Doble Grado en Ingeniería Informática y Matemáticas

APRENDIZAJE AUTOMÁTICO

Práctica 2: Programación

UNIVERSIDAD DE GRANADA

Alberto Estepa Fernández - *albertoestep@correo.ugr.es*

Abril de 2020

Índice

1. Ejercicio sobre la complejidad de H y el ruido

Hemos usado la plantilla proporcionada para la realización del ejercicio, modificando alguna parte si fuese necesario.

1.1. Apartado 1 (1 punto)

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

1. **Considere** $N = 50$**,** $dim = 2$**,** $rango = [-50, +50]$ **con** $simula\_unif()$**.**

   La función $simula\_unif(N, dim, rango)$ se proporcionaba ya implementada. Solo hemos obtenido los datos introduciendo los parámetros $N = 50$, $dim = 2$, $rango = [-50, +50]$.

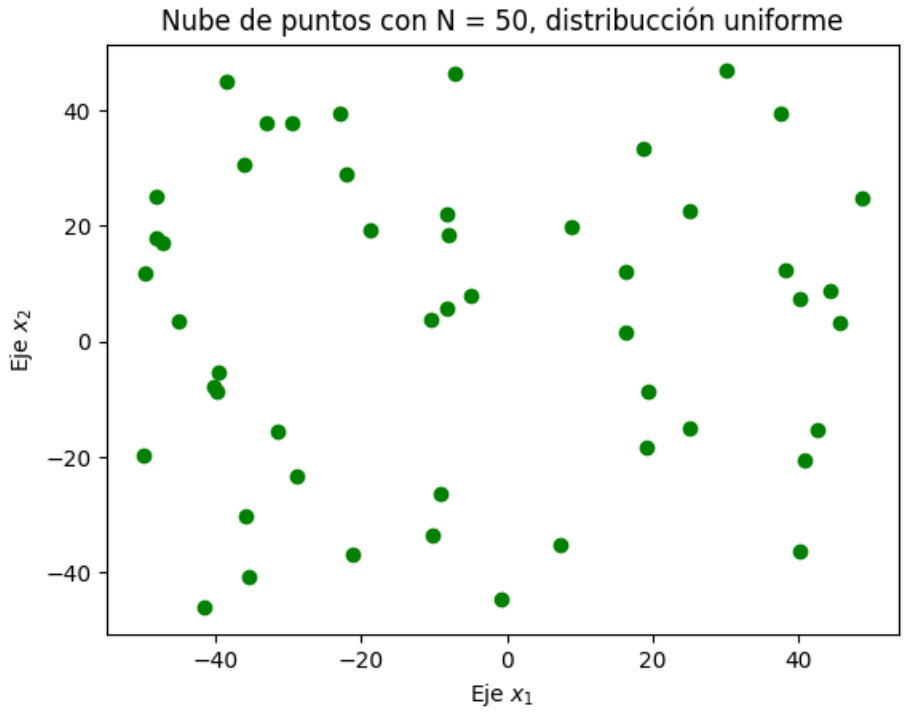

Figura 1: Nube de 50 puntos con distrubución uniforme

2. **Considere $N = 50$, $dim = 2$ y $sigma = [5, 7]$ con $simula\_gaus()$.**

   La función $simula\_gaus(N, dim, sigma)$ se proporcionaba ya implementada. Solo hemos obtenido los datos introduciendo los parámetros $N = 50$, $dim = 2$, $sigma = [5, 7]$.

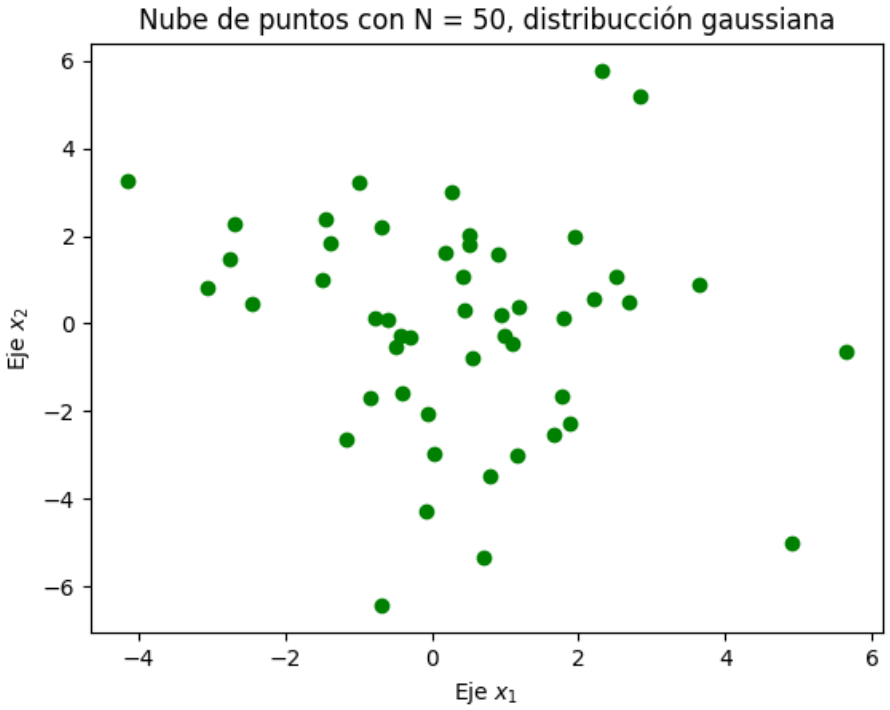

Figura 2: Nube de 50 puntos con distrubución gaussiana de $\mu = 0$ y $\sigma = 5$ y 7

1.2. Apartado 2 (1.5 puntos)

Vamos a valorar la influencia del rudio en la seleccion de la complejidad de la clase de funciones. Con 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 $simula\_recta()$.

1. **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 de la recta).**

   Para este ejercicio, dado que la función $sign()$ de $Numpy$ puede devuelvor los valores $\{-1, 0, 1\}$, hemos implementado la función signo donde si el valor es menor que 0 se le asigna el valor -1 y si es mayor o igual que 0 se le asigna el valor 1.

Por otro lado hemos usado la función ya proporcionada $simula\_recta([-50, 50])$, para calcular los coeficientes de la recta y asignar una etiqueta a los datos según dicha recta.

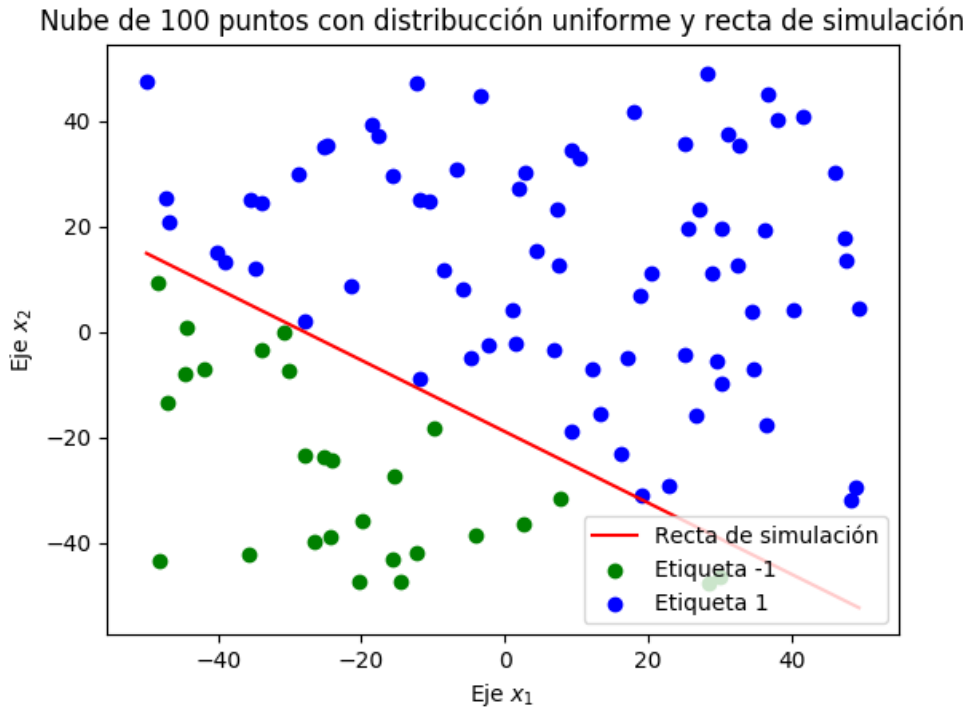

Figura 3: Nube de 100 puntos con distribucción uniforme y recta de simulación.

Comprobamos que todos los puntos están bien clasificados con respecto a la recta.

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

   Para este ejercicio hemos creado la función $introduce\_ruido(y, porcentaje)$ donde calculamos el número de puntos con etiqueta 1 y -1 a las que hay que cambiar de valor dicha etiqueta y luego realiza dicha acción con aleatoriamente sobre los puntos así etiquetados. Así el resultado conseguido es el siguiente:

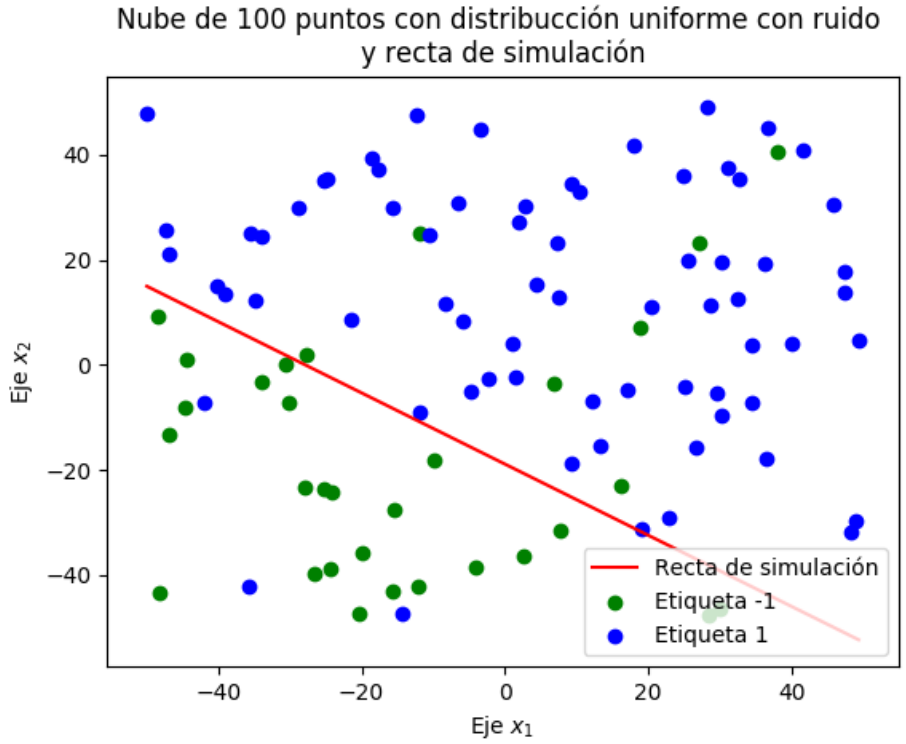

Figura 4: Nube de 100 puntos con distribucción uniforme con ruido y recta de simulación.

Comprobamos así que se han modificado las etiquetas de algunos puntos y ya no están bien clasificados con respecto a la recta.

1.3. Apartado 3 (2.5 puntos)

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

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.

Hemos usado la función proporcionada por al grupo 1 de prácticas para dibujar las gráficas resultantes. De esta forma podemos comparar de forma sencilla las regiones definidas como positivas y negativas por las diferentes funciones con las que trabajamos.

1. $f(x, y) = (x - 10)^2 + (y - 20)^2 - 400$

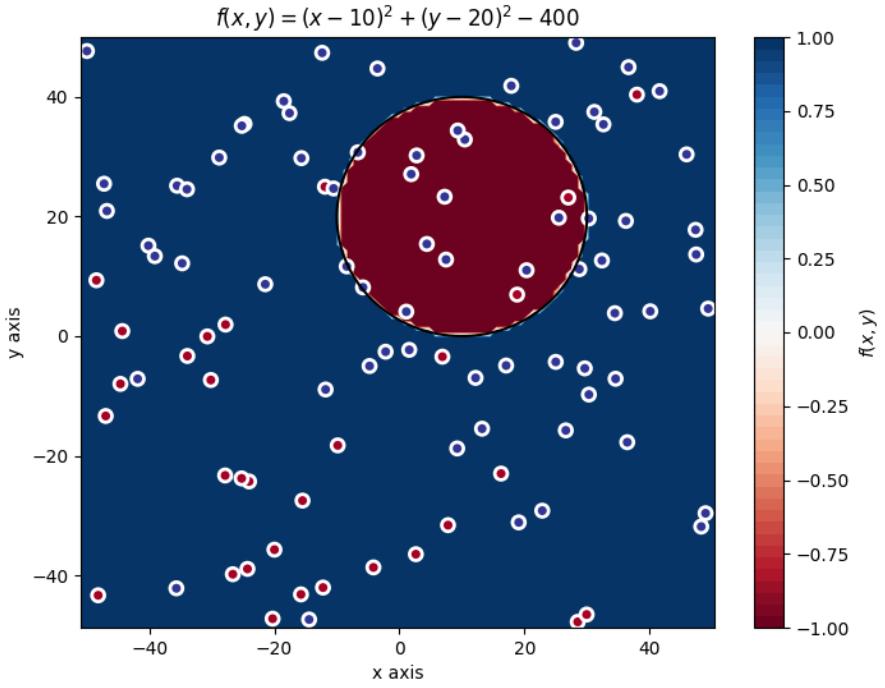

Figura 5: Función de clasificación $f(x, y) = (x - 10)^2 + (y - 20)^2 - 400$.

Vemos a simple vista que el resultado no es el más óptimo ya que hay una gran mezcla de puntos fuera de la elipse. Incluso hay puntos que no corresponden a dicha clasificación dentro de la elipse.

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

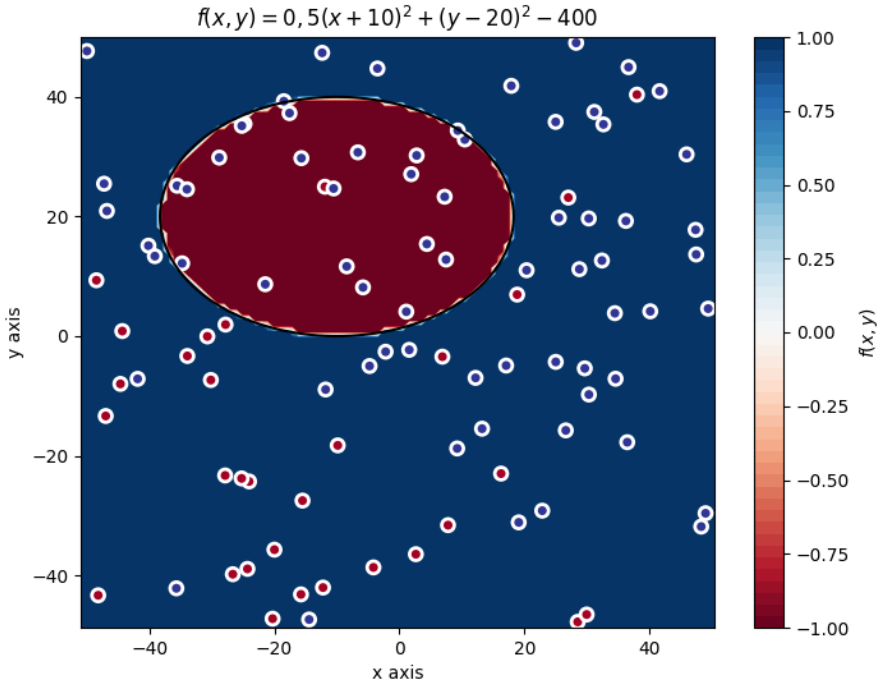

Figura 6: Función de clasificación $f(x, y) = 0, 5(x + 10)^2 + (y - 20)^2 - 400$.

De nuevo observamos a simple vista que el resultado no es el más óptimo posible ya que hay una gran mezcla de puntos fuera de la elipse. Incluso hay puntos que no corresponden a dicha clasificación dentro de la elipse.

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

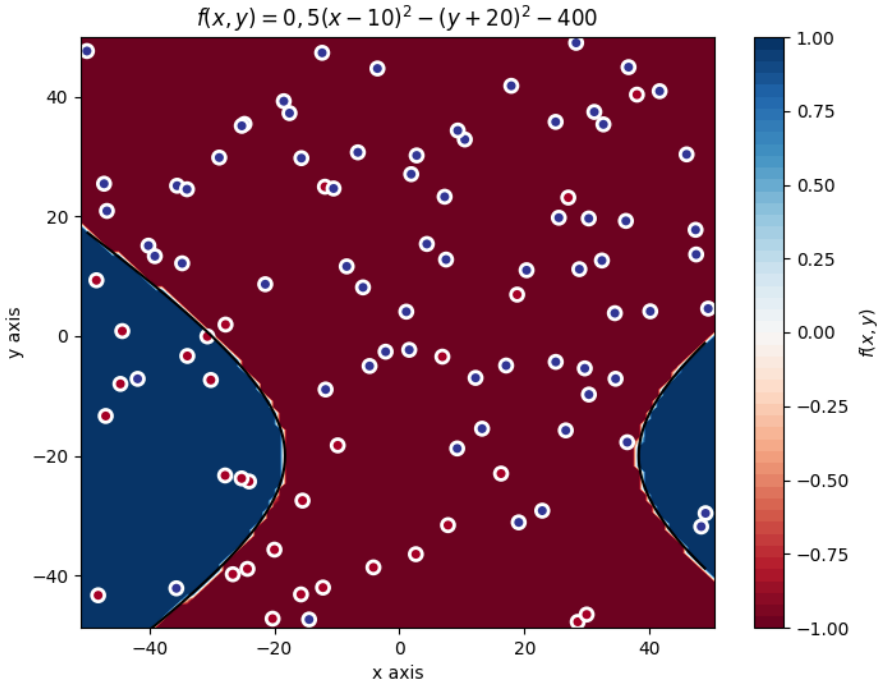

Figura 7: Función de clasificación $f(x, y) = 0, 5(x - 10)^2 - (y + 20)^2 - 400$.

Otra vez observamos que a simple vista el resultado no es el más óptimo posible ya que hay una gran mezcla de puntos fuera de la hipérbola. Incluso hay puntos que no corresponden a dicha clasificación dentro de ella.

4. $f(x, y) = y - 20x^2 - 5x + 3$

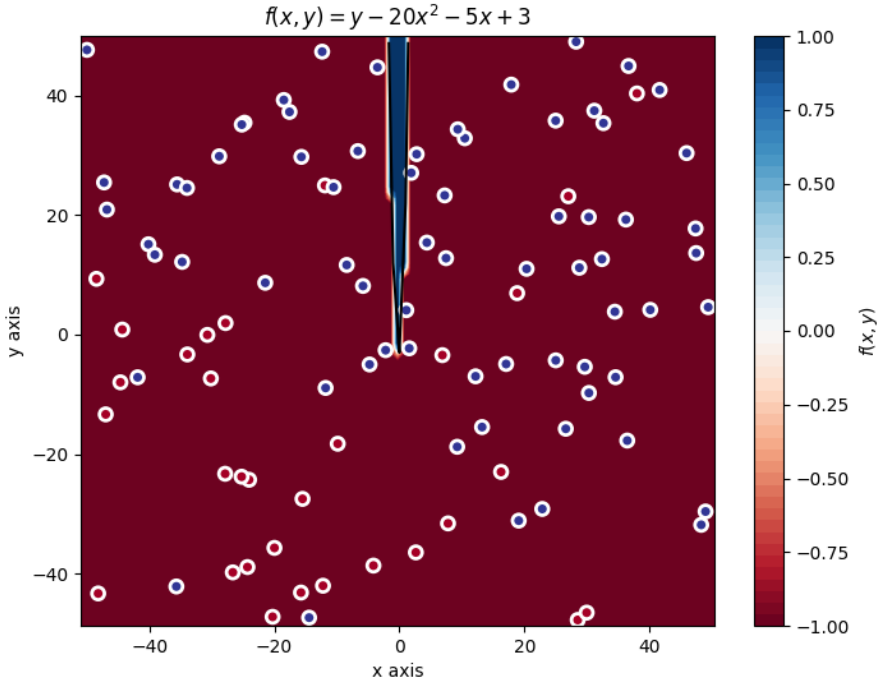

Figura 8: Función de clasificación $f(x, y) = y - 20x^2 - 5x + 3$.

En este caso, la parabola no es para nada un buen ajuste de los datos. Directamente no toma ningun valor como positivo.

Evaluación del rendimiento:

Hemos implementado una función que calcula la matriz de confusión de cada una de las posibles clasificaciones dadas. Los datos se pueden visualizar en el *.py* en la función *evaluar_rendimiento*. A continuación incluimos la matriz de confusión obtenida de la clasificación de la recta de la figura 4:

Matriz de confusión Recta		**Predicción**	
		-1	1
Real	-1	24	7
	1	3	66

Comprobamos que tenemos un porcentaje de acierto del 90 %, es decir una tasa de acierto del 0.9 (Como sabemos que tenemos 100 datos, y hay 24 puntos que hemos predicho correctamente con etiqueta -1 y 66 puntos que hemos predicho correctamente con la etiqueta 1. Así la suma nos da 90 puntos correctamente etiquetados de los 100 que hay en total). Ésto es lo esperado, pues previamente habíamos introducido ruido en la muestra perfectamente clasificada del 10 %.

A continuación incluímos las matrices de confusión de las demás funciones que queremos comparar con la recta anterior:

Matriz de confusión $f_1(x,y) = (x-10)^2 + (y-20)^2 - 400$		**Predicción** -1	1
Real	-1	2	29
	1	12	57

Esta matriz se corresponde con la figura 5. Comprobamos que tiene un 59 % de acierto. Es decir una tasa de acierto del 0.59

Matriz de confusión $f_2(x,y) = 0,5(x+10)^2 + (y-20)^2 - 400$		**Predicción** -1	1
Real	-1	1	30
	1	21	48

Esta matriz se corresponde con la figura 6. Comprobamos que tiene un 49 % de acierto. Es decir una tasa de acierto del 0.49

Matriz de confusión $f_3(x,y) = 0,5(x-10)^2 - (y+20)^2 - 400$		**Predicción** -1	1
Real	-1	20	11
	1	65	4

Esta matriz se corresponde con la figura 7. Comprobamos que tiene un 24 % de acierto. Es decir una tasa de acierto del 0.24

Matriz de confusión $f_4(x,y) = y - 20x^2 - 5x + 3$		**Predicción** -1	1
Real	-1	31	0
	1	69	0

Esta matriz se corresponde con la figura 8. Comprobamos que tiene un 31 % de acierto. Es decir una tasa de acierto del 0.31

Éstos valores de la tasa de acierto que hemos calculado corresponden a la exactitud o 'accuracy' de la clasificación que es una a métrica de los sistemas de clasificación que nos indica el porcentaje de predicciones que el modelo realizó correctamente.

Podríamos calcular otras métricas para obtener datos numéricos que nos ayuden en nuestra comparativa entre los diferentes modelos, pero es obvio que con éste queda claro que la recta de clasificación es bastante mejor que las otros 4 modelos.

Por otro lado el proceso de modificar las etiquetas influje claramente en el porcentaje de acierto de todos los clasificadores. Vemos así que, como lo normal es esperar cierto ruido en los datos, no debemos ceñirnos a modelos que se ajusten de forma perfecta a los datos, pues estaremos seguramente ante un ejemplo de sobreajuste.

2. Modelos Lineales

2.1. Apartado 1 (3 puntos)

Algoritmo Perceptrón: Implementar la función que calcula el hiperplano solución a un problema de clasificación binaria usando el algoritmo PLA $ajusta\_PLA(datos, label, max\_iter, vini)$**. 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.**

```
def adjust_PLA(datos, label, max_iter, vini):
    w = np.copy(vini)
    n_iteraciones = 0
    continuar = True
    while continuar:
        continuar = False
        n_iteraciones += 1
        for x, y in zip(datos, label):
            y_predicha = funcion_signo(w.dot(x.reshape(-1, 1)))
            if y_predicha != y:
                w += y * x
                continuar = True
        if n_iteraciones == max_iter:
            break

    return w, n_iteraciones
```

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.**

   Como hemos realizado el código en diferentes ficheros *.py* hemos decidido de guardar la semilla correspondiente en un fichero aparte al ejecutar el ejercicio 1 y recuperarla en el este fichero justo cuando necesitemos dichos datos. Esto es posible gracias a la librería *pickle* de Python. Ademas proporcionamos el fichero *estado_semilla.dat*, donde está guardado este estado de la semilla, en el zip de la entrega.

   a) Inicializando con el vector cero:
   Obtenemos un valor de $w = [661, 23.20241712, 32.39163606]$ como resultado de las 75 iteraciones, que es lo que tarda en converger.

   b) Inicializando con vectores aleatorios (10 veces): Obtenemos una media de 99.8 iteraciones hasta que el algoritmo converge. Además hemos normalizado los coeficientes obtenidos mediante $\|\cdot\|_1$ para poder visualizar los coeficientes debidamente regularizados y mediante un simple vistazo observamos que obtenemos coeficientes del modelo casi idénticos para todos los casos. (Estos valores

se incluyen en la ejecución del fichero $P2Ejercicio2.py$, ya que incluir tantos datos en esta memoria me parecia engorroso para el lector).

Es claro que el algoritmo del Percerptron separa de forma correcta nuestros datos y además converge con bastante rapidez. Esto no hace mas que confirmar aquello que hemos estudiado: el algoritmo Perceptrón separa perfectamente los datos si éstos son separables linealmente (como en nuestro caso).

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 cuál y las razones para que ello ocurra.**

   a) Inicializando con el vector cero:
   Obtenemos un valor de $w : [384, 24.2796483, 43.54404583]$. Hemos puesto un máximo de 1000 iteraciones, y ésto es justo lo que tarda en obtener dichos datos. A continuación lo explicaremos.

   b) Inicializando con vectores aleatorios (10 veces): Obtenemos una media de 1000 iteraciones de nuevo hasta que el algoritmo devuelve los datos. Este valor es idéntico al número de iteraciones máximo que hemos permitido al algoritmo. Además, como antes, hemos normalizado los coeficientes obtenidos mediante $\|\cdot\|_1$ para poder visualizar los coeficientes debidamente regularizados. Así, observamos que los datos aunque no varían de forma brusca, si que son más diferentes que los obtenidos con los anteriores datos. (Estos valores de nuevo se incluyen en la ejecución del fichero $P2Ejercicio2.py$, por el mismo motivo anteriormente descrito).

   Era obvio que nuestro algoritmo no iba a converger para dichos datos, incluso sin poner un límite máximo de iteraciones. Ésto es debido a que el algoritmo del Perceptrón no converge si los datos no son separables y al incluir ruido en la muestra hemos conseguido que nuestros datos no sean separables linealmente.

Hemos de decir que los datos pueden variar bastante con respecto a la semilla que usemos.

2.2. Apartado 2 (4 puntos)

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 $X = [0, 2]x[0, 2]$ 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.

Hemos generado los datos como en el ejercicio anterior pero ahora on los parámetros que se indican en el enunciado:

```
intervalo = [0, 2]
a, b = simula_recta(intervalo)
N = 100
datos = simula_unif(N, 2, intervalo)
```

Añadimos la columna de unos a los datos y evaluamos dichos datos con la etiqueta correspondiente

```
matriz_datos = np.hstack((np.ones((N, 1)), datos))
etiquetas = np.empty((N, ))
for i in range(N):
    etiquetas[i] = asigna_etiquetas(datos[i, 0], datos[i, 1], a, b)
```

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** $\eta = 0.01$

   Incluimos la implementación de la función de regresión logística con gradiente descendente estocástico tal y como nos piden:

   ```
   def sgdRL(datos, etiquetas, umbral=0.01, lr=0.01):
       N, dimension = datos.shape
       w = np.zeros(dimension)
       delta = np.inf
       while delta > umbral:
           indices = np.random.permutation(N)
           w_anterior = np.copy(w)
           for indice in indices:
               w = w - lr * gradiente(datos[indice],
                   etiquetas[indice], w)
           delta = np.linalg.norm(w_anterior - w)
       return w
   ```

   donde la función del gradiente es la siguiente:

   ```
   def gradiente(x, y, w):
       return -y * x/(1 + np.exp(y * w.dot(x)))
   ```

Incluímos una gráfica del resultado obtenido:

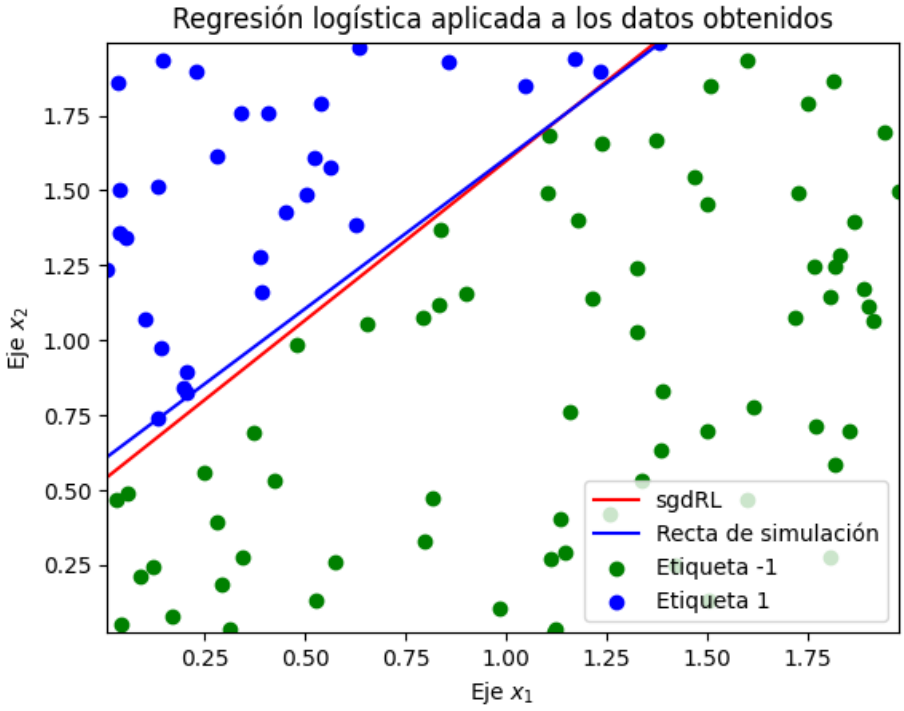

Figura 9: Aplicación de la regresión logística a los datos.

Vemos que se separa la regresión de forma correcta los datos y es bastante similar a la recta que determina realmente los puntos.

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)**.**

   Hemos generado 1000 puntos de muestra que serán los test. Aplicando el error de máxima verosimilitud hemos obtenido un error de 0.080054867464555 sobre 1, que es bastante correcto. Mientras que si aplicamos el error de tasa de fallos en la clasificación obtenemos un error de 0.009000000000000008 sobre 1. Así comprobamos que el modelo obtenido es bastante bueno y separa los datos de forma correcta.

3. Bonus (1.5 puntos)

Clasificación de Dígitos. Considerar el conjunto de datos de los dígitos manuscritos y seleccionar las muestras de los dígitos 4 y 8. Usar los ficheros de entrenamiento (training) y test que se proporcionan. Extraer las características de intensidad promedio y simetría en la manera que se indicó en el ejercicio 3 del trabajo 1.

1. **Plantear un problema de clasificación binaria que considere el conjunto de entrenamiento como datos de entrada para aprender la función g.**

   Como se indica, tenemos la característica de intensidad promedio y la característica de simetría para identificar dos clases: dígitos 4 y 8.

   Nosotros vamos a representar con la etiqueta -1 los puntos de la clase *dígito 4* y con la etiqueta 1 los puntos de la clase *dígito 8*.

   Aprendemos la función g usando el algoritmo PLA-Pocket calculando previamente una aproximación de los coeficientes del modelo partir un algoritmo de regresión lineal como es el algoritmo de la pseudoinversa realizado en la práctica 1.

2. **Usar un modelo de Regresión Lineal y aplicar PLA-Pocket como mejora. Responder a las siguientes cuestiones.**

   - **Generar gráficos separados (en color) de los datos de entrenamiento y test junto con la función estimada.**

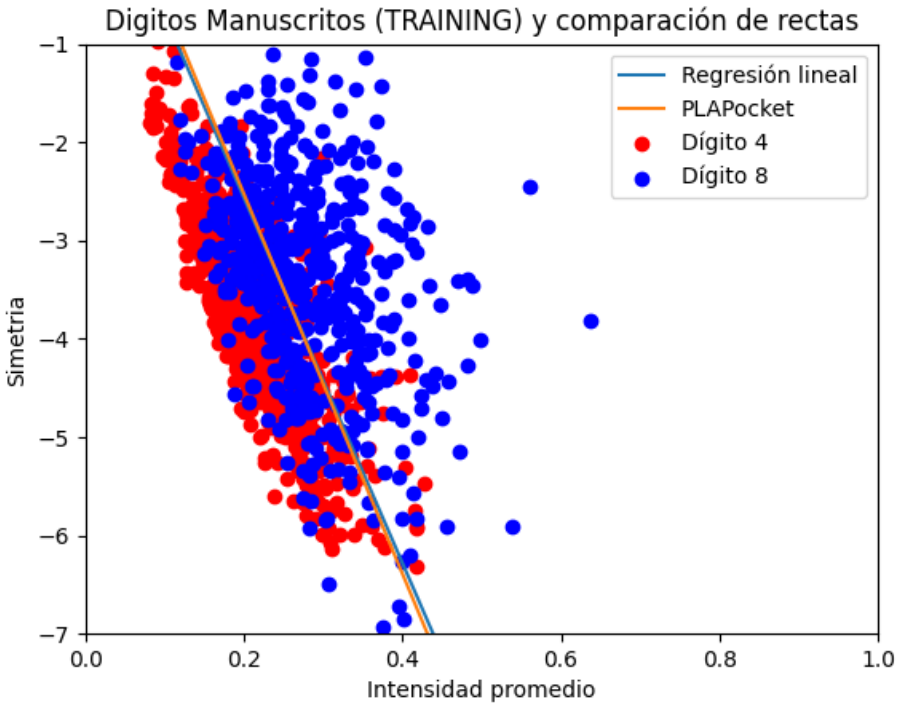

Figura 10: Datos de entrenamiento junto con la función estimada.

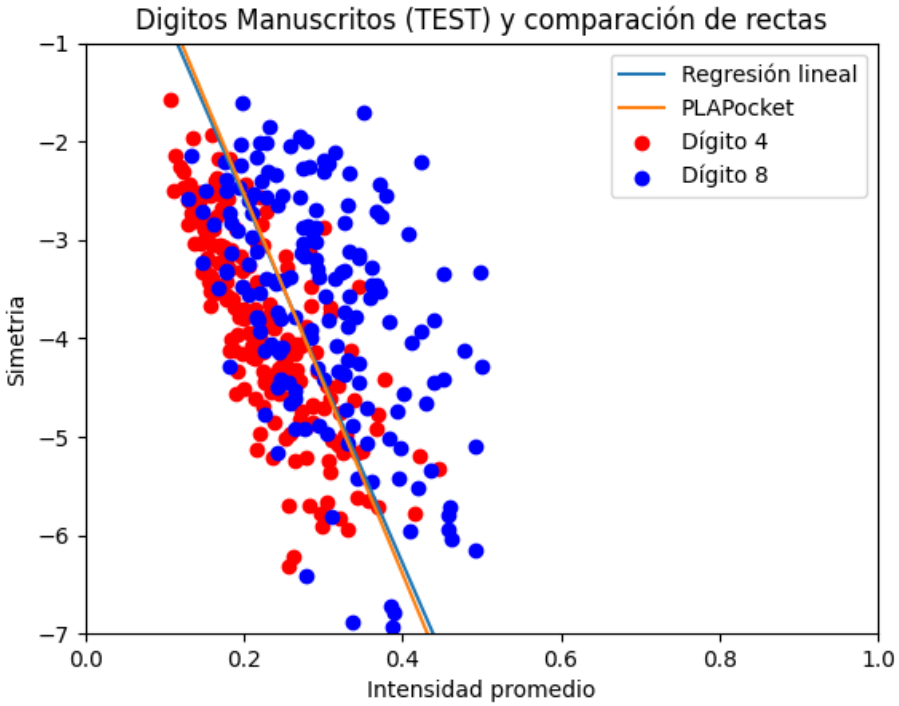

Figura 11: Datos de test junto con la función estimada.

Hemos implementado el algoritmo PLA_Pocket tal y como se proporciona en las diapositivas de teoría. Para más información, está implementado en el fichero *P2Bonus.py*.

Por otra parte hemos usado como algoritmo para el cálculo de la regresión lineal, el algoritmo de la pseudoinversa.

Así, los resultados obtenidos se deben a la aplicación del algoritmo de la pseudoinversa y a partir de éstos aplicarle directamente el algoritmo de PLA_Pocket implementado. Le hemos puesto un máximo de iteraciones de 1000, aunque si se aumentase dicho número seguramente los resultados mejoren.

Además, observando los resultados, comprobamos que se aprecian ínfimas diferencias entre ambos modelos. Así parece que la aplicación del algoritmo PLA_Pocket no ha tenido gran influencia en el problema de obtener un mejor modelo que clasifique los datos.

- **Calcular E_{in} y E_{test} (error sobre los datos de test).**

  El error que hemos considerado es el error de clasificación, es decir el porcentaje de fallos de clasificación de nuestro modelo.

  Así obtenemos:

  - Regresión lineal: $E_{in} = 22.780569514237854$ y $E_{test} = 25.13661202185793$
  - PLAPocket: $E_{in} = 22.529313232830816$ y $E_{test} = 25.40983606557377$

  Vemos que aunque se mejora E_{in} apenas se aprecia dicha mejora. Mientras que E_{test} empeora pero también de forma ínfima. Los dos modelos son bastante similares.

- **Obtener cotas sobre el verdadero valor de E_{out}. Pueden calcularse dos cotas una basada en E_{in} y otra basada en E_{test}. Usar una tolerancia $\delta = 0,05$. ¿Que cota es mejor?.**

  Usando la cota H estudiada en teoría, para la que necesitamos la tasa de error (valor real del error entre 0 y 1) y el numero de elementos de la muestra, obtenemos que los resultados son:

  E_{out} a partir de E_{in}: 0.4646154745114326
  E_{out} a partir de E_{test}: 0.686358177602074.

  Obtenemos algo coherente: la cota de E_{out} a partir de E_{in} es mejor que la obtenida a partir de E_{test} por la fórmula directa de la cota. Se debe a que el error de E_{in} es menor y la muestra de datos es de tamaño mayor.