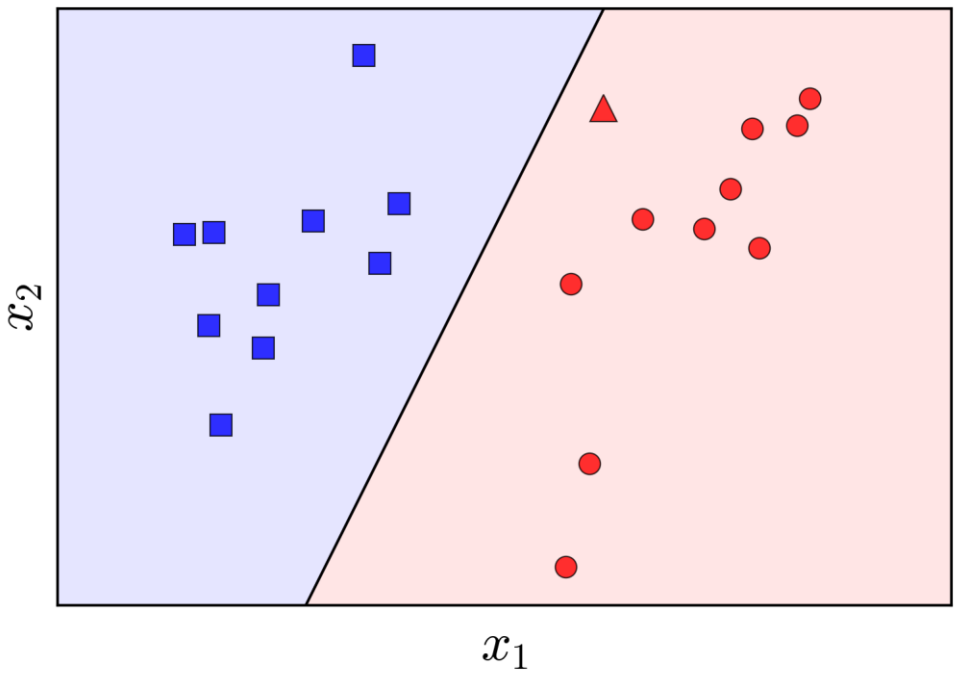
Trabajo 2: Programación



Sergio Aguilera Ramírez

Índice:

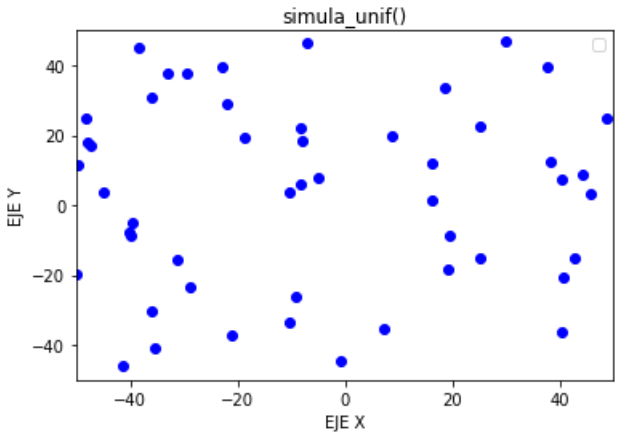
1- EJERCICIO 1

1.1- Gráficas (simula_unif() y simula_gauss

Para generar la nube de puntos de los siguientes apartados, he empleado dos funciones proporcionadas por el profesor, las cuales son simula_unif() (esta función genera una distribución uniforme en un determinado rango) y simula_gauss (esta función genera una distribución gaussiana).

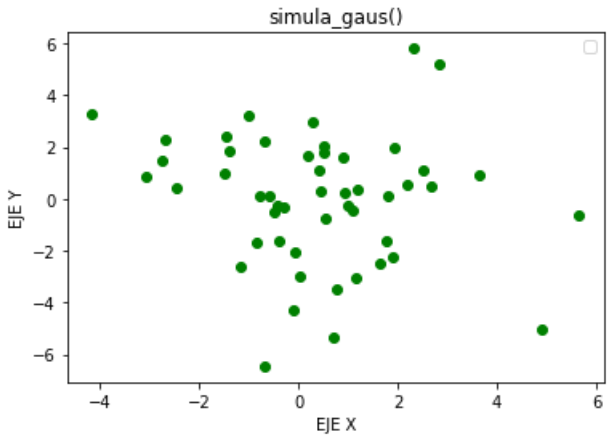
1.1.1- simula_unif()

N = 50, dim = 2, rango = [-50,50]



1.1.2- simula_gaus()

N = 50, dim = 2, rango = [-50,50]



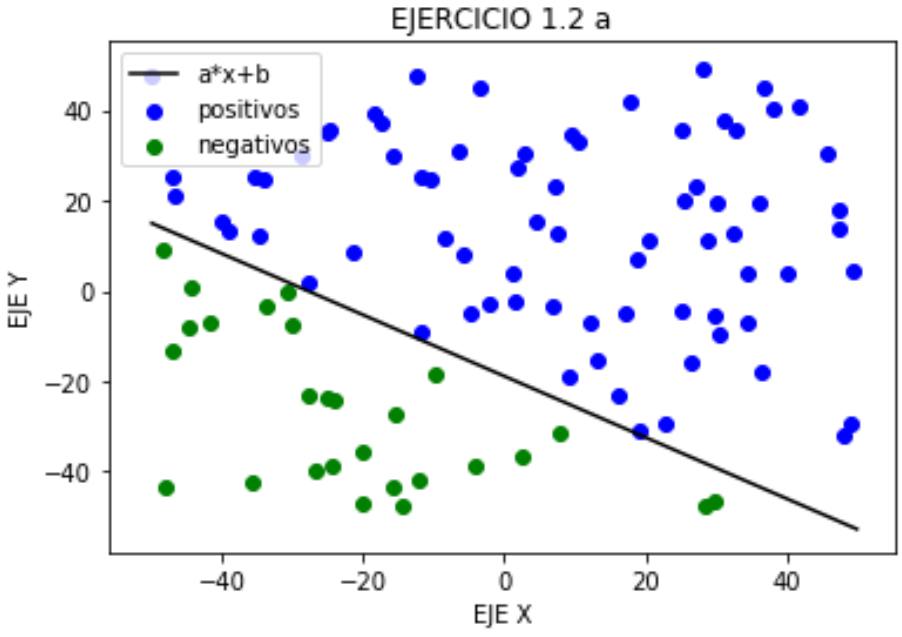
1.2- Etiquetado de la nube de puntos

En este apartado he generado una muestra de 100 puntos con ayuda de la función simula_unif(). Mediante la función $f(x, y) = y - ax - b$ calculamos la etiqueta de cada punto. Necesitamos calcular una recta aleatoria comprendida en nuestro rango. Dicha recta se calcula con la función simula_recta() que nos devuelve la pendiente (a) y el termino independiente (b). El etiquetado de los puntos dependerá de la distancia de cada punto a la recta.

La recta se dibuja de la forma plt.plot(x,y, 'k', label='..'), donde 'x' es un conjuntos de valores comprendidos entre -50 y 50 (np.linspace(-50,50,100)) y 'y' es el conjunto de valores calculados de cada componente 'x' mediante $y = a * x + b$.

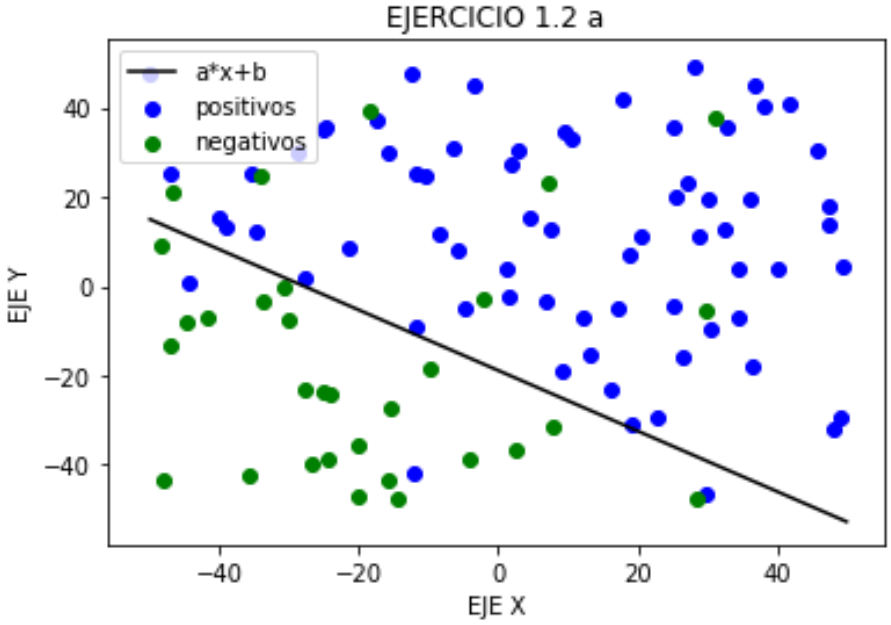
1.2-1. Gráfica sin ruido

Generemos la nube de puntos y etiquetamos los puntos como hemos detallado anteriormente, dibujamos la nube de puntos con su respectiva recta de hiperplano. Como podemos observar la recta divide perfectamente los puntos.



1.1-1. Gráfica con ruido

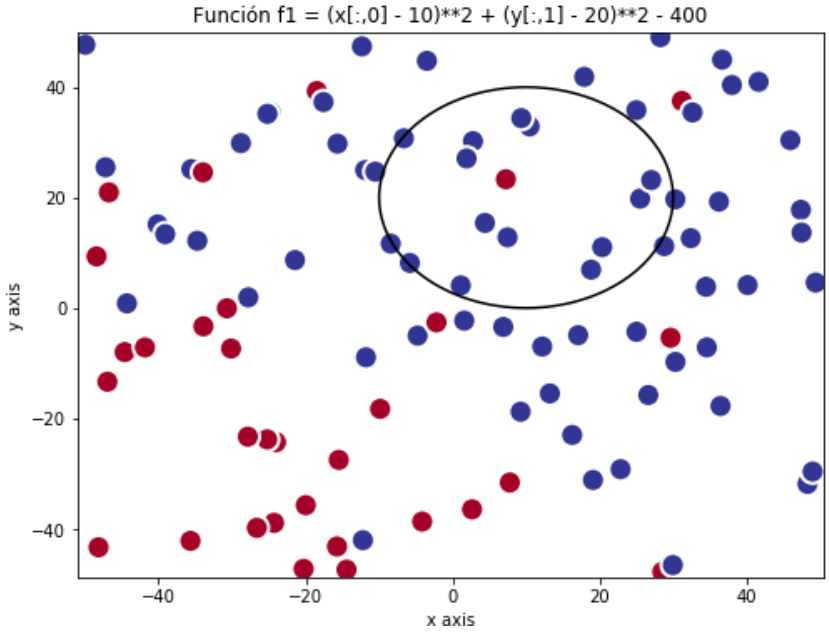
Este apartado consiste en introducir un 10% de ruido a cada grupo de puntos (negativos y positivos) obtenidos en el apartado anterior. Debido al ruido introducido podemos observar como los datos están mal clasificados respecto a la recta.



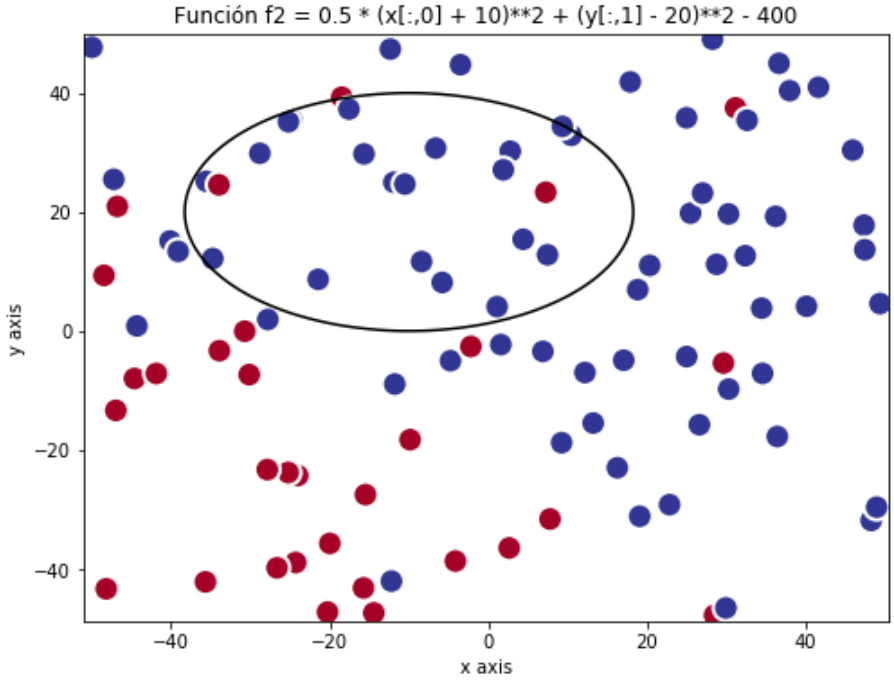
1.3- Clasificación usando diferentes funciones complejas

En este apartado vamos a definir cuatro funciones más complejas, donde la frontera de clasificación no es una recta si no que pueden ser círculos u elipses. La recta verde (a*x+b) es la recta clasificatoria del conjunto.

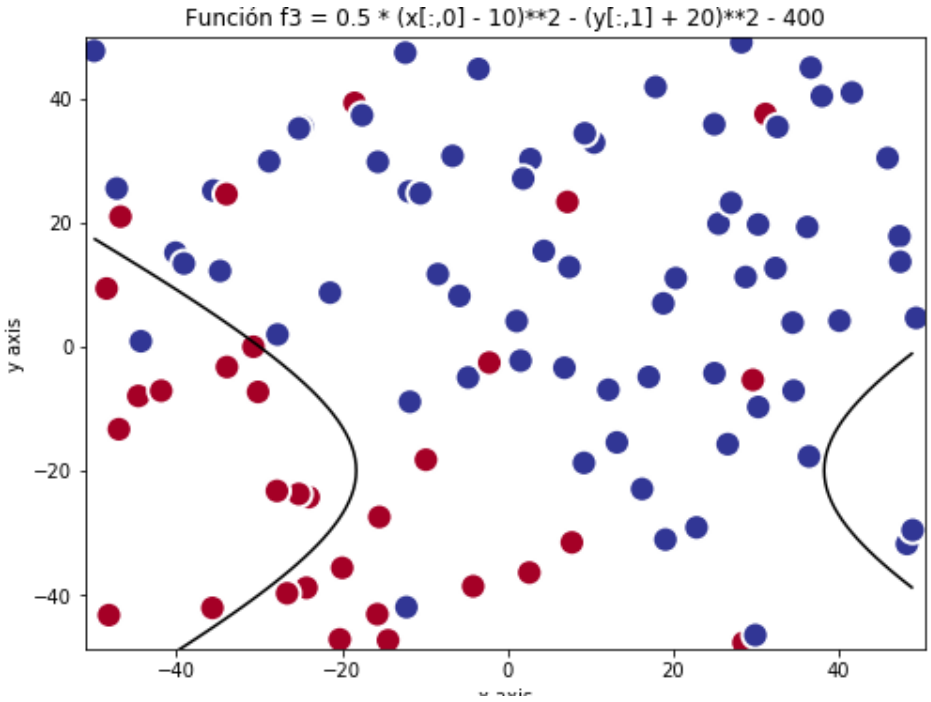
1.3.1- $f(x, y) = (x - 10)^2 + (y - 20)^2 - 400$



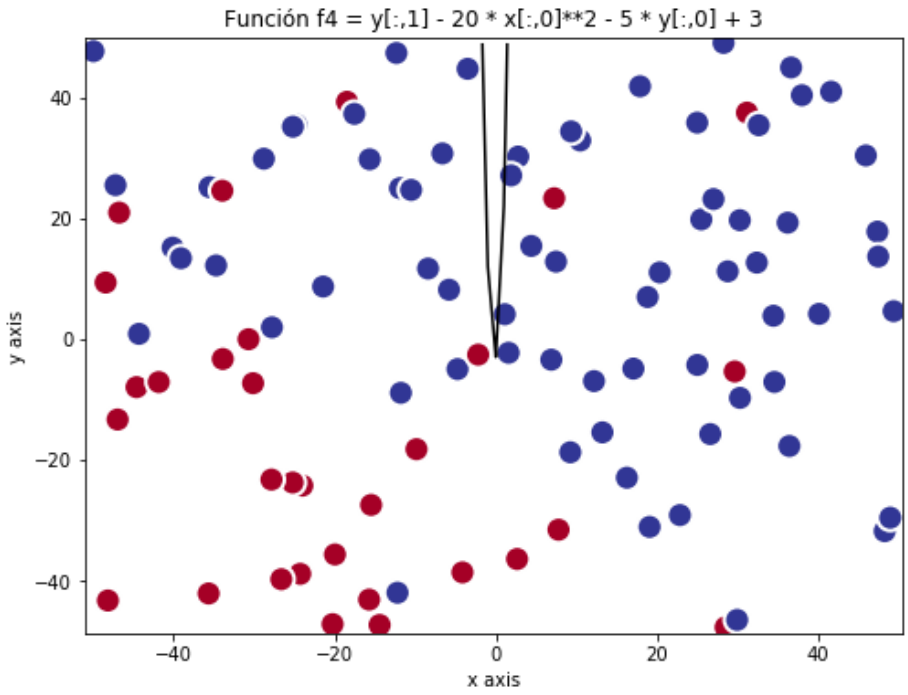
1.3.2- $f(x,y) = 0.5 * (x+10)^2 + (y-20)^2 - 400$



1.3.3- $f(x,y) = 0.5 * (x-10)^2 - (y+20)^2 - 400$



1.3.4- $f(x, y) = y - 20x^2 - 5x + 3$



Conclusión

Como podemos observar estas funciones tendrían un error mayor en la clasificación de los puntos que la propia recta clasificatoria, esto es debido al ruido introducido en el apartado 2b. Estas funciones son realmente buenas en conjuntos de datos que no son separables linealmente, si etiquetásemos los datos respecto a la distancia de cada dato a la función f, esto nos proporcionaría una división de clases del conjunto de datos que no podríamos clasificar de manera correcta mediante una recta. En este caso, estas funciones más complejas no obtienen un mejor resultado que la propia recta clasificatoria.

2- EJERCICIO 2

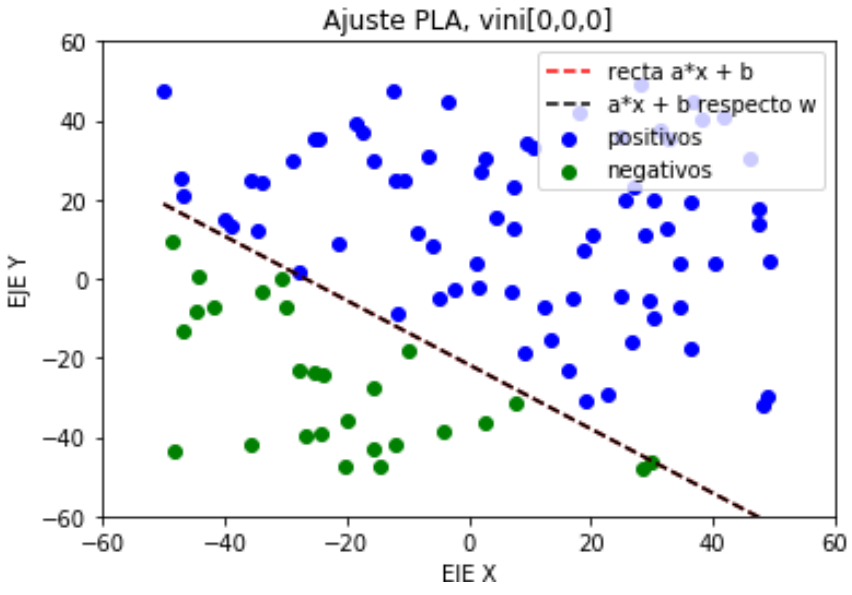
2.1- Algoritmo Perceptron

El algoritmo Perceptron es un método de regresión lineal. Este método consiste en actualizar el vector de pesos de la forma $w_{new} = w_{current} + y * x$ si la etiqueta obtenida mediante $sign(w^T * x_i)$ es distinta que la etiqueta real del valor, en caso contrario si la etiqueta esta bien clasificada el vector de pesos no cambia. Esto permite redireccionar el vector de pesos de forma que pueda obtener una mejor clasificación de los datos. Se toma como condición de parada que el vector de pesos w_{new} al final de una iteración completa de los datos sea igual a $w_{anterior}$, ya que esto supondría una clasificación correcta de todos los datos, también tomamos como condición de parada un número máximo de iteraciones.

2.1.1- Ejecución del algoritmo PLA con datos sin ruido

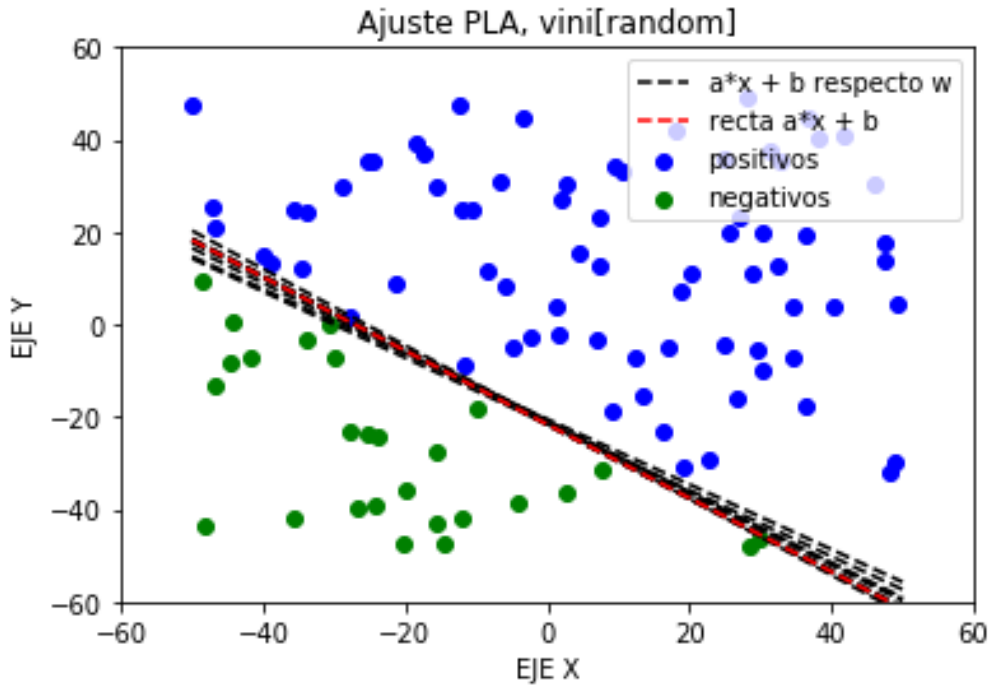
En este apartado tomamos como conjunto de datos, los datos simulados en el apartado 2a del ejercicio 1 (sin ruido). Sobre esta ejecución vamos a realizar dos experimentos a) w inicial a 0.0 y b) w inicial aleatorio entre [0,1].

A) W = [0.0,0.0,0.0]



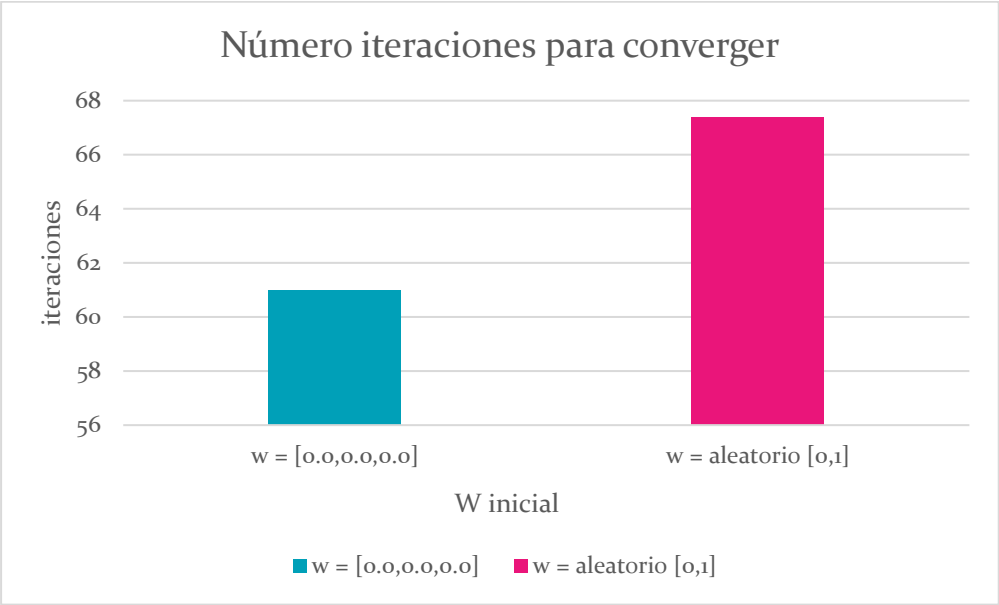

```
Numero de iterationes (vector inicial a 0):  61
```

B) W aleatorio entre [0,1]




```
Valor medio de iteraciones necesario para converger (vector inicial
aleatorio): 67.4
```

Comparativa

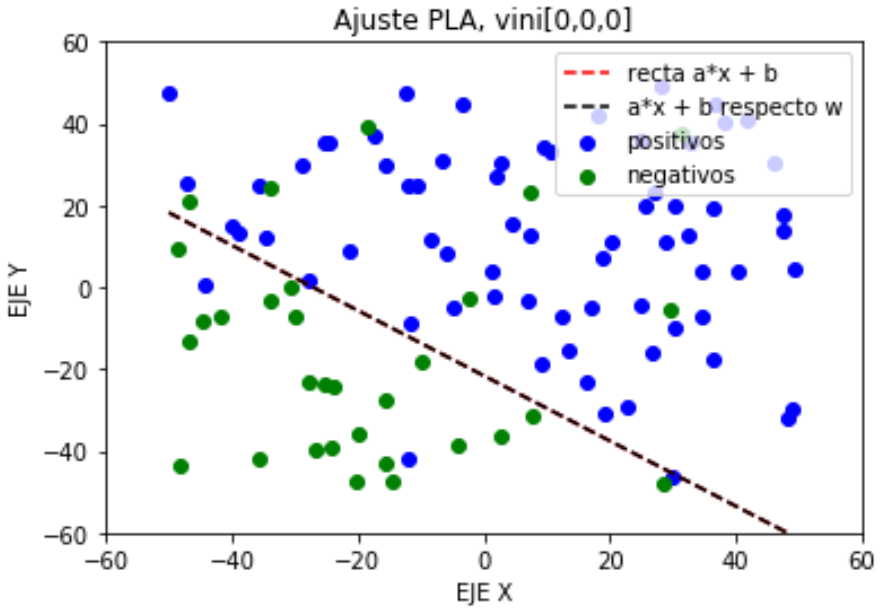


Como podemos observar el punto de inicio de w influye en el número de iteraciones para converger. La diferencia entre las iteraciones es pequeña.

2.1.2- Ejecución del algoritmo PLA con datos con ruido

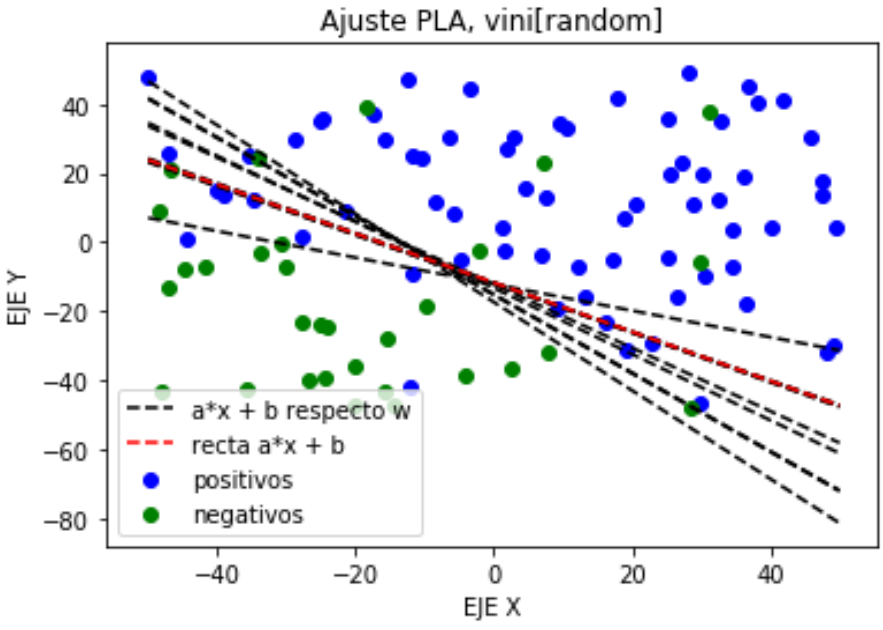
En este apartado tomamos como conjunto de datos, los datos simulados en el apartado 2b del ejercicio 1 (con ruido). Sobre esta ejecución vamos a realizar dos experimentos a) w inicial a 0.0 y b) w inicial aleatorio entre [0,1].

A) W = [0.0,0.0,0.0]



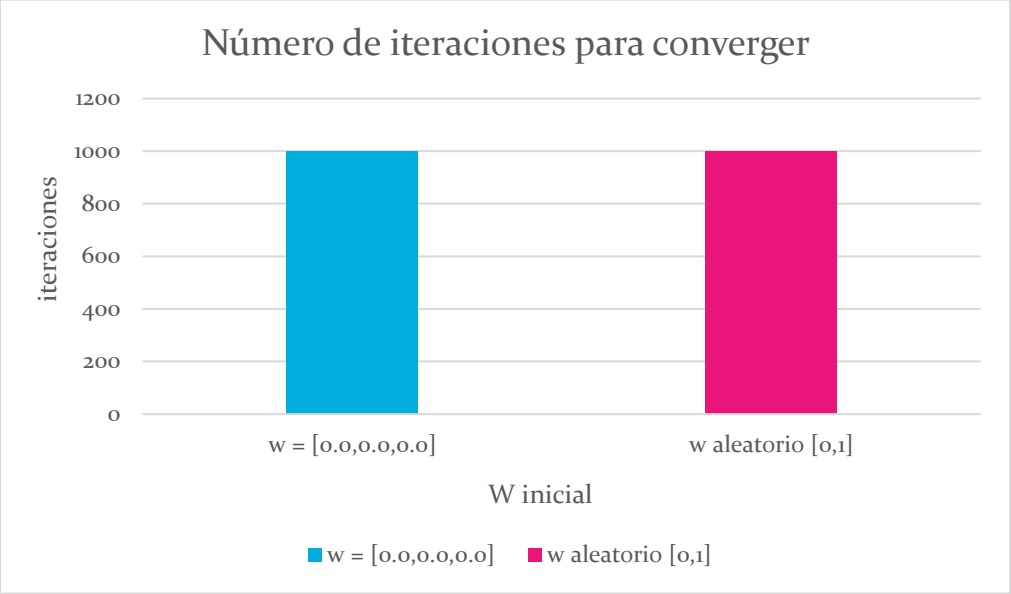

```
Numero de iterationes (vector inicial a 0):  1000
```

A) W aleatorio entre [0,1]



```
Valor medio de iteraciones necesario para converger (vector inicial aleatorio): 1000.0
```

Comparativa



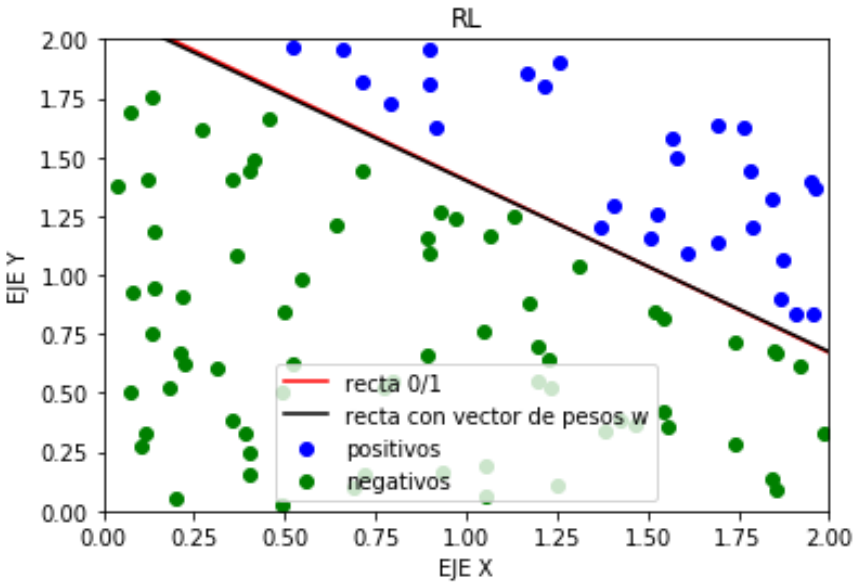
Podemos observar que el número de iteraciones necesario para converger en ambos experimentos es el número máximo de iteraciones, esto es debido al ruido introducido en los datos no permite clasificar los datos de manera correcta.

2.2- Regresión Logística

Este algoritmo consiste en la búsqueda del vector de pesos que obtenga una buena clasificación de los datos. Se pide implementar Regresión Logística con Gradiente Descendiente Estocástico, el funcionamiento de este algoritmo consiste en: inicializamos el vector de pesos a 0.0, escogemos aleatoriamente conjuntos de minitbaches de tamaño 32 sobre los datos (en cada época), sobre estos minibatches calculamos el gradiente de la forma $\sum_{i=0}^{N} -y_i x_i \sigma(-y_i w^T x_i)$, con dicho gradiente actualizamos w ($w = w - learning_{rate} * gradiente$), siendo las condiciones de para las siguientes: que $\|w^{epoca_{actual}-1} - w^{epoca_{actual}}\| <$ épsilon (0.01) o superara un máximo de épocas.

A) Ejecución del algoritmo RL

Generamos un conjunto de 100 puntos de dimensión 2 comprendido en el rango [0,2], simulamos la recta (simula_recta([0,2]). Clasificamos los datos según su distancia a la recta. Llamamos a la función sgdRL pasandole como argumento conjunto de datos (x2_aux), las etiquetas(etiq), learning_rate (0.01) , épsilon (0.01) y un número máximo de épocas (3000), esto nos devuelve un vector de pesos, calculamos la recta respecto a este vector.



Como podemos observar el ajuste de la recta obtenido a partir del vector de pesos w calculado con el algoritmo RL es muy parecida a la recta clasificatoria.

B) Errores

La función de error viene representada con la fórmula $e(w) = \ln(1 + e^{-y_n w^T x_n})$.

El error de entrada lo he calculado mediante los datos generados en el apartado anterior y con el w obtenido en el mismo, y el error de salida lo he calculado realizando 1000 experimentos (simulados con simula_unif()) y calculando el error medio de todos ellos (utilizando el vector de pesos w obtenido en el apartado anterior).

Ein: 0.10840682534356544

Eout medio: 0.11622650032890273

3- BONUS

3.1- Regresión Lineal

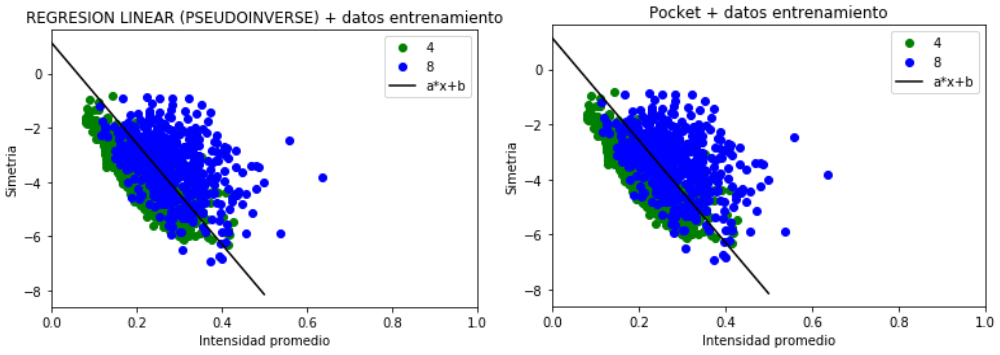
En este ejercicio debemos clasificar los datos mediante una regresión lineal. Para ello he utilizado la pseudoinversa cuya fórmula es $w = ((X^T X)^{-1} X^T) * y$. Obteniendo los siguientes errores de entrada y salida.

Ein: 0.22780569514237856
Eout: 0.25136612021857924

3.2- PLA-Pocket

Este algoritmo consiste en hacer una mejora al algoritmo de perceptrón. PLA-Pocket lo que hace es almacenar por cada iteración la mejor solución encontrada de modo que finalmente si supera el máximo de iteraciones pueda devolver el mejor resultado de todos los procesados. Inicializamos el vector de pesos al w obtenido según la regresión lineal del apartado anterior, establecemos el error_min a 999.0, actualizamos w cuando la etiqueta de algún valor del conjunto de datos sea distinta a la etiqueta real de dicho dato ($sign(w^T x_i) \neq y_i \; then \; w = w + y_i x_i$), si $Err(x, y, w) < error\_min$ entonces error_min = Err(x,y,w) y w_final = w, las condiciones de parada son $\|w^{ant} - w^{actual}\|$ o superar un máximo de iteraciones.

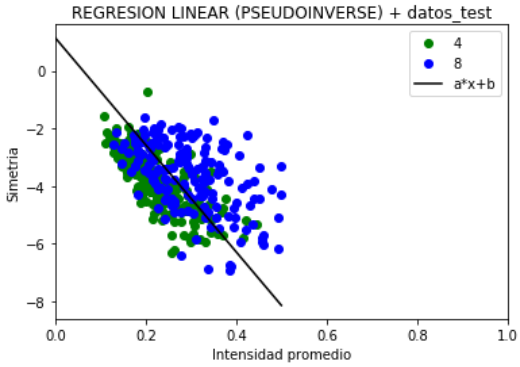
a) Datos entrenamiento

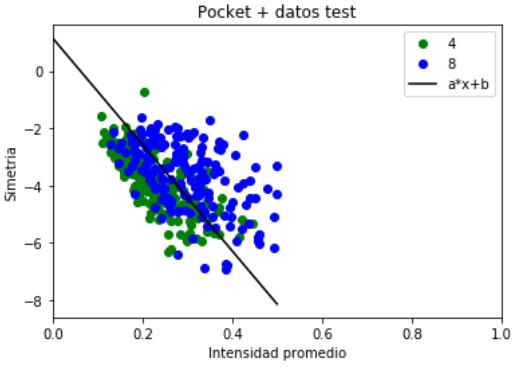



```
Ein (Pseudoinversa):  0.22780569514237856
```

```
Ein (PLA-Pocket):  0.22529313232830822
```

b) Datos test



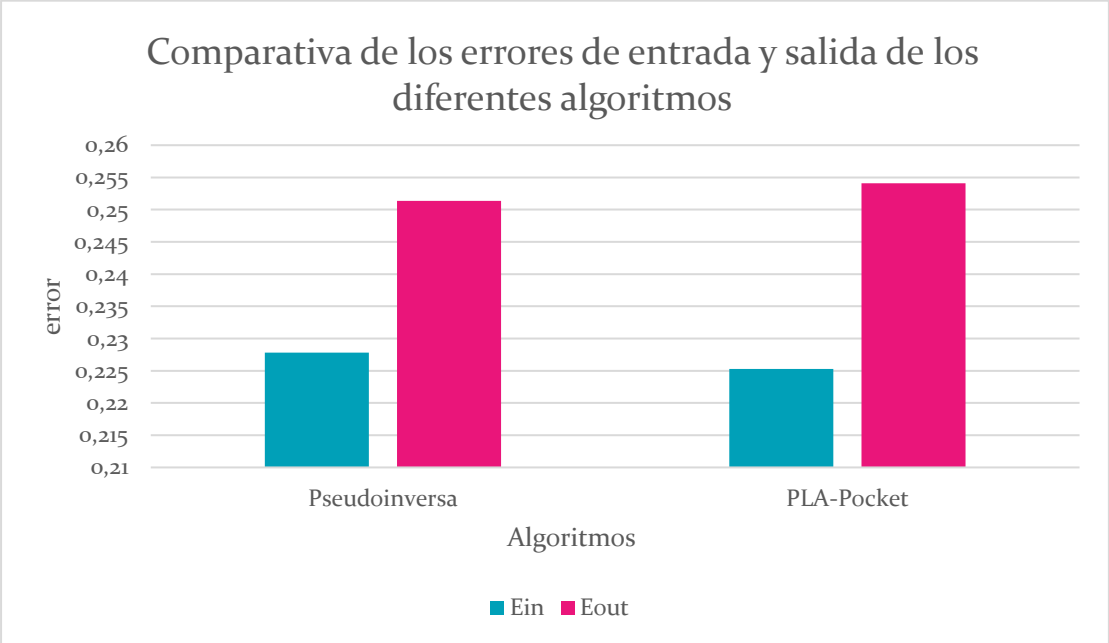



```
Eout (Pseudoinversa):  0.25136612021857924
```

```
Eout (PLA-Pocket):  0.2540983606557377
```

Comparativa

	Ein	Eout
Pseudoinversa	0.22780569514237856	0.25136612021857924
PLA-Pocket	0.22529313232830822	0.2540983606557377



Como se representa en la gráfica, podemos ver como el algoritmo PLA-Pocket obtiene un error menor en el conjunto de entrenamiento con respecto a la pseudoinversa, pero en el error obtenido en el conjunto test es mayor respecto a la pseudoinversa.

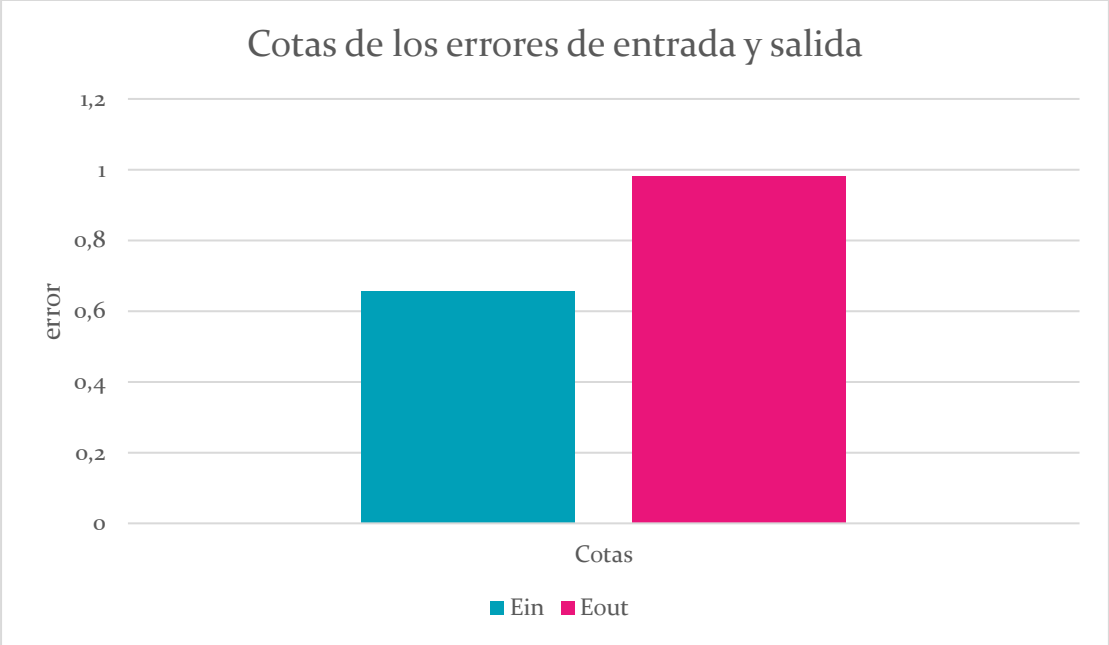
c) Cotas

Para el cálculo de las cotas he utilizado la fórmula:

$$cota = err + \sqrt[2]{\frac{8}{tam} * \ln\left(\frac{4*((2*tam)^{dimension}+1)}{tolerancia}\right)}$$

obteniendo las siguientes cotas:

Cota de Ein: 0.6562296377990118
Cota de Eout: 0.9809354817745115



Como podemos observar existe una gran diferencia entre la cota de Ein y la cota de Eout esto puede ser debido a que el tamaño de entrenamiento es mayor que el de test, asi como la existencia de una menor diversidad entre los puntos en el conjunto de entrenamiento respecto al conjunto de test. Los resultados de las cotas son bastante altas.