Entrega 4: clasificación multi-clase

Aprendizaje Automatico y Big Data- Alejandro Barrachina Argudo

Introducción

En este documento se explicará el código del entregable 4 y el proceso del clasificador multi-clase. Esta práctica se divide en 2 apartados: clasificador multi-clase y redes neuronales. El *dataset* consiste de imágenes de números escritos a mano como los vistos en la figura 0.3

Para esta práctica se usarán los siguientes *imports* vistos en la figura 0.1. Parte del código se reutiliza de la práctica anterior. A los *strings* estáticos anteriores añadimos uno para guardar el modelo entrenado (figura 0.2)

```
1 import csv
2 import numpy as np
3 import scipy.io as sio
4 import concurrent.futures
5 import matplotlib.pyplot as plt
6 import os
7 import logistic_reg
```

Figura 0.1: Código de las bibliotecas usadas

```
1 model_folder = "./models/"
```

Figura 0.2: Código de los nuevos strings estáticos

Figura 0.3: Ejemplo de los dígitos del *dataset*

1. Parte A: clasificación multi-clase

Para esta sección vamos a reutiliza todo lo referente a la regresión logística de la práctica anterior. Los cambios implementados son la distinción entre clases distintas y la concurrencia para acortar los tiempos de entrenamiento.

Para la concurrencia usaremos 3 funciones distintas: *train_model*, *train_all* y *oneVsAll* (figuras 1.4, 1.5 y 1.6 respectivamente). La función *train_model* es la misma que la de la práctica anterior, pero ahora se le pasa un parámetro extra, *class_num*, que indica la clase que se está entrenando. La función *train_all* se encarga de generar los hilos para entrenar cada etiqueta y juntar todos los resultados. La función *train_model* se encarga de generar los datos de entrenamiento (α, λ y el número de iteraciones.) La función *oneVsAll* hace de wrapper para las dos funciones anteriores. Finalmente, la función *run_one_vs_all* (figure 1.3) se encarga de hacer de driver para esta sección.

Tras esto, utilizamos la función *predictOneVsAll* (figura 1.7) para predecir la clase de un conjunto de datos. Esta función es muy similar a la de la práctica anterior, pero ahora diferencia entre los distintos tipos de clases dadas según el *dataset*. La función *run_one_vs_all* será la función principal de este apartado. Cargaremos los datos del *dataset* desde aquí y daremos valores al número de clases y a λ, así como mostraremos los distintos gráficos y predicciones.

Para la elaboración de los gráficos usaremos las funciones *plot_confusion_matrix* (figura 1.8) para hacer la matriz de confusión (figura 1.1) y *print_predictions* (figura 1.9) para mostrar las predicciones a lo largo del entrenamiento (figura 1.2).

Tras una sesión de entrenamiento con $\alpha = 0{,}2$, $\lambda = 0{,}01$ y 2500 iteraciones, obtenemos una predicción del 94.7 %, con estos gráficos (matriz de confusión 1.1 y predicciones 1.2) para ver mejor los resultados. Con la función *save_model* (figura 1.10) guardamos el modelo para futuras predicciones y para evitar nuevos entrenamientos. Las funciones *save_predict* y *save_last_predictions* (figura 1.11 y 1.12) guardan las predicciones en distintos tipos de archivo para poder usarlos en otras ocasiones para hacer gráficos.

Las últimas predicciones de cada label son:

0	1	2	3	4	5	6	7	8	9
0.981	0.9976	0.9878	0.9894	0.9818	0.9932	0.9762	0.9694	0.995	0.9816

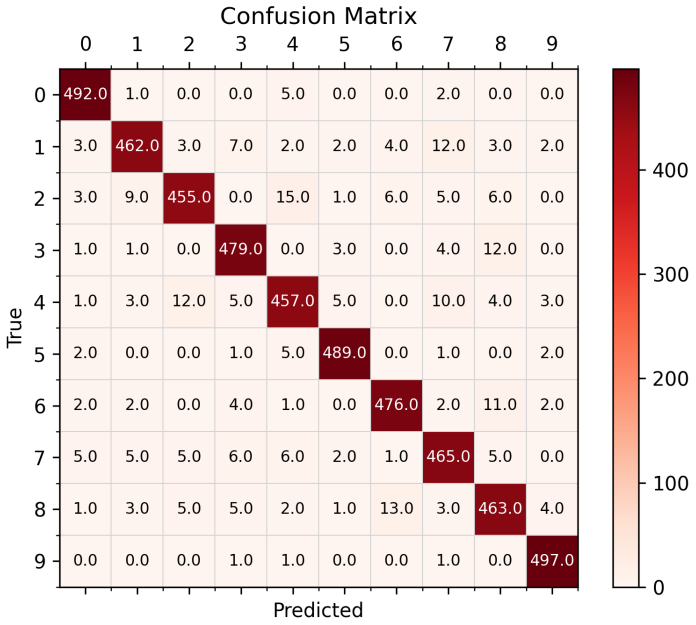

Figura 1.1: Matriz de confusión

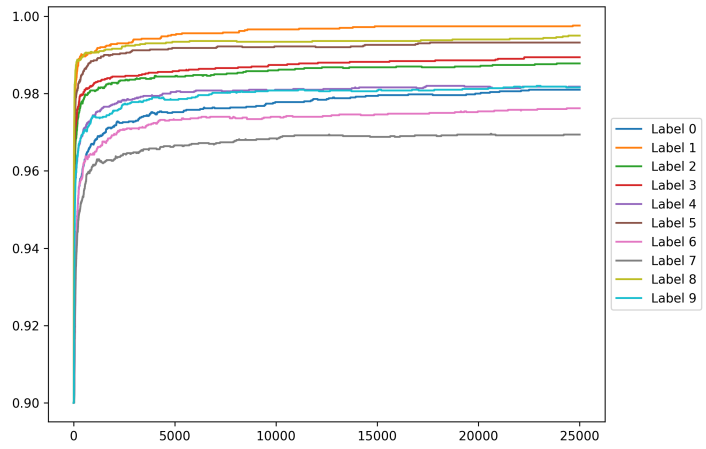

Figura 1.2: Predicciones

```python
def run_one_vs_all() -> None:
    """Runs the oneVsAll simulation
    """
    data = sio.loadmat('./data/ex3data1.mat', squeeze_me=True)
    X = data['X']
    y = data['y']
    m, n = X.shape
    num_labels = 10
    lambda_ = 0.01
    all_theta = []
    predict_all = []

    if not os.path.isfile(model_folder + "one_vs_all_train.mat"):
        print("Training one vs all...")
        all_theta, predict_all = oneVsAll(X, y, num_labels, lambda_)
        print(f'saving model to {model_folder}...')
        save_model(all_theta, model_folder + "one_vs_all_train.mat")
        print(f'saving predictions to {model_folder}...')
        save_predict(predict_all, model_folder + "one_vs_all_predict.mat")
        print(f'saving last predictions to {model_folder}...')
        save_last_predictions(predict_all)
    else:
        print("Loading model from file...")
        aux = sio.loadmat(
            f'{model_folder}one_vs_all_train.mat', squeeze_me=True)
        all_theta = np.array(aux['all_theta'])
        print('Loading predictions from file...')
        predict_all = np.array(sio.loadmat(
            f'{model_folder}one_vs_all_predict.mat', squeeze_me=True)['predict'])

    print('Plotting predictions...')
    print_predictions(predict_all, 25001)

    p = predictOneVsAll(all_theta, X)
    print(f'Prediction for A: {(np.sum(np.array(p) == y) / m) * 100}')

    print('Plotting confusion matrix...')
    plot_confusion_matrix(y, p, logistic_reg.plot_folder +
                          "confusion_matrix_one_vs_all.png")
```

Figura 1.3: Código de la función *run_one_vs_all*

```python
def train_model(c: int, X: np.ndarray, y: np.ndarray, lambda_: float) -> tuple[int, (np.ndarray, np.ndarray)]:
    """Trains a single label using logistic regression.

    Args
        c (int): The label to train
        X (np.ndarray): The input data
        y (np.ndarray): The values of the data
        lambda_ (float): The regularization parameter
    Returns:
        tuple[int, (np.ndarray, np.ndarray)]: The label, the trained parameters and the predictions
    """
    print(f"Training {c}...")
    initial_theta = np.zeros(X.shape[1] + 1)
    y_i = np.array([1 if label == c else 0 for label in y])
    alpha = 0.2
    lambda_ = 0.01
    num_iters = 25000
    (w, b, _, predict) = logistic_reg.gradient_descent(X, y_i, initial_theta[1:], initial_theta[0],
                                                       logistic_reg.compute_cost_reg, logistic_reg.compute_gradient_reg, alpha, num_iters, lambda_)
    print(f"Training {c}... Done")
    return (c, ([b] + w.tolist(), predict))
```

Figura 1.4: Código de la función *train_model*

```python
def train_all(X: np.ndarray, y: np.ndarray, n_labels: int, lambda_: float) -> tuple[list[np.ndarray], list[np.ndarray]]:
    """Trains all the labels concurrently using logistic regression.

    Args:
        X (np.ndarray): input data
        y (np.ndarray): expected values
        n_labels (int): number of labels to train
        lambda_ (float): regularization parameter

    Returns:
        tuple[list[np.ndarray], list[np.ndarray]]: all theta for all labels and the predictions for each training
    """
    all_theta = [0 for _ in range(n_labels)]
    predict_history = []
    with concurrent.futures.ThreadPoolExecutor() as executor:
        futures = [executor.submit(train_model, c, X, y, n_labels, lambda_)
                   for c in range(n_labels)]
        for future in concurrent.futures.as_completed(futures):
            c, (result, predict) = future.result()
            all_theta[c] = result
            predict_history.append(predict)
    return all_theta, predict_history
```

Figura 1.5: Código de la función *train_all*

```python
def oneVsAll(X: np.ndarray, y: np.ndarray, n_labels: int, lambda_: float):
    """
     Trains n_labels logistic regression classifiers and returns
     each of these classifiers in a matrix all_theta, where the i-th
     row of all_theta corresponds to the classifier for label i.

     Parameters
     ----------
     X : array_like
         The input dataset of shape (m x n). m is the number of
         data points, and n is the number of features.

     y : array_like
         The data labels. A vector of shape (m, ).

     n_labels : int
         Number of possible labels.

     lambda_ : float
         The logistic regularization parameter.

     Returns
     -------
     all_theta : array_like
         The trained parameters for logistic regression for each class.
         This is a matrix of shape (K x n+1) where K is number of classes
         (ie. `n_labels`) and n is number of features without the bias.
     """
    all_theta, predict_history = train_all(X, y, n_labels, lambda_)

    return all_theta, predict_history
```

Figura 1.6: Código de la función *oneVsAll*

```python
def predictOneVsAll(all_theta: np.ndarray, X: np.ndarray) -> np.ndarray:
    """
    Return a vector of predictions for each example in the matrix X.
    Note that X contains the examples in rows. all_theta is a matrix where
    the i-th row is a trained logistic regression theta vector for the
    i-th class. You should set p to a vector of values from 0..K-1
    (e.g., p = [0, 2, 0, 1] predicts classes 0, 2, 0, 1 for 4 examples) .

    Parameters
    ----------
    all_theta : array_like
        The trained parameters for logistic regression for each class.
        This is a matrix of shape (K x n+1) where K is number of classes
        and n is number of features without the bias.

    X : array_like
        Data points to predict their labels. This is a matrix of shape
        (m x n) where m is number of data points to predict, and n is number
        of features without the bias term. Note we add the bias term for X in
        this function.

    Returns
    -------
    p : array_like
        The predictions for each data point in X. This is a vector of shape (m, ).
    """
    p = []

    for c in range(len(all_theta)):
        theta = all_theta[c]
        p.append(logistic_reg.function(X, theta[1:], theta[0]))
    p = np.argmax(p, axis=0)
    return p
```

Figura 1.7: Código de la función *predictOneVsAll*

```python
def plot_confusion_matrix(y: np.ndarray, p: np.ndarray, filename: str) -> None:
    """Plots the confusion matrix for a given prediction.

    Args:
        y (np.ndarray): expected values
        p (np.ndarray): predicted values
        filename (str): file to store the plot
    """
    fig, ax = plt.subplots()
    ax.set_title("Confusion Matrix")
    ax.set_xlabel("Predicted")
    ax.set_xticks(np.arange(0, 10))
    ax.set_yticks(np.arange(0, 10))
    ax.set_ylabel("True")
    cm = np.zeros((10, 10))
    for i in range(len(y)):
        cm[y[i] - 1][p[i] - 1] += 1
    cax = ax.matshow(cm, cmap='Reds')
    ax.set_xticks(np.arange(0, 10))
    ax.set_yticks(np.arange(0, 10))
    ax.set_yticks(np.arange(0.5, 10.5), minor='True')
    ax.set_xticks(np.arange(0.5, 10.5), minor='True')
    plt.grid(which='minor', color='lightgrey', linestyle='-', linewidth=0.5)
    fig.colorbar(cax)
    for (i, j), z in np.ndenumerate(cm):
        if i == j:
            ax.text(j, i, '{:0.1f}'.format(z),
                    ha='center', va='center', fontsize=8, color='white')
        else:
            ax.text(j, i, '{:0.1f}'.format(z),
                    ha='center', va='center', fontsize=8)
    plt.savefig(filename, dpi=300)
    plt.clf()
```

Figura 1.8: Código de la función *plot_confusion_matrix*

```python
def print_predictions(predict: np.ndarray, num_iters: np.ndarray) -> None:
    """Plots the prediction progress of a model.

    Args:
        predict (np.ndarray): predictions of each label
        num_iters (np.ndarray): number of iterations of the training
    """
    plt.clf()
    i = 0
    plt.figure(figsize=(10, 6))
    ax = plt.subplot(111)
    for prediction in predict:
        ax.plot(range(num_iters), prediction,
                label=f'Label {i}')
        i += 1
    box = ax.get_position()
    ax.set_position([box.x0, box.y0, box.width * 0.8, box.height])

# Put a legend to the right of the current axis
    ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))
    plt.savefig(logistic_reg.plot_folder +
                "predictions_one_vs_all.png", dpi=300)
    plt.clf()
```

Figura 1.9: Código de la función *print_predictions*

```python
def save_model(all_theta: np.ndarray, filename: str) -> None:
    """Saves a trained model to a matlab binary file.

    Args:
        all_theta (np.ndarray): trained model
        filename (str): file to store the model
    """
    sio.savemat(filename, {'all_theta': all_theta})
```

Figura 1.10: Código de la función *save_model*

```python
def save_predict(predict: np.ndarray, filename: str) -> None:
    """Saves the predictions of a model in a matlab binary file.

    Args:
        predict (np.ndarray): predictions of a model
        filename (str): file to store the predictions
    """
    sio.savemat(filename, {'predict': predict})
```

Figura 1.11: Código de la función *save_predict*

```python
def save_last_predictions(predict_all: np.ndarray) -> None:
    """Saves the last predictions of a model to a csv file.

    Args:
        predict_all (np.ndarray): last predictions of each file
    """
    with open(model_folder + "one_vs_all_predict.csv", "w") as f:
        csv_writer = csv.writer(f)
        headers = [str(i) for i in range(10)]
        csv_writer.writerow(headers)
        data = []
        for i in range(10):
            data.append(str(predict_all[i][-1]))
        print(predict_all[-1].shape)
        csv_writer.writerow(data)
```

Figura 1.12: Código de la función *save_last_predictions*

2. Parte B: Regresión lógica regularizada

En el apartado B usaremos el *dataset* anterior, pero esta vez nos dan un modelo ya entrenado para hacer la red neuronal. Para ello usaremos la función *predict* (figura 2.2) para hacer la predicción basándonos en el modelo dado. Usaremos la función de matriz de antes (1.8) para hacer la matriz de confusión (figura 2.1) y la función *run_neural_network* (figura 2.3) para como driver de este apartado.

Tras ejecutar la red, conseguimos una predicción de 97.52 %.

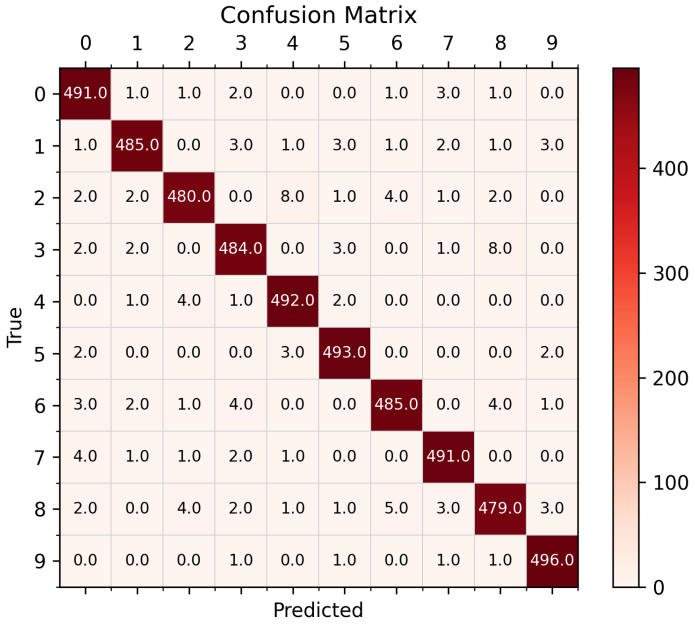

Figura 2.1: Matriz de confusión

```python
def predict(theta1: np.ndarray, theta2: np.ndarray, X: np.ndarray) -> np.ndarray:
    """
    Predict the label of an input given a trained neural network.

    Parameters
    ----------
    theta1 : array_like
        Weights for the first layer in the neural network.
        It has shape (2nd hidden layer size x input size)

    theta2: array_like
        Weights for the second layer in the neural network.
        It has shape (output layer size x 2nd hidden layer size)

    X : array_like
        The image inputs having shape (number of examples x image dimensions).

    Return
    ------
    p : array_like
        Predictions vector containing the predicted label for each example.
        It has a length equal to the number of examples.
    """

    a_2 = logistic_reg.function(X, theta1[1:], theta1[0])
    p = logistic_reg.function(a_2, theta2[1:], theta2[0])

    return np.argmax(p, axis=1)
```

Figura 2.2: Código de la función *predict*

```python
def run_neural_network() -> None:
    """Runs the neural network simulation
    """
    data = sio.loadmat('./data/ex3data1.mat', squeeze_me=True)
    X = data['X']
    y = data['y']
    m, n = X.shape
    weights = sio.loadmat('./data/ex3weights.mat', squeeze_me=True)
    theta1, theta2 = weights['Theta1'], weights['Theta2']
    p = predict(theta1.T, theta2.T, X)

    print(f'Prediction for B: {(np.sum(np.array(p) == y) / m) * 100}')
    print('Plotting confusion matrix...')
    plot_confusion_matrix(
        y, p, logistic_reg.plot_folder + "confusion_matrix_nn.png")


def main() -> None:
    run_one_vs_all()
    run_neural_network()


if __name__ == "__main__":
    main()
```

Figura 2.3: Código de la función *run_neural_network*