PRÁCTICA 2:

Complejidad de H y ruido. Modelos Lineales

Aprendizaje Automático

Ángel Cabeza Martín

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

Este ejercicio consistirá en generar distintas nubes de puntos y clasificarlas según distintas funciones, en las que observaremos la dificultad que el ruido añade a la hora de clasificar los puntos.

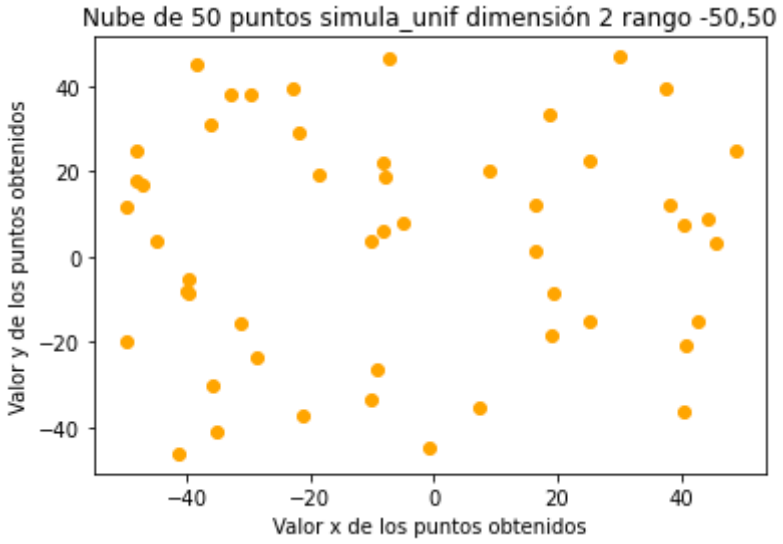
Ejercicio 1.

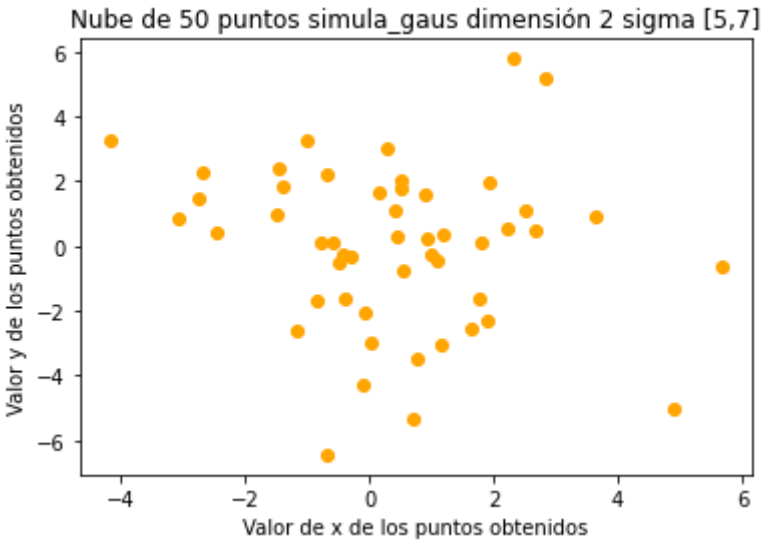
En este ejercicio se nos pide dibujar dos nubes de puntos, la primera generando los puntos de manera uniforme con la función `simula_unif` y la segunda utilizando una distribución gaussiana con la función `simula_gaus`.

La función `simula_unif` acepta tres parámetros, el número de puntos (N), la dimensión de los puntos (2D,3D,etc..) y el rango de las coordenadas de los puntos y devuelve N puntos en el rango indicado y con la dimensión indicada.

La función `simula_gaus` hace lo mismo que simula_unif pero en vez de generarlos en una distribución uniforme lo hace con una distribución normal.

Las nubes de puntos que he obtenido son estas:





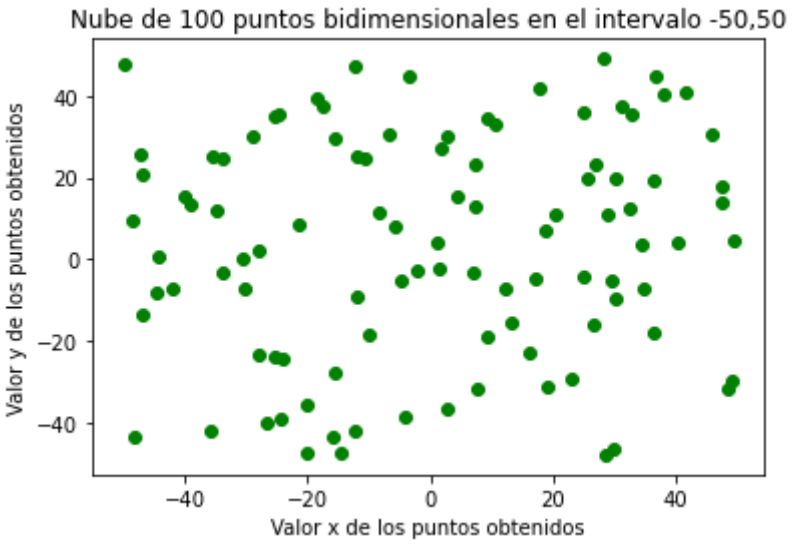
Vemos como en la nube de puntos que nos sale utilizando una distribución gaussiana, estos están en localizados en su mayoría en el centro del intervalo, mientras que en la nube de puntos que obtenemos de forma uniforme estos puntos se distribuyen por todo el espacio
disponible.

Hay que tener en cuenta que los intervalos que se piden en las dos nubes de puntos son distintos. En la nube de puntos uniforme el intervalo es [-50,50] y en la distribución gaussiana es [5,7].

Ejercicio 2.

En este ejercicio se nos pide generar etiquetar 100 puntos obtenidos de forma uniforme en el intervalo [-50,50] según el signo de la distancia de cada punto a una recta aleatoria que pasa por ese intervalo.

Los puntos que vamos a etiquetar son los siguientes (NO ESTÁN ETIQUETADOS EN ESTA IMÁGEN).



Para etiquetar los puntos, primero obtendremos la recta aleatoria utilizando la función `simula_recta,` que nos dará dos puntos aleatorios utilizando la función de la recta puesta abajo, con estos puntos aleatorios utilizaremos la siguiente función de la recta:

```
y = ax + b
```

para dibujarla donde a y b serán los puntos obtenidos con `simula_recta` y x serán los extremos del intervalo, es decir -50 y 50. Esta forma la usaremos para dibujar todas las rectas que vayan apareciendo a partir de ahora.

Una vez tenemos la recta aplicaremos la función signo (ambas funciones dadas en el template) para saber si un punto está por encima o por debajo de la recta que acabamos de generar, esta información es la que usaremos para etiquetar los puntos, es decir, estamos realizando un etiquetado binario donde el valor 1 corresponde a un punto que está por encima de la recta (dibujados en naranja) y -1 por debajo de la recta (dibujados en negro).

También he implementado una función para saber cuántos errores de media tiene una recta (una función) al clasificar unos puntos ya etiquetados con otra recta (otra función). Su pseudocódigo es el siguiente:

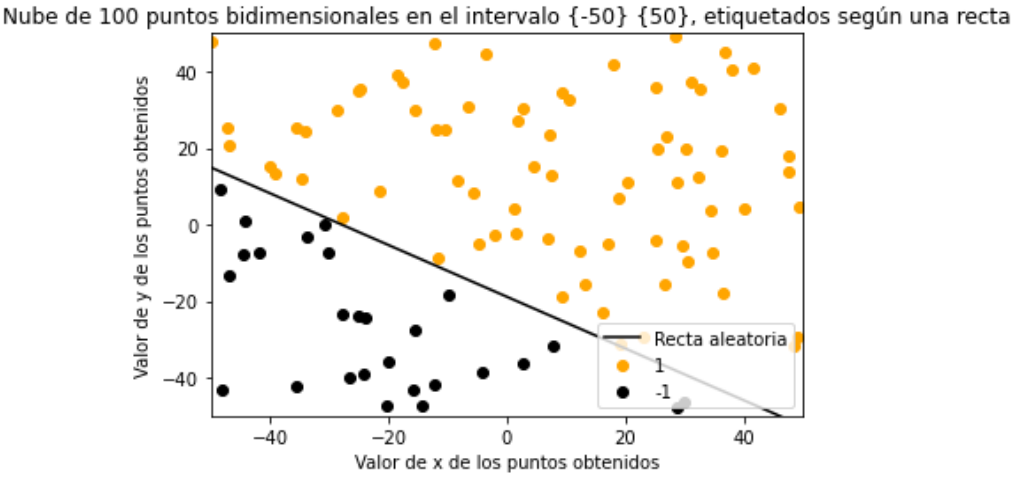
```
def num_errores(x,y,a,b)
        errores = 0

        Para i = 0 hasta len(x)
        |
        |       if signo(f(x[i][0],x[i][1],a,b)) != y[i]
        |       |
```

```
|       |       errores += 1
|       end
end

errores = errores / len(x)

return errores
```

end

No hay mucho que comentar respecto a la implementación, simplemente recorremos los datos y vamos comprobando si el signo respecto de nuestra recta (calculada a partir de la pendiente (a) y el término independiente (b)) es el mismo que la etiqueta que tiene asignado ese punto, si no es el mismo aumentamos el contador de errores a uno y si no pasamos al siguiente dato. Una vez hemos recorrido todos los datos hacemos la media de errores y lo devolvemos. Esta función la vamos a usar varias veces a lo largo de la práctica haciéndole pequeñas modificaciones para que en vez de aceptar la pendiente y el término independiente de una recta acepte un vector de pesos o una función pero el procedimiento es el mismo.

Una vez sabemos esto, dibujamos los puntos con su etiquetado correspondiente y obtenemos:



Porcentaje de errores en el etiquetado: 0.0

Podemos observar como los puntos están perfectamente etiquetados como era de esperar al utilizar como criterio de etiquetado si un punto está por encima o por debajo de la recta y al ser la nube de puntos linealmente separable.

En el siguiente apartado se pide que modifiquemos de forma aleatoria el 10% de las etiquetas positivas y el 10% de las etiquetas negativas y volvamos a dibujar la gráfica. Para hacer esto es muy importante aplicar el ruido teniendo en cuenta las etiquetas originales, porque si modificamos primero las positivas y luego, sobre el conjunto de puntos con las etiquetas positivas ya modificadas, aplicamos ruido en las positivas puede que cambiemos una etiqueta negativa a causa del ruido introducido a positiva nuevamente con lo que no estaríamos introduciendo ruido en ese caso.

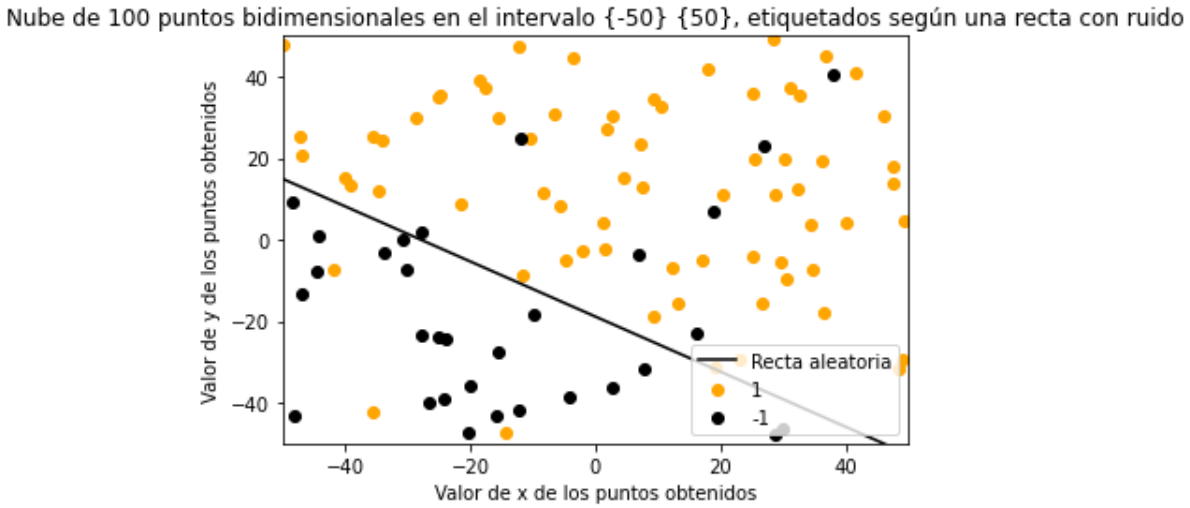
Para introducir el ruido que se nos pide de manera correcta, he creado la siguiente función (he hecho una función porque la vamos a utilizar varias veces más adelante):

```
def ruido (etiquetas):
      indices_pos = np.where(etiquetas == 1)

      indices_neg = np.where(etiquetas == -1)

      ruido_aplicar = len(etiquetas_pos) * 0.1
      indices = np.random.choice(indices_pos,ruido_aplicar)


      for i in indices
      |     etiquetas[i] = -etiquetas[i]
      end

      ruido_aplicar = len(etiquetas_pos) * 0.1
      indices = np.random.choice(indices_pos,ruido_aplicar)

      for i in indices
      |     etiquetas[i] = -etiquetas[i]
      end

      return etiquetas
end
```

En esta función lo primero que hacemos es obtener los índices de aquellos puntos con etiqueta positiva y etiqueta negativa y guardarlos en dos vectores distintos. Una vez hecho esto, nos vamos a centrar en aplicarle el ruido a las etiquetas positivas; para ello primero almacenamos en ruido_aplicar el número de etiquetas cuyo signo debemos cambiar y en índices los índices que vamos a modificar elegidos de

manera aleatoria. Una vez hecho esto recorremos el vector de índices aleatorios y cambiamos el signo del contenido. Para las etiquetas negativas repetimos el mismo proceso solo que utilizando el vector de etiquetas negativas.

Una vez hemos aplicado el ruido de la manera en la que hemos comentado obtenemos la siguiente gráfica.



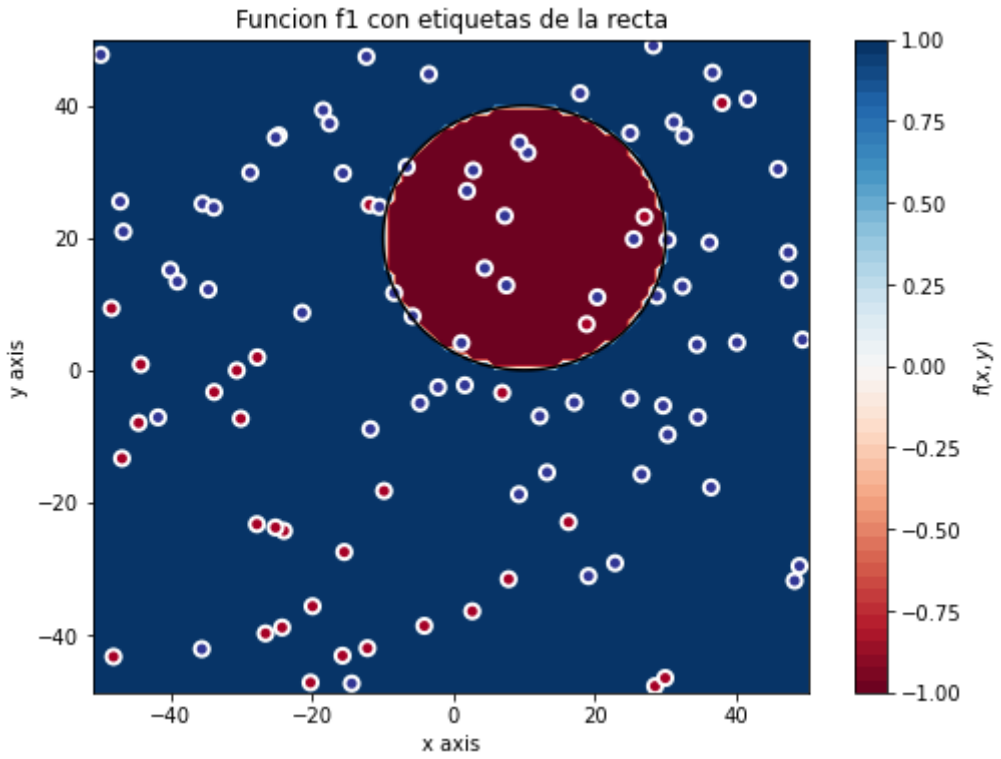
Porcentaje de errores en el etiquetado: 0.1

Y como es obvio, obtenemos un 10% de error cuando etiquetamos con esta recta.

Ejercicio 3

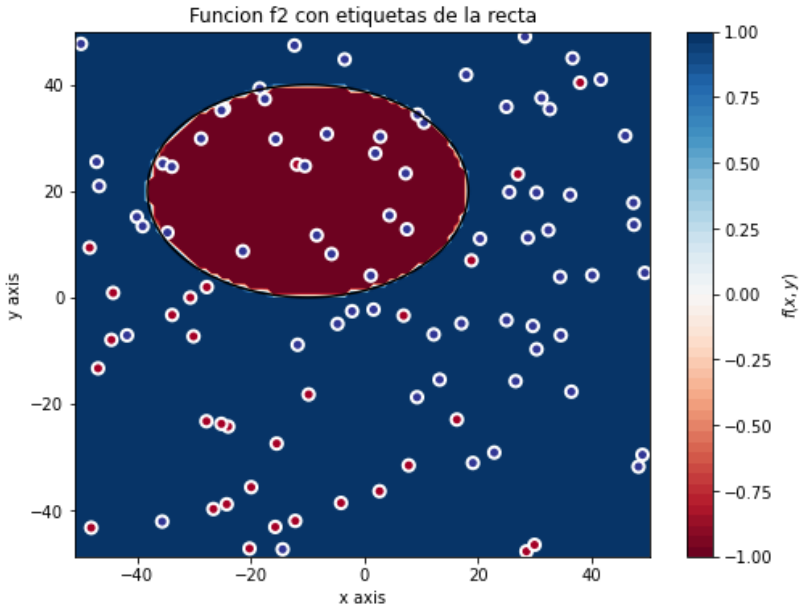
En este apartado debemos cambiar la función con la que dibujamos la frontera de clasificación, es decir vamos a intentar clasificar los puntos con ruido del apartado anterior utilizando funciones diferentes (lo entendí así y mi idea era hacerlo de las dos maneras pero al final no me dió tiempo). Estos son los resultados que he obtenido:

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



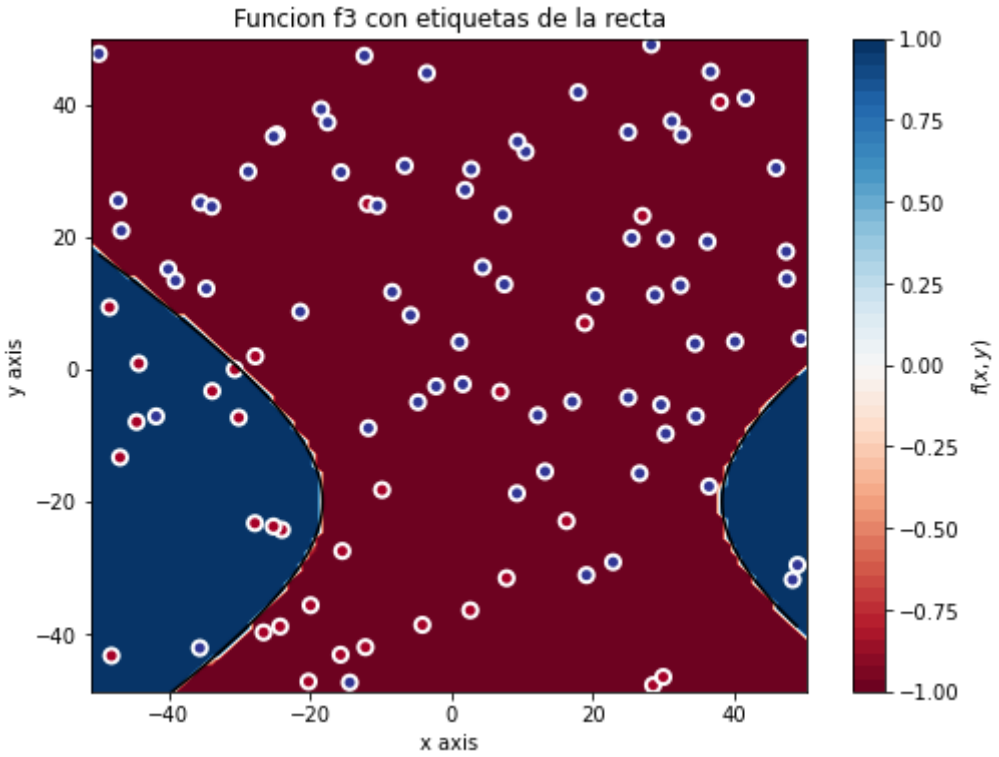
Porcentaje de errores en el etiquetado de f1: 0.41

Resultado función f(x,y) = 0.5(x + 10)² + (y -20)² - 400



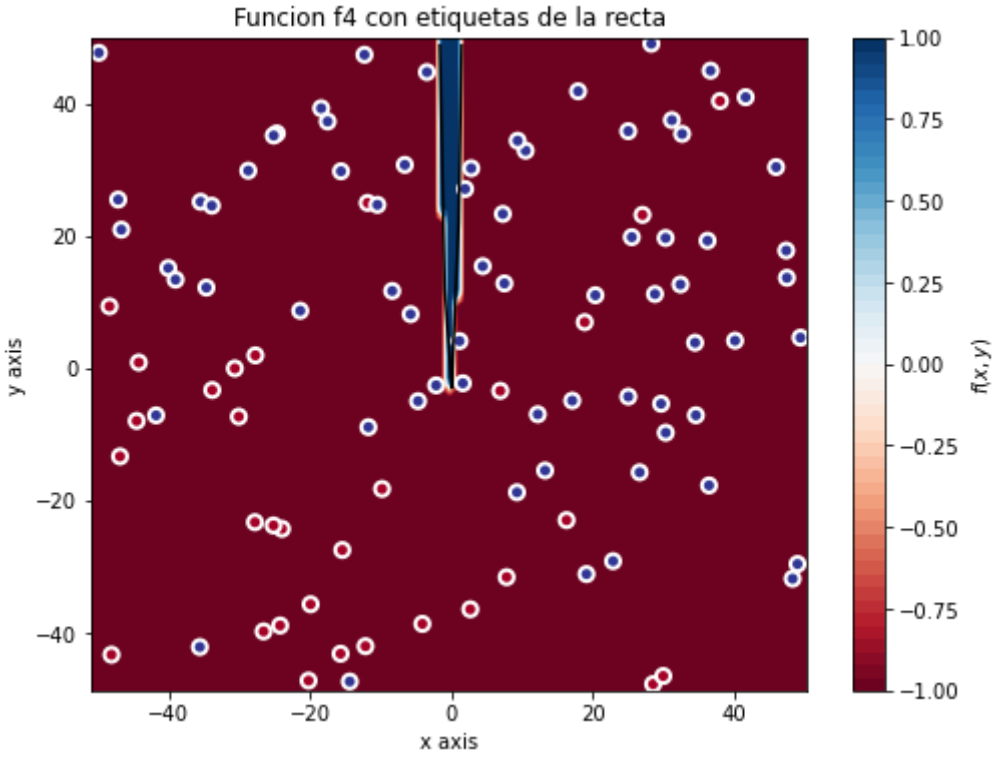
Porcentaje de errores en el etiquetado de f2: 0.51

Resultado función f(x,y) = 0.5(x - 10)² - (y - 20)² - 400



Porcentaje de errores en el etiquetado de f3: 0.76

Resultado función f(x,y) = y - 20x² -5x + 3



Porcentaje de errores en el etiquetado de f4 : 0.69

Vemos como todas los predictores que hemos obtenido utilizando las distintas funciones son muy malos, y es que el que una función sea más compleja no implica mejores clasificadores, ya que las etiquetas las hemos obtenido con una función lineal. En este sentido, la calidad de un clasificador dependerá de cómo se etiquetan los valores y el ruido aplicado, por ejemplo, si aplicamos ruido a las etiquetas cuya distancia a la recta es menor que un δ, la clasificación con otras funciones más complejas nos daría resultados mucho mejores.

De hecho,como llegamos a porcentajes muy grandes en los errores al etiquetar, ya que las clases más complejas no se adaptan de ninguna forma, puede ocurrir lo que pasa en el caso 4 que no etiqueta ningún punto azul en la zona azul y el es error más bajo que en la función 3 ya que todos los rojos sí los etiqueta de manera correcta.

Como conclusión el ruido hace que sea mucho más complejo obtener un buen clasificador debido a que la frontera es mucho más grande e incluso, como en este ejemplo, es posible que amplíe la frontera a todo el espacio en el que trabajamos. Si aplicamos el ruido exclusivamente a etiquetas cercanas a la función con la que etiquetamos, seguimos manteniendo la frontera, aunque un poco mayor.

2. Modelos Lineales.

En este ejercicio implementaremos dos métodos lineales, el algoritmo Perceptron y Regresión Logística con SGD (gradiente descendiente estocástico y vamos a estudiar cómo se comportan estos algoritmos.

Ejercicio 1

Este algoritmo va sobre el algoritmo Perceptron. Vamos a utilizar este algoritmo para calcular una recta que clasifique los puntos que hemos obtenido en el apartado anterior y vamos a ver cómo se diferencia lo que encuentra el algoritmo de la verdadera función f.

El algoritmo Perceptron se basa en actualizar el vector de pesos w dependiendo del signo de los datos, de forma que el vector de pesos se vaya modificando hasta que tenga todos los datos correctamente clasificados, lo que hará que, si los datos son linealmente separables, nos encontrará un óptimo en tiempo finito. Este algoritmo no

tiene ningún tipo de memoria por lo que si pasamos de un buen ajuste a uno peor, no podemos asegurar que vuelva a pasar por este.

Lo primero que he hecho ha sido implementar el algoritmo siguiendo el pseudocódigo dado en teoría. Mi algoritmo es el siguiente:

```
def ajustaPLA(datos, label, max_iter, vini)

    w = vini.copy()
    iteraciones = 0
    hay_mejora = True

    while hay_mejora and iteraciones < max_iter
    |    hay_mejora = False
    |
    |    Para cada dato i de datos
    |    |    valor = signo(w.T.dot(datos[i]))
    |    |
    |    |    Si el signo que obtenemos != la etiqueta verdadera
    |    |    |    w = w + label[i] * datos[i]
    |    |    |    hay_mejora = True
    |    |    end
    |    end
    |    iteraciones += 1
    end

    return w, iteraciones
end
```

En este algoritmo lo primero que hacemos es igualar el vector de pesos al punto inicial (vini), después creamos una variable llamada iteraciones que va a controlar el número de épocas del algoritmo para que no supere un máximo definido en max_iter y una variable booleana que va a controlar si hemos cambiado en una pasada a los datos el vector de pesos o no lo hemos cambiado.

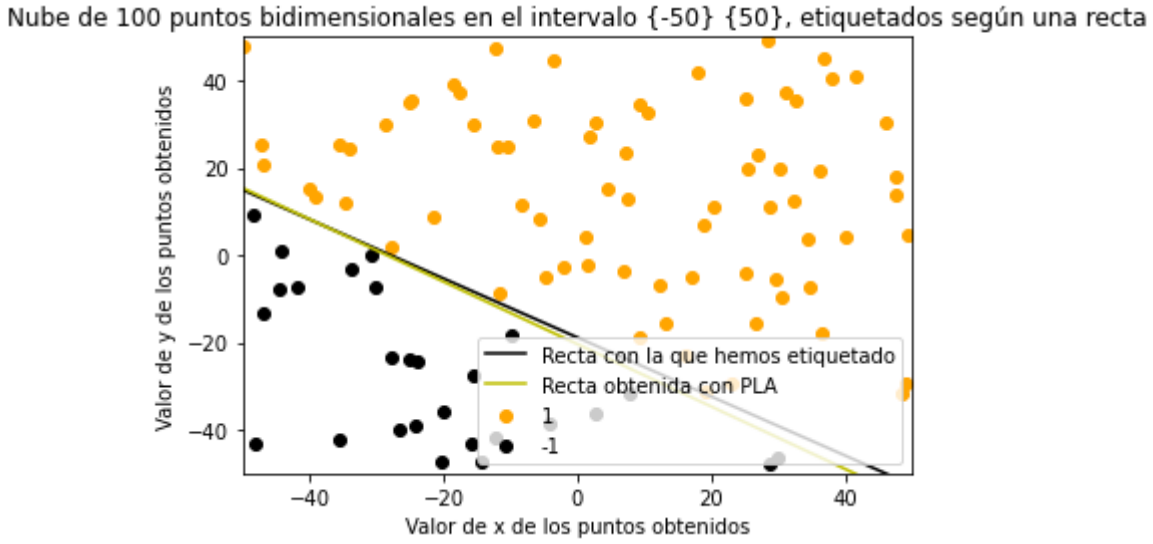
Una vez tenemos estas variables de control, nos metemos en el bucle principal e iteramos sobre él mientras hayamos cambiado el vector de pesos en una época entera o no hayamos superado las iteraciones máximas. Dentro de este bucle vamos a iterar sobre el vector de datos, y para cada dato vamos a calcular el signo de la traspuesta de w por el dato, es decir, el signo de la clase de funciones para el algoritmo Perceptron y si este signo es distinto al que debería ser (el asignado por la función f) actualiza el vector de pesos intentando corregir este error.

$$h(x) = \text{sign}(w^T x)$$

Una vez tenemos el algoritmo implementado, el ejercicio nos pide ejecutarlo con dos puntos iniciales. Estos son los resultados obtenidos:

Inicialización con el vector a 0.

Para este experimento he puesto un número de iteraciones máximo infinito porque sabemos que los datos son linealmente separables por lo que el algoritmo va a converger en tiempo finito. El resultado obtenido ha sido el siguiente:



Vemos como el algoritmo nos proporciona una recta que no comete ningún error en la clasificación de los puntos, sin embargo vemos que es distinta a la recta original (se puede apreciar en la esquina inferior derecha). Esto se debe a que el número de puntos es bajo, para este algoritmo es muy importante tener una muestra grande de datos ya que para el ajuste se usa únicamente los datos.

Para este experimento hemos necesitado 75 iteraciones.

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

Para este experimento también he puesto un tope de épocas infinito porque sabemos que va a converger, lo interesante de este experimento es ver cómo se comporta el algoritmo con puntos iniciales distintos.

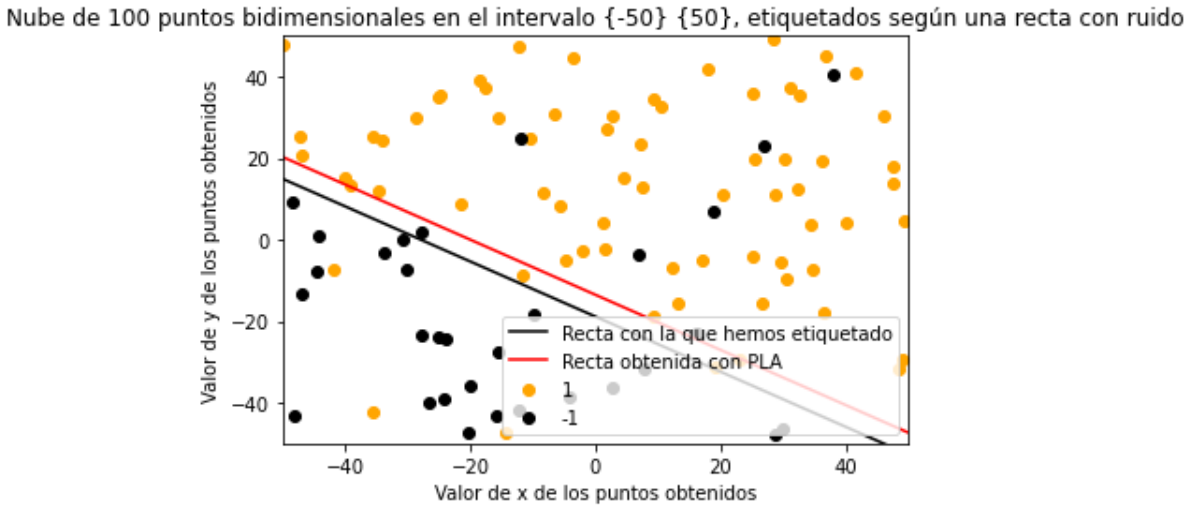
Punto Inicial	**W obtenida**	**Iteraciones**
[0.583, 0.928, 0.107]	[644.583, 23.507, 35.030]	76
[0.452, 0.155, 0.304]	[663.452, 22.126, 30.903]	84
[0.764, 0.626, 0.711]	[557.764, 19.958, 29.853]	59
[0.193,0.068,0.879]	[633.193, 22.177, 30.004]	69
[0.319, 0.604, 0.955]	[647.319, 23.411, 34.687]	81
[0.415, 0.195, 0.166]	[1148.415, 39.919, 60.904]	274
[0.624, 0.056, 0.328]	[692.624, 22.714, 34.605]	86
[0.980, 0.771, 0.209]	[551.980, 20.679, 30.108]	58
[0.585, 0.439, 0.613]	[828.585, 29.823, 44.775]	132
[0.319, 0.111, 0.313]	[701.319, 23.863, 35.112]	79

El valor medio de iteraciones que hemos necesitado ha sido de 99.8.

Vemos como el punto inicial influye mucho en el número de épocas que el algoritmo necesita para finalmente converger. Si empezamos con un punto inicial que da lugar a una recta que clasifica de manera errónea muchos puntos, el algoritmo va a necesitar muchas épocas para ir corrigiendo poco a poco esta recta y al final converger en una que clasifique bien todos los puntos (si los datos son linealmente separables), mientras que si empezamos con una recta que lo hace bien tardará menos en converger y por tanto irá más rápido. Lo que sí observamos es que da igual el punto de inicio que le demos, si los datos son linealmente separables el algoritmo va a converger.

Ahora vamos a hacer el mismo experimento pero los puntos van a tener ruido, con lo que van a dejar de ser linealmente separables y por tanto este algoritmo no va a ser capaz de converger.

Inicialización con el vector a 0.
Para este caso sí he puesto un número máximo de épocas porque como hemos dicho antes el algoritmo no va a ser capaz de converger y por tanto si no ponemos un máximo de iteraciones (o épocas) este ciclará infinitamente. El resultado que he obtenido es este:



Iteraciones realizadas: 10000.

Como vemos la recta que hemos obtenido es muy diferente a la recta que verdaderamente separa los puntos. Además como vemos hemos necesitado todas las épocas porque como ya hemos dicho el algoritmo no converge y obtenemos la siguiente recta porque llegamos al número máximo de épocas, por lo que si hubiéramos hecho menos iteraciones habríamos obtenido una solución distinta y puede que hasta incluso mejor o peor porque el algoritmo al no tener memoria puede volver a configuraciones peores.

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

Este experimento lo hemos hecho con las mismas condiciones que el anterior, es decir, 10000 épocas máximas.

Los resultados que hemos obtenido son:

Punto Inicial	**W obtenida**	**Iteraciones**
[0.993, 0.165, 0.462]	[389.993, 8.895, 36.042]	10000
[0.635, 0.037, 0.253]	[387.635, 9.096, 30.146]	10000
[0.870, 0.139, 0.018]	[386.870, 29.077, 45.285]	10000
[0.689, 0.871, 0.355]	[391.689, 18.306, 33.894]	10000
[0.396, 0.862, 0.932]	[388.396, 23.004, 48.910]	10000
[0.450, 0.974, 0.697]	[385.450, 17.190, 56.471]	10000
[0.753, 0.577, 0.134]	[392.753, 17.180, 35.037]	10000
[0.807, 0.104, 0.335]	[391.807, 29.973, 44.223]	10000
[0.564, 0.622, 0.458]	[383.564, 22.926, 46.714]	10000
[0.812, 0.248, 0.422]	[381.812, 12.245, 34.892]	10000

Como vemos en ninguna de las ejecuciones el algoritmo, sin importar el punto inicial, ha sido capaz de converger. El algoritmo ha parado cuando ha llegado al tope de épocas.

En conclusión, el algoritmo perceptrón es un algoritmo muy útil cuando los datos son linealmente separables ya que siempre va a encontrar una solución que separe a los datos de manera correcta. Sin embargo, con ruido no es una buena opción porque en cada iteración arreglará el vector de pesos para incluir al nuevo punto pero empeorará el vector para otros puntos que ya ha visitado y ciclará infinitamente.

Ejercicio 2

En este apartado implementaremos Regresión Logística con SGD. Para representar el problema, el conjunto de datos será una serie de 100 puntos escogidos aleatoriamente de forma uniforme en [0,2] x [0,2] y serán etiquetados de acuerdo a una recta que pasará por dos puntos aleatorios del conjunto de datos.

Los puntos los obtendremos con la función simula_unif y la recta la obtendremos con la función simula_recta como explicamos anteriormente.

Para la implementación de Regresión Logística usaremos el algoritmo SGD que implementé en la práctica inicial además del criterio de aprendizaje ERM (Empirical Risk Minimization), que en teoría hemos visto que se puede aplicar a Regresión Logística.

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

Y su respectivo gradiente:

$$\nabla E_{in} = -\frac{1}{N}\sum_{n=1}^{N} \frac{y_n x_n}{1+ e^{y_n w^T(t) x_n}}$$

E implementados en código quedan así:

```
def ERM (x,y,w)
      num_elementos = x.shape[1]

      error = np.float(0.0)

      Para todo i = 0 hasta num_elementos
      |
      |     error += np.log(1 + np.e**(-y[i]*w.T.dot(x[i])))
      |
      end

      error = error/num_elementos

      return error
end
```

Para calcular el ERM calculamos el valor de N, es decir, el tamaño de la muestra viendo cuantas filas tiene x. Inicializamos el error a 0 y procedemos a calcular la sumatoria de la fórmula anterior. Para hacer esto hacemos un bucle desde 0 hasta N y vamos acumulando los resultados que vamos obteniendo para cada i. Una vez tenemos la sumatoria solo nos queda dividirlo por el tamaño muestral y devolver el error.

```
def grad(x,y,w)
      return -(y *x)/(1 + np.exp(y * w.T.dot(x)))
end
```

Para calcular el gradiente simplemente hemos pasado a código la ecuación del gradiente y le hemos quitado la N porque en el algoritmo SGD que vamos a ver a continuación cogemos tamaños de minibatch de 1 (por recomendación de los profesores) por lo que estaríamos dividiendo entre uno, eso sí el signo negativo de la división hay que mantenerlo.

Ahora vamos a ver cómo hemos implementado la Regresión Logística con SGD:

```
def sgdRL (x,y,learning_rate,error2get = 0.0.1)
      w = np.zeros((x.shape[1],), np.float64)

      w_ant = w.copy()

      acabar = False
      epocas  = 0

      while not acabar
      |
      |     indices_minibatch = random_choice(x.shape[0],x.shape[0])
      |
      |     Para todo i = indices_minibatch
      |     |
      |     |     w = w - learning_rate * grad(x[i],y[i],w)
      |     end
      |
      |     epocas += 1
      |     dist = np.linalg.norm(w_ant - w)
      |
      |     Si dist < error2get
      |     |     acabar = False
      |     end
      |
      |     w_ant = w.copy()
      end

      return w, epocas
end
```

Para implementar este algoritmo primero inicializamos w a tantos 0 como elementos de x y creamos una variable llamada w_ant que almacenará la w de la época anterior. Inicializamos la variable de épocas y una variable de control para saber cuándo acabar y cuándo no y entramos en el bucle principal. En este bucle aplicamos una permutación aleatoria a los índices del minibatch y entramos en un bucle que recorrerá cada minibatch (de tamaño 1) e irá actualizando w. Cuando salimos de este bucle aumentamos el contador de épocas y calculamos la distancia entre la w actual y la anterior, si esta distancia es menor que el error acabamos el algoritmo ahí y si no seguimos con el algoritmo.

El ejercicio nos pide crear una recta aleatoria con la que clasificar 100 puntos aleatorios, ejecutar el algoritmo sobre estos puntos y luego calcular el error fuera de la muestra con una muestra de más de 1000 puntos, en mi caso yo he usado 10000 puntos y repetir este experimento con la misma recta aleatoria 100 veces y calcular el valor medio de épocas necesitadas y el valor medio de Eout.

La recta y los puntos los he obtenido como en los apartados anteriores y he obtenido los siguientes resultados:

Ein medio: 0.12286995516177984
Eout medio: 0.14918833957443545
Media de épocas necesitadas para converger: 425.05

Como vemos el algoritmo realiza una buena clasificación de los datos porque los errores que obtenemos son bajos, un poco más alto el valor fuera de la muestra que dentro como es normal pero al fin y al cabo bajo.

3. Bonus

En este ejercicio vamos a usar los datos de intensidad promedio y simetría en clasificación de números como en la práctica 1, pero esta vez de los números 4 y 8.

Para resolver este problema se nos pide usar un algoritmo de regresión lineal como los vistos en la práctica 1 y sobre la solución que este algoritmo nos de, aplicar el algoritmo Pocket como mejora.

Como algoritmo de regresión lineal he decidido usar el algoritmo de la pseudoinversa que no voy a comentar porque ya lo hice en la práctica anterior y los mínimos cuadrados como función de error

Y el algoritmo pocket lo he implementado de la siguiente manera:

```
def pocket(x,y,iteraciones,w_ini)

      w_mejor = w_ini
      ein_w_mejor = num_errores(x,y,w)
      w = w_mejor.copy()

      it = 0

      while it < iteraciones:
      |     w = ajusta_PLA(x,y,1,w.copy())
      |     ein_w = num_errores(x,y,w)
      |
      |     Si ein_w es mejor que la actual mejor
      |     |     w_mejor = w.copy()
      |     |     ein_w_mejor = ein_w
      |     end
      |
      |     it += 1
      end

      return w_mejor
end
```

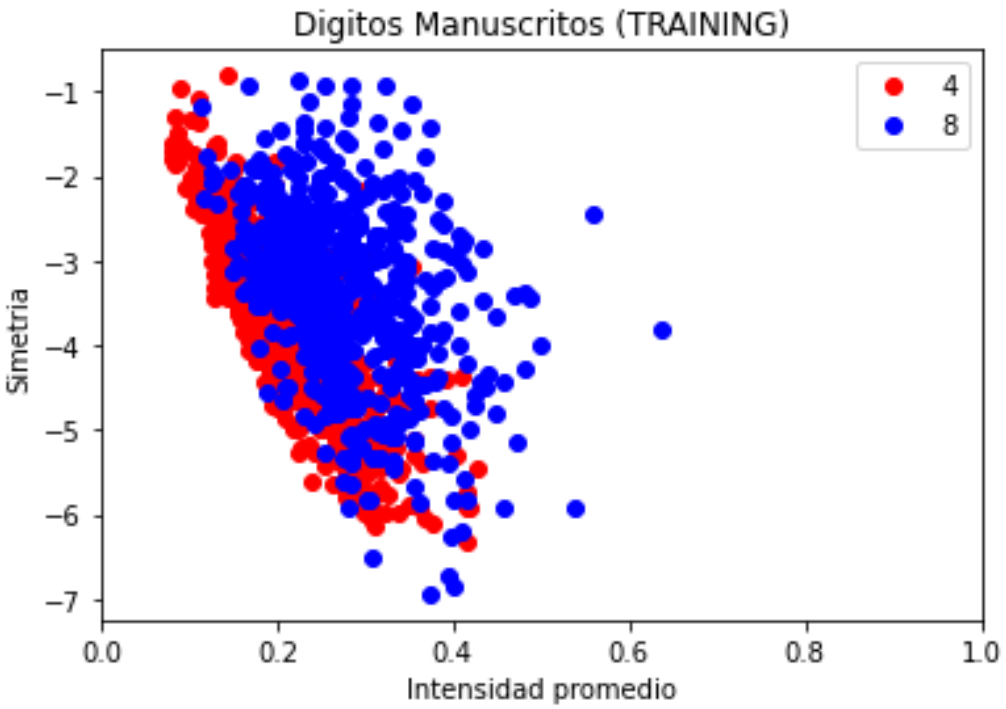
Como vemos en este algoritmo, lo primero que hacemos es crearnos unas variables que nos llevarán la cuenta del vector de pesos que produce el mínimo error dentro de la muestra y el mínimo error dentro de la muestra hasta ahora y una variable que llevará el control de las iteraciones que llevamos hechas. Una vez tenemos estas variables configuradas, nos metemos en el bucle principal del algoritmo en el que ejecutaremos una época del algoritmo Perceptrón para calcular un vector de pesos w, he decidido ejecutar solo una época ya que el algoritmo Perceptrón al no tener memoria es posible que si ejecutamos más de una época se salte el mínimo que buscamos.

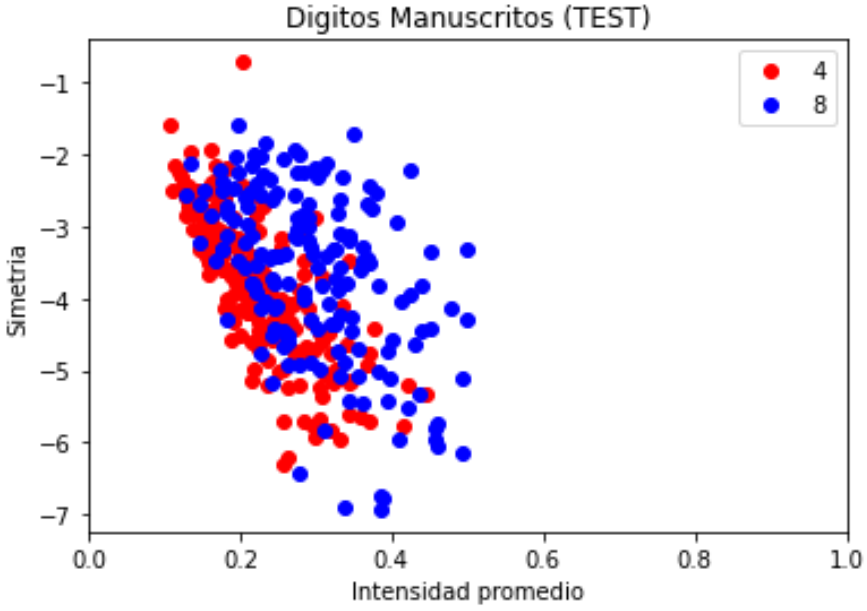
Cuando el algoritmo perceptrón ha actualizado el vector de pesos actual, sacamos su ein que será la media de los puntos que están mal clasificados y comprobamos si este ein es mejor que el mejor que tenemos almacenado, si es mejor actualizamos los valores de w_mejor y ein_mejor y pasamos a la siguiente iteración, si no es menor simplemente pasamos a la siguiente iteración.

La función de error del algoritmo Pocket es la media de los puntos que están mal clasificados, y la he implementado como en el primer ejercicio de esta práctica, solo

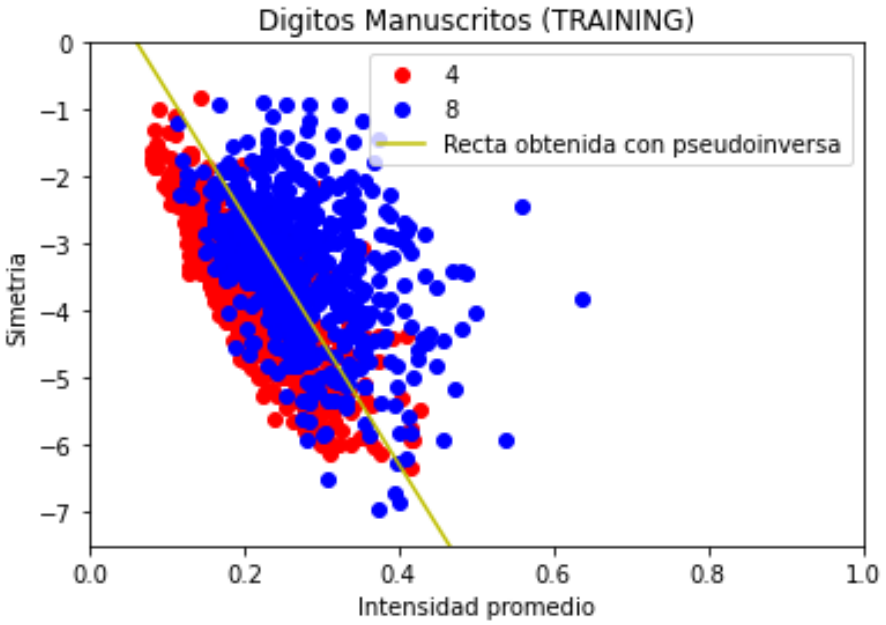
que esta vez acepta un vector de pesos y vemos si el signo de $x * w^T$ es igual al de su etiqueta.

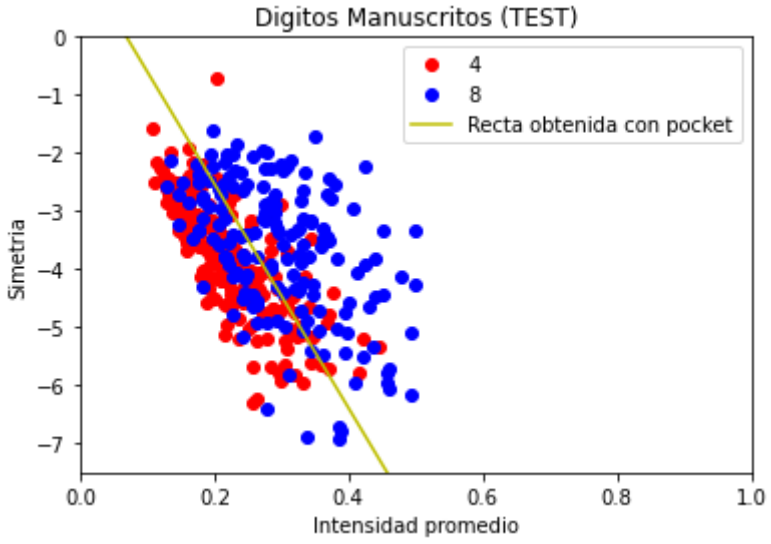
Una vez hemos definido todos estos algoritmos y errores vamos a hacer lo que nos pide el ejercicio. Estos son los datos de entrenamiento y test que tenemos:





Esto es lo que he obtenido al ejecutar el algoritmo de la pseudoinversa:

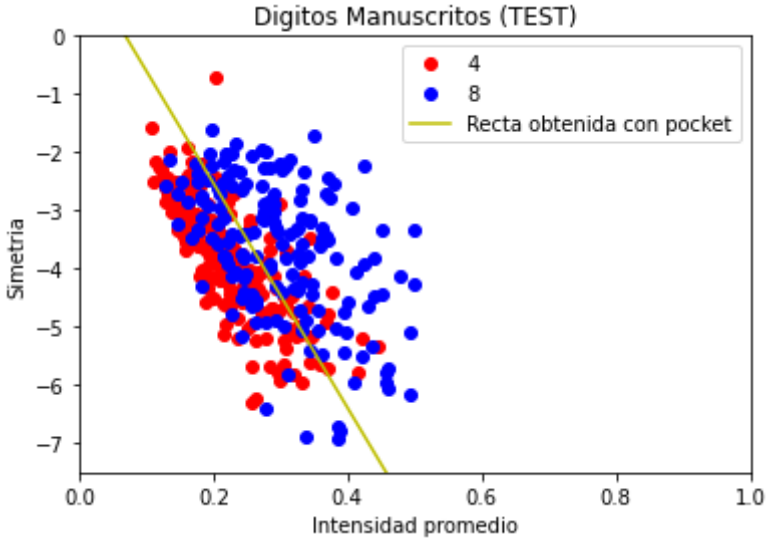




y estos son los errores obtenidos:

Error datos de training usando pseudoinversa: 0.6428532963367775
Error datos de test usando pseudoinversa: 0.7087148141159975

Como era de esperar los errores son bastante altos debido al ruido que tienen los datos. Ahora vamos a intentar mejorar este ajuste con el algoritmo Pocket, los resultados que obtenemos son los siguientes



y los errores que he obtenido son:

Error de pocket dentro de la muestra: 0.22529313232830822
Error de pocket fuera de la muestra: 0.2540983606557377

Podemos observar como el algoritmo pocket ha realizado una gran mejora en la solución que nos daba la pseudoinversa, reduciendo los errores muchísimo. Estos errores que aún tenemos son debidos al ruido que tiene la muestra.

Ahora el ejercicio nos pide calcular una cota sobre el verdadero valor de Eout. Una cota basada en Ein y otra basada en Etest con una tolerancia de 0.05.

Como hemos supuesto durante toda esta asignatura, las muestras con las que trabajamos para entrenar son muestras independientes e idénticamente distribuidas, por lo que sabemos que si reducimos el error dentro de la muestra, también lo hará

fuera de la muestra, por lo tanto, podemos calcular una cota para el error fuera de la muestra usando esto.

Esta cota se basa en que el error fuera de la muestra es el error dentro de la muestra (porque son i.i.d) más un pequeño error.

$$E_{out}(h) \leq E_{in} + error$$

Para calcular la cota sobre el verdadero valor de Eout usaremos esta fórmula aprendida en teoría:

$$E_{out}(h) \leq E_{in} + \sqrt{\frac{1}{2*N} * log \frac{2}{\delta}}$$

tras aplicar esta fórmula hemos obtenido los siguientes valores:

Cota de Eout usando Ein = 0.2645965277329354
Cota de Eout usando Etest = 0.2934017560603649.

Vemos como al calcular las cotas, la cota en Etest es mayor ya que el segundo sumando es el mismo para ambas cotas, sin embargo Etest es mayor.

Bibliografía

https://es.wikipedia.org/wiki/Distribuci%C3%B3n_normal
https://towardsdatascience.com/perceptron-algorithms-for-linear-classification-e1bb3dcc7602