UNIVERSIDAD DE GRANADA

Aprendizaje Automático

Práctica 2
Complejidad de H y Modelos Lineales

Alumnos:
Jacobo Casado de Gracia

5º Curso - Doble Grado Ingeniería Informática
y Administración y Dirección de Empresas

Índice

1. Trabajo a realizar

En la primera parte de esta práctica se trabajará con la presencia de ruido en los datos y la capacidad de diferentes funciones a la hora de ajustar datos con ruido. Veremos si existe alguna correlación entre la complejidad del modelo y del set de hipótesis H con la capacidad de ajustar de mejor manera datos con ruido.

En la segunda parte de la práctica trabajaremos con dos modelos lineales, el *PLA, o Perceptron Learning Algorithm* y su variante el *PLA Pocket*, y el modelo de *Regresión Logística, o RL*, ajustando un conjunto de datos con estos dos modelos y comparando los resultados obtenidos.

Finalmente, en el bonus se verá el comportamiento de estos algoritmos en un conjunto de datos de dígitos manuscritos, midiendo su eficacia a la hora de clasificar.

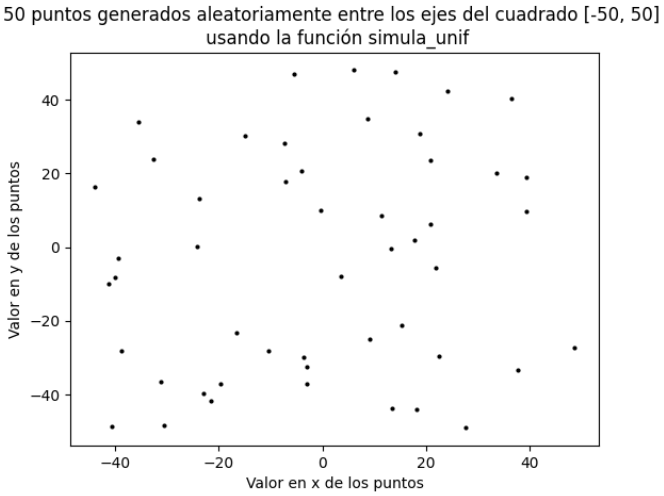
2. Ejercicio sobre la complejidad de H y el ruido

2.1. Apartado 1

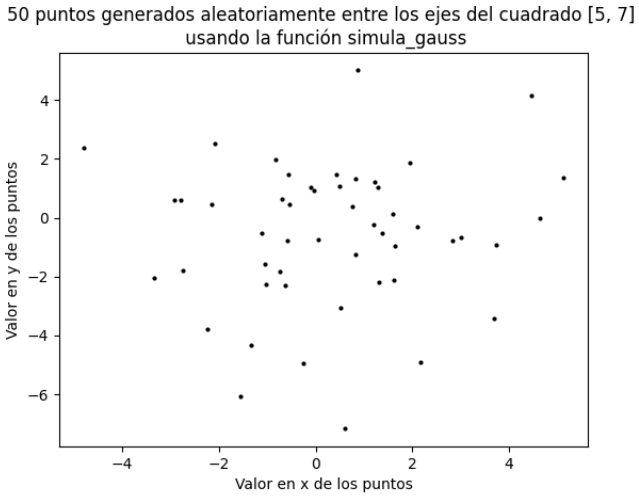
En este apartado se usan las dos funciones, `simula_unif` y `simula_gauss` para calcular una nube de puntos en un intervalo dado siguiendo dos distribuciones de probabilidad, la distribución *uniforme* y la distribución *gaussiana*, respectivamente.
Usaremos posteriormente estos puntos como nuestro set de datos para crear un modelo que ajuste estos datos, añadirle ruido a los datos y usar otros modelos para clasificar estos datos, comparando el resultado obtenido.

Lllamando a la función `simula_unif(50, 2, [-50,50]` y `simula_gauss(50, 2, [5,7])`, creamos dos nubes de puntos en el los intervalos [-50, 50] y [5,7], respectivamente. Cada punto tiene su coordenada en x y en y, respectivamente.

Representando estas dos nubes de puntos obtenemos los siguientes *plots*:



(a) Nube de puntos usando `simula_unif`



(b) Nube de puntos usando `simula_gauss`

2.2. Apartado 2

En este apartado se valorará cómo afecta el ruido al seleccionar clases de funciones diferentes.

Usaremos la función `simula_unif(100, 2, [-50,50])` para generar 100 puntos con sus coordenadas (x, y), puntos a los que añadiremos una **etiqueta** de manera determinista usando el **signo** de la función $f(x, y) = y - ax - b$, usando para ello la distancia de cada punto (x,y) a la recta simulada con la función `simula_recta` (que simulará una recta 2D con parámetros **a** y **b** aleatorios) y, una vez utilizada esta distancia, tomaremos el signo.

Por tanto, tenemos 100 puntos etiquetados (con un valor de **-1 ó 1**) respecto a una función **f** que graficaremos a continuación.

2.2.1. Apartado a)

Dibujaremos el gráfico 2D con las coordenadas *(x,y)* de cada punto y mostrando el resultado de la etiqueta de cada punto usando colores.

Además, dibujaremos la recta con la que hemos clasificado los puntos, para comprobar que la recta efectivamente ha sido la función objetivo con la que hemos generado las etiquetas de los puntos (usando el signo).

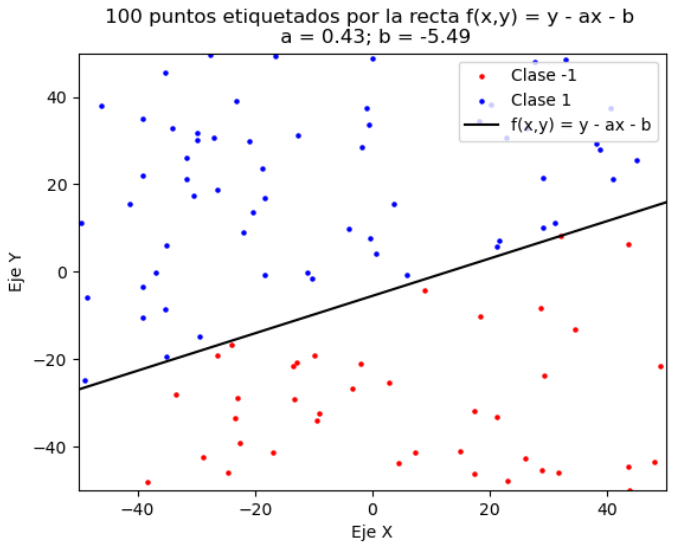


Figura 1: 100 puntos etiquetados (azul/rojo) en función de la recta f (negro). En la leyenda se indica la información de la clase y en el título del gráfico los parámetros **a y b** de la recta.

Como podemos ver, los puntos son completamente clasificados respecto de esta recta **f** y no hay ningun error dentro de la muestra (E_{in}) si usamos la misma recta para clasificar los puntos que para predecir su valor.

Aun así, he diseñado un método para este apartado llamado `get_accuracy_recta(x,a,b,y)` que devuelve el **error de clasificación** en tanto por uno de la recta con parámetros a y b respecto a los puntos x (en este caso, E_{in}). Ejecutando este método, efectivamente, obtenemos un valor de `0.0`.

2.2.2. Añadiendo ruido a los puntos (apartado b)

En este apartado modificaremos de manera aleatoria un 10 % de las etiquetas positivas y un 10 % de las etiquetas negativas (en total, alteramos un 10 % de los puntos, y creando un 10 % de ruido) y dibujamos la misma gráfica anterior. El gráfico obtenido es el siguiente:

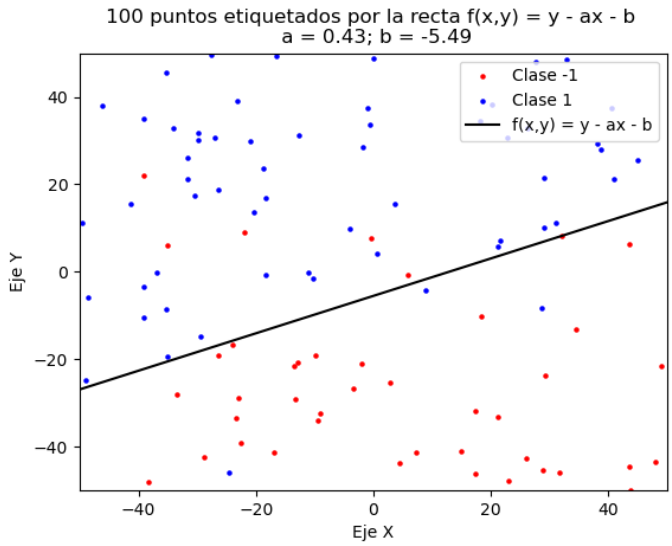


Figura 2: 100 puntos etiquetados (azul/rojo) en función de la recta f (negro) a los que se le ha aplicado ruido. En la leyenda se indica la información de la clase y en el título del gráfico los parámetros **a y b** de la recta.

Como vemos, ahora la recta no clasifica los puntos de manera completamente precisa (con un E_{in} de 0) sino que el error dentro de la muestra es de un **10** %, correspondiente a los puntos que han cambiado de valor.

Para cerciorarnos de esto, ejecutamos el método `get_accuracy_recta(x,a,b,y)` con esta nueva y que almacenará ruido y obtenemos un valor de `0.1`, exactamente un 10 % de error de clasificación, correspondiente a las muestras con ruido.

A poco que pensemos, hemos etiquetado los puntos de manera original usando una recta por lo que conocemos la función objetivo para esos puntos y es la propia recta.

Los únicos puntos que no podrán ser bien clasificados por la recta son aquellos que han cambiado de valor **tras** el etiquetado por ella misma, es decir, el **ruido**.

La pregunta que nos surge a continuación es: **¿Existirá alguna función capaz de clasificar estos puntos de una manera más precisa, evitando el 10 % de error de clasificación debido al ruido y obteniendo un menor valor?**

2.2.3. Apartado c)

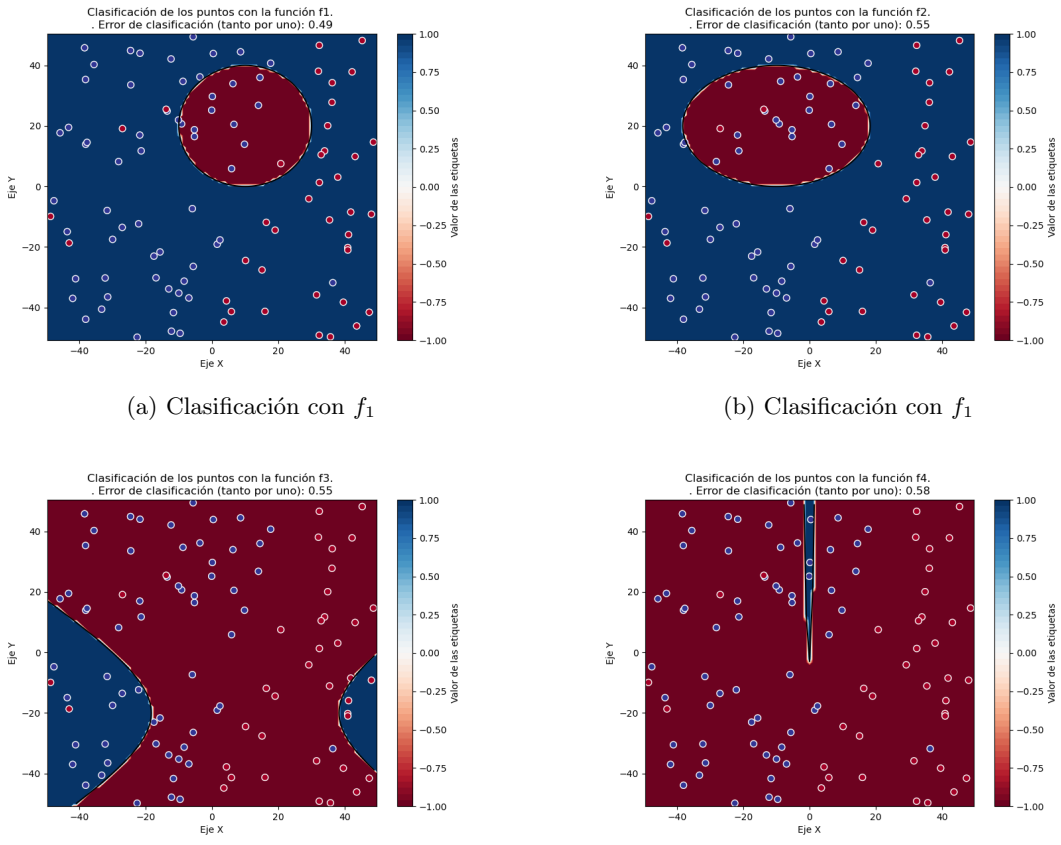
En este apartado usaremos cuatro funciones que definirán una frontera de clasificación de los puntos en lugar de una recta. Las funciones son las siguientes:

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

2.2.4. Clasificación vs. función f

En primer lugar, aplicaremos estas fronteras de clasificación no lineales sobre los datos creados anteriormente para obtener el error de clasificación (en este caso, el porcentaje de ejemplos incorrectamente clasificados empleando dichas fronteras de decisión).

Gráficamente, obtenemos estas fronteras de clasificación (se ha usado la función `plot_datos_cuad()` proporcionada por el profesorado para ello):



(a) Clasificación con f_1

(b) Clasificación con f_1

(c) Clasificación con f_1

(d) Clasificación con f_1

Figura 3: Clasificación de los puntos con las nuevas fronteras.
E_{in} de cada una en el título.

Podemos ver gráficamente que el ajuste de los datos con estos modelos no es mejor que el ajuste de la recta original que se usó para dar las etiquetas a los puntos; en el caso de la elipse, por

ejemplo, hay cantidad de puntos mal etiquetados fuera de la elipse que esta no es capaz de recoger.

Aun así, para llegar a una conclusión analítica, se ha extraído el error E_{in} de clasificación de estas cuatro funciones. Este error de clasificación aparece en el título de cada gráfico y en la siguiente tabla:

Frontera	f_{recta}	f_1	f_2	f_3	f_4
E_{in}	0.1	0.52	0.54	0.63	0.62

Cuadro 1: Error de clasificación (tanto por uno) con las nuevas fronteras de clasificación. Los datos tienen un 10 % de ruido y han sido etiquetados con f.

Numéricamente comprobamos que ninguno de los modelos nuevos se han comportado como mejores clasificadores para nuestros datos y, tiene sentido, ya que los datos fueron **generados** con un modelo y estamos **usando otros modelos para clasificarlos**, por mucho que estas funciones sean más complejas.

De hecho, estos modelos tienen más de un 50 % de error de clasificación, valor bastante pobre para un problema de clasificación binaria; usando fronteras de clasificación más complejas para clasificar puntos cuya función objetivo es una recta no nos da mejora.

La siguiente pregunta que nos hacemos es:

¿Qué pasa si repetimos el proceso con estas funciones más complejas (las empleamos para etiquetar los datos y luego metemos un 10 % de ruido)?
¿Qué error de clasificación tenemos? ¿Es menor que ese 10 %? ¿Podremos eliminar el error causado por el ruido?

2.2.5. Etiquetado con otras funciones (f_i) más complejas

A continuación se hace el estudio anterior pero usando las funciones f_i para clasificar los datos (como se ha hecho en el apartado a)), se añadirá un 10 % de ruido por clase y se probará el error de clasificación E_{in} con el conjunto de funciones f_i.
Realizando este proceso se obtiene la siguiente tabla, donde en las filas se ubican las funciones con las que se han **etiquetado** los puntos y en las columnas las funciones con las que se han **clasificado** los puntos, y el valor indica el error de clasificación:

Clasificado / Etiquetado	f_1	f_2	f_3	f_4
función f_1	0.1	0.2	0.6	0.81
función f_2	0.19	0.1	0.57	0.78
función f_3	0.62	0.56	0.1	0.31
función f_4	0.81	0.75	0.29	0.1

Cuadro 2: Tabla con los errores de clasificación en tanto por uno de las fronteras no lineales. La función que etiqueta está en las filas, y se evalúan con las funciones de las columnas.

En la tabla superior podemos comprobar que los valores más pequeños del error de clasificación se encuentran en el caso en el que **la misma función que etiqueta los puntos los clasifica posteriormente**; si usamos una función para clasificar los puntos diferente a la usada para etiquetarlos obtenemos errores de clasificación superiores, sin lograr ninguna mejora.

Además de eso, se puede observar que tampoco somos capaces de **eliminar el error de clasificación debido al ruido**, por muchas funciones complejas que probemos. En ninguno de los casos se ha podido eliminar el 10 por ciento de muestras mal clasificadas; estas funciones no son mejores bordes de decisión (en el aspecto de reducir el ruido) por más complejas que las diseñemos, ya que no logran reducir este error.

3. Ajuste con modelos lineales

3.1. Algoritmo Perceptrón (PLA)

En este apartado introducimos el algoritmo de aprendizaje del Perceptrón (PLA, o *perceptron learning algorithm.*). El algoritmo determinará el valor de los pesos, **w**, en base a nuestro conjunto de datos **X**.

El funcionamiento del perceptrón sigue un proceso iterativo:

- El algoritmo elige un ejemplo de los datos (x_t, y_t) que está actualmente mal clasificado. Al estar mal clasificado, sabemos que $y(t)$ (su valor real) no coincide con el **signo de** $w^T(t)x(t)$.
- El algoritmo actualiza los pesos, w, de acuerdo a la siguiente regla:

$$w(t+1) = w(t) + y(t)x(t) \tag{1}$$

  Esta regla mueve la frontera de clasificación hacia la dirección que clasifica $x(t)$ de manera correcta.

El funcionamiento del perceptron, tal y como es citado en la bibliografía de la asignatura, *Learning From Data* [1], continua iterativamente hasta que no hay ejemplos mal clasificados en el data set. Está demostrado que el algoritmo converge a una solución en la que todos los ejemplos $(x_1, y_1)...(x_N, y_N)$ están correctamente clasificados, pero bajo la condición de que el **conjunto de datos es linearmente separable.**

En el caso de que nuestros datos no sean linearmente separables, el algoritmo no asegura converger y por tanto, es necesario establecer un criterio de parada alternativo.
En el caso de esta práctica, se usará un número máximo de iteraciones, teniendo en cuenta que cada iteración se valorará cuando se haya comprobado si está bien o no etiquetado todo el conjunto de datos.

Implementaremos este criterio debido a que no sabemos si nuestro set de datos, en un principio, es linealmente separable, por lo que hay que asegurarse de que el algoritmo converge a una solución.

3.1.1. Datos sin ruido

En primer lugar, trabajaremos con un set de datos sin ruido.
Para ello, generaremos 100 puntos y los etiquetaremos usando una recta arbitraria con parámetros (a,b), **al igual que en el apartado 2.a del ejercicio 1.**.

Posteriormente, haremos lo siguiente:

- Inicializaremos nuestros pesos **w** a un valor inicial de [0,0,0] y ejecutaremos *PLA* (con un máximo de 1000 iteraciones).
- Inicializaremos nuestros pesos **w** a un valor aleatorio entre [0,1] y ejecutaremos 10 veces *PLA*, y almacenando los resultados de todas las ejecuciones, con un máximo de 1000 iteraciones por cada ejecución.

Los resultados obtenidos son los siguientes, en forma de tabla:

w_{ini}	w_{final}	Iteración de parada	E_{in}
[0.64, 0.16, 0.35]	[358.64, 11.78, 18.07]	25	0.0
[0.87, 0.88, 0.42]	[569.87, 19.59, 27.34]	43	0.0
[0.42, 0.15, 0.55]	[604.42, 19.01, 29.35]	45	0.0
[0.71, 0.16, 0.61]	[604.71, 19.02, 29.42]	45	0.0
[0.66, 0.94, 0.33]	[917.66, 31.53, 47.53]	84	0.0
[0.00, 0.00, 0.00]	[942.00, 33.49, 47.11]	91	0.0
[0.50, 0.87, 0.25]	[948.50, 33.60, 46.88]	97	0.0
[0.27, 0.84, 0.25]	[975.27, 34.94, 50.42]	99	0.0
[0.79, 0.04, 0.63]	[993.79, 35.62, 51.64]	100	0.0
[0.03, 0.48, 0.08]	[1169.03, 40.07, 55.65]	158	0.0
[0.75, 0.81, 0.45]	[1364.76, 47.11, 64.16]	235	0.0

Cuadro 3: Comparativa de las ejecuciones del *PLA* utilizando diferentes puntos de inicio. La tabla se ha ordenado en orden creciente respecto a la iteración de parada del algoritmo.

Podemos sacar varias conclusiones a partir de la tabla anterior.
En primer lugar, el punto de inicio afecta en el resultado, alterando el valor del vector de pesos solución y el número necesario de iteraciones para converger. Vemos en la tabla que, mientras que en una ejecución el algoritmo converge en **25** iteraciones, en otra lo hace en casi 10 veces más, en **235**, pero siempre antes de alcanzar las 1000 iteraciones impuestas como criterio alternativo.

Esto nos advierte de que quizás, en casos reales, sea necesario ejecutar nuestros algoritmos de aprendizaje varias veces desde varios puntos de inicio, ya que los resultados pueden variar bastante dependiendo de éste.

Por último, podemos dar cuenta de que, como hemos usado una recta para separar los datos originalmente, el conjunto de datos **es linealmente separable** y, por tanto, el algoritmo **encuentra un hiperplano que separa los puntos antes de las 1000 iteraciones (con alrededor de 93 iteraciones de media), llegando a clasificar correctamente todos los puntos en todas las ejecuciones.**
Si los datos son linealmente separables existen infinitas rectas que los separan, y, como vemos, cambiando el punto inicial encontraremos una u otra, pero todas con el mismo error de clasificación.

Mostramos en la siguiente figura una gráfica con la evolución temporal del E_{in} a lo largo de las iteraciones del algoritmo, observando que, aunque en algunas iteraciones ascienda levemente, la tendencia es a descender, llegando al 0.0 %.

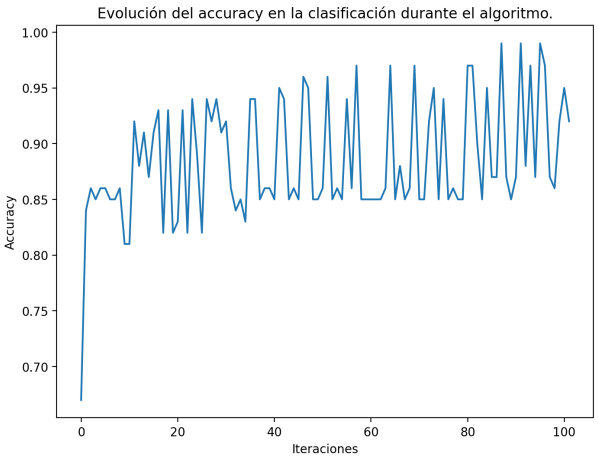


Figura 4: Evolución del valor de E_{in} conforme avanza el número de iteraciones del algoritmo PLA.
Datos sin ruido.

Por último, se muestra el ajuste de la recta creada por este algoritmo en comparación con la recta original.

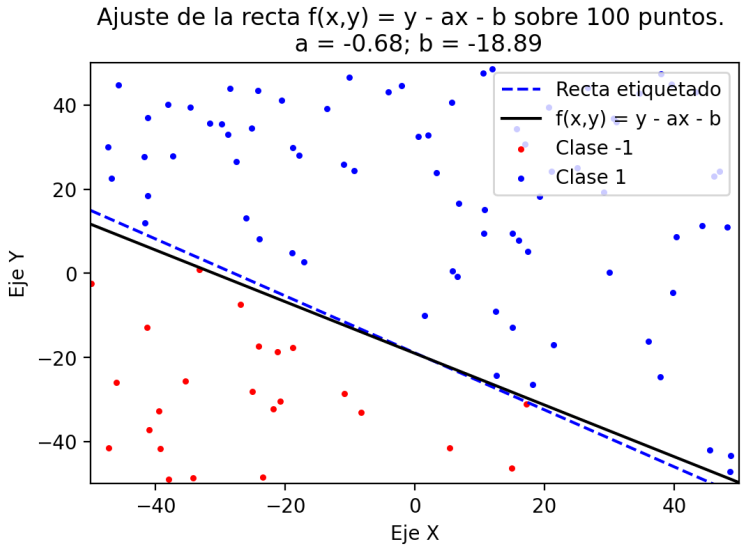


Figura 5: Ajuste del perceptrón (negro) VS recta que etiqueta los puntos (azul)
Datos sin ruido.

Vemos que la recta generada es muy parecida a la original y se comprueba lo que hemos visto en clases de teoría: **dos conjuntos diferentes de hipótesis *g* pueden separar los puntos de la misma manera**. En nuestro caso, ambas rectas separan los puntos de manera perfecta, pero son distintas rectas.

3.1.2. Aplicando ruido a los datos

A continuación se procede a realizar exactamente el mismo procedimiento de la sección anterior, pero aplicando previamente **un 10 % de ruido a los datos**, para ver el comportamiento del algoritmo con un conjunto de datos donde hay ruido de por medio.
Al igual que en el apartado anterior, los resultados obtenidos son los siguientes:

w_{ini}	w_{final}	Iteración de parada	E_{in}
[0.77, 0.95, 0.90]	[437.77, 22.67, 26.47]	1000	0.11
[0.16, 0.97, 0.85]	[459.16, 27.37, 21.02]	1000	0.11
[0.66, 0.51, 0.60]	[445.66, 22.11, 28.22]	1000	0.12
[0.95, 0.02, 0.81]	[429.95, 25.91, 18.86]	1000	0.12
[0.18, 0.10, 0.18]	[439.18, 30.56, 18.71]	1000	0.16
[0.60, 0.85, 0.17]	[431.60, 28.96, 20.44]	1000	0.16
[0.66, 0.69, 0.99]	[437.66, 31.78, 37.28]	1000	0.18
[0.94, 0.88, 0.25]	[431.94, 39.51, 32.70]	1000	0.18
[0.83, 0.33, 0.60]	[431.83, 39.71, 33.80]	1000	0.18
[0.00, 0.00, 0.00]	[438.00, 40.16, 31.64]	1000	0.19
[0.18, 0.06, 0.63]	[433.18, 39.20, 18.55]	1000	0.21

Cuadro 4: Comparativa de las ejecuciones del *PLA* aplicado a **datos con ruido** utilizando diferentes puntos de inicio.
La tabla se ha ordenado en orden creciente respecto al error de clasificación, E_{in}.

Podemos apreciar que, en este caso, algoritmo PLA consume todas las iteraciones dadas, independientemente del punto de inicio.
Esto ocurre debido a que hemos etiquetado los datos con una recta **pero posteriormente hemos añadido ruido a éstos**, haciendo que el conjunto de datos no sea linealmente separable (antes de aplicarle ruido sí lo era).
Al no ser linealmente separable, el algoritmo no converge de manera natural, sino que para al consumir todas las iteraciones indicadas, en este caso, 1000.

Podemos ver que el error de clasificación E_{in}, al igual que el vector de pesos final, varían dependiendo del punto de inicio, obteniendo en ocasiones un valor cercano a 0.1, y un promedio de 0.18 (alrededor del 18 % de los datos mal clasificados), un valor superior al ruido, que es de un 10 % de los datos.

Lo que sí podemos comprobar es que ninguna solución es capaz de ajustar los datos con un error inferior al 10 % en ninguna de sus ejecuciones, comprobando de nuevo que es imposible reducir el error causado por el ruido.

El problema de este algoritmo cuando tenemos un conjunto de datos que no es linealmente separable es que el algoritmo puede tener un error bajo de clasificación en la iteración 999, y al ajustar los pesos en la iteración 1000, **puede ocurrir que al ajustar la recta para clasificar bien un punto, deje de clasificar otros puntos bien**, finalizando por tanto con un error de clasificación superior. Esto se debe a que el algoritmo no tiene forma de saber **cómo de buena es la clasificación que realiza en cada iteración**, y únicamente distingue entre incorrecta y correctamente clasificados.

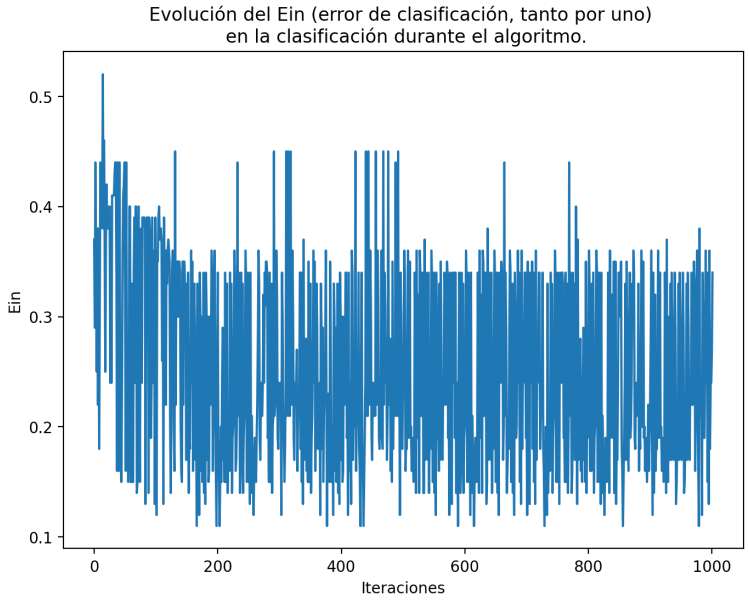


Figura 6: Evolución del valor de E_{in} conforme avanza el número de iteraciones del algoritmo PLA.
Datos con ruido.

Vemos en el siguiente gráfico el fenómeno que acabamos de describir anteriormente: el E_{in} oscila de forma significativa de una iteración a otra y nunca llega a converger. Podemos concluir que este algoritmo no es adecuado cuando tenemos datos que no son linealmente separables.

Si representamos gráficamente el ajuste de esta recta en comparación a la recta original, comprobamos lo anteriormente comentado:

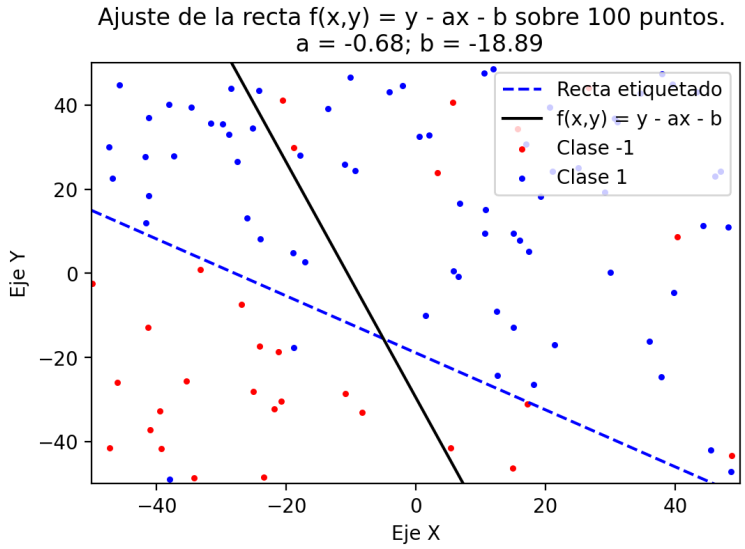


Figura 7: Ajuste del perceptrón (negro) vs. recta que etiqueta los puntos (azul)
Datos con ruido.

El algoritmo perceptrón ha acabado generando esta recta, que de lejos es la mejor, en esta última iteración.

Es por eso por lo que se diseñó una modificación sutil de este algoritmo, el algoritmo **PLA-Pocket**.
En la siguiente sección se comentarán sus principales diferencias y se comparará con la versión original, **PLA**.

3.2. Algoritmo PLA-Pocket (variante del perceptrón)

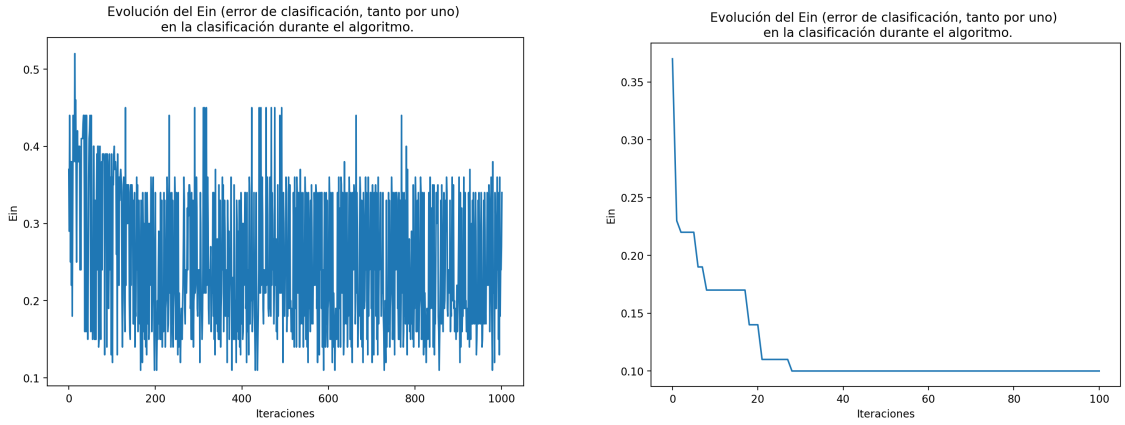
Esta versión del algoritmo PLA clásico tiene una pequeña modificación para evitar lo comentado al final del apartado anterior y consiste en mantener almacenado el vector de pesos w_{best} que menor valor de clasificación, E_{in}, da para nuestros datos. El vector de pesos w_{best} es aquel que se mantiene y se devuelve al final del algoritmo. Coloquialmente se podría decir que es el "mejor vector de pesos".

El funcionamiento es exactamente el mismo con la sutil diferencia de que, antes de avanzar a la siguiente iteración, el algoritmo evalúa el error E_{in} con el vector **actual** y lo compara con w_{best}; en caso de que sea mejor (**w** tenga menor error de clasificación, este nuevo **w** se convierte en el w_{best}) [1].

Podemos ver que esta variante, computacionalmente es más costosa debido a que tiene que evaluar la función de error en cada iteración, pero cuenta con el plus de que **almacenamos el mejor valor de w**, evitando depender del valor de la última iteración como en el PLA clásico.

Para ver gráficamente y entender mejor lo que pasa con el perceptrón y lo que corrige este algoritmo, se ha procedido a hacer el experimento del apartado anterior con un vector de pesos $w = [0., 0., 0.]$ y se han ejecutado ambos algoritmos, con un máximo de 1000 iteraciones.

A continuación se ha graficado el avance del error E_{in} obtenido en cada iteración, así como el valor de la *accuracy (en nuestro caso, el tanto por uno de aciertos)* de ambos algoritmos. **El conjunto de datos tiene ruido.** Los gráficos son los siguientes:

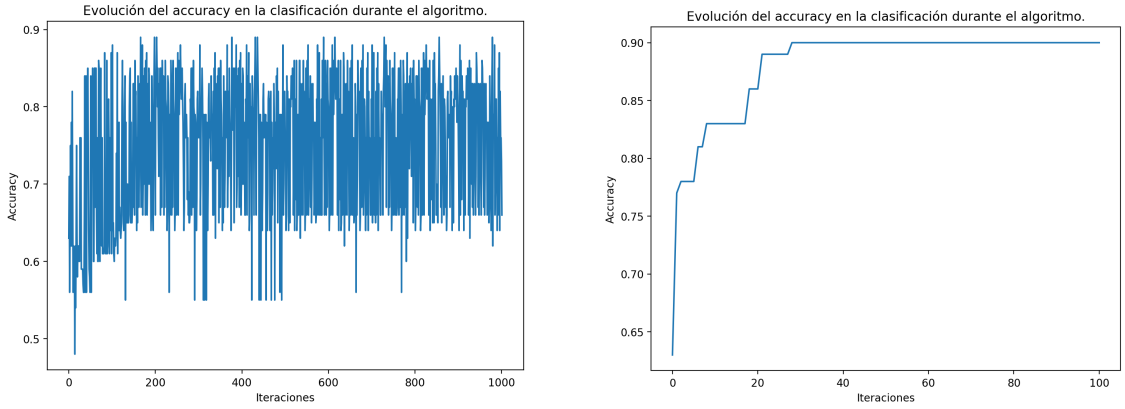


(a) Evolución de E_{in} con PLA

(b) Evolución de E_{in} con PLA-pocket

Figura 8: Comparativa de la evolución del error de clasificación, o E_{in}, en ambos algoritmos.

El comportamiento gráfico es tal y como hemos comentado anteriormente. El algoritmo **PLA-pocket** almacena el vector de pesos con el que menor valor de E_{in} ha obtenido, a diferencia del **PLA**.
Si observamos el comportamiento de ambos algoritmos midiendo la *accuracy o precisión*, los resultados son análogos:



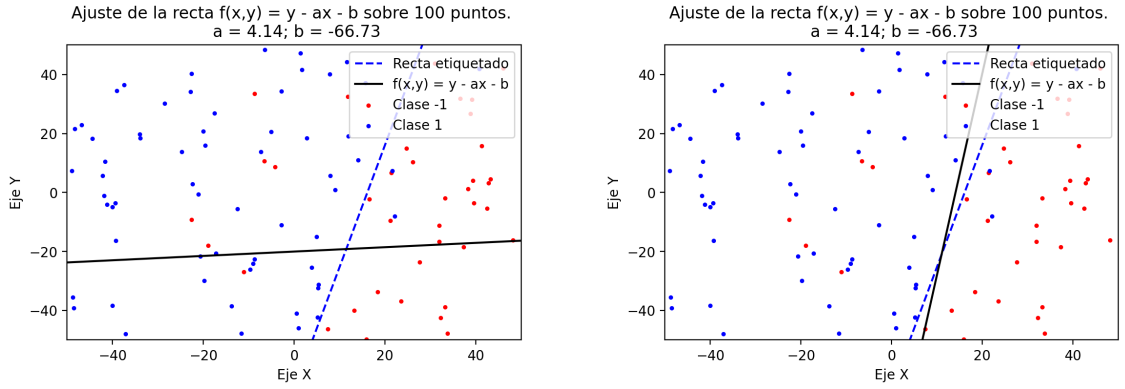
(a) Evolución de *accuracy* con PLA

(b) Evolución de *accuracy* con PLA-pocket

Figura 9: Comparativa de la evolución de la *accuracy*, o tanto por uno de aciertos, en ambos algoritmos.

Podemos ver que el **PLA** es más "sólido"porque no depende de la última iteración **al almacenar el mejor vector de pesos**, a diferencia del **PLA**, cuyos resultados varían mucho de una iteración a otra debido a que no sigue ningún criterio para almacenar el vector de pesos **w**.

Por último, veremos el ajuste de ambas rectas y cuantificaremos el error de clasificación final obtenido.



(a) Ajuste con **PLA**.
$E_{in} = 0.29$

(b) Ajuste con **PLA-pocket**.
$E_{in} = 0.10$

Figura 10: Comparativa del ajuste de la recta del algoritmo PLA vs PLA-Pocket.
Punto de inicio = $[0., 0., 0.]$

El ajuste de la recta por el algoritmo **PLA** obtiene un error de clasificación final de 0.29, y como podemos ver gráficamente, la recta que ajusta los datos no es la mejor ni de lejos. Esto se debe a lo comentado anteriormente, quizás en la iteración anterior la recta era muy buena clasificando los puntos y en la última iteración se ha generado este vector de pesos **w (que es la recta obtenida)** que no clasifica del todo bien los datos.

Sin embargo, con el algoritmo **PLA-Pocket** obtenemos un error de clasificación del 10 %, es decir, la recta que mejor clasifica los datos teniendo en cuenta el ruido, que es imposible eliminar. Esto se debe a que en alguna de las iteraciones, el vector de pesos ha conseguido obtener este error de clasificación y **el algoritmo ha almacenado el vector de pesos w correspondiente, que es el mostrado en la imagen**, ya que no hay otro mejor.

Por último, comentar que el algoritmo **PLA-pocket**, a pesar de necesitar evaluar E_{in} cada iteración, obtiene mejores resultados en **100** iteraciones en comparación al **PLA**, que continúa ciclando tras las **1000** iteraciones sin converger a un valor en concreto.

3.3. Regresión Logística (RL)

En este apartado queremos ver cómo funciona la regresión logística en su caso más simple, estimar una probabilidad [0, 1].
Consideraremos como espacio de características X = [0,2] × [0,2] y como etiquetas el espacio Y =-1,1, de manera que la asignación de las etiquetas sea una función que depende de las características.

Para ello, elegiremos una recta en el plano que pase por X como frontera entre la región en la que la probabilidad es 0 **(y = -1)**, y la región en la que la probabilidad es 1 **(y = +1)**.

El algoritmo de aprendizaje de regresión logística consiste en estimar unos pesos w para la hipótesis $h(x) = \theta(w^T x)$, donde en nuestro caso, θ es llamada *función logística*, siendo la siguiente:

$$\theta(s) = \frac{e^s}{1 + e^s} \tag{2}$$

Su salida es un número entre [0,1], número que interpretaremos como **una probabilidad**.
Al adaptar esta probabilidad a un problema de clasificación binaria, estableciendo el umbral a 0.5, podemos ver que:

$$y = 1 \Longleftrightarrow \theta(w^T x) \geq 0.5 \tag{3}$$

y, análogamente:

$$y = -1 \Longleftrightarrow \theta(w^T x) \leq 0.5 \tag{4}$$

Por tanto podemos seguir con el enfoque de los ejercicios anteriores y etiquetar los puntos según su distancia a una recta.
La implementación de la función logística se encuentra en la función `logistic_sgd`, y en ella se minimiza el error logístico:

$$E_{in}(w) = \frac{1}{N} \sum_{n=1}^{N} log(1 + e^{-y_n w^T x_n}) \tag{5}$$

Para minimizar este error logístico se utiliza el *gradiente descendiente estocástico*, algoritmo de aprendizaje que observamos en la práctica anterior.

Un breve resumen de lo que realiza este algoritmo es avanzar en la dirección y sentido de la máxima pendiente, **la del gradiente cambiado de signo**.
Al ser *estocástico*, utilizamos un subconjunto o *batch* de N datos, y calculamos el gradiente de la siguiente forma:

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

Los datos se barajan cada época (**cada vez que se recorren todos los batches del conjunto de datos**), y detenemos el algoritmo cuando **la norma de la diferencia de los pesos entre dos épocas consecutivas sea menor que 0.01**.

A continuación, se genera una muestra de 100 puntos con `simula_unif` y los etiquetamos según el signo de la distancia a esta recta.
Aplicando el algoritmo con un *learning rate* de 1 y un *batch size* de 32 (**tras probar experimentalmente con otros valores de learning rate, y batch size, pero el algoritmo no convergía tan rápido hacia una solución ni se obtenían mejores resultados que estos**), los resultados son los siguientes:

- w_{final}: [-11.16, -8.69, 12.97]
- Iteración final: 643
- Error de clasificación (tanto por uno): 0.0
- Error E_{in} (error **cross-entropy**): 0.048

Como vemos, con el criterio de parada que hemos usado, en este caso, obtenemos un error de clasificación de 0; **se ha ejecutado 10 veces, obteniendo un valor muy cercano a 0 ($\approx 10^{-2}$) en todas las ejecuciones**. Lo que tenemos que apreciar es que el algoritmo consigue separar de manera correcta los datos, con bastante precisión. Gráficamente, el ajuste y comparativa con la recta original es el siguiente:

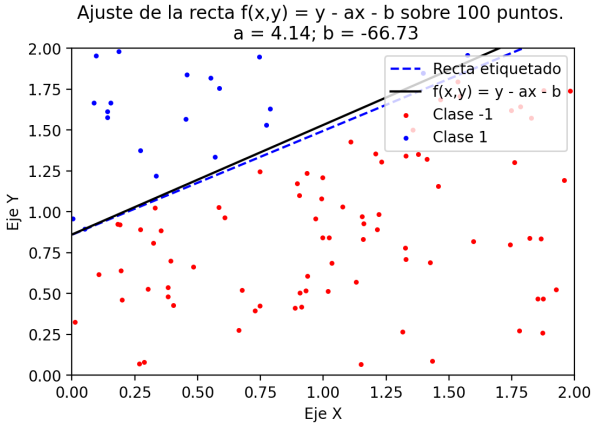


Figura 11: Ajuste de la recta generada por RL vs recta original (azul) sobre los datos de entrenamiento.
Datos sin ruido.

Como ya vimos numéricamente y comprobamos visualmente, el ajuste es muy cercano a la recta que etiqueta realmente los puntos.

Vamos a probar con un conjunto de **test** para ver cómo ajusta nuevos datos esta recta generada por el algoritmo y estimar E_{out}, el error de generalización, usando estos nuevos datos.
Para ello, generamos una muestra de **1000** puntos, los etiquetamos con la recta que definimos, y los clasificamos según la recta encontrada con regresión logística.

Los datos obtenidos son los siguientes:

- Error de clasificación (tanto por uno): 0.011
- Error E_{out} (error **cross-entropy**): 0.0426

Y, gráficamente, la recta obtenida ajusta los datos de test de esta forma:

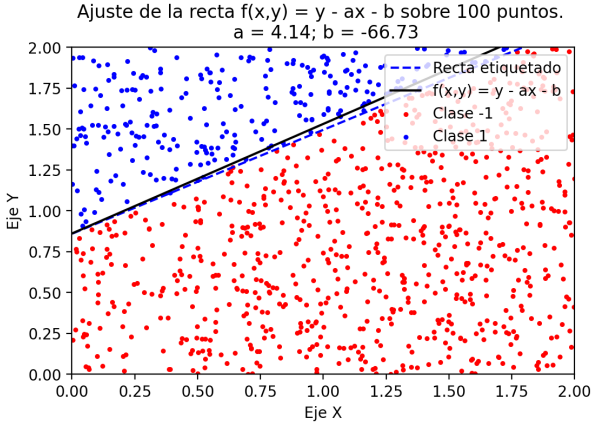


Figura 12: Ajuste de la recta generada por RL vs recta original (azul) sobre los datos de test. Datos sin ruido.

Como podemos ver, la generalización es bastante buena, tanto gráficamente como numéricamente: obtenemos un valor pequeño de E_{out} y clasificamos correctamente casi el 100 % de los datos (si bien es cierto que ambos errores son un poco más altos que en entrenamiento, pero eso es normal al tratar con datos nuevos).

A continuación, se procede a realizar este mismo experimento, pero repitiendo el experimento 100 veces, y calculando los valores promedio de E_{out}, de porcentaje de error de clasificación, y de épocas necesarias para converger, para llegar a una **conclusión más representativa del comportamiento del algoritmo**.
Tras ejecutar el algoritmo 100 veces y hacer la media de estas métricas, los resultados son los siguientes:

- Iteración final: 588
- Error E_{in} (error **cross-entropy** con los 100 datos de entrenamiento): 0.099
- Error de clasificación (tanto por uno) en **entrenamiento**: 0.006
- Error E_{out} (error **cross-entropy** con los 1000 datos de test): 0.057
- Error de clasificación (tanto por uno) en **test**: 0.0134

Como podemos ver, la **generalización** es bastante buena; hemos obtenido un valor bastante bajo de E_{out} y se han clasificado correctamente casi el 100 % de los puntos; podemos acabar diciendo que el algoritmo de **Regresión Logística** ha conseguido dar una buena solución a un problema de ajuste con modelos lineales.

4. Bonus. Clasificación binaria de dígitos

4.1. Descripción del problema

En este ejercicio se trabaja con un conjunto de datos de dígitos manuscritos, donde extraeremos, para los **dígitos 4 y 8**, información sobre la **intensidad promedio** y la simetría respecto al eje vertical, y, con esta información, se planteará un problema de clasificación binaria.

Por tanto, nuestro espacio de características X estará definido por datos $x_i = (i, s)$, donde ambos valores definen la intensidad y simetría del número, y seán las únicas características que lo definen. A cada dato x_i se le asocia una etiqueta y_i que formará parte del espacio de etiquetas Y, y queremos aproximar la verdadera función del etiquetado $f : X \to Y$ que es desconocida.

Para ello contamos con un conjunto de 1194 datos de entrenamiento con la que buscaremos el vector de pesos w adecuado, que separe ambas clases y funcione como una frontera de clasificación, y para buscarlo usaremos los algoritmos vistos en esta práctica, con la adición de un modelo de regresión lineal conocido como la *pseudoinversa*.

Para comparar el ajuste entre estos modelos contamos también con un conjunto de *test*, conjunto formado por 366 datos con sus etiquetas correspondientes, con el que estimaremos el valor de E_{test} y el error de clasificación en *test*, para ver el poder de generalización de todos nuestros algoritmos.

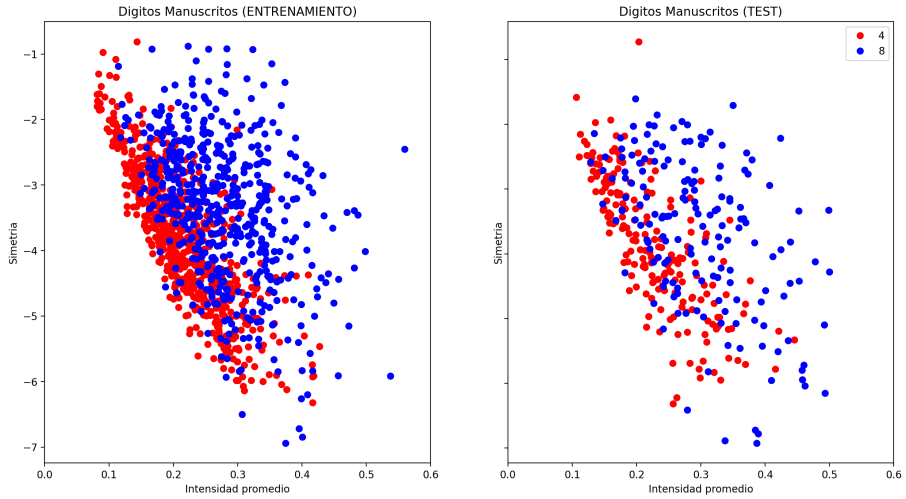


Figura 13: Representación de los datos de entrenamiento y de test, respectivamente. Las clases se indican en la leyenda.

4.2. Ajuste con los modelos

En esta sección se aplican los modelos de **regresión lineal (pseudoinversa), PLA, PLA-pocket, y Regresión Logística** y se calculan los valores de E_{in} y de E_{test}, así como el porcentaje de clasificación en entrenamiento y test, para cada uno de los modelos.
Primero se graficará el comportamiento de cada uno de los modelos y luego se generará una **tabla comparativa** de resultados.

4.2.1. Regresión lineal (pseudoinversa)

Para el método de la pseudoinversa obtenemos estos resultados:

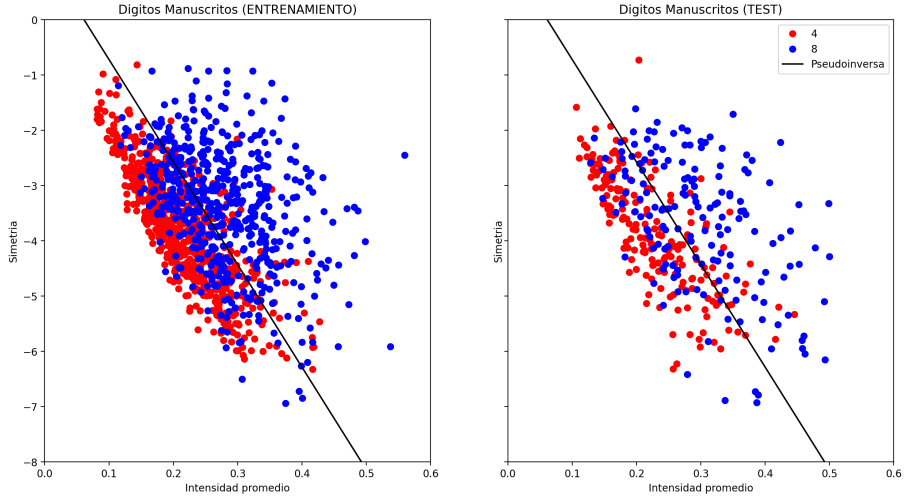


Figura 14: Ajuste de los datos de entrenamiento y de test, respectivamente, usando la **pseudoinversa**.
E_{in}: 0.642; E_{test}: 0.708

4.2.2. Algoritmo PLA

Al ver que los datos no son linealmente separables, podemos hacernos una idea de que el algoritmo PLA no es el mejor para clasificar los datos. Aun así, obtenemos, tras 100 iteraciones, el siguiente ajuste:

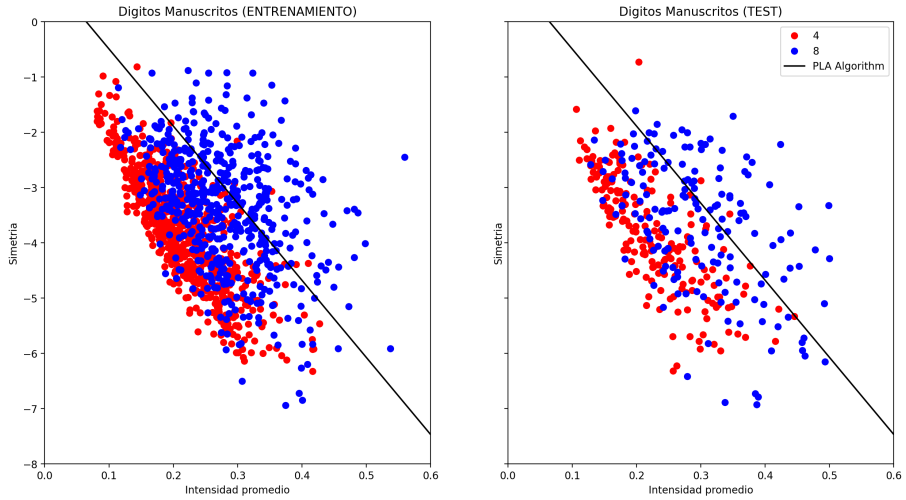


Figura 15: Ajuste de los datos de entrenamiento y de test, respectivamente, usando el algoritmo **PLA**.
E_{in}: 0.309; E_{test}: 0.301

4.2.3. Algoritmo PLA-pocket

El PLA-pocket es una buena alternativa al algoritmo PLA cuando los datos no son linealmente separables. Con 10 iteraciones, obtenemos este ajuste:

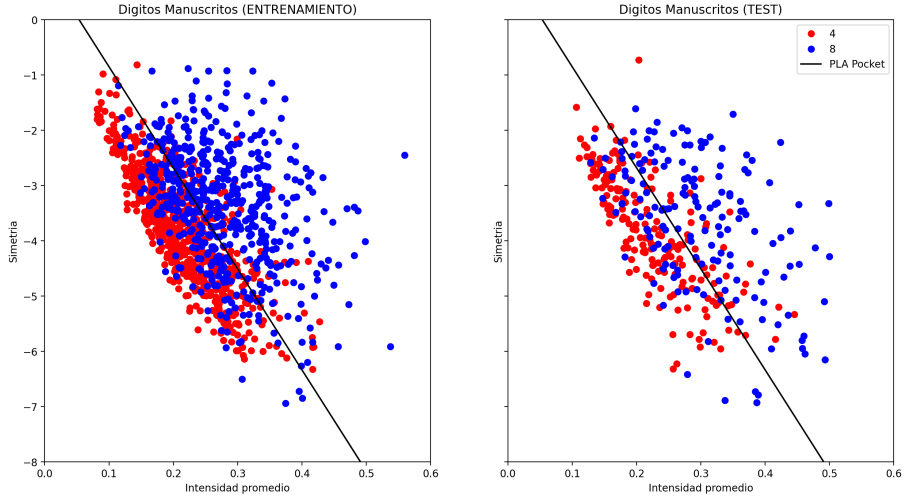


Figura 16: Ajuste de los datos de entrenamiento y de test, respectivamente, usando el algoritmo **PLA Pocket**.
E_{in}: 0.211; E_{test}: 0.251

4.2.4. Regresión Logística

Finalmente, escogiendo los mismos parámetros que en el apartado anterior, debido a su buen comportamiento con este tipo de datos, obtenemos estos resultados aplicando el algoritmo de Regresión Logística:

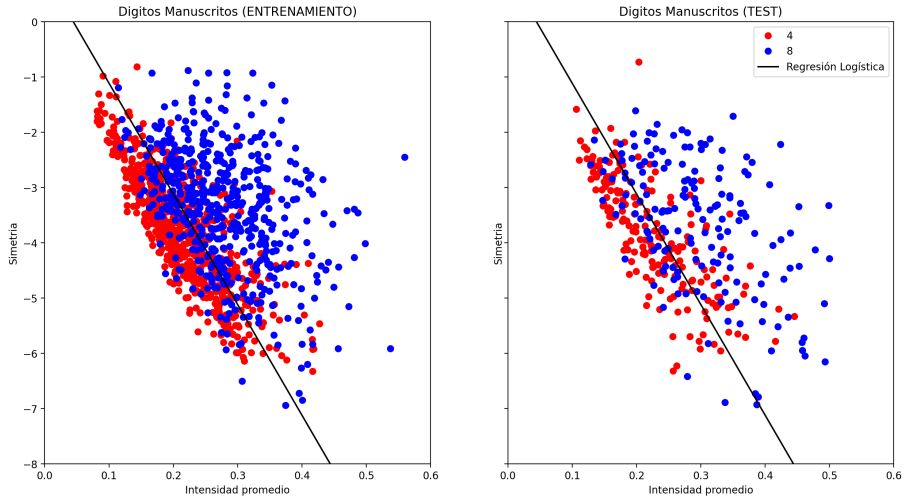


Figura 17: Ajuste de los datos de entrenamiento y de test, respectivamente, usando el algoritmo de **Regresión Logística**.
E_{in}: 0.504; E_{test}: 0.623

4.2.5. Tabla comparativa

A pesar de que ya hemos calculado el valor de E_{in} y E_{test} para todos los algoritmos, al tratarse de funciones de error diferentes, es más interesante ver el porcentaje de clasificación que cada una de estas ha obtenido, tanto en entrenamiento como en *test*. Por eso, se adjunta la siguiente tabla:

Error	Pseudoinversa	PLA	PLA-Pocket	RL
Entrenamiento	0.227	0.309	**0.211**	0.229
Test	**0.251**	0.300	**0.251**	0.295

Cuadro 5: Error de clasificación en entrenamiento, y test, de cada una de las funciones. El error de clasificación está en tanto por uno.

Podemos inferir varias cosas de esta tabla. En primer lugar, el algoritmo PLA es el que peores resultados obtiene, tanto en entrenamiento como en test, siendo el que peor clasifica los datos. Sin embargo, su variante **PLA-Pocket** es la que mejor comportamiento tiene en ambos conjuntos de datos, reduciendo el error de clasificación a alrededor de un 20 %. Vemos como una simple modificación puede cambiar el comportamiento de un algoritmo de una manera tan drástica, para el mismo conjunto de datos.
Comentar que el algoritmo más rápido es el de la Pseudoinversa, obteniendo resultados bastante buenos, sobre todo en *test*, y el algoritmo que más tarda en encontrar una solución debido a su criterio de parada es el algoritmo de **RL**, no destacando en ningún conjunto de datos.

4.3. Empleando los pesos de regresión lineal

Si se emplean los pesos obtenidos con regresión lineal para inicializar los otros tres métodos, conseguimos disminuir el error en el conjunto de entrenamiento, pero no el error en el conjunto de *test*.
Sin embargo, se obtiene una convergencia hacia la solución más rápida, ya que el método de la pseudoinversa es muy rápido y ayuda a encontrar la solución final, haciendo que el resto de algoritmos, al tomar este punto w_{pseudo} generado por la pseudoinversa comiencen, tras pocas iteraciones, a obtener valores de *accuracy* elevados en comparación a tomar un valor arbitrario, como el [0., 0., 0.].

Por tanto, quizás sea mejor darle un valor significativo al vector de pesos inicial en vez de generar uno de manera arbitraria o aleatoria. Si es cierto que se aporta una velocidad de convergencia mayor y se obtengan mejores resultados en el entrenamiento.

4.4. Cota de E_{out}

A continuación se calculan dos cotas para E_{out}: una basada en E_{in} y otra basada en E_{test}.
Se calculan ambas cotas para cada uno de los algoritmos, con una tolerancia de $\delta = 0.05$.

4.4.1. Cota usando E_{in}

Sabemos que, para un nivel de tolerancia $\delta \geq 0$, se tiene que

$$E_{out}(g) \leq E_{in}(g) + \sqrt{\frac{8}{N} log(\frac{4((2N)^{d_{VC}} + 1}{\delta})} \tag{7}$$

con probabilidad (1 - δ), y donde N es el tamaño del conjunto de entrenamiento y d_{VC} es la dimensión VC del set de hipótesis utilizado. En nuestro caso, en toda la práctica estamos tratando con un algoritmo de clasificación 2D de tipo perceptrón, cuya dimensión VC es 3 (visto en teoría).

4.5. Cota usando E_{test}

Al evaluar el conjunto de test con una sola hipótesis g, la final, podemos usar la desigualdad de Hoeffding para hallar una cota de generalización.
Siendo N el tamaño del conjunto de *test*, y $\delta \geq 0$, se tiene que

$$E_{out}(g) \leq E_{in}(g) + \sqrt{\frac{1}{2N} log(\frac{2|H|}{\delta})} \tag{8}$$

con probabilidad (1 - δ). En nuestro caso, $|H|$ es 1, ya que al calcular el error en el conjunto de test, hemos fijado la hipótesis **g** y no hemos explorado el conjunto entero; para eso están los datos de entrenamiento Con test sólo estamos evaluando.

4.6. Comparativa de cotas

Comparando ambas cotas, aplicadas a nuestros 4 algoritmos de aprendizaje, se obtiene la siguiente tabla:

Cota superior para E_{out}	Pseudoinversa	PLA	PLA-Pocket	RL
Usando E_{in}	1.073	0.739	0.641	0.935
Usando E_{test}	0.779	0.371	0.322	0.694

Cuadro 6: Cota superior para E_{out} usando E_{in} y E_{test}, aplicada a los 4 algoritmos de aprendizaje.
Recordemos que $E_{out} \leq$ cota.

Como podemos ver, **para todos los algoritmos de aprendizaje**, la cota obtenida con el conjunto de *test* y aplicando la desigualdad de Hoeffding es mucho más ajustada que usando el conjunto de entrenamiento y aplicando la desigualdad VC.

Este comportamiento es algo que no nos debe de extrañar, ya que, al calcular la cota para el conjunto de *test* lo estamos haciendo sobre datos que jamás hemos visto, lo cual acaba dándonos una cota **más representativa** que usando datos con los que hemos entrenado y hemos ajustado nuestras fronteras de decisión.

Por último, comentar que para el perceptrón (PLA y PLA-pocket), la medida de error E equivale al porcentaje de clasificados incorrectamente.
Con estas cotas, en su versión más optimista (usando E_{test}), podemos afirmar, con un 95 % de probabilidad, que el algoritmo PLA tendrá, como mucho, un error del 37.1 % clasificando nuevos datos, y el algoritmo PLA-pocket, como mucho un 32.2 %. Vemos que la variante pocket, de nuevo, supera al algoritmo base incluso para la generalización sobre nuevos datos.

Referencias

[1] Yaser S. Abu-Mostafa, Malik Magdon-Ismail y Hsuan-Tien Lin. *Learning From Data*. AML-Book, 2012.