Detect Malware on Android using Classification Methods

Paul-Niklas Kandora , Zhixuan Sun, Dahe Pan

December 20, 2022

Abstract

With the popularity of mobile phones, detecting malware when downloading software applications is getting more and more attention. To evaluate different machine learning classification methods for distinguishing between malware and benign Android applications, this paper refers to the paper 'Malware Dataset Generation and Evaluation'. After reproducing five classifiers mentioned in the original paper and provide deep analysis and comparison of those methods, we recommend two additional methods that have not been considered in the paper. This paper focuses on the 'TUANDROMD' dataset (android-based malware detection) while in 'Malware Dataset Generation and Evaluation', the 'TUMALWD' (web-based malware detection) is also considered.

1 Introduction

The Internet is the global system, which connects computers and networks and enables communication among devices. With the rapid development of the Internet over the past thirty years, it has become a vital part of our daily life. Additionally, the spread of the Internet results in the growth of technology across the world. Nowadays, people can use a variety of devices, such as phones, pads, and laptops to connect to the Internet. Among all these devices, phones are definitely the most convenient and popular devices. Furthermore, according to Statcounter, the Android operating system gains the largest global market of mobile phones, which is 72.2%. After obtaining a new phone, thousands of mobile applications can be downloaded to provide specific functionalities, including calculating, searching, gaming and etc. Unfortunately, when downloading applications, phones might be attacked by viruses. Therefore, this paper aims to detect malware on Android using several classification methods and also evaluate the accuracy and efficiency.

1. Malware and its influence

   Malware is an abbreviation for "malicious software". It refers to code or files which can cause harmful effects on phone systems, including stealing sensitive data, disrupting operations, and damaging or even destroying phones or phone systems and it is commonly developed by hackers.

2. Comparison with the original paper

   The original paper is *Malware Dataset Generation and Evaluation*. It focuses on collecting data and applying five classifiers on two platforms datasets: Windows and Andriod systems. However, because of the limitation of access, we can only download the TUANDROMD (Tezpur University Android Malware Dataset) dataset, from the UCI machine learning repository, which is for Andriod systems. Moreover, this is a project of implementing machine learning methods on a dataset instead of presenting methods of gathering data. Thus this paper emphasizes reproducing five classifiers same as the original paper, including random forest, extra tree, Ada boost, Xg Boost, and gradient boosting, and also trying some new methods.

3. Our contribution First, we implement the methods proposed in the We try two methods that are not mentioned in the original paper: feedforward neural network and Support Vector Machine (SVM).

   After presenting the main idea and applying them to the dataset, we analyze the accuracy, precision, recall, and F1-score for these two methods. Additionally, for every method in our paper, we discuss the CPU time which is essential to identify efficiency and classification errors. Cross-validation error versus training error is also considered.

The rest of the paper is organized as follows. Section 2 analysis the dataset used for machine learning training. In Section 3, we implement the same methods in the original paper and compare results. Moreover, we investigate CPU execution time and errors. After that, Section 4 presents the results of our methods and also an evaluation of CPU time and errors. Finally, in Section 5, the conclusion is given.

2 Data set and paper

According to different labels, we split data into two catalogs which '1' refers to MALWARE and '0' refers to GOODWARE. In addition, the dataset includes about 4465 observations and one row containing only nan values (we dropped this row). In addition, the dataset includes 241 features, which are divided into Permission based features (like RECEIVE_SMS) that need the permission by the user to be performed on the mobile device and API based features (like Ljava/net/URL;openConnection) that are performed through API calls in the application. It should also be mentioned that the features are exclusively binary and state for each observation whether this feature was present in a malware/goodware case. In the following we will look at how our label and record is balanced and whether it may need techniques to balance it. In graph 1 we see the number of goodware and malware observations in the label:

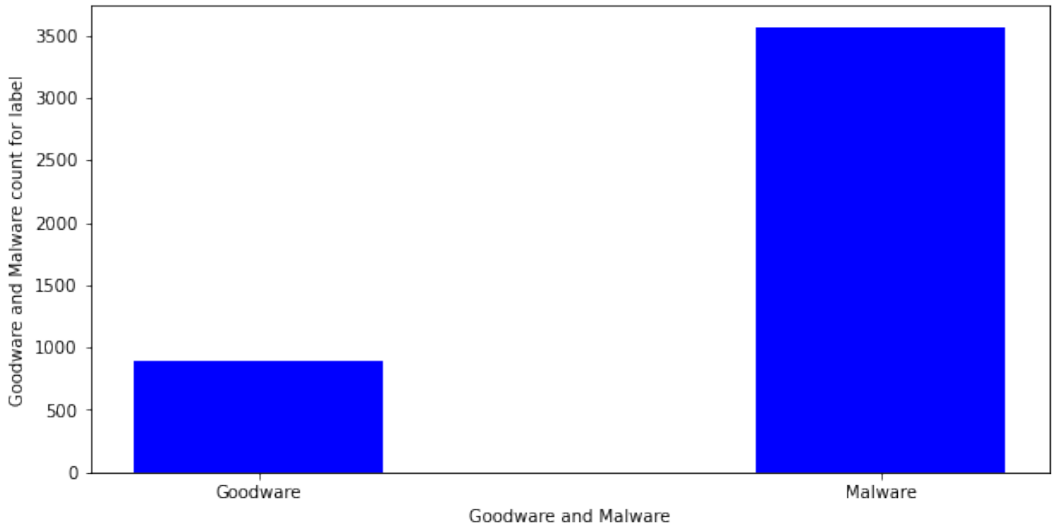

Figure 1: Goodware vs Malware in label

Here we see that there are 899 goodware and 3565 malware observations in the label, indicating that the label is unbalanced.

In the following we will look at how the distribution of goodware/malware is in the features. To do this, we calculate the ratio of malware observations / goodware observations for each feature and calculate the number of features that have a ratio higher than 1 (more malware features) and lower than 1 (more goodware features). From figure 2 we can see there are 232 features with more negative than positive responses while only 9 features are have more positive than negative responses, so it is necessary to oversample the positive samples. We use synthetic oversampling to finish this process which generates new samples in the minority class by interpolating between existing samples.

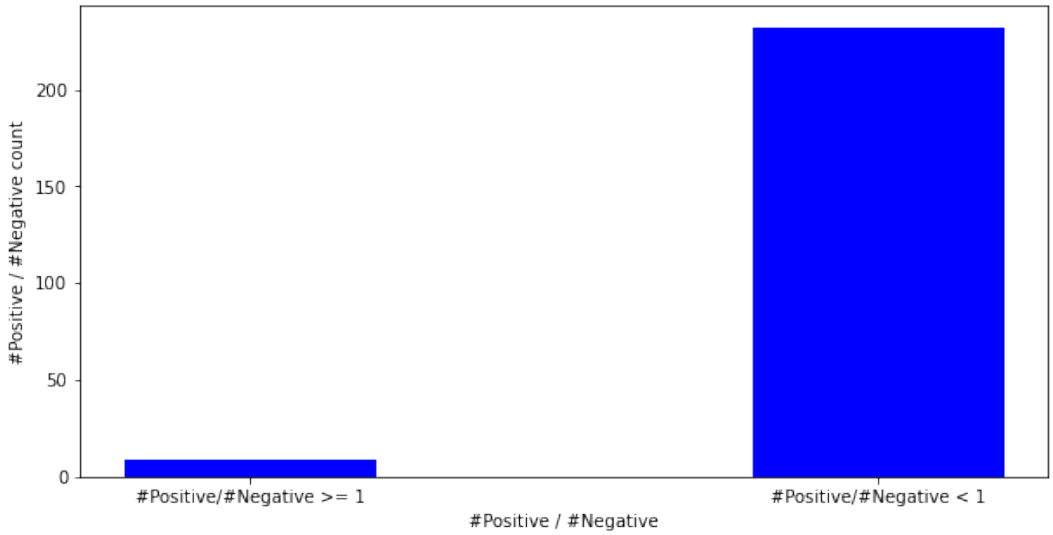

Figure 2: Positive/Negative Feature Ratio

The question now is how to evaluate the importance of the feature. In the paper 'Malware Dataset Generation and Evaluation' a ranking of the features is presented. However, it is not known how this ranking is generated. In table 1 we see the feature ranking from the original article:
Since we have a dataset that contains only binary features and labels, we can perform a Chi-Squared

Table 1: Features ranked by importance from paper

Feature Rank	Feature
1	SEND_SMS
2	RECEIVE_BOOT_COMPLETED
3	GET_TASKS
4	Ljava/net/URL; → openConnection
5	VIBRATE
6	WAKE_LOCK
7	KILL_BACKGROUND_PROCESSES
8	SYSTEM_ALERT_WINDOW
9	ACCESS_WIFI_STATE
10	DISABLE_KEYGUARD
11	Landroid/location/LocationManager; → getLastKnownLocation
12	READ_PHONE_STATE
13	RECEIVE_SMS
14	CHANGE_WIFI_STATE
15	WRITE_EXTERNAL_STORAGE

test of independence to investigate the statistical dependence of features and malware/goodware. The Chi-squared tests for the following hypotheses:

$$H_0 : \text{Features and malware/goodware label are statistically independent.} \quad (1)$$

$$H_1 : \text{Features and malware/goodware label are statistically dependent.} \quad (2)$$

It is known that if there is a sufficiently small p-value (at a given significance level), we reject the H_0 hypothesis and assume, based on the significance level, that feature and malware/goodware label are statistically dependent. Table 2 illustrates the ranking by lowest p-value for the feature. It should be mentioned that the highlighted features in the following tables refer to simalar features compared to table 1.

Table 2: Features ranked by p-value from Chi-Squared test

Feature Rank	Feature	P-Value
1	Ljava/net/URL;→openConnection	0
2	Landroid/location/LocationManager;→getLastKgo...	0
3	Ljava/lang/System;→load	1.23E-205
4	Ljava/lang/System;→loadLibrary	1.14E-194
5	Ldalvik/system/DexClassLoader;→loadClass	9.83E-193
6	Landroid/telephony/TelephonyManager;→getSimOp...	6.04E-146
7	Landroid/telephony/TelephonyManager;→getSimOp...	1.83E-120
8	GET_TASKS	1.86E-112
9	Landroid/telephony/TelephonyManager;→getNetwo...	1.56E-100
10	Landroid/content/pm/PackageManager;→getInstal...	2.06E-99
11	KILL_BACKGROUND_PROCESSES	1.841268e-94
12	RECEIVE_BOOT_COMPLETED	9.028054e-90
13	Landroid/telephony/TelephonyManager;→getNetwo...	9.285886e-82
14	ACCESS_FINE_LOCATION	1.238144e-69
15	ACCESS_COARSE_LOCATION	2.232058e-65

If the compare table 1 and table 2 we can see that Ljava/net/URL;openConnection, GET_TASKS, KILL_BACKGROUND_PROCESSES and RECEIVE_BOOT_COMPLETED are similar features in the Chi-squared feature selection and feature selection from the paper.

We also used the feature importance function of random forest to calculate relevant features. For the random forest framework, we first determine the average contribution of each feature to the decision made by each tree and then got the mean value of importance scores of all the trees. After that, we calculated the average decrease in impurity for each feature based on its usage in the tree nodes and assigned higher importance to features that had a greater average decrease in impurity. The importance of the features is ranked by its weights which are calculated with the procedure mentioned before. Table 3 highlights the results for the Random forest feature importance:

Table 3: Features ranked by Random forest feature importance

Feature Rank	Feature	Weight	Weight_sum
1	RECEIVE_BOOT_COMPLETED	0.162753	0.162753
2	Ljava/net/URL;→openConnection	0.091256	0.254009
3	GET_TASKS	0.083866	0.337874
4	KILL_BACKGROUND_PROCESSES	0.076948	0.392615
5	Landroid/location/LocationManager;→getLastKgo...	0.052287	0.444902
6	WAKE_LOCK	0.047169	0.492071
7	SYSTEM_ALERT_WINDOW	0.033288	0.525359
8	READ_PHONE_STATE	0.031640	0.556999
9	Ljava/lang/System;→load	0.028362	0.585362
10	RECEIVE_SMS	0.026135	0.611497
11	VIBRATE	0.022299	0.633796
12	DISABLE_KEYGUARD	0.020240	0.654036
13	ACCESS_WIFI_STATE	0.016377	0.670414
14	Landroid/telephony/TelephonyManager;→getSimOp...	0.016005	0.686418
15	Landroid/telephony/TelephonyManager;→getSimOp...	0.015851	0.702269

In comparison of table 1 and table 3 we can see 11 features out of 15 are similar. As a consequence, the performance to replicate the features from Random forest compared to Chi-Squared is better with the Random Forest.

For Xg Boost, feature importance is based on the amount that each feature contributes to reducing the loss function. It can be decided in three ways: weight, gain, and coverage. Weight refers to the amount of each feature that contributes to reducing the loss function and gain indicates the improvement in the loss function by using a specific feature to split. Moreover, coverage defines the number of samples affected by the splits because of this feature. For all these parameters, more important features are obtaining higher values. We can see the ranked features in table 4: Xg boost identifies 9

Table 4: Features ranked by Xg boost feature importance

Feature Rank	Feature	Weight	Weight_sum
1	RECEIVE_BOOT_COMPLETED	0.470284	0.470284
2	Ljava/net/URL;→openConnection	0.163478	0.633762
3	Landroid/location/LocationManager;→getLastKgo...	0.078593	0.712355
4	SEND_SMS	0.033638	0.745993
5	KILL_BACKGROUND_PROCESSES	0.028278	0.774270
6	GET_TASKS	0.023492	0.797762
7	READ_SMS	0.022215	0.819977
8	RECEIVE_SMS	0.009771	0.829748
9	ADD_VOICEMAIL	0.009568	0.839316
10	BATTERY_STATS	0.009387	0.848703
11	VIBRATE	0.008857	0.857559
12	Ljava/lang/System;→loadLibrary	0.007574	0.865134
13	WAKE_LOCK	0.007271	0.872405
14	CHANGE_WIFI_STATE	0.007186	0.879591
15	Landroid/telephony/TelephonyManager;→getNetwo..	0.006681	0.886272

similar features and performs still better than Chi-squared in replication but worse than Random forest.

The meaning of the highlighted features is explained as follows:
VIBRATE: A type of malicious activity or behavior that involves causing a device, such as a smartphone or computer, to vibrate.
WAKE_LOCK: A type of permission or function that allows an app or process to keep the device awake or prevent it from going into a dormant or "sleep" state.
CHANGE_WIFI_STATE: A type of permission or function that allows an app or process to change the state of the device's WiFi connection.
RECEIVE_SMS: A type of permission or function that allows an app or process to receive SMS (short message service) messages
GET_TASKS: A type of permission or function that allows an app or process to retrieve a list of tasks that are currently running on the device.
KILL_BACKGROUND_PROCESSES: A type of permission or function that allows an app or process to terminate other tasks or processes that are running on the device.
SEND SMS: A type of function or capability that allows an app or process to send SMS (short message service) messages.
Ljava/net/URL;→openConnection: This method may be used by malware to establish a connection to a remote server in order to download additional payloads or to exfiltrate data.
RECEIVE_BOOT_COMPLETED: A broadcast action in the Android operating system that is sent after the system has finished booting. It is used to allow apps to perform tasks that they need to do after the device has been booted, such as setting up alarms or syncing data.
ACCESS_WIFI_STATE: ACCESS_WIFI_STATE is a permission in the Android operating system that allows an app to access information about the device's WiFi state, such as the name and MAC address of the WiFi network the device is connected to.
DISABLE_KEYGUARD: A permission in the Android operating system that allows an app to disable the keyguard, which is the lock screen of the device. This permission allows an app to prevent the user from being able to unlock the device, even if the device is protected by a PIN, password, or pattern.
READ_PHONE_STATE: A permission in the Android operating system that allows an app to access information about the device's phone state, such as the phone number, IMEI, and SIM serial number.

SYSTEM_ALERT_WINDOW: A permission in the Android operating system that allows an app to display overlay windows on top of other apps.

3 Reproduce the results from the paper

In this chapter we will first discuss the individual methods that were used in the context of the paper and our implementation. Afterwards we will compare the results of our implementation and the results of the paper. We will discuss under which conditions we achieved the results (hyper-parameters, features, etc.). After that we will have a deeper analysis of the methods, e.g. confusion matrices for our methods results, the relation between CPU time and classification error and training set size and cross validation error vs training error. It should be mentioned that we trained all models (also the new ones) on the whole data including all provided features.

3.1 Test results

Now, we would like to discuss which methods have been used and give a short introduction to these methods:
Random forest: A random forest trains a certain number of decision trees on a dataset and combines the results to get a final prediction. Each decision tree is trained on a randomly generated subset of the dataset, which should reduce overfitting (and thus overall accuracy on unknown datasets). Finally, the average of all predictions of the individual decision trees is taken as the final prediction.
Extra tree: The Extra tree from sklearn follows the same logic as the Random forest but the subset of features used for each decision tree would be even more random, which could potentially further improve the performance of the model.
Ada boost: In the Ada boost framework, weak learners are combined to create a strong learner. In the context of Ada boost, weak learners are decision trees which are trained on data in a sequential way and each tree is trained on the errors of the previous tree. The final prediction is a combination of the results of all trees. Here each result is weighted by its accuracy (each weight is proportional to its accuracy so that stronger trees have a higher weight).
Gradient boost: Gradient boosting is a technique to train decision trees sequentially in a stage-wise-manner. In each iteration, a tree is trained to correct the errors from the previous tree. Finally, the prediction is composed as the combination of all the trees before. The gradient of the loss function is computed (with respect to the predictions of the model) to minimize the loss function (negative gradient points in the direction of the steepest descent in its loss function).
Xg boost: Xg boost is a popular, efficiently written and scalable implementation of the gradient boosting algorithm.
Before we introduce the parameter selection for our methods, it should be mentioned that all parameters that are not specified in the function, are chosen as default values. We used Hyper-parameter tuning for out methods and these are our final model parameters:

Random forest: `RandomForestClassifier(bootstrap=True, max-depth= 90, min-samples-leaf= 3, min-samples-split= 12, n-estimators= 100).`
`bootstrap:` This specifies whether or not to use bootstrap sampling when training the decision trees. Through the bootstrap, we want to avoid overfitting.
`max-depth:` The maximum depth is the length of the longest path from the root node to a leaf node for a decision tree. Here we want to prevent complex models and therefore overfitting.
`min-samples-leaf:` The minimum number of samples that must be present at a leaf node in the decision trees. Used to prevent overfitting that trees have less complexity.
`min-samples-split:` The minimum number of samples that must be present at a node in order for the node to be split. We use this parameter to avoid complex models.
`n-estimators:` This parameter refers to the number of decision trees used to create the prediction in the random forest. Here we can improve the number of trees to achieve better performance but increase computational complexity.

Extra tree: `ExtraTreesClassifier(max-depth= 90,min-samples-leaf= 3,min-samples-split= 8,n-estimators= 200).`
The parameters have the same meaning as in the Random forest classifier framework.

Ada boost: `AdaBoostClassifier(learning-rate = 0.1,n-estimators= 1000).`
`learning-rate:` The Learning rate specifies the contribution of each individual weak learner to the final prediction. When we work with smaller learning rate, the weak learners will have less impact on the final result. Here the algorithm will be 'less aggressive' in updating the predictions. For a larger learning rate, weak learners will have a stronger influence on the final result.
`n-estimators:` Same as before.

Xg boost: `XGBClassifier(learning-rate = 0.1,n-estimators = 200).`
`learning-rate:` The learning rate specifies the step size at which the model updates its predictions based on the gradient of the loss function. When we pick a small learning rate, we shrink the gradient and make smaller steps to minimize the loss function. Here we may avoid overfitting but the computational complexity may be higher (slower convergence). For a larger learning rate, we may have faster convergence but have the risk of overfitting.
`n-estimators:` Same as before.

Gradient boost: `GradientBoostingClassifier(learning-rate = 0.1,n-estimators = 1000, max-depth = 7).`
The meanings of all parameters was discussed before. The `learning-rate` definition refers to the definition made in the context of Xg boost.

In the following we will see what results the methods described above have achieved and how they can be considered with the results from the paper. It should be mentioned that deviations of our results from the results of the paper are due to the fact that it is not known which features, which over-sampling/under-sampling method, which hyper-parameters and which metrics were used in the paper to evaluate the model performance.

Method	Best cv accuracy	Mean cv accuracy	Paper result
Random forest	0.9925	0.9746	0.987
Extra tree	1.	0.9710	0.988
Ada boost	1.	0.9666	0.979
Xg boost	1.	0.9753	0.978
Gradient boost	1.	0.9641	0.974

In the column 'best cv accuracy', we have chosen the highest accuracy that resulted from the 10-fold cross validation. The 'mean cv accuracy' was calculated from all results of the 10-fold cross validation. For the results from the paper, we only know that they were calculated on the basis of a 10-fold cross validation, but not whether they are the best value or the average.
From the results we can see that the results from the paper are higher than the average cross validation results but not better than the best cross validation results. It can be assumed that the average cv scores in the paper have been used. Basically, it can be emphasized that both, the Random forest and the Extra tree, among others, were the models with the best performance.
One way that Random forest and Extra tree methods perform better than Ada boost and Gradient boost (in terms of mean cv score) is that they are less susceptible to overfitting, since the decision trees are trained and assembled independently, and in Ada boost and gradient boost, the predictions are built sequentially. The following graph illustrates this:

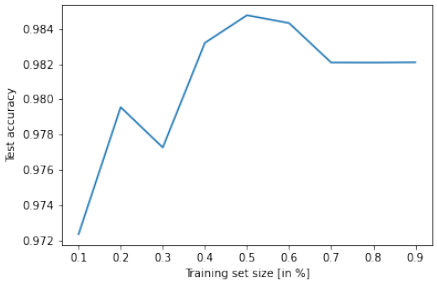

Figure 3: Extra tree

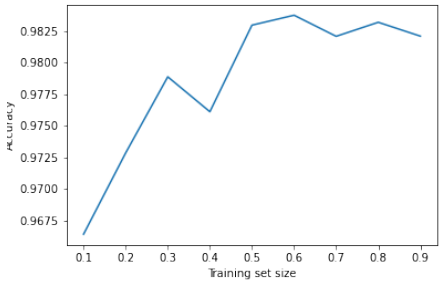

Figure 4: Random forest

We see in figure 3 and figure 4 a coordinate system consisting of training set size in percent (of the total data set) and the test accuracy. Here we see that even if the training set is increased, the test accuracy can maintain a constant level, which speaks against an overfit based on the data.

In comparison, we see in figure 5 and figure 6 that both Ada boost and Gradient boost cannot

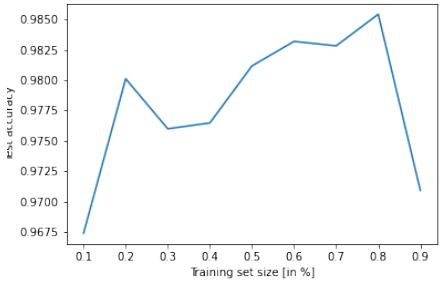

Figure 5: Ada boost

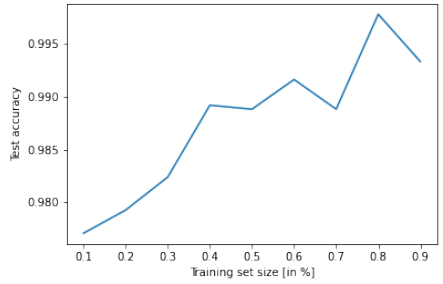

Figure 6: Gradient boost

maintain a constant test accuracy when increasing the training set size. For Ada boost, the overfit is most extreme for a set size of 0.8, where the test accuracy has a very sharp drop.

For comparison, see 7 for the results of the best performing model according to the average cv score, the Xg boost. Here it becomes clear that a constant level of test accuracy cannot be maintained until a training set size of 0.8. After that Xg boost can avoid overfit for very large training data. The reason for outperforming sklearns Gradient boost could be that Xg boost utilizes more efficient algorithms and data structures than Gradient boost (for example more efficient and better performing optimization algorithms for the loss functions such as BFGS instead of Gradient descent).

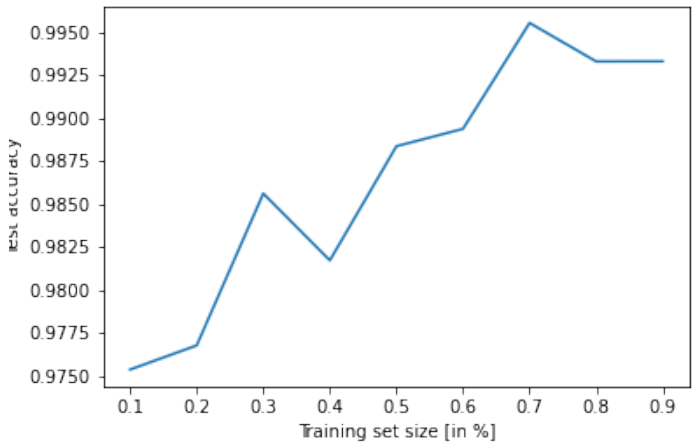

Figure 7: Xg boost

3.2 Confusion matrix for test results

In this chapter we will present and compare the confusion matrix of our models. We use the following format for a confusion matrix:

	Predicted	
	Negative	Positive
Actual Negative	True Negative	False Positive
Actual Positive	False Negative	True Positive

Table 5: Confusion matrix general

	Predicted	
	Negative	Positive
Actual Negative	88	0
Actual Positive	4	355

Table 6: Confusion matrix Random forest

	Predicted	
	Negative	Positive
Actual Negative	87	1
Actual Positive	4	355

Table 7: Confusion matrix Extra tree

Table 8: Confusion matrices for different models

	Predicted	
	Negative	Positive
Actual Negative	85	3
Actual Positive	4	355

Table 9: Confusion matrix Ada boost

	Predicted	
	Negative	Positive
Actual Negative	88	0
Actual Positive	1	358

Table 10: Confusion matrix Xg boost

Table 11: Confusion matrices for different models

	Predicted	
	Negative	Positive
Actual Negative	88	0
Actual Positive	1	358

Table 12: Confusion matrix Gradient boost

Based on the confusion matrices, we can calculate the following metrics:

$$\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{True Positives} + \text{True Negatives} + \text{False Positives} + \text{False Negatives}}, \tag{3}$$

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}, \tag{4}$$

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}, \tag{5}$$

$$\text{F1 score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}. \tag{6}$$

The following table aggregates the performance metrics introduced for our models:

Method	Accuracy	Precision	Recall	F1-score
Random forest	0.9910	1.0	0.9888	0.9943
Extra tree	0.9888	0.9972	0.9889	0.993
Ada boost	0.9843	0.9916	0.9888	0.99
Xg boost	0.9977	1.0	0.9972	0.9986
Gradient boost	0.9977	1.0	0.9972	0.9986

It should be noted that in the table showing the values for Accuracy, Precision, Recall and F1-score, the same training set and test set were used for all models. In the table where we presented the test results of our trained models, we always reported performance values based on the 10-fold cross validation. In the context of the analysis from the chapter 3 we have already discussed the overall accuracy of the models. It should be noted that Xg boost and Gradient boost are the models that perform best on the dataset (same classification accuracy). This may be due to the fact that both methods basically perform the same operations, but differ in optimization algorithms and data structures (in the analysis more randomness is introduced through the cross validation error which might lead to deviations from the solutions). In addition, the entire data set is not very large, which is why the test data also have rather few observations and therefore lead to similar results.
The table shows that Random forest, Xg boost and Gradient boost have the highest precision. Therefore, if the application requires to identify as much malware (positive responses) as possible (or to minimize the number of false positives), the above models are more suitable than Ada boost and Extra tree. In order to generate a model that reduces the number of false negatives (detected goodware which is malware), we might take a look at the next metric.
In the context of recall, Xg boost and Gradient boost are the best performing models. Generally, we can assume that those models are better in identifying all relevant instances (malware) in the dataset even though we are at risk to include some false positives (detect malware but it is goodware). Here false positives are not crucial for the use case because the costs of detecting false malware are low compared to detecting false goodware. As a consequence, recall may be an important metric to look at when we want to identify most malware cases correctly.
For the F1-score the performance of all methods is pretty close to each other (and pretty accurate). From (6) we can see that the number of false positive and false negative instances is included. Here a balance between both metrics is useful but for the detection of malware/goodware, the recall may be an important performance metrics as highlighted before.

3.3 CPU time and classification error

Comparing classification error and CPU time can be useful to compare our methods in terms of trade-off between classification error (on test data) and efficiency. The classification test error is a measurement to evaluate the models performance to correctly predict the class of an instance on data which was previously unknown while the CPU time measures how fast the model can make a prediction. The test error is calculated as the mean of 10-fold cross validation error.
First, we want to highlight the models that reach an acceptable error level in a comparatively short time (compared to the other methods): Random forest and Extra tree.

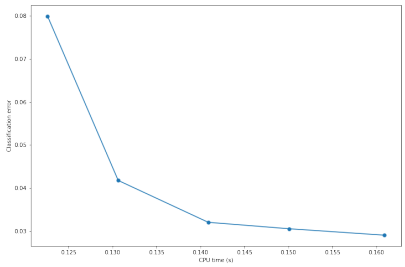

Figure 8: Random forest

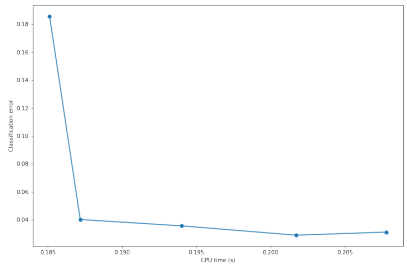

Figure 9: Extra tree

Using the figures 8 and 9, we can see that the Random forest has the largest drop in text error after 0.13 seconds of CPU time and the Extra tree has the largest drop in text error after 0.185 seconds of CPU time and then reaches a constant level of a test error. In this context, we want to compare the results with Xg boost, Gradient boost and Ada boost:

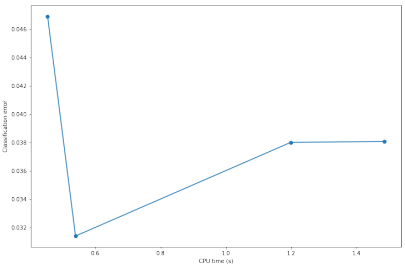

Figure 10: Gradient boost

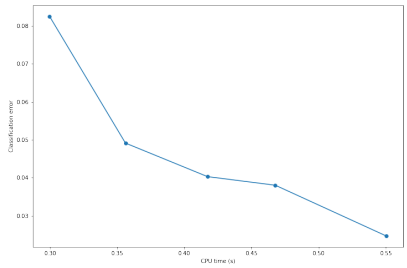

Figure 11: Xg boost

In figure 10 and figure 11 we observe that Gradient boost reaches its lowest level for the test error around 0.5 seconds CPU time and Xg boost around 0.55 seconds.

Here we can see that Gradient boost test error increases again after 1.2 seconds CPU time. One reason for this jump in the test error could be the learning rate, which may have been chosen too high with 0.1. First order methods such as stochastic gradient methods have the problem that if the learning rate is too high, a large descent is initially generated, but subsequently ends up in a suboptimal solution (e.g. a saddle point or local minimum with a high objective function value), which makes a significant descent in the loss function more difficult.
Furthermore, it is interesting to observe that Xg boost cannot achieve a significant drop in test error between 0.35 and 0.5 seconds, but can generate another drop afterwards. One reason for this could be the more efficient data structures and optimization algorithms that Xg boost uses. Here, by a better chosen optimization method (for example BFGS for the approximation of second order curvature information), the case could occur that iterations are generated out of a suboptimal solution.
Comparing the boosting approaches with random forest and extra tree, the latter provide a significant

decrease in test error much faster, but do not reach the level of test errors of the boosting methods. One reason for this could be the general complexity of the models; in a random forest, individual decision trees are trained and then the results of the individual trees are aggregated, while the gradient boost builds up its solution sequentially and then tunes it using an optimization procedure.
Finally, let's look at the results of the Adaboost and compare them with the previous test results:
Using figure 12, we see that Ada boost takes the highest CPU time to reach its lowest test error

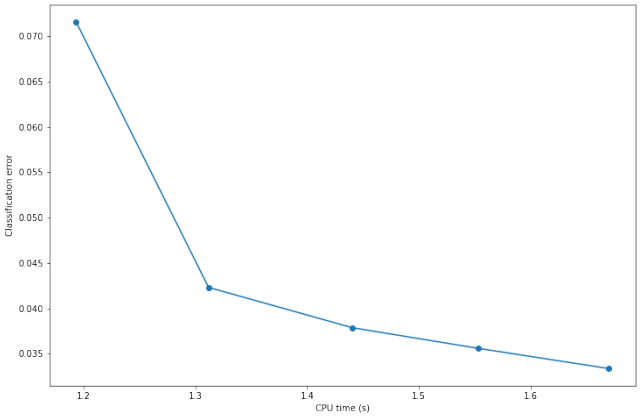

Figure 12: Ada boost

level (more than 1.6 seconds). Nevertheless, the solution can be significantly improved with increased CPU effort compared to all other methods. One reason for this could be that Ada boost, as already mentioned, generates learners that learn sequentially from the errors of previous learners and thus sequentially turns weaker learners into 'stronger learners'.

3.4 Training set size and cross validation error vs training error

In this section, we will look at how the training accuracy performs against the cross validation accuracy when we increase the training set size. For this experiment we have looked at 2 cross validation runs and the color pairs for the training run and cross validation run are always (blue, green and orange, red).
Regarding the ability of the models to generalize well, it is useful to look at the gap between training accuracy and cross validation accuracy:

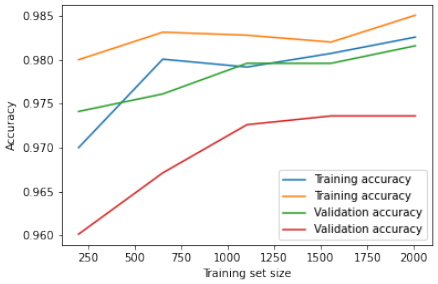

Figure 13: Random forest

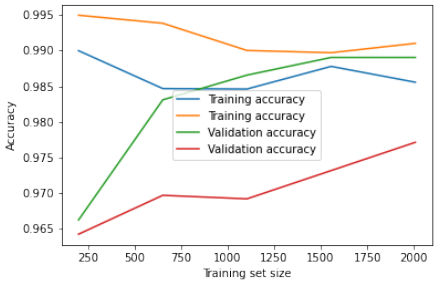

Figure 14: Extra tree

Using the figure 13 we can observe that with increasing training set size for the red and orange pair, the training and cv accuracy seems to converge (initially distance is still relatively large and becomes smaller with increasing training set size). For the blue and green pair an interesting observation arises that in some sections the cv accuracy is above the trainings accuracy and in other sections the trainings accuracy is above the cv accuracy (the same observation can be made in figure 14). One reason for this could be that Random forest (and Extra tree) randomly select the features for training the decision trees. The cross validation adds another random factor, so that the training set, despite its growing size, does not provide enough relevant information (features) to outperform the cross validation set (where the cross validation set may contain more relevant features). In the Extra Tree plot, the red-orange pair also shows that the distance between trainings and cv accuracy decreases with increasing training set size. In figure 13 slight tendencies of an overfit can be seen, since from a training set size of approximately 1000 the red cv accuracy does not increase anymore, but in both plots we have an increasing tendency of the cross validation accuracy above the training set size. In figure 14 it is interesting to see that the orange training accuracy decreases with increasing training set size. This may be due to the fact that the Extra tree has chosen inappropriate features for the labels (statistically independent).
In the following we will analyze the boosting methods:
In figure 15 and figure 16 we can basically make the observation that the distance between trainings

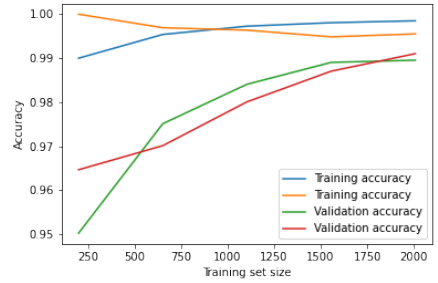

Figure 15: Gradient boost

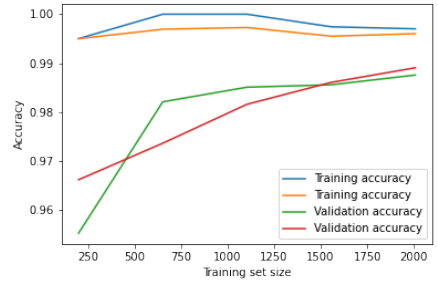

Figure 16: Xg boost

and cv accuracy is already smaller than in figure 13 and 14 for a small trainings set size and both accuracies seem to converge to each other with a smaller distance than Random forest or Extra tree. Moreover, we do not have the phenomenon that in certain areas, cv accuracy is higher than trainings accuracy. This may be due to the fact that in both gradient boosting methods, the extent of randomness is not present as in Random forest or Extra tree. Nevertheless, for both methods we can observe that the training accuracy does not increase significantly when increasing the training set size (orange for Gradient boost and blue for Xg boost). One reason could be that the capacity of the model with its used features and parameters is reached and noise in the dataset leads to small deviations. The training accuracy is consistently in a range of 0.99 to 1.0.

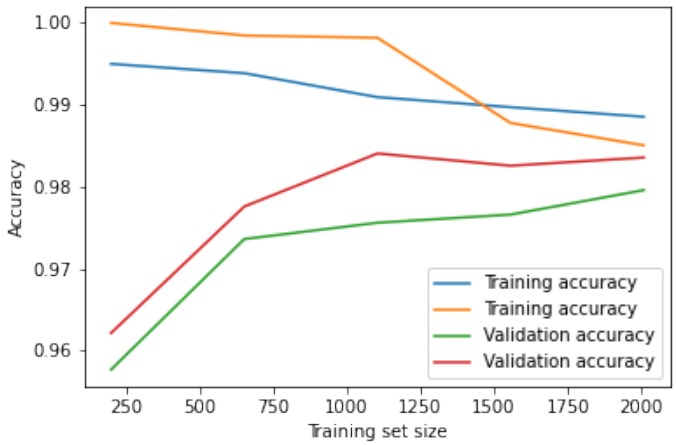

Figure 17: Ada boost

Finally, we want to discuss the results of the Ada boost. In figure 17 we can observe that the distance between trainings accuracy and cv accuracy is still very large with a small trainings set size and becomes smaller with increasing set size. Here, however, we can observe that the trainings accuracy decreases and the cv accuracy increases. From a training set size of 1000 it can be seen that the cv accuracy also stagnates. This can have several reasons. Basically, it can be assumed that the parameters for the Ada boost have not been chosen optimally (or other parameters should have been chosen differently) or that the model is simply not optimally suited to achieve a certain accuracy on this data set.

4 New models

In the following we want to present two methods, which we have generated in addition to the already implemented methods from the paper: Feedforward neural network and Support Vector Machine. We have chosen these methods because Feedforward neural network can learn complex and nonlinear relationships between features and labels on large datasets. Support vector machines have the advantage that they can effectively separate classes via a nonlinear or linear decision boundary and can also efficiently solve high-dimensional data via the kernel trick. However, SVMs have the disadvantage that the user must choose the kernel function, while the kernel is generated 'automatically' in a feedforward neural network.

4.1 Test results

Now let's give a small introduction to the methods, which hyperparameters and architecture (for the neural network) we have chosen.
Feedforward neural network (NN): In a NN, the features pass through the input layer which consists of input nodes (represent features) and runs through more layers afterwards, the hidden layers. The amount of information which flows between neurons in the layers are algorithmically determined by the weights. The format of the information which is passed from one neuron to another is determined by the activation function. In the hidden layers, the neural network learns complex relationships between the input data and the label. The last layer is the output layer which produces the final prediction of the input data.
Support Vector Machine (SVM): SVM are methods that try to find a hyperplane in a high-dimensional feature space that maximally separates classes. The algorithm finds the hyperplane that maximally separates the classes by maximizing the margin between the two classes. In the case of a nonlinear SVM, the algorithm finds the hyperplane by mapping the data into a higher-dimensional space using a kernel function and then finding the hyperplane in this space.

Feedforward neural network: `MLPClassifier(hidden-layer-sizes=(20, 20, 15, 20), solver='sgd', learning-rate-init=0.1, learning-rate = 'adaptive', batch-size=64, activation='relu').`

`hidden-layer-sizes:` The hidden-layer-sizes are a parameter that determine the basic architecture of the NN. We have identified by hyperparameter tuning (20, 20, 15, 20), which means that the first hidden layer has 20 hidden units (neurons), the second hidden layer has 20 neurons etc.

`Solver, batch-size, learning-rate-init and learning-rate:` We are using the batch stochastic gradient descent (SGD) to solve the following optimization problem:

$$\min_{w \in R^N} L = -\frac{1}{B} \sum_{i=1}^{B} [y_i \log f(x_i; w) + (1 - y_i) \log(1 - f(x_i; w))] \tag{7}$$

where w are the weights between neurons, N is the number of weights in the hidden layers, $f(x_i; w)$ is the predicted probability of the positive class for sample x_i, y_i as the true value of the label and B is the Batch size.
In a batch sgd, the training dataset is divided into smaller batches and the gradient of the loss function from (7) is generated on this smaller batch and updates the model parameters via a batch gradient step. It should be mentioned here that this is a heuristic and convergence of the method against a local or global minimum point of (7) is by no means guaranteed. Nevertheless, from a practical point of view, the problem has proven to be useful, especially for high dimensional problems. The `batch-size` parameter therefore defines the size of the batches in (7). The `learning-rate` and `learning-rate-init` has an identical meaning to the learning rate and from chapter 3.1; it is the step size of the SGD. The parameter learning-rate-init determines the starting step size which is passed to the step size algorithm and learning rate determines whether I use a step size algorithm (for example Adam) or constant step sizes (then learning-rate-init is used).

`Activation:` The activation function is applied to the output of a neuron. The purpose of the activation function to introduce non-linearity into the model in order to learn more complex relation-

ships between features and labels (for example, a neural network with one hidden layer and a linear activation function in the hidden layer, and a logistic activation function in the output layer can be viewed as a variant of logistic regression). Here we use the rectified linear unit (ReLU) activation function:

$$f(x) = \max(0, x). \tag{8}$$

SVM: `SVC(kernel='rbf', C=100.0, gamma = 0.1, probability=True).`

`Kernel:` The kernel function, $k(x, y)$, maps input data into a higher-dimensional space, where we hope that data will become more linearly separable. The choice of the kernel function can have a significant impact on the performance of the SVM, as it determines the decision boundary that is learned by the model:

$$k(x, y) = \exp(-\gamma \|x - y\|^2). \tag{9}$$

The `gamma` parameter refer to (9) and was determined by hyperparameter tuning.
The parameter `C` refers to the convex optimization problem which is solved in SVM:

$$\min_{w,b,\{x_i\}} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n} x_i \text{ s.t. } y_i(w^T k(x_i, x) + b) \geq 1 - x_i \quad \forall x, \tag{10}$$

with w is the weight vector, b is the bias term, x_i is the slack variable for the i-th sample, y_i is the label for the i-th sample, `C` is the hyperparameter that controls the trade-off between the complexity of the decision boundary and the level of allowed misclassification, and $k(x_i, x)$ is the RBF kernel function from (9) applied to the i-th sample and an arbitrary sample x. The parameter `C` scales the relevance of slack variables, which are used to allow for misclassification in the optimization problem that is solved by the SVM.

The test results were collected under the same conditions as from chapter 3.1:

Method	Best cv accuracy	Mean cv accuracy
Neural network	1.0	0.9731
SVM	1.0	0.9753

Comparing the cv mean accuracy of NN and SVM with the results from the result table in chapter 3.1, we see that both models are among the best performing models, but only SVM comes close to the result of Xg boost. One possibility that the results of the neural network do not match the results of the Xg boost could be an overfit. The following graphs, which compare training set size in percent and test accuracy, can provide information.

The results from figure 18 and 19 show that both methods cannot achieve a stable level of test

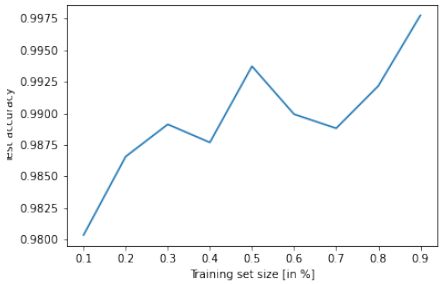

Figure 18: Neural network

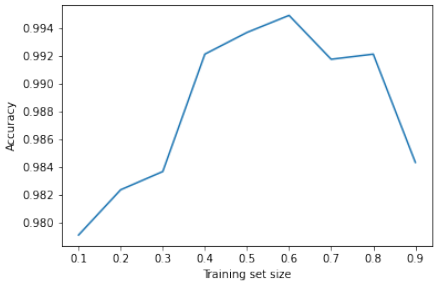

Figure 19: SVM

accuracy compared to the Random forest and Extra tree. Nevertheless, the neural network manages to increase its test accuracy with increasing training set size. Both methods have a drop at a training set size of 0.7. In figure 19 it can be seen that the SVM seems to be strongly affected by an overfit, because from a training set size of 0.7 the SVM does not manage to generalize anymore and the test accuracy drops continuously.

4.2 Accuracy, Precision, Recall and F1-score

In this chapter we will look at what Accuracy, Precision, Recall and F1-score look like for the neural network and SVM.

Method	Accuracy	Precision	Recall	F1-score
Neural network	0.9955	1.0	0.9945	0.9972
SVM	0.9932	1.0	0.9917	0.9958

When we directly compare the neural network with the SVM, the neural network is superior in terms of accuracy, recall, and F1-score. Here we can make the basic assumption that the neural network is better suited for the application of identifying malware than the SVM. If we compare the results with Xg boost and Gradient boost, both the neural network and the SVM are at a disadvantage. One reason why boosting techniques might perform better in this context than NN is that gradient boosting techniques can be trained more effectively on smaller data sets and/or neural networks are more sensitive to hyperparameters in their performance.

4.3 CPU time and classification error for NN and SVM

In the following we will compare the CPU time against the classification error for the neural network and the SVM, analogous to the procedure from chapter 3.3.

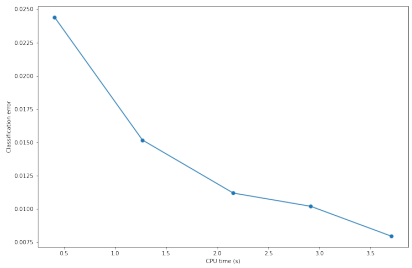

Figure 20: Neural network

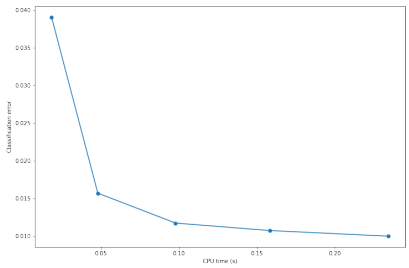

Figure 21: SVM

In figure 20 it can be seen that the Neural Network achieves the smallest classification error of all treated models so far, but the cpu time needed to reach this error level is by far the highest. On the one hand, this may be related to the structure of the neural network, since the neural network passes forward and backward through the network, adapting weights and biases of the network based on the input data and the corresponding labels. This procedure is repeated why the training phases could be very time consuming. Another reason could be the optimization algorithm we use. As indicated before, the SGD is a heuristic that performs iterations quickly, but is just not guaranteed to converge to a local or global minimum point. With an insufficient descent in the loss function, the process described above continues. In this context, one can, for example, use a limited memory BFGS procedure to approximate second order curvature information using batch gradients to perform more efficient iterations where the iterations take longer. The advantage that iterations are executed more efficiently, for example via a BFGS procedure, has the disadvantage that iterations have to be generated more expensively.
The SVM in figure 21 cannot reach the level of error generated by the neural network, but it can achieve the largest drop in classification error of all methods for the time frame. One reason for this could be that the underlying optimization problem (10) is convex and a critical point is a global minimum point (problem is solvable), which means that only one critical point has to be found. Comparing the situation with a neural network, where the loss function from (7) is not convex and thus can have several saddle points or local minimum points (which do not provide a low objective function value), more computational resources have to be spent to generate a sufficiently small error.

4.4 Cross validation error vs training error for NN and SVM

Last, we will compare the Neural Network and SVM in terms of training and cv accuracy. In figure

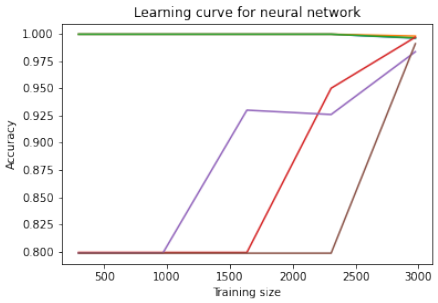

Figure 22: Neural network

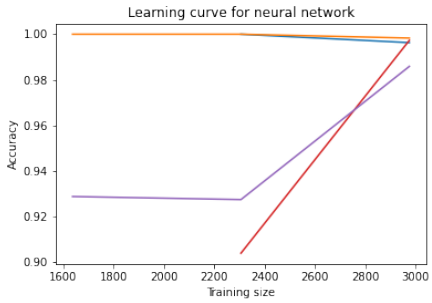

Figure 23: SVM

22 it is interesting to see that up to a certain training set size, the accuracy is at a certain level and then it increases at an accelerated rate. Furthermore, we see that the earlier the increase of the test accuracy starts, the more likely there is a flattening in the course (e.g. purple cv accuracy flattens again, while brown cv accuracy increases continuously when reaching a certain training set size, until cv accuracy converges to training set accuracy). Moreover, there is no model for which the gap between cv and trainings accuracy is as narrow as for the neural network. In addition, the training accuracy is continuously at a very high level (at very high training set sizes, small overfit tendencies can also be seen).
For the SVM in figure 23 similar observations result, but the SVM does not manage to close the gap between the cv accuracy and the training accuracy as the neural network does.
Overall, it can be said that both SVM and neural network can minimize the distance from cv accuracy to training accuracy compared to the other methods and it can be suggested that both methods are very suitable for making generalized predictions on the malware dataset.

5 Conclusion and future outlook

In this paper, we have analysed the TUANDROMD dataset by using different techniques to identify relevant features and looking at basic statistics around the dataset. We then analysed and implemented tree methods from the referenced paper. Here, both Random Forest and Extra tree turned out to be the best methods in terms of low overfit and low cpu time, but when it came to overall performance, correctly identifying as much malware as possible and avoiding the 'false goodware' error, the gradient boosting techniques were better suited. Subsequently, we proposed neural networks and SVM as new methods. Here we saw that both methods could compete with the boosting methods in terms of overall performance, but were also affected by overfitting. The neural network even achieved the lowest classification error on the test data set in relation to the CPU time, but also had to use by far the most resources to achieve it.

Basically, all implemented methods have delivered acceptable performance. In our opinion, the greatest potential lies in neural networks. Here, we have by far not achieved what is possible with our presented architecture (more hidden layers and/or hidden units). Furthermore, additional hyperparameters outside the architecture (learning_rate, etc.) could be tuned more accurately. Another possibility is the use of other techniques:
Recurrrent neural network (CNN): CNNs originally have their application in the classification of images, but can also be applied to classification applications such as our malware use case.
Transfer learning: A deep learning model is generated which has previously been trained on a very large data set and the model is adapted to the malware detection use-case through targeted tuning (for example, certain model parameters might react more sensitively to the malware/goodware data set).
Ensemble Deep Learning: Here we can train different deep learning models with different architectures on different malware/goodware datasets and ultimately aggregate their predictions into a final

prediction (for example, take the average).
Finally, it should be mentioned that many of the proposed methods also require more data (observations and features), as the data set is relatively small with approximately 4500 observations. Moreover, additional and more effective feature selection methods than the Chi-squared test could be used for binary features only.

6 References

[1] P. Borah, D. Bhattacharyya and J. Kalita, "Malware Dataset Generation and Evaluation," 2020 IEEE 4th Conference on Information Communication Technology (CICT), 2020, pp. 1-6,doi: 10.1109/CICT51604.2020.9312053.