Practica 3

Regresión logística multi-clase y redes neuronales

Aaron Reboredo Vázquez, Pablo Martín García

Previo a la implementación de las funciones necesarias para llevar a cabo la práctica es interesante tener métodos que nos permitan cargar los datos de archivos externos y otros que nos permitan visualizar en pantalla una primera representación de esos valores.

Utilizaremos los métodos **load_data y load_data_neuronal_red** para volcar en variables los datos obtenidos de los diferentes archivos para las partes uno y dos respectivamente.

```
def load_data(file_name):
    data = loadmat(file_name)

    y = data['y']
    y_2 = np.ravel(y)
    X = data['X']

    return y_2, X

def load_data_neuronal_red(file_name):
    weights = loadmat(file_name)
    theta1 , theta2 = weights [ 'Theta1' ] , weights [ 'Theta2' ]

    return theta1, theta2
```

Así mismo se nos brinda un método que nos permite imprimir una imagen que representa de manera gráfica un conjunto aleatorio de muestras para poder visualizar el tipo de datos con el que vamos a trabajar.

```
def draw_rnd_selection_data(X):
    sample = np.random.choice(X.shape[0], 10)
    plt.imshow(X[sample, :].reshape(-1, 20).T)
    plt.axis('off')
    plt.show()
```

Finalmente y durante la práctica necesitaremos hacer uso de las funciones sigmoide y hipótesis que venimos utilizando en ocasiones anteriores.

```
def H(X, Z): #Hipótesis del mocelo lineal vectorizada
    return np.dot(X, Z)
```

```
def g_z(X):

    e_z = 1 / np.power(math.e, X) #np.power => First array elements raise
d to powers from second array, element-wise.

    sigmoide = 1/(1 + e_z)

    return sigmoide
```

Regresión logística multi-clase

Los primeros pasos para el reconocimiento de imágenes mediante regresión logística es necesario implementar las funciones propias de la regresión, en este caso utilizaremos las funciones de gradiente y costes regularizados que ya habíamos visto en prácticas anteriores.

```
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 = g_z(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 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(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 = g_z(X_Teta)

    X_T = np.transpose(X)

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

    return gradient

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
```

Así mismo también haremos uso de la función que utiliza **opt.fmin_tnc** que nos permite calcular los thetas óptimos para un conjunto de muestras y unas funciones de gradiente y costes dadas.

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

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

    return theta_opt
```

Mediante las funciones de coste y de gradiente, así como la función que nos permite obtener los thetas óptimos, podemos entrenar un clasificador que obtenga los thetas óptimos para cada una de las clases y que nos permitirá hacer predicciones a la hora de clasificar un elemento entrante.

Se nos pide que hallemos la matriz de thetas resultante de entrenar el clasificador y que posteriormente calculemos el porcentaje de acierto a la hora de clasificar los resultados respecto al resultado real.

La función a implementar es la llamada **oneVsAll(X, y, num_etiquetas , reg)** y que nos devuelve una matriz en el que cada fila se corresponde con los thetas óptimos para cada una de las clases o etiquetas en las que se divide la entrada :

```
def oneVsAll(X, y, num_etiquetas, reg, Thetas):

    y_= (y == 10).astype(np.int)
    Thetas_ = optimized_parameters_regularized(Thetas, X, y_, reg)

    optimiced_parameters_matrix = Thetas_

    for i in range(1, num_etiquetas):
        y_ = (y == i).astype(np.int)
        Thetas_ = optimized_parameters_regularized(Thetas, X, y_, reg)
        optimiced_parameters_matrix = np.vstack((optimiced_parameters_mat
rix, Thetas_))


    return optimiced_parameters_matrix
```

Para ello nos apoyamos en la función **optimized_parameters_regularized(Thetas, X, Y, reg)** que es la que nos devuelve el array de thetas correspondientes para cada entrada

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

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

    return theta_opt
```

Para que la regularización funcione los valores de Y deben tomar valores de 0 y 1. De esta manera debemos transformar el Y de entrada para que adquiera la forma que nos permitirá calcular los thetas óptimos para cada caso.

La transformación convierte en 1 todos los resultados equivalentes al valor de la clase o la etiqueta para la que se están calculando los thetas óptimos y en 0 el resto.

De esta manera si estamos calculando los thetas óptimos para la etiqueta o la clase de valor 7 y nuestro vector Y de entrada es el [0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6 7 7 7 8 8 8 9 9 9], nuestro vector Y_ transformado que utilizaremos para el cálculo de nuestros thetas será el

[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0].

De esta manera vamos iterando y calculando nuestra matriz con todos los thetas para cada una de las clases, pasando en cada caso la matriz de entrada X y las transformadas para cada caso o vuelta de la iteración, que en este caso coinciden con los valores o etiquetas de las diferentes clases.

A la hora de calcular el porcentaje de acierto es necesario ver los valores que obtendríamos haciendo uso de nuestro clasificador y comparar este con los resultados reales dados.

Para ello vamos a aplicar a la matriz de entrada los thetas óptimos para cada una de las clases y calcular el sigmoide de la función hipótesis para todos los casos de entrada (recordad que cada fila es un caso). Obtenemos un array de sigmoides, correspondiente a cada una de las muestras, y resultado de aplicar cada una de las ristras de thetas optimas de cada etiqueta. El valor más alto se encontrará en la posición equivalente al valor de la etiqueta o la clase a la que el comparador predice que se encuentra cada una de muestras.

Nuestra función **`Sample_clasifier(sample, num_etiquetas, Thetas_matrix)`** nos devuelve la etiqueta a la que se supone que pertenece nuestra muestra sample.

```
def Sample_clasifier(X, num_etiquetas, Thetas_matrix):

    sigmoids = np.zeros(num_etiquetas)

    for i in range(num_etiquetas): #selects the theta optimized values fo
r each tag or numner, the coincidences or number ones will be greater in
the elements that fits with the
                                        #same number the thetas values re
present
        Z_ = Thetas_matrix[i, :]

        H_ = H(X, Z_)
        H_sigmoid = g_z(H_)
        sigmoids[i] = H_sigmoid

    num_tag = np.argmax(sigmoids)

    return num_tag
```

Vamos recogiendo en un nuevo array todos los valores de las etiquetas obteniendo el valor Y predicho por el comparador y que será el que tendremos que comparar con el original para comprobar la tasa de acierto.

Es nuestro método **all_samples_comparator_percentage(X, y, num_etiquetas, reg, Thetas)** con el uso de el anterior el que va iterando por todas las muestras y va obteniendo el nuevo vector de predicciones. Finalmente compara ambos resultados y nos da el porcentaje de acierto deseado, en nuestro caso 94,4%.

```
def all_samples_comparator_percentage(X, y, num_etiquetas, reg, Thetas):

    Thetas_matrix = oneVsAll(X, y, num_etiquetas, reg, Thetas)

    samples = X.shape[0]
    y_ = np.zeros(samples)
    y = np.where(y == 10, 0, y)

    for i in range(samples):
        y_[i] = Sample_clasifier(X[i, :], num_etiquetas, Thetas_matrix)


    percentage = vectors_coincidence_percentage(y_, y)

    return percentage
```

Para comparar los Ys utilizamos la función auxiliar **vectors_coincidence_percentage(y_, y),** que compara uno a uno los valores de los dos arrays de entrada y devuelve el porcentaje de coincidencia.

```
def vectors_coincidence_percentage(a, b):

    coincidences_array = a == b

    coincidences = sum(map(lambda coincidences_array : coincidences_array
 == True, coincidences_array  ))
    percentage =100 * coincidences/coincidences_array.shape

    return percentage
```

Tenemos que tener en cuenta la transformación realizada en el vector original de entrada a la hora de realizar la comparación, ya que en aquel caso las etiquetas para el 0 tenían un valor de 10 que ahora hemos transformado en 0s.

Redes neuronales

Para este apartado se nos pide que implementemos la propagación hacia delante de la red neuronal. Esto nos devolverá una matriz con los sigmoides correspondientes de evaluar para cada muestra las posibilidades de pertenecer a una clase u a otra. Nuevamente el valor más alto estará en el índice correspondiente con la etiqueta o la clase a la que pertenece cada muestra.

```
def forward_propagation(X, theta1, theta2):

    #V1
    a1 = X
    a1_ones = np.hstack([np.ones([a1.shape[0],1]),a1])
    z2 = H(a1_ones, np.transpose(theta1))
    a2 = g_z(z2)

    a2_ones = np.hstack([np.ones([a2.shape[0],1]),a2])

    z3 = H(a2_ones, np.transpose(theta2))
    a3 = g_z(z3)

    y = a3

    return y
```

```
def neuronal_succes_percentage(X, y, theta1, theta2) :

    sigmoids_matrix = forward_propagation(X, theta1, theta2)
    y_ = neuronal_prediction_vector(sigmoids_matrix)
    print(y_)
    percentage = vectors_coincidence_percentage(y_, y)

    return percentage
```

Una vez obtenidos los sigmoides a partir de la propagación forward nos ayudamos de la función **neuronal_prediction_vector(sigmoids_matrix)** que analizan cada uno de los arrays de sigmoides para cada ejemplo de entrada y determina a que clase pertenece.

En este caso no debemos ajustar los arrays para que sustituya el 10 por un 0 a la hora de hacer la comparación, pues los dos que utilizamos ya tienen esa codificación por defecto.

La posición en el array fila nos indica a que clase pertenece. El sistema saca una matriz en la que la última posición se encuentran los valores que hacen referencia al 0, que se codifica en el vector de Y como 10, de esta manera al valor obtenido en la columna tenemos que sumarle un uno para que la correspondencia con la clase a la que pertenece sea total.

```
def neuronal_prediction_vector(sigmoids_matrix) :
    samples = sigmoids_matrix.shape[0]
    y = np.zeros(samples)

    for i in range(samples):
        y[i] = np.argmax(sigmoids_matrix[i, :]) + 1

    return y
```

Finalmente podemos calcular el porcentaje de coincidencia o éxito de aplicar la red neuronal para hacer la clasificación de las muestras, que en nuestro caso es del 97,5 %.