UNIVERSIDAD
DE GRANADA

Aprendizaje Automático
grado en ingeniería informática

PRÁCTICA 2

Programación

Autor
Vladislav Nikolov Vasilev

Rama
Computación y Sistemas Inteligentes

Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación

Curso 2018-2019

Índice

1. Ejercicio sobre la complejidad de H y el ruido

En este ejercicio debemos aprender la dificultad que introduce la aparición de ruido en las etiquetas a la hora de elegir la clase de funciones más adecuada. Haremos uso de tres funciones ya programadas:

- $simula\_unif(N, dim, rango)$, que calcula una lista de N vectores de dimensión dim. Cada vector contiene dim números aleatorios uniformes en el intervalo $rango$.
- $simula\_gaus(N, dim, sigma)$, que calcula una lista de longitud N de vectores de dimensión dim, donde cada posición del vector contiene un número aleatorio extraido de una distribucción Gaussiana de media 0 y varianza dada, para cada dimension, por la posición del vector $sigma$.
- $simula\_recta(intervalo)$, que simula de forma aleatoria los parámetros, $v = (a, b)$ de una recta, $y = ax + b$, que corta al cuadrado $[-50, 50] \times [-50, 50]$.

Apartado 1

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

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

Como el código de la función $simula\_unif$ se ha proporcionado, no se va a mostrar su funcionamiento porque se supone que ya es conocido. Habiendo dicho esto, se procede a mostrar los datos generados:

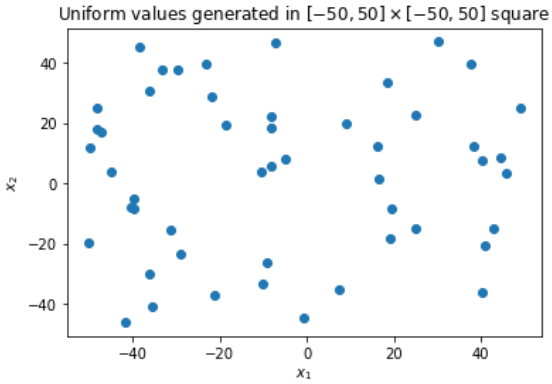

Figura 1: Datos generados por la función $simula\_unif(N, dim, rango)$ con $N = 50$, $dim = 2$, $rango = [-50, +50]$

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

De nuevo, como en el caso anterior, como ya se ha proporcionado la función $simula\_gaus$, no se va a mostrar su funcionamiento porque se supone que ya es conocido. Así que, con esto en mente, podemos mostrar los siguientes puntos generados:

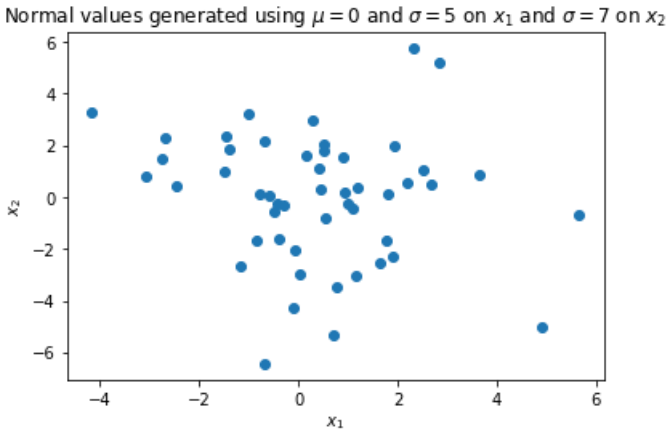

Figura 2: Datos generados por la función $simula\_gaus(N, dim, sigma)$ con $N = 50$, $dim = 2$ y $sigma = [5, 7]$

Apartado 2

Con ayuda de la función $simula\_unif()$ 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 $simula\_recta()$.

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 de la recta).

Para realizar esto, se nos ha proporcionado la función $simula\_recta()$ desde el principio, con lo cuál no la vamos a comentar, ya que se supone conocida. Para clasificar los puntos, vamos a utilizar una función llamada $f(x, y, a, b)$, donde x es la coordenada X del punto, y la coordenada Y, a la pendiente de la recta simulada anteriormente y b el termino independiente. Esta función también se ha proporcionado, con lo cuál no se mostrará su implementación porque, de nuevo, se supone conocida.

Los puntos generados, junto con la recta, son los siguientes:

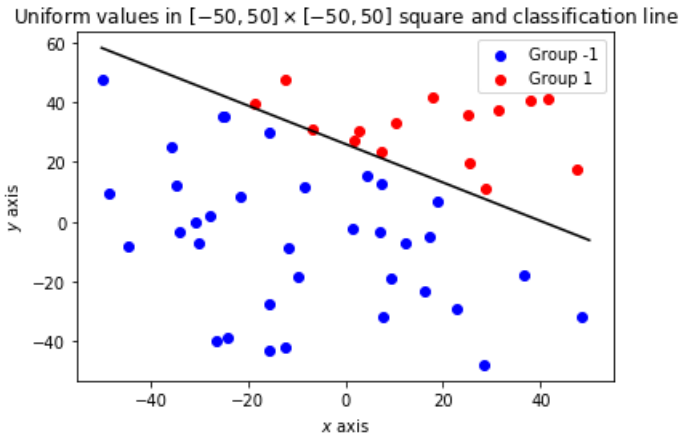

Figura 3: Gráfica de puntos generados uniformemente con la recta que separa las dos clases.

Adicionalmente, para ver que la clasificación es la correcta, se ha calculado el ratio de puntos clasificados correcta e incorrectamente mediante una función, la cuál se puede ver a continuación:

In [5]:
```python
def error_rate(x, y, a, b):
    """
    Funcion para calcular los ratios de acierto y error entre los valores
    reales de las etiquetas y los predichos.

    :param x: Array de vectores de caracteristicas
    :param y: Array de etiquetas
    :param a: Pendiente de la recta
    :param b: Valor independiente de la recta

    :return Devuelve el ratio de aciertos y el ratio de errores
    """

    # Crear lista de y predichas
    predicted_y = []

    # Predecir cada valor
    for value in x:
        predicted_y.append(f(value[0], value[1], a, b))

    # Convertir a array
    predicted_y = np.array(predicted_y)

    return np.mean(y == predicted_y), np.mean(y != predicted_y)
```

Figura 4: Función para el cálculo del ratio de aciertos y errores.

Los resultados obtenidos, como se espera en este caso ya que no hay ruido en la muestra, son los siguientes:

In [7]:
```
# Obtener ratios de acierto y error
accuracy, error = error_rate(x, y, a, b)

print('Ratio de aciertos: {}'.format(accuracy))
print('Ratio de error: {}'.format(error))
```

```
Ratio de aciertos: 1.0
Ratio de error: 0.0
```

Figura 5: Ratios de acierto y error obtenidos de la muestra sin ruido.

b) 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 insertar ruido en la muestra, de forma proporcional a la cantidad de etiquetas de las distintas clases que tenemos, nos hemos ayudado de la siguiente función:

In [11]:
```
def insert_noise(y, ratio=0.1):
    """
    Función para insertar ruido en una muestra de forma proporcional en cada
    clase.

    :param y: Etiquetas sobre las que insertar ruido
    :param ratio: Ratio de elementos de cada clase que modificar
    """
    # Obtener número de elementos sobre los que aplicar ruido
    # (redondear)
    noisy_pos = round(np.where(y == 1)[0].shape[0] * ratio)
    noisy_neg = round(np.where(y == -1)[0].shape[0] * ratio)

    # Obtener las posiciones de forma aleatoria
    pos_index = np.random.choice(np.where(y == 1)[0], noisy_pos, replace=False)
    neg_index = np.random.choice(np.where(y == -1)[0], noisy_neg, replace=False)

    # Cambiar los valores
    y[pos_index] = -1
    y[neg_index] = 1
```

Figura 6: Función para insertar ruido en una muestra de datos.

Al aplicar esta función sobre nuestras etiquetas, obtenemos los siguientes datos:

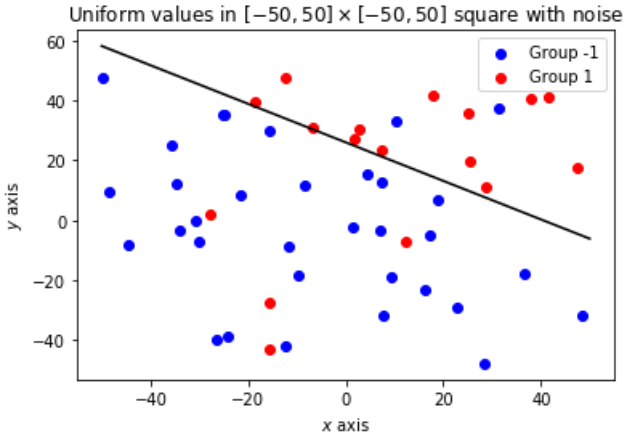

Figura 7: Gráfica de puntos generados uniformemente con ruido junto con la recta que los separa.

Como se puede ver, como había un mayor número de datos con etiqueta -1, se han modificado más elementos de esta clase, siendo este número de elementos modificados proporcional respecto al número de elementos totales de la clase.

Como era de esperarse, al haber modificado las etiquetas, los ratios de aciertos y error se han visto modificados, dejándolos de la siguiente forma:

```
Ratio de aciertos: 0.88
Ratio de error: 0.12
```

Figura 8: Ratios de acierto y error obtenidos de la muestra con ruido.

Apartado 3

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

1. $f(x, y) = (x - 10)^2 + (y - 20)^2 - 400$
2. $f(x, y) = 0.5(x + 10)^2 + (y - 20)^2 - 400$
3. $f(x, y) = 0.5(x - 10)^2 + (y + 20)^2 - 400$
4. $y - 20x^2 - 5x + 3$

Visualizar el etiquetado generado en $2b$ junto con cada una de las gráficas de cada una de las funciones. Comparar las formas de 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? ¿En qué ganan a la función lineal? Explicar el razonamiento.

Para ayudarnos con la visualización de los datos, hemos utilizado una función ya proporcionada, con lo cuál no vamos a discutir como funciona. En vez de eso, veamos cómo son las funciones anteriores una vez que se han implementado:

In [15]:
```python
# Función del primer apartado
def f1(X):
    return (X[:, 0] - 10) ** 2 + (X[:, 1] - 20) ** 2 - 400

# Función del segundo apartado
def f2(X):
    return 0.5 * (X[:, 0] + 10) ** 2 + (X[:, 1] - 20) ** 2 - 400

# Función del tercer apartado
def f3(X):
    return 0.5 * (X[:, 0] - 10) ** 2 - (X[:, 1] + 20) ** 2 - 400

# Función del cuarto apartado
def f4(X):
    return X[:, 1] - 20 * X[:, 0] ** 2  - 5 * X[:, 0]  + 3
```

Figura 9: Funciones implementadas en Python.

Estas funciones están pensadas para que se modifiquen todos los datos a la vez y se devuelva la función aplicada a éstos.

Para calcular estos ratios, nos hemos ayudado de la siguiente función, la cuál ha sido parametrizada para que pueda recibir cualquiera de las funciones anteriores, predecir las etiquetas y devolver los correspondientes ratios:

In [18]:
```python
def error_rate_func(x, y, func):
    """
    Función para calcular los ratios de acierto y error al predecir un conjunto
    de puntos x mediante una función func, con respecto de los valores reales y

    :param x: Puntos que se usarán para predecir
    :param y: Etiquetas reales de los puntos
    :param func: Función con la que se predecirán los puntos

    :return Devuelve el ratio de puntos predichos correctamente y el ratio de
            puntos predichos erróneamente
    """

    # Predecir las etiquetas
    predicted_y = func(x)

    # Hacer que los valores predichos estén en el rango (-1, 1)
    predicted_y = np.clip(predicted_y, -1, 1)

    return np.mean(predicted_y == y), np.mean(predicted_y != y)
```

Figura 10: Función para calcular las tasas de aciertos y error para las nuevas funciones.

A continuación, se presentan las gráficas de las funciones junto con los valors de los ratios de aciertos y error:

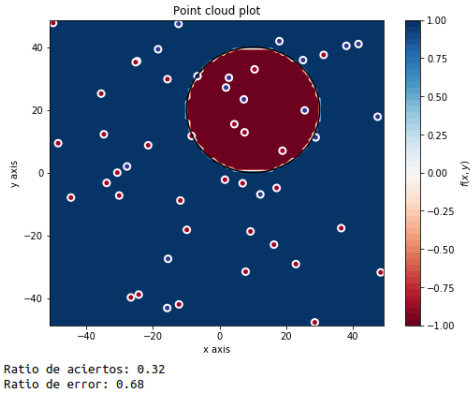

Figura 11: Gráfica y ratios de 1.

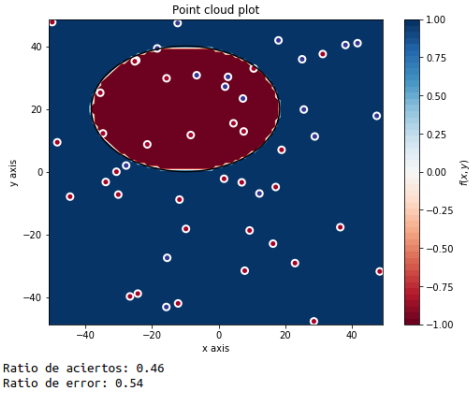

Figura 12: Gráfica y ratios de 2.

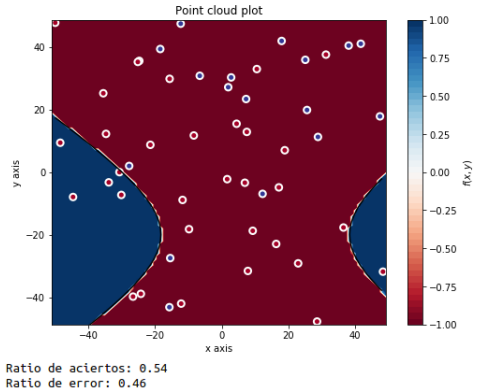

Figura 13: Gráfica y ratios de 3.

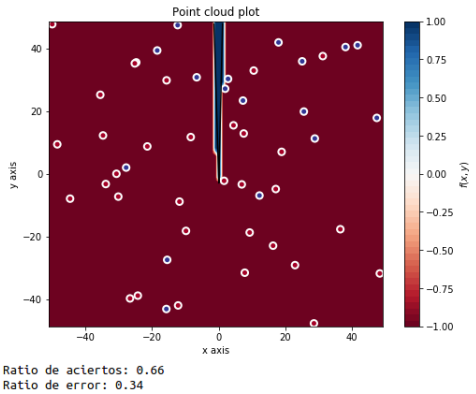

Figura 14: Gráfica y ratios de 4.

Como se puede ver, son funciones más complejas que las lineales. Algunas de ellas son figuras geométricas, como circunferencias o elipses (correspondientes a las funciones 1 y 2). Estas funciones serían capaces de explicar datos muy complejos, datos con los que las funciones lineales tendrían problemas, debido a que no son linealmente separables mediante una recta. Por ejemplo, si una clase estuviese dentro de otra, se podría usar perfectamente una función como 1 o 2.

Sin embargo, en este caso no parecen ser demasiado buenas, ya que en todos los casos los ratios de aciertos son menores que los de la función lineal (recordemos que su ratio de aciertos era 0.88). Esto se debe a que, aunque los datos presenten un cierto ruido, se pueden llegar a separar mejor utilizando una recta en vez de

algo mucho más complejo como una elipse o dividiendo el espacio en regiones como en el caso de 3.

También hay que considerar que en este caso estamos cogiendo directamente funciones, sin entrenar ningún tipo de modelo para intentar ajustarlas a los datos. A lo mejor haciendo esto se podrían obtener algunos resultados un poco mejores en algunas de las funciones, pero como existe ruido en la muestra, es muy difícil dar con la función adecuada y que tenga el mínimo error. Además, en caso de encontrarla, por haber recurrido a una clase de funciones muy compleja por ejemplo, nos podríamos encontrar con el caso de *overfitting*, lo cuál es un problema gravísimo y nada deseable.

En resumen, las funciones no lineales tienen sus puntos fuertes debido a que pueden explicar ciertos tipos de muestras mejor de lo que lo haría una función lineal, pero no siempre son la solución a nuestros problemas. Pueden darse casos, como por ejemplo este, que la función lineal le gane a todas las no lineales, debido a que sea más fácil dividir los datos mediante una recta. Hay que ser muy consciente de qué tipo de función elegir en cada momento. Por ejemplo, un comportamiento no lineal en algunas carcterísticas nos podría indicar que sería mejor intentar recurrir a alguna función no lineal que a una lineal para resolver el problema.