# Final Project Write-up

**CSCE 421 - 500 - Spring 2023**

**Gaultier Delbarre** **Brian Nguyen** **Ian Sabolik**

## Abstract

This project attempts to predict in-hospitality mortality based on clinical health data, which could be used to improve patient outcomes. The testing/training data is a preprocessed subset of the eICU dataset. After preprocessing, the data contains about 50 columns of standardized, numerical data that we fed into a densely connected neural network, with the goal of predicting the mortality of the patients. Our method produced strong results, with an AUC-ROC score of 0.88522 from a held-out dataset. Overall, our success in developing a highly effective model reflects the strength of our methodology and the knowledge of machine learning that our team has gained throughout the semester.

## 1 Introduction

The goal of this project is to create a machine learning model that is able to predict in-hospital mortality after being trained on a preprocessed subset of the eICU dataset, which is a dataset that contains deidentified, clinical health data for admissions to intensive care units across the United States, between 2014-2015. The ability to make these predictions is important because it can potentially improve patient outcomes, as an early identifier of patients at a high risk of mortality can allow hospitals to allocate more resources to those patients and give them treatment in a more timely manner, increasing their chances of survival.

To solve this problem, we began by preprocessing the data so that it was in a form that a model could use, while still containing valuable information. This involved several steps that we go into more depth about in the Data Preprocessing section (see "Data Preprocessing"). It involved one-hot encodings, making new columns from means, medians, and maxes, rearranging the data, dealing with empty cells, and standardizing the data. We built a model around this data and performed significant hyperparameter tuning to make the model as accurate as possible. The model itself is a dense neural network with a linear input layer and densely connected hidden layers (the number and size of which were also tuned as hyperparameters), with Adam as our optimizer of choice.

The results of this method were promising, with a final AUC-ROC score of 0.88522, indicating that the model performs reasonably well at distinguishing between positive and negative cases. Our team's approach was carefully considered and executed, and these strong results show the effectiveness of our methodologies in both the design of the neural network and the preprocessing of the dataset.

## 2 Our Method

### 2.1 Data Preprocessing

Our team did extensive data preprocessing to arrange the data into a more usable form while maximizing the information available for our model to train on. To achieve this, we decided to shape the data such that there was only one row per patient, and that all cells were filled with numerical values.

#### 2.1.1 Labs and Nursing Chart Cells

For both types of lab tests, the labs and nursing chart columns, we separated each type of test to have its own unique column, in order for each patient to have their own readings. Then, by matching the patient unit stay ID to each test, we filled said columns with the matching test data for each test. If a patient had multiple readings for a certain test, the average over all of the tests of this type was used. Each patient now has columns indicating their specific readings for each available test.

#### 2.1.2 One Hot Encoding

The columns for ethnicity and gender lent themselves simply to one hot encode, as we wanted numerical data to feed into the model.

Looking at the dataset as a whole, not every patient has data for every lab, test, or attribute. Thus, we needed to find a way to fill those blank spaces as leaving cells blank was not an option as they showed up as NaN values, which are incompatible with our model. We decided to fill those spaces with 0, but this would lead the model to assume that this specific data was measured at 0, rather than meaning such data was not available. To remedy this, we implemented a system of essentially one hot encoding, creating columns that indicated whether or not this patient had this certain attribute or took this lab or test. This strategy was applied to columns that had missing data, specifically age, weight, height, GCS Total, Heart Rate, Invasive BP Diastolic, Invasive BP Mean, Invasive BP Systolic, Non-Invasive BP Diastolic, Non-Invasive BP Mean, Non-Invasive BP Systolic, O2 Saturation, Respiratory Rate, glucose, and pH.

#### 2.1.3 Offset and Capillary Refill

For column offset, which measures the time after admission when a measurement was taken, we realized that organizing this data to attach to every test was going to exponentially increase our preprocessing time and size, as each test would need to have its offset recorded, and this would be done for each patient.

Instead, we created a max offset column and median offset column for each patient and recorded across all measurements taken what was the maximum time after admission a measurement was administered and what the median time was, in order to give the model reasoning for how long a patient was under care for, as well as when the bulk of their measurements were taken. We reasoned that the original data had offsets for measurements in order to get context on the timeline of a patient's stay, which we believe our measurements do as well.

The columns for celllabel and cellattribute contain information about a single test administered, called Capillary Refill. Celllabel contained multiple attributes and results of this test, such as where it was taken and how long it took for blood to flow back. However, when analyzing the data, we realized that for many patients, there was no uniformity in what tests were administered to a patient, leading some patients to have readings that said they took it on their hands or feet but didn't have a reading for how long it took for blood to flow back. In addition, many patients simply did not have readings for this test, just that the few that did had many measurements taken. Lastly, research on capillary refill tests from the NIH suggests that this test be administered to ill patients.

Considering all these points, we reasoned that the time it cost to process this would not be worth the data, especially given that it applies to few patients. Therefore, we created a new column that indicates whether or not a patient had a capillary refill measurement taken, letting our model see that these patients required this assessment, and thus are more likely to be ill or have a predisposed condition.

#### 2.1.4 Standardization

The measurements taken in labs and tests, as well as attributes of patients, varies widely in terms of numeric range. Thus, in order for the model to equate equal importance to changes in measurements

and across features, we utilized standardization across virtually all features, save for patient unit stay ID, as that acts as an identifier to differentiate patients.

#### 2.1.5 Oversampling

Due to the high imbalance in the number of positive samples and negative samples (168 positive vs. 1868 negative), we implemented an oversampling function which would produce a sample of a given size with equal amounts of positive and negative labels contained. We did this by splitting our labels and data by hospital discharge status. We then sampled with replacement from both the positive and negative labels until we had the desired amount of both. We then created a new DataFrame of sample data by fetching the corresponding data for each patient ID found in the resampled labels. This produced an oversampled dataset that contained a known proportion of labels.

Obviously, oversampling introduced the risk of overfitting on our data when training. We deemed this to not be problematic however (see "Tuning Loss Functions").

### 2.2 Model Design

We used a dense neural network for our model. We used a Linear input layer taking in all 49 features that we preprocessed, and a Linear output layer followed by a sigmoid layer to produce a probability of in-hospital death. In between the input and output layers, we used densely connected hidden layers. We used hyperparameter tuning to find the optimal number of hidden layers and the size of the hidden layers. We used Adam as our optimizer and binary cross entropy as our loss function. The reasoning behind these decisions is explained in "Tuning Optimizer Functions" and "Tuning Loss Functions".

### 2.3 Model Training

We split our dataset into training and testing sets using a random permutation of our original dataset. We then passed this to our custom dataset class based on PyTorch's `DataSet` class. We used the custom dataset to create `DataLoaders` for the training and validation sets. Lastly, this prepared data was sent to the GPU for hardware-accelerated training and sent to the model's `fit` function to train on.

In the model's `fit` function, training was split between training and validation, based on the two `DataLoaders` given to the function. The training portion simply ran a feed-forward cycle, calculated the loss using the loss function, and then back-propagated the gradients using the optimizer function. A small selection of performance metrics was calculated for each batch trained (see "Metrics Used"). The same metrics were used for each batch of the validation portion. These metrics were averaged by epoch trained and then returned to provide insight into how well the model was trained.

After training the model to our satisfaction and hyperparameter tuning, we got metrics on the model's performance on the entire dataset by using our `getMetrics` method. These metrics were usually slightly different than the ones received during training for 2 reasons. Firstly, because we used oversampling (see "Oversampling"), the training dataset was different in composition compared to the whole dataset. Secondly, the metrics are calculated for each batch so their composition is also different than that of the whole dataset.

### 2.4 Hyperparameter Tuning

In this project, we used hyperparameter tuning extensively to understand what our models were doing and how to optimize them as well as possible.

#### 2.4.1 Metrics Used

We used 4 different metrics while hyperparameter tuning in order to best comprehend the changes happening to our models. We used balanced accuracy, $F_1$ score, AUC-ROC score, and loss. We settled on these metrics after trying a variety of metrics. Realizing that our dataset was highly imbalanced in favor of the negative class, we decided to stop using raw accuracy as a performance metric, as it did not provide useful information on whether we were properly classifying the positive class. We settled on a combination of $F_1$ score and balanced accuracy to replace it. This was better

than simple accuracy because balanced accuracy gave a clearer picture of whether we were accurately classifying both classes, and $F_1$ score provided a view of the classifying strength for positive samples.

### 2.4.2 Tuning Procedure

Our tuning procedure was akin to grid search with either 1 or 2 dimensions. For certain parameters such as model structure, we grid searched over the number of hidden layers and their size. For other parameters such as learning rate and number of epochs, we simply optimized a single parameter at a time.

We did not use a pre-made grid search such as SciKit-Learn's `GridSearchCV`. We simply ran a single or double for-loop and created and trained our models on the selected hyperparameter. We chose to train models multiple times for each hyperparameter being trained in order to reduce the variance in our metrics. We did this because we noticed that identical models would converge to slightly different answers which would produce different metrics, and we wanted to capture a good slice of what the "true" trained model would look like for a certain hyperparameter.

### 2.4.3 Tuning Loss Functions

Due to our highly imbalanced dataset, we knew that the choice of loss function would have a large impact on our final model. Some loss functions, such as Cross Entropy and Negative Log Likelihood are able to weigh the output classes' loss differently to help train on imbalanced datasets. Other loss functions, such as Mean Squared Error and Binary Cross Entropy cannot do this by default. We also tested the weighted and unweighted loss functions on our oversampled dataset (see OverSampling).

Our tuning for loss was very straightforward: we set the loss function, trained 5 models on it, and took the average of our predefined metrics to compare. We saw no difference in weighted Cross-Entropy and weighted Negative Log Likelihood loss, both with and without using an oversampled dataset. However, we did notice a very large difference between the weighted and unweighted variants when not using an oversampled dataset. Except for when we used the weighted loss functions, attempting to train the neural network without the oversampled dataset proved nearly impossible. The model would quickly minimize its loss by realizing that it could predict the negative class for every sample and thereby achieve a decent loss and raw accuracy score. By switching to using oversampling in combination with a non-weighted loss function, we were able to reliably train a model which would predict reasonably accurately for the positive and negative classes.

We decided to use Binary Cross Entropy loss because it trained the most stably and was specifically suited to a binary classification problem. In conjunction with our oversampling technique, it produced the most accurate models when compared to the other loss functions.

### 2.4.4 Tuning Optimizer Functions

Owing to the advice given in class by Dr. Mortazavi, as well as the recommendations of numerous online articles on the subject, we decided to forgo serious tuning in selecting the optimizer function and settled on Adam as our optimizer. Preliminary testing was performed with other optimizers, such as RMSProp, SGD, and Adamax. No significant difference was measured in the performance of these optimizers (except for stochastic gradient descent which predictably took very long to converge).

We did attempt to tune the hyperparameters of Adam. Namely, we tuned the learning rate and the L2 penalty. Our results showed that our models trained the most stably when the learning rate was very low (around $1 \times 10^{-5}$). Our results for L2 penalty show very little difference between a high penalty (1) and very low penalties ($1 \times 10^{-5}$). Because of this, we opted to use the default set by PyTorch which was 0.

### 2.4.5 Hyperparameters Tuned

Along with the hyperparameters mentioned in the previous two sections, we also tuned the number of epochs to train the model for, the oversampling size (see 2.1.5), and batch size. These were simple to tune, and we simply created a list of good values using `numpy.linspace` or `numpy.logspace` and iteratively tested training a group of models on the hyperparameters.

We found that batch size had no effect on the overall training of the model except for its speed, so we set the batch size to 128. When tuning the number of epochs to train for, we increased the number of epochs until the validation and training losses flat-lined when plotted. This resulted in us testing a range of epochs, from only 10 epochs all the way up to 1000 epochs. Due to our very small learning rate, we knew that we would require a large number of epochs to get a well-trained model. At the same time, however, we were conscious of possible overfitting that could happen. Bearing this in mind, we selected to train our models for 125 epochs at the chosen learning rate, as it was the point on our loss-vs-epochs graphs where the loss began to flat-line for both the validation and training. Additionally, we verified that this point coincided with a flat-lining of the increase in the AUC-ROC score and $F_1$ score.

## 3 Results

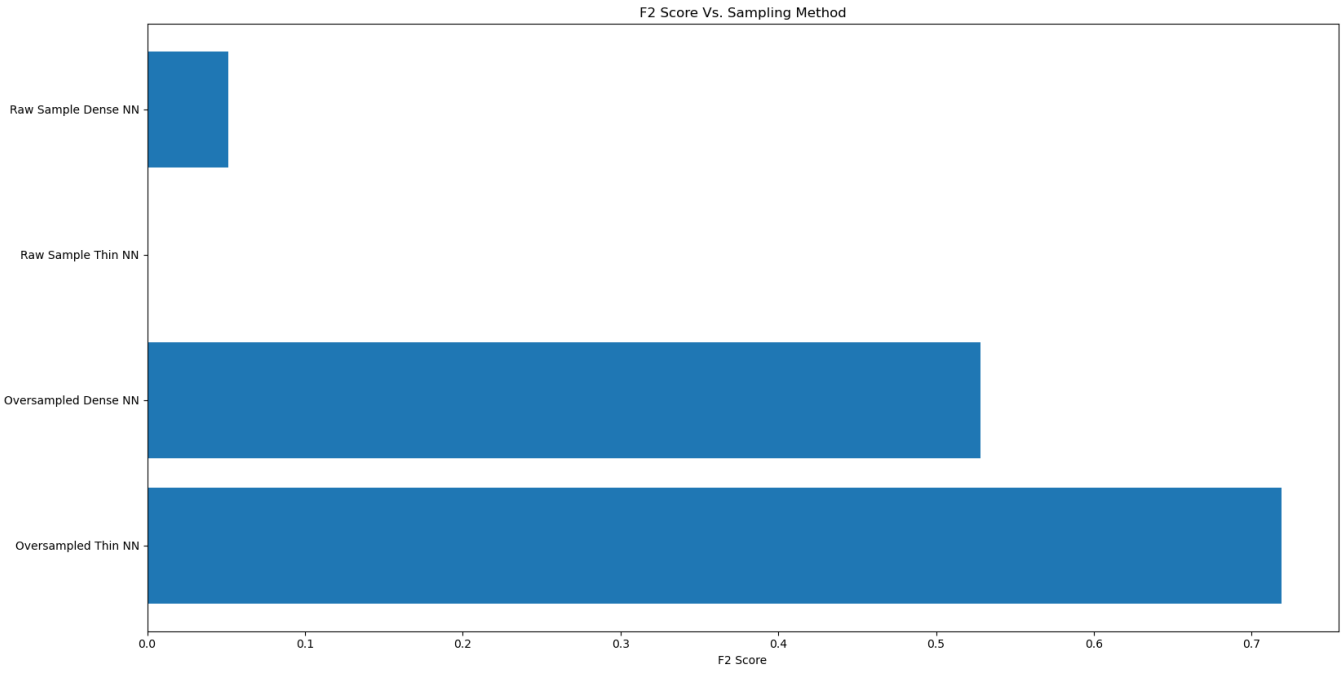


Figure 1: A comparison of $F_2$ scores by type of model. "Thin NN" means no hidden layers and an input layer size of 50. "Dense NN" means 3 hidden layers and a layer size of 100.

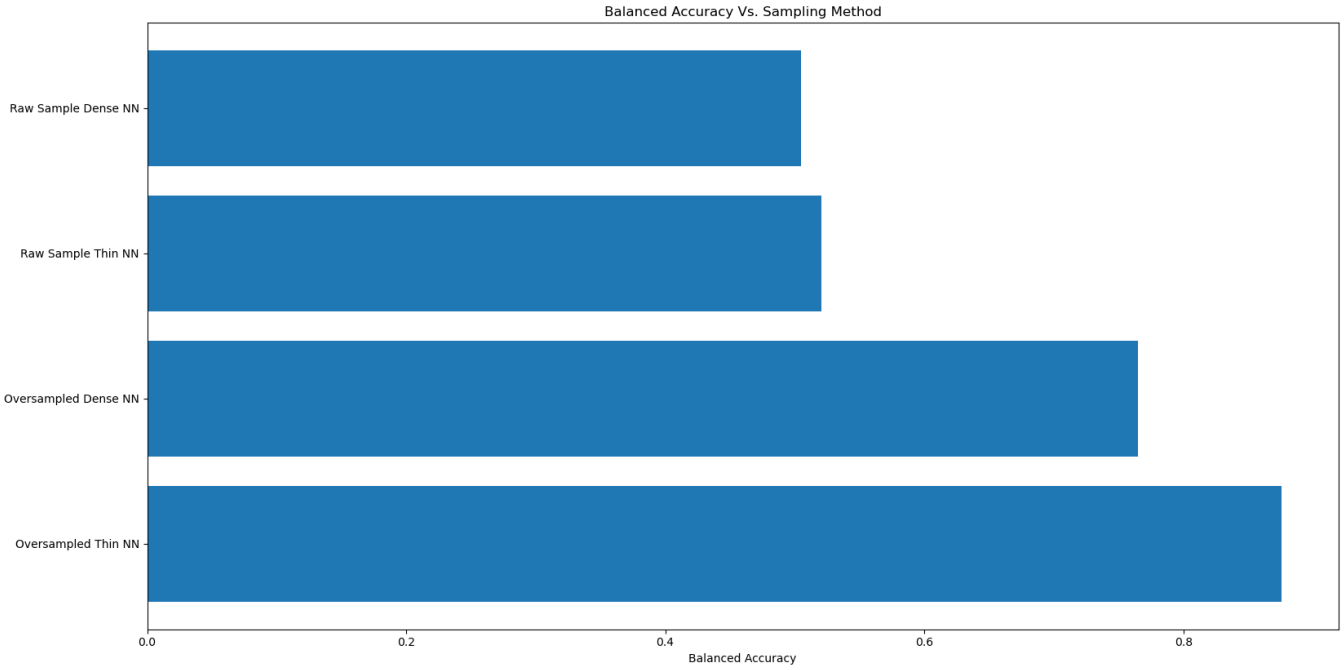


Figure 2: Comparing the use of oversampling to their relative accuracy, in order to help decide if oversampling benefits the model. "Thin NN" means no hidden layers and an input layer size of 50. "Dense NN" means 3 hidden layers and layer size of 100.

Our best recorded AUC-ROC against Dr. Mortazavi's work was 0.88522, which demonstrates a strong capability to classify hospital mortality. We also tested our own models against a variety of modifications during our hyperparameter tuning stages. As seen in Figure 1, models trained on non-oversampled data ("raw sampling") perform significantly worse than models trained using oversampling. Figure 2 paints a similar picture using balanced accuracy as a performance metric.

## 4 Conclusion

We believe that we did a thorough job of preprocessing our data and converting highly variable data with many different features and a large variation in the amount of data per sample into data of uniform size with a high amount of retained information. We also successfully used oversampling to overcome the issue of imbalanced training data. In addition, our hyperparameter tuning was meticulous, experimenting with different loss functions and parameters to narrow down the best choice for our data and goal. All these factors combined led us to create a model which outputs predictions that we are confident in and that metrics show is valid and accurate.

In an ideal scenario, we would want much more training data to work with. More data would allow us to train a variational autoencoder or other models to produce more training data without introducing the possibility of overfitting on our positive cases. Not only more data, but more time to preprocess data would also be advantageous. For the majority of the features given, it was extracted without having to make a compromise. However, for features such as the measurements on capillary refills and the offset of time between admission and assessment, compromises had to be made to reduce the viability of this data in order to deliver the model on time. Both were reduced in a manner to convey to the model the same context around the patient's health, but in a way that differed from the original data. With more time, this original context could be preserved and would lead to a better

representation of the patient. Despite these limitations, we believe that we delivered a model that is robust, accurate, and worthy of our standard of quality and work.