Práctica 2

Celia Arias Martínez

Ejercicio 1

En este ejercicio vamos a estudiar cómo se comportan determinadas funciones que definen la frontera de clasificación al intentar separar puntos de una muestra que contiene ruido. Para ello utilizaremos las tres funciones proporcionadas en el template: *simula_unif*, *simula_gaus* y *simula_recta*.

Ejercicio 1.1

En este apartado vamos a dibujar dos gráficas de nubes de puntos, cada una con una distribución diferente. Las dos tienen un tamaño de puntos igual a `50` y dimensión `2`.

La primera gráfica de nubes de puntos sigue una distribución uniforme y los puntos están en un rango de $[-50, 50]$.

El código es:

```
def simula_unif(N, dim, rango):
    return np.random.uniform(rango[0],rango[1],(N,dim))

x_1 = simula_unif(50, 2, [-50,50])
plt.scatter(x_1[:,0], x_1[:,1], c ='r')
plt.show()
```

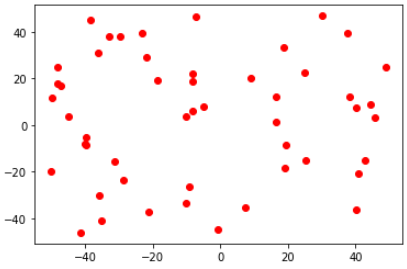


Figura 1: Distribución uniforme

La segunda gráfica de nubes sigue una distribución de Gauss de parámetros $\mu = 0$ y $\sigma = [5, 7]$. Para generarla vemos que llama a la función *simula_gaus*. Esta función fija la media a cero, y luego llama a la función *np.random.normal* para que genere el conjunto de puntos siguiendo una distribución normal, y utilizando para cada columna un σ determinado. En este caso para el eje `x` utilizamos $\sigma = 5$, y para el eje `y` $\sigma = 7$.

El código explicado es:

```
def simula_gaus(N, dim, sigma):
    media = 0
    out = np.zeros((N,dim),np.float64)
    for i in range(N):
        out[i,:] = np.random.normal(loc=media, scale=np.sqrt(sigma), size=dim)
    return out
```

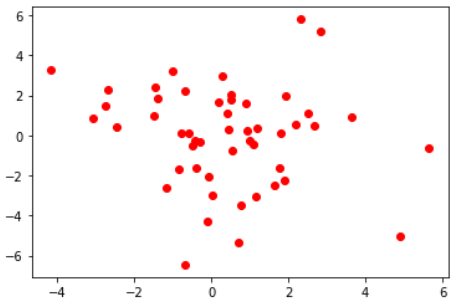


Figura 2: Distribución Gaussiana

Ejercicio 1.1

En este apartado introducimos una función para etiquetar los puntos y un ruido.

Primero dibujamos una nube de puntos que sigue una distribución uniforme, según lo explicado en el apartado anterior. En este caso el tamaño de puntos será de `100`, la dimensión seguirá siendo `2`, y el intervalo será $[-50, 50]$

La recta que usamos para etiquetar los puntos es $f(x, y) = y - a * x - b$ donde los parámetros `a` y `b` los hemos obtenido con la función *simula_recta*.

La recta obtenida es :

$f(x, y) = y + 0{,}6771584922002485 * x + 18{,}89022818933684$

Dibujamos el gráfico de puntos generado, así como la recta usada para etiquetar.

```
x_3 = simula_unif(100, 2, [-50,50])
etiquetas=[]
a,b = simula_recta([-50,50])
```

```
for i in range(0,len(x_3)):
    etiquetas.append(f(x_3[i,0], x_3[i,1], a,b))
etiquetas = np.asarray(etiquetas)
t = np.linspace(min(x_3[:,0]),max(x_3[:,0]), 100)
plt.scatter(x_3[:,0], x_3[:,1], c =etiquetas)
plt.plot( t, a*t+b, c = 'red')
plt.show()
plot_datos_cuad(x_3, etiquetas_originales,g0_to_vector )
```

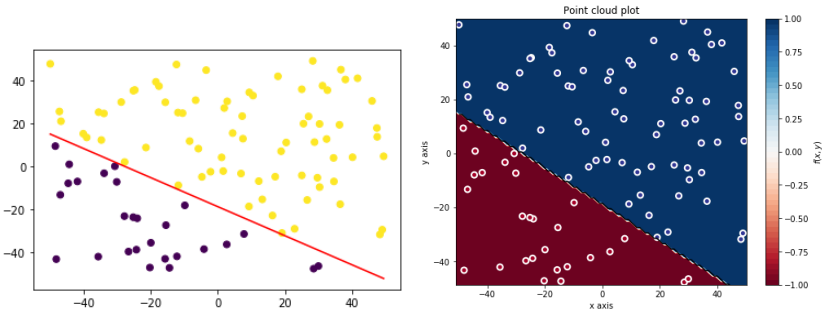


Figura 3: Etiquetas sin ruido

He definido una función para calcular el porcentaje de puntos mal etiquetados, que en este caso no tiene mucha utilidad porque sabemos que sera del 0 %, pero la utilizaremos en los apartados siguientes.

El código es:

```
"""
calculaPorcentaje: calcula la proporción de puntos mal clasificados
        x: muestra de puntos
        y: vector con las etiquetas
        g: función que clasifica los puntos de la muestra

        porcentaje_mal: proporción de puntos mal clasificada
"""
def calculaPorcentaje(x, y, g):
    mal_etiquetadas = 0
    for i in range(0,len(x[:,0])):
        etiqueta_real = y[i]
        etiqueta_obtenida = g(x[i,0], x[i,1])
        if (etiqueta_real != etiqueta_obtenida):
            mal_etiquetadas+=1

    porcentaje_mal = mal_etiquetadas / x[:,0].size

    return porcentaje_mal
```

Esta función lo que hace es calcular el valor de una variable llamada `mal_etiquetadas`, recorriendo todos los vectores de características y viendo si la etiqueta asignada coincide con la real. Por último calcula la media y devuelve dicho valor.

Ejercicio 1.2

En este apartado modifico de forma aleatoria un 10 % de los valores positivos, y un 10 % de los valores negativos. Para eso he creado dos vectores auxiliares, uno con las posiciones de los valores positivos, y uno con las de los negativos. Después he seleccionado aleatoriamente un 10 % de números entre 0 y el tamaño del vector de positivos, y esos valores los he utilizado de índices en el vector de positivos, con lo que he obtenido un 10 % de los índices de los valores positivos del vector original. Con los valores negativos he hecho lo mismo, he concatenado los dos vectores y he cambiado los valores de los índices obtenidos.

```
positivas = np.where(etiquetas == 1)
negativas = np.where(etiquetas == -1)
positivas = np.asarray(positivas).T
negativas = np.asarray(negativas).T
ind_pos = np.random.choice(len(positivas), int(0.1*len(positivas)), replace = True)
cambiar_signo = positivas[ind_pos,:]
ind_neg = np.random.choice(len(negativas), int(0.1*len(negativas)), replace = True)
cambiar_signo = np.concatenate((cambiar_signo, negativas[ind_neg,:]), axis=0)
for i in range(0, len(cambiar_signo)):
    etiquetas[cambiar_signo[i]]=-etiquetas[cambiar_signo[i]]
```

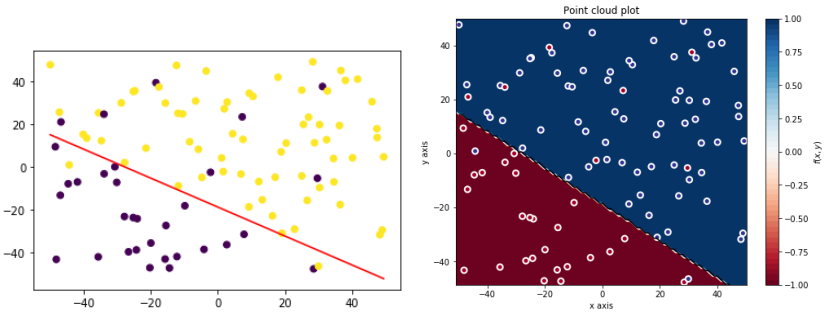


Figura 4: Etiquetas con ruido

Porcentaje mal etiquetadas: 0.09

Ejercicio 1.3

Tenemos ahora cuatro funciones diferentes, y vamos a ver cómo separan la muestra que tenemos, y cómo se comportan en cuánto al ruido.

Ejercicio 1.3.1

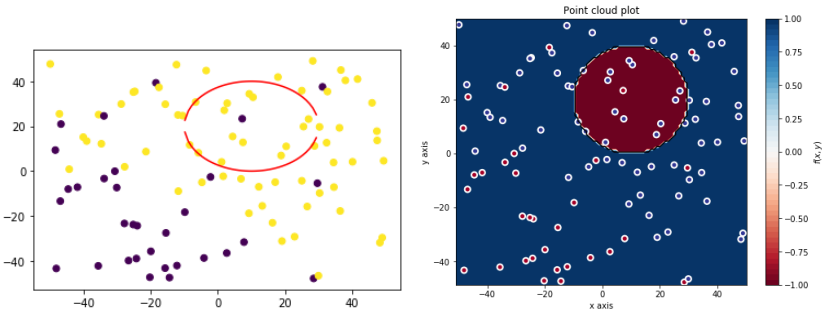


Figura 5: $f(x, y) = (x - 10)^2 + (y - 20)^2 - 400$

Porcentaje mal etiquetadas: 0.44

Ejercicio 1.3.2

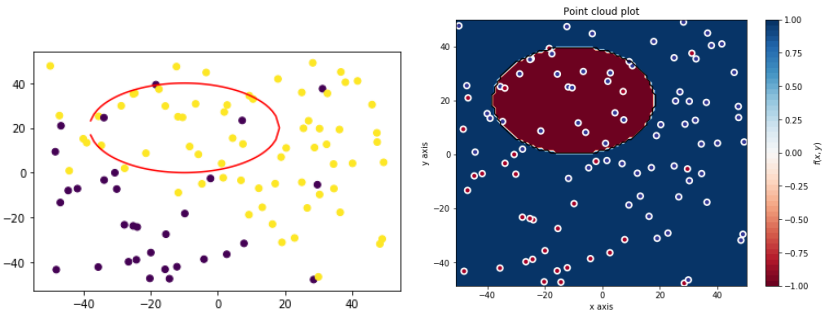


Figura 6: $f(x, y) = 0{,}5 * (x + 10)^2 + (y - 20)^2 - 400$

Porcentaje mal etiquetadas: 0.5

Ejercicio 1.3.3

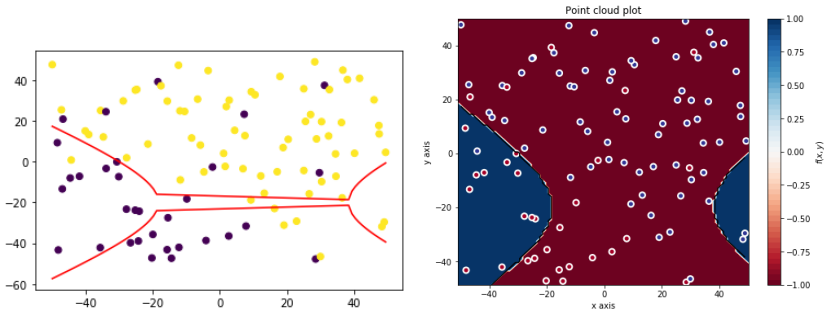


Figura 7: $f(x, y) = 0{,}5 * (x - 10)^2 - (y + 20)^2 - 40$

Porcentaje mal etiquetadas: 0.77

Ejercicio 1.3.4

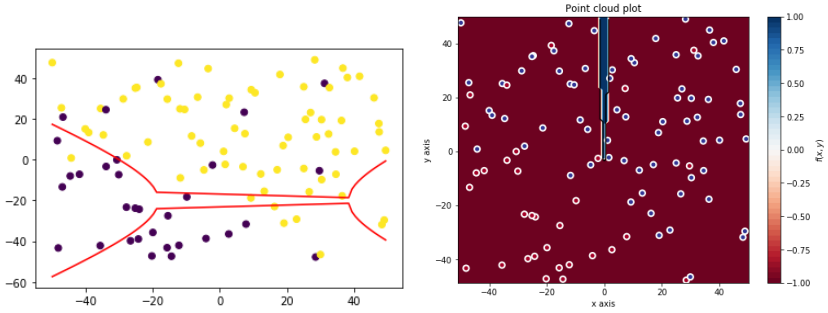


Figura 8: $f(x, y) = y - 20 * x^2 - 5 * x + 3$

Porcentaje mal etiquetadas: 0.68

Conclusiones

Lo primero que hay que resaltar es que en estos ejercicios no hay aprendizaje, por lo tanto vamos a tener siempre un problema de base, ya que va a depender de la casualidad que la función se ajuste bien a los datos o no.

También podemos extraer de los resultados que hemos obtenido que no siempre una función más compleja va a clasificar mejor los datos: en este caso la frontera era una recta, por eso no tiene sentido pensar que una función cuadrática va a explicarlos mejor. Lo mismo pasará en los problemas de aprendizaje reales en

los que no conozcamos la función objetivo: puede ser que una función compleja se adapte mejor pero tampoco tiene por qué pasar así.

Por último, vamos a realizar un experimento para analizar cómo influye el ruido según nuestra función frontera. Para ello vamos a etiquetar los puntos según la función:

$f(x, y) = 0{,}5 * (x - 10)^2 - (y + 20)^2 - 40$

y vamos a ver qué porcentaje de puntos mal clasificados obtenemos al introducir ruido:

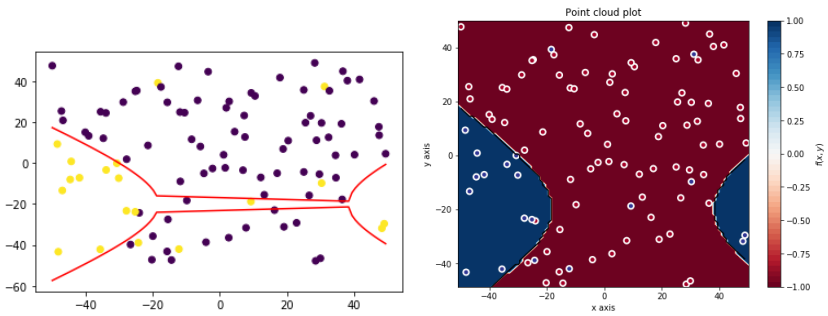


Figura 9: $f(x, y) = 0{,}5 * (x - 10)^2 - (y + 20)^2 - 40$

Porcentaje mal etiquetadas: 0.07

Vemos que el porcentaje de puntos mal clasificados ha bajado ligeramente, de 0.9 a 0.7. Esto creo que puede deberse a que la frontera de decisión tiene en este caso más superficie, y esto hace que haya más probabilidad de que algún punto pueda caer justo en el punto medio de cada hiperplano. Sin embargo creo que a grandes rasgos el efecto del ruido en uno y otro modelo será parecido, pues siempre tendremos alrededor de un 10 % de puntos mal clasificados -los que corresponden al error estocástico-, que no podremos modelar de ninguna forma.

Ejercicio 2

En el ejercicio anterior nos limitamos a estudiar cómo se comportaban diferentes funciones respecto a los datos, pero no había ningún proceso de aprendizaje pues las funciones estaban dadas. En este ejercicio vamos a implementar dos técnicas de aprendizaje lineal: el **perceptron** y **regresión logística**.

Ejercicio 2.1

El algoritmo **perceptron** ajusta los pesos según una función signo, y obtiene como resultado un hiperplano que separa el espacio en dos regiones. En el caso

de que el problema sea separable sabemos que la convergencia está asegurada, aunque el número de iteraciones dependerá de factores como el punto de inicio y la distribución de la muestra. En problemas no separables sabemos que no podrá converger nunca.

El código de dicho algoritmo es:

```
"""
ajusta_PLA: algoritmo perceptron
    datos: muestra de entrenamiento
    label: etiquetas de dichos datos
    max_iter: máximo de iteraciones que puede realizar
    vini: vector con el punto inicial

    w: vector de pesos encontrado
    iteraciones: número de iteraciones realizadas
"""
def ajusta_PLA(datos, label, max_iter, vini):
    w = np.copy(vini)
    iteraciones = 0
    for i in range ( 0, max_iter):
        iteraciones+=1
        stop = True
        for j in range (0, len(datos)):
            if(signo(w.T.dot(datos[j,:]).reshape(-1,1)) != label[j]):
                stop = False
                w = w + label[j]*datos[j,:].reshape(-1,1)

        if (stop):break

    return w, iteraciones
```

Lo primero que hace la función es inicializar el vector de pesos `w` con el valor del punto inicial. Después tenemos dos bucles: el bucle interior recorre todos los puntos de la muestra, y si el punto está mal clasificado corrige el vector `w` para adaptarlo a él. El bucle exterior se encarga de repetir el proceso las veces necesarias para que podamos recorrer todos los puntos sin cambiar nada, es decir que todos los puntos estén bien clasificados, o en su defecto hasta que lleguemos al número máximo de repeticiones.

El vector `w` se ajusta en cada iteración de acuerdo a esta fórmula:

$w_{new} = w_{old} + y_i * x_i$

Ejercicio 2.1.1

Vamos a probar ahora el algoritmo de **perceptron** con los datos del apartado 2a), es decir, con las etiquetas sin ruido. Vamos a variar el vector inicial para ver como influye en la convergencia hacia la solución.

Para este ejercicio no he utilizado la función proporcionada *plot_datos_cuad* porque he considerado que se veían mejor los cambios en las diferentes rectas generadas según el valor inicial si lo dibujaba solo con la recta generada, igual que en la práctica anterior.

Recordamos que la función frontera obtenida en el ejercicio anterior, y a partir de la cual etiquetamos los datos es:

$f(x, y) = y + 0{,}6771584922002485 * x + 18{,}89022818933684$

Los resultados obtenidos son:

- **Vector inicial cero:**

Número de iteraciones: 75

Porcentaje mal etiquetadas: 0.0

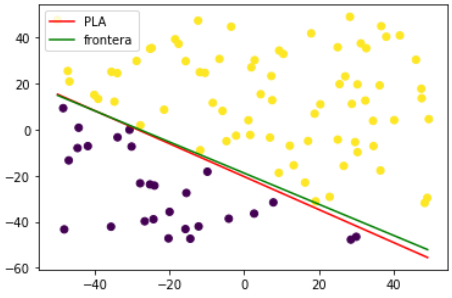
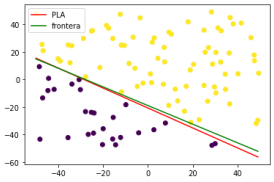
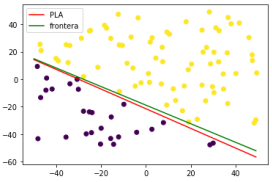


Figura 10: Perceptron, vector inicial 0

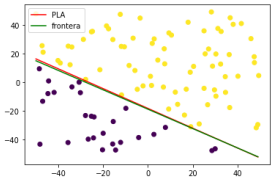
- **Vector inicial con números aleatorios entre [0,1]:**



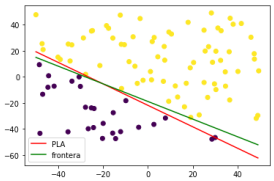
(a) Nº iteraciones: 70
Error : 0.0



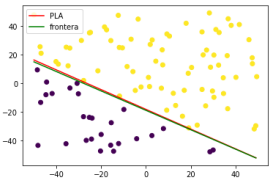
(b) Nº iteraciones: 84
Error : 0.0



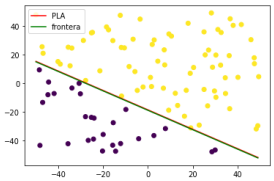
(c) Nº iteraciones: 122
Error : 0.0



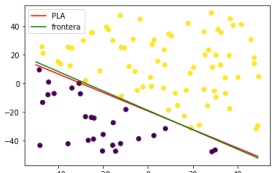
(d) Nº iteraciones: 37
Error : 0.0



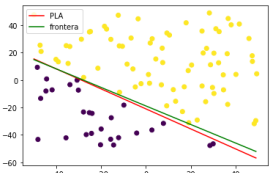
(e) Nº iteraciones: 119
Error : 0.0



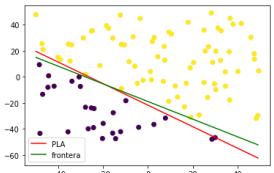
(f) Nº iteraciones: 76
Error : 0.0



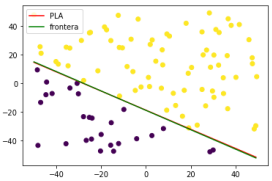
(g) Nº iteraciones: 43
Error : 0.0



(h) Nº iteraciones: 71
Error : 0.0



(i) Nº iteraciones: 37
Error : 0.0



(j) Nº iteraciones: 277
Error : 0.0

Figura 11: Perceptron, vector inicial aleatorio entre [0,1]

Número medio de iteraciones para converger: 93.6

Porcentaje medio de mal etiquetadas: 0.0

Ejercicio 2.1.2

Hacemos ahora el mismo experimento realizado antes con los datos del apartado 2b), es decir, con la introducción de ruido en las etiquetas.

- **Vector inicial cero:**

Número de iteraciones: 1000

Porcentaje mal etiquetadas: 0.18

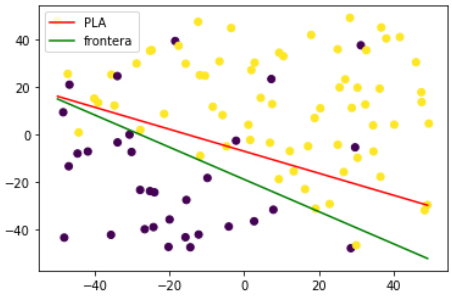
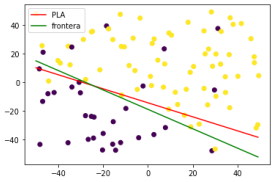
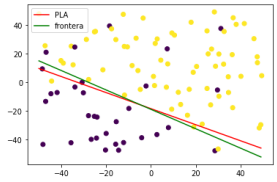


Figura 12: Perceptron, vector inicial 0 con ruido

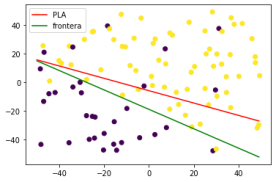
- **Vector inicial con números aleatorios entre [0,1]:**



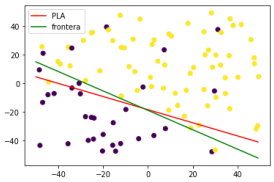
(a) Nº iteraciones: 1000
Error : 0.13



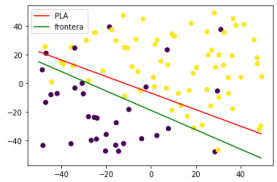
(b) Nº iteraciones: 1000
Error : 0.12



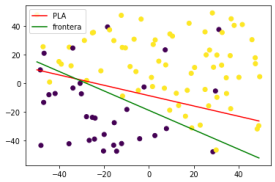
(c) Nº iteraciones: 100
Error : 0.19



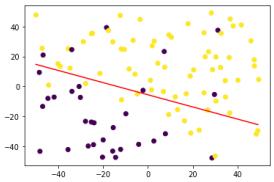
(d) Nº iteraciones: 1000
Error : 0.13



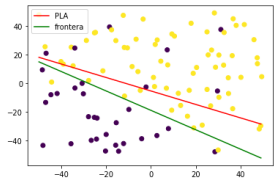
(e) Nº iteraciones: 1000
Error : 0.2



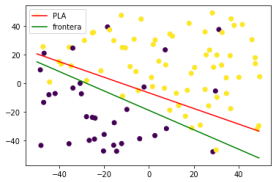
(f) Nº iteraciones: 1000
Error : 0.18



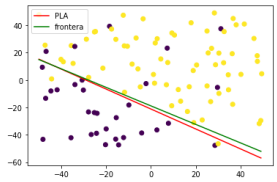
(g) Nº iteraciones: 1000
Error : 0.21



(h) Nº iteraciones: 1000
Error : 0.19



(i) Nº iteraciones: 1000
Error : 0.2



(j) Nº iteraciones: 1000
Error : 0.09

Figura 13: Perceptron, vector inicial aleatorio entre [0,1]

Número medio de iteraciones para converger: 1000.0 (en este caso no converge)

Porcentaje medio de mal etiquetadas: 0.164

Conclusiones

En primer lugar, podemos observar claramente que el algoritmo **perceptron** solo converge cuando los datos son separables. Vemos que cuando no tenemos ruido

los resultados son buenos siempre, ya que el problema es linealmente separable, pero al introducir ruido el algoritmo llega siempre al máximo de iteraciones, ya que nunca va a ser capaz de encontrar una recta que separe totalmente los datos.

Por otro lado el punto inicial influye mucho en los resultados que obtenemos. En el caso en el que no tenemos ruido vemos que el número de iteraciones necesarias para converger varía entre 37 y 277. La media de las iteraciones necesarias con los valores iniciales que hemos probado es 96.3, y con el vector cero tenemos 75, por lo que con los datos de los que disponemos podríamos decir que el vector cero no es un mal punto inicial, aunque creo que sería necesario disponer de más valores para poder decirlo con más certeza, pues vemos que la variabilidad es bastante alta.

En el caso del problema con ruido también tenemos muchas diferencias en los errores según los puntos iniciales, pues van desde 0.09 a 0.21. Para el valor inicial cero tenemos 0.18, algo superior a la media de lo que obtenemos con los valores aleatorios, por lo que en principio podríamos decir que no es un buen valor inicial. Por último añadir que, en el caso con ruido, en principio el error estaría acotado inferiormente por 0.1 -0.09 en este caso ya que al introducir ruido en las etiquetas tomamos valores enteros-.

Ejercicio 2.2

En este apartado vamos a aplicar el algoritmo de **regresión logística**. Este algoritmo está a medio camino entre **clasificación lineal o perceptron** y **regresión lineal**, ya que su función es un sigmoide en lugar de una función que crece a trozos o una recta. Además ahora nuestra función f es una función de probabilidad, que por simplificar suponemos que puede tomar valores 0 y 1.

El código de algoritmo es:

```
"""
sgdRL: algoritmo de regresión logística con N = 1
    x : matriz de características
    y: etiquetas
    max_iter: número máximo de iteraciones que puede realizar
    eta: learning rate
    epsilon: cota de error que permitimos

    w.T: vector de pesos encontrado
    iteraciones: número de iteraciones realizado

"""
def sgdRL(x, y, max_iter, eta ,epsilon):
    w = np.zeros((x[0].size,1)).reshape((1,-1))
    y = y.reshape((-1, 1))
    iteraciones= 0
```

```
    indices = np.arange(len(y))
    for i in range (0, max_iter):
        iteraciones+=1
        np.random.shuffle(indices)
        w_old = np.copy(w)
        for j in indices:
            exponencial = y[j]*((w.dot(x[j,:])))
            grad = -(y[j]*(x[j,:]))/(1+math.e**(exponencial))
            w = w -eta*grad

        if (np.linalg.norm(w_old-w) < epsilon): break

    return w.T, iteraciones
```

En este caso hemos utilizado un tamaño de *mini-bach* = 1.

Lo primero que hace la funcion es inicializar el vector inicial de pesos a 0. Como queremos que en cada iteración tengamos una disposición diferente en el orden de los datos lo que hacemos es crear un vector de índices que permutamos en cada iteración. Después hacemos una copia del vector w, que utilizaremos para calcular la norma de la diferencia. Luego recorremos todos los datos y actualizamos w según nuestra función de parada. Por último comprobamos el criterio de parada, que consiste en que w no haya cambiado significativamente.

El experimento que vamos a realizar tiene dos partes: una primera en la que utilizamos una muestra de entrenamiento para encontrar el vector de pesos w, y una segunda en la que utilizamos una muestra de test para validar los resultados, y ver si la predicción que hemos conseguido con los datos de entrenamiento se ajusta bien fuera de ellos.

2.2.1.Entrenamiento:

Tomamos una muestra uniforme de tamaño 100, dos puntos de dicha muestra y calculamos la recta que los une. Dicha recta será la frontera que utilizaremos para clasificar los puntos. Después procedemos como siempre: calculamos las etiquetas de dichos puntos y dibujamos la muestra.

Fijamos los parámetros :

- máximo de iteraciones : 1000
- learning rate : 0.01
- epsilon : 0.01

Llamamos a la función y los resultados son los siguientes:

Número de iteraciones: 344

Porcentaje mal etiquetados: 0.0

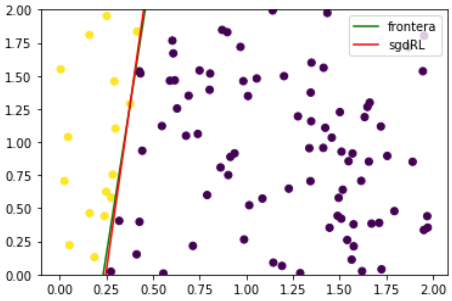


Figura 14: sgdRL

2.2.2. Test:

Tomamos ahora un tamaño de test igual 1000 y procedemos igual que antes, generamos una muestra de puntos distribuidos uniformemente, calculamos las etiquetas de dichos puntos y separamos la muestra según los pesos calculados en el entrenamiento.

Los resultados son:

Porcentaje mal etiquetados: 0.003

E_{out} : 0.0933

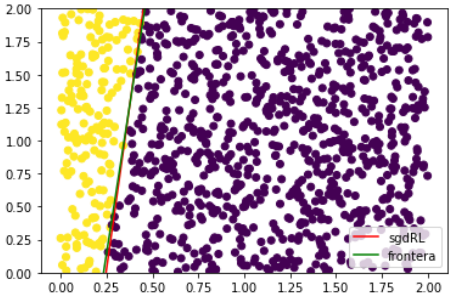


Figura 15: sgdRL

Conclusiones

Para poder analizar los resultados he utilizado el porcentaje de puntos mal etiquetados, ya que es un valor más intuitivo que el E_{out}. Podemos ver que el porcentaje es bastante bajo, lo que quiere decir que hemos podido aprender bastante bien la función, ya que al probarlo en los datos de test se comporta bien. El E_{out} también es bastante bajo, así que podemos decir que se comportará bien fuera de la muestra.

Vamos a estudiar ahora **regresión logística** en el caso en el que tenemos ruido.

Entrenamiento

Número de iteraciones: 64

Porcentaje mal etiquetados: 0.15

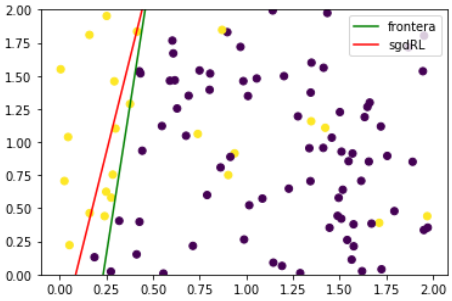


Figura 16: sgdRL

Test

Porcentaje mal etiquetados: 0.127

Eout : 0.421

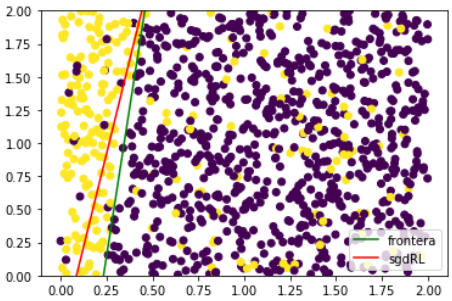


Figura 17: sgdRL

Podemos observar que, respecto a la muestra sin ruido, E_{out} sube lo cual es algo esperable, pues hemos introducido un 10 % de puntos mal etiquetados, y el porcentaje de puntos mal etiquetados también sube, pero llama la atención que se comporta mejor en los datos de test que en los de entrenamiento. Pienso que puede ser porque el ruido determinístico -el que genera el modelo- viene dado por la franja que hay entre la recta verde y la roja. Por tanto el porcentaje de error que añadirá será el que le corresponde a esa sección dentro de todo el espacio. Vamos a probar ahora con un tamaño de test igual a 10000, para ver si nuestras suposiciones son correctas.

Porcentaje mal etiquetados: 0.126

Eout : 0.414

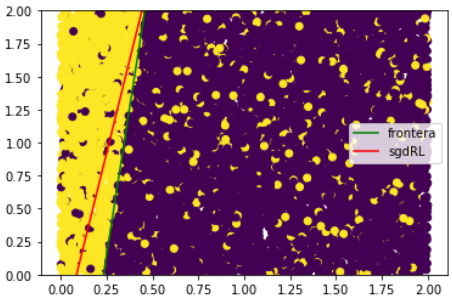


Figura 18: sgdRL

Vemos que, como habíamos supuesto, el porcentaje de mal etiquetadas se ha estabilizado en torno a 0.126, y, como sabemos que el error correspondiente al ruido en las etiquetas -error estocástico- es un 0.1, podemos decir que el error que corresponde al algoritmo será algo aproximado a 0.02, lo cual parece lo suficientemente pequeño.

Por último, comparando con el algoritmo de **perceptron**, y tomando en los dos vector inicial a 0, vemos que **regresión logística** se comporta mejor dentro de la muestra, pues comete un error de 0.15, en lugar de 0.18. De todas formas tenemos que tener en cuenta que nuestras rectas fronteras son diferentes en cada caso, y que deberíamos estudiar cómo se comporta **perceptron** fuera de la muestra, con lo cual no podemos tener mucha seguridad en nuestros resultados.