UNIVERSIDAD DE GRANADA

ETSIIT
Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Práctica 2: Complejidad $\mathcal{H}$ y Modelos Lineales

Aprendizaje Automático

Abril-Mayo, 2022

Autor:
Lugli, Valentino Glauco · YB0819879

Índice

0. Notas sobre la implementación

- La práctica fue realizada completamente en Python haciendo uso del IDE "Spyder", este IDE permite tener "celdas" de código que se pueden ejecutar independientemente del código que tienen antes o después, es decir, es como una celda de código de un Colab Notebook: no es necesario ejecutar todo el programa entero, se puede ir paso a paso.
- Al igual que un Colab Notebook, se han de ejecutar las celdas en orden la primera vez que se carga el fichero para tener las funciones y variables de celdas anteriores en memoria. Una vez realizado esto se pueden ejecutar las celdas en cualquier orden.
- Las celdas se delimitan con un comentario de la forma "`#%%`" y pueden ejecutarse haciendo `Ctrl+Enter` en una celda resaltada o bien haciendo clic en el icono en la Figura 1 que está a la derecha del icono de "Play" que ejecuta el código entero secuencialmente.
- Las celdas están organizadas de manera que la primera celda abarca todas las funciones implementadas, y luego existe una celda por cada ejercicio, de esta manera solamente es necesario ejecutar la celda inicial, denominada la celda 0 y la celda del ejercicio que se desee ejecutar, el cual se encuentra apropiadamente identificada en el código.
- También se muestra donde comienzan y terminan las funciones de cada ejercicio para facilitar la legibilidad de la práctica.
- Se tiene una función auxiliar `stop()` que puede añadirse en cualquier lugar que se piense sea necesario para detener la ejecución.

Figura 1: Icono para ejecutar la celda seleccionada

1. Complejidad $\mathcal{H}$ y el ruido

1.1. Dibujando nubes de puntos simuladas

Para este ejercicio se han generado dos gráficas de nubes de puntos, para ello se ha utilizado las funciones provistas `simula_unif(N, dim, rango)` y `simula_gauss(N, dim, sigma)` que calculan una lista de `N` vectores de dimensión `dim` de números aleatorios con distribución uniforme sobre un `rango` y gaussiana sobre un `sigma` con media 0 respectivamente.

En particular para este ejercicio se ha utilizado `N=50`, `dim=2` y `rango=[-50, 50]` para la distribución uniforme, `sigma=[5,7]` para la gaussiana.

Una vez realizado esto, se obtienen las gráficas que se pueden apreciar en la Figura 2.

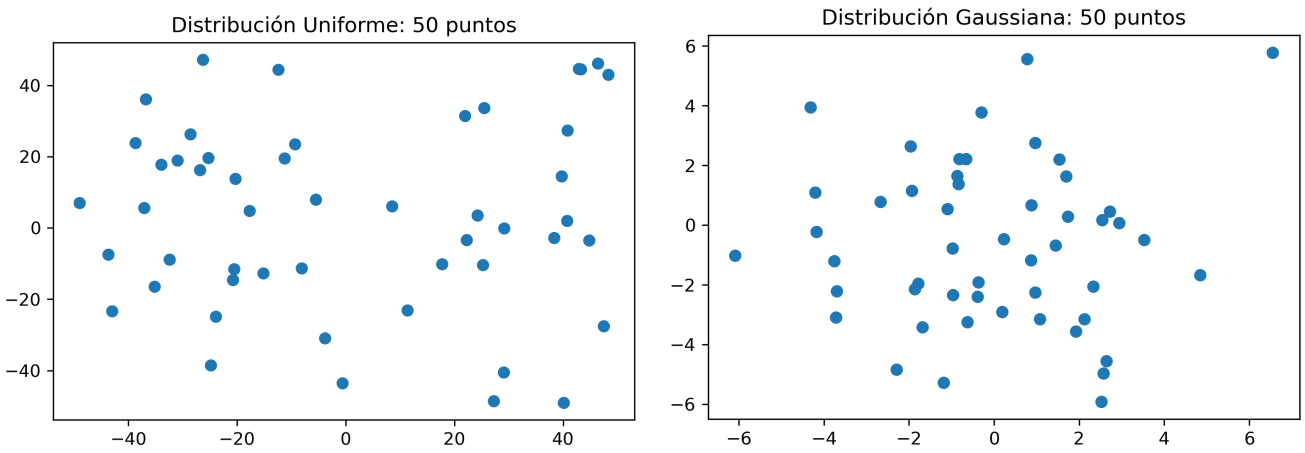

Figura 2: Comparación entre dos distribuciones de puntos

Pueden observarse las diferencias entre los tipos de distribuciones que se están utilizando, la distribución uniforme como su nombre lo indica genera puntos aleatorios pero uniformes por todo el rango, es decir que cualquier parte del rango tiene la misma probabilidad de obtener un valor, mientras que la distribución gaussiana genera más puntos cercanos a la media, que en este caso es 0, y mientras más los valores se alejan de la media la probabilidades de generar un punto allí se reducen drásticamente.

1.2. Influencia del ruido en selección de clases de funciones

Para este ejercicio, con el que se desea valorar cómo influye el ruido en las etiquetas a la elección de una clase de funciones, primero se generan datos idénticamente distribuidos utilizando la función de `simula_unif()` la cual será etiquetada utilizando el signo de la función $(f, y) = y - ax - b$, con a y b siendo una recta que generada con `simula_recta()`.

1.2.1. Generando datos etiquetados por una recta

Para la generación de los datos, de igual manera se utilizó la función `simula_unif()` para generar 100 valores, de dimensión 2 y en el rango de $[-50, 50]$.

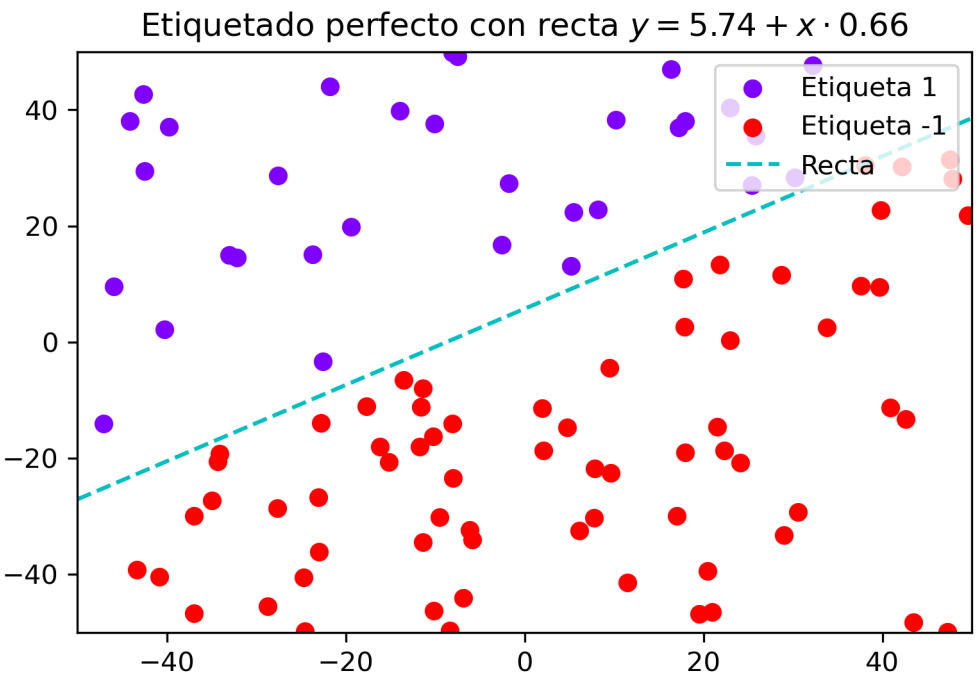

Figura 3

Una vez realizado esto, se obtuvieron los puntos a y b de una recta generada aleatoriamente por `simula_recta(intervalo)`, con el `intervalo=[11, 16]` elegido arbitrariamente pues no se indica ningún valor específico del mismo.

Realizado esto, se llama a la función `getTags(x, a, b)` que dado un vector de valores `x` y los puntos de la recta, etiqueta `x` basándose en el signo de $f(x, y)$, que lo realiza la función ya provista `f(x, y, a, b)`.

Finalmente, se utiliza la función auxiliar `scatterPlot()` para visualizar un resultado del etiquetado, que puede apreciarse en la Figura 3, cabe notar que la función `simula_recta()` obtendrá diferentes puntos en diferentes ejecuciones, por lo tanto la gráfica variará de la que se muestra ahora si se ejecuta nuevamente.

Se puede observar que la recta está dividiendo los datos perfectamente pues todos los puntos etiquetados como "1" están a un lado de la recta, al igual que los puntos con "-1".

1.2.2. Introduciendo un 10 % de ruido

A los datos anteriores se les añadió ruido haciendo uso de la función `addNoise(y, percent)` que toma el vector de etiquetas `y` y un porcentaje de error que se va a añadir, en este caso 10 %.

Internamente la función añade el ruido proporcionalmente a las etiquetas, es decir que se cambia un 10 % de los valores cada etiqueta, lo que era antes 1 se vuelve -1 y viceversa.

Se puede observar la diferencia con el ruido en la Figura 4 y se puede corroborar que se tiene exactamente 10 % de ruido pues teniendo 100 datos se puede contar que existen 10 etiquetas mal clasificadas, dependiendo de la recta generada, la proporción variará acordemente al número de

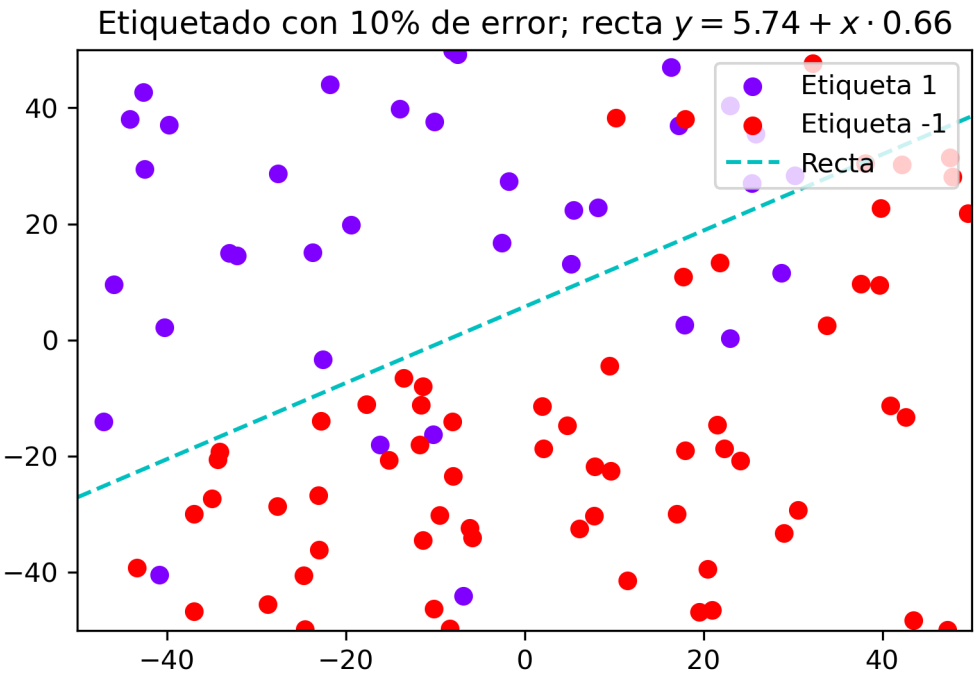

Figura 4

datos etiquetados de una manera u otra, para que se sumen 10 puntos mal clasificados en general.

1.2.3. Cambiando la función que define la frontera

Ahora, con los mismos datos obtenidos se van a aplicar las siguientes expresiones para definir la frontera.

$$f_1(x, y) = (x - 10)^2 + (y - 20)^2 - 400 \tag{1}$$
$$f_2(x, y) = 0{,}5 \cdot (x + 10)^2 + (y - 20)^2 - 400 \tag{2}$$
$$f_3(x, y) = 0{,}5 \cdot (x - 10)^2 - (y + 20)^2 - 400 \tag{3}$$
$$f_4(x, y) = y - 20x^2 - 5x + 3 \tag{4}$$

Aplicándola a los datos obtenidos en el apartado 1.2.2 con la función auxiliar `scatterPlotNonL()`, se puede observar la figura 5 para visualizar las áreas con signo negativo y positivo en comparación con las etiquetas.

Adicionalmente, para obtener una métrica del accuracy de estas funciones, se escribió la función `getFunAcc(x, y, fun)` que dado los datos `x`, sus etiquetas lineales `y` y una de las cuatro funciones como `fun`, compara la etiqueta que posee `y` con la que se obtendría al aplicar la función `fun` a los datos `x`, devolviendo la proporción de etiquetas que coinciden con `y`; los resultados se pueden observar en el la Tabla 1.

Visualmente se puede verificar los porcentajes de la clasificación obtenidos, que de por sí son bastante pobres, aunque resulte sorprendente que el mejor ajuste tiene un 64 % de accuracy, aunque

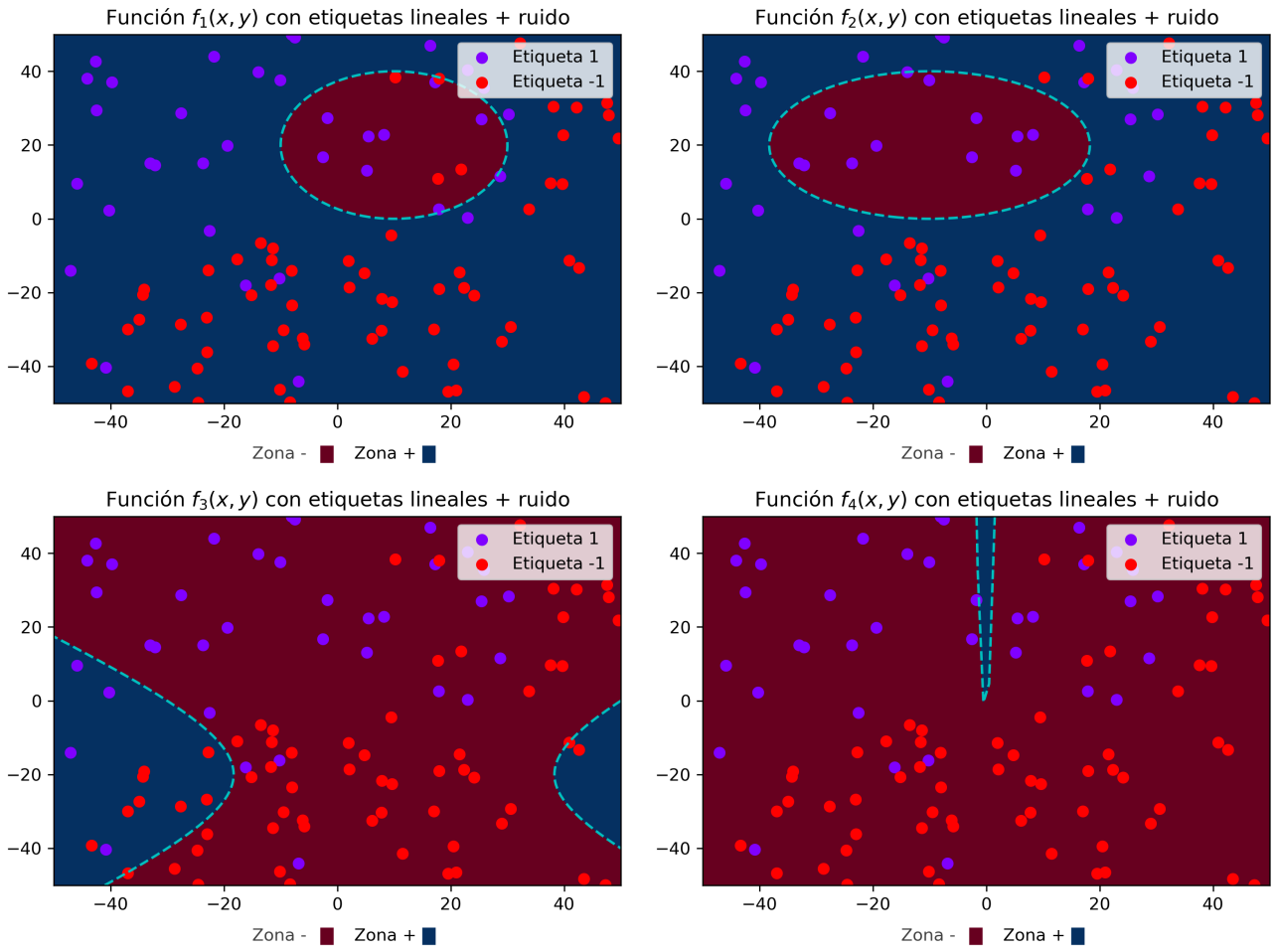

Figura 5: Datos lineales con frontera de decisión no lineal

Función	Accuracy
$f_1(x, y)$	32,00 %
$f_2(x, y)$	25,00 %
$f_3(x, y)$	59,00 %
$f_4(x, y)$	64,00 %

Tabla 1: Porcentaje de etiquetas correctamente acertadas por las funciones no lineales a las etiquetas lineales

esto sea más que nada influido porque todos los datos etiquetados con un valor negativo se encuentran dentro de la zona de valores negativos pero con la gráfica se puede ver que la curva que describe no está ni de lejos ajustada a la distribución de puntos, esto también es cierto para el resto de funciones. El error de 10 % de la recta no ha podido ser igualado ni mucho menos mejorado.

Por lo tanto aunque sean funciones más complejas no necesariamente son mejores clasificadores de los datos lineales, esto tiene sentido pues estas funciones complejas no han sido ni entrenadas con un algoritmo de aprendizaje para ajustar los datos, y aún así, realizar esto podría ser problemático pues un ajuste a los datos de muestra por funciones muy complejas puede obtener un error más pequeño en esos datos, pero al probar con otros datos fuera de la muestra, es probable que se obtenga un error mucho peor, como se sabe por teoría, en particular lo referente a la dimensión de Vapnik-Chervonenkis, mientras más se aumenta la dimensión el error dentro de la muestra tenderá más a 0 pero la diferencia con el error fuera de la muestra será mucho mayor.

Si, por otro lado, se utilizan dichas funciones no lineales para etiquetar los datos con un 10 % de error, se puede observar que nuevamente, no es posible mejorar el 10 % de error. Notar la tasa de aciertos en la Tabla 2 y las gráficas de la Figura 6.

Esto se realizó definiendo una función nueva `getTagsFF()` equivalente a `getTags()` adaptada para esta clase de funciones complejas.

Función	Accuracy
$f_1(x, y)$	90,00 %
$f_2(x, y)$	90,00 %
$f_3(x, y)$	90,00 %
$f_4(x, y)$	90,00 %

Tabla 2: Porcentaje de etiquetas no lineales con un 10 % de error correctamente acertadas por funciones no lineales

Esto tiene sentido, el hecho de añadirle ruido a los datos luego de generarlos con cualquier función por fuerza hace que esa función no los clasifique perfectamente, y posea esa cantidad de ruido imposible de mejorar, extrapolando esto al proceso de aprendizaje lo que se puede concluir es que dados datos reales, sea cual sea la mítica función f que los haya producido, están sujetos a ruido puesto que f no es una función determinista en la mayoría de los casos, y de hecho, esto es lo que permite en primer lugar el aprendizaje gracias a la desigualdad de Hoeffding; entonces siempre existirá un ruido en los datos por su naturaleza probabilística, además de ruido también por la complejidad de la clase de funciones sumado por errores cometidos al recabar los datos —aunque

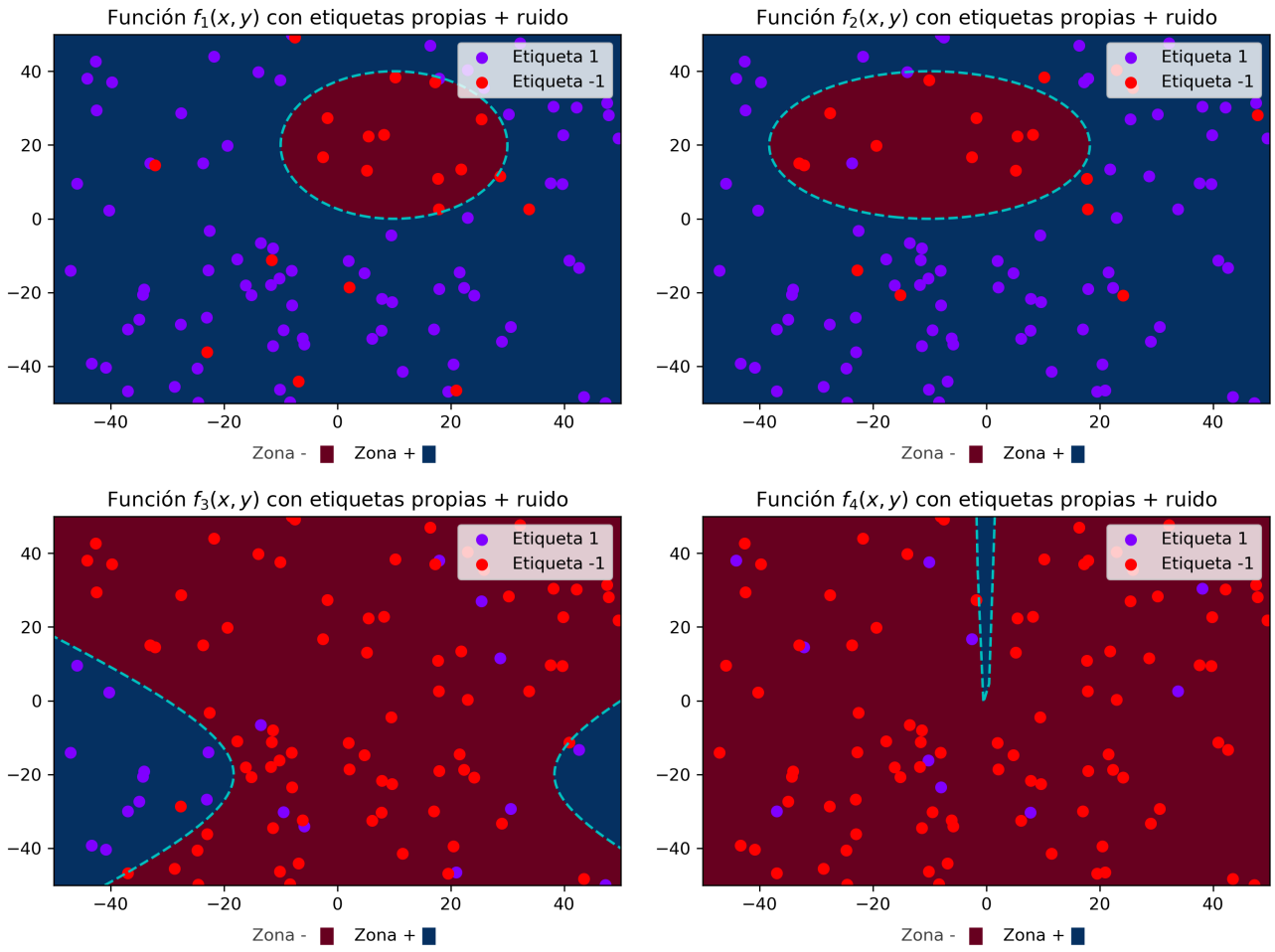

Figura 6: Datos no lineales + ruido con frontera de decisión no lineal

esto no aplica en este caso— y por lo tanto es esperable que exista siempre una cantidad de error en la clasificación, lo más importante entonces es poder reducir ese error lo más que se pueda.

2. Modelos lineales

2.1. Algoritmo Perceptrón / PLA

Para este ejercicio se implementó el algoritmo del Perceptrón y con él se ajustaron los datos generados en las secciones 1.2.1 y 1.2.2.

Concretamente, se implementó la función `ajusta_PLA(datos, label, max_iter, vini, pocket =False)` que toma `datos` y sus etiquetas, `label` con un límite de iteraciones `max_iter` y dado un vector de pesos iniciales `vini` implementa el algoritmo del Perceptrón para ajustar los datos a las etiquetas; el parámetro `pocket` indica si se utiliza la versión original o la versión Pocket, en este momento se describirá el algoritmo original, mientras que la versión Pocket será descrita en la Sección 3 pues es utilizada solamente en el bonus.

Dentro de la función, una vez inicializados los valores pertinentes para mantener los pesos y las iteraciones se entra en un bucle principal que va contando el número de iteraciones que se van a realizar; dentro se tiene otro bucle que recorrerá cada elemento de `datos` y de `label` correspondiente.

Dentro de este segundo bucle, para el valor actual del peso w y un x_i que es un elemento de `datos` y un y_i, la etiqueta correspondiente y perteneciente a `label`, si el signo de $w^T x_i$ no es el mismo de y_i, entonces se realiza la actualización de los pesos tal que $w' = w + y_i x_i$.

Se tiene un booleano que se asigna a `True` si al menos una vez se realiza la actualización de los pesos, en el momento que se realice un pase entero de los datos sin que haya una actualización, se finaliza el algoritmo; también si se llega al límite de iteraciones, lo que suceda primero.

Una vez terminado el bucle, se devuelven los últimos pesos `w` obtenidos y el número de iteraciones requerido para obtenerlos.

2.1.1. Ajuste a datos sin ruido

Utilizando los datos utilizados en la Sección 1.2.1 añadiéndole el parámetro de sesgo con `addBias()` —que le añade a la matriz de características una columna de 1 al principio— y en primer lugar con los pesos inicializados a 0, se llama a la función `ajusta_PLA()` con un máximo de 15000 iteraciones.

Una vez el algoritmo finaliza, se obtiene que para esta inicialización a cero, en 11800 iteraciones se obtuvo el ajuste perfecto a los datos con 100 % de accuracy, lo que corresponde a un 0 % de error de clasificación, haciendo uso de las funciones `accPLA()` y `errPLA()` —si bien una función es complementaria de la otra pues la métrica del error de PLA es una proporción de las etiquetas mal clasificadas, por lo que el accuracy es lo opuesto, la proporción de etiquetas bien clasificadas, se han mantenido separadas por comodidad— y los pesos finales $[-567; -84{,}0536; 120{,}9914]$, se puede observar en la Figura 7 el ajuste del hiperplano a los datos.

Ahora, se realiza otro experimento inicializando los pesos a valores aleatorios entre 0 y 1 por medio de la función `np.random.uniform(0, 1, 3)`, y se repite este experimento 10 veces; se puede observar

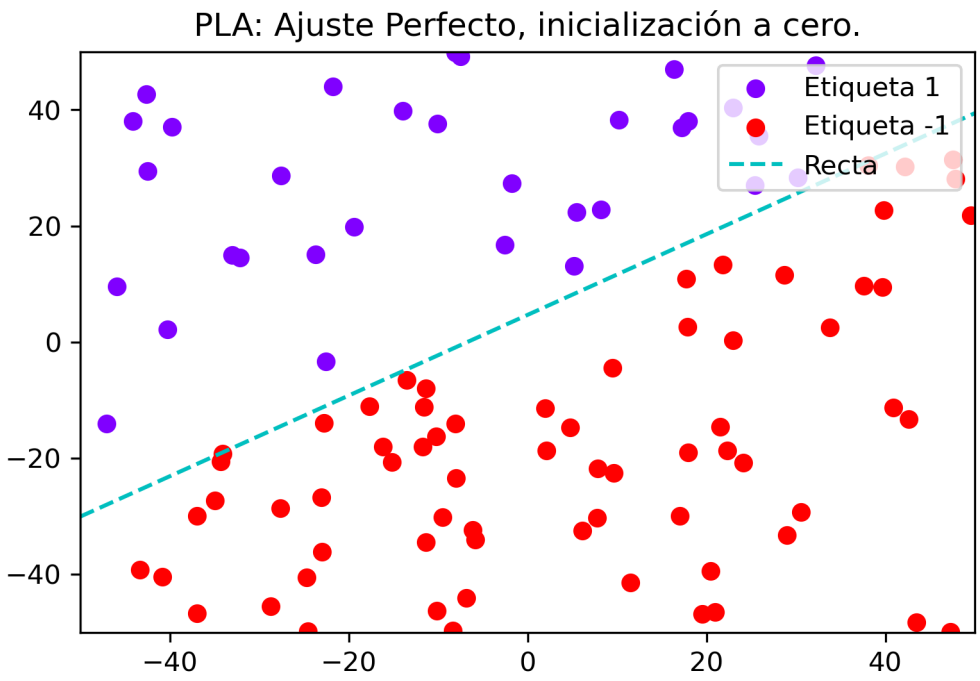

Figura 7: Ajustando PLA a los datos separables.

los resultados en la Tabla 3, con una media de iteraciones para converger de 10420. Se utilizó la métrica de accuracy por ser más intuitiva, además, al ser la complementaria de la métrica de error dentro de la muestra, la conversión es sencilla para obtener el error.

#	Pesos Iniciales	Pesos Finales	Accuracy	Iteraciones
0	[0,6124; 0,5967; 0,5300]	[−550,3876; −83,7799; 122,4597]	100.00 %	11300
1	[0,5886; 0,8170; 0,7558]	[−541,4114; −82,5440; 119,8936]	100.00 %	11000
2	[0,2062; 0,4983; 0,4964]	[−526,7938; −82,8589; 120,6036]	100.00 %	10700
3	[0,9331; 0,9901; 0,1162]	[−548,0669; −82,9928; 121,4270]	100.00 %	11100
4	[0,6537; 0,9713; 0,6994]	[−522,3463; −82,1961; 119,2507]	100.00 %	9500
5	[0,3438; 0,0701; 0,4418]	[−515,6562; −80,7432; 116,6910]	100.00 %	9700
6	[0,4999; 0,2868; 0,9043]	[−511,5001; −79,9473; 115,8883]	100.00 %	9300
7	[0,7989; 0,3745; 0,7997]	[−529,2011; −82,8027; 120,6157]	100.00 %	10200
8	[0,1147; 0,6052; 0,1279]	[−558,8853; −81,8364; 117,5692]	100.00 %	11800
9	[0,8020; 0,8202; 0,0284]	[−506,1980; −80,5364; 117,1008]	100.00 %	9600

Tabla 3: Ejecuciones de PLA con pesos iniciales aleatorios entre $[0, 1]$ para datos separables.

Con estos resultados se puede confirmar lo estudiado en teoría: que, dado un conjunto de datos linealmente separables, el PLA es capaz de converger y obtener un hiperplano que separa perfectamente los datos, como se puede observar claramente en la Figura 7 donde el hiperplano es muy similar a la recta original utilizada para etiquetar los datos, en la Figura 3.

Respecto al número de iteraciones, si se observa la Tabla 3, existen ejecuciones que tomaron significativamente menos iteraciones, por ejemplo, en la ejecución #6, convergió en 9300 iteraciones, o la ejecución #4 con 9500, a diferencia de otras donde se tardaron 2000 iteraciones más, esto mues-

tra claramente como los valores de inicialización influyen de una gran manera al funcionamiento de PLA, algo que se confirma nuevamente pues ya se conocía de teoría.

2.1.2. Ajuste a datos con ruido

Ahora se repite el experimento de la misma manera con los datos que poseen un 10 % de ruido de la Sección 1.2.2.

Nuevamente, primero se realiza un entrenamiento con los pesos inicializados a 0, una vez finalizado se obtiene el resultado que aparece en la Figura 8 obteniendo un 75 % de accuracy, lo que sería un error dentro de la muestra $E_{in} = 0{,}25$ y utilizando las 15000 iteraciones máximas se obtuvieron los pesos $[-206; -56{,}5586; 34{,}2793]$.

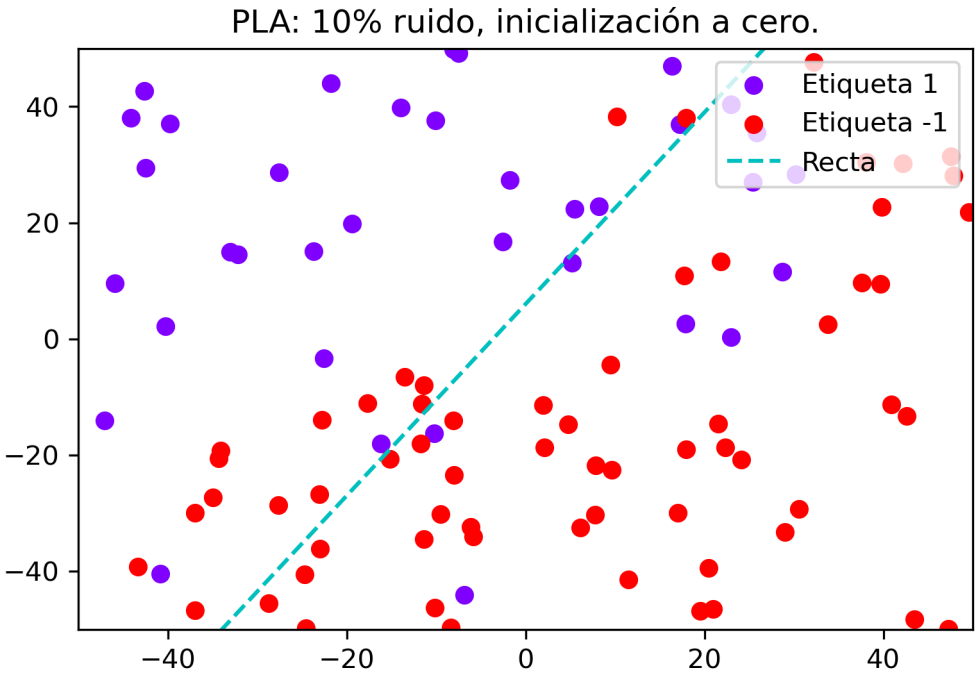

Figura 8

Luego, se realizan las 10 ejecuciones con valores de inicialización aleatorios entre 0 y 1, los resultados de estas ejecuciones se pueden observar en la Tabla 4, donde se obtiene de promedio un 81,5 % de accuracy.

Como se puede observar en ambos experimentos, la calidad de la solución obtenida ha empeorado con tan solo un 10 % de ruido añadido a los datos.

Se puede notar que se han utilizado en todo momento las 15000 iteraciones, ahora que se tiene un 10 % de ruido los datos no pueden ser separados con una recta y el algoritmo no converge, además de esto, por el funcionamiento del algoritmo, los pesos finales no serán siempre los mejores puesto que en ningún momento se compara un peso anterior con alguno nuevo obtenido, sino que simplemente se actualizan los pesos con lo calculado en la última iteración y dado que el algoritmo no posee "memoria" de los cambios que se han realizado antes, cuando se realice un ajuste a un

#	Pesos Iniciales	Pesos Finales	Accuracy	Iteraciones
0	[0,2417; 0,1163; 0,3511]	[−193,7583; −47,6483; 35,6849]	79.00 %	15000
1	[0,8536; 0,0993; 0,2996]	[−209,1464; −10,0341; 42,7290]	80.00 %	15000
2	[0,6953; 0,2682; 0,9502]	[−214,3047; −35,3309; 51,1001]	89.00 %	15000
3	[0,5577; 0,6877; 0,0981]	[−191,4423; −44,3328; 33,8396]	79.00 %	15000
4	[0,2219; 0,0057; 0,4625]	[−211,7781; −6,4267; 38,9441]	81.00 %	15000
5	[0,4919; 0,1117; 0,0855]	[−213,5081; −5,2952; 43,9170]	80.00 %	15000
6	[0,9254; 0,1590; 0,6428]	[−207,0746; −56,2762; 33,8552]	75.00 %	15000
7	[0,1812; 0,4305; 0,5408]	[−209,8188; −35,8217; 52,1202]	88.00 %	15000
8	[0,2979; 0,9703; 0,4671]	[−197,7021; −33,4071; 35,7427]	84.00 %	15000
9	[0,4741; 0,3577; 0,7157]	[−208,5259; −3,7335; 36,2946]	80.00 %	15000

Tabla 4: Ejecuciones de PLA con pesos iniciales aleatorios entre $[0, 1]$ para datos con ruido.

dato y su etiqueta, el nuevo valor de los pesos puede hacer que otros valores previamente clasificados bien dejen de serlo.

La diferencia entre inicializar con ceros y con valores aleatorios es que con los valores aleatorios se puede dar que en ocasiones se obtengan mejores pesos finales por la casualidad de que con esos valores iniciales y el máximo número de iteraciones se detenga el algoritmo cuando posea un valor de los pesos óptimo, en este caso, se observa que hubieron ejecuciones muy cercanas a llegar al teórico máximo de 90 % de accuracy, la cota máxima dado el 10 % de ruido de los datos.

Esto confirma nuevamente lo conocido por teoría, que el algoritmo PLA funciona muy bien solamente si los datos son linealmente separables, cuando esta condición no se cumple el algoritmo se ejecutará indefinidamente sin converger en ningún momento, y también, no se puede asegurar que el valor obtenido al final de la ejecución sea el mejor valor posible para los pesos que describen el hiperplano que divide los datos.

2.2. Regresión Logística

Para este ejercicio se utilizó el algoritmo de Gradiente Descendente Estocástica para minimizar la gradiente de la Regresión Logística, y con dicha gradiente se ajustaron datos generados de manera uniforme en 2 dimensiones.

2.2.1. Implementación

Concretamente, se reutilizó la función `sgd(x, y, wIni, lr, batchSize, maxIters, gradFun, wStop = True)` de la práctica anterior con ligeras modificaciones para permitir el nuevo criterio de parada.

Toma un vector de características `x`, con sus etiquetas `y`, el vector de pesos iniciales `wIni`, la tasa de aprendizaje `lr`, el tamaño de lote `batchSize`, el máximo de iteraciones `maxIters`, la función de gradiente `gradFund` y adicional, si se desea aplicar el nuevo criterio de parada con `wStop`.

Internamente, la función primero obtiene la cantidad de lotes necesarios dependiendo del tamaño de lote que se le envía como `batchSize`, luego comienza dentro el bucle principal, se almacenan

los pesos antiguos y luego se obtienen una permutación de índices del tamaño de los datos y con ellas se barajean de igual forma los datos y sus etiquetas.

Luego dentro de otro bucle, se recorren los lotes y se realiza la llamada a la función de gradiente con los datos que se encuentran dentro del lote actual y se realiza la actualización de los pesos ya conocida, es decir, $w' = w - \eta \cdot \nabla E_{in}(h(x), y)$.

Cuando se terminan de recorrer todos los lotes, lo que se denomina una época, si se tiene que los valores actuales de los pesos `w` y los anteriores `wOld` cumplen con la condición de que su norma vectorial es menor de 0.01, es decir, la expresión 5. Una vez se cumple la desigualdad, se detiene la ejecución.

$$||w - w_{old}|| < 0{,}01 \tag{5}$$

Si también se llega al máximo de iteraciones, se detiene la ejecución; el algoritmo para con la condición que suceda primero, una vez suceda alguna se retornan los pesos y las iteraciones utilizadas.

Para el gradiente, se implementó la función `gradRL(x, y, w)` que toma un vector de características `x`, sus etiquetas `y` y el vector de pesos `w` se calcula de manera vectorial el gradiente haciendo uso de las funciones de Numpy para acelerar dicho cálculo, es una adaptación de la expresión 6.

$$\nabla E_{in} = -\frac{1}{N}\sum_{n=1}^{N}\frac{y_n x_n}{1 + e^{y_n w^T x_n}} \tag{6}$$

Esto se realiza obteniendo la multiplicación del numerador para todos los datos de `x` e `y`; luego se obtiene el denominador de la misma manera.

Una vez se tienen estos dos vectores, se realiza una división elemento a elemento del numerador con el denominador y finalmente, se realiza la media por filas de la matriz obtenida para tener un vector columna, que será el vector con el que se actualizará el nuevo `w`.

Para el error, de igual forma se implementó la función `errRL(x, y, w)` que dentro calcula el error asociado a la Regresión Lineal, es decir, el error de Entropía Cruzada, la expresión 7. Se realizó una adaptación directa matricial con Numpy.

$$E_{in} = \frac{1}{N}\sum_{n=1}^{N}\ln(1 + e^{y_n w^T x_n}) \tag{7}$$

Finalmente, para poder evaluar con una métrica más intuitiva, se implementó la función `accRL(x, y, w)` que internamente utiliza la función sigmoide, 8, para obtener una probabilidad de que sea una etiqueta $+1$ o -1 dependiendo si el valor que se obtiene es $\geq 0{,}5$ puesto que la función está acotada entre $[0, 1]$.

$$h(x) = \frac{1}{1 + e^{-w^T x}} \tag{8}$$

Luego, se contabilizan las probabilidades y se obtiene una proporción de aciertos que se devuelve de la función.

2.2.2. Experimento

Una vez definidas las funciones, se realizó el experimento descrito, se obtienen 100 puntos del intervalo $\mathcal{X} = [0, 2] \times [0, 2]$ de dimensión 2 con probabilidad uniforme, y luego se obtienen los valores para una recta dado dos puntos dentro de ese espacio $\mathcal{X}$ usando las funciones provistas.

Los datos utilizados para esta ejecución en particular se pueden observar en la Figura 9.

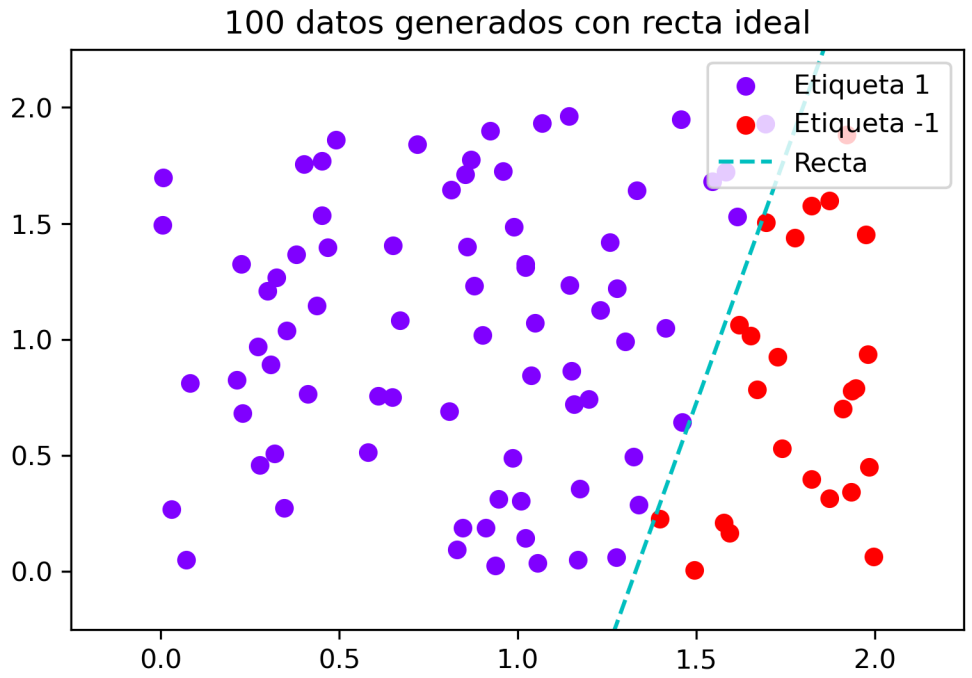

Figura 9: Datos generados de manera uniforme en $\mathcal{X} = [0, 2] \times [0, 2]$

Se obtienen las etiquetas con la función ya anteriormente comentada `getTags()` y a los datos se le añade el sesgo con `addBias()`, y dado que se necesita obtener los valores de la tasa de aprendizaje y del tamaño de lote, se implementó la función `getBestParams(x, y, wIn, eta, batch, maxIters, gradFun)` la cual dado los datos `x`, sus etiquetas `y`, el vector inicial de los pesos, las iteraciones máximas y la función de gradiente, internamente va probando con los valores de tasa de aprendizaje y lote que se le pasan como un vector `eta` y `batch` que tiene el formato `[inicio, fin, paso]` para ir probando los valores desde el valor de inicio hasta el final, paso a paso, la función se llama a `sgd()` para entrenar y se compara el error de entropía cruzada de los pesos finales para almacenar el mejor vector de pesos. Para los experimentos, se probó con una tasa de aprendizaje inicial de 0,0001 hasta 0,1 y de paso 0,0005, para los lotes el tamaño inicial es de 2 hasta 100, de 2 en 2; se utilizaron los pesos inicializados a cero y con 10^5 iteraciones máximas.

El mejor valor de los pesos se obtuvo con la tasa de aprendizaje a 0,0996 y un tamaño de lote de 2.

Una vez realizado esto, se procede a entrenar con dichos valores, se realiza una llamada a la función `sgd()` con los parámetros apropiados.

Una vez finalizado el entrenamiento, se puede observar el hiperplano generado en la Figura 10; se obtuvo un error de entropía cruzada $E_{in} = 0{,}04757$ en 46400 iteraciones y esto da una tasa de aciertos del 100 %.

Una vez realizado esto, se generó otra muestra de datos de test de tamaño 1611, un número elegido arbitrariamente mayor que 999, como se pide en el guión.

Se puede observar este conjunto de datos en la Figura 11, en el cual se obtuvo un error de fuera de la muestra $E_{out} = 0{,}056843$ y un accuracy de 99,26 %.

Una vez realizado esto, se repite el experimento 100 veces más, obteniendo 100 nuevos datos de entrenamiento, una nueva recta que los corte, se generan 1611 datos de test. Una vez finalizado, se obtiene que el error fuera de la muestra medio $\bar{E}_{out} = 0{,}054896$, un accuracy medio de 98,44 % y unas 43000.5 iteraciones media para la convergencia.

Se puede observar y con confianza acertar que el algoritmo está funcionando correctamente pues logra clasificar de manera correcta todos los datos de entrenamiento, algo que es lógico dado que se no se tiene ruido añadido a los mismos; como se puede notar si se realizan diferentes ejecuciones no necesariamente se obtiene una clasificación del 100 % en todo momento, puesto que con el criterio de parada impuesto sobre SGD, la expresión 5, puede suceder que el luego de varias épocas hiperplano deba de moverse ligeramente para poder clasificar perfectamente los datos pero puede suceder que este movimiento sea de un rango más pequeño del criterio de parada y por ende termina el algoritmo con uno o dos puntos mal clasificados pero con el hiperplano muy cerca del mismo.

3. Bonus: Clasificación de dígitos con diferentes algoritmos

Para el bonus de la práctica, se utilizan los dígitos manuscritos nuevamente, esta vez se extraen los dígitos 4 y 8 para poder clasificarlos haciendo uso de dos características: la simetría de los dígitos y su intensidad promedio, y serán clasificados haciendo uso de la Regresión Lineal, el algoritmo del Perceptrón, su variante Pocket y la Regresión Logística.

Se utiliza la función provista, `readData()` para cargar los datos de entrenamiento y test a memoria.

Se pueden observar la distribución de los datos de los números según las características en la Figura 12.

3.1. Entrenamiento y Test con los algoritmos

Con los datos cargados, se procedió a realizar un entrenamiento con el algoritmo de la Pseudoinversa, también se reutiliza el algoritmo PLA, la Regresión Logística y la variante de PLA conocida como PLA-Pocket. Todos los algoritmo, a excepción de la Pseudoinversa que no lo necesita, tuvieron sus pesos inicializados a 0.

El algoritmo de la Pseudoinversa se trae directamente de la práctica anterior sin cambios, se llama con `pseudoinverse(x,y)` siendo `x` los datos de entrenamiento e `y` sus etiquetas.

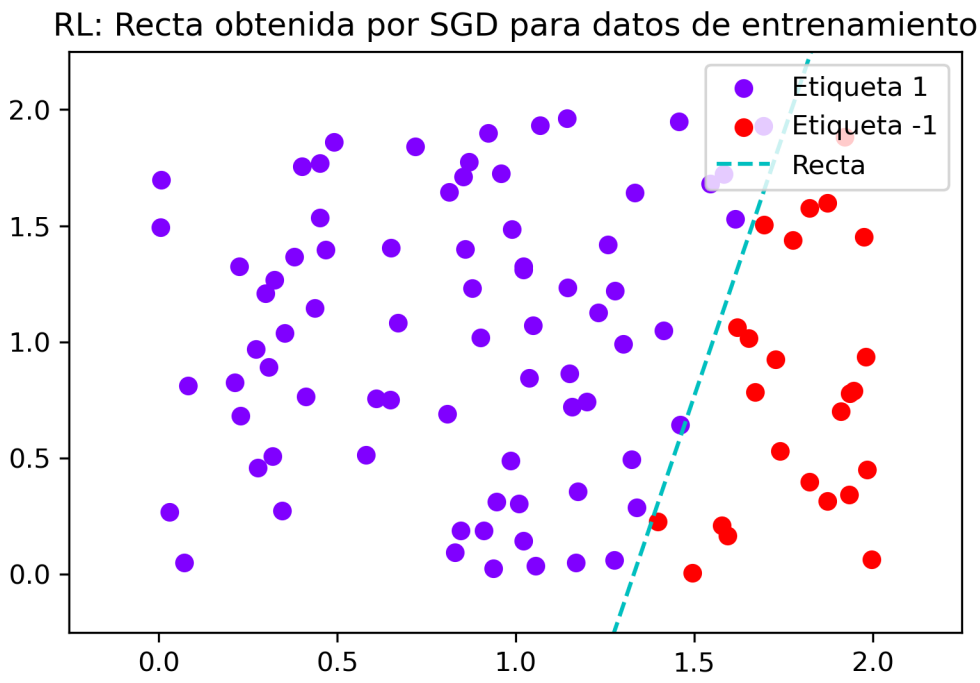

Figura 10: Hiperplano obtenido por Regresión Logística para 100 datos en el espacio $\mathcal{X} = [0, 2] \times [0, 2]$

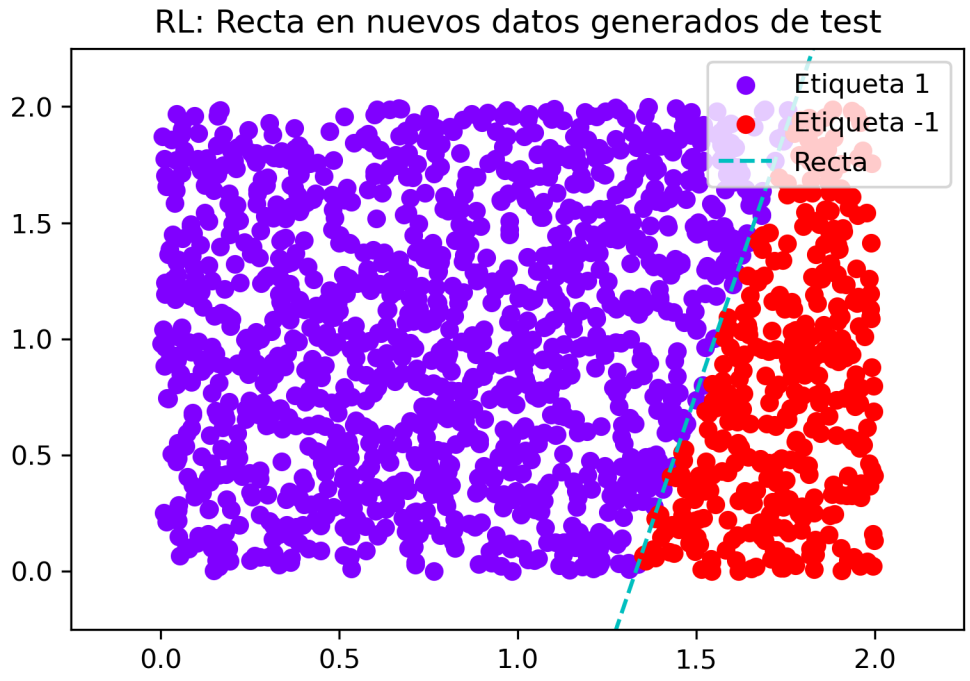

Figura 11: Hiperplano obtenido por Regresión Logística para 1611 datos en el espacio $\mathcal{X} = [0, 2] \times [0, 2]$

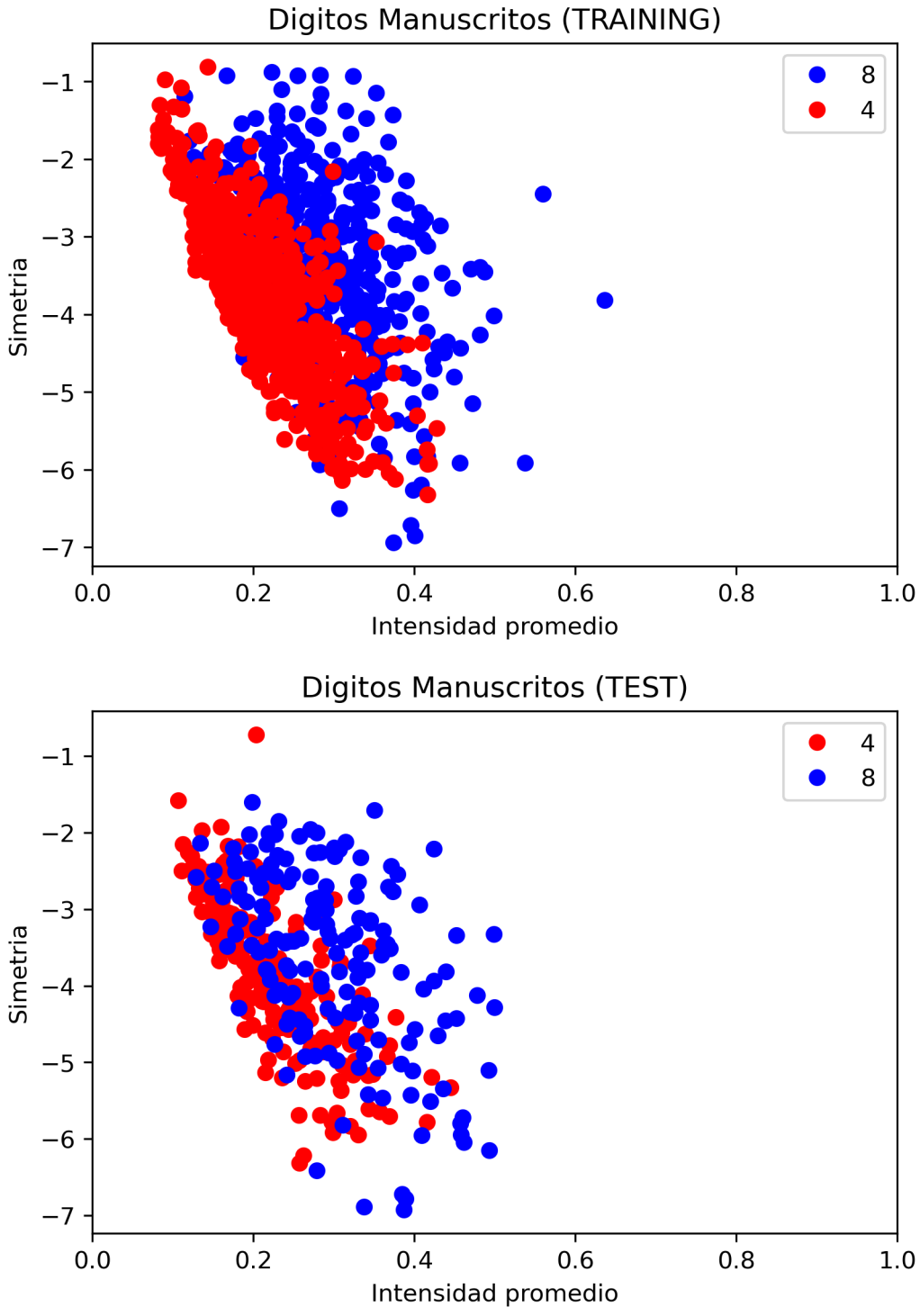

Figura 12: Visualización de los datos de entrenamiento y test

De una manera similar, se ejecuta `ajusta_PLA()` con `pocket=False` en primer lugar para realizar una ejecución con el algoritmo del Perceptrón original.

Luego, sí se utiliza `pocket=True` para realizar la variante PLA-Pocket, para esto se modificó el código de `ajusta_PLA()` de forma que ahora se mantiene una variable con el mejor error obtenido hasta el momento, y el peso que obtiene dicho error, entonces cada vez que se realiza una modificación de los pesos se obtiene el error y si es un error más pequeño que el actual, se almacena hasta que se acaban las iteraciones, luego, se devuelve ese error.

Finalmente, la regresión logística se realiza normalmente, con la misma cantidad de iteraciones que antes pero se varía el valor de la tasa de aprendizaje pues se ha utilizado la función `getBestParams()` para estos datos, donde se obtuvo que con un tamaño de lote de 2 y una tasa de aprendizaje de 0.087 se obtenían los mejores resultados; finalmente se realiza el entrenamiento haciendo una llamada a `sgd()` pasando por parámetro la función de gradiente `gradRL()`.

Realizado esto, se obtienen de resultado las siguientes gráficas de Entrenamiento y Test por cada algoritmo, haciendo uso de la función auxiliar `evalFunction()` que dibuja las gráficas de manera cómoda.

Se pueden observar y comparar los algoritmos en las Figuras 13, 14, 15 y 16, además de la Tabla 5 donde se resumen los valores obtenidos de Error y Accuracy.

Algoritmo	E_{in}	Train accuracy	E_{test}	Test accuracy
Pseudoinversa	0.64285	77.22 %	0.70871	74.86 %
PLA	0.33417	66.58 %	0.38525	61.48 %
PLA-Pocket	0.21106	78.89 %	0.25137	74.86 %
RL	0.47160	77.55 %	0.54900	73.77 %

Tabla 5: Tabla resumen de los resultados de Entrenamiento y Test para cada algoritmo

Se puede observar que, salvo PLA, el resto de algoritmos funcionan de manera similar y logran dividir los datos de una forma sensible, se puede apreciar que la modificación de PLA, PLA-Pocket, obtiene el mejor error dentro y fuera de la muestra, compartiendo el mejor porcentaje de Accuracy con la Pseudoinversa, siguiéndole muy de cerca la Regresión Logística con a penas 1 % menos de aciertos.

Esto permite confirmar en primer lugar, que la modificación a PLA logra mejorar muchísimo su rendimiento y utilidad, pues ahora PLA no está limitado por el requerimiento de que los datos sean estrictamente separables, además de que ahora se sabe que los pesos obtenidos de vuelta son justamente los mejores que se han obtenido, queda más que clara el salto de calidad de 61 % a 74 % en el error fuera de la muestra, y una historia similar ocurre con los errores dentro de la muestra.

Si bien uno esperaría que Regresión Lineal obtuviera mejores resultado, se tiene que tomar en cuenta nuevamente la naturaleza estocástica de la función en sí, además de que como se comentó también previamente, el criterio de parada puede influir a que no se obtenga los mejores pesos posibles si ya se encuentra muy, muy cerca de ese óptimo. Aún así, el resultado es bastante respetable, y también tiene sentido que PLA-Pocket clasifique muy bien pues es igualmente un algoritmo de clasificación, igual de bien que la Pseudoinversa aunque este algoritmo sea de regresión da muy buenos resultados,

Algoritmo	E_{in}	Train accuracy	E_{test}	Test accuracy
PLA	0.28308	71.69 %	0.32787	67.21 %
PLA-Pocket	0.21273	78.73 %	0.25410	74.59 %
RL	0.46685	78.56 %	0.53239	75.68 %

Tabla 6: Tabla resumen de los resultados de Entrenamiento y Test para cada algoritmo de clasificación con pesos inicializados por la Pseudoinversa

como ya se conocía de la práctica anterior y de teoría, por lo que se sigue confirmando lo conocido de la misma, y su funcionamiento.

3.2. Entrenamiento con pesos inicializados por la Pseudoinversa

Ahora bien, se repite el experimento anterior pero utilizando como vector de pesos iniciales los pesos obtenidos por la Pseudoinversa, para esta ejecución en particular, los pesos iniciales son $[-0{,}5068; 8{,}252; 0{,}4446]$.

El resto de variables pertinente a cada algoritmo se mantiene igual, se puede observar el resultado en otra tabla resumen, la Tabla 6 y las gráficas se pueden apreciar en las Figuras 17, 18 y 19.

Se puede observar que por ejemplo, PLA que tenía un accuracy de 66.58 % en Train y 61.48 % en test pasó a tener 71 % y 67 % respectivamente, PLA-Pocket pasó de 78.89 % y 74.86 % a 78.73 % y 74.59 %, así como RL que tenía 77.55 % y 73.77 % a 78.56 % y 75.68 %; claramente la calidad del aprenizaje a aumentado, aunque sea ligeramente. Esto concuerda con lo estudiado en teoría sobre utilizar la Pseudoinversa como un paso previo antes de utilizar un algoritmo de clasificación para obtener rápidamente unos pesos de calidad y mejorarlos desde allí, en efecto puede observarse que dicho fenómeno se ha manifestado en este experimento, por lo que se concluye que sí, es una buena idea realizar el "pre-entrenamiento" con la pseudoinversa. También se concluye que tanto RL como PLA-Pocket tienen una potencia equiparable, pues ahora RL ha salido victoriosa, teniendo la tasa de aciertos mayor aunque Pocket está muy cerca también.

3.3. Cálculo de cotas de E_{out}

Para obtener las cotas, se almacenaron los errores en dos vectores, estos son los errores obtenidos en la Sección para los cuatro algoritmos utilizados.

Una vez realizado esto, para cada E_{in} y E_{test} se realizó el cálculo de las expresiones 9 y 10, las cuales se adaptaron a código dentro de las funciones `boundEin(Ein, N, dVC, delta)` y `boundEtest (Etest, N, cardH, delta)`.

$$E_{out}(h) \leq E_{in}(h) + \sqrt{\frac{8}{N}\log(\frac{4\cdot((2N)^{d_{VC}}+1)}{\delta})} \tag{9}$$

$$E_{out}(h) \leq E_{test}(h) + \sqrt{\frac{1}{2N}\log(\frac{2|\mathcal{H}|}{\delta})} \tag{10}$$

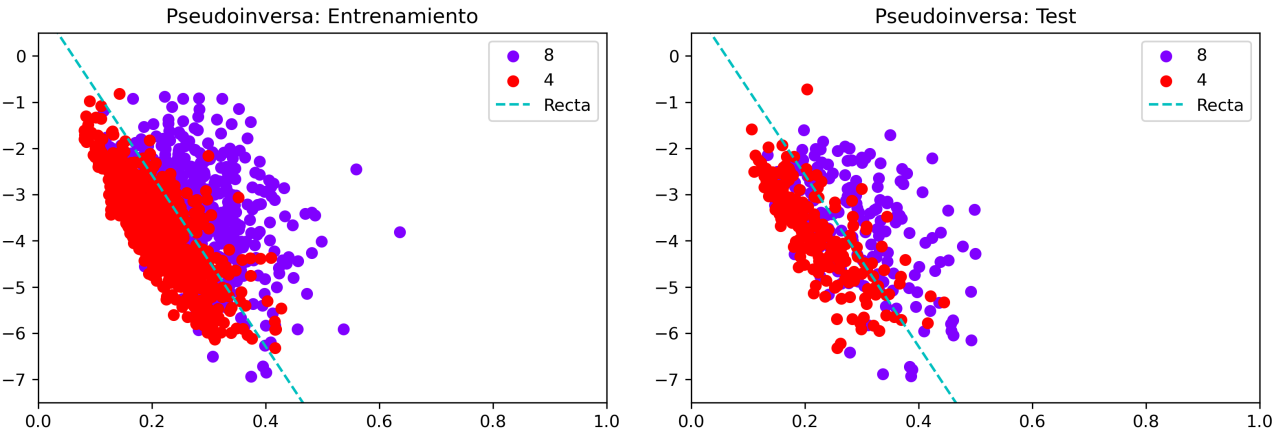

Figura 13: Clasificación con la Pseudoinversa

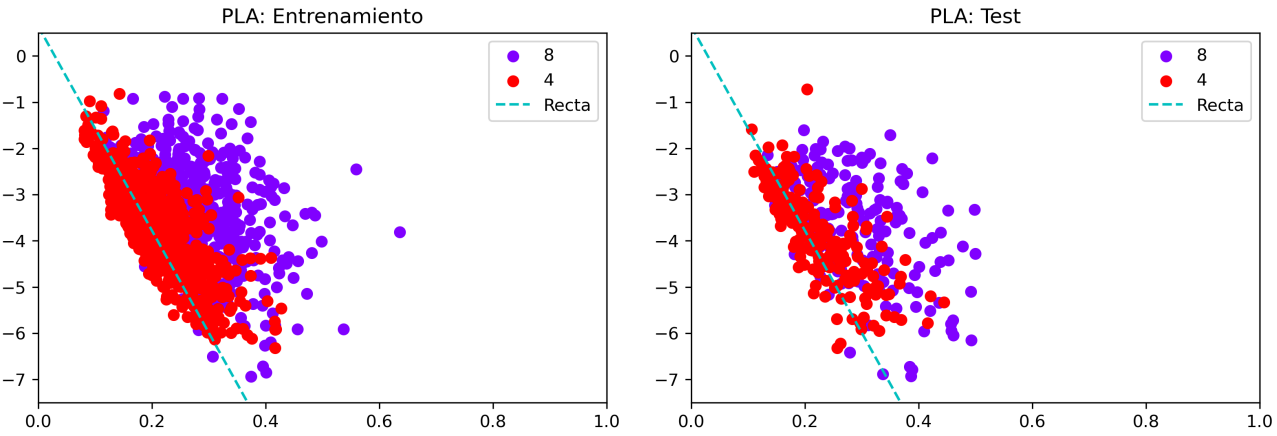

Figura 14: Clasificación con PLA

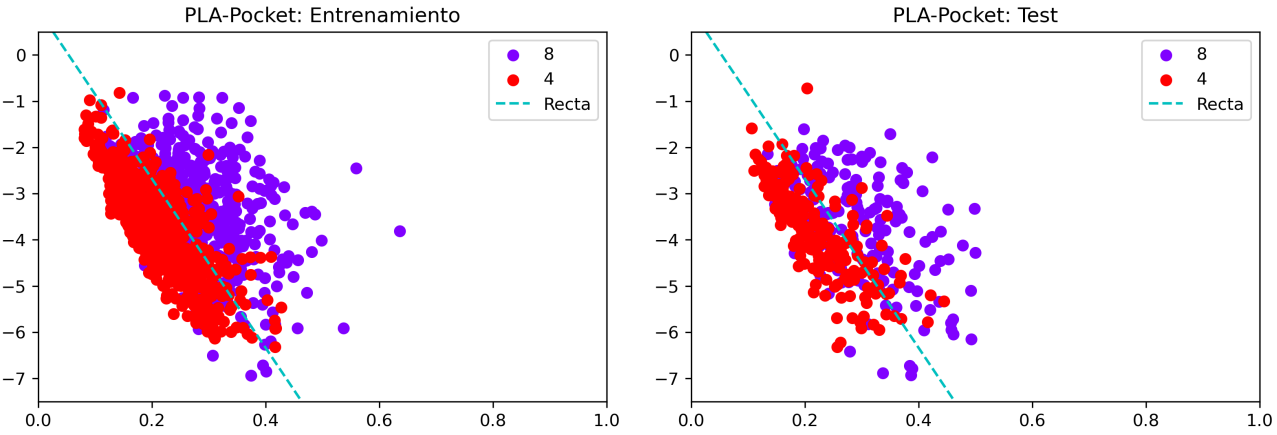

Figura 15: Clasificación con PLA-Pocket

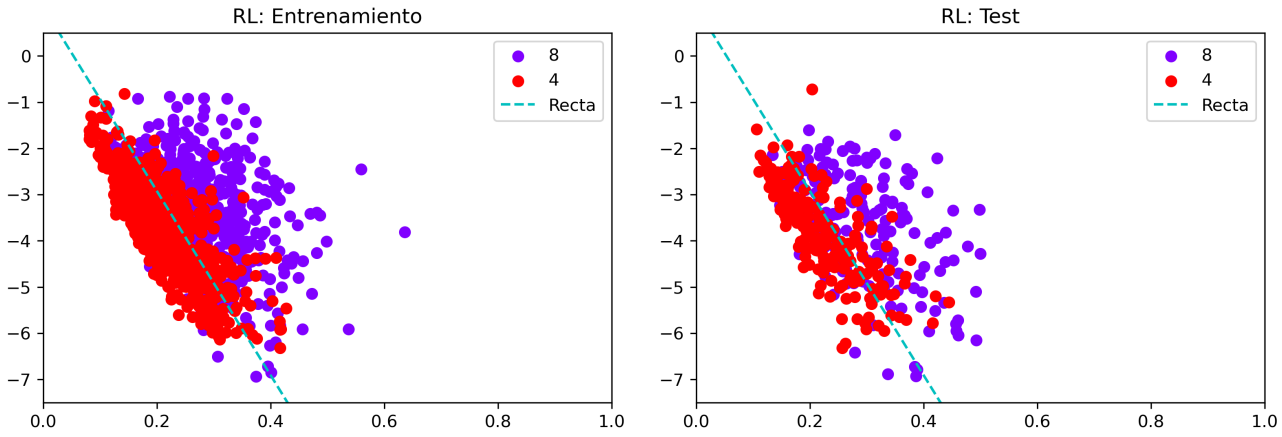

Figura 16: Clasificación con Regresión Logística

Se utilizó un $\delta = 0{,}05$ además de que, se sabe por teoría y se puede corroborar en prácticas que la dimensión VC de todos estos algoritmos es 3, que es justamente el número de parámetros efectivos, o sea, el vector w que tiene longitud 3, por otro lado la cardinalidad de $\mathcal{H}$ en test se sabe es 1, puesto que solamente se tiene una única función en test, aquella que fue elegida de entre todas las de la clase que reduce de mejor forma el error.

Realizando los cálculos, se obtiene que la cota de $E_{out}(h)$ para cada algoritmo y se puede visualizar en la Tabla 7.

Como se puede observar, la cota del error E_{test} es significativamente menor para cada uno de los algoritmos utilizados respecto a la cota de E_{in}, esto nuevamente afirma lo aprendido en teoría sobre el utilizar un conjunto de test, como se puede observar claramente, es un mejor estimador del verdadero error de E_{out} puesto que es una cota más pequeña, por lo tanto se acerca mucho más al valor real que posee la distribución de probabilidad, y dentro de esto, se puede observar que la cota que más se asemeja al verdadero E_{out} sería PLA-Pocket, seguido de PLA, RL y la Pseudoinversa.

Algoritmo	Cota E_{in}	Cota E_{test}
Pseudoinversa	0.92684	0.75550
PLA	0.61816	0.43203
PLA-Pocket	0.64199	0.29815
RL	0.75559	0.59579

Tabla 7: Cotas de los errores por cada algoritmo utilizado

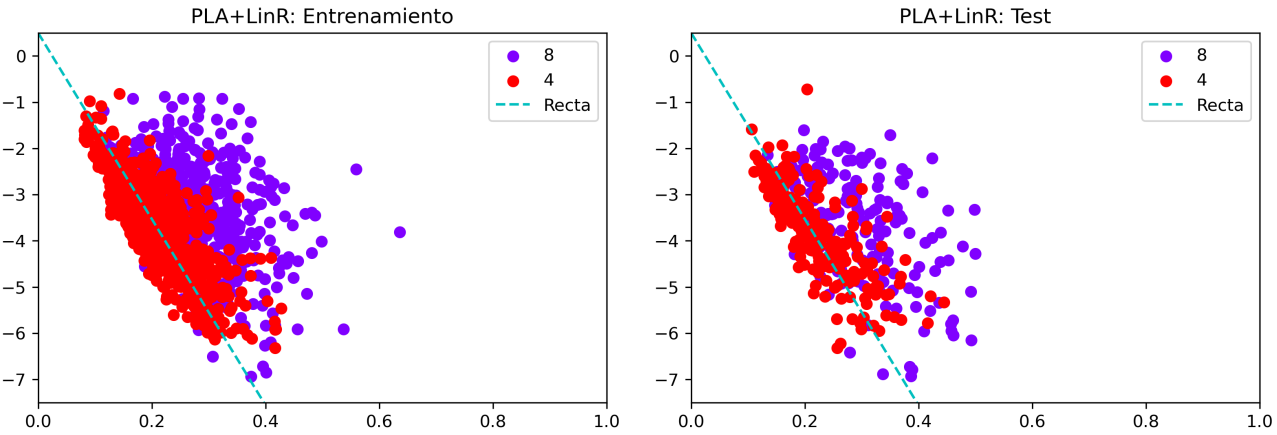

Figura 17: Clasificación con LinR+PLA

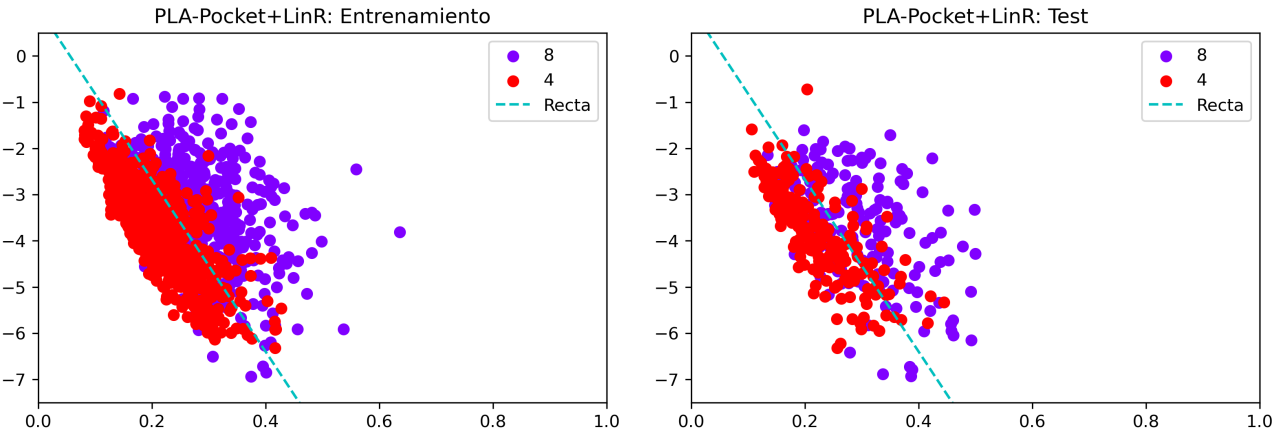

Figura 18: Clasificación con LinR+PLA-Pocket

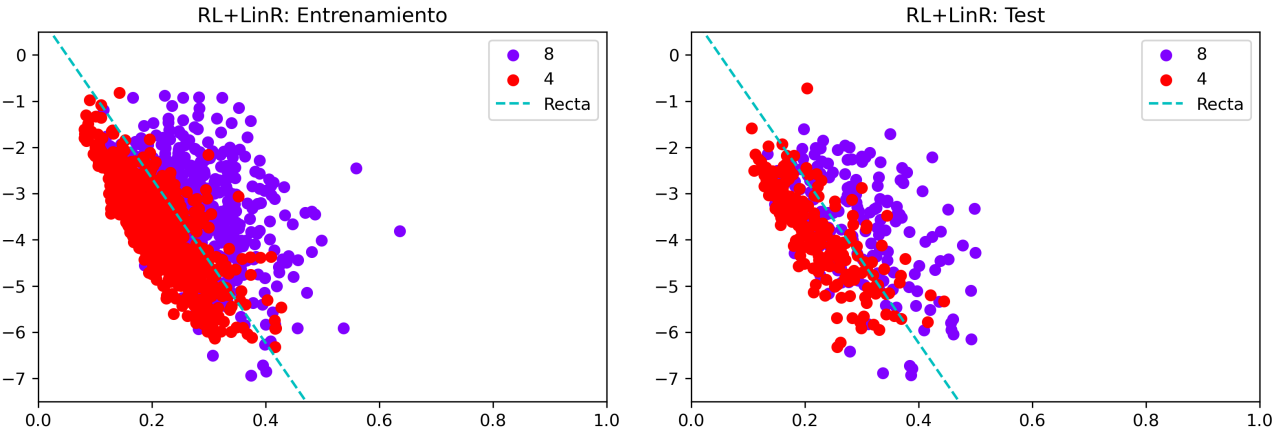

Figura 19: Clasificación con LinR+RL