UNIVERSIDAD DE GRANADA

Aprendizaje Automático

Práctica 2

Programación

Autor: Francisco Javier Bolívar Expósito

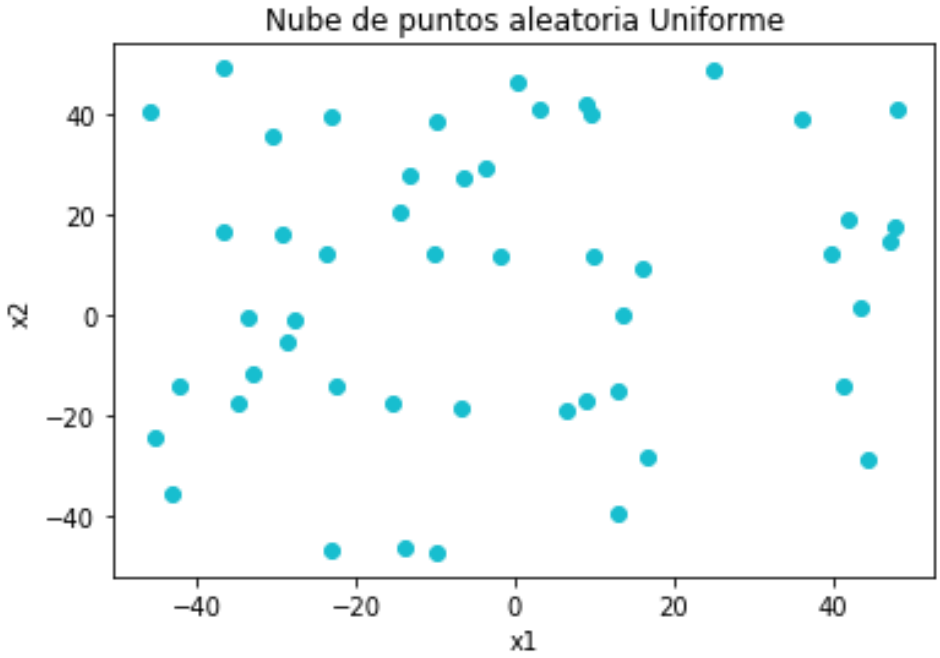
Curso: 2019-2020

Índice

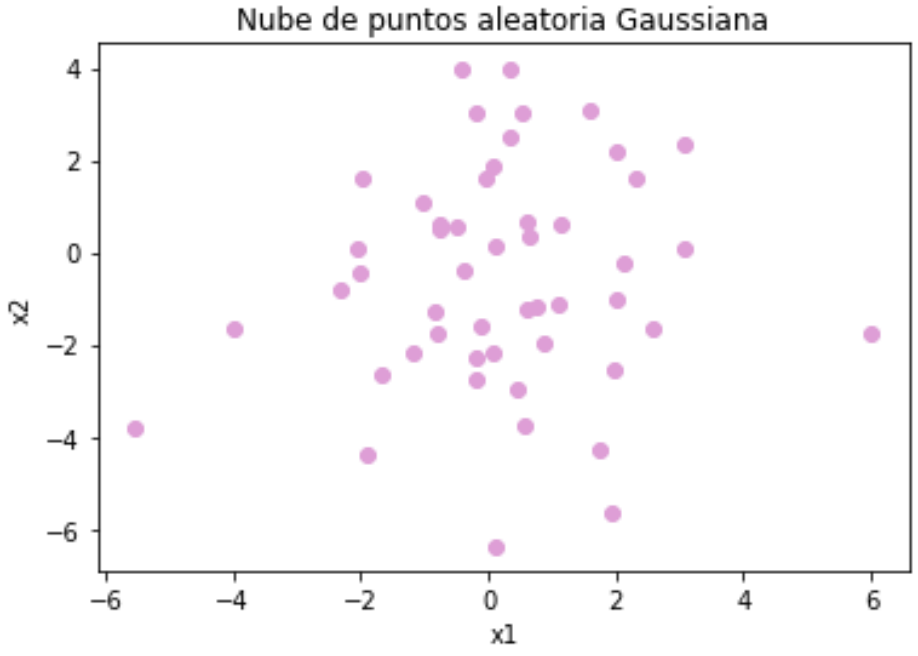
1. Ejercicio sobre la complejidad de H y el ruido.

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).

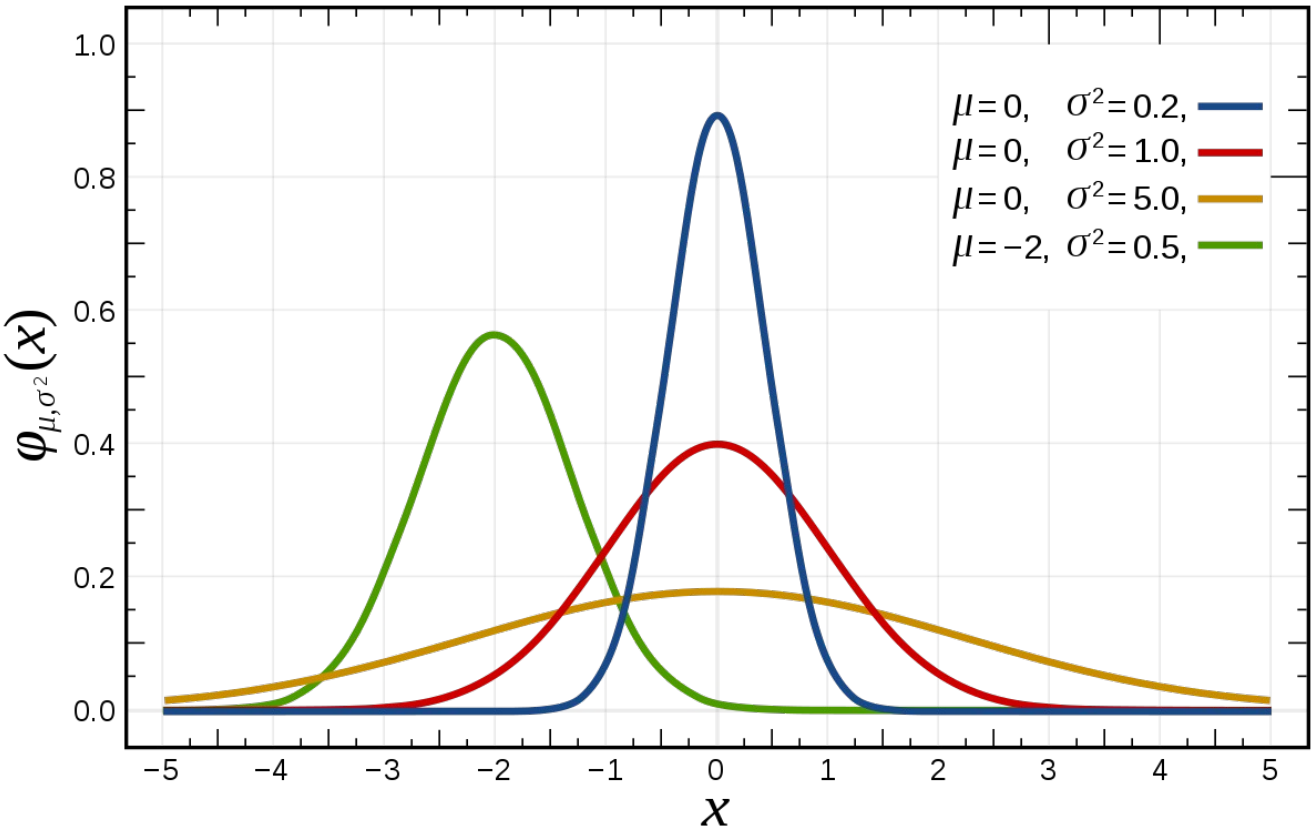


b) Considere N = 50, dim = 2 y sigma = [5, 7] con simula_gaus(N, dim, sigma).



Con la distribución uniforme podemos ver como los puntos quedan repartidos por todo el rango, sin una concentración especial, lo esperable en esta distribución.

Sin embargo, en la distribución normal observamos como los puntos tienden a concentrarse en el centro. Esto es lógico, ya que la función de densidad de una distribución normal o de Gauss tiene forma de campana y es simétrica, concentrándose la mayor densidad en un punto y disminuyendo esta conforme más nos alejamos de este.

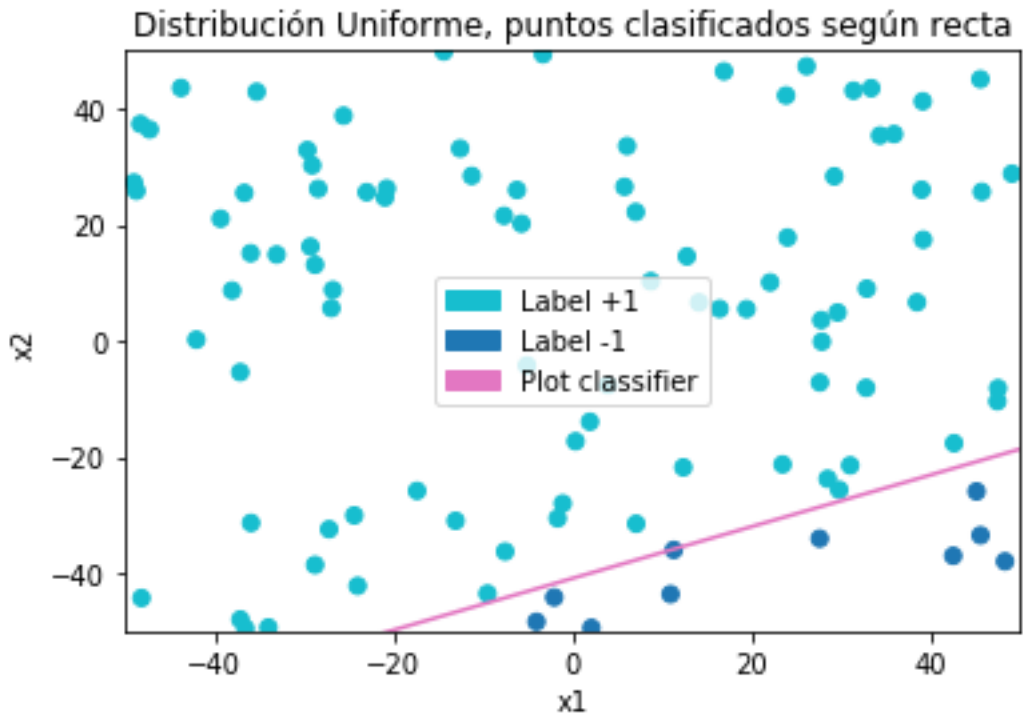


Función de densidad de probabilidad de distribuciones normales. Imagen obtenida de Wikimedia

2. Vamos a valorar la influencia del ruido en la selección de la complejidad de la clase de funciones.

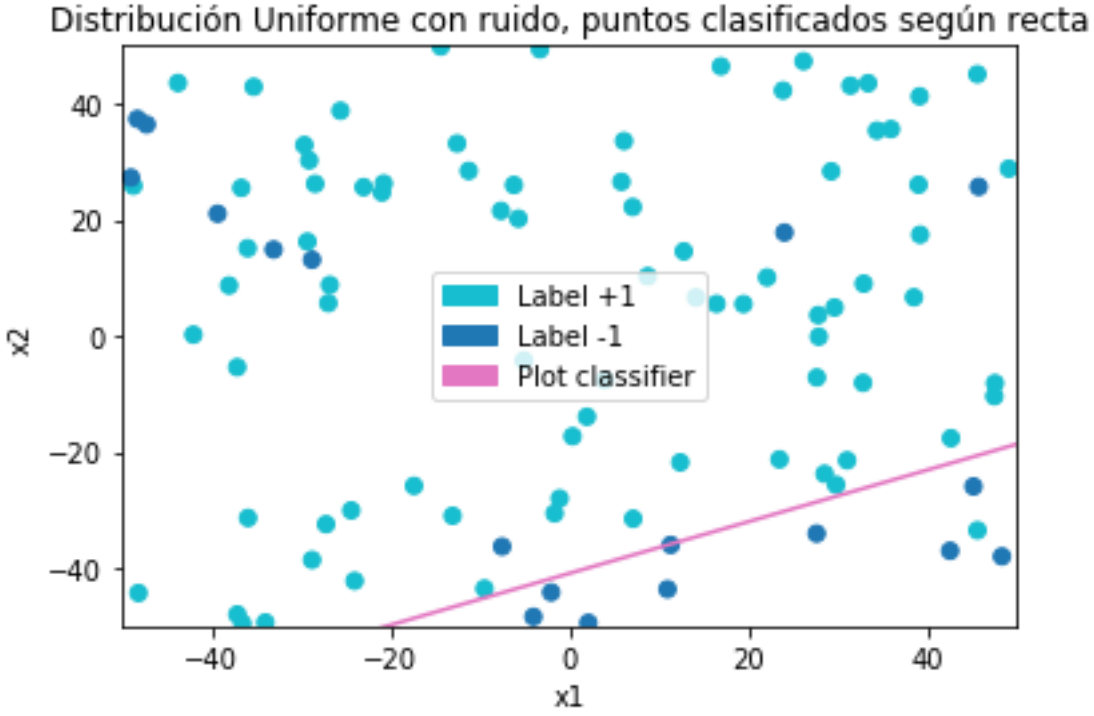
Con ayuda de la función simula_unif () generar una muestra de puntos 2D a los que vamos añadir una etiqueta usando el signo de la función f (x, y) = y −ax−b, es decir el signo de la distancia de cada punto a la recta simulada con simula_recta().

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



De esta forma obtenemos una clasificación perfecta de los puntos.

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)



Tras introducir un 10% de ruido para cada etiqueta podemos ver como se han modificado un mayor número de las etiquetas que tenían valor -1, ya que se modifican de manera proporcional a cada clase.

Ya no obtenemos una clasificación perfecta, sin embargo la recta original, que no está teniendo en cuenta el ruido, sigue siendo una buena frontera de decisión.

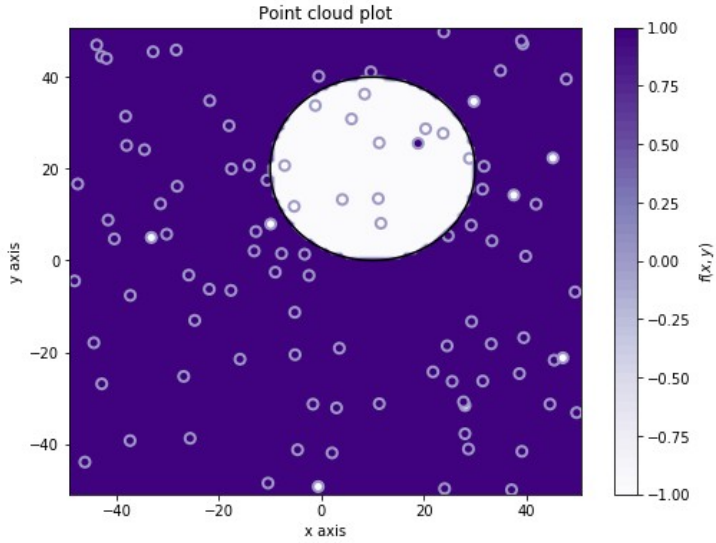
c) Supongamos ahora que las siguientes funciones definen la frontera de clasificación de los puntos de la muestra en lugar de una recta:

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

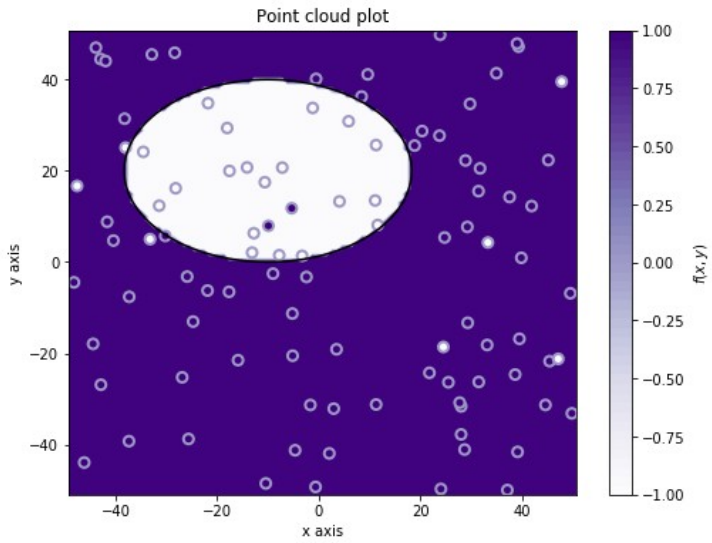
Visualizar el etiquetado generado en 2b junto con cada una de las gráficas de cada una de las funciones. Comparar las formas de las regiones positivas y negativas de estas nuevas funciones con las obtenidas en el caso de la recta ¿Son estas funciones más complejas mejores clasificadores que la función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

Se han vuelto a generar las etiquetas según estas nuevas fronteras de clasificación y se les ha añadido ruido.

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

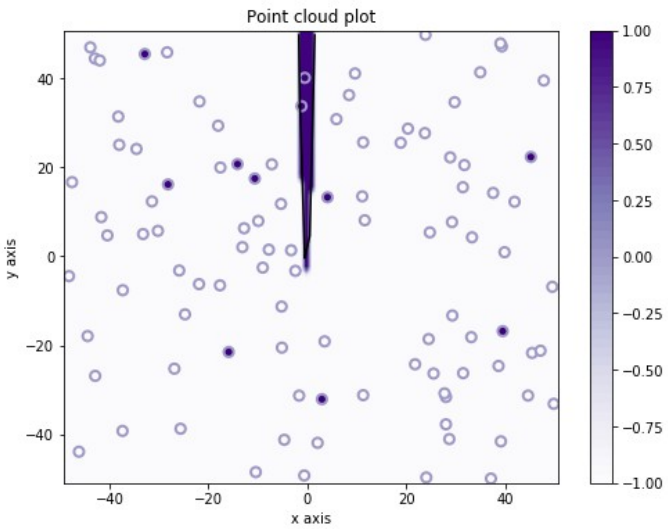


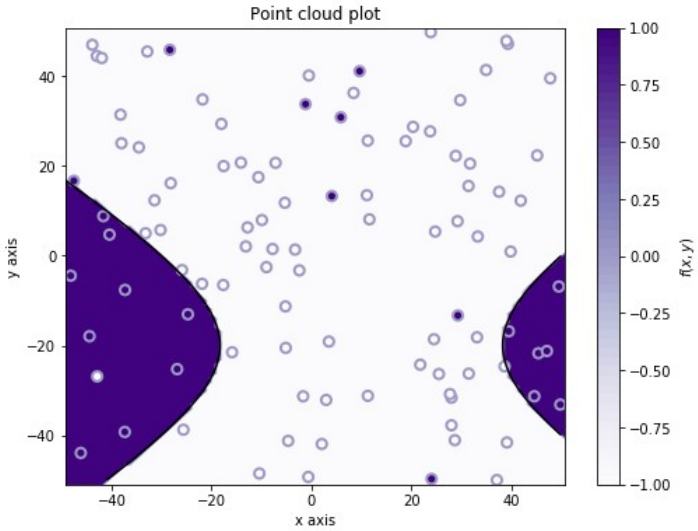
$f(x, y) = 0.5(x + 10)^2 + (y - 20)^2 - 400$



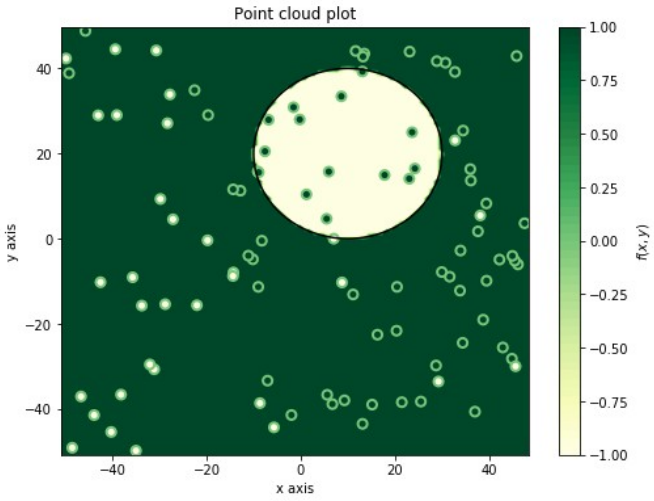
$f(x, y) = 0.5(x - 10)^2 + (y + 20)^2 - 400$

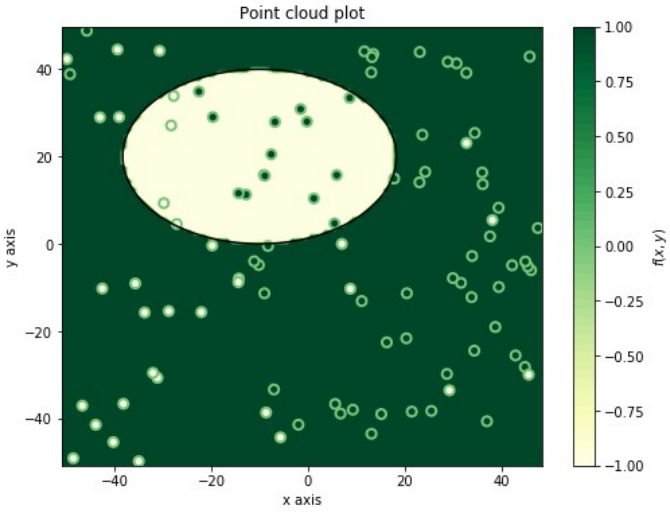
$f(x, y) = y - 20x^2 - 5x + 3$

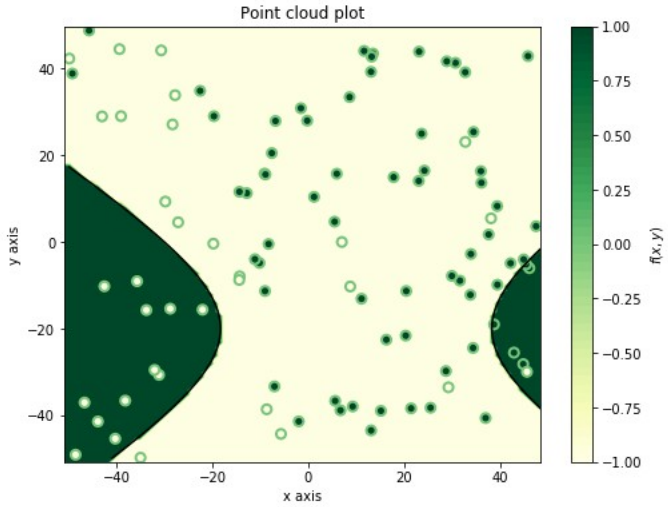


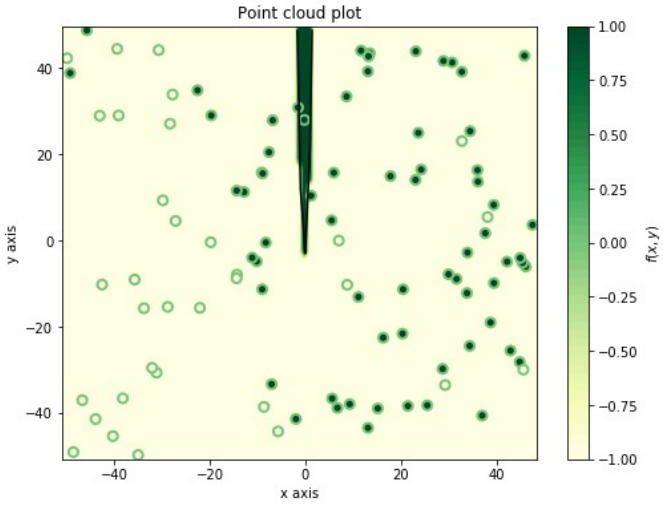


También se ha considerado interesante dibujar estas funciones sin re-calcular las etiquetas. (gráficas en el mismo orden)









Respondiendo a la pregunta de si las funciones más complejas son mejores clasificadores podemos sacar una conclusión con las gráficas generadas.

Si generamos las etiquetas basándonos en la frontera de decisión vemos que esa función separa muy bien a los datos.

Si intentamos usar las funciones para separar los datos generados con una recta como frontera de decisión, observamos que además de obtener muy malos resultados por no estar ajustada la función, aunque se ajustara las formas obtenidas no son las más adecuadas para clasificar los datos.

Por lo tanto dependiendo del problema, del conjunto de muestras, una función sera mejor o peor clasificadora, pero no por ser más compleja tiene que ser mejor o peor. Si los datos son linealmente separables es lógico separarlos con la función más simple que se adapta a ellos, una recta.

¿Que consecuencias tiene la modificación de etiquetas?

Podemos ver que aun con un 10% de ruido, ya sea con fronteras de decisión complejas o simples, el error introducido es el mismo. Las fronteras de decisión siguen siendo buenas con solo 10% de ruido, si entrenáramos un modelo teniendo en cuenta el ruido y una función lo suficientemente compleja podríamos reducir el error de entrada, pero esto no es precisamente bueno, ya que podríamos sobre-ajustar con respecto al ruido encontrando un caso de overfitting y obteniendo mayor error de salida.

Finalmente la conclusión que se extrae es simular a la anterior, una función más compleja no tiene porque ser mejor, entrenar el modelo con la función más simple que obtenga buenos resultados es una buena idea y es muy importante ver el error en una muestra de test con la que no se haya entrenado el modelo.

2. Modelos Lineales

Algoritmo Perceptron

Implementar la función ajusta_P LA(datos, label, max_iter, vini) que calcula el hiperplano solución a un problema de clasificación binaria usando el algoritmo PLA. La entrada datos es una matriz donde cada item con su etiqueta está representado por una fila de la matriz, label el vector de etiquetas (cada etiqueta es un valor +1 o −1), max_iter es el número máximo de iteraciones permitidas y vini el valor inicial del vector. La función devuelve los coeficientes del hiperplano.

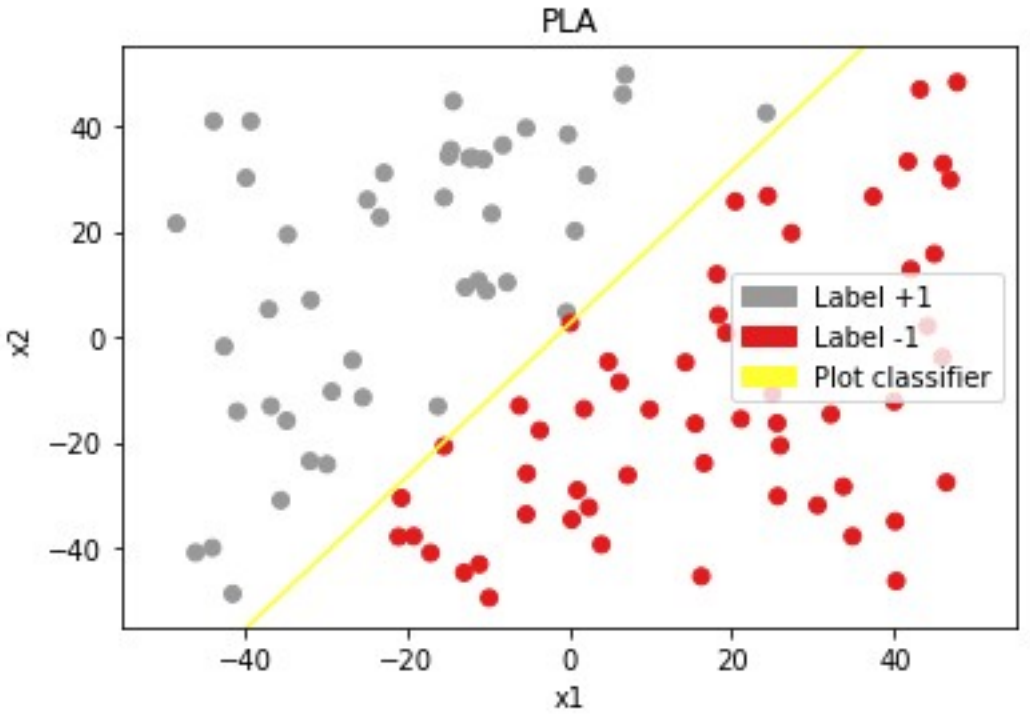
```python
# @brief Perceptron Learning Algorithm. Usado para clasificación
# @param data Características de la muestra de entrenamiento
# @param labels Etiquetas de la muestra de entrenamiento
# @param max_iter Iteraciones máximas
# @param w_ini Punto inicial
def ajusta_PLA(data, labels, max_iter, w_ini):

    iterations = 0
    altered = True

    # Hasta que pase una época sin ningún cambio
    while altered:
        iterations += 1
        altered = False

        # Permutamos los datos
        index = np.random.permutation(len(labels))
        data_rng = data[index]
        labels_rng = labels[index]

        # Recorremos todos los datos actualizando los pesos w por cada uno que
        # no esté correctamente clasificado
        for x, y in zip(data_rng, labels_rng):
            if signo(np.dot(w_ini, x)) != y:
                w_ini += y * x
                altered = True

        if iterations == max_iter:
            break

    return w_ini, iterations
```

Implementación realizada

En cada época permutamos el orden en el que recorremos las muestras, para no aprender en todas de la misma manera. Tras esto, actualizamos los coeficientes W del hiperplano en cada caso en el que la predicción del modelo actual no sea correcta.

Si tras una época completa no se han actualizado los pesos W el modelo ha convergido y se ha obtenido una solución sin error para al problema de clasificación binaria con los datos de entrenamiento, por lo que se detiene el algoritmo.

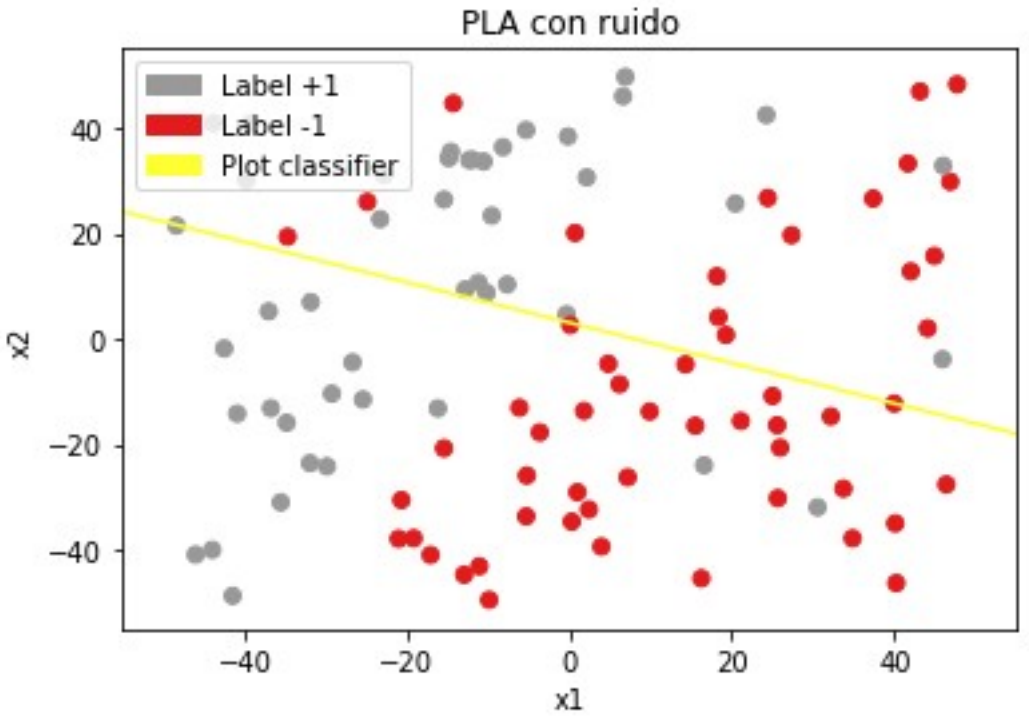
1) Ejecutar el algoritmo PLA con los datos simulados en los apartados 2a de la sección 1. Inicializar el algoritmo con: a) el vector cero y, b) con vectores de números aleatorios en [0, 1] (10 veces). Anotar el número medio de iteraciones necesarias en ambos para converger. Valorar el resultado relacionando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




```
PLA sin ruido en los datos
Con w_inicial = [0, 0, 0]
Iteraciones necesarias para converger: 205

Con w_inicial con valores aleatorios entre 0 y 1
Valor medio de iteraciones necesario para converger: 209.4
```

De media obtenemos un valor similar entre los valores aleatorios y el vector inicializado a 0. Este algoritmo no es tan dependiente del punto inicial, influye ya que según lo cerca que esté de una solución se llegará a esta en menos iteraciones y puede ser diferente pero si los datos son linealmente separables siempre llegará a converger independientemente del punto inicial en una solución sin error.

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.




```
PLA con ruido en los datos
Con w_inicial = [0, 0, 0]
Iteraciones necesarias para converger: 500

Con w_inicial con valores aleatorios entre 0 y 1
Valor medio de iteraciones necesario para converger: 500.0
```

Todas las ejecuciones usan 500 iteraciones, el número máximo de iteraciones permitido. Este se debe a que al introducir ruido en cada clase las datos ya no son linealmente separables. Por lo tanto el algoritmo PLA siempre va a encontrar una predicción errónea, va a modificar los pesos y no va a converger nunca ni llegar a la condición de parada (solo a la de iteraciones máximas).

Por esto obtenemos una solución muy mala y este algoritmo solo tiene sentido si sabemos que los datos son linealmente separables. En el bonus veremos una modificación del algoritmo que soluciona este problema.

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.

Consideremos d = 2 para que los datos sean visualizables, y sea X = [0, 2] × [0, 2] con probabilidad uniforme de elegir cada $x \in X$. Elegir una línea en el plano que pase por X como la frontera entre f (x) = 1 (donde y toma valores +1) y f (x) = 0 (donde y toma valores −1), para ello seleccionar dos puntos

aleatorios del plano y calcular la línea que pasa por ambos. Seleccionar N = 100 puntos aleatorios {xn } de X y evaluar las respuestas {yn} de todos ellos respecto de la frontera elegida.

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

```python
# Gradiente para la regresión logística
def logisticGradient(x, y, w):
    gradient = np.zeros((x.shape[1]))

    for sample, label in zip(x, y):
        gradient -= (label * sample) / (1 + np.exp(label * (sample @ w)))

    return (gradient / x.shape[0])

# @brief Regresión logística implementada con SGD
# @param data Característica de la muestra
# @param labels Etiquetas de la muestra
# @param lr Learning rate
# @param m Tamaño de los minibatches
# @param threshold Umbral para la condición de parada
# @param maxiter Iteraciones máximas
def sgdRL(data, labels, lr=0.01, m=1, threshold=0.01, maxiter=1000):
    # Inicializamos w a un vector de 0 y calculamos cuantos minibatches
    w = np.zeros((data.shape[1]))
    n_minibatch = int(len(labels)/m)

    # Barajamos las muestras
    index = np.random.permutation(len(labels))
    rng_data = data[index]
    rng_labels = labels[index]

    # Dividimos las muestras en minibatchs
    minibatch_data = np.array_split(rng_data, n_minibatch)
    minibatch_labels = np.array_split(rng_labels, n_minibatch)

    # Iteramos hasta llegar al límite de iteraciones o pasar del umbral
    # En cada iteración se hace un pase completo a los datos
    for i in range(maxiter):
        index = np.random.permutation(len(minibatch_data))

        # Guardamos el resultado de la época anterior
        last_w = np.copy(w)

        # Actualizamos w para cada minibatch
        for i in index:
            w = w - lr * logisticGradient(minibatch_data[i], minibatch_labels[i], w)

        # Si la norma de la diferencia con la anterior época es menor al umbral
        # paramos el algoritmo
        if (np.linalg.norm(last_w - w) < threshold):
            break

    return w
```

La implementación realizada se basa en el algoritmo SGD implementado en la práctica 1.

Inicializamos el vector de pesos a 0. En cada época permutamos el orden en el que recorremos las muestras, igual que en el PLA.

Actualizamos los pesos para cada minibatch (tras probar distintos tamaños, se ha decidido hacerlo por defecto a 1 ya que así actualiza más veces los pesos en cada época y converge en menos iteraciones) con un learning rate de 0.01 como se especifica en el ejercicio.

Si la norma de la diferencia entre el peso anterior y el actual (la distancia euclídea entre los pesos) es menor que el threshold (definido por defecto a 0.01 como especifica el ejercicio) detenemos el algoritmo.

El cambio fundamental es el cambio del gradiente, ya que ahora usamos el del sigmoide, la función sigmoide o función logística es apropiada para los problemas de regresión logística. Al estar acotada entre 0 y 1 podemos interpretar sus resultados como la probabilidad de que una muestra pertenezca a una clase, en la clasificación binaria por ejemplo si esta probabilidad es mayor a 0.5 podemos suponer que pertenece a una clase y si es menor a 0.5 a la contraria.

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).

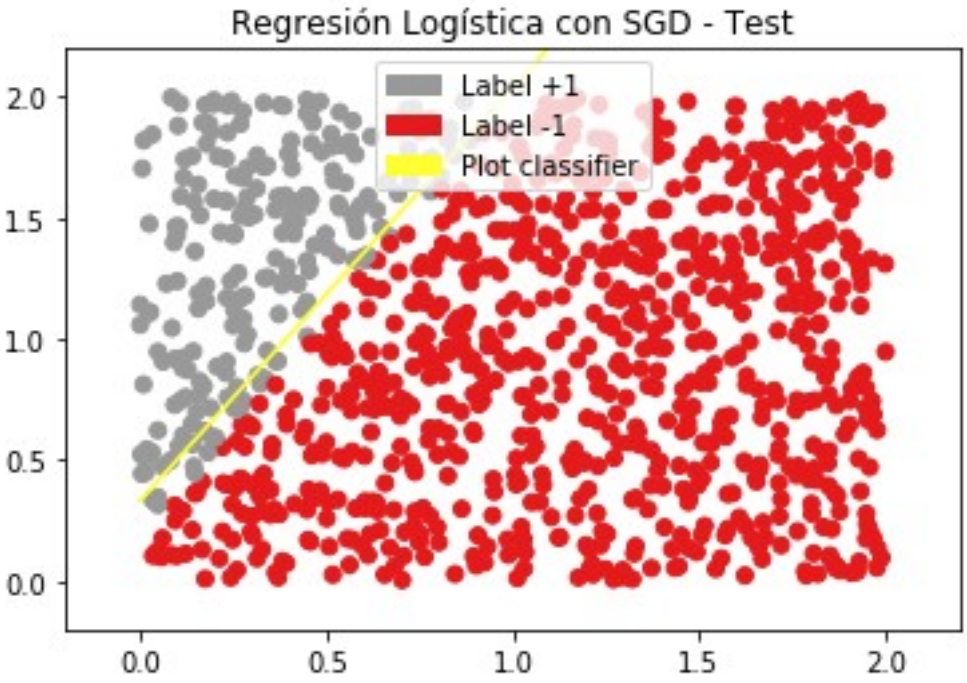
Simulamos 100 muestras de datos etiquetados con la función simula_unif y simula_recta y tras aplicar nuestro algoritmo obtenemos el siguiente resultado.

Se ha obtenido un error de 0.01. Aunque los datos sean linealmente separables no han encontrado la solución perfecta, sin embargo la solución obtenida con la condición de parada definida sigue siendo de mucha calidad.

El error no se calcula igual que en los problemas de regresión lineal, ya que aquí cada muestra solo puede estar bien o mal clasificada, no hay termino medio. El error se ha calculado con la siguiente función que calcula la proporción de predicciones erróneas.

```python
# Calcula la proporción de errores para un problema de clasificación binario
def missclassification(data, labels, w):
    errors = 0.0

    for x, y in zip(data, labels):
        if signo(np.dot(w, x)) != y:
            errors += 1.0

    return errors / len(labels)
```

Para estimar Eout se han generado 1000 muestras de la misma manera que las muestras del test.



Se ha obtenido un Eout de 0.028. Algo más grande al haber conseguido encontrar una solución para el test que no era exactamente la recta original, pero no aumenta de manera significativa y sigue clasificando correctamente gran parte de los datos. Podemos observar también que los errores en la clasificación se encuentran muy próximos a la frontera de clasificación.

3. Bonus. 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.

Los datos se han obtenido como se indicó en el ejercicio 3 del trabajo 1, colocando los ficheros en una carpeta datos dentro del directorio donde se desarrolla y ejecuta la práctica.

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

Podemos plantear el problema de clasificación binaria igual que en el resto de ejercicios, solo tenemos dos clases y queremos minimizar la cantidad de errores que tenga nuestro modelo al hacer predicciones.

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

Como modelo de regresión lineal se ha utilizado la pseudoinversa implementada en el trabajo 1.

El algoritmo PLA-Pocket se ha implementado usando como base el algoritmo PLA implementado en el ejercicio 2. Se muestra solo el código añadido.

Guardamos el W inicial y su error como el mejor w encontrado hasta el momento.

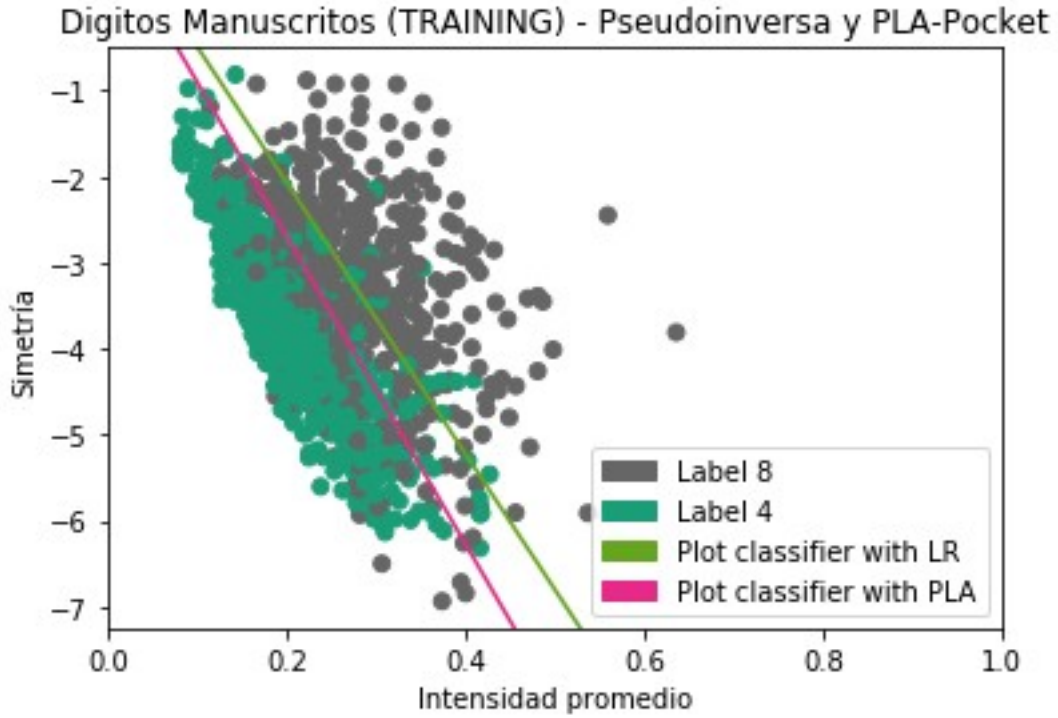
```
best_w = w.copy()
min_error = missclassification(data, labels, w)
```

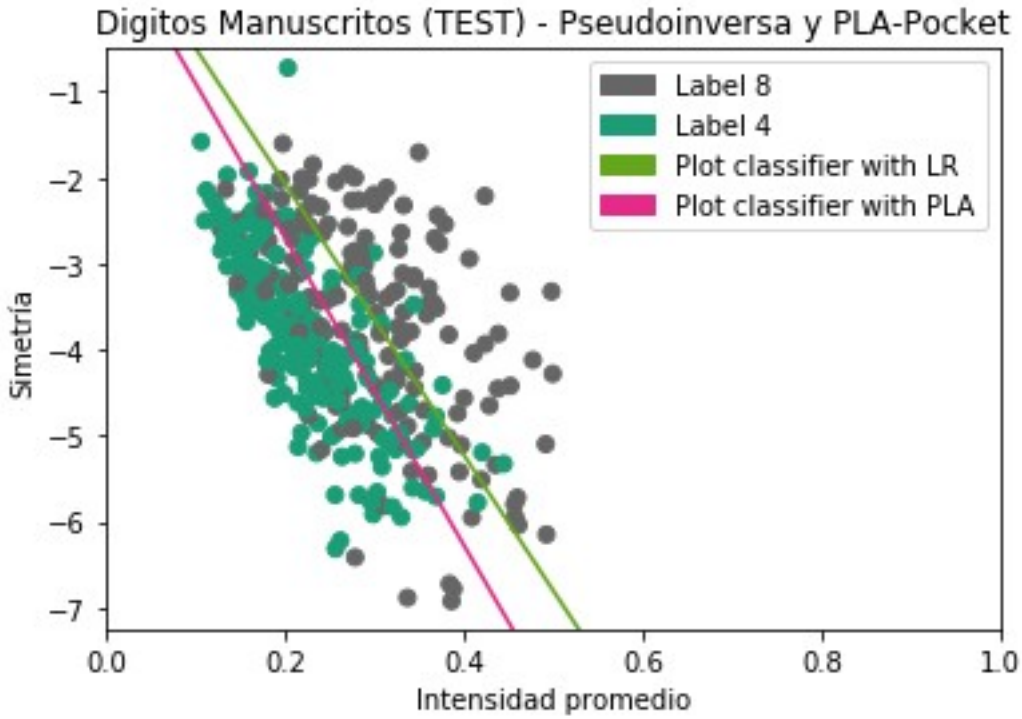
Tras cada época calculamos el error del W actual, y si es el mejor W hasta el momento lo guardamos. Finalmente no devolvemos el W de la última iteración, si no el W con menor error encontrado hasta el momento.

```
    # Calculamos el error del w calculado en la época actual
    w_error = missclassification(data, labels, w)

    # Si el error es menor que el del mejor w hasta el momento lo guardamos
    # como mejor w hasta el momento
    if (w_error < min_error):
        best_w = w.copy()
        min_error = w_error

    if iterations == max_iter:
        break

return best_w, iterations
```

Este cambio permite devolver buenas soluciones cuando no tenemos datos linealmente separables, el PLA normal nos devolvía una solución muy mala en este caso, vamos a ver ahora como PLA-Pocket se adapta mejor a estos casos.

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).

Errores calculados con la función missclassification de apartados anteriores, apta para un problema de clasificación binaria.

Ein con PseudoInversa: 0.22780569514237856

Etest con PseudoInversa: 0.25136612021857924

Ein con Pseudoinversa + Pocket: 0.21273031825795644

Etest con Pseudoinversa + Pocket: 0.25136612021857924

No se obtienen resultados mucho mejores a la Pseudoinversa, ya que la pseudoinversa ya nos proporciona la mejor solución (teniendo en cuenta un error de regresión). Pero podemos observar como el PLA-Pocket mantiene la solución con menor error.

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?

Cota basada en Ein obtenida.

$$Eout(h) \leq Ein(h) + \sqrt{\frac{1}{2N}\log\frac{2}{\delta}} \qquad Eout(h) \leq 0.2528712346006071$$

Cota basada en Etest.

$$Eout(h) \leq 0.2852050347488895$$

La cota basada en Etest puede ser más representativa al no ser los datos con los que se ha entrenado el modelo, además es más amplia por lo que es más posible que el error de salida real se encuentre en la cota.