Aprendizaje automático y minería de datos

Trabajo final – Clasificador de flores

Ramón Arjona Quiñones
Celia Castaños Bornaechea

Contenido

Dataset

El *dataset* utilizado contiene 4242 imágenes de flores. Están divididas en cinco clases: margarita, tulipán, rosa, girasol y diente de león. Cada clase cuenta con unas 800 fotos de media cada una de distintas proporciones.

Objetivo

Diferenciar de qué tipo es la flor de la fotografía.

Librerías

```
# numpy # (para todo)
import numpy as np
# matplotlib # (para dibujar)
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
# sklearn # (para SVM y para dividir los conjuntos de datos)
from sklearn.svm import SVC
from sklearn.multiclass import OneVsRestClassifier
from sklearn.model_selection import train_test_split
# scipy # (para cargar y guardar matrices)
from scipy.io import savemat, loadmat
# opencv # (para leer imágenes)
import cv2
# sistema operativo # (para los paths)
import os
# tqdm # (para la barra de progreso)
from tqdm import tqdm
from tqdm import trange
# nuestro #
from ML_utilities import trainNeutralNetwork, forward_prop,
calcula_porcentaje, calcula_porcentaje_Y, sigmoid, hMatrix, oneVsAll,
makeOneHot, getBestSVMMultiClass
# joblib # (para guardar y cargar las SVM)
from joblib import dump, load
```

Variables globales

```
# RECURSOS
IMG_SIZE = 64 # Con esto parece que es suficiente (no hay diferencia con
subirlo a 32 o 64, y como tenemos muchos datos, nos vale)
RES_PATH = 'flowers/'
FLOWER_NAMES = ['daisy', 'dandelion', 'rose', 'sunflower', 'tulip']
FLOWER_COUNT = [0, 0, 0, 0, 0] #Se rellena solo
SAMPLE_COUNT = 200 #Número de flores de cada tipo

# DATOS CRUDOS (SIN DIVIDIR)
X = []
```

```
y = []

# PORCENTAJES PARA DIVIDIR EL DATASET (60-20-20)
TRAIN_FRACTION = 0.6
VAL_FRACTION = 0.2
TEST_FRACTION = 0.2

# Para cargar y guardar matrices de datos
DATA_FILENAME = "flowersData.mat"
WEIGHTS_NAMES = ["OneVsAllWeights.mat", "NetworkWeights.mat", "SVM.joblib
"]
```

Carga de los datos

Generamos un archivo .mat con las imágenes y las dividimos en tres grupos (entrenamiento, validación y test).

Para que posteriormente no tarde tanto hemos cogido 200 fotos de cada tipo de flor en vez de las 800 que vienen por defecto. Además, puesto que inicialmente cada imagen tiene unas proporciones, se reescalan al mismo tamaño todas y con proporciones cuadradas.

La creación de la matriz solo tiene que ejecutarse una vez, después ya se puede trabajar con la matriz que se ha generado y guardado.

```
def LoadAllImages():
    '''
    Carga todas las imágenes
    '''
    print("··· Cargando las imágenes de las flores ··· ")
    for i in range (len(FLOWER_NAMES)):
        LoadFlowerImages(i)

    # Devolvemos los sets ya divididos
    return DivideSets(X, y)


def LoadFlowerImages(flowerType):
    '''
    Carga las imágenes del directorio especificado y rellena la Y
    (one_hot) con el índice especificado
    '''
    folder = RES_PATH + FLOWER_NAMES[flowerType]
    indice = 0
    #Calculamos el índice del 1er elemento que vamos a meter
    for i in range(flowerType):
        indice+=FLOWER_COUNT[i]
```

```
    #Recorre las imágenes de ese directorio
    images = os.listdir(folder)[:SAMPLE_COUNT]
    for img in tqdm(images):
        # label = assign_label(img,flower_type)
        path = os.path.join(folder,img) #Path de la imagen
        img = cv2.imread(path, cv2.IMREAD_COLOR) #Lee la imagen
        new_array = cv2.resize(img, (IMG_SIZE, IMG_SIZE)) #La reescala a
                                                          SIZEXSIZE


        #Añade la imagen a los casos de entrenamiento
        X.append(np.array(new_array / 255).ravel())
        y.append(flowerType)
        FLOWER_COUNT[flowerType] += 1 #Añadimos una flor al recuento


def DivideSets(X, y):
    '''
    Divide el dataset en 3 sets de entrenamiento, validación y test
    '''
    #Dividimos en entrenamiento / test (80% - 20%)
    X_train, X_test, y_train, y_test = train_test_split(np.asarray(X),
    np.asarray(y), test_size=TEST_FRACTION, random_state=42)
    #Volvemos a dividir en entrenamiento / validación, para tener un
     total de 60% (train), 20% (val), 20% (test)
    X_train, X_val, y_train, y_val =train_test_split(np.asarray(X_train),
    np.asarray(y_train), test_size=0.25, random_state=42)
    #X_mock, X_val, y_mock, y_val = train_test_split(np.asarray(X_train),
    np.asarray(y_train), test_size=0.25, random_state=42)

    #Devolvemos una tupla con los conjuntos troceados
    return [np.asarray(X_train), np.asarray(y_train), np.asarray(X_val),
           np.asarray(y_val), np.asarray(X_test), np.asarray(y_test)]
```

```
··· Cargando las imágeners de las flores ···
100%|| 200/200 [00:07<00:00, 26.73it/s]
100%|| 200/200 [00:05<00:00, 37.70it/s]
100%|| 200/200 [00:05<00:00, 35.58it/s]
100%|| 200/200 [00:05<00:00, 39.73it/s]
100%|| 200/200 [00:06<00:00, 32.93it/s]
··· Datasets guardados en flowersData.mat ···
```

Regresión logística

Funcionamiento

Es multiclase ya que tiene varias salidas distintas. El método recibe una tupla de valores para λ y repite los cálculos con todos ellos para encontrar el óptimo, si la longitud de la tupla es mayor que uno.

Recorre con un bucle la tupla y realiza primero el entrenamiento con el oneVsAll. Cuando termina pasa a ver la precisión que tiene con los ejemplos de validación. Y por último comprueba si con el valor de λ actual el resultado es mejor que con el anterior.

Finalmente guardamos los valores de los pesos con el valor de λ óptimo en un archivo *.mat* para utilizarlos después al calcular el porcentaje de aciertos, que se encuentra en un método aparte.

Ejemplo

Para una lista de valores λ = [0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30, 100] los resultados son:

```
··· Haciendo el descenso de gradiente ···
 Probando con lamda = 0.01
* 37.5% *
 Probando con lamda = 0.03
* 40.5% *
 Probando con lamda = 0.1
* 39.0% *
 Probando con lamda = 0.3
* 39.5% *
 Probando con lamda = 1
* 39.0% *
 Probando con lamda = 3
* 38.5% *
 Probando con lamda = 10
* 37.5% *
 Probando con lamda = 30
* 40.5% *
 Probando con lamda = 100
* 44.0% *
··· Pesos guardados guardados en OneVsAllWeights.mat ···
-> El OneVsAll tiene una precisión sobre validación del 44.0% con lamda = 100 <-
··· Comprobando la precisión sobre el conjunto de test ···
-> El OneVsAll tiene una precisión sobre test del 46.5% <-
```

Por lo tanto, para ese conjunto de valores el óptimo es 100 con un 93% de precisión con los ejemplos de entrenamiento, un 44% para los de validación y un 46.5% para test.

Código

```
def oneVsAll(X: np.array, y: np.array, num_etiquetas: int, reg: float):
    '''
    Implementa la regresión lineal multiclase (reg = término de
    regularización)
    '''
```

```
    # Columna de 1's
    m = X.shape[0]
    X = np.hstack([np.ones([m, 1]), X])

    #Creamos la matriz de thetas
    thetas = np.zeros((num_etiquetas, X.shape[1]))

    #Clasificador para cada una de las etiquetas
    for i in range (num_etiquetas):
        # Vector de 'y' para la iteración concreta
        iterY = np.copy(y)
        iterY = np.where (iterY == i, 1, 0)

        # Calculamos el vector de pesos óptimo para ese clasificador
        thetas[i] = fmin_tnc(func = regularizedCost, x0=thetas[i], fprime
=regularizedGradient, args=(reg, X, iterY.ravel()), messages=0)[0]

    return thetas


# Regresión logística multiclase #
def LogisticRegressionClassifier(data:dict, lamdas:tuple):
    '''
    Clasificador por regresión logística; entrena el modelo
    buscando el lamda óptimo y guarda los pesos en un archivo .mat
    '''
    # Atributos
    num_etiquetas = len(FLOWER_NAMES)

    # Sacamos los conjuntos del diccionario
    X_train = data["X_train"]
    y_train = data["y_train"].ravel()
    X_val = data["X_val"]
    y_val = data["y_val"].ravel()

    # Resolvemos el one vs All
    bestLamda = lamdas[0]
    bestPorc = 0
    bestThetas = np.zeros((num_etiquetas, X_train.shape[1] + 1))

    #Encontramos el mejor valor de lamda usando el conjunto de validación
    print("··· Haciendo el descenso de gradiente ··· ")
    for i in range(len(lamdas)):
        print(" Probando con lamda = " + str(lamdas[i]))
        # 1. Primero entrenamos el modelo
        thetas = oneVsAll(X_train, y_train, num_etiquetas, lamdas[i])

        # 2. Luego vemos su precisión sobre el conjunto de validación
        m = X_val.shape[0]
```

```
        unosX = np.hstack([np.ones([m, 1]), X_val])
        z = sigmoid(hMatrix(unosX, thetas))
        porc = calcula_porcentaje(y_val, z, 4)

        print("* " + str(porc) + "% *")

        # 3. Actualizamos si es mejor con este lambda
        if(porc > bestPorc):
            bestLamda = lamdas[i]
            bestPorc = porc
            bestThetas = thetas

    # Nos guardamos los pesos óptimos
    print("···Pesos guardados guardados en " + WEIGHTS_NAMES[0] + " ···")
    savemat(WEIGHTS_NAMES[0], mdict={
        'Theta1': bestThetas[0],
        'Theta2': bestThetas[1],
        'Theta3': bestThetas[2],
        'Theta4': bestThetas[3],
        'Theta5': bestThetas[4],
    })

    # Log
    print("-> El OneVsAll tiene una precisión sobre validación del " +
str(bestPorc) + "% con lamda = " + str(bestLamda) + " <-")


def TestLogisticRegression(X_test, y_test):
    '''
    Prueba el clasificador logístico sobre el conjunto de tests,
    cogiendo los pesos ya entrenados.
    '''
    # Cargamos los thetas óptimos del archivo (son 5, uno por cada flor)
    weights = loadmat(WEIGHTS_NAMES[0])
    num_etiquetas = 5 #TODO: cambiarlo
    thetaOpt = np.zeros((num_etiquetas, X_test.shape[1] + 1))
    for i in range(num_etiquetas):
        thetaOpt[i] = weights['Theta' + str(i+1)]

    # Comprobamos la efectividad
    print("··· Comprobando la precisión sobre el conjunto de test ··· ")
    m = X_test.shape[0]
    unosX = np.hstack([np.ones([m, 1]), X_test])
    z = sigmoid(hMatrix(unosX, thetaOpt))
    porc = calcula_porcentaje(y_test, z, 4)

    # Log
    print("-> El OneVsAll tiene una precisión sobre test del " +
str(porc) + "% <-")
```

Red Neuronal

Funcionamiento

Cuenta con 3 capas la de entrada y la de salida y una oculta.

El número de nodos de entrada corresponde al tamaño de la imagen al cuadrado multiplicado por tres, por los atributos RGB. La capa de salida cuenta con 5 nodos, uno para cada tipo de flor.

Los parámetros los recibe en listas de valores. Lambda, el número de iteraciones y el número de capas ocultas son tuplas, prueba todas las combinaciones posibles y selecciona la óptima.

1. Entrena la red neuronal y calcula los pesos óptimos

```
def trainNeutralNetwork(num_entradas, num_ocultas, num_etiquetas, X,y,
 lamda, num_iter):
    '''
    Entrena una red neuronal de 2 capas y devuelve las matrices de
    pesos para cada capa
    La "y" debe estar en formato onehot
    '''
    # 1. Comenzamos con unos pesos aleatorios
    theta1 = pesosAleatorios(num_entradas, num_ocultas)
    theta2 = pesosAleatorios(num_ocultas, num_etiquetas)
    nn_params = np.concatenate((theta1.ravel(), theta2.ravel())) #Los
    # unimos en 1 solo vector

    # 2. Llamamos a la función minimize para obtener las matrices de
    #    pesos óptimos
    # (las que hacen que haya un mínimo en el coste devuelto,
    # usando back_prop)
    thetaOpt = minimize(fun=back_prop,
                        x0=nn_params,
                        args=(num_entradas,
                              num_ocultas,
                              num_etiquetas,
                              X, y, lamda),
                        method='TNC',
                        jac=True,
                        options={'maxiter':num_iter}).x

    # 3. Tenemos que reconstruir los pesos a partir del vector
    theta1 = np.reshape(thetaOpt[:num_ocultas * (num_entradas + 1)],
                        (num_ocultas, num_entradas + 1))
    theta2 = np.reshape(thetaOpt[num_ocultas * (num_entradas + 1):],
                        (num_etiquetas, num_ocultas + 1))

    # Devolvemos los pesos óptimos
    return [theta1, theta2]
```

2. Hace la propagación hacia delante.

```
def forward_prop(X, theta1, theta2):
    '''
    Propagación hacia delante en la red neuronal de 2 capas
    '''
    m = X.shape[0]

    a1 = np.hstack([np.ones([m, 1]), X])
    z2 = np.dot(a1, theta1.T)
    a2 = np.hstack([np.ones([m, 1]), sigmoid(z2)])
    z3 = np.dot(a2, theta2.T)
    h = sigmoid(z3)

    return a1, z2, a2, z3, h
```

3. Calcula el porcentaje sobre validación

```
def calcula_porcentaje(Y, Z, digitsNo: int):
    '''
    Calcula el porcentaje de aciertos del entrenador
    '''

    m = Y.shape[0]

    # Creamos la matriz
    results = np.empty(m)

    # Recorremos todos los ejemplos de entrenamiento...
    for i in range (m):
        results[i] = np.argmax(Z[i])
    results = results.T

    # Vemos cuántos de ellos coinciden con Y
    coinciden = ( Y == results )
    aciertos = np.sum(coinciden)

    # Porcentaje sobre el total de ejemplos redondeado
    return round((aciertos / m) * 100, digitsNo)
```

4. Si el resultado es mejor que cualquier otro obtenido hasta ahora se establece ese como el mejor.
5. Una vez se han probado todas las combinaciones se guardan los pesos óptimos calculados en un archivo *.mat* para luego comprobarlos con los ejemplos de test.

Ejemplo

Tras muchas pruebas los valores óptimos son 25 capas ocultas, λ = 1 y 210 iteraciones. Que dan como resultado un 93% en entrenamiento, 37.5% en validación y 47% para test.

```
Probando con 25 ocultas, lamda = 1 y 210 it.
* 37.5% *
100%|                                                                                                  | 1/1 [09:00<00:00, 540.54s/it]
·· Pesos guardados guardados en NetworkWeights.mat ···
-> La red tiene una precisión sobre validacion del 37.5 % con 25 capas ocultas, lamda = 1 y 210 iteraciones <-
··· Pesos guardados guardados en NetworkWeights.mat ···
··· Comprobando la precisión sobre el conjunto de test ···
-> La red tiene una precisión del  47.0  % <-
```

Código

```python
# Red neuronal de 2 capas #
def NeutralNetworkClassifier(data:dict, ocultas:tuple, lamdas:tuple, iters:tuple):
    '''
    Clasificador por red neuronal de 2 capas
    Recibe el término de regularización, el número de capas ocultas
    y las iteraciones máximas para el descenso de gradiente
    '''
    # 1. Montamos la red neuronal
    # Atributos
    num_entradas = IMG_SIZE * IMG_SIZE * 3 # 3 por cada píxel (R, G, B)
    # num_ocultas = nos la pasan como parámetro
    num_etiquetas = len(FLOWER_NAMES) # 5

    # Sacamos los datos del diccionario
    X_train = data["X_train"]
    y_train_onehot = makeOneHot(data["y_train"].ravel(), num_etiquetas)
    X_val = data["X_val"]
    y_val = data["y_val"].ravel()
    m = X_train.shape[0]

    # Mejores parámetros para la red
    bestOcultas = ocultas[0]
    bestLamda = lamdas[0]
    bestIters = iters[0]
    bestThetas = []
    bestPorc = 0

    print("··· Entrenando la red neuronal (puede tardar varios lustros) ···")
    # Barra de progreso
    pbar = tqdm(total= (len(ocultas) * len(lamdas) * len(iters)))

    # Probamos todas las combinaciones de valores que nos pasan
    for i in range (len(ocultas)):
        for j in range (len(lamdas)):
```

```
            for k in range(len(iters)):
                print(" Probando con " + str(ocultas[i]) + " ocultas, lam
da = " + str(lamdas[j]) + " y " + str(iters[k]) + " it.")
                # La entrenamos y cogemos los pesos óptimos (es el código
                # de la Práctica 4)
                theta1, theta2 = trainNeutralNetwork(num_entradas, oculta
s[i], num_etiquetas, X_train, y_train_onehot, lamdas[j], iters[k])

                # 2. Con los pesos obtenidos, hacemos la propagación
                # hacia delante y vemos el porcentaje sobre validación
                a1, z2, a2, z3, h = forward_prop(X_val, theta1, theta2)
                porcentaje = calcula_porcentaje(y_val, h, 3)

                print("* " + str(porcentaje) + "% *")

                # Comprobamos que sea mejor el porcentaje
                if(porcentaje > bestPorc):
                    bestPorc = porcentaje
                    bestOcultas = ocultas[i]
                    bestLamda = lamdas[j]
                    bestIters = iters[k]
                    bestThetas = [theta1, theta2]

                pbar.update(1) #Actualizar el GUI

    # Guardamos los pesos óptimos
    print("··· Pesos guardados guardados en " + WEIGHTS_NAMES[1] + " ···
")
    savemat(WEIGHTS_NAMES[1], mdict={
        'Theta1': theta1,
        'Theta2': theta2})

    # Sacamos el porcentaje de aciertos
    print("-
> La red tiene una precisión sobre validacion del " +  str(bestPorc) +  "
 % con " + str(bestOcultas) + " capas ocultas, lamda = " + str(bestLamda)
 + " y " + str(bestIters) + " iteraciones <-")


def TestNeutralNetwork(X_test, y_test):
    '''
    Prueba la red neuronal sobre el conjunto de tests,
    cogiendo los pesos ya entrenados.
    '''

    # Cargamos los pesos óptimos desde la matriz
    print("··· Pesos guardados guardados en " + WEIGHTS_NAMES[1] + " ···
")
    weights = loadmat(WEIGHTS_NAMES[1])
```

```
    theta1 = weights['Theta1']
    theta2 = weights['Theta2']

    # 2. Con los pesos óptimos obtenidos, hacemos la propagación hacia
    # delante y obtenemos la predicción de la red
    print("··· Comprobando la precisión sobre el conjunto de test ··· ")
    a1, z2, a2, z3, h = forward_prop(X_test, theta1, theta2)

    # Sacamos el porcentaje de aciertos
    porcentaje = calcula_porcentaje(y_test, h, 3)
    print("-> La red tiene una precisión del ",  porcentaje, " % <-")
```

Support Vector Machines (SVM)

Funcionamiento

Recibe los datos, el tipo de kernel que es y una lista de valores para sigma y C.

Lo primero es elegir los mejores valores para C y sigma entre los que le hemos pasado por parámetro. Para esto se comprueba el resultado de todas las combinaciones posibles.

Se hace el SVM con kernel gaussiano, ya que es multiclase, se entrena y se comprueba qué porcentaje de aciertos tiene con los datos de validación. Si es mejor que el último mejor guardado, de los anteriores, lo actualizamos.

Finalmente se devuelve el mejor valor y los parámetros usados para ello.

```
def getBestSVMMultiClass(kernelType:str, values:list, Xtrain, ytrain,
      Xval, yval):
    '''
    A partir del conjunto de datos de validación, encuentra los
    hiperparámetros (C y sigma de entre la lista que se da) que hacen
    el porcentaje de aciertos mayor
    Tarda bastante en ejecutarse si la lista tiene más de 4 elementos
    '''
    #Lista de modelos
    svm = []

    mejor = 0 #El mejor modelo con la validación
    mejor_porc = 0

    cOPt = values[0]
    sigmaOpt = values[0]

    #Todas las posibles combinaciones de modelos con los valores dados
    #para C y sigma
    actual = 0
    for i in range (len(values)):
```

```
        for j in range (len(values)):
            print(" Probando con C = " + str(values[i]) + ", sigma = " +
str(values[j]))
            # Hacemos el SVM con kernel gaussiano
            svm.append(SVC(kernel=kernelType, C=values[i], gamma=1 / (2 *
 values[j] ** 2)))
            #Lo entrenamos con el conjunto de entrenamiento
            svm[actual] = OneVsRestClassifier(svm[actual]) # Lo convertim
os en multiclass
            svm[actual].fit(Xtrain, ytrain)

            #Vemos el porcentaje de aciertos sobre el conjunto de val
            h = svm[actual].predict(Xval)
            porc = calcula_porcentaje_Y(yval.ravel(), h, 4)

            print("* " + str(porc) + "% *")

            #Si el porcentaje es mejor que el máximo, actualizamos
            if(porc > mejor_porc):
                mejor = actual
                mejor_porc = porc
                cOpt = values[i]
                sigmaOpt = values[j]

            actual+=1 #Avanzamos

    #Devolvemos el mejor, junto a los parámetros usados
    return svm[mejor], cOPt, sigmaOpt
```

Una vez se han obtenido los parámetros óptimos guarda los valores en un archivo para utilizarlos luego al hacer la comprobación con los valores de test, y se calcula el porcentaje para los valores de validación.

Por último, se llama al método encargado de comprobar el porcentaje en los casos de test.

Ejemplo

Tras numerosas pruebas concluimos que los mejores valores para C y sigma son 3 y 30, correspondientemente.

```
··· Entrenando la SVM (puede tardar varios minutos) ···
 Probando con C = 3, sigma = 3
* 16.5% *
 Probando con C = 3, sigma = 30
* 53.5% *
 Probando con C = 30, sigma = 3
* 16.5% *
 Probando con C = 30, sigma = 30
* 49.5% *
··· SVM guardada en SVM.joblib ···
··· Comprobando la precisión sobre el conjunto de validación con C = 3, sigma = 30 ···
-> La SVM tiene una precisión del  52.0 % <-
··· Comprobando la precisión sobre el conjunto de test ···
-> La SVM tiene una precisión del  52.0 % <-
```

Otros ejemplos:

```
 Probando con C = 0.01, sigma = 0.01
* 18.5% *
 Probando con C = 0.01, sigma = 0.03
* 18.5% *
 Probando con C = 0.01, sigma = 0.1
* 18.5% *
 Probando con C = 0.01, sigma = 0.3
* 18.5% *
 Probando con C = 0.01, sigma = 1
* 18.5% *
 Probando con C = 0.01, sigma = 3
* 18.5% *
 Probando con C = 0.01, sigma = 10
* 46.0% *
 Probando con C = 0.01, sigma = 30
* 49.0% *
 Probando con C = 0.03, sigma = 0.01
* 18.5% *
 Probando con C = 0.03, sigma = 0.03
* 18.5% *
 Probando con C = 0.03, sigma = 0.1
* 18.5% *
```

```
 Probando con C = 0.03, sigma = 0.3
* 18.5% *
 Probando con C = 0.03, sigma = 1
* 18.5% *
 Probando con C = 0.03, sigma = 3
* 18.5% *
 Probando con C = 0.03, sigma = 10
* 44.5% *
 Probando con C = 0.03, sigma = 30
* 49.5% *
* 18.5% *
 Probando con C = 0.1, sigma = 0.03
* 18.5% *
 Probando con C = 0.1, sigma = 0.1
* 18.5% *
 Probando con C = 0.1, sigma = 0.3
* 18.5% *
 Probando con C = 0.1, sigma = 1
* 18.5% *
 Probando con C = 0.1, sigma = 3
* 18.5% *
 Probando con C = 0.1, sigma = 10
```

Código

```
# Support Vector Machines #
def SVMClassifier(data:dict, kernelType:str, values:list):

    # Sacamos los datos del diccionario
    X_train = data["X_train"]
    y_train = data["y_train"].ravel()
    X_val = data["X_val"]
    y_val = data["y_val"].ravel()


    #Cogemos el mejor modelo
    print("··· Entrenando la SVM (puede tardar varios minutos) ··· ")
    svm, cOpt, sigmaOpt = getBestSVMMultiClass(kernelType, values, X_trai
n, y_train, X_val, y_val)

    #Guardamos la SVM
    print("··· SVM guardada en " + WEIGHTS_NAMES[2] + " ··· ")
    dump(svm, WEIGHTS_NAMES[2])

    # Vemos el porcentaje de aciertos sobre el conjunto de tests
    print("··· Comprobando la precisión sobre el conjunto de validación c
on C = " + str(cOpt) + ", sigma = " + str(sigmaOpt) + " ··· ")
    h = svm.predict(X_test)
    porcentaje = calcula_porcentaje_Y(y_test, h, 4)
```

```
    print("-> La SVM tiene una precisión del ",  porcentaje, "% <-")



def TestSVMClassifier(X_test, y_test):
    '''
    Prueba la red neuronal sobre el conjunto de tests,
    cogiendo los pesos ya entrenados.
    '''

    #Cargamos
    svmMultiClass = load(WEIGHTS_NAMES[2])

    # Vemos el porcentaje de aciertos sobre el conjunto de tests
    print("··· Comprobando la precisión sobre el conjunto de test ··· ")
    h = svmMultiClass.predict(X_test)
    porcentaje = calcula_porcentaje_Y(y_test, h, 4)

    print("-> La SVM tiene una precisión del ",  porcentaje, "% <-")
```