PMS: Predicting Malware Susceptibility

Manjot Bilkhu[1]
Electrical and Computer Engineering
University of California San Diego

Tushar Dobhal[1]
Electrical and Computer Engineering
University of California San Diego

Abstract—**In this day and age of the internet, it is very tough to characterize the type of data we deal with. This naturally brings forward the need to be careful in understanding the type of data we deal with and whether it is safe for our machines or not. This project aims at predicting malware in machines by identifying the prominent features, filtering them appropriately and then using state of the machine learning techniques. In this project, we start by analyzing the dataset, compressing it to enable it to fit in the memory, identifying irrelevant features and then implementing and comparing Gradient Boosted Trees Classification using two of the prominent libraries available today, namely, XGBoost and LightGBM.**

Index Terms—**Malware prediction, Logistic Regression, Random Forest, XGBoost, LightGBM**

I. Introduction

The amount of data that we interact with these days can be very hard to quantify. Thanks to the internet, not all data is good for our computers. This brings forward a huge concern for security of our systems and how to protect ourselves. It is important to note that malware is just one of the many big problems that the security industry needs to address. However, with the availability of data, it is definitely an interesting thing to inspect whether one can study specific patterns to identify or predict malware attacks in a data-driven way. This study aims to answer this question by looking at different machine characteristics and identifying whether some users might be prone to malware attacks. We introduce the Microsoft Malware Prediction Dataset [1] in section 2, our methodology in section 3, and results in section 4. We summarize our conclusions in section 5. All our code is available on github[1].

II. Microsoft Malware Prediction Dataset

The Microsoft Malware Prediction Dataset was released with an intent to advance research in the field of data-driven malware prediction and modeling. This was part of a challenge which was hosted on Kaggle in 2015. The dataset has 7.85 million counts of different combinations of machine characteristics and the goal is to identify which of these machines have malware. Since the dataset is collected and provided by Microsoft, it is not surprising to learn that all of these machines run some form of Windows operating systems. It would have been interesting to look at UNIX based operating systems as well to get a better sense of the entire picture, but there were no large-scale datasets that even come close to this one.

[1]https://github.com/tushardobhal/malware_prediction

Each row in the dataset corresponds to a unique machine identifier. There are a total of 83 features for each machine. We highlight some of the important and diverse features here. The column *HasDetections* is the ground truth binary label for this classification problem, and tells us whether the machine has any malware or not. Some columns like *ProductName* tell which version of Windows Defender, Microsoft's default anti-malware tool, is running in these machines. *DefaultBrowsersIdentifier* tells us which is the default browser running in the machine. *AVProductStatesIdentifier, AVProductsInstalled* and *AVProductsEnabled* tell us about the state and type of anti-virus software product installed on the machine. There are a whole lot of features like *OsVer, OsBuild* and *Platform* which identify the type and version of the operating system running on the machine.

There are a set of features like *GeoNameIdentifier* and *CityIdentifier* which identify the geographic location and the city from where the particular machine is. Features like *IsProtected* and *Firewall* tell whether the machine has an active subscription for an anti-virus and firewall respectively. Note that there are a total of 83 columns corresponding to each entry in the dataset, and we have only highlighted a few interesting ones which might help us in predicting malware. In the following subsections, we formally highlight our approach to cleaning the dataset, and some exploratory data analysis which tells us which features might be important.

A. Data Cleaning

The training dataset consists of a total of 7.85 million rows. Each row, or machine, has a maximum of 83 features, some of which we introduced above. There is a column *HasDetections* which corresponds to our ground truth. Most of the columns in our dataset have features which are not categorical. We use *pandas* to encode these into categorical features and save these transforms to later apply them to our test set.

The entire dataset takes about 7 minutes to load using *pandas*, using the naive loader. We make a few optimizations at the floating point level to encode these columns in a way which takes less memory. This saves us about 28% memory and compute power, which turns out to be significant when dealing with dataset's of this order.

B. Exploratory Data Analysis

In this section, we present some interesting insights from the dataset. The very first thing we look at, before approaching this classification problem is whether the dataset is balanced.

We show this in figure 1 and note that it is pretty well-balanced as nearly half of the machines have malware, and the other half don't. This ensures that there is no class imbalance that needs to be addressed and we can move forward with training.

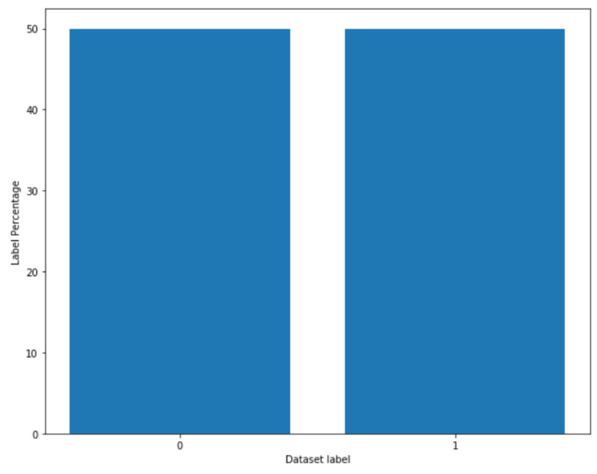

Fig. 1. Number of data points vs ground truth

Since our source of data is not perfectly clean, there are a lot of rows and columns in our dataset which do not contain any value. In other words, we have a few columns for which we have very little data. We create a plot of the top 10 columns with missing data and show it in figure 2.

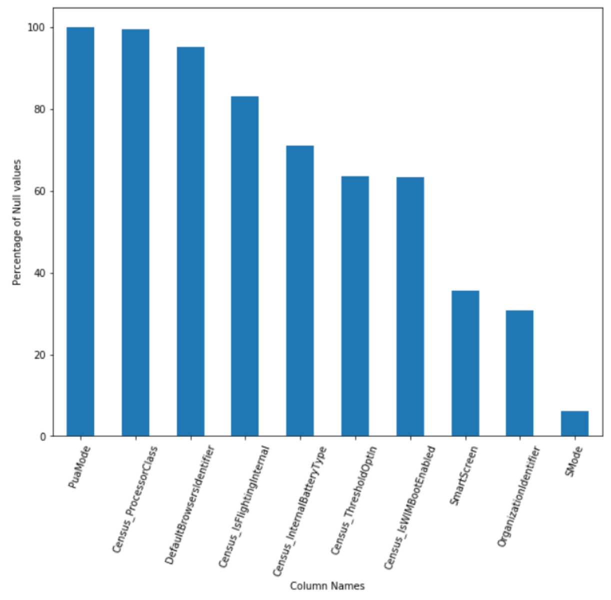

Fig. 2. Top 10 columns with missing values

As we can see that for some of the columns, we hardly have any machines which have non-null values. *PuaMode*, *Census_ProcessorClass* and *DefaultBrowserIdentifier* have the highest number of null values. We drop all such columns which have more than 5% missing values. After this pre-processing step, we are left with about 7.7 million data points with 62 columns, which is still a lot of data. After applying this filter, we note that there is still no class imbalance. Both classes are distributed roughly 50-50, as shown in Figure 1.

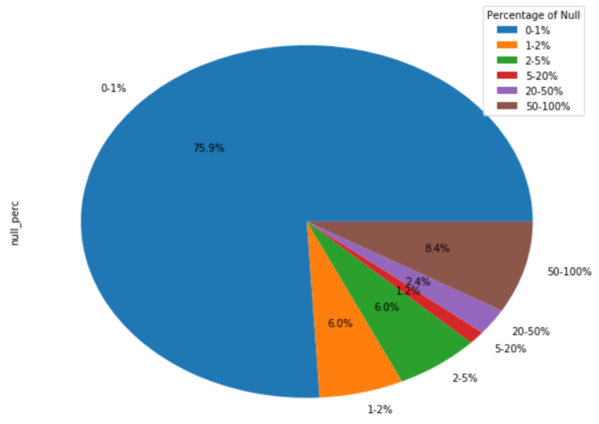

Fig. 3. Percentage of missing values

Figure 3 shows the exact percentage of columns having missing values binned into different bins. As one can see, a fair amount of the data is complete, having values for all features.

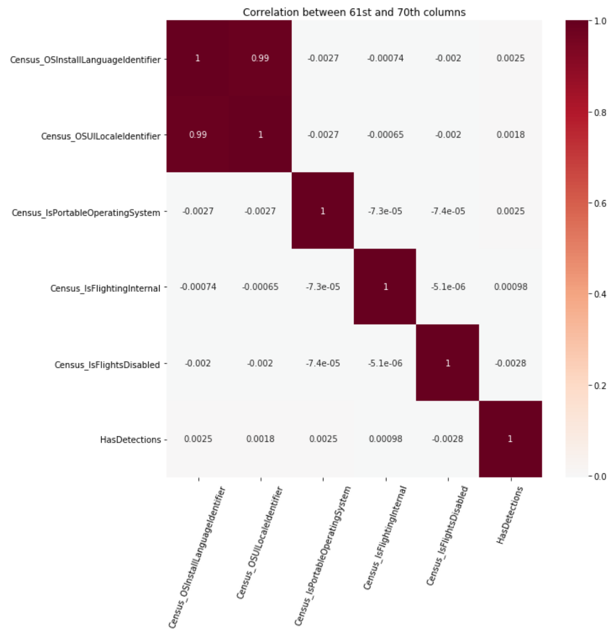

Fig. 4. Correlation between columns

Next, we investigate whether there is correlation amongst the retained columns. We noticed that two columns have very high correlation, with the pearson correlation coefficient almost approaching 1. These two columns were *OSInstallLanguageIdentifier* and *OSUILocaleIdentifier*. We observed that by dropping any one of these columns does not have any impact on the final accuracy obtained.

III. Methodology

This section highlights the methods used for prediction, the reason for the selection of said techniques and the results obtained from using them. We begin our predictive analysis by establishing our baseline and then diving deep into the more complicated Classification algorithms. We make use of Gradient Boosted Trees Classification, and analyze the differences between the implementation of the algorithm in two of the most popular libraries in use today, XGBoost [2] and LightGBM [4].

Our evaluation metric is the Mean Prediction Accuracy obtained over the entire test set. Accuracy gives us a measure of the correctness of the algorithm to predict a machine as malware or safe. Since we do not want any incorrect identification in either case and given the balanced nature of our dataset with almost equal number of positive and negative samples, Accuracy seems to be the appropriate metric for our case.

The dataset was split into Training, Test and Validation sets. Our training dataset comprised of about 5 million examples, validation set comprised of 1.23 million examples and the testing dataset consisted of 1.54 million rows.

IV. Results

We first discuss our Baseline model and formulate the results obtained using them. We then dwell deeper into Gradient Boosted Trees, their implementations in both XGBoost and LightGBM, and compare the performance of each.

A. Baseline Techniques

The first baseline technique that we present here is what we call the Random Classifier. This classifier is similar to the *sklearn.dummy.DummyClassifier* implemented in the popular *sci-kit learn* library [5]. It predicts a label uniformly at random, for the entire test set, without any prior training.

We thought that the above baseline was too naive to be compared to. Hence, we trained another popular technique, namely, Logistic Regression, to compare our more complex methods with. Logistic Regression is a technique that makes use of the Sigmoid function to select a class label during prediction. Its training involves selecting a set of parameters that minimize the Binary Cross-Entropy Loss function, using Gradient Descent. Logistic Regression is selected as it is one of the most simple algorithm to train and test, and it does not take a lot of time as well to run.

The results obtained from our baseline technique is shown in Table I. As expected, the prediction accuracy and running time of the Logistic Regression model is more than that of the Random Classifier, and this result is consistent even when the experiment was run repeated number of times.

TABLE I
Timings and Accuracy of our Baseline models

Method	Random Classifier	Logistic Regression
Training Time (s)	NA	114.88
Prediction Time (s)	0.076	0.879
Mean Accuracy (%)	48.71	52.63

B. Gradient Boosted Trees Classification

Since most of the features in our dataset were categorical, gradient boosted trees were definitely the go-to choice for this problem. Gradient boosted trees are inspired by boosting algorithms like AdaBoost [6]. Boosting, in this context, is a technique using which we can learn a strong learner from many weak learners. Gradient boosted trees first build trees for each feature and recursively build weak classifiers by identifying which examples were hard to classify. We limit our discussion of gradient boosting trees to just an intuitive one as advancing them through the underlying math is beyond the scope of this course and the goal of this project.

The implementation of Gradient Boosted trees stems from the paper by Friedman [3]. This technique involves the computation of decision trees by minimizing a differential loss function. For our binary classification application, we have utilized a Binary Cross-Entropy Loss function.

C. Comparison of XGBoost and LightGBM Libraries

XGBoost has been the choice of library for Gradient boosted trees for a long time, while LightGBM was introduced two years after XGBoost. Although both the libraries implement the same algorithm, there is structural difference in the implementation. XGBoost uses a pre-sort algorithm, which involves sorting the data points based on each feature value, and then doing a linear scan to find the best split value. It also implements a histogram binning technique, which distributes the data into discrete bins. Although, creation of data takes almost the same time as pre-sorting, subsequent computations are linear in the number of bins and not in the number of data points, as in pre-sorting. Histogram technique also reduces memory consumption, since it replaces continuous values with discrete bins.

LightGBM, on the other hand, use a novel technique called Gradient-based One Side Sampling (GOSS). Similar to how AdaBoost gives weights to each individual example, this algorithm tries to do the same by considering examples with large gradients as samples which need more training, and those with smaller gradients as those that can be ignored. Therefore, it selects all the examples which have a large gradient and randomly samples the others. Since, not all the parameters are chosen at each step, it is usually faster than the XGBoost implementation.

A comparison of the important tuning parameters of both the libraries, along with the corresponding values used is given in Table II. As seen from the table, parameters like *num_boost_rounds* and *early_stopping_rounds* were kept the same and for the remaining parameters grid search technique was used on half the dataset and the most optimized hyper-parameters on the validation set were chosen. For both the implementations, Binary Cross-Entropy Loss with Area under the Curve (AuC) evaluation metric is chosen.

The results of both of these libraries on our dataset is given in Table III. It shows the time taken and the prediction

TABLE II
IMPORTANT PARAMETERS OF EACH LIBRARY

Parameter Type	XGBoost	LightGBM
Regularization	1. learning_rate or eta (0.05) 2. max_depth (11) 3. min_child_weight (7) 4. reg_lambda (1) 5. reg_alpha (0.6)	1. learning_rate (0.05) 2. max_depth (5) 3. num_leaves (256), since trees also grow depth-wise 4. min_data_in_leaves (42) 5. lambda_l1 (0.15) 6. lambda_l2 (0.15)
Categorical Features	Not Available	1. categorical_feature (Not used, instead Pandas was used)
Speed	1. num_boost_round (20,000) 2. early_stopping_rounds (200) 3. gamma or min_split_loss (0.2) 4. col_sample_bytree (0.4) 5. subsample (1)	1. num_boost_round (20,000) 2. early_stopping_rounds (200) 3. feature_fraction (0.8) 4. bagging_fraction (0.8) 5. bagging_freq (5)

accuracy obtained by each implementation. We notice that XGBoost takes more than twice the time for both training (18 hours) and prediction step, for the best-tuned hyper-parameters, and gives almost similar prediction accuracy, with LightGBM implementation falling short by only about 0.45%. The similarity in the accuracy measure can be attributed to the fact that they both use the same algorithm, and the difference in timings can be seen from the respective implementations that were discussed before.

TABLE III
TIMINGS AND ACCURACY OF GRADIENT BOOSTED TREES

Library	XGBoost	LightGBM
Dataset Size (GB)	1.04	1.04
Training Time (hrs)	18.89	7.96
Prediction Time (hrs)	1.13	0.54
No. of Iterations	13,312	15,916
Best Training (AuC)	0.745	0.737
Best Validation (AuC)	0.712	0.709
Mean Accuracy (%)	65.59	65.15

D. Further Investigations

After obtaining the results in the previous section, we decided to further investigate the performance of the prediction model after feature pruning. Boosted trees is a very useful algorithm to know what features were actually used for tree splitting, and obtain their respective feature importances. Since the performance of both the libraries were quite similar and LightGBM, due to its GOSS implementation technique, was much faster, we decided to do our analysis using LightGBM implementation.

Since the training of the entire dataset was a time-consuming process, we selected the top k features and compared the dataset size, training and testing times, prediction accuracy in each case. These results are shown in Table IV. This shows that by using only 20 of the entire 62 columns, we can reduce the dataset size to half, and only reduce the prediction accuracy by about 0.8%. This shows that the trade-off in accuracy for increased speed and reduction in size is not very high.

TABLE IV
TIMINGS AND ACCURACY USING ONLY TOP K FEATURES

No. Of Columns	10	20	40
Dataset Size (MB)	352.85	536.63	852.72
Training Time (hrs)	5.13	5.04	7.37
Prediction Time (hrs)	0.46	0.43	0.56
No. of Iterations	13,775	12,667	16,636
Best Training (AuC)	0.704	0.721	0.736
Best Validation (AuC)	0.681	0.698	0.708
Mean Accuracy (%)	63.17	64.36	65.08

V. CONCLUSION

In this project we develop techniques to perform Malware Prediction on a large dataset. We first compressed the dataset so as to load into a single machine efficiently and performed dataset cleaning on it to remove null and highly correlated feature columns. Next, we define our baseline methods and then perform Gradient Boosted Trees classification on the data set to obtain the results. We also note the important differences between the two implementations (XGBoost and LightGBM), mention their respective parameters and then compare their performance on our dataset. Finally, we notice that not all features are important for prediction and that pruning the features can help to reduce the size and running time of the algorithm without hurting the performance accuracy by much.

REFERENCES

[1] Microsoft malware dataset. *https://www.kaggle.com/c/microsoft-malware-prediction/* (2015).

[2] CHEN, T., AND GUESTRIN, C. Xgboost: A scalable tree boosting system. *ArXiv abs/1603.02754* (2016).

[3] FRIEDMAN, J. H. Greedy function approximation: A gradient boosting machine. *Annals of Statistics 29* (2000), 1189–1232.

[4] KE, G., AND ET AL., Q. M. Lightgbm: A highly efficient gradient boosting decision tree. In *NIPS* (2017).

[5] PEDREGOSA, F., VAROQUAUX, G., AND GRAMFORT, E. A. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res. 12* (Nov. 2011).

[6] SCHAPIRE, R. E., AND FREUND, Y. *Boosting: Foundations and Algorithms.* The MIT Press, 2012.