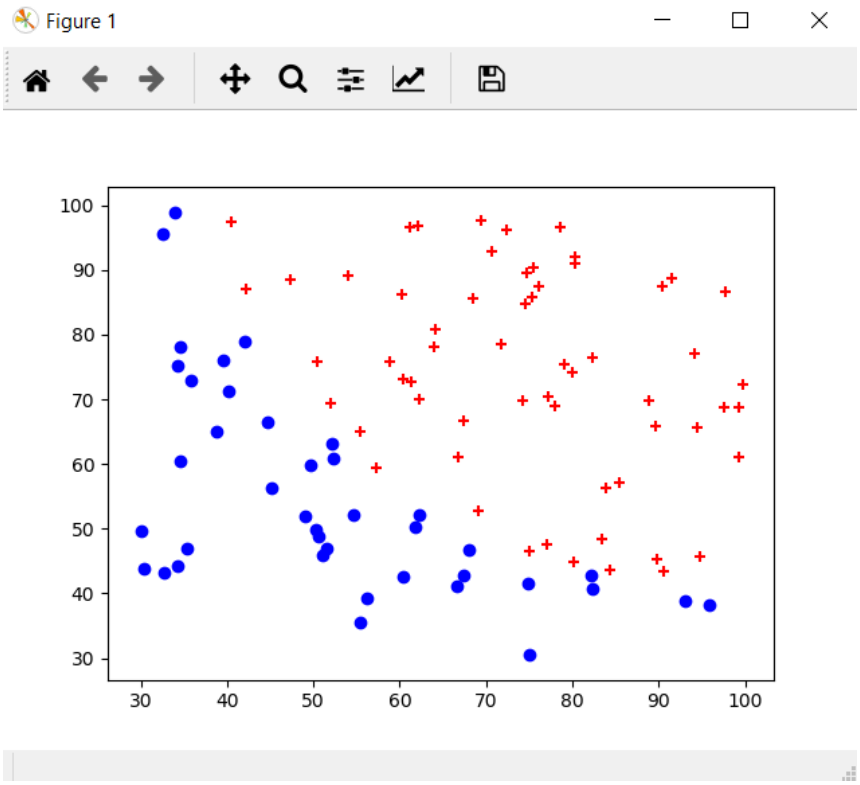
REGRESIÓN LOGÍSTICA

En la primera parte de la práctica se nos pide aplicar el método de regresión logística para una distribución de datos dada. La finalidad es encontrar una recta que clasifique los datos de la forma más correcta posible.

El primer paso consiste en visualizar los datos proporcionados por el archivo .csv. Para ello hemos implementado una función **data_visualization(X, Y)** que recibe como parámetros las matrices X e Y y devuelve una gráfica con la información representada.

```
def data_visualization(X, Y):

    pos_0 = np.where(Y == 0)
    pos_1 = np.where(Y == 1)

    plt.scatter(X[pos_0, 0], X[pos_0, 1], c = "green")
    plt.scatter(X[pos_1, 0], X[pos_1, 1], marker='+', c = 'k')

    plt.show()
```

Los datos representados gráficamente son los siguientes:



A continuación, implementaremos una función **sigmoid_function(X)** para calcular la función sigmoide sobre un parámetro, que puede ser un número, un array o una matriz. En caso de ser un array o una matriz, la función se aplicará para cada valor de este.

```
def sigmoid_function(X):
    return 1 / (1 + np.exp(-X))
```

Más adelante, implementaremos dos funciones **cost(Thetas, X, Y)** y **gradient(Thetas, X, Y)** para calcular la función de coste y su gradiente aplicando las correspondientes fórmulas. Ambas funciones reciben como parámetros las matrices X e Y y el vector de Theta.

```
def cost(Thetas, X, Y):

    m = X.shape[0]
    #J(θ) = -(1/m) * (A + B * C)
    #J(θ) = -(1/m) * ((log (g(Xθ)))T * y + (log (1 - g(Xθ)))T * (1 - y))

    #A
    X_Teta = np.dot(X, Thetas)
    g_X_Thetas = sigmoide_function(X_Teta)
    log_g_X_Thetas = np.log(g_X_Thetas)
    T_log_g_X_Thetas = np.transpose(log_g_X_Thetas)
    y_T_log_g_X_Thetas = np.dot(T_log_g_X_Thetas, Y)

    A = y_T_log_g_X_Thetas

    #B
    one_g_X_Thetas = 1 - g_X_Thetas
    log_one_g_X_Thetas = np.log(one_g_X_Thetas)
    T_log_one_g_X_Thetas = np.transpose(log_one_g_X_Thetas)

    B = T_log_one_g_X_Thetas

    #C
    C = 1 - Y

    J = (-1/m) * (A + (np.dot(B, C)))


    return J
```

```
def gradient(Thetas, X, Y):

    m = X.shape[0]
    #(δJ(θ)/δθj) =(1/m)*XT*(g(Xθ) - y)
```

```
    X_Teta = np.dot(X, Thetas)
    g_X_Thetas = sigmoide_function(X_Teta)

    X_T = np.transpose(X)

    gradient = (1/m)*(np.dot(X_T, g_X_Thetas - Y ))

    return gradient
```

A continuación, hemos implementado una función **optimized_parameters(Thetas, X, Y)** para optimizar los parámetros y así obtener un vector de thetas óptimo. La función recibe como parámetros las matrices X e Y y el vector de Thetas.

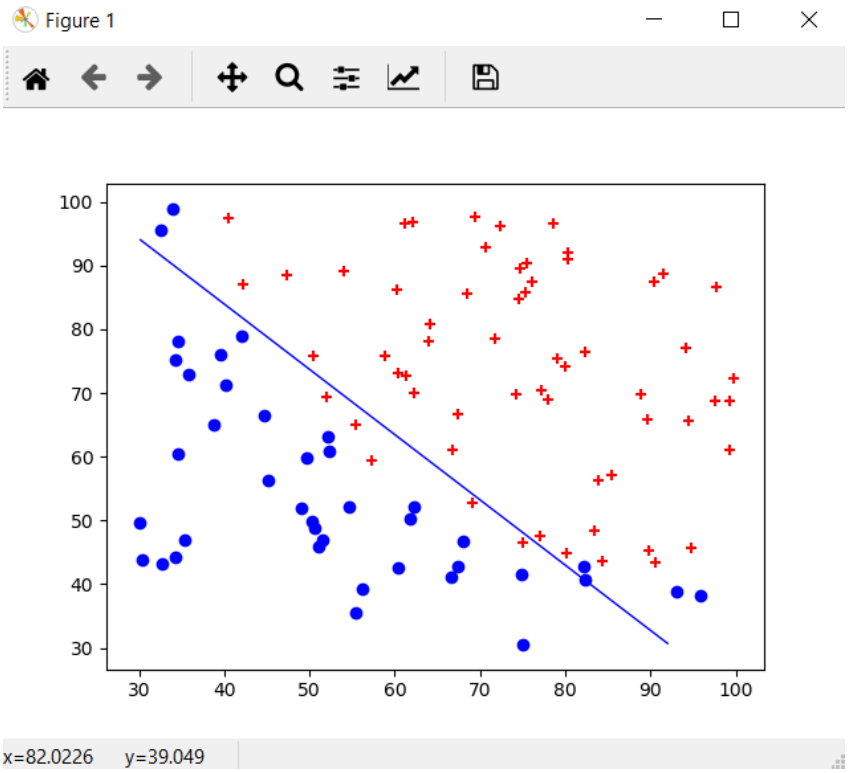
Para realizar la optimización haremos uso de la función opt.fmin_tnc que llamaremos importando el módulo **scipy.optimize**

```
def optimized_parameters(Thetas, X, Y):

    result = opt.fmin_tnc(func = cost, x0 = Thetas, fprime = gradient,
args = (X, Y) )
    theta_opt = result[0]

    return theta_opt
```

También hemos implementado la función **draw_frontier(X, Y, Thetas)** que dibuja la recta de regresión con los nuevos valores optimizados. Esta función recibe como parámetros las matrices X e Y y el vector de Thetas optimizado.

```
def draw_frontier(X, Y, Thetas):

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

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

    h = sigmoide_function(np.c_[np.ones((xx1.ravel().shape[0], 1)),
xx1.ravel(), xx2.ravel()].dot(Thetas))
    h = h.reshape(xx1.shape)

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

    data_visualization(X, Y)
```

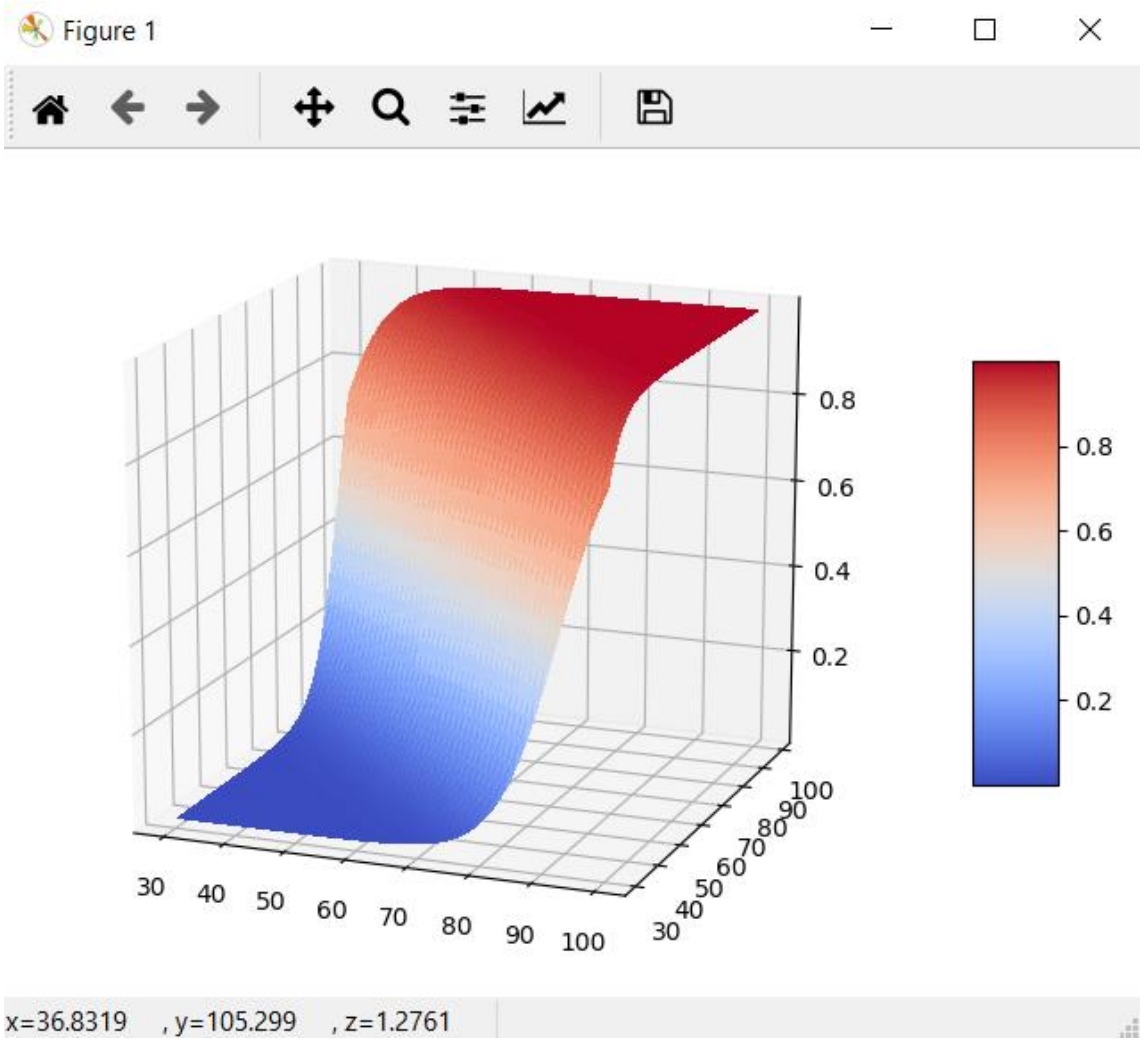
La gráfica obtenida es la siguiente:



Además, hemos implementado un función **draw_frontier_3D(X, Y, Thetas)** que nos muestra los resultados de la regresión en un entorno 3D, para una mejor visualización de los datos obtenidos.

```
def draw_frontier_3D(X, Y, Thetas):



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

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

    h = sigmoide_function(np.c_[np.ones((xx1.ravel().shape[0], 1)),
xx1.ravel(), xx2.ravel()].dot(Thetas))
    h = h.reshape(xx1.shape)

    fig = plt.figure()
    ax = Axes3D(fig)
    surf = ax.plot_surface(xx1, xx2, h, cmap= cm.coolwarm, linewidths= 0,
antialiaseds = False)
    fig.colorbar(surf, shrink = 0.5, aspect = 5)
    plt.show()
```

Y la gráfica 3D que obtenemos es la siguiente



Por último, hemos creado una función **logistic_regresion_evaluation(X, Y, Z)** para evaluar el porcentaje de acierto del método de regresión logística. Para ello hemos comparado los casos clasificados adecuadamente aplicando el método de regresión logística con los casos correctos proporcionados por los propios datos y finalmente hemos obtenido una fiabilidad para el algoritmo del 83%

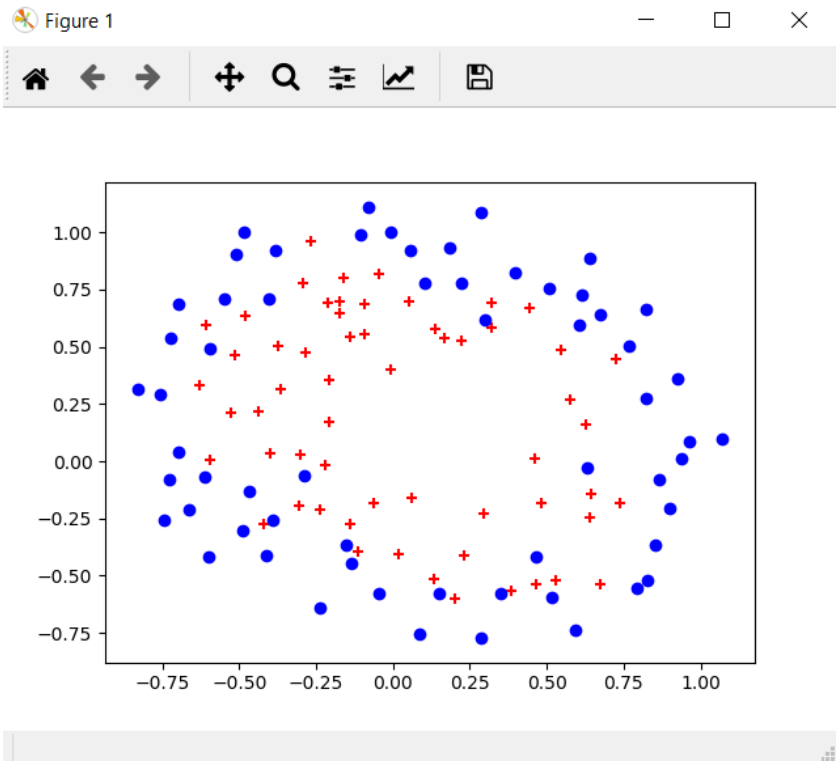
```
def logistic_regresion_evaluation(X, Y, Z):

    H_ = H(X, Z)
    H_sigmoid = sigmoide_function(H_)

    H_sigmoid_evaluated = (H_sigmoid >= 0.5).astype(np.float) #every
value that keeps the condition will return true. astyping it to int it
will turn it into a one value, having an array ready to compare with Y_

    comparison_array = H_sigmoid_evaluated == Y #returns an array where
each value will be true if the condition is keeped
```

```
    coincidences = comparison_array[comparison_array == True] #return an
array with only the elements that keep the condition from the original
array

    percentage_correct_clasification = (coincidences.shape[0] /
comparison_array.shape[0]) * 100.0 #how many trues(coincidences) do we
have in comparison with all the succesful and not succesful coincidences
(trues/(trues+falses))

    return(percentage_correct_clasification)
```

REGRESIÓN LOGÍSTICA REGULARIZADA

El objetivo de esta segunda parte es implementar el algoritmo de regresión logística de forma regularizada. Esta vez, la distribución de los datos no puede ser separada linealmente, por eso necesitamos regularizar la regresión.

Lo primero que haremos será visualizar los datos nuevamente para comprobar que efectivamente, no son linealmente separables.



Para obtener un mejor ajuste a los métodos de entrenamiento, añadiremos nuevos atributos a la descripción de los ejemplos, combinándolos con los originales. Usando la función sklearn.preprocessing.PolynomialFeatures extenderemos cada ejemplo de entrenamiento hasta la sexta potencia , obteniendo así, un total de 28 atributos para cada ejemplo.

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

*X_ es la matriz que contiene los ejemplos de entrenamiento

A continuación, hemos implementado las versiones regularizadas para calcular el coste y el gradiente.

```
def cost_regularized(Thetas, X, Y, h):

    m = X.shape[0]
    Thetas_ = Thetas

    #J(θ) = (cost(Thetas, X, Y)) + D
    #J(θ) = [-(1/m) * ((log (g(Xθ)))T * y + (log (1 - g(Xθ)))T * (1 -
y)))] + (λ/2m)*E(Theta^2)

    cost_ = cost(Thetas, X, Y)

    #D
    Thetas_ = two_power(Thetas_)

    D = h/(2*m) * np.sum(Thetas_)

    J_regularized = (cost_) + D

    return J_regularized
```

```
def gradient_regularized(Thetas, X, Y, h):

    m = X.shape[0]
    #(δJ(θ)/δθj) =(1/m)*XT*(g(Xθ) - y) + (λ/2m)(Theta)
    gradient_ = gradient(Thetas, X, Y)

    g_regularized = gradient_ + (h/m)*Thetas

    return g_regularized
```

El nuevo parámetro ‘h’ hace referencia a lambda, que por defecto será igual a 1.

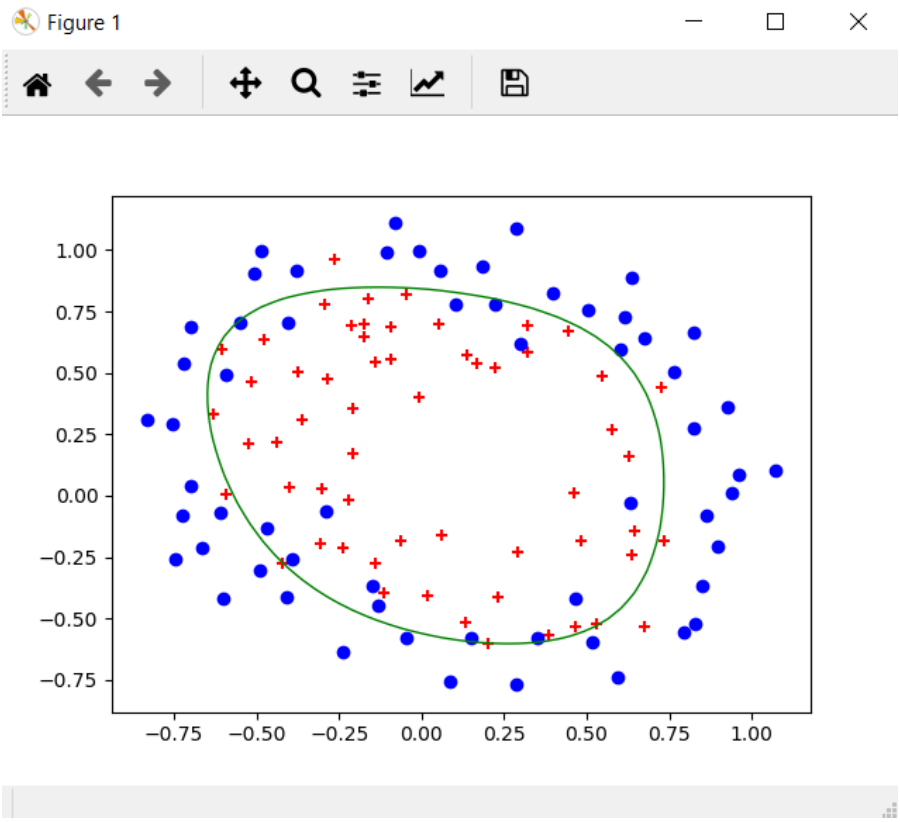
Tal como hicimos en la parte 1, optimizaremos los parámetros para así obtener un versión del vector de Thetas óptima usando la función optimized_parameters_regularized(Thetas, X, Y, h)

```
def optimized_parameters_regularized(Thetas, X, Y, h):

    result = opt.fmin_tnc(func = cost_regularized, x0 = Thetas, fprime =
gradient_regularized, args = (X, Y, h) )
    theta_opt = result[0]

    return theta_opt
```

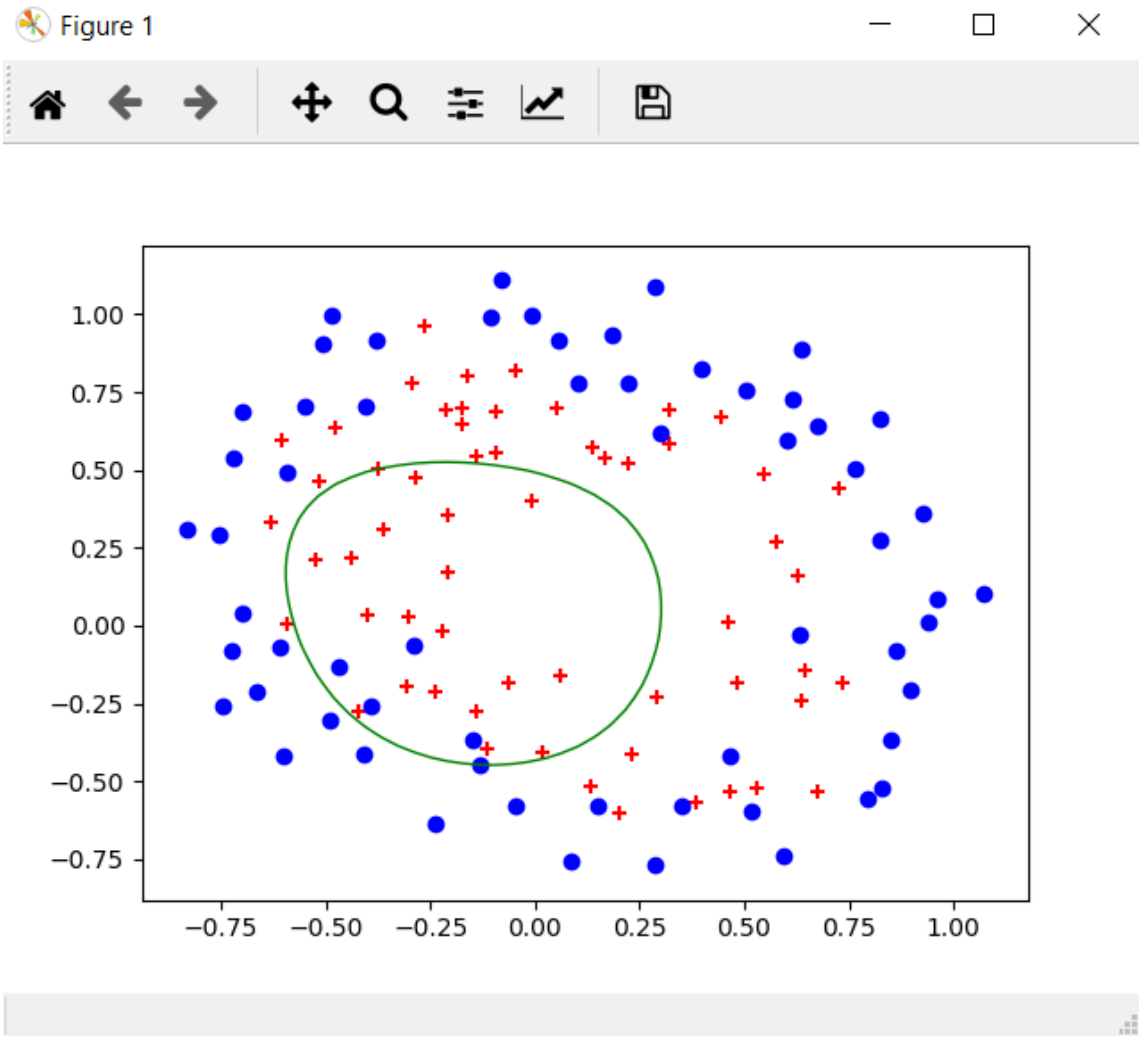
A continuación, creamos una función draw_frontier_regularized(X, Y, Theta, poly) para dibujar de forma gráfica la frontera obtenida que clasifica los datos tras aplicar la regresión regularizada.

```
def draw_frontier_regularized(X, Y, Theta, poly):

    plt.figure()
    x1_min, x1_max = X[:, 0].min(), X[:, 0].max()
    x2_min, x2_max = X[:, 1].min(), X[:, 1].max()
    xx1, xx2 = np.meshgrid(np.linspace(x1_min, x1_max),
np.linspace(x2_min, x2_max))

    h = sigmoide_function(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='g')

    data_visualization(X, Y)
```

La gráfica de datos obtenida es la siguiente:



Podemos comprobar que según vamos aumentando el valor de lambda, el tamaño de la frontera disminuye, perdiendo precisión a la hora de clasificar los ejemplos de aprendizaje.



Ejemplo con lambda = 60.

Tal como hicimos con la anterior parte, en esta también hemos implementado una función draw_frontier_regularized_3D(X, Y, Theta, poly) para mostrar los resultados en 3D.

```
def draw_frontier_regularized_3D(X, Y, Theta, poly):

    x1_min, x1_max = X[:, 0].min(), X[:, 0].max()
    x2_min, x2_max = X[:, 1].min(), X[:, 1].max()
    xx1, xx2 = np.meshgrid(np.linspace(x1_min, x1_max),
np.linspace(x2_min, x2_max))

    h = sigmoide_function(poly.fit_transform(np.c_[xx1.ravel(),
xx2.ravel()]).dot(Theta))
    h = h.reshape(xx1.shape)

    fig = plt.figure()
    ax = Axes3D(fig)
    surf = ax.plot_surface(xx1, xx2, h, cmap= cm.coolwarm, linewidths= 0,
antialiaseds = False)
    fig.colorbar(surf, shrink = 0.5, aspect = 5)
```

```
    plt.show()
```

La gráfica 3D que obtenemos es la siguiente: