Aprendizaje Automático - Segunda Práctica
Complejidad de H y el ruido
Modelos Lineales

Sergio Quijano Rey - 72103503k
4º Doble Grado Ingeniería Informática y Matemáticas
sergioquijano@correo.ugr.es

30 de abril de 2021

Índice

Índice de figuras

Referencias

[1] "Lecture 4 - pac." https://www.cs.ubc.ca/~murphyk/Teaching/CS340-Fall06/lectures/L4pac.pdf. (Accessed on 27/04/2021).

[2] "machine learning - with regards to vc-dimension, why can you shatter 3 points with circles but not 4 points? - data science stack exchange." https://datascience.stackexchange.com/questions/21693/with-regards-to-vc-dimension-why-can-you-shatter-3-points-with-circles- (Accessed on 27/04/2021).

1. Ejercicio 1 - Sobre la complejidad de H el ruido

1.1. Observaciones iniciales

Para este primer ejercicio, hacemos uso de tres funciones que se nos dan en el fichero `template_trabajo2.py`:

- `simula_unif`: genera una lista de N vectores aleatorios de una dimensión dada. La distribución aleatoria que siguen es una distribución uniforme en un intervalo dado
- `simula_gauss`: genera una lista de N vectores aleatorios de dimensión dada. La distribución aleatoria es una normal de media cero y varianza dada
- `simula_recta`: genera una recta aleatoria que pasa a través de un intervalo 2-dimensional dado

Además, al principio de cada función asociada a los ejercicios (`ejercicio1()`, `ejercicio2()`, `ejercicio_bonus()`) establecemos una semilla aleatoria fija con la orden `np.random.seed(123456789)`. Así, los resultados que obtenemos serán reproducibles. Aunque por estar usando probablemente distintos configuraciones de sistemas operativos, hardware, versión de `python`, ..., los resultados obtenidos por los profesores de prácticas no serán exactamente los mismos cuando exista un factor aleatorio.

1.2. Apartado 1

En este apartado se pide que dibujemos las nubes de puntos simuladas con las dos primeras funciones dadas por los profesores. Para ello empleamos los siguientes parámetros:

- $N = 50$
- $dim = 2$
- $rango = [-50, 50]$

- $\sigma = [5, 7]$ donde σ indica la varianza (por tanto sería más adecuada la notación σ^2) en el eje x e y

Lanzando el código obtenemos las dos siguientes nubes de puntos:

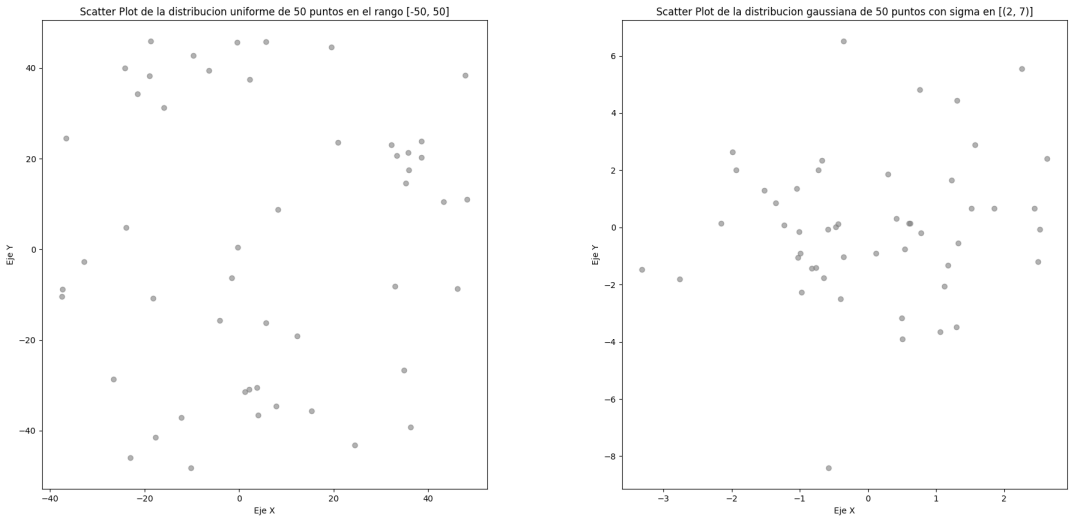


Figura 1: Nubes de puntos de las dos distribuciones

1.3. Apartado 2

Vamos a valorar la influencia del ruido en la selección de la complejidad de la clase de funciones. Para ello generamos una muestra de puntos bidimensionales. En concreto, generamos 100 puntos en el intervalo $[-50, 50]$ con una distribución aleatoria uniforme.

Una vez que generamos esta muestra de datos, los etiquetamos usando una recta generada aleatoriamente, usando la función dada por los profesores `simula_recta`. Con la recta $f(x) = ax + b$, etiquetamos los datos con la función de etiquetado $(x, y) \rightarrow sign(y - ax - b)$. Esta función toma un punto x, y, calcula su distancia a la recta f (que se corresponde con $y - f(x)$) y como etiqueta asigna el signo de esta distancia. Es decir, estamos mirando

si un punto se queda por encima de la recta (etiqueta $+1$) o por debajo (etiqueta -1).

Observación importante: en el subapartado c), hemos considerado una interpretación alternativa del ejercicio, pues parece tener más interés académico a la hora de analizar las propiedades de las funciones que se nos dan para clasificar. Por tanto, desarrollamos una sección 1.3.3, donde realizamos la tarea especificada, y una sección 1.3.4 donde realizamos el experimento alternativo. Ambos procesos nos parecen pobres a la hora de extraer información sobre dichas funciones (como se comentará en el análisis de cada uno de los subapartados), pero la interpretación alternativa parece dar algo más de juego a la hora de realizar los análisis.

1.3.1. Subapartado a)

El enunciado de este apartado especifica que se vuelva a generar la muestra de datos, como se muestra anteriormente, o al menos así lo hemos interpretado. Por tanto, se puede ver que la nube de datos no es la misma que la nube de datos uniforme del Apartado 1.

La gráfica de los datos etiquetados, junto a la recta que se usa para etiquetar, es la siguiente:

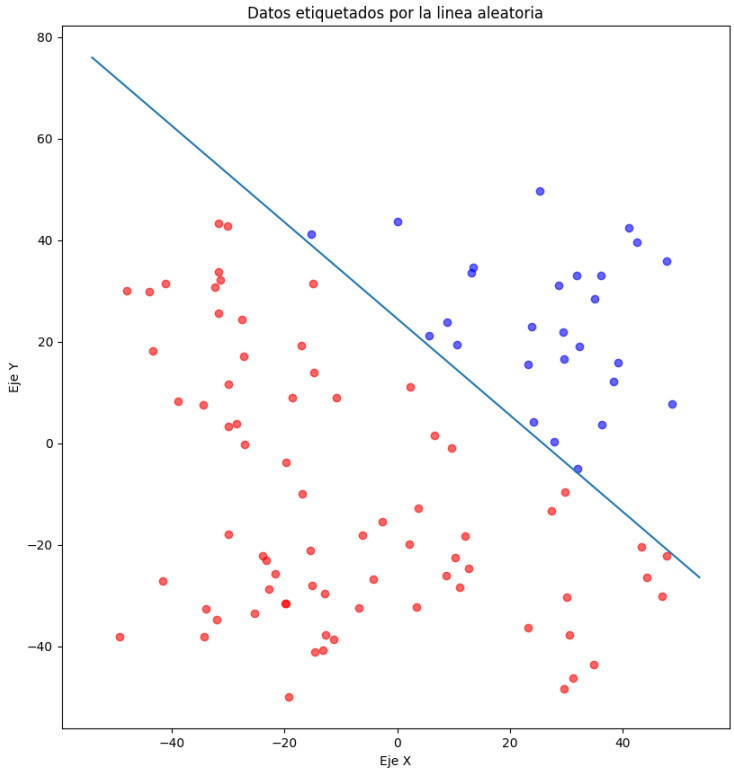


Figura 2: Datos etiquetados junto a la recta de clasificación

La gráfica deja claro que los puntos han sido correctamente clasificados a partir de la recta generada aleatoriamente.

1.3.2. Subapartado b)

Modificamos el 10 % de las etiquetas positivas y un 10 % de las etiquetas negativas, y volvemos a mostrar la gráfica.

Para realizar esto de forma eficiente, lo que hacemos es tomar dos vectores, uno con las posiciones de etiquetas positivas y otro con las posiciones

de las etiquetas negativas. Calculamos el número de etiquetas de cada tipo a cambiar (N_1, N_2), a partir de un porcentaje arbitrario (para nuestro caso concreto, usamos 0,1). Hacemos un *shuffle* de los dos vectores de posiciones y cambiamos el etiquetado de las N_1 y N_2 primeras posiciones de los vectores remezclados. Con ello modificamos un porcentaje dado de las etiquetas de forma aleatoria. Todo esto se puede ver en la función `change_labels`.

Notar que podemos tener clases sin puntos. Es decir, que no etiquetamos ningún dato con o bien $+1$ o bien -1. En el caso de la recta, es muy improbable que esto pase. Pero en el siguiente subapartado, con ciertas funciones es muy probable que pase.

El gráfico de clasificación tras esta modificación aleatoria es:

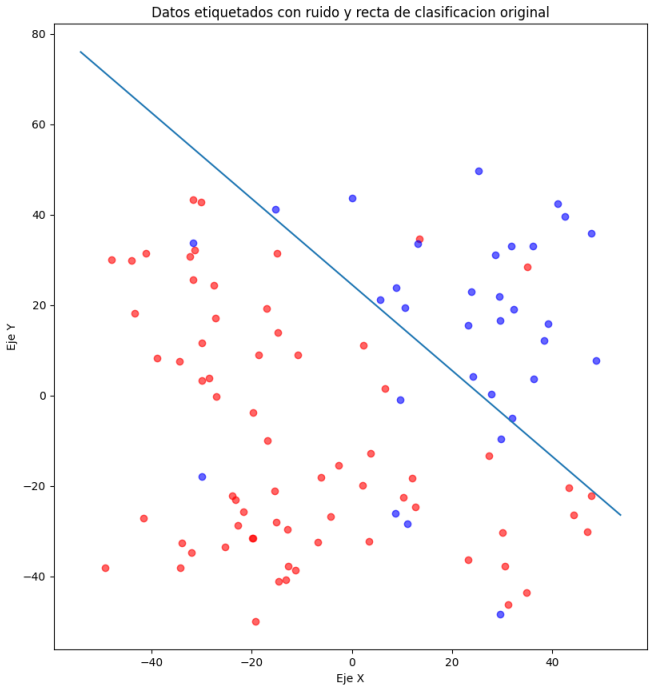


Figura 3: Datos etiquetados con ruido, y recta original de clasificación

Se ve con claridad cómo esta operación introduce ruido sobre nuestras etiquetas. Aproximadamente, tendremos un error entorno al 10 % por la forma en la que hemos modificado las etiquetas (dependiendo del ratio positivos / negativos, el error porcentual puede descender hasta el 9 %).

1.3.3. Subapartado c)

En este apartado, debemos considerar las siguientes funciones que definen una frontera de clasificación:

- $f_0(x, y) := (x - 10)^2 + (y - 20)^2 - 400$
- $f_1(x, y) := \frac{1}{2}(x + 10)^2 + (y - 20)^2 - 400$
- $f_2(x, y) := \frac{1}{2}(x - 10)^2 - (y + 20)^2 - 400$
- $f_3(x, y) := y - 20x^2 - 5x + 3$

Estas funciones no van a modificar el etiquetado que ya teníamos (el generado a partir de una recta y la introducción sintética de ruido). Simplemente mostramos las regiones de clasificación de estas nuevas funciones y los puntos con el clasificado original, buscando así realizar un análisis sobre el uso de estos clasificadores más complejos.

Podríamos haber intentado mostrar las líneas de frontera de etiquetado, como hemos hecho con la recta. Pero para ello, tenemos que calcular un despeje de $f(x, y)$, o bien de x o bien de y, pues $f(x, y)$ viene dada de forma implícita. No todas las funciones tienen un despeje global, por lo que habría que emplear el teorema de la función implícita para realizar despejes locales, y después concatenar las gráficas obtenidas con los despejes locales. Hemos considerado que esto entorpecía demasiado el código. Así que la solución encontrada ha sido usar la función `contourf` de la librería `matplotlib.pyplot`, con la que pintaremos las regiones en las que clasificamos positivamente y las regiones en las que clasificamos negativamente.

Una vez hechos estos comentarios, mostramos las gráficas obtenidas como se nos pedía:

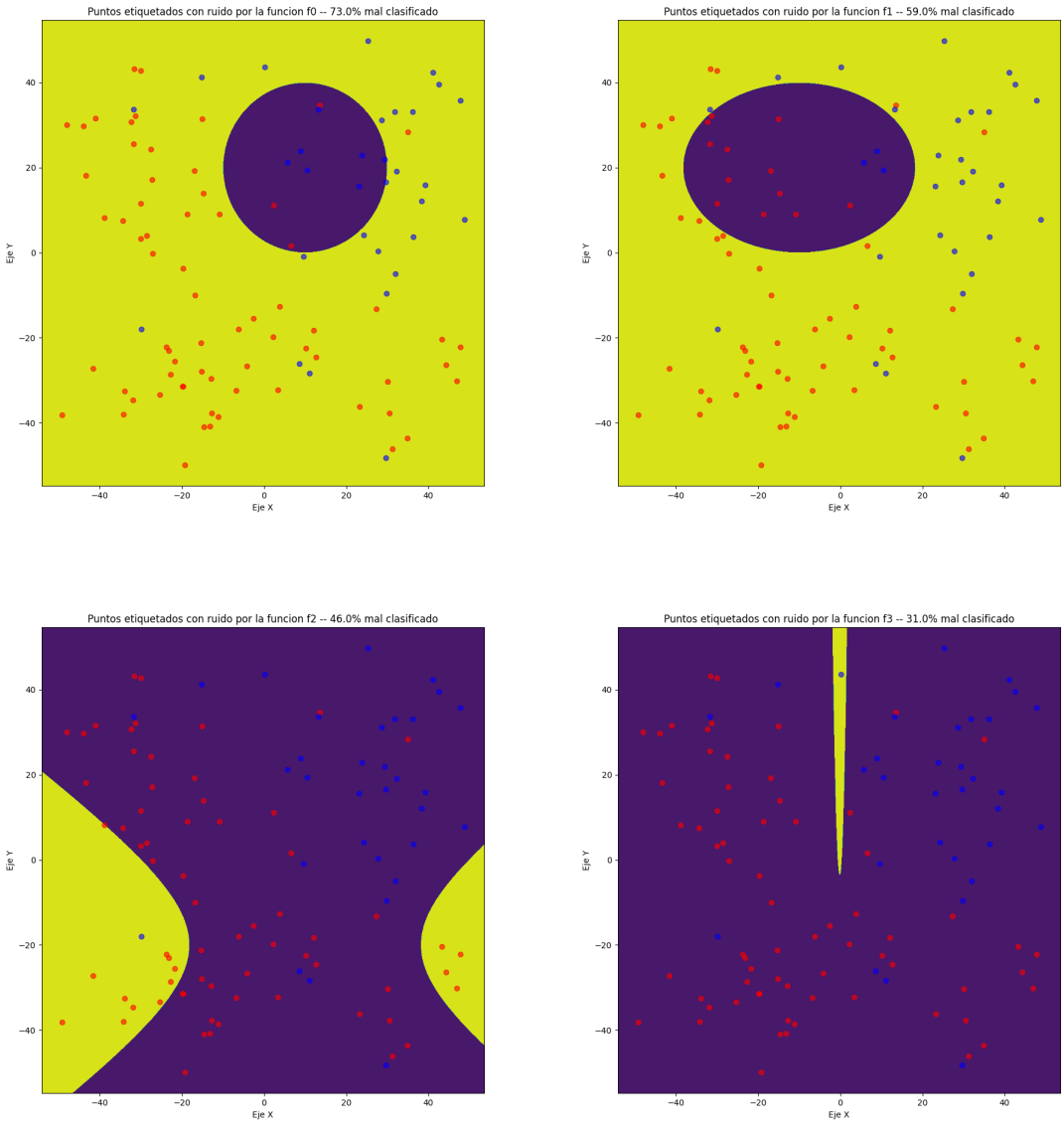


Figura 4: Etiquetado original sobre las nuevas funciones de etiquetado

Para comenzar, todas las funciones que estamos mostrando tienen un error porcentual mucho más alto que la recta que hemos usado para clasi-

ficar, estando los cuatro errores en un rango de [31,0 %, 73,0 %]. Por tanto, podemos ver que ciertas funciones tienen un error porcentual mucho más bajo que otras. En la interpretación alternativa realizamos una discusión sobre la *Dimensión Vapnik-Chervonenkis*, concluyendo que tienen mismo valor de d_{VC}, por lo que estas diferencias de error no pueden ser explicadas por una diferencia de complejidad de las funciones.

Como se ha comentado con el profesor de prácticas, a partir de este ejercicio se busca concluir que funciones más complejas no implican que realicemos un mejor ajuste de los datos. Y aunque esto se puede concluir por otros métodos más acertados, llegar a esas conclusiones a partir de este escenario parece algo falaz.

En primer lugar, no hemos realizado ningún tipo de proceso de aprendizaje. Los datos de la muestra han sido etiquetados por una recta que se ha generado aleatoriamente, mientras que las cuatro funciones de etiquetado $f_0, \ldots, f_3$, se han dado prefijadas. De hecho, a la hora de analizar la adecuación de las cuatro nuevas funciones, podemos considerar que la recta es prefijada y $f_0, \ldots f_3$ aleatorias, pues estas no tienen información alguna de cómo etiquetamos la muestra de datos.

Una correcta discusión se podría hacer ajustando ciertos parámetros de las cuatro funciones al etiquetado generado a partir de la recta aleatoria y la introducción de ruido. O por simplicidad, se podrían haber dado la recta de clasificación y las cuatro funciones $f_0, \ldots, f_3$ ya ajustadas a este etiquetado, ahorrando el proceso de ajuste pero permitiendo una correcta discusión sobre las funciones de clasificación.

Por tanto, consideramos el experimento muy pobre a la hora de concluir que no necesariamente una función más compleja se ajustará mejor a una muestra etiquetada dada.

Un estudio de propiedades más abstractas sobre las funciones puede realizarse a la hora de discutir en mayor profundidad que se nos plantea. Esto lo hacemos en el siguiente apartado. Se podría hacer también en este apartado, pero a la hora de razonar sin fijarnos tanto en el mal etiquetado de $f_0, \ldots, f_3$, parece más adecuado realizar dicha discusión en el siguiente apartado. De nuevo, todo lo dicho en el siguiente apartado (discusión sobre d_{VC}, longitud de las fronteras del clasificador, características lineales o no lineales, ...) es aplicable (y por tanto, debe ser considerado) en la discusión

de este experimento.

1.3.4. Subapartado c) - Interpretación alternativa

En este experimento, consideramos las mismas cuatro funciones que en 1.3.3. La diferencia ahora es que usamos estas funciones de frontera para clasificar los datos generados en el Subapartado a). Es decir, *no estamos conservando el etiquetado original*, como sí se hacía en 1.3.3. También modificamos un 10 % de las etiquetas positivas y negativas aleatoriamente. Además, mostramos las regiones de clasificado positivo y negativo de las funciones empleadas.

Con las funciones de frontera $f_i(x, y)$, realizamos el etiquetado de forma análoga que con la recta, etiquetando con el valor $sign(f_i(x, y))$.

De nuevo, usamos la función `contourf` para mostrar las regiones de clasificado que generan estas nuevas funciones dadas.

Mostramos las gráficas obtenidas:

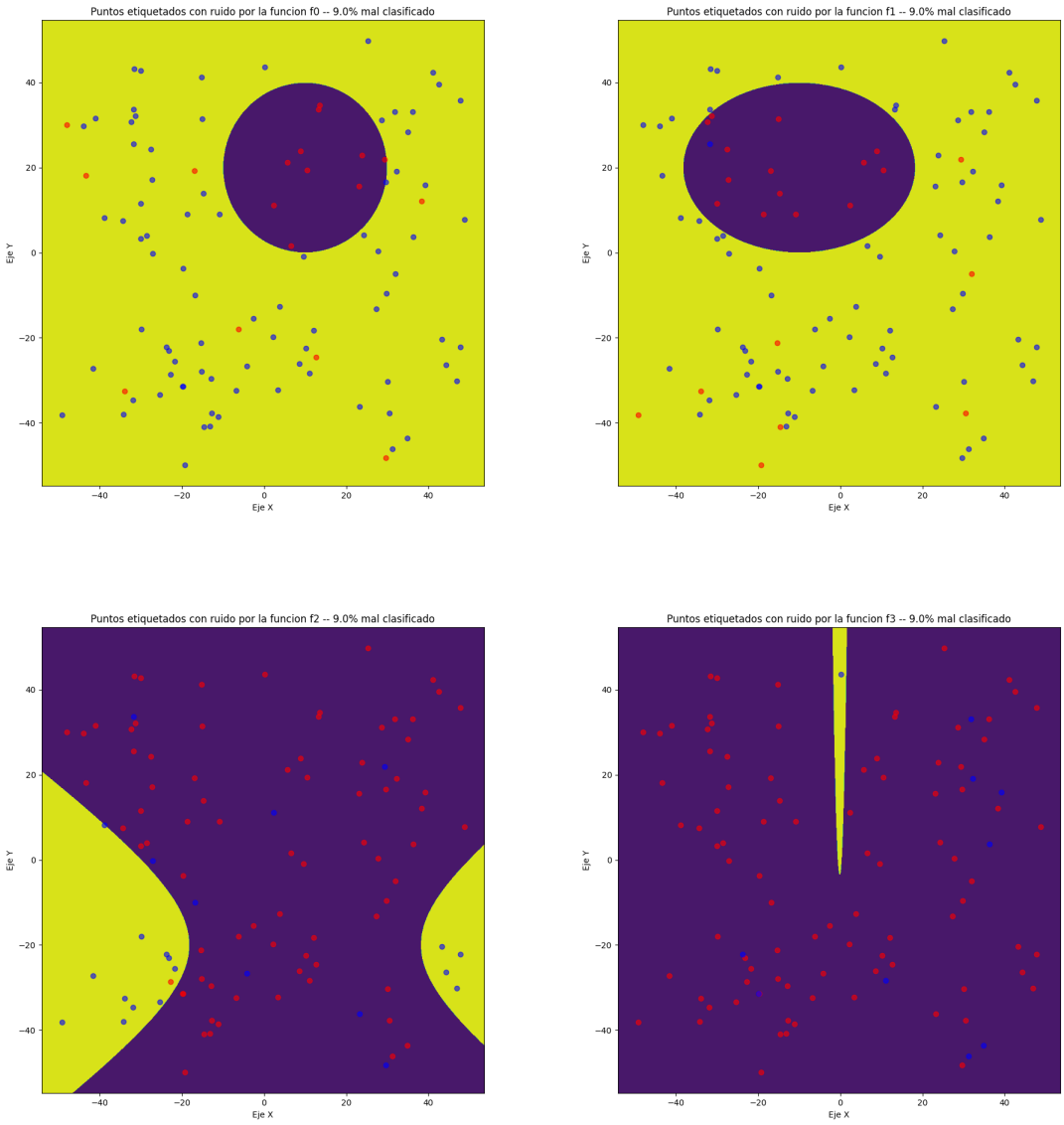


Figura 5: Etiquetado ruidoso con las cuatro funciones dadas

Comenzamos valorando la complejidad aparente de las funciones. Toda esta discusión es desde un punto de vista intuitivo. Pero no estamos cuan-

tificando estas nociones, pudiendo llegar a ser falaz. Por tanto, después de este análisis superficial, realizamos un análisis más profundo y cuantitativo.

Es claro que la recta realiza una clasificación mucho más sencilla que estas nuevas funciones de clasificación, pues divide el espacio en dos regiones sencillas: la región por encima de la recta y la región por debajo de la recta. f_0 ya es más compleja pues estamos considerando la región fuera y dentro de un círculo, que no está centrado en el origen. f_1 es algo más compleja, pues ahora no consideramos un círculo, sino una elipse (añadimos un nuevo parámetro al modelo, el achatamiento del eje x dado por el factor $\frac{1}{a^2}$). En f_2 estamos considerando regiones hiperbólicas, que viene determinada por el mismo número de parámetros que f_1, aunque la forma de las regiones cambie al cambiar la cónica modificando el signo. En este caso, hay un predominio de región de etiquetado positivo frente a la región de etiquetado negativo. Y finalmente en f_3 consideramos una parábola, cuya región de etiquetado negativo tiene un área muy pequeña comparada a la región de etiquetado positivo.

Bajo estas condiciones es algo complicado comentar si estas funciones más complejas son mejores clasificadores que la función lineal, pues esto dependerá de una serie de factores.

Pasamos a valorar la complejidad de las funciones de una forma objetiva, cuantitativa, y teniendo en cuenta la muestra de datos con la que estamos trabajando.

Para empezar, estamos trabajando con 100 puntos o datos. Debemos emplear algún método para conocer si la complejidad de nuestras funciones se adecua a la cantidad de datos con los que estamos trabajando. Esta herramienta será la **dimensión de Vapnik-Chervonenkis**. Sabemos que en el caso de la recta, $d_{VC} = d + 1 = 3$, y por tanto, la regla práctica para generalizar bien $N \geq 10 d_{VC}$ se cumple. Para el caso del círculo y de la elipse, tenemos también que $d_{VC} = 3$. Esto no es difícil de probar, al tener una dimensión tan baja, con ejemplos [1]. Por tanto, a nivel de generalización, con los 100 datos que tenemos es suficiente pues $100 \geq 3 * 10$. Para el caso de la hipérbola y de la parábola hemos comprobado que $d_{VC} = 3$ yendo ejemplo por ejemplo (puede estar mal hecho el cálculo). Por lo cual, todas las funciones tienen una cantidad suficiente de datos acorde a d_{VC} como para que no haya problemas para generalizar (siempre siguiendo la regla

práctica $N \geq d_{VC} * 10$).

Sería más adecuado realizar experimentos con un dataset de entrenamiento y otro de test para comprobar empíricamente que, efectivamente, con 100 datos no tenemos problemas de generalización.

Notar que la función lineal sigue siendo más sencilla aunque d_{VC} parezca indicar lo contrario. Esto se debe a que en el análisis VC, nos situamos en el peor de los casos, es un análisis muy pesimista.

Otro punto a tener en cuenta es el problema que estamos considerando. Sigue sin tener demasiado sentido plantearse el clasificado en abstracto y comparar lo bueno que son los clasificadores desde este punto de vista. Si las características del problema tienen un carácter lineal (al doble de ingresos, el doble de probabilidad de que nos acepten la solicitud de un crédito, por ejemplo), las funciones de clasificado no lineales no tienen demasiado sentido. Si por otro lado las características de los datos no son lineales, entonces puede tener más sentido usar clasificadores no lineales a pesar de que representen un modelo más complejo, pues serían más correctos desde la perspectiva del problema a resolver.

Respecto al proceso de modificación de etiquetas, las condiciones en la que hemos realizado este proceso también complica algo las cosas. Hemos modificado, aleatoriamente, un 10 % de las etiquetas positivas y otro 10 % de las etiquetas negativas. En este proceso da bastante igual la forma que tome f. En el sentido en el que, podemos pensar que la recta es muy robusta frente al ruido, o más robusta que una elipse. Pero a la hora de ejecutar el cambio aleatorio, esto no se tiene en cuenta.

Esto se refleja en que obtenemos constantemente un error del 9 %. Que no sea un 10 % puede ser por la cantidad de etiquetas negativas y positivas. Al no ser números que al multiplicarlos por 0,1 den un entero, se puede redondear por la baja, obteniendo así este error porcentual algo más bajo que el 10 %.

Sin embargo, si que hemos detectado una característica en la que la forma de f influye en el proceso. Esto se ve claro con f_3. Al tener una región mucho más pequeña que la contraria (en este caso, región de etiquetado negativo

[1] En [1] se muestra de forma visual la prueba de esto con los conjuntos dados de datos. En [2] se realiza una prueba formal de este hecho.

mucho más pequeña que la región de etiquetado positivo), es muy probable que no hayamos clasificado ningún punto como negativo, y por tanto realizamos modificaciones del tipo $negativo \rightarrow positivo$, solo modificaciones del tipo $positivo \rightarrow negativo$. Aún así, esto no es demasiado relevante

Así queda justificado que el proceso de introducción de ruido no es muy informativo. Un proceso de introducción de ruido mucho más interesante podría haber sido seleccionar aleatoriamente un porcentaje de las etiquetas, tanto positivas como negativas, y en vez de cambiar arbitrariamente su etiqueta, modificar en un cierto rango acotado su posición, y ver si en esa nueva posición se etiquetaría con el otro valor.

Esto nos daría más información sobre la robustez de nuestro clasificador frente al ruido. Por ejemplo, en el caso de la recta, un punto alejado de la frontera no cambiaría de etiqueta (con un rango de variación de coordenadas razonable), mientras que un punto muy cercano a la frontera sí que cambiaría. Como se puede ver en la imagen, la mayoría de puntos no se encuentran muy pegados a la frontera. Por otro lado, considerando f_3, podríamos pensar que todos los puntos de etiquetado negativo se saldrían de la región de etiquetado negativo, al ser esta muy estrecha, concluyendo que el clasificador es muy frágil frente al ruido.

Otra forma de medir esta *robustez* sería calcular la longitud de las fronteras de nuestro clasificador. A mayor longitud, tenemos un modelo más complejo, que es más probable de sobreajustar. Además, a mayor longitud es más probable que un cambio de coordenadas aleatoria, anteriormente discutida, provoque un cambio de etiqueta. Y por tanto es más probable que el ruido afecte a nuestro problema.

Resumiendo, consideramos insuficiente la información dada para comparar las funciones de clasificación. Por tanto, realizamos nuevas consideraciones para tratar de realizar un mejor análisis, y proponemos una forma sintética de introducir ruido que pensamos que puede ser más adecuada a la hora de analizar la robustez de los clasificadores frente al ruido.

2. Ejercicio 2 - Modelos lineales

La función `ajusta_PLA(datos, label, max_iter, vini)` está implementada bajo el nombre `perceptron_learning_algorithm`. Esta función recibe los parámetros indicados por el guión, y uno adicional. Este parámetro adicional es el booleano `verbose`, que cuando se pasa como `True`, devuelve la solución, las iteraciones consumidas y el error porcentual en la muestra de cada iteración. Cuando es `False`, solo devolvemos la solución y el número de iteraciones consumidas. Notar que las iteraciones consumidas se corresponden con la cantidad de *Epochs* consumidos (pasadas completas sobre todo el conjunto de datos). El error por cada iteración también se refiere al error por cada *Epoch*. El algoritmo es tan sencillo que no parece necesario incluir el pseudo-código. Además, la implementación es también muy directa, por lo que no parece necesario hacer aclaraciones sobre decisiones tomadas para implementar el algoritmo.

2.1. Apartado a) - Algoritmo Perceptron

Ejecutamos el algoritmo *PLA* sobre los datos generados en 1.3.1. Recordemos que estos datos han sido clasificados por una recta aleatoria, sin introducir todavía error. Por lo tanto, son claramente linealmente separables, y por ello, el algoritmo deberá alcanzar una solución en un número finito de iteraciones, con un error dentro de la muestra del 0,0 %.

2.1.1. Subapartado 1

Para usar los mismos datos que en 1.3.1, usamos variables globales para guardar en ese primer ejercicio los datos generados. Otra observación pertinente es que a la función `perceptron_learning_algorithm` le tenemos que pasar un número máximo de iteraciones. Queremos alcanzar la solución antes de agotar las iteraciones, porque sabemos que la muestra es linealmente separable. Por tanto hacemos una asignación lo suficientemente grande: `max_iterations = 5000`.

Mostramos de nuevo la gráfica de los datos etiquetados para asegurarnos que estamos en el mismo ambiente que en 1.3.1:

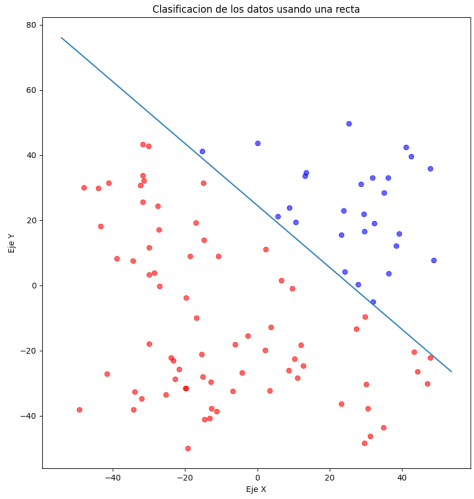


Figura 6: Datos clasificados por la recta aleatoria

Lanzamos 10 veces el algoritmo para el vector inicial cero, y otras 10 para vector inicial aleatorio. Es de esperar que con vector inicial cero, tengamos los mismos resultados, pues PLA no tiene ningún componente aleatorio si fijamos la solución inicial. Pero al considerar diez experimentos para vector inicial cero, lanzamos diez ejecuciones para que quede más clara la comparación. Mostramos los resultados del experimento por pantalla

```
-> Errores finales(tanto por uno): [0.0, 0.0, 0.0, 0.0, 0.0,
                                    0.0, 0.0, 0.0, 0.0, 0.0]
-> Valor medio de los errores: 0.0 %
-> Desviacion tipica de los errores: 0.0
-> Iteraciones consumidas: [26, 26, 26, 26, 26, 26, 26, 26, 26, 26]
-> Valor medio de iteraciones: 26.0
-> Desviacion tipica del numero de iteraciones: 0.0
```

Listing 1: Resultados para vector inicial cero

```
-> Errores finales(tanto por uno): [0.0, 0.0, 0.0, 0.0, 0.0,
                                    0.0, 0.0, 0.0, 0.0, 0.0]
-> Valor medio de los errores: 0.0 %
-> Desviacion tipica de los errores: 0.0
-> Iteraciones consumidas: [63, 49, 27, 52, 45, 28, 13, 60, 39, 39]
-> Valor medio de iteraciones: 41.5
```

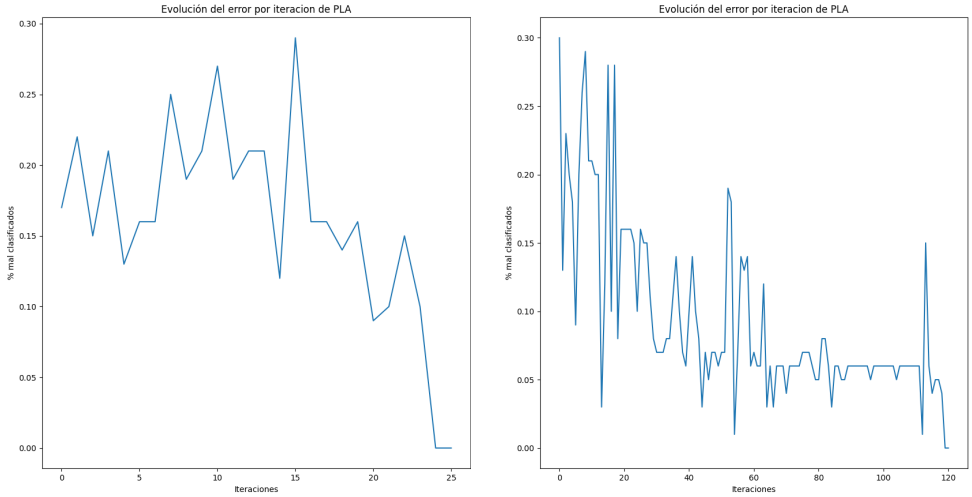
```
-> Desviacion tipica del numero de iteraciones: 14.8340
```

Listing 2: Resultados para vector inicial aleatorio

Ambos algoritmos, como era de esperar, convergen a una solución de error $E_{in} = 0{,}0\,\%$. Por lo tanto, no merece hacer comparaciones sobre los errores alcanzados. Sin embargo, las iteraciones necesarias para converger sí que son distintas. Para el vector inicial cero, tenemos un menor número medio necesario de iteraciones necesarias (realmente al no tener aleatoriedad no hace falta considerar valores medios) que con vector inicial aleatorio, y la diferencia es significativa.

Además, el vector inicial aleatorio tiene una variabilidad, indicada por la desviación típica, que con vector inicial cero no tenemos. Y con ello, podemos pensar que partiendo de un vector inicial cero, tenemos resultados más estables, con ausencia de variabilidad por factores aleatorios.

También mostramos los gráficos de convergencia para una ejecución de *PLA* con vector inicial cero, y otro para vector inicial aleatorio:



(a) V_{ini} inicial cero

(b) V_{ini} aleatorio

Figura 7: Gráficas de convergencia para PLA

Con estos gráficos y los resultados de los 10 experimentos concluimos que el vector inicial cero es mucho más rápido en converger, y mucho más estable que el vector inicial aleatorio. Además, en esta gráfica, vemos que hemos tomado una mala ejecución con vector inicial aleatorio. Esto es bastante probable por la alta variabilidad que ya hemos comentado.

2.1.2. Subapartado 2

Repetimos el mismo experimento pero usando ahora los datos con ruido, generados en 1.3.2. De nuevo, para conseguir esto, lo que hacemos es guardar las etiquetas ruidosas en una variable global, y reusarlas aquí. De nuevo, mostramos la gráfica de etiquetado ruidoso para asegurarnos de que estamos trabajando con los mismos datos que en 1.3.2:

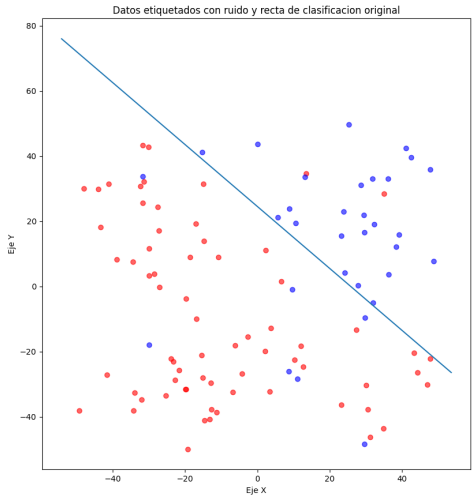


Figura 8: Datos clasificados por la recta aleatoria con ruido sintético

Y de nuevo, mostramos los resultados del experimento:

```
-> Errores finales(tanto por uno): [0.09, 0.09, 0.09, 0.09, 0.09,
                                    0.09, 0.09, 0.09, 0.09, 0.09]
-> Valor medio de los errores: 8.999999 %
-> Desviacion tipica de los errores: 0.0
```

```
-> Iteraciones consumidas: [5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000]
-> Valor medio de iteraciones: 5000.0
-> Desviacion tipica del numero de iteraciones: 0.0
```

Listing 3: Resultados para vector inicial cero y ruido en las etiquetas

```
-> Errores finales(tanto por uno): [0.16, 0.18, 0.12, 0.13, 0.12,
                                    0.13, 0.18, 0.17, 0.26, 0.27]
-> Valor medio de los errores: 17.2 %
-> Desviacion tipica de los errores: 0.0515
-> Iteraciones consumidas: [5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000]
-> Valor medio de iteraciones: 5000.0
-> Desviacion tipica del numero de iteraciones: 0.0
```

Listing 4: Resultados para vector inicial aleatorio y ruido en las etiquetas

En este caso, los datos no son linealmente separables por el ruido, y por lo tanto, *PLA* no convergerá. Por lo cual, el algoritmo consumirá el número máximo de iteraciones, el cual está establecido a un valor de 5000. Teniendo en cuenta que anteriormente necesitábamos de media aproximadamente 40-45 iteraciones sobre *Epochs*, este valor de 5000 parece más que razonable para ajustarnos a la muestra.

Al consumir ambos el máximo de iteraciones, no podemos realizar comparaciones respecto a esto entre los dos algoritmos. De nuevo, con vector inicial cero tenemos un error significativamente menor que el error medio con vector inicial aleatorio. Y de nuevo, al tener ausencia de factores aleatorios, tenemos una estabilidad que el vector inicial aleatorio no tiene.

Al igual que hacíamos en el subapartado anterior, mostramos la gráfica de convergencia. Con la diferencia de que esta vez, solo mostramos la ejecución para vector inicial aleatorio.

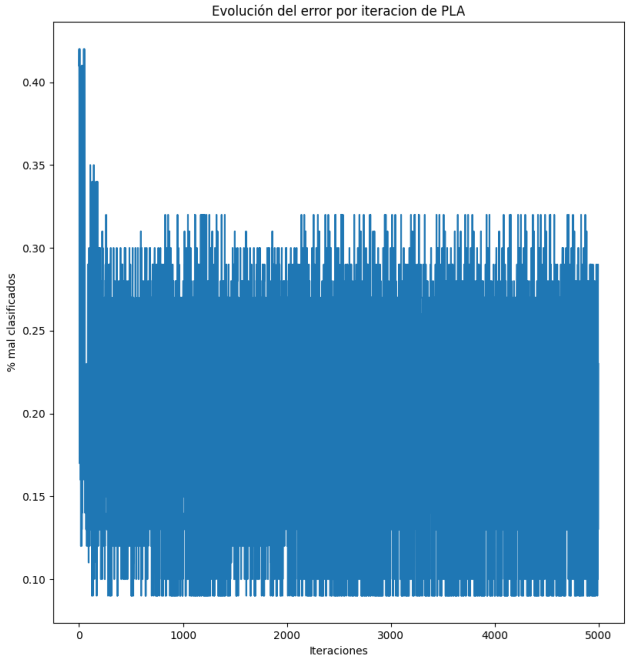


Figura 9: Gráfica de convergencia - PLA con $V_{ini} = 0$ y ruido sintético

Se ve claramente el comportamiento errático de *PLA* que conocíamos de teoría. Por tanto, no tiene sentido que mostremos otro gráfico viendo el progreso usando vector inicial aleatorio. Este gráfico es útil para motivar el uso de la mejora *PLA-Pocket*. Usando esta mejora del algoritmo, obtendríamos una gráfica de convergencia monótonamente decreciente.

Además, queda claro que 5000 iteraciones son más que suficientes, pues nunca vamos a lograr $E_{in} = 0$, y además, al no decrecer el error monótonamente tampoco parece beneficioso consumir más iteraciones.

Concluyendo, hemos comprobado empíricamente el mejor comportamiento de PLA cuando usamos vector inicial cero. Además, hemos visto el comportamiento algo errático de *PLA* cuando los datos no son linealmente separables.

2.2. Apartado b) - Regresión logística

Empezamos esta sección explicando cómo vamos a generar nuestra muestra de datos. Tenemos una población hipotética $\chi = [0, 2] \times [0, 2]$ y la probabilidad de tomar un $x \in \chi$ es uniforme. Por tanto, para generar tanto las muestras de entrenamiento como de test, lo que hacemos es invocar a una función que nos da un punto aleatorio con probabilidad uniforme en ese intervalo bidimensional.

La función `generate_dataset` toma un número dado de puntos aleatorios en χ. Usaremos esta función tanto para las muestras de entrenamiento como para las de test.

Para generar la recta aleatoria de etiquetado, tomamos dos puntos aleatorios (con `generate_dataset(2)`) y calculamos la recta que pasa por esos dos puntos con `calculate_straight_line_from_two_points`.

En regresión logística, usamos la siguiente función de etiquetado:

$$f_w(x) := \begin{cases} 1 & si \quad \sigma(signal) \geq 0{,}5 \\ \\ -1 & si \quad \sigma(signal) < 0{,}5 \end{cases}$$

donde $\sigma(x) := \frac{1}{1+e^{-x}}$ y $signal$ es la señal lineal dada por $signal := w^T x$. Es claro que lo que aprendemos de este modelo son los valores de los pesos: w.

Antes de realizar el experimento 100 veces, vamos a ejecutar una repetición del experimento, mostrando ciertas gráficas y realizando algunos análisis antes. Tras esto, lanzamos 100 veces el experimento, obteniendo más información para realizar los análisis.

2.2.1. Regresión logística con Gradiente Descendente Estocástico

La primera consideración para implementar regresión logística junto a gradiente descendente estocástico es la necesidad de añadir una columna de unos a la matriz de datos. Esta columna en la primera práctica ya venía dada. Pero en este caso, necesitamos añadirla manualmente. Esta columna

representa el término independiente de la combinación lineal que subyace al modelo lineal. Podríamos hacerlo sin esta columna, pero en ese caso no podemos fijar un *offset* a los datos. Para añadir la columna de unos disponemos de la función `add_colum_of_ones`.

El algoritmo de gradiente descendente estocástico ya fue explicado en la anterior práctica. Todas las consideraciones sobre la implementación de este algoritmo ya fueron expuestas en la anterior memoria (por ejemplo, la generación de *minibatches* mezclados aleatoriamente de forma eficiente y cómoda). Así que para este problema, tenemos que realizar las siguientes modificaciones:

- El error que consideramos viene dado por $E_{in} = \frac{1}{N} \sum ln(1+e^{-y_i w^T x_i})$. Este es el error que vamos a mostrar en las gráficas de convergencia.
- Por tanto, para el gradiente usamos la expresión $\nabla E_{in} = -\frac{1}{N} \sum \frac{y_n x_n}{1+e^{y_n w^T x_n}}$
- La solución inicial será un vector de ceros
- La condición de parada será $||w(t+1) - w(t)|| \leq 0{,}01$
- Learning rate $\eta = 0{,}01$

Al tener una condición distinta de parada, dejamos el número máximo de iteraciones a `None`, para iterar hasta que tengamos la distancia entre dos soluciones consecutivas por debajo de la cota dada.

Otro aspecto crítico es establecer el *batch size*. El guión nos indica que debemos usar un `batch size = 1`, pero realizamos una corta exploración sobre distintos valores. Con un valor de 1, necesitamos muchas iteraciones pero llegamos a un muy buen error, aproximadamente en $[0{,}005 - 0{,}01]$ según la semilla aleatoria. Para `batch_size = 4`, requerimos entre 1500 - 5000 iteraciones, pero el error pasa a estar en $[0{,}01 - 0{,}05]$. A partir de `batch_size = 8`, tenemos muy pocas iteraciones (por debajo de 1000), pero con un error por encima de 0,2. Por tanto, hemos comprobado de forma empírica y rápida que el valor adecuado para `batch_size` es 1. No tenemos mucha intuición por lo que esto es así. Una hipótesis es que, al estar en un problema de clasificación, lo adecuado es visitar punto a punto e ir ajustando localmente con la información de ese punto. Este es un esquema parecido a *PLA*, donde visitamos punto a punto y ajustamos si hemos fallado en ese punto.

Además, al estar considerando solo la condición de distancia entre soluciones consecutivas, con $batch\_size > 1$ puede ser que acabemos compensando la dirección del error entre los múltiples puntos (por ejemplo, errores con la misma dirección pero distinto sentido), provocando una parada prematura. Intuimos por tanto que estableciendo otras condiciones de parada (como que el error esté por debajo de cierta cota) puede funcionar mejor si queremos establecer un $batch\_size > 1$. Esta última idea se refuerza más adelante, cuando vemos que la muestra de datos es linealmente separable, por lo que deberíamos ser capaces de alcanzar error $0.0\,\%$ en la muestra, iterando el suficiente número de veces.

2.2.2. Conjunto de datos con el que trabajamos

Generamos el conjunto de datos, de la forma que ya hemos comentado:

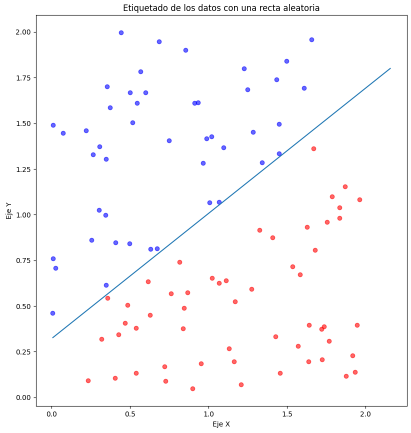


Figura 10: Muestra de datos etiquetada por la recta aleatoria

Claramente el proceso de etiquetado hace que la muestra de datos sea linealmente separable. Por tanto, es de esperar que regresión logística con gradiente descendente estocástico alcance un $E_{in} = 0{,}0\,\%$.

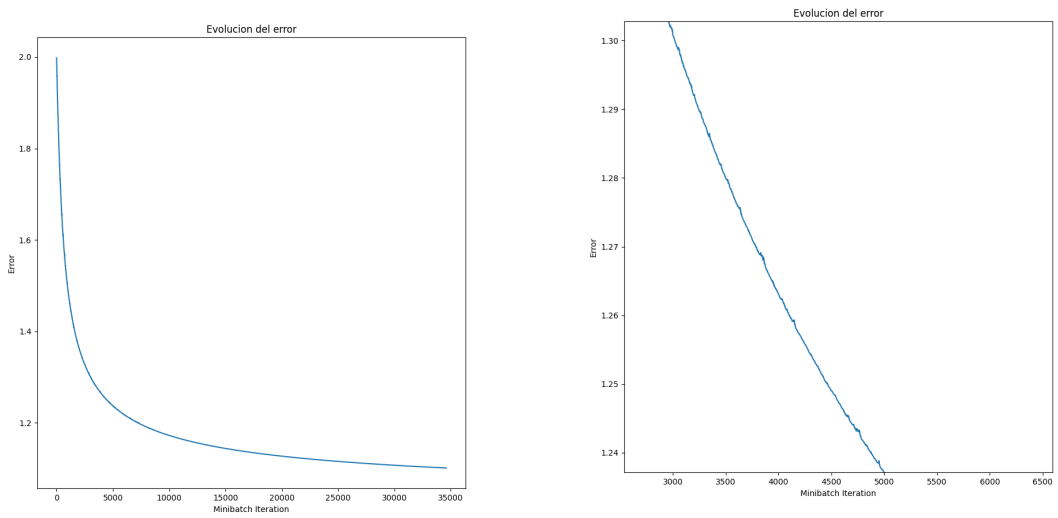
2.2.3. Entrenamiento y resultados

Al lanzar *LGR* con *SGD* obtenemos la siguiente salida por pantalla:

```
- Pesos obtenidos: [-2.57253883 -4.89587868  7.85922036]
- Iteraciones consumidas: 34600
- Error final alcanzado (error LGR): 1.1013766997935295
- Error final alcanzado (porcentaje mal clasificado): 0.0 %
```

Listing 5: Resultados del entrenamiento

Los resultados obtenidos eran los que esperábamos. El error *cross-entropy* no se anula, mientras que el porcentaje de puntos mal clasificados es cero. Mostramos la gráfica de convergencia:



(a) Gráfica de convergencia

(b) Gráfica de convergencia ampliada

Figura 11: Gráficas de convergencia para LGR-SGD

Vemos que el comportamiento del gradiente descendiente es adecuado. Pareciera que en la figura 11a tenemos convergencia monótona. Por eso mostramos un *"zoom"* de la gráfica para ver que no es así, como es lo normal cuando usamos minibatches para el gradiente descendente.

Podemos también mostrar la recta de frontera junto a los datos con su etiquetado original:

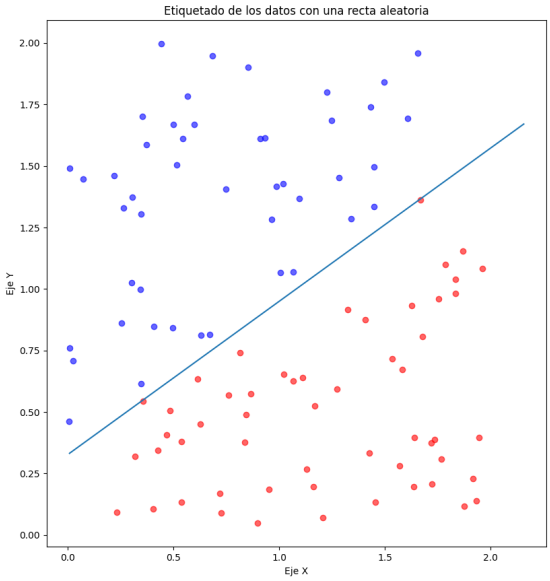


Figura 12: Frontera solución junto al etiquetado original

Algunos puntos están más cercanos a la frontera que con la recta de etiquetado original. Con esto, podemos prever que los puntos mal clasificados por el modelo estarán en esa zona fronteriza.

2.2.4. Un ejemplo concreto de E_{test}

Antes de lanzar los 100 experimentos, generamos un *dataset* de test y mostramos cómo lo estamos haciendo fuera de la muestra. El tamaño de este *test_dataset* será de 10^4 puntos. Este tamaño será también el usado en las 100 iteraciones del experimento.

Por pantalla mostramos que estamos fallando el 3,08 % de los datos del test. Es un valor razonable fuera de la muestra, por lo que podemos suponer

que estamos generalizando bastante bien. Mostramos los puntos que han sido mal clasificados:

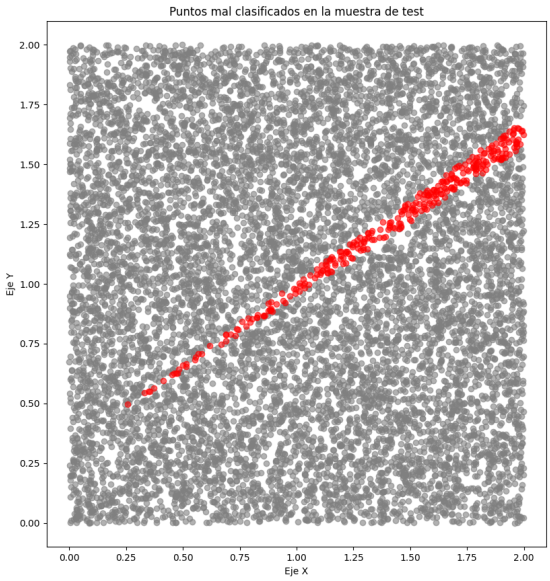


Figura 13: Puntos mal clasificados en el conjunto de test

Y como ya habíamos comentado, los puntos mal clasificados se encuentran en la región fronteriza

2.2.5. Experimento

Lanzamos 100 veces el experimento y mostramos los resultados. En cada iteración del experimento, generamos una nueva muestra de datos (como en 2.2.2). Con una recta aleatoria, etiquetamos los datos. Lanzamos *LGR-SGD* para este etiquetado. Generamos del mismo modo un *test_dataset* de 10^4 puntos, etiquetados con la recta aleatoria. Y con esto, evaluamos el valor de E_{test}.

Este proceso es algo lento porque ya hemos visto que se necesitan muchas iteraciones de *SGD* con `batch_size = 1`.

El resultado que mostramos por pantalla es:

```
--> Media de iteraciones sobre minibatches consumidas: 40640.0
--> Media de iteraciones sobre epochs consumidas: 406.4
--> Media del porcentaje de puntos mal clasificados fuera de
    la muestra: 2.777 %
--> Desviacion tipica del error fuera de la muestra: 0.01779
```

Listing 6: Resultado de las 100 iteraciones del experimento

Estos resultados reafirman lo que hemos ido comentando. Notar que no estamos mostrando el error dentro de la muestra, pues al generar *datasets* linealmente separables, potencialmente estamos alcanzando un $E_{in} = 0{,}0\,\%$. Este $E_{in} = 0{,}0\,\%$ debería hacernos pensar si estamos cometiendo *overfitting* sobre la muestra de entrenamiento. Sin embargo la media de error fuera de la muestra es de $2{,}777\,\%$, por lo que estamos obteniendo una buena generalización.

Por otro lado, la desviación típica del error fuera de la muestra indica una gran estabilidad de la capacidad de generalización fuera de la muestra, pues es un valor para la desviación muy bajo.

Es notable la gran cantidad de épocas que necesita *SGD* para converger. Sin embargo, los buenos resultados justifican estos tiempos de entrenamiento.

3. Ejercicio Extra - Clasificación de Dígitos

Consideramos el conjunto de datos manuscritos y seleccionamos las muestras asociadas con los dígitos 4 y 8. Tomamos la muestra de entrenamiento y la muestra de test de los dos tipos de ficheros proporcionados. Para ello, hacemos uso de la función dada por los profesores de prácticas: `readData`.

Esta muestra de datos nos da las características *intensidad promedio* y *simetría*.

3.1. Apartado 1

Es bastante sencillo plantear un problema de clasificación binaria. Queremos encontrar una función $g : \chi \to \{1, -1\}$ de modo que si toma un dato asociado a un 4, devuelva 1. Y si toma un dato asociado al 8, devuelva -1. Lógicamente los datos vienen dados en la forma:

$$x \in \chi \Rightarrow x = (simetria_x, intensidad_x)$$

Con esto, podemos formalizar nuestro espacio como $\chi = \mathbb{R}^2$.

Como vamos a trabajar con modelos lineales, lo que haremos será tomar como g una recta de clasificación. Es decir

$$g_w(x) := sign(w^T x)$$

donde los pesos w son los parámetros que queremos aprender.

Para que el modelo lineal de clasificación funcione lo mejor posible, debemos añadir una columna de unos a la matriz de datos. Esto es para representar el término independiente de la combinación lineal

$$w^T x = \sum_{i=0}^{3} w_i x_i$$

Como ya hemos comentado en ejercicios anteriores, esto lo podemos lograr usando la función `add_colum_of_ones`.

Esto último nos hace considerar $\chi = \{1\} \times \mathbb{R}^2$.

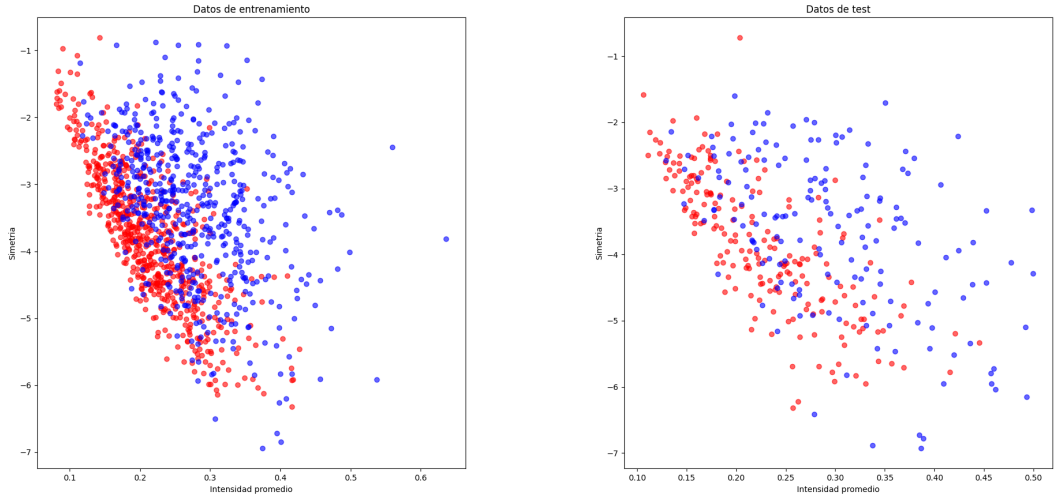
3.2. Apartado 2

En este apartado usamos un modelo de regresión lineal y aplicamos *PLA-Pocket* como mejora. Esto es, ajustamos los datos con un modelo de regresión lineal clásico, obteniendo una primera solución. Después, lanzamos *PLA-POCKET* con solución inicial la primera solución, buscando mejorar dicha solución.

Para la parte de regresión lineal, usaremos el algoritmo de la pseudo-inversa. Tenemos una cantidad no muy grande de datos, así que nos parece adecuado. En un principio se podría pensar que, al tener una primera solución por una fórmula cerrada, esta solución no puede ser mejorada. Sin embargo, esta solución cerrada por la pseudo-inversa minimiza el error cuadrático medio, al resolver un problema de regresión. Nuestra intención es minimizar el porcentaje de puntos mal clasificados, y ahí es donde entra en juego aplicar *PLA-POCKET* con la anterior solución como punto de partida.

3.3. Datos con los que trabajamos

A continuación mostramos los conjuntos de entrenamiento y de test:



(a) Conjunto de entrenamiento (b) Conjunto de test

Figura 14: Conjuntos de datos cargados

3.3.1. Entrenamiento usando regresión lineal

Lanzamos en primera instancia el algoritmo de la pseudo-inversa. Con la cantidad tan pequeña de datos que estamos usando, se obtiene una solución cerrada rápidamente. El *performance* de esta solución se muestra como:

```
--> Lanzando el algoritmo de la pseudoinversa
- Solucion inicial: [-0.50676351  8.25119739  0.44464113]
- Error conseguido en la muestra: 0.2278
- Error conseguido fuera de la muestra: 0.2513
```

Listing 7: Salida del entrenamiento de la regresión lineal

Los dos errores que mostramos son porcentajes en tantos por uno. Por tanto, tenemos un $E_{in} = 22{,}78\,\%$ y un $E_{test} = 25{,}13\,\%$. El algoritmo *PLA-POCKET* todavía tiene margen de mejora.

3.3.2. Entrenamiento usando PLA-POCKET

A partir de la anterior solución, lanzamos PLA-POCKET buscando mejorar dicha solución. Como parámetro a este algoritmo, establecemos un número máximo de iteraciones (sobre *Epochs*) de 1000. Restringimos el número máximo de épocas, porque este algoritmo tiene una penalización de tiempo de ejecución respecto *PLA* clásico. En cada iteración sobre un dato (que no sobre una época), tenemos que evaluar el error porcentual para saber si estamos obteniendo un mejor candidato. Por tanto, estamos haciendo llamadas constantes a esta función de error, que es lenta.

La salida de este entrenamiento es:

```
- Solucion: [ -8.50676351 163.00130454   8.99007863]
- Iteraciones consumidas: 1000
- Error conseguido en la muestra: 0.2102
- Error conseguido fuera de la muestra: 0.2513
```

Listing 8: Salida del entrenamiento de PLA-POCKET

No hemos mejorado apenas el error porcentual. Dentro de la muestra, 22,78 % $\rightarrow$ 21,02 %, mientras que fuera de la muestra, 25,13 % $\rightarrow$ 25,13 % la mejora no es visible con dos cifras decimales.

Mostramos la gráfica de convergencia:

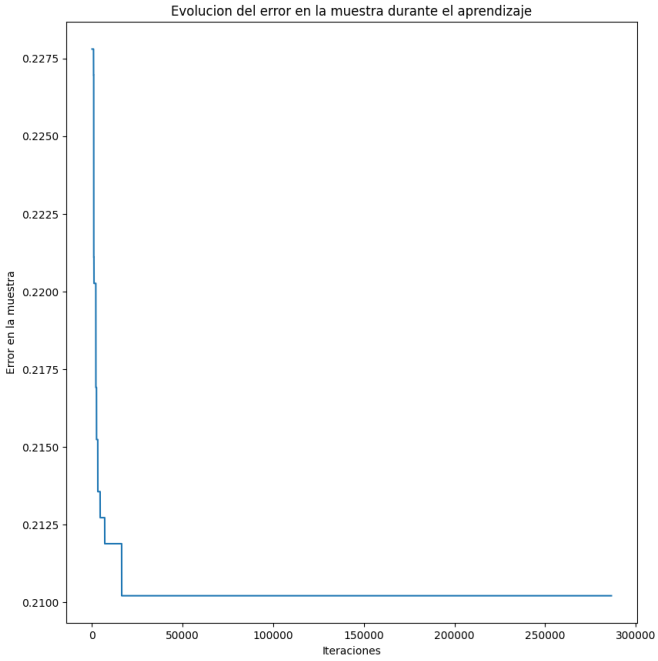
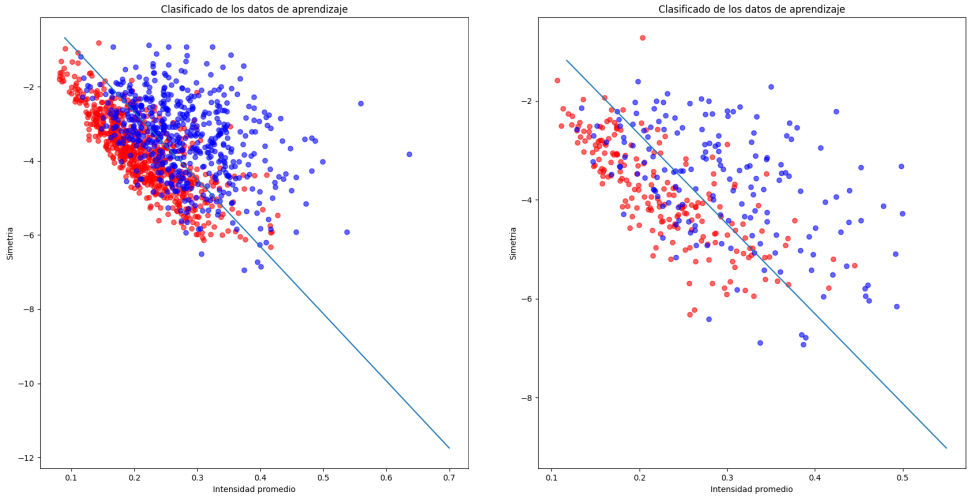


Figura 15: Gráfica de convergencia para PLA-POCKET

Estamos mostrando el error por iteraciones sobre datos. Vemos que a partir de las 50000 iteraciones no conseguimos mejorar el error. Sabiendo que nuestra muestra de datos de entrenamiento contiene 1194 datos, parece entonces razonable acotar el número de épocas a 50 épocas. Más adelante lanzaremos el algoritmo de aprendizaje con solo 50 épocas.

Vemos que hemos mejorado la solución dada por el algoritmo de la pseudoinversa, aunque es una mejora tan ligera que cabe preguntarse si merece la pena gastar ese tiempo de cómputo.

Las gráficas de etiquetado sobre las dos muestras de datos son:



(a) Clasificación en la muestra de aprendizaje

(b) Clasificación en la muestra de test

Figura 16: Clasificación tras el entrenamiento de POCKET

Lanzando el algoritmo con un máximo de 50 iteraciones sobre *EPOCH*, obtenemos unos resultados ligeramente peores, pero en un tiempo mucho más razonable (aunque sigue tardando entre medio minuto y un minuto), y una gráfica de convergencia:

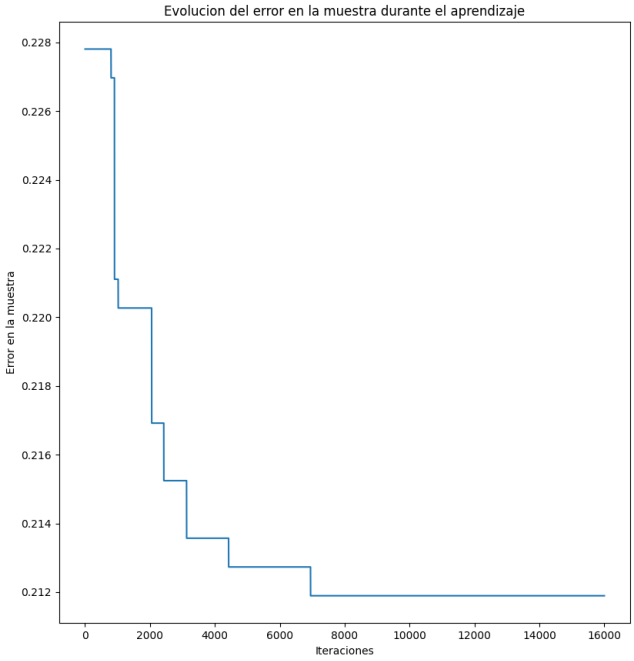


Figura 17: Gráfica de convergencia para PLA-POCKET con menos iteraciones

Los resultados obtenidos son:

```
- Solucion: [ -6.50676351 121.7775949    6.66351613]
- Iteraciones consumidas: 50
- Error conseguido en la muestra: 0.2118
- Error conseguido fuera de la muestra: 0.2513
```

Listing 9: Salida del entrenamiento de PLA-POCKET

Por tanto, en una aplicación real, deberíamos entrar a valorar si merece la pena perder tiempo de cómputo mejorando ligeramente la solución, o si nos conformamos con la solución dada por la pseudoinversa. Si estamos explorando distintas soluciones y probando ideas, probablemente nos conformemos con el resultado de la pseudoinversa. Pero si ya hemos decidido que este es un buen modelo, y estamos entrenando para aplicar la función

g_w aprendida a datos reales, entonces es claro que merece la pena el tiempo invertido en mejorar la solución (sin llegar a provocar *overfitting*).

3.3.3. Cotas sobre E_{out}

Para esto vamos a dar dos cotas, una basada en E_{in} y otra basada en E_{test}. En ambas vamos a considerar una tolerancia de $\delta = 0{,}05$. La cota de generalización VC usando E_{in} viene dada por:

$$E_{out}(h) \leq E_{in}(h) + \sqrt{\frac{8}{N} log \frac{4((2N)^{d_{VC}+1})}{\delta}}$$

Para el perceptron bidimensional sabemos que $d_{VC} = 3$. Estamos trabajando con $N = 1194$. Además, hemos mostrado la salida por pantalla que nos indicaba que $E_{in} = 0{,}2102$. Así que podemos sustituir estos valores en la fórmula:

$$E_{out}(h) \leq 0{,}2102 + \sqrt{\frac{8}{1194} log \frac{4((2 * 1194)^3)}{0{,}05}}$$

operando:

$$E_{out}(h) \leq 0{,}2102 + 0{,}4309 = 0{,}6411 \Rightarrow E_{out}(h) \leq 0{,}6411$$

Recordemos que estamos trabajando con errores porcentuales en tantos por uno, por que que nuestra cota es más interpretable si la expresamos en tantos por ciento: $E_{out} \leq 64{,}11\,\%$.

Ahora, para E_{test} vamos a utilizar la siguiente fórmula para la cota:

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

considerando $E_{in} \equiv E_{test}$. Ahora tenemos una muestra de test con $N = 366$. Además, como ya la función g fue escogida por el proceso de en-

trenamiento sobre la muestra de aprendizaje, podemos considerar $|H| = 1$. Se ha mostrado por pantalla que $E_{test} = 0{,}2513$. Con esto podemos operar:

$$E_{out}(h) \leq 0{,}2513 + \sqrt{\frac{1}{2 * 366} log \frac{2}{0{,}05}}$$

Y con ello tenemos:

$$E_{out}(h) \leq 0{,}2513 + 0{,}0709 \Rightarrow E_{out}(h) \leq 0{,}3222$$

Y de nuevo, al expresar esta cota en tantos por ciento en vez de tantos por uno, tenemos que $E_{out} \leq 32{,}22\,\%$.

Claramente la mejor cota la obtenemos empleando E_{test}, como era de esperar (pues la intención de usar esta muestra de datos es que nos de una información más fiel sobre E_{out}). Esta cota es casi la mitad que la que obtenemos con E_{in}.