Predicting the Chance of Insurance Claims Using Classification Techniques and Neural Networks

ESOF-0151 Term Project – Undergraduate

Aidan Johnston, Angel Martinez and Christopher Silver
{ajjohnst, aamarti1, crsilver}@lakeheadu.ca

***Abstract*—When it comes to machine learning algorithms, there are a variety of different models that can be used to solve machine learning problems. Specifically, when it comes to classification problems, the models that are commonly used are neural network, random forest (RF), XGBoost, LGBMRegressor and support vector machine model to name a few. The key to success in solving machine learning problems lies in selecting the best model with the highest accuracy, and in our project we aim to do this. To select the best model, we first took the time to understand the models themselves, what they succeed in and where they fail to perform well; this knowledge is obtained through the various literature mentioned in this paper. From here we implemented the models and used various preprocessing techniques to select the best model. The most successful model we implemented was a stacking algorithm that used 2XGBoost and 2LGBMRegressor models which helped us attain a score of 0.77404 which is only 0.04371 off the highest score for the entire Kaggle competition of 0.81775.**

I. Introduction

In our project we are working to predict whether or not an insurance customer made a claim upon an insurance policy. This is a binary classification problem. Our target variable is "claim" which has a range from 0 (no claim) to 1 (claim).

This project is unique compared to other data science projects because the input features have been anonymized. Rather than having information about customers, where they live, what they drive, etc. We are given a list of 118 features named f1, f2, f3, etc. and a target variable claim. Based on this fact, it may seem like this project is not very useful, since this would not have a direct real-world application since in real life you know what input features you have. Rather, we are approaching this project as a way where we can improve our skills as data science students. If we are able to make an accurate prediction without any features, we will be able to only improve our model once we understand what the input features are. We would be able to use feature engineering to combine similar features, we could eliminate features based on our knowledge of the domain, we could normalize data more effectively if we understand the nature of the attributes and know the normal range of data. We could also identify outliers more easily. In this project we are focusing on using our knowledge of data science to improve the model as much as we can using experimental and theoretical methods we have learned, and then we would be able to improve our model using our knowledge of the domain. In order to be successful in this project, we have to understand and implement many core concepts from Big Data as we won't be able to use any prior knowledge from the domain.

Once this project has been completed, we will have mastered a lot of data science concepts and this will allow us to make more accurate predictions in any environment or setting. The information in the train and test data sets are also real data collected from insurance companies. Once we create accurate predictions we would be able to approach an insurance company, show them what we have done, and entice them into hiring us to do the same with real labelled data.

Being able to predict whether a client will make an insurance claim is important for insurance companies. Although these companies protect individuals in case of an accident, they are a business and their focus is on making money. There are many insurance companies all competing for the same amount of customers. If a company offers insurance at too low of a rate, they may lose money if clients make more claims than they pay into. If a company offers insurance at too high of a rate, customers will choose a different insurance broker. Being able to predict whether someone will make a claim will allow them to adjust their prices to increase their market share (get the most amount of customers) while making an overall profit.

II. Literature Review

A. Model Structure and Selection

The literature paper *Pattern Recognition* [1] covers solving data mining problems through the use of random forest (RF), which is a technique for classification problems. In this paper, they created an RF model with the use of a bootstrap sample of the training data set while ensuring that each tree of the random forest reached maximum depth and randomization was used. This was done to get a low bias and low correlation which is important when it comes to the accuracy of the RF model. When performing tests this study mainly focused on variable importance, it also needs to be noted that the data set used was much smaller in comparison to the dataset we will be using. Overall, this study managed to solve multiple issues regarding the random forest model, one being the problem of overfitting. The paper suggested increasing the size of the trees in the RF as well as when selecting the features used for splitting a node it should not be set to a default value instead

Aidan Johnston, Angel Martinez and Christopher Silver are with the Department of Software Engineering, Lakehead University, Thunder Bay, ON, Canada.

through testing, the value should be changed until overfitting is reduced or removed completely. While this study deals with solving classification problems using the RF model, in our solution we focus on not only an RF model but rather have multiple models that we will compare to obtain the model that will result in the greatest accuracy.

To obtain a better understanding of what models could be used in order to solve the classification problem at hand we have reviewed an article by Shahab, Brain, and Dongmei [2] that focuses on comparing different models based on object-based urban land cover classification. The problem that this paper tries to solve is different from our problem but reading their findings can help guide us in what models we should give more attention to. Their finding concluded that multilayer perceptron (MLP) was the most accurate model overall, but the paper also covered other models such as the random forest model (RF), XGB model, and support vector machine model (SVM). With this information, we concluded that it is worthwhile to implement both a XGB model and a SVM model since both of these models are commonly used when solving various classification problems. This paper also allowed us to gain knowledge about the parameters that we should focus on when training the XGB model. For the XGB model, the two parameters that are of importance when it comes to increasing the speed and accuracy are the number of trees, learning rate, and depth of each tree. It was also noted that the SVM can perform well even when there are few amounts of training samples, this is useful information because we can train the SVM model on a small amount of data at first to see if the model is worthwhile rather than going through a long wait time to find out the model is not accurate enough when compared to others.

The paper by Saso, and Bernard [3] compare the combination of classifiers with stacking and compare it to simply selecting the best one. In this study, they used two stacking methods, those being the probability distribution (PDs) and multi-response linear regression (MLR) with only a slight modification made to these methods. A few conclusions were made, the first being that stacking with a multi-response model tree outperforms stacking with multi-response linear regression. IT was also found that MLR performs better on datasets that have two-class datasets rather than multi-class datasets. Finally, when comparing stacking to selecting the best classifier, it was found that stacking was at best equal to selecting the best classifier. With this information, we can see that stacking can be useful but does not necessarily increase performance but we need to keep in mind that different datasets might give different results. So, for this reason, we will be trying to do stacking and since our dataset is a two-class dataset, we can use either PDs or MLR stacking methods to achieve stacking.

B. Kaggle Disscusion Posts

We also searched on the kaggle site for suggestions made by other competitors who shared their models, what worked for them, and what they have learned from this project. A lot of the discussion posts were not meaningful and didn't push our knowledge any further, however some discussions brought us some insight that will help us with our models. Initially, we wanted to spend a lot of time fine-tuning the correlation feature as we figured that since the features were anonymized, feature engineering would be more difficult, and we would use correlations to help find connections with the data. We got the correlation between every input variable and the target variable and found that there was very low correlation in almost every input variable, as seen in figure 1. We weren't sure if we did something wrong but upon consulting [4] we were reassured that correlations are quite low and not very meaningful in this setting. We will still use correlations, but we won't spend nearly as much time fine-tuning them now after seeing this discussion.

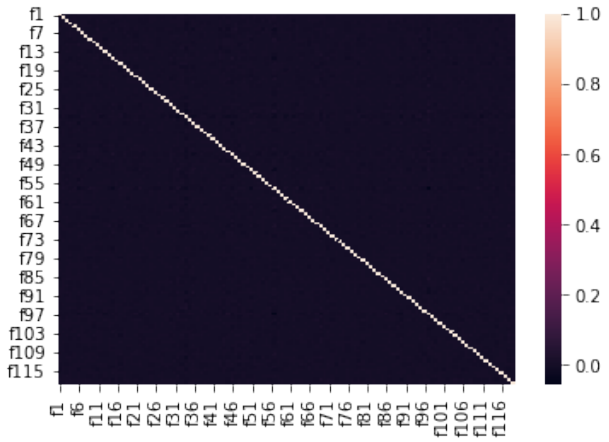

Fig. 1. Correlation matrix for all features in data set, all features have extremely low correlation.

Upon analyzing the data we initially noticed there were a lot of NaN values. Upon realizing the sheer amount of NaNs, we hypothesized that these NaN's were actually meaningful; however we did not know what to do with this data. Upon reading [5] we saw the idea that people have been counting the number of NaNs in each row, and then appending this information as a feature in the dataset. The author of this discussion noted how doing this increased his score by 2%. This is quite significant since our initial models are getting about 77% accuracy and the highest accuracy on the leaderboard is less than 82%. We implemented this feature in our code, but we have not yet tested it on a model.

When considering what models we would like to use, we hypothesized that a neural network might be quite effective in this application because of the sheer amount of data. We know that neural networks usually thrive in Big Data environments with lots of data points. Upon examining [6], we realized that using neural networks may not be the best solution. Users on this discussion post talked about only getting accuracy around 55% to 60%, and people who were using highly sophisticated models only got around 80%.

As a group we don't have much experience with model stacking so we were quite intrigued when we saw a discussion post from someone who achieved second place on the leaderboard [7]. We analyzed his block diagram and gained

the understanding that he used the base models of CatBoost, XGBoost, LGBMRegressor, HistGradientBoostingClassifier, TabNet, and a Generalized Additive Model (GAM), and then performed Elastic-Net Regression on these models to come up with the output. This gives us an idea on some models to try and introduces us to stacking. Stacking works well because all models have some strength and some weakness. Being able to stack multiple models together means that we will be able to harness the strengths in multiple models while trying to suppress the weaknesses in each as well. This is something that we will try moving forward.

h model stacking so we were quite intrigued when we saw a discussion post from someone who achieved second place on the leaderboard [7]. We analyzed his block diagram and gained the understanding that he used the base models of CatBoost, XGBoost, LGBMRegressor, HistGradientBoosting-Classifier, TabNet, and a Generalized Additive Model (GAM), and then performed Elastic-Net Regression on these models to come up with the output. This gives us an idea on some models to try and introduces us to stacking. Stacking works well because all models have some strength and some weakness. Being able to stack multiple models together means that we will be able to harness the strengths in multiple models while trying to suppress the weaknesses in each as well. This is something that we will try moving forward.

There was another discussion post from someone who finished 9th in the competition [8]. They explained how they approached stacking quite well and explained what did and what didn't work. They had 2 levels: level-0 and level-1. Level-0 models are base models where cross-validations are used. The "out-of-fold" predictions from level-0 would then be used to train a level-1 model which created the final prediction. This user explained that linear regression worked better than logistic regression as a level 1 model. They also experimented with creating a third level (level-2) which used both level-0 and level-1 models as inputs. This worked well. They also mentioned some things they tried that didn't help as much. They said that XGBoost and LightGBM models did not work well as a level-1 model. They also tried to incorporate many models into level 0 to stack as many models as possible, however some of these models had extremely low scores (Logistic Regression, RF) that only hindered the performance of the final model. We can use this information to try to implement stacking ourselves. It also gives us an idea that linear regression worked great, while XGBoost, LightGBM, Logistic Regression, and RF may not be the best choice for this project. Regardless, we will test these to be sure.

C. Hyper-Parameter Optimization

When looking to implement suggested models such as CatBoost, XGBoost, LGBMRegressor, HistGradientBoosting-Classifier, TabNet, and Generalized Additive Model (GAM), selecting the correct hyper-parameters for each is an important task. The chapter *Hyper-Parameter Optimization* [9] from A*utomated Machine Learning: Methods, Systems, Challenges* suggests and compares several hyper-parameter tuning algorithms. This includes methods such as grid search, random search, and Bayesian Optimization. Grid search involves defining a set of values for hyper-parameters, and then evaluates each combination of these sets. Grid search suffers from the fact that the number of evaluations required grows exponentially with each added set [9]. The chapter also discusses Random Search, which evaluates random combinations of our hyper-parameter set. Random Search does not suffer from the exponential growth of Grid Search; however, while searching a larger number of combinations Random Search will not be able to cover the whole domain of possible set combinations. Finally the chapters also suggests Bayesian Optimization, a hyper-parameter optimization, the method uses Bayes theorem to determine parameter values for model in order to maximize accuracy. We plan on using Bayesian Optimization for each model to determine the best hyper-parameters.

While picking methods for optimizing hyper-parameters is important, selecting the correct range of values for each parameters search space is also important. Different models have different parameters and each has a different healthy range of inputs. Looking at some of the major models we will be investigating: XGBoost, TabNet, and CatBoost; the paper *Tabular Data: Deep Learning is Not All You Need* [10] attests to several of these models for engaging in deep learning. We will be starting with the search spaces they used for each model.

1) CatBoost:

- Learning rate: Log-Uniform distribution [e^-5, 1]
- Random strength: Discrete uniform distribution [1, 20]
- Max size: Discrete uniform distribution [0, 25]
- L2 leaf regularization: Log-Uniform distribution [1, 10]
- Bagging temperature: Uniform distribution [0, 1]
- Leaf estimation iterations: Discrete uniform distribution [1, 20]

2) XGBoost:

- Eta: Log-Uniform distribution [e^7, 1]
- Max depth: Discrete uniform distribution [1, 10]
- Subsample: Uniform distribution [0.2, 1]
- Colsample bytree: Uniform distribution [0.2, 1]
- Min child weight: Log-Uniform distribution [$e^1$6, e^5]
- Alpha: Uniform choice 0, Log-Uniform distribution [e^{16}, e^2]
- Lambda: Uniform choice 0, Log-Uniform distribution [e^{16}, e^2]
- Gamma: Uniform choice 0, Log-Uniform distribution [e^{16}, e^2]

3) TabNet:

- Feature dim: Discrete uniform distribution [20, 60]
- Output dim: Discrete uniform distribution [20, 60]
- n steps: Discrete uniform distribution [1, 8]
- bn epsilon: Uniform distribution [e^5, e^1]
- relaxation factor: Uniform distribution [0.3, 2]

III. The Proposed Model

A. Data Preprocessing

A big part of preprocessing for this dataset revolves around properly handling the extremely large around of NA values.

We use techniques such as creating an NA count feature as well as a custom Smart Imputer for removing NA values. The following methods were used for data preprocessing and feature engineering.

1) NA Count: As seen in the literature review as well as our initial look at the NA encoded TSNE diagram, the NA values are just as much an important piece of information as regular features. To harness this we create a feature NACount.

2) Smart Imputer: Because of the large amount of NA values, properly imputing required more than just replacing our data with the mean or median. Rather, an approach of replacing features NA values selectively with different imputing methods was taken.

Currently, if a feature has a skewness higher than our defined threshold, we impute our NA values using the features mean, if it is below, we use the median. We used five fold grid search from a threshold of 0 to 1 on a default randomForest model to optimize the threshold. Determining this threshold is discussed later in the experimental analysis.

3) MinMaxScaler: We use the MinMaxScaler from Sklearn to prepare the dataset for being unskew. The MinMaxScaler clamps our dataset between 0 and 1, effectively removing negative values which certain unskewing methods such as sqrt() and boxcox() struggle to handle.

4) unSkew: Like the Smart Imputer, we aimed to selectively assign methods to each feature. If a feature has a skewness above a certain threshold, we apply the unskew method to it. If the skewness gets worse after the method is applied we revert it to before it was unskewed. We will discuss determining the unskew threshold later in the experimental analysis.

5) NAEncoder: The NA encoder is simple, it encodes any NA values as 1 otherwise 0. We do this to pass its output to the PCA and TSNE for feature creations.

6) PCA: After receiving an NA ecoded dataset, we pass these values into a PCA. It's output, as previously seen in figure 8 is passed to be clustered. We use Kmean clustering to group the 15 "slashes". Figure 2 shows the clustering. These clusters are one hot encoded.

7) TSNE: Like the PCA, the TSNE is passed a dataset that has been NA encoded. The tsne produces two features, tsne1, and tsne2. We do not cluster the output of the tsne. A figure of the output of the tsne can be seen in figure 3.

8) ANOVA: We will be using anova to reduce the total amount of features in our dataset before training. Currently this method is not yet implemented and all features are being used. We plan to use SKLearns SelectKBest to select k amount of features to use for our model. Similar to the smartImputer and the unskew methods, we will use a five fold grid search to search each possible k value. We will use the randomForest classifier.

B. Dataset Formation

1) Train Test Set Split: When it comes to splitting the dataset we decided on a 70% training and 30% testing split. This is a standard way to split the data and the function used to perform the split was the Scikit-learn function *trainTestsSplit* function. This function takes two datasets as inputs the first being the training dataset that contains all the columns and the second dataset being a dataset containing only the column 'claim' which is the column we are trying to predict, thus we can call the two datasets 'X' and 'Y' respectively. Also as done in the data preprocessing both datasets have had the 'id' column removed since this has zero use, additionally, the X dataset also had the 'claim' column removed. The final step is to apply the size of the training and testing data, for this we set the size of the training set to be 0.7 and testing to be 0.3. Therefore giving us a 70 - 30% split.

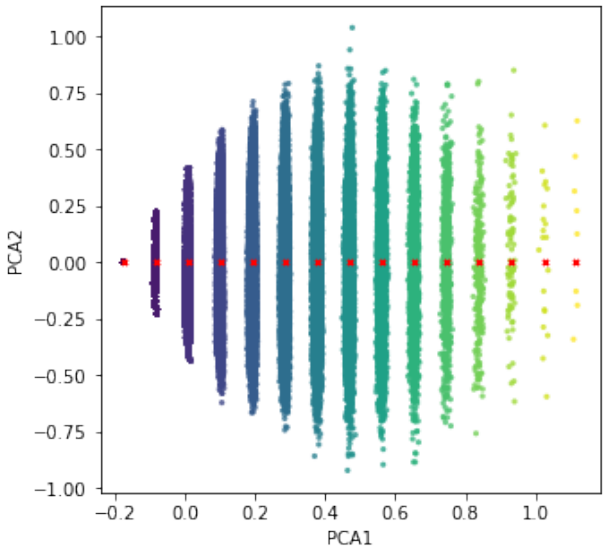

Fig. 2. Grouped PCA.

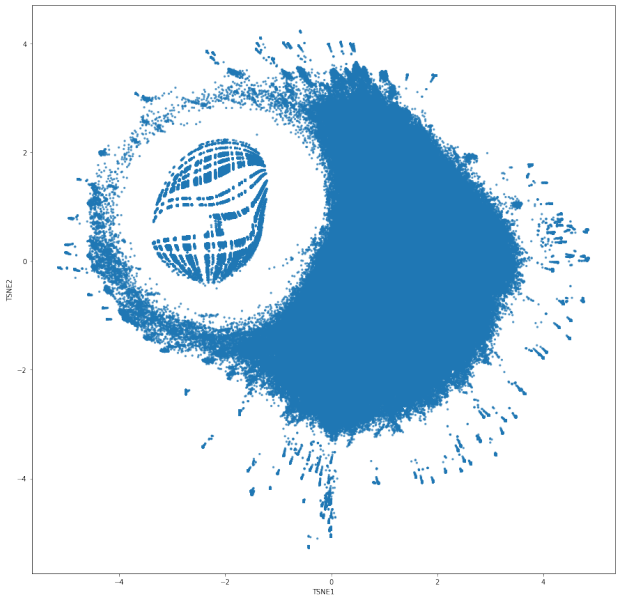

Fig. 3. Output of TSNE for feature engineering.

C. Proposed Methods

Our overall proposed model consists of preprocessing our data, feature engineering, and model stacking. Model stacking involves fitting multiple models simultaneously and then

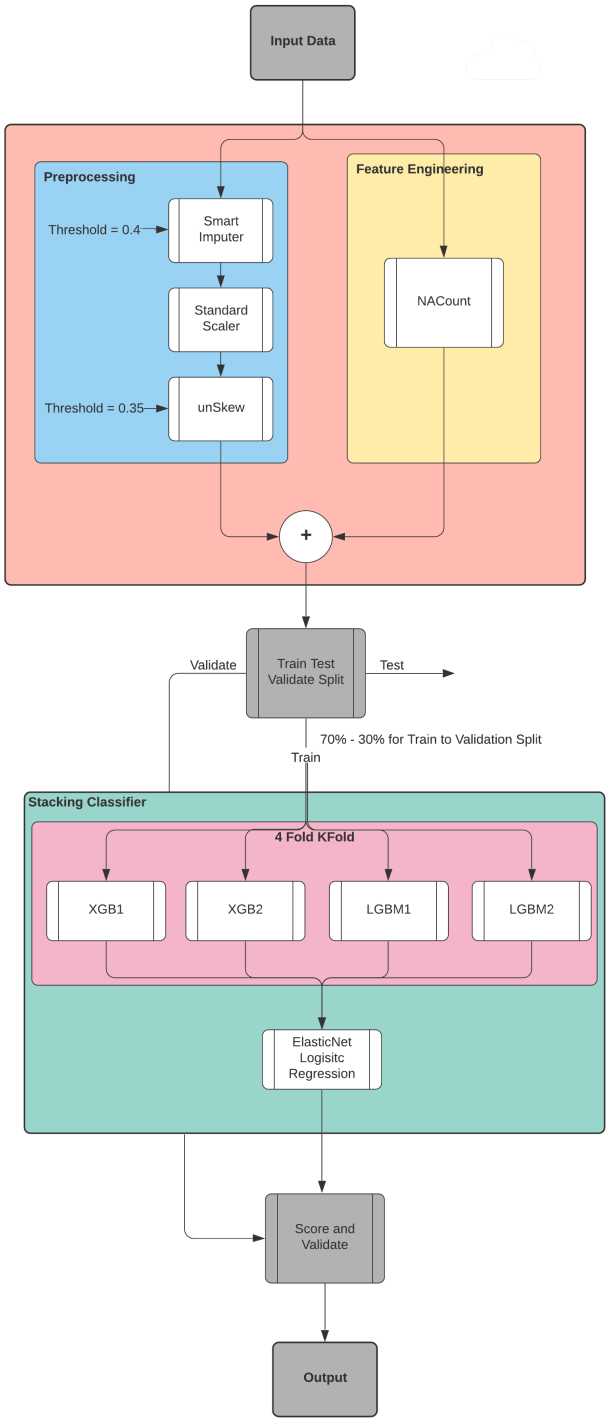

Fig. 4. Flow chart of complete design for our model.

using a final classifier to interpret the output as each model. Currently we are using randomForest, XGBoostClassifier, and SVM as the three stacked models, and a logistic elastic net regressor as our final classifier. With the modular design of stacking, we can easily add more models without breaking any of the existing models. Figure 4 shows a complete diagram of our proposed model.

1) Random Forest: Random Forest has been used as a default for this project, it was the first method attempted and has consistently been the fastest to train. We used Bayesian optimization to tune random forest hyperparameters. After 100 iterations the best parameters found for random forest were:

```
bestParamRF = {'criterion': 'gini',
'max_depth': 10,
'max_features': 'sqrt',
'min_samples_leaf': 0.024684704247566212,
'min_samples_split': 0.26868411877216003,
'n_estimators': 85,
'n_jobs': -1}
```

Figure 5 shows all 100 interactions of bayesian optimization. We can see that a low min sample leaf and low min sample split result in higher accuracies.

2) XGBoostClassifier: Another method that was tested was XGB, We used Bayesian optimization to tune XGBoosts hyperparameters. After 100 iterations the best parameters found for random forest were:

```
bestParamXGB = {'alpha':
↪  3.0098950557831317e-06,
'colsample_bylevel': 0.8888223977605876,
'colsample_bytree': 0.6629640277225338,
'eta': 0.005710790861655478,
'gamma': 0.0071953427713228905,
'lambda': 5.204086077282118e-06,
'max_depth': 5,
'min_child_weight': 106.36834970939083,
'subsample': 0.7639245020607291,
'n_jobs': -1}
```

Figure 6 shows all 100 interactions of Bayesian optimization. A note that columns, eta, gamma, and lamba should be on a logarithmic scale but are not.

3) TabNetClassifier: One of the models we plan to include in our stacking classifier is the TabNet classifier. This model was designed to apply deep neural networks on tabular data. This model's strength comes from being able to learn how to efficiently reduce the error and works perfectly for us since we are using tabular data. As of now this model has not been implemented thus, we can not know what the speed or accuracy of the model is but it is still being considered as a proposed model, we will decide if the model will be used in the final stacking model once implementation and testing have been performed.

4) SVM: One of the proposed models to solve this problem was the Support Vector Machines (SVM) model. The SVM model offers high accuracy when compared to other classification models as seen in the literature review. For this reason, we decided to implement an SVM model but once implemented we noticed that the complexity of the SVM makes it so the system requires a lot of ram to be able to train the model on our large dataset. To solve this problem, we simply reduced the amount of training data being used to 10% as well as the kernel function used by the SVM was set to linear since this is the fastest kernel available. At this point, the time taken to train the classifier was reduced to 1 to 2 hours which was more manageable. In the end, we tested the model and got a score of

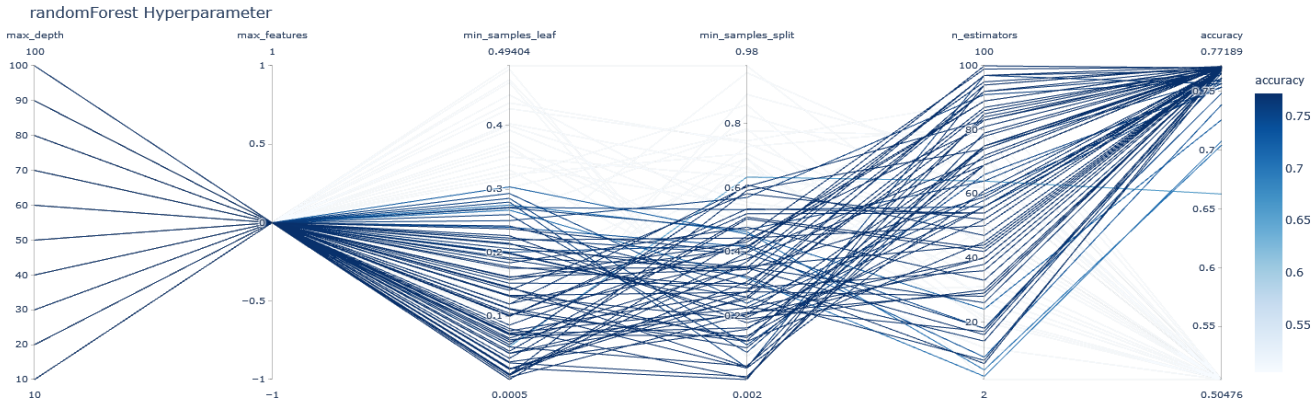

Fig. 5. Parallel Plot of randomForest hyperparameter tuning.

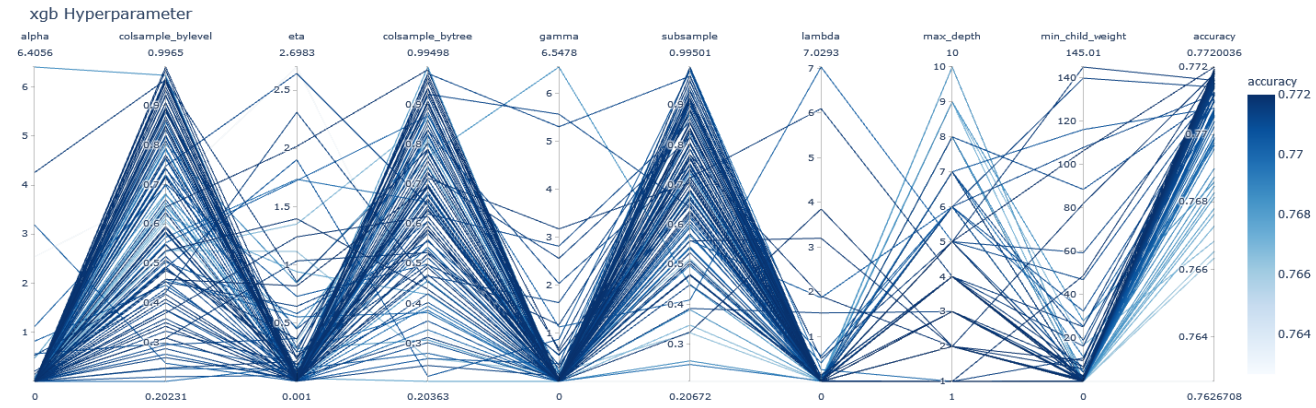

Fig. 6. Parallel Plot of XGBoost hyperparameter tuning.

0.773 which was the highest score we have achieved thus far. The only disadvantage of using this model is the time it will take to train for example since we have added it to our stacking model the time that it will take to run will not be increased by a significant amount. We decided that even though the model is highly time-consuming it was still beneficial for us to use since we were able to achieve a better score at the end, and all we need to do is to keep the training size to around 10% to have a realistic training time. Additionally hyperparameter tuning has not been applied to the SVM, since it has not been decided if we will perform the tuning. In order to perform the tuning we would first need to make sure it can be done in a reasonable amount of time using our system that has limited resources.

D. Deep Neural Multilayer Perceptron (MLP) Model (not being used)

MLP is an artificial neural network, we implemented the MLP but found that it performed poorly even without being compared to other models that we proposed. Because of the bad performance we decided not to spend time trying to tune the parameters such as the number of layers and neurons in each layer, thus this model was dropped and is not being used.

E. Model Stacking

As a final step in our model we use model stacking, which allows us to use each separate model and combine their outputs together. We use an elastic net logistic regressor as our final classifier. This decision was primarily made in relation to [7] who saw success from this method.

IV. Experiment Analysis

A. Dataset

1) A Large amount of NA values - TSNE: To confirm the importance of NaN values, we used t-distributed stochastic neighbor embedding (TSNE) to visualize these NaN values. We encoded our data set so that if a value was NaN it became a 1 and 0 otherwise. When passed into the TSNE, figure 7 was produced, showing a relationship between claim and just the NaN values.

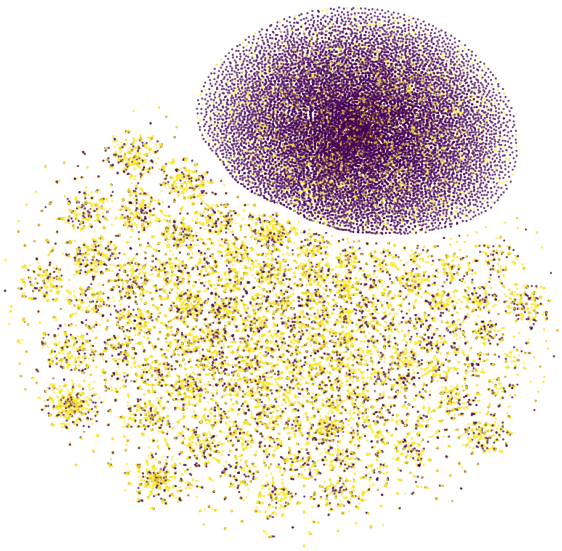

Fig. 7. Visualition of TSNE of our dataset when all NA values are set to 1, else 0.

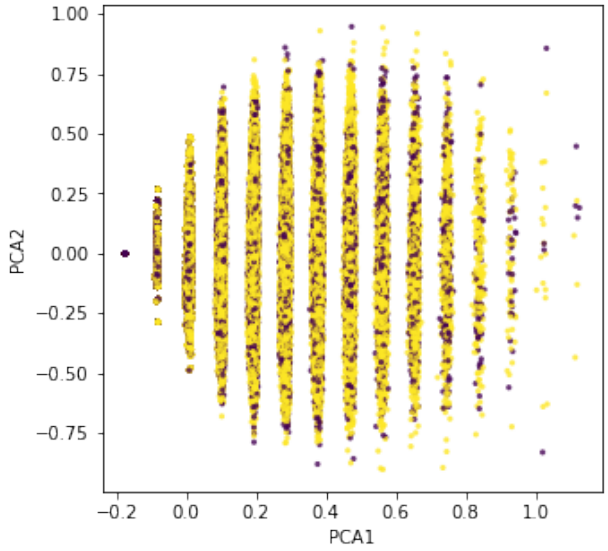

Fig. 8. PCA of our NA Encoded dataset.

2) A Large amount of NA values - PCA: Along with TSNE we also applied our NA encoded values to a PCA. The resulting PCA shows 15 unique stripes. Later on, we use clustering algorithms to group these stripes and use them as features. We can see the results of the PCA in figure 8.

3) Low Feature Correlation: We wanted to improve the model by dropping some attributes based on correlation. First we printed and sorted the correlation values of each attribute with respect to the target attribute "claim." We first thought of removing some features with the lowest absolute correlation with respect to claim. We figured that we should remove correlations that are closest to zero and not lowest in general because attributes with a negative correlation are valuable in that they work in the opposite direction than positive correlations, so they are useful for predicting. Correlations close to zero mean that there is little to no relationship between the input attribute and the target attribute "claim." When we sorted the correlation values, we noticed that the highest correlated value only had a value of 0.014771, meaning it had a 1.5% correlation, which is quite small. The lowest correlated value had a value of -0.021505, meaning it had a -2.2% correlation, which is again small. In past projects we have seen correlations as high as 90%, so seeing every single correlation between -2.2% and 1.5% was interesting.

Each feature has a meaning, but we don't know what the meaning is. To create more accurate predictions, usually you want more input attributes as there are more variables that go into the predictions. For this reason, we didn't want to eliminate too many attributes, however we still wanted to remove attributes that may bring our model scores down. Overall, there are 119 input attributes, so we worked with some different correlation thresholds when trying to find the values that worked best for our models. We tried 3 different thresholds. First, we tried to remove any input attributes that had between a -0.0005 and 0.0005 correlation with respect to the target attribute. Then we tried the threshold expanded to a range of -0.00075 to 0.00075, and finally a threshold range of -0.001 to 0.001. Our first threshold (with a correlation range of -0.0005 to 0.0005) produced our best correlation manipulation result which was 0.76991 as seen in Table 1.

We hypothesized that this threshold was the best because it only eliminated 9 of the 119 input attributes. The threshold range of -0.00075 to 0.00075 got a worse result since 13 features were eliminated, and the worst result was from our widest threshold (-0.001 to 0.001). In this case 15 features were eliminated. We noticed the more features were eliminated, the worse the result was, but we wanted to still eliminate the extremely low correlated features so we kept this information in mind for a while longer while we did our next correlation exploration.

Next we wanted to explore correlations between the input attributes. We wanted to eliminate some of the input attributes that were highly correlated with each other. Having multiple input attributes highly correlated with each other may skew the model to have more of an emphasis on these attributes which may be a very similar feature. Since the features are anatomized, we had to spend more time exploring this as we couldn't logically connect input attributes to each other. We created a correlation matrix and exported it to Microsoft Excel to do some analysis. We selected "Conditional Formatting" $\rightarrow$ "Highlight Cells Rules" $\rightarrow$ "Greater/Less Than." After this we highlighted all features with an absolute correlation value higher than 0.01. We noticed there were many features highlighted and as mentioned before, we don't want to remove many features. This threshold was changed to 0.02. Table 2 shows the input features that had higher than an absolute correlation value of 0.02. This means these are the features that are highly correlated with other input features. There are 14 attributes in Table 2 but we didn't want to remove them all since in the previous correlation exploration we noticed that the less attributes we removed, the better the results were. Therefore we looked at which features each feature was highly

correlated with, and found a pattern. If we remove features f57, f97, and f34, the majority of the features in Table 2 would no longer be highly correlated with other input features as many features are correlated with these three features. Upon removal of these features, the only other input features not satisfied with our correlation threshold of 0.02 would be f36, f15, and f46. These features didn't only highly correlate with the already removed features, however they were correlated with each other. Examining Table 3, we found that both f36 and f15 would be satisfied with our correlation threshold if f46 was removed as they are both highly correlated with f46. On top of this, we know that we want to eliminate features with low correlations with respect to the target attribute. Since f46 has a low correlation with the target attribute, and is highly correlated with the remaining input attributes, removing this would be the best choice. Therefore we removed f57, f97, f34, and f46. Running this data on the model produced a worse result than the previous correlation work (which can be seen in Table 1). We then wanted to try to combine these two thought processes. We would eliminate both the attributes highly correlated with each other, and the attributes least correlated with the target attribute. Interestingly, there was no overlap between the two, therefore 13 features were removed. Upon running this data through our models, we received the worst score yet as seen in Table 1.

In conclusion, removing features based on correlation regardless if this correlation was with respect to other input attributes or the target attribute was not effective and only made our score worse. If we wanted to work further with correlations we could try to add the z-scores of the input attributes which are highly correlated with each other rather than just completely removing them, thus retaining these features. However we think this won't result in any improvements as these features aren't very highly correlated together (only about 0.02 or 2%). Also, the more we work with correlations. the worse our results get, so we decided to focus on other ways to improve our results.

B. Experimental Setup

1) Computing Platform: For this project we primarily used Google Colab for our computing platform. One of the major downsides of Google Colab is the fact that google will give whatever servers are currently available, meaning the hardware gained is often somewhat random. This being said, the process's power felt sufficient for this project with the exception of often a lack of enough available ram. Because of the size of our dataset, certain clustering algorithms (Ward, Birch, etc) which scale poorly in terms of memory, would require terabytes of ram in order to process our whole dataset. As well we often encountered crashes for using too much RAM. This required us to be meticulously careful when declaring large variables as one forgotten DataFrame could cause the whole Colab session to crash.

2) Caching and Pickling: Another technique for managing memory was done with caching. If a calculation took a significant amount of time, the results were often saved either as a csv file or a pickle file. "Pickling" is the process whereby a Python object hierarchy is converted into a byte stream [11]. We cached things such as the TSNE features. We pickled our hyperparameter tuning trails. As long as the search space is the same, the tuning process can pick up where it left off even during a session crash by loading and unpickling our trails.

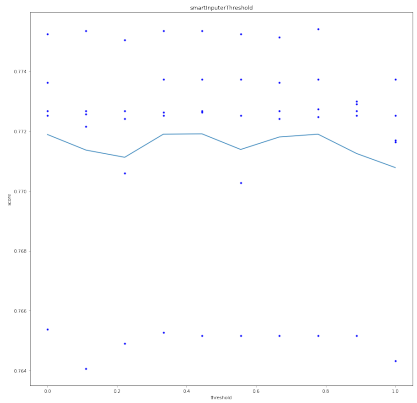

Fig. 9. smartImputer Threshold vs. randomForest model score.

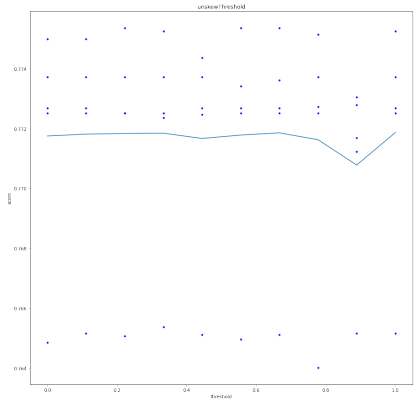

Fig. 10. unskew Threshold vs randomForest model Score.

3) Testing on SmartImputer(): For testing the threshold to set for the SmartImputer we used a grid search from 0 to 1 on a RandomForestClassifier. For each instance we used five-fold cross validation, taking the mean of each test. The results can be seen in figure 9. It was found that the best threshold was around 0.4. That being said, the figure produced is rather uneventful. The changes in the threshold appear to have little effect on the score. For future work we plan to attempt a multivariate imputer, which may handle our large amount of NA values better.

4) Testing on UnSkew(): For testing the threshold to set for the UnSkew method, similarly to the SmartImputer, we used a grid search from 0 to 1. Each test had five-fold cross validation, again where we took the mean of each test. The results can be seen in 10. The best threshold found was around 0.4. Just like the Smart Imputer, the unskew threshold appears to have very little effect on the model score, so we plan to revisit this method later for improvements.

TABLE I
RESULTING SCORES OF DIFFERENT CORRELATION MANIPULATIONS.

Correlation Manipulation	Resulting Score
None	0.77173
Remove Low Correlated Input Attributes with respect to "Claim"	0.76991
Remove High Correlated Input Attributes with respect to "Claim"	0.76979
Remove Both Low and High Correlated Input Attributes with respect to "Claim"	0.76901

TABLE II
EXPLORING INPUT ATTRIBUTES THAT ARE HIGHLY CORRELATED WITH OTHER INPUT ATTRIBUTES

Feature	Number of Occurrences	Features Highly Correlated With
f57	5	f6, f15, f34, f51, f97
f46	2	f36, f15
f97	5	f90, f86, f57, f45, f32
f111	1	f34
claim	1	f34
f34	4	f111, claim, f32, f57
f32	2	f37, f97
f15	2	f46, f57
f51	1	f57
f6	1	f57
f45	1	f97
f86	1	f97
f90	1	f97
f36	1	f46

C. Results and Discussion

1) Problems Encountered: There were quite a few reasons why this problem was difficult to solve. Initially we noticed that the features in the dataset were anonymized and therefore we couldn't make logical connections between the features. Thus, feature engineering was difficult. In past projects we could look at features and know whether to combine them, drop them, etc. For example, in a past project dealing with the prediction of a house price based on features, we could create a feature called "total_bathrooms" which is a combination of "downstairs_bathrooms", and "upstairs_bathrooms." Doing work like this improved our results in those previous projects, but since all the features were anonymized, we could not use our intuition and experiment with these ideas. Furthermore, in past projects we have focused a lot on data correlation in order to improve our results. We would drop attributes that had a very low absolute correlation with respect to the target variable and we would drop or combine attributes that had a very high correlation with respect to other input attributes. When we did our correlation analysis, we were unable to improve our results at all. In fact, the results got worse the more we considered correlation. We think this is because of the values of the correlations. Not many features were highly correlated with each other or with the target attribute, making the removal of lowly correlated features less impactful as there is less compensation (towards a better score) by the highly correlated features. Since feature engineering was difficult and correlation analysis reduced our score, we needed to focus on our models to improve our score. We tried a multitude of different techniques and used hyperparameter tuning to make them as effective as possible. We did get to do a bit of preprocessing work though. We were able to add features such as PCA (dimensionality reduction) and TSNE (nonlinear dimensionality reduction). We also used a preprocessing pipeline in which we implemented a smart imputer which replaced NA values with the median if the skewness of that feature is high (since mean might not be as accurate with high variance in data), and it replaced NA values with the mean if the skewness isn't high. This pipeline also used a min max scaler which simply scales features to a range, thus normalizing the features. Finally we used an unskew method which unskews features that have a high variance (above our threshold) and tries to unskew this feature. If the result turns out to be worse, then the feature is reverted to what it was before. On top of this, we also added a feature that counted the number of NANs in a row. We found this to be useful and improved our score.

2) Handling Overfitting: When working with machine learning algorithms there is always a risk that overfitting will occur, in our case, we managed to reduce overfitting in our models by managing the parameters of the models and complexity. Since we are using a stacking algorithm, we needed to make sure that each algorithm in the stack did not overfit. For the models random forest and XGBClassifier to reduce overfitting, we performed hyperparameter tuning as well as 5k-fold cross-validation to make sure that if overfitting was occurring we would be able to recognize it. By doing this we not only reduced overfitting but also increased the accuracy of the models by selecting the parameters that achieved the best score without causing the model to overfit. To prevent overfitting in the SVM model we used a linear kernel as one of the parameters, this reduces overfitting because the linear kernel is the least complex kernel available which works well with our dataset since we have so much data.

3) Use in Other Applications: Since we were simply working with data that has its features anonymized, we were forced to pay extra attention to other aspects such as model selection, model stacking, and hyperparameter tuning. As previously mentioned, we could not do much feature engineering in the classic sense. This includes combining and/or removing features based solely on the description of the feature and our knowledge of the domain. Doing this, we would normally focus on feature engineering and we would eliminate irrelevant features and create more powerful features through the combination of other features, but we could not do this. Because of this, we have greatly improved our modelling skills, and every big data application requires the use of models. Therefore the knowledge we gained while completing this project will allow us to be more effective in future projects and produce more accurate results faster when we have a normal dataset to deal with. We will use our previous expertise in preprocessing and feature engineering mixed with the knowledge gained about models, stacking, and hyperparameter tuning we learned in this project to be able to make better predictions in any domain.

TABLE III
INPUT ATTRIBUTES THAT ARE STILL HIGHLY CORRELATED WITH OTHER INPUT ATTRIBUTES WHEN F97, F57, AND F34 ARE REMOVED

Feature	Remaining Input Features Highly Correlated with	Correlation with Respect to Target Attribute "Claim"
f46	f36, f15	-0.00798
f15	f46, f57 (f57 already removed)	-0.00725
f36	f46	0.01164

4) Incorporating into a more significant system: We could improve our model and evolve it into an actual insurance system. The point of the project was to be able to predict whether someone will make a claim on their insurance policy based on the provided anonymized features. Once we are able to know the features provided, we could tune the models based on some feature engineering, and then we could deploy the system. We could create a system that takes a data structure with a customer's information as an input, runs it through our model, and the output is the prediction on whether or not this customer will make a claim on their policy. We would also like to implement a confidence interval to display the likely results and how confident the model is.

An insurance company can use this to their advantage. If they have a customer who is extremely likely to make a claim, the insurance company could increase the premiums for this customer. Conversely, the company can use this system to try and lure customers by offering cheaper premiums to the customers who are extremely unlikely to make a claim. They could use this information to try and maximize their profit and will get a better understanding of what to charge people for their insurance to ensure that they will make money. Of course, the system is not perfect and is prone to errors, therefore we would be unlikely to give any advice in the system, just displaying the results of the model and the confidence interval of the results. This way we would assume less liability if the system wrongly predicts for many clients and the company loses money.

5) Current Best Score and Kaggle Ranking: Our current best score from the stacking classifier is around 0.77404. This score proved rather difficult to pass with several different models achieving near this amount. The highest achieving score during the competition was 0.81775 showing that we were rather close to the best score. Because the competition ran only for the duration of September 2021 and our first submission was later than that, we cannot see an official position on the leaderboard.

V. CONCLUSION

In this paper, we go over various techniques used by others when solving classification problems to better grasp which models would give us better accuracy to solve our specific classification problem. By doing this we can save time by not focusing on models that will only be able to give us a low accuracy such as neural networks and instead focus on the models that have higher chances of giving us better accuracy based on the literature review we did. Also since our literature review was split between papers published using machine learning models to solve different classification problems and literature based on the actual problem we are solving, we can get an idea of what people have tried when solving our specific problem and improve on these ideas by using knowledge gained from reading literature where different classification problems were solved and the models used were compared based on accuracy and speed.

REFERENCES

[1] A. l. o. o. panelA.VerikasabA.GelzinisbM.Bacauskieneb, A.Verikasab, a, b, A.Gelzinisb, M.Bacauskieneb, and A. forests (RF) has become a popular technique for classification, "Mining data with random forests: A survey and results of new tests," Aug 2010. [Online]. Available: https://reader.elsevier.com/reader/sd/pii/S0031320310003973?token=A2921E1FB3AF90 east-1originCreation=20211022051455

[2] S. E. Jozdani and B. A. Johnson, "Comparing deep neural networks, ensemble classifiers, and support vector machine algorithms for object-based urban land use/land cover classification," Jul 2019. [Online]. Available: https://www.mdpi.com/2072-4292/11/14/1713

[3] S. Džeroski and B. Ženko, "Is combining classifiers with stacking better than selecting the best one?" [Online]. Available: https://link.springer.com/article/10.1023/B%3AMACH.0000015881.36452.6e

[4] D. Coxon, "Data engineering and scoring." [Online]. Available: https://www.kaggle.com/c/tabular-playground-series-sep-2021/discussion/274570

[5] P. Singla, "Add number of nans in a row as a feature." [Online]. Available: https://www.kaggle.com/c/tabular-playground-series-sep-2021/discussion/270206

[6] M. Berk, "Why doesn't anyone try to implement a neural network?" [Online]. Available: https://www.kaggle.com/c/tabular-playground-series-sep-2021/discussion/270533

[7] V. Konstantakos, "2nd place solution - (a lot of) stacking." [Online]. Available: https://www.kaggle.com/c/tabular-playground-series-sep-2021/discussion/275740

[8] Aries, "My 9th place solution to the september tabular competition." [Online]. Available: https://www.kaggle.com/c/tabular-playground-series-sep-2021/discussion/276248

[9] M. Feurer and F. Hutter, *Hyperparameter Optimization*. Cham: Springer International Publishing, 2019, pp. 3–33. [Online]. Available: https://doi.org/10.1007/978-3-030-05318-5$_1$

[10] R. Shwartz-Ziv and A. Armon, "Tabular data: Deep learning is not all you need," *CoRR*, vol. abs/2106.03253, 2021. [Online]. Available: https://arxiv.org/abs/2106.03253

[11] "Pickle - python object serialization." [Online]. Available: https://docs.python.org/3/library/pickle.html

APPENDIX A
PROJECT FILES

A. ESOF-0151Project.ipynb - Main Project File

https://colab.research.google.com/drive/1MZ-aT4f8fbgDjc99hua6-LDRS2rcTC7C?usp=sharing

B. ESOF-0151Project NN.ipynb - Neural Network Project File

https://colab.research.google.com/drive/1_I0azVg5fmx0Aw-tElBFUYj9CvmIyz1s?usp=sharing

C. Data Exploration

https://colab.research.google.com/drive/1dLPAuPLKeBiESUdE2fprVuxYtXqGYKex?usp=sharing

APPENDIX B
CODE

1) Preprocessing Custom Classes:

```
class SmartImputer(BaseEstimator, TransformerMixin):

  # smart inputer
  # if skewness is high replace na with median
  # else use mean

  def __init__(self, threshold=1):

    self.threshold = threshold

  def fit(self, x, y=None):
    return self

  def transform(self, x, y=None):

    for column in x.columns.to_list():

      skewness = abs(x[column].skew())

      if (skewness > self.threshold):
        # Column has large skewness, use median
        x[column] = x[column].fillna(x[column].median())

      else:
        # low skewness, use mean
        x[column] = x[column].fillna(x[column].mean())
    return x
```

```
class UnSkew(BaseEstimator, TransformerMixin):

  # Unskew method
  # Looks at each column and determines if it needs to be unskewed
  # based off an skewness threshold
  #
  # undoes the unskew method if the skewness of columne becomes worse

  def __init__(self, threshold=0.35, method=np.log1p):
    self.threshold = threshold
    self.method = method

  def fit(self, x, y=None):
    return self

  def transform(self, x, y=None):

    for i in range(x.shape[1]):

      oSkew = abs(skew(x[:, i]))

      if(oSkew > self.threshold):

        unSkewCol = self.method(x[:, i])
        nSkew = abs(skew(x[:, i]))
```

```
        if(oSkew > nSkew):
          x[:, i] = unSkewCol

    return x
```

```
class NAEncoder(BaseEstimator, TransformerMixin):

  # encodes NA values as 1, non NA as 0

  def fit(self, x, y=None):
    return self

  def transform(self, x, y=None):
    return x.isnull().astype(int)
```

```
class NACount(BaseEstimator, TransformerMixin):

  # Returns the count of NA values for each row

  def fit(self, x, y=None):
    return self

  def transform(self, x, y=None):
    return x.isnull().sum(axis=1).to_numpy().reshape(-1,1)
```

A. Hyper-parameter Tuning Functions

```
def sampler(x, y, frac=0.1, seed=0):
  # Samples a random fraction of two np arrays
  # x is shape (n , m) and y is shape (1, m)

  random.seed(seed)

  sampleIndex = random.sample(range(0, len(y)),
                              math.floor(frac * len(y)))

  return x[sampleIndex, :], y.loc[sampleIndex]
```

```
def run_trials(max_evals, save_iter, space, objective, filename, reset=False):

  # used to save trials to disk, usefull


  # if reset, start from begining

  # attempt to load pickle file
  if not reset:
    try:
      # try to find file
      trials = pickle.load(open(filename, 'rb'))
      print(f"Trials file found.  Resuming from {len(trials.trials)} iterations.")
    except:
      # cant find
      trials = Trials()
      print("No Trials file found, will create new file.")

  # Dont attempt to load pickle file
  else:
```

```
    print("Resetting, will write over trials file.")
    trials = Trials()

  best = trials

  # iter until max_evals is reached
  i = len(trials.trials)
  while(i < max_evals):
    i = i + save_iter

    print(f"{i} of {max_evals}")
    best = fmin(fn=objective,
                space=space,
                algo=tpe.suggest,
                max_evals=i,
                trials=trials)

    with open(filename, 'wb') as f:
      pickle.dump(trials, f)

    clear_output()
  return best
```

```
def visualize_trials(filename, title):

  # Display trials file as a parallel coordinates plot

  randForestSpaceDF = pd.DataFrame()
  inputDump = {}

  trials = load_trials(filename)

  for i in range(len(trials.trials)):
    for key in trials.trials[i]['misc']['vals']:
      inputDump[key] = trials.trials[i]['misc']['vals'][key][0]

    inputDump['accuracy'] = trials.trials[i]['result']['loss'] * -1
    randForestSpaceDF = randForestSpaceDF.append(inputDump, ignore_index=True)

  fig = px.parallel_coordinates(randForestSpaceDF, color="accuracy",
                                color_continuous_scale=px.colors.sequential.Blues,
                                title=title)
  fig.show()
```

```
def getBestParam(filename):.

  # Get best hyper parameters found from trials files

  trials = load_trials(filename)
  return trials.best_trial['misc']['vals']
```

B. Hyper-parameter Objective Functions

```
def RandomForestObjective(space):

    clf = RandomForestClassifier(**space,
                                 n_jobs = -1)
```

```
    accuracy = cross_val_score(clf, xTrain, yTrain).mean()

    # we aim to maximize accuracy, therefore we return it as a negative value
    return {'loss': -accuracy, 'status': STATUS_OK }
```

```
def XGBObjective(space):

    clf = xgb.XGBClassifier(**space,
                            n_jobs=-1,
                            use_label_encoder=False,
                            eval_metric = 'auc',
                            objective='binary:logistic',
                            tree_method='gpu_hist',
                            predictor='gpu_predictor')

    accuracy = cross_val_score(clf, xTrain, yTrain, cv=4).mean()

    # we aim to maximize accuracy, therefore we return it as a negative value
    return {'loss': -accuracy, 'status': STATUS_OK }
```

```
def objectiveTab(space):

    space['searchSpace']['n_a'] = space['searchSpace']['n_d']

    clf = TabNetClassifier(**space['searchSpace'])

    clf.fit(xHyper, yHyper,
            eval_set = [(xVal, yVal)],
            eval_metric = ['auc'],
            max_epochs = 75)

    auc = max(clf.history['val_0_auc'])

    # we aim to maximize auc, therefore we return it as a negative value
    return {'loss': -auc, 'status': STATUS_OK }
```

```
def objectiveLGBM(space):

    if (space['boosting_type'] == 'goss'):
      space['subsample'] = 1

    kf = KFold(n_splits = 5, shuffle = True)

    scores = []

    for train_index, test_index in kf.split(xTrain):

      xTrainSplit, xTestSplit = xTrain[train_index], xTrain[test_index]
      yTrainSplit, yTestSplit = yTrainFull[train_index], yTrainFull[test_index]

      clf = LGBMClassifier(**space)
      clf.fit(xTrainSplit, yTrainSplit,
              verbose=0,
              eval_set=[(xTestSplit, yTestSplit)])
      y_pred = clf.predict_proba(xVal)[:, 1]

      score = roc_auc_score(yVal, y_pred)
      print(f'LGBM fold auc: {score}')
```

```
    scores.append(score)

  # we aim to maximize accuracy, therefore we return it as a negative value
  return {'loss': -np.mean(scores), 'status': STATUS_OK }
```

C. Model Stacking

```
class TabNetClassifierWrapper(BaseEstimator, ClassifierMixin):
  '''TabNet is special and needs a wrapper to work properly with sklearns
  stacking classifier.

  space (dict): the search space for tabnet
  eval (list of tuple): a list of x & y validation sets

  '''
  def __init__(self, space, eval_set, eval_metric, max_epochs):

    self.space = space
    self.eval_set = eval_set
    self.eval_metric = eval_metric
    self.max_epochs = max_epochs

    self.clf = TabNetClassifier(**self.space)

  def fit(self, x, y=None):

    self.clf.fit(x, y,
                 eval_set = self.eval_set,
                 eval_metric = self.eval_metric,
                 max_epochs = self.max_epochs)

    return self
```

```
class LGBMClassifierWrapper(BaseEstimator, ClassifierMixin):
  '''LGBM is special and needs a wrapper to work properly with sklearns
  stacking classifier.
  '''
  def __init__(self, space, eval_set, eval_metric):

    self.space = space
    self.eval_set = eval_set
    self.eval_metric = eval_metric

    self.clf = LGBMClassifier(**self.space)

  def fit(self, x, y=None):

    self.clf.fit(x, y,
                 eval_set = self.eval_set,
                 eval_metric = self.eval_metric,
                 verbose = 0)

    return self

  def predict(self, x, y=None):
    return self.clf.predict(x)
```

APPENDIX C
HYPER-PARAMETERS

A. Search Space

1) Random Forest:

- *max_depth*: Int-Uniform [10, 100]
- *max_features*: Choice [*auto*, *sqrt*]
- *min_samples_leaf*: Float-Uniform [0, 0.5]
- *min_samples_split*: Float-Uniform [0, 1]
- *n_estimators*: Int-Uniform [1, 100]

2) XGBClassifier:

- *booster*: Choice [*gbtree, gblinear, dart*]
- *eta*: Log-Uniform [e^{-7}, e]
- *max_depth*: Int-Uniform [1, 10]
- *subsample*: Float-Uniform [0.2, 1]
- *colsample_bytree*: Float-Uniform [0.2, 1]
- *colsample_bylevel*: Float-Uniform [0.2, 1]
- *min_child_weight*: Log-Uniform [e^{-16}, e^{5}]
- *lambda*: Log-Uniform [e^{-16}, e^{2}]
- *alpha*: Log-Uniform [e^{-16}, e^{2}]
- *gamma*: Log-Uniform [e^{-16}, e^{2}]

3) Tabnet Classifier:

- *mask_type*: Choice [*entmax, sparsemax*]
- *n_d*: Int-Uniform [8, 64]
- *n_a*: Should always equal *n_d*.
- *n_steps*: Int-Uniform [3, 5]
- *gamma*: Float-Uniform [1, 2]
- *n_independent*: Int-Uniform [1, 5]
- *n_shared*: Int-Uniform [1, 5]
- *momentum*: Float-Uniform [0.01, 0.4]
- *lambda_sparse*: Log-Uniform [e^{-6}, e^{-3}]

4) LightGBM Classifier:

- *boosting_type*: Choice [gbdt, goss, dart]
- *num_leaves*: Int-Uniform [20, 150]
- *learning_rate*: Log-Uniform [e^{-6}, e]
- *subsample_for_bin*: Int-Uniform [20000, 500000]
- *min_child_samples*: Int-Uniform [20, 500]
- *reg_alpha*: Float-Uniform [0, 1]
- *reg_lambda*: Float-Uniform [0, 1]
- *colsample_bytree*: Float-Uniform [0.5, 1]
- *subsample*: Float-Uniform [0.5, 1]
- *is_unbalance*: Choice [*False, True*]

B. Final Selection

```
bestParamXGB1 = {
    'alpha': 3.0098950557831317e-06,
    'colsample_bylevel': 0.8888223977605876,
    'colsample_bytree': 0.6629640277225338,
    'eta': 0.005710790861655478,
    'gamma': 0.0071953427713228905,
    'lambda': 5.204086077282118e-06,
    'max_depth': 5,
    'min_child_weight': 106.36834970939083,
    'subsample': 0.7639245020607291,
    'n_jobs': -1,
    'eval_metric' : 'auc',
    'random_state': 0,
```

```
    'use_label_encoder': False,
    'objective': 'binary:logistic',
    'tree_method': 'gpu_hist',
    'predictor': 'gpu_predictor'}

bestParamXGB2 = {
    'eval_metric' : 'auc',
    'lambda': 0.004562711234493688,
    'alpha': 7.268146704546314,
    'colsample_bytree': 0.6468987558386358,
    'colsample_bynode': 0.29113878257290376,
    'colsample_bylevel': 0.8915913499148167,
    'subsample': 0.37130229826185135,
    'learning_rate': 0.021671163563123198,
    'grow_policy': 'lossguide',
    'max_depth': 18,
    'min_child_weight': 215,
    'max_bin': 272,
    'n_estimators': 10000,
    'random_state': 0,
    'use_label_encoder': False,
    'objective': 'binary:logistic',
    'tree_method': 'gpu_hist',
    'predictor': 'gpu_predictor'
}


bestParamRF = {
    'criterion': 'gini',
    'max_depth': 10,
    'max_features': 'sqrt',
    'min_samples_leaf': 0.024684704247566212,
    'min_samples_split': 0.26868411877216003,
    'n_estimators': 85,
    'n_jobs': -1
}

bestParamTab = {
    'gamma': 1.2215519222663302,
    'lambda_sparse': 0.0036912807595016355,
    'mask_type': 'entmax',
    'momentum': 0.33845753977793525,
    'n_d': 28,
    'n_independent': 3,
    'n_shared': 5,
    'n_steps': 5,
    'verbose': 0
}

bestParamLGBM1 = {
    'boosting_type': 'gbdt',
    'colsample_bytree': 0.803928347810881,
    'is_unbalance': True,
    'learning_rate': 0.17025191721735602,
    'min_child_samples': 30,
    'num_leaves': 140,
    'reg_alpha': 0.7333444384270381,
    'reg_lambda': 0.5221658401562319,
```

```
    'subsample': 0.5786269381707934,
    'subsample_for_bin': 380000
}

bestParamLGBM2 = {
    'boosting_type': 'gbdt',
    'colsample_bytree': 0.6077154792966379,
    'is_unbalance': False,
    'learning_rate': 0.08028224397812166,
    'min_child_samples': 385,
    'num_leaves': 60,
    'reg_alpha': 0.09019592529911713,
    'reg_lambda': 0.08245955371473801,
    'subsample': 0.8360916597569221,
    'subsample_for_bin': 340000
}
```