APRENDIZAJE AUTOMÁTICO

PRÁCTICAS - PRÁCTICA 2

Pedro Gallego López

Doble Grado de Ingeniería Informática y Matemáticas

Universidad de Granada
Escuela Técnica Superior de Ingeniería Informática y Telecomunicaciones

28 de abril de 2021

Práctica 2

Introducción

En esta práctica abarcaremos el problema del aprendizaje automático desde el punto de vista de la complejidad de nuestra clase de funciones y el ruido. Nos enfrentaremos a problemas de clasificación en donde se hará uso del algoritmo PLA, PLA pocket y en donde también veremos el problema desde el punto de vista de la regresión logística.

Ejercicio 1. Ejercicio sobre la complejidad de $\mathcal{H}$ y el ruido

Ejercicio 1.1. Dibujar las gráficas con las nubes de puntos simuladas con las siguientes condiciones que siguen.

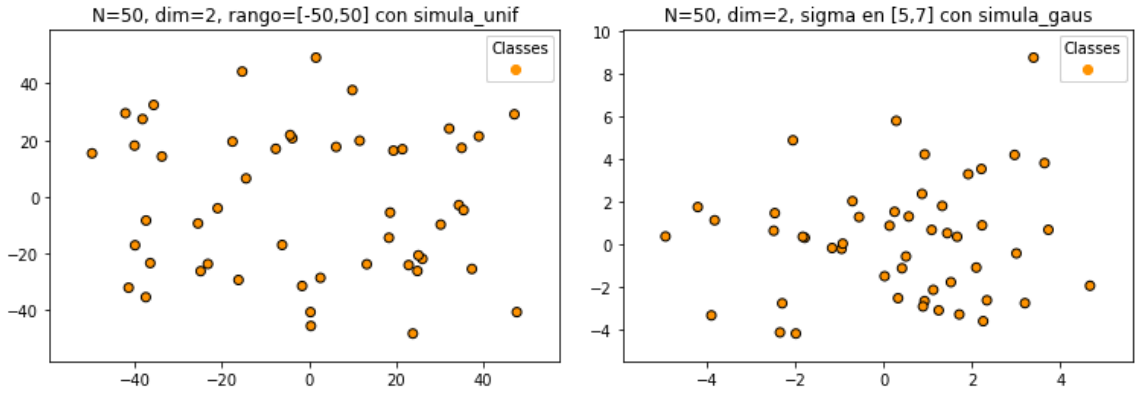


Figura 1.1

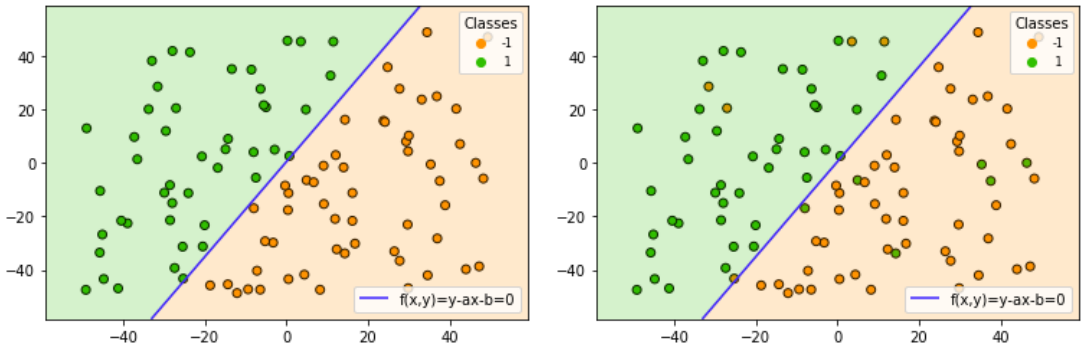
Ejercicio 1.2. Valorar la influencia del ruido.

En este apartado cogeremos una nube de puntos similar a la que se observa en la Figura 1.1 en la parte izquierda, cogeremos 100 puntos en lugar de 100. Utilizaremos una función de la forma

$$f_{a,b}(x, y) = y - ax - b$$

para etiquetar, cogiéndonos el nivel 0, es decir, $f_{a,b}(x, y) = 0$. Los parámetros a y b se obtienen a través de la función *simula_recta*. Obteniendo los valores $a = -0{,}4537$, $b = 12{,}7986$

En la Figura 1.2 podemos ver la nube de puntos que vamos a utilizar, viéndose a la izquierda (1.2a) la nube de puntos etiquetadas en base a $f_{a,b}$ y en la parte derecha (1.2b) la misma nube de puntos donde se le ha añadido un ruido en el etiquetado. Concretamente el ruido afecta al 10 % de las etiquetas.

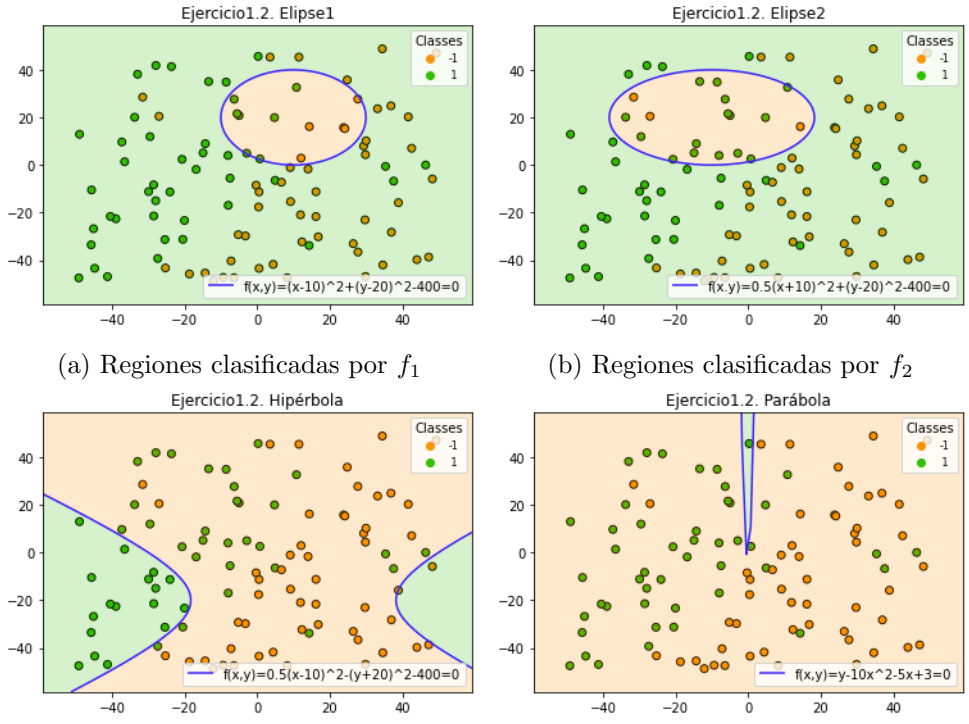


(a) Sin ruido (b) Con ruido

Figura 1.2: 100 puntos de 2 dimensiones distribuidos uniformemente en [-50,50].

Nuestro estudio ahora se basa en comprobar el desempeño de clases de funciones distintas a las rectas escogiendo una hipótesis concreta de la clase de funciones. En este caso usaremos las hipótesis:

- Elipse (Figura 1.3a): $f_1(x, y) = (x - 10)^2 + (y - 20)^2 - 400$
- Elipse (Figura 1.3b): $f_2(x, y) = 0{,}5(x + 10)^2 + (y - 20)^2 - 400$
- Hipérbola (Figura 1.3c): $f_3(x, y) = 0{,}5(x - 10)^2 + (y - 20)^2 - 400$
- Parábola (Figura 1.3d): $f_4(x, y) = 0{,}5(x - 10)^2 + (y - 20)^2 - 400$



(a) Regiones clasificadas por f_1 (b) Regiones clasificadas por f_2

(c) Regiones clasificadas por f_3 (d) Regiones clasificadas por f_4

Figura 1.3

En la Figura 1.3 podemos intuir un muy mal desempeño de estas hipótesis. Pero vamos a cuantificarlo. Para ello vamos a utilizar la gráfica usada para la curva ROC, necesitaremos hacer

uso de las métricas de **recall** y **specificity** para con ello calcular el "*True Positive Rate*"(TPR) y "*False Positive Rate*"(FPR). Además utilizaremos la métrica **accuracy**. Siendo

- TP: True Positives
- FP: False Positives
- TN: True Negatives
- FN: False Negatives
- P=TP+FN: Positives
- N=TN+FP: Negatives

Las métricas anteriores podemos definirlas ahora como:

$$accuracy = \frac{TP + TN}{P + N} \tag{1.1}$$

$$specificity = \frac{TN}{N} \tag{1.2}$$

$$recall = \frac{TP}{P} \tag{1.3}$$

Ahora poniendo "*False Positive Rate*"=*1-specificity* y "*True Positive Rate*"=*recall* podríamos hacer la gráfica. Esta gráfica va a reflejarnos como de cerca esta nuestro modelo de clasificar como lo haría un clasificador aleatorio. El clasificador aleatorio sigue la recta TPR=FPR.

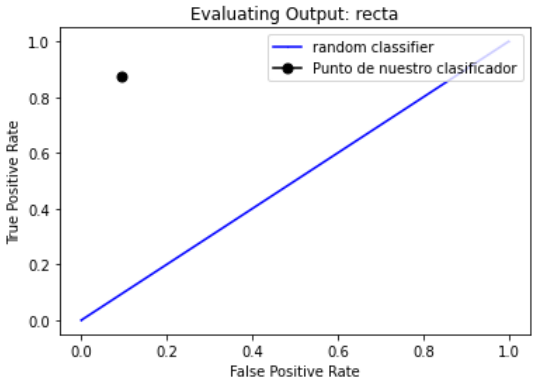
Los resultados del accuracy se pueden observar en la tabla 1.1. Por otro lado, podemos ver lo que dista cada modelo del desempeño de un clasificador aleatorio en las figuras 1.4

Función	**Accuracy**
$f_{a,b}(x,y) = y - ax - b$	0.89
$f_1(x,y) = (x-10)^2 + (y-20)^2 - 400$	0.45
$f_2(x,y) = 0{,}5(x+10)^2 + (y-20)^2 - 400$	0.35
$f_3(x,y) = 0{,}5(x-10)^2 + (y-20)^2 - 400$	0.68
$f_4(x,y) = 0{,}5(x-10)^2 + (y-20)^2 - 400$	0.54

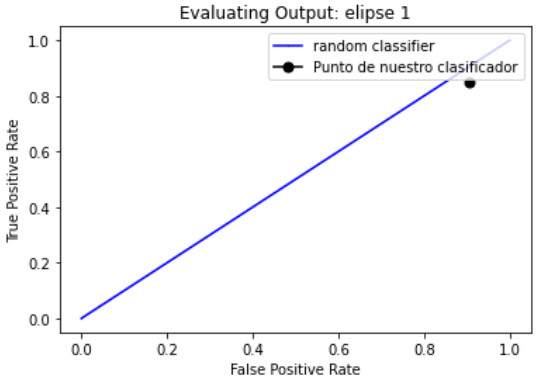
Tabla 1.1: Accuracy obtenidos por las distintas hipótesis

En vista de los accuracy de la Tabla 1.1 se observa como hay un muy mal desempeño en general, salvándose nuestra función $f_{a,b}$ y f_3. Nos fijamos en el accuracy de $f_{a,b}$ que es de 0.89, un resultado próximo al 0.9, algo que cabía esperar que resultase al haber modificado el 10 % de las etiquetas positivas y el 10 % de las negativas. Por otro lado, el accuracy de nuestra hipótesis f_3 es relativamente mayor que los demás, pero dejando aún una cifra bastante baja. Las demás hipótesis son realmente malas, arrojando valores cercanos e incluso por debajo del 0.5.

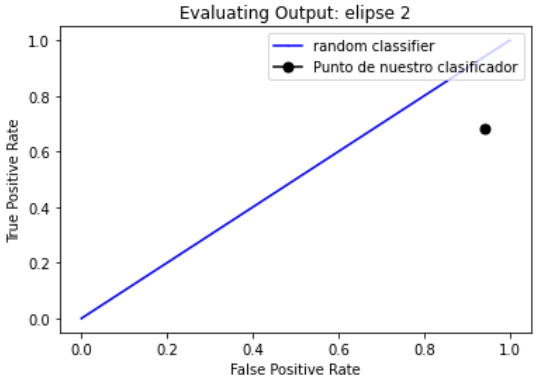
Analizando ahora la relación entre TPR("*True Positive Rate*") y FPR("*False Positive Rate*") mostrado en las gráficas 1.4 vemos que nos muestran información similar a la analizada en la tabla 1.1. Nuestra mejor hipótesis es de largo $f_{a,b}$, quedando muy cerca de la esquina (1,0). Por otro lado, en segundo lugar volvemos a tener nuestra f_3, donde el punto se encuentra relativamente cerca, aún así, de lo que sería un clasificador aleatorio. El resto de modelos conllevan un desempeño igual o incluso peor que un modelo aleatorio, quedando el punto, en el caso de las elipses, por debajo de la recta del random classifier.



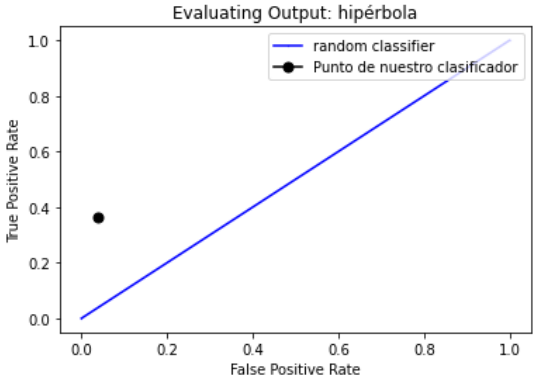
(a) Desempeño dado por $f_{a,b}$



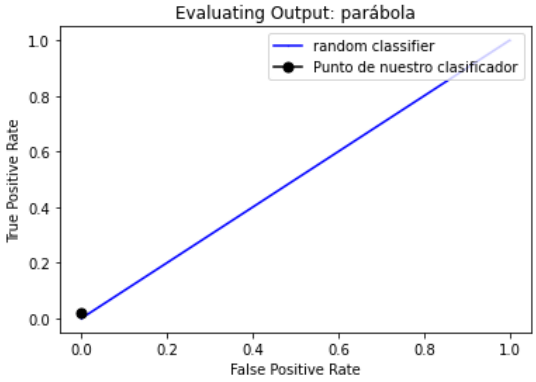
(b) Desempeño dado por f_1



(c) Desempeño dado por f_2



(d) Desempeño dado por f_3



(e) Desempeño dado por f_4

Figura 1.4: Gráficas representativas de la evaluación del desempeño

Como cabía esperar a simple vista, estas hipótesis eran malas para clasificar nuestros datos. Podemos concluir por lo tanto, que funciones más complejas no tienen por qué mejorar la clasificación de nuestra función lineal $f_{a,b}$.

Por último, comentar la influencia del proceso de modificación de etiquetas en el proceso de aprendizaje. Al incorporar ruido en nuestra población, si cogiésemos una muestra de ella podríamos incorporar este ruido a ella, y con ello realizar un proceso de aprendizaje con una muestra que contiene ruido. Evidentemente, ajustarse al ruido nos entorpecerá el camino para lograr una buena solución al problema. Como consecuencia podría darse que alguna de las funciones f_1, f_2, f_3, f_4 fueran buenas funciones que minimizasen el E_{in} de la muestra, algo que sabemos ya que es malo puesto que hemos visto que ajustan mal al modelo. Por otro lado, la existencia de ruido hace que ni nuestra función $f_{a,b}$ que hemos usado para etiquetar explica la muestra.

Ejercicio 2. Modelos Lineales

Ejercicio 2.a) Algoritmo Perceptrón.

En este ejercicio se nos pide implementar el algoritmo PLA para utilizarlo con el conjunto generados en la sección 1. En primer lugar se nos pide hacerlo con el conjunto sin ruido, que sabemos que este conjunto es linealmente separable, luego el algoritmo perceptrón acabará en un número finito encontrando una solución que separe la muestra sin error.

Algoritmo Perceptrón

1. **Entrada**: conjunto de datos y etiquetas $(x_n, y_n),\ n = 1, 2, ..., N$
2. Escoger los pesos iniciales como $w = 0$
3. **Repetir**:
4. **Para cada** $x_i \in \mathcal{D}$
5. **Si** $signo(w^T x_i) \neq y_i$ **Entonces**:
6. Actualizar w: $w = w + y_i x_i$
7. **Hasta**: que no haya cambios en w una época (una pasada entera de $\mathcal{D}$)
8. **Devolver** w

Para la implementación se nos pide que como entrada le podamos asignar los pesos. Es decir, no tienen por qué estar inicializados a cero. Por otro lado tiene otro parámetro llamado max_iter que reduce el número de iteraciones que hace el algoritmo a su valor. Esto podrá impedir que se llegue a la solución óptima en caso linealmente separables donde se necesiten más iteraciones que max_iter, pero por otro lado, permitirá hacer un ajuste sobre conjuntos no linealmente separables, en donde el algoritmo PLA ciclaría indefinidamente.

En el código 1.1 se puede ver la implementación que se ha llevado a cabo. Se ha añadido una variable $mostrarProceso$ para el caso en que queramos ver como evoluciona el algoritmo en cada iteración. Por defecto este parámetro estará a `False`.

Implementación de PLA

Listing 1.1: Perceptron Learning Algorithm

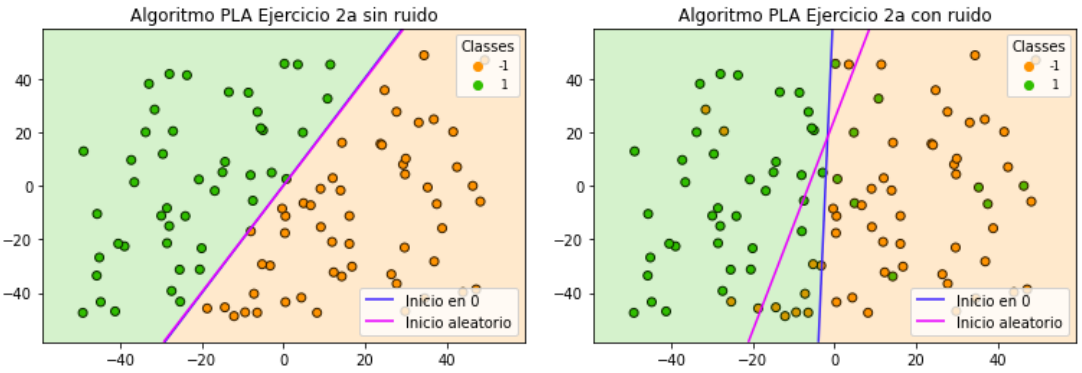
```
def ajusta_PLA(datos, label, max_iter, vini, mostrarProceso=False):
   """
   PLA algorithm.
      - datos:          Conjunto de datos
      - label:          Etiquetas de datos
      - max_iter:       Maximo de iteraciones permitidas
      - vini:           Valor inicial de nuestros pesos
      - mostrarProceso: Muestra graficamente el proceso del algoritmo
   """
   w = vini
   hay_cambio = False
   for iteration in range(max_iter):
      hay_cambio = False
      for i in range(len(datos)): # Para cada item de datos
         if(funcionlineal(w,datos[i])*label[i]<=0): # Si son de distinto signo
            # Actualizamos
            x = np.array([1, datos[i][0], datos[i][1]])
            w = w + label[i]*x
            hay_cambio = True

      if(hay_cambio == False): # Si no ha habido ningun cambio en toda la iteracion
         return w, iteration

      # Mostramos por pantalla el estado actual
      if(mostrarProceso):
         display_graph(datos, label,'Algoritmo PLA Ejercicio 2a:
             ITER:'+str(iteration), etiquetas=['-1',' 1'], reg=[funcionlineal], w=[w],
             etiquetafun=['f(x)=w@x=0'] )


   return w, max_iter
```

En las gráficas 1.5 podemos ver el resultado de una única ejecución del algoritmo para valores iniciales a cero o aleatorio, tanto para el caso con ruido como sin ruido. En la tabla 1.2 se puede observar la media de las ejecuciones para los casos con ruido y sin ruido.

	Media en 10 ejecuciones
Sin ruido ($w_0 = \vec{0}$)	1
Sin ruido ($w_0 = random$)	1.8
Con ruido ($w_0 = \vec{0}$)	1000
Con ruido ($w_0 = random$	1000

Tabla 1.2: Media de iteraciones en conseguir una solución con el algoritmo del código 1.1

En primer lugar vamos a hablar sobre cómo influye la inicialización de los pesos. En el caso sin ruido, el caso separable linealmente, obtenemos prácticamente el mismo resultado: en la gráfica 1.5a se ve como ambas rectas se superponen prácticamente, pero la realidad es que si nos fijamos muy bien, no son la misma recta. Sin embargo, el error es de 0, es decir, explican ambas perfectamente la muestra. Esto es así por lo que se habló al principio, PLA es un algoritmo que, en caso de datos separables, siempre converge a una solución óptima. Podría haberse dado el caso de que el algoritmo no consiguiese la solución en $\#max\_iters$ iteraciones, pero es cuestión de aumentar el número de las iteraciones para conseguir que converja. En nuestro caso,



(a) Resultado de 1 ejecución (b) Resultado de 1 ejecución

Figura 1.5

los valores próximos a 0 nos dan muy buen desempeño, en la tabla 1.2 se observa como no pasa la media de 2 iteraciones y donde se ve también que la inicialización $w_0 = \vec{0}$ es mejor en media que la inicialización aleatoria.

Concretamente, PLA necesita $(RB)^2$ iteraciones donde:

$$B = \text{mín}\{||w|| : \forall i \in [m],\ y_i w^T x_i \geq 1,\ w \in \mathbb{R}^d\} \tag{1.4}$$

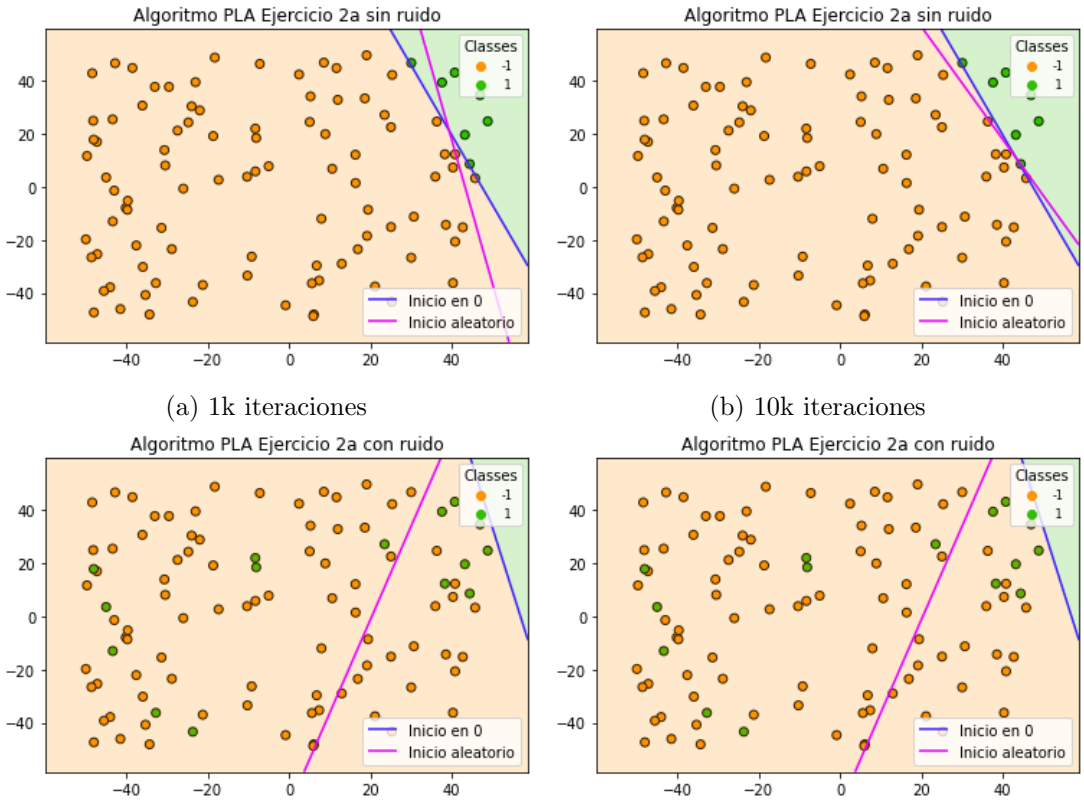
$$R = \max_{i \in 1,\ldots,N} ||x_i|| \tag{1.5}$$

De todas formas, la muestra actual que tenemos esta distribuida de forma muy uniforme por la región $[-50, 50] \times [-50, 50]$ en cuanto a sus etiquetas, es decir, un valor cercano a 0 para los pesos iniciales va a encontrar de forma muy rápida una solución, no hay que hacer traslaciones ni nada, simplemente girar sobre su eje prácticamente. Así que se ha decidido cambiar la semilla para que saliese otro ejemplo más complicado para este caso. Lo obtenido se observa en la tabla 1.3 y en las gráficas 1.6. Vemos como han aumentado drásticamente las iteraciones. Es más, en la gráfica 1.6a se ve como no se ha llegado a conseguir una solución óptima. Eso es porque se alcanzó el máximo de iteraciones. Aumentando el número máximo de iteraciones se ha obtenido 1.6b que si alcanza una solución óptima.

	Media en 10 ejecuciones
Sin ruido ($w_0 = \vec{0}$)	443
Sin ruido ($w_0 = random$)	736.1

Tabla 1.3: Media de iteraciones en conseguir una solución con el algoritmo del código 1.1 con otra semilla aleatoria

Por otro lado, nos fijamos en la gráfica 1.5b donde aquí si que cambian drásticamente los resultados. Estamos ahora en un caso no separable y por lo tanto nunca se llegará a una solución óptima, es más se irán pegando saltos de error, luego de una iteración a otra la solución puede variar muchísimo. Esto también provoca que tampoco se pueda valorar con facilidad el valor del peso inicial, puesto que al no haber convergencia, no acelera ni decelera el proceso. Si es verdad que en estos casos, mientras que los pesos estén cercanos a 0 se comportarán de forma similar dando no muy buenos resultados pero haciendo las cosas medianamente bien. En la tabla 1.2 se ve como no convergen: gastan todas las iteraciones. Si nos vamos al ejemplo de las gráficas de 1.6 vemos como aquí hay un comportamiento similar, teniendo peores resultados debido a que el modelo es más complicado con esos pesos. La idea sigue siendo la misma.



(a) 1k iteraciones

(b) 10k iteraciones

(c) 1k iteraciones

(d) 10k iteraciones

Figura 1.6

Ejercicio 2.b) Regresión Logística.

En este apartado vamos a crear nuestra propia función objetivo f (una probabilidad en este caso) y nuestro conjunto de datos $\mathcal{D}$ para ver cómo funciona regresión logística. Se supondrá, por simplicidad, que f es una probabilidad con valores 0 ó 1, y por tanto que la etiqueta y es una función determinista de x.

Consideraremos datos en dos dimensiones para que sean visibles: $\mathcal{X} = [0, 2] \times [0, 2]$ con probabilidad uniforme de elegir cada $x \in \mathcal{X}$. Se escogerá una línea en el plano que pase por $\mathcal{X}$ que sirva como frontera entre $f(x) = 1$ (donde y toma valores +1) y $f(x) = 0$ (donde y toma valores -1), para ello se seleccionarán dos puntos aleatorios de $\mathcal{X}$ y se calculará la línea que existe entre ellos.

Se propone un experimento, seleccionando 100 puntos aleatorios de $\mathcal{X}$ sobre los que tendremos que aplicar regresión logística. En primer lugar hablaremos del algoritmo de regresión logística.

Regresión Logística

1. Inicializar los pesos w_0
2. **Para cada** $t = 0, 1, 2, ...$ **hacer**
3. Actualizar pesos: $w_{t+1} = w_t - \eta \nabla E_{in}$
4. **Si** $||w_{t-1} - w_t|| < 0{,}01$ **Entonces**:
5. **Devolver** w_t

Implementación Regresión Logística

Listing 1.2: Regresión Logística

```
def sgdRL(X, y, w, lr, epsilon):
    """
    Algoritmo de regresion logistica con SGD. Batches de size 1.
    Devuelve los pesos obtenidos y el numero de iteraciones.
        - X:          datos
        - y:          etiquetas
        - w:          punto inicial
        - lr:         learning rate
        - epsilon:    condicion de parada (distancia entre pesos de epocas)
    """
    indices = np.arange(len(X))
    iterations = 0
    cond_parada = False

    while not cond_parada:
        # Guardamos los pesos anteriores
        w_old = w.copy()

        # Barajamos los indices
        np.random.shuffle(indices)

        # Actualizamos pesos
        for i in indices:
            w = w - lr*graderror_reglog(w,X[i],y[i])

        # Comprobamos si se cumple la condicion de parada
        cond_parada = np.linalg.norm(w_old - w) < epsilon

        iterations += 1


    return w, iterations
```

Regresión logística se basa en el uso de la siguiente clase de funciones

$$\mathcal{H} = \left\{ \theta(w^T x) = \frac{e^{w^T x}}{1 + e^{w^T x}} : \ w \in \mathbb{R}^{d+1}, x \in \mathcal{X} \right\}$$

Una hipótesis $\theta \in \mathcal{H}$ toma valores en [0,1], lo que hace que podamos interpretar su valor como una probabilidad. Siguiendo el criterio de etiquetado:

$$\theta(w^T x) \geq 0{,}5 \longrightarrow y = +1$$

En cuanto el error E_{in}, tomamos una filosofía similar a la de regresión con SGD en donde escogemos la dirección contraria al gradiente para minimizar el error. El tamaño del minibatch será de 1. De la misma manera que en regresión tendremos que elegir un learning rate, que hemos fijado de partida en 0.01 por su buen funcionamiento en las pruebas.

Como nuestros datos son linealmente separables y va a tener un mínimo local y global, nuestro error con un learning rate algo más elevado puede llevarle a la situación reflejada en la Figura 1.7. Es más, se han comparado las iteraciones medias subiendo el learning rate a 0.05 en la tabla 1.4, donde claramente vemos que se llega a una situación como la descrita en la Figura 1.7 al subir el learning rate.

Learning Rate	Media iteraciones
$\eta = 0{,}01$	487.19
$\eta = 0{,}05$	997.94

Tabla 1.4: Iteraciones medias cambiando el learning rate

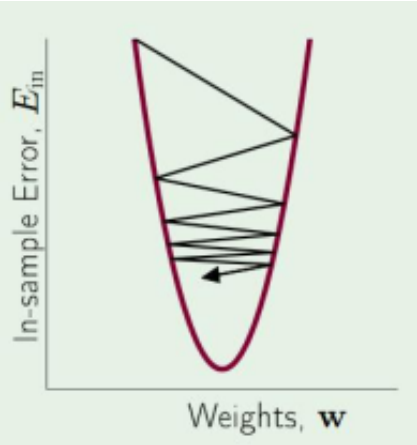


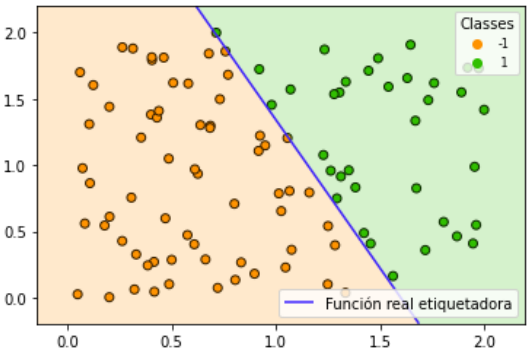
Figura 1.7: Si cogemos un learning rate alto en nuestro ejercicio 2.b) podríamos llegar a esta situación

Se nos pide ajustar una muestra de 100 puntos y ver como funciona en una población que simularemos de la misma forma pero esta vez de 1000 puntos.

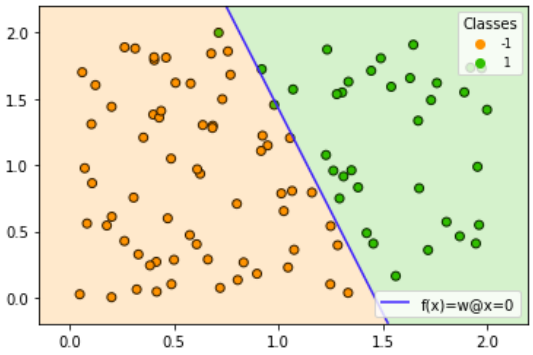
Los resultados medios obtenidos tras la ejecución de los 100 experimentos están descritos en la tabla 1.5 y en las gráficas de 1.8. En esta tabla podemos observar tanto los errores obtenidos por entropía cruzada como el Accuracy obtenido en cada conjunto de datos. Si nos fijamos en los Accuracy vemos que ambos rozan el 100 % de datos bien clasificados, con lo que se puede concluir que se han obtenido unos pesos muy buenos para explicar nuestros datos y su población.

	Valor medio en 100 experimentos
Iteraciones	487.19
Muestra E_{in}	0.1271
Muestra *Accuracy*	0.9944
Población E_{out}	1.844
Población *Accuracy*	0.9859

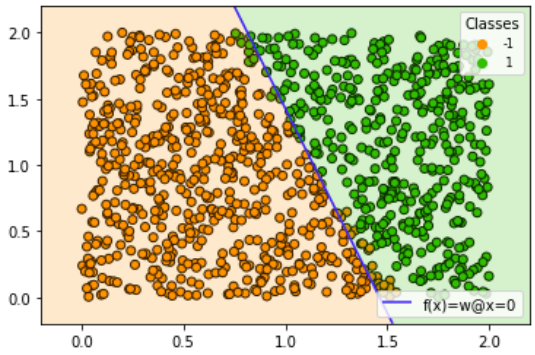
Tabla 1.5: Valores medios



(a) Función f de etiquetado



(b) Conjunto train



(c) Conjunto test

Figura 1.8: Ejercicio 2.b) de Regresión lineal donde se muestran las gráficas de uno de los 100 experimentos propuestos

BONUS. Clasificación de dígitos

En este apartado consideraremos el conjunto de dígitos 4 y 8. Usando las características *intensidad promedio* y *simetría* se pide plantear un problema de clasificación binaria que considere el conjunto de entrenamiento como datos de entrada para aprender la función g.

Por lo tanto nuestro problema está formado por las características $\mathcal{X} = 1 \times \mathbb{R}^2$, las etiquetas son $\mathcal{Y} = -1, +1$ donde la etiqueta -1 clasifica a los 4 y la etiqueta $+1$ clasifica a los 8. Suponiendo que existe una función de etiquetado $f : \mathcal{X} \longrightarrow \mathcal{Y}$, que desconocemos, queremos estimarla dentro de una clase de funciones $\mathcal{H}$. Suponemos que hay una distribución de probabilidad $\mathcal{P}$ en $\mathcal{X} \times \mathcal{Y}$ y que los elementos de la muestra se extraen de forma i.i.d.

Basándonos en el ERM (empirical risk minimization) estimaremos nuestra función $g \in \mathcal{H}$. Nuestra medida para evaluar el error va a consistir en ver qué elementos están bien clasificados y cuáles no:

$$E_{in}(h) = \frac{1}{N} \sum_{n=1}^{N} [[h(x_n) \neq y_n]], \qquad h \in \mathcal{H}$$

Para encontrar una solución se va usar un algoritmo de regresión lineal (en nuestro caso SGD) y el algoritmo PLA-Pocket. Para el caso de PLA-Pocket haremos dos experimentos en primer lugar:

1. Probaremos un inicio aleatorio, haciendo 10 ejecuciones de estas y cogiendo la media de los valores. **Nota**: en vista de que 10 ejecuciones ralentiza mucho la ejecución del código, y en vista de que las diferencias de resultados entre una ejecución aleatoria y la media

de varias de estas no varía mucho, en el fichero `.py` solo habrá una ejecución con inicio aleatorio.

2. Probaremos un inicio cogiendo los pesos obtenidos en el algoritmo de regresión lineal sgd.

Por otro lado obtendremos cotas sobre el verdadero valor de E_{out}. Calcularemos la cota tanto usando la dimensión de Vapnik-Chervonenkis como usando la desigualdad de Hoeffding. Al usar la desigualdad de Hoeffding podemos calcular la cota bien usando E_{in} bien usando E_{test}:

Desigualdad de Hoeffding

Dado N tamaño del conjunto train, $\delta > 0$, se tiene, con probabilidad $1 - \delta$, que:

$$E_{out}(g) \leq E_{in}(g) + \sqrt{\frac{1}{2N} log \frac{2|\mathcal{H}|}{\delta}}, \tag{1.6}$$

Al usar flotantes de 64bits podemos asumir que $|\mathcal{H}| = 2^{64\times 3} = 2^{192}$. Esta desigualdad, como bien se ha dicho antes podemos usarla usando E_{test} en lugar de E_{in}. Lo que conseguimos haciendo este cambio es que antes de ver el conjunto de datos test ya hemos fijado nuestra hipótesis $g \in \mathcal{H}$ por lo que podemos asumir directamente que $|\mathcal{H}| = 1$. Fijaremos un $\delta = 0{,}05$.

Dimensión de Vapnik-Chervonenkis

Dado N tamaño del conjunto train, $\delta > 0$, se tiene, con probabilidad $1 - \delta$, que:

$$E_{out}(g) \leq E_{in}(g) + \sqrt{\frac{8}{N} log \frac{4((2N)^{d_{vc}} + 1)}{\delta}}, \tag{1.7}$$

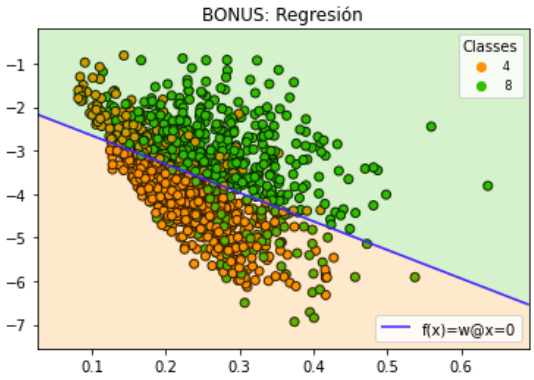
Fijaremos un $\delta = 0{,}05$.

Resultados

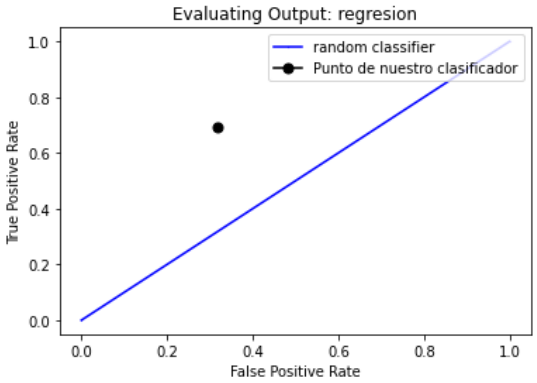
Hemos ejecutado los 3 casos distintos. En las gráficas 1.9, 1.10, 1.11 se pueden ver visualmente como clasifican las funciones obtenidas del algoritmo SGD, PLA-Pocket con inicio aleatorio y PLA-Pocket inicializado con los pesos del SGD respectivamente. En estas gráficas vemos visualmente como el algoritmo PLA-Pocket trata de buscar una recta más vertical, distanciándose de la del SGD y asemejándose mucho entre los dos casos de PLA-Pocket. En los tres casos fijándonos en la gráfica (b) de cada Figura, vemos como clasifican mejor que un clasificador aleatorio.

Por otra parte podemos ver a las 3 rectas actuando conjuntamente en las gráficas 1.12. Aquí se ve más claramente como el algoritmo de regresión lineal tiende más a hacer una recta horizontal. La explicación a esto es porque en el algoritmo SGD se intenta minimizar la distancia a los puntos y en el PLA-Pocket nos centramos simplemente en clasificar dejando a un lado unos y a un lado otros, sin importar como de lejos de la frontera de clasificación estén los puntos.

Por último, los resultados cuantitativos obtenidos se ven reflejados en la salida que se puede ver en 1.3. A continuación de ello viene una conclusión acerca de los mismos.

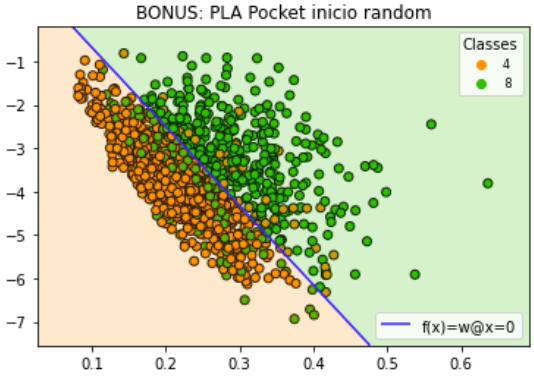


(a) Función g obtenida

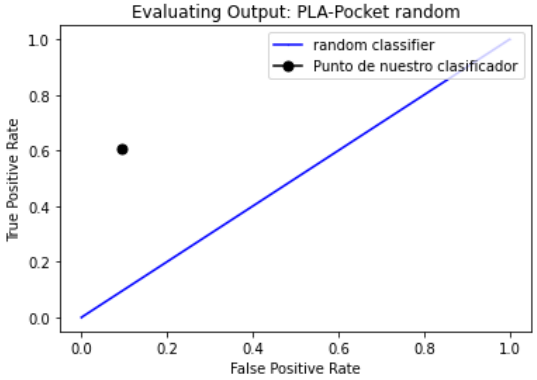


(b) Desempeño del resultado respecto a un clasificador aleatorio

Figura 1.9: Ajuste obtenido con el algoritmo SGD.

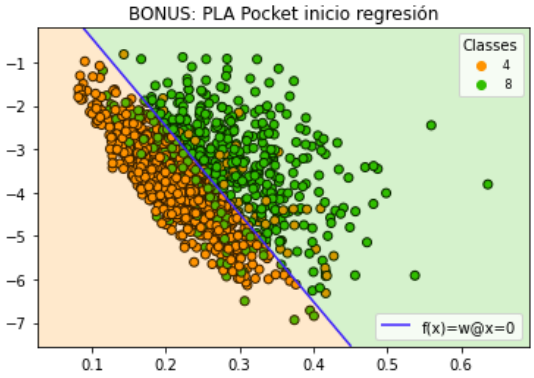


(a) Función g obtenida

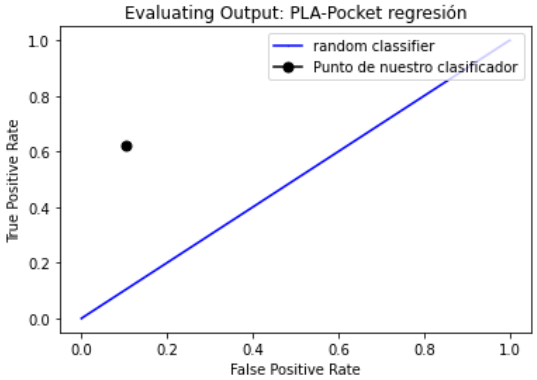


(b) Desempeño del resultado respecto a un clasificador aleatorio

Figura 1.10: Ajuste obtenido con el algoritmo PLA-Pocket con inicialización aleatoria.

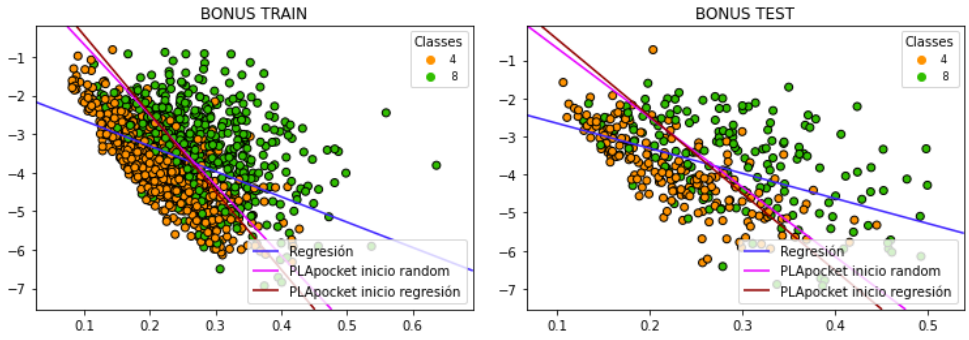


(a) Función g obtenida



(b) Desempeño del resultado respecto a un clasificador aleatorio

Figura 1.11: Ajuste obtenido con el algoritmo PLA-Pocket con pesos iniciales obtenidos por SGD.



(a) Conjunto Train (b) Conjunto Test

Figura 1.12: Representación de los tres resultados conjuntamente

Comentando en primer lugar las cotas. Comparar las cotas entre los distintos algoritmos es equivalente a comparar los errores puesto que el otro término es igual para ambos ya que estamos con la misma clase de funciones. Así que en este análisis de resultados sobre las cotas vamos a centrarnos únicamente en los valores que se han tomado por separado.

La cota de Vapnik-Chervonenkis vemos que es mayor a la obtenida por la desigualdad de h¡Hoeffding. Tomando esta cota como una probabilidad de error, vemos que la cota deja bastante que desear en todos los casos puesto que se da una cota superior al 0.5, que sería el error esperado de un clasificador aleatorio. Por otro lado, la cota dada por la desigualdad de Hoeffding es más fina, diciéndonos poco todavía sobre el resultado obtenido con el SGD pero dando esperanzas en los resultados obtenidos con PLA-Pocket sobre como se comportará nuestra hipótesis fuera de la muestra ya que la cota está por debajo del 0.5. En cualquier caso, son cotas bastante poco finas, al final son cotas usadas más en teoría que en la práctica, a la vista está que no arrojan mucha información.

Comentando ahora los valores de error junto al Accuracy (Salida 1.3) tenemos que el algoritmo PLA-Pocket mejora el método de regresión en ambos casos, iniciándose con valores aleatorios o con valores ya entrenados con el propio algoritmo SGD. Se consiguen errores relativamente bajos con PLA-Pocket y un accuracy que se puede considerar bueno, acercándose al 80 % de accuracy.

Por otro lado, el inicio de pesos aleatorios versus el inicio con pesos preentrenados con SGD no tiene una importancia relativa con el comportamiento de PLA-Pocket en este caso. Y es en este caso por cómo están distribuidos los puntos y porque la inicialización de los pesos aleatorios han sido valores cercanos a ceros: estos dos factores hacen que de partida se tenga una solución relativamente "buena" (con buena nos referimos a que, dentro de ser una mala solución, está bien posicionada para mejorar).

Podemos concluir, que el algoritmo PLA-Pocket, con un número suficiente de iteraciones puede mejorar al método de regresión. Por otro lado, que no se ha comentado, es bastante más lento. De aquí podemos extraer de conclusión que es buena idea usar SGD para obtener una buena solución de manera rápida y hacer, con esa solución, unas cuantas iteraciones con el algoritmo PLA-Pocket, para conseguir mejorar en lo que se pueda la solución del método de regresión lineal.

Listing 1.3: Resultados de ejecución del bonus

```
-----------------------Regresion-----------------------
ERRORES:
    Accuracy=0.6876046901172529
    Ein=0.31239530988274705
    Etest=0.35792349726775957
COTAS:
    Cota con Vapnik-Chervonenkis: 0.7433318153534507
    Cota con Hoeffding: 0.5517176520658715
    Cota con Hoeffding usando test: 0.42891260070037507

-----PLA pocket inicio random (media de 10 iteraciones)-----
ERRORES:
    Accuracy=0.7696817420435511
    Ein=0.23031825795644892
    Etest=0.25136612021857924
COTAS:
    Cota con Vapnik-Chervonenkis: 0.6612547634271525
    Cota con Hoeffding: 0.4696406001395734
    Cota con Hoeffding usando test: 0.32235522365119473

---------------PLA pocket inicio con regresion---------------
ERRORES:
    Accuracy=0.7730318257956449
    Ein=0.2269681742043551
    Etest=0.26229508196721313
COTAS:
    Cota con Vapnik-Chervonenkis: 0.6579046796750587
    Cota con Hoeffding: 0.4662905163874796
    Cota con Hoeffding usando test: 0.3332841853998286
```

Experimento extra. Añadir estocasticidad a PLA-Pocket

Esta sección surge de la idea de recorrer los datos en PLA-Pocket de forma aleatoria. Se puede ver en la salida 1.4 como los resultados mejoran levemente con respecto a 1.3.

Listing 1.4: Resultados de ejecución con PLA-Pocket estocástico

```
-----------------PLA pocket inicio random-----------------
ERRORES:
    Accuracy=0.7864321608040201
    Ein=0.2135678391959799
    Etest=0.25136612021857924
COTAS:
    Cota con Vapnik-Chervonenkis: 0.6445043446666835
    Cota con Hoeffding: 0.4528901813791044
    Cota con Hoeffding usando test: 0.32235522365119473

--------------PLA pocket inicio con regresion--------------
ERRORES:
    Accuracy=0.788107202680067
    Ein=0.211892797319933
    Etest=0.25136612021857924
COTAS:
    Cota con Vapnik-Chervonenkis: 0.6428293027906367
    Cota con Hoeffding: 0.45121513950305747
    Cota con Hoeffding usando test: 0.32235522365119473
```

El sentido que tiene es que sin estocasticidad, en cada iteración los datos empujan en cada época de forma más o menos equivalente. Veamos un caso que ejemplifique lo que digo:

Nos situamos en un problema de clasificación binaria en donde usaremos regresión logística con todas las hipótesis sobre el problema que son necesarias para que tenga sentido el estudio. Supongamos que nuestra clase de funciones son $\mathcal{H} = \{f(x_1, x_2) = w_0 + w_1x_1 + w_2x_2, \ \ con\ w_i \in \mathbb{R}\}$ y que los datos de nuestra muestra son $\{X_1, X_2, ..., X_n\}$ en ese orden. Al recorrerlos, imaginemos que el dato X_i empuja "*hacia la izquierda*" a la recta con el algoritmo PLA-Pocket y esto hace que el siguiente dado $Xi + 1$ se quede mal clasificado y que este ahora empuje "*hacia la derecha*", creando así una cadena. Esto acabará convergiendo puesto que el orden va a ser siempre siempre el mismo al empujar y cada vez con menos fuerza debido a que los datos cada vez están más cerca de la frontera y por tanto de estar mejor clasificados (recordamos que la actualización de pesos se hace sumando un vector que apoya a la buena clasificación del actual punto mal clasificado en una fuerza menor a la del propio vector por el learning rate). Sin embargo es un proceso lento.

Si por otro lado, metemos estocasticidad a la hora de recorrer los datos, se podría pensar que estos "*empujones*" estarán desordenados y podrá provocar que no sean cada vez con menos fuerza, sino que en un inicio serán fuertes y en cuanto se encuentre en una posición más o menos buena los datos empujarán más débilmente. Es decir, se podría pensar que con esta aleatoriedad se consigue lo que se habló en su momento del learning rate: lo ideal es tener un learning rate alto para descender rápido por la pendiente e ir reduciendo el learning rate conforme te acerques al mínimo local para no pasarte.

Explicando estos dos razonamientos nos damos cuenta de una cosa, que sin saber si la opción de la estocasticidad es realmente viable ya que solo lo hemos probado en un problema aunque con resultados positivos, no hay en principio ningún indicio de que esto vaya a ser siempre así. Sin embargo, el algoritmo original, sin estocasticidad, al llevar todo el rato el mismo orden los "*empujones*" serán cada vez más débiles y esto hará que se converja con total probabilidad y que en cada época (sin contar la primera puesto que partimos de pesos que no han sido calculados por el algoritmo) se mejore el resultado.

También es de esperar que ambos hagan el mismo número de "*empujones*" a cada lado, porque al final recorren los mismos datos, lo único que varía es la fuerza con la que los dan y es aquí en donde está el interés, por si acelera el proceso.

Los resultados dados nos dicen que mejoran un poco lo que teníamos, pero esta mejora es pequeña y tendría que estudiarse si para distintas semillas y distintos problemas esta técnica tiene sentido.