Ece5307 Au22 Final Project

Group 13

Introduction:

This report presents the work done in the final project of the course *Introduction to Machine Learning (ECE-5307).* The individual working model was used in this project; each group member worked on their part of the project. Names of the group members and the part of the project they are coping with include:

1. **Linear Methods ~** Anusheel Goswami
2. **Neural Networks ~** Hassan Iftikhar
3. **Tree-Based Methods ~** Utkarsh Pratap Singh Jadon

Chapter 1: Linear Methods (Anusheel Goswami):

In this section, we are going to discuss how we have arrived at the best method for linear techniques and the design process involved in it. Logistic regression and SVM models were used to find the best possible AUC score.

1.1. Data Pre-processing

The training dataset was split into the training dataset and a testing dataset with a ratio of 80:20. The dataset contained 10000 samples for training, so it was split into 8000 training examples and 2000 validation examples.

1.2. Standardization of dataset

The first step we went forward with was to standardize the dataset. For this, it was important to find a suitable standardization. Standardization is an important technique that is mostly performed as a pre-processing step before machine learning models, to standardize the range of features of an input data set. In this project, StandardScalar and MinMaxScalar standardization was used [1]. In StandardScalar, the data is distributed according to normal distribution i.e., it will transform your data such that its distribution will have a mean value of 0 and a standard deviation of 1. The Minmax Scalar on the other hand transform features by scaling each feature to a given range. This estimator scales and translates each feature individually such that it is in the given range on the training set, e.g., between [0,1]. Since we are using the default values, we have used the range of [0,1]. The performance of both standardization techniques will be explained in the individual training models used in this project. However, on average, for both Logistic regression and SVC/SVM, StandardScalar returned a better ROC AUC score in comparison to MinMaxScalar for both the training dataset and validation test data.

1.3. Logistic regression

Logistic regression was applied to both the standardized data and the raw dataset. It is seen that when standardization is applied, there is only a marginal improvement in the AUC scores of the training dataset and testing dataset. However, it was observed that there wasn't a big difference between the two types of standardization techniques used while calculating performance for both training AUC and test AUC using the L1 penalty or L2 penalty. It was seen that MinMaxScalar does not work with the L1 penalty applied [2]. The data observed are tabulated below in Table 1.1.

It was observed that there wasn't a significant difference in the AUC scores for both test data and training data for both L1 and L2 regularization.

Table 1.1: AUC scores with regularization and standardization

Standardization	Penalty	AUC score on the training dataset	AUC score on the test dataset
StandardScalar	L1	81.8128 %	79.8070 %
StandardScalar	L2	82.5726 %	79.7817 %
MinMaxScalar	L2	82.2588 %	79.8187 %

1.3.1. L1 regularization tuned with GridSearchCV and a pipeline

Here, we will discuss the dataset that was standardized using StandardScalar. L1 regularization when tuned with GridSearchCV gave the value optimal value of inverse regularization strength.

Here C = 0.0031622776601683794 gave the best results. Using L1 regularization, feature selection was done too, which increased the AUC score on the validation test dataset. Figure 1.1 shows the value for C with the least CV error-rate and which was selected.

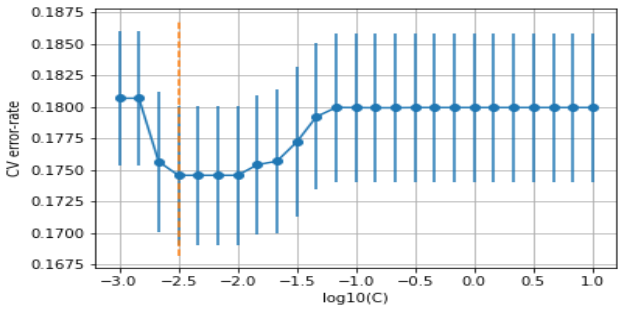


Figure 1.1: Optimal inverse regularization strength using GridSearchCV and pipeline

1.4. Feature Selection:

Considering the features from 1 to 8, it was found that the feature that gave maximum AUC scores for both the test and training datasets was 3, 4, and 8. This was done using L1 regularization. This led to an increase in the AUC score on the validation testing dataset. The AUC on the testing data is 0.808859. The plot for selected features is shown in Figure 1.2.

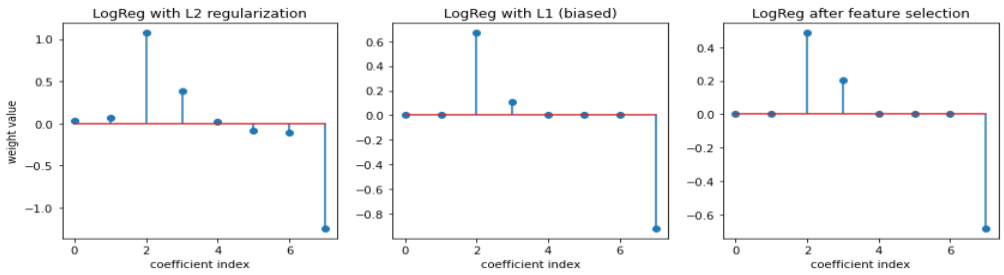


Figure 1.2: Weights with L1 and L2 regularization and feature selection

Later on, feature selection was done using SelectKBest and it returned the features 3, 4, 6, 7, and 8 to be selected. These features were later selected in the SVM model which gave the best performance in Kaggle [3]. The dataset was then transformed to only contain the data for the aforementioned features.

1.5. SVC/SVM

For SVM, we started by using random parameters and then followed it up with GridSearchCV to find the best parameters for this problem. We used grid search CV for the values in the range for C of [0.1, 1, 10, 15, 20, 25, 50] and [0.001, 0.01, 0.1,0.00001,0.05] for gamma [4]. Also, the kernels that were checked were linear, polynomial, sigmoid, and rbf. The best results were given by the value for C = 20 and gamma = 0.2. And the best kernel for this problem was rbf. The dataset here was standardized both using StandardScalar and MinMaxScalar. Better results were obtained using StandardScalar for preprocessing the data. Without feature selection, the AUC score on the test data was above 94%. And on the test dataset for validation, it gave a score of 88%. This method gave a Kaggle score of 89.5%.

After that feature selection was done and features 3, 4, 6, 7, and 8 were selected as mentioned above, the AUC score on training data reduced to 92% but the score on validation test data increased to 91% and the public Kaggle score increased to 90.9%. This shows that feature selection definitely helped with the problem. We can say that overfitting was taking place when all the features were selected. With the best features being selected, the AUC test scores increased.

It can be clearly observed that the SVM model performed significantly better than logistic regression for this problem. And there was an increase in the AUC score with feature selection. The best scores were given for C = 20, gamma = 0.2, and kernel = rbf.

Chapter 2: Neural Networks (Hassan Iftikhar)**:**

A base model was designed by hand to solve the binary classification problem. In this study, the problem was solved using fully connected neural networks (FCNN) and Convolutional Neural Networks (CNN). The results of both CNN and DNN were compared to each other.

2.1. Design Process:

The base model was designed manually with a random guess of the number of layers, and the number of neurons in the layer, with Adams optimizer (because it works well for most of the datasets) and binary cross entropy loss with a logic-less selection of learning rate as 1e-3. The hyperparameters were changed by keeping an eye on "Test Accuracy as a performance metric." When the test accuracy is better, the model's performance seems to increase. After designing this model, the software framework Optuna is used to maximize the average test accuracy of the model. This process is first performed with a fully connected neural network (FCNN), then it is tried on with convolutional neural networks (CNN).

2.2. Preprocessing

The design process involved splitting the data into training and testing data with a split ratio of 80/20, respectively. The dataset contained 10000 samples for training, so it was split into 8000 training examples and 2000 validation examples. It was a binary classification problem, so the output was just one class. The dataset was both standardized and normalized. When testing the models with each of the standardized data and normalized data, the results showed that data standardization seems to work better than normalization [1]. The handcrafted fccn model was used to determine which preprocessing would serve better for this dataset.

2.3. Fully Connected Neural Networks (FCNN) Model Design:

A FCCN model was handcrafted, keeping in mind the performance of the model using test accuracy as a performance metric. The test accuracy was used as a performance metric because AUC score seemed to improve if the model was designed properly without overfitting.

2.3.1. Handcrafted Model FCNN (*Handcrafted Model FCNN.ipyjnb*):

The design process involved the handcrafting of the model. The base model with two layers, each having 100 neurons, was selected (randomly). Keeping an eye on test accuracy as a performance metric to check whether the model is learning or not, another layer was added, and the model's performance rose.

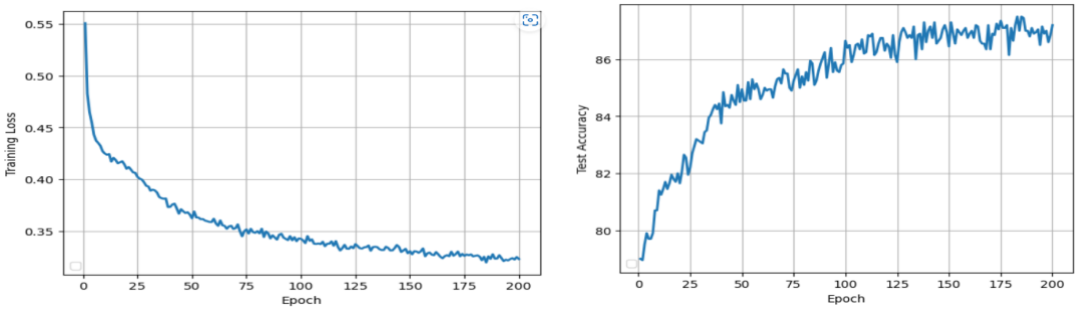
2.3.1.1. Hyperparameters Tuning:

The base model was tuned with different hyperparameters. The different hyperparameters and their randomly tried values are listed below:

- Number of layers ~ [1, 10, 5, 2, 3] ~ 3 layers were found to give better results.
- Learning rates ~ [1e-4, 1e-2, 1e-3, 3e-3, 2e-3] ~ 2e-3 was found to be suitable.
- Activation functions for Hidden Units ~ [ReLU, ELU, Leaky ReLU, Sigmoid, Softmax, Tanh] ~ Sigmoid served the best role.
- Loss Functions ~ [Binary Cross Entropy Loss, Cross Entropy Loss, Binary Cross Entropy with Logits Loss] ~ Binary Cross Entropy Loss (BCE Loss) gave the best results.
- Optimizers ~ [Adam, RMSProp, NAdam] ~ Adam worked well as an optimizer for this FCNN.
- Dropouts ~ [0.2, 0.3, 0.5, 0.1] these values of dropouts were tried in every layer turn by turn. It was found that the dropout in the first layer helps in the betterment of results. The learning curves degraded when the dropouts were added in the second and the third hidden layer.
- Batch Normalization ~ The normalizations were tried layer by layer, and they weren't suitable.
- Epochs ~ [10-200] ~ The AUC score was observed and uploaded to Kaggle.

2.3.1.2. Results:

The results after finalizing all the above parameters showed that a test accuracy of up to 87.5% was achieved by this model, and this model achieved a training accuracy of up to 86%. The results of the models, when saved and uploaded to Kaggle, showed an AUC score of **0.91223**. The figures below represent the testing accuracy and training loss in our model.



2.3.2. Optuna Suggested FCNN (*Optuna Suggested FCNN.ipyjnb*)

The Optuna framework was used for the hyperparameter tuning of the neural network [5]. Optuna is an automated framework for optimizing hyperparameters. It used the trial and error approach to find optimal hyperparameters. A range of values for different hyperparameters is specified in Optuna, and it performs some trials, and tries to maximize the mean of the test accuracy during last 10 epochs.

2.3.2.1. Hyperparameters Tuning:

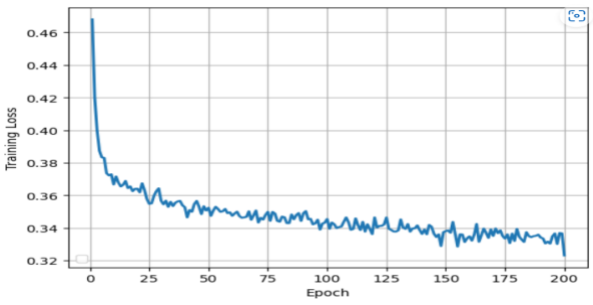
The optuna framework was provided with various choices, and it uses hit and trial approach to maximize the testing accuracy. The hyperparameters and their corresponding choices are given below:

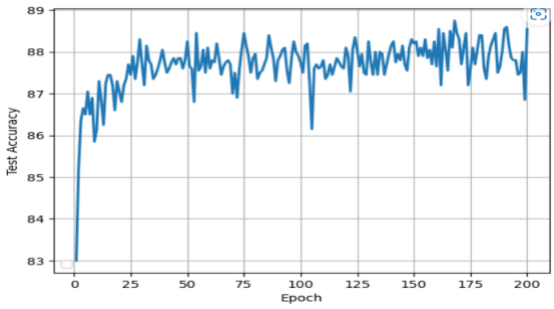
- Number of layers ~ [4-120] ~ 3 layers were found to give better results.
- Number of Neurons in each layer ~ [4-120] ~ This range of choices was given to Optuna.
- Learning rates ~ [1e-4 -1e-2] ~ 0.0045 was found to be suitable among the tried values.
- Activation functions for Hidden Units ~ [ReLU, ELU, Leaky ReLU, Sigmoid] ~ LeakyReLU showed the best performance in terms of testing accuracy in the hidden layers.

- Loss Functions ~ [Binary Cross Entropy Loss] ~ BCE Loss was the appropriate loss function.
- Optimizers ~ [Adam, RMSProp, SGD] ~ Adam worked well as an optimizer for this FCNN.
- Dropouts ~ [0.2-0.5] these dropout ranges were tried in every layer turn by turn. It was found that the dropout in the first layer helps in the betterment of results.
- Batch Normalization ~ Optuna suggested no normalization should be used
- Epochs ~ 200 epochs were selected by keeping an eye on learning criteria and overfitting.

2.3.2.2. Results:

The results after finalizing all the above parameters showed that this model achieved a test accuracy of up to 88%, and this model achieved a training accuracy of up to 86%. The results of the models, when saved and uploaded to Kaggle, showed an AUC score of **0.91238**. The figures below represent the testing accuracy and training loss in our model. The results can be seen in "*Optuna Model FCCN.ipyjnb*"



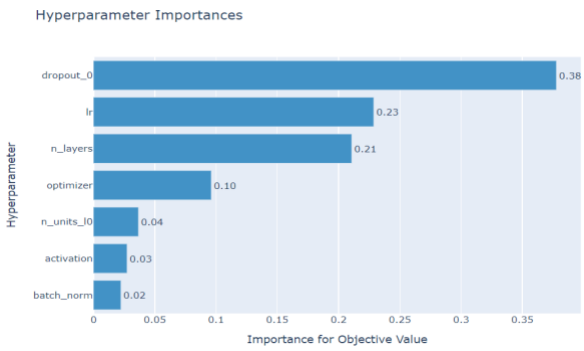


The results the Optuna displayed are shown below:

```
In [56]: best_trial = study.best_trial

         for key, value in best_trial.params.items():
             print("{}: {}".format(key, value))
         print("Accuracy:",study.best_value)

         n_layers: 3
         activation: LeakyReLU
         batch_norm: No
         n_units_l0: 74
         dropout_0: 0.294568362859655
         n_units_l1: 34
         dropout_1: 0.4788295287409655
         n_units_l2: 109
         dropout_2: 0.1289882165935476
         optimizer: Adam
         lr: 0.004577889577112163
         Accuracy: 88.13499999999999
```



2.4. Convolutional Neural Networks:

The CNN model was first handcrafted, then fine-tuned with Optuna [6]. The details of processing the results using CNN are described below:

2.4.1. Handcrafted Model CNN (*Handcrafted Model CNN.ipyjnb*):

The CNN model was roughly designed with 2 convolutional layers and 3 fully connected layers. The kernel filter sizes were selected randomly by the hit and trial method, and the number of layers of FCN was also adjusted by hit and trial. There were many hyperparameters that asked for tuning, but different combinations were tried, and only a few of them showed better performance. These parameters involved dropouts

2.4.1.1. Hyperparameter Tuning:

The base model was selected with two convolutional layers, and three fully connected layers. The number of layers, the number of channels in the convolutional layers, number of neurons in each layer, pooling method, hidden layer activation functions, learning rate, dropout and the batch normalization were the few parameters that were played with in order to design a better CNN based model for this dataset

2.4.1.2. Results:

The handcrafted CNN doesn't seem to give away good performance. The model was learning things in a lesser number of epochs, but the test accuracy didn't seem to go up beyond roughly 85.5%, and the model started to overfit afterward. So, this study showed that CNN might not be a good choice for this dataset. However, to verify this claim, an optuna study was performed on the CNN dataset too. The results for this model can be found in "*Handcrafted Model CNN.ipyjnb"*

2.4.2. Optuna Based CNN Model Design (*Optuna Model CNN.ipyjnb*):

The Optuna framework was used to tune the hyperparameters for CNN, but Optuna also wasn't able to converge to a better model for this dataset [7]. The mean of last 10 epochs' test accuracy was dropped down

to 84.8% for the best trial of Optuna. This suggests that convolutional neural network doesn't seem to work for this dataset. The output of Optuna for suggested CNN is specified as shown below:

```
conv_layers: 3
conv_kernel: 5
activation: LeakyReLU
batch_norm: Without_BN
pooling: Avg_Pooling
pool_kernel: 1
dropout: 0.08110457293601621
n_units_l0: 48
n_units_l1: 16
n_units_l2: 48
optimizer: RMSprop
lr: 0.0006645887720724897
Accuracy: 84.89
```

2.5. Summary of Experiments Performed:

From all of the above experiments, the reported ROC score on Kaggle is shown below [8]:

Model	**Mean of last 10 epochs of Test Accuracy on 20% of training data (which behaves as test data because of train test split)**	**Test AUC on Kaggle Data**
Handcrafted FCCN	87.20%	0.91223
Optuna Tuned FCCN	88.13%	**0.91238**
Handcrafted CNN	85.3%	<88%
Optuna Tuned CNN	84.8%	<88%

From the table above, it can be seen that FCCN model tuned by Optuna gave the best results.

Chapter 3: Tree-Based Methods (Utkarsh Pratap Singh Jadon)**:**

Decision Trees are a non-parametric supervised learning method used for classification and regression. They can be used to represent decisions and decision-making visually and explicitly [9]. Algorithm identified boundary conditions on features to determine which sample belongs to which category. Data gets segmented into smaller groups, also known as 'leaves', in an iterative approach. Although decision trees are computationally fast, they are prone to overfitting. We can make use of multiple decision trees to tackle this problem. This technique is known as the ensemble method.

This section covers the design process, analysis, and results obtained from applying two tree-based methods on the given classification problem.:

- Random Forest Classifiers - Random Forest Classifiers algorithm involves application of an ensemble of decision tree models. Each of the trees vote to attain final classification. These classifiers follow a process called 'bagging', which implies usage of only a subset of data to create each tree. This is done to maintain variation between the trees. Unused data is used for cross-validation. Variance reduces with the increase in number of trees. But a large forest can lead to overfitting as well. There is a need to find an optimal solution that reduces the variance but does not lead to overfitting on data.
- XG Boost Classifiers – Unlike Random Forest Classifiers, in XG Boost, the trees are trained sequentially. Each tree attempts to rectify the error generated by previous tree [10]. XG Boost has many parameters like number of estimators, tree depth, learning rate, regularization alpha, lambda, etc., which can be optimized to accurate classification.

3.1. Dataset Study

Before implementation of any method, the dataset was studied, loaded, and split into training and test samples. 80:20 splitting ratio was adopted for all our work.

Table 3.1.1: Train – Test Split

Training Samples	Test Samples	Number of Classes
8000	2000	2

3.2 Choice of Standardization

To compare the impact of standardization of dataset with all other techniques explored, we focused on two methods mainly, using StandardScaler and using MinMaxScaler from sklearn. After applying both standardizing techniques to given dataset, we implemented basic models of both Random Forest and XG

Boost Classifiers. We observed that training AUC-ROC and test AUC-ROC scores both reduced when data was standardized. By varying the parameters as well it was observed that AUC-ROC scores are better without standardizing data. With this observation, we decided not to standardize and run our models on the given dataset itself.

3.3. Basic Models

Before any hyperparameter tuning, we designed basic models of Random Forests and XG Boost classifiers to observe the train and test accuracy and AUC-ROC scores. This was done to get a vague idea about how different parameters affect the two models. Parameters like number of estimators, max depth, max samples, regularization alpha and lambda, and few others were varied to study the impact of change in these parameters on the desired output [11]. Once familiarized with the methods, hyperparameter tuning was started.

3.4. Feature Selection

After creating a basic model for both Random Forests and XG Boost, all 8 features were analyzed. It was observed that 5 out of those 8 features were important, as shown in Figure 3.4.1. Hence, feature selection was done using SelectKBest class available through Scikit-learn API. Score function of 'f_classif' was used for this operation. But upon calculating the AUC scores for training data and test data on Kaggle, it was seen that model performed slightly worse than how it performed without feature selection. Hence, the approach of feature selection was subsequently dropped for the tree-based methods.

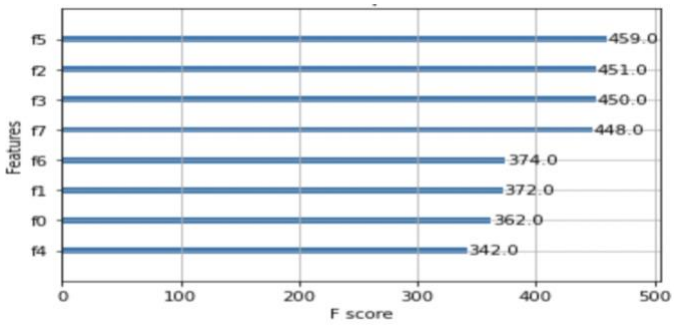


Figure 3.4.1: Feature Importance

3.5. Hyperparameter Search

We started off with Grid Search to try a bunch of combinations and see what works best. Since this is a common method, Scikit-learn has built-in functionality that we used, called GridSearchCV. Here, CV stands for Cross-Validation which is another technique to improve model performance. By defining a dictionary of parameters and creating a GridSearchCV object, we fit it on the training data obtained from splitting the given dataset. But Grid Search tries all possible hyperparameters combinations which increases the time and computation complexity. Also, the AUC-ROC scores obtained from this method were not up to the mark. Because of this, we then attempted Random Search. It is a cheaper alternative wherein a global optimum can be reached within just few iterations. This algorithm explores the random distinct values of the parameter space, unlike Grid Search which tests all distinct points. Here, the selection of hyperparameters is completely random. We used Scikit-learn in-built functionality, RandomizedSearchCV, which works like the GridSearchCV API. But upon optimizing the parameters through this approach, we saw an increase in train AUC-ROC but decrease in test AUC-ROC. We concluded that this technique made the model over-fit on training data.

Finally, we opted for Optuna, an automatic Bayesian hyperparameter optimization framework [12]. It is extremely light weight and can efficiently search large spaces and prune unpromising trials for faster results. It also parallelizes the hyperparameter search over multiple threads which improves its overall performance. Using best parameters obtained from Grid Search and Random Search, firstly a function was created, named ‘objective’, which takes an object, called ‘trial’. ‘Trial’ object is responsible to provide best results using a wide range of parameters. Smaller ranges were uniformly distributed, and logarithmic sampling was done for larger ranges. We get a single value score from ‘objective’ function, which must be minimized or maximized. Once the ‘objective’ function is created, we generate a ‘study’ object, which takes in two parameters: the ‘objective’ function, and the number of experiments [13]. Once the ‘study’ was completed,

we analyzed sliced plots of all features, optimization history plots, and hyperparameter importance, as shown in Figure 3.5.1, Figure 3.5.2, and Figure 3.5.3, respectively.

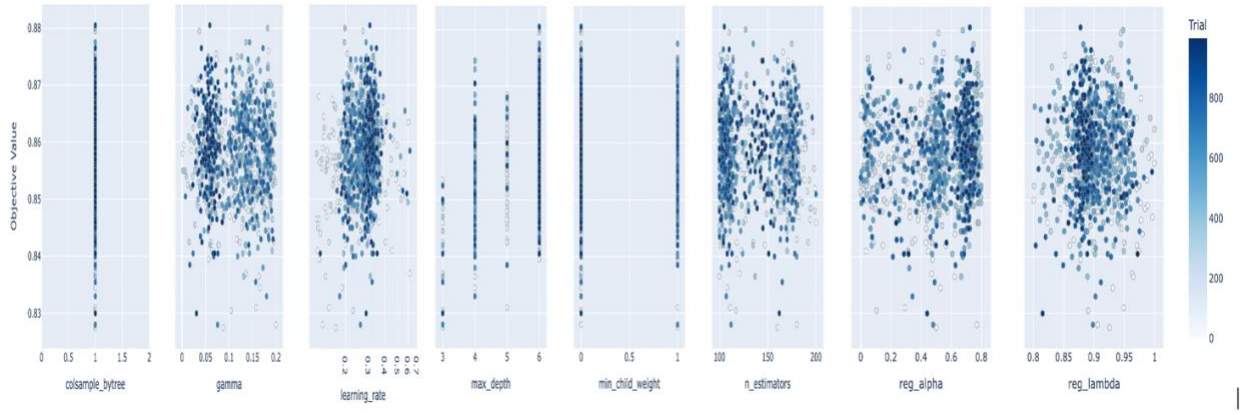


Figure 3.5.1: Sliced Plots of all Hyperparameters

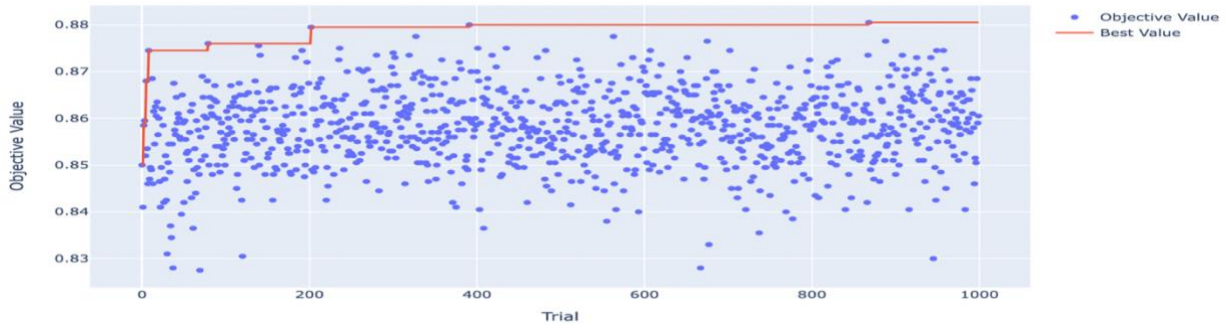


Figure 3.5.2: Optimization History Plot

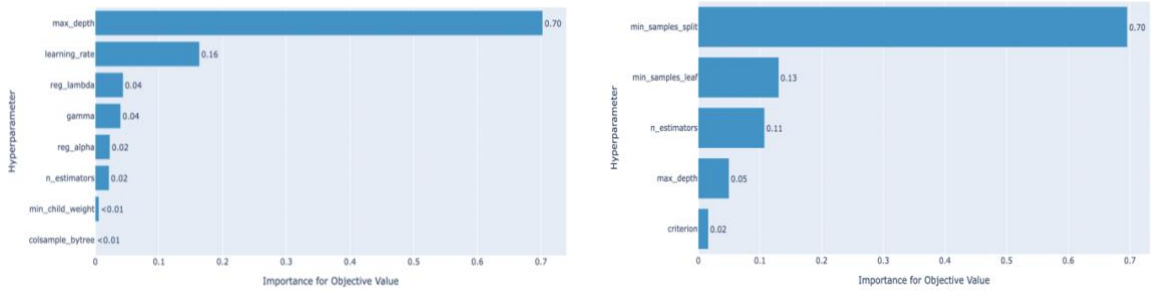


Figure 3.5.3: Hyperparameter Importance in XG Boost vs Random Forests

Hyperparameter importance was plotted to determine the correlation between various parameters and the model accuracy. It helped us to focus on the parameters that were more important than others. Then, the sliced plots were used to adjust the important parameters, so that their optimal values centers within the given range. Using these adjusted values and important features, further 'study' objects were created and XG Boost models were trained using these parameters. We created multiple 'studies' and generated test estimates that were later uploaded to Kaggle, to check model's performance with unknown test data. This technique was applied to both tree-based methods: Random Forests and XG Boost, to generate best and optimal parameters, as shown in Figure 3.5.4.

```
Best hyperparameters:

{'criterion': 'gini',
'max_depth': 14,
'n_estimators': 424,
'min_samples_split': 27,
'min_samples_leaf': 27}
```

```
Best hyperparameters:

{'max_depth': 6,
 'reg_alpha': 0.5901749643346343,
 'reg_lambda': 0.931544839038767,
 'min_child_weight': 0,
 'gamma': 0.1574782659759584,
 'learning_rate': 0.2746329430371991,
 'colsample_bytree': 1.0,
 'subsample': 0.9830898929263234,
 'booster': 'gbtree'}
```

Figure 3.5.4: Best hyperparameters for Random Forest and XG Boost

3.6. Comparison of Methods

Initially, both methods displayed similar results. This could be due to lack of robust hyperparameter tuning. After performing Optuna Bayesian hyperparameter optimization process, we observed that XG Boost worked better than Random Forests model. It generated better results on both the training-testing split data, and the unknown test data on Kaggle.

On comparison, we see that XG Boost performed better than Random Forests, after the hyperparameter tuning was done. Results obtained are shown in Table 3.6.1.

Table 3.6.1: Evaluation of Bayesian models developed

	XG Boost w/o Feature Selection	XG Boost with Feature Selection	Random Forest w/o Feature Selection	Random Forest with Feature Selection
Train AUC	99.88 %	95.31 %	83.95 %	84.96 %
Test AUC	90.74 %	91.24 %	80.97 %	82.24 %

References

[1] Data Preprocessing: https://scikit-learn.org/stable/modules/preprocessing.html

[2] Serafeim Loukas, Everything you need to know about Min-Max normalization: A Python tutorial https://towardsdatascience.com/everything-you-need-to-know-about-min-max-normalization-in-python-b79592732b79

[3] Feature selection using SelectKBest: https://www.kaggle.com/code/jepsds/feature-selection-using-selectkbest/notebook

[4] M. A. Hearst, S. T. Dumais, E. Osuna, J. Platt and B. Scholkopf, "Support vector machines," in IEEE Intelligent Systems and their Applications, vol. 13, no. 4, pp. 18-28, July-Aug. 1998, doi: 10.1109/5254.708428.

[5] Optuna Examples: https://github.com/optuna/optuna-examples/blob/main/pytorch/pytorch_simple.py

[6] Understanding 1D and 3D Convolutional Network: https://towardsdatascience.com/understanding-1d-and-3d-convolution-neural-network-keras-9d8f76e29610

[7] Hyperparameter optimization in CNN for pytorch: https://github.com/elena-ecn/optuna-optimization-for-PyTorch-CNN

[8] Understanding AUC, ROC curves: https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5#:~:text=AUC%20%2D%20ROC%20curve%20is%20a,capable%20of%20distinguishing%20between%20classes.

[9] Decision Trees: https://scikit-learn.org/stable/modules/tree.html

[10] XGBoost, XGBoost Parameters: https://xgboost.readthedocs.io/en/latest/parameter.html

[11] Christophe Pere, What is XGBoost? And how to optimize it?: https://towardsdatascience.com/what-is-xgboost-and-how-to-optimize-it-d3c24e0e41b4

[12] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama. Optuna: A next-generation hyperparameter optimization framework. In Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining, pages 2623–2631, 2019.

[13] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. Journal of machine learning research, 13(2), 2012.