

# 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`.

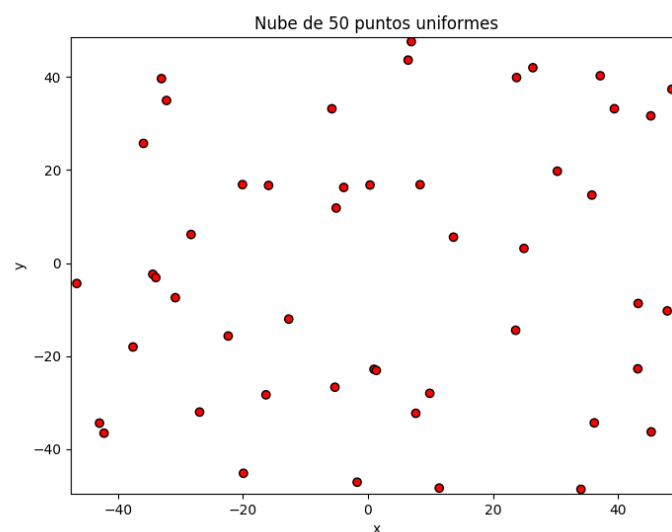


Figura 1: Nube de 50 puntos generados de forma uniforme en  $[-50, 50]^2$ .

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`.

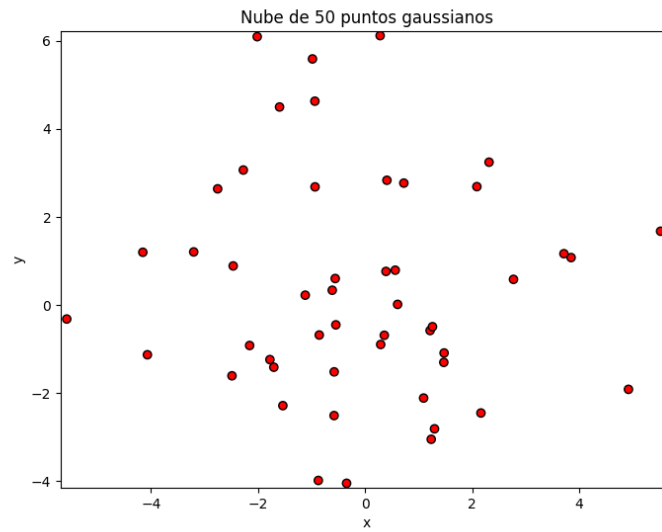


Figura 2: Nube de 50 puntos generados a partir de dos Gaussianas de media 0 y desviaciones típicas  $\sigma_x = \sqrt{5}$  y  $\sigma_y = \sqrt{7}$ .

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  $\text{signo}(0) = 1$  (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.

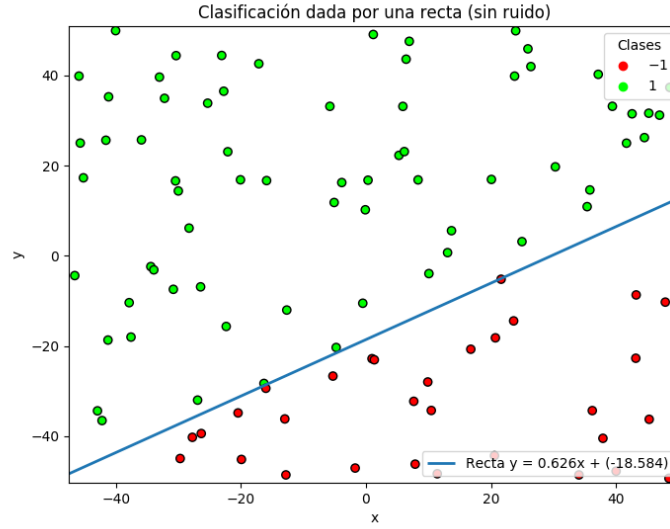


Figura 3: 100 puntos generados uniformemente etiquetados respecto de una recta.

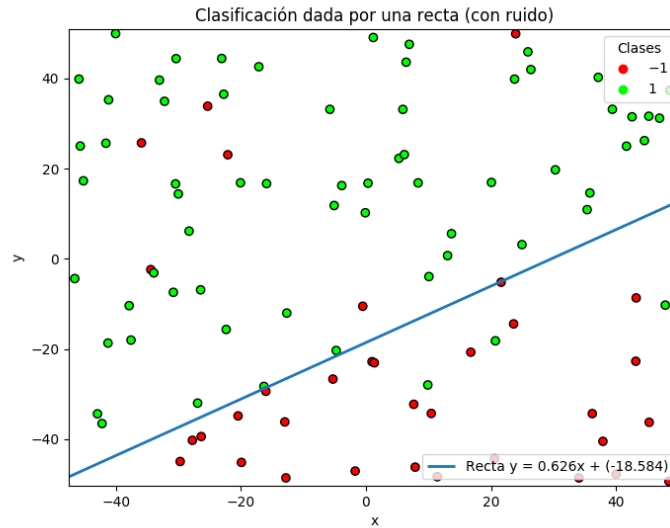


Figura 4: 100 puntos generados uniformemente etiquetados respecto de una recta y con ruido.

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  $f(x, y) = 0$ .

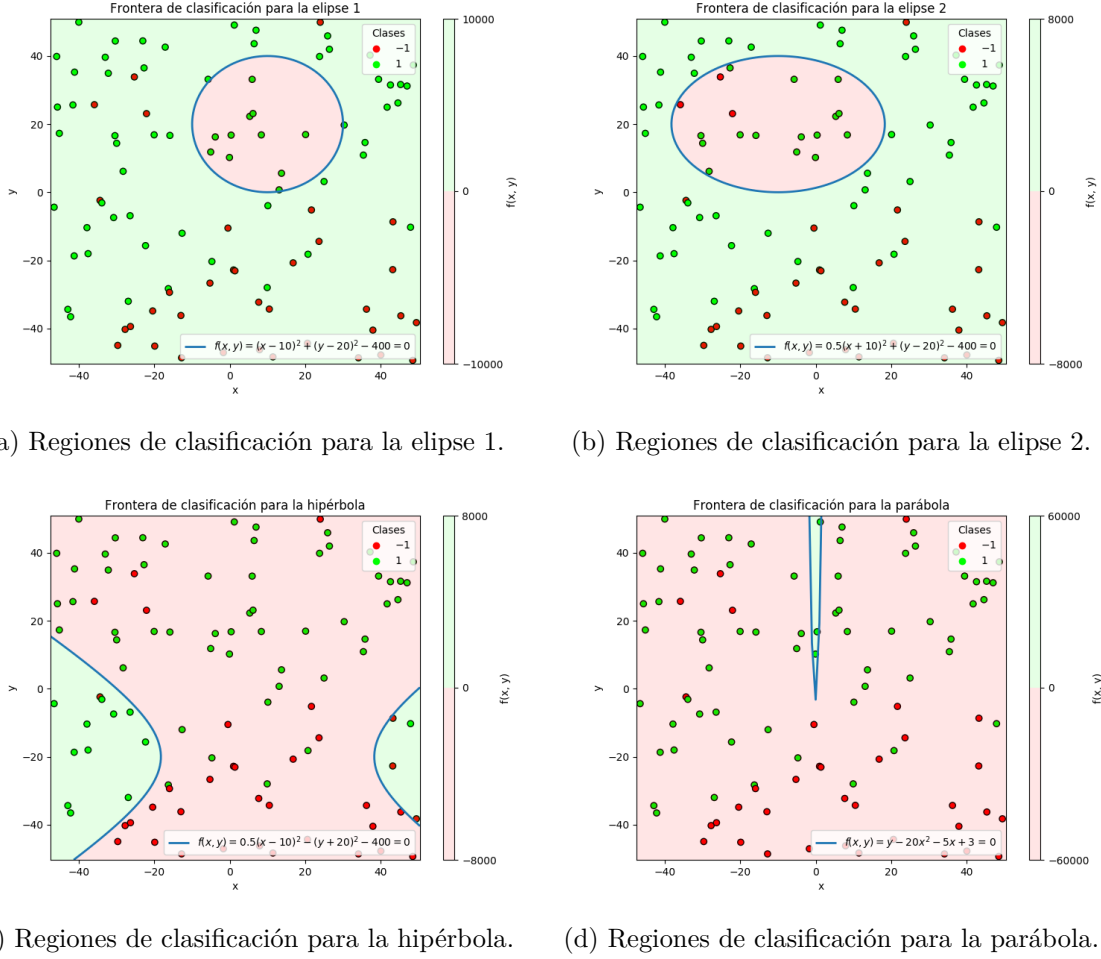


Figura 5: Regiones positiva y negativa para los distintos clasificadores.

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

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

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 al tener una región negativa bastante mayor. 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.

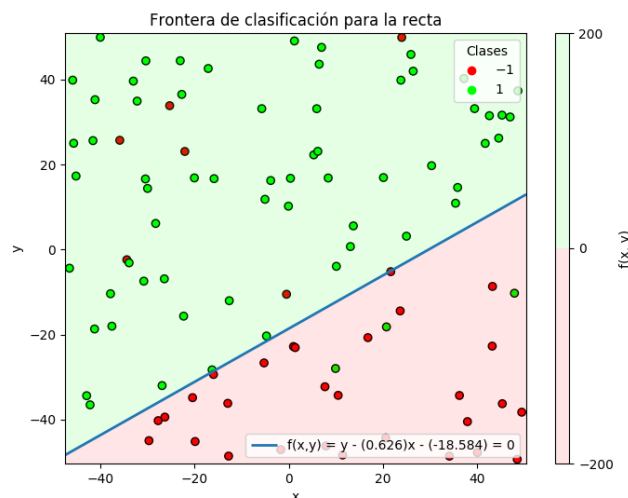


Figura 6: Regiones de clasificación para la recta.

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 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.

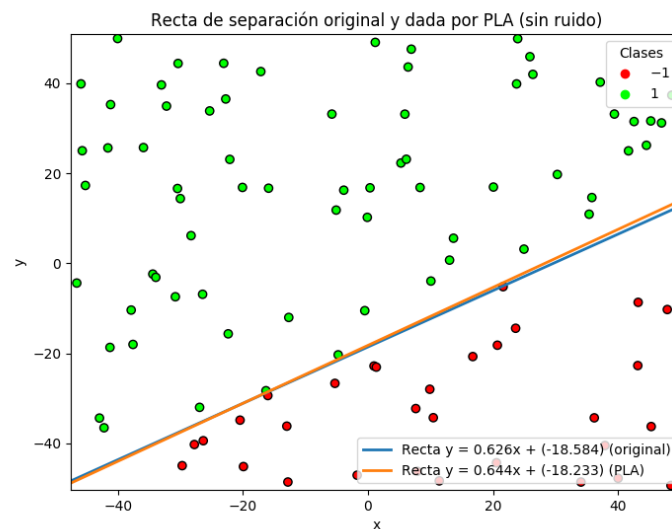


Figura 7: Recta original y obtenida por PLA con inicial vector aleatorio (sin ruido).

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%.

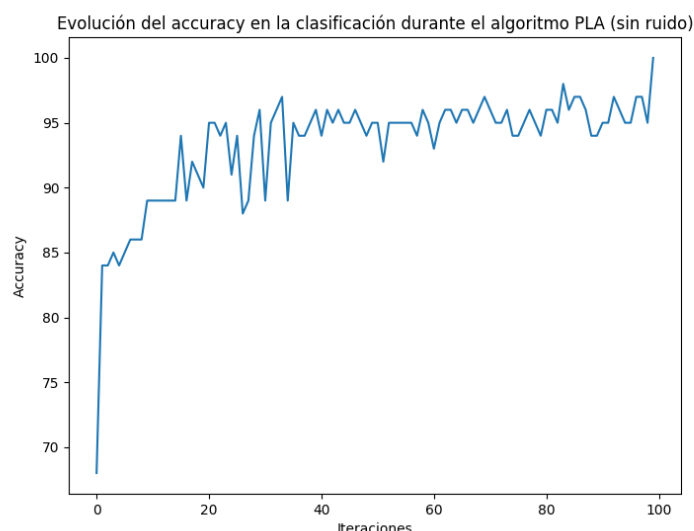


Figura 8: Evolución temporal del *accuracy* en PLA (sin ruido).

### 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 9 la evolución temporal del *accuracy* podemos observar el fenómeno que describíamos antes: como nunca se va a alcanzar la clasificación



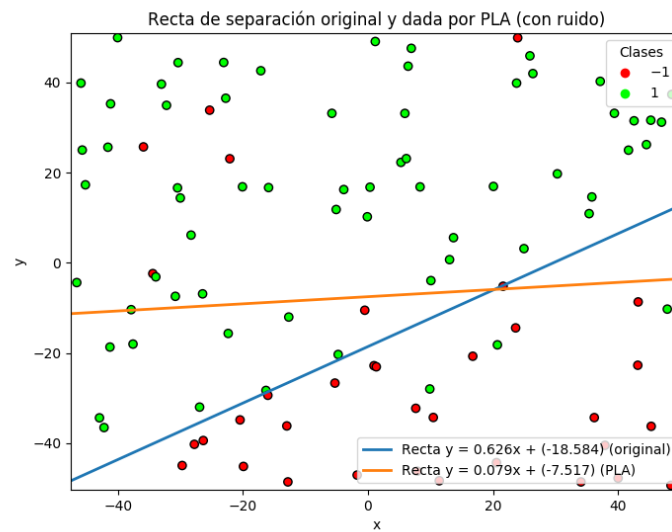


Figura 9: Recta original y obtenida por PLA con vector inicial aleatorio (con ruido).

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**.

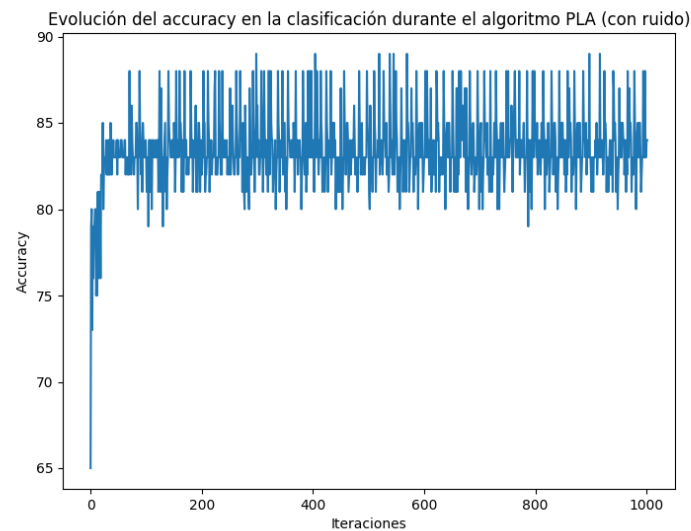


Figura 10: Evolución temporal del *accuracy* en PLA (con ruido).

## RL

- SGD con  $bs=1$ , si no converge (se ha probado) Instead of reading everything and then correct yourself at the end, you correct yourself on the way, making the next reads more useful since you correct yourself from a better guess.

### --- EJERCICIO 2: REGRESIÓN LOGÍSTICA ---

Iteraciones: 501  
E\_in = 0.128  
E\_out (en 1000 nuevos puntos) = 0.119  
Accuracy en training = 98.000%  
Balanced accuracy en training = 96.667%  
Accuracy en test = 99.500%  
Balanced accuracy en test = 99.284%

## Bonus

-1 -> 4 1 -> 8

- sabemos que los datos no son linealmente separables (por eso usamos pocket, que es más lento que solo PLA), así que el número de iters es fijo.
- Se usa como ein el error de clasificación
- Cotas: usando VC y usando hoeffding. Los puntos de test < puntos de training.
- mostrar gráfica con la evolución de pla para contrastar

#### ---- Pseudoinversa

Vector de pesos = [-0.507 8.251 0.445]  
Errores:  
E\_in = 0.22781  
E\_test = 0.25137  
Cotas para E\_out:  
Cota usando E\_in (VC) = 0.65874  
Cota usando E\_in (Hoeffding) = 0.46713  
Cota usando E\_test (Hoeffding) = 0.32236

#### ---- PLA-Pocket (aleatorio)

Vector de pesos = [-8.014 140.224 8.222]  
Errores:  
E\_in = 0.22864  
E\_test = 0.24863  
Cotas para E\_out:  
Cota usando E\_in (VC) = 0.65958  
Cota usando E\_in (Hoeffding) = 0.46797  
Cota usando E\_test (Hoeffding) = 0.31962

#### ---- PLA-Pocket (pseudoinversa)

Vector de pesos = [-6.507 94.333 4.884]  
Errores:  
E\_in = 0.22529  
E\_test = 0.25410  
Cotas para E\_out:  
Cota usando E\_in (VC) = 0.65623  
Cota usando E\_in (Hoeffding) = 0.46462  
Cota usando E\_test (Hoeffding) = 0.32509