Omar Battisha
Siva Bhavani
3/9/2020
MACS 33002: Final Report

Predicting Mortality from Infection using Vital Signs

Overview

Our goal is to use longitudinal data of an infected patient's vital signs from the first 24 hours of their hospitalization, to predict whether or not they will die within 30 days using machine learning models.

Introduction

Sepsis is defined as a dysregulated response to infection that results in life-threatening organ dysfunction often leading to death. Sepsis is responsible for over 20 billion dollars in healthcare spending and is the cause of over 250,000 deaths in the U.S. every year. It is very important to understand the prognosis of patients with sepsis. First, prognosis could help with resource allocation. For instance, patients at higher risk of decompensation and mortality may need higher levels of care. This may entail transfer to the ICU or even a different specialty hospital. Second, identifying patients at high risk of mortality could lead to earlier more aggressive interventions such as vasopressors or broad spectrum antibiotics. Third, prognostication could be very important in making end of life decisions. For instance, a patient who has metastatic cancer and liver disease, who is admitted with an infection may have very, very low chance of survival. Knowing this information through an objective prognostication tool could inform conversations with the patient and the family about their wishes.

Literature Review

Several methods have been used to prognosticate mortality in patients with infection. APACHE (Applied Physiology and Chronic Health Evaluation) scores are commonly used prognostication tools. These scores use static measures of physiological data like vital signs to predict mortality in patients in the intensive care unit (ICU). SOFA (Sequential Organ Failure Assessment) scores are another prognostic tool that uses static vital signs and laboratory data to predict mortality. eCART is a continuous predictive algorithm that uses data at each time point to give an updated risk of cardiac arrest or mortality in the hospital. MEWS (Modified Early Warning Score) is a simple physiological score that also is used for prognostic purposes. However, all of these prognostic models use static measurements of vitals or laboratory data. Static measurements may not adequately capture the dynamic nature of sepsis. Instead, leveraging the widely available longitudinal data in the electronic health record (EHR) could result in much more sophisticated and accurate prognostic models. There is a paucity of research on utilizing this longitudinal data for prognosticating patients with infection.

Patient Population

Patients with infection were identified using the criteria published by Rhee et al., which require a blood culture order and at least four consecutive days of antibiotics (or antibiotics continued until one day prior to death or discharge), with the first day of antibiotics required to be a parenteral agent given within 48 hours before or after the blood culture order. In order to create a more homogenous population, the cohort was narrowed to patients who met Rhee criteria for infection

in the emergency department and received antibiotics within 24 hours of presentation, with presentation defined as time of first measured vital signs.

Dataset

The derivation dataset will be gathered from 2,580 de-identified adult patients admitted to a tertiary medical center. There will be no identifying names, dates of birth, or hospitalization dates. The data will consist of time-varying and time-invariant data from the first 24 hours of hospitalization (see Data Structure below). The outcome our model will predict is 30-day in hospital mortality. We will validate the model in a separate test dataset with a similar number of de-identified adult patients. All numerical data were normalized to values between 0 and 1. For missing vital sign values, the last value was carried forward.

Data structure of time-variant data

Patient	Hour of hospitalization	HR	RR	sBP	dBP	Temp	O2
1	0						
1	1						
1	2						
1	3						
1	4						

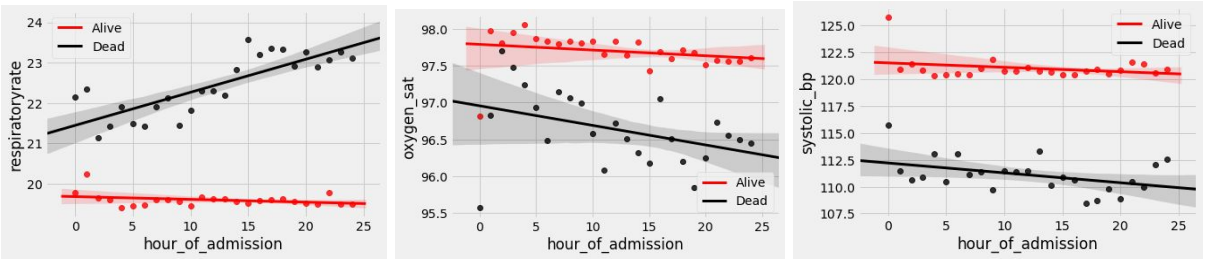
1	5						
1	6						
1	...						
1	24						

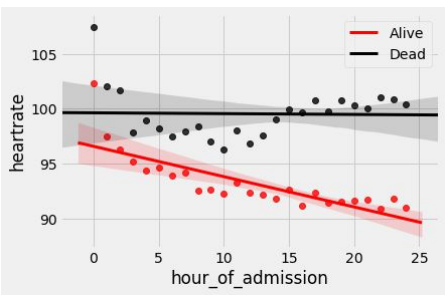
Definition of abbreviations: HR - heart rate, RR - respiratory rate, sBP - systolic blood pressure, dBP - diastolic blood pressure, Temp - temperature, O2 - oxygen saturation

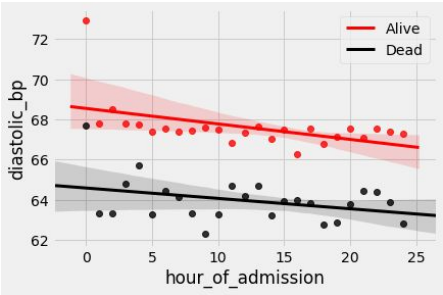
In addition to these time-varying measurements, patients will have the following information: age, presence of chronic diseases (heart failure, lung disease, liver disease, renal disease, diabetes mellitus, hypertension and cancer) and the outcome (mortality in 30 days).

Initial Observations in the Data

As an initial test to determine if our vital signs parameters could even be used to predict patient mortality, we've charted averages of each vital sign over time for both mortal patients and survivors, and attached 95% confidence intervals. The graphs for respiratory rate, blood pressure, heart rate and oxygen saturation are shown below.







As our charts show, not only non-survivors and survivors have disparate vital sign values, but they also tend to have differing vital sign trends in the first 24 hours. For example, while patients who end up surviving tend to have lower, and slightly decreasing respiratory rates in the first 24 hours, mortal patients tend to have higher and strongly increasing respiratory rates in the first 24 hours. Importantly, the confidence interval shading for the graphs show that the differences between the vital signs for the survivors and the non-survivors are unlikely to be due to chance, and therefore give us confidence in using vital signs as parameters for determining patient mortality.

Methods

Three separate methodologies were used: 1) traditional machine learning models 2) time-based prediction, and 3) recurrent neural networks.

Traditional machine learning models

In addition to the features in the dataset, additional features were extracted from the time-variant vital signs. Maximum, minimum and standard deviation of each vital sign were calculated and used as features in the model. Additionally, linear regression models were applied to model each

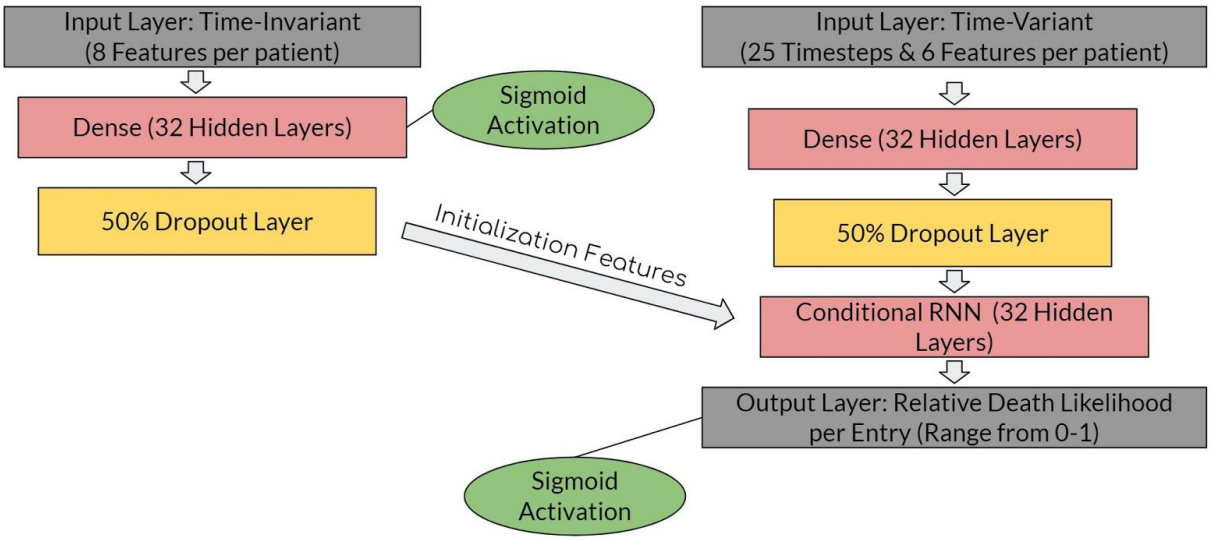
vital sign over time for each patient, and the regression coefficients for each patient were used as features in the model. The dataset was partitioned into training and testing data sets with a random 80/20 split. The models were trained on the training data set and validated on the testing data set. We applied logistic regression, random forest and gradient boosted machine models to the training data set with 5-fold cross validation, with the metric of AUC for tuning. In the random forest model, the number of trees were left at the default of 500. The mtry was tuned across 3 potential values (13,14,15), calculated based on the square root of the total number of predictors in the model. In the gradient boosted machine (GBM) model, only a single grid was tested given computational limitations. The hyperparameters in the GBM grid were as follows: nrounds = 2000, max_depth = 20, gamma = 0, colsample_bytree = 1, min_child_weight = 2, subsample = 0.5. The grid was chosen based on prior work with similar datasets and outcomes.

Time-based prediction

The dataset was partitioned into 25 cohorts, each cohort representing a different time period (from hour 0 to hour 24). Each cohort was then randomly partitioned into training and testing data sets with a 50/50 split. A random forest model was developed on each of the 25 training cohorts, and applied to the 25 testing cohorts. The random forest model was developed on a single grid given computational limitations (ntrees = 500 and mtry = 14). The resulting predicted probabilities of death over the 25 hours were compared between survivors and non-survivors.

Recurrent Neural Networks

In order to effectively take advantage of the longitudinal data that we have, we required a Machine Learning model that had a "memory" of sorts; it needed to be able to analyze each hourly data point in the context of the data points that preceded it to effectively make a prediction. Furthermore, that "memory" had to be able to take into account multiple preceding predictions, as our initial observations show that the trends present in the vital signs are best understood within the context of many preceding points. The model we believed fit these criteria best was a Recurrent Neural Network (RNN) with Long Short Term Memory (LSTM). While RNNs allow us to take into account previous data points, the addition of LSTM units--as opposed to typical RNN units--allow us to consider a larger trailing string of data points and effectively decide which data points in that string are of relevance, making their combination an optimal application for the task at hand. Finally, the last layer of nuance that we introduced to our RNN is that of multiple input types—taking in both time-variant and time-invariant features. In order to do this, we utilized a Conditional RNN, which allows the use of time-invariant features as initialization features for the RNN.



To implement this sort of RNN, we used the Cond_RNN package in combination with the Tensorflow Keras Functional API, which allows us to choose the specifics of our RNN's structure and layering. **(Figure Above)** details the structure that we eventually decided to use. Multiple features and placements used in the model require appropriate justification. Firstly, unlike typical RNNs, we have two inputs (one time variant and one time invariant), with the outputs of the time invariant chain being used to initialize the Conditional RNN (i.e. they are placed in the initial time step as variables, and the the LSTM decides whether to keep them or not). This is necessary to utilize both types of data in our Conditional RNN. A second feature of importance is the Dropout. Our largest issue in attaining a high accuracy with the RNN was overfitting; while training accuracies would regularly keep increasing, testing accuracies would easily stagnate. To remedy this, we looked towards research from the University of Toronto on using Dropout to decrease overfitting[1]. On a basic level, dropout hides a different random percentage of data and weights in each forward and backward propagation, in order to force the model to generalize over epochs. Since Srivastava's research further suggests the placement of Dropout layers between the functional layers in a neural network, we placed our dropout layers in between each dense layer and the Conditional RNN layer. The last unique, unexplained piece of the RNN's structure is our activation layers. Typically, activation layers are placed at the end of each cluster of hidden layers to normalize the outputs so that they don't become artificially inflated. For this reason we placed an activation function at the end of each neural network layer (Dense, and Cond_RNN layers), though we eventually removed the activation function on one

[1] Srivastava, Nitish, et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." *Journal of Machine Learning Research*, vol. 15, June 2014, doi:https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf.

dense layer (which we explain in the RNN Hyperparameter Selection section). We chose to use a sigmoid activation function as opposed to a ReLU or TanH, as ours was a classification problem with outputs from 0-1, so we wanted our predictions to resemble that. Finally, in order to test and optimize hyperparameters for the RNN, we are using Stratified Cross-Validation as well as Stratified training and testing sets, as it is of vital importance that all sets have an equal number of death cases (which are quite rare in our dataset, and can sway results significantly).

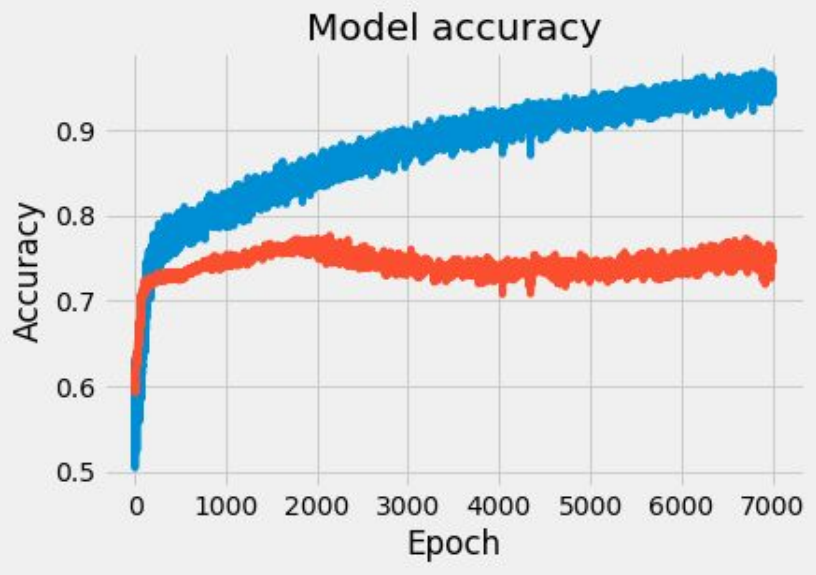
RNN Hyperparameter Selection:

In addition to layer placement and selection, multiple hyperparameters can have a significant impact on the RNN's efficacy. We identified the most essential of these hyperparameters be the following:

- The type of optimizer used for compilation (Stochastic Gradient Descent, RMSprop, Adagrad)
- The learning rate of the optimizer (Default, Custom: 0.1-0.5)
- The dropout rate (0.1, 0.2, 0.3, 0.4, 0.5)
- The number of LSTM and Dense nodes (16, 32, 64, 128, 256)
- The number of epochs

In optimal conditions, we would test all hyperparameter combinations (a total of 750 combinations) in a formalized manner, and use Stratified 5-fold cross validation to select the best performing combination (about 10 minutes of runtime per test). Unfortunately with our current computing power, performing such expansive tests would take over 100 hours of continuous runtime, and given our time and resource constraints, we thought it more prudent to narrow down the hyperparameters tested to a more manageable size. Consequently, we decided to stick

to the default learning rate of each optimizer (recommended by the Keras documentation), and chose a subset of the number of nodes we initially wanted to test. Furthermore, we tested the dropout rates whilst holding all other variables constant (at 32 Nodes, SGD optimizer, 180 epochs), prior to testing, as our research indicated that dropout rates were least likely to be codependent on the other variables, and decided to stick with 50% dropout throughout testing. Finally, in order to select our number of epochs for each combination, we ran each combination with a 20% testing set to either 500, 1000 or 2000 epochs, and picked the epoch with the highest localized validation accuracy before the model started overfitting and validation accuracy dropped. The figure below shows an example of this test for our Non-Conditional RNN, where the best validation accuracy occured at around epoch 1800.



This test allowed us to make an epoch variable selection for each model without having to re-run the model for every epoch we considered. With our hyperparameters narrowed down, we proceeded with our hyperparameter testing, with our three optimizers and their various node and epoch combinations, each with 5-fold cross validation. Our results (shown in the figure below),

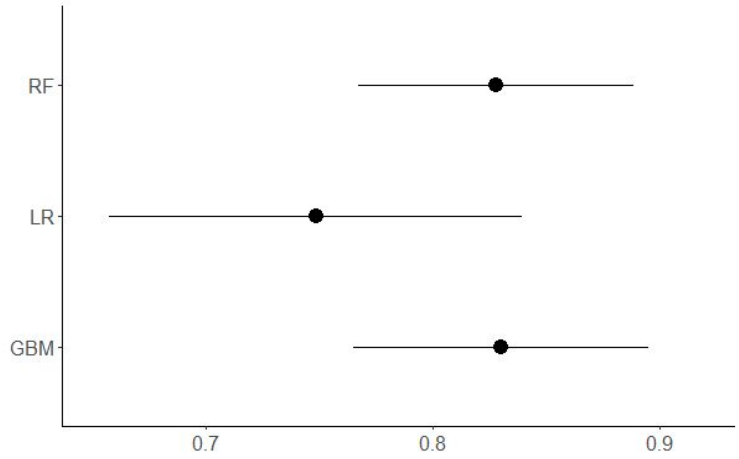
showed that our best combination was the Adagrad optimizer, with 32 LSTM and Dense nodes and 225 Epochs, which gave us a mean AUC 0.809311.

Optimizer	# of LSTM Nodes	# of Dense Nodes	# of Epochs	Mean AUC
RMSprop	32	32	25	0.804462
RMSprop	64	64	40	0.793586
SGD	16	16	459	0.789030
SGD	32	32	180	0.802440
SGD	64	64	200	0.808068
SGD	256	256	215	0.808052
Adagrad	**32**	**32**	**255**	**0.809411**
Adagrad	64	64	50	0.802720

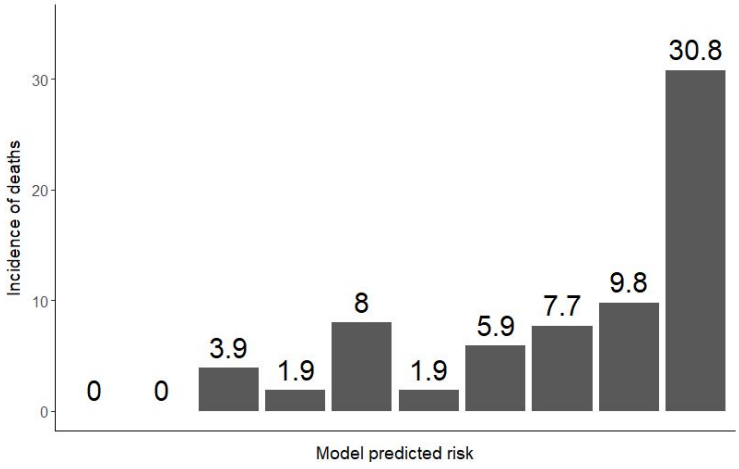
By setting our optimal hyperparameters, we gained a bit more flexibility to tweak the more minor layering variables to optimize the results under our given AUCs. In our trials, we found that removing the sigmoid activation from our time variant dense layer resulted in higher model AUCs (this change is already represented in our layers outline above). Our intuition for this is that because the outputs from the time-invariant dense layer are the inputs for the RNN layer, applying sigmoid activation to them could handicap the RNN's capacity to utilize them to their full potential.

Results

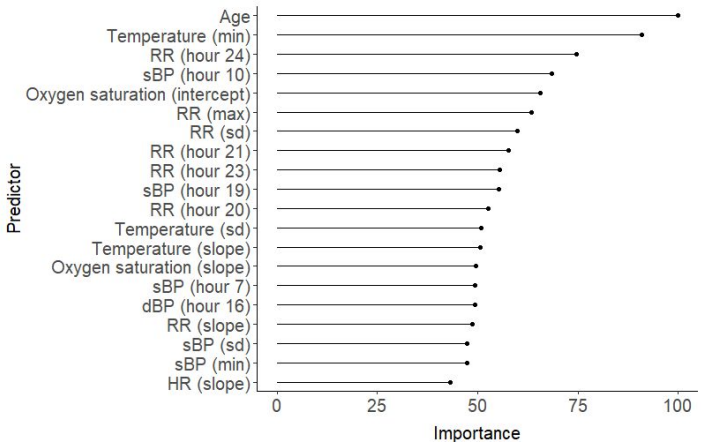
Traditional machine learning models



On cross validation, the random forest model had the best performance at mtry of 13. When the logistic regression, random forest and GBM models were applied to the testing dataset, performance did not differ significantly (95% confidence intervals all crossed each other). The logistic regression model had an AUC of 0.75 (0.66-0.84), the random forest model had an AUC of 0.83 (0.77-0.89) and GBM had an AUC of 0.83 (0.76-0.89).



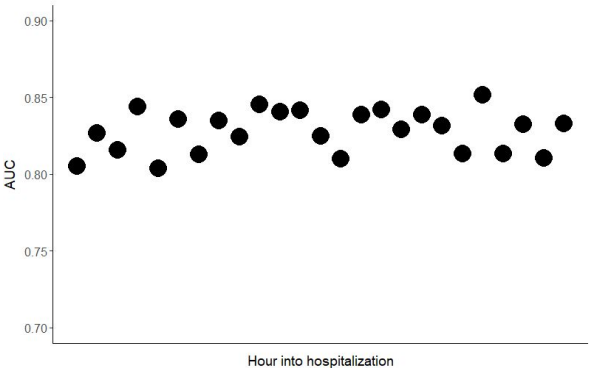
The calibration plot for the GBM model is shown above. The x-axis is the deciles of model predicted risk, and the y-axis is the actual incidence of deaths within these deciles. The model demonstrated good calibration, with increasing mortality rates across increasing deciles of risk. The bottom two deciles had 0 percent deaths, while the top decile had 30.8% mortality rate.

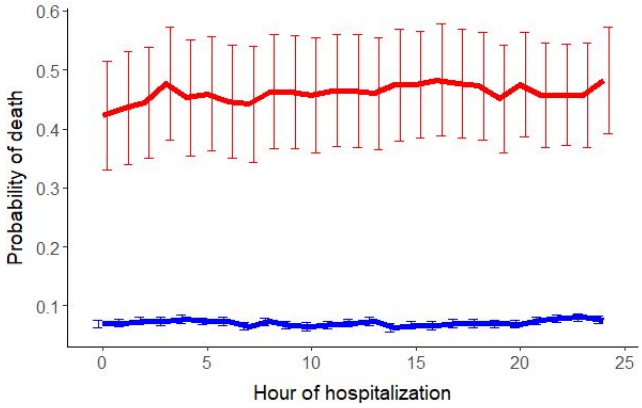


The variable importance plot demonstrates the top 20 most important variables. The most important feature in predicting mortality was age. This was followed by both extracted features of longitudinal vital signs (maximum, minimum, standard deviation), as well as specific vital signs at specific times. Respiratory rate seemed to be an important feature in the model.

Time-based prediction

The model performance at each hour varied, with no discernable trend. Importantly, there was not a linear improvement in AUC with each passing hour of hospitalization.

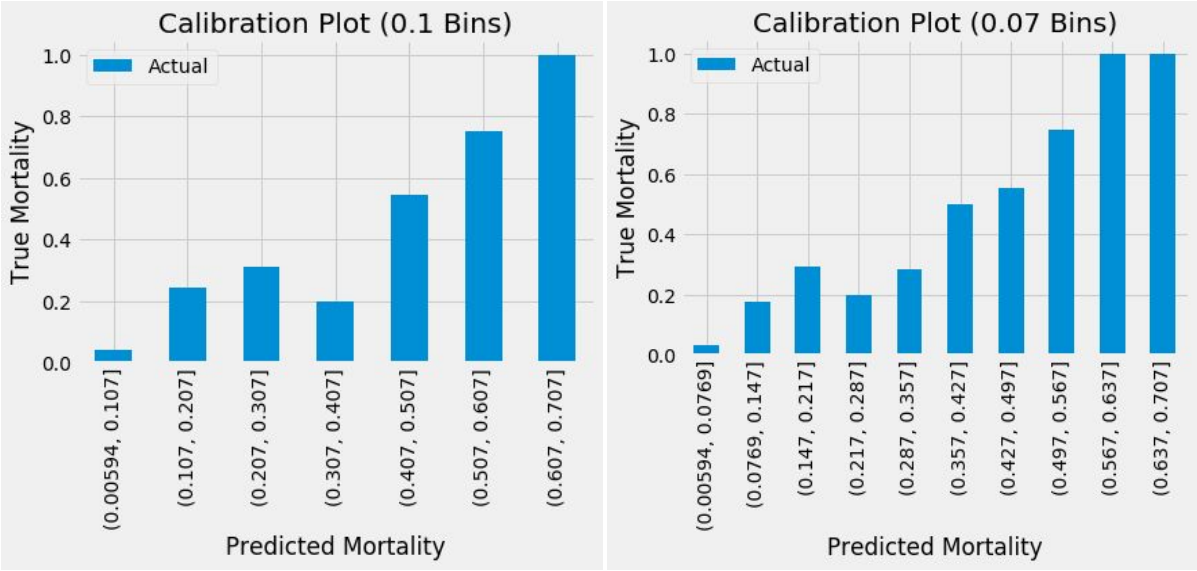




Above is the probability of death for survivors (in blue) and non-survivors (in red) at each time point based on the model developed for that time point. In general, both curves are fairly flat, suggesting that there was not an overall trend in the probability of death in survivors or non-survivors over the first 24 hours of hospitalization.

Conditional Recurrent Neural Network with Long Short-Term Units

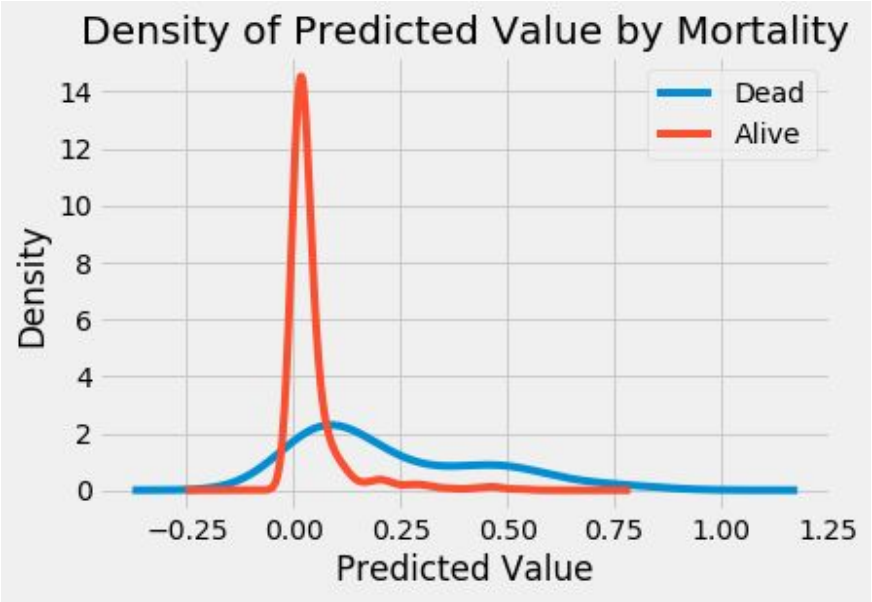
On stratified 5-fold cross-validation, our optimized RNN had an mean AUC of 0.8265 with the AUC range varying from 0.8063 and 0.8449. For the RNN that was trained on a stratified 80% segment of the data, and tested on a stratified 20% segment of the data, we had an AUC of 0.8741. Since our outputs in the RNN are neither binary nor full probabilities (i.e probabilities that sum up to 1), but simply relative probabilities (i.e. each output only gains meaning when compared to the other outputs), we used a calibration plot (instead of more traditional methods, like a confusion matrix) to demonstrate our results. Our calibration plots in figure below compare the actual death rate with the RNN's predicted likelihood. With the exception of the [0.307-0.407] bin in the 0.1 bin plot and the [0.147-0.217] bin in the 0.007 bin plot, our calibration plots show an absolutely increasing trend, demonstrating that our RNN's probabilities matched up well with the actual mortality rates.



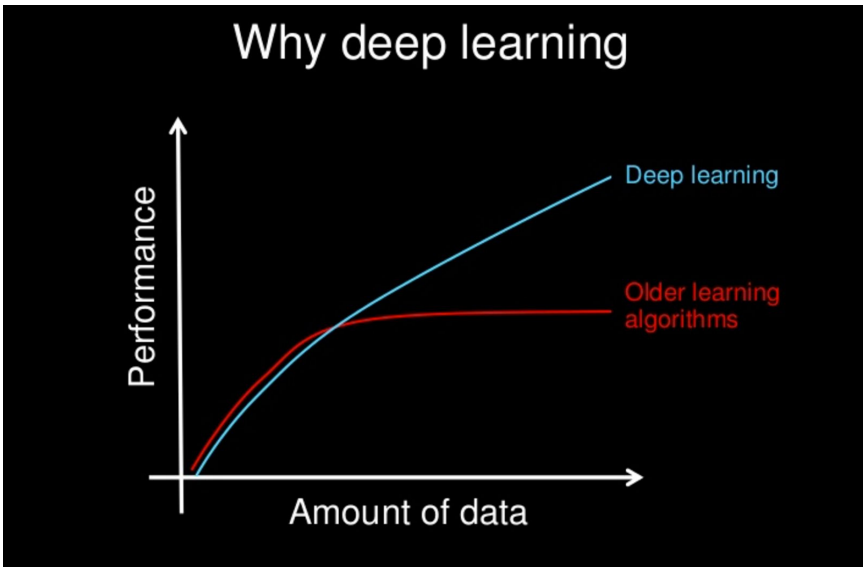
We also used our output data from the RNN to output a density plot comparing the RNN's predictions for dead vs. alive patients. While our two curves are not as distinct as would be optimal, there is still a clear separation between the RNN's predictions for dead and alive patients. Note that the disparate sizes of the two curves is due to the much higher number of alive patients in our dataset.

Discussion

In comparing our models, we find that the Random Forest performed best, with a 0.83 Mean AUC and a 0.77-0.89 range, followed by Gradient Boosting with a 0.83 Mean AUC and a 0.76-0.89 range. Interestingly, our RNN model−which was specifically engineered to take advantage of the longitudinal nature of our data−performed slightly worse, with a 0.8265 mean cross-validated AUC and a testing AUC of 0.8741, which is well within the confidence interval ranges for the Random Forest and Gradient Boosting models. In diagnosing why the model that



was theoretically more optimal for our data, performed worse, we propose three theories. One possibility is that we simply do not have enough data. Generally speaking, deep learning models like RNNs perform best when given large amounts of data, and could perform slightly worse that more traditional machine learning models with smaller amounts of data. The graph below comes from Andrew Ng, the Chief Scientist at Baidu Research, and founder of Google Brain, and shows how with lower amounts of data, deep learning might actually perform worse than older



deep learning algorithms, which perfectly explains our scenario. With this in mind, we suggest that if we were to run the same algorithms on a dataset with much more observations than our 2580 patients, our RNN model would start to outperform our Random Forest and Gradient Boosting models. Our second possible explanation for the RNN's lower performance is that the traditional models were simply more practical for this case than the RNN. As is often the case in Machine Learning, the more complex models are not always the most effective ones. In this case, there is no guarantee that taking advantage of the longitudinality in our data and applying a more complex model--like an RNN--will result in better results. In fact, the RNN's ability to take in more complexity may have caused it to notice smaller nuances in the training data and incorrectly classify them as predictors, where a less complex Random Forest or a Gradient Boosting model would not have done. Our third and final possible explanation for the RNN's lower performance is that our data simply had a limit to its generalizability. Though there were slight differences in performance between our best models, all their average AUCs seemed to be

around 0.83, which might suggest that that is the limit to which our data could be generalized for this use case.

Conclusion

To gain some perspective on how accurate our predictions truly are from a medical standpoint, we looked towards Dr. Mandrekar's research on the utilization of AUCs in Medical Diagnostic Tests[2]. His work classifes AUCs in the following manner: "an AUC of 0.5 suggests no discrimination...0.7 to 0.8 is considered acceptable, 0.8 to 0.9 is considered excellent, and more than 0.9 is considered outstanding". Following this categorization measure, we find that our best results, from the Random Forest, Gradient Boosting and RNN models, had AUCs that centered around 0.83 and 0.82, with confidence intervals from the high .70s to the high .80s, and would therefore be considered "excellent" predictors. In conclusion, we find that longitudinal vital signs data can effectively be generalized to determine 30-day mortalities for hospitalized patients at an "excellent" level with Random Forest, Gradient Boosting and RNN models. Furthermore, we suspect that with a larger dataset, under which deep learning models are more capable of thriving, our RNN model would be capable of outperforming the Random Forest and Gradient Boosting model, and perhaps attain an AUC in the "outstanding" range (0.9+).

Appendix

R Code for Traditional Machine Learning Algorithms and Time-based Predictions

#SETUP

[2] Mandrekar, Jayawant N. "Receiver Operating Characteristic Curve in Diagnostic Test Assessment." *Journal of Thoracic Oncology*, vol. 5, no. 9, 2010, pp. 1315–1316., doi:10.1097/jto.0b013e3181ec173d.

```{r setup, include=FALSE}

library(haven)

library(keras)

library(dplyr)

library(zoo)

library(reshape2)

library(readxl)

library(readr)

library(caret)

library(pROC)

library(ROCR)

library(ggplot2)

library(gtools)

library(BBmisc)

```

#Data Cleaning

```{r setup, include=FALSE}

rm(list = ls())

mortality <- read.csv("V:/Study - Temperature

Infection/TempTrajAnalysis/MLclass/ml_fixed.csv")
```

```
drop = c(17)
mortality = mortality[-(drop)]

###FEATURE CREATION
mortality$beta_0h = 0
mortality$beta_1h = 0
##FIXED EFFECTS
#FOR EACH PATIENT, RUN LINEAR REGRESSION OF HEART_RATE ~ TIME OF
ADMISSION. USE BETA COEFFICIENTS AS FEATURES FOR THAT PATIENT
for(r in 1:2580){
    r_1 = ((r-1)*25)+1
    r_2 = r_1 + 24
    names = c(r_1:r_2)
    temp = mortality[c(names),]
model = lm(heartrate ~ hour_of_admission, data=temp)
mortality[c(r_1:r_2),18] = model$coefficients[1]
mortality[c(r_1:r_2),19] = model$coefficients[2]
}

mortality$beta_0r = 0
mortality$beta_1r = 0
```

```
for(r in 1:2580){
    r_1 = ((r-1)*25)+1
    r_2 = r_1 + 24
    names = c(r_1:r_2)
    temp = mortality[c(names),]
model = lm(respiratoryrate ~ hour_of_admission, data=temp)
mortality[c(r_1:r_2),20] = model$coefficients[1]
mortality[c(r_1:r_2),21] = model$coefficients[2]
}

mortality$beta_0d = 0
mortality$beta_1d = 0
for(r in 1:2580){
    r_1 = ((r-1)*25)+1
    r_2 = r_1 + 24
    names = c(r_1:r_2)
    temp = mortality[c(names),]
model = lm(diastolic_bp ~ hour_of_admission, data=temp)
mortality[c(r_1:r_2),22] = model$coefficients[1]
mortality[c(r_1:r_2),23] = model$coefficients[2]
}
```

```
mortality$beta_0s = 0

mortality$beta_1s = 0

for(r in 1:2580){

    r_1 = ((r-1)*25)+1

    r_2 = r_1 + 24

    names = c(r_1:r_2)

    temp = mortality[c(names),]

model = lm(systolic_bp ~ hour_of_admission, data=temp)

mortality[c(r_1:r_2),24] = model$coefficients[1]

mortality[c(r_1:r_2),25] = model$coefficients[2]

}


mortality$beta_0x = 0

mortality$beta_1x = 0

for(r in 1:2580){

    r_1 = ((r-1)*25)+1

    r_2 = r_1 + 24

    names = c(r_1:r_2)

    temp = mortality[c(names),]

model = lm(oxygen_sat ~ hour_of_admission, data=temp)

mortality[c(r_1:r_2),26] = model$coefficients[1]

mortality[c(r_1:r_2),27] = model$coefficients[2]
```

```
}


mortality$beta_0t = 0

mortality$beta_1t = 0

for(r in 1:2580){

    r_1 = ((r-1)*25)+1

    r_2 = r_1 + 24

    names = c(r_1:r_2)

    temp = mortality[c(names),]

model = lm(temperature ~ hour_of_admission, data=temp)

mortality[c(r_1:r_2),28] = model$coefficients[1]

mortality[c(r_1:r_2),29] = model$coefficients[2]

}


for(r in 1:2580){

    r_1 = ((r-1)*25)+1

    r_2 = r_1 + 24

    names = c(r_1:r_2)

    temp = mortality[c(names),]

model = lm(temperature ~ hour_of_admission, data=temp)

mortality[c(r_1:r_2),28] = model$coefficients[1]

mortality[c(r_1:r_2),29] = model$coefficients[2]
```

```
}
mortality$beta_0t = 0
mortality$beta_1t = 0
for(r in 1:2580){
    r_1 = ((r-1)*25)+1
    r_2 = r_1 + 24
    names = c(r_1:r_2)
    temp = mortality[c(names),]
model = lm(temperature ~ hour_of_admission, data=temp)
mortality[c(r_1:r_2),28] = model$coefficients[1]
mortality[c(r_1:r_2),29] = model$coefficients[2]
}

#NORMALIZE FIXED EFFECTS
mortality$beta_0t = normalize(mortality$beta_0t, method = "range", range = c(0, 1), margin =
1L, on.constant = "quiet")
mortality$beta_1t = normalize(mortality$beta_1t, method = "range", range = c(0, 1), margin =
1L, on.constant = "quiet")
mortality$beta_0h = normalize(mortality$beta_0h, method = "range", range = c(0, 1), margin =
1L, on.constant = "quiet")
mortality$beta_1h = normalize(mortality$beta_1h, method = "range", range = c(0, 1), margin =
1L, on.constant = "quiet")
```

```
mortality$beta_0r = normalize(mortality$beta_0r, method = "range", range = c(0, 1), margin =
1L, on.constant = "quiet")
mortality$beta_1r = normalize(mortality$beta_1r, method = "range", range = c(0, 1), margin =
1L, on.constant = "quiet")
mortality$beta_0s = normalize(mortality$beta_0s, method = "range", range = c(0, 1), margin =
1L, on.constant = "quiet")
mortality$beta_1s = normalize(mortality$beta_1s, method = "range", range = c(0, 1), margin =
1L, on.constant = "quiet")
mortality$beta_0d = normalize(mortality$beta_0d, method = "range", range = c(0, 1), margin =
1L, on.constant = "quiet")
mortality$beta_1d = normalize(mortality$beta_1d, method = "range", range = c(0, 1), margin =
1L, on.constant = "quiet")
mortality$beta_0x = normalize(mortality$beta_0x, method = "range", range = c(0, 1), margin =
1L, on.constant = "quiet")
mortality$beta_1x = normalize(mortality$beta_1x, method = "range", range = c(0, 1), margin =
1L, on.constant = "quiet")



#MAKE MAX/MIN/SD FOR HR
maxhr = c()
minhr = c()
sdhr = c()
```

```
for(r in 1:2580){
    r_1 = ((r-1)*25)+1
    r_2 = r_1 + 24
    names = c(r_1:r_2)
    temp = mortality[c(names),11]
maxhr[r] = max(temp)
}
for(r in 1:2580){
    r_1 = ((r-1)*25)+1
    r_2 = r_1 + 24
    names = c(r_1:r_2)
    temp = mortality[c(names),11]
minhr[r] = min(temp)
}
for(r in 1:2580){
    r_1 = ((r-1)*25)+1
    r_2 = r_1 + 24
    names = c(r_1:r_2)
    temp = mortality[c(names),11]
sdhr[r] = sd(temp)
}
#MAKE MAX/MIN/SD FOR RR
```

```
maxrr = c()
minrr = c()
sdrr = c()
for(r in 1:2580){
    r_1 = ((r-1)*25)+1
    r_2 = r_1 + 24
    names = c(r_1:r_2)
    temp = mortality[c(names),12]
maxrr[r] = max(temp)
}
for(r in 1:2580){
    r_1 = ((r-1)*25)+1
    r_2 = r_1 + 24
    names = c(r_1:r_2)
    temp = mortality[c(names),12]
minrr[r] = min(temp)
}
for(r in 1:2580){
    r_1 = ((r-1)*25)+1
    r_2 = r_1 + 24
    names = c(r_1:r_2)
    temp = mortality[c(names),12]
```

```
sdrr[r] = sd(temp)
}
#MAKE MAX/MIN/SD FOR SBP
maxsbp = c()
minsbp = c()
sdsbp = c()
for(r in 1:2580){
    r_1 = ((r-1)*25)+1
    r_2 = r_1 + 24
    names = c(r_1:r_2)
    temp = mortality[c(names),13]
maxsbp[r] = max(temp)
}
for(r in 1:2580){
    r_1 = ((r-1)*25)+1
    r_2 = r_1 + 24
    names = c(r_1:r_2)
    temp = mortality[c(names),13]
minsbp[r] = min(temp)
}
for(r in 1:2580){
    r_1 = ((r-1)*25)+1
```

```
    r_2 = r_1 + 24
    names = c(r_1:r_2)
    temp = mortality[c(names),13]
sdsbp[r] = sd(temp)
}
#MAKE MAX/MIN/SD FOR DBP
maxdbp = c()
mindbp = c()
sddbp = c()
for(r in 1:2580){
    r_1 = ((r-1)*25)+1
    r_2 = r_1 + 24
    names = c(r_1:r_2)
    temp = mortality[c(names),14]
maxdbp[r] = max(temp)
}
for(r in 1:2580){
    r_1 = ((r-1)*25)+1
    r_2 = r_1 + 24
    names = c(r_1:r_2)
    temp = mortality[c(names),14]
mindbp[r] = min(temp)
```

```
}
for(r in 1:2580){
    r_1 = ((r-1)*25)+1
    r_2 = r_1 + 24
    names = c(r_1:r_2)
    temp = mortality[c(names),14]
sddbp[r] = sd(temp)
}
#MAKE MAX/MIN/SD FOR TEMP
maxtemp = c()
mintemp = c()
sdtemp = c()
for(r in 1:2580){
    r_1 = ((r-1)*25)+1
    r_2 = r_1 + 24
    names = c(r_1:r_2)
    temp = mortality[c(names),15]
maxtemp[r] = max(temp)
}
for(r in 1:2580){
    r_1 = ((r-1)*25)+1
    r_2 = r_1 + 24
```

```
    names = c(r_1:r_2)
    temp = mortality[c(names),15]
mintemp[r] = min(temp)
}
for(r in 1:2580){
    r_1 = ((r-1)*25)+1
    r_2 = r_1 + 24
    names = c(r_1:r_2)
    temp = mortality[c(names),15]
sdtemp[r] = sd(temp)
}
#MAKE MAX/MIN/SD FOR O2SAT
maxo2sat = c()
mino2sat = c()
sdo2sat = c()
for(r in 1:2580){
    r_1 = ((r-1)*25)+1
    r_2 = r_1 + 24
    names = c(r_1:r_2)
    temp = mortality[c(names),14]
maxo2sat[r] = max(temp)
}
```

```
for(r in 1:2580){
    r_1 = ((r-1)*25)+1
    r_2 = r_1 + 24
    names = c(r_1:r_2)
    temp = mortality[c(names),14]
mino2sat[r] = min(temp)
}
for(r in 1:2580){
    r_1 = ((r-1)*25)+1
    r_2 = r_1 + 24
    names = c(r_1:r_2)
    temp = mortality[c(names),14]
sdo2sat[r] = sd(temp)
}
```

```
########## RESHAPE WIDE#######################################
#RESHAPE WIDE - Age + 7 diseases + 2FE*6variables + 3features*6variables + 6*25
changing variables + Mortality = 189
new = matrix(0, ncol=189, nrow=2580)
```

```
new = as.data.frame(new)

numbers = c(0:24)

name = rep(c("hr"), times=25)

hrnames = paste(name,numbers, sep="_", collapse=NULL)

name = rep(c("rr"), times=25)

rrnames = paste(name,numbers, sep="_", collapse=NULL)

name = rep(c("systolic"), times=25)

systolicnames = paste(name,numbers, sep="_", collapse=NULL)

name = rep(c("diastolic"), times=25)

diastolicnames = paste(name,numbers, sep="_", collapse=NULL)

name = rep(c("temperature"), times=25)

temperaturenames = paste(name,numbers, sep="_", collapse=NULL)

name = rep(c("oxygen"), times=25)

oxygennames = paste(name,numbers, sep="_", collapse=NULL)

vitals = c("hrmax", "hrmin", "hrsd", "rrmax", "rrmin", "rrsd","smax", "smin", "ssd","dmax",

"dmin", "dsd","tmax", "tmin", "tsd","omax", "omin", "osd")

columns = colnames(mortality[3:10])

columns2 = colnames(mortality[17:29])


columns = c(columns, columns2)

rm(columns2)
```

```
columns2 = c(columns, hrnames, rrnames, systolicnames, diastolicnames, temperaturenames,
oxygennames, vitals)
colnames(new)= columns2
#HR WIDE
for (c in 0:24){
for(r in 1:2580){
     r_1 = ((r-1)*25)+1
   new[r,22+c] = mortality[r_1+c,11]
}
}
#RR WIDE
for (c in 0:24){
for(r in 1:2580){
     r_1 = ((r-1)*25)+1
   new[r,47+c] = mortality[r_1+c,12]
}
}

#SBP WIDE
for (c in 0:24){
for(r in 1:2580){
     r_1 = ((r-1)*25)+1
```

```
    new[r,72+c] = mortality[r_1+c,13]
}
}
#DBP WIDE
for (c in 0:24){
for(r in 1:2580){
      r_1 = ((r-1)*25)+1
    new[r,97+c] = mortality[r_1+c,14]
}
}
#TEMP WIDE
for (c in 0:24){
for(r in 1:2580){
      r_1 = ((r-1)*25)+1
    new[r,122+c] = mortality[r_1+c,15]
}
}
#O2 WIDE
for (c in 0:24){
for(r in 1:2580){
      r_1 = ((r-1)*25)+1
    new[r,147+c] = mortality[r_1+c,16]
```

```
}
}
#FIXED VARIABLES
#AGE TO DIABETES
for(c in 1:8){
for(r in 1:2580){
     r_1 = ((r-1)*25)+1
   new[r,c] = mortality[r_1,c+2]
}
}
#FIXED EFFECT REGRESSION COEFFS
for(c in 10:21){
for(r in 1:2580){
     r_1 = ((r-1)*25)+1
   new[r,c] = mortality[r_1,c+8]
}
}
#MORTALITY
for(r in 1:2580){
     r_1 = ((r-1)*25)+1
   new[r,9] = mortality[r_1,17]
}
```

#MAX MIN SD

new$hrmax=maxhr

new$hrmin=minhr

new$hrsd=sdhr

new$rrmax=maxrr

new$rrmin=minrr

new$rrsd=sdrr

new$smax=maxsbp

new$smin=minsbp

new$ssd=sdsbp

new$dmax=maxdbp

new$dmin=mindbp

new$dsd=sddbp

new$tmax=maxtemp

new$tmin=mintemp

new$tsd=sdtemp

new$omax=maxo2sat

new$omin=mino2sat

new$osd=sdo2sat

```
combined = new

colnames(combined)[9] = "mortality"

write.csv(combined, "mortality_wide.csv")
```

#A) Model training on all features

```{r setup, include=FALSE}
################ USING ALL FEATURES ############

rm(list=ls())

combined = read.csv("mortality_wide.csv")

combined = combined[-(1)]

#Combined is all features

################ PARTITION WITH CARET     SPLITTING DATASETS RANDOM

set.seed(122)

trainIndex <- createDataPartition(combined$mortality, p = .8,

                                  list = FALSE,

                                  times = 1)

trainData <- combined[ trainIndex,]

testData  <- combined[-trainIndex,]



trainData$mortality = as.factor(trainData$mortality)
```

```
testData$mortality = as.factor(testData$mortality)

levels(trainData$mortality) = c("Alive", "Dead")

levels(testData$mortality) = c("Alive", "Dead")

write.csv(testData, "test.csv")

write.csv(trainData, "train.csv")


###############       RANDOM FOREST                              ###################

set.seed(122)

#Cross validation settings

cvCtrl <- trainControl(method = "repeatedcv"

                       , number = 10

                       , repeats = 1

                       , classProbs = TRUE

                       , summaryFunction = twoClassSummary

                       , allowParallel = TRUE

)




rfGrid<-expand.grid(mtry = c(13,14,15))

t1 <- Sys.time()

RF<- train(mortality~.
```

```
                , method = "rf"
                , data = trainData
                , metric = "ROC"
                , trControl = cvCtrl
                , tuneGrid = rfGrid
                , na.action = na.pass
                , tree_method = 'hist'
                , grow_policy = "lossguide"
)
t2 <- Sys.time()
difftime(t2,t1)
save(RF, file="randomforest.rda")
RFprobs<-predict(RF, newdata=testData, type="prob", na.action = na.pass)
RFprobsDF<-as.data.frame(RFprobs)
RF_Roc<-roc(testData$mortality, (RFprobsDF$Dead))
RF_Roc
testData$prediction = (RFprobsDF$Dead)
testData=transform(testData,prediction=ifelse(prediction>=0.5,1,0))
testData$prediction = as.factor(testData$prediction)
levels(testData$prediction) <- c("Alive", "Dead")
confusionMatrix(reference=testData$mortality, data=testData$prediction, mode='everything',
positive='Dead')
```

```
################       LOGISTIC REGRESSION                    ###################

#Cross validation settings

cvCtrl <- trainControl(method = "repeatedcv"

                       , number = 10

                       , repeats = 1

                       , classProbs = TRUE

                       , summaryFunction = twoClassSummary

                       , allowParallel = TRUE

)

t1 <- Sys.time()

LM<- train(mortality~.

                  , method = "glm"

                  , data = trainData

                  , metric = "ROC"

                  , trControl = cvCtrl

                  , na.action = na.pass

)

t2 <- Sys.time()

difftime(t2,t1)

save(LM, file="logistic.rda")

LM_Probs<-predict(LM, newdata=testData, type="prob", na.action = na.pass)
```

```
LM_ProbsDF<-as.data.frame(LM_Probs)

LM_Roc<-roc(testData$mortality, (LM_ProbsDF$Dead))

LM_Roc

testData$prediction = (LM_ProbsDF$Dead)

testData=transform(testData,prediction=ifelse(prediction>=0.5,1,0))

testData$prediction = as.factor(testData$prediction)

levels(testData$prediction) <- c("Alive", "Dead")

confusionMatrix(reference=testData$mortality, data=testData$prediction, mode='everything',

positive='Dead')


################       XGBOOST                                  ####################

#Cross validation settings

cvCtrl <- trainControl(method = "repeatedcv"

                       , number = 10

                       , repeats = 1

                       , classProbs = TRUE

                       , summaryFunction = twoClassSummary

                       , allowParallel = TRUE

)
```

```
xgbGrid<-expand.grid(nrounds=c(2000)
                     , max_depth=c(20)
                     , eta=c(0.005)
                     , gamma = c(0)
                     , colsample_bytree = c(1)
                     , min_child_weight = c(2)
                     , subsample = c(0.5)
)
set.seed(122)
t1 <- Sys.time()
gbm<- train(mortality~.
                     , method = "xgbTree"
                     , data = trainData
                     , metric = "ROC"
                     , trControl = cvCtrl
                     , tuneGrid = xgbGrid
                     , na.action = na.pass
                     , tree_method = 'hist'
                     , grow_policy = "lossguide"
)
t2 <- Sys.time()
difftime(t2,t1)
```

gbm

save(gbm, file="xgboost.rda")

xgbGridProbs<-predict(gbm, newdata=testData, type="prob", na.action = na.pass)

xgbGridProbsDF<-as.data.frame(xgbGridProbs)

xgbGridRoc<-roc(testData$mortality, (xgbGridProbsDF$Dead))

xgbGridRoc

```

#Tables for storage of AUC information

```{r setup, include=FALSE}

###############      AUC TABLE                                  ###################

auc = matrix("", ncol=3, nrow=3)

new = as.data.frame(auc)

colnames(auc) = c("LR", "RF", "GBM")

rownames(auc) = c("Feature1", "Feature2", "Feature3")

LM_ci = ci.auc(LM_Roc)

LM_ci = round(LM_ci, digits=2)

auc[1,1] = paste(LM_ci[2], " (", LM_ci[1], "-", LM_ci[3], ")", sep = "")

LM_ci = ci.auc(LM_Roc_B)

LM_ci = round(LM_ci, digits=2)

auc[2,1] = paste(LM_ci[2], " (", LM_ci[1], "-", LM_ci[3], ")", sep = "")

LM_ci = ci.auc(LM_Roc_C)

LM_ci = round(LM_ci, digits=2)

```
auc[3,1] = paste(LM_ci[2], " (", LM_ci[1], "-", LM_ci[3], ")", sep = "")

LM_ci = ci.auc(RF_Roc)

LM_ci = round(LM_ci, digits=2)

auc[1,2] = paste(LM_ci[2], " (", LM_ci[1], "-", LM_ci[3], ")", sep = "")

LM_ci = ci.auc(RF_Roc_B)

LM_ci = round(LM_ci, digits=2)

auc[2,2] = paste(LM_ci[2], " (", LM_ci[1], "-", LM_ci[3], ")", sep = "")

LM_ci = ci.auc(RF_Roc_C)

LM_ci = round(LM_ci, digits=2)

auc[3,2] = paste(LM_ci[2], " (", LM_ci[1], "-", LM_ci[3], ")", sep = "")

LM_ci = ci.auc(xgbGridRoc)

LM_ci = round(LM_ci, digits=2)

auc[1,3] = paste(LM_ci[2], " (", LM_ci[1], "-", LM_ci[3], ")", sep = "")

LM_ci = ci.auc(xgbGridRoc_B)

LM_ci = round(LM_ci, digits=2)

auc[2,3] = paste(LM_ci[2], " (", LM_ci[1], "-", LM_ci[3], ")", sep = "")

LM_ci = ci.auc(xgbGridRoc_C)

LM_ci = round(LM_ci, digits=2)

auc[3,3] = paste(LM_ci[2], " (", LM_ci[1], "-", LM_ci[3], ")", sep = "")

write.csv(auc, "auc.csv")
```

```
###############       AUC FIGURES                                  ###################

auc_figure = data.frame(model = character(length = 9),feature_set = character(length = 9), auc =

numeric(length = 9), min = numeric(length = 9), max = numeric(length = 9))

names = rep(c("LR", "RF", "GBM"), times=3)

auc_figure$model = names

names = rep(c("1", "2", "3"), each=3)

auc_figure$feature_set = names

auc_figure$auc[1] = ci.auc(LM_Roc)[2]

auc_figure$auc[2] = ci.auc(RF_Roc)[2]

auc_figure$auc[3] = ci.auc(xgbGridRoc)[2]

auc_figure$auc[4] = ci.auc(LM_Roc_B)[2]

auc_figure$auc[5] = ci.auc(RF_Roc_B)[2]

auc_figure$auc[6] = ci.auc(xgbGridRoc_B)[2]

auc_figure$auc[7] = ci.auc(LM_Roc_C)[2]

auc_figure$auc[8] = ci.auc(RF_Roc_C)[2]

auc_figure$auc[9] = ci.auc(xgbGridRoc_C)[2]


auc_figure$min[1] = ci.auc(LM_Roc)[1]

auc_figure$min[2] = ci.auc(RF_Roc)[1]

auc_figure$min[3] = ci.auc(xgbGridRoc)[1]

auc_figure$min[4] = ci.auc(LM_Roc_B)[1]

auc_figure$min[5] = ci.auc(RF_Roc_B)[1]
```

```
auc_figure$min[6] = ci.auc(xgbGridRoc_B)[1]

auc_figure$min[7] = ci.auc(LM_Roc_C)[1]

auc_figure$min[8] = ci.auc(RF_Roc_C)[1]

auc_figure$min[9] = ci.auc(xgbGridRoc_C)[1]

auc_figure$max[1] = ci.auc(LM_Roc)[3]

auc_figure$max[2] = ci.auc(RF_Roc)[3]

auc_figure$max[3] = ci.auc(xgbGridRoc)[3]

auc_figure$max[4] = ci.auc(LM_Roc_B)[3]

auc_figure$max[5] = ci.auc(RF_Roc_B)[3]

auc_figure$max[6] = ci.auc(xgbGridRoc_B)[3]

auc_figure$max[7] = ci.auc(LM_Roc_C)[3]

auc_figure$max[8] = ci.auc(RF_Roc_C)[3]

auc_figure$max[9] = ci.auc(xgbGridRoc_C)[3]


 ggplot(auc_figure, aes(x=model, y=auc, color=feature_set)) +

  geom_point(size = 4) +

    geom_errorbar(aes(ymin=auc_figure$min, ymax=auc_figure$max), width=0,

                  position=position_dodge(0)) + theme_classic() + xlab("Model") +

scale_y_continuous(name="AUC", limits=c(0.5,0.95))+ theme(axis.text =

element_text(size=12),

                                                                                axis.title.x =

element_text(size = 14, face="bold",vjust=-1),
```

axis.title.y =

element_text(size = 14, face="bold", vjust=2.8)) + coord_flip()

write.csv(auc_figure, "auc_figure.csv")

```

#Creating AUC Figures and Tables

```{r setup, include=FALSE}

################################################################################

#############

###############       AUC TABLE                              ###################

################################################################################

#############

auc <- read.csv("Y:/Study - Temperature Infection/TempTrajAnalysis/MLclass/auc.csv")

################################################################################

#############

###############       AUC FIGURES                            ###################

################################################################################

#############

auc_figure <- read.csv("Y:/Study - Temperature

Infection/TempTrajAnalysis/MLclass/auc_figure.csv")

```
auc_figure$feature_set=as.factor(auc_figure$feature_set)
auc_figure=auc_figure[which(auc_figure$feature_set=="1"),]
 ggplot(auc_figure, aes(x=model, y=auc, color=feature_set)) +
  geom_point(size = 4) +
    geom_errorbar(aes(ymin=auc_figure$min, ymax=auc_figure$max), width=0,
                  position=position_dodge(0)) + theme_classic() + xlab("Model") +
scale_y_continuous(name="AUC", limits=c(0.65,0.92))+ theme(axis.text =
element_text(size=12),
                                                                    axis.title.x =
element_blank(),
                                                                    axis.title.y =
element_blank(), legend.position = "none") + coord_flip()
```

#VARIABLE IMPORTANCE PLOTS

```
```{r setup, include=FALSE}
combined = read.csv("Y:/Study - Temperature
Infection/TempTrajAnalysis/MLclass/mortality_wide.csv")
combined = combined[-(1)]
```
```

```
set.seed(122)
trainIndex <- createDataPartition(combined$mortality, p = .8,
                                  list = FALSE,
                                  times = 1)
trainData <- combined[ trainIndex,]
testData  <- combined[-trainIndex,]
#READ IN TEST AND TRAIN DATA SETS
testData = read.csv("Y:/Study - Temperature Infection/TempTrajAnalysis/MLclass/test.csv")
trainData = read.csv("Y:/Study - Temperature Infection/TempTrajAnalysis/MLclass/train.csv")
load("Y:/Study - Temperature Infection/TempTrajAnalysis/MLclass/xgboost.rda")
gbm
imp = varImp(gbm)
imp = as.data.frame(imp$importance)
Feature = rownames(imp)
rownames(imp)=NULL
imp = cbind(Feature,imp)

imp = imp[1:20,]
names = c("Age", "Temperature (min)", "RR (hour 24)", "sBP (hour 10)", "Oxygen saturation
(intercept)", "RR (max)", "RR (sd)", "RR (hour 21)", "RR (hour 23)", "sBP (hour 19)", "RR
(hour 20)", "Temperature (sd)", "Temperature (slope)", "Oxygen saturation (slope)", "sBP (hour
7)", "dBP (hour 16)", "RR (slope)", "sBP (sd)", "sBP (min)", "HR (slope)")
```

```
imp$Feature = names

g <- ggplot(imp, aes(x=reorder(Feature,Overall), y=Overall))

g + geom_point() + geom_segment(aes(x=Feature,xend=Feature,y=0,yend=Overall))  +

xlab('Predictor') + ylab('Importance') + coord_flip() + theme_classic() +

scale_x_discrete(expand=c(0.025,0.00001)) + theme(axis.title.y = element_text(size=15),

axis.title.x = element_text(size=15,vjust=-1), axis.text.x = element_text(size=15), axis.text.y =

element_text(size=15))
```

#CALIBRATION plot

```{r setup, include=FALSE}
combined = read.csv("Y:/Study - Temperature

Infection/TempTrajAnalysis/MLclass/mortality_wide.csv")

combined = combined[-(1)]

set.seed(122)

trainIndex <- createDataPartition(combined$mortality, p = .8,

                                  list = FALSE,

                                  times = 1)

trainData <- combined[ trainIndex,]

testData  <- combined[-trainIndex,]

#READ IN TEST AND TRAIN DATA SETS

testData = read.csv("Y:/Study - Temperature Infection/TempTrajAnalysis/MLclass/test.csv")

trainData = read.csv("Y:/Study - Temperature Infection/TempTrajAnalysis/MLclass/train.csv")
```

```
load("Y:/Study - Temperature Infection/TempTrajAnalysis/MLclass/xgboost.rda")

gbm

testModel<-predict(gbm, newdata=testData, type="prob", na.action = na.pass)

testModel<-as.data.frame(testModel)

testData$prediction=quantcut(testModel$Dead, q=10)

levels(testData$prediction) = c(1:10)

testData$mortality=as.numeric(testData$mortality)

testData$mortality=testData$mortality-1

agg = aggregate(mortality ~ prediction, testData, mean)

agg$mortality= agg$mortality*100

agg$mortality=round(agg$mortality, digits=1)

ggplot(data = agg, aes(x = prediction, y = mortality))+geom_bar(stat =

"identity")+geom_text(label=agg$mortality, vjust=-0.5, size=8) + theme_classic() + xlab("Model

predicted risk") + ylab("Incidence of deaths") + theme(axis.text = element_text(size=12),

                                                                                axis.title.x =

element_text(size = 14, vjust=-1), axis.text.x = element_blank(),axis.ticks.x = element_blank(),

                                                                                axis.title.y =

element_text(size = 14,  vjust=2.8), legend.position = "none") +

scale_y_continuous(limits=c(0,35))
```

#Time based predictive modeling

```{r setup, include=FALSE}

###############       TIME LEVEL PREDICTION MODEL                    #############

rm(list = ls())

mortality <- read.csv("C:/Users/sbhavani/Desktop/ml_fixed.csv")

drop = c(17)

mortality = mortality[-(drop)]

colnames(mortality)[17] = "mortality"

set.seed(122)

n = c()

n = sample(1:10, 2580, replace=T)

#set.seed(122)

#trainIndex <- createDataPartition(mortality$mortality, p = .8,

 #                         list = FALSE,

  #                        times = 1)


mortality$n = 0

for(r in 1:2580){

   r_1 = ((r-1)*25)+1

   r_2 = r_1+24

   mortality[r_1:r_2,18] = n[r]

}
```

```
trainData <- mortality[mortality$n>=5,]
testData  <- mortality[mortality$n<5,]
trainData$mortality = as.factor(trainData$mortality)
testData$mortality = as.factor(testData$mortality)
levels(trainData$mortality) = c("Alive", "Dead")
levels(testData$mortality) = c("Alive", "Dead")
trainData <- trainData[order(trainData$hour_of_admission),]
testData <- testData[order(testData$hour_of_admission),]
cvCtrl <- trainControl(method = "repeatedcv"
                       , number = 10
                       , repeats = 1
                       , classProbs = TRUE
                       , summaryFunction = twoClassSummary
                       , allowParallel = TRUE
)
rfGrid<-expand.grid(mtry = c(14))
for(i in 0:24){
df = trainData[which(trainData$hour_of_admission==i),]
xgbGridmod<- train(mortality~.
                   , method = "rf"
                   , data = df
                   , metric = "ROC"
```

```
                , tuneGrid = rfGrid
                , trControl = cvCtrl
                , na.action = na.pass
                , tree_method = 'hist'
                , grow_policy = "lossguide"
)
name = (paste("hour", i, sep = ""))
assign(name, xgbGridmod)
}

#WILL GIVE YOU THE ACCURACY OF THE MODEL OVER 24 HOURS OF DATA
a= c()
for(i in 0:24){
df = testData[which(testData$hour_of_admission==i),]
name = (paste("hour", i, sep = ""))
model = get(name)
xgbGridProbs<-predict(model, newdata=df, type="prob", na.action = na.pass)
xgbGridProbsDF<-as.data.frame(xgbGridProbs)
xgbGridRoc<-roc(df$mortality, (xgbGridProbsDF$Dead))
a[i+1] = ci.auc(xgbGridRoc)[2]
}
```

```
a = as.data.frame(a)
a$hour = c(0:24)
ggplot(a, aes(x=hour, y=a)) + geom_point(size=8) + theme_classic() + xlab("Hour into
hospitalization") + ylab("AUC") + theme(axis.text = element_text(size=12),
                                                                                    axis.title.x =
element_text(size = 14, vjust=-1), axis.text.x = element_blank(),axis.ticks.x = element_blank(),
                                                                                    axis.title.y =
element_text(size = 14,  vjust=2.8), legend.position = "none") +
scale_y_continuous(limits=c(0.7,0.9))



#PREDICT HOURLY PROBABILITY ON TEST SET
timeprobability = matrix(0, ncol=26, nrow=1001)
timeprobability = as.data.frame(timeprobability)
for(i in 0:24){
df = testData[which(testData$hour_of_admission==i),]
name = (paste("hour", i, sep = ""))
model = get(name)
xgbGridProbs<-predict(model, newdata=df, type="prob", na.action = na.pass)
xgbGridProbsDF<-as.data.frame(xgbGridProbs)
timeprobability[i+1] = (xgbGridProbsDF$Dead)
}
```

```
colnames(timeprobability)[26] = "mortality"
for(r in 1:1001){
     r_1 = ((r-1)*25)+1
   timeprobability[r,26] = testData[r,17]
}

timeprobability$mortality = as.factor(timeprobability$mortality)
levels(timeprobability$mortality) = c("Alive", "Dead")

write.csv(timeprobability, "auc_over_time.csv")

timeprobability = read.csv("auc_over_time.csv")
timeprobability = timeprobability[-c(1)]

#CREATE HOURLY MEAN FOR PROBABILITY SCORES FOR ALIVE AND DEAD
PATIENTS
hourly = matrix(0, ncol=5, nrow=50)
hourly = as.data.frame(hourly)
colnames(hourly) = c("hour", "probability", "mortality", "sd", "total")
hourly$hour =c(0:24,0:24)
```

```
hourly$mortality = rep(c("Alive","Dead"),each=25)
for (x in 1:25) {
hourly$probability[x]=  mean(timeprobability[,x][which(timeprobability$mortality=="Alive")])
}
for (x in 1:25) {
hourly$sd[x]=  sd(timeprobability[,x][which(timeprobability$mortality=="Alive")])
}
for (x in 1:25) {
hourly$total[x]=  sum(timeprobability$mortality=="Alive")
}
for (x in 1:25) {
hourly$probability[x+25]=
mean(timeprobability[,x][which(timeprobability$mortality=="Dead")])
}
for (x in 1:25) {
hourly$sd[x+25]=  sd(timeprobability[,x][which(timeprobability$mortality=="Dead")])
}
for (x in 1:25) {
hourly$total[x+25]=  sum(timeprobability$mortality=="Dead")
}
hourly$upper = hourly$probability + ((1.96 * hourly$sd)/(hourly$total^0.5))
hourly$lower = hourly$probability - ((1.96 * hourly$sd)/(hourly$total^0.5))
```

```
limits <- aes(ymax=upper, ymin=lower)
dodge <- position_dodge(width=0.9)

ggplot(data = hourly, aes(x = hour, y = probability, group = mortality, color = mortality)) +
geom_line(size=2) + theme_classic() + xlab("Hour of hospitalization") + ylab("Probability of
death") + theme(axis.text = element_text(size=12),
                                                                    axis.title.x =
element_text(size = 14, vjust=-1),
                                                                    axis.title.y =
element_text(size = 14,  vjust=2.8), legend.position = "none") + scale_colour_manual(values =
c("blue", "red")) +
  geom_errorbar(limits, position=dodge)
```

Python Code for Initial Data Cleaning

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:



import numpy as np
```

```
from datascience import *
import matplotlib
get_ipython().run_line_magic('matplotlib', 'inline')
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')

import pandas as pd
pd.set_option('display.max_rows', 500)
pd.set_option('display.max_columns', 500)
pd.set_option('display.width', 1000)
import scipy as sc
#!pip install seaborn
import seaborn as sns
#!pip install sklearn
import sklearn as sk
from sklearn import preprocessing
plt.rcParams["figure.figsize"] = (10,5)



# In[2]:
```

```
mortality=pd.read_stata("mortality.dta")
```

```
# In[3]:
```

```
mortality.isnull().sum()
```

```
# In[9]:
```

```
mortality.groupby("patient").count()
```

```
# In[12]:
```

```
mortality=mortality.astype(float)
```

```
# In[14]:
```

```
addrows=mortality.copy()
for p in mortality.patient.unique():
    for i in np.arange(0,25):
        if len(mortality[mortality["patient"]==p][mortality["hour_of_admission"]==i]) <1:
            addrows=addrows.append(pd.DataFrame([[p,i,np.nan, np.nan, np.nan, np.nan, np.nan,
np.nan, np.nan, np.nan, np.nan, np.nan, np.nan, np.nan, np.nan, np.nan, np.nan, np.nan]],
columns=mortality.columns)).reset_index().drop(columns="index")
            print(p,i)
addrows=addrows.sort_values(["patient",
"hour_of_admission"]).reset_index().drop(columns="index")


# In[25]:


fill=pd.DataFrame(columns=addrows.columns)
for i in addrows.patient.unique():
    fill=pd.concat([fill,
addrows[addrows["patient"]==i].fillna(method="ffill").fillna(method="bfill")], axis=0)
fill.isnull().sum()
```

In[34]:

```
standard_scaler = preprocessing.StandardScaler()
np_scaled = standard_scaler.fit_transform(fill.drop(columns=["days_in_hospital", "patient",
"hour_of_admission", "mortality_30d"]))
normalized = pd.DataFrame(np_scaled, columns = fill.drop(columns=["days_in_hospital",
"patient", "hour_of_admission", "mortality_30d"]).columns)
normalized.insert(0, "patient", fill.patient)
normalized.insert(1, "hour_of_admission", fill.hour_of_admission)
normalized.insert(16, "days_in_hospital", fill.days_in_hospital)
normalized.insert(17, "mortality_30d", fill.mortality_30d)
normalized
```

In[35]:

```
normalized.groupby("patient").count().reset_index()
```

In[36]:

```
dropped=normalized.drop(np.concatenate((np.array(normalized.temperature.isnull()[normalized.
temperature.isnull()==True].index),
np.array(normalized.oxygen_sat.isnull()[normalized.oxygen_sat.isnull()==True].index)),
axis=None))
```

In[37]:

```
dropped["patient"]=np.repeat(np.arange(1,2581),25)
```

In[38]:

```
dropped.to_csv("ml_standardized.csv", index=False)
```

Python Code for Conditional RNN:

```
#!/usr/bin/env python

# coding: utf-8


# In[1]:



get_ipython().system('virtualenv --system-site-packages -p python3 ./venv')



# In[2]:



get_ipython().system('.\\venv\\Scripts\\activate')



# In[167]:



import numpy as np

import pandas as pd

import matplotlib

get_ipython().run_line_magic('matplotlib', 'inline')
```

```
import matplotlib.pyplot as plots
plots.style.use('fivethirtyeight')
from tensorflow.keras.models import *
from tensorflow.keras.layers import *
import tensorflow as tf
from sklearn.model_selection import *
from sklearn import *
import sklearn as sk
from cond_rnn import ConditionalRNN
import seaborn as sns



# In[2]:



mortality=pd.read_csv("ml_standardized.csv")
mortality



# In[3]:
```

```
#Training and Testing Splits
np.random.seed(444)
selections=np.random.choice(2580, 2000, replace=False)
select=np.in1d(range(2580),selections)


# In[4]:


#Cleaning Data for Entry into Conditional RNN
timevariant=mortality[["heartrate","respiratoryrate", "systolic_bp", "diastolic_bp",
"temperature", "oxygen_sat"]].to_numpy().reshape(2580,25,6)
nottimevariant=mortality[["patient","hour_of_admission","age","copd","heart_failure","renal_di
sease","liver_disease", "metastatic_cancer", "hypertension",
"diabetes"]].groupby("patient").mean().reset_index().drop(columns=["patient","hour_of_admissi
on"]).to_numpy()
result=mortality[["patient","mortality_30d"]].groupby("patient").mean().reset_index()["mortality
_30d"].to_numpy().reshape(2580,1).astype(int)


# In[5]:
```

result=preprocessing.LabelEncoder().fit_transform(result)

In[173]:

#Cond_RNN package

##Time Variant Inputs

inputs = Input(name='in',shape=(25,6)) # Each obseravation has 6 features at 25 time-steps each

#Dense Layer

x= Dense(32)(inputs)

#Dropout Layer

x= Dropout(0.5)(x)

##Time Invariant Inputs

inputs_aux = Input(name='in_aux', shape=[8]) # Each observation has 8 non time-variant

features

#Dense Layer

x_aux= Dense(32, activation="sigmoid")(inputs_aux)

#Dropout Layer

```
x_aux= Dropout(0.5)(x_aux)

#ConditionalRNN Layer
y = ConditionalRNN(32, cell='LSTM')([x,x_aux])

#Outputs
predictions = Dense(1, activation="sigmoid")(y)

cond_model = Model(inputs=[inputs, inputs_aux], outputs=predictions)
cond_model.compile(optimizer='Adagrad', loss="binary_crossentropy", metrics=['AUC'])
data = timevariant[select] # Sample of 2000 patients with 6 features at 25 time-steps each
data_aux = nottimevariant[select] # Sample of 2000 patients with 8 non-time-series featuers each
labels = result[select] # For each of the 2000 patients, a corresponding (single) outcome variable
(dead-1 or alive=0)

condrnn = cond_model.fit([data,data_aux], labels, epochs=255, validation_split=0)


# In[174]:


#Evaluate Model
```

```
cond_model.evaluate([timevariant[~select], nottimevariant[~select]], result[~select])



# In[456]:



cond_model.summary()



# In[450]:



# Plot training & validation accuracy values

plots.plot(condrnn.history['AUC'])

plots.plot(condrnn.history['val_AUC'])

#plots.plot(condrnn.history['val_loss'])

plots.title('Model accuracy')

plots.ylabel('Accuracy')

plots.xlabel('Epoch')

plots.show()
```

In[92]:

```
x_train, x_test, y_train, y_test=train_test_split(timevariant, result, test_size=0.2,
random_state=7, stratify=result)
```

In[120]:

```
#LSTM only

#Time Variant Inputs
inputs = Input(name='in',shape=(25,6)) # Each obseravation has 6 features at 25 time-steps each

#Activation Layer
x = Activation('sigmoid')(inputs)

#Dense Layer
x= Dense(32)(x)

#RNN Layer
```

```
x= LSTM(32, recurrent_dropout=0.5, dropout=0.5)(x)

#Outputs
predictions = Dense(1, activation="sigmoid")(x)

lstm_model = Model(inputs=inputs, outputs=predictions)
#sgd_modified=tf.keras.optimizers.SGD(learning_rate=0.01, momentum=0.0, nesterov=False)
lstm_model.compile(optimizer="Adagrad", loss="binary_crossentropy", metrics=['AUC'])
data = timevariant[select] # Sample of 2000 patients with 6 features at 25 time-steps each
labels = result[select] # For each of the 2000 patients, a corresponding (single) outcome variable
(dead-1 or alive=0)
lstm = lstm_model.fit(data, labels, epochs=1800, validation_split=0.2)


# In[447]:


#Evaluate Model
lstm_model.evaluate(timevariant[~select], result[~select])


# In[118]:
```

```
lstm_model.summary()



# In[345]:



plots.plot(lstm.history['AUC'])

plots.plot(lstm.history['val_AUC'])

plots.title('Model accuracy')

plots.ylabel('Accuracy')

plots.xlabel('Epoch')

plots.show()



# In[176]:



#Outputted Predictions do not follow 0-1 Format as Boolean Inputs

vlaues=cond_model.predict([timevariant[~select], nottimevariant[~select]])
```

In[177]:

```
vlaues=vlaues.reshape(580)
```

In[178]:

```
binaryoutputs=np.where(vlaues<((vlaues.max()-vlaues.min())/2), 0, 1)
```

In[179]:

```
data = {'y_Actual':    result[~select],
        'y_Predicted': binaryoutputs
        }
df = pd.DataFrame(data, columns=['y_Actual','y_Predicted'])
```

```
confusion_matrix = pd.crosstab(df['y_Actual'], df['y_Predicted'], rownames=['Actual'],
colnames=['Predicted'])


# In[181]:


vdf=pd.DataFrame({"Predictions": vlaues, "Actual": result[~select]})
tf.math.confusion_matrix(labels=result[~select], predictions=vlaues)


# In[182]:


axis=vdf[vdf.Actual==1].drop(columns="Actual").plot.density()
axis=vdf[vdf.Actual==0].drop(columns="Actual").plot.density(ax=axis)
plots.legend(["Dead", "Alive"])

plots.title('Density of Predicted Value by Mortality')
plots.xlabel('Predicted Value')

plots.figure(figsize=(100, 50))
```

In[189]:

binned

In[191]:

```
binned=vdf.groupby(pd.cut(vdf["Predictions"], bins=np.arange(min(vdf["Predictions"]),
max(vdf["Predictions"]), 0.1).tolist(), include_lowest=True)).mean()
binned=binned.drop(columns="Predictions").reset_index()
binned.plot.bar(x="Predictions", y="Actual", )
plots.title('Calibration Plot (0.1 Bins)')
plots.ylabel('True Mortality')
plots.xlabel('Predicted Mortality')
plots.figure(figsize=(100, 150))
```

In[194]:

```
smallerrange=vdf.groupby(pd.cut(vdf["Predictions"], bins=np.arange(min(vdf["Predictions"]),
max(vdf["Predictions"])-0.07, 0.07).tolist(), include_lowest=True)).mean()
smallerrange=smallerrange.drop(columns="Predictions").reset_index()
smallerrange.plot.bar(x="Predictions", y="Actual", )
plots.title('Calibration Plot (0.07 Bins)')
plots.ylabel('True Mortality')
plots.xlabel('Predicted Mortality')
```

In[93]:

```
condaucs=np.array([])
condpreds=np.array([])
for train_index,test_index in StratifiedKFold(5, random_state=777,
shuffle=True).split(timevariant, result):
    ##Time Variant Inputs
    inputs = Input(name='in',shape=(25,6)) # Each obseravation has 6 features at 25 time-steps
each
    #Dense Layer
```

```
    x= Dense(32)(inputs)
    #Dropout Layer
    x= Dropout(0.5)(x)
    ##Time Invariant Inputs
    inputs_aux = Input(name='sigmoid', shape=[8]) # Each observation has 8 non time-variant
features
    #Dense Layer
    x_aux= Dense(32, activation="sigmoid")(inputs_aux)
    #Dropout Layer
    x_aux= Dropout(0.5)(x_aux)
    #ConditionalRNN Layer
    y = ConditionalRNN(64, cell='LSTM')([x,x_aux])

    #Outputs
    predictions = Dense(1, activation="sigmoid")(y)

    cond_model = Model(inputs=[inputs, inputs_aux], outputs=predictions)
    cond_model.compile(optimizer="Adagrad", loss="binary_crossentropy", metrics=['AUC'])
    data = timevariant[train_index] # Sample of 2000 patients with 6 features at 25 time-steps
each
    data_aux = nottimevariant[train_index] # Sample of 2000 patients with 8 non-time-series
featuers each
```

```
    labels = result[train_index] # For each of the 2000 patients, a corresponding (single) outcome variable (dead-1 or alive=0)
    condrnn = cond_model.fit([data,data_aux], labels, epochs=255, validation_split=0)
    condaucs=np.append(condaucs, cond_model.evaluate([timevariant[test_index], nottimevariant[test_index]], result[test_index])[1])
    condpreds=np.append(condpreds, cond_model.predict([timevariant[test_index], nottimevariant[test_index]]))
```

In[94]:

```
np.mean(condaucs)
```

In[95]:

```
condaucs
```

In[35]:

```
resultstable=pd.DataFrame(data={"Optimizer": [], "# of LSTM Layers":[], "# of Dense Layers":[], "# of Epochs":[], "Mean AUC":[]})
resultstable=resultstable.append({"Optimizer": "RMSprop", "# of LSTM Layers":32, "# of Dense Layers":32, "# of Epochs":25, "Mean AUC":0.8044620275497436}, ignore_index=True)
resultstable=resultstable.append({"Optimizer": "RMSprop", "# of LSTM Layers":64, "# of Dense Layers":64, "# of Epochs":40, "Mean AUC":0.7935864925384521}, ignore_index=True)
resultstable=resultstable.append({"Optimizer": "SGD", "# of LSTM Layers":32, "# of Dense Layers":32, "# of Epochs":180, "Mean AUC":0.8024403810501098}, ignore_index=True)
resultstable=resultstable.append({"Optimizer": "SGD", "# of LSTM Layers":16, "# of Dense Layers":16, "# of Epochs":459, "Mean AUC":0.7890304684638977}, ignore_index=True)
resultstable=resultstable.append({"Optimizer": "SGD", "# of LSTM Layers":64, "# of Dense Layers":64, "# of Epochs":200, "Mean AUC":0.8080680251121521}, ignore_index=True)
resultstable=resultstable.append({"Optimizer": "SGD", "# of LSTM Layers":256, "# of Dense Layers":256, "# of Epochs":215, "Mean AUC":0.8080519676208496}, ignore_index=True)
resultstable=resultstable.append({"Optimizer": "Adagrad", "# of LSTM Layers":32, "# of Dense Layers":32, "# of Epochs":255, "Mean AUC":0.8094108819961547}, ignore_index=True)
resultstable=resultstable.append({"Optimizer": "Adagrad", "# of LSTM Layers":64, "# of Dense Layers":64, "# of Epochs":50, "Mean AUC":0.8027197480201721}, ignore_index=True)

resultstable
```

In[143]:

```
aucs=np.array([])
for train_index,test_index in StratifiedKFold(5, random_state=777).split(timevariant, result):
    #Time Variant Inputs
    inputs = Input(name='in',shape=(25,6)) # Each obseravation has 6 features at 25 time-steps
each
    #Activation Layer
    x = Activation('sigmoid')(inputs)
    #Dense Layer
    x= Dense(32)(x)
    #RNN Layer
    x= LSTM(32, recurrent_dropout=0.5, dropout=0.5)(x)
    #Outputs
    predictions = Dense(1, activation="sigmoid")(x)

    lstm_model = Model(inputs=inputs, outputs=predictions)
    #sgd_modified=tf.keras.optimizers.SGD(learning_rate=0.01, momentum=0.0, nesterov=False)
    lstm_model.compile(optimizer="RMSprop", loss="binary_crossentropy", metrics=['AUC'])
```

```
    data = timevariant[train_index] # Sample of 2000 patients with 6 features at 25 time-steps
each
    labels = result[train_index] # For each of the 2000 patients, a corresponding (single) outcome
variable (dead-1 or alive=0)
    lstm = lstm_model.fit(data, labels, epochs=1800, validation_split=0)
    aucs=np.append(aucs, lstm_model.evaluate(timevariant[test_index], result[test_index])[1])
    lstm_model.evaluate(timevariant[test_index], result[test_index])



# In[145]:



np.mean(aucs)
```