Microsoft Malware Prediction

CZ4032 Data Analytics & Mining
Group 3

Name	Matric No.	Contribution
Tu Anqi	U1622399F	30%
Castillo Fitzgerald Gumtang Clarence	U1622291F	30%
Andre Kristanto	U1620068C	8%
Tang Jiayun	U1621004H	13%
Eko Edita Limanta	U1620574A	7%
Hans Albert Lianto	U1620116K	12%

Abstract

With the fast development of information network, cyber security has become more and more important in safeguarding the interest of individuals, corporations and even nations. Malware, software with malicious purposes, has always been the most serious threat to cyber security. Moreover, attacker groups nowadays are well-resourced and technically experienced, dictating an urgent need for reliable and efficient malware prevention methods.

Traditionally, malware protection aims to detect malware with either signature-based or behavior-based techniques, and then take countermeasures. However, the detection happens after the malware has attacked the machine, by which time, the malicious executable may have already achieved its aims and cause irreversible damages. In this paper, we developed effective models for predicting if a machine will soon be hit with malware, so that preventive measures could be implemented to protect machines from such malware threats before any damage is done.

Data mining was performed on data provided by Microsoft on Kaggle, which consists of 82 properties of 8,921,483 Microsoft machines and whether malware was detected on each machine, which is the target variable. Data preprocessing was first performed on the data, consisting of handling missing data, invalid values and outliers, selecting relevant features, aggregating certain features, label encoding non-numeric features, normalizing data, and engineering features.

To evaluate and compare the performances of each model, the preprocessed data were then split into train and test set in a ratio of 7:3. A diverse range of machine learning algorithms were then trained on the train set, including Naïve Bayes, Logistic Regression, Decision Tree, Neural Network, Random Forest and Tree-based Gradient Boosting. This variety ensured a good coverage of all machine learning techniques from which the best one could be selected. Inclusion of the Random Forest and the Gradient Boosting models also allowed us to leverage on the ensemble techniques which had been proven to be effective in improving the prediction performance as compared to any single model. To combine the advantage of boosting and bagging, 100 gradient boosting models were also trained on different bootstrap sampled after which the results were averaged.

For some algorithms, hyperparameters could largely affect their performance. Hyperparameters are configuration parameters whose values are set before the learning process. Therefore, 5-fold cross-validation grid-search was applied for selected models to find the optimal hyperparameters.

Each model were then trained on the entire train set with its optimized hyperparameters. The prediction by the trained model on the test set was compared to the actual target values to provide a reliable measurement of the model's performance. The metric used was the Area Under Receiver Operating Characteristic (ROC) Curve (AUC) score. The Bagging of Gradient Boosting model achieved the highest score (0.749), followed by Single Gradient Boosting (0.742) and Random Forest (0.728). Models with ensemble techniques generally performed better. Being trained on the entire original Kaggle train set, the Bagging of Gradient Boosting model's prediction for the Kaggle test set ranked top 14% on Kaggle leaderboard.

Though label encoding was preferred for lower memory usage and lower dimensionality, the imposed ordinality for nominal attributes might be misleading for models. Further improvements could be done by using more advanced encoding techniques, eg. target encoding or frequency encoding. The effectiveness of one-hot-encoding combined with Principal component analysis (PCA) could also be experimented. Moreover, finer model tuning could be performed, eg. tuning number of layers for the neural network model, and using smaller intervals for hyperparameter candidates. Besides, model stacking, another ensemble technique, could be used for combining diverse models trained using different machine learning algorithm, hyperparameter settings, training sets or features. Furthermore, Recursive Feature Elimination (RFE) could also be used for features selections. Lastly, it was observed that the Kaggle train and test set were split in a time

series manner. Therefore, it is recommended to use walk-forward validation when performing hyperparameter optimization and model selection.

1. Problem Description

1.1 Motivation

With the rapid development of the Internet, malware has become one of the major cyber threats nowadays. Malware, or malicious software, is intentionally designed to cause damage to a personal computer or even a corporate network, leading to significant loss of the interest of an individual, a community or even a nation on the whole. Moreover, the malware industry nowadays is organized and experienced, aiming to challenge traditional security policies or measures and thus having an even greater impact. In fact, the global cost due to increasing cybercrime has reached us $600 billion, accounting for 0.8% of global GDP (Lau, 2018).

Therefore, malware protection of computer systems is one of the most important cybersecurity tasks for single users, businesses and even nations nowadays. For single users, It is a crucial factor to enable them to enjoy the convenience and benefits brought by the fast development of information network without being threatened by hostile acts from unauthorised parties. For businesses, It ensures the security of crucial business information and continuity of operations. For a nation, It guarantees its financial environment, national security and social stability.

1.2 Related Work

Traditionally, malware detection techniques are divided into two broad categories, signature based and behaviour based (Tahir, 2018). These techniques aim to detect malware and then take countermeasures against those malwares.

The signature based detection technique is used by most of the antivirus programs, which maintains malware signatures and disassembles the code of each file to search for the pattern that belongs to a malware family. This detects known malwares with relatively lower resources, but unknown malwares cannot be detected. This makes it particularly unsuited to detecting zero-day malware unless it shares signatures with previously known strains. Considering the exploding development of new threats today, the effects of this method is limited (Tahir, 2018).

The behaviour based detection technique detects abnormal behavior of a system so that ultimately both the known and unknown malware attacks can be identified and resolved. However, data needs to be updated regarding new and unknown malwares and relatively more resources are required in terms of time and space.

In recent years, the application of malware detection mechanisms to recognize malicious files through data mining techniques have increased. (Souri, Norouzi, & Asghari, 2017).

Dong (2017) presented in his thesis a novel Android Permission based malware detection technique. Huge datasets on both malware and information on benign Apps (e.g. permissions) have been gathered and an ensemble model is developed using machine learning algorithms, including Logistic Regression Model, Tree Model with Ensemble techniques and Neural Network.

Rhode et al. have used recurrent neural networks to perform early stage malware prediction, based on a short snapshot of behavioural data. The focus of their research is to predict whether a file is malicious during execution, instead of post-execution.

1.3 Problem Definition

One common disadvantage of aforementioned techniques is that the detection happens after the malware has attacked the machine, by which time, the malicious executable may have already achieved its aims and cause irreversible damages (Rhode, Burnap, & Jones, 2018).

Therefore, the aim of our project is to use data mining techniques to predict the probability of a machine being soon hit with malware, so that preventive measures could be implemented to protect machines from such malware threats before any damage is done.

2. Approach

To solve the problem of predicting whether a machine will be hit with malware, data mining was performed on the data provided by Microsoft on Kaggle, which consists of records of 82 properties of 8,921,483 Microsoft machines and whether malware was detected on each machine, which is the target variable to predict. The approach taken was summarized in Figure 1 below.

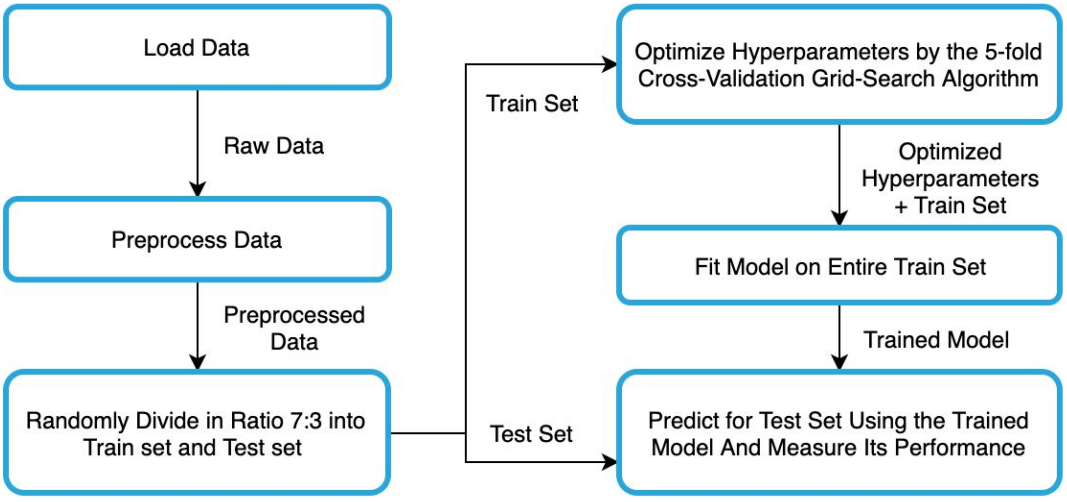


Figure 1. Overall Approach

As shown in the figure above, data preprocessing was first performed on the data, consisting of handling missing data, invalid values and outliers, selecting relevant features, aggregating certain features, label encoding non-numeric features, normalizing data, and engineering features.

To evaluate and compare the performances of each model, the preprocessed data were then split into train and test set in a ratio of 7:3. A diverse range of machine learning algorithms were then trained on the train set, including Naïve Bayes, Logistic Regression, Decision Tree, Neural Network, Random Forest and Tree-based Gradient Boosting. This variety ensured a good coverage of all machine learning techniques from which the best one could be selected. Inclusion of the Random Forest and the Gradient Boosting models also allowed us to leverage on the ensemble techniques which had been proven to be effective in improving the prediction performance as compared to any single model. To combine the advantage of boosting and bagging, several gradient boosting models were also trained on several bootstrap sampled after which the results were averaged.

For some algorithms, the hyperparameters could largely affect the model's performance. Hyperparameters are configuration parameters whose values are set before the learning process. By contrast, the values of other model parameters were derived via training. Tuning hyperparameters could ensure the optimized ones are selected, in order to maximize the effective capacity of each model for solving the target task. Selected hyperparameters of some models were optimized by the 5-fold cross-validation grid-search algorithm over a parameter grid on the train set. This model validation technique used an independent data set to assess how the model would generalize, thus providing a more reliable evaluation. Another advantage was that all observations were used for both training and validation, and each observation was used for validation exactly once, as compared to random subsampling. With averaging of multiple runs, this method also avoided misleading results as compared to the holdout method which only involves a single run. Compared to training with leave-one-out validation technique, training with k-fold cross validation was also the more realistic option duration-wise, given the large data size.

For each model, the optimized hyperparameters would then be used when training the model on the entire train set. The trained model was then used to make predictions for the test set, after which the prediction would be compared to the actual target values to provide an unbiased and reliable measurement of the model's performance.

3. Implementations

3.1 Tools

Python was selected to conduct the experiments, due to the availability of many well-documented open source packages related to data mining. It was also effective in automating the tuning process to find the best hyperparameters for each model. Moreover, Python was the most capable and accessible language to handle the processing and computation given our huge volume of data.

Several open source Python libraries were used in this project with links listed in **Appendix 8.3**:

- **Pandas**, a Python library for data manipulation and analysis, was used to preprocess the data.
- **Scikit-Learn** (sklearn), a Python machine learning library, provided solid implementations of a range of machine learning algorithms and several useful helper functions such as scaling data.
- **LightGBM**, a Python library implementing the gradient boosting framework with tree based learning algorithms, was used for implementing the tree-based gradient boosting model.
- **Keras,** a free and open-source software library for dataflow and differentiable programming across a range of tasks, was used for implementing the neural network model.

3.2 Data Preparation

3.2.1 Data Cleaning

Duplicate Data: There is no duplicate data in the dataset to be removed.

Missing Data: For each feature, we used the following rules to deal with missing values if any:

- If **more than half** of the attribute values are missing, **drop** the feature.
- If **less than 1%** of the attribute values are missing, **fill** the missing values by **mode** for categorical features, by **median** for numeric features.
- If **more than 1%** of the attribute values are missing, **fill** the missing values by **'Unknown'** for categorical features.

For the **Census_PrimaryDiskTypeName** feature, **0.14%** of the attribute values are missing in the dataset and **UNKNOWN is an attribute value** that occurred in 4.02% of the data. To deal with this, this 4.02% of missing data is **filled with 'UNKNOWN'**.

Invalid Values: There are invalid values in the following features.

- For the feature **SmartScreen**, **s**ome attributes were redundant or misspelled, resulting in multiple attribute values even though they are meant to be the same attribute value. For example, attribute value of 'prompt' are available, but so are attribute value of 'promt'. To counter this, a dictionary mapping (**Appendix 8.4)** is applied to the dataset to map invalid or redundant values to the correctly declared value.
- The following two features contains invalid values of -1, which is **converted to median**, as the display resolution cannot be negative:
    - **Census_InternalPrimaryDisplayResolutionHorizontal**
    - **Census_InternalPrimaryDisplayResolutionVertical**

Numeric Outliers: The feature **Census_PrimaryDiskTotalCapacity** contains 2 extremely high and impossible outlier values (**Appendix 8.5)** whose rows were removed from the dataset.

3.2.2 Feature Selection

When performing feature selection, the following features are dropped.

- The feature **MachineIdentifier** was dropped since it contains no valuable information.

- The **Census_OSSkuName** and **Census_OSEdition** features are almost identical. Thus, the **Census_OSSkuName** feature was dropped.
- Many features have 98% or more of the data contained in one category value (**Appendix 8.6)**. These features were dropped as the variance captured by these features are low.

3.2.3 Aggregation

Many features have over 1000 distinct attribute values, with relatively very low frequencies. To reduce cardinality, any category value that contains 0.1% or less of total data were grouped into 'others'. Besides, two features **OsVer** and **Census_OSVersion** were aggregated by the first number in the version number (either 6 or 10) to create two new features.

3.2.4 Label Encoding

The sklearn and TensorFlow libraries requires all the data to be numeric for model training. Thus, label encoding was performed on most categorical data in string format, such as **CountryIdentifier** or **CityIdentifier**, with the LabelEncoder helper object from sklearn library, which encoded the columns with numeric value between 0 and num_of_classes - 1.

3.2.5 Data Normalization

Normalization was done for all features with numeric/ratio data with the **MinMaxScaler** helper object from **sklearn** library, which transforms each feature by scaling it to the given range 0 to 1. Normalization a common requirement for many machine learning models which might behave badly if individual features do not have a common scale. This normalization operates under the assumption that all **features** are independent and have equal footing / chance to 'contribute' to the final result, which is whether the machine has malwares or not.

3.2.6 Feature Engineering

Feature engineering was done for the **AVSigVersion** and **Census_OSVersion** features. By intuition and logic, the newer the **AVSigVersion** (which is the antivirus/defender state version) or **Census_OSVersion** (the Windows OS version), the more security measures there would be in the Microsoft computer. However, these two features were discrete nominal data. Hence, we sourced for auxiliary data sources to map these two features to timestamp.

The dictionary mapping from AVSig/OS version number to date released were extracted from the following websites:

- https://www.microsoft.com/en-us/wdsi/definitions/antimalware-definition-release-notes
- https://support.microsoft.com/en-us/help/4043454/windows-10-windows-server-update-history https://changewindows.org/build/17134.

3.2.7 Data Splitting

As mentioned before, the **train.csv** dataset on Kaggle was split to a train and test dataset with 70% being training data and 30% being testing data. This was done using the **train_test_split** function in sklearn. Stratified sampling was also applied to make sure the distribution of machines with malware and without malware in train and test are almost, if not exactly the same.

3.3 Application of Machine Learning Algorithms

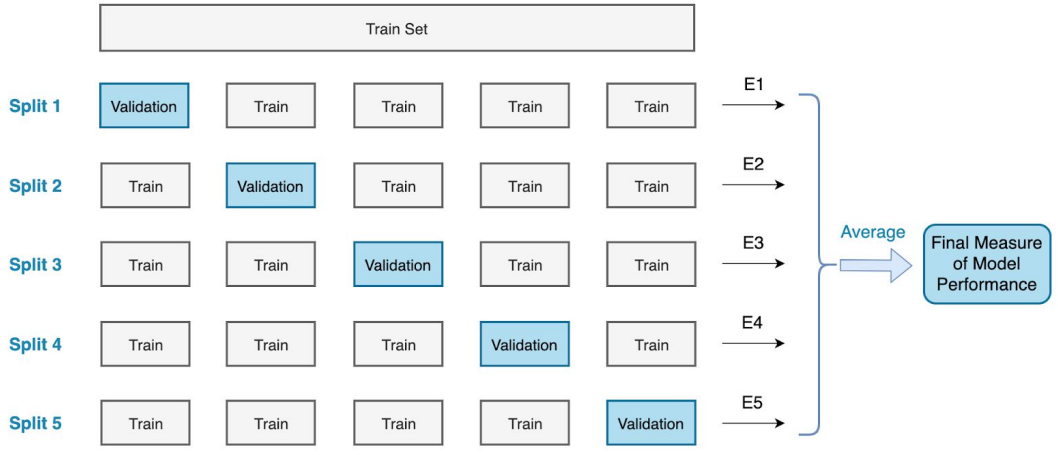
Different machine learning models were applied to the prepared data. The machine learning algorithms to be applied, as discussed previously, were Naïve Bayes, Logistic Regression, Decision Tree, Neural Network, Random Forest and Tree-based Gradient Boosting. ***Table 1*** below shows a summary of python packages used for implementing each algorithm.

Algorithm	Package and Class	Remarks

Naïve Bayes	sklearn.naive_bayes.GaussianNB[1]	Gaussian Naive Bayes estimates the distribution of the data with Gaussian (or Normal) distribution.
Logistic Regression	sklearn.linear_model.LogisticRegression[2]	The 'saga' solver was used because it is suitable for very large dataset (**Appendix 8.7**).
Decision Tree	sklearn.tree.DecisionTreeClassifier[3]	Gini impurity was used to measure the quality of a split during training.
Random Forest	sklearn.tree.RandomForestClassifier[4]	Gini impurity was used to measure the quality of a split during training. Bootstrap samples were used when building trees.
Tree-based Gradient Boosting	lightgbm.LGBMClassifier[5]	LightGBM is a gradient boosting framework that uses the leaf-wise tree growth algorithm, which can converge much faster than depth-wise growth. It is also designed to be distributed and efficient with the following advantages: - faster training speed and higher efficiency, - lower memory usage, - better accuracy, - support of parallel and GPU learning, - capable of handling large-scale data.
Bagging of Gradient Boosting	sklearn.ensemble.BaggingClassifier[6] + lightgbm.LGBMClassifier	BaggingClassifier automates the process of fitting base learners (LGBMClassifier in this case) on bootstrap samples and aggregating their individual predictions to form a final prediction.
Neural Network	keras.models.Sequential[7]	The Sequential model is a linear stack of layers.

Table 1. Python Packages for Algorithm Implementation

3.4 5-Fold Cross-Validation Grid-Search



[1] https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html
[2] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html
[3] https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html
[4] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html
[5] https://lightgbm.readthedocs.io/en/latest/pythonapi/lightgbm.LGBMClassifier.html
[6] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html
[7] https://keras.io/models/sequential/

Figure 2. 5-Fold Cross Validation Algorithm

The grid search provided by sklearn.model_selection.GridSearchCV was used to automate this 5-Fold Cross-Validation Grid-Search process for hyperparameter optimization. It exhaustively generates candidates from the specified grid of parameter values for each model, and then search the hyperparameter space for the best cross validation score. Each model was then trained with the optimized hyperparameters on the entire train set and then predict for the test set for model evaluation.

Figure 2 *above* shows how the 5-fold cross validation algorithm worked. First, the train set was randomly partitioned into 5 equal sized subsamples. Of the 5 subsamples, a single subsample was retained as the validation data for testing the model, while the remaining 4 subsamples were used as training data. The cross-validation process was repeated 5 times, with each of the 5 subsamples used exactly once as the validation data. The 5 evaluation results were then averaged to produce a single estimation of the model's performance with that hyperparameters setting.

4. Experiments and Results Analysis

4.1 Experimental Setup

Here are the steps involved to set up the experimental environment.

First, download data from the following kaggle website:
https://www.kaggle.com/c/microsoft-malware-prediction/data.

Second, install Python version 3.6.8 and required packages. The versions of python packages used for implementing machine learning algorithms are as listed in ***Table 2*** below.

pandas	0.25.1	**lightgbm**	2.3.0
scikit-learn	0.21.3	**keras**	2.3.0

Table 2. Required Python Packages

Third, to ensure the reproducibility of experimental results, a global seed of 2019 was set for all algorithms involving randomness, such as shuffling data before splitting, bootstrapping (random sampling) for random forest and initializing random weights for neural network models.

4.2 Comparison Schemes

The metric used to evaluate model performance was Area under the Receiver Operating Characteristic Curve (AUC). This metric was also used when conducting cross-validation grid search to evaluate and find the optimal hyperparameters for selected models. Each model's performance on the test set was also evaluated using AUC.

AUC was chosen because the Kaggle competition used AUC to evaluate the prediction submission for the competition's test set. It was also suitable for this dataset whose target class had a balanced distribution.

4.3 Hyperparameter Optimization

For Logistic Regression, Decision Tree, Random Forest and Gradient Boosting models, 5-Fold Cross-Validation Grid-Search was conducted to find their optimized hyperparameters. In sklearn and lightgbm packages, hyperparameters are passed as arguments to the constructor of the estimator classes. Due to time and resource constraints, selected hyperparameters were tuned for selected models. **Table 3** below shows a summary of the constructor arguments of the selected hyperparameters for selected models and the candidate parameter values **(Appendix 8.2.1)**.

Besides, the number of training epochs for neural network was also optimized to prevent underfitting and overfitting, with a hold-out validation set.

Algorithm	**Constructor Arguments**	**Meaning**	**Candidate Values**
Logistic Regression	C	Inverse of the regularization strength	0.001, 0.01, 0.1, 1, 10, 100, 1000
	penalty	Type of regularization technique	l1, l2
Decision Tree	min_samples_split	The minimum number of samples required to split an internal node.	1600, 1800, 2000, 2200, 2400
Random Forest	min_samples_split	The minimum number of samples required to split an internal node.	50, 100, 200, 300
	n_estimators	Number of base learners.	50, 100, 200, 300, 400
Gradient Boosting	num_leaves	Maximum tree leaves for base learners.	500, 1000, 1500, 2000, 2500
	n_estimators	Number of base learners.	200, 400, 600, 800, 1000

Table 3. Selected Hyperparameters and Candidate Values

4.3.1 Logistic Regression

The regularization technique and regularization strength of the logistic regression model was fine tuned using cross validation. The class sklearn.linear_model.LogisticRegression uses the constructor argument *'penalty'* for the type of regularization, and *'C'* for the inverse of the regularization strength. Regularization applied a penalty for magnitude of parameter values in order to reduce overfitting. The candidates parameter for *'penalty'* were L2 regularization (square of the magnitude of coefficients) and L1 regularization (absolute value of the magnitude of coefficients), while the candidates parameter for *'C'* were 0.001, 0.01, 0.1, 1, 10, 100 and 1000. As shown in **Appendix 8.8 Table 8**, the best model setting was with L1 regularization and '*C'* as 1000, thus the regularization strength as 0.0.001, achieving the best cross-validation score of 0661.

4.3.2 Decision Tree

To prevent the decision tree model from overfitting, pre-pruning was applied to stop the algorithm before it becomes a full-grown tree. The condition was set such that the algorithm would stop splitting a node if the number of instances was less than a specified threshold. Cross validation was used to find the best threshold for the minimum number of samples required to be at a leaf node, which was represented by *'min_samples_split'* constructor argument in the class sklearn.tree.DecisionTreeClassifier. As shown in **Appendix 8.8 Table 9**, the optimized *'min_samples_split'* was 2000 achieving a highest cross-validation score of 0.704.

4.3.3 Random Forest

The random forest model implemented by sklearn fits a number of base decision tree models on various bootstrap samples of the dataset and uses averaging of the probability to improve the predictive accuracy and control over-fitting. Each base decision tree model was pre-pruned using the same technique as the decision tree model in **Section 4.3.2**, which was to stop growing a node when the number of instances in the node was less than a specified threshold. Same as the DecisionTreeClassifier class, the sklearn class RandomForestClassifier also has the '*min_samples_split'* attribute which determines the minimum number of samples required to split a node. Apart from *'min_samples_split'*, the number of estimators to ensemble was another hyperparameter to optimize. In general, the more trees, the better the results. However, as the number of trees increases above a certain point, the benefit in prediction performance

improvement from learning more trees would be lower than the cost in computation time for learning these additional trees. Thus, cross validation was used to find the best combination of *'min_samples_split'* and *'n_estimator'* (represents the number of base trees in the forest) which could balance the trade off between model performance and training time.

As shown in **Appendix 8.8 Table 10**, the best combination is when *'min_samples_split'* was 50 and *'n_estimators'* was 400, which achieved a cross-validation score of 0.726. It is worth noticing that, with *'min_samples_split'* as 50 when number of base learners increased from 300 to 400, the validation score increased marginally, hinting that 400 was the point where the performance improvement from learning more trees would be lower than the cost in computation time.

One interesting finding was that the best *'min_samples_split'* for random forest (50) was much lower than that for a single decision tree (2000). The reason could be that the bagging reduced overfitting and benefited from the diversity of base tree learners.

4.3.4 Gradient Boosting

For the gradient boosting model, each base decision tree in the gradient boosting model was pre-pruned by setting a rule such that the algorithm would stop growing when the number of leaf nodes reached a specified value. To get good results using the leaf-wise tree growth algorithm implemented by LightGBM, the number of maximum leaf nodes is the main parameter to control the complexity of the tree model, and thus prevent overfitting ("Parameters Tuning"). The class lightgbm.LGBMClassifer used the *'num_leaves'* attribute to determine the number of maximum leaf nodes for all the base tree learners. Apart from *'min_samples_split'*, the number of estimators to ensemble was another hyperparameter to optimize. The model could underfit when the number of estimators is too small, or overfit if the number of estimators is too large. The attribute *'n_estimators'* of the class LGBMClassifer set the number of base learners (or training iterations). As shown in **Appendix 8.8 Table 11**, the best combination was when *'num_leaves'* was 1500 and *'n_estimators'* was 600, achieving a cross-validation score of 0.739.

4.3.5 Bagging of Gradient Boosting

To combine the techniques of both bagging and boosting, we trained 100 tree-based gradient boosting models (LightGBM) on different bootstrap samples, after which individual predicted probabilities were aggregated by averaging to form the final prediction. Three important hyperparameters were identified, namely number of base learners and number of leave nodes in each gradient boosting model, and the number of gradient boosting models. However, due to time and resource constraints, hyperparameter tuning were not performed. As observed previously, bagging could reduce overfitting and benefit from the diversity of base learners. Thus, larger numbers of leaf nodes (5000) and estimators (1000) were set for the base gradient boosting model compared to the found best hyperparameters in **Section 4.3.4**.

4.3.6 Neural Network

The feedforward neural network **(Appendix 8.2.3)** had 1 input layer, 2 densely connected hidden layers, each with 128 neurons, and 1 output layer. For each hidden layer, dropout was first applied to randomly drop out values of 40% nodes, offering a computationally cheap and remarkably effective regularization method to reduce overfitting. Batch normalization was also applied to enable faster and more stable training of the model. It made the optimization landscape significantly smoother, thus inducing a more predictive and stable behaviour of the gradients, allowing for faster training. After the batch normalization, the Rectified Linear Unit (relu) activation function was applied to add non-linearity to the model. For the output layer, the sigmoid activation function was used due to the nature of the prediction task which was binary classification.

During the training phase **(Appendix 8.2.4)**, Early Stopping was applied to the model to find the optimal number of epochs and prevent both underfitting and overfitting. The train set was shuffled and split into a train and validation set in a ratio of 7:3. At the end of each epoch (when all training instances had propagated in the model), the model was evaluated on the hold-out validation set. As the keras package did not provide an existing AUC scoring metric, a custom metric function was implemented **(Appendix 8.2.2).** If the AUC score of the model on the validation dataset did not improve for continuous 5 rounds, then the training process was stopped. Finally, the model restored the model weights from the epoch with the best value of the monitored quantity, ROC score.

As shown in **Appendix 8.9 figure 5**, the training for the model stopped at the 6th epoch, because the AUC score for the validation set did not improve for 5 epochs after the 1st epoch.

4.4 Model Performance Comparison

After hyperparameter optimization, all algorithms were trained with the best hyperparameters as shown in **Table 4** below on the train dataset, and predicted for the test dataset. The predicted probability and actual label for 'HasDetection' were then used to calculate the AUC score. The Receiver Operating Characteristic (ROC) Curves for each model on the test data can be found in **Appendix 8.10**.

Algorithm	**Hyperparameters**
Naïve Bayes	NA.
Logistic Regression	L1 regularization with regularization strength as 0.0001.
Decision Tree	Minimum number of samples required to be at a leaf node as 2000
Random Forest	Number of trees as 400. Minimum number of samples required to be at a leaf node in each tree as 50
Tree-based Gradient Boosting	Number of trees as 600. Maximum leaf nodes in each tree as 1500.
Bagging of Gradient Boosting	Number of gradient boosting models as 50. Number of trees in each boosting model as 600. Maximum leaf nodes in each tree as 4000.
Neural Network	Number of epoch

Table 4. Optimized Hyperparameters

As shown in ***Figure 3*** below, the Bagged Gradient Boosting algorithm achieved the highest AUC score.

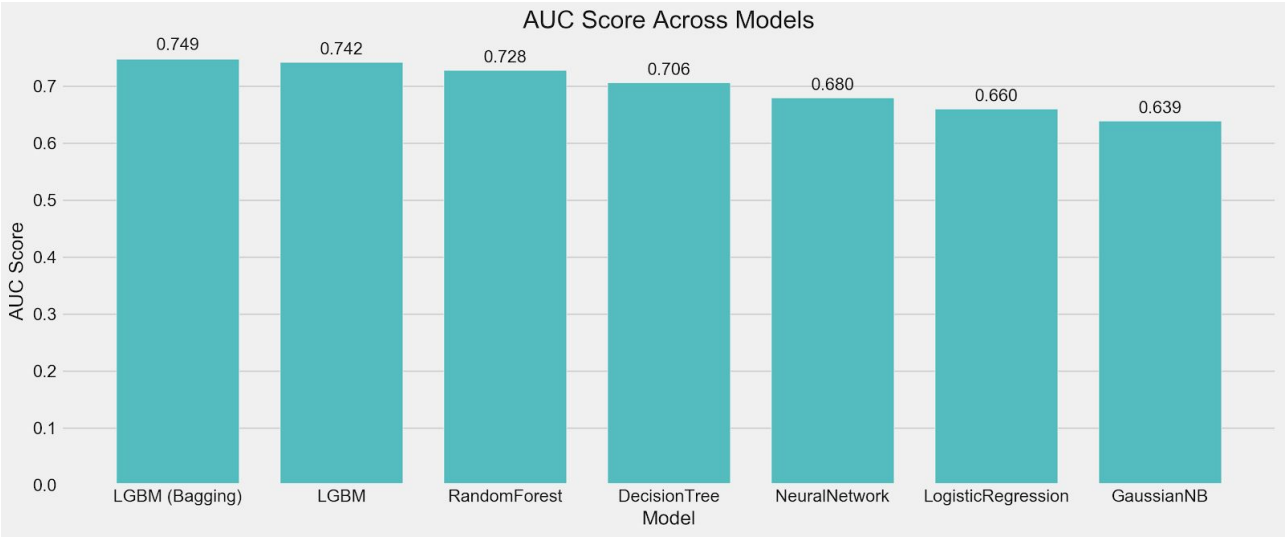


Figure 3. AUC Score Across Models

5. Discussion of Pros and Cons

5.1 Label Encoding

To deal with non-numeric categorical variables, we used label encoding which encoded labels with value between 0 and n_classes-1. This converted values into machine-readable format with the following advantages:

1. It requires lower memory usage as the number of variables after conversion is unchanged.
2. It avoids the curse of dimensionality due to less resulting columns.

These were especially important for our dataset, which consisted of mainly categorical features and had a large number of levels for some features. One-hot encoding would lead to high memory usage and high dimensionality in this case, since the high cardinality would increase the feature space tremendously.

However, as most categorical columns are nominal, the imposed ordinality by label encoding would not represent the attribute values well and undermine some models' performance (eg.. Neural network model would treat nominal data as ratio data).

5.2 5-Fold Cross-Validation Grid-Search

When performing the 5-Fold Cross-Validation Grid-Search, the intervals between candidates values were quite larger for some hyperparameters, eg. min_samples_split for Decision Tree. This reduced fine tuning time and saved computational power as compared to much finer intervals. However, the drawback is that this found optimal hyperparameters might not be the best ones.

6. Conclusions

6.1 Project Achievements

With the optimized hyperparameters, we trained each model on the entire original train set from Kaggle and submitted predictions for the Kaggle test set. The model using Bagging of LGBMClassifier got the highest private score of 0.64170, placing us on the 338th place. Given there were 2412 submissions, our Kaggle Leaderboard Ranking would be the Top 14% out of all submissions.

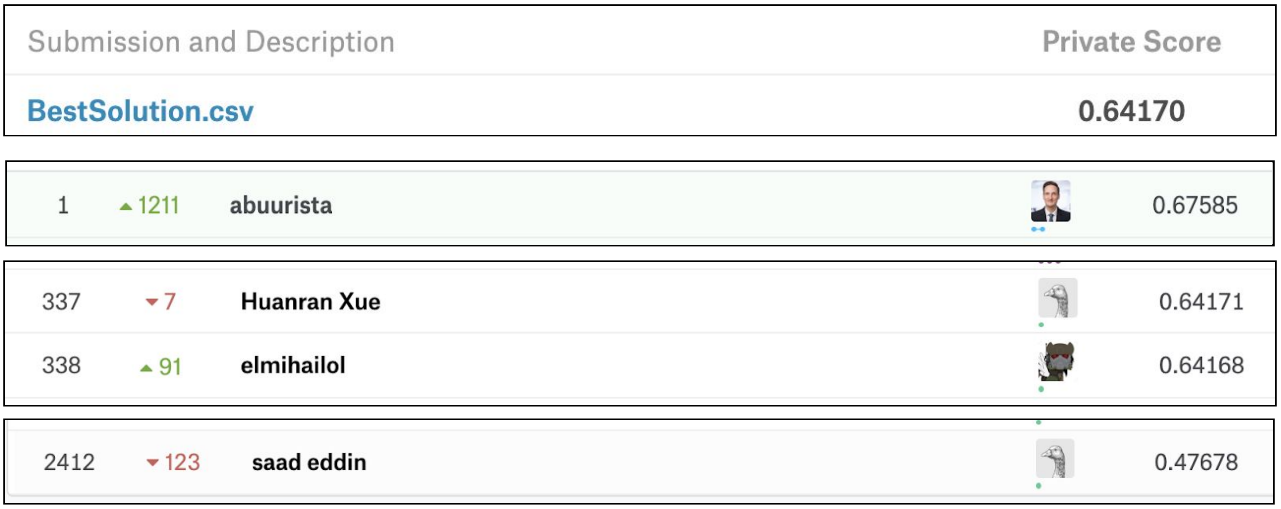


Figure 4. Kaggle Score and Ranking

6.2 Directions for Improvements

6.2.1 More Advanced Encoding

As discussed previously, label encoding is not suitable for nominal variables. Thus, alternative encoding techniques could be used, eg. target encoding and frequency encoding.

Target encoding is the process of replacing each unique value of the categorical feature with the ratio of occurrence of the positive class in the target variable. The new feature represents a probability of the target variable, conditional on each value of the feature, thus embodying the target variable in its encoded value.

Frequency Encoding is a way to utilize the frequency of the categories as labels. In the cases where the frequency is related somewhat with the target variable, it helps the model to understand and assign the weight in direct and inverse proportion, depending on the nature of the data. This applies to variables that show a positive or negative association between frequency and detection rate, such as Census_OSInstallTypeName **(Appendix 8.11)**. The rationale is that a hacker might prefer finding leaks of popular software. If one software has many users, and then if the hacker attacks successfully, he or she will get many profits.

6.2.2 One Hot Encoding with Dimensionality Reduction

The reason for not using one-hot-encoding was due to the high cardinality of the transformed data, which could increase the memory usage tremendously and lead to the problem of curse of dimensionality. However, this encoding technique has the advantage that the result is binary rather than ordinal and that everything sits in an orthogonal vector space. In these cases, we could employ one-hot-encoding followed by Principal Component Analysis (PCA) for dimensionality reduction. PCA finds the linear overlap, so will naturally tend to group similar features into the same feature.

6.2.3 Finer Model Tuning

Due to time and resource constraints, only a few selected hyperparameters of some models were tuned. More hyperparameters could be be tuned for each model, such as number of layers, number of neurons for each layer and learning rate for Neural Network. Tuning more hyperparameters could potentially improve the performance of the model.

Also, the selected hyperparameters were tuned with a relatively large interval between each candidate value. For example, the optimal value of minimum samples to split for the decision tree was found from a list of numbers [1600, 1800, 2000, 2200, 2400] through cross validation. A potential way to improve the model performance could be to use smaller intervals.

6.2.4 Apply model stacking

Models trained using different machine learning algorithm, hyperparameter settings, training sets and features provides a huge diversity. Model stacking is another ensemble technique to combine models and take advantage of such diversity to improve prediction. Thus, it is recommended to experiment with training a meta model on the predictions from the base models to form a new set of prediction.

6.2.5 Walk-forward Cross Validation

It was later observed that the Kaggle train and test set were split in a time series manner, where the majority of train data were in August and September 2018 while test data were in October and November 2018. However, for our own train and test split, we simply used a stratified split with the target variable, regardless of the time factor. This explained why there was a huge discrepancy between the AUC score of our own test set and that of the Kaggle test set on the Leaderboard.

Thus, to better mimic the train and test set relationship, it is recommended to use walk-forward cross validation when optimizing hyperparameters and selecting models. Walk-forward validation would provide a more realistic evaluation of models on such a time series data.

6.2.6 Recursive Feature Elimination (RFE)

Recursive feature elimination (RFE) is a feature selection method that fits a model and removes the weakest feature (or features) until the specified number of features or optimal set of features is reached. Features are ranked by the model's ***coef_*** or ***feature_importances_*** attributes, and by recursively eliminating a small number of features per loop, RFE attempts to eliminate dependencies and collinearity that may exist in the model. This could potentially be applied as an extension of the manual dimensionality reduction method outlined above.

RFE requires a specified number of features to keep, however it is often not known in advance how many features are valid. To find the optimal number of features cross-validation is used with RFE to score different feature subsets and select the best scoring collection of features.

However, RFE comes with a time cost problem since it involves training and fitting the model exceedingly many times to find the best set of features for optimal training.

7. References

1. Lau, L. (2018, February 23). Cybercrime 'pandemic' may have cost the world $600 billion last year. Retrieved from https://www.cnbc.com/2018/02/22/cybercrime-pandemic-may-have-cost-the-world-600-billion-last-year.html.

2. Tahir, R. (2018). A Study on Malware and Malware Detection Techniques. International Journal of Education and Management Engineering, 8(2), 20–30. doi: 10.5815/ijeme.2018.02.03

3. Souri A, Norouzi M, Asghari P (2017) An analytical automated refinement approach for structural modeling large-scale codes using reverse engineering. Int J Inf Technol 9:329–333.

4. Rhode, M., Burnap, P., & Jones, K. (2018). Early-stage malware prediction using recurrent neural networks. Computers & Security, 77, 578–594. doi: 10.1016/j.cose.2018.05.010

5. Dong, Y. (2017). Android Malware Prediction by Permission Analysis and Data Mining.

6. Parameters Tuning¶. (n.d.). Retrieved from https://lightgbm.readthedocs.io/en/latest/Parameters-Tuning.html.

8. Appendix

8.1 Datasets

The dataset is obtained from Microsoft's Kaggle competition for malware prediction: https://www.kaggle.com/c/microsoft-malware-prediction/data

8.2 Source Codes

8.2.1 Set 5-fold Cross-Validation Grid

```
estimators_params_grid = {
   'LogisticRegression': {'C' : [10**i for i in range(-3,4)], 'penalty': ['l2', 'l1']},
   'DecisionTreeClassifier': {'min_samples_split': [1600, 1800, 2000, 2200, 2400]},
   'RandomForestClassifier': {'n_estimators' : [50,100,200,300,400],
                              'min_samples_split': [50, 100, 150, 200]},
   'LGBMClassifier': {'num_leaves': [500, 1000, 1500, 2000, 2500],
                      'n_estimators': [200, 400, 600, 800, 1000]}}
```

8.2.2 Implement Custom Metric

```
from tensorflow import numpy_function, double
from sklearn.metrics import roc_auc_score
def auroc(y_true, y_pred):
   return numpy_function(roc_auc_score, (y_true, y_pred), double)
```

8.2.3 Build Neural Network Model

```
from keras.models import Sequential
from keras.layers import Dense, Dropout, BatchNormalization, Activation
def build_nn(train_x):
   model = Sequential()
   model.add(Dense(128,input_dim=train_x.shape[1]))
   model.add(Dropout(0.4))
   model.add(BatchNormalization())
   model.add(Activation('relu'))
   model.add(Dense(128))
   model.add(Dropout(0.4))
   model.add(BatchNormalization())
   model.add(Activation('relu'))
   model.add(Dense(1, activation='sigmoid'))
   model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy', auroc])
   return model
```

8.2.4 Train Neural Network Model

```
from keras.callbacks import EarlyStopping
def train_nn(model, train_x, train_y):
   history = model.fit(train_x, train_y, epochs = 1000, batch_size=128, validation_split = 0.3,
                       callbacks = [EarlyStopping(patience = 5, monitor='val_auroc', mode='max',
                       restore_best_weights = True)])
   return model
```

8.3 Links of Used Python Packages

- **Pandas:** https://pandas.pydata.org/pandas-docs/stable/index.html
- **Scikit-Learn**: https://scikit-learn.org/stable/
- **LightGBM**: https://lightgbm.readthedocs.io/en/latest/
- **Keras:** https://keras.io/.

8.4 SmartScreen Mapping Pairs

Origin	Promt, promprt	00000000, deny	enabled	of	requiredadmin
Mapping	prompt	0	on	off	requireadmin

Table 5. SmartScreen Mapping Pairs

8.5 Outliers for Census_PrimaryDiskTotalCapacity

As shown by the boxplot below, there are two **outliers** (8160436813824 MB and 6523912192000 MB) for primary disk total capacity.

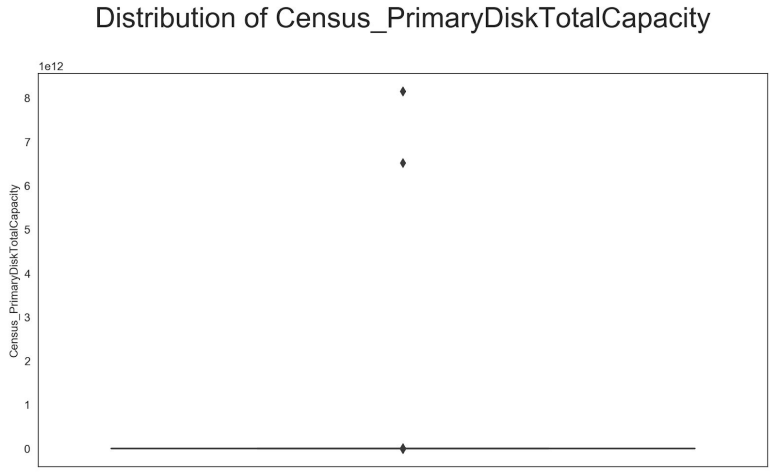


Figure 5. Neural Network Training History

8.6 Highly Skewed Categorical Features

Here are the columns with more than 98% data contained in one category value. There are 8,921,483 rows in total.

Column	Mode	Mode Count	Percentage
ProductName	win8defender	8921483	0.9894
IsBeta	0	8921483	1.0000
IsSxsPassiveMode	0	8921483	0.9827
HasTpm	1	8921483	0.9880
AutoSampleOptIn	0	8921483	1.0000
PuaMode	on	2309	0.9991
SMode	0	8383724	0.9995
UacLuaenable	1	8910645	0.9939
Census_DeviceFamily	Windows.Desktop	8921483	0.9984
Census_IsPortableOperatingSystem	0	8921483	0.9995
Census_IsFlightingInternal	0	1512724	1.0000
Census_IsFlightsDisabled	0	8760960	1.0000
Census_ThresholdOptIn	0	3254158	0.9997
Census_IsWIMBootEnabled	0	3261780	1.0000
Census_IsVirtualDevice	0	8905530	0.9930

Table 6. Features have 98% or more of the data contained in one category value

8.7 Logistic Regression Solvers

The following table summarizes the penalties supported by each solver and its behaviours[8].

	Solvers				
Penalties	'liblinear'	'lbfgs'	'newton-cg'	'sag'	'saga'

[8] https://scikit-learn.org/stable/modules/linear_model.html

Multinomial + L2 penalty	no	yes	yes	yes	yes
OVR + L2 penalty	yes	yes	yes	yes	yes
Multinomial + L1 penalty	no	no	no	no	yes
OVR + L1 penalty	yes	no	no	no	yes
Elastic-Net	no	no	no	no	yes
No penalty (‘none’)	no	yes	yes	yes	yes
Behaviors					
Penalize the intercept (bad)	yes	no	no	no	no
Faster for large datasets	no	no	no	yes	yes
Robust to unscaled datasets	yes	yes	yes	no	no

Table 7. Logistic Regression Solvers

8.8 Cross-Validation Grid-Search Results

Model Hyperparameters	**Mean Validation Score**	**Rank**
{'C': 0.001, 'penalty': 'l1''}	0.6568927161	13
{'C': 0.01, 'penalty': 'l1''}	0.6596046751	11
{'C': 0.1, ‘penalty': 'l1'}	0.6604657098	9
{'C': 1, ‘penalty': 'l1'}	0.6605199312	7
{'C': 10, ‘penalty': 'l1'}	0.6605252364	5
{'C': 100, ‘penalty': 'l1'}	0.6605257285	3
{'C': 1000, 'penalty': 'l1'}	**0.6605257787**	**1**
{'C': 0.001, 'penalty': ‘l2'}	0.6565910161	14
{'C': 0.01, 'penalty': ‘l2'}	0.6590684418	12
{'C': 0.1, 'penalty': ‘l2'}	0.6602401236	10
{'C': 1, 'penalty': ‘l2'}	0.6604891429	8
{'C': 10, 'penalty': ‘l2'}	0.6605219984	6
{'C': 100, 'penalty': 'l2'}	0.6605253979	4
{'C': 1000, 'penalty': 'l2'}	0.6605257426	2

Table 8. Logistic Regression Model Cross Validation Results

Model Hyperparameters	**Mean Validation Score**	**Rank**
{'min_samples_split': 1600}	0.7036623937	4
{'min_samples_split': 1800}	0.7037129994	3
{'min_samples_split': 2000}	**0.7037383798**	**1**
{'min_samples_split': 2200}	0.7037163711	2
{'min_samples_split': 2400}	0.7035273703	5

Table 9. Decision Tree Model Cross Validation Results

Model Hyperparameters	Mean Validation Score	Rank
{'min_samples_split': 50, 'n_estimators': 50}	0.7215	15
{'min_samples_split': 100, 'n_estimators': 50}	0.7226	12
{'min_samples_split': 150, 'n_estimators': 50}	0.7214	16
{'min_samples_split': 200, 'n_estimators': 50}	0.7201	18
{'min_samples_split': 50, 'n_estimators': 100}	0.7237	9
{'min_samples_split': 100, 'n_estimators': 100}	0.7240	6
{'min_samples_split': 150, 'n_estimators': 100}	0.7223	13
{'min_samples_split': 200, 'n_estimators': 100}	0.7208	17
{'min_samples_split': 50, 'n_estimators': 200}	0.7248	3
{'min_samples_split': 100, 'n_estimators': 200}	0.7239	7
{'min_samples_split': 150, 'n_estimators': 200}	0.7195	19
{'min_samples_split': 200, 'n_estimators': 200}	0.7129	20
{'min_samples_split': 50, 'n_estimators': 300}	0.7252	2
{'min_samples_split': 100, 'n_estimators': 300}	0.7241	5
{'min_samples_split': 150, 'n_estimators': 300}	0.7236	10
{'min_samples_split': 200, 'n_estimators': 300}	0.7221	14
{'min_samples_split': 50, 'n_estimators': 400}	**0.7255**	**1**
{'min_samples_split': 100, 'n_estimators': 400}	0.7243	4
{'min_samples_split': 150, 'n_estimators': 400}	0.7238	8
{'min_samples_split': 200, 'n_estimators': 400}	0.7232	11

Table 10. Random Forest Model Cross Validation Results

Model Hyperparameters	Mean Validation Score	Rank
{'n_estimators': 200, 'num_leaves': 500}	0.7349366527	25
{'n_estimators': 200, 'num_leaves': 1000}	0.7372330846	24
{'n_estimators': 200, 'num_leaves': 1500}	0.7379936459	22
{'n_estimators': 200, 'num_leaves': 2000}	0.7384338445	18
{'n_estimators': 200, 'num_leaves': 2500}	0.7386285803	15
{'n_estimators': 400, 'num_leaves': 500}	0.7375049364	23
{'n_estimators': 400, 'num_leaves': 1000}	0.7388561893	12
{'n_estimators': 400, 'num_leaves': 1500}	0.7391202681	8
{'n_estimators': 400, 'num_leaves': 2000}	0.7391998033	3
{'n_estimators': 400, 'num_leaves': 2500}	0.7391489225	6
{'n_estimators': 600, 'num_leaves': 500}	0.7382071298	20
{'n_estimators': 600, 'num_leaves': 1000}	0.7391582693	5

{'n_estimators': 600, 'num_leaves': 1500}	**0.7392373723**	**1**
{'n_estimators': 600, 'num_leaves': 2000}	0.7390686261	9
{'n_estimators': 600, 'num_leaves': 2500}	0.7388959516	11
{'n_estimators': 800, 'num_leaves': 500}	0.7386088781	16
{'n_estimators': 800, 'num_leaves': 1000}	0.7392197189	2
{'n_estimators': 800, 'num_leaves': 1500}	0.7391266416	7
{'n_estimators': 800, 'num_leaves': 2000}	0.7387525730	14
{'n_estimators': 800, 'num_leaves': 2500}	0.7384926651	17
{'n_estimators': 1000, 'num_leaves': 500}	0.7388328662	13
{'n_estimators': 1000, 'num_leaves': 1000}	0.7391678074	4
{'n_estimators': 1000, 'num_leaves': 1500}	0.7389135514	10
{'n_estimators': 1000, 'num_leaves': 2000}	0.7383642135	19
{'n_estimators': 1000, 'num_leaves': 2500}	0.7380119238	21

Table 11. Gradient Boosting Model Cross Validation Results

8.9 Neural Network Training History

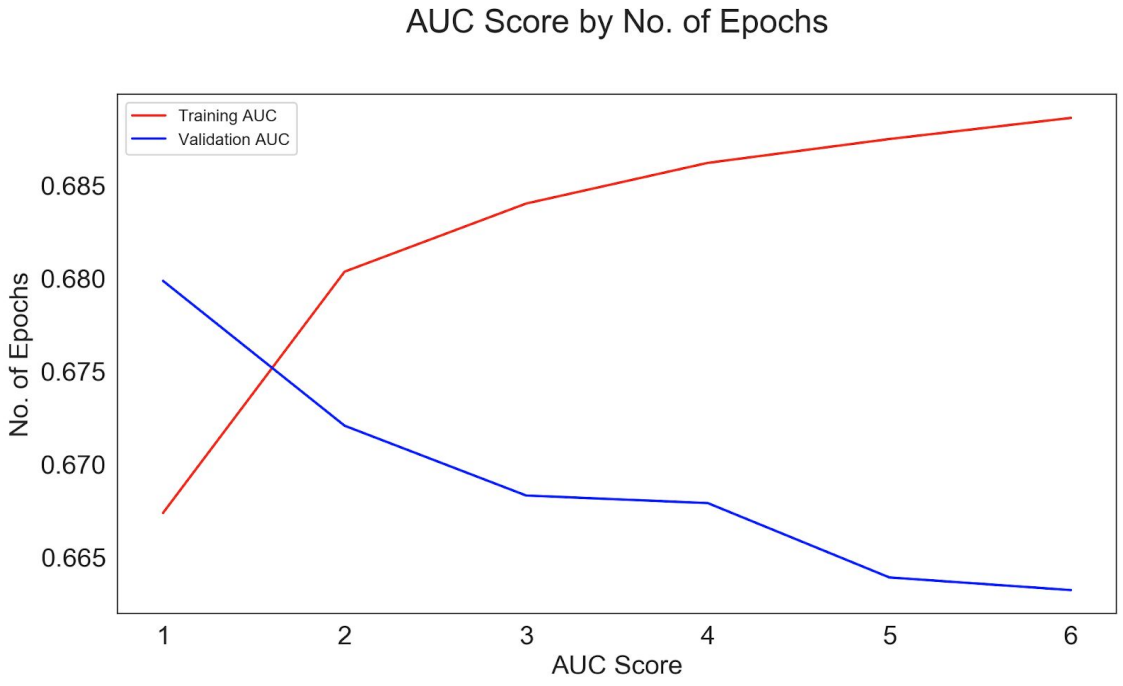
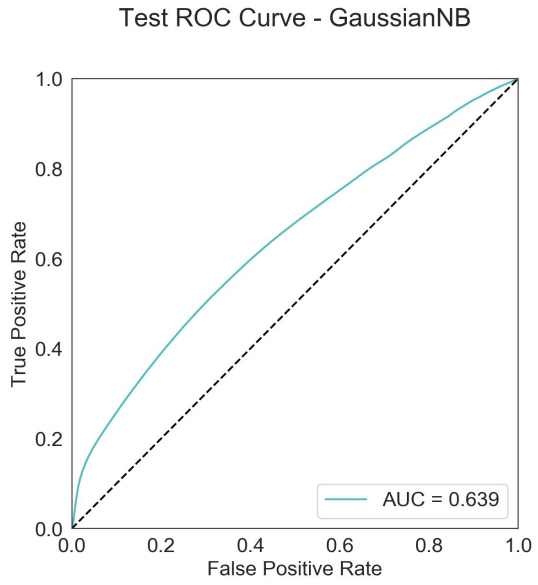
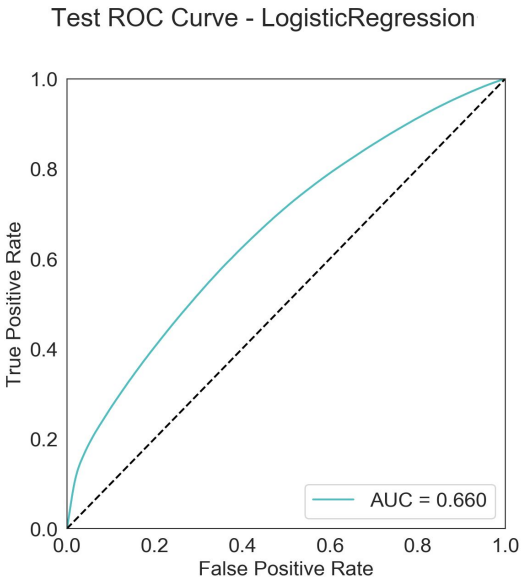
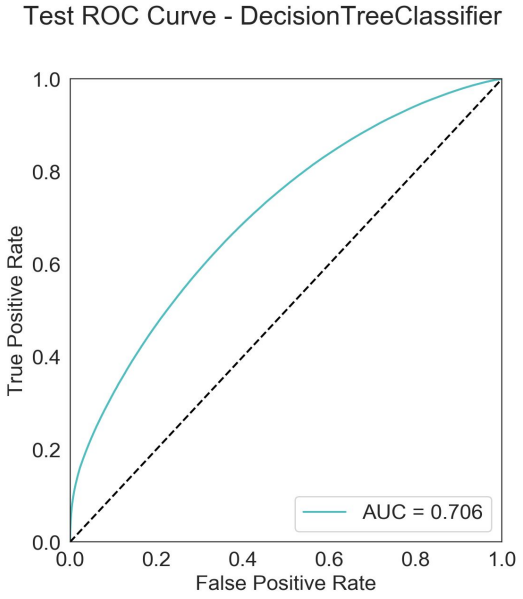


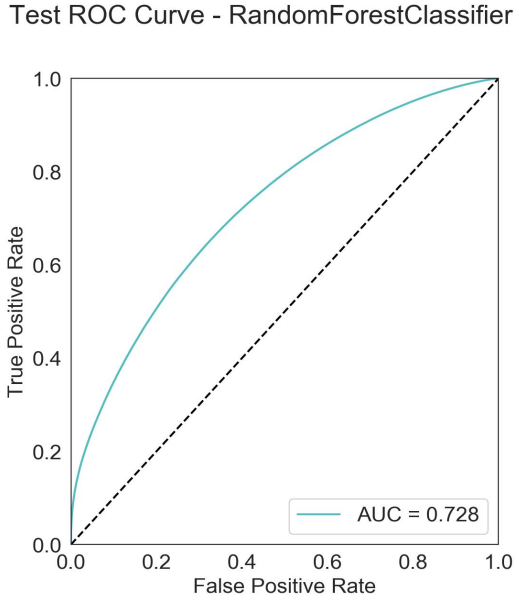
Figure 6. Neural Network Training History

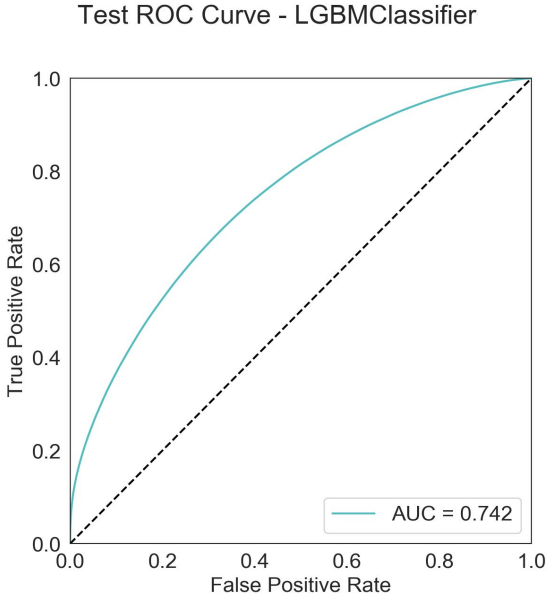
8.10 ROC Curve for Test Set

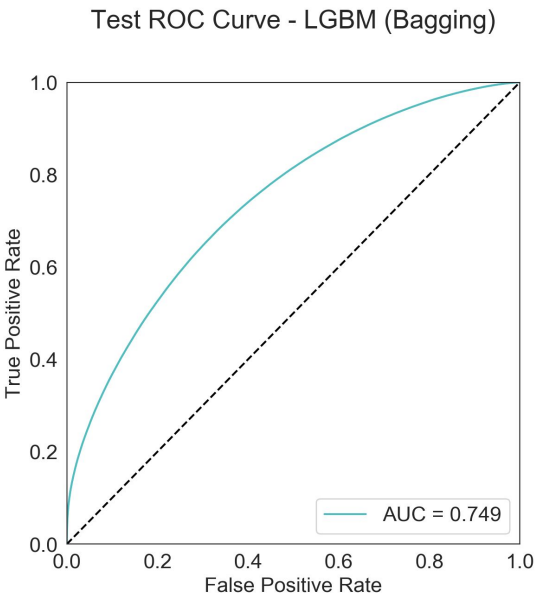


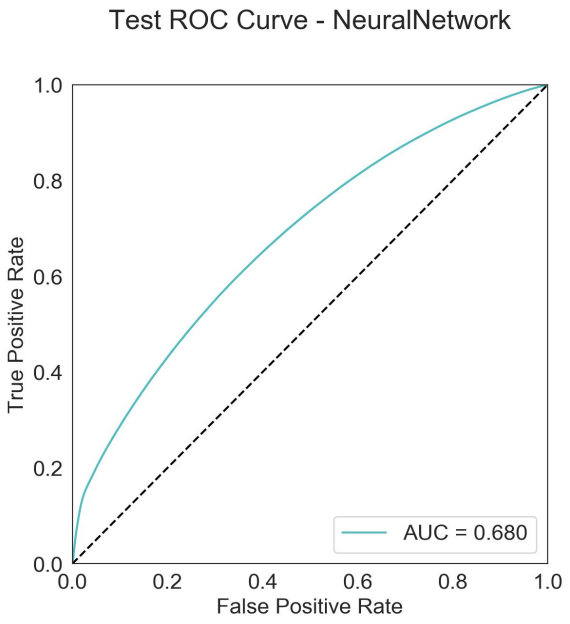




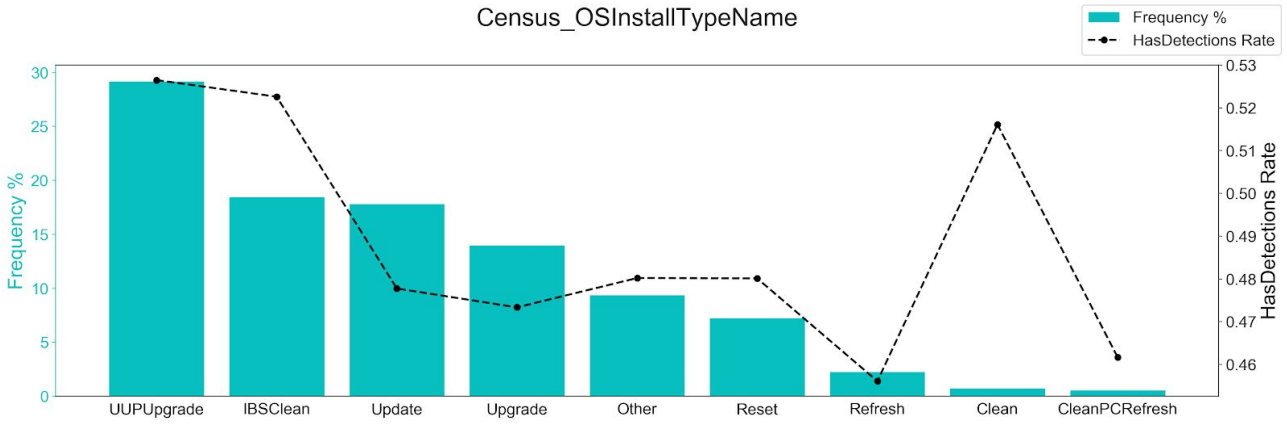


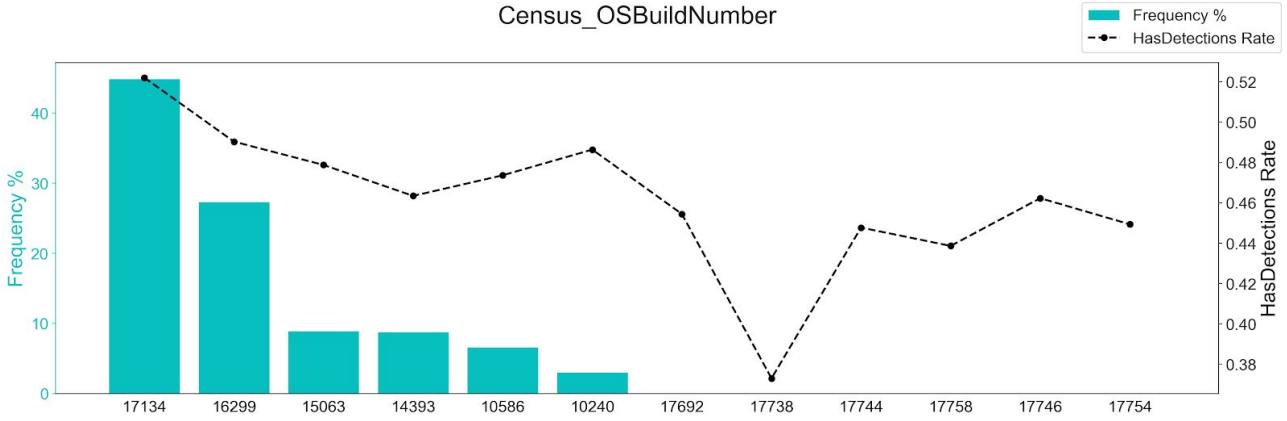


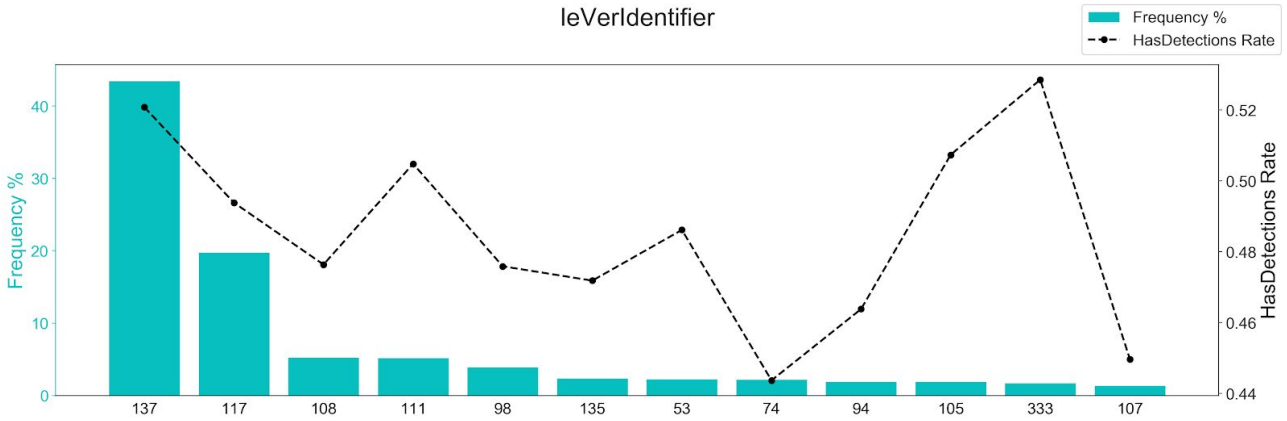


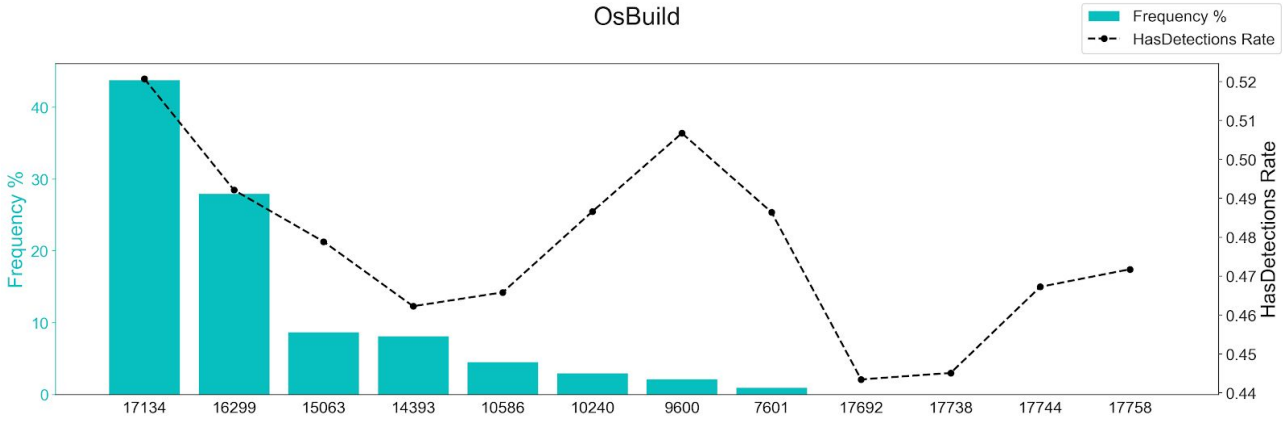


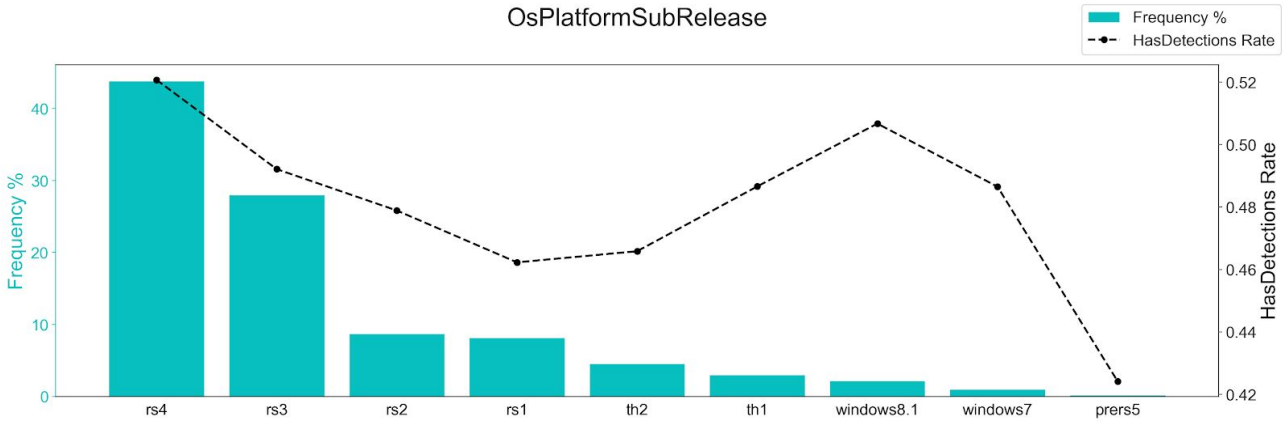
8.11 Features with Frequency and Detection Rate Associated

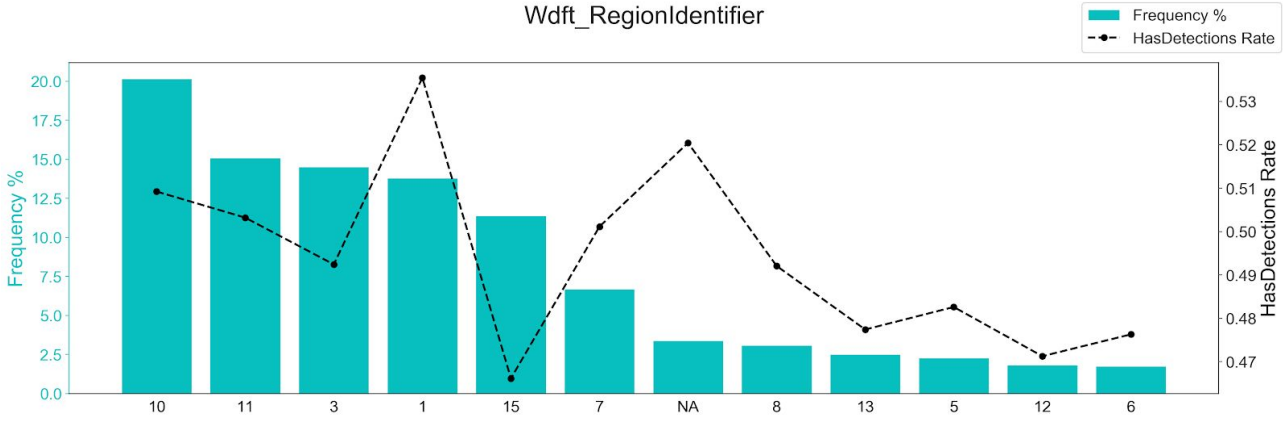





IeVerIdentifier
Frequency %
HasDetections Rate
Frequency %
HasDetections Rate
0
10
20
30
40
0.44
0.46
0.48
0.50
0.52
137
117
108
111
98
135
53
74
94
105
333
107


OsBuild
Frequency %
HasDetections Rate
Frequency %
HasDetections Rate
0
10
20
30
40
0.44
0.45
0.46
0.47
0.48
0.49
0.50
0.51
0.52
17134
16299
15063
14393
10586
10240
9600
7601
17692
17738
17744
17758


OsPlatformSubRelease
Frequency %
HasDetections Rate
Frequency %
HasDetections Rate
0
10
20
30
40
0.42
0.44
0.46
0.48
0.50
0.52
rs4
rs3
rs2
rs1
th2
th1
windows8.1
windows7
prers5


Wdft_RegionIdentifier
Frequency %
HasDetections Rate
Frequency %
HasDetections Rate
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
0.47
0.48
0.49
0.50
0.51
0.52
0.53
10
11
3
1
15
7
NA
8
13
5
12
6