APRENDIZAJE AUTOMÁTICO

Práctica 2 – Programación

Juan Pablo García Sánchez

Grupo 1
3º GII – UGR

Índice

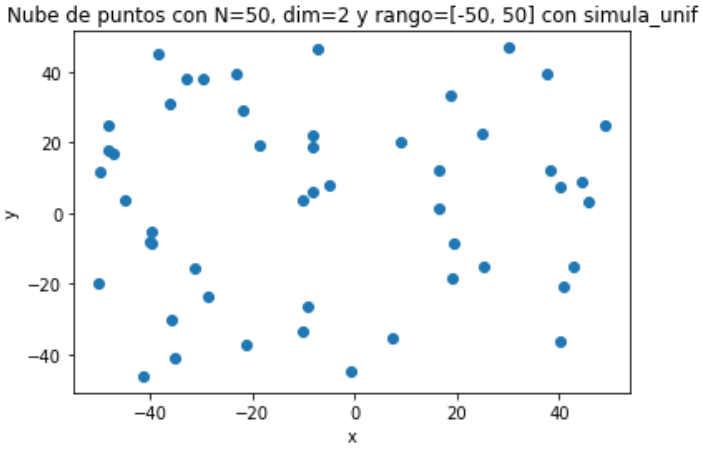
1 – Ejercicio sobre la complejidad de H y el ruido

Ejercicio 1

Dibujar una gráfica con la nube de puntos de salida correspondiente.

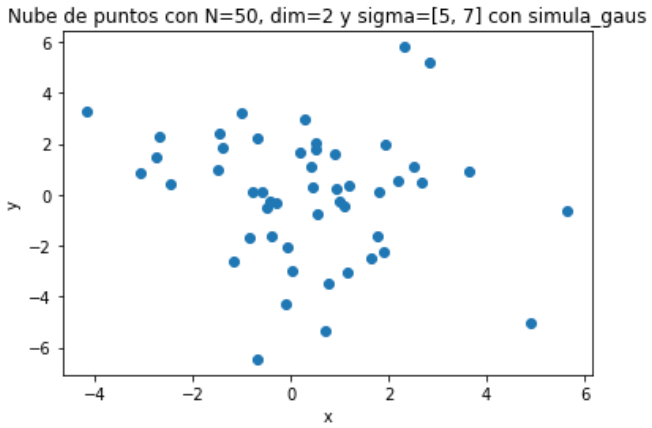
Apartado 1.a

Considere *N* = 50, *dim* = 2, *rango* = [-50, +50] con *simula_unif*(*N*, *dim*, *rango*).



Apartado 1.b

Considere *N* = 50, *dim* = 2 y *sigma* = [5, 7] con *simula_gaus*(*N*, *dim*, *sigma*).



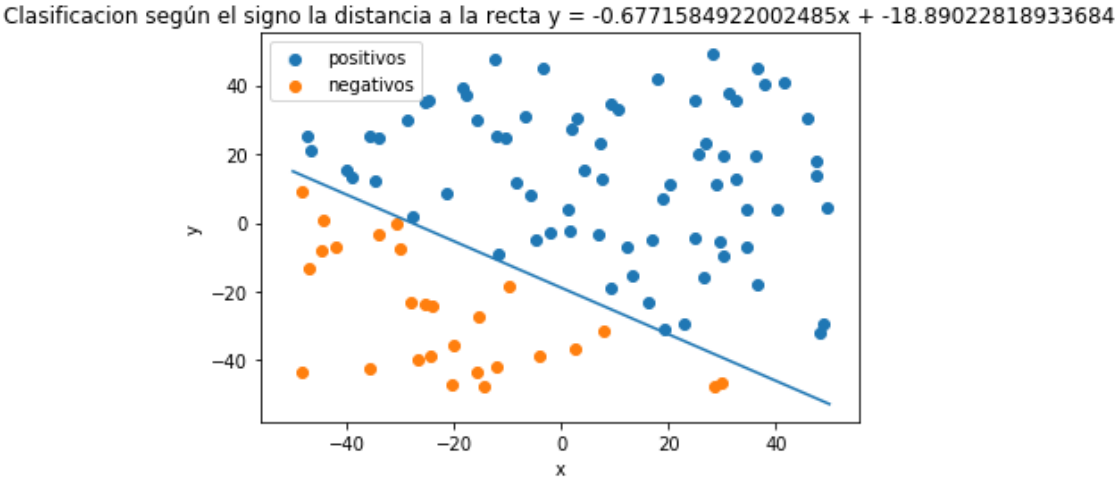
Ejercicio 2

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]) generar 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()*.

Apartado 2.a

Dibujar una gráfica donde los puntos muestren el resultado de su etiqueta junto con la recta usada para ello. (Observe que todos los puntos están bien clasificados respecto a la recta)

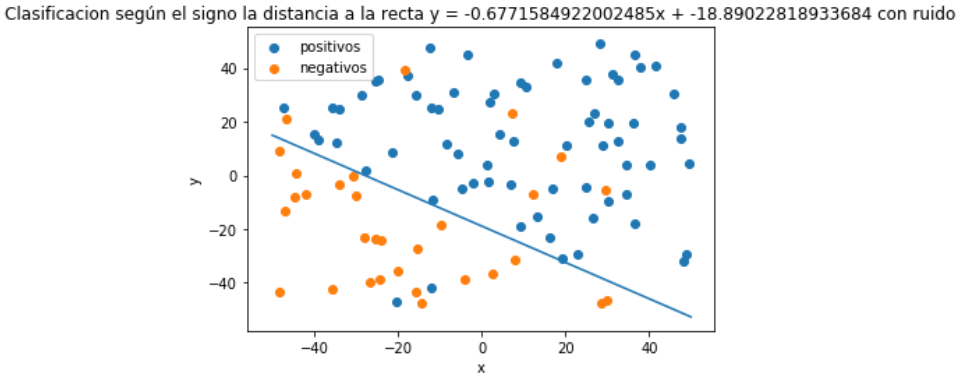
Para dibujar la recta he guardado 100 números equidistantes entre sí con *np.linspace* entre [-50, 50] que serán la coordenada *x* de los puntos de la recta. Luego, calculo la coordenada *y* de cada número con la ecuación de la recta.



Apartado 2.b

Modifique de forma aleatoria un 10% etiquetas positivas y otro 10% de negativas y guarde los puntos con sus nuevas etiquetas. (Ahora hay puntos mal clasificados con respecto de la recta).

Para añadir ruido simplemente he seleccionado un 10% de los puntos clasificados como positivos aleatoriamente, les he cambiado la etiqueta y los he movido del vector de positivos al de negativos. Lo mismo para el caso de los negativos. El primer *if* asegura que hay suficientes muestras para cambiar sus etiquetas, así que si la muestra es muy pequeña no se le añadirá ruido.



Apartado 2.c

Supongamos ahora que las siguientes funciones definen la frontera de clasificación de los puntos de la muestra en lugar de una recta:

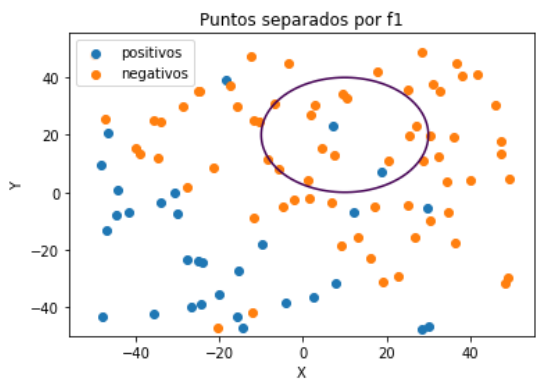
- $f(x, y) = (x - 10)^2 + (y - 20)^2 - 400$
- $f(x, y) = 0{,}5(x + 10)^2 + (y - 20)^2 - 400$
- $f(x, y) = 0{,}5(x - 10)^2 - (y + 20)^2 - 400$
- $f(x, y) = y - 20x^2 - 5x + 3$

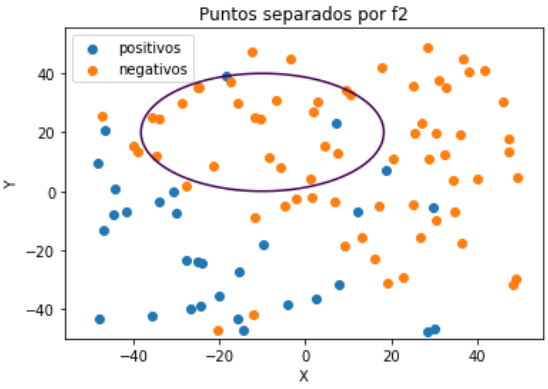
Visualizar el etiquetado generado en 2b junto con cada una de las gráficas de cada una de las funciones. Comparar las regiones positivas y negativas de estas nuevas funciones con las obtenidas en el caso de la recta. ¿Son estas funciones más complejas mejores clasificadores que la función lineal? Observe las gráficas y diga que consecuencias extrae sobre la influencia del proceso de modificación de etiquetas en el proceso de aprendizaje. Explicar el razonamiento.

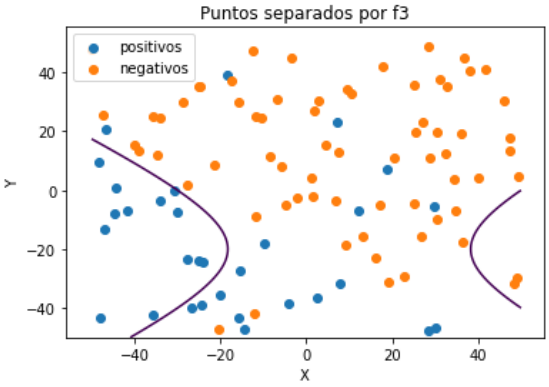
En ninguno de los 4 casos mejora la clasificación, de hecho, empeora. Esto se debe a que los puntos que queremos clasificar se pueden clasificar fácilmente con una función lineal y estas funciones cuadráticas no ayudan a solucionar el problema del ruido. Serían mejores clasificadores si los datos que queremos separar estuvieran distribuidos en una elipse.

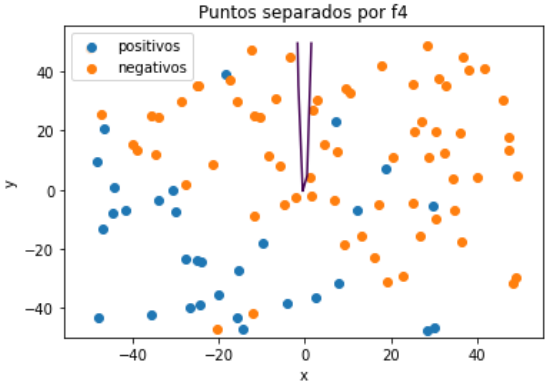
Si modificáramos las etiquetas en el proceso de aprendizaje, la función H se intentaría ajustar a este ruido, aunque es muy posible que no consigue clasificar todos los puntos bien porque esto daría lugar a overfitting y tendríamos un clasificador malo: se ajusta perfectamente a los datos de entrenamiento, pero da resultados horribles con muestras de prueba.

Para dibujar las elipses he utilizado el mismo método que para las rectas, pero esta vez creando también 100 números para la coordenada *y*. Así, con 100 números para la coordenada *x* y 100 para la *y* puedo calcular el valor en la elipse.


Puntos separados por f1
positivos
negativos
X
Y


Puntos separados por f2
positivos
negativos
X
Y


Puntos separados por f3
positivos
negativos
X
Y


Puntos separados por f4
positivos
negativos
x
y

2 – Modelos lineales

Algoritmo Perceptron

Implementar la función *ajusta_PLA*(*datos*, *max_iter*, *vini*) que calcula el hiperplano solución a un problema de clasificación binaria usando el algoritmo PLA. La entrada *datos* es una matriz donde cada item con su etiqueta está representado por una fila de la matriz, *label* el vector de etiquetas (cada etiqueta es un valor +1 o -1), *max_iter* es el número máximo de iteraciones permitidas y *vini* el valor inicial del vector. La función devuelve los coeficientes del hiperplano.

Los pasos que sigue el algoritmo son:

- Inicializa *contador* a 0 y el vector de pesos *w* a *vini*
- Mientras *contador* sea menor que *max_iter* y se produzcan cambios en el vector de pesos:
    - Por cada dato *x* calcula si la etiqueta que se obtiene con $w^T x$ es igual a la etiqueta que le corresponde. Si es igual, no se producen cambios y se pasa al siguiente dato. En caso contrario, el vector de pesos se modifica sumándole *label[i] * datos[i]*.

1) Ejecutar el algoritmo PLA con los datos simulados en los apartados 2a de la sección 1. Inicializar el algoritmo con: a) el vector cero y, b) con vectores de números aleatorios en [0, 1] (10 veces). Anotar el número medio de iteraciones necesarias en ambos para converger. Valorar el resultado relacionando el punto de inicio con el número de iteraciones.

a)

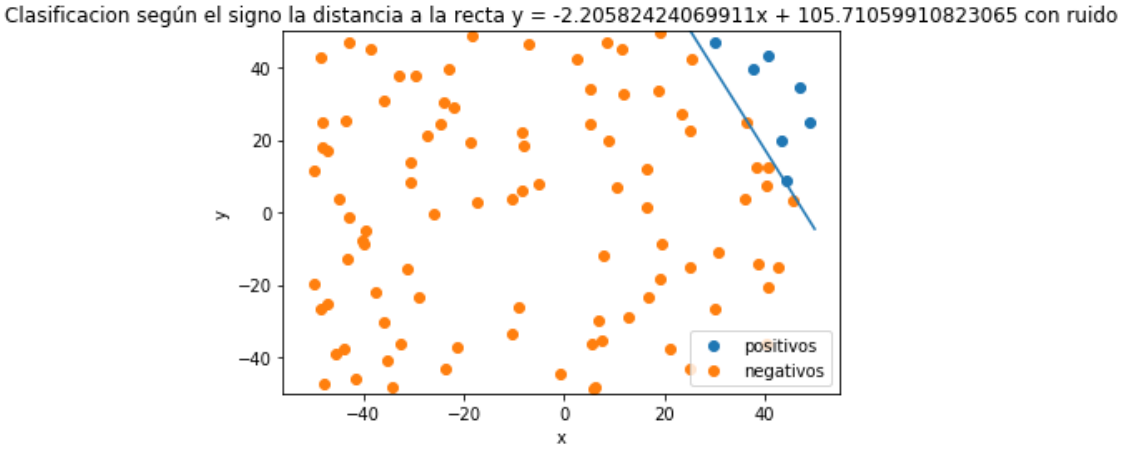
```
Apartado 2.a.1
Iteraciones:  444


--- Pulsar tecla para continuar ---
```

b)

```
Iteraciones:  47
Iteraciones:  42
Iteraciones:  1541
Iteraciones:  230
Iteraciones:  1588
Iteraciones:  1573
Iteraciones:  17
Iteraciones:  1519
Iteraciones:  1607
Iteraciones:  17
Valor medio de iteraciones necesario para converger: 818.1
```

Y la frontera calculada (pesos de la última iteración):



Como se puede ver, el número de iteraciones cuando se inicializa aleatoriamente es o mucho más pequeño o mucho más grande que si utilizamos un vector de solo ceros. Esto muestra que es muy importante tomar un vector de pesos iniciales bueno para calcular los pesos finales en el menor número de iteraciones posible. También vemos que, de media, necesita el doble de iteraciones inicializado aleatoriamente.

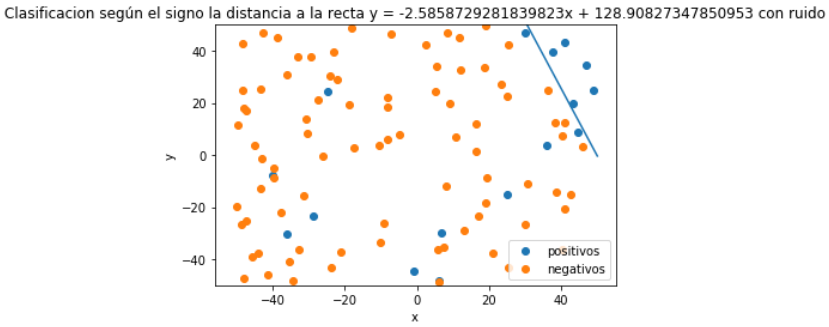
2) Hacer lo mismo que antes usando ahora los datos del apartado 2b de la sección 1. ¿Observa algún comportamiento diferente? En caso afirmativo diga cual y las razones para que ello ocurra.

En este caso, como los datos tienen ruido nunca puede llegar a conseguir que la recta clasifique todos los puntos bien. Por eso siempre agota las 10000 iteraciones, independientemente de cómo esté inicializado el vector de pesos.

```
Apartado 2.a.2
Iteraciones:  10000


--- Pulsar tecla para continuar ---

Iteraciones:  10000
Iteraciones:  10000
Iteraciones:  10000
Iteraciones:  10000
Iteraciones:  10000
Iteraciones:  10000
Iteraciones:  10000
Iteraciones:  10000
Iteraciones:  10000
Iteraciones:  10000
Valor medio de iteraciones necesario para converger: 10000.0
```



Regresión Logística

En este ejercicio crearemos nuestra propia función objetivo *f* (una probabilidad en este caso) y nuestro conjunto de datos *D* para ver cómo funciona regresión logística. Supondremos por simplicidad que *f* es una probabilidad con valores 0/1 y por tanto que la etiqueta *y* es una función determinista de x.

Consideremos $d = 2$ para que los datos sean visualizables, y sea $X = [0, 2] \times [0, 2]$ con probabilidad uniforme de elegir cada $x \in X$. Elegir una línea en el plano que pase por X como la frontera entre $f(x) = 1$ (donde y toma valores +1) y $f(x) = 0$ (donde y toma valores -1), para ello seleccionar dos puntos aleatorios del plano y calcular la línea que pasa por ambos. Seleccionar $N = 100$ puntos aleatorios $\{x_n\}$ de X y evaluar las respuestas $\{y_n\}$ de todos ellos respecto de la frontera elegida.

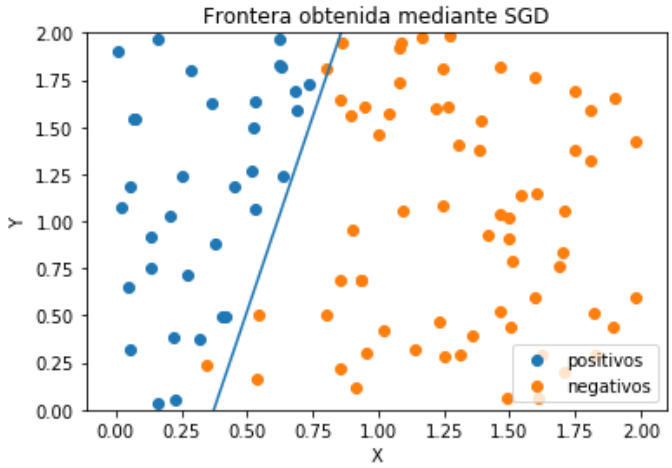
1) Implementar Regresión Logística (RL) con Gradiente Descendente Estocástico (SGD) bajo las siguientes condiciones:

- **Inicializar el vector de pesos con valores 0.**
- **Parar el algoritmo cuando $\|w^{(t-1)} - w^{(t)}\| < 0{,}01$, donde $w^{(t)}$ denota el vector de pesos al final de la época t. Una época es un pase completo a través de los N datos.**
- **Aplicar una permutación aleatoria 1,2,…,*N*, en el orden de los datos antes de usarlos en cada época del algoritmo.**
- **Usar una tasa de aprendizaje de $\eta = 0{,}01$**

El algoritmo *sgdRL*:

- Inicializa el contador a 0 y el vector de pesos "anteriores" a 1 para que la resta entre ese vector y el de pesos sea mayor a > 0.01
- Mientras la diferencia entre el vector de pesos de la época anterior y la actual sea mayor a > 0.01 o hasta que se hagan 50000 iteraciones:
    - Se copian los pesos en pesos anteriores
    - Se barajan los vectores de datos y sus etiquetas (usamos la variable estado para asegurarnos de que se hace la misma permutación).
    - Recorre los datos en mini-batches de tamaño 16 y actualiza los pesos utilizando SGD. Las etiquetas se predicen con RL (una función sigmoide que devuelve la probabilidad de que un dato tenga una etiqueta u otra).
    - Se calcula la diferencia entre estos pesos y los anteriores.

Por alguna razón, con $\eta = 0{,}01$ el algoritmo acaba en menos iteraciones (46) pero luego la recta que ha calculado ni siquiera corta al cuadrado $[0, 2] \times [0, 2]$. En cambio, con $\eta = 0{,}1$ encuentra una solución buena así que he decidido dejarlo con ese learning rate.



2) Usar la muestra de datos etiquetada para encontrar nuestra solución *g* y estimar E_{out} usando para ello un número suficientemente grande de nuevas muestras (>999).

Esta es la clasificación de una muestra de 1000 datos con los pesos calculados en el apartado anterior. Con esto calculamos el error:

```
Apartado 2.b

Eout:  0.038
```