Save the Attack! Project
Team: Good People
AI 511: Machine Learning

Kautuk Raj
IMT2019043

Mohd Rizwan Shaikh
IMT2019513

Abstract

The goal of this project is to predict a Windows machine's probability of getting infected by various families of malware, based on different properties of that machine. Each row in this data-set corresponds to a machine, uniquely identified by a `MachineIdentifier`. `HasDetections` is the ground truth and indicates that a malware was detected on the machine. Using the information and labels in training dataset, we predict the values for `HasDetections` for each machine in the testing dataset.

1 Introduction

Machine learning plays a major role in today's internet era. From Amazon's book suggestions to managing traffic in large cities, machine learning is applied in one or the other way. Hence it is very essential to have a better understanding of machine learning techniques and algorithms, which can efficiently predict the futuristic problems and find their solutions.

Malware is shorthand for malicious software. It is a type of software developed by cyber attackers with the intention of gaining access or causing damage to a computer or network, often while the victim remains oblivious to the fact there has been a compromise. The malware industry continues to be a well-organized, 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. Malware can wreak havoc on a computer and its network. Hackers use it to steal passwords, delete files and render computers inoperable. A malware infection can cause many problems that affect the daily operations and the long-term security of any company.

2 Dataset

The dataset used here is a part of *Save the Attack!* Competition which is hosted on Kaggle. The training data has 83 columns. The training data has a `HasDetections` column which tells us whether the Windows machine is infected by various families of malware, based on different properties of that machine. The training dataset is perfectly balanced with approximately same number of $0s$ and $1s$. In the training dataset we have nearly 71 lakh records which we will use to build machine learning models. The testing data has 82 columns and we need to predict the `HasDetections` column basis the other columns using the developed models. Here, 0 means that the machine is not infected and 1 means that it is infected. Given the unique values (0 and 1) in `HasDetections` column, it is clear that the problem is a binary classification problem.

Another file `sample_submission.csv` is given to let us know how the output is supposed to be like. It contains a `MachineIdentifier` column and a `HasDetections` output column. The `MachineIdentifier` column holds a unique ID for each machine.

3 Preprocessing

Surprisingly, the target variable `HasDectections` is balanced. No duplicated entries were found in the dataset.

On a closer look at the data, we realize that all the features can't be used for predicting the output. Problems such as lots of missing values, heavily skewed numerical and categorical columns, correlated columns, mismatching data types needs to be addressed accordingly. Moreover, since the dataset was large, we had to figure out alternate ways of handling the dataset to prevent crashing of the notebook.

Our method of making the dataset more manageable changed drastically as we implemented

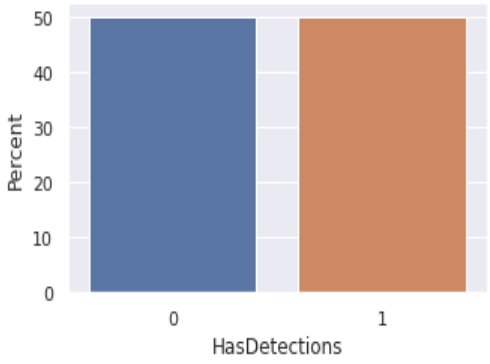

Figure 1: 'HasDectections' column percentage-wise plot

new models. This has been discussed in detail pointwise below.

3.1 Handling notebook crashes

The initial attempt of starting the preprocessing directly got hampered due to large size of the dataset. First of all, we downcasted all the columns using the minimum and maximum value present in them. We also deleted variables after using them and used garbage collector to reduce RAM usage. For more flexibility, we made separate notebooks for preprocessing the data and training the models. The preprocessed datasets were stored separately and used accordingly.

3.2 Missing Values

As expected, there were a lot of NaN values in the dataset. We tried to make the training dataset more manageable by dropping the columns with high percentage of missing values. For features with a lesser percentage of missing values, we dropped the data-points.

The columns which are dropped because of high percentage of missing values are: `Census_ProcessorClass`, `PuaMode`, `DefaultBrowsersIdentifier`, `Census_IsFlightingInternal`, `Census_InternalBatteryType`, `Census_ThresholdOptIn`, `Census_IsWIMBootEnabled`, `SmartScreen` and `OrganizationIdentifier`. All these columns had more than 30% of missing values.

Missing values were also present in the test dataset. In this case, we replaced NaN values with the mode of the column.

3.3 Skewed Columns

In the dataset, there were skewed categorical columns as well. We decided to drop the categorical columns which were heavily skewed. We didn't define any specific threshold for dropping the skewed columns. However, we dropped the columns wherein one category occupied more than 90% of values.

The columns which are dropped are `ProductName`, `IsBeta`, `RtpStateBitfield`, `Census_IsVirtualDevice`, `IsSxsPassiveMode`, `HasTpm`, `Platform`, `OsVer`, `AutoSampleOptIn`, `SMode`, `UacLuaenable`, `Census_IsPortableOperatingSystem` and `Census_DeviceFamily`.

There were skewed numerical columns as well in the dataset. We removed the skewness from the columns by using relevant skew removal techniques (applying either log or raising to exponential power).

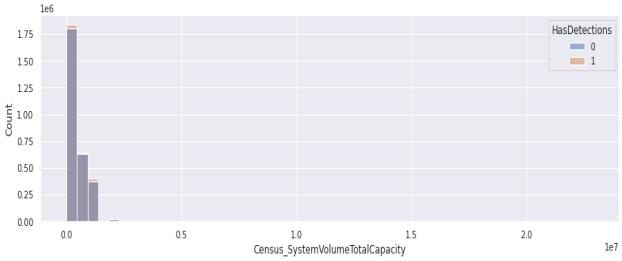

Figure 2: Skewed numerical column 'Census_SystemVolumeTotalCapacity'

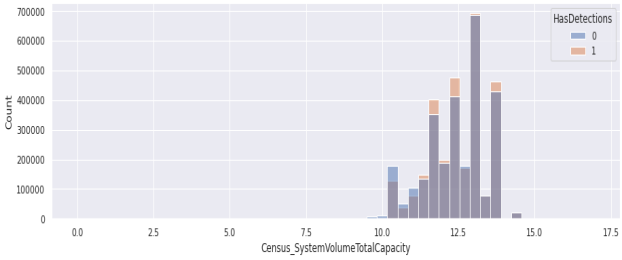

Figure 3: 'Census_SystemVolumeTotalCapacity' after removing skewness by using log.

3.4 Columns with Large Number of Categories

We dropped the categorical columns which had large number (possibly in the range of 1 lakh) of unique values in them. The following columns are dropped: `AvSigVersion`, `AVProductStatesIdentifier`, `CityIdentifier`, `Census_OEMNameIdentifier`, `Census_OEMModelIdentifier`, `Census_FirmwareVersionIdentifier` and `Census_ProcessorModelIdentifier`.

3.5 One Hot Encoding and Bucketing

Our initial attempt of preprocessing heavily involved bucketing and one hot encoding the columns. The reason for one hot encoding is that most of the models (except tree based models) tend to give better results with one hot encoded data. Moreover, bucketing was also used to reduce the number of categories and thus make the dataset usable for training. Minor versions were combined into one and care was taken to prevent the `MinorVersions` category to become the major one.

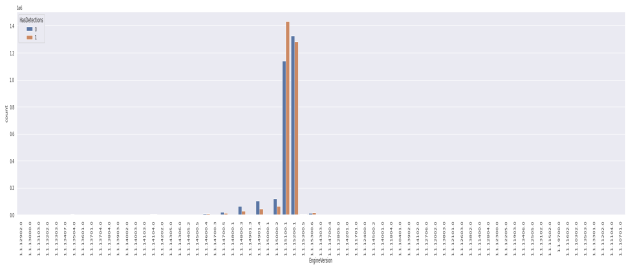

Figure 4: 'EngineVersion' column with many categories

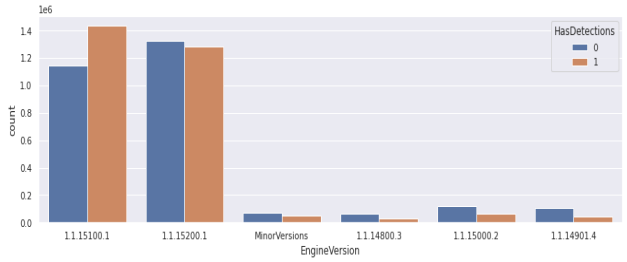

Figure 5: 'EngineVersion' column after stacking the minority categories into the category 'MinorVersions'

3.6 Label Encoding

We used label encoding explicitly for tree based models. Since tree based models work well with label encoding, we did label encoding for most of the categorical columns. `AppVersion` is one column on which label encoding was done.

3.7 Domain knowledge

While carrying out the preprocessing, we considered domain knowledge as well instead of following the machine learning rules strictly. For example, some non-numerical columns like `IsProtected`, `Firewall` were kept untouched despite the skewness due to their useful nature towards the problem to be solved.

4 Models and Training Process

After completing the preprocessing, we started exploring different ways of training the whole dataset with different models. Our initial attempts involved training 10 to 50 lakh data points (out of 57 lakh datapoints) using different models. We started with linear and non linear logistic regression, Naive Bayes and SGDClassifier using `Pandas` and `Sklearn`. After a few rounds of training, we decided to drop logistic regression and Gaussian Naive Bayes because of their poor score despite tuning the models. SGDClassifier gave promising results.

We soon switched to Dask-ML for training the models as it took comparatively less time and was also able to train the whole dataset in one go. We trained the SGDClassifier on the whole dataset using Dask-ML. With hyperparameter tuning, we were able to achieve a score of 0.59.

After this, we started training tree-based models. We decided to explore Decision Trees, Random Forest, XGBoost and LightGBM before fixing one model and going in for hyperparameter tuning. We also split the training dataset into three (or more) files to implement incremental learning. While training the models, we realized the Random Forest Classifier was performing better than Decision Trees and hence, we ruled out the possibility of training Decision Tree Classifier. Since XGBoost was giving almost identical performance (sometimes better) as compared to Random Forest Classifier and was taking less time, we decided to go with XGBoost and drop Random Forest Classifier. After hyperparameter tuning, we were able to achieve the best score of 0.683.

Of all the models, we found that LightGBM was giving us the best score by a minor margin. So we decided to try hyperparameter tuning on this model. We manually tuned the hyperparameters (picked a few probable values and then used grid search) after reading the documentation to see which of them would be useful for us. We feel that by manually tuning instead of directly using a grid searching technique, we saved time and were able to make more dynamic adjustments depending on the cross-validations scores. We initially started with 100 estimators. We tuned the number of estimators, learning rate, tree depths, number of leaves. By this method, we were able to achieve a public test score of 0.628. After this, we decided to increase the number of estimators to 440 to try and im-

prove our score. These models took a lot more time to train but promised better results. By tuning, we were able to achieve a public test score of 0.685.

5 Results

Refer **Table 1**, found on the next page.

6 Conclusion

We were able to make a fairly accurate model to predict the risk of a malware attack. The main challenge we faced were the fact that the certain features (columns) of the dataset were quite skewed and the information regarding one particular class was limited. More data from these classes would most likely improve the performance of any model on this task. More computing power to deal with big data would be helpful too. Using machine learning to predict malware attacks can prove quite helpful for anti-virus software development and the personal computer industry in general.

7 References

- https://www.kaggle.com/pavansanagapati/14-simple-tips-to-save-ram-memory-for-1-gb-dataset
- http://ethen8181.github.io/machine-learning/trees/lightgbm.html
- https://ml.dask.org/
- https://xgboost.readthedocs.io/en/stable/
- https://stackoverflow.com/questions/38079853/how-can-i-implement-incremental-training-for-xgboost
- https://towardsdatascience.com/how-to-make-sgd-classifier-perform-as-well-as-logistic-regression-using-parfit-cc10bca2d3c4

Model	**Hyperparameters tuned?**	**Rows in dataset**	**ROC-AUC score**
Linear Logistic Regression	YES	50 Lakh	0.646
Gaussian Naïve Bayes	NO	30 Lakh	0.651
XGBoost	NO	10 Lakh	0.669
XGBoost	YES	Entire Dataset	0.683
Decision Trees	YES	50 Lakh	0.609
SGD Classifier	YES	Entire Dataset	0.586
Random Forest	YES	Entire Dataset	0.641
LightGBM	NO	Entire Dataset	0.658
LightGBM	YES	Entire Dataset	0.685

Table 1: Trained models and the obtained ROC-AUC scores