Machine Learning Assignment 2

Name	Student ID	Class
Saikrishna Javvadi	20236648	1CSD

Algorithm Description

Selection: Logistic Regression

The logistic regression model is a classification algorithm that predicts the outcome of one dichotomous variable in its simplest model. This assignment considers implementing and evaluating a multi-class version of this algorithm.[1] The beer.txt dataset considers three classes of beer style (stout, lager and ale). The multiple logistic regression model considers the prediction of the style which has three classes that could apply. The data is fitted to the sigmoid function and the probability of each style of beer is calculated based of these unique attributes extending the logistic regression by using it in one vs all method. Most of the implementation details and equations for this algorithm are referred to from Machine Learning course of Professor Andrew NG.[5]

Implementation: Logistic Regression Algorithm

1. Import data (beer.xlsx) using Pandas. The file headings have already been added into the layout and it is assumed that any files input into the program would be in this layout. Each column has only one attribute and the first row contains column headings. The last column consists of the attribute being predicted (style of beer).
2. Declare the column headings and the beer_id column is removed as has no statistical relevance.
3. The X-data is (a numpy array of independent variables) defined as the all rows in the columns of the beer.txt except the last column (i.e. the style) while the Y_data is the dependent variable(style) having true labels of data.
4. The X-data is then standardised by assigning z-score in the feature scaling function. This quantifies the distance of each instance from the mean in terms of standard deviation.
5. The algorithm is iterated over 10 randomly shuffled splits of training and test data. The split is 2/3 training data and 1/3 test data. Individual accuracy each time the algorithm is run is captured across 10 iterations and the average across all these iterations is considered as the final accuracy of the model.
6. Classes are defined as array of each unique instance in the style column.
7. The logistic regression algorithm is then trained for each instance in the classes array, where the instance in the iteration is encoded as 1 and everything else as 0. Multi-class logistic regression involves repeating the binary algorithm for all instances where we assign the prediction to the given example as the highest probability among all the different classes. [1]
8. The hypothesis representation fits the sigmoid function to the data given, shown in equation 1. [1] This creates a function which approaches limits at the asymptotes y=1 and y=0. The classification threshold is at y=0.5, so when the classification output (Y) is equal to or over 0.5 the hypothesis is accepted. If the classification output is less than 0.5 the hypothesis is rejected for a binary logistic regression algorithm. Since implementation is a one vs all implementation, the class with highest probability value found after training the model is chosen.

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

Where x is the scaled x data and θ^T is the transposed array of algorithm parameters. This defines the classes of beer in terms of the eight independent variables and is optimised later in the code. This equation is simplified to one over one plus the natural log to the negative power of the transposed theta times x, in the code implementing the algorithm.

9. Regularization reduces the effect of overfitting the data in classification by constraining the magnitude of parameters instead of the number of parameters, so the complexity of the problem remains. This factor calculated in equation 2 is added to the end of the cost function.[2, 3]

$$Reg = \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j{}^2 \qquad \textit{Equation 2}$$

10. The cost function defines the parameters of theta dependent on whether Y is true of false in the training set. Where Y is equal to 1 the equation finds the average of the negative log of the sigmoid function. Where Y is equal to 0 the equation finds the average of the negative log of the 1 minus the sigmoid function. Regularization is added to the end of the cost function to prevent overfitting.[2]

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} -Y\log\left(h_\theta(x^{(i)})\right) - (1-Y)\log(1-h_\theta(x^{(i)})) + Reg \qquad \textit{Equation 3}$$

11. Gradient Descent optimises the algorithm used while training. It changes the theta (θ^T) parameter iteratively to minimise the derivative of the cost function which results in finding the dot product of the transpose of the X-data and the prediction error divided by the total number of instances.[2]
12. Optimize by multiplying the learning rate by the gradient descent. The learning rate and number of iterations of the optimize function are both hyperparameters which control the learning process. Equation 4 shows the final theta equation including the equation for gradient descent. This equation is repeated to optimise the theta value by minimising the derivative of the cost function.

$$\theta_j = \theta_j - \alpha(\frac{1}{m}\sum_{i=1}^{m} x_j{}^{(i)}(h_\theta\left(x^{(i)}\right) - y^{(i)}) + \frac{\lambda}{m}\theta_j) \qquad \textit{Equation 4}$$

Where α is the learning rate, m is the total number of instances and $[j = 1,2,3 \ldots . n]$.

13. Predictions are made for the style of the beer from the X data based off the sigmoid function.
14. The accuracy of the algorithm is assessed by taking the number of correct predictions divided by the total number of instances predicted. The predictions and true labels are then written to a file.

Design Decisions

The X-data was standardised due to the differing range of values for different attributes. For example, the nitrogen content of a beer usually has a numerical content less that one while the degree of fermentation can have a numerical value in in the sixties.

The cost function is regularised to prevent the problem of overfitting the training data. Regularisation essentially keeps all the attributes in the dataset , but reduces their magnitude/values which makes the hypothesis less prone to overfitting.[3]

Gradient descent requires an appropriate learning rate to ensure the function works. The learning rate (α) determines how rapidly the parameters are updated and must be well-tuned to the dataset. A learning rate that is too small will require to many iterations to converge on the best value and a rate to large overshoots the optimal value.

The scikit-learn logistic regression algorithm is compared to the algorithm to quantify its accuracy. This package can be used quickly and is easy to implement in python. It also has a high accuracy rating already so there is a high standard to compare to.

Testing

Evaluation: Logistic Regression Algorithm

The implemented logistic regression algorithm is assessed by comparing it to a logistic regression algorithm provided by Scikit-Learn. K fold cross validation is completed on both algorithms where both algorithms are completed using the same training and testing datasets 10 times. From these ten iterations the accuracy of each iteration is calculated and plotted in figure 1. This compares the performance of the

algorithms when they are executed on identical data sets. It is standard practice to complete this type of cross validation ten times and is called tenfold cross validation. [4]

The confusion matrix relays a visual representation of the accuracy of an algorithm. It displays the row and column for each class. Each row element in the actual class and each column element displays the predicted class. The ideal confusion matrix will have a diagonal of large numbers while all other elements will equal zero.[4]

Finally, the last method of testing used is a graph depicting the learning curve. The learning curve plots the number of iterations against the cost value. This is used to help determine an appropriate learning rate. It is also used to ensure the accuracy of the previous tests and avoid an estimation error.[4]. The learning curve is plotted for a number of different learning rates to ensure the accuracy of the learning rate chosen.

Results

The accuracies for ten iterations of each implementation are plotted in figure 1. The average test accuracy for the implemented logistic regression code is 95.96% and for the scikit-learn algorithm is 96.04%. While the algorithms only have a differing average accuracy of 0.08% the implemented algorithm has a lower range of values than the scikit-learn algorithm which could suggest more consistent results.

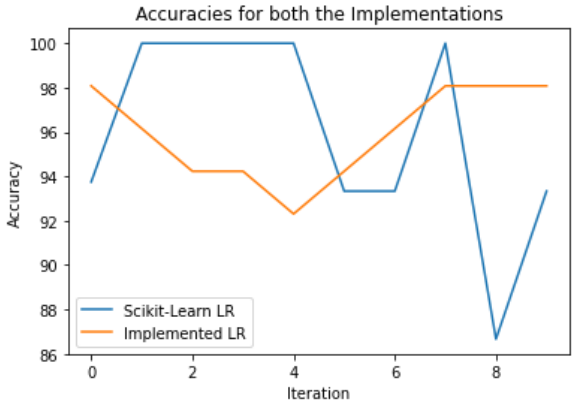
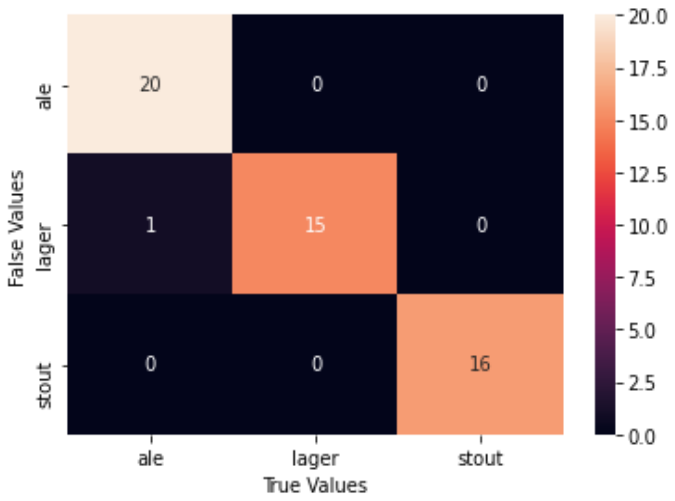


Figure 1 Graph comparing the accuracy of the implemented logistic regression model created from scratch and the Scikit-Learn logistic algorithm for ten iterations. The implemented logistic regression algorithm has a more consistent level of accuracy while the Scikit-Learn algorithm has a higher range of accuracy.

Figure 2 Confusion matrix for the implemented logistic regression algorithm. The rows show the actual values for each of the classes while the columns show the predicted values for each of the classes.



The confusion matrix of the implemented logistic regression algorithm is shown is figure 2. This matrix shows that our algorithm incorrectly labelled an ale as a lager. The diagonal of the matrix shows the correct prediction that were made while the other elements show any false predictions.

Figure 3 shows a plot of the of cost against the number of iterations. The different curves represent models trained with different learning rates (α). In order for our Gradient Descent to work we must choose the learning rate wisely. The learning rate alpha determines how rapidly we update the parameters theta. If the learning rate is too large we may overshoot the optimal value. Similarly, if it is too small we will need too many iterations to converge to the best values. Given our dataset is small there's not much difference in the accuracy of the model for whatever alpha we chose, but we can observe that when the learning rate is high(0.05 and 0.1) we reach the minimum at faster rate .

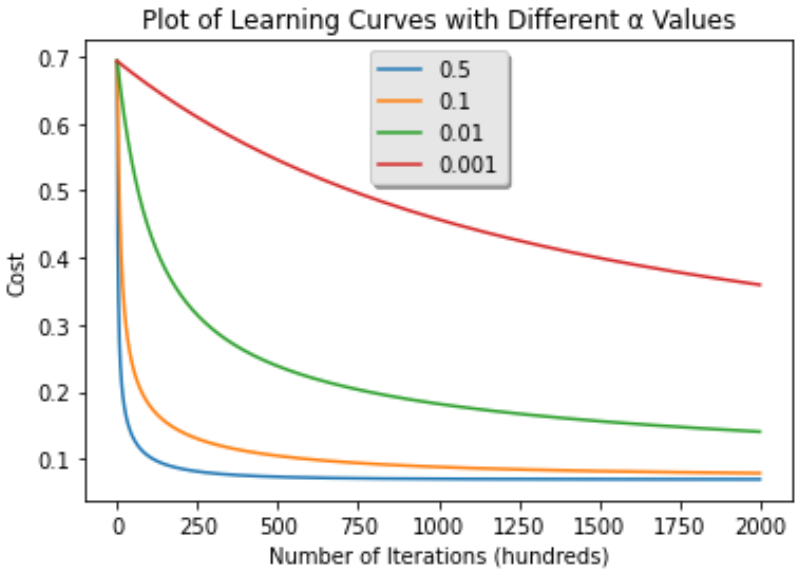


Figure 3 Plot of the number of iterations against the cost for different learning rate values. The learning rate (α) determines the speed at which the parameters update. This plot ensures a well-tuned learning rate is used to avoid "overshooting" and requiring too many iterations of the algorithm.

Conclusions and Observations

The implemented logistic regression algorithm and the scikit-learn logistic regression algorithm both have similar average test accuracies at 95.96% and 96.04% respectively. The high accuracy of the implemented algorithm when compared to the scikit learn algorithm could mean that the implemented algorithm will work well with other datasets despite being validated for the beer.txt data set specifically.

Bibliography

[1] S. L. David W. Homer, Applied logistic regression (Wiley Series in Probability and Statistics). Wiley.

[2] M. A. Nielsen, Neural Networks and Deep Learning. Determination Press, 2015.

[3] J. Lever, M. Krzywinski, and N. Altman, "Regularization," Nature Methods, vol. 13, no. 10, pp. 803-804, 2016/10/01 2016, doi: 10.1038/nmeth.4014.

[4] I. H. Witten, E. Frank, M. A. Hall, and G. Holmes, Data Mining: Practical Machine Learning Tools and Techniques. San Francisco, UNITED STATES: Elsevier Science & Technology, 2011.

[5] Professor Andrew NG, Machine Learning by Stanford University, Coursera.

Appendix : Code

```
"""
Author: Saikrishna Javvadi
Description : Importing the data and setting up the stage to create algorithm from scratch
"""
import pandas as pd
import numpy as np
from sklearn.linear_model import LogisticRegression
```

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn import preprocessing
from sklearn.model_selection import cross_val_score
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
import warnings
warnings.simplefilter(action = "ignore", category = RuntimeWarning)
#Reading the data file
df = pd.read_excel('C:\\Users\\javva\\Downloads\\beer.xlsx')
#Rearranging the columns
df = df[['beer_id','calorific_value', 'nitrogen', 'turbidity', 'alcohol','sugars','bitterness','color',
         'degree_of_fermentation','style']]

column_names = ['beer_id','calorific_value', 'nitrogen', 'turbidity', 'alcohol','sugars','bitterness','color',
         'degree_of_fermentation','style']
#droping the beer_id column
df = df.drop(columns = ['beer_id'])
#Getting the data into numpy arrays i.e X_data(independent variables) and Y_data(dependent variable/label)
X_data,Y_data = df.iloc[:,:-1].values, df.iloc[:,-1].values
#Unique classes that are to be  classified in the data
classes = np.unique(Y_data)
print(classes)
df.head()

def feature_scaling(X):
    """
    Description: Feature scaling using standarization
    Arguments:
    X -- A scalar or numpy array of any size.
    Return:
    X_scaled -- A numpy array of scaled data
    """
    mean = np.mean(X_data, axis=0)
    sd = np.std(X_data, axis=0)
    X_scaled= (X -  mean) / sd
    return X_scaled

def sigmoid(z):
    """
    Description: Compute the sigmoid of z
    Arguments:
    z -- A scalar or numpy array of any size.
    Return:
    s -- sigmoid(z)
    """
```

```
    s = 1 / (1 + np.exp(-z))
    return s

def cost_function(theta, X, Y,lambda_reg = 0.09):
    """

    Description:Implemention of the cost function with regularization

    Arguments:
    theta -- parameters, a numpy array of shape (3,8) i.e (3 - unique classes,8 - number of independent variables)
    X -- a numpy array of independent variables
    Y -- a numpy array of dependent variable(style) having true labels of data
    lambda_reg -- regularization constant lambda

    Return:
    cost -- negative log-likelihood cost for logistic regression
    A -- the probabilities for X with respect to theta
    """
    m = len(Y)
    A = sigmoid(np.dot(X,theta))
    regularization = (lambda_reg/(2 * m)) * np.sum(theta**2)
    cost=(- 1 / m) * np.sum(Y * np.log(A) + (1 - Y) * (np.log(1 - A))) + regularization
    return cost,A

def gradient(theta, X, Y,lambda_reg = 0.09):
    """

    Description:Implementation of gradient for regularized logistic regression

    Arguments:
    theta -- parameters, a numpy array of shape (3,8) i.e (3 - unique classes,8 - number of independent variables)
    X -- a numpy array of independent variables
    Y -- a numpy array of dependent variable(style) having true labels of data
    lambda_reg -- regularization constant lambda

    Return:
    cost -- negative log-likelihood cost for logistic regression
    gradient -- gradient for regularized logistic regression that will be descented in the next step
    """
    cost,A = cost_function(theta, X, Y)
    m, n = X.shape
    theta = theta.reshape((n, 1))
    gradient = (1 / m) * np.dot(X.T,(A - Y))+ (lambda_reg /m)*theta
    return cost, gradient

def optimize(X, Y, theta, num_iterations , learning_rate,print_cost = False):
    """

    Description: This function optimizes theta by running a gradient descent algorithm
```

```
    Arguments:
    X -- a numpy array of independent variables
    Y -- a numpy array of dependent variable(style) having true labels of data
    theta -- parameters, a numpy array of shape (3,8) i.e (3 - unique classes,8 - number of independent variables)
    num_iterations -- hyperparameter representing the number of iterations to optimize the parameters
    learning_rate -- hyperparameter representing the learning rate used in the update rule of optimize()
    print_cost -- True to print the loss every 100 steps

    Return:
    costs -- list of all the costs computed during the optimization, this will be used to plot the learning curve.
    theta -- updated vector of parameters(theta) after performing the gradient descent
    """
    costs = []
    for i in range(num_iterations):
        cost, _gradient = gradient(theta, X, Y)
        theta -= learning_rate * _gradient
        costs.append(cost)
        # Print the cost every 100 training iterations if required
        if print_cost and i % 100 == 0:
            print ("Cost after iteration %i: %f" %(i, cost))
    return costs,theta.reshape(8)

def predict(theta, X):
    """

    Description: Convert probabilities A[0,i] to actual predictions p[0,i]

    Arguments:
    theta -- parameters, a numpy array of shape (3,8) i.e (3 - unique classes,8 - number of independent variables)
    X -- a numpy array of independent variables

    Return:
    prediction : a numpy array containing the predictions for the data
    """
    A = sigmoid(np.dot(X,theta.T))
    #Predicting the class by taking the highest probability value
    predictions = [classes[np.argmax(A[i, :])] for i in range(X.shape[0])]
    return predictions

def accuracy(Y, predictions):
    """

    Description: Calculate the accuracy score by checking how many predictions our algorithm got right

    Arguments:
    Y -- a numpy array of dependent variable(style) having true labels of data
    predictions -- a numpy array of dependent variable(style) having predicted labels of data
```

```
    Return:
    accuracy : the accuracy of the implemented algorithm
    """
    accuracy = sum(predictions == Y)/len(Y)
    return accuracy

def model( num_iterations = 2000, learning_rate = 0.005,print_cost = False, print_accuracy = True):
    """

    Description: Builds the logistic regression model by calling the function we have implemented previously

    Arguments:
    num_iterations -- hyperparameter representing the number of iterations to optimize the parameters
    learning_rate -- hyperparameter representing the learning rate used in the update rule of optimize()
    print_cost -- Set to true to print the cost every 100 iterations

    Returns:
    d -- dictionary containing information about the model.
    """
    classification = []
    logistic_reg_accuracy = np.zeros((10))
    X_data_scaled = feature_scaling(X_data)
    ##Iterating over the implemented Logistic Regression for 10 times
    for t in range(10):
        # Splitting the data into traning and testing where training data is 2/3 of dataset randomly with the help
of "Shuffle"
        X_train, X_test, Y_train, Y_test = train_test_split(X_data_scaled, Y_data, train_size = 2/3, shuffle =
True)
        classes = np.unique(Y_data)
        theta = np.zeros((3, 8))
        i = 0
        for j in classes:
            #converting the categorical variables to 0 and 1
            Y_temp = np.array(Y_train == j, dtype = int)
            Y_temp = Y_temp.reshape((Y_temp.shape[0], 1))
            costs, optimal_theta = optimize(X_train, Y_temp, np.zeros((8,1)),num_iterations , learning_rate ,
print_cost)
            theta[i] = optimal_theta
            i += 1
        #Predicting for X_test for every iteration
        predictions = predict(theta, X_test)
        for j in range(len(predictions)):
            classification.append("%s, %s" %(predictions[j], Y_test[j]))
        #Storing the accuracies of each iteration
        logistic_reg_accuracy[t] = accuracy(Y_test, predictions)*100
        if print_accuracy:
            ##Printing the accuracies of each iteration
            print("Accuracy for ", t+1, " iteration: ", logistic_reg_accuracy[t])
    ##printing the average of accuracies across 10 iterations
    print("\nAverage Test Accuracy for Implemented Logistic Regression: ", logistic_reg_accuracy.mean(),
'%')
```

```
    ##Writing the results to file
    f = open('C:\\Users\\javva\\Downloads\\predictions.csv', 'w')
    for line in classification:
        f.write(line + "\n")
    f.close()
    d = {"costs": costs,
         "Y_test": Y_test,
         "Y_test_predictions" : predictions,
         "learning_rate" : learning_rate,
         "num_iterations": num_iterations,
         "logistic_reg_accuracy": logistic_reg_accuracy}
    return d

"""

Description : Executing the logistic algorithm built from scratch and plotting the convergence of cost
"""
d = model(num_iterations = 10000, learning_rate = 0.001)
plt.plot(range(len(d["costs"])),d["costs"],'r')
plt.title("Learning Curve ")
plt.xlabel("Number of Iterations")
plt.ylabel("Cost")
plt.show()
#Confusion matrix for the last iteration of Logistic Regression model
conf_matrix = confusion_matrix(d["Y_test"],d["Y_test_predictions"])
matrix = sns.heatmap(conf_matrix, annot = True, xticklabels = classes, yticklabels = classes)
matrix.set(xlabel = "True Values", ylabel = "False Values")

"""

Description : Comparing the learning curve of our model with several choices of learning rates
"""
learning_rates = [0.5,0.1,0.01, 0.001]
models = {}
for i in learning_rates:
    print ("learning rate is: " + str(i))
    models[str(i)] = model( num_iterations = 2000, learning_rate = i, print_accuracy = False)
    print ('\n' + "-------------------------------------------------------" + '\n')
for i in learning_rates:
    plt.plot(np.squeeze(models[str(i)]["costs"]), label= str(models[str(i)]["learning_rate"]))
plt.ylabel('cost')
plt.xlabel('iterations (hundreds)')
legend = plt.legend(loc='upper right', shadow=True)
frame = legend.get_frame()
frame.set_facecolor('0.90')
plt.show()

"""

Description : Implementation of Logistic Regression using  scikit-learn
"""
```

```
sci_learn_accuracy= np.zeros((10))
X_data = preprocessing.scale(X_data)
logistic_reg_model = LogisticRegression(solver='lbfgs', multi_class='multinomial')
kfold_result = cross_val_score(logistic_reg_model, X_data, Y_data, cv=10, scoring='accuracy')
sci_learn_accuracy= kfold_result*100
#Printing the accuracies for each iteration
for j in range(10):
    print("Accuracy for ", j+1, " iteration: ", sci_learn_accuracy [j])
print("\nAverage Scikit-Learn Logistic Regression accuracy for test data: ", sci_learn_accuracy.mean(), '%')

"""

Description : Plotting the accuracies for both the implementations
"""
plt.plot(range(10), sci_learn_accuracy, label = 'Scikit-Learn Logistic Regression')
plt.plot(range(10), d["logistic_reg_accuracy"], label = 'Implemented Logistic regression')
plt.ylabel('Accuracy')
plt.xlabel('Iteration')
plt.title('Comparing Accuracies for both the Implementations')
plt.legend()
plt.show
```