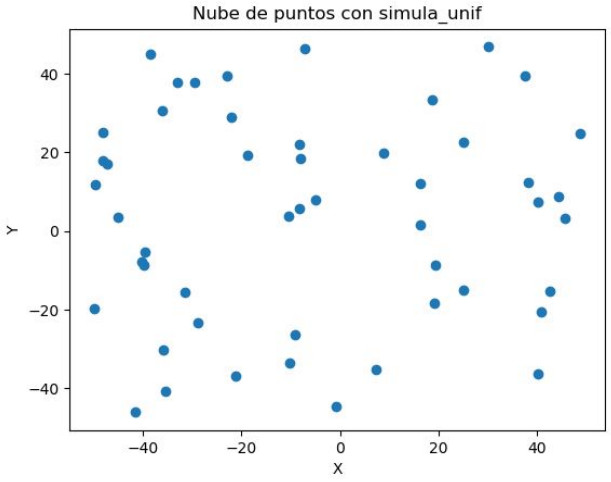
Universidad de Granada
Escuela Técnica Superior de Ingeniería Informática y de Telecomunicación
Grado en Ingeniería Informática
Departamento de Ciencias de la Computación e Inteligencia Artificial
Aprendizaje Automático
Curso 2019/2020
Grupo A2
Javier Rodríguez Rodríguez

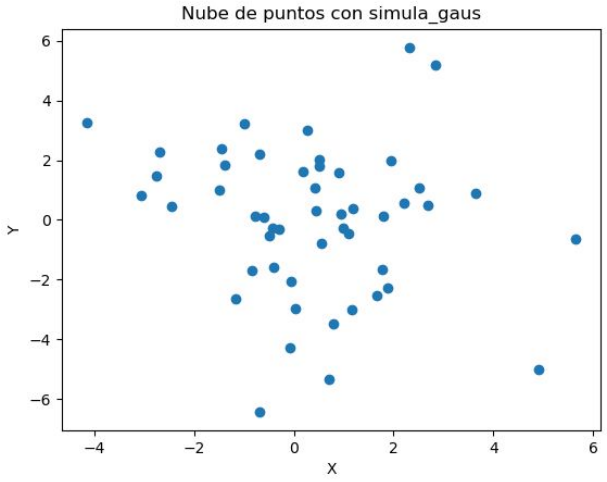
Trabajo 2: Programación

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

Para este apartado nos apoyamos en las funciones auxiliares provistas como parte de la sugerencia de código: simula_unif, simula_gaus y simula_recta.

1. Se pide la graficación de datos generados con simula_unif y simula_gaus, según parámetros específicos. Utilizando matplotlib se obtiene

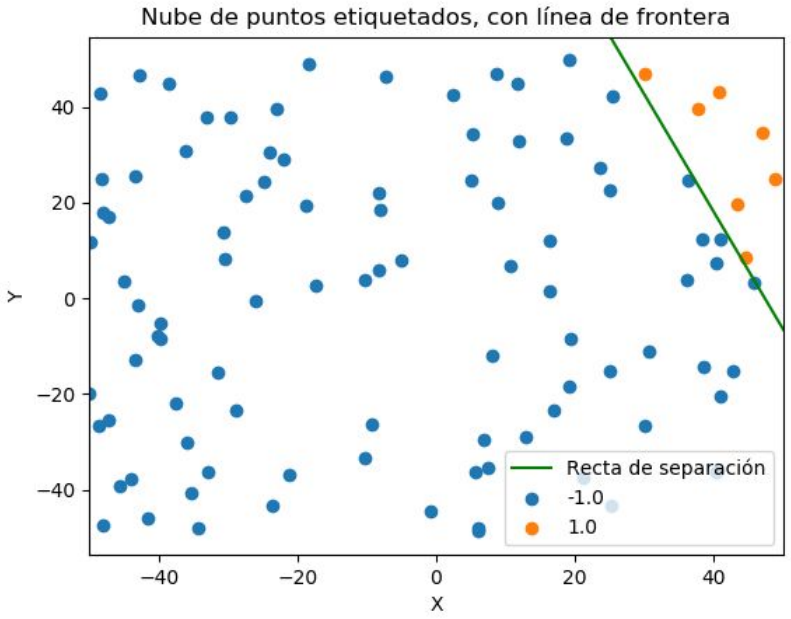




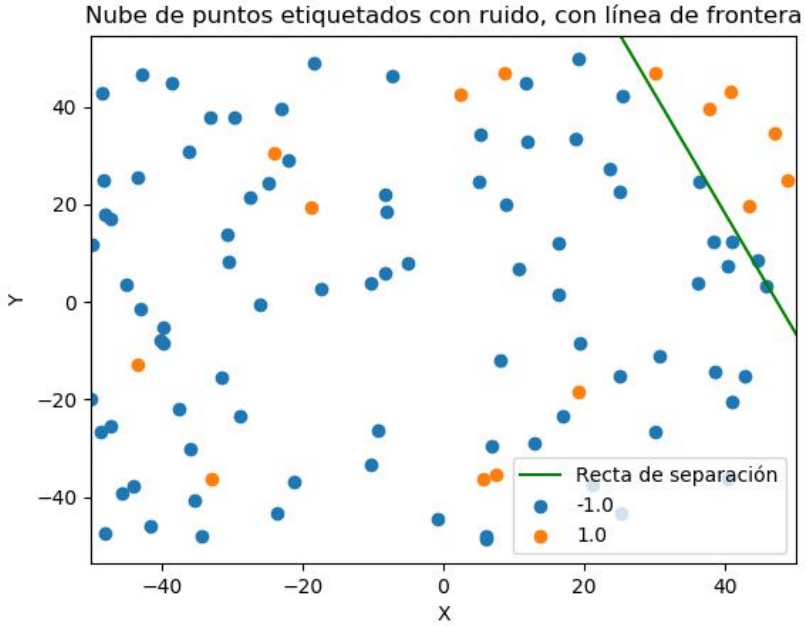
Estas gráficas evidencian la diferencia en la elección de valores según las distribuciones de probabilidad: una distribución uniforme selecciona de forma homogénea valores de todo el intervalo, la distribución normal o gaussiana da más peso a los valores centrales o promedio, aglomerando la mayoría de los puntos en el centro del intervalo.

2. Nuevamente utilizamos simula_unif para generar una nueva muestra de datos según parámetros especificados. Además, utilizamos simula_recta para obtener la pendiente y el término independiente asociados a la función de una recta que corta dos puntos aleatorios del espacio delimitado. Finalmente, generamos un vector de etiquetas donde por cada punto se asigna -1 o 1, según el resultado de la función signo de la destancia entre la recta calculada y cada uno de los puntos.

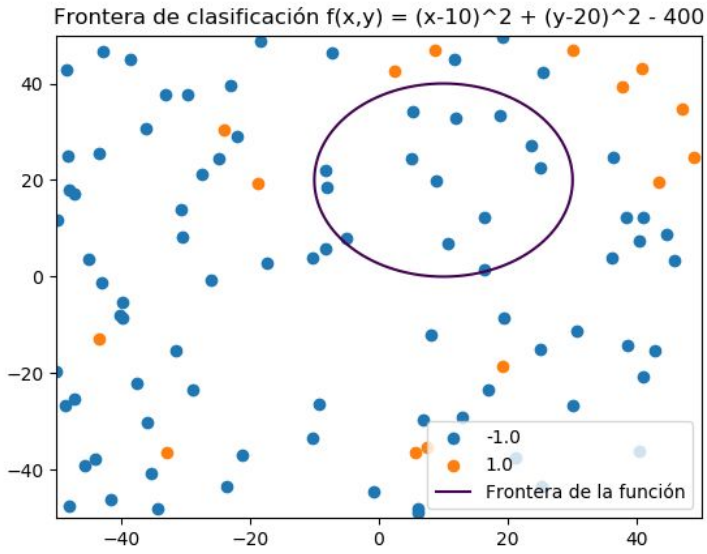
a) Graficamos mediante matplotlib los puntos etiquetados junto a la recta

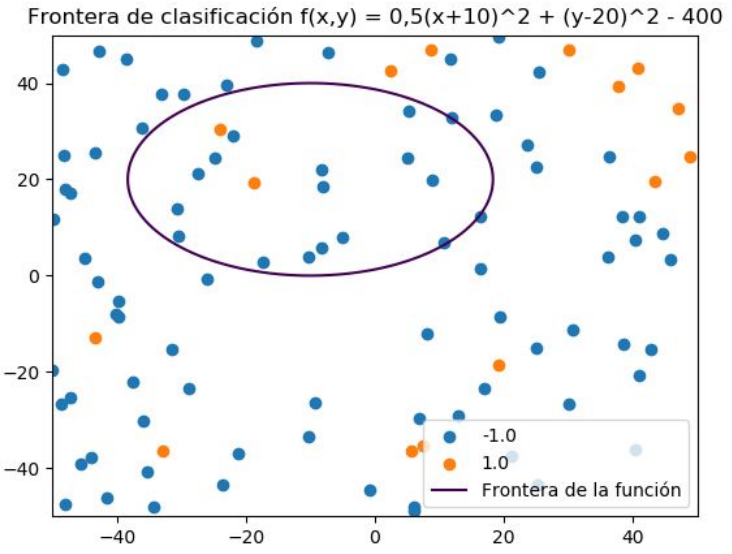


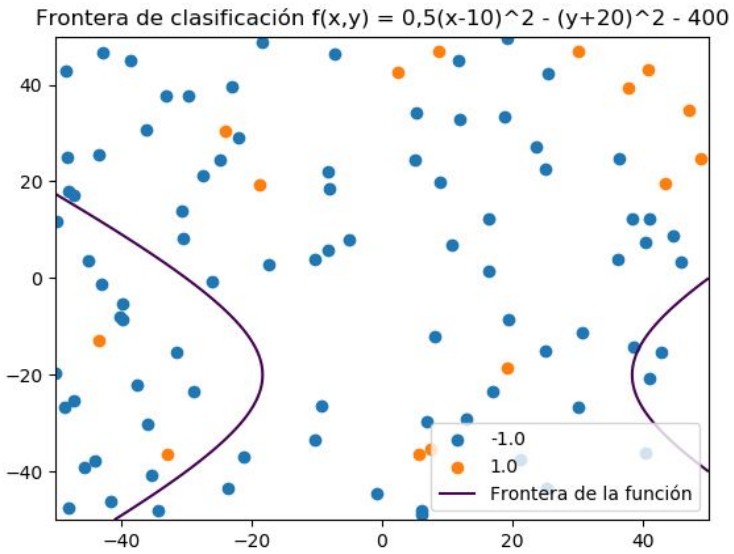
b) Modificamos el 10% de cada conjunto de etiquetas, aleatoriamente. Estas modificaciones se almacenan en una copia, de forma que no se contabilice erróneamente el 10% de un conjunto tras modificar el otro. La gráfica resultante es:

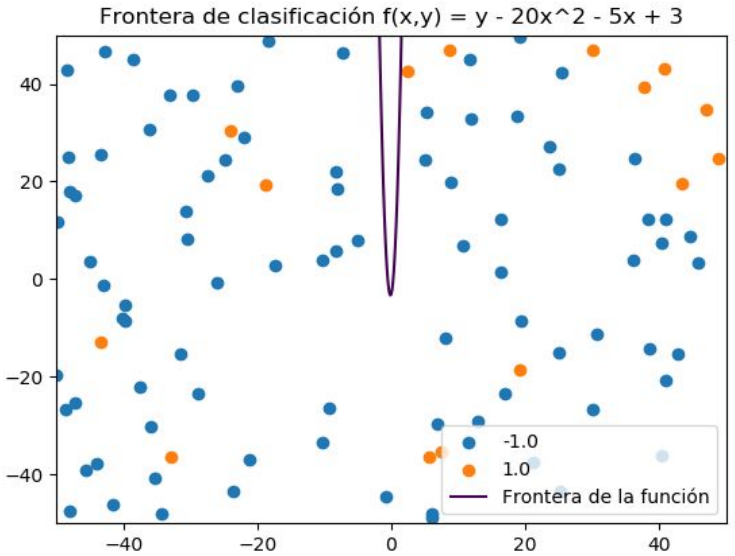


c) Consideramos cuatro nuevas funciones indicadas, no lineales, para las que analizamos la calidad de la clasificación que hacen de nuestros datos etiquetados.









Calculamos a través de la función de medición de calidad de un modelo de clasificación de scikit-learn, sklearn.metrics.accuracy_score, la calidad de cada una de estas funciones y la del modelo lineal original tras el ruido. Llamemos a estas cuatro funciones f1, f2, f3 y f4 respectivamente.

Función empleada	Modelo lineal	f1	f2	f3	f4
Precisión de la clasificación	90%	28%	28%	67%	85%

Como vemos, el modelo lineal resulta el de mayor calidad, conclusión razonable considerando que las etiquetas están inicialmente determinadas por esta función, tan solo se aplicó 10% de ruido explícitamente. Los otros 4 modelos resultan arbitrarios respecto a los datos y no tienen mayor poder de clasificación real. f1 y f2 poseen formas similares y coincidencialmente las lleva a similar precisión, pero no tiene por qué ser así. Las predicciones de f3 y f4, particularmente esta última, sugieren cierto nivel de precisión, pero la realidad es que, dada la gran mayoría de datos etiquetados negativamente en la muestra respecto a los positivos, basta con clasificar todos los datos como negativos (que es lo que hace f4) para obtener un resultado que aparente calidad. Ante una muestra distinta de datos, los resultados varían radicalmente. Por tanto, como las funciones f1 a f4 no tienen ninguna correlación con los datos, a pesar de que sean más complejas no representan en ningún punto un mejor clasificador que la recta inicial.

2. **Modelos lineales:**

a) Algoritmo Perceptron: la implementación del algoritmo es estándar, respetando los requerimientos en cuanto a los parámetros y sus nombres.

1) Replicamos los datos del apartado 1.2.a) mediante la ejecución de las mismas funciones y la misma semilla en el generador de números aleatorios. Procesamos estos datos 20 veces, 10 usando como entrada w inicial un vector de ceros, y 10 usando como entrada w inicial un vector randomizado de 0s y 1s. Tras los experimentos, obtenemos el número de iteraciones necesarias para converger en cada caso:

Como vemos, para el vector de ceros como entrada el número de iteraciones no cambia. Este resultado es coherente con el algoritmo: ningún dato se randomiza y el algoritmo es determinista. Todos los experimentos serán iguales.

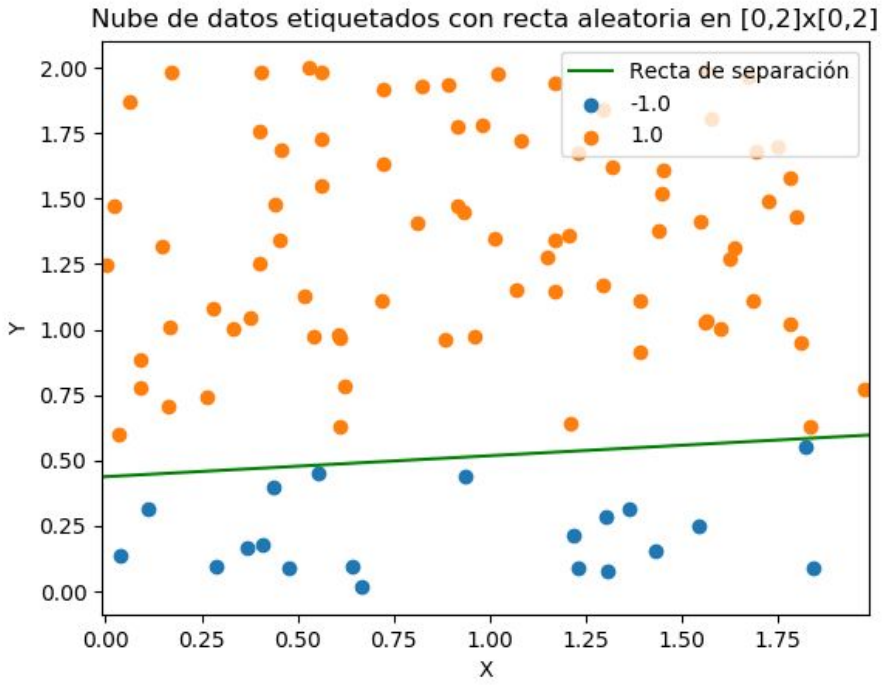
# Experimento	1	2	3	4	5	6	7	8	9	10	Promedio
Vector de 0s	444	444	444	444	444	444	444	444	444	444	444
Vector randomizado	889	1249	1587	1190	1548	1601	921	1595	1613	1140	1333.3

Para los vectores randomizados la diferencia es evidente: el número varía para cada ejecución, pues en cada una el vector inicial es distinto, y en todos los casos está muy por encima de las 444 iteraciones del vector de ceros. La razón principal para esto es la forma de la función que clasifica a los valores: los coeficientes del algoritmo son mucho más cercanos a 0, y en los valores aleatorios no tenemos la garantía de que la entrada sea cercana a estos valores bajos

2) Ahora consideramos los datos del apartado 1.2.b), es decir, los anteriores pero aplicando ruido al 10% de las etiquetas. En este caso, en los 20 experimentos alcanza el máximo de iteraciones (10000 en nuestro caso, pero no importa qué cota se establezca) antes de converger. La razón de esto

es que al añadir ruido es imposible establecer una recta que separe correctamente todas las muestras positivas de las negativas, la función real deja de ser lineal y por tanto el perceptron es incapaz de ajustar una función, independientemente de la entrada. Dado que en todos los casos llega al máximo, el promedio será este máximo.

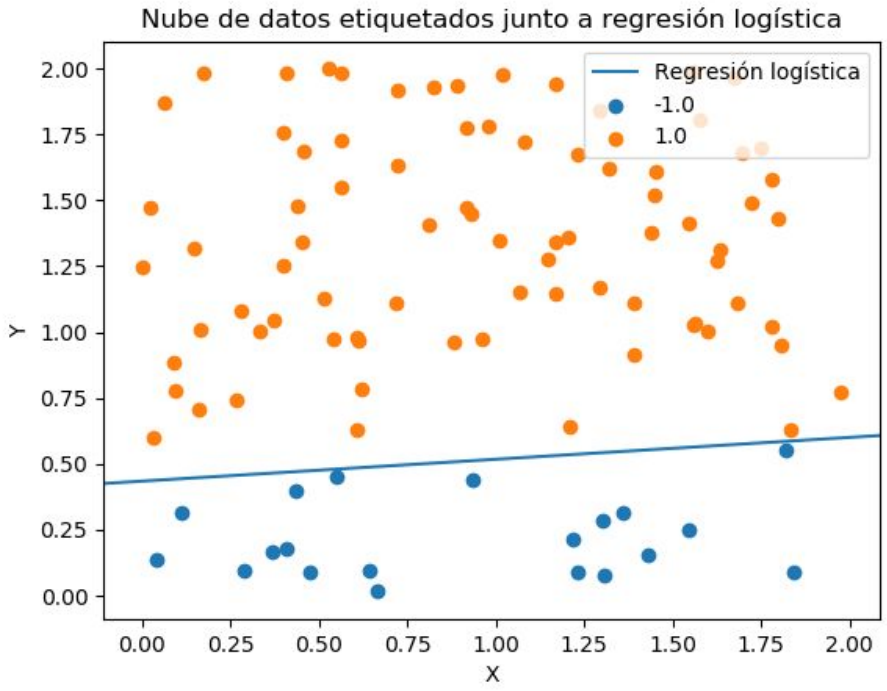
b) Regresión Logística: en primer lugar, generamos los datos según los parámetros establecidos, a través de simula_recta y simula_unif, y etiquetamos los datos al igual que en el apartado anterior. La gráfica resultante es la siguiente



1) Ahora implementamos la Regresión Logística (RL) a través de Gradiente Descendente Estocástico (SGD). Para esto, utilizamos la implementación de SGD de la práctica 1, así como su función auxiliar de graficación, y la modificamos para adecuarse a los nuevos requerimientos. En este caso consideramos una interpretación más "pura" de SGD, donde los minibatches son de tamaño 1. Esto nos permite evitar la construcción de estos, y en su lugar, barajar un vector de índices que establece el orden en el que se consideran los datos. Cada uno de estos datos es considerado, modifica el valor de w (inicialmente vector de 0, como se indica) según la función de costo indicada en el material de teoría, y establecemos como condición de parada el tamaño de paso.

2) Tras utilizar este algoritmo sobre los datos generados anteriormente,

resulta:

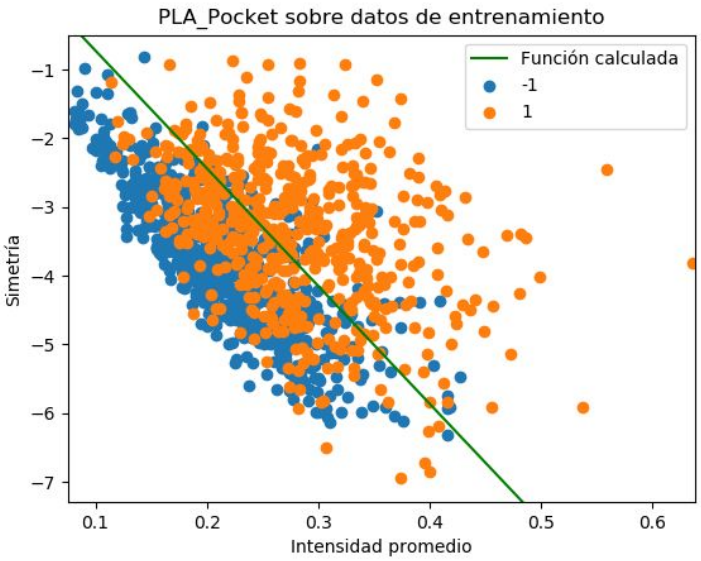


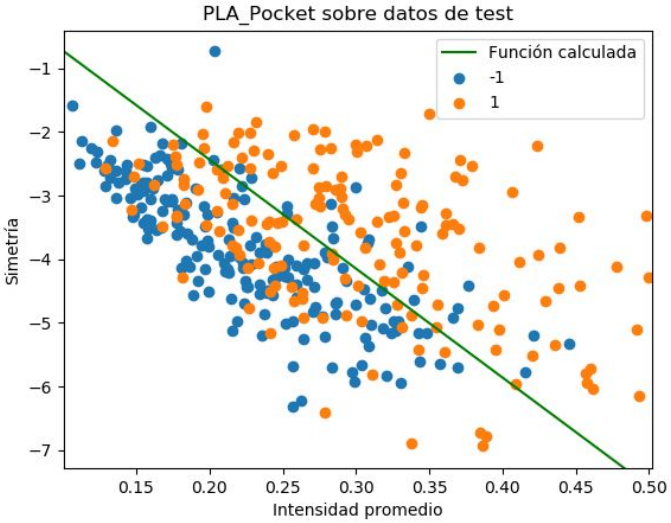
Como podemos ver, se ajusta perfectamente a los datos. Calculando E_{in} a través de la función definida en el material de teoría resulta
E_{in} = 0.06432975114287447.

Ahora generamos una nueva muestra de 2000 datos, la etiquetamos adecuadamente y calculamos E_{out} de forma análoga a E_{in} pero con esta nueva muestra, utilizando la regresión lineal ya calculada, obteniendo
E_{out} = 0.11655698259101326. Ambos resultados resultan sumamente favorables y prueban que la regresión logística aplicada genera una solución de calidad. La diferencia entre E_{in} respecto a E_{out} tiene sentido, pues la función es ajustada respecto a los datos de entrenamiento, por lo que tiene sentido que se adapte mejor a ellos.

3. **Bonus:** para este apartado extra se solicita considerar los datos de intensidad promedio y simetría de dígitos manuscritos de la práctica 1. Para leerlos y almacenarlos recurrimos a los mismos mecanismos en aquel momento pero cambiando los dígitos de interés, que en este caso son 4 y 8.
a) En caso de que sea 4 lo etiquetamos como -1, y si se trata de un 8, lo etiquetamos como 1. Con esta definición ya podemos plantear un problema de clasificación binaria respecto a los datos.
b) Decidimos usar como modelo de regresión lineal el algoritmo Perceptron, sobre el que aplicamos la mejora PLA-Pocket.

1) Graficamos los resultados sobre los conjuntos de entrenamiento y test con la función estimada (ajustada respecto a los datos de entrenamiento)





2) Si calculamos el error medio sobre los conjuntos de entrenamiento y test respectivamente resulta

E_{in} = 0.228643216080402

E_{Test} = 0.2459016393442623

3) Podemos determinar cotas máximas de E_{out} en función de los errores calculados en el paso anterior. Para ello utilizamos la ecuación $E_{out} \leq E + \sqrt{((1/2N)\log(2/\delta))}$, donde N es el tamaño de la muestra, E es E_{in} o E_{test} en función de qué conjunto, y por ende, qué error estemos utilizando, y δ corresponde a la tolerancia, que en este caso es de 0,05.

Con estos valores resulta

$E_{out/in}$ = 0.2679466114850292

$E_{out/test}$ = 0.3168907427768778

Como vemos, la cota de E_{out} en función de E_{in} es mejor. Si bien el hecho de que E_{in} es marginalmente mejro que E_{test}, pues la función lineal calculada mediante PLA-Pocket se ajustó con los datos de entrenamiento, es igual o más importante la diferencia en el tamaño de las muestras, siendo el conjunto de entrenamiento notablemente más grande (1194 puntos) que el de test (366 puntos), probando que a mayor tamaño de muestra se minimiza el E_{out} máximo