COMPLEJIDAD DE $\mathcal{H}$ Y MODELOS LINEALES
MEMORIA P2 DE APRENDIZAJE AUTOMÁTICO

Ignacio Garach Vélez

8 de mayo de 2022

Índice

1. Introducción

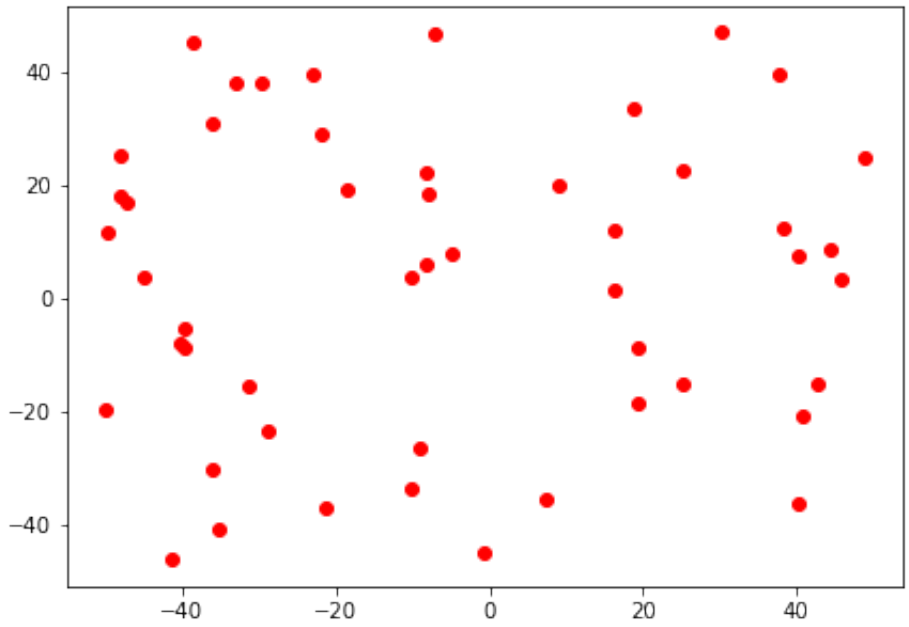
En esta práctica vamos a abordar el estudio de la influencia de la complejidad de los modelos en el problema de aprendizaje, trataremos de ajustar nubes de puntos con ruido introducido artificialmente y observaremos que por mucho que compliquemos un modelo no será posible ajustar ni controlar dicho ruido. Además se tendrá una gran pérdida de capacidad de generalización del modelo como hemos estudiado mediante la teoría de Vapnik y Chervonenkis.

Por otro lado implementaremos dos modelos lineales clásicos, el Perceptron y la Regresión Logística, estudiaremos como reaccionan a variaciones a sus parámetros e inicializaciones y comprobaremos su capacidad en diferentes condiciones. Finalmente se utilizarán para abordar

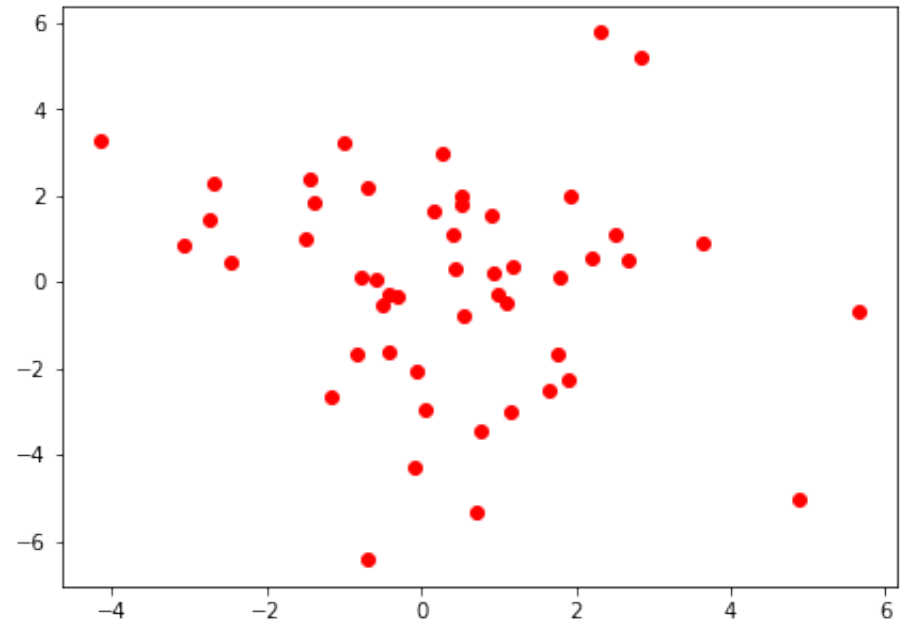
de nuevo el problema de clasificación de dígitos, utilizaremos los pesos obtenidos por regresión para inicializar el resto de algoritmos y ver como se comportan, finalmente calcularemos cotas del error fuera de la muestra mediante el error en la muestra y la complejidad del modelo, a través de una desigualdad teórica obtenida a partir de la desigualdad de Hoeffding.

2. Ejercicio sobre complejidad de $\mathcal{H}$ y ruido

En el primer apartado se nos pide probar las funciones de generación de datos proporcionadas.
Primero utilizaremos *simula_unif(N, dim, rango)* que genera una lista de N vectores de dimensión dim con elementos distribuidos en el intervalo rango en cada uno de ellos. Probamos con $N = 50$, dim $= 2$ y rango $[-50, 50]$:

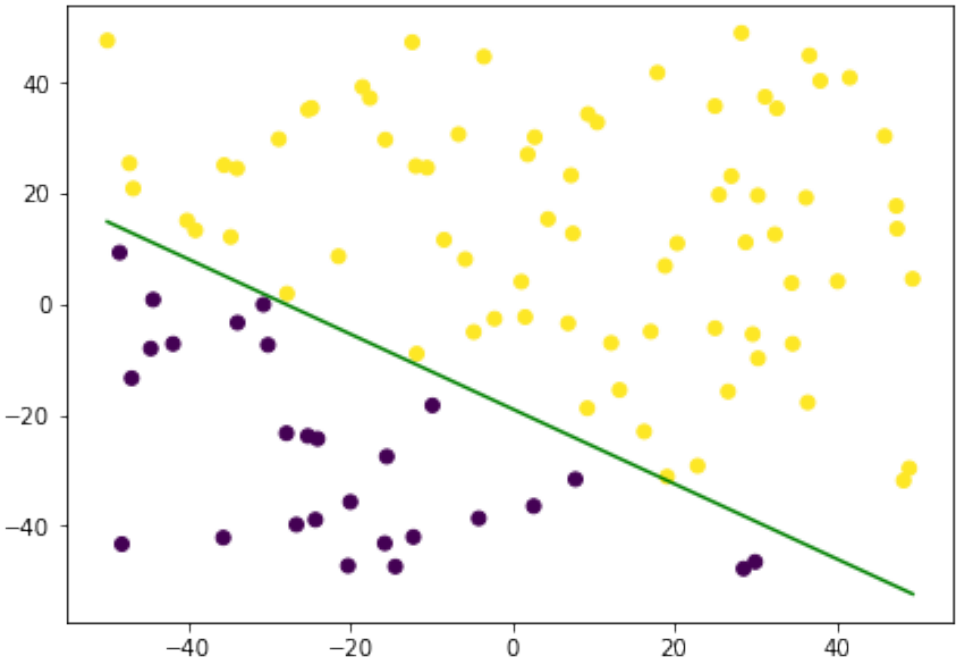


En segundo lugar utilizamos *simula_gauss(N, dim, sigma)* que genera una lista de N vectores de dimensión dim con componentes tomadas de una distribución gaussiana de media 0 y varianza dada por cada posición de sigma. Usamos $N = 50$, dim$= 2$ y sigma$= [5, 7]$

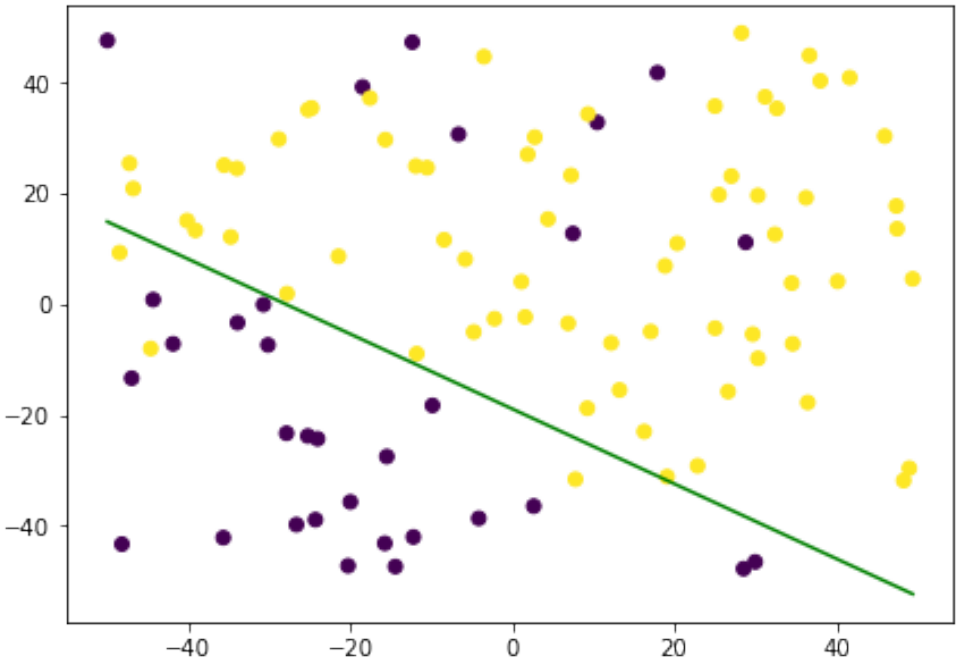


2.1. Influencia del ruido en la selección de la complejidad

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(100, 2, [50, 50])* generamos 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()*. Mostramos a continuación el resultado de dicho etiquetado junto con la recta usada para etiquetar:



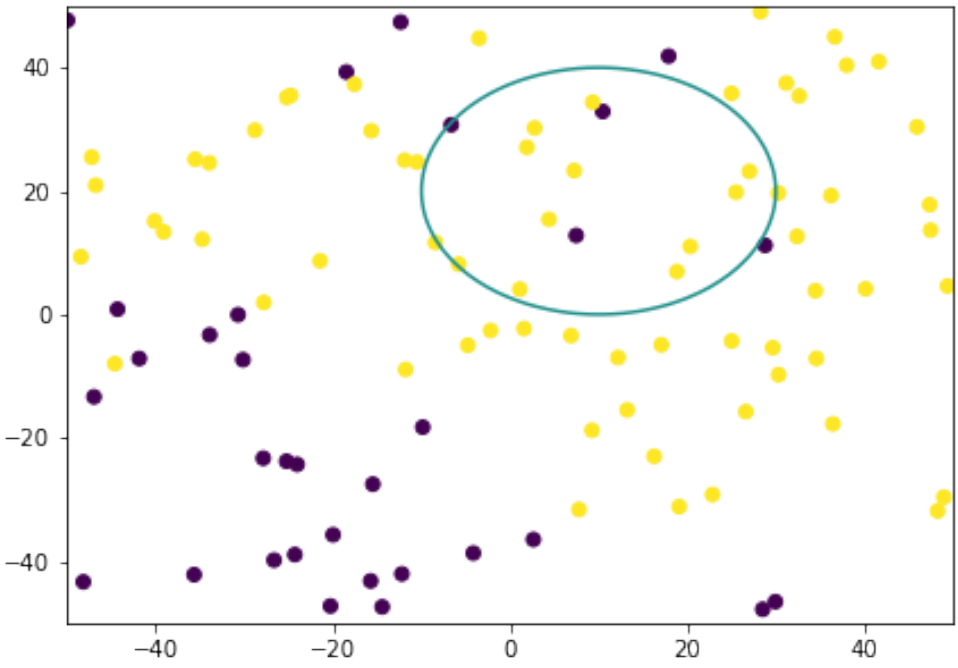
En efecto por la propia forma de etiquetar, todos los puntos estan correctamente clasificados. Ahora introduciremos un 10 por ciento de ruido en cada clase cambiando algunas etiquetas elegidas aleatoriamente y observamos que ahora hay elementos mal clasificados, de hecho si calculamos le exactitud(accuracy) de clasificación observamos que como era esperable siempre va a estar acotada a causa de dicho porcentaje de ruido:



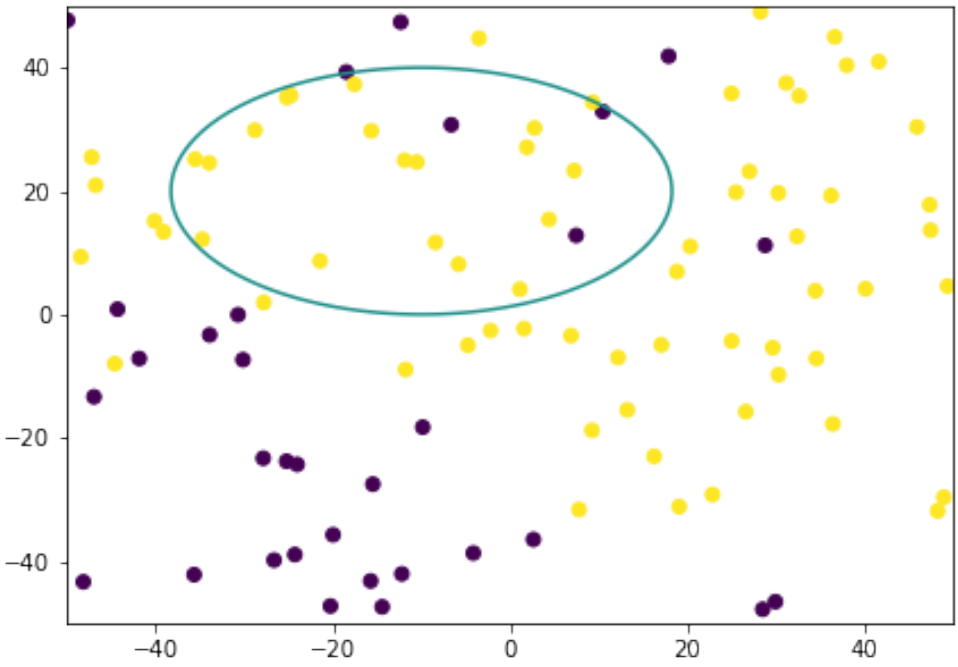
$$Accuracy = 0.89$$

A continuación vamos a considerar varias funciones más complejas como fronteras de decisión para el problema de clasificación, veremos como se comportan con respecto a nuestros datos. En particular se trata de 2 elipses, una hipérbola y una parábola, dibujaremos dichas fronteras y calcularemos su puntuación de Accuracy en nuestros datos.

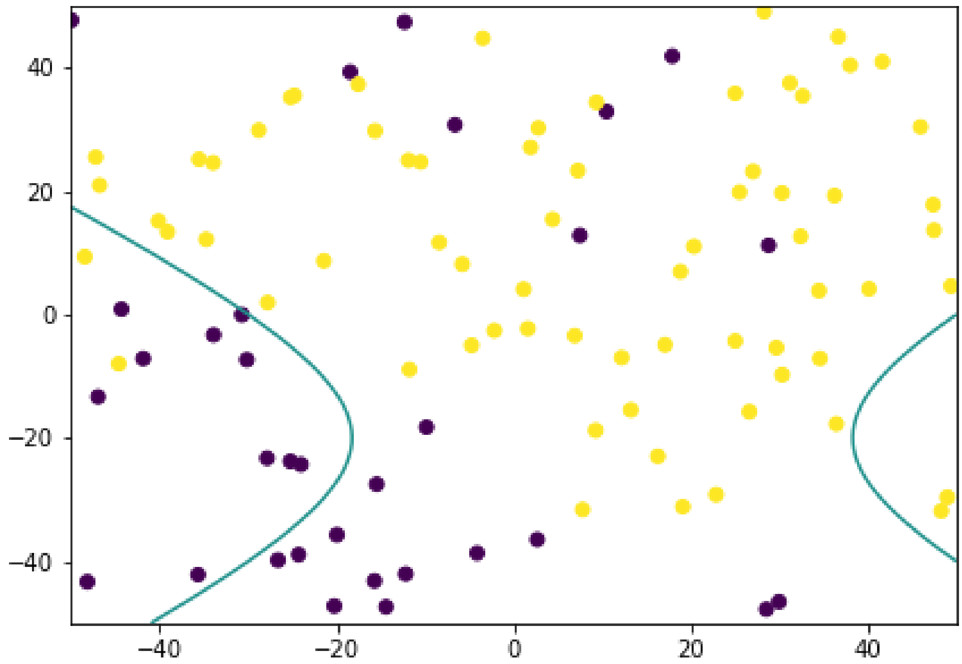
- $f_1(x, y) = (x - 10)^2 + (y - 20)^2 - 400$ $\qquad$ $Accuracy = 0.6$



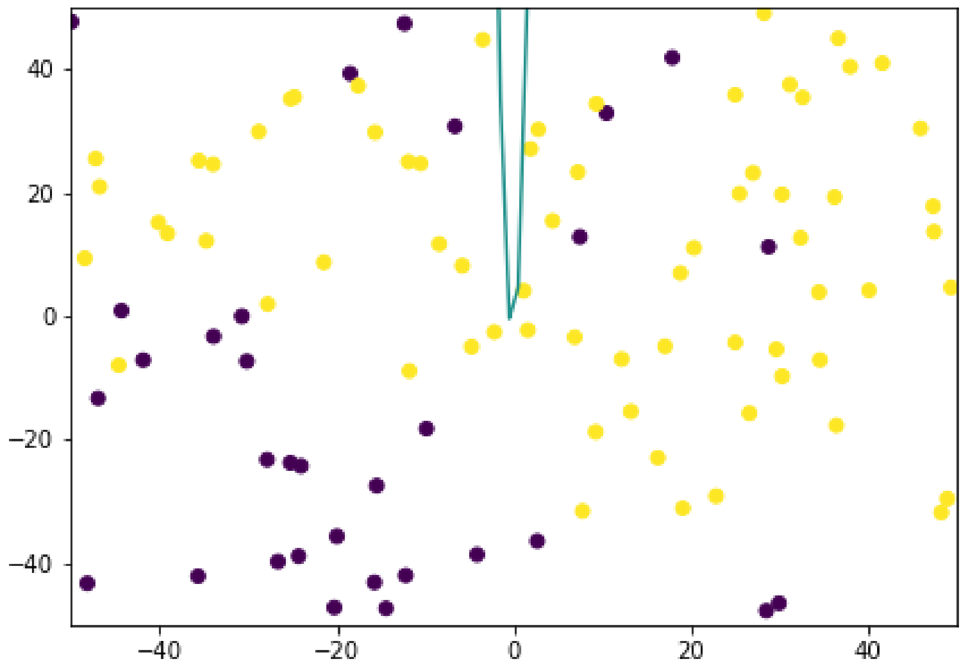
- $f_2(x, y) = 0.5(x + 10)^2 + (y - 20)^2 - 400$ $\qquad$ $Accuracy = 0.52$



- $f_3(x, y) = 0.5(x - 10)^2 - (y + 20)^2 - 400$ $\qquad Accuracy = 0.25$



- $f_4(x, y) = y - 20x^2 - 5x + 3$ $\qquad Accuracy = 0.32$

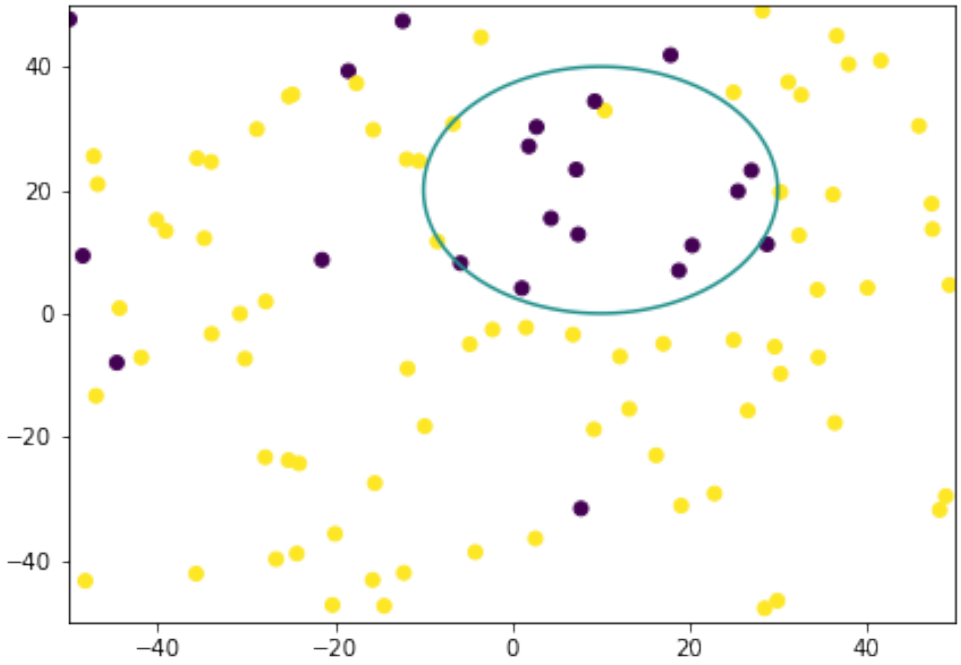


En primer lugar podemos observar que ninguna se acerca al porcentaje de acierto proporcionado por la recta original(lo cuál es lógico porque hemos etiquetado con dicha recta y despues introducido el ruido) pese a tratarse de modelos más complicados. De hecho tampoco mejoran apenas un clasificador aleatorio que obtendría una puntuación promedio de 0.5.
También cabe reseñar que en estas fronteras no se han utilizado los datos para aprender, luego de base dependemos de que casualmente las funciones se ajusten bien a esos datos.

Como conclusiones obtenemos que no es posible sortear el ruido, a la hora de ajustar un modelo este siempre será un impedimento. Por otro lado también se observa que el uso de modelos más complejos no siempre garantiza la mejora en prestaciones, de hecho se sabe que dado un dataset de entrenamiento con un cierto tamaño, si aumentamos la complejidad del modelo, su capacidad de generalización tiende a disminuir .
Finalmente se concluye que si se usara un modelo muy complejo que ajustara muy bien los datos, su capacidad de generalización sería muy mala, es por esto que no debemos dejar que el ruido nos lleve a pensar que la función objetivo es más compleja de lo que en realidad es.

Como experimento final, si etiquetamos los datos con alguna de las fronteras definidas por las funciones dadas e introducimos el ruido despues, se obtiene de nuevo una puntuación que está acotada por el porcentaje de ruido estocástico, lo mostramos por ejemplo con una de las elipses:

$f_1(x, y) = (x - 10)^2 + (y - 20)^2 - 400$ $\qquad\qquad$ $Accuracy = 0.89$



La conclusión es la misma que con la recta, el ruido no se puede aprender, debemos además evitar que la presencia de este nos influya a la hora de tomar decisiones que impliquen un aumento de complejidad.

3. Modelos lineales

En esta sección vamos a implementar 2 modelos lineales que junto a los de la primera práctica cubren las opciones más básicas para el problema de aprendizaje.

3.1. Algoritmo Perceptrón PLA

El algoritmo Perceptrón, en inglés *Perceptron Learning Algorithm* es un algoritmo muy sencillo que trata de obtener un hiperplano que separe conjuntos de datos en el problema de clasificación binaria. Para ello va comprobando si la predicción del hiperplano que se tiene en ese momento es correcta para un dato y en caso contrario lo actualiza moviendo el hiperplano para que mejore. Se sabe que si los datos son linealmente separables, el algoritmo convergerá en algún momento al hiperplano que los separa.
Atendiendo a estas características obtenemos el siguiente pseudocódigo:

PLA(datos, etiquetas, max_iter, wInicial)

$niteraciones$=0
W=Winicial
Mientras no se alcanze el máximo de iteraciones y w!=w_{old} :

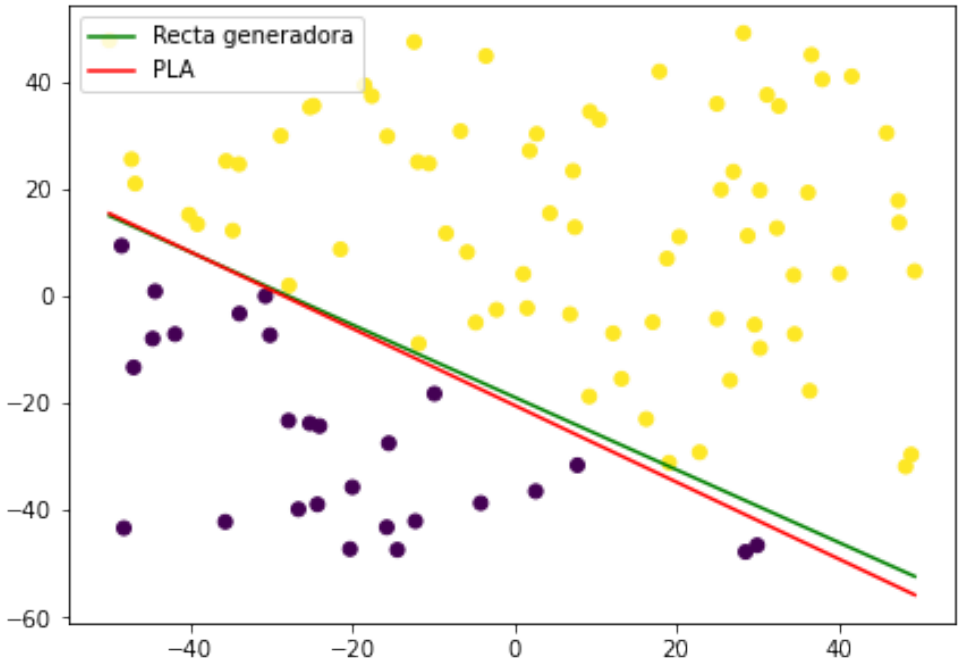
$$w_{old} = w$$

Para cada dato de entrenamiento (x, y) $\in (\mathcal{D}atos, etiquetas)$
Si signo($w^T x$) != y entonces w+=yx

$Iteraciones++$
Fin-Mientras
Devolver W

Vamos a utilizarlo con los datos del primer ejercicio sin ruido, por construcción son claramente linealmente separables, Probaremos distintas inicializaciones para el vector de pesos y estudiaremos cuantas iteraciones tarda en converger, entendiendo por iteración una época, es decir un recorrido completo de los datos. La naturaleza del algoritmo proporciona una clasificación perfecta en este caso de clara separabilidad.

Inicializamos en primer lugar con un vector de ceros, obteniendo el siguiente resultado con convergencia en 75 iteraciones w = [661 , 23.20241712, 32.39163606]:

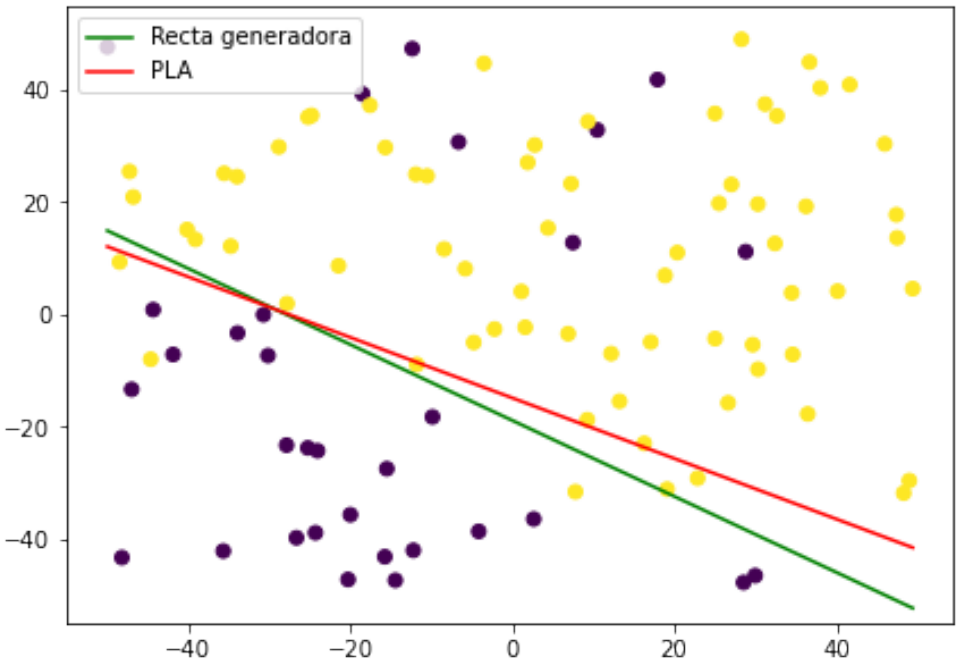


Y ahora llevaremos a cabo inicializaciones aleatorias en el intervalo $[0, 1]$ para los pesos y haremos 10 experimentos, mostramos los resultados en la siguiente tabla:

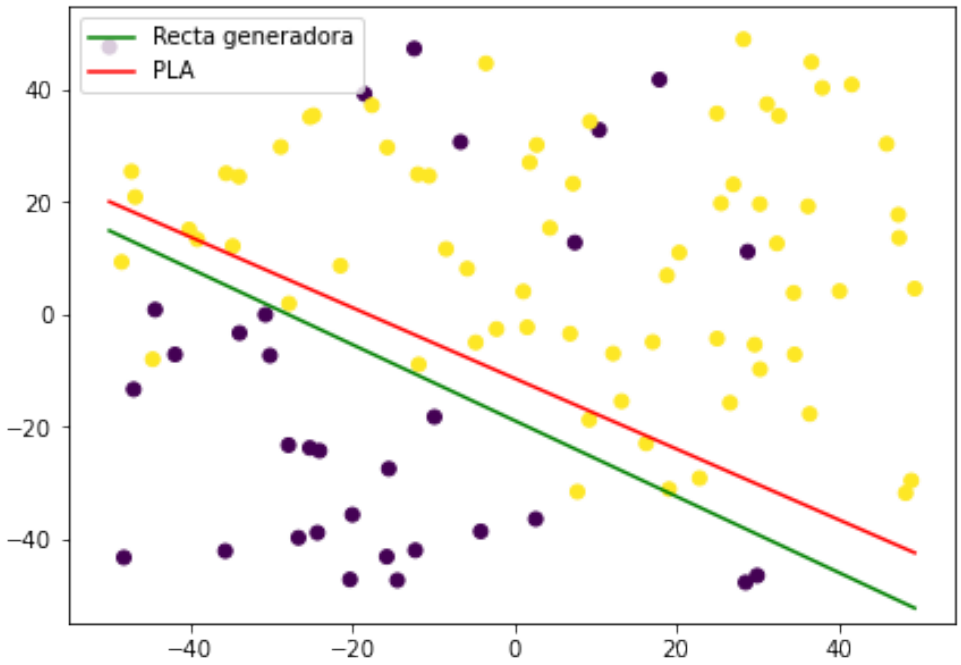
Iteraciones	W[0] Inicial	W[1] Inicial	W[2] Inicial	W[0] Final	W[1] Final	W[2] Final
60	0.619	0.010	0.539	555.619	19.401	29.254
248	0.003	0.951	0.905	1118.003	39.141	59.613
43	0.796	0.915	0.146	458.796	15.332	23.838
72	0.158	0.188	0.622	609.158	24.124	33.260
129	0.906	0.990	0.711	831.906	32.422	46.454
244	0.732	0.909	0.401	1087.732	39.491	53.521
70	0.250	0.173	0.119	641.250	22.452	30.991
84	0.813	0.147	0.264	663.813	22.118	30.864
122	0.819	0.311	0.982	826.819	31.220	45.127
37	0.267	0.534	0.314	383.267	14.432	17.531

Observamos que en promedio el número de iteraciones son 110.9 lo que nos muestra al compararlo con las 75 de la inicialización nula que es posible que inicializar a 0 los pesos sea una buena heurística de inicio para el algoritmo. De igual forma podemos comprobar a simple vista en la tabla sin un ánalisis en detalle que existe una gran variabilidad en el número de iteraciones necesarias para converger, por tanto el peso inicial tiene cierta importancia en el tiempo de convergencia.
Como en todas las ejecuciones se ha parado por el criterio de convergencia, la tasa de clasificación es siempre del 100 por 100 en este caso como se esperaba.

Pasamos ahora a repetir el experimento con las etiquetas del primer ejercicio pero esta vez ya con un 10 por ciento de ruido introducido, lo que es esperable es que los nuevos datos no sean linealmente separables y por tanto el algoritmo no consiga converger, terminando por el criterio del máximo de iteraciones, además puede ser que la solución devuelta no sea la mejor por la que haya pasado el algoritmo en algún momento.
Inicializamos en primer lugar con un vector de ceros y un límite de 400 iteraciones, vemos que no converge y obtiene un 86 por ciento de acierto con la recta dada por w = [660 , 23.95520539, 44.36098373], de nuevo estamos acotados por el porcentaje de ruido:



Para comprobar que al no ser linealmente separables los datos, el algoritmo no converge y oscila, mostraré una ejecución con más iteraciones. En este caso, lanzo 800 iteraciones con la misma inicialización y de hecho, el resultado es peor que con 400 iteraciones y la frontera cambia, se obtiene un 83 por ciento de acierto:



Esto muestra los problemas de esta versión del algoritmo en la que al pasar por cada punto se mejora para ese punto, pero se puede empeorar en otros. Lo mejoraremos en el apartado de Bonus con el algoritmo Pocket, guardando siempre en memoria la mejor solución por la que hayamos pasado.
Ahora llevaremos a cabo inicializaciones aleatorias en el intervalo $[0, 1]$ para los pesos y haremos 10 experimentos, mostramos los resultados en la siguiente tabla:

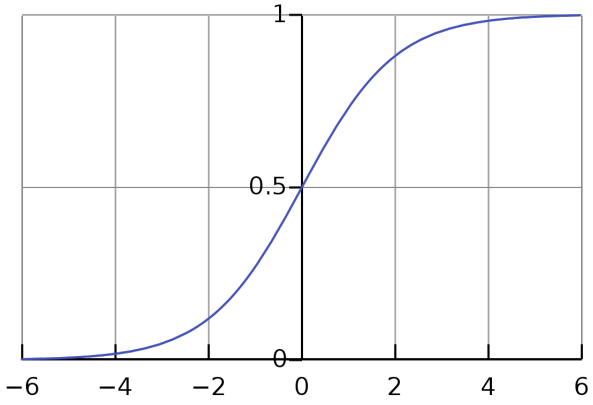
Iteraciones	Acierto %	$W_{ini}[0]$	$W_{ini}[1]$	$W_{ini}[2]$	$W_{fin}[0]$	$W_{fin}[1]$	$W_{fin}[2]$
400	0.84	0.911	0.367	0.434	662.911	33.835	45.568
400	0.83	0.512	0.939	0.031	653.512	32.794	65.750
400	0.81	0.717	0.891	0.027	664.717	12.467	53.006
400	0.83	0.522	0.326	0.859	649.522	39.809	71.358
400	0.84	0.559	0.690	0.453	655.559	25.074	49.723
400	0.8	0.628	0.290	0.009	647.628	14.847	56.853
400	0.89	0.577	0.311	0.517	650.577	24.873	33.239
400	0.83	0.916	0.426	0.247	644.916	32.780	42.404
400	0.88	0.371	0.932	0.937	655.371	20.066	33.027
400	0.87	0.844	0.920	0.228	649.844	16.270	37.787

Finalizamos esta sección comentando brevemente estos resultados. De nuevo reafirmamos la no convergencia del algoritmo con estos datos obteniendose un promedio de 84,2 % para la tasa de acierto en clasificación. En la séptima iteración obtenemos casualmente una frontera tremendamente similar a la recta que se usó en el primer ejercicio para generar los datos, pero como tenemos el ruido introducido, este 89 por ciento es la mejor tasa de acierto posible, pues es la misma que lograba obtener la propia recta generadora de etiquetas despues de introducirle el ruido aleatorio en el ejercicio 1.

3.2. Regresión Logística RL

Este método es un modelo lineal que se utiliza para clasificación y obtiene como salida un número entre 0 y 1 que puede entenderse como probabilidad de que un dato pertenezca a una clase, se establece un umbral normalmente en 0.5 a partir del cúal se determina la clase del dato. Se basa en el uso de una función sigmoidal, en este caso basada en la exponencial, que suaviza la transición entre las 2 clases, la cual es muy abrupta en la función signo.

$$\sigma(w^T x) = \frac{e^{w^T x}}{1 + w^{w^T x}}$$



Al ser un método basado en regresión, trataremos de minimizar una función de error denominada error de entropía cruzada con la siguiente expresión obtenida a través del criterio de máxima verosimilitud:

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

Para ello calculamos su gradiente y lo usaremos en el algoritmo de Gradiente Descendente Estocástico que implementamos en la primera práctica.

$$\nabla E_{in}(w) = \frac{1}{N}\sum_{i=1}^{N} -y_i x_i \frac{e^{-y_i w^T x_i}}{1 + e^{-y_i w^T x_i}}$$

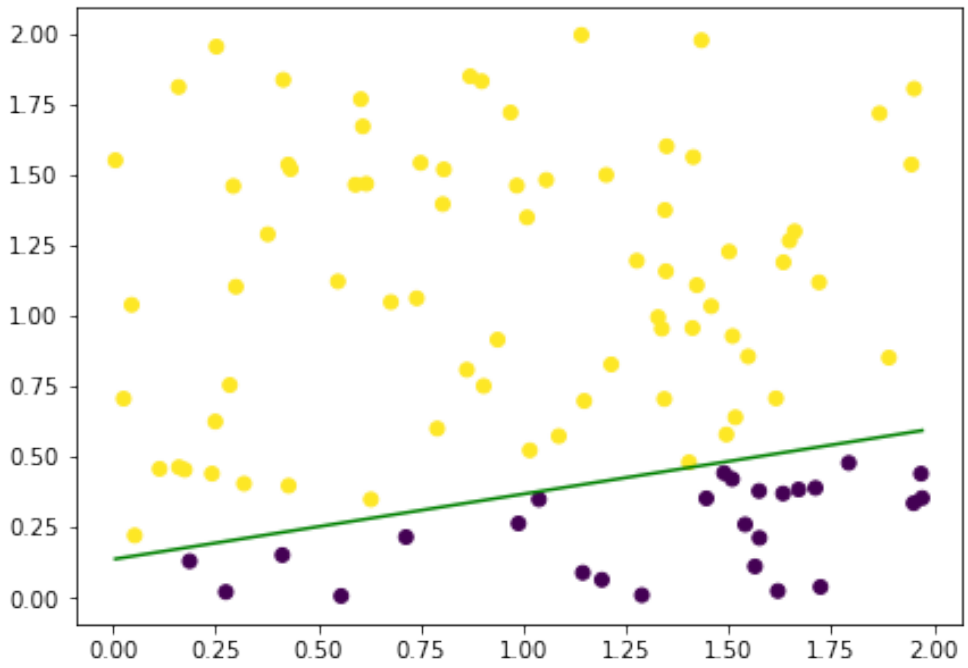
Necesitamos modificar ligeramente el algoritmo, vamos a tener 2 criterios de parada, por un lado un máximo de iteraciones y por otro pararemos si la variación de los pesos trás el transcurso de una época es mínima, en particular si:

$$||w_{t+1} - w_T|| < 0.01$$

Para ello necesitamos almacenar al en cada época el vector de pesos anterior y hacer la comprobación. Finalmente debemos escoger un número adecuado para la tasa de aprendizaje η y el tamaño de minibatch, esta elección se hará de forma empírica probando con distintos valores en el siguiente experimento. Recordamos que el algoritmo de gradiente descendente estocástico se basa en la siguiente regla de actualización:

$$w_j = w_j - \eta \frac{\partial E_{in}(w)}{\partial w_j}$$

En este ejercicio emplearemos nuestra propia función objetivo f y un conjunto de datos $\mathcal{D}$ para ver cómo funciona regresión logística. Consideraremos $X = [0, 2] \times [0, 2]$ con probabilidad uniforme de elegir cada $x \in X$. Elegiremos una línea en el plano que pase por X como la frontera que separa la región en donde y toma valores +1 y -1. Para ello, seleccionaremos dos puntos aleatorios de X y usamos la que pasa por ambos. Este es el gráfico que muestra dicho conjunto de 100 puntos que usaremos para entrenar el modelo:

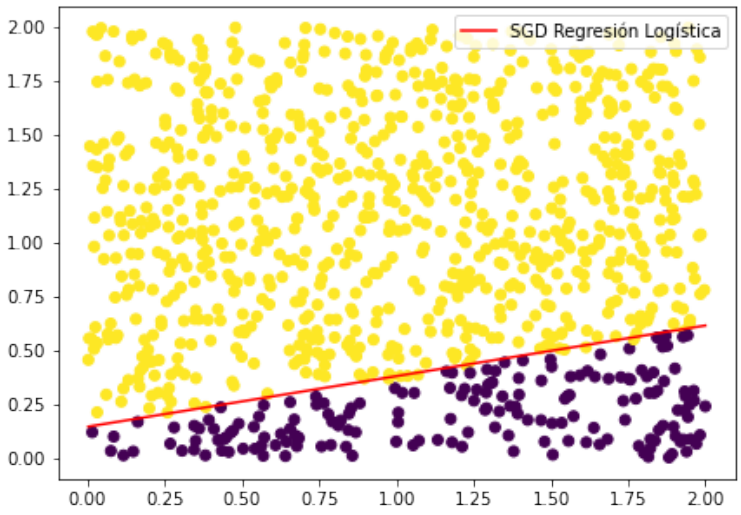


Como conjunto de test se utilizará una nueva muestra de 1000 datos también etiquetados de la misma forma (según la misma recta). Trás hacer ciertas simulaciones con distintas tasas de aprendizaje y tamaños de batch, los rangos de valores mostrados en la siguiente tabla parecen los más prometedores por su error y número de iteraciones necesarias para estabilizarse los pesos:

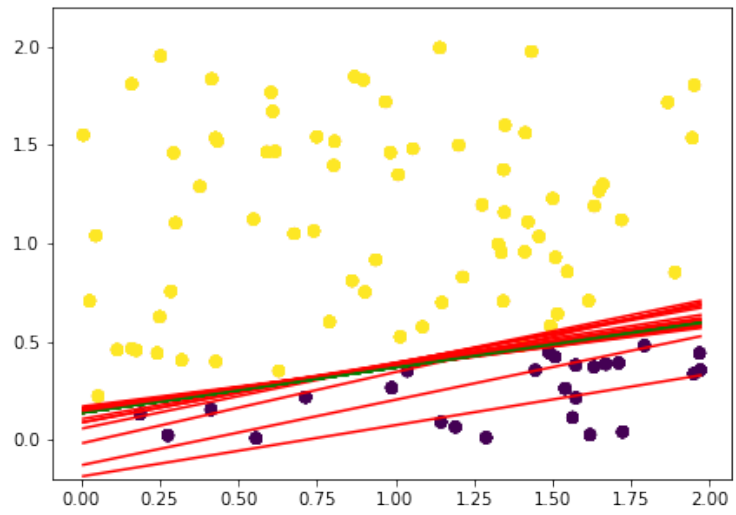
Learning Rate	Batch Size	Épocas	E_{in}	$Accuracy_{in}$	E_{test}	$Accuracy_{test}$
0.01	1	385	0.0906	0.99	0.0942	0.975
0.01	8	136	0.3014	0.9	0.2618	0.922
0.01	16	56	0.4539	0.74	0.3870	0.826
0.01	100	1	0.6908	0.74	0.6901	0.826
0.1	1	993	0.0211	1	0.0316	0.991
0.1	8	422	0.0779	0.99	0.0832	0.976
0.1	16	296	0.1171	0.98	0.1165	0.973
0.1	100	109	0.3532	0.81	0.3019	0.878
1	1	2591	0.0030	1	0.0145	0.994
1	8	1103	0.0176	1	0.0282	0.991
1	16	770	0.0280	0.99	0.0390	0.989
1	100	384	0.0907	0.99	0.0944	0.975
10	1	1184	0.0003	1	0.0195	0.991
10	8	2231	0.0026	1	0.0144	0.994
10	16	2010	0.0046	1	0.0153	0.995
10	100	984	0.0211	1	0.0320	0.991

Si analizamos estos resultados, obtenemos que en general para todas las tasas de aprendizaje, cuanto mayor sea el tamaño del minibatch, menos iteraciones realiza el algoritmo, pero los resultados en cuanto a error son más pobres. Por tanto, podemos establecer un criterio que escoja un tamaño de minibatch lo más pequeño posible siempre que los tiempos de ejecución sean razonables. En el caso de nuestro problema no es conflictivo pues estamos en pequeña escala, pero puede ser que con más datos la necesidad de iterar más produzca un importante coste computacional.
Elegido este parámetro, pasamos a estudiar el comportamiento con respecto a las distintas tasas de aprendizaje. Se observa que en este problema cuando aumentamos la tasa, se obtienen mejores resultados para E_{in} y E_{test} y tasas de clasificación cada vez más perfectas, la cantidad de iteraciones también va aumentando, estableciendo de nuevo el compromiso eficacia-coste computacional, pero en nuestro problema con minibatches unitarios, incluso usando tasas muy altas (no será lo habitual) el rendimiento es bueno.
Nuestra conclusión es que la elección de batches muy pequeños obtiene buenos resultados y, en el caso de nuestro problema podemos usar tasas de aprendizaje altas. Sin embargo, parece que la elección de $\eta = 0.1$ que sería mas extrapolable a un caso general también consigue buenos resultados (tasa de clasificación muy alta).

Mostramos el gran acierto en test para los parámetros $\eta = 0.1$ y BatchSize=1:



Y aquí los resultados para todas las elecciones de los parámetros, algunas se descartan rápidamente, para otras se ha tenido que comparar resultados numéricos:



3.2.1. Experimento con $\eta = 10$ y BatchSize=1

Vamos a repetir esta ejecución 100 veces para obtener conclusiones más fiables para este problema, los resultados han sido los siguientes en valor promedio:

Épocas	E_{in}	E_{test}	$Accuracy_{in}$	$Accuracy_{test}$
1796.14	0.00072	0.01373	100	0.99542

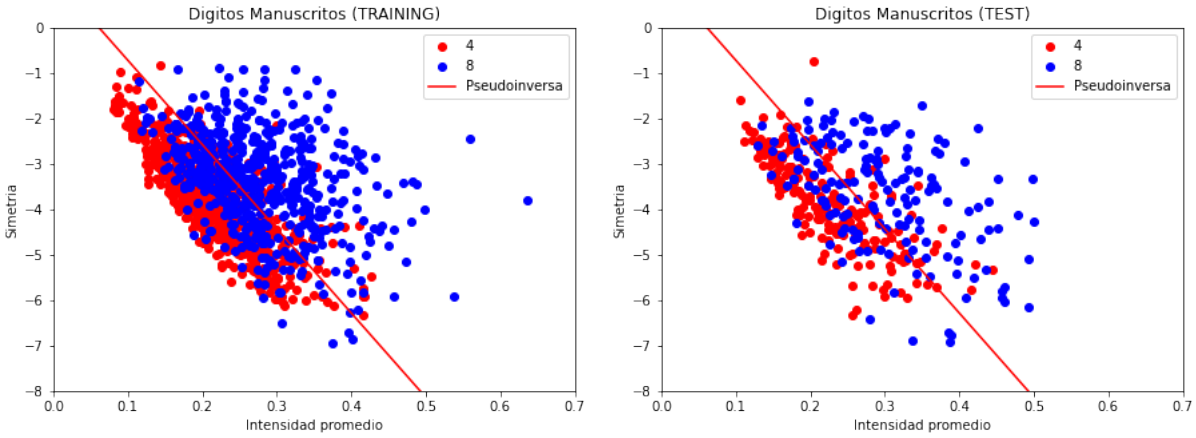
Confirman lo mostrado en la tabla anterior, son buenas elecciones y llevan a cabo una generalización correcta para este problema. Debemos comentar que la ejecución de este experimento es bastante costosa pero aporta más fiablidad que una única ejecución.

4. Aplicación y comparación : Clasificación de dígitos

En esta sección vamos a utilizar los modelos implementados en las dos primeras prácticas en un caso concreto de clasificación. Usaremos un conjunto de datos basado en imágenes de dígitos con 2 características (intensidad y simetría) y el dígito que representan como etiqueta. En particular se tratará de separar el 4 del 8, teniendo el 4 la etiqueta -1 y el 8 la +1. Disponemos también de un conjunto de ejemplos disponibles para testear la desenvoltura de los distintos modelos.
Como se trata de un problema de clasificación usaremos principalmente sus propias métricas para valorar los resultados, hemos implementado funciones para obtener la tasa de acierto o *accuracy* y para la matriz de confusión, la cual representa en su diagonal las instancias correctamente clasificadas de cada clase, y en la antidiagonal las clasificaciones incorrectas de los 2 tipos posibles, en nuestro problema falso 4 y falso 8. Como en varios de los métodos se usa regresión como paso previo para obtener la frontera de clasificación también tomaremos en cuenta el error de regresión.

4.1. Regresión lineal

Utilizaremos el algoritmo de pseudoinversa por simplicidad y optimalidad en el problema de regresión, tenemos el código implementado de la práctica anterior, por ello no daré mas detalles del mismo. Estas gráficas muestran la recta obtenida junto con los datos de entrenamiento y test respectivamente:



Los resultados obtenidos son los siguientes:

$E_{in} = 0.6428$ $E_{test} = 0.7087$ $Accuracy_{in}$: 0.7722 $Accuracy_{test}$: 0.7486

Y esta la matriz de confusión del conjunto de test:

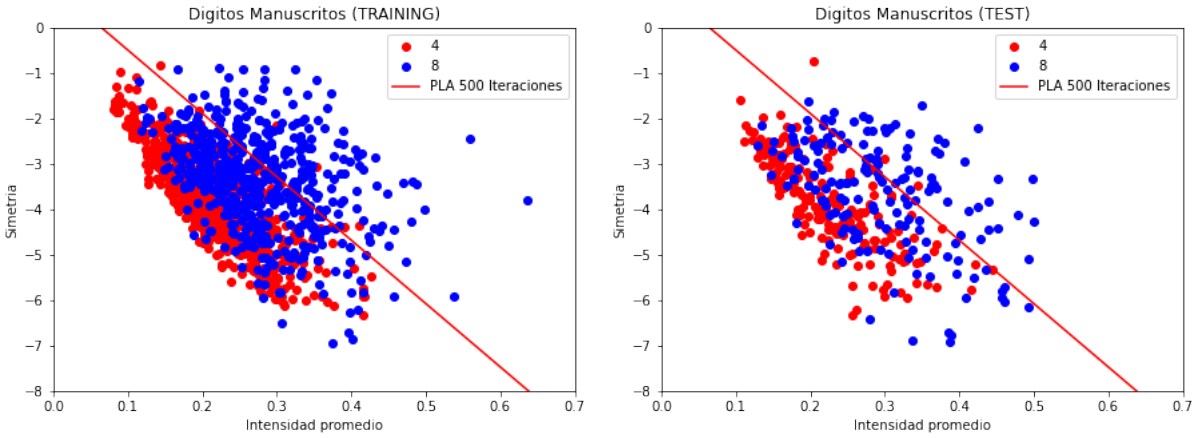
	Etiqueta	
	8	4
8	107	33
4	59	167

Observamos que se obtiene un buen punto de partida, los resultados no son especialmente buenos pero es un problema complicado porque si miramos la gráfica en la parte central los datos de las dos clases estan muy entrelazados, a pesar de ello el modelo parece haber captado la tendencia general obteniendo un 75 por ciento de acierto en test. Por otra parte la matriz de confusión nos muestra que el modelo entrenado es mas o menos igual de fiable a la hora de reconocer un 8 o un 4, en ambos casos cuando hace una predicción acierta en un 76 y 74 por ciento de los casos respectivamente. Donde sí se aprecia diferencia es en cuanto al porcentaje

de acierto en cada clase, de todos los 8 acierta el 64 por ciento mientras que de los 4 acierta el 83 por ciento . Esto último se puede intuir mirando la gráfica.

4.2. PLA

Usamos el algoritmo Perceptrón, que sabemos que como los datos no son linealmente separables no convergerá y al ser la versión primitiva, el resultado puede no ser el mejor por el que haya pasado el algoritmo.



Los resultados obtenidos son los siguientes:

$$Accuracy_{in} : 0.6909 \qquad Accuracy_{test} : 0.6994$$

Y esta la matriz de confusión del conjunto de test:

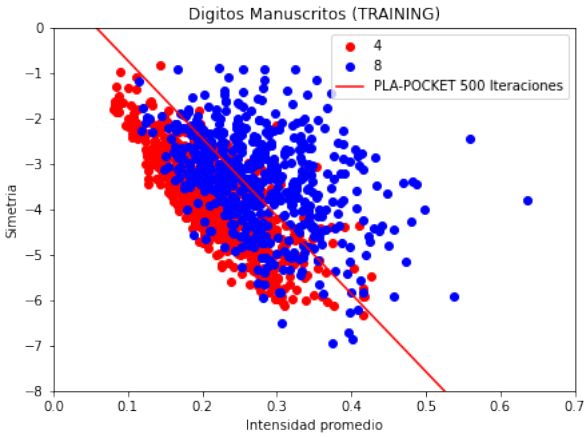
	Etiqueta	
	8	4
8	59	3
4	107	197

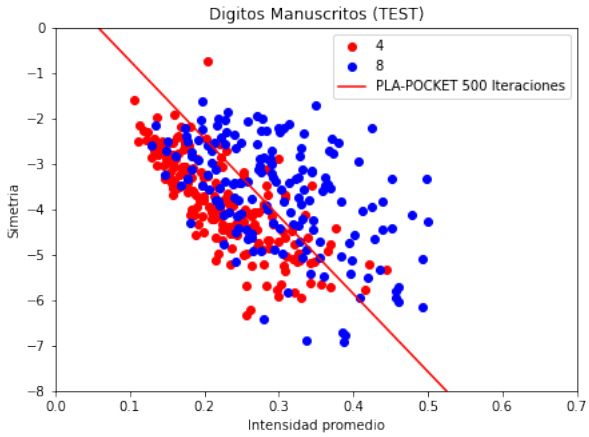
Como vemos los resultados son muy mediocres por todas las carencias que ya hemos comentado del algoritmo, los mejoraremos con el algoritmo Pocket en la siguiente sección. En cuanto a la matriz de confusión destaca que el modelo reconoce prácticamente todos los 4 pero falla bastante con los 8, esto es apreciable en la propia frontera de decisión que hemos graficado.

4.3. PLA POCKET

Para esta sección se ha implementado el algoritmo POCKET usando como plantilla PLA. La única pero importante diferencia es que al finalizar cada época se evalua la solución de ese momento y se evalua su error, si ha disminuido se guardan en memoria dichos pesos para devolverlos al final en caso de que no se mejoren. Así evitamos los problemas de la versión anterior y mejoramos los resultados. A cambio tenemos un coste en complejidad debido a la necesidad de computar el error con cada vector de pesos resultante de cada iteración.

Pasamos a ejecutarlo sobre los datos:





Los resultados obtenidos son los siguientes:

$Accuracy_{in}$: 0.7714 $Accuracy_{test}$: 0.7541

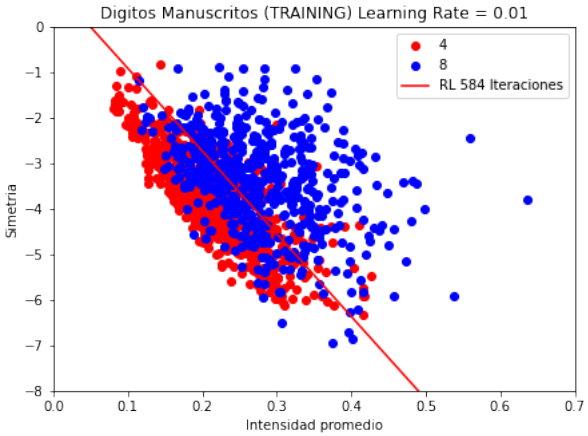
Y esta la matriz de confusión del conjunto de test:

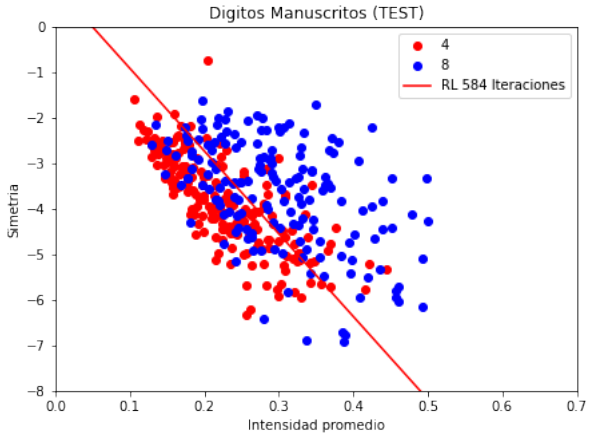
	Etiqueta	
	8	4
8	97	21
4	69	179

Comprobamos que el resultado es muy bueno en comparación con PLA simple y prácticamente iguala a regresión lineal. De los errores que muestra la matriz se infiere que el modelo continua reconociendo correctamente la mayoría de 4 y mejora algo con los 8.

4.4. Regresión Logística

Aplicando ahora el algoritmo SGD con las funciones asociadas al error de entropía cruzada. Hemos probado con varias tasas de aprendizaje y hemos encontrado que los mejores resultados los conseguimos con $\eta = 0.01$ Se obtiene el siguiente resultado:





Los resultados obtenidos son los siguientes:

$E_{in} = 0.4637$ $E_{test} = 0.5258$ $Accuracy_{in}$: 0.7780 $Accuracy_{test}$: 0.7486

Y esta la matriz de confusión del conjunto de test:

	Etiqueta	
	8	4
8	113	39
4	53	161

Concluimos que los resultados en cuanto a tasa de clasificación son muy similares a regresión lineal, muy ligeramente mejores al menos en test. El análisis de la matriz de confusión llevado a cabo para regresión lineal es válido aquí pues los valores arrojados son muy similares.

4.5. Uso de pesos iniciales de regresión lineal

Mostramos ahora los resultados de inicializar los algoritmos iterativos más costosos (PLA, POCKET y Regresión Logística) con los pesos que obtiene la pseudoinversa en regresión lineal. Para poder comparar con sentido, vamos a usar en todos los casos el error entendido como tasa de error en clasificación:

Algoritmo	E_{in}	E_{out}	Iteraciones
PLA	0.704	0.702	500
POCKET	0.774	0.746	20
RL	0.768	0.751	563

Comparando los resultados no se observan grandes mejoras en cuanto al error, quizás una leve mejora para el algoritmo de regresión logística pero no relevante porque pùede deberse al factor aleatorio del algoritmo. Donde si se comprueba una importante mejora es en el número de iteraciones para el algoritmo POCKET. Haciendo pruebas hemos comprobado que con sólo 20 épocas ya obtenemos resultados igual de buenos que inicializando los pesos a 0 con muchas más épocas. Para regresión logística también hay una leve mejora en este sentido. Concluimos que para tamaños de muestra similares al usado es una buena estrategia partir con los pesos proporcionados por el algoritmo de pseudoinversa ya que obtiene buenos resultados y mejora los tiempos de cómputo.

4.6. Cota del error de Hoeffding

Hemos estudiado en teoría que la base de la teoría de aprendizaje PAC (*Probably Aproximately Correct*) es la desigualdad de Hoeffding que nos garantiza el aprendizaje:

$$P[|E_{out} - E_{in}| > \epsilon] \leq 2e^{-2\epsilon^2 N}$$

A partir de ella si establecemos un nivel de tolerancia δ podemos establecer otra desigualdad teórica que afirma que la siguiente cota para E_{out} es cierta con probabilidad $1 - \delta$:

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

Si la utilizamos con E_{test} en lugar de E_{in} conseguimos que el tamaño de la clase de funciones sea 1, mejorando la cota. Para nuestros algoritmos se obtienen las siguientes:

	Regresión lineal	PLA	POCKET	Regresión Logística
Cota de Hoeffding	0.2907	0.3398	0.2852	0.2906

Observamos que establecen una buena frontera asegurando cierto nivel de aprendizaje y generalización. Si tuvieramos más datos, la cota tendería al error dentro de la muestra.

4.7. Cota del error de Vapnik-Chervonenkis

La dimensión de Vapnik-Chervonenkis refleja el número efectivo de parámetros del modelo, es una abstracción que permite garantizar bajo ciertas condiciones que si la dimensión es finita, el aprendizaje es posible. En nuestro problema, es claro que la dimensión es $d_{VC} = 3$ porque ajustamos 3 pesos. La siguiente desigualdad establece otra cota para el error en función del error en la muestra con probabilidad $1 - \delta$:

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

Tomando $\delta = 0.05$ obtenemos las siguientes cotas de error que, como breve análisis son laxas y poco informativas pero garantizarían un mejor aprendizaje si tuvieramos más datos.

	Regresión lineal	PLA	POCKET	Regresión Logística
Cota de Vapnik-Chervonenkis	0.6587	0.7399	0.6596	0.6528

Para el cálculo de estas cotas se ha utilizado el error de clasificación.

5. Bibliografía

Fuentes de consulta on-line, en general documentación de las librerías:

1. Numpy https://numpy.org/doc/stable/reference/generated/numpy.hstack.html

2. https://www.tutorialspoint.com/matplotlib/matplotlib_contour_plot.html

3. https://numython.github.io/posts/2016/02/graficas-de-contorno-en-matplotlib/