APRENDIZAJE AUTOMÁTICO

UNIVERSIDAD
DE GRANADA

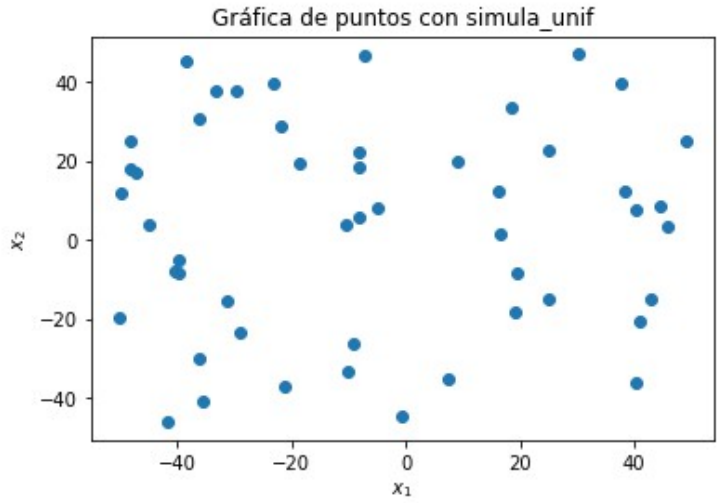
Práctica 2: Programación

Alba Casillas Rodríguez
76738108B
albacaro@correo.ugr.es

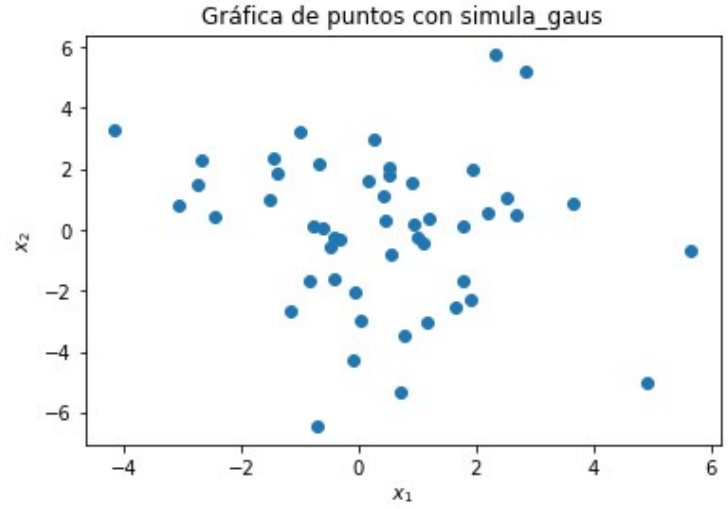
Ejercicio 1: Ejercicio sobre la complejidad de H y el ruido

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**).



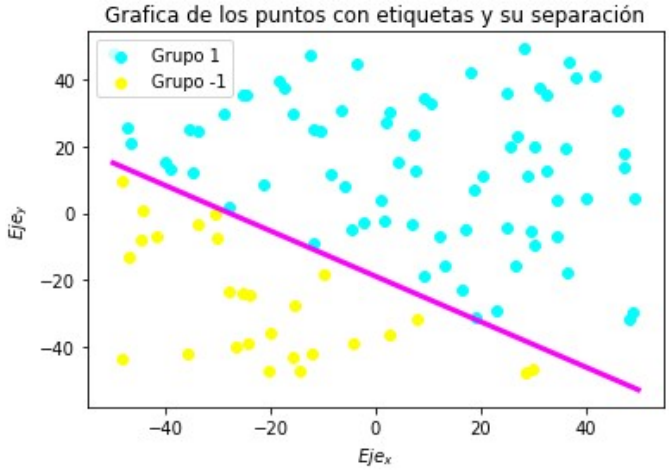
b) **Considere N** = 50, **dim** = 2 **y sigma** = [5, 7] **con simula_gaus**(**N**, **dim**, **sigma**).



2. **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)**

Tras generar la muestra de puntos 2D con ***simula_unif(N,dim,rango)*** para N = 100, dim= 2 y rango = [-50,50], y los parámetros (a,b) de la recta mediante ***simula_recta(intervalo)*** ; clasificamos los puntos obteniendo la etiqueta para cada punto. Para ello, se usarán las funciones ***signo(x)*** y ***f(x,y,a,b)*** (donde x es la coordenada X del punto, y la coordenada Y del punto, y a y b los valores obtenidos simula_recta).

Mostramos los puntos generados y su recta:



2. **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)**

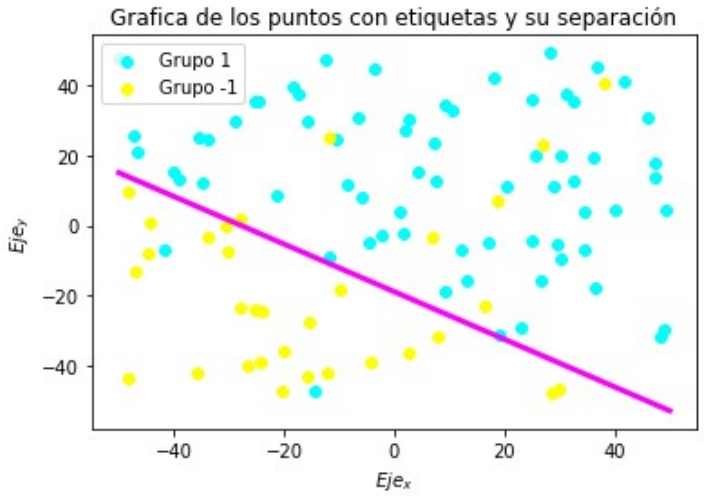
Para resolver este apartado, se ha implementado una función que insertará ruido en nuestra muestra con un porcentaje del 10% para cada tipo de muestra que tenemos (positivas y negativas)

```
def aniade_ruido(Y, porcentaje = 0.1):
        N_pos ← número de elementos positivos (10%) a los que le aplicamos ruido
        N_neg ← número de elementos negativos(10%) a los que le aplicamos ruido

        indices_pos ← array( N_pos posiciones aleatorias positivas de los elementos a cambiar)
        indices_neg ← array(N_neg posiciones aleatorias negativas de los elementos a cambiar)

        return indices_pos, indices_neg
```

Cambiamos el valor de las etiquetas de los elementos que ocupan las posiciones de los indices generados y mostramos de nuevo los puntos y su recta:



2. **c**) **Supongamos ahora que las siguientes funciones definen la frontera de clasificación de los puntos de la muestra en lugar de una recta**:

$$-\mathbf{f}(\mathbf{x}, \mathbf{y}) = (\mathbf{x} - 10)^2 + (\mathbf{y} - 20)^2 - 400$$

```
def f_1(X):
        return (X[:, 0] - 10) ** 2 + (X[:, 1] - 20) ** 2 – 400
```

$$-\mathbf{f}(\mathbf{x}, \mathbf{y}) = 0{,}5(\mathbf{x} + 10)^2 + (\mathbf{y} - 20)^2 - 400$$

```
def f_2(X):
        return 0.5 * (X[:, 0] + 10) ** 2 + (X[:, 1] - 20) ** 2 – 400
```

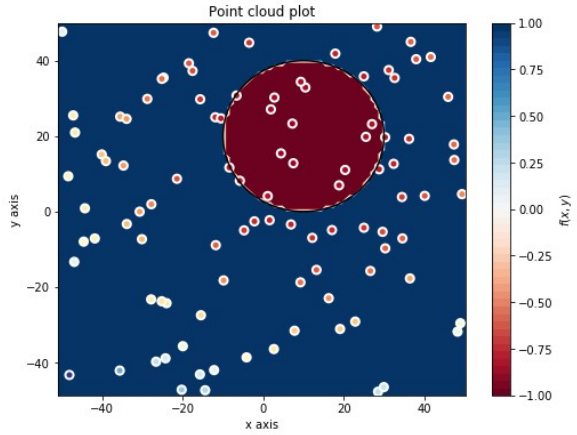
$-\mathbf{f}(\mathbf{x}, \mathbf{y}) = 0{,}5(\mathbf{x} - 10)^2 - (\mathbf{y} + 20)^2 - 400$

```
def f_3(X):
        return 0.5 * (X[:, 0] - 10) ** 2 - (X[:, 1] + 20) ** 2 – 400
```

$-\mathbf{f}(\mathbf{x}, \mathbf{y}) = \mathbf{y} - 20\mathbf{x}^2 - 5\mathbf{x} + 3$

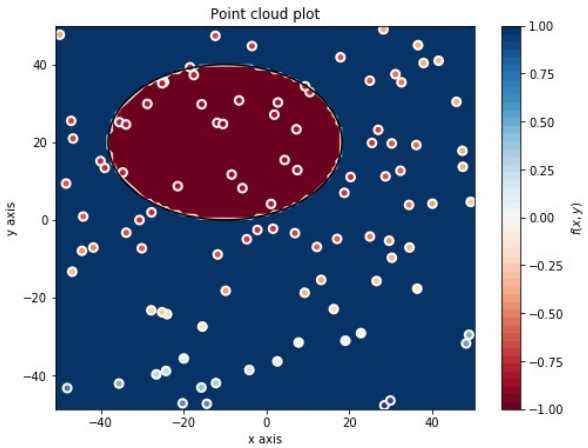
```
def f_4(X):
        return X[:, 1] - (20 * X[:, 0] ** 2)  - 5 * X[:, 0]  + 3
```

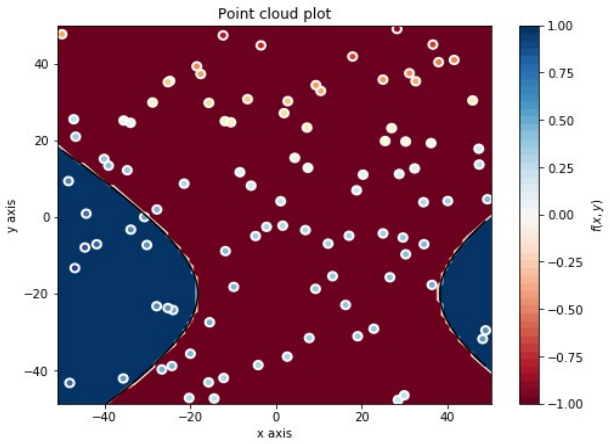
Visualizar el etiquetado generado en 2b junto con cada una de las gráficas de cada una de las funciones. Comparar las regiones positivas y negativas de estas nuevas funciones con las obtenidas en el caso de la recta. ¿Son estas funciones más complejas mejores clasificadores que la función lineal? Observe las gráficas y diga que consecuencias extrae sobre la influencia del proceso de modificación de etiquetas en el proceso de aprendizaje Explicar el razonamiento.



Gráfica función f_1(X)

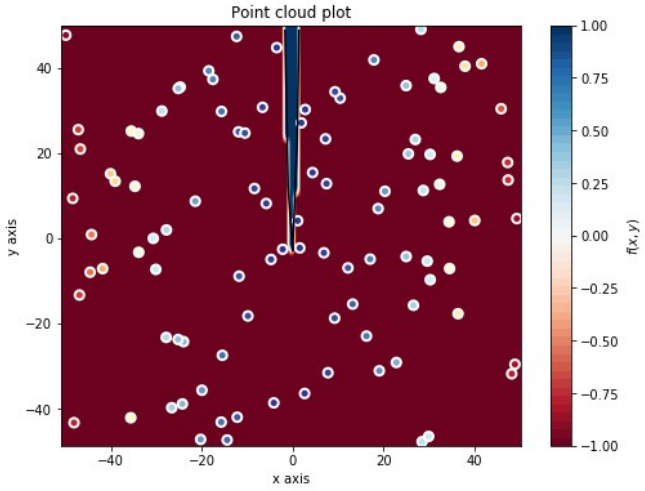
Gráfica función f_2(X)





Gráfica función f_3(X)

Gráfica función f_4(X)



Al comparar las regiones positivas y negativas con las de la recta, no es difícil encontrar diferencias. En el caso de la recta, los datos están claramente separados, situando debajo de la recta los casos negativos y encima los positivos. En las siguientes funciones, podemos encontrar diversos casos: en las dos primeras gráficas, los datos positivos rodean a los negativos, formándose unas regiones con forma de elipse; y en las dos últimas, las regiones positivas quedan aisladas formando parábolas (verticales y horizontales) en los laterales o en una zona pequeña de la gráfica, respectivamente; ***siendo así funciones más complejas que las lineales***.

Sin embargo, a pesar de la complejidad, no se observa una mejora a la hora de clasificar los datos en comparación a la recta, la cual realiza una separación exacta de los datos.

Con ello podemos concluir que este tipo de funciones no siempre tienen por qué ser mejores clasificadores que la función lineal, ya que esto ***dependerá del tipo de problema*** con el que se trate. Estas funciones se adaptarán muy bien a problemas más complejos, donde los datos no sean linealmente separables por una recta; y sin embargo, sean peores en casos como este.

Ejercicio 2: Modelos Lineales

a) Algoritmo Perceptron: Implementar la función

ajusta_PLA(datos,label,max_iter,vini)

1) **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.**

Pseudocodigo del algoritmo Perceptron:

```
def ajusta_PLA(datos, label, max_iter, vini):
        err ← True
        iteracion ←  0
        w ← vini

        while err:
                err ← False
                iteracion ++

        for i ∈{1...len(datos)}:
                if(signo(w^T*datos[i])) != label[i]:
                        w ←  w + label[i]*datos[i]
                        err ← True
        if iteracion == max_iter:
                break

return w, iteracion
```

donde ***datos*** es la muestra de 100 puntos 2D generados con simula_unif, ***label*** las etiquetas para cada punto a la recta simulado, ***max_iter*** el número máximo de iteraciones permitidas y ***vini*** el valor inicial del vector.

Inicialmente, probé con max_iter = 500:

- con el vector a cero : v_ini = np.array([0.0,0.0,0.0])

Peso inicial	Número iteraciones
[0,0,0]	444

- con vectores de 10 números aleatorios en [0,1]

Peso inicial	Número iteraciones
[0.39000771 0.48599067 0.60431048]	47
[0.54954792 0.92618143 0.91873344]	42
[0.39487561 0.96326253 0.17395567]	500
[0.12632952 0.13507916 0.50566217]	230
[0.02152481 0.94797021 0.82711547]	500
[0.01501898 0.17619626 0.33206357]	500
[0.13099684 0.80949069 0.34473665]	17
[0.94010748 0.58201418 0.87883198]	500
[0.84473445 0.90539232 0.45988027]	500
[0.54634682 0.79860359 0.28571885]	17

Valor medio de iteraciones necesario para converger: 285.3

Sin embargo, al fijarnos en los valores subrayados, podemos darnos cuenta de que varios puntos iniciales no converge, llegando al máximo número de iteraciones.

Por ello, cambiamos max_iter = 2000

- con el vector a cero : v_ini = np.array([0.0,0.0,0.0])

Peso inicial	Número iteraciones
[0,0,0]	444

- con vectores de números aleatorios en [0,1](10 veces)

Peso inicial	Número iteraciones
[0.39000771 0.48599067 0.60431048]	47
[0.54954792 0.92618143 0.91873344]	42
[0.39487561 0.96326253 0.17395567]	1451
[0.12632952 0.13507916 0.50566217]	230
[0.02152481 0.94797021 0.82711547]	1588
[0.01501898 0.17619626 0.33206357]	1573
[0.13099684 0.80949069 0.34473665]	17
[0.94010748 0.58201418 0.87883198]	1519
[0.84473445 0.90539232 0.45988027]	1607
[0.54634682 0.79860359 0.28571885]	17

Valor medio de iteraciones necesario para converger: 818.1

Podemos observar que aumentando el número de iteraciones máximas, se converge con todos los vectores aleatorios, aunque algunos tarden más que otros en hacerlo. Obtenemos un valor medio de iteraciones necesario para converger de 818.1, siendo prácticamente el doble que con el vector a cero.

Con distintos vectores aleatorios, además, se obtienen coeficientes del hiperplano que divide los puntos diferentes; esto probablemente se deba a que dependiendo de los valores de los puntos iniciales, estarán más cerca de un plano u otro.

La ***elección de un buen punto de partida*** es decisivo en el número de épocas que se ejecutan hasta que la función converge; y un buen punto de partida nos ahorrará notablemente el número de iteraciones que necesitaremos para obtener los coeficientes del hiperplano que separará los elementos.

2) **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**.

- con el vector a cero : v_ini = np.array([0.0,0.0,0.0])

Peso inicial	Número iteraciones
[0,0,0]	2000

- con vectores de números aleatorios en [0,1](10 veces)

Peso inicial	Número iteraciones
[0.39000771 0.48599067 0.60431048]	2000

[0.54954792 0.92618143 0.91873344]	2000
[0.39487561 0.96326253 0.17395567]	2000
[0.12632952 0.13507916 0.50566217]	2000
[0.02152481 0.94797021 0.82711547]	2000
[0.01501898 0.17619626 0.33206357]	2000
[0.13099684 0.80949069 0.34473665]	2000
[0.94010748 0.58201418 0.87883198]	2000
[0.84473445 0.90539232 0.45988027]	2000
[0.54634682 0.79860359 0.28571885]	2000

Valor medio de iteraciones necesario para converger: 2000

Como se puede observar, todas las ejecuciones acaban en el mismo número de iteraciones. Cuando los datos no están bien clasificados (debido al ruido en la muestra), el algoritmo Perceptron entra en un bucle infinito, el cual solo termina cuando llega al número máximo de iteraciones.

Esto sucede ya que los datos no son linealmente separables, es decir, los datos mal colocados por el ruido no pueden ser separados mediante un hiperplano. Por tanto, concluimos que con una muestra de datos no linealmente separables, ***el algoritmo PLA no convergerá nunca***.

b) Regresión Logística: En este ejercicio crearemos nuestra propia función objetivo f (una probabilidad en este caso) y nuestro conjunto de datos D para ver cómo funciona regresión logística. Supondremos por simplicidad que f es una probabilidad con valores 0/1 y por tanto que la etiqueta y es una función determinista de x.

1) **Implementar Regresión Logística(RL) con Gradiente Descendente Estocástico (SGD) bajo las siguientes condiciones:**

Inicializar el vector de pesos con valores 0.

Parar el algoritmo cuando ||**w**(**t**-1) - **w**(**t**)|| < 0,01, **donde w**(**t**) **denota el vector de pesos al final de la época t. Una época es un pase completo a través de los N datos**.

Aplicar una permutación aleatoria, 1, 2, . . . , **N**, **en el orden de los datos antes de usarlos en cada época del algoritmo**.

Usar una tasa de aprendizaje de η = 0,01

Pseudocodigo del algoritmo sgdRL:

```
def sgdRL(datos, label, diferencia = 0.01 , tasa_aprend = 0.01):
   w ← ([0.0,0.0,0.0])
   num_elem ←  datos.shape[0] #número de elementos
   normal ← INF

   w_actual  ← w

   while normal > diferencia:
      indices ← permutamos los indices (de la misma manera para datos y etiquetas)
      datos ←  datos[indices, :]
      label ←  label[indices]

      w ← w_actual

      for x, y ∈{1...len(datos), 1...len(labels)}:
         grad ←  gradiente(x, y, w_actual)
         w_actual ←  w_actual - (tasa_aprend * grad)

      normal ← normal(w-w_actual)

    return w_actual
```

Comenzaremos con el vector de pesos a 0, y permitimos que el algoritmo se ejecute mientras que cada época, definida mediante el valor de la normal de $w^{(t-1)} - w^{(t)}$ (es decir, ||w(t-1) - w(t)||) sea mayor que un tope (“diferencia”), cuyo valor será 0.01.

En cada época, se realizará una permutación aleatoria para no consultar lo datos siempre en el mismo orden antes de calcular el nuevo valor de w mediante un “batch” de tamaño N = 1 (tamaño recomendado en el pseucódigo proporcionado en las transparencias).

La actualización del vector de pesos viene dada por:

$$\text{Update the weights: } \mathbf{w}(t+1) = \mathbf{w}(t) - \eta \nabla E_{\text{in}}$$

donde n es la tasa de aprendizaje, la cual en nuestro problema tomará valor 0.01 y ∇E_{in} estará compuesto por la expresión:

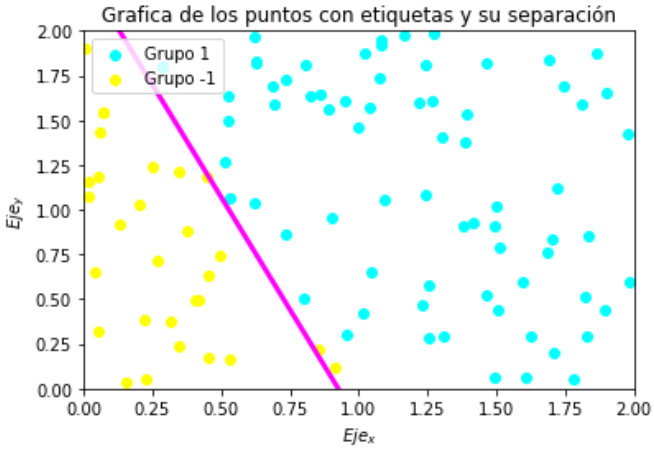
$$\nabla E_{\text{in}} = -\frac{1}{N} \sum_{n=1}^{N} \frac{y_n \mathbf{x}_n}{1 + e^{y_n \mathbf{w}^{\intercal}(t) \mathbf{x}_n}}$$

Esta expresión ha sido implementada en una función auxiliar:

def gradiente(x, y, w):
return (-(y*x)/(1 + e^{ywTx})

2) **Usar la muestra de datos etiquetada para encontrar nuestra solución g y estimar Eout usando para ello un número suficientemente grande de nuevas muestras** (>999).

Inicialmente, utilizo los datos indicados en el enunciado del problema para ejecutar el algoritmo sobre 100 puntos aleatorios:



w = [-6.26565779 6.76837873 2.69024665]

Ein: 0.09751693574420742

Podemos observar que se ha generado una recta capaz de dividir prácticamente de forma perfecta nuestro conjunto, a excepción de un par de puntos mal ajustados por encima de la recta. Para acompañar a la gráfica, hemos calculado el error de este conjunto de entrenamiento.

El ***error logístico*** viene dado por la expresión:

$$Err(w) = \frac{1}{N}\sum_{i=0}^{N} \ln(1 + e^{-y_i w^T x_i})$$

de manera que nuestra función para calcularlo tendrá la siguiente estructura:

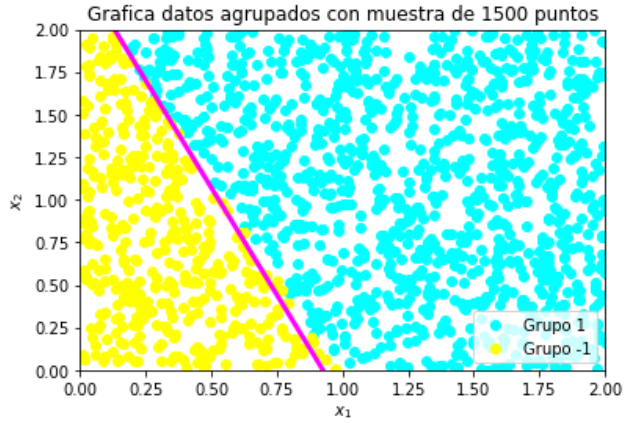
```
def calcula_error(data, labels, w):
  N ← data.shape[0] #Número de elementos
  error ← 0.0

  for x, y ϵ{1...len(datos), 1...len(labels)}:
    error += ln(1+ e^-ywTx)

  return error / N
```

El error (E_{in}) que obtenemos es muy pequeño, lo cual nos indica que nuestros datos de entrenamiento han podido ***entrenar al modelo*** y se ha producido ***un buen ajuste***.

Ahora, generamos una muestra nueva de tamaño 1500, nuestra muestra de test, y mostramos sus valores junto a la recta generada con los valores obtenidos del sgdRL:



Eout: 0.11683539997017482

Vemos como nuestra solución g (la recta rosa) que queríamos calcular parte el conjunto separando la nueva muestra casi perfectamente, exceptuando unos pocos puntos amarillos que quedan al otro lado y que el valor Eout, a pesar de ser algo más grande que Ein, sigue siendo muy bajo.

Con estos resultados podemos concluir que la muestra de entrenamiento usada representa muy bien al modelo y que este ha sido entrenado. Por eso, el modelo ***ha aprendido*** el patrón que hay entre los datos de entrada y los resultados; de manera que nos asegura un buen funcionamiento ante nuevos datos.

Ejercicio 3: BONUS

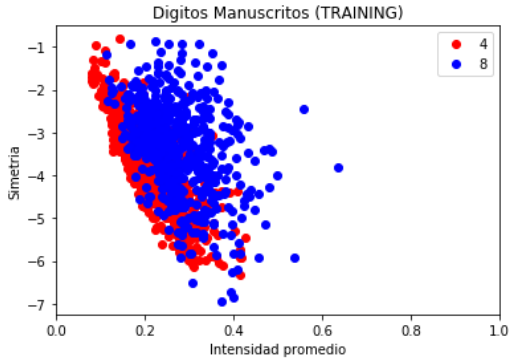
3) **Clasificación de Dígitos. Considerar el conjunto de datos de los dígitos manuscritos y seleccionar las muestras de los dígitos 4 y 8. Usar los ficheros de entrenamiento (training) y test que se proporcionan. Extraer las características de intensidad promedio y simetría en la manera que se indicó en el ejercicio 3 del trabajo 1.**

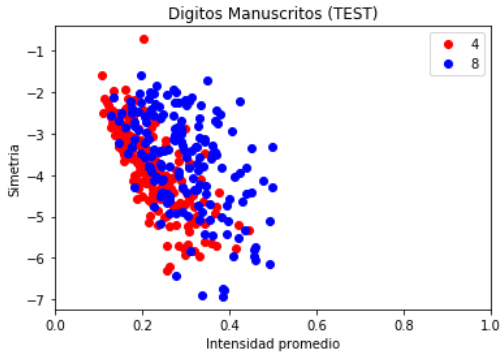
a) Plantear un problema de clasificación binaria que considere el conjunto de entrenamiento como datos de entrada para aprender la función g.

Nuestro objetivo en este problema es realizar un problema de clasificación de las clases cuyas muestras tienen los dígitos 4 y 8, a partir de la intensidad promedio y simetría.

b) Usar un modelo de Regresión Lineal y aplicar PLA-Pocket como mejora. Responder a las siguientes cuestiones.

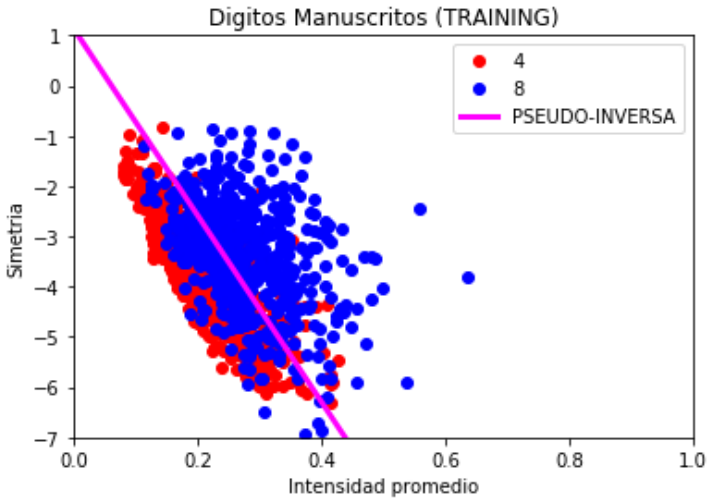
1) **Generar gráficos separados (en color) de los datos de entrenamiento y test junto con la función estimada.**





2) **Calcular Ein y Etest (error sobre los datos de test).**

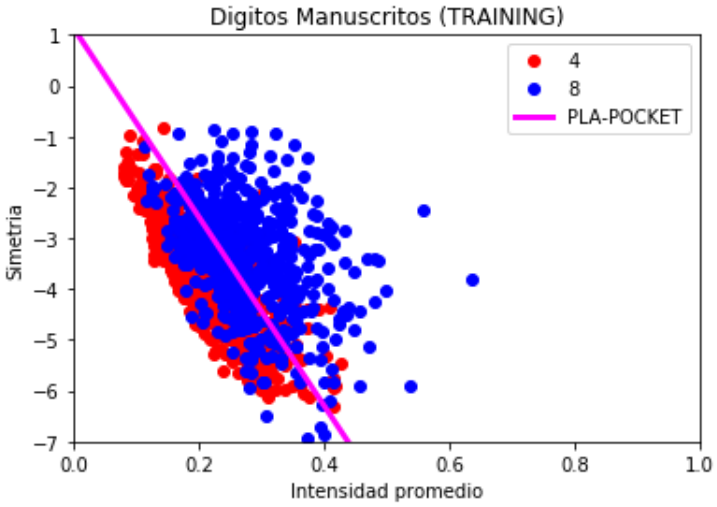
Aplicando el modelo de regresión lineal usado, la pseudoinversa (ya definida en la práctica 1)



Ein: 0.642853296336777

Eout: 0.7087148141159985

Aplicando el algoritmo PLA-Pocket:



Ein: 0.477796848423598

Eout: 0.5610310143360734

Como podemos comprobar, para ninguno de los dos modelos obtenemos resultados buenos, de ahí a que los valores del Ein y Eout sean tan altos. Esto se debe a que los datos están superpuestos unos sobre otros, y no son linealmente separables.

3) **Obtener cotas sobre el verdadero valor de Eout. Pueden calcularse dos cotas una basada en Ein y otra basada en Etest. Usar una tolerancia δ** = 0,05. **¿Que cota es mejor**?

Obtenemos las cotas sobre el verdadero valor de EOut

Cota superior de Eout (**con Ein**): 0.5171002438282252

Cota superior de Eout (**con Etest**): 0.6320201177686888

Es mejor la cota del Eout sobre Ein puesto que se obtiene el valor más pequeño.

Bibliografía

https://www.youtube.com/watch?v=8dXYLwPiqME

https://iartificial.net/analisis-de-errores-en-machine-learning/

https://docs.python.org/3/library/