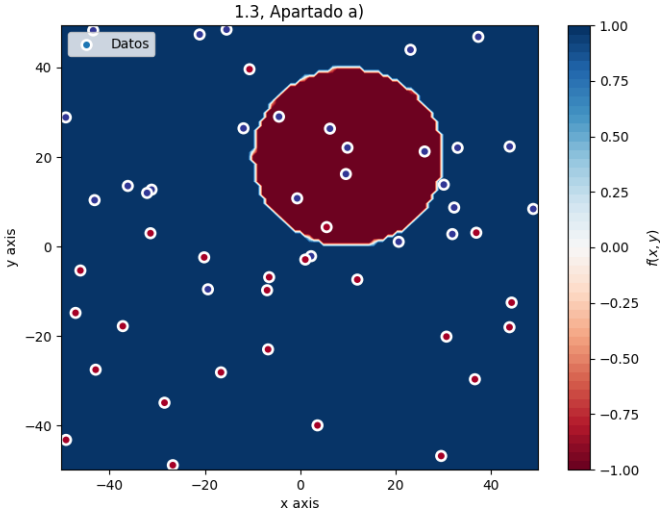


3. Supongamos ahora que las siguientes funciones definen la frontera de clasificación de los puntos de la muestra en lugar de una recta:

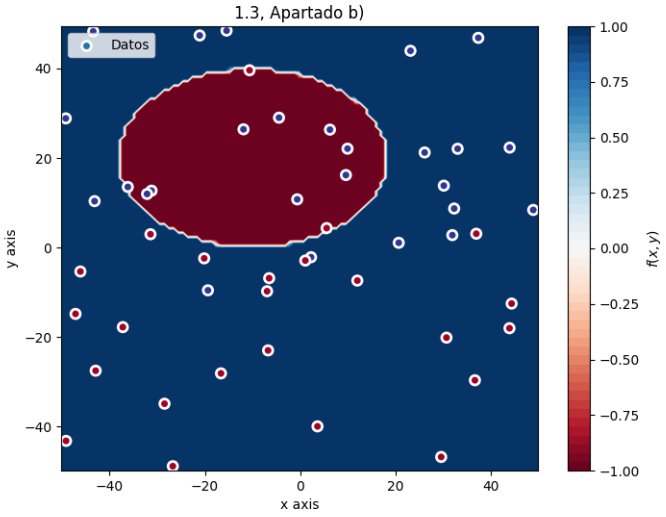
   - $f(x, y) = (x - 10)^2 + (y - 20)^2 - 400$
   - $f(x, y) = 0,5(x + 10)^2 + (y - 20)^2 - 400$
   - $f(x, y) = 0,5(x - 10)^2 - (y + 20)^2 - 400$
   - $f(x, y) = y - 20x^2 - 5x + 3$

   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 que ganan a la función lineal? Explicar el razonamiento.

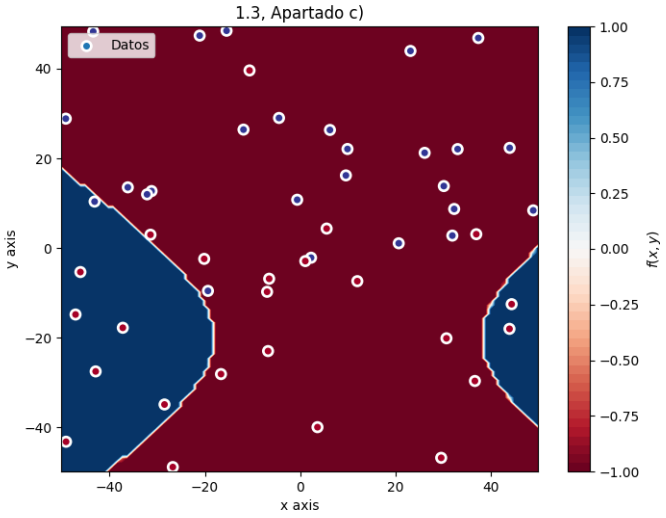
   - $f(x, y) = (x - 10)^2 + (y - 20)^2 - 400$



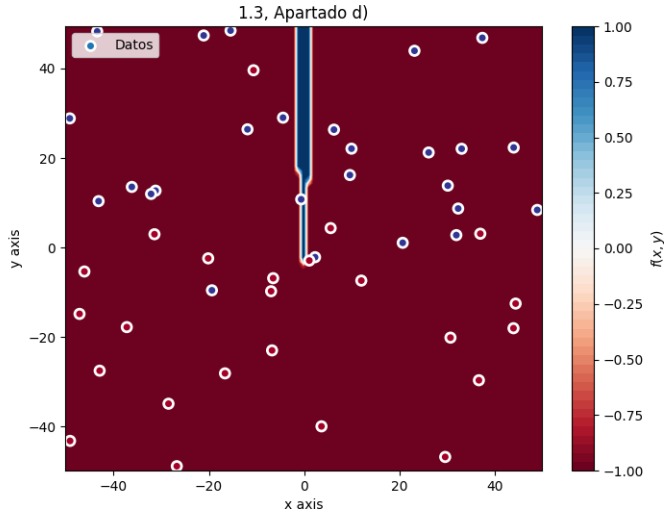
- $f(x, y) = 0,5(x + 10)^2 + (y - 20)^2 - 400$



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



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



La diferencia salta rápidamente a la vista viendo las gráficas. En el ejercicio anterior las regiones positiva y negativa quedaban separadas por una función que define una línea recta. Con las funciones presentes en el enunciado, las regiones no quedan divididas por una recta, sino por una esfera, una elípse, una hipérbola...

Con respecto a si estas funciones más complejas son mejores clasificadoras o no que la recta, la respuesta es depende. En el caso de que tengamos unos datos cuyas etiquetas sean linealmente separables (como en el caso del ejercicio 2a por ejemplo), obviamente la línea recta va a conseguir un mejor ajuste que cualquiera de las funciones de este ejercicio. Sin embargo, si tenemos una función cuyas etiquetas más o menos se ajustan a estas formas, el ajuste que realice una recta va a ser mucho peor que si utilizáramos alguna de estas funciones (ya que con una línea sería imposible separarlos) Por tanto, estas funciones "ganan" a una recta solo en el caso de que las etiquetas se distribuyan según estos patrones.

El caso del ejercicio 2b, nosotros mismos hemos introducido ruido aleatorio en un 10% de ambas regiones. Esto hace que los datos dejen de ser linealmente separables (no se puede encontrar una línea recta que divida sin ningún error los datos), sin embargo, por mucho que utilicemos algunas de las funciones de estos apartados tampoco conseguiríamos separarlos totalmente. ¿Cuál conviene más? Para responder a esa pregunta deberíamos calcular sus errores y así poder valorar cuál nos interesa más.

En definitiva, que utilicemos una función más compleja no nos va a garantizar que tengamos un mejor ajuste, todo depende de los datos que empleemos. Aun así, no debemos olvidar que, en aprendizaje automático, la clase de funciones siempre se elige antes de ver los datos.

2 Modelos Lineales

1. **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 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.

La implementación del Perceptrón es la siguiente:

```
def ajusta_PLA(X, y, MAX_ITERS, w0):
    w = w0
    iters = MAX_ITERS # Por si no lo encontrara
    for n in range(1, MAX_ITERS+1):
        waux = w.copy()

        for i in range(X.shape[0]):
            if(np.sign((w.transpose()).dot(X[i])) != y[i]):
                w = w + y[i] * X[i]

        if(np.all(np.abs(w-waux)<10**(-6))):
            iters = n
            break

    return w, iters
```

(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.

- $w_0 = \{0, 0, 0\} \rightarrow$ Número de iteraciones en converger = 2.
- $w_0 = \{$Aleatorio entre 0 y 1$\} \rightarrow$ Número medio de iteraciones en converger tras 10 ejecuciones = 2.5.

Como valoración del resultado, decir que realmente no hay mucha diferencia con respecto al número de iteraciones que tarda en converger. Existe una recta que divide a los datos perfectamente, y tanto si los pesos inciales están a cero, como si son un número aleatorio entre 0 y 1, para la muestra generada, el perceptrón la encuentra en pocas iteraciones.

(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.

- $w_0 = \{0, 0, 0\} \rightarrow$ No converge, llega hasta el número máximo de iteraciones (1000).
- $w_0 = \{$Aleatorio entre 0 y 1$\} \rightarrow$ Tampoco converge, y llega a las 1000 iteraciones en todos los casos.

La razón de que esto ocurra es precisamente el ruido que hemos metido en el apartado 2b. Tras meter este ruido, los datos dejan de ser linealmente separables, y en esos casos, el perceptrón es incapaz de converger, por lo que cicla indefinidamente. Nosotros para que no cicle hemos limitado las iteraciones. Por eso da como salida 1000 en todos los experimentos, da igual el número de iteraciones que pongamos, no va a encontrar una solución en ningún caso, independientemente de cuáles sean los pesos iniciales.

2. **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 para que los datos sean visualizables, y sea $X = [0, 2] \times [0, 2]$ con probabilidad uniforme de elegir cada $x \in X$. Elegir una línea en el plano que pase por 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 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:

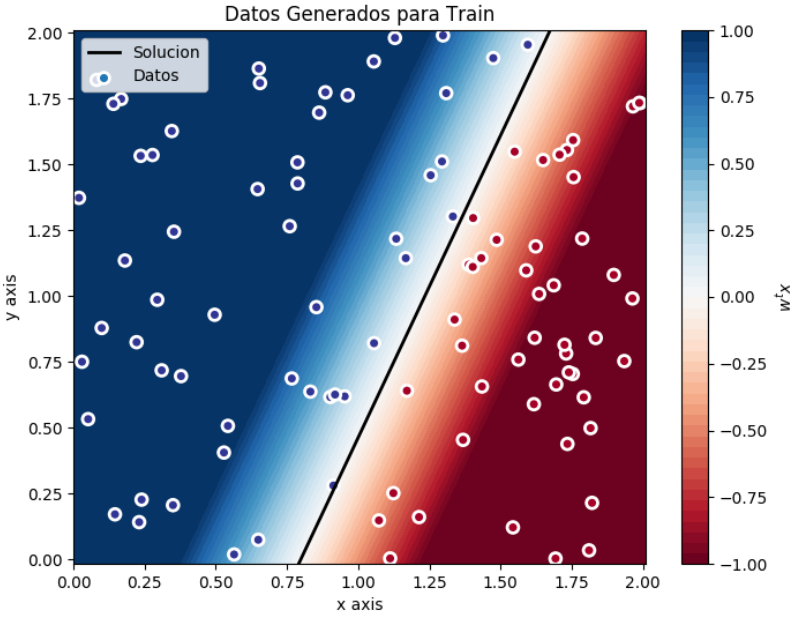
- Inicializar el vector de pesos con valores 0.
- Parar el algoritmo cuando $||w(t-1) - w(t)|| < 0,01$, donde w (t) denota el vector de pesos al final de la época t. Una época es un pase completo a través de los N datos.
- Aplicar una permutación aleatoria, $1, 2, \ldots, N$, en el orden de los datos antes de usarlos en cada época del algoritmo.
- Usar una tasa de aprendizaje de $\nu = 0,01$

Aquí se muestra el código de las tres funciones implicadas en la regresión logística: sigmoide, gradiente y el algoritmo del SGD:

```
# Implementacion del sigmoide
def sigmoid(x):
    return 1 / float(1 + np.exp(-x))


# Implementacion del gradiente de la regresion lineal logistica:
def grad(X, y, w):
    aux = np.zeros(w.shape[0], np.float64)

    for i in range(X.shape[0]):
        aux += -y[i] * X[i] * sigmoid(-y[i] * (w.transpose()).dot(X[i]))

    aux *= 1/float(X.shape[0])

    return aux

# Implementacion del gradiente desdendente estocastico:
def SGD(X, y, w, n, gd_func, MAX_ITERS, BATCHSIZE):
    # Las epocas:
    for _ in range(MAX_ITERS):
        waux = w.copy()
        # Barajar la muestra
        idx = np.arange(X.shape[0])
        np.random.shuffle(idx)
        X = X[idx]
        y = y[idx]
        # Iterar en los batches
        for i in range(0, X.shape[0], BATCHSIZE):
            w = w - n * gd_func(X[i:i+BATCHSIZE], y[i:i+BATCHSIZE], w)

        if(np.all(np.abs(w-waux) < 0.01)):
            break

    return w
```

(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).

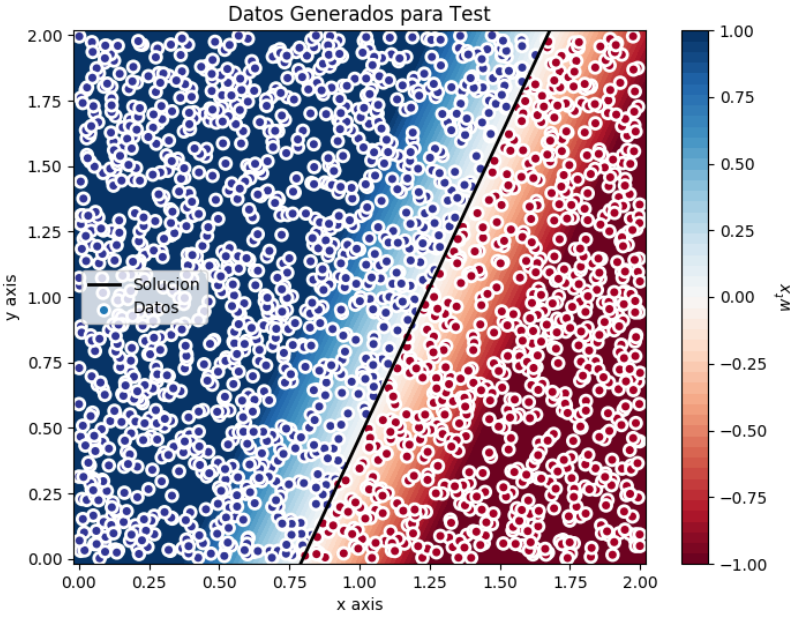
Los datos utilizados para el training, junto con la recta para etiquetar, son los siguientes:



Tras ajustar los pesos, se consiguen los siguientes errores:

- $E_{in} = 0.6893794576110509$
- Error de clasificación: 30 de 100 (0.3)

Los datos utilizados para el test, junto con la recta para etiquetar, son los siguientes (se ha generado un conjunto de 2000 ejemplos):



Se consiguen los siguientes errores:

- $E_{out} = 0.6884670561197037$
- Error de clasificación: 477 de 2000 (0.24)

Como podemos observar, tanto el E_{in} como el E_{out} son bastante similares y no demasiado altos (cercanos a 0.7, lo que me parece aceptable). No obstante, debemos recordar que nos encontramos ante un problema de clasificación y lo que realmente importa es el error de clasificación. Este error también es muy parecido tanto en el train como en el test (básicamente ambos se equivocan en un 30%). Un 30% de los datos es bastante, y posiblemente, la pobreza de este ajuste se deba a que el umbral de los pesos entre épocas es demasiado grande (0.01). Modificando este umbral y aumentando el número de iteraciones (por ejemplo a 10^{-6} como en el caso del perceptrón y dejándole hasta las 2000 iteraciones) se consiguen mejores resultados: (0.2 de E_{in} y E_{out} y sólo 0.01 de error de clasificación en el train y 0.035 en el test).

3 Bonus

1. **Clasificación de Dígitos**. Considerar el conjunto de datos de los dígitos manuscritos y seleccionar las muestras de los dígitos 4 y 8. Usar los ficheros de entrenamiento (training) y test que se proporcionan. Extraer las características de intensidad promedio y simetría en la manera que se indicó en el ejercicio 3 del trabajo 1.

    1. Plantear un problema de clasificación binaria que considere el conjunto de entrenamiento como datos de entrada para aprender la función g.

        La clasificación es sencilla. Simplemente debemos de modificar las etiquetas que se asignan en el conjunto y. Para esta práctica, se ha asignado la etiqueta -1 al número 4, y la etiqueta 1 al número 8. El ajuste y los errores se realizan con la etiquetas $[-1, 1]$. Si luego fuese necesario mostrar la predicción hecha, bastaría con hacer el cambio a la inversa. Cuando nuestro clasificador responda -1, diremos que predice un 4 y cuando responda 1, diremos que predice un 8.

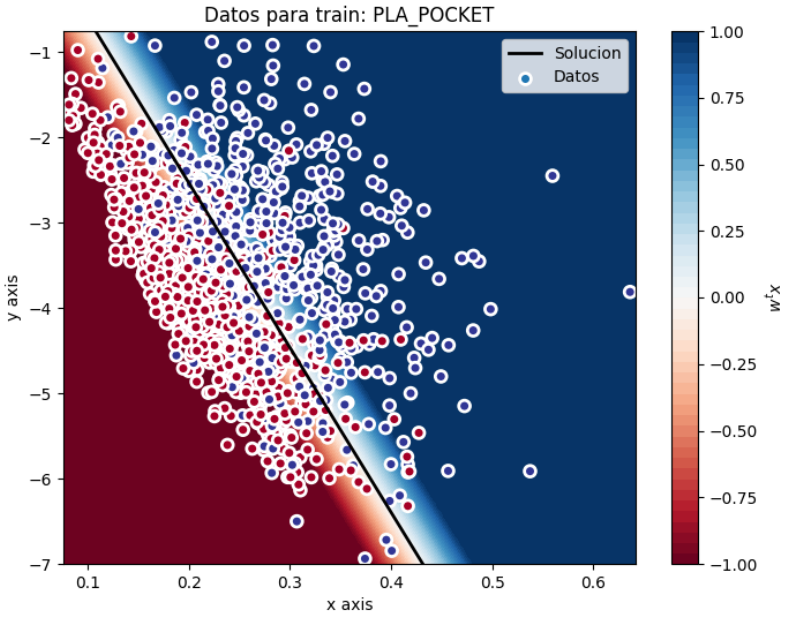
    2. Usar un modelo de Regresión Lineal y aplicar PLA-Pocket como mejora. Responder a las siguientes cuestiones.

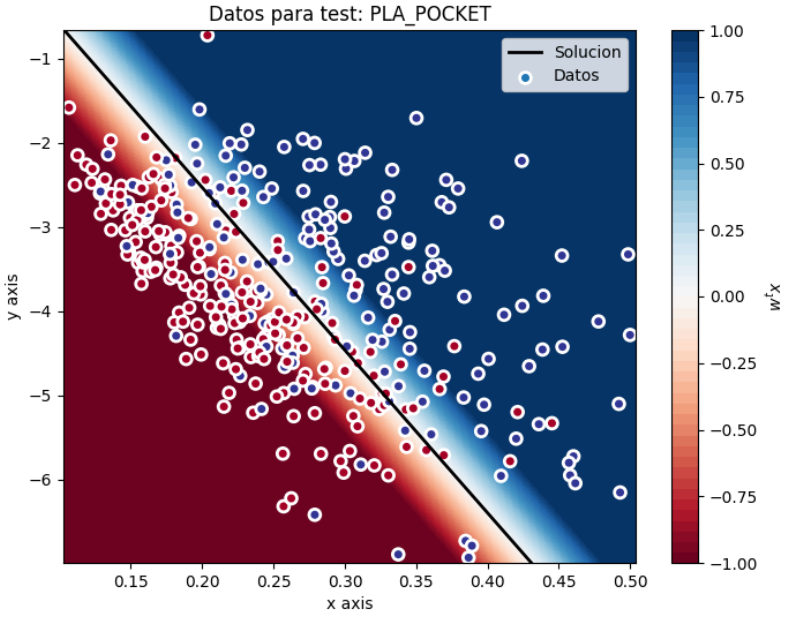
        Como modelo de Regresión Lineal se ha utilizado el algoritmo de la Pseudo-Inversa. Posteriormente, los pesos obtenidos por dicho algoritmo se pasan al algoritmo PLA-Pocket para que los mejore. El código del algoritmo PLA-Pocket es el siguiente:

```
def PLA_pocket(X, y, MAX_ITERS, w0, Error):
    w = w0
    best = w0
    for n in range(1, MAX_ITERS+1):
        waux = w.copy()

        for i in range(X.shape[0]):
            if(np.sign((w.transpose()).dot(X[i])) != y[i]):
                w = w + y[i] * X[i]

        if(Error(X, y, w) < Error(X, y, best)):
            best = w.copy()

        if(np.all(np.abs(w-waux)<10**(-6))):
            break

    return best
```

Donde la función de Error calcula cuál de las soluciones es mejor. En esta práctica, se ha utilizado como función de error la que calcula el error de clasificación.

(a) Generar gráficos separados (en color) de los datos de entrenamiento y test junto con la función estimada.





(b) Calcular E_{in} y E_{test} (error sobre los datos de test).

- E_{in}: 38.699724310256265
- Error de clasificacion para train: 269 de 1194 (0.22529313232830822)
- E_{test}: 40.74009067556704

- Error de clasificacion para test: 93 de 366 (0.2540983606557377)

(c) Obtener cotas sobre el verdadero valor de E_{out}. Pueden calcularse dos cotas una basada en E_{in} y otra basada en E_{test}. Usar una tolerancia $\delta = 0,05$. ¿Que cota es mejor?

La fórmula para calcular la cota basada en E_{in} es la siguiente:

$$E_{out}(h) \leq E_{in}(h) + \sqrt{\frac{8}{N} \log \frac{4((2N)^{d_{vc}} + 1)}{\delta}} \tag{1}$$

El tamaño de la muestra utilizado en el train es 1194, en el modelo lineal, la dimensión de VC es el número de variables (3 en nuestro caso), y el E_{in} redondeando es 38,7. Si sustituimos tenemos que:

$$E_{out}(g) \leq 39.13$$

con una probabilidad del 95%

Ahora vamos a calcular la cota pero basándonos en el E_{test}. Ahora N es 366 y E_{test} redondeando es 40,74:

$$E_{out}(g) \leq 41,47$$

con una probabilidad del 95%

Con respecto a que cota es mejor, podríamos pensar que E_{in} es más pequeña y más conveniente para nosotros, pero esto no es así. Si bien el error en el train puede ser útil como una guía para el proceso de entrenamiento, es inútil si el objetivo es obtener un pronóstico preciso de E_{out}. Si estuviéramos desarrollando un sistema para un cliente, se necesitaría una estimación más precisa para que el cliente sepa qué tan bien se espera que el sistema funcione. Por tanto deberíamos de darle una mayor importancia a la cota calculada a partir de E_{test}, ya que está más próxima a como realmente el ajuste se va a comportar para los datos que estén fuera de la muestra (nunca olvidemos que este es el verdadero objetivo del Aprendizaje Automático).

En resumen, con respecto a valor es mejor la del E_{in}, pero deberíamos darle una mayor importancia a la del E_{test}, por no decir que realmente la útil y la mejor para los estudios es esta última.