IBA

MACHINE LEARNING I

ASSIGNMENT II

INSTITUTE OF BUSINESS ADMINISTRATION - IBA

Classification Kaggle Competition

Submitted By:

Bilal Naseem - ERP ID: 13216

Date: April 20, 2023

Abstract

This report details the steps taken to prepare and analyze a dataset for a Kaggle Classification competition. Various algorithms, including Logistic Regrssion, KNN, Decision Tree, Random Forest, Gradient Boost, CatBoost, LightGBM, XGBoost, and Random Forest using cuML, along with feature selection and transformation techniques. 2 layer stacking with XGBoost as the meta model yielded the best results, with an **AUC of 0.70454** on the Kaggle leaderboard (**Rank 1**). The report provides insights into the performance of each algorithm and feature selection/transformation approach taken.

Instructions:

The second competition (Classification problem) has been posted on Kaggle. You can access it via the following link:
https://www.kaggle.com/t/ff4c8949133c4722b85d7822b06667f7
You are expected to participate in the competition individually and use your proper full name. This is important because many people have similar first names and it may create problems at the time of grading. Please submit a report in MS Word describing the data preparation steps you took and the algorithms you applied. Describe in sufficient details what works for you against which algorithm. There should be a section on your feature selection/transformation effort (be it using one-hot or label encoding, feature selection function in sklearn, selection using p-values, variance/correlation filters, lasso/ridge based feature selection, and/or anything else). You may eventually submit dozens of submissions in an attempt to become the leader on the leaderboard but I would like you to specify what worked for you against these algorithms:

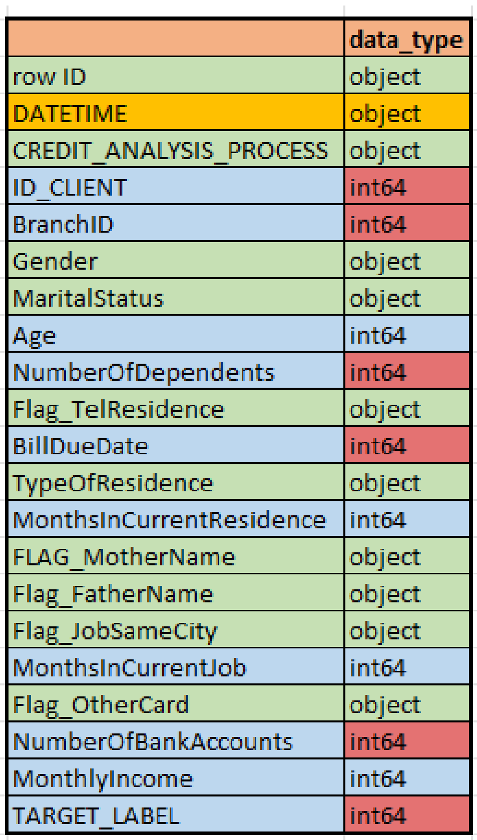
- Logistic Regression
- Decision Tree
- k-Nearest Neighbor
- Random Forest using sklearn
- Gradient Boosting using sklearn
- Boosting using XGBoost library
- Boosting using LightGBM library
- Random Forest using cuML (cuda ML that works on GPU)

For different variations of random forest and boosting listed above, compare them in terms of time taken by each implementation. Which one turned out to be the fastest. Those of you who may not have GPU in their PCs, I would encourage you to use Kaggle notebooks. The GPU is normally available and is super-fast.

Contents

1 Exploring Dataset & Feature Engineering

The training data set consisted of 70,000 rows with 21 columns, whereas the testing data had 30,000 rows and 20 columns. The data types of the columns of the dataframe were the following:



	data_type
row ID	object
DATETIME	object
CREDIT_ANALYSIS_PROCESS	object
ID_CLIENT	int64
BranchID	int64
Gender	object
MaritalStatus	object
Age	int64
NumberOfDependents	int64
Flag_TelResidence	object
BillDueDate	int64
TypeOfResidence	object
MonthsInCurrentResidence	int64
FLAG_MotherName	object
Flag_FatherName	object
Flag_JobSameCity	object
MonthsInCurrentJob	int64
Flag_OtherCard	object
NumberOfBankAccounts	int64
MonthlyIncome	int64
TARGET_LABEL	int64

Figure 1: Column datatypes

Datatypes of `ID_CLIENT`, `BranchID`, `BillDueDate`, `NumberOfDependents`, `NumberofBankAccounts`, `TARGET_LABEL` were changed from numerical to Categorical. Initial exploration of the data showed that there are 0 nulls in all columns of the dataframe, however in the `TypeOfResidence` column, a category was represented by a blank space. It was replaced with the letter "B" in order to avoid confusion after One-Hot Encoding. This was done using the following code:

```
training['TypeOfResidence'] = training['TypeOfResidence'].replace(' ', 'B')
```

1.1 Feature Extraction from DateTime Column

The `DateTime` Column which was initially of the object datatype was converted to datatime (datetime64[ns]) using `training['DATETIME'] = pd.to_datetime(training['DATETIME'])`.

1.1.1 Feature Extraction from Date

From date in the DATETIME column: Day of Year, Month of Year, Season and Year features were extracted.

Day of Year

The day of the year feature was extracted using the `np.sin` and `np.cos` methods, rather than using `pd.Series.dt.dayofyear`. The sine and cosine values are used to encode the day of the year as cyclic features, which can be useful in some contexts, such as in time series analysis. The code is present in the appendix.

Month of Year

Similarly, Month of year was also extracted using this method. The code for this is present in Appendix.

Season

Seasons feature was created which contained seasons: `Summer, Autumn, Winter, Spring`, corresponding to dates in the datetime column. The code is present in Appendix.

Year

Years feature was created using the DATETIME column using the code:

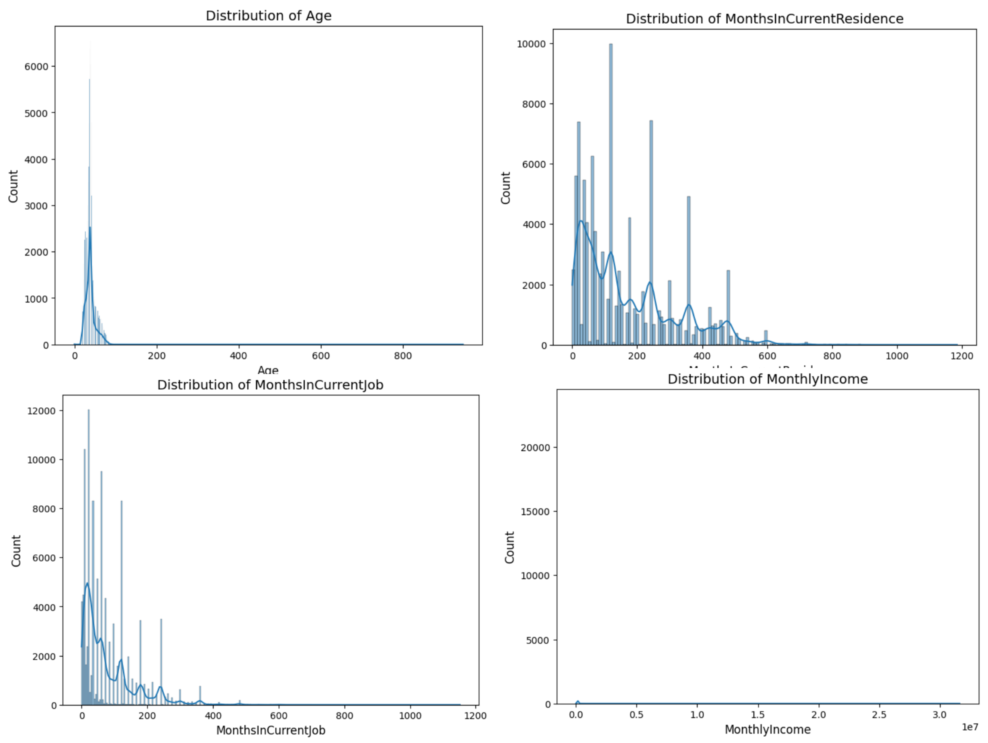
```
training['year'] = training['DATETIME'].dt.year
```

1.1.2 Feature Extraction from Time

Features of time were extracted from the datetime column at the hour, minute and second level using the np.sin and np.cos methods. As we only have 21 of columns in our dataframe and extracting the time at a higher level of granularity does not add too much computational cost or complexity, I found it better to extract the time at a more granular level. Code of this is present in Appendix.

1.2 Exploration of Numerical Features

Distribution of numerical columns were plotted which gave the following results:



Q-Q plots were plotted of these columns to get a closer look at the distributions of these columns.

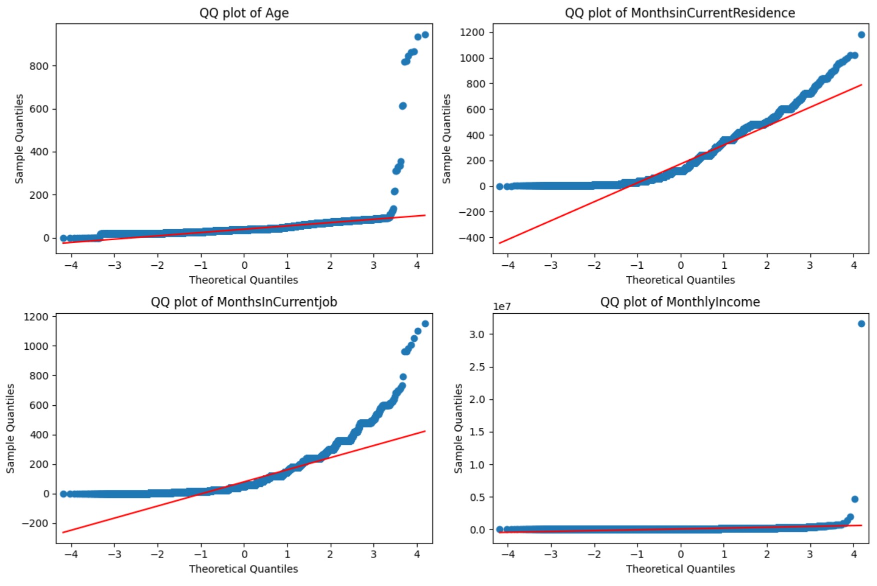


Figure 2: Q-Q plots of numerical columns

1.2.1 Mathematical Transformations on Numerical Columns

Multiple Mathematical transformations were applied on these numerical columns including log, sqaure root, and exponentiol, to make the distributions more Normal. However applying the `Yeo-Johnson Transformation` gave the best results (Code in Appendix). Q-Q plots after applying the transformation are as follows:

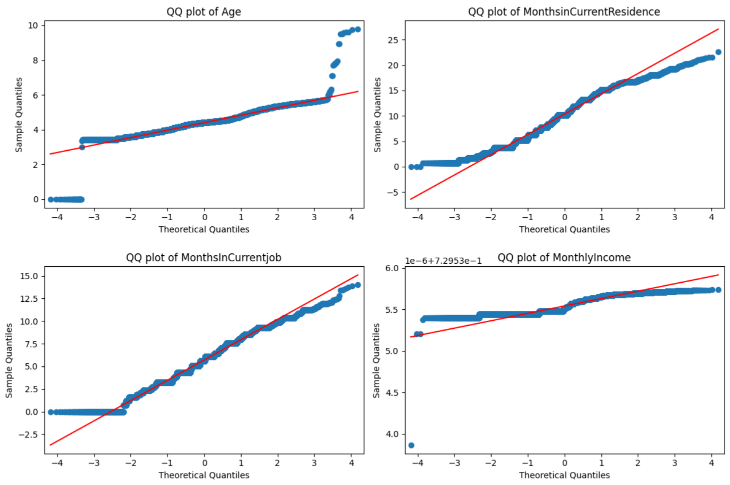


Figure 3: Q-Q plots of numerical columns after Yeo-Johnson Transformation

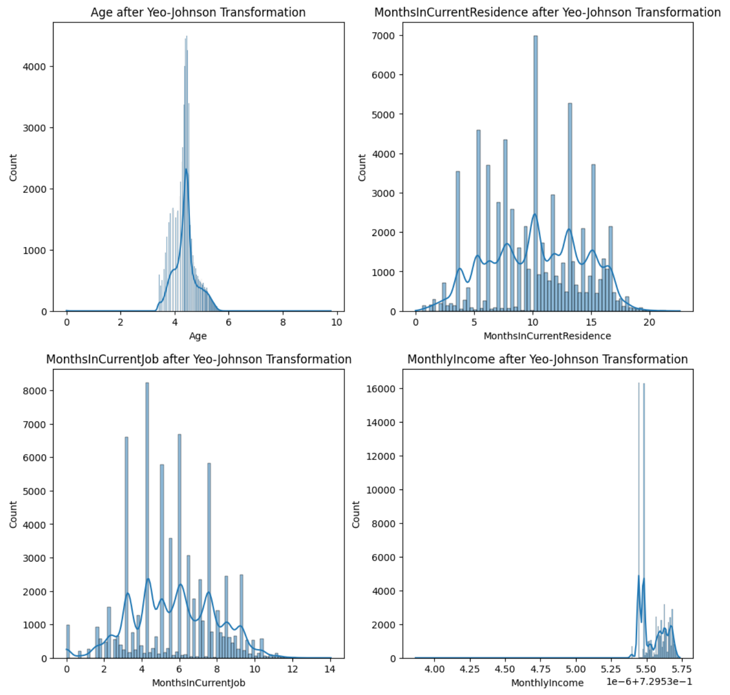


Figure 4: Distributions of numerical columns after Yeo-Johnson Transformation

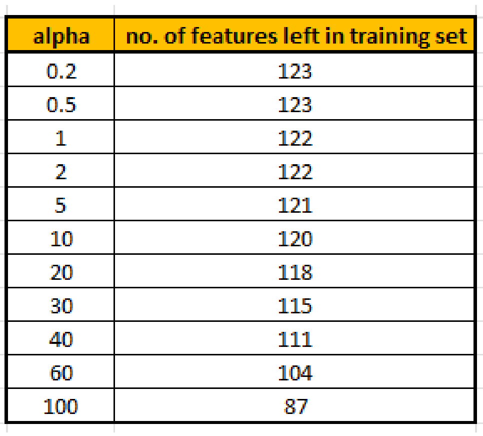
Next, Yeo-Johnson was applied on the entire Dataset using sklearn's `PowerTransformer` class. `transformer = PowerTransformer(method='yeo-johnson', standardize=True)`. The transformer was fitted on X_train and then applied on both X_train and X_test. Standard Scalee was also applied in the same stage.

1.3 One-Hot Encoding

The column `ID_CLIENT` was not One-Hot encoded as it contained all unique IDs and was dropped, so was the original `DATETIME` column. Training and tested datasets were concatenated before One-Hot Encoding which was done through get_dummies, after which they were sliced again.

1.4 Feature Reduction

Feature reduction was applied on the One-Hot encoded dataframe with the help of logistic regression with the L1 penalty, and multiple datasets were created by varying the value of α (0.2, 0.5, 1, 2, 5, 10, 20, 30, 40, 60, and 100). Code for this is present in Appendix. Multiple models were ran on these datasets along with the one without feature reduction (Naive Bayes and LightGBM). Number of features left after each value of alpha are shown in the figure below:



alpha	no. of features left in training set
0.2	123
0.5	123
1	122
2	122
5	121
10	120
20	118
30	115
40	111
60	104
100	87

Figure 5: Number of features left after each value of alpha

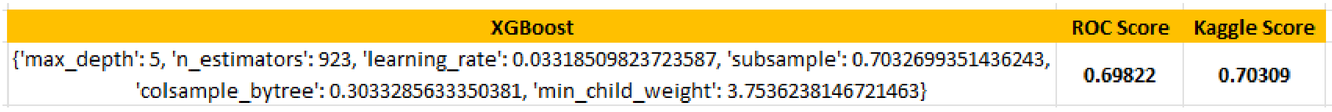
Multiple Models (Naive Bayes, LighGBM, XGBoost, Stacking) were run on these reduced feature datasets however no improvement in score was observed through feature reduction.

2 Results from Boosting and Random Forest Models

After the preparation of the dataset, Boosting and Random Forest algorithms were applied initially, as they are known to give better results.

2.1 XGBoost

Multiple iterations of `GridSearch` and `RandomizedSearch` were tried out, to find the optimal hyperparameters. Due to computational constraints GridSearch wasn't completed for any model, and so `Bayesian Search` was performed which consistently gave better results than Randomized Search. The following approach gave the best results on the kaggle scoreboard: **Bayesian Search with RepeatedStratifiedKFold (n_splits=5, n_repeats=3) with Early Stopping Rounds = 10.**
Also the score (mean roc_auc) using RepeatedStratifiedKFold approach, consistently matched the score on the Kaggle leaderboard with the accuracy of 0.01. Early stopping rounds were increased to 20 and 40 but gave no improvement in the score.
Bayesian search was performed using `scikit-opt` library with 50 iterations and 50 random starting points. The the code is present in the appendix. The following are the best hyperparameters and Kaggle submission score:



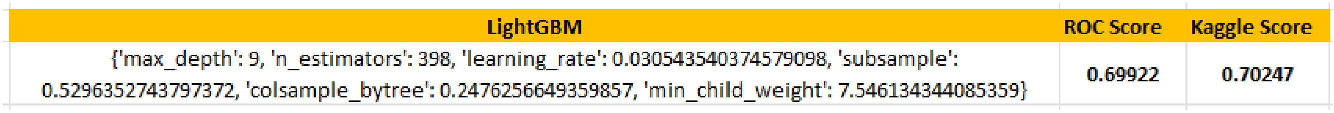
XGBoost	ROC Score	Kaggle Score
{'max_depth': 5, 'n_estimators': 923, 'learning_rate': 0.03318509823723587, 'subsample': 0.7032699351436243, 'colsample_bytree': 0.3033285633350381, 'min_child_weight': 3.7536238146721463}	**0.69822**	**0.70309**

Figure 6: Best Hyperparameters of XGBoost

This approach was also carried out on the reduced features datasets but did not give any improvement in the score. For subsequent sections the same approach was adopted for hyperparameter optimization.

2.2 LightGBM

Multiple runs of Bayesian optimization was performed on LightGBM with 100 iterations with early stopping rounds of 10, 20 and 40. The following hyperparameters gave the best results:

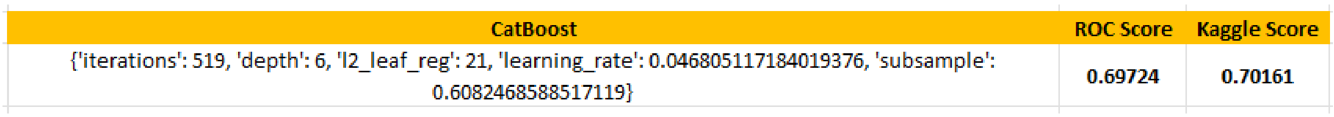


LightGBM	ROC Score	Kaggle Score
{'max_depth': 9, 'n_estimators': 398, 'learning_rate': 0.030543540374579098, 'subsample': 0.5296352743797372, 'colsample_bytree': 0.2476256649359857, 'min_child_weight': 7.546134344085359}	**0.69922**	**0.70247**

Figure 7: Best Hyperparameters of LightGBM

2.3 CatBoost

Multiple runs of Bayesian optimization was performed on CatBoost with 100 iterations with early stopping rounds of 10, 20 and 40. The following hyperparameters gave the best results:

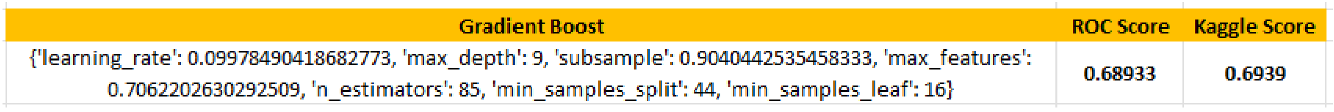


CatBoost	ROC Score	Kaggle Score
{'iterations': 519, 'depth': 6, 'l2_leaf_reg': 21, 'learning_rate': 0.046805117184019376, 'subsample': 0.6082468588517119}	**0.69724**	**0.70161**

Figure 8: Best Hyperparameters of CatBoost

2.4 Gradient Boost

Early stopping rounds in GradientBoost was implemented using `no_iter_no_change`, n=20, and the following best hyperparameters were obtained using Bayesian search:

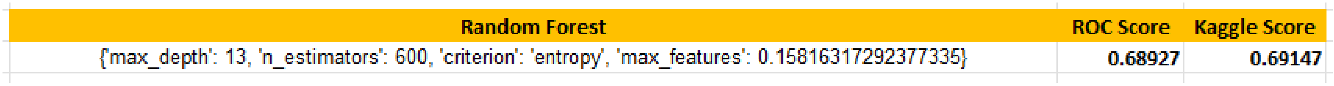


Gradient Boost	ROC Score	Kaggle Score
{'learning_rate': 0.09978490418682773, 'max_depth': 9, 'subsample': 0.9040442535458333, 'max_features': 0.7062202630292509, 'n_estimators': 85, 'min_samples_split': 44, 'min_samples_leaf': 16}	**0.68933**	**0.6939**

Figure 9: Best Hyperparameters of GradientBoost

2.5 Random Forest

The following best hyperparameters were obtained of Random Forest with Bayesian Search. 15 iterations were performed which took 10 hours to complete.



Random Forest	ROC Score	Kaggle Score
{'max_depth': 13, 'n_estimators': 600, 'criterion': 'entropy', 'max_features': 0.15816317292377335}	**0.68927**	**0.69147**

Figure 10: Best Hyperparameters of Random Forest

2.5.1 Random Forest using cuML

Random Forest was also run using cuML library with the same hyperparameters of that regular Random Forest. The model fitted to the training dataset in under 2 minutes as compared to regular Random Forest which took near 4 minutes. This model gave a Kaggle submission score of 6.90234.

2.6 Comparison of Results

Performance of all models of the previous 5 subsections were assessed by running them on the training dataset with RepeatedStratifiedKFold with 10 splits and 3 iterations. The results were plotted on a boxplot along with actual Kaggle submission scores (Red Diamonds).

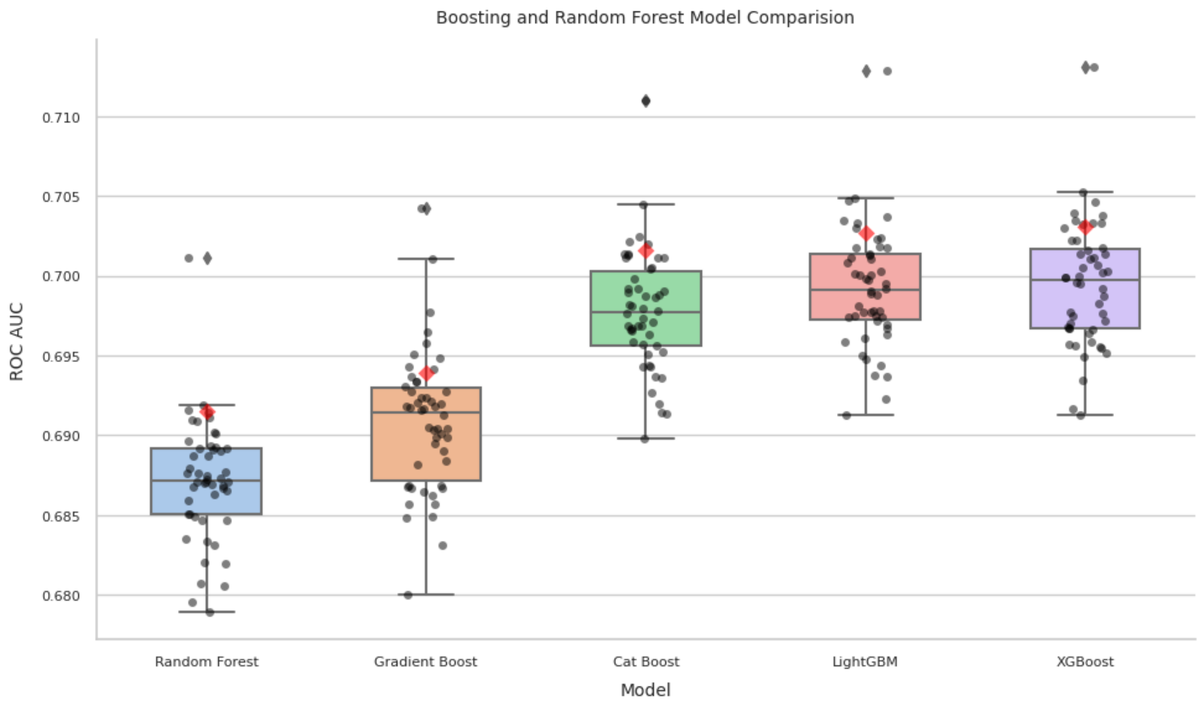


Figure 11: Comparision of Results of Boosting and Random Forest Models

The results show that CatBoost, LightGBM and XGBoost give better results on both training and testing datasets.

3 Results from Simpler Models

3.1 Naive Bayes Classifier

GridSearch was applied on Naive Bayes with RepeatedStratifiedKFold (n_splits=10, n_repeats=10) which gave the best hyperparameter of `'var_smoothing': 1.0`. AUC score on dataset was 0.63154 and Kaggle submission score was **0.6328**.

3.2 K-Nearest Neighbors (KNN)

Bayesian Search was applied on KNN model, and the following optimal hyperparameters were obtained: `'n_neighbors': 17, 'weights': 'distance'`. AUC Score on training dataset was 0.6116, and Kaggle submission score was **0.62274**.

3.3 Logistic Regression

Bayesian Search was applied on the Logistic Regression model, and the following optimal hyperparameters were obtained: `'C': 1.0, 'penalty': 'l2', 'solver': 'liblinear'`. AUC Score on training dataset was 0.6809, and Kaggle submission score was **0.68125**.

3.4 Decision Trees

Bayesian Search was applied on the Decision Tree model, and the following optimal hyperparameters were obtained:
`'max_depth':7,'min_samples_split':14,'min_samples_leaf':9,'criterion':'entropy','splitter':'random'`.
AUC Score on training dataset was 0.65623, and Kaggle submission score was **0.63802**.

3.5 Comparision of Results

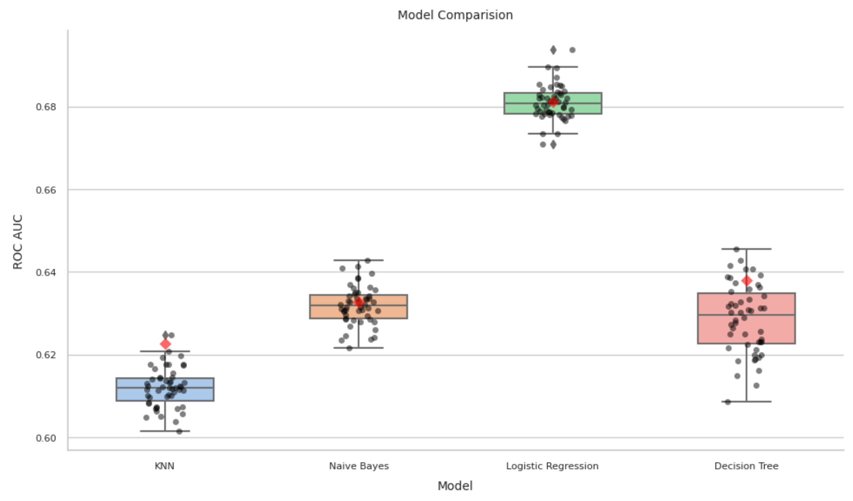


Figure 12: Comparision of Results of Simple Classification Models

4 Results from Voting and Stacking

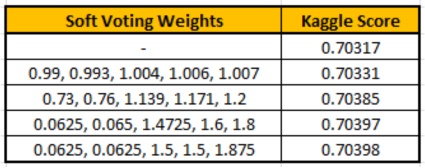
4.1 Voting

4.1.1 Weighted Average

Weighted average voting is used when we want to assign more weight to a particular model based on its performance or domain knowledge. Weighted average of all models of chapter 2 with their best hyperpameters was taken. Using equal weights of all models (0.2) gave a score of **0.69855** on Kaggle. However Normalizing the scores (0.99, 1.007, 1.004, 1.006, 0.993) gave a score of **0.70264** on Kaggle.

4.1.2 Soft Voting

Soft voting is used when the models produce probability estimates, and we want to take the average probability of each class to decide the final output. Soft Voting was carried out of all models of chapter 2 with their best hyperpameters. `EnsembleVoteClassifier` of the `mlxtend` library was used. Results of various runs of soft voting can be found in the table below:



Soft Voting Weights	Kaggle Score
-	0.70317
0.99, 0.993, 1.004, 1.006, 1.007	0.70331
0.73, 0.76, 1.139, 1.171, 1.2	0.70385
0.0625, 0.065, 1.4725, 1.6, 1.8	0.70397
0.0625, 0.0625, 1.5, 1.5, 1.875	0.70398

Figure 13: Results from Soft Voting

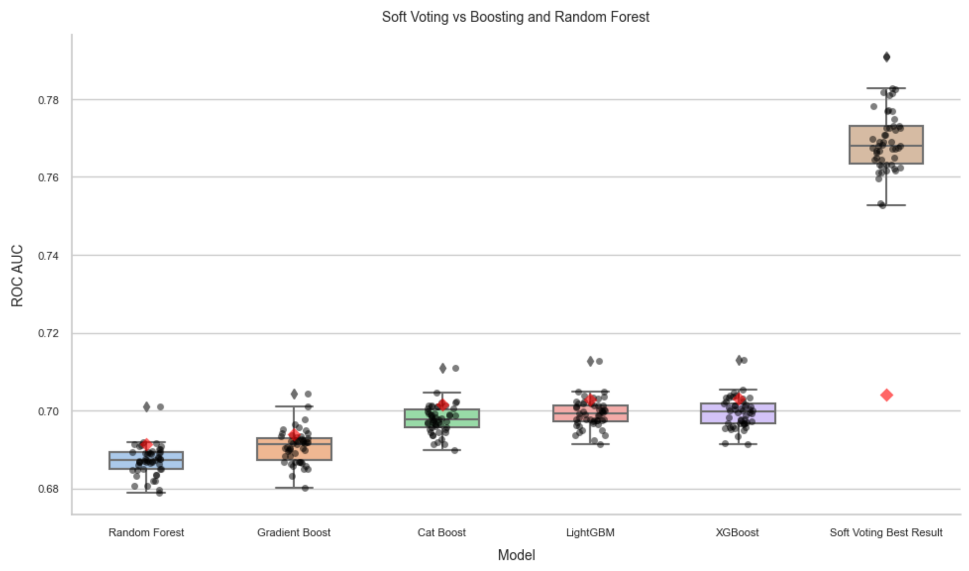


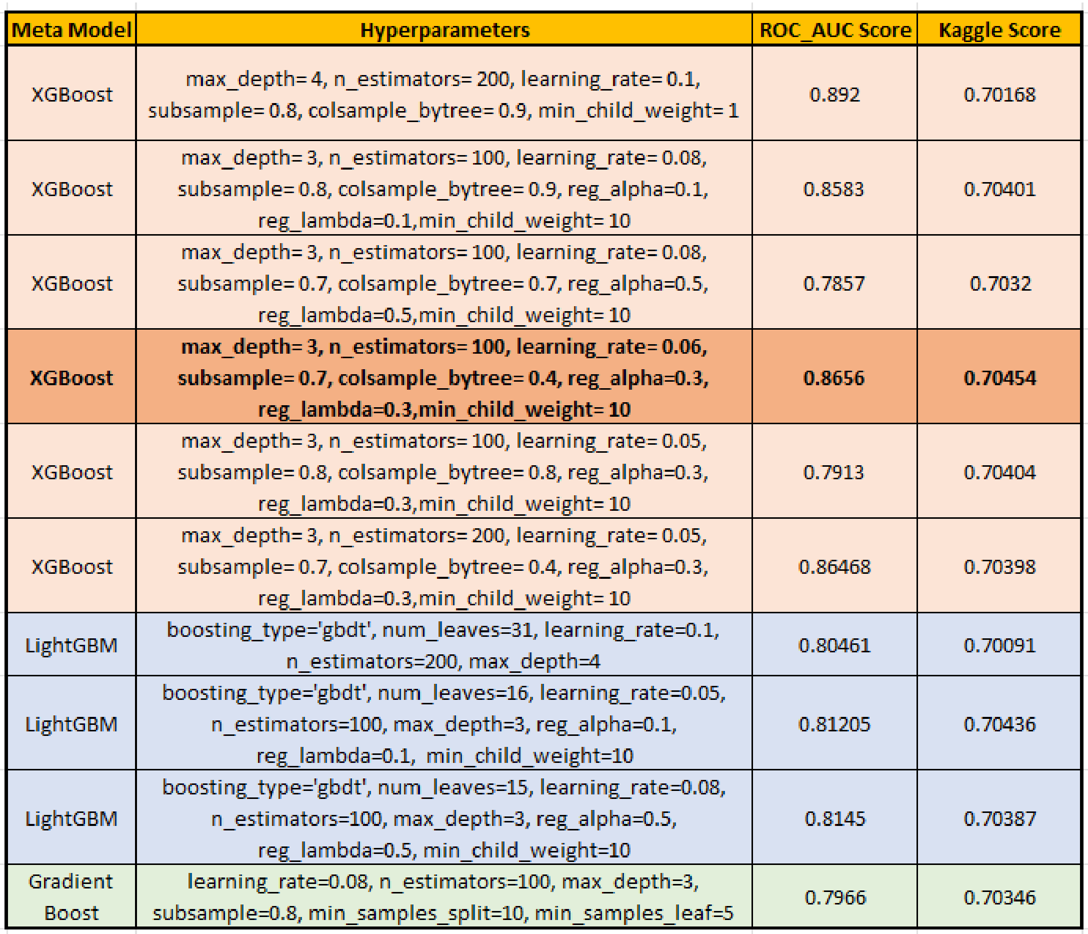
Figure 14: Soft Voting vs Boosting and Random Forest

4.2 Stacking

All of all 9 models of Chapter 2 and 3 with their best hyperparameters were stacked. `StackingCVClassifier` of the `mlxtend` library was used. Stacking was done using 2 layers and 3 layers, and in both roc_auc on training data was reaching above 0.9, and through regularization in meta-models overfitting was reduced.

4.2.1 2 Layer Stacking

2 layer stacking was done with base models set as all 9 models of chapter 2 and 3 with their best hyperparameters. LightGBM, CatBoost and XGBoost were tried as meta-models. Multiple runs were done by setting each of them as a meta-model by changing hyperparameters. Meta-model of XGBoost gave the best result. 10 Fold CV was tried but didnt give any improvement in score. Adding regularization in the meta-model showed improvement in score and reduced overfitting. Some of the results are in the table below:

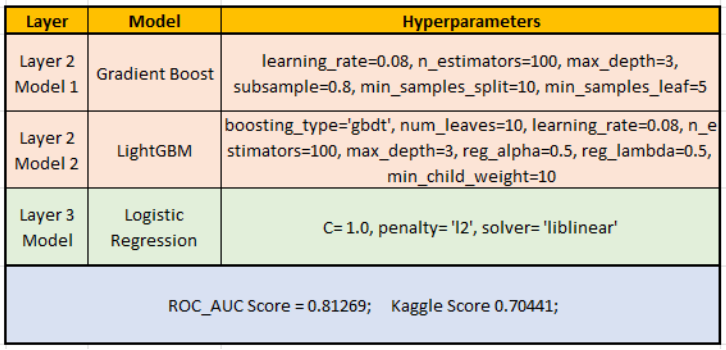


Meta Model	Hyperparameters	ROC_AUC Score	Kaggle Score
XGBoost	max_depth= 4, n_estimators= 200, learning_rate= 0.1, subsample= 0.8, colsample_bytree= 0.9, min_child_weight= 1	0.892	0.70168
XGBoost	max_depth= 3, n_estimators= 100, learning_rate= 0.08, subsample= 0.8, colsample_bytree= 0.9, reg_alpha=0.1, reg_lambda=0.1,min_child_weight= 10	0.8583	0.70401
XGBoost	max_depth= 3, n_estimators= 100, learning_rate= 0.08, subsample= 0.7, colsample_bytree= 0.7, reg_alpha=0.5, reg_lambda=0.5,min_child_weight= 10	0.7857	0.7032
XGBoost	**max_depth= 3, n_estimators= 100, learning_rate= 0.06, subsample= 0.7, colsample_bytree= 0.4, reg_alpha=0.3, reg_lambda=0.3,min_child_weight= 10**	**0.8656**	**0.70454**
XGBoost	max_depth= 3, n_estimators= 100, learning_rate= 0.05, subsample= 0.8, colsample_bytree= 0.8, reg_alpha=0.3, reg_lambda=0.3,min_child_weight= 10	0.7913	0.70404
XGBoost	max_depth= 3, n_estimators= 200, learning_rate= 0.05, subsample= 0.7, colsample_bytree= 0.4, reg_alpha=0.3, reg_lambda=0.3,min_child_weight= 10	0.86468	0.70398
LightGBM	boosting_type='gbdt', num_leaves=31, learning_rate=0.1, n_estimators=200, max_depth=4	0.80461	0.70091
LightGBM	boosting_type='gbdt', num_leaves=16, learning_rate=0.05, n_estimators=100, max_depth=3, reg_alpha=0.1, reg_lambda=0.1, min_child_weight=10	0.81205	0.70436
LightGBM	boosting_type='gbdt', num_leaves=15, learning_rate=0.08, n_estimators=100, max_depth=3, reg_alpha=0.5, reg_lambda=0.5, min_child_weight=10	0.8145	0.70387
Gradient Boost	learning_rate=0.08, n_estimators=100, max_depth=3, subsample=0.8, min_samples_split=10, min_samples_leaf=5	0.7966	0.70346

Figure 15: Results from 2 Layer Stacking

4.2.2 3 Layer Stacking

3 layer stacking was done by setting all 9 models of chapter 2 and 3 as base models, 2 models were used in layer 2, and a single model in layer 3. LightGBM, Gradient Boost, XGBoost, and Logistic Regression were tried as layer 2 and 3 models. The best results were achieved by setting LightGBM and Gradient Boost in layer 2, and Logistic Regression in Layer 3. Numerous runs were made, the combination which gave the best result is shown in the figure below:



Layer	Model	Hyperparameters
Layer 2 Model 1	Gradient Boost	learning_rate=0.08, n_estimators=100, max_depth=3, subsample=0.8, min_samples_split=10, min_samples_leaf=5
Layer 2 Model 2	LightGBM	boosting_type='gbdt', num_leaves=10, learning_rate=0.08, n_estimators=100, max_depth=3, reg_alpha=0.5, reg_lambda=0.5, min_child_weight=10
Layer 3 Model	Logistic Regression	C= 1.0, penalty= 'l2', solver= 'liblinear'
ROC_AUC Score = 0.81269; Kaggle Score 0.70441;		

Figure 16: Results from 3 Layer Stacking

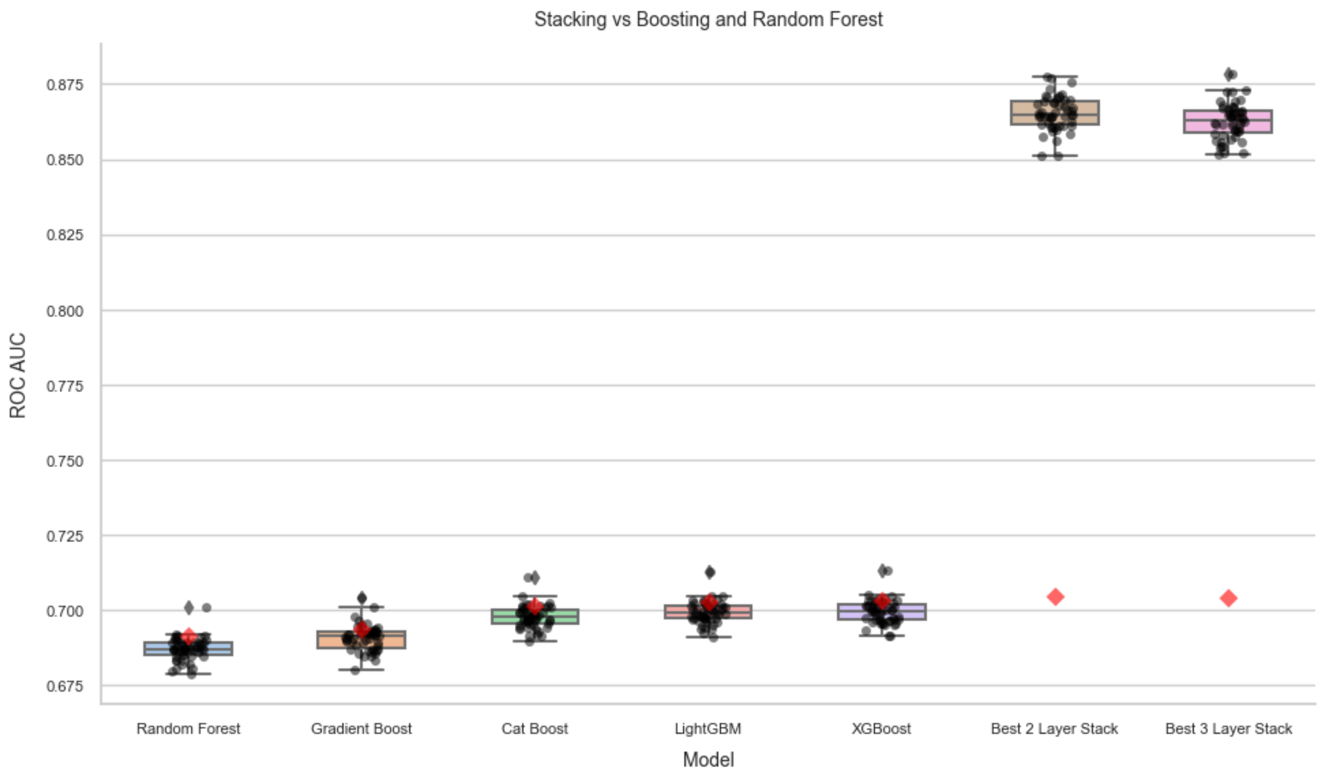


Figure 17: 2 and 3 Layer Stacking vs Boosting and Random Forest

5 Conclusion

Initially in the dataset column types were corrected, new features were extracted from the DATETIME column, mathematical transformation was applied, and then One-Hot Encoding was applied. Feature Reduction was done using Logistic Regression via L1 penalty but gave no improvement in score. 9 Machine learning models were applied after each one's hyperparameter was optimized. Finally through Voting and Stacking the model was improved. Results of all models applied on the dataframe and their Kaggle submission score can be found in the figure below:

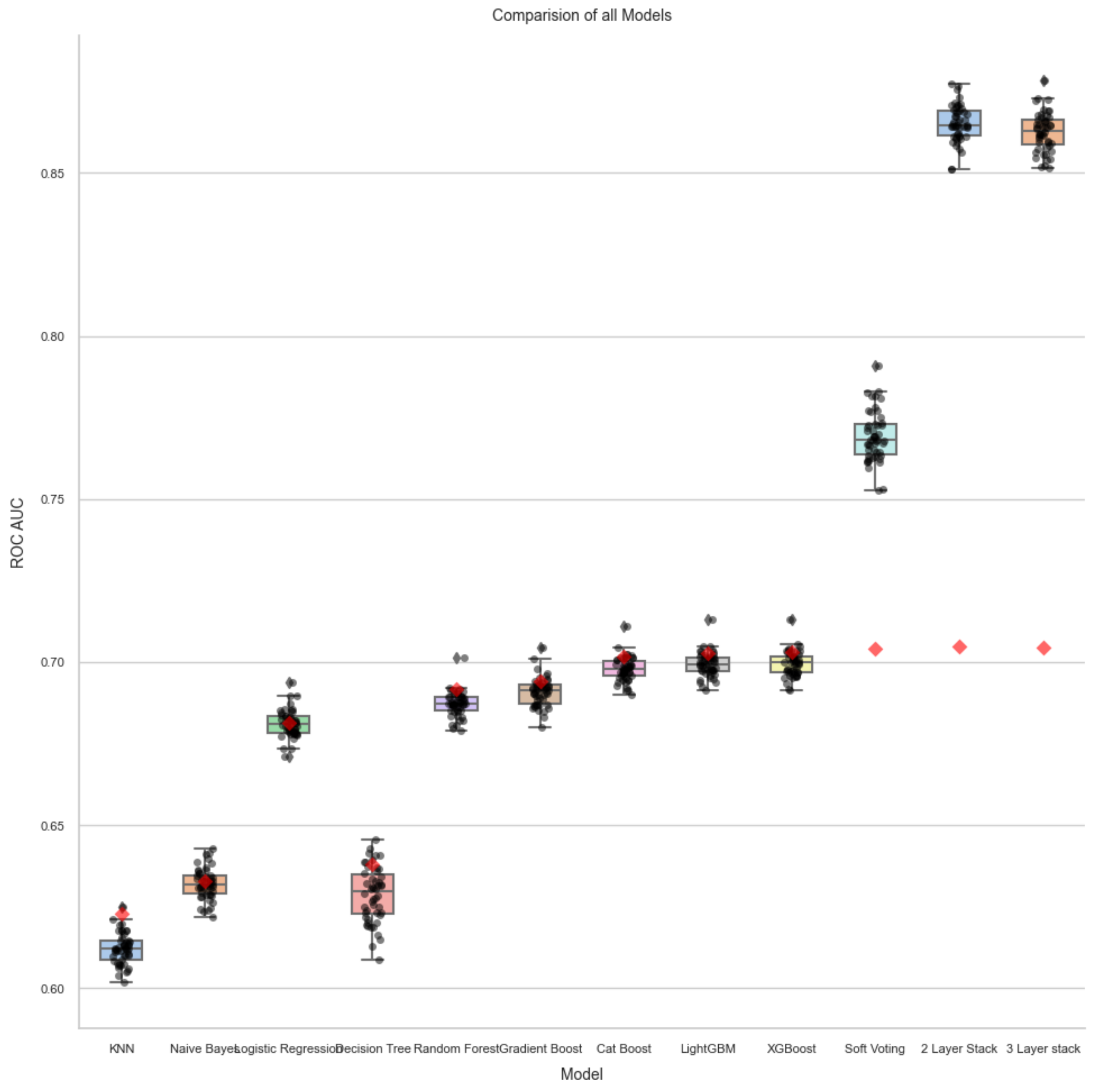


Figure 18: Comparison of all Models