Machine Learning - Coursework 2

Elena Martina

March 2019

1 Implementation

In my implementation, I firstly define the dataset and visualize the data points in a scatterplot. From this plot, we can already have a visual understanding of how the data will be classified according to the Sigmoid function. After assigning the data and the labels to the X and y variables, I defined all the functions needed in order to perform logistic regression and batch gradient descent:

- "$sigmoid(z)$:" The Sigmoid activation function. It takes an input z and returns a probability between 0 and 1.
- "$predict\_hypothesis(x, theta, bias)$:" The function to predict the hypothesis h. It takes 3 parameters (x, $theta$ and a $bias$), and calculates the sum of the bias and the dot product between the input x and the weight $theta$. It then parses the result inside the Sigmoid function to return the hypothesis.
- "$cost(h_x, y)$ and $compute\_loss(h_x, y)$:" Functions to compute, respectively, the cost function for logistic regression and the loss $J(\theta)$ based on that cost. In order to test my functions and visualize the cost in respect to a range of X between 0 and 1, I also plotted the cost functions given $y = 0$ and $y = 1$.
- "$update(theta, bias, \alpha, h_x, y)$:" The Batch Gradient Descent update rule. It takes the weights $theta1$ and $theta0(bias)$ and updates them using to the given learning rate α and the difference dh between the hypothesis h_x and the true label y.
- "$monitor\_loss(history, patience = 2)$:" Function to monitor the gradient of the loss in order to early stop the algorithm when it converges. It returns False when the division between the current loss and a previous loss returns a number greater than 0.99999, which means that the loss is either increasing or not decreasing anymore by a large enough factor (in this case, a factor bigger than 0.99999). This means that the algorithm has converged. The function takes as parameters the history of the loss values, in order to be able to compare them, and a patience value ($default : 2$), which indicates how many losses we need to consider before stopping the training.

After defining those functions, I just needed to initialize the theta values and create a training loop that would only stop at convergence (according to the "$monitor\_loss()$ function). In this loop, I firstly calculate the hypothesis and the loss accordingly and store the loss in a list ($history$); then, I call the update rule function in order to update the weights $theta0$ and $theta1$.

When the training process is completed, it is possible to make predictions by calling the *predict_hypothesis*() function with the best parameters found during training. My implementation obtains a final loss of 0.49637458 and an accuracy on the training set of 80%. The best parameters found were the following:

theta1: 1.047598 — theta0: -2.867993 — Learning Rate used: 0.3

2 Questions

2.1 Question 1

"After how many iterations, and for which learning rate (α) did your algorithm converge? Plot the loss function with respect to iterations to illustrate this point."

As showed in Figure 1, my algorithm converges after about 50 iterations, with the learning rate set to 0.3.

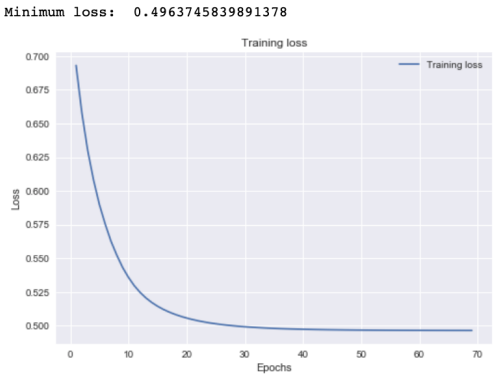

Figure 1: Plot of the Loss function with respect to iterations with learning rate=0.3.

My choice of learning rate value was made after training with different learning rates; specifically, $[0.01, 0.03, 0.05, 0.08, 0.1, 0.3]$. As it can be noticed in Figure 2 and from the values reported in Table 1, the number of iterations needed for the algorithm to converge depends on the value of the learning rate. In particular, the larger the learning rate, the faster the algorithm converges. For this reason, I chose 0.3 as the learning rate, as it converges smoothly after only about 50 epochs, without excessively decreasing the final loss value and performance of the algorithm.

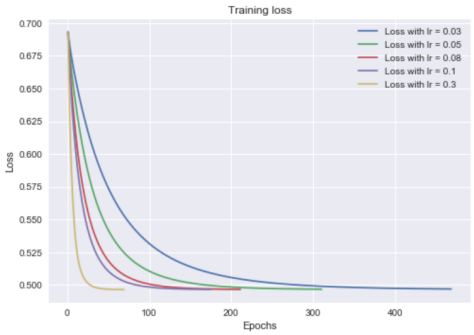

Figure 2: Plot showing how the algorithm converges differently according to different learning rates (α), specifically for $\alpha = [0.03, 0.05, 0.08, 0.1, 0.3]$

Learning Rate	Convergence	Final Loss
0.01	1100	0.498
0.03	467	0.497
0.05	309	0.497
0.08	210	0.497
0.1	174	0.496
0.3	68	0.496

Table 1: Table showing the values from the example in Figure 2. The "convergence" column indicates how many epochs (iterations) it takes for the algorithm to converge.

2.2 Question 2

What happens if α is too large? How does this affect the loss function? Plot the loss function with respect to iterations to illustrate this point.

In general, large learning rates allow the algorithm to converge faster, even at the cost of achieving sub-optimal final results. When the learning rate becomes too large, however, the model might overshoot and "jump over" the minima without ever reaching it: in this case, its loss function oscillates over the training epochs, phenomenon that can be observed in Figure 3 with learning rates >0.5. This oscillating phenomenon is caused by the weight updates being too large.

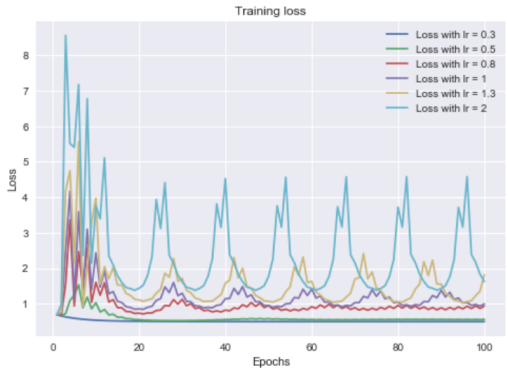

Figure 3: Plot showing how the Loss function is affected by different learning rates, specifically by $\alpha = [0.03, 0.05, 0.08, 0.1, 0.3]$

2.3 Question 3

Assume that you are applying logistic regression to the iris (flower) dataset, as in the previous assignment. Answer the following questions:

1. *How would your hypothesis function change in this case and why?*
2. *How would you utilize your implementation of logistic regression in order to perform (multi-class) classification on the iris dataset?*

Include some pseudocode while discussing your approach.

2.3.1 Question 3.1

When using logistic regression in binary classification problems, the hypothesis function is defined as:

$$h_\theta(x) = P(y = 1|x; \theta)$$

which corresponds to the probability that $y = 1$, given X, parametherized by θ.

Consequently, as the sum of probabilities over all possible events is 1, the probability that $y = 0$ will be:

$$P(y = 0|x; \theta) = 1 - P(y = 1|x; \theta)$$

The iris dataset, however, represents a multiclass classification problem, as it requires classifying the instances into one of three classes. In this case, it would be unreasonable to use the hypothesis function described above, which considers only two possible outcomes.
An approach that could be used when applying logistic regression to multiclass classification is called "One vs all". In this approach, we train a logistic regression classifier for each class i in order to predict the probability $h_\theta^{(i)}(x)$ that $y = i$. The hypothesis function can therefore be re-written as:

$$h_\theta^{(i)}(x) = P(y = i|x; \theta) \quad for \;\; i = 1, 2, ..., n\_classes$$

2.3.2 Question 3.2

In order to implement "One vs all" logistic regression, as mentioned above, I would train a logistic regression classifier for each class i. Firstly, I would therefore loop through the classes, and create a temporary $y\_$ for the current class, in which all the labels corresponding to the current class will be set to 1, while all the other classes would become 0. I would then normally run the training with this new labels array, as for a simple binary classification problem of 1 class against all the others. The code would look something similar to the one in Figure 4.

In [ ]:

```
lr = 0.001
classes = np.unique(y)  # array of the unique classes in the labels
history = []  # list to store the loss values
classifiers = []  # list to store the parameters of the classifiers for each class.

for i in range(len(classes)):

    # y_ is a temporary labels array having 1
    # if the label corresponds to the current class, 0 otherwise
    y_ = (y == i).astype(int)

    history.append([])          # add an array to store the loss of the current class
    classifiers.append([])      # add an array to store the best parameters of the current class

    # Parameters
    theta = np.zeros((X.shape[1]))
    b = np.zeros((1))

    while(monitor_loss(history[i])):  # while the algorithm has not converged . . .
        hx = predict_hypothesis(X, theta, b)     # get the hypotesis
        loss = compute_loss(hx, y_)              # calculate the loss
        history[i].append(loss)                  # save current loss
        theta, b = update(theta, b, lr, hx, y_)  # update the weight and the bias

    # store the best parameters at the end of training
    classifiers[i].append(theta)
    classifiers[i].append(b)
```

Figure 4: Training my model on a Multiclass Classification problem.

After the training, I would calculate the hypothesis with the best parameters found for that particular class; then, in order to make the prediction, pick the class i that maximises the probability (has the highest h):

$$max_i(h_\theta^{(i)}(x))$$