Trabajo 2: Programación

Curso 2019/2020

Aprendizaje Automático

SOFÍA ALMEIDA BRUNO
sofialmeida@correo.ugr.es

Índice

1. Ejercicio sobre la búsqueda complejidad de H y el ruido

El código correspondiente a este ejercicio se encuentra en el archivo `ej1.py`.

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

Para dibujar las gráficas con la nube de puntos se ha implementado la función `draw_points` que toma como parámetro el vector N de puntos.

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

Llamando a la función `simula_unif(N, dim, rango)` con los parámetros correspondientes, se generó el conjunto de puntos que podemos observar en la Figura 1. Notamos que, efectivamente, los puntos se han generado uniformemente en el rango.

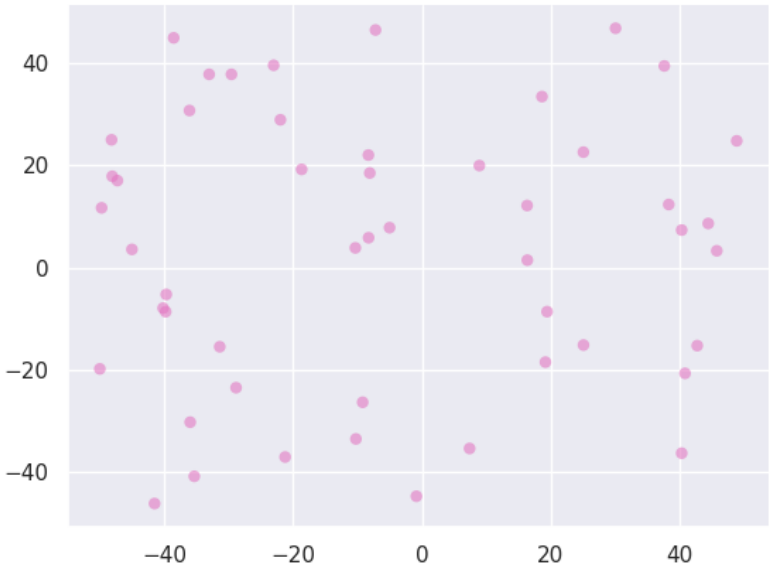


Figura 1: Puntos generados con `simula_unif`.

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

Ejecutando la función `simula_gaus` con los parámetros pedidos se obtiene el conjunto de puntos mostrado en la Figura 2. En este caso los puntos se agrupan en torno al $(0, 0)$, ya que la función usada para generarlos fue una gaussiana de media 0.

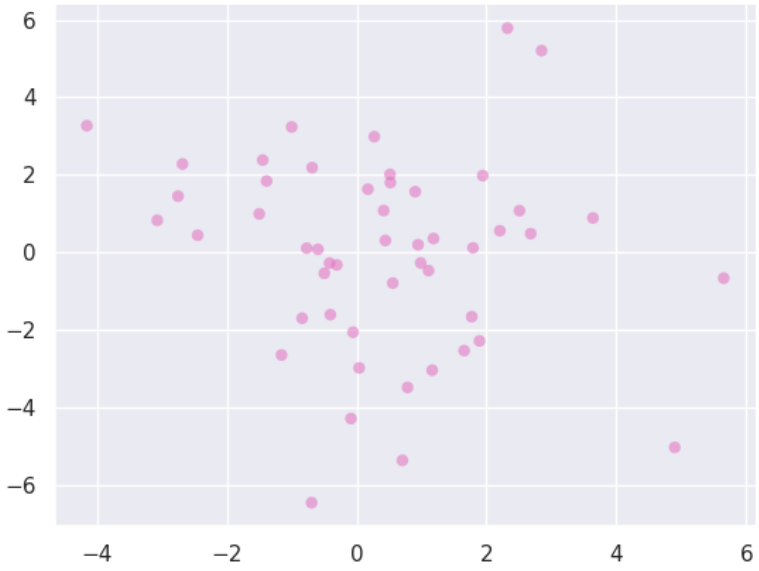


Figura 2: Puntos generados con `simula_gaus`.

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])` 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()`.

1.2.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 generan los puntos como en el caso anterior, en este caso tomamos una muestra de 100 puntos en el mismo rango. Se les asigna su clase mediante la función `f`, que toma los parámetros `a, b`, obtenidos al ejecutar `simula_recta`. Se ha utilizado la función `plot_line` para generar la gráfica.

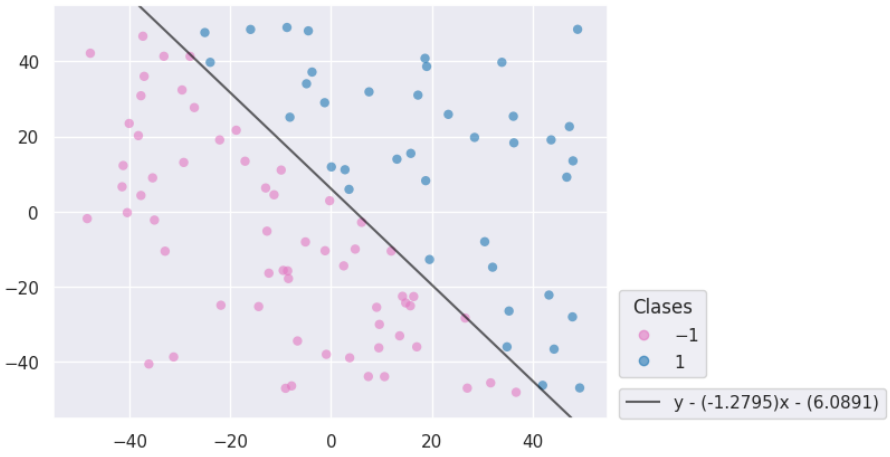


Figura 3: Puntos generados con `simula_unif`, clase asignada a partir del signo de la recta.

En la Figura 3 vemos los puntos generados, coloreados según su clase y notamos cómo la recta es una frontera perfecta entre ambas. Esto es debido a que las etiquetas se asignaron en función al signo[1] obtenido por la función que define la recta.

1.2.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).

En la función `modify_classes` modificamos un 10 % de las etiquetas positivas y un 10 % de las etiquetas negativas. Dibujamos la gráfica resultante, que podemos ver en la Figura 4. La gráfica es similar a la del apartado anterior, con la diferencia de que esta vez la recta no separa correctamente a todos los puntos, al haber añadido ruido hay puntos que quedan mal clasificados.

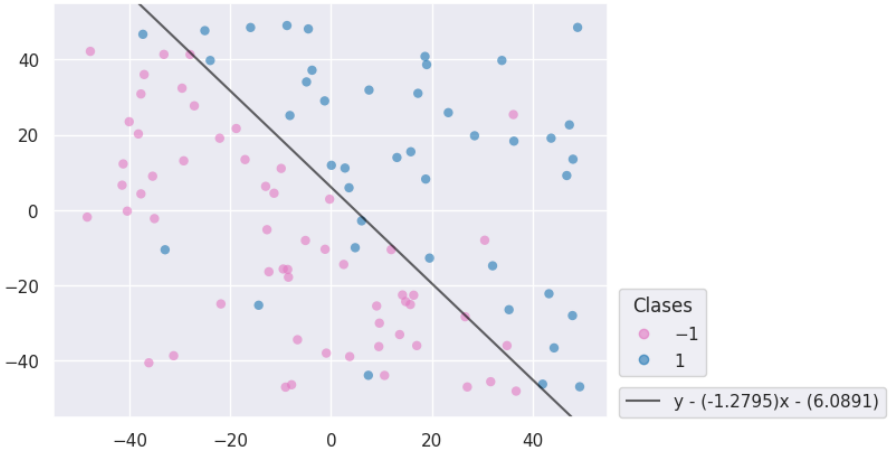


Figura 4: Puntos generados con `simula_unif`, clase asignada a partir del signo de la recta y posterior adición de ruido al 10 % de las etiquetas de cada clase.

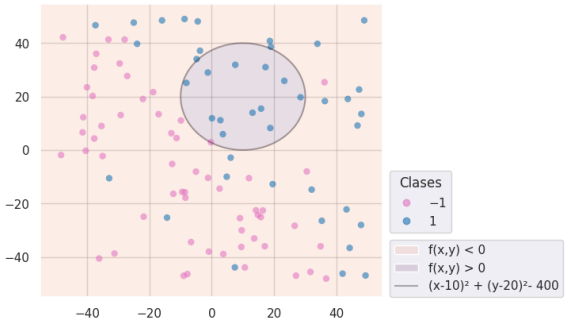
[1] De aquí en adelante se considerará que signo(0) = 1 a la hora de asignar las clases.

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

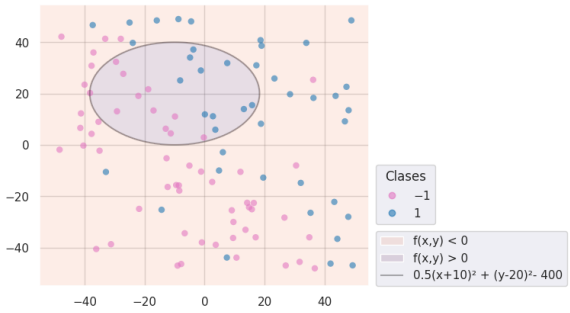
- $f_1(x, y) = (x - 10)^2 + (y - 20)^2 - 400$
- $f_2(x, y) = 0,5(x + 10)^2 + (y - 20)^2 - 400$
- $f_3(x, y) = 0,5(x - 10)^2 - (y + 20)^2 - 400$
- $f_4(x, y) = y - 20x^2 - 5x + 3$

Visualizar el etiquetado generado en 2b junto con cada una de las gráficas de cada una de las funciones. Comparar las 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? Observe las gráficas y diga que consecuencias extrae sobre la influencia del proceso de modificación de etiquetas en el proceso de aprendizaje Explicar el razonamiento.

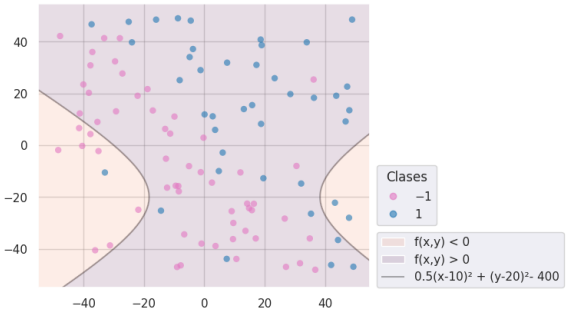
La Figura 5 muestra las fronteras de clasificación al usar las funciones dadas en el enunciado para definirlas. Para generarlas se usó la función `plot`. A simple vista no parece que las nuevas funciones se ajusten correctamente a las clases dadas. La función f_1 clasifica la mayor parte de los puntos como -1, cuando realmente pertenecen a ambas clases, y solo un pequeño grupo como 1, estos, exceptuando los puntos ruidosos, están bien clasificados. Los puntos de la clase -1 quedan, salvo ruido, bien clasificados. La función f_2 es parecida a f_1, aunque en este caso la elipse es mayor y ahora recoge a puntos de las dos clases (aproximadamente la mitad de ellos quedarán mal clasificados). Al resto de puntos se les asigna clase -1, clasificando correctamente aproximadamente la mitad de los puntos. Las funciones f_3 y f_4, al contrario, asignan a la mayor parte de los puntos la etiqueta 1. En el caso de f_4 es más exagerado, dejando con clase 1 a puntos que eran de la clase contraria prácticamente en su totalidad, aunque casi todos los puntos de la clase 1 sí que tienen su clase correcta. La función f_3 genera unas regiones en las que asignará clase -1. Una de ellas (la derecha) asigna incorrectamente etiqueta -1 a puntos de la clase 1. La otra región en que asigna clase -1 (la izquierda) está formada, en su mayoría, por puntos de esta clase.



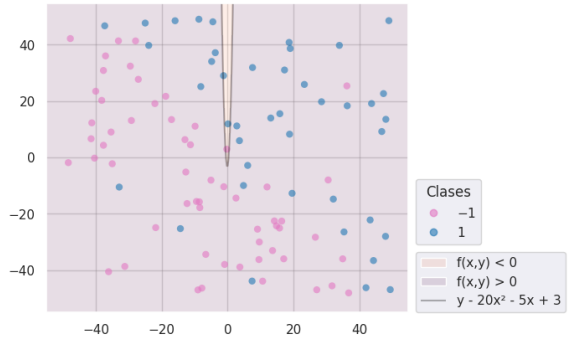
(a) $f_1(x,y) = (x-10)^2 + (y-20)^2 - 400$.



(b) $f_2(x,y) = 0,5(x+10)^2 + (y-20)^2 - 400$.



(c) $f_3(x,y) = 0,5(x-10)^2 - (y+20)^2 - 400$.



(d) $f_4(x,y) = y - 20x^2 - 5x + 3$.

Figura 5: Fronteras de clasificación dadas por f_i.

A la vista de las gráficas, observamos imprecisión en las clasificaciones dadas por estas regiones, sin embargo, para realizar una comparación más exacta entre ellas podemos hacer uso de alguna métrica. Por un lado, utilizaremos el *accuracy*, que promedia la cantidad de elementos bien clasificados,

$$accuracy = \frac{TP + TN}{P + N},$$

donde en este caso resulta indiferente qué clase es la positiva y cuál la negativa, ya que no representan nada. El *accuracy* será un valor entre 0 y 1, cuanto más cercano a 1, mejor clasificados estarán los datos.

Como las clases están ligeramente desbalanceadas, hay menos puntos de la clase 1 que de la clase -1, podemos utilizar también una métrica que tenga esto en cuenta. Elegimos *balanced accuracy*, esta es una media del *recall* de cada clase. El *recall* viene dado por

$$recall = \frac{TP}{TP + FN},$$

es una medida de cómo de bien se clasifican los ejemplos positivos, por ello nos interesa promediarlo para ambas clases. También es una medida que se encuentra entre 0 y 1, que cuanto más cercana 1 mejores resultados indica.

En la Tabla 1 se incluyen los valores obtenidos usando las métricas descritas de las funciones de este apartado, además de la función $f(x,y) = y - ax - b$ que determinó en un principio la clase de los puntos.

Tabla 1: *Accuracy* y *balanced accuracy* obtenido al clasificar con distintas funciones.

	Accuracy	*Balanced accuracy*
f	0.91	0.903
f_1	0.31	0.366
f_2	0.43	0.476
f_3	0.53	0.467
f_4	0.59	0.515

Como es natural, la función f es la que presenta mejores valores de ambas métricas. No llegan a ser un 1 exacto porque hemos generado un ruido de un 10 % en cada clase, que se corresponde con el 0.1 restante del *accuracy*. Las funciones más complejas f_i, como habíamos observado, no separan correctamente las clases fijadas, siendo el mejor *accuracy* de 0.59. En este caso, elegir modelos más complejos no ha proporcionado buenos resultados, al contrario que en la práctica anterior.

2. Modelos lineales

2.1. 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.

Las funciones implementadas en este apartado, en particular `ajusta_PLA`, se encuentran en el archivo `ej2.py`.

2.1.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.

En este caso, nos encontramos frente a un conjunto de puntos separable, el algoritmo PLA, aunque necesite muchas iteraciones, encontrará el hiperplano que separa a los conjuntos.

Tomando como punto inicial $w_0 = (0, 0, 0)^T$, las diez ejecuciones el algoritmo necesita **315 iteraciones** para encontrar el hiperplano que clasifica correctamente a todos los puntos. En la Figura 6 podemos ver el hiperplano generado por el algoritmo PLA al partir de w_0. Notamos cómo todos los puntos están bien clasificados y que el algoritmo es determinista, pues al empezar en el mismo punto, obtiene los mismos resultados.

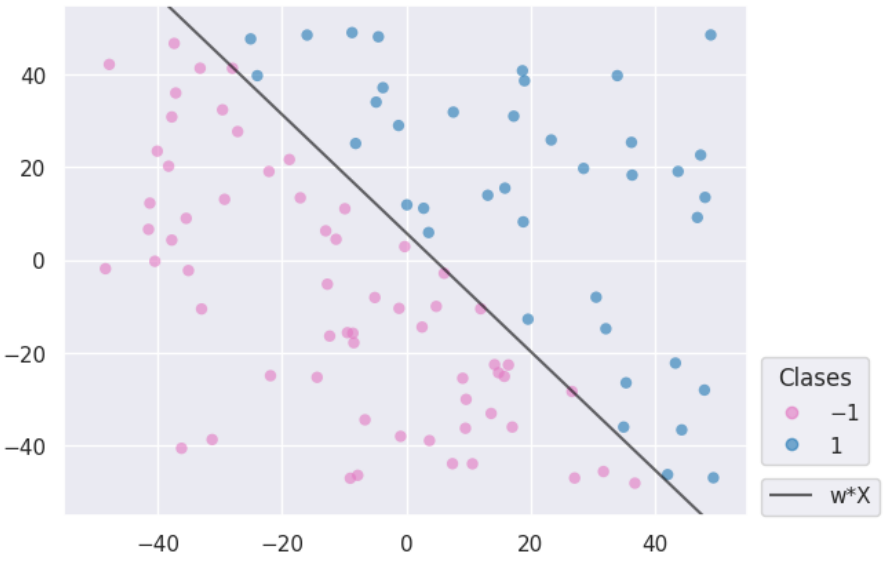


Figura 6: Hiperplano generado por el algoritmo PLA cuando el punto inicial fue $(0,0,0)^T$.

Si tomamos como puntos iniciales vectores aleatorios en $[0,1]$ el número de iteraciones necesarios varía. En el mejor caso necesita 172 iteraciones y en el peor 370. El **número medio de iteraciones requerido es de 291**, ligeramente inferior al caso anterior, sin embargo la desviación típica es de 61. Destacamos que cada punto inicial requiere un número de iteraciones variable, algunos mejores y otros peores que el caso anterior.

Como ocurre en general, los parámetros de un algoritmo afectan al desempeño del mismo. En este caso hemos observado cómo el punto inicial provoca un aumento o disminución en el número de iteraciones necesario para que el algoritmo encuentre los coeficientes del hiperplano que separa ambas clases. Sin embargo, notamos que, aunque el número de iteraciones varíe según el punto inicial, se logra encontrar el hiperplano que separa correctamente ambas clases. Obtenemos en todas las ocasiones un *accuracy* de 1.

2.1.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 cuál y las razones para que ello ocurra.

Repetimos la experimentación utilizando el conjunto de datos con ruido. En este caso las clases ya no son separables, los puntos ruidosos se encuentran rodeados por elementos de la clase contraria.

En esta ocasión, el algoritmo alcanza el número máximo de iteraciones sin llegar a conseguir un hiperplano que separe ambas clases. El *accuracy* obtenido varía según el punto inicial, aunque para todos los puntos iniciales es cercano a 0.9, pues solo tenemos un 10 % de error. En la Figura 7 observamos el hiperplano generado cuando el punto inicial fue $(0,0,0)^T$. Notamos como los puntos ruidosos, además de los cercanos a la recta, quedan mal clasificados.

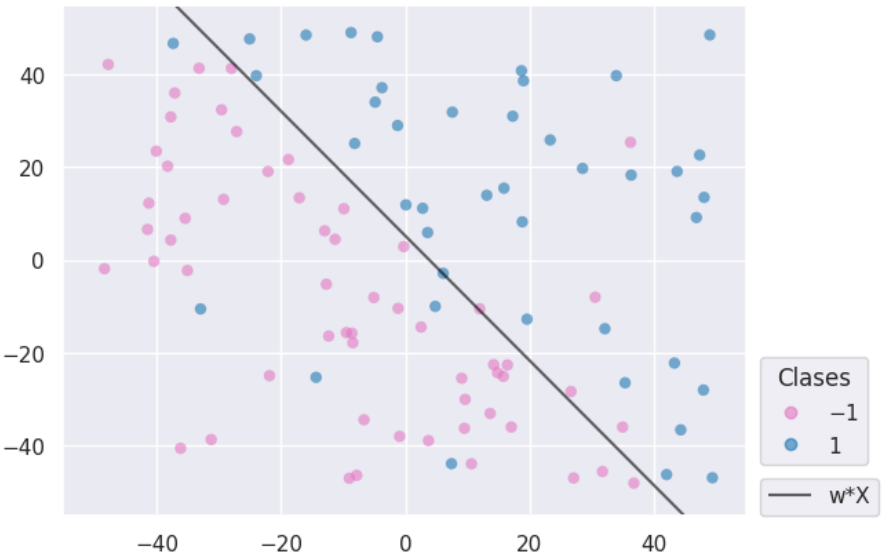


Figura 7: Hiperplano generado por el algoritmo PLA partiendo del punto $(0,0,0)^T$.

2.2. 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.

2.2.a. Implementar Regresión Logística (RL) con Gradiente Descendente Estocástico (SGD).

Esta función se encuentra implementada en la función `sgdRL`, cumpliendo las condiciones dadas. Para crear la muestra y la función se han utilizado las mismas funciones que en los apartados anteriores.

2.2.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).

Tras crear la muestra y etiquetarla, se utiliza la función `sgdRL` para encontrar el w que determinará los coeficientes de la recta que buscamos. En la Figura 8 observamos representados los puntos de la muestra, coloreados según su etiqueta junto a la recta obtenida. Vemos que la recta clasifica bien los puntos dados, el E_{in} obtenido es pequeño 0.1356.

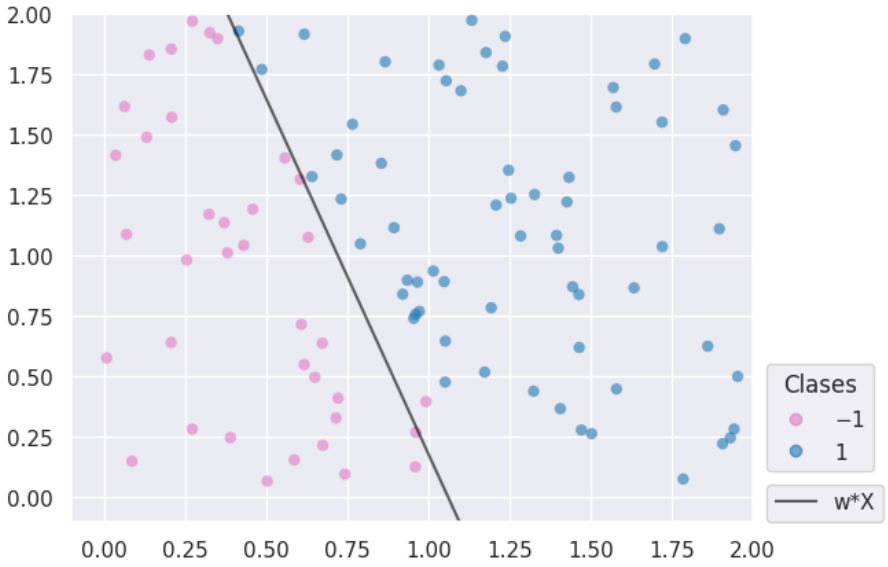


Figura 8: Puntos de la muestra y recta obtenida mediante SGD (RL).

Creamos un nuevo conjunto formado por 1000 elementos, que utilizaremos para estimar el error fuera de la muestra, y los etiquetamos. Calculamos el error obtenido al clasificar estos elementos con la recta conseguida con SGD, el $E_{out} = 0.1229$, ligeramente inferior. Sin embargo, en la Figura 9 vemos cómo ya la recta no se ajusta exactamente a los datos, los puntos en la frontera de las clases quedan mal clasificados. Esto se debe a que regresión logística no busca clasificar los datos, sino asignarles su probabilidad.

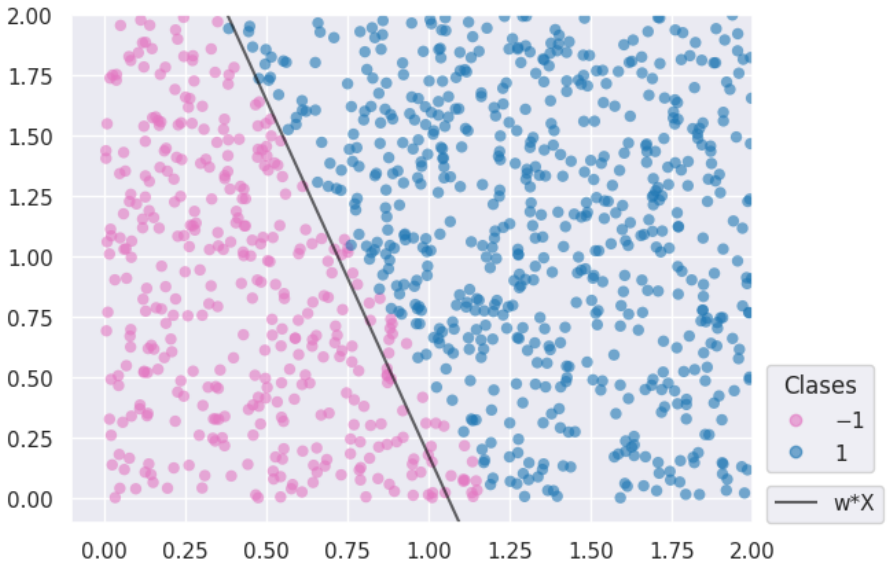


Figura 9: Nuevos puntos y recta obtenida mediante SGD (RL).