Microsoft Malware Detection Using Ensemble Methods

Jungin Hong, junginh@uci.edu
Steven Seader, sseader@uci.edu
Rikako Yamamoto, rikakoy@uci.edu

Abstract

Our project seeks to predict malware presence in machines running Microsoft Windows (based on a large data set) using various ensemble methods. In this paper we investigate several methods of feature elimination and ultimately end with a classification algorithm that predicts malware presence with an accuracy of 65 percent. We began by eliminating features that are either entirely, or only slightly better than, useless in preparation for training, using an assortment of techniques. We then applied several classification methods, trained on the data we prepared. Finally, we compared each model's complexity and accuracy/error rate and chose classification method that yielded the best score.

1. Introduction

Malware detection as a whole is an extremely important ability of any business seeking to provide personal safety and/or privacy. The ability to detect malware allows a company to take steps toward mitigating and removing threats to users of their products, and in the case of non-threatening malware, to ensure that the customer's experience while using their product runs smoothly and correctly.
With this in mind, it is understandable that Microsoft would want the the ability to accurately classify malware-infected machines given a set of features and characteristics: not only would this allow them to make their malware detection more efficient, but it would also allow them to both predict machines that may become infected in the future as well as note common characteristics of infected machines (such as software versions, hardware versions, types of antivirus, etc.) in order to potentially seal points of entry for malware before it becomes common.

2. Related Work

Detecting malware in software has been successful with several softwares. However, writing malicious software is becoming easier and easier with the use of more advanced tools, to mention nothing of the level of complexity to which malware has grown.[9] Therefore, newer versions of malware detection studies are necessary. There are obviously a nearly infinite number of ways to do this, but there is information available that can lead this hunt:

- Boosted decision trees are a viable method of classification (as an alternative to Naive Bayes and Support Vector Machines)[1]
- Random Forest and KNN provide superior results when compared to Naive Bayes, Support Vector Machines, and Logistic Regression. [2]
- Using a Perceptron and Neural Network (multilayer perceptron) for the purpose of detecting malware has been shown to be capable of providing a zero false positive rate. [3]

Based on previous work done, we know that a solid approach would be based in decision trees and perceptron classification. Armed with this knowledge, we decided to pursue a similar route, using a variety of trees and methods using trees.

3. Dataset

Our Microsoft Malware Detection is trained on the Kaggle Microsoft Malware Prediction dataset [11], which contains just under 9 million machines in the provided training set, and just under 8 million machines in the provided test set. Each machine in the dataset has 83 features (though there are some machines that do not have data for a given feature). The data we used for training was roughly 1/18th of the provided training data (500,000 individual machines).

When preprocessing, we sought to remove features that would provide little-to-no information upon analysis in order to save time when processing the data. [10]

Initially, We removed a feature that was included as a way to identify each machine, but due to the way we planned to manage the data, we realized that we would already have a unique integer to represent each machine. This also allowed us to save a great deal of space, and a bit of precious time, since the numbers used to identify the machines were massive.

Our next step was to remove features that consisted of more than 99 percent missing data, our justification being that any conclusions drawn from them would not be applicable to the population of machines as a whole. This resulted in two features being removed.

Our next targets were features that were heavily skewed, in this case meaning that more than 99 percent of the data in a feature was the same value. This was done for a similar reason as above: any conclusions drawn from them would likely not be applicable to the population as a whole. This resulted in 11 features being removed.

After these feature removals, for features that had over 10% of their values missing, we manually replaced the missing values with placeholder values of our own. Additionally, we merged values that were likely meant to be the same group (for example, under the feature category "SmartScreen", there were three "off" values: "Off", "off", and "OFF", which we grouped into a single "Off").

Our final removal technique was to find the correlation coefficient between each feature, and for pairs of features with a correlation coefficient of over 0.99, we removed the feature that had the smaller number of unique values. This was done based on the idea that we would have more data to work with if we kept the feature with more values. This resulted in four features being removed.

We split the target data from the main dataset, where it was initially located, so the true initial feature count was 82, which was reduced by a grand total of 18 features (roughly 22% of the original feature count, from 82 to 64).
Our final step in preprocessing was to change all categorical features into discrete features using integer values (e.g. for a feature with values "hot", "cold", and "warm", we would have reassigned all the values as 0, 1, and 2).

Our data was split 80/20 for training/testing respectively.

4. Methods

Initially we thought of training our algorithm with several other methods like KNN or SVM etc. However these algorithms (which compare each datum with every other) are not suitable for large dataset, and based on previous work done on the subject that we found, we chose the following methods: Decision Tree classifiers, Bagging (in relation to Decision Trees), Random Forest classifiers, and Boosting (in relation to Decision Trees). These methods splits data on a feature basis, which aids in reducing time consumption.

4.1 Decision Tree

4.1.1 Overview
The methods we used were all based on a decision tree algorithm, due to the rationale that this would run faster than other direct comparison methods, since we divide classifications on a feature basis instead of relative values to other data points.

4.1.2 Basic Algorithm for a Decision Tree

1. Find the feature that best splits the data among all of the available features (a choice for this decision could be the largest information gain among the features available). In our case, we used entropy to decide the criteria.

$$Entropy = \sum_{c \in class} -P(c) \log_2 P(c)$$

2. Based on the best split of the feature above, assign a dataset for each branch that is equal to the dataset in the previous step, except with the splitting feature removed.
3. Repeat this algorithm recursively for each branch until there are no features left in the dataset, or if pruning is in effect, ends if further splits would not be effective.

4.1.3 Advantages

- Decision Trees tend to have a low bias, since each model is created solely by the training data
- They will be more flexible than other more rigid methods (like some types of regression)
- They will capture many of the nuances in the data given that they are deep enough

4.1.4 Disadvantages

- Decision tree tends to have high a high variance, since each model is only created by the training data, and so attempts to perfectly fits the classifier to the training data
- Their flexibility means that no two sets of differing training data will get the same decision tree, so there is dubious consistency between various trees
- While nuances may be captured, Decision Trees also capture and act on all of the noise in the data

4.2 Bagging with Decision Trees

4.2.1 Overview

Bootstrap Aggregation (for short, Bagging) is the application of the bootstrap method to high-variance algorithm, Decision Tree.

Before delving into the Bagging algorithm, it is helpful to explain the term "Bootstrapping". Bootstrapping in its purest form is a resampling technique meant to estimate the sampling distribution of a parameter. It is executed by creating a random sample of the original data with replacement. In each random sample, some data points might be duplicated, and many will not be present, but overall provides a viable sample of the original data, useful for estimating sample statistic parameters.

4.2.2 Algorithm

The standard method of Bagging uses Bootstrapping repeatedly to obtain a set of samples, as detailed below:

1. Construct B bootstrap samples $\{S_1, S_2, .., S_B\}$.
2. Train a model on each sample S_b to get the prediction $\widehat{f}_b$
3. Compute the average of the predictions:

$$\widehat{f} = \frac{1}{B} \sum_{b=1}^{B} \widehat{f}_b$$

However, in the case of classification, step three above shifts from taking the average of the predictions of each model to returning the mode of the predictions, so that the most common prediction is used as $\widehat{f}$.

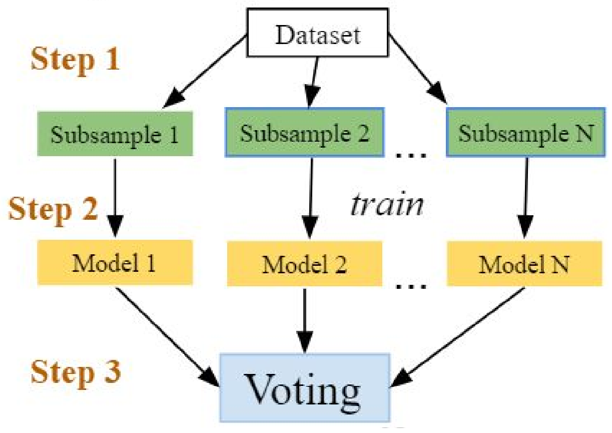


Fig 1: Bagging algorithm

4.3.3 Advantages

- Due to the random nature of each sample, Bagging in the context of a Decision Tree will mean that each tree is trained on a different dataset. One of the main issues of Decision Trees is their tendency to overfit the data they are modelling. Since Bagging uses the most common prediction based on a large number of different Decision Trees, we are able to dampen the normally strong overfitting results of a single tree, and ultimately reduce the high variance that is characteristic of Decision Trees.

4.3.4 Disadvantages

- As hinted at above, a major disadvantage of Bagging is that given a poor model, or a model with a strong bias, we will gain nothing useful from bagging, other than having one set of bad (or heavily biased) predictions be chosen over another. In our case, Decision Trees attempt perfectly classify the data they are given (at the very least are quite overfitted to the data), and have the nice perk of having a generally low bias in general, due to being created solely on the basis of a given dataset. However, a model such as a linear classifier would not work as well, since it'd be hit and miss situation, where the results are assuming that the values can be linearly split, when in many cases this is not possible.

4.3 Random Forest

4.3.1 Overview

A supervised learning algorithm, it is an ensemble of decision trees, trained with previously mentioned bagging method. However, it presents an improvement over bagging alone, by using a random subset of features to make a splitting decision for each level of the tree (as opposed to making a decision based on all of the features).

4.3.2 Algorithm

The algorithm for a random forest class is the same as that of a regular decision tree, with the the only change being that a random subset of features is available at a given level.

Can either be a standalone tree, or can be an additional detail of the bagging algorithm, where each tree uses a random subset of data, in which case the following steps apply to each tree in the ensemble.

a. A random subset of the available features is created from the current dataset with all features.
b. Find the feature that best splits the data among the features in the subset.
c. Based on the best split of the feature above, assign a dataset for each branch that is equal to the dataset in the previous step, except with the splitting feature removed.
d. Repeat this algorithm recursively for each branch until there are no features left in the dataset, or if pruning is in effect, ends if further splits would not be effective.

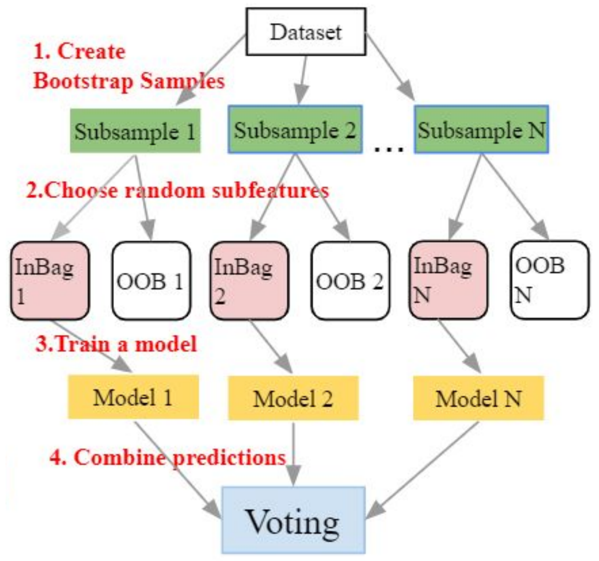


Fig 2: Random Forest Algorithm

4.3.3 Advantages

- The standalone version of a random forest classifier has the same advantages of a normal tree classifier, but with the added noise reduction of random feature selection
- The bagging version of a random forest classifier has the same advantages as the bagging classifier, since overfitting will still be dealt with, but additionally handles the disadvantage present in bagging of sensitivity to the base estimator. This is the case because each tree in the ensemble will not be based on the

same model of tree due to a different order of feature splitting from the random aspect of the random forest classifier.

- The random property of a random forest classifier allows a measure of relative feature importance, since the best features for splitting will be found towards the top, while also avoiding the same ordering issue that occurs in bagging.

4.3.4 Disadvantages

- Many of the disadvantages from its components are nulled, or at the very least are mitigated by the random forest classifier. Persisting major issues of the random forest classifier are due to time or data size constraints

4.4 AdaBoost (aka Adaptive Boosting)

4.4.1 Overview

Like Bagging, Boosting is an ensemble method (in our case, of Decision Trees). Bagging uses data sets produced by random sampling with replacement, and each data set is built independently. However, Boosting iteratively learns weak classifiers with respect to a distribution, then uses them to make up a final strong classifier.

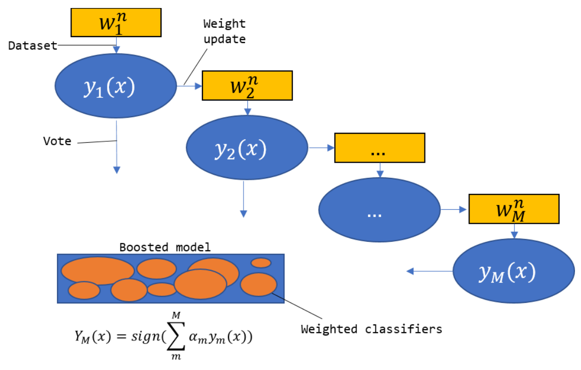


Fig 3 : Boosting updates weight of each data and feedback to next classifier

After a weak learner is added, the weights are adjusted. Each new weak learner places higher weights on misclassified data points. Different boosting algorithms have different methods of weighting training data and hypotheses.

4.4.2 AdaBoost Algorithm

Suppose we are given a set of training data $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$, where the input variables $X_i \in R^d$ (d is the number of features), and the binary target variables $y_i \in \{-1,\ 1\}$. [8]

1. Initialize the weights of data points. (AdaBoost constructs a series of weak classifiers $F(x)$).
   - Weight $w^{(0)}{}_i = \frac{1}{N},\ i = 1,\ 2,\ \ldots.,\ n.$ (equal weights for each sample)
   - Classifier $F_0(X) = -1.$

2. For $m = 1, 2,\ \ldots,\ M$:

   A. Fit a classifier $F_m(X)$ to the training data by minimizing the weighted error function where

   $$err^{(m)} = \sum_{i=1}^{N} w^{(m)}{}_i\, I(F_t\,(x_i) \neq y_i) \,/\, \sum_{i=1}^{n} w_i$$

   - $I(F_m\,(x_n) \neq y_n)$ is the indicator function and equals 1 when $F_m\,(x_n) \neq y_n$ and 0 otherwise
   - $err^{(m)}$ represents weighted measures of the error rates of each of the base classifiers on the dataset.

   B. Compute the weighting coefficients $\alpha^{(t)}$.

   $$\alpha^{(m)} = ln(\frac{1-err^{(m)}}{err^{(m)}})$$

   - $\alpha^{(m)}$ assigns a larger weight to classifiers with smaller error, a smaller weight to classifiers with larger error.

   C. Update the weight for each datapoint

   $$w_i = w_i \cdot exp(\alpha^{(m)} \cdot I(F_m\,(x_i) \neq y_i))$$

   for $i = 1, 2,\ \ldots,\ n.$

3. After M iterations, make the final prediction

$$H(x) = sign(\sum_{m=1}^{M} \alpha^{(t)} F_m\,(x))$$

4.4.3 Advantages

- Adaboost can do feature selection on very large sets of features, which results in a simpler classifier as we explain in section 4.5.

- The iterative nature of Adaboost allows a poor weak classifier to do well, since improvements will be made over the course of generating more weak classifiers
- The Adaboost algorithm is resistant to overfitting, in a general case, since once it has correctly classified the data, further weak classifiers increase the margins of the votes, resulting in improvement on test data accuracy far after training accuracy reaches 100%

4.4.4 Disadvantage

- While AdaBoost is resistant to overfitting as mentioned above, there is also a possibility to overfit if the weak classifiers are too complex, and themselves are overfitting the data.
- On the other side, AdaBoost is likely to underfit the data if weak classifiers are too simple. Thus, there is a somewhat delicate balancing act to get best performance; base classifiers should be chosen carefully.

4.5 Feature Reduction Methodologies

For solely convenience purposes, we used a fast classifier, Logistic Regression, to compare different methods of feature reduction (despite it having a lower accuracy rate than the other methods we used). Accuracy rate for Logistic regression with only the preliminary feature reduction is 0.52533 (based on 64 features).

We then compared the accuracy rate after reducing features with several methods.

Methods applied	Accuracy of Log Regression
Log Regression with no feature reduction (base)	0.52533
PCA best (maximum of 38, minimum of 10)	0.52849
Correlation with target (54 features)	0.50394
Adaboost (38 features)	0.55536

In theory, since PCA is dimensionality reduction, it should cause no change in the accuracy rate (nor improve the accuracy rate), since we would assume the features will be split optimally in our tree.[5] However there was actually a slight accuracy increase, potentially from weeding out features that were forcing the model(s) to stray from the correct classifications. We reduced the the feature count to various values between 10 and 39, with repeated values having the same resulting accuracy (though some did result in a lower accuracy). This method resulted in a maximum removal of 54 features, to a minimum of 25 features, and in a best-case accuracy rate increase of 0.00316.

The next method was checking the correlation between each feature and the target, and removing those features having zero correlation with the target, surmising that a feature with no correlation with the target will not ultimately be useful in attempting to classify machines on the basis of that feature. However, a correlation of zero only means that there is no linear relationship, and as such, this method ends up removing features that are actually helpful in estimating malware presence, and it ended up reducing the accuracy rate. This resulted in a removal of 10 features on this basis, and in an accuracy rate decrease of 0.02139.

Our final method was to use an implementation of AdaBoost to classify our data, and then to remove those features which were given a weight of zero upon classification. A feature with weight of zero means that the feature is not used as a node that splits data. When we use Adaboost with default parameters:
(base_estimator = DecisionTreeClassifier(
max_depth = 1),
n_estimators = 500,
Learning_rate = 1.0), we obtained 10 features that were assigned weights of zero. Depth is controlled because left unchecked, it would produce an overfitting of the training data, thus likely resulting in a lower accuracy rate on test data. The removal of these features results in an accuracy rate increase of 0.03003.

5. Results

5.1 Hyperparameter Tuning

The hyperparameters for each model were selected via Grid-Search and manual tuning in order to achieve the best classification performance.

First for Decision Tree, we improved the accuracy rate by limiting the tree depth. We choose tree depth by testing in a loop, comparing the error rate on train and test data. We choose data where accuracy score on test data is highest. Tree depth 10 proves to be optimal as the figure below.

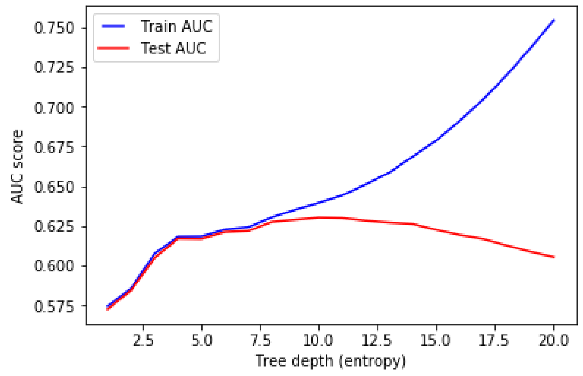


Fig 4: Tuning max tree depth in Decision Tree

For Bagging and AdaBoost, we use a Decision Tree that has maximum depth 10 as a base estimator.

As Figure 5 shows, in Random Forest and Bagging, a more accurate prediction requires larger number of trees. However, AdaBoost can get the best performance with smaller number of trees, which is 50 trees.

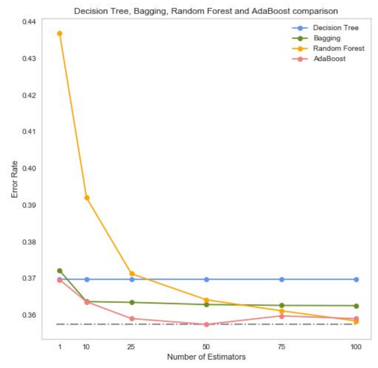


Fig 5: accuracy vs number of trees

Methods		
	max_depth	
Decision Tree	10	
	n_estimators	
Bagging	100	
	n_estimators	
Random Forest	100	
	n_estimators	learning_rate
AdaBoost	50	0.06

Table 1: Hyperparameters after tuning

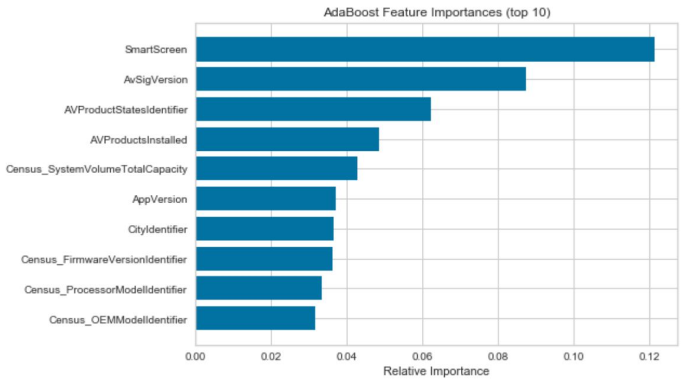
5.2 Model Performance

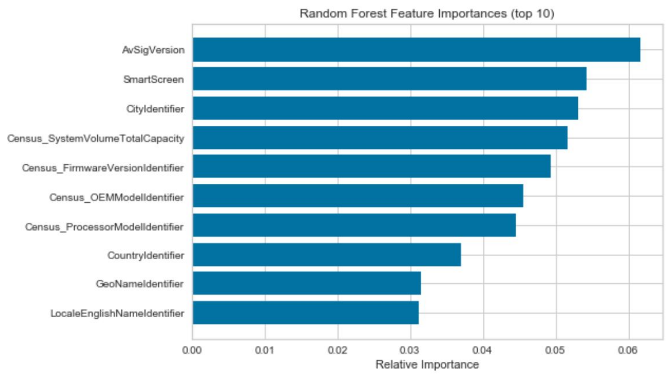
Model	Accuracy rate
Decision Tree	0.6303076620233671
Bagging	0.6374173527402416
Random Forest	0.6416090596449786
AdaBoost	0.6425469830131657

Our malware detection models were able to achieve accuracy rates (of correctly predicting malware presence) between 63 to 65 percent. We used an 80/20 train/test split on the dataset, and the model was trained using 64 features and 500,000 machines.

Also, we compared the feature importances of AdaBoost and Random Forest after tuning each of the hyperparameters. The table below shows the top 10 features in each algorithm.

Table 2: AdaBoost and Random Forest feature importance (Top 10)





Using only the top 10 features, as we did in Section 4.5, we used a Logistic Regression to compare the accuracy rate with the top 10 features of each method.

As we can see in Figure 6, Adaboost is much more useful for feature reduction purposes as compared to Random Forest.

Fig 6: Accuracy rate with the top 10 features of AdaBoost and Random Forest

Model	Accuracy
AdaBoost	0.5407880872152939
Random Forest	0.5131367168050394

Overall, AdaBoost performed the best in terms of speed and accuracy. Additionally, as a byproduct of using AdaBoost, we can reduce the number of features, which would result in a faster prediction for other models as well.

6. Conclusion and Future Work

Using different kinds of methods to reduce features and train and test, we concluded that AdaBoost method has the highest accuracy rate. We also were able to improve the speed and accuracy rate by deleting features according to the results of Boosting (removing those that had zero feature weight). We tried to reduce values further using PCA, or removing features that has zero correlation with target values, but none beyond the very first removal produced better results.

There are a number of tweaks and alternate routes this project could take in future work. Perhaps the most substantial addition to our project would be to attempt many of the same techniques, but instead of only using decision trees, we use an assortment of classifiers. This would add several new angles and potential branches to the work done here, and it may be possible that new implementations of the processes we used lead to better or at the very least different results, which even in the worst case of providing no accuracy or speed improvements, would filter out a group of classifiers that aren't useful for this application.

Another potential path for future work could lie in finding exact measures of what features would betray the presence of malware; obviously not all malware use the same exploits, nor are all potential exploits necessarily utilized, but further separate research into exactly what aspects of the machines in question allow them to be compromised (as opposed to the current project of which features are simply present) may provide valuable insight to further heighten success rates.

7. Contributions

Jungin focused on Random Forest and feature selection. Steven collected datasets, focused on Decision Tree and Bagging and compared model performances. Rikako

focused on Adaboost, tune hyperparameters for each model, compared model performances, and created the model diagram.
All three members contributed equally to data preprocessing, analysis, and writing this report and the poster. When preprocessing, we used an external code from Kaggle discussion forums [10].

8. Code

In our project, we worked everything on Jupyter Notebook. We used the following libraries: pandas, numpy, matplotlib, sklearn, scipy, and mpl_toolkits.

1. First, since the original datasets from Kaggle consisted of more than 10 million, for efficiency, we split them into smaller sized data sets. Each smaller sub-dataset had approximately 500,000 individual machine information.
2. Reading a text file, using pandas library.
3. Preprocessing (we explained in Section 3). We use scipy.stats to find the correlation coefficient between each feature, numpy to replace nan values, and sklearn.preprocessing.LabelEncoder to convert categorical values into numerical values [10].
4. Generating train and test datasets using sklearn.model_selection.train_test_split.
5. Building Decision Tree, Bagging, Random Forest, AdaBoost models: Import classifier functions from sklearn.tree and sklearn.ensemble library. Fit the train data then get accuray rate. Adjust parameter of a function. These parameters results maximization of accuracy. Using GridSearchCV from sklearn.model_selection to tune hyperparameters for each model.
6. To compare different methods of feature reduction, we used LogisticRegression from sklearn.linear_model.
7. To create a graph for comparing with the relationship between the accuracy for each model and the number of trees, we used matplotlib mpl_toolkits.axes_grid1 toolkit.For other graphs, we just used basic matplotlib libraries.

9. References

[1] J.Z. Kolter and M.A. Maloof, “Learning to detect and classify malicious executables in the wild,” Journal of Machine Learning Research, vol. 7, pp. 2721-2744, December 2006, special Issue on Machine Learning in Computer Security.
[2] M. Kedziora, P. Gawin, M. Szczepanik, and I. Jozwiak. "Android Malware Detection Using Machine Learning And Reverse Engineering." Computer Science & Information Technology (CS & IT), 2018. doi:10.5121/csit.2018.81709.
[3] D. Gavrilut, M. Cimpoesu, D. Anton, and L. Ciortuz, “Malware Detection Using Machine Learning,” Computer Science and Information Technology, pp 735-741, November 2009.
[4] A. Navlani, (2018, May 16). Random Forests Classifiers in Python. Retrieved March 8, 2019, from https://www.datacamp.com/community/tutorials/random-forests-classifier-python
[5] L. Merckel, (2017, June 30). Preliminary Investigation: PCA & Boosting [Digital image]. Retrieved March 11, 2019, from https://www.kaggle.com/merckel/preliminary-investigation-pca-boosting
[6] T.G. Dietterich, Machine Learning (2000) 40: 139. https://doi.org/10.1023/A:1007607513941
[7] 18: Bagging. (n.d.). Retrieved March 11, 2019, from http://www.cs.cornell.edu/courses/cs4780/2015fa/web/lecturenotes/lecturenote18.html
[8] C. M. Bishop, *Pattern Recognition and Machine Learning*. New York, NY: Springer New York, 2016.
Fig 3: Boosting_visual_approach [Digital image]. (n.d.). Retrieved March 8, 2019, from https://www.python-course.eu/images/Boosting_visual_approach.png
[9]Rankin, Bert. “Bureau Director’s Opening Remarks Data Security Hearing ...” *Lastline*, Lastline Inc., 5 Apr. 2018, www.ftc.gov/system/files/documents/public_statements/1448447/opening_remarks_of_andrew_smith_at_the_f

tcs_hearing_on_data_security_9th_session_in_the_com
missions.pdf.
[10]“Load the Totality of the Data.” *Kaggle*, © 2019
Kaggle Inc.,
www.kaggle.com/theoviel/load-the-totality-of-the-data.
[11]”Microsoft Malware Prediction.” *Kaggle*, © 2019
Kaggle Inc.,
https://www.kaggle.com/c/microsoft-malware-predictio
n/data