Hanoi University of Science and Technology
School of Information and Communication Technology

Report on Machine Learning project

Decision Tree and Random Forest for detecting computer malware

Students

Nguyen Thi Minh Chau

Supervisor: Dr. Nguyen Nhat Quang

June 25, 2022

Contents

Abstract

Nowadays, computer malware has become one of the undeniable problems of Computer Science. Its threats are big concern of digital world, and analyzing, as well as detecting malware in a responsive manner, is considered a difficult task when the number of malware and number of malware categories has been blooming recently. With the progress of Data Science, machine learning algorithms are applied to evaluate and determine malicious code, approaching to Artificial Intelligence for classifying malware and put an early alarm before these threats have the chance to execute.

In this project, we aimed at studying some simple machine learning algorithms for this purpose. These algorithms run based on properties of a binary file to decide if it is a malware or a benign. We also investigated on some hyperparameters that affect the accuracy of algorithms, and have some small discuss on the applications.

1 Introduction

One of the concerns of users with digital devices and connection with the Internet is malware. Malware, which is implemented for different purposes of attackers, is a computer program that can cause harm to the device. While some malware directly disrupt computers' workflow, some other lie passively and collect user data, as well as user behavior and send this information to the hackers' server, for preparing an attack. Regardless of type, malware has a negative impact on user and their devices, and is always under the haunt of computer scientists.

With the evolution of computer technologies, the war between computer malware and computer scientist has never stopped. Malware changes continuously to perform more harmful activities as attackers' desire, and to hide itself more effectively under the search of different antivirus (AV) tools. However, no matter how they are disguised, there are still some same architecture patterns for each type of malware, which is must-have for a malware to perform its tasks. They make up some significant properties of a specific malware type, and can be used to detect and classify malware. Typically, signature, which is a specific bytecode pattern, and entropy, are focused. While entropy represents the level of chaotics in a binary's code and is propotional to the chance of a file being malware, bytecode directly represents the structure of the malware that causes harm.

Different methodologies for analyzing files, entropy and detecting signatures are being updated and automated to lessen the amount of human work. When Data Science born, it introduced a valuable database and a wide range of algorithms for enhancing the quality of AV tools. Machine learning has being used to solve the very basic problem of malware protection, which is to detect if a file is malicious. This is a matter of (binary) classification, using given information of a file, and there has been multiple algorithms developed for this purpose. One of them is Decision Tree and Random Forest, which are easy to implement and also effective.

In this project, we focused on how these algorithms are implemented and performed on a simple malware dataset. We also investigated on the effect of some hyperparameters used for the two algorithms on their accuracy, and how should the model be applied into real situation. This report describes how we conducted the work, and is organized as bellow

- Section 2 introduces the basic theoretical background of Decision Tree and Random Forest, as well as other algorithms used in the project.
- Section 3 gives an implementation with explanation and discussion of the algorithm using TensorFlow library. We also discuss about the effectiveness of the models on real scenarios.
- Section 4 give a conclusion on the project.

2 Theoretical background

This section briefly discusses the theory of algorithms used in the project: Decision Tree using CART algorithm and Random Forest

2.1 Decision tree

Decision tree is a decision-support tool, using tree structure to represent all possibles and consequences. Decision tree usually consists of

- Internal nodes: represent attributes to be examined
- Leaf nodes: represent a classification
- Branches: Connections between internal nodes of higher level and lower level, or between internal nodes and leaf nodes. Each branch corresponds to a possible value of the attribute associated with that node

Decision tree can be made on both discrete, continuous or mixed dataset. For discrete values, each branch is a distinguished possible of currently examining attribute. For continuous values, each branch corresponds to a range of values. For dataset that mixes both discrete and continuous information, there will be mappings for continuous data to a set of discrete classes, and they will be treat as discrete values. These mappings functions are made based on specific problem.

A decision tree is usually constructed by exhaustively searching all the space of possibilities (given by the dataset). The order of each node is decided by some algorithms to make sure the tree is as simple as possible. Depending on certain problem, the algorithm will be chosen to fit specific type of data.

A decision is made with a decision tree by examining value of each attribute, from root node to leaf node. For a dataset, there are more than one decision tree can be built, and thus, the result for categorizing one sample on different trees might be different.

2.1.1 CART algorithm

CART (Classification and Regression Tree) is a type of decision tree that can use for both purposes of machine learning: classifying and regressing. CART works well on both discrete and continuous data. In this project, since the dataset consists of real values, CART is a suitable choice for building decision tree.

A CART tree is a binary tree, with the structure as normal decision trees. The process of building a CART tree consists of diving the input space using a greedy approach, called recursive binary splitting. In this procedure, all the values are lined up, different split points are tried and tested using a cost function. The split stops when the tree reach maximum depth, or the nodes contain a minimum number of samples, or the whole search space is examined.

2.1.1.1 Gini index

Similar to Information Gain in ID3 algorithm for decision tree with discrete values, Gini index is the cost function used for determine splits in the dataset. If Information Gain illustrates which feature gives the maximum information about a class, Gini index measures the probability of a particular variable being wrongly classified when it is randomly chosen.

The formula for Gini index for each child node is

$$Gini_{child\_node} = 1 - \sum_{i=1}^{n} (p_i)^2 \tag{1}$$

where p_i is the probability of an object being classified to a class C_i, n is total number of class. The Gini index for the attribute is the weighted sum of all children nodes

$$Gini_{attribute} = \sum_{j=1}^{k} Gini_{child\_node}(j) \times \frac{S_j}{|N|} \tag{2}$$

where k is the number of choice for the attribute, S_j is the number of occurrence of choice i, and $|N|$ is the total samples considered for the attribute.

The value of Gini index varies between 0 and 1. If an attribute has Gini index of 0, all samples put in that node are of one class, or there is only one class. If the opposite happens, all elements considered in that feature are randomly distributed across classes. If the Gini index of a node is 0.5, then all samplings are equally distributed over the categories.

When examining the candidate feature for the next split node, the one with smallest Gini index should be used, since it helps the tree grow terminate faster, thus save resources and reduce complexity

For example, given the stock price dataset in Table 1

Past trend	Open interest	Trading volume	Return
Positive	Low	High	Up
Negative	High	Low	Down
Positive	Low	High	Up
Positive	High	High	Up
Negative	Low	High	Down
Positive	Low	Low	Down
Negative	High	High	Down
Negative	High	Low	Down
Positive	Low	Low	Down
Positive	High	High	Up

Table 1: Impact of trends, open interest and trading volume on stock price

the Gini index of atribute *Past trend* is calculated as:

$$P_{past\_trend=Positive} = \frac{6}{10} \tag{3}$$

$$P_{past\_trend=Positive\&return=Up} = \frac{4}{6} \tag{4}$$

$$P_{past\_trend=Positive\&return=Down} = \frac{2}{6} \tag{5}$$

$$Gini_{past\_trend=Positive} = 1 - \left[\left(\frac{4}{6}\right)^2 + \left(\frac{2}{6}\right)^2\right] = \frac{9}{20} \tag{6}$$

$$\tag{7}$$

$$P_{past\_trend=False} = \frac{4}{10} \tag{8}$$

$$P_{past\_trend=False\&return=Up} = \frac{0}{4} \tag{9}$$

$$P_{past\_trend=False\&return=Down} = \frac{4}{4} \tag{10}$$

$$Gini_{past\_trend=Negative} = 1 - \left[\left(\frac{0}{4}\right)^2 + \left(\frac{4}{4}\right)^2\right] = 0 \tag{11}$$

$$\tag{12}$$

$$Gini_{past\_trend} = \frac{9}{20} \times \frac{6}{10} + 0 \times \frac{4}{10} = \frac{27}{100} = 0.27 \tag{13}$$

Similarly, we obtain the Gini index for all attributes as shown in Table 2

Feature	Gini index
Past trend	0.27
Open interest	0.47
Trading volume	0.34

Table 2: Gini index of features of the stock price dataset

It is obvious that *Past trend* has the lowest Gini index, which indicates that elements put in this category will be distributed in a more purely manner than others. Thus, this should be taken as the next child node for growing the decision tree.

For continuous values, Gini index can be calculated in some strategies:

- Examine all values of current attribute. For each value, split the set of samples into 2 regions: samples with values smaller than selected, and samples with values higher than selected. Then, calculate Gini index for that value.
- Sort all samples and divide them into multiple regions. For each region, select an appropriate value (mean, median, etc. depending on characteristic of dataset), then calculate Gini index for that value

2.1.1.2 Splitting dataset

After having chosen an appropriate split node, candidates for this node are divided into 2 groups, each group for one possibility. Then, the process repeats for the remaining attributes and samples until terminate conditions are met.

Theoretically, the tree growing progress stops when all examples are examined. However, this may cause the tree being overcomplicated and wasting computing resources. Usually, the tree stop growing further when reaching a maximum depth, or the new created node has a minimum number of candidates.

The class of the leaf node is determined as the most popular class of candidates put in that node.

2.1.1.3 Using CART tree

Once the tree has been grown, the model can be saved for future prediction.

Given a sample, the CART tree categorizes it by examining its attributes, starting from the root. The class of the sample is determined as the class of the leaf it reaches.

2.2 Random forest

One problem of using decision tree for classification is about the tree sensitiveness toward noise. If the dataset contains false values, the Gini index might be heavily affected, thus lead to a very different tree configuration. Therefore, using only one tree might not give a good generalization for the problem since it is prone to - even small - errors.

Another problem is about the tree's configuration. Even if the input dataset is different, the number of features taken to consideration is still the same. Since the algorithm for growing tree is preserved over the process, there will be high chance that the structure of the tree only differ slightly.

One solution for these concerns is to build multiple trees from different datasets to enhance robustness of the classification over errors. In addition, for each tree, only a subset of features is examined to increase the varriance of trees. The final prediction will be the final votes over the trees' results. The more trees built, the more generalized the final decision is.

Such method of building trees is called *Random Forest*. The term *Random* comes from the fact that the input dataset for each tree in the forest is randomly *sampled with repetition* from the original data, and the features for categorizing is also *randomly selected*. These random datasets is called *bootstraped dataset*, and the size of each set is exactly equal to the size of original dataset. The random features used for each tree is called *randomly selected features*, and the number of features of each feature set is

$$Feature\_set\_size = \sqrt{|N|} \tag{14}$$

The final decision is made based on votes over the results from the trees. Usually, the most frequent class is taken as the final classification. However, some trees will be built with less important features and create poor predictions on some aspects, while some others does the same on other aspects. We can put *weights* on these trees to indicate the importance of each results to the final conclusion. Combining or *aggregating* these results will help minimize the errors.

2.2.1 Extra Tree Classifier

Extra Tree Classifier is another form of Random Forest, which also consists of multiple trees and decision is made based on the major votes on all trees in the forest. The main differences between Extra Tree Classifier and Random Forest lie on:

- The sub-datasets are not created with sampling-with-repetition methodology. In other words, the records in these subsets are completely random without reoccurence of any samples.
- Splitting on nodes is randomized, without depending on Gini index, entropy or any metrics

Because of these characteristics, Extra Tree Classifier still keeps the diverse properties and can be built in a quick manner. However, it is less generalize and optimal than Random Forest since the node division is not at the best point.

Extra Tree Classifier is usually used for those scenarios which do not require high accuracy, and as an extra tool for feature selection and hyperparameter tuning.

2.3 Tuning hyperparameters: Hyperband Optimization

In order for the model to perform best, important parameters for building the model should be carefully selected. These parameters are called hyperparameters, and the select process is done in validation stage with different optimization method.

The easiest strategy for select a set of proper hyperparameters is to randomly select from a set of predefined parameters given, and combine them for *full training and evaluation on these combination*. However, this randomly selection is obviously not effective for the complex hyperparameter set with a wide range of selections.

For our project, we will use Hyperband Optimization to select proper set of hyperparameters. The logic of Hyperband algorithm is to randomly sampling all combination of parameters, then instead of *doing a full train and evaluation*, only perform the training stage for some epochs, then select the best candidate from these epochs for final evaluation.

2.4 Cross-validation: Hold-out

In order to examine the model, we need to put a test on it. The easiest way to do this is to perform a hold-out cross validation. The dataset is randomly sampling and divided into training set and test set. After using the training set for building the model, the test set is immediately used to perform testing.

3 Implementation

This section introduces how the aforementioned algorithms were built. We studied the naive implementation to understand how they works, and used the libraries for actual examination on dataset to boost up speed. Differences between the two implementation is also discussed to point out the pros of the libraries.

3.1 Naive implementation

3.1.1 CART tree

The pseudocode for CART is decribed as in Algorirthm 1 below.

The algorithm has some fundamental functions

Algorithm 1 CART Tree naive implementation

```
function GiniIndex(groups, classes)
    N ← total number of instances in groups
    gini ← 0
    for group in groups do
        score ← 0
        S ← total number of instances in group
        weight ← S/N
        for value in classes do
            calculate p_value
            score ←score + p_value^2
        gini ← gini + (1 - score) * weight
    return gini

function TestSplit(index, value, dataset)
    left ←List()
    right ←List()
    for row in dataset do
        if row[index] < value then
            Append row to left
        else
            Append row to right
    return left, right

function GetSplit(dataset)
    C ← set of all classes
    bestIdx ← ∞
    bestValue ← ∞
    bestScore ← ∞
    bestGroups ← None
    for attributeIdx in dataset do
        for row in dataset do
            groups ← TestSplit(attributeIdx, row[attributeIdx], dataset)
            gini ← GiniIndex(groups, C)
            if gini < bestScore then
                bestIdx ← attributeIndex
                bestValue ← row[attributeIndex]
                bestScore ← gini
                bestGroups ← groups
    return Node(bestIdx, bestValue, bestScore, bestGroups)
```

```
40: function ToTerminal(group)
41:     return  class of maximum occurrence in group
42:
43: function Split(node, maxDepth, minSize, currentDepth)
44:     left ← node['groups'].left
45:     right ← node['groups'].right
46:     Free(node['group'])
47:     if !left or !right then
48:         node['left'] ← ToTerminal(left + right)
49:         node['right'] ← ToTerminal(left + right)
50:         return
51:     else if currentDepth ≥ depth then
52:         node['left'] ← ToTerminal(left)
53:         node['right'] ← ToTerminal(right)
54:         return
55:     if size of node['left'] < minSize then
56:         node['left'] ← ToTerminal(left)
57:     else
58:         node['left'] ← GetSplit(left)
59:         Split(node['left'], maxDepth, minSize, depth + 1)
60:     if size of node['right'] < minSize then
61:         node['right'] ← ToTerminal(right)
62:     else
63:         node['right'] ← GetSplit(right)
64:         Split(node['right'], maxDepth, minSize, depth + 1)
65:
66: function BuildTree(dataset, maxDepth, minSize)
67:     root ← GetSplit(dataset)
68:     Split(root, maxDepth, minSize, 1)
69:     return root
70:
71: function Predict(root, sample)
72:     if sample[root['bestIdx']] ≤ root['bestValue'] then
73:         if root['left'] is a subtree then
74:             return Predict(root['left'], sample)
75:         else
76:             return root['left']
77:     else
78:         if root['right'] is a subtree then
79:             return Predict(root['right'], sample)
80:         else
81:             return root['right']
```

- *GiniIndex(groups, classes)*: This function receives a set of groups for an attribute and a set of classes, returns the Gini index for the current considering attribute
- *TestSplit(index, value, dataset)*: This function receives an index of the attribute in the attribute list, a value for splitting, and a group of samples. It splits the set based on actual value of the sample's attribute in comparision with Gini index. If the record has its attribute value smaller than Gini index, it will be put in the left portion and vice versa
- *GetSplit(dataset)*: This function receives a group of samples and returns a structure containing the index of attribute for splitting, the best value used for splitting and its Gini index, and 2 portions of splitted group. It exhaustively tries all features as splitting node, and for each feature, determines Gini index of candidate values. After each examination, it checks if the best Gini index is found and assigns information if appropriate.
- *ToTerminal(group)*: This function identifies the final class of a group of samples. Class with most occurences will be chosen
- *Split(node, maxDepth, minSize, currentDepth)*: This function tries to split the current node to 2 branches by recursively calling *Split* on the left and right subtrees. If it realizes that the tree has met the maximum depth or minimum number of samples per node, it terminates the growth of the branch and creates a leaf node.
- *BuildTree(dataset, maxDepth, minSize)*: This is the entry point for building a tree. This function receives the full dataset with terminate conditions of maximum tree depth and minimum size of the leaf nodes.
- *Predict(root, sample)*: This function is used to predict the category of *sample* given the root of the decision tree. It recursively checks for the attribute's feature of candidates to find the leaf node of the tree, which will be considered as the class of the sample.

The full implementation of this pseudocode in Python can be found in `/scratch/CARTTree.ipynb` file of the project.

3.1.2 Random forest

Random forest algorithm can be implemented in the same manner as CART Tree. The most significant difference lies in the *GetSplit* function. In this case, we only examine only one subset of features instead of all features in the dataset. Thus, we need a small modification to sampling a set of attributes for spliting, shown in Algorithm 2

Algorithm 2 GetSplit modification for Random Forest

```
function GetSplit(dataset, numOfFeatures)
    C ← set of all classes
    bestIdx ← ∞
    bestValue ← ∞
    bestScore ← ∞
    bestGroups ← None
    features ← List()
    while size of features ≤ numOfFeatures do
        Random a feature index
        if index ∉ features then
            Add index to features
    for index in features do
        for row in dataset do
            groups ← TestSplit(index, row[index], dataset)
            gini ← GiniIndex(groups, C)
            if gini < bestScore then
                bestIdx ← index
                bestValue ← row[index]
                bestScore ← gini
                bestGroups ← groups
    return Node(bestIdx, bestValue, bestScore, bestGroups)
```

Algorithm 3 Subsample function for Random Forest

```
function SubSample(dataset, ratio)
    samples ← List()
    numOfSamples ← Round((size of dataset) * ratio)
    while size of samples ≤ numOfSamples do
        Random a sample in dataSet
        Add sample to samples
    return samples
```

For easily building trees for Random forest, we conduct some additional helper functions

Algorithm 4 Random Forest building function

```
function RandomForest(dataset, maxDepth, minSize, numOfTrees)
    sampleSize ← 1
    featureSize ← Sqrt(total number of features)
    forest ← List()
    for idx in range numOfTrees do
        subsamples ← SubSample(dataset, sampleSize)
        tree ← BuildTree(subsample, maxDepth, minSize)
        Add tree to forest
    return forest
```

Algorithm 5 Bagging prediction

```
function BaggingPredict(forest, sample)
    predictions ← List()
    for treeRoot in forest do
        prediction ← Predict(treeRoot, sample)
        Add prediction to predictions
    return Class of majority in predictions
```

- *Subsample(dataset, ratio)*: This function performs sampling with repetition on the given dataset. Usually, $ratio = 1$, however, for some special usecase, we can adjust just ratio to get desired size of dataset.
- *RandomForest(dataset, maxDepth, minSize, numOfTrees)*: This function perform a loop to build $numberOfTrees$ trees and put them into a $forest$ for return.

For prediction, we use a method called *bagging*, which means to evaluate all predictions and take the votes over them. In this case, the class having more instance in the result lists is taken as final result. Details of the bagging prediction algorithm is shown in Algorithm 5.

Details of the implementation in Python is in `/scratch/RandomForest.ipynb` file of the project [1]

3.2 TensorFlow implementation

Examining the TensorFlow libraries for *CartModel* and *RandomForest*, the functions are divided carefully with optimization coming from the programming language, which makes the algorithms run faster. In addition, there are some also some notes about the algorithm implementation that bring the enhancements.

- For evaluating the value for splitting: instead of examining all values in search space, the algorithm performs sorting and divide the samples into intervals before evaluation. It also has a quick function to predict if the *presort* or *in-place* sorting while splitting samples is more efficient. Therefore, the process of finding splitting value is faster
- Pruning: The algorithm has a function for evaluating if prunning a node leads to better overall result. Prunning means turning a non-leaf node to a leaf node to better generalizing prediction and eliminate complexity. The score for prunning is based on *accuracy*

Furthermore, each model allows user to specify the number of *threads* used for training. Using multithreading to boost up an exhaustive task significantly improves runtime

The implementation of CART Tree and Random forest can be found at [6] [7] [8] [9]. The core tree builder, including splitter and prunner, lies in *decision_tree* folder, while other corresponding classes and interface for model implementation is in *cart* and *random_forest* folders

3.3 Problem statement

For the project, our aim is to build an effective decision tree and random forest for predicting if a sample file is a malware or benign in real scenarios. Also, we want to find the best hyperparameters for these models to give the most optimal predictions. In this current context, we focus on *maximum depth*, *minimum number of samples per leaf*, and `num_trees`for the forest. The dataset is a **.csv* file contains 138,047 records.

For application to an actual IDS[1] for a system with an event rate[2] of 12,000 events/s (average event rate for a Windows machine in non-peak time), analyze the effectiveness of the trained models.

3.4 Experiments, evaluation & discussion

3.4.1 Experiments

In this project, we built 2 sets of experiments for CART and Random Forest, along with data analyzing and cleaning before model training

1. Data analysis and cleaning

[1]Intrusion Dectection System, a system that monitor devices and give alerts when there are suspicious behavior or actor
[2]The number of events occurred in a time unit

2. Experiments on models (for each CART and Random Forest)
    - Train models on the whole dataset without feature selection
    - Train models on the whole dataset with feature selection and compare with the non-feature selection model.
    - Investigate hyperparameters impact on models
    - Tune hyperparameters for models

3.4.2 Evaluation

3.4.2.1 Evaluation metrics

Normally, accuracy metric $\frac{TP+TN}{P+N}$ is used to evaluate a machine learning model. However, for an imbalanced dataset, this can cause serious misinterpretation. We need to discover other metrics to better assess the algorithm.

For this project, in addition to accuracy, precision and recall, we introduce two definitions: *True positive rate* (TPR) and *False positive rate* (FPR)

For binary classification as of this project, a score called threshold T is usually calculated (in this case, it can be considered the Gini index). If a sample has a score $X > T$, then it will be categorized as positive and vice versa. If we define $f_1(x)$ as the probability density function of X when X is truly positive, and $f_0(x)$ as that of the counterpart case, then:

$$TPR(T) = \int_T^\infty f_1(x)dx = \frac{TP}{TP + FN} \tag{15}$$

$$FPR(T) = \int_T^\infty f_0(x)dx = \frac{FP}{FP + TN} \tag{16}$$

Equation 15 and 16 can be proven using the original definition of TP, TN, FP, TN.

True positive rate and False positive rate are important in malware classification. While True positive rate is exact Recall, which indicates the percentage of actual positive cases detected, False positive rate represent the proportion of negative cases that is falsely regconized as positive. While we want a high recall, we also want to minimze False positive rate as much as possbile. For an IDS, high False possitive rate means security analysts have to coop with an enormous traffic of alarms, but most of them are irrelevant. This makes the security team exhausted and the IDS less reliable.

These two components, TPR and FPR, creates *Receiver operating character curve* (ROC), which diagnoses the prediction ability of a binary classification model. This curve can be plotted with TPR as the y axis, FPR as the y axis, and T is the varying parameter. The diagonal line x=y indicates that the model has no predicting ability, while any curves lies above this line can be considered acceptable and vice versa. A perfect classifier gives a predict point at $(x, y) = (0, 1)$. The higher value of T, the more chance the curve leans toward point $(x, y) = (1, 0)$, which implies that the model performs worse.

While being good evaluating models that needs to consider false positive rate, ROC is not sensitive toward imbalanced dataset. In order to give a good illustration on the model's ability on skewed data, Precision-Recall curve is used. This curve shows how a binary classification model performs with a threshold T over a dataset that has one majority class. The x-axis of the plot is Recall score, and the y-axis of the plot is Precision score.

Although the curves are useful for demonstrating the model's ability, it composes of 2 values (for x and y), which makes it not suitable for comparison. To do this, a metric called Area Under Curve (AUC) is introduced. This is the score recorded by the area on the lower half of the plot, divided by the ROC or Precision-Recall curve. Since this is only a real value, AUC is more frequently used for determining which model performs better. AUC score ranges from $[0, 1]$ with the higher the score, the better the model.

In this project, we used Accuracy, Precision, Recall, F1 score, False positive rate, ROC, Precision-Recall Curve to better understand pros and cons of the model. In addition, we compared AUC of ROC and Precision-Recall Curve (will be called AUROC and AUPR later on) to understand the impact of different hyperparameters on models.

Finally, since the AUPR is better for evaluation in this scenarios, we used it to tune the set of hyperparameters for our CART and Random Forest.

3.4.2.2 Data analysis and cleaning

The input dataset contains 138047 records with 41323 benigns (approximately 30%) and 96724 malware (approximately 70%). The label for benign and malware are respectively 1 and 0 at the `legitimate` column. For each sample, except `Name` and `md5` fields, which are in string format, there are a total of 54 features recorded, represented as non-negative floating numbers.

By quick statistical analysis for each feature, there appears 3 attributes, `Machine`, `FileAlignment`, and `SizeOfOptionalHeader`, that have records distributed in clear groups, and their noise can easily be detected and removed. Other attributes have record scattered over the range, without being any special cluster.

Therefore, although there are minimal number of records at each point of value, they cannot be considered as noise or errors.

Thus, the dataset is imbalance and highly distributed.

3.4.2.3 Experiments on CART model

3.4.2.3.1 Train model on the whole dataset without feature selection

The experiment is conducted with the help of TensorFlow Decision Tree library. The snippet of code for experiment is at `/training/CART_Experiments_refined.ipynb`

In this experiment, we simply built a CART with the cleaned dataset splitted into 3 parts for training, validation and testing. The testing groups took half of the original dataset. The other half was divided into two equal groups, one for training and the other for validating. All features were considered for selection. Hold-out were used for the validation stage. The result is shown in Table 3

	Accuracy	Precision	Recall	F1
Training	0.9721	0.9914	0.9891	0.9891
Test	0.9729	0.9922	0.9902	0.9886

Table 3: Experiment result for training CART on the whole dataset, without feature selection

It can be seen that the accuracy of the trained model was 97.29% for testing phase, which is very high. The precision, recall and F1 score of the CART were also at high level: 99.22%, 99.02% and 98.86%, respectively. The training time took 0.1643s, the total training and compile time took 12.2417s.

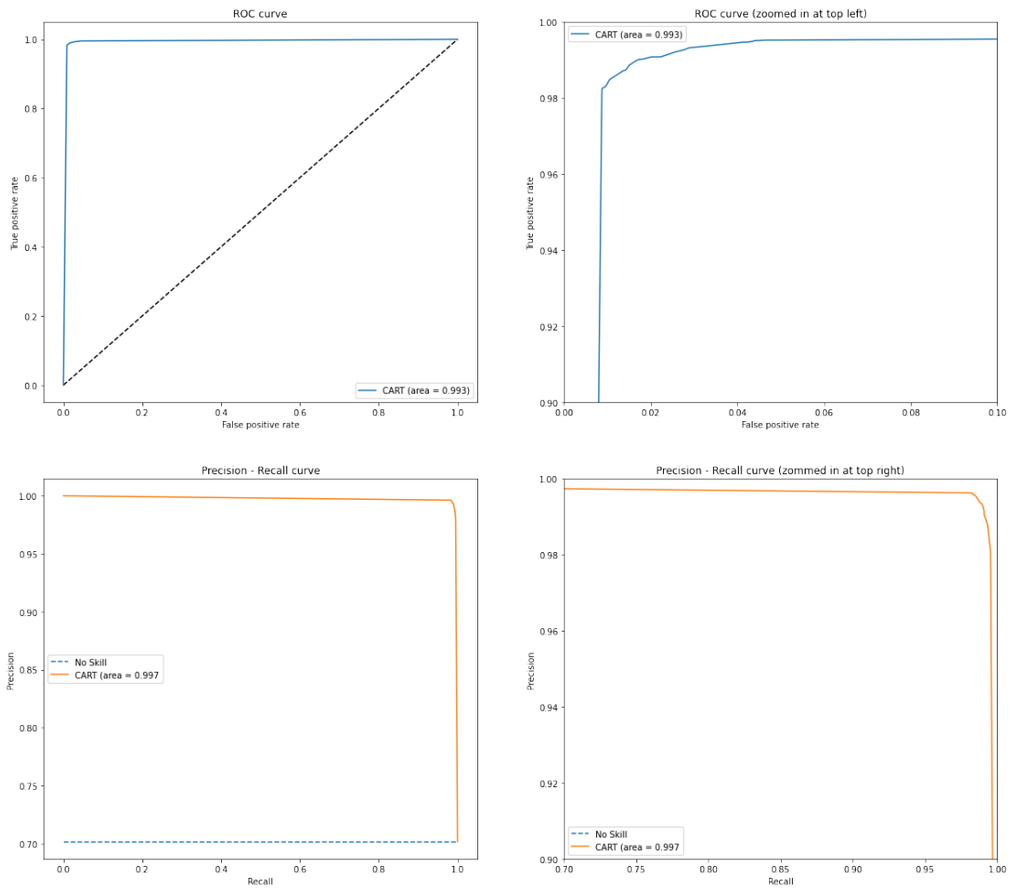

Figure 1: ROC and Precision-Recall curves

Figure 1 illustrates the ROC and Precision - Recall curve of the model. As we can see, the ROC curve leans toward point (0,1) and the Precision recall curve leans toward point (1,1), which indicates good binary classification ability of the model.

3.4.2.3.2 Train model on the whole dataset with feature selection and compare with non-feature selection model

We investigated the impact of feature selection onto the model.

Feature selection bases on two factors:

- Feature importance in the dataset
- Feature importance in the real problem

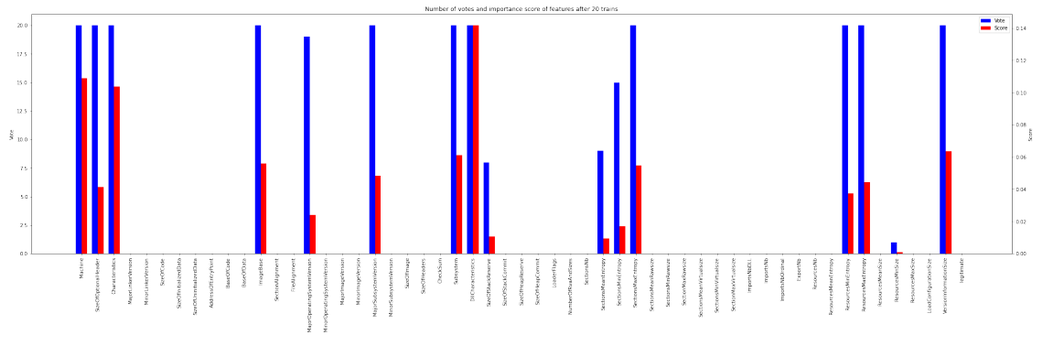

Figure 2: Votes and scores on features after 20 rounds

Feature importance in the dataset In order to determine feature importance in the given dataset, we can investigate a pre-trained model. This model should be generalized and can be quickly built so as not to consume time and resources. In this project, we use Extra Tree Classifier to build such a model. Scikit-learn library support building this model via `ExtraTreeClassifier` class

After having build an `ExtraTreeClassifier`, we start calculating the Gini index of each attribute over all extra trees. Those attributes with smaller Gini index will be considered as more important since it indicate purer features for splitting the nodes in the future.

The script for running `ExtraTreeClassifier` can be found at `/training/CART_Experiments_refined.ipynb`

Feature importance in the real problem Malware is basically the same as other normal executables, which includes *Optional headers* and *Section headers*, as well as other features. Optional header gives additional information about the state and form of the binary, and it is a real value within a predefined range. Optional header also shows insights about entry point (the address for `main()` function), image base, etc. for running the program. Section header illustrates location of raw data, as well as special characteristic of the binary, size of the binary, address of data in virtual memory space, etc.

Among all the features, there are two special features that need to taken to consideration, which is represented in the dataset

- *`Characteristic`*: a real value within a non-negative range, showing type of normal executable and its running strategy
- *`DllCharacteristic`*: a real value within a non-negative range, showing type of a Dll file[3] and its running strategy

These are all basic information of any file that can be extracted by popular malware analysis tools.

In addition, malware analysis tools also give other valuable information, which usually lies in the *metadata* section of the file. They also help in calculating *entropy* of the file. Entropy in malware analysis is a metrics to illustrate the disorder level of data in a file. This is an important factor to decide the potential of a file to be a malware. A normal file should have entropy level between 0 and 6 (on a scale of 10). If a file's entropy is higher than this level, there is a high chance that the file is a malware.

However, it is not enough to consider only the mean entropy of a file. There are very likely cases where a malware only have high entropy on some small portion of the file, and the overall entropy of the file is still on low level. On the contrary, there are also normal files that have high entropy on the majority of its body due to its complex nature. Therefore, we also have to take into consideration the minimum entropy and maximum entropy of a file to put it into a good category.

In this problem, the represented values that should be take into account are:

- *`SectionsMeanEntropy`*
- *`SectionsMinEntropy`*
- *`SectionsMaxEntropy`*
- *`ResourcesMeanEntropy`*
- *`ResourcesMinEntropy`*
- *`ResourcesMaxEntropy`*

The selection of features will include the consideration of both tool analysis and actual problem analysis

Feature importance selection We ran Extra Tree Classifier on the dataset for 20 times and got the votes, as well as average scores of each feature, as shown in Figure 2

As in the bar chart, there were 11 candidates with the same votes. Although the entrophies attributes had lower scores, they are important to the problem, as explained in the previous part. We took into

[3]Dynamic-link library, contains functions and variables to be loaded on use instead of on compilation

account these features and excluded *MajorSubsystemVersion*. In addition, we added some crucial properties as analyzed above to the set. In conclusion, the set of features for training were:

1. *DllCharacteristics*
2. *Characteristics*
3. *SizeOfOptionalHeader*
4. *Machine*
5. *SectionsMaxEntropy*
6. *Subsystem*
7. *ImageBase*
8. *VersionInformationSize*
9. *ResourcesMinEntropy*
10. *ResourcesMaxEntropy*
11. *MajorOperatingSystemVersion*
12. *SectionsMeanEntropy*
13. *SectionsMinEntropy*
14. *ResourcesMeanEntropy*

Training model with selected features and compare with non-feature selection model In this experiment, we splitted the dataset as in previous experiment, and conducted training both model on the same train, validate and test set. The script for this experiment can be found at `/training/CART_Experiments_refined.ipynb`.The result is shown in Table 4

	Accuracy	Precision	Recall	F1
Training	0.9708	0.9911	0.9899	0.9882
Test	0.9714	0.9912	0.9901	0.9884

Table 4: Experiment result for training CART on the whole dataset, with feature selection

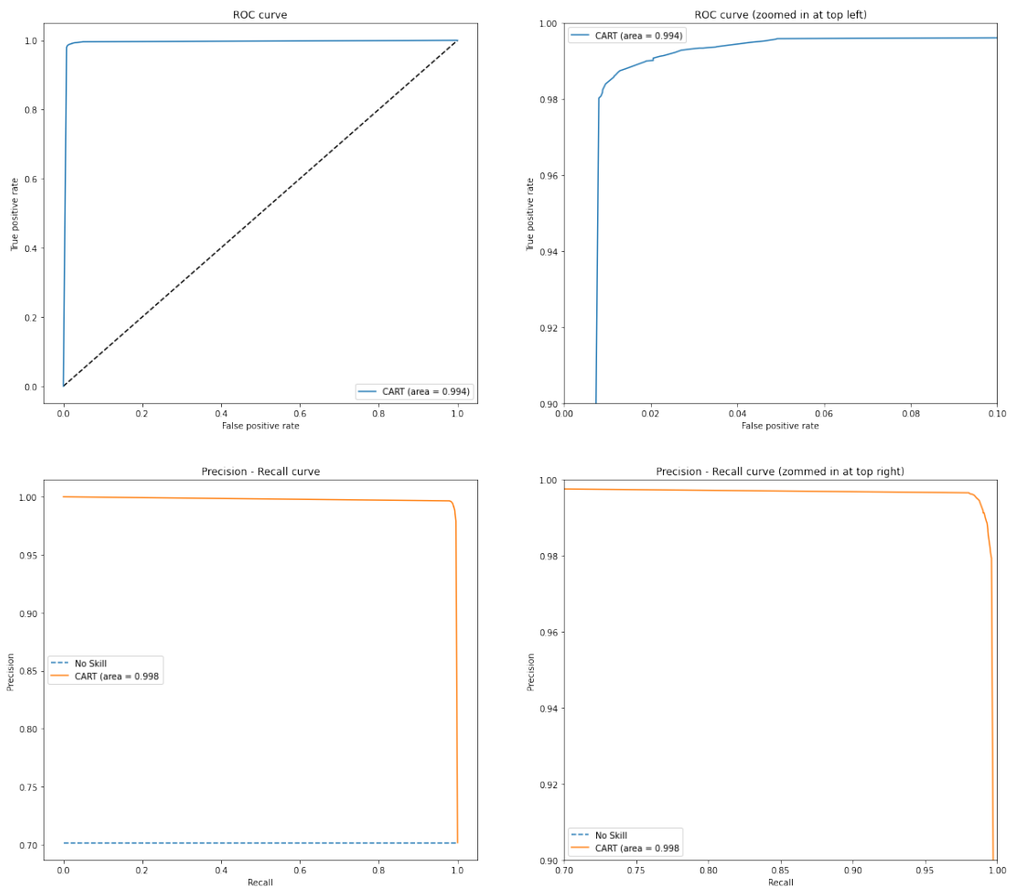

Figure 3: ROC and Precision - Recall with AUC for feature-selected CART

Its can be seen that, for all 4 metrics, the featured model had lower score than the full-featured model. However, the classification ability of the featured model was better, since it had higher AUC scores than the

former. This is due to the fact that the selection had been based on critical attributes of the samples and was not biased by the less important. The generalization of the model is, thus, better.

The training time of the model was 0.1030, and the total time for training and compiling model was 7.0347, which is significantly less than the full-featured version. Less attributes to consider leads to less training time.

3.4.2.3.3 Investigate impact of hyperparameters on model

For simplicity in comparison, we considered four scores: AUROC, AUPR, F1 and False positive rate

In this experiment, we investigated the impact of two hyperparameters, *maximum depth of the tree* (`max_depth`) and *minimum number of samples per leaf* (`min_examples`), to CART model.

In order to do this, we splitted the dataset as in previous sections and performed training models on these subsets. Each model would receive different parameters of `max_depth`or `min_examples`.

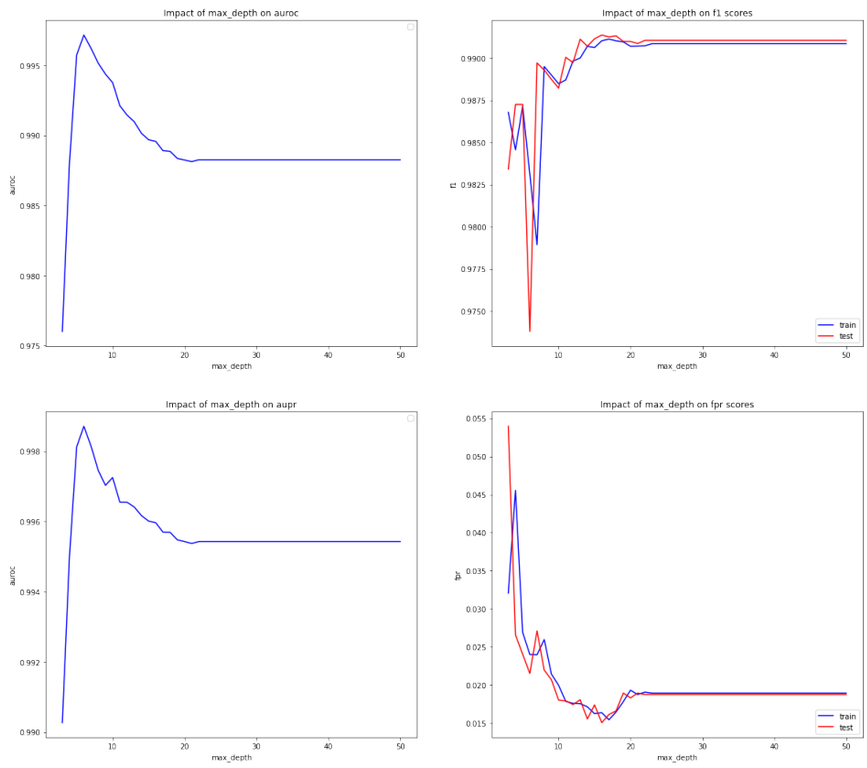

Figure 4: Impact of `max_depth` on prediction scores of CART

Figure 4 shows scores for 4 metrics when the depth of tree changed from 3 to 50, and `min_examples`hold at 5. It can be seen that for both AOC scores, the peak was reached when the depth of tree was around 5, then drastically dropped before leveling from the depth of 20. This indicates that the training progress should not make the tree grows more than 5 in depth. Further training will cost resources for nothing and even make the model perform worse.

The reason for increasing tree depth leads to worse performance of the model is because of the difficulty of the model when the algorithm goes further. The tree's depth affects pureness of nodes when the tree grows further. By default, the tree will be grown until each leaf has exactly one sample represent one class, or the minimum number of samples per leaf is reached, or the Gini index of that node reaches 0. Even though in this test, `min_examples`= 5, swallower tree will make the leaf has more candidates and thus, more generalized. In other words, a deep tree will lead to overfitting since it captures information about the sample in too details, which limit the ability of forecasting on new data.

Although F1 score of the model reached the highest point when the tree depth is around 20, the classification ability of the model decreased at the same point. This means that F1 score can give misinterpretation for evaluating the model in this situation.

False positive rate decreased significantly when the tree depth ranges from 3 to around 28, then slightly increased and leveled afterward. The more pureness are the leaves, the lower is false positive rate. However, this also drives the model to overfit, results in AUC scores to be lower.

Figure 5 illustrate the observed 4 metrics on `min_examples` hyperparameter. It can be seen that the 2 AUC score increased gradually when the minimum number of samples per tree increased. When the size of each leaves is larger, the generalization ability of the model is also increased. On the other hand, increased `min_examples` makes more normal sample falsely recognized as positive, thus made false positive rate proportionally increased. This drives the F1 scores to sunk, accordingly.

`min_examples` is a good hyperparameter for preventing the tree to overfit. It stops the growth of the tree in a proper manner to make the votes on each leaf more meaningful. `max_depth` limits the training faster,

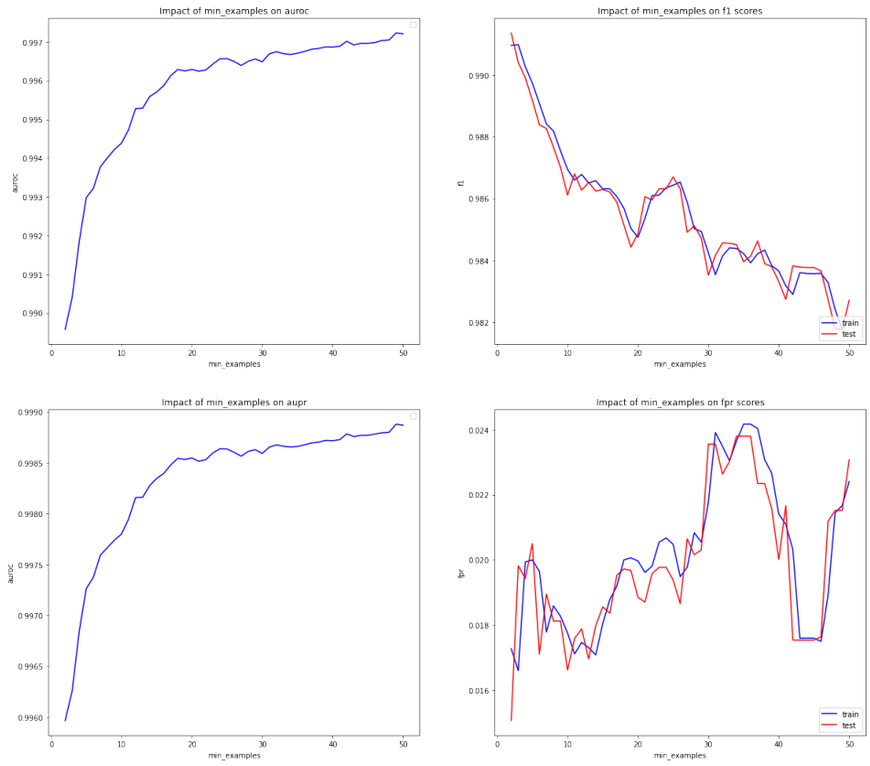

Figure 5: Impact of `min_examples` on prediction scores of CART

however, it may lead to too much samples per leaves and make the False positive rate higher.

3.4.2.3.4 Tuning hyperparameters for CART

In this experiment, we tried tuning hyperparameters for the featured CART model to get better prediction. Our objective was to maximize AUPR score of the model using Hyperband in order to get good classification over imbalanced dataset.

Details of the tuning script for CART Tree can be found in `/training/CART_Experiments_refined.ipynb`. The process of tuning CART Tree involved 500 iterations on one same training set, validation set and test set. The results are shown in Table 5

No.	max_depth	min_examples	Recall	FPR	AUPR
1	5	10	0.9911	0.0240	0.9973
2	6	36	0.9899	0.0241	0.9976
3	8	44	0.9904	0.0245	0.9986
4	7	24	0.9918	0.0268	0.9977
5	7	18	0.9916	0.0264	0.9976

Table 5: Tuning result for optimizing AUPR score of CART Tree

The results indicates that the optimal depth of tree should range from 5-8. The more samples per leaf, the better is the classification score of the model. We can choose the model with `max_depth`= 8 and `min_examples`= 44 to get the maximum categorization ability. With this selection, we reduced the complexity of the model and still obtain good AUPR score as of the original non-tuned model (`max_depth`= 15, `min_examples`= 5, AUPR = 0.998).

However, if the analyst are intense more on FPR, then we can also trade the classification score for this metric. Slight decrement for improvement in training time and resource savings is acceptable.

It's worth notice that the first set of `max_depth` and `min_examples` will give a better generalization and lower FPR. Thus, it is the best choice among the candidates.

3.4.2.4 Experiments on Random Forest model

We performed the same set of experiments as of CART and made comparision on prediction ability of the two models. The whole script for experiments can be found at `/training/RandomForest_Experiments_refined.ipynb`

3.4.2.4.1 Train model on the whole dataset without feature selection

Table 6 shows the result for training Random Forest on the whole dataset, without feature selection. The scores were taken by calculating the average score of 5 iterations.

Put in comparision with CART result, Random Forest gives better prediction scores for both training and testing phase by approximately 0.5%. This is due to its nature of better generalization ability compared

	Accuracy	Precision	Recall	F1	AUROC	AUPRC
Training	0.9779	0.9964	0.9828	0.9895	0.9932	0.9955
Test	0.9782	0.9968	0.9830	0.9899	0.9938	0.9960

Table 6: Experiment result for training Random Forest on the whole dataset, without feature selection

to CART. Using more tree to get more predictions, we will have a better look on the sample on different aspects, thus, have better conclusion about the sample's category.

The running time for training all 5 iterations is 78.2327s

3.4.2.4.2 Train model on the whole dataset with feature selection and compare with non-feature selection model

In this experiment, we used the selected features for training CART to make a featured random forest. This is appropriate since all tree in this forest will also be of type CART. Thus, there is no need of re-pretrain to select a new set of important features. However, this step should be done if the trees are of other types, or are a mix of different types.

Table 7 illustrates the results for training random forest with selected important features.

	Accuracy	Precision	Recall	F1	AUROC	AUPRC
Training	0.9790	0.9963	0.9830	0.9896	0.9927	0.9953
Test	0.9796	0.9966	0.9835	0.9900	0.9930	0.9956

Table 7: Comparing non-featured and featured CART models

Although the predicting ability of the featured Random Forest is still higher than that of CART, the metrics scores of Random Forest featured model were less than the its full-featured version because the variance is now less due to less features examined. However, the training time for 5 iterations was 75.4026s, lightly less than the non-featured version

3.4.2.4.3 Investigate impact of hyperparameters on model

We performed the same procedure for CART to understand impact of hyperparameters on Random Forest

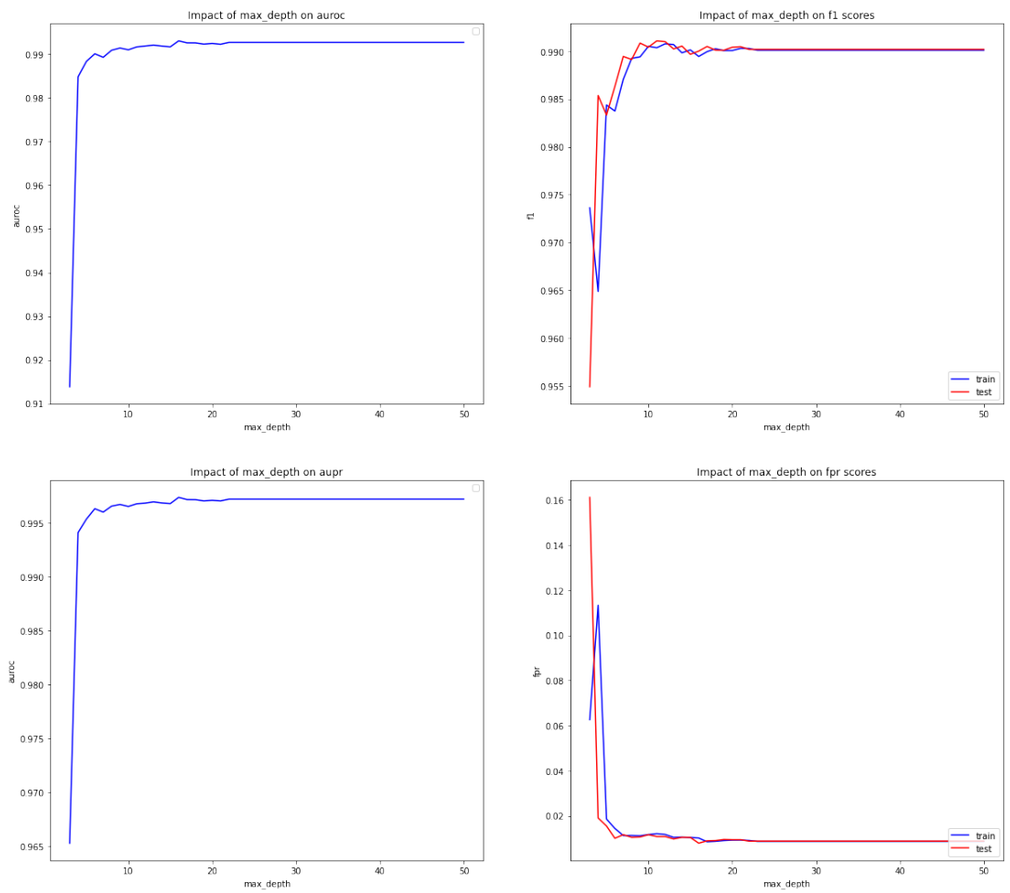

Figure 6: Impact of `max_depth` on prediction scores of Random Forest

Figure 6 illustrates the classification ability and some scores with `max_depth` ranging from 3 to 50, `min_examples` of 5 and `num_trees` of 300. It can be seen that the AUC scores both got to the highest level at the depth

of less than 20, before punctuating later on. However, there was no phase of sinking like the case of CART. This can be due of the good generalization ability of Random Forest compared to CART. The errors in one tree will be balanced by accuracy of other trees. This might also be the reason for the closeness of training and testing F1 and FPR scores.

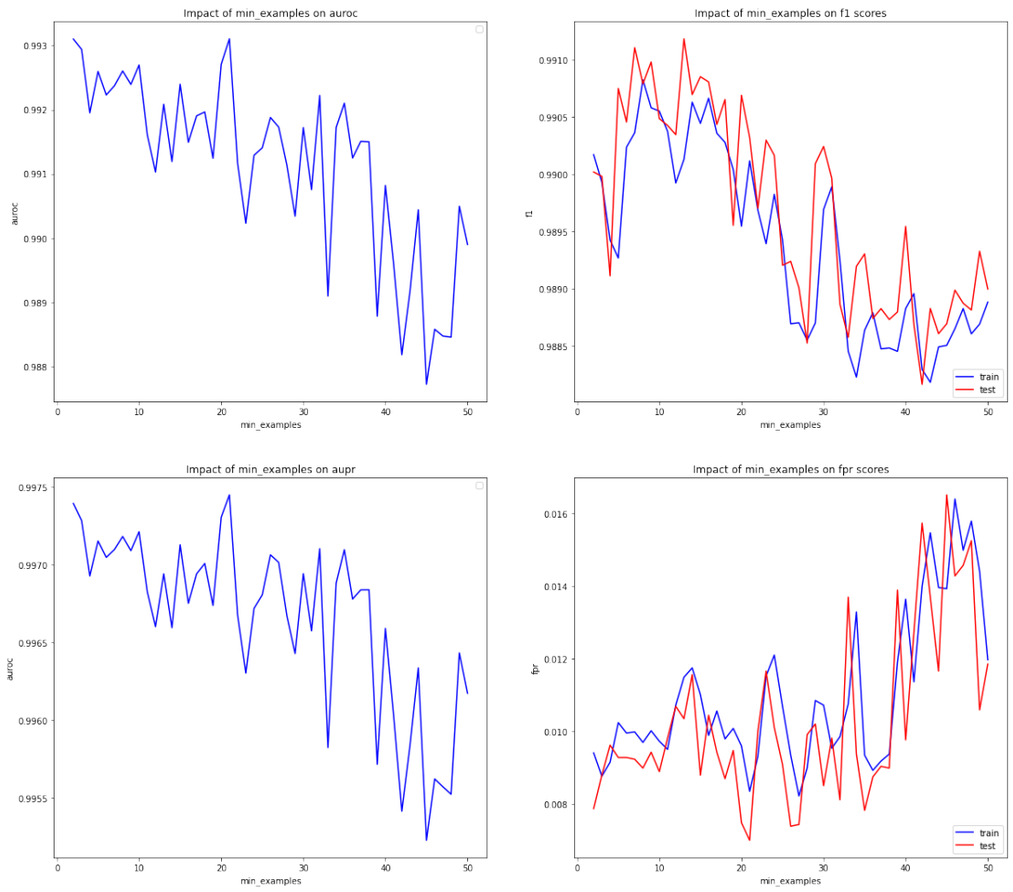

Figure 7: Impact of `min_examples` on prediction scores of Random Forest

Figure 7 shows the prediction scores when `min_examples`varied from 2 to 50, `max_depth` of 16 and `num_trees`is 300. Although having gone under drastic changes, it is obvious that the AUC scores and F1 score had a tendency of going down, while FPR went up. With a large number of trees, letting more samples per leaf also means that the tree grows swallower. This makes the classification less exact.

Figure 8 demonstrates the effect of `num_trees`, while `max_depth`and `min_examples`hold at 16 and 5, respectively. It can be seen that increasing the number of trees had slowlier impact on the model than leveling up the depth of trees. However, while AUC metrics went up, which is a good sign, F1 score dramatically hovered and tended to go down, while FPR also experienced great changes to increase.

For a balanced dataset, more number of trees in the forest will increase F1 and FPR to a certain scores before leveling. The effect observed in this experiment might due to the bias of imbalanced dataset on random forest. When growing the forest with sampling with repetion on a skewed input, there will be high chances that there are more trees that works with subsets with classification largely different from the original dataset, which leads to varying score.

This is another proof for the state that F1 score alone is not enough to assess a tree model on imbalanced data.

3.4.2.4.4 Tuning hyperparameters for Random Forest

In this experiment, we tried to select the set of `max_depth`, `min_examples` and `num_trees` so that our forest gives a good classification ability, measured by AUPRC.

No.	max_depth	min_examples	num_trees	Recall	FPR	AUPR
1	4	40	34	0.9891	0.0234	0.9986
2	8	21	83	0.9905	0.0195	0.9989
3	9	40	69	0.9909	0.0179	0.9989
4	6	33	76	0.9911	0.0182	0.9990
5	5	43	62	0.9907	0.0188	0.9989

Table 8: Tuning result for optimizing AUPR score of Random Forest

It can be seen from the result set, again, the depth of tree should not range over 10, while the more samples per leaf, the more generalization is the model. For the number of tree, higher number of trees does not help in making the forest having better classification ability. For example, even 83 trees in the second forest did

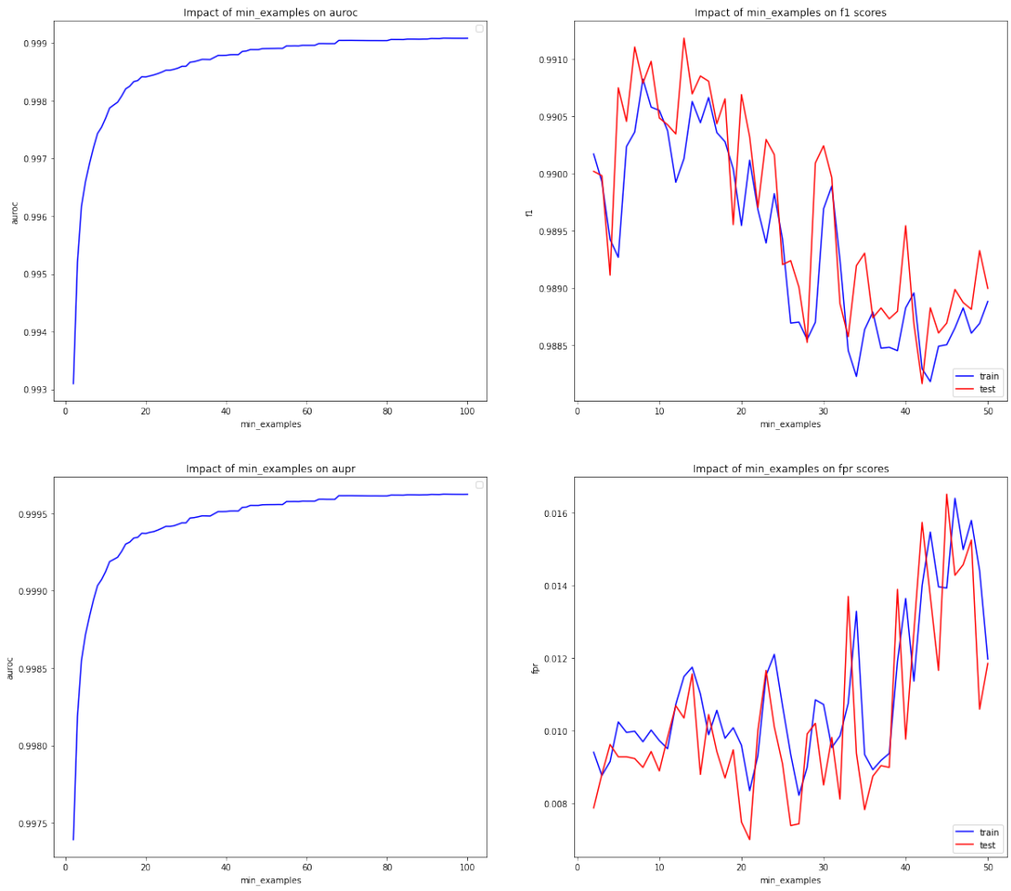

Figure 8: Impact of `num_trees` on prediction scores of Random Forest

not make AUPR score better than the 1st candidates, which only used 34 trees, but still gave an approximate score.

If the analyst focus more on the simplicity, the 1st hyperparameters set will be the best choice. On the other hand, if classification ability takes more weights, then the last tuple introduces a good model. We evaluate the 5th candidate more than the 4th since the latter makes the model more complex, but gives only a slight improvement in categorizing ability.

3.4.3 Discuss

3.4.3.1 CART and Random Forest for actual problem

From the experiment results, we can see that CART and Random Forest are algorithms that perform well on binary classification, for specificity, detect malware and benign as of this project. These two algorithms are easy to explain, as well as training and tuning.

However, such metrics obtained on the model might not enough for building a real IDS. In reality, the *malicious over legitimate traffic ratio* should be taken care of. This can be different from the distribution in the dataset. We can understand that the ratio of malware over benign in the example as ratio of total kinds of malware and benign that can be met in the environment. The malicious over legitimate ratio considers that ratio over a period of time, and this is usually much less than the distribution in the dataset.

For example, for each 100,000 events comes to the system, there is 1 malicious event. Then, every day, the system meets 1,036,800,000 events (12,000 events/s), 10368 of which are positive cases.

We define the false negative rate FNR as the rate that a sample is wrongly indicated as normal, and this is by example equal to $1 - Recall$.

Suppose that we choose the CART tree with `max_depth`= 5 and `min_examples`= 10. Then $FNR = 1 - 0.9911 = 0.0089$. For each day

- Number of positive cases: $P = 1,036,800,000 \times \frac{1}{100,000} = 10,368$
- Number of negative cases: $N = 1,036,800,000 - P = 1,036,789,632$
- Number of positive cases that are falsely rejected: $P \times FNR \approx 93$ cases
- Number of positive cases that are correctly catched: $10,368 - 93 = 10,275$ cases
- Number of negative cases that are falsely catched: $N \times FPR \approx 24,882,952$ cases

Therefore, total number of alerts that analysts receive each day is $24,882,952 + 0,275 = 24883045$. The proportion of useful alarm is $\frac{10,275}{24,882,952} \approx 0.0004 = 0.04\%$

This indicates a poor IDS system.

We can use the above analysis for Random Forest model with `max_depth`=5, `min_examples`=43, and `num_trees`=62 to obtain the figure of approximately 0.05%, which is higher than CART, but still not a good number.

There are ways to tackle this problems

- Investigate deeply the environment: get actual traffic rate, malicious traffic and normal traffic rate, as well as other parameters for training the model.
- Adapt a learning algorithm to the decision tree, that can automatically tune and prune the tree to adjust accordingly to real situation

3.4.3.2 Problems when training the model

The models were implemented using TensorFlow and Scikit-learn libraries. Therefore, there were little problem occurred when building and training. However, the training result varied over times, which made the scores unpredictable. This is due to the fact that the libraries do not set a *random seed*. Therefore, each run created a completely new train set, validation set and test set that affect the overall results.

We resolved this problem by setting a random seed globally before starting training. The metrics then became much more stable and understandable.

The most difficult part is evaluating and examining the impact of hyperparameters on the model. Since the problem introduced an imbalanced dataset, normal metric (in this case, accuracy) was not suitable for scoring models and investigating effects of different hyperparameters. Having not understand fully the evaluating metrics drives the project stuck on unwanted and unexplainable score results.

After having deeply investigated the impact of imbalanced dataset on models, as well as explore different metrics for evaluating a classifcation model, we had come up with curves, AUC, as well as other metrics to better assessing the algorithms.

4 Conclusion

Decision tree and Random forest are two of the strategies for building machine learning model to detect malware. Although decision tree is easier to implement, Random forest gives a better generalization for future prediction. However, the weakness of these two models, especially the latter, is sensitivity toward imbalanced data. In order to have a good evaluation on the models, we need to investigate the characteristic of the dataset, as well as metrics that are suitable for assessment.

The models trained in this project used an imbalanced dataset with minor errors. However, its impact on the model is still high. Besides data cleaning, feature selection also gives a better prediction since the model is then not prone to unimportant information.

Although accuracy is a popular metrics for evaluating a model, actual problem requires more investigation, as well as other proper metrics. In this project, we analyzed ROC, Precision-recall curves, AUC scores, in addition with False positive rate, put in comparision with F1 scores to see the differences. The curves and AUC scores do not only gives a better visualization on how the models performs, but also a more stable and explainable results for impacts of hyperparameters to classification ability of models.

Tuning is an important process to find good hyperparameters for the models to behave nicely toward future data. For CART and Random forest, we should limit the depth of the trees at around of less than 10. The larger minimum number of samples per tree, the better the prediction results since it better generalize the terminals for unseen samples. For the number of tree in Random Forest, it does not need to have so many trees to get good generalization.

However, such training might not enough for the model to be applied in real scenarios. We had analyzed other factor that exists in actual environment that current model have not had any ability to tackle with, resulted in poor performance of IDS or similar system adopting the model. Solutions involve deeply investigating the situation that the model will be working in, and introduce a learning mechanism so that the tree can transform itself to adapt to actual cases.

In this project, we have investigated the implementation of each machine learning model, as well as Hyperband as a strategy for tuning models. Although there are still problems to resolve, this has set a good point for understanding how Decision tree and Random forest works, as well as steps to conduct a machine learning model using TensorFlow toolkit.

References

[1] *Project's GitHub repository*, https://github.com/ntmchau2202/malware-detection-ml

[2] *Lectures note*

[3] *Tutorial on building Decision Tree from scratch*, https://machinelearningmastery.com/implement-decision-tree-algorithm-scratch-python/

[4] *Tutorial on building RandomForest from scratch*, https://machinelearningmastery.com/implement-random-forest-scratch-python/

[5] *TensorFlow guidance on APIs*, https://www.tensorflow.org/

[6] *Core implementation for Tensorflow Decision Trees*, https://github.com/google/yggdrasil-decision-forests/

[7] *TensorFlow CART Model*, https://github.com/tensorflow/decision-forests/blob/4c2b85f83e9af1ed80478115c49e273682da3cc5/tensorflow_decision_forests/keras/wrappers_pre_generated.py#L41

[8] *TensorFlow RandomForest Model*, https://github.com/tensorflow/decision-forests/blob/4c2b85f83e9af1ed80478115c49e273682da3cc5/tensorflow_decision_forests/keras/wrappers_pre_generated.py#L1624

[9] *TensorFlow Core Tree Model*, https://github.com/tensorflow/decision-forests/blob/4c2b85f83e9af1ed80478115c49e273682da3cc5/tensorflow_decision_forests/keras/core.py#L319

[10] *An empirical study on hyperparameter tuning of decision trees*, Rafael Gomes Mantovani, Tomáš Horváth, Ricardo Cerri, Sylvio Barbon Junior, Joaquin Vanschoren, André Carlos Ponce de Leon Ferreira de Carvalho, https://arxiv.org/abs/1812.02207

[11] *When accuracy isn't enough, use Precision and Recall to evaluate your classification model*, https://builtin.com/data-science/precision-and-recall

[12] *What metrics should be used for evaluating a model on an imbalanced dataset?*, https://towardsdatascience.com/what-metrics-should-we-use-on-imbalanced-data-set-precision-recall-roc-e2e79252aeba

[13] *Simple guide on how to generate ROC plot for Keras classifier*, https://www.dlology.com/blog/simple-guide-on-how-to-generate-roc-plot-for-keras-classifier/

[14] *How To Use ROC Curves and Precision-Recall Curves for Classification in Python*, https://machinelearningmastery.com/roc-curves-and-precision-recall-curves-for-classification-in-python/

[15] *Why do I get different results each time in Machine Learning?*, https://machinelearningmastery.com/different-results-each-time-in-machine-learning/

[16] *An Emphasis on the Minimization of False Negatives/Positives in Binary Classification*, https://medium0.com/@Sanskriti.Singh/an-emphasis-on-the-minimization-of-false-negatives-false-positives-in-binary-classification-9c22f3f9f73

[17] *Extra Tree Classifier for Feature Selection*, https://www.geeksforgeeks.org/ml-extra-tree-classifier-for-feature-selection/

[18] *How does ExtraTreeClassifier reduce the risk of overfitting?*, https://medium0.com/@namanbhandari/extratreesclassifier-8e7fc0502c7

[19] *Surviving in a Random Forest with Imbalanced Dataset*, https://medium0.com/sfu-cspmp/surviving-in-a-random-forest-with-imbalanced-datasets-b98b963d52eb

[20] *How to tune a Decision Tree?*, https://towardsdatascience.com/how-to-tune-a-decision-tree-f03721801680