Práctica 2 (Regresión Logística)

¿Cómo he planteado la práctica?

Como es lógico, primero necesitamos los datos, en este caso leemos el CSV 'ex2data1', y separamos los datos en X e Y, siendo Y la última columna, y X las anteriores.

Además, por comodidad, he añadido una columna de 1's al principio del eje X para que, al multiplicar las Thetas, se pueda hacer de forma vectorizada.

Posteriormente he dibujado los ejemplos positivos (o) y negativos(x), esto nos servirá para posteriormente verificar que la frontera de decisión tiene sentido.

También he implementado las funciones de coste y gradiente, que tienen como parámetros:

- Theta
- X
- Y

En el siguiente paso, como pide el enunciado, he calculado el valor óptimo de las θ's a través de la función FMIN_TNC de Scipy.

Finalmente he dibujado la frontera de decisión y evaluado el porcentaje de fiabilidad de la Regresión Logística.

En todos los pasos, he verificado el correcto cálculo de las funciones.

Código comentado y gráficas

1. Importar las librerías, PolynomialFeatures se usa en la parte de regularización de la práctica.

```
from matplotlib import pyplot as plt
import numpy as np
from pandas.io.parsers import read_csv
import scipy.optimize as opt
from sklearn.preprocessing import PolynomialFeatures

```

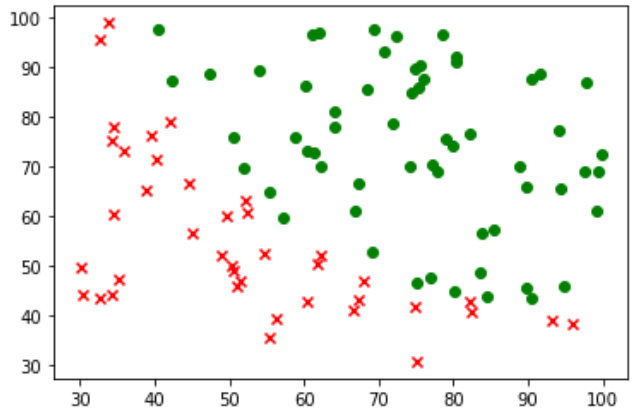
2. Obtener los datos del CSV:

```
data = read_csv("ex2data1.csv", header=None).to_numpy()
X = np.array(data[:, :-1])
Y = np.array(data[:, -1])

m = X.shape[0]
X = np.hstack([np.ones([m, 1]), X])

n = X.shape[1]
theta = np.zeros(n)
```

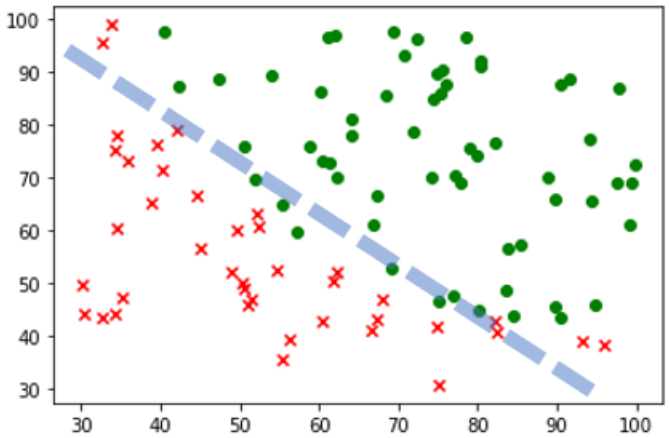
Como he comentado anteriormente, he añadido con hstack una columna de unos por comodidad y, además, he inicializado las θ's.

3. Como siempre, es recomendable que, antes de usar los datos, los visualicemos, como es lógico es una práctica y los datos se sobreentiende que son correctos, pero en un entorno real, los datos pueden ser incorrectos e incluso te pueden dar ideas de cómo tratarlos, por ejemplo, normalizándolos.
   Para clasificarlos, debemos usar la función 'where' (#1 y #2), que devuelve un array con las posiciones de los elementos que cumplen la condición.

```
1  pos = np.where(Y == 1)
2  neg = np.where(Y == 0)
3
4
5  plt.figure()
6  plt.scatter(X[pos, 1], X[pos, 2], marker='o', color='g')
7  plt.scatter(X[neg, 1], X[neg, 2], marker='x', color='r')
8  plt.show()
```



Tras dibujarlo, podemos deducir que queremos la frontera de decisión debe ser algo similar a:



4. Ahora creamos las funciones:
   - Sigmoide

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
```

   - De coste: en la que divido los dos términos (positivo y negativo) y posteriormente los sumo. Tras ello, multiplico por $\frac{-1}{m}$

```
def coste(theta, X, Y):
    m = X.shape[0]
    n = X.shape[1]

    h_theta = np.dot(X, theta)
    sig = sigmoid(h_theta)
    positive = np.dot(np.log(sig).T, Y)
    negative = np.dot(np.log(1 - sig).T, 1 - Y)
    J_theta = (-1 / m) * (positive + negative)
    return J_theta
```

Finalmente, compruebo los resultados:

```
# PRUEBAS
print(coste(theta, X, Y))
```

```
0.6931471805599453
```

   - De gradiente

```
def gradiente(theta, X, Y):
    m = X.shape[0]
    n = X.shape[1]

    h_theta = np.dot(X, theta.T)
    sig = sigmoid(h_theta)
    gradient = (1/m) * np.dot(sig.T - Y, X)
    return gradient
```

Y compruebo los resultados:

```
# PRUEBAS
print(gradiente(theta, X, Y))
```

```
[ -0.1        -12.00921659 -11.26284221]
```

5. En este paso obtenemos las θ's óptimas gracias a la función FMIN_TNC de Scipy

```
theta_optima, _, _ = opt.fmin_tnc(
    func=coste,
    x0 = theta,
    fprime=gradiente,
    args=(X, Y)
)
```

NOTA: Esta función solo puede utilizarse cuando, a la función de coste y a la función de gradiente, le pasamos primero el parámetro θ y después 'X' e 'Y'

Tras ello, comprobamos que el coste óptimo es el que se pide en el enunciado:

```
# Pruebas
coste_optimo = coste(theta_optima, X, Y)
print(coste_optimo)
```

```
0.20349770158947508
```

6. Para dibujar la gráfica y la frontera de decisión, primero debemos crear una función que nos permita dibujar esta última.
   Esta función obtiene el mínimo y máximo de las características 'X', después creamos una matriz que vaya desde el mínimo de cada característica hasta su máximo.
   - Recordemos que:
     - Por defecto, np.linspace genera 50 números distribuidos de forma lineal, desde el inicio hasta el final.
     - np.ravel vectoriza una matriz (#17 y #18)
     - np.ones, crea un vector o matriz con las dimensiones pedidas, en este caso 50x1 (#16)
     - np.c_[matriz1, matriz2…], relaciona las posiciones de los vectores

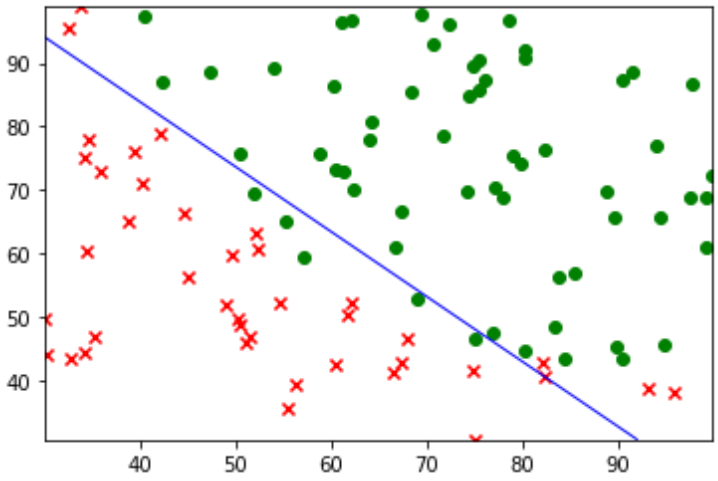
   El resultado de ello se multiplica por θ (#19) y de este resultado se calcula la función sigmoide (#14)

   Finalmente se hace un reshape para que los valores cuadren (#22) y a la función contour le pasamos el eje X, el eje Y, la lista de valores, los niveles, el grosor y el color.

```
1  def pinta_frontera(X, Y, theta):
2
3      x1_min, x1_max = np.min(X[:, 1]), np.max(X[:, 1])
4      x2_min, x2_max = np.min(X[:, 2]), np.max(X[:, 2])
5
6      # print(x1_min, x1_max)
7      # print(x2_min, x2_max)
8
9      xx1, xx2 = np.meshgrid(
10         np.linspace(x1_min, x1_max),
11         np.linspace(x2_min, x2_max)
12     )
13
14     h = sigmoid(
15         np.c_[
16             np.ones((xx1.ravel().shape[0], 1)),
17             xx1.ravel(),
18             xx2.ravel()
19         ].dot(theta)
20     )
21
22     h = h.reshape(xx1.shape)
23
24
25     plt.contour(xx1, xx2, h, [.5], linewidths=1, colors='b')
```

Tras todo ello, volvemos a hacer lo mismo del paso 3, pero ahora también llamamos a la función 'pinta_frontera':

```
pos = np.where(Y == 1)
neg = np.where(Y == 0)

plt.scatter(X[pos, 1], X[pos, 2], marker='o', color='g')
plt.scatter(X[neg, 1], X[neg, 2], marker='x', color='r')

pinta_frontera(X, Y, theta_optima)

plt.show()

```



7. Finalmente, para evaluar la precisión de nuestra regresión logística, debemos calcular los aciertos respecto al Dataset que tenemos, para ello predecimos los resultados con las θ's óptimas, si es mayor o igual a 0'5 es un aprobado (1), en caso contrario es un suspenso (0).
   Ahora comparamos nuestras predicciones con la 'Y' y sumamos todos los aciertos.
   Finalmente aplicamos la fórmula de $\frac{aciertos}{total} * 100$ que nos devolverá un porcentaje de aciertos.

```
def evaluar(X, Y, theta):
    predicciones = np.dot(X, theta_optima) >= 0.5
    aciertos = np.sum(predicciones == Y)
    porcentaje = aciertos / X.shape[0] * 100

    print("La regresión logística es fiable en un",
          porcentaje,
          "% de ocasiones")

evaluar(X, Y, theta_optima)

```

```
La regresión logística es fiable en un 89.0 % de ocasiones
```

Práctica 2 (Regularización)

¿Cómo he planteado la práctica?

Como es lógico, primero necesitamos los datos, en este caso leemos el CSV 'ex2data2', y separamos los datos en X e Y, siendo Y la última columna, y X las anteriores.

Posteriormente he dibujado los ejemplos positivos (o) y negativos(x), esto nos servirá para posteriormente verificar que la frontera de decisión tiene sentido.

También he implementado las funciones de coste y gradiente regularizados, que tienen como parámetros:

- Theta
- X
- Y
- Lambda

En el siguiente paso, como pide el enunciado, he calculado el valor óptimo de las θ's a través de la función FMIN_TNC de Scipy.

Finalmente he dibujado la frontera de decisión y experimentado con diferentes valores para λ y, por ende, diferentes θ's. Además, he evaluado la fiabilidad que nos ofrecen dichos valores.

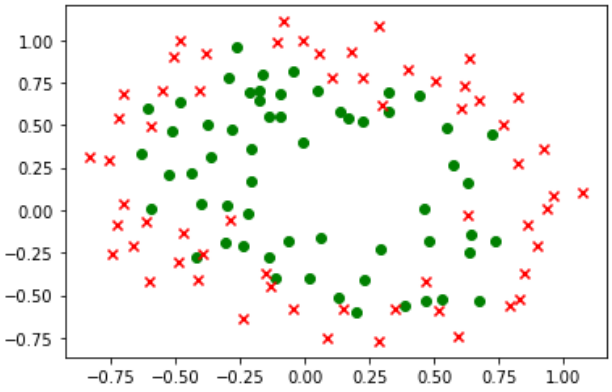
En todos los pasos, he verificado el correcto cálculo de las funciones.

1. Al igual que en la anterior parte, primero debemos leer los datos, siendo 'Y' la última columna, y 'X' las anteriores.

```
data = read_csv("ex2data2.csv", header=None).to_numpy()
X = np.array(data[:, :-1])
Y = np.array(data[:, -1])
```

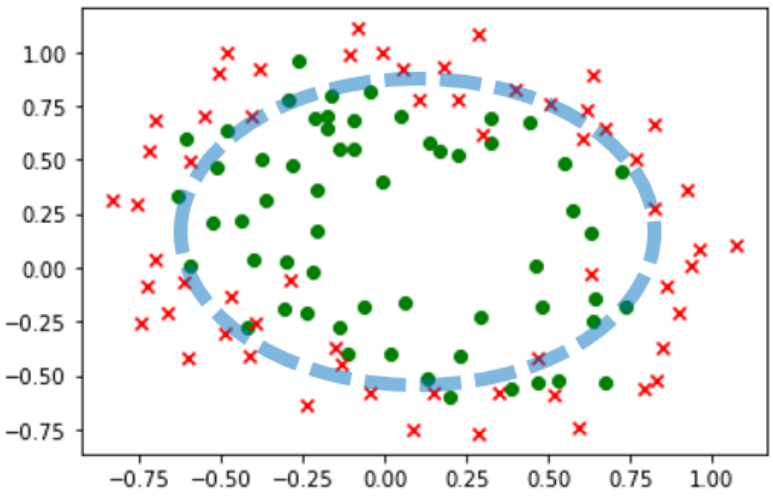
2. También mostramos los datos

```
pos = np.where(Y == 1)
neg = np.where(Y == 0)

plt.figure()
plt.scatter(X[pos, 0], X[pos, 1], marker='o', color='g')
plt.scatter(X[neg, 0], X[neg, 1], marker='x', color='r')
plt.show()
```



NOTA: En este caso, a diferencia de la primera parte, no he añadido la columna de 1's, dado que al usar la función 'PolynomialFeatures' te añade por defecto dicha columna, por lo que las posiciones son 0 y 1.

Tras dibujarlo, podemos deducir que queremos la frontera de decisión debe ser algo similar a:



3. Ahora debemos crear la X polinómica de grado 6, para ello usaremos la función 'PolynomialFeatures' y además ajustaremos la X a dichos datos.

```
poly = PolynomialFeatures(degree=6)
X_poly = poly.fit_transform(X)
```

4. Inicializamos las variables sobre nuestra nueva 'X' y, por ende, las θ's deben crearse con el tamaño nuevo.

```
m = X_poly.shape[0]
n = X_poly.shape[1]

theta = np.zeros(n)
```

5. La función de coste regularizada es igual que la función de coste normal, a excepción de que se añade un nuevo término, por lo cual se puede reutilizar la función de la primera parte.

```
def coste_regularizado(theta, X, Y, lam):
    m = X_poly.shape[0]
    reg = (lam /(2 * m)) * np.sum(np.square(theta))
    return coste(theta, X, Y) + reg
```

```
# PRUEBAS
print(coste_regularizado(theta, X_poly, Y, 1))
```

```
0.6931471805599454
```

6. La función de gradiente regularizada es igual que la función de gradiente normal, a excepción de que se añade un nuevo término, por lo cual se puede reutilizar la función de la primera parte.

```
def gradiente_regularizada(theta, X, Y, lam):
    m = X_poly.shape[0]
    reg = (lam / m) * theta
    return gradiente(theta, X, Y) + reg
```

```
# PRUEBAS
print(gradiente_regularizada(theta, X_poly, Y, 1))
```

```
[8.47457627e-03 1.87880932e-02 7.77711864e-05 5.03446395e-02
 1.15013308e-02 3.76648474e-02 1.83559872e-02 7.32393391e-03
 8.19244468e-03 2.34764889e-02 3.93486234e-02 2.23923907e-03
 1.28600503e-02 3.09593720e-03 3.93028171e-02 1.99707467e-02
 4.32983232e-03 3.38643902e-03 5.83822078e-03 4.47629067e-03
 3.10079849e-02 3.10312442e-02 1.09740238e-03 6.31570797e-03
 4.08503006e-04 7.26504316e-03 1.37646175e-03 3.87936363e-02]
```

7. En este paso obtenemos las θ's óptimas gracias a la función FMIN_TNC de Scipy.

```
theta_optima, _, _ = opt.fmin_tnc(
    func=coste_regularizado,
    x0 = theta,
    fprime=gradiente_regularizada,
    args=(X_poly, Y, 1)
)
```

```
# PRUEBAS
coste_optimo = coste_regularizado(theta_optima, X_poly, Y, 1)
print(coste_optimo)
```

```
0.5351602547833242
```

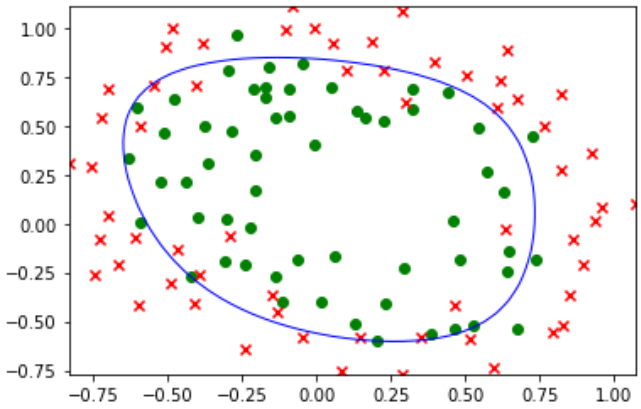
NOTA: Esta función solo puede utilizarse cuando, a la función de coste y a la función de gradiente, le pasamos primero el parámetro θ y después 'X','Y' y λ = 1

8. Ahora creamos una función que nos permita pintar la frontera pero que acepte parámetros polinómicos. Para ello hacemos lo mismo que en la primera parte, la única diferencia es que ahora transformamos los datos a polinómicos (#12)

```
def pinta_frontera_polinomica(X, Y, theta, poly):

    x1_min, x1_max = np.min(X[:, 0]), np.max(X[:, 0])
    x2_min, x2_max = np.min(X[:, 1]), np.max(X[:, 1])

    xx1, xx2 = np.meshgrid(
        np.linspace(x1_min, x1_max),
        np.linspace(x2_min, x2_max)
    )

    h = sigmoid(
        poly.fit_transform(
            np.c_[
                xx1.ravel(),
                  xx2.ravel()
            ]
        ).dot(theta)
    )

    h = h.reshape(xx1.shape)
    plt.contour(xx1, xx2, h, [0.5], linewidths=1, colors='b')
```

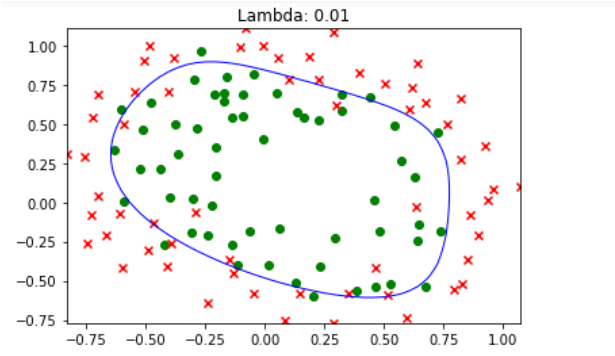
9. Y como anteriormente, dibujamos todos los datos y llamamos a la función del paso 8.

```
pos = np.where(Y == 1)
neg = np.where(Y == 0)

plt.figure()
plt.scatter(X[pos, 0], X[pos, 1], marker='o', color='g')
plt.scatter(X[neg, 0], X[neg, 1], marker='x', color='r')
pinta_frontera_polinomica(X, Y, theta_optima, poly)
plt.show()
```

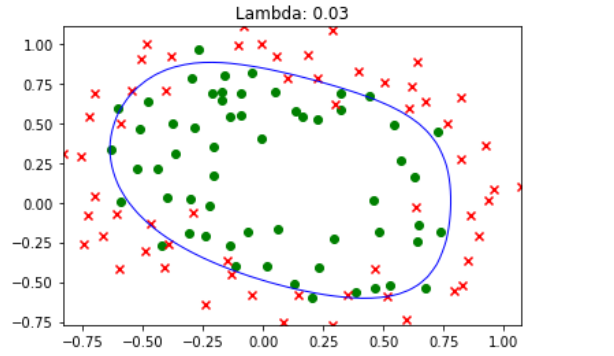


10. Ahora vamos a experimentar con las diferentes λ's y, por consiguiente, θ's.
Para ello vamos a crear un array de lambdas y vamos a recorrerlo con un bucle, llamando a FMIN_TNC para que nos devuelva las θs óptimas.
Posteriormente dibujamos los datos y la frontera.
Finalmente evaluamos la precisión de la regresión.

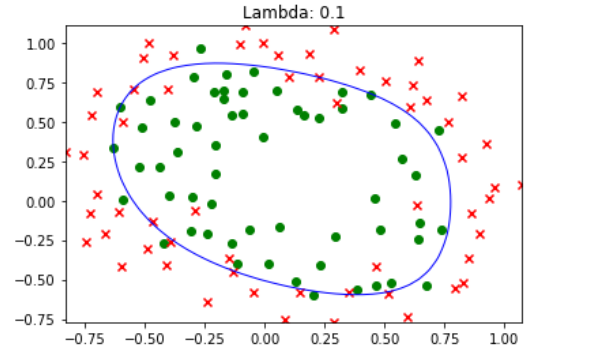
```
lambdas = [.01, .03, .1, .3, 1, 3]

for l in lambdas:
    theta_optima, _, _ = opt.fmin_tnc(
        func=coste_regularizado,
        x0 = theta,
        fprime=gradiente_regularizada,
        args=(X_poly, Y, l)
    )

    pos = np.where(Y == 1)
    neg = np.where(Y == 0)

    plt.figure()
    plt.title('Lambda: {}'.format(l))
    plt.scatter(X[pos, 0], X[pos, 1], marker='o', color='g')
    plt.scatter(X[neg, 0], X[neg, 1], marker='x', color='r')
    pinta_frontera_polinomica(X, Y, theta_optima, poly)

    plt.show()

    evaluar(X_poly, Y, theta_optima)
    print("\n")
```



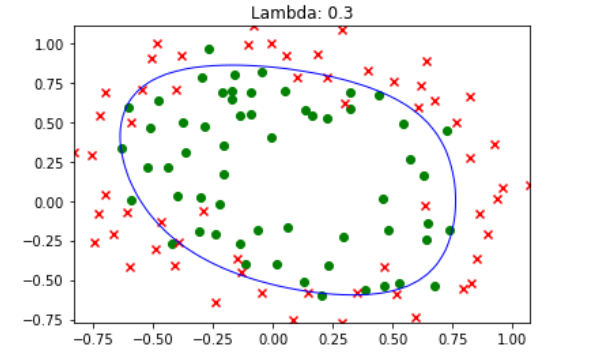
La regresión logística es fiable en un 83.90% de ocasiones



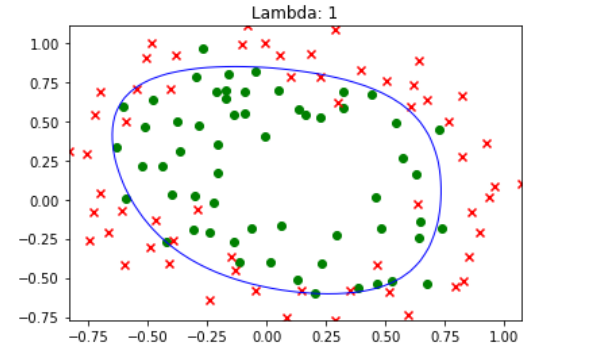
La regresión logística es fiable en un 84.75% de ocasiones



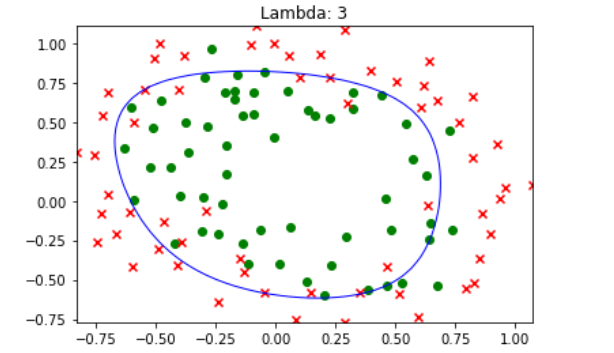
La regresión logística es fiable en un 82.20% de ocasiones



La regresión logística es fiable en un 81.36% de ocasiones



La regresión logística es fiable en un 76.27% de ocasiones



La regresión logística es fiable en un 59.32% de ocasiones

Podemos observar que, cuando la regularización es muy baja, las predicciones tienden a sobreajustarse a los datos de entrenamiento, es decir, que no se generaliza correctamente, esto lo sabemos porque la forma es un tanto irregular.

Por otro lado, cuando la regularización es muy alta, podemos ver que tiende a subajustarse a los datos de entrenamiento, es decir, se generaliza demasiado.