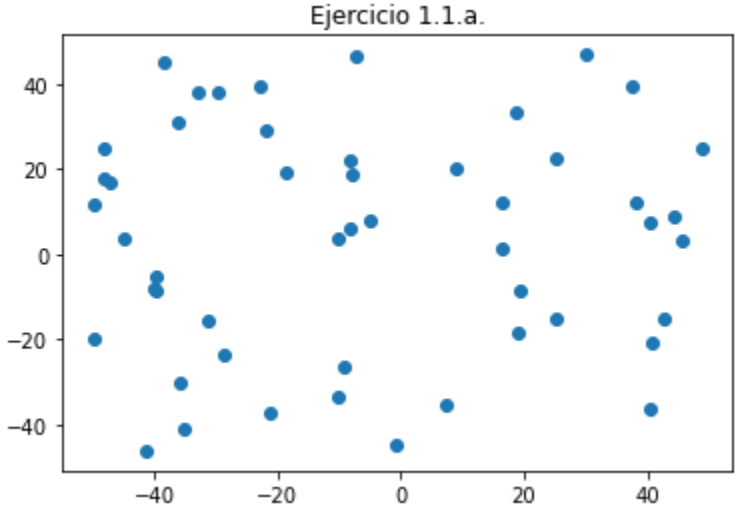
Guillermo García Arredondo - Grupo 3

Aprendizaje Automático
Práctica 2

1. EJERCICIO SOBRE LA COMPLEJIDAD DE H Y EL RUIDO

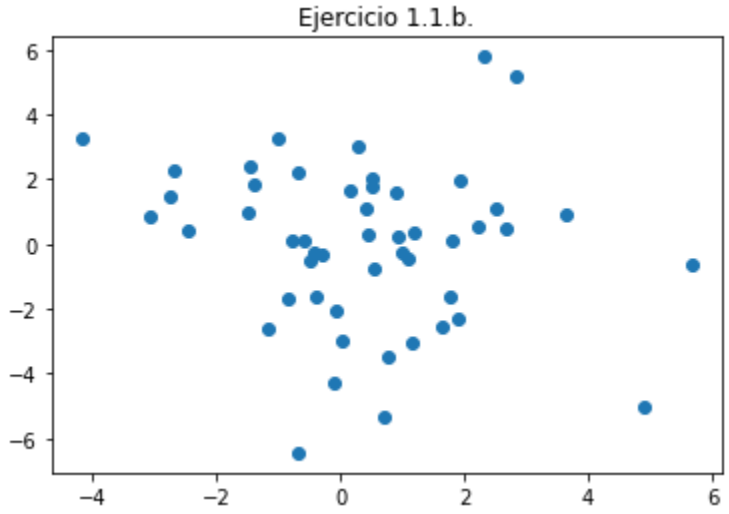
1.1. Dibujar las gráficas con las nubes de puntos simuladas con las siguientes condiciones:

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



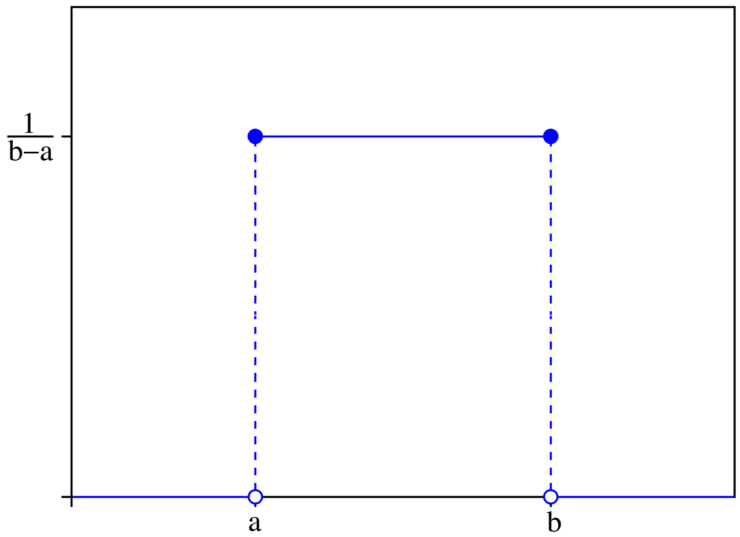
La distribución uniforme es una distribución de probabilidad donde, dentro de un intervalo dado, todos los puntos son igual de probables. Es coherente entonces que la gráfica obtenida tenga los 50 puntos aleatorios dispersos por todo el intervalo [-50, 50]x[-50, 50].

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

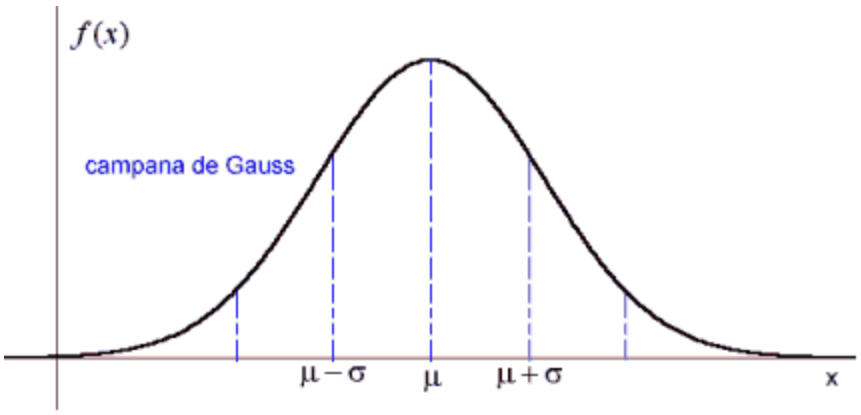


La distribución normal o gaussiana se caracteriza por la forma acampanada y simétrica de su función de densidad (conocida como Campana de Gauss), donde el valor más probable es el central y conforme se aleja uno de ese valor disminuye la probabilidad.

Si la función de densidad para la distribución normal era ésta:



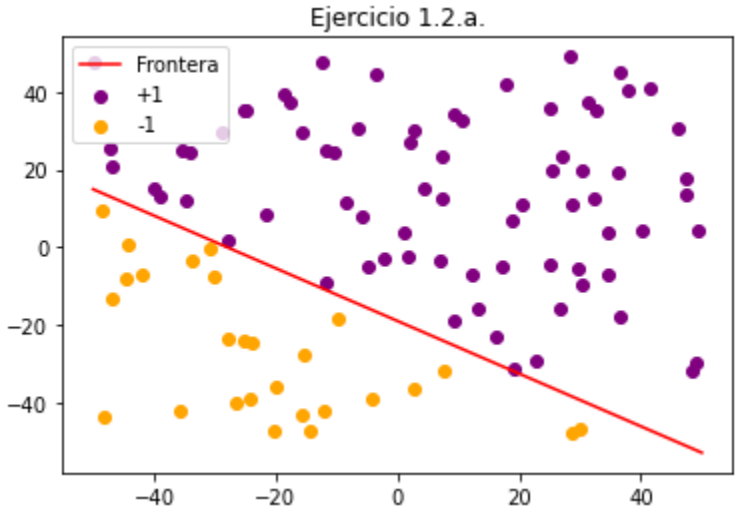
La función de densidad para la distribución gaussiana es ésta:



Por ello, en nuestra gráfica, la mayoría de nuestros puntos se hallan cercanos al valor [0, 0], y encontramos cada vez menos puntos conforme nos alejamos de ese valor central.

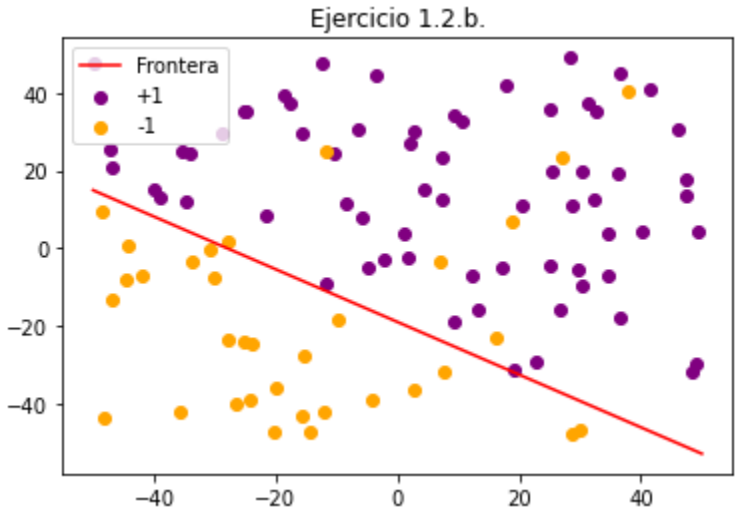
1.2. Vamos a valorar la influencia del ruido en la selección de la complejidad de la clase de funciones. Con ayuda de la función simula_unif(100,2,[−50,50]) generamos una muestra de puntos 2D a los que vamos añadir una etiqueta usando el signo de la función f(x,y) = y−ax−b, es decir el signo de la distancia de cada punto a la recta simulada con simula_recta().

a) **Dibujar un gráfico 2D donde los puntos muestren (use colores) el resultado de su etiqueta. Dibuje también la recta usada para etiquetar (observe que todos los puntos están bien clasificados respecto de la recta).**



Los puntos están bien clasificados respecto a la frontera. Los que están por debajo de la recta son etiquetados como *-1* y los que están por encima como *+1*.

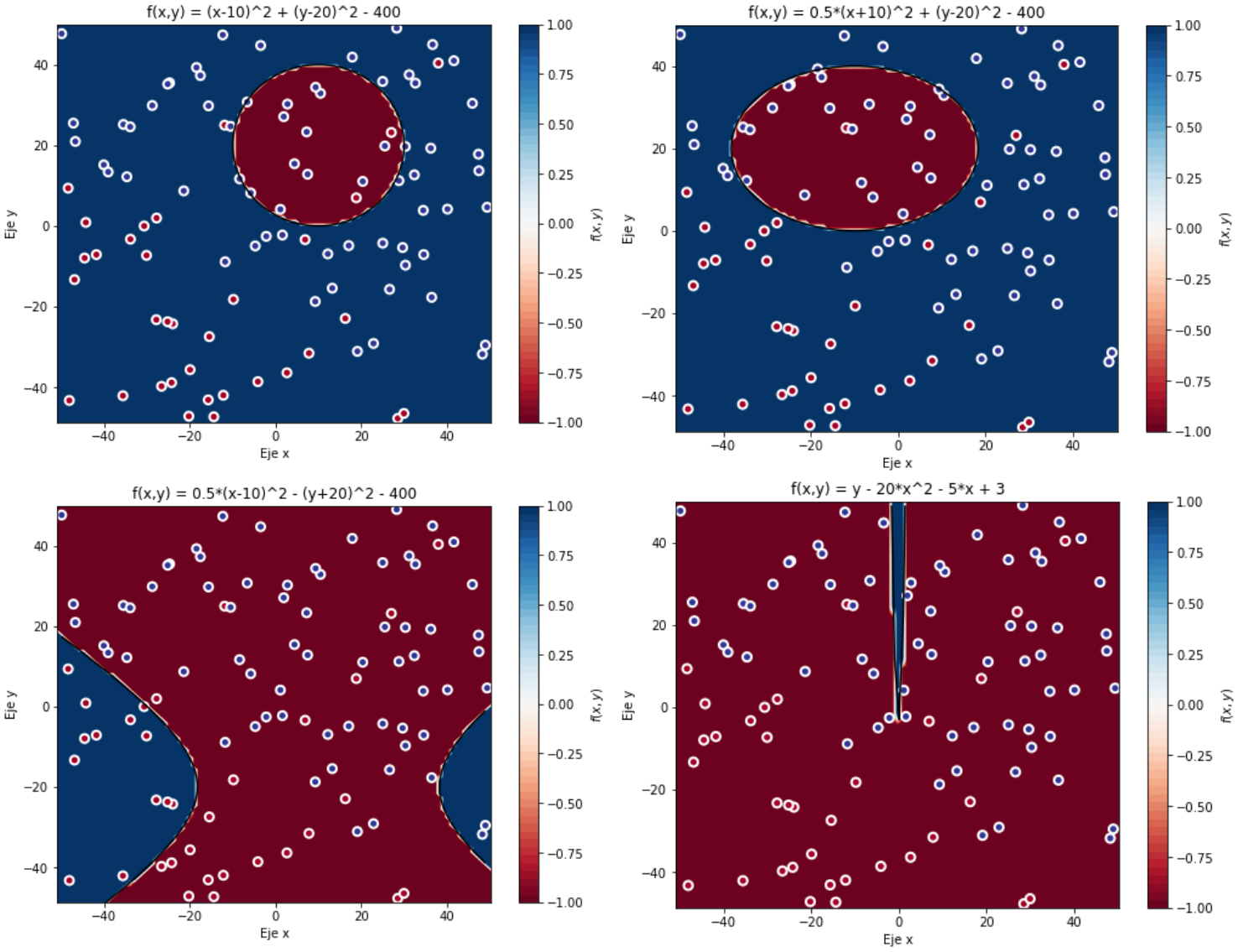
b) Modifique de forma aleatoria un 10 % de las etiquetas positivas y otro 10% de las negativas y guarde los puntos con sus nuevas etiquetas. Dibuje de nuevo la gráfica anterior (ahora habrá puntos mal clasificados respecto de la recta).



c) 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 1.2.b. junto con cada una de las gráficas de cada una de las funciones. Comparar las regiones positivas y negativas de estas nuevas funciones con las obtenidas en el caso de la recta. Argumente si estas funciones más complejas son mejores clasificadores que la función lineal. Observe las gráficas y diga qué consecuencias extrae sobre la influencia del proceso de modificación de etiquetas en el proceso de aprendizaje. Explicar el razonamiento.



Como es de esperar, cuatro funciones arbitrarias no son clasificadores óptimos para los datos, con o sin ruido. Además, evidencia que el hecho de que funciones más complejas no implican necesariamente ser mejores clasificadores que funciones lineales. Para un conjunto de datos como la muestra sobre la que estamos trabajando en este ejercicio, una recta siempre será un mejor separador (pues inicialmente los datos eran linealmente separables). La presencia de ruido en la muestra complica la estimación y hace que una función lineal ya no pueda separar perfectamente los datos (deja de ser linealmente separable), pero aun así sigue

siendo una buena opción como clasificador ya que es relativamente sencilla de calcular computacionalmente y aporta un error bastante reducido.

No nos interesa buscar una función extremadamente compleja que sea capaz de clasificar perfectamente la muestra ya que supondría un malgasto enorme de poder computacional para clasificar datos que por definición son irrelevantes o insignificantes y que no se tendrían que tener en cuenta para la estimación (ruido).

2. MODELOS LINEALES

2.a. 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 ítem 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.

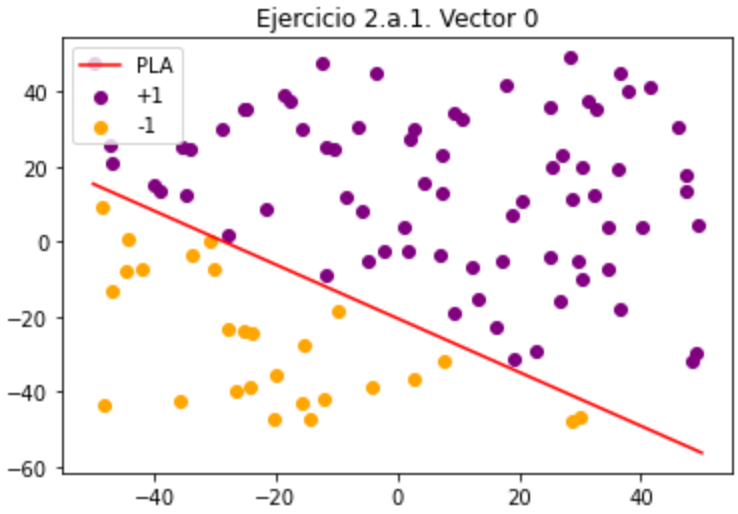
Se ha implementado el algoritmo Perceptron con un bucle *for* que itera hasta *max_iter* iteraciones (escogí 1000 iteraciones de máximo para garantizar que en el apartado con el conjunto de datos linealmente separable se llegase al w óptimo independientemente del punto de inicio) y mientras el vector w cambie.

Dentro de ese bucle se recorre el conjunto de datos y se actualiza w si se da que $sign(w^t x_i) \neq y_i$. Es decir, si la estimación respecto a las características de x_i no coincide con y_i. Si tras recorrer el conjunto de datos entero w no ha cambiado, significa que el algoritmo ha alcanzado el óptimo y termina.

Para más información, consultar el código donde se explica línea por línea el algoritmo.

1) **Ejecutar el algoritmo PLA con los datos simulados en el apartado 1.2.a. 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.**

a) Vector cero.



Valor de iteraciones necesario para converger: 74

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

Valor de iteraciones necesario para converger: 75

Valor de iteraciones necesario para converger: 83

Valor de iteraciones necesario para converger: 58

Valor de iteraciones necesario para converger: 68

Valor de iteraciones necesario para converger: 80

Valor de iteraciones necesario para converger: 273

Valor de iteraciones necesario para converger: 85

Valor de iteraciones necesario para converger: 57

Valor de iteraciones necesario para converger: 131

Valor de iteraciones necesario para converger: 78

Valor medio de iteraciones necesario para converger: 98.8

PLA no es dependiente del punto de inicio para encontrar la solución (como lo era, por ejemplo, gradiente descendente en la práctica anterior) ya que, para un conjunto de datos linealmente separable como el del apartado 1.2.a, el algoritmo por definición garantiza encontrar un vector w tal que $h(x_i) = y_i$ en tiempo finito. Es decir, nos garantiza encontrar la mejor función posible en un número finito de iteraciones.

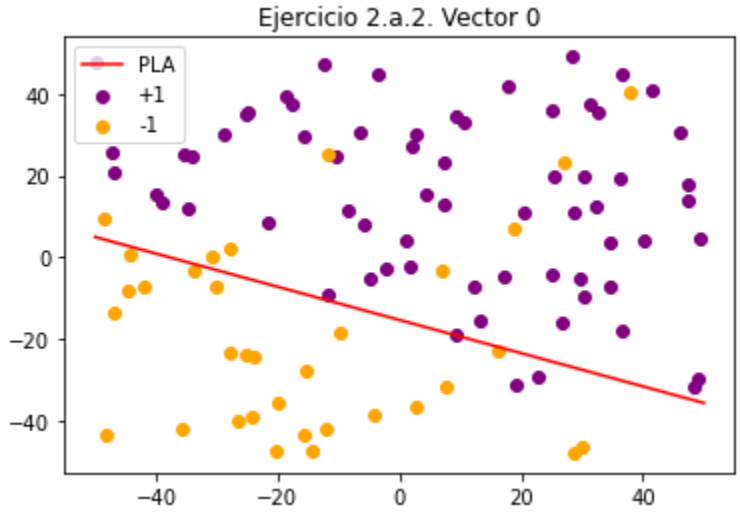
Sin embargo, escoger un punto inicial por encima de otro sí puede suponer una diferencia considerable en el número de iteraciones que necesita el algoritmo para alcanzar esa función perfecta. En el caso de los vectores con números aleatorios, tenemos desde 57 iteraciones para converger hasta 273. Por tanto tampoco podemos calificar la tarea de escoger el vector inicial como trivial.

En vez de aleatoriamente, podría ser más interesante (depende del conjunto de datos y del problema de clasificación en el que nos encontremos) usar otro algoritmo para escoger el vector inicial, como por ejemplo gradiente descendente con un número reducido de iteraciones (ya que nuestra intención es obtener un punto de inicio, no resolver el problema).

Nota: para poder usar gradiente descendente en clasificación el problema tendría que ser necesariamente de clasificación binaria y las dos etiquetas posibles deberían de tener el mismo peso (en este caso se cumple). Se comentará con más detalle en el ejercicio bonus.

2) **Hacer lo mismo que antes usando ahora los datos del apartado 1.2.b. ¿Observa algún comportamiento diferente? En caso afirmativo diga cuál y las razones para que ello ocurra.**

a) Vector cero.



Valor de iteraciones necesario para converger: 999

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

Valor de iteraciones necesario para converger: 999

Valor de iteraciones necesario para converger: 999

Valor de iteraciones necesario para converger: 999

Valor de iteraciones necesario para converger: 999

Valor de iteraciones necesario para converger: 999

Valor de iteraciones necesario para converger: 999

Valor de iteraciones necesario para converger: 999

Valor de iteraciones necesario para converger: 999

Valor de iteraciones necesario para converger: 999

Valor de iteraciones necesario para converger: 999

Valor medio de iteraciones necesario para converger: 999.0

Evidentemente hay un comportamiento diferente y es que independientemente del punto inicial PLA siempre alcanza el número máximo de iteraciones. Si analizamos el algoritmo, podemos deducir que esto se debe a que *w* siempre cambia después de recorrer los datos en cada iteración, y esto es porque, a la hora de recorrer los datos, siempre encuentra una etiqueta para la cual la estimación no coincide ($sign(w^t x_i) \neq y_i$).

Como la muestra utilizada en este apartado presenta ruido, los datos dejan de ser linealmente separables y PLA ya no puede garantizar encontrar la solución óptima. Es por esto que tenemos que limitar el número de iteraciones de PLA, para que no cicle indefinidamente buscando una solución lineal óptima que no existe.

Todo sea dicho, pasado un número suficiente de iteraciones, PLA encontrará una función que bien no será la óptima pero sí bastante decente para separar los datos (como vemos en la gráfica dada para el vector inicial 0). Y como se ha explicado antes, no tiene sentido buscar intencionadamente la clasificación “correcta” del ruido.

2.b. 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 $\chi = [0, 2] \times [0, 2]$ con probabilidad uniforme de elegir cada $x \in \chi$. Elegir una línea en el plano que pase por χ 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 de χ y calcular la línea que pasa por ambos.

EXPERIMENTO: Seleccione $N = 100$ puntos aleatorios $\{x_n\}$ de χ y evalúe las respuestas $\{y_n\}$ de todos ellos respecto de la frontera elegida. Ejecute Regresión Logística para encontrar la función solución g y evalúe el error E_{out} usando para ello una nueva muestra grande de datos (> 999). Repita el experimento 100 veces, y

- Calcule el valor de E_{out} para el tamaño muestral $N = 100$.
- Calcule cuántas épocas tarda en converger en promedio RL para $N = 100$.

Implementación de Gradiente Descendente Estocástico (SGD) para RL

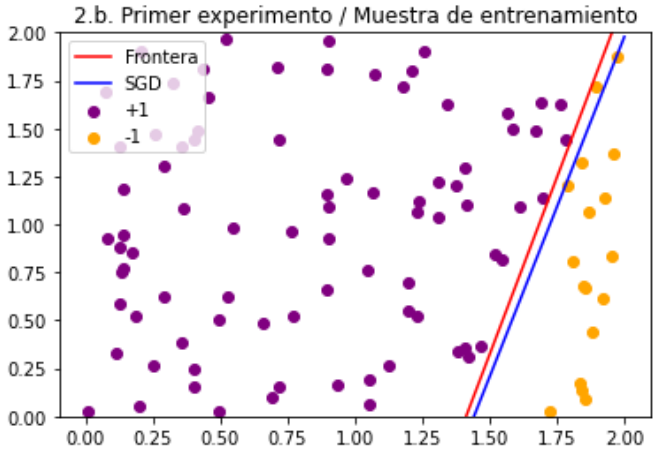
Gradiente Descendente Estocástico se ha implementado exactamente como en la práctica anterior pero con las siguientes particularidades:

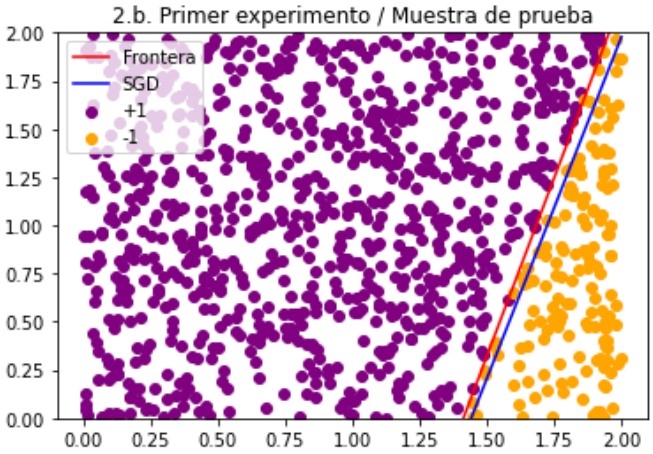
- ❏ Se han eliminado el número máximo de iteraciones y se ha establecido un nuevo criterio de parada $||w^{(t-1)} - w^{(t)}|| < 0.01$, siendo $w^{(t)}$ el vector de pesos al final de la época t y $w^{(t-1)}$el vector de pesos anterior a la misma época. Es decir, el algoritmo termina cuando el cambio del vector w tras una época t es menor a 0.01

- La función de error viene dada por $E_{in}(w) = \frac{1}{N}\sum_{i=0}^{N} ln(1 + e^{-y_i w^T x_i})$, y el gradiente por $\nabla_w E_{in}(w) = \frac{1}{N}\sum_{i=0}^{N} -y_i x_i \sigma(-y_i w^T x_i)$, siendo σ la función logística o sigmoide $\sigma(x) = \frac{1}{1+e^{-x}}$.

Además, se ha establecido una tasa de aprendizaje $\eta = 0.01$ como indicaban las condiciones del ejercicio y un tamaño por minibatch de 1 (pues tras varias pruebas era el único tamaño para el que se obtenía una solución decente).

Resultados para el primer experimento





Número de épocas: 462

E_{in} =0.12075158971373538

E_{out} =0.1176312337141598

En este caso, el error de salida resulta ser superior pero por mera casualidad a la hora de generar la muestra de prueba.

Resultados para los 100 experimentos

Número de épocas promedio: 420.58

$E_{in}\, promedio$ = 0.11873932056873646

$E_{out}\, promedio$ = 0.13180997098544825

Comentario

Gradiente descendente estocástico prueba ser, una vez más, un algoritmo potente para estimar una función lineal que se adapta bien al problema de regresión logística, como podemos ver en las gráficas del primer experimento y los errores promedios del 10%. En la mayoría de casos encuentra una función lineal cuya recta es bastante cercana a la frontera y separa los datos casi perfectamente.

Sin embargo (aun siendo consciente de que se escogió una función determinista para el ejercicio por motivos académicos) veo relevante mencionar que utilizar regresión logística para una f determinista no es, lógicamente, la mejor opción. Sería mucho más interesante utilizar PLA, pues es una muestra linealmente separable y el algoritmo sería capaz de encontrar la mejor función posible en un número de épocas bastante inferior al de SGD dado un vector inicial adecuado.

3. BONUS

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.

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

La clasificación de dígitos se planteará como un problema de clasificación binaria donde, como indica el ejercicio, se tendrá solo en cuenta las características de intensidad promedio y simetría, y donde la etiqueta *-1* corresponderá al dígito 4 y la etiqueta *+1* corresponderá al dígito 8.

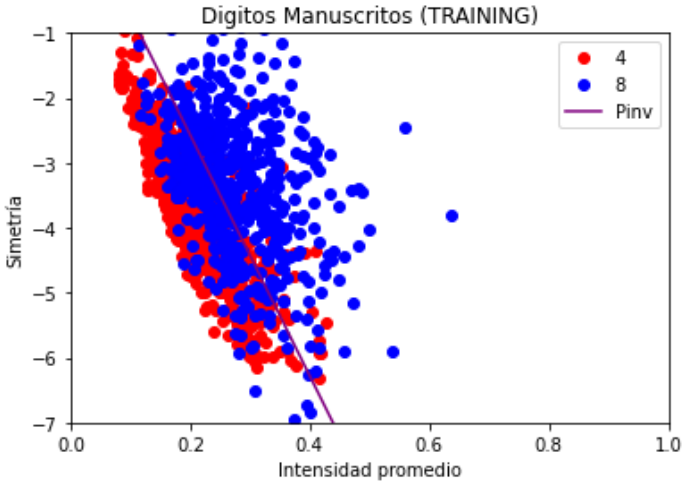
3.2. Usar un modelo de Regresión Lineal y aplicar PLA-Pocket como mejora.

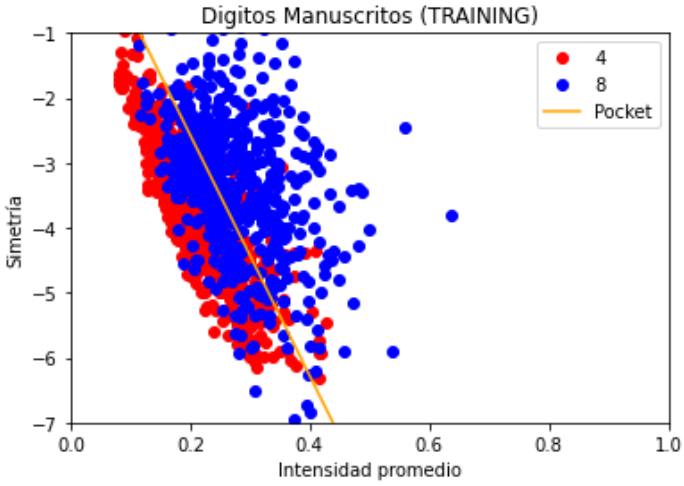
PLA-Pocket se ha implementado con un bucle *for* que itera hasta un máximo establecido de iteraciones. Dentro del bucle se llama cada vez a PLA (una sola iteración) para calcular un nuevo vector de pesos *w* y se compara con el mejor *w* encontrado hasta el momento usando el error de clasificación (para más información, consultar el código donde se explica línea por línea la implementación).

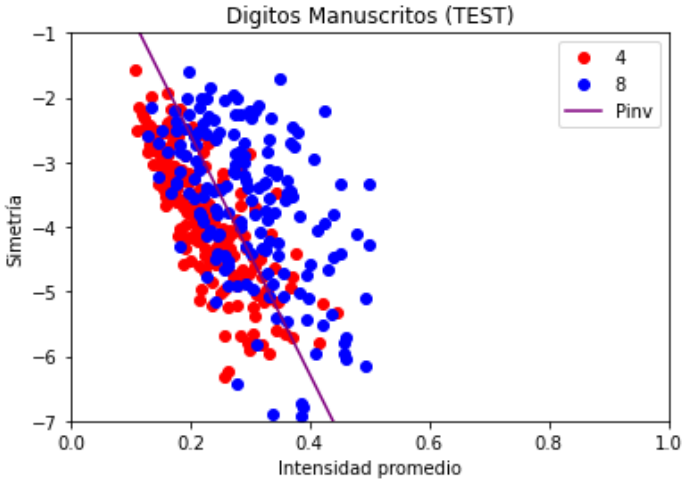
Para el modelo de Regresión Lineal he utilizado el algoritmo de la pseudoinversa con tal de simplificar la comparación (pues no necesito ajustar parámetros de entrada).

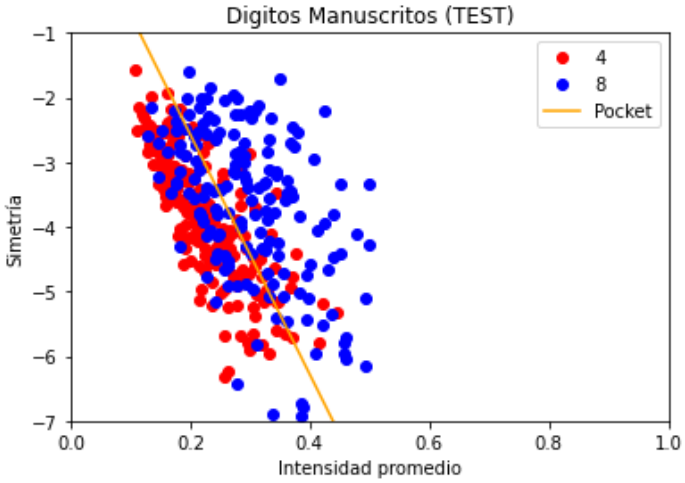
Responder a las siguientes cuestiones.

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).**

<u>Pseudoinversa</u>

E_{in} =0.22780569514237856

E_{test} =0.25136612021857924

Pocket

E_{in} =0.22780569514237856

E_{test} =0.25136612021857924

Comentario de a) y c).

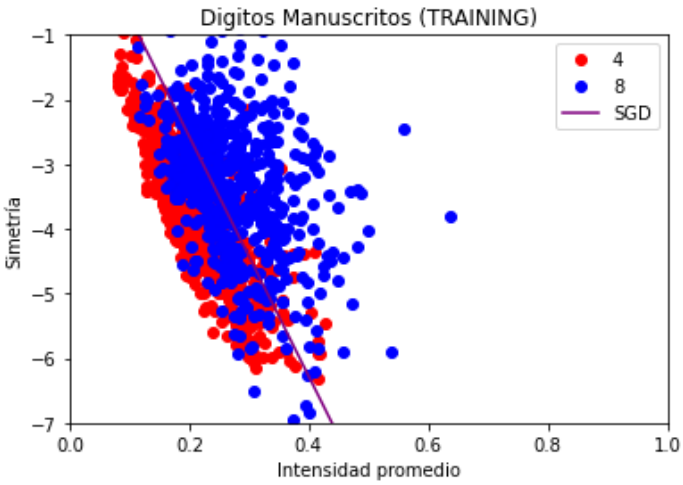
Como podemos observar, los resultados son exactamente iguales. Dada esta muestra de datos no linealmente separable, obtenemos la misma estimación lineal en ambos algoritmos con un error de clasificación del 22-25%. PLA-Pocket, empezando con el vector de pesos w dado por la pseudoinversa, no es capaz de mejorar la estimación.

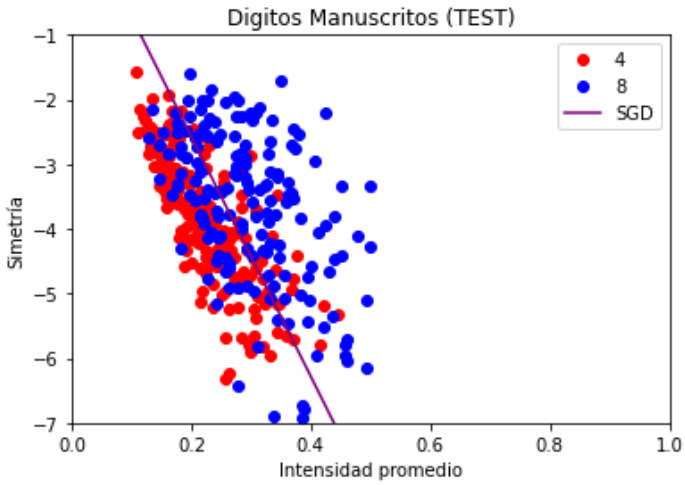
¿Es entonces la pseudoinversa simplemente un algoritmo superior no solo para regresión lineal sino también para clasificación? No, y son varios los motivos.

Para empezar, tenemos el problema de coste computacional que mencioné en la práctica anterior. La pseudoinversa es una solución analítica y directa que rinde bien en este ejemplo académico porque el vector de características es de tan solo dos elementos (sin incluir el término independiente), pero si fuésemos a un problema real donde el vector de características podría ser desde cientos hasta miles de elementos, su uso sería inviable. La pseudoinversa tiene una complejidad computacional de n^3, siendo n el número de características. Para tan solo una imagen pequeña de 100x100 píxeles, estaríamos hablando de una complejidad de (100*100)³ = 1000000000000.

Sabiendo entonces que no es una comparación justa, ¿qué pasaría si probásemos en su lugar el gradiente descendente estocástico? Con un número máximo de iteraciones de 1000 y un tamaño de minibatch de 24, que eran los parámetros para los que nos había rendido bien en la práctica anterior en un tiempo más que aceptable de ejecución.

Ocurre lo siguiente:





E_{in} =0.22780569514237856

E_{test} =0.25136612021857924

Obtenemos los mismos errores que la pseudoinversa. ¿Implica esto que podamos prescindir de algoritmos como Pocket y utilizar modelos de Regresión Lineal para cualquier problema de clasificación? No.

En este ejercicio hemos podido hacer uso de un modelo de Regresión Lineal con sus correspondientes algoritmos porque hemos adaptado el problema a ello. Lo hemos convertido en un problema de clasificación binaria (4 y 8) donde las dos etiquetas posibles tienen el mismo peso (-1 y 1). Si no se dieran esas condiciones no podríamos utilizar Regresión Lineal y tendríamos que hacer uso necesariamente de un algoritmo como PLA. Un ejemplo de un problema de clasificación que no podríamos resolver con un modelo de Regresión Lineal sería el problema de clasificación de dígitos real, donde tenemos 10 etiquetas posibles (0, 1, 2, 3, 4, 5, 6, 7, 8 y 9).

Pocket es costoso ya que tenemos que calcular el error de clasificación para cada vector de pesos w nuevo con tal de comparar con el mejor w encontrado hasta el momento, pero nos garantiza encontrar una solución buena tras el suficiente número de iteraciones y puede ser nuestra única opción en caso de encontrarnos con una muestra no linealmente separable donde no podemos utilizar Regresión Lineal.

c) Obtener cotas sobre el verdadero valor de E_{test}. Pueden calcularse dos cotas, una basada en E_{in} y otra basada en E_{test}. Usar una tolerancia de $\delta = 0.05$. ¿Qué cota es mejor?

Dado E_{in} y el E_{test} (que son el mismo independientemente del algoritmo utilizado), calculamos la cota según la fórmula

$$E_{out}(h) \leq E_{in}(h) + \sqrt{\frac{1}{2N} log \frac{2}{\delta}}$$

Para $E_{in}(h)$, tenemos que $E_{out}(h) \leq 0.2671090905470057$

Para $E_{test}(h)$, tenemos que $E_{out}(h) \leq 0.2906695156232064$

La cota basada en E_{in} es claramente mejor y es coherente ya que es una muestra de un tamaño N mayor que la de test. Un N mayor reduce el sumando $\sqrt{\frac{1}{2N} log \frac{2}{\delta}}$ y por tanto reduce el intervalo.