Práctica 1 - Aprendizaje Automático

Luis Miguel Guirado Bautista - Universidad de Granada

3 de abril de 2022

Índice

Ejercicio I

Búsqueda iterativa de óptimos

1 Algoritmo de gradiente descendente

El algoritmo de gradiente descendente tiene como objetivo **optimizar una función** moviendo un punto inicial w_0 a un óptimo local mediante un vector de forma **iterativa**. El vector encargado de mover nuestro punto w_j depende de la derivada de la función a optimizar. Entonces **si nuestra función no es derivable, entonces el algoritmo no es aplicable**.
En caso de tener más variables, dependerá de las derivadas parciales de cada una de las variables, de modo que el punto se moverá en el eje de la variable j según su derivada parcial.

La actualización del punto w viene dada por la siguiente expresión, siendo $\eta \in [0, 1]$ la *tasa de aprendizaje* de nuestro algoritmo y f la función a optimizar

$$w := w - \eta \frac{\partial f}{\partial w}$$

El algoritmo se detendrá y devolverá el valor w cuando se haya alcanzado el valor deseado de la función por debajo de un valor ε o cuando el número de iteraciones actual sea mayor que N.
Ahora vamos a formalizar la implementación de nuestro algoritmo.

function GRADIENT_DESCENT(w_0, η, f, ε, N)
    $iter \leftarrow 0$
    $w \leftarrow w_0$
    while $f(w) > \varepsilon$ **and** $iter \leq N$ **do**
        $iter \leftarrow iter + 1$
        $w \leftarrow w - \eta \cdot \frac{\partial f}{\partial w}$
    end while
    return $(w, iter)$
end function

2 Ejemplos prácticos

2.1 Función E

Siendo nuestra función $E(u, v)$:

$$E(u,v) \;=\; (uve^{(-u^2-v^2)})^2 \;=\; u^2v^2e^{(-2u^2-2v^2)}$$

Podemos ver que E es derivable al ser composición de funciones continuas. Por tanto podemos aplicar el algoritmo de gradiente descendente a E.
La derivada parcial con respecto a u es:

$$\frac{\partial E}{\partial u} = 2uv^2e^{(-2u^2-2v^2)}(1-2u^2)$$

Y la derivada parcial con respecto a v es:

$$\frac{\partial E}{\partial v} = 2vu^2e^{(-2u^2-2v^2)}(1-2v^2)$$

Entonces **la expresión del gradiente** sería:

$$\frac{\partial E}{\partial w} = \left(\frac{\partial E}{\partial u}, \frac{\partial E}{\partial v}\right) = \left(2uv^2e^{(-2u^2-2v^2)}(1-2u^2)\ ,\ 2vu^2e^{(-2u^2-2v^2)}(1-2v^2)\right)$$

Ahora, considerando los siguientes parámetros, incluyendo nuestra función E y su gradiente $\frac{\partial E}{\partial w}$:

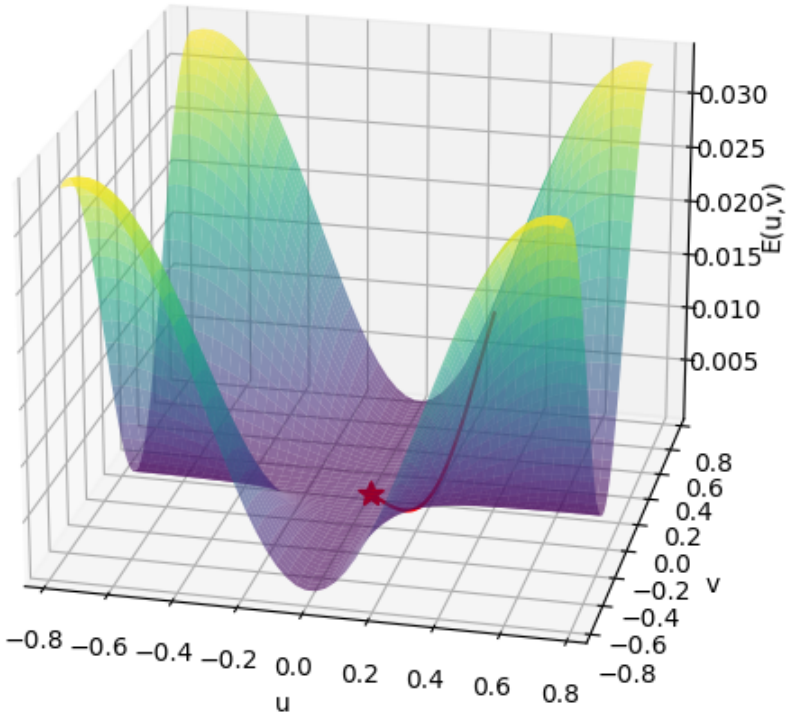
η	N	ε	w_0
0.1	10^9	10^-8	$(0.5, -0.5)$

Al realizar una ejecución de nuestro algoritmo con estos parámetros, obtenemos los siguientes resultados:

Número de iteraciones: 25117

Coordenadas alcanzadas: $w = (0.01, -0.01)$

Figura 1: Representación visual de la ejecución del algoritmo sobre E



En la figura 1 podemos ver la función dibujada en un gráfico 3D y el recorrido del algoritmo que desemboca en una punto estrellado, que representa el mínimo local alcanzado. En este primer ejemplo podemos ver que el algoritmo ha cumplido con su objetivo, ahora veamos un segundo ejemplo más complicado.

2.2 Función f

Ahora vamos a realizar los mismos pasos con una función $f(x, y)$:

$$f(x, y) = x^2 + 2y^2 + 2\sin(2\pi x)\sin(\pi y)$$

f es derivable ya que es composición de funciones continuas.
Calculamos sus derivadas parciales:

$$\frac{\partial f}{\partial x} = 2x + 4\pi\sin(\pi y)\cos(2\pi x)$$

$$\frac{\partial f}{\partial y} = 4y + 2\pi\sin(2\pi x)\cos(\pi y)$$

Por tanto el gradiente de f sería:

$$\frac{\partial f}{\partial w} = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right) = \left(2x + 4\pi\sin(\pi y)\cos(2\pi x)\ ,\ 4y + 2\pi\sin(2\pi x)\cos(\pi y)\right)$$

Ahora, con nuestro gradiente calculado, ejecutamos el algoritmo con los siguientes parámetros.

η	N	ε	w_0
0.01	50	–	$(-1, 1)$

Nos daría los siguientes resultados, con el resultado visual de la figura 2, más adelante:

Coordenadas alcanzadas: $w = (-1.217, 0.413)$

Valor alcanzado: $f(w) = -0.062$

Si cambiamos la tasa de aprendizaje η a 0.1, el resultado final empeora, con la figura 3:

η	N	ε	w_0
0.1	50	–	$(-1, 1)$

Coordenadas alcanzadas: $w = (-0.524, -0.255)$

Valor alcanzado: $f(w) = 0.187$

Figura 2: Ejecución del algoritmo sobre f con $\eta = 0.01$

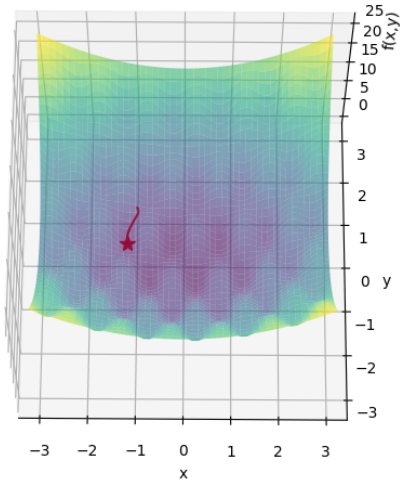
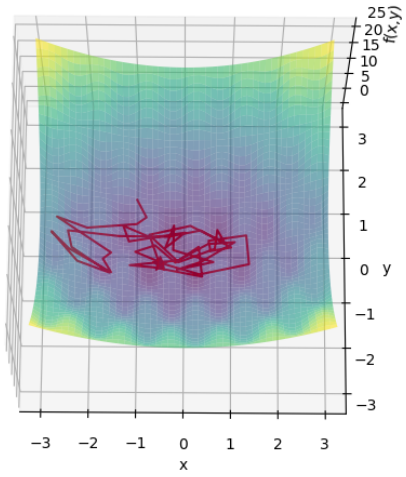


Figura 3: Ejecución del algoritmo sobre f con $\eta = 0.1$



En la figura 2 podemos ver que el algoritmo ha seguido un recorrido uniforme, encontrando un **mínimo local**.
Aunque en la figura 3, los pasos se han visto muy dispersos durante la ejecución del algoritmo con resultados peores que con una η menor, de manera que este valor de η no es adecuado para aplicarlo en la ejecución de este algoritmo.

En la siguiente tabla podemos ver otros ejemplos de ejecución, cambiando el punto inicial w_0 con el que se inicia el algoritmo y su tasa de aprendizaje η

	$\eta = 0.1$	$\eta = 0.01$
(-0.5,-0.5)	(-1.987, -0.005) = 3.947	(-0.244, 0.415) = -1.524
(1,1)	(0.557, -0.311) = 1.09	(0.731, 0.414) = -1.037
(2.1,-2.1)	(-1.638, -0.273) = 1.679	(1.665, -1.173) = 4.634
(-3,3)	(0.244, -0.324) = -1.43	(-2.189, 0.587) = 3.694
(-2,2)	(-0.813, 0.491) = 2.986	(-1.664, 1.171) = 4.634

Figura 4: Tabla de ejecuciones con distintas w_0 y η

3 Conclusión

¿Cuál es la verdadera dificultad de encontrar el mínimo global de una función arbitraria?
Podemos ver en la figura 4 que los resultados cambian al usar una η distinta en unos casos u otros. Los mejores casos con $\eta = 0.01$ son el primero y el segundo, mientras que el resto de casos son mejores con $\eta = 0.1$. El mejor caso en general ha sido el primero con $\eta = 0.01$. *¿Por qué?*
Porque el punto inicial era el más próximo al origen de coordenadas y el cual estaba más cerca del supuesto mínimo global de la función, ya que ha dado un resultado mejor que el resto. Aunque esto no pasa con $\eta = 0.1$ al ser un valor tan elevado que el algoritmo no llega a converger en esa zona. *¿Por qué han conseguido mejores resultados con una η más pequeña?*
Porque el punto estaba lo suficientemente cerca del mínimo global que en las siguientes iteraciones podía seguir un recorrido prácticamente uniforme y con pasos cada vez más cortos, llegando a converger en un mínimo. Esto también puede verse en el segundo caso, pero no en el resto de casos, ya que están suficientemente alejados del origen de manera que el valor de η no llega a ser suficientemente grande y no llegan a iterar lo suficientemente rápido antes de alcanzar el máximo de iteraciones.

En resumen La verdadera dificultad de este problema está en acertar con un punto inicial w_0 que esté lo suficientemente lejos o cerca (según η) que sea capaz de llegar al mínimo global en un máximo de N iteraciones que también sea acertada.
Una posible solución sería adaptar η de modo que sea directamente proporcional a la distancia recorrida, pero el único inconveniente es que también puede converger hacia mínimos locales. En mi opinión, podría ser una solución acertada si tenemos conocimiento sobre la función, es decir, saber donde podría estar el mínimo global, y en este caso adaptar η según la distancia con respecto al mínimo.

Ejercicio II

Regresión lineal

4 Números manuscritos

Siendo las matrices:

w: Los coeficientes del modelo lineal actual

$$\mathbf{w} = \begin{bmatrix} w_0 & w_1 & w_2 \end{bmatrix} \qquad w_i \in \mathbb{R}$$

X: Los datos de la muestra

$$\mathbf{X} = \begin{bmatrix} x_{10} & x_{11} & x_{12} \\ x_{20} & x_{21} & x_{22} \\ \vdots & \vdots & \vdots \\ x_{N0} & x_{N1} & x_{N2} \end{bmatrix}$$

Los elementos x_{i0} valen 1 y el par restante $(x_{i1}, x_{i2}) \in \mathbb{R}$ representan el nivel medio de gris y la simetría de cada número manuscrito i

y: Los datos objetivo, esto es, los valores que se pretenden alcanzar

$$\mathbf{y} = \begin{bmatrix} y_0 & y_1 & \dots & y_N \end{bmatrix} \qquad y_i \in \{-1, 1\}$$

4.1 Gradiente descendente estocástico

El planteamiento del problema es bastante similar al del ejercicio anterior. Esta vez, el objetivo es encontrar un modelo de regresión lineal **w** tal que sea capaz de distinguir unos elementos de otros dentro de la muestra **x** con el mínimo error posible. Al ser una variante estocástica, tendremos la muestra dividida en subconjuntos de un tamaño M equitativo, razonable y sin dejar algún dato sin escoger. En esto último, el algoritmo se encargará de escoger un valor adecuado, ya que tiene que ser divisor de N y mucho más pequeño.
Una posible hipótesis de modelo lineal será el siguiente:

$$h(\mathbf{x}_i) = \mathbf{w}^{\mathrm{T}}\mathbf{x}_i = sign(w_0 x_{i0} + w_1 x_{i1} + w_2 x_{i2})$$

Para medir el error entre la hipótesis y el valor objetivo, usaremos la fórmula del error cuadrático medio como función a optimizar:

$$E(\mathbf{x}_i) = (h(\mathbf{x}_i) - y_i)^2$$

Por tanto, consideraremos que el error dentro de la muestra E_{in} es:

$$E_{in}(\mathbf{w}) = \frac{1}{M}\sum_{i=1}^{M} E(\mathbf{x}_i) = (h(\mathbf{x}_i) - y_i)^2$$

Siendo esta nuestra función a minimizar.
Este es el auténtico objetivo del algoritmo en este ejercicio, es decir, queremos encontrar una recta que sea capaz de dividir unos números de otros **con el mínimo de errores**

Entonces, el gradiente a calcular es:

$$\frac{\partial E_{in}(\mathbf{w})}{\partial w_j} = \frac{2}{M}\sum_{i=1}^{M} x_{ij}\left(h(\mathbf{x}_i) - y_i\right)$$

No cambia nada sobre la actualización de la hipótesis, es decir: el punto se sigue actualizando de la misma manera

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

Y ahora formalizamos el algoritmo.

function SGD($\mathbf{X}$, $\mathbf{y}$, η, *max iters*)
  iter $\leftarrow 0$
  $w \leftarrow 0$
  data $\leftarrow \mathbf{X}$
  batchsize $\leftarrow 2$
  while (N % *batchsize*) $\neq 0$ **and** *batchsize* $<$ ($N \div 2$) **do**
    batchsize $\leftarrow$ *batchsize* $+ 1$
  end while
  number of batches $\leftarrow N \div$ *batchsize*
  do
    data $\leftarrow$ SHUFFLE(*data*)
    minibatches $\leftarrow$ SPLIT(*data*, *number of batches)*
    for *minibatch* $\in$ *minibatches* **do**
      iter $\leftarrow$ *iter* $+ 1$
      $w \leftarrow w - \eta \cdot \frac{\partial E_{in}(\mathbf{w})}{\partial \mathbf{w}}$
    end for
  while *iter* $\leq$ *max iters*
  return w
end function

Denotamos que este algoritmo busca un tamaño de subconjunto:

$$batchsize \in [2, N \div 2] \in \mathbb{N} \text{ t.q } N \text{ div } batchsize$$

Podemos poner un valor minimo del algoritmo más grande, pero puede influir negativamente en las actualizaciones de w. Al trabajar con el mismo caso, siempre se dará que $batchsize = 7$, de modo que tendremos 227 *batches*. Además, por simplicidad, asumiremos que $max\ iters = 400$.

¿Y si cambiaramos sus valores?
El resultado de la ejecución empeoraría con un tamaño de subconjunto más grande, además de su tiempo de ejecución. Demostrado experimentalmente.

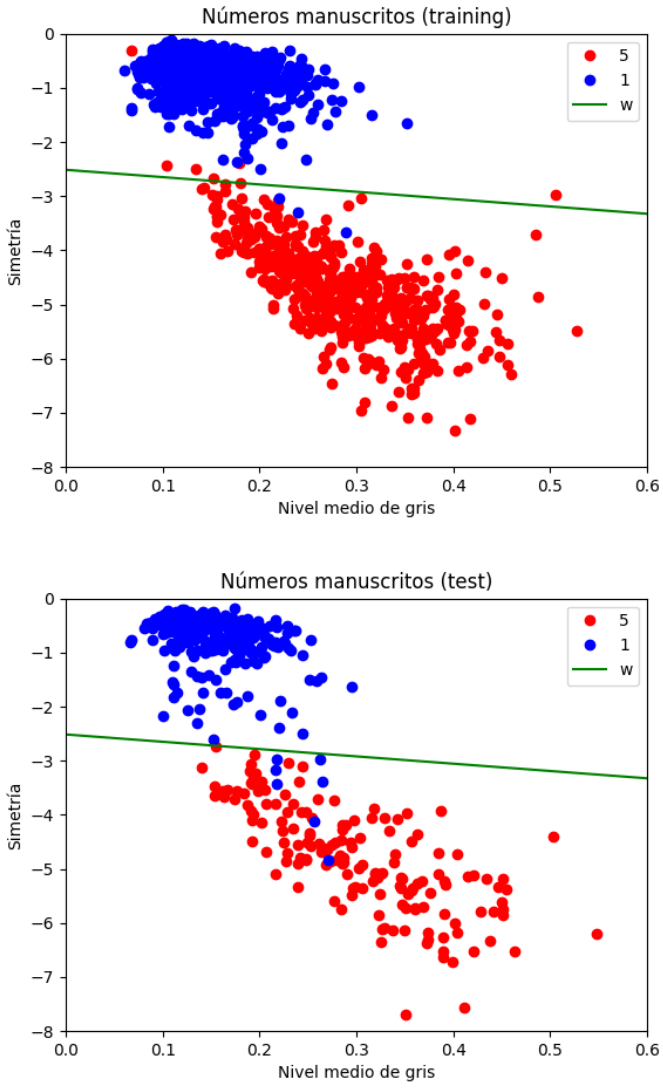
Ejecutando SGD con $\eta = 0.2$, obtenemos los siguientes resultados:

$$\mathbf{w} \approx \begin{bmatrix} -2.514 & -0.271 & -1.080 \end{bmatrix}$$

	Error	**Fallos**
Training (in)	0.128	5
Test (out)	0.094	10

Podemos ver el modelo gráficamente en la figura 5, en la página siguiente.
Si aumentamos η la recta que representa al modelo lineal baja drásticamente, tanto de forma que la importancia del valor que posea

Figura 5: Representación de **w** y los datos de entrenamiento y de test, respectivamente



4.2 Pseudoinversa

Este es un método que no necesita iterar, ya que solo se basa en operaciones con matrices. Esto es, podemos encontrar un modelo lineal a nuestro problema sin necesidad de iterar y sin necesidad de escoger una η.

$$\mathbf{w} = \mathbf{X}^{\dagger}\mathbf{y}$$

La única complejidad de este método se encuentra en $\mathbf{X}^{\dagger}$, la pseudoinversa de $\mathbf{X}$.

$$\mathbf{X}^{\dagger} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$$

Ya que la multiplicación $(\mathbf{X}^T\mathbf{X})^{-1}$ puede llegar a ser inviable computacionalmente debido al tamaño que puede llegar a tener $\mathbf{X}$ o la forma que tenga la matriz de manera que directamente no se pueda realizar la multiplicación, que se da prácticamente casi siempre.
Aplicaremos descomposición en valores singulares, resolviendo el problema de la forma de la matriz:

$$\mathbf{X} = \mathbf{UDV}^T$$

$$\mathbf{X}^T\mathbf{X} = \mathbf{VDDV}^T$$

Esto último puede hacerse en Python gracias al método de NumPy: `numpy.linalg.svd()`, que devuelve las matrices $\mathbf{U}$, $\mathbf{D}$, y $\mathbf{V}^T$ Es importante realizar la transformación siguiente antes de proceder:

```
D = np.diag(d)
```

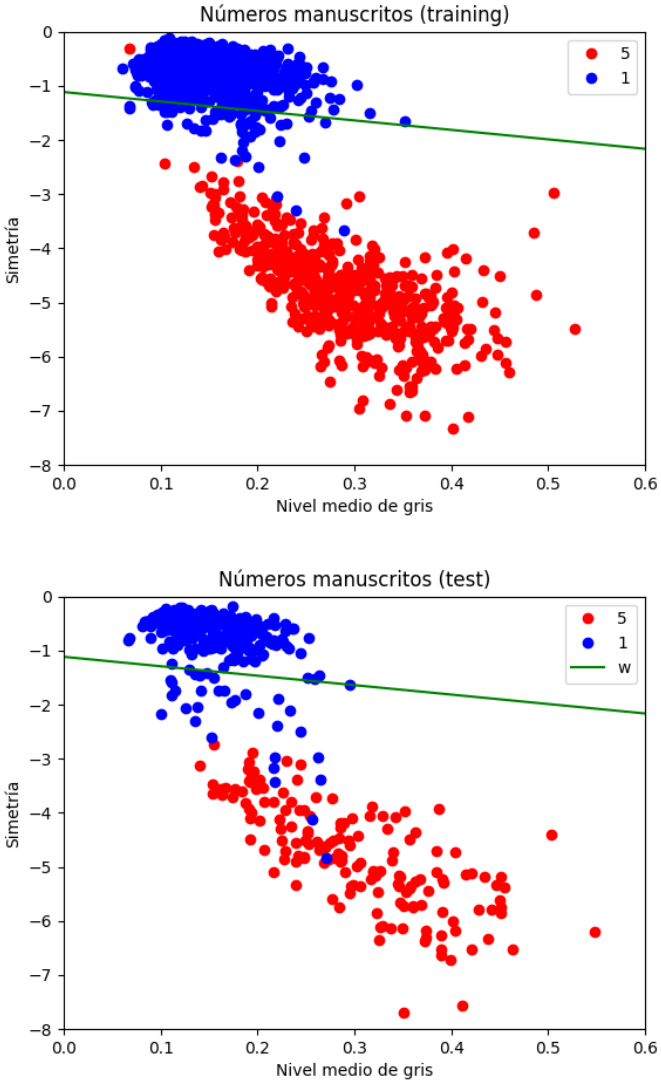
Siendo `d` la segunda matriz devuelta por el método anterior para el SVD.

Una vez calculado $\mathbf{w}$, tenemos los siguientes resultados:

$$\mathbf{w} \approx \begin{bmatrix} -1.115 & -1.248 & -0.497 \end{bmatrix}$$

	Error	**Fallos**
Training (in)	0.02	8
Test (out)	0.066	7

Figura 6: Representación de **w** y los datos de entrenamiento y de test, respectivamente



5 Clasificación de datos uniformemente aleatorios

5.1 Modelo lineal

Ahora vamos a considerar que $\mathbf{X}$ es un conjunto coordenadas de $N = 1000$ puntos aleatorios dentro del rango $[-1, 1] \times [-1, 1]$ y tenemos que clasificarlos con respecto a una función f

$$f(x_1, x_2) = sign((x_1 - 0.2)^2 + x_2^2 - 0.6) \qquad f : \mathbb{R}^2 \to \{-1, 1\}$$

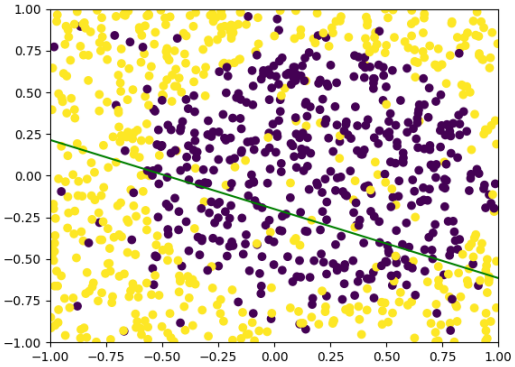
De manera que si el punto dado está dentro de la función dará -1, y 1 en caso contrario. Es importante decir que existe un 10% de ruido dentro de cada muestra, es decir, un 10% de los puntos estarán clasificados incorrectamente.
Usaremos SGD para la regresión lineal en este tipo de problemas. No obstante, si seguimos aplicando regresión lineal, tendremos los siguientes resultados (esta vez no hemos generado un conjunto test).

$$\eta = 0.1 \qquad \mathbf{w} \approx \begin{bmatrix} -0.2 & -0.044 & -0.369 \end{bmatrix}$$

	Error	**Fallos**
Training (in)	1.64	410

Podemos ver que son muchos fallos (mas o menos la mitad de la muestra), pero esto se debe a que la recta *corta por la mitad* el grupo de puntos dentro de f que necesita separar de los de fuera, esto se puede ver en la figura 7. Además, **el w resultante puede variar**.

Figura 7: Representación de $\mathbf{w}$ y los N puntos clasificados por f



Ahora vamos a realizar este mismo experimento 1000 veces y vamos a calcular el error medio de todas las muestras de entrenamiento y también generaremos una muestra adicional en cada iteración para el test.

Medias	**Training**	**Test**
Error	1.833	1.839
Fallos (%)	45.972	46.272

Entonces, ¿que podemos hacer para reducir el error? Tenemos que transformar el conjunto de características y aplicar regresión no linear, esto implica modificar $h(\mathbf{x})$ y por tanto $\mathbf{w}$ y la longitud de cada $\mathbf{x}$

5.2 Modelo no lineal

A cada vector de características $\mathbf{x}$ se le aplica una transformación ϕ y pasa a ser:

$$(1, x_1, x_2) \rightarrow (1, x_1, x_2, x_1x_2, x_1^2, x_2^2)$$

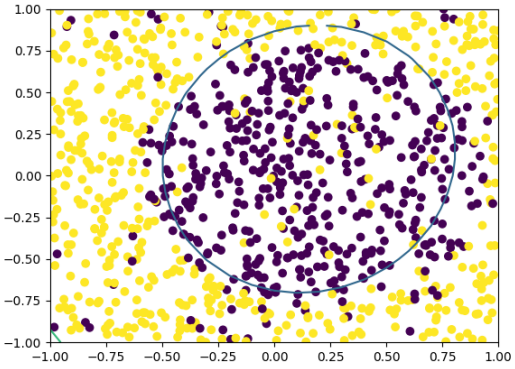
De modo que

$$h(\mathbf{x}) = sign(w_0 + w_1x_1 + w_2x_2 + w_3x_1x_2 + w_4x_1^2 + w_5x_2^2) \qquad h : \mathbb{R}^2 \rightarrow \{-1, 1\}$$

No cambia nada más, ahora si volvemos a repetir el experimento, los resultados mejoran:

Medias	**Training**	**Test**
Error	0.945	0.952
Fallos (%)	23.629	23.815

Figura 8: Representación de $\mathbf{w}$ no lineal y los N puntos clasificados por f



5.3 Conclusión

¿Qué modelo es más adecuado?
El modelo más adecuado es el no lineal. Podemos ver que los puntos dentro de f componen una forma elíptica, de manera que podemos transformar la ecuación lineal a una ecuación de la elipse, como hemos visto en esta página con ϕ. De modo que, en vez de clasificar el punto si está por encima o por debajo de la recta, podemos transformar este criterio en ver si el punto está dentro o fuera de la elipse formada por nuestro modelo.