(3) One classification algorithm that was implemented in this assignment was that of logistic regression, which was used to classify whether an article was either fake news or it was not, i.e. ‘real’ news. Logistic regression is an appropriate algorithm to implement on the chosen dataset. Since the target variable of the dataset is binary, logistic regression can be used to predict the probabilities of each data point x, and then categorise each data point into 0 or 1, depending on its probability (categorise it 0 if p(x) < 0.5 or 1 if p(x) ≥ 0.5).

In an attempt to determine the best implementation of the algorithm it was implemented using two methods: gradient descent (with loss minimization) and gradient ascent (with the maximum likelihood estimation), with both implementations yielding very interesting and accurate results. By continually adjusting the hyperparameters, such as the learning rate, it was determined that the loss minimization implementation is the more accurate implementation with an accuracy of just over 95%. However, by adjusting the learning rate (alpha) on the gradient ascent algorithm, it was discovered that we could slightly increase the algorithm’s accuracy, but it would take significantly longer for that algorithm to complete running. See the discussion on hyperparameters and their effect on accuracy and time in sections i and ii.

Both implementations of the algorithm use the same set of initially randomised weights (Please see the output in the ‘LogisiticRegression.ipynb’ for the entire set, only the first 8 elements will be selected for this document for readability purposes):

ϴ = [0.002547, 0.455505, 0.833198, 0.32285, 0.471227, 0.691664, 0.20063, 0.438592]

1. Gradient Descent with Loss Minimisation (Logistic Regression)

Running this algorithm on the dataset yields following set of weights:

ϴ = [4.526196, 4.979154, 5.356846, -0,5.5113, -3.170662, -6.099912, -0.800181, -7.758719]

As one can see, the weights that are eventually learnt do not grow/decrease to incredibly large amounts when compared to the initially randomised values for the weights.

See below the error on the test set taken directly from the Jupyter Notebook output in the ‘LogisticRegression.ipynb’ file (illustrated with the assistance of the scikit-learn library):

* **Confusion Matrix:** [1944 104] [77 1897]
* **Accuracy Score:** 0.9549975136747887
* **Report:**

**precision recall f1-score support**

0.0 0.96 0.95 0.96 2048

1.0 0.95 0.96 0.95 1974

**accuracy** 0.95 4022

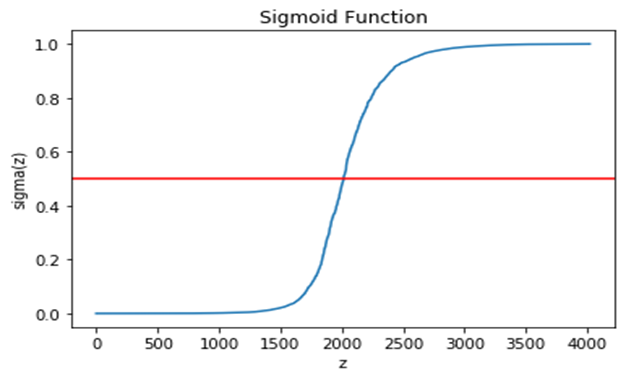
**macro avg** 0.95 0.96 0.95 4022

**weighted avg** 0.96 0.95 0.96 4022

For this implementation of the gradient descent algorithm, we chose alpha = 0.8. As you will see in the table below there is an almost positive, linear correlation between alpha and the accuracy of the model. Increasing alpha by 0.1 slightly increases the accuracy of the model, but it also somewhat increases the time in which it takes for the gradient descent function to complete running. After running and timing the duration of the gradient descent algorithm on a test set (which was about 20% of the data) we got the following results:

|  |  |  |
| --- | --- | --- |
| Alpha | Accuracy | Time (in seconds) |
| 0.95 | 0.950522 | 117.660653 |
| 0.9 | 0.950273 | 100.889569 |
| 0.8 | 0.950273 | 96.5892434 |
| 0.7 | 0.949528 | 98.431086 |
| 0.6 | 0.949279 | 96.678892 |
| 0.5 | 0.949030 | 85.790105 |
| 0.4 | 0.948284 | 99.308077 |
| 0.3 | 0.94729 | 100.289893 |
| 0.2 | 0.946544 | 110.43432 |
| 0.1 | 0.94182 | 98.144306 |

As you can see, if we were to pick an alpha in the interval (0.8, 1) we could possibly find an alpha that would yield a higher accuracy, but that would make the gradient descent take a slightly longer amount of time to run. In addition, slightly increasing after alpha = 0.9 increases the accuracy by 0.000249%, which is almost insignificant if there’s already an accuracy of 95%. Clearly, alpha = 0.8 seemed to be the most optimal value to pick when scaled up to a much larger dataset. Below is the Sigmoid function produced by the gradient descent function implemented on the test set:



1. Maximum Likelihood Estimation with gradient descent (Logistic Regression)

Running this algorithm on the dataset yields following set of weights:

ϴ = [7046.150991, 7046.150991, -1113.332136, -4830.573828, -9784.339699, -635.85943, -9935.953705, 4516.882078]

As one can see, the weights grow/decrease to incredibly large values when compared to those after running the loss minimization implementation of gradient descent, and even those of the randomly initialised set. Perhaps implementing this algorithm with regularization would be necessary to ensure that the values do not become too high.

See below the error on the test set taken directly from the Jupyter Notebook output in the ‘LogisticRegression.ipynb’ file (illustrated with the assistance of the scikit-learn library):

* **Confusion Matrix:** [2015 33]

[272 1702]

* **Accuracy Score:** 0.9241670810542019
* **Report:**

**precision recall f1-score support**

0.0 0.88 0.98 0.93 2048

1.0 0.98 0.86 0.92 1974

**accuracy** 0.92 4022

**macro avg** 0.93 0.92 0.92 4022

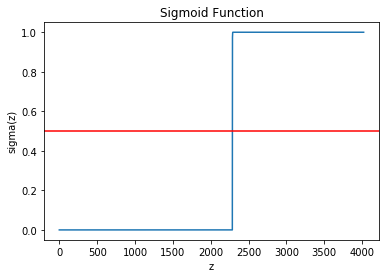
**weighted avg** 0.93 0.92 0.92 4022

The accuracy and results of this specific iteration of this algorithm slightly differs from those of the discussions of alpha values that are to follow. This could be because of the initial weights that are randomly initialised, so the algorithm is always working on a different set of thetas.

For this implementation of the gradient ascent algorithm, we chose alpha = 0.5. As you will see in the table below there is an almost positive, linear correlation between alpha and the accuracy of the model (though the correlation is not as strong as in the loss minimization implementation of the gradient descent algorithm). Increasing alpha by 0.1 slightly increases the accuracy of the classifying model, but it also somewhat increases the time in which it takes for the gradient ascent function to finish running. We ran and timed the duration of the gradient ascent on a test set, and got the following results:

|  |  |  |
| --- | --- | --- |
| Alpha | Accuracy | Time (in seconds) |
| 0.95 | 0.954251 | 165.928035 |
| 0.9 | 0.94903 | 145.716267 |
| 0.8 | 0.948036 | 146.313839 |
| 0.7 | 0.944306 | 148.20403 |
| 0.6 | 0.950025 | 144.38948 |
| 0.5 | 0.948036 | 102.46435 |
| 0.4 | 0.935107 | 154.129854 |
| 0.3 | 0.948533 | 143.555976 |
| 0.2 | 0.935853 | 152.79072 |
| 0.1 | 0.867976 | 137.240029 |

Similar to the loss minimization implementation of gradient descent, picking alpha in the interval (0.5, 1) could possibly yield a higher accuracy, but that would result in longer computation of the gradient ascent. Clearly, alpha = 0.5 would be the most optimal value to pick when scaled up to a much larger dataset. Below is the Sigmoid function produced by the gradient descent function implemented on the test set:



References

1. Kaggle.com. 2020. *Logistic Regression From Scratch - Python*. [online] Available at: <https://www.kaggle.com/jeppbautista/logistic-regression-from-scratch-python> [Accessed 17 April 2020].
2. Bird, R., 2020. *Logistic Regression*. [online] Medium. Available at: <https://medium.com/greyatom/logistic-regression-89e496433063> [Accessed 18 April 2020].