Machine Learning Project report

Introduction:

Our dataset contains two different files. The first one is the 'admission.csv' file which has 11 features. The second file is 'vitals.csv.' It contains the same ID as the "admission.csv," but this file has distinctive features. We can use only the admission file to do our prediction, or we can use both files to increase the score after using different features. Our main goal in this project is to predict accurately if the patient will survive or not based on several features. We will discuss in depth the different approaches that we had to follow to get a model that predicts accurately if our patient will survive. In our case, we found that we can predict the 0 values with high accuracy. Therefore, we decide to focus more on predicting the positive values of our target variable.

Approach, Discussion, and Results:

In this problem, as we have a classification problem where we need to find whether the patient dies or lives, therefore our target variable is binary. So, in short, our problem is Binary Classification. According to our problem, we can use Multiple Logistic Regression as it is best and suitable for our problem. However, we have 19342 patients and 14 features (columns). Therefore, Multiple Logistic Regression model might fail to give the best prediction because it only performs better when the number of noise variables is less than or equal to the number of explanatory variables. The problem can be addressed with other models such as Random Forest, KNN, SVM, and Decision Tree. However, in general, the SVM model may not perform well because the number of features was extremely low compared to the number of rows. In addition, KNN did not perform well on high dimensional data, and we have 19342 rows because of which KNN might give us a bad prediction. Keeping the problem in mind and details about the dataset we will be using Random Forest this time because it seems suitable and may give us the best prediction in comparison to other models. Also, Random Forest is considered an efficient and strong model because it does calculate the average of all the predictions while removing the biases. Random forests are used for regression and classification problems. In situations with large datasets where interpretability is not a large concern, Random Forest is a suitable possibility. Random forests provide the highest level of accuracy among all classification methods. As well as handling big data, the random forest can handle data with thousands of variables. A class that is infrequent compared to other classes can be automatically balanced. In conclusion, we will get a model that prevents overfitting with Random Forest.

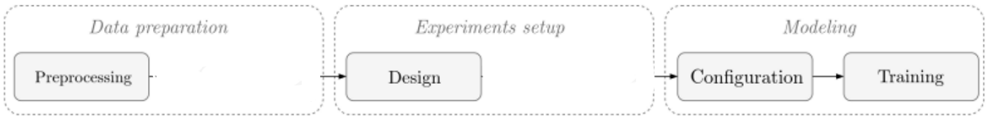


Figure 1: Model Design

Moving forward to the problem, now we will pre-process our dataset to pass into the model. We merged both Vitals and Admission datasets because all the parameters seem too important based on the correlation graph. Before moving to the actual pre-process part, we have a time-series dataset because we merged both datasets where one was Timeseries and the second was tabular data. After merging we got the dataset for 19342 patients in 48 hours format. We then used statistical features to import or convert the dataset into a tabular format to use for model prediction. The most used and powerful method seems to merge the dataset by the Mean. We used this method to merge our

dataset by mean on HADM_ID to covert the dataset in a tabular format where we got 1 row per patient. In simple words, now we have 19342 rows for 19342 patients, not like the old one where we were having 928416 rows in the 48 hours format (time-series).

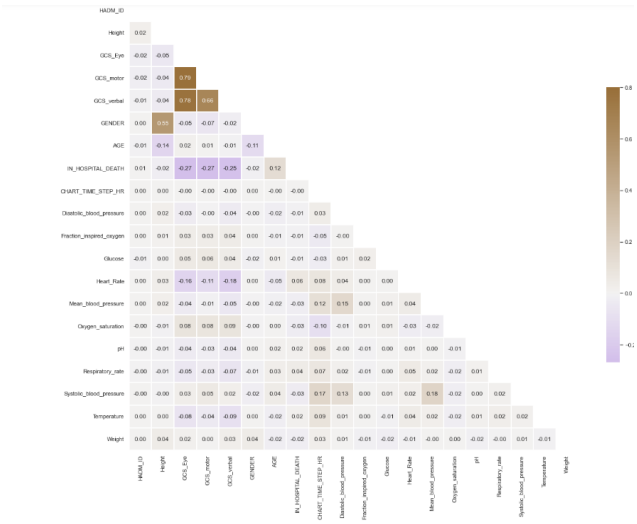


Figure 2: Correlation-plot

```
In [42]: n_obs_tst = train_dataset.shape[0]
         train_dataset.isna().sum()/n_obs_tst

Out[42]: HADM_ID                    0.000000
         Height                     0.808913
         GCS_Eye                    0.000362
         GCS_motor                  0.000672
         GCS_verbal                 0.000465
         ADMISSION_TYPE             0.000000
         ETHNICITY                  0.000000
         GENDER                     0.000000
         FIRST_CAREUNIT             0.000000
         AGE                        0.000000
         IN_HOSPITAL_DEATH          0.000000
         CHART_TIME_STEP_HR         0.000000
         Diastolic_blood_pressure   0.263996
         Fraction_inspired_oxygen   0.880626
         Glucose                    0.789626
         Heart_Rate                 0.242903
         Mean_blood_pressure        0.252336
         Oxygen_saturation          0.277549
         pH                         0.854873
         Respiratory_rate           0.247691
         Systolic_blood_pressure    0.253796
         Temperature                0.729326
         Weight                     0.906899
         dtype: float64
```

Figure 3: Proportion of missing values

After merging the dataset by mean, we lost the three categorical columns we have to add them again to the dataset. After adding the dataset, we converted these three categorical columns to the float by using the most useful and powerful method OneHotEncoder.

Here, we successfully converted the dataset into a tabular format with all float values now we can move forward to the pre-processing part. However, before moving on to the pre-processing part it is better to see if we have any outliers or missing values in the dataset. We checked that in the time-series dataset (928416 rows) we have 5 columns with more than 70% missing values like height. However, after merging the dataset by ID most of the columns did not contain any missing values except “height.” So, we dropped it (height column). In addition, other columns also contain missing values however the proportion for missing values is less than 30%, so to deal with these columns, we can adopt a *simple missing value imputation* that considers every column separately. In principle, we will impute the *mean* value, which is perhaps the simplest strategy. *Scikit-learn* implements a simple imputer through the class `SimpleImputer`.

After dealing with the missing values, we moved to the split part where we drop the target variable and HADM_ID column from our dataset and create one new dataset only with the target variable. Moving forward to the pre-processing part now we did the scaling part. We did the scaling only on the X dataset (the dataset without the target variable). In this context, we used a standard scaler because it seems suitable, and with a MinMax scaler, we might lose the details of the data. After scaling the data, we were ready to move forward to design a model for predicting the target algorithm. However, before moving forward we will just use the Train dataset, not the Test. So, we did split our Train data into two parts with an 80:20 ratio. We just used 20% of Train data as Test data for deciding the best model and then we will apply the same model to our original test data.

Moving forward to the model prediction, we summarise that we deal with missing values, cleaning the dataset, scaling by standard scaler, and splitting the dataset to find the best model to use on our Original Test data. Now we were ready to use our model which was Random Forest as it seems perfect and suitable for the problem. To get a better result, we used Hyper-tunning to get the best model with the best parameter. We used GridSearchCV for Hyper-tunning from “sklearn.” In GridSearchCV, we will first decide a range for our hyper-parameters to use on our data. The fundamental concept is to

make a grid of hyper-parameters and then try all possible combinations. After finding the best parameter we then used the best parameter for defining our model for Random Forest and we can check if our model is best based on Confusion Matrix, ROC_AUC score, Accuracy, Precision, and Re-call.

```
In [27]: from sklearn.metrics import confusion_matrix, accuracy_score
cm = confusion_matrix(y_test, y_pred)
print(cm)
accuracy_score(y_test, y_pred)
from sklearn.metrics import classification_report
matrix = classification_report(y_test,y_pred,labels=[1,0])
print('Classification report : \n',matrix)
#Area under ROC curve
from sklearn.metrics import roc_auc_score
roc_auc_score(y_test,y_pred)

[[3463   36]
 [ 307   63]]
Classification report :
              precision    recall  f1-score   support

           1       0.64      0.17      0.27       370
           0       0.92      0.99      0.95      3499

    accuracy                           0.91      3869
   macro avg       0.78      0.58      0.61      3869
weighted avg       0.89      0.91      0.89      3869

Out[27]: 0.5799908081845778
```

Figure 4: Random forest summary table

In the figure shown above, we can see that the model is not performing well with the best parameters. It might be a problem with the Threshold. We tried to use thresholding to find the optimal value to get a higher score. The precision-recall curve can help us focus on the positive values (The minority class). The default threshold for a model is 0.5, we found the best threshold is 0.3 based on the formula for Threshold which is $\frac{2}{(\frac{1}{Precision}+\frac{1}{Recall})}$.

```
from sklearn.metrics import roc_curve, precision_recall_curve
def threshold_search(y_true, y_proba, plot=False):
    precision, recall, thresholds = precision_recall_curve(y_true, y_proba)
    thresholds = np.append(thresholds, 1.001)
    F = 2 / (1/precision + 1/recall)
    best_score = np.max(F)
    best_th = thresholds[np.argmax(F)]
    if plot:
        plt.plot(thresholds, F, '-b')
        plt.plot([best_th], [best_score], '*r')
        plt.show()
    search_result = {'threshold': best_th , 'f1': best_score}
    return search_result
```

Figure 5: threshold function

```
: from sklearn.metrics import confusion_matrix, accuracy_score
cm = confusion_matrix(y_test, predicted)
print(cm)
accuracy_score(y_test, predicted)
from sklearn.metrics import classification_report
matrix = classification_report(y_test,predicted,labels=[1,0])
print('Classification report : \n',matrix)
#Area under ROC curve
from sklearn.metrics import roc_auc_score
roc_auc_score(y_test,predicted)

[[3331  168]
 [ 219  151]]
Classification report :
              precision    recall  f1-score   support

           1       0.47      0.41      0.44       370
           0       0.94      0.95      0.95      3499

    accuracy                           0.90      3869
   macro avg       0.71      0.68      0.69      3869
weighted avg       0.89      0.90      0.90      3869

: 0.6800471949514534
```

Figure 6: Random forest table with a threshold of 0.3

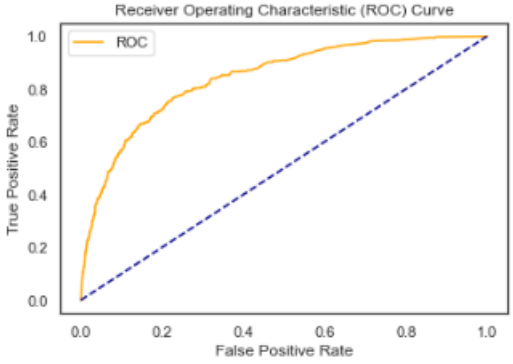


Figure 7: ROC Curve of Random forest model with a threshold of 0.3

After running our model for few times with the function to find optimal threshold. The result shows that the optimal threshold is 0.3. Based on the F1 score, precision, recall, and ROC_AUC scores, this threshold generated the best model. Generally, Precision is defined as the ability of the classifier not to misidentify a sample as positive that is actually negative. The recall is TP/ (TP + FN), where TP is the number of true positives and FN is the number of false negatives. This refers to the ability of the classifier to find all positive samples. An AUC ranges between 0.5 and 1, with 0.5 standing for poor quality and 1 being excellent quality. We can see that our model works well based on all the values and we can make good judgments based on our results.

Our model has an imbalanced target variable which will produce more "0" values and it is hard to predict positive values. Still, our model performs the best with a ROC_AUC score of 0.68 and an accuracy of 90% with good precision and recall on both values. We will keep this model for our Test data.

After using random forest and thresholding we got an F1 score of 44. Therefore, we decide to use neural networks to get a higher score. Our dataset contains 48 hours of data of patients which is a time series dataset. Therefore, we have chosen to use an RNN model that is well suited for time series. In RNN there are so many deep learning models that perform best on time series. In the field of deep learning, long short-term memory is an artificial recurrent neural network (RNN). In this problem, first, we will use the LSTM model for our problem, because LSTM seems more powerful in the time-series data. Time series data is well-suited to LSTM networks for classification, processing, and making predictions, as major events can occur in unknown time intervals. This provides increased accuracy for forecasters, which promotes better decisions. Before choosing our LSTM model, we have to make sure that the data is in a three-dimensional array (Batch size, timesteps, features). We have to use our old dataset which we got after merging the Vitals and Admission with 928416 rows where we can simply convert this into a suitable format for the LSTM model which is 19342 (patients), 48 Hours, 24 features. Before converting it into three-dimensional array, remember that we were having missing values for 5 columns with dataset before merging it by mean. So, to make sure we checked it again and got more than 70% missing values for 5 columns. We dropped those columns and converted the data into three-dimensional array. Then, we simply defined the LSTM model to check if we get the best score. In our LSTM model, we used 3 layers architecture with 1 input layer, 1 drop-out layer to avoid overfitting, and one dense layer. We used 1 neuron for our dense layer as our target variable is only in 1 column. We used 0.2 as a drop-out rate as it is most useful for avoiding overfitting. With the optimal threshold and LSTM architectural time-series model, we then complied the model by using the loss function as "binary_cross_entropy" because this loss function is suitable for our problem and our target variable is binary. We use the accuracy function as "Adam" which is useful in almost every context. We used 10 epochs for our deep learning model because it gives us optimal score and avoid overfitting. After compiling and running 10 epochs we got the F1 score of 0.48 which is much similar to Random Forest. To improve the model more, we used the activation function as "sigmoid" because most of the time, the Sigmoid activation function is relevant to the binary classification problem. After using the Sigmoid Function our score improved from 52 to 54. We will keep this model to use on our original test data.

To improve our result for positive class we then used the CNN model for 1dimensional data (Conv1D) which is also quite useful and gives the best prediction. Using trainable filters, Convolutional Neural Networks are capable of successfully capturing spatial and temporal patterns through their ability to classify time series, while assigning weight to these patterns according to their relative importance, making them the most popular Deep Learning technique for Time Series Classification. In the architecture, we used 4 layers, where the first layers were input layers with the activation function "relu". In most cases, it is used because Relu is simple, fast, and empirically it seems to work well. In addition, we use a filter size of 48 and kernel size of 3, as in principle kernel size should be used from

2 to 10 and most of the time 3 performs best. However, we have tried to use 48 as well because kernel size is to define the total number of parameters as we have 48 hours timesteps, so it shows relevant to use 48. However, with a 48-kernel size, we got a poor result. So, we used 3 as kernel size in this architecture. To make model architecture better we used padding as "Valid." The definition implies no padding and assumes that all dimensions are valid so that the input shapes are completely covered by the filter. To avoid overfitting, we used one more drop-out layer before we went to the second layer. This was followed by the MaxPool layer which reduces the dimensionality of the data by reducing the number of features in the output from the prior layer. We gave the pool size as 2 for this layer. Then again, we used a drop-out layer to avoid overfitting followed by Flatten layer which will flatten our output in one direction array. Then we used a dense layer with 1 neuron for our target variable. After using CNN for 1-dimensional data, with all the best parameters we got a poor result as compared to LSTM. The F1 score for the model was around 50 after running the model several times. However, as it shows the best result based on the resulting factor, so we kept this model to try on Test data.

Furthermore, to get a better result we tried the combination of the two models which was CNN with LSTM. This model performs the best because CNN Layers are used to extract features from the time data and then LSTM will be used to do the model prediction on extracted features. We decide to use a 5 Layers architecture. The first 3 layers are the CNN layer which was similar to our last CNN model then we have an LSTM layer with 100 units and the last layer for the Dense layer. However, in this model architecture, we used recurrent drop-out and drop-out in the LSTM layer followed by the default recurrent activation function as "Hard Sigmoid." To prevent overfitting in the recurrent layers, we used the recurrent dropout method. Then, we used a dense layer with 1 neuron followed by an Activation layer with an activation function as "linear".

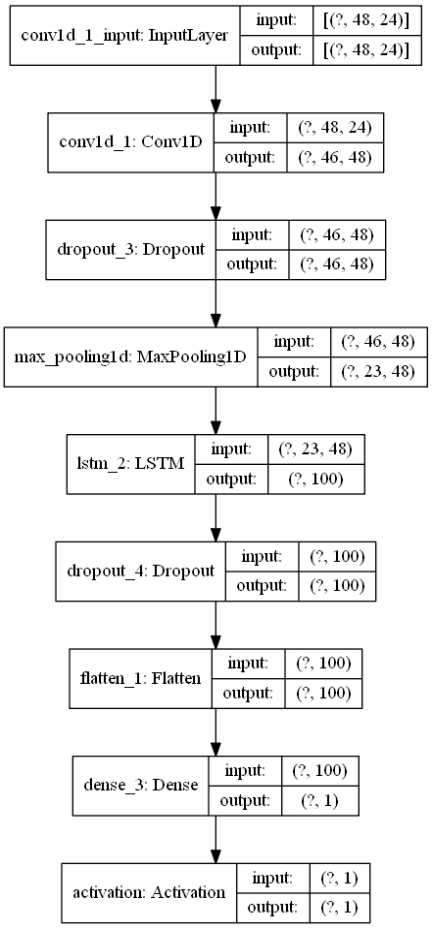


Figure 8: Model architecture

By using the Linear Activation function, we merge all layers of the neural network. Even if there are many layers in a neural network, the last layer is still linearly related to the one layer. Because, the neural network is reduced to just one layer by using a linear activation function. Further, a linear layer without bias can calculate an average correlation between the output and input. After running the model several times with an optimal threshold, we got the best result, this result varies between 58 to 62 which is quite surprising but best as compared to all the models.

```
In [41]: from sklearn.metrics import confusion_matrix, accuracy_score
         cm = confusion_matrix(y_test, seq_predictions)
         print(cm)
         accuracy_score(y_test, seq_predictions)
         from sklearn.metrics import classification_report
         matrix = classification_report(y_test,seq_predictions,labels=[1,0])
         print('Classification report : \n',matrix)
         #Area under ROC curve
         from sklearn.metrics import roc_auc_score
         roc_auc_score(y_test,seq_predictions)

         [[3387  120]
          [ 157  205]]
         Classification report :
                       precision    recall  f1-score   support

                    1       0.63      0.57      0.60       362
                    0       0.96      0.97      0.96      3507

             accuracy                           0.93      3869
            macro avg       0.79      0.77      0.78      3869
         weighted avg       0.93      0.93      0.93      3869

Out[41]: 0.7660405314075874
```

Figure 9: summary table of CNN-LSTM model

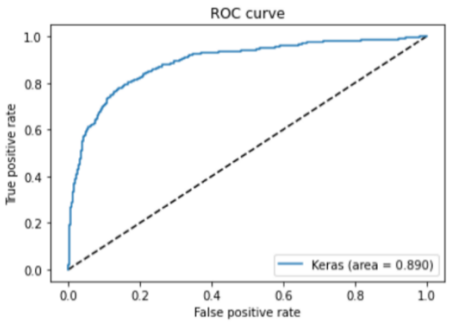


Figure 10: ROC curve of the model

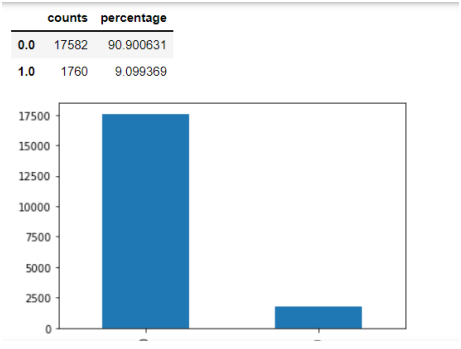
This model gives us the best F1 score, precision, recall, and accuracy were also around 93 with a ROC_AUC score of around 76. To make sure about the model, we plotted the ROC_AUC Curve. It is generally believed that the closer the ROC curve is to the upper left corner, the more effective our model is. Based on the curve, it seems the algorithm is a perfect match for our model. As it is hard to predict the more positive values and hard to get the F1 score near to or more than 70 because our data is Imbalanced. However, getting a score of around 60 is much better in this scenario.

In the end, we used all these 4 models and the same architectures on our original test data and got the score nearly around the same with Random Forest, LSTM, CNN, and CNN-LSTM of around 44,50,53,56, respectively.

Reflection (Future Improvements):

We can see that using every function, layer, and hyperparameter in deep learning turns into getting a more accurate score. So, in the future to get a better score for positive values we can use GridSearchCV, KerasClassifier, and others for defining all the hyper-parameter. We can also define a function for creating a model which will accept different architectures and simulate each architecture and will find the best score. We can also use a loop function for each layer to generate more possibilities for a different architecture. Using the hyper-tunning in deep learning and a model which is already performing well without it will surely give us the best score and we will end up with a better F1 score for a positive value.

In addition to this, we know that in our dataset our target variable is Imbalanced, and it is skewed with a ratio of 9:91.



	counts	percentage
0.0	17582	90.900631
1.0	1760	9.099369

Figure 11: Imbalanced target values

Because we have imbalanced data, it is hard to predict more positive values (minority class). To deal with this type of problem we can implement SMOTE or any other resampling techniques to resample the data and normalize it. Adopting this will help us to get a better score based on F1 score, precision, and recall. Moreover, we can also try to change the weights while fitting the model to our Train data. However, this might give us overfitting. Resampling will also give us overfitting; however, we can use oversample and undersample together with the help of SMOTE function to avoid this. Furthermore if possible, we can also try to get more datasets to overcome the unbalancing problem.

We can also define a function for finding the optimal threshold for our Test data. As our optimal threshold function fluctuate between 0.18 to 0.3. To get the best threshold, we can adopt Monte Carlo Simulation or Bootstrap to run it 1000 times and find one optimal threshold which we can use on our Test data.