



# UNIVERSIDAD DE GRANADA

APRENDIZAJE AUTOMÁTICO  
GRADO EN INGENIERÍA INFORMÁTICA

---

## PRÁCTICA 2

### PROGRAMACIÓN

---

#### **Autor**

Vladislav Nikolov Vasilev

#### **Rama**

Computación y Sistemas Inteligentes



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE  
TELECOMUNICACIÓN

CURSO 2018-2019

## Índice

<b>1. EJERCICIO SOBRE LA COMPLEJIDAD DE <math>H</math> Y EL RUIDO</b>	<b>2</b>
Apartado 1 . . . . .	2
Apartado 2 . . . . .	3
Apartado 3 . . . . .	6
<b>2. MODELOS LINEALES</b>	<b>9</b>
Apartado 1 . . . . .	9

## 1. EJERCICIO SOBRE LA COMPLEJIDAD DE H Y EL RUIDO

En este ejercicio debemos aprender la dificultad que introduce la aparición de ruido en las etiquetas a la hora de elegir la clase de funciones más adecuada. Haremos uso de tres funciones ya programadas:

- *simula\_unif*( $N, dim, rango$ ), que calcula una lista de  $N$  vectores de dimensión  $dim$ . Cada vector contiene  $dim$  números aleatorios uniformes en el intervalo  $rango$ .
- *simula\_gaus*( $N, dim, sigma$ ), que calcula una lista de longitud  $N$  de vectores de dimensión  $dim$ , donde cada posición del vector contiene un número aleatorio extraído de una distribución Gaussiana de media 0 y varianza dada, para cada dimension, por la posición del vector  $sigma$ .
- *simula\_recta*( $intervalo$ ), que simula de forma aleatoria los parámetros,  $v = (a, b)$  de una recta,  $y = ax + b$ , que corta al cuadrado  $[-50, 50] \times [-50, 50]$ .

### Apartado 1

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

- a) Considere  $N = 50$ ,  $dim = 2$ ,  $rango = [-50, +50]$  con *simula\_unif*( $N, dim, rango$ ).

Como el código de la función *simula\_unif* se ha proporcionado, no se va a mostrar su funcionamiento porque se supone que ya es conocido. Habiendo dicho esto, se procede a mostrar los datos generados:

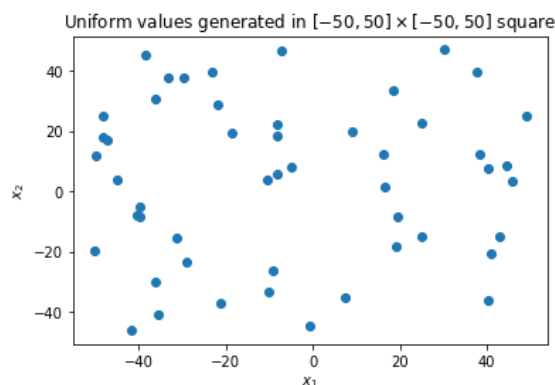


Figura 1: Datos generados por la función *simula\_unif*( $N, dim, rango$ ) con  $N = 50$ ,  $dim = 2$ ,  $rango = [-50, +50]$

b) Considere  $N = 50$ ,  $dim = 2$  y  $sigma = [5, 7]$  con  $simula\_gaus(N, dim, sigma)$ .

De nuevo, como en el caso anterior, como ya se ha proporcionado la función  $simula\_gaus$ , no se va a mostrar su funcionamiento porque se supone que ya es conocido. Así que, con esto en mente, podemos mostrar los siguientes puntos generados:

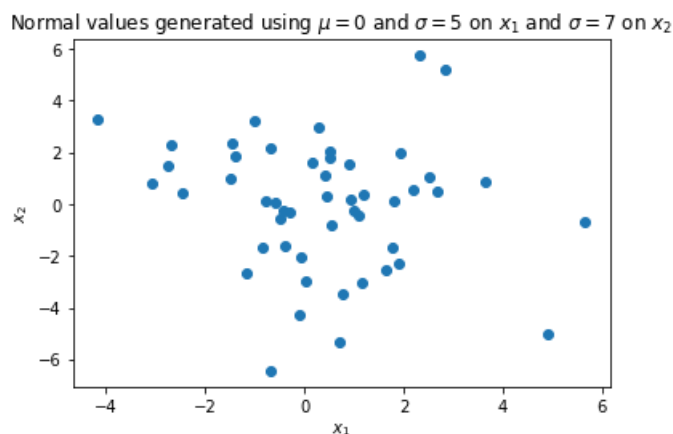


Figura 2: Datos generados por la función  $simula\_gaus(N, dim, sigma)$  con  $N = 50$ ,  $dim = 2$  y  $sigma = [5, 7]$

## Apartado 2

Con ayuda de la función  $simula\_unif()$  generar una muestra de puntos 2D a los que vamos añadir una etiqueta usando el signo de la función  $f(x, y) = y - ax - b$ , es decir el signo de la distancia de cada punto a la recta simulada con  $simula\_recta()$ .

a) Dibujar una gráfica donde los puntos muestren el resultado de su etiqueta, junto con la recta usada para ello. (Observe que todos los puntos están bien clasificados respecto de la recta).

Para realizar esto, se nos ha proporcionado la función  $simula\_recta()$  desde el principio, con lo cuál no la vamos a comentar, ya que se supone conocida. Para clasificar los puntos, vamos a utilizar una función llamada  $f(x, y, a, b)$ , donde  $x$  es la coordenada  $X$  del punto,  $y$  la coordenada  $Y$ ,  $a$  la pendiente de la recta simulada anteriormente y  $b$  el termino independiente. Esta función también se ha proporcionado, con lo cuál no se mostrará su implementación porque, de nuevo, se supone conocida.

Los puntos generados, junto con la recta, son los siguientes:

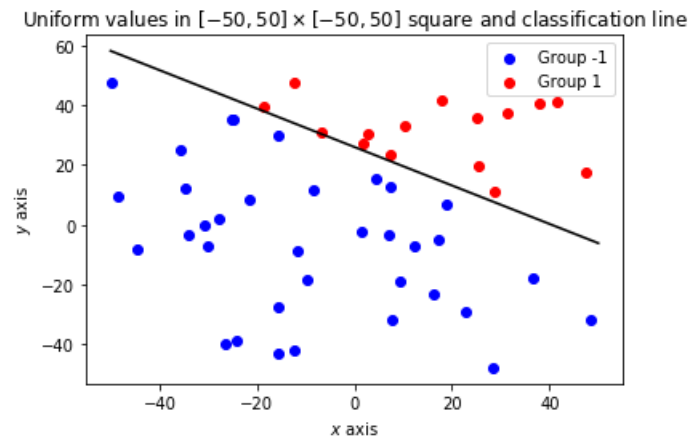


Figura 3: Gráfica de puntos generados uniformemente con la recta que separa las dos clases.

Adicionalmente, para ver que la clasificación es la correcta, se ha calculado el ratio de puntos clasificados correcta e incorrectamente mediante una función, la cuál se puede ver a continuación:

```
In [5]: def error_rate(x, y, a, b):
        """
        Funcion para calcular los ratios de acierto y error entre los valores
        reales de las etiquetas y los predichos.

        :param x: Array de vectores de características
        :param y: Array de etiquetas
        :param a: Pendiente de la recta
        :param b: Valor independiente de la recta

        :return Devuelve el ratio de aciertos y el ratio de errores
        """

        # Crear lista de y predichas
        predicted_y = []

        # Predecir cada valor
        for value in x:
            predicted_y.append(f(value[0], value[1], a, b))

        # Convertir a array
        predicted_y = np.array(predicted_y)

        return np.mean(y == predicted_y), np.mean(y != predicted_y)
```

Figura 4: Función para el cálculo del ratio de aciertos y errores.

Los resultados obtenidos, como se espera en este caso ya que no hay ruido en la muestra, son los siguientes:

```
In [7]: # Obtener ratios de acierto y error
accuracy, error = error_rate(x, y, a, b)

print('Ratio de aciertos: {}'.format(accuracy))
print('Ratio de error: {}'.format(error))

Ratio de aciertos: 1.0
Ratio de error: 0.0
```

Figura 5: Ratios de acierto y error obtenidos de la muestra sin ruido.

- b) Modifique de forma aleatoria un 10 % etiquetas positivas y otro 10 % de negativas y guarde los puntos con sus nuevas etiquetas. Dibuje de nuevo la gráfica anterior. (Ahora hay puntos mal clasificados respecto de la recta)

Para insertar ruido en la muestra, de forma proporcional a la cantidad de etiquetas de las distintas clases que tenemos, nos hemos ayudado de la siguiente función:

```
In [11]: def insert_noise(y, ratio=0.1):
    """
    Función para insertar ruido en una muestra de forma proporcional en cada
    clase.

    :param y: Etiquetas sobre las que insertar ruido
    :param ratio: Ratio de elementos de cada clase que modificar
    """
    # Obtener número de elementos sobre los que aplicar ruido
    # (redondear)
    noisy_pos = round(np.where(y == 1)[0].shape[0] * ratio)
    noisy_neg = round(np.where(y == -1)[0].shape[0] * ratio)

    # Obtener las posiciones de forma aleatoria
    pos_index = np.random.choice(np.where(y == 1)[0], noisy_pos, replace=False)
    neg_index = np.random.choice(np.where(y == -1)[0], noisy_neg, replace=False)

    # Cambiar los valores
    y[pos_index] = -1
    y[neg_index] = 1
```

Figura 6: Función para insertar ruido en una muestra de datos.

Al aplicar esta función sobre nuestras etiquetas, obtenemos los siguientes datos:

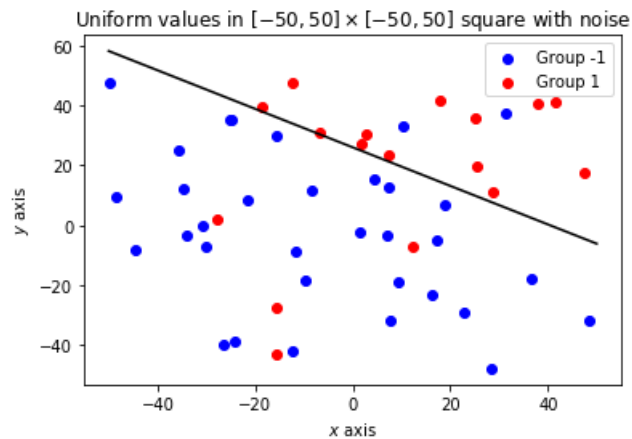


Figura 7: Gráfica de puntos generados uniformemente con ruido junto con la recta que los separa.

Como se puede ver, como había un mayor número de datos con etiqueta  $-1$ , se han modificado más elementos de esta clase, siendo este número de elementos modificados proporcional respecto al número de elementos totales de la clase.

Como era de esperarse, al haber modificado las etiquetas, los ratios de aciertos y error se han visto modificados, dejándolos de la siguiente forma:

**Ratio de aciertos: 0.88**  
**Ratio de error: 0.12**

Figura 8: Ratios de acierto y error obtenidos de la muestra con ruido.

### Apartado 3

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

1.  $f(x, y) = (x - 10)^2 + (y - 20)^2 - 400$
2.  $f(x, y) = 0.5(x + 10)^2 + (y - 20)^2 - 400$
3.  $f(x, y) = 0.5(x - 10)^2 + (y + 20)^2 - 400$
4.  $y - 20x^2 - 5x + 3$

Visualizar el etiquetado generado en 2b junto con cada una de las gráficas de cada una de las funciones. Comparar las formas de las regiones positivas y negativas de estas nuevas funciones con las obtenidas en el caso de la recta ¿Son estas funciones más complejas mejores clasificadores que la función lineal? ¿En qué ganan a la función lineal? Explicar el razonamiento.

Para ayudarnos con la visualización de los datos, hemos utilizado una función ya proporcionada, con lo cuál no vamos a discutir como funciona. En vez de eso, veamos cómo son las funciones anteriores una vez que se han implementado:

```
In [15]: # Función del primer apartado
def f1(X):
    return (X[:, 0] - 10) ** 2 + (X[:, 1] - 20) ** 2 - 400

# Función del segundo apartado
def f2(X):
    return 0.5 * (X[:, 0] + 10) ** 2 + (X[:, 1] - 20) ** 2 - 400

# Función del tercer apartado
def f3(X):
    return 0.5 * (X[:, 0] - 10) ** 2 - (X[:, 1] + 20) ** 2 - 400

# Función del cuarto apartado
def f4(X):
    return X[:, 1] - 20 * X[:, 0] ** 2 - 5 * X[:, 0] + 3
```

Figura 9: Funciones implementadas en Python.

Estas funciones están pensadas para que se modifiquen todos los datos a la vez y se devuelva la función aplicada a éstos.

Para calcular estos ratios, nos hemos ayudado de la siguiente función, la cuál ha sido parametrizada para que pueda recibir cualquiera de las funciones anteriores, predecir las etiquetas y devolver los correspondientes ratios:

```
In [18]: def error_rate_func(x, y, func):
    """
    Función para calcular los ratios de acierto y error al predecir un conjunto
    de puntos x mediante una función func, con respecto de los valores reales y

    :param x: Puntos que se usarán para predecir
    :param y: Etiquetas reales de los puntos
    :param func: Función con la que se predecirán los puntos

    :return Devuelve el ratio de puntos predichos correctamente y el ratio de
             puntos predichos erróneamente
    """

    # Predecir las etiquetas
    predicted_y = func(x)

    # Hacer que los valores predichos estén en el rango (-1, 1)
    predicted_y = np.clip(predicted_y, -1, 1)

    return np.mean(predicted_y == y), np.mean(predicted_y != y)
```

Figura 10: Función para calcular las tasas de aciertos y error para las nuevas funciones.



A continuación, se presentan las gráficas de las funciones junto con los valores de los ratios de aciertos y error:

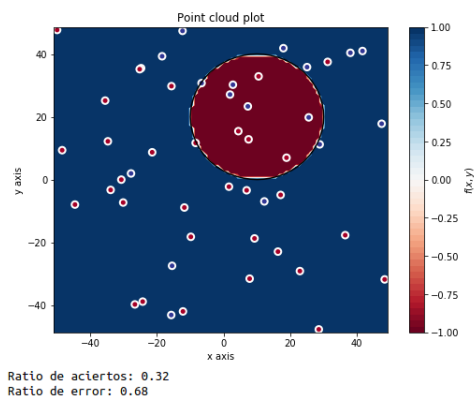


Figura 11: Gráfica y ratios de 1.

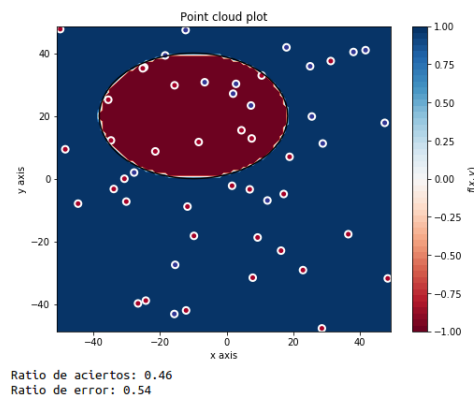


Figura 12: Gráfica y ratios de 2.

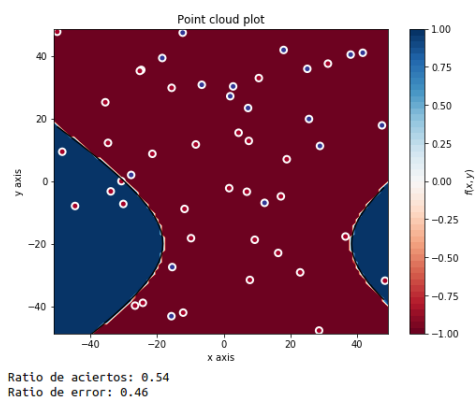


Figura 13: Gráfica y ratios de 3.

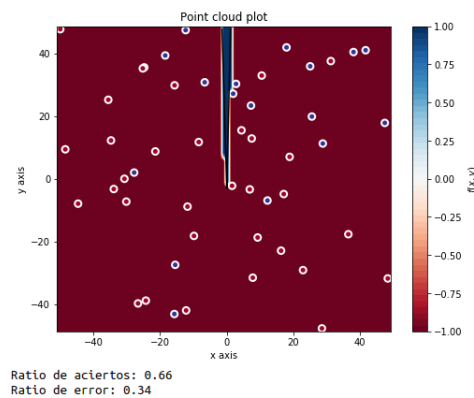


Figura 14: Gráfica y ratios de 4.

Como se puede ver, son funciones más complejas que las lineales. Algunas de ellas son figuras geométricas, como circunferencias o elipses (correspondientes a las funciones 1 y 2). Estas funciones serían capaces de explicar datos muy complejos, datos con los que las funciones lineales tendrían problemas, debido a que no son linealmente separables mediante una recta. Por ejemplo, si una clase estuviese dentro de otra, se podría usar perfectamente una función como 1 o 2.

Comparándolas con la recta, las regiones positivas y negativas cambian. En el caso de la recta, todo lo que había debajo de ella se supone que es negativo (aunque luego no sea así debido a la presencia de ruido). En estos nuevos casos, encontramos por ejemplo que las regiones negativas están dentro de las positivas (funciones 1

y 2), las regiones positivas están en los extremos laterales (función 3) o son tan pequeñas que casi no existen o parecen una parábola (caso de la función 4).

A pesar de esto, las funciones no parecen ser demasiado buenas, ya que en todos los casos los ratios de aciertos son menores que los de la función lineal (recordemos que su ratio de aciertos era 0.88). Esto se debe a que, aunque los datos presenten un cierto ruido, se pueden llegar a separar mejor utilizando una recta en vez de algo mucho más complejo como una elipse o dividiendo el espacio en regiones como en el caso de 3.

También hay que considerar que en este caso estamos cogiendo directamente funciones, sin entrenar ningún tipo de modelo para intentar ajustarlas a los datos. A lo mejor haciendo esto se podrían obtener algunos resultados un poco mejores en algunas de las funciones, pero como existe ruido en la muestra, es muy difícil dar con la función adecuada y que tenga el mínimo error. Además, en caso de encontrarla, por haber recurrido a una clase de funciones muy compleja por ejemplo, nos podríamos encontrar con el caso de *overfitting*, lo cuál es un problema gravísimo y nada deseable.

En resumen, las funciones no lineales tienen sus puntos fuertes debido a que pueden explicar ciertos tipos de muestras mejor de lo que lo haría una función lineal, pero no siempre son la solución a nuestros problemas. Pueden darse casos, como por ejemplo este, que la función lineal le gane a todas las no lineales, debido a que sea más fácil dividir los datos mediante una recta. Hay que ser muy consciente de qué tipo de función elegir en cada momento. Por ejemplo, un comportamiento no lineal en algunas características nos podría indicar que sería mejor intentar recurrir a alguna función no lineal que a una lineal para resolver el problema. Sin embargo, antes de recurrir a los modelos no lineales, podemos asegurarnos comprobando alguna información como por ejemplo el sesgo y la varianza que nuestro modelo lineal se queda corto para explicar los datos, teniendo en cuenta siempre que no se debe hacer *overfitting* con los datos disponibles.

## 2. MODELOS LINEALES

### Apartado 1

**Algoritmo Perceptrón:** Implementar la función *ajusta\_PLA(datos, label, max\_iter, vini)* que calcula el hiperplano solución a un problema de clasificación binaria usando el algoritmo PLA. La entrada *datos* es una matriz donde cada item con su etiqueta está representado por una fila de la matriz, *label* el vector de etiquetas (cada etiqueta es un valor +1 o -1), *max\_iter* es el número máximo de iteraciones permitidas y *vini* el valor inicial del vector. La función devuelve los coeficientes del hiperplano.

Primeramente, vamos a ver la implementación de la función:

```
In [14]: def adjust_PLA(data, label, max_iter, initial_values):
        """
        Implementación del PLA para ajustar una serie de pesos
        para un perceptrón

        :param data: Conjunto de datos con los que entrenar el perceptrón
        :param label: Conjunto de etiquetas, una por cada grupo de características
        :param max_iter: Número máximo de iteraciones (épocas) que hace el PLA
                        (limitar el número de iteraciones en caso de que no converja)
        :param initial_values: Valores iniciales de w

        :return Devuelve los pesos obtenidos (w) junto con el número de épocas
                que ha tardado en converger (epoch)
        """

        # Copiar valores iniciales de w
        w = np.copy(initial_values)

        # Inicializar la convergencia a falso
        convergence = False

        # Inicializar el número de épocas realizadas a 0
        epoch = 0

        # Mientras no se haya convergido, ajustar el Perceptron
        while not convergence:
            # Incrementar el número de épocas y decir que se ha convergido
            convergence = True
            epoch += 1

            # Recorrer cada elemento de los datos con su correspondiente etiqueta
            # Si se ve que el valor predicho no se corresponde con el real
            # se dice que no se ha convergido en esta época
            for x, y in zip(data, label):
                # Calcular valor predicho (función signo)
                predicted_y = signo(w.dot(x.reshape(-1, 1)))

                # Comprobar si el valor predicho es igual al real
                if predicted_y != y:
                    w += y * x
                    convergence = False

            # Si se ha alcanzado el máximo de épocas, terminar
            if epoch == max_iter:
                break

        return w, epoch
```

Figura 15: Implementación de la función *ajusta\_PLA()*.

- a) Ejecutar el algoritmo PLA con los datos simulados en los apartados 2a de la sección.1. Inicializar el algoritmo con: a) el vector cero y, b) con vectores de números aleatorios en  $[0, 1]$  (10 veces). Anotar el número medio de iteraciones necesarias en ambos para converger. Valorar el resultado relacionando el punto de inicio con el número de iteraciones.

Los resultados de ejecutar el algoritmo con los distintos datos son los siguientes:

Algoritmo PLA con  $w_0 = [0.0, 0.0, 0.0]$

Valor w: [-360.                      8.60062752    15.34534818]      Num. iteraciones: 44

Figura 16: Resultado del algoritmo PLA con un  $w_0 = [0, 0, 0]$ .

Como se puede comprobar, la haber partido de  $w_0 = [0, 0, 0]$ , el algoritmo PLA tarda 44 iteraciones (es decir, 44 épocas, o lo que es lo mismo, 44 vueltas completas a los datos) en converger.

#### Algoritmo PLA con $w_0$ aleatorio

```

w_0 = [0.45815087 0.88730805 0.53393433]
Valor w: [-308.54184913  7.39880231  13.09609343]  Num. iteraciones: 35

w_0 = [0.57727945 0.91749829 0.31414652]
Valor w: [-358.42272055  10.42306079  17.3459754 ]  Num. iteraciones: 46

w_0 = [0.72613844 0.80523584 0.48006293]
Valor w: [-308.27386156  7.3167301  13.04222203]  Num. iteraciones: 35

w_0 = [0.48757437 0.84141006 0.55552753]
Valor w: [-308.51242563  7.35290432  13.11768663]  Num. iteraciones: 35

w_0 = [0.26988129 0.48674271 0.27584589]
Valor w: [-326.73011871  10.7676405  16.48956701]  Num. iteraciones: 45

w_0 = [0.22539047 0.64675391 0.92113358]
Valor w: [-264.77460953  7.65188314  11.6387396 ]  Num. iteraciones: 30

w_0 = [0.20047475 0.8798531  0.4658077 ]
Valor w: [-308.79952525  7.39134736  13.0279668 ]  Num. iteraciones: 35

w_0 = [0.72498012 0.30287132 0.48900729]
Valor w: [-465.27501988  16.50949988  20.63142906]  Num. iteraciones: 65

w_0 = [0.20367433 0.08861626 0.16568964]
Valor w: [-445.79632567  15.64429041  19.04615998]  Num. iteraciones: 64

w_0 = [0.50299653 0.78758101 0.90360187]
Valor w: [-473.49700347  11.27102087  17.83989574]  Num. iteraciones: 68

```

Valor medio de iteraciones necesario para converger: 45.8

Figura 17: Resultado del algoritmo PLA con un  $w_0$  aleatorio.

En este caso, aunque no se especificaba, también se ha mostrado individualmente cuánto tarda el algoritmo PLA en converger para cada punto aleatorio. En algunos casos tarda más que en otros, pero de media tarda 45.8 épocas en converger, valor que se queda muy cerca de las 44 épocas anteriores. Se puede ver además que, dependiendo del punto inicial, se encuentra un plano u otro que divida a los puntos, aunque en algunos casos estos planos son bastante parecidos. Esto no es de extrañar, ya que existen muchísimos planos que pueden separar a los datos si estos son linealmente separables.

Con esto podemos decir que la posición inicial de  $w$  influye en el número de épocas hasta que el algoritmo converja, pero que si los datos son linealmente

separables, como es este caso, se llega a converger siempre, aunque sea de forma un poco lenta ya que se requieren muchas iteraciones hasta poder converger.

- b) Hacer lo mismo que antes usando ahora los datos del apartado 2b de la sección.1. ¿Observa algún comportamiento diferente? En caso afirmativo diga cual y las razones para que ello ocurra.

Vamos a ver cuáles son los resultados obtenidos en cada caso.

Algoritmo PLA con  $w_0 = [0.0, 0.0, 0.0]$  y datos con ruido

Valor  $w$ : [-389.                    14.87257812      4.5849045 ]      Num. iteraciones: 10000

Figura 18: Resultado del algoritmo PLA con un  $w_0 = [0, 0, 0]$  y con ruido en la muestra.

Algoritmo PLA con  $w_0$  aleatorio y datos con ruido

```
w_0 = [0.21952409 0.73813097 0.25780289]
Valor w: [-388.78047591 15.62892281 4.39934244]      Num. iteraciones: 10000

w_0 = [0.56420874 0.13242963 0.37208768]
Valor w: [-3.91435791e+02 -2.78712499e-01 1.53508762e+01]      Num. iteraciones: 10000

w_0 = [0.08285546 0.50572622 0.69440685]
Valor w: [-399.91714454 -2.17596444 44.14282903]      Num. iteraciones: 10000

w_0 = [0.45697791 0.36127471 0.81600751]
Valor w: [-382.54302209 29.28179251 10.19639859]      Num. iteraciones: 10000

w_0 = [0.04581152 0.38782779 0.45852318]
Valor w: [-390.95418848 -15.37579277 39.84245551]      Num. iteraciones: 10000

w_0 = [0.73573212 0.90963695 0.27740547]
Valor w: [-405.26426788 -19.18979239 38.63832631]      Num. iteraciones: 10000

w_0 = [0.81776839 0.65631586 0.01128031]
Valor w: [-395.18223161 2.24818188 15.84450593]      Num. iteraciones: 10000

w_0 = [0.73535369 0.60349464 0.68001914]
Valor w: [-390.26464631 13.65392192 8.52446406]      Num. iteraciones: 10000

w_0 = [0.58395979 0.57352097 0.36101747]
Valor w: [-395.41604021 4.26188126 57.85715118]      Num. iteraciones: 10000

w_0 = [0.9586999 0.98694282 0.38713543]
Valor w: [-397.0413001 7.15980974 10.01618516]      Num. iteraciones: 10000
```

Valor medio de iteraciones necesario para converger: 10000.0

Figura 19: Resultado del algoritmo PLA con un  $w_0$  aleatorio y con ruido en la muestra.

Como se puede comprobar a simple vista, todos los algoritmos terminan en el mismo número de iteraciones. No es casualidad, ya que al algoritmo se le ha especificado que *max\_iter* sea 10000 iteraciones o épocas. Entonces, ¿por qué sucede esto?

La respuesta es muy simple, y es que los datos no son linealmente separables. En los datos con ruido hay puntos de una clase que están mezclados con los de la otra, y no existe ninguna forma de separar esos datos con una recta/plano (habría que intentar utilizar algo más complejo como una circunferencia o una elipse, aunque tampoco se garantiza que se consiga algo con estos, aparte de que el PLA solo trabaja con planos/rectas). Por tanto, el algoritmo PLA nunca va a poder converger en estos casos, que es precisamente lo que sucede en este ejemplo.