Práctica 3: Regresión logística multi-clase y redes neuronales

Manuel Hernández Nájera-Alesón
Mario Jiménez Contreras
Grupo 17

Parte 1: Regresión Logística multi-clase

1.1 Planteamiento

Hacemos uso de la función sigmoide, puesto que agrupamos los valores de entrenamiento en ceros y unos.

$$h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$$

La función de coste es igual que la utilizada en regresión lineal, pero puede simplificarse teniendo en cuenta que la hipótesis empleada es un logaritmo, por lo que la función resultante es:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left[-y^{(i)}\log(h_\theta(x^{(i)})) - (1-y^{(i)})\log(1-h_\theta(x^{(i)}))\right]$$

De esta manera además, la vectorización es más sencilla.
Función de descenso de gradiente, de nuevo igual que en regresión lineal.

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

Llegamos a la parte de regularización. Cuando empezamos a tener en cuenta una gran cantidad de características en nuestra función, aumenta el grado del polinomio de la función, podríamos sobreajustarnos a los datos de entrenamiento y perderíamos la capacidad de generalizar sobre otros propuestos en problemas de la misma naturaleza, por eso se recurre a estas funciones.

Función de coste regularizada.

$$J(\theta) = \left[\frac{1}{m}\sum_{i=1}^{m}\left[-y^{(i)}\log(h_\theta(x^{(i)})) - (1-y^{(i)})\log(1-h_\theta(x^{(i)}))\right]\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

Función de descenso de gradiente regularizada

$$\left(\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}\right) + \frac{\lambda}{m}\theta_j$$

Como función clasificatoria hemos utilizado uno frente a todos, que funciona como el nombre indica, cogemos una clase y creamos una pasada de entrenamiento cogiendo el resto de clases como una única, hacemos esto con todas las clases, creando todas las combinaciones posibles y con ellas, todos los clasificadores posibles, que coincide con el número de clases implicadas. De todos los obtenidos, escogemos el de mayor valor.

1.2 Código:

Sigmoide:

```
def sigmoid(z):
   return 1.0/(1.0 + np.exp(-z))
```

Hipótesis:

```
def hipotesis (thetas, X):
    return sigmoid(X.dot(thetas))
```

Función de coste. En nuestro caso separamos la función en varios operadores, para mayor comodidad y menor riesgo de equivocación:

```
def costFunc(thetas, X, Y):
   H = sigmoid(np.dot(X, thetas))
   oper1 = -(float(1)/len(X))
   oper2 = np.dot((np.log(H)).T, Y)
   oper3 = (np.log(1-H)).T
   oper4 = 1-Y
   return oper1 * (oper2 + np.dot(oper3, oper4))
```

Descenso de gradiente:

```
def alg_gradDesc(thetas, X, Y):
    H = sigmoid(np.dot(X, thetas))
    return np.dot((1.0/len(X)), X.T).dot(H-Y)
```

Función de coste y descenso de gradiente regularizadas, para evitar sobreajuste.

```
def costFunc_reg(thetas, X, Y, _lambda):
    H = sigmoid(np.dot(X, thetas))
    oper1 = -(float(1)/len(X))
    oper2 = np.dot((np.log(H)).T, Y)
    oper3 = (np.log(1-H)).T
    oper4 = 1-Y
    oper5 = (_lambda/(2*len(X)))*np.sum(thetas**2)
    return (oper1 * (oper2 + np.dot(oper3, oper4)))+ oper5
```

```
def alg_gradDesc_reg(thetas, X, Y, _lambda):
    H = sigmoid(np.dot(X, thetas))
    return (np.dot((1.0/len(X)),
X.T).dot(H-Y))+(_lambda/len(X))*thetas
```

Como función clasificatoria hemos utilizado uno frente a todos.

```
def oneVsAll(X, Y, num_etiquetas, reg):
    thetas = np.zeros([num_etiquetas, X.shape[1]]) #Thetas es un
vector de shape (num_etiquetas, 401)

    for i in range(num_etiquetas):
        if(i == 0):
             iaux = 10
        else:
            iaux = i

        a = (Y == iaux)*1
        thetas[i] = scopt.fmin_tnc(costFunc_reg, thetas[i],
alg_desGrad_reg,args = (X, a, reg))[0]
```

```
    return thetas
```

1.3 Procedimiento:

```
#Extraccion de los datos
data = loadmat('ex3data1.mat')
Y = data['y']  # Representa el valor real de cada ejemplo de
entrenamiento de X (y para cada X)
X = data['X']  # Cada fila de X representa una escala de grises de
20x20 desplegada linearmente (400 pixeles)
nMuestras = len(X)
X_unos = np.hstack([np.ones((len(X),1)),X])
thetas = np.zeros(len(X_unos[0]))
Y = np.ravel(Y)

thetas_opt = oneVsAll(X_unos, Y, 10, 0.1)
resultados = hipotesis(thetas_opt.T, X_unos) #Resultados es un array
de (10, 5000) con el resultado de aplicar hipotesis a X

prediccion = resultados.argmax(axis = 1) #Este será un array de (1,
5000) con las posibilidades de que un numero haya sido predecido
correctamente
prediccion[prediccion == 0] = 10

Z = (prediccion == Y)*1
probabilidad = sum(Z)/len(Y)

print("La probabilidad de acierto del programa es: ",
probabilidad*100, "%")
```

1.4 Resultados:

El porcentaje de aciertos con el clasificador elegido es de 96.78%, es un gran resultado pero requiere mucho tiempo de cómputo.

Parte 2: Redes neuronales:

2.1 Planteamiento:

Para poder resolver el problema anterior pero en mucho menos tiempo.

2.2 Código:

```
data = loadmat('ex3data1.mat')
Y = data['y']  # Representa el valor real de cada ejemplo de
entrenamiento de X (y para cada X)
X = data['X']  # Cada fila de X representa una escala de grises de
20x20 desplegada linearmente (400 pixeles)
Y = np.ravel(Y)
X_unos = np.hstack([np.ones((len(X), 1)),X])
nMuestras = len(X)
weights = loadmat('ex3weights.mat')
theta1, theta2 = weights['Theta1'], weights ['Theta2']

aux = sigmoid(X_unos.dot(theta1.T))
aux = np.hstack([np.ones((len(aux),1)), aux])
#El resultado de utilizar la red neuronal será una matriz de 5000 x
10, con las probabilidades de que cada caso sea un numero.
results = sigmoide(aux.dot(theta2.T))

prediccion = results.argmax(axis = 1)+1 #Este será un array de (1,
5000) con las posibilidades de que un numero haya sido predecido
correctamente

Z = (prediccion == Y)*1
probabilidad = sum(Z)/len(Y)

print("La probabilidad de acierto del programa es: ",
probabilidad*100, "%")
```

2.2 Resultados:

Probabilidad de éxito de 97,52%