Logistic regression

Grzegorz Rybak

1. Algorithm definition.

Logistic regression is a classification model that aims to determine a discrete (non-continuous) value of the output (i.e. “True” or “False”, 1 or 0, etc.) given a set of continuous values as the input. Although the model’s name is slightly confusing at the beginning, given the context, it fully describes the mechanisms behind it.

1.1 Logistic regression explanation.

Let us firstly explore the word “logistic”. It comes from the fact the model uses the “logistic function” (also called “sigmoid function”) [figure 1]. The return value of the logistic function is always in range [0,1] regardless of the input (see [figure 2]) and this allows for later discrimination of whether the predicted outcome will fall into “0” or “1” category. The discrimination process is a simple conditional choice of 0, given the result smaller than 0.5, or 1, given the result greater or equal to 0.5.

$$g(z) = \frac{1}{1+e^{-z}}$$

Figure 1: Logistic function formula.

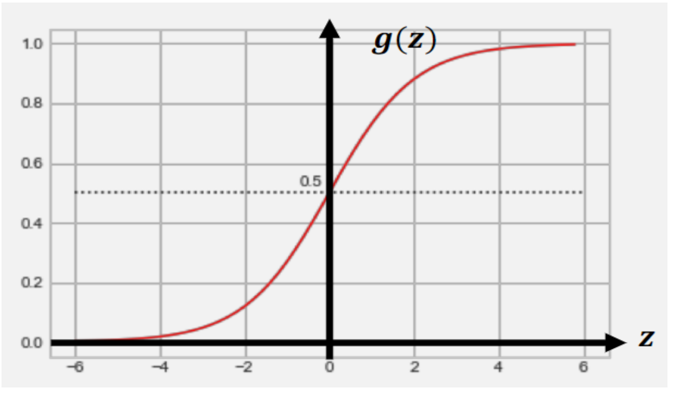


*Figure 2: Logistic function values plot. Notice the values **asymptote** to 0 or 1 on both ends.*

The word “regression” comes from the fact the same formula used in the linear regression ($h_\theta(x) = \theta^T x$) is used in the logistic regression. The difference is that here, it is provided as the “z” argument in logistic equation ([figure 1]). This creates the complete formula of the equation as shown in [figure 3].

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

Figure 3: Complete formula for the logistic regression model (comprising of both the linear regression and the logistic function parts).

1.2 Gradient decent.

The above allows the logistic regression for predicting the label given a datapoint’s input value. However, to correctly determine the correlation between the input and the output data, appropriate values of the Thetas (θ_j) must be established. In other words, the algorithm needs to repeatedly predict the sample data and update the predicted values according to their true values. This process is called “gradient decent” and it involves calculating a “loss value” that determines how far the prediction was from the actual value and recurrently updating the Thetas to decrease the loss value and, as a result, improve the accuracy of the predictions.

This is the high-level overview of the concepts behind the algorithm implemented in the attached Jupyter notebook. For the brevity, the complete description (or the formula) of the function calculating the loss and the process of updating the values after each dataset run (also called “epoch”) are not included but can be further inspected in the supplied code.

2 Dataset analysis.

To test the performance of the implemented logistic regression, a dataset from the Wikipedia called “Probability of passing an exam versus hours of study” will be used. It depicts a correlation between time of study for an exam and whether was it passed or failed ([figure 4]).

Hours	0.50	0.75	1.00	1.25	1.50	1.75	1.75	2.00	2.25	2.50	2.75	3.00	3.25	3.50	4.00	4.25	4.50	4.75	5.00	5.50
Pass	0	0	0	0	0	0	1	0	1	0	1	0	1	0	1	1	1	1	1	1

Figure 4: All 20 datapoints in the Wikipedia’s “Probability of passing an exam versus hours of study” dataset.

3 Logistic regression implementation.

The algorithm is implemented as a Python class. It contains 2 “main” methods, 3 “print” methods 4 “utility” methods.

- The main methods are called “fit” and “predict”. “Fit” is the method containing all the business logic of the logistic regression: initialising thetas, predicting, obtaining loss, updating values and repeating this until the loss’ convergence. The “predict” is method to be used after running the “fit” which allows for a prediction of the exam result (“pass”, “fail”) given a float number of hours studied.
- The Print methods are called “print_saved_loss”, “print_train_data_predictions” and “print_trained_parameter_values”. These can be used for user’s convenience to check the current values of these particular variables in the class’s scope.
- The “utility” methods are called: “sigmoid”, “loss_function”, “has_converged” and “decision_boundary”. The “sigmoid” returns the value of the logistic function, the loss calculates the correct loss given the predicted and the true values and the other two simply return a Boolean value if loss has converged and 1/0 value depending on the prediction value. Notice all “utility” methods having a “@staticmethod” decorator. This is a design choice signalling those methods are rather independent of the core state of the class and therefore can be used without instantiating the class object if the user desires to use their functionality outside of the logistic regression.

4 Implementation results.

The loss value of the implementation has converged on the level of ~0.496 after 50010 epochs. The learning rate (α) was set on α = 0.001 whereas the epsilon (ε), used to determine whether the compared loss values should be considered as “the same”, was set to ε = 0.005. The following plot ([figure 5]) depicts the results.

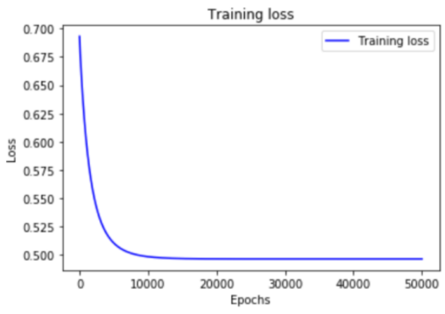


Figure 5: Loss function values of the model.

5 Exploring the case of a too big learning rate.

Running the model for only 110 epochs (running it for too long would clutter the plot values making it unreadable) with a big alpha α = 1 shows that the algorithm can never achieve a reasonably low loss function, greatly affecting any accuracy capabilities of this machine learning function ([figure 6]). The loss starts at the value 0.5 as all the Theta values (the weights and the bias) are initially 0, thus performing the prediction via the sigmoid function would equal to $h_\theta(x) = g(\theta^T x) = 1/1+e^{-0} = \mathbf{0.5}$. Then, due to a big learning rate we can observe the “gradient explosion” effect that results in the updated value being “overshoot” compared with the original loss value. This is later constrained but due to the learning rate performing “too big steps”, the loss value can never converge and keeps the loop indefinitely.

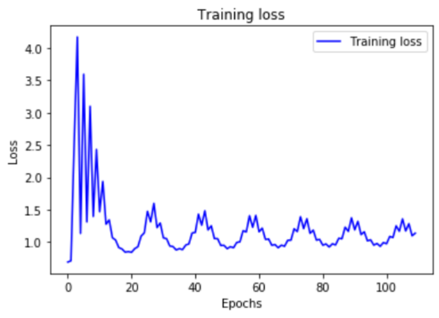


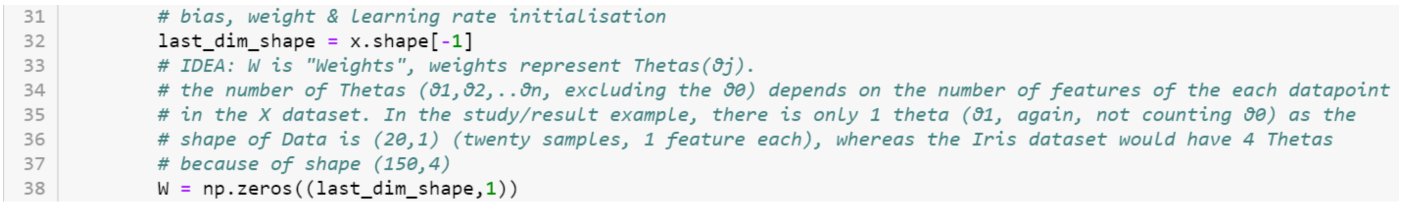
Figure 6: Loss function values when α = 1.

6 Applying the implementation to Iris dataset.

6.1 Changes in the hypothesis function.

If the current implementation was to be used to classify the Iris classes from the Iris dataset, the only difference would be in the amount of Thetas (θ) from 2 (1 weight and 1 bias) to 5 (4 weights and 1 bias). This is because the Iris dataset has the shape of (150,4), i.e. 150 samples, having 4 features each, instead of the shape (20,1) of the Wikipedia dataset.

This difference is accounted for in the implementation, as the shape of the weight array depends on the number of features of the input x (which can be extracted by calling the last dimension of x: “x.shape[-1]”, see [figure 7])

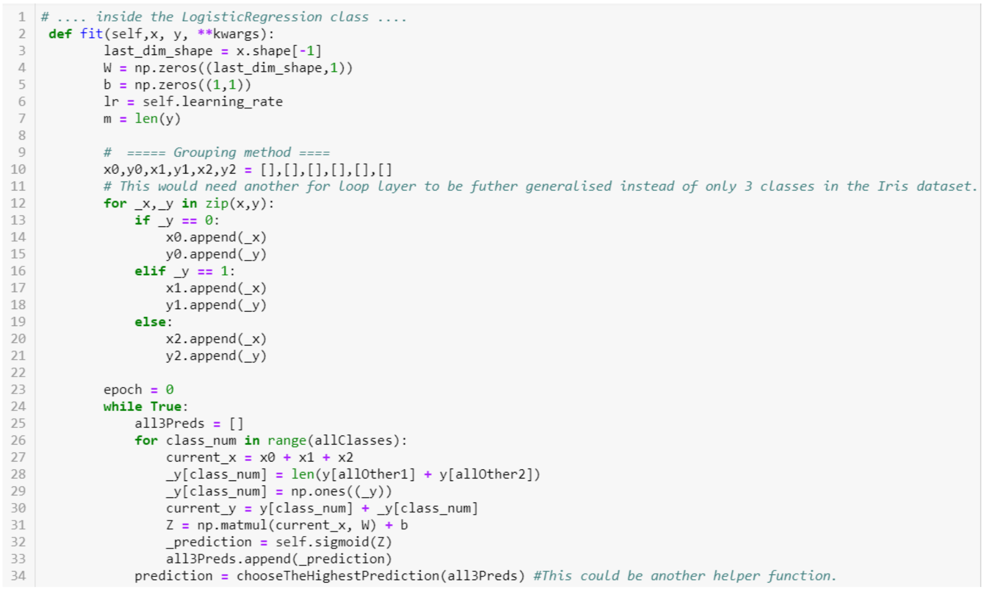


```
        # bias, weight & learning rate initialisation
        last_dim_shape = x.shape[-1]
        # IDEA: W is "Weights", weights represent Thetas(ϑj).
        # the number of Thetas (ϑ1,ϑ2,..ϑn, excluding the ϑ0) depends on the number of features of the each datapoint
        # in the X dataset. In the study/result example, there is only 1 theta (ϑ1, again, not counting ϑ0) as the
        # shape of Data is (20,1) (twenty samples, 1 feature each), whereas the Iris dataset would have 4 Thetas
        # because of shape (150,4)
        W = np.zeros((last_dim_shape,1))
```

Figure 7: Using the implementation with the Iris dataset will result in the "w" array having different shape.

6.2 Performing multi-class classification.

In order to perform a correct classification task on a dataset that has more than 2 classes, like the Iris dataset (3 classes of iris flowers as the possible labels), the “one-vs-all” classification strategy needs to be employed. This, in essence, relies on feeding the datasets into the regression model in a “one-other” arrangement (where “one” is one class and “other” is all the other possible classes), performing the prediction, doing the same prediction manner for all other classes and selecting the class that has the biggest prediction value. Therefore, this strategy requires implementing a new method repeatedly grouping the datapoints into “one-other” groups and an inner loop inside the “fit” method that performs the prediction for all the arrangements and selects the one with the biggest score. The pseudocode for this can be seen below.



```
# .... inside the LogisticRegression class ....
 def fit(self,x, y, **kwargs):
        last_dim_shape = x.shape[-1]
        W = np.zeros((last_dim_shape,1))
        b = np.zeros((1,1))
        lr = self.learning_rate
        m = len(y)

        #  ===== Grouping method ====
        x0,y0,x1,y1,x2,y2 = [],[],[],[],[],[]
        # This would need another for loop layer to be futher generalised instead of only 3 classes in the Iris dataset.
        for _x,_y in zip(x,y):
            if _y == 0:
                x0.append(_x)
                y0.append(_y)
            elif _y == 1:
                x1.append(_x)
                y1.append(_y)
            else:
                x2.append(_x)
                y2.append(_y)

        epoch = 0
        while True:
            all3Preds = []
            for class_num in range(allClasses):
                current_x = x0 + x1 + x2
                _y[class_num] = len(y[allOther1] + y[allOther2])
                _y[class_num] = np.ones((_y))
                current_y = y[class_num] + _y[class_num]
                Z = np.matmul(current_x, W) + b
                _prediction = self.sigmoid(Z)
                all3Preds.append(_prediction)
            prediction = chooseTheHighestPrediction(all3Preds) #This could be another helper function.
```

Figure 8: Pseudocode for extending the implementation with the multi-class "one-vs-all" strategy.

7 Bibliography.

https://blog.goodaudience.com/logistic-regression-from-scratch-in-numpy-5841c09e425f
Andrew Ng’s Machine Learning course, section 6: https://www.youtube.com/watch?v=-la3q9d7AKQ
Lecture 6: “Logistic regression” slides from “IS53051A: Machine Learning (2018-19)”