**PRÁCTICA 2 - COMPLEJIDAD DE $\mathcal{H}$ , PERCEPTRON Y REGRESIÓN LOGÍSTICA**

Javier Sáez Aprendizaje Automático

# Complejidad de $\mathcal{H}$ y ruido.

En este ejercicio, trabajaremos con clases de funciones sencillas y con datos que contienen ruido. El objetivo será estudiar cómo afecta la complejidad de la clase de funciones al modelo y cómo afecta el ruido a los resultados del mismo.

Lo primero que vamos a hacer es generar nuestros datos. Para ello, se utilizan las funciones `generate_uniform_data(N,dim,range)` y `generate_gaussian_data(N,dim,sigma)`. Comentamos primero que, en los datos uniformes, todos tienen la misma probabilidad y sabemos que si $X \sim U(a,b)$, se tiene que

$$f_X(x) = \frac{1}{b-a}.$$

En este caso, se pide que $a = -50, b = 50$ , por lo que tomaremos puntos del cuadrado $[-50,50] \times [-50,50]$. El resultado que obtenemos es:

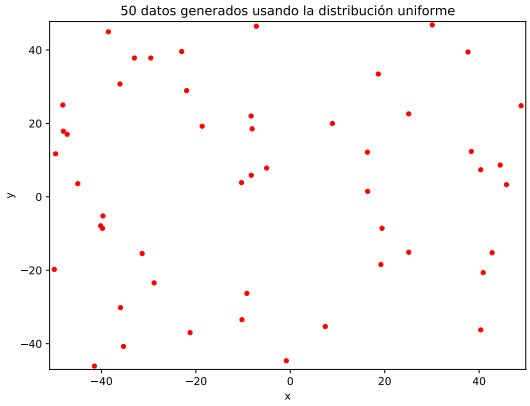



Figura 1: Datos generados por la distribución uniforme en $[-50,50] \times [-50,50]$.

De igual manera, se pide generar datos pero usando la distribución normal o Gaussiana. Sabemos que en una distribución Gaussiana $X \sim \mathcal{N}(\mu, \sigma^2)$ , cuyos parámetros son la media $\mu$ y la varianza $\sigma^2$, la función de densidad viene dada por:

$$F_{\mathcal{N}}(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}.$$

Generamos de nuevo 50 valores en dos dimensiones. Para ello, se nos indica que $\sigma_x = \sqrt{5}$ y $\sigma_y = \sqrt{7}$. El resultado obtenido es el siguiente:

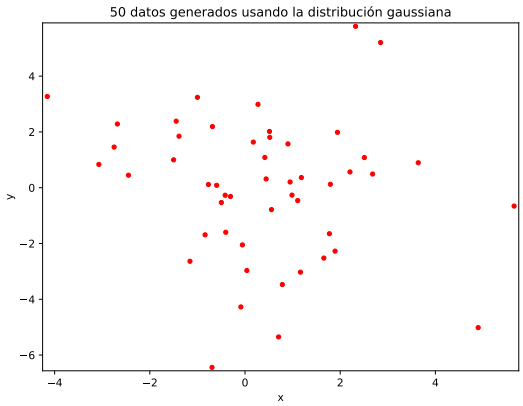



Figura 2: Datos generados por la distribución normal en $[-50,50]\times[-50,50]$ con $\mu = 0, \sigma = (\sqrt{5},\sqrt{7})$.

Como podemos ver, prácticamente la totalidad de los datos se encuentran en los intervalos conformados por 3 veces las desviaciones típicas de cada eje, esto es: $[-3\sigma_x, 3\sigma_x]\times[-3\sigma_y, 3\sigma_y]$, como es esperado en esta distribución.

Una vez que hemos obtenido nuestros datos, vamos a proporcionarles unas etiquetas y vamos a introducir ruido en estos datos. Vamos primero a asignarle las etiquetas. Para ello, vamos generar una recta aleatoria usando la función dada (a la que le hemos cambiado el nombre) generate_line(interval) a la que le pasamos un intervalo $interval = [c,d]$, genera dos puntos en $\mathbb{R}^2$ dentro del cuadrado $[c,d]\times[c,d]$ y devuelve la pendiente $(a)$ y el término independiente $(b)$ de la recta que pasa por ellos. Una vez tenemos estos dos valores, sabemos que la recta es simplemente:

$$y = ax + b.$$

Considerando la función $f(x,y) = y - ax + b$, obtenemos la distancia del punto $(x,y)$ a la recta, y podemos considerar como etiqueta el **signo** de esta distancia.

$$f(x,y) = sign(y - ax - b).$$

Se considera que los puntos que están sobre la recta son de la clase del 1. Si dibujamos una recta aleatoria y dividimos 100 puntos generados mediante la distribución uniforme, obtenemos el siguiente gráfico:

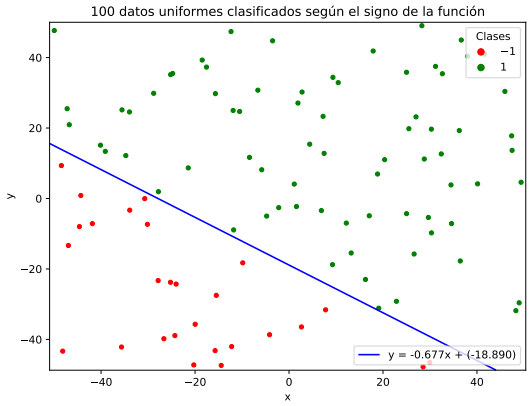



Figura 3: 100 datos etiquetados mediante la separación por una recta aleatoria.

Nos toca entonces introducir ruido en estos datos. Para ello, simplemente tomamos el vector de etiquetas y de nuestros datos, obtenemos los índices de las etiquetas positivas y de esas seleccionamos aleatoriamente un 10 % y las cambiamos a negativas. Hacemos lo mismo con las negativas. Esto se hace en la función generate_noise, cuyo código es simple:

```
def generate_noise(y,per = 0.1):

    ycopy = np.copy(y)

    for label in {-1,1}:
        # Get index of labels
        idx = np.where(y==label)[0]
        # Random selection of percentage*length labels
        changes = np.random.choice(idx,int(per*len(idx)),replace=False)
        # Change labels
        ycopy[changes] = -label

return y_copy
```

Se realiza una copia de las etiquetas para mantener las originales. Si volvemos a dibujar los datos , el resultado es el siguiente:

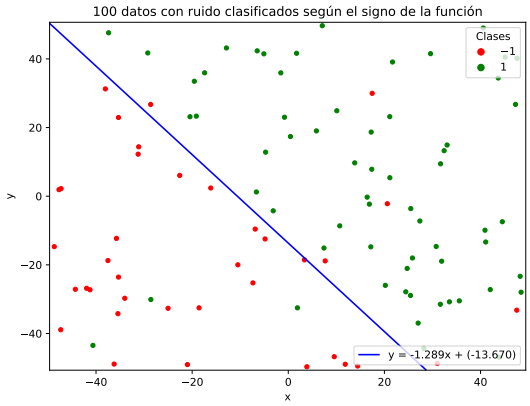
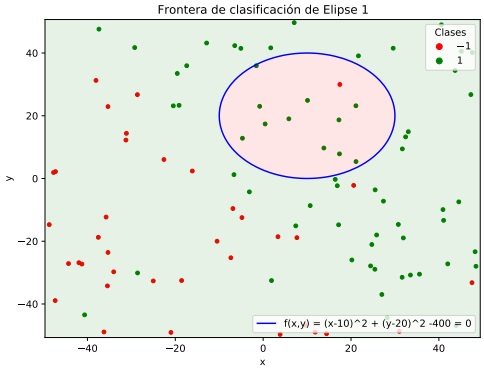



Figura 4: 100 datos etiquetados y con 10 % de ruido.

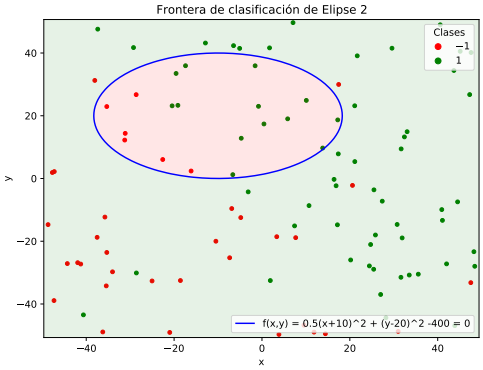
Como queríamos obtener, tenemos los puntos mal etiquetados respecto a la recta. Para continuar en este ejercicio, definimos las siguientes funciones:

- $f(x,y) = (x-10)^2 + (y-20)^2 - 400$, que es una elipse.
- $f(x,y) = \frac{1}{2}(x+10)^2 + (y-20)^2 - 400$, que es otra elipse.
- $f(x,y) = \frac{1}{2}(x-10)^2 - (y+20)^2 - 400$, que es una hipérbola.
- $f(x,y) = y - 20x^2 - 5x + 3$, que es una parábola clásica.

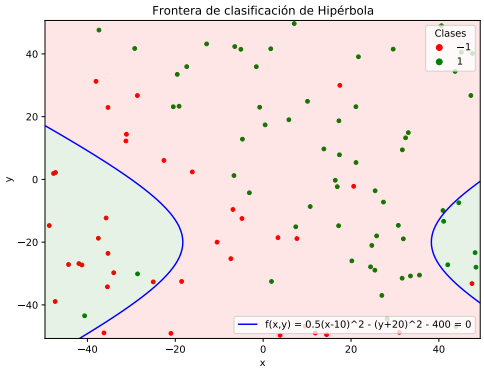
Estas serán nuestras nuevas funciones frontera de clasificación para la muestra. Sin embargo, ellas no definirán etiquetas para los puntos que generamos, sino que seguiremos usando las **etiquetas con ruido** del apartado anterior. Recordamos que la frontera con estas funciones podemos definirla igualando las expresiones a cero, y usando esto podemos dibujar las cuatro con lars regiones negativa y positiva para ver qué ocurre con los puntos etiquetados que habíamos generado:



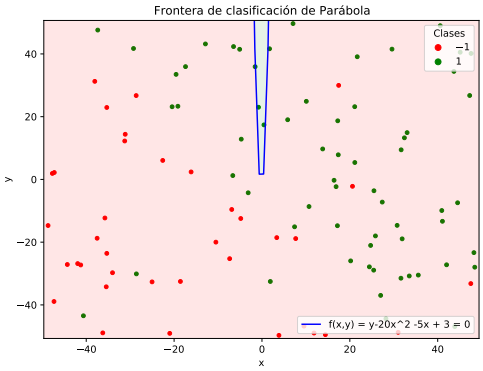

(a) Primera elipse




(b) Segunda elipse




(c) Hipérbola




(d) Parábola

Figura 5: Fronteras de clasificación. En verde zona positiva. En rojo, negativa.

Como se puede observar, ninguna de las funciones ajusta nada bien los datos que hemos generado inicialmente. Podemos tratar de justificar esto obteniendo la precisión (o *accuracy*) de cada uno de ellos. Sabemos que esta viene dada por

$$Accuracy = \frac{TP + TN}{P + N}$$

donde $TP$ es el número de ejemplos positivos que se han clasificado bien, $TN$ es el número de ejemplos negativos que se han clasificado bien, y $P, N$ son los números de positivos y negativos respectivamente. Además, si las clases no estuviesen balanceadas, existe .Para calcular esto, `sklearn` dispone de un conjunto de funciones que se encuentra en sklearn.metrics, y que son `accuracy_score(y_true,y_predicted), balanced_accuracy_score(y_true, y_predicted).` Los resultados que obtenemos al ejecutar un trozo de código que nos calcula para cada una de las funciones estos valores, son los siguientes:

| Funcion | Precisión | Precisión balanceada |
|---|---|---|
| Recta | 91 | 89,51 |
| Elipse 1 | 53 | 42,62 |
| Elipse 2 | 56 | 47,79 |
| Hipérbola | 30 | 36,66 |
| Parábola | 39 | 51,58 |

Como ya nos indicaban las gráficas, las **precisiones** que obtienen las funciones dadas son muy bajas. Los cambios en aumentando en la precisión balanceada se deben a que el número de datos no es igual en ambas clases. Además, la precisión de estos clasificadores es muy inferior a la que obtenemos por la recta, así que lo que podemos concluir es que si fuésemos a considerar alguno de estos clasificadores para nuestro conjunto de datos, los resultados serían obviamente mucho peores que los del clasificador lineal inicial. Por tanto, hemos visto que añadir complejidad al mdelo no implica que los resultados vayan a mejorar, por lo que es mejor usar el **clasificador lineal**. Esto además tiene mucho sentido en este caso, pues sabemos de antemano que las etiquetas han sido generadas usando este modelo lineal.

En cuanto al **ruido** que obtenemos en las etiquetas, debemos asumir que con datos reales esto es algo que pasará muy a menudo. Esto además influirá de forma muy directa en el aprendizaje, pues ocasiona que los datos **no sean separables** y tengamos que estar siempre tratando con aproximaciones o probabilidades de que nuestros clasificadores sean correctos, y obtener clasificadores que tengan error cero con datos reales es prácticamente imposible.

# Modelos lineales

## Algoritmo Perceptron

En este apartado, estudiaremos el algoritmo *PLA* para el problema de clasificación binaria. Este algoritmo recibe dos vectores $X, y$ con los datos y las etiquetas y devuelve los coeficientes del hiperplano que separa completamente los datos.

Veamos este algoritmo en pseudocódigo. Si $h(x) = sign(w^T x + b)$ es la función que usamos para clasificar:

**Algorithm 1:** Algoritmo Perceptron ( $X$, $y$, $max$)

**Result:** Hiperplano separador
$w \leftarrow$ inicialización
**while** *w cambie* **do**
  **for** $x \in X$ **do**
    **if** $h(x)$ *no es igual a* $y_x$ **then**
      $w \leftarrow w + etiqueta(x) \cdot x$
    **end**
  **end**
**end**
**return** $w$

Como vemos, si la etiqueta que nuestros pesos están produciendo no es la correcta, esto es, si $h(x) \neq y_x$, actualizamos en cada iteración $w$ de la forma:

$$w(t+1) = w(t) + x\dot{y}_x,$$

y realizamos esta operación tantas veces como sea necesario hasta que no haya ningún cambio en los pesos, es decir, todos los puntos del conjunto de datos estén bien clasificados. Es claro que para que este algoritmo termine , los datos deben ser **linealmente separables** pues, de lo contrario, para cualesquiera pesos obtendríamos que existe un punto $x \in X$ para el cual $h(x) \neq y_x$, por lo que nuestro algoritmo iteraría indefinidamente. De hecho, se sabe que de por sí es un algoritmo muy costoso pues, si los datos son separables, encontrará la solución pero:

**Proposición.-** Si $B = \min\{\|w\| : y_i w^T \geq 1, \quad w \in \mathbb{R}^d\}$ y $R = \max_i \|x_i\|$, entonces PLA encontrará la solución óptima en, a lo sumo:

$$(RB)^2 \text{ iteraciones}$$

Esto puede ser un número muy elevado según las dimensiones de los datos. Es por ello que, para cuando trabajemos con datos con ruido, hemos establecido un número máximo de iteraciones a realizar, `max_iter = 1000`, pues sabemos que nunca encontrará los pesos que separen los datos.

Una vez explicado, vamos a pasar a ejecutar el algoritmo. Lo hacemos primero usando los **datos sin ruido** de la sección anterior.

- Si inicializamos con el vector $w_0 = (0, 0, 0)$, obtenemos lo siguiente:

```
Pesos obtenidos para peso inicial [0,0,0]: [564. 65.01749303
   50.37298718]
Iteraciones para peso inicial [0,0,0]: 105
Accuracy = 100.0
```

Como podíamos esperar, el resultado es perfecto y el número de iteraciones es pequeño.

- Ahora, se pide que se inicialice el vector de pesos inicial $w_0$ con valores aleatorios entre 0 y 1. Para ello, usando `np.random.uniform(0,1,3)` obtenemos esos vectores aleatorios (esta es la función que usa `generate_uniform_data`, por lo que podríamos haberla usado). Se pide que se repita el experimento 10 veces y se dé el valor medio de las ejecuciones, y el resultado obtenido es el siguiente:

```
Valor medio de iteraciones necesario para converger: 120.0
Valor medio de accuracy en pesos aleatorios: 100.0
La desviacin  tpica  en las iteraciones ha sido: 14.65, pues las
   iteraciones son: [97, 134, 134, 108, 111, 121, 150, 116, 111, 118]
```

  Como podemos ver, se ha dado un muy leve incremento en el número de iteraciones necesarias para encontrar los mejores pesos, y tenemos una desviación típica de $\approx 15$, lo cual es alto por lo que no podemos indicar que inicializar los pesos aleatoriamente pueda ser mejor que inicializarlos a cero.

Vamos a analizar ahora el caso en el que usamos los **datos con ruido**. Usamos los datos y etiquetas obtenidos en el ejercicio anterior y vamos a ver qué ocurre y tratar de dar una explicación al resultado. Si ejecutamos el mismo código que antes pero con las etiquetas con ruido, obtenemos el siguuiente resultado:

```
Iteraciones para peso inicial [0,0,0]: 1000
Accuracy = 73.0
Resultados medios para 10 pesos iniciales aleatorios:
Valor medio de iteraciones necesario para converger: 1000.0
Valor medio de accuracy en pesos aleatorios: 85.5
La desviacin  tpica  en las iteraciones ha sido: 0.0, pues las iteraciones
   son: [1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000]
```

Observamos que en todos los casos, se han hecho todas las iteraciones posibles que habíamos marcado como tope. Pero, esto era bastante previsible. Hemos comentado anteriormente que para que *PLA* siempre obtiene la solución que tiene error cero **si, y solo si,** los datos sobre los que se aplica son **separables**. En este caso, los datos no son separables debido al ruido, por lo que nuestro algoritmo seguirá iterando infinitamente tratando de encontrar el hiperplano que clasifique bien a todos los puntos pero no será capaz de encontrarlo. En todas las épocas habrá puntos que no estén bien clasificados y por tanto se seguirá actualizando los pesos e iterando.

Guardando los pesos que se obtienen en cada época, podemos ver el error que se obtiene:

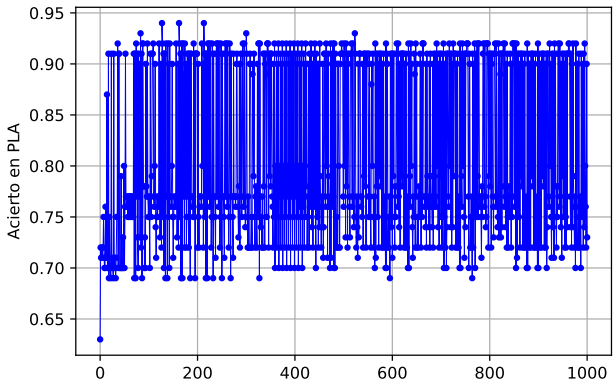



Figura 6: Porcentaje de acierto en datos con ruido por cada época usando *PLA*.

El porcentaje de acierto va variando de manera muy brusca según las épocas, y no consigue en ningún momento converger, como esperábamos.

## Regresión logística

Hasta ahora, habíamos trabajado siempre con etiquetas determinísticas. Es decir, para cada dato $x \in X$, producíamos una etiqueta $y_x \in \{-1, 1\}$. En **regresión logística**, vamos a producir una *probabilidad* $p \in [0, 1]$.

En la regresión lineal, obteníamos las etiquetas del siguiente modo:

$$f(x) = \sigma_{lin}(w^T x) \quad \text{con} \quad \sigma_{lin}(a) = sign(a).$$

Ahora, para la regresión lineal, utilizaremos una nueva función $\sigma$, llamada función logística, que viene dada por

$$\sigma(t) = \frac{1}{1 + e^{-t}},$$

que siempre toma valores en el intervalo $[0, 1]$. Por tanto, ahora lo que estamos obteniendo con esta $f(x)$ es la probabilidad de que $x$ pertenezca a la clase positiva $+1$, esto es:

$$f(x) = \mathbb{P}[y = +1 \mid x].$$

Tenemos que definir por tanto también una función de error para ver cómo de bueno es nuestro modelo. Usando el método de *máxima verosimilitud*, trataremos de seleccionar $h$ de nuestro conjunto de hipótesis $\mathcal{H}$ que , dado un conjunto de datos $X = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ maximice la probabilidad de que se den todas las etiquetas correctas para esos datos usando $h$, esto es $\prod_{i=1}^{N} P(y_n \mid x_n)$. Sabemos que maximizar esa probabilidad equivale a minimizar el siguiente error:

$$E_{\text{in}}(w) = \frac{1}{N} \sum_{n=1}^{N} \ln\left(1 + e^{-y_n w^T x_n}\right).$$

Puesto que la técnica que utilizaremos para minimizar esta función de error será el gradiente descendente estocástico (SGD), es necesario denotar que la derivada de esta función viene dada por:

$$\nabla E_{\text{in}}(w) = -\frac{1}{N}\sum_{n=1}^{N}\frac{y_n x_n}{1+e^{y_n x^T x_n}}.$$

En nuestro caso, como usaremos SGD, lo evaluaremos punto a punto actualizando los pesos según el error en cada punto.

El Gradiente Descendente Estocástico será implementado con las siguientes condiciones:

- Los pesos iniciales serán el vector cero.
- El algoritmo dejará de iterar cuando la distancia entre los pesos de la época actual y la anterior sea menor que un $\epsilon$, que fijaremos a $\epsilon = 0{,}01$.
- Se aplicará una permutación del orden de los datos antes de usarlos en cada época del algoritmo.
- La tasa de aprendizaje se fija a $\eta = 0{,}01$.

Una vez que hemos plantado las condiciones, vamos a realizar el siguiente experimento:

1. Generamos $N = 100$ puntos aleatorios y una recta $= y = ax + b$. Etiquetamos los puntos usando el signo de la distancia del estos puntos a la recta.

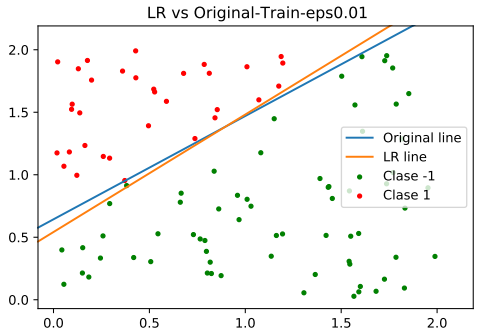
```
# Generate data and line
a,b = generate_line([0,2])
X = generate_uniform_data(N_train,2,[0,2])
# Create tags using the line
y = np.array([f(x[0],x[1],a,b) for x in X])
X = np.hstack((np.ones((X.shape[0], 1)), X))
```

2. Aplicamos el algoritmo sgdRL que realiza la regresión logística para obtener los pesos de la recta que aproxima nuestros datos. Recordamos que en Gradiente descendente, la actualización de los pesos se hace en cada paso de la forma:

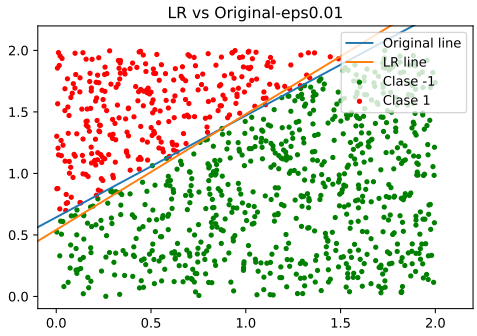
$$w(t+1) = w(t) - \eta * \nabla E_{\text{in}}(w).$$

3. Generamos un conjunto de test $X_{\text{test}}$ de $N_test = 1000$ puntos nuevos, y los etiquetamos con la misma función que etiquetamos los puntos que utilizamos para el entrenamiento.
4. Calculamos el error en el conjunto de testcon la función de error logística.

Vamos a ejecutar el experimento una única vez para ver cómo ajusta la recta obtenido al conjunto de datos de test $X_{\text{test}}$. El resultado tanto dentro como fuera de la muestra es el siguiente:




(a) Regresión Logística sobre conjunto de entrenamiento.



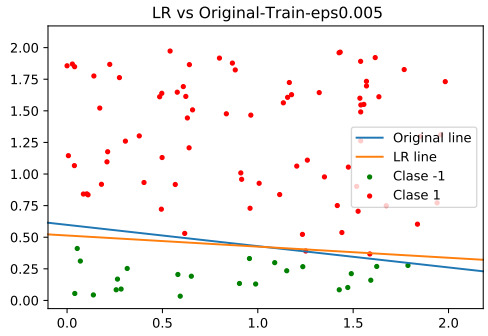

(b) Regresión Logística fuera de la muestra de entrenamiento.

Figura 7: Regresión logística sobre $N = 100$ datos de muestra y sobre nueva muestra de 1000 datos.

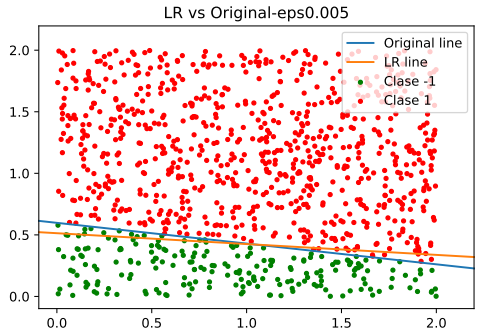
El ajuste como podemos ver es casi perfecto, aunque algún dato queda mal clasificado. Si mostramos por pantalla el error en la muestra, el resultado es:

```
El error en la muestra es de [0.08456819097661489]
Average results in 1 experiments: // Estas lneas se refieren a fuera de la
    muestra
  Iterations: 333.0
  Error: 0.10187873937789174
```

Lo cual nos indica que , como podíamos ver, es bajo pero no es cero. Esto es debido a que la diferencia entre los pesos en dos iteraciones ha sido muy pequeña aunque estos no ajusten perfectamente a los datos. Si le diésemos un número mayor de iteraciones o estableciésemos otra condición de parada con un epsilon menor, es posible que obtuviésemos un mejor ajuste de los datos, aunque con un mayor número de iteraciones. Aunque no sea el objetivo del ejercicio, se ha realizado una prueba de esto para ver si esto era cierto y el resultado es el siguiente:




(a) Regresión Logística sobre conjunto de entrenamiento.




(b) Regresión Logística fuera de la muestra de entrenamiento.

Figura 8: Regresión logística con epsilon menor. Iteraciones necesarias: 850.

Se obtiene que el error en la muestra es $E_{in} = 0{,}053$ y el error en el conjunto de test es $E_{out} = 0{,}08$, ligeramente menores que en el caso anterior, aunque las iteraciones aumentan como vemos en el pie de la figura.

Por último, lo que haremos será **repetir este experimento** $N_{experiments} = 100$ veces y calcular la media de los errores fuera de la muestra y el número medio de iteraciones, para hacernos una idea de cómo de bien funciona la regresión logística sobre los datos generados de esta forma. El resultado de la ejecución de este experimento es el siguiente:

```
Average results in 100 experiments:
    Iterations: 424.61
    Error out sample: 0.11801947037697608
```

Observamos que el número de iteraciones no es muy alto para obtener unos pesos que ajusten razonablemente bien nuestros datos incluso fuera de la muestra, obteniendo un error de 0,11.

# Bonus - Clasificación de Dígitos

Consideramos el conjunto de datos de dígitos manuscritos en los que hemos restringido a aquellos cuya etiqueta es 4 u 8. Vamos a utilizar las características *intensidad promedio* y *simetría* igual que hicimos en la práctica anterior para clasificar dígitos. Comenzamos planteando el problema de clasificación , identificando sus elementos:

- Tendremos el conjunto de datos $\mathcal{X}$. Este se crea del siguiente modo: primero, se seleccionan aquellas parejas $(x_i, y_i)$ en las cuales $y_i \in \{4, 8\}$. Una vez seleccionados, nos quedamos con dos características que son simetría e intensidad promedio, por lo que nos estamos quedando con $x_i \in \mathbb{R}^2$. Para homogeneizar los datos como hicimos en la práctica

anterior, añadiremos un 1 como primera componente de cada dato $x_i$, por lo que nuestro espacio final de características será:

$$\mathcal{X} = \{1\} \times \mathbb{R}^2$$

- Tendremos las etiquetas $y_i \in \{4, 8\}$ que indicarán para un vector de características $x_i$, el dígito al que pertenece, ya sea el 4 u el 8. Para la comodidad en la clasificación, identificaremos el 4 como la etiqueta $-1$, y el 8 como la etiqueta 1. Por tanto, $\mathcal{Y} = \{-1, 1\}$.

- Juntando los dos anteriores, obtenemos nuestro conjunto de entrenamiento que será:

$$\mathcal{D} = \{(x_n, y_n) \ : x_n \in \mathcal{X}, y_n \in \mathcal{Y}, n = 1, \ldots, N\}$$

  con un total de $N = 1194$ datos.

- Tenemos la función objetivo $f : \mathcal{X} \to \mathcal{Y}$ es la función de etiquetado que queremos encontrar, es desconocida.

- La clase de funciones $\mathcal{H}$ que usaremos para buscar nuestra $f$ sí que la conocemos, y será la clase de las funciones lineales de $\mathbb{R}^3$ en $\mathbb{R}$, que además clasifican a los datos al conjunto $\{-1, 1\}$. Por tanto, podemos definirla formalmente como:

$$\mathcal{H} = \left\{h : \mathbb{R}^3 \to \mathbb{R} \mid h(x) = signo(w^T x),\ w \in \mathbb{R}^3\right\}$$

- La técnica que se usaremos es la minimización del riesgo empírico (ERM) para hallar la función $g$ en nuestra clase de funciones que haga que el error de la muestra sea mínimo. Este error, recordamos que si $h \in \mathcal{H}$, es:

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

- Como algoritmos $\mathcal{A}$, usaremos un modelo de regresión lineal, que luego intentaremos mejorar utilizando `PLA-Pocket`, que será comentado brevemente más adelante.

- Para evaluar la bondad de nuestro modelo, tenemos un conjunto de *test* dado, que tiene 366 datos definidos de la misma forma que se definen para el conjunto de entrenamiento.

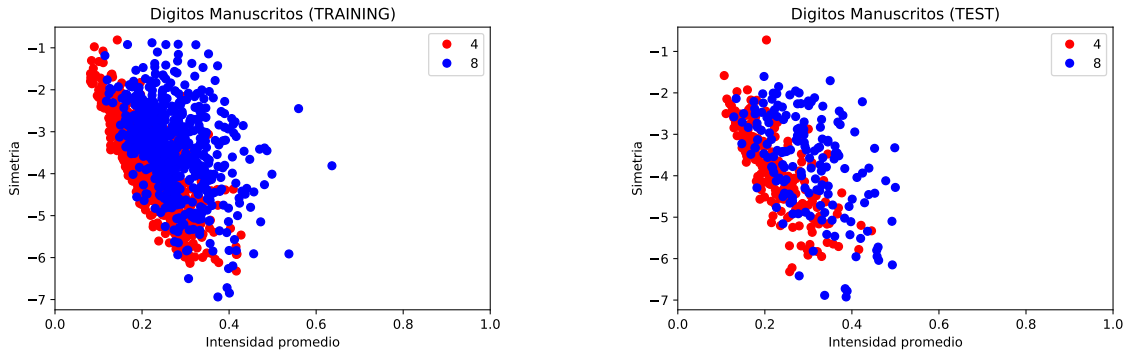
Con estos elementos, queda bien definido nuestro problema y podemos pasar a explicar los modelos que usaremos.

1. Como modelo de **regresión lineal**, utilizaremos el algoritmo de **pseudoinversa**, ya que lo utilizamos en la práctica anterior y, aunque no esté computado por la descomposición *SVD*, para este conjunto de datos pequeño es más que suficiente el código que ya teníamos implementado.

2. Aplicaremos posteriormente el algoritmo PLA-Pocket. Este no es más que el mismo **PLA** que hemos aplicado en los apartados anteriores, solo que almacenando .[en] el bolsillo

   el vector de pesos que se ha obtenido en alguna de las iteraciones y es el mejor en toda la muestra, es decir, el que menor error de clasificación nos proporciona. Esta solución, por supuesto, no soluciona el problema de que si los datos no son linealmente separables entonces no se encontrará el hiperplano separador. Sin embargo, se consigue que la solución no proporcione peores resultados conforme avanzan las iteraciones, es decir, no empeore con el tiempo.

Vamos primeramente a visualizar los conjuntos de entrenamiento y de test, para ver cómo son nuestros conjuntos de datos. El resultado es el siguiente:



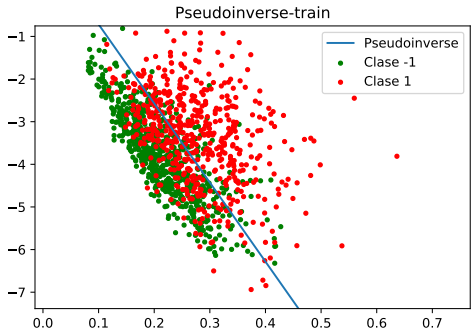

(a) Conjunto de Train.

(b) Conjunto de Test.

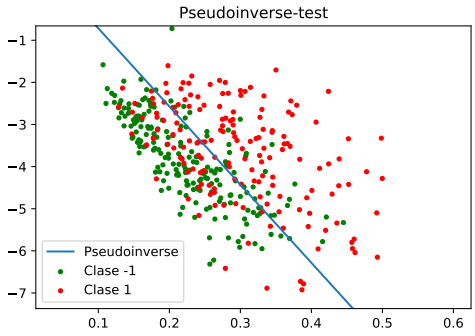
Figura 9: Conjuntos de train y test para el problema de clasificación de dígitos 4 y 8.

Como podemos observar, estos datos son claramente **NO separables**. El algoritmo de *pseudoinversa* nos proporcionará el mínimo teórico que podemos alcanzar. Sin ambargo, si quisiésemos aplicar aquí el algoritmo `PLA` tendríamos un problema, pues nunca encontraríamos una solución debido a la no separabilidad de los datos. Es por ello que aplicar `PLA-Pocket` ahora tiene bastante sentido, pues podemos apreciar si es una mejora al algoritmo `PLA`,pues este podría obtener una solución que sea razonablemente parecida a la que nos da el algoritmo de pseudoinversa.

Primero, observamos cómo se comporta el algoritmo de pseudoinversa sobre los datos:




(a) Clasificación usando pseudoinversa en train.



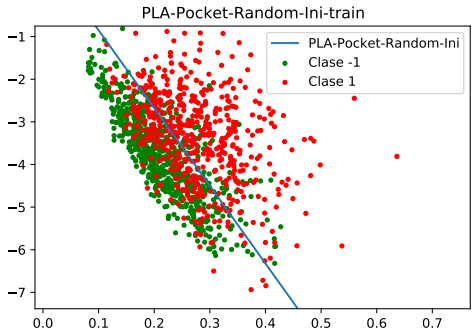

(b) Clasificación usando pseudoinversa en train.

Figura 10: Clasificación usando pseudoinversa en train y test.

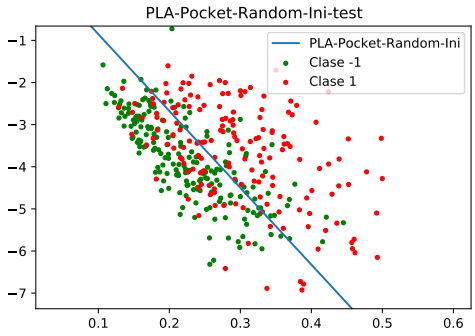
Además, podemos calcular el *accuracy* en ambos conjuntos para pseudoinversa, para en el futuro comparar. El resultado es:

```
El porcentaje de acierto de pseudoinversa es:
  Train: 0.7721943048576214
  Test: 0.7486338797814208
```

Los resultados no son tan buenos como serían si los datos fuesen separables. Vamos a ver qué ocurre cuando ejecutamos el algoritmo `PLA-Pocket`. Como sabemos que los datos son no separables, realizamos una implementación de este algoritmo que solo tiene en cuenta que se haga el número máximo de iteraciones y se devuelven los pesos que dan menor error. Comenzamos inicializando los pesos del algoritmo de forma aleatoria. Hay que notar que el tiempo de ejecución aumenta muy considerablemente, al tener que estar calculando cada vez que se modifica el vector de pesos $w$ el error sobre la muestra para ver si es el mejor vector de pesos actual. Es por ello que reducimos el número de iteraciones a realizar a un máximo de 500. El resultado gráfico es el siguiente:




(a) Clasificación usando PLA-Pocket en train.




(b) Clasificación usando PLA-Pocket en test.

Figura 11: Clasificación usando PLA-Pocket con inicialización aleatoria en train y test.

Podemos ver como el ajuste parece también razonablemente bueno. Podemos ver el error tanto en la muestra como en el conjunto de test que produce el algoritmo:

```
El porcentaje de acierto de PLA-Pocket pesos iniciales aleatorios es:
  Train: 0.7897822445561139
  Test: 0.7486338797814208
```

Como podemos apreciar, hemos mejorado un poco el porcentaje de acierto y hemos mantenido el

Por último, una buena idea para `PLA-Pocket` puede ser inicializar el algoritmo con los pesos que nos proporciona la regresión lineal y ver si es capaz de mejorar estos