Annie He

Clinical Readmission Prediction Report

For this project, the setting included over 28,000 ICU admissions and 68 features. Each data point (patient) had a binary classification label (0, 1) of whether they were readmitted or not within 30 days after their initial discharge. Our goal was to determine and implement a machine learning model that would best predict whether a patient will be readmitted or not.

We have chosen to implement Logistic Regression, a regression analysis used to model a dichotomous dependent variable. We believe this model is suitable for this dataset because it calculates a probability for readmission. This probability can be converted to a binary classification label based on a decision threshold, which we believed to be easy to understand. In addition, we are able to optimize our model for Logistic Regression by fine tuning the hyperparameters. Logistic Regression is similar to a one layer neural network, in that the weights are being updated in one layer of data, not multiple. Furthermore, Logistic Regression uses a logarithmic transformation on the outcome variable to model a nonlinear association in a linear way. We decided not to implement other models because they were either computationally heavy (Random Forest), assumed that features were independent (Naive Bayes), or would result in overfitting (AdaBoost).

Logistic Regression continuously updates the weights and produces unbounded continuous values. Each weight represents how important the corresponding feature contributes to the outcome prediction. We first multiply the features and their respective weights together, along with adding the bias weight. By plugging in this value to the sigmoid function (Figure 1), we squash the arbitrary probabilities to be bounded in the [0,1] interval. This results in an S-shaped curve of probability predictions.

$$f(\mathbf{x}) = \frac{1}{1 + e^{-(w_0 + \sum w_i x_i)}}$$

Figure 1. Sigmoid function

To evaluate how accurate our predictions were and to optimize the loss to be as close to the minimum, I implemented the stochastic gradient descent method (SGD) and the cross entropy loss (Figures 2 and 3). By calculating the derivative of the sigmoid function, a parabola-shaped curve, or gradient, was created. This derivative was then multiplied by the feature values. After normalizing and multiplying by the learning rate, these values were subtracted from the weights to produce the updated weights. In the program, I trained on mini batches to update the weights iteratively. With each batch, my team adjusted the weights in the direction of the gradient and according to the learning rate. With each iteration (epoch), we calculated and documented the loss.

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{w}_{t-1}}$$

Figure 2. Stochastic gradient descent

$$\mathcal{L}(x) = -\sum_{c=1}^{K} y_c^* log(p(c|x))$$

Figure 3. Cross entropy loss

After our initial implementation of this model, the F1 measures and accuracy scores were not what we expected. We were getting over a 90% accuracy rate for Class 0, but only an 8% for Class 1, which meant that our model was predicting way more 0's. Therefore, we analyzed the data and realized that almost 90% of the dataset were Class 0, while only 10% were Class 1 (readmitted patients). This was a huge class imbalance! We thought of oversampling and undersampling as a way to address the imbalanced data. Undersampling would decrease the runtime, but exclude some data points. Thus, Wonyoung and I decided oversampling would be the better option (Figure 4). To oversample the minority data, we created a factor value, which is the truncated value of the size of the majority class divided by the size of the minority class. After implementing oversampling, we noticed a much better balance in the data, and higher accuracy and F1 scores all around. I also tried to decrease the factor by 1, and the results were nearly the same, while decreasing runtime.

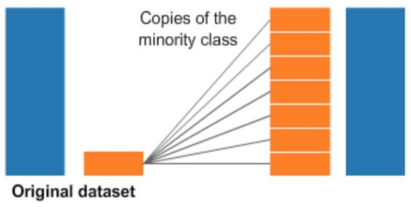

Figure 4. Oversampling representation

Joshua also suggested that we try to implement Principal Component Analysis (PCA), a dimensionality reduction method. By reducing the number of features while maintaining the variance, we decided on 50 features (Figure 5), which kept the variance close to 100%. After fitting and transforming the data, we expected for PCA to improve our model. However, although the accuracy and F1 scores were slightly lower than those without PCA, the loss increased over time. This meant that the weights were not being adjusted according to the gradient. Therefore, we have decided not to include PCA in our model, but our code attempt is included in our GitHub folder as `pcaattempt.py`.

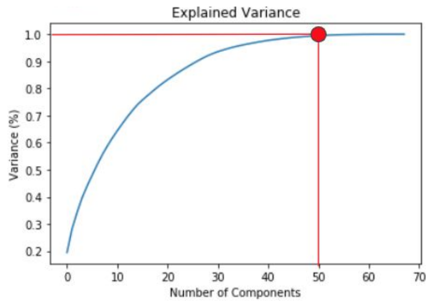

Figure 5. PCA analysis

Lastly, we tuned the hyperparameters (batch size, number of epochs, and learning rate) to achieve the optimal accuracy rates. Through numerous attempts, I managed to finalize on a batch size of 1200, 500 epochs, a learning rate of 0.0015 or 0.001, and a decision threshold of 0.5 (Table 1). We noticed that a lower learning rate led to better accuracy, but increased runtime, due to the fact that a lower learning rate traverse down the gradient at a slower but more precise pace.

Batch size	Number of epochs	Learning rate	F1 score (Class 0)	F1 score (Class 1)	Total accuracy	Class 0 accuracy	Class 1 accuracy
1200	500	0.1	0.84	0.37	0.744	0.745	0.735
1200	500	0.01	0.87	0.39	0.78	0.793	0.684
1200	500	0.003	0.878	0.384	0.8	0.817	0.618
1200	500	0.002	0.889	0.389	0.813	0.839	0.583
1200	500	0.0015	0.901	0.396	0.831	0.864	0.542
1200	500	0.001	0.923	0.402	0.864	0.912	0.444
1200	500	0.0008	0.923	0.37	0.872	0.93	0.367

Table 1. F1 scores and accuracy rates based on different hyperparameters

Our Logistic Regression model, with oversampling and batching and without PCA, performed quite well. We achieved peak F1 scores of 0.92 (Class 0) and 0.40 (Class 1), and accuracy rates of 0.91 (Class 0) and 0.54 (Class 1) (Table 1). At first, our runtime was over 10 minutes. However, I was able to reduce the runtime through the use of the NumPy package. By taking advantage of parallelism, our NumPy operations significantly decreased our runtime compared to previous for loops and Python lists. Future directions would include decreasing our runtime further, improving our PCA approach, and further preventing overfitting with cross-validation and regularization.