# Malware Detection - Unbalanced Dataset Problem

Team Name: Survey Corps

**Sai Rithwik** M
IMT2018061
sai.rithwik@iiitb.org

**Saksham Agarwal**
IMT2018065
saksham.agarwal@iiitb.org

**Sama Sai Karthik**
IMT2018067
sai.karthik@iiitb.org

***Abstract*— Through this document we are trying to understand how with a given set of features we can predict the probability of a certain system being affected by malware.**

***Index Terms*— AUROC, Balanced Bagging Classifier, Catboost Encoder, Column Transformer, Decision Tree Classifier, Label Encoder, LGBM Classifier, Logistic Regression, Min-Max Normalisaton, One Hot Encoder, PCA, Pipeline, Sampling, Random Forest Classifier, SGD Classifier, SMOTE, Stacking, Stratified K Fold Validation, Target Encoder, Voting, XGB Classifier**

## Problem Statement

The malware industry continues to be a well-organised, well-funded market dedicated to evading traditional security measures. Once a computer is infected by malware, criminals can hurt consumers and enterprises in many ways. Can you help protect more than one billion machines from damage before it happens?

## Dataset

The training dataset has 567730 rows and test dataset had 243313 records. The dataset consisted of 81 columns indexed with *MachineIdentifier* and the column *HasDetections* was the target which had to be predicted for the test dataset using these columns. Details regarding the data columns can be found in the link mentioned in references[1].

## Introduction

Malware is defined as software designed to infiltrate or damage a computer system without the owner's informed consent. The rapid rise of the Internet and the ensuing growth in malware meant that manually created detection rules were no longer practical - and new, advanced protection technologies were needed, hence the need for a new generation model that can predict if a system would have virus based on the system hardware and software features provided to us. Many companies turn to machine learning, an area of computer science that had been used successfully in image recognition, searching, and decision making, to augment their malware detection and classification. Santander was one of them as well.

In our work, we haven't exactly made a prediction of whether a system will have Malware or not but instead give a probabilistic score of its appearance and we try to find how prone a system is to have Malware in it, given its system software and hardware details.

## Preprocessing

One of the major issues that this entire dataset had was that it was heavily categorical in nature due to presence of various columns which had details regarding versions, platforms, OS etc. Hence preprocessing the data was of utmost importance as this was one of the huge factors in establishment of a good score for the model.

Initially we had planned to build a sophisticated column transformation which imputes each feature separately, but on checking the number of null values we had found that there were just a handful of columns which had crossed the 30% null data mark. So our initial task was to remove the columns which had over 60% of null values. Once these columns had been removed, imputation was the most important task needed to be done. Initially we had grouped columns into various lists based on the dtypes and unique values present in that particular column. This helped us in a long run by easily segregating the columns so that they can be reused for pipeline method to encode. The criteria we chose to go with while selecting which column to be encoded by a certain method has been mentioned in Table 1

| | **dtype** | **nunique** | **Comments** |
|---|---|---|---|
| **numerical_pipeline** | int, float | any | truly numerical columns |
| **oh_pipeline** | any | <= 10 | |
| **hc_pipeline** | any | >10 | |

TABLE I
Categorical Partition

Once this criterion was established we had two options, whether to manually encode train and test or create a pipeline for it. We chose to go ahead with the latter as it makes sure all the columns are encoded properly both in train and test and this would leave out any probability of us forgetting to encode a separate row or column specifically. Before making the pipeline we made sure by checking each column's unique values to check for any redundant values in it and we found a few in *Census_PrimaryDiskTypeName*, and we changed it's values accordingly. Next comes the task of imputing. We decided to keep encoding simple by choosing to impute the data in numerical pipeline with the median of the data. And the remaining categorical data had been imputed with the mode of the column data. We had given a thought of using the scikit multivariate imputation method *IterativeImputer*, but it had some time overhead[2] which was unfavourable,

hence we decided not to proceed with it.

Out of the most popular methods for encoding for columns which had less unique values, we decided to go with *One Hot Encoding*, mainly because the number of columns is considerably less and most probably would not grow by considerable amount after encoding using One Hot method. Our major roadblock was which method of encoding to use for the categorical columns which had more than 10 unique values. Our initial choices were

- **Target Encoding** - Our first choice as this was a good way encode features with a blend of the expected value of the target given particular categorical value and the expected value of the target over all the training data.
- **Feature Hashing** - This ensures speed, but there is a huge issue of overlap of the hashed data, hence we decided not to go with it. There were a few alternatives. Also it was similar to One Hot Encoding in someway, and in a way or other it would increase the dimensions of the dataframe, hence we decided not to go with it.

We had considered other complex encoding techniques like *GLMM Encoding*, but all of them had a huge time overhead, hence we decided to drop them out of consideration. Fortunately, all these implementations were provided by *category_encoders* library to try them out. Another advantage of using this library is that they impute the values automatically[3].

After running our model few times, we had realised that our estimates from the validation test data was not upto our expectations with the score that we were expecting. So digging deeper into the encoding methods used, we had figured out that *Target Encoder* had a huge disadvantage of target data leakage[4]. Hence we had immediately fixed it by using *CatBoost Encoder*, as it had fixed the inherent issues of target data leakage by *Target Encoder*. The final pipeline that we used to transform the columns is shown in Figure 1

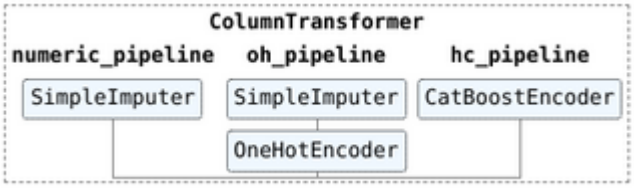



Fig. 1. Column Transformation Pipeline

Subsequently we made a few more changes to our pipeline to improve the model. One of the most important additions has been scaling the entire encoded data using Min-Max Normalisation.

## Exploratory Data Analysis

Before we could begin with encoding we surely wanted to see how the data was distributed and choose models accordingly. One of the most important thing to lookout was for how evenly the classes were distributed in the target column.

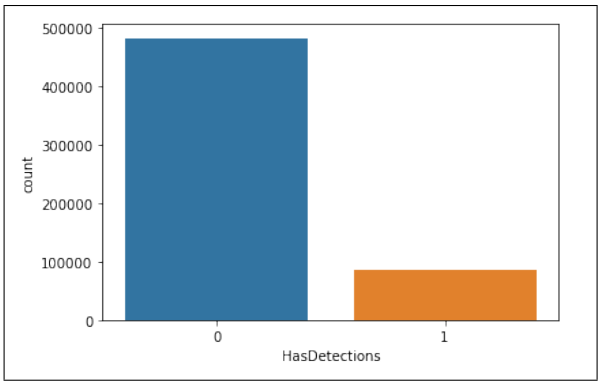



Fig. 2. Target Class Distribuition

As expected from the problem statement we had observed that the target variable was unbalanced, with the majority class taking upto 85% and the minority class taking upto 15%. This is evident from Figure 2

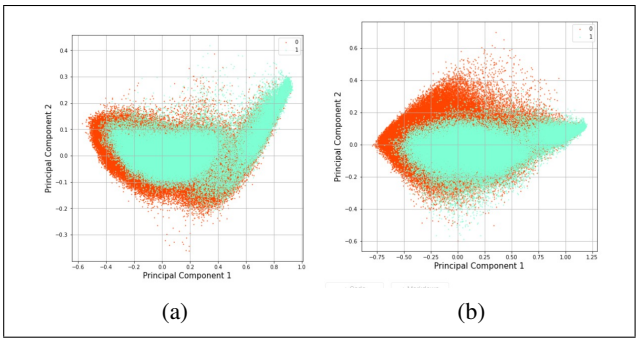
This implied that while choosing a model we had to make sure this unbalanced dataset doesn't give us wrong predictions by always returning probabilities for the majority class. Hence there was some sort balancing that needed to be done. Initially we had planned to take the traditional approach of Oversampling the dataframe using SMOTE provided by the imblearn package. Similarly we had tried other methods of sampling like Random Oversampling, Random Undersampling, Near Miss Undersampling, Cluster Centroids etc. out of which SMOTE and Near Miss worked the best for us by providing us with more accurate results. It was strange to see Undersampling method work, but after some discussion we came to the conclusion that it was working well because of the fact that there was a considerable percentage of the minority class to train upon. One disadvantage using them was that we could not find an index to the validation set, which we shall see further while using Voting method. Hence we had decided to not go with such sampling methods. On further research we had observed that most of the Decision Tree models in scikit package provided an option called as *class_weights*. More details regarding this is discussed in Model Selection section.

Another important thing that we had to make sure while choosing the model was to choose which kind of classifier we had to go with. So our simple intuition was to plot the PCA columns and check for the distribution graphs. Figure 3(a) and 3(b) shows our result which came after a bit of tuning parameters.

PCA clearly showed that this task cannot be performed by linear classifiers, so we decided to try out models which could classify this non linear distribution. Some of the ones which came on top of our heads were Gaussian based SVM and Decision Tree based classifiers like Random Forests.

## Model Selection

Two things we were certain by now, we have a classification problem and the dataset is unbalanced. From our findings, models like Random Forests, Boosting algorithms




(a) (b)

Fig. 3. PCA Analysis

are often the go-to choices especially when the classes are not linearly separable. Through PCA we did see that along with the principal components, the two classes are highly overlapping.

For handling unbalanced data, we tried exploring the imbalance-learn library. We found out models like Balanced Bagging, Balanced Random Forests, Extra Trees Classifier, Easy Ensemble, RUSBoostClassifier. The advantage that these models had was they sampled data of unbalanced classes on the fly, while the strategy of sampling could be mentioned as a hyperparameter.

Easy Ensemble can work with boosting algorithms as base estimators. It uses AdaBoost as the default base estimator, we also tried running CatBoost, XGBoost, and LGBM classifiers as base estimators to have the goodness of better classification alongside sampling on the fly.

Whenever we felt this composition didn't produce better results we tried the go-to classification models with oversampled data using SMOTE and undersampled data using NearMiss strategies. We also tried RandomOverSampler which again overfit the data for some of the estimators that were performing well and RandomUnderSampler wasn't very successful.

But as discussed earlier in the EDA section, sampling strategies had their own issues. Hence we decided to go with the standard ensembles with a small change. We decided to measure the impurity of a class in each node with *Gini* method, and penalise heavily for incorrect classification. This provided us some good results. It was achieved by adding *class_weight* parameter.

In the end we had a pool of models giving us very similar results, thus we decided to switch to voting and stacking strategies. For voting we gathered the results of the models which produced best results along with their corresponding AUROC score. We took the smoothed average of the probabilities with weights as their AUROC scores to improve the predictions. The motive behind voting was to smooth the different aspects that a single model might fail to capture.

Stacking was our other go to strategy, we now re-ran most of our best models to also store the predicted probabilities of the train set and test set. We gathered this data for 5 models and applied a classification algorithm which didn't was distinct from the algorithms used in the 5 models.

We also tried Gradient Boosting Methods like LightGBM and XGBoost. One of the main reasons being, their ability to handle classification problems really well[5]. Hence we decided to go with them, and infact XGBoost had provided us really satisfactory results even fetching us one of our best scores.

## Training Details

In our Machine Learning project, we trained different models on the dataset and tried selecting the one with the best performance. However, there is almost room for improvement as we cannot say for sure that a particular model is best for the problem at hand as there were many performing well, hence our aim was to improve the model in any way possible. One important factor that determines these are the hyperparameters, and below are some of the methods that we tried for hyperparameter tuning.

- **GridSearchCV**: In GridSearchCV, we gave an initial dictionary which contains a list of hyper parameters that the model can take. This is then cross-validated over all the specified params and we can choose the one with the best performance. But one drawback of this was that it took a lot of time when specified with more parameter, as it had to bruteforce through all the possible combinations.
- **RandomizedSearchCV**: RandomizedSearchCV is very useful when we have many parameters to try and the training time is very long. In this instead of all the params that are specified inside the dict , only a fixed number of parameter settings is sampled from the specified distributions.
- **Hyperopt**: Hyperopt uses a form of Bayesian optimization for parameter tuning that allows you to get the best parameters for a given model. It can optimize a model with hundreds of parameters on a large scale.

In the end both RandomisedSearchCV and Hyperopt were used to tune the parameters. Some of the best results we had observed has been tabulated below in Table 2.

| Model | Private Score | Public Score |
|---|---|---|
| Voting | 0.71103 | 0.70696 |
| XGBoost | 0.70693 | 0.70297 |
| Catboost | 0.70124 | 0.69631 |
| LGBM | 0.69734 | 0.69270 |
| Stacking | 0.69407 | 0.69149 |
| Balanced Bagging[0] | 0.63750 | 0.63431 |
| Easy Ensemble[0]. | 0.63038 | 0.62801 |

TABLE II
Models and Scores

## CONCLUSIONS

We can clearly see that the predictions were not perfect as the AUROC score was near 0.71. In the end we had a private

[0]We had predicted the binary values for the detections and hence there was a low score, later we had implemented to predict the probabilities

score of 0.71103 and a public score of 0.70696. As one would have predicted, with more data the results could have been more accurate, and identification of True Positives and True Negatives would have been much better. Also sampling techniques were not of much use as the data was reasonably well sampled, hence using various sampling techniques only improved the score by a bit. We understood the importance of tuning hyperparameters and how efficient a model becomes on changing a slight parameter.

## ACKNOWLEDGMENT

Firstly we would like to thank Prof. G Srinivasaraghavan and Prof. Neelam Sinha for teaching such a wonderful course. Without the concepts taught by them, we believe it would have been much harder to understand what we were doing. We would like to thank our Teaching Assistant Tejas Kotha for sparing time and helping us whenever we asked for. Without his valuable feedback, it would have been hard to implement the concepts. Thanks to Shreyas Gupta for clarification of doubts on Slack, whenever we raised any. Special Thanks to Saiakash Konidena, Nikitha Adivi, members of Team Breaking Code and Team Modulo 3 with whom we had very fruitful discussions on how to approach this problem, and for sharing tips.

Overall this assignment has been a great learning experience for the entire team and we would like to surely implement the concepts that we learnt here on the project.

## REFERENCES

[1] Dataset - Malware Detection: Unbalanced Dataset Problem, Kaggle
[2] Stef van Buuren, Karin Groothuis-Oudshoorn (2011). “mice: Multivariate Imputation by Chained Equations in R”. Journal of Statistical Software 45: 1-67.
[3] Categorical Imputation in category_encoders Library
[4] Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, Andrey Gulin (2019). “CatBoost: unbiased boosting with categorical features”.
[5] Friedman, J. H. (1999a). “Greedy function approximation: A gradient boosting machine” Technical Report, Department of Statistics section 4.6; Algorithm 6