Aprendizaje automático

Práctica 2: Modelos lineales

Antonio Coín Castro

Curso 2019-20

Ejercicio sobre la complejidad de $\mathcal{H}$ y el ruido

Este ejercicio se desarrolla en el script `p2_complejidad.py`. En él perseguimos estudiar las dificultades que introduce la aparición de ruido a la hora de elegir la clase de funciones más adecuada.

En primer lugar definimos un par de funciones para simular puntos. En la primera generamos un número determinado de puntos de forma uniforme en un hipercubo, es decir, en cada componente se extrae una muestra de una distribución uniforme. Recordamos que la distribución uniforme en un intervalo $[a, b]$ es aquella cuya función de densidad viene dada por

$$f_{\mathcal{U}}(x) = \frac{1}{b-a}, \quad a < x < b.$$

La probamos en dimensión 2 para poder visualizar la nube de puntos resultante. En este caso, generamos 50 puntos en el cuadrado $[-50, 50] \times [-50, 50]$, usando para ello la función `uniform_sample`.

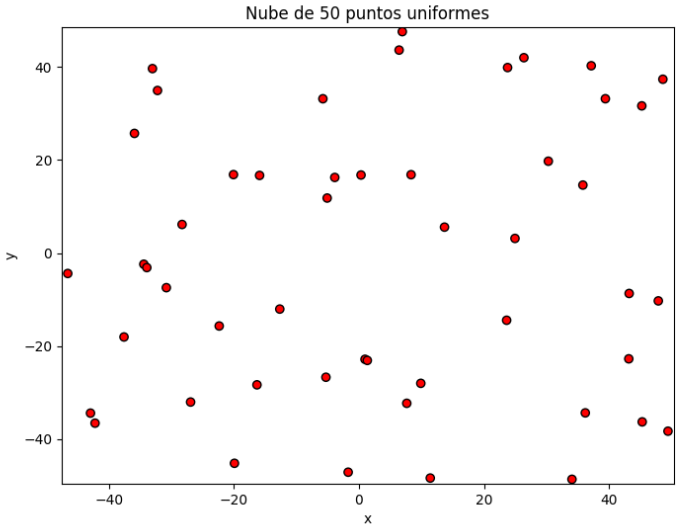


Figura 1: Nube de 50 puntos generados de forma uniforme en $[-50, 50]^2$.

La segunda función de simulación nos proporciona una forma de generar puntos que en cada componente provienen de una distribución normal *distinta*, cuya función de densidad es

$$f_{\mathcal{N}}(x;\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right).$$

Como en el caso anterior, visualizamos el resultado en dimensión 2, generando 50 puntos a partir de dos normales de media 0 y desviaciones típicas $\sigma_x = \sqrt{5}$ y $\sigma_y = \sqrt{7}$, respectivamente. Cabe esperar que los puntos se encuentren en la región $[-3\sigma_x, 3\sigma_x] \times [-3\sigma_y, 3\sigma_y]$, pues sabemos que es ahí donde se concentran casi el 100 % de los valores (Regla 68-95-99.7). Empleamos la función `gaussian_sample`.

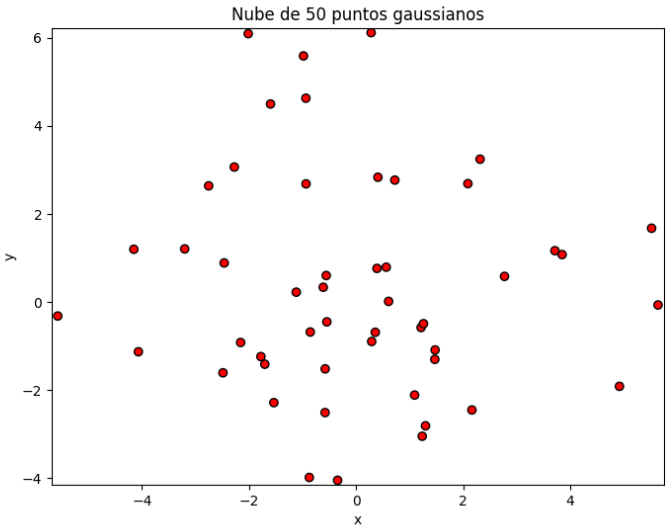


Figura 2: Nube de 50 puntos generados a partir de dos Gaussianas de media 0 y desviaciones típicas $\sigma_x = \sqrt{5}$ y $\sigma_y = \sqrt{7}$.

Procedemos ahora a hacer un pequeño experimento. Generamos 100 puntos de manera uniforme en el cuadrado $[-50, 50] \times [-50, 50]$, y empleando la función `line_sample` simulamos también de manera uniforme una recta que corte a dicho cuadrado. Esta recta queda determinada por su pendiente (a) y su punto de corte con el eje Y (b). Vamos a etiquetar los puntos generados en dos clases, utilizando la recta como frontera de clasificación. En particular, cada punto (x, y) se etiqueta a partir de la función

$$f_{a,b}(x, y) = \text{signo}(y - ax - b),$$

que no es más que el signo de la distancia del punto a la recta. Para evitar problemas suponemos que signo(0) = 1 (es decir, los puntos que están sobre la recta pertenecen a la clase del 1). Podemos ver en la Figura 3 cómo quedan los puntos generados y clasificados en función de la recta elegida. Es evidente que están todos bien clasificados, pues los hemos generado así.

Vamos ahora a introducir ruido en las etiquetas. Mediante la función `simulate_noise` perturbamos el signo de un 10 % de etiquetas positivas y otro 10 % de etiquetas negativas.

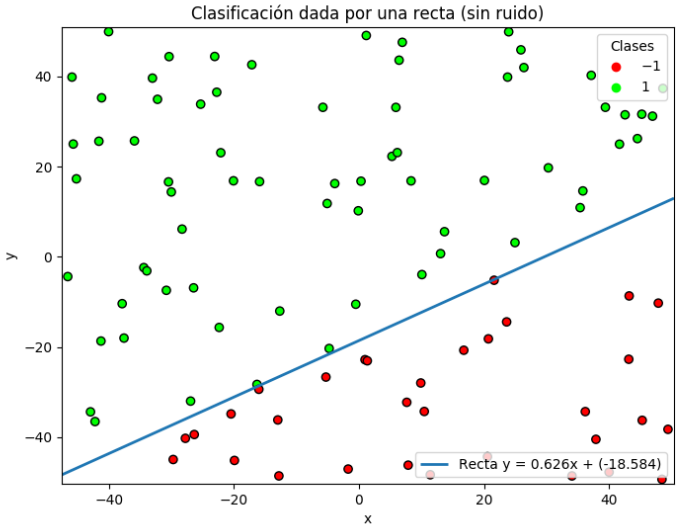


Figura 3: 100 puntos generados uniformemente etiquetados respecto de una recta.

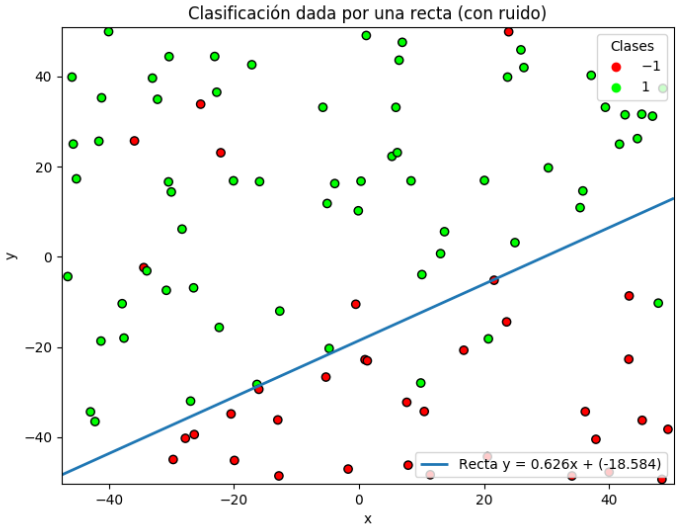


Figura 4: 100 puntos generados uniformemente etiquetados respecto de una recta y con ruido.

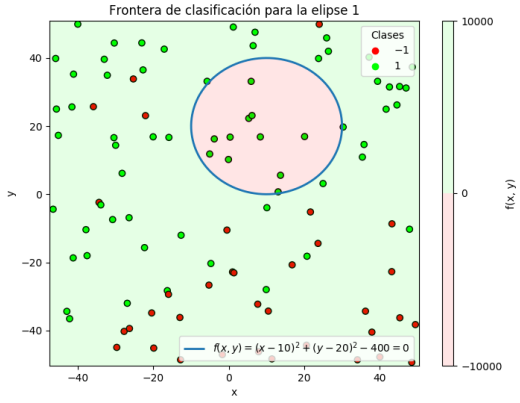
Si volvemos a dibujar los puntos con sus nuevas etiquetas, vemos en la Figura 4 cómo un 10 % de ellos están mal clasificados con respecto a la recta.

Queremos estudiar cómo se comportarían otros clasificadores más complejos a la hora de clasificar estos puntos con ruido, suponiendo que la función original que genera las etiqueta es $f_{a,b}$. Es decir, queremos ver si aumentando la complejidad de la clase de funciones $\mathcal{H}$ podemos obtener una mejor clasificación. Los clasificadores (cuadráticos) que consideramos son:

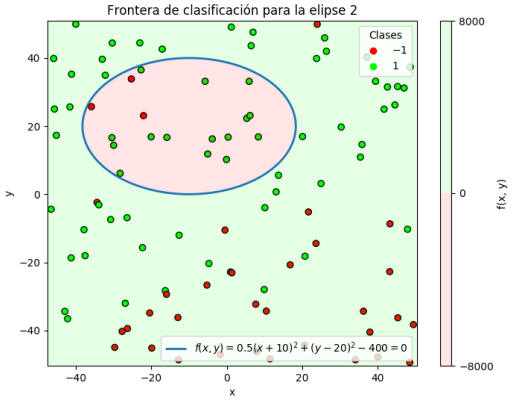
- (Elipse 1) $f(x, y) = (x - 10)^2 + (y - 20)^2 - 400$.

- (Elipse 2) $f(x, y) = 0.5(x + 10)^2 + (y - 20)^2 - 400$.
- (Hipérbola) $f(x, y) = 0.5(x - 10)^2 - (y + 20)^2 - 400$.
- (Parábola) $f(x, y) = y - 20x^2 - 5x + 3$.

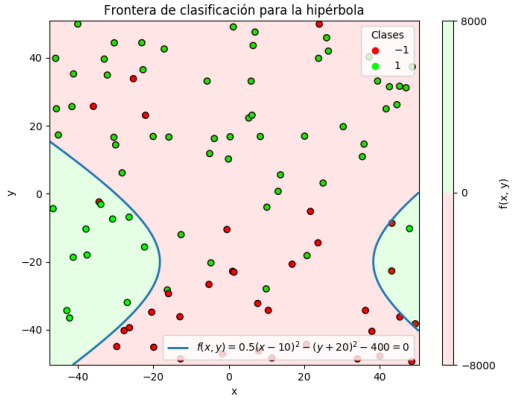
Mostramos en la Figura 5 las regiones positiva (en verde) y negativa (en rojo) de estos clasificadores binarios, notando que la frontera de todos ellos coincide con la curva de nivel $f(x, y) = 0$.



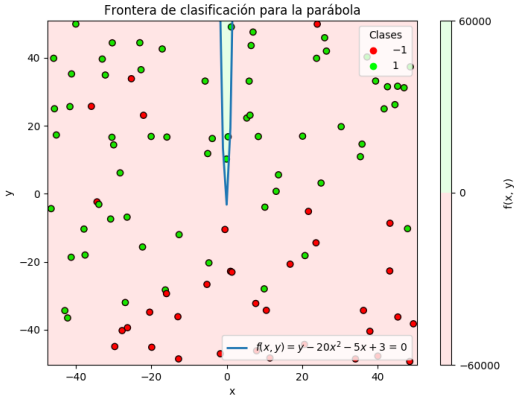
(a) Regiones de clasificación para la elipse 1.



(b) Regiones de clasificación para la elipse 2.



(c) Regiones de clasificación para la hipérbola.



(d) Regiones de clasificación para la parábola.

Figura 5: Regiones positiva y negativa para los distintos clasificadores.

Aunque a simple vista puede verse que ninguna de las cuatro funciones consigue clasificar siquiera de forma decente los datos, estudiamos un par de métricas que nos permitan justificar esto. Vamos a medir el *accuracy* de cada clasificador, es decir, el porcentaje de datos bien clasificados sobre el total. Además, como es posible que la distribución de etiquetas no esté balanceada, medimos también lo que se conoce como *balanced accuracy*, que corrige la medida de *accuracy* teniendo en cuenta la distribución de clases. Si P indica el número de ejemplos positivos, N el número de ejemplos negativos, TP el número de ejemplos positivos correctamente clasificados (*true positives*) y TN el número de ejemplos negativos correctamente clasificados (*true negatives*), tenemos que

$$\mathrm{ACC} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{P} + \mathrm{N}}, \quad \mathrm{BACC} = \frac{\mathrm{TP/P}}{2} + \frac{\mathrm{TN/N}}{2}.$$

Si calculamos estas dos métricas sobre todos nuestros clasificadores, usando las funciones `accuracy_score` y `balanced_accuracy_score` de `sklearn`, obtenemos lo siguiente:

```
Clasificador: recta
    Accuracy = 91.000%
    Balanced accuracy = 89.121%
Clasificador: elipse 1
    Accuracy = 54.000%
    Balanced accuracy = 41.538%
Clasificador: elipse 2
    Accuracy = 53.000%
    Balanced accuracy = 42.747%
Clasificador: hipérbola
    Accuracy = 43.000%
    Balanced accuracy = 53.516%
Clasificador: parábola
    Accuracy = 38.000%
    Balanced accuracy = 52.308%
```

Vemos cómo de los cuatro clasificadores cuadráticos las elipses son las que más *accuracy* consiguen, mientras que la hipérbola y la parábola obtienen una puntuación muy baja (como ya podíamos adivinar a partir de las gráficas). En nuestro caso la proporción de clases es del 65 % para la clase del 1 y del 35 % para la clase del -1, por lo que en la métrica balanceada los dos últimos clasificadores mejoran un poco. Sin embargo, ninguno de estos clasificadores cuadráticos consigue siquiera destacar ante un *clasificador aleatorio*, que obtendría de media un *accuracy* del 50 %, y no se acerca ni por asomo a los resultados obtenidos con el clasificador lineal.

Podemos concluir que en este caso el aumento en la complejidad del modelo no conduce a una mejora de resultados, pues aún habiendo introducido ruido en las etiquetas, el mejor clasificador sigue siendo el más sencillo: la función lineal.

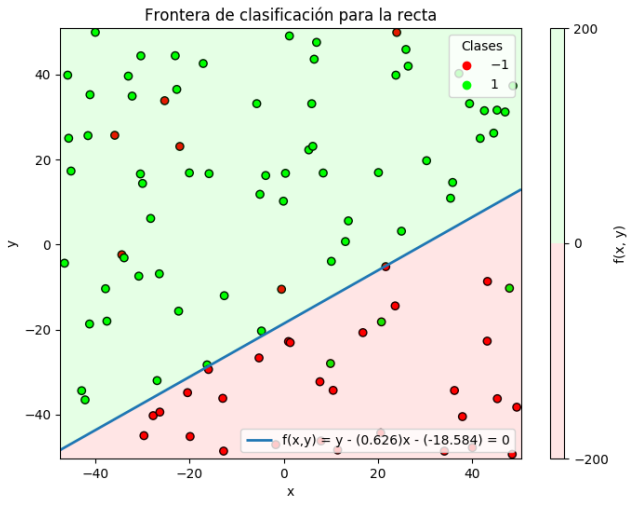


Figura 6: Regiones de clasificación para la recta.

El hecho de permitir que aumente la complejidad de la clase de funciones a elegir no siempre hace que aumente el rendimiento del modelo. De hecho, si intentásemos aprender un clasificador muy complejo (por ejemplo, un polinomio de grado elevado) que se ajustase a estos datos mejor que una recta, lo más probable es que cayéramos en el *sobreajuste* y dicho clasificador no consiguiera generalizar sobre nuevos puntos. Es por esto que en general debemos evitar que la presencia de ruido en los datos nos induzca a pensar que el modelo subyacente es mucho más complejo de lo que es en realidad.

Ejercicio sobre modelos lineales

En este ejercicio, desarrollado en el script `p2_lineales.py`, implementamos un par de modelos lineales para clasificación binaria.

Algoritmo Perceptrón (PLA)

En la función `fit_pla` se desarrolla el algoritmo perceptrón que calcula el hiperplano solución a un problema de clasificación binaria. Acepta como parámetros la matriz de datos en coordenadas homogéneas (con un 1 como primera componente), el vector de etiquetas reales, el número máximo de iteraciones y los pesos iniciales. Devuelve un vector de pesos w tal que la frontera de clasificación viene determinada por $w^T x = 0$.

La estrategia es simple: en cada iteración se realiza una pasada por el conjunto de datos completo. Para cada dato $x(t)$, si el vector de pesos actual $w(t)$ lo clasifica correctamente según la etiqueta $y(t)$ no hacemos nada, y si lo clasifica erróneamente (es decir, $\text{signo}(w(t)^T x(t)) \neq y(t)$), se actualizan los pesos mediante la fórmula

$$w(t+1) = w(t) + y(t)x(t).$$

El algoritmo se detiene cuando todos los datos están bien clasificados, o cuando alcanza el número máximo de iteraciones. Hemos añadido también una variable que va guardando la evolución temporal del vector de pesos, para poder hacer después una gráfica y analizarla.

Generamos ahora los mismos puntos que en el ejercicio anterior, y los volvemos a clasificar de acuerdo a la misma recta que teníamos antes. Consideramos ahora dos escenarios: etiquetas sin ruido (datos linealmente separables) y etiquetas con ruido (datos no linealmente separables).

Etiquetas sin ruido

En primer lugar, aplicamos el algoritmo a nuestros datos utilizando como vector inicial $w_0 = (0, 0, 0)^T$ y con un máximo de 1000 iteraciones. Como el algoritmo PLA es determinístico (para un mismo vector inicial y unos mismos datos devuelve siempre los mismos pesos), está claro que si lo ejecutamos varias veces vamos a obtener el mismo resultado:

```
Recta encontrada por PLA con vector inicial 0
    Iteraciones: 99
    Accuracy = 100.000%
    Balanced accuracy = 100.000%
```

Vemos que consigue clasificar perfectamente los datos de entrenamiento y se detiene antes de alcanzar el número máximo de iteraciones. Introducimos ahora un componente aleatorio, inicializando el vector de pesos de manera uniforme en $[0, 1]$ para cada componente. Como ahora sí que pueden variar los resultados, realizamos 10 ejecuciones y presentamos la media de los resultados:

```
Recta encontrada por PLA con vector inicial aleatorio (de media)
    Iteraciones: 96.7
    Accuracy = 100.000%
    Balanced accuracy = 100.000%
```

Observamos que de nuevo se clasifican perfectamente los datos, y de media necesitamos unas tres iteraciones menos para conseguirlo que con el vector inicial 0. No es ninguna sorpresa que consigamos clasificar todos los datos en ambos casos, pues sabemos que si los datos son linealmente separables, el algoritmo perceptrón encuentra siempre un hiperplano que los separa.

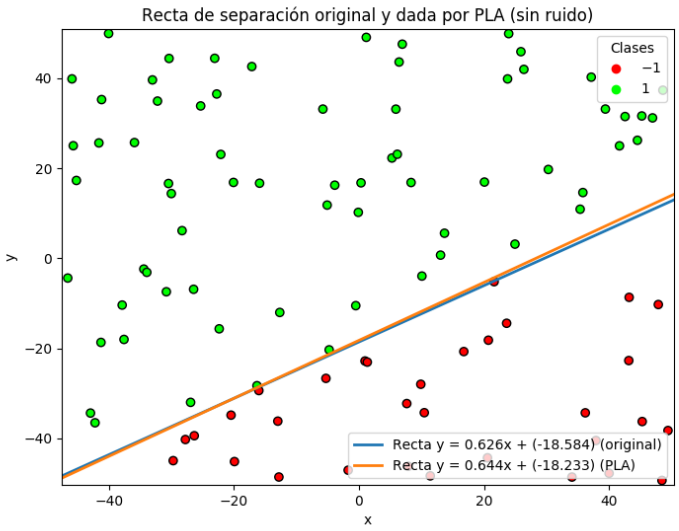


Figura 7: Recta original y obtenida por PLA con inicial vector aleatorio (sin ruido).

Concluimos que **para un mismo punto de inicio obtenemos siempre el mismo resultado**. Además, si los datos son linealmente separables existen infinitas rectas que los separan, y si cambiamos el punto inicial encontraremos una u otra, pero todas con idéntico error de clasificación: 0. En nuestra ejecución concreta, permitiendo que ese punto de inicio fuera aleatorio obtenemos una convergencia más rápida que fijándolo en el vector 0, pero es posible que esto sea debido a los datos concretos con los que estamos trabajando. Probando con otras semillas aleatorias se obtienen conclusiones contradictorias, por lo que no podemos afirmar nada con seguridad en cuanto a este comportamiento.

Mostramos por último en la Figura 8 una gráfica con la evolución temporal del *accuracy* a lo largo de las iteraciones del algoritmo, observando que aunque en iteraciones concretas pueda descender, la tendencia general es que va aumentando hasta llegar al 100 %.

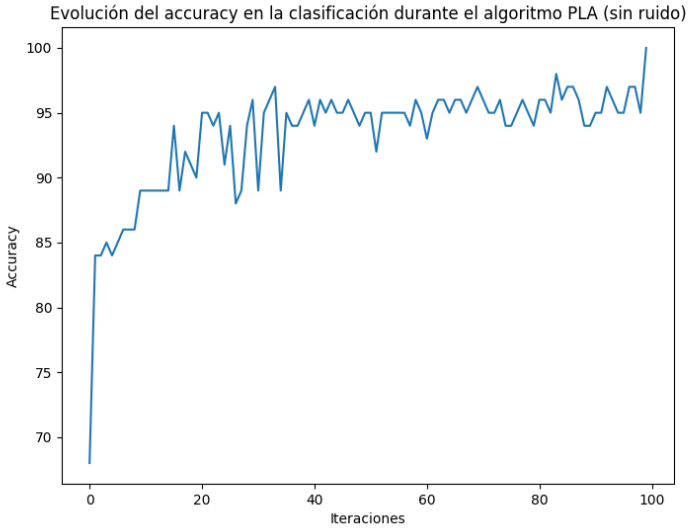


Figura 8: Evolución temporal del *accuracy* en PLA (sin ruido).

Etiquetas con ruido

Repetimos las ejecuciones del apartado anterior, pero esta vez utilizando las etiquetas con ruido del primer ejercicio. Obtenemos los siguientes resultados:

```
Recta original
    Accuracy = 91.000%
    Balanced accuracy = 89.121%
Recta encontrada por PLA con vector inicial 0
    Iteraciones: 1000
    Accuracy = 84.000%
    Balanced accuracy = 84.396%
Recta encontrada por PLA con vector inicial aleatorio (de media)
    Iteraciones: 1000.0
    Accuracy = 83.700%
    Balanced accuracy = 83.967%
```

Observamos que en este caso, tanto partiendo del vector 0 como partiendo de un vector aleatorio, siempre se consumen las 1000 iteraciones y no se consiguen clasificar de forma correcta los datos de entrenamiento. Esto es algo que ya sabíamos de forma teórica: si los datos no son linealmente separables, el algoritmo perceptrón no puede encontrar un hiperplano que los separe, y siempre realizará todas las iteraciones que le indiquemos. A partir de los resultados obtenidos y de la representación en la Figura 9 comprobamos que tampoco consigue clasificar los datos de forma *suficientemente buena* (es decir, como lo haría la recta original), pues por la propia estructura del algoritmo no tiene forma de saber cómo de buena es la clasificación que realiza en cada iteración (solo distingue entre clasificación perfecta e imperfecta).

Si visualizamos esta vez en la Figura 10 la evolución temporal del *accuracy* podemos observar el fenómeno que describíamos antes: como nunca se va a alcanzar la clasificación

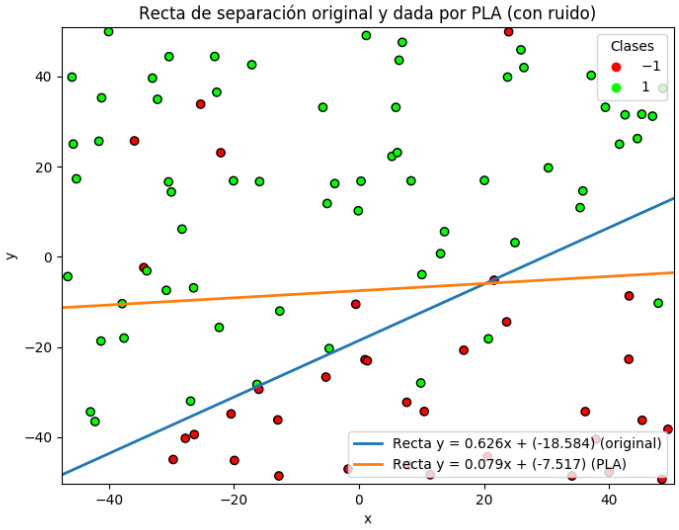


Figura 9: Recta original y obtenida por PLA con vector inicial aleatorio (con ruido).

perfecta, el *accuracy* 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**.

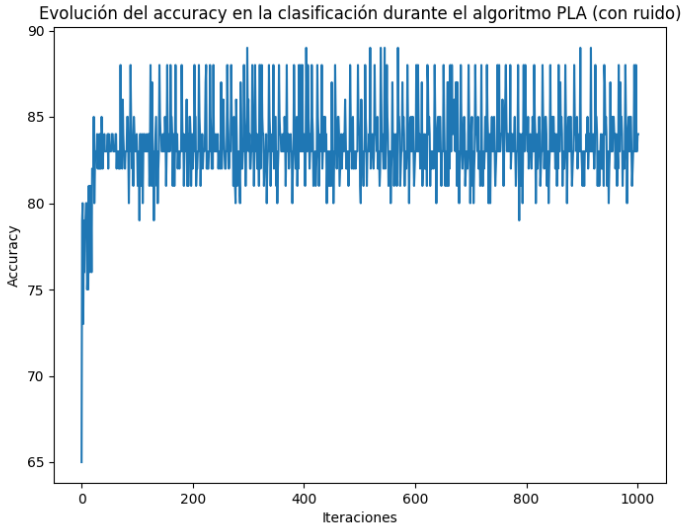


Figura 10: Evolución temporal del *accuracy* en PLA (con ruido).

Regresión logística

En este apartado queremos ver cómo funciona la regresión logística en el caso simple en el que estimamos una probabilidad 0/1. Consideramos como espacio de características $\mathcal{X} = [0, 2] \times [0, 2]$ y como etiquetas $\mathcal{Y} = \{-1, 1\}$, de forma que la asignación de las mismas sea una función determinista de las características. Elegimos una recta en el plano que

pase por $\mathcal{X}$ como frontera entre la región en la que la probabilidad es 0 ($y = -1$) y la región en la que la probabilidad es 1 ($y = 1$).

La regresión logística consiste en estimar unos pesos w para la hipótesis $h(x) = \theta(w^T x)$, donde θ es la *función logística*:

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

Sabemos que el resultado es un número en $[0, 1]$, que interpretamos como una probabilidad. Si queremos adaptarlo a un problema de clasificación binaria establecemos un *umbral* en 0.5, e interpretamos que $y = 1 \iff \theta(w^T x) \geq 0.5$. Se puede comprobar que esta última condición equivale a que $w^T x \geq 0$, por lo que podemos seguir con el enfoque de los ejercicios anteriores y etiquetar los puntos según el signo de su distancia a una recta.

Para implementar la regresión logística utilizamos el algoritmo de minimización iterativa SGD, desarrollado en la función `logistic_sgd`, que consiste en minimizar el *error logístico*:

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

Para ello avanzamos en la dirección y sentido de máxima pendiente: la del gradiente cambiado de signo. Para el cálculo de este gradiente utilizamos un subconjunto (*batch*) de los datos, siendo su expresión general para N datos

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

Barajamos los datos tras cada época (cada pasada completa por los datos), y detenemos el algoritmo cuando la norma de la diferencia de los pesos entre dos épocas consecutivas sea menor que 0.01.

Ahora, generamos una muestra de 100 puntos y los etiquetamos según el signo de la distancia a la recta elegida. A continuación aplicamos el algoritmo con un *learning rate* de 0.01, y un tamaño de batch de 1 (se ha probado con otros tamaños más elevados pero no se obtenían buenos resultados). Los resultados son los siguientes:

```
Iteraciones: 501
E_in = 0.128
Accuracy en training = 98.000%
Balanced accuracy en training = 96.667%
```

Observamos que debido al criterio de parada impuesto no conseguimos un *accuracy* del 100 %. Sin embargo, es un resultado suficientemente bueno, ya que prácticamente clasificamos bien todos los puntos. Pasamos a estimar el error de generalización E_{out} generando una nueva muestra de 1000 puntos y clasificándolos según la recta encontrada con regresión logística. Los resultados son:

```
E_out = 0.119
Accuracy en test = 99.500%
Balanced accuracy en test = 99.284%
```

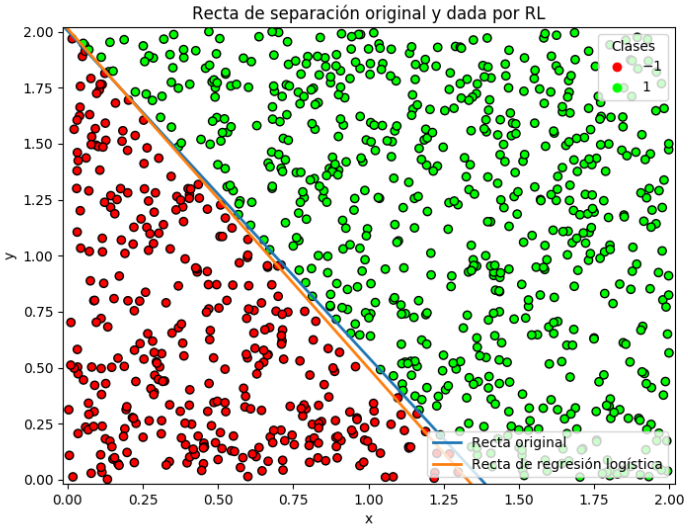


Figura 11: Conjunto de test junto a la recta de clasificación original y la encontrada con regresión logística.

Como podemos ver la generalización es bastante buena, obteniendo un valor pequeño para E_{out} y clasificando correctamente casi el 100 % de los puntos. Concluimos que hemos conseguido adaptar con éxito el modelo de regresión logística para realizar clasificación binaria.

Ejercicio sobre clasificación de dígitos (bonus)

Este ejercicio se desarrolla en el script `p2_digitos.py`, donde trabajamos sobre los mismos datos de dígitos manuscritos de la práctica anterior. Extraemos para los dígitos 4 y 8 información sobre la intensidad promedio y la simetría respecto al eje vertical, y planteamos con esta información un problema de clasificación binaria.

El espacio de características es $\mathcal{X} = \{1\} \times \mathbb{R}^2$, y el espacio de etiquetas considerado es $\mathcal{Y} = \{-1, 1\}$, entendiendo que el -1 corresponde al dígito 4 y el 1 al dígito 8. Queremos estimar la verdadera función de etiquetado $f : \mathcal{X} \to \mathcal{Y}$, que es desconocida. Para ello, consideramos nuestro conjunto de datos de entrenamiento junto con sus verdaderas etiquetas:

$$\mathcal{D} = \{(x_n, y_n) \subseteq \mathcal{X} \times \mathcal{Y} : n = 1, \dots N\}, \qquad \text{con } N = 1194.$$

Suponemos que hay una distribución de probabilidad $\mathcal{P}$ (desconocida) en $\mathcal{X} \times \mathcal{Y}$, y que los elementos de $\mathcal{D}$ se extraen de forma independiente y son idénticamente distribuidos respecto de $\mathcal{P}$. Nuestro objetivo es entonces apender un función que estime a f, dentro de un conjunto de posibles funciones candidatas que en nuestro modelo es:

$$\mathcal{H} = \{h : \mathbb{R}^3 \to \mathbb{R} \mid h(x) = \text{signo}(w^T x),\ w \in \mathbb{R}^3\}.$$

Utilizamos la técnica de *empirical risk minimization*, empleando como medida de error la pérdida 0-1. El problema se reduce entonces a encontrar una función $g \in \mathcal{H}$ de forma que

minimice el error de clasificación, a saber:

$$E_{in}(h) = \frac{1}{N}\sum_{n=1}^{N}[[h(x_n) \neq y_n]], \quad h \in \mathcal{H}.$$

Notamos que basta buscar el vector de pesos w adecuado, que representa una recta en el plano y define la frontera de clasificación entre las clases. Vamos a usar como algoritmos de aprendizaje primero un modelo de regresión lineal y después intentar una mejora con el algoritmo PLA-Pocket. Mediremos la bondad del modelo calculando el error en el conjunto de test (E_{test}), que consta de 366 datos en las mismas condiciones que los de $\mathcal{D}$.

Como modelo de regresión lineal elegimos el método de la pseudoinversa, que ya lo estudiamos en la práctica anterior. Si lo aplicamos a los datos de entrenamiento obtenemos lo siguiente:

```
---- Pseudoinversa
Vector de pesos = [-0.507 8.251 0.445]
Errores:
    E_in = 0.22781
    E_test = 0.25137
```

Si representamos los datos observamos que no son linealmente separables, por lo que es un problema adecuado para utilizar el algoritmo PLA-Pocket. Este es solo una mejora del algoritmo perceptrón que retiene el mejor vector de pesos encontrado hasta el momento (el que obtiene el menor error de clasificación en toda la muestra), y solo lo actualiza si el nuevo vector de pesos proporciona un error menor. De esta forma nos aseguramos de que la calidad de la solución no empeora conforme avanzan las iteraciones. Se implementa en la función `pla_pocket`, y en este caso el único criterio de parada es el número máximo de iteraciones (sabemos que los datos no son linealmente separables, por lo que nunca va a conseguir clasificarlos todos correctamente).

Lo ejecutamos con 500 iteraciones, primero usando como vector inicial un vector aleatorio con componentes uniformes en $[0, 1]$ y depués usando como vector inicial el vector w_{pseudo} que obtenemos tras aplicar el método de la pseudoinversa. Los resultados obtenidos son:

```
---- PLA-Pocket (aleatorio)
Vector de pesos = [-8.014 140.224 8.222]
Errores:
    E_in = 0.22864
    E_test = 0.24863
---- PLA-Pocket (pseudoinversa)
Vector de pesos = [-6.507 94.333 4.884]
Errores:
    E_in = 0.22529
    E_test = 0.25410
```

Mostramos a continuación dos gráficas de los conjuntos de entrenamiento y test junto a las tres rectas estimadas.

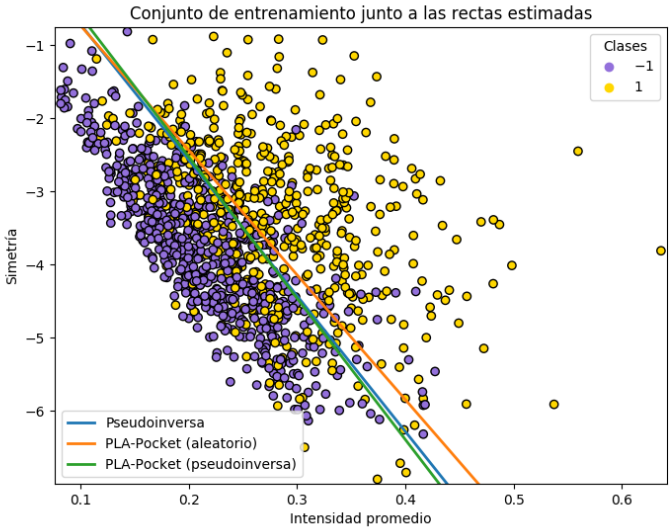


Figura 12: Conjunto de entrenamiento junto a las rectas estimadas.

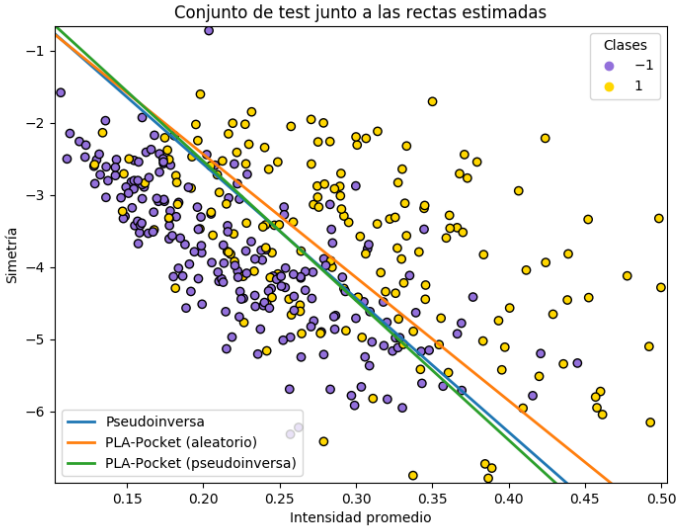


Figura 13: Conjunto de test junto a las rectas estimadas.

Vemos cómo en el caso aleatorio obtenemos un E_{in} ligeramente más grande que con la pseudoinversa, pero el error en el conjunto de test es más pequeño, lo cual es bueno. En el caso de utilizar el vector w_{pseudo} como punto inicial conseguimos disminuir el error de entrenamiento, pero no el error en test. Sin embargo, la mejora que obtenemos partiendo de unos pesos iniciales "buenos" es que la convergencia es más rápida, y aunque no siempre ocurra, es probable que podamos mejorar la clasificación obtenida.

Para comprobar empíricamente la última afirmación sobre la velocidad de convergencia realizamos unas gráficas de evolución temporal como las que hicimos en el segundo ejercicio para el algoritmo perceptrón. En este caso ya sabemos de antemano que nunca empeoraremos.

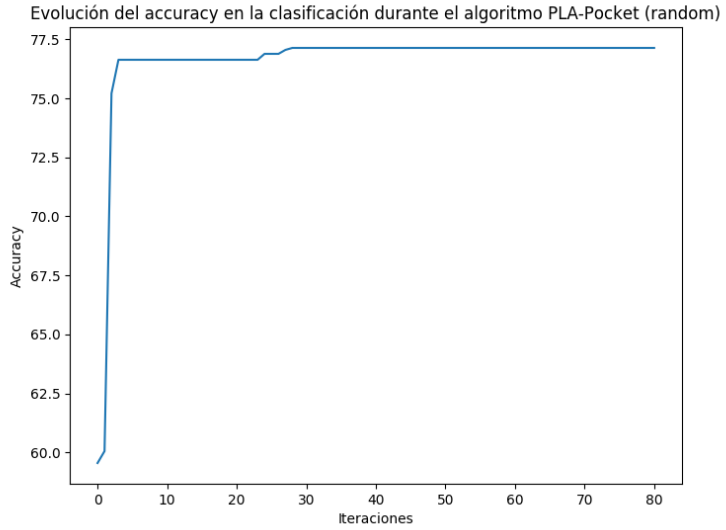


Figura 14: Evolución del *accuracy* con las iteraciones en PLA-Pocket (aleatorio).

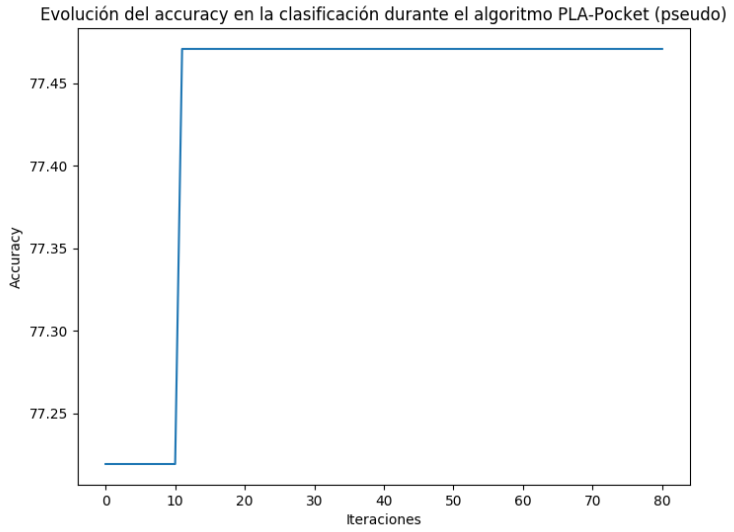


Figura 15: Evolución del *accuracy* con las iteraciones en PLA-Pocket (pseudoinversa).

Observamos cómo efectivamente la convergencia se alcanza antes partiendo de los pesos encontrados por el método de regresión lineal, que ya comienzan en un *accuracy* elevado.

Pasamos por último a calcular unas cotas para el error fuera de la muestra (E_{out}), ya conocida nuestra hipótesis elegida g. Para esto tenemos tenemos dos enfoques distintos.

Cota usando la dimensión VC

Sabemos que, para cualquier nivel de tolerancia $\delta > 0$, se tiene

$$E_{out}(g) \leq E_{in}(g) + \sqrt{\frac{8}{N} \log \left(\frac{4((2N)^{d_{\text{VC}}} + 1)}{\delta} \right)}, \quad \text{con probabilidad } \geq 1 - \delta,$$

donde N es el tamaño del conjunto de entrenamiento y d_{VC} es la dimensión VC del clasificador utilizado. En nuestro caso estamos ante un perceptrón 2D, cuya dimensión VC sabemos que es 3. Implementamos este cálculo en la función `err_bound_vc`, y calculamos las cotas sobre nuestras tres hipótesis anteriores, tomando como tolerancia $\delta = 0.05$.

```
---- Pseudoinversa
Cota para E_out usando E_in (VC) = 0.65874
---- PLA-Pocket (aleatorio)
Cotas para E_out usando E_in (VC) = 0.65958
---- PLA-Pocket (pseudoinversa)
Cotas para E_out usando E_in (VC) = 0.65623
```

Cota utilizando la desgiualdad de Hoeffding

La desigualdad de Hoeffding nos proporciona también una cota de generalización. Si N es el tamaño del conjunto de entrenamiento y $\delta > 0$, tenemos que

$$E_{out}(g) \leq E_{in}(g) + \sqrt{\frac{1}{2N} \log \frac{2|\mathcal{H}|}{\delta}}, \quad \text{con probabilidad } \geq 1 - \delta.$$

En nuestro caso tenemos un inconveniente, pues $|\mathcal{H}| = \infty$. Sin embargo, como estamos trabajando en un ordenador con precisión finita, podemos discretizar el espacio y estimar el cardinal de $\mathcal{H}$ como el número de rectas distintas que podemos representar con tres parámetros. Como cada flotante ocupa 64 bits, tenemos que $|\mathcal{H}| \approx 2^{64 \cdot 3}$.

Por otro lado, la cota de Hoeffding también se puede aplicar utilizando E_{test} en lugar de E_{in}. En este caso N sería el número de puntos de test, pero la gran diferencia es que podemos tomar $|\mathcal{H}| = 1$. Esto se debe a que al calcular el error en el conjunto de test ya hemos fijado nuestra hipótesis g, por lo que **el conjunto de posibles candidatas se reduce a ella misma**.

Calculamos las cotas para E_{out} de estas dos maneras, a través de la función `err_bound_hoeffding`, y obtenemos:

```
---- Pseudoinversa
Cotas para E_out:
    Cota usando E_in (Hoeffding) = 0.46713
    Cota usando E_test (Hoeffding) = 0.32236
---- PLA-Pocket (aleatorio)
Cotas para E_out:
    Cota usando E_in (Hoeffding) = 0.46797
```

```
    Cota usando E_test (Hoeffding) = 0.31962
---- PLA-Pocket (pseudoinversa)
Cotas para E_out:
    Cota usando E_in (Hoeffding) = 0.46462
    Cota usando E_test (Hoeffding) = 0.32509
```

Como vemos, estas cotas son más finas que las que nos proporcionaba la dimensión VC. Además, dentro de estas dos la que utiliza E_{test} es mejor, ya que nos da una cota más ajustada sobre el error de generalización. Esto no es de extrañar, ya que al calcular el error sobre el conjunto de test lo estamos haciendo sobre datos que no hemos visto previamente, lo cual es más representativo que si usamos datos con los que hemos entrenado y a los que nos hemos podido ajustar demasiado.