Implementing Logistic Regression

Ege Kaan Özalp

Sabancı University

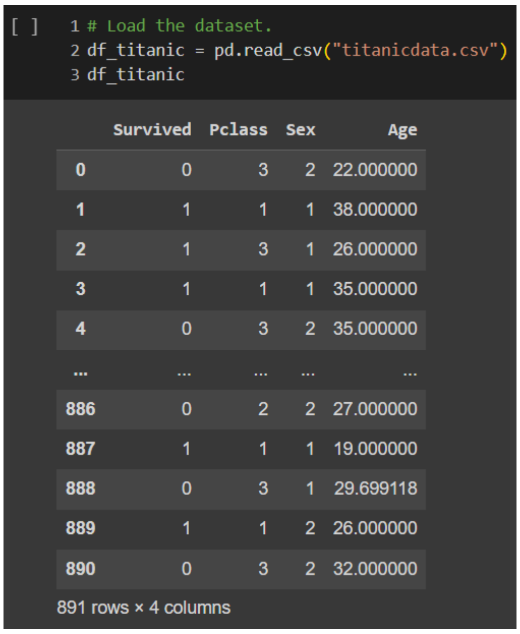
CS 412: Machine Learning

Professor Öznur Taştan

April 29, 2024

(This paper includes 7 parts. In part 7, the results are interpreted.)

1. Load the dataset and preprocess the data:
    a. Load the dataset:

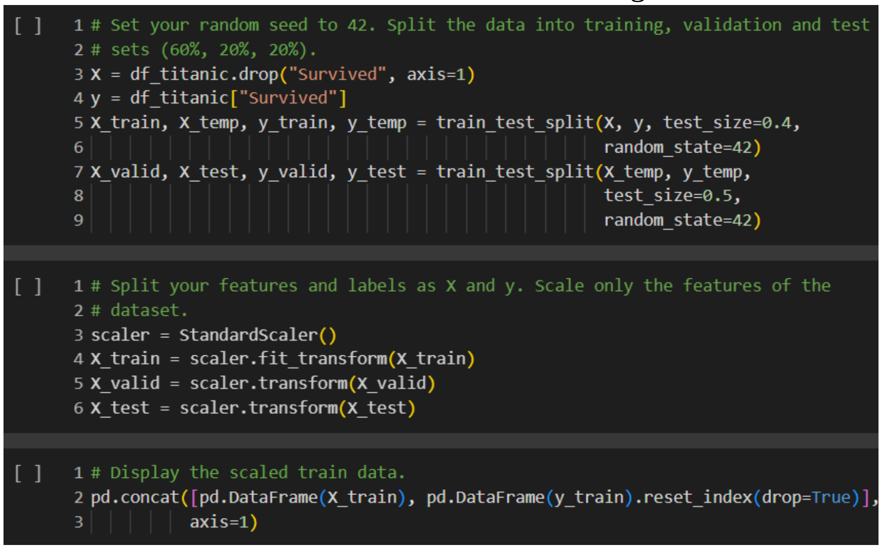


```
[ ] # Load the dataset.
    df_titanic = pd.read_csv("titanicdata.csv")
    df_titanic
```

```
     Survived  Pclass  Sex        Age
0           0       3    2  22.000000
1           1       1    1  38.000000
2           1       3    1  26.000000
3           1       1    1  35.000000
4           0       3    2  35.000000
...       ...     ...  ...        ...
886         0       2    2  27.000000
887         1       1    1  19.000000
888         0       3    1  29.699118
889         1       1    2  26.000000
890         0       3    2  32.000000
891 rows × 4 columns
```

Figure 1.1.: Load the dataset.

    b. Set the random seed to 42. Split the data into training, validation, and test sets (60%, 20%, 20%). Scale the features of the training set with a scaler first, then scale the features of the validation and the test set using the fitted scaler:



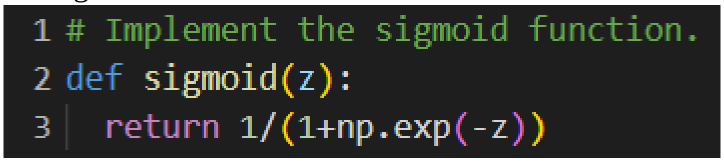
```
[ ] # Set your random seed to 42. Split the data into training, validation and test
    # sets (60%, 20%, 20%).
    X = df_titanic.drop("Survived", axis=1)
    y = df_titanic["Survived"]
    X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.4,
                                                        random_state=42)
    X_valid, X_test, y_valid, y_test = train_test_split(X_temp, y_temp,
                                                        test_size=0.5,
                                                        random_state=42)
```

```
[ ] # Split your features and labels as X and y. Scale only the features of the
    # dataset.
    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)
    X_valid = scaler.transform(X_valid)
    X_test = scaler.transform(X_test)
```

```
[ ] # Display the scaled train data.
    pd.concat([pd.DataFrame(X_train), pd.DataFrame(y_train).reset_index(drop=True)],
              axis=1)
```

Figure 1.2.: Split and scale the data.

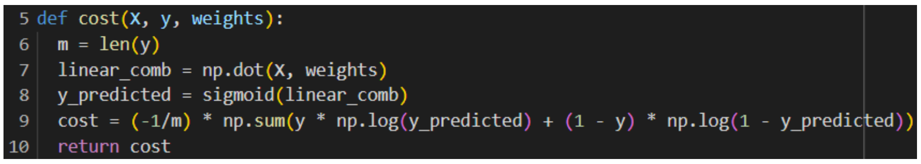
2. Implement the Logistic Regression Model
    a. Implement the Sigmoid function:



```
# Implement the sigmoid function.
def sigmoid(z):
  return 1/(1+np.exp(-z))
```

Figure 2.1.: Sigmoid function implementation.

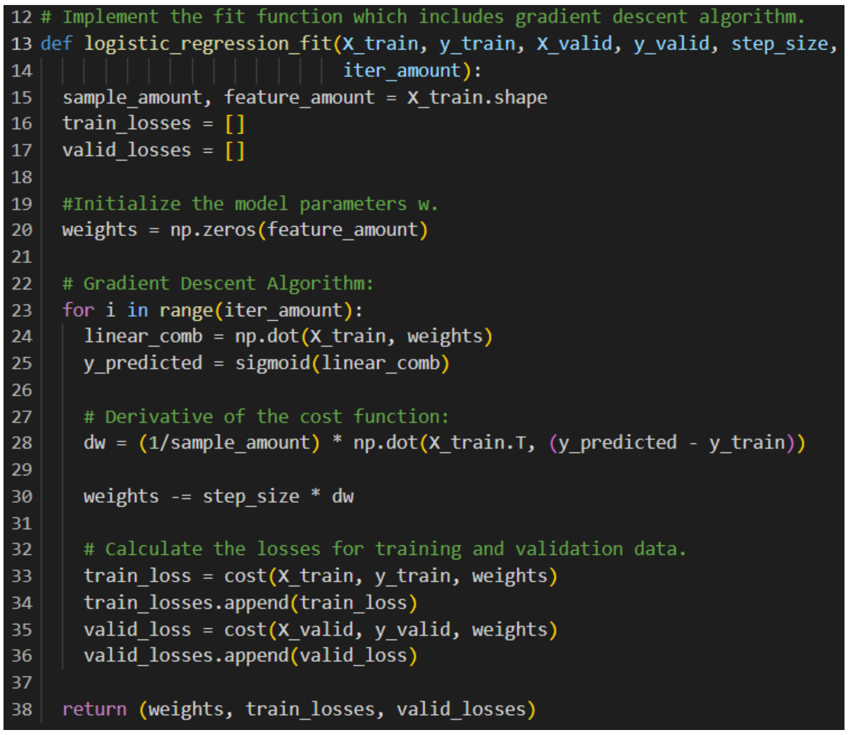
b. Implement the Cost function:



```python
def cost(X, y, weights):
  m = len(y)
  linear_comb = np.dot(X, weights)
  y_predicted = sigmoid(linear_comb)
  cost = (-1/m) * np.sum(y * np.log(y_predicted) + (1 - y) * np.log(1 - y_predicted))
  return cost
```

Figure 2.2.: Cost function implementation.

c. Implement the Logistic Regression Fit function which includes the algorithm of gradient descent:

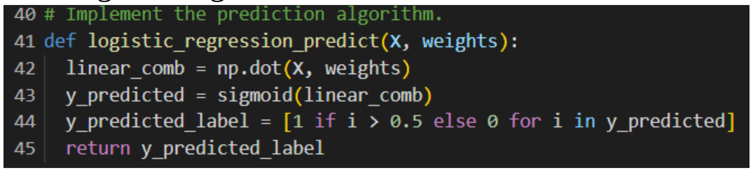


```python
# Implement the fit function which includes gradient descent algorithm.
def logistic_regression_fit(X_train, y_train, X_valid, y_valid, step_size,
                            iter_amount):
  sample_amount, feature_amount = X_train.shape
  train_losses = []
  valid_losses = []

  #Initialize the model parameters w.
  weights = np.zeros(feature_amount)

  # Gradient Descent Algorithm:
  for i in range(iter_amount):
    linear_comb = np.dot(X_train, weights)
    y_predicted = sigmoid(linear_comb)

    # Derivative of the cost function:
    dw = (1/sample_amount) * np.dot(X_train.T, (y_predicted - y_train))

    weights -= step_size * dw

    # Calculate the losses for training and validation data.
    train_loss = cost(X_train, y_train, weights)
    train_losses.append(train_loss)
    valid_loss = cost(X_valid, y_valid, weights)
    valid_losses.append(valid_loss)

  return (weights, train_losses, valid_losses)
```

Figure 2.3.: Logistic Regression Fit function implementation.

In this function, first the sample amount and feature amount values are obtained. Then the arrays for train losses and validation losses, and the model parameters “w” are initialized. After these initializations, the Gradient Descent algorithm takes place. For the specified times of iteration, the program creates the linear combination of the features using the current values of the weights (np.dot(X_train, weights)). Then it calculates the predicted probability values of labels. Using these predicted values, the derivative of the cost function (dCost/dw) and the step size, it finds the gradient (dw) and updates the weights. In each step, it also calculates the training and validation losses and appends them to the corresponding arrays. Once the iterations are over, it returns the final updated weights with the arrays of the train losses and the validation losses.

d. Implement the Logistic Regression Predict function:



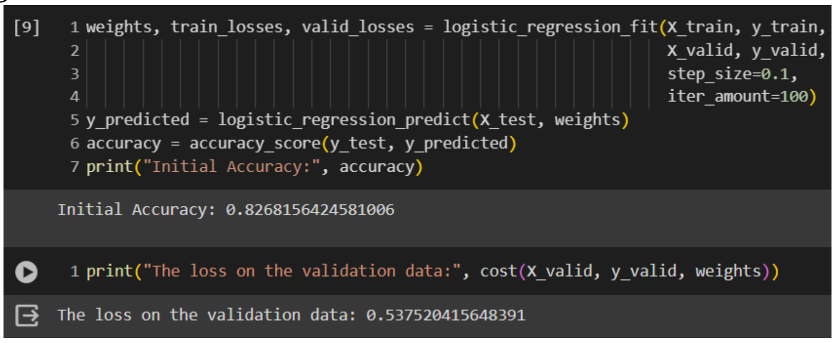
```
# Implement the prediction algorithm.
def logistic_regression_predict(X, weights):
    linear_comb = np.dot(X, weights)
    y_predicted = sigmoid(linear_comb)
    y_predicted_label = [1 if i > 0.5 else 0 for i in y_predicted]
    return y_predicted_label
```

Figure 2.4.: Logistic Regression Prediction function implementation.

In this function, the linear combination of the features is created using the values of the weights. Then, to find the predicted probability values of the data points, the program applies the Sigmoid function to the newly created linear combination. After this step, to decide on the labels of the given X_test set, the program checks the values of y_predicted: If the value is higher than 0.5 then it is considered as survived (y=1), otherwise it is considered as not survived (y=0).

3. Train the Model and Plot the Losses

a. Set the step size to 0.1 and iteration amount to 100. Train the model using the training data. Calculate the loss on the validation data:



```
[9]
weights, train_losses, valid_losses = logistic_regression_fit(X_train, y_train,
                                                              X_valid, y_valid,
                                                              step_size=0.1,
                                                              iter_amount=100)
y_predicted = logistic_regression_predict(X_test, weights)
accuracy = accuracy_score(y_test, y_predicted)
print("Initial Accuracy:", accuracy)

Initial Accuracy: 0.8268156424581006
```

```
print("The loss on the validation data:", cost(X_valid, y_valid, weights))

The loss on the validation data: 0.537520415648391
```

Figure 3.1.: Initial accuracy and the loss on the validation data.

b. Plot both the training and validation losses across 100 iterations:

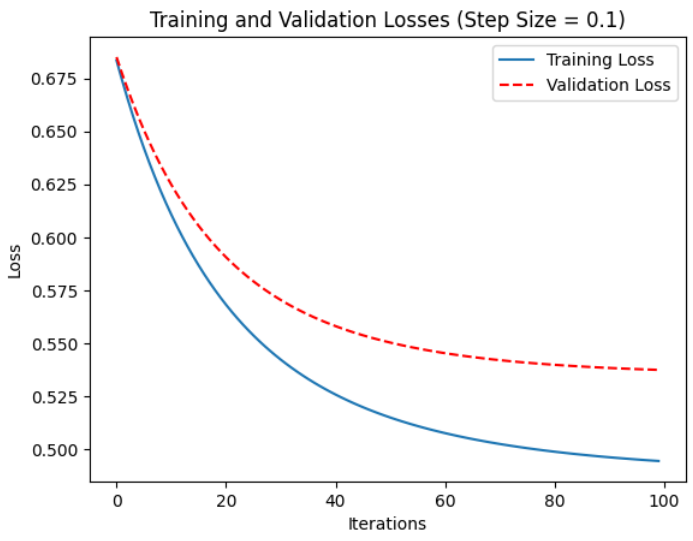
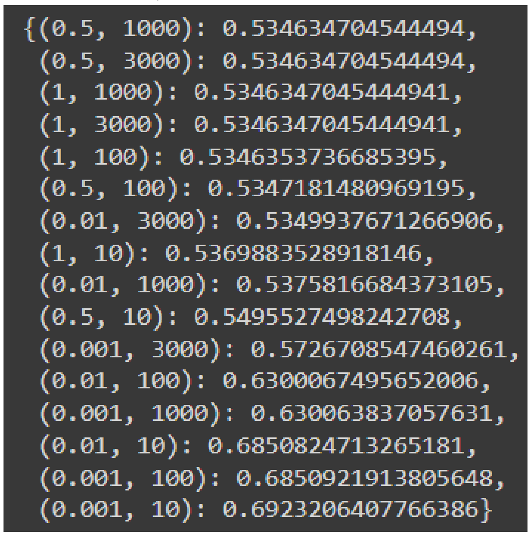


Figure 3.2.: Training and Validation Losses.

4. Vary the Step Size and Number of Iterations and Pick the Best Combination
   a. To vary the step size and iteration hyperparameters, the values that are used are:
      Step sizes = [0.001, 0.01, 0.5, 1]
      iterations = [10, 100, 1000, 3000]

   b. After trying every combination, the observed values of validation losses are:



```
{(0.5, 1000): 0.534634704544494,
 (0.5, 3000): 0.534634704544494,
 (1, 1000): 0.5346347045444941,
 (1, 3000): 0.5346347045444941,
 (1, 100): 0.5346353736685395,
 (0.5, 100): 0.5347181480969195,
 (0.01, 3000): 0.5349937671266906,
 (1, 10): 0.5369883528918146,
 (0.01, 1000): 0.5375816684373105,
 (0.5, 10): 0.5495527498242708,
 (0.001, 3000): 0.5726708547460261,
 (0.01, 100): 0.6300067495652006,
 (0.001, 1000): 0.630063837057631,
 (0.01, 10): 0.6850824713265181,
 (0.001, 100): 0.6850921913805648,
 (0.001, 10): 0.6923206407766386}
```

Figure 4.1.: Validation Losses based on different hyperparameters (in increasing order) (key = (step size, iteration amount)).

   c. Using the best hyperparameter values (in our case it is step_size = 0.5 and iter_amount = 1000) plot the training and validation losses:

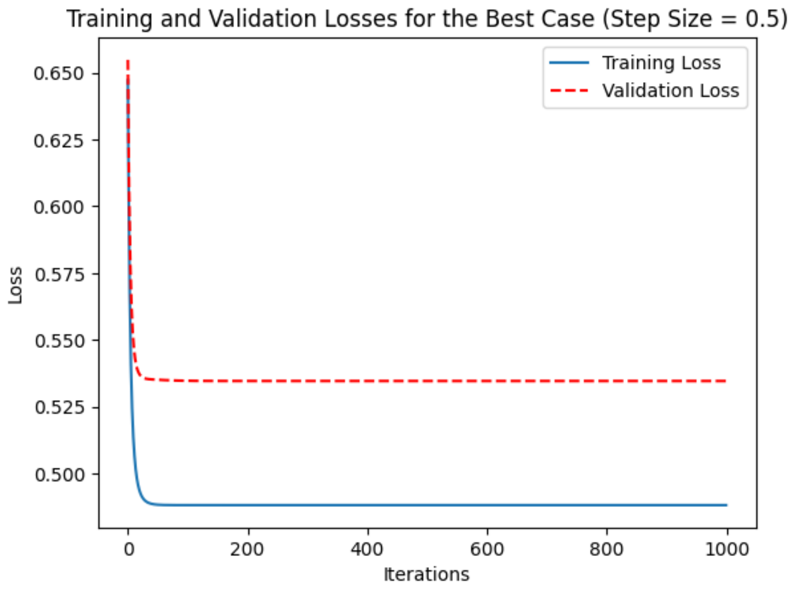
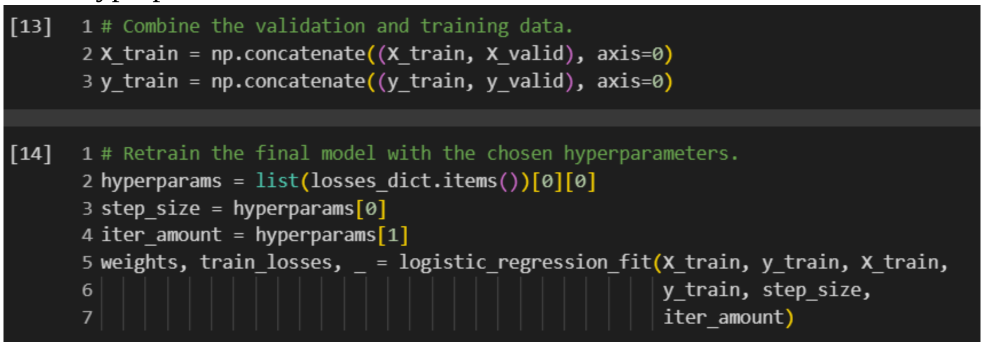


Figure 4.2.: Training and Validation Losses for the Best Case.

5. Retrain the Final Model
   a. Combine the validation and training data and retrain the final model with the chosen hyperparameters:

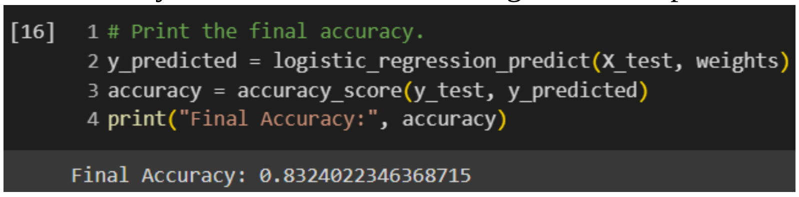


```
[13]  # Combine the validation and training data.
      X_train = np.concatenate((X_train, X_valid), axis=0)
      y_train = np.concatenate((y_train, y_valid), axis=0)
```

```
[14]  # Retrain the final model with the chosen hyperparameters.
      hyperparams = list(losses_dict.items())[0][0]
      step_size = hyperparams[0]
      iter_amount = hyperparams[1]
      weights, train_losses, _ = logistic_regression_fit(X_train, y_train, X_train,
                                                        y_train, step_size,
                                                        iter_amount)
```

Figure 5.1.: Validation Losses based on different hyperparameters (in increasing order) (key = (step size, iteration amount)).

6. Evaluate the Accuracy of the Model
   a. Evaluate the accuracy of the model on the testing data and report the results:



```
[16]  # Print the final accuracy.
      y_predicted = logistic_regression_predict(X_test, weights)
      accuracy = accuracy_score(y_test, y_predicted)
      print("Final Accuracy:", accuracy)

      Final Accuracy: 0.8324022346368715
```

Figure 6.1.: Final accuracy.

   b. Plot the test losses for the final hyperparameter values:

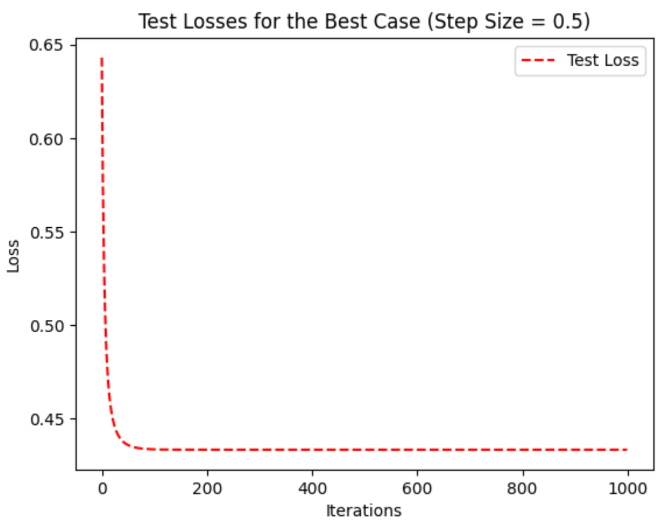
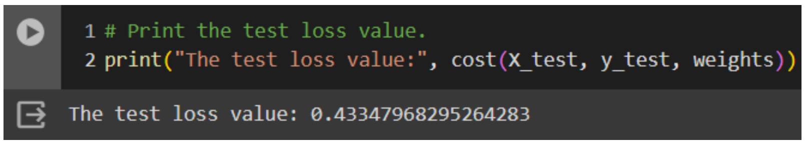


Figure 6.2.: Test losses for the final case.

   c. Print the final test loss value:



```
# Print the test loss value.
print("The test loss value:", cost(X_test, y_test, weights))

The test loss value: 0.43347968295264283
```

Figure 6.3.: Test loss value for the final case.

7. Interpretation of the Outcomes
    a. As the validation loss value decreases, the accuracy tends to increase: In Figure 4.1., it is shown that the hyperparameter value (step_size = 0.5, iteration_amount = 1000) has less validation loss value than the (step_size = 0.1, iteration_amount = 100) which can be read from Figure 3.1. As a result of this, once we selected (step_size = 0.5, iteration_amount = 1000) as our hyperparameter values, the accuracy increased from 0.826 to 0.832.
    b. The acceleration of the decline of validation and training losses tend to decrease after a certain point given a certain step size: In Figure 4.2., it is shown that after around 40-50 iterations, values of validation and training losses tend to stay stable.
    c. Final accuracy is approximately 0.8324.
    d. Final validation loss is approximately 0.5346.
    e. Final test loss is approximately 0.4334.